插件开发

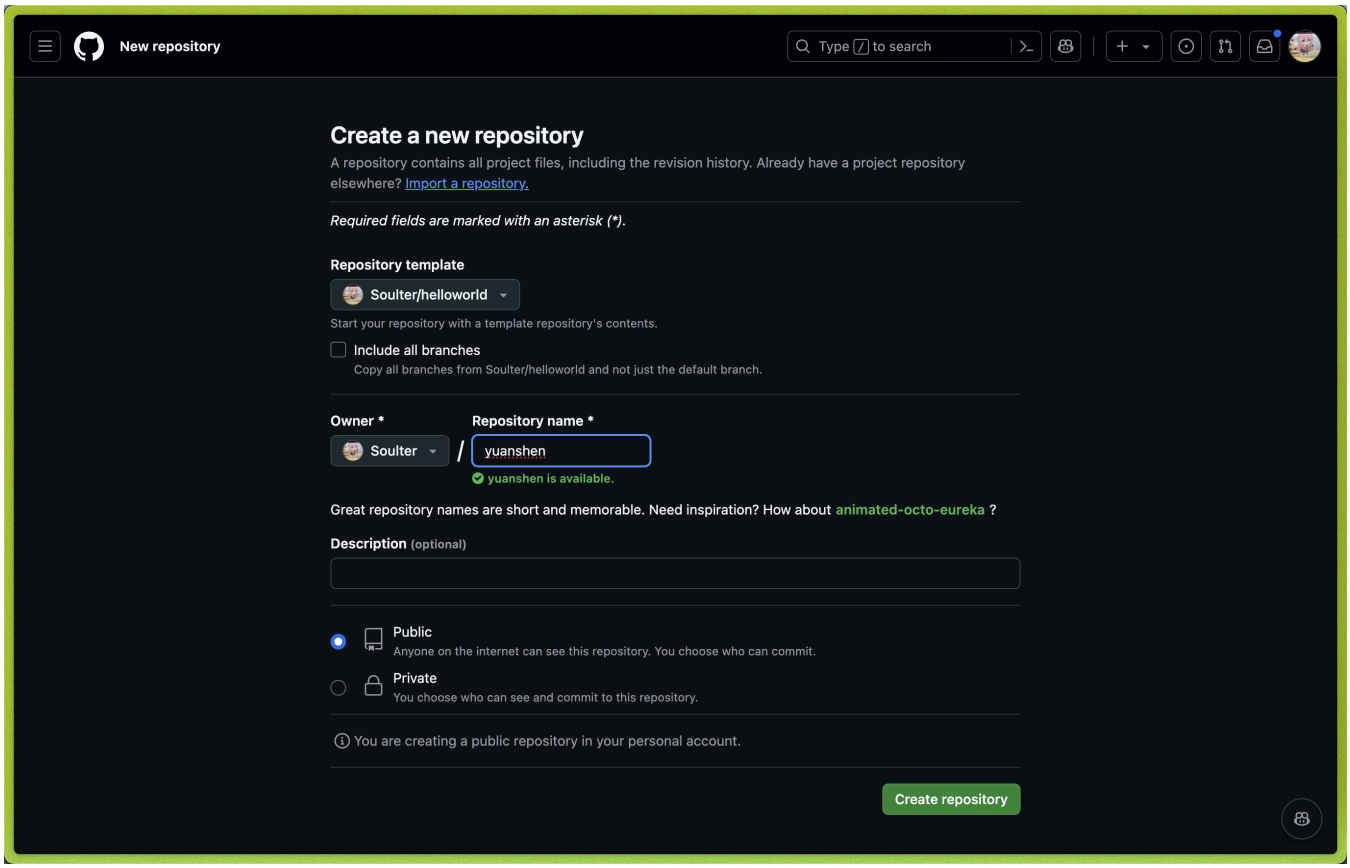
几行代码开发一个插件!

> **TIP**
>
> - 推荐使用 VSCode 开发。
> - 需要有一定的 Python 基础。
> - 需要有一定的 Git 使用经验。
>
> 欢迎加群 `322154837` 讨论！！

开发环境准备

获取插件模板

打开 helloworld。点击右上角的 `Use this template` ，然后点击 `Create new repository` 。在 `Repository name` 处输入你的插件名字，不要中文。建议以 `astrbot_plugin_` 开头，如 `astrbot_plugin_genshin` 。



然后点击右下角的 `Create repository` 。

Clone 插件和 AstrBot 项目

首先 Clone AstrBot 项目本体到本地。

```bash
git clone https://github.com/Soulter/AstrBot
mkdir -p AstrBot/data/plugins
cd AstrBot/data/plugins
git clone <你的插件仓库地址>
```

然后，使用 VSCode 打开 AstrBot 项目。找到 `data/plugins/<你的插件名字>` 目录。

开发环境准备完毕!

提要

最小实例

打开 `main.py` ，这是一个最小的插件实例。

python

```python
from astrbot.api.event import filter, AstrMessageEvent, MessageEventResult
from astrbot.api.star import Context, Star, register

@register("helloworld", "Your Name", "一个简单的 Hello World 插件", "1.0.0", "repo ur
class MyPlugin(Star):
    def __init__(self, context: Context):
        super().__init__(context)

    # 注册指令的装饰器。指令名为 helloworld。注册成功后，发送 `/helloworld` 就会触发这个指
    @filter.command("helloworld")
    async def helloworld(self, event: AstrMessageEvent):
        '''这是一个 hello world 指令''' # 这是 handler 的描述，将会被解析方便用户了解插件
        user_name = event.get_sender_name()
        message_str = event.message_str # 获取消息的纯文本内容
        yield event.plain_result(f"Hello, {user_name}!") # 发送一条纯文本消息

    async def terminate(self):
        '''可选择实现 terminate 函数，当插件被卸载/停用时会调用。'''
```

一个插件就是一个类，这个类继承自 `Star` 。 `Star` 是 AstrBot 插件的基类，还额外提供了一些基础的功能。请务必使用 `@register` 装饰器注册插件，否则 AstrBot 无法识别。

在 `__init__` 中会传入 `Context` 对象，这个对象包含了 AstrBot 的大多数组件

具体的处理函数 `Handler` 在插件类中定义，如这里的 `helloworld` 函数。

> **WARNING**
>
> `Handler` 需要在插件类中注册，前两个参数必须为 `self` 和 `event` 。如果文件行数过长，可以将真正的服务函数写在外部，然后在 `Handler` 中调用。
>
> 插件类所在的文件名需要命名为 `main.py` 。

API 文件结构

所有的 API 都在 `astrbot/api` 目录下。

```
api
├── __init__.py
├── all.py # 无脑使用所有的结构
```

```
├── event
│   └── filter # 过滤器，事件钩子
├── message_components.py # 消息段组建类型
├── platform # 平台相关的结构
├── provider # 大语言模型提供商相关的结构
└── star
```

AstrMessageEvent

`AstrMessageEvent` 是 AstrBot 的消息事件对象。你可以通过 `AstrMessageEvent` 来获取消息发送者、消息内容等信息。里面的方法都有足够的注释。

AstrBotMessage

`AstrBotMessage` 是 AstrBot 的消息对象。你可以通过 `AstrBotMessage` 来查看消息适配器下发的消息的具体内容。通过 `event.message_obj` 获取。

py

```py
class AstrBotMessage:
    '''AstrBot 的消息对象'''
    type: MessageType  # 消息类型
    self_id: str  # 机器人的识别id
    session_id: str  # 会话id。取决于 unique_session 的设置。
    message_id: str  # 消息id
    group_id: str = "" # 群组id，如果为私聊，则为空
    sender: MessageMember  # 发送者
    message: List[BaseMessageComponent]  # 消息链。比如 [Plain("Hello"), At(qq=123456
    message_str: str  # 最直观的纯文本消息字符串，将消息链中的 Plain 消息（文本消息）连接i
    raw_message: object
    timestamp: int  # 消息时间戳
```

其中， `raw_message` 是消息平台适配器的**原始消息对象**。

消息链

`消息链` 描述一个消息的结构，是一个有序列表，列表中每一个元素称为 `消息段` 。

引用方式：

```py
import astrbot.api.message_components as Comp


[Comp.Plain(text="Hello"), Comp.At(qq=123456), Comp.Image(file="https://example.com/
```

> qq 是对应消息平台上的用户 ID。

消息链的结构使用了 `nakuru-project` 。它一共有如下种消息类型。常用的已经用注释标注。

```py
ComponentTypes = {
    "plain": Plain, # 文本消息
    "text": Plain, # 文本消息，同上
    "face": Face, # QQ 表情
    "record": Record, # 语音
    "video": Video, # 视频
    "at": At, # At 消息发送者
    "music": Music, # 音乐
    "image": Image, # 图片
    "reply": Reply, # 回复消息
    "forward": Forward, # 转发消息
    "node": Node, # 转发消息中的节点
    "nodes": Nodes, # Node 的列表，用于支持一个转发消息中的多个节点
    "poke": Poke, # 戳一戳
    "xml": Xml,
    "json": Json,
    "cardimage": CardImage,
    "tts": TTS,
    "unknown": Unknown,
    "rps": RPS,
    "dice": Dice,
    "shake": Shake,
    "anonymous": Anonymous,
    "share": Share,
    "contact": Contact,
    "location": Location,
    "redbag": RedBag,
}
```

请善于 debug 来了解消息结构：

python

```python
@event_message_type(EventMessageType.ALL) # 注册一个过滤器，参见下文。
async def on_message(self, event: AstrMessageEvent):
    print(event.message_obj.raw_message) # 平台下发的原始消息在这里
    print(event.message_obj.message) # AstrBot 解析出来的消息链内容
```

开发指南

CAUTION

接下来的代码中处理函数可能会忽略插件类的定义，但请记住，所有的处理函数都需要写在插件类中。

事件监听器

事件监听器可以收到平台下发的消息内容，可以实现指令、指令组、事件监听等功能。

事件监听器的注册器在 `astrbot.api.event.filter` 下，需要先导入。请务必导入，否则会和 python 的高阶函数 filter 冲突。

py

```py
from astrbot.api.event import filter, AstrMessageEvent
```

注册一个指令

python

```python
from astrbot.api.event import filter, AstrMessageEvent
from astrbot.api.star import Context, Star, register

@register("helloworld", "Soulter", "一个简单的 Hello World 插件", "1.0.0")
class MyPlugin(Star):
    def __init__(self, context: Context):
        super().__init__(context)

    @filter.command("helloworld") # from astrbot.api.event.filter import command
    async def helloworld(self, event: AstrMessageEvent):
        '''这是 hello world 指令'''
        user_name = event.get_sender_name()
        message_str = event.message_str # 获取消息的纯文本内容
        yield event.plain_result(f"Hello, {user_name}!")
```

TIP

指令不能带空格。类似的，可以使用下面的指令组功能。或者也使用监听器自己解析消息内容。

注册一个带参数的指令

AstrBot 会自动帮你解析指令的参数。

python

```python
@filter.command("echo")
def echo(self, event: AstrMessageEvent, message: str):
    yield event.plain_result(f"你发了: {message}")

@filter.command("add")
def add(self, event: AstrMessageEvent, a: int, b: int):
    # /add 1 2 -> 结果是: 3
    yield event.plain_result(f"结果是: {a + b}")
```

注册一个指令组

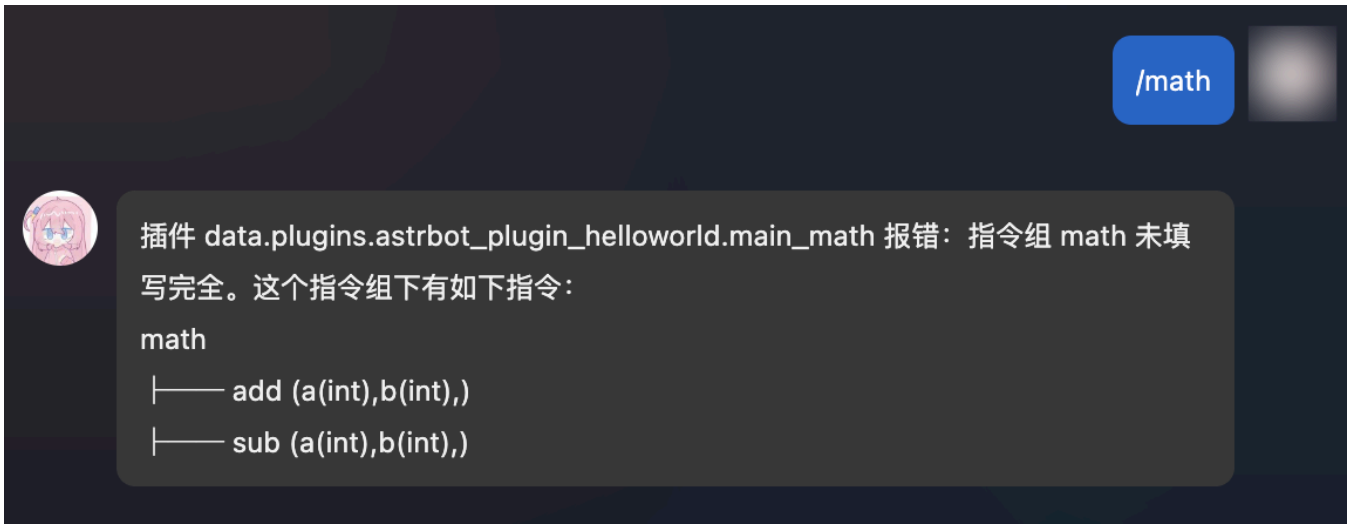
指令组可以帮助你组织指令。

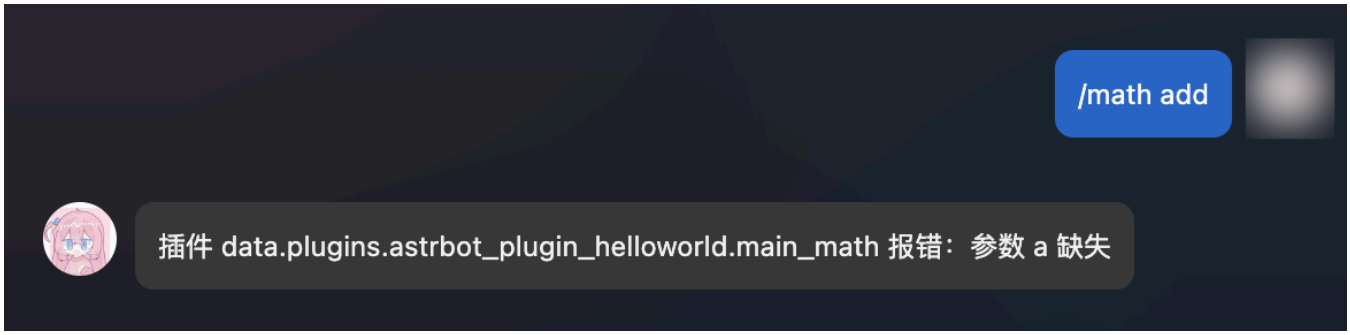
python

```python
@filter.command_group("math")
def math(self):
    pass

@math.command("add")
async def add(self, event: AstrMessageEvent, a: int, b: int):
    # /math add 1 2 -> 结果是: 3
    yield event.plain_result(f"结果是: {a + b}")

@math.command("sub")
async def sub(self, event: AstrMessageEvent, a: int, b: int):
    # /math sub 1 2 -> 结果是: -1
    yield event.plain_result(f"结果是: {a - b}")
```

指令组函数内不需要实现任何函数，请直接 `pass` 或者添加函数内注释。指令组的子指令使用 `指令组名.command` 来注册。

当用户没有输入子指令时，会报错并，并渲染出该指令组的树形结构。





理论上，指令组可以无限嵌套！

py

```
'''
math
├── calc
│   ├── add (a(int),b(int),)
│   ├── sub (a(int),b(int),)
│   ├── help (无参数指令)
'''

@filter.command_group("math")
def math():
    pass

@math.group("calc") # 请注意，这里是 group，而不是 command_group
def calc():
    pass
```

```python
@calc.command("add")
async def add(self, event: AstrMessageEvent, a: int, b: int):
    yield event.plain_result(f"结果是: {a + b}")

@calc.command("sub")
async def sub(self, event: AstrMessageEvent, a: int, b: int):
    yield event.plain_result(f"结果是: {a - b}")

@calc.command("help")
def calc_help(self, event: AstrMessageEvent):
    # /math calc help
    yield event.plain_result("这是一个计算器插件，拥有 add，sub 指令。")
```

指令和指令组的别名(alias)

> v3.4.28 后

可以为指令或指令组添加不同的别名：

python

```python
@filter.command("help", alias=['帮助', 'helpme'])
def help(self, event: AstrMessageEvent):
    yield event.plain_result("这是一个计算器插件，拥有 add，sub 指令。")
```

群/私聊事件监听器

python

```python
@event_message_type(EventMessageType.PRIVATE_MESSAGE)
async def on_private_message(self, event: AstrMessageEvent):
    message_str = event.message_str # 获取消息的纯文本内容
    yield event.plain_result("收到了一条私聊消息。")
```

`EventMessageType` 是一个 `Enum` 类型，包含了所有的事件类型。当前的事件类型有 `PRIVATE_MESSAGE` 和 `GROUP_MESSAGE` 。

接收所有消息事件

这将接收所有的事件。

python

```python
@event_message_type(EventMessageType.ALL)
async def on_all_message(self, event: AstrMessageEvent):
```

```python
    yield event.plain_result("收到了一条消息。")
```

只接收某个消息平台的事件

```python
@platform_adapter_type(PlatformAdapterType.AIOCQHTTP | PlatformAdapterType.QQOFFICIA
async def on_aiocqhttp(self, event: AstrMessageEvent):
    '''只接收 AIOCQHTTP 和 QQOFFICIAL 的消息'''
    yield event.plain_result("收到了一条信息")
```

当前版本下， `PlatformAdapterType` 有 `AIOCQHTTP` , `QQOFFICIAL` , `GEWECHAT` , `ALL` 。

限制管理员才能使用指令

```python
@permission_type(PermissionType.ADMIN)
@filter.command("test")
async def test(self, event: AstrMessageEvent):
    pass
```

仅管理员才能使用 `test` 指令。

多个过滤器

支持同时使用多个过滤器，只需要在函数上添加多个装饰器即可。过滤器使用 `AND` 逻辑。也就是说，只有所有的过滤器都通过了，才会执行函数。

```python
@filter.command("helloworld")
@event_message_type(EventMessageType.PRIVATE_MESSAGE)
async def helloworld(self, event: AstrMessageEvent):
    yield event.plain_result("你好！")
```

事件钩子【New】

> **TIP**
>
> 事件钩子不支持与上面的 @command, @command_group, @event_message_type, @platform_adapter_type, @permission_type 一起使用。

AstrBot 初始化完成时

> v3.4.34 后

python

```
from astrbot.api.event import filter, AstrMessageEvent

@filter.on_astrbot_loaded()
async def on_astrbot_loaded(self):
    print("AstrBot 初始化完成")
```

收到 LLM 请求时

在 AstrBot 默认的执行流程中，在调用 LLM 前，会触发 `on_llm_request` 钩子。

可以获取到 `ProviderRequest` 对象，可以对其进行修改。

ProviderRequest 对象包含了 LLM 请求的所有信息，包括请求的文本、系统提示等。

python

```
from astrbot.api.event import filter, AstrMessageEvent
from astrbot.api.provider import ProviderRequest

@filter.on_llm_request()
async def my_custom_hook_1(self, event: AstrMessageEvent, req: ProviderRequest): # 请
    print(req) # 打印请求的文本
    req.system_prompt += "自定义 system_prompt"
```

> 这里不能使用 yield 来发送消息。如需发送，请直接使用 `event.send()` 方法。

LLM 请求完成时

在 LLM 请求完成后，会触发 `on_llm_response` 钩子。

可以获取到 `ProviderResponse` 对象，可以对其进行修改。

python

```
from astrbot.api.event import filter, AstrMessageEvent
from astrbot.api.provider import LLMResponse

@filter.on_llm_response()
```

```python
async def on_llm_resp(self, event: AstrMessageEvent, resp: LLMResponse): # 请注意有三
    print(resp)
```

> 这里不能使用 yield 来发送消息。如需发送，请直接使用 `event.send()` 方法。

发送消息给消息平台适配器前

在发送消息前，会触发 `on_decorating_result` 钩子。

可以在这里实现一些消息的装饰，比如转语音、转图片、加前缀等等

python

```python
from astrbot.api.event import filter, AstrMessageEvent

@filter.on_decorating_result()
async def on_decorating_result(self, event: AstrMessageEvent):
    result = event.get_result()
    chain = result.chain
    print(chain) # 打印消息链
    chain.append(Plain("!")) # 在消息链的最后添加一个感叹号
```

> 这里不能使用 yield 来发送消息。这个钩子只是用来装饰 event.get_result().chain 的。如需发送，请直接使用 `event.send()` 方法。

发送消息给消息平台适配器后

在发送消息给消息平台后，会触发 `after_message_sent` 钩子。

python

```python
from astrbot.api.event import filter, AstrMessageEvent

@filter.after_message_sent()
async def after_message_sent(self, event: AstrMessageEvent):
    pass
```

> 这里不能使用 yield 来发送消息。如需发送，请直接使用 `event.send()` 方法。

优先级

> 大于等于 v3.4.21 版本才有这个功能，低于这个版本的 AstrBot 会报错。

指令、事件监听器可以设置优先级，先于其他指令、监听器执行。默认优先级都是 0。

python

```python
@filter.command("helloworld", priority=1)
async def helloworld(self, event: AstrMessageEvent):
    yield event.plain_result("Hello!")
```

会话控制 [NEW]

> 大于等于 v3.4.36

为什么需要会话控制？考虑一个 成语接龙 插件，某个/群用户需要和机器人进行多次对话，而不是一次性的指令。这时候就需要会话控制。

txt

```
用户：/成语接龙
机器人：请发送一个成语
用户：一马当先
机器人：先见之明
用户：明察秋毫
...
```

AstrBot 提供了开箱即用的会话控制功能：

导入：

py

```
from import astrbot.api.message_components as Comp
from astrbot.core.utils.session_waiter import (
    session_waiter,
    SessionController,
)
```

handler 内的代码可以如下：

python

```python
from astrbot.api.event import filter, AstrMessageEvent

@filter.command("成语接龙")
async def handle_empty_mention(self, event: AstrMessageEvent):
    """成语接龙具体实现"""
    try:
        yield event.plain_result("请发送一个成语~")
```

```
        # 具体的会话控制器使用方法
        @session_waiter(timeout=60, record_history_chains=False) # 注册一个会话控制器,
        async def empty_mention_waiter(controller: SessionController, event: AstrMes
            idiom = event.message_str # 用户发来的成语，假设是 "一马当先"

            # ...
            message_result = event.make_result()
            message_result.chain = [Comp.Plain("先见之明")] # from import astrbot.api
            await event.send(bot_reply) # 发送回复，不能使用 yield

            controller.keep(timeout=60, reset_timeout=True) # 重置超时时间为 60s，如果

            # controller.stop() # 停止会话控制器，会立即结束。
            # 如果记录了历史消息链，可以通过 controller.get_history_chains() 获取历史消息

        try:
            await empty_mention_waiter(event)
        except TimeoutError as _: # 当超时后，会话控制器会抛出 TimeoutError
            yield event.plain_result("你超时了！")
        except Exception as e:
            yield event.plain_result("发生错误，请联系管理员: " + str(e))
        finally:
            event.stop_event()
    except Exception as e:
        logger.error("handle_empty_mention error: " + str(e))
```

当激活会话控制器后，该发送人之后发送的消息会首先经过上面你定义的 `empty_mention_waiter` 函数处理，直到会话控制器被停止或者超时。

SessionController

用于开发者控制这个会话是否应该结束，并且可以拿到历史消息链。

- keep(): 保持这个会话
  - timeout (float): 必填。会话超时时间。
  - reset_timeout (bool): 设置为 True 时, 代表重置超时时间, timeout 必须 > 0, 如果 <= 0 则立即结束会话。设置为 False 时, 代表继续维持原来的超时时间, 新 timeout = 原来剩余的timeout + timeout (可以 < 0)
- stop(): 结束这个会话
- get_history_chains() -> List[List[Comp.BaseMessageComponent]]: 获取历史消息链

自定义会话 ID 算子

默认情况下，AstrBot 会话控制器会将基于 `sender_id` （发送人的 ID）作为识别不同会话的标识，如果想将一整个群作为一个会话，则需要自定义会话 ID 算子。

py

```
from import astrbot.api.message_components as Comp
from astrbot.core.utils.session_waiter import (
    session_waiter,
    SessionFilter,
    SessionController,
)

# 沿用上面的 handler
# ...
class CustomFilter(SessionFilter):
    def filter(self, event: AstrMessageEvent) -> str:
        return event.get_group_id() if event.get_group_id() else event.unified_msg_o

await empty_mention_waiter(event, session_filter=CustomFilter()) # 这里传入 session_f
# ...
```

这样之后，当群内一个用户发送消息后，会话控制器会将这个群作为一个会话，群内其他用户发送的消息也会被认为是同一个会话。

甚至，可以使用这个特性来让群内组队！

发送消息

上面介绍的都是基于 `yield` 的方式，也就是异步生成器。这样的好处是可以在一个函数中多次发送消息。

python

```
@filter.command("helloworld")
async def helloworld(self, event: AstrMessageEvent):
    yield event.plain_result("Hello!")
    yield event.plain_result("你好！")

    yield event.image_result("path/to/image.jpg") # 发送图片
    yield event.image_result("https://example.com/image.jpg") # 发送 URL 图片，务必以
```

如果是一些定时任务或者不想立即发送消息，可以使用 `event.unified_msg_origin` 得到一个字符串并将其存储，然后在想发送消息的时候使用 `self.context.send_message(unified_msg_origin, chains)` 来发送消息。

python

```python
from astrbot.api.event import MessageChain

@filter.command("helloworld")
async def helloworld(self, event: AstrMessageEvent):
    umo = event.unified_msg_origin
    message_chain = MessageChain().message("Hello!").file_image("path/to/image.jpg")
    await self.context.send_message(event.unified_msg_origin, message_chain)
```

通过这个特性，你可以将 unified_msg_origin 存储起来，然后在需要的时候发送消息。

TIP

关于 unified_msg_origin。 unified_msg_origin 是一个字符串，记录了一个会话的唯一ID，AstrBot能够据此找到属于哪个消息平台的哪个会话。这样就能够实现在 `send_message` 的时候，发送消息到正确的会话。有关 MessageChain，请参见接下来的一节。

发送图文等富媒体消息

AstrBot 支持发送富媒体消息，比如图片、语音、视频等。使用 `MessageChain` 来构建消息。

python

```python
from astrbot.api.message_components import *

@filter.command("helloworld")
async def helloworld(self, event: AstrMessageEvent):
    chain = [
        At(qq=event.get_sender_id()), # At 消息发送者
        Plain("来看这个图："),
        Image.fromURL("https://example.com/image.jpg"), # 从 URL 发送图片
        Image.fromFileSystem("path/to/image.jpg"), # 从本地文件目录发送图片
        Plain("这是一个图片。")
    ]
    yield event.chain_result(chain)
```

上面构建了一个 `message chain` ，也就是消息链，最终会发送一条包含了图片和文字的消息，并且保留顺序。

你也可以快捷发送图文而不用显式构建 `message chain` 。

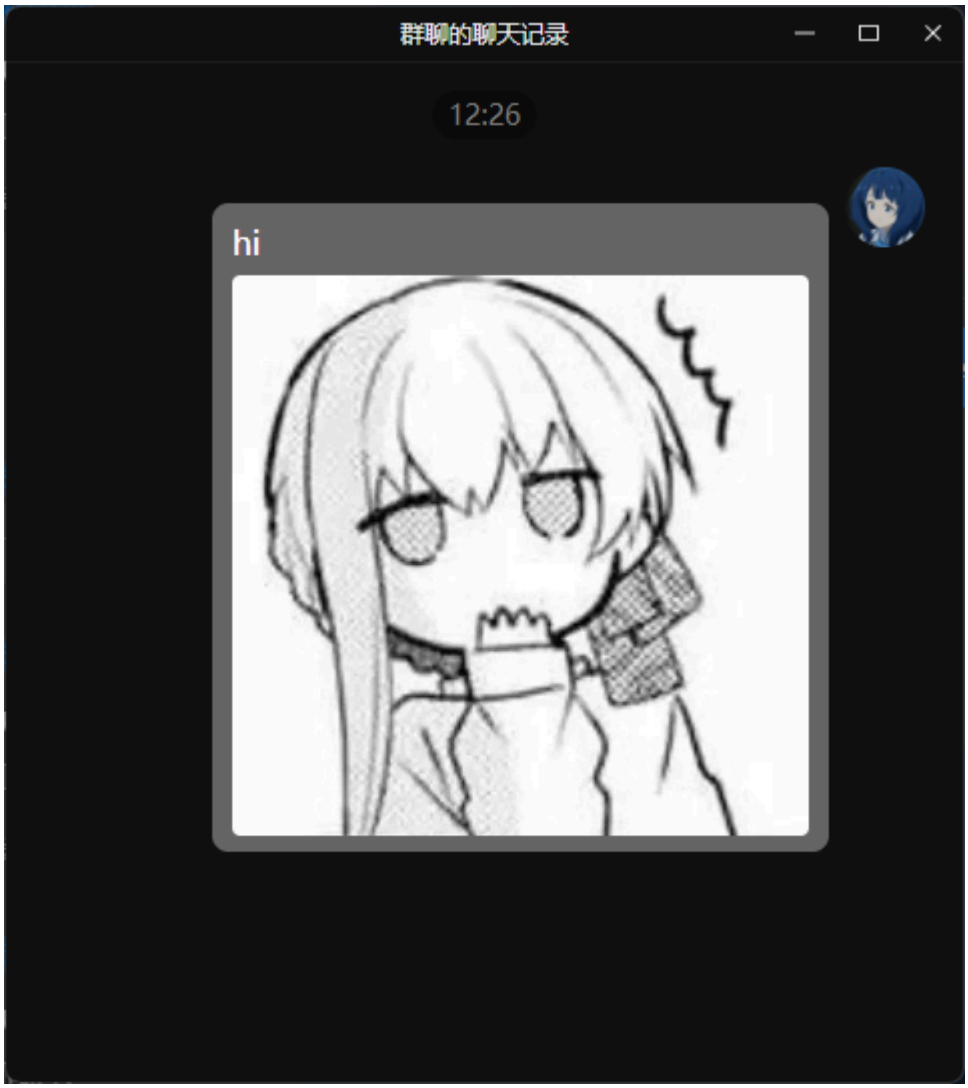
python
```python
yield event.make_result().message("文本消息")
                         .url_image("https://example.com/image.jpg")
                         .file_image("path/to/image.jpg")
```

发送群合并转发消息

> 当前适配情况：aiocqhttp

可以按照如下方式发送群合并转发消息。

py
```python
from astrbot.api.event import filter, AstrMessageEvent

@filter.command("test")
async def test(self, event: AstrMessageEvent):
    from astrbot.api.message_components import Node, Plain, Image
    node = Node(
        uin=905617992,
        name="Soulter",
        content=[
            Plain("hi"),
            Image.fromFileSystem("test.jpg")
        ]
    )
    yield event.chain_result([node])
```

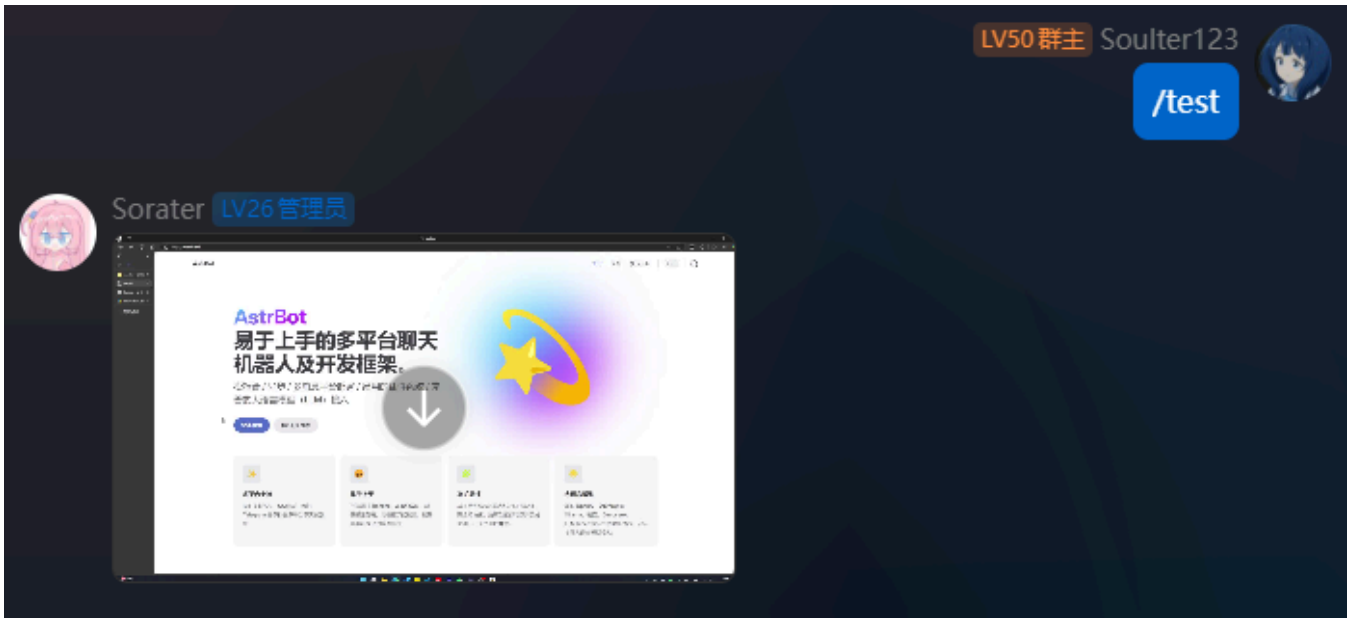


发送视频消息

> 当前适配情况：aiocqhttp

python

```python
from astrbot.api.event import filter, AstrMessageEvent

@filter.command("test")
async def test(self, event: AstrMessageEvent):
    from astrbot.api.message_components import Video
    # fromFileSystem 需要用户的协议端和机器人端处于一个系统中。
    music = Video.fromFileSystem(
        path="test.mp4"
    )
    # 更通用
    music = Video.fromURL(
        url="https://example.com/video.mp4"
    )
    yield event.chain_result([music])
```



发送 QQ 表情

> 当前适配情况：仅 aiocqhttp

QQ 表情 ID 参考：https://bot.q.qq.com/wiki/develop/api-v2/openapi/emoji/model.html#EmojiType

python

```python
from astrbot.api.event import filter, AstrMessageEvent

@filter.command("test")
async def test(self, event: AstrMessageEvent):
    from astrbot.api.message_components import Face, Plain
    yield event.chain_result([Face(id=21), Plain("你好呀")])
```

获取平台适配器/客户端

> v3.4.34 后

python

```python
from astrbot.api.event import filter, AstrMessageEvent

@filter.command("test")
async def test_(self, event: AstrMessageEvent):
    from astrbot.api.platform import AiocqhttpAdapter # 其他平台同理
    platform = self.context.get_platform(filter.PlatformAdapterType.AIOCQHTTP)
    assert isinstance(platform, AiocqhttpAdapter)
    # platform.get_client().api.call_action()
```

[aiocqhttp] 直接调用协议端 API

py

```python
@filter.command("helloworld")
async def helloworld(self, event: AstrMessageEvent):
    if event.get_platform_name() == "aiocqhttp":
        # qq
        from astrbot.core.platform.sources.aiocqhttp.aiocqhttp_message_event import
        assert isinstance(event, AiocqhttpMessageEvent)
        client = event.bot # 得到 client
        payloads = {
            "message_id": event.message_obj.message_id,
        }
        ret = await client.api.call_action('delete_msg', **payloads) # 调用 协议端  A
        logger.info(f"delete_msg: {ret}")
```

关于 CQHTTP API，请参考如下文档：

Napcat API 文档：https://napcat.apifox.cn/

Lagrange API 文档：https://lagrange-onebot.apifox.cn/

[gewechat] 平台发送消息

py

```python
@filter.command("helloworld")
async def helloworld(self, event: AstrMessageEvent):
    if event.get_platform_name() == "gewechat":
        from astrbot.core.platform.sources.gewechat.gewechat_platform_adapter import
        assert isinstance(event, GewechatPlatformEvent)
        client = event.client
        to_wxid = self.message_obj.raw_message.get('to_wxid', None)
```

```
        # await client.post_text()
        # await client.post_image()
        # await client.post_voice()
```

控制事件传播

python

```python
@filter.command("check_ok")
async def check_ok(self, event: AstrMessageEvent):
    ok = self.check() # 自己的逻辑
    if not ok:
        yield event.plain_result("检查失败")
        event.stop_event() # 停止事件传播
```

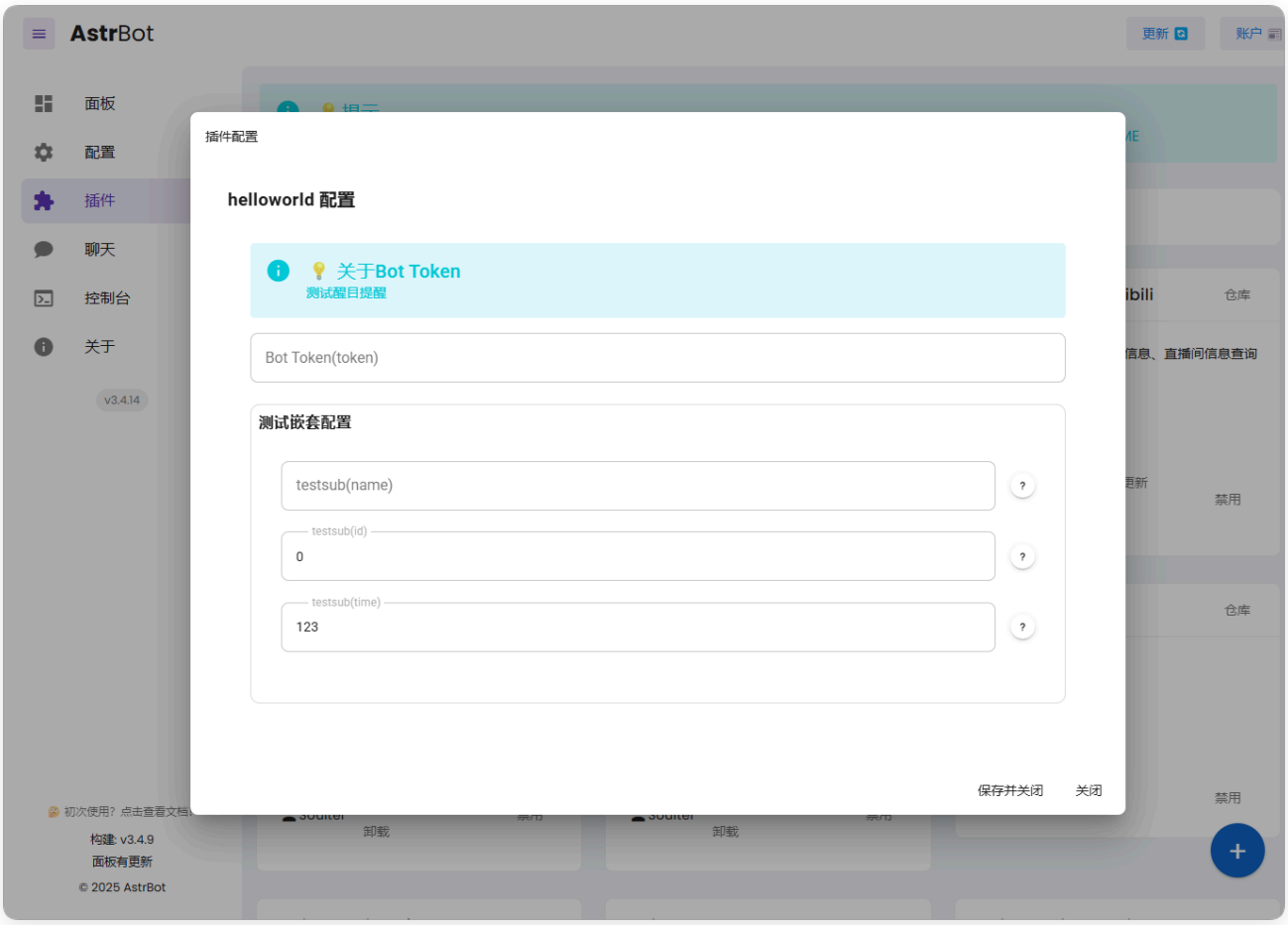
当事件停止传播，**后续所有步骤将不会被执行。**假设有一个插件A，A终止事件传播之后所有后续操作都不会执行，比如执行其它插件的handler、请求LLM。

注册插件配置(beta)

> 大于等于 v3.4.15

随着插件功能的增加，可能需要定义一些配置以让用户自定义插件的行为。

AstrBot 提供了”强大“的配置解析和可视化功能。能够让用户在管理面板上直接配置插件，而不需要修改代码。



Schema 介绍

要注册配置，首先需要在您的插件目录下添加一个 `_conf_schema.json` 的 json 文件。

文件内容是一个 `Schema` （模式），用于表示配置。Schema 是 json 格式的，例如上图的 Schema 是:

json

```
{
    "token": {
        "description": "Bot Token",
        "type": "string",
        "hint": "测试醒目提醒",
        "obvious_hint": true
    },
    "sub_config": {
        "description": "测试嵌套配置",
        "type": "object",
        "hint": "xxxx",
        "items": {
            "name": {
                "description": "testsub",
                "type": "string",
```

```
            "hint": "xxxx"
        },
        "id": {
            "description": "testsub",
            "type": "int",
            "hint": "xxxx"
        },
        "time": {
            "description": "testsub",
            "type": "int",
            "hint": "xxxx",
            "default": 123
        }
    }
}
}
```

- `type`：**此项必填**。配置的类型。支持 `string`，`int`，`float`，`bool`，`object`，`list`。
- `description`：可选。配置的描述。建议一句话描述配置的行为。
- `hint`：可选。配置的提示信息，表现在上图中右边的问号按钮，当鼠标悬浮在问号按钮上时显示。
- `obvious_hint`：可选。配置的 hint 是否醒目显示。如上图的 `token`。
- `default`：可选。配置的默认值。如果用户没有配置，将使用默认值。int 是 0，float 是 0.0，bool 是 False，string 是 ""，object 是 {}，list 是 []。
- `items`：可选。如果配置的类型是 `object`，需要添加 `items` 字段。`items` 的内容是这个配置项的子 Schema。理论上可以无限嵌套，但是不建议过多嵌套。
- `invisible`：可选。配置是否隐藏。默认是 `false`。如果设置为 `true`，则不会在管理面板上显示。
- `options`：可选。一个列表，如 `"options": ["chat", "agent", "workflow"]`。提供下拉列表可选项。

使用配置

AstrBot 在载入插件时会检测插件目录下是否有 `_conf_schema.json` 文件，如果有，会自动解析配置并保存在 `data/config/<plugin_name>_config.json` 下（依照 Schema 创建的配置文件实体），并在实例化插件类时传入给 `__init__()`。

py

```
from astrbot.api import AstrBotConfig
```

```python
@register("config", "Soulter", "一个配置示例", "1.0.0")
class ConfigPlugin(Star):
    def __init__(self, context: Context, config: AstrBotConfig): # AstrBotConfig 继承
        super().__init__(context)
        self.config = config
        print(self.config)

        # 支持直接保存配置
        # self.config.save_config() # 保存配置
```

配置版本管理

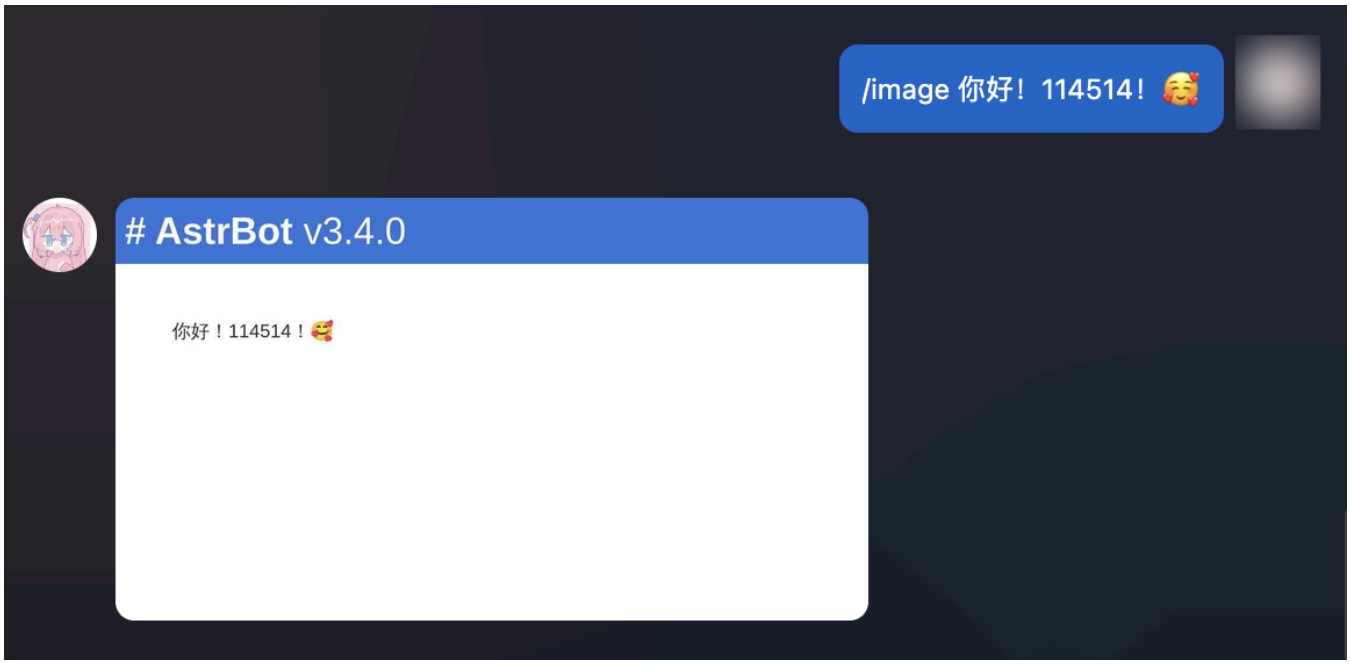
如果您在发布不同版本时更新了 Schema，请注意，AstrBot 会递归检查 Schema 的配置项，如果发现配置文件中缺失了某个配置项，会自动添加默认值。但是 AstrBot 不会删除配置文件中**多余的**配置项，即使这个配置项在新的 Schema 中不存在（您在新的 Schema 中删除了这个配置项）。

文字渲染成图片

AstrBot 支持将文字渲染成图片。

python

```python
@filter.command("image") # 注册一个 /image 指令，接收 text 参数。
async def on_aiocqhttp(self, event: AstrMessageEvent, text: str):
    url = await self.text_to_image(text) # text_to_image() 是 Star 类的一个方法。
    # path = await self.text_to_image(text, return_url = False) # 如果你想保存图片到本
    yield event.image_result(url)
```



自定义 HTML 渲染成图片

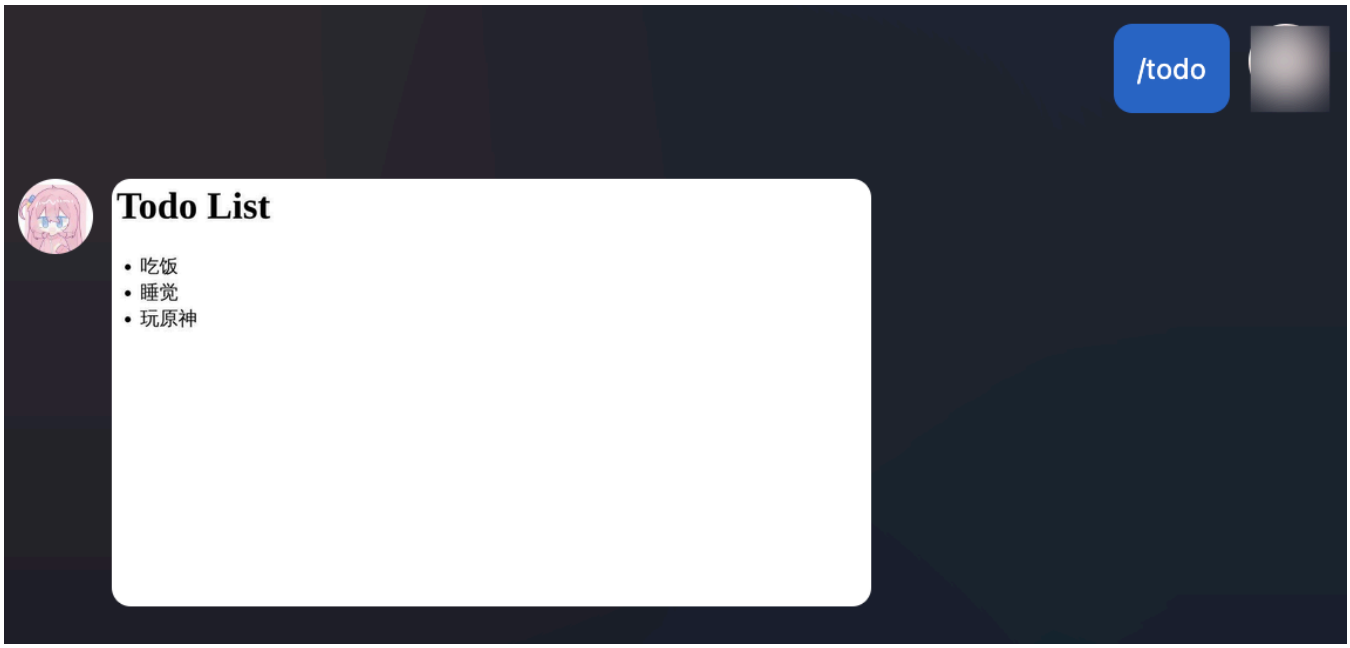
如果你觉得上面渲染出来的图片不够美观，你可以使用自定义的 HTML 模板来渲染图片。

AstrBot 支持使用 `HTML + Jinja2` 的方式来渲染文转图模板。

py

```
# 自定义的 Jinja2 模板，支持 CSS
TMPL = '''
<div style="font-size: 32px;">
<h1 style="color: black">Todo List</h1>

<ul>
{% for item in items %}
    <li>{{ item }}</li>
{% endfor %}
</div>
'''

@filter.command("todo")
async def custom_t2i_tmpl(self, event: AstrMessageEvent):
    url = await self.html_render(TMPL, {"items": ["吃饭", "睡觉", "玩原神"]}) # 第二个
    yield event.image_result(url)
```

返回的结果:



> 这只是一个简单的例子。得益于 HTML 和 DOM 渲染器的强大性，你可以进行更复杂和更美观的的设计。除此之外，Jinja2 支持循环、条件等语法以适应列表、字典等数据结构。你可以从网上了解更多关于 Jinja2 的知识。

调用 LLM

AstrBot 支持调用大语言模型。你可以通过 `self.context.get_using_provider()` 来获取当前使用的大语言模型提供商，但是需要启用大语言模型。

python

```python
from astrbot.api.event import filter, AstrMessageEvent

@filter.command("test")
async def test(self, event: AstrMessageEvent):
    func_tools_mgr = self.context.get_llm_tool_manager()

    # 获取用户当前与 LLM 的对话以获得上下文信息。
    curr_cid = await self.context.conversation_manager.get_curr_conversation_id(even
    conversation = None # 对话对象
    context = [] # 上下文列表
    if curr_cid:
        conversation = await self.context.conversation_manager.get_conversation(even
        context = json.loads(conversation.history)
    # 可以用这个方法自行为用户新建一个对话
    # curr_cid = await self.context.conversation_manager.new_conversation(event.unif

    # 方法1. 最底层的调用 LLM 的方式，如果启用了函数调用，不会进行产生任何副作用（不会调用函
    llm_response = await self.context.get_using_provider().text_chat(
        prompt="你好",
```

```
        session_id=None, # 此已经被废弃
        contexts=[], # 也可以用上面获得的用户当前的对话记录 context
        image_urls=[], # 图片链接，支持路径和网络链接
        func_tool=func_tools_mgr, # 当前用户启用的函数调用工具。如果不需要，可以不传
        system_prompt=""  # 系统提示，可以不传
    )
    # contexts 是历史记录。格式与 OpenAI 的上下文格式格式一致。即使用户正在使用 gemini，也
    # contexts = [
    #     { "role": "system", "content": "你是一个助手。"},
    #     { "role": "user", "content": "你好"}
    # ]
    # text_chat() 将会将 contexts 和 prompt,image_urls 合并起来形成一个上下文，然后调用
    if llm_response.role == "assistant":
        print(llm_response.completion_text) # 回复的文本
    elif llm_response.role == "tool":
        print(llm_response.tools_call_name, llm_response.tools_call_args) # 调用的函数
    print(llm_response.raw_completion) # LLM 的原始响应，OpenAI 格式。其存储了包括 toke

    # 方法2. 以下方法将会经过 AstrBot 内部的 LLM 处理机制。会自动执行函数工具等。结果将会直
    yield event.request_llm(
        prompt="你好",
        func_tool_manager=func_tools_mgr,
        session_id=curr_cid, # 对话id。如果指定了对话id，将会记录对话到数据库
        contexts=context, # 列表。如果不为空，将会使用此上下文与 LLM 对话。
        system_prompt="",
        image_urls=[], # 图片链接，支持路径和网络链接
        conversation=conversation # 如果指定了对话，将会记录对话
    )
```

注册一个 LLM 函数工具

`function-calling` 给了大语言模型调用外部工具的能力。

注册一个 `function-calling` 函数工具。

请务必按照以下格式编写一个工具（包括**函数注释**，AstrBot 会尝试解析该函数注释）

py

```
@llm_tool(name="get_weather") # 如果 name 不填，将使用函数名
async def get_weather(self, event: AstrMessageEvent, location: str) -> MessageEventR
    '''获取天气信息。
```

```
    Args:
        location(string): 地点
    '''
    resp = self.get_weather_from_api(location)
    yield event.plain_result("天气信息: " + resp)
```

在 `location(string): 地点` 中， `location` 是参数名， `string` 是参数类型， `地点` 是参数描述。

支持的参数类型有 `string` , `number` , `object` , `array` , `boolean` 。

> **WARNING**
>
> 请务必将注释格式写对！

获取 AstrBot 配置

```py
config = self.context.get_config()
# 使用方式类似 dict，如 config['provider']
# config.save_config() 保存配置
```

获取当前载入的所有提供商

```py
providers = self.context.get_all_providers()
providers_stt = self.context.get_all_stt_providers()
providers_tts = self.context.get_all_tts_providers()
```

获取当前正在使用提供商

```py
provider = self.context.get_using_provider() # 没有使用时返回 None
provider_stt = self.context.get_using_stt_provider() # 没有使用时返回 None
provider_tts = self.context.get_using_tts_provider() # 没有使用时返回 None
```

通过提供商 ID 获取提供商

py

```
self.context.get_provider_by_id(id_str)
```

获取当前载入的所有插件

py

```
plugins = self.context.get_all_stars() # 返回 StarMetadata 包含了插件类实例、配置等等
```

获取函数调用管理器

py

```
self.context.get_llm_tool_manager() # 返回 FuncCall

# self.context.get_using_provider().text_chat(
#     prompt="你好",
#     session_id=None,
#     contexts=[],
#     image_urls=[],
#     func_tool=self.context.get_llm_tool_manager(),
#     system_prompt=""
# )
```

注册一个异步任务

直接在 init() 中使用 `asyncio.create_task()` 即可。

py

```
import asyncio

@register("task", "Soulter", "一个异步任务示例", "1.0.0")
class TaskPlugin(Star):
    def __init__(self, context: Context):
        super().__init__(context)
        asyncio.create_task(self.my_task())

    async def my_task(self):
        await asyncio.sleep(1)
        print("Hello")
```

获取载入的所有人格(Persona)

```py
from astrbot.api.provider import Personality
personas = self.context.provider_manager.personas # List[Personality]
```

获取会话正在使用的对话

```py
from astrbot.core.conversation_mgr import Conversation
uid = event.unified_msg_origin
curr_cid = await self.context.conversation_manager.get_curr_conversation_id(uid)
conversation = await self.context.conversation_manager.get_conversation(uid, curr_ci
# context = json.loads(conversation.history) # 获取上下文
# persona_id = conversation.persona_id # 获取对话使用的人格

# 当 persona_id 为 None 时，这个对话为默认人格，即 self.context.provider_manager.select
# 当 persona_id 为 "[%None]" 时，为用户主动在这个对话取消了人格（通过 /persona unset 设置
```

获取会话的所有对话

```py
from astrbot.core.conversation_mgr import Conversation
uid = event.unified_msg_origin
conversations = await self.context.conversation_manager.get_conversations(uid) # Lis
```

获取加载的所有平台

```py
from astrbot.api.platform import Platform
platforms = self.context.platform_manager.get_insts() # List[Platform]
```