目錄

前言

准备

微信支付

H5 移动端支付

公众号支付

附录


based on node.js implementation
微信支付
实施细节
by
ninghao.net

介绍

这本电子书里介绍了在你自己的网站应用里实现微信支付功能你能需要做些什么。这是本书的主要目的，所以我会尽量减小其它东西带来的干扰，让你专注在微信支付的功能，方法，还有流程上。

Node.js 使用简单，也具备现代开发的特性。所以书里我用了一个 Node.js 框架，实现了微信支付功能。这并不要求你精通 Node.js 或者我特别使用的这套框架，你只需要跟着做就行了。打算深入学习 Node.js 的同学可以参考宁皓网的《Node.js》这个课程包。

需求

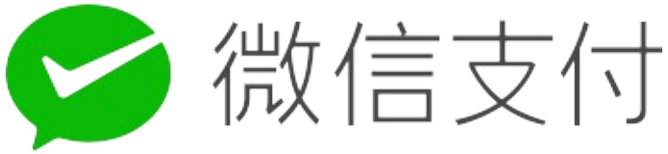
需要基本的程序设计知识，不然看起来会比较吃力。初学者，我建议先学会 JavaScript 语言，学会页面设计与开发，然后再学会 Node.js 。

书里会用到命令行界面，比如起动本地开发环境，跟服务器之间打个 SSH 通道等等。另外书中的代码放在了 github 提供的远程仓库里了，所以你需要理解 Git 这个工具的使用方法。

搭建微信支付的本地开发环境，需要做点特别的配置，因为需要我们的本地开发环境能在互联网上被访问到。我用的是服务器 SSH 通道的方法，具体参考本书的开发环境这一节内容。

视频

这本电子书是宁皓网原创视频《微信支付》的文字版，订阅宁皓网以后，您就可以在线学习这个视频课程了。视频比文字更直接一些，如果您是宁皓会员，建议您先浏览视频，然后再把本书作为练习用的一个参考。



作者

王皓，自由职业者，宁皓网作者。目前主要是在独立运营一个叫宁皓网的在线学习网站。

项目代码

书中出现的所有代码你可以在 github 网站上，ninghao/ninghao-sandbox-v2 这个仓库里找到。注意你需要切换不同的分支查看对应的代码。本书涉及的三个分支分别是：*wxpay*，*wxpay-h5*，*wxpay-jsapi*。把项目切换到不同的分支，在说明文档里有创建项目需要做的事情。

准备工具

1. 命令行工具 Windows 下载安装完整版的 Cmder，macOS 使用系统自带的终端（Terminal），当我在文章里提到打开命令行工具的时候，指的就是打开命令行界面（CLI），意思就是让你打开这两个工具里的其中的一个（Cmder / Terminal）。
2. 文本编辑器 我用的是 Atom ，在上面安装几个需要的插件（package）：
    - emmet 可以用缩写形式创建 HTML 与 CSS 代码。
    - language-log 高亮显示 log 类型的日志文件。
    - edge 高亮显示 edge 模板引擎代码。
3. 浏览器 推荐谷歌浏览器（Chrome），或者其它带 Chromium 核心的浏览器。

创建项目

书中我会在一个 Node.js 应用里实施微信支付的功能，要用的框架是Adonis.js。确定本地电脑上已经安装好了 Node.js**v9.x.x**（主要是因为代码里用到了 spread 操作符），推荐使用 NVM 来管理安装的 Node.js，这样可以很容易切换使用不同版本的 Node.js。有了 Node.js，再去安装框架需要的命令行工具。

安装框架命令行

```
npm install @adonisjs/cli --global
```

创建项目

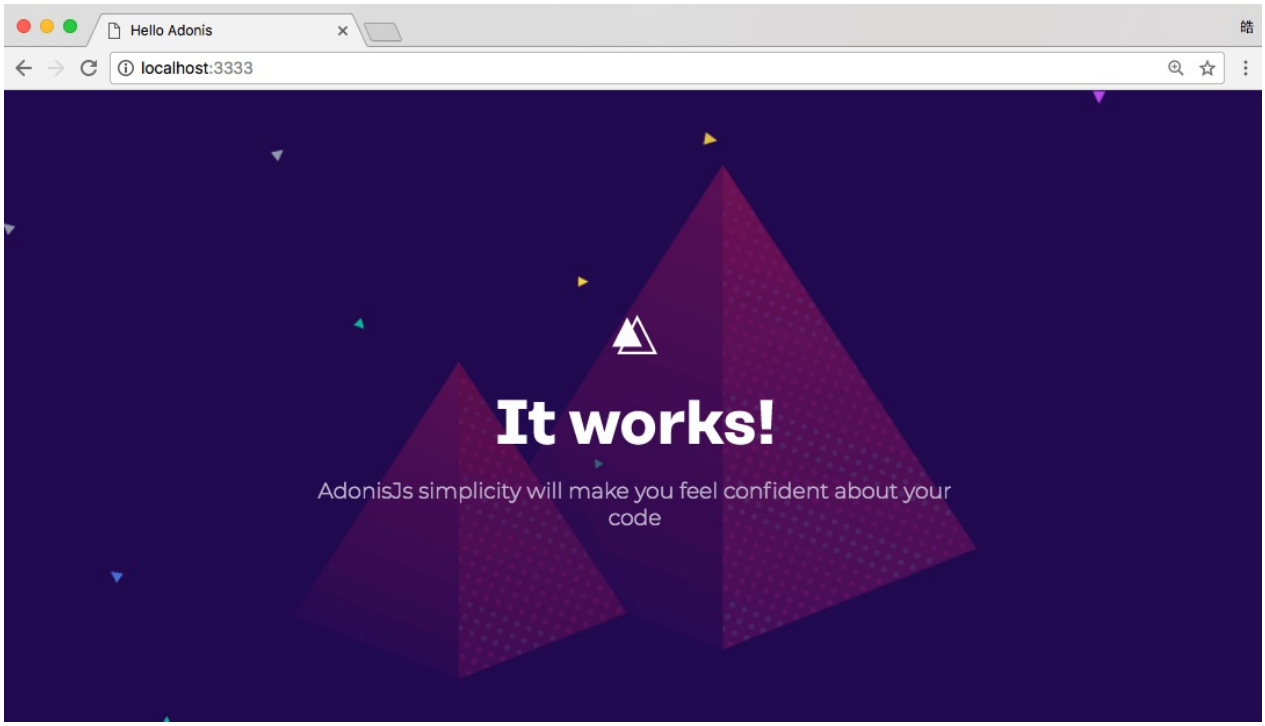
使用安装好的 Adonis 框架的命令行工具，去创建一个基于 Adonis 框架的 Node.js 应用。

```
adonis new ninghao-sandbox-v2
```

完成以后进入到项目所在的目录，再去运行一下项目。

```
cd ninghao-sandbox-v2
adonis serve --dev
```

打开浏览器，访问一下*localhost:3333*，可以打开创建的 Adonis 项目。


Hello Adonis
localhost:3333
It works!
AdonisJs simplicity will make you feel confident about your code

开发环境

搭建微信支付的开发环境，要保证开发环境可以在互联网上被访问到。因为微信支付系统会通过应用支付的结果，如果开发环境在互联网上不能被访问到，也就收不到这个支付结果。在开发很多其它外部服务的时候，都需要这样做，比如支付宝，微信公众号等等。下面介绍的方法同样适用于搭建开发这些服务的环境。

我用的方法是使用了一台能在互联网上被访问到的服务器，在上面用 Nginx 配置一个代理，把请求转发到服务器的某个端口上。然后用 ssh 在本地跟服务器之间打一个通道，通道用的端口就是配置 Nginx 的时候，设置的上游服务（*upstream*）的端口。再去设置一个主机名，让它指向服务器，服务器会把请求转发到通道用的端口，这样实际提供服务的就是我们在本地上搭建的开发环境了。

1. 准备一台能在互联网上被访问到的服务器，比如 阿里云 ECS。
2. 设置域名 DNS，让某个主机名指向服务器。
3. 配置服务器的 Nginx 代理。
4. 在本地电脑上用 ssh 打通道。

服务器

我用了一台阿里云服务器，任何能在互联网上被访问到的服务都可以。服务器的 IP 地址是*42.120.40.68*。在本地电脑，使用命令行工具可以登录到服务器。Windows 推荐使用 Cmder 作为命令行工具，macOS 用户可以使用系统自带的终端（Terminal）。

```
ssh wanghao@42.120.40.68
```

*wanghao*是在我的服务器上的某个用户的名字，这个用户是我自己创建的。一开始，你可以使用服务器的超级管理员：*root*登录到你的服务器，然后你可以创建新的用户，为用户分配权限等等。

主机名

配置一个主机名，指向服务器的 IP 地址，这个主机名就是访问本地开发环境用的主机名。我让*sandbox.ninghao.net*这个主机名指向了我的服务器：*42.120.40.68*。

验证主机名是否已经解析到指定的服务器，可以在命令行下面用 ping 命令测试一下：

```
→ ping sandbox.ninghao.net
PING sandbox.ninghao.net (42.120.40.68): 56 data bytes
```

用*ping*命令测试*sandbox.ninghao.net*的时候，返回的 IP 地址就是我的服务器的 IP 地址。

Nginx 代理

在服务器上安装 Nginx，再去配置一台 Nginx 代理服务器。我的服务器操作系统是 CentOS 7.x，安装 Nginx ，执行：

```
# 安装包含 Nginx 的仓库
sudo yum install epel-release -y

# 安装 Nginx
sudo yum install nginx -y

# 启动 Nginx
sudo systemctl start nginx

# 开机自启动 Nginx
sudo systemctl enable nginx
```

代理配置

在 Nginx 的配置目录的下面，创建一个新的配置文件，放在 */etc/nginx/conf.d*，这个目录是*/etc/nginx/nginx.conf*配置文件里面设置的。配置文件所在的目录可能会有变化，比如有可能在 */etc/nginx/sites-enabled*。具体配置文件所在的目录，你要根据自己的实际情况，检查 Nginx 的主配置，就是 nginx.conf 里面的配置。

/etc/nginx/conf.d/sandbox.ninghao.net.conf：

```
upstream tunnel {
  server 127.0.0.1:7689;
}

server {
  listen                  443 ssl;
  server_name              sandbox.ninghao.net;
  ssl                      on;
  client_max_body_size     128m;
  ssl_certificate          /etc/nginx/ssl/sandbox.ninghao.net/214241634170706.pem;
  ssl_certificate_key      /etc/nginx/ssl/sandbox.ninghao.net/214241634170706.key;
  ssl_session_timeout      5m;
  ssl_protocols            TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers              AESGCM:ALL:!DH:!EXPORT:!RC4:+HIGH:!MEDIUM:!LOW:!aNULL:!eNU
LL;
  ssl_prefer_server_ciphers on;

  location / {
    proxy_set_header  X-Real-IP  $remote_addr;
    proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header  Host $http_host;
    proxy_redirect    off;
    expires           off;
    sendfile          off;
    proxy_pass        http://tunnel;
  }
}
```

上面配置了一台服务器，里面用到了 ssl ，这样服务器就支持使用 https 来传输数据了，这是现代网站或应用并且要做的事情。配置 ssl ，你需要申请 ssl 证书，在阿里云，你可以申请免费的 ssl 证书，或者使用 Let's Encrypt，也可以免费申请到证书。

有了证书，就按上面的配置，证书一般有两个文件，把它们存储在服务器上的某个目录的下面，然后分别设置*ssl_certificate*与*ssl_certificate_key*指令的值。

> .key 与 .pem 只是证书惯用的文件扩展名，这个扩展名不重要，重要的是文件里面的内容。.key 指的是证书的密钥，.pem 是证书内容。用 Let's Encrypt 申请的证书，证书文件的扩展名应该是 .cert。

证书密钥

证书密钥文件的扩展名一般是*.key*，这个文件里的内容应该作为 Nginx 服务器配置里的*ssl_certificate_key*指令的值。文件里的内容像下面这样：

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAiDnt/zboVuY23iQBYcocvJSewGDol87FoLOquz61L8rNKc5p
...
-----END RSA PRIVATE KEY-----
```

证书内容

证书内容文件的扩展名一般是.*pem*或者.*cert*，文件要作为 Nginx 服务器配置里的 *ssl_certificate* 指令的值。文件里的内容像下面这样：

```
-----BEGIN CERTIFICATE-----
MIIFgjCCBGqgAwIBAgIQdtV9VRMTtKBDDfN3UO3ujTANBgkqhkiG9w0BAQsFADCB
...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFYjCCBEqgAwIBAgIQTEzYoPxP6q4VVKh/CQ7ahzANBgkqhkiG9w0BAQsFADCB
...
-----END CERTIFICATE-----
```

重新启动

对 Nginx 服务做了新的配置，要重新加载或者重新启动 Nginx 才能让新的配置生效。执行：

```
# 检查配置文件是否正常
sudo nginx -t

# 重新加载 Nginx
sudo systemctl reload nginx
```

ssh 通道

准备好了服务器，现在要在本地电脑上用 ssh 在本地与服务器之间打个通道。执行：

```
ssh -vnNT -R 7689:localhost:3333 wanghao@42.120.40.68
```

7689，是通道用的端口，这个端口是在配置 Nginx 的时候设置的，具体是在*upstream*里面设置的：

/etc/nginx/conf.d/sandbox.ninghao.net.conf

```
upstream tunnel {
  server 127.0.0.1:7689;
}
```

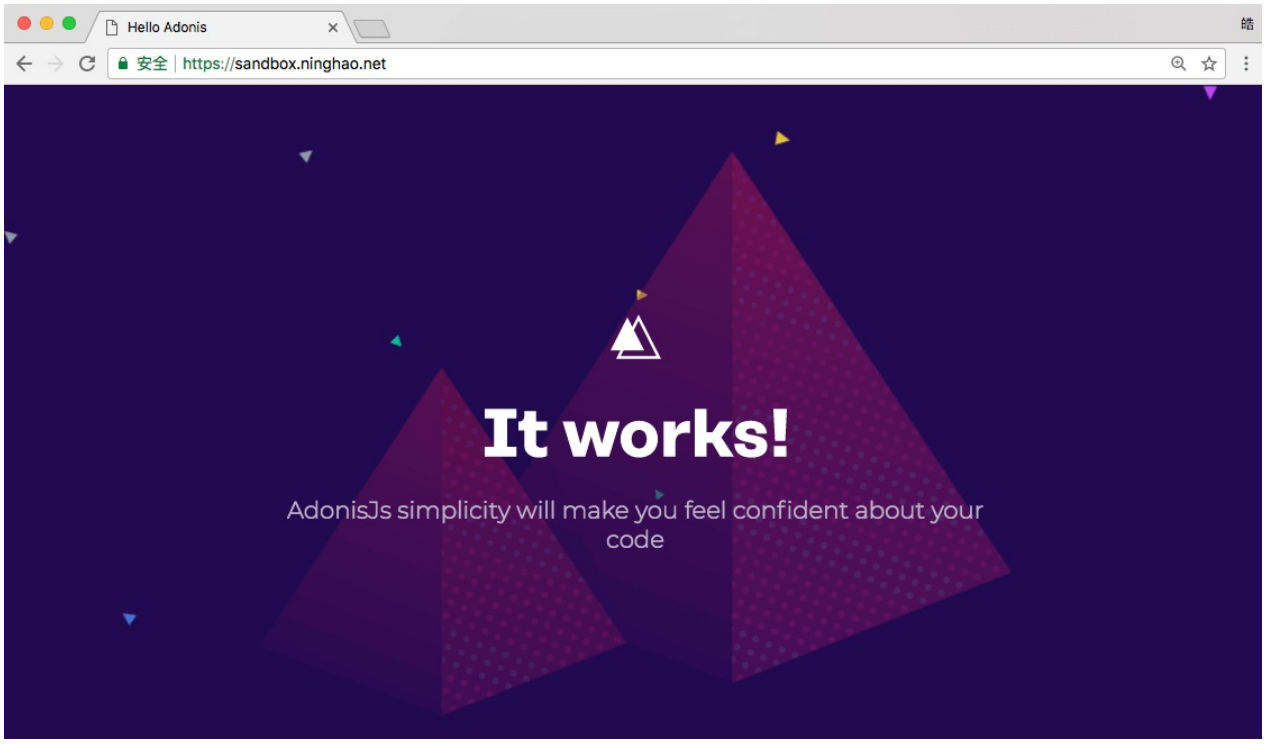
localhost:3333，这是本地项目服务的地址，就是之前我们创建的那个 Node.js 应用，在本地可以使用这个地址访问到创建的这个应用。

wanghao@42.120.40.68，是登录服务器的时候用的东西，*wanghao*是服务器上的某个用户的名字，*42.120.40.68*是服务器的 IP 地址，在这台服务器上之前我们已经配置好了 Nginx 。

成功以后，在输出的信息里，你会看到类似下面的字样：

```
remote forward success for: listen 7689, connect localhost:3333
```

现在，就可以直接使用***sandbox.ninghao.net***这个主机名，访问到我们在本地开发环境上运行的项目了。



项目路由，控制器，视图

我们的主要目的是去理解微信支付的流程，所以项目本身尽量保持简单。大部分的代码都会放在一个控制器文件里，这样你就很轻松，清楚的知道，实施微信支付的扫码支付，你都需要去做什么。用 Atom 编辑器打开创建的项目目录（*ninghao-sandbox-v2*）。

路由

先去添加两条路由。

start/routes.js：

```
Route.get('checkout', 'CheckoutController.render')

Route.post('wxpay/notify', 'CheckoutController.wxPayNotify')
```

/checkout：访问 *checkout* 这个地址可以显示一个支付用的二维码，用户扫描二维码完成支付。这个地址用 *CheckoutController* 控制器里的 *render* 方法来处理。等会儿再去创建它。

/wxpay/notify：用户支付成功以后，微信支付系统会把支付结果异步地通知我们，通知的地址可以使用 */wxpay/notify*，这里用*CheckoutController* 控制器里的 *wxPayNotify* 方法来处理对应用里的这个地址的请求。

控制器

创建路由里面要用的 *CheckoutController* 控制器，打开命令行，在项目所在目录的下面，执行：

```
adonis make:controller Checkout
```

这样会创建一个叫 *CheckoutController.js* 的文件，位置是：*app/Controllers/Http/CheckoutController.js*，打开这个控制器文件，在控制器里添加两个方法：

```
'use strict'

class CheckoutController {
  async render ({ view }) {
    // 显示支付页面
    return view.render('checkout')
  }

  async wxPayNotify ({ request }) {
    // 处理支付结果通知
  }
}

module.exports = CheckoutController
```

在 render 方法里，指定使用了一个视图。

视图

创建 *CheckoutController* 里面需要用到的视图。

布局

先在应用里创建一个视图的布局。

resources/views/layouts/main.edge：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  {{ css('https://unpkg.com/bootstrap@4.0.0/dist/css/bootstrap.min.css') }}
  {{ css('main') }}
  <title>ninghao</title>
</head>
<body>
  @!section('content')
  {{ script('https://unpkg.com/jquery@3.3.1/dist/jquery.js') }}
  {{ script('https://unpkg.com/bootstrap@4.0.0/dist/js/bootstrap.min.js') }}
  {{ script('main') }}
</body>
</html>
```

布局里链接使用了 Bootstrap 框架。

@!section('content')，定义了一个 *content* 区域，在使用了这个布局的地址可以去定义 *content* 区域里面具体要显示的内容。

css('main')，链接了一个自定义的样式表叫 *main.css*，在项目的 *public* 目录的下面，你可以去创建这个自定义的样式表文件。

，在布局中嵌入了一个自定义的脚本文件：*main.js*，在项目的 *public* 目录的下面，可以去创建一个 *main.js* 文件。

视图

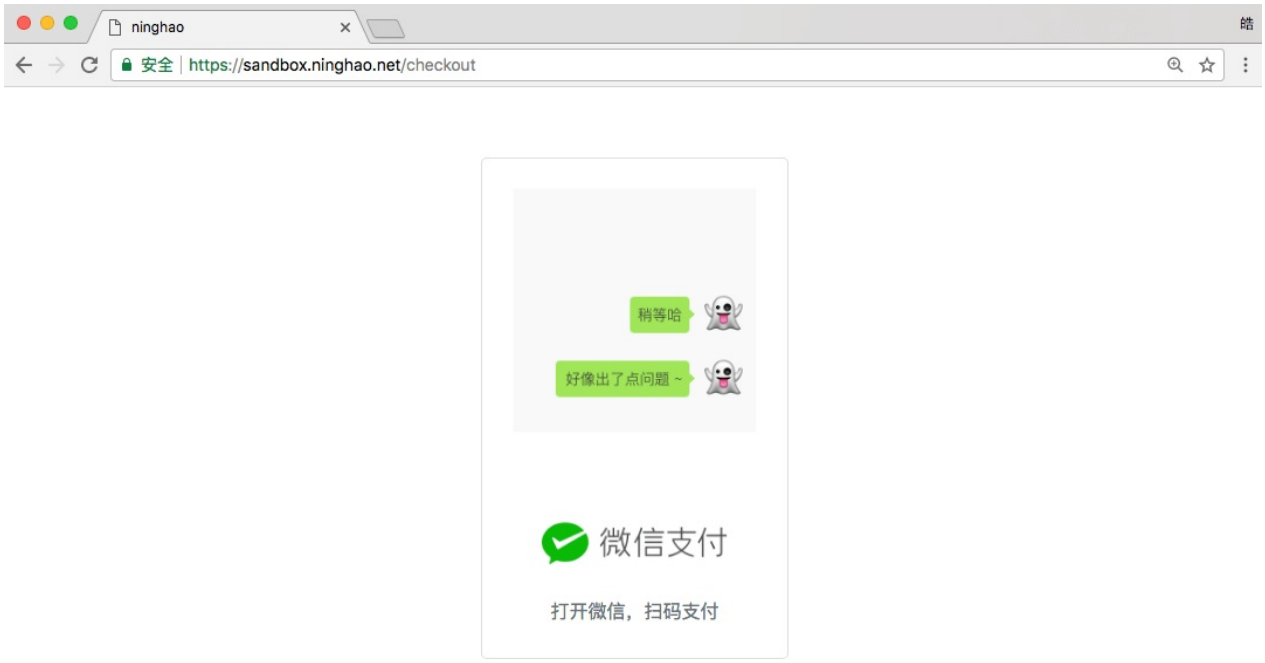
再去创建一个视图文件。

resources/views/commerce/checkout.edge：

```
@layout('layouts.main')

@section('content')
<div class="container">
  <div class="row justify-content-center">
    <div class="col-md-4">
      <div class="card text-center mt-5">
        <img class="card-img-top" src="{{ assetsUrl('something-wrong.png') }}">
        <div class="card-body">
          <img class="w-100" src="{{ assetsUrl('wxpay.png') }}" />
          <p class="card-text">
            <small class="text-muted">打开微信，扫码支付</small>
          </p>
        </div>
      </div>
    </div>
  </div>
</div>
@endsection
```

视图使用了 *layouts.main* 这个布局，里面定义了布局里的 *content* 区域要显示的内容。这里用了一个 Bootstrap 的 card 组件，显示的就是一个支付卡片，不过暂时还没有要显示的支付二维码，所以我们用了一个点位符图片（something-wrong.png），以后在应用里生成了支付用的二维码以后，可以用二维码图片替换这个点位符图片的显示。

访问地址：https://sandbox.ninghao.net/checkout


ninghao
安全 | https://sandbox.ninghao.net/checkout
稍等哈
好像出了点问题 ~
微信支付
打开微信，扫码支付

支付流程

微信扫码支付模式二

1. 请求预支付 应用为用户生成订单以后，根据订单内容向微信支付系统请求一个预支付的链接。
2. 生成二维码 应用得到了链接，根据链接内容生成支付用的二维码图像展示在支付页面上。
3. 确认并完成支付 用户用微信扫描支付二维码，确认并完成支付。
4. 接收并处理支付结果 支付完成以后，微信支付系统会通知我们的应用支付的结果。应用可以验证支付结果是否有效，再去执行对应的动作，比如更新用户的订单状态。
5. 回复检查支付结果的状态 最后要回复一下微信支付系统我们自己验证的支付结果的状态，可以是 SUCCESS（成功）😀，也可以是 FAIL（失败） 。

注意事项

实施微信支付功能并不难，只是有很多细节需要注意。在请求得到预支付的时候，你要准备很多数据，请求的时候要带着这些数据。这些数据有些是微信支付提供给我们的，比如公众账号ID（*appid*），商户号（*mch_id*）。有些数据是跟订单或商品相关的，比如订单号（*out_trade_no*），商品描述（*body*），商品金额（*total_fee*）等等。还有些数据我们要自己现去生成，比如随机字符串（*nonce_str*），还有签名（*sign*）。

数据格式

还要注意微信支付的请求或响应的数据都要使用 xml 格式的。也就是如果你要把数据发送给微信支持系统，你要把数据的格式转换成 xml。从微信支付那里获取到的数据也是 xml 格式的，在应用里想要使用它们，你需要把 xml 格式的数据转换成应用能懂的数据格式。比如，我们会在 Node.js 应用里实施微信支付，所以数据的转换就是把 JavaScript 的对象（object）转换成 xml ，还需要把 xml 转换成 JavaSciprt 的对象。

接口

微信支付提供了一些支付的方式，比如公众号支付，扫码支付，APP 支付等等，支付方式确定了微信支付的流程。另外，还有一些 API 接口， 比如下单，查询，退款，这些功能都有一个对应的 API 接口。文章里我们暂时只关注支付功能，所以只需要用到统一下单这个 API 。在请求预支付的时候，请求的就是这个统一下单接口。

请求预支付时需要的数据

请求统一下单接口获取预支付，请求的时候需要带着一些数据。先去准备这些数据，这些数据就是在统一下单接口里面规定好的（除了*key*，它是在创建签名时需要用的密钥）。下面我们暂时只准备必填的数据，具体可用的数据，你要参考官方文档，去查看统一下单的文档。

账户

1. *appid*：公众账号 ID
2. *mch_id*：商户号

*appid*与*mch_id*是成功开通微信支付以后的邮件里，或者微信支付的后台可以找到。

支付

1. *trade_type*：交易类型

如果选择扫码支付功能，trade_type 的值应该是 NATIVE。

商品 / 订单

1. *out_trade_no*：商户订单号
2. *body*：商品描述
3. *total_fee*：标价金额
4. *product_id*：商品 ID

这些东西是跟商品或者订单相关的。

应用

1. *notify_url*：通知地址

支付成功以后微信支付系统会把结果发送到 notify_url 设置的地址。这个地址我们应该在应用里自己去定义，它的功能就是得到请求里的数据，也就是支付结果。然后验证结果是否有效，再去做出对应的动作，比如更新用户在应用上提交的订单的状态。

签名

1. *nonce_str*：随机字符串
2. *key*：密钥
3. *sign*：签名

在请求预支付的时候，我们要带的数据里面要包含一个签名。这个签名的算法是微信支付系统规定好的，我们要按照这个规定，一步一步地算出这个签名。

日志服务

先在应用里添加一个日志服务，可以方便做调试，这个日志服务可以把指定和信息输出到一个文本文件里。

安装

```
npm install log4js --save
```

服务

在应用里，创建一个文件。*app/Services/Logger.js*：

```
const log4js = require('log4js')

log4js.configure({
  appenders: {
    file: {
      type: 'file',
      filename: 'app.log',
      layout: {
        type: 'pattern',
        pattern: '%r %p - %m',
      }
    }
  },
  categories: {
    default: {
      appenders: ['file'],
      level: 'debug'
    }
  }
})

const logger = log4js.getLogger()

module.exports = logger
```

使用

在*CheckoutController.js*文件的顶部，导入创建的日志服务：

```
const logger = use('App/Services/Logger')
```

输出日志信息到日志文件里（*app.log*），可以使用下面这样的形式：

```
logger.debug('日志数据')
```

配置文件

跟微信支付相关的一些数据可以保存在一个配置文件里。创建一个配置文件 *wxpay.js*，放在应用的 *config* 目录的下面。

配置

config/wxpay.js：

```
'use strict'

const Env = use('Env')

module.exports = {
  // 公众账号ID
  appid: Env.get('WXPAY_APP_ID'),

  // 商户号
  mch_id: Env.get('WXPAY_MCH_ID'),

  // 密钥
  key: Env.get('WXPAY_KEY'),

  // API
  api: {
    // 统一下单
    unifiedorder: 'https://api.mch.weixin.qq.com/pay/unifiedorder'
  },

  // 通知地址
  notify_url: 'https://wxpay-dev-demo.ninghao.net/wxpay/notify'
}
```

环境变量

注意上面有些配置的值用到了 *Env.get* 方法获取到，是因为这些数据比较敏感，不应该直接保存在配置文件里。所以我们把它放在了环境变量里，*Env.get* 可以从环境变量里获取到值。在项目下面的 *.env* 文件里，存储的就是一些环境变量。

.env

```
...

WXPAY_APP_ID=wx5826s149db20f28e
WXPAY_MCH_ID=1328538902
WXPAY_KEY=3fx1815ed8bf4908sde12287eb6a7f92
```

> .env 文件不会在项目的版本控制里出现。

使用配置

下面可以在之前创建的 *CheckoutController* 里面用一下刚才定义的这些配置。

app/Controllers/Http/CheckoutController.js：

先在文件里导入 Config：

```
const Config = use('Config')
```

然后把*CheckoutController*的*render*方法，改成下面这样：

```
  async render ({ view }) {
    // 公众账号 ID
    const appid = Config.get('wxpay.appid')

    // 商户号
    const mch_id = Config.get('wxpay.mch_id')

    // 密钥
    const key = Config.get('wxpay.key')

    // 通知地址
    const notify_url = Config.get('wxpay.notify_url')

    // 统一下单接口
    const unifiedOrderApi = Config.get('wxpay.api.unifiedorder')
  }
```

这里就是用了*Config.get* 方法，得到了 *wxpay* 这个配置文件里面的对应的配置数据。

商品 / 订单

再去准备商品 / 订单相关的数据，这些数据我们直接手工设置一下。在实际应用中，这些数据你应该根据应用的购物车系统去生成。在 *CheckoutController* 的 *render* 方法里，再添加下面这些代码：

```
// 商户订单号
const out_trade_no = moment().local().format('YYYYMMDDHHmmss')

// 商品描述
const body = 'ninghao'

// 商品价格
const total_fee = 3

// 支付类型
const trade_type = 'NATIVE'

// 商品 ID
const product_id = 1
```

out_trade_no 是订单号，每笔交易的订单号都应该不一样。这里我用了 *moment* 得到了当前时间，用它作为订单号。这个 *moment* 要去安装一下，打开命令行，然后在项目目录的下面，执行：

```
npm install moment --save
```

然后在 *CheckoutController* 文件的顶部导入 *moment*，添加下面这行代码：

```
const moment = use('moment')
```

签名

现在可以去生成签名。签名的计算方法是微信支付规定的：

1. 给数据排序。
2. 把数据转换成地址查询符字符串。
3. 在转换后的字符串结尾添加 key（密钥）。
4. 用 md5 处理字符串。
5. 将处理结果全部转换成大写字母。

先再准备一个随机字符（*nonce_str*），可以为项目安装一个 *randomstring* 这个 package：

```
npm install randomstring --save
```

然后在*CheckoutController.js*文件的顶部，导入*randomstring*，添加代码：

```
const randomString = use('randomstring')
```

在*render*方法里，再添加一个*nonce_str*：

```
// 随机字符
const nonce_str = randomString.generate(32)
```

准备签名数据

在 *render* 方法里，再去准备一下参与签名的数据，可以放在一个 object 里面：

```
// 统一下单
let order = {
  appid,
  mch_id,
  out_trade_no,
  body,
  total_fee,
  trade_type,
  product_id,
  notify_url,
  nonce_str
}
```

签名方法

在 *CheckoutController* 这个类里面添加一个生成签名用的方法，名字是 *wxPaySign*，代码如下：

```
  wxPaySign (data, key) {
    // 1. 排序
    const sortedOrder = Object.keys(data).sort().reduce((accumulator, key) => {
      accumulator[key] = data[key]
      // logger.debug(accumulator)
      return accumulator
    }, {})

    // 2. 转换成地址字符
    const stringOrder = queryString.stringify(sortedOrder, null, null, {
      encodeURIComponent: queryString.unescape
    })

    // 3. 结尾加上密钥
    const stringOrderWithKey = `${ stringOrder }&key=${ key }`

    // 4. md5 后全部大写
    const sign = crypto.createHash('md5').update(stringOrderWithKey).digest("hex").toU
pperCase()

    return sign
  }
```

1，排序

在方法里，先根据传递进来的 data ，去生成一个排序之后的新的排序之后的对象。

2，转换

把对象转换成地址查询符，这里用到了 Node.js 里的 *querystring* 这个模块提供的功能。在文件的顶部要去导入这个模块：

```
const queryString  = use('querystring')
```

3，加密钥

在字符的结尾再添加上密钥的值，这个密钥你可以在微信支付管理后台去设置。

4，md5 处理后全部大写

用 md5 处理字符，再把结果转换成全部大写字母。这里用到了 Node.js 的 *crypto* 模块，所以要在文件顶部导入这个模块：

```
const crypto = use('crypto')
```

生成签名

现在可以在*render*方法里，使用上面定义的*wxPaySign*方法去生成签名数据了。方法需要的两个参数，提前已经在*render*方法里准备好了。

```
const sign = this.wxPaySign(order, key)
```

签名也要作为请求统一下单接口的时候带着的数据，所以我们得到了签名值以后，可以把它放到要发送的数据里。这里可以重新设置一下之前定义的 order 的值 。

```
    order = {
       xml: {
         ...order,
         sign
       }
     }
```

注意上面我们重新定义 *order* 的时候，在它里面先添加了一个 *xml* 属性，这个属性一会在把数据转换成 *xml* 格式的时候会用到，它会被转换成一组 *<xml>* 标签，里面会包装着要发送的具体的数据。*...order* 的意思是把之前 *order* 里的东西放进来，... 是 *spread* 操作符。然后又在对象里添加了一个 *sign*，它是刚才我们生成的签名的值。

转换成 XML

现在已经准备好了请求统一下单接口的时候要带着的数据，下面要做的是把这些数据转换成 XML 格式的。这里要用到一个 xml-js，先去安装一下它：

```
npm install xml-js --save
```

然后在 *CheckoutController.js* 文件的顶部导入这个模块：

```
const convert = use('xml-js')
```

在控制器的*render*方法里可以去转换数据：

```
    // 转换成 xml 格式
    const xmlOrder = convert.js2xml(order, {
      compact: true
    })
```

上面用到了*xml-js*里的 *js2xml* 这个方法，它可以把 object 格式的数据转换成 xml 格式的数据。比如：

```
{
  xml: {
    appid: 'xxx'
  }
}
```

上面这个对象转换成 xml 以后，会变成：

```
<xml>
  <appid>xxx</appid>
</xml>
```

请求统一下单接口

现在我们就真正准备好了请求统一下单接口需要的数据，这样就可以去请求统一下单接口来得到预支付的链接了。在 Node.js 应用里发出 HTTP 请求，可以使用 axios 这个包。先去安装一下它：

```
npm install axios --save
```

导入包：

```
const axios = use('axios')
```

发出请求

在 *CheckoutController.js* 里面的 *render* 方法里，可以去执行请求动作，然后把得到的响应交给了*wxPayResponse*。

```
    // 调用统一下单接口
    const wxPayResponse = await axios.post(unifiedOrderApi, xmlOrder)
```

注意 *axios* 会返回 Promise ，所以在它前面可以用一个 *await* 去等待执行的结果。之前我们在这行代码所在的 render 方法的前面已经添加了一个 *async*，意思就是这个 *render* 方法里有一些异步动作。

```
async
 render ({ view }) {
  ...
}
```

处理响应

如果一切正常，请求统一下单接口返回的响应的数据（wxPayResponse.data），像下面这样：

```
<xml>
  <return_code><![CDATA[SUCCESS]]></return_code>
  <return_msg><![CDATA[OK]]></return_msg>
  ...
  <code_url><![CDATA[weixin://wxpay/bizpayurl?pr=yCYZHGS]]></code_url>
</xml>
```

这里有我们需要生成支付二维码用到的 *code_url*。但是响应回来的数据是 xml 格式的，所以我们不能直接用，需要转换一下它。

转换

```
    const _prepay = convert.xml2js(wxPayResponse.data, {
      compact: true,
      cdataKey: 'value',
      textKey: 'value'
    }).xml

    const prepay = Object.keys(_prepay).reduce((accumulator, key) => {
      accumulator[key] = _prepay[key].value
      return accumulator
    }, {})
```

转换之后的响应数据交给了*prepay*，它里面的东西应该像下面这样：

```
{
  return_code: 'SUCCESS',
  return_msg: 'OK',
  ...
  code_url: 'weixin://wxpay/bizpayurl?pr=yCYZHGS'
}
```

这个响应里的 *code_url* 的值，就是支付二维码表示的数据。

处理支付结果通知

得到微信发送过来的支付结果以后，要验证支付结果的签名，还有用户支付的金额。然后要把验证的结果回复给微信支付系统，可以回复 SUCCESS（成功），或者 FAIL（失败）。

修改配置

一开始我们就在应用里定义了通知的地址 *wxpay/notify*，现在要去修改一下应用的 *config/shield.js*，在 *filterUris* 里面，添加一个 ***'/wxpay/notify'***。意思就是去告诉应用，*/wxpay/notify* 不需要 *csrf* 保护，不然的话微信支付系统发过来的支付结果我们没法用。

```
  csrf: {
    enable: true,
    methods: ['POST', 'PUT', 'DELETE'],
    filterUris: ['/wxpay/notify'],
    cookieOptions: {
      httpOnly: false,
      sameSite: true,
      path: '/',
      maxAge: 7200
    }
  }
```

处理支付结果的方法

下面修改一下 *CheckoutController* 里的 *wxPayNotify* 这个方法：

```
  wxPayNotify ({ request }) {
    logger.warn('------------------------------------------------------------------------
-----')
    logger.info(request)

    const _payment = convert.xml2js(request._raw, {
      compact: true,
      cdataKey: 'value',
      textKey: 'value'
    }).xml

    const payment = Object.keys(_payment).reduce((accumulator, key) => {
      accumulator[key] = _payment[key].value
      return accumulator
    }, {})

    logger.info('支付结果：', payment)

    const paymentSign = payment.sign

    logger.info('结果签名：', paymentSign)

    delete payment['sign']

    const key = Config.get('wxpay.key')

    const selfSign = this.wxPaySign(payment, key)

    logger.info('自制签名：', selfSign)

    const return_code = paymentSign === selfSign ? 'SUCCESS' : 'FAIL'

    logger.debug('回复代码：', return_code)

    const reply = {
      xml: {
        return_code,
      }
    }

    return convert.js2xml(reply, {
      compact: true
    })
  }
```

方法收到的 *request* 里面，有支付结果数据，是在 *_raw* 这个属性里面。但是数据格式是 xml，所以我们要去转换一下它。转换之后的结果交给了 *payment*。

然后把支付结果里的签名提取出来交给了 *paymentSign*。接着我们要自己去算出一个签名，再去比较结果里的签名，看看是否一致。自己算签名的时候去掉了数据里的*sign*属性的值。然后用 *wxPaySign* 这个方法去计算签名。

根据比对签名的结果，确定 *return_code* 的值，它就是我们要给微信回复的信息。回复的信息应该是 xml 格式的数据，所以我们用 *js2xml* 方法转换了一下。方法最终 *return* 的东西，就是给微信支付系统做的回复。

项目代码

文章项目里的代码全部在 github ，*ninghao/ninghao-sandbox-v2* 这个仓库里，注意项目的分支是 *wxpay*。

克隆项目

```
git clone git@github.com:ninghao/ninghao-sandbox-v2.git
cd ninghao-sandbox-v2
git branch -a
git checkout -b wxpay remotes/origin/wxpay
npm install
cp .env.example .env
adonis key:generate
```

设置

打开项目以后，在项目根目录下的 .env 文件里，添加下面这些配置，注意把配置对应的值设置成你自己的：

```
WXPAY_APP_ID=wx58263149db20f28e
WXPAY_MCH_ID=1328508902
WXPAY_KEY=3fa1815e38bf4908sse12287eb6a7f92
WXPAY_NOTIFY_URL=https://sandbox.ninghao.net/wxpay/notify
```

支付流程

1. 用户在手机浏览器上，在应用的支付页面提交支付。
2. 应用请求微信的统一下单接口。
3. 微信支付系统返回跳转链接。
4. 应用页面重定向到微信支付返回的跳转链接。
5. 链接会调开用户的微信 App。
6. 用户在微信 App 确认并完成支付。
7. 用户被重定向到原来申请支付时的页面。
8. 页面可以引导用户查询微信支付订单状态。

开通 **H5** 支付

登录到微信支付后台，在产品中心，可以开通微信支付里的 H5 支付功能。

路由

start/routes.js

```
Route.get('checkout', 'CheckoutController.render')
Route.post('wxpay/notify', 'CheckoutController.wxPayNotify')
Route.post('checkout/pay', 'CheckoutController.pay')
Route.post('checkout/query', 'CheckoutController.query')
Route.get('checkout/completed', 'CheckoutController.completed')
```

之前我们添加的两条路由：

1. *checkout* 结账页面。
2. *wxpay/notify* 处理微信支付发送过来的支付结果通知。

相比之前，我们在应用里又添加了几条新的路由。

1. *checkout/pay* 用户点击支付页面上的确认支付按钮，会请求这个地址。它做的事主要是去组织 H5 支付需要的数据，然后请求微信支付统一下单接口，返回跳转链接。
2. *checkout/query* 支付完成以后，可以查询微信支付交易状态。比如我们可以在支付页面上显示一个对话框，提示用户查询微信交易的状态。
3. *checkout/completed* 如果查询的交易状态是 SUCCESS，可以把用户带到这个页面上，提示用户完成了交易。

H5 支付

结账页面视图

重新设计一下 *checkout* 页面视图，在实施扫码支付功能的时候，结账页面上会显示支付用的二维码。使用 H5 支付的时候，我们可以在这个结账页面上显示订单相关的信息，还有一个结账按钮，按下去可以用 Ajax 方式请求支付。

resources/views/commerce/checkout.edge

```
  <div class="container">
    <div class="row justify-content-center">
      <div class="col-md-4">
        <div class="card text-center mt-5 mx-3">
          <div class="card-body">
            <img src="{{ assetsUrl('wxpay.png') }}" alt="" class="w-50">
            <div class="card-price my-5 pb-5">
              <p class="card-amount"><small>¥</small>0.03</p>
              <p class="card-text text-muted"><small>订单金额</small></p>
            </div>
            <button data-csrf="{{ csrfToken }}" id="pay" class="btn btn-primary btn-bl
ock">确认支付</button>
          </div>
        </div>
      </div>
    </div>
  </div>
```

要注意的是在确认支付按钮上添加的 *id* 属性，等会儿我们可以在页面使用的自定义脚本里面，使用这个*id*属性的值来定位到这个按钮元素（*#pay*）。还有在按钮上的 *data-csrf* 属性，对应的值绑定了一个 *csrfToken*，在页面自定义脚本里面，可以利用这个自定义的*data*属性，得到*csrfToken*的值。应用（Adonis.js）对 POST 类型的请求会做 CSRF 保护，所以提交这种请求的时候，要带着应用生成的 CSRF Token 的值。

样式

视图需要点自定义的样式，在视图使用的布局（*layouts.main*）里面链接了一个自定义样式表（*main.css*），在这个样式表里你可以添加自定义的样式。

public/main.css：

```
.card-price {
  font-family: "Century Gothic", sans-serif;
  font-weight: bold;
}

.card-amount small {
  font-size: 16px;
}

.card-amount {
  font-size: 32px;
  padding: 0;
  margin: 0;
}
```

功能

在之前我们实施扫码支付的时候，功能代码主要都放在了 *CheckoutController* 这个控制器里了。

1. *wxPaySign* 用来生成微信支付签名。
2. *wxPayNotify* 处理微信支付发送过来的支付结果通知。
3. *render* 请求统一下单接口，生产二维码图像，交给 *checkout* 页面视图使用。

下面我们要对 *render* 方法做一些调整，只让它返回 *checkout* 页面视图，把其余的代码分割成几个方法。

render

显示结账页面。

```
render ({ view }) {
  return view.render('commerce.checkout')
}
```

orderToXML

要发送给微信支付接口的数据需要转换成 xml 格式，单独创建一个方法可以把 object 数据转换成 xml 格式。

```
 orderToXML (order, sign) {
   order = {
     xml: {
       ...order,
       sign
     }
   }

   // 转换成 xml 格式
   const xmlOrder = convert.js2xml(order, {
     compact: true
   })

   return xmlOrder
 }
```

xmlToJS

从微信支付那里得到的数据，格式是 xml，要在应用里使用的话需要转换成 object。这块代码可能会重复用到，所以单独定义一个*xmlToJS* 方法，功能就是把 xml 数据转换成 Object。

```
xmlToJS (xmlData) {
  const _data = convert.xml2js(xmlData, {
    compact: true,
    cdataKey: 'value',
    textKey: 'value'
  }).xml

  const data = Object.keys(_data).reduce((accumulator, key) => {
    accumulator[key] = _data[key].value
    return accumulator
  }, {})

  return data
}
```

pay

然后再把之前在 *render* 方法里的其余的代码放到 *pay* 这个方法里面。它是 *checkout/pay* 路由使用的处理请求用的方法。

```
async pay ({ request, session }) {
  logger.info('请求支付 ------------------------')

  // 公众账号 ID
  const appid = Config.get('wxpay.appid')

  // 商户号
  const mch_id = Config.get('wxpay.mch_id')

  // 密钥
  const key = Config.get('wxpay.key')

  // 商户订单号
  const out_trade_no = moment().local().format('YYYYMMDDHHmmss')
  session.put('out_trade_no', out_trade_no)

  // 商品描述
  const body = 'ninghao'

  // 商品价格
  const total_fee = 3

  // 支付类型
  const trade_type = 'MWEB'

  // 用户 IP
  const spbill_create_ip = request.header('x-real-ip')

  // 商品 ID
  const product_id = 1
```

```
    // 通知地址
    const notify_url = Config.get('wxpay.notify_url')

    // 随机字符
    const nonce_str = randomString.generate(32)

    // 统一下单接口
    const unifiedOrderApi = Config.get('wxpay.api.unifiedorder')

    let order = {
      appid,
      mch_id,
      out_trade_no,
      body,
      total_fee,
      trade_type,
      product_id,
      notify_url,
      nonce_str,
      spbill_create_ip
    }

    const sign = this.wxPaySign(order, key)

    const xmlOrder = this.orderToXML(order, sign)

    // 调用统一下单接口
    const wxPayResponse = await axios.post(unifiedOrderApi, xmlOrder)

    const data = this.xmlToJS(wxPayResponse.data)

    logger.debug(data)

    return data.mweb_url
  }
```

trade_type 设置的是交易类型，跟扫码支付（*NATIVE*）不同的是，我们把它设置成了 *MWEB*：

```
trade_type = 'MWEB'
```

还有在方法里，我们把生成的订单号保存在了用户的 session 里面了，这样在后面调用微信支付查询订单接口的时候，可以从 session 里面读取订单号，然后组织好查询需要带的数据。

```
session.put('out_trade_no', out_trade_no)
```

请求支付

在结账页面上有个确认支付按钮，按一下它可以请求支付。可以使用表单或者 Ajax 的形式去执行这个动作。下面我们要使用 Ajax 的方法去请求支付，在页面使用的自定义脚本文件里面，可以添加几行代码：

public/main.js

```
(function() {
 'use strict'
  // 自定义脚本
}())
```

把代码放在上面的立即执行函数（IIFE）里面：

```
  const _csrf = $('#pay').data('csrf')

  $('#pay').click(() => {
    $.ajax({
      url: '/checkout/pay',
      method: 'POST',
      data: {
        _csrf
      },
      success: (response) => {
        console.log(response)
        if (response) {
          window.location.href = response
        }
      },
      error: (error) => {
        console.log(error)
      }
    })
  })
```

先从 *#pay* 按钮上得到了 *csrfToken* 的值。然后找到页面上的 *#pay* 元素，监听它的点击事件（*click*）。点击了确认支付按钮，就会使用 jQuery 的 Ajax 方法去发出请求。如果你做的是前端应用，可以使用其它的 HTTP 客户端，比如 axios（浏览器与 Node.js 都可以使用它）。效果都差不多，就是对指定的地址发出各种不同类型的 HTTP 请求。

请求成功会调用 *success* 方法，得到的响应叫 *response*，它的值应该就是支付跳转用的地址。下面这行代码，会把用户带到得到的这个支付跳转地址上：

```
window.location.href = response
```

在*checkout*页面点一下确认支付按钮，就会请求*checkout/pay* 这个地址，这个地址请求的处理方法用的是 *CheckoutController* 里的 *pay* 方法。在这个方法里面，我们组织好了请求带的数据，然后对微信支付系统的统一下单接口发出请求，并且返回了请求得到的 *mweb_url*，它的值就是支付要跳转的地址。

打开应用日志 *app.log*，里面会出现请求统一下单接口返回的数据， *mweb_url* 就是我们需要用的那个跳转地址：

```
23:45:56 DEBUG - { return_code: 'SUCCESS',
 return_msg: 'OK',
 appid: 'wx58263139db20f28e',
 mch_id: '1528508902',
 nonce_str: 'fnAUH4QVn7L2yrQo',
 sign: '9FF86FF3EA9C99D78A2C832638E62649',
 result_code: 'SUCCESS',
 prepay_id: 'wx20180202234556a1bff266860143681661',
 trade_type: 'MWEB',
 mweb_url: 'https://wx.tenpay.com/cgi-bin/mmpayweb-bin/checkmweb?prepay_id=wx201802022
34556a1bff266860143681661&package=3740852957' }
```

试验

在手机设备上打开支付页面，按一下 确认支付，应该就会调用微信支付 App 进行支付了。

查询订单状态

使用我们自己系统内部的订单号（*out_trade_no*），或者微信支付生成的订单号（*transaction_id*），调用微信支付订单查询接口，可以查询交易的状态。

查询对话框视图

用户使用 H5 方式支付完成以后会被重定向到原来发起支付时的页面。就是我们的结账页面，在这个页面上，可以显示一个对话框，提示用户查询交易状态（*trade_state*），如果状态是 *SUCCESS* ，可以再把用户带到一个完成页面。

resources/views/commerce/checkout.edge：

```
  <div class="modal" id="modal-query">
    <div class="modal-dialog">
      <div class="modal-content">
        <div class="modal-header">
          <h5 class="modal-title">支付结果</h5>
          <button class="close" type="button" data-dismiss="modal">
            <span>&times;</span>
          </button>
        </div>
        <div class="modal-body">
          <p>支付完成以后，请再确定一下支付结果。</p>
        </div>
        <div class="modal-footer">
          <button id="order-query" class="btn btn-primary btn-block" data-csrf="{{ csr
fToken }}">
            支付成功
          </button>
        </div>
      </div>
    </div>
  </div>
```

> 视图用到了 Bootstrap 框架提供的Modal 组件。

页面脚本

按下结账页面的 确认支付，成功得到了响应以后，可以再弹出对话框视图。支付完成以后再次跳转到这个结账页面的时候，iOS 设备会刷新这个页面，所以我们需要一种方法记住对话框的开启状态。这里我们把这个开启状态保存在了用户设备的 LocalStorage 里面。

public/main.js：

```
// 找到页面上的对话框元素
const modalQuery = $('#modal-query')

// 对话框完全隐藏时，设置它的开启状态为 hide
modalQuery.on('hidden.bs.modal', () => {
  localStorage.setItem('#modal-query', 'hide')
})

// 页面显示时，读取对话框的开启状态
const modalQueryState = localStorage.getItem('#modal-query')

// 如果对话框开启状态为 show，我们就显示它
if (modalQueryState === 'show') {
  modalQuery.modal()
}

// 执行查询订单，交易状态为成功，就把用户带到 /checkout/completed 页面。
$('#order-query').click(() => {
  $.ajax({
    url: '/checkout/query',
    method: 'POST',
    data: {
      _csrf
    },
    success: (response) => {
      switch (response.trade_state) {
        case 'SUCCESS':
          window.location.href = '/checkout/completed'
          break
        default:
          console.log(response)
      }
    },
    error: (error) => {
      console.log(error)
    }
  })
})
```

完成页面视图

resources/views/commerce/completed.edge：

```
@layout('layouts.main')

@section('content')
  <div class="container">
    <div class="jumbotron mt-3">
      <h1 class="dispay-4">成功啦！</h1>
      <p class="lead">您的订单已经完成了。</p>
      <p class="head">
        <a href="/checkout" class="btn btn-primary my-2">返回</a>
      </p>
    </div>
  </div>
@endsection
```

订单查询功能

实现订单查询功能，主要的代码放在了 *CheckoutController* 里的 *query* 这个方法里面，它是 *checkout/query* 地址的处理方法。按下对话框上的 支付成功 按钮，请求的就是这个地址。

query

在方法里，准备好调用微信支付订单查询接口需要的数据，注意我们在用户的 Session 里面读取了之前在 *pay* 方法里保存的 *out_trade_no* 的值。在实际的应用中，我们的应用收到了微信支付结果以后，可以把结果里的微信支付订单号（*transaction_id*）保存在数据库里。这样在查询交易状态的时候，也可以根据这个微信支付订单号去查询交易状态。

```
 async query ({ session }) {
   logger.info('请求查询 ------------------------')
   // 公众账号 ID
   const appid = Config.get('wxpay.appid')

   // 商户号
   const mch_id = Config.get('wxpay.mch_id')

   // 密钥
   const key = Config.get('wxpay.key')

   // 商户订单号
   const out_trade_no = session.get('out_trade_no')

   // 随机字符
   const nonce_str = randomString.generate(32)

   // 查询订单接口
   const orderQueryApi = Config.get('wxpay.api.orderquery')

   const order = {
     appid,
     mch_id,
     out_trade_no,
     nonce_str
   }

   const sign = this.wxPaySign(order, key)
   const xmlOrder = this.orderToXML(order, sign)
   const wxPayQueryResponse = await axios.post(orderQueryApi, xmlOrder)
   const result = this.xmlToJS(wxPayQueryResponse.data)

   logger.debug(result)

   return result
 }
```

completed

添加一个 *completed* 方法，显示一个 *commerce.completed* 视图。

```
completed ({ view }) {
  return view.render('commerce.completed')
}
```

数据

调用订单查询接口返回的数据：

```
09:25:42 DEBUG - { return_code: 'SUCCESS',
 return_msg: 'OK',
 appid: 'wx58263139db20f28e',
 mch_id: '1228508902',
 nonce_str: 'QqwBIqZDv4ogsgrg',
 sign: '68C38CB0738F20D8B2DB6F135121DCA6',
 result_code: 'SUCCESS',
 openid: 'osbKIjtJPwZzfMea5X7Q_q2tH_EU',
 is_subscribe: 'Y',
 trade_type: 'MWEB',
 bank_type: 'CFT',
 total_fee: '3',
 fee_type: 'CNY',
 transaction_id: '4200000063201802046578989993',
 out_trade_no: '20180204092514',
 attach: undefined,
 time_end: '20180204092536',
 trade_state: 'SUCCESS',
 cash_fee: '3' }
```

项目代码

应用的代码在 *ninghao/ninghao-sandbox-v2* 这个仓库的 ***wxpay-h5*** 这个分支上。

公众号支付介绍

用户在微信应用内部的浏览器打开我们应用的支付页面，页面会去请求一个登录凭证（code），通过这个登录凭证又会得到用户的 Openid，这个 Openid 就是加密之后的用户的微信号。

这个时候用户就可以按一下支付按钮请求支付，我们的应用会组织好好需要调用统一下单带的数据，去获取到一个微信支付的预支付。然后我们的应用会组织好公众号支付需要的一些参数， 再把它们返回给应用的前端，再由前端去调用微信的 JSAPI 来请求支付。请求支付会调起用户的微信支付功能，用户可以确认并且完成支付 。

支付成功以后，会把用户带回到我们指定的页面上，在页面上可以引导用户查询交易状态，如果查询的结果是 SUCCESS，可以把用户带到一个成功提示的页面。这个就是我们要实现的一个微信支付的公众号支付功能。

配置

在公众平台的功能设置里，可以设置一下网页授权域名。

在微信支付的开发设置里，设置一下公众号支付里的支付授权目录。

微信用户的 OpenID

实现公众号微信支付的关键一步就是获取到用户的 OpenID。这个 OpenID 的值会包含在用户的相关信息里面，在公众号里你可以向用户申请授权得到用户的信息。用户同意以后（如果请求得到用户相关信息会提示授权，如果只是请求 OpenID 的话不会出现授权提示），微信会给我们返回微信用户的相关信息，在这个相关信息里面，就会有这个 OpenID。

OpenID 其实就是加密之后的用户的微信号，在同一个应用之下，用户的 OpenID 是唯一的。不过要注意你可能会在同一主体下创建多个应用，比如网站应用，或者一些小程序什么的。微信用户在同一主体下的不同的应用里面，他们的 OpenID 是不一样的。

如果你想识别唯一的用户，那你需要开通并认证微信的开放平台，然后在开放平台去绑定同一主体下的不同的应用。这样你就有能力得到用户的 UnionID，你可以使用这个 UnionID 的值来确定某个特定的用户，因为在你绑定在开放平台里的不同的应用下面，用户的 UnionID 的值是一样的。

登录凭证（**code**）

想要获取用户的信息需要使用 access_token 去换，得到这个 access_token 得先有个 code，这个 code 就是我们说的登录凭证。获取到这个 code 我们得先去准备一些数据。先组织好获取登录凭证需要的一些数据，带着这些数据把用户重定向到微信授权地址上。如果没问题，会被重定向回我们自己指定的页面上，重定向回来的时候，地址上就会带着我们需要用的登录凭证。

配置

在项目里可以先去创建一个新的配置文件，文件的名字是 wexin.js，在里面可以存储一些跟微信相关的配置数据。这些数据我们会在项目的其它地方用到。下面是文件里面的内容，暂时添加了一个 *open.auth*，它的值是微信登录授权用的一个地址。在请求获取登录凭证的时候会用到这个地址。

config/weixin.js：

```
'use strict'

module.exports = {
  open: {
    auth: 'https://open.weixin.qq.com/connect/oauth2/authorize'
  }
}
```

获取登录凭证

获取登录凭证需要的代码，我们可以放在 CheckoutController 里面的 render 这个方法里。它个方法目前渲染的就是结账页面（*/checkout*），用户在访问这个页面的时候，可以试着去获取登录凭证。

app/Controllers/Http/CheckoutController.js：

```
/**
  * 结账页面。
  * @param  {Object}  view
  * @return 渲染结账页面视图。
  */
 async render ({ view, request, response, session }) {
   /**
    * 获取申请 access_token 需要的 code。
    */
   const code = request.input('code')
   logger.debug('code: ', code)

   const appid = Config.get('wxpay.appid')

   if (!code) {
     const redirect_uri = `https://${ request.hostname() }${ request.url() }`
     const response_type = 'code'
     const scope = 'snsapi_base'

     const openAuthUrlParams = {
       appid,
       redirect_uri,
       response_type,
       scope
     }
     const openAuthUrlString = queryString.stringify(openAuthUrlParams)

     const openAuthApi = Config.get('weixin.open.auth')
     const openAuthUrl = `${ openAuthApi }?${ openAuthUrlString }`

     return response.redirect(openAuthUrl)
   }

   return view.render('commerce.checkout')
 }
```

在上面的*render*方法里， 我们先把 *view*，*request*，*response*，还有*session* 解构出来。方法的一开始，添加了一个 *code*，表示登录凭证，使用 *request.input* 方法可以得到请求里面包含的数据。成功得到登录凭证返回到这个页面的时候，地址里面会包含 *code* 这个数据的值，它就是我们需要的登录凭证。

```
const code = request.input('code')
logger.debug('code: ', code)
```

然后又去判断了一下，如果访问 *checkout* 页面的时候，没有得到 *code* 的值，就去组织好请求登录凭证需要的数据，然后重定向到微信授权地址。重定向的时候，用到了 *response.redirect*。

```
return response.redirect(openAuthUrl)
```

获取登录凭证需要的数据

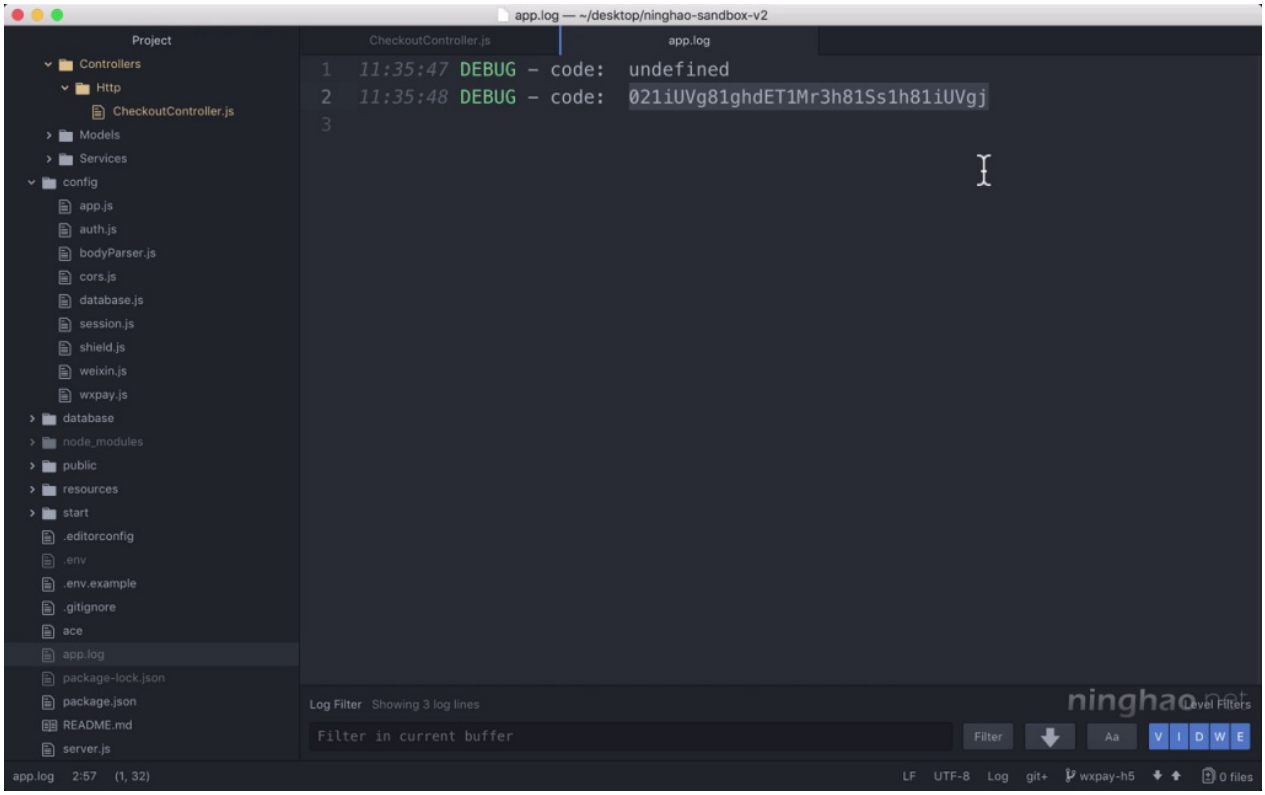
请求登录凭证需要几个数据，这些数据要使用地址查询符（Query String）的形式发送到微信授权地址。 下面是几个必要的数据：

- *appid* 公众号应用 ID。
- *redirect_uri* 重定向到的地址，得到登录凭证以后会被重定向到这里设置的地址上。
- *response_type* 这里要设置成 code，因为我们需要的是登录凭证。
- *scope* 可以是 *snsapi_base* 或者 *snsapi_userinfo*，因为我们只想得到微信用户的 OpenID，所以可以设置成 *snsapi_base*，如果想获取用户的其它的信息，可以设置成 *snsapi_userinfo*。

我们把上面这些数据放在了一个叫 *openAuthUrlParams* 的对象里，然后又把这个对象转换成了地址查询符的形式。把转换之后的数据跟微信授权地址拼接在一起，组织成一个最终要重定向到的地址（openAuthUrl）。

测试

把应用的 */checkout* 页面地址发送给微信用户，用户在微信内部浏览器打开这个页面以后，我们再回到项目里面检查 *app.log* 这个日志文件。你会发现，第一次访问 */checkout* 页面的时候，要求输出的 *code*（登录凭证）的值是 *undefined*，这样就会把用户重定向到微信授权地址，成功以后，再次返回*/checkout* 页面，这时候请求里面就会带着 *code* 的值了。所以第二条记录里面，*code* 是有值的。



app.log

```
10:08:25 DEBUG - code:  undefined
10:08:39 DEBUG - code:  011O1Na51dTrvM1Z7Ja51XNOa51O1Nai
```

获取 access_token

有了登录凭证（code）以后，下一步就可以去请求得到 access_token，这里面会包含 access_token 本身，还有我们需要的微信用户的 OpenID。

配置

先添加两个配置，一个是 *appSecret*，它是公众账号的应用密钥，你在公众平台的管理后台可以得到这个密钥的值。我们可以把这个密钥的值放在项目的 *.env* 文件里面，这里我给它起了一个名字叫 *WXMP_APP_SECRET*，也就是在 *.env* 文件里，添加一个叫 *WXMP_APP_SECRET* 的东西，对应的值就是公众平台里面的应用的密钥。

然后再修改配置文件，添加一个 *appSecret*，对应的值可以使用 *Env.get* 去得到*.env* 文件里面添加的环境变量的值。在 *weixin.js* 这个配置文件里，我又添加了一个 *api.accessToken*，它的值是请求 *access_token* 的时候需要用的接口的地址。

config/weixin.js：

```
'use strict'

const Env = use('Env')

module.exports = {
  appSecret: Env.get('WXMP_APP_SECRET'),
  open: {
    auth: 'https://open.weixin.qq.com/connect/oauth2/authorize'
  },
  api: {
    accessToken: 'https://api.weixin.qq.com/sns/oauth2/access_token'
  }
}
```

控制器

请求 *access_token* 需要的代码我们也可以把它们放在 *CheckoutController* 里的 *render* 方法里面。要做的就是先去组织好请求*access_token* 需要的数据，把数据转换成地址查询符的形式，然后把转换之后的数据跟 *access_token* 接口地址拼接在一起，再用一个 HTTP 客户端（比如 axios）去请求这个地址。得到的响应里面，就会包含 *access_token* 相关的数据。

改造 *render* 方法，添加下面这些代码：

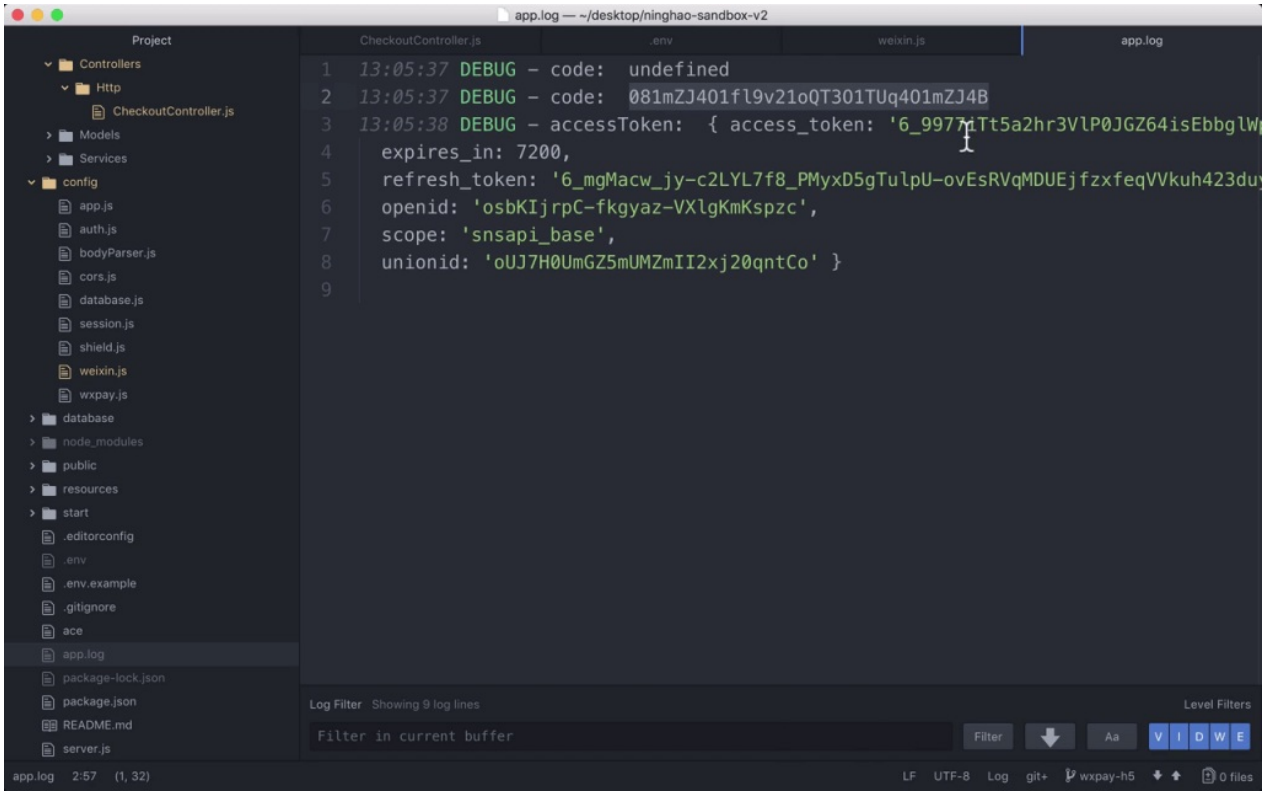
```
/**
 * 获取 access_token
 */
const secret = Config.get('weixin.appSecret')
const grant_type = 'authorization_code'

const accessTokenUrlParams = {
  appid,
  secret,
  grant_type,
  code
}

const accessTokenUrlString = queryString.stringify(accessTokenUrlParams)
const accessTokenApi = Config.get('weixin.api.accessToken')
const accessTokenUrl = `${ accessTokenApi }?${ accessTokenUrlString }`

const wxResponse = await axios.get(accessTokenUrl)
logger.debug('accessToken: ', wxResponse.data)
session.put('accessToken', wxResponse.data)
```

请求 **access_token** 需要的数据

- *appid* 公众号应用 ID。
- *secret* 公众号应用密钥。
- *grant_type* 授权类型，这里要设置成 authorization_code。
- *code* 登录凭证。

我把上面这些数据放在了一个叫 *accessTokenUrlParams* 的对象里，又把这个对象转换成了地址查询符形式的字符串，然后把它跟 *access_token* 拼接在了一起，接着用 *axios* 的 *get* 方法去请求最终组织好的地址。得到的响应给它起了个名字叫 *wxResponse*，它里面的 *data* 属性的数据就是我们需要的东西。

我们把需要的数据输出到了应用的日志文件里了，然后又把这个数据放在了 *session* 里面存储起来了。这样在支付的时候，可以读取用户 *session* 里的数据，获取到我们需要使用的 *openid* 的值。



app.log

```
15:09:08 DEBUG - accessToken:
{
 access_token: '7_1_zu2EpTHYS6gNn88xZyY6zfIypl-69QVXh3Ya7pcFtVV9P6IIE3Qj182S0f27lowQeO
OpAj8U4r4nEQpAmiFeDu_JKVK7I6g0EEJElyl6o',
 expires_in: 7200,
 refresh_token: '7_J-g1AtVc0K89_9v8XME_kFhhC4Nes5Jt3YNR51BJ-XPKykQ77z6QsG-f-pf9vfgcaMx
fUm43o9ymx-kvlkocH_6KhGUf3OPHt3XssU4jNNc',
 openid: 'osbKIjtJPwZzfMea5X7Q_q2tH_EU',
 scope: 'snsapi_base',
 unionid: 'oUJ7H0caaUFoviKZrU9-DFHvtOdo'
}
```

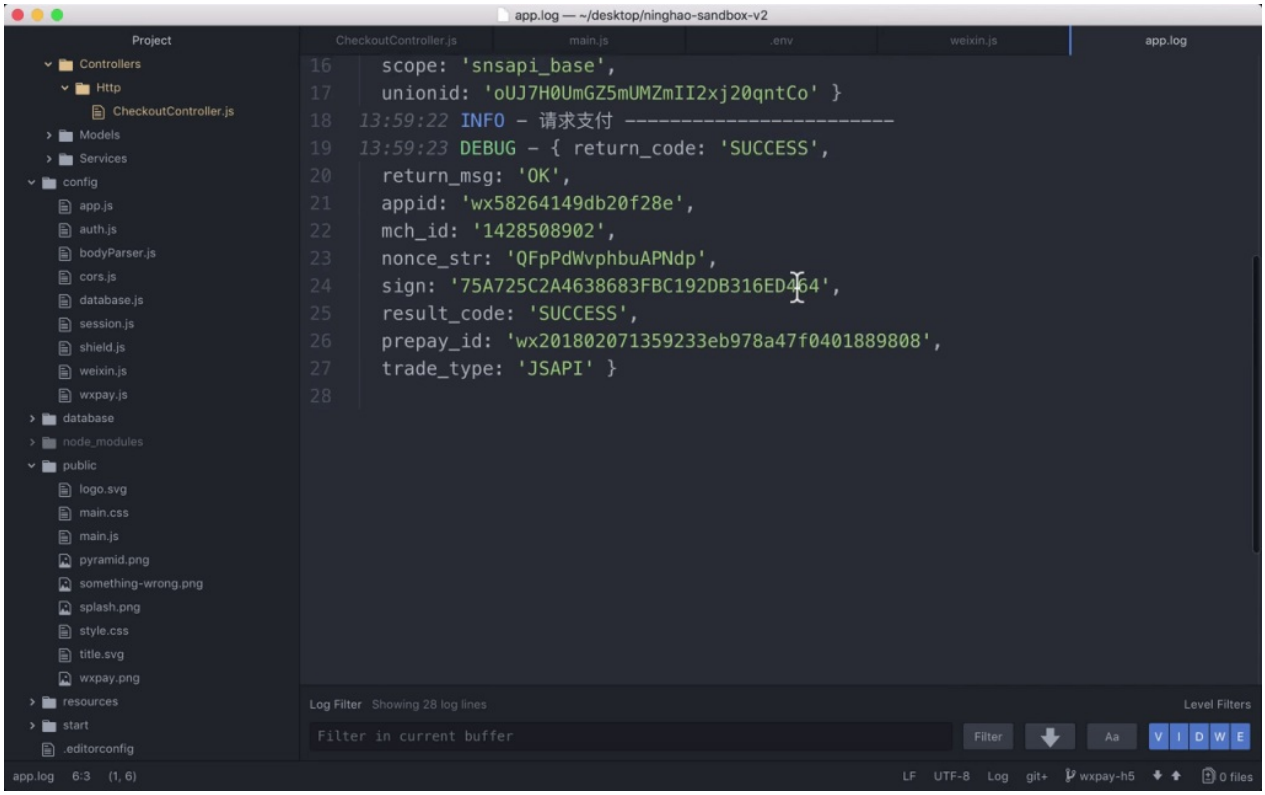
统一下单

跟其它的微信支付方式一样，支付的时候我们得去请求微信支付的统一下单接口。这里需要改动几个地方，请求统一下单的代码是在 *CheckoutController* 里的 *pay* 这个方法里面。

app/Controllers/Http/CheckoutController.js：

```
async pay ({ request, session }) {
  ...
  /** 支付类型 */
  const trade_type = 'JSAPI'
  ...
  /** 微信用户 openid */
  const accessToken = session.get('accessToken')
  const openid = accessToken.openid
  ...
  /**
   * 准备支付数据。
   */
  let order = {
   ...
   openid
  }
  ...
}
```

使用公众号支付方式的时候，调用统一下单接口，*trade_type* 的值应该设置成 *JSAPI*。还有就是需要带着微信用户的 *openid*，这个值我们之前已经得到了，并且把它存储在了用户的 *session* 里了，这里我们用 *session.get* 得到了存储在 *session* 里的 *accessToken*，它里面的 *openid* 属性的值就是我们需要的 *openid*。


app.log — ~/desktop/ninghao-sandbox-v2
Project
Controllers
Http
CheckoutController.js
Models
Services
config
app.js
auth.js
bodyParser.js
cors.js
database.js
session.js
shield.js
weixin.js
wxpay.js
database
node_modules
public
logo.svg
main.css
main.js
pyramid.png
something-wrong.png
splash.png
style.css
title.svg
wxpay.png
resources
start
.editorconfig
CheckoutController.js
main.js
.env
weixin.js
app.log
scope: 'snsapi_base',
unionid: 'oUJ7H0UmGZ5mUMZmII2xj20qntCo' }
13:59:22 INFO - 请求支付 ----------
13:59:23 DEBUG - { return_code: 'SUCCESS',
return_msg: 'OK',
appid: 'wx58264149db20f28e',
mch_id: '1428508902',
nonce_str: 'QFpPdWvphbuAPNdp',
sign: '75A725C2A4638683FBC192DB316ED464',
result_code: 'SUCCESS',
prepay_id: 'wx201802071359233eb978a47f0401889808',
trade_type: 'JSAPI' }
Log Filter Showing 28 log lines
Level Filters
Filter in current buffer
Filter
Aa
V I D W E
app.log 6:3 (1, 6)
LF UTF-8 Log git+ wxpay-h5
0 files

公众号支付

同样是在这个 *pay* 这个方法里，继续去添加一些代码。去准备 JSAPI 需要的一些参数数据，组织好这些数据以后，把它响应给应用的前端，在应用的前端可以调用 JSAPI 去请求支付。

```
/**
 * JSAPI 参数
 */
const timeStamp = moment().local().unix()
const prepay_id = data.prepay_id

let wxJSApiParams = {
  appId: appid,
  timeStamp: `${ timeStamp }`,
  nonceStr: nonce_str,
  package: `prepay_id=${ prepay_id }`,
  signType: 'MD5'
}

const paySign = this.wxPaySign(wxJSApiParams, key)

wxJSApiParams = {
  ...wxJSApiParams,
  paySign
}

/**
 * 为前端返回 JSAPI 参数，
 * 根据这些参数，调用微信支付功能。
 */
return wxJSApiParams
```

JSAPI 参数数据

- *appId* 公众号应用 ID。
- *timeStamp* 时间戳。
- *nonceStr* 随机数。
- *package* 它的值应该使用这样的形式 *`prepay_id=${ prepay_id }`*，*prepay_id* 的值就是请求统一下单接口返回来的数据里面的预支付的 *id* 号。
- *signTyp_e* 签名类型，可以是 *_MD5*。
- *sign* 签名。根据微信支付提供的规则算出签名的值。这个签名的值要根据一些特定的数据生成，这里这些数据就是 JSAPI 需要的一些参数的值（除 *sign* 本身）。

最后得到的参数数据的名字叫*wxJSApiParams*，我们把这个数据响应给了前端。在前端，点击确认支付以后，会用 Ajax 的形式请求应用的支付接口，这个接口会使用*CheckoutController*里的这个*pay*方法来处理，所以这个方法里返回的东西，就是给前端提供的响应数据。

请求支付

在结账页面上的自定义脚本里面，添加一个新的方法（*public/main.js*）：

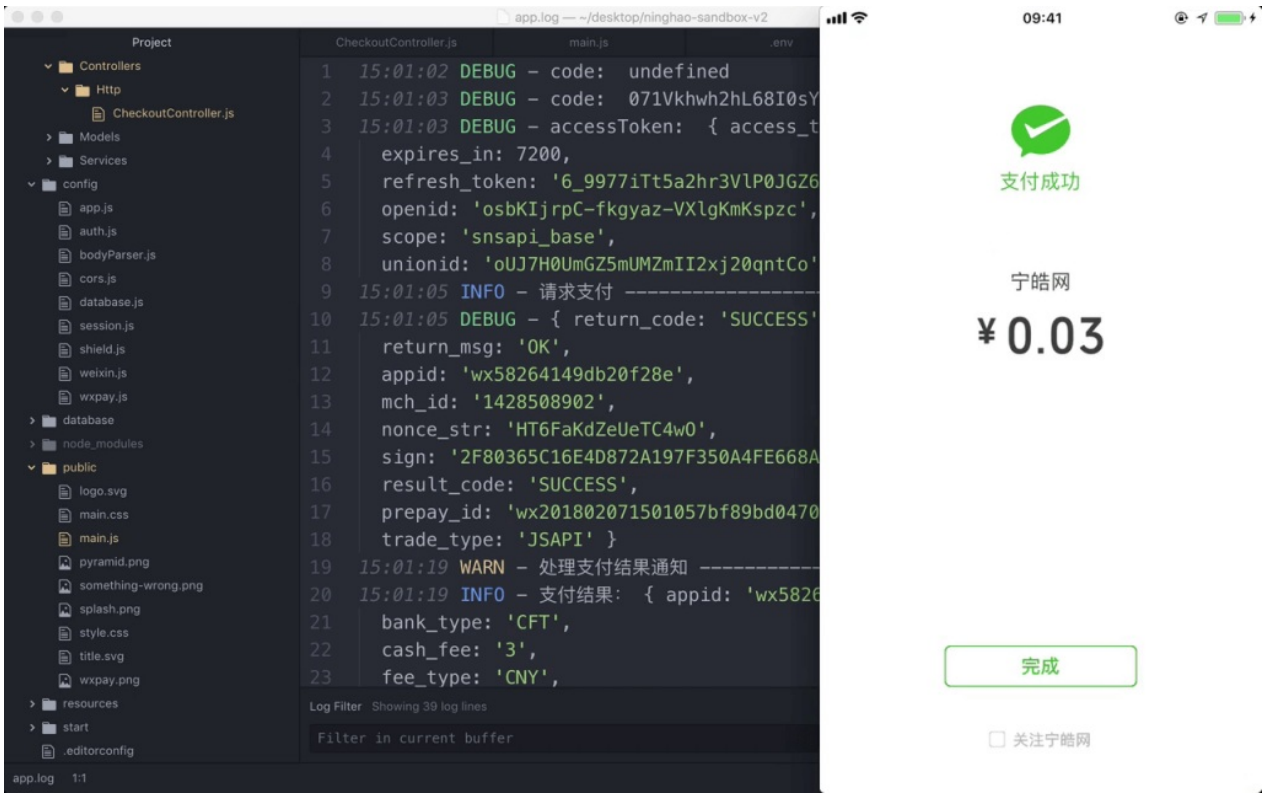
```
/**
 * JSAPI 微信支付。
 *
 * @param  {Object} wxJSApiParams 支付需要的参数数据。
 */
const wxPay = (wxJSApiParams) => {
  WeixinJSBridge.invoke(
    'getBrandWCPayRequest',
    wxJSApiParams,
    (response) => {
      console.log(response)
    }
  )
}
```

改造前端确认支付按钮的事件处理（*public/main.js*）：

```
/**
 * 请求支付。
 */
$('#pay').click(() => {
  $.ajax({
    url: '/checkout/pay',
    method: 'POST',
    data: {
      _csrf
    },
    success: (response) => {
      console.log(response)
      if (response) {
        modalQuery.modal()
        localStorage.setItem('#modal-query', 'show')

        wxPay(response)
      }
    },
    error: (error) => {
      console.log(error)
    }
  })
})
```

上面这块请求支付的代码，就是用户点击了结账页面上的确认支付按钮以后的事件处理。这里我们用 ajax 请求 */checkout/pay* 页面，页面用的处理方法是*CheckoutController* 里的 *pay* 方法，这个方法会给我们准备好 JSAPI 需要的参数数据。得到的这些数据就是 *response*。

然后我们把请求支付的功能放在了一个自定义的方法里面，名字是 *wxPay*，在这个方法里使用了微信内部浏览器特有的 *WeixinJSBridge.invoke* 请求支付。


app.log — ~/desktop/ninghao-sandbox-v2
Project
CheckoutController.js
main.js
.env
Controllers
Http
CheckoutController.js
Models
Services
config
app.js
auth.js
bodyParser.js
cors.js
database.js
session.js
shield.js
weixin.js
wxpay.js
database
node_modules
public
logo.svg
main.css
main.js
pyramid.png
something-wrong.png
splash.png
style.css
title.svg
wxpay.png
resources
start
.editorconfig
15:01:02 DEBUG - code: undefined
15:01:03 DEBUG - code: 071Vkhwh2hL68I0sY
15:01:03 DEBUG - accessToken: { access_t
expires_in: 7200,
refresh_token: '6_9977iTt5a2hr3VlP0JGZ6
openid: 'osbKIjrpC-fkgyaz-VXlgKmKspzc',
scope: 'snsapi_base',
unionid: 'oUJ7H0UmGZ5mUMZmII2xj20qntCo'
15:01:05 INFO - 请求支付
15:01:05 DEBUG - { return_code: 'SUCCESS'
return_msg: 'OK',
appid: 'wx58264149db20f28e',
mch_id: '1428508902',
nonce_str: 'HT6FaKdZeUeTC4wO',
sign: '2F80365C16E4D872A197F350A4FE668A
result_code: 'SUCCESS',
prepay_id: 'wx201802071501057bf89bd0470
trade_type: 'JSAPI' }
15:01:19 WARN - 处理支付结果通知
15:01:19 INFO - 支付结果: { appid: 'wx5826
bank_type: 'CFT',
cash_fee: '3',
fee_type: 'CNY',
Log Filter Showing 39 log lines
Filter in current buffer
app.log 1:1
09:41
支付成功
宁皓网
¥0.03
完成
关注宁皓网

项目代码

应用的代码在 *ninghao/ninghao-sandbox-v2* 这个仓库的 *wxpay-jsapi* 这个分支上。

相关资源

1. 《微信支付：开发准备与扫码支付》
2. 《微信支付：H5 移动端支付》
3. 《微信支付：公众号支付》
4. 《微信支付：小程序支付》
5. 《Node.js》
6. 《在互联网访问本地开发环境》
7. 微信支付开发文档
8. 微信支付平台
9. 微信公众平台
10. 微信开发平台