CIIMS 技术评审 PPT 大纲

校园智能库存管理系统 - **基于AI助手的桌面端**库存管理解决方案

项目全景图 (Technical Overview)

技术栈:

Frontend

PySide6 (Qt6 Python bindings)

Backend

Python 3.9+ with MySQL Connector

Database

MySQL 8.0

AI Integration

OpenRouter API (本地化部署)

Architecture

Desktop Application with Embedded AI Assistant

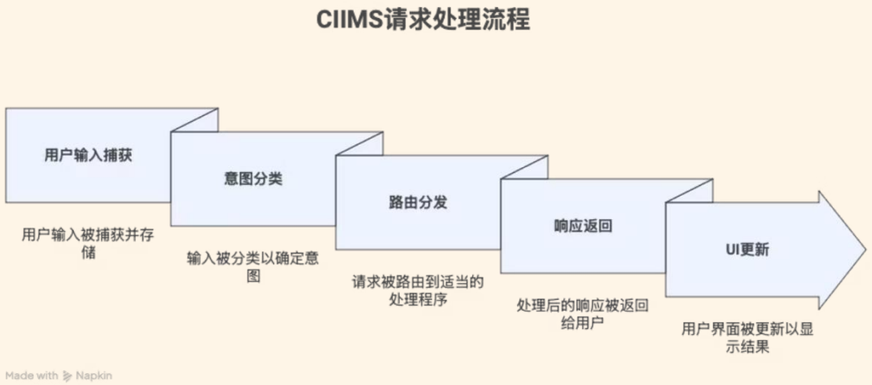
系统定位:

校园智能库存管理系统 (Campus Intelligent Inventory Management System) - **基于AI助手的桌面端**库存管理解决方案

核心模块:

- ui/screens/login_window.py - **身份**认证
- ui/screens/main_window.py - **主控制台**
- ui/screens/borrow_window.py - **物品借**阅
- ui/screens/return_window.py - **物品**归还
- ui/dialogs/ai_assistant_window.py - AI**助手核心**
- utils/assistant_service.py - AI**服**务层
- utils/assistant_data.py - SQL**生成与**验证
- utils/database_helper.py - **数据**库操作封装

AI Assist 核心架构 (The Brain)



系统位置:

嵌入式模块 (Embedded Module) - **集成在桌面**应用中的AI**助手,非独立服**务

请求生命周期:

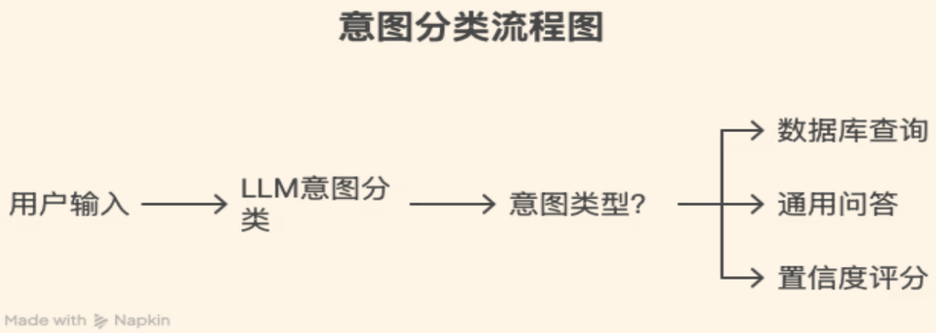
```
# 1. 用户输入捕获handle_send() -> _get_latest_user_prompt()
# 2. 意图分类intent_result = service.classify_intent(latest_prompt)
# 3. 路由分发if intent_type == "database_query":    result = _handle_database_query()else:    result = _handle_general_qa()
# 4. 响应返回finished.emit(result) -> _handle_success() -> UI更新
```

架构模式:

生产者-消费者模式	双路径处理	异步处理保证UI响应性
(QThread + Signal/Slot)	(General Q&A vs Database Query)	

意图识别与置信度逻辑 (Intent & Confidence)

核心判断机制:




```
def classify_intent(self, user_input: str) -> Dict[str, Any]:    # LLM-based 意图分类    classification_prompt = f"""    Classify this user request for CIIMS:    - "database_query": 数据查询类请求    - "general_qa": 通用问答类请求    - "confidence": 0.0-1.0 置信度评分    """        payload = {        "model": Config.OPENROUTER_MODEL,        "messages": messages,        "response_format": {"type": "json_object"}  # 强制JSON输出
}
```

置信度评分逻辑:

LLM 自评估置信度

(0.0-1.0)

默认阈值: 0.7

(隐含在分类逻辑中)

回退策略

分类失败时默认为 general_qa

代码片段展示:

```
# 置信度低于阈值时的回退处理except Exception as exc: return { "intent_type": "general_qa", "confidence": 0.3, "reasoning": "Classification failed, defaulting to general" }
```

查询构建与模糊处理 (Query Construction & Fuzzy Logic)

模糊匹配策略:

```
# assistant_data.py 中的模糊搜索指令instructions = f"""5. Use fuzzy search for semantic matching:   - For "books", "reading": 使用 LIKE '%book%' OR item_category IN ('books', 'textbooks')   - For "laptop", "computer": 使用 LIKE '%laptop%' OR item_category IN ('electronics')   - Always search both item_name and item_category columns   - Use LOWER() function for case-insensitive matching"""
```

NL到SQL转换流程:

01

上下文构建

```
sql_context = build_runtime_context(user_role, user_name)
```

02

SQL提案生成

```
first_reply = service.ask(history, sql_context)
```

03

SQL提取与验证

```
payload = extract_sql_payload(content)
```

```
proposed_sql = payload.get("proposed_sql", "")
```

04

安全执行

```
if proposed_sql: follow_up = _handle_sql_follow_up(proposed_sql, sql_context, first_reply)
```

实体抽取机制:

- **基于**语义理解的模糊匹配
- **多列搜索**: item_name + item_category
- **大小写不敏感**: LOWER() 函数包装

安全与权限控制 (Security & RBAC)

权限校验机制:

```
# assistant_data.py - 角色访问控制ROLE_ACCESS: Dict[str, List[str]] = {    "admin": ["items", "borrows", "users"],    "user": ["items", "borrows"],  # 普通用户不能访问用户表}def validate_sql(sql: str, user_role: str) -> Tuple[bool, Optional[str]]:    # 1. SQL类型检查 (仅允许SELECT)    if not upper_stmt.startswith("SELECT"):        return False, "Only SELECT statements are allowed"        # 2. 关键词拦截    for keyword in SQL_BLOCKLIST:  # ["DELETE", "UPDATE", "INSERT", "DROP"]        if keyword in upper_stmt:            return False, f"Keyword '{keyword}' is not allowed"        # 3. 表级权限验证    tables = _extract_tables(statement)    allowed = set(ROLE_ACCESS.get(user_role, ROLE_ACCESS["user"]))    if not tables.issubset(allowed):        return False, f"Forbidden table(s): {forbidden}"
```

拦截逻辑:


```
def execute_safe_sql(sql: str, user_role: str, user_name: str):    ok, message = validate_sql(sql, user_role)    if not ok:        raise ValueError(message or "Invalid SQL")  # 拦截并抛出异常        # 占位符替换 (防止SQL注入)    prepared_sql = message    placeholder_values = _resolve_placeholders(user_name)    for key, value in placeholder_values.items():        prepared_sql = prepared_sql.replace(key, value)
```

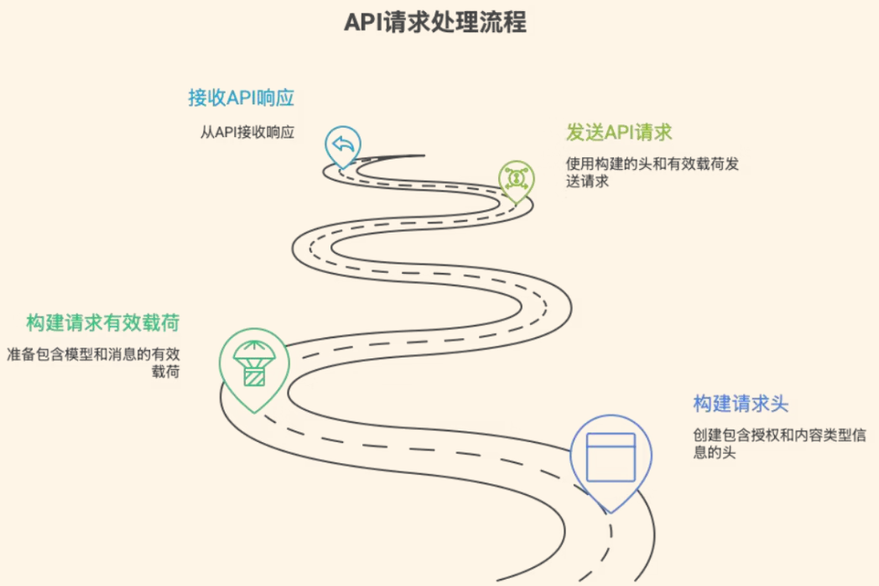
安全级别:

接口级控制 (API-level) + **表**级权限 (Table-level RBAC)

API 交互与本地化部署 (Local LLM Integration)

本地部署方案:

```
# config.py - API配置OPENROUTER_BASE_URL = "http://127.0.0.1:7897"  # 本地代理地址OPENROUTER_MODEL = "deepseek/deepseek-chat"  # 本地模型PROXY_URL = "http://127.0.0.1:7897"  # 代理配置
```



通信层实现:

```
def _build_headers(self) -> Dict[str, str]:    headers = {        "Authorization": f"Bearer {Config.OPENROUTER_API_KEY}",        "Content-Type": "application/json",        "HTTP-Referer": Config.OPENROUTER_REFERER,        "X-Title": Config.OPENROUTER_TITLE    }def ask(self, history, runtime_context="", reasoning_enabled=False):    payload = {        "model": Config.OPENROUTER_MODEL,        "messages": self._build_messages(history, runtime_context)    }        if reasoning_enabled:        payload["reasoning"] = {"enabled": True}        response = self.client.post(        self.endpoint,        headers=self._build_headers(),        json=payload,        timeout=60    )
```

超时与错误处理:

连接超时	重试机制	错误分类
60秒	无 (快速失败策略)	HTTP错误 vs 服务错误 vs 业务错误

总结与技术挑战

当前架构优势:

嵌入式设计

无需额外服务部署

双路径处理

智能路由分发

严格的SQL安全验证机制

多语言自适应支持

性能瓶颈:

1. LLM调用延迟

每次意图分类需要额外API调用

2. 同步SQL执行

大查询可能阻塞UI线程

3. 内存占用

完整对话历史在内存中维护

下一步优化方向:

意图分类缓存

基于用户历史模式预测意图

流式响应

实现实时响应流显示

异步SQL执行

使用连接池 + 异步I/O

本地模型优化

量化模型减少推理延迟

技术债务:

- 异常处理粒度过粗 (大量 broad Exception)
- 配置管理分散在多个文件
- 缺乏完整的单元测试覆盖