ChatGPT

Mini-Bot CLI 版本实现详解

下面根据项目给出的结构，详细说明各个主要文件中的类定义、函数功能，以及它们之间如何交互。在实现中我们将优先实现 **CLI**（命令行）版本，并在代码中添加中文注释以提高可读性。

项目概览与 CLI 流程

Mini-Bot 项目旨在构建一个支持聊天和工具调用的本地聊天机器人。CLI 版本通过命令行与用户交互，实现以下基本流程：

1. **读取配置：** 从 `config/settings.json` 加载配置参数（如模型名称、Ollama 服务端口、消息token裁剪阈值等）。
2. **初始化核心组件：** 包括 `ChatSession`（管理聊天记录）、`OllamaClient`（与本地模型接口通信）、`ToolRegistry`（注册可用工具）以及将它们整合的 `BotService`。
3. **注册工具函数：** 通过 `ToolRegistry` 注册本地工具，例如天气查询和计算器。这些工具的名称、参数格式和实现函数会提供给模型使用。
4. **开始对话循环：** 在 CLI 中读取用户输入，将其发送给 `BotService` 处理：
5. `BotService` 会先将用户消息保存到 `ChatSession` 历史。
6. 调用 `OllamaClient` 向本地模型（Ollama）发送对话请求，包含当前聊天记录和工具列表，启用**流式**响应。
7. **流式接收模型响应：** 模型可能直接返回文本答案，也可能请求调用某个工具。`BotService` 边接收边处理：
    - 当接收到普通对话内容时，实时输出（流式打印）给用户。
    - 如果检测到模型的**工具调用请求**（包含函数名和参数），则暂停接收，调用对应的本地工具函数获取结果。
    - 将工具执行结果作为特殊消息追加到对话历史，然后继续请求模型生成后续回答（再次通过 `OllamaClient` 调用模型），使模型利用工具结果继续回答。
8. 如此循环，直到模型给出最终回答内容并结束对话响应。最后将完整回答打印给用户。
9. **循环等待下一次用户输入**，直到用户退出。

以上流程实现中，各模块通过清晰的接口协作。下面分模块详细说明各文件中的类和函数设计。

ChatSession.hpp / ChatSession.cpp – 聊天会话管理

ChatSession 类负责维护多轮对话的历史记录，并提供消息裁剪和Token统计等功能，以防止上下文过长。主要特性包括：

- **对话消息存储：** 使用适当的数据结构（例如 `std::vector`）保存消息序列，每条消息包含角色和内容。典型的角色有：
- `"user"`：用户消息
- `"assistant"`：AI助手回复
- `"tool"`：工具返回结果（用于函数调用机制）
  （可以定义一个结构体如 `Message` 存储 `role` 和 `content` 字段。）

- **Token 计数与裁剪：** 跟踪当前会话的Token数量。当新增消息使总Tokens超过阈值时，裁剪最早的消息以保持上下文长度在允许范围。`settings.json`中可能有配置如`"max_tokens"`或`"clip_threshold"`指定阈值。

主要成员变量 (private)：

```
class ChatSession {
private:
  struct Message {
    std::string role;   // 消息角色: "user"/"assistant"/"tool"等
    std::string content; // 消息内容文本
    // 可以视需要加入其他字段，比如函数名（对于tool角色消息，可以扩展一个name字段）
  };
  std::vector<Message> messages;  // 存储所有对话消息
  size_t tokenCount;          // 当前消息总的Token估算数
  size_t maxTokens;           // Token阈值（从配置加载）
  // 其他辅助成员，例如用于估计Token的函数指针或工具
};
```

主要成员函数：

- **构造函数** `ChatSession(size_t maxTokens)`: 接收最大Token数阈值进行初始化。
  实现： 初始化`maxTokens`，将`tokenCount`设为0，`messages`清空。

- `void addUserMessage(const std::string& content)`：添加一条用户消息到历史。
  实现：

- 创建`Message`对象`{role: "user", content: content}`，追加到`messages`末尾。
- 使用`util.hpp`中的Token估算工具计算此消息的Token长度，累加到`tokenCount`【註：Token估算可基于字符长度或调用模型的分词库】。

- 调用内部裁剪函数（如`trimHistory()`），若当前`tokenCount`超出`maxTokens`，则从最早的消息开始移除，直到`tokenCount`降至阈值以下。

- `void addAssistantMessage(const std::string& content)`：添加一条助手(AI)回复消息。流程与`addUserMessage`类似，只是`role`为`"assistant"`。

- `void addToolResultMessage(const std::string& toolName, const std::string& result)`：添加一条工具函数执行结果消息。
  实现： 创建`Message` `{role: "tool", content: result}`并可能附加工具名，用于告知模型这是哪个函数的输出。例如可以在Message结构中新增`name`字段存放工具名。将此消息追加到历史，更新token计数并裁剪。【在Ollama工具功能中，使用角色"tool"携带结果和函数名是约定做法，模型会据此继续回答【9†L171-L179】[1]】。

- `std::vector<nlohmann::json> getMessagesJson() const`：生成当前会话消息的JSON数组，以便发送给Ollama接口。
  实现： 遍历`messages`向量，把每条消息转换为JSON对象，例如：

```
{"role": "user", "content": "Hello"}
{"role": "assistant", "content": "Hi, I'm an AI."}
{"role": "tool", "name": "get_current_weather", "content": "20°C, Sunny"}
```

注意对于 `tool` 角色消息，包含工具函数名称字段 `name`，以匹配Ollama对工具结果的识别要求 [1]。最终返回JSON对象数组。

- **`size_t getTokenCount() const`**：获取当前Token计数。
- **`void trimHistory()`**：内部辅助函数，用于裁剪过老的消息。当 `tokenCount > maxTokens` 时，按照先入先出的顺序删除 `messages` 开头的消息，并相应减去其Token数，直到满足阈值。
  说明： 可以确保至少保留最近若干条对话，以免上下文完全丢失。

模块间调用：
`ChatSession` 提供简洁的接口供外部使用。例如 `BotService` 每当有新用户输入或新AI回复时，就调用 `ChatSession.addUserMessage(...)` 或 `addAssistantMessage(...)` 记录下来。在发送请求给模型之前，`BotService` 调用 `getMessagesJson()` 获取历史记录的JSON表示。裁剪逻辑在新增消息时自动进行，对外透明。

OllamaClient.hpp / OllamaClient.cpp – 模型客户端封装

OllamaClient 类封装了通过HTTP接口与本地 Ollama 模型服务交互的细节，利用库 **cpr** 发送请求并处理响应。该类支持**流式回调**，即逐步接收模型输出，以便及时处理工具调用和实时显示。

主要成员变量：

```
class OllamaClient {
private:
  std::string baseUrl;  // Ollama API 基础URL，例如 "http://localhost:11434"
  std::string modelName; // 使用的模型名称（从配置读取, 如 "qwen3" 或 "llama2" 等）
  // 其他配置：如是否启用流式(stream)等，可直接在请求中指定
};
```

构造函数：
`OllamaClient(const std::string& baseUrl, const std::string& modelName)`：设置模型服务URL和模型名称。
说明： `baseUrl` 通常形如 `http://localhost:<port>/api/chat` 的主机和端口部分，端口由配置提供（默认为11434）。`modelName` 由配置提供，指定默认使用的模型。

主要成员函数：

- **`void sendChatRequest(const std::vector<nlohmann::json>& messages, const std::vector<nlohmann::json>& tools, std::function<bool(const nlohmann::json&)> onChunk)`**：发送聊天请求并流式处理响应。
  参数说明：
- `messages`：对话消息列表（JSON数组形式，由 `ChatSession.getMessagesJson()` 提供）。

- `tools`：工具列表的JSON数组定义（由`ToolRegistry`提供，每个元素包含工具schema，如名称、参数类型等）。
- `onChunk`：回调函数，接受每一块流式数据解析后的JSON对象。当回调返回`false`时，中断流读取（用于在需要时提前终止，例如遇到工具调用需要暂停）。
  实现思路：

  1. **准备请求数据：** 使用 nlohmann::json 组装 POST 请求体，包括模型名称、messages数组、tools数组以及`"stream": true`选项。例如：

     ```
     {
      "model": "<modelName>",
      "messages": [ ... ],
      "tools": [ ... ],
      "stream": true
     }
     ```

  2. **发送 HTTP POST 请求：** 利用 cpr库执行 POST。例如：

     ```
     cpr::Response response = cpr::Post(cpr::Url{baseUrl + "/api/chat"},
                        cpr::Body{request.dump()},
                        cpr::Header{{"Content-Type", "application/json"}},
                        cpr::WriteCallback{...});
     ```

     这里使用 cpr 的 **WriteCallback** 来实现流式读取 [2]。设置一个回调函数，每当有一段响应数据到达，就调用我们提供的lambda。在该lambda中，我们负责将字节流拼接并解析为完整JSON对象。
  3. **流式解析响应：** Ollama 的响应是**连续的 JSON 块**流。例如模型在回答天气问题时，可能流式返回多个JSON：一些包含回答片段，还有一个包含工具调用指令 [3] [4]。
     我们可以在 WriteCallback 累积数据，当检测到完整的JSON块（通常以换行或`}`结尾）时，用 nlohmann::json 解析。对每个解析出的 JSON 对象，调用`onChunk(jsonObj)`回调处理。
  4. 如果`onChunk`返回 true，则继续读取后续数据。
  5. 如果`onChunk`返回 false，则说明调用方决定中止流（例如模型请求了工具调用，需要暂停等待工具结果），此时可以让 WriteCallback 返回 false 以终止 HTTP流 [5]。
  6. **处理结束：** 当HTTP响应完整结束或者被我们手动终止，函数退出。对于正常回答结束，最后一个JSON会有`"done": true`标志，我们也应结束循环。

- **`std::string sendChatRequestSync(..., ...)`**（可选）：视情况也可提供一个同步获取完整响应的函数，不启用stream。这在不需要工具调用时可直接返回整个回答文本。但由于我们需要支持工具调用的中途交互，CLI实现主要采用上述`sendChatRequest`流式方式。

模块间调用：
`BotService`会调用`OllamaClient.sendChatRequest(...)`来与模型交互。它会构建好 messages 和 tools 的JSON，并提供一个 onChunk 回调，用于实时处理模型输出：例如调用工具或将部分回答输出到终端。`OllamaClient`并不知道上层业务逻辑，只负责HTTP通信和数据流解析，将每个JSON片段传给回调函数处理。这样设计使得对于UI/WebSocket等不同上层，处理方式可以通过不同回调灵活应对。

ToolRegistry.hpp / ToolRegistry.cpp – 工具注册与管理

ToolRegistry 类管理本地可调用工具函数的**注册**和**调度**。模型通过调用这些工具，获取额外的信息或计算结果（类似 OpenAI 的函数调用能力）。ToolRegistry 提供工具元数据（名称、参数说明等）给模型，并在运行时根据模型请求调用相应的C++函数实现。

主要成员结构和变量：

```
// 工具信息结构
struct ToolInfo {
  std::string name;     // 工具函数名称（供模型调用）
  std::string description; // 描述
  nlohmann::json paramSchema; // 参数JSON模式定义 (JSON Schema 格式)
  std::function<std::string(const nlohmann::json&)> handler; // 工具实现函数，传入参数JSON，返回结果字符串
};
class ToolRegistry {
private:
  std::unordered_map<std::string, ToolInfo> tools; // 已注册工具映射表(name -> ToolInfo)
public:
  // ... 构造/单例等
};
```

主要功能函数：

- `void registerTool(const std::string& name, const std::string& description, const nlohmann::json& paramSchema, std::function<std::string(const nlohmann::json&)> handler)`：注册一个工具。
  实现： 将工具名称和对应的ToolInfo存入 `tools` 映射，以名称为键方便查找。`paramSchema` 定义了此函数的参数要求（采用 JSON Schema 格式，与Ollama工具接口匹配 6 7 ），`handler` 是实际执行的C++函数。
  示例： 对于天气查询工具：

  ```
  nlohmann::json schema = {
    {"type", "object"},
    {"properties", {
      {"location", {{"type", "string"}, {"description", "The location to get weather for"}}},
      {"format", {{"type", "string"}, {"description", "Temperature format (celsius or fahrenheit)"}}}
    }},
    {"required", {"location", "format"}}
  };
  toolRegistry.registerTool(
    "get_current_weather",
    "Get the current weather for a location",
    schema,
    [](const nlohmann::json& args)->std::string {
      // 从args中取出location和format，调用天气API或返回模拟数据
  ```

```
    std::string loc = args.at("location");
    std::string fmt = args.at("format");
    // 这里可以调用真实天气API获取数据，演示中返回虚拟结果:
    if (fmt == "celsius")
      return loc + "当前天气：25°C，晴";
    else
      return loc + " current weather: 77°F, Sunny";
  }
);
```

此注册函数通常在程序初始化时调用，一般由各具体工具模块（如`tool_weather.cpp`）内部执行。

- **`ToolInfo* getToolInfo(const std::string& name)`**：根据名称获取工具信息结构指针。
用途： `BotService`在收到模型请求要调用某工具时，会通过名称从注册表查询对应的ToolInfo（尤其需要其中的handler以调用实现）。

- **`std::vector<nlohmann::json> getToolDefinitionsJson() const`**：获取所有工具的JSON定义列表，供发送给模型接口。
实现： 遍历已注册工具，将每个`ToolInfo`转换为Ollama API期望的格式：

```
{
 "type": "function",
 "function": {
  "name": <name>,
  "description": <description>,
  "parameters": <paramSchema>
 }
}
```

将这些 JSON 对象加入一个数组并返回 [8] [9]。模型会据此了解有哪些函数可调用以及参数要求。

模块间交互：
- 初始化时，`ToolRegistry`由各工具cpp调用`registerTool`添加工具。
- 在对话过程中，当`BotService`检测到模型输出包含`tool_calls`请求时，会使用`ToolRegistry.getToolInfo(name)`找到对应工具，然后调用其中的`handler`执行，并获取结果。
- 同时，在每次发送请求给模型时，`BotService`会调用`ToolRegistry.getToolDefinitionsJson()`将工具列表提供给模型，使其有权限调这些函数。

关于工具参数与结果：
- 参数传递： 模型在需要工具时，会在输出JSON中包含所调用函数的名称和参数。例如：

```
"tool_calls": [ { "function": { "name": "get_current_weather", "arguments": { "location": "Toronto", "format": "celsius" } } } ]
```

`BotService`解析到这个内容后，将`arguments`部分作为 nlohmann::json 传给对应ToolInfo.handler执行。
- 结果返回： 工具执行完成后返回一个字符串结果。`BotService`会把这个结果封装成一个带有角色`"tool"`的消息添加到`ChatSession` [1]。稍后会把该结果也发送回模型，让模型继续它原本未完成的回答。

tool_weather.cpp – 天气查询工具实现

文件职责： 注册并实现“天气查询”工具。假设模型调用函数名为 `"get_current_weather"`，传入参数包括地点和温度单位，返回对应地点当前天气。

实现内容：

- 包含必要头文件：`ToolRegistry.hpp`（用于注册）、`cpr`或其他HTTP库（如果需要真实API请求），`nlohmann/json.hpp`（处理JSON参数）等。
- 定义工具的参数JSON Schema：如上节所示，包括 `"location"` 和 `"format"` 字段，类型分别为string。
- **实现工具函数**（lambda或普通函数形式皆可）：可以通过外部API获取真实天气，这里可使用**示例实现**返回模拟数据。比如对每个请求直接返回“`<地点> <天气描述>`”。为了简单，可以固定返回，如 `20°C，晴` 或根据 format 返回摄氏/华氏温度。
  （如需真实数据，可集成公共天气API并将API调用写在handler内，注意这需要网络请求与JSON解析，此处可暂时不深入。）
- 调用 `ToolRegistry::registerTool(...)` 完成注册。可能采取的方式：
- 定义一个初始化函数，如 `registerWeatherTool(ToolRegistry& registry)`，在其中构造schema和lambda后调用 `registry.registerTool(...)`。然后在主程序初始化时调用此函数。
  或
- 利用全局/静态对象：例如定义一个静态对象，其构造函数执行注册，使得在模块加载时自动注册工具。（需确保注册顺序问题，这里更直观的方式是由主程序调用。）

工具函数参数与返回：
`handler` 会接受模型传来的参数JSON，例如 `{"location": "Toronto", "format": "celsius"}`，函数应解析出 `location="Toronto"` 和 `format="celsius"`。根据这些参数，返回相应结果字符串（例如“Toronto当前天气：25°C，晴”）。

中文注释示例： 工具实现部分可能这样编写：

```
// 注册 "get_current_weather" 工具
void registerWeatherTool(ToolRegistry& registry) {
  // 定义参数JSON Schema
  nlohmann::json schema = {/* ... 构造如上 ... */};
  // 注册工具
  registry.registerTool(
    "get_current_weather",
    "Get the current weather for a location",
    schema,
    [](const nlohmann::json& args) -> std::string {
      // 提取参数
      std::string loc = args.value("location", "");
      std::string fmt = args.value("format", "celsius");
      // 调用实际天气服务（此处简化为模拟数据）
      if (loc.empty()) return "Location not provided.";
      std::string result;
      // 根据格式返回不同单位（这里只是示例固定值）
      if (fmt == "fahrenheit")
        result = loc + " Weather: Sunny, 77°F";
      else // 默认 celsius
```

```
            result = loc + " 天气：晴，25°C";
        return result;
    }
  );
}
```

tool_calc.cpp – 简易计算器工具实现

文件职责： 注册并实现“计算”工具。它可以执行简单算术计算，供模型在需要数学计算时调用。可能的函数名称为 `"calculate"` 或 `"add_two_numbers"` 等。

实现内容：

- 定义工具参数Schema：例如实现一个加法/减法函数，可以有两个参数：

```
{
 "type": "object",
 "properties": {
  "a": { "type": "number", "description": "第一个操作数" },
  "b": { "type": "number", "description": "第二个操作数" },
  "operation": { "type": "string", "description": "操作: add或sub", "enum":
["add","subtract"] }
 },
 "required": ["a","b","operation"]
}
```

这样模型可以请求此函数计算 `a` 和 `b` 的和或差。（或者更加简单地，只实现加法，不需要operation参数）

- **实现工具函数：** 解析参数，进行相应计算。例如：

```
registry.registerTool(
  "calculate",
  "Calculate a basic arithmetic operation (add/subtract) on two numbers",
  schema,
  [](const nlohmann::json& args) -> std::string {
    double a = args.value("a", 0.0);
    double b = args.value("b", 0.0);
    std::string op = args.value("operation", "add");
    double result = 0.0;
    if (op == "subtract") result = a - b;
    else result = a + b;
    return std::to_string(result);
  }
);
```

这样，当模型调用例如 `calculate` 函数，给定 `{"a": 3, "b": 1, "operation": "subtract"}`，我们计算得到 `2` 并返回字符串 `"2"`。【在Ollama的Python示例中，他们提供了类似 `add_two_numbers` 函数供模型调用【1†L130-L139】[10]。】

- 或者，实现为只做加法的示例工具：参数仅有两个数字，函数直接返回两数之和。这样Schema和代码会更简单。

用法说明：
模型在需要计算时会输出工具请求，例如：

```
"tool_calls": [ { "function": { "name": "calculate", "arguments": { "a": 3, "b": 1, "operation": "subtract" } } } ]
```

`BotService` 捕获到后，会调用此工具的handler计算出结果 `2`，并把它作为 `"tool"` 角色消息发送回模型上下文，让模型利用结果继续回答问题。

BotService.hpp / BotService.cpp – 聊天机器人服务核心

BotService 类将聊天会话、模型客户端和工具调用整合起来，充当应用主控制器。它负责处理用户输入，调用模型获取回复，并处理可能的工具调用。对于CLI版本，BotService直接与控制台交互（读取用户输入、打印输出）；对于UI版本，BotService还会通过WebSocket与前端通信。

主要成员变量：

```
class BotService {
private:
  ChatSession session;     // 管理会话历史
  OllamaClient client;     // 模型接口客户端
  ToolRegistry toolRegistry; // 工具注册表
  // （如果需要WS，可有WebSocket服务器相关成员，但CLI版可忽略）
};
```

说明： `BotService` 可以将 `ToolRegistry` 作为成员，或在内部以单例使用。初始化时会注册所有工具。

构造函数：
`BotService(const ChatSession& session, const OllamaClient& client, const ToolRegistry& registry)`：构造时注入已有的会话管理、模型客户端和工具库。也可以在内部构造，但为了灵活和测试，注入依赖更好。

初始化工具：
在构造后，调用各工具模块的注册函数将工具加入ToolRegistry。例如：

```
// BotService 构造完成后
registerWeatherTool(toolRegistry);
registerCalcTool(toolRegistry);
```

这样保证在对话开始前工具列表已就绪。

主要成员函数：

- `std::string processUserMessage(const std::string& userInput)`：处理一次用户输入，并返回最终的模型回答。
  实现流程：
- **记录用户消息：** 调用 `session.addUserMessage(userInput)` 保存到会话历史。
- **准备请求模型：** 从 `session` 获取消息JSON列表；从 `toolRegistry` 获取工具定义JSON列表。
- **发送请求并流处理：** 使用 `client.sendChatRequest(messages, tools, onChunkCallback)`.
  - onChunkCallback 是一个lambda，捕获 BotService 和一些输出缓冲，用于处理每个到来的JSON片段：
  - 检查 JSON 中是否有 `"tool_calls"` 字段：
    - **如果有工具调用请求：** 提取函数名和参数构成的 JSON。然后：
      a. 在控制台打印一个提示（例如“[调用工具] 模型请求使用工具X...正在查询”），以告知用户过程。
      b. 查找对应 ToolInfo： `auto* toolInfo = toolRegistry.getToolInfo(name)`，确保不为空。
      c. 调用 `toolInfo->handler(argumentsJson)` 执行工具函数，得到结果字符串。
      d. 将结果打印或记录日志（CLI下可打印，比如“→ 工具返回: ...”）。
      e. **停止当前流**：令回调返回false，中止 `sendChatRequest`（因为模型等待工具结果，我们需要重新调用模型）。
      f. 将工具结果加入会话： `session.addToolResultMessage(name, resultStr)`。
      g. **二次请求模型：** 准备包含新messages的请求，再次调用 `sendChatRequest`，但这一次不需要附加tools列表了吗？（其实tools列表还是要提供，以防模型后续仍可调用其他工具）。继续使用相同的onChunkCallback处理后续。
      （可以用循环来实现：外层while，当检测到tool_calls时循环再次请求模型。）
    - **如果没有tool_calls：** 正常的模型内容输出：
    - 提取 `json["message"]["content"]` 文本，如果非空，则将其追加到输出缓冲区。为了在CLI实时显示，也可以**逐步打印**该文本片段并刷新stdout，这样用户可以实时看到模型回答逐字出现。
    - 当 `json["done"] == true` 时，表示模型回答结束，onChunkCallback可以返回true并让流正常结束。我们在此可以结束外层对模型调用的循环。
  - 其它信息：模型输出中可能还包含一些辅助内容（例如 `<think>` 标签等思考过程标记），可以选择过滤或直接显示。通常 `<think>` ... `</think>` 之间的内容可能是模型内部思考，不需要全部展示，可以按需求处理。
  - **返回值：** onChunkCallback 对普通内容片段返回 true 表示继续，遇到工具请求返回 false 触发中断。
- **输出最终结果：** 累积的回答文本在函数返回前可以整理一下。由于我们在过程中可能已经实时打印，这里也可以直接返回完整字符串用于上层使用（在CLI中其实不再需要，因为已经打印，但为了接口完备还是返回）。

循环调用与状态管理： BotService需要能够处理**多轮工具调用**的情形。例如模型可能连续调用多个工具：则每次都中断请求、执行工具、追加结果、再继续请求模型，直到模型最终完成回答 [11] [12]。可以实现为一个 `while` 循环：

```
std::string answer;
bool needToolCall = true;
while (needToolCall) {
  needToolCall = false;
  client.sendChatRequest(messages, tools, [&](const nlohmann::json& chunk){
    // 检查chunk，如果有tool_calls:
    if (chunk.contains("message") && chunk["message"].contains("tool_calls")) {
```

```
            // ... 执行上述工具调用流程
            needToolCall = true;
            return false; // 中止当前流
        }
        // 累积正常内容:
        std::string content = chunk["message"]["content"].get<std::string>();
        answer += content;
        if (!content.empty()) {
            std::cout << content; // 实时输出
            std::fflush(stdout);
        }
        // 如果结束:
        if (chunk.value("done", false)) {
            // 回答完成
        }
        return true;
    });
    // 若needToolCall为true, loop继续，会话已包含工具结果，新一轮调用会接着回答剩余部分
}
std::cout << std::endl; // 完整回答换行
return answer;
```

如上，使用`needToolCall`标志控制循环，当一次调用过程中触发了工具请求，标志置true，则在执行完工具后进入下一轮模型请求继续回答。【注：每次调用都使用更新后的messages和原始tools列表】

- **（可选）** `void startCliLoop()`：在CLI模式下启动交互循环，不断读取用户输入并调用`processUserMessage`。
  实现： 简单的`while(true)`读取标准输入字符串，如果为空或等于某个退出命令则跳出，否则调用`processUserMessage(input)`获取/打印回答。这个函数也可以直接在 main 中实现，不一定作为类成员。

模块间交互：
`BotService`几乎连接了所有模块： - 使用`ChatSession`保存对话并管理Token。 - 通过`OllamaClient`与模型对话，`ToolRegistry`提供工具定义及工具处理函数。 - `BotService`也是 CLI 主程序和 UI 层进行交互的接口。在CLI下，它直接和`main_cli.cpp`交互；在UI下，它可能通过WebSocket事件处理用户消息和返回结果。

main_cli.cpp – CLI 主程序入口

文件职责： 初始化整个应用并运行命令行下的演示。对于最小Demo，此文件会创建所需对象并进行一次简单对话，或实现一个交互循环。

主要实现步骤：

1. **加载配置：** 使用 nlohmann::json 读取`config/settings.json`文件。解析所需字段，例如：
2. `port`（如 11434），
3. `model`（模型名称，如 "qwen3"），
4. `clip_threshold`（最大Token阈值），
5. 其他设置（UI主题可忽略）。
   例如：

```
std::ifstream configFile("config/settings.json");
nlohmann::json settings;
configFile >> settings;
int port = settings.value("port", 11434);
std::string model = settings.value("model", "qwen3");
size_t maxTokens = settings.value("clip_threshold", 2048);
```

读到的配置用于后续初始化。

6. **初始化核心对象：**

7. 创建 `ChatSession session(maxTokens)` 使用配置的Token上限。
8. 创建 `OllamaClient client("http://localhost:" + std::to_string(port), model)`. （假设Ollama提供HTTP API，此URL前缀指向聊天接口）
9. 创建 `ToolRegistry toolRegistry` 对象。
10. 调用各工具注册函数，注册工具到 `toolRegistry`：

```
registerWeatherTool(toolRegistry);
registerCalcTool(toolRegistry);
```

11. 创建 `BotService bot(session, client, toolRegistry)` 将上述对象组合。

12. **开始CLI对话：**
打印欢迎信息，例如 `"欢迎使用 Mini-Bot CLI. 输入内容开始对话，输入 'exit' 退出."`。然后进入循环：

```
std::string input;
while (std::getline(std::cin, input)) {
  if (input == "exit" || input == "quit") break;
  if (input.empty()) continue;
  std::string answer = bot.processUserMessage(input);
  // （通常processUserMessage已打印输出，这里answer可不使用或用于其他目的）
}
```

通过 `std::getline` 获取用户整行输入。如果用户输入退出命令则跳出循环。否则将输入传给 `BotService` 处理并得到回答。由于 `BotService` 内部已负责打印模型回答，`main` 处可以不用重复打印，仅根据需要对返回值做处理（或直接忽略返回值）。

13. **结束清理：** 循环结束后，打印告别信息并正常退出程序。

CLI 版本特点：
- 所有输入输出都在终端，`BotService.processUserMessage` 内部使用 `std::cout` 实时打印模型输出（逐字流式打印）。
- 简化处理：没有并发需求，不需要考虑多用户，直接顺序执行工具调用和模型交互。
- 错误处理：应考虑基本的错误处理，例如：如果 `OllamaClient` 请求失败或网络错误，需提示用户模型不可用；如果模型输出无法解析JSON，也需处理异常（可以简单跳过该chunk或终止回答）。

中文注释示例： main_cli.cpp 主要逻辑可能如下：

```
int main() {
    // 1. 加载配置
    std::ifstream cfg("config/settings.json");
    nlohmann::json settings;
    cfg >> settings;
    int port = settings.value("port", 11434);
    std::string model = settings.value("model", "qwen3");
    size_t maxTokens = settings.value("clip_threshold", 2048);
    // 2. 初始化核心组件
    ChatSession session(maxTokens);
    OllamaClient client("http://localhost:" + std::to_string(port), model);
    ToolRegistry registry;
    registerWeatherTool(registry);
    registerCalcTool(registry);
    BotService bot(session, client, registry);
    // 3. 进入交互循环
    std::cout << "Mini-Bot 已启动 (CLI 模式)。输入您的问题:\n";
    std::string userInput;
    while (true) {
        std::cout << "\n> ";  // 提示符
        if(!std::getline(std::cin, userInput)) break;  // 读入用户输入
        if(userInput == "exit" || userInput == "quit") {
            std::cout << "再见！ \n";
            break;
        }
        if(userInput.empty()) continue;
        bot.processUserMessage(userInput);
        // 结果已在 bot 内输出
    }
    return 0;
}
```

以上展示了CLI模式下各模块如何组装和运行。

结语

通过以上设计，我们详细补充了 **CLI 版本** mini-bot 项目各文件/模块中应实现的类和函数，以及它们之间的调用关系和数据传递方式。**ChatSession** 管理对话上下文和Token长度、**OllamaClient** 负责与LLM服务通信并支持流式回调、**ToolRegistry** 提供工具的注册和查找、**各工具模块** 定义具体工具逻辑，**BotService** 整合所有部分，实现对话和工具调用流程。最终在 **main_cli.cpp** 中初始化并运行，从而完成一个可在终端交互的本地聊天机器人。

上述实现添加了丰富的中文注释和说明，方便理解各部分的功能。这样，先实现的CLI版本可以作为基础进行测试验证，之后再扩展到 Qt 本地界面或 Web 界面，只需在 `BotService` 中添加 WebSocket 通信等功能即可，无需改变核心逻辑。

---

1 11 12 Ollama Tool support (aka “Function Calling”) | by Laurent Kubaski | Medium
https://medium.com/@laurentkubaski/ollama-tool-support-aka-function-calling-23a1c0189bee

2 5 cpr - Advanced Usage
https://docs.libcpr.org/advanced-usage.html

3 4 6 7 8 9 10 Streaming responses with tool calling · Ollama Blog
https://ollama.com/blog/streaming-tool