



黑马程序员™
www.itheima.com

传智播客旗下
高端IT教育品牌

前后端的身份认证

目录 Contents

- ◆ 数据库的基本概念
- ◆ 安装并配置 MySQL
- ◆ MySQL的基本使用
- ◆ 在项目中操作 MySQL
- ◆ 前后端的身份认证

5. 前后端的身份认证

5.1 Web 开发模式

目前主流的 Web 开发模式有两种，分别是：

- ① 基于服务端渲染的传统 Web 开发模式
- ② 基于前后端分离的新型 Web 开发模式

5. 前后端的身份认证

5.1 Web 开发模式

1. 服务端渲染的 Web 开发模式

服务端渲染的概念：服务器发送给客户端的 HTML 页面，是在服务器通过字符串的拼接，动态生成的。因此，客户端不需要使用 Ajax 这样的技术额外请求页面的数据。代码示例如下：

```
1 app.get('/index.html', (req, res) => {
2   // 1. 要渲染的数据
3   const user = { name: 'zs', age: 20 }
4   // 2. 服务器端通过字符串的拼接，动态生成 HTML 内容
5   const html = `

# 姓名: ${user.name}, 年龄: ${user.age}</h1>` 6 // 3. 把生成好的页面内容响应给客户端。因此，客户端拿到的是带有真实数据的 HTML 页面 7 res.send(html) 8 })


```

■ 5. 前后端的身份认证

5.1 Web 开发模式

2. 服务端渲染的优缺点

优点：

- ① **前端耗时少。**因为服务器端负责动态生成 HTML 内容，浏览器只需要直接渲染页面即可。尤其是移动端，更省电。
- ② **有利于SEO。**因为服务器端响应的是完整的 HTML 页面内容，所以爬虫更容易爬取获得信息，更有利于 SEO。

缺点：

- ① **占用服务器端资源。**即服务器端完成 HTML 页面内容的拼接，如果请求较多，会对服务器造成一定的访问压力。
- ② **不利于前后端分离，开发效率低。**使用服务器端渲染，则**无法进行分工合作**，尤其对于**前端复杂度高**的项目，不利于项目高效开发。

5. 前后端的身份认证

5.1 Web 开发模式

3. 前后端分离的 Web 开发模式

前后端分离的概念：前后端分离的开发模式，**依赖于 Ajax 技术的广泛应用**。简而言之，前后端分离的 Web 开发模式，就是**后端只负责提供 API 接口，前端使用 Ajax 调用接口**的开发模式。

■ 5. 前后端的身份认证

5.1 Web 开发模式

4. 前后端分离的优缺点

优点：

- ① **开发体验好。** 前端专注于 UI 页面的开发，后端专注于api 的开发，且前端有更多的选择性。
- ② **用户体验好。** Ajax 技术的广泛应用，极大的提高了用户的体验，可以轻松实现页面的局部刷新。
- ③ **减轻了服务器端的渲染压力。** 因为页面最终是在每个用户的浏览器中生成的。

缺点：

- ① **不利于 SEO。** 因为完整的 HTML 页面需要在客户端动态拼接完成，所以爬虫对无法爬取页面的有效信息。（解决方案：利用 Vue、React 等前端框架的 **SSR**（server side render）技术能够很好的解决 SEO 问题！）



5. 前后端的身份认证

5.1 Web 开发模式

5. 如何选择 Web 开发模式

不谈业务场景而盲目选择使用何种开发模式都是耍流氓。

- 比如企业级网站，主要功能是展示而没有复杂的交互，并且需要良好的 SEO，则这时我们就需要使用服务器端渲染；
- 而类似后台管理项目，交互性比较强，不需要考虑 SEO，那么就可以使用前后端分离的开发模式。

另外，具体使用何种开发模式并不是绝对的，为了**同时兼顾**了**首页的渲染速度**和**前后端分离的开发效率**，一些网站采用了
首屏服务器端渲染 + 其他页面前后端分离的开发模式。

■ 5. 前后端的身份认证

5.2 身份认证

1. 什么是身份认证

身份认证 (Authentication) 又称“身份验证”、“鉴权”，是指**通过一定的手段，完成对用户身份的确认**。

- 日常生活中的身份认证随处可见，例如：高铁的验票乘车，手机的密码或指纹解锁，支付宝或微信的支付密码等。
- 在 Web 开发中，也涉及到用户身份的认证，例如：各大网站的**手机验证码登录**、**邮箱密码登录**、**二维码登录**等。



5. 前后端的身份认证

5.2 身份认证

2. 为什么需要身份认证

身份认证的目的，是为了**确认当前所声称某种身份的用户，确实是所声称的用户**。例如，你去找快递员取快递，你要怎么证明这份快递是你的。

在互联网项目开发中，如何对用户的身份进行认证，是一个值得深入探讨的问题。例如，如何才能保证网站不会错误的将“马云的存款数额”显示到“马化腾的账户”上。

5. 前后端的身份认证

5.2 身份认证

3. 不同开发模式下的身份认证

对于服务端渲染和前后端分离这两种开发模式来说，分别有着不同的身份认证方案：

- ① 服务端渲染推荐使用 Session 认证机制
- ② 前后端分离推荐使用 JWT 认证机制

5. 前后端的身份认证

5.3 Session 认证机制

1. HTTP 协议的无状态性

了解 HTTP 协议的无状态性是进一步学习 Session 认证机制的必要前提。

HTTP 协议的无状态性，指的是客户端的**每次 HTTP 请求都是独立的**，连续多个请求之间没有直接的关系，**服务器不会主动保留每次 HTTP 请求的状态**。

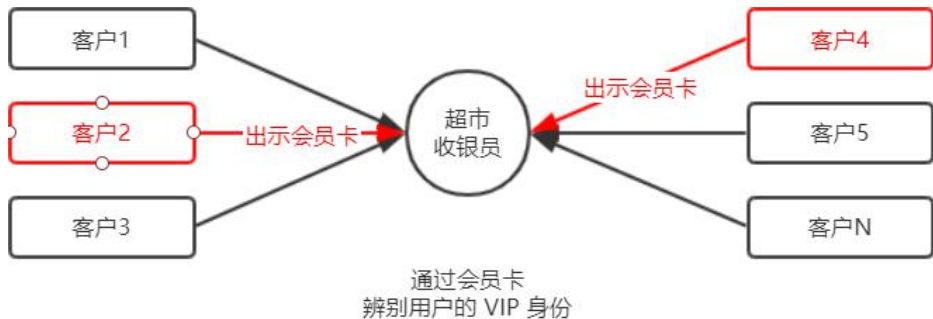


5. 前后端的身份认证

5.3 Session 认证机制

2. 如何突破 HTTP 无状态的限制

对于超市来说，为了方便收银员在进行结算时给 VIP 用户打折，超市可以为每个 VIP 用户发放会员卡。



注意：现实生活中的**会员卡身份认证方式**，在 Web 开发中的**专业术语**叫做 **Cookie**。



5. 前后端的身份认证

5.3 Session 认证机制

3. 什么是 Cookie

Cookie 是**存储在用户浏览器中的一段不超过 4 KB 的字符串**。它由一个**名称** (Name) 、一个**值** (Value) 和其它几个用于控制 Cookie **有效期**、**安全性**、**使用范围**的**可选属性**组成。

不同域名下的 Cookie 各自独立，每当客户端发起请求时，会**自动把当前域名下所有未过期的 Cookie**一同发送到服务器。

Cookie的几大特性：

- ① 自动发送
- ② 域名独立
- ③ 过期时限
- ④ 4KB 限制

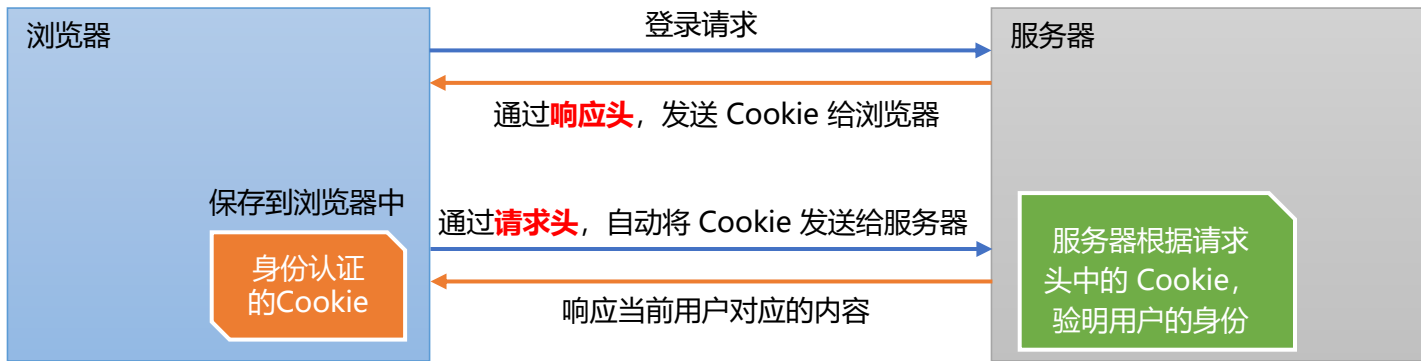
5. 前后端的身份认证

5.3 Session 认证机制

4. Cookie 在身份认证中的作用

客户端第一次请求服务器的时候，服务器**通过响应头的形式**，向客户端发送一个身份认证的 Cookie，客户端会自动将 Cookie 保存在浏览器中。

随后，当客户端浏览器每次请求服务器的时候，浏览器会**自动**将身份认证相关的 Cookie，**通过请求头的形式**发送给服务器，服务器即可验明客户端的身份。



5. 前后端的身份认证

5.3 Session 认证机制

5. Cookie 不具有安全性

由于 Cookie 是存储在浏览器中的，而且浏览器也提供了读写 Cookie 的 API，因此 Cookie 很容易被伪造，不具有安全性。因此不建议服务器将重要的隐私数据，通过 Cookie 的形式发送给浏览器。



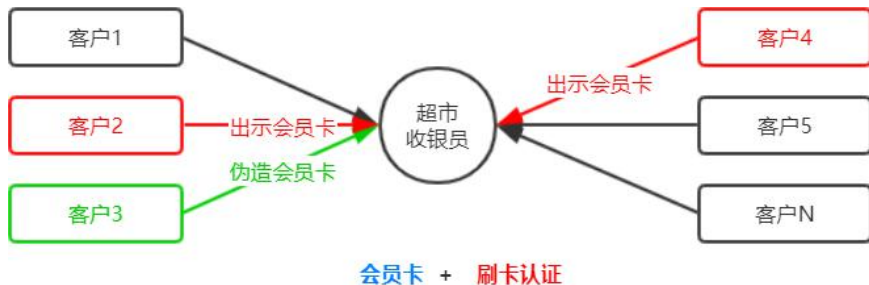
注意：千万不要使用 Cookie 存储重要且隐私的数据！比如用户的身份信息、密码等。

5. 前后端的身份认证

5.3 Session 认证机制

6. 提高身份认证的安全性

为了防止客户伪造会员卡，收银员在拿到客户出示的会员卡之后，可以在**收银机上进行刷卡认证**。只有收银机确认存在的会员卡，才能被正常使用。

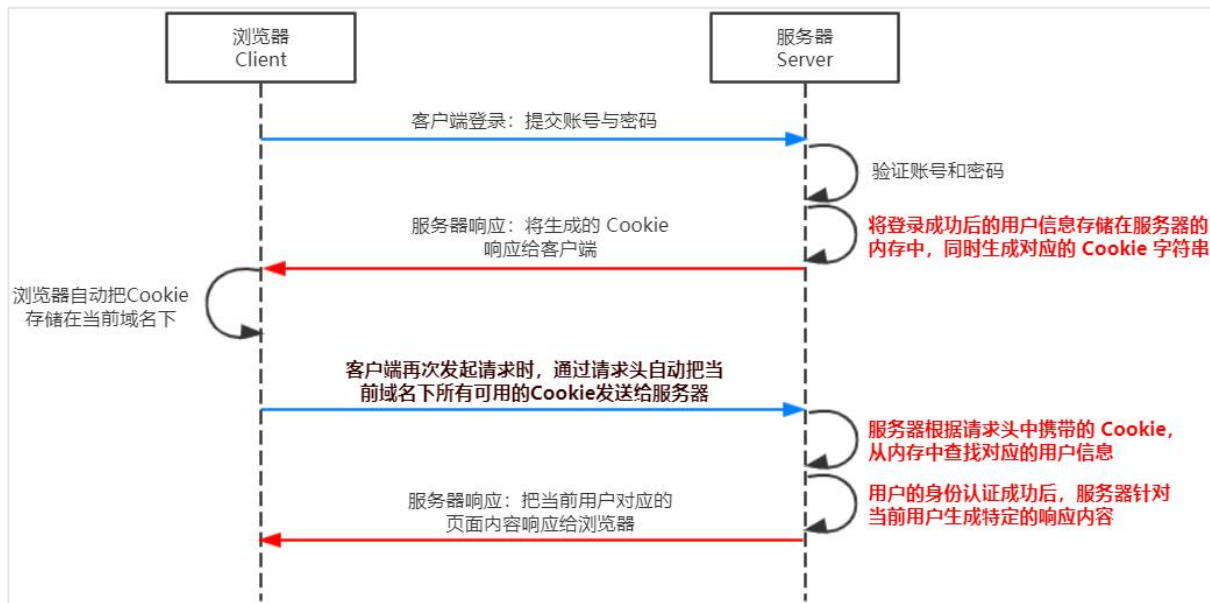


这种“**会员卡** + **刷卡认证**”的设计理念，就是 Session 认证机制的精髓。

5. 前后端的身份认证

5.3 Session 认证机制

7. Session 的工作原理



5. 前后端的身份认证

5.4 在 Express 中使用 Session 认证

1. 安装 express-session 中间件

在 Express 项目中，只需要安装 `express-session` 中间件，即可在项目中使用 Session 认证：

```
1 npm install express-session
```

5. 前后端的身份认证

5.4 在 Express 中使用 Session 认证

2. 配置 express-session 中间件

express-session 中间件安装成功后，需要通过 `app.use()` 来注册 session 中间件，示例代码如下：

```
1 // 1. 导入 session 中间件
2 var session = require('express-session')
3
4 // 2. 配置 Session 中间件
5 app.use(session({
6   secret: 'keyboard cat', // secret 属性的值可以为任意字符串
7   resave: false,          // 固定写法
8   saveUninitialized: true // 固定写法
9 })))
```

5. 前后端的身份认证

5.4 在 Express 中使用 Session 认证

3. 向 session 中存数据

当 express-session 中间件配置成功后，即可通过 **req.session** 来访问和使用 session 对象，从而存储用户的关键信息：

```
1 app.post('/api/login', (req, res) => {
2   // 判断用户提交的登录信息是否正确
3   if (req.body.username !== 'admin' || req.body.password !== '000000') {
4     return res.send({ status: 1, msg: '登录失败' })
5   }
6
7   req.session.user = req.body // 将用户的信息, 存储到 Session 中
8   req.session.islogin = true // 将用户的登录状态, 存储到 Session 中
9
10  res.send({ status: 0, msg: '登录成功' })
11 })
```

5. 前后端的身份认证

5.4 在 Express 中使用 Session 认证

4. 从 session 中取数据

可以直接从 `req.session` 对象上获取之前存储的数据，示例代码如下：

```
1 // 获取用户姓名的接口
2 app.get('/api/username', (req, res) => {
3   // 判断用户是否登录
4   if (!req.session.islogin) {
5     return res.send({ status: 1, msg: 'fail' })
6   }
7   res.send({ status: 0, msg: 'success', username: req.session.user.username })
8 })
```

5. 前后端的身份认证

5.4 在 Express 中使用 Session 认证

5. 清空 session

调用 `req.session.destroy()` 函数，即可清空服务器保存的 session 信息。

```
1 // 退出登录的接口
2 app.post('/api/logout', (req, res) => {
3   // 清空当前客户端对应的 session 信息
4   req.session.destroy()
5   res.send({
6     status: 0,
7     msg: '退出登录成功'
8   })
9 })
```

5. 前后端的身份认证

5.5 JWT 认证机制

1. 了解 Session 认证的局限性

Session 认证机制**需要配合 Cookie 才能实现**。由于 Cookie 默认不支持跨域访问，所以，当涉及到**前端跨域请求后端接口**的时候，**需要做很多额外的配置**，才能实现跨域 Session 认证。

注意：

- 当前端请求后端接口**不存在跨域问题**的时候，**推荐使用 Session** 身份认证机制。
- 当前端需要跨域请求后端接口的时候，不推荐使用 Session 身份认证机制，推荐使用 JWT 认证机制。

5. 前后端的身份认证

5.5 JWT 认证机制

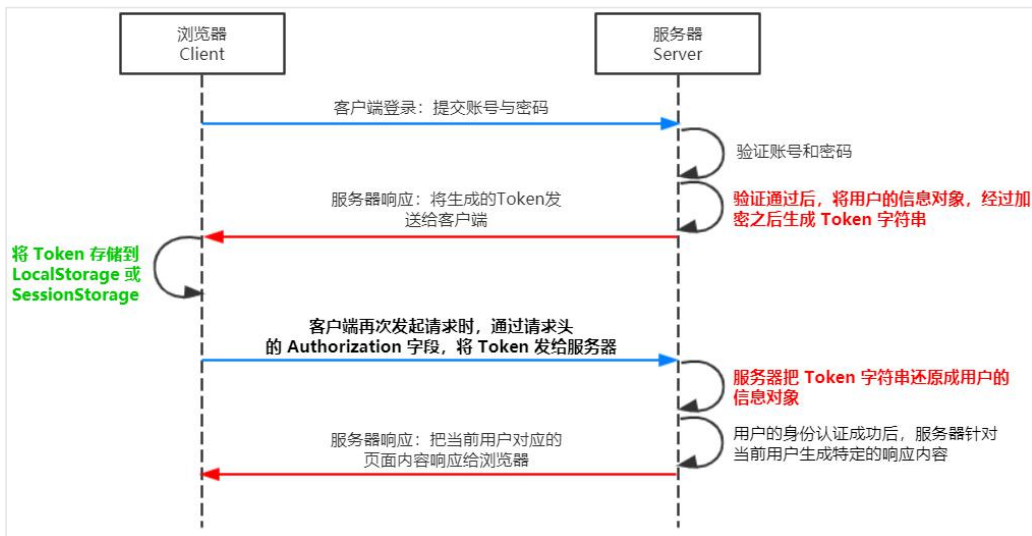
2. 什么是 JWT

JWT (英文全称: JSON Web Token) 是目前最流行的跨域认证解决方案。

5. 前后端的身份认证

5.5 JWT 认证机制

3. JWT 的工作原理



总结：用户的信息通过 Token 字符串的形式，保存在客户端浏览器中。服务器通过还原 Token 字符串的形式来认证用户的身份。

5. 前后端的身份认证

5.5 JWT 认证机制

4. JWT 的组成部分

JWT 通常由三部分组成，分别是 **Header**（头部）、**Payload**（有效荷载）、**Signature**（签名）。

三者之间使用英文的 “.” 分隔，格式如下：

```
1 Header.Payload.Signature
```

下面是 JWT 字符串的示例：

```
1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJhZG1pbSI6InBhc3N3b3JkIjoIiWibmlja25hbnwUiOiIms6X1t7T1t7QilLCJlbWFPbCI6Im5pYmFiYUBpdG9hZC3QuY24iLCJ1c2VyX3BpYyI6IiIsIm1hdCI6MTU3ODAzNjY4MiwiaXZhwIjoxNTc4MDcyNjgyfQ.Mwq7GqCxJPK-EA8LNrtMG041lKdZ33S9KBL3XeuBxuI
```

5. 前后端的身份认证

5.5 JWT 认证机制

6. JWT 的三个部分各自代表的含义

JWT 的三个组成部分，从前到后分别是 Header、Payload、Signature。

其中：

- **Payload** 部分**才是真正的用户信息**，它是用户信息经过加密之后生成的字符串。
- Header 和 Signature 是**安全性相关**的部分，只是为了保证 Token 的安全性。





5. 前后端的身份认证

5.5 JWT 认证机制

7. JWT 的使用方式

客户端收到服务器返回的 JWT 之后，通常会将它储存在 `localStorage` 或 `sessionStorage` 中。

此后，客户端每次与服务器通信，都要带上这个 JWT 的字符串，从而进行身份认证。推荐的做法是**把 JWT 放在 HTTP 请求头的 `Authorization` 字段中**，格式如下：

```
1 Authorization: Bearer <token>
```

5. 前后端的身份认证

5.6 在 Express 中使用 JWT

1. 安装 JWT 相关的包

运行如下命令，安装如下两个 JWT 相关的包：

```
1 npm install jsonwebtoken express-jwt
```

其中：

- **jsonwebtoken** 用于生成 JWT 字符串
- **express-jwt** 用于将 JWT 字符串解析还原成 JSON 对象



5. 前后端的身份认证

5.6 在 Express 中使用 JWT

2. 导入 JWT 相关的包

使用 `require()` 函数，分别导入 JWT 相关的两个包：

```
1 // 1. 导入用于生成 JWT 字符串的包
2 const jwt = require('jsonwebtoken')
3 // 2. 导入用于将客户端发送过来的 JWT 字符串，解析还原成 JSON 对象的包
4 const expressJWT = require('express-jwt')
```



5. 前后端的身份认证

5.6 在 Express 中使用 JWT

3. 定义 secret 密钥

为了保证 JWT 字符串的安全性，防止 JWT 字符串在网络传输过程中被别人破解，我们需要专门定义一个用于加密和解密的 secret 密钥：

- ① 当生成 JWT 字符串的时候，需要使用 secret 密钥对用户的信息进行加密，最终得到加密好的 JWT 字符串
- ② 当把 JWT 字符串解析还原成 JSON 对象的时候，需要使用 secret 密钥进行解密

```
1 // 3. secret 密钥的本质：就是一个字符串
2 const secretKey = 'itheima No1 ^_^'
```


5. 前后端的身份认证

5.6 在 Express 中使用 JWT

4. 在登录成功后生成 JWT 字符串

调用 `jsonwebtoken` 包提供的 `sign()` 方法，将用户的信息加密成 JWT 字符串，响应给客户端：

```
1 // 登录接口
2 app.post('/api/login', function(req, res) {
3   // ... 省略登录失败情况下的代码
4   // 用户登录成功之后，生成 JWT 字符串，通过 token 属性响应给客户端
5   res.send({
6     status: 200,
7     message: '登录成功! ',
8     // 调用 jwt.sign() 生成 JWT 字符串，三个参数分别是：用户信息对象、加密密钥、配置对象
9     token: jwt.sign({ username: userinfo.username }, secretKey, { expiresIn: '30s' })
10  })
11 })
```



5. 前后端的身份认证

5.6 在 Express 中使用 JWT

5. 将 JWT 字符串还原为 JSON 对象

客户端每次在访问那些有权限接口的时候，都需要主动通过请求头中的 **Authorization 字段**，将 Token 字符串发送到服务器进行身份认证。

此时，服务器可以通过 **express-jwt** 这个中间件，自动将客户端发送过来的 Token 解析还原成 JSON 对象：



```
1 // 使用 app.use() 来注册中间件
2 // expressJWT({ secret: secretKey }) 就是用来解析 Token 的中间件
3 // .unless({ path: [/^\/api\//] }) 用来指定哪些接口不需要访问权限
4 app.use(expressJWT({ secret: secretKey }).unless({ path: [/^\/api\//] })))
```



5. 前后端的身份认证

5.6 在 Express 中使用 JWT

6. 使用 req.user 获取用户信息

当 express-jwt 这个中间件配置成功之后，即可在那些有权限的接口中，使用 **req.user** 对象，来访问从 JWT 字符串中解析出来的用户信息了，示例代码如下：

```
1 // 这是一个有权限的 API 接口
2 app.get('/admin/getinfo', function(req, res) {
3   console.log(req.user)
4   res.send({
5     status: 200,
6     message: '获取用户信息成功! ',
7     data: req.user
8   })
9 })
```

5. 前后端的身份认证

5.6 在 Express 中使用 JWT

7. 捕获解析 JWT 失败后产生的错误

当使用 express-jwt 解析 Token 字符串时，如果客户端发送过来的 Token 字符串**过期**或**不合法**，会产生一个**解析失败**的错误，影响项目的正常运行。我们可以通过 **Express 的错误中间件**，捕获这个错误并进行相关的处理，示例代码如下：

```
1 app.use((err, req, res, next) => {
2   // token 解析失败导致的错误
3   if(err.name === 'UnauthorizedError') {
4     return res.send({ status: 401, message: '无效的token' })
5   }
6   // 其它原因导致的错误
7   res.send({ status: 500, message: '未知错误' })
8 })
```



传智播客旗下高端IT教育品牌