

数据库结构说明

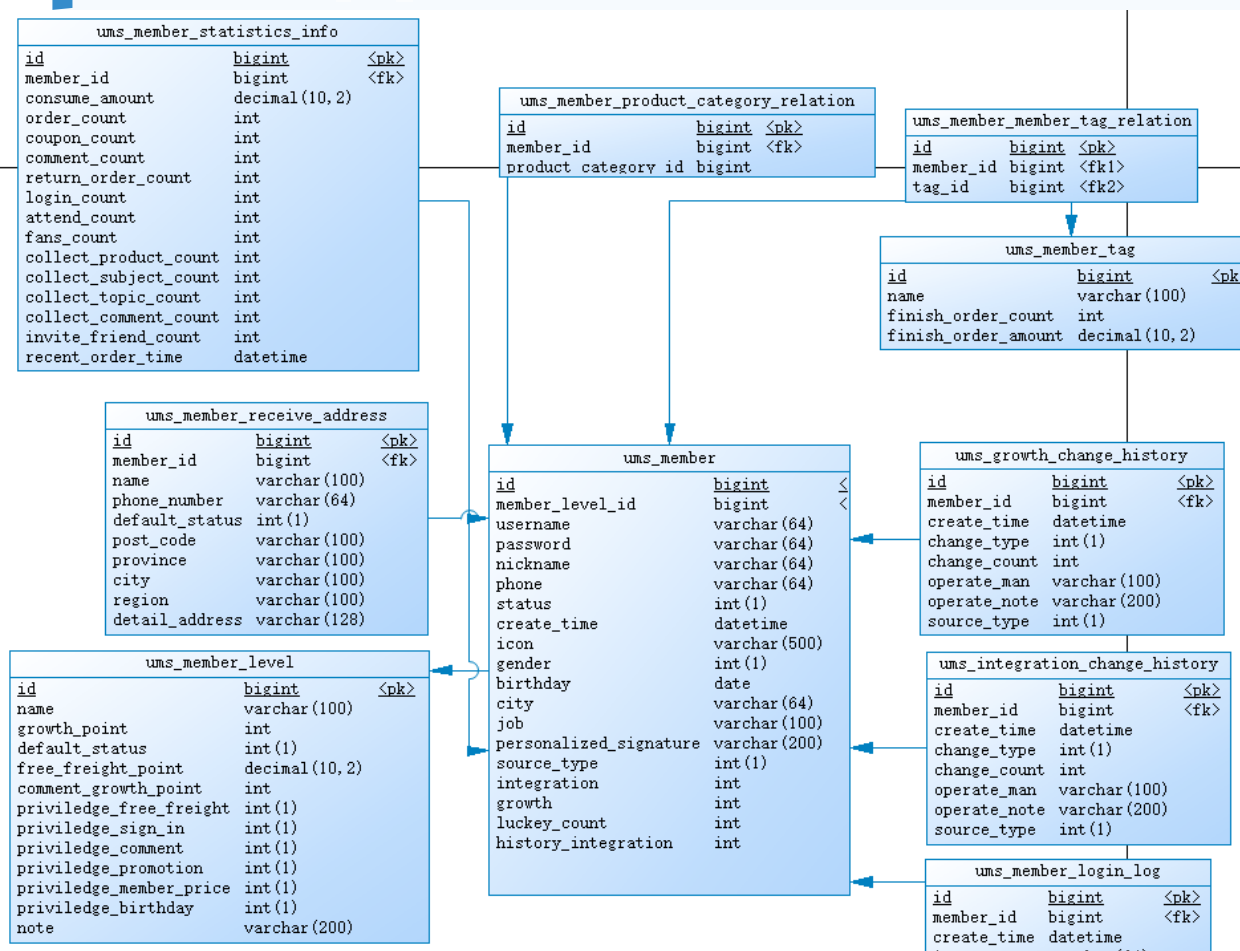
- cms_*: 内容管理模块相关表
- oms_*: 订单管理模块相关表
- pms_*: 商品模块相关表
- sms_*: 营销模块相关表
- ums_*: 会员模块相关表

会员模块从0到1过程

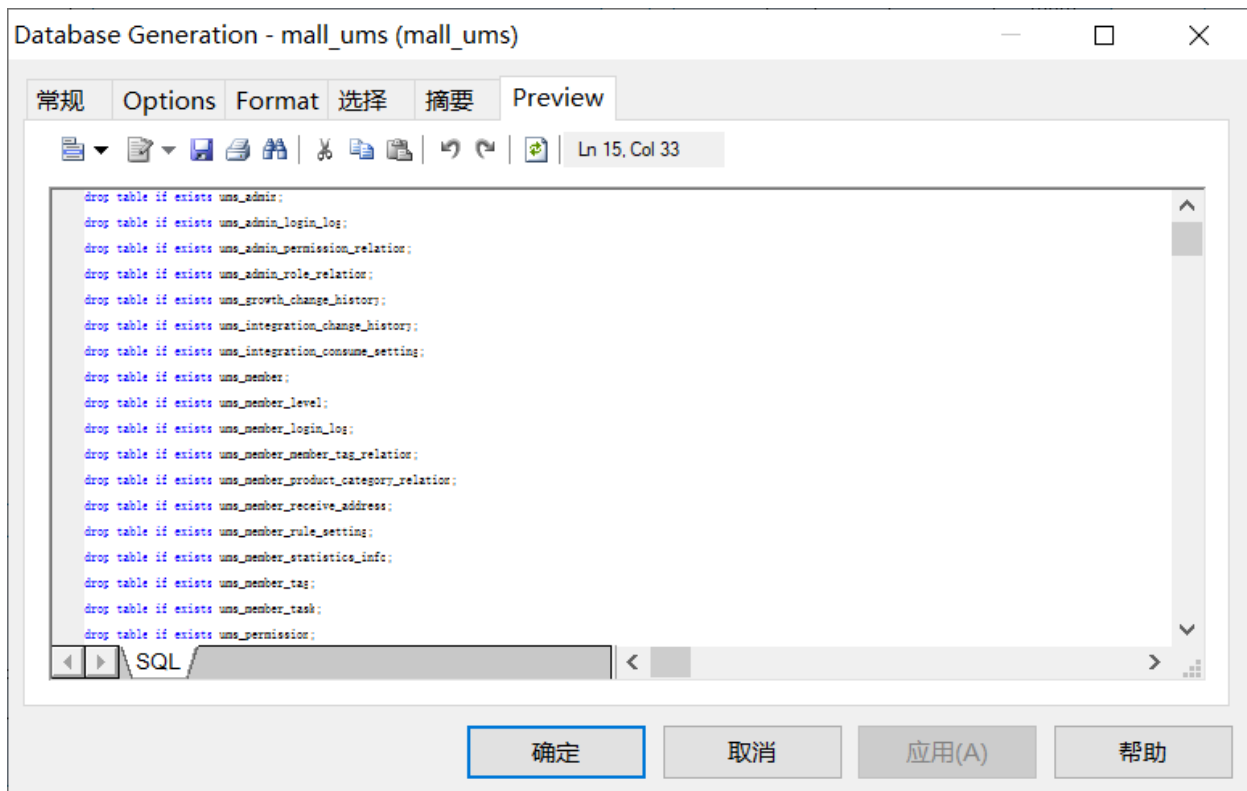
1、初始会员模块表结构

会员模块表间结构关系：

打开项目：document/pdm/mall_ums.pdm,表关系与结构做了相关描述



表结构映射到MySQL库micromall:



2、创建mall-member工程

在商城工程目录下创建mall-member子工程

- ▶ mall-mbg
- ▶ mall-member
- ▶ mall-monitor
- ▶ **mall-portal**
- ▶ mall-registry
- ▶ mall-search

3、MBG逆向工程

3.1、引入依赖

```
<!-- 依赖通用工程 -->
<dependency>
<groupId>com.macro.mall</groupId>
<artifactId>mall-common</artifactId>
</dependency>
<!-- 逆向工程依赖包 -->
<dependency>
<groupId>org.mybatis.generator</groupId>
<artifactId>mybatis-generator-core</artifactId>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
</dependency>
```

3.2 MBG配置

generator.properties-数据库连接

```
jdbc.driverClass=com.mysql.cj.jdbc.Driver
jdbc.connectionURL=jdbc:mysql://192.168.241.198:3306/micromall?serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=UTF-8
jdbc.userId=root
jdbc.password=root
```

3.3-逆向工程配置

generatorConfig.xml

```
<generatorConfiguration>
  <properties resource="generator.properties"/>
  <context id="MySQLContext" targetRuntime="MyBatis3"
    defaultModelType="flat">
    <property name="beginningDelimiter" value="`"/>
    <property name="endingDelimiter" value="`"/>
    <property name="javaFileEncoding" value="UTF-8"/>
    <!-- 为模型生成序列化方法-->
    <plugin type="org.mybatis.generator.plugins.SerializablePlugin"/>
    <!-- 为生成的Java模型创建一个toString方法 -->
    <plugin type="org.mybatis.generator.plugins.ToStringPlugin"/>
    <!--生成mapper.xml时覆盖原文件-->
    <plugin type="org.mybatis.generator.plugins.UnmergeableXmlMappersPlugin" />
    <commentGenerator type="mbg.CommentGenerator">
    <!-- 是否去除自动生成的注释 true: 是 : false:否 -->
    <property name="suppressAllComments" value="true"/>
    <property name="suppressDate" value="true"/>
    <property name="addRemarkComments" value="true"/>
    </commentGenerator>

    <jdbcConnection driverClass="${jdbc.driverClass}"
      connectionURL="${jdbc.connectionURL}"
      userId="${jdbc.userId}"
      password="${jdbc.password}">
    <!--解决mysql驱动升级到8.0后不生成指定数据库代码的问题-->
    <property name="nullCatalogMeansCurrent" value="true" />
    </jdbcConnection>
    <!--entity生成位置-->
    <javaModelGenerator targetPackage="com.macro.mall.model" targetProject="mall-member\src\main\java"/>
    <!--mapper生成位置-->
    <sqlMapGenerator targetPackage="com.macro.mall.mapper" targetProject="mall-member\src\main\resources"/>
    <!--mapper.xml生成位置在resource下-->
```

```

<javaClientGenerator type="XMLMAPPER" targetPackage="com.macro.mall.mapper"
targetProject="mall-member\src\main\java"/>
<!--生成全部表tableName设为%,指定前缀如下: 只逆向会员模块-->
<table tableName="ums_member%" enableCountByExample="false" enableUpdateByE
    xample="false"
enableDeleteByExample="false" enableSelectByExample="false"
selectByExampleQueryId="false">
<generatedKey column="id" sqlStatement="MySql" identity="true"/>
</table>
</context>
</generatorConfiguration>

```

自定义注释生成器CommentGenerator

```

/**
 * 自定义注释生成器
 */
public class CommentGenerator extends DefaultCommentGenerator {
    private boolean addRemarkComments = false;
    private static final String EXAMPLE_SUFFIX="Example";
    private static final String API_MODEL_PROPERTY_FULL_CLASS_NAME="io.swagger.
        annotations.ApiModelProperty";

    /**
     * 设置用户配置的参数
     */
    @Override
    public void addConfigurationProperties(Properties properties) {
        super.addConfigurationProperties(properties);
        this.addRemarkComments = StringUtility.isTrue(properties.getProperty("addRe
            markComments"));
    }

    /**
     * 给字段添加注释
     */
    @Override
    public void addFieldComment(Field field, IntrospectedTable introspectedTabl
        e,
        IntrospectedColumn introspectedColumn) {
        String remarks = introspectedColumn.getRemarks();
        //根据参数和备注信息判断是否添加备注信息
        if(addRemarkComments&&StringUtility.stringHasValue(remarks)){
            // addFieldJavaDoc(field, remarks);
            //数据库中特殊字符需要转义
            if(remarks.contains("\\")){

```

```

remarks = remarks.replace("\'", "'");
}
//给model的字段添加swagger注解
field.addJavaDocLine("@ApiModelProperty(value = \""+remarks+"\")");
}
}

/**
 * 给model的字段添加注释
 */
private void addFieldJavaDoc(Field field, String remarks) {
//文档注释开始
field.addJavaDocLine("/**");
//获取数据库字段的备注信息
String[] remarkLines = remarks.split(System.getProperty("line.separator"));
for(String remarkLine:remarkLines){
field.addJavaDocLine(" * "+remarkLine);
}
addJavadocTag(field, false);
field.addJavaDocLine(" */");
}

@Override
public void addJavaFileComment(CompilationUnit compilationUnit) {
super.addJavaFileComment(compilationUnit);
//只在model中添加swagger注解类的导入
if(!compilationUnit.isJavaInterface()&&!compilationUnit.getType().getFullyQualified().contains(EXAMPLE_SUFFIX)){
compilationUnit.addImportedType(new FullyQualifiedJavaType(API_MODEL_PROPERTY_FULL_CLASS_NAME));
}
}
}
}

```

执行逆向生成

```

public class Generator {
public static void main(String[] args) throws Exception {
//MBG 执行过程中的警告信息
List<String> warnings = new ArrayList<String>();
//当生成的代码重复时，覆盖原代码
boolean overwrite = true;
//读取我们的 MBG 配置文件
InputStream is =
    Generator.class.getResourceAsStream("/generatorConfig.xml");
ConfigurationParser cp = new ConfigurationParser(warnings);

```











































```

Configuration config = cp.parseConfiguration(is);
is.close();

DefaultShellCallback callback = new DefaultShellCallback(overwrite);
//创建 MBG
MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config, callback,
    warnings);
//执行生成代码
myBatisGenerator.generate(null);
//输出警告信息
for (String warning : warnings) {
    System.out.println(warning);
}
}
}
}

```

逆向生成结果：

- ▼  mapper
 -   UmsMemberLevelMapper
 -   UmsMemberLoginLogMapper
 -   UmsMemberMapper
 -   UmsMemberMemberTagRelationMapper
 -   UmsMemberProductCategoryRelationMapper
 -   UmsMemberReceiveAddressMapper
 -   UmsMemberRuleSettingMapper
 -   UmsMemberStatisticsInfoMapper
 -   UmsMemberTagMapper
 -   UmsMemberTaskMapper
- ▼  model
 -   UmsMember
 -   UmsMemberLevel
 -   UmsMemberLoginLog
 -   UmsMemberMemberTagRelation
 -   UmsMemberProductCategoryRelation
 -   UmsMemberReceiveAddress
 -   UmsMemberRuleSetting
 -   UmsMemberStatisticsInfo
 -   UmsMemberTag
 -   UmsMemberTask

4、会员登录会话

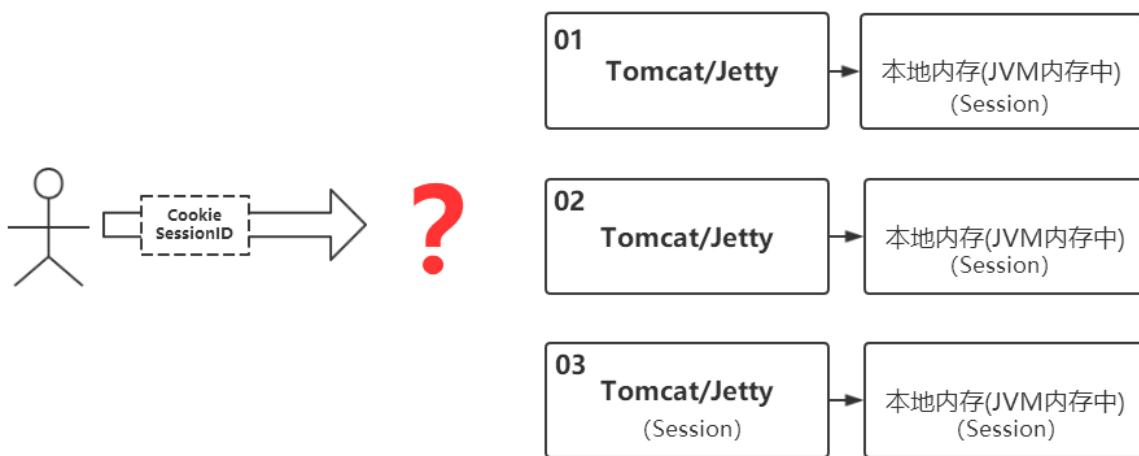
这里有两种会话方式Session与JWT，在项目当中我们选择使用JWT方式作为会话技术，以下是这两种会话方式的区别。

- **Session-Cookie**

这种会话方式我们需要区分为传统单机Session与分布式Session。

在传统单机web应用中，一般使用tomcat/jetty等web容器时，用户的session都是由容器(单JVM内有效)管理。浏览器使用cookie中记录sessionId，容器根据sessionId判断用户是否存在会话session。分布式与集群场景下，请求将被负载发送至不同服务器，Session必需要被共享方可满足需求。

传统Session存储架构：



分布式Session登场

会话信息存储在服务器端Session对象当中，这种方式在分布式会话场景下，相对比较繁琐一点，一般我们用SpringSession做分布式会话，会话信息一般会选择第三方中间件进行存储，比如：MongoDB，Redis，Mysql，这三种存储方式分布式会话框架SpringSession都支持，已经提供springboot的集成，我们只需要做相关配置即可。

比如我们一Redis作为分布式Session存储服务

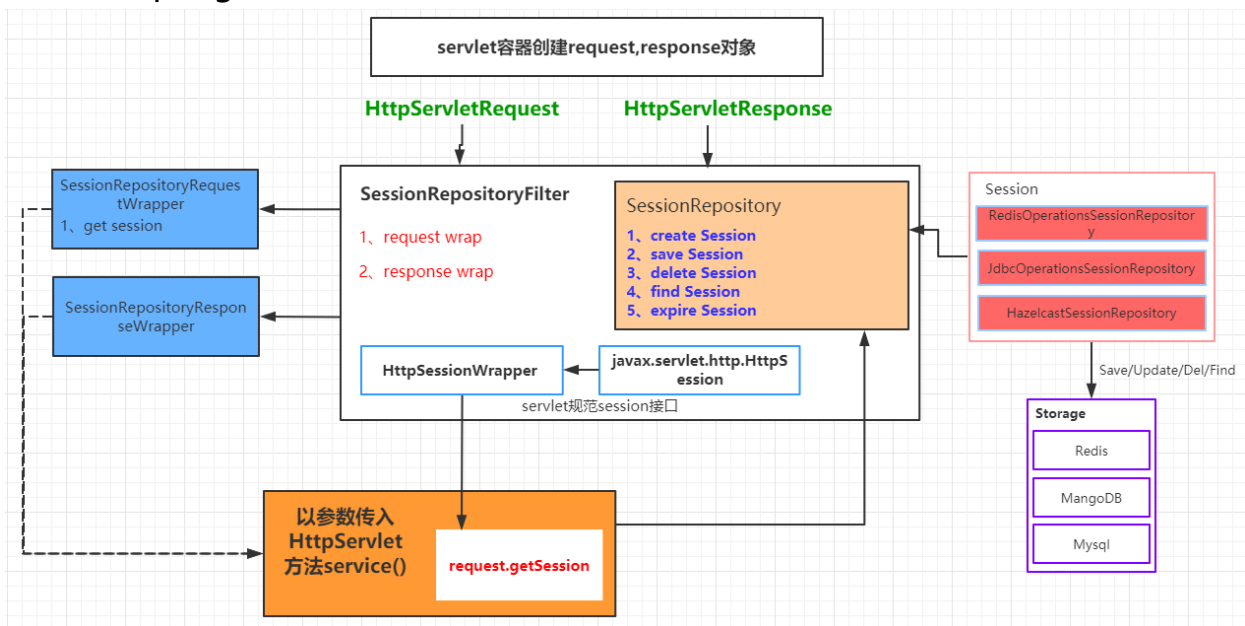
第一步：引入依赖

```
<!--sessions 依赖-->
<dependency>
<groupId>org.springframework.session</groupId>
<artifactId>spring-session-data-redis</artifactId>
<version>2.0.6.RELEASE</version>
</dependency>
<!--redis 依赖-->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-redis</artifactId>
<version>2.0.6.RELEASE</version>
</dependency>
```

```
store-type: redis
```

分布式会话搞定！就这么简单

SpringSession原理如下图:



- **JWT**

全称JSON Web Token，用户会话信息存储在客户端浏览器，它定义了一种紧凑且自包含的方式，用于在各方之间以JSON对象进行安全传输信息。这些信息可以通过对称/非对称方式进行签名，防止信息被篡改。。由此可知JWT是：

- 1、是JSON格式数据
- 2、是一个Token，也就是一个令牌方式

JWT数据包包含三部分:

- 1、Header
- 2、Payload
- 3、Signature

三者组合在一起

Header.Payload.Signature

示例：

eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ0ZXN0IiwiaWY3JlYXRlZCI6MTU3NzE3NjE0OTQ3OCwiZXhwIjojNTc3Nzg0OTQ5fQ.qSlhJNpom2XeeqMyXST2AdHvAjztWqR4zvQQEc-K8qMsJ3XQpwpQ

snG7tK06YoYrjcnH5NW2EGjtemIc_00VIw

Header

描述JWT元数据

{

```
alg: "HS256",
```

```
typ: "JWT"
```

1、alg属性表示签名的算法，默认算法为HS256，可以自行别的算法。

2、**typ**属性表示这个令牌的类型，JWT令牌就为JWT。

Header = Base64(上方json数据)

Payload

使用方式如下, `userid`, `created`(token创建时间), `exp`(最近更新时间)

```
data = {"userid": "yangguo", "created": 1489079981393, "exp": 1489684781}
```

```
Payload = Base64(data) //可以被反编码，所以不要放入敏感信息
```

Signature

HMACSHA256(

```
base64UrlEncode(header) + "." + base64UrlEncode(payload),
```

```
secret)
```

secret为加密的密钥，密钥存在服务端

JWT工作方式

通常我们把token设置在request-Header头中，每次请求前都在请求头加上下方配置

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

示例: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ0ZXN0IiwiaWF0IjY3JlYXR1ZCI6MTU3NzE3NjE0OTQ3OCwiZXhwIjoxNTc3NzgWOTQ5fQ.qSlhJNpom2XeeqMyXST2AdHvAjztWqR4zvQQEc-K8qMsJ3XQpwpQsnG7tK06YoYrjcnH5NW2EGjtemIc_00VIw

JWT身份认证流程

1、用户提供用户名和密码登录

2、服务器校验用户是否正确，如正确，就返回token给客户端，此token可以包含用户信息

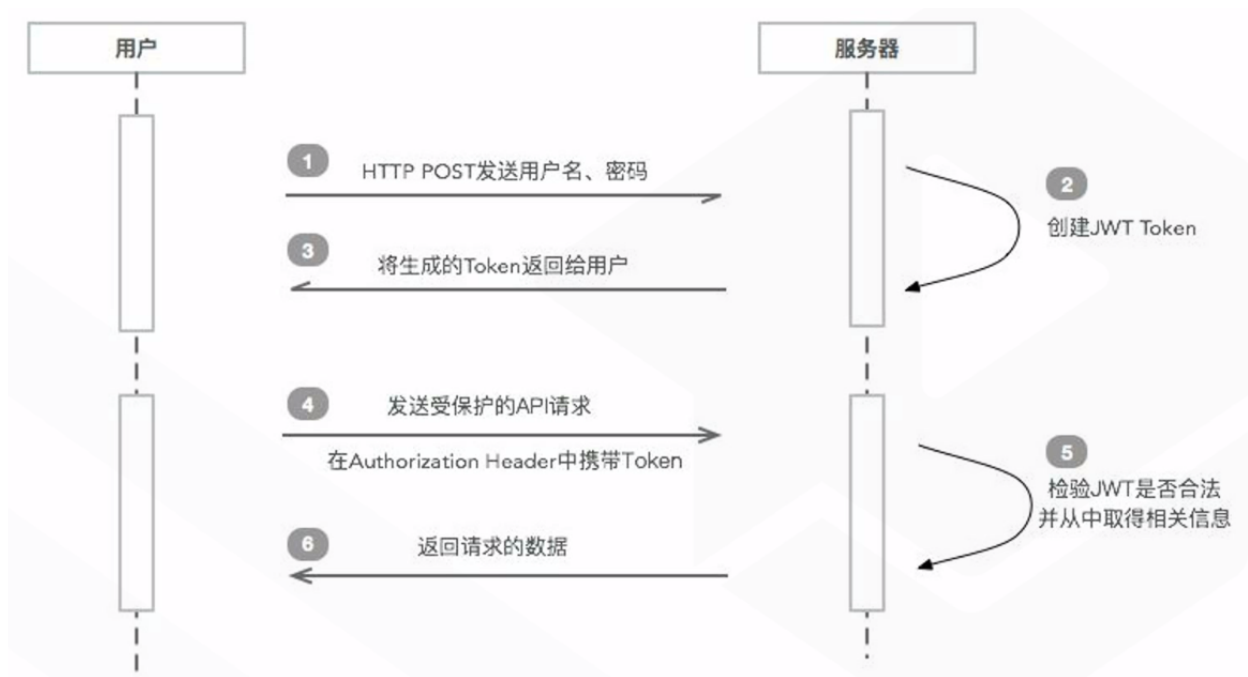
3、客户端存储token，可以保存在cookie或者local storage

4、客户端以后请求时，都要带上这个token，一般放在请求头中

5、服务器判断是否存在token，并且解码后就可以知道是哪个用户

6、服务器这样就可以返回该用户的相关信息了

流程图如下：



jwt与session会话的对比

jwt: 用户信息存在客户端(storage,cookie),

jwt-泄露后? 密码丢失后, 修改密码! 修改之后

jwt-每次传递token;

Session:

本地: jessionid-服务端: redis里的用户信息

有状态: