**Spring boot**

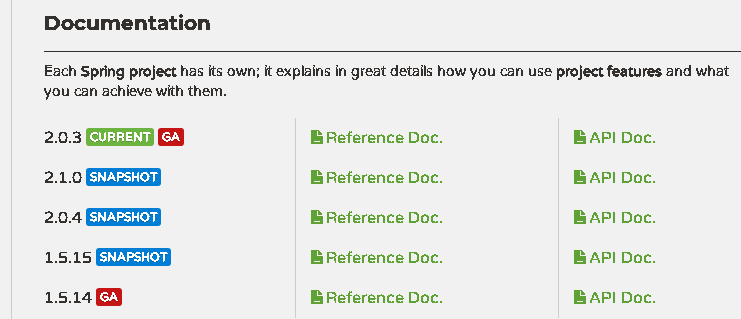
# Spring工程

<https://spring.io/projects>

# 版本选择

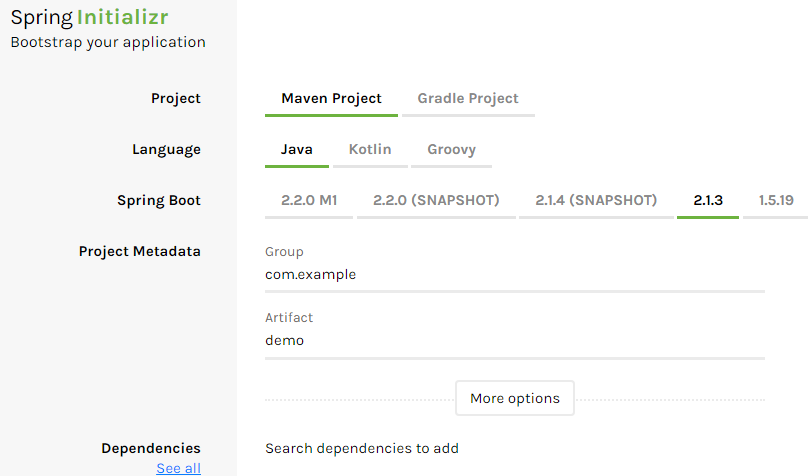
https://spring.io/projects/spring-boot

<https://spring.io/projects/spring-boot#learn>



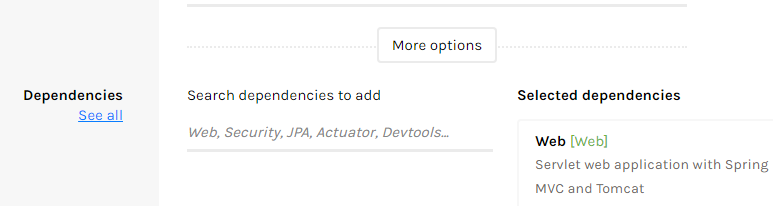
# 试用

<https://start.spring.io/>





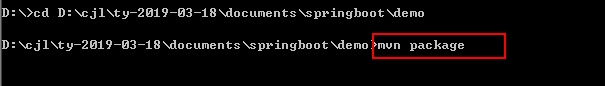
再搞个web项目看看



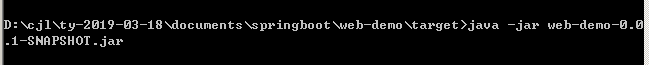
Maven配置

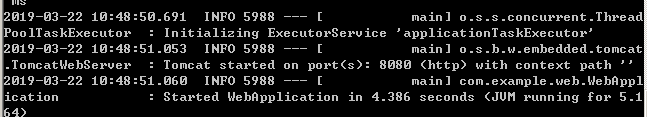
|  |
| --- |
| <localRepository>D:\cjl\mvn-repository</localRepository>  <mirror>  <id>alimaven</id>  <name>aliyun maven</name>  　　<url>http://maven.aliyun.com/nexus/content/groups/public/</url>  <mirrorOf>central</mirrorOf>  </mirror> |

下载到本地后，解压，然后打包



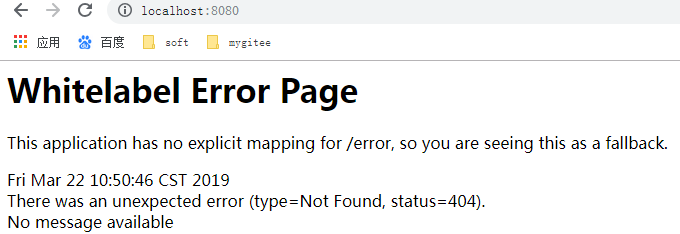
再运行项目，默认会启动http服务，端口号是8080！！





可以查看代码，发现只有一句代码！！

但却实现了，一个web项目！！！



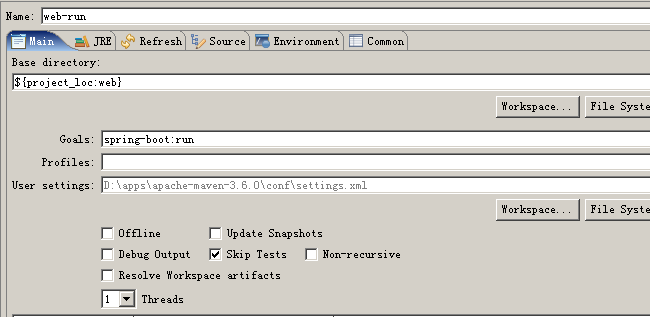
# 入门

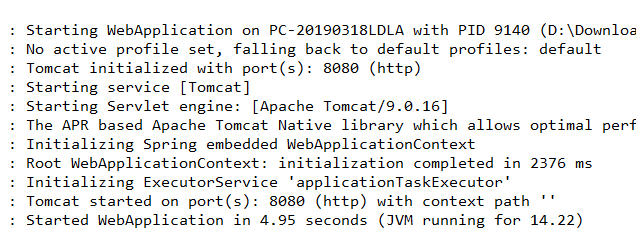
官网帮助文档：

<https://docs.spring.io/spring-boot/docs/1.5.14.RELEASE/reference/htmlsingle/#getting-started>

## 集成环境启动

spring-boot:run

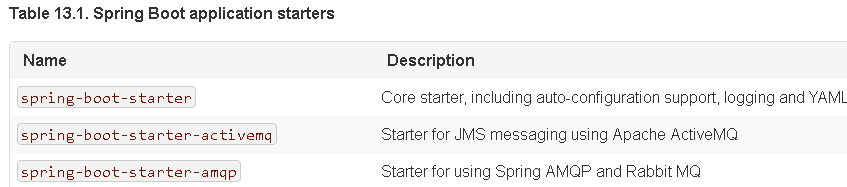




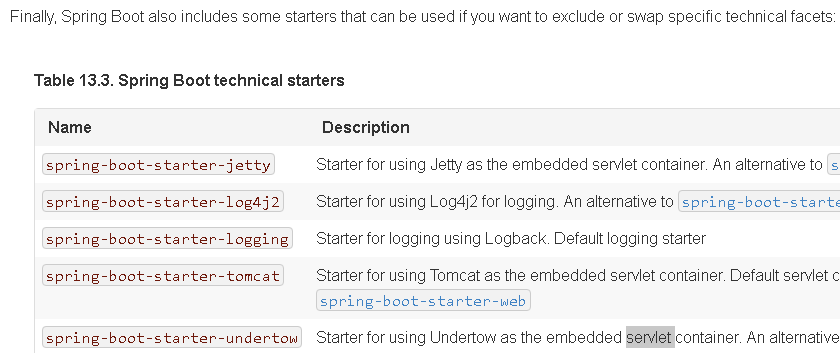
## Starters

一种类型项目所有依赖的集合，方便你快速开始开发，不需要自己再去配置依赖关系！！

All official starters follow a similar naming pattern; spring-boot-starter-\*, where \* is a particular type of application.





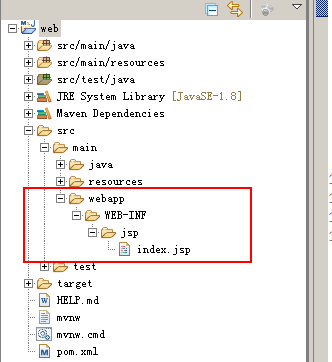


## 改造项目

添加jsp依赖

|  |
| --- |
| <!-- servlet 依赖. -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>  <scope>provided</scope>  </dependency>  <!-- tomcat 的支持. -->  <dependency>  <groupId>org.apache.tomcat.embed</groupId>  <artifactId>tomcat-embed-jasper</artifactId>  <scope>provided</scope>  </dependency> |

添加页面index.jsp



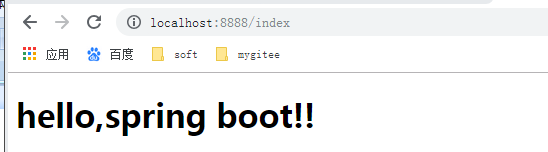
添加控制器

|  |
| --- |
| **package** com.example.web;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  @Controller  **public** **class** JspController {  @RequestMapping("/index")  **public** String index() {  **return** "index";  }  } |

添加jsp解析器配置application.properties：

|  |
| --- |
| server.port=8888  spring.mvc.view.prefix=/WEB-INF/jsp/  spring.mvc.view.suffix=.jsp |

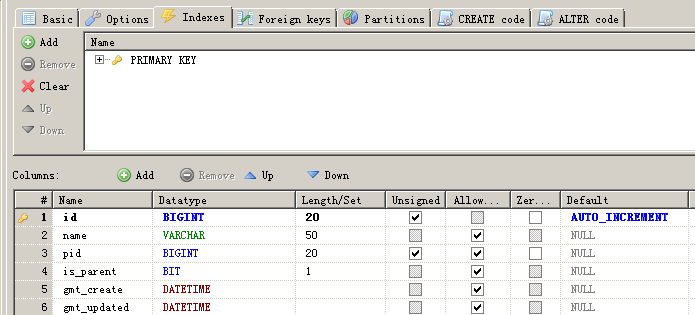
运行：



# 实战练习

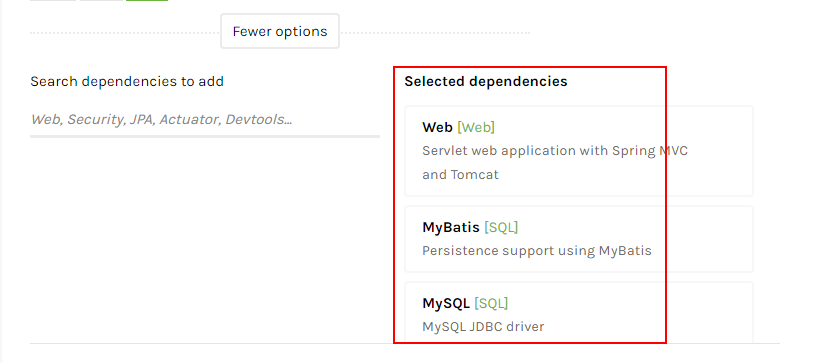
## 商品分类

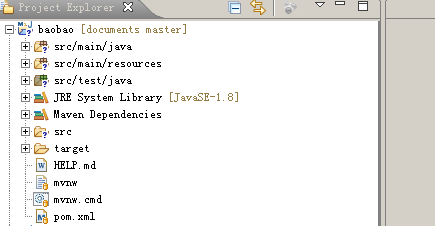
说明：商品类目是基础，实际分为后端类目、前端类目，作用不同，这里只是**后端**的类目表。



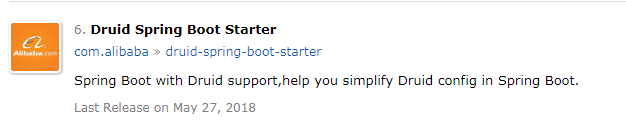
**前端类目表略**，都是根据后端类目编辑生成的，比较灵活，各平台差异大。知道即可。

## 建立新工程





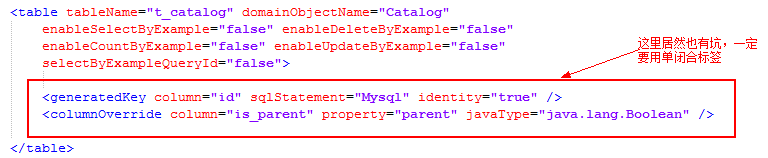
## 集成dao



|  |
| --- |
| <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid-spring-boot-starter</artifactId>  <version>1.1.10</version>  </dependency> |

反向生成dao代码，见提供的资源，解压缩后修改配置文件，再启动运行即可。

关键配置截图：

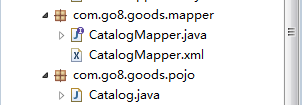


详细配置参考官网：<http://www.mybatis.org/generator/configreference/xmlconfig.html>

执行界面



生成结果

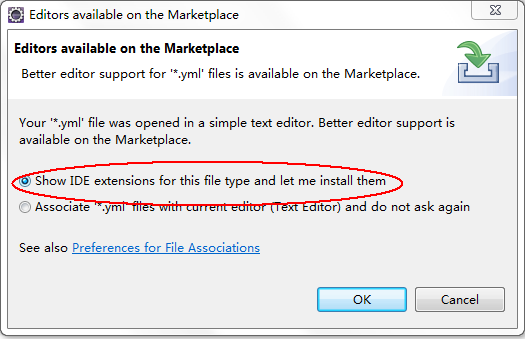


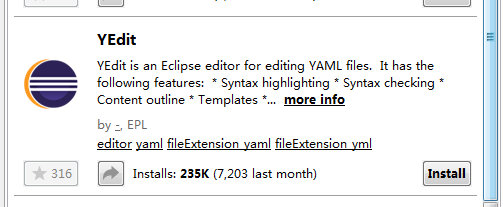
## boot配置

有2种配置格式，流行的格式：yml格式。

编辑**application.yml**文件

注意，第一次创建yml文件需要安装插件，eclipse如下图





安装下一步即可！！！

目前配置如下

|  |
| --- |
| spring:  datasource:  type: com.alibaba.druid.pool.DruidDataSource  url: jdbc:mysql://localhost:3306/go8db?characterEncoding=utf-8&autoReconnect=true&failOverReadOnly=false  username: root  password: 111111  driver-class-name: com.mysql.cj.jdbc.Driver  #druid 配置，线上千万别这么配置，使用默认值就好，否则不能自动重连接  druid:  connectionErrorRetryAttempts: 0  breakAfterAcquireFailure: **true** |

同时创建 boot启动类**GoodsApplication.java**

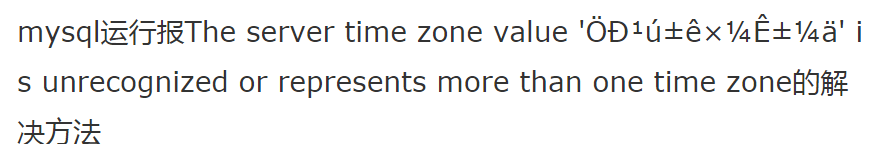
|  |
| --- |
| package com.go8.goods;  import org.mybatis.spring.annotation.MapperScan;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  @MapperScan("com.go8.goods.mapper")  public class GoodsApplication {  public static void main(String[] args) {  SpringApplication.run(GoodsApplication.class, args);  }  } |

## dao单元测试

|  |
| --- |
| package com.go8.goods.mapper;  import static org.junit.Assert.\*;  import java.util.Date;  import org.junit.Test;  import org.junit.runner.RunWith;  import org.mybatis.spring.annotation.MapperScan;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.test.context.SpringBootTest;  import org.springframework.test.context.junit4.SpringRunner;  import com.go8.goods.pojo.Catalog;  @RunWith(**SpringRunner**.class)  @SpringBootTest  public class CatalogMapperTest {  @Autowired  private CatalogMapper catalogMapper;    @Test  public void testInsert() {  Catalog cat = new Catalog();  cat.setCode("yifu");  cat.setName("衣服");  cat.setOrder1(1);  cat.setParent(true);  cat.setPid(0L);  Date now = new Date();  cat.setGmtCreate(now);  cat.setGmtModified(now);  int insert = catalogMapper.insert(cat);  assertEquals(1, insert);  }  } |

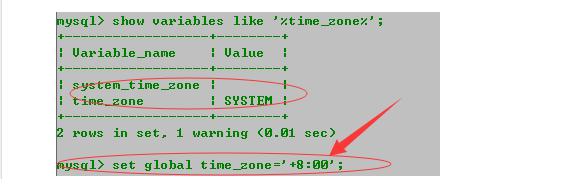
## 可能的问题(8以后)

单元测试插入数据出错，时区的锅（出现在新版本里）



新的版本驱动，包名也改了，会检测数据库的本地时区，默认使用gmt。所以要将数据库时区修改成和系统匹配的真实时区。china的系统都是东八区。

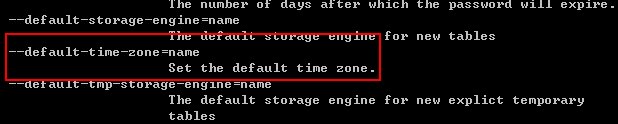
临时修改



解决完时区问题，测试应该会正常通过。

永久修改方法：





启动脚本：

|  |
| --- |
| **mysqld --default-time-zone="+8:00" --console** |

## CatalogService

|  |
| --- |
| package com.go8.goods.service;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import com.go8.goods.mapper.CatalogMapper;  import com.go8.goods.pojo.Catalog;  @Service  public class CatalogServiceImpl implements CatalogService {  @Autowired  private CatalogMapper catalogMapper;    @Override  public void add(Catalog catalog) {  Date now = new Date();  catalog.setGmtCreate(now);  catalog.setGmtModified(now);  catalogMapper.insert(catalog);  }  } |

**依次完成其他查询、修改和删除代码。**

## 发布接口

接口发布的协议，有多种形式，总体分为http和非http的，http形式的多用于对外开放接口，好处不受语言和平台的影响；非http的rpc协议接口，协议种类繁多，如dubbo、thrift等等，多用于内部服务之间的调用，好处速度快点，坏处受平台和语言影响。

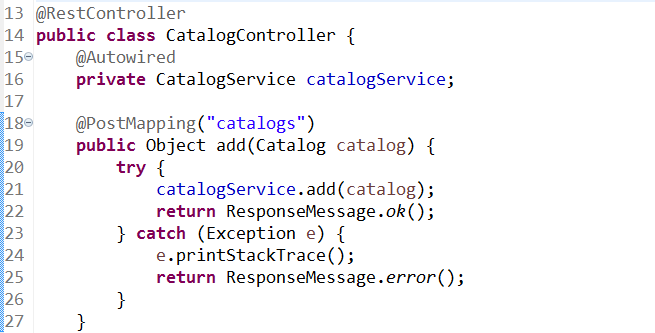
我们目前完成的项目是电商后台管理系统，此系统是一个基于vue的前端项目，语言不同源，因此适合使用http+json的接口协议。

http + json

约定基础接口

|  |  |  |  |
| --- | --- | --- | --- |
| 业务类型 | method | url | return body |
| 添加 | post | catalogs | 返回一个统一的json  对象，  {  “code”: xxx,  “message”:”xxxx”,  “data”:{} or []  } |
| 修改 | put | catalogs/{id} |
| 删除 | delete | catalogs/{id} |
| 查询单体 | get | catalogs/{id} |
| 查询子集 | get | catalogs/sons?pid |
| 查询后代 | get | catalogs/all?pid |
| 唯一性验证 | get | catalogs/code?code |

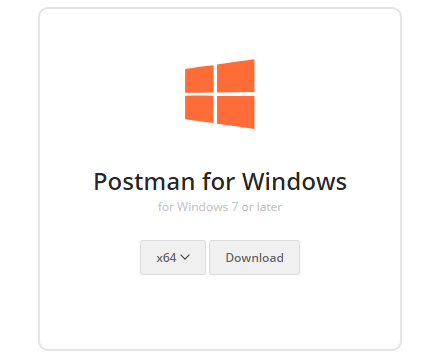
发布代码（部分）



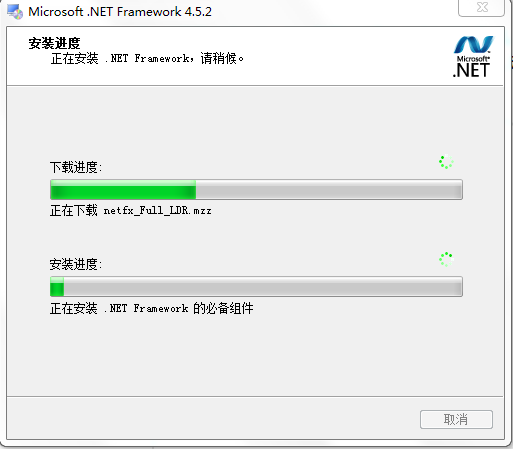
## 接口测试

后台接口开发好以后，我们需要对接口先进行功能性测试，也就是对不对。这里介绍一款专业的api测试工具postman。下载，安装都很简单。注意，这个软件用微软的.net开发的，安装时会下载.net，需要联网，还有耐心。

下载：百度到首页再寻找下载页面，自己下载，也可以去提供的资源包去找。



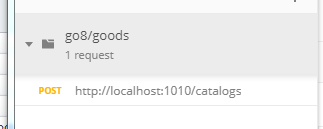
安装时下载.net



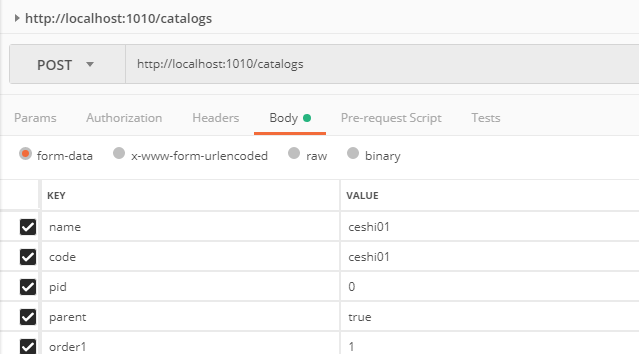
修改端口：

|  |
| --- |
| spring:  datasource:  type: com.alibaba.druid.pool.DruidDataSource  url: jdbc:mysql://localhost:3306/test?characterEncoding=utf-8&autoReconnect=true&failOverReadOnly=false  username: root  password: 111111  driver-class-name: com.mysql.cj.jdbc.Driver  server:  port: 8888 |

新建测试目录，统一管理测试用例

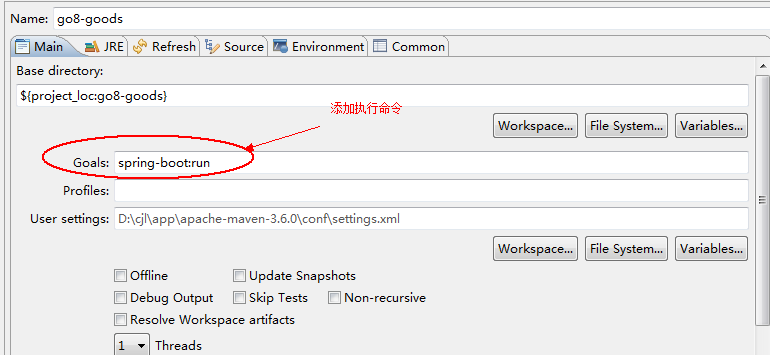


新建一个测试用例，添加一个catalog



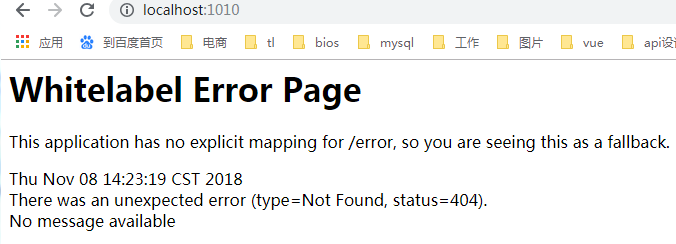
点击send按钮，测试失败，提示找不到服务器。

## 启动服务



## 踩坑记录

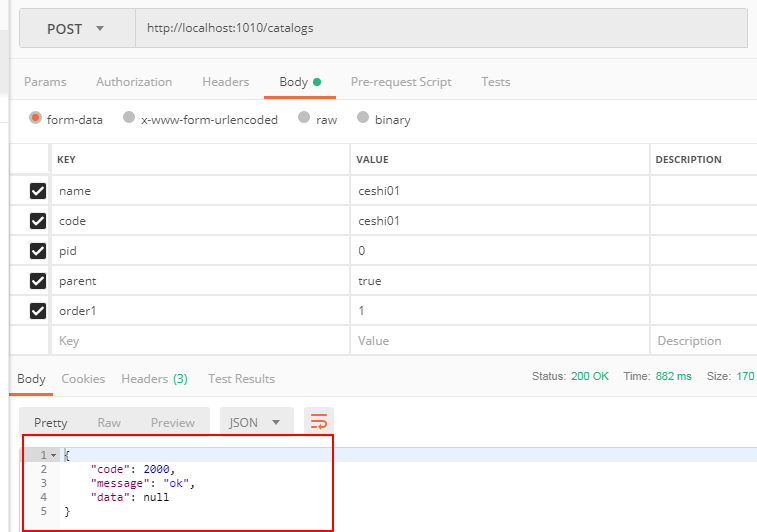
浏览器访问



这里有的同学可能会遇到一个更有意思的事情，就是刚才好不容易测试通过的CatalogMapperTest测试添加一条类目的操作，你再次去测试，会发现一个找不到**CatalogMapper.xml**配置文件的错误，就是提示xxxx is not bound，这种错误啦。这是maven的一个bug啦，而且它不总是会出现。一般会出现在install阶段，现在出现，就是因为你前面执行安装了父工程，安装父工程时会安装子工程，会重新打包，这时基本都会丢失mybatis的配置文件。如果你遇到了，请这样来解决，没遇到的同学，也请添加一下。请在父工程的pom文件里添加资源配置，如下图

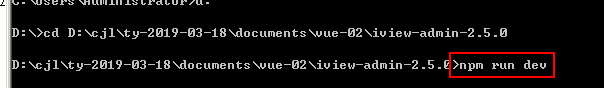


## 再测接口

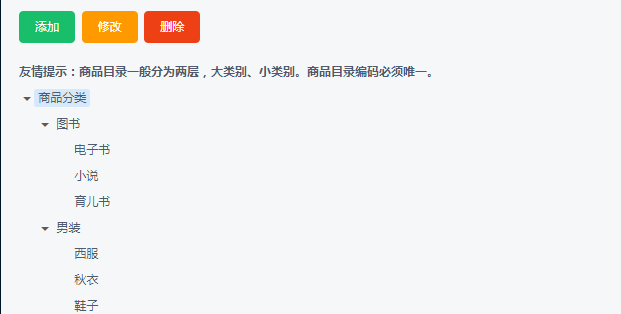


课后有时间的同学最好再测试下异常情况。

# 再回前端



## 类目界面



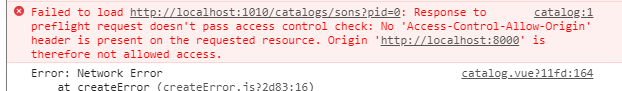
## 代码

|  |
| --- |
| <template>  <div class="alert alert-info" style="margin-bottom: 2px;">  <div style="margin-bottom: 20px;">  <Button type="success" @click="onClick('add')">添加</Button>&nbsp;  <Button type="warning" @click="onClick('update')">修改</Button>&nbsp;  <Button type="error" @click="onClick('delete')">删除</Button>  </div>  <h5>友情提示：商品目录一般分为两层，大类别、小类别。商品目录编码必须唯一。</h5>  <Divider />  <Tree ref="catalogTree" :data="data" :load-data="loadData"></Tree>  <!--对话框-->  <Modal ref="catalogModal" v-model="show" :title="modalTitle">  <!--表单-->  <Form ref="formData" :model="formData" :rules="ruleValidate" :label-width="80">  <FormItem prop="id" style="display: none;">  <Input v-model="formData.id"></Input>  </FormItem>  <FormItem prop="pid" style="display: none;">  <Input v-model="formData.pid"></Input>  </FormItem>  <FormItem label="父类" prop="pname">  <Input v-model="formData.pname" readonly="readonly" disabled></Input>  </FormItem>  <FormItem label="子类名称" prop="name">  <Input v-model="formData.name" @on-blur="getCode" placeholder="请输入名称"></Input>  </FormItem>  <FormItem label="编码" prop="code">  <Input v-model="formData.code" placeholder="请输入编码"></Input>  </FormItem>  <FormItem label="顺序" prop="order1">  <InputNumber :max="100" :min="1" v-model="formData.order1"></InputNumber>  </FormItem>  <FormItem>  <Button type="primary" @click="handleSubmit('formData',formType)" style="margin-left: 250px">确 定</Button>  <Button @click="handleReset('formData')" style="margin-left: 20px">取 消</Button>  </FormItem>  </Form>  </Modal>  </div>  </template>  <script>  import {  getCatalogByPid,  getCatalogById,  checkCatalogCode,  addCatalog,  updateCatalog,  deleteCatalogById  } from '../../api/catalog.js'  import {  ConvertPinyin  } from '../../libs/pinyin.js'  //记录修改时加载的初始编码值  let old\_catalog\_code = '';  export default {  data() {  return {  data: [{  id: 0,  title: '商品分类',  loading: false,  children: []  }],  //对话框是否显示  show: false,  //对话框标题  modalTitle: '',  //表单类型  formType: 'add',  //表单数据  formData: {  id: '',  pid: '',  pname: '',  name: '',  code: '',  order1: 1  },  //表单数据验证  ruleValidate: {  name: [{  required: true,  whitespace: true,  message: '名称不能为空',  trigger: 'blur'  },  {  pattern: /^\S\*$/,  message: '名称不能包括空字符',  trigger: 'blur'  }  ],  code: [{  required: true,  whitespace: true,  message: '编码不能为空',  trigger: 'blur'  },  {  pattern: /^\S\*$/,  message: '编码不能包括空字符',  trigger: 'blur'  },  {  //自定义验证,验证唯一性  validator(rule, value, callback, source, options) {  var errors = [];  //如果修改时，是原来的旧编码，不需要验证唯一性  if (old\_catalog\_code == value) {  callback(errors);  } else {  checkCatalogCode(value).then(res => {  if (res.status.code == 2000) {  if (res.data) {  errors.push('编码已被使用');  }  }  callback(errors);  }).catch(function(error) {  console.log(error);  });  console.log("异步验证code。。。。。。");  }  },  trigger: 'change'  }  ]  }  }  },  methods: {  //相当于刷新吧，目前没找到好方法来刷新tree  init() {  this.data = [{  id: 0,  title: '商品分类',  loading: false,  children: []  }];  },  loadData(item, callback) {  console.log(item);  var data = [];  //根据pid获取商品分类数据  getCatalogByPid(item.id).then(res => {  console.log("res : " + res);  data = res.data;  console.log(data);  //由于后端返回数据，都带有children属性，bug,  //要处理掉parent=false时的children属性和loading属性  for (let i = 0; i < data.length; i++) {  if (!data[i]['parent']) {  console.log(data[i]);  //删除  delete data[i]['children'];  delete data[i]['loading'];  }  }  callback(data);  }).catch(err => {  console.log(err)  });  },  //初始化窗口  initDialog(type) {  //显示  this.show = true;  //初始化对话框，异步刷新dom  this.$nextTick(function() {  //引用技术，方便父组件根据名称访问子组件  console.log(this.$refs.catalogModal);  //隐藏底部对话框  this.$refs["catalogModal"].footerHide = true;  });  if (type == 'add') {  //弹出添加商品分类的界面  this.modalTitle = '添加商品分类';  this.formType = 'add';  } else if (type == 'update') {  this.modalTitle = '修改商品分类';  this.formType = 'update';  }  },  //按钮单击事件回调  onClick(type) {  //判断是否选择了节点  var nodes = this.$refs.catalogTree.getSelectedNodes();  if (nodes.length == 0) {  this.$Message.error('您没有选择节点');  } else {  //判断选择了什么按钮  if (type == 'add') {  //初始化表单数据  this.formData.pid = nodes[0].id;  this.formData.pname = nodes[0].title;  this.formData.code = '';  this.formData.id = '';  this.formData.name = '';  this.formData.order1 = '1';  this.initDialog(type);  } else if (type == 'update') {  let node\_id = nodes[0].id;  //根节点不能修改  if (node\_id == 0) {  this.$Message.error('不能修改根节点!');  return;  }  //根据id获取数据  var cat = {};  //根据id获取商品分类数据  getCatalogById(nodes[0].id).then(res => {  cat = res.data;  this.formData.id = cat.id;  this.formData.pid = cat.pid;  this.formData.pname = cat.pname;  this.formData.code = cat.code;  this.formData.name = cat.name;  this.formData.order1 = cat.order1;  this.initDialog(type);  //初始化旧的编码  old\_catalog\_code = cat.code;  }).catch(err => {  console.log(err)  });  } else if (type == 'delete') {  let node\_id = nodes[0].id;  //根节点不能被删除  if (node\_id == 0) {  this.$Message.error('不能删除根节点!');  } else {  //确认删除  this.$Modal.confirm({  title: '删除提醒',  content: '<p>您确定要删除该【' + nodes[0].title + '】分类吗？</p>',  onOk: () => {  //根据id删除类型  deleteCatalogById(node\_id).then(res => {  if (res.status.code == 2000) {  this.init();  this.$Message.success('删除分类成功!');  } else {  this.$Message.error('删除分类失败!');  }  }).catch(err => {  console.log(err);  });  }  });  }  }  }  },  //生成编码  getCode() {  var code = ConvertPinyin(this.formData.name);  this.formData.code = code;  },  //提交表单  handleSubmit(name) {  this.$refs[name].validate((valid) => {  if (valid) {  if (this.formType == 'add') {  //添加分类  addCatalog({  pid: this.formData.pid,  name: this.formData.name,  order1: this.formData.order1,  code: this.formData.code,  parent:0  })  .then(res => {  if (res.status.code == 2000) {  this.init();  this.$Message.success('添加分类成功!');  } else {  this.$Message.error('添加分类失败!');  }  })  .catch(function(error) {  console.log(error);  });  } else if (this.formType == 'update') {  //修改分类  updateCatalog({  id: this.formData.id,  name: this.formData.name,  order1: this.formData.order1,  code: this.formData.code  })  .then(res => {  if (res.status.code == 2000) {  this.init();  this.$Message.success('修改分类成功!');  } else {  this.$Message.error('修改分类失败!');  }  })  .catch(function(error) {  console.log(error);  });  }  } else {  this.$Message.error('表单数据不合法!');  }  })  },  //重置表单  handleReset(name) {  //this.$refs[name].resetFields();  //全局关闭对话框方法，好像不灵哎！  //this.$Modal.remove();  this.show = false;  }  },  mounted: function() {  //this.init();  }  }  </script> |

# 联合测试

接口写好后，启动go8-goods项目，服务的地址：**locahost:1010**

这时前端按道理就能请求到后台数据了，但是很不幸，依然是同样的错误界面啦



错误原因很明显：“**Access-Control-Allow-Origin**”，这个就是不同源访问错误，业内叫做跨域请求错误。

# 跨域请求

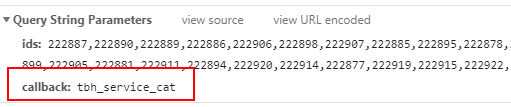
何为跨域？想简单点说的话，就是先解释什么是同域，何为同域？就是协议、域名、端口都相同。其他的都是跨域。解决跨域最简单的方法就是想办法让他们同域，这个是废话，等于没说，C:\Users\ADMINI~1\AppData\Local\Temp\SGPicFaceTpBq\6264\BEB73DD4.pngC:\Users\ADMINI~1\AppData\Local\Temp\SGPicFaceTpBq\6264\BEB7A280.png。我只是想活跃下气氛而已，C:\Users\ADMINI~1\AppData\Local\Temp\SGPicFaceTpBq\7436\BEC2CEDD.png。常用的方法有很多：JSONP，代理，CORS，XDR等等。这里大家只需要知道JSONP，掌握CORS就好。

## JSONP

以前的w3c旧标准下一种“钻漏洞”的解决方法，甚至有人都不认为它是一种解决方案，哎，随便啦，爱怎么说咋说吧。具体原理了解一下：**浏览器对script的资源引用没有同源限制**，那么JSONP就是在前端向后台请求一个js方法调用代码，然后把真正想要的数据作为参数传给这个方法，返回给客户端，这时前后端通常约定一个参数**callback**，该参数的值，就是处理返回数据的函数名称。怎么样，以前的程序员imba吧。基本上以前都是这么干的，包括淘宝、京东这样的大厂。有图为证：

淘宝首页分类加载的api：data.htm的请求参数

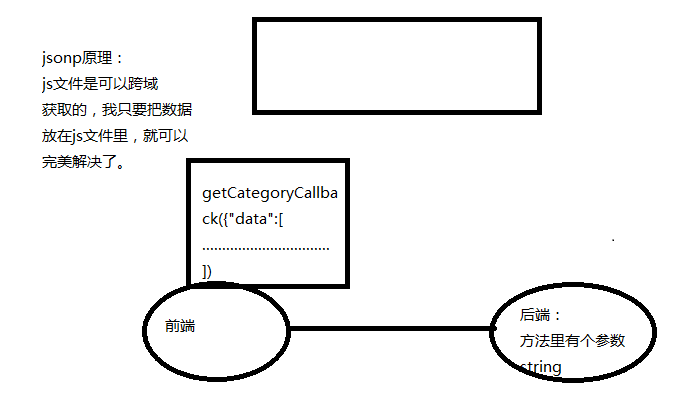




再看看响应体的数据：



有兴趣的同学再去看看京东，一样的套路。



## CORS

同源策略（same origin policy）是浏览器安全的基石。在同源策略的限制下，非同源的网站之间不能发送 ajax 请求的。为了解决这个问题，w3c 提出了跨源资源共享，即 CORS(Cross-Origin Resource Sharing)。CORS 做到了两点：

1）不破坏即有规则

2）服务器实现了 CORS 接口，就可以跨源通信

由于这个规则是新出的，所以以前的项目就见不到啦，不过现在新的项目一般都用这个方法，**spring也是从4.0**以后才提供这样的支持的。使用方法很简单，主要有3种场景。

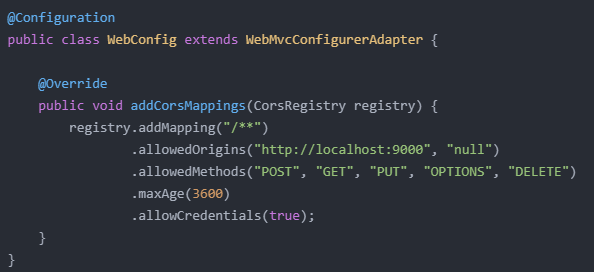
第一种情况，如果想要对某一接口配置 CORS，可以在方法上添加 CrossOrigin 注解：

@CrossOrigin(origins = {"http://localhost:9000", "null"})

第二种情况，如果想对一系列接口添加 CORS 配置，可以在类上添加注解，对该类声明所有接口都有效：

@CrossOrigin(origins = {"http://localhost:9000", "null"})

第三种情况，添加全局配置，则需要添加一个配置类：



**重点来了：前方高能，坑来了！！！！**

以上使用**WebMvcConfigurerAdapter**方法在spring 4.0版本下都没有问题，在**spring5.0下**提示过时，查看官网提示有2种方法：

1）直接实现WebMvcConfigurer接口

2）继承WebMvcConfigurationSupport

|  |
| --- |
| **Deprecated.**  **as of 5.0 WebMvcConfigurer has default methods (made possible by a Java 8 baseline) and can be implemented directly without the need for this adapter**  **An implementation of WebMvcConfigurer with empty methods allowing subclasses to override only the methods they’re interested in.** |

那么问题又来了，我使用了第二种方法，然后并没有卵用，好吧，网上有人遇到相同问题，说要用第一种方法就行。好吧，我照做了，还是然并卵。其实这两种方法并没有本质的区别，因为WebMvcConfigurationSupport类就是实现的WebMvcConfigurer接口，本质上是一个东西，怎么会出现一个行，一个不行呢？所以我2个都不行，才是合理解释啊，哈哈！！ 那怎么办呢？老程序员可能会跟你说，小伙子，回家睡一觉就好了，嘿嘿，有时候这方法还真挺不错的。可是我现在不想睡觉啊，于是抱着试试看的心态，尝试了另一种方法，终于，终于 OJBK啦，C:\Users\ADMINI~1\AppData\Local\Temp\SGPicFaceTpBq\9588\BEF67412.pngC:\Users\ADMINI~1\AppData\Local\Temp\SGPicFaceTpBq\9588\BEF682E0.png。就是直接操控**CorsConfiguration类，其他的方法最终都是变相配置该类的。**

|  |
| --- |
| /\*\*  \* CORS configuration  \*/  @Configuration  **public** **class** CORSConfig {  **private** CorsConfiguration buildConfig() {  CorsConfiguration corsConfiguration = **new** CorsConfiguration();  corsConfiguration.addAllowedOrigin("\*"); // 1允许任何域名使用  corsConfiguration.addAllowedHeader("\*"); // 2允许任何头  corsConfiguration.addAllowedMethod("\*"); // 3允许任何方法（post、get等）  **return** corsConfiguration;  }  @Bean  **public** CorsFilter corsFilter() {  UrlBasedCorsConfigurationSource source = **new** UrlBasedCorsConfigurationSource();  source.registerCorsConfiguration("/\*\*", buildConfig()); // 4  **return** **new** CorsFilter(source);  }  } |

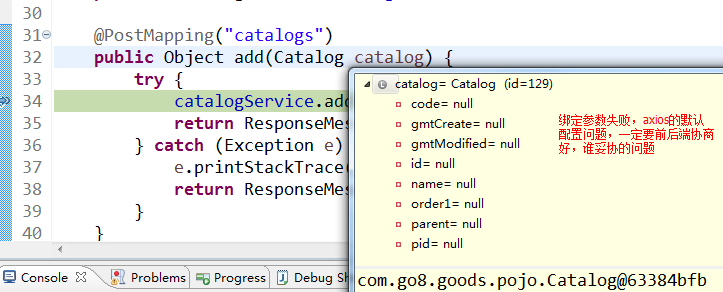
## 总结

在生产中所有的jar包应该尽量使用自己熟悉的，已经被大多数人证明过的稳定的版本，不到万不得已，千万别用最新的版本。新版本意味着未知的bug，而且由于版本太新，你很可能在网上找不到解决办法，这时你只有去啃源码了，这个方法需要耐心、时间，这个在追求项目进度的公司里，是不可行的。还是那句话：一代版本坑一代。

**我们这里使用新版本，是因为在学习的过程中，多踩坑利于大家的成长，同时还能给同学们留下印象，血的经验教训，对大家以后的工作会很有帮助的。**

# 第二次测试

我们通过添加配置类**CORSConfig**的方法，再次来测试，会发现不再出现跨域请求的错误了。但是我们前端无法添加商品类目数据，通过打断点调试发现，如下图：



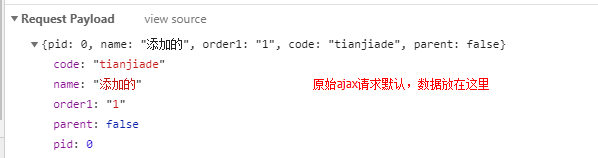
显然是参数绑定失败了，要想解决这个问题，就得好好说道说道这个前端**axios**了。

## Axios

Axios你可以简单点，就把它理解成是一个ajax框架。大家可以把它与前面学的jQuery里的$.ajax进行类比。不过在发送ajax请求时，它会默认序列化你的json对象。官网原话：**By default, axios serializes JavaScript objects to JSON. To send data in the application/x-www-form-urlencoded format instead, you can use one of the following options.**

**也就是我们现在要添加的数据被放在了请求体里，而且头还指明了是一个json类型，如下图：**

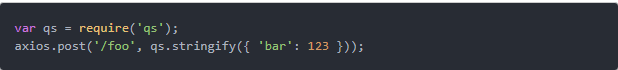




所以后台方法默认接受参数的方法就获取不到数据了，默认是getParameter()方法，还记得前面servlet的内容吗？？ Springmvc用一个注解**@RequestBody**轻松解决这个问题。这是修改后台代码的方式解决，如果后台不妥协呢？？那么第二种方法就是前端开发人员需要修改代码了，如下图：



具体的axios就是利用一些前端工具库如qs，把json对象转换成字符串格式，例如：



# 第三次测试



保证各个功能测试通过。最后别忘了，提交你的代码，谢谢大家！！

# 问题

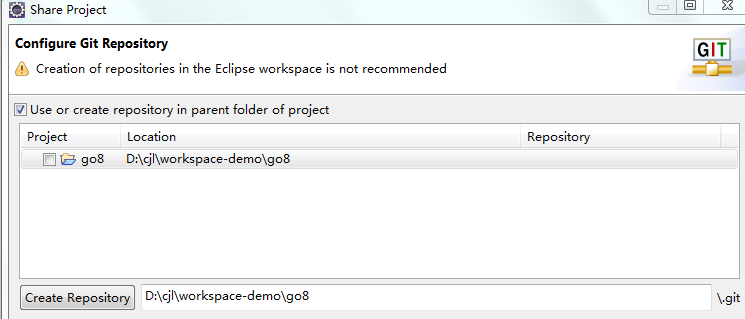
如何备份，分享你的代码？？？？

================================学完git后再回来================================

# 分享项目

以IDE：eclipse的egit为例子，说明分享流程：

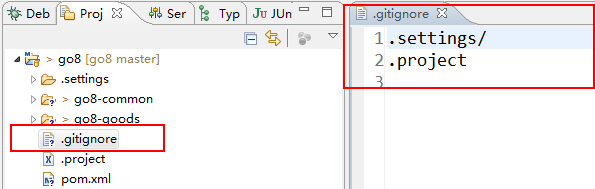
1）在项目的右键菜单中找到 team -🡪 share project



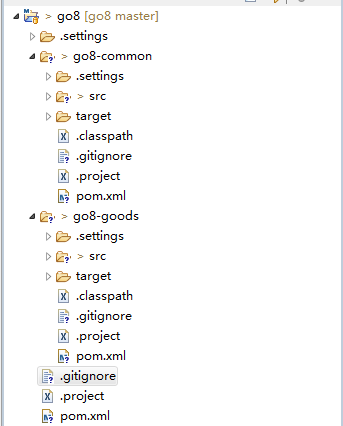
2）添加.gitignore文件

小技巧，不需要自己去新建，右键选择要忽略的文件添加即可自动创建此文件

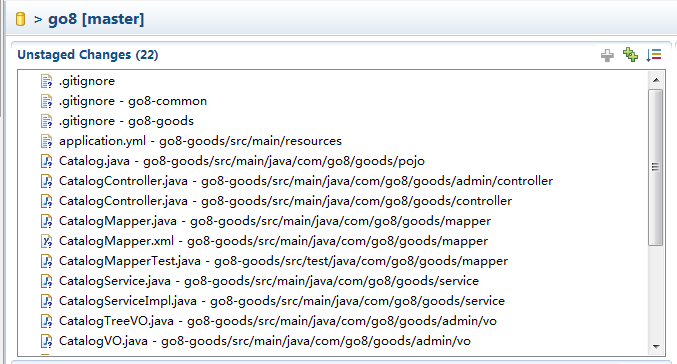
该文件按目录存储，各个子工程目录下也需要添加。



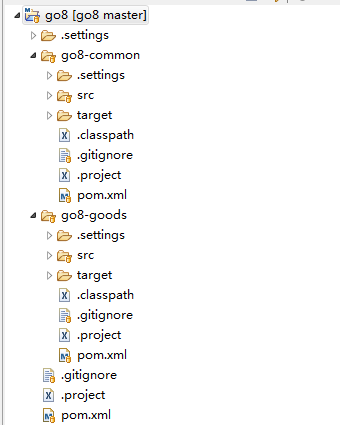
3）未提交状态，如下图



4）提交项目到本地版本库



5）提交后状态

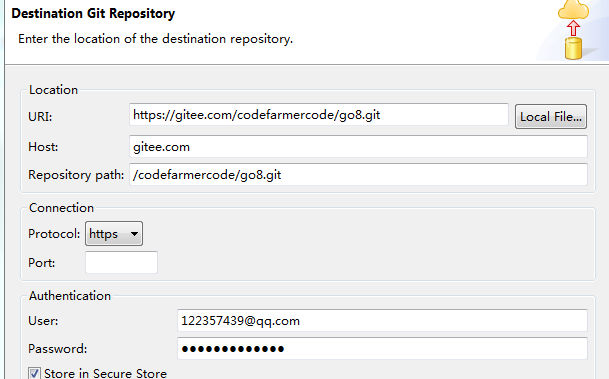


6）建立远程仓库

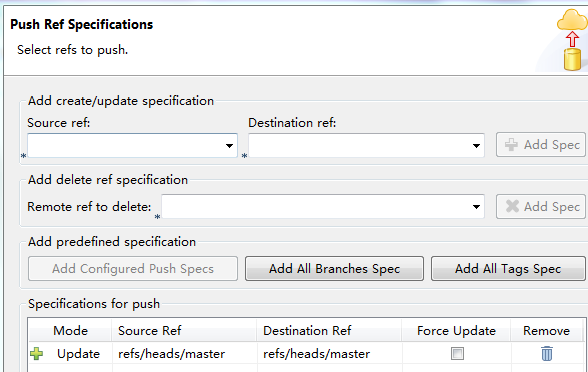
各位同学，自由选择，常用的有github，gitee，大厂都有免费使用的版本库，这里以gitee为例子，建好后记得复制远程仓库地址。



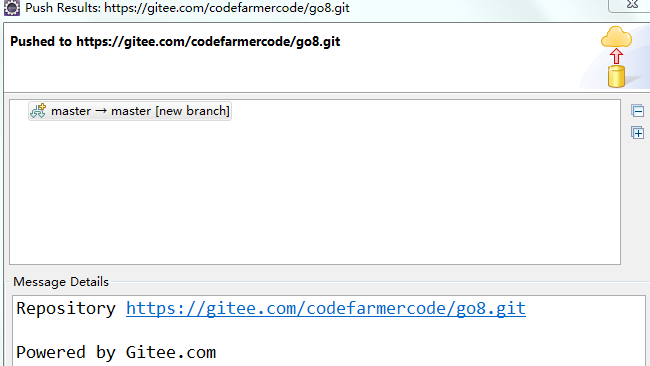
7）推送远程仓库



8）设置推送的分支



9）推送成功



10）查看远程仓库

