APIContext2Com：结合预定义 API 文档生成代码注释*

Ramin Shahbazi不列颠哥伦比亚大学计算机科学系加拿大奥肯那根，
ramin20@mail.ubc.ca

Fatemeh Fard不列颠哥伦比亚大学计算机科学系加拿大奥肯那根，
fatemeh.fard@ubc.ca

arXiv:2303.01645v1 [cs.SE] 3 Mar 2023

摘要—代码注释对于理解软件程序非常有帮助，也能帮助开发者在软件维护中节省大量时间。代码注释生成旨在根据代码片段自动预测自然语言注释。已有多项研究探讨了整合外部知识对生成注释质量的影响。在本研究中，我们提出了一种名为 APIContext2Com 的解决方案，通过整合预定义的应用程序编程接口（API）上下文来提高生成注释的效果。API 上下文包括代码片段中使用的预定义 API 的定义和描述。由于详细的 API 信息能够表达代码片段的功能，因此有助于更好地生成代码摘要。我们介绍了一种序列到序列的编码器—解码器神经网络模型，该模型采用多种不同组合的多个编码器，以有效地将不同输入转换为目标注释。我们还设计了一种排序机制，用于排除信息量不足的 API，从而筛除无关 API。我们使用来自 CodeSearchNet 的 Java数据集评估了我们的方法。结果表明，所提模型相较于最佳基线，在 BLEU1、BLEU2、BLEU3、BLEU4、METEOR 和 ROUGE-L 指标上分别提升了 1.88（8.24%）、2.16（17.58%）、1.38（18.3%）、0.73（14.17%）、1.58（14.98%）和 1.9（6.92%）。人工评估和消融研究证实了生成注释的质量，以及架构和 API 排序的作用。

I. 引言

代码注释生成旨在生成可读的摘要，用以描述源代码的功能。注释对于以尽可能少的时间和精力理解源代码至关重要 [1, 2]。软件开发人员需要花费大量时间理解代码，尤其是在维护期间。若有适当的文档和摘要可供查阅，就能节省大量时间 [3, 4]。另一方面，手动编写适当的注释既耗时，有时也十分繁琐。此外，随着软件项目的发展，注释也需要更新；否则，随着时间推移，注释可能无法准确描述代码片段的实际功能，从而产生误导。在许多情况下，随着软件规模的扩大，注释会被遗漏或过时 [5, 6]。因此，采用自动化技术为代码片段生成注释至关重要。

本研究得到加拿大自然科学与工程研究委员会的资助，资助编号为 RGPIN-2019-05175。

多项研究采用了基于规则的方法，例如模板生成 [3] 和信息检索（IR）[7, 8, 9, 10, 11] 技术来生成代码注释。前一种方法提供了一些预定义模板，帮助开发者编写可读注释。尽管如此，这些模板针对特定项目，开发者需要花时间理解并调整这些模板。此外，为不同项目制作模板需要丰富的领域知识。信息检索方法通过从源代码中选取词元，或重复使用相似源代码的注释来生成注释。然而，信息检索方法有两个局限。首先，当命名不佳（例如方法名称与其功能无关）时，它无法提供有意义的注释。其次，代码仓库中可能不存在相似的代码片段。

研究人员已将神经机器翻译（NMT）用于代码注释生成，并取得了令人鼓舞的成果 [12, 13, 14, 15, 16, 17]。神经网络模型通常由编码器-解码器框架构成，并遵循某种变体的 RNN 架构。Iyer 等人 [12] 提出了一种序列到序列长短期记忆（LSTM）模型，其中编码器接收源代码作为输入，解码器则利用注意力机制，根据编码器的输出生成注释。许多研究 [14, 18, 19] 利用抽象语法树（AST），以更好地融入源代码的句法信息。多项研究的结果证实，神经网络框架在代码注释生成任务上的优势。然而，通过融入一些隐藏的外部信息，代码注释生成仍有改进空间。最近，有多项研究探讨了将外部信息整合到模型中对提升模型性能的影响。检索相似的源代码及其对应注释，并将这些注释作为外部输入来提升模型性能，就是此类方法的例子[20, 21]。

应用程序编程接口（API）也被以往的研究用作外部输入，以改进生成的注释 [22, 23]。源代码功能可以定义为代码中使用的一系列预定义API [22]。开发人员通常会在代码中使用一系列相关API来完成特定任务 [22]。尽管源代码可能采用不同的风格约定，但对于

某个特定功能，API 序列通常相似。例如，要编写一段代码并将内容写入 JSON 文件，常见做法是依次调用以下 API：FileWriter.write 和 FileWriter.flush。每个预定义API都有完整的定义和说明，可在 JDK 参考文档中找到。在此示例中，上述 API 的说明分别为[1]：“写入字符串的一部分”和“刷新流”。我们可以看到，这些说明定义了主方法的功能。因此，我们认为，提取 API 文档并将其融入代码注释生成任务，有助于提升模型的性能。

Hu 等人 [22] 利用 API 序列摘要任务，将知识从 API 名称序列迁移到代码摘要任务。Shahbazi 等人 [23] 将 API 描述纳入其中，以改进生成的代码注释。然而，他们未能充分提取 API 文档中的知识。他们得出结论：方法中 API 的数量越多，模型预测注释的性能就越差。他们假设，API 数量增加会给模型带来噪声，误导模型生成更好的注释。我们认为，这是因为他们只是预先拼接 API 描述文本，并将长篇序列文本输入模型。

在本文中，我们提出了 **APIContext2Com**，并借助新思路解决先前工作中的问题 [23]。首先，除了 API 描述之外，我们还纳入了 API 定义，其中包含 *API* 参数的类型和名称。API 定义通过捕捉 API 的一些隐藏信息来丰富模型，而这些信息有助于生成注释。其次，我们提出了一种新颖的架构，使模型能够通过不同的编码器分别处理每个 API 描述和定义。第三，我们设计了一种排序机制，用于识别信息丰富的 API，从而移除对生成代码注释无用的 API。提出这种排序机制的原因是，有些 API（例如 toString()）通常较为通用或与方法无关，无法为模型提供任何信息。另一方面，有些 API 则具体得多，能够提供关于给定代码片段主要功能的详细知识，因此有助于生成注释。当一个方法包含三个以上的 API 时，这种排序机制尤其有用。

APIContext2Com 针对每个方法使用四种不同的输入：源代码、抽象语法树、一组 API 描述以及一组 API 定义。其架构包含两个单独的编码器和两组编码器：前两个编码器接收源代码和抽象语法树，两组编码器则分别接收 API 描述和 API 定义。我们在一个大规模 Java 语料库上开展研究，该语料库包含 Husain 等人收集的 137,007 个样本。 [24]我们还进行了多项实验，以评估使用排序机制移除无关 API，以及将 API 文档添加到模型中的效果。我们的研究发现，移除排名较低的 API 能够提升生成代码的质量。APIContext2Com 的表现优于最佳基线模型，BLEU1、BLEU2、BLEU3、BLEU4、METEOR 和 ROUGE-L 分别提高了 1.88（8.24%）、2.16（17.58%）、1.38（18.3%）、0.73（14.17%）、1.58（14.98%）和 1.9（6.92%）。

我们工作的贡献如下：

- 我们纳入了 API 定义，包括 API 参数的名称及其类型。
- 我们设计了一种排序机制，根据 API 与主方法的相似度为其评分，并且仅将相关 API 纳入模型。
- 我们采用了不同的架构，以捕获 API 上下文中的知识。

本文其余部分的安排如下：

下一节中，我们通过两个示例阐述本研究的动机。随后，我们在第三节详细介绍新方法，并分别在第四节和第五节介绍实验及结果。第六节将讨论实验结果的不同方面。接着，我们在第七节说明有效性威胁，并在第八节回顾相关研究。最后，我们在第九节总结本文。

```
public char match(String str1, String str2) {
        invocationCount++;
        char rel = IMappingElement.IDK;

        if (str1 == null || str2 == null) {
            rel = IMappingElement.IDK;
        } else {
            if ((str1.length() > 3) && (str2.length() > 3)) {
                if (str1.startsWith(str2)) {
                    if (str1.contains(" ")) {
                        rel = IMappingElement.LESS_GENERAL;
                    } else {
                        rel = IMappingElement.EQUIVALENCE;
                    }
                } else {
                    if (str2.startsWith(str1)) {
                        if (str2.contains(" ")) {
                            rel = IMappingElement.MORE_GENERAL;
                        } else {
                            rel = IMappingElement.EQUIVALENCE;
                        }
                    }
                }
            }
        }
        if (rel != IMappingElement.IDK) {
            relCount++;
            addCase(str1, str2, rel);
        }
        return rel;
    }
```

Reference:

computes the relation with prefix matcher

APIs Definition:

length()
startsWith(String prefix)
contains(CharSequence s)

图 1. API 定义，即其参数的类型和名称，有助于生成代码注释。“前缀”一词出现在参考摘要和该方法中使用的第二个 API 的定义中，但代码片段中并未提及。

[1]https://docs.oracle.com/javase/7/docs/api/java/io/OutputStreamWriter.html

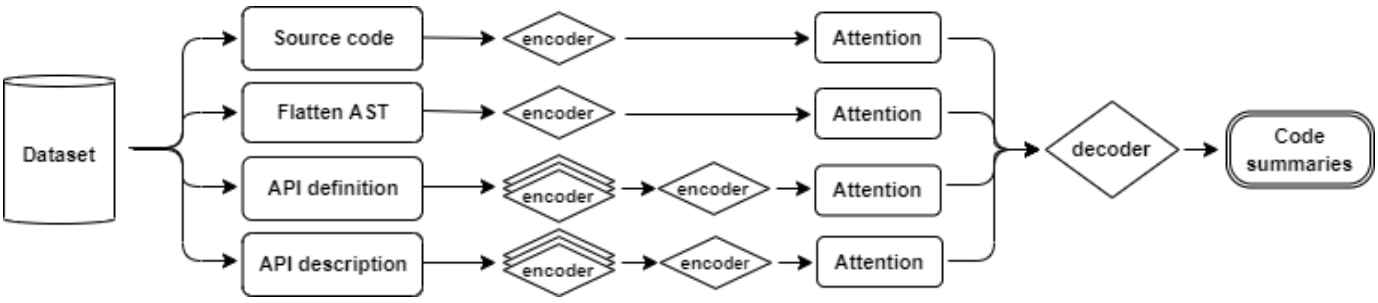

图 2. APIContext2Com 架构概览 e

II. 动机示例

本节展示数据集中的两个示例：第一个说明 API 定义如何帮助模型，第二个则详细介绍排序机制以及用于排除无关 API 的策略。

API 定义由 API 名称以及参数类型和名称组成。在代码片段中，缺少有关 API 参数的信息。代码中只会调用 API，并向其传递所需数据。然而，查阅 JDK 参考文档中的 API，可以获得参数的精确定义；更重要的是，还能获取有意义的名称，这些名称可能蕴含有用的信息。图 1 展示了一个使用了三个内部 API 的 Java 方法，其中给出了源代码、其参考摘要，以及所调用 API 的定义。参考摘要为“计算与前缀匹配器的关系”，其中使用了“前缀”一词。查看代码片段可知，该词并未出现在代码中。有趣的是，该词可以在第二个所用 API 的定义参数中找到：startsWith(string prefix)。这是一个有力的线索，表明整合 API 定义会有所帮助，因此我们假设 API 定义能够提升模型的性能。

图 3 展示了一个 Java 方法，其后依次是参考摘要，以及 API 的定义和描述。快速浏览这些 API 可以发现，第二个 API 的描述与参考摘要高度相似。同样，它们的输入参数也完全相同。这一观察表明，输入参数相似可能意味着功能相似，进而意味着摘要相似。然而，并非方法中的所有 API 都适合纳入模型。许多 API（例如 toString()）与给定代码片段的主要功能无关。在这些情况下，描述和定义都无法为模型提供实用信息。此外，Shahbazi 等人 [23] 得出结论：对于包含三个以上 API 的方法，API 文档并无帮助，反而会误导模型生成摘要。因此，我们设计了一种排序机制，通过为每个 API 评分，计算方法中使用的各个 API 与给定方法之间的相似度。为此，我们比较 API 与方法的输入参数，每出现一处差异，就给该 API 记一个负分。这样，我们便能识别出更有可能为代码摘要提供有用信息的 API。在图 3 的示例中，await(long timeout,TimeUnit unit) 排名最高；而另外两个 API——getCount() 和 dispose()——由于参数存在差异，各被记两个负分，因此根据纳入的 API 数量，它们被用于生成注释的可能性较低。

```
public Throwable blockingGetError(long timeout, TimeUnit unit) {
        if (getCount() != 0) {
            try {
                BlockingHelper.verifyNonBlocking();
                if (!await(timeout, unit)) {
                    dispose();
                    throw ExceptionHelper.wrapOrThrow(
                        new TimeoutException(
                            timeoutMessage(timeout, unit)));
                }
            } catch (InterruptedException ex) {
                dispose();
                throw ExceptionHelper.wrapOrThrow(ex);
            }
        }
        return error;
    }
```

Reference:

block until the latch is counted down and return the error received or when the wait is interrupted or times out null otherwise

APIs:

getCount() : Returns the count of values recorded.

await(long timeout, TimeUnit unit) : Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted, or the specified waiting time elapses.

dispose() : Allows any resources held by this object to be released.

图 3. API 排序的动机示例。第二个 API 的参数与代码片段的参数相似，而另外两个 API 的参数与代码示例的参数并不相似。因此，第二个 API 的定义有助于生成代码摘要，但另外两个 API 的定义则没有这种作用。

III. 所提方法

A. 模型概述

继 [19], 之后，我们在本研究中提出了一种采用多个编码器的序列到序列架构。Shahbazi 等人

[23] 提出了一种包含三个不同编码器的架构，每个编码器分别接收一种输入，包括源代码、抽象语法树和 API 描述。对于 API 描述，他们将所有不同的描述拼接起来，作为单一输入传递给一个编码器。由于每条描述都说明了某个特定 API 的功能，在将它们传递给模型之前进行拼接，可能会生成一段冗长且无意义的词元序列。因此，在本研究中，我们针对 API 描述和 API 定义采用了多个编码器，每条 API 描述或定义都由一个编码器处理，并将所有编码器的最终隐藏状态作为最终编码器的输入。图 2 展示了所提模型的整体架构。

如图 2 所示，代码被输入到一个编码器中。该编码器包含一个嵌入层和一个 GRU 层。我们将编码器应用于抽象语法树（AST）。首先，使用 SrcML 工具生成源代码的 AST 树表示，然后使用 SBT 方法 [13] 将其展平。展平后的 AST 被视为纯文本序列输入编码器。API 描述/API 定义与代码/AST 是不同的输入。每个方法只有一段代码和一个 AST，而每个方法中都存在一组 API。因此，每个 API 都被输入到一个编码器中，然后将各编码器的隐藏状态输入最终编码器。最终编码器的输出与代码和 AST 编码器的输出相同。如前所述，我们定义了一个阈值数量，因此仅根据排序机制整合特定数量的 API。接下来，按照典型编码器-解码器模型的方式计算每个编码器的注意力。对于 AST 和代码，注意力权重针对每个词元计算；对于 API，则针对每个 API 计算注意力分数。随后，将各编码器的隐藏状态拼接起来，传递给全连接层，再传递给解码器。

B. 模型详情

编码器： 每个编码器接收来自不同输入的一种文本序列，包括用 $code$ 表示的代码片段、用 ast 表示的扁平化抽象语法树、用 def 表示的 API 定义序列，以及用 des 表示的 API 描述序列。代码和抽象语法树的编码器架构相似。下面以代码片段编码器为例，详细说明其结构。首先，编码器将词元序列映射为词元嵌入。然后，将嵌入输出传递给双向 GRU，以提取上下文信息。GRU 框架经过设计和优化，可避免长期依赖问题 [25]。在每个步骤中，GRU 的隐藏状态定义如下：

$$h_t^{code} = GRU(x_t^{code}, h_{t-1}^{code}) \tag{1}$$

其中，x_t^{code} 是步骤 t 时模型的代码输入（代码序列），h_{t-1}^{code} 表示步骤 $t-1$ 时代码输入对应的隐藏状态。

抽象语法树的隐藏状态通过如下类似方式获得：

$$h_t^{ast} = GRU(x_t^{ast}, h_{t-1}^{ast}) \tag{2}$$

其中，x_t^{ast}和 h_t^{ast}分别是展平后的抽象语法树输入和隐藏状态。

由于每个方法可能包含多个 API，也就是说包含多个 API 定义和描述，因此我们没有像 [23], 中那样将描述串联起来，而是将每个 API 映射到一个独立的编码器。下面以 API 定义为例说明这一过程。单个编码器会处理每个 API 定义，从而为每个 API 生成一个唯一的隐藏状态，如下所示：

$$\begin{aligned} h_t^{def_1} &= GRU(x_t^{def_1}, h_{t-1}^{def_1}) \\ h_t^{def_2} &= GRU(x_t^{def_2}, h_{t-1}^{def_2}) \\ &\vdots \\ h_t^{def_n} &= GRU(x_t^{def_n}, h_{t-1}^{def_n}) \end{aligned} \tag{3}$$

此外，这些隐藏状态会作为词元序列输入最终编码器，从而得到一个表示整个 API 定义的单一隐藏状态。由于输入已经是数值向量，因此最终编码器不包含嵌入层。其形式如下：

$$h_t^{def} = GRU(h^{def_t}, h_{t-1}^{def}) \tag{4}$$

其中，h_t^{def} 是 API 定义的最终隐藏状态。类似地，我们可以为 API 描述（h_t^{des}）定义所有公式。

注意力： 注意力是一种组件，使解码器能够更有效地关注相关词元 [26]。注意力机制可帮助解码器在每个预测步骤中聚焦于不同输入中的适当词元。例如，代码中的关键字 return 或 set 可以分别表示注释中的“get”或“update”等词语。因此，注释中的不同部分可以与代码中的不同部分相关联。让我们考虑最终注释与源代码之间的关系。为了计算目标词元 y_i 的注意力权重，会使用前一个词元的解码器隐藏状态 $h_{i-1}^{'}$，如下所示：

$$\alpha_{ij} = a(h_{i-1}^{'}, h_j^{code}) \tag{5}$$

其中，a 表示对齐模型，$h_{i-1}^{'}$ 表示解码器隐藏状态，α_{ij} 表示位置 j处的代码输入与位置 i 处的目标注释之间的相关程度。此外，上下文向量定义为输入隐藏状态的加权和，

$$c_i^{code} = \sum_j (\alpha_{ij} h_j^{code}) \tag{6}$$

同样，对其他三个输入也会计算上下文向量。

解码器： 解码器组件旨在生成人类语言的最终注释。以下公式用于生成输出以及时间步 t 的解码器隐藏状态：

$$h_t^{'} = GRU(h_{t-1}^{'}, y_{t-1}) \tag{7}$$

h'_{t-1} 表示前一个词元对应的解码器隐藏状态，y_{t-1} 是前一个词元。解码器的初始隐藏状态由四个输入对应的最终编码器隐藏状态组合而成。要计算初始隐藏状态，需要将四个编码器的隐藏状态拼接起来，并传入一个全连接层：

$$h'_0 = W_c[h^{code}; h^{ast}; h^{des}; h^{def}] + B_c \quad (8)$$

W_c 和 B_c 表示可训练参数。最终的上下文向量也以类似的方式计算。

$$c_t = W_c[c^{code}; c^{ast}; c^{des}; c^{def}] + B_c \quad (9)$$

接下来，根据前一个词元、隐藏状态和上下文向量计算时间步 t 的目标词元概率。

$$p(y_t|y_{t-1}, ..., y_1, x) = g(y_{t-1}, h'_t, c_t) \quad (10)$$

这里，g 表示 softmax 激活函数。

我们还在第 V-E 节研究了使用另一种基于循环神经网络（RNN）的架构，即长短期记忆网络（LSTM）替代门控循环单元的效果。

C. 排序系统

如前所述，输入模型的 API 列表基于排序系统生成；也就是说，我们可以对函数中使用的 API 进行排序，并将排名最高的 n 个 API 输入模型。排序依据是函数参数与 API 参数之间的差异数量。设 $P = \{p_1, ...p_n\}$ 为函数参数列表，$M_i = \{m_i1, ...m_ik\}$ 为 API_i 的参数列表（函数中使用了 API$_i$）。API_i 的得分 S_i 定义如下：

$$S_i = -Diff(P, M_i) \quad (11)$$

其中，$Diff$ 计算 P与 M_i之间的差异数量。该分数针对函数中使用的所有 API 计算。S_i 值最高的 API 排名最高。

IV. 实验

A. 数据集

在本研究中，我们使用 CodeSearchNet [24] 数据集开展实验，包括训练和测试所提模型，以及运行基线方法。近期许多研究都利用这一知名数据集开展实验，尤其是代码注释生成方面的研究 [27, 28]。CodeSerachNet 由来自 GitHub 仓库的多个不同数据集来源组成。这些数据集通过名为 Libraries.io 的工具收集，该工具旨在识别有效性较高的项目。所有没有有效许可证的项目都已从最终数据集中移除。该数据集包含六种不同的语言，包括 Python、JavaScript、Ruby、Go、Java 和 PHP。为了开展本研究，我们提取了其中的 Java 部分进行实验。每条数据记录都包含一个完整的 Java 方法及其对应的参考摘要，这些内容是本研究所需的。与 [19], 类似，我们在训练模型前进行了若干预处理步骤。遇到驼峰命名法或蛇形命名法时，会拆分词元。此外，我们会移除代码中的标点，并将所有词元转换为小写。与 [27] 研究类似，我们将代码和摘要的最大长度分别设为 256 和 64。我们移除了那些文本过长、无法根据截断后的代码生成抽象语法树的记录，以免丢失有价值的信息。这一预处理步骤是必要的，因为本研究中的部分模型会使用抽象语法树，而且我们必须确保条件一致，才能进行公平比较。如表 I 所示，数据集被划分为训练集、验证集和测试集。训练集和验证集用于训练模型并选择最佳模型，而测试集则用于评估模型。

表 I 数据集统计

数据集	之前的记录数 filtering	筛选后的记录数ing
训练集	164,923	137,007
验证集	5,183	4,326
Test	10,955	9,011

B. API 文档提取

我们修改了 [23], 开发的网页抓取器，以便从 Java 文档工具包（JDK）下载 API 定义。API 定义包含所有参数的类型和名称。为了匹配方法中的 API 与下载的 API 上下文，我们首先从给定的方法中提取所有 API。我们使用 SrcML 工具 [2] 将源代码转换为抽象语法树表示。然后，我们利用该表示识别 API。将识别出的 API 与抓取到的 API 匹配后，我们提取参数的名称和类型。

C. 实验设置

我们在 PyTorch 框架[3] 上实现了所提出的方法。训练模型时，嵌入和隐藏状态的大小分别设为 256 和 512。我们将批次大小和迭代次数（训练轮次）分别设为 32 和 100。不过，如果连续七次迭代均未观察到改进，训练就会停止。随机梯度下降 被用作优化器，初始学习率设为 0.1。为降低过拟合的可能性，我们采用了多个值为 0.2 的丢弃层。模型在训练集上训练，通过验证集从不同迭代轮次中选出最佳模型权重，最后在测试集上评估所选模型。

V. 实验结果

A. 基线方法

本节将介绍基线方法的详细信息及其架构。我们考虑以下四种最先进的基线方法：

[2]https://www.srcml.org/ [3] https://pytorch.org/

表 II APIContext2Com 与其他模型的 BLEU、METEOR 和 ROUGE-L 得分。最高分以粗体标示。

模型	指标					
	BLEU1 (%)	BLEU2 (%)	BLEU3 (%)	BLEU4（%）	METEOR（%）	ROUGE-L（%）
DeepCom	17.25	8.90	4.83	2.72	9.5	21.06
AST-AttendGRU	22.02	9.97	5.21	3.20	9.45	23.06
TL-CodeSum	21.65	10.10	5.69	3.74	8.36	22.10
Rencos	22.82	12.29	7.54	5.15	10.55	27.46
APIContext2Com	**24.70**	**14.45**	**8.92**	**5.88**	**12.13**	**29.36**

Rencos [21] Rencos 的作者提出了一种注意力编码器-解码器模型，并借助基于检索的摘要生成任务加以改进。他们通过检索相似方法并将其融入模型来提升模型效果。首先，模型在由代码和摘要对组成的训练集上进行训练。之后，在测试阶段，根据语义和语法特征，从训练集中检索出最相似的两种方法。代码和检索到的方法会被编码，解码器则通过融合这些输入生成最终摘要。他们在模型中使用双向长短期记忆网络层，作为循环神经网络的一种变体。

TL-CodeSum [22] 提出了一种采用门控循环单元框架的序列到序列模型，该模型能够适应从另一项不同但相关的任务中学习到的 API 知识。事实上，他们借助从 API 序列摘要模型迁移而来的知识，开发了一个代码注释生成模型。后者旨在建立 API 序列与相应摘要之间的映射关系。在 API 序列摘要任务中学习到的知识被迁移至代码注释生成模型，以提升模型性能。

AST-AttendGRU [19] 提出了一种具有注意力机制的门控循环单元编码器—解码器代码注释生成模型。他们将扁平化的抽象语法树序列纳入模型，以有效捕获代码片段的句法信息。他们利用两个不同的编码器，其中一个处理代码片段词元，另一个处理抽象语法树词元。通过这种方式，可以同时考虑代码片段的语义和句法信息，从而改进生成的注释。

DeepCom [18] DeepCom 提出了一种将抽象语法树融入模型的新方法。他们设计了一种树遍历技术，即基于结构的（SBT）遍历，能够有效地捕捉源代码的句法信息。此外，他们使用 SBT 将抽象语法树的表示展平，并以文本序列的形式将抽象语法树融入模型。

我们按照作者报告的最佳超参数运行所有这些模型。请注意，此处未选用 CodeBERT [27] 和 GraphCodeBERT [29]等预训练模型进行结果比较，因为：i）它们采用不同的范式，即通过在大型语料库上进行预训练来进行语言建模；ii）它们基于 Transformer [30] 架构；iii）它们是规模更大的模型，而且研究表明，语言模型规模越大，性能就越好 [31]。此外，类似研究也没有将这些模型与其方法进行比较 [20, 21, 32]。由于将这类语言模型纳入比较并不公平，我们仅将 APIContext2Com 与基于循环神经网络的 [21] 方法进行比较，这些方法并非语言模型。

B. 评估指标

与以往类似的研究 [14, 21, 33] 一样，我们采用三种广为人知的指标，包括 BLEU 1-4 [34],ROUGE-L[35] 和 METEOR [36] ，来评估所提出的方法。这些指标广泛应用于代码注释生成任务，用于评估生成注释与参考注释之间的相似度。换言之，这些指标体现了模型预测相似摘要的能力。

BLEU 借助简短惩罚来衡量参考摘要与生成摘要之间平均 n 元语法的精确率 [34],，其中 n 表示 n 元语法，并报告 $n \in [1, 4]$。简短惩罚会惩罚较短的译文，因为较短的句子往往会获得较高的 BLEU 分数。

ROUGE-L 计算召回率和精确率的加权调和平均数，称为 F 分数 [35]。另一方面，METEOR 是一种基于召回率的指标，用于衡量模型掌握参考句子内容的效果[36]。

C. RQ1：*APIContext2Com* 与其他模型的性能比较

在这个研究问题中，我们比较了所提出的方法与其他模型。结果见表 II。APIContext2Com 的表现优于所有其他模型；与表现最接近的基线 Rencos 相比，其 BLEU1、BLEU2、BLEU3、BLEU4、METEOR 和 ROUGE-L 分别提高了 1.88（8.24%）、2.16（17.58%）、1.38（18.3%）、0.73（14.17%）、1.58（14.98%）和 1.9（6.92%）。这一提升得益于从 API 上下文中提取信息，而相关 API 的筛选体现了这一点。此外，采用合适的架构分别处理每个 API，对捕获信息有显著影响。

Rencos 在基线方法中取得了最高分，这并不意外，因为它们通过检索相似的方法来引入外部知识。外部知识以及对.....的利用

表 III 排序机制和 API 数量对模型性能的影响。最高分以粗体标出。

模型	指标					
	BLEU1 (%)	BLEU2 (%)	BLEU3 (%)	BLEU4 (%)	METEOR(%)	ROUGE-L (%)
2 个 API	24.02	13.98	8.65	5.65	11.64	28.33
3 个 API	**24.70**	**14.45**	**8.92**	**5.88**	**12.13**	**29.36**
4 个 API	24.43	14.24	8.85	5.84	11.95	28.98
所有 API	23.91	13.92	8.58	5.73	11.84	28.52

表 IV 不同架构下所提模型与简单模型的比较。最高分以粗体显示。

模型	指标					
	BLEU1 (%)	BLEU2 (%)	BLEU3 (%)	BLEU4（%）	METEOR（%）	ROUGE-L（%）
APIContext2Com$_{GRU}$-APIcontext	21.23	9.85	5.45	3.72	9.21	22.83
APIContext2Com$_{GRU}$	**24.70**	**14.45**	**8.92**	**5.88**	**12.13**	**29.36**
APIContext2Com$_{LSTM}$-APIcontext	21.87	10.07	5.65	3.85	9.76	23.64
APIContext2Com$_{LSTM}$	24.23	14.18	8.69	5.71	12.04	28.72

抽象语法树结构信息使 Rencos 的表现优于其他模型。接下来是 TL-CodeSum 和 AST-AttendGRU。类似地，TL-CodeSum 引入 API 以改进摘要生成。然而，其结果并未达到预期，且与包含代码和抽象语法树的 AST-AttendGRU 几乎相同。原因在于，它们没有使用任何额外信息，只使用了 API 名称。这些名称已存在于源代码中，并且在处理源代码时已被处理。因此，仅添加 API 名称可能无法带来更多知识。在所有模型中，DeepCom 的得分最低。我们推测，这是因为源代码标记未被模型直接使用，导致它们未能有效捕获模型的词汇信息。

D. RQ2：排序机制对模型性能的影响，以及 *APIContext2Com* 中纳入 *API* 的最佳数量

由于有些 API 不仅没有帮助，反而会产生误导，因此本节将通过额外实验来确定 API 的最佳数量。如前所述，我们使用基于点的系统，根据 API 与主方法的相似度对其进行排序。因此，排序系统使我们能够选择并纳入排名前 n 的 API，而舍弃其余 API。不过，我们需要确定纳入这些 API 的数量（即 n）。我们尝试纳入不同数量的 API 进行实验，最终找到了使平均得分最高的 API 最佳数量。我们尝试了 2、3 和 4 个 API，结果如表 III 所示，使用 3 个 API 的模型得分最高。选择少于 3 个 API 会使性能下降 ≈ 0.7 BLEU-1。同样，纳入多于 3 个 API 也会降低性能；使用 4 个 API 时，性能会降低 ≈ 0.3 BLEU-1。表 III 的最后一行显示了不使用排序系统、而是考虑所有 API 时的得分。在这种情况下，与使用 3 个 API 相比，性能下降了 0.8 分。我们得出结论：根据所提出的排序系统筛选 API，有助于模型生成性能更高的注释。

有趣的是，尽管纳入模型的最佳数量是排名前 3 的 API，但纳入所有 API 或其他数量的 API，结果都与纳入 3 个 API 时不相上下。这证实了，除了排序系统之外，纳入 API 上下文和所提架构也有助于模型捕获相关信息。下一项研究问题将对此进行详细探讨。

E. RQ3：纳入 *API* 上下文及不同循环神经网络变体（门控循环单元和 *LSTM*）的影响

在这一研究问题中，我们探讨所提模型的简单版本，以更好地了解整合 API 上下文的影响。简单模型指的是不包含 API 上下文的所提模型，其输入仅包括源代码和平铺后的抽象语法树，如 APIContext2Com$^{-APIcontext}$ 所示。表 IV 的第一行展示了采用 GRU 架构的 APIContext2Com$^{-APIcontext}$ 的性能。与简单模型相比，包含 API 上下文的 APIContext2Com 在 BLEU 1–4、METEOR 和 ROUGE-L 指标上分别提升了 $\approx$3.47、4.6、3.47、2.16、2.92 和 6.53。因此，整合 API 的外部知识所带来的影响不容忽视，它能为模型提供有价值的信息，从而生成更好的注释。

我们进一步通过使用 LSTM 重复实验，探究不同架构（即 RNN 变体）的影响。我们用 LSTM 替换 GRU 层，并保持模型的其他组成部分不变。所用架构以模型名称的下标形式显示在表 IV 中。有趣的是，简单的 LSTM 模型 APIContext2Com$_{LSTM}$-APIcontext 的表现略高于 APIContext2Com$_{GRU}$-APIcontext。不过，与 LSTM 版本相比，融入 API 后，GRU 变体的结果提升更明显

。以 BLEU-1 作为代表性指标，APIContext2Com $_{GRU}$-APIcontext 和 APIContext2Com$_{LSTM}$-APIcontext 的得分分别为 21.23 和 21.87。加入 API 上下文后，APIContext2Com$_{GRU}$ 和 APIContext2Com$_{LSTM}$ 的得分分别达到 24.7 和 24.23；这意味着，相较于简单的 GRU 和 LSTM 模型，得分分别提高了 3.47 和 2.36。这可能与 GRU 架构能够更有效地捕获 API 信息有关，因为加入 API 上下文后，该架构带来的提升更大。

F. 人工评估

在本研究中，我们采用 BLEU、ROUGE-L 和 METEOR 三项指标来衡量参考注释与模型生成注释之间的相似度。然而，由于这些指标只考虑文本相似度，而非语义相似度，因此有时无法正确计算相似度。因此，我们开展人工评估，以验证这些指标所得到的量化结果。

为进行人工评估，我们使用亚马逊土耳其机器人（AMT）网站[4]。这是一个众包应用，可以将流程或任务分配给不同的远程工作人员。任务以人工智能任务（HITs，Human Intelligence Tasks）的形式定义，远程人员通过完成任务获得报酬。我们定义了一项任务，用于比较四个模型生成的注释与参考注释之间的相似度。这四个模型分别是所提模型和三个最佳基线模型，包括 Rencos、TL-Codesum 和 AST-AttendGRU。与先前的研究类似 [23]，我们从测试集中随机选取 100 个样本，其中每个样本包含参考注释和三个生成注释。评估人员（远程工作人员）需要使用 5 点李克特量表为每条生成注释评分，其中 1 表示生成注释与参考注释之间的语义相似度最低，5 表示相似度最高。为获得更准确且一致的结果，每项任务由三名不同的评估人员进行评分，并计算平均值作为最终得分。因此，总共需要完成 300 项任务。

首先通过提供示例对评估人员进行培训。为确保比较公平，评估人员不会获知生成注释的模型名称。评估人员给出的结果分别为：APIContext2Com 为 2.53，Rencos 为 2.39，AST-attendGRU 为 2.11，TL-Codesum 为 1.81。与 Rencos、AST-attendGRU 和 TL-Codesum 相比，APIContext2Com 分别提升了约 6%、20% 和 39%。这一结果证实，与其他方法相比，APIContext2Com 生成的注释与真实注释更为相似。

VI. 讨论

A. API 文档的分数分布与影响

在本节中，我们分析模型在记录 级别的性能，并研究将 API *ext* 文档整合到模型中的效果。为简化起见，本部分的计算仅考虑 BLEU-1。

[4]https://www.mturk.com/

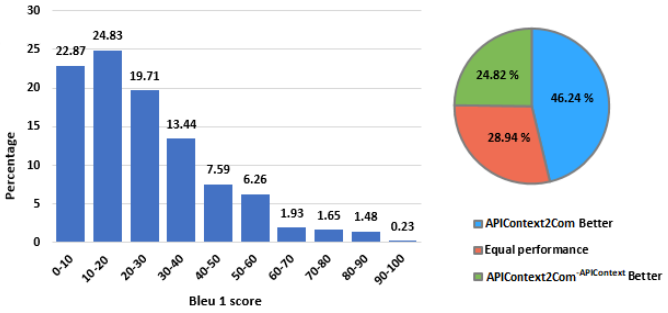

图 4. API 文档的影响

图 4 中的柱状图展示了 APIContext2Com 的 BLEU-1 分数分布。有相当一部分数据（～47%）的分数低于 20。这并不令人意外，因为有大量样本的输入无法为模型进行注释预测提供任何有用信息。这可能有三个主要原因。第一，代码可读性较差，例如命名不规范。第二，代码本身没有问题，模型也生成了合理的摘要，但参考注释未必总能反映代码的功能，例如技术债务注释 [37]。第三，代码和注释都没有问题，但模型在训练期间没有见过足够多的相似示例。约 20% 的样本分数分布在 20 到 30 之间，这一区间的分数接近平均值。超过 32% 的样本得分高于 30，值得注意的是，有时分数接近 100。调查这些情况后，我们发现许多高分案例中，API 的注释与参考注释非常相似。

图 4 中的饼图展示了向简单模型 APIContext2Com -APIcontext（即仅使用代码和抽象语法树）添加 API 文档的效果。该饼图比较了 APIContext2Com-APIcontext 和 APIContext2Com 模型在记录 层面的得分。对于 ～ 29% 的样本，两个模型生成的注释相似。这是合理的，因为这些样本大多没有使用任何 API。在超过 46% 的案例（占比最高）中，APIContext2Com 给出的注释得分更高。这些样本中的 API 文档大多很有帮助且相关。其余样本通常包含 API，而其文档在某种程度上具有误导性，因此 APIContext2Com-APIcontext 的得分更高。

B. 代码和注释长度

分析注释生成模型的一种常见方法，是观察代码长度或注释长度增加时 BLEU 分数的变化。图 5 展示了 APIContext2Com 以及两个表现最佳的基线方法 Recos 和 AST-AttendGRU 的 BLEU-1 分数。这里列出基线方法，是为了比较不同模型呈现的趋势。图中展示了代码长度以 20 为步长、注释长度以 3 为步长时的平均分数。例如，右图中显示的分数

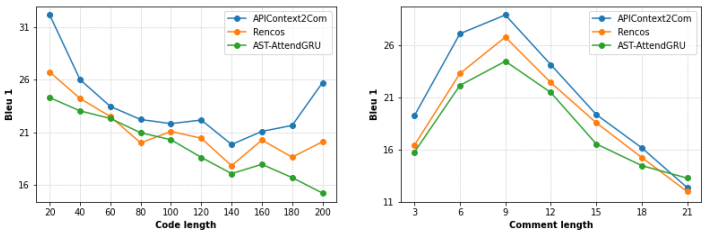

图 5. 不同代码和注释长度下的 BLEU 分数

对应注释长度为 6 时的分数，反映的是长度在 3 到 6 之间的注释的平均分数。总体而言，APIContext2Com 的分数高于其他模型。图表显示，APIContext2Com 在不同长度下均有显著提升。然而，随着代码或注释长度增加，所有模型的分数都呈下降趋势。在代码长度方面，所有模型中，APIContext2Com 的分数稳步下降，在长度超过 140 后略有回升。相比之下，Rencos 的趋势起伏不定，而 AST-AttendGRU 的表现则随着代码长度增加而下降。在注释长度方面，所有模型的趋势相似。表现随着长度增加而提升，直到长度达到 9，之后开始下降。与 Rencos 和 AST-AttendGRU 相比，APIContext2Com 在几乎所有注释长度下的分数都更高。

C. 案例研究

```
public static void touch(File... files) throws IOException
    {
        long now = System.currentTimeMillis();
        for (File f : files) {
            if (!f.createNewFile() && !f.setLastModified(now)) {
                throw new IOException("Failed to touch " + f);
            }
        }
    }
```

Reference:

creates empty files at the specified paths or updates the last modification time of the files at the specified paths

Generation:

APIContext2Com: creates new file or set the last modified time

APIContext2Com$^{-APIcontext}$: creates a file

APIs Description:

API 1 : currentTimeMillis : Returns the current time in milliseconds.
API 2 : createNewFile : Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
API 3 : setLastModified : Sets the last-modified time of the file or directory named by this abstract pathname.

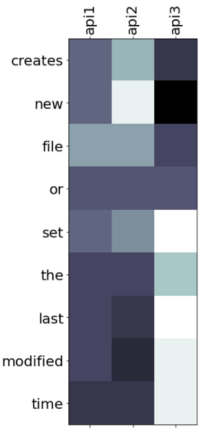

图 6. 使用 API 描述后生成的注释得到改进的示例。

本节展示两个示例，说明 API 上下文如何提升预测注释的质量。第一个示例展示 API 描述的作用，第二个示例则展示 API 定义的优势。在图 6 的示例 1 中，简单

```
public static String cutToIndexOf(String string, final String substring) {
        int i = string.indexOf(substring);
        if (i != -1) {
            string = string.substring(0, i);
        }
        return string;
    }
```

Reference:

cuts the string from beginning to the first index of provided substring

Generation:

APIContext2Com: cut the string from the begin index of the string

APIContext2Com$^{-APIcontext}$: cut the string from the string

APIs Definition:

API 1 : indexOf(int ch)
API 2 : substring(int beginIndex, int endIndex)

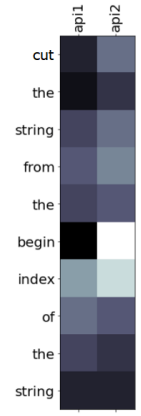

图 7. 使用 API 定义后生成的注释得到改进的示例。

模型 APIContext2Com$^{-APIcontext}$ 生成的“创建文件”只提供了有限的信息，与参考注释相比尤为明显。然而，从参考注释来看，APIContext2Com 预测出的注释更加准确。快速查看该方法中使用的 API 及其各自的注释后可以发现，API2 和 API3（createNewFile 和 setLastModified）的描述与参考注释非常相似。图中绘制的热力图展示了 APIContext2Com 对 API 描述的注意力。颜色越浅，表示注意力越强。如图所示，APIContext2Com 在预测“创建新文件”时会更多地关注第二个 API，在预测“设置最后修改时间”时也会关注第三个 API。这证实了我们的假设：如果选择得当，API 描述有助于生成注释。

图 7 展示了第二个示例：APIContext2Com$^{-APIcontext}$ 模型生成的注释未能包含关键字“beginning”，而 APIContext2com 在生成的注释中加入了这个词。快速查看代码片段可以发现，其中没有与“beginning”相似的词。因此，预期 APIContext2Com$^{-APIcontext}$ 会遗漏这个词。然而，查看此方法中的 API 后发现，API 2 的定义中有一个参数名为 beginIndex，热力图也证实模型在预测“begin”时关注了 API2。

VII. 有效性威胁

需要考虑设计神经网络模型，同时复现基线方法的输出。为了确认代码的正确性，我们对每个部分都进行了仔细核查。对于基线方法，我们使用了作者提供的公开源代码，并采用作者推荐的配置。

我们采用了一个大规模且知名的 Java数据集来确保输出的有效性，但这一发现可能无法推广到其他语言。还需要开展更多工作，以

了解 API 上下文对其他编程语言的影响。

在所有定量实验中，分数均根据注释生成研究中使用的指标计算。为减轻单一指标可能带来的问题，我们采用了多个指标。此外，我们还进行了人工评估实验，以验证自动指标所给出结果的正确性。

VIII. 相关工作

以往研究采用基于模板和信息检索的方法 [3, 5] 来完成自动代码注释生成任务。近年来，深度神经网络被应用于代码注释生成，Iyer 等人 [12]是最早开展这项工作的研究者。他们提出了端到端的编码器—解码器框架 CODE-NN，用于为 C# 和 SQL 生成注释。

其他研究以不同方式利用抽象语法树来改进注释。Hu 等人 [18]将抽象语法树纳入序列到序列 LSTM 模型，以更好地捕获源代码的句法信息。LeClair 等人 [19] 将源代码和抽象语法树作为两种不同的输入结合到模型中。Alon 等人[38]提出了一种将抽象语法树纳入模型的创新方案。他们在抽象语法树表示树中选取多种遍历路径，并将其作为 LSTM 模型的不同输入。在一些研究中，不同框架采用基于树的编码器来利用抽象语法树，包括图神经网络（GNN） [39],、Tree-LSTM [40] 和 Tree-transformer [41] 。Fernandes 等人 [42] 设计了一种图神经网络模型，采用三种图分别表示源代码、抽象语法树和词汇用法。LeClair 等人 [39]提出了一种基于图神经网络的模型，能够有效适配抽象语法树结构（默认结构）。Wan 等人 [14]将序列化源代码文本和抽象语法树纳入强化学习模型，以生成可读注释。Zhou 等人 [43]提出了一种图注意力模型，用于处理抽象语法树表示并提取代码的结构信息。

其他方法则考虑将不同方面纳入模型。Nie 等人 [44]使用源代码和摘要的时间戳，并提出了一种分时段评估技术。Wei 等人 [45]研究了注释生成与代码生成（两项相反的任务）之间的关系，并开发了一种双重训练框架，将对偶性融入注意力权重，以提升这两项任务的性能。Allamanis 等人 [46]提出了一种采用卷积架构的模型，能够捕捉长程注意力信息。Sharma 等人 [47] 提出了一种基于字符的语言模型来提取语义知识，并提出了一种命名实体识别模型来捕捉模型的结构信息。

一些研究也关注抽象语法树的评估或其不同方面。Z¨ugner 等人 [48]采用了语言无关特征，例如代码片段，以及可直接从抽象语法树中提取的特征。Tang 等人 [49] 认为，抽象语法树长度过长，无法直接用于模型，而且会降低模型性能。Shi 等人 [50] 进行了系统分析，并揭示了 BLEU 分数的不同变体会对模型性能产生显著影响。

在文献中，利用外部知识提升性能是另一种方法。Wei [20]和 Zhang 等人 [21] 采用信息检索技术检索相似的代码片段，并利用相应的注释。EditSum [32] 提出了一种检索并编辑的方案，以利用相似的代码模式。Wang 等人 [28]纳入了统一建模语言和封闭类名。Haque 等人 [51]采用同一文件中主方法所在位置的其他方法（文件上下文）作为额外输入。Bansal 等人 [52] 通过结合项目上下文并纳入整个项目中的其他文件作为额外输入，扩展了外部知识的使用范围。Hu 等人 [22] 利用 API 序列的名称训练 API 序列摘要模型，然后将知识迁移到代码注释生成任务中。Shahbazi 等人 [23] 将 API 文档集成到 Transformer 模型中。

除了循环神经网络的变体 LSTM 和门控循环单元外，Transformer[30] 也被用作主要架构 [23, 33] ，或在一些近期研究中得到探讨 [53]。最后，另一种方法是使用预训练语言模型（PLM） [27, 47, 54, 55, 56, 57, 58,59]，例如 CodeBERT [27]。该模型最初在大规模通用语料库上进行预训练，随后针对包括注释生成在内的目标任务进行微调。

在所有相关研究中，我们的工作与 Hu 等人和 Shahbazi 等人 [22, 23]的工作最为相似。前者仅使用 API 名称，后者则融入了 API 文档。尽管 [23], 使用了 API 文档，他们报告称使用 API 知识会产生负面结果，尤其是在 API 数量较多时。我们的工作在多个方面与这项研究不同。我们提出了一种新架构，分别对每个 API 进行编码，添加 API 定义，并采用排序系统。

IX. 结论

在本研究中，我们提出了一种序列到序列、多编码器代码注释生成架构，即 APIContext2Com。我们使用有关预定义 API 和 API 上下文的外部信息作为额外输入，并在 Java 上评估了模型。API 上下文由 API 描述和定义组成，这些信息收集自 JDK 文档。我们还设计了一种排序机制，用于筛除那些无助于增加知识、因而无法帮助预测解释源代码功能的注释的 API。研究结果表明，该方法优于当前最先进的模型 Rencos，且与人工评估结果一致。实验还揭示了使用 API 上下文的有效性。未来，我们计划改进排序机制，并利用其他外部知识来源拓展这项工作，同时将该技术应用于其他编程语言。

X. 数据可用性

我们将收集的 API 文档开源：https://zenodo.org/record/7689869#.Y K0nbMJD-

参考文献

[1] A. J. Ko、B. A. Myers、M. J. Coblenz 和 H. H. Aung，《软件维护任务中开发者如何查找、关联和收集相关信息的探索性研究》，IEEE 软件工程学报，第 32 卷，第 12 期，第 971–987 页，2006 年。[2] T. D. LaToza、G. Venolia 和 R. DeLine，《维护心智模型：开发者工作习惯研究》，载于第 28 届国际软件工程会议论文集，2006 年，第 492–501 页。[3] G. Sridhara、E. Hill、D. Muppaneni、L. Pollock 和 K. Vijay-Shanker，《朝着自动生成 Java 方法的摘要注释迈进》，载于 IEEE/ACM 自动化软件工程国际会议论文集，2010 年，第 43–52 页。[4] X. Xia、L. Bao、D. Lo、Z. Xing、A. E. Hassan 和 S. Li，《衡量程序理解：一项针对专业人士的大规模实地研究》，IEEE 软件工程学报，第 44 卷，第 10 期，第 951–976 页，2017 年。[5] L. Moreno、J. Aponte、G. Sridhara、A. Marcus、L. Pollock 和 K. Vijay-Shanker，《自动生成 Java 类的自然语言摘要》，载于 2013 年第 21 届国际程序理解会议（ICPC）。IEEE，2013 年，第 23–32 页。[6] J. Singer、T. Lethbridge、N. Vinson 和 N. Anquetil，《软件工程工作实践考察》，载于《CASCON 首个十年高影响力论文集》，2010 年，第 174–188 页。[7] B. P. Eddy、J. A. Robinson、N. A. Kraft 和 J. C. Carver，《评估源代码摘要技术：复现与扩展》，载于 2013 年第 21 届 ICPC。IEEE，2013 年，第 13–22 页。[8] S. Haiduc、J. Aponte 和 A. Marcus，《通过源代码摘要支持程序理解》，载于 2010 年 ACM/IEEE 第 32 届国际软件工程会议论文集，第 2 卷。IEEE，2010 年，第 223–226 页。[9] S. Haiduc、J. Aponte、L. Moreno 和 A. Marcus，《自动文本摘要技术在源代码摘要中的应用》，载于 2010 年第 17 届逆向工程工作会议。IEEE，2010 年，第 35–44 页。[10] P. Rodeghero、C. McMillan、P. W. McBurney、N. Bosch 和 S. D'Mello，《通过针对程序员的眼动追踪研究改进自动源代码摘要》，载于第 36 届国际软件工程会议论文集，2014 年，第 390–401 页。[11] E. Wong、T. Liu 和 L. Tan，《Clocom：挖掘现有源代码以自动生成注释》，载于 2015 年 IEEE 第 22 届软件分析、演化与再工程国际会议（SANER）。IEEE，2015 年，第 380–389 页。[12] S. Iyer、I. Konstas、A. Cheung 和 L. Zettlemoyer，《使用神经注意力模型总结源代码》，载于计算语言学协会第 54 届年会论文集（第 1 卷：长篇论文），2016 年，第 2073–2083 页。[13] X. Hu、G. Li、X. Xia、D. Lo 和 Z. Jin，《深度代码注释生成》，载于 2018 年 IEEE/ACM 第 26 届国际程序理解会议（ICPC）。IEEE，2018 年，第 200–20 010 页。[14] Y. Wan、Z. Zhao、M. Yang、G. Xu、H. Ying、J. Wu 和 P. S. Yu，《通过深度强化学习改进自动源代码摘要》，载于第 33 届 ACM/IEEE 自动化软件工程国际会议论文集，2018 年，第 397–407 页。[15] D. Movshovitz-Attias 和 W. Cohen，《用于预测编程注释的自然语言模型》，载于计算语言学协会第 51 届年会论文集（第 2 卷：短篇论文），2013 年，第 35–40 页。[16] D. Bahdanau、K. Cho 和 Y. Bengio，《通过联合学习对齐与翻译实现神经机器翻译》，arXiv 预印本：1409.0473，2014 年。[17] T. Haije、B. O. K. Intelligentie、E. Gavves 和 H. Heuer，《使用神经翻译模型自动生成注释》，《信息与软件技术》，第 55 卷，第 3 期，第 258–268 页，2016 年。[18] X. Hu、G. Li、X. Xia、D. Lo 和 Z. Jin，《融合混合词汇与句法信息的深度代码注释生成》，《实证软件工程》，第 25 卷，第 3 期，第 2179–2217 页，2020 年。[19] A. LeClair、S. Jiang 和 C. McMillan，《用于生成程序子程序自然语言摘要的神经模型》，载于 2019 年 IEEE/ACM 第 41 届国际软件工程会议（ICSE）。IEEE，2019 年，第 795–806 页。[20] B. Wei，《检索与精炼：基于范例的神经注释生成》，载于 2019 年第 34 届 IEEE/ACM 自动化软件工程国际会议（ASE）。IEEE，2019 年，第 1250–1252 页。[21] J. Zhang、X. Wang、H. Zhang、H. Sun 和 X. Liu，《基于检索的神经源代码摘要》，载于第 42 届国际软件工程会议论文集。IEEE，2020 年。[22] X. Hu 、 G. Li 、X. Xia、D. Lo、S. Lu 和 Z. Jin，《总结源代码利用迁移的 API 知识编写代码，”2018. [23] R. Shahbazi、R. Sharma 和 F. H. Fard，“Api2com：利用 API 文档改进自动生成的代码注释”，载于*2021* 年 *IEEE/ACM* 第 *29* 届国际程序理解会议（*ICPC*）。IEEE，2021 年，第 411–421 页。[24] H. Husain、H.-H. Wu、T. Gazit、M. Allamanis 和 M. Brockschmidt，“CodeSearchNet 挑战赛：评估语义代码搜索的现状”，*arXiv* 预印本 *arXiv:1909.09436*，2019 年。[25] J. Chung、C.. G¨ulc.ehre、K. Cho 和 Y. Bengio，“门控循环神经网络在序列建模中的实证评估”，*CoRR*，第 abs/ 1412 3555, 2014 [ ] 1412 3555 卷。. 在线。可访问：http://arxiv.org/abs/。I. Sutskever、O. Vinyals 和 Q. V. Le，“利用神经网络进行序列到序列学习”，神经信息处理系统进展，第 27 卷，2014 年。[27] Z. Feng、D. Guo、D. Tang、N. Duan、X. Feng、M. Gong、L. Shou、B. Qin、T. Liu、D. Jiang 和 M. Zhou，“CodeBERT：一种用于编程语言和自然语言的预训练模型”，载于计算语言学协会论文集：*EMNLP 2020*。在线：计算语言学协会，2020 年 11 月，第 1536–1547 页。[26] [ ] 2020 139 在线。可访问：https://aclanthology.org/.findings-emnlp。Y. Wang、L. Du、E. Shi、Y. Hu、S. Han 和 D. Zhang，“CocoGUM：基于 UMLS 上的多关系图神经网络进行上下文代码摘要”，Microsoft，技术报告 MSR-TR-2020-16，2020 年 5 月。 [在线]。可访问：https://www.microsoft.com/en-us/research/publication/cocogum-contextual-code-summarization-with-multi-relational-gnn-on-umls/

[28] [29] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert：通过数据流预训练代码表示”，载于第*9*届国际学习表征会议（*ICLR 2021*），线上会议，奥地利，*2021*年*5*月*3*日至*7*日。OpenReview.net，2021年。[在线]。获取地址：https://openreview.net/forum?id=jLoC4ez43PZ [30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “注意力就是你所需要的一切”，载于神经信息处理系统进展，2017年，第5998–6008页。[31] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT：用于语言理解的深度双向 Transformer 预训练”，载于*2019*年计算语言学协会北美分会年会论文集：人类语言技术，第*1*卷（长篇与短篇论文）。明尼阿波利斯，明尼苏达州：计算语言学协会，2019年6月，第 4171 4186 [ ] 19 1423 –页。在线。[32] 获取地址：https://aclanthology.org/N - J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, “EditSum：用于源代码摘要的检索与编辑框架”，载于*2021*年第*36*届 *IEEE/ACM* 自动化软件工程国际会议（*ASE*）。IEEE，2021年，第155–166页。[33] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “一种用于源代码摘要的基于 Transformer 的方法”，载于计算语言学协会第*58*届年会论文集。线上：计算语言学协会，2020年7月，第4998–5007 [ ] 2020 449 页。在线。[34] 获取地址：https://aclanthology.org/ .acl-main. K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU：一种用于机器翻译自动评估的方法”，载于计算语言学协会第*40*届年会论文集，2002年，第311–318页。[35] C.-Y. Lin, “ROUGE：一种用于自动评估摘要的软件包”，载于文本摘要的新方向，2004年，第74–81页。[36] S. Banerjee and A. Lavie, “METEOR：一种用于机器翻译评估的自动指标，与人工判断的相关性更高”，载于计算语言学协会机器翻译和/或摘要内在与外在评估指标研讨会论文集，2005年，第65–72页。[37] R. Sharma, R. Shahbazi, F. H. Fard, Z. Codabux, and M. Vidoni, “R语言中的自认技术债务：检测与成因”，自动化软件工程，第29卷，第2期，第1–41页，2022年。[38] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq：从代码的结构化表示生成序列”，载于第*7*届国际学习表征会议（*ICLR 2019*），美国路易斯安那州新奥尔良，*2019*年*5*月*6*日至*9*日。OpenReview.net，2019年。 [在线]。获取地址：https://openreview.net/forum?id=H1gKYo09tX [39] A. LeClair, S. Haque, L. Wu, and C. McMillan, “通过图神经网络改进代码摘要”，载于第*28*届国际程序理解会议论文集，2020年，第184–195页。[40] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, “采用扩展 Tree-LSTM 的自动源代码摘要”，载于*2019*年国际联合神经网络会议（*IJCNN*）。IEEE，2019年，第1–8页。

[41] J. Harer, C. P. Reale, and P. Chin, “Tree-transformer：一种用于校正树结构数据的基于 Transformer 的方法”，*CoRR*，第abs/1908.00449卷，2019年。 [在线]。获取地址：http://arxiv.org/abs/1908. 00449[42] P. Fernandes, M. Allamanis, and M. Brockschmidt, “结构化神经摘要”，*arXiv*预印本 *arXiv:1811.01824*，2018年。[43] Y. Zhou, J. Shen, X. Zhang, W. Yang, T. Han, and T. Chen, “采用图注意力网络的自动源代码摘要”，《系统与软件杂志》，第188卷，第111257页，2022年。[44] P. Nie, J. Zhang, J. J. Li, R. Mooney, and M. Gligoric, “评估方法对代码摘要的影响”，载于计算语言学协会第*60*届年会论文集（第*1*卷：长篇论文），2022年，第4936–4960页。[45] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, “作为代码摘要双重任务的代码生成”，载于神经信息处理系统进展，2019年，第6563–6573页。[46] M. Allamanis, H. Peng, and C. Sutton, “用于源代码极限摘要的卷积注意力网络”，载于国际机器学习会议。PMLR，2016年，第2091–2100页。[47] R. Sharma, F. Chen, F. Fard, and D. Lo, “关于 BERT 中代码注意力的探索性研究”，载于第*30*届 *IEEE/ACM* 国际程序理解会议论文集，ICPC ’22系列。美国纽约州纽约：计算机协会，2022年，第437–448页。 [ ] 10 1145 3524610 3527921 在线。获取地[48] 址：https://doi.org/ . / . D. Zugner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Gunnemann, “从结构和上下文中学习源代码的语言无关表示”，*arXiv*预印本 *arXiv:2103.11318*，2021年。[49] Z. Tang, X. Shen, C. Li, J. Ge, L. Huang, Z. Zhu, and B. Luo, “AST-Trans：采用高效树结构注意力的代码摘要”。ICSE，2022年。[50] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, “关于神经代码摘要的评估”，载于第*44*届国际软件工程会议论文集，2022年，第1597–1608页。[51] S. Haque, A. LeClair, L. Wu, and C. McMillan, “通过关注文件上下文改进子程序的自动摘要”，载于

第*17*届国际软件仓库挖掘会议论文集, 2020, 第300–310页。[52] A. Bansal、S. Haque 和 C. McMillan，“用于子程序神经源代码摘要的项目级编码”，载于*2021 IEEE/ACM* 第*29*届国际程序理解会议（*ICPC*）。IEEE，2021，第253–264页。[53] R. Sharma，“用于注释生成和程序理解的源代码表示”，硕士论文，不列颠哥伦比亚大学，2021。[54] X. Jiang、Z. Zheng、C. Lyu、L. Li 和 L. Lyu，“Treebert：一种面向编程语言的树形预训练模型”，载于人工智能中的不确定性。PMLR，2021，第54–63页。[55] Z. Zeng、H. Tan、H. Zhang、J. Li、Y. Zhang 和 L. Zhang，“关于程序理解与生成预训练模型的全面研究”，载于第*31*届 *ACM SIGSOFT*软件测试与分析国际研讨会论文集，2022，第39–51页。[56] D. Wang、Z. Jia、S. Li、Y. Yu、Y. Xiong、W. Dong 和 X. Liao，“连接预训练模型与源代码理解下游任务”，载于第*44*届国际软件工程会议论文集，2022，第287–298页。[57] Y. Wan、W. Zhao、H. Zhang、Y. Sui、G. Xu 和 H. Jin，“它们捕获了什么？源代码预训练语言模型的结构分析”，载于第*44*届国际软件工程会议论文集，系列：ICSE ’22。美国纽约州纽约：计算机协会，2022，第2377–2388页。 [在线]。网址：https://doi.org/ 10 1145 3510003 3510050 。/。[58] M. A. Hadi、I. N. B. Yusuf、F. Thung、K. G. Luong、J. Lingxiao、F. H. Fard 和 D. Lo，“关于预训练模型在 API学习中的有效性”，载于第*30*届 *IEEE/ACM*国际程序理解会议论文集，系列：ICPC ’22。美国纽约州纽约：计算机协会，2022，第309–320页。 [ ] 10 1145 3524610 3527886 在线。网[59] 址：https://doi.org/。/。M. A. Hadi 和 F. H. Fard，“评估预训练模型在软件工程用户反馈分析中的应用：应用评论分类研究”，*CoRR*，第abs/2104.05861卷，2021。 [在线]。网址：https://arxiv.org/abs/2104.05861