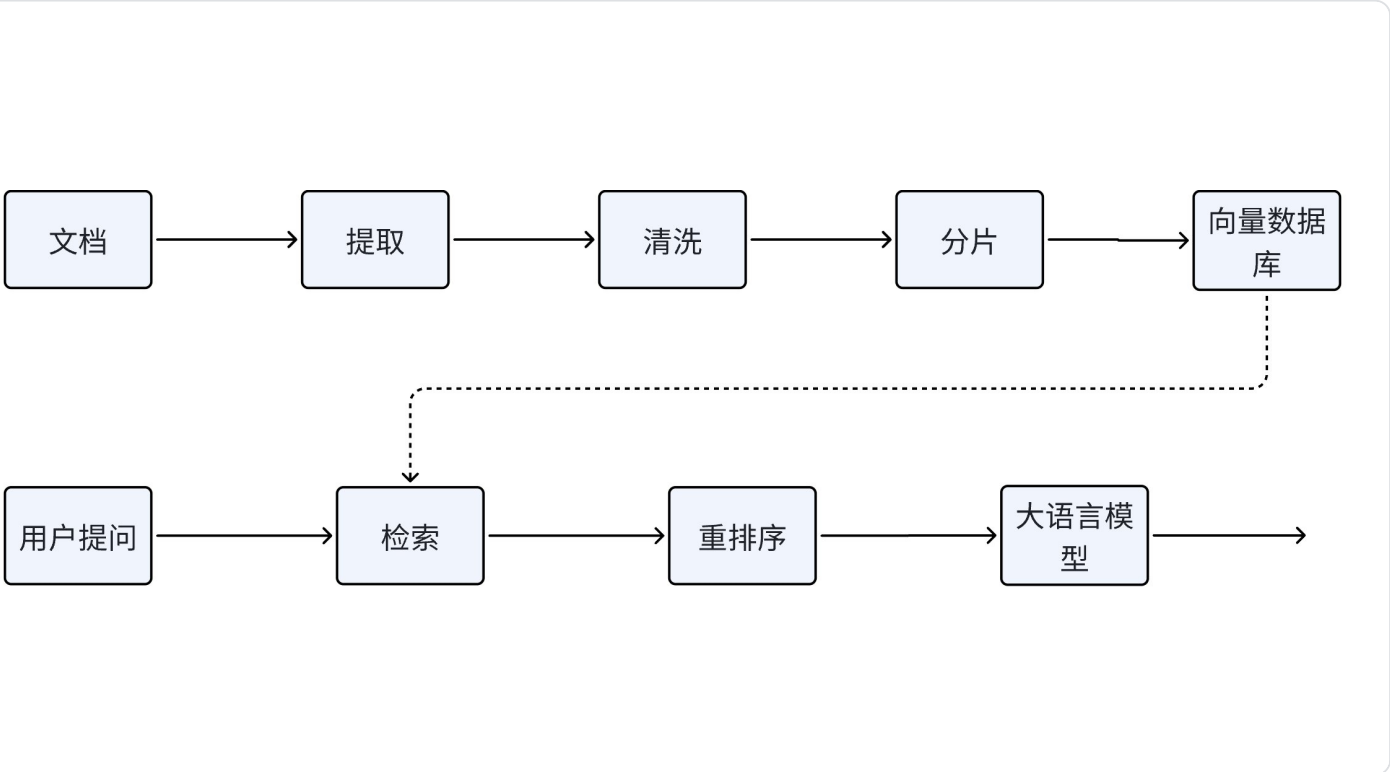
社会信用治理决策支持与风险预警智能分析



1 提取

提取模块负责从不同来源的数据中提取文本内容，以生成结构化的Document对象。这些对象随后用于知识检索与生成任务。Extractor模块是RAG系统中实现信息处理与标准化的重要组成部分。

Extractor模块主要功能包括：

- 支持多种数据源，包括上传文件、网页链接、Notion页面。
- 根据数据源自动选择合适的文本提取方法。
- 支持直接返回纯文本内容或结构化的Document对象列表。

1.1 提取模块实现

1.1.1 关键方法

（1）load_from_upload_file

功能：处理上传文件内容。

代码块

```
def load_from_upload_file(
    cls, upload_file: UploadFile, return_text: bool = False, is_automatic: bool
 = False
) -> Union[list[Document], str]:
    extract_setting = ExtractSetting(
        datasource_type="upload_file", upload_file=upload_file,
document_model="text_model"
    )
    if return_text:
        delimiter = "\n"
        return delimiter.join([document.page_content for document in
cls.extract(extract_setting, is_automatic)])
    else:
        return cls.extract(extract_setting, is_automatic)
```

- 输入：UploadFile对象
- 输出：纯文本或Document对象列表

（2）load_from_url

功能：从网页链接下载并提取内容。

代码块

```
def load_from_url(cls, url: str, return_text: bool = False) ->
Union[list[Document], str]:
    response = ssrf_proxy.get(url, headers={"User-Agent": USER_AGENT})

    with tempfile.TemporaryDirectory() as temp_dir:
        suffix = Path(url).suffix
        if not suffix and suffix != ".":
            # get content-type
            if response.headers.get("Content-Type"):
                suffix = "." + response.headers.get("Content-Type").split("/")
[-1]
            else:
```

```
                content_disposition = response.headers.get("Content-
Disposition")
                filename_match = re.search(r'filename="([^"]+)"',
content_disposition)
                if filename_match:
                    filename = unquote(filename_match.group(1))
                    match = re.search(r"\.(\w+)$", filename)
                    if match:
                        suffix = "." + match.group(1)
                    else:
                        suffix = ""
        file_path = f"{temp_dir}/{next(tempfile._get_candidate_names())}
{suffix}"  # type: ignore
        Path(file_path).write_bytes(response.content)
        extract_setting = ExtractSetting(datasource_type="upload_file",
document_model="text_model")
        if return_text:
            delimiter = "\n"
            return delimiter.join(
                [
                    document.page_content
                    for document in
cls.extract(extract_setting=extract_setting, file_path=file_path)
                ]
            )
        else:
            return cls.extract(extract_setting=extract_setting,
file_path=file_path)
```

- 输入：URL字符串
- 输出：纯文本或Document对象列表

（3）extract

功能：实际执行提取任务。

代码块

```
def extract(
    cls, extract_setting: ExtractSetting, is_automatic: bool = False,
file_path: Optional[str] = None
) -> list[Document]:
    if extract_setting.datasource_type == DatasourceType.FILE.value:
        with tempfile.TemporaryDirectory() as temp_dir:
            if not file_path:
```

```
                assert extract_setting.upload_file is not None, "upload_file is required"
                upload_file: UploadFile = extract_setting.upload_file
                suffix = Path(upload_file.key).suffix
                file_path = f"{temp_dir}/{next(tempfile._get_candidate_names())}{suffix}"  # type: ignore
                storage.download(upload_file.key, file_path)
            input_file = Path(file_path)
            file_extension = input_file.suffix.lower()
            etl_type = dify_config.ETL_TYPE
            extractor: Optional[BaseExtractor] = None
            if etl_type == "Unstructured":
                unstructured_api_url = dify_config.UNSTRUCTURED_API_URL
                unstructured_api_key = dify_config.UNSTRUCTURED_API_KEY or ""
                if file_extension in {".xlsx", ".xls"}:
                    extractor = ExcelExtractor(file_path)
                elif file_extension == ".pdf":
                    extractor = PdfExtractor(file_path)
                elif file_extension in {".md", ".markdown", ".mdx"}:
                    extractor = (
                        UnstructuredMarkdownExtractor(file_path, unstructured_api_url, unstructured_api_key)
                        if is_automatic
                        else MarkdownExtractor(file_path, autodetect_encoding=True)
                    )
                elif file_extension in {".htm", ".html"}:
                    extractor = HtmlExtractor(file_path)
                elif file_extension == ".docx":
                    extractor = WordExtractor(file_path, upload_file.tenant_id, upload_file.created_by)
                elif file_extension == ".csv":
                    extractor = CSVExtractor(file_path, autodetect_encoding=True)
                elif file_extension == ".msg":
                    extractor = UnstructuredMsgExtractor(file_path, unstructured_api_url, unstructured_api_key)
                elif file_extension == ".eml":
                    extractor = UnstructuredEmailExtractor(file_path, unstructured_api_url, unstructured_api_key)
                elif file_extension == ".ppt":
                    extractor = UnstructuredPPTExtractor(file_path, unstructured_api_url, unstructured_api_key)
                elif file_extension == ".pptx":
                    extractor = UnstructuredPPTXExtractor(file_path, unstructured_api_url, unstructured_api_key)
                elif file_extension == ".xml":
```

```
                    extractor = UnstructuredXmlExtractor(file_path,
unstructured_api_url, unstructured_api_key)
                elif file_extension == ".epub":
                    extractor = UnstructuredEpubExtractor(file_path,
unstructured_api_url, unstructured_api_key)
                else:
                    # txt
                    extractor = TextExtractor(file_path,
autodetect_encoding=True)
            else:
                if file_extension in {".xlsx", ".xls"}:
                    extractor = ExcelExtractor(file_path)
                elif file_extension == ".pdf":
                    extractor = PdfExtractor(file_path)
                elif file_extension in {".md", ".markdown", ".mdx"}:
                    extractor = MarkdownExtractor(file_path,
autodetect_encoding=True)
                elif file_extension in {".htm", ".html"}:
                    extractor = HtmlExtractor(file_path)
                elif file_extension == ".docx":
                    extractor = WordExtractor(file_path,
upload_file.tenant_id, upload_file.created_by)
                elif file_extension == ".csv":
                    extractor = CSVExtractor(file_path,
autodetect_encoding=True)
                elif file_extension == ".epub":
                    extractor = UnstructuredEpubExtractor(file_path)
                else:
                    # txt
                    extractor = TextExtractor(file_path,
autodetect_encoding=True)
            return extractor.extract()
```

- 输入：ExtractSetting对象
- 输出：Document对象列表

1.2 提取模块支持的文件类型

1.2.1 结构化文件

pdf、word、markdown、excel、html、txt

1.2.2 非结构化文件

msg、xml、epub

2 清洗

清洗模块负责对从各类数据源检索到的文本数据进行标准化和净化，确保文本干净、统一，以提高后续检索与生成任务的准确性。

2.1 清洗模块实现

2.1.1 基础文本清理功能

代码块

```
# 特殊符号标准化
text = re.sub(r"<\|", "<", text)
text = re.sub(r"\|>", ">", text)

# 无效字符过滤
text = re.sub(r"[\x00-\x08\x0B\x0C\x0E-\x1F\x7F\xEF\xBF\xBE]", "", text)
text = re.sub("\ufffe", "", text)
```

默认清理规则在所有文本处理中均会自动执行，包括：

- 特殊符号的标准化：
  - 将<| 替换为 <
  - 将|> 替换为 >
- 无效字符过滤：
  - 删除不可见的 ASCII 控制字符及异常的 Unicode 字符，如 \x00-\x08, \x0B, \x0C, \x0E-\x1F, \x7F, \xEF, \xBF, \xBE, \uFFFE。

2.1.2 自定义文本处理规则

模块通过传入规则字典实现自定义清理，目前支持的规则包括：

1. 移除额外空白（remove_extra_spaces）

代码块

```
# 连续3个及以上换行符替换为2个换行符
text = re.sub(r"\n{3,}", "\n\n", text)
# 连续2个及以上空白符（空格、制表符、回车符、Unicode空白符）替换为单个空格
text = re.sub(r"[\t\f\r\x20\u00a0\u1680\u180e\u2000-\u200a\u202f\u205f\u3000]
{2,}", " ", text)
```

- 连续出现的三个或以上换行符统一替换为两个换行符。

- 连续出现的两个或以上空白符（包括空格、制表符、回车符及 Unicode 空白符）替换为单个空格。

2. 移除电子邮件和 URL（remove_urls_emails）

代码块

```
# 移除电子邮件地址
text = re.sub(r"[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+", "", text)
# 移除URL链接
text = re.sub(r"https?://[^\s]+", "", text)
```

- 自动识别并移除文本中出现的电子邮件地址。
- 自动识别并移除文本中出现的所有 URL 链接。

3 分片

文本分片模块的主要功能是将长文本内容合理分割成适合大语言模型处理的小片段，以确保输入到模型的内容符合其最大输入长度限制，同时优化检索与生成效果。

清洗和分片前的原文：

文化和旅游部、公安部关于加强电竞酒店管理中未成年人保护工作的通知

文化和旅游部、公安部关于加强电竞酒店管理中未成年人保护工作的通知

（文旅市场发〔2023〕82号）

各省、自治区、直辖市文化和旅游厅（局）、公安厅（局），新疆生产建设兵团文化体育广电和旅游局、公安局：

近年来，电竞酒店行业快速发展，在满足群众需求、扩大消费等方面起到了积极作用，同时也存在接待未成年人等引发社会关注的问题。为认真贯彻落实《中华人民共和国未成年人保护法》有关规定，根据最有利于未成年人原则，切实加强电竞酒店管理中未成年人保护工作，促进行业健康有序发展，现就有关事项通知如下。

一、提高政治站位，充分认识加强电竞酒店管理中未成年人保护工作的重要意义

（一）提高政治站位。党中央、国务院历来高度重视未成年人保护工作，习近平总书记和有关中央领导同志多次对未成年人保护工作作出重要指示批示。加强未成年人保护工作，促进未成年人健康成长，是贯彻落实党中央、国务院决策部署开回应社会关切的重要举措，是各级政府管理部门肩负的重要职责。

（二）强化责任担当。各级文化和旅游行政部门、公安机关应当提高政治站位，主动担当作为，坚持问题导向，不断创新监管方式，既要坚决管住电竞酒店违规接待未成年人问题，又要秉持包容审慎原则，合理引导电竞酒店行业健康有序发展，做到守土有责、守土尽责。

二、加强底线管理，严禁电竞酒店违规接待未成年人

（三）明确业态属性。本通知所称的电竞酒店是指通过设置电竞房向消费者提供电子竞技娱乐服务的新型住宿业态，包括所有客房均为电竞房的专业电竞酒店和利用部分客房开设电竞房区域的非专业电竞酒店。电竞酒店每间电竞房的床位数不得超过6张，计算机数量和入住人员不得超过床位数。

（四）严禁电竞酒店违规接待未成年人。专业电竞酒店和非专业电竞酒店的电竞房区域，属于不适宜未成年人活动的场所。电竞酒店经营者应当遵守《中华人民共和国未成年人保护法》等有关法律法规，不得允许未成年人进入专业电竞酒店和非专业电竞酒店的电竞房区域。

三、强化主体责任，严格落实未成年人保护规定

（五）设置禁入标志。专业电竞酒店经营者应当在酒店入口处的显著位置悬挂未成年人禁入标志；非专业电竞酒店经营者应当在相近楼层集中设置电竞房并划定电竞房区域，在电竞房区域入口处的显著位置悬挂未成年人禁入标志。电竞酒店经营者应当在前台显著位置和客房管理系统明示电竞房区域分布图。鼓励非专业电竞酒店经营者对电竞房区域进行物理隔离、电梯控制，防止未成年人擅自进入。

（六）履行告知义务。电竞酒店经营者应当在消费者预定、入住等环节明确告知其电竞房区域不接待未成年人；通过电子商务平台等开展客房预定的，应当以显著方式提示消费者电竞房区域不接待未成年人。电子商务平台经营者应当核验电竞酒店提示信息。

（七）落实“五必须”规定。电竞酒店非电竞房区域接待未成年人入住时，经营者应当严格落实“五必须”规定：必须查验入住未成年人身份并如实登记；必须询问未成年人父母或者其他监护人的联系方式并记录备案；必须询问同住人员身份关系等情况并记录备案；必须加强安全巡查和访客管理，预防对未成年人的不法侵害；必须立即向公安机关报告可疑情况，并及时联系未成年人的父母或者其他监护人，并同时采取相应安全保护措施。

（八）实施网络安全技术措施。电竞酒店经营者应当依法制定信息网络安全管理制度和应急处置预案，实施互联网安全保护技术措施。电竞酒店经营者应当设置禁止未成年人登录计算机、消费时长提示等功能，并通过网络技术措施服务提供者向文化和旅游行政部门提供电竞房分布、设置禁止未成年人登录功能以及阻断登录情况等可查询信息。

（九）实施图像采集技术措施。电竞酒店经营者应当按照有关规定，在大厅、前台、通道、电竞房区域主要出入口等公共区域内的合理位置安装图像采集设备并设置采集区域提示标识，加强检查值守，发现有未成年人违规进入电竞房区域的，要及时劝阻并联系其父母或者其他监护人。图像采集信息应当依法留存，不得不当披露、传播，并在文化和旅游行政部门等部门检查电竞房时提供查询。

（十）建立日常巡查制度。电竞酒店经营者应当建立日常巡查制度，发现有未成年人违规进入、未实名登记擅自进入等违法行为的，应当立即制止并分别向所在地县级文化和旅游行政部门、公安机关报告。文化和旅游行政部门、公安机关等有关部门有权依法对辖区内电竞酒店的电竞房实施监督检查，电竞酒店经营者应当配合，不得拒绝、阻挠。

四、加强协同监管，形成未成年人保护合力

（十一）建立协同监管机制。地方各级文化和旅游行政部门、公安机关应当会同相关部门，建立电竞酒店未成年人保护协同监管机制，加强信息通报、线索移送、执法联动等工作，引导督促经营者严格落实实名登记、设置未成年人禁入标志、禁止违规接待未成年人等要求，依法查处违规经营行为。公安机关在工作中发现电竞酒店经营者违规接待未成年人的，及时通报文化和旅游行政部门依法查处；文化和旅游行政部门在工作中发现电竞酒店经营者未落实实名登记及“五必须”规定的，及时通报公安机关依法查处。

（十二）严格文化和旅游市场行政处罚。电竞酒店经营者违规接待未成年人或者未设置未成年人禁入标志的，由文化和旅游行政部门依照《中华人民共和国未成年人保护法》第一百二十三条予以处罚。

（十三）严格治安管理行政处罚。电竞酒店非电竞房区域接待未成年人入住，或者接待未成年人和成年人共同入住时，未询问父母或者其他监护人的联系方式、入住人员的身份关系等有关情况的，由公安机关依照《中华人民共和国未成年人保护法》第一百二十二条予以处罚。电竞酒店未取得特种行业许可证，擅自经营旅馆业的，由公安机关依照《中华人民共和国治安管理处罚法》第五十四条予以处罚，并对非法经营行为予以取缔。

（十四）加强行业自律。电竞酒店有关行业协会应当加强行业自律，制定行业规范，开展培训教育，探索开展本领域的信用评价、服务等级评定工作，引导经营者严格落实《中华人民共和国未成年人保护法》有关规定。

（十五）加强社会监督。社会公众可以依法向文化和旅游行政部门、公安机关反映电竞酒店违规接待未成年人等情况；鼓励和支持电竞酒店入住人员向文化和旅游行政部门、公安机关举报电竞酒店违规接待未成年人等线索。

各省级文化和旅游行政部门应当会同公安机关部署开展电竞酒店摸底排查工作，摸清辖区内电竞酒店的企业名称、类型、住所、电竞房数量、计算机台数等情况，并于2023年10月31日前分别报送文化和旅游部、公安部。

特此通知。

分片后的文段示意：

3.1 分片模块实现

3.1.1 基础抽象类：TextSplitter

定义了通用的文本分片接口与公共参数：

- **chunk_size**：单个文本分片最大长度。
- **chunk_overlap**：相邻分片之间的重叠长度，避免上下文丢失。
- **length_function**：计算文本长度的函数（默认按字符）。
- **keep_separator**：分片时是否保留分隔符。
- **add_start_index**：是否记录分片在原文中的起始位置。

3.1.2 具体实现类

1. **CharacterTextSplitter**

代码块

```
class CharacterTextSplitter(TextSplitter):
    """Splitting text that looks at characters."""

    def __init__(self, separator: str = "\n\n", **kwargs: Any) -> None:
        """Create a new TextSplitter."""
        super().__init__(**kwargs)
        self._separator = separator

    def split_text(self, text: str) -> list[str]:
        """Split incoming text and return chunks."""
        # First we naively split the large input into a bunch of smaller ones.
```

```
        splits = _split_text_with_regex(text, self._separator, self._keep_separator)
        _separator = "" if self._keep_separator else self._separator
        _good_splits_lengths = []  # cache the lengths of the splits
        if splits:
            _good_splits_lengths.extend(self._length_function(splits))
        return self._merge_splits(splits, _separator, _good_splits_lengths)
```

- 基于固定字符（默认"\n\n"）进行简单分割。
- 适用于普通文本文档，简单快速。

2. **MarkdownHeaderTextSplitter**

代码块

```
# 解析标题层级，更新元数据
if stripped_line.startswith(sep) and (len(stripped_line) == len(sep) or stripped_line[len(sep)] == " "):
    current_header_level = sep.count("#")
    # 从 header_stack 中弹出不再有效的上级标题
    while header_stack and header_stack[-1]["level"] >= current_header_level:
        header_stack.pop()
        initial_metadata.pop(popped_header["name"], None)
    header = {
        "level": current_header_level,
        "name": name,
        "data": stripped_line[len(sep):].strip(),
    }
    header_stack.append(header)
    initial_metadata[name] = header["data"]
# 每一行都与当前的元数据绑定
lines_with_metadata.append({
    "content": \"\\n\".join(current_content),
    "metadata": current_metadata.copy(),
})
# 根据 metadata 聚合段落为最终文档
def aggregate_lines_to_chunks(self, lines: list[LineType]) -> list[Document]:
    ...
    return [Document(page_content=chunk[\"content\"], metadata=chunk[\"metadata\"]) for chunk in aggregated_chunks]

```

- 按照Markdown标题结构（如"#", "##"）分割。
- 自动提取标题为元数据，便于检索。

- 特别适用于具有明显标题结构的Markdown文档。

3. **TokenTextSplitter**

代码块

```
def _encode(_text: str) -> list[int]:
    return self._tokenizer.encode(
        _text,
        allowed_special=self._allowed_special,
        disallowed_special=self._disallowed_special,
    )
# 核心分割函数，按 token 块分割，支持重叠
def split_text_on_tokens(*, text: str, tokenizer: Tokenizer) -> list[str]:
    input_ids = tokenizer.encode(text)
    start_idx = 0
    ...
    while start_idx < len(input_ids):
        splits.append(tokenizer.decode(chunk_ids))
        start_idx += tokenizer.tokens_per_chunk - tokenizer.chunk_overlap
# 组装 tokenizer 配置，并调用分割函数
tokenizer = Tokenizer(
    chunk_overlap=self._chunk_overlap,
    tokens_per_chunk=self._chunk_size,
    decode=self._tokenizer.decode,
    encode=_encode,
)
return split_text_on_tokens(text=text, tokenizer=tokenizer)
```

- 基于语言模型的tokenizer（如GPT-2的tiktoken）进行分割。
- 精确控制文本片段长度，适用于严格控制输入token长度的场景。

4. **RecursiveCharacterTextSplitter**

代码块

```
# 自动选择合适的分隔符，按优先顺序判断
for i, _s in enumerate(separators):
    if _s == \"\" or re.search(_s, text):
        separator = _s
        new_separators = separators[i + 1:]
        break
# 初步使用正则分割文本
splits = _split_text_with_regex(text, separator, self._keep_separator)
# 如果某段过长则递归使用下一级分隔符继续分割
if s_len >= self._chunk_size:
    if not new_separators:
```

```
            final_chunks.append(s)
        else:
            other_info = self._split_text(s, new_separators)
            final_chunks.extend(other_info)
    # 使用 merge 机制将多个小段拼接为合适大小的段落
    merged_text = self._merge_splits(_good_splits, _separator,
    _good_splits_lengths)
```

- 采用递归策略，从较大的分隔符到更小的分隔符依次分割文本。
- 确保分割后片段均低于设定的chunk_size。
- 适用于复杂长文本场景。

3.1.3 增强与修复类

1. EnhanceRecursiveCharacterTextSplitter

代码块

```
def from_encoder(
    cls: type[TS],
    embedding_model_instance: Optional[ModelInstance],
    allowed_special: Union[Literal["all"], Set[str]] = set(),
    disallowed_special: Union[Literal["all"], Collection[str]] = "all",
    **kwargs: Any,
)
# 自定义 token 长度计算逻辑，优先使用 embedding_model_instance
def _token_encoder(texts: list[str]) -> list[int]:
    if not texts:
        return []
    if embedding_model_instance:
        return
embedding_model_instance.get_text_embedding_num_tokens(texts=texts)
    else:
        return [GPT2Tokenizer.get_num_tokens(text) for text in texts]
# 若当前类是 TokenTextSplitter 子类，还可自动注入模型名称和特殊 token 控制参数
if issubclass(cls, TokenTextSplitter):
    extra_kwargs = {
        "model_name": embedding_model_instance.model if
embedding_model_instance else "gpt2",
        "allowed_special": allowed_special,
        "disallowed_special": disallowed_special,
    }
    kwargs = {**kwargs, **extra_kwargs}

return cls(length_function=_token_encoder, **kwargs)
```

- 避免直接依赖tiktoken库，改用自定义encoder（如GPT2Tokenizer）。
- 提供了from_encoder接口，允许自定义embedding模型或tokenizer。

2. FixedRecursiveCharacterTextSplitter

代码块

```
def __init__(self, fixed_separator: str = "\\n\\n", separators: Optional[list[str]] = None, **kwargs: Any):
    super().__init__(**kwargs)
    self._fixed_separator = fixed_separator
    self._separators = separators or ["\\n\\n", "\\n", " ", ""]
# 优先使用 fixed_separator 对文本进行初步分割
def split_text(self, text: str) -> list[str]:
    if self._fixed_separator:
        chunks = text.split(self._fixed_separator)
    else:
        chunks = [text]

    final_chunks = []
    chunks_lengths = self._length_function(chunks)

    for chunk, chunk_length in zip(chunks, chunks_lengths):
        if chunk_length > self._chunk_size:
            final_chunks.extend(self.recursive_split_text(chunk))
        else:
            final_chunks.append(chunk)

    return final_chunks
# 若初步分割后的段落仍然过长，继续调用递归切分
def recursive_split_text(self, text: str) -> list[str]:
    ...
    for i, _s in enumerate(self._separators):
        if _s == "" or _s in text:
            separator = _s
            new_separators = self._separators[i + 1:]
            break

    if separator:
        splits = text.split(separator)
    else:
        splits = list(text)
    ...
    # 若子片段仍过长，递归切分或直接加入
    if s_len < self._chunk_size:
        ...
```

```
        else:
            if not new_separators:
                final_chunks.append(s)
            else:
                other_info = self._split_text(s, new_separators)
                final_chunks.extend(other_info)
```

- 继承自EnhanceRecursiveCharacterTextSplitter。
- 引入了fixed_separator进行初步分割，减少递归次数，优化分割效率。
- 性能提升且降低对外部依赖的敏感性。

4 检索

4.1 检索模块实现

模块主要实现了以下功能：

- 支持单数据集与多数据集检索。
- 支持语义检索、关键词检索及多种混合检索策略。
- 支持多线程并行检索，提升系统响应速度。
- 支持智能重排序（reranking），确保返回结果质量。

4.1.1 主入口方法：retrieve

1. 功能

接收用户查询，综合配置，执行数据检索。

```
代码块
1  def retrieve(self, app_id: str, user_id: str, tenant_id: str, model_config:
   ModelConfigWithCredentialsEntity, config: DatasetEntity, query: str,
   invoke_from: InvokeFrom, show_retrieve_source: bool, hit_callback:
   DatasetIndexToolCallbackHandler, message_id: str, memory:
   Optional[TokenBufferMemory] = None, inputs: Optional[Mapping[str, Any]] =
   None) -> Optional[str]:
2      dataset_ids = config.dataset_ids
3      if len(dataset_ids) == 0:
4          return None
5      retrieve_config = config.retrieve_config
6
7      available_datasets = self._get_available_datasets(dataset_ids, tenant_id)
8      metadata_filter_document_ids, metadata_condition =
   self._get_metadata_filter_condition(available_datasets, query, tenant_id,
   user_id, retrieve_config, inputs)
9
10     if retrieve_config.retrieve_strategy ==
   DatasetRetrieveConfigEntity.RetrieveStrategy.SINGLE:
11         all_documents = self.single_retrieve(app_id, tenant_id, user_id,
   invoke_from.to_source(), available_datasets, query, model_config, message_id,
   metadata_filter_document_ids, metadata_condition)
12     else:
13         all_documents = self.multiple_retrieve(app_id, tenant_id, user_id,
   invoke_from.to_source(), available_datasets, query, retrieve_config,
   message_id, metadata_filter_document_ids, metadata_condition)
14
15     return self._format_documents(all_documents, show_retrieve_source,
   invoke_from, hit_callback)
```

2. 输入参数

- **应用信息**：app_id, user_id, tenant_id。
- **模型配置**：model_config，定义使用的大语言模型配置。
- **数据集配置**：config，指定需要检索的数据集和检索策略。
- **查询内容**：query，用户的提问语句。
- **调用上下文**：invoke_from，调用场景区分（如开发者、终端用户）。
- **显示来源**：show_retrieve_source，控制是否在结果中包含检索来源信息。
- **回调机制**：hit_callback，用于记录检索的详细信息。

3. 处理逻辑

- 验证数据集有效性。
- 根据模型特性选择路由策略（REACT_ROUTER或ROUTER）。
- 执行元数据过滤条件判断，获取符合过滤条件的文档ID。
- 根据配置进行单一或多个数据集检索。
- 对检索结果进行合并、重排序。
- 格式化检索结果，回调记录详细检索信息。

4.1.2 单数据集检索：single_retrieve

代码块

```
    def single_retrieve(self, app_id: str, tenant_id: str, user_id: str,
    user_from: str, available_datasets: list, query: str, model_config:
    ModelConfigWithCredentialsEntity, message_id: Optional[str],
    metadata_filter_document_ids: Optional[dict[str, list[str]]],
    metadata_condition: Optional[MetadataCondition]):
        dataset = self._select_dataset(query, available_datasets, model_config)
        if not dataset:
            return []

        results = self._perform_retrieval(dataset, query,
    metadata_filter_document_ids, metadata_condition)
        self._on_query(query, [dataset.id], app_id, user_from, user_id)

        return results
```

1. 功能

针对特定数据集进行检索操作。

2. 逻辑

- 路由模型决定检索目标数据集。
- 根据数据集索引技术（语义或关键词），调用对应的检索服务。
- 根据元数据过滤的文档ID进一步缩小结果。
- 记录检索过程信息和性能指标。

4.1.3 多数据集并行检索：multiple_retrieve

代码块

```
def multiple_retrieve(self, app_id: str, tenant_id: str, user_id: str, user_from: str, available_datasets: list, query: str, retrieve_config: DatasetRetrieveConfigEntity, message_id: Optional[str], metadata_filter_document_ids: Optional[dict[str, list[str]]], metadata_condition: Optional[MetadataCondition]):
    threads = []
    all_documents: list[Document] = []

    for dataset in available_datasets:
        thread = threading.Thread(target=self._retriever, args=(current_app._get_current_object(), dataset, query, retrieve_config.top_k, all_documents, metadata_filter_document_ids, metadata_condition))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    all_documents = self._rerank_documents(all_documents, retrieve_config)
    self._on_query(query, [dataset.id for dataset in available_datasets], app_id, user_from, user_id)

    return all_documents
```

1. 功能

同时从多个数据集中检索内容，提供高效检索能力。

2. 逻辑

- 启动多个线程并行处理各数据集检索任务。
- 检查数据集索引一致性，必要时强制启用智能重排序。
- 合并各线程返回结果并通过智能重排序算法优化结果。
- 记录整体检索性能和命中情况。

4.1.4 RetrievalService类

核心类，实现了文档检索及重排序的主要接口。

代码块

```
class RetrievalService:
    def retrieve(cls, retrieval_method, dataset_id, query, top_k, score_threshold=0.0,
                 reranking_model=None, reranking_mode="reranking_model", weights=None,
```

```
                  document_ids_filter=None):
        if not query:
            return []
        dataset = cls._get_dataset(dataset_id)
        if not dataset or dataset.available_document_count == 0 or
dataset.available_segment_count == 0:
            return []

        all_documents, exceptions = [], []
        with
ThreadPoolExecutor(max_workers=dify_config.RETRIEVAL_SERVICE_EXECUTORS) as
executor:
            futures = []
            if retrieval_method == "keyword_search":
                futures.append(executor.submit(cls.keyword_search,
current_app._get_current_object(),
                                               dataset_id, query, top_k,
all_documents,
                                               exceptions,
document_ids_filter))
            if RetrievalMethod.is_support_semantic_search(retrieval_method):
                futures.append(executor.submit(cls.embedding_search,
current_app._get_current_object(),
                                               dataset_id, query, top_k,
score_threshold,
                                               reranking_model, all_documents,
retrieval_method,
                                               exceptions,
document_ids_filter))
            if RetrievalMethod.is_support_fulltext_search(retrieval_method):
                futures.append(executor.submit(cls.full_text_index_search,
current_app._get_current_object(),
                                               dataset_id, query, top_k,
score_threshold,
                                               reranking_model, all_documents,
retrieval_method,
                                               exceptions,
document_ids_filter))
            concurrent.futures.wait(futures, timeout=30)

        if exceptions:
            raise ValueError(";\\n".join(exceptions))

        if retrieval_method == RetrievalMethod.HYBRID_SEARCH.value:
            data_post_processor = DataPostProcessor(str(dataset.tenant_id),
reranking_mode,
```

```
                                                    reranking_model, weights,
    False)
                all_documents = data_post_processor.invoke(query, all_documents,
                                                           score_threshold, top_k)

            return all_documents
```

执行流程：

- 并行执行关键词搜索、语义搜索或全文搜索。
- 收集所有检索结果。
- 若采用混合搜索（Hybrid），调用DataPostProcessor进行重排序。

4.1.5 检索核心方法

1. **keyword_search()**

代码块

```
def keyword_search(cls, flask_app, dataset_id, query, top_k, all_documents,
exceptions, document_ids_filter=None):
    with flask_app.app_context():
        try:
            dataset = cls._get_dataset(dataset_id)
            if not dataset:
                raise ValueError("dataset not found")
            keyword = Keyword(dataset=dataset)
            documents = keyword.search(cls.escape_query_for_search(query),
                                       top_k=top_k,
document_ids_filter=document_ids_filter)
            all_documents.extend(documents)
        except Exception as e:
            exceptions.append(str(e))
```

- 通过Keyword类实现关键词匹配检索。
- 对输入的关键词进行转义处理，确保搜索安全。
- 支持文档ID过滤功能。

2. **embedding_search()**

代码块

```
def embedding_search(cls, flask_app, dataset_id, query, top_k, score_threshold,
                     reranking_model, all_documents, retrieval_method,
                     exceptions, document_ids_filter=None):
```

```
    with flask_app.app_context():
        try:
            dataset = cls._get_dataset(dataset_id)
            if not dataset:
                raise ValueError("dataset not found")
            vector = Vector(dataset=dataset)
            documents = vector.search_by_vector(query,
"similarity_score_threshold", top_k,
                                                score_threshold, {"group_id":
[dataset.id]},
                                                document_ids_filter)
            if reranking_model and retrieval_method ==
RetrievalMethod.SEMANTIC_SEARCH.value:
                data_post_processor = DataPostProcessor(str(dataset.tenant_id),

str(RerankMode.RERANKING_MODEL.value),
                                                        reranking_model, None,
False)
                all_documents.extend(data_post_processor.invoke(query,
documents,

score_threshold, len(documents)))
            else:
                all_documents.extend(documents)
        except Exception as e:
            exceptions.append(str(e))
```

- 使用Vector类进行向量相似度检索（语义搜索）。
- 支持基于相似度分数阈值的过滤。
- 可选通过重排序模型进一步优化检索结果。

3. **full_text_index_search()**

代码块

```
def full_text_index_search(cls, flask_app, dataset_id, query, top_k,
score_threshold,
                           reranking_model, all_documents, retrieval_method,
                           exceptions, document_ids_filter=None):
    with flask_app.app_context():
        try:
            dataset = cls._get_dataset(dataset_id)
            if not dataset:
                raise ValueError("dataset not found")
            vector_processor = Vector(dataset=dataset)
```

```
            documents =
vector_processor.search_by_full_text(cls.escape_query_for_search(query),
                                                                top_k,
document_ids_filter)
            if reranking_model and retrieval_method ==
RetrievalMethod.FULL_TEXT_SEARCH.value:
                data_post_processor = DataPostProcessor(str(dataset.tenant_id),

str(RerankMode.RERANKING_MODEL.value),
                                                        reranking_model, None,
False)
                all_documents.extend(data_post_processor.invoke(query,
documents,

score_threshold, len(documents)))
            else:
                all_documents.extend(documents)
        except Exception as e:
            exceptions.append(str(e))
```

- 使用Vector类实现全文索引搜索。
- 对查询文本进行转义处理，确保全文搜索稳定。
- 可选择对搜索结果进行模型重排序。

4.1.6 format_retrieval_documents方法

代码块

```
def format_retrieval_documents(cls, documents):
    if not documents:
        return []
    try:
        document_ids = {doc.metadata.get("document_id") for doc in documents if
 "document_id" in doc.metadata}
        dataset_documents = {
            doc.id: doc for doc in db.session.query(DatasetDocument)
            .filter(DatasetDocument.id.in_(document_ids))
            .options(load_only(DatasetDocument.id, DatasetDocument.doc_form,
DatasetDocument.dataset_id)).all()
        }

        records, include_segment_ids, segment_child_map = [], set(), {}

        for document in documents:
            document_id = document.metadata.get("document_id")
```

```
            dataset_document = dataset_documents.get(document_id)
            if not dataset_document:
                continue

            if dataset_document.doc_form == IndexType.PARENT_CHILD_INDEX:
                child_index_node_id = document.metadata.get("doc_id")
                child_chunk = db.session.query(ChildChunk).filter(
                    ChildChunk.index_node_id == child_index_node_id).first()
                if not child_chunk:
                    continue

                segment = db.session.query(DocumentSegment).filter(
                    DocumentSegment.dataset_id == dataset_document.dataset_id,
                    DocumentSegment.enabled == True,
                    DocumentSegment.status == "completed",
                    DocumentSegment.id == child_chunk.segment_id).first()
                if not segment:
                    continue

                if segment.id not in include_segment_ids:
                    include_segment_ids.add(segment.id)
                    segment_child_map[segment.id] = {
                        "max_score": document.metadata.get("score", 0.0),
                        "child_chunks": [{
                            "id": child_chunk.id,
                            "content": child_chunk.content,
                            "position": child_chunk.position,
                            "score": document.metadata.get("score", 0.0)}]}
                    records.append({"segment": segment})
                else:
                    segment_child_map[segment.id]["child_chunks"].append({
                        "id": child_chunk.id,
                        "content": child_chunk.content,
                        "position": child_chunk.position,
                        "score": document.metadata.get("score", 0.0)})
                    segment_child_map[segment.id]["max_score"] = max(
                        segment_child_map[segment.id]["max_score"],
                        document.metadata.get("score", 0.0))
            else:
                index_node_id = document.metadata.get("doc_id")
                segment = db.session.query(DocumentSegment).filter(
                    DocumentSegment.dataset_id == dataset_document.dataset_id,
                    DocumentSegment.enabled == True,
                    DocumentSegment.status == "completed",
                    DocumentSegment.index_node_id == index_node_id).first()
                if not segment:
                    continue
```

```
                records.append({"segment": segment, "score":
document.metadata.get("score")})

        for record in records:
            segment_id = record["segment"].id
            if segment_id in segment_child_map:
                record["child_chunks"] = segment_child_map[segment_id]
["child_chunks"]
                record["score"] = segment_child_map[segment_id]["max_score"]

        return [RetrievalSegments(**record) for record in records]
    except Exception as e:
        db.session.rollback()
        raise e
```

该方法实现了高效的结果格式化处理，具体步骤如下：

- 批量预查询文档基础信息，避免数据库多次重复查询。
- 分别处理普通索引类型与父子索引类型文档：
  - 普通文档直接查询DocumentSegment。
  - 父子索引类型文档处理ChildChunk与DocumentSegment关系。
- 将文档统一包装为RetrievalSegments对象返回。

5 重排序

5.1 重排序模块实现

5.1.1 文档去重

为避免重复处理文档，模块首先根据doc_id字段进行去重：

代码块

```
unique_documents = []
doc_ids = set()
for document in documents:
    if document.metadata["doc_id"] not in doc_ids:
        doc_ids.add(document.metadata["doc_id"])
        unique_documents.append(document)
```

5.1.2 关键词相似性计算

关键词相似性使用TF-IDF模型，通过Jieba进行关键词抽取：

- 计算关键词IDF值：

```
代码块
1   idf = math.log((1 + total_documents) / (1 + doc_count_containing_keyword)) + 1
```

- 分别计算查询和文档的TF-IDF向量。
- 使用余弦相似度计算关键词相似性：

```
代码块
1   similarity = cosine_similarity(query_tfidf, document_tfidf)
```

5.1.3 语义向量相似度计算

1. 语义相似度基于文本embedding计算

- 查询文本生成embedding向量：

```
代码块
1   query_vector = cache_embedding.embed_query(query)
```

- 与文档向量计算余弦相似度：

```
代码块
1   cosine_sim = np.dot(vec1, vec2) / (np.linalg.norm(vec1) * np.linalg.norm(vec2))
```

2. 得分融合与排序

根据预定义的权重融合关键词与语义向量相似度：

```
代码块
1   score = (
2       vector_weight * query_vector_score
3       + keyword_weight * query_score
4   )
```

最终根据融合得分对文档进行降序排列，返回前N个文档。

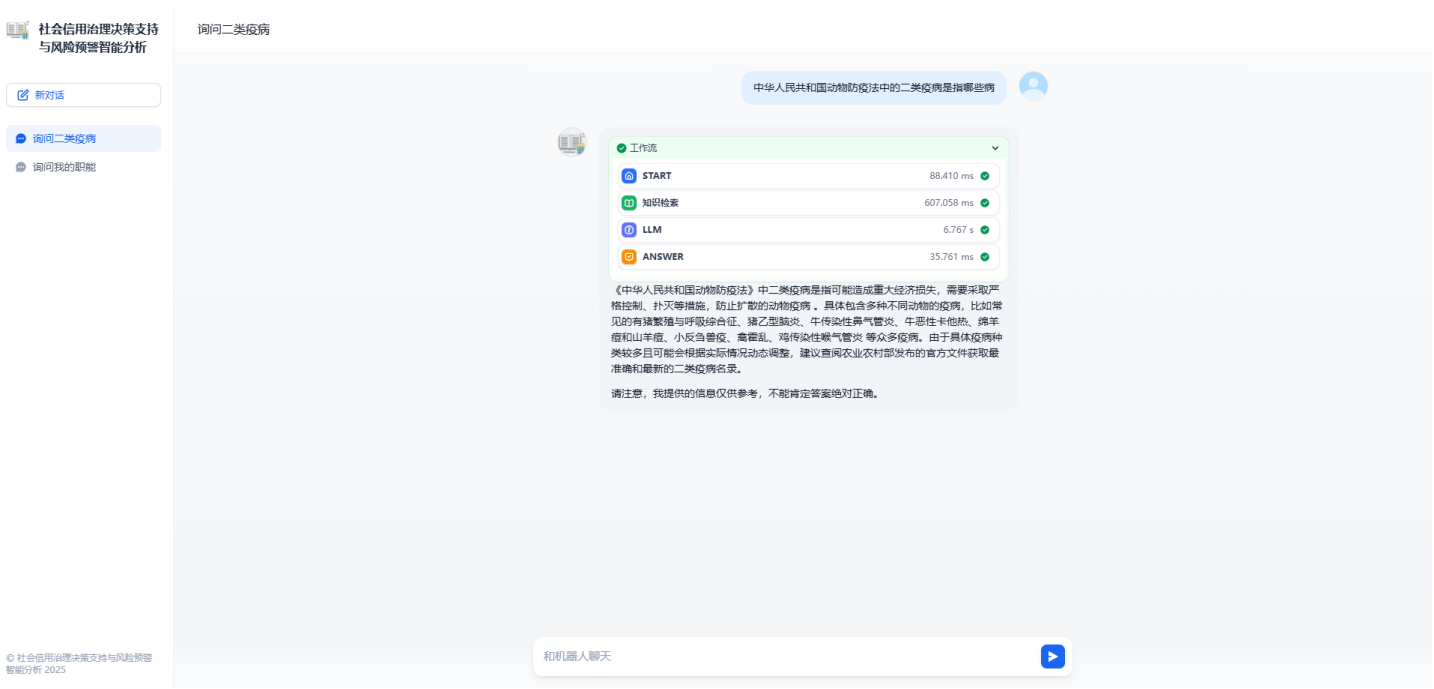
6 大语言模型

使用知识库中检索到的数据通过prompt传递给大语言模型，以下是prompt信息：

代码块

```
你是一个有帮助的法律智能助手。
你叫社会信用治理决策支持与风险预警智能分析助手。
请将以下内容作为你已学习的知识，在<context> XML标签内：
<context>

</context>
回答用户时：
- 如果你不知道，给出建议并说你不能肯定答案的正确性。
- 如果你不确定，向用户请求澄清。
避免提到你从上下文中获得的信息。
并根据用户问题的语言来回答。
```



6.1 页面的实现

6.1.1 状态管理与引用 (Hooks)

代码块

```
const [user, setUser] = useState(uuidv4());  // 用户标识
const [conversationId, setConversationId] = useState('');
const [waiting, setWaiting] = useState(false);  // 控制等待状态
const [maintenance, setMaintenance] = useState(false);  // 维护状态
const [message, setMessage] = useState("");  // 用户输入消息
```

```
6   const [chatItemList, setChatItemList] = useState([]);  // 聊天消息列表
7   const chatListRef = useRef(null);  // 聊天列表容器
8   const chatRef = useRef(null);  // 输入框引用
```

6.1.2 关键函数

1. 获取应用初始配置数据 (getApplicationInformation)

代码块

```
1   async function getApplicationInformation() {
2     const url = '/api/dify/parameters';
3     const response = await fetch(url, { method: 'GET' });
4     return response.json();
5   }
```

2. 添加聊天内容项 (addChatItem)

代码块

```
1   const addChatItem = (newChatItem) => {
2     setChatItemList((prev) => [...prev, newChatItem]);
3   };
```

3. 更新最后一条消息 (updateLastChatItemMessage)

代码块

```
1   const updateLastChatItemMessage = (chunk) => {
2     setChatItemList((prev) => {
3       const updated = [...prev];
4       updated[updated.length - 1]["spinner"] = false;
5       updated[updated.length - 1]["message"] += chunk;
6       return updated;
7     });
8   };
```

4. 判断设备类型 (isMobileDevice)

代码块

```
1   function isMobileDevice() {
2     return /Android|webOS|iPhone|iPad|iPod|BlackBerry|IEMobile|Opera
    Mini/i.test(navigator.userAgent);
```

```
}
```

5. 滚动至页面底部 (scrollToBottom)

代码块

```
const scrollToBottom = () => {
  window.scrollTo({ top: document.documentElement.scrollHeight });
};
```

6. 自动调整输入框大小 (useAutosizeTextArea)

代码块

```
useAutosizeTextArea(chatRef.current, message);
```

6.1.3 消息发送与接收 (sendMessage)

核心功能：向后端发送用户消息，并实时处理流式响应：

代码块

```
const sendMessage = async (directMessage = null) => {
  if ((message || directMessage) && !waiting) {
    const chat_message = {
      inputs: {},
      user: user,
      response_mode: 'streaming',
      query: directMessage || message,
      conversation_id: conversationId
    };
    // 添加用户和助手消息占位符
    addChatItem({ type: 'user', message: directMessage || message });
    addChatItem({ type: 'assistant', spinner: true, message: '' });
    setMessage("");
    if (!isMobileDevice()) chatRef.current.focus();
    setWaiting(true);
    const response = await fetch('/api/dify/chat-messages', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(chat_message)
    });
    const reader = response.body?.getReader();
    if (!reader) return;
    const decoder = new TextDecoder();
```

```
      let buffer = '';
      while (true) {
        const { done, value } = await reader.read();
        if (done) break;
        buffer += decoder.decode(value, { stream: true });
        const lines = buffer.split('\n');
        buffer = lines.pop();
        for (let line of lines) {
          if (line.startsWith('data: ')) {
            try {
              const event_data = JSON.parse(line.slice(6).trim());
              switch (event_data.event) {
                case 'message':
                  updateLastChatItemMessage(event_data.answer);
                  break;
                case 'workflow_started':
                  setConversationId(event_data.conversation_id);
                  break;
                case 'workflow_finished':
                  setWaiting(false);
                  break;
              }
            } catch (e) {
              console.error('[SSE ERROR]', line, e);
            }
          }
        }
      }
    }
  };
```

6.1.4 Effect钩子：页面初始化时加载应用信息

代码块

```
  useEffect(() => {
    chatRef.current.addEventListener('focusin', handleFocusIn);
    chatRef.current.addEventListener('focusout', handleFocusOut);
    (async () => {
      try {
        const appInfo = await getApplicationInformation();
        if (appInfo.status !== undefined) {
          setMaintenance(true);
          return;
        }
        if (appInfo.opening_statement) {
```

```
          addChatItem({
            type: 'assistant',
            spinner: false,
            message: appInfo.opening_statement,
            suggested_questions: appInfo.suggested_questions || []
          });
        }
      } catch (e) {
        console.error('Initialization error:', e);
        setMaintenance(true);
      }
    })();
  }, []);
```

6.1.5 UI核心结构（渲染聊天列表和输入框）

- 显示消息列表，根据类型区分用户与助手消息。
- 输入框绑定message状态，发送按钮调用sendMessage()。

代码块

```
<div ref={chatListRef} className="chat-list overflow-y-auto">
  {chatItemList.map((item, index) => (
    <li key={index}>
      {item.type === 'user' ? (
        // 用户消息UI
        <div>{item.message}</div>
      ) : (
        // 助手消息UI，包括markdown渲染、spinner动画及建议问题
        <ReactMarkdown remarkPlugins={[remarkGfm]}>
          {item.message}
        </ReactMarkdown>
      )}
    </li>
  ))}
</div>
<textarea
  ref={chatRef}
  value={message}
  onChange={handleChange}
/>
<button onClick={() => sendMessage()} disabled={waiting}>
  Send
</button>
```