链滴

Django 使用 Channels 实现 websocket

作者：alaikis
原文链接：https://ld246.com/article/1518533741083
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

首先放上官方文档

安装配置

安装channels

如果使用的django是1.9包括以上的话，可以不用输入文档中**-U**参数，直接使用pip在终端中输入下命令即可

|

$ pip install channels

|

配置channels

想要使用channels，我们首先需要在setting里配置一下channels。

在INSTALLED_APPS中添加channels

|

INSTALLED_APPS = (

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.sites',

...

'channels',

)

|

配置channels路由和通道后端

简单的话我们可以使用内存作为后端，路由配置放在合适的地方
配置如下:

|

CHANNEL_LAYERS = {

```
“default” : {
    “BACKEND” : “asgiref.inmemory.ChannelLayer” ,
   # 这里是路由的路径，怎么填写都可以，只要能找到
    “ROUTING” : “你的工程名.routing.channel_routing” ,
},
}
```

由于我们已经使用了redis作为缓存系统，在这里我们也就正好使用redis在作为我们的通道后端。
为了使用redis作为channels的后端，我们还需要安装一个库asgi_redis。

1. 使用pip安装asgi_redis，在终端中输入

```
$ pip install asgi_redis
```

安装之后我们就可以使用redis作为channels的后端了

1. 修改channels的BACKEND

在**settings.py**修改

```
CHANNEL_LAYERS = {

"default": {
    "BACKEND": "asgi_redis.RedisChannelLayer",
    "CONFIG": {
        "hosts": [os.environ.get('REDIS_URL', 'redis://127.0.0.1:6379/2')],
    },
    # 配置路由的路径
    "ROUTING": "你的工程名.routing.channel_routing",
},
}
```

使用channels

使用channels，笔者主要是用来解决websocket连接和传输，这里不讨论http部分。

最简单的例子

1. 在合适的app下创建一个 **customers.py**,在其中编写代码如下

|

|

```
def ws_message(message):

# ASGI WebSocket packet-received and send-packet message types

# both have a "text" key for their textual data.

message.reply_channel.send({

    "text": message.content['text'],

})
```

|

2. 在同一个app下创建一个 **router.py**,在其中编写代码如下

|

|

```
from channels.routing import route
from .consumers import ws_message
channel_routing = [

route("websocket.receive", ws_message),
]
```

|

这里的意思就是当接收到前端发来的消息时，后端会触发**ws_message**函数，这里写的是一个回音壁序，就是把原数据在发送回去。

1. 前端代码如下，在浏览器的控制台或者一个html的js代码区域编写如下代码

|

|

```
// Note that the path doesn't matter for routing; any WebSocket
// connection gets bumped over to WebSocket consumers
socket = new WebSocket("ws://127.0.0.1:8000/chat/");
socket.onmessage = function(e) {
```

```
consoe.log(e.data);
}
socket.onopen = function() {

socket.send("hello world");
}
// Call onopen directly if socket is already open
if (socket.readyState == WebSocket.OPEN) socket.onopen();
|
```

然后就可以执行**python manage.py runserver**查看运行效果，如果不出意外的话应该可以看到效。

利用组的概念实现多个浏览器(用户)之间的交互

1. 在 **customers.py**中编写代码如下

```
|
|
from channels import Group

# Connected to websocket.connect
def ws_add(message):

# Accept the connection

message.reply_channel.send({"accept": True})

# Add to the chat group

Group("chat").add(message.reply_channel)

# Connected to websocket.receive
def ws_message(message):

Group("chat").send({

    "text": "[user] %s" % message.content['text'],

})

# Connected to websocket.disconnect
def ws_disconnect(message):
```

```
Group("chat").discard(message.reply_channel)
```

分为三个部分，分别是websocket连接的时候进行的操作，收到消息的时候进行的操作，和关闭链接时候进行的操作，这里利用了组的概念，在触发连接的时候，把其加入chat组，当收到消息时候，在内所有用户发送信息，最后关闭连接的时候退出组。

1. 由于将一次连接分为了三个部分，其路由也得配置三遍，所以在 **router.py**中编写代码如下

```
from channels.routing import route
from .consumers import ws_add, ws_message, ws_disconnect
channel_routing = [

route("websocket.connect", ws_add),

route("websocket.receive", ws_message),

route("websocket.disconnect", ws_disconnect),
]
```

2. 测试用前端代码如下:

```
// Note that the path doesn't matter right now; any WebSocket
// connection gets bumped over to WebSocket consumers
socket = new WebSocket("ws://127.0.0.1:8000/chat/");
socket.onmessage = function(e) {

 consoe.log(e.data);
```

```
}
socket.onopen = function() {

socket.send("hello world");
}
// Call onopen directly if socket is already open
if (socket.readyState == WebSocket.OPEN) socket.onopen();
```

然后就可以执行**python manage.py runserver**查看运行效果,

建议同时打开两个浏览器选项卡同时运行上述JavaScript代码，就能看到对方发来的消息啦。

上述代码还有一个问题，就是无论是谁访问同一个url都可以进到这个组里，我们也不能知道是谁进入这个组中，得到他的一些信息，所以就需要一些认证功能，不能让任何人都能加入该组，所以我们需认证

channels的认证

channels自带了很多很好用的修饰器来帮我们解决这个问题，我们可以访问到当前的session回话，者cookie。

- 使用 **http_session**修饰器就可以访问用户的session会话，拿到**request.session**
- 使用 **http_session_user**修饰器就可以获取到session中的用户信息,拿到**message.user**
- 使用 **channel_session_user**修饰器，就可以在通道中直接拿到**message.user**
- **channel_session_user_from_http**修饰器可以将以上修饰器的功能集合起来，直接获取到所需的户

以下是一个用户只能和用户名第一个字符相同的人聊天的程序代码

```
1
2
3
4
5
6
7
8
9
```

10

11

12

13

14

15

16

17

18

19

20

21

22

23

|

from channels import Channel, Group

from channels.sessions import channel_session

from channels.auth import channel_session_user, channel_session_user_from_http

Connected to websocket.connect

@channel_session_user_from_http

def ws_add(message):

Accept connection

message.reply_channel.send({"accept": True})

Add them to the right group

Group("chat-%s" % message.user.username[0]).add(message.reply_channel)

Connected to websocket.receive

@channel_session_user

```
def ws_message(message):

Group("chat-%s" % message.user.username[0]).send({

    "text": message['text'],

})
```

Connected to websocket.disconnect

```
@channel_session_user

def ws_disconnect(message):

Group("chat-%s" % message.user.username[0]).discard(message.reply_channel)
```

|

由于笔者的项目使用的是Json Web Token作为身份认证，对于服务器来说没有session，所以需要自实现一个认证。

Json Web Token认证

本来在http中使用ajax是将token放在请求头中的，但是在websocket中这样的方式并不可以，所以而求其次，我们只能将其放在url中或者发送的数据中了。

又因为笔者不想每次发消息都携带token，所以选择了在url中携带的方式，

最后发到服务器的url形式是这样的”ws://127.0.0.1:8000/chat/?token=eyJhbGciOiJIUzI1NiIsInR5CI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRlc3QxMjMiLCJvcmlnX2lhdCI6MTUwMzA0MzUyOCwidXNlc9pZCI6MSwiZW1haWwiOiIxNzkxNTM4NjA5QHFxLmNvbSIsImV4cCI6MTUwMzEyOTkyOH0.jNjNxUqXb1Ig6e3tdB9Xq2jH5LrqQe8zFLH40J9694”

我们需要实现一个修饰器去解决对token验证的问题，以备其他的使用

1. 在合适的地方创建一个 **ws_authentication.py**

|

1

2

3

4

5

6

7

8

9

10

85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121

coding=utf-8

```
from functools import wraps
from django.utils.translation import ugettext_lazy as _
from rest_framework import exceptions
from channels.handler import AsgiRequest
import jwt
from django.contrib.auth import get_user_model
from rest_framework_jwt.settings import api_settings
import logging
logger = logging.getLogger(name)  # 为loggers中定义的名称
User = get_user_model()
jwt_payload_handler = api_settings.JWT_PAYLOAD_HANDLER
jwt_encode_handler = api_settings.JWT_ENCODE_HANDLER
jwt_decode_handler = api_settings.JWT_DECODE_HANDLER
jwt_get_username_from_payload = api_settings.JWT_PAYLOAD_GET_USERNAME_HANDLER
def token_authenticate(token, message):

    """

    Tries to authenticate user based on the supplied token. It also checks

    the token structure and validity.

    """

    payload = check_payload(token=token, message=message)

    user = check_user(payload=payload, message=message)

    """Other authenticate operation"""

    return user, token
```

检查负载

```
def check_payload(token, message):

    payload = None

    try:
```

```
    payload = jwt_decode_handler(token)
except jwt.ExpiredSignature:
    msg = _('Signature has expired.')
    logger.warn(msg)
    # raise ValueError(msg)
    _close_reply_channel(message)
except jwt.DecodeError:
    msg = _('Error decoding signature.')
    logger.warn(msg)
    _close_reply_channel(message)
return payload
```

检查用户

```
def check_user(payload, message):

username = None
try:
    username = payload.get('username')
except Exception:
    msg = _('Invalid payload.')
    logger.warn(msg)
    _close_reply_channel(message)
if not username:
    msg = _('Invalid payload.')
    logger.warn(msg)
    _close_reply_channel(message)
    return
    # Make sure user exists
try:
```

```
    user = User.objects.get_by_natural_key(username)
except User.DoesNotExist:
    msg = _("User doesn't exist.")
    logger.warn(msg)
    raise exceptions.AuthenticationFailed(msg)
if not user.is_active:
    msg = _('User account is disabled.')
    logger.warn(msg)
    raise exceptions.AuthenticationFailed(msg)
return user
```

关闭websocket

```
def _close_reply_channel(message):

message.reply_channel.send({"close": True})
```

验证request中的token

```
def ws_auth_request_token(func):

"""
Checks the presence of a "token" request parameter and tries to
authenticate the user based on its content.
The request url must include token.
eg: /v1/channel/1/?token=abcdefghijklmn
"""
@wraps(func)

def inner(message, *args, **kwargs):
    try:
        if "method" not in message.content:
            message.content['method'] = "FAKE"
```

```
        request = AsgiRequest(message)
    except Exception as e:
        raise ValueError("Cannot parse HTTP message - are you sure this is a HTTP consumer? %
" % e)
    token = request.GET.get("token", None)
    print request.path, request.GET
    if token is None:
        _close_reply_channel(message)
        raise ValueError("Missing token request parameter. Closing channel.")
    # user, token = token_authenticate(token)
    user, token = token_authenticate(token, message)
    message.token = token
    message.user = user
    return func(message, *args, **kwargs)
return inner
```

由于笔者使用了django-restframework-jwt,其中的token验证方法是和其一样的，如果你的验证方不一样，可以自行替换。

有了上述代码，我们就可以在连接的时候判断token是否有效，以及是否还建立连接。

不过其中代码在错误处理的时候有些问题，我这里简单的处理为用日志打印和关闭连接。有知道怎么馈异常信息的可以在评论区告知我。

1. 在 **consumers.py**中使用修饰器去认证token

```
1
2
3
4
5
6
7
8
```

9
10
11
12
13
14
15
16
17
18
19
20
|

```
from channels import Group
from .ws_authentication import ws_auth_request_token
```

Connected to websocket.connect

```
@ws_auth_request_token
def ws_add(message):

# Accept the connection

message.reply_channel.send({"accept": True})

# Add to the chat group

Group("chat").add(message.reply_channel)
```

Connected to websocket.receive

```
def ws_message(message):

Group("chat").send({

    "text": "[user] %s" % message.content['text'],

})
```

Connected to websocket.disconnect

```
def ws_disconnect(message):

Group("chat").discard(message.reply_channel)
```

|

这样就能轻易的验证了。

使用类视图

django有一种类视图，在channels这里也可以，使用类视图可以让代码看着更简洁明了

1. 类视图可以将三种状态，连接，收到消息，关闭的时候写到一个类中，原来的 **consumers.py**代就可以改为如下代码

|

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

```
29
30
31
32
33
34
35
36
37
38
39
|
from channels.generic.websockets import WebsocketConsumer
class MyConsumer(WebsocketConsumer):

# Set to True to automatically port users from HTTP cookies

# (you don't need channel_session_user, this implies it)

http_user = True

# Set to True if you want it, else leave it out

strict_ordering = False

def connection_groups(self, **kwargs):

    """

    Called to return the list of groups to automatically add/remove

    this connection to/from.

    """

    return ["test"]

def connect(self, message, **kwargs):

    """

    Perform things on connection start

    """

    # Accept the connection; this is done by default if you don't override
```

```
    # the connect function.
    self.message.reply_channel.send({"accept": True})
def receive(self, text=None, bytes=None, **kwargs):
    """
    Called when a message is received with either text or bytes
    filled out.
    """
    # Simple echo
    self.send(text=text, bytes=bytes)
def disconnect(self, message, **kwargs):
    """
    Perform things on connection close
    """
    pass
```

|

然后在不同状态出发的函数中填入自己需要的逻辑即可

如果你想使用**channel_session**或者**channel_session_user**，那么只要在类中设置

|

1

|

channel_session_user = True

|

如果你想使用session里的用户，那么也需要在类中添加一个参数

|

1

|

http_user = True

|

1. 配置路由也需要做出一些变化

|

1

2

3

4

5

|

```
from channels import route, route_class
channel_routing = [

route_class(consumers.ChatServer, path=r"^/chat/"),
]
```

|

或者更简单一点

|

1

2

3

4

5

|

```
from . import consumers
channel_routing = [

consumers.ChatServer.as_route(path=r"^/chat/"),
]
```

|

在channels类视图中使用token认证

在类视图中添加修饰器较为麻烦，笔者认为将认证方法写在**connect(self, message, **kwargs)**即可。

所以**consumers.py**代码如下

|

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

56

57

|

```
class MyConsumer(WebsocketConsumer):

# Set to True if you want it, else leave it out
strict_ordering = False
http_user = True
# 由于使用的是token方式，需要使用session将user传递到receive中
channel_session_user = True
def connection_groups(self, **kwargs):
    """
    Called to return the list of groups to automatically add/remove
    this connection to/from.
    """
    return ['test']
    def connect(self, message, **kwargs):
    """
    Perform things on connection start
    """
    try:
        request = AsgiRequest(message)
    except Exception as e:
        self.close()
        return
    token = request.GET.get("token", None)
    if token is None:
        self.close()
```

```
        return
    user, token = token_authenticate(token, message)
    message.token = token
    message.user = user
    message.channel_session['user']=user
    self.message.reply_channel.send({"accept": True})
    print '连接状态', message.user
def receive(self, text=None, bytes=None, **kwargs):
    print '接收到消息', text, self.message.channel_session['user']
    """
    Called when a message is received with decoded JSON content
    """
    # Simple echo
    value = cache.get('test')
    print value
    while True:
        if cache.get('test') is not None and cache.get('test') != value:
            value = cache.get('test')
            break
        time.sleep(1)
    self.send(json.dumps({
        "text": cache.get('test')
    }))
def disconnect(self, message, **kwargs):
    """
    Perform things on connection close
    """
```

```
    pass
```

|

只需要看**connect(self, message, **kwargs)**函数中代码即可，**(self, text=None, bytes=None, **wargs)**中为我要实现的一个简单逻辑。

笔者发现，channels中的三个状态,其中每个自身只能发一次信息，无论我在一次方法中send几次，以我没办法，只能在前端的**onmessage**处理完数据，在发一次信息，后台将线程休眠等到参数变化发送到前端。前端代码改为如下

|

1

2

3

4

5

6

7

8

9

10

11

12

13

|

```
socket = new WebSocket("ws://127.0.0.1:8000"+

        "/chat/?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRlc3QxMjMiLJvcmlnX2lhdCI6MTUwMzA3Mzg0NiwidXNlcl9pZCI6MSwiZW1haWwiOiIxNzkxNTM4NjA5QHFLmNvbSIsImV4cCI6MTUwMzE2MDI0Nn0.Za0BlGKn2JMpFoU0GYVZXIC-rwi8uWN420bIwy0bUc"

    );

    socket.onmessage = function (e) {

        console.log(e.data);

        // socket.send("test")
```

```
    }
    socket.onopen = function () {
        socket.send({'test':'hello world'});
    }
    // Call onopen directly if socket is already open
    if (socket.readyState == WebSocket.OPEN) socket.onopen();
```

配合redis就可以实现django的websocket了，也可以满足我的需求，实时更新。