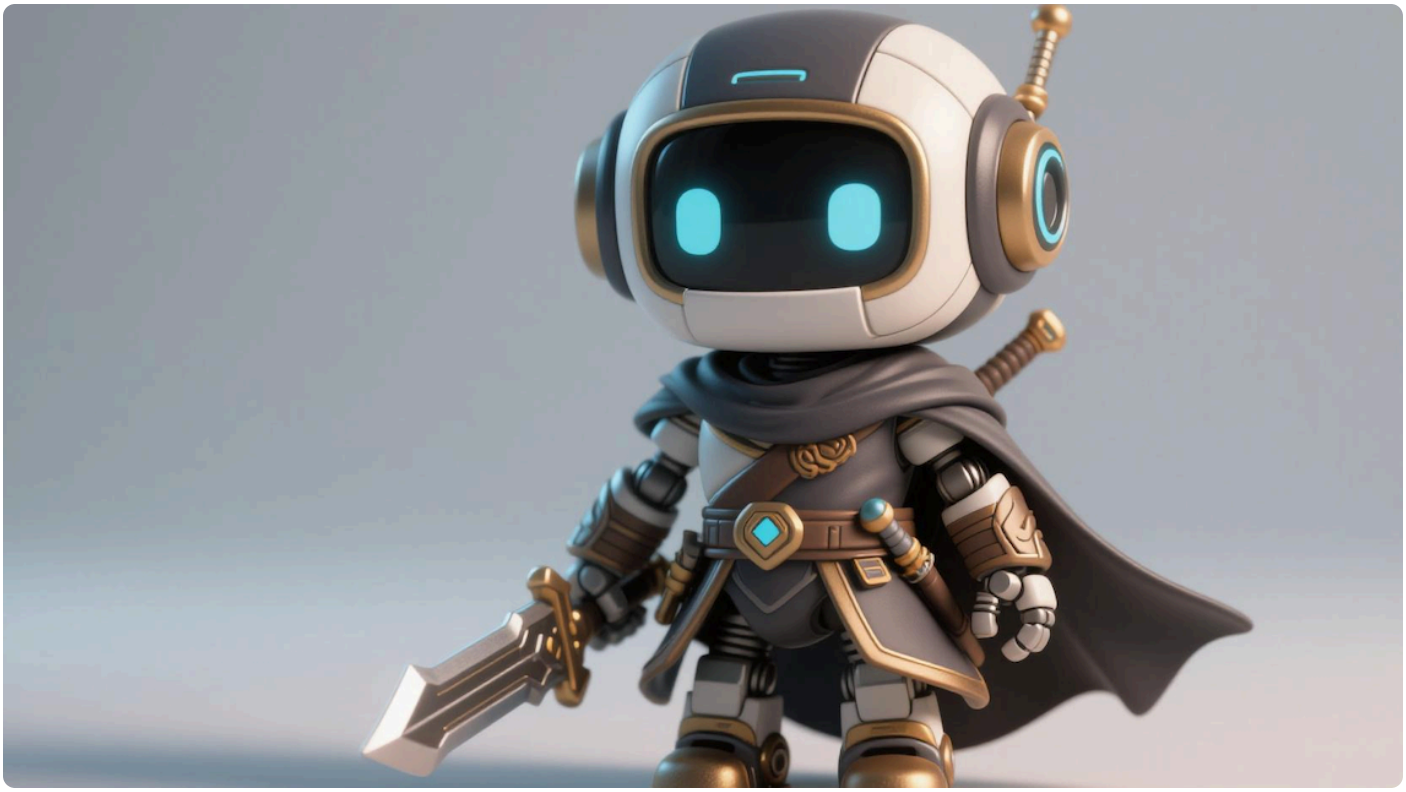
13 | 深度剖析（下）：Ling框架的底层实现

月影 · 跟月影学前端智能体开发



你好，我是月影。给假期还在学习的你点个赞。

上一节课我们介绍了 Ling 框架的底层 Adapter 模块。这一节课我们继续讲解 Ling 框架非常核心的另外两个模块—— Bot 和 Tube。

Bot 子模块

Bot 子模块是 Ling 框架大模型工作流节点的主体，它的完整代码如下：

复制代码

```
import EventEmitter from 'node:events';

import { Tube } from "../tube";
import nunjucks from 'nunjucks';
import { getChatCompletions } from "../adapter/openai";
import { getChatCompletions as getCozeChatCompletions } from "../adapter/coze";

import type { ChatConfig, ChatOptions } from "../types";
```

```ts
import type { ChatCompletionAssistantMessageParam, ChatCompletionSystemMessagePar

type ChatCompletionMessageParam = ChatCompletionSystemMessageParam | ChatCompleti

export enum WorkState {
  INIT = 'init',
  WORKING = 'chatting',
  INFERENCE_DONE = 'inference-done',
  FINISHED = 'finished',
  ERROR = 'error',
}

export abstract class Bot extends EventEmitter {
  abstract get state(): WorkState;
}

export class ChatBot extends Bot {
  private prompts: ChatCompletionSystemMessageParam[] = [];
  private history: ChatCompletionMessageParam[] = [];
  private customParams: Record<string, string> = {};
  private chatState = WorkState.INIT;
  private config: ChatConfig;
  private options: ChatOptions;

  constructor(private tube: Tube, config: ChatConfig, options: ChatOptions = {})
    super();
    this.config = { ...config };
    this.options = { ...options };
  }

  isJSONFormat() {
    return this.options.response_format?.type === 'json_object';
  }

  get root() {
    return this.options.response_format?.root;
  }

  setJSONRoot(root: string | null) {
    if(!this.options.response_format) {
      this.options.response_format = { type: 'json_object', root };
    } else {
      this.options.response_format.root = root;
    }
  }

  setCustomParams(params: Record<string, string>) {
    this.customParams = {...params};
  }
```

```
58
59    addPrompt(promptTpl: string, promptData: Record<string, any> = {}) {
60      const promptText = nunjucks.renderString(promptTpl, { chatConfig: this.config
61      this.prompts.push({ role: "system", content: promptText });
62    }
63
64    setPrompt(promptTpl: string, promptData: Record<string, string> = {}) {
65      this.prompts = [];
66      this.addPrompt(promptTpl, promptData);
67    }
68
69    addHistory(messages: ChatCompletionMessageParam []) {
70      this.history.push(...messages);
71    }
72
73    setHistory(messages: ChatCompletionMessageParam []) {
74      this.history = messages;
75    }
76
77    addFilter(filter: ((data: unknown) => boolean) | string | RegExp) {
78      this.tube.addFilter(filter);
79    }
80
81    clearFilters() {
82      this.tube.clearFilters();
83    }
84
85    userMessage(message: string): ChatCompletionUserMessageParam {
86      return { role: "user", content: message };
87    }
88
89    botMessage(message: string): ChatCompletionAssistantMessageParam {
90      return { role: "assistant", content: message };
91    }
92
93    async chat(message: string | ChatCompletionContentPart[]) {
94      try {
95        this.chatState = WorkState.WORKING;
96        const isJSONFormat = this.isJSONFormat();
97        const prompts = this.prompts.length > 0 ? [...this.prompts] : [];
98        if(this.prompts.length === 0 && isJSONFormat) {
99          prompts.push({
100           role: 'system',
101           content: `[Output]\nOutput with json format, starts with '{'\n[Example]
102         });
103       }
104       const messages = [...prompts, ...this.history, { role: "user", content: mes
105       if(this.config.model_name.startsWith('coze:')) {
106         return await getCozeChatCompletions(this.tube, messages, this.config, {..
```

```
          (content) => { // on complete
            this.chatState = WorkState.FINISHED;
            this.emit('response', content);
          }, (content) => { // on string response
            this.emit('string-response', content);
          }, (content) => { // on object response
            this.emit('object-response', content);
          }).then((content) => { // on inference done
            this.chatState = WorkState.INFERENCE_DONE;
            this.emit('inference-done', content);
          });
      }
      return await getChatCompletions(this.tube, messages, this.config, this.opti
        (content) => { // on complete
          this.chatState = WorkState.FINISHED;
          this.emit('response', content);
        }, (content) => { // on string response
          this.emit('string-response', content);
        }, (content) => { // on object response
          this.emit('object-response', content);
        }).then((content) => { // on inference done
          this.chatState = WorkState.INFERENCE_DONE;
          this.emit('inference-done', content);
        });
    } catch(ex: any) {
      console.error(ex);
      this.chatState = WorkState.ERROR;
      // 不主动发error给客户端
      // this.tube.enqueue({event: 'error', data: ex.message});
      this.emit('error', ex.message);
      // this.tube.cancel();
    }
  }

  finish() {
    this.emit('inference-done', 'null');
    this.emit('response', 'null');
    this.chatState = WorkState.FINISHED;
  }

  get state() {
    return this.chatState;
  }
`
```

首先我们通过枚举对象定义 Bot 节点的工作状态，Ling 管理模块依赖这些状态对 Bot 节点进行统一管理。

复制代码

```
export enum WorkState {
  INIT = 'init',
  WORKING = 'chatting',
  INFERENCE_DONE = 'inference-done',
  FINISHED = 'finished',
  ERROR = 'error',
}
```

默认创建 Bot 对象时，它处于 INIT 状态。当 Bot 开始异步工作时，它处于 WORKING 状态。如果 Bot 是进行大模型推理，那么当推理完成而数据流式传输还未结束时，它处于 INFERENCE_DONE 状态。最后当所有工作都完成时，它处于 FINISHED 状态。在工作过程中，出现任何错误，它将处于 ERROR 状态。

接着我们定义一个抽象类 Bot：

复制代码

```
export abstract class Bot extends EventEmitter {
  abstract get state(): WorkState;
}
```

这个抽象类只有一个抽象属性 state，Ling 对象通过读取它来确认 Bot 当前状态。这个类允许我们自定义 Bot，在扩展工作流节点类型时非常有帮助，我们后续的课程中会通过实操案例来详细说明。

另外 Bot 继承 EventEmitter，所以 Ling 对象可以随时监听 Bot 执行过程中数据和状态的变化。

接着我们定义 ChatBot 继承自 Bot：

```
export class ChatBot extends Bot {
...
}
```

首先定义一些必要的属性和方法。

isJSONFormat()：判断当前 Bot 输出是否是 JSON 格式。

root：获取当前 Bot 的父路径，如果设置了这个值，那么当前 Bot 如果输出 JSON 数据，这些数据的 jsonuri 会叠加上 root 指定的前缀，例如 root 设置为 a，当前输出的 jsonuri 是 b/c，那么最终输出的 jsonuri 就是 a/b/c；如果当前 Bot 输出普通的文本数据，那么该文本数据的 url 就是 root 制定的 uri。uri 为 b/c，那么最后输出时完整的 uri 会是 a/b/c。

setCustomParams(params: Record<string, string>)：自定义公共参数，这些参数会传给 Bot 设置的提示词，在后续添加提示词时，会使用 nunjucks 模版完成提示词的解析。

addPrompt(promptTpl: string, promptData: Record<string, any> = {})：添加一个提示词，可调用多次以添加多个，注意第二个参数传一个对象设置模板变量，Bot 会将它和公共参数合并后传给模板引擎解析，如果对象中的 key 和公共参数相同，公共参数中的对应 key 属性的参数会被覆盖。

setPrompt(promptTpl: string, promptData: Record<string, string> = {})：和 addPrompt 类似，不同的是 addPrompt 是追加参数，而 setPrompt 会删除之前已经添加的提示词，设置新的提示词。

addHistory(messages: ChatCompletionMessageParam [])：追加历史聊天记录。

setHistory(messages: ChatCompletionMessageParam [])：和 addHistory 类似，只不过它会清除之前已经追加的记录，设置新的聊天记录。

addFilter(filter: ((data: unknown) => boolean) | string | RegExp)：增加过滤器，Bot 将它传给 Tube，这样满足过滤条件的数据就不会被 Tube 发给前端。

clearFilters()：清除所有已添加的过滤器。

除此以外，最核心的就是 chat 方法。

复制代码

```
async chat(message: string) {
  try {
    this.chatState = WorkState.WORKING;
    const isJSONFormat = this.isJSONFormat();
    const prompts = this.prompts.length > 0 ? [...this.prompts] : [];
    if(this.prompts.length === 0 && isJSONFormat) {
      prompts.push({
        role: 'system',
        content: `[Output]\nOutput with json format, starts with '{'\n[Example]
      });
    }
    const messages = [...prompts, ...this.history, { role: "user", content: mes
    if(this.config.model_name.startsWith('coze:')) {
      return await getCozeChatCompletions(this.tube, messages, this.config, {..
        (content) => { // on complete
          this.chatState = WorkState.FINISHED;
          this.emit('response', content);
        }, (content) => { // on string response
          this.emit('string-response', content);
        }, (content) => { // on object response
          this.emit('object-response', content);
        }).then((content) => { // on inference done
          this.chatState = WorkState.INFERENCE_DONE;
          this.emit('inference-done', content);
        });
    }
    return await getChatCompletions(this.tube, messages, this.config, this.opti
      (content) => { // on complete
        this.chatState = WorkState.FINISHED;
        this.emit('response', content);
      }, (content) => { // on string response
        this.emit('string-response', content);
      }, (content) => { // on object response
        this.emit('object-response', content);
      }).then((content) => { // on inference done
        this.chatState = WorkState.INFERENCE_DONE;
        this.emit('inference-done', content);
      });
  } catch(ex: any) {
    console.error(ex);
    this.chatState = WorkState.ERROR;
    // 不主动发error给客户端
    // this.tube.enqueue({event: 'error', data: ex.message});
    this.emit('error', ex.message);
```

```
45        // this.tube.cancel();
46      }
47    }
```

上面的代码并不复杂，主要就是针对配置的 model_name 判断当前是 Open AI 兼容的大模型还是 coze，从而调用 Adapter 中不同的方法；另外还有转发 parser 发送的事件，方便 Ling 管理工具后续的处理，有兴趣的同学可以自行认真阅读一下，以掌握更多细节。

setCustomParams(params: Record<string, string>)：设置一个对象作为默认的自定义参数，当这个对象被设置后，**每一次**执行 addPrompt 时，模板变量对象会和这个对象合并后作为最终的模板变量传入提示词模板进行编译。

addPrompt(promptTpl: string, promptData: Record<string, any> = {})：可**重复**调用，**每次**添加一个提示词模板，然后用 nunjucks 编译，传入的对象属性作为编译的模板变量，如果用户设置了默认自定义参数对象，那么这两个对象会合并作为模板变量对象，传入的对象中与自定义参数对象相同属性的值会覆盖自定义对象上的属性值。

setPrompt(promptTpl: string, promptData: Record<string, string> = {})：和 addPrompt 类似，但是它执行前会将之前创建的旧提示词全部清除。

addHistory(messages: ChatCompletionMessageParam [])：追加多条历史聊天记录，历史聊天记录是形如 `[{role: 'system', content: '...'}...]` 的数组。

setHistory(messages: ChatCompletionMessageParam [])：设置多条历史聊天记录，与 addHistory 不同的是，它执行前会将之前追加过的所有聊天记录清空。

addFilter(filter: ((data: unknown) => boolean) | string | RegExp)：添加过滤函数，这个函数会被传给 Tube 对象，在 Tube 中会通过该函数过滤 jsonuri 对象，符合过滤条件的 uri，将不会被发送给客户端。

clearFilters()：清空所有添加过的过滤函数。

finish()：强制结束 Bot 的工作，将状态置为 FINISHED。

接着是最核心的 chat 方法：

```
async chat(message: string | ChatCompletionContentPart[]) {
  try {
    this.chatState = WorkState.WORKING;
    const isJSONFormat = this.isJSONFormat();
    const prompts = this.prompts.length > 0 ? [...this.prompts] : [];
    if(this.prompts.length === 0 && isJSONFormat) {
      prompts.push({
        role: 'system',
        content: `[Output]\nOutput with json format, starts with '{'\n[Example]
      });
    }
    const messages = [...prompts, ...this.history, { role: "user", content: mes
    if(this.config.model_name.startsWith('coze:')) {
      return await getCozeChatCompletions(this.tube, messages, this.config, {..
        (content) => { // on complete
          this.chatState = WorkState.FINISHED;
          this.emit('response', content);
        }, (content) => { // on string response
          this.emit('string-response', content);
        }, (content) => { // on object response
          this.emit('object-response', content);
        }).then((content) => { // on inference done
          this.chatState = WorkState.INFERENCE_DONE;
          this.emit('inference-done', content);
        });
    }
    return await getChatCompletions(this.tube, messages, this.config, this.opti
      (content) => { // on complete
        this.chatState = WorkState.FINISHED;
        this.emit('response', content);
      }, (content) => { // on string response
        this.emit('string-response', content);
      }, (content) => { // on object response
        this.emit('object-response', content);
      }).then((content) => { // on inference done
        this.chatState = WorkState.INFERENCE_DONE;
        this.emit('inference-done', content);
      });
  } catch(ex: any) {
    console.error(ex);
    this.chatState = WorkState.ERROR;
    this.emit('error', ex.message);
  }
}
```

这个方法最核心的部分就是根据 model_name，选择调用 Coze 或者 OpenAI 的对应方法，然后通过回调函数，将对应的 reponse、string-response、object-response、inference 事件发送给 Ling 管理器。

因为 Bot 将 Tube 对象传给了 Adapter 模块，所以 Adapter 更新内容的时候，Tube 对象可以自己处理 data 事件，不需要 Bot 的具体参与。

接下来我们就了解一下 Tube 模块究竟是如何处理的。

Tube 子模块

Tube 是我们要了解的第三个子模块，它的创建由 Ling 统一负责，这样确保 LIng 托管下的所有 Bot 都采用同一个 Tube 发送数据，这样我们在前端就可以从一个流里面完整拿到所有数据了。

以下是 Tube 的完整代码：

复制代码

```
import EventEmitter from 'node:events';
import { shortId } from "../utils";

export class Tube extends EventEmitter {
  private _stream: ReadableStream;
  private controller: ReadableStreamDefaultController | null = null;
  private _canceled: boolean = false;
  private _closed: boolean = false;
  private _sse: boolean = false;
  private messageIndex = 0;
  private filters: ((data: unknown) => boolean)[] = [];

  constructor(private session_id: string = shortId()) {
    super();
    const self = this;
    this._stream = new ReadableStream({
      start(controller) {
        self.controller = controller;
      }
    });
  }

```

```
  addFilter(filter: ((data: unknown) => boolean) | string | RegExp) {
    if(typeof filter === 'string') {
      this.filters.push((data: any) => data.uri === filter);
    } else if(filter instanceof RegExp) {
      this.filters.push((data: any) => filter.test(data.uri));
    } else {
      this.filters.push(filter);
    }
  }

  clearFilters() {
    this.filters = [];
  }

  setSSE(sse: boolean) {
    this._sse = sse;
  }

  enqueue(data: unknown, isQuiet: boolean = false) {
    const isFiltered = this.filters.some(filter => filter(data));
    const id = `${this.session_id}:${this.messageIndex++}`;
    if (!this._closed) {
      try {
        if(typeof data !== 'string') {
          if(this._sse && (data as any)?.event) {
            const event = `event: ${(data as any).event}\n`
            if(!isQuiet && !isFiltered) this.controller?.enqueue(event);
            this.emit('message', {id, data: event});
            if((data as any).event === 'error') {
              this.emit('error', {id, data});
            }
          }
          data = JSON.stringify(data) + '\n'; // use jsonl (json lines)
        }
        if(this._sse) {
          data = `data: ${(data as string).replace(/\n$/,'')}\nid: ${id}\n\n`;
        }
        if(!isQuiet && !isFiltered) this.controller?.enqueue(data);
        this.emit('message', {id, data});
      } catch(ex: any) {
        this._closed = true;
        this.emit('error', {id, data: ex.message});
        console.error('enqueue error:', ex);
      }
    }
  }

  close() {
    if(this._closed) return;
```

```
    this.enqueue({event: 'finished'});
    this.emit('finished');
    this._closed = true;
    if(!this._sse) this.controller?.close();
  }

  async cancel() {
    if(this._canceled) return;
    this._canceled = true;
    this._closed = true;
    try {
      this.enqueue({event: 'canceled'});
      this.emit('canceled');
      await this.stream.cancel();
    } catch(ex) {}
  }

  get canceled() {
    return this._canceled;
  }

  get closed() {
    return this._closed;
  }

  get stream() {
    return this._stream;
  }
}
```

Tube 在自己的构造器中创建了一个 ReadableStream 对象：

复制代码

```
  constructor(private session_id: string = shortId()) {
    super();
    const self = this;
    this._stream = new ReadableStream({
      start(controller) {
        self.controller = controller;
      }
    });
  }
```

有一点需要我们特别注意的是，我们在创建流的时候，从 ReadableStream 对象中拿到 controller 对象，这个对象是一个流控制器，它可以用于向流中加入数据或关闭流。

Tube 最核心的就是 **enqueque 方法**。

复制代码

```
enqueue(data: unknown, isQuiet: boolean = false) {
  const isFiltered = this.filters.some(filter => filter(data));
  const id = `${this.session_id}:${this.messageIndex++}`;
  if (!this._closed) {
    try {
      // 如果 data 不是字符串
      if(typeof data !== 'string') {
        // 如果启用了 SSE 且 data 有 event 字段
        if(this._sse && (data as any)?.event) {
          const event = `event: ${(data as any).event}\n`;
          if(!isQuiet && !isFiltered) this.controller?.enqueue(event);
          this.emit('message', {id, data: event});
          if((data as any).event === 'error') {
            this.emit('error', {id, data});
          }
        }
        data = JSON.stringify(data) + '\n'; // 每条用换行符分隔(JSON Lines格式)
      }
      if(this._sse) {
        data = `data: ${(data as string).replace(/\n$/,'')}\nid: ${id}\n\n`;
      }
      if(!isQuiet && !isFiltered) this.controller?.enqueue(data);
      this.emit('message', {id, data});
    } catch(ex: any) {
      this._closed = true;
      this.emit('error', {id, data: ex.message});
      console.error('enqueue error:', ex);
    }
  }
}
```

这个方法非常重要，主要做了以下几件事：

- **检查过滤**：`isFiltered = this.filters.some(filter => filter(data))` 表示只要有一个 filter 函数返回 true，就认为该 data 需要过滤。被过滤的 data 将不会通过流

发送给客户端。

生成消息 ID：`id = session_id + ":" + messageIndex`。`messageIndex++` 用来递增消息编号。

SSE：如果 `_sse === true` 并且 data 里有 event 字段，那么会先输出一行 `event: ...`。然后再把原本的 data 转成 JSON，并加上 `data: ...\nid: ...` 这种 SSE 格式，然后再添加到流里。这样浏览器或 SSE 客户端会把这条数据流解析成相应的 SSE 事件流。

过滤和静默处理：如果 `isQuiet === true` 或 `isFiltered === true`，则不会真正 enqueue 到流中，即不会把内容推送给客户端；但还是会触发内部的事件（比如 `this.emit('message', ...)`）。

发出事件：`this.emit('message', {id, data})`，在事件系统中（EventEmitter）抛出一条消息事件，从而在 Ling 管理器中可以监听 message。

最后是 close 和 cancel 方法，前者关闭流，后者强制取消流。两者逻辑类似，但是关闭流将会发送 finished 事件，而取消流将会发送 canceled 事件给客户端，以便于在客户端进行相应的处理。

这样我们就完成了 Tube 子模块。

Ling 管理模块

接着我们看最后的部分，看看这些子模块是怎么被 Ling 管理起来的。

Ling 的管理模块完整代码如下：

复制代码

```
import EventEmitter from 'node:events';
import merge from 'lodash.merge';

import { ChatBot, Bot, WorkState } from './bot/index';
import { Tube } from './tube';
import type { ChatConfig, ChatOptions } from "./types";
import { sleep, shortId } from './utils';

```

```
export type { ChatConfig, ChatOptions } from "./types";
export type { Tube } from "./tube";

export { Bot, ChatBot, WorkState } from "./bot";

export class Ling extends EventEmitter {
  protected _tube: Tube;
  protected customParams: Record<string, string> = {};
  protected bots: Bot[] = [];
  protected session_id = shortId();
  private _promise: Promise<any> | null = null;
  private _tasks: Promise<any>[] = [];
  constructor(protected config: ChatConfig, protected options: ChatOptions = {})
    super();
    if(config.session_id) {
      this.session_id = config.session_id;
      delete config.session_id;
    }
    this._tube = new Tube(this.session_id);
    if(config.sse) {
      this._tube.setSSE(true);
    }
    this._tube.on('message', (message) => {
      this.emit('message', message);
    });
    this._tube.on('finished', () => {
      this.emit('finished');
    });
    this._tube.on('canceled', () => {
      this.emit('canceled');
    });
  }

  handleTask(task: () => Promise<any>) {
    return new Promise((resolve, reject) => {
      this._tasks.push(task().then(resolve).catch(reject));
    });
  }

  get promise() {
    if(!this._promise) {
      this._promise = new Promise((resolve, reject) => {
        let result: any = {};
        this.on('inference-done', (content, bot) => {
          let output = bot.isJSONFormat() ? JSON.parse(content) : content;
          if(bot.root != null) {
            result[bot.root] = output;
          } else {
            result = merge(result, output);
```

```
          }
          setTimeout(async () => {
            // 没有新的bot且其他bot的状态都都推理结束
            if(this.bots.every(
              (_bot: Bot) => _bot.state === WorkState.INFERENCE_DONE
              || _bot.state === WorkState.FINISHED
              || _bot.state === WorkState.ERROR || bot === _bot
            )) {
              await Promise.all(this._tasks);
              resolve(result);
            }
          });
        });
        // this.once('finished', () => {
        //   resolve(result);
        // });
        this.once('error', (error, bot) => {
          reject(error);
        });
      });
    }
    return this._promise;
  }

  createBot(root: string | null = null, config: Partial<ChatConfig> = {}, options
    const bot = new ChatBot(this._tube, {...this.config, ...config}, {...this.opt
    bot.setJSONRoot(root);
    bot.setCustomParams(this.customParams);
    bot.addListener('error', (error) => {
      this.emit('error', error, bot);
    });
    bot.addListener('inference-done', (content) => {
      this.emit('inference-done', content, bot);
    });
    this.bots.push(bot);
    return bot;
  }

  addBot(bot: Bot) {
    this.bots.push(bot);
  }

  setCustomParams(params: Record<string, string>) {
    this.customParams = {...params};
  }

  setSSE(sse: boolean) {
    this._tube.setSSE(sse);
  }
```

```
107
108   protected isAllBotsFinished() {
109     return this.bots.every(bot => bot.state === 'finished' || bot.state === 'erro
110   }
111
112   async close() {
113     while (!this.isAllBotsFinished()) {
114       await sleep(100);
115     }
116     await sleep(500); // 再等0.5秒，确保没有新的 bot 创建，所有 bot 都真正结束
117     if(!this.isAllBotsFinished()) {
118       this.close(); // 如果还有 bot 没有结束，则再关闭一次
119       return;
120     }
121     await Promise.all(this._tasks); // 看还有没有任务没有完成
122     this._tube.close();
123     this.bots = [];
124     this._tasks = [];
125   }
126
127   async cancel() {
128     this._tube.cancel();
129     this.bots = [];
130     this._tasks = [];
131   }
132
133   sendEvent(event: any) {
134     this._tube.enqueue(event);
135   }
136
137   get tube() {
138     return this._tube;
139   }
140
141   get model() {
142     return this.config.model_name;
143   }
144
145   get stream() {
146     return this._tube.stream;
147   }
148
149   get canceled() {
150     return this._tube.canceled;
151   }
152
153   get closed() {
154     return this._tube.closed;
155   }
```

```
156
157   get id() {
158     return this.session_id;
159   }
160 }
```

首先，我们看一下构造器：

复制代码

```
1   constructor(protected config: ChatConfig, protected options: ChatOptions = {})
2     super();
3     if(config.session_id) {
4       this.session_id = config.session_id;
5       delete config.session_id;
6     }
7     this._tube = new Tube(this.session_id);
8     if(config.sse) {
9       this._tube.setSSE(true);
10    }
11    this._tube.on('message', (message) => {
12      this.emit('message', message);
13    });
14    this._tube.on('finished', () => {
15      this.emit('finished');
16    });
17    this._tube.on('canceled', () => {
18      this.emit('canceled');
19    });
20  }
```

在构造器里，我们创建了一个 Tube 对象，保存在 _tube 私有属性中，然后我们监听这个对象的 message、finished 和 canceled 事件，将它们转发。

接着我们看 createBot：

复制代码

```
1 createBot(root: string | null = null, config: Partial<ChatConfig> = {}, options:
2   const bot = new ChatBot(this._tube, {...this.config, ...config}, {...this.optic
3   bot.setJSONRoot(root);
4   bot.setCustomParams(this.customParams);
```

```
  bot.addListener('error', (error) => {
    this.emit('error', error, bot);
  });
  bot.addListener('inference-done', (content) => {
    this.emit('inference-done', content, bot);
  });
  this.bots.push(bot);
  return bot;
}
```

Ling 管理对象通过调用这个方法创建并托管 Bot 对象，它自动将 _tube 属性下的 Tube 对象传入 Bot 构造器，并且监听 bot 的 `error` 和 `inference-done` 事件，将它们转发。

再看 close 方法：

复制代码

```
async close() {
  while (!this.isAllBotsFinished()) {
    await sleep(100);
  }
  await sleep(500); // 再等0.5秒，确保没有新的 bot 创建，所有 bot 都真正结束
  if(!this.isAllBotsFinished()) {
    this.close(); // 如果还有 bot 没有结束，则再关闭一次
    return;
  }
  await Promise.all(this._tasks); // 看还有没有任务没有完成
  this._tube.close();
  this.bots = [];
  this._tasks = [];
}
```

这是一个需要特别注意的方法，它并不是指立即结束所有的工作，而是会异步轮询所有托管下的 Bot，判断它们的状态是否是 FINISHED 或 ERROR，只有当它们全部结束之后，才会真正将 _tube 关闭。

由于 Bot 是被异步创建的，因此我们不确定什么时候不再创建新的 Bot。所以我们建立了一个约定，那就是当所有已创建并托管的 Bot 工作都结束后，500 毫秒内未创建新的 Bot，则判断为工作全部结束。

有了这个规则，我们就可以很方便地提前调用这个方法，并等待全部工作完成，不过更稳妥一些的方式还是应该要尽可能在确认工作已经全部完成之后，再去调用这个方法。在我们后续实战项目中，我们会通过实际代码来理解。

Ling 管理模块其他的方法比较简单，我就挑两个细节讲一讲，其他的大家自己有兴趣可以深入研究一下。

复制代码

```
handleTask(task: () => Promise<any>) {
  return new Promise((resolve, reject) => {
    this._tasks.push(task().then(resolve).catch(reject));
  });
}
```

除了默认托管的 Bot 之外，我们可以给 Ling 管理器添加外部的异步操作，这样会产生两个影响，一是 close 的时候，Ling 会判断 _tasks 是否完成，然后再关闭 Tube 对象。二是，Ling 暴露一个外部属性 promise，它是一个 getter：

复制代码

```
get promise() {
  if(!this._promise) {
    this._promise = new Promise((resolve, reject) => {
      let result: any = {};
      this.on('inference-done', (content, bot) => {
        let output = bot.isJSONFormat() ? JSON.parse(content) : content;
        if(bot.root != null) {
          result[bot.root] = output;
        } else {
          result = merge(result, output);
        }
        setTimeout(async () => {
          // 没有新的bot且其他bot的状态都都推理结束
          if(this.bots.every(
            (_bot: Bot) => _bot.state === WorkState.INFERENCE_DONE
            || _bot.state === WorkState.FINISHED
            || _bot.state === WorkState.ERROR || bot === _bot
          )) {
            await Promise.all(this._tasks);
            resolve(result);
          }
```

```
        });
      });
      // this.once('finished', () => {
      //   resolve(result);
      // });
      this.once('error', (error, bot) => {
        reject(error);
      });
    });
  }
  return this._promise;
}
```

在业务使用的时候，如果需要等待 Ling 的推理结束，然后执行其他的操作，可以去 await 这个对象，并拿到最终的数据结果，例如：

复制代码

```
const result = await ling.promise;
```

这么说可能比较抽象，没关系，我们后续实战项目会经常用到 Ling 框架，到时候我们遇到问题再详细解释，大家就能明白了。

要点总结

好了，那我们这一节课的内容就到这里。

在这一节课里，我们详细了解了 Ling 框架的 Bot、Tube 子模块的核心功能，以及如何通过 Ling 管理模块将它们联系起来。

这两节课主要是讲理论和剖析代码偏多，下一节课，我们就要通过具体实战来进一步深入学习如何用好 Ling 框架了，敬请期待。

课后练习

Bot 模块设计了一个抽象类，这个类可以用来扩展其他类型的 Bot。大家想一想，如果我要让 Ling 管理一个绘图的 Bot，应该怎么自己扩展呢？你可以尝试写一个 ImageBot extends Bot，然后将它也通过 Ling 管理起来吗？可以把你的实现分享到评论区。

AI智能总结

1. ChatBot类是Bot子模块的核心，包含关键方法和属性，如isJSONFormat、setCustomParams、addPrompt等，用于设置参数、添加提示词和记录历史聊天记录。

2. ChatBot类中的chat方法是最核心的，根据配置的model_name判断调用Open AI兼容的大模型还是coze，并处理用户消息，发送相应的事件。

3. Bot子模块通过枚举对象定义了Bot节点的工作状态，包括INIT、WORKING、INFERENCE_DONE、FINISHED和ERROR，允许Ling对象监听Bot执行过程中数据和状态的变化。

4. Ling管理模块中的构造器创建了一个Tube对象，并监听其message、finished和canceled事件，将它们转发。

5. Ling管理模块的createBot方法用于创建并托管Bot对象，自动传入Tube对象，并监听bot的error和inference-done事件。

6. Ling管理模块的close方法异步轮询托管的Bot，判断它们的状态是否是FINISHED或ERROR，只有当它们全部结束之后，才会真正将_tube关闭。

7. Ling管理模块的handleTask方法允许给Ling管理器添加外部的异步操作，影响close方法和暴露一个外部属性promise。

8. Ling管理模块的promise属性用于等待Ling的推理结束，然后执行其他的操作，例如：const result = await ling.promise;

9. Ling框架的Bot模块设计了一个抽象类，可以用来扩展其他类型的Bot，例如ImageBot，通过Ling管理起来。

10. Ling框架的下一节课将通过具体实战来进一步深入学习如何用好Ling框架。

精选留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。