# 3.1 界面原型设计

## **主窗口：**

## **主窗口是CAT-Desktop-Tool的主体，将四个功能选项组合在一起**

1. 功能导航选项：点击这个按钮，出现四个选项，用户可选择跳 转到项目管理页，翻译编辑页，术语库管理页，翻译记忆库管理页
2. 子功能界面：由用户选择的功能显示相应的功能界面

2.1项目管理界面：默认展示的界面，用于提供项目管理功能

\*新建项目按钮：位于子页面顶端，用于新建空项目；

\*项目记录：位于新建项目按钮下方，分条展示项目信息；

\*删除按钮：位于项目条目中，用户点击后可以删除这一个项 目，并更新显示；

\*进入文档按钮：位与项目条目中，用户点击后可以进入这个项目的文档管理界面；

2.2文档管理界面：提供选中项目中的文档信息，并提供管理功能

\*导入文档按钮：位于子页面顶端，用于导入已有的文件；

\*文档记录：位于导入文档按钮下方，分条展示文档信息；

\*删除按钮：位于文档条目中，用户点击后可以删除这一个项目，并更新显示；

\*进入翻译按钮：位与文档条目中，用户点击后可以进入这个文档的翻译界面；

2.3翻译界面：为用户提供翻译的环境，并提供一些管理功能与翻译帮助功能

2.3.1翻译主界面

\*翻译内容：位于子界面左部最上方，展示段落进度和原文；

\*译文输入框：位于翻译内容正下方，用户在这里输入自己的 译文；

\*翻译状态进度条:位于译文输入框下方，为用户展示文档的翻 译进度；

\*翻译状态选择按钮：分为全部，未翻译，已翻译，需审校四 个状态，用户点击不同的状态则展示不同状态下的原文与译 文；

\*功能选择框：位于子界面左方底部；

\*上一段按钮：使翻译界面的状态跳转到当前 落上一段落 的翻译状态；

\*保存译文按钮：点击后确认保存输入的译文；

\*下一段按钮：使翻译界面的状态跳转到当前 落下一段落 的翻译状态；

\*复制原文按钮：点击后复制原文内容到剪切板；

\*机器翻译按钮：点击后出现机器翻译界面；

\*查询文本输入框：位于子界面右方的顶部由用户输入需要查询得文本；

\* 术语库选择框：展示已有的术语库，由用户选择用于查询的 术语库；

\*翻译记忆库选择框：展示已有的TM库，由用户选择用于查 询的TM库；

\*匹配结果展示：展示精确匹配或模糊匹配的结果；

\*查询按钮：位于子界面右方的底部，点击后可对用户输入的 文本进行匹配；

2.3.2机器翻译子界面

\*语言选择：用户可在下来框中选择源语言与目标语言；

\*内容输入框：供用户输入想要机器翻译的文本；

\*翻译结果展示：展示目标语言的翻译结果；

\*复制翻译：为用户提供一键复制翻译结果的功能；

\*导出为TXT：将翻译结果导出为TXT文件；

2.4术语库管理界面：提供术语库的信息与管理功能

2.4.1库管理界面：展示术语库的记录和对库的操作

\*术语库记录：分条记录术语库名，用户单击后可在右方出现预览；

\*术语库预览：展示术语库名，库中条目的数量，创建时间，用户 描述，创建者等信息；

\*新建按钮：点击后出现创建空的术语库；

\*编辑按钮：点击后可编辑术语预览中的相关信息；

\*删除按钮：点击后可删除选中的术语库；

2.4.2条目管理界面：展示术语库中的条目信息，提供条目管理

\*术语条目记录：记录术语信息

\*编辑按钮：用户点击后可对选中的术语信息进行编辑；

\*删除按钮：用户点击后删除选中的术语条目；

\*保存按钮：用户点击后保存术语条目；

\*新建按钮：用户点击后可新建空的术语条目；

\*导出按钮：用户点击后可将这个属于库中的条目导出我为csv文 件；

\*导入按钮：用户点击后可导入csv文件；

2.5记忆库管理界面：提供记忆库的信息与管理功能

2.5.1库管理界面：展示记忆库的记录和对库的操作

\*记忆库记录：分条记录记忆库名，用户单击后可在右方出现预览；

\*记忆库预览：展示记忆库名，库中条目的数量，创建时间，用户 描述，创建者等信息；

\*新建按钮：点击后出现创建空的记忆库；

\*编辑按钮：点击后可编辑记忆预览中的相关信息；

\*删除按钮：点击后可删除选中的记忆库；

2.5.2条目管理界面：展示记忆库中的条目信息，提供条目管理

\*记忆条目记录：记录记忆信息

\*编辑按钮：用户点击后可对选中的记忆信息进行编辑；

\*删除按钮：用户点击后删除选中的记忆条目；

\*保存按钮：用户点击后保存记忆条目；

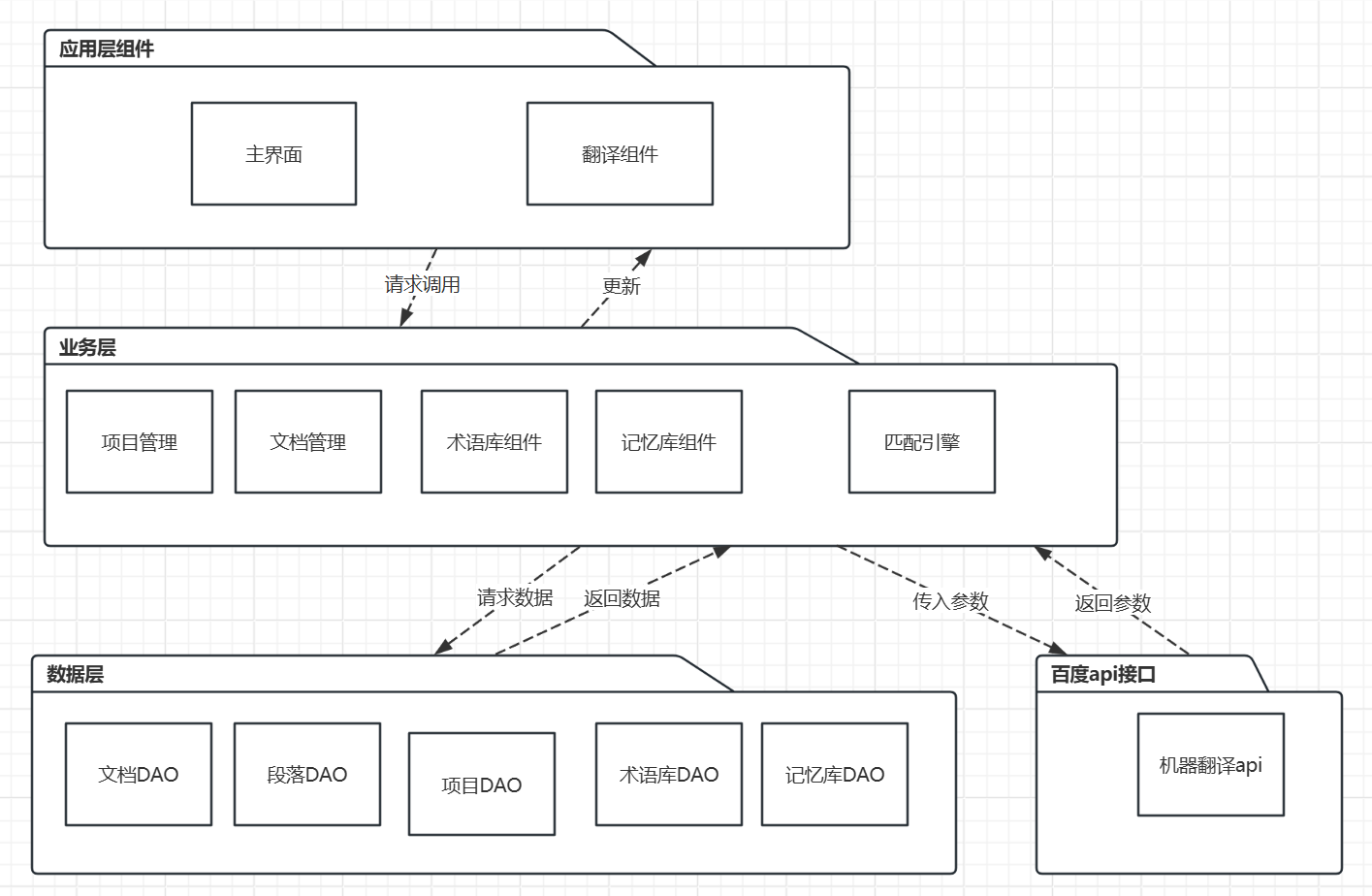
\*新建按钮：用户点击后可新建空的记忆条目；

\*导出按钮：用户点击后可将这个属于库中的条目导出我为csv文 件；

\*导入按钮：用户点击后可导入csv文件；

# 3.2详细设计

3.2.1组件设计



3.2.2接口设计

3.2.2.1数据库接口

1. 文档库接口

* add\_document(project\_id, name, file\_format, status="未翻译")：void：向数据库插入一个新文档
* get\_documents\_by\_project(project\_id):[(id, name, file\_format, status, created\_at), ...]：查询某项目下的所有文档
* delete\_document(doc\_id):void:删除指定文档

1. 项目库接口

* add\_project(name: str, source\_lang: str, target\_lang: str): None：向数据库中添加一个新项目。
* get\_all\_projects(): List[Tuple]：查询数据库中所有项目，按id降序排列并返回项目列表，列表中的每项为元组。
* delete\_project(pid: int): None：根据项目id从数据库中删除指定项目。

3.元表接口

* validate\_table\_type(table\_type: str): bool：验证传入的 table\_type 是否在允许范围内，返回布尔值。
* get\_table\_type\_description(table\_type\_key: str): str：根据英文标识获取对应的中文描述，若不存在则返回“未知类型”。
* create\_table\_description(): None：创建 table\_description 表，用于存储记忆库描述信息，包含表类型字段。
* insert\_table\_description(tm\_name: str, item\_number: int, description: str, owner: int, create\_time: str = None, table\_type: str = None): None：插入一条记忆库描述记录，会验证 table\_type 的合法性。
* get\_record\_count(tm\_name: str): Optional[int]：查询指定翻译记忆库表的记录总数，成功返回整数，失败返回 None。
* update\_table\_description(tm\_name: str, item\_number: int = None, description: str = None, owner: int = None, table\_type: str = None): None：更新指定记忆库描述信息，支持部分字段更新，会验证 table\_type 的合法性。
* delete\_table\_description(tm\_name: str, table\_type: str = None): None：删除指定记忆库描述记录，若传入 table\_type，则同时根据 tm\_name 和 table\_type 删除，会验证 table\_type 的合法性。
* get\_table\_description(tm\_name: str, table\_type: str = None): Optional[dict]：查询指定记忆库描述信息，返回字典或 None，若传入 table\_type，则同时根据 tm\_name 和 table\_type 查询，会验证 table\_type 的合法性。
* list\_table\_descriptions\_by\_type(table\_type: str = None): List[dict]：根据 table\_type 查询记忆库描述信息，返回列表。若 table\_type 为 None 或空字符串，则查询所有记录，会验证 table\_type 的合法性。
* update\_item\_by\_id(item\_id: int, new\_name: str = None, new\_description: str = None): Optional[int]：根据 id 更新 table\_description 表中的记录，支持部分字段更新，返回受影响的行数，出错时返回 None。
* get\_item\_id(tm\_name: str, table\_type: str): Optional[int]：根据 tm\_name 和 table\_type 查询 table\_description 表中的 id，未找到返回 None，会验证 table\_type 的合法性。

4.术语库接口

* sanitize\_table\_name(name: str): str：对表名进行处理，只保留字母、数字和下划线，其他替换为下划线。
* create\_user\_terminology\_table(table\_name: str): bool：创建 terminology 表，包含必要字段，创建成功返回 True，失败返回 False。
* rename\_table(old\_name: str, new\_name: str): bool：更新表名，更新成功返回 True，失败返回 False。
* delete\_terminology\_table(table\_name: str): bool：删除 terminology 表，删除成功返回 True，失败返回 False。
* add\_terminology(term: str, translation: str, table\_name: str, definition: str = None, domain: str = None, project\_id: int = None): int：新增术语条目，返回插入条目的 id。
* export\_terminology(csv\_path: str, table\_name: str): None：导出术语库到 CSV 文件。
* import\_terminology(csv\_path: str, table\_name: str): None：从 CSV 文件导入术语库，支持批量导入。
* get\_terminology\_by\_id(term\_id: int, table\_name: str): Optional[dict]：根据 term\_id 查询术语条目，返回字典或 None。
* update\_terminology(term\_id: int, table\_name: str, term: str = None, translation: str = None, definition: str = None, domain: str = None, project\_id: int = None): bool：更新术语条目，传入 None 的字段不更新，更新成功返回 True，未找到条目返回 False。
* delete\_terminology(term\_id: int, table\_name: str): bool：删除术语条目，删除成功返回 True，否则返回 False。
* list\_terminologies(table\_name: str, project\_id: int = None, domain: str = None): List[dict]：查询术语列表，可根据项目 ID 和领域过滤，返回字典列表。

5.记忆库接口

* sanitize\_identifier(name: str): str：过滤表名或字段名中的非法字符，只保留字母、数字和下划线。
* create\_user\_tm\_table(db\_name: str): str：根据用户输入的库名动态创建翻译记忆表，返回处理后的表名。
* delete\_tm\_table(table\_name: str): None：删除指定表。
* rename\_table(old\_name: str, new\_name: str): bool：更新表名，更新成功返回 True，失败返回 False。
* insert\_tm\_entry(table\_name: str, source\_text: str, target\_text: str, source\_lang: str = None, target\_lang: str = None, created\_by: int = None): Tuple[int, str]：向指定表插入一条翻译记忆条目，返回插入 ID 和创建时间。
* query\_tm\_entries(table\_name: str, keyword: str = None): List[Tuple]：查询指定表中的翻译记忆条目，支持关键字模糊搜索，返回元组列表。
* update\_tm\_entry(table\_name: str, tm\_id: int, source\_text: str = None, target\_text: str = None, source\_lang: str = None, target\_lang: str = None): None：更新指定表中某条翻译记忆条目，支持部分字段更新。
* delete\_tm\_entry(table\_name: str, tm\_id: int): None：删除指定表中某条翻译记忆条目。
* batch\_insert(table\_name: str, rows: Iterable[Tuple]): bool：批量插入数据，插入成功返回 True，出错返回 False。
* batch\_export(table\_name: str): Optional[List[Tuple]]：导出指定翻译记忆表的全部数据到 CSV 文件，返回数据列表，出错或无数据时返回 None。

6.段落库接口

* create\_table(): None：创建翻译段落表，只需运行一次。
* add\_fragment(document\_id: int, seq: int, source\_text: str, status: str = "未翻译"): None：新增一个段落。
* get\_fragments(document\_id: int): List[Tuple]：获取某文档下所有段落。
* update\_fragment(fragment\_id: int, translated\_text: str, status: str): None：更新段落译文和状态。

3.2.2.2主界面接口设计

show\_document\_page(project\_id: int): None ：切换到文档管理页面，并传递 project\_id 。

3.2.2.3翻译界面接口设计

* \_\_init\_\_(document\_id: Optional[int] = None): None ：初始化翻译编辑页面，接收要编辑的文档ID。
* init\_ui(): None ：初始化翻译编辑页面的UI组件。 on\_filter\_changed(): None ：状态过滤切换时，刷新段落列表。 load\_fragments(document\_id: int): None ：读取指定文档的所有段落，支持按状态筛选。
* show\_fragment(): None ：显示当前段落的原文、译文、状态和进度。 refresh\_progress(): None ：统计已翻译段落数与总段落数，刷新进度条。
* save\_translation(): None ：保存当前译文及状态到数据库。 prev\_fragment(): None ：切换到上一段落。
* next\_fragment(): None ：切换到下一段落。
* copy\_source\_text(): None ：将当前段落原文复制到剪贴板，并弹窗提示。
* lookup\_word(): None ：点击机器翻译按钮后，跳转到机器翻译界面。

3.2.2.4术语库管理接口设计

3.2.2.4.1库管理接口设计

|  |  |  |
| --- | --- | --- |
| **类名** | **接口定义** | **功能描述** |
| TermPage | \_\_init\_\_(): None | 初始化术语库管理页面，创建 Widget\_Terminology 实例，为后续的术语库管理操作搭建基础界面框架。 |
| Widget\_Terminology | \_\_init\_\_(): None | 初始化术语库管理界面，加载术语库名称列表，让用户可以直观看到当前系统中的术语库。 |
| Widget\_Terminology | createTerminology(): None | 打开新建术语库对话框，引导用户输入新建术语库的相关信息，并在数据库中创建新的术语库。 |
| Widget\_Terminology | deleteTerminology(): None | 删除选中的术语库，同时会在数据库中移除该术语库的相关数据，释放存储空间。 |
| Widget\_Terminology | editTerminology(): None | 打开编辑术语库对话框，允许用户修改选中术语库的信息，并将修改后的数据更新到数据库中。 |
| Widget\_Terminology | on\_item\_double\_clicked(index: QtCore.QModelIndex): None | 双击术语库条目时，打开术语库条目展示和管理窗口，方便用户对该术语库下的具体条目进行操作。 |
| Widget\_Terminology | on\_item\_clicked(index: QtCore.QModelIndex): None | 单击术语库条目时，显示术语库详细信息，使用户可以快速查看该术语库的基本情况。 |
| CreateTerminologyShowDialog | \_\_init\_\_(text: Optional[dict] = None): None | 初始化新建/编辑术语库对话框，根据传入的参数决定是新建还是编辑术语库操作，并准备好相应的输入界面。 |
| CreateTerminologyShowDialog | get\_data(): dict | 获取用户输入的术语库信息，将用户在对话框中输入的数据整理成字典格式，以便后续保存到数据库。 |
| CreateTerminologyShowDialog | get\_edited\_data(): Optional[dict] | 获取用户编辑后的术语库信息，如果用户进行了编辑操作，则返回编辑后的数据字典；否则返回 None。 |

3.2.2.4.2术语管理接口设计

|  |  |  |
| --- | --- | --- |
| **类名** | **接口定义** | **功能描述** |
| TerminologyItemsShow | \_\_init\_\_(name: str = "术语库详情"): None | 初始化术语库条目展示和管理窗口，加载术语数据，为用户提供一个管理术语库具体条目的界面。 |
| TerminologyItemsShow | createTerminologyitems(): None | 打开新建术语条目对话框，引导用户输入新术语条目的相关信息，并在数据库中创建新的术语条目。 |
| TerminologyItemsShow | editTerminologyitems(): None | 启用选中术语条目的编辑功能，允许用户修改选中术语条目的信息。 |
| TerminologyItemsShow | saveTerminologyitems(): None | 保存编辑后的术语条目，将用户修改后的术语条目数据更新到数据库中。 |
| TerminologyItemsShow | deleteTerminologyitems(): None | 删除选中的术语条目，同时会在数据库中移除该术语条目的相关数据。 |
| TerminologyItemsShow | exportTerminology(): None | 导出术语库到 CSV 文件，方便用户备份或与其他系统进行数据交互。 |
| TerminologyItemsShow | importTerminology(): None | 从 CSV 文件导入术语库，将外部的术语库数据加载到当前系统的数据库中。 |
| NewTerminologyEntryDialog | \_\_init\_\_(parent: Optional[QWidget] = None): None | 初始化新建术语条目对话框，为用户提供输入新术语条目信息的界面。 |

3.2.2.5记忆库管理接口设计

3.2.2.5.1库管理接口设计

|  |  |  |
| --- | --- | --- |
| **类名** | **接口定义** | **功能描述** |
| TMPage | init(): None | 初始化翻译记忆库管理页面，创建 Widget\_TM 实例，为后续的翻译记忆库管理操作搭建基础界面框架。 |
| Widget\_TM | init(): None | 初始化翻译记忆库管理界面，加载 UI 文件，设置窗口标题和大小，连接按钮点击事件，从数据库加载记忆库名称列表。 |
| Widget\_TM | close\_window(): None | 关闭翻译记忆库管理窗口。 |
| Widget\_TM | set\_lineedit\_read(read: bool): None | 设置翻译记忆库描述信息输入框的只读状态。 |
| Widget\_TM | load\_tm\_names\_from\_db(): list[str] | 从数据库的 table\_description 表中获取所有翻译记忆库名称列表。 |
| Widget\_TM | createTM(): None | 打开新建翻译记忆库对话框，根据用户输入创建新的翻译记忆库，并更新数据库和界面显示。 |
| Widget\_TM | editTM(): None | 启用选中的翻译记忆库编辑功能，打开编辑对话框，根据用户输入更新数据库和界面显示。 |
| Widget\_TM | deleteTM(): None | 删除选中的翻译记忆库，更新数据库和界面显示。 |
| Widget\_TM | on\_item\_double\_clicked(index: QtCore.QModelIndex): None | 双击翻译记忆库列表中的条目时，打开该记忆库的详情页面。 |
| Widget\_TM | on\_item\_clicked(index: QtCore.QModelIndex): None | 单击翻译记忆库列表中的条目时，显示该记忆库的描述信息，并设置输入框为只读状态。 |
| CreateTMShowDialog | init(text=None): None | 初始化新建或编辑翻译记忆库对话框，加载 UI 文件，连接按钮点击事件，根据传入的数据初始化输入框内容。 |
| CreateTMShowDialog | get\_data(): dict | 获取用户在对话框中输入的翻译记忆库名称和描述信息。 |
| CreateTMShowDialog | push\_data(data): None | 根据传入的数据初始化对话框的输入框内容。 |
| CreateTMShowDialog | get\_edited\_data(): dict | 获取用户在对话框中编辑后的翻译记忆库名称和描述信息，如果没有修改则返回空。 |

3.2.2.5.2记忆对管理接口设计

|  |  |  |
| --- | --- | --- |
| **类名** | **接口定义** | **功能描述** |
| NewTMEntryDialog | init(current\_user\_id, parent=None): None | 初始化新建翻译记忆条目对话框，设置窗口标题和大小，创建输入框和按钮，连接信号和槽，检测输入文本的语言。 |
| NewTMEntryDialog | detect\_source\_language(): None | 检测源文本输入框中的文本语言，并更新源语言标签。 |
| NewTMEntryDialog | detect\_target\_language(): None | 检测目标文本输入框中的文本语言，并更新目标语言标签。 |
| NewTMEntryDialog | get\_data(): dict | 获取用户在对话框中输入的翻译记忆条目信息。 |
| TM\_items\_show | init(name="记忆库详情"): None | 初始化翻译记忆库条目展示与管理窗口，加载 UI 文件，初始化表格和控件状态，连接信号和槽，加载翻译记忆库数据。 |
| TM\_items\_show | \_load\_ui(): None | 加载翻译记忆库条目展示与管理窗口的 UI 文件。 |
| TM\_items\_show | \_init\_ui(): None | 初始化翻译记忆库条目展示与管理窗口的表格和控件状态。 |
| TM\_items\_show | \_connect\_signals(): None | 连接翻译记忆库条目展示与管理窗口的信号和槽。 |
| TM\_items\_show | closeEvent(event): None | 关闭翻译记忆库条目展示与管理窗口时，更新数据库中的描述信息。 |
| TM\_items\_show | create\_tm\_items(): None | 打开新建翻译记忆条目对话框，根据用户输入创建新的翻译记忆条目，并更新数据库和界面显示。 |
| TM\_items\_show | save\_tm\_items(): None | 保存用户在翻译记忆库条目展示与管理窗口中编辑的翻译记忆条目信息。 |
| TM\_items\_show | edit\_tm\_items(): None | 启用选中的翻译记忆库条目编辑功能。 |
| TM\_items\_show | delete\_tm\_items(): None | 删除选中的翻译记忆库条目，更新数据库和界面显示。 |
| TM\_items\_show | export\_tm\_items(): None | 导出翻译记忆库条目到 CSV 文件。 |
| TM\_items\_show | import\_tm\_items(): None | 从 CSV 文件导入翻译记忆库条目，更新数据库和界面显示。 |
| TM\_items\_show | on\_selection\_changed(): None | 当翻译记忆库条目表格中的选中行变化时，根据选中状态启用或禁用编辑、删除和保存按钮。 |

3.2.2.6项目管理接口设计

* \_\_init\_\_(main\_window: Optional[QMainWindow] = None): None ：初始化项目管理页面，接收主窗口引用用于页面跳转。
* refresh\_table(): None ：刷新项目列表表格。 enter\_document\_page(project\_id: int): None ：跳转到文档管理页面，传递当前项目ID。
* on\_add\_project(): None ：弹出对话框获取新项目信息，并将新项目添加到数据库，最后刷新表格。
* delete\_project(pid: int): None ：根据项目ID删除项目，并刷新表格。

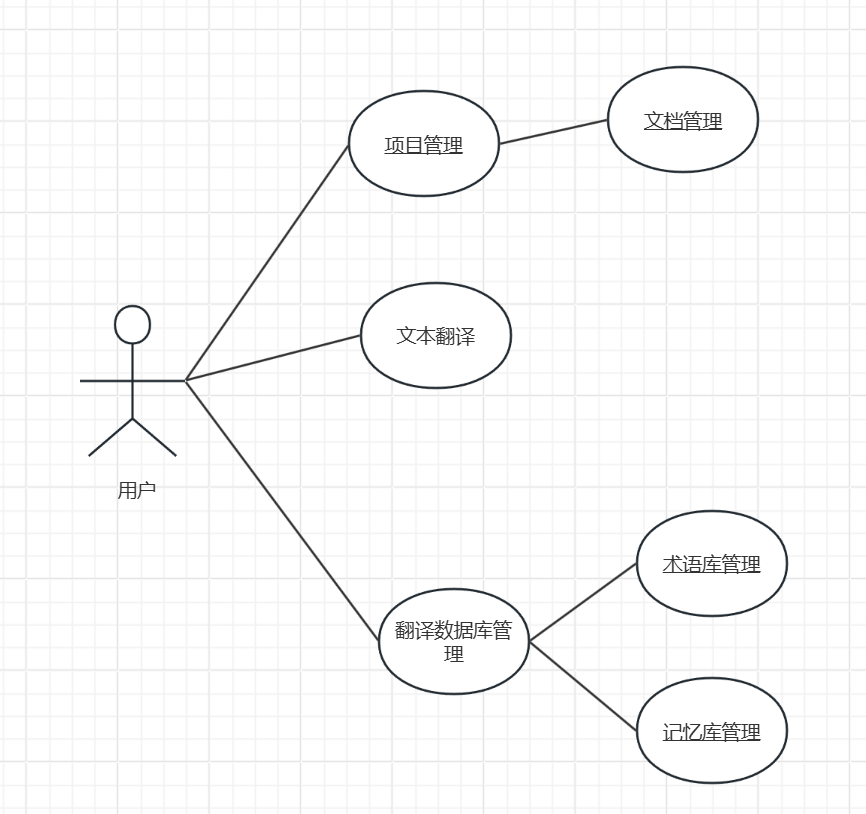
3.2.2.7文档管理接口设计

* \_\_init\_\_(project\_id: int = 1, main\_window: Optional[QMainWindow] = None): None ：初始化文档管理页面，接收项目ID和主窗口引用。 refresh\_table(): None ：刷新文档表格。
* on\_import\_document(): None ：选择文件并导入为文档，同时按段落自动写入 translation\_fragments 表。
* delete\_document(doc\_id: int): None ：删除指定文档，删除前会进行二次确认。
* enter\_translation\_page(document\_id: int): None ：跳转到翻译编辑页面，传递文档ID。

3.2.2.8 API调用接口设计

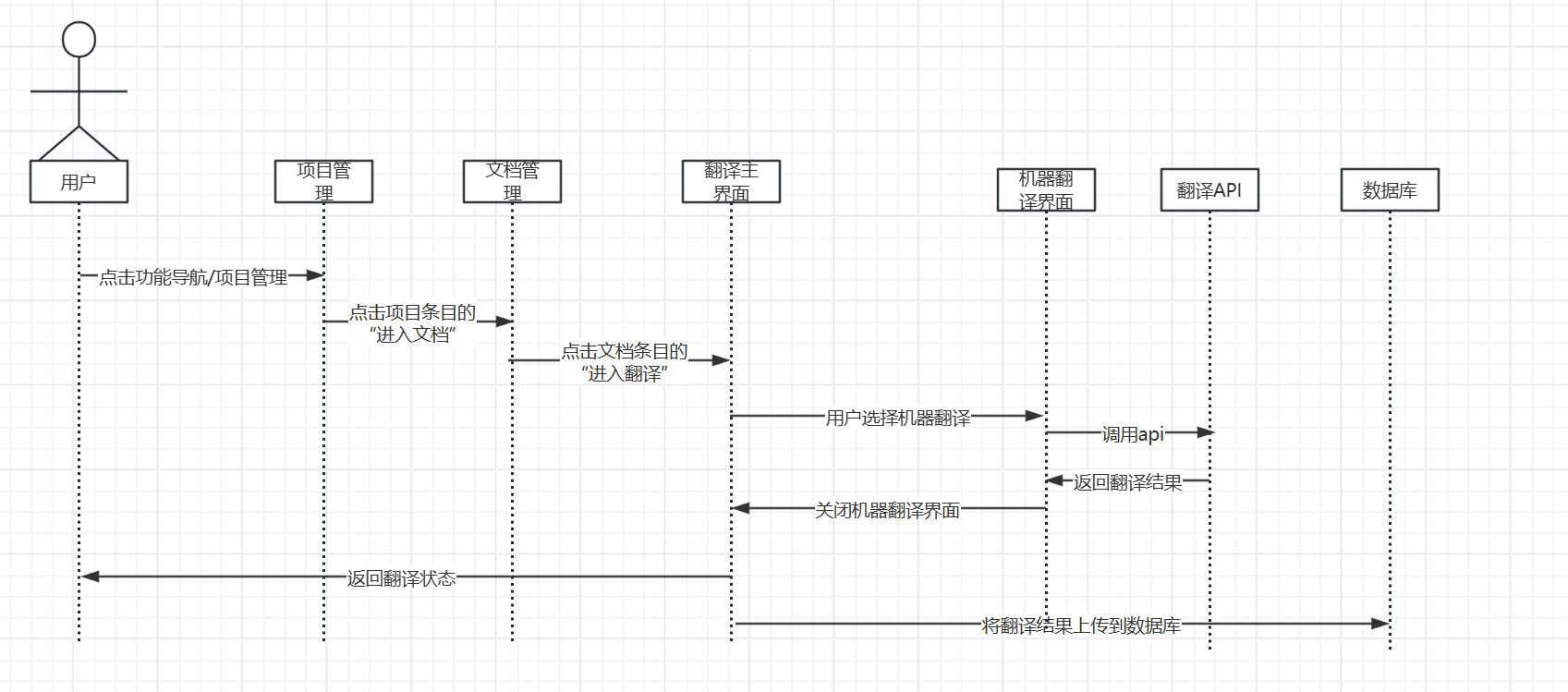
# 3.3系统流程分析

1. **系统流程**

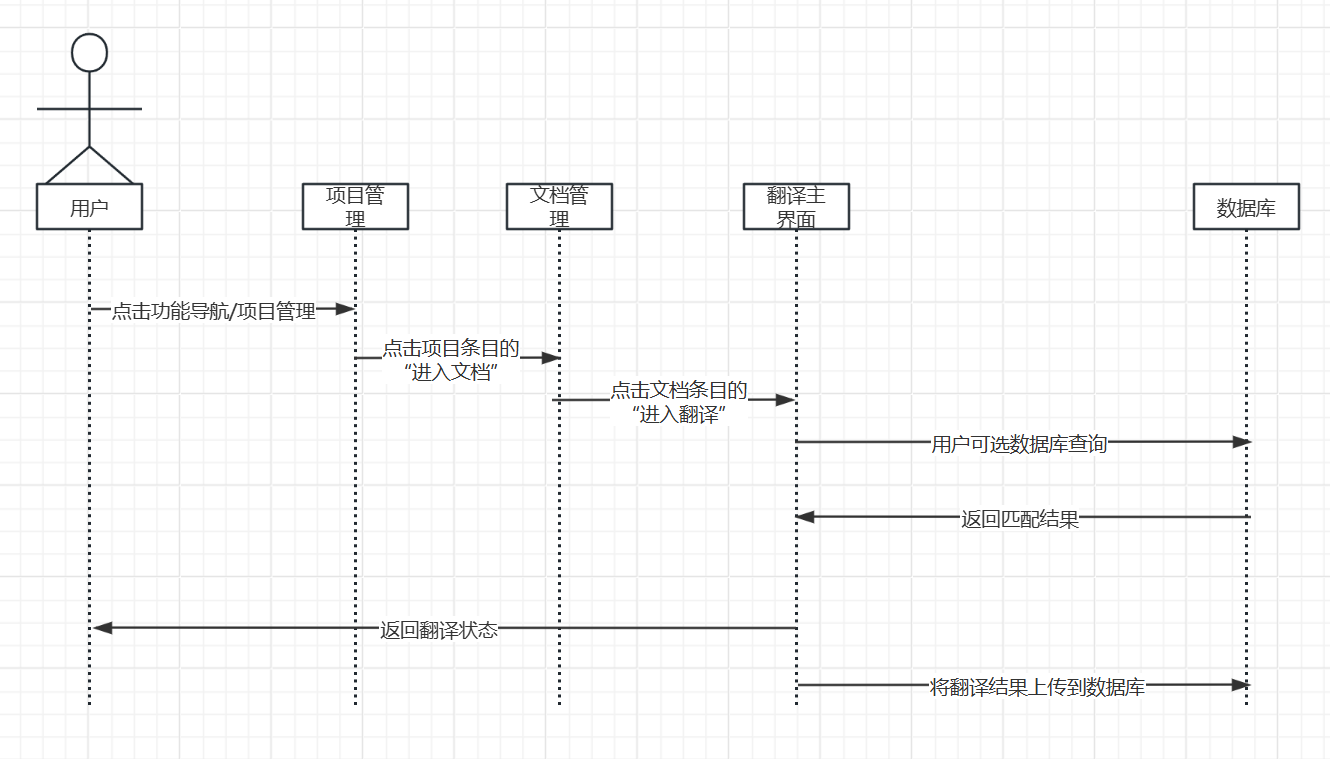


**2.翻译流程**

2.1使用机器翻译辅助翻译

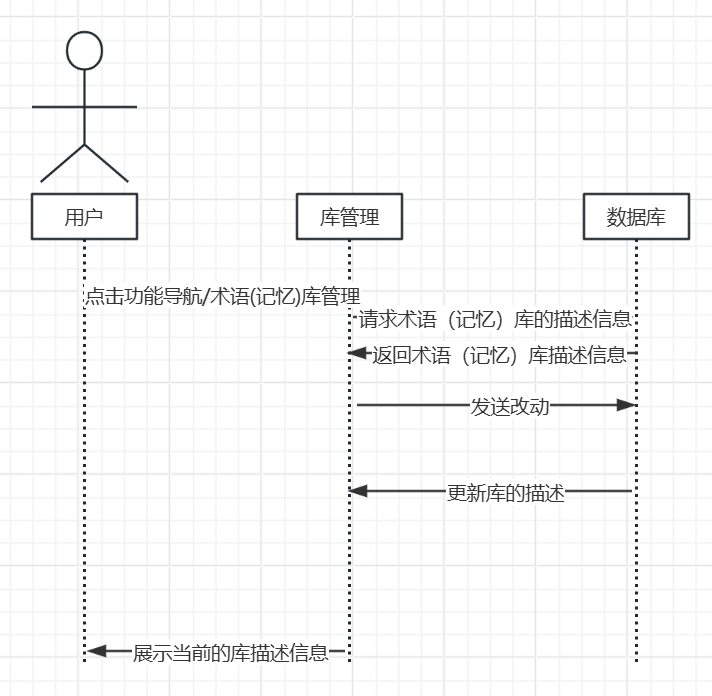


2.2使用术语库记忆库辅助翻译

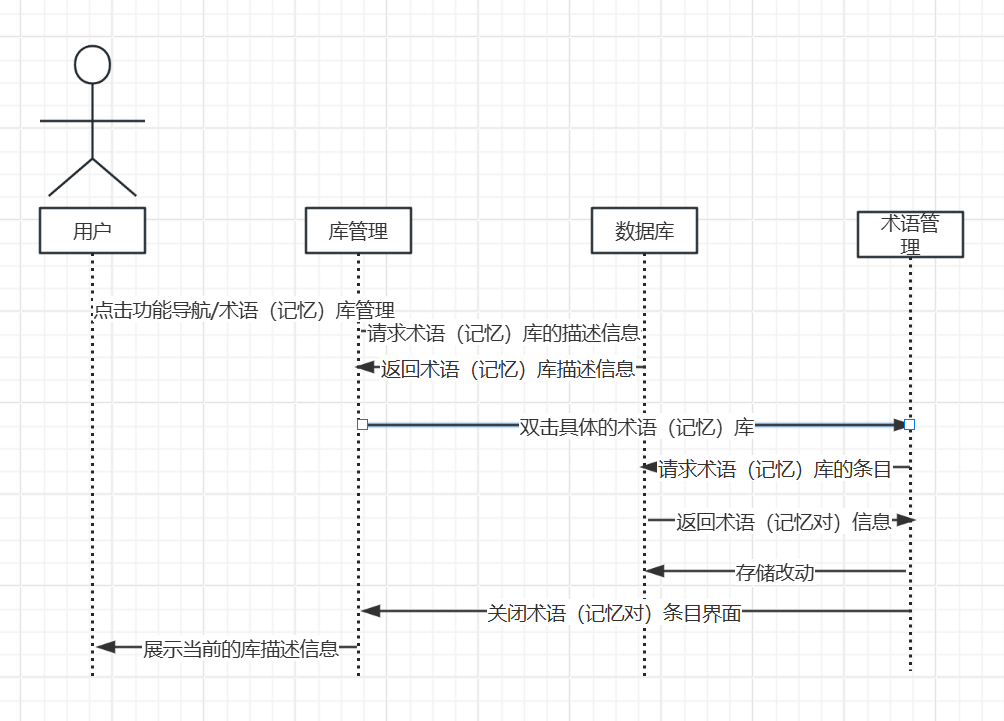


1. **术语库管理流程**

3.1术语库总体管理

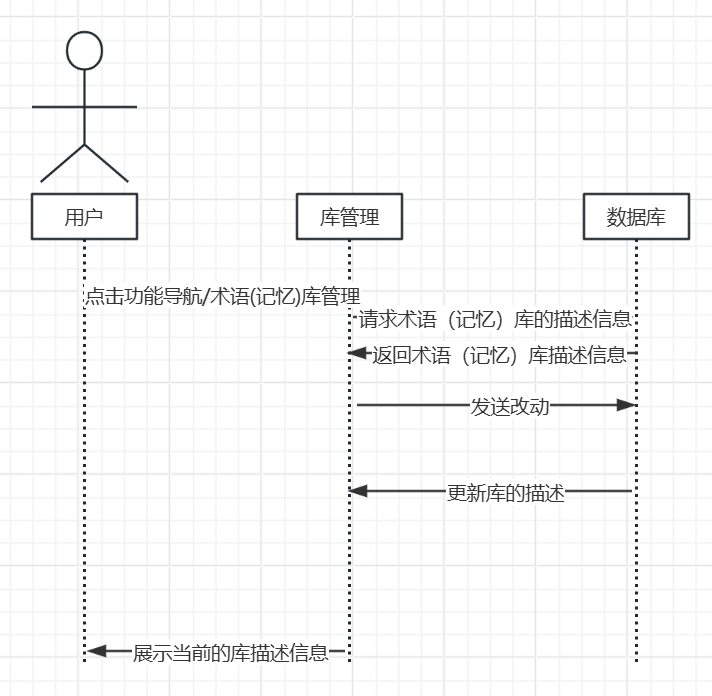


3.2术语条目管理

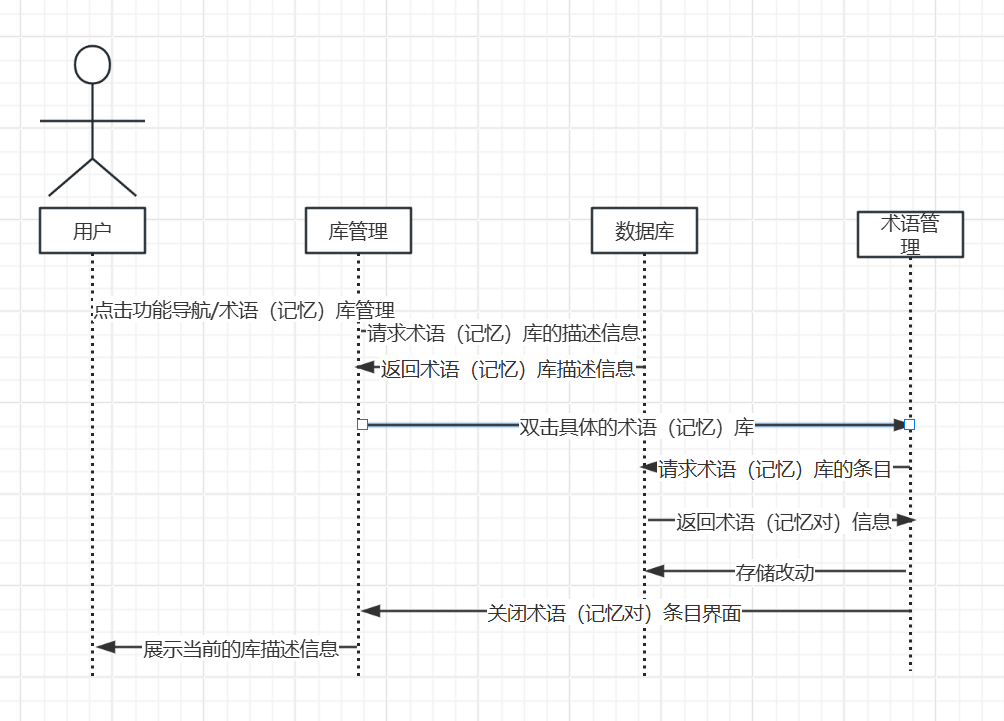


1. **记忆库管理流程**

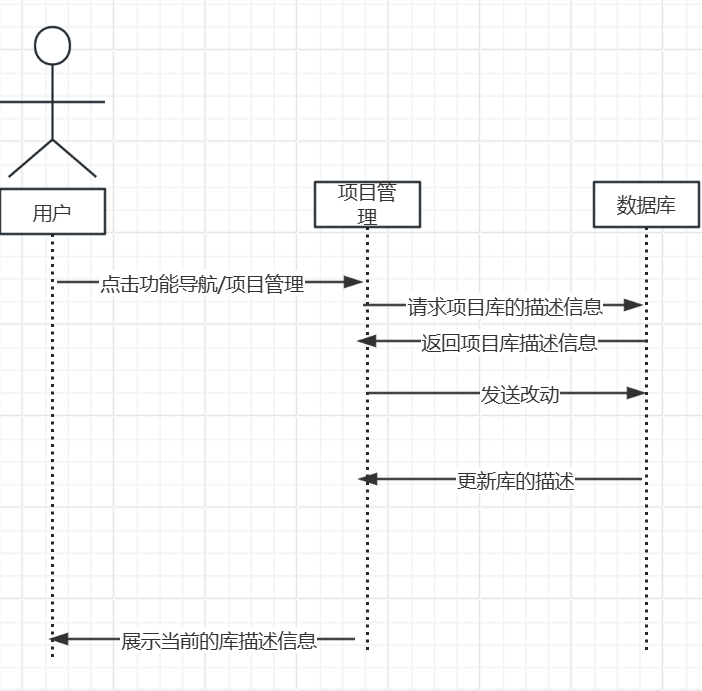
4.1记忆库总体管理

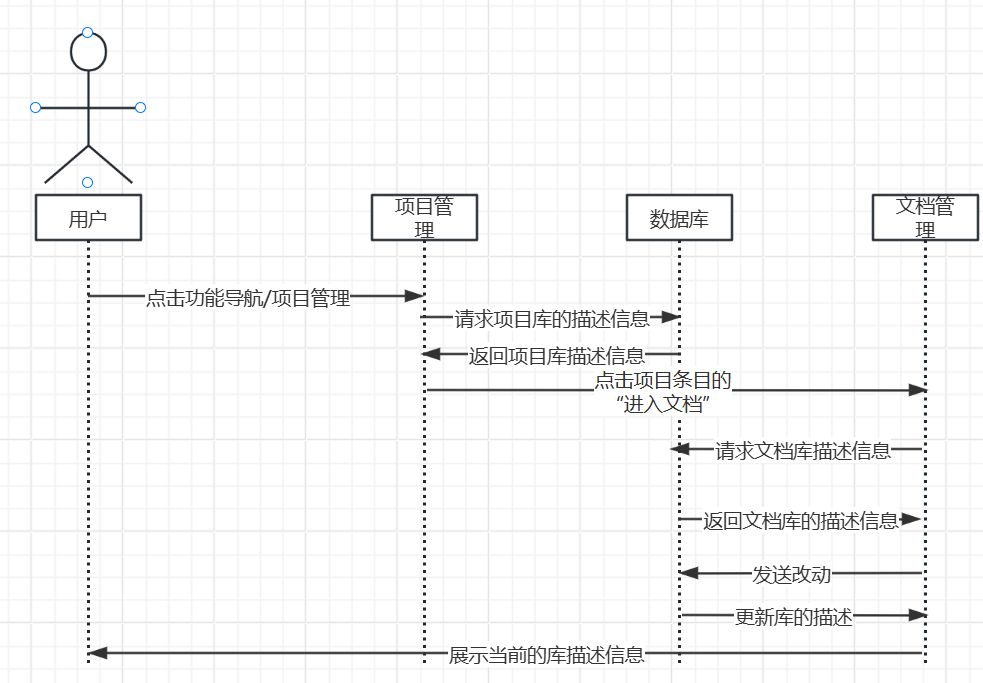


4.2记忆对管理



1. **项目管理**



1. **文档管理**  

# 3.4 模块实现

**3.4.1主界面**

`main\_window.py` 文件实现逻辑分析

1. 文件概述

`main\_window.py` 是CAT翻译工具的主窗口实现文件，负责应用程序的整体界面布局、页面管理和导航控制，基于PyQt5框架构建。

1. 核心类结构

class MainWindow(QMainWindow):

def \_\_init\_\_(self):

# ... 初始化代码 ...

def show\_document\_page(self, project\_id):

# ... 页面切换代码 ...

def show\_translation\_page(self, document\_id):

# ... 页面切换代码 ...

3. 主要功能实现

3.1 窗口初始化

- 设置窗口标题为"CAT 翻译工具"

- 配置初始窗口大小(1000x600)和建议尺寸(1200x900)

- 创建`QStackedWidget`作为中央部件，用于管理多页面切换

3.2 页面管理系统

- 初始化四大核心页面：

- 项目管理页面：`ProjectPage`

- 翻译编辑页面：`TranslationPage`

- 术语库管理页面：`TermPage`

- 记忆库管理页面：`TMPage`

- 通过字典`self.pages`维护页面映射关系

- 使用`QStackedWidget`管理所有页面的添加和切换

3.3 导航菜单实现

- 创建"功能导航"菜单栏

- 为每个页面生成对应的`QAction`

- 通过`triggered`信号连接页面切换逻辑

- 使用lambda表达式传递页面索引参数

3.4 页面切换功能

- `show\_document\_page(project\_id)`：

- 接收项目ID参数

- 创建新的`DocumentPage`实例

- 添加到栈式部件并切换显示

- `show\_translation\_page(document\_id)`：

- 接收文档ID参数

- 创建新的`TranslationPage`实例

- 添加到栈式部件并切换显示

4. 关键技术特点

- \*\*页面切换机制\*\*：采用栈式部件(`QStackedWidget`)实现无刷新页面切换

- \*\*松耦合设计\*\*：主窗口与各功能页面通过ID参数传递实现数据交互

- \*\*动态页面创建\*\*：文档页面和翻译页面采用动态创建方式，确保数据刷新

- \*\*菜单导航系统\*\*：通过菜单栏实现全局功能导航

5. 依赖关系

- \*\*外部依赖\*\*：PyQt5 GUI框架

- \*\*内部依赖\*\*：

- 各页面组件：`ProjectPage`、`DocumentPage`等

- 页面间数据传递：通过ID参数实现

# main\_window.py

import sys

from PyQt5.QtWidgets import QApplication, QMainWindow, QStackedWidget, QAction

from pages.project\_page import ProjectPage

from pages.document\_page import DocumentPage

from pages.translation\_page import TranslationPage

from pages.term\_page import TermPage

from pages.tm\_page import TMPage

class MainWindow(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("CAT 翻译工具")

self.setGeometry(100, 100, 1000, 600)

self.resize(1200, 900) # 建议宽1200像素，高900像素

self.stack = QStackedWidget()

self.setCentralWidget(self.stack)

# 先加载所有主页面

self.project\_page = ProjectPage(self)

self.translation\_page = TranslationPage()

self.term\_page = TermPage()

self.tm\_page = TMPage()

self.pages = {

"项目管理": self.project\_page,

"翻译编辑": self.translation\_page,

"术语库管理": self.term\_page,

"记忆库管理": self.tm\_page

}

for page in self.pages.values():

self.stack.addWidget(page)

# 菜单栏

menu = self.menuBar().addMenu("功能导航")

for idx, (name, page) in enumerate(self.pages.items()):

action = QAction(name, self)

action.triggered.connect(lambda \_, i=idx: self.stack.setCurrentIndex(i))

menu.addAction(action)

def show\_document\_page(self, project\_id):

"""

切换到文档管理页面，并传递project\_id

"""

# 每次都新建一个文档页面（保证刷新内容），并切换到它

doc\_page = DocumentPage(project\_id, main\_window=self)

self.stack.addWidget(doc\_page)

self.stack.setCurrentWidget(doc\_page)

def show\_translation\_page(self, document\_id):

"""

切换到翻译编辑页面，并传递文档ID

"""

page = TranslationPage(document\_id)

self.stack.addWidget(page)

self.stack.setCurrentWidget(page)

if \_\_name\_\_ == "\_\_main\_\_":

app = QApplication(sys.argv)

win = MainWindow()

win.show()

sys.exit(app.exec\_())

**3.4.2翻译界面**

`translation\_page.py` 文件实现逻辑分析

1. 文件概述

`translation\_page.py` 实现了CAT翻译工具的核心翻译编辑功能，提供文档段落翻译、进度跟踪、模糊匹配参考等完整翻译工作流支持，基于PyQt5构建用户界面。

2. 核心类结构

```python:absolute/path/to/translation\_page.py

class TranslationPage(QWidget):

def \_\_init\_\_(self, document\_id=None):

# ... 初始化代码 ...

def init\_ui(self):

# ... UI初始化代码 ...

def load\_fragments(self, document\_id):

# ... 加载文档段落 ...

def show\_fragment(self):

# ... 显示当前段落 ...

def save\_translation(self):

# ... 保存译文 ...

# 其他核心方法 ...

```

3. 主要功能实现

3.1 翻译编辑界面布局

采用左右分栏设计：

- \*\*左侧翻译区\*\*（占比3/5）：

- 段落进度指示器（`progress\_label`）

- 原文显示区（支持自动换行）

- 译文编辑区（`QTextEdit`）

- 翻译状态标签和进度条

- 状态筛选下拉框（全部/未翻译/已翻译/需审校）

- 功能按钮组（上一段/保存译文/下一段/复制原文/机器翻译）

- \*\*右侧参考区\*\*（占比2/5）：

```python:absolute/path/to/translation\_page.py

# 右边：模糊匹配显示区

self.fuzzy\_match\_widget = test\_demo\_show\_fuzzy\_match()

main\_layout.addWidget(self.fuzzy\_match\_widget, stretch=2)

```

集成翻译记忆库模糊匹配结果展示组件

##### 3.2 文档段落管理

- \*\*数据加载流程\*\*：

通过`translation\_fragment\_dao`从数据库获取段落数据，并支持按翻译状态筛选

- \*\*段落导航机制\*\*：

- 维护`current\_index`跟踪当前编辑段落

- `prev\_fragment()`/`next\_fragment()`实现段落切换

- 自动控制导航按钮可用性（首段禁用上一段，末段禁下一段）

3.3 翻译进度跟踪

- \*\*实时进度计算\*\*：

```python:absolute/path/to/translation\_page.py

def refresh\_progress(self):

all\_count = len(self.fragments)

done\_count = sum(1 for f in self.fragments if f[4] == "已翻译")

percent = int(done\_count / all\_count \* 100)

self.progress\_bar.setValue(percent)

```

- 进度条显示已翻译段落占比，进度标签显示当前段落位置

4. 数据交互流程

1. \*\*数据加载\*\*：`document\_id` → `translation\_fragment\_dao.get\_fragments()` → `self.fragments`缓 存

2. \*\*数据展示\*\*：`self.fragments` → `show\_fragment()` → UI组 件更新

3. \*\*数据保存\*\*：用户输入 → `save\_translation()` → `translation\_fragment\_dao.update\_fragment()` → 数据库更新

5. 关键技术特点

- \*\*状态驱动更新\*\*：筛选条件变化自动触发数据重新加载和界面刷新

- \*\*本地缓存机制\*\*：段落数据本地缓存减少数据库访问，提升响应速度

- \*\*松耦合设计\*\*：通过DAO层与数据库交互，通过组件化方式集成模糊匹配功能

- \*\*用户体验优化\*\*：提供复制原文、进度提示、操作反馈等辅助功能

6. 依赖关系

- \*\*内部依赖\*\*：

- 数据访问：`translation\_fragment\_dao`（数据库操作）

- 模糊匹配：`test\_demo\_show\_fuzzy\_match`（翻译记忆库查询）

- 机器翻译：`api\_ui\_.TranslatorApp`（外部翻译接口）

- \*\*外部依赖\*\*：PyQt5（UI框架）、sys/os（系统功能）

# src/pages/translation\_page.py

from PyQt5.QtWidgets import (

QWidget, QVBoxLayout, QLabel, QTextEdit, QPushButton, QHBoxLayout, QMessageBox

)

from PyQt5.QtCore import Qt

from PyQt5.QtWidgets import QProgressBar

from PyQt5.QtWidgets import QComboBox

import sys

import os

import api\_ui

from Translation.ui\_select\_tmAndterminology import test\_demo\_show\_fuzzy\_match

from api\_ui import api\_ui\_

sys.path.append(os.path.join(os.path.dirname(\_\_file\_\_), '..'))

from dao import translation\_fragment\_dao

class TranslationPage(QWidget):

def \_\_init\_\_(self, document\_id=None):

"""

:param document\_id: 要编辑的文档ID

"""

super().\_\_init\_\_()

self.setWindowTitle("翻译编辑页面")

self.document\_id = document\_id

self.fragments = [] # 当前文档所有段落

self.current\_index = 0 # 当前段落下标

self.init\_ui()

if document\_id is not None:

self.load\_fragments(document\_id)

def init\_ui(self):

# 创建主水平布局，左右分栏

main\_layout = QHBoxLayout()

self.setLayout(main\_layout)

# 左边：翻译编辑区，放到一个单独的 QWidget 里

self.left\_widget = QWidget()

left\_layout = QVBoxLayout()

self.left\_widget.setLayout(left\_layout)

# 段落进度标签

self.progress\_label = QLabel("段落进度：0/0")

left\_layout.addWidget(self.progress\_label)

# 原文显示标签

self.source\_label = QLabel("原文内容：")

self.source\_label.setWordWrap(True)

left\_layout.addWidget(self.source\_label)

# 译文输入区

left\_layout.addWidget(QLabel("请输入译文："))

self.text\_edit = QTextEdit()

left\_layout.addWidget(self.text\_edit)

# 状态显示标签

self.status\_label = QLabel("当前状态：")

left\_layout.addWidget(self.status\_label)

# 翻译进度条

self.progress\_bar = QProgressBar()

self.progress\_bar.setMinimum(0)

self.progress\_bar.setMaximum(100)

self.progress\_bar.setValue(0)

self.progress\_bar.setFormat("翻译进度：%p%")

left\_layout.addWidget(self.progress\_bar)

# 状态过滤下拉框

self.filter\_box = QComboBox()

self.filter\_box.addItems(["全部", "未翻译", "已翻译", "需审校"])

self.filter\_box.currentIndexChanged.connect(self.on\_filter\_changed)

left\_layout.addWidget(self.filter\_box)

# 按钮区

btn\_layout = QHBoxLayout()

self.btn\_prev = QPushButton("上一段")

self.btn\_prev.clicked.connect(self.prev\_fragment)

btn\_layout.addWidget(self.btn\_prev)

self.btn\_save = QPushButton("保存译文")

self.btn\_save.clicked.connect(self.save\_translation)

btn\_layout.addWidget(self.btn\_save)

self.btn\_next = QPushButton("下一段")

self.btn\_next.clicked.connect(self.next\_fragment)

btn\_layout.addWidget(self.btn\_next)

self.btn\_copy\_source = QPushButton("复制原文")

self.btn\_copy\_source.clicked.connect(self.copy\_source\_text)

btn\_layout.addWidget(self.btn\_copy\_source)

self.btn\_lookup = QPushButton("机器翻译")

self.btn\_lookup.clicked.connect(self.lookup\_word)

btn\_layout.addWidget(self.btn\_lookup)

left\_layout.addLayout(btn\_layout)

# 把左边控件添加到主布局

main\_layout.addWidget(self.left\_widget, stretch=3) # stretch参数控制宽度比例

# 右边：模糊匹配显示区

self.fuzzy\_match\_widget = test\_demo\_show\_fuzzy\_match()

main\_layout.addWidget(self.fuzzy\_match\_widget, stretch=2)

def on\_filter\_changed(self):

"""

状态过滤切换，刷新段落列表

"""

self.current\_index = 0 # 重新回到第1段

self.load\_fragments(self.document\_id)

def load\_fragments(self, document\_id):

"""

读取指定文档的所有段落，支持按状态筛选

"""

all\_frags = translation\_fragment\_dao.get\_fragments(document\_id)

filter\_status = self.filter\_box.currentText()

if filter\_status == "全部":

self.fragments = all\_frags

else:

self.fragments = [frag for frag in all\_frags if frag[4] == filter\_status]

if not self.fragments:

QMessageBox.warning(self, "无内容", "该文档暂无分段或筛选结果为空。")

self.current\_index = 0

self.show\_fragment()

return

self.current\_index = 0

self.show\_fragment()

self.refresh\_progress()

def show\_fragment(self):

"""

显示当前段落（原文+译文+状态+进度）

"""

total = len(self.fragments)

idx = self.current\_index

self.progress\_label.setText(f"段落进度：{idx+1}/{total}")

if total == 0:

self.source\_label.setText("原文内容：无")

self.text\_edit.setText("")

self.status\_label.setText("当前状态：无")

return

frag = self.fragments[idx]

frag\_id, seq, source\_text, translated\_text, status = frag

self.source\_label.setText(f"原文内容（第{seq}段）：{source\_text}")

self.text\_edit.setText(translated\_text if translated\_text else "")

self.status\_label.setText(f"当前状态：{status if status else '未翻译'}")

# 按钮可用性

self.btn\_prev.setEnabled(idx > 0)

self.btn\_next.setEnabled(idx < total - 1)

self.refresh\_progress()

def refresh\_progress(self):

"""

统计已翻译/总段数，刷新进度条

"""

all\_count = len(self.fragments)

if all\_count == 0:

self.progress\_bar.setValue(0)

return

done\_count = sum(1 for f in self.fragments if f[4] == "已翻译")

percent = int(done\_count / all\_count \* 100)

self.progress\_bar.setValue(percent)

def save\_translation(self):

"""

保存当前译文及状态到数据库

"""

if not self.fragments:

return

frag = self.fragments[self.current\_index]

frag\_id = frag[0]

text = self.text\_edit.toPlainText().strip()

if not text:

QMessageBox.warning(self, "保存失败", "请输入译文后再保存！")

return

status = "已翻译" if text else "未翻译"

translation\_fragment\_dao.update\_fragment(frag\_id, text, status)

# 更新本地缓存

lst = list(self.fragments[self.current\_index])

lst[3] = text

lst[4] = status

self.fragments[self.current\_index] = tuple(lst)

QMessageBox.information(self, "保存成功", "译文已保存。")

self.show\_fragment()

def prev\_fragment(self):

"""

切换到上一段

"""

if self.current\_index > 0:

self.current\_index -= 1

self.show\_fragment()

def next\_fragment(self):

"""

切换到下一段

"""

if self.current\_index < len(self.fragments) - 1:

self.current\_index += 1

self.show\_fragment()

def copy\_source\_text(self):

"""

将当前段落原文复制到剪贴板，并弹窗提示

"""

text = self.source\_label.text()

# 去除前缀（如“原文内容（第x段）：”）只保留原文

if "：" in text:

text = text.split("：", 1)[1]

from PyQt5.QtWidgets import QApplication

QApplication.clipboard().setText(text)

QMessageBox.information(self, "提示", "原文已复制到剪贴板！")

def lookup\_word(self):

"""

机器翻译按钮，点击后跳转到机器翻译界面

"""

try:

self.translation\_app = api\_ui.api\_ui\_. TranslatorApp()

self.translation\_app.show()

except Exception as e:

print(e)

**3.4.2.1机器翻译实现**

**3.4.2.2翻译数据辅助翻译实现**

1. 文件概述

- Fuzzy\_match.py ：提供基于字符串相似度的模糊匹配算法，支持从翻译记忆库中检索相似句段

- ui\_select\_tmAndterminology.py ：提供可视化界面，支持用户选择术语库/翻译记忆库，输入查询文本并展示匹配结果

2. Fuzzy\_match.py 核心实现 2.1 模糊匹配算法

```

def fuzzy\_match(query, sentences, threshold=0.0):

    results = []

    for sentence in sentences:

        # 使用difflib计算字符串相似度

        ratio = difflib.SequenceMatcher(None, query,

        sentence).ratio()

        results.append((sentence, ratio))

    # 按相似度降序排序

    results.sort(key=lambda x: x[1], reverse=True)

    # 过滤低于阈值的结果

    filtered = [r for r in results if r[1] >= threshold]

    return filtered

```

- 核心技术 ：采用 difflib.SequenceMatcher 计算文本相似度（0-1之间）

- 处理流程 ：计算相似度→排序→阈值过滤

- 参数说明 ：

- query ：查询文本

- sentences ：待匹配句子列表

- threshold ：相似度阈值（默认0.0，返回所有结果） 2.2 Fuzzy\_match类实现

```

class Fuzzy\_match:

    def \_\_init\_\_(self, query\_text=None, memories=None):

        self.text = query\_text or "测试句子"

        self.memories = memories  # 外部传入的记忆库列表

    def fuzzy\_match\_table\_terms(self, threshold=0.3):

        # 从数据库查询记忆库条目并进行匹配

        try:

            all\_source\_texts = []

            source\_text\_map = []

            # ... 数据库查询与数据处理 ...

            matches = fuzzy\_match(self.text,

            all\_source\_texts, threshold)

            # ... 结果映射与格式化 ...

            return results

        except Exception as e:

            print(f"fuzzy\_match\_table\_terms error: {e}")

            return []

```

- 核心功能 ：从指定翻译记忆库中查询并匹配相似句段

- 数据流程 ：

1. 通过 TM\_dao.query\_tm\_entries() 获取记忆库数据

2. 构建源文本列表与映射关系

3. 调用 fuzzy\_match() 函数执行匹配

4. 格式化结果并返回（包含库名、ID、原文、译文、相似度）

3. ui\_select\_tmAndterminology.py

核心实现

3.1 界面布局设计

```

class test\_demo\_show\_fuzzy\_match(QWidget):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.setWindowTitle("测试匹配结果界面")

        self.resize(800, 600)

        # 查询输入区（多行文本框）

        self.input\_edit = QTextEdit()

        self.input\_edit.setPlaceholderText("请输入查询目标文

        本")

        # 术语库/记忆库选择区（多选择列表）

        self.terms\_list = QListWidget()  # 术语库列表

        self.memories\_list = QListWidget()  # 记忆库列表

        # 结果显示区

        self.label = QLabel("匹配结果将在这里显示")

        # 查询按钮

        self.button = QPushButton("查询")

        self.button.clicked.connect(self.on\_query\_clicked)

```

- 界面结构 ：上中下布局

- 上部：查询文本输入框（支持多行输入）

- 中部：左右分栏的术语库/记忆库选择列表（支持多选）

- 下部：查询按钮与结果显示区 3.2 核心交互逻辑

查询处理流程 ：

```

def on\_query\_clicked(self):

    query\_text = self.input\_edit.toPlainText().strip()

    if not query\_text:

        QMessageBox.warning(self, "输入错误", "请输入查询目标

        文本！")

        return

    # 获取用户选择

    selected\_terms = [(item.text(), item.data(Qt.

    UserRole)) for item in self.terms\_list.selectedItems

    ()]

    selected\_memories = [(item.text(), item.data(Qt.

    UserRole)) for item in self.memories\_list.

    selectedItems()]

    # 选择校验（术语库和记忆库只能选一个）

    if selected\_terms and selected\_memories:

        QMessageBox.warning(self, "选择错误", "术语库和翻译记

        忆库只能选择其中一个！")

        return

    # 执行匹配

    if selected\_memories:

        # 翻译记忆库 - 模糊匹配

        match = Fuzzy\_match.Fuzzy\_match(query\_text,

        selected\_memories)

        text = match.fuzzy\_match\_table\_terms(0.3)

        # ... 结果显示 ...

    elif selected\_terms:

        # 术语库 - 精确匹配

        matched\_terms\_all = []

        for name, \_ in selected\_terms:

            matched\_terms = self.exact\_match\_terms\_in\_text

            (query\_text, name)

            matched\_terms\_all.extend(matched\_terms)

        # ... 结果显示 ...

``` 3.3 术语精确匹配实现

```

def exact\_match\_terms\_in\_text(self, input\_text,

table\_name):

    """

    在输入文本中查找术语库中所有术语（term），做精确子串匹配

    返回匹配的术语条目列表

    """

    matched\_terms = []

    terms = list\_terminologies(table\_name)  # 从术语库获取所

    有术语

    for term\_entry in terms:

        term\_text = term\_entry['term']

        if term\_text and term\_text in input\_text:

            matched\_terms.append(term\_entry)

    return matched\_terms

```

- 匹配逻辑 ：判断术语是否为输入文本的子串

- 数据来源 ：通过 Terminology\_dao.list\_terminologies() 获取术语数据

- 返回结果 ：包含术语、译文、定义的完整术语条目

4. 模块交互关系 4.1 调用关系图

4.2 数据流程

1. 初始化阶段 ：

- UI通过 table\_description\_dao 加载术语库/记忆库列表

- 列表项存储显示名称与数据库ID（通过 Qt.UserRole ）

2. 查询阶段 ：

- 用户输入查询文本并选择库

- UI验证输入和选择合法性

- 根据选择类型调用不同匹配方式：

- 记忆库 ：实例化 Fuzzy\_match 类，调用 fuzzy\_match\_table\_terms()

- 术语库 ：调用 exact\_match\_terms\_in\_text() 执行精确匹配

- 匹配结果格式化后显示在UI

**3.4.3术语库管理**

**3.4.3.1库管理**

代码实现：

1. 核心类结构

该文件包含两个主要类，实现术语库的创建、管理和展示功能：

- CreateTerminologyShowDialog ：术语库创建/编辑对话框

- Widget\_Terminology ：术语库管理主界面组件

1. CreateTerminologyShowDialog 对话框
2. Widget\_Terminology 主界面

初始化流程 ：

1 加载UI文件 Terminology\_manage\_show.ui

2 设置按钮事件绑定（创建/删除/编辑术语库）

3 初始化列表视图，加载术语库名称

核心功能实现 ：

1 术语库列表管理

2 术语库创建流程

2.1 点击创建按钮弹出 CreateTerminologyShowDialog

2.2. 验证输入（名称非空、不重复）

2.3 调用DAO层创建数据库表和描述记录

3术语库删除/编辑

- 删除 ：验证选中状态 → 确认对话框 → 调用DAO删除表和描 述

- 编辑 ：获取选中项数据 → 弹出编辑对话框 → 更新数据库记 录

4 术语库详情展示

- 单击列表项：在右侧面板显示术语库元数据（名称、数量、创 建时间等）

- 双击列表项：打开 TerminologyItemsShow 窗口查看术语库条目 详情

import sys

import os

from PyQt5 import QtCore, QtGui, QtWidgets ,uic

from PyQt5.QtWidgets import QMessageBox

from dao import Terminology\_dao, table\_description\_dao

from 术语库 import Terminology\_show # 你已有的显示模块

class CreateTerminologyShowDialog(QtWidgets.QDialog ):

def \_\_init\_\_(self , text = None):

super().\_\_init\_\_()

self.original\_data = None

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

ui\_path = os.path.join(current\_dir, "Create\_Terminology.ui")

uic.loadUi(ui\_path, self)

self.Create\_Terminology\_sure.clicked.connect(self.accept)

self.Create\_Terminology\_Cancle.clicked.connect(self.reject)

self.push\_data(text)

def get\_data(self):

#获取用户输入的数据，以字典的形式返回

return {

"Terminology\_name": self.lineEdit\_Terminology\_name.text().strip(),

"description": self.textEdit\_Terminology\_descripte.toPlainText().strip()

}

def push\_data(self, data):

if data is None:

return

self.original\_data = data.copy() # 备份原始数据

self.lineEdit\_Terminology\_name.setText(data["Terminology\_name"])

self.textEdit\_Terminology\_descripte.setText(data["description"])

def get\_edited\_data(self):

current\_name = self.lineEdit\_Terminology\_name.text()

current\_description = self.textEdit\_Terminology\_descripte.toPlainText()

if self.original\_data is None:

return None # 没有原始数据，无法判断

# 比较是否有修改

if (current\_name != self.original\_data["Terminology\_name"] or

current\_description != self.original\_data["description"]):

# 返回修改后的结果

return {

"Terminology\_name": current\_name,

"description": current\_description

}

else:

# 没有修改，返回空

return None

class Widget\_Terminology(QtWidgets.QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

ui\_path = os.path.join(current\_dir, "Terminology\_manage\_show.ui")

uic.loadUi(ui\_path, self)

self.resize(800, 600)

self.setWindowTitle("术语库管理")

self.pushButton\_createTerminology.clicked.connect(self.createTerminology)

self.pushButton\_deleteTerminology.clicked.connect(self.deleteTerminology)

self.pushButton\_edit.clicked.connect(self.editTerminology)

self.listView\_Terminologylist.doubleClicked.connect(self.on\_item\_double\_clicked)

self.listView\_Terminologylist.clicked.connect(self.on\_item\_clicked)

self.groupBox\_Terminologydescripte.hide()

self.current\_terminology\_name = None

self.table\_type = 'terminology'

#设置listView的条目模式，并且设置为双击不可编辑,且只能选中一项

self.entries = self.load\_terminology\_names\_from\_db()

self.model = QtCore.QStringListModel(self.entries)

self.listView\_Terminologylist.setModel(self.model)

self.listView\_Terminologylist.setEditTriggers(QtWidgets.QAbstractItemView.NoEditTriggers)

self.listView\_Terminologylist.setSelectionMode(QtWidgets.QAbstractItemView.SingleSelection)

def load\_terminology\_names\_from\_db(self):

try:

all\_descriptions = table\_description\_dao.list\_table\_descriptions\_by\_type(self.table\_type)

terminology\_names = [desc['tm\_name'] for desc in all\_descriptions if 'tm\_name' in desc]

return terminology\_names

except Exception as e:

print(f"加载术语库名称失败: {e}")

return []

def createTerminology(self):

try:

print("打开新建对话框")

self.Create\_Terminology\_show = CreateTerminologyShowDialog()

result = self.Create\_Terminology\_show.exec\_()

print("对话框返回结果:", result)

if result == QtWidgets.QDialog.Accepted:

data = self.Create\_Terminology\_show.get\_data()

terminology\_name = data["Terminology\_name"]

if not terminology\_name:

QtWidgets.QMessageBox.warning(self, "输入错误", "术语库名称不能为空！")

return

if terminology\_name in self.entries:

QtWidgets.QMessageBox.warning(self, "重复错误",

f"术语库名称 '{terminology\_name}' 已存在，请使用其他名称。")

return

self.current\_terminology\_name = terminology\_name

Terminology\_dao.create\_user\_terminology\_table(self.current\_terminology\_name)

table\_description\_dao.insert\_table\_description(

tm\_name = Terminology\_dao.sanitize\_table\_name(self.current\_terminology\_name) ,

item\_number='0',

description=data['description'],

owner='owner',

table\_type=self.table\_type

)

print("用户输入的数据：", data)

entry\_str = self.current\_terminology\_name

self.entries.append(entry\_str)

self.model.setStringList(self.entries)

QtWidgets.QMessageBox.information(self, "收到数据",

f"术语库名称：{data['Terminology\_name']}\n描述：{data['description']}")

else:

print("用户取消输入")

QtWidgets.QMessageBox.information(self, "取消", "你取消了输入")

except Exception as e:

print(f"创建出错: {e}")

def deleteTerminology(self):

terminology\_name = self.current\_terminology\_name

if not terminology\_name:

QtWidgets.QMessageBox.warning(self, "错误", "请选择一个术语库再删除")

return

reply = QtWidgets.QMessageBox.question(self, "确认删除", f"确定要删除术语库 {terminology\_name} 吗？",

QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No)

if reply == QtWidgets.QMessageBox.Yes:

try:

Terminology\_dao.delete\_terminology\_table(terminology\_name)

table\_description\_dao.delete\_table\_description(terminology\_name, self.table\_type)

self.entries = [e for e in self.entries if e != terminology\_name]

self.model.setStringList(self.entries)

QtWidgets.QMessageBox.information(self, "成功", f"术语库 {terminology\_name} 删除成功")

self.groupBox\_Terminologydescripte.hide()

self.current\_terminology\_name = None

except Exception as e:

QtWidgets.QMessageBox.critical(self, "错误", f"删除失败: {e}")

print("删除数据库错误", e)

def editTerminology(self):

"""启用选中行编辑"""

try:

selected\_indexes = self.listView\_Terminologylist.selectedIndexes()

if selected\_indexes:

index = selected\_indexes[0]

item\_text = self.model.data(index, QtCore.Qt.DisplayRole)

print("当前选中条目文本：", item\_text)

text\_all = table\_description\_dao.get\_table\_description(item\_text, self.table\_type)

table\_id = table\_description\_dao.get\_item\_id(text\_all['tm\_name'],text\_all['table\_type'])

text = {'Terminology\_name': text\_all['tm\_name'],'description': text\_all['description']}

self.Create\_Terminology\_show = CreateTerminologyShowDialog(text)

result = self.Create\_Terminology\_show.exec\_()

print("对话框返回结果:", result)

if result == QtWidgets.QDialog.Accepted:

data = self.Create\_Terminology\_show.get\_edited\_data()

table\_description\_dao.update\_item\_by\_id(table\_id, data['Terminology\_name'], data['description'])

Terminology\_dao.rename\_table(item\_text, data['Terminology\_name'])

self.listView\_Terminologylist.model().setData(index, data['Terminology\_name'], QtCore.Qt.EditRole)

self.groupBox\_Terminologydescripte.hide()

else:

print("没有选中任何条目")

QMessageBox.warning(self, "提示", "请先选择一行进行编辑！")

return

except Exception as e:

print("编辑出错" ,e)

def on\_item\_double\_clicked(self, index: QtCore.QModelIndex):

try:

item\_text = self.model.data(index, QtCore.Qt.DisplayRole)

self.Terminology\_show = Terminology\_show.TerminologyItemsShow(name=item\_text)

result = self.Terminology\_show.exec()

self.groupBox\_Terminologydescripte.hide()

except Exception as e:

print(f"双击术语库出错: {e}")

def on\_item\_clicked(self, index: QtCore.QModelIndex):

try:

item\_text = self.model.data(index, QtCore.Qt.DisplayRole)

self.current\_terminology\_name = str(item\_text)

self.groupBox\_Terminologydescripte.show()

self.current\_terminology\_name = Terminology\_dao.sanitize\_table\_name(self.current\_terminology\_name)

data = self.get\_data(self.current\_terminology\_name)

# 设置界面控件显示数据，类似 Widget\_TM.on\_item\_clicked

self.set\_lineedit\_read(True)

# 使用 findChild 方法获取控件

lineEdit\_Terminologyname = self.groupBox\_Terminologydescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_Terminologyname")

lineEdit\_Terminologynumber = self.groupBox\_Terminologydescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_Terminologynumber")

lineEdit\_owner = self.groupBox\_Terminologydescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_owner")

lineEdit\_Terminologycreatetime = self.groupBox\_Terminologydescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_Terminologycreatetime")

textEdit\_description = self.groupBox\_Terminologydescripte.findChild(QtWidgets.QTextEdit, "textEdit\_description")

# 确保控件存在后再设置文本

if lineEdit\_Terminologyname:

lineEdit\_Terminologyname.setText(data.get('tm\_name', ''))

if textEdit\_description:

textEdit\_description.setText(data.get('description', ''))

if lineEdit\_Terminologynumber:

lineEdit\_Terminologynumber.setText(str(data.get('item\_number', '')))

if lineEdit\_owner:

lineEdit\_owner.setText(str(data.get('owner', '')))

if lineEdit\_Terminologycreatetime:

lineEdit\_Terminologycreatetime.setText(str(data.get('create\_time', '')))

except Exception as e:

print(f"单击术语库条目时发生错误: {e}")

def set\_lineedit\_read(self, read: bool):

lineedits = [

self.groupBox\_Terminologydescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_Terminologyname"),

self.groupBox\_Terminologydescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_Terminologynumber"),

self.groupBox\_Terminologydescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_owner"),

self.groupBox\_Terminologydescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_Terminologycreatetime"),

]

for le in lineedits:

le.setReadOnly(read)

textEdit\_description = self.groupBox\_Terminologydescripte.findChild(QtWidgets.QTextEdit, "textEdit\_description")

if textEdit\_description:

textEdit\_description.setReadOnly(read)

def get\_data(self, item\_text):

data = table\_description\_dao.get\_table\_description(item\_text, self.table\_type)

print(data)

if data is None:

data = {

'tm\_name': item\_text,

'item\_number': '0',

'description': '',

'owner': None,

'create\_time': None

}

return data

3.4.3.2术语管理

代码实现：

1. 核心类结构

该文件包含三个主要类，实现术语库条目的展示、编辑和管理功能：

- EditableTextEdit ：自定义文本编辑控件（失去焦点后自动只读）

- NewTerminologyEntryDialog ：新建术语条目对话框

- TerminologyItemsShow ：术语条目管理主窗口

2. EditableTextEdit 控件

功能 ：提供只读/编辑状态自动切换的文本编辑框

3. NewTerminologyEntryDialog 对话框

功能 ：提供术语条目五要素的输入界面

- 术语原文（必填）

- 对应译文（必填）

- 术语定义

- 术语领域

- 所属项目ID

4. TerminologyItemsShow 主窗口

核心功能 ：术语条目管理的完整生命周期处理

4.1 初始化与UI设置

4.2 数据加载与显示

4.3 核心业务功能

创建条目流程 ：

1. 弹出 NewTerminologyEntryDialog 对话框

2. 验证必填字段（术语原文和译文）

3. 调用DAO层添加记录

编辑/保存机制 ：

- 编辑模式：设置选中行（除ID列外）可编辑

- 保存逻辑：验证数据 → 调用DAO更新 → 恢复只读状态

- 状态管理：通过 row\_edit\_status 字典跟踪编辑状态

导入/导出功能 ：def exportTerminology(self):

def importTerminology(self):

4.4 生命周期管理

- 关闭事件 ：自动更新术语库条目数量到元数据表

5. 数据交互层

通过 Terminology\_dao 实现与数据库的交互，核心操作包括：

- list\_terminologies() ：查询术语条目列表

- add\_terminology() ：新增术语

- update\_terminology() ：更新术语

- delete\_terminology() ：删除术语

- export\_terminology() / import\_terminology() ：CSV导入导出

import sys

import os

from PyQt5 import QtCore, QtWidgets, uic

from PyQt5.QtWidgets import QAbstractItemView, QTableWidgetItem, QMessageBox

from src.dao import Terminology\_dao,table\_description\_dao

class EditableTextEdit(QtWidgets.QTextEdit):

"""可编辑的 QTextEdit，失去焦点后自动切回只读状态"""

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.setReadOnly(True)

def focusOutEvent(self, event):

self.setReadOnly(True)

super().focusOutEvent(event)

class NewTerminologyEntryDialog(QtWidgets.QDialog):

"""新建术语条目对话框"""

def \_\_init\_\_(self, parent=None):

super().\_\_init\_\_(parent)

self.setWindowTitle("新建术语条目")

self.resize(400, 400)

self.term\_edit = QtWidgets.QLineEdit()

self.translation\_edit = QtWidgets.QLineEdit()

self.definition\_edit = QtWidgets.QTextEdit()

self.domain\_edit = QtWidgets.QLineEdit()

self.project\_id\_edit = QtWidgets.QLineEdit()

form\_layout = QtWidgets.QFormLayout()

form\_layout.addRow("术语原文:", self.term\_edit)

form\_layout.addRow("对应译文:", self.translation\_edit)

form\_layout.addRow("术语定义:", self.definition\_edit)

form\_layout.addRow("术语领域:", self.domain\_edit)

form\_layout.addRow("所属项目ID:", self.project\_id\_edit)

self.save\_button = QtWidgets.QPushButton("保存")

self.save\_button.clicked.connect(self.accept)

self.cancel\_button = QtWidgets.QPushButton("取消")

self.cancel\_button.clicked.connect(self.reject)

button\_layout = QtWidgets.QHBoxLayout()

button\_layout.addWidget(self.save\_button)

button\_layout.addWidget(self.cancel\_button)

main\_layout = QtWidgets.QVBoxLayout()

main\_layout.addLayout(form\_layout)

main\_layout.addLayout(button\_layout)

self.setLayout(main\_layout)

def get\_data(self):

project\_id\_text = self.project\_id\_edit.text().strip()

project\_id = int(project\_id\_text) if project\_id\_text.isdigit() else None

return {

"term": self.term\_edit.text().strip(),

"translation": self.translation\_edit.text().strip(),

"definition": self.definition\_edit.toPlainText().strip(),

"domain": self.domain\_edit.text().strip(),

"project\_id": project\_id

}

class TerminologyItemsShow(QtWidgets.QDialog):

"""

术语条目展示与管理窗口

列索引说明：

0: term\_id

1: term (术语原文)

2: translation (对应译文)

3: definition (术语定义)

4: domain (术语领域)

5: project\_id (所属项目)

"""

col\_term\_id = 0

col\_term = 1

col\_translation = 2

col\_definition = 3

col\_domain = 4

col\_project\_id = 5

def \_\_init\_\_(self, name="术语库详情"):

super().\_\_init\_\_()

# 获取当前文件所在目录

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

ui\_path = os.path.join(current\_dir, "Terminology\_items\_show.ui")

print("UI 文件路径:", ui\_path)

print("UI 文件存在吗？", os.path.exists(ui\_path))

print("\_\_file\_\_ 的值:", \_\_file\_\_)

print([attr for attr in dir(self) if not attr.startswith('\_')])

uic.loadUi(ui\_path, self)

self.setWindowTitle(name)

self.resize(700, 500)

self.name = name

self.table\_name = Terminology\_dao.sanitize\_table\_name(self.name)

self.table\_type = 'terminology'

# 表格设置：6列，列宽自动拉伸，单行选择

self.tableWidget\_item.setColumnCount(6)

self.tableWidget\_item.setHorizontalHeaderLabels(

["ID", "术语原文", "对应译文", "术语定义", "术语领域", "所属项目ID"]

)

header = self.tableWidget\_item.horizontalHeader()

header.setSectionResizeMode(QtWidgets.QHeaderView.Stretch)

self.tableWidget\_item.setSelectionMode(QAbstractItemView.SingleSelection)

self.tableWidget\_item.setSelectionBehavior(QAbstractItemView.SelectRows)

# 连接信号

self.tableWidget\_item.itemSelectionChanged.connect(self.on\_selection\_changed)

self.pushButton\_createTerminologyitems.clicked.connect(self.createTerminologyitems)

self.pushButton\_save.clicked.connect(self.saveTerminologyitems)

self.pushButton\_edit.clicked.connect(self.editTerminologyitems)

self.pushButton\_delete.clicked.connect(self.deleteTerminologyitems)

self.pushButton\_export.clicked.connect(self.exportTerminology)

self.pushButton\_import.clicked.connect(self.importTerminology)

# 记录行编辑状态，key: row，value: bool（True表示编辑中未保存）

self.row\_edit\_status = {}

# 加载数据

self.load\_terminology\_data()

def closeEvent(self, event):

"""关闭事件，更新描述信息"""

try:

count = table\_description\_dao.get\_record\_count(self.table\_name)

table\_description\_dao.update\_table\_description(self.table\_name, count , None, None,self.table\_type)

print("更新成功")

except Exception as e:

print(f"关闭时更新描述失败: {e}")

super().closeEvent(event)

def load\_terminology\_data(self):

"""从数据库加载术语数据并显示"""

self.tableWidget\_item.clearContents()

self.tableWidget\_item.setRowCount(0)

terminology\_list = Terminology\_dao.list\_terminologies(self.table\_name)

for row\_idx, item in enumerate(terminology\_list):

self.tableWidget\_item.insertRow(row\_idx)

self.tableWidget\_item.setItem(row\_idx, self.col\_term\_id, QTableWidgetItem(str(item.get("term\_id", ""))))

self.tableWidget\_item.setItem(row\_idx, self.col\_term, QTableWidgetItem(item.get("term", "")))

self.tableWidget\_item.setItem(row\_idx, self.col\_translation, QTableWidgetItem(item.get("translation", "")))

self.tableWidget\_item.setItem(row\_idx, self.col\_definition, QTableWidgetItem(item.get("definition") or ""))

self.tableWidget\_item.setItem(row\_idx, self.col\_domain, QTableWidgetItem(item.get("domain") or ""))

project\_id = item.get("project\_id")

self.tableWidget\_item.setItem(row\_idx, self.col\_project\_id, QTableWidgetItem(str(project\_id) if project\_id is not None else ""))

# 默认所有单元格只读

for col in range(self.tableWidget\_item.columnCount()):

item\_widget = self.tableWidget\_item.item(row\_idx, col)

if item\_widget:

item\_widget.setFlags(item\_widget.flags() & ~QtCore.Qt.ItemIsEditable)

def on\_selection\_changed(self):

"""选中行变化时触发"""

selected\_rows = self.tableWidget\_item.selectionModel().selectedRows()

if not selected\_rows:

self.pushButton\_edit.setEnabled(False)

self.pushButton\_delete.setEnabled(False)

self.pushButton\_save.setEnabled(False)

else:

self.pushButton\_edit.setEnabled(True)

self.pushButton\_delete.setEnabled(True)

# 只有编辑状态时才启用保存按钮

row = selected\_rows[0].row()

self.pushButton\_save.setEnabled(self.row\_edit\_status.get(row, False))

def createTerminologyitems(self):

"""新建术语条目"""

dlg = NewTerminologyEntryDialog(self)

if dlg.exec\_() == QtWidgets.QDialog.Accepted:

data = dlg.get\_data()

if not data["term"] or not data["translation"]:

QMessageBox.warning(self, "提示", "术语原文和对应译文不能为空！")

return

term\_id = Terminology\_dao.add\_terminology(

data["term"], data["translation"],self.table\_name, data["definition"], data["domain"], data["project\_id"]

)

if term\_id:

QMessageBox.information(self, "成功", f"新增术语条目成功，ID={term\_id}")

self.load\_terminology\_data()

else:

QMessageBox.warning(self, "失败", "新增术语条目失败！")

def editTerminologyitems(self):

"""启用选中行编辑"""

selected\_rows = self.tableWidget\_item.selectionModel().selectedRows()

if not selected\_rows:

QMessageBox.warning(self, "提示", "请先选择一行进行编辑！")

return

row = selected\_rows[0].row()

# 设置该行所有单元格可编辑，除了ID列

for col in range(1, self.tableWidget\_item.columnCount()):

item = self.tableWidget\_item.item(row, col)

if item:

item.setFlags(item.flags() | QtCore.Qt.ItemIsEditable)

self.row\_edit\_status[row] = True

self.pushButton\_save.setEnabled(True)

def saveTerminologyitems(self):

"""保存编辑后的术语条目"""

selected\_rows = self.tableWidget\_item.selectionModel().selectedRows()

if not selected\_rows:

QMessageBox.warning(self, "提示", "请先选择一行进行保存！")

return

row = selected\_rows[0].row()

if not self.row\_edit\_status.get(row, False):

QMessageBox.information(self, "提示", "该行没有处于编辑状态，无需保存。")

return

try:

term\_id = int(self.tableWidget\_item.item(row, self.col\_term\_id).text())

term = self.tableWidget\_item.item(row, self.col\_term).text().strip()

translation = self.tableWidget\_item.item(row, self.col\_translation).text().strip()

definition = self.tableWidget\_item.item(row, self.col\_definition).text().strip()

domain = self.tableWidget\_item.item(row, self.col\_domain).text().strip()

project\_id\_text = self.tableWidget\_item.item(row, self.col\_project\_id).text().strip()

project\_id = int(project\_id\_text) if project\_id\_text.isdigit() else None

if not term or not translation:

QMessageBox.warning(self, "提示", "术语原文和对应译文不能为空！")

return

success = Terminology\_dao.update\_terminology(term\_id,self.table\_name, term, translation, definition, domain, project\_id)

if success:

QMessageBox.information(self, "成功", "术语条目保存成功！")

# 取消编辑状态，设置单元格只读

for col in range(self.tableWidget\_item.columnCount()):

item = self.tableWidget\_item.item(row, col)

if item:

item.setFlags(item.flags() & ~QtCore.Qt.ItemIsEditable)

self.row\_edit\_status[row] = False

self.pushButton\_save.setEnabled(False)

else:

QMessageBox.warning(self, "失败", "术语条目保存失败！")

except Exception as e:

QMessageBox.warning(self, "错误", f"保存时发生异常: {e}")

def deleteTerminologyitems(self):

"""删除选中术语条目"""

selected\_rows = self.tableWidget\_item.selectionModel().selectedRows()

if not selected\_rows:

QMessageBox.warning(self, "提示", "请先选择一行进行删除！")

return

row = selected\_rows[0].row()

term\_id = self.tableWidget\_item.item(row, self.col\_term\_id).text()

ret = QMessageBox.question(self, "确认删除", f"确定删除ID={term\_id}的术语条目吗？", QMessageBox.Yes | QMessageBox.No)

if ret == QMessageBox.Yes:

success = Terminology\_dao.delete\_terminology(int(term\_id),self.table\_name)

if success:

QMessageBox.information(self, "成功", "术语条目删除成功！")

self.load\_terminology\_data()

else:

QMessageBox.warning(self, "失败", "术语条目删除失败！")

def exportTerminology(self):

"""导出术语库到CSV文件"""

from PyQt5.QtWidgets import QFileDialog

options = QFileDialog.Options()

file\_path, \_ = QFileDialog.getSaveFileName(self, "导出术语库", "", "CSV Files (\*.csv);;All Files (\*)",

options=options)

if file\_path:

try:

Terminology\_dao.export\_terminology(file\_path ,self.table\_name)

QMessageBox.information(self, "导出成功", f"术语库已成功导出到:\n{file\_path}")

except Exception as e:

QMessageBox.warning(self, "导出失败", f"导出时发生错误:\n{e}")

def importTerminology(self):

"""从CSV文件导入术语库"""

from PyQt5.QtWidgets import QFileDialog

options = QFileDialog.Options()

file\_path, \_ = QFileDialog.getOpenFileName(self, "导入术语库", "", "CSV Files (\*.csv);;All Files (\*)",

options=options)

if file\_path:

try:

Terminology\_dao.import\_terminology(file\_path ,self.table\_name)

QMessageBox.information(self, "导入成功", f"术语库已成功从文件导入:\n{file\_path}")

self.load\_terminology\_data() # 重新加载数据

except Exception as e:

QMessageBox.warning(self, "导入失败", f"导入时发生错误:\n{e}")

if \_\_name\_\_ == "\_\_main\_\_":

app = QtWidgets.QApplication(sys.argv)

window = TerminologyItemsShow()

window.show()

sys.exit(app.exec\_())

3.4.4**记忆库管理**

3.4.4.1**库管理**

代码实现

1. 核心类结构

该文件包含两个主要类，实现翻译记忆库(TM)的创建、管理和展示功能：

CreateTMShowDialog：翻译记忆库创建/编辑对话框

Widget\_TM：翻译记忆库管理主界面组件

2. CreateTMShowDialog 对话框

功能：提供翻译记忆库名称和描述的输入界面

核心方法：

TM\_manage.py

3. Widget\_TM 主界面

初始化流程：

加载UI文件 TM\_manage\_show.ui

设置按钮事件绑定（创建/删除/编辑/关闭窗口）

初始化列表视图，加载翻译记忆库名称

路径处理：

TM\_manage.py

4. 核心功能实现

4.1 翻译记忆库列表管理

TM\_manage.py

4.2 翻译记忆库创建流程

点击创建按钮弹出 CreateTMShowDialog 对话框

验证输入（名称非空、不重复）

调用DAO层创建数据库表和描述记录

TM\_manage.py

4.3 翻译记忆库编辑/删除

编辑：获取选中项数据 → 弹出编辑对话框 → 更新数据库记录

删除：验证选中状态 → 确认对话框 → 调用DAO删除表和描述

4.4 记忆库详情展示

单击列表项：在右侧面板显示记忆库元数据（名称、数量、创建时间等）

双击列表项：打开 TM\_items\_show 窗口查看记忆库条目详情

5. 数据交互层

通过以下DAO组件与数据库交互：

TM\_dao：处理翻译记忆库表的创建、删除、重命名

table\_description\_dao：管理记忆库元数据（描述、条目数量等）

import sys

import os

from PyQt5 import QtCore, QtGui, QtWidgets ,uic

from PyQt5.QtWidgets import QMessageBox

# 获取 src 目录的绝对路径

src\_path = os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), ".."))

if src\_path not in sys.path:

sys.path.insert(0, src\_path)

# 现在可以导入 dao 下的模块

from dao import TM\_dao , table\_description\_dao

from temp\_tm\_manage翻译记忆库 import temp\_TM\_item\_show

class CreateTMShowDialog(QtWidgets.QDialog):

def \_\_init\_\_(self , text =None):

super().\_\_init\_\_()

self.original\_data = None

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

ui\_path = os.path.join(current\_dir, "Create\_TM\_show.ui")

uic.loadUi(ui\_path, self)

self.Create\_TM\_sure.clicked.connect(self.accept)

self.Create\_TM\_Cancle.clicked.connect(self.reject)

self.push\_data(text)

def get\_data(self):

#获取用户输入的数据，以字典的形式返回

return {

"TM\_name": self.lineEdit\_TM\_name.text().strip(),

"description": self.textEdit\_TM\_descripte.toPlainText().strip()

}

def push\_data(self, data):

if data is None:

print("push\_data : data is None")

return

self.original\_data = data.copy() # 备份原始数据

self.lineEdit\_TM\_name.setText(data["TM\_name"])

self.textEdit\_TM\_descripte.setText(data["description"])

def get\_edited\_data(self):

current\_name = self.lineEdit\_TM\_name.text()

current\_description = self.textEdit\_TM\_descripte.toPlainText()

if self.original\_data is None:

return None # 没有原始数据，无法判断

# 比较是否有修改

if (current\_name != self.original\_data["TM\_name"] or

current\_description != self.original\_data["description"]):

# 返回修改后的结果

return {

"TM\_name": current\_name,

"description": current\_description

}

else:

# 没有修改，返回空

return None

class Widget\_TM(QtWidgets.QWidget):

def \_\_init\_\_(self):

super(Widget\_TM, self).\_\_init\_\_()

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

ui\_path = os.path.join(current\_dir, "TM\_manage\_show.ui")

uic.loadUi(ui\_path, self)

self.resize(800, 600)

self.setWindowTitle("翻译记忆库管理")

self.pushButton.clicked.connect(self.close\_window)

self.pushButton\_createTM.clicked.connect(self.createTM)

self.pushButton\_edit.clicked.connect(self.editTM)

self.pushButton\_deleteTM.clicked.connect(self.deleteTM)

self.groupBox\_TMdescripte.hide()

self.current\_TM\_name = None

self.table\_type = 'tm'

# 从数据库加载记忆库名称列表

self.entries = self.load\_tm\_names\_from\_db()

self.model = QtCore.QStringListModel(self.entries)

self.listView\_TMlist.setModel(self.model)

self.listView\_TMlist.doubleClicked.connect(self.on\_item\_double\_clicked)

self.listView\_TMlist.clicked.connect(self.on\_item\_clicked)

self.listView\_TMlist.setEditTriggers(QtWidgets.QAbstractItemView.NoEditTriggers)

self.listView\_TMlist.setSelectionMode(QtWidgets.QAbstractItemView.SingleSelection)

def close\_window(self):

self.close()

def set\_lineedit\_read(self, read: bool):

lineedits = [

self.groupBox\_TMdescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_TMname"),

self.groupBox\_TMdescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_TMnumber"),

self.groupBox\_TMdescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_owner"),

self.groupBox\_TMdescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_TMcreatetime"),

]

for le in lineedits:

le.setReadOnly(read)

textEdit\_description = self.groupBox\_TMdescripte.findChild(QtWidgets.QTextEdit, "textEdit\_description")

if textEdit\_description:

textEdit\_description.setReadOnly(read)

def load\_tm\_names\_from\_db(self):

"""

从数据库的 table\_description 表中获取所有记忆库名称列表

:return: list[str]

"""

try:

# 假设 table\_description\_dao 有方法 list\_table\_descriptions\_by\_type 返回所有记录列表

all\_descriptions = table\_description\_dao.list\_table\_descriptions\_by\_type(self.table\_type)

# all\_descriptions 应该是列表，每个元素是 dict，包含 'tm\_name' 字段

tm\_names = [desc['tm\_name'] for desc in all\_descriptions if 'tm\_name' in desc]

return tm\_names

except Exception as e:

print(f"加载记忆库名称失败: {e}")

return []

def createTM(self):

try :

print("打开新建对话框")

self.Create\_TM\_show = CreateTMShowDialog()

result = self.Create\_TM\_show.exec\_()

print("对话框返回结果:", result)

if result == QtWidgets.QDialog.Accepted:

data = self.Create\_TM\_show.get\_data()

tm\_name = data["TM\_name"]

# 检查名称是否为空

if not tm\_name:

QtWidgets.QMessageBox.warning(self, "输入错误", "记忆库名称不能为空！")

return

# 检查名称是否重复

if tm\_name in self.entries:

QtWidgets.QMessageBox.warning(self, "重复错误", f"记忆库名称 '{tm\_name}' 已存在，请使用其他名称。")

return

#更新数据库

self.current\_TM\_name = TM\_dao.sanitize\_identifier(data['TM\_name'])

TM\_dao.create\_user\_tm\_table(self.current\_TM\_name )

table\_description\_dao.insert\_table\_description(

tm\_name=self.current\_TM\_name,

item\_number='0',

description=data['description'],

owner='owner', # 你需要传入owner，或者默认值

table\_type=self.table\_type

)

print("用户输入的数据：", data)

# 构造显示字符串，比如显示记忆库名称和语言对

entry\_str = self.current\_TM\_name

# 添加到条目列表

self.entries.append(entry\_str)

# 更新模型，刷新listView显示

self.model.setStringList(self.entries)

QtWidgets.QMessageBox.information(self, "收到数据", f"记忆库名称：{data['TM\_name']}\n"

f"描述：{data['description']}")

else:

print("用户取消输入")

QtWidgets.QMessageBox.information(self, "取消", "你取消了输入")

except Exception as e:

print(f"创建出错: {e}")

#编辑记忆库

def editTM(self):

"""启用选中行编辑"""

try:

selected\_indexes = self.listView\_TMlist.selectedIndexes()

if selected\_indexes:

index = selected\_indexes[0]

item\_text = self.model.data(index, QtCore.Qt.DisplayRole)

self.current\_TM\_name = item\_text

print("当前选中条目文本：", item\_text)

text\_all = table\_description\_dao.get\_table\_description(item\_text, self.table\_type)

table\_id = table\_description\_dao.get\_item\_id(text\_all['tm\_name'], text\_all['table\_type'])

text = {'TM\_name': text\_all['tm\_name'], 'description': text\_all['description']}

self.Create\_TM\_show = CreateTMShowDialog(text)

result = self.Create\_TM\_show.exec\_()

print("对话框返回结果:", result)

if result == QtWidgets.QDialog.Accepted:

data = self.Create\_TM\_show.get\_edited\_data()

table\_description\_dao.update\_item\_by\_id(table\_id, data['TM\_name'], data['description'])

TM\_dao.rename\_table(item\_text, data['TM\_name'])

self.listView\_TMlist.model().setData(index, data['TM\_name'], QtCore.Qt.EditRole)

self.groupBox\_TMdescripte.hide()

else:

print("没有选中任何条目")

QMessageBox.warning(self, "提示", "请先选择一行进行编辑！")

return

except Exception as e:

print("编辑出错", e)

#删除翻译记忆库

def deleteTM(self):

tm\_name = self.current\_TM\_name

if not tm\_name:

QtWidgets.QMessageBox.warning(self, "错误", "请选择一个记忆库再删除")

return

reply = QtWidgets.QMessageBox.question(self, "确认删除", f"确定要删除记忆库 {tm\_name} 吗？",

QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No)

if reply == QtWidgets.QMessageBox.Yes:

try:

TM\_dao.delete\_tm\_table(TM\_dao.sanitize\_identifier(tm\_name))

table\_description\_dao.delete\_table\_description(tm\_name, self.table\_type)

# 从列表中移除对应条目（entries中存的是带语言的字符串，需要匹配）

self.entries = [e for e in self.entries if tm\_name not in e]

self.model.setStringList(self.entries)

QtWidgets.QMessageBox.information(self, "成功", f"记忆库 {tm\_name} 删除成功")

self.groupBox\_TMdescripte.hide()

self.current\_TM\_name = None

except Exception as e:

QtWidgets.QMessageBox.critical(self, "错误", f"删除失败: {e}")

print("删除数据库错误",e)

def on\_item\_double\_clicked(self, index:QtCore.QModelIndex):

try:

item\_text = self.model.data(index, QtCore.Qt.DisplayRole)

self.TM\_item\_show = temp\_TM\_item\_show.TM\_items\_show(name=item\_text)

self.TM\_item\_show.exec()

except Exception as e:

print(f"双击出错: {e}")

def on\_item\_clicked(self, index: QtCore.QModelIndex):

try:

item\_text = self.model.data(index, QtCore.Qt.DisplayRole)

self.current\_TM\_name =TM\_dao.sanitize\_identifier(item\_text ) # 更新当前选中记忆库名

self.groupBox\_TMdescripte.show()

data = self.get\_data(self.current\_TM\_name)

self.set\_lineedit\_read(True)

# 使用 findChild 方法获取控件

lineEdit\_TMname = self.groupBox\_TMdescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_TMname")

lineEdit\_TMnumber = self.groupBox\_TMdescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_TMnumber")

lineEdit\_owner = self.groupBox\_TMdescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_owner")

lineEdit\_TMcreatetime = self.groupBox\_TMdescripte.findChild(QtWidgets.QLineEdit, "lineEdit\_TMcreatetime")

textEdit\_description = self.groupBox\_TMdescripte.findChild(QtWidgets.QTextEdit, "textEdit\_description")

# 确保控件存在后再设置文本

if lineEdit\_TMname:

lineEdit\_TMname.setText(data.get('tm\_name', ''))

if textEdit\_description:

textEdit\_description.setText(data.get('description', ''))

if lineEdit\_TMnumber:

lineEdit\_TMnumber.setText(str(data.get('item\_number', '')))

if lineEdit\_owner:

lineEdit\_owner.setText(str(data.get('owner', '')))

if lineEdit\_TMcreatetime:

lineEdit\_TMcreatetime.setText(str(data.get('create\_time', '')))

except Exception as e:

print(f"单击条目时发生错误: {e}")

def get\_data(self , item\_text):

"""

从数据库得到数据

:param item\_text: tm\_name，记忆库名称

:return: dict，包含库信息，如果没找到返回默认值

"""

data = table\_description\_dao.get\_table\_description(item\_text,self.table\_type)

if data is None:

# 如果数据库没有对应记录，返回默认值

data = {

'tm\_name': item\_text,

'item\_number': '0',

'description': '',

'owner': None,

'create\_time': None

}

return data

3.4.4.2**记忆对管理**

1. temp\_TM\_item\_show.py文件概述

该文件是翻译记忆库（TM）条目的管理界面实现，基于PyQt5构建，主要负责翻译记忆库条目的可视化管理，包括创建、编辑、保存、删除、导入和导出等功能

2核心类结构

2.1 EditableTextEdit 类

- 功能 ：自定义文本编辑控件，实现焦点控制的只读/可编辑切换

- 核心实现 ：

class EditableTextEdit(QtWidgets.QTextEdit):

def init (self, \*args, \*\*kwargs):

super(). init (\*args, \*\*kwargs)

self.setReadOnly(True) # 初始只读

- \*\*特点\*\*：仅在获得焦点时可编辑，失去焦点自动恢复只读状态，防止误操作

2.2 `NewTMEntryDialog` 类

- \*\*功能\*\*：新建翻译记忆条目对话框

- \*\*核心方法\*\*：

- `\_\_init\_\_`：初始化对话框布局，包含源文本/目标文本编辑框和语言显示标签

- `detect\_source\_language`/`detect\_target\_language`：通过`Tool.LanguageMapper`自动检测文本语言

- `get\_data`：返回用户输入的条目数据（源文本、目标文本、语言、创建者ID等）

- \*\*界面交互\*\*：文本变化时自动触发语言检测，保存按钮触发数据验证和提交

2.3 `TM\_items\_show` 类（核心类）

- \*\*功能\*\*：翻译记忆库条目管理主界面

- \*\*核心属性\*\*：

- `TM\_name`：当前操作的记忆库名称

- `row\_edit\_status`：记录行编辑状态的字典

- 列索引常量（COL\_ID、COL\_SOURCE\_TEXT等）

3. 数据交互层

- \*\*数据库操作\*\*：通过`TM\_dao`模块实现与数据库的交互，主要方法包括：

- `insert\_tm\_entry`：插入新条目

- `update\_tm\_entry`：更新条目

- `delete\_tm\_entry`：删除条目

- `batch\_export`：批量导出数据

- `batch\_insert`：批量插入数据

- \*\*用户信息\*\*：当前用户ID硬编码为1（`current\_user\_id = 1`），需后续优化为动态获取

4. 关键技术特点

1. \*\*编辑状态管理\*\*：通过`row\_edit\_status`跟踪编辑状态，切换选中行时自动保存

2. \*\*语言自动检测\*\*：集成`Tool.LanguageMapper`实现文本语言自动识别

3. \*\*UI与数据分离\*\*：通过DAO层隔离数据库操作，提高代码可维护性

4. \*\*错误处理\*\*：关键操作包含异常捕获和用户提示

6. 依赖关系

- \*\*UI文件\*\*：`TM\_items\_show.ui`（界面布局定义）

- \*\*工具模块\*\*：`Tool.py`（提供语言映射功能）

- \*\*数据访问\*\*：`TM\_dao.py`和`table\_description\_dao.py`（数据库操作）

- \*\*第三方库\*\*：PyQt5（界面）、sqlite3（数据库）、csv（文件处理）

代码分析

import sys

import sqlite3

import os

import csv

from PyQt5.QtWidgets import QFileDialog, QMessageBox

from PyQt5 import QtCore, QtGui, QtWidgets ,uic

from PyQt5.QtWidgets import QAbstractItemView, QTableWidgetItem

from . import Tool

# 获取 src 目录的绝对路径

src\_path = os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), ".."))

if src\_path not in sys.path:

sys.path.insert(0, src\_path)

# 现在可以导入 dao 下的模块

from dao import TM\_dao

from dao import table\_description\_dao

class EditableTextEdit(QtWidgets.QTextEdit):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.setReadOnly(True) # 初始只读

def focusOutEvent(self, event):

# 失去焦点时切回只读

self.setReadOnly(True)

super().focusOutEvent(event)

self.setMouseTracking(True) # 开启鼠标追踪

class NewTMEntryDialog(QtWidgets.QDialog):

def \_\_init\_\_(self, current\_user\_id, parent=None):

super().\_\_init\_\_(parent)

self.current\_user\_id = current\_user\_id

self.setWindowTitle("新建翻译记忆条目")

self.resize(400, 300)

self.source\_text\_edit = QtWidgets.QTextEdit()

self.target\_text\_edit = QtWidgets.QTextEdit()

# 改为 QLabel 显示语言，禁止编辑

self.source\_lang\_label = QtWidgets.QLabel()

self.target\_lang\_label = QtWidgets.QLabel()

form\_layout = QtWidgets.QFormLayout()

form\_layout.addRow("源文本:", self.source\_text\_edit)

form\_layout.addRow("目标文本:", self.target\_text\_edit)

form\_layout.addRow("源语言:", self.source\_lang\_label)

form\_layout.addRow("目标语言:", self.target\_lang\_label)

self.save\_button = QtWidgets.QPushButton("保存")

self.save\_button.clicked.connect(self.accept)

self.cancel\_button = QtWidgets.QPushButton("取消")

self.cancel\_button.clicked.connect(self.reject)

button\_layout = QtWidgets.QHBoxLayout()

button\_layout.addWidget(self.save\_button)

button\_layout.addWidget(self.cancel\_button)

main\_layout = QtWidgets.QVBoxLayout()

main\_layout.addLayout(form\_layout)

main\_layout.addLayout(button\_layout)

self.setLayout(main\_layout)

self.source\_text\_edit.textChanged.connect(self.detect\_source\_language)

self.target\_text\_edit.textChanged.connect(self.detect\_target\_language)

# 打开对话框时主动检测

self.detect\_source\_language()

self.detect\_target\_language()

def detect\_source\_language(self):

text = self.source\_text\_edit.toPlainText().strip()

if text:

try:

mapper = Tool.LanguageMapper()

lang = mapper.detect\_language(text)

self.source\_lang\_label.setText(lang)

except Exception as e:

print(e)

self.source\_lang\_label.setText("")

else:

print("3")

self.source\_lang\_label.setText("")

def detect\_target\_language(self):

text = self.target\_text\_edit.toPlainText().strip()

if text:

try:

mapper = Tool.LanguageMapper()

lang = mapper.detect\_language(text)

self.target\_lang\_label.setText(lang)

except Exception as e:

print(e)

self.target\_lang\_label.setText("")

else:

self.target\_lang\_label.setText("")

def get\_data(self):

return {

"source\_text": self.source\_text\_edit.toPlainText().strip(),

"target\_text": self.target\_text\_edit.toPlainText().strip(),

"source\_lang": self.source\_lang\_label.text().strip(),

"target\_lang": self.target\_lang\_label.text().strip(),

"created\_by": self.current\_user\_id

}

'''

TM\_items\_show中的提示可以考虑提示其他关键词，行号索引会动态变化，不直观

editTMitems 的QtWidgets.QMessageBox.warning(self, "提示", "可以编辑" + str(row + 1))

saveTMitems(self): QtWidgets.QMessageBox.warning(self, "提示", "已经保存" + str(row + 1))

deleteTMitems(self): QtWidgets.QMessageBox.warning(self, "提示", "已删除行" + str(row + 1))

on\_selection\_changed(self): QtWidgets.QMessageBox.warning(self, "提示", "已经保存" + str(row + 1))

'''

class TM\_items\_show(QtWidgets.QDialog):

# 列索引常量

COL\_ID = 0

COL\_SOURCE\_TEXT = 1

COL\_TARGET\_TEXT = 2

COL\_SOURCE\_LANG = 3

COL\_TARGET\_LANG = 4

COL\_CREATED\_BY = 5

COL\_CREATED\_AT = 6

def \_\_init\_\_(self, name="记忆库详情"):

super().\_\_init\_\_()

self.name = name

self.TM\_name = TM\_dao.sanitize\_identifier(name)

self.row\_edit\_status = {} # 记录行编辑状态

self.prev\_selected\_items = set()

self.ignore\_selection\_change = False

self.is\_create\_now = False

self.\_load\_ui()

self.\_init\_ui()

self.\_connect\_signals()

self.load\_tm\_data()

def \_load\_ui(self):

"""加载UI文件"""

current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

ui\_path = os.path.join(current\_dir, "TM\_items\_show.ui")

if not os.path.exists(ui\_path):

raise FileNotFoundError(f"UI文件未找到: {ui\_path}")

uic.loadUi(ui\_path, self)

self.setWindowTitle(self.name)

self.resize(600, 500)

self.table\_type = "tm"

def \_init\_ui(self):

"""初始化表格和控件状态"""

header = self.tableWidget\_item.horizontalHeader()

header.setSectionResizeMode(QtWidgets.QHeaderView.Stretch)

self.tableWidget\_item.setSelectionMode(QAbstractItemView.SingleSelection)

self.tableWidget\_item.setSelectionBehavior(QAbstractItemView.SelectRows)

def \_connect\_signals(self):

"""连接信号槽"""

self.tableWidget\_item.itemSelectionChanged.connect(self.on\_selection\_changed)

self.pushButton\_createTMitems.clicked.connect(self.create\_tm\_items)

self.pushButton\_save.clicked.connect(self.save\_tm\_items)

self.pushButton\_edit.clicked.connect(self.edit\_tm\_items)

self.pushButton\_delete.clicked.connect(self.delete\_tm\_items)

self.pushButton\_export.clicked.connect(self.export\_tm\_items)

self.pushButton\_import.clicked.connect(self.import\_tm\_items)

def closeEvent(self, event):

"""关闭事件，更新描述信息"""

try:

count = table\_description\_dao.get\_record\_count(self.TM\_name)

table\_description\_dao.update\_table\_description(self.TM\_name, count , None, None,self.table\_type)

print("更新成功")

except Exception as e:

print(f"关闭时更新描述失败: {e}")

super().closeEvent(event)

def create\_tm\_items(self):

"""创建新条目"""

try:

current\_user\_id = 1 # TODO: 动态获取用户ID

dialog = NewTMEntryDialog(current\_user\_id, self)

if dialog.exec\_() != QtWidgets.QDialog.Accepted:

return

data = dialog.get\_data()

if not data["source\_text"] or not data["target\_text"]:

QMessageBox.warning(self, "提示", "请填写所有字段")

return

tm\_id, created\_at = TM\_dao.insert\_tm\_entry(

self.TM\_name,

data["source\_text"], data["target\_text"],

data["source\_lang"], data["target\_lang"],

data["created\_by"]

)

self.\_insert\_row(tm\_id, data, created\_at)

self.is\_create\_now = True

self.save\_tm\_items()

self.is\_create\_now = False

QMessageBox.information(self, "成功", f"新条目已创建，ID: {tm\_id}")

except Exception as e:

print(f"创建条目失败: {e}")

def \_insert\_row(self, tm\_id, data, created\_at):

"""向表格插入一行数据"""

row = self.tableWidget\_item.rowCount()

self.tableWidget\_item.insertRow(row)

# ID列

self.tableWidget\_item.setItem(row, self.COL\_ID, QTableWidgetItem(str(tm\_id)))

# 源文本和目标文本使用EditableTextEdit控件

source\_edit = EditableTextEdit()

source\_edit.setPlainText(data["source\_text"])

source\_edit.setReadOnly(True)

self.tableWidget\_item.setCellWidget(row, self.COL\_SOURCE\_TEXT, source\_edit)

target\_edit = EditableTextEdit()

target\_edit.setPlainText(data["target\_text"])

target\_edit.setReadOnly(True)

self.tableWidget\_item.setCellWidget(row, self.COL\_TARGET\_TEXT, target\_edit)

# 其他列

def create\_item(text):

item = QTableWidgetItem(str(text))

item.setToolTip(str(text))

item.setFlags(item.flags() & ~QtCore.Qt.ItemIsEditable)

return item

self.tableWidget\_item.setItem(row, self.COL\_SOURCE\_LANG, create\_item(data["source\_lang"]))

self.tableWidget\_item.setItem(row, self.COL\_TARGET\_LANG, create\_item(data["target\_lang"]))

self.tableWidget\_item.setItem(row, self.COL\_CREATED\_BY, create\_item(data["created\_by"]))

self.tableWidget\_item.setItem(row, self.COL\_CREATED\_AT, create\_item(created\_at))

self.row\_edit\_status[row] = False

self.tableWidget\_item.selectRow(row)

def edit\_tm\_items(self):

"""编辑选中行"""

row = self.\_get\_selected\_row()

if row is None:

QMessageBox.warning(self, "提示", "请先选择一行进行编辑")

return

self.set\_table\_cells\_read(row, False)

self.row\_edit\_status[row] = True

QMessageBox.information(self, "提示", f"可以编辑第 {row + 1} 行")

def save\_tm\_items(self):

"""保存选中行"""

row = self.\_get\_selected\_row()

if row is None:

QMessageBox.warning(self, "提示", "请先选择一行进行保存")

return

self.set\_table\_cells\_read(row, True)

self.push\_item\_data(row, False)

self.row\_edit\_status[row] = False

if not self.is\_create\_now:

QMessageBox.information(self, "提示", f"已保存第 {row + 1} 行")

self.tableWidget\_item.clearSelection()

def delete\_tm\_items(self):

"""删除选中行"""

row = self.\_get\_selected\_row()

if row is None:

QMessageBox.warning(self, "提示", "请先选择一行进行删除")

return

try:

self.ignore\_selection\_change = True # 禁止选中变化信号处理

self.push\_item\_data(row, True)

self.tableWidget\_item.removeRow(row)

# 调整 row\_edit\_status 字典，删除对应行的状态

self.row\_edit\_status.pop(row, None)

# 重新整理 row\_edit\_status 键，避免行号错乱

new\_status = {}

for r, status in self.row\_edit\_status.items():

if r > row:

new\_status[r - 1] = status

elif r < row:

new\_status[r] = status

self.row\_edit\_status = new\_status

QMessageBox.information(self, "提示", f"已删除第 {row + 1} 行")

except Exception as e:

QMessageBox.warning(self, "错误", f"删除失败: {e}")

finally:

self.ignore\_selection\_change = False

def export\_tm\_items(self):

"""导出为CSV"""

file\_path, \_ = QFileDialog.getSaveFileName(self, "导出翻译记忆库", "", "CSV文件 (\*.csv)")

if not file\_path:

return

try:

rows = TM\_dao.batch\_export(self.TM\_name)

with open(file\_path, 'w', encoding='utf-8-sig', newline='') as f:

writer = csv.writer(f)

writer.writerow(['ID', '源文本', '目标文本', '源语言', '目标语言', '创建者', '创建时间'])

writer.writerows(rows)

QMessageBox.information(self, "导出成功", f"已成功导出到 {file\_path}")

except Exception as e:

QMessageBox.warning(self, "导出失败", f"导出过程中出错: {e}")

def import\_tm\_items(self):

"""从CSV导入"""

file\_path, \_ = QFileDialog.getOpenFileName(self, "导入翻译记忆库", "", "CSV文件 (\*.csv)")

if not file\_path:

return

try:

rows\_to\_insert = []

with open(file\_path, encoding='utf-8-sig') as f:

reader = csv.DictReader(f)

for row in reader:

if not row.get('源文本') or not row.get('目标文本') or not row.get('源语言') or not row.get('目标语言'):

continue

created\_by = int(row.get('创建者', '1')) if row.get('创建者', '1').isdigit() else 1

rows\_to\_insert.append((

row['源文本'], row['目标文本'], row['源语言'], row['目标语言'], created\_by

))

if not rows\_to\_insert:

QMessageBox.warning(self, "导入失败", "没有有效数据可导入")

return

success = TM\_dao.batch\_insert(self.TM\_name, rows\_to\_insert)

if success:

self.load\_tm\_data()

QMessageBox.information(self, "导入成功", f"成功导入 {len(rows\_to\_insert)} 条记录")

else:

QMessageBox.warning(self, "导入失败", "批量插入数据库时出错")

except Exception as e:

QMessageBox.warning(self, "导入失败", f"导入过程中出错: {e}")

def on\_selection\_changed(self):

"""选中行变化时自动保存未保存编辑"""

if self.ignore\_selection\_change:

return

try:

current\_selected\_rows = set(item.row() for item in self.tableWidget\_item.selectedItems())

deselected\_rows = self.prev\_selected\_items - current\_selected\_rows

for row in deselected\_rows:

if self.row\_edit\_status.get(row, False):

self.set\_table\_cells\_read(row, True)

self.push\_item\_data(row, False)

self.row\_edit\_status[row] = False

QMessageBox.information(self, "提示", f"第 {row + 1} 行已自动保存")

self.prev\_selected\_items = current\_selected\_rows

except Exception as e:

print(f"选中变化处理异常: {e}")

def set\_table\_cells\_read(self, row, read: bool):

"""设置指定行的单元格是否可编辑"""

source\_text = self.tableWidget\_item.cellWidget(row, self.COL\_SOURCE\_TEXT)

target\_text = self.tableWidget\_item.cellWidget(row, self.COL\_TARGET\_TEXT)

if source\_text:

source\_text.setReadOnly(read)

if target\_text:

target\_text.setReadOnly(read)

for col in range(self.tableWidget\_item.columnCount()):

if col in (self.COL\_CREATED\_AT, self.COL\_ID, self.COL\_CREATED\_BY):

item = self.tableWidget\_item.item(row, col)

if item:

item.setFlags(item.flags() & ~QtCore.Qt.ItemIsEditable)

elif col not in (self.COL\_SOURCE\_TEXT, self.COL\_TARGET\_TEXT):

item = self.tableWidget\_item.item(row, col)

if item:

if read:

item.setFlags(item.flags() & ~QtCore.Qt.ItemIsEditable)

else:

item.setFlags(item.flags() | QtCore.Qt.ItemIsEditable)

self.tableWidget\_item.viewport().update()

def load\_tm\_data(self):

"""加载数据库数据到表格"""

try:

rows = TM\_dao.batch\_export(self.TM\_name)

self.tableWidget\_item.clear()

self.tableWidget\_item.setRowCount(len(rows) if rows else 0)

self.tableWidget\_item.setColumnCount(7)

self.tableWidget\_item.setHorizontalHeaderLabels(

['ID', '源文本', '目标文本', '源语言', '目标语言', '创建者', '创建时间']

)

if not rows:

return

for row\_idx, row\_data in enumerate(rows):

for col\_idx, value in enumerate(row\_data):

if col\_idx in (self.COL\_SOURCE\_TEXT, self.COL\_TARGET\_TEXT):

text\_edit = EditableTextEdit()

text\_edit.setPlainText(str(value) if value else "")

text\_edit.setReadOnly(True)

self.tableWidget\_item.setCellWidget(row\_idx, col\_idx, text\_edit)

else:

item = QTableWidgetItem(str(value) if value else "")

item.setFlags(item.flags() & ~QtCore.Qt.ItemIsEditable)

self.tableWidget\_item.setItem(row\_idx, col\_idx, item)

except Exception as e:

print("加载数据库数据错误", e)

def push\_item\_data(self, row, is\_delete):

"""更新或删除数据库中的数据"""

tm\_id\_item = self.tableWidget\_item.item(row, self.COL\_ID)

if tm\_id\_item is None:

print("该行无有效ID，无法操作")

return

try:

tm\_id = int(tm\_id\_item.text())

except ValueError:

print("tm\_id格式错误，无法操作")

return

if is\_delete:

TM\_dao.delete\_tm\_entry(self.TM\_name, tm\_id)

print(f"已删除 tm\_id={tm\_id} 的记录")

return

source\_text\_widget = self.tableWidget\_item.cellWidget(row, self.COL\_SOURCE\_TEXT)

target\_text\_widget = self.tableWidget\_item.cellWidget(row, self.COL\_TARGET\_TEXT)

source\_text = source\_text\_widget.toPlainText() if source\_text\_widget else ""

target\_text = target\_text\_widget.toPlainText() if target\_text\_widget else ""

source\_lang = self.tableWidget\_item.item(row, self.COL\_SOURCE\_LANG).text()

target\_lang = self.tableWidget\_item.item(row, self.COL\_TARGET\_LANG).text()

TM\_dao.update\_tm\_entry(

self.TM\_name, tm\_id,

source\_text=source\_text,

target\_text=target\_text,

source\_lang=source\_lang,

target\_lang=target\_lang

)

print(f"已更新 tm\_id={tm\_id} 的记录")

def \_get\_selected\_row(self):

"""获取当前选中行，若无选中返回None"""

selected\_ranges = self.tableWidget\_item.selectedRanges()

if not selected\_ranges:

return None

return selected\_ranges[0].topRow()

**3.4.5项目管理**

### project\_page.py 文件实现逻辑分析 1. 文件概述

该文件实现了项目管理页面的核心功能，基于PyQt5构建，主要负责项目的创建、删除、列表展示以及跳转到文档管理页面等操作，是CAT工具的项目管理中枢模块。

2. 核心类 ProjectPage 分析 2.1 类初始化与UI布局

- 数据来源 ：通过 `get\_all\_projects` 从数据库获取所有项目

- 表格结构 ：7列设计，包含项目元数据和操作按钮

- 按钮绑定 ：为每行动态创建删除和进入文档按钮，并绑定对应事件处理函数

- 动态参数传递 ：使用 lambda 表达式传递项目ID给事件处理函数 2.2.2 文档页面跳转 enter\_document\_page

- 页面导航 ：通过主窗口引用调用 `show\_document\_page`

- 参数传递 ：将项目ID传递给文档页面，实现上下文关联

- 依赖条件 ：需确保 main\_window 引用有效 2.2.3 新建项目 on\_add\_project

- 安全机制 ：删除前弹出确认对话框，强调操作不可恢复

- 数据操作 ：调用 `delete\_project` 执行删除

- 界面同步 ：删除后刷新表格显示最新状态 3. 模块依赖关系

- 直接依赖 ：

- `project\_dao.py` ：项目数据访问

- PyQt5 ：UI组件和事件处理

- 间接依赖 ：

- 主窗口：提供页面切换接口

- 文档页面：接收项目ID并加载对应文档

# src/pages/project\_page.py

from PyQt5.QtWidgets import (

QWidget, QVBoxLayout, QTableWidget, QTableWidgetItem, QPushButton,

QLabel, QMessageBox, QHBoxLayout

)

from PyQt5.QtCore import Qt

import sys

import os

sys.path.append(os.path.join(os.path.dirname(\_\_file\_\_), '..'))

from dao import project\_dao

class ProjectPage(QWidget):

def \_\_init\_\_(self, main\_window=None):

super().\_\_init\_\_()

self.setWindowTitle("项目管理页面")

self.layout = QVBoxLayout()

self.main\_window = main\_window # 主窗口引用，用于页面跳转

self.layout.addWidget(QLabel("📁 项目管理页面"))

btn\_new = QPushButton("新建项目")

btn\_new.clicked.connect(self.on\_add\_project)

self.layout.addWidget(btn\_new)

self.table = QTableWidget()

self.layout.addWidget(self.table)

self.setLayout(self.layout)

self.refresh\_table()

def refresh\_table(self):

projects = project\_dao.get\_all\_projects()

self.table.setRowCount(len(projects))

self.table.setColumnCount(7) # 新增一列：进入文档

self.table.setHorizontalHeaderLabels(["项目ID", "项目名称", "源语言", "目标语言", "创建时间", "操作", "进入文档"])

for row, proj in enumerate(projects):

for col, value in enumerate(proj):

self.table.setItem(row, col, QTableWidgetItem(str(value)))

btn\_delete = QPushButton("删除")

btn\_delete.clicked.connect(lambda \_, pid=proj[0]: self.delete\_project(pid))

self.table.setCellWidget(row, 5, btn\_delete)

# 新增“进入文档”按钮

btn\_enter = QPushButton("进入文档")

btn\_enter.clicked.connect(lambda \_, pid=proj[0]: self.enter\_document\_page(pid))

self.table.setCellWidget(row, 6, btn\_enter)

self.table.resizeColumnsToContents()

def enter\_document\_page(self, project\_id):

"""

跳转到文档管理页面，传递当前项目ID

"""

if self.main\_window:

self.main\_window.show\_document\_page(project\_id)

def on\_add\_project(self):

from PyQt5.QtWidgets import QInputDialog

name, ok = QInputDialog.getText(self, "新建项目", "请输入项目名称：")

if not ok or not name.strip():

return

src\_lang, ok = QInputDialog.getText(self, "新建项目", "请输入源语言（如zh）：")

if not ok or not src\_lang.strip():

return

tgt\_lang, ok = QInputDialog.getText(self, "新建项目", "请输入目标语言（如en）：")

if not ok or not tgt\_lang.strip():

return

project\_dao.add\_project(name.strip(), src\_lang.strip(), tgt\_lang.strip())

QMessageBox.information(self, "提示", "项目创建成功！")

self.refresh\_table()

def delete\_project(self, pid):

reply = QMessageBox.question(

self, '确认删除', '确定要删除该项目吗？此操作不可恢复。',

QMessageBox.Yes | QMessageBox.No, QMessageBox.No

)

if reply == QMessageBox.Yes:

project\_dao.delete\_project(pid)

QMessageBox.information(self, "提示", "项目已删除。")

self.refresh\_table()

**3.4.6文档管理**

1. document\_page.py概述

该文件实现了文档管理页面的核心功能，基于PyQt5构建，主要负责项目文档的导入、管理、删除以及跳转到翻译页面等操作，是CAT工具中连接文档与翻译功能的关键模块。

2. 核心类 DocumentPage 分析 2.1 类初始化与UI布局

- 核心属性 ：

- project\_id ：当前操作的项目ID（默认为1，支持动态传入）

- main\_window ：主窗口引用，实现页面间跳转

- UI组件 ：垂直布局包含标题标签、导入按钮和文档表格

- 初始化流程 ：设置布局后立即调用 refresh\_table() 加载文档数据 2.2 核心方法详解 2.2.1 表格数据加载 refresh\_table

- 数据来源 ：通过 `get\_documents\_by\_project` 从数据库获取文档列表

- 表格结构 ：7列设计，包含文档元数据和操作按钮

- 按钮绑定 ：为每行动态创建删除和进入翻译按钮，并绑定对应事件处理函数

- 动态参数传递 ：使用 lambda 表达式传递文档ID给事件处理函数 2.2.2 文档导入 on\_import\_document

- 核心流程 ：

1. 文件选择：通过 QFileDialog 获取TXT文件路径

2. 文档入库：调用 `add\_document` 添加文档元数据

3. 内容处理：按空行分割文本，清理换行符并过滤空段落

4. 片段存储：通过 translation\_fragment\_dao.add\_fragment 将分段内容存入数据库

- 依赖模块 ： document\_dao 和 translation\_fragment\_dao 两个数据访问对象

- 用户反馈 ：操作完成后显示提示框并刷新表格 2.2.3 文档删除 delete\_document

- 安全机制 ：删除前弹出确认对话框防止误操作

- 数据操作 ：调用 `delete\_document` 执行删除

- 界面同步 ：删除后刷新表格显示最新状态 2.2.4 翻译页面跳转 enter\_translation\_page

- 页面导航 ：通过主窗口引用调用 `show\_translation\_page`

- 参数传递 ：将文档ID传递给翻译页面，实现上下文关联

- 依赖条件 ：需确保 main\_window 引用有效 3. 模块依赖关系

- 直接依赖 ：

- `document\_dao.py` ：文档数据访问

- `translation\_fragment\_dao.py` ：翻译片段数据访问

- PyQt5 ：UI组件和事件处理

- 间接依赖 ：

- 主窗口：提供页面切换接口

- 翻译页面：接收文档ID并加载对应内容

# src/pages/document\_page.py

from PyQt5.QtWidgets import (

QWidget, QVBoxLayout, QTableWidget, QTableWidgetItem, QPushButton,

QLabel, QMessageBox, QFileDialog

)

import sys

import os

sys.path.append(os.path.join(os.path.dirname(\_\_file\_\_), '..'))

from dao import document\_dao

class DocumentPage(QWidget):

def \_\_init\_\_(self, project\_id=1,main\_window=None):

"""

:param project\_id: 当前显示的项目ID（测试可以写死为1，后续页面跳转会自动传入）

"""

super().\_\_init\_\_()

self.project\_id = project\_id

self.main\_window = main\_window # 主窗口引用，用于页面跳转

self.setWindowTitle("文档管理页面")

self.layout = QVBoxLayout()

self.layout.addWidget(QLabel(f"📄 文档管理页面 - 项目ID: {self.project\_id}"))

# 导入文档按钮

btn\_import = QPushButton("导入文档")

btn\_import.clicked.connect(self.on\_import\_document)

self.layout.addWidget(btn\_import)

# 文档表格

self.table = QTableWidget()

self.layout.addWidget(self.table)

self.setLayout(self.layout)

self.refresh\_table()

def refresh\_table(self):

"""

刷新文档表格

"""

docs = document\_dao.get\_documents\_by\_project(self.project\_id)

self.table.setRowCount(len(docs))

self.table.setColumnCount(7) # 多一列用于操作

self.table.setHorizontalHeaderLabels(["文档ID", "文档名", "格式", "状态", "创建时间", "操作","进入翻译"])

for row, doc in enumerate(docs):

for col, value in enumerate(doc):

self.table.setItem(row, col, QTableWidgetItem(str(value)))

# 删除按钮

btn\_delete = QPushButton("删除")

btn\_delete.clicked.connect(lambda \_, doc\_id=doc[0]: self.delete\_document(doc\_id))

self.table.setCellWidget(row, 5, btn\_delete)

# 新增“进入翻译”按钮

btn\_edit = QPushButton("进入翻译")

btn\_edit.clicked.connect(lambda \_, doc\_id=doc[0]: self.enter\_translation\_page(doc\_id))

self.table.setCellWidget(row, 6, btn\_edit)

self.table.resizeColumnsToContents()

def on\_import\_document(self):

"""

选择文件并导入为文档（暂时只支持txt），同时按段落自动写入 translation\_fragments

"""

from dao import translation\_fragment\_dao # 导入你刚写的DAO

file\_path, \_ = QFileDialog.getOpenFileName(self, "选择文档", "", "文本文件 (\*.txt);;所有文件 (\*)")

if not file\_path:

return

file\_name = os.path.basename(file\_path)

file\_format = "txt"

# 1. 数据库添加文档（获得文档id）

from dao import document\_dao

document\_dao.add\_document(self.project\_id, file\_name, file\_format)

# 2. 获取新导入文档的id

docs = document\_dao.get\_documents\_by\_project(self.project\_id)

# 按 id 最大值/最新顺序获取最新一个

new\_doc\_id = docs[0][0] # [(id, name, ...), ...]，id在第一个元素

# 3. 读取文件内容并分段

with open(file\_path, "r", encoding="utf-8") as f:

content = f.read()

# 按“空行”分段，strip()移除首尾空格，splitlines()保证兼容性

raw\_paragraphs = content.strip().split("\n\n") # 两个换行符=空行

paragraphs = [p.strip().replace('\n', ' ') for p in raw\_paragraphs if p.strip()]

# 4. 写入 translation\_fragments 表

translation\_fragment\_dao.create\_table() # 保证表已存在

for idx, para in enumerate(paragraphs, 1):

translation\_fragment\_dao.add\_fragment(new\_doc\_id, idx, para)

QMessageBox.information(self, "提示", f"文档导入并分段成功！共{len(paragraphs)}段。")

self.refresh\_table()

def delete\_document(self, doc\_id):

"""

删除文档，二次确认

"""

reply = QMessageBox.question(

self, "确认删除", "确定要删除该文档吗？",

QMessageBox.Yes | QMessageBox.No, QMessageBox.No

)

if reply == QMessageBox.Yes:

document\_dao.delete\_document(doc\_id)

QMessageBox.information(self, "提示", "文档已删除。")

self.refresh\_table()

def enter\_translation\_page(self, document\_id):

"""

跳转到翻译编辑页面，传递文档ID

"""

if self.main\_window:

self.main\_window.show\_translation\_page(document\_id)