cookie-session

cookie和session的区别及使用

众所周知，HTTP 是一个无状态协议，所以客户端每次发出请求时，下一次请求无法得知上一次请求所包含的状态数据，如何能把一个用户的状态数据关联起来呢?

cookie

首先产生了 cookie 这门技术来解决这个问题，cookie 是 http 协议的一部分，它的处理分为如下几步：

服务器向客户端发送 cookie。
通常使用 HTTP 协议规定的 set-cookie 头操作。
规范规定 cookie 的格式为 name = value 格式，且必须包含这部分。
浏览器将 cookie 保存。
每次请求浏览器都会将 cookie 发向服务器。
其他可选的 cookie 参数会影响将 cookie 发送给服务器端的过程，主要有以下几种：

`path：表示 cookie 影响到的路径，匹配该路径才发送这个 cookie。`
`expires 和 maxAge：告诉浏览器这个 cookie 什么时候过期，expires 是 UTC 格式时间，maxAge 是 cookie 多久后过期的相对时间。`
`当不设置这两个选项时，会产生 session cookie，session cookie 是 transient 的，当用户关闭浏览器时，就被清除。`
`一般用来保存 session 的 session_id。`
`secure：当 secure 值为 true 时，cookie 在 HTTP 中是无效，在 HTTPS 中才有效。`
`httpOnly：浏览器不允许脚本操作 document.cookie 去更改 cookie。一般情况下都应该设置这个为 true，这样可以避免被 xss 攻击拿到 cookie。`

express 中的 cookie

express 在 4.x 版本之后，session管理和cookies等许多模块都不再直接包含在express中，而是需要单独添加相应模块。

express4 中操作 cookie 使用 cookie-parser 模块(https://github.com/expressjs/cookie-parser)。
```

var express = require('express');
```

```
// 首先引入 cookie-parser 这个模块
var cookieParser = require('cookie-parser');

var app = express();
app.listen(3000);

// 使用 cookieParser 中间件，cookieParser(secret, options)
// 其中 secret 用来加密 cookie 字符串（下面会提到 signedCookies）
// options 传入上面介绍的 cookie 可选参数
app.use(cookieParser());

app.get('/', function (req, res) {
// 如果请求中的 cookie 存在 isVisit, 则输出 cookie
// 否则，设置 cookie 字段 isVisit, 并设置过期时间为1分钟
if (req.cookies.isVisit) {
console.log(req.cookies);
res.send("再次欢迎访问");
} else {
res.cookie('isVisit', 1, {maxAge: 60 * 1000});
res.send("欢迎第一次访问");
}
});
```

session

cookie 虽然很方便，但是使用 cookie 有一个很大的弊端，cookie 中的所有数据在客户端就可以被修改，数据非常容易被伪造，那么一些重要的数据就不能存放在 cookie 中了，而且如果 cookie 中数据字段太多会影响传输效率。为了解决这些问题，就产生了 session，session 中的数据是保留在服务器端的。

session 的运作通过一个 session_id 来进行。session_id 通常是存放在客户端的 cookie 中，比如在 express 中，默认是 connect.sid 这个字段，当请求到来时，服务端检查 cookie 中保存的 session_id 并通过这个 session_id 与服务器端的 session data 关联起来，进行数据的保存和修改。

这意思就是说，当你浏览一个网页时，服务端随机产生一个 1024 比特长的字符串，然后存在你 cookie 中的connect.sid 字段中。当你下次访问时，cookie 会带有这个字符串，然后浏览器就知道你是上次访问过的某某某，然后从服务器的存储中取出上次记录在你身上的数据。由于字符串是随机产生的，而且位数足够多，所以也不担心有人能够伪造。伪造成功的概率比坐在家里编程

时被邻居家的狗突然闯入并咬死的几率还低。

session 可以存放在 1）内存、2）cookie本身、3）redis 或 memcached 等缓存中，或者4）数据库中。线上来说，缓存的方案比较常见，存数据库的话，查询效率相比前三者都太低，不推荐；cookie session 有安全性问题，下面会提到。

express 中操作 session 要用到 express-session (https://github.com/expressjs/session) 这个模块，主要的方法就是session(options)，其中 options 中包含可选参数，主要有：

`name: 设置 cookie 中，保存 session 的字段名称，默认为 connect.sid 。`
`store: session 的存储方式，默认存放在内存中，也可以使用 redis，mongodb 等。express 生态中都有相应模块的支持。`
`secret: 通过设置的 secret 字符串，来计算 hash 值并放在 cookie 中，使产生的 signedCookie 防篡改。`
`cookie: 设置存放 session id 的 cookie 的相关选项，默认为 (default: { path: '/', httpOnly: true, secure: false, maxAge: null })`
`genid: 产生一个新的 session_id 时，所使用的函数， 默认使用 uid2 这个 npm 包。`
`rolling: 每个请求都重新设置一个 cookie，默认为 false。`
`resave: 即使 session 没有被修改，也保存 session 值，默认为 true。`

1） 在内存中存储 session

express-session 默认使用内存来存 session，对于开发调试来说很方便。

```
var express = require('express');
// 首先引入 express-session 这个模块
var session = require('express-session');

var app = express();
app.listen(5000);

// 按照上面的解释，设置 session 的可选参数
app.use(session({
secret: 'recommand 128 bytes random string', // 建议使用 128 个字符的随机字符串
cookie: { maxAge: 60 * 1000 }
}));

app.get('/', function (req, res) {

// 检查 session 中的 isVisit 字段
// 如果存在则增加一次，否则为 session 设置 isVisit 字段，并初始化为 1。
```

```
if(req.session.isVisit) {
req.session.isVisit++;
res.send('

第 ' + req.session.isVisit + '次来此页面

');
} else {
req.session.isVisit = 1;
res.send("欢迎第一次来这里");
console.log(req.session);
}
});
```

2） 在 redis 中存储 session

session 存放在内存中不方便进程间共享，因此可以使用 redis 等缓存来存储 session。

假设你的机器是 4 核的，你使用了 4 个进程在跑同一个 node web 服务，当用户访问进程1时，他被设置了一些数据当做 session 存在内存中。而下一次访问时，他被负载均衡到了进程2，则此时进程2的内存中没有他的信息，认为他是个新用户。这就会导致用户在我们服务中的状态不一致。

使用 redis 作为缓存，可以使用 connect-redis 模块(https://github.com/tj/connect-redis)来得到 redis 连接实例，然后在 session 中设置存储方式为该实例。

```
var express = require('express');
var session = require('express-session');
var redisStore = require('connect-redis')(session);

var app = express();
app.listen(5000);

app.use(session({
// 假如你不想使用 redis 而想要使用 memcached 的话，代码改动也不会超过 5 行。
// 这些 store 都遵循着统一的接口，凡是实现了那些接口的库，都可以作为 session 的 store 使用，比如都需要实现 .get(keyString) 和 .set(keyString, value) 方法。
// 编写自己的 store 也很简单
store: new redisStore(),
secret: 'somesecrettoken'
}));
```

```
app.get('/', function (req, res) {
if(req.session.isVisit) {
req.session.isVisit++;
res.send('

第 ' + req.session.isVisit + '次来到此页面

');
} else {
req.session.isVisit = 1;
res.send('欢迎第一次来这里');
}
});
```

各种存储的利弊

上面我们说到，session 的 store 有四个常用选项：1）内存 2）cookie 3）缓存 4）数据库

其中，开发环境存内存就好了。一般的小程序为了省事，如果不涉及状态共享的问题，用内存 session 也没问题。但内存 session 除了省事之外，没有别的好处。

cookie session 我们下面会提到，现在说说利弊。用 cookie session 的话，是不用担心状态共享问题的，因为 session 的 data 不是由服务器来保存，而是保存在用户浏览器端，每次用户访问时，都会主动带上他自己的信息。当然在这里，安全性之类的，只要遵照最佳实践来，也是有保证的。它的弊端是增大了数据量传输，利端是方便。

缓存方式是最常用的方式了，即快，又能共享状态。相比 cookie session 来说，当 session data 比较大的时候，可以节省网络传输。推荐使用。

数据库 session。除非你很熟悉这一块，知道自己要什么，否则还是老老实实用缓存吧。

signedCookie

上面都是讲基础，现在讲一些专业点的。

上面有提到

cookie 虽然很方便，但是使用 cookie 有一个很大的弊端，cookie 中的所有数据在客户端就可以被修改，数据非常容易被伪造
其实不是这样的，那只是为了方便理解才那么写。要知道，计算机领域有个名词叫 签名，专业点说，叫 信息摘要算法。

比如我们现在面临着一个菜鸟开发的网站，他用 cookie 来记录登陆的用户凭证。相应的 cookie 长这样：dotcom_user=alsotang，它说明现在的用户是 alsotang 这个用户。如果我在浏览器中装个插件，把它改成dotcom_user=ricardo，服务器一读取，就会误认为我是 ricardo。然后我就可以进行 ricardo 才能进行的操作了。之前 web 开发不成熟的时候，用这招甚至可以黑个网站下来，把 cookie 改成 dotcom_user=admin 就行了，唉，那是个玩黑客的黄金年代啊。

OK，现在我有一些数据，不想存在 session 中，想存在 cookie 中，怎么保证不被篡改呢？答案很简单，签个名。

假设我的服务器有个秘密字符串，是 this_is_my_secret_and_fuck_you_all，我为用户 cookie 的 dotcom_user 字段设置了个值 alsotang。cookie 本应是

`{dotcom_user: 'alsotang'}`
这样的。

而如果我们签个名，比如把 dotcom_user 的值跟我的 secret_string 做个 sha1

```
sha1('this_is_my_secret_and_fuck_you_all' + 'alsotang') ===
'4850a42e3bc0d39c978770392cbd8dc2923e3d1d'
```

然后把 cookie 变成这样

```
{
dotcom_user: 'alsotang',
'dotcom_user.sig': '4850a42e3bc0d39c978770392cbd8dc2923e3d1d',
}
```

这样一来，用户就没法伪造信息了。一旦它更改了 cookie 中的信息，则服务器会发现 hash 校验的不一致。

毕竟他不懂我们的 secret_string 是什么，而暴力破解哈希值的成本太高。

cookie-session

上面一直提到 session 可以存在 cookie 中，现在来讲讲具体的思路。这里所涉及的专业名词叫做 对称加密。 假设我们想在用户的 cookie 中存 session data，使用一个名为 `session_data` 的字段。 存 `js var sessionData = {username: 'alsotang', age: 22, company: 'alibaba', location: 'hangzhou'}` 这段信息的话，可以将 `sessionData` 与我们的 `secret_string` 一起做个对称加密，存到 cookie 的 `session_data` 字段中，只要你的 `secret_string` 足够长，那么攻击者也是无法获取实际 session 内容的。对称加密之后的内容对于攻击者来说相当于一段乱码。而当用户下次访问时，我们就可以用 `secret_string` 来解密 `sessionData`，得到我们需要的 session data。 signedCookies 跟 cookie-session 还是有区别的： 1）是前者信息可见不可篡

改，后者不可见也不可篡改 2）是前者一般是长期保存，而后者是 session cookie
cookie-session 的实现跟 signedCookies 差不多。

不过 cookie-session 我个人建议不要使用，有受到回放攻击的危险。

回放攻击指的是，比如一个用户，它现在有 100 积分，积分存在 session 中，session 保存在 cookie 中。他先复制下现在的这段 cookie，然后去发个帖子，扣掉了 20 积分，于是他就只有 80 积分了。而他现在可以将之前复制下的那段 cookie 再粘贴回去浏览器中，于是服务器在一些场景下会认为他又有了 100 积分。

如果避免这种攻击呢？这就需要引入一个第三方的手段来验证 cookie session，而验证所需的信息，一定不能存在 cookie 中。这么一来，避免了这种攻击后，使用 cookie session 的好处就荡然无存了。如果为了避免攻击而引入了缓存使用的话，那不如把 cookie session 也一起放进缓存中。

session cookie

初学者容易犯的一个错误是，忘记了 session_id 在 cookie 中的存储方式是 session cookie。即，当用户一关闭浏览器，浏览器 cookie 中的 session_id 字段就会消失。

常见的场景就是在开发用户登陆状态保持时。

假如用户在之前登陆了你的网站，你在他对应的 session 中存了信息，当他关闭浏览器再次访问时，你还是不懂他是谁。所以我们要在 cookie 中，也保存一份关于用户身份的信息。

比如有这样一个用户

```
{username: 'alsotang', age: 22, company: 'alibaba', location: 'hangzhou'}
```

我们可以考虑把这四个字段的信息都存在 session 中，而在 cookie，我们用 signedCookies 来存个 username。

登陆的检验过程伪代码如下：

```
if (req.session.user) {
// 获取 user 并进行下一步
next()
} else if (req.signedCookies['username']) {
// 如果存在则从数据库中获取这个 username 的信息，并保存到 session 中
getuser(function (err, user) {
req.session.user = user;
next();
```

```
});
} else {
// 当做为登陆用户处理
next();
}
```