

今日授课目标

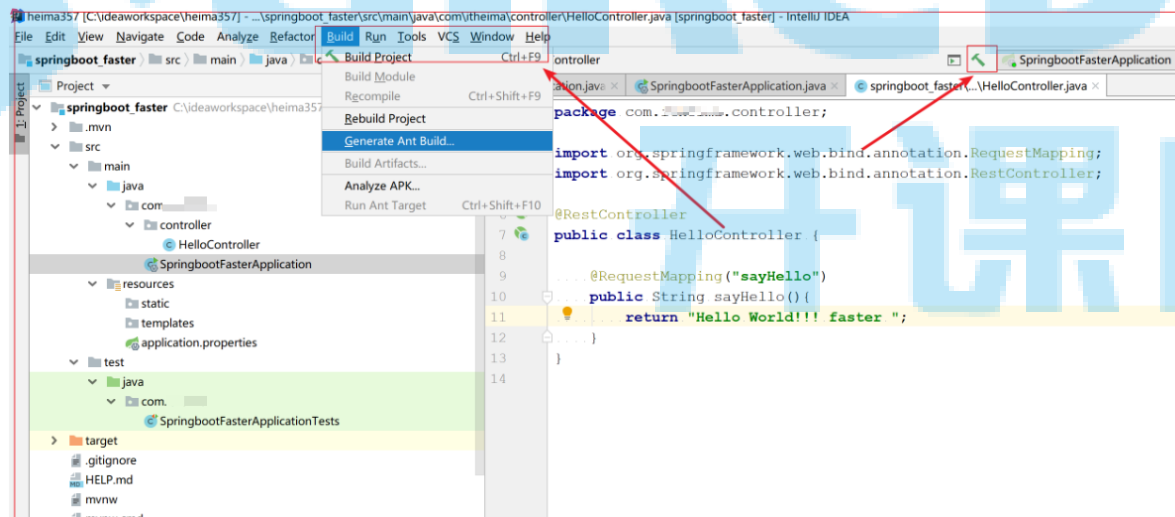
- 掌握工程热部署
- 掌握多环境的配置文件
- 掌握配置文件存放路径，及其加载顺序
- 掌握自定义配置文件名称
- 掌握内置web应用服务器的切换：tomcat切换为jetty
- 掌握为SpringBoot配置生产级监控Actuator
- 搭建Spring Boot Admin服务链接SpringBoot项目
- 了解SpringBoot自动配置实现原理【不要求掌握】

七、SpringBoot工程热部署

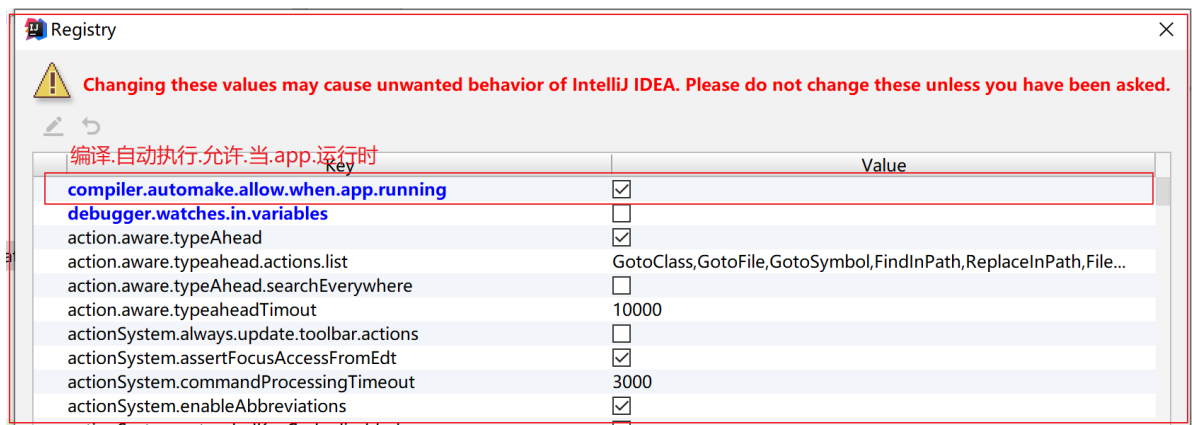
只需导入开发者工具依赖坐标，即可实现热部署功能：

```
1 <!--spring-boot开发工具jar包，支持热部署-->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-devtools</artifactId>
5 </dependency>
```

但还需注意：加入坐标之后，如果想要代码立即生效，必须在修改代码之后进行代码构建。默认情况IDEA不会自动构建，需要手动构建。如图两处地方均可。



每次手动构建很麻烦！！还有一种自动构建解决方案，但不建议使用。就是设置 Build Project Automatically。同时打开Maintenance维护(打开快捷键 Shift + Ctrl + Alt + /)，选择 Registry(注册表)，设置运行时自动编译。



八、配置文件延伸

8.1 多环境配置文件

我们在开发Spring Boot应用时，通常同一套程序会被安装到不同环境，比如：开发dev、测试test、生产pro等。其中数据库地址、服务器端口等等配置都不同，如果每次打包时，都要修改配置文件，那么非常麻烦。profile功能就是来进行动态配置切换的。

profile配置方式有两种：

- 多profile文件方式：提供多个配置文件，每个代表一种环境。
 - application-dev.properties/yml 开发环境
 - application-test.properties/yml 测试环境
 - application-pro.properties/yml 生产环境
- yml多文档方式：在yml中使用 --- 分隔不同配置

profile激活方式：

- 配置文件：再配置文件中配置：spring.profiles.active=dev
- 虚拟机参数：在VM options 指定：-Dspring.profiles.active=dev
- 命令行参数：java -jar xxx.jar --spring.profiles.active=dev

8.2 松散绑定

不论配置文件中的属性值是短横线、驼峰式还是下划线分隔配置方式，在注入配置时都可以通过短横线方式取出值；

使用范围：properties文件、YAML文件、系统属性

命名风格	示例
短横线分隔	abc.spring-boot.one-example
驼峰式	abc.SpringBoot.OneExample
下划线分隔	abc.spring_boot.one_example

注意：@Value取值配置不能写驼峰式和下划线，只能写短横线，否则会报错

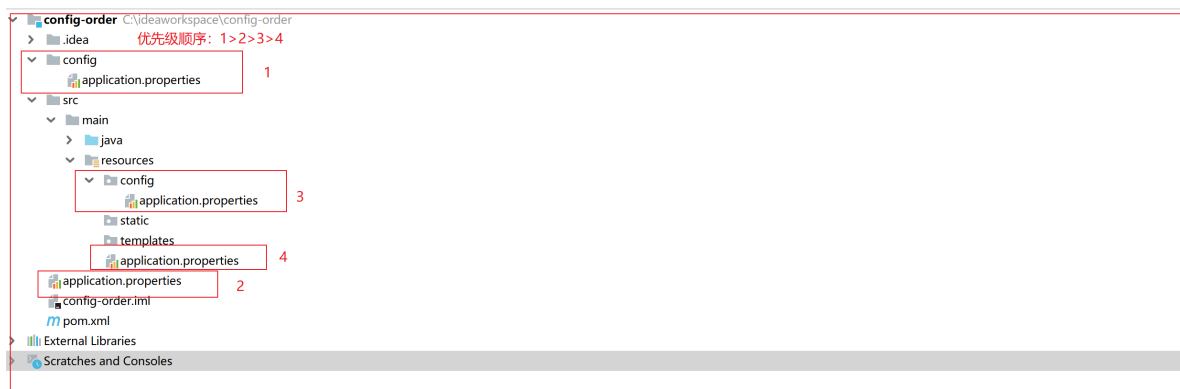
8.3 配置路径及其加载顺序

Springboot程序启动时，会从以下位置加载配置文件：

- 1.file:./config/: 当前项目下的/config目录下

- 2.file:/ : 当前项目的根目录
- 3.classpath:/config/: classpath的/config目录
- 4.classpath:/ : classpath的根目录

加载顺序为上文的排列顺序，高优先级配置的属性会生效



注意：在多个model的工程中，这种配置顺序不生效

8.4 外部配置加载顺序

1. 开启 DevTools 时， ~/.spring-boot-devtools.properties
2. 测试类上的 @TestPropertySource 注解
3. @SpringBootTest#properties 属性
4. 命令行参数 (--server.port=9000)
5. SPRING_APPLICATION_JSON 中的属性
6. ServletConfig 初始化参数
7. ServletContext 初始化参数
8. java:comp/env 中的 JNDI 属性
9. System.getProperties()
10. 操作系统环境变量
11. random.* 涉及到的 RandomValuePropertySource
12. jar 包外部的 application-{profile}.properties 或 .yaml
13. jar 包内部的 application-{profile}.properties 或 .yaml
14. jar 包外部的 application.properties 或 .yaml
15. jar 包内部的 application.properties 或 .yaml
16. @Configuration 类上的 @PropertySource
17. SpringApplication.setDefaultProperties() 设置的默认属性

官方说明[地址](#)：

8.5 修改配置文件的位置及默认名称：

第一种方式：IDEA通过参数注入方式配置：

1、js中的事件监听机制：onclick=fun()

2、Servlet中的监听器：web.xml

```
1 <listener-  
  class>org.springframework.web.context.ContextLoaderListener</listener-class>
```

3、SpringBoot中的监听器

- CommandLineRunner：应用程序启动完成后
- ApplicationRunner：应用程序启动完成后
- ApplicationContextInitializer：应用程序初始化之前【对框架开发者有意义】
- SpringApplicationRunListener：应用程序全阶段监听【对框架开发者有意义】
 1. 应用程序开始启动--starting
 2. 环境准备完成--environmentPrepared
 3. spring容器准备完成--environmentPrepared
 4. spring容器加载完成--environmentPrepared
 5. 应用程序启动完成--started
 6. 应用程序开始运行--running
 7. 应用程序运行时抛出异常时调用--failed

Spring Boot的扩展机制Spring Factories

- 主要目的是解耦：将监听器的配置权交给第三方厂商、插件开发者
- 框架提供接口，实现类由你自己来写，释放原生API能力，增加可定制性
- META-INF/spring.factories文件中配置接口的实现类名称

在ApplicationContextInitializer接口的实现类中必须写一个构造方法：

```
1 //注意：此构造方法必须要写  
2 public My04SpringApplicationRunListener(SpringApplication application,  
  String[] args) {}
```

2、监听器案例

需求：当应用程序启动完成时，初始化redis缓存。

实现步骤：

1. 实现ApplicationRunner接口，重写run()方法
2. 注入UserService接口实现类及RedisTemplate对象
3. 将用户数据存入redis缓存

实现过程：

```
1 //1.实现ApplicationRunner接口，重写run()方法  
2 @Component  
3 public class InitialzerRedisCacheListnener implements ApplicationRunner {  
4     //2.注入UserService接口实现类及RedisTemplate对象  
5     @Autowired  
6     private UserService userService;  
7     @Autowired  
8     private RedisTemplate redisTemplate;
```

```

9
10     @Override
11     public void run(ApplicationArguments args) throws Exception {
12         //3.将缓存数据存入redis
13         String key = UserService.class.getName() + ".findAll()";
14         redisTemplate.boundValueOps(key).set(userService.findAll());
15     }
16 }

```

9.2 自动配置实现原理详解

redisTemplate的bean怎么注入spring容器中的?

redisTemplate中的host和port怎么配置的?

1、@Import注解进阶

三种Jar包外导入类的方式:

- 直接导入
- 通过配置类导入
- 通过ImportSelector接口实现类导入

2、@Configuration注解进阶

配置类中的条件注解@Conditional及其衍生注解

- class类条件
 - @ConditionalOnClass==存在类条件
 - @ConditionalOnMissingClass==不存在类条件
- 属性条件
 - @ConditionalOnProperty==属性条件, 还可以为属性设置默认值
- Bean条件
 - @ConditionalOnBean==存在Bean条件
 - @ConditionalOnMissingBean==不存在Bean条件
 - @ConditionalOnSingleCandidate==只有一个Bean条件
- 资源条件
 - @ConditionalResource==资源条件
- Web应用条件
 - @ConditionalOnWebApplication==web应用程序条件
 - @ConditionalOnNotWebApplication==不是web应用程序条件
- 其他条件
 - @ConditionalOnExpression==EL表达式条件
 - @ConditionalOnJava==在特定的Java版本条件

自动配置类的执行顺序

1. @AutoConfigureBefore==在那些自动配置之前执行
2. @AutoConfigureAfter==在那些自动配置之后执行
3. @AutoConfigureOrder==自动配置顺序

Redis的自动配置

```

@Configuration
@ConditionalOnClass(RedisOperations.class)
@EnableConfigurationProperties(RedisProperties.class)
@Import({ LettuceConnectionFactory.class, JedisConnectionFactory.class })
public class RedisAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean(name = "redisTemplate")
    public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory)
        throws UnknownHostException {
        RedisTemplate<Object, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }

    @Bean
    @ConditionalOnMissingBean
    public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory redisConnectionFactory)
        throws UnknownHostException {
        StringRedisTemplate template = new StringRedisTemplate();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }
}

```

Mybatis的自动配置

```

@org.springframework.context.annotation.Configuration
@ConditionalOnClass({ SqlSessionFactory.class, SqlSessionFactoryBean.class })
@ConditionalOnSingleCandidate(DataSource.class)
@EnableConfigurationProperties(MybatisProperties.class)
@AutoConfigureAfter({ DataSourceAutoConfiguration.class, MybatisLanguageDriverAutoConfiguration.class })
public class MybatisAutoConfiguration implements InitializingBean {

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

    private final MybatisProperties properties;

    private final Interceptor[] interceptors;

    private final TypeHandler[] typeHandlers;
}

```

3、@EnableAutoConfiguration注解

其本质是配置类@Import与@Configuration的组合

```

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

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    /**
     * Exclude specific auto-configuration classes such that they will never be applied.
     * @return the classes to exclude
     */
    Class<?>[] exclude() default {};
}

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {
}

```

9.3 自定义auto-configuration及starter

目标：

为传智播客教育集团研究院开发的，基于自主研发的模板引擎代码生成神器加入自动配置。要求当导入starter坐标时，自动创建代码生成器框架的核心对象codeutil。

分析过程：

参考Mybatis实现的自动配置

自动配置和starter是针对已有框架进行的整合！

必备四角色：框架、自动配置模块、starter模块、开发者

- 定制自动配置必要内容
 - autoconfiguration模块，包含自动配置代码。自定义kkb-spring-boot-autoconfigure。
 - starter模块。自定义kkb-spring-boot-starter
- 自动配置命名方式
 - 官方的Starters
 - spring-boot-starter-*
 - 非官方的starters
 - *-spring-boot-starter
- SpringBoot起步依赖，Starter Dependencies

一些注意事项

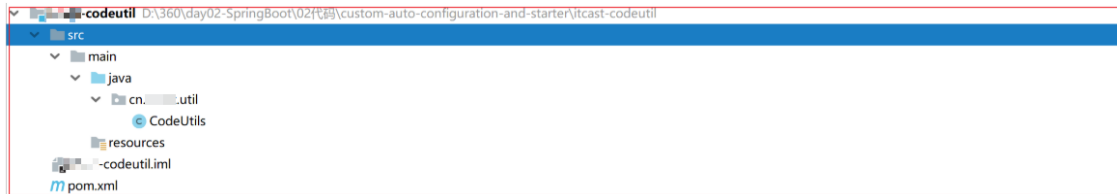
- starter中仅添加必备依赖坐标
- 需要声明对spring-boot-autoconfigure的依赖
- 测试项目的包名不能与自定义starter和自动配置的包名相同
 - com.abc
 - cn.kkb

实现步骤：

1. 获取准备好的框架--人人架构-代码生成器
2. 创建 kkb-spring-boot-autoconfigure 模块
 - 定义代码生成器核心对象的配置类信息
 - 在META-INF/spring.factories扩展自动配置
3. 创建 kkb-spring-boot-starter 模块
4. 将自动配置模块及starter模块安装到本地仓库
5. 测试模块springboot04-test-my-auto-configuration
 - 引入自定义的 kkb-spring-boot-starter依赖坐标
 - 测试核心代码生成器对象，是否自动配置成功了

实现过程：

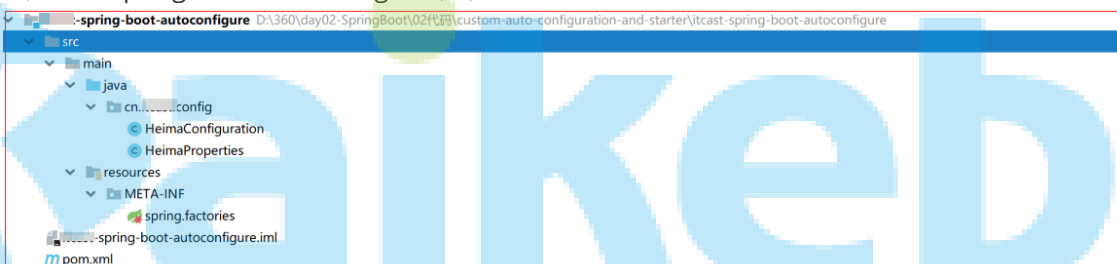
1. 获取准备好的框架--人人架构-代码生成器



- CodeUtils:

```
1  /**
2   * 代码生成器，程序执行入口
3   */
4  public class CodeUtils {
5
6      public void generatorCode(){
7          //生成代码
8          System.out.println("-----人人架构代码生成器执行-----");
9      }
10 }
```

2. 创建 kkb-spring-boot-autoconfigure 模块



- HeimaConfiguration

```
1  @Configuration
2  @ConditionalOnClass(CodeUtils.class)
3  @Import(HeimaProperties.class)
4  public class HeimaConfiguration {
5
6      //配置人人架构的核心对象CodeUtils
7      @Bean
8      @ConditionalOnProperty(name = "spring.heima.enable",havingValue
9      = "true",matchIfMissing = true)
10     public CodeUtils codeutil(){
11         return new CodeUtils();
12     }
13 }
```

- HeimaProperties

```
1  @ConfigurationProperties(prefix = "spring.heima")
2  public class HeimaProperties {
3      private String username;
4      private String password;
5      //getter setter toString
6  }
```

- META-INF/spring.factories

```

1 # 注册自定义自动配置
2 org.springframework.boot.autoconfigure.EnableAutoConfiguration=cn.kkb
   .config.HeimaConfiguration

```

o pom.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <parent>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-parent</artifactId>
10        <version>2.1.11.RELEASE</version>
11    </parent>
12    <groupId>cn.kkb</groupId>
13    <artifactId>kkb-spring-boot-autoconfigure</artifactId>
14    <version>1.0-SNAPSHOT</version>
15
16    <dependencies>
17        <dependency>
18            <groupId>org.springframework.boot</groupId>
19            <artifactId>spring-boot-autoconfigure</artifactId>
20        </dependency>
21        <!-- 导入需要加入自定义自动配置的框架 -->
22        <dependency>
23            <groupId>cn.kkb</groupId>
24            <artifactId>kkb-codeutils</artifactId>
25            <version>1.0-SNAPSHOT</version>
26        </dependency>
27    </dependencies>
28 </project>

```

3. 创建 kkb-spring-boot-starter 模块

spring-boot-starter D:\360\day02-SpringBoot\02代码\custom-auto-configuration-and-starter\itcast-spring-boot-starter
 it-spring-boot-starter.iml
 pom.xml

o pom.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.kkb</groupId>
8     <artifactId>kkb-spring-boot-starter</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <dependencies>
11        <!-- 人人架构-代码生成器的依赖 -->
12        <dependency>
13            <groupId>com.abc</groupId>

```

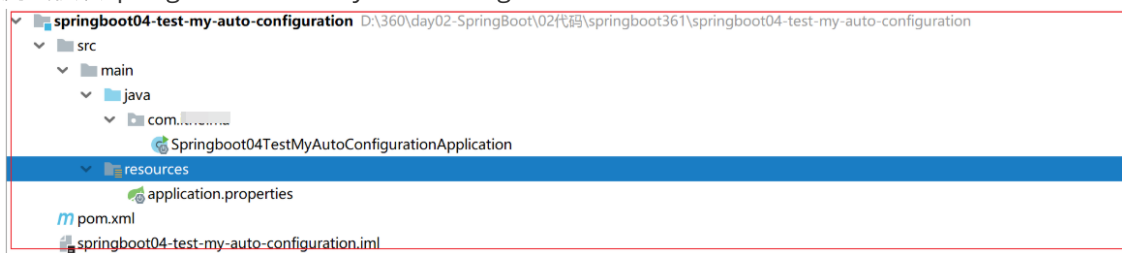
```

14         <artifactId>kkb-codeutil</artifactId>
15         <version>1.0-SNAPSHOT</version>
16     </dependency>
17     <!--框架的自动配置依赖-->
18     <dependency>
19         <groupId>com.kkb</groupId>
20         <artifactId>kkb-spring-boot-autoconfigure</artifactId>
21         <version>1.0-SNAPSHOT</version>
22     </dependency>
23 </dependencies>
24 </project>

```

4. 将自动配置模块及starter模块安装到本地仓库，执行maven的install命令

5. 测试模块springboot04-test-my-auto-configuration



◦ 引入自定义的 kkb-spring-boot-starter 依赖坐标

```

1 <!--导入人人架构的起步依赖-->
2 <dependency>
3     <groupId>cn.kkb</groupId>
4     <artifactId>kkb-spring-boot-starter</artifactId>
5     <version>1.0-SNAPSHOT</version>
6 </dependency>

```

◦ 测试核心代码生成器对象，是否自动配置成功了



9.4 切换内置的web应用服务器

前置条件，必须会使用Maven helper的插件，懂得依赖排除

SpringBoot的web环境中默认使用tomcat作为内置服务器，其实SpringBoot提供了另外2中内置服务器供我们选择，我们可以很方便的进行切换。

Jetty: Jetty 是一个开源的servlet容器，它为基于Java的web容器，例如JP和servlet提供运行环境。Jetty是使用[Java语言](#)编写的，它的API以一组JAR包的形式发布。

Undertow: 是红帽公司开发的一款基于 NIO 的高性能 Web 嵌入式服务器

操作过程:

1. 在spring-boot-starter-web排除tomcat

SpringApplication类的构造方法

```
1 //SpringApplication类的构造方法
2 public SpringApplication(ResourceLoader resourceLoader, Class<?>...
   primarySources) {
3     //设置资源加载器
4     this.resourceLoader = resourceLoader;
5     //判断启动引导类是否为空
6     Assert.notNull(primarySources, "PrimarySources must not be null");
7     this.primarySources = new LinkedHashSet<>
   (Arrays.asList(primarySources));
8     //判断是否是web应用
9     this.webApplicationType = webApplicationType.deduceFromClasspath();
10    //设置应用的初始化器
11    setInitializers((Collection)
   getSpringFactoriesInstances(ApplicationContextInitializer.class));
12    //设置应用的监听器
13    setListeners((Collection)
   getSpringFactoriesInstances(ApplicationListener.class));
14    //
15    this.mainApplicationClass = deduceMainApplicationClass();
16 }
```

SpringApplication类的run()方法

```
1 //SpringApplication类的run()方法
2 public ConfigurableApplicationContext run(String... args) {
3     //创建计时器，记录SpringBoot启动耗时
4     Stopwatch stopwatch = new Stopwatch();
5     stopwatch.start(); //开始计时
6     //创建Spring的IOC容器
7     ConfigurableApplicationContext context = null;
8     //SpringBoot异常报告对象
9     Collection<SpringBootExceptionHandler> exceptionReporters = new
   ArrayList<>();
10    configureHeadlessProperty();
11    //获取所有监听器
12    SpringApplicationRunListeners listeners = getRunListeners(args);
13    listeners.starting(); //执行监听器的starting方法
14    try {
15        //创建应用参数对象
16        ApplicationArguments applicationArguments = new
   DefaultApplicationArguments(args);
17        //准备应用环境
18        ConfigurableEnvironment environment = prepareEnvironment(listeners,
   applicationArguments);
19        configureIgnoreBeanInfo(environment);
20        //打印Banner内容
21        Banner printedBanner = printBanner(environment);
22        //创建Spring容器
23        context = createApplicationContext();
24        //获取异常报告的实例
25        exceptionReporters =
   getSpringFactoriesInstances(SpringBootExceptionHandler.class,
   new Class[] {
   ConfigurableApplicationContext.class }, context);
26        //准备Spring的容器
27    }
```

```

28     prepareContext(context, environment, listeners,
applicationArguments, printedBanner);
29     //刷新Spring的容器，将所有对象注入Spring容器
30     refreshContext(context);
31     afterRefresh(context, applicationArguments);
32     stopwatch.stop();//停止计时
33     if (this.logStartupInfo) {
34         new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
stopwatch);
35     }
36     //执行所有监听器的started方法
37     listeners.started(context);
38     //启动完成
39     callRunners(context, applicationArguments);
40 }
41 catch (Throwable ex) {
42     handleRunFailure(context, ex, exceptionReporters, listeners);
43     throw new IllegalStateException(ex);
44 }
45
46 try {
47     //执行监听器的running方法
48     listeners.running(context);
49 }
50 catch (Throwable ex) {
51     handleRunFailure(context, ex, exceptionReporters, null);
52     throw new IllegalStateException(ex);
53 }
54 //启动完成返回Spring的容器
55 return context;
56 }

```

10、生产级监控Actuator

10.1 Actuator简介

SpringBoot自带监控功能Actuator，可以帮助实现对程序内部运行情况监控，比如监控状况、Bean加载情况、配置属性、日志信息等。

使用步骤：

1. 导入依赖坐标

```

1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-actuator</artifactId>
4 </dependency>

```

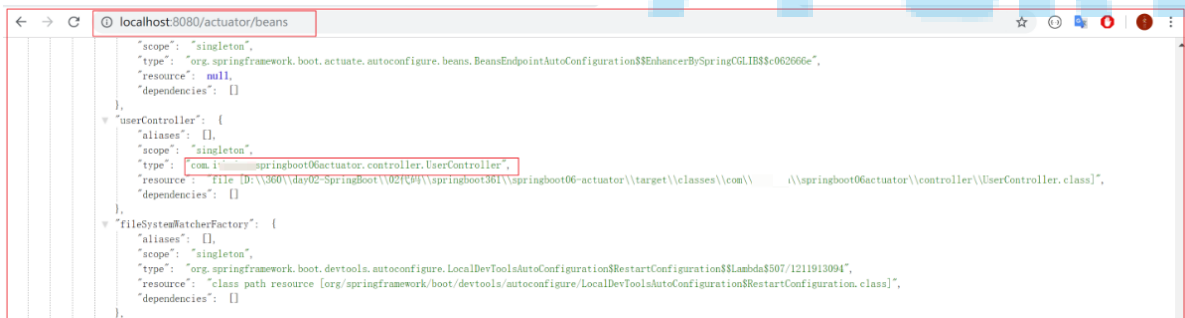
2. 访问监控地址：<http://localhost:8080/actuator>

10.2 监控应用endpoint

路径	描述	默认开启
/beans	显示容器的全部的Bean，以及它们的关系	Y
/env	获取全部环境属性	Y
/env/{name}	根据名称获取特定的环境属性值	Y
/health	显示健康检查信息	Y
/info	显示设置好的应用信息	Y
/mappings	显示所有的@RequestMapping信息	Y
/metrics	显示应用的度量信息	Y
/scheduledtasks	显示任务调度信息	Y
/httptrace	显示Http Trace信息	Y
/caches	显示应用中的缓存	Y
/conditions	显示配置条件的匹配情况	Y
/configprops	显示@ConfigurationProperties的信息	Y
/loggers	显示并更新日志配置	Y
/shutdown	关闭应用程序	N
/threaddump	执行ThreadDump	Y
/headdump	返回HeadDump文件，格式为HPROF	Y
/prometheus	返回可供Prometheus抓取的信息	Y

举例：

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

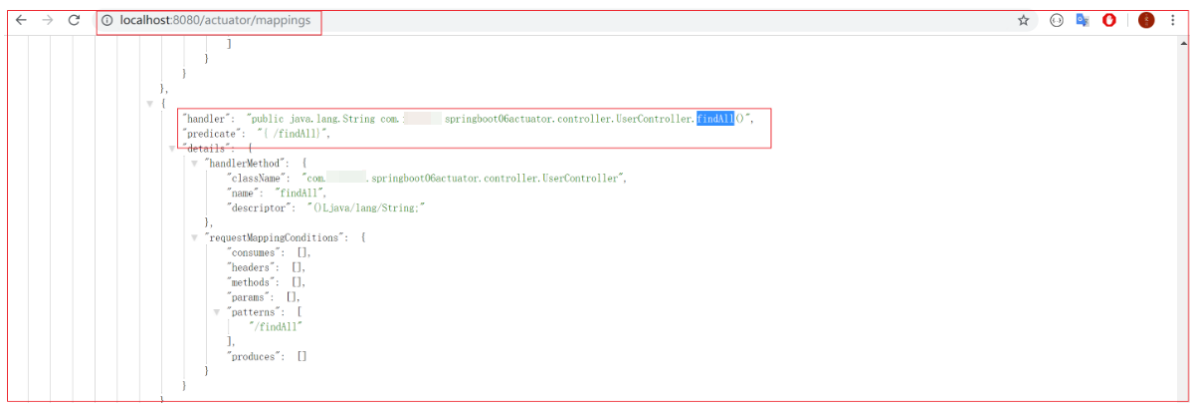


```

{
  "scope": "singleton",
  "type": "org.springframework.boot.actuate.autoconfigure.beans.BeaEndpointAutoConfiguration$EnhancerBySpringGLIB$c062606e",
  "resource": null,
  "dependencies": []
},
{
  "scope": "singleton",
  "type": "com.intellij.springboot06actuator.controller.UserController",
  "resource": "File [D:\\360\\day02-SpringBoot\\021\\09\\springboot361\\springboot06-actuator\\target\\classes\\com\\...\\springboot06actuator\\controller\\UserController.class]",
  "dependencies": []
},
{
  "scope": "singleton",
  "type": "org.springframework.boot.devtools.autoconfigure.LocalDevToolsAutoConfiguration$RestartConfiguration$Lambda$507/1211913094",
  "resource": "class path resource [org.springframework.boot.devtools.autoconfigure.LocalDevToolsAutoConfiguration$RestartConfiguration.class]",
  "dependencies": []
}

```

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



10.3 配置

```
1 # 暴露所有的监控点
2 management.endpoints.web.exposure.include=*
3 # 定义Actuator访问路径
4 management.endpoints.web.base-path=/act
5 # 开启endpoint 关闭服务功能
6 management.endpoint.shutdown.enabled=true
7
8 # 连接SpringBoot的admin
9 spring.boot.admin.client.url=http://localhost:9000
```

10.4 通过Spring Boot Admin了解程序的运行状态

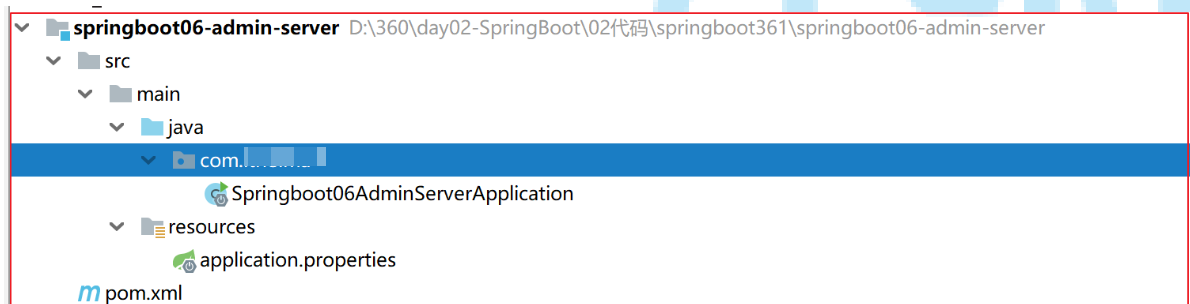
目的：为SpringBoot应用程序提供一套管理界面

主要功能：

- 集中展示应用程序Actuator相关的内容
- 变更通知

搭建服务端：

①创建 admin-server 模块



②导入依赖坐标 admin-starter-server

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5     http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <parent>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-parent</artifactId>
```



```

8         <version>2.1.9.RELEASE</version>
9         <relativePath/> <!-- lookup parent from repository -->
10    </parent>
11    <groupId>com.abc</groupId>
12    <artifactId>springboot06-admin-server</artifactId>
13    <version>0.0.1-SNAPSHOT</version>
14    <name>springboot06-admin-server</name>
15    <description>Demo project for Spring Boot</description>
16
17    <properties>
18        <java.version>1.8</java.version>
19        <spring-boot-admin.version>2.1.5</spring-boot-admin.version>
20    </properties>
21
22    <dependencies>
23        <!--导入spring boot admin 服务端的启动依赖坐标
24        注意：必须导入其依赖管理清单bom
25        -->
26        <dependency>
27            <groupId>de.codecentric</groupId>
28            <artifactId>spring-boot-admin-starter-server</artifactId>
29        </dependency>
30    </dependencies>
31    <!--spring boot admin的依赖管理清单-->
32    <dependencyManagement>
33        <dependencies>
34            <dependency>
35                <groupId>de.codecentric</groupId>
36                <artifactId>spring-boot-admin-dependencies</artifactId>
37                <version>${spring-boot-admin.version}</version>
38                <type>pom</type>
39                <scope>import</scope>
40            </dependency>
41        </dependencies>
42    </dependencyManagement>
43 </project>
44

```

③在引导类上启用监控功能@EnableAdminServer

```

1  @SpringBootApplication
2  @EnableAdminServer
3  public class Springboot06AdminServerApplication {
4
5      public static void main(String[] args) {
6          SpringApplication.run(Springboot06AdminServerApplication.class,
7          args);
8      }
9  }

```

配置客户端：

①创建 admin-client 模块

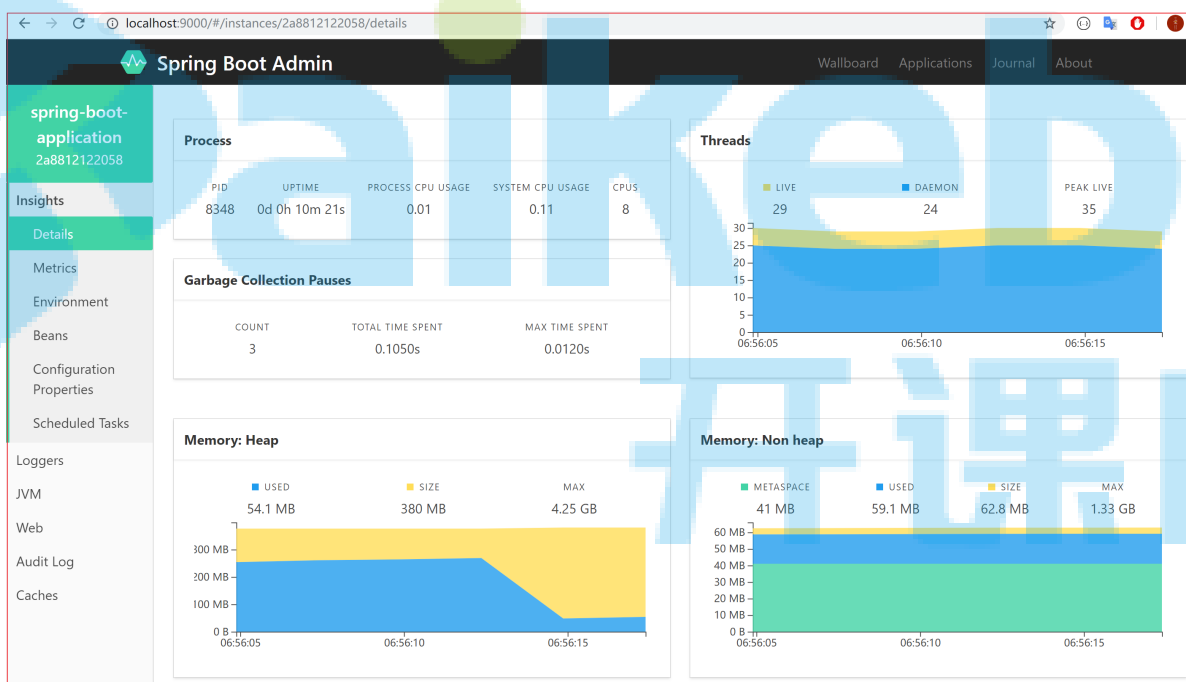
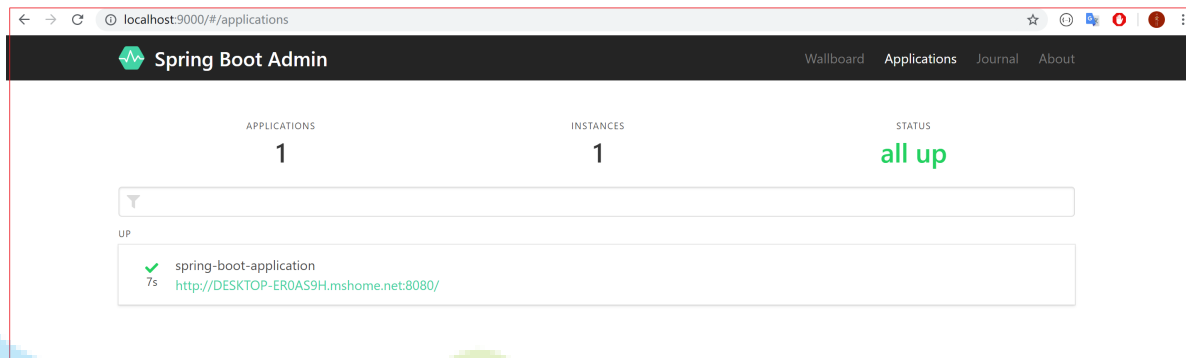
②导入依赖坐标 admin-starter-client

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>org.springframework.boot</groupId>
7         <artifactId>spring-boot-starter-parent</artifactId>
8         <version>2.1.9.RELEASE</version>
9         <relativePath/> <!-- lookup parent from repository -->
10    </parent>
11    <groupId>com.abc</groupId>
12    <artifactId>springboot06-actuator</artifactId>
13    <version>0.0.1-SNAPSHOT</version>
14    <name>springboot06-actuator</name>
15    <description>Demo project for Spring Boot</description>
16    <dependencies>
17
18        <!--spring boot admin 客户端依赖坐标-->
19        <dependency>
20            <groupId>de.codecentric</groupId>
21            <artifactId>spring-boot-admin-starter-client</artifactId>
22        </dependency>
23        <!--SpringBoot的生产级监控 Actuator依赖坐标-->
24        <dependency>
25            <groupId>org.springframework.boot</groupId>
26            <artifactId>spring-boot-starter-actuator</artifactId>
27        </dependency>
28
29        <dependency>
30            <groupId>org.springframework.boot</groupId>
31            <artifactId>spring-boot-starter-web</artifactId>
32        </dependency>
33    </dependencies>
34    <!--spring boot admin的bom-->
35    <dependencyManagement>
36        <dependencies>
37            <dependency>
38                <groupId>de.codecentric</groupId>
39                <artifactId>spring-boot-admin-dependencies</artifactId>
40                <version>2.1.5</version>
41                <type>pom</type>
42                <scope>import</scope>
43            </dependency>
44        </dependencies>
45    </dependencyManagement>
46 </project>
```

③配置相关信息：server地址等

```
1 # 暴露所有的监控点
2 management.endpoints.web.exposure.include=*
3 # 定义Actuator访问路径
4 management.endpoints.web.base-path=/act
5 # 开启endpoint 关闭服务功能
6 management.endpoint.shutdown.enabled=true
7
8 # 连接SpringBoot的admin
9 spring.boot.admin.client.url=http://localhost:9000
```

④启动server和client服务，访问server



总结

- 掌握工程热部署
- 掌握多环境的配置文件
- 掌握配置文件存放路径，及其加载顺序
- 掌握自定义配置文件名称application.yml
- 了解SpringBoot自动配置实现原理【不要求掌握】
- 掌握内置web应用服务器的切换：tomcat切换为jetty
- 了解为SpringBoot配置生产级监控Actuator
- 搭建Spring Boot Admin服务链接SpringBoot项目

