

目录

1. 引言	3
1.1. 研究背景	3
1.1.1. 智能问答系统的重要性和应用价值	3
1.1.2. Deepin 操作系统用户面临的问题和挑战	3
1.1.3. RocketQA、MacBERT 和 ChatGLM 模型介绍	4
1.1.4. 研究动机和意义	4
1.2. 问题陈述	5
1.3. 目标与意义	5
1.4. 技术路线介绍	6
2. 文献综述	7
2.1. 智能问答系统	7
2.1.1. 问答系统分类	7
2.1.2. 问题类型	7
2.1.3. 问答系统用例	8
2.2. RocketQA 模型	8
2.2.1. 对偶式检索模型	8
2.2.2. 双塔式召回模型	9
2.2.3. 交互式排序模型	10
2.3. MacBERT 模型	11
2.3.1. MLM 的基本原理	11
2.3.2. MacBERT 中的 MLM 策略	11
2.3.3. MLM 在 MacBERT 中的作用	11
2.3.4. MacBERT 中 MLM 的训练比例	11
2.4. ChatGLM 模型	12
2.4.1. GLM 架构	12
2.4.2. 监督微调 (Supervised Fine-Tuning)	13
2.4.3. 反馈自助 (Self-Supervised Learning)	13
2.4.4. 人类反馈强化学习技术 (Human Feedback Reinforcement Learning)	13
3. 系统设计与架构	14
3.1. 整体架构	14
3.2. 模块设计	16
3.2.1. 数据处理与索引构建模块	16
3.2.2. 问答服务应用程序模块	16
3.2.3. 模型应用模块	17
3.3. 核心算法	17
3.3.1. RocketQA 模型在系统中的应用	17
3.3.2. MacBERT 模型在系统中的应用	18
3.3.3. ChatGLM 模型在系统中的应用	19
4. 项目实施	20
4.1. 数据集描述	20
4.2. 项目配置	20
4.2.1. 环境配置	20
4.2.2. 代码配置	20

4.2.3. 数据配置	20
4.2.4. 模型配置	21
4.3. 数据预处理流程	21
4.3.1. 数据集扩展：玲珑使用手册内容	22
4.4. 模型调整：从 MacBERT-Large 到 ChatGLM 的转变	23
4.4.1. MacBERT-Large 与 ChatGLM 的比较	23
4.4.2. 转变的必要性	23
4.4.3. 技术挑战与解决方案	23
4.5. 在 Deepin 系统上的实现	24
4.5.1. 环境搭建	24
4.6. 各个模块的核心代码	25
4.6.1. 数据处理与构建索引模块	25
4.6.2. 问答服务应用程序模块	27
5. GUI 界面设计与实现	30
5.1. 设计思路	30
5.2. 组件使用	30
5.3. 事件处理	30
5.4. 布局管理	30
5.5. 用户交互	30
5.6. 错误处理和验证	31
5.7. 页面实现	31
5.8. 核心代码	31
6. Web 界面设计与实现	34
6.1. Flask 技术的选择原因	34
6.2. 规划与设计	34
6.3. Flask 技术应用	35
6.4. 界面实现	36
7. 项目结果与分析	37
7.1. 项目结果展示	37
7.2. 结果分析与讨论	38
7.2.1. 项目总结	38
7.2.2. 项目分析	39
8. 博客记录	40
8.1. 开发过程中的心得与体会	40

1. 引言

1.1. 研究背景

1.1.1. 智能问答系统的重要性和应用价值

智能问答系统是一种基于自然语言处理技术的智能对话系统，旨在帮助用户快速准确地获取文档信息。用户可以通过输入问题或关键词来与机器人进行交互，机器人会根据用户的输入在文档库中搜索相关内容，并给出相应的答案或建议。文档问答机器人可以应用在各种领域，如客服、教育、医疗等，帮助用户解决问题、提高效率。通过不断学习和优化，文档问答机器人可以逐渐提升其问答能力，为用户提供更加个性化、智能化的服务。

提高效率：智能问答系统能够快速、准确地响应用户的问题，为用户提供所需的信息，从而大大提高工作效率。在客户服务、在线教育、医疗咨询等领域，智能问答系统都能显著减少等待时间，提升服务效率。**降低成本：**相较于人工客服或专家咨询，智能问答系统能够降低企业的人力成本。同时，系统可以 24 小时不间断地提供服务，无需考虑人员轮班和休息问题，进一步降低了运营成本。**扩大服务范围：**智能问答系统不受地域、时间等限制，能够覆盖更广泛的用户群体。无论用户身处何地，只要有网络连接，就能随时随地获取所需信息，享受便捷的服务。**提升用户体验：**智能问答系统具有自然、流畅的语言交互能力，能够为用户提供更加友好、人性化的服务。通过智能推荐、个性化定制等功能，系统还能根据用户的需求和偏好提供精准的信息，进一步提升用户体验。

随着人们对信息获取和问题解决效率的不断追求，智能问答系统逐渐成为了满足用户需求的重要工具。

1.1.2. Deepin 操作系统用户面临的问题和挑战

Deepin 操作系统，原名 Linux Deepin，中文通称深度操作系统，是由武汉深之度科技有限公司开发的 Linux 操作系统。Deepin 操作系统基于 Linux 内核，以桌面应用为主，致力于为用户提供美观易用、安全可靠的操作系统体验。基于 Linux 内核，整个系统采用开源授权，代码公开可见并接受社区审核与贡献。然而，即使是对用户友好的操作系统，用户仍然可能面临各种各样的问题，例如软件安装、系统设置、故障排除等。这些问题可能会影响用户的体验和工作效率，因此需要一种快速、准确的解决方案。

1.1.3. RocketQA、MacBERT 和 ChatGLM 模型介绍

RocketQA 是一个基于预训练的大规模问答模型，具有强大的语言理解和知识获取能力，可以准确地回答各种类型的问题。RocketQA 是全世界首个面向中文的端到端搜索问答工具包，该工具包提供了一个在百万级中文阅读理解数据集 DuReader 上训练的中文段落检索模型，同时也提供了面向普通开发者的一键索引问答数据和一键启动问答服务的功能。RocketQA 是一个基于 BERT 的深度语义检索模型，其内部包含了两个子模型：双塔式召回模型和交互式排序模型，其中召回模型负责从文档库中快速筛选出与用户查询度高度相关的候选文档集，排序模型则会进一步从候选文档集中找到与用户查询最相关的文档，并返回给用户。

MacBERT（MLM as correction BERT）是一种基于预训练语言模型 BERT 的改进模型，主要用于自然语言处理任务。它通过采用 Masked Language Modeling（MLM）作为修正器，对原始 BERT 模型的训练过程进行优化，从而提高了模型在各项 NLP 任务中的性能。

与此同时，ChatGLM 是一个生成式对话模型，能够生成具有上下文连贯性的自然语言对话。Chat-GLM（Chat-based Generative Language Model）是一种基于对话的生成式语言模型，通常构建在大型神经网络架构上，如 GPT（Generative Pre-trained Transformer）模型。Chat-GLM 旨在模仿自然语言对话，并能够生成连贯、流畅的文本回复，从而与用户进行交互。Chat-GLM 通常通过大规模的对话数据集进行预训练，以学习自然语言的语法、语义和上下文关系。在预训练完成后，它可以用于各种对话型任务，如智能客服、聊天机器人、虚拟助手等。Chat-GLM 模型的核心思想是利用大规模的对话数据来学习对话的模式和规律，从而使其能够生成自然、连贯的对话文本。该模型在对话生成方面表现出色，能够模拟人类对话的特点，并且能够根据上下文信息作出合适的回复。

这两个模型的结合为我们提供了一个强大的工具，可以用于构建智能问答系统，以解决 Deepin 操作系统用户可能遇到的各种问题。

1.1.4. 研究动机和意义

Deepin 是一款具有自主知识产权的国产操作系统，它拥有丰富的功能和良好的用户体验，逐渐在国内市场获得广泛认可。然而，对于大多数用户来说，深入了解 Deepin 系统的功能和特性仍然存在一定的难度。因此，我们基于 Deepin 的官方文档，构建了一个全面的文档数据库，作为问答机器人的数据源。该问答机器人能够解析用户输入的问题，并在 Deepin 文档数据库中进行智能搜索，快速定位到与问题相关的内容，并给出准确的答案。无论是关于 Deepin 系统的安

装、配置、使用技巧还是故障排除，用户都可以通过该问答机器人轻松获取所需信息。

鉴于 Deepin 操作系统用户可能面临的各种问题，我们的研究旨在开发一种智能问答系统，利用 RocketQA 和 ChatGLM 模型，为用户提供快速、准确的问题解答和技术支持。通过这种方式，我们旨在提高 Deepin 系统用户的使用体验，增强其对操作系统的满意度，并推动智能问答技术在实际应用中的发展和应用。

1.2. 问题陈述

实现一个依据 wiki.deepin.org 网站内容回答问题的机器人。<https://wiki.deepin.org> 上有 900 多条 deepin 系统相关的中文教程和词条，编写能根据这些内容回答问题的中文聊天机器人。使用者通过命令行界面输入问题，机器人输出回答和参考的 wiki 文档的链接。

核心问题：我们的研究旨在解决 Deepin 操作系统用户在使用过程中可能遇到的问题，主要包括但不限于软件安装、系统设置、故障排除等方面的技术支持和问题解答。

主要挑战：Deepin 用户可能在操作系统使用过程中遇到各种问题，但是获取相关技术支持和解答需要花费大量时间和精力；传统的技术支持方法往往依赖于用户手册或在线搜索，但准确性和实用性无法保障，尤其是对于复杂或个性化的问题。

1.3. 目标与意义

研究目标：基于上述的问题和挑战，我们的研究旨在开发一个智能问答系统，利用 RocketQA 和 ChatGLM 模型，用户可以以更直观、更自然的方式与系统进行交互，获得更及时、准确的问题解答和技术支持。旨在提高用户的满意度，增强用户对 Deepin 系统的信任感和使用意愿。

提高系统的实用性和可用性：用户可以更轻松地解决各种技术问题和疑惑，无需依赖繁琐的手册或在线搜索，从而更好地发挥 Deepin 系统的功能和优势，能够为 Deepin 系统用户提供全面的技术支持和问题解答，提高操作系统的实用性和用户粘性。推动人工智能在操作系统领域的应用：通过结合自然语言处理和深度学习技术，实现智能问答系统的设计和实现，将为操作系统领域的研究和实践带来新的思路和方法，推动该领域的技术创新和进步。促进智能化服务的发展：智能问答系统不仅可以应用于 Deepin 操作系统，还可以推广到其他操作系统或软件平台，为更广泛的用户群体提供智能化服务和技术支持。这将促进智能化服务的发展，推动人机交互方式的创新，提升用户体验和生活品质。

1.4. 技术路线介绍

数据收集与准备：我们拥有 deepin 网站的全部数据，以 markdown 形式保存。

模型选择与调优：在选择模型时，我们一开始选择了 RocketQA 和 MacBERT-Large 这两种模型。为了提高答案质量，我们最终选定了 RocketQA 和 ChatGLM 这两种模型。RocketQA 模型擅长处理问答任务，而 ChatGLM 模型则更适合处理对话生成任务。我们计划结合这两种模型，通过模型融合或迁移学习等方法，实现更准确、更灵活的问答系统。

系统集成与测试：完成模型训练后，我们将进行系统集成和测试工作。这包括将模型嵌入到 Deepin 操作系统中，并设计合适的用户界面和交互方式。在测试阶段，我们将对系统进行全面的功能测试和性能评估，确保其稳定性和可靠性。

2. 文献综述

2.1. 智能问答系统

2.1.1. 问答系统分类

我们可以从知识领域、答案来源等角度来替问答系统做分类。

从知识领域来看，可以分为“封闭领域”以及“开放领域”两类系统。

封闭领域系统专注于回答特定领域的问题，如医药或特定公司等。由于问题领域首先，系统又比较大的发挥空间，可以导入如专属本体论等知识，或将答案来源全部转换成结构性资料，来提升系统的表现。

开放领域系统则希望不设限问题的内容范围，天文地理无所不问，系统中所有知识与元件都必须尽量做到与领域不相关，当然难度也相对地提高。

若根据答案来源来区分，可以分为“数据库问答”、“常问问题问答”、“新闻问题”、“互联网问答”等系统。

2.1.2. 问题类型

问答系统接受自然语言问句，为了有效控制研究变因，多会订定可接受的问题类型来限制研究范围。最基本的类型为“仿真陈述回答”，此类系统根绝答案语料所述资讯，取出一小段字串作为答案。由于答案的正确与否是根据答案语料的内容来决定，在现实生活中不一定为真，故称为仿真陈述回答。有些系统把问答范围进一步缩小，限定在人、地、组织等明确的专有名词上。此类系统有能力回答如“请列举美国历届总统”这种清单型的问句，则称为“清单问答”。若能回答定义问题，则成为“定义回答”，以此类推还能定义出其他类型的问题。

除了这些与问句资讯内容有关的类型外，最近评鉴会议引进如“时间限制问题”与“序列问题”等复杂的问题类型。时间限制型的问题会在问句中明确指出答案的时间范围限制，比如说以“民国九十年时国民党主席是谁”这问句来说，系统必须有根据答案预料结构化资料，或上下文来推论正确答案的能力。序列问题则把问答系统未来的应用定位在互动式的系统上。经过来回多次问答的方式来满足使用者的资讯需求。了解这些问题类型分类，有助于研究范围界定，同时在分析比较上也比较有依据。

2.1.3. 问答系统用例

可以使用问答 QA 模型，通过使用知识库（文档）作为上下文来自动响应常见问题，可以从这些文档中得出客户问题的答案。如果想节省推理时间，可以先使用段落排名模型来查看哪个文档可能包含问题的答案，然后使用 QA 模型迭代该文档。

根据输入和输出，有不同的 QA 变体。

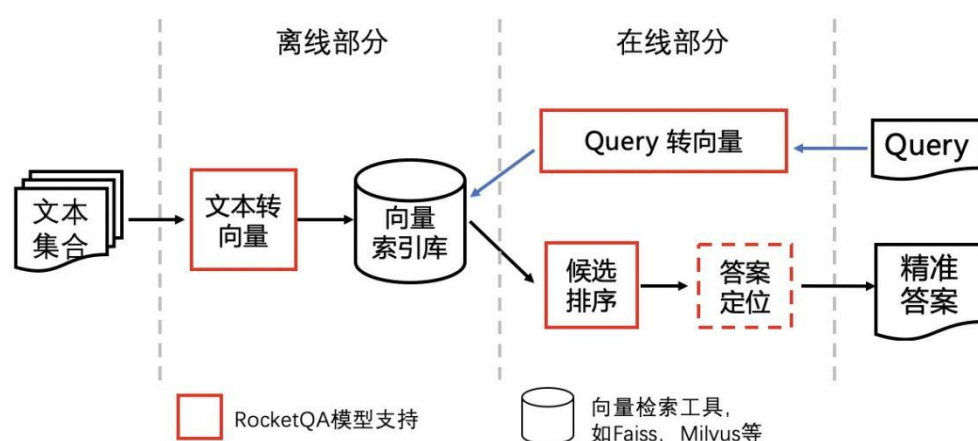
抽取式 QA: 该模型从上下文中提取答案。这里的上下文可以是提供的文本、表格甚至 HTML，这通常用类似 bert 的模型来解决。

开放式生成式 QA: 该模型直接根据上下文生成自由文本。可以在其页面中了解有关“文本生成”任务的更多信息。

封闭式生成 QA: 在这种情况下，不提供上下文。答案完全由模型生成。

2.2. RocketQA 模型

RocketQA 是一个基于 BERT 的深度语义检索模型，其内部包含了两个子模型：双塔式召回模型和交互式排序模型，其中召回模型负责从文档库中快速筛选出与用户查询度高度相关的候选文档集，排序模型则会进一步从候选文档集中找到与用户查询最相关的文档，并返回给用户。RocketQA 通过跨批次负采样、去噪的强负例采样与数据增强三项技术，来提高训练能力。



2.2.1. 对偶式检索模型

传统的基于检索的方法中，系统会首先依据用户的查询从文档库中检索出可能相关的文档，然后从这些文档中提取答案。这种方法的优点在于可以利用成熟的信息检索技术来快速定位潜在答案所在的文档，但缺点是在提取答案时可能受

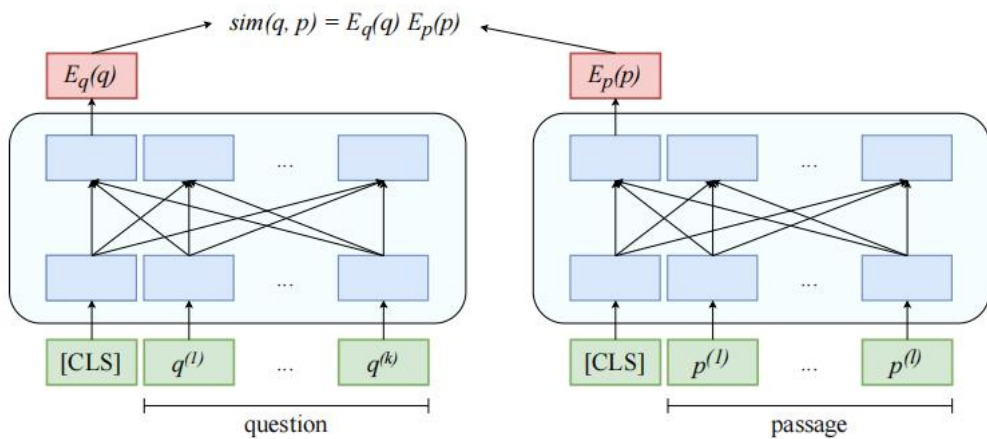
限于传统的模式匹配和关键词匹配方法，无法充分利用文本的语义信息。

基于神经网络的问答模型能够利用深度学习技术从大规模语料库中学习语义信息，以实现更准确的答案提取。这种方法的优点是能够理解文本背后的语义和上下文信息，但缺点是在大规模语料库中进行答案搜索时效率较低。

对偶式检索模型试图结合这两种方法的优点，以实现更好的性能。具体来讲，对偶式检索模型首先使用传统的基于检索的方法从文档库中检索出候选文档，然后利用神经网络模型对这些文档进行进一步筛选和排名，以确定最终的答案。

RocketQA 就是一个利用对偶式检索模型增强训练的问答模型。它采用了一种名为 Query-Document Dual Encoder 的架构，其中包括一个查询编码器和一个文档编码器，分别用于将查询和文档表示为向量。这些向量通过一个评分函数进行比较，以确定查询和文档之间的相关性，并最终提取最佳答案。

2.2.2. 双塔式召回模型



(a) A dual-encoder based on pre-trained LMs.

双塔式召回模型（Dual-Tower Retrieval Model）通常用于从大规模文本库中高效地检索与用户查询相关的文档或段落。这种模型之所以被称为“双塔式”，是因为它包含两个并行的神经网络结构，分别称为查询塔（Query Tower）和文档塔（Document Tower）。

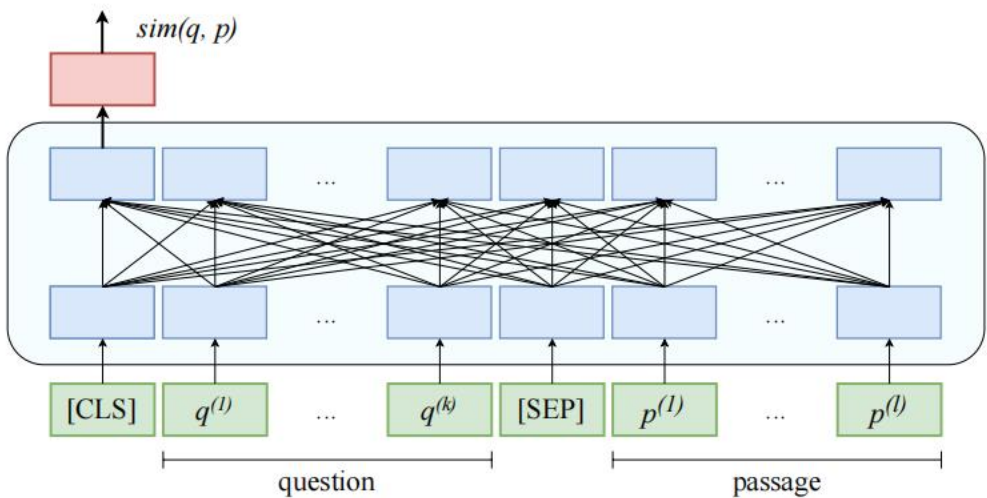
查询塔：这个部分接受用户的查询作为输入，并将其编码为一个向量表示，该向量捕捉了查询的语义信息。通常使用卷积神经网络 CNN、循环神经网络 RNN 或 Transformer 架构来实现。

文档塔：这个部分接受每个文档或段落作为输入，并将其编码为向量表示，捕获了文档或段落的语义信息。与查询塔类似，文档塔通常也使用 CNN、RNN 或 Transformer 结构。

相似度计算：在训练过程中，通过计算查询向量与每个文档向量之间的相似度分数来衡量他们之间的相关性。常用的相似度计算方法包括余弦相似度（Cosine Similarity）或点积等。

训练方式：双塔式召回模型通常采用对比损失（Contrastive Loss）或三元组损失（Triplet Loss）来优化查询和相关文档之间的相似度匹配。

2.2.3. 交互式排序模型



(b) A cross-encoder based on pre-trained LMs.

交互式排序模型用于对召回的文档或段落进一步排序，以确保返回给用户的最终结果是最相关的。这个模型通常在召回模型之后应用，并且可以利用用户的反馈信息进行优化。

特征提取：对于每个召回的文档或段落，交互式排序模型会提取丰富的特征，这些特征可以包括语义信息、文本结构、关键词匹配程度等。

排序器：这个部分接受从召回模型获取的文档或段落，以及用户的查询作为输入，并为每个文档或段落分配一个排名分数，表示其与查询的相关性。

训练方式：交互式排序模型通常使用监督学习方法进行训练，训练数据包括用户查询、召回的文档或段落以及相关性的标签。常用的损失函数包括交叉熵损失（Cross-Entropy Loss）或者排序损失（Ranking Loss）。

反馈机制：在某些情况下，交互式排序模型可能会利用用户的反馈信息进行优化，例如点击率（Click-Through Rate, CTR）或转化率（Conversion Rate）等，进一步提高排序质量。

2.3. MacBERT 模型

Masked Language Modeling (MLM) 在 MacBERT 模型中的应用，主要是一种预训练策略，用于提升模型在自然语言处理任务中的性能。

2.3.1. MLM 的基本原理

MLM 的主要目标是预测句子中被“mask”（隐藏）的词的潜在值。在训练过程中，模型会在输入句子中随机挑选一些词，并用特殊的“[MASK]”标记来替换它们。然后，模型会尝试预测这些被“mask”的词的原始值。

2.3.2. MacBERT 中的 MLM 策略

MacBERT 在 MLM 策略上进行了优化，具体体现在以下几个方面：

不使用固定的[MASK]标记：与 BERT 不同，MacBERT 在掩码阶段没有使用在微调阶段从未出现的[MASK]标记进行屏蔽，而是使用了与被屏蔽单词相似的单词进行替换。这种策略有助于模型更好地理解上下文信息，提高预测的准确性。

全字掩码与 Ngram 掩码策略：MacBERT 在 MLM 任务中采用了全字掩码以及 Ngram 掩码策略来选择掩码候选标记。具体来说，字级一元到四元的比例为 40%、30%、20%、10%，这种策略有助于模型捕获更丰富的语言结构信息。

相似词替换：MacBERT 使用基于 word2vec 相似度计算的 Synonyms 工具包获得相似词，如果被选中的 N-gram 需要被屏蔽，模型会找到相似的单词进行替换。在极少数情况下，如果没有相似的单词，模型会降级使用随机单词替换。

2.3.3. MLM 在 MacBERT 中的作用

MLM 作为 MacBERT 的一个核心预训练策略，通过预测被掩码的单词，使模型能够学习到丰富的语言知识和上下文信息。这种预训练方式有助于模型在后续的微调阶段更好地适应各种自然语言处理任务，如文本分类、命名实体识别、问答系统等。

2.3.4. MacBERT 中 MLM 的训练比例

在 MacBERT 的训练过程中，大约 15%的输入单词被用于掩码。其中，80%的掩码单词被替换为相似单词，10%被替换为随机单词，剩余的 10%保留原始单

词。这种设置使得模型在训练过程中能够接触到多样化的输入，从而提高其泛化能力。

2.4. ChatGLM 模型

ChatGLM-6B 是由清华团队开发的开源大语言模型，可以用中英文进行问答对话，采用了 General Language Model (GLM) 架构，并且通过模型量化技术，可以在普通的显卡上运行（需要 6GB 显存）。为了优化中文问答和对话，ChatGLM-6B 结合了监督微调、反馈自助和人类反馈强化学习等技术。ChatGLM 是一种生成式语言模型，用于聊天和对话任务。它是基于 OpenAI 的 GPT 模型框架构建的，采用了大规模的预训练数据集来学习语言模式和生成文本的能力。ChatGLM 可以理解上下文并生成连贯、自然的回复。它可以用于构建对话系统、智能客服、聊天机器人等应用，能够提供更加交互性和人性化的对话体验。

2.4.1. GLM 架构

GLM (General Language Model) 架构是一种针对自然语言处理任务而设计的预训练模型架构，是一种生成式语言模型，其核心目标是生成自然语言文本，以响应特定的输入或提示。GLM 架构基于 Transformer 模型，特别是其 Decoder 部分，通过引入自回归空白填充的预训练目标，旨在解决自然语言理解、无条件生成和有条件生成等多个任务。GLM 模型在预训练过程中，通过掩码 (mask) 部分文本序列，并要求模型恢复这些被掩码的部分，从而学习到丰富的上下文信息。

自回归空白填充：GLM 通过自回归的方式对 mask 的词汇进行预测。为了能够让模型理解到上下文信息，GLM 模型将被掩码的片段的顺序打乱，这样能够让模型充分捕捉到不同片段之间的相互依赖关系。**二维位置编码：**GLM 通过添加二维位置编码来增强对文本序列中位置信息的理解。这有助于模型更好地捕捉文本中的序列和层次结构。**掩码策略：**GLM 在预训练过程中，采用两种掩码策略：**Document-level** 和 **sentence-level**。在 **Document-level** 策略中，整个文档的 50%~100% 的连续 token 被 mask；而在 **sentence-level** 策略中，mask 完整的句子，直到 mask 大约 15% 的 token。

预训练目标：GLM 的预训练目标是要求模型从含有 masked token 的文本中预测被 mask 的片段。这通过自回归的方式实现，即模型根据文本中的上文信息来预测下一个被 mask 的 token。在预训练过程中，生成的 masked token 先后顺序是随机的，同时每一个 masked span 都有两个特殊标记[START]和[END]。

模型结构：**Decoder-only Transformer：**GLM 模型基于 Decoder-only 的 Transformer 架构，这意味着模型只包含 Transformer 的解码器部分。解码器使用

单向注意力机制，从左到右地处理文本序列。Layer Normalization 和 Residual Connection: GLM 在模型架构中更换了 Layer Normalization 和 residual connection 的顺序，以提高模型的训练效率和性能。激活函数和 Position Embedding: GLM 使用 GeLUs 作为激活函数，并定义了两个不同的绝对位置嵌入（position embedding）来捕捉文本中的位置信息。

2.4.2. 监督微调（Supervised Fine-Tuning）

监督微调是一种使用带有标签的数据集对预训练模型进行进一步训练的技术。在 ChatGLM 中，这意味着使用标注的对话语料库来训练模型，使其更好地理解 and 生成对话。通过监督微调，ChatGLM 能够学习到特定任务（如对话生成）的特定模式，从而提高其在该任务上的性能。使用大量带有正确答案或标记的对话数据；通过对模型参数进行微调，使其适应特定任务；可以显著提高模型在特定任务上的性能。

2.4.3. 反馈自助（Self-Supervised Learning）

反馈自助是一种利用模型自身生成的输出来改进模型的技术。通过反馈自助，ChatGLM 可以不断地从自己的输出中学习，依赖于模型自身的输出作为新的训练数据或反馈，从而逐步提高其性能。反馈自助技术通常设计自动生成任务，例如使用自动编码器或生成对抗网络（GANs）。在聊天生成模型种，反馈自助技术可以帮助模型学习对话的结构和语义，从而提高生成文本的质量和多样性。

2.4.4. 人类反馈强化学习技术（Human Feedback Reinforcement Learning）

人类反馈强化学习是一种结合人类反馈和强化学习来优化模型的技术。在 ChatGLM 中，这意味着使用人类提供的反馈（如对话的质量、正确性、有用性等）作为奖励信号，通过强化学习算法来优化模型的参数。通过人类反馈强化学习，ChatGLM 可以更好地捕捉和理解人类的偏好和需求，从而生成更符合人类期望的对话。

3. 系统设计与架构

3.1. 整体架构

为了完成根据用户输入，给出“参考链接+答案”格式回答的任务，我们采取了一下步骤和方法：

1、数据准备

首先，我们拥有 Deepin 网站的全部数据，这些数据是以 markdown 格式保存的网页内容，并且为了处理参考链接部分，我们找到了该网站的 sitemap 文件。我们选择 RocketQA 和 ChatGLM 模型来构建问答系统，因为 RocketQA 在文档搜索和排序方面表现出色，而 ChatGLM 在生成自然语言回答方面有着优秀的性能。

2、数据处理与索引构建

我们利用 RocketQA 的双塔模型对 Deepin 网站的 markdown 文档进行编码，将文档内容转化为向量表示。这是为了后续能够高效地根据用户查询在大量文档中进行搜索。我们选择 Faiss 库来构建文档的向量索引，因为它提供了高效的近邻搜索算法，能够快速找到与用户查询最相关的文档。

3、问答服务应用程序开发

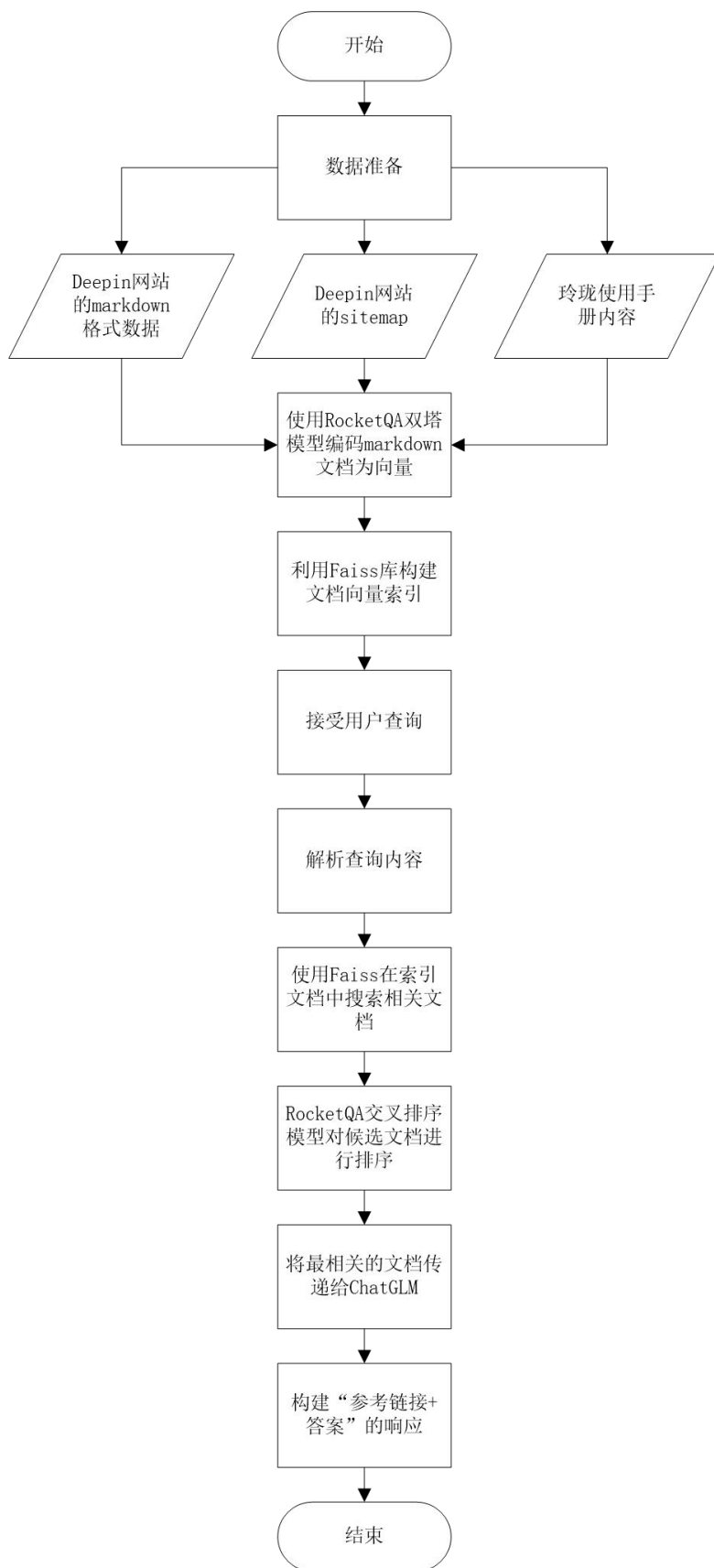
基于 Tornado 框架，我们开发了一个问答服务应用程序。选择 Tornado 是因为它是一个异步的 web 框架，能够处理高并发的 HTTP 请求。我们的应用程序接受用户提交的查询，解析查询内容，并使用 Faiss 在文档索引中搜索最相关的文档。然后，我们使用 RocketQA 的交叉排序模型对候选文档进行二次排序，以确保选取的文档最符合用户的问题。

4、答案生成

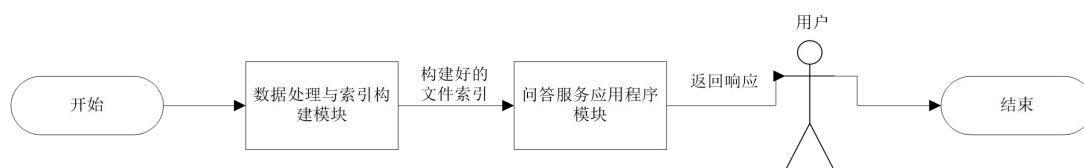
得到最相关的文档后，我们将其传递给 ChatGLM 模型。ChatGLM 模型是一个基于 Transformer 的生成式模型，能够生成自然语言回答。我们将文档作为上下文信息，结合用户查询，让 ChatGLM 生成一个针对于该问题的回答。

5、响应构建与返回

最后，我们构建包含“参考链接+答案”的响应，并通过 HTTP 接口返回给用户。



3.2. 模块设计

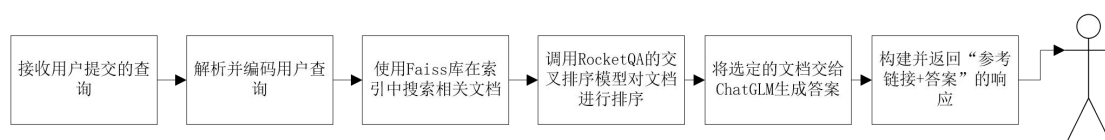


3.2.1. 数据处理与索引构建模块



- **输入：**Deepin 网站的 markdown 格式的数据文件和 sitemap 文件
- **处理：**数据解析：解析 sitemap 和 markdown 文件，提取出网页链接和对应的内容
- **文档编码：**使用 RocketQA 模型对文档内容进行编码，生成文档向量
- **构建索引：**利用 Faiss 库对文档向量进行索引，创建高效检索的索引结构
- **输出：**构建好的文件索引

3.2.2. 问答服务应用程序模块



技术栈：基于 Tornado 框架

功能：

- **HTTP 接口：**提供 HTTP 接口，接受用户提问的 POST 请求
- **请求处理：**解析请求中的用户输入，对输入进行编码
- **搜索与排序：**使用 Faiss 库在文档索引中搜索最相关的文档，调用 RocketQA 的交叉排序模型对候选文档二次排序，选取最符合问题的文档
- **答案生成：**将选定的文档传递给 ChatGLM 模型，生成回答
- **响应构建：**构建包含“参考链接+答案”格式的最终响应，返回给用户

3.2.3. 模型应用模块

RocketQA:

双塔模型：用于文档和查询的初步匹配，生成文档向量和查询向量

交叉排序模型：对初步匹配的文档进行二次排序，确保最相关的文档被选中

ChatGLM:

生成式模型：接收 RocketQA 选定的文档作为上下文，生成针对用户问题的自然语言回答

3.3. 核心算法

3.3.1. RocketQA 模型在系统中的应用

1、双塔模型（Dual-Encoder）

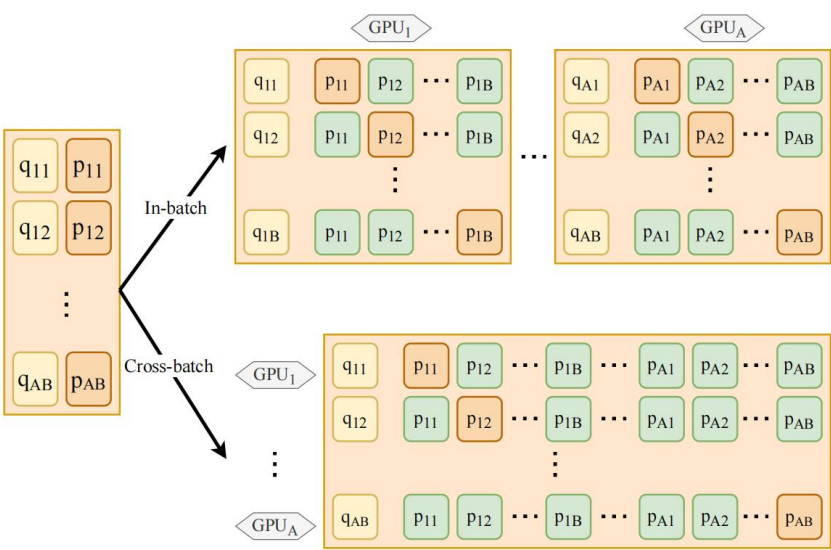
作用：用于学习问题和段落的向量化表示，以进行语义匹配。

特点：将问题和段落分别通过两个独立的编码器进行编码，得到各自的向量表示。这些向量表示可以被用于计算问题和段落之间的相似度，从而进行匹配。

2、跨批次负采样（Cross-BatchNegatives）

目的：提高模型的区分能力，确保模型能够准确地区分相关和不相关的文档。

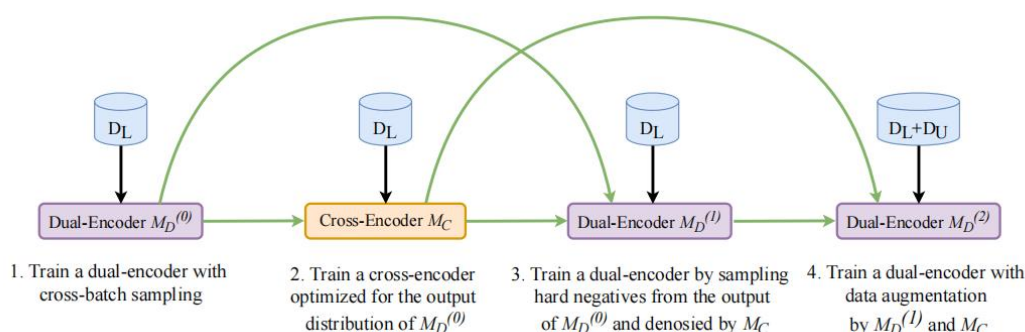
方法：在训练过程中，不仅使用当前批次中的负样本（不相关的文档），还使用之前批次中的负样本来作为负例。这有助于模型更好地学习到文档之间的区分性特征。



3、去噪的强负例采样（Denoised Hard Negatives Sampling）

目的：进一步提高模型的区分能力，特别是对于那些与问题非常相似但实际上并不相关的文档。

方法：从候选文档中选取那些与问题在语义上非常接近但实际上并不相关的文档作为“强负例”。对这些强负例进行去噪处理，以减少其中可能存在的误标注或噪声。使用去噪后的强负例来训练模型，使其能够更准确地区分相关和不相关的文档。



4、数据增强（Data Augmentation）

目的：增强模型的泛化能力，使其能够处理更多样化的查询和文档。

方法：对原始的训练数据进行拓展和变换，生成更多的训练样本。这些扩展和变换可以包括同义词替换、随机插入、删除或替换单词等。通过使用增强后的数据来训练模型，可以提高其泛化能力和鲁棒性。

5、交叉排序模型（Cross-Encoder）

作用：在初步匹配的基础上，对候选文档进行二次排序，以进一步提高匹配的准确性。

特点：交叉排序模型将问题和段落一起输入到一个编码器中进行编码，得到一个联合的向量表示。这个联合的向量表示可以更好地捕捉到问题和段落之间的语义关系，从而进行更准确的排序。

3.3.2. MacBERT 模型在系统中的应用

Transformer 架构：MacBERT 模型的核心是 Transformer 架构，它利用自注意力机制来捕捉输入文本中的上下文信息。Transformer 架构中的双向编码器使得模型能够同时考虑输入文本的前后文信息。

Masked Language Modeling（MLM）：在预训练阶段，MacBERT 模型使用 MLM 策略进行训练。具体而言，模型会随机遮盖输入文本中的一部分单词，并

尝试预测这些被遮盖的单词。这种训练方式有助于模型学习到丰富的语言知识和上下文信息。

MacBERT 模型在项目中的具体应用：文档筛选后的处理：首先，RocketQA 交叉排序模型从大量候选文档中选出与用户查询相关的文档。然后，这些筛选出的文档将作为 MacBERT 模型的输入。生成回答：MacBERT 模型利用其在预训练阶段学习到的语言知识和上下文信息，对输入的文档进行深入理解和分析。接着，模型根据用户查询的意图和上下文，生成与查询相关的回答。

3.3.3. ChatGLM 模型在系统中的应用

1、基于 Transformer 架构

ChatGLM 采用了 Transformer 架构，这是一种基于自注意力机制的深度学习模型，广泛应用于自然语言处理任务中。Transformer 通过多头自注意力机制和位置编码，能够捕捉输入序列中的长距离依赖关系，从而生成高质量的文本输出。

2、自注意力机制

在 Transformer 架构中，自注意力机制是核心组件之一。它通过计算输入序列中不同位置之间的相关分数，来确定在生成输出时应该关注哪些位置的信息。这种机制使得 ChatGLM 能够理解和生成复杂的自然语言文本。

3、旋转位置嵌入（Rotary Position Embedding）

ChatGLM 使用旋转位置嵌入来编码文本中的位置信息。与传统的绝对位置编码相比，旋转位置嵌入能够更好地处理长序列中的位置信息，并且不会随着序列长度的增加而累积误差。这有助于 ChatGLM 在生成文本时保持正确的位置顺序和语义结构。

4、GELU 激活函数

ChatGLM 使用 GELU（Gaussian Error Linear Unit）作为激活函数。GELU 结合了 ReLU 和 Sigmoid 的优点，具有更好的非线性特性和更平滑的梯度。这有助于 ChatGLM 在训练过程中更好地拟合复杂的函数关系，并提高模型的泛化能力。

4. 项目实施

4.1. 数据集描述

数据集来源：Deepin 系统的数据是公开数据（[GitHub - linuxdeepin/wiki.deepin.org: wiki content](https://github.com/linuxdeepin/wiki.deepin.org/wiki/content)）；Deepin 网站的 sitemap 文件（wiki.deepin.org/sitemap.xml）

数据集内容：是以 markdown 格式进行保存。本数据集以文件夹的形式进行组织，整体结构呈现出一种树形结构。数据集根目录对应着网站的首页目录，而根目录下的各个子文件夹则对应着网站上不同的分类。

4.2. 项目配置

4.2.1. 环境配置

操作系统：Linux（Deepin 20.9）、Windows（Windows 11）

Python 版本：Python 3.8。

依赖库：numpy 1.24.3; faiss-cpu 1.8.0; rocketqa 1.1.0 ; Markdown 3.6; tornado 6.3.3; transformers 4.33.2; requests 2.31.0; ppaddlepaddle 2.4.2; scipy 1.10.1; sentencepiece 0.2.0; torch 2.2.2（其中 python 的版本必须 3.6+，paddlepaddle 必须 2.0+）

4.2.2. 代码配置

代码编辑器/IDE：PyCharm

代码风格：PEP 8

4.2.3. 数据配置

数据源：Deepin 公开数据集([GitHub - linuxdeepin/wiki.deepin.org: wiki content](https://github.com/linuxdeepin/wiki.deepin.org/wiki/content))、Deepin 网站的 sitemap 文件（wiki.deepin.org/sitemap.xml）

数据格式：JSON

4.2.4. 模型配置

模型选择：RocketQA 模型、MacBERT 模型、ChatGLM 模型

训练策略：

- 批量大小（Batch Size）：512*8
- 优化器：ADAM
- 学习率：双编码器：学习率设置为 3e-5，线性预热（Linear Warmup）的比例设置为 0.1。交叉编码器：学习率设置为 1e-5。
- 序列长度：问题的最大长度：32
- 段落的最大长度：128
- 自动混合精度（Automatic Mixed Precision）：RocketQA 使用了自动混合精度训练，可以在不牺牲模型精度的前提下，减少 GPU 显存的使用，并提高训练速度。
- 梯度检查点（Gradient Checkpointing）：RocketQA 还使用了梯度检查点技术，它允许模型在有限的资源下使用更大的批量大小进行训练。通过在反向传播过程中保存某些中间结果，并在需要时重新计算它们，梯度检查点可以显著减少显存的使用。

4.3. 数据预处理流程

RocketQA 模型的数据集格式：【标题\t 内容】。

匹配模型的数据集格式，加上题目要求，我们的数据集格式为【参考链接\t 答案】。

1、数据格式转换

原始数据集为 wiki.deepin.org 网站的 sitemap，其格式为 HTML。我们首先将 HTML 格式中的冗余信息去除，整理出所有网页的 URL。由于数据集中文件的相对路径与网页 URL 之间存在关联，我们根据基础链接 <https://wiki.deepin.org/zh/>，结合当前文件的相对路径，拼接得到每个文件对应的网页 URL。

2、数据分块

在获得网页内容后，我们按照 markdown 中的标题格式“#”对网页进行分块。每个标题下的内容被视为一个独立的块，并进行相应的保存。同时，我们对代码块等特殊内容进行了特殊处理，确保它们不会对后续的数据处理产生影响。

3、建立索引

在数据分块的基础上，我们将每个块的内容与其对应的标题进行索引建立。

4、数据存储

为了方便后续的数据处理和模型训练，我们将网页的正文内容和链接内容分开保存。正文内容存储在 `para_list` 数组中，每个元素为一个标题下的内容块；链接内容（即 URL）存储在 `title_list` 数组中，与 `para_list` 中的元素一一对应。

4.3.1. 数据集扩展：玲珑使用手册内容

1. 数据收集

我们使用 `MarkDownload` 这个插件，直接对玲珑使用手册的数据进行了下载。下载后，我们获得了玲珑使用手册的原始 `md` 格式文件。

2. 数据清洗与调整

在数据收集的过程中，我们发现了一些格式或内容上的差错。为了确保数据的质量，我们进行了手动调整。由于玲珑使用手册的数据量相对较小，整个调整过程并没有花费过多的时间。通过仔细检查和修正，我们确保了数据的准确性和一致性。

3. 数据格式调整

为了使玲珑使用手册的数据与之前的 `deepin` 数据格式保持一致，我们进行了相应的格式调整。主要步骤包括：

修改文件名：将玲珑数据的文件名按照 `deepin` 数据的命名规则进行修改，以便后续根据文件的相对路径拼接为该页面的网址。

网址拼接：根据玲珑使用手册的基础链接（例如：<https://linglong.dev/guide/>），结合文件的相对路径，我们拼接得到了每个文件对应的网页 URL。

4. 数据集成与测试

在完成格式调整后，我们将玲珑使用手册的数据直接添加到项目中。为了确保数据的正确性和可用性，我们进行了测试。测试结果表明，通过新添加的玲珑使用手册数据，模型能够正确地回答关于玲珑使用的问题，且未发现明显的数据错误或格式问题。

4.4. 模型调整：从 MacBERT-Large 到 ChatGLM 的转变

4.4.1. MacBERT-Large 与 ChatGLM 的比较

MacBERT-large: 这是一个基于 BERT 架构的预训练模型，适用于各种 NLP 任务，包括文本分类、命名实体识别等。然而，在对话生成方面，其表现可能不如专门为此设计的模型 ChatGLM。

ChatGLM: ChatGLM 是一个基于 GLM 架构的生成式预训练模型，特别优化了对话生成任务。它采用了更大的模型和更多的数据进行训练，因此在对话生成方面表现出更高的性能和更自然的输出。

4.4.2. 转变的必要性

性能提升: ChatGLM 模型在对话生成方面的性能优于 MacBERT-large，能够生成更自然、更准确的回答。

用户体验: 使用 ChatGLM 模型可以为用户提供更流畅、更智能的对话体验，提高用户满意度。

趋势与未来: 随着 NLP 技术的不断发展，生成式模型在对话系统中的应用越来越广泛。转向 ChatGLM 模型有助于我们跟上这一趋势，并为未来的技术升级做好准备。

4.4.3. 技术挑战与解决方案

1、遇到的问题：

在将 ChatGLM 模型集成到我们的系统中后，我们发现在现有的设备上，每次用户询问问题后，系统需要等待 15-30 分钟才能生成并返回一条回答。这种延迟严重影响了用户体验，使得系统无法满足实时对话的要求。

我的资源信息如下：

```
名称: AMD Radeon(TM) Graphics
制造商: Advanced Micro Devices, Inc.
芯片类型: AMD Radeon Graphics Processor (0x1636)
DAC 类型: Internal DAC(400MHz)
设备类型: 完整显示设备
总内存约为: 8367 MB
显示内存(VRAM): 496 MB
```

2、问题分析：

经过初步调查，我们发现这种延迟主要是由于计算资源不足导致的。ChatGLM 模型是一个复杂的深度学习模型，需要大量的计算资源来进行推理。在资源受限的设备上，模型的推理速度会受到严重影响，导致用户需要等待很长时间才能获得回答。

3、解决方案：

借用服务器资源：我们向老师寻求帮助，借用了—个性能更高的服务器。在这个服务器上，我们重新部署了我们的项目，并进行了测试。

优化部署配置：在服务器上，我们优化了模型的部署配置，确保模型能够充分利用服务器的计算资源。我们调整了模型的批处理大小、并行计算设置等参数，以提高推理速度。

测试与验证：在服务器上部署完成后，我们进行了多次测试，以确保系统能够在 1 分钟之内生成并返回—条回答。测试结果表明，我们的系统现在能够满足实时对话的要求，用户体验得到了显著提升。

相信有性能更高的服务器，我们会做到实时对话。

4.5. 在 Deepin 系统上的实现

为了使我们的项目更加完整，提高 Deepin 系统用户的使用体验，增强其对操作系统的满意度，我们不仅使用 Deepin 的数据集，最终在 Deepin 系统上实现该项目。

安全性增强：Linux 操作系统通常被认为比 Windows 更安全，因为它们更少受到病毒和恶意软件的攻击。在 Deepin 上运行我们的项目，您可以利用 Linux 的安全特性来保护用户数据和系统安全。

学习机会：将项目从—个操作系统迁移到另—个操作系统是一个很好的学习机会。我们可以深入了解两个操作系统的架构、特性和差异，以及如何在不同的环境中进行开发和测试。

4.5.1. 环境搭建

这一部分可以参考我们的博客，里边有非常详细的记录。

以下是我们的实现步骤：

1、在 Deepin 官网上下载最新版的镜像文件

- 2、下载 VMWare
- 3、在 VMWare 安装 Deepin
- 4、在 Deepin 系统中安装 Anaconda
- 5、安装 pycharm
- 6、配置项目环境

4.6. 各个模块的核心代码

4.6.1. 数据处理与构建索引模块

构造文档对应网页链接：

```
def list_files(directory, pre_of_website):
    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith('.md'):
                file_path = os.path.join(root, file)
                relative_path = os.path.relpath(file_path, directory) # 获取相对路
径

                # 移除文件扩展名
                relative_path, file_extension = os.path.splitext(relative_path)
                # 将反斜杠替换为正斜杠
                relative_path = relative_path.replace('\\', '/')
                # 打开文件进行读取
                with open(file_path, 'r', encoding='utf-8') as file:

                    content = file.read() # 读取文件内容
                    paragraphs = spilt_content(content) # 分割文本

                    for p in paragraphs:
                        title_list.append(pre_of_website + relative_path)
                        para_list.append(p)

                    print(f"文件路径: {pre_of_website + relative_path}")
                    print(f"正文: {p}")

print('-----')
```

分割文档内容：

```
def spilt_content(content):

    # 初始化一个列表来保存分割后的内容
    result = []
    # 初始化一个变量来保存当前正在处理的内容
    current = []

    # 遍历每一行
    for line in content.splitlines():

        # 检查当前行是否是标题或分隔符
        if line.startswith('#') or line.startswith('---'):
            # 如果当前列表不为空，意味着我们已经收集了一些内容
            if current:
                # 将当前收集的内容添加到分割内容列表中
                result.append('\n'.join(current))
                # 重置当前内容列表
                current = []
            # 添加二级标题到当前内容列表
            current.append(line)
        else:
            # 处理代码块语法
            if line.startswith('```'):
                line = '---'
            # 将非标题行添加到当前内容列表
            current.append(line)

    # 检查是否有剩余的内容需要添加到分割内容列表中
    if current:
        result.append('\n'.join(current))

    for i in range(len(result)):
        result[i] = markdown.markdown(result[i])          # 将 markdown 转成 html
        result[i] = re.sub('<[^\>]+\>', '', result[i])    # 去除 html 标签

    filter_result = [p.strip() for p in result if p.strip()] # 过滤并处理段落列表

    return filter_result
```

建立索引：

```
def build_index(encoder_conf, index_file_name, title_list, para_list):

    dual_encoder = rocketqa.load_model(**encoder_conf)
    para_embs = dual_encoder.encode_para(para=para_list, title=title_list)
    para_embs = np.array(list(para_embs))

    print("Building index with Faiss...")
    indexer = faiss.IndexFlatIP(768)
    indexer.add(para_embs.astype('float32'))
    faiss.write_index(indexer, index_file_name)
```

4.6.2. 问答服务应用程序模块

处理 HTTP 请求：

```
def post(self):
    input_request = self.request.body
    output = {}
    output['error_code'] = 0
    output['error_message'] = ''
    output['answer'] = []
    if input_request is None:
        output['error_code'] = 1
        output['error_message'] = "Input is empty"
        self.write(json.dumps(output))
        return

    try:
        input_data = json.loads(input_request)
    except:
        output['error_code'] = 2
        output['error_message'] = "Load input request error"
        self.write(json.dumps(output))
        return

    if "query" not in input_data:
        output['error_code'] = 3
        output['error_message'] = "[Query] is missing"
        self.write(json.dumps(output))
        return
```

使用 Faiss 库在索引中搜索相关文档：

```
class FaissTool():
    """
    Faiss index tools
    """

    def __init__(self, directory, index_filename):
        self.engine = faiss.read_index(index_filename)
        self.id2text = []

        for root, dirs, files in os.walk(directory):
            for file in files:
                if file.endswith('.md'):
                    file_path = os.path.join(root, file)
                    relative_path = os.path.relpath(file_path, directory)
                    # 移除文件扩展名
                    relative_path, file_extension = os.path.splitext(relative_path)
                    # 将反斜杠替换为正斜杠
                    relative_path = relative_path.replace('\\', '/')
                    # 打开文件进行读取
                    with open(file_path, 'r', encoding='utf-8') as file:
                        content = file.read() # 读取文件内容
                        paragraphs = spilt_content(content)

                        for p in paragraphs:
                            self.id2text.append(('https://wiki.deepin.org/zh/' +
relative_path + str_split + p).strip())
def search(self, q_embs, topk=5):
    res_dist, res_pid = self.engine.search(q_embs, topk)
    result_list = []
    for i in range(topk):
        result_list.append(self.id2text[res_pid[0][i]])
    return result_list
```

调用 RocketQA 的交叉排序模型对文档进行排序：

```
search_result = self._faiss_tool.search(q_embs, topk)

titles = []
paras = []
queries = []
for t_p in search_result:
    queries.append(query)
    t, p = t_p.split(str_split)
    titles.append(t)
```

```

        paras.append(p)
ranking_score = self._cross_encoder.matching(query=queries, para=paras, title=titles)
ranking_score = list(ranking_score)

```

将选定的文档传递给 ChatGLM 生成答案:

```

for i in range(len(paras)):
    final_result[query + str_split + titles[i] + str_split + paras[i]] = ranking_score[i]
sort_res = sorted(final_result.items(), key=lambda a:a[1], reverse=True)

for qtp, score in sort_res:
    one_answer = {}
    one_answer['probability'] = score
    q, t, p = qtp.split(str_split)
    one_answer['title'] = t

    response, history = model.chat(tokenizer, prompt[0] + p + prompt[1] + q + prompt[2],
history=[])

    one_answer['para'] = response
    output['answer'].append(one_answer)

```

构建答案响应:

```

result = requests.post(SERVICE_ADD, json=input_data)
res_json = json.loads(result.text)

print("QUERY:\t" + query)
for i in range(TOPK):
    title = res_json['answer'][i]['title']
    para = res_json['answer'][i]['para']
    score = res_json['answer'][i]['probability']

print('-----')
print('-----')

print('\n 结果' + str(i + 1) + ': \n')
print('参考链接: \t' + title + '\n')
print('答案: ')
print(para + '\n')

```

5. GUI 界面设计与实现

5.1. 设计思路

思路：创建一个名为“RocketQA 问答系统”的图形界面（GUI）。允许用户输入问题并查询答案。当用户输入问题，点击查询后，输入框可以自动清除其中内容。结果显示区可以保存用户的查询历史记录，用户可以通过滚动条来进行翻阅查看。

结构：包括初始化界面、创建组件、发送查询、显示结果。

5.2. 组件使用

Tkinter 库：用于创建 GUI 的窗口和组件，如标签 Label、输入框 Entry、按钮 Button 和滚动文本框 ScrolledText。

Requestes 库：用于发送 HTTP 请求到后端服务。

Json 库：用于解析从后端服务返回的 JSON 格式的数据。

5.3. 事件处理

当用户点击“查询”按钮时，触发 submit_query 函数。该函数首先获取用户输入的查询内容，然后调用 send_query 函数向后端发送查询请求，并接收返回的 JSON 结果。最后，调用 display_results 函数显示查询结果。

5.4. 布局管理

pack 布局：使用 pack 方法管理组件的布局，通过 pady 参数设置组件之间的垂直间距。

组件顺序：按照标签、输入框、按钮、错误标签和滚动文本框的顺序排列组件。

5.5. 用户交互

输入框：用户可以在输入框中输入问题。

按钮：用户点击“查询”按钮触发查询操作。

滚动文本框：用于显示查询结果，用户可以滚动查看。

5.6. 错误处理和验证

查询内容验证：在 `submit_query` 方法中，检查输入框是否为空。如果为空，则显示错误消息“请输入查询内容”。

异常处理：在发送查询请求时，使用 `try-except` 块捕获可能出现的异常，并在错误标签中显示错误信息。

5.7. 页面实现



5.8. 核心代码

```
class RocketQA_GUI:
    def __init__(self, root):
        self.root = root
        self.root.title("RocketQA 问答系统")
        self.root.geometry("800x600") # 调整窗口大小

        self.history = [] # 存储历史记录
```

```

self.create_widgets()

def create_widgets(self):
    self.query_label = tk.Label(self.root, text="请输入您的问题:")
    self.query_label.pack(pady=10)

    self.query_entry = tk.Entry(self.root, width=70) # 调整输入框大小
    self.query_entry.pack(pady=5)

    self.submit_button = tk.Button(self.root, text="查询",
command=self.submit_query)
    self.submit_button.pack(pady=5)

    self.error_label = tk.Label(self.root, text="", fg="red")
    self.error_label.pack(pady=5)

    self.result_text = scrolledtext.ScrolledText(self.root, wrap=tk.WORD,
width=100, height=20) # 调整结果显示区域大小
    self.result_text.pack(pady=10)

def send_query(self, query):
    input_data = {'query': query, 'topk': TOPK}
    result = requests.post(SERVICE_ADD, json=input_data)
    return json.loads(result.text)

def display_results(self, query, res_json):
    result_str = "您的查询: " + query + "\n"
    for i in range(TOPK):
        title = res_json['answer'][i]['title']
        para = res_json['answer'][i]['para']
        result_str +=
'-----\n'

        result_str += '结果 ' + str(i + 1) + ': \n'
        result_str += '参考链接: ' + title + '\n'
        result_str += '答案: ' + para + '\n\n'
        result_str +=
'-----\n\n'

    self.result_text.insert(tk.END, result_str)
    self.history.append(result_str) # 将结果添加到历史记录中

def submit_query(self):
    query = self.query_entry.get()
    if query:

```



```
        res_json = self.send_query(query)
        self.display_results(query, res_json)
        self.query_entry.delete(0, tk.END) # 清除输入框内容
    else:
        self.error_label.config(text="请输入查询内容！")

if __name__ == "__main__":
    root = tk.Tk()
    app = RocketQA_GUI(root)
    root.mainloop()
```

6. Web 界面设计与实现

6.1. Flask 技术的选择原因

- 轻量级和灵活性：Flask 是一个微框架，它只包含实现一个 Web 应用程序所需的最基本的功能，可以根据自己的需要快速构建和扩展应用程序，而不需要处理不必要的复杂性和重量。
- 易于学习和使用：Flask 的 API 设计得非常直观和易于理解，能快速上手并开始构建 Web 应用程序。
- 强大的模板引擎：Flask 默认使用 Jinja2 作为模板引擎，它提供了强大的模板语言功能，能够轻松地生成动态的 HTML 页面。Jinja2 还支持变量替换、循环、条件语句等高级功能，使得模板的编写更加灵活和强大。
- 适合小型到中型项目：虽然 Flask 是一个微框架，但它仍然适用于构建小型到中型规模的 Web 应用程序。对于许多项目来说，Flask 的轻量级和灵活性使其成为一个理想的选择。

6.2. 规划与设计

我们规划了一个简洁易用的 web 界面，包括一个输入框和一个结果显示框。输入框用于接收用户输入，结果显示框用于展示机器人的响应和用户的查询记录。用户的输入框和系统的结果显示框都有滚动条可以翻阅查看所输入的长信息和历史查询记录。页面左侧由关于 Deepin 系统的常用信息。

1、需求分析

功能需求：用户提问、机器人回答、查询记录保存等

2、系统架构设计

前端界面：使用 HTML、CSS 和 JavaScript 构建用户交互界面，包括输入框、结果显示框等元素。

后端逻辑：使用 Flask 框架编写处理用户请求和返回响应的逻辑代码。这包括接收用户输入、调用问答机器人的 API、保存查询记录等功能。

3、界面设计

布局设计：确定 web 界面的整体布局，包括输入框、结果显示框等元素的位置和大小。布局应简洁明了，方便用户操作。

样式设计：使用 CSS 为界面元素添加样式，如字体、颜色、边框等。

交互设计：定义用户与界面之间的交互方式，如点击按钮提交问题、滚动条

查看结果等。交互设计应自然流畅，提高用户体验。

4、功能模块设计

根据需求分析的结果，设计具体的功能模块：

用户提问模块：实现用户通过输入框提问的功能。当用户输入问题后，后端应接收请求并调用问答机器人的 API 获取回答。

机器人回答模块：实现机器人根据用户问题返回回答的功能。后端应接收问答机器人的响应并展示在结果显示框中。

6.3. Flask 技术应用

1、设置 Flask 环境

```
pip install Flask
```

创建一个新的 Flask web 应用程序实例

```
app = Flask(__name__)
```

2、定义路由和视图函数

在 Flask 中，路由决定了应用程序如何响应 URL 请求。

```
@app.route('/')
def index():
    return render_template('index.html')

@app.route('/query', methods=['POST'])
def query():
    query = request.form['query']
    if query.strip() == '':
        return render_template('index.html', error_message="Please
input a valid query.")

    answers = get_answers(query)

    conversation = [{'text': query, 'type': 'user'}]
    for answer in answers:
        conversation.append({
            'text': answer['para'],
            'link': answer['title'],
            'type': 'bot'
        })
    return render_template('results.html', query=query,
conversation=conversation)
```

在上面的代码中，`index()`函数渲染了主页面模板，而 `query()`函数处理 POST 请求，从表单中获取用户输入的问题，调用问答机器人的 API 获取答案，并将结

果传递给另一个模板进行渲染。

3、集成问答机器人 API

```
def get_answers(query):
    input_data = {'query': query, 'topk': TOPK}
    result = requests.post(SERVICE_ADD, json=input_data)
    res_json = json.loads(result.text)
    answers = []
    for i in range(TOPK):
        title = res_json['answer'][i]['title']
        para = res_json['answer'][i]['para']
        score = res_json['answer'][i]['probability']
        answers.append({'title': title, 'para': para})
    return answers
```

4、使用模板渲染 HTML 页面

Flask 使用 Jinja2 作为模板引擎来渲染 HTML 页面。在项目的 templates 文件夹中创建 HTML 模板文件（index.html 和 result.html），并在这些文件中定义页面的结构和内容。

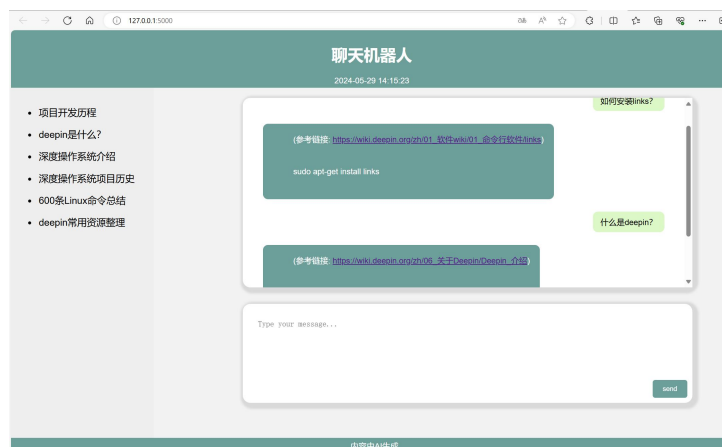
在视图函数中，使用 render_template() 函数来渲染模板，并将数据传递给模板。模板引擎将使用这些数据来动态生成 HTML 页面。

5、运行应用程序

```
if __name__ == '__main__':
    app.run()
```

启动一个开发服务器，并在本地主机的 5000 端口上运行 web 应用程序。打开浏览器并访问 <http://127.0.0.1:5000/> 来查看 web 界面。

6.4. 界面实现



可以点击参考链接进行页面跳转。左侧页面为常用信息的链接，以及我们小组的完整开发历程，可点击进行查看。

7. 项目结果与分析

7.1. 项目结果展示

命令行界面展示：

```
please input a query: deepin是什么
QUERY: deepin是什么

-----

结果1:

参考链接:      https://wiki.deepin.org/zh/home

答案:
deepin 是一个中国开源操作系统项目的名称，它致力于为全球用户提供美观易用、安全可靠的 Linux 发行版。该项目于 2008 年启动，并于 2009 年发布了第一个版本。随后在 2014 年更名为 deepin，并在全球范围内广泛传播。deepin 操作系统采用了全新的桌面环境、丰富的应用软件和工具，以及开源免费的授权模式，旨在为用户提供更好的操作系统使用体验。截止到 2023 年，deepin 操作系统下载次数超过 8000 万次，遍布全球 6 大洲 140 多个镜像站点，是全球排名最高的中国操作系统产品。
CSDN @JuGLe_
```

GUI 界面展示：



Flask web 界面展示:



7.2. 结果分析与讨论

7.2.1. 项目总结

本项目成功开发了一个基于 RocketQA 和 ChatGLM 模型的问答系统，该系统旨在从 Deepin 网站和玲珑使用手册的数据集中检索并生成与用户查询问题相关的答案。通过利用 RocketQA 模型建立索引和 ChatGLM 模型进行答案生成，结合 Tornado 框架构建问答服务应用程序，并设计可视化界面提供用户交互，我们为用户提供了一个高效、便捷的问答服务。

1. 技术实现与集成

索引建立与检索：利用 RocketQA 模型对文档和网页链接进行编码，并通过 FAISS 库创建索引。这种方法极大地提高了检索效率，使得系统能够快速响应用户的查询请求。

答案生成：引入 ChatGLM 模型对检索到的文档进行答案生成。ChatGLM 模型凭借其强大的语言理解和生成能力，能够生成准确、流畅的答案，极大地提升了系统的性能。

服务框架：选择 Tornado 框架作为问答服务应用程序的基础。Tornado 框架以其高效、异步的特性，确保了系统在高并发环境下的稳定性和性能。

2. 用户交互与可视化界面

界面设计：设计简洁明了的可视化界面，包括输入框、查询按钮、结果显示区域和历史记录查阅功能。这些设计使得用户能够轻松地使用系统，并快速获取所需信息。

交互体验：通过提供历史询问记录查阅功能，增强了用户的交互体验。用户可以随时回顾之前的查询和答案，提高了系统的使用便捷性。

3. 部署与运行

环境选择：在 Deepin 操作系统上进行系统部署，确保了系统的稳定性和兼容性。

服务运行：通过创建 Tornado 应用程序实例，使问答服务可运行并提供用户交互。系统的稳定运行为用户提供了持续、可靠的服务支持。

7.2.2. 项目分析

1. 优点与亮点

模型集成：项目成功集成了 RocketQA 和 ChatGLM 两个先进的模型。RocketQA 模型在文档检索方面表现出色，能够快速准确地从大量数据中检索出与查询相关的文档；而 ChatGLM 模型则具有强大的语言理解和生成能力，能够基于检索到的文档生成准确、流畅的答案。这种模型集成的方式极大地提高了系统的整体性能。

索引技术：利用 FAISS 库创建索引，有效提高了检索效率。这使得系统能够在短时间内响应用户的查询请求，并返回准确的结果。

用户友好性：项目设计了简洁明了的可视化界面，用户无需复杂的操作即可轻松使用系统。输入框、查询按钮、结果显示区域等元素的合理布局，使得用户能够直观地了解系统的功能和操作流程。

历史记录查阅：系统提供了历史询问记录查阅功能，用户可以随时回顾之前的查询和答案。这不仅增强了用户的交互体验，还帮助用户更好地管理自己的查询历史。

稳定性与可扩展性：基于 Tornado 框架构建的问答服务应用程序具有高效、异步的特性，确保了系统在高并发环境下的稳定性和性能。这种框架的选择使得系统能够应对大量用户的并发请求，并保持稳定的运行状态。

模块化设计：系统的模块化设计使得各个部分之间具有清晰的边界和接口，便于后期的维护和扩展。未来可以方便地添加新的功能模块或优化现有模块，以

满足不断变化的用户需求。

2. 项目创新点

模型集成方式的创新：将 RocketQA 和 ChatGLM 两个模型进行集成，并应用于问答系统领域，是一种创新性的尝试。这种模型集成方式不仅提高了系统的性能，还为用户提供了更加准确、丰富的答案。这种创新性的应用方式在问答系统领域具有一定的开创性意义。

问答服务框架的创新：基于 Tornado 框架构建的问答服务应用程序采用了高效、异步的设计思想，确保了系统在高并发环境下的稳定性和性能。这种框架的选择和应用是项目在问答服务框架方面的创新点之一。同时，系统的模块化设计也为后期的维护和扩展提供了便利。

用户交互方式的创新：项目提供了历史询问记录查阅功能，这种用户交互方式的创新使得用户能够更加方便地管理自己的查询历史，并随时回顾之前的查询和答案。这种创新性的用户交互方式不仅增强了用户的交互体验，还提高了系统的使用便捷性。

3. 挑战与不足

数据规模：目前系统的数据集主要来自于 Deepin 网站和玲珑使用手册，数据规模相对较小。未来可以考虑扩大数据集范围，引入更多领域的知识，以满足用户更广泛的需求。

模型优化：虽然 RocketQA 和 ChatGLM 模型在系统中表现良好，但仍存在进一步优化的空间。未来可以探索更多的模型优化方法，提高答案的准确性和相关性。

8. 博客记录

8.1. 开发过程中的心得与体会

在整个项目开发过程中，一直在使用博客记录开发历程。详情请参照（[基于国产操作系的问答机器人——博客 2-CSDN 博客](#)）