SpringBoot笔记

# 简介：

## SpringBoot介绍

Spring Boot使开发独立的，产品级别的基于Spring的应用变得非常简单，你只需"just run"。 我们为Spring平台及第三方库提供开箱即用的设置，这样你就可以有条不紊地开始。多数Spring Boot应用需要很少的Spring配置。

你可以使用Spring Boot创建Java应用，并使用java -jar启动它或采用传统的war部署方式。

### 解决的问题

* 依赖太多了, 且存在版本问题
* 配置太多了且每次都一样, 大部分工程, 配置每次都是一样的, 从一个地方拷贝到另外一个地方. 且Spring发展10多年, 各种配置版本太多, 对于很多程序员来说, 分不清哪个是有效, 哪个无效.
* 部署太麻烦. 需要tomcat部署, 项目结构也需要照着[Java EE](http://lib.csdn.net/base/17" \o "Java EE知识库" \t "http://blog.csdn.net/haoyifen/article/details/_blank)的目录结构来写.

### SpringBoot特点

* 创建独立的Spring应用程序
* 嵌入的Tomcat，无需部署WAR文件
* 简化Maven配置
* 自动配置Spring
* 提供生产就绪型功能，如指标，健康检查和外部配置
* 绝对没有代码生成和对XML没有要求配置

### SpringBoot功能

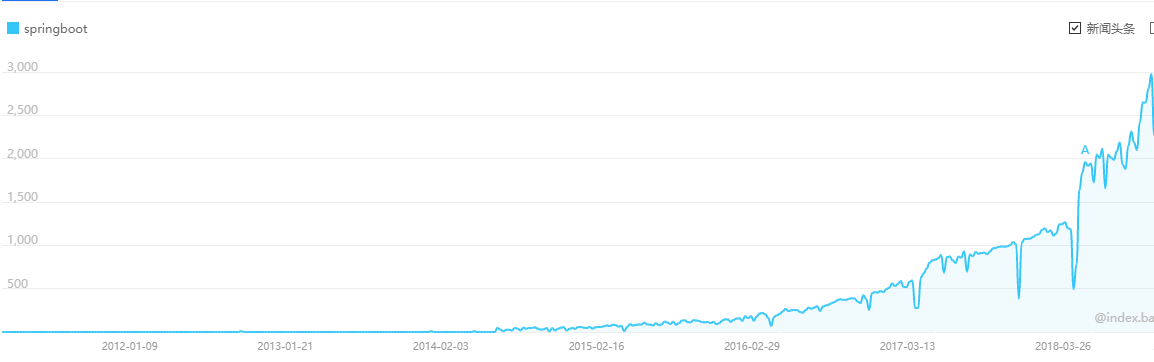
* **自动配置(auto-configuration)**

一项简化配置的功能，比如在classpath中发现有spring security的jar包，则自动创建相关的bean等

* **starters(简化依赖)**

这个比较关键，方便spring去集成各类组件，比如redis、mongodb等等。

### SpringBoot的发展



## 系统要求

默认情况下，本堂课使用SpringBoot 2.1.2最新版本，最好安装JDK8以及以上的版本，maven使用3.3或者以上的版本（本教程使用maven3.6版本）

## 第一个SpringBoot项目

新建一个普通maven项目

创建pom文件

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>enjoy</groupId>

<artifactId>springbootvip</artifactId>

<version>1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

</project>

编写代码：

为了完成应用程序，我们需要创建一个单独的Java文件。Maven默认会编译src/main/java下的源码

新建：cn.enjoy.Example

package cn.enjoy;

import org.springframework.boot.\*;

import org.springframework.boot.autoconfigure.\*;

import org.springframework.stereotype.\*;

import org.springframework.web.bind.annotation.\*;

@RestController

@EnableAutoConfiguration

public class Example {

@RequestMapping("/")

String home() {

return "Hello World!";

}

public static void main(String[] args) throws Exception {

SpringApplication.run(Example.class, args);

}

}

@RestController和@RequestMapping说明：

@RestController。这被称为一个构造型（stereotype）注解。它为阅读代码的人们提供建议。对于Spring，该类扮演了一个特殊角色。在本示例中，我们的类是一个web @Controller，所以当处理进来的web请求时，Spring会询问它。

@RequestMapping注解提供路由信息。它告诉Spring任何来自"/"路径的HTTP请求都应该被映射到home方法。@RestController注解告诉Spring以字符串的形式渲染结果，并直接返回给调用者。

@EnableAutoConfiguration。这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于spring-boot-starter-web添加了Tomcat和Spring MVC，所以auto-configuration将假定你正在开发一个web应用并相应地对Spring进行设置。

main方法。这只是一个标准的方法，它遵循Java对于一个应用程序入口点的约定。我们的main方法通过调用run，将业务委托给了Spring Boot的SpringApplication类。SpringApplication将引导我们的应用，启动Spring，相应地启动被自动配置的Tomcat web服务器。我们需要将Example.class作为参数传递给run方法来告诉SpringApplication谁是主要的Spring组件。

执行main方法，使用一个浏览器打开 **localhost:8080**，以下输出：

Hello World!

## 注意事项

Spring Boot不需要使用任何特殊的代码结构，然而，这里有一些地方需要注意

**使用"default"包**

当类没有包含package声明时，它被认为处于default package下。通常不推荐使用default package，并应该避免使用它。因为对于使用@ComponentScan，@EntityScan或@SpringBootApplication注解的Spring Boot应用来说，来自每个jar的类都会被读取，这会造成一定的问题。

**定位main应用类**

通常建议你将main应用类放在位于其他类上面的根包（root package）中。通常使用@EnableAutoConfiguration注解你的main类，并且暗地里为某些项定义了一个基础“search package”。例如，如果你正在编写一个JPA应用，被@EnableAutoConfiguration注解的类所在包将被用来搜索@Entity项。

使用根包允许你使用@ComponentScan注解而不需要定义一个basePackage属性。如果main类位于根包中，你也可以使用@SpringBootApplication注解。

下面是一个典型的结构：

cn

+- enjoy

+- myproject

+- Application.java

|

+- domain

| +- Customer.java

| +- CustomerRepository.java

|

+- service

| +- CustomerService.java

|

+- web

+- CustomerController.java

# SpringBoot快速入门

可以继承spring-boot-starter-parent项目来获取合适的默认设置。

想配置你的项目继承spring-boot-starter-parent只需要简单地设置parent为：

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

</parent>

注：你应该只需要在该依赖上指定Spring Boot版本。如果导入其他的starters，你可以放心的省略版本号。

**使用没有父POM的Spring Boot**

不是每个人都喜欢继承spring-boot-starter-parent POM。你可能需要使用公司标准parent，或你可能倾向于显式声明所有Maven配置。

如果你不使用spring-boot-starter-parent，通过使用一个scope=import的依赖，你仍能获取到依赖管理的好处：

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>2.1.2.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

**登陆\注册**

## 建表

**CREATE TABLE `enjoy\_user` (**

**`id` int NOT NULL AUTO\_INCREMENT ,**

**`passwd` varchar(255) NULL ,**

**`username` varchar(255) NULL ,**

**PRIMARY KEY (`id`)**

**);**

## 搭建springboot环境

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

新建App.java

@SpringBootApplication

public class App {

public static void main(String[] args) throws Exception {

SpringApplication.run(App.class, args);

}

}

## 新建Controller

package cn.enjoy.controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class UserController {

@RequestMapping("/hello")

public Object sayHello() {

return "hello";

}

}

运行App，在浏览器输入：localhost:8080/hello，发现“hello”说明第一步部署成功。

## 集成mybatis

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>1.2.0</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

## 生成mapper

在resources目录，新建application.properties文件，增加内容如下

spring.datasource.driverClassName=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/spring?serverTimezone=GMT%2B8

spring.datasource.username=root

spring.datasource.password=root1234%

mybatis.mapperLocations=classpath:mapping/\*.xml

准备mybatis的生成文件generatorConfig.xml，并在相应目录创建好model,dao,mapping文件夹

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE generatorConfiguration

PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"

"http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">

<generatorConfiguration>

<!-- 数据库驱动:选择你的本地硬盘上面的数据库驱动包-->

<classPathEntry location="C:\Users\VULCAN\.m2\repository\mysql\mysql-connector-java\5.1.37\mysql-connector-java-5.1.37.jar"/>

<context id="DB2Tables" targetRuntime="MyBatis3">

<commentGenerator>

<property name="suppressDate" value="true"/>

<!-- 是否去除自动生成的注释 true：是 ： false:否 -->

<property name="suppressAllComments" value="true"/>

</commentGenerator>

<!--数据库链接URL，用户名、密码 -->

<jdbcConnection driverClass="com.mysql.jdbc.Driver" connectionURL="jdbc:mysql://127.0.0.1:3306/spring" userId="root" password="root1234%">

</jdbcConnection>

<javaTypeResolver>

<property name="forceBigDecimals" value="false"/>

</javaTypeResolver>

<!-- 生成模型的包名和位置-->

<javaModelGenerator targetPackage="cn.enjoy.model" targetProject="src/main/java">

<property name="enableSubPackages" value="true"/>

<property name="trimStrings" value="true"/>

</javaModelGenerator>

<!-- 生成映射文件的包名和位置-->

<sqlMapGenerator targetPackage="mapping" targetProject="src/main/resources">

<property name="enableSubPackages" value="true"/>

</sqlMapGenerator>

<!-- 生成DAO的包名和位置-->

<javaClientGenerator type="XMLMAPPER" targetPackage="cn.enjoy.dao" targetProject="src/main/java">

<property name="enableSubPackages" value="true"/>

</javaClientGenerator>

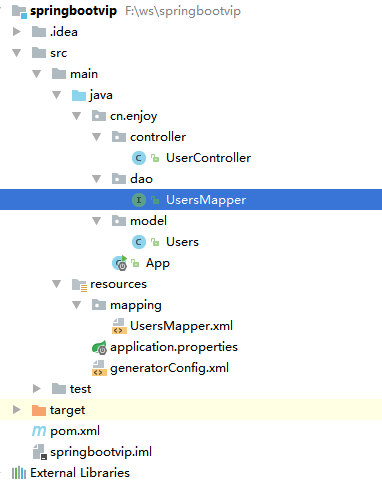
<!-- 要生成的表 tableName是数据库中的表名或视图名 domainObjectName是实体类名-->

<table tableName="enjoy\_user" domainObjectName="Users" enableCountByExample="false" enableUpdateByExample="false" enableDeleteByExample="false" enableSelectByExample="false" selectByExampleQueryId="false"></table>

</context>

</generatorConfiguration>

右键生成：结构图如下



注意：得再App启动类上增加@MapperScan扫描注解

@SpringBootApplication

@MapperScan("cn.enjoy.dao")

public class App {

public static void main(String[] args) throws Exception {

SpringApplication.run(App.class, args);

}

}

## 新增mapper登陆方法

Users findByUsernameAndPasswd(@Param("username") String username, @Param("passwd") String passwd);

对应的XML配置

<select id="findByUsernameAndPasswd" resultType="cn.enjoy.model.Users" parameterType="map">

select

<include refid="Base\_Column\_List" />

from enjoy\_user where 1=1

<if test="passwd != null" >

and passwd = #{passwd,jdbcType=VARCHAR}

</if>

<if test="username != null" >

and username = #{username,jdbcType=VARCHAR}

</if>

limit 1

</select>

## SpringBoot单元测试

要测试刚才新增的Mapper方法是否成功，这里需要单元测试，在springboot中是有专门的组件来做单元测试的，在pom文件中新增依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

</dependency>

新建UserTest放置在test目录下

package cn.enjoy.test;

import cn.enjoy.App;

import cn.enjoy.dao.UsersMapper;

import cn.enjoy.model.Users;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringRunner;

import javax.annotation.Resource;

@SpringBootTest(classes = {App.class})

@RunWith(SpringRunner.class)

public class UserTest {

@Resource

private UsersMapper usersMapper;

@Test

public void testAdd() {

Users user = new Users() ;

user.setPasswd("123");

user.setUsername("enjoy");

usersMapper.insertSelective(user);

}

@Test

public void testFindUser() {

Users enjoy = usersMapper.findByUsernameAndPasswd("enjoy", "123");

System.out.println(enjoy);

}

}

这样SpringBoot集成单元测试成功！

## 新建service

新增接口IUserService

public interface IUserService {

boolean login(String username,String passwd);

boolean register(String username,String passwd);

}

创建实现类

package cn.enjoy.service.impl;

import cn.enjoy.dao.UsersMapper;

import cn.enjoy.model.Users;

import cn.enjoy.service.IUserService;

import javax.annotation.Resource;

@Service

public class UserServiceImpl implements IUserService {

@Resource

private UsersMapper usersMapper;

@Override

public boolean login(String username, String passwd) {

Users users = usersMapper.findByUsernameAndPasswd(username, passwd);

return users != null;

}

@Override

public boolean register(String username, String passwd) {

Users users = new Users();

users.setUsername(username);

users.setPasswd(passwd);

int cnt = usersMapper.insertSelective(users);

return cnt > 0;

}

}

## 修改controller

修改UserController,增加login和register方法

|  |
| --- |
| package cn.enjoy.controller;  import cn.enjoy.service.IUserService;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  import javax.annotation.Resource;  @RestController  public class UserController {  @Resource  private IUserService iUserService;  @RequestMapping("/hello")  public Object sayHello() {  return "hello";  }  @RequestMapping("/login")  public String login(String username,String passwd) {  boolean login = iUserService.login(username, passwd);  if(login) {  return "登陆成功";  }else {  return "登陆失败";  }  }  @RequestMapping("/register")  public String register(String username,String passwd) {  boolean login = iUserService.register(username, passwd);  if(login) {  return "注册成功";  }else {  return "注册失败";  }  }  } |

在浏览器上输入：[localhost:8080/register?username=deer&passwd=123，显示“注册成功”](http://localhost:8080/register?username=deer&passwd=123，显示\“注册成功\”)

在浏览器上输入：localhost:8080/login?username=deer&passwd=123，显示“登陆成功”

## 事务支持

修改IUserService接口，增加一个新增batchAdd方法，在UserServiceImpl增加相应实现类，在实现类中故意产生一个被0整除得异常

package cn.enjoy.service;

public interface IUserService {

boolean login(String username,String passwd);

boolean register(String username,String passwd);

void batchAdd(String username,String passwd);

}

package cn.enjoy.service.impl;

import cn.enjoy.dao.UsersMapper;

import cn.enjoy.model.Users;

import cn.enjoy.service.IUserService;

import org.springframework.stereotype.Service;

import javax.annotation.Resource;

@Service

public class UserServiceImpl implements IUserService {

@Resource

private UsersMapper usersMapper;

@Override

public boolean login(String username, String passwd) {

Users users = usersMapper.findByUsernameAndPasswd(username, passwd);

return users != null;

}

@Override

public boolean register(String username, String passwd) {

Users users = new Users();

users.setUsername(username);

users.setPasswd(passwd);

int cnt = usersMapper.insertSelective(users);

return cnt > 0;

}

@Override

public void batchAdd(String username, String passwd) {

Users users = new Users();

users.setUsername(username);

users.setPasswd(passwd);

usersMapper.insertSelective(users);

int i = 10 /0;

users = new Users();

users.setUsername(username+"2");

users.setPasswd(passwd);

usersMapper.insertSelective(users);

}

}

修改UserContoller,增加batchAdd方法

@RequestMapping("/batchAdd")

public String batchAdd(String username,String passwd) {

iUserService.batchAdd(username, passwd);

return "成功";

}

重新运行，在浏览器上输入：localhost:8080/batchAdd?username=enjoy&passwd=123

可以发现在浏览器上出现



检查数据库，发现表里面已经产生了一个错误的数据，产生了事务问题。

**事务支持：**

在batchAdd方法上增加@Transactional注解，重启服务后，在浏览器上输入

localhost:8080/batchAdd?username=enjoy&passwd=123

浏览器还继续报错，但检查数据库，事务问题已经得到了解决

## 全局异常处理

通过上面步骤，虽然已经解决了事务问题，但界面上出现这500错误，这对用户来说还是不友好。

一般在企业里面对这些异常一般都会统一捕获，由一个专门的异常处理类来统一处理。

### 异常捕获

package cn.enjoy.utils;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.ResponseBody;

import javax.servlet.http.HttpServletRequest;

/\*\*

\* 异常处理类

\*/

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(value = RuntimeException.class)

@ResponseBody

public Object defaultErrorHandler(HttpServletRequest req, Exception e) throws Exception {

e.printStackTrace();

return "我是个异常处理类";

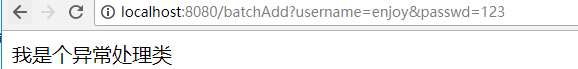
}

}

重启服务后，在浏览器上输入会出现异常的服务

localhost:8080/batchAdd?username=enjoy&passwd=123

界面返回：



### 404页面处理

在浏览器上故意输错地址

[localhost:8080/batchAddx?username=enjoy&passwd=123，后端并没有这服务，虽然已经做了相关的异常捕获](http://localhost:8080/batchAddx?username=enjoy&passwd=123，后端并没有这服务，虽然已经做了相关的异常捕获)，但浏览器还是显示了。



这个时候就要做404（其他异常代码一样）

在配置这样错误页面的时候，以前是在WEB.XML中进行配置，而在这里，需要有个WebServerFactoryCustomizer的实例进行配置

在前面建立的GlobalExceptionHandler，新建一个方法

@Bean

public WebServerFactoryCustomizer<ConfigurableWebServerFactory> webServerFactoryCustomizer(){

return (factory->{

ErrorPage error404Page = new ErrorPage(HttpStatus.NOT\_FOUND, "/404.do");

factory.addErrorPages( error404Page);

});

}

新建BaseController

package cn.enjoy.controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class BaseController {

@RequestMapping("/404.do")

public Object error\_404() {

return "你要找的页面，被lison偷吃了！";

}

}

重启服务，在浏览器上输入

localhost:8080/batchAddx?username=enjoy&passwd=123

此时，页面返回“你要找的页面，被lison偷吃了！”

注意：WebServerFactoryCustomizer这种配置方式是在SpringBoot2之后才这样配置的，在1.X的版本需要用到的是EmbeddedServletContainerCustomizer

代码如下

@Bean

public EmbeddedServletContainerCustomizer containerCustomizer() {

return (container -> {

ErrorPage error404Page = new ErrorPage(HttpStatus.NOT\_FOUND, "/404.do");

container.addErrorPages( error404Page);

});

}

## 静态资源访问

静态资源：js, css, html, 图片，音视频等

静态资源路径：是指系统可以直接访问的路径，且路径下的所有文件均可被用户直接读取。

Spring Boot默认提供静态资源目录位置需置于classpath下，目录名需符合如下规则：

/static

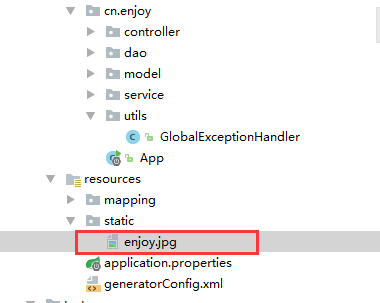
/public

/resources

/META-INF/resources

在resources目录下面建立static文件夹，在文件夹里面任意放张图片。

命名为：enjoy.jpg



在地址栏上输入localhost:8080/enjoy.jpg，可以看到图片

## 前端界面

### JSP集成

一般来说springboot不建议直接使用jsp页面，但不排除有些公司的项目依然使用jsp做前端界面。

springboot内置的tomcat并没有集成对jsp的支持，也没有对EL表达式的支持，因此要使用jsp应该先把相关的依赖集成进来

在pom文件里面新增

<!--JavaServer Pages Standard Tag Library，JSP标准标签库-->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

<!--内置tocat对Jsp支持的依赖，用于编译Jsp-->

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

</dependency>

由于要springmvc解析jsp,要配置试图解析器，在applicaiton.properties 里面新增

spring.mvc.view.prefix=/WEB-INF/jsp/

spring.mvc.view.suffix=.jsp

在resources里面新建WEB-INF文件夹，在里面放一个index.jsp页面



内容如下：

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>享学课堂</title>

</head>

<body>

<h1>这是个jsp页面!!</h1>

</body>

</html>

最后新建一个controller，注意这里的注解是@Controller,千万不能用@RestController

package cn.enjoy.controller;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

@Controller

@RequestMapping("/jsp")

public class JspController {

@RequestMapping("/hi")

public String sayHello() {

return "index";

}

}

在浏览器上输入：localhost:8080/jsp/hi，可以看到JSP页面。

### 模板引擎

SpringBoot 推荐使用模板引擎来渲染html，如果你不是历史遗留项目，一定不要使用JSP，常用的模板引擎很多，有freemark,thymeleaf等，其实都大同小异

其中springboot 强烈推荐的是用thymeleaf

pom文件种添加thymeleaf的支持，并且删除JSP的支持

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

~~<!--&lt;!&ndash;JavaServer Pages Standard Tag Library，JSP标准标签库&ndash;&gt;-->~~

~~<!--<dependency>-->~~

~~<!--<groupId>javax.servlet</groupId>-->~~

~~<!--<artifactId>jstl</artifactId>-->~~

~~<!--</dependency>-->~~

~~<!--&lt;!&ndash;内置tocat对Jsp支持的依赖，用于编译Jsp&ndash;&gt;-->~~

~~<!--<dependency>-->~~

~~<!--<groupId>org.apache.tomcat.embed</groupId>-->~~

~~<!--<artifactId>tomcat-embed-jasper</artifactId>-->~~

~~<!--</dependency>-->~~

删除application.properties文件里面视图解析器内容

~~#spring.mvc.view.prefix=/WEB-INF/jsp/~~

~~#spring.mvc.view.suffix=.jsp~~

新建Controller内容如下

package cn.enjoy.controller;

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

import org.springframework.web.bind.annotation.RequestMapping;

@Controller

@RequestMapping("/tpl")

public class ThymeleafController {

@RequestMapping("/testThymeleaf")

public String testThymeleaf(ModelMap map) {

// 设置属性

map.addAttribute("name", "enjoy");

// testThymeleaf：为模板文件的名称

// 对应src/main/resources/templates/testThymeleaf.html

return "testThymeleaf";

}

}

Springboot默认的模板配置路径为：src/main/resources/templates

在resources目录里面新建一个templates目录，在目录里面新建testThymeleaf.html文件

<!DOCTYPE html>

<html xmlns:th="http://www.w3.org/1999/xhtml">

<head lang="en">

<meta charset="UTF-8" />

<title>enjoy</title>

</head>

<body>

<h1 th:text="${name}"/>

</body>

</html>

在浏览器上输入：localhost:8080/tpl/testThymeleaf，可以看到页面。

## 集成Swagger2构建API文档

Swagger2 的作用

* 随项目自动生成强大RESTful API文档，减少工作量
* API文档与代码整合在一起，便于同步更新API说明
* 页面测试功能来调试每个RESTful API

修改pom文件，添加swagger2的相关依赖

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.7.0</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.7.0</version>

</dependency>

新建一个swagger的配置类SwaggerConfig.java

package cn.enjoy.utils;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.ApiInfoBuilder;

import springfox.documentation.builders.PathSelectors;

import springfox.documentation.builders.RequestHandlerSelectors;

import springfox.documentation.service.ApiInfo;

import springfox.documentation.service.Contact;

import springfox.documentation.spi.DocumentationType;

import springfox.documentation.spring.web.plugins.Docket;

import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration

@EnableSwagger2

public class SwaggerConfig {

@Bean

public Docket createRestApi() {

return new Docket(DocumentationType.SWAGGER\_2)

.apiInfo(apiInfo())

.select()

.apis(RequestHandlerSelectors.basePackage("cn.enjoy"))// 指定扫描包下面的注解

.paths(PathSelectors.any())

.build();

}

// 创建api的基本信息

private ApiInfo apiInfo() {

return new ApiInfoBuilder()

.title("集成Swagger2构建RESTful APIs")

.description("集成Swagger2构建RESTful APIs")

.termsOfServiceUrl("http://www.xiangxueketang.cn/")

.contact(new Contact("enjoy","cn.xiangxueketang","enjoy@enjoy.cn"))

.version("1.0.0")

.build();

}

}

新建Controller用于显示相关接口

package cn.enjoy.controller;

import io.swagger.annotations.ApiImplicitParam;

import io.swagger.annotations.ApiOperation;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;

import java.util.Map;

@RestController

@RequestMapping(value="/swagger")

public class SwaggerController {

@ApiOperation(value="获取用户信息", notes="根据id来获取用户详细信息")

@ApiImplicitParam(name="id", value="用户ID", required=true, dataType="String")

@RequestMapping(value="/{id}", method= RequestMethod.GET)

public Map<String,String> getInfo(@PathVariable String id) {

Map<String ,String> map = new HashMap<String, String>();

map.put("name", "lison");

map.put("age", "38");

return map;

}

}

访问：http://localhost:8080/swagger-ui.html

## 日志集成

java有许多的日志组件，比如 log4j,log4j2,logback还有java自生提供的Java Util Logging，其实在springboot中对这些组件都提供了支持，log4j，log4j2和logback都提供相应的组件支持。

### Logback

在springboot中默认使用的日志工具是logback,不过在提及具体的日志工具之前要提一个名词，这个名词就是slf4j(Simple Logging Facade For Java)

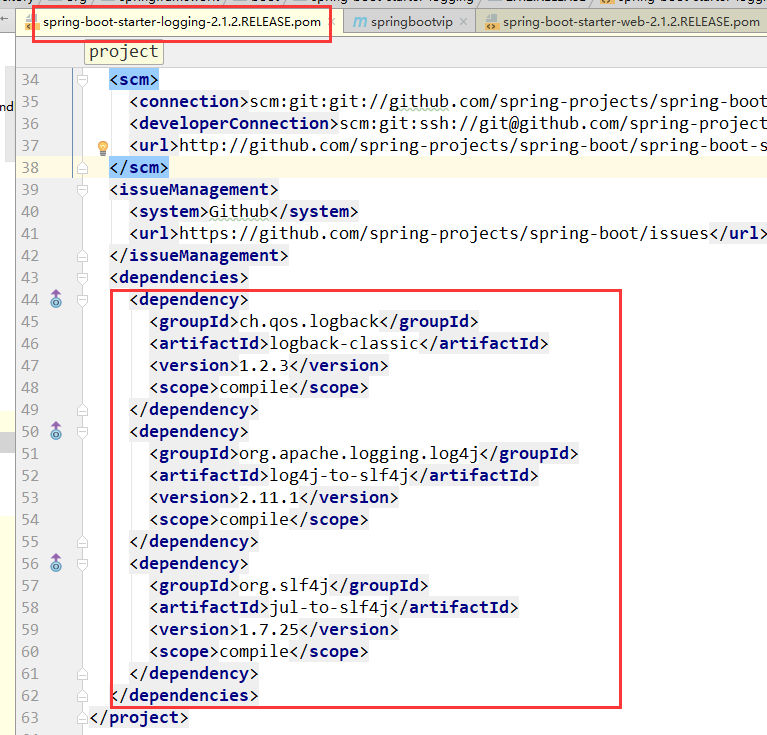
百度百科解释

<https://baike.baidu.com/item/slf4j/6408868>

slf4j不是具体的日志解决方案，它有点类似于jdbc,使用了门面模式，是一个针对各类日志的抽象实现，既然是抽象的日志实现，在springboot中肯定不需要额外导入。

注意：spring-boot-starter中就提供了对spring-boot-starter-logging的依赖

在spring-boot-starter-logging中可以看到以及集成了slf4j与具体实现logback的默认支持

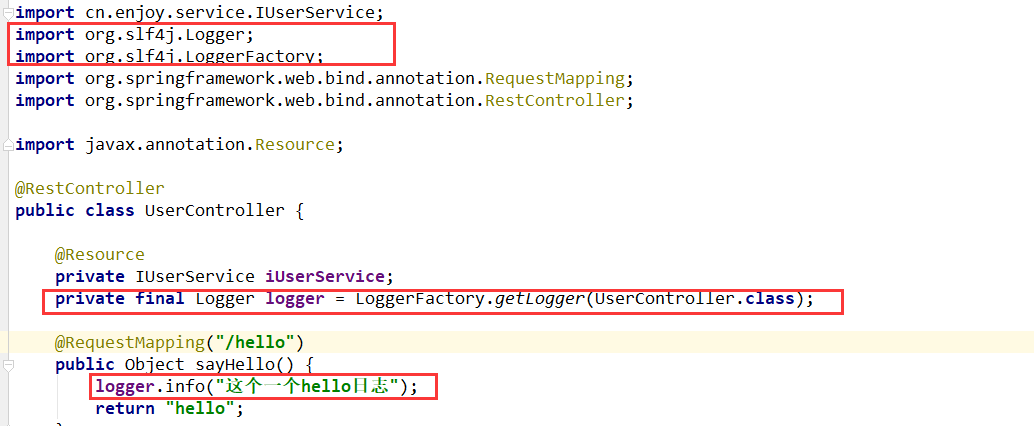


修改UserController

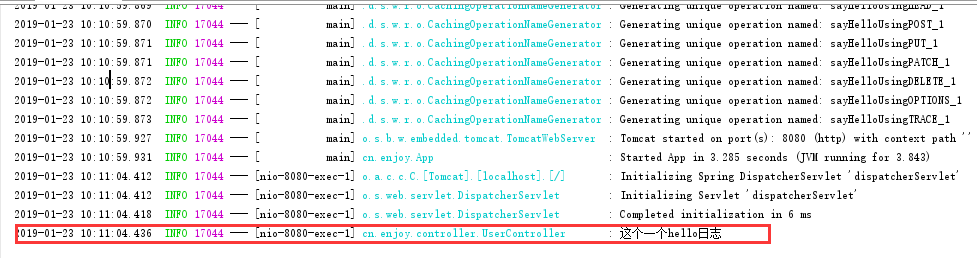
import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

private final Logger logger = LoggerFactory.getLogger(UserController.class);



在浏览器上输入：localhost:8080/hello，可以看控制台日志的输出



#### 日志级别

修改controller 把日志的输出改成

logger.debug("这个一个hello日志");

这个时候重启，再调用，发现后台并不会有任何输出，这原因是日志级别在作祟

默认情况下，Spring Boot 配置的是INFO 日志级别，也就是会输出INFO级别以上的日志(ERROR, WARN, INFO)。

如果需要 Debug 级别的日志。在 src/main/resources/application.properties 中配置。

debug=true

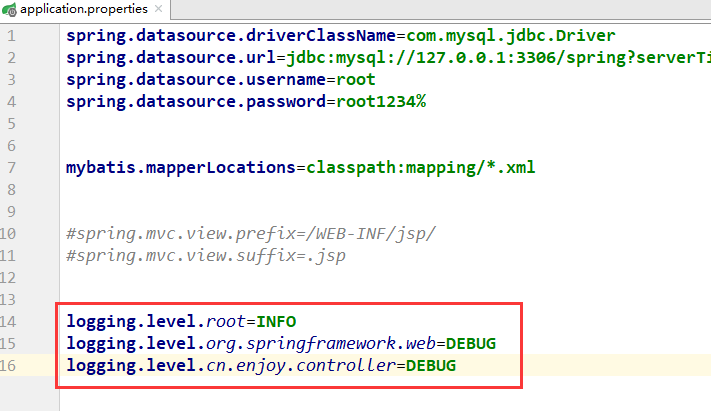
此外，配置 logging.level.\* 来具体输出哪些包的日志级别。

例如

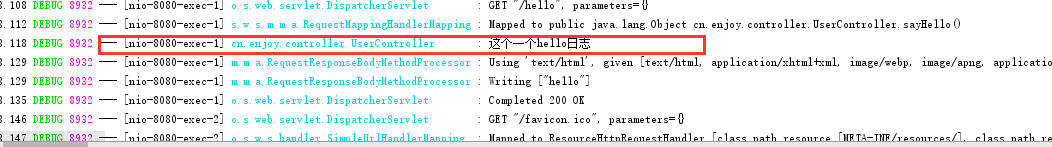
logging.level.root=INFO

logging.level.org.springframework.web=DEBUG

logging.level.cn.enjoy.controller=DEBUG



这个时候，包括springframework.web以及cn.enjoy.controller的debug日志都可以输出来了



#### 日志文件

一般情况下，springboot日志只会输出到控制台，并不会写入到日志文件，但是，在一些正式环境的应用中，我们需要通过在 application.properites 文件中配置 logging.file 文件名称和 logging.path 文件路径，将日志输出到日志文件中。

logging.path = /var/tmp

logging.file = xxx.log

logging.level.root = info

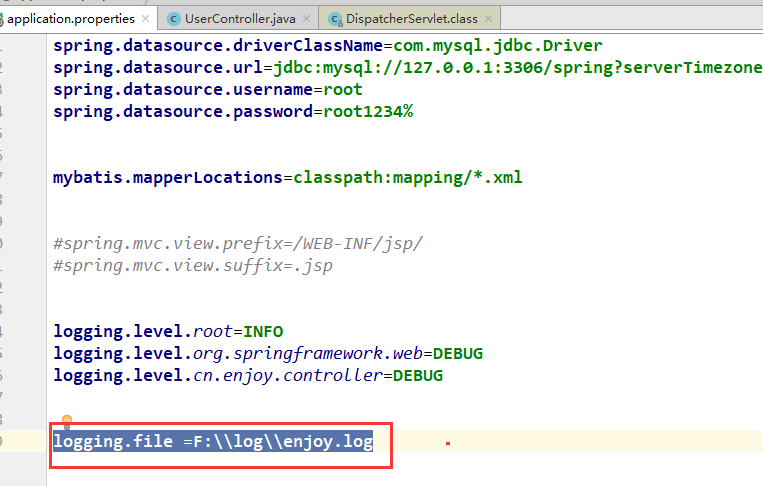
注意：

如果只配置 logging.path，在 /var/tmp文件夹生成一个日志文件为 spring.log。如果只配置 logging.file，会在项目的当前路径下生成一个 xxx.log 日志文件。

这里有一个坑，logging.path 和logging.file都配置了，只会有logging.file生效，所以，如果要指定日志生成的具体位置使用logging.file 配置就好

在application.properties中配置

logging.file =F:\\log\\enjoy.log



这样在F盘的相应位置出现日志文件

### log4j2

在spring-boot-dependencies POMs中搜索spring-boot-starter-log4j2

发现Spring boot父Pom中自己提供了这个依赖，于是我们加入如下jar依赖：

修改pom.xml文件

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-log4j2</artifactId>

</dependency>

注意: 由于默认使用logback在扩展log4j2之前先要把logback移除

日志使用跟上面logback一样。

### log4j

如果不只为了学习集成log4j,在工作是最好不要使用log4j,毕竟有了log4j2,也有了logback使用log4j就是吃饱了撑着

在springboot中并没有提供对log4j这个依赖的支持，因此要使用它配置起来还是挺麻烦的。

在： mvnrepository.com 中发现log4j最新版本spring-boot-starter-log4j是1.3.8.RELEASE

修改pom.xml文件

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-logging</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-log4j</artifactId>  <version>1.3.8.RELEASE</version>  </dependency> |

在classpath下增加log4j.properties文件

|  |
| --- |
| log4j.rootCategory=INFO, stdout, file, errorfile  log4j.category.cn.enjoy=INFO, myFile  log4j.logger.error=errorfile  # 控制台输出  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L - %m%n  # root日志输出  log4j.appender.file=org.apache.log4j.DailyRollingFileAppender  log4j.appender.file.file=logs/all.log  log4j.appender.file.DatePattern='.'yyyy-MM-dd  log4j.appender.file.layout=org.apache.log4j.PatternLayout  log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L - %m%n  # error日志输出  log4j.appender.errorfile=org.apache.log4j.DailyRollingFileAppender  log4j.appender.errorfile.file=logs/error.log  log4j.appender.errorfile.DatePattern='.'yyyy-MM-dd  log4j.appender.errorfile.Threshold = ERROR  log4j.appender.errorfile.layout=org.apache.log4j.PatternLayout  log4j.appender.errorfile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L - %m%n  # cn.enjoy下的日志输出  log4j.appender.myFile=org.apache.log4j.DailyRollingFileAppender  log4j.appender.myFile.file=logs/my.log  log4j.appender.myFile.DatePattern='.'yyyy-MM-dd  log4j.appender.myFile.layout=org.apache.log4j.PatternLayout  log4j.appender.myFile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L ---- %m%n |

在代码中使用

import org.apache.log4j.Logger;

private final Logger logger = Logger.getLogger(xxx.class);

### 使用AOP统一日志处理

为了防止在工作中经常在代码中加入大量的日志处理代码，在实际项目开发中，一般使用AOP统一完成日志处理工作

修改pom文件，引入springboot对aop的支持

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency> |

新增AOP日志处理类

|  |
| --- |
| package cn.enjoy.utils;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.annotation.AfterReturning;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.aspectj.lang.annotation.Pointcut;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.stereotype.Component;  import org.springframework.web.context.request.RequestContextHolder;  import org.springframework.web.context.request.ServletRequestAttributes;  import javax.servlet.http.HttpServletRequest;  import java.util.Enumeration;  @Aspect  @Component  public class WebLogAspect {  private static final Logger logger = LoggerFactory.getLogger(WebLogAspect.class);  @Pointcut("execution(public \* cn.enjoy.controller.\*.\*(..))")  public void webLog() {  }  @Before("webLog()")  public void doBefore(JoinPoint joinPoint) throws Throwable {  // 接收到请求，记录请求内容  ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();  HttpServletRequest request = attributes.getRequest();  // 记录下请求内容  logger.info("URL : " + request.getRequestURL().toString());  logger.info("HTTP\_METHOD : " + request.getMethod());  logger.info("IP : " + request.getRemoteAddr());  Enumeration<String> enu = request.getParameterNames();  while (enu.hasMoreElements()) {  String name = (String) enu.nextElement();  logger.info("name:{},value:{}", name, request.getParameter(name));  }  }  @AfterReturning(returning = "ret", pointcut = "webLog()")  public void doAfterReturning(Object ret) throws Throwable {  // 处理完请求，返回内容  logger.info("RESPONSE : " + ret);  }  } |

## Spring Boot 热加载/部署

热部署不会用在生产环境，但对于程序员开发的效率，还是有一定帮助的，所谓的热部署，就是在应用程序在不停止的情况下，实现新的部署

spring-boot-devtools 是一个为开发者服务的一个模块，其中最重要的功能就是自动应用代码更改到最新的App上面去。原理是在发现代码有更改之后，重新启动应用，但是速度比手动停止后再启动还要更快，更快指的不是节省出来的手工操作的时间。

其深层原理是使用了两个ClassLoader，一个Classloader加载那些不会改变的类（第三方Jar包），另一个ClassLoader加载会更改的类，称为 restart ClassLoader

,这样在有代码更改的时候，原来的restart ClassLoader 被丢弃，重新创建一个restart ClassLoader，由于需要加载的类相比较少，所以实现了较快的重启时间

修改pom文件，增加

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

修改properties文件

如果使用的 Thymeleaf 模板，那么请直接在application.properties中添加

spring.thymeleaf.cache=false

如果使用的 FreeMarker 模板，那么请直接在application.properties中添加

spring.freemarker.cache=false

### IDEA中使用

如果你是使用eclipse,请忽略，但如果你是使用IDEA，由于idea 没有保存修改的，也就是说在idea中并不会因为你ctrl+s 就重新编译代码。

那么就需要额外的配置

在pom文件中，增加编译插件，让代码有变动的时候也编译

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

**<!-- 如果不设置fork,那么不会restart,devtools热部署不会起作用-->**

<fork>true</fork>

</configuration>

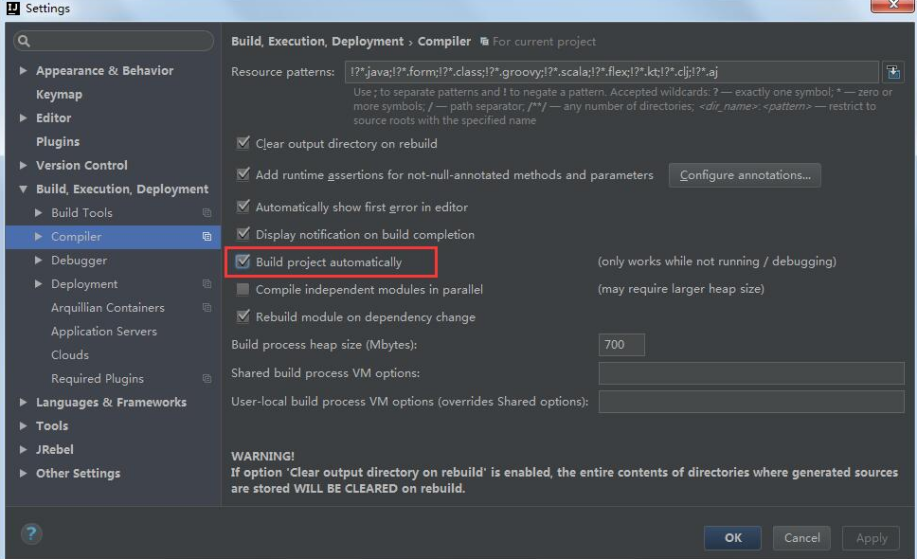
</plugin>

</plugins>

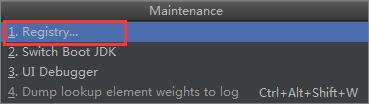
</build>

1. 手动：修改完代码，按快捷键Ctrl+F9，手动构建项目，或者只修改单个类文件的话，按Ctrl+Shift+F9，重新编译该类文件，即可触发重启服务。
2. 自动

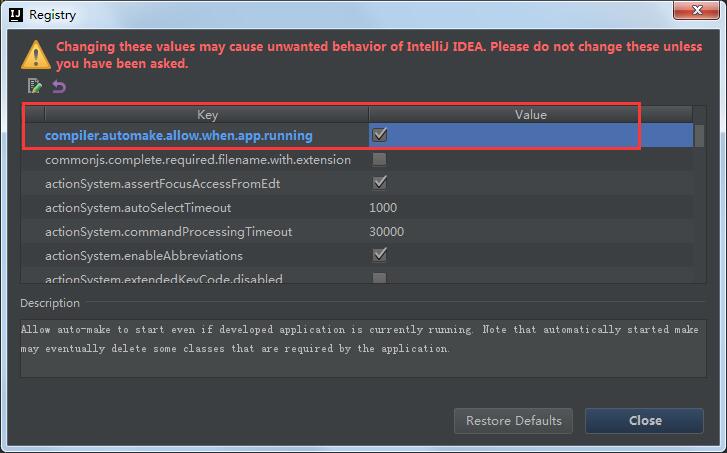
        1）File -> Settings -> Compiler，勾选 Build Project automatically



         2）按快捷键Ctrl+Shift+Alt+/，选择1.Registry...



         3）勾选 compiler.automake.allow.when.app.running 即可



这个时候修改JAVA文件或者模板文件都自动会生效

## 编译，打包

快速入门进行到这里，其实已经差不多了，能应对绝大多数开发常见，接下来就是导包部署。

在pom文件中新增

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<!-- 如果不设置fork,那么不会restart,devtools热部署不会起作用-->

<fork>true</fork>

</configuration>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

使用mvn clean package 打包

java –jar 运行

### war部署

如果并不希望使用内置的tomcat，希望部署到其他tomcat服务器，那么就需要使用war包部署了。

修改pom文件，打包方式改成war



修改在pom文件，剔除内置tomcat的支持，否则会和外面的tomcat冲突

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

<!--打包的时候可以不用包进去，别的设施会提供。事实上该依赖理论上可以参与编译，测试，运行等周期。

相当于compile，但是打包阶段做了exclude操作-->

<scope>provided</scope>

</dependency>

修改启动类，使其继承

org.springframework.boot.web.servlet.support.SpringBootServletInitializer，并重写configure方法

package cn.enjoy;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.builder.SpringApplicationBuilder;

import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication

@MapperScan("cn.enjoy.dao")

public class App extends SpringBootServletInitializer {

public static void main(String[] args) throws Exception {

SpringApplication.run(App.class, args);

}

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {

return builder.sources(App.class);

}

}

使用mvn clean package 打包

把war包拷贝到tomcat webapps中

访问：http://localhost:8080/springbootvip/hello

# 进阶部分

## 集成Redis

修改pom文件，增加依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

在properties增加redis配置

# REDIS (RedisProperties)

# Redis数据库索引（默认为0）

spring.redis.database=0

# Redis服务器地址

spring.redis.host=127.0.0.1

# Redis服务器连接端口

spring.redis.port=6379

# Redis服务器连接密码（默认为空）

spring.redis.password=

# 连接超时时间（毫秒）

spring.redis.timeout=5000

新建单元测试

package cn.enjoy.test;

import cn.enjoy.App;

import junit.framework.TestCase;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.data.redis.core.RedisTemplate;

import org.springframework.data.redis.core.ValueOperations;

import org.springframework.test.context.junit4.SpringRunner;

import javax.annotation.Resource;

@SpringBootTest(classes = {App.class})

@RunWith(SpringRunner.class)

public class SpringRedisTest {

@Resource

private RedisTemplate<String,String> redisTemplate;

@Test

public void testRedis() throws Exception {

ValueOperations<String, String> ops = redisTemplate.opsForValue();

ops.set("name", "enjoy");

String value = ops.get("name");

System.out.println(value);

}

}

## 集成RabbitMQ

RabbitMq在后面讲到springcloud的时候还会用到，在这会简单的安装使用下RabbitMq，至于RabbitMQ的详细使用请参考分布式消息队列课题

### Windows安装RabbitMQ

先安装Erlang

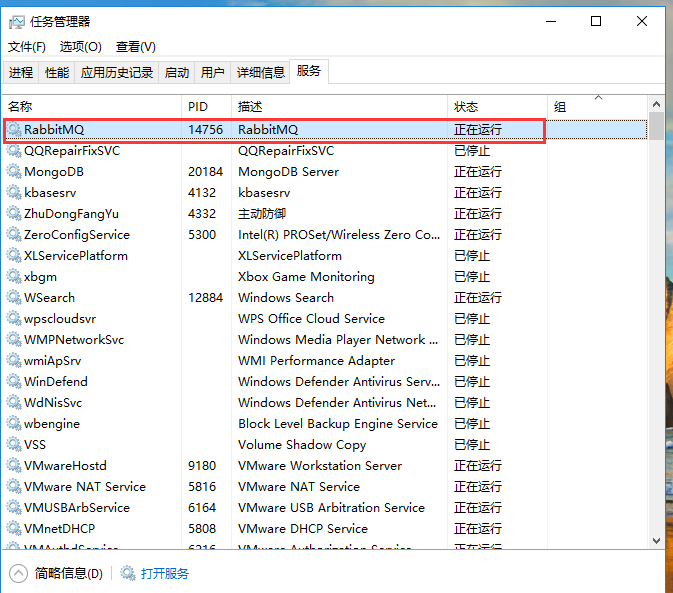
下载地址：

<http://erlang.org/download/otp_win64_20.3.exe>

RabbitMQ Server 3.7.4下载地址

<https://bintray.com/rabbitmq/all/download_file?file_path=rabbitmq-server%2F3.7.4%2Frabbitmq-server-3.7.4.exe>

安装好后，启动服务



开启web插件

进入rabbitmq安装目录的sbin目录，在命令行界面开启web插件

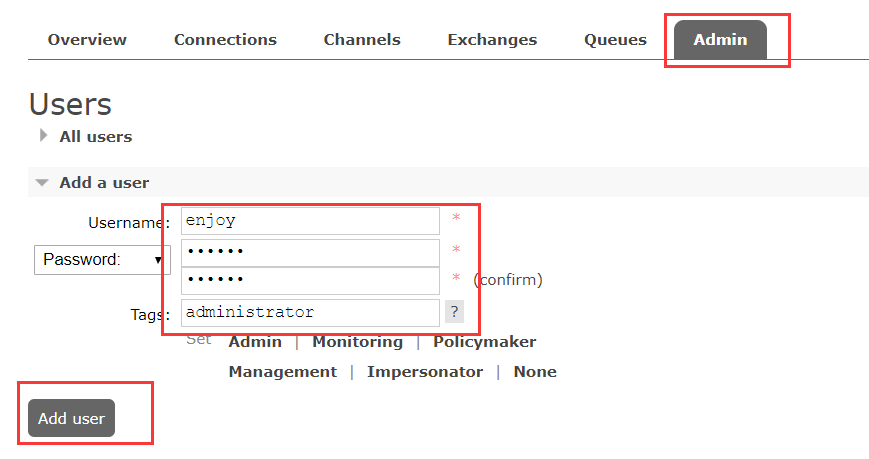
rabbitmq-plugins enable rabbitmq\_management



重新启动reabbitmq服务，在地址栏输入<http://localhost:15672/>

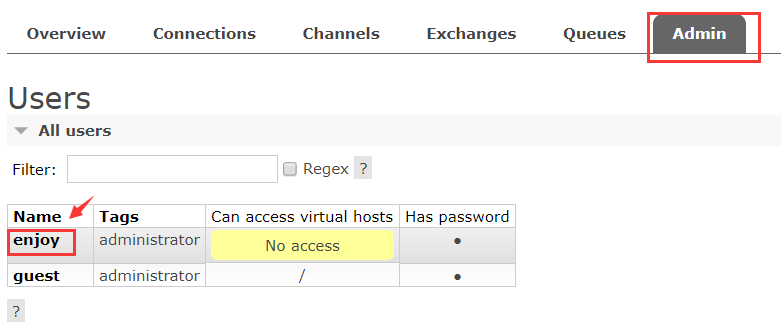
使用默认用户 guest/guest登陆界面

### 新增用户，并设置权限

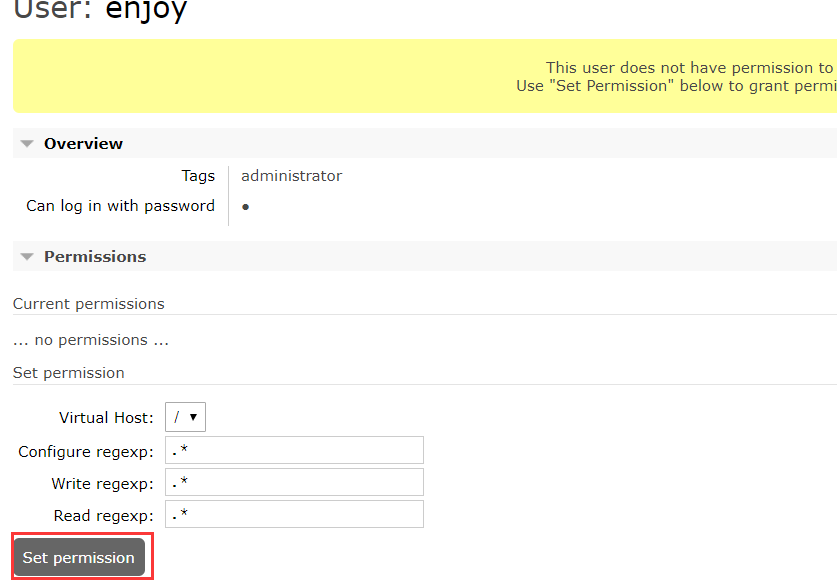


用新创建的enjoy用户登陆，发现新用户已经新增成功

设置权限



点击这用户，设置虚拟主机的权限（全部可读，可写，可配置）



这个时候一个enjoy新用户设置完毕

### SpringBoot中使用RabbitMq

修改pom文件，增加rabbitmq依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

修改配置properties文件，增加rabbitmq的连接信息

## rabbitmq config

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=enjoy

spring.rabbitmq.password=123456

创建Rabbit配置类，用来配置队列，交换器，路由等高级信息

package cn.enjoy.mq;

import org.springframework.amqp.core.Queue;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class RabbitConfig {

@Bean

public Queue firstQueue() {

// 创建一个队列，名称为：enjoy

return new Queue("enjoy");

}

}

创建消息的生产者

package cn.enjoy.mq;

import org.springframework.amqp.core.AmqpTemplate;

import org.springframework.stereotype.Component;

import javax.annotation.Resource;

@Component

public class Sender {

@Resource

private AmqpTemplate rabbitTemplate;

public void send() {

rabbitTemplate.convertAndSend("enjoy", "this a messages !!!");

}

}

创建消息的消费者

package cn.enjoy.mq;

import org.springframework.amqp.rabbit.annotation.RabbitHandler;

import org.springframework.amqp.rabbit.annotation.RabbitListener;

import org.springframework.stereotype.Component;

@Component

//定义该类需要监听的队列

@RabbitListener(queues = "enjoy")

public class Receiver {

@RabbitHandler //指定对消息的处理

public void process(String msg) {

System.out.println("receive msg : " + msg);

}

}

新增单元测试

package cn.enjoy.test;

import cn.enjoy.App;

import cn.enjoy.mq.Sender;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringRunner;

import javax.annotation.Resource;

@RunWith(SpringRunner.class)

@SpringBootTest(classes = App.class)

public class RabbitmqTest {

@Resource

private Sender sender;

@Test

public void testRabbitmq() throws Exception {

sender.send();

}

}

## Actuator监控管理

Actuator是spring boot的一个附加功能,可帮助你在应用程序生产环境时监视和管理应用程序。可以使用HTTP的各种请求来监管,审计,收集应用的运行情况.特别对于微服务管理十分有意义

缺点：没有可视化界面(Spring cloud 还会用到这功能，就可以看到界面了)

修改pom文件，添加依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

修改application.properties文件，启动监控端点

# 加载所有的端点/默认只加载了 info / health

management.endpoints.web.exposure.include=\*

# 描述信息

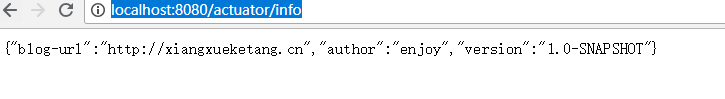
info.blog-url=http://xiangxueketang.cn

info.author=enjoy

info.version=@project.version@

重新启动，在地址栏输入

<http://localhost:8080/actuator/info>



在界面看到这说明监控成功

Actuator访问路径

通过actuator/+端点名就可以获取相应的信息。

|  |  |
| --- | --- |
| 路径 | 作用 |
| /actuator/beans | 显示应用程序中所有Spring bean的完整列表。 |
| /actuator/configprops | 显示所有配置信息。 |
| /actuator/env | 陈列所有的环境变量。 |
| /actuator/mappings | 显示所有@RequestMapping的url整理列表。 |
| /actuator/health | 显示应用程序运行状况信息 up表示成功 down失败 |
| /actuator/info | 查看自定义应用信息 |

## 自定义Starter

在学习SpringBoot的过程中，不管是集成redis还是RabbitMQ，甚至是前面集成mybatis已经学习了很多starter，这些starter都是springboot为我们提供的一些封装，这些starter能非常方便快捷的增加功能，并不需要很多配置，即使需要配置也就在application.properties稍微配置下就可以了。

那么接下来就学习下怎么创建属于自己的starter

### redis-starter插件

前面已经使用过spring-boot-starter-data-redis，这个starter是用来集成redis的，那么接下来完成一个starter,这个starter也就集成下redis

新建一个项目，这个项目不需要web功能

pom文件内容如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>enjoy</groupId>

<artifactId>redis-starter</artifactId>

<version>1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>3.0.1</version>

</dependency>

</dependencies>

</project>

创建一个RedisProperties用于加载Redis需要的配置，这里为简单起见，并没有设置密码

package cn.enjoy.redis;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "redis")

public class RedisProperties {

private String host;

private int port;

public int getPort() {

return port;

}

public void setPort(int port) {

this.port = port;

}

public String getHost() {

return host;

}

public void setHost(String host) {

this.host = host;

}

}

创建一个配置类，这个配置类用于加载配置，并实例化Jedis客户端

package cn.enjoy.redis;

import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;

import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;

import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import redis.clients.jedis.Jedis;

@Configuration //开启配置

@ConditionalOnClass(Jedis.class)

@EnableConfigurationProperties(RedisProperties.class) //开启使用映射实体对象

@ConditionalOnProperty//存在对应配置信息时初始化该配置类

(

prefix = "redis",//存在配置前缀redis

value = "enabled",//开启

matchIfMissing = true//缺失检查

)

public class RedisAutoConfiguration {

@Bean

@ConditionalOnMissingBean

public Jedis jedis(RedisProperties redisProperties){

return new Jedis(redisProperties.getHost(), redisProperties.getPort());

}

}

自动化配置代码中有很多我们之前没有用到的注解配置，我们从上开始讲解

@Configuration：这个配置就不用多做解释了，我们一直在使用

@EnableConfigurationProperties：这是一个开启使用配置参数的注解，value值就是我们配置实体参数映射的ClassType，将配置实体作为配置来源。

SpringBoot内置条件注解

有关@ConditionalOnXxx相关的注解这里要系统的说下，因为这个是我们配置的关键，根据名称我们可以理解为具有Xxx条件，当然它实际的意义也是如此，条件注解是一个系列，下面我们详细做出解释

@ConditionalOnBean：当SpringIoc容器内存在指定Bean的条件

@ConditionalOnClass：当SpringIoc容器内存在指定Class的条件

@ConditionalOnExpression：基于SpEL表达式作为判断条件

@ConditionalOnJava：基于JVM版本作为判断条件

@ConditionalOnMissingBean：当SpringIoc容器内不存在指定Bean的条件

@ConditionalOnMissingClass：当SpringIoc容器内不存在指定Class的条件

@ConditionalOnNotWebApplication：当前项目不是Web项目的条件

@ConditionalOnProperty：指定的属性是否有指定的值

@ConditionalOnResource：类路径是否有指定的值

@ConditionalOnSingleCandidate：当指定Bean在SpringIoc容器内只有一个，或者虽然有多个但是指定首选的Bean

@ConditionalOnWebApplication：当前项目是Web项目的条件

以上注解都是元注解@Conditional演变而来的，根据不用的条件对应创建以上的具体条件注解。

到目前为止我们还没有完成自动化配置starter，我们需要了解SpringBoot运作原理后才可以完成后续编码。

Starter自动化运作原理

在注解@SpringBootApplication上存在一个开启自动化配置的注解@EnableAutoConfiguration来完成自动化配置，注解源码如下所示：

@AutoConfigurationPackage

@Import({EnableAutoConfigurationImportSelector.class})

public @interface EnableAutoConfiguration {

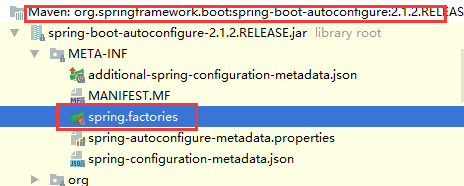
String ENABLED\_OVERRIDE\_PROPERTY = "spring.boot.enableautoconfiguration";

Class<?>[] exclude() default {};

String[] excludeName() default {};

}

在@EnableAutoConfiguration注解内使用到了@import注解来完成导入配置的功能，而EnableAutoConfigurationImportSelector内部则是使用了SpringFactoriesLoader.loadFactoryNames方法进行扫描具有META-INF/spring.factories文件的jar包。我们可以先来看下spring-boot-autoconfigure包内的spring.factories文件内容，如下所示：



可以看到配置的结构形式是Key=>Value形式，多个Value时使用,隔开，那我们在自定义starter内也可以使用这种形式来完成，我们的目的是为了完成自动化配置，所以我们这里Key则是需要使用org.springframework.boot.autoconfigure.EnableAutoConfiguration

**自定义spring.factories**

我们在src/main/resource目录下创建META-INF目录，并在目录内添加文件spring.factories，具体内容如下所示：

#配置自定义Starter的自动化配置

org.springframework.boot.autoconfigure.EnableAutoConfiguration=cn.enjoy.redis.RedisAutoConfiguration

目前为止自定义的starter已经开发完毕

### 新建项目测试startser

创建一个新的项目，这项目用来测试前面创建的redis-starter

Pom文件配置如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>enjoy</groupId>

<artifactId>testRedisStarter</artifactId>

<version>1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

</dependency>

<dependency>

<groupId>enjoy</groupId>

<artifactId>redis-starter</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

</dependencies>

</project>

新建一个springboot启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class,args);

}

}

新建application.properties在里面配置redis连接相关信息

redis.port=6379

redis.host=127.0.0.1

准备好这些后，启动redis，新建立一个测试类，运行测试方法

package cn.enjoy.test;

import cn.enjoy.App;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringRunner;

import redis.clients.jedis.Jedis;

import javax.annotation.Resource;

@SpringBootTest(classes = App.class)

@RunWith(SpringRunner.class)

public class RedisTest {

@Resource

private Jedis jedis;

@Test

public void test() {

jedis.set("enjoy","enjoy");

String enjoy = jedis.get("enjoy");

System.out.println(enjoy);

}

}

到这一步，自定义redis-stater搞定

## SpringBoot CLI

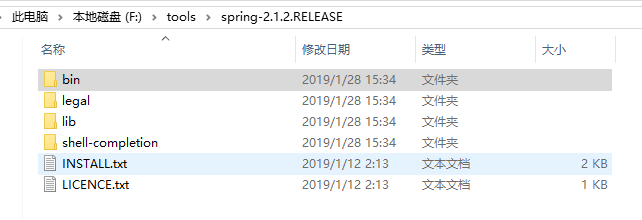
Spring Boot CLI是一个命令行工具，如果想使用Spring进行快速开发可以使用它。它允许你运行Groovy脚本，这意味着你可以使用熟悉的类Java语法，并且没有那么多的模板代码。你可以通过Spring Boot CLI启动新项目，或为它编写命令。

Groovy是个基于JVM(Java虚拟机)的敏捷开发语音，既然是基于jvm,那么在groovy里面使用任何java的组件他都是可以支持识别的，在大概5，6年前Groovy比较火，尤其微信公众刚开放的那段时间，很多微信的后端程序都是基于grails开发的。

解压安装SpringBoot CLI

https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/2.1.2.RELEASE/spring-boot-cli-2.1.2.RELEASE-bin.zip

解压后目录如下



当然，为了方便你可以配置环境变量，但由于这个cli使用得并不频繁，老师就不配置了。

使用命令

.\spring.bat --version



可以查看Spring CLI的版本信息。

新建一个groovy的文件 hello.groovy

@RestController

class WebApplication {

@RequestMapping("/")

String home() {

"Hello World!"

}

}

使用命令可以启动

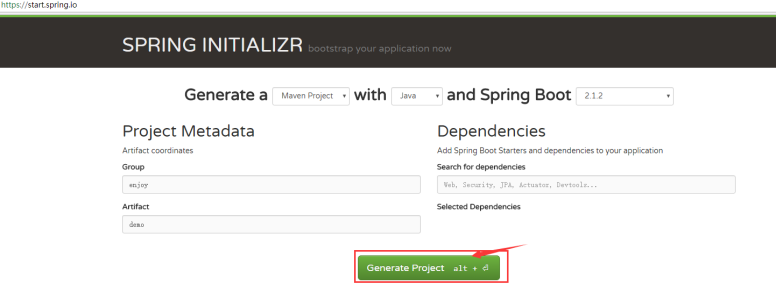
.\spring.bat run .\hello.groovy



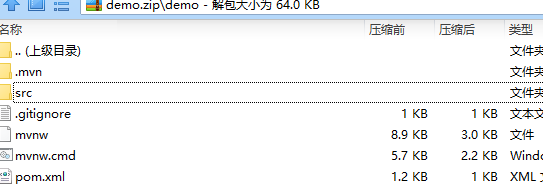
这个时候他会自动下载依赖，并运行项目。

### 构建项目

刚开始学习springboot的时候经常使用http://start.spring.io/ 构建项目



生成后



其实这个功能你完全也可以使用 SpringBoot CLI，完成构建

使用命令

./spring init --build=maven --java-version=1.8 --dependencies=web --packaging=jar --boot-version=1.5.3.RELEASE --groupId=enjoy --artifactId=demo

项目创建成功！

## 性能优化

### 扫描优化

在默认情况下，我们会使用@SpringBootApplication注解来自动获取应用的配置信息，但这样也会带来一些副作用。使用这个注解后，会触发自动配置（auto-configuration）和组件扫描（component scanning），这跟使用@Configuration、@EnableAutoConfiguration和@ComponentScan三个注解的作用是一样的。这样做给开发带来方便的同时，会有以下的一些影响：

* 会导致项目启动时间变长（原因：加载了我们不需要使用的组件,浪费了cpu资源和内存资源）。当启动一个大的应用程序，或将做大量的集成测试启动应用程序时，影响会特别明显。
* 会加载一些不需要的多余的实例（beans）。
* 会增加CPU消耗和内存的占用。

本着有问题就要解决的心态，针对以上的问题，我们要怎么解决呢？很明显，既然@SpringBootApplication加载了一些不必要的配置，那么我们想是否可以就加载我们自己指定的配置呢？

我们的思路不使用@SpringBootApplication，并且不使用@ComponentScan注解（此注解会自动扫描我们注解了@Controller，@Service的注解的类，加载到Spring IOC容器中），然后我们使用@Configuration和@EnableAutoConfiguration进行配置启动类

为了讲课方便，新建立一个项目

pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>enjoy</groupId>

<artifactId>springboot</artifactId>

<version>1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

</project>

这个pom文件非常简单。仅仅引入了web

新建一个测试controller

package cn.enjoy.controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class UserController {

@RequestMapping("/hi")

public Object sayHi() {

return "hi enjoy";

}

}

新增启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class,args);

}

}

这个代码非常简单，优化的效果也不会很明显，但重点同学们需要理解的是这原理

前面已经说过直接用@SpringBootApplication有各种问题。

那么现在改变启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

//@SpringBootApplication

@EnableAutoConfiguration

@Configuration

@ComponentScan("cn.enjoy.controller")

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class,args);

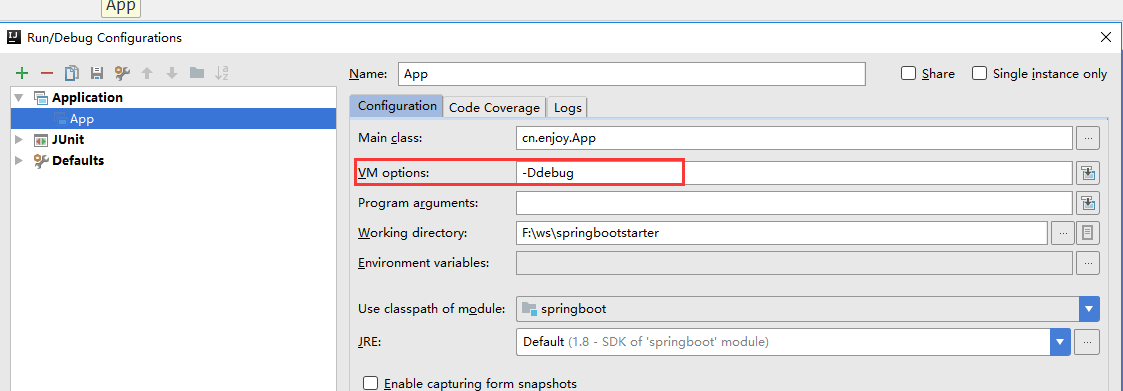
}

}

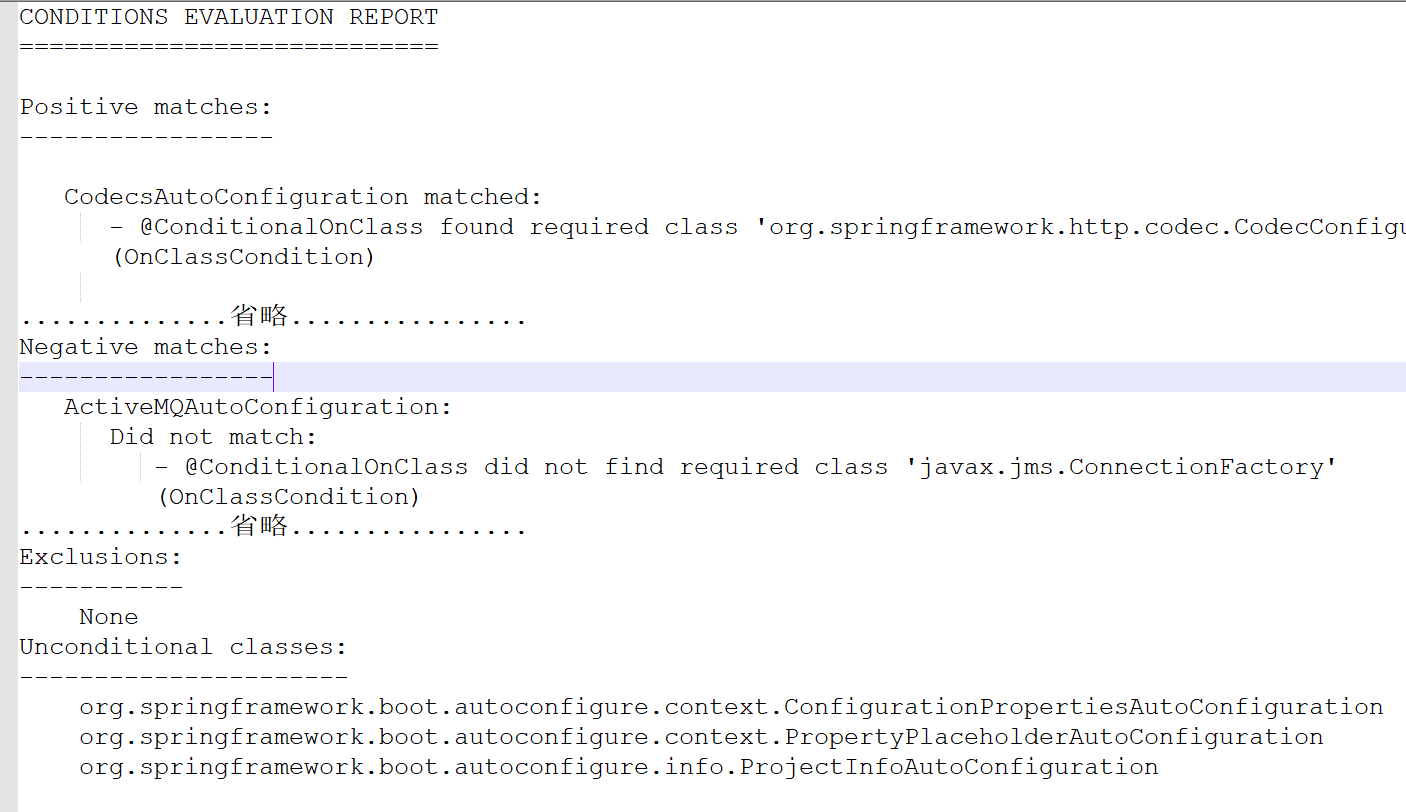
这样直接使用自己定义ComponentScan的扫描类，让他只扫描自己需要的组件能增加启动时间，并且介绍spring容器bean的数目。

[前面讲自定义starter的时候讲过，@EnableAutoConfiguration注解会导入META-INF/spring.factories里面配置的很多Configuration，这些Configuration他都会去扫描](mailto:前面讲自定义starter的时候讲过，@EnableAutoConfiguration注解会导入META-INF/spring.factories里面配置的很多Configuration，这些Configuration他都会去扫描)

在启动VM参数里面加入 -Ddebug



重新启动，发现在控制台



控制台输入的信息，大概分成四大类

1. Positive matches:

匹配（以及匹配的原因）

1. Negative matches:

忽略匹配（以及忽略的原因）

1. Exclusions:

排除的配置类

1. Unconditional classes:

没有带任何条件，肯定要扫描的类

  根据上面的理论知识，我们只需要在启动的时候，显式地引入这些组件

需要的组件=Positive matches+Unconditional classes

我们可以不使用@EnableAutoConfiguration，转而显示的使用@Import来导入需要的配置类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration;

import org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration;

import org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration;

import org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration;

import org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration;

import org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration;

import org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration;

import org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration;

import org.springframework.boot.autoconfigure.task.TaskSchedulingAutoConfiguration;

import org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration;

import org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration;

import org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoConfiguration;

import org.springframework.boot.autoconfigure.web.servlet.\*;

import org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration;

import org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAutoConfiguration;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Import;

@Configuration

//@EnableAutoConfiguration

@Import({

CodecsAutoConfiguration.class,

DispatcherServletAutoConfiguration.class,

EmbeddedWebServerFactoryCustomizerAutoConfiguration.class,

ErrorMvcAutoConfiguration.class,

HttpEncodingAutoConfiguration.class,

HttpMessageConvertersAutoConfiguration.class,

JacksonAutoConfiguration.class,

ServletWebServerFactoryAutoConfiguration.class,

WebMvcAutoConfiguration.class,

ValidationAutoConfiguration.class,

MultipartAutoConfiguration.class,

JmxAutoConfiguration.class,

RestTemplateAutoConfiguration.class,

WebSocketServletAutoConfiguration.class,

TaskExecutionAutoConfiguration.class,

TaskSchedulingAutoConfiguration.class,

ConfigurationPropertiesAutoConfiguration.class,

PropertyPlaceholderAutoConfiguration.class,

ProjectInfoAutoConfiguration.class

})

@ComponentScan("cn.enjoy.controller")

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class,args);

}

}

另外也可以删除一些虽然匹配到了，但是在项目中目前并没有使用到的配置，比如

任务调度：TaskExecutionAutoConfiguration，TaskSchedulingAutoConfiguration

WebSocket:WebSocketServletAutoConfiguration

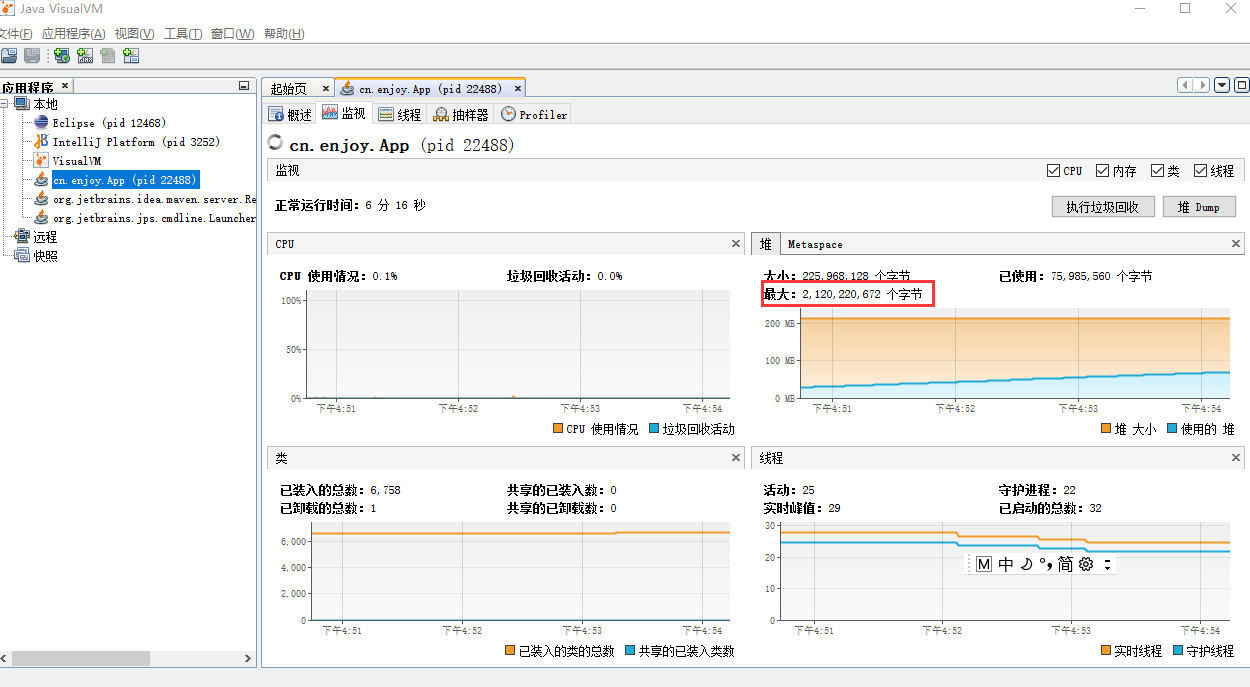
附件上传：MultipartAutoConfiguration

JMX：JmxAutoConfiguration

等等……

### JVM参数调优

启动App，使用jdk里面jvisualvm.exe



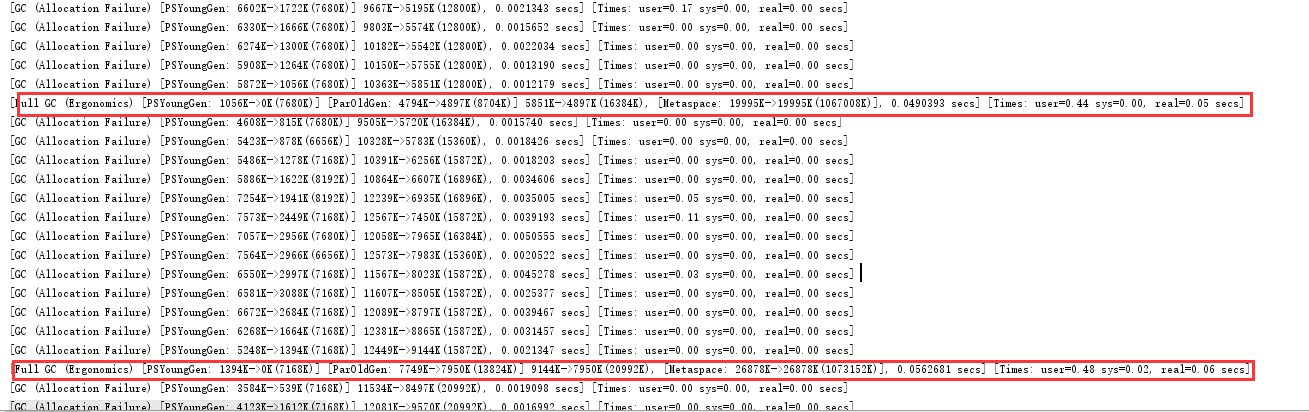
这个时候内存分配了2个G，可以根据需要，判断是否需要这么大，一般来说1个G足够，尤其是微服务

另外还发现最大值和最小值两个设置的并不一样，来看下会有什么问题。

设置JVM参数

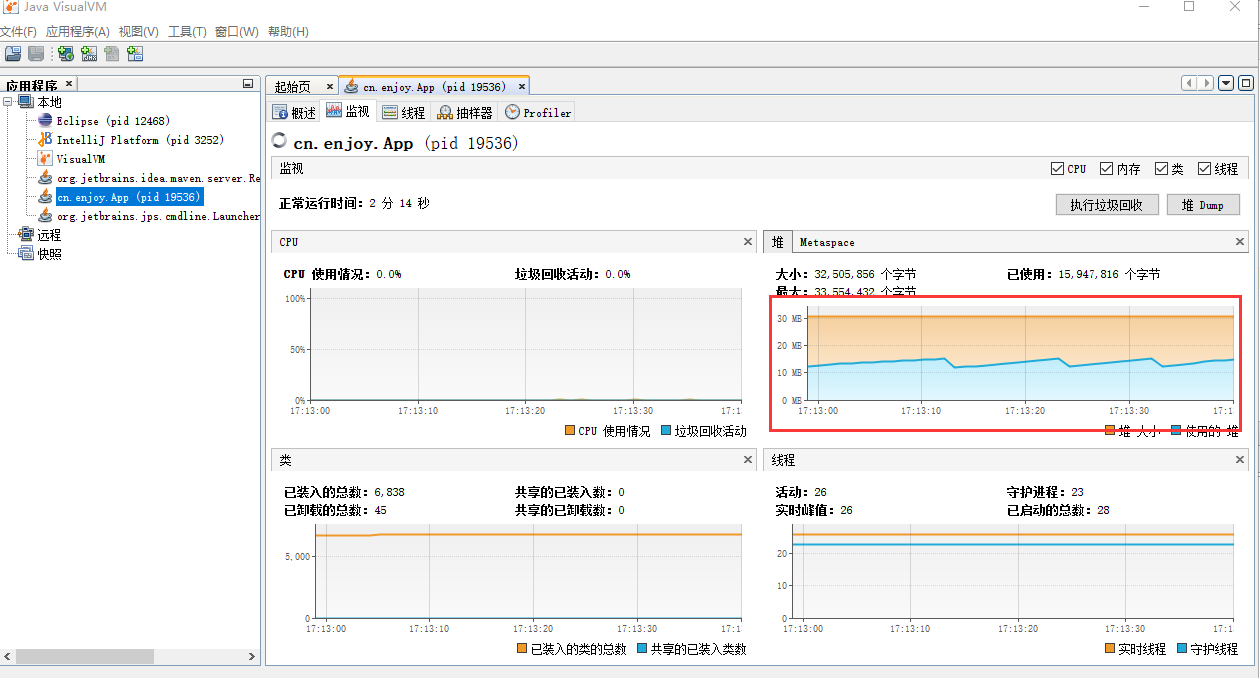
-XX:+PrintGCDetails -Xmx32M -Xms1M

这样设置后发现有大量的GC，更可怕的还有大量的FULL GC存在



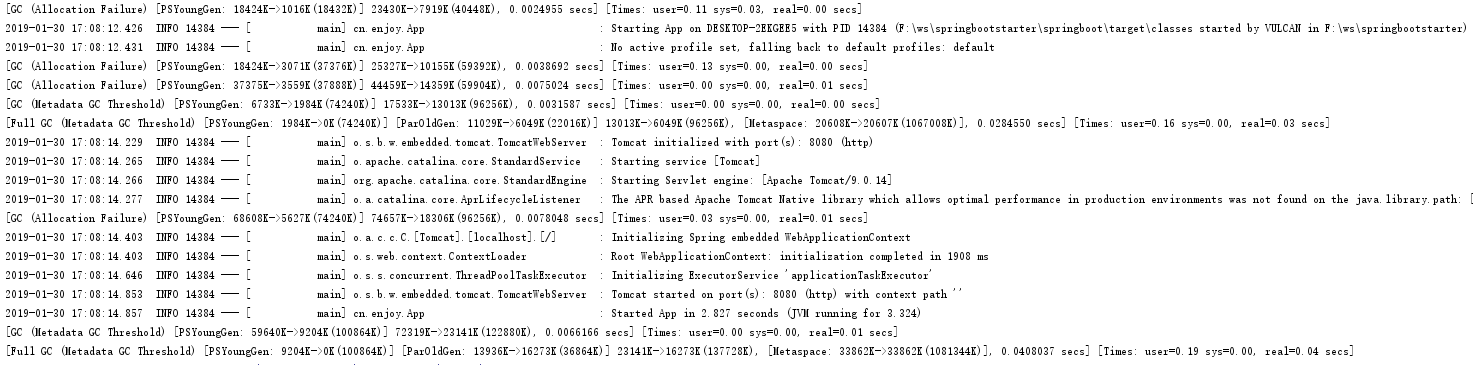
频繁的GC对性能影响是很大的。

频繁调用http://localhost:8080/hi，发现垃圾回收特别频繁



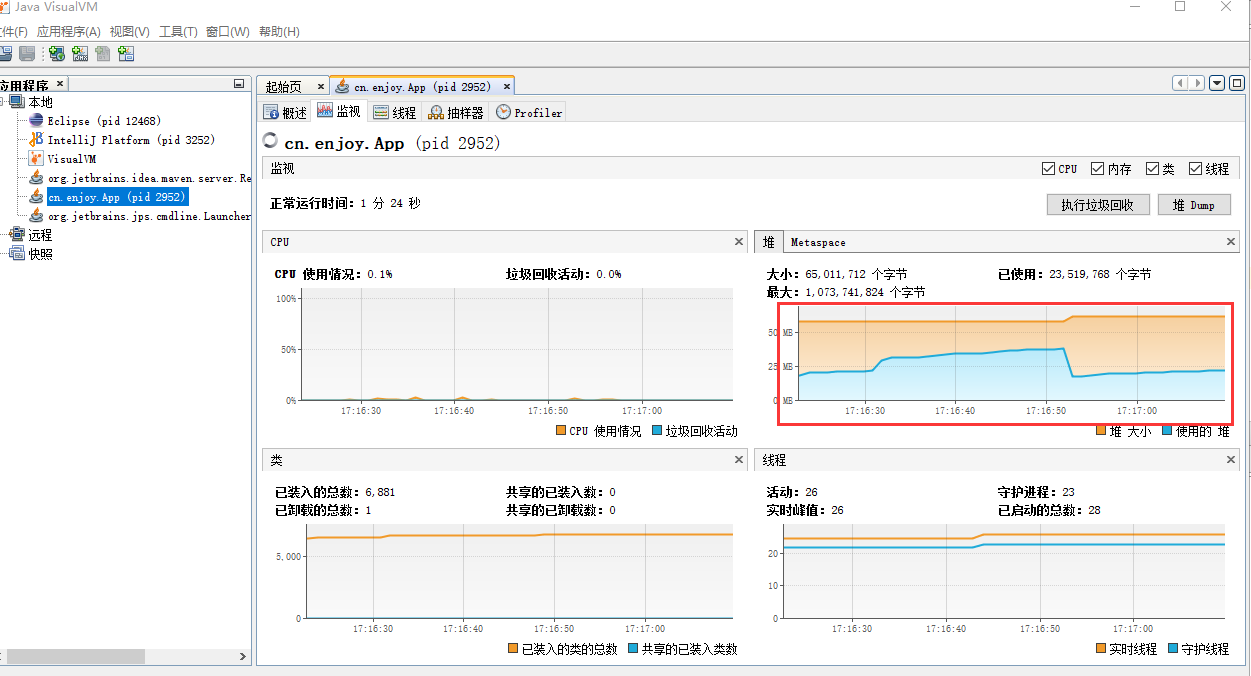
置JVM参数，把最大的内存数设置成1024

-XX:+PrintGCDetails -Xmx1024M -Xms1M



GC次数明显减少，但既然还会出现几个full GC

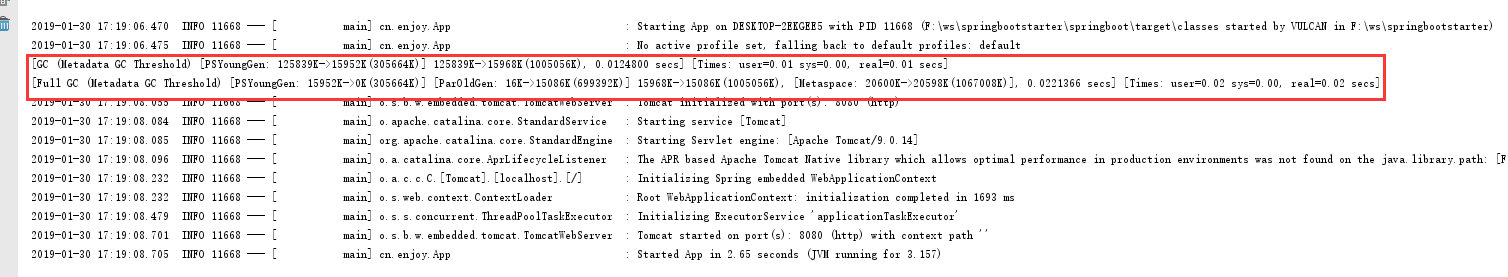
频繁调用http://localhost:8080/hi，发现垃圾回收依然特别频繁，这不断的申请内存，释放内存对性能是有不小的影响



置JVM参数，把最大的内存数设置成1024，最小内存也设置成1024

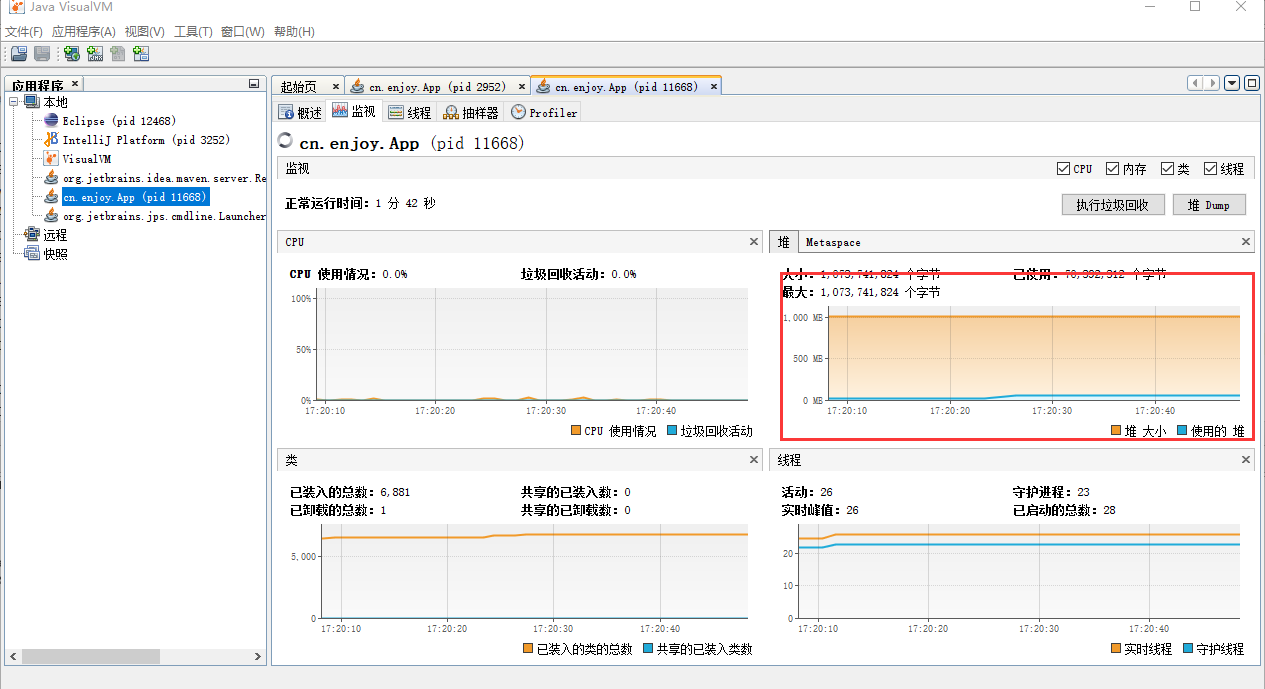
-XX:+PrintGCDetails -Xmx1024M -Xms1024M

这个时候再重启，发现



不管是gc还是full gc 都明显的减少了次数

这个时候再来频繁调用http://localhost:8080/hi，发现内存的使用比较均匀了



### Undertow容器

默认情况下，Spring Boot 使用 Tomcat 来作为内嵌的 Servlet 容器

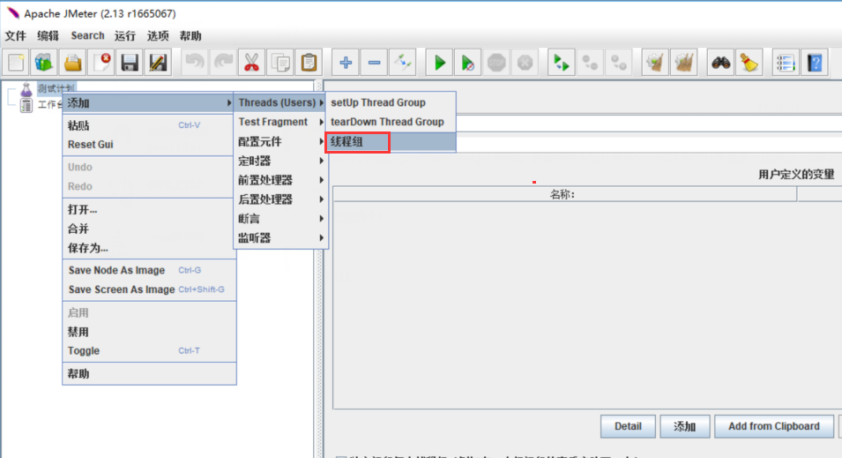
可以将 Web 服务器切换到 Undertow 来提高应用性能。

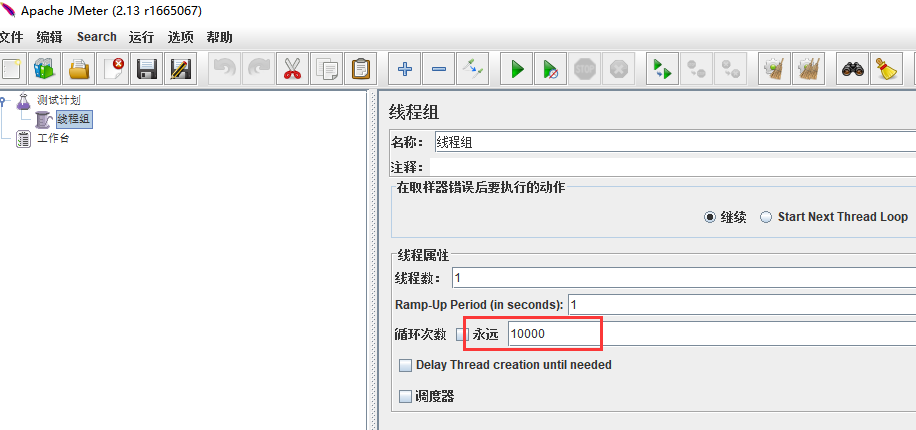
Undertow 是一个采用 Java 开发的灵活的高性能 Web 服务器，提供包括阻塞和基于 NIO 的非堵塞机制。Undertow 是红帽公司的开源产品

#### Tomcat测试

先测试下默认情况tomcat性能

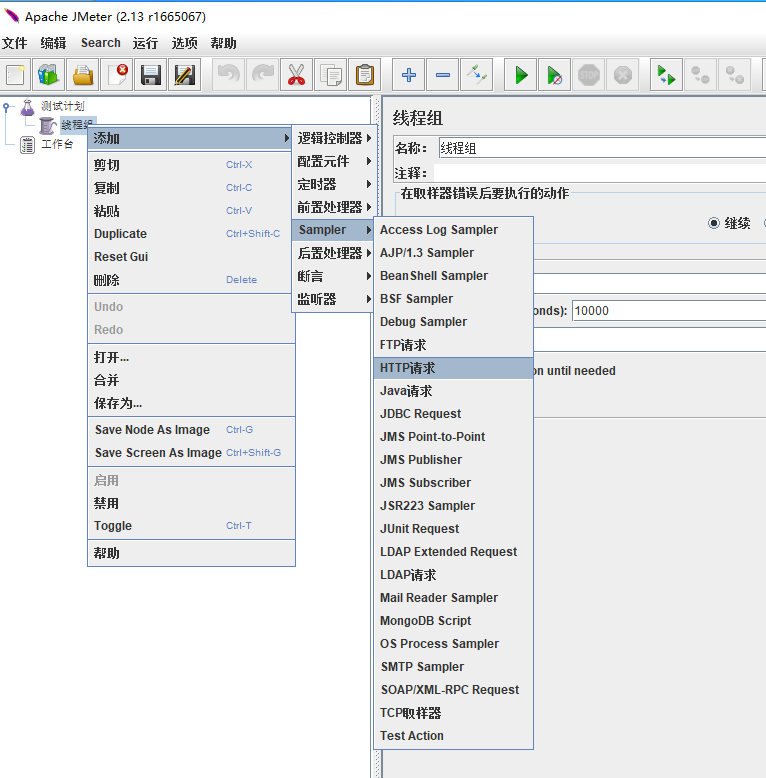
启动Apache JMeter



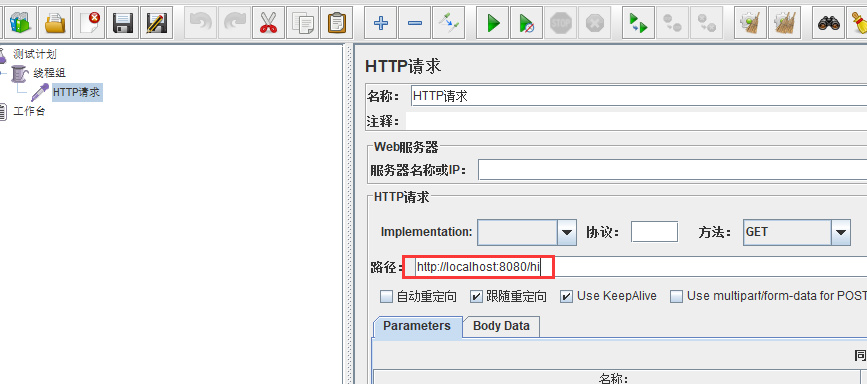


这里设置10000

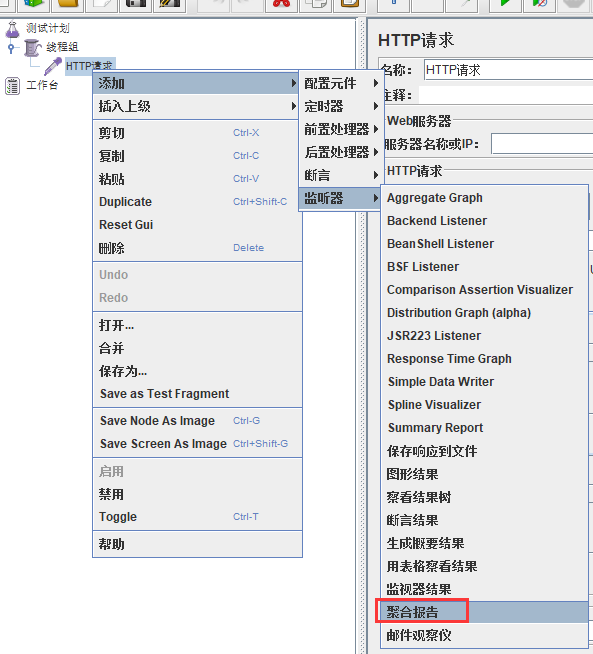
添加一个”http请求”



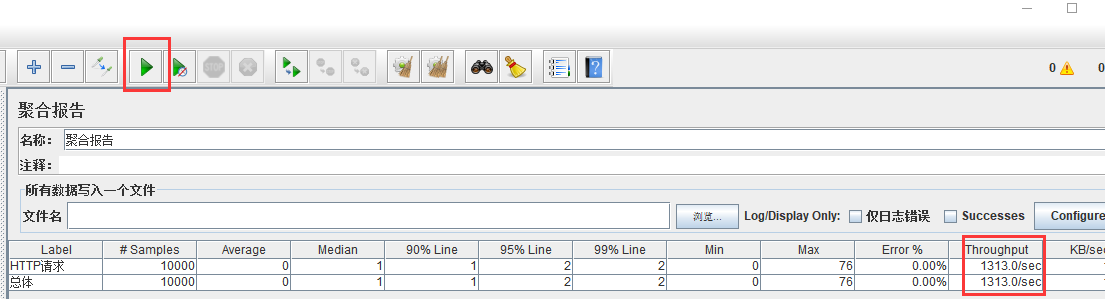
填写路径



添加一个“聚合报告”



重复测试3次



结果为：1313，1761，1720

可见tomcat在1秒内处理请求数不到2000

#### Undertow测试

修改pom文件

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>enjoy</groupId>

<artifactId>springboot</artifactId>

<version>1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

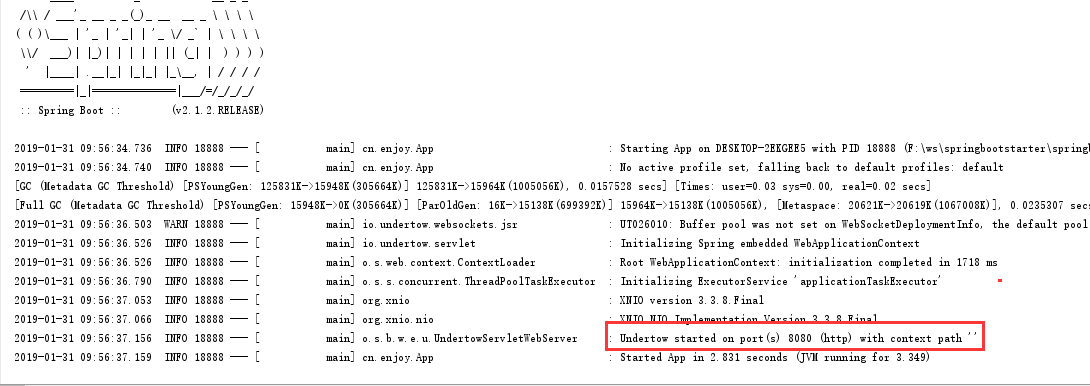
<artifactId>spring-boot-starter-undertow</artifactId>

</dependency>

</dependencies>

</project>

重新启动后发现容器以及切换成undertow



继续使用JMeter测试

1494，1972 ，2261

发现性能提升还是比较大的

## 多数据源与jta+atomikos分布式事务

SpringBoot默认是集成事务的，只要在方法上加上@Transactional既可

但某些项目用到了多个数据库，也就代表有多个数据源。

### 多数据源

#### 数据库

有两个数据库

数据库1：

Databases:spring

Table:enjoy\_user

**CREATE TABLE `enjoy\_user` (**

**`id` int NOT NULL AUTO\_INCREMENT ,**

**`passwd` varchar(255) NULL ,**

**`username` varchar(255) NULL ,**

**PRIMARY KEY (`id`)**

**);**

数据库2：

Databases:spring2

Table:enjoy\_order

**CREATE TABLE `enjoy\_order` (**

**`id` int(10) NOT NULL AUTO\_INCREMENT ,**

**`name` varchar(255) ,**

**`user\_id` int(11) NOT NULL ,**

**`account` int(255) NULL DEFAULT NULL ,**

**PRIMARY KEY (`id`)**

**)**

#### pom文件

新建一个项目，在pom文件里面增加mybatis等依赖信息

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId> enjoy</groupId>

<artifactId>springbootatomikos</artifactId>

<version>1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>1.3.0</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

</dependencies>

</project>

#### 新增model(Orders,Users)

package cn.enjoy.model;

public class Users {

private Integer id;

private String passwd;

private String username;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getPasswd() {

return passwd;

}

public void setPasswd(String passwd) {

this.passwd = passwd == null ? null : passwd.trim();

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username == null ? null : username.trim();

}

}

package cn.enjoy.model;

public class Orders {

private Integer id;

private String name;

private Integer userId;

private Integer account;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name == null ? null : name.trim();

}

public Integer getUserId() {

return userId;

}

public void setUserId(Integer userId) {

this.userId = userId;

}

public Integer getAccount() {

return account;

}

public void setAccount(Integer account) {

this.account = account;

}

}

#### 新增mapper接口

注意：由于这两mapper接口对应不同的数据库，为了配置与管理方便，不同的数据库的mapper接口单独放置在一个package中

package cn.enjoy.dao.orders;

import cn.enjoy.model.Orders;

public interface OrdersMapper {

int deleteByPrimaryKey(Integer id);

int insert(Orders record);

int insertSelective(Orders record);

Orders selectByPrimaryKey(Integer id);

int updateByPrimaryKeySelective(Orders record);

int updateByPrimaryKey(Orders record);

}

package cn.enjoy.dao.users;

import cn.enjoy.model.Users;

import org.apache.ibatis.annotations.Param;

public interface UsersMapper {

int deleteByPrimaryKey(Integer id);

int insert(Users record);

int insertSelective(Users record);

Users selectByPrimaryKey(Integer id);

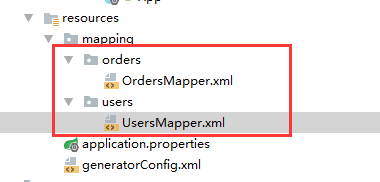
int updateByPrimaryKeySelective(Users record);

int updateByPrimaryKey(Users record);

}

#### Mapper的XML配置

为了管理方便，xml的配置也根据数据库的不同放置到不同的文件夹中。



<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >

<mapper namespace="cn.enjoy.dao.users.UsersMapper" >

<resultMap id="BaseResultMap" type="cn.enjoy.model.Users" >

<id column="id" property="id" jdbcType="INTEGER" />

<result column="passwd" property="passwd" jdbcType="VARCHAR" />

<result column="username" property="username" jdbcType="VARCHAR" />

</resultMap>

<sql id="Base\_Column\_List" >

id, passwd, username

</sql>

<select id="selectByPrimaryKey" resultMap="BaseResultMap" parameterType="java.lang.Integer" >

select

<include refid="Base\_Column\_List" />

from enjoy\_user

where id = #{id,jdbcType=INTEGER}

</select>

<delete id="deleteByPrimaryKey" parameterType="java.lang.Integer" >

delete from enjoy\_user

where id = #{id,jdbcType=INTEGER}

</delete>

<insert id="insert" parameterType="cn.enjoy.model.Users" >

insert into enjoy\_user (id, passwd, username

)

values (#{id,jdbcType=INTEGER}, #{passwd,jdbcType=VARCHAR}, #{username,jdbcType=VARCHAR}

)

</insert>

<insert id="insertSelective" parameterType="cn.enjoy.model.Users" >

insert into enjoy\_user

<trim prefix="(" suffix=")" suffixOverrides="," >

<if test="id != null" >

id,

</if>

<if test="passwd != null" >

passwd,

</if>

<if test="username != null" >

username,

</if>

</trim>

<trim prefix="values (" suffix=")" suffixOverrides="," >

<if test="id != null" >

#{id,jdbcType=INTEGER},

</if>

<if test="passwd != null" >

#{passwd,jdbcType=VARCHAR},

</if>

<if test="username != null" >

#{username,jdbcType=VARCHAR},

</if>

</trim>

</insert>

<update id="updateByPrimaryKeySelective" parameterType="cn.enjoy.model.Users" >

update enjoy\_user

<set >

<if test="passwd != null" >

passwd = #{passwd,jdbcType=VARCHAR},

</if>

<if test="username != null" >

username = #{username,jdbcType=VARCHAR},

</if>

</set>

where id = #{id,jdbcType=INTEGER}

</update>

<update id="updateByPrimaryKey" parameterType="cn.enjoy.model.Users" >

update enjoy\_user

set passwd = #{passwd,jdbcType=VARCHAR},

username = #{username,jdbcType=VARCHAR}

where id = #{id,jdbcType=INTEGER}

</update>

</mapper>

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >

<mapper namespace="cn.enjoy.dao.orders.OrdersMapper" >

<resultMap id="BaseResultMap" type="cn.enjoy.model.Orders" >

<id column="id" property="id" jdbcType="INTEGER" />

<result column="name" property="name" jdbcType="VARCHAR" />

<result column="user\_id" property="userId" jdbcType="INTEGER" />

<result column="account" property="account" jdbcType="INTEGER" />

</resultMap>

<sql id="Base\_Column\_List" >

id, name, user\_id, account

</sql>

<select id="selectByPrimaryKey" resultMap="BaseResultMap" parameterType="java.lang.Integer" >

select

<include refid="Base\_Column\_List" />

from enjoy\_order

where id = #{id,jdbcType=INTEGER}

</select>

<delete id="deleteByPrimaryKey" parameterType="java.lang.Integer" >

delete from enjoy\_order

where id = #{id,jdbcType=INTEGER}

</delete>

<insert id="insert" parameterType="cn.enjoy.model.Orders" >

insert into enjoy\_order (id, name, user\_id,

account)

values (#{id,jdbcType=INTEGER}, #{name,jdbcType=VARCHAR}, #{userId,jdbcType=INTEGER},

#{account,jdbcType=INTEGER})

</insert>

<insert id="insertSelective" parameterType="cn.enjoy.model.Orders" >

insert into enjoy\_order

<trim prefix="(" suffix=")" suffixOverrides="," >

<if test="id != null" >

id,

</if>

<if test="name != null" >

name,

</if>

<if test="userId != null" >

user\_id,

</if>

<if test="account != null" >

account,

</if>

</trim>

<trim prefix="values (" suffix=")" suffixOverrides="," >

<if test="id != null" >

#{id,jdbcType=INTEGER},

</if>

<if test="name != null" >

#{name,jdbcType=VARCHAR},

</if>

<if test="userId != null" >

#{userId,jdbcType=INTEGER},

</if>

<if test="account != null" >

#{account,jdbcType=INTEGER},

</if>

</trim>

</insert>

<update id="updateByPrimaryKeySelective" parameterType="cn.enjoy.model.Orders" >

update enjoy\_order

<set >

<if test="name != null" >

name = #{name,jdbcType=VARCHAR},

</if>

<if test="userId != null" >

user\_id = #{userId,jdbcType=INTEGER},

</if>

<if test="account != null" >

account = #{account,jdbcType=INTEGER},

</if>

</set>

where id = #{id,jdbcType=INTEGER}

</update>

<update id="updateByPrimaryKey" parameterType="cn.enjoy.model.Orders" >

update enjoy\_order

set name = #{name,jdbcType=VARCHAR},

user\_id = #{userId,jdbcType=INTEGER},

account = #{account,jdbcType=INTEGER}

where id = #{id,jdbcType=INTEGER}

</update>

</mapper>

#### 新建service接口与实现类

package cn.enjoy.service;

import cn.enjoy.model.Orders;

import cn.enjoy.model.Users;

public interface IOrderService {

void addOrder(Orders orders, Users users);

}

实现类

package cn.enjoy.service.impl;

import cn.enjoy.dao.orders.OrdersMapper;

import cn.enjoy.dao.users.UsersMapper;

import cn.enjoy.model.Orders;

import cn.enjoy.model.Users;

import cn.enjoy.service.IOrderService;

import org.springframework.stereotype.Service;

import javax.annotation.Resource;

@Service

public class OrderServiceImpl implements IOrderService{

@Resource

private UsersMapper usersMapper;

@Resource

private OrdersMapper ordersMapper;

@Override

public void addOrder(Orders orders, Users users) {

usersMapper.insertSelective(users);

ordersMapper.insertSelective(orders);

}

}

上面只是准备工作，目前未知没有配置与数据源相关的任何配置

#### application.properties

新增application.properties，里面配置数据源相关信息

分别配置了两个数据库

spring.datasource.spring.driverClassName=com.mysql.jdbc.Driver

spring.datasource.spring.jdbcUrl=jdbc:mysql://127.0.0.1:3306/spring?serverTimezone=GMT%2B8

spring.datasource.spring.username=root

spring.datasource.spring.password=root1234%

spring.datasource.spring2.driverClassName=com.mysql.jdbc.Driver

spring.datasource.spring2.jdbcUrl=jdbc:mysql://127.0.0.1:3306/spring2?serverTimezone=GMT%2B8

spring.datasource.spring2.username=root

spring.datasource.spring2.password=root1234%

#### 数据源配置类

package cn.enjoy.config;

import org.apache.ibatis.session.SqlSessionFactory;

import org.mybatis.spring.SqlSessionFactoryBean;

import org.mybatis.spring.SqlSessionTemplate;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.core.io.support.PathMatchingResourcePatternResolver;

import org.springframework.jdbc.datasource.DataSourceTransactionManager;

import javax.sql.DataSource;

@Configuration

@MapperScan(basePackages = "cn.enjoy.dao.users", sqlSessionFactoryRef = "test1SqlSessionFactory")

public class DataSource1Config {

@Bean(name = "test1DataSource")

@ConfigurationProperties(prefix = "spring.datasource.spring")

@Primary

public DataSource testDataSource() {

return DataSourceBuilder.create().build();

}

@Bean(name = "test1SqlSessionFactory")

@Primary

public SqlSessionFactory testSqlSessionFactory(@Qualifier("test1DataSource") DataSource dataSource)

throws Exception {

SqlSessionFactoryBean bean = new SqlSessionFactoryBean();

bean.setDataSource(dataSource);

bean.setMapperLocations(new PathMatchingResourcePatternResolver().getResources("classpath:mapping/users/\*.xml"));

return bean.getObject();

}

@Bean(name = "test1TransactionManager")

@Primary

public DataSourceTransactionManager testTransactionManager(@Qualifier("test1DataSource") DataSource dataSource) {

return new DataSourceTransactionManager(dataSource);

}

@Bean(name = "test1SqlSessionTemplate")

@Primary

public SqlSessionTemplate testSqlSessionTemplate(

@Qualifier("test1SqlSessionFactory") SqlSessionFactory sqlSessionFactory) throws Exception {

return new SqlSessionTemplate(sqlSessionFactory);

}

}

package cn.enjoy.config;

import org.apache.ibatis.session.SqlSessionFactory;

import org.mybatis.spring.SqlSessionFactoryBean;

import org.mybatis.spring.SqlSessionTemplate;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.core.io.support.PathMatchingResourcePatternResolver;

import org.springframework.jdbc.datasource.DataSourceTransactionManager;

import javax.sql.DataSource;

@Configuration

@MapperScan(basePackages = "cn.enjoy.dao.orders", sqlSessionFactoryRef = "test2SqlSessionFactory")

public class DataSource2Config {

@Bean(name = "test2DataSource")

@ConfigurationProperties(prefix = "spring.datasource.spring2")

public DataSource testDataSource() {

return DataSourceBuilder.create().build();

}

@Bean(name = "test2SqlSessionFactory")

public SqlSessionFactory testSqlSessionFactory(@Qualifier("test2DataSource") DataSource dataSource)

throws Exception {

SqlSessionFactoryBean bean = new SqlSessionFactoryBean();

bean.setDataSource(dataSource);

bean.setMapperLocations(new PathMatchingResourcePatternResolver().getResources("classpath:mapping/orders/\*.xml"));

return bean.getObject();

}

@Bean(name = "test2TransactionManager")

public DataSourceTransactionManager testTransactionManager(@Qualifier("test2DataSource") DataSource dataSource) {

return new DataSourceTransactionManager(dataSource);

}

@Bean(name = "test2SqlSessionTemplate")

public SqlSessionTemplate testSqlSessionTemplate(

@Qualifier("test2SqlSessionFactory") SqlSessionFactory sqlSessionFactory) throws Exception {

return new SqlSessionTemplate(sqlSessionFactory);

}

}

这两个配置文件是重中之重，配置了数据源，连接工厂，Mapper扫描的包， mapper xml配置的位置等

#### 单元测试

package enjoy.test;

import cn.enjoy.App;

import cn.enjoy.model.Orders;

import cn.enjoy.model.Users;

import cn.enjoy.service.IOrderService;

import org.junit.runner.RunWith;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringRunner;

import javax.annotation.Resource;

@SpringBootTest(classes = {App.class})

@RunWith(SpringRunner.class)

public class Test {

@Resource

private IOrderService iOrderService;

@org.junit.Test

public void test1() {

Users users = new Users();

users.setUsername("enjoy");

users.setPasswd("123");

users.setId(1);

Orders orders = new Orders();

orders.setAccount(12);

orders.setName("娃娃");

orders.setUserId(1);

iOrderService.addOrder(orders,users);

}

}

这个时候以及集成了2套数据源，并且已经测试发现可以同时入库。

### jta+atomikos分布式事务

#### 修改pom文件

增加atomikos支持

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jta-atomikos</artifactId>

</dependency>

#### 新增配置类（DBConfig1，DBConfig2）

package cn.enjoy.config;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "spring.datasource.spring")

@Component

public class DBConfig1 {

private String driverClassName;

private String jdbcUrl;

private String username;

private String password;

public String getDriverClassName() {

return driverClassName;

}

public void setDriverClassName(String driverClassName) {

this.driverClassName = driverClassName;

}

public String getJdbcUrl() {

return jdbcUrl;

}

public void setJdbcUrl(String jdbcUrl) {

this.jdbcUrl = jdbcUrl;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

}

package cn.enjoy.config;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "spring.datasource.spring2")

@Component

public class DBConfig2 {

private String driverClassName;

private String jdbcUrl;

private String username;

private String password;

public String getDriverClassName() {

return driverClassName;

}

public void setDriverClassName(String driverClassName) {

this.driverClassName = driverClassName;

}

public String getJdbcUrl() {

return jdbcUrl;

}

public void setJdbcUrl(String jdbcUrl) {

this.jdbcUrl = jdbcUrl;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

}

#### 修改数据源配置类

package cn.enjoy.config;

import com.mysql.cj.jdbc.MysqlXADataSource;

import org.apache.ibatis.session.SqlSessionFactory;

import org.mybatis.spring.SqlSessionFactoryBean;

import org.mybatis.spring.SqlSessionTemplate;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import org.springframework.boot.jta.atomikos.AtomikosDataSourceBean;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.core.io.support.PathMatchingResourcePatternResolver;

import org.springframework.jdbc.datasource.DataSourceTransactionManager;

import javax.sql.DataSource;

@Configuration

@MapperScan(basePackages = "cn.enjoy.dao.users", sqlSessionFactoryRef = "test1SqlSessionFactory")

public class DataSource1Config {

@Bean(name = "test1DataSource")

@Primary

public DataSource testDataSource(DBConfig1 config1) {

MysqlXADataSource mysqlXADataSource=new MysqlXADataSource();

mysqlXADataSource.setUrl(config1.getJdbcUrl());

mysqlXADataSource.setPassword(config1.getPassword());

mysqlXADataSource.setUser(config1.getUsername());

AtomikosDataSourceBean atomikosDataSourceBean=new AtomikosDataSourceBean();

atomikosDataSourceBean.setXaDataSource(mysqlXADataSource);

atomikosDataSourceBean.setUniqueResourceName("test1Datasource");

return atomikosDataSourceBean;

}

@Bean(name = "test1SqlSessionFactory")

@Primary

public SqlSessionFactory testSqlSessionFactory(@Qualifier("test1DataSource") DataSource dataSource)

throws Exception {

SqlSessionFactoryBean bean = new SqlSessionFactoryBean();

bean.setDataSource(dataSource);

bean.setMapperLocations(new PathMatchingResourcePatternResolver().getResources("classpath:mapping/users/\*.xml"));

return bean.getObject();

}

~~@Bean(name = "test1TransactionManager")~~

~~@Primary~~

~~public DataSourceTransactionManager testTransactionManager(@Qualifier("test1DataSource") DataSource dataSource) {~~

~~return new DataSourceTransactionManager(dataSource);~~

~~}~~

@Bean(name = "test1SqlSessionTemplate")

@Primary

public SqlSessionTemplate testSqlSessionTemplate(

@Qualifier("test1SqlSessionFactory") SqlSessionFactory sqlSessionFactory) throws Exception {

return new SqlSessionTemplate(sqlSessionFactory);

}

}

package cn.enjoy.config;

import com.mysql.cj.jdbc.MysqlXADataSource;

import org.apache.ibatis.session.SqlSessionFactory;

import org.mybatis.spring.SqlSessionFactoryBean;

import org.mybatis.spring.SqlSessionTemplate;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import org.springframework.boot.jta.atomikos.AtomikosDataSourceBean;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.core.io.support.PathMatchingResourcePatternResolver;

import org.springframework.jdbc.datasource.DataSourceTransactionManager;

import javax.sql.DataSource;

@Configuration

@MapperScan(basePackages = "cn.enjoy.dao.orders", sqlSessionFactoryRef = "test2SqlSessionFactory")

public class DataSource2Config {

@Bean(name = "test2DataSource")

public DataSource testDataSource(DBConfig2 config2) {

MysqlXADataSource mysqlXADataSource=new MysqlXADataSource();

mysqlXADataSource.setUrl(config2.getJdbcUrl());

mysqlXADataSource.setPassword(config2.getPassword());

mysqlXADataSource.setUser(config2.getUsername());

AtomikosDataSourceBean atomikosDataSourceBean=new AtomikosDataSourceBean();

atomikosDataSourceBean.setXaDataSource(mysqlXADataSource);

atomikosDataSourceBean.setUniqueResourceName("test2Datasource");

return atomikosDataSourceBean;

}

@Bean(name = "test2SqlSessionFactory")

public SqlSessionFactory testSqlSessionFactory(@Qualifier("test2DataSource") DataSource dataSource)

throws Exception {

SqlSessionFactoryBean bean = new SqlSessionFactoryBean();

bean.setDataSource(dataSource);

bean.setMapperLocations(new PathMatchingResourcePatternResolver().getResources("classpath:mapping/orders/\*.xml"));

return bean.getObject();

}

~~@Bean(name = "test2TransactionManager")~~

~~public DataSourceTransactionManager testTransactionManager(@Qualifier("test2DataSource") DataSource dataSource) {~~

~~return new DataSourceTransactionManager(dataSource);~~

~~}~~

@Bean(name = "test2SqlSessionTemplate")

public SqlSessionTemplate testSqlSessionTemplate(

@Qualifier("test2SqlSessionFactory") SqlSessionFactory sqlSessionFactory) throws Exception {

return new SqlSessionTemplate(sqlSessionFactory);

}

}

#### 修改Service方法

package cn.enjoy.service.impl;

import cn.enjoy.dao.orders.OrdersMapper;

import cn.enjoy.dao.users.UsersMapper;

import cn.enjoy.model.Orders;

import cn.enjoy.model.Users;

import cn.enjoy.service.IOrderService;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import javax.annotation.Resource;

@Service

public class OrderServiceImpl implements IOrderService{

@Resource

private UsersMapper usersMapper;

@Resource

private OrdersMapper ordersMapper;

@Override

@Transactional

public void addOrder(Orders orders, Users users) {

usersMapper.insertSelective(users);

int i=10/0;

ordersMapper.insertSelective(orders);

}

}

使用之前的单元测试

# 实战核心原理

对于springboot来说，还有两块是比较有意思的，第一就是发现他内置了tomcat,接下来一快就是他对springmvc进行了无缝整合。我们也可以来试试。

## 内嵌tomcat

首先来看下最简单的tomcat集成。

新建一个项目，pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>enjoy</groupId>

<artifactId>java\_tomcat</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<!--Java语言操作tomcat -->

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-core</artifactId>

<version>8.5.16</version>

</dependency>

<!-- tomcat对jsp支持 -->

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-jasper</artifactId>

<version>8.5.16</version>

</dependency>

</dependencies>

</project>

新建立一个servlet

package cn.enjoy.servlet;

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class IndexServlet extends HttpServlet {

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

doPost(req, resp);

}

@Override

protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

resp.getWriter().print("this is index... tomcat");

}

}

有了这个servlet,就可以使用tomcat 整合了。

package cn.enjoy.tomcat;

import cn.enjoy.servlet.IndexServlet;

import org.apache.catalina.LifecycleException;

import org.apache.catalina.core.StandardContext;

import org.apache.catalina.startup.Tomcat;

public class DeerTomcat {

private static int PORT = 8080;

private static String CONTEX\_PATH = "/enjoy";

private static String SERVLET\_NAME = "indexServlet";

public static void main(String[] args) throws LifecycleException, InterruptedException {

// 创建tomcat服务器

Tomcat tomcatServer = new Tomcat();

// 指定端口号

tomcatServer.setPort(PORT);

// 是否设置自动部署

tomcatServer.getHost().setAutoDeploy(false);

// 创建上下文

StandardContext standardContex = new StandardContext();

standardContex.setPath(CONTEX\_PATH);

// 监听上下文

standardContex.addLifecycleListener(new Tomcat.FixContextListener());

// tomcat容器添加standardContex

tomcatServer.getHost().addChild(standardContex);

// 创建Servlet

tomcatServer.addServlet(CONTEX\_PATH, SERVLET\_NAME, new IndexServlet());

// servleturl映射

standardContex.addServletMappingDecoded("/index", SERVLET\_NAME);

tomcatServer.start();

System.out.println("tomcat服务器启动成功..");

// 异步进行接收请求

tomcatServer.getServer().await();

}

}

运行main方法，在浏览器输入：

http://localhost:8080/enjoy/index

## SpringMVC整合

### pom文件

新建项目，pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>enjoy</groupId>

<artifactId>springboot\_mvc</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<!--Java语言操作tomcat -->

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-core</artifactId>

<version>8.5.16</version>

</dependency>

<!-- tomcat对jsp支持 -->

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-jasper</artifactId>

<version>8.5.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>5.0.4.RELEASE</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.0.4.RELEASE</version>

<scope>compile</scope>

</dependency>

</dependencies>

</project>

### 新增UserService

package cn.enjoy.service;

import org.springframework.stereotype.Service;

@Service

public class UserService {

public String index() {

return "springboot 2.0 我正在加载UserService";

}

}

### 新建RestController

package cn.enjoy.controller;

import cn.enjoy.service.UserService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class IndexController {

@Autowired

private UserService userService;

@RequestMapping(value = "/index", produces = "text/html;charset=UTF-8")

public String index() {

return userService.index();

}

}

这样，一些常用业务类已经准备完毕，接下来就是真正的整合了。

### DispatcherServlet配置类

虽然前面有了service,有了controller,但依然没有把这些组建交给spring,对于springmvc来说，有个DispatcherServlet，这是springmvc的前端控制器，以前是配置在web.xml中。只是现在用的是注解。

package cn.enjoy.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

@Configuration

public class SpringWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

// 加载根配置信息 spring核心

protected Class<?>[] getRootConfigClasses() {

return new Class[] { RootConfig.class };

}

// springmvc 加载 配置信息

protected Class<?>[] getServletConfigClasses() {

return new Class[] { WebConfig.class };

}

// springmvc 拦截url映射 拦截所有请求

protected String[] getServletMappings() {

return new String[] { "/" };

}

}

package cn.enjoy.config;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("cn.enjoy")

public class RootConfig {

}

package cn.enjoy.config;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = { "cn.enjoy.controller" })

//public class WebConfig extends WebMvcConfigurerAdapter {

public class WebConfig implements WebMvcConfigurer {

}

### 集成tomcat

package cn.enjoy;

import org.apache.catalina.LifecycleException;

import org.apache.catalina.WebResourceRoot;

import org.apache.catalina.core.StandardContext;

import org.apache.catalina.startup.Tomcat;

import org.apache.catalina.webresources.DirResourceSet;

import org.apache.catalina.webresources.StandardRoot;

import javax.servlet.ServletException;

import java.io.File;

public class TomcatApp {

public static void main(String[] args) throws ServletException, LifecycleException {

start();

}

public static void start() throws ServletException, LifecycleException {

// 创建Tomcat容器

Tomcat tomcatServer = new Tomcat();

// 端口号设置

tomcatServer.setPort(9090);

// 读取项目路径 加载静态资源

StandardContext ctx = (StandardContext) tomcatServer.addWebapp("/", new File("springboot\_mvc/src/main").getAbsolutePath());

// 禁止重新载入

ctx.setReloadable(false);

// class文件读取地址

File additionWebInfClasses = new File("target/classes");

// 创建WebRoot

WebResourceRoot resources = new StandardRoot(ctx);

// tomcat内部读取Class执行

resources.addPreResources(

new DirResourceSet(resources, "/WEB-INF/classes", additionWebInfClasses.getAbsolutePath(), "/"));

tomcatServer.start();

// 异步等待请求执行

tomcatServer.getServer().await();

}

}

启动，在地址栏输入：

[http://localhost:9090/index](http://localhost:9090/index，发现能访问了)

### 对JSP支持

要支持JSP，回忆学习springmvc中的内容，需要用到一个试图解析器

#### 修改WebConfig

增加对jsp的支持

package cn.enjoy.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.ViewResolver;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.ViewResolverRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = { "cn.enjoy.controller" })

//public class WebConfig extends WebMvcConfigurerAdapter {

public class WebConfig implements WebMvcConfigurer {

// 创建SpringMVC视图解析器

public void configureViewResolvers(ViewResolverRegistry registry) {

InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

viewResolver.setPrefix("/WEB-INF/views/");

viewResolver.setSuffix(".jsp");

// 可以在JSP页面中通过${}访问beans

viewResolver.setExposeContextBeansAsAttributes(true);

registry.viewResolver(viewResolver);

}

}

#### 新增controller

package cn.enjoy.controller;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

@Controller

public class UserController {

@RequestMapping("/pageIndex")

public String pageIndex() {

return "pageIndex";

}

}

#### 增加JSP

在resources里面新增WEB-INF\views\pageIndex.jsp

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>享学课堂</title>

</head>

<body>

<h1>这是个jsp页面</h1>

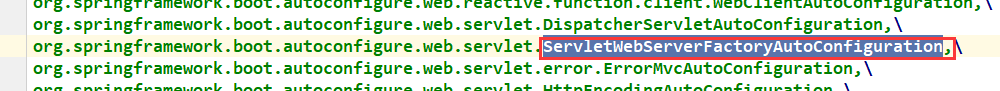
</body>

</html>

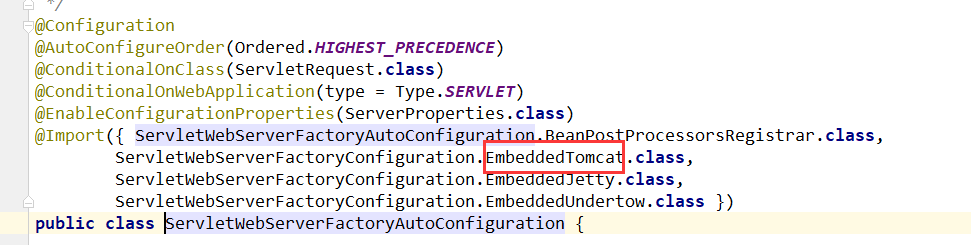
重启tomcat ，访问http://localhost:9090/pageIndex

## Tomcat加载流程

问题1：tomcat哪启动的

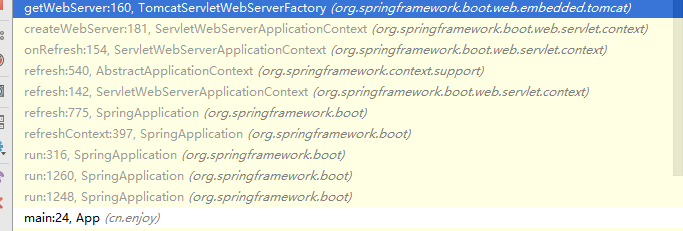


ServletWebServerFactoryAutoConfiguration 这类里面有个TomcatServletWebServerFactoryCustomizer这个类实现了WebServerFactoryCustomizer



EmbeddedTomcat->TomcatServletWebServerFactory->TomcatServletWebServerFactory.getWebServer()->getTomcatWebServer->TomcatWebServer->启动tomcat

问题2：getWebServer谁调用的。



## SpringBoot启动流程分析

### 准备工作

ApplicationContextInitializer Context初始化后调用的类

SpringApplicationRunListener SpringBoot运行监听的类

ApplicationRunner

CommandLineRunner

上面着两个几乎可以等价，用于启动后做客户自定义的操作

新建java类EnjoyCommandLineRunner

package cn.enjoy.listener;

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

import java.util.Arrays;

@Component

public class EnjoyCommandLineRunner implements CommandLineRunner {

@Override

public void run(String... args) throws Exception {

System.out.println("EnjoyCommandLineRunner.run()执行了"+ Arrays.asList(args));

}

}

新建java类EnjoyApplicationRunner

package cn.enjoy.listener;

import org.springframework.boot.ApplicationArguments;

import org.springframework.boot.ApplicationRunner;

import org.springframework.stereotype.Component;

@Component

public class EnjoyApplicationRunner implements ApplicationRunner {

@Override

public void run(ApplicationArguments args) throws Exception {

System.out.println("EnjoyApplicationRunner.run()执行了");

}

}

新建java类EnjoyApplicationContextInitializer

package cn.enjoy.listener;

import org.springframework.context.ApplicationContextInitializer;

import org.springframework.context.ConfigurableApplicationContext;

public class EnjoyApplicationContextInitializer

implements ApplicationContextInitializer<ConfigurableApplicationContext> {

@Override

public void initialize(ConfigurableApplicationContext applicationContext) {

System.out.println("EnjoyApplicationContextInitializer.initialize()执行了"+applicationContext);

}

}

新建java类EnjoySpringApplicationRunListener

package cn.enjoy.listener;

import org.springframework.boot.SpringApplicationRunListener;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.core.env.ConfigurableEnvironment;

public class EnjoySpringApplicationRunListener implements SpringApplicationRunListener {

//必须有的构造器

public EnjoySpringApplicationRunListener (SpringApplication application, String[] args){

}

@Override

public void starting() {

System.out.println("EnjoySpringApplicationRunListener.starting()执行了");

}

@Override

public void environmentPrepared(ConfigurableEnvironment environment) {

System.out.println("EnjoySpringApplicationRunListener.environmentPrepared()执行了");

}

@Override

public void contextPrepared(ConfigurableApplicationContext context) {

System.out.println("EnjoySpringApplicationRunListener.contextPrepared()执行了");

}

@Override

public void contextLoaded(ConfigurableApplicationContext context) {

System.out.println("EnjoySpringApplicationRunListener.contextLoaded()执行了");

}

@Override

public void started(ConfigurableApplicationContext context) {

System.out.println("EnjoySpringApplicationRunListener.started()执行了");

}

@Override

public void running(ConfigurableApplicationContext context) {

System.out.println("EnjoySpringApplicationRunListener.running()执行了");

}

@Override

public void failed(ConfigurableApplicationContext context, Throwable exception) {

System.out.println("EnjoySpringApplicationRunListener.failed()执行了");

}

}

在resources/META-INF/spring.factories增加

org.springframework.context.ApplicationContextInitializer=\

cn.enjoy.listener.EnjoyApplicationContextInitializer

org.springframework.boot.SpringApplicationRunListener=\

cn.enjoy.listener.EnjoySpringApplicationRunListener

为什么如此，后面源代码分析

### 创建SpringApplication

public static ConfigurableApplicationContext run(Class<?>[] primarySources,

String[] args) {

return new SpringApplication(primarySources).run(args);

}

其实SpringBoot启动就着两个步骤，先创建ConfigurableApplicationContext ，然后再调用Run方法。

public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {

this.resourceLoader = resourceLoader;

Assert.notNull(primarySources, "PrimarySources must not be null");

//保存主类

this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));

//判断当前是什么类型项目

this.webApplicationType = WebApplicationType.deduceFromClasspath();

//从类路径下找到META-INF/spring.factories配置的所有ApplicationContextInitializer

setInitializers((Collection) getSpringFactoriesInstances(

ApplicationContextInitializer.class));

//从类路径下找到META-INF/spring.factories配置的所有ApplicationListener

setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));

this.mainApplicationClass = deduceMainApplicationClass();

}

可见着一步非常简单，只是把一些相关的类都加载了而已，并没执行。

### Run方法

真的重要是run方法

public ConfigurableApplicationContext run(String... args) {

StopWatch stopWatch = new StopWatch();

stopWatch.start();

ConfigurableApplicationContext context = null;

Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();

configureHeadlessProperty();

//从类路径下META‐INF/spring.factories，取得SpringApplicationRunListeners；

SpringApplicationRunListeners listeners = getRunListeners(args);

//回调所有的获取SpringApplicationRunListener.starting()方法

listeners.starting();

try {

//封装命令行参数

ApplicationArguments applicationArguments = new DefaultApplicationArguments(

args);

//准备环境

ConfigurableEnvironment environment = prepareEnvironment(listeners,

applicationArguments);

configureIgnoreBeanInfo(environment);

//创回调SpringApplicationRunListener.environmentPrepared()；

//表示环境准备完成

//打印Banner

Banner printedBanner = printBanner(environment);

//根据环境创建context

context = createApplicationContext();

//错误的异常报表

exceptionReporters = getSpringFactoriesInstances(

SpringBootExceptionReporter.class,

new Class[] { ConfigurableApplicationContext.class }, context);

//准备上下文环境;

//将environment保存到ioc中；

//applyInitializers()调用所有的ApplicationContextInitializer的initialize方法

//调用所有的SpringApplicationRunListener的contextPrepared()；

prepareContext(context, environment, listeners, applicationArguments,

printedBanner);

//SpringApplicationRunListener的contextLoaded

//刷新容器

//扫描，创建，加载所有组件；

refreshContext(context);

afterRefresh(context, applicationArguments);

stopWatch.stop();

if (this.logStartupInfo) {

new StartupInfoLogger(this.mainApplicationClass)

.logStarted(getApplicationLog(), stopWatch);

}

//所有的SpringApplicationRunListener回调started方法

listeners.started(context);

//获取所有的ApplicationRunner和CommandLineRunner进行调用

callRunners(context, applicationArguments);

}

catch (Throwable ex) {

handleRunFailure(context, ex, exceptionReporters, listeners);

throw new IllegalStateException(ex);

}

try {

//所有的SpringApplicationRunListener的running()；

listeners.running(context);

}

catch (Throwable ex) {

handleRunFailure(context, ex, exceptionReporters, null);

throw new IllegalStateException(ex);

}

return context;

}