Python 事件响应器

前言

最近做的项目用到的事件驱动这一功能，简单的说就是当后台检测到事件时执行对应函数。由于$Nonebot2$中的事件驱动非常完善，于是仔细研究了$Nonebot2$的源码。

组成

事件响应有三部分组成：

1. 事件响应管理器($MatcherManager$)
2. 事件响应器($Matcher$)
3. 事件($Event$)
   流程
   $Event \rightarrow MatcherManager \rightarrow Matcher \rightarrow handler$

实现

事件响应管理器

> 事件管理器应该是一个储存事件和响应器的对象。

```python
class matcherManager:
    def __init__(self) -> None:

        self._eventQueue = Queue()
        """存放事件队列"""

        self._thread = Thread(target=self._Run)
        """事件处理线程"""

        self._active = False
        """事件开关"""

        self._provider: defaultdict[int,List[Type[Matcher]]] = {}
        """存放字典：事件 => 处理函数"""

        self._delay = 0.1
        """分发事件延迟"""
```

由于不同事件响应器的优先度不同，所以使用字典来存放响应器，字典的Key就是优先级$Priority$，事件队列$eventQueue$用来存放收到的事件。

而后应该有一个线程来轮询事件队列，如果有新的事件发生，那么就将它传给对应的

响应器。

```python
def _Run(self) -> None:
    """分发事件函数"""
    while self._active:
        try:
            event = self._eventQueue.get(block = True, timeout = 1)
            # logger.info(f'检索到新事件 {event.name}')
            self._EventProcess(event)
        except Empty:
            pass
        time.sleep(self._delay)

def _EventProcess(self, event) -> None:
    """事件处理函数"""
    if event.type_ in self._provider.keys():
        """取出对应事件的每一个Matcher类"""
        for matcher in self._provider[event.type_]:
            """临时响应器,使用后删除"""
            asyncio.run(matcher.run())
            if matcher.disposable:
                self._provider[event.type_].remove(matcher)

        return
    """如果没有对应Matcher且event.keep=True则保留事件"""
    if event.keep:
        self._eventQueue.put(event)
```

这样就是一个事件响应管理器的雏形，我们只需要管理$_{R}un$这一线程，就可以开关管理器。在$Nonebot2$中，他还添加了$Rule$, $Permisson$等权限来匹配事件，需要可以进一步研究

事件响应器

> 事件响应器$Matcher$是一个存储触发事件类型和响应函数的对象

```python
class Matcher:
    def __init__(self,
                 type_: str,
                 priority: int = 1,
                 disposable: bool = False,
                 plugin: Plugin = None) -> None:

        self._type_: str = type_
        """事件响应器类型"""
        self._handlers: List = []
        """事件响应器拥有的事件处理函数列表"""
        self._priority: int = priority
        """事件响应器优先级"""
        self._disposable: bool = disposable
        """事件响应器是否为临时"""
        self._plugin: Optional["Plugin"] = plugin
        """事件响应器所在插件"""
```

图片中我们使用了$handlers$来储存响应函数，当我们成功在管理器$Manager$中注册了响应器$Matcher$, 只需要在响应器中添加响应函数，就可以实现上文流程。

$Nonebot2$中很巧妙的使用了装饰器将函数添加到了响应器中。

```python
@classmethod
def handle(
    cls, parameterless: Optional[Iterable[Any]] = None
) -> Callable[[T_Handler], T_Handler]:
    """装饰一个函数来向事件响应器直接添加一个处理函数

    参数:
        parameterless: 非参数类型依赖列表
    """

    def _decorator(func: T_Handler) -> T_Handler:
        cls.append_handler(func, parameterless=parameterless)
        return func

    return _decorator
```

我无法理解的是$Nonebot2$使用$classmethod$装饰$handle$函数，这样只能使用cls操作静态对象，也就是说实例化的不同$Matcher$对象都将共享同一个处理函数列表$handlers$。举个例

子，我注册了两个*Matcher*，一个是*start*来监听程序启动，一个是*end*来监听程序关闭。并且我用`@start.handle`来装饰启动程序，用`@end.handle`来装饰关闭程序，如下：

```
start = Matcher('start')
end = Matcher('end')

@start.handle
async def startFunc():
        print('程序启动')
@end.handle
async def endFunc():
        print('程序关闭')
```

那么`start`和`end`的*handlers*中都会出现这两个函数，因为*handlers*是静态对象。我的水平暂时还无法理解这个问题，所以更换了一种方法实现这个功能

```
def _append_handler(self, handle) -> None:
    """添加事件处理"""
    self._handlers.append(handle)
    logger.info(f'register a new handle on Matcher {self._type_}')


def _remove_handler(self, handle) -> None:
    """移除事件处理"""
    if handle not in self._handlers:
        raise RuntimeError(f"Event {handle.__name__} doesn't exist!")
    self._handlers.remove(handle)

def handle(self, func) -> Callable[..., Any]:
    """事件装饰器，将handle添加到事件处理器中"""
    self._append_handler(func)
    return func
```

这样就可以将响应函数添加到列表中，其实这样的装饰器是一个语法糖，他的功能和在函数之后添加一个`matcher._handlers.append(...)`作用相同，不过更方便使用。

最后我们还需要一个处理函数，当响应器被触发时调用所有的响应函数。

```
async def run(self, event: Event) -> None:
    for handle in self._handlers:
        try:
            await handle(event)
        except:
            logger.error(f"there are problems when handling the event on matcher {self._type_}")
```

这样就完成了响应器$Matcher$的功能

事件

事件应该是最简单的部分，存放需要的属性即可。

```python
class Event:
    """事件类"""
    def __init__(self,
                 name,
                 type_,
                 keep=False,
                 data=None) -> None:

        self.name: str = name
        """事件名称"""

        self.type_: str = type_
        """事件类型"""

        self.data: list = data
        """事件信息"""

        self.keep = keep
        """保留"""

    def __str__(self) -> str:
        return f"{self.name} - {self.priority}"
```

整合

1. 在事件响应管理器中创建事件响应器

```python
def new(
    self,
    type_: str = "",
    disposable: bool = False,
    priority: int = 1,
    plugin: Optional["Plugin"] = None,
) -> Type["Matcher"]:
    """
    创建一个新的事件响应器，并存储至 `matchers <#matchers>`_
    参数:
        type_: 事件响应器类型
        disposable: 是否为临时事件响应器，即触发一次后删除
        priority: 响应优先级
        plugin: 事件响应器所在插件
    """
    """存在Matcher则返回"""
    NewMatcher = Matcher(
            plugin = plugin,
            type_ =  type_,
            disposable = disposable,
            priority = priority,
    )
    if priority in self._provider.keys():
        self._provider[priority].append(NewMatcher)
    else:
        self._provider[priority] = [NewMatcher]
    logger.info(f'created a new matcher {type_} in matchers')
    return NewMatcher
```

2. 发送事件

```python
def send(self, event: Event) -> None:
    """发送事件"""
    self._eventQueue.put(event)
```

3. 接口

我们只需要在模块`__init__.py`中实例化一个$MatcherManager$单例，对其进行操作即可。

```python
from .event import Event
from plugin import Plugin
from matcher.matcher import Matcher
from matcher.matcherManager import matcherManager

matchers = matcherManager()
```

改进

1. 注册响应器的new函数应该检测注册的$Matcher$是否存在，如果已存在应该返回其实例，不应该创建重复响应器。
2. 注册响应器的new函数应该定义为内部函数，在`__init__.py`中实现注册响应器接口，这样能使响应器注册更加规范。（如图我建立一个api开关事件响应器的注册接口，new函数应该在内部进行操作）

```python
def on_api(mode: bool = True,
           disposable: bool = False,
           priority: int = 1,
           plugin: Plugin = None) -> Matcher:
    return matchers.new(
        type_='apiRun' if mode else 'apiClose',
        disposable=True,
        priority=priority,
        plugin=plugin
    )
```

最后

1. 事件轮询部分还可以改进，用`while self._active:`轮询事件队列感觉效率较低，目前水平还没想到解决办法。
2. $Nonebot2$的插件管理部分会在下次分享。