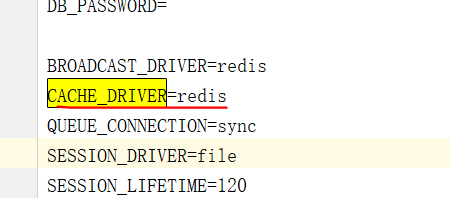
## 一．Redis

### 安装

composer require predis/predis

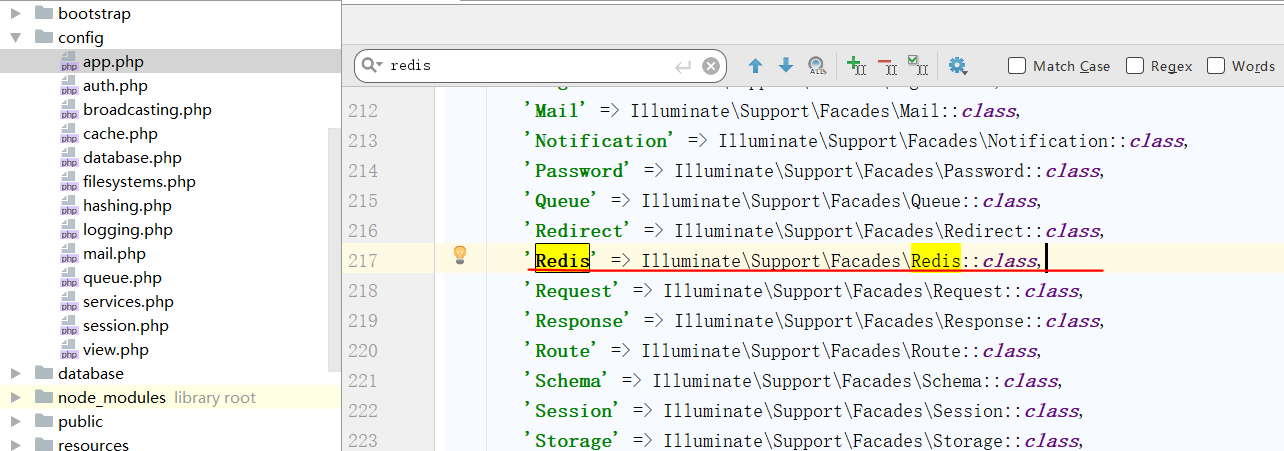
### 配置驱动信息

Laravel默认驱动是file我们改成redis



### 3.

并注释 config/app.php 中的 alias 中 Redis 一行防止 Redis 命名空间冲突



## 二．事件

官方文档：

<https://xueyuanjun.com/post/19508.html#bkmrk-%E4%BA%8B%E4%BB%B6%E7%9B%91%E5%90%AC%E5%99%A8%E9%98%9F%E5%88%97>

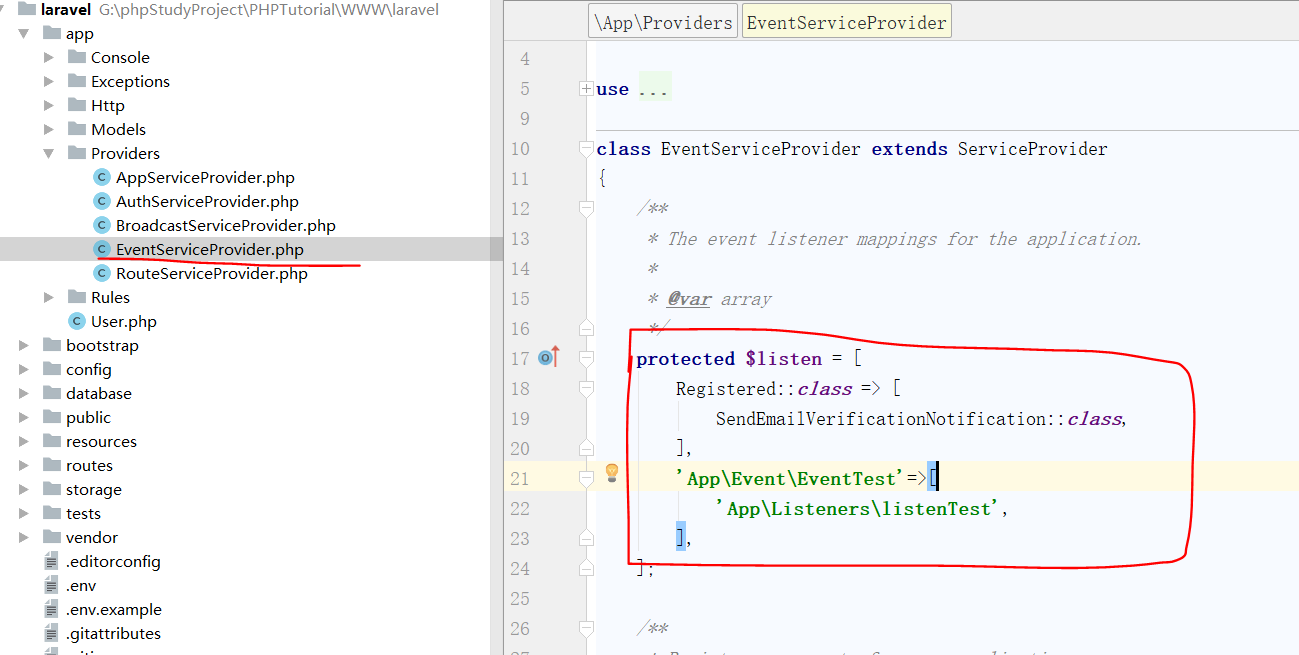
<https://learnku.com/laravel/t/28421>

<https://www.cnblogs.com/fps2tao/p/9681455.html>

场景：用户注册的时候发短信，邮件给用户

流程：

Controller-🡪触发事件类->事件类触发监听器



Php artisan event:generate 自动生成

**定义事件**

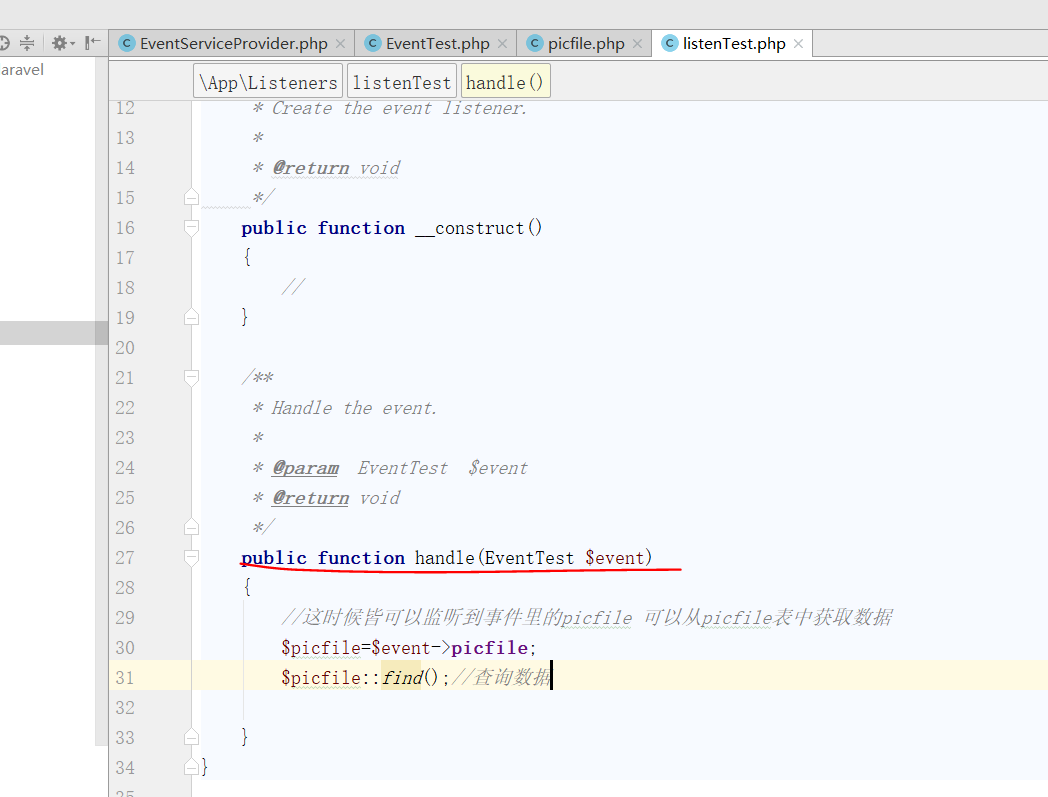


**定义监听器**

监听器在 handle 方法中接收事件实例

**停止事件继续往下传播**

有时候，你希望停止事件被传播到其它监听器，你可以通过从监听器的 handle 方法中返回 false 来实现

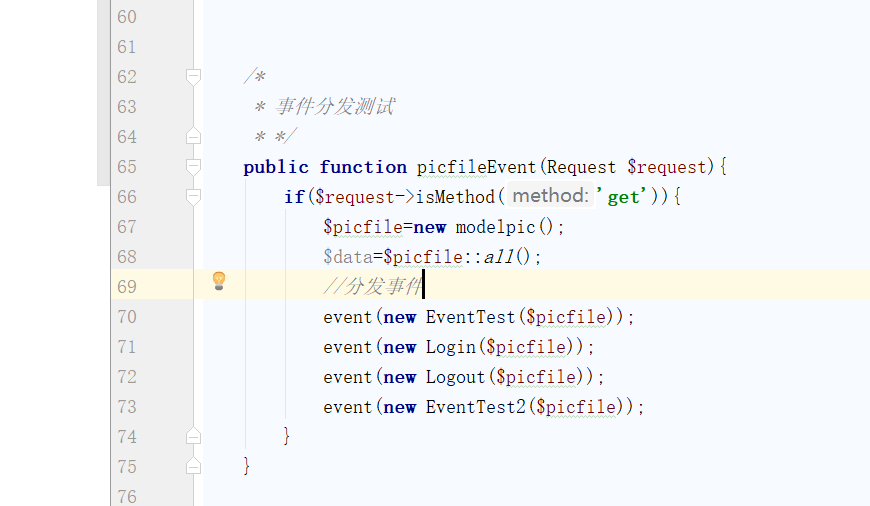


**事件监听器队列**

操作之前要配置好队列

**事件分发**

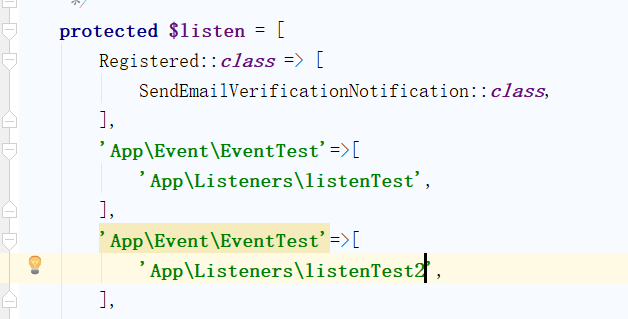
即什么时候触发监听器



**事件订阅者**

很多时候同一事件会被不同监听器监听

像这样



这样会让我们多生成许多监听器功能差不多

因此我们是有事件订阅者，用来在一个监听器中监听器中区分不同事件

**定义订阅类：**

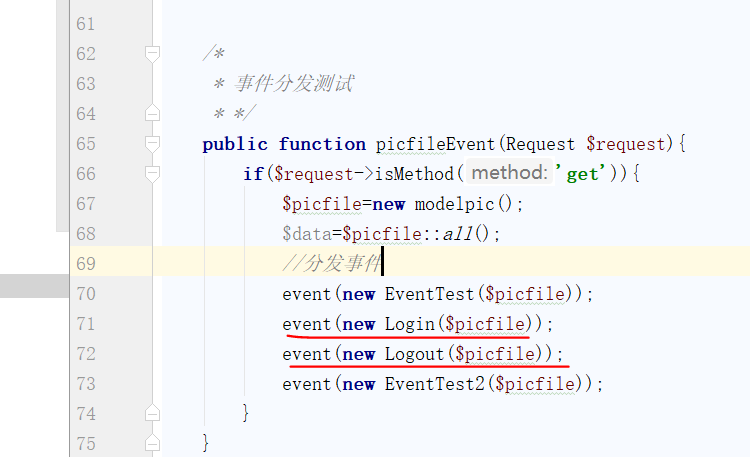
**<?php**/\*\*  
 \* Created by PhpStorm.  
 \* User: SWESWE  
 \* Date: 2020/2/19  
 \* Time: 10:40  
 \*/  
  
**namespace** App\Listeners;  
  
  
**class** EventSubscriber  
{  
 /\*  
 \* 事件订阅者  
 \* \*/  
 /\*\*  
 \* 处理用户登录事件.  
 \* **@translator** laravelacademy.org  
 \*/  
 **public function** onUserLogin($event) {  
 **echo"事件订阅者登录"**;  
 **echo'<br>'**;  
  
 }  
 /\*\*  
 \* 处理用户退出事件.  
 \*/  
 **public function** onUserLogout($event) {  
 **echo"事件订阅者登出"**;  
 **echo'<br>'**;  
 }  
  
 /\*\*  
 \* 为订阅者注册监听器.  
 \*直接用方法监听事件不再需要新建一个监听器类来监听  
 \* **@param** \*/  
 **public function** subscribe($events)  
 {  
 $events->listen(  
 **'App\Event\Login'**,  
 **'App\Listeners\EventSubscriber@onUserLogin'** );  
  
 $events->listen(  
 **'App\Event\Logout'**,  
 **'App\Listeners\EventSubscriber@onUserLogout'** );  
 }



**注册订阅者**

  
  
}

**分发事件：**



## 广播

### Websocket：

WebSocket 是一种双向通信协议，客户端和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。

http只能客户端发起连接

Socket其实并不是一个协议，而是为了方便使用TCP或UDP而抽象出来的一层，是位于应用层和传输控制层之间的一组接口。

官方文档：

<https://xueyuanjun.com/post/19505.html>

<https://qianjinyike.com/laravel-%E5%86%85%E7%BD%AE%E8%81%8A%E5%A4%A9%E5%AE%A4/>

应用场景

实时聊天

用户不用刷新页面就可以更新其他用户的评论

**广播配置文件**

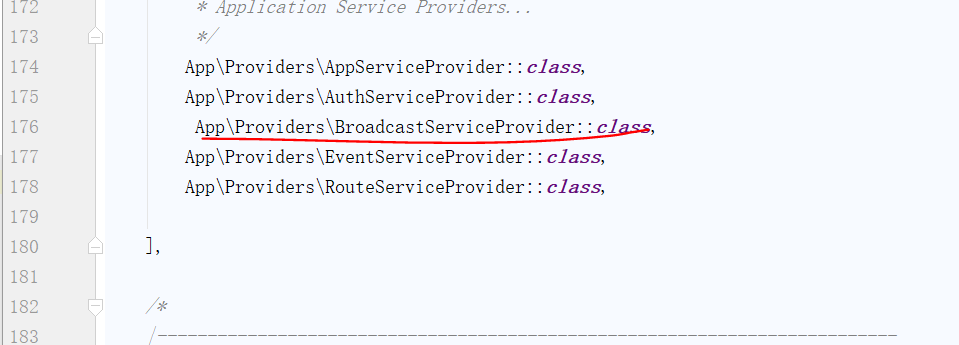
**流程参考：**

<https://qianjinyike.com/laravel-%E5%85%AC%E6%9C%89%E5%B9%BF%E6%92%AD/>

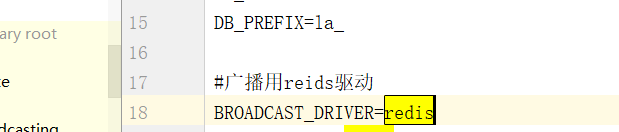
config/broadcasting.php

Laravel 开箱支持多种广播驱动：[Pusher](https://pusher.com/)、[Redis](https://xueyuanjun.com/post/19525.html)以及一个服务于本地开发和调试的 log 驱动。此外，还提供了一个 null 驱动用于完全禁止事件广播

### 1.去掉注释 config/app.php provider 中的BroadcastServiceProvider



2.在 .env 中配置驱动 BROADCAST\_DRIVER=redis



### 安装redis

安装广播之前要安装redis

参考：菜鸟教程windwos 下安装redis

<https://www.runoob.com/redis/redis-install.html>

运行命令：

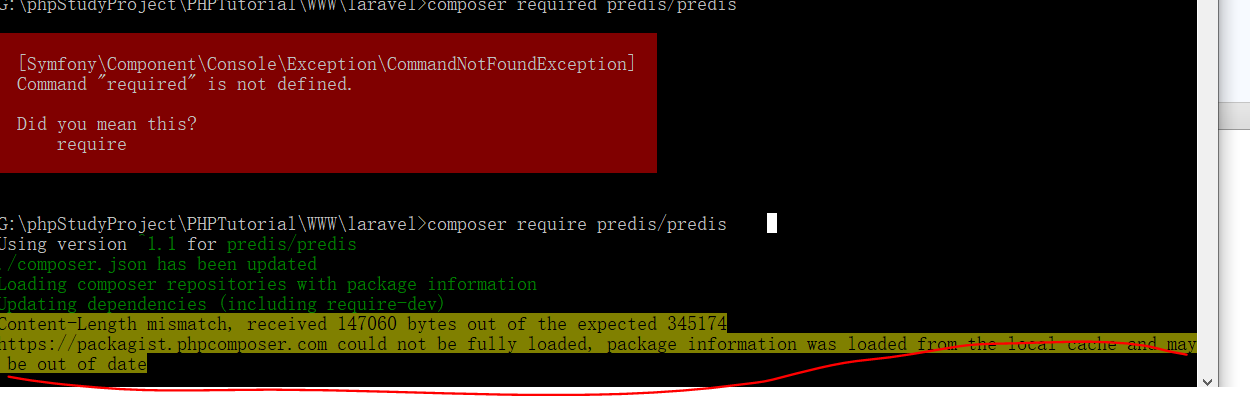
切到redis目录下

redis-server.exe redis.windows.conf

windows安装redis桌面应用 redis desktop manager

composer required predis/predis//composer安装有问题安装不了

出现下面问题

  
运行composer config -g repo.packagist composer <https://packagist.phpcomposer.com>后再次运行上面命令

在开始介绍广播事件之前，还需要配置并运行一个[队列监听器](https://xueyuanjun.com/post/19516.html)。所有事件广播都通过队列任

### **初始化 websocket 客户端**

import Echo from "laravel-echo"

window.io = require('socket.io-client');

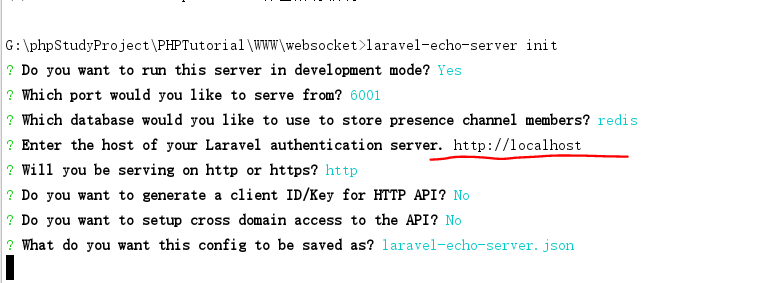
window.Echo = new Echo({

broadcaster: 'socket.io',

host: window.location.hostname + ':6001'

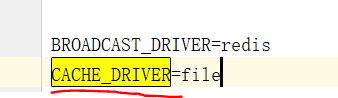
});

创建事件



画红线的要配成自己的域名

配置公有频道：

画线的可以配redis也可以配file,但BROADCAST\_DRIVER

要配成redis驱动

npm install --save socket.io-client | echo 'websocket 客户端'

npm install --save laravel-echo | echo 'websocket 客户端封装'

npm install -g laravel-echo-server | echo 'websocket 服务端'

npm install | echo '安装所有其他依赖'

下面三个严格执行

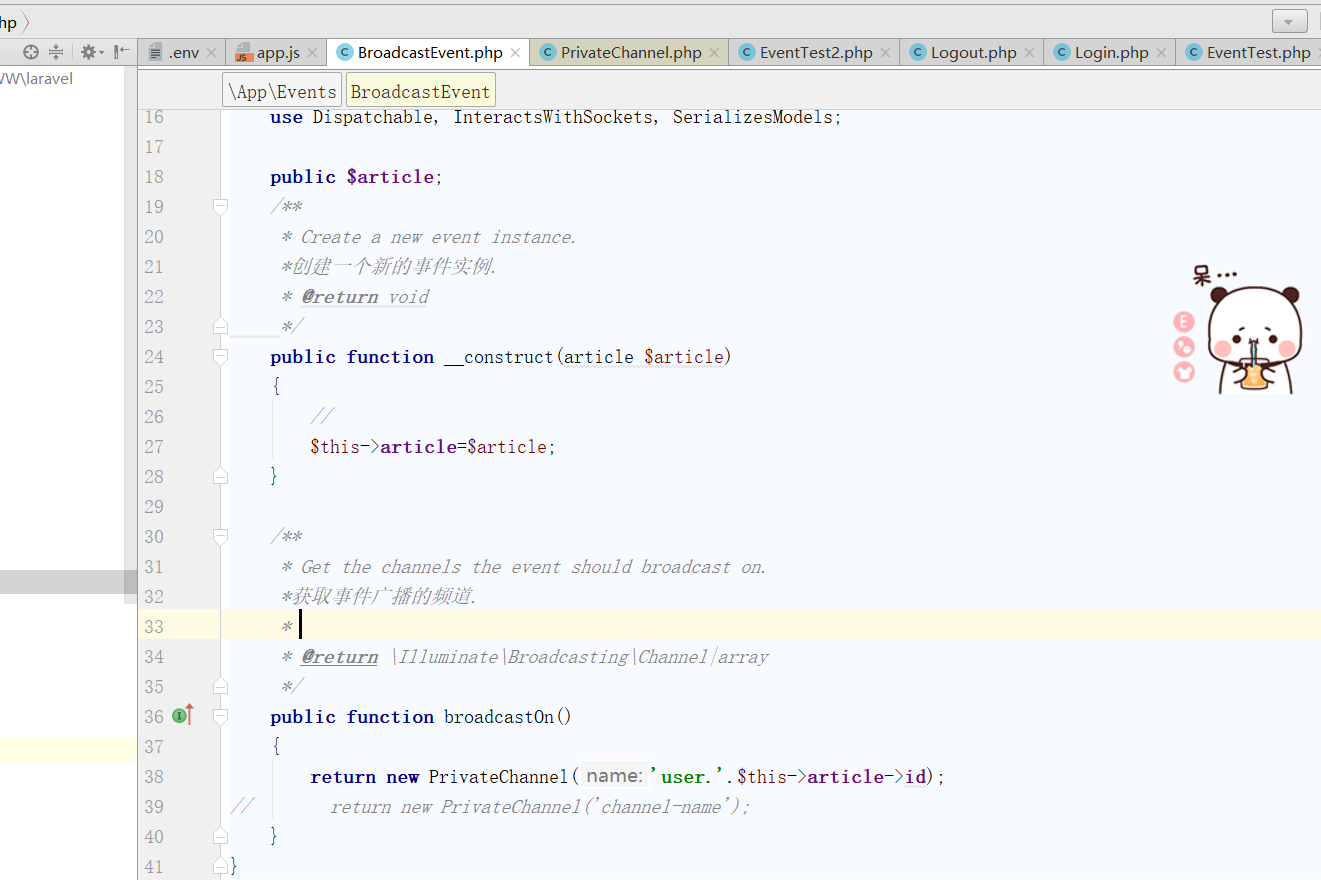
npm run watch | echo '监控文件变化编译前端资源'

laravel-echo-server init | echo '初始化 websocket 服务端'

laravel-echo-server start | echo '启动 websocket 服务端'

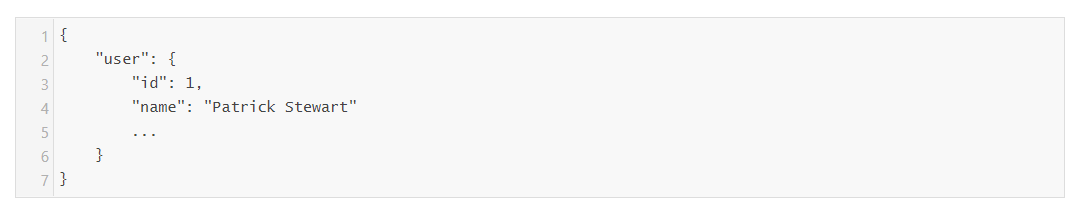
务来完成以便应用的响应时间不受影响。

### 3.定义广播事件

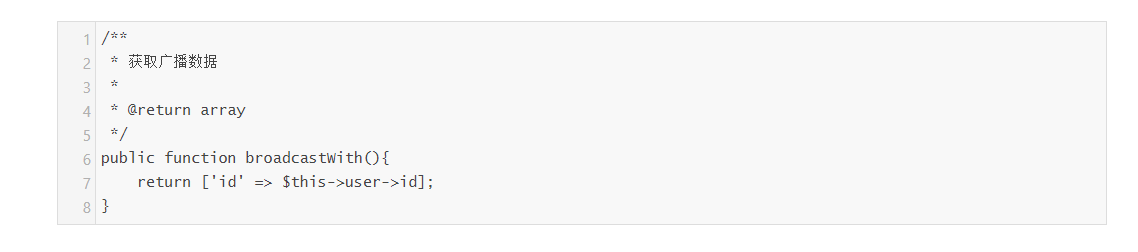


#### 广播数据

如果某个事件被广播，其所有的 public 属性都会按照事件负载（payload）自动序列化和广播，从而允许你从 JavaScript 中访问所有 public 数据，举个例子，如果你的事件有一个单独的包含 Eloquent 模型的 $user 属性，广播负载定义如下：

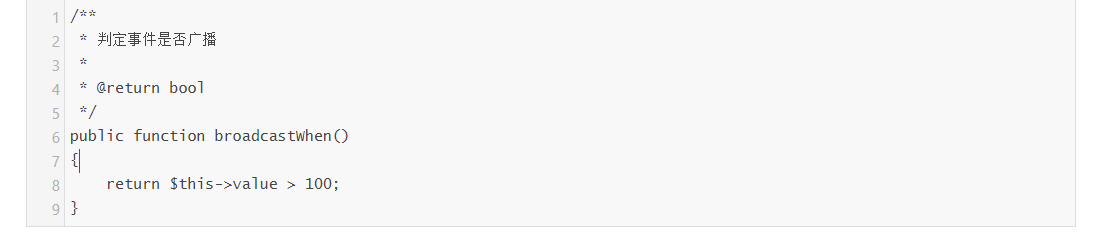


不过，如果你希望对广播负载有更加细粒度的控制，可以添加 broadcastWith 方法到事件，该方法会返回你想要通过事件广播的数组数据：



#### 广播条件

有时候你想要在指定条件为 true 的前提下才广播事件，可以通过添加 broadcastWhen 方法到事件类来定义这些条件：



Controller—分发事件—》

事件广播类中绑定频道

在routes/channels.php 中授权频道

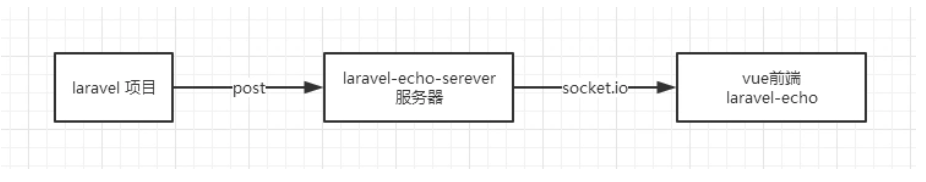
Broadcast::channel('order.{$orderId}', function ($user, $orderId) {

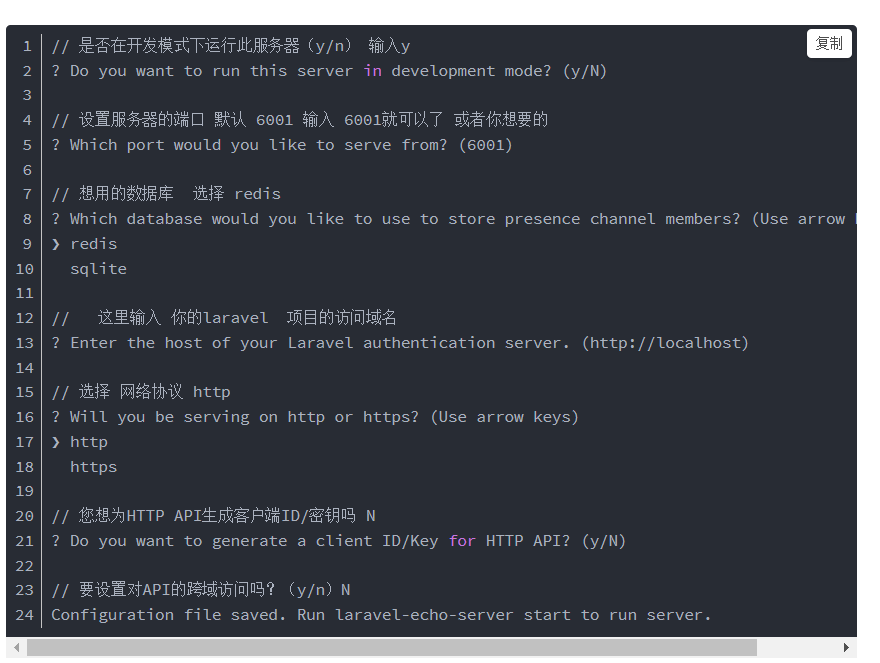
return $user->id === Order::findOrNew($orderId)->user\_id;

});

使用 Laravel Echo 来监听事件广播

**整体架构：**





**Phpstorm快捷键**

CTRL + SHIFT + N 文件名查找

调试数据库中的数据。通过使用 Laravel artisan 内建的 php artisan [tinker](https://link.jianshu.com/?t=http://laravelacademy.org/tags/tinker)

## 三．广播

### 3.1公共频道

#### 配置：

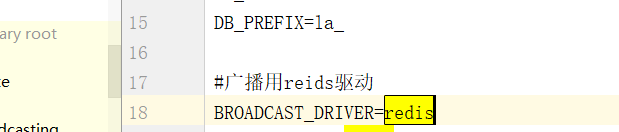
##### 配置广播驱动

//安装redis

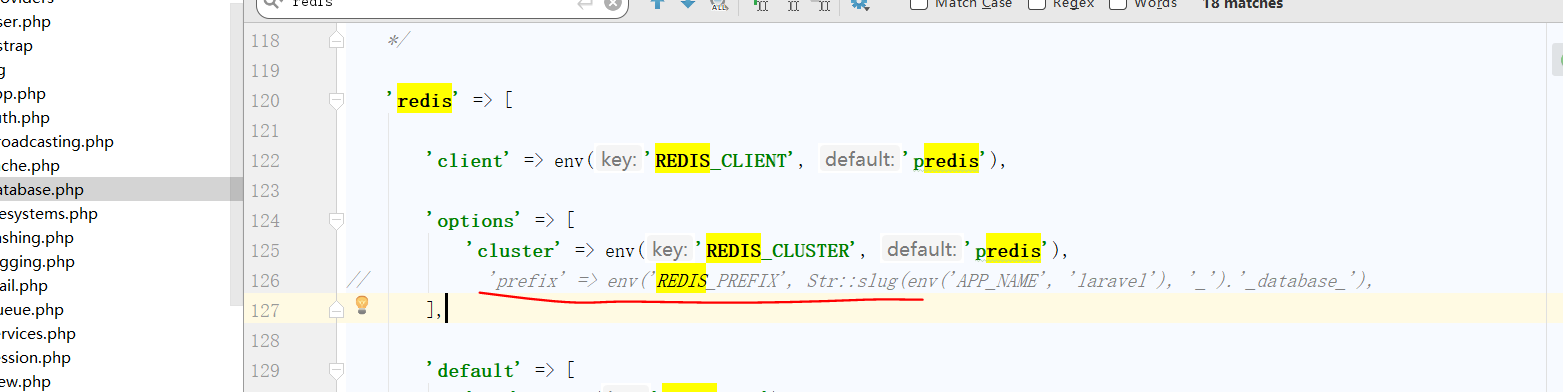
#### composer require predis/predis

必须安装

在 .env 中配置驱动 BROADCAST\_DRIVER=redis



取消redis前缀 config/database



否则会在频道前加上前缀



##### 2．安装

###### 2.1安装redis

安装广播之前要安装redis

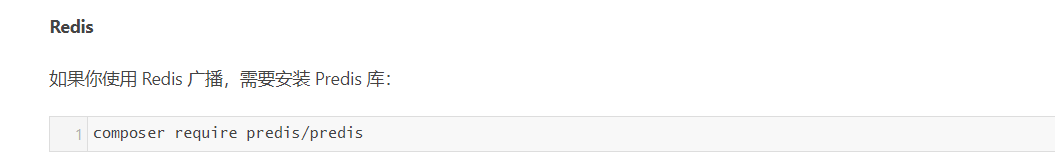
参考：菜鸟教程windwos 下安装redis

<https://www.runoob.com/redis/redis-install.html>

运行命令：

切到redis目录下

redis-server.exe redis.windows.conf

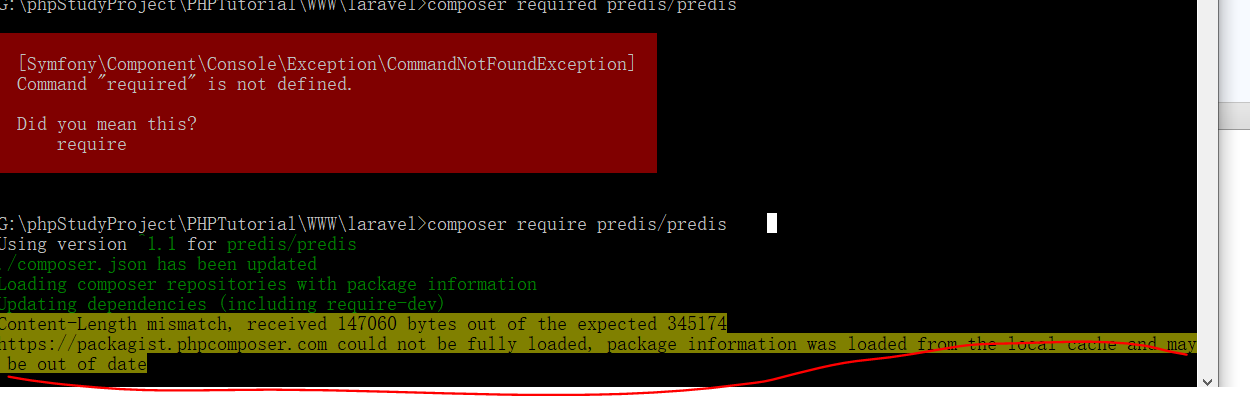


windows安装redis桌面应用 redis desktop manager

composer dumpautoload

composer require predis/predis//composer安装有问题安装不了

出现下面问题

  
运行composer config -g repo.packagist composer <https://packagist.phpcomposer.com>后再次运行上面命令composer update

composer config -l

###### 2.2npm安装

npm install --save socket.io-client | echo 'websocket 客户端'

npm install --save laravel-echo | echo 'websocket 客户端封装'

npm install -g laravel-echo-server | echo 'websocket 服务端'

npm install | echo '安装所有其他依赖'

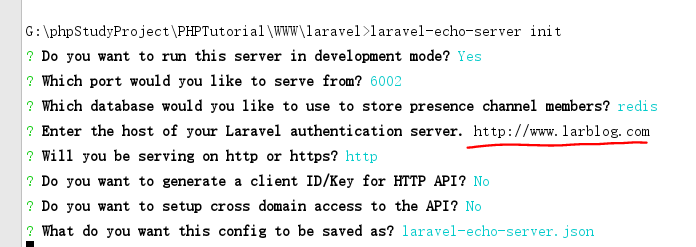
###### 2.3运行

npm run watch | echo '监控文件变化编译前端资源'

laravel-echo-server init | echo '初始化 websocket 服务端'

laravel-echo-server start | echo '启动 websocket 服务端'

运行laravel-echo-server init 的时候



画线部分一般为自己域名

3.定义事件

Php artisan make:event free

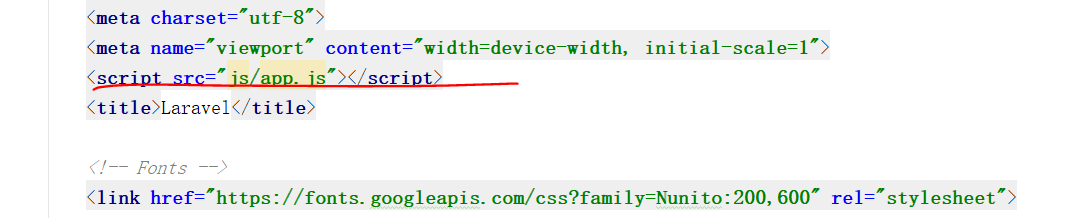
**<?php  
  
namespace** App\Events;  
  
**use** Illuminate\Broadcasting\Channel;  
**use** Illuminate\Queue\SerializesModels;  
**use** Illuminate\Broadcasting\PrivateChannel;  
**use** Illuminate\Broadcasting\PresenceChannel;  
**use** Illuminate\Foundation\Events\Dispatchable;  
**use** Illuminate\Broadcasting\InteractsWithSockets;  
**use** Illuminate\Contracts\Broadcasting\ShouldBroadcast;  
  
**class** free **implements** ShouldBroadcast *// 1. 事件是要广播出去的,*这边一定要**继承**接口

patchable, InteractsWithSockets, SerializesModels;  
 **public $msg**;  
 */\*\*  
 \* Create a new event instance.  
 \*  
 \** ***@return*** *void  
 \*/* **public function** \_\_construct($msg) *// 2. 广播出去的内容* {  
 *//* $this->**msg**=$msg;  
 }  
  
 */\*\*  
 \* Get the channels the event should broadcast on.  
 \*  
 \** ***@return*** *\Illuminate\Broadcasting\Channel|array  
 \*/* **public function** broadcastOn() *// 3. 对哪些频道进行广播* {  
*// return new PrivateChannel('channel-name');  
// return [new Channel('countryside'), new Channel('a')];* **return new** Channel(**'countryside'**);  
 }  
  
}

##### 定义客户端监听广播



并在welcome页面中引入



注销redis前缀否则会在频道前加上前缀



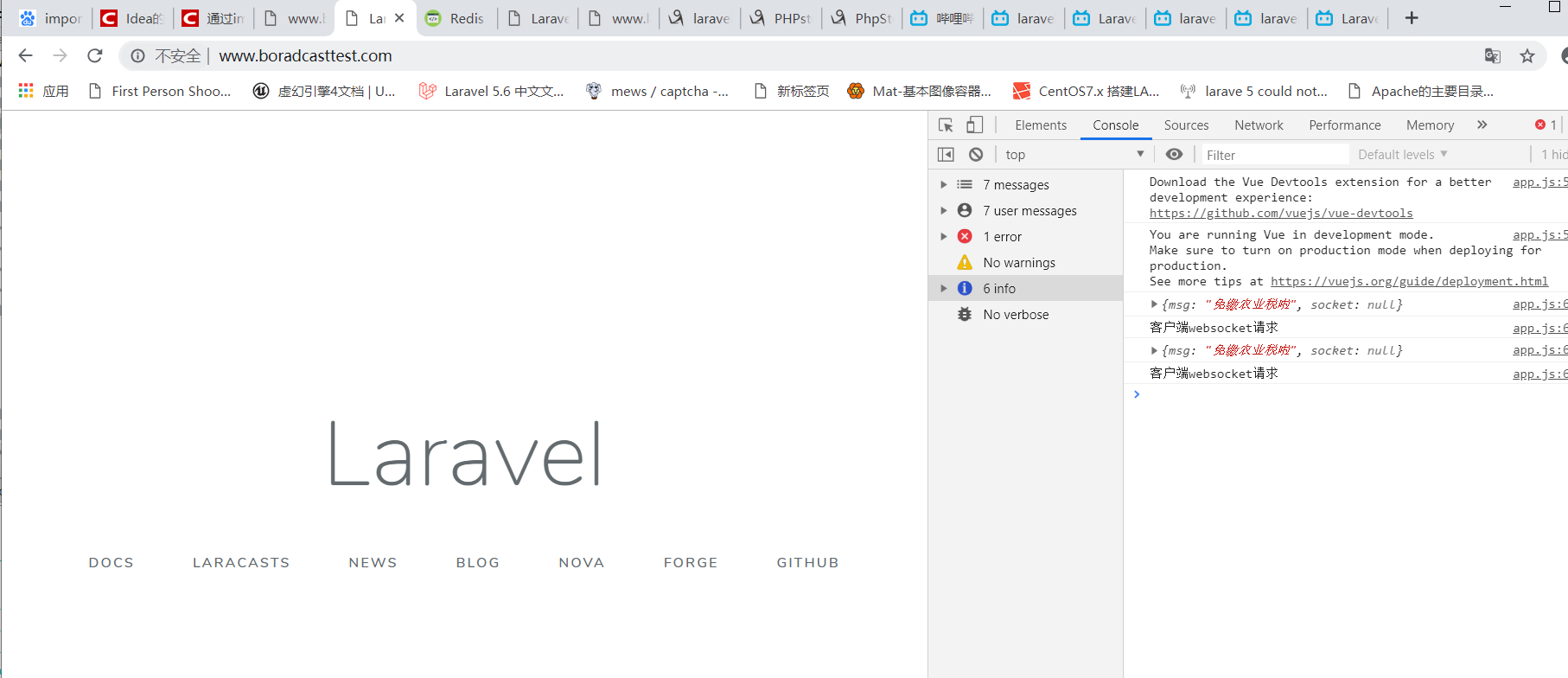
##### 定义路由

Route::*get*(**'/event'**, **function** () {  
 event(**new** \App\Events\free(**'免缴农业税啦'**));  
});

##### 测试

访问<http://www.boradcasttest.com/event>路由

再访问<http://www.boradcasttest.com> 发现控制台打印出消息

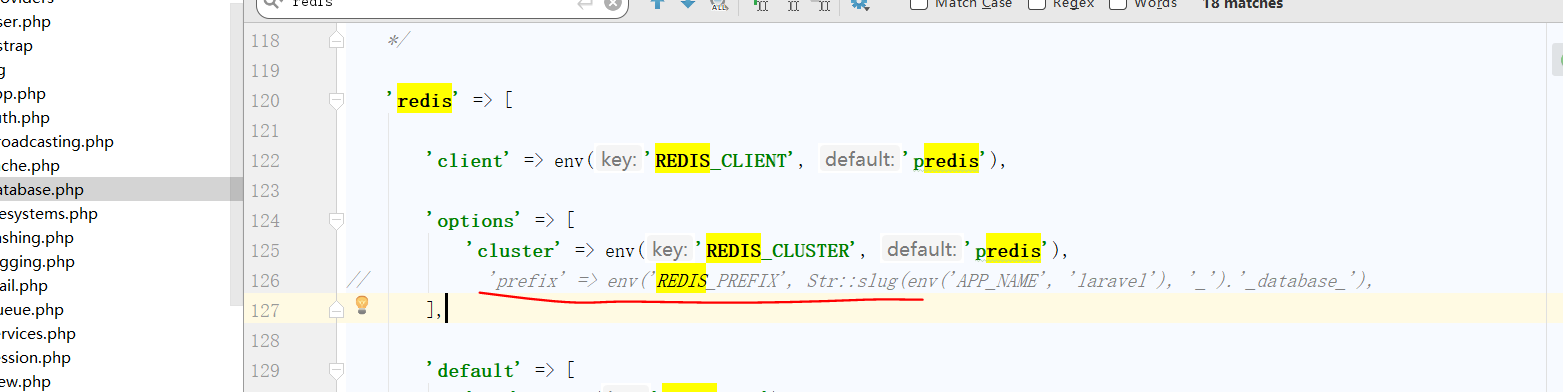


### 3.2私有频道

#### 1. 配置

在config/app/php中取消BroadcastServiceProvider注释，判断用户是否有权限监听我的的频道，用来注册广播授权路由和闭包在 .env 中配置驱动 BROADCAST\_DRIVER=redis

取消redis前缀

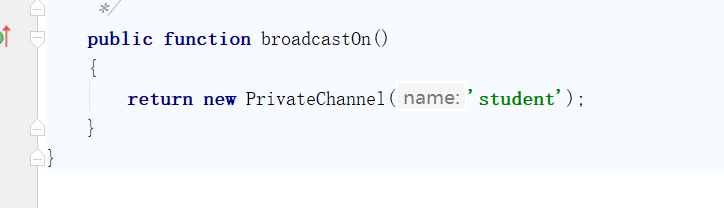


否则会在频道前加上前缀



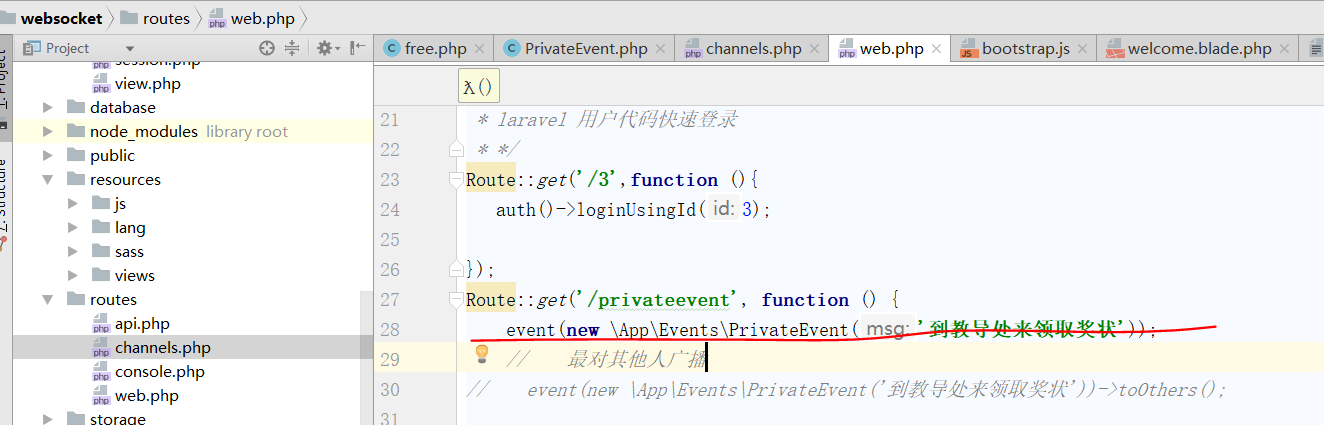
#### 安装 同公有频道一致

#### 定义事件

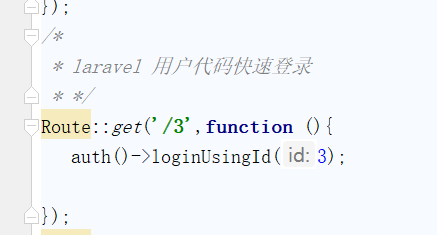


#### 4.设置频道权限

Route/channel



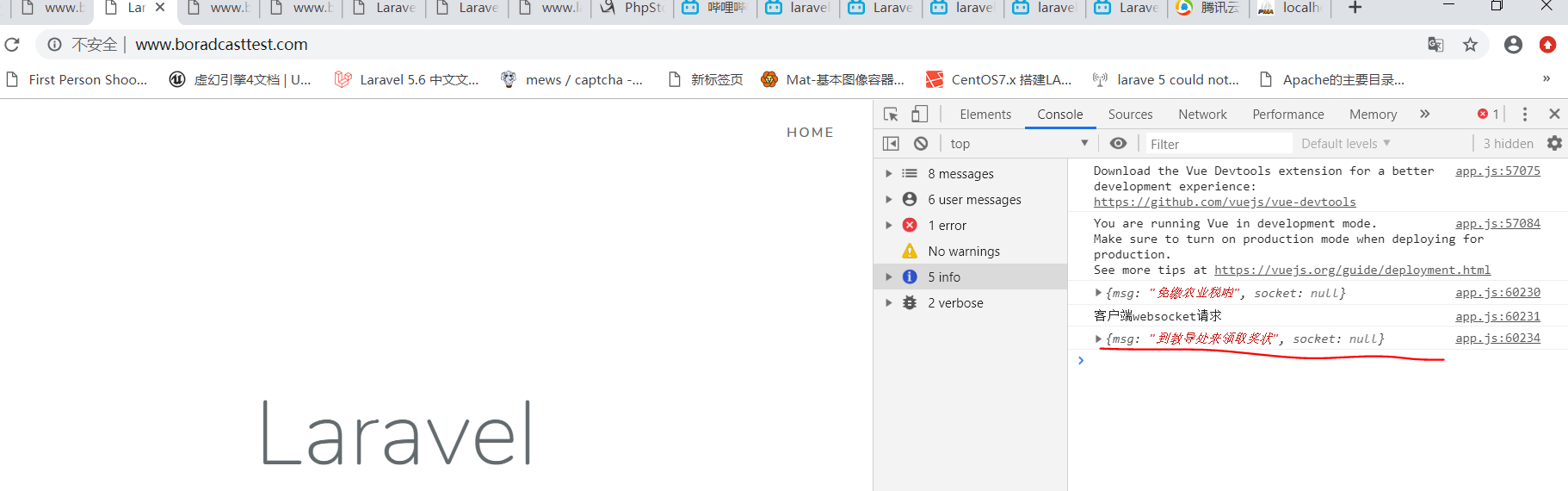
#### 5.模拟用户登录



#### 6.测试

输入<http://www.boradcasttest.com/3>

<http://www.boradcasttest.com/privateevent>



### 3.3实时聊天

<https://learnku.com/articles/36574>

composer config --global cache-files-maxsize 5000MiB

## 四．Laravel数据库操作

### 3.1查询构建器

## Linux广播环境搭建：

### Linuxan安装composer

<https://www.cnblogs.com/sz-xioabai/p/11548573.html>

1. 下载composer

curl -sS https://getcomposer.org/installer | php

1. 将composer.phar文件移动到bin目录以便全局使用composer命令

mv composer.phar /usr/local/bin/composer

3.切换国内源

composer config -g repo.packagist composer https://packagist.phpcomposer.com

### 安装predis

composer require predis/predis

### linux安装node.js

<https://www.cnblogs.com/sirdong/p/11447739.html>

### linux安装redis

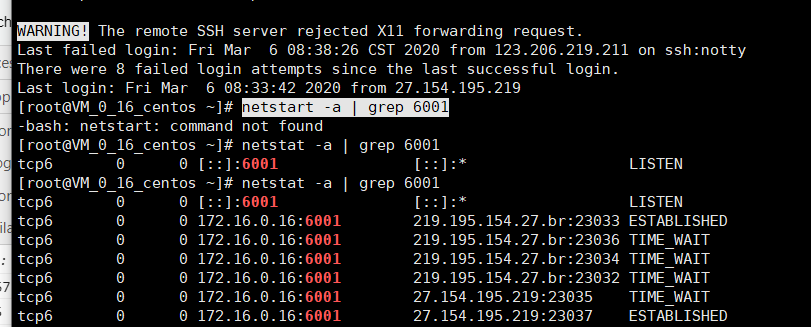
redis没有设置开机启动，服务器重启要再次启动

<https://www.runoob.com/redis/redis-install.html>

Laravel广播在服务器上搭建要开启相应的端口 6001



netstart -a | grep 6001 查看端口是否开启

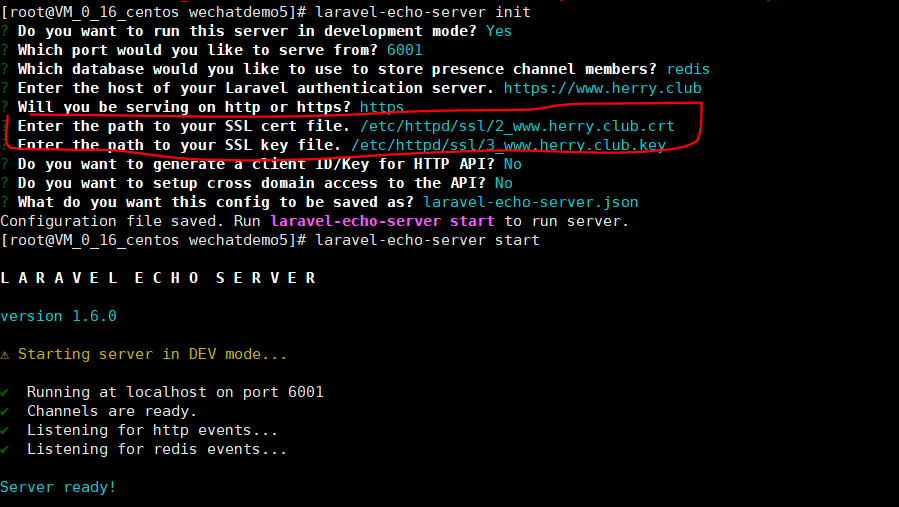


安装 redis ，node.js ,composer,其他安装和windows一样

Npm run watch 要在项目路径下启用

Laravel-echo-server init :

选择https要配置证路径



服务器上 要让laravel-echo-server 一直运行

参考：

<https://segmentfault.com/a/1190000019969184?utm_source=tag-newest>

<https://www.cnblogs.com/quzq/p/11192993.html>

<https://blog.csdn.net/donggege214/article/details/80264811>

### 安装supervisor

yum install supervisor

### 设置开机自启动

编辑服务文件

# vim /usr/lib/systemd/system/supervisord.service

1

内容如下

[Unit]

Description=Supervisor daemon

[Service]

Type=forking

PIDFile=/var/run/supervisord.pid

ExecStart=/bin/supervisord -c /etc/supervisord.conf

ExecStop=/bin/supervisorctl shutdown

ExecReload=/bin/supervisorctl reload

KillMode=process

Restart=on-failure

RestartSec=42s

[Install]

WantedBy=multi-user.target

保存退出

Esc:x

1

启动服务

# systemctl enable supervisord

### 配置一个进程

编辑/etc/supervisord.conf文件

[program:laravel-echo-server]

process\_name=%(program\_name)s

directory=/home/www/wechatdemo5

command=laravel-echo-server start

autostart=true

autorestart=true

user=root

numprocs=1

redirect\_stderr=true

stdout\_logfile=/var/log/supervisor/laravel-echo-server.log

日志文件laravel-echo-server.log要自己建

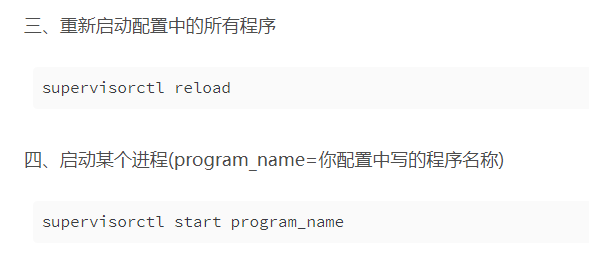
### 启动：

supervisord -c /etc/supervisord.conf

或者输入supervisord      启动

关闭所有

supervisorctl shutdown



supervisorctl reload

supervisorctl start laravel-echo-server

基本命令<https://www.jianshu.com/p/c8071234f081>

监听某个端口

lsof -i:6001

一、开启命令

1. supervisor的服务器端部分启动命令：

* sudo unlink /var/run/supervisor.sock
* supervisord -c /etc/supervisor/supervisord.conf  
  此时默认开启了所有服务

1. supervisor的客户端部分命令：

* supervisorctl status 查看进程运行状态
* supervisorctl start 进程名 启动进程
* supervisorctl stop 进程名 关闭进程
* supervisorctl restart 进程名 重启进程
* supervisorctl update 重新载入配置文件
* supervisorctl shutdown 关闭supervisord
* supervisorctl clear 进程名 清空进程日志
* supervisorctl 进入到交互模式下。使用help查看所有命令。
* start stop restart + all 表示启动，关闭，重启所有进程。

二、关闭命令

supervisorctl stop all先关闭supervisor服务  
之后再关闭supervisord服务  
kill -9 pid

## 五．Laravel认证

### 认证与授权

参考：

<https://www.cnblogs.com/jinhengyu/p/10257792.html>

认证就是认证用户的身份，像飞机票用你的身份证验证你是不是你自己

授权就是系统成功验证你的身份后。最终会授予你访问资源的权限，

简单的说就是你认证成功后系统授予你哪些权限。

我们知道，HTTP是无状态的，所以，当我们需要获得用户是否在登录的状态时，我们需要检查用户的登录状态，一般来说，用户的登录成功后，服务器会发一个登录凭证（又被叫作Token），就像你去访问某个公司，在前台被认证过合法后，这个公司的前台会给你的一个访客卡一样，之后，你在这个公司内去到哪都用这个访客卡来开门，而不再校验你是哪一个人。在计算机的世界里，这个登录凭证的相关数据会放在两种地方，一个地方在用户端，以Cookie的方式（一般不会放在浏览器的Local Storage，因为这很容易出现登录凭证被XSS攻击），另一个地方是放在服务器端，又叫Session的方式（SessonID存于Cookie

Auth(认证)

认证的方式有很多种，常见的认证方式有：

### 常见的认证方式

#### http basic auth

这是传统的认证方式使用username和password来进行登录。它实现的原理是

1. 把username和password做成username:password这样即用表单分隔。得到一个字符串s
2. 使用base64编码，将字符串进行base64编码
3. 将base64编码后的字符串放到http的头[Authorization](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization" \t "_blank) 字段中，形成 Authorization: Basic aGFvZW86Y29vbHNoZWxsCg，然后发送到服务端。
4. 服务端在http头中没有看到认证字段返回错误否则返回200

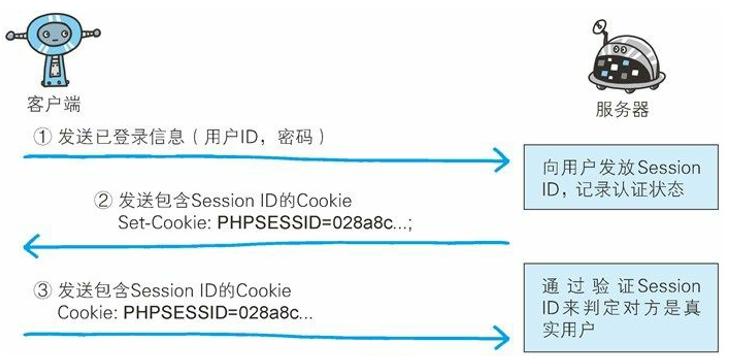
采用base64编码，放在请求中，很容易被破解不是很安全

#### HTTP 表单认证

这种认证方式也是我们常用的认证方式。

基于表单认证一般会使用Cookie 来管理 Session（会话）

基于表单认证本身是通过服务器端的 Web 应用，将客户端发送过来的用户 ID 和密码进行校验。但鉴于 HTTP 是无状态协议，之前已认证成功的用户状态无法通过协议层面保存下来。即无法实现状态管理，因此即使当该用户下一次继续访问，也无法区分他与其他的用户。于是我们会使用 Cookie 来管理 Session，以弥补 HTTP 协议中不存在的状态管理功能。



还有像

JWT – JSON Web Tokens认证和oauth2.0认证。

#### JWT

参考：<https://www.jianshu.com/p/576dbf44b2ae>

##### 什么是JWT

Json web token 是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准（[(RFC 7519](https://link.jianshu.com?t=https:/tools.ietf.org/html/rfc7519" \t "_blank)).该token被设计为紧凑且安全的，特别适用于分布式站点的单点登录（SSO）场景。JWT的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源，也可以增加一些额外的其它业务逻辑所必须的声明信息，该token也可直接被用于认证，也可被加密。

##### 基于token机制的认证流程

用户使用用户名密码来请求服务器

服务器进行验证用户的信息

服务器通过验证发送给用户一个token

客户端存储token，并在每次请求时附送上这个token值

服务端验证token值，并返回数据

##### jwt的结构

###### 头部(header)

承载两个部分信息，声明类型，声明加密算法

{

'typ': 'JWT',

'alg': 'HS256'

}

将头部进行base64加密构成第一部分

###### 载荷(payload)

存放有效信息包括三部分

* 标准中注册的声明
* 公共的声明
* 私有的声明

**标准中注册的声明** (建议但不强制使用) ：

* **iss**: jwt签发者
* **sub**: jwt所面向的用户
* **aud**: 接收jwt的一方
* **exp**: jwt的过期时间，这个过期时间必须要大于签发时间
* **nbf**: 定义在什么时间之前，该jwt都是不可用的.
* **iat**: jwt的签发时间
* **jti**: jwt的唯一身份标识，主要用来作为一次性token,从而回避重放攻击。

**公共的声明** ：  
公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息.但不建议添加敏感信息，因为该部分在客户端可解密.

**私有的声明** ：  
私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为base64是对称解密的，意味着该部分信息可以归类为明文信息。

定义一个payload:

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

然后将其进行base64加密，得到Jwt的第二部分。

###### 签证(signature)

wt的第三部分是一个签证信息，这个签证信息由三部分组成：

* header (base64后的)
* payload (base64后的)
* secret

这个部分需要base64加密后的header和base64加密后的payload使用.连接组成的字符串，然后通过header中声明的加密方式进行加盐secret组合加密，然后就构成了jwt的第三部分。

// javascript

var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(payload);

var signature = HMACSHA256(encodedString, 'secret'); // TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

将这三部分用.连接成一个完整的字符串,构成了最终的jwt:

###### 如何应用

一般是在请求头里加入Authorization，并加上Bearer标注：

fetch('api/user/1', {

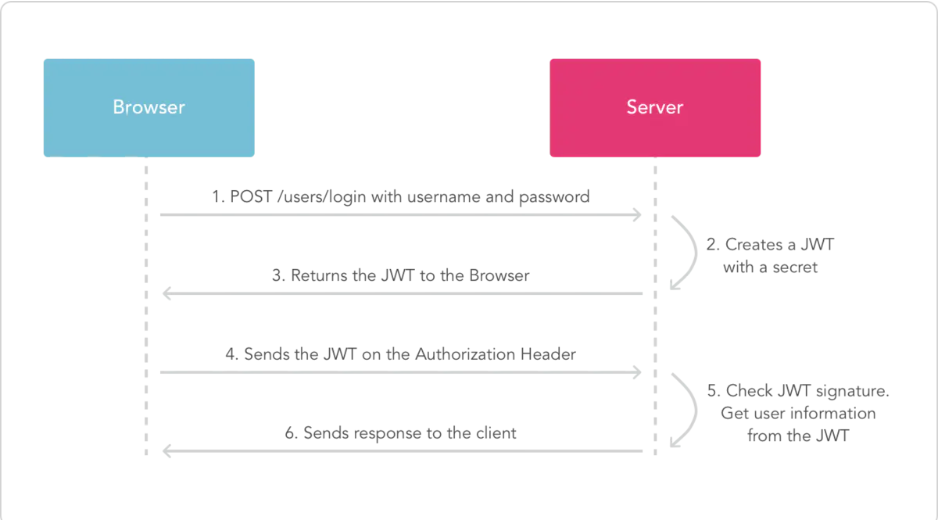
headers: {

'Authorization': 'Bearer ' + token

}

})

###### 认证流程



官网：<https://jwt.io/>

里面有各种语言的jwt,用composer安装

###### 用户登录成功，获取到token，再次访问登录接口会怎样？

#### OAuth2.0协议

oAuth2.0协议(开放授权)是一个开放标准，允许用户授权给第三方用户访问存储在服务器上的资源，并且用户不需要向第三方提供账号密码。

##### 应用场景：

原生app授权：app登录请求后台接口，为了安全认证，所有请求都带token信息，如果登录验证、请求后台数据。

前后端分离单页面应用（spa）：前后端分离框架，前端请求后台数据，需要进行oauth2安全认证，比如使用vue、react后者h5开发的app。

涉及角色：

* Resource Owner（资源拥有者：用户）
* Client （第三方接入平台：请求者）
* Resource Server （服务器资源：数据中心）
* Authorization Server （认证服务器）

##### 实现流程

（A）用户打开客户端以后，客户端要求用户给予授权。  
（B）用户同意给予客户端授权。  
（C）客户端使用上一步获得的授权，向认证服务器申请令牌。  
（D）认证服务器对客户端进行认证以后，确认无误，同意发放令牌。  
（E）客户端使用令牌，向资源服务器申请获取资源。  
（F）资源服务器确认令牌无误，同意向客户端开放资源。

##### 授权模式

参考：<https://www.jianshu.com/p/84a4b4a1e833>

<https://www.jianshu.com/p/84a4b4a1e833>

中的授权模式

 授权码模式（authorization code）

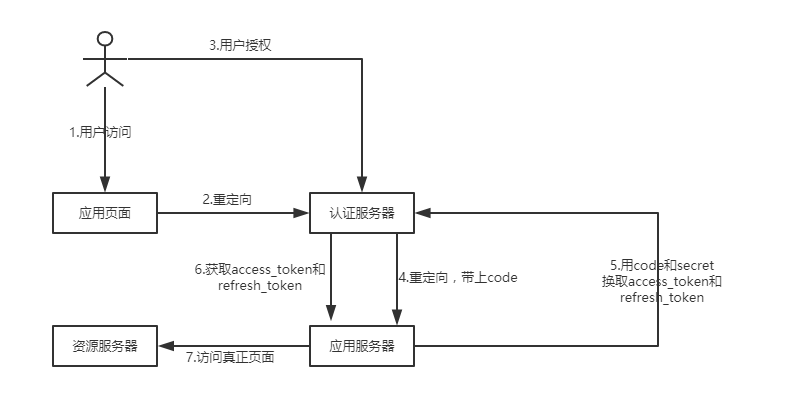
 简化模式（implicit） （适用于单页面应用）

 密码模式（resource owner password credentials）

 客户端模式（client credentials）

###### 授权码模式

流程：



用户访问客户端，重定向到认证服务器，

认证服务器向用户展示授权页面等待用户授权，

用户授权后认证服务器生成 code和client\_id并发送给服务器

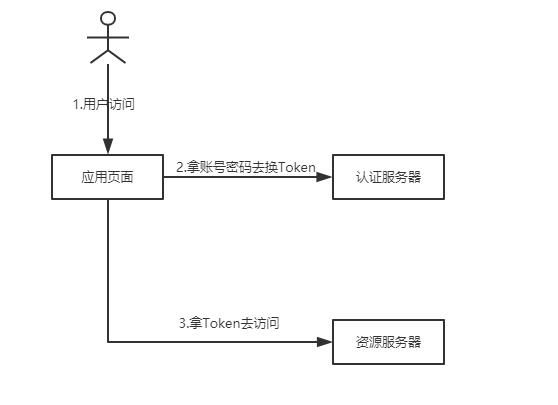
服务器拿到client\_id去后台查询相应的client\_secret

用code和client\_secret到认证服务器换区access\_token和refresh\_token

客户端发送访问令牌（access token）和更新令牌（refresh token）

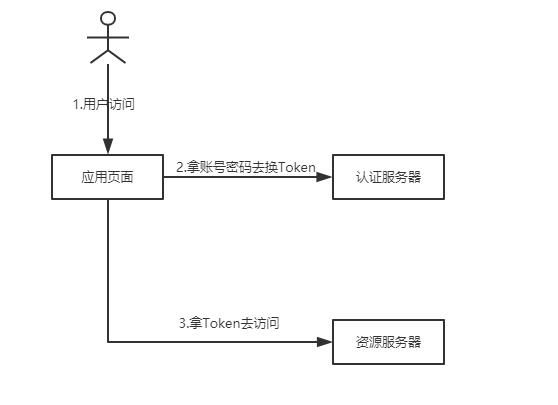
客户端携带令牌访问相关资源

###### 简化模式



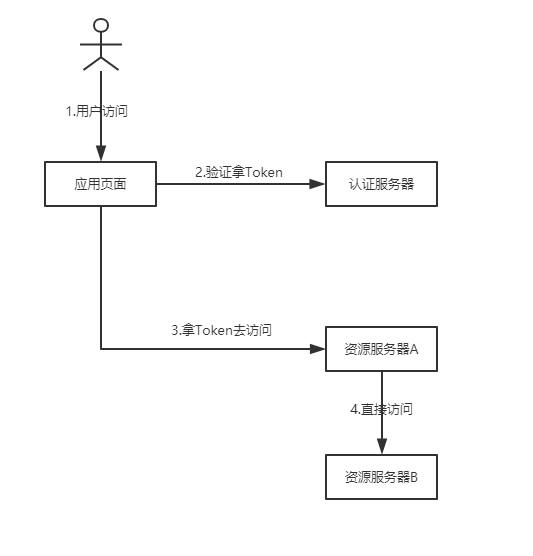
* 第一步：用户访问页面时，重定向到认证服务器。
* 第二步：认证服务器给用户一个认证页面，等待用户授权。
* 第三步：用户授权，认证服务器想应用页面返回Token
* 第四步：验证Token，访问真正的资源页面

###### 密码模式



* 第一步：用户访问用页面时，输入第三方认证所需要的信息(QQ/微信账号密码)
* 第二步：应用页面那种这个信息去认证服务器授权
* 第三步：认证服务器授权通过，拿到token，访问真正的资源页面

###### 客户端凭证模式



* 第一步：用户访问应用客户端
* 第二步：通过客户端定义的验证方法，拿到token，无需授权
* 第三步：访问资源服务器A
* 第四步：拿到一次token就可以畅通无阻的访问其他的资源页面

### Laravel内置的登录认证

### laravel内置的API认证

默认情况下laravel为每个用户分配一个随机令牌，在配置文件config/auth.app中默认定义了一个api守卫(guard),对应的驱动为token。该驱动任务是检查每个请求上的API令牌并验证是否与分配给用户的令牌值相同（分配给用户的令牌一般存储在数据库中）。

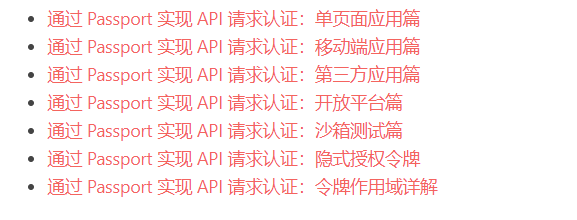
**认证过程：**

1. 注册：用户注册成功后，随机生成长字符串作为 token，原生 token 返回给用户。哈希后的 token 存到数据库里。
2. 登陆：用户使用账号密码登陆成功，随机生成长字符串作为 token，原生 token 返回给用户。哈希后的 token 存到数据库里。
3. 认证：将用户传来的 token 进行哈希，然后去数据库中查找哈希后的 token ，找到了就认证成功，否则失败。

### 基于 [Laravel Passport](https://xueyuanjun.com/post/19485.html)实现的 API 认证

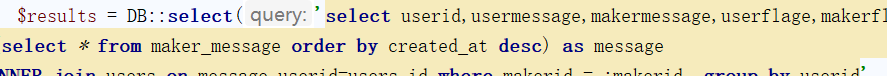
项目：apitestindex

参考：<https://xueyuanjun.com/post/19485.html>



## 六．数据操作

原生语句查询结果集



select 方法以数组的形式返回结果集，数组中的每一个结果都是一个 PHP stdClass 对象：因此使用$results[0][‘id’]是无法访问的

因此我们进行数据转换

**foreach**($results **as** $k=>$v) {  
 $results[$k] = (array)$v;  
}

将results转成一个数组。

### Eloqunet orm

Eloquent orm 不支持批量插入

## 七．队列

项目：websocket

测试时使用redis队列，任务是写入日志，cmd redis界面界面报rdbSave failed in qfork: Permission denied 错误，给redis下的redis-cli.pdb写入权限即可。

参考：<https://blog.csdn.net/u010049198/article/details/89715009>

### 我为什么要用队列

用到队列是由于系统功能的需要，在毕设中需要完成卖家发货后到一定时间自动收货的功能。

像发邮件这样的也是。

### 什么是队列

队列这个词并不陌生，在常用数据结构中队列就是其中一种，他它的特点就是先进先出像排队一样。

我个人的理解就是你有一堆任务要完成，但是考虑到系统的性能或者具体的功能实现比如自动收货，我们将这些任务放到队列中，系统会根据自动完成队列中的任务。

队列的目的是将耗时的任务延时处理

，队列就是为了在某种程度上替代多线程而设计的一种处理并发的方式，同时，也就具备天生的秉性：异步！用于处理耗时的工作，比如一个流程走到一个地方，要发送一封邮件通知，那我不能让整个程序等到邮件发完了再继续下去，那我就把发邮件的工作丢到一个队列中去，让这个工作慢慢做，我继续处理响应的逻辑。

Laravel实现队列的过程原理：

### 7.1驱动

-在文档中laravel有数据库、[Beanstalkd](https://kr.github.io/beanstalkd/)、[Amazon SQS](https://aws.amazon.com/sqs/)、[Redis](http://redis.io/)以及同步（本地使用）驱动。默认的是同步驱动sync,那这些不同驱动的队列有什么区别呢，

参考：<https://www.jianshu.com/p/eb047c297e69>

我们将一个一个的任务放在队列中，用键值对的形式来区分不同的任务，键名是任务名，键值则是任务的执行代码。最先想到的是数组，在laravel中sync即同步，用来做测试用的。如果代码太大的话，会放在数据库中，用的时候将键值对从数据库中取出来，这就是数据库驱动，但是这样每次操作数据库速度会慢，因此就有了redis驱动，redis是专门用来处理键值对的，redis的数据存在内存中，这样速度就上来了。当然还有其他第三方的驱动，但终归就是把任务的键名和键值储存起来而已，储存的媒介用什么都可以，而所谓的**队列驱动，就是把储存在这些媒介中的任务代码拿住来按照队列的算法进行执行的一种调配方式的一段代码而已**。

### 7.2配置

每个驱动的配置不一样，以redis驱动配置为例

Laravel的队列配置文件在config/queue.php中，里面可以修改不同驱动的相关配置。

**'redis'** => [  
 **'driver'** => **'redis'**,*//驱动名称* **'connection'** => **'default'**,*//* **'queue'** => env(**'REDIS\_QUEUE'**, **'default'**),*//队列名称默认为default* **'retry\_after'** => 90,*//当任务运行超过 retry\_after 这个时间后，该任务会被重新放入队列当中* **'block\_for'** => **null**,  
 *//block\_for 配置项来指定驱动在迭代队列进程循环并重新轮询 Redis 数据库之前等待可用队列任务的时间。*],

首先在.env中修改QUEUE\_CONNECTION=redis，因为是redis驱动所以要先现在predis,使用composer require predis/predis安装。

### 7.3创建任务

Php artisan make:job test

生成的任务在app/job下，生成的类实现了Illuminate\Contracts\Queue\ShouldQueue接口，里面有两个方法construct，handle。Handle方法里就是你所要做的任务。

(创建好任务在controller中分发任务，在使用php artisan queue:work监听即可)

### 7.4发布任务

创建好任务类后，就可以通过任务自身的dispatch方法将其分发到队列

ProcessPodcast::dispatch($podcast);

Laravel的任务有**延时分发**和**同步分发**两种

#### 延时任务

ProcessPodcast::dispatch()->delay(now()->addMinutes(1));

还有addSeconds()，addDays()

#### 同步分发

ProcessPodcast::dispatchNow($podcast);

### 7.5任务链

所谓的任务链就是有多个任务需要按照一定的顺序执行

**官方定义**：任务链允许你指定一个需要在一个序列中执行的队列任务列表，如果序列中的某个任务失败，其它任务将不再运行

使用方法

ProcessPodcast::withChain([

new OptimizePodcast,

new ReleasePodcast

])->dispatch();

注：使用 $this->delete() 方法删除任务不会阻断正在被处理的任务链中的任务。任务链中的任务只有在执行失败时才会停止执行。

如果你想要指定任务链使用的默认连接和队列，可以使用 allOnConnection 和 allOnQueue 方法。这些方法指定需要用到的队列连接和队列名称，除非队列任务显式指定了分配的连接/队列

ProcessPodcast::withChain([

new OptimizePodcast,

new ReleasePodcast

])->dispatch()->allOnConnection('redis')->allOnQueue('podcasts');

### 7.6分发到指定队列

Laravel队列默认的名称是default，分发任务的时候也是分发到默认的队列上，

如果要分发到指定的队列上可以这样

ProcessPodcast::*dispatch*()->onQueue(**'my'**);

运行队列的时候

php artisan queue:work 也是默认运行default

运行 指定队列

php artisan queue:work redis --queue=my

一旦向上面那样将分发到指定的队列上时，在像这样php artisan queue:work运行default队列是监听不到的。

这里的onQueue（）方法放在任务类的相关介绍中介绍

### 7.7****分发到指定的连接****

如果你使用了多个连接来管理队列，那么可以分发任务到指定的连接。请在任务实例中使用 onConnection 方法来指定连接:

指定连接就是指不同的驱动

ProcessPodcast::dispatch($podcast)->onConnection('sqs');

运行的的时候像这样监听

php artisan queue:work redis就是监听redis连接的队列

### 7.8任务失败处理

在fail()方法中进行相关操作

### 7.9关于任务类的相关介绍

参考：

<https://learnku.com/articles/7037/laravel-queue-analysis-of-message-queue-tasks-and-distribution-source-code>

<https://blog.csdn.net/szuaudi/article/details/105809256>

通过php artisan make job test生成任务类，

任务类有\_\_construct构造方法，和handle方法，构造方法里可以传递一些数据，handle方法里用来处理任务一般用来实现具体任务。

另外任务类中还引用了Dispatchable, InteractsWithQueue, Queueable, SerializesModels，四个trait。

Dispatchable中定义了分发任务的方法有dispatch（），dispatchNow()

和withChain()

InteractsWithQueue中定义了attempts()，delete()，fail（）方法

Fail()方法在失败发生时执行指定的动作，比如发送任务失败的通知，记录日志等

Queueable中定义一些队列的相关配置像onQueue()这样指定队列名称的

### 7.9.1任务的相关配置

参考：

<https://learnku.com/articles/7037/laravel-queue-analysis-of-message-queue-tasks-and-distribution-source-code>

任务的相关配置一般有两种，一种是在任务类的配置，另一种是用artisan来设置

**指定最大失败次数**

php artisan queue:work --tries=3

或者

public $tries = 5;

**基于时间的尝试次数**

除了定义在任务失败前的最大尝试次数外，还可以定义在指定时间内允许任务的最大尝试次数，这可以通过在任务类中添加 retryUntil方法来实现

public function retryUntil()

{

return now()->addSeconds(5);

}

**队列任务最大运行时长（秒）**

队列进程 queue:work 可以设定超时 --timeout 项。该 --timeout 控制队列进程执行每个任务的最长时间，如果超时，该进程将被关闭。各种错误都可能导致某个任务处于“冻结”状态，比如 HTTP 无响应等。队列进程超时就是为了将这些“冻结”的进程关闭

**php artisan queue:work --timeout=30**

**public $timeout = 120;**

，任务中指定的超时时间优先级也高于命令行定义的数值

让任务延迟推送

public $delay = 30;

如果想要使得任务被推送到队列中，而不是同步执行，那么需要实现 Illuminate\Contracts\Queue\ShouldQueue 接口。

如果想要让任务推送到特定的连接中，例如 redis 或者 sqs，那么需要设置 conneciton 变量。

如果想要让任务推送到特定的队列中去，可以设置 queue 变量。

如果想要让任务延迟推送，那么需要设置 delay 变量。

如果想要设置任务至多重试的次数，可以使用 tries 变量；

如果想要设置任务可以运行的最大秒数，那么可以使用 timeout 参数。

如果想要手动访问队列，可以使用 trait : Illuminate\Queue\InteractsWithQueue。

如果队列监听器任务执行次数超过在工作队列中定义的最大尝试次数，监听器的 failed 方法将会被自动调用。 failed 方法接受事件实例和失败的异常作为参数：

### 7.92任务事件

关于在任务中的事件监听一般放在boot()方法中，

{

//任务运行前

Queue::before(function (JobProcessing $event) {

// $event->connectionName

// $event->job

// $event->job->payload()

});

//任务运行后

Queue::after(function (JobProcessed $event) {

// $event->connectionName

// $event->job

// $event->job->payload()

});

//任务循环前

Queue::looping(function () {

while (DB::transactionLevel() > 0) {

DB::rollBack();

}

});

//任务失败后

Queue::failing(function (JobFailed $event) {

// $event->connectionName

// $event->job

// $event->exception

});

//异常发生后

Queue::exceptionOccurred(function (JobFailed $event) {

// $event->connectionName

// $event->job

// $event->exception

});

}

### Laravel队列配置supervisor

参考：<https://www.cnblogs.com/DXYHW/p/10558340.html>

[program:laravel-worker]

process\_name=%(program\_name)s\_%(process\_num)02d

command=php /home/www/wechatdemo5/artisan queue:work redis

autostart=true

autorestart=true

user=root

numprocs=1

redirect\_stderr=true

stdout\_logfile=/var/log/supervisor/laravel-work.log

查看进程管理状态

ps -ef | grep 'supervisord'

## 八．基础组件

### 表单验证

Php artisan make:request test

将authorize方法return true表示用户已经经过授权

#### 使用方法：

在接口方法出依赖注入，在接受参数之前会自动拦截请求并验证 错误则抛出错误信息

#### 自己定义错误信息返回格式

重写FormRequest的failedValidation方法

**protected function** failedValidation($validator)  
{  
 $error = $validator->errors()->first();  
 *// $allErrors = $validator->errors()->all(); 所有错误* $response = response()->json([  
 **'code'** => 0,  
 **'msg'** => $error,  
 **'data'** => **null**,  
 ]);  
  
 **throw new** HttpResponseException($response);  
}

#### 添加使用场景

### 中间件

## 九．异常处理

### Laravel默认的异常处理类：app\Exceptions\Handler

Handler中两个重要方法report()和render()

Php artisan make:exception MyException

## git下来的laravel缺少vendor

composer install 报错的话就删掉composer.lock 再composer install

laravel git项目要先 composer install

## Composer.lock作用

Composer安装的时候从composer.lock中读取依赖版本，而不是根据 composer.json 文件去获取依赖。这确保了该库的每个使用者都能得到相同的依赖版本。

如果没有 composer.lock 文件，composer 将在处理完依赖关系后创建它。

## response()->json()

## 响应类response