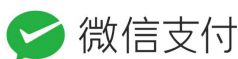


制作不易、如果觉的好不妨打个赏:



一、SpringBoot简介

1、Spring的优缺点

Spring的优点

Spring是Java企业版 (Java Enterprise Edition, JEE, 也称J2EE) 的轻量级代替品。无需开发重量级的EnterpriseJavaBean (EJB), Spring为企业级Java开发提供了一种相对简单的方法, 通过依赖注入和面向切面编程, 用简单的Java对象 (Plain Old Java Object, POJO) 实现了EJB的功能。

1. 使用Spring的IOC容器,将对象之间的依赖关系交给Spring,降低组件之间的耦合性,让我们更专注于应用逻辑
2. 可以提供众多服务,事务管理,WS等。
3. AOP的很好支持,方便面向切面编程。
4. 对主流的框架提供了很好的集成支持,如Hibernate,Struts2,JPA等
5. Spring DI机制降低了业务对象替换的复杂性。
6. Spring属于低侵入,代码污染极低。
7. Spring的高度可开放性,并不强制依赖于Spring,开发者可以自由选择Spring部分或全部

Spring的缺点

虽然Spring的组件代码是轻量级的,但它的配置却是重量级的。

一开始, Spring用XML配置, 而且是很多XML配置。Spring 2.5引入了基于注解的组件扫描, 这消除了大量针对应用程序自身组件的显式XML配置。Spring 3.0引入了基于Java的配置, 这是一种类型安全的可重构配置方式, 可以代替XML。

所有这些配置都代表了开发时的损耗。因为在思考Spring特性配置和解决业务问题之间需要进行思维切换, 所以编写配置挤占了编写应用程序逻辑的时间。和所有框架一样, Spring实用, 但与此同时它要求的回报也不少。

除此之外，项目的依赖管理也是一件耗时耗力的事情。在环境搭建时，需要分析要导入哪些库的坐标，而且还需要分析导入与之有依赖关系的其他库的坐标，一旦选错了依赖的版本，随之而来的不兼容问题就会严重阻碍项目的开发进度。

1. jsp中要写很多代码、控制器过于灵活,缺少一个公用控制器。
2. Spring不支持分布式,这也是EJB仍然在用的原因之一。

2、SpringBoot的概述

(1) SpringBoot解决上述Spring的缺点

SpringBoot对上述Spring的缺点进行的改善和优化，**基于约定优于配置**的思想，可以让开发人员不必在配置与逻辑业务之间进行思维的切换，全身心的投入到逻辑业务的代码编写中，从而大大提高了开发的效率，一定程度上缩短了项目周期。

(2) SpringBoot的特点

- 为基于Spring的开发提供更快的入门体验
- **开箱即用**，没有代码生成，也无需XML配置。同时也可以修改默认值来满足特定的需求
- 提供了一些大型项目中常见的非功能性特性，如嵌入式服务器、安全、指标、健康检测、外部配置等
- SpringBoot不是对Spring功能上的增强，而是提供了一种快速使用Spring的方式

(3) SpringBoot的核心功能

- **起步依赖** 起步依赖本质上是一个Maven项目对象模型（Project Object Model，POM），定义了对其他库的传递依赖，这些东西加在一起即支持某项功能。简单的说，起步依赖就是将具备某种功能的坐标打包到一起，并提供一些默认的功能。
- **自动配置** Spring Boot的自动配置是一个运行时（更准确地说，是应用程序启动时）的过程，考虑了众多因素，才决定Spring配置应该用哪个，不该用哪个。该过程是Spring自动完成的。

二、SpringBoot快速入门

1、体验

spring的官方文档为我们提供了quick、start

我们看看spring官网给我们提供了什么方式

Project

☒ Maven Project

☐ Gradle Project

Language

☒ Java

☐ Kotlin

☐ Groovy

Spring Boot

☐ 2.3.1 (SNAPSHOT)

☒ 2.3.0

☐ 2.2.8 (SNAPSHOT)

☐ 2.2.7

☐ 2.1.15 (SNAPSHOT)

☐ 2.1.14

Project Metadata

Group

com.example

Artifact

demo4

Name

demo4

Description

Demo project for Spring Boot

Package name

com.example.demo4

Packaging

☒ Jar

☐ War

Java

☐ 14

☐ 11

☒ 8

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

GENERATE CTRL + G

EXPLORE CTRL + SPACE

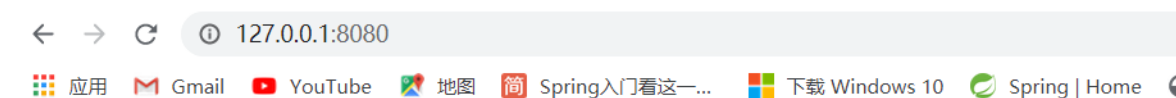
SHARE...

mvn spring-boot:run 运行

c) 2019 Microsoft Corporation。保留所有权利。

```
: \Users\zn\Downloads\demo4>mvn spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.example:demo4 >-----
[INFO] Building demo4 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot-maven-plugin:2.3.0.RELEASE:run (default-cli) @ demo4 >>>
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ demo4 ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ demo4 ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\Users\zn\Downloads\demo4\target\classes
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @ demo4 ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\Users\zn\Downloads\demo4\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ demo4 ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\Users\zn\Downloads\demo4\target\test-classes
[INFO]
```

测试，报错，说明成功了，应为没写controller



Whitelabel Error Page

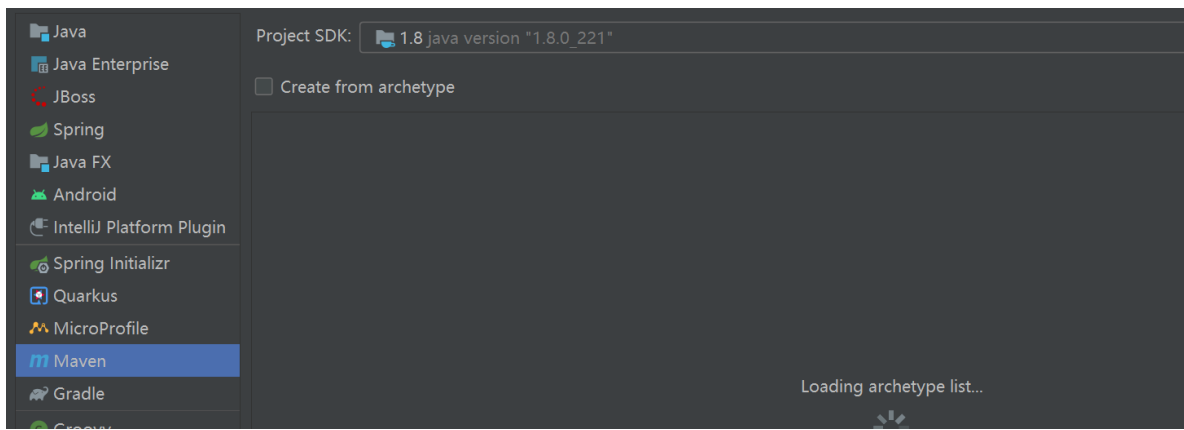
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Jun 02 16:51:45 CST 2020

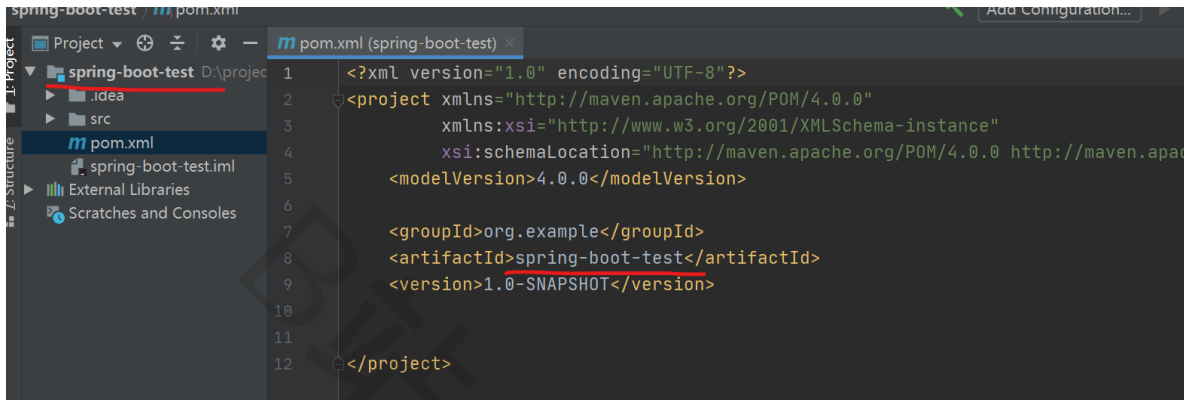
There was an unexpected error (type=Not Found, status=404).

2、idea创建工程

1、普通maven工程创建



工程名spring-boot-test



2、添加依赖

```
<?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>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-parent</artifactId>
    <version>2.2.2.RELEASE</version>
  </parent>

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

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
```

```
</project>
```

编写启动引导类

```
package com.xinzhi;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * @author IT楠老师
 * @date 2020/6/2
 */
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class);
    }
}
```

写一个controller

```
/**
 * @author IT楠老师
 * @date 2020/6/2
 */
@Controller
public class UserController {

    @RequestMapping("/")
    @ResponseBody
    public String test(){
        return "hello springboot";
    }

}
```

运行引导类的main方法

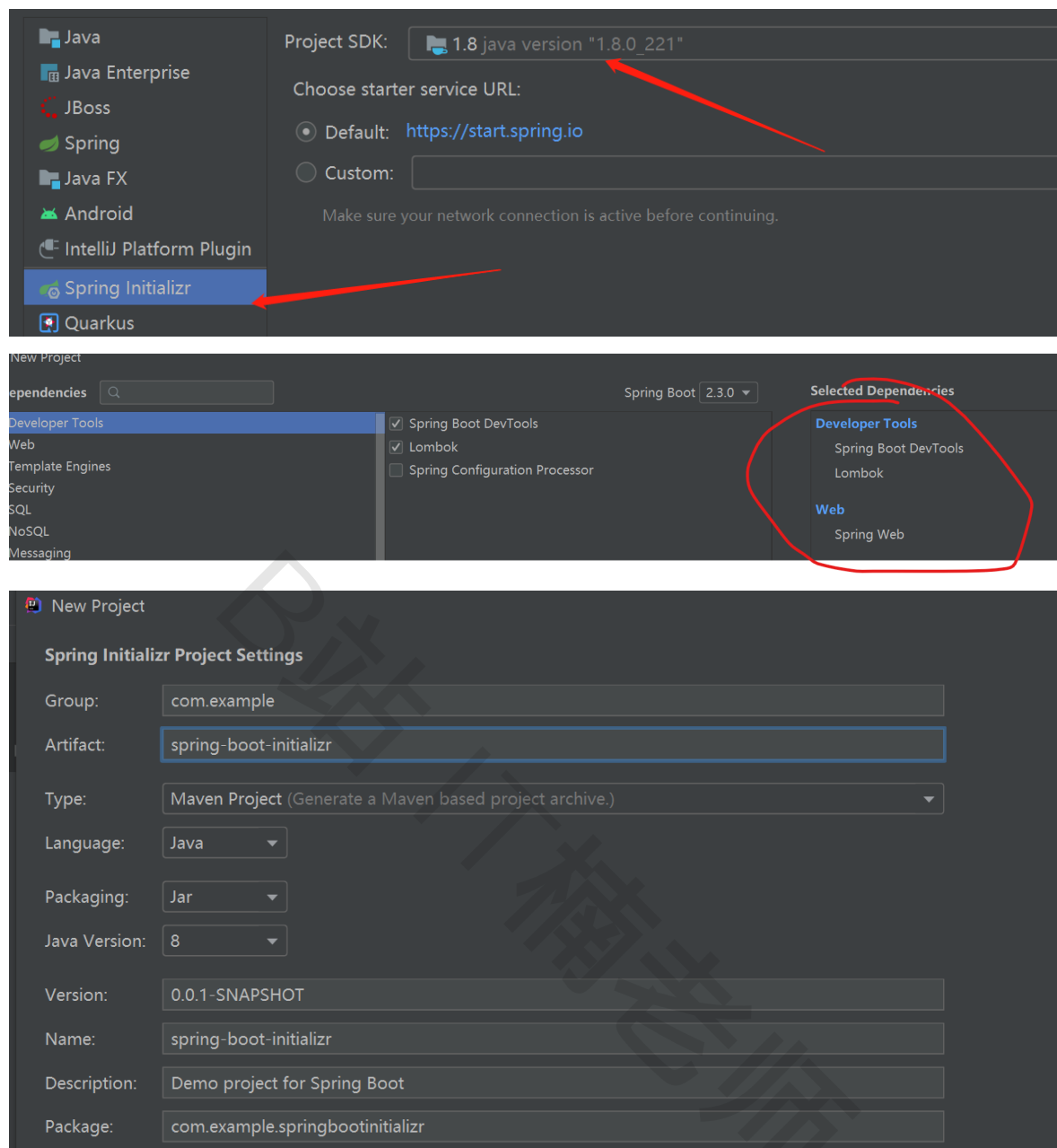
结果ok!

← → ↻ ⓘ 127.0.0.1:8080/user/login

应用 Gmail YouTube 地图 简 Spring入门看这一... 下载 \

hello, zhangsan!

3、使用idea中的spring引导创建



其他的都一样是不是很快是不是很爽！

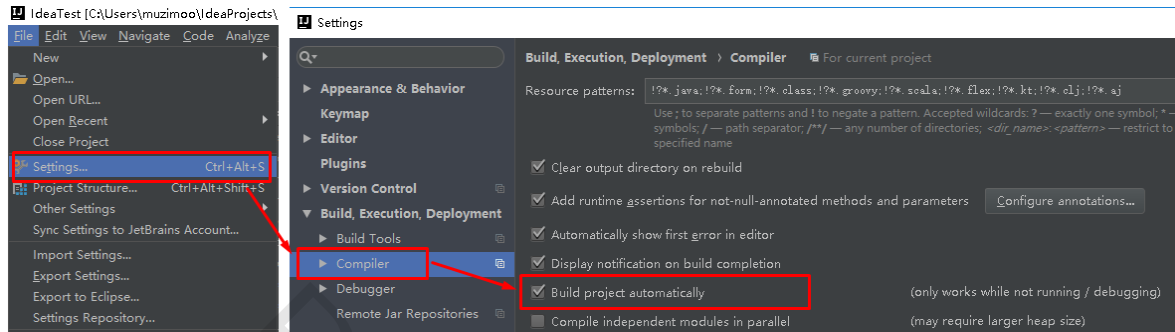
4、SpringBoot工程热部署

我们在开发中反复修改类、页面等资源，每次修改后都是需要重新启动才生效，这样每次启动都很麻烦，浪费了大量的时间，我们可以在修改代码后不重启就能生效，在 pom.xml 中添加如下配置就可以实现这样的功能，我们称之为热部署。

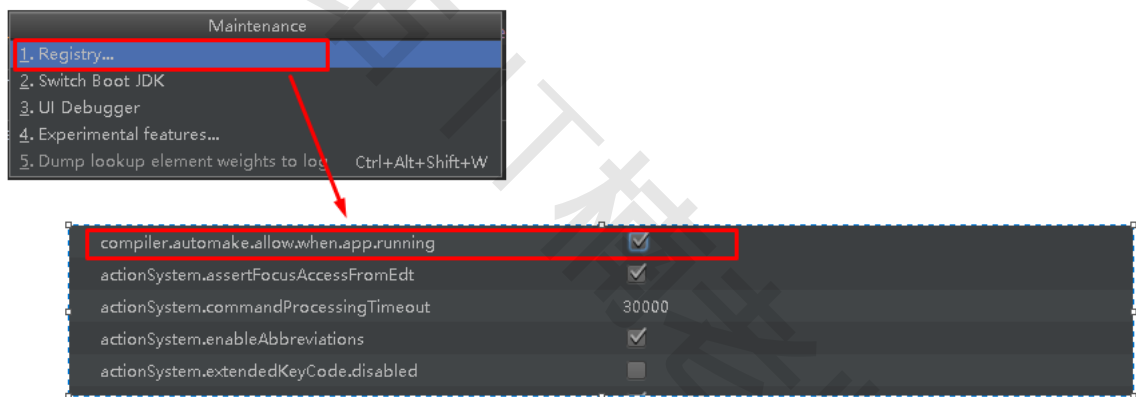
```
<!--热部署配置-->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

注意：IDEA进行SpringBoot热部署失败原因

出现这种情况，并不是热部署配置问题，其根本原因是因为IntelliJ IDEA默认情况下不会自动编译，需要对IDEA进行自动编译的设置，如下：



然后 Shift+Ctrl+Alt+/, 选择Registry



三、SpringBoot原理分析

3.1 起步依赖原理分析

3.1.1 分析spring-boot-starter-parent

按住Ctrl点击pom.xml中的spring-boot-starter-parent，跳转到了spring-boot-starter-parent的pom.xml，xml配置如下（只摘抄了部分重点配置）：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.3.0.RELEASE</version>
</parent>
```

按住Ctrl点击pom.xml中的spring-boot-starter-dependencies，跳转到了spring-boot-starter-dependencies的pom.xml，xml配置如下（只摘抄了部分重点配置）：

```
<properties>
  <activemq.version>5.15.12</activemq.version>
  <commons-codec.version>1.14</commons-codec.version>
  <commons-dbcp2.version>2.7.0</commons-dbcp2.version>
  <commons-lang3.version>3.10</commons-lang3.version>
  <commons-pool.version>1.6</commons-pool.version>
  <commons-pool2.version>2.8.0</commons-pool2.version>
  <ehcache.version>2.10.6</ehcache.version>
  <ehcache3.version>3.8.1</ehcache3.version>
  <elasticsearch.version>7.6.2</elasticsearch.version>
  <freemarker.version>2.3.30</freemarker.version>
  <hibernate.version>5.4.15.Final</hibernate.version>
  <hibernate-validator.version>6.1.5.Final</hibernate-validator.version>
  <hikaricp.version>3.4.5</hikaricp.version>
  <jedis.version>3.3.0</jedis.version>
  <junit.version>4.13</junit.version>
  <junit-jupiter.version>5.6.2</junit-jupiter.version>
  <kafka.version>2.5.0</kafka.version>
  <maven-compiler-plugin.version>3.8.1</maven-compiler-plugin.version>
  <maven-dependency-plugin.version>3.1.2</maven-dependency-plugin.version>
  <mysql.version>8.0.20</mysql.version>
  <neo4j-ogm.version>3.2.11</neo4j-ogm.version>
  <netty.version>4.1.49.Final</netty.version>
  <netty-tcnative.version>2.0.30.Final</netty-tcnative.version>
  <quartz.version>2.3.2</quartz.version>
  <servlet-api.version>4.0.1</servlet-api.version>
  <slf4j.version>1.7.30</slf4j.version>
  <snakeyaml.version>1.26</snakeyaml.version>
  <solr.version>8.5.1</solr.version>
  ...
</properties>
<dependencyManagement>
  ...
</dependencyManagement>
```

从上面的spring-boot-starter-dependencies的pom.xml中我们可以发现，一部分坐标的版本、依赖管理、插件管理已经定义好，所以我们的SpringBoot工程继承spring-boot-starter-parent后已经具备版本锁定等配置了。所以起步依赖的作用就是进行依赖的传递。

3.1.2 分析spring-boot-starter-web

按住Ctrl点击pom.xml中的spring-boot-starter-web，跳转到了spring-boot-starter-web的pom.xml，xml配置如下（只摘抄了部分重点配置）：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- This module was also published with a richer model, Gradle metadata, -->
  <!-- which should be used instead. Do not delete the following line which -->
  <!-- is to indicate to Gradle or any Gradle module metadata file consumer -->
  <!-- that they should prefer consuming it instead. -->
```



```
<!-- do_not_remove: published-with-gradle-metadata -->
<modelVersion>4.0.0</modelVersion>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<version>2.3.0.RELEASE</version>
<name>spring-boot-starter-web</name>
<description>Starter for building web, including RESTful, applications using
Spring MVC. Uses Tomcat as the default embedded container</description>
<url>https://spring.io/projects/spring-boot</url>
<organization>
  <name>Pivotal Software, Inc.</name>
  <url>https://spring.io</url>
</organization>
<licenses>
  <license>
    <name>Apache License, Version 2.0</name>
    <url>http://www.apache.org/licenses/LICENSE-2.0</url>
  </license>
</licenses>
<developers>
  <developer>
    <name>Pivotal</name>
    <email>info@pivotal.io</email>
    <organization>Pivotal Software, Inc.</organization>
    <organizationUrl>https://www.spring.io</organizationUrl>
  </developer>
</developers>
<scm>
  <connection>scm:git:git://github.com/spring-projects/spring-
boot.git</connection>
  <developerConnection>scm:git:ssh://git@github.com/spring-projects/spring-
boot.git</developerConnection>
  <url>https://github.com/spring-projects/spring-boot</url>
</scm>
<issueManagement>
  <system>GitHub</system>
  <url>https://github.com/spring-projects/spring-boot/issues</url>
</issueManagement>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.3.0.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.3.0.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
```

```

        <artifactId>spring-boot-starter-json</artifactId>
        <version>2.3.0.RELEASE</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <version>2.3.0.RELEASE</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <scope>compile</scope>
    </dependency>
</dependencies>
</project>

```

从上面的spring-boot-starter-web的pom.xml中我们可以发现，spring-boot-starter-web就是将web开发要使用的spring-web、spring-webmvc等坐标进行了“打包”，这样我们的工程只要引入spring-boot-starter-web起步依赖的坐标就可以进行web开发了，同样体现了依赖传递的作用。

3.2 自动配置原理解析

```

@SpringBootApplication
public class HelloSpringBootStarterApplication {
    public static void main(String[] args) {
        SpringApplication.run(HelloSpringBootStarterApplication.class, args);
    }
}

```

按住Ctrl点击查看启动类MySpringBootApplication上的注解@SpringBootApplication

```

@SpringBootApplication
public class HelloSpringBootStarterApplication {
    public static void main(String[] args) {
        SpringApplication.run(HelloSpringBootStarterApplication.class, args);
    }
}

```

注解@SpringBootApplication的源码

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {

```

其中,

@SpringBootConfiguration: 等同与@Configuration, 既标注该类是Spring的一个配置类

@EnableAutoConfiguration: SpringBoot自动配置功能开启

按住Ctrl点击查看注解@EnableAutoConfiguration

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {

```

其中, @Import(AutoConfigurationImportSelector.class) 导入了AutoConfigurationImportSelector类

按住Ctrl点击查看AutoConfigurationImportSelector源码

```

@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    AutoConfigurationEntry autoConfigurationEntry =
getAutoConfigurationEntry(annotationMetadata);
    return
StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}

protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata
annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations = getCandidateConfigurations(annotationMetadata,
attributes);
    configurations = removeDuplicates(configurations);
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = getConfigurationClassFilter().filter(configurations);

```

```

        fireAutoConfigurationImportEvents(configurations, exclusions);
        return new AutoConfigurationEntry(configurations, exclusions);
    }

    protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
        AnnotationAttributes attributes) {
        List<String> configurations =
            SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),
                getBeanClassLoader());
        Assert.notEmpty(configurations, "No auto configuration classes found in
            META-INF/spring.factories. If you "
                + "are using a custom packaging, make sure that file is
            correct.");
        return configurations;
    }

    public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable
        ClassLoader classLoader) {
        String factoryTypeName = factoryType.getName();
        return loadSpringFactories(classLoader).getOrDefault(factoryTypeName,
            Collections.emptyList());
    }

    /**
     * The location to look for factories.
     * <p>Can be present in multiple JAR files.
     */
    public static final String FACTORIES_RESOURCE_LOCATION = "META-
        INF/spring.factories";

    private static Map<String, List<String>> loadSpringFactories(@Nullable
        ClassLoader classLoader) {
        MultivaluedMap<String, String> result = cache.get(classLoader);
        if (result != null) {
            return result;
        }

        try {
            Enumeration<URL> urls = (classLoader != null ?
                classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
                ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
            result = new LinkedMultivaluedMap<>();
            while (urls.hasMoreElements()) {
                URL url = urls.nextElement();
                UrlResource resource = new UrlResource(url);
                Properties properties =
                    PropertiesLoaderUtils.loadProperties(resource);
                for (Map.Entry<?, ?> entry : properties.entrySet()) {
                    String factoryTypeName = ((String) entry.getKey()).trim();
                    for (String factoryImplementationName :
                        StringUtils.commaDelimitedListToStringArray((String) entry.getValue())) {
                        result.add(factoryTypeName,
                            factoryImplementationName.trim());
                    }
                }
            }
        }
    }

```

```

        cache.put(classLoader, result);
        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load factories from
location [" +
            FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}

```

其中，SpringFactoriesLoader.loadFactoryNames 方法的作用就是从META-INF/spring.factories文件中读取指定类对应的类名称列表

1591333679901

spring.factories 文件中有关自动配置的配置信息如下：

```

... ..

org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAuto
oConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfigur
ation,\
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoCo
nfiguration,\
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfigurati
on,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration
,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
... ..

```

上面配置文件存在大量的以Configuration为结尾的类名称，这些类就是存有自动配置信息的类，而SpringApplication在获取这些类名后再加载

我们以ServletWebServerFactoryAutoConfiguration为例来分析源码：

```

@Configuration
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@ConditionalOnClass(ServletRequest.class)
@ConditionalOnWebApplication(type = Type.SERVLET)
@EnableConfigurationProperties(ServerProperties.class)
@Import({
    ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class,
    ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,
    ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
    ServletWebServerFactoryConfiguration.EmbeddedUndertow.class })
public class ServletWebServerFactoryAutoConfiguration {
    ... ..
}

```

其中，

@EnableConfigurationProperties(ServerProperties.class) 代表加载ServerProperties服务器配置属性类

进入ServerProperties.class源码如下：

```
@ConfigurationProperties(prefix = "server", ignoreUnknownFields = true)
public class ServerProperties {

    /**
     * Server HTTP port.
     */
    private Integer port;

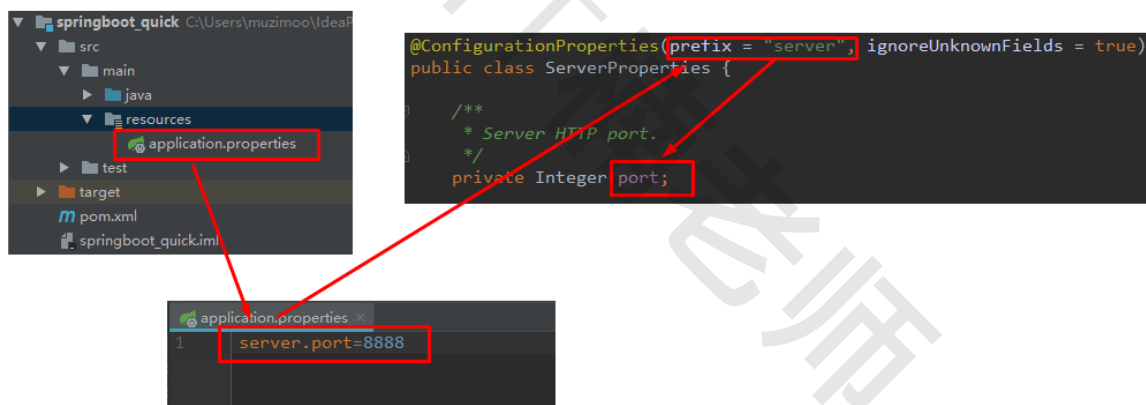
    /**
     * Network address to which the server should bind.
     */
    private InetAddress address;

    ... ..

}
```

其中，

prefix = "server" 表示SpringBoot配置文件中的前缀，SpringBoot会将配置文件中以server开始的属性映射到该类的字段中。映射关系如下：



四、SpringBoot的配置文件

4.1 SpringBoot配置文件类型

4.1.1 SpringBoot配置文件类型和作用

SpringBoot是基于约定的，所以很多配置都有默认值，但如果想使用自己的配置替换默认配置的话，就可以使用application.properties或者application.yml（application.yaml）进行配置。

SpringBoot默认会从Resources目录下加载application.properties或application.yml（application.yaml）文件

其中，application.properties文件是键值对类型的文件，之前一直在使用，所以此处不对properties文件的格式进行阐述。除了properties文件外，SpringBoot还可以使用yaml文件进行配置，下面对yaml文件进行讲解。

4.1.2 application.yml配置文件

4.1.2.1 yaml配置文件简介

YML文件格式是YAML (YAML Aint Markup Language)编写的文件格式，YAML是一种直观的能够被电脑识别的数据序列化格式，并且容易被人类阅读，容易和脚本语言交互的，可以被支持YAML库的不同的编程语言程序导入，比如：C/C++，Ruby，Python，Java，Perl，C#，PHP等。YML文件是以数据为核心的，比传统的xml方式更加简洁。

YML文件的扩展名可以使用.yml或者.yaml。

4.1.2.2 yaml配置文件的语法

4.1.2.2.1 配置普通数据

- 语法：key: value
- 示例代码：
- name: haohao
- 注意：value之前有一个空格

4.1.2.2.2 配置对象数据

- 语法：
key:
key1: value1
key2: value2
或者：
key: {key1: value1,key2: value2}
- 示例代码：

- ```
person:
 name: haohao
 age: 31
 addr: beijing

#或者

person: {name: haohao,age: 31,addr: beijing}
```

- 注意：key1前面的空格个数不限定，在yaml语法中，相同缩进代表同一个级别

#### 4.1.2.2.2 配置Map数据

同上面的对象写法

#### 4.1.2.2.3 配置数组 (List、Set) 数据

- 语法：  
key:  
- value1  
- value2  
或者：  
key: [value1,value2]

- 示例代码:

- ```
city:
  - beijing
  - tianjin
  - shanghai
  - chongqing

#或者

city: [beijing,tianjin,shanghai,chongqing]

#集合中的元素是对象形式
student:
  - name: zhangsan
    age: 18
    score: 100
  - name: lisi
    age: 28
    score: 88
  - name: wangwu
    age: 38
    score: 90
```

- 注意: value1与之间的 - 之间存在一个空格

4.1.3 SpringBoot常用配置

```
# QUARTZ SCHEDULER (QuartzProperties)
spring.quartz.jdbc.initialize-schema=embedded # Database schema initialization
mode.
spring.quartz.jdbc.schema=classpath:org/quartz/impl/jdbcjobstore/tables_@@platfo
rm@@.sql # Path to the SQL file to use to initialize the database schema.
spring.quartz.job-store-type=memory # Quartz job store type.
spring.quartz.properties.*= # Additional Quartz Scheduler properties.

# -----
# WEB PROPERTIES
# -----

# EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.port=8080 # Server HTTP port.
server.servlet.context-path= # Context path of the application.
server.servlet.path=/ # Path of the main dispatcher servlet.

# HTTP encoding (HttpEncodingProperties)
spring.http.encoding.charset=UTF-8 # Charset of HTTP requests and responses.
Added to the "Content-Type" header if not set explicitly.

# JACKSON (JacksonProperties)
spring.jackson.date-format= # Date format string or a fully-qualified date
format class name. For instance, `yyyy-MM-dd HH:mm:ss`.

# SPRING MVC (WebMvcProperties)
spring.mvc.servlet.load-on-startup=-1 # Load on startup priority of the
dispatcher servlet.
spring.mvc.static-path-pattern=/** # Path pattern used for static resources.
```



```
spring.mvc.view.prefix= # Spring MVC view prefix.
spring.mvc.view.suffix= # Spring MVC view suffix.

# DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.driver-class-name= # Fully qualified name of the JDBC driver.
Auto-detected based on the URL by default.
spring.datasource.password= # Login password of the database.
spring.datasource.url= # JDBC URL of the database.
spring.datasource.username= # Login username of the database.

# JEST (Elasticsearch HTTP client) (JestProperties)
spring.elasticsearch.jest.password= # Login password.
spring.elasticsearch.jest.proxy.host= # Proxy host the HTTP client should use.
spring.elasticsearch.jest.proxy.port= # Proxy port the HTTP client should use.
spring.elasticsearch.jest.read-timeout=3s # Read timeout.
spring.elasticsearch.jest.username= # Login username.
```

我们可以通过配置application.properties 或者 application.yml 来修改SpringBoot的默认配置

例如:

application.properties文件

```
server.port=8888
server.servlet.context-path=demo
```

application.yml文件

```
server:
  port: 8888
  servlet:
    context-path: /demo
```

4.2 配置文件与配置类的属性映射方式

4.2.1 使用注解@Value映射

我们可以通过@Value注解将配置文件中的值映射到一个Spring管理的Bean的字段上

例如:

application.properties配置如下:

```
person:
  name: zhangsan
  age: 18
```

或者, application.yml配置如下:

```
person:
  name: zhangsan
  age: 18
```

实体Bean代码如下：

```
@Controller
public class QuickStartController {

    @Value("${person.name}")
    private String name;
    @Value("${person.age}")
    private Integer age;

    @RequestMapping("/quick")
    @ResponseBody
    public String quick(){
        return "springboot 访问成功! name="+name+",age="+age;
    }

}
```

浏览器访问地址：<http://localhost:8080/quick> 结果如下：



4.2.2 使用注解@ConfigurationProperties映射

通过注解@ConfigurationProperties(prefix="配置文件中的key的前缀")可以将配置文件中的配置自动与实体进行映射

application.properties配置如下：

```
person:
  name: zhangsan
  age: 18
```

或者，application.yml配置如下：

```
person:
  name: zhangsan
  age: 18
```

实体Bean代码如下：

```
@Controller
@ConfigurationProperties(prefix = "person")
public class QuickStartController {

    private String name;
    private Integer age;
```

```

@RequestMapping("/quick")
@ResponseBody
public String quick(){
    return "springboot 访问成功! name="+name+",age="+age;
}

public void setName(String name) {
    this.name = name;
}

public void setAge(Integer age) {
    this.age = age;
}
}

```

浏览器访问地址: <http://localhost:8080/quick> 结果如下:



注意: 使用@ConfigurationProperties方式可以进行配置文件与实体字段的自动映射, 但需要字段必须提供set方法才可以, 而使用@Value注解修饰的字段不需要提供set方法

五、web配置

1、定制banner

创建Banner文件 `src/main/resource/banner.txt`

```

${AnsiColor.BRIGHT_YELLOW}
////////////////////////////////////
//                                _ooOoo_                                //
//                                o8888888o                               //
//                                88" . "88                               //
//                                (| ^_^ |)                               //
//                                O\  =  /O                               //
//                                _____\___                          //
//                                .' \\\|      |// \.                     //
//                                /  \\\|| :  ||\\ \  \                   //
//                                / _\\||| -:- ||||- \  \                  //
//                                | | \\\ -  /// | |                    //
//                                | \|  "'\---/'" | |                    //
//                                \ .-\_  \_  _/-. /                      //
//                                _.' . ' /---.\  \ . _                  //
//                                ."" '<  \_<|>/_.' >'""                  //
//                                | | :  \  \; \_ /; \ / - \ : | |         //
//                                \ \ \_  \_  _/  _/  .- / /              //

```

```
//      ===== \-.____ \-.____ \____/ ____.-'____.-'===== //
//      \=====//
//      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ //
//      佛祖保佑      永不宕机      永无BUG //
////////////////////////////////////
${AnsiColor.BRIGHT_RED}
Application Version: ${application.version}${application.formatted-version}
Spring Boot Version: ${spring-boot.version}${spring-boot.formatted-version}
```

从上面的内容中可以看到，还使用了一些属性设置：

- `${AnsiColor.BRIGHT_RED}`：设置控制台中输出内容的颜色，可以自定义，具体参考 `org.springframework.boot.ansi.AnsiColor`
- `${application.version}`：用来获取MANIFEST.MF文件中的版本号，这就是为什么要在 `Application.java`中指定 `SpringVersion.class`
- `{application.formatted-version}`：格式化后的`{application.version}`版本信息
- `${spring-boot.version}`：Spring Boot的版本号
- `{spring-boot.formatted-version}`：格式化后的`{spring-boot.version}`版本信息

2、自定义欢迎页

默认回去static中找index.html

```
classpath:/static/index.html
classpath:/public/index.html
```

3、自定义favicon

浏览器左上角的图标可以放在静态资源下static中，

[随便找个网站http://www.bitbug.net/](http://www.bitbug.net/)生成了放进去就行了

4、其他的服务器配置如jetty

在某个博客看到改Jetty的好处，也真是我现在开发的项目后面要用长连接

好处：

- 1、Jetty适合长连接应用，就是聊天类的长连接
- 2、Jetty更轻量级。这是相对Tomcat而言的。
- 3、jetty更灵活，体现在其可插拔性和可扩展性，更易于开发者对Jetty本身进行二次开发，定制一个适合自身需求的Web Server。
- 4、使用Jetty，需要在spring-boot-starter-web排除spring-boot-starter-tomcat，因为SpringBoot默认使用tomcat

对于配置内置服务器的springBoot，都必定会配置

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

以上配置springBoot的启动web服务器，但默认是Tomcat

所以呢，要配置为jetty要去掉默认tomcat配置

```
<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>
```

并且加上jetty启动

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

5、资源定义

默认，就用默认

```
classpath:/META-INF/resources/
classpath:/resources/
classpath:/static/
classpath:/public/
```

第一种：在配置文件中配置

```
#静态资源访问路径
spring.mvc.static-path-pattern=/**
#静态资源映射路径
spring.resources.static-locations=classpath:/
```

第二种：通过编程进行设置

```

@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        // 这里之所以多了一"/",是为了解决打war时访问不到问题

        registry.addResourceHandler("/**").addResourceLocations("/", "classpath:/");
    }
}

```

6、json数据转换

默认jackson

切换

```

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.35</version>
</dependency>

```

加入消息转换器的配置

```

@Configuration
public class FJsonConfig {

    @Bean
    public HttpMessageConverter configureMessageConverters() {
        FastJsonHttpMessageConverter converter = new
        FastJsonHttpMessageConverter();
        FastJsonConfig config = new FastJsonConfig();
        config.setSerializerFeatures(
            // 保留map空的字段
            SerializerFeature.WriteMapNullValue,
            // 将String类型的null转成""
            SerializerFeature.WriteNullStringAsEmpty,
            // 将Number类型的null转成0
            SerializerFeature.WriteNullNumberAsZero,
            // 将List类型的null转成[]
            SerializerFeature.WriteNullListAsEmpty,
            // 将Boolean类型的null转成false
            SerializerFeature.WriteNullBooleanAsFalse,
            // 避免循环引用
            SerializerFeature.DisableCircularReferenceDetect
        );

        converter.setFastJsonConfig(config);
        converter.setDefaultCharset(Charset.forName("UTF-8"));
        List<MediaType> mediaTypeList = new ArrayList<>();
    }
}

```

```
// 解决中文乱码问题，相当于在Controller上的@RequestMapping中加了个属性produces
= "application/json"
mediaTypeList.add(MediaType.APPLICATION_JSON);
converter.setSupportedMediaTypes(mediaTypeList);
return converter;
}
}
```

7、配置错误页

在Static目录下新建error目录

放入对应的错误页面就行了如404.html 500.html

8、拦截器、过滤器、监听器

首先我们实现拦截器类：

```
@Service
public class UserTokenInterceptor implements HandlerInterceptor {
    @Autowired
    private SysusertokenMapper sysusertokenMapper;
    @Autowired
    private SysloginuserMapper sysloginuserMapper;
    @Autowired
    private SysuserMapper sysuserMapper;
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)
        throws Exception {
        //处理内容
        return isAccess;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler,
        ModelAndView modelAndView) throws Exception {

    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex)
        throws Exception {

    }
}
```

这里我们需要实现HandlerInterceptor这个接口，这个接口包括三个方法，preHandle是请求执行前执行的，postHandler是请求结束执行的，但只有preHandle方法返回true的时候才会执行，afterCompletion是视图渲染完成后才执行，同样需要preHandle返回true，该方法通常用于清理资源等工作。除了实现上面的接口外，我们还需对其进行配置：

```
package com.gcexe.monitor.filter;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;

@Component
public class UserTokenAppConfigurer extends WebMvcConfigurationSupport{

    @Autowired
    private UserTokenInterceptor userTokenInterceptor;
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // 多个拦截器组成一个拦截器链
        // addPathPatterns 用于添加拦截规则
        // excludePathPatterns 用户排除拦截
        registry.addInterceptor(userTokenInterceptor).addPathPatterns("/**")
            .excludePathPatterns("/account/login","/account/register");
        super.addInterceptors(registry);
    }
}
```

这里我们继承了WebMvcConfigurerAdapter，看过前面的文章的朋友应该已经见过这个类了，在进行静态资源目录配置的时候我们用到过这个类。这里我们重写了addInterceptors这个方法，进行拦截器的配置，主要配置项就两个，一个是指定拦截器，第二个是指定拦截的URL。

过滤器、监听器自行学习

9、WebMvcConfigurer总结

WebMvcConfigurer配置类其实是 spring 内部的一种配置方式，采用 JavaBean 的形式来代替传统的 xml 配置文件形式进行针对框架个性化定制，可以自定义一些Handler，Interceptor，ViewResolver，MessageConverter。基于java-based方式的spring mvc配置，需要创建一个配置类并实现WebMvcConfigurer 接口；

在Spring Boot 1.5版本都是靠重写WebMvcConfigurerAdapter的方法来添加自定义拦截器，消息转换器等。SpringBoot 2.0 后，该类被标记为@Deprecated（弃用）。官方推荐直接实现WebMvcConfigurer或者直接继承WebMvcConfigurationSupport，方式一实现WebMvcConfigurer接口（推荐），方式二继承WebMvcConfigurationSupport类。

WebMvcConfigurer接口

常用的方法：

```
/* 拦截器配置 */
void addInterceptors(InterceptorRegistry var1);
```



```

/* 视图跳转控制器 */
void addViewControllers(ViewControllerRegistry registry);

/**
 *静态资源处
 */
void addResourceHandlers(ResourceHandlerRegistry registry);

/* 默认静态资源处理器 */
void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer);

/**
 * 这里配置视图解析器
 */
void configureViewResolvers(ViewResolverRegistry registry);

/* 配置内容裁决的一些选项*/
void configureContentNegotiation(ContentNegotiationConfigurer configurer);

/** 解决跨域问题 */
public void addCorsMappings(CorsRegistry registry) ;

```

addInterceptors: 拦截器

- addInterceptor: 需要一个实现HandlerInterceptor接口的拦截器实例
- addPathPatterns: 用于设置拦截器的过滤路径规则; addPathPatterns("/**") 对所有请求都拦截
- excludePathPatterns: 用于设置不需要拦截的过滤规则

默认静态资源处理器

```

@Override
public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
    configurer.enable();
    configurer.enable("defaultServletName");
}

```

此时会注册一个默认的Handler: DefaultServletHttpRequestHandler, 这个Handler也是用来处理静态文件的, 它会尝试映射/。当DispatcherServlet映射/时 (/ 和/ 是有区别的), 并且没有找到合适的Handler来处理请求时, 就会交给DefaultServletHttpRequestHandler 来处理。注意: 这里的静态资源是放置在web根目录下, 而非WEB-INF 下。可能这里的描述有点不好懂 (我自己也这么觉得), 所以简单举个例子, 例如: 在webroot目录下有一个图片: 1.png 我们知道Servlet规范中web根目录 (webroot) 下的文件可以直接访问的, 但是由于DispatcherServlet配置了映射路径是: / , 它几乎把所有的请求都拦截了, 从而导致1.png 访问不到, 这时注册一个DefaultServletHttpRequestHandler 就可以解决这个问题。其实可以理解为DispatcherServlet破坏了Servlet的一个特性 (根目录下的文件可以直接访问), DefaultServletHttpRequestHandler是帮助回归这个特性的。

configureViewResolvers: 视图解析器

这个方法是用来配置视图解析器的, 该方法的参数ViewResolverRegistry 是一个注册器, 用来注册你想自定义的视图解析器等。ViewResolverRegistry 常用的几个方法: <https://blog.csdn.net/fmwind/article/details/81235401>

```

/**
 * 配置请求视图映射
 * @return
 */
@Bean
public InternalResourceViewResolver resourceViewResolver(){

    InternalResourceViewResolver internalResourceViewResolver = new
    InternalResourceViewResolver();
    //请求视图文件的前缀地址
    internalResourceViewResolver.setPrefix("/WEB-INF/jsp/");
    //请求视图文件的后缀
    internalResourceViewResolver.setSuffix(".jsp");
    return internalResourceViewResolver;
}

```

###

addCorsMappings: 跨域

```

@Override
public void addCorsMappings(CorsRegistry registry) {

    super.addCorsMappings(registry);
    registry.addMapping("/cors/**")
        .allowedHeaders("*")
        .allowedMethods("POST", "GET")
        .allowedOrigins("*");
}

```

configureMessageConverters: 信息转换器

```

/**
 * 消息内容转换配置
 * @param converters
 */

@Override
public void configureMessageConverters(List<HttpMessageConverter<?>> converters)
{

    //调用父类的配置
    super.configureMessageConverters(converters);
    FastJsonHttpMessageConverter converter = new FastJsonHttpMessageConverter();
    FastJsonConfig config = new FastJsonConfig();
    config.setSerializerFeatures(
        // 保留map空的字段
        SerializerFeature.WriteMapNullValue,
        // 将String类型的null转成""
        SerializerFeature.WriteNullStringAsEmpty,
        // 将Number类型的null转成0
        SerializerFeature.WriteNullNumberAsZero,
        // 将List类型的null转成[]
    );
}

```

```

        SerializerFeature.WriteNullListAsEmpty,
        // 将Boolean类型的null转成false
        SerializerFeature.WriteNullBooleanAsFalse,
        // 避免循环引用
        SerializerFeature.DisableCircularReferenceDetect
    );

    converter.setFastJsonConfig(config);
    converter.setDefaultCharset(Charset.forName("UTF-8"));
    List<MediaType> mediaTypeList = new ArrayList<>();
    // 解决中文乱码问题,相当于在Controller上的@RequestMapping中加了个属性produces =
    "application/json"
    mediaTypeList.add(MediaType.APPLICATION_JSON);
    converter.setSupportedMediaTypes(mediaTypeList);

    //将fastjson添加到视图消息转换器列表内
    converters.add(converter);
}

```

10、数据校验

和springmvc一样

11、https配置

使用jdk自带的 keytools 创建证书

打开cmd窗口,输入如下命令

```
keytool -genkey -alias tomcat -keyalg RSA -keystore ./server.keystore
```

按照提示进行操作

```

再次输入新口令:
keytool 错误: java.lang.NullPointerException

C:\Users\zn>keytool -genkey -alias tomcat -keyalg RSA -keystore ./xinzhi.keystore
输入密钥库口令:
再次输入新口令:
您的名字与姓氏是什么?
[Unknown]: zhang
您的组织单位名称是什么?
[Unknown]: xinzhi
您的组织名称是什么?
[Unknown]: xinzhi
您所在的城市或区域名称是什么?
[Unknown]: aa_

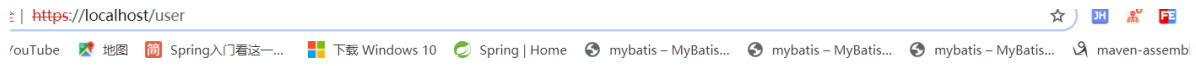
```

创建完成后,可在用户根目录查看生成的keystore文件

将上一步生成的keystone文件复制到项目的根目录,在application.properties添加如下配置

```
server.port=443
server.ssl.key-store=server.keystore
server.ssl.key-alias=tomcat
server.ssl.enabled=true
server.ssl.key-store-password=123456
server.ssl.key-store-type=JKS
```

这里将服务器端口号设置成443端口,即https的默认访问端口,那么在进行https访问的时候可以不带端口号直接访问



您的连接不是私密连接

攻击者可能会试图从 **localhost** 窃取您的信息（例如：密码、通讯内容或信用卡信息）。了解详情

NET::ERR_CERT_AUTHORITY_INVALID

☐ 将您访问的部分网页的网址、有限的系统信息以及部分网页内容发送给 Google，以帮助我们提升 Chrome 的安全性。隐私权政策

隐藏详情

返回安全连接

此服务器无法证明它是**localhost**；您计算机的操作系统不信任其安全证书。出现此问题的原因可能是配置有误或您的连接被拦截了。

继续前往localhost (不安全)

因为我们的证书不是专业机构提供的，而是自己搞的，所以会显示不安全

springboot不能提供http和https两种请求同事存在，所有需要配几个bean实现

http访问自动转https访问

向spring容器中注入两个Bean,代码如下

```
@Bean
public Connector connector() {
    Connector connector=new
Connector("org.apache.coyote.http11.Http11NioProtocol");
    connector.setScheme("http");
    connector.setPort(80);
    connector.setSecure(false);
    connector.setRedirectPort(443);
    return connector;
}

@Bean
public TomcatServletWebServerFactory tomcatServletWebServerFactory(Connector
connector){
    TomcatServletWebServerFactory tomcat=new TomcatServletWebServerFactory()
{
    @Override
```

```

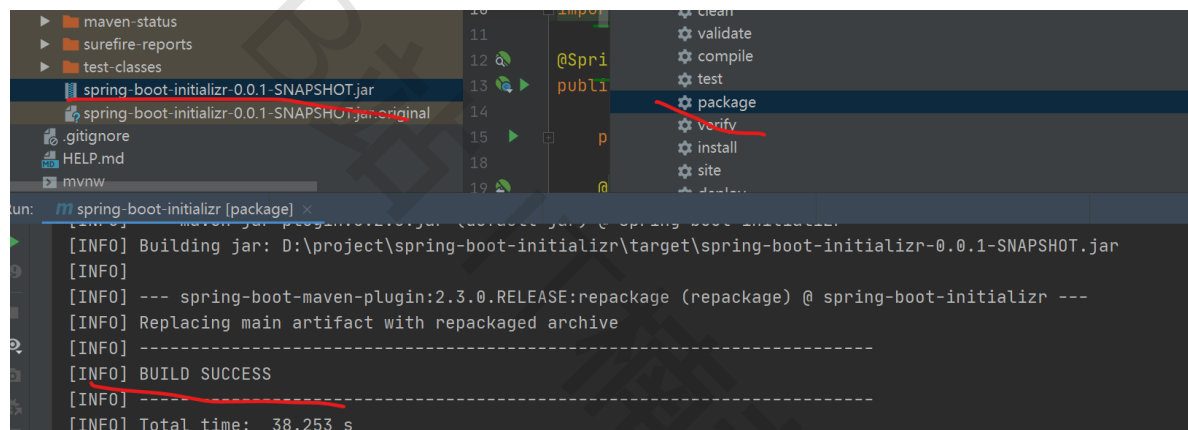
protected void postProcessContext(Context context) {
    SecurityConstraint securityConstraint=new SecurityConstraint();
    securityConstraint.setUserConstraint("CONFIDENTIAL");
    SecurityCollection collection=new SecurityCollection();
    collection.addPattern("/*");
    securityConstraint.addCollection(collection);
    context.addConstraint(securityConstraint);
}
};
tomcat.addAdditionalTomcatConnectors(connector);
return tomcat;
}

```

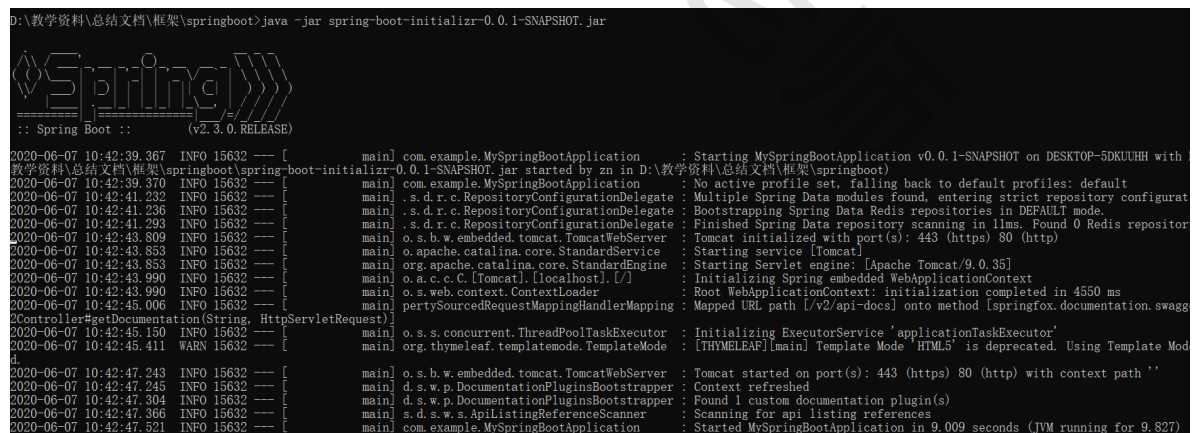
其次在这里设置http的监听端口为80端口,http默认端口,这样在访问的时候也可以不用带上端口号. 完成以上配置后,我们访问 <http://localhost> 即可自动跳转为 <https://localhost>

12、打包

jar包



搞定



war包

标志打包为war

```
<packaging>war</packaging>
```

```
<dependency>
```

```

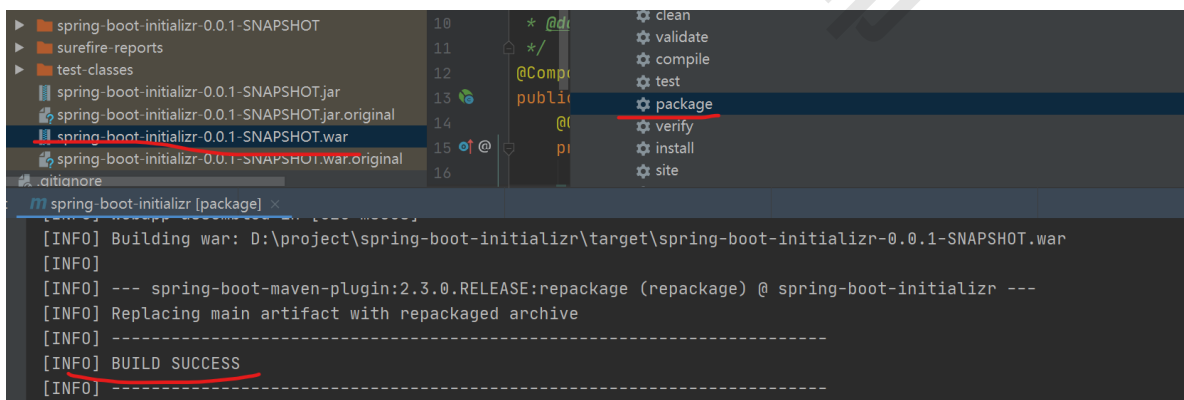
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<exclusions>
    <exclusion>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
    <exclusion>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
</exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-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>

```

```

/**
 * @author IT楠老师
 * @date 2020/6/7
 */
@Component
public class ServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(MySpringBootApplication.class);
    }
}

```



13、启动系统任务

有两个接口CommandLineRunner和ApplicationRunner，实现了这两项接口的类会在系统启动后自动调用执行run方法

```

/**
 * @author IT楠老师
 * @date 2020/6/7
 */
@Component
@Order(1)
public class CommandLineRunnerOne implements CommandLineRunner {
    //args是获取的main方法传入的参数
    @Override
    public void run(String... args) throws Exception {
        //这里可以做一些事情比如redis预热，系统检查等工作
        System.out.println("commandOne---->" + Arrays.toString(args));
    }
}

@Component
@Order(2)
public class CommandLineRunnerTwo implements CommandLineRunner {
    //args是获取的main方法传入的参数
    @Override
    public void run(String... args) throws Exception {
        //这里可以做一些事情比如redis预热，系统检查等工作
        System.out.println("commandTwo---->" + Arrays.toString(args));
    }
}

```

测试

```

FixedRate:Sun Jun 07 11:32:26 CST 2020
fixedDelay:Sun Jun 07 11:32:26 CST 2020
2020-06-07 11:32:26.344 INFO 13188 --- [main] co
commandOne---->[欣知大数据, 楠哥, 真牛逼]
commandTwo---->[欣知大数据, 楠哥, 真牛逼]
initialDelay:Sun Jun 07 11:32:27 CST 2020

```

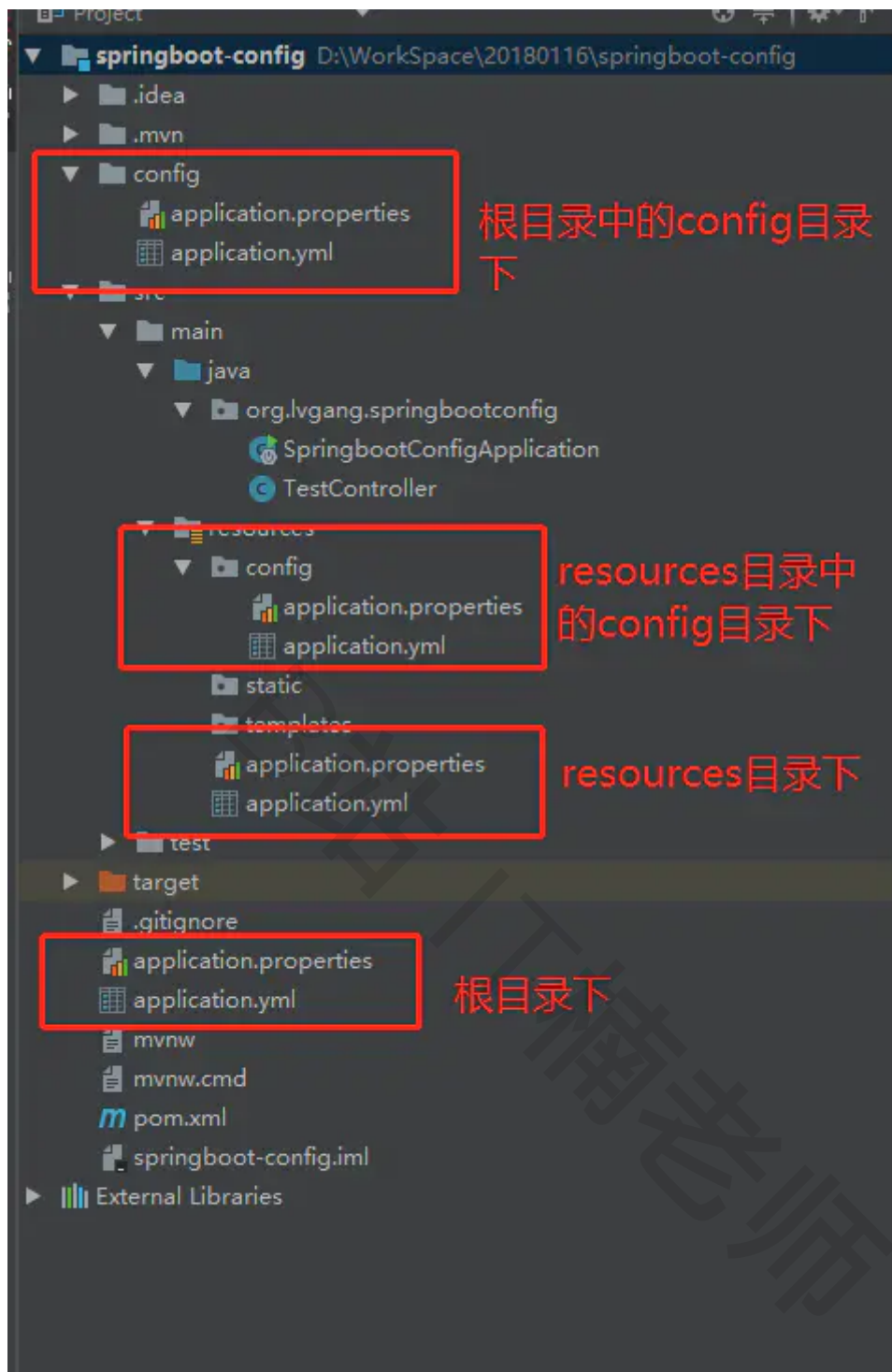
一个和这个接口用法一样，知识参数不同，有兴趣的同学可以研究一下。

14、配置文件的读取顺序

Application属性文件，按优先级排序，位置高的将覆盖位置

1. 当前项目目录下的一个/config子目录
2. 当前项目目录
3. 项目的resources即一个classpath下的/config包
4. 项目的resources即classpath根路径 (root)

如图：



15、读取顺序

如果在不同的目录中存在多个配置文件，它的读取顺序是：

- 1、config/application.properties（项目根目录中config目录下）
- 2、config/application.yml
- 3、application.properties（项目根目录下）
- 4、application.yml
- 5、resources/config/application.properties（项目resources目录中config目录下）
- 6、resources/config/application.yml
- 7、resources/application.properties（项目的resources目录下）
- 8、resources/application.yml

有啥好处，打包后我们可以再jar包之外放置配置文件，随时修改

16、SpringBoot Profile多环境配置

多Profile文件

我们在主配置文件编写的时候，文件名可以是 application-{profile}.properties/yml 例如：

- application-dev.yml
- application-prod.yml
- application-test.yml

默认使用application.properties的配置；

yml支持多文档块方式

```
spring:
  profiles:
    active: dev #指定使用哪个环境
---
server:
  port: 8082
spring:
  profiles: dev
---
server:
  port: 8083
spring:
  profiles: test
---
server:
  port: 8084
spring:
  profiles: prod
```

激活指定Profile

- 在配置文件中指定 spring.profiles.active=dev
- 命令行：java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev;
- 可以直接在测试的时候，配置传入命令行参数
- 虚拟机参数：-Dspring.profiles.active=dev

优先级

命令行参数>JVM参数>配置文件

17、springboot解决跨域

<https://spring.io/blog/2015/06/08/cors-support-in-spring-framework>

当然解决跨域的方法有很。

七、SpringBoot与整合其他技术

1、SpringBoot整合Mybatis

添加Mybatis的起步依赖

```

<!--mybatis起步依赖-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.2</version>
</dependency>
<!-- MySQL连接驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>

```

添加数据库连接信息

在application.properties中添加数据库的连接信息

```

#DB Configuration:
spring:
    datasource:
        url: jdbc:mysql://127.0.0.1:3306/ssm?
characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStatements=
true
        driver-class-name: com.mysql.cj.jdbc.Driver
        username: root
        password: root

```

创建user表

在test数据库中创建user表

```

-- -----
-- Table structure for `user`
-- -----
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(50) DEFAULT NULL,
  `password` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8;

-- -----
-- Records of user
-- -----
INSERT INTO `user` VALUES ('1', 'zhangsan', '123');
INSERT INTO `user` VALUES ('2', 'lisi', '123');

```

创建实体Bean

```

/**
 * @author IT楠老师
 * @date 2020/6/5
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {

    private int id;
    private String username;
    private String password;

}

```

编写Mapper

```

@Mapper
public interface UserMapper {
    public List<User> queryUserList();
}

```

注意：@Mapper标记该类是一个mybatis的mapper接口，可以被spring boot自动扫描到spring上下文中

配置Mapper映射文件

在src\main\resources\mapper路径下加入UserMapper.xml配置文件"

```

<?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="com.xinzhi.dao.UserMapper">
    <select id="queryUserList" resultType="user">
        select * from user
    </select>
</mapper>

```

在application.properties中添加mybatis的信息

```

#spring集成Mybatis环境
mybatis:
    type-aliases-package: com.example.entity
    mapper-locations: mapper/*Mapper.xml

```

service层

```

/**
 * @author IT楠老师
 * @date 2020/6/5
 */
public interface IUserService {

    /**
     * 获取用户信息

```

```

        * @return
        */
        List<User> getAllUsers();
    }

    @Service
    public class UserServiceImpl implements IUserService {

        @Resource
        private UserMapper userMapper;

        @Override
        public List<User> getAllUsers() {
            return userMapper.queryUserList();
        }
    }

```

编写测试Controller

```

/**
 * @author IT楠老师
 * @date 2020/6/5
 */
@Controller
@RequestMapping("/user")
public class UserController {

    @Resource
    private IUserService userService;

    @GetMapping
    @ResponseBody
    public List<User> getUsers(){
        return userService.getAllUsers();
    }
}

```

测试



2、SpringBoot整合Junit

添加Junit的起步依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

编写测试类

```
package com.xinzhi.test;

import com.xinzhi.MySpringBootApplication;
import com.xinzhi.domain.User;
import com.xinzhi.mapper.UserMapper;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = MySpringBootApplication.class)
public class DemoApplicationTests {

    @Resource
    private UserMapper userMapper;

    @Test
```

```

void testUser() {
    System.out.println(userMapper.queryUserList());
}

}

```

3、德鲁伊数据源

默认数据源

```

2020-06-05 22:09:58.684 INFO 5844 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 -
2020-06-05 22:09:58.886 INFO 5844 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 -
[User(id=1, username=123, password=32), User(id=2, username=321, password=123), User(id=3, username=321, password=1
2020-06-05 22:09:58.976 INFO 5844 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down B
2020-06-05 22:09:58.977 INFO 5844 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-1 -
2020-06-05 22:09:58.989 INFO 5844 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-1 -

```

```

/**
 * @author IT楠老师
 * @date 2020/6/5
 */
@Configuration
public class DruidConfig {

    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DataSource druid() {
        return new DruidDataSource();
    }
}

```

```

spring:
  datasource:
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://127.0.0.1:3306/ssm?
characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStatements=
true
    type: com.alibaba.druid.pool.DruidDataSource
    # 数据源其他配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true

```

```
# 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙
filters: stat,wall
maxPoolPreparedStatementPerConnectionSize: 20
useGlobalDataSourceStat: true
connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
```

重新运行

```
2020-06-05 22:30:25.056 INFO 9368 --- [main] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} init
[User(id=1, username=123, password=32), User(id=2, username=321, password=123), User(id=3, username=321, password=123)]
2020-06-05 22:30:25.292 INFO 9368 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down Execu
2020-06-05 22:30:25.294 INFO 9368 --- [extShutdownHook] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} clos
2020-06-05 22:30:25.313 INFO 9368 --- [extShutdownHook] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} clos
```

看后台已经切换了

还有监控呢？

完善配置文件

```
/**
 * @author IT楠老师
 * @date 2020/6/5
 */
@Configuration
public class DruidConfig {

    /**
     * 注入数据源
     * @return
     */
    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DataSource druid() {
        return new DruidDataSource();
    }

    /**
     * 配置监控
     * @return
     */
    @Bean
    public ServletRegistrationBean statViewServlet(){
        ServletRegistrationBean bean = new ServletRegistrationBean(new
        StatViewServlet(), "/druid/*");
        HashMap<String, String> map = new HashMap<>(2);
        map.put("loginUsername", "xinzhi");
        map.put("loginPassword", "123456");
        bean.setInitParameters(map);
        return bean;
    }

    @Bean
    public FilterRegistrationBean webStatFilter(){
        FilterRegistrationBean<Filter> bean = new FilterRegistrationBean<>();
```

```

        bean.setFilter(new WebStatFilter());
        HashMap<String, String> map = new HashMap<>(8);
        map.put("exclusions", "*.js");
        bean.setInitParameters(map);
        bean.setUrlPatterns(Arrays.asList("/*"));
        return bean;
    }
}

```

搞定

N	SQL	执行数	执行时间	最慢	事务执行	错误数	更新行数	读取行数	执行中	最大并发	执行时间分布	执行+RS时分布	读
1	select * from user where ...	1	2	2				1		1	[0,1,0,0,0,0,0]	[1,0,0,0,0,0,0]	[0
2	select * from user	1	7	7				2		1	[0,1,0,0,0,0,0]	[1,0,0,0,0,0,0]	[0

4、Thymeleaf模板引擎

Thymeleaf整合SpringBoot

在pom.xml文件引入thymeleaf

```

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

```

在application.properties (application.yml) 文件中配置thymeleaf

```

spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.check-template-location=true
spring.thymeleaf.suffix=.html
spring.thymeleaf.encoding=UTF-8
spring.thymeleaf.content-type=text/html
spring.thymeleaf.mode=HTML5
spring.thymeleaf.cache=false

```

新建编辑控制层代码HelloController，在request添加了name属性，返回到前端hello.html再使用thymeleaf取值显示。

```

@GetMapping("/login")
public String toLogin(HttpServletRequest request){
    request.setAttribute("name","zhangsan");
    return "login";
}

```


新建编辑模板文件，在resources文件夹下的templates目录，用于存放HTML等模板文件，在这新增hello.html，添加如下代码。

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8"/>
  <title>springboot-thymeleaf demo</title>
</head>

<body>
  <p th:text="'hello, ' + ${name} + '!'" />
</body>
</html>
```

切记：使用Thymeleaf模板引擎时，必须在html文件上方添加该行代码使用支持Thymeleaf。

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

1. 启动项目，访问<http://localhost:8080/user/login>，看到如下显示证明SpringBoot整合Thymeleaf成功。

1591408102619

博客学习该模板引擎语法

https://blog.csdn.net/qg_24598601/article/details/89190411

路径映射

如果觉得写controller太麻烦一次性多映射一些常用的地址

```
@Override
protected void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/login").setViewName("login.html");
    registry.addViewController("/register").setViewName("register.html");
    super.addViewControllers(registry);
}
```

5、集成Swagger

引入依赖

```

<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>

<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>

```

写配置

```

/**
 * @author IT楠老师
 * @date 2020/6/6
 */
@EnableSwagger2
@Configuration
public class SwaggerConfig {
    @Bean
    public Docket customDocket() {
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .select()

        .apis(RequestHandlerSelectors.basePackage("com.example.controller"))
        .build();
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            //文档说明
            .title("欣知测试专用")
            //文档版本说明
            .version("1.0.0")
            .description("欣知学习测试专用")
            .license("Apache 2.0")
            .build();
    }
}

```

测试

```

/**
 * @author IT楠老师
 * @date 2020/6/5
 */
@Controller
@RequestMapping("/user")
@Api("用户接口测试")
public class UserController {

```

```

@Resource
private IUserService userService;

@GetMapping("/toLogin")
public String toLogin(HttpServletRequest request) {
    request.setAttribute("user", "zhangsan");
    return "login";
}

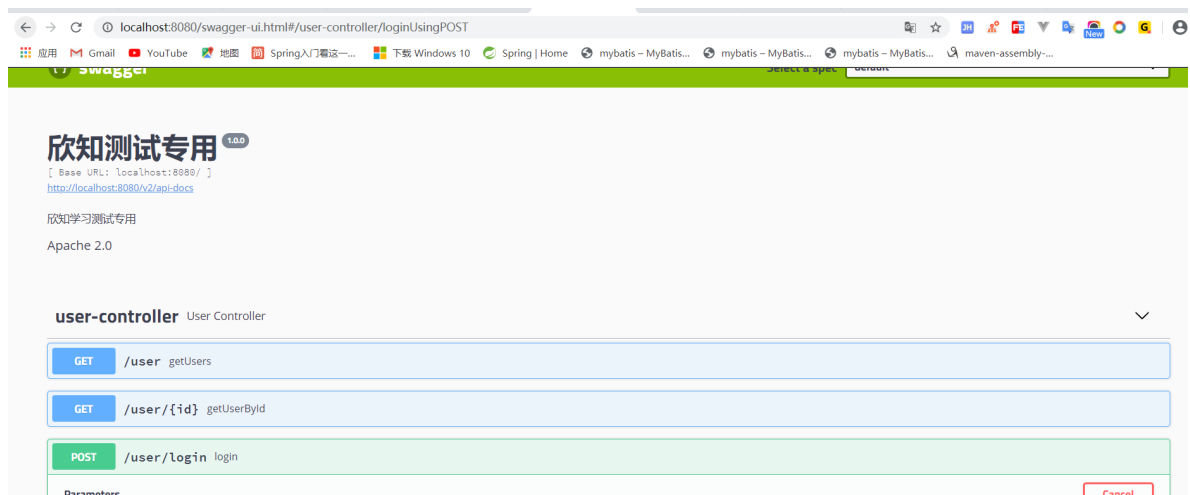
@GetMapping
@ResponseBody
@ApiImplicitParams(
    {

    }
)
public List<User> getUsers() {
    return userService.getAllUsers();
}

@GetMapping("/{id}")
@ResponseBody
@ApiImplicitParam(name = "id", value = "用户id", dataType = "int")
@ApiResponse(code = 200,message = "查找成功")
public User getUserById(@PathVariable int id) {
    return new User(1,"qwe","ewrwe");
}

@PostMapping("/login")
@ResponseBody
@ApiImplicitParams({
    @ApiImplicitParam(name = "username", value = "用户名", dataType =
"string"),
    @ApiImplicitParam(name = "password", value = "密码", dataType =
"string")
})
@ApiResponses({
    @ApiResponse(code = 200,message = "登录成功"),
    @ApiResponse(code = 500,message = "登录失败"),
})
public User login(String username,String password) {
    return new User(12,username,password);
}
}

```



6、定时任务

开启定时任务

```
@EnableScheduling
public class MySpringBootApplication
```

写代码

```
/**
 * @author IT楠老师
 * @date 2020/6/7
 */
@Component
public class Myschedule {

    @Scheduled(fixedDelay = 3000)
    public void fixedDelay(){
        System.out.println("fixedDelay:"+new Date());
    }

    @Scheduled(fixedRate = 3000)
    public void fixedRate(){
        System.out.println("fixedRate:"+new Date());
    }

    @Scheduled(initialDelay = 1000,fixedDelay = 2000)
    public void initialDelay(){
        System.out.println("initialDelay:"+new Date());
    }

    @Scheduled(cron = "0 * * * * ?")
    public void cron(){
        System.out.println("cron:"+new Date());
    }
}
```

```
}
```

结果

```
fixedRate:Sun Jun 07 11:16:17 CST 2020
fixedDelay:Sun Jun 07 11:16:17 CST 2020
2020-06-07 11:16:17.337 INFO 11928 --- [           main]
com.example.MySpringBootApplication : Started MySpringBootApplication in
6.028 seconds (JVM running for 8.211)
initialDelay:Sun Jun 07 11:16:18 CST 2020
fixedRate:Sun Jun 07 11:16:20 CST 2020
fixedDelay:Sun Jun 07 11:16:20 CST 2020
initialDelay:Sun Jun 07 11:16:20 CST 2020
initialDelay:Sun Jun 07 11:16:22 CST 2020
fixedRate:Sun Jun 07 11:16:23 CST 2020
fixedDelay:Sun Jun 07 11:16:23 CST 2020
initialDelay:Sun Jun 07 11:16:24 CST 2020
fixedRate:Sun Jun 07 11:16:26 CST 2020
fixedDelay:Sun Jun 07 11:16:26 CST 2020
initialDelay:Sun Jun 07 11:16:26 CST 2020
initialDelay:Sun Jun 07 11:16:28 CST 2020
fixedRate:Sun Jun 07 11:16:29 CST 2020
fixedDelay:Sun Jun 07 11:16:29 CST 2020
initialDelay:Sun Jun 07 11:16:30 CST 2020
```

区别

- 1、fixedDelay控制方法执行的间隔时间，是以上一次方法执行完开始算起，如上一次方法执行阻塞住了，那么直到上一次执行完，并间隔给定的时间后，执行下一次。
- 2、fixedRate是按照一定的速率执行，是从上一次方法执行开始的时间算起，如果上一次方法阻塞住了，下一次也是不会执行，但是在阻塞这段时间内累计应该执行的次数，当不再阻塞时，一下子把这些全部执行掉，而后再按照固定速率继续执行。
- 3、cron表达式可以定制化执行任务，但是执行的方式是与fixedDelay相近的，也是会按照上一次方法结束时间开始算起。
- 4、initialDelay。如：@Scheduled(initialDelay = 10000,fixedRate = 15000 这个定时器就是在上一个的基础上加了一个initialDelay = 10000 意思就是在容器启动后,延迟10秒后再执行一次定时器,以后每15秒再执行一次该定时器

quartz自学

7、SpringBoot整合Redis

添加redis的起步依赖

```
<!-- 配置使用redis启动器 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

配置redis的连接信息

```
#Redis
spring.redis.host=127.0.0.1
spring.redis.port=6379
```

注入RedisTemplate测试redis操作

```
/**
 * @author IT楠老师
 * @date 2020/6/6
 */
@RunWith(SpringRunner.class)
@SpringBootTest(classes = MySpringBootApplication.class)
public class RedisTest {

    @Resource
    private UserMapper userMapper;

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    @Test
    public void test() throws JsonProcessingException {

        BoundHashOperations<String, Object, Object> hash =
redisTemplate.boundHashOps("user:1");
        hash.put("username", "zhangsan");
        hash.put("age", "12");
        hash.put("password", "1233");

        BoundValueOperations<String, String> userList =
redisTemplate.boundValueOps("user:list");

        //从redis缓存中获得指定的数据
        String usersJson = userList.get();
        //如果redis中没有数据的话
        if(null==usersJson){
            //查询数据库获得数据
            List<User> users = userMapper.queryUserList();
            //转换成json格式字符串
            ObjectMapper om = new ObjectMapper();
            usersJson = om.writeValueAsString(users);
            //将数据存储到redis中，下次在查询直接从redis中获得数据，不用在查询数据库
            redisTemplate.boundValueOps("user:list").set(usersJson);

            System.out.println("=====从数据库获得数据=====");
        }else{
            System.out.println("=====从redis缓存中获得数据=====");
        }

        System.out.println(usersJson);
    }
}
```

第一次

```
Run: RedisTest.test
>> Tests passed: 1 of 1 test - 2 s 908 ms
2020-06-06 21:23:00.517 INFO 11540 --- [main] s.u.s.w.s.ApiListingReferenceScanner : Scanning for api
2020-06-06 21:23:00.687 INFO 11540 --- [main] com.xinzhi.hahah.demo.RedisTest : Started RedisTest
2020-06-06 21:23:03.336 INFO 11540 --- [main] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} in
=====从数据库获得数据=====
[{"id":1,"username":"123","password":"32"}, {"id":2,"username":"321","password":"123"}, {"id":3,"username":"321","password":"123"}]
2020-06-06 21:23:03.747 INFO 11540 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'defaultExecutorService'
2020-06-06 21:23:03.858 INFO 11540 --- [extShutdownHook] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} in
```

第二次

```
Run: RedisTest.test
>> Tests passed: 1 of 1 test - 2 s 735 ms
2020-06-06 21:23:46.086 INFO 7776 --- [main] com.xinzhi.hahah.demo.RedisTest : Started RedisTest
=====从redis缓存中获得数据=====
[{"id":1,"username":"123","password":"32"}, {"id":2,"username":"321","password":"123"}, {"id":3,"username":"321","password":"123"}]
2020-06-06 21:23:49.004 INFO 7776 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'defaultExecutorService'
2020-06-06 21:23:49.007 INFO 7776 --- [extShutdownHook] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} in
```

8、缓存注解开发

博客入门:

<https://www.jianshu.com/p/b8fd074e2802>

开启 @EnableCaching

```
@Configuration
@EnableCaching
public class RedisConfig {

    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory)
        throws UnknownHostException {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setKeySerializer(RedisSerializer.string());
        template.setValueSerializer(RedisSerializer.java());
        template.setHashKeySerializer(RedisSerializer.string());
        template.setHashValueSerializer(RedisSerializer.string());
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }

    @Bean(name = "UserKeyGenerate")
    public KeyGenerator UserKeyGenerate() {
        KeyGenerator keyGenerator = new KeyGenerator() {
            @Override
            public Object generate(Object target, Method method, Object...
params) {
                return method.getName()+"-"+ Arrays.toString(params);
            }
        };
        return keyGenerator;
    }
}
```

```
}  
  
}
```

关键点：

key的生成：

- 使用key属性，可以使用表达式进行精细化的设置
- 使用KeyGenerate，设置key的生成策略，统一配置

拓展点：

- 自行学习自定义缓存管理器，
- 能够对某些命名空间的缓存进行统一管理，比如管理实效时间等。

9、集成shiro（先学shiro）

(1) shiro功能介绍

- Authentication：身份认证/登录
- Authorization：验证权限，即，验证某个人是否有做某件事的权限。
- Session Management:会话管理。管理用户特定的会话，支持web,非web,ejb。
- Cryptography: 加密，保证数据安全。

(2) 其他特性。

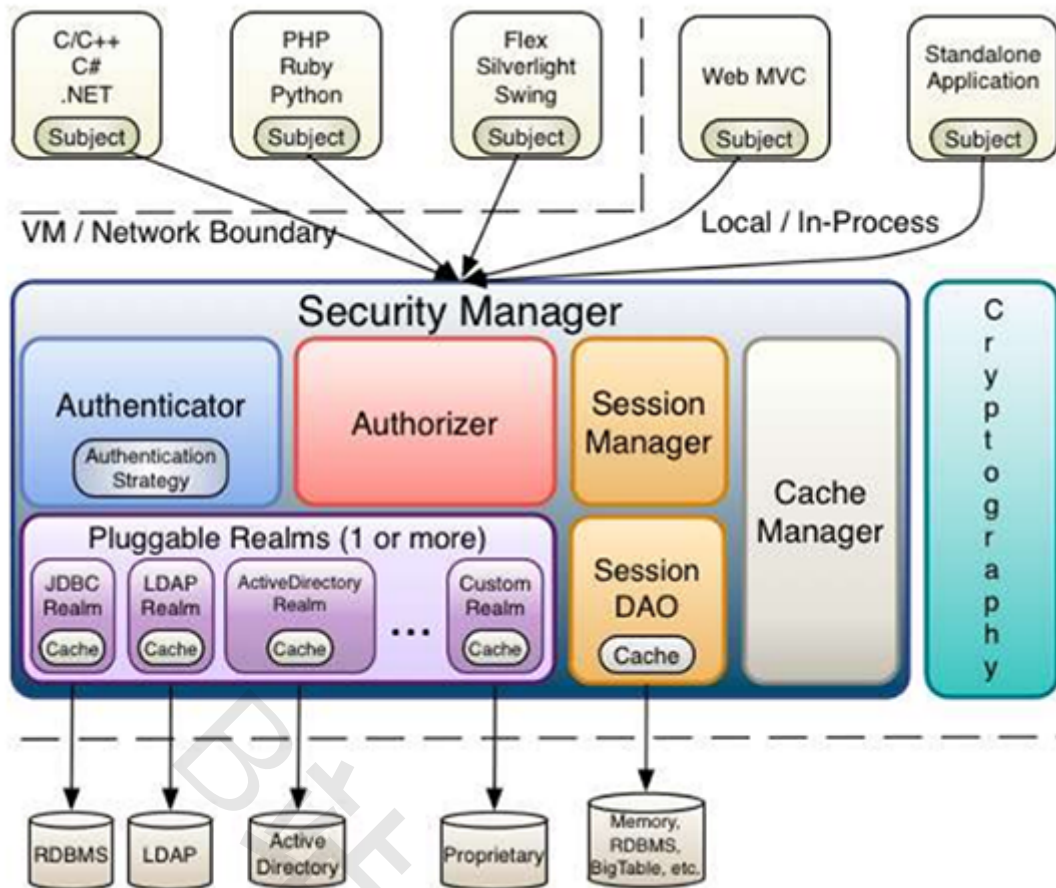
- Web Support:web支持，更容易继承web应用。
- Caching：缓存
- Concurrency：多线程应用的并发验证，即如在一个线程中开启另一个线程，能把权限自动传播过去；
- Testing：提供测试支持。
- Run As：允许一个用户假装为另一个用户（如果他们允许）的身份进行访问；
- Remember Me：记住我，即记住登录状态，一次登录后，下次再来的话不用登录了。

(3) 架构介绍

从最顶层看shiro,有三个最基本概念：Subject, SecurityManager 和Realms。

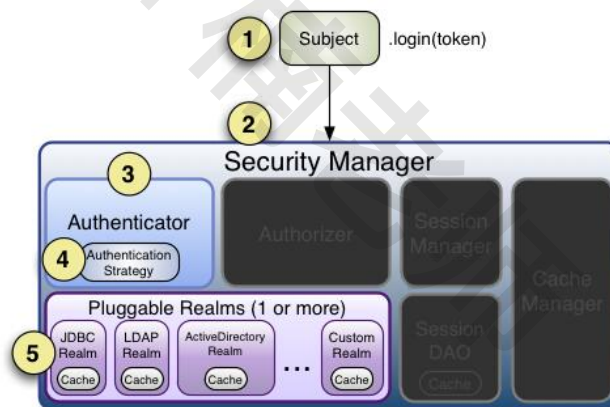
- Subject：主体。抽象概念，代表了当前“用户”，这个用户不一定是具体的人，与当前应用交互的任何东西都是Subject，如网络爬虫，机器人等。
- SecurityManager：安全管理器。shiro的核心，所有与安全有关的操作都会与SecurityManager交互；且它管理着所有Subject。
- Realms：shiro和应用程序的权限数据之间的桥梁，为shiro提供安全数据。SecurityManager要验证用户身份，那么它需要从Realm获取相应的用户进行比较以确定用户身份是否合法；也需要从Realm得到用户相应的角色/权限进行验证用户是否能进行操作；可以把Realm看成DataSource，即安全数据源。

Shiro的架构，如下图所示：



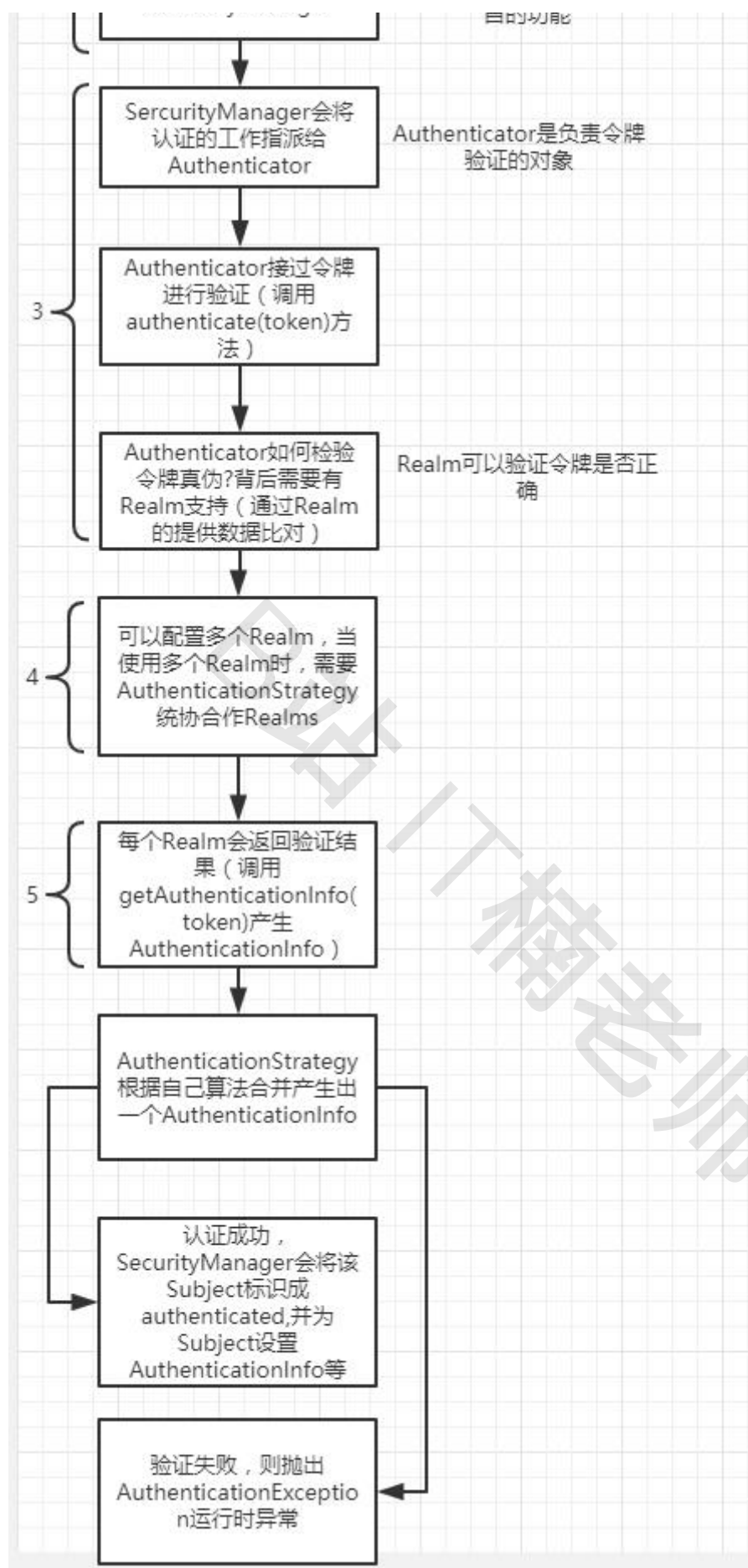
Authentication和Authenticator的主要流程

- Authentication (认证)：与认证流程相关的Shiro各对象关系如下：

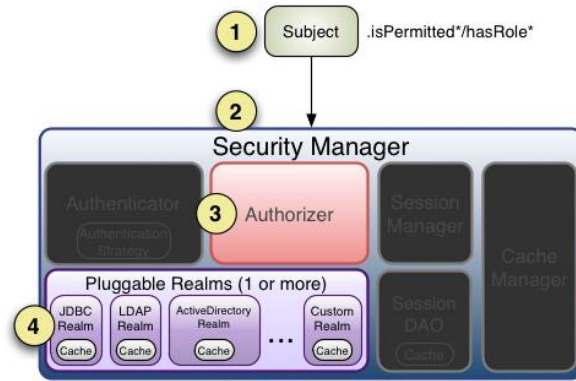


以更加清晰的流程图对应上述的步骤：

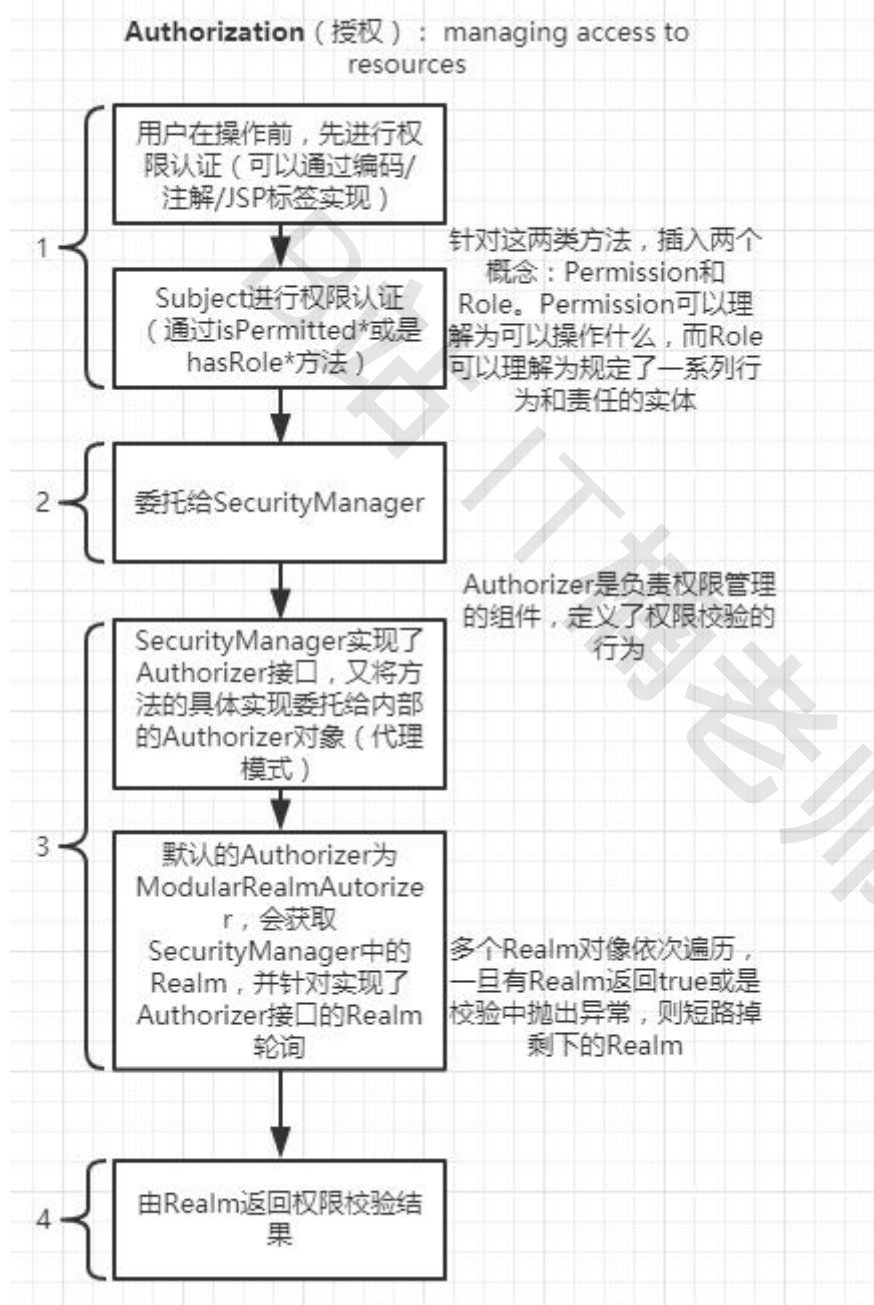




- Authenticator (授权) : 与授权相关的Shiro个对象关系如下:



换成流程图对应上述的步骤:



- Subject (org.apache.shiro.subject.Subject) 与应用交互的主体, 例如用户, 第三方应用等。
- SecurityManager (org.apache.shiro.mgt.SecurityManager) SecurityManager是shiro的核心, 负责整合所有的组件, 使他们能够方便快捷完成某项功能。例如: 身份验证, 权限验证等。
- Authenticator (org.apache.shiro.authc.Authenticator) 认证器, 负责主体认证的, 这是一个扩展点, 如果用户觉得Shiro默认的不好, 可以自定义实现; 其需要认证策略 (Authentication

Strategy) , 即什么情况下算用户认证通过了。

- Authorizer (org.apache.shiro.authz.Authorizer) 来决定主体是否有权限进行相应的操作; 即控制着用户能访问应用中的哪些功能。
- SessionManager (org.apache.shiro.session.mgt.SessionManager) 会话管理。
- SessionDAO (org.apache.shiro.session.mgt.eis.SessionDAO) 数据访问对象, 对session进行 CRUD。
- CacheManager (org.apache.shiro.cache.CacheManager) 缓存管理器。创建和管理缓存, 为 authentication, authorization 和 session management 提供缓存数据, 避免直接访问数据库, 提高效率。
- Cryptography (org.apache.shiro.crypto.*) 密码模块, 提供加密组件。
- Realms (org.apache.shiro.realm.Realm) 可以有1个或多个Realm, 可以认为是安全实体数据源, 即用于获取安全实体的; 可以是JDBC实现, 也可以是LDAP实现, 或者内存实现等等; 由用户提供; 注意: Shiro不知道你的用户/权限存储在哪及以何种格式存储; 所以我们一般在应用中都需要实现自己的Realm。

从博客学习整合

乾乾君子: 博客学习

https://blog.csdn.net/sirchenhua/article/details/100200498?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase

10、pagehelper

为了方便分页

引入依赖

```
<dependency>
  <groupId>com.github.pagehelper</groupId>
  <artifactId>pagehelper-spring-boot-starter</artifactId>
  <version>1.2.10</version>
</dependency>
```

在application.yml中做如下配置

```
# 分页配置
pagehelper:
  helper-dialect: mysql
  reasonable: true
  support-methods-arguments: true
  params: count=countSql
```

在代码中使用, (service或controller)

```
//这行是重点, 表示从pageNum页开始, 每页pageSize条数据
PageHelper.startPage(pageNum, pageSize);
List<Tools> list = toolsMapper.findAll();
PageInfo<Tools> pageInfo = new PageInfo<Tools>(list);
return ServerResponse.createBySuccess("查询成功", pageInfo);
```

开启日志

```
mybatis:
  mapper-locations: mapper/*.xml
  type-aliases-package: com.xinzhi.studyspringboot.entity
  configuration:
    map-underscore-to-camel-case: true
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

结果

```
JDBC Connection [com.alibaba.druid.proxy.jdbc.ConnectionProxyImpl@55099d12] will not
==> Preparing: SELECT count(0) FROM user
==> Parameters:
<== Columns: count(0)
<== Row: 2
<== Total: 1
==> Preparing: select * from user LIMIT ?
==> Parameters: 5(Integer)
<== Columns: id, username, password
<== Row: 1, 123, 321
<== Row: 2, 222, 111
<== Total: 2
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSe
```

127.0.0.1:8089/user

127.0.0.1:8089/user

应用 Gmail YouTube 地图 简 Spring入门看这一... 下载 Windows 10 Spring | Home mybatis - MyBatis...

JSON

```
{
  "total": 2,
  "list": [
    {
      "id": 1,
      "username": 123,
      "password": 321,
      "roles": null
    },
    {
      "id": 2,
      "username": 222,
      "password": 111,
      "roles": null
    }
  ],
  "pageNum": 1,
  "pageSize": 5,
  "size": 2,
  "startRow": 1,
  "endRow": 2,
  "pages": 1
}
```

11、整合Spring Data JPA（自学）

添加Spring Data JPA的起步依赖

```
<!-- springBoot JPA的起步依赖 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

添加数据库驱动依赖

```
<!-- MySQL连接驱动 -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

在application.properties中配置数据库和jpa的相关属性

```
#DB Configuration:
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test?
useUnicode=true&characterEncoding=utf8
spring.datasource.username=root
spring.datasource.password=root

#JPA Configuration:
spring.jpa.database=MySQL
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.naming_strategy=org.hibernate.cfg.ImprovedNamingStrategy
```

创建实体配置实体

```
@Entity
public class User {
    // 主键
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    // 用户名
    private String username;
    // 密码
    private String password;
    // 姓名
    private String name;

    //此处省略setter和getter方法... ..
}
```

编写UserRepository

```
public interface UserRepository extends JpaRepository<User, Long>{
    public List<User> findAll();
}
```

编写测试类

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes=MySpringBootApplication.class)
public class JpaTest {

    @Autowired
```



```

private UserRepository userRepository;

@Test
public void test(){
    List<User> users = userRepository.findAll();
    System.out.println(users);
}
}

```

控制台打印信息

```

1 test passed - 302ms
2018-05-10 10:12:51.109 INFO 1604 --- [main] com.itheima.test.JpaTest : Started Jp
2018-05-10 10:12:51.289 INFO 1604 --- [main] o.h.h.i.QueryTranslatorFactoryInitiator : HHH000397:
Hibernate: select user0.id as id1 0 , user0.name as name2 0 , user0.password as password3 0 , user0.userna
[User{id=1, username='zhangsan', password='123', name='张三'}, User{id=2, username='lisi', password='123', na
2018-05-10 10:12:51.441 INFO 1604 --- [Thread-1] o.s.w.c.s.GenericWebApplicationContext : Closing or
2018-05-10 10:12:51.451 INFO 1604 --- [Thread-1] j.LocalContainerEntityManagerFactoryBean : Closing JP

```

注意：如果是jdk9，执行报错如下：

```

Caused by: java.lang.ClassNotFoundException: javax.xml.bind.JAXBException
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:582)
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:185)
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:496)
    ... 50 more

```

原因：jdk缺少相应的jar

解决方案：手动导入对应的maven坐标，如下：

```

<!--jdk9需要导入如下坐标-->
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>

```

六、自定义starter

自定义starter实例

我们需要先创建两个工程 **hello-spring-boot-starter** 和 **hello-spring-boot-starter-autoconfigurer**

1. hello-spring-boot-starter

1.pom.xml

```

<?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>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>

```

```

        <version>2.3.0.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.xinzhi</groupId>
    <artifactId>hello-spring-boot-starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>hello-spring-boot-starter</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-autoconfigure</artifactId>
        </dependency>
    </dependencies>

</project>

```

同时删除 启动类、resources下的文件，test文件。

2. HelloProperties

```

/**
 * @author IT楠老师
 * @date 2020/6/5
 */
@ConfigurationProperties(prefix = "xinzhi.user")
public class UserProperties {
    private String username = "张三";
    private String password = "123";

    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;
    }
}

```

3. HelloService

```

/**
 * @author IT楠老师

```



```

    * @date 2020/6/5
    */
@Component
public class User {

    @Resource
    private UserProperties userProperties;

    public void say(){
        System.out.println(userProperties.getUsername()+"的密码
是: "+userProperties.getPassword());
    }
}

```

4. UserAutoConfiguration

```

/**
 * @author IT楠老师
 * @date 2020/6/5
 */
@Configuration
@ConditionalClass(User.class)
@ConditionalOnProperty(prefix = "xinzhi.user",value = "enable",matchIfMissing =
true)
@EnableConfigurationProperties(UserProperties.class)
public class UserAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public User greeter() {
        return new User();
    }
}

```

5. spring.factories

在 **resources** 下创建文件夹 **META-INF** 并在 **META-INF** 下创建文件 **spring.factories** , 内容如下:

```

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.xinzhi.config.UserAutoConfiguration

```

到这儿, 我们的配置自定义的starter就写完了

执行: `mvn install` 安装带本地仓库

三、测试自定义starter

我们创建个项目 **hello-spring-boot-starter-test**, 来测试系我们写的stater。

1. pom.xml

```

<?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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.0.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>spring-boot-initializr</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>spring-boot-initializr</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>com.xinzhi</groupId>
      <artifactId>hello-spring-boot-starter</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>


```

2. HelloController

```
@Controller
public class UserController {

    @Resource
    private User user;

    @GetMapping("/user")
    @ResponseBody
    public void getUser(){
        user.say();
    }
}
```

1591340121810

3. 修改application.properties


```
xinzhi.user.username=李四
xinzhi.user.password=abc
```

1591340475864

B站: IT楠老师 公众号: IT楠说java QQ群: 1083478826 新知大数据

制作不易、如果觉的好不妨打个赏:



 微信支付