前几天在做性能测试时候问了一些登录凭证的问题,我在旁边忽然想着等都凭证是如何弄出来的，于是开始对登陆凭证开始查找资料,并且写了一些demo，现在写一篇博客分享。

# 为什么需要登录凭证

在web开发中，使用的最多的协议http，但是http协议是一个无效的协议。

举个例子

我们登录一个网站 [www.lexiangla.com](http://www.chenjiawei.com)

登录的时候需要输入账号和密码,比如用户名为chenjiawei，密码为123456

登录成功之后，我以admin的身份去访问其他二点用户数据和资源，还是通过http请求去访问

lexiangla的服务器会问:你谁啊你?

chenjiawei说:我是chenjiawei啊!

lexiangla的服务器:怎么证明你是chenjiawei啊!

chenjiawei说:这。。。http请求没有告诉你吗!而且我刚刚登录请求过了呀!

lexiangla的服务器: http的每一次请求对我来说都是单独的请求，和之前请求过有什么关系!

# Cookie

Cookie(复数是Cookies),类型为小型文本文件,某些网站为了辨别用户身份而存储在本地终端上数据

Cookie总是保存在客户端中，按在客户端中的存储位置，Cookie可以分为内存Cookie和硬盘Cookie

内存Cookie由浏览器维护，保存在内存中，浏览器一关就没了

硬盘Cookie保存在硬盘中，有一个过期时间，手动清理或者过期时间到了，才会被清理

客户端设置cookie

写一段代码，把代码在浏览器里打开

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h2>Cookie</h2>

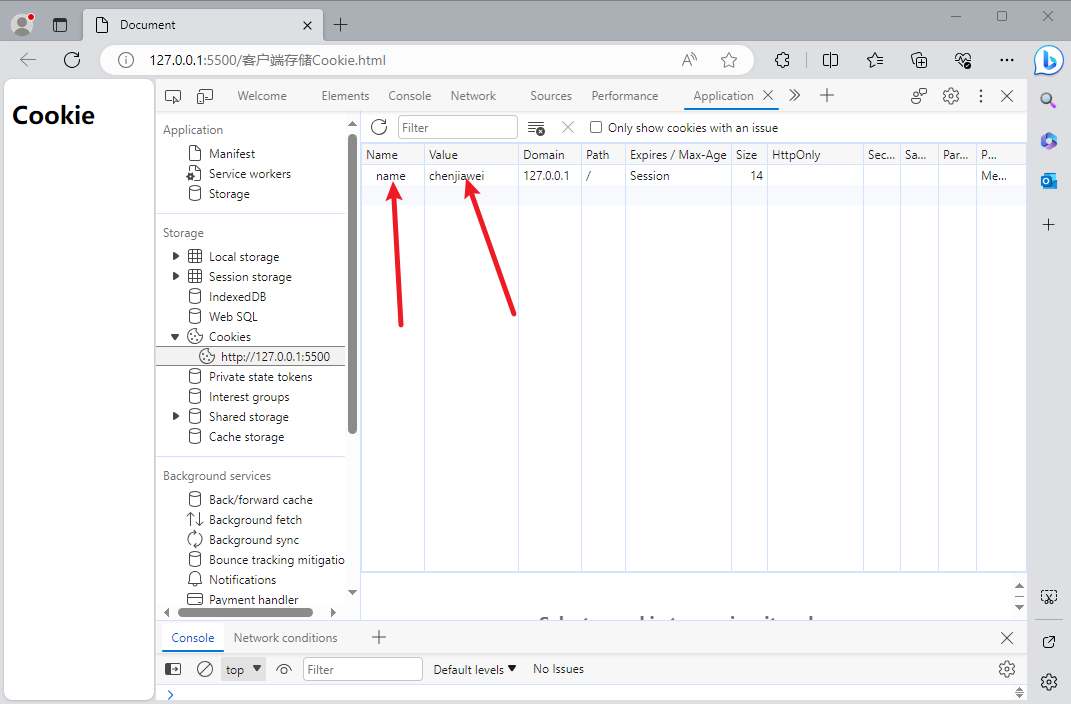
    <script>

        document.cookie = "name=chenjiawei;"

    </script>

</body>

</html>

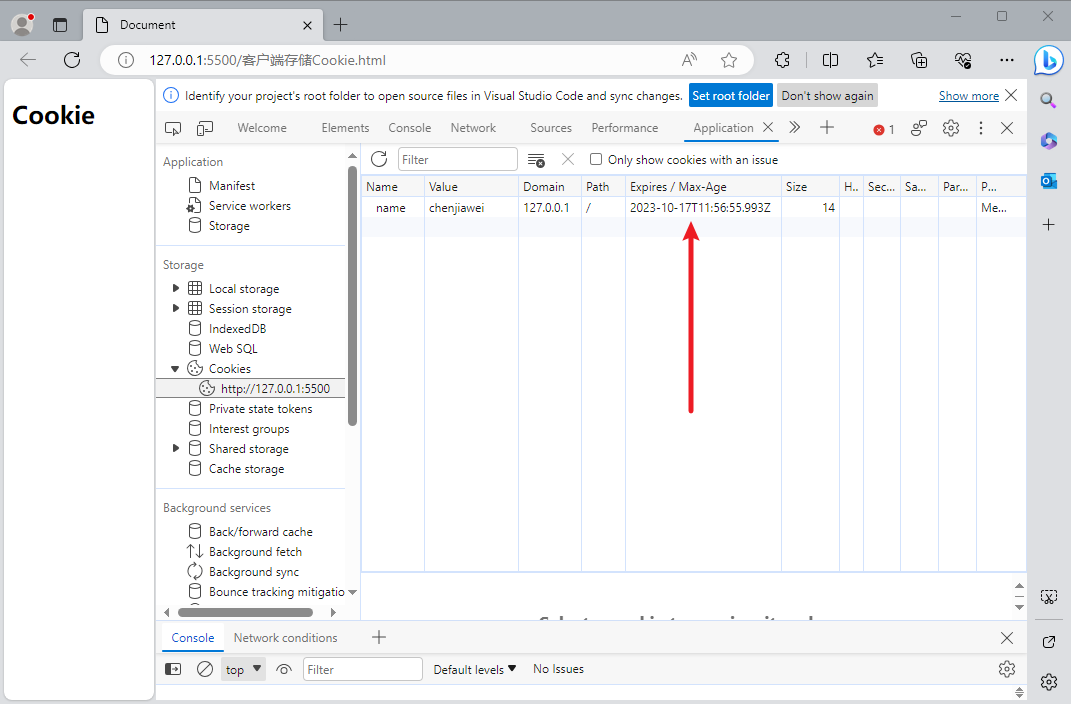


在浏览器里可以看到本地存储的内存cookie，但是在实际项目中很少使用内存cookie

在设置cookie时可以继续添加过期时间属性，cookie就会变成硬盘cookie

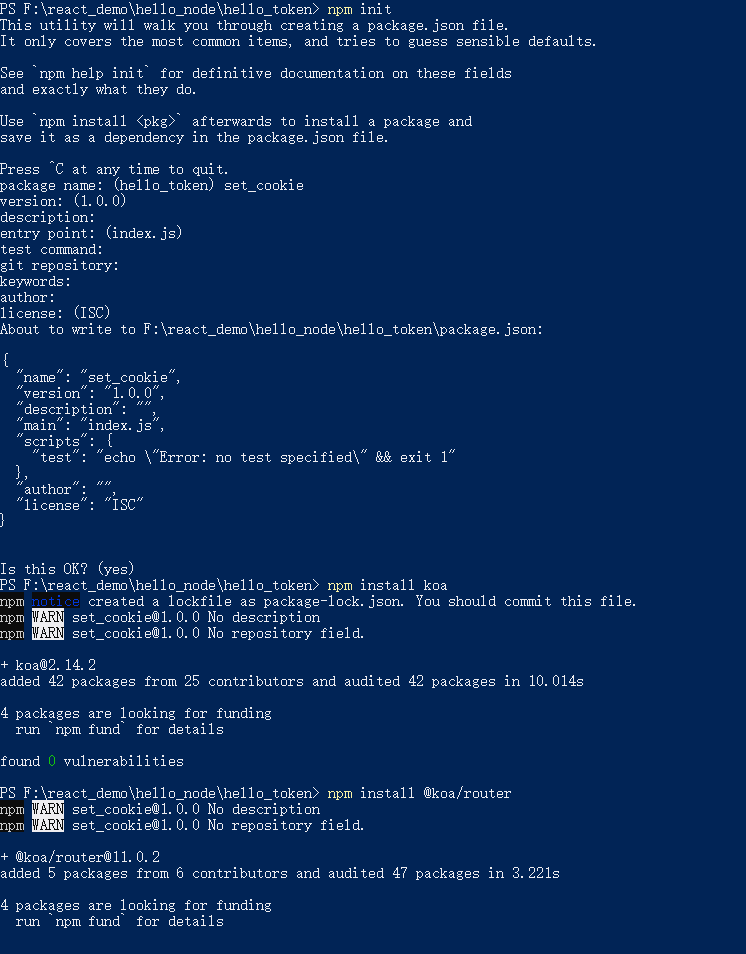
document.cookie = "name=chenjiawei;max-age=30"

Cookie会在30秒之后过期,浏览器会在cookie过期时间到达主动将cookie删除(所以正确删除cookie的方式不是把cookie设置为空,而是把cookie过期时间设置为0)



服务器设置cookie

这里用Koa写了一个小demo



初始化一个项目,并且安装koa

const Koa = require('koa')

const KoaRouter = require('@koa/router')

const app = new Koa()

const userRouter = KoaRouter({ prefix: '/user' })

userRouter.get('/login', (ctx, next) => {

    // 登录设置到客户端cookie chenjiawei

    ctx.cookies.set('name','chenjiawei',{

        maxAge: 60 \* 60 \* 1000

    })

    ctx.body = `登录成功,设置cookie成功~`

})

userRouter.get('/list', (ctx, next) => {

    // 用户操作验证cookie 令牌口号为chenjiawei

    const name = ctx.cookies.get('name')

    if(name === 'chenjiawei') {

        ctx.body = {

            code: 1,

            message: "sucess"

        }

    }else {

        ctx.body = {

            code: 200,

            message: "cookie失效"

        }

    }

})

app.use(userRouter.routes())

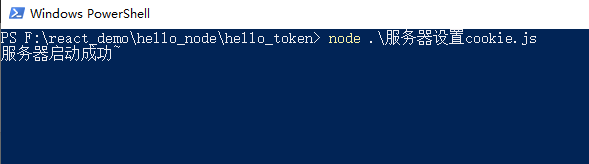
app.use(userRouter.allowedMethods())

app.listen(8000, () => {

    console.log('服务器启动成功~')

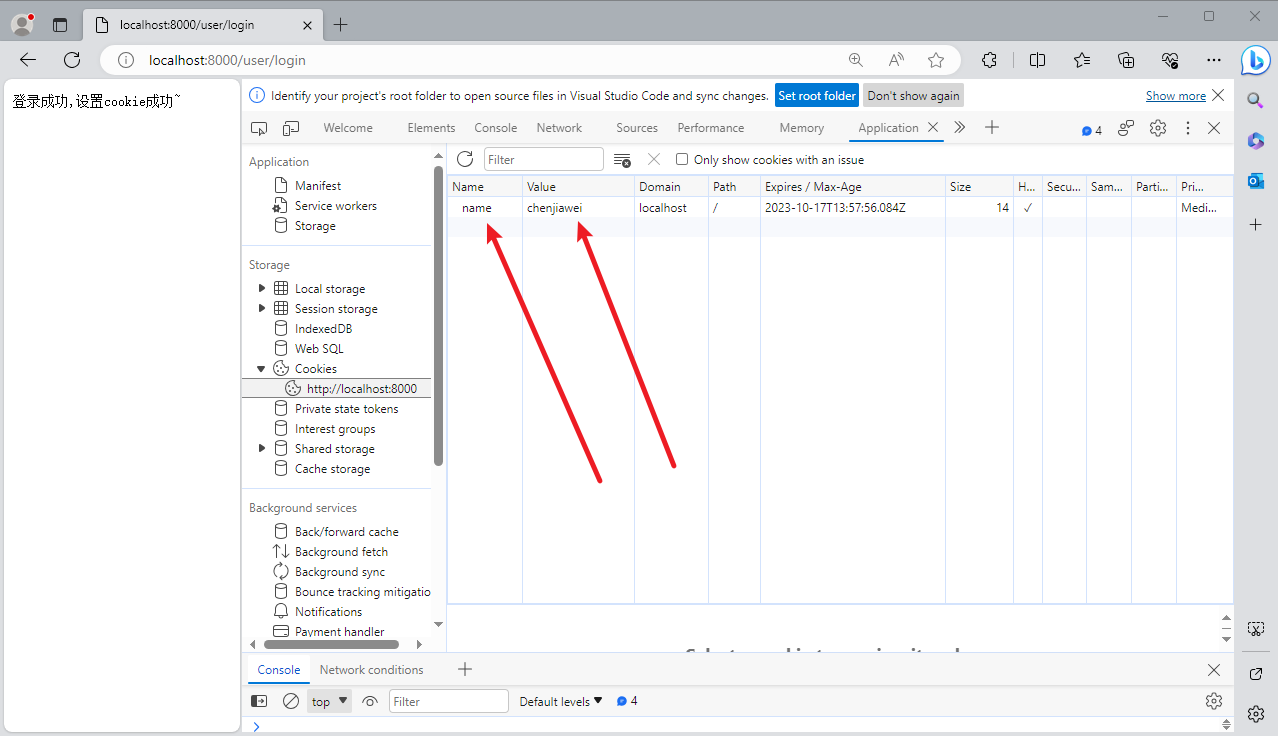
})

为了方便测试我这里使用get请求

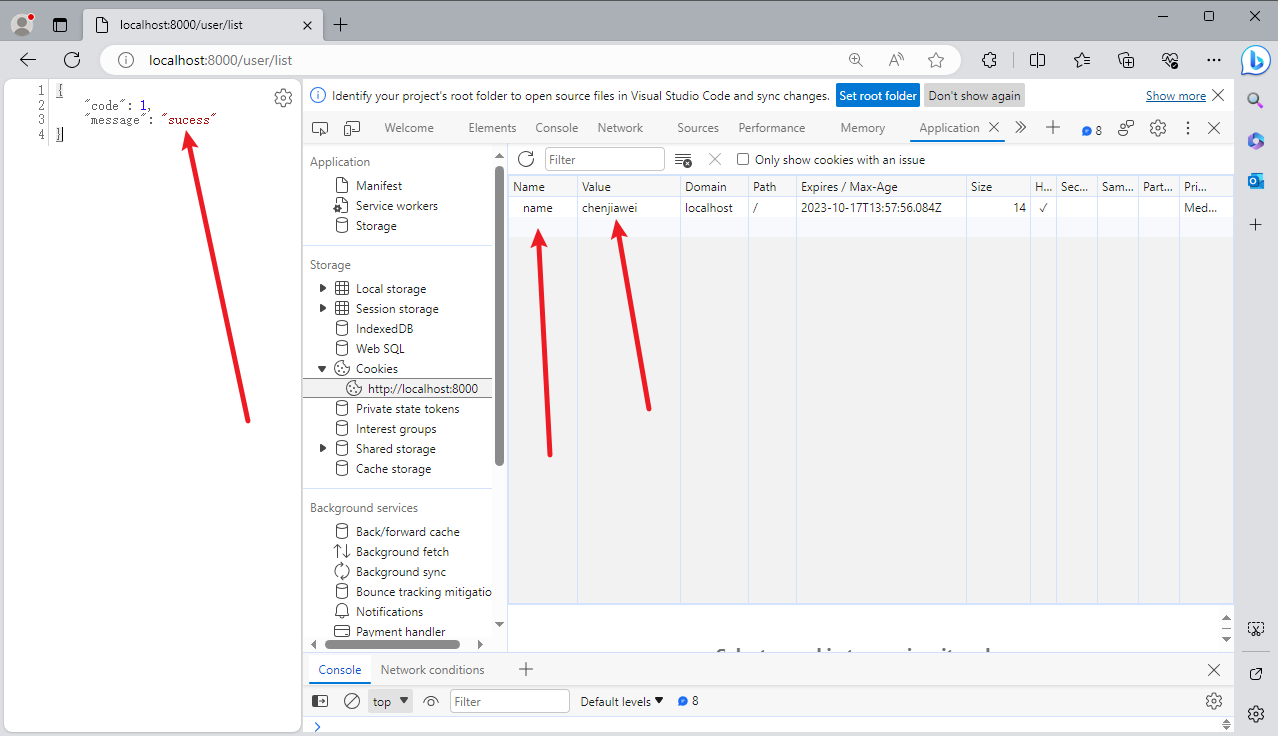


使用node启动服务

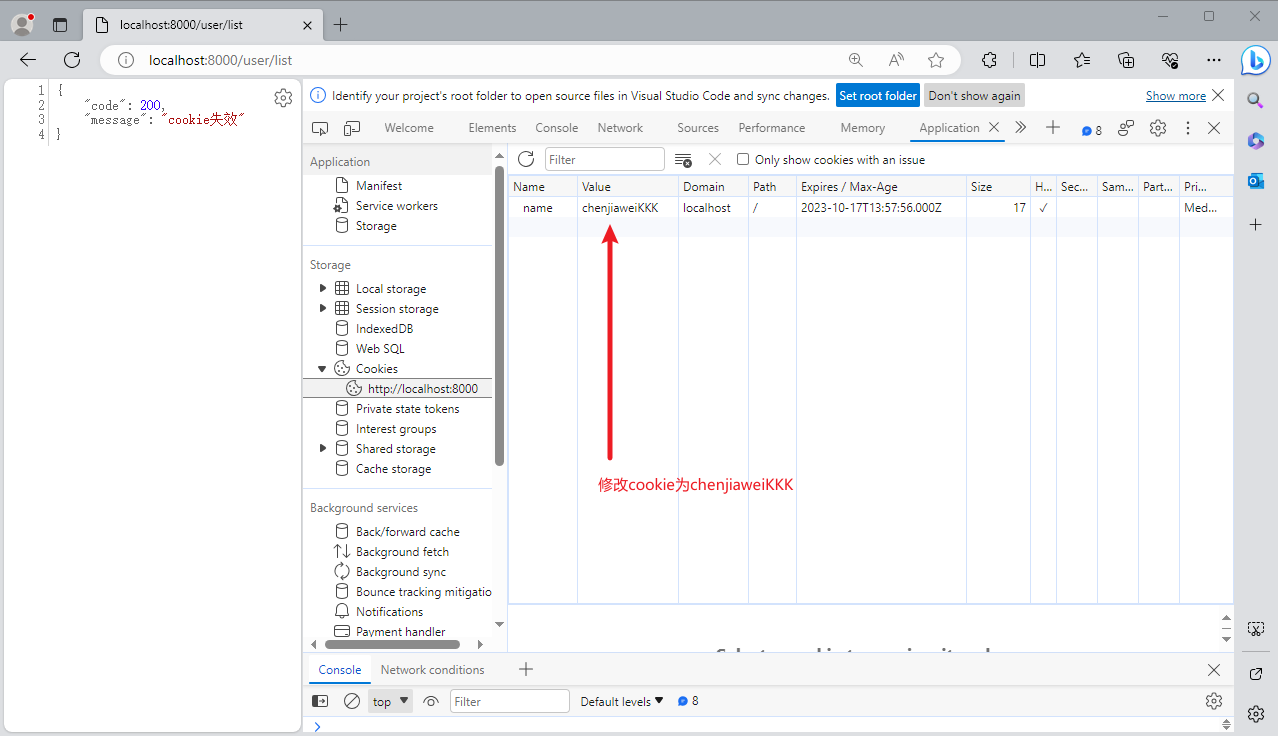
访问本地localhost端口为8000,访问/user/login接口，可以看到服务器设置了cookie，name为chenjiawei



继续访问/user/list接口,可以看到验证成功



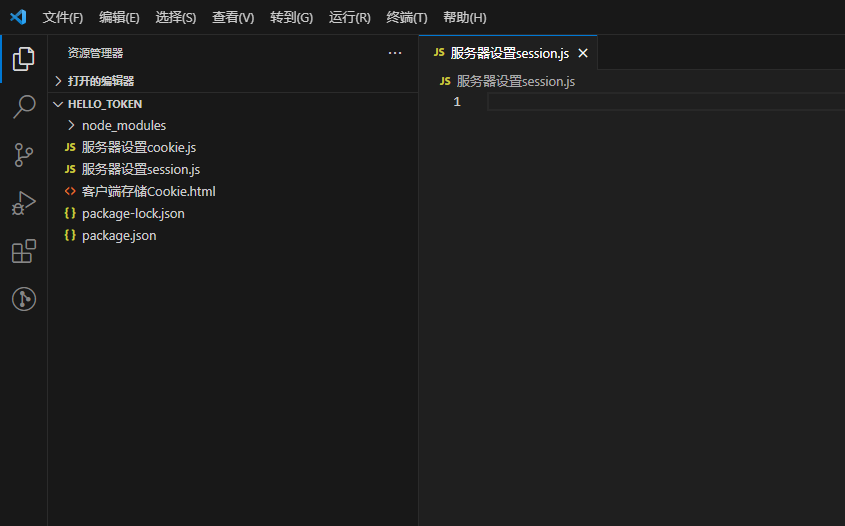
cookie修改之后可以发现cookie失效



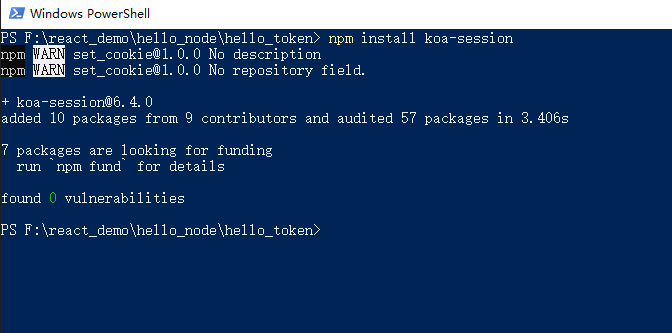
但这式样携带cookie都是明文的，非常容易被伪造，所以最好对cookie做一个加密，生成一个唯一的id，比如sessionid

Session

session是基于cookie的实现机制



创建一个js文件



安装koa-session库

const Koa = require('koa')

const KoaRouter = require('@koa/router')

const koaSession = require('koa-session')

const app = new Koa()

const userRouter = KoaRouter({ prefix: '/user' })

const session = koaSession({

    key: 'sessionid',

    signed: false

},app)

app.use(session)

userRouter.get('/login', (ctx, next) => {

    ctx.session.name = 'chenjiawei'

    ctx.body = `登录成功,设置session成功~`

})

userRouter.get('/list', (ctx, next) => {

    // 用户操作验证session

    const name = ctx.session.name

    console.log(name)

    if(name === 'chenjiawei') {

        ctx.body = {

            code: 1,

            message: "sucess"

        }

    }else {

        ctx.body = {

            code: 200,

            message: "session失效"

        }

    }

})

app.use(userRouter.routes())

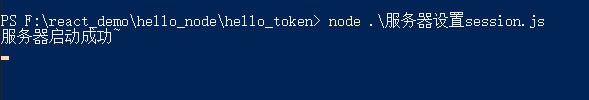
app.use(userRouter.allowedMethods())

app.listen(8000, () => {

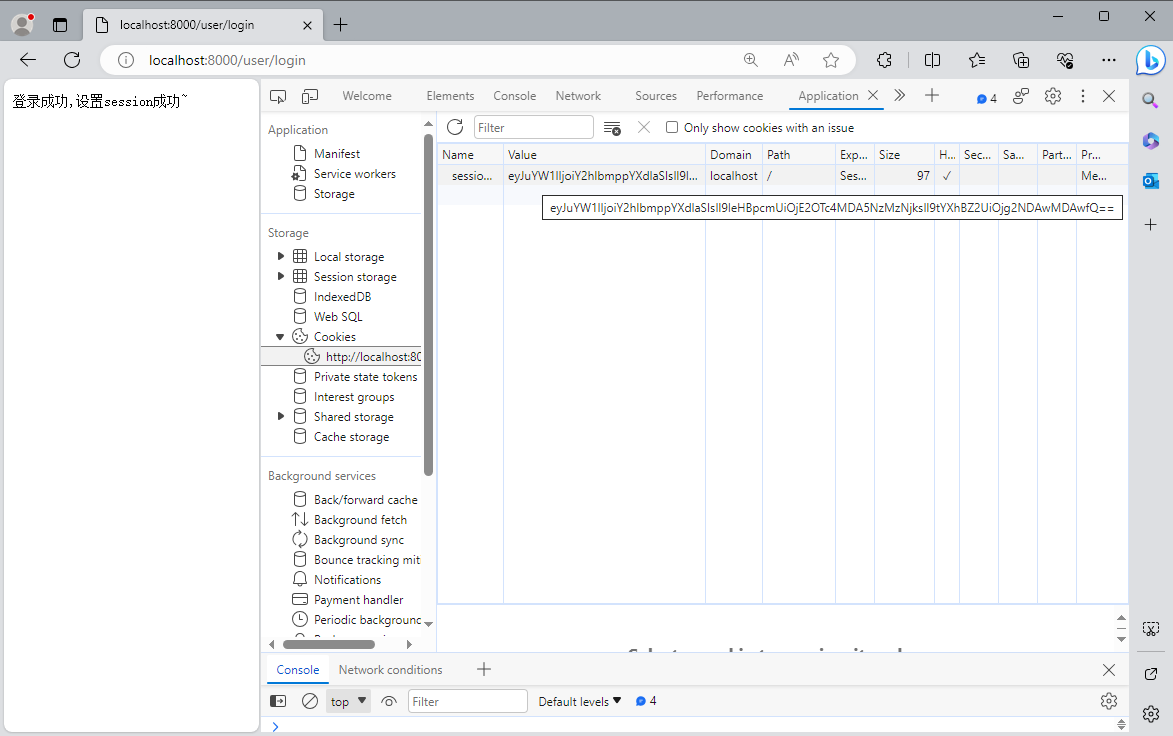
    console.log('服务器启动成功~')

})

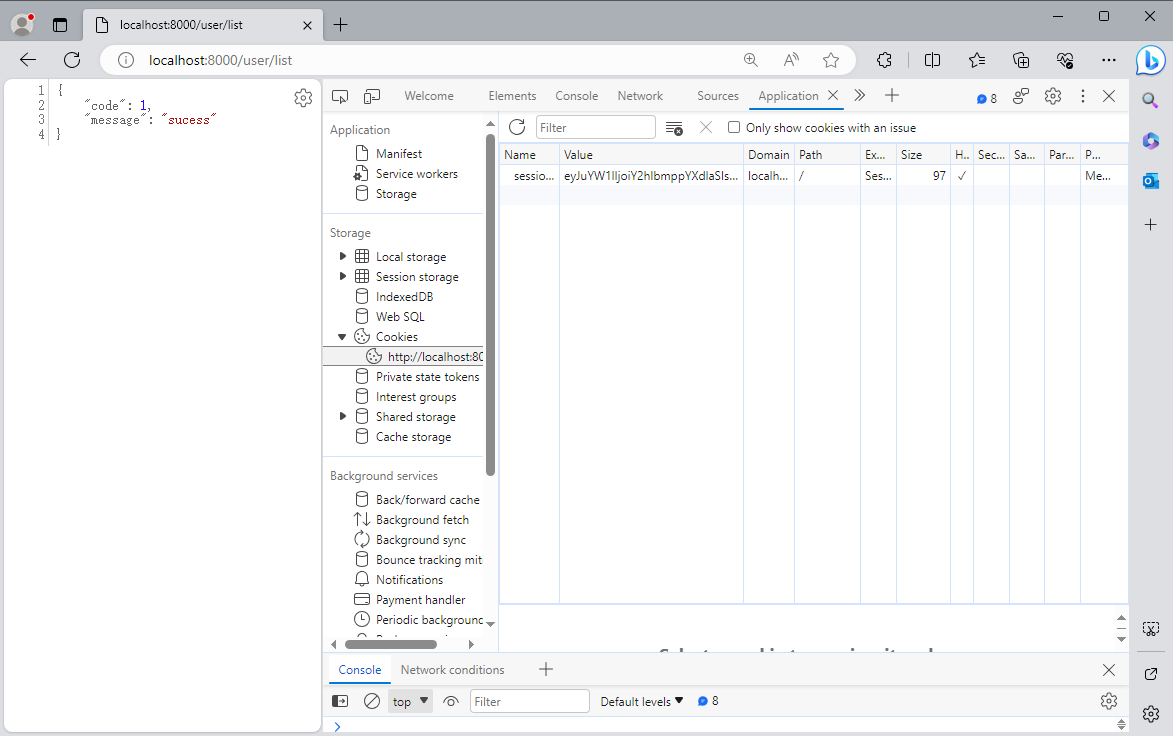
在原来的代码中创建一个session的中间件，把他在app中应用，修改部分代码



启动服务



访问/user/login可以看到服务器设置的sessionid，sessionid的内容是不知道的



也可以进行正确的身份验证

这样的sessionid相对比较安全，但是其实还可以对session再进行一个加密，可以对sessionid进行一个加盐操作，加盐操作是sessionid里比较多的操作，设置一个key加密，之后对sessionid解密时拿到这个key来解密才可以拿到原始session

const Koa = require('koa')

const KoaRouter = require('@koa/router')

const koaSession = require('koa-session')

const app = new Koa()

const userRouter = KoaRouter({ prefix: '/user' })

const session = koaSession({

    key: 'sessionid',

    signed: true

},app)

app.keys = ['123','chenjiawei','minecraft','jinitaimei']

app.use(session)

userRouter.get('/login', (ctx, next) => {

    ctx.session.name = 'chenjiawei'

    ctx.body = `登录成功,设置session成功~`

})

userRouter.get('/list', (ctx, next) => {

    // 用户操作验证session

    const name = ctx.session.name

    console.log(name)

    if(name === 'chenjiawei') {

        ctx.body = {

            code: 1,

            message: "sucess"

        }

    }else {

        ctx.body = {

            code: 200,

            message: "session失效"

        }

    }

})

app.use(userRouter.routes())

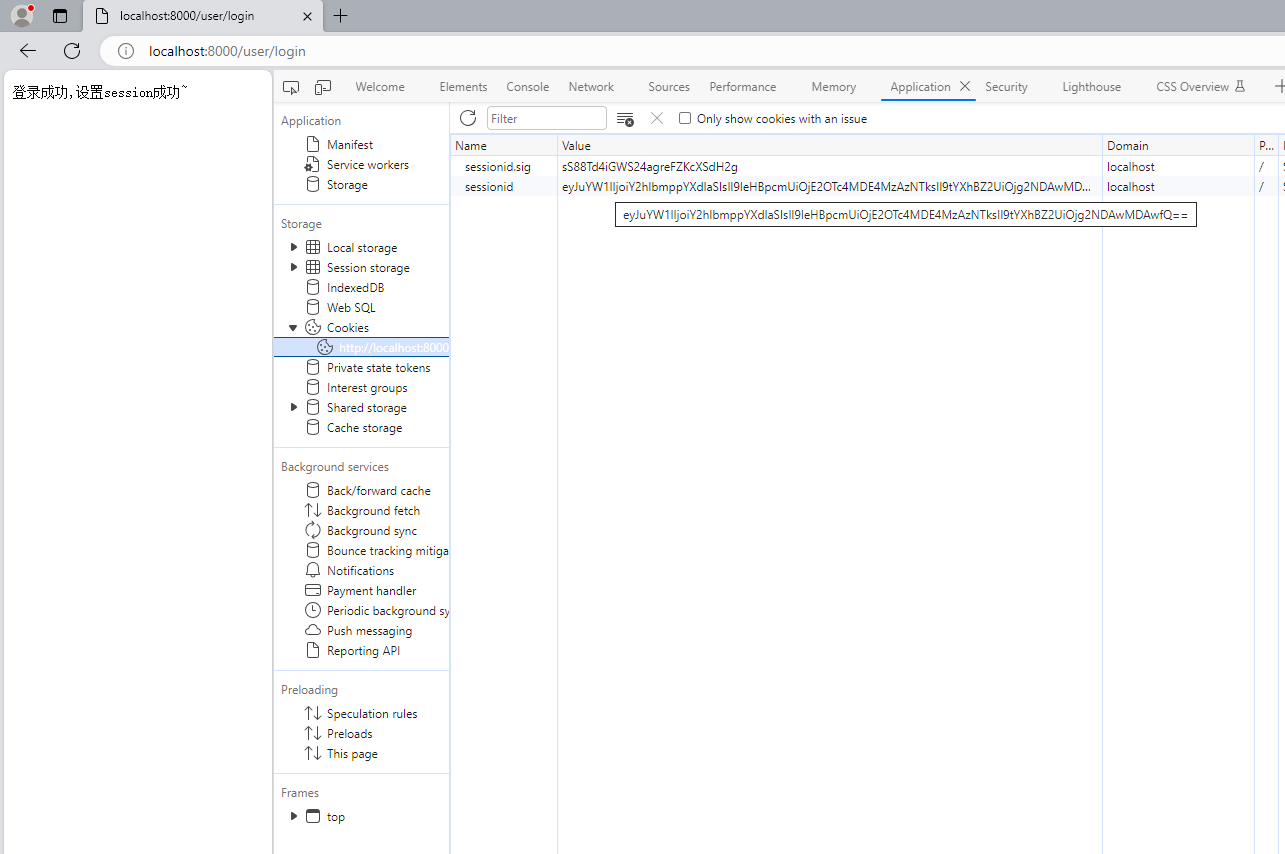
app.use(userRouter.allowedMethods())

app.listen(8000, () => {

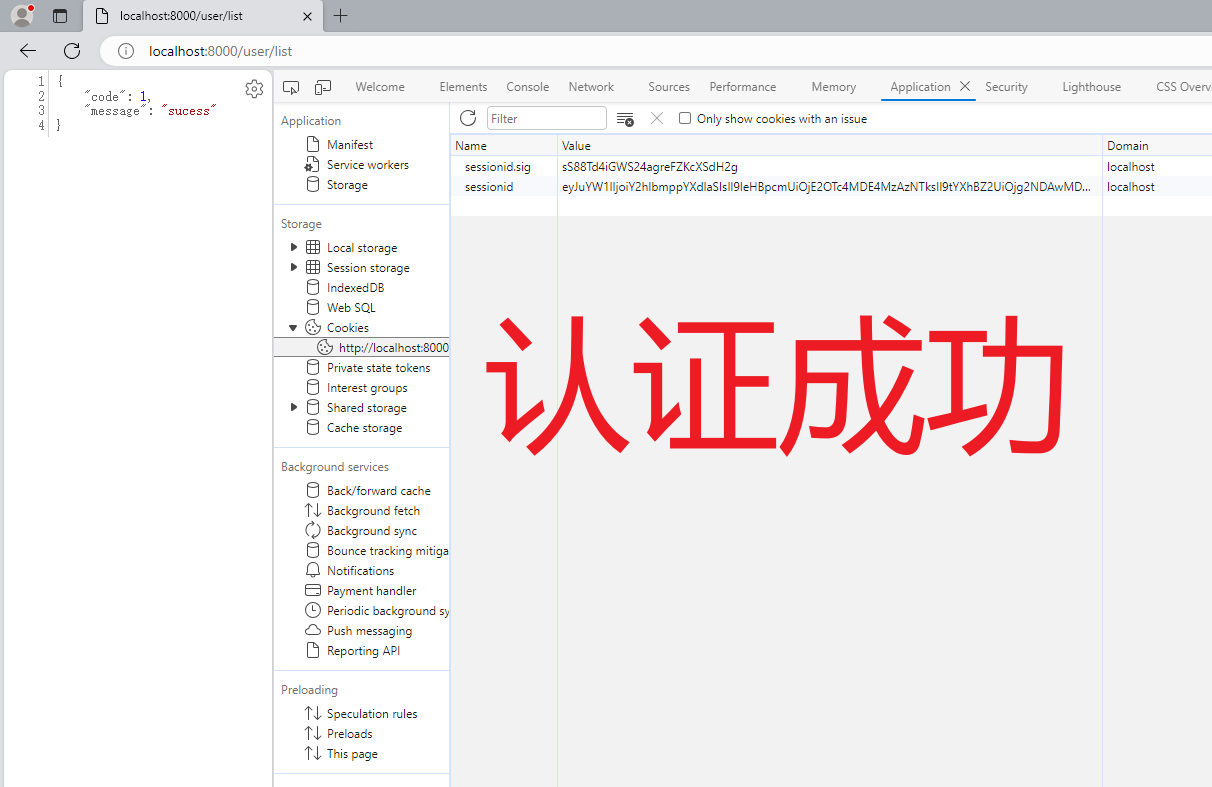
    console.log('服务器启动成功~')

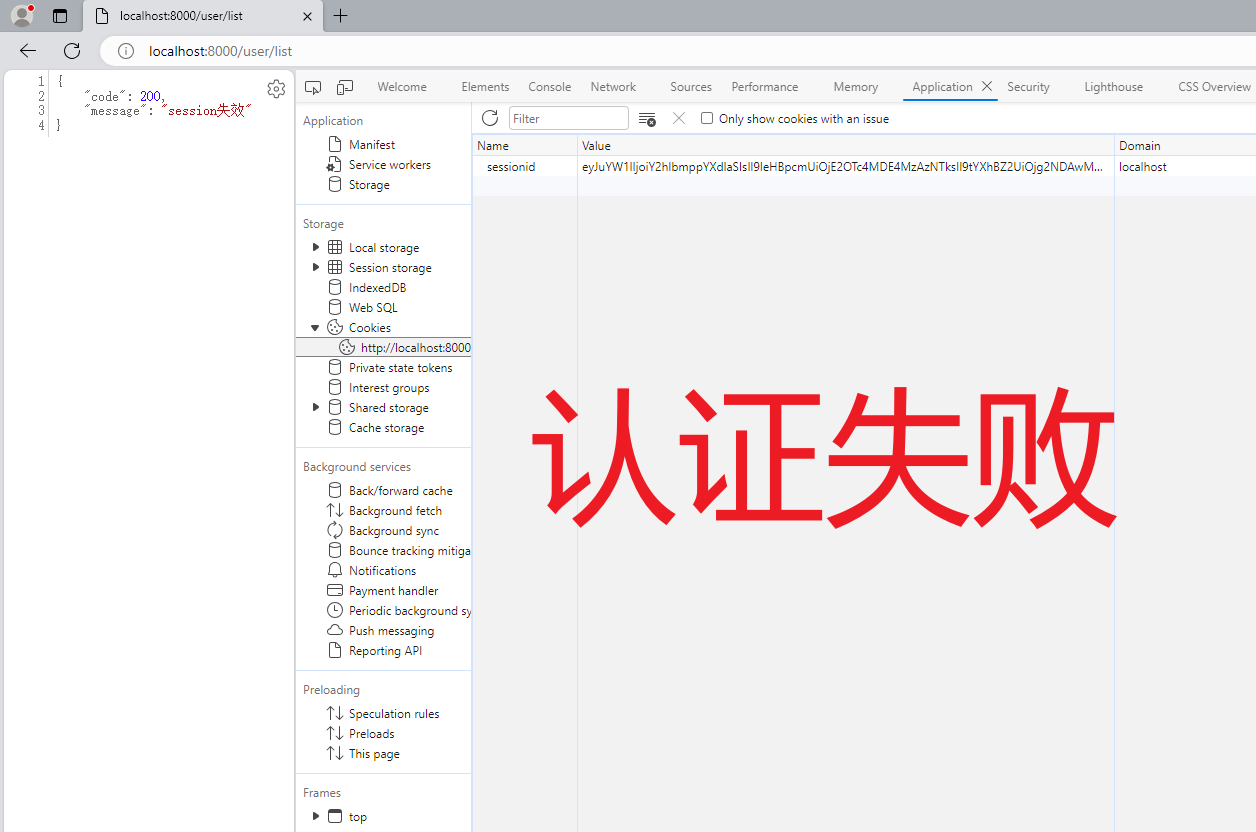
})

修改部分代码并且重新启动服务



可以发现两个cookie，一个是sessionid.sig,这个时加密签名，sessionid.sig和sessionid都会被发送到服务器，想要把sessionid解析，这两只数据都是必不可少的，这样的话如果想伪造sessionid是比较困难的，或者客户端不小心在网络传输中泄露了，然后拿到sessionid想验证也是会失败的，相当于一个双重认证





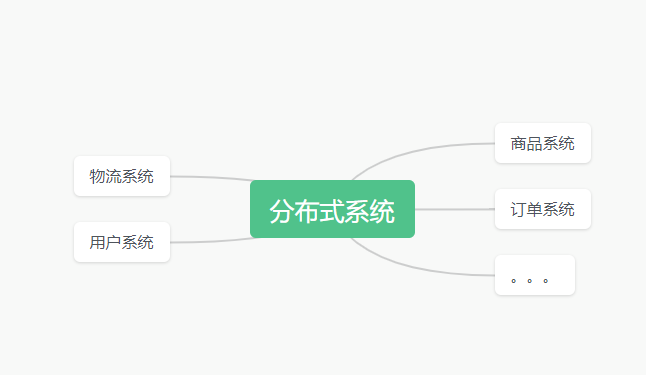
这是通过node实现的比较简单的身份验证，实际开发中一般用java或者Go写后台,也是比较喜欢用session来做身份验证，实际开发中还需要做更多复杂的操作，并没有那么简单，这里只是写了简单的例子来理解。

但是现在用session来验证的时变得越来越少了，他这个缺点也越来越明显了，用cookie或者cookie加密的session方式验证有很多缺点，这是我在网上查到并且整理的一些缺点:

1. cookie会被附加到每个http请求中，无形中增加了流量(事实上某学请求是不需要的)
2. cookie是明文传递的，所以存在安全性的问题
3. cookie的大小限制是4kb,对一些复杂的需求来说是不够的
4. 对于浏览器外的其他客户端(比如IOS和安卓),必须手动设置cookie和session
5. 对于分布式系统和服务器集群中如何保证其他系统也可以正确解析session

其上上面3个问题不是很大，主要是4和5

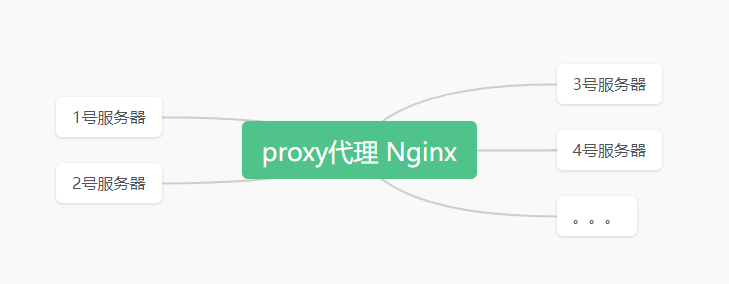
在一个完整的项目中，比如淘宝系统，特别双十一的时候，或者春运抢票，同一个服务器会被很多的用户来访问，那么这个时候服务器非常容易宕机，这样会产生比较严重的后果。虽然可以通过程序上优化，比如线程池等等，也可以通过分布式系统或者服务器集群来解决。



在分布式系统中我们可以把一个系统划分成很多子系统，我们只要保证每个子系统可以独立运行就可以了，使用了分布式系统例如在用户系统中进行了登录，由用户系统生成了sessionid，并且用户拿到了sessionid，那么之后在还是在这个域名下访问了比如商品系统时，携带上了sessionid，那么在用户系统中的sessionid如何在商品系统中认证会变得比较麻烦，在多个系统中来验证sessionid不是不可以，只是比较麻烦。

服务器集群就是有时候处理高并发，你不是部署了一个服务器，你是部署了很多服务器，你只要专门部署另外个服务器来代理，比如可以用Nginx服务器来代理，你要访问这个服务器之前访问代理武器，让这个代理服务器来做个转发，代理服务器看一下哪个服务器比较闲置，

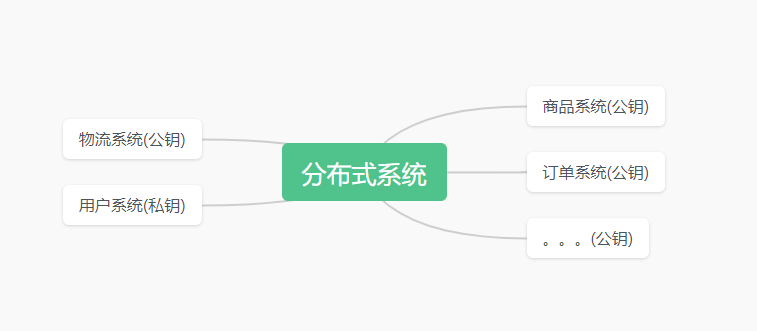
转发之后就可以分担某个服务器的压力



比如你在访问1号服务器时候拿到sessionid，但是1号服务器变得繁忙了，然后代理服务器就给我转发到2号服务器上去了，但是这现在又要去验证sessionid了，这样验证不是不好验证，只是比较验证很麻烦。

所以现在用cookie的很少的，虽然还是很多系统中还是用的cookie，因为保留了原来的系统架构，比如医教的智慧医教还是用的sessionid，所以在现在的前后端分离的开发过程中，使用token来进行身份验证的是最多的

Token它的翻译就是令牌，相当于给用户发了个令牌，这个令牌作为后续用户访问一些接口或者资源的凭证



比如在分布式系统中，你可以在用户系统里颁发令牌，然后告诉别的系统如何认证就可以了，

比如这里用非对称加密可以实现，在用户系统可以用私钥办法令牌，你可以在别的系统里解析这个令牌，但是别的系统是没有颁发令牌的权限的。在服务集群里也是这样的原理。

使用token生成是有具体的算法，基本都是由JWT来实现Token机制 json web token

JWT生成的Token由三部分组成

Header

alg: 采用的加密算法，默认使用HS256(这个是个对称加密，这个不太行，因为每个人都有公钥和私钥，因为密钥越多越容易暴露，，暴露了就可以伪造token了，暴露了就完蛋了)，采用同一个密钥进行解密还有加密

typ：应该就是类型，好像没什么用

Header最后会进行base64url的算法进行编码

{

“alg”:”HS256”,

“typ”:”JWT”

}

Payload

携带的数据，比如你可以把id或者name放到payload中

iat默认会携带iat(issued at)，令牌的签发时间，格式是时间戳格式

Exp (expiration time)过期时间，格式也是时间戳格式

Payload 最后会进行base64url的算法进行编码

{

“name”:”chenjiawei”,

“iat”:”1697804890”,

“iat”:”1752792958”

}

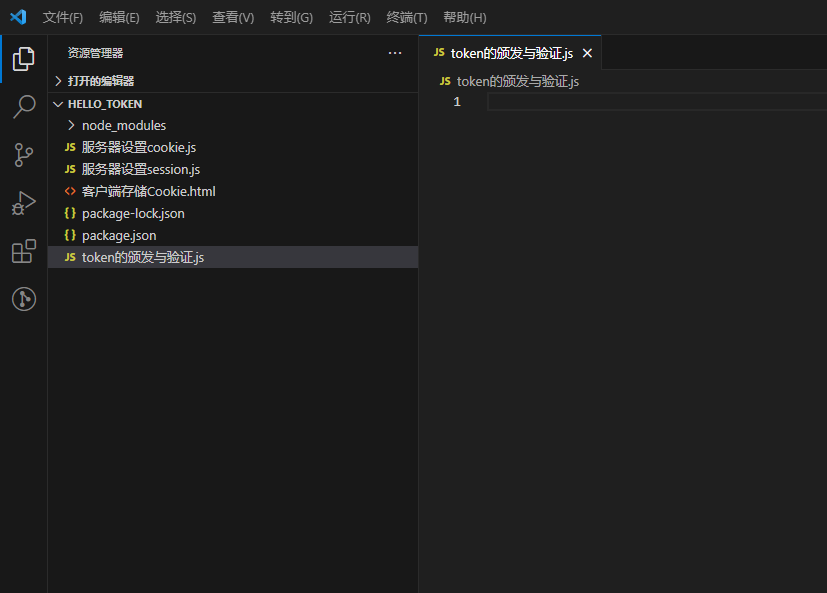
上面这俩最后都是一串字符串，她们因为都是base64编码出来的，所以肯定也可以被解码出来的,第三个信息是防止token被仿造或者逆向的

Signature

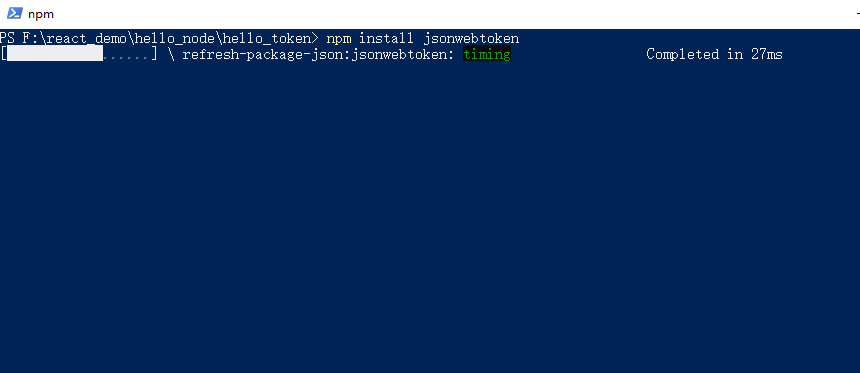
secretKey，这个secretKey超级重要，一定要被保管好，他是通过把前两个结果合并后进行HS256的算法，所以这个东西一般不能被看到，被发现就可以模拟颁发token，也可以解密token，我认为只有技术总监这种可以看到

HS256(base64Url(header) + . +baseUrl(payload),secretKey)

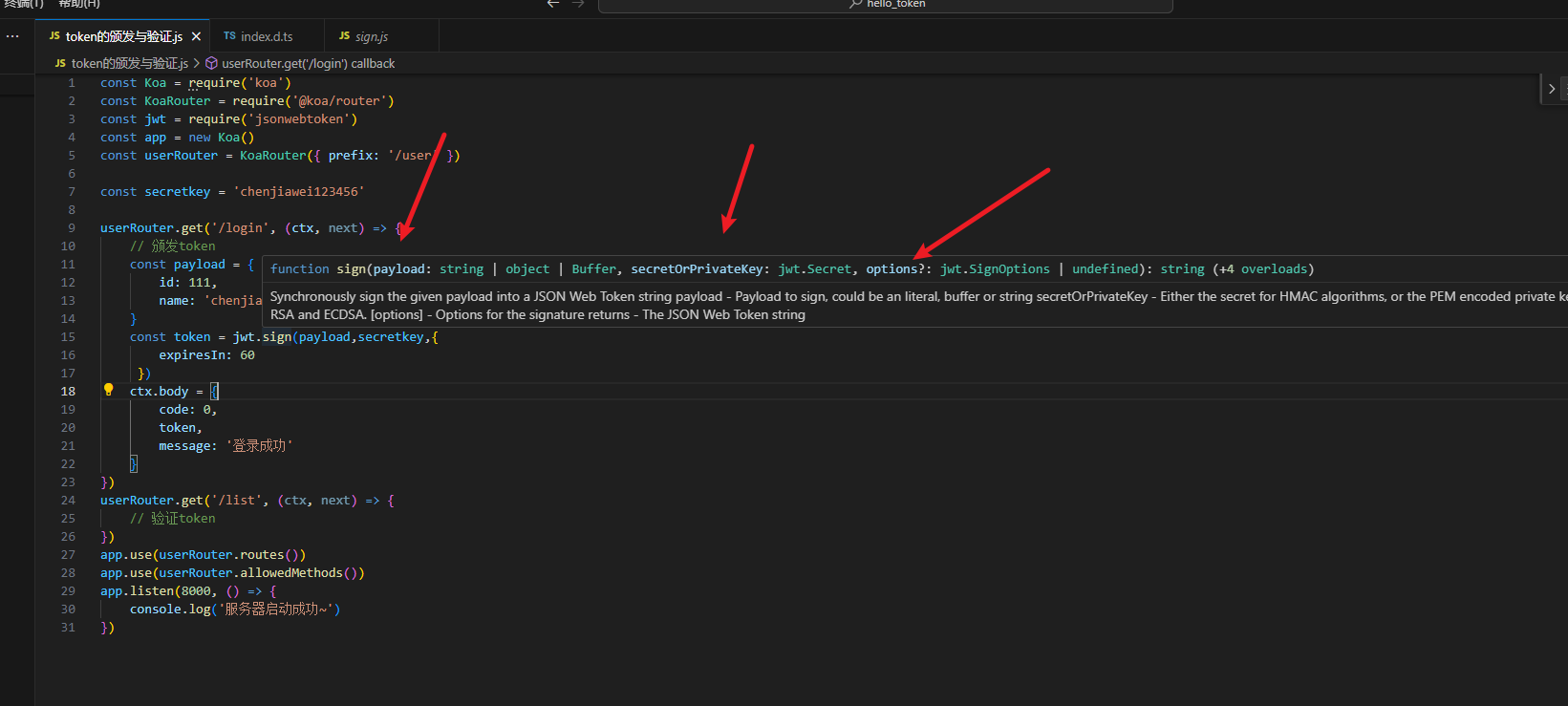
这三个结合到一起就是一个token



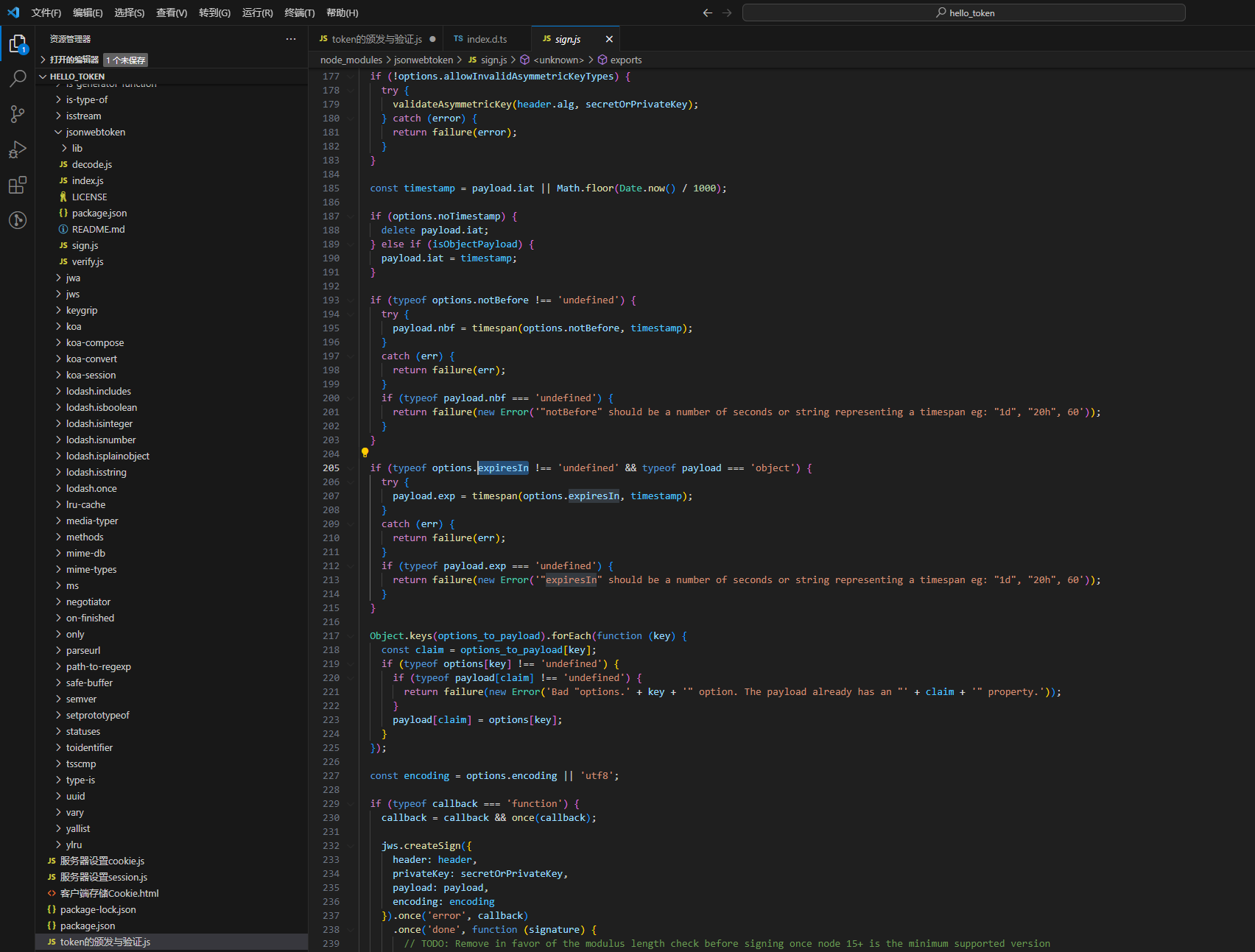
创建一个js文件，可以根据上面的算法自己生成一个token，我这里直接用一个开源库jsonwebtoken



安装库



授权token时可以传入三个参数，分别是payload，secretkey，还有一个option，payload你可以把个人信息放在里面，secretkey就是相当于私钥，options是个对象里面可以传一些配置，翻阅源码在源码里可以看到你可以传入过期时间等等



options.expiresIn过期时间，这里的单位是秒钟

const Koa = require('koa')

const KoaRouter = require('@koa/router')

const jwt = require('jsonwebtoken')

const app = new Koa()

const userRouter = KoaRouter({ prefix: '/user' })

const secretkey = 'chenjiawei123456'

userRouter.get('/login', (ctx, next) => {

    // 颁发token

    const payload = {

        id: 111,

        name: 'chenjiawei'

    }

    const token = jwt.sign(payload,secretkey,{

        expiresIn: 60

     })

    ctx.body = {

        code: 0,

        token,

        message: '登录成功'

    }

})

userRouter.get('/list', (ctx, next) => {

    // 验证token

})

app.use(userRouter.routes())

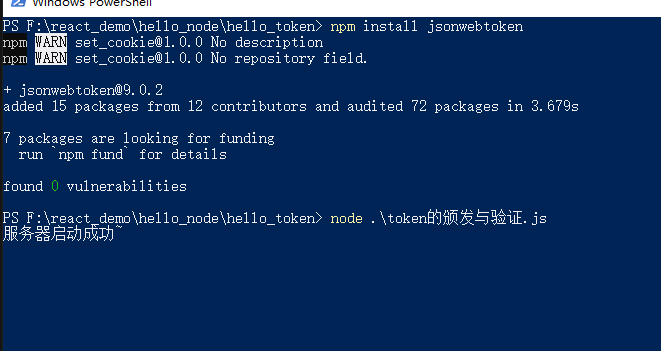
app.use(userRouter.allowedMethods())

app.listen(8000, () => {

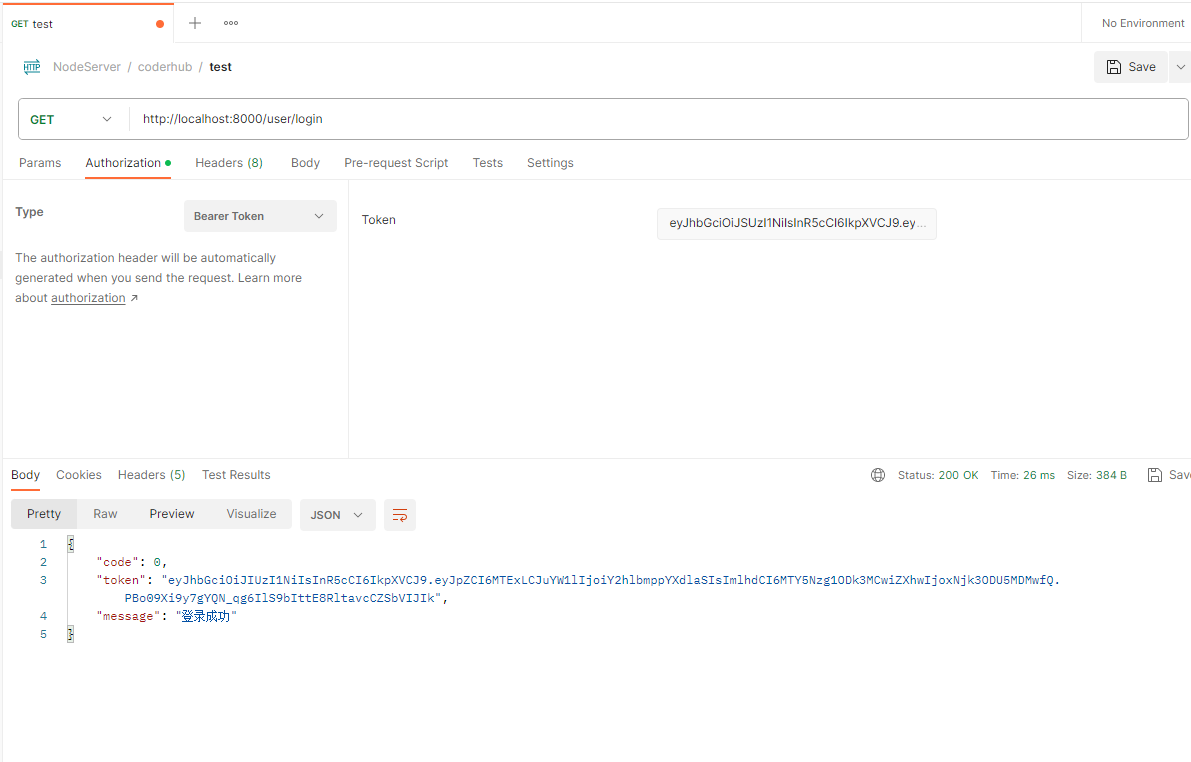
    console.log('服务器启动成功~')

})

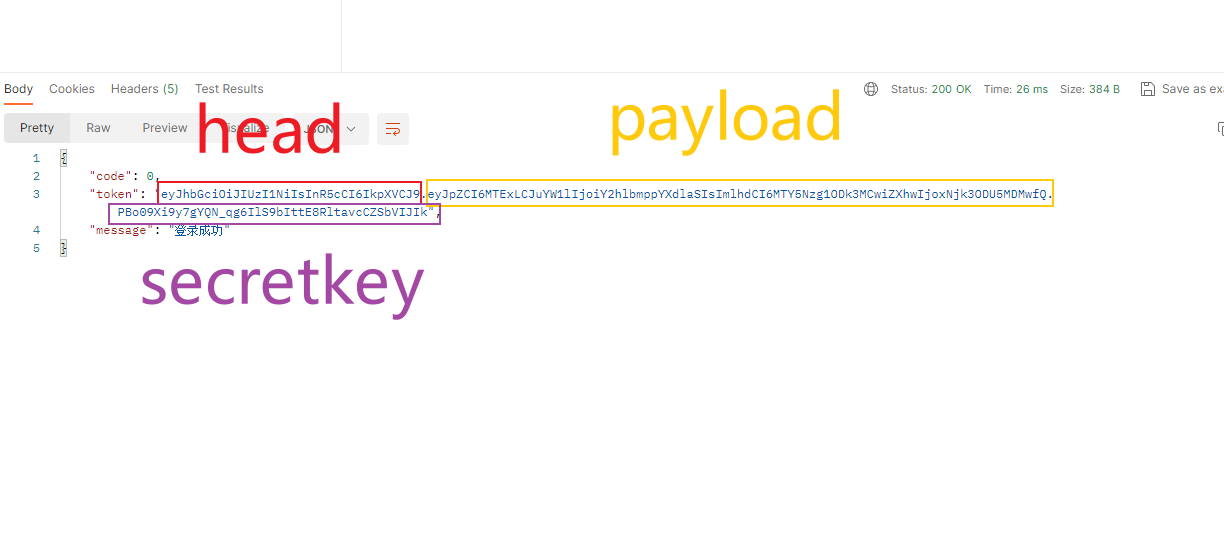
修改代码



启动服务

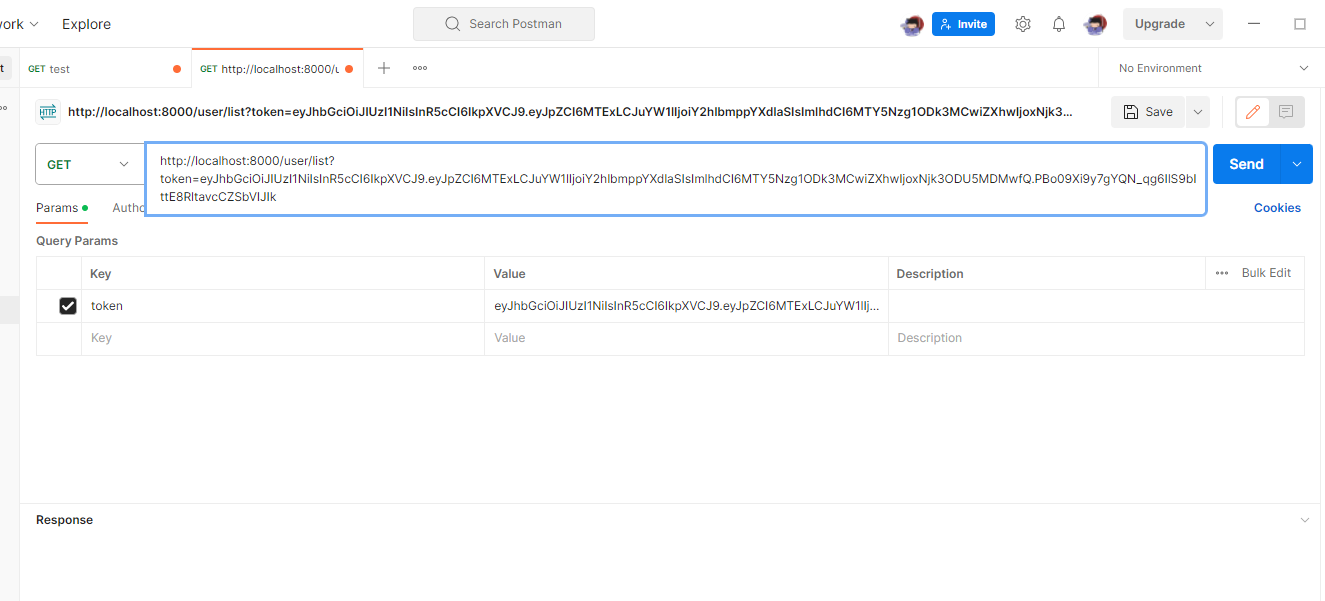


可以看到获取到的token

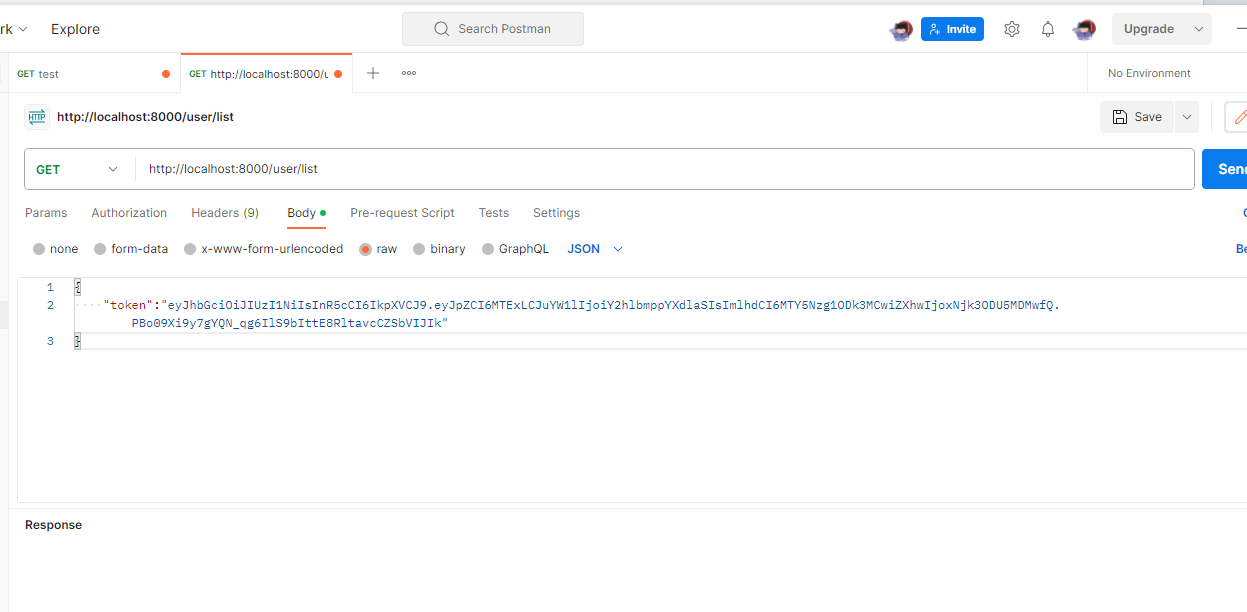


Token的构成

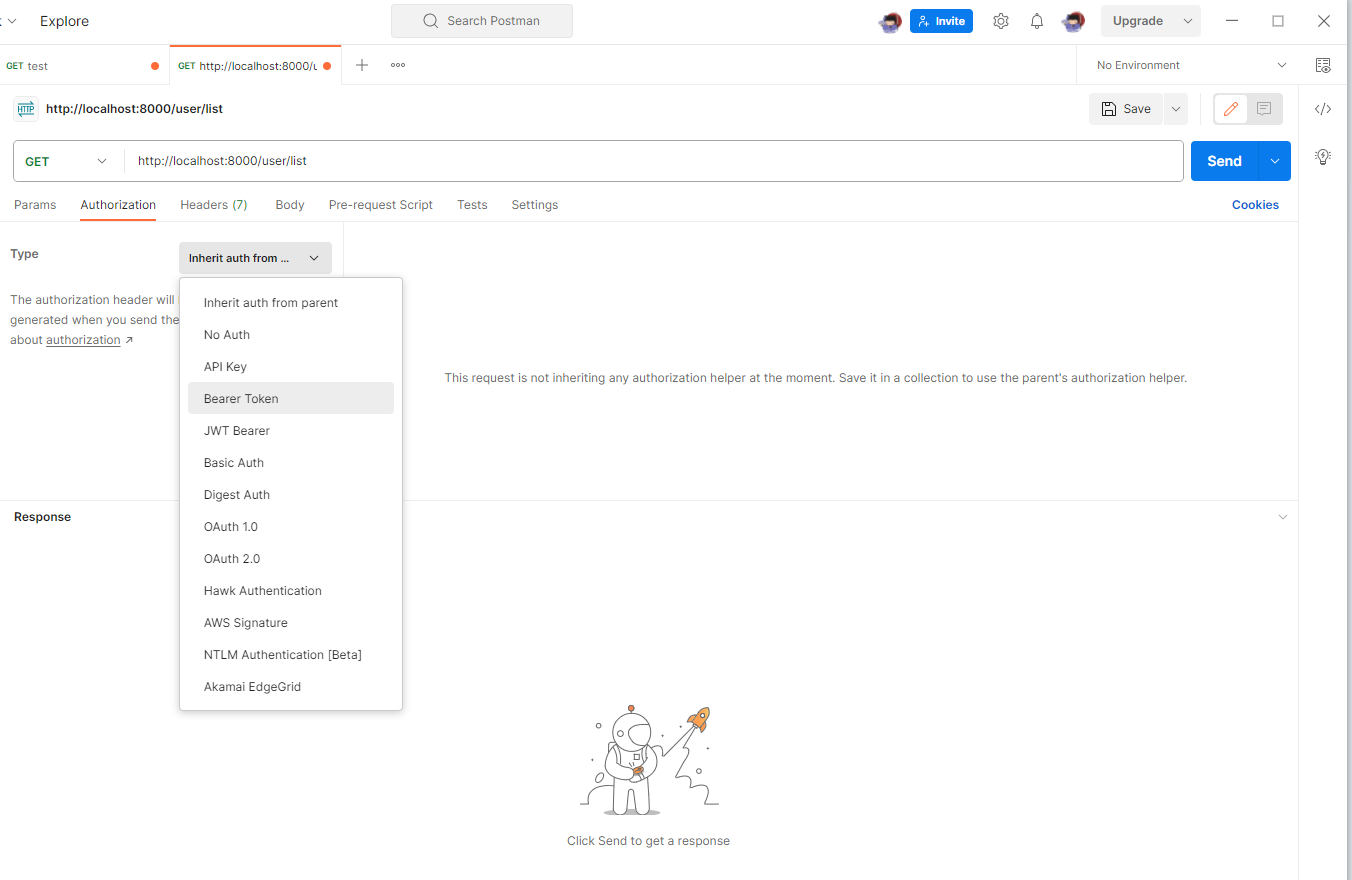
在验证token时候请求可以携带token，很多方法可以携带，主要是服务器说让你咋携带你就咋携带



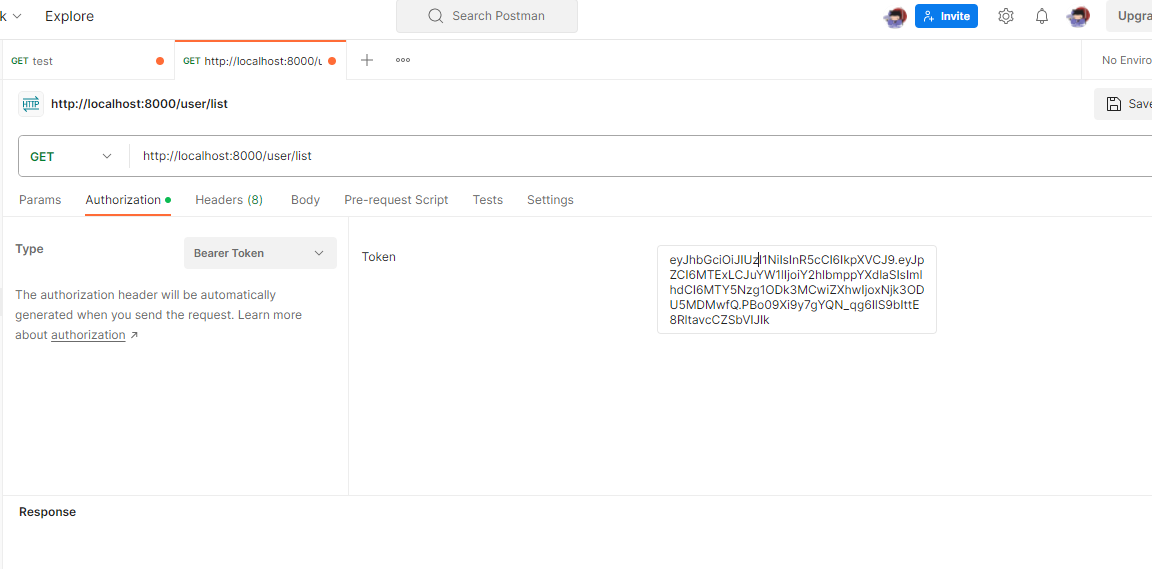
可以放在queryString携带



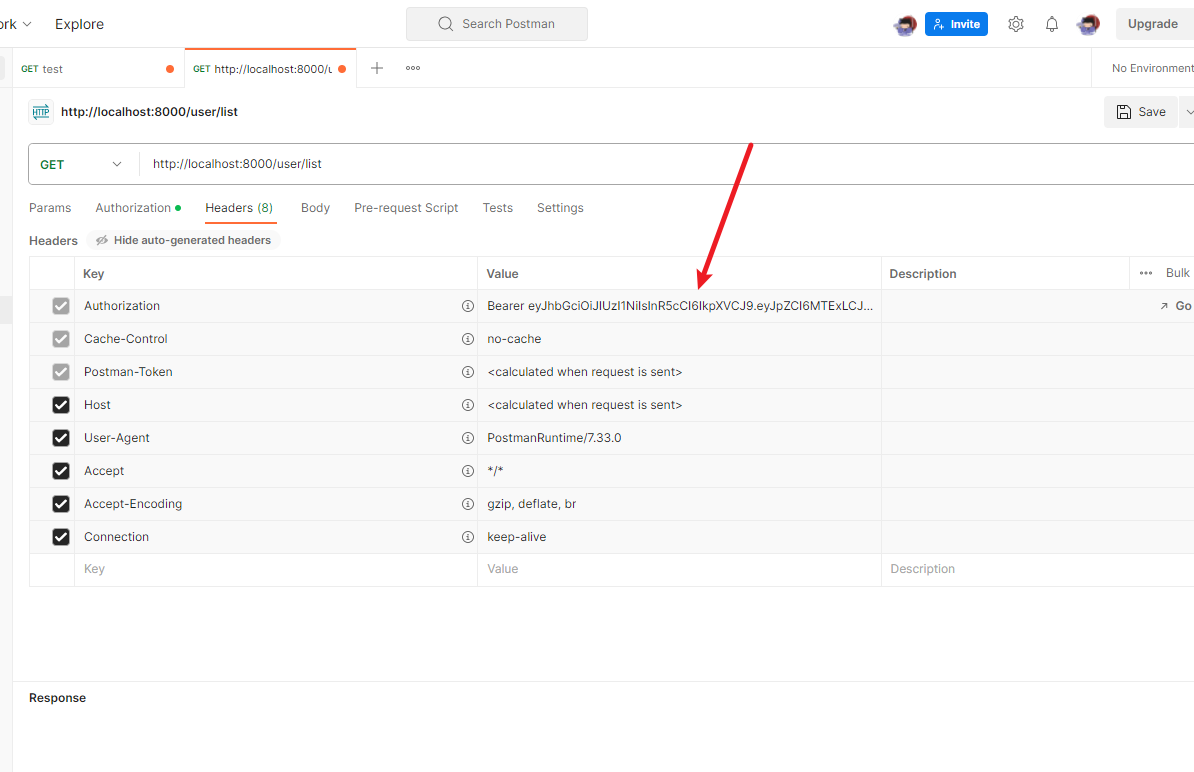
你也可以直接放在json里携带



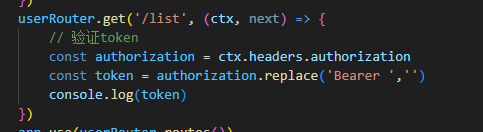
放在授权里携带是比较多的携带方法，这是比较常见的，Bearer Token里携带，bearer翻译过来就是送信人



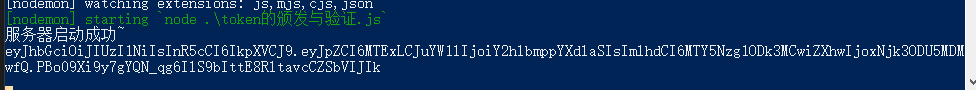
它的本质其实就似乎设置到header里去了



我们可以在header里找到他



把Bearer(空格)字符串替换成空字符串，输出验证一下



可以看到token

const Koa = require('koa')

const KoaRouter = require('@koa/router')

const jwt = require('jsonwebtoken')

const app = new Koa()

const userRouter = KoaRouter({ prefix: '/user' })

const secretkey = 'chenjiawei123456'

userRouter.get('/login', (ctx, next) => {

    // 颁发token

    const payload = {

        id: 111,

        name: 'chenjiawei'

    }

    const token = jwt.sign(payload,secretkey,{

        expiresIn: 60

     })

    ctx.body = {

        code: 0,

        token,

        message: '登录成功'

    }

})

userRouter.get('/list', (ctx, next) => {

    // 验证token

    const authorization = ctx.headers.authorization

    const token = authorization.replace('Bearer ','')

    try {

        const result = jwt.verify(token, secretkey)

        ctx.body = {

            code: 0,

            message: '验证成功'

        }

    } catch(error) {

        ctx.body = {

            code: -1001,

            message: 'token失效'

        }

    }

})

app.use(userRouter.routes())

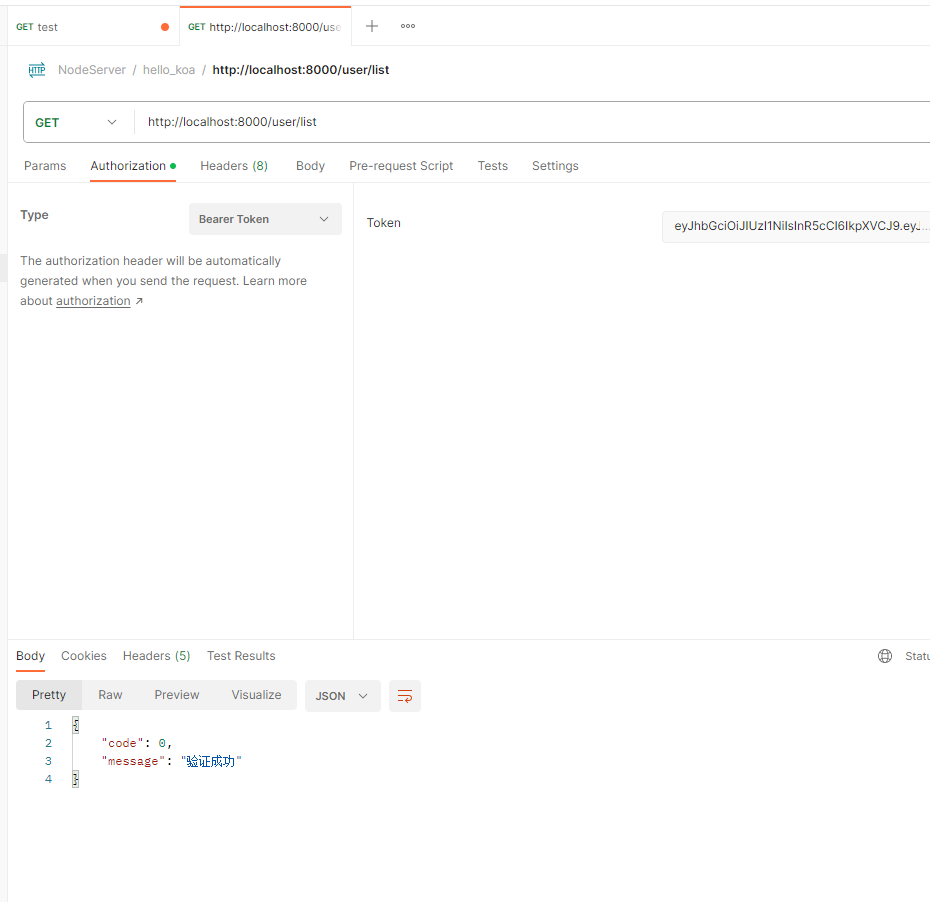
app.use(userRouter.allowedMethods())

app.listen(8000, () => {

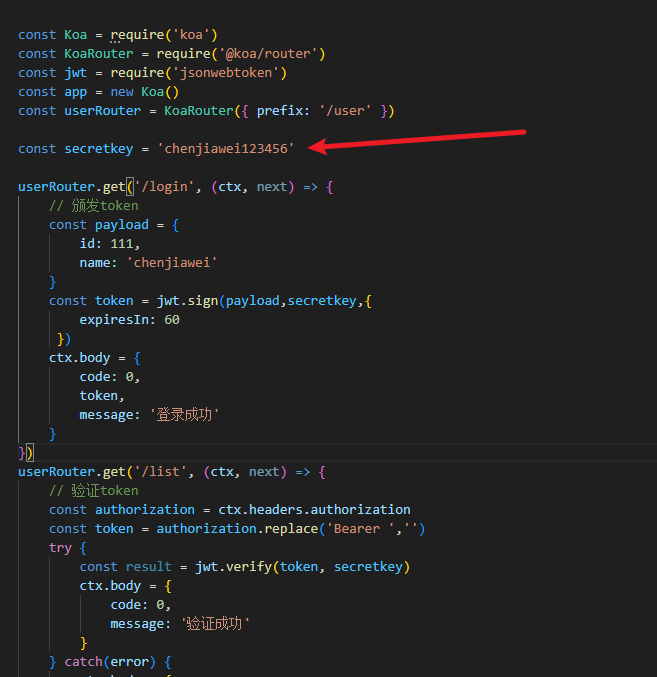
    console.log('服务器启动成功~')

})

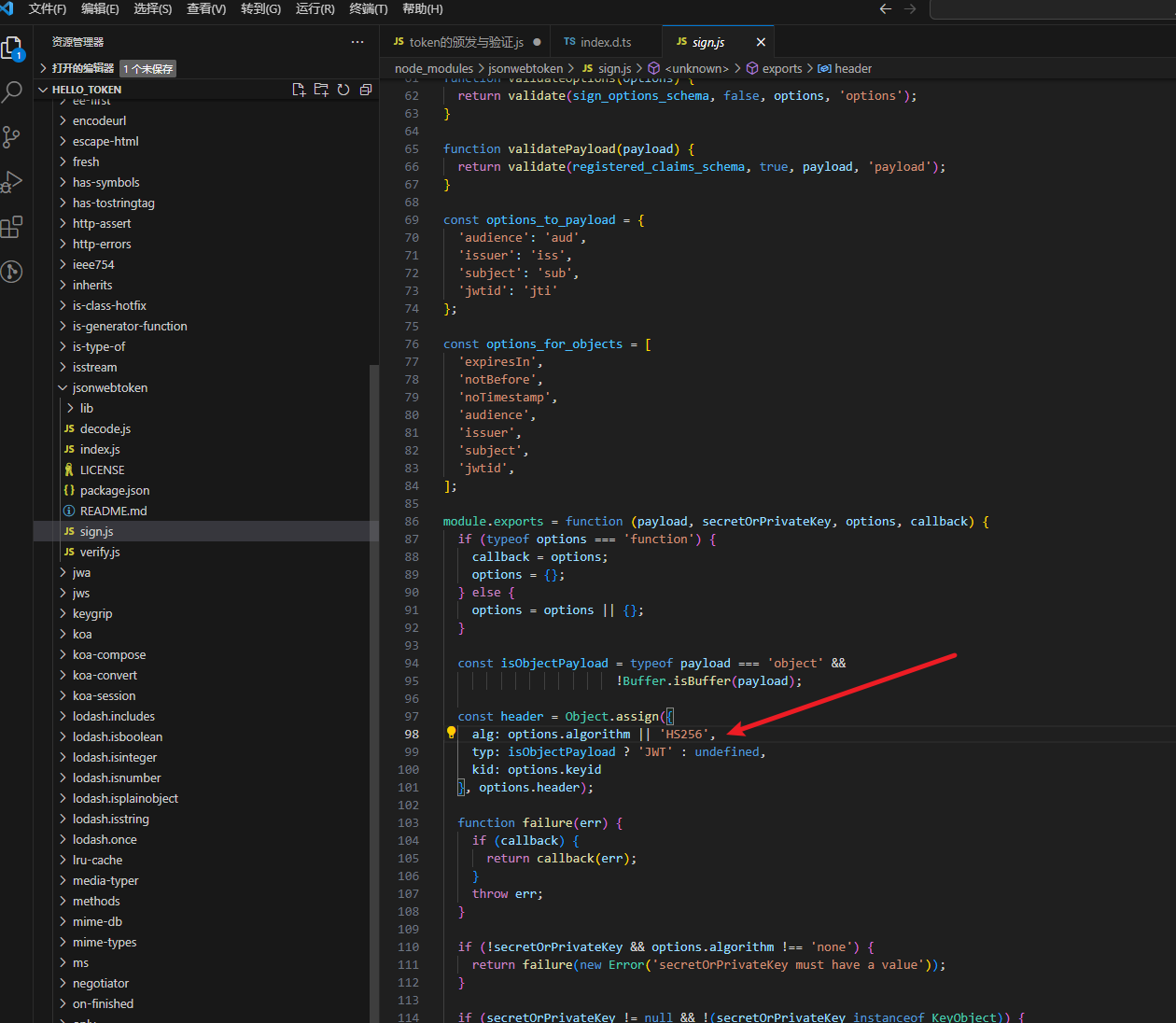
修改一下代码



Token验证成功



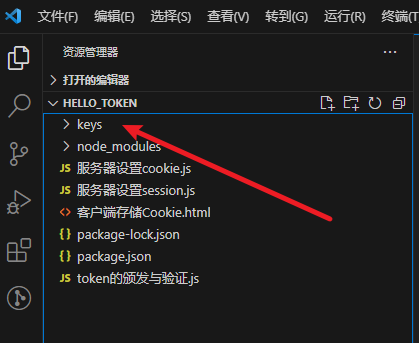
但是secretkey这样写比较危险，相当于被共享了，相当于任何一个系统都有颁发secretkey的权限，非常非常容易被伪造，被攻击任意一个系统就可以拿到secretkey，非常的危险，所以采用的算法尽量不要选择HS256这个算法。



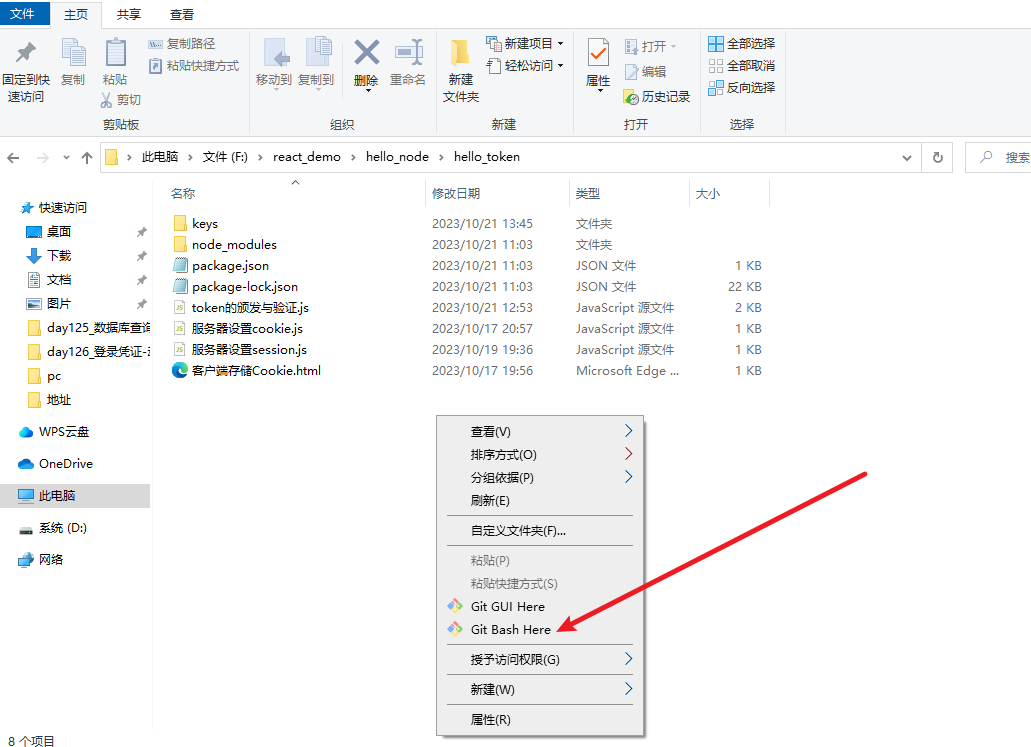
源码中默认使用的就是HS256

所以应该采用非对称加密，就是它会产生两个key，私钥(private\_key)和公钥(public\_key)，因为只有私钥有颁发token的权限，你就算拿到了公钥也是没有颁发token的权限，他只能验证token是否有效，这样可以让系统相对来说比较安全，所以只要把颁发token的服务器进行安全的多多加固就比较安全了。这里我用一下RS256，

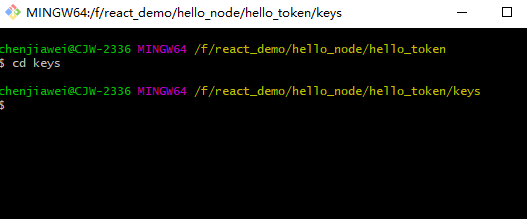
可以使用opnessl来生成私钥和公钥，windows的cmd没有安装openssl，需要单独安装，比较麻烦，这可以直接用git bash，你电脑装了git就有这个工具了



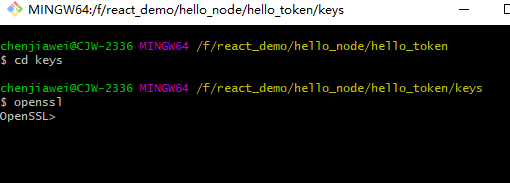
新建一个放私钥和公钥的文件夹



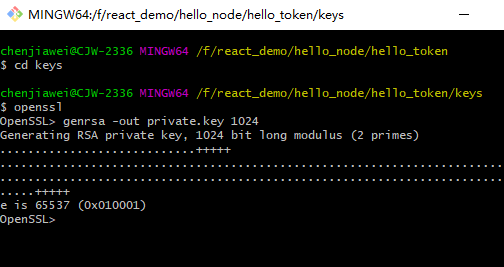
这里使用git bash



cd到keys这个文件夹里

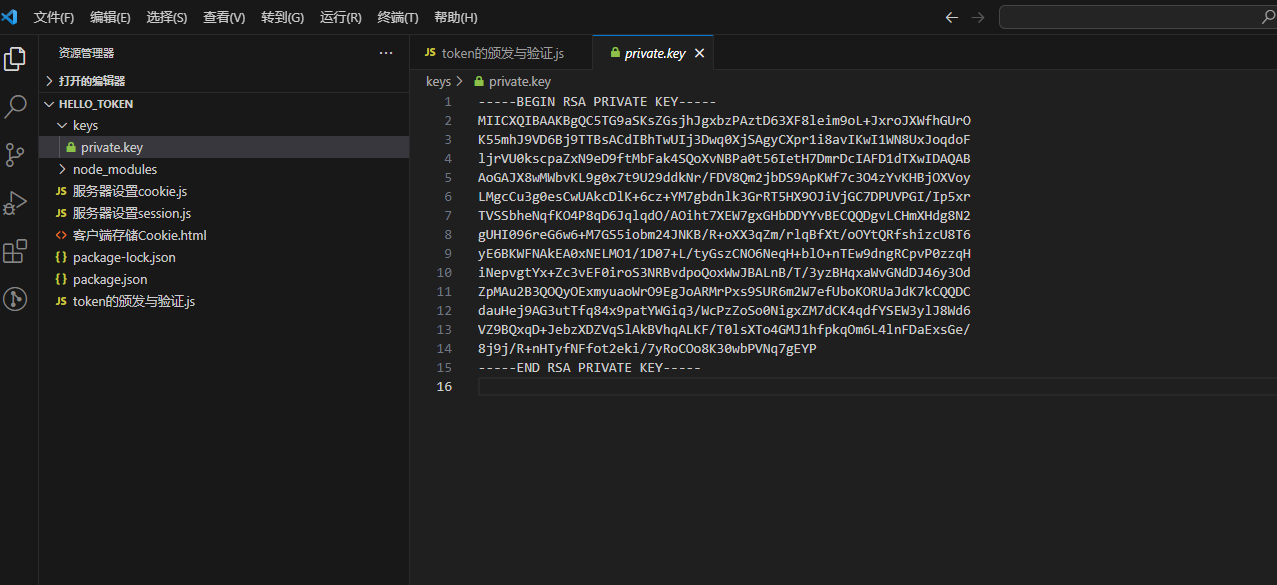


先输入指令openssl，然后进入到openssl的交互式命令里去

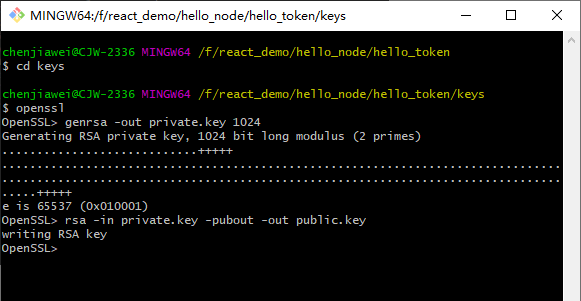


输入命令genrsa -out private.key 1024

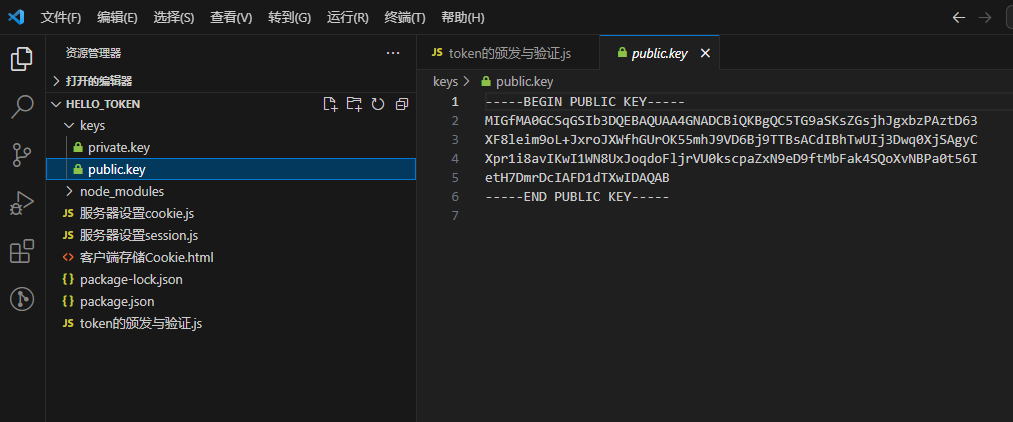
gen就是generate的缩写，意思就是生成，rsa就是生成的私钥是rsa这样的算法，-out意思是输出，就是在当前文件下输出， private.key是名字， 1024是长度



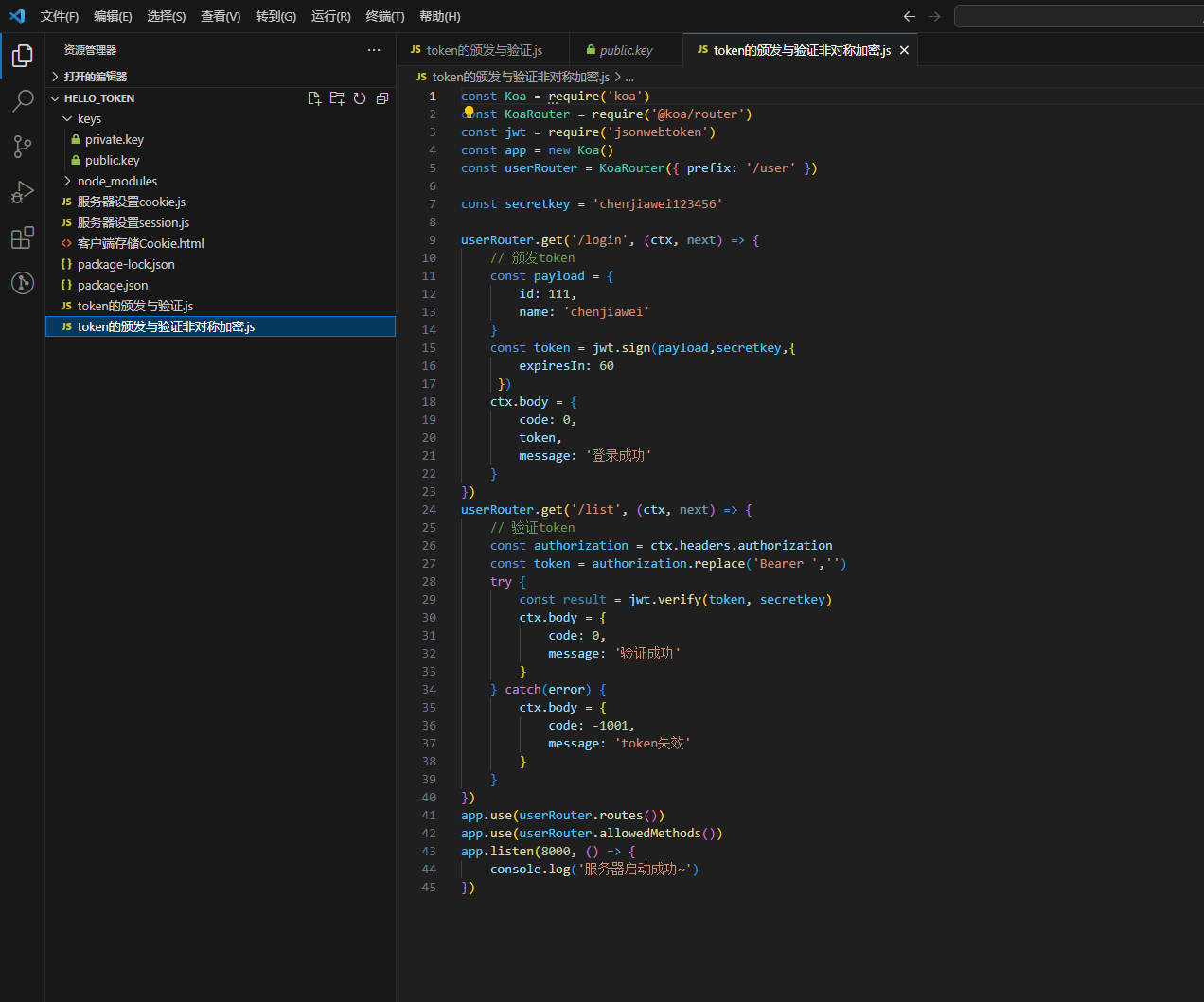
这样就可以看到一串私钥



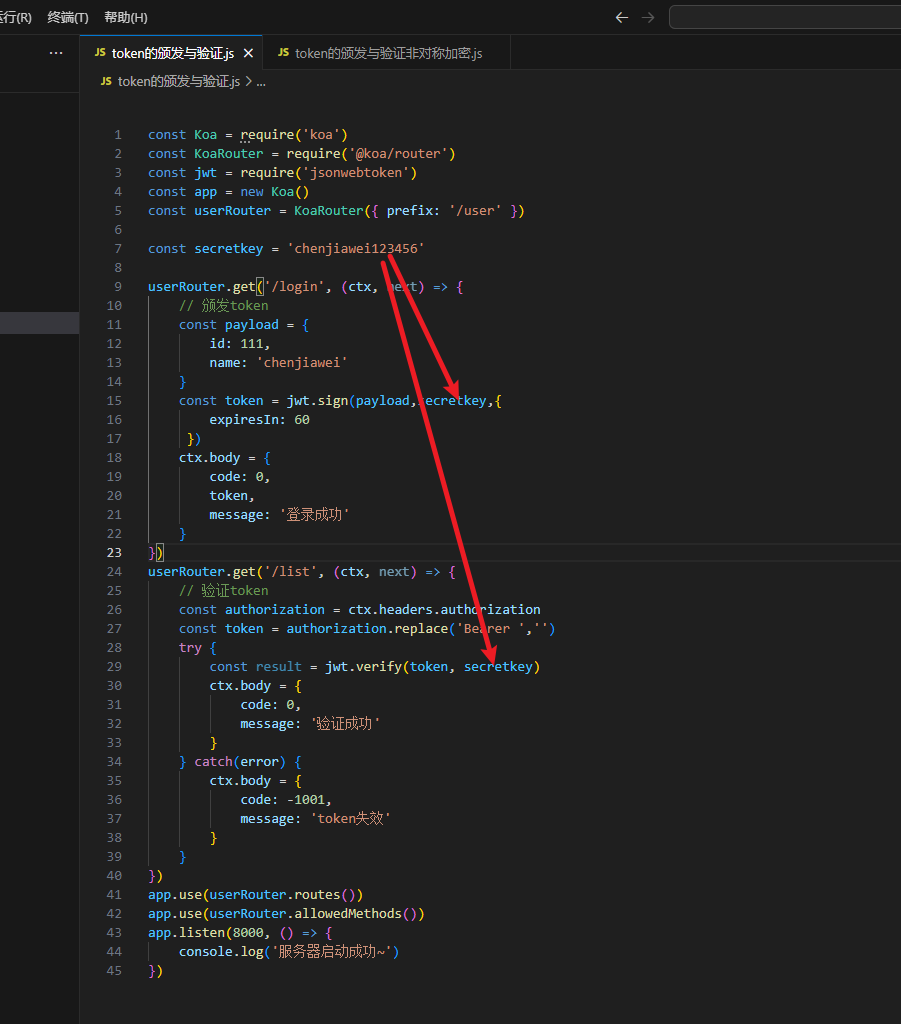
公钥是利用私钥才可以生成，这个是一对东西，-in是输入，把刚刚创建的私钥输入，然后-pubout是我要输出一个公钥 -out书输出到当前文件夹下，public.key就是名字



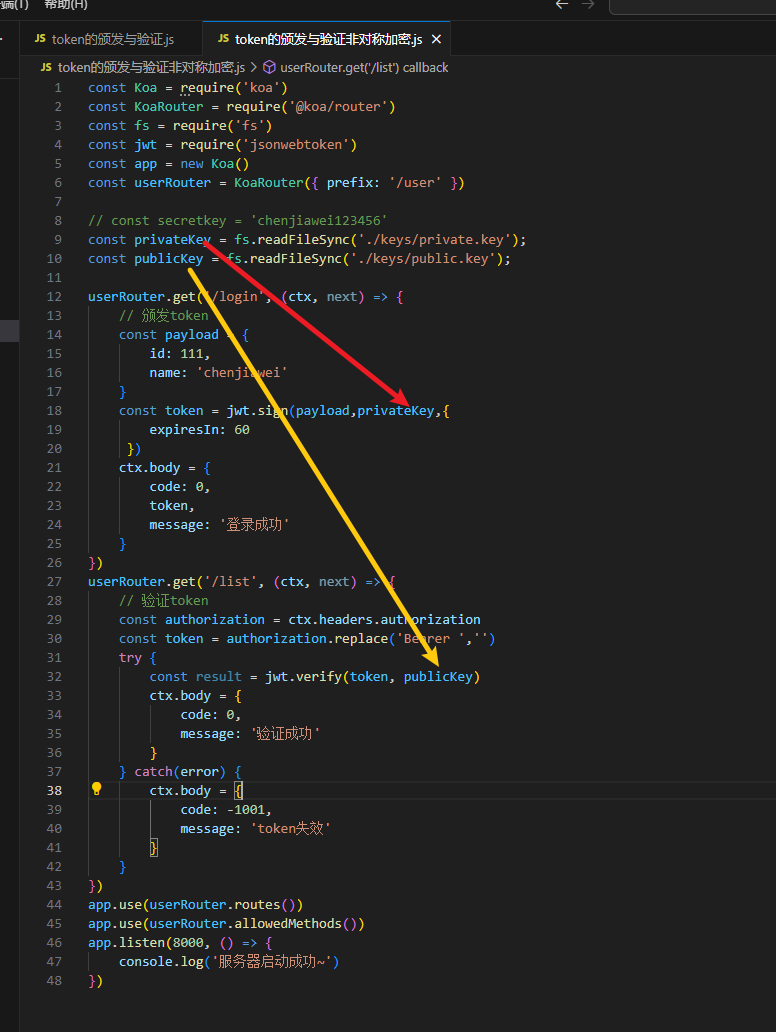
这里就可以看到我们的公钥，这样就有一个私钥和公钥了



把刚刚代码复制一份出来



原来是共用的同一串密钥



这里引用一下node的file system模块