# **4** РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

# **4.1** Алгоритм работы функции process\_received\_events

Функция process\_received\_events предназначена для обработки и отправки событий файловой системы в отслеживаемых директориях. Она принимает в качестве аргументов объект контекста Inotify – monitor, менеджер обмена сообщениями - monitor\_messaging\_manager, список, предназначенный для возможных парных событий - pair\_events\_list, и объект события Event - stop\_events\_processing\_flag. Сама функция выглядит следующим образом:

def process\_received\_events(monitor, \

messaging\_manager, pair\_events\_list, \

stop\_events\_processing\_flag):

for event in monitor.event\_gen():

if event is not None:

messages = process\_possible\_pair\_event(event, \

pair\_events\_list)

if messages is None:

message = get\_message\_by\_event(event)

messaging\_manager.send\_message(str(message))

else:

for message in messages:

messaging\_manager.send\_message( \

str(message))

else:

if stop\_events\_processing\_flag.is\_set():

stop\_events\_processing\_flag.clear()

break

Основная работа функции происходит в цикле считывания событий из объекта контекста Inotify:

for event in monitor.event\_gen():

Метод event\_gen возвращает объект события если оно есть в очереди событий и None если очередь пуста. После цикла идёт проверка того появилось новое событие или нет. Если оно появилось – мы передаём событие и список возможных парных событий функции process\_possible\_pair\_event. Она возвращает пустой список, если текущее событие первое из парных, один объект сообщения если в списке уже было парное событие, а текущее событие также парное, два объекта сообщения, если текущее событие не парное, а в списке уже было возможно парное событие и None, если список пуст и текущее событие не парное.

Далее проверяется вернула ли функция обработки возможных парных событий какие-либо сообщения:

if messages is None:

Если сообщения\сообщений нет – оно получается по событию с помощью функции get\_message\_by\_event и отправляется в виде строки с помощью метода send\_message объекта менеджера пересылки сообщений:

message = get\_message\_by\_event(event)

messaging\_manager.send\_message(str(message))

Если сообщения есть – они также отправляются через объект менеджера пересылки сообщений, но уже в цикле:

for message in messages:

messaging\_manager.send\_message(str(message))

В ситуации, когда событие не пришло и метод event\_gen возвратил None проверяется флаг выхода из функции - stop\_events\_processing\_flag, который устанавливается таймером раз в 10 секунд в вызывающем коде.

stop\_event\_timer = Timer(10, stop\_event.set)

stop\_event\_timer.start()

Этот флаг нужен для того что бы обработка событий не занимала всё время монитора и каждые 10 секунд монитор имел бы возможность проверять входящие сообщения. Такое неравное распределение времени объясняется тем, что входящие сообщения присылаются гораздо реже чем отсылаются исходящие.

**4.2** Алгоритм работы функции process\_possible\_pair\_event

Функция process\_possible\_pair\_event предназначена для обработки событий, которые могут быть парными и для которых нужно генерировать одно сообщение вместо двух. Примером такого события может служить переименование файла, которое состоит из пары событий: файл перемещён из директории и файл перемещён в директорию. Эту ситуацию надо отличать от событий реального перемещения файла. Код данной функции представлен ниже:

def process\_possible\_pair\_event( \

current\_event, pair\_event\_list):

(header, \_, watch\_path, \_) = current\_event

if len(pair\_event\_list) == 0 and is\_first\_pair\_event( \

header):

pair\_event\_list.append(current\_event)

return []

elif len(pair\_event\_list) != 0:

previous\_event = pair\_event\_list.pop()

(\_, \_, previous\_watch\_path, \_) = previous\_event

if previous\_watch\_path == watch\_path and \

is\_second\_pair\_event(header):

pair\_events\_message = get\_message\_by\_event\_pair(

previous\_event, current\_event)

return [ pair\_events\_message ]

elif is\_first\_pair\_event(header):

pair\_event\_list.append(current\_event)

previous\_event\_message = get\_message\_by\_event( \

previous\_event)

return [ previous\_event\_message ]

else:

previous\_event\_message = get\_message\_by\_event( \

previous\_event)

current\_event\_message = get\_message\_by\_event( \

current\_event)

return [ previous\_event\_message, \

current\_event\_message ]

return None

Функция принимает в качестве аргументов текущее событие current\_event и список с возможными парными событиями - pair\_event\_list.

Объект current\_event представляет собой кортеж, из которого нужно выделить заголовок с метаданными события и путь к отслеживаемой директории.

(header, \_, watch\_path, \_) = current\_event

Далее проверяется список возможных парных событий. Если список пуст, нужно проверить может ли текущее событие являться первым из парных. Проверку осуществляет функция is\_first\_pair\_event, в которую передаётся заголовок с метаданными о событии. Если оба условия соблюдаются – событие добавляется в список pair\_event\_list и функция возвращает пустой список.

pair\_event\_list.append(current\_event)

return []

Если в списке уже есть первое парное событие, то оно извлекается и раскладывается на составные части для получения пути к директории:

previous\_event = pair\_event\_list.pop()

(\_, \_, previous\_watch\_path, \_) = previous\_event

Возникает три ответвления для трёх разных ситуаций. Для первой ситуации проверяется совпадает ли у двух событий имя отслеживаемой директории и, с помощью функции is\_second\_pair\_event, является ли текущее событие вторым парным. В случае если оба условия соблюдены – с помощью функции get\_message\_by\_event\_pair получаем один объект сообщения для двух событий и возвращаем его в виде списка.

pair\_events\_message = get\_message\_by\_event\_pair(\

previous\_event, current\_event)

return [ pair\_events\_message ]

Для второй ситуации с помощью функции is\_first\_pair\_event идёт проверка - является ли текущее событие первым из парных. Если условие соблюдается – текущее событие добавляется в список, а предыдущее событие конвертируется в объект сообщения с помощью функции get\_message\_by\_event и возвращается в виде списка.

Если первые две ветки условного оператора не сработали – значит текущее и предыдущее события непарные. В таком случае они оба конвертируются в отдельные сообщения, которые возвращаются в виде списка:

previous\_event\_message =get\_message\_by\_event(previous\_event)

current\_event\_message = get\_message\_by\_event(current\_event)

return [ previous\_event\_message, current\_event\_message ]

Если список пуст и текущее событие не является первым парным – возвращается значение None, что даёт понять вызывающему коду что текущее событие нужно обрабатывать обычным образом.

В итоге, результатом выполнения функции при таком алгоритме является либо список, который может быть пустым или содержать объекты сообщений либо значение None.

## **4.3** Алгоритм работы метода run потока приёма и передачи сообщений

Метод run класса MessagingManagerBackgroudReceiver, представляющего собой поток Thread, предназначен для последовательного приёма и передачи сообщений. Этот метод запускается при старте потока и при его завершении поток завершает свою работу. Код метода выглядит следующим образом:

def run(self):

polling\_delay = 100

while not self.\_stop\_event.is\_set():

if self.\_send\_queue.qsize() != 0:

self.\_socket.send\_unicode( \

self.\_send\_queue.get())

events = self.\_socket.poll(polling\_delay)

if events != 0:

self.\_receive\_queue.put( \

self.\_socket.recv(flags=zmq.NOBLOCK))

self.\_socket.close()

self.\_context.destroy()

При старте этого метода запускается бесконечный цикл, который работает до тех пор, пока внешний код, создавший данный поток не установит событие, которое позволит выйти из цикла и корректно освободить ресурсы.

На первом этапе цикла проверяется есть ли в очереди \_send\_queue

сообщения на отправку. Если очередь не пуста одно сообщение из очереди отправляется с помощью метода send\_unicode сокета \_socket. Если очередь пуста – цикл переходит к следующему этапу.

if self.\_send\_queue.qsize() != 0

self.\_socket.send\_unicode( \

self.\_send\_queue.get())

На втором этапе идёт попытка получить сигнал о пришедшем сообщении за промежуток времени 100 мс. Такую возможность предоставляет метод сокета poll, куда в качестве задержки передаётся константа polling\_delay и который возвращает 0, если за заданное время никаких сообщений не пришло. Если сообщение пришло – оно принимается методом сокета recv и помещается в очередь принятых сообщений \_receive\_queue.

После того как вызывающий код выставил событие \_stop\_event и цикл завершился, освобождаются все ресурсы потока, а именно: контекст и сокет ZeroMQ.

## **4.4** Алгоритм работы метода get\_rules

Метод get\_rules класса RulesManager предназначен для получения списка правил, которые должны последовательно применится к файлу, для которого произошло событие в файловой системе. Метод принимает в качестве входных параметров информацию о событии event\_info и результат анализа файла analyze\_results. Код метода выглядит следующим образом:

def get\_rules(self, \

event\_info, analyze\_results=None):

all\_suitable\_rules = self.\_get\_all\_suitable\_rules( \

event\_info, analyze\_results)

if self.\_is\_ignore\_rule\_exist(all\_suitable\_rules):

return []

specific\_sorted\_rules = \

self.\_sort\_by\_specific\_level(all\_suitable\_rules)

top\_specific\_rules = \

self.\_get\_top\_specific\_rules( \

specific\_sorted\_rules)

delete\_rules = self.\_get\_rules\_by\_action\_type( \

top\_specific\_rules, action\_types.DELETE\_FILE)

if len(delete\_rules) != 0:

return [ delete\_rules[0] ]

return self.\_get\_final\_rules\_sequence( \

specific\_sorted\_rules)

Вся информация о событии и файле передаётся в приватный метод класса \_get\_all\_suitable\_rules, который возвращает список кортежей, где первый элемент это правило, подходящее для данного события, а второй – специфичность этого правила, которая считается как количество совпавших ограничений, установленных для файла.

Дальше идёт условие, которое проверяет, содержится ли в списке правило с действием IGNORE. Если оно есть – метод возвращает пустой список, так как к этому файлу никакие правила применятся не должны.

if self.\_is\_ignore\_rule\_exist(all\_suitable\_rules):

return []

Далее необходимо определить, есть ли среди наиболее специфичных правил, правило с действием удаления файла - DELETE\_FILE. Для этого полученный список правил с помощью приватного метода \_sort\_by\_specific\_level сортируется по убыванию специфичности правил, метод \_get\_top\_specific\_rules возвращает правила с наибольшей одинаковой специфичностью, а \_get\_rules\_by\_action\_type с параметрами в виде списка наиболее специфичных правил и типом DELETE\_FILE возвращает список правил удаления. Если этот список не пустой, искать цепочки правил не имеет смысла – файл всё равно будет удалён. Поэтому для такой ситуации мы возвращаем в списке одно из правил, которое приведёт к удалению файла:

if len(delete\_rules) != 0:

return [ delete\_rules[0] ]

Если правила для данного файла не игнорируются и файл не удаляется – следующим шагом будет построение цепочки правил, действия для которых можно было бы выполнить последовательно. Например, может выстроится цепочка правил, в которой последовательность действий будет такой: переименование файла, перемещение его в подкаталог, и отнесение его к одной из групп в этом подкаталоге. Для того что бы выстроить подобную цепочку правил, используется приватный метод \_get\_final\_rules\_sequence, в который передаются правила, отсортированные по специфичности. Цепочка из правил, полученная с помощью этого метода возвращается вызывающему коду.

В данном разделе были подробно описаны алгоритмы основных функций и методов приложения. Алгоритм работы всего приложения можно посмотреть на (ссылка на схему программы).