SpringBoot【一】

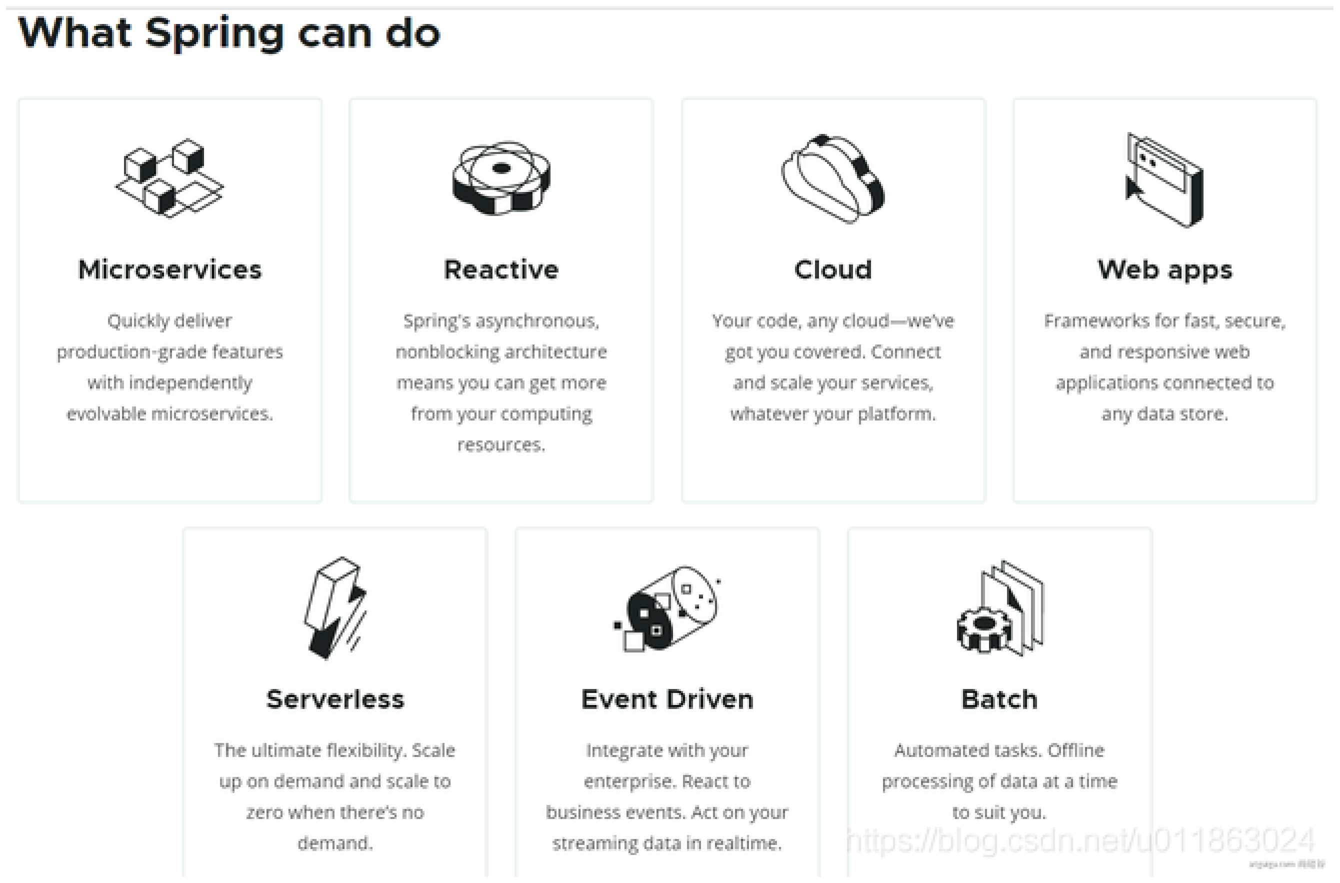
01、基础入门-SpringBoot2课程介绍

1. Spring Boot 2核心技术
2. Spring Boot 2响应式编程

- 学习要求
  -熟悉Spring基础
  -熟悉Maven使用
- 环境要求
  - Java8及以上
  - Maven 3.3及以上
- 学习资料
  - Spring Boot官网
  - Spring Boot官方文档
  - 本课程文档地址
  - 视频地址1、视频地址2
  - 源码地址

02、基础入门-Spring生态圈

一、Spring能做什么



What Spring can do

Microservices
Quickly deliver production-grade features with independently evolvable microservices.

Reactive
Spring's asynchronous, nonblocking architecture means you can get more from your computing resources.

Cloud
Your code, any cloud—we've got you covered. Connect and scale your services, whatever your platform.

Web apps
Frameworks for fast, secure, and responsive web applications connected to any data store.

Serverless
The ultimate flexibility. Scale up on demand and scale to zero when there's no demand.

Event Driven
Integrate with your enterprise. React to business events. Act on your streaming data in realtime.

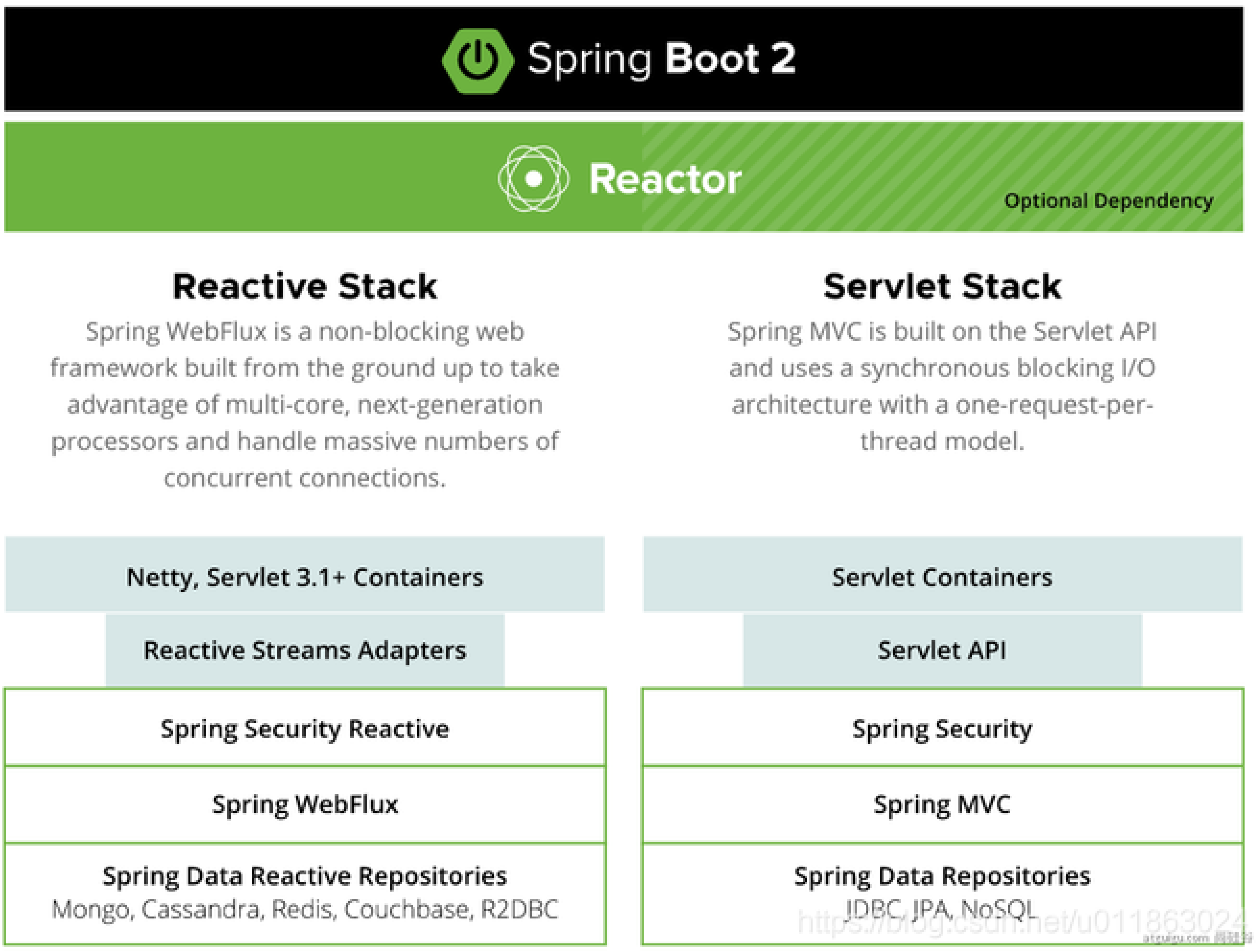
Batch
Automated tasks. Offline processing of data at a time to suit you.

二、Spring的生态

- web开发
- 数据访问
- 安全控制
- 分布式
- 消息服务
- 移动开发
- 批处理
- ......

三、Spring5重大升级

- 响应式编程


Spring Boot 2
Reactor
Optional Dependency
Reactive Stack
Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.
Servlet Stack
Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.
Netty, Servlet 3.1+ Containers
Reactive Streams Adapters
Spring Security Reactive
Spring WebFlux
Spring Data Reactive Repositories
Mongo, Cassandra, Redis, Couchbase, R2DBC
Servlet Containers
Servlet API
Spring Security
Spring MVC
Spring Data Repositories
JDBC, JPA, NoSQL

- 内部源码设计

基于Java8的一些新特性，如：接口默认实现。重新设计源码架构。

四、为什么用SpringBoot

> Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".link
>
> 能快速创建出生产级别的Spring应用。

SpringBoot优点

- Create stand-alone Spring applications
  - 创建独立Spring应用
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
  - 内嵌web服务器
- Provide opinionated 'starter' dependencies to simplify your build configuration
  - 自动starter依赖，简化构建配置
- Automatically configure Spring and 3rd party libraries whenever possible
  - 自动配置Spring以及第三方功能
- Provide production-ready features such as metrics, health checks, and externalized configuration
  - 提供生产级别的监控、健康检查及外部化配置
- Absolutely no code generation and no requirement for XML configuration
  - 无代码生成、无需编写XML

- SpringBoot是整合Spring技术栈的一站式框架
- SpringBoot是简化Spring技术栈的快速开发脚手架

SpringBoot缺点

- 人称版本帝，迭代快，需要时刻关注变化
- 封装太深，内部原理复杂，不容易精通

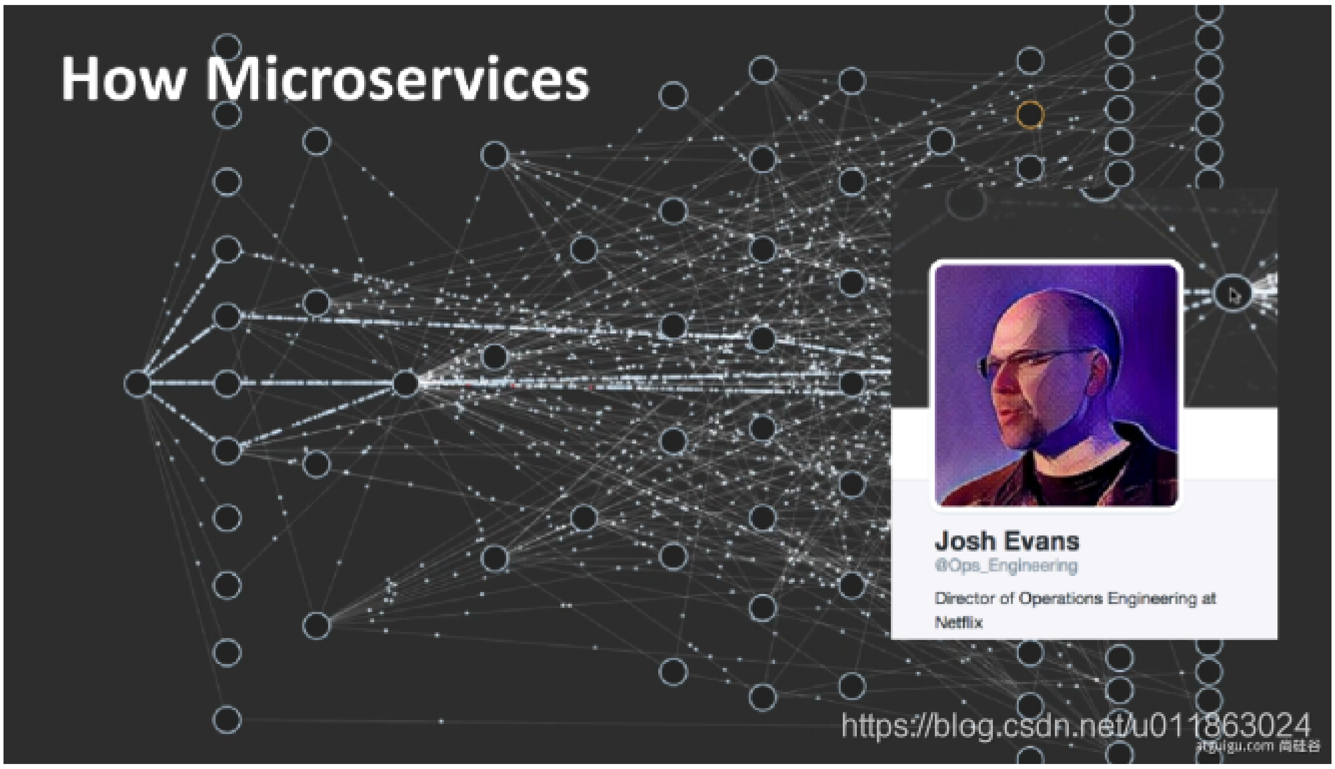
03、基础入门-SpringBoot的大时代背景

一、微服务

> In short, the **microservice architectural style** is an approach to developing a single application as a **suite of small services**, each **running in its own process** and communicating with **lightweight** mechanisms, often an **HTTP** resource API. These services are built around **business capabilities** and **independently deployable** by fully **automated deployment** machinery. There is a bare minimum of centralized management of these services, which may be **written in different programming languages** and use different data storage technologies.——James Lewis and Martin Fowler (2014)

- 微服务是一种架构风格
- 一个应用拆分为一组小型服务
- 每个服务运行在自己的进程内，也就是可独立部署和升级
- 服务之间使用轻量级HTTP交互
- 服务围绕业务功能拆分
- 可以由全自动部署机制独立部署
- 去中心化，服务自治。服务可以使用不同的语言、不同的存储技术

二、分布式

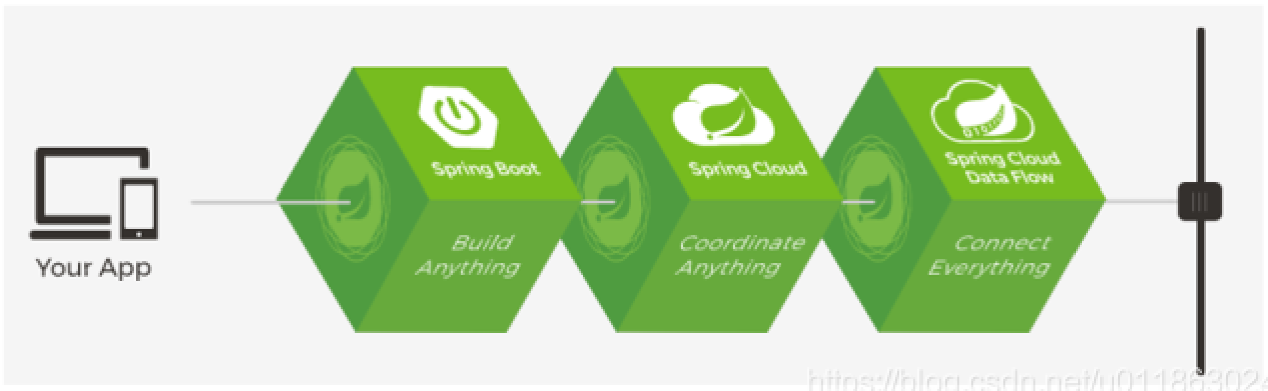

How Microservices
Josh Evans
@Ops_Engineering
Director of Operations Engineering at Netflix
https://blog.csdn.net/u011863024
atguigu.com 尚硅谷

分布式的困难

- 远程调用
- 服务发现
- 负载均衡
- 服务容错
- 配置管理
- 服务监控
- 链路追踪
- 日志管理
- 任务调度
- ……

分布式的解决

- SpringBoot + SpringCloud



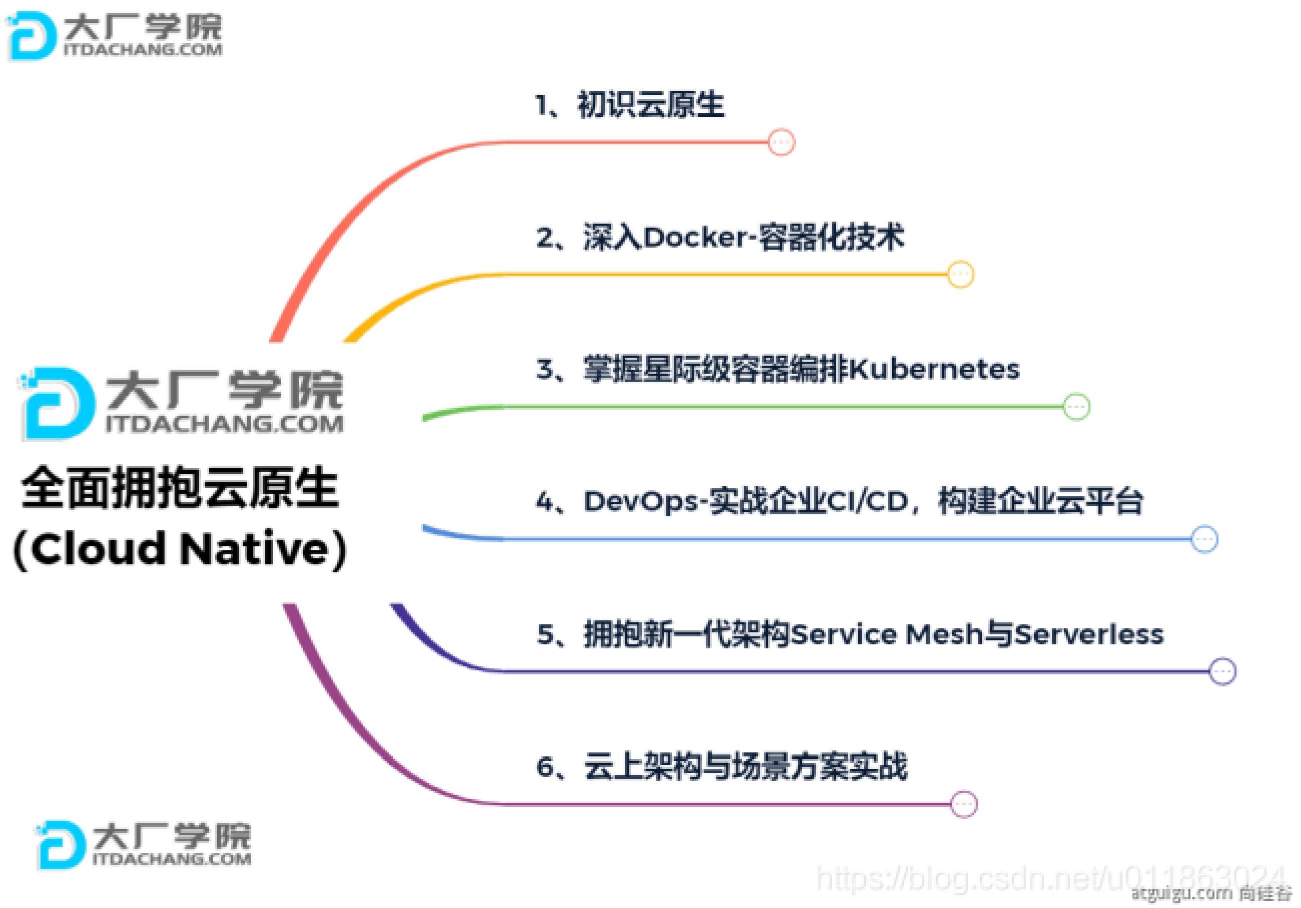
三、云原生

原生应用如何上云。 Cloud Native

上云的困难

- 服务自愈
- 弹性伸缩
- 服务隔离
- 自动化部署
- 灰度发布
- 流量治理
- ……

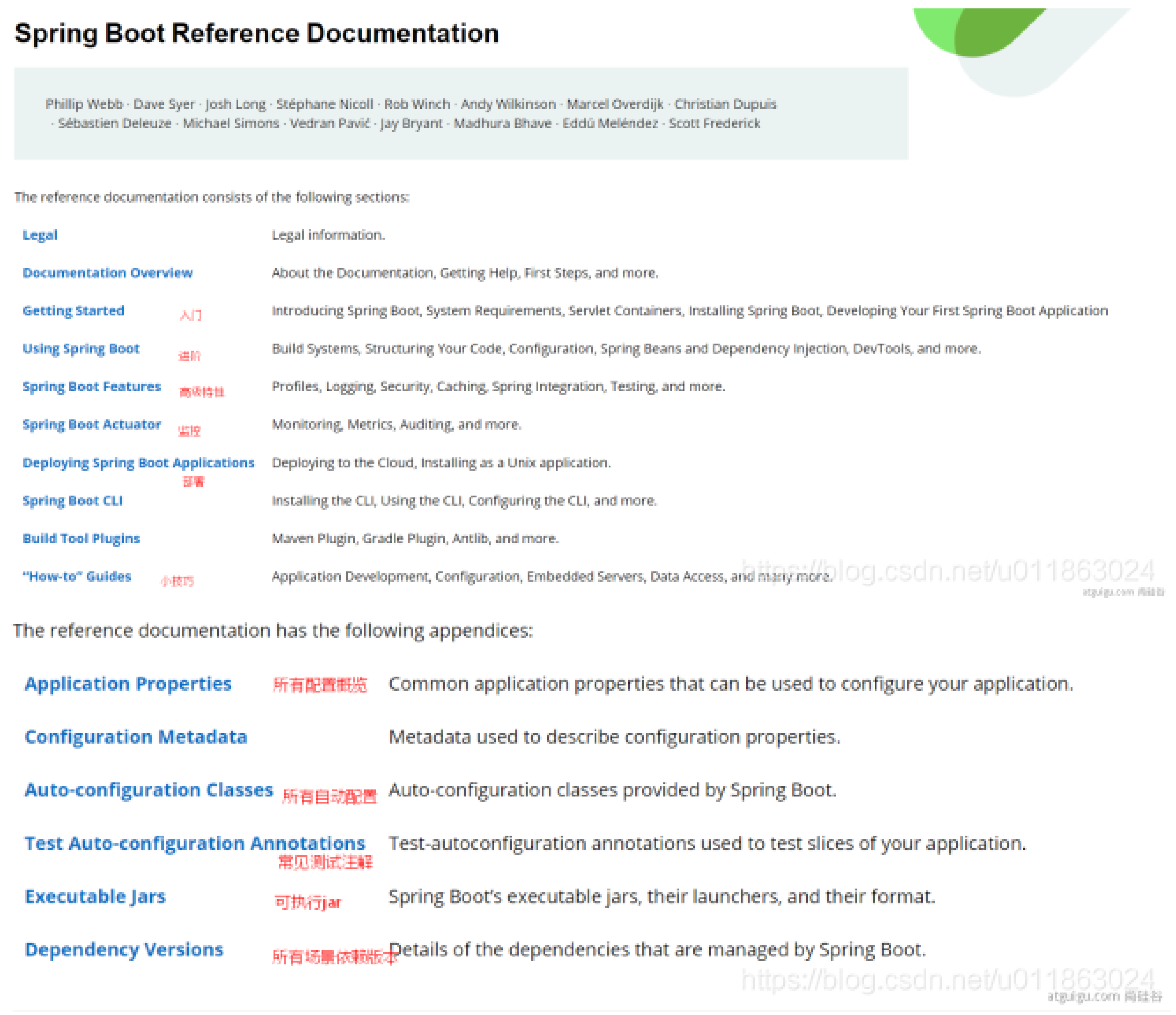
上云的解决


大厂学院 ITDACHANG.COM
全面拥抱云原生 (Cloud Native)
1、初识云原生
2、深入Docker-容器化技术
3、掌握星际级容器编排Kubernetes
4、DevOps-实战企业CI/CD，构建企业云平台
5、拥抱新一代架构Service Mesh与Serverless
6、云上架构与场景方案实战
https://blog.csdn.net/u011863024
atguigu.com 尚硅谷

04、基础入门-SpringBoot官方文档架构

- [Spring Boot官网]
- [Spring Boot官方文档]

官网文档架构



Spring Boot Reference Documentation

Phillip Webb · Dave Syer · Josh Long · Stéphane Nicoll · Rob Winch · Andy Wilkinson · Marcel Overdijk · Christian Dupuis · Sébastien Deleuze · Michael Simons · Vedran Pavić · Jay Bryant · Madhura Bhave · Eddú Meléndez · Scott Frederick

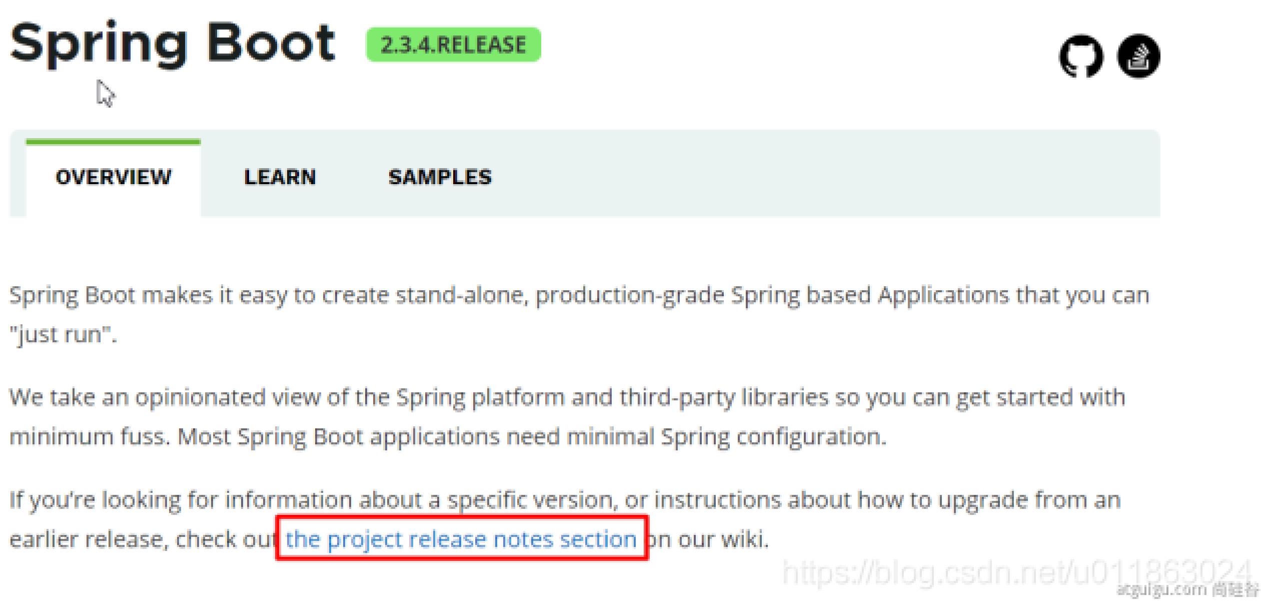
The reference documentation consists of the following sections:

Legal		Legal information.
Documentation Overview		About the Documentation, Getting Help, First Steps, and more.
Getting Started	入门	Introducing Spring Boot, System Requirements, Servlet Containers, Installing Spring Boot, Developing Your First Spring Boot Application
Using Spring Boot	进阶	Build Systems, Structuring Your Code, Configuration, Spring Beans and Dependency Injection, DevTools, and more.
Spring Boot Features	高级特性	Profiles, Logging, Security, Caching, Spring Integration, Testing, and more.
Spring Boot Actuator	监控	Monitoring, Metrics, Auditing, and more.
Deploying Spring Boot Applications	部署	Deploying to the Cloud, Installing as a Unix application.
Spring Boot CLI		Installing the CLI, Using the CLI, Configuring the CLI, and more.
Build Tool Plugins		Maven Plugin, Gradle Plugin, Antlib, and more.
"How-to" Guides	小技巧	Application Development, Configuration, Embedded Servers, Data Access, and many more.

The reference documentation has the following appendices:

Application Properties	所有配置概览	Common application properties that can be used to configure your application.
Configuration Metadata		Metadata used to describe configuration properties.
Auto-configuration Classes	所有自动配置	Auto-configuration classes provided by Spring Boot.
Test Auto-configuration Annotations	常见测试注解	Test-autoconfiguration annotations used to test slices of your application.
Executable Jars	可执行jar	Spring Boot's executable jars, their launchers, and their format.
Dependency Versions	所有场景依赖版本	Details of the dependencies that are managed by Spring Boot.

[查看版本新特性]



Spring Boot 2.3.4.RELEASE

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need minimal Spring configuration.

If you're looking for information about a specific version, or instructions about how to upgrade from an earlier release, check out the project release notes section on our wiki.

05、基础入门-SpringBoot--打印HelloWorld

一、系统要求

- Java 8
- Maven 3.3+
- IntelliJ IDEA 2019.1.2

针对于SpringBoot的Maven配置文件【是针对maven本身的xml文件进行配置】

```xml
<mirrors>
    <mirror>
        <id>nexus-aliyun</id>
        <mirrorOf>central</mirrorOf>
        <name>Nexus aliyun</name>
        <url>http://maven.aliyun.com/nexus/content/groups/public</url>
    </mirror>
</mirrors>

<profiles>
    <profile>
        <id>jdk-1.8</id>

        <activation>
            <activeByDefault>true</activeByDefault>
            <jdk>1.8</jdk>
        </activation>

        <properties>
            <maven.compiler.source>1.8</maven.compiler.source>
            <maven.compiler.target>1.8</maven.compiler.target>
            <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
        </properties>
    </profile>
</profiles>
```

二、HelloWorld项目

需求：浏览发送/hello请求，响应 “Hello，Spring Boot 2”

1. 创建maven工程【新建maven项目】

2. 引入依赖【在pom文件中导入】

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.4.RELEASE</version>
</parent>

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

3. 创建主程序【引导SpringBoot的启动】

意义在于指明这是一个SpringBoot项目。

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

@SpringBootApplication
public class MainApplication {

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

```

4. 编写业务【自己的业务逻辑】

```java
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String handle01(){
        return "Hello, Spring Boot 2!";
    }
}
```

5. 运行&测试

- 运行`MainApplication`类
- 浏览器输入`http://localhost:8888/hello`，将会输出`Hello, Spring Boot 2!`。

6. 设置配置

maven工程的resource文件夹中创建application.properties文件。

```properties
# 设置端口号
server.port=8888
```

更多配置信息

7. 打包部署

在pom.xml添加

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

在IDEA的Maven插件上点击运行 clean 、package，把helloworld工程项目的打包成jar包，

打包好的jar包被生成在helloworld工程项目的target文件夹内。

该jar包可以直接执行。

用cmd运行`java -jar boot-01-helloworld-1.0-SNAPSHOT.jar`，既可以运行helloworld工程项目。

将jar包直接在目标服务器执行即可。

06、基础入门-SpringBoot-依赖管理特性

- 父项目做依赖管理

```xml
<!--依赖管理-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.4.RELEASE</version>
</parent>

<!--上面项目的父项目如下：-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>2.3.4.RELEASE</version>
</parent>

<!--它几乎声明了所有开发中常用的依赖的版本号，自动版本仲裁机制-->
```

- 开发导入starter场景启动器——`starter`场景启动器
    1. 见到很多 spring-boot-starter-* ： *就某种场景
    2. 只要引入starter，这个场景的所有常规需要的依赖我们都自动引入
    3. 更多SpringBoot所有支持的场景
    4. 见到的 *-spring-boot-starter： 第三方为我们提供的简化开发的场景启动器。
    5. 所有场景启动器最底层的依赖：

    ```xml
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <version>2.3.4.RELEASE</version>
        <scope>compile</scope>
    </dependency>
    ```

- 无需关注版本号，自动版本仲裁
    1. 引入依赖默认都可以不写版本
    2. 引入非版本仲裁的jar，要写版本号。
- 可以修改默认版本号
    1. 查看spring-boot-dependencies里面规定当前依赖的版本用的 key。
    2. 在当前项目里面重写配置，如下面的代码。

```xml
<properties>
    <mysql.version>5.1.43</mysql.version>
</properties>
```

IDEA快捷键：

- `ctrl + shift + alt + U`：以图的方式显示项目中依赖之间的关系。
- `alt + ins`：相当于Eclipse的 Ctrl + N，创建新类，新包等。

07、基础入门-SpringBoot-自动配置特性

- 自动配好Tomcat
  - 引入Tomcat依赖。
  - 配置Tomcat

    ```xml
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <version>2.3.4.RELEASE</version>
        <scope>compile</scope>
    </dependency>
    ```

- 自动配好SpringMVC
  - 引入SpringMVC全套组件

    ```xml
        <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-webmvc</artifactId>
          <version>5.2.9.RELEASE</version>
          <scope>compile</scope>
        </dependency>
    ```

  - 自动配好SpringMVC常用组件（功能）
- 自动配好Web常见功能，如：字符编码问题
  - SpringBoot帮我们配置好了所有web开发的常见场景（Dispatcher、文件上传解析器、Filter、字符编码）
- 默认的包结构
  - **主程序所在包及其下面的所有子包里面的组件都会被默认扫描进来**
    - 如下的例子中，主程序是MainApplication，所以boot包下面的全部内容都可以扫描
    - otherboot文件夹下面的组件和otherApplication程序不能被扫描

    > |----boot文件夹（主程序所在的文件夹）
    >
    > |----controller文件夹
    >
    > |----MainApplication程序（主程序）
    >
    > |----otherboot文件夹
    >
    > |----otherApplication程序

  - 无需以前的包扫描配置
  - 想要改变扫描路径

- @SpringBootApplication(scanBasePackages="com.atguigu")

  在主程序的入口，将包扫描的范围进行手动指定
- @ComponentScan 指定扫描路径

```
@SpringBootApplication
等同于
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan("com.atguigu.boot") // 默认
```

- 各种配置拥有默认值
  - 默认配置最终都是映射到某个类上，如：`MultipartProperties`
  - 配置文件的值最终会绑定每个类上，这个类会在容器中创建对象
- 按需加载所有自动配置项
  - 非常多的starter
  - 引入了哪些场景这个场景的自动配置才会开启，自动配置也是按需加载
  - SpringBoot所有的自动配置功能都在 `spring-boot-autoconfigure` 包里面

08、底层注解-@Configuration详解

- 基本使用
  - Full模式与Lite模式
  - 示例

```java
/**
 * 1、配置类里面使用@Bean标注在方法上给容器注册组件，默认也是单实例的
 * 2、配置类本身也是组件
 * 3、proxyBeanMethods：代理bean的方法
 *      Full(proxyBeanMethods = true)（保证每个@Bean方法被调用多少次返回的组件都是单实例的）（默认）
 *      Lite(proxyBeanMethods = false)（每个@Bean方法被调用多少次返回的组件都是新创建的）
 */
@Configuration(proxyBeanMethods = false) //告诉SpringBoot这是一个配置类 == 配置文件
public class MyConfig {

    /**
     * Full:外部无论对配置类中的这个组件注册方法调用多少次获取的都是之前注册容器中的单实例对象
     * @return
     */
    @Bean //给容器中添加组件。以方法名作为组件的id。返回类型就是组件类型。返回的值，就是组件在容器中的实例
    public User user01(){
        User zhangsan = new User("zhangsan", 18);
        //user组件依赖了Pet组件
        zhangsan.setPet(tomcatPet());
        return zhangsan;
    }

    @Bean("tom")
    public Pet tomcatPet(){
        return new Pet("tomcat");
    }
}
```

@Configuration测试代码如下:

```java
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan("com.atguigu.boot")
public class MainApplication {

    public static void main(String[] args) {
    //1、返回我们IOC容器
        ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class, args);

    //2、查看容器里面的组件
        String[] names = run.getBeanDefinitionNames();
        for (String name : names) {
            System.out.println(name);
        }

    //3、从容器中获取组件
        Pet tom01 = run.getBean("tom", Pet.class);
        Pet tom02 = run.getBean("tom", Pet.class);
        System.out.println("组件："+(tom01 == tom02));

    //4、com.atguigu.boot.config.MyConfig$$EnhancerBySpringCGLIB$$51f1e1ca@1654a892
        MyConfig bean = run.getBean(MyConfig.class);
        System.out.println(bean);

    //如果@Configuration(proxyBeanMethods = true)代理对象调用方法。SpringBoot总会检查这个组件是否在容器中有。
        //保持组件单实例
        User user = bean.user01();
        User user1 = bean.user01();
        System.out.println(user == user1);

        User user01 = run.getBean("user01", User.class);
        Pet tom = run.getBean("tom", Pet.class);

        System.out.println("用户的宠物："+(user01.getPet() == tom));
    }
}
```

- 最佳实战
  - 配置 类组件之间**无依赖关系**用Lite模式加速容器启动过程，减少判断【lite=非单例】
  - 配置 类组件之间**有依赖关系**，方法会被调用得到之前单实例组件，用Full模式（默认）【full=单例】

IDEA快捷键:

- `Alt + Ins`:生成getter，setter、构造器等代码。
- `Ctrl + Alt + B`:查看类的具体实现代码。

09、底层注解-@Import导入组件

@Bean、@Component、@Controller、@Service、@Repository，它们是Spring的基本标签，在Spring Boot中并未改变它们原来的功能。

@ComponentScan 在【07、基础入门-SpringBoot-自动配置特性】有用例。

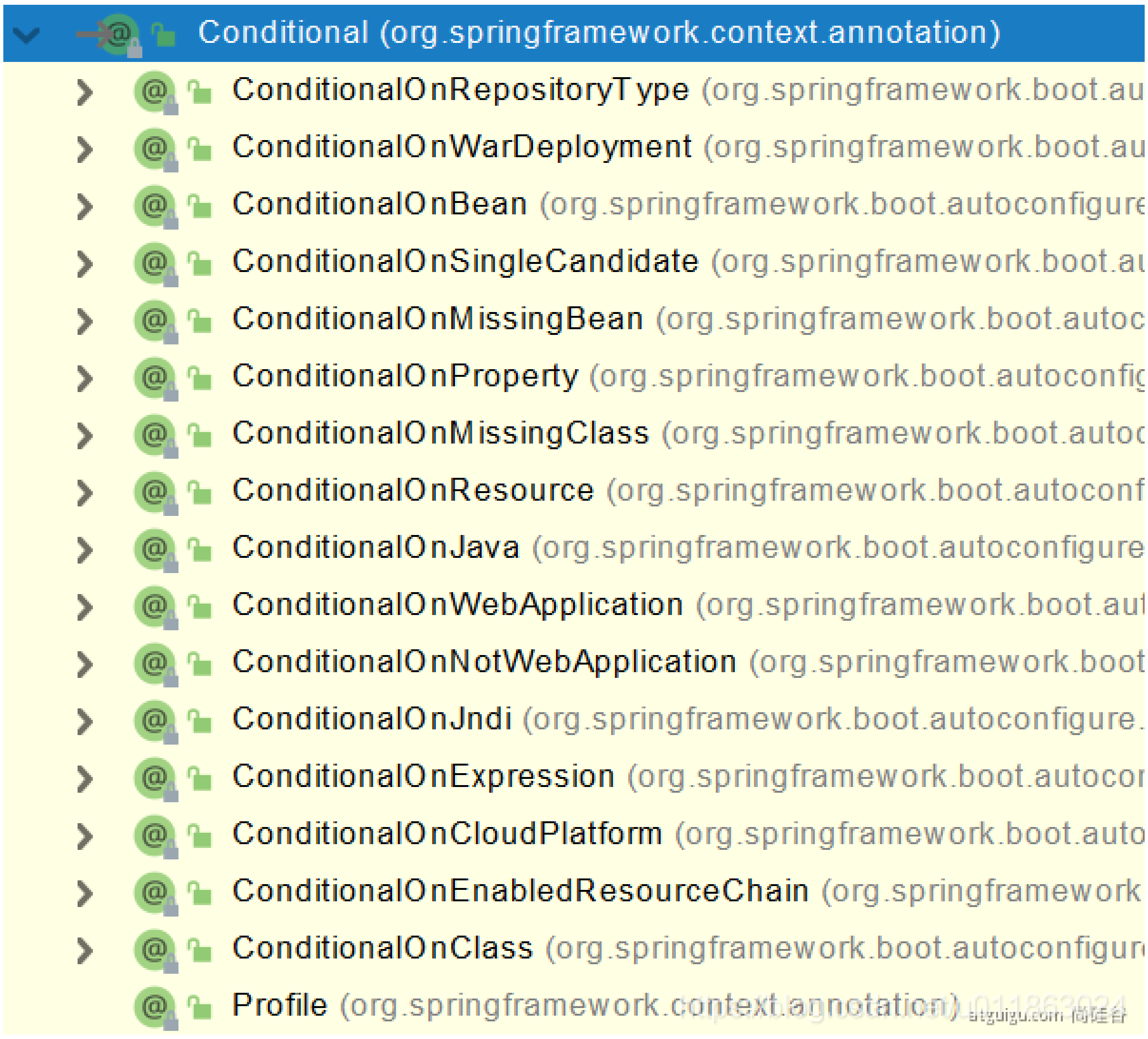
@Import({User.class, DBHelper.class})给容器中**自动创建出这两个类型的组件**、默认组件的名字就是全类名

```java
@Import({User.class, DBHelper.class}) // 通过使用这个注解，给容器中自动创建了这两个类型的组件
@Configuration(proxyBeanMethods = false) //告诉SpringBoot这是一个配置类 == 配置文件
public class MyConfig {
}
```

10、底层注解-@Conditional条件装配

条件装配：满足Conditional指定的条件，则进行组件注入

- `Conditional`下面包括如下的一些具体注解



Conditional (org.springframework.context.annotation)
ConditionalOnRepositoryType (org.springframework.boot.au
ConditionalOnWarDeployment (org.springframework.boot.au
ConditionalOnBean (org.springframework.boot.autoconfigure
ConditionalOnSingleCandidate (org.springframework.boot.au
ConditionalOnMissingBean (org.springframework.boot.autoc
ConditionalOnProperty (org.springframework.boot.autoconfig
ConditionalOnMissingClass (org.springframework.boot.autoc
ConditionalOnResource (org.springframework.boot.autoconf
ConditionalOnJava (org.springframework.boot.autoconfigure
ConditionalOnWebApplication (org.springframework.boot.aut
ConditionalOnNotWebApplication (org.springframework.boot
ConditionalOnJndi (org.springframework.boot.autoconfigure.
ConditionalOnExpression (org.springframework.boot.autocor
ConditionalOnCloudPlatform (org.springframework.boot.auto
ConditionalOnEnabledResourceChain (org.springframework
ConditionalOnClass (org.springframework.boot.autoconfigur
Profile (org.springframework.context.annotation)

用@ConditionalOnMissingBean举例说明（当容器中没有名字为tom的bean时会执行）

```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(name = "tom")//没有tom名字的Bean时，MyConfig类的Bean才能生效。
public class MyConfig {

    @Bean
    public User user01(){
        User zhangsan = new User("zhangsan", 18);
        zhangsan.setPet(tomcatPet());
        return zhangsan;
    }

    @Bean("tom22")
    public Pet tomcatPet(){
        return new Pet("tomcat");
    }
}

public static void main(String[] args) {
    //1、返回我们IOC容器
    ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class, args);

    //2、查看容器里面的组件
    String[] names = run.getBeanDefinitionNames();
    for (String name : names) {
        System.out.println(name);
    }

    boolean tom = run.containsBean("tom");
    System.out.println("容器中Tom组件："+tom);//false

    boolean user01 = run.containsBean("user01");
    System.out.println("容器中user01组件："+user01);//true

    boolean tom22 = run.containsBean("tom22");
    System.out.println("容器中tom22组件："+tom22);//true

}
```

11、底层注解-@ImportResource导入Spring配置文件

importResource的诞生方便了继续复用bean.xml配置文件。

比如，公司使用bean.xml文件生成配置bean，然而你为了省事，想继续复用bean.xml，@ImportResource粉墨登场。

bean.xml：

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

    <bean id="haha" class="com.lun.boot.bean.User">
        <property name="name" value="zhangsan"></property>
        <property name="age" value="18"></property>
    </bean>

    <bean id="hehe" class="com.lun.boot.bean.Pet">
        <property name="name" value="tomcat"></property>
    </bean>
</beans>
```

使用方法：

```java
@ImportResource("classpath:beans.xml")
public class MyConfig {
...
}
```

测试类：

```java
public static void main(String[] args) {
    //1、返回我们IOC容器
    ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class, args);

    boolean haha = run.containsBean("haha");
    boolean hehe = run.containsBean("hehe");
    System.out.println("haha："+haha);//true
    System.out.println("hehe："+hehe);//true
}
```

12、底层注解-@ConfigurationProperties配置绑定

如何使用Java读取到properties文件中的内容，并且把它封装到JavaBean中，以供随时使用

一、传统方法：

```java
public class getProperties {
     public static void main(String[] args) throws FileNotFoundException, IOException {
         Properties pps = new Properties();
         pps.load(new FileInputStream("a.properties"));
         Enumeration enum1 = pps.propertyNames();//得到配置文件的名字
         while(enum1.hasMoreElements()) {
             String strKey = (String) enum1.nextElement();
             String strValue = pps.getProperty(strKey);
             System.out.println(strKey + "=" + strValue);
             //封装到JavaBean。
         }
     }
 }
```

二、Spring Boot一种配置配置绑定：

`@ConfigurationProperties` + `@Component`

假设有配置文件`application.properties`

```properties
mycar.brand=BYD
mycar.price=100000
```

只有在容器中的组件，才会拥有`SpringBoot`提供的强大功能，所以要先使用`@Component`注解

```java
@Component
@ConfigurationProperties(prefix = "mycar") // 对应匹配配置文件中所有以mycar开头的
public class Car {
    ...
}
```

三、Spring Boot另一种配置配置绑定：

`@EnableConfigurationProperties` + `@ConfigurationProperties`

1. 开启Car配置绑定功能
2. 把这个Car这个组件自动注册到容器中

```java
@EnableConfigurationProperties(Car.class)
public class MyConfig {
    ...
}
```

```java
@ConfigurationProperties(prefix = "mycar")
public class Car {
    ...
}
```

13、自动配置【源码分析】-自动包规则原理

> @SpringBootApplication = @SpringBootConfiguration + @EnableAutoConfiguration + @ComponentScan
>
> @EnableAutoConfiguration = @AutoConfigurationPackage，@Import(AutoConfigurationImportSelector.class)

Spring Boot应用的启动类：

```java
@SpringBootApplication
public class MainApplication {

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

}
```

分析下`@SpringBootApplication`

```java
@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 {
    ...
}
```

`@SpringBootApplication` = `@SpringBootConfiguration` + `@EnableAutoConfiguration` + `@ComponentScan`

一、@SpringBootConfiguration：代表当前是一个配置类

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
    @AliasFor(
        annotation = Configuration.class
    )
    boolean proxyBeanMethods() default true;
}
```

二、@ComponentScan：指定扫描注解的范围

三、@EnableAutoConfiguration：最重要的一个注解，主要包括 `@AutoConfigurationPackage`，`@Import()`

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

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

    String[] excludeName() default {};
}
```

重点分析`@AutoConfigurationPackage`，`@Import(AutoConfigurationImportSelector.class)`。

@AutoConfigurationPackage

标签名直译为：自动配置包，指定了默认的包规则。

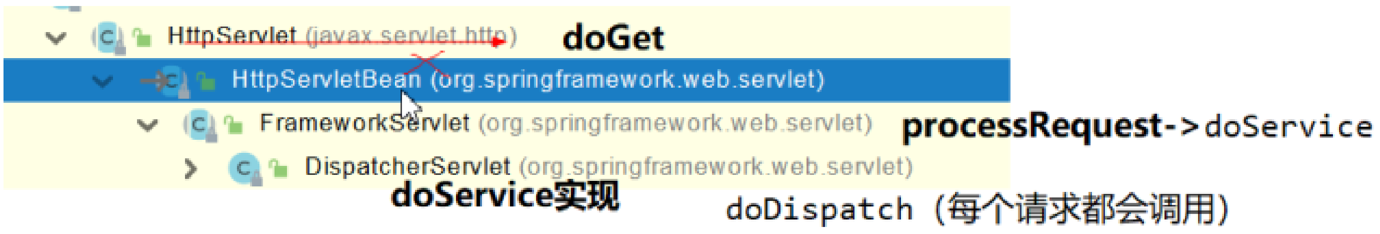
```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)//给容器中导入一个组件
public @interface AutoConfigurationPackage {
    String[] basePackages() default {};

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

1. 利用Registrar给容器中导入一系列组件
2. 将指定的一个包下的所有组件导入进MainApplication所在包下。

14、自动配置【源码分析】-初始加载自动配置类

@Import(AutoConfigurationImportSelector.class)

1. 利用`getAutoConfigurationEntry(annotationMetadata);`给容器中批量导入一些组件
2. 调用`List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes)`获取到所有需要导入到容器中的配置类
3. 利用工厂加载 `Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader);`得到所有的组件
4. 从`META-INF/spring.factories`位置来加载一个文件。
    - 默认扫描我们当前系统里面所有`META-INF/spring.factories`位置的文件
    - `spring-boot-autoconfigure-2.3.4.RELEASE.jar`包里面也有`META-INF/spring.factories`


HttpServlet (javax.servlet.http)
HttpServletBean (org.springframework.web.servlet)
FrameworkServlet (org.springframework.web.servlet)
DispatcherServlet (org.springframework.web.servlet)

```properties
# 文件里面写死了spring-boot一启动就要给容器中加载的所有配置类
# spring-boot-autoconfigure-2.3.4.RELEASE.jar/META-INF/spring.factories
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
...
```

虽然我们127个场景的所有自动配置启动的时候默认全部加载，但是`xxxxAutoConfiguration`按照条件装配规则（`@Conditional`），最终会按需配置。

如`AopAutoConfiguration`类：

```java
@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnProperty(
    prefix = "spring.aop",
    name = "auto",
    havingValue = "true",
    matchIfMissing = true
)
public class AopAutoConfiguration {
    public AopAutoConfiguration() {
    }
    ...
}
```

15、自动配置【源码分析】-自动配置流程

以`DispatcherServletAutoConfiguration`的内部类`DispatcherServletConfiguration`为例子：【Spring做了一次规范化】

```java
@Bean
@ConditionalOnBean(MultipartResolver.class)  //容器中有这个类型组件
@ConditionalOnMissingBean(name = DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME) //容器中没有这个名字 multipartResolver 的组件
public MultipartResolver multipartResolver(MultipartResolver resolver) {
    //给@Bean标注的方法传入了对象参数，这个参数的值就会从容器中找。
    //SpringMVC multipartResolver。防止有些用户配置的文件上传解析器不符合规范
    // Detect if the user has created a MultipartResolver but named it incorrectly
    return resolver;//给容器中加入了文件上传解析器；
}
```

SpringBoot默认会在底层配好所有的组件，但是**如果用户自己配置了以用户的优先**。`@ConditionalOnMissingBean`

总结：

- SpringBoot先加载所有的自动配置类 xxxxxAutoConfiguration
- 每个自动配置类按照条件进行生效，默认都会绑定配置文件指定的值。（xxxxProperties里面读取，xxxProperties和配置文件进行了绑定）
- 生效的配置类就会给容器中装配很多组件
- 只要容器中有这些组件，相当于这些功能就有了
- 定制化配置
  - 用户直接自己@Bean替换底层的组件
  - 用户去看这个组件是获取的配置文件什么值就去修改。

xxxxxAutoConfiguration ---> 组件 ---> xxxxProperties里面拿值 ----> xxxxProperties优先采纳application.properties中指定的值

16、最佳实践-SpringBoot应用如何编写

简洁版本：引入场景依赖————自动配置————手动配置（application.properties）

- 引入场景依赖【比如缓存、消息队列、配置对应的starter】
  - 官方文档
- 查看自动配置了哪些（选做）
  - 自己分析，引入场景对应的自动配置一般都生效了
  - 配置文件中debug=true开启自动配置报告。
    - Negative（不生效）
    - Positive（生效）
- 是否需要修改
  - 参照文档修改配置项
    - 官方文档
    - 自己分析。xxxxProperties绑定了配置文件的哪些。
  - 自定义加入或者替换组件
    - @Bean、@Component...
  - 自定义器 XXXXXCustomizer；
  - ......

17、最佳实践-Lombok简化开发

Lombok用标签方式代替构造器、getter/setter、toString()等鸡肋代码。

spring boot已经管理Lombok。引入依赖：

```xml
 <dependency>
     <groupId>org.projectlombok</groupId>
     <artifactId>lombok</artifactId>
</dependency>
```

IDEA中File->Settings->Plugins，搜索安装Lombok插件。

```java
@NoArgsConstructor // 无参构造器
@AllArgsConstructor // 全参构造器
@Data // 自动getset方法
@ToString // toString方法
@EqualsAndHashCode // equal方法
public class User {

    private String name;
    private Integer age;

    private Pet pet;

    public User(String name,Integer age){
        this.name = name;
        this.age = age;
    }
}
```

简化日志开发

```java
@Slf4j
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String handle01(@RequestParam("name") String name){
        log.info("请求进来了...."); // 使用.info打印对应的日志信息
        return "Hello, Spring Boot 2!"+"你好："+name;
    }
}
```

18、最佳实践-dev-tools

添加依赖：

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

在IDEA中，引入了dev-tools以后，再在项目或者页面修改以后按住`Ctrl+F9`就可以对资源进行更新，不需要再重新部署等。

19、最佳实践-Spring Initailizr

Spring Initailizr是创建Spring Boot工程向导。

在IDEA中，菜单栏New -> Project -> Spring Initailizr。

> ```
> —src
>  ├─main
>  │  ├─java
>  │  │  └─com
>  │  │      └─atguigu
>  │  │          └─boot
>  │  │              └─boot01learning
>  │  └─resources
>  │      ├─static : 存放一些静态资源
>  │      └─templates
>  └─test
>      └─java
>          └─com
>              └─atguigu
> ```

> └─boot
>   └─boot01learning

20、配置文件-yaml的用法

同以前的properties用法

YAML 是 "YAML Ain't Markup Language"（YAML 不是一种标记语言）的递归缩写。在开发的这种语言时，YAML 的意思其实是："Yet Another Markup Language"（仍是一种标记语言）。

非常适合用来做以数据为中心的配置文件。

一、基本语法

- key: value；kv之间有空格
- 大小写敏感
- 使用缩进表示层级关系
- 缩进不允许使用tab，只允许空格
- 缩进的空格数不重要，只要相同层级的元素左对齐即可
- '#'表示注释
- 字符串无需加引号，如果要加，单引号''、双引号""表示字符串内容会被 转义、不转义

二、数据类型

- 字面量：单个的、不可再分的值。date、boolean、string、number、null

```yaml
k: v
```

- 对象：键值对的集合。map、hash、set、object

```yaml
#行内写法：
k1\k2\k3是对象的属性
k: {k1:v1,k2:v2,k3:v3}

#或

k:
  k1: v1
  k2: v2
  k3: v3
```

- 数组：一组按次序排列的值。array、list、queue

```yaml
#行内写法：

k: [v1,v2,v3]

#或者

k:
 - v1
 - v2
 - v3
```

三、实例

```java
@Data
public class Person {
    private String userName;
    private Boolean boss;
    private Date birth;
    private Integer age;
    private Pet pet;
    private String[] interests;
    private List<String> animal;
    private Map<String, Object> score;
    private Set<Double> salarys;
    private Map<String, List<Pet>> allPets;
}

@Data
public class Pet {
    private String name;
    private Double weight;
}
```

用yaml表示以上对象【yaml的优先级比properties低】

```yaml
person:
  userName: zhangsan
  boss: false
  birth: 2019/12/12 20:12:33
  age: 18
  pet:
    name: tomcat
    weight: 23.4
  interests: [篮球,游泳]
  animal:
    - jerry
    - mario
  score:
    english:
      first: 30
      second: 40
      third: 50
    math: [131,140,148]
    chinese: {first: 128,second: 136}
  salarys: [3999,4999.98,5999.99]
  allPets:
    sick:
      - {name: tom}
      - {name: jerry,weight: 47}
    health: [{name: mario,weight: 47}]
```

```
// 关于单引号和双引号的问题
person:
    username: 'zhangsan \n lisi' // 使用单引号会将\n作为字符串处理，也就是说单引号会转义，会将\n应该有的换行含义进行转换

person:
    username: "zhangsan \n lisi" // 使用双引号会将\n作为换行输出，也就是双引号不会转义，就按照\n应该有的意思进行换行输出
```

21、配置文件-自定义类绑定的配置提示

自定义的类和配置文件绑定一般没有提示。若要提示，添加如下依赖：

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>

<!-- 下面插件作用是工程打包时，不将spring-boot-configuration-processor打进包内，让其只在编码的时候有用 -->
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-configuration-processor</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>
```

22、web场景-web开发简介

- web开发的大多场景我们都无需自定义配置
- SpringBoot自动配置的内容如下：
    - 内容协商视图解析器和BeanName视图解析器
    - 静态资源（包括webjars）
    - 自动注册 `Converter，GenericConverter，Formatter`
    - 支持 `HttpMessageConverters` （后来我们配合内容协商理解原理）

- 自动注册 `MessageCodesResolver` （国际化用）
- 静态index.html页支持（欢迎页）
- 自定义 `Favicon` ，网站图标
- 自动使用 `ConfigurableWebBindingInitializer` ，（DataBinder负责将请求数据绑定到JavaBean上）（数据绑定器）

> If you want to keep those Spring Boot MVC customizations and make more MVC customizations (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`.
>
> **不用@EnableWebMvc注解。使用 `@Configuration` + `WebMvcConfigurer` 自定义规则**

> If you want to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, and still keep the Spring Boot MVC customizations, you can declare a bean of type `WebMvcRegistrations` and use it to provide custom instances of those components.
>
> **声明 `WebMvcRegistrations` 改变默认底层组件**

> If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`, or alternatively add your own `@Configuration`-annotated `DelegatingWebMvcConfiguration` as described in the Javadoc of `@EnableWebMvc`.
>
> **使用 `@EnableWebMvc+@Configuration+DelegatingWebMvcConfiguration 全面接管SpringMVC`**

23、web场景-静态资源规则与定制化

一、静态资源目录

静态资源的存放目录： `/static`、 `/public` 、 `/resources`、 `/META-INF/resources`

访问： 当前项目根路径/ + 静态资源名

原理： 静态映射`/**`，拦截所有请求。

请求进来，先去找Controller看能不能处理。不能处理的所有请求又都交给静态资源处理器。静态资源也找不到则响应404页面。【先找Cintroller再找静态资源】

也可以改变默认的静态资源路径，`/static`，`/public`,`/resources`,`/META-INF/resources`全部失效

```
resources:
  static-locations: [classpath:/haha/]
```

二、静态资源访问前缀【默认是无前缀的】

项目的时候我们希望对静态资源的访问不再是全部拦截，而是带有一个前缀的部分拦截，在网页端的URL中先匹配前缀，再进行文件的搜索，这就导致了静态资源访问前缀的出现。【与静态资源本身的路径无关，路径还是自己指定的路径或默认路径】

【一部分是访问前缀】：URL的访问路径中先进行前缀匹配

【一部分是资源存放路径】：项目中静态资源的存放路径

```yaml
// yaml配置文件
spring:
  mvc:
    static-path-pattern: /res/**
```

当前项目 + static-path-pattern + 静态资源名 = 静态资源文件夹下找

三、webjar

可用jar方式添加css，js等资源文件，

https://www.webjars.org/

例如，添加jquery

```xml
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>3.5.1</version>
</dependency>
```

访问地址：http://localhost:8080/webjars/**jquery/3.5.1/jquery.js** 后面地址要按照依赖里面的包路径。

24、web场景-welcome与favicon功能

一、欢迎页支持

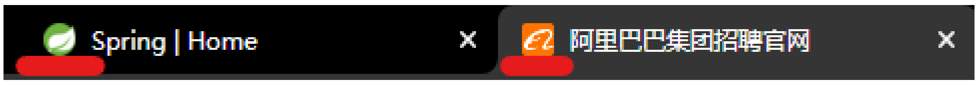
会被SpringBoot当成欢迎页面的两种书写方法：静态资源路径下的`index.html`、`Controller`中处理`/index`的`controller`方法。

- 静态资源路径下`index.html`。
  - 可以配置静态资源路径
  - 但是**不可以配置静态资源的访问前缀**。否则导致`index.html`不能被默认访问【欢迎页的作用就是默认访问，配置了访问前缀就失效了】

```yaml
spring:
#  mvc:
#    static-path-pattern: /res/**    这个会导致welcome page功能失效
  resources:
    static-locations: [classpath:/haha/]
```

- controller能处理`/index`。

二、自定义Favicon

指网页标签上的小图标。

favicon.ico 放在静态资源目录下即可。【名字必须是favicon.ico】

```yaml
spring:
#  mvc:
#    static-path-pattern: /res/**   这个会导致 Favicon 功能失效
```

> ctrl + shift + r / ctrl + f5强制不使用缓存刷新chrome页面

25、web场景-【源码分析】-静态资源原理

- SpringBoot启动默认加载 xxxAutoConfiguration 类（自动配置类）
- SpringMVC功能的自动配置类`WebMvcAutoConfiguration`，生效

```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class,
		ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {
    ...
}
```

- 给容器中配置的内容：
    - 配置文件的相关属性的绑定：`WebMvcProperties == spring.mvc`、`ResourceProperties == spring.resources`

```java
@Configuration(proxyBeanMethods = false)
@Import(EnableWebMvcConfiguration.class)
@EnableConfigurationProperties({ WebMvcProperties.class, ResourceProperties.class })
@Order(0)
public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer {
    ...
}
```

配置类只有一个有参构造器

```java
////有参构造器所有参数的值都会从容器中确定
public WebMvcAutoConfigurationAdapter(WebProperties webProperties, WebMvcProperties mvcProperties,
		ListableBeanFactory beanFactory, ObjectProvider<HttpMessageConverters> messageConvertersProvider,
		ObjectProvider<ResourceHandlerRegistrationCustomizer> resourceHandlerRegistrationCustomizerProvider,
		ObjectProvider<DispatcherServletPath> dispatcherServletPath,
		ObjectProvider<ServletRegistrationBean<?>> servletRegistrations) {
    this.mvcProperties = mvcProperties;
    this.beanFactory = beanFactory;
    this.messageConvertersProvider = messageConvertersProvider;
    this.resourceHandlerRegistrationCustomizer = resourceHandlerRegistrationCustomizerProvider.getIfAvailable();
    this.dispatcherServletPath = dispatcherServletPath;
    this.servletRegistrations = servletRegistrations;
    this.mvcProperties.checkConfiguration();
}
```

- WebProperties webProperties：获取和spring.resources绑定的所有的值的对象
- WebMvcProperties mvcProperties 获取和spring.mvc绑定的所有的值的对象
- ListableBeanFactory beanFactory Spring的beanFactory
- `ObjectProvider<HttpMessageConverters> messageConvertersProvider` 找到所有的HttpMessageConverters
- ResourceHandlerRegistrationCustomizer 找到 资源处理器的自定义器。
- DispatcherServletPath
- ServletRegistrationBean 给应用注册Servlet、Filter....

资源处理的默认规则

```java
...
public class WebMvcAutoConfiguration {
    ...
    public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration implements ResourceLoaderAware {
        ...
        @Override
        protected void addResourceHandlers(ResourceHandlerRegistry registry) {
            super.addResourceHandlers(registry);
            if (!this.resourceProperties.isAddMappings()) { // 通过配置来实现对静态资源的访问，false全部禁用
                logger.debug("Default resource handling disabled");
                return;
            }
            ServletContext servletContext = getServletContext();
            addResourceHandler(registry, "/webjars/**", "classpath:/META-INF/resources/webjars/");
            addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(), (registration) -> {
                registration.addResourceLocations(this.resourceProperties.getStaticLocations());
                if (servletContext != null) {
                    registration.addResourceLocations(new ServletContextResource(servletContext, SERVLET_LOCATION));
                }
            });
        }
        ...

    }
    ...
}
```

根据上述代码，我们可以同过配置禁止所有静态资源规则。

```yaml
spring:
  resources:
    add-mappings: false   #禁用所有静态资源规则
```

静态资源规则：

```java
@ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields = false)
public class ResourceProperties {

    // 对应的四个静态资源目录
    private static final String[] CLASSPATH_RESOURCE_LOCATIONS = { "classpath:/META-INF/resources/",
            "classpath:/resources/", "classpath:/static/", "classpath:/public/" };

    /**
     * Locations of static resources. Defaults to classpath:[/META-INF/resources/,
     * /resources/, /static/, /public/].
     */
    private String[] staticLocations = CLASSPATH_RESOURCE_LOCATIONS;
    ...
}

```

欢迎页的处理规则

```java
...
public class WebMvcAutoConfiguration {
    ...
    public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration implements ResourceLoaderAware {
        ...
        @Bean
        public WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext applicationContext,
                FormattingConversionService mvcConversionService, ResourceUrlProvider mvcResourceUrlProvider) {
            WelcomePageHandlerMapping welcomePageHandlerMapping = new WelcomePageHandlerMapping(
                    new TemplateAvailabilityProviders(applicationContext), applicationContext, getWelcomePage(),
                    this.mvcProperties.getStaticPathPattern());
            welcomePageHandlerMapping.setInterceptors(getInterceptors(mvcConversionService, mvcResourceUrlProvider));
            welcomePageHandlerMapping.setCorsConfigurations(getCorsConfigurations());
            return welcomePageHandlerMapping;
        }

```

`WelcomePageHandlerMapping`的构造方法如下：

```java
public WelcomePageHandlerMapping welcomePageHandlerMapping(TemplateAvailabilityProviders templateAvailabilityProviders,
                          ApplicationContext applicationContext, Resource welcomePage, String staticPathPattern) {
    if (welcomePage != null && "/**".equals(staticPathPattern)) {
        //要用欢迎页功能，必须是/**
        logger.info("Adding welcome page: " + welcomePage);
        setRootViewName("forward:index.html");
    }
    else if (welcomeTemplateExists(templateAvailabilityProviders, applicationContext)) {
        //调用Controller /index
        logger.info("Adding welcome page template: index");
        setRootViewName("index");
    }
}
```

这构造方法内的代码也解释了web场景-welcome与favicon功能中配置`static-path-pattern`了，welcome页面和小图标失效的问题。

26、请求处理-【源码分析】-Rest映射及源码解析

请求映射

- @xxxMapping;
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
- Rest风格支持（使用**HTTP**请求方式动词来表示对资源的操作）
  - 以前：
    - /getUser 获取用户
    - /deleteUser 删除用户
    - /editUser 修改用户
    - /saveUser保存用户
  - 现在： /user
    - GET-获取用户
    - DELETE-删除用户
    - PUT-修改用户
    - POST-保存用户
  - 核心Filter；HiddenHttpMethodFilter
- **用法**
  - 开启页面表单的Rest功能
  - 页面 form的属性method=post，隐藏域 _method=put、delete等（如果直接get或post，无需隐藏域）
  - 编写请求映射

```yaml
spring:
  mvc:
    hiddenmethod:
      filter:
        enabled: true   #开启页面表单的Rest功能
```

```html
<form action="/user" method="get">
    <input value="REST-GET提交" type="submit" />
</form>

<form action="/user" method="post">
    <input value="REST-POST提交" type="submit" />
</form>

<form action="/user" method="post">
    <input name="_method" type="hidden" value="DELETE"/>
    <input value="REST-DELETE 提交" type="submit"/>
</form>

<form action="/user" method="post">
    <input name="_method" type="hidden" value="PUT" />
    <input value="REST-PUT提交"type="submit" />
<form>
```

```
@GetMapping("/user")
//@RequestMapping(value = "/user",method = RequestMethod.GET)
public String getUser(){
    return "GET-张三";
}

@PostMapping("/user")
//@RequestMapping(value = "/user",method = RequestMethod.POST)
public String saveUser(){
    return "POST-张三";
}

@PutMapping("/user")
//@RequestMapping(value = "/user",method = RequestMethod.PUT)
public String putUser(){
    return "PUT-张三";
}

@DeleteMapping("/user")
//@RequestMapping(value = "/user",method = RequestMethod.DELETE)
public String deleteUser(){
    return "DELETE-张三";
}
```

- Rest原理（表单提交要使用REST的时候）
    - 表单提交会带上`\_method=PUT`
    - **请求过来被**`HiddenHttpMethodFilter`拦截
        - 请求是否正常，并且是POST
            - 获取到`_method`的值。
            - 兼容以下请求；**PUT.DELETE.PATCH**
            - **原生request（post），包装模式requesWrapper重写了getMethod方法，返回的是传入的值。**
            - **过滤器链放行的时候用wrapper。以后的方法调用getMethod是调用requesWrapper的。**

```
public class HiddenHttpMethodFilter extends OncePerRequestFilter {

    private static final List<String> ALLOWED_METHODS =
            Collections.unmodifiableList(Arrays.asList(HttpMethod.PUT.name(),
                    HttpMethod.DELETE.name(), HttpMethod.PATCH.name()));

    /** Default method parameter: {@code _method}. */
    public static final String DEFAULT_METHOD_PARAM = "_method";

    private String methodParam = DEFAULT_METHOD_PARAM;


    /**
     * Set the parameter name to look for HTTP methods.
     * @see #DEFAULT_METHOD_PARAM
     */
    public void setMethodParam(String methodParam) {
        Assert.hasText(methodParam, "'methodParam' must not be empty");
        this.methodParam = methodParam;
    }
```

```java

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        HttpServletRequest requestToUse = request;

        if ("POST".equals(request.getMethod()) && request.getAttribute(WebUtils.ERROR_EXCEPTION_ATTRIBUTE) == null) {
            String paramValue = request.getParameter(this.methodParam);
            if (StringUtils.hasLength(paramValue)) {
                String method = paramValue.toUpperCase(Locale.ENGLISH);
                if (ALLOWED_METHODS.contains(method)) {
                    requestToUse = new HttpMethodRequestWrapper(request, method);
                }
            }
        }

        filterChain.doFilter(requestToUse, response);
    }


    /**
     * Simple {@link HttpServletRequest} wrapper that returns the supplied method for
     * {@link HttpServletRequest#getMethod()}.
     */
    private static class HttpMethodRequestWrapper extends HttpServletRequestWrapper {

        private final String method;

        public HttpMethodRequestWrapper(HttpServletRequest request, String method) {
            super(request);
            this.method = method;
        }

        @Override
        public String getMethod() {
            return this.method;
        }
    }

}
```

- Rest使用客户端工具。
  - 如PostMan可直接发送put、delete等方式请求。

27、请求处理-【源码分析】-怎么改变默认的_method

```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
```

```java
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class,
        ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {

    ...

    @Bean
    @ConditionalOnMissingBean(HiddenHttpMethodFilter.class)
    @ConditionalOnProperty(prefix = "spring.mvc.hiddenmethod.filter", name = "enabled", matchIfMissing = false)
    public OrderedHiddenHttpMethodFilter hiddenHttpMethodFilter() {
        return new OrderedHiddenHttpMethodFilter();
    }

    ...
}

```

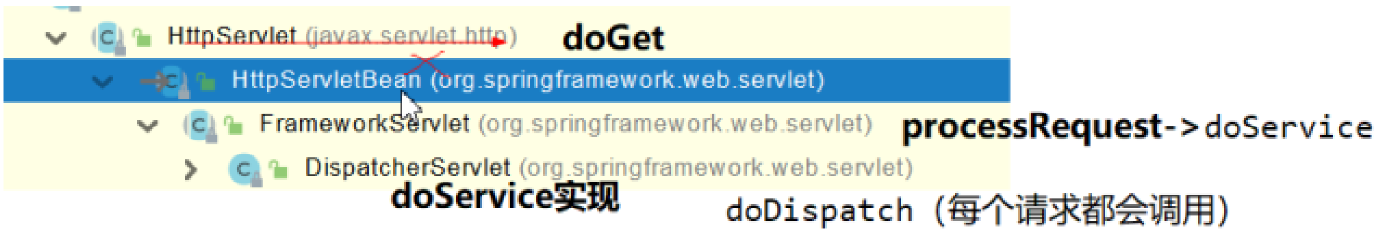
`@ConditionalOnMissingBean(HiddenHttpMethodFilter.class)`意味着在没有`HiddenHttpMethodFilter`时，才执行`hiddenHttpMethodFilter()`。因此，我们可以自定义filter，改变默认的`_method`。例如：

```java
@Configuration(proxyBeanMethods = false)
public class WebConfig{
    //自定义filter
    @Bean
    public HiddenHttpMethodFilter hiddenHttpMethodFilter(){
        HiddenHttpMethodFilter methodFilter = new HiddenHttpMethodFilter();
        methodFilter.setMethodParam("_m");
        return methodFilter;
    }
}
```

将`\_method`改成`_m`。

```html
<form action="/user" method="post">
    <input name="_m" type="hidden" value="DELETE"/>
    <input value="REST-DELETE 提交" type="submit"/>
</form>
```

28、请求处理-【源码分析】-请求映射原理


HttpServlet (javax.servlet.http)
doGet
HttpServletBean (org.springframework.web.servlet)
FrameworkServlet (org.springframework.web.servlet)
processRequest->doService
DispatcherServlet (org.springframework.web.servlet)
doService实现
doDispatch（每个请求都会调用）

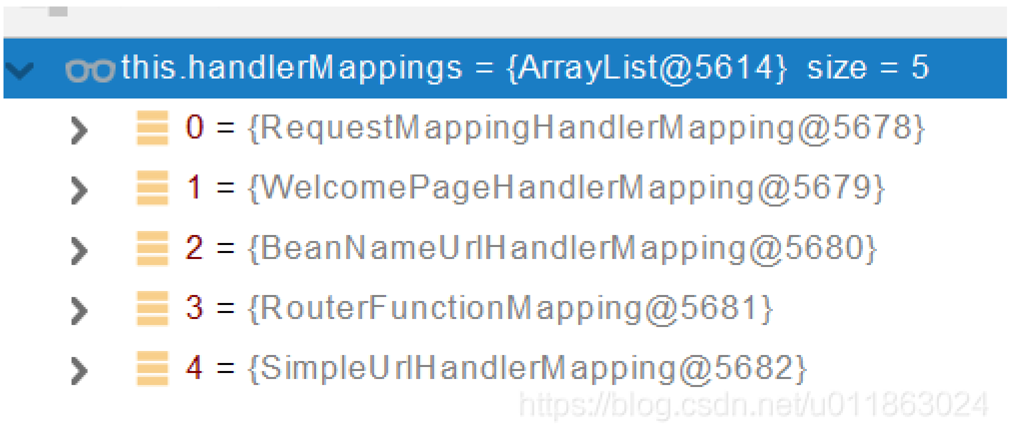
SpringMVC功能分析都从 `org.springframework.web.servlet.DispatcherServlet` -> `doDispatch()`

```java
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

        try {
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            // 找到当前请求使用哪个Handler（Controller的方法）处理
            mappedHandler = getHandler(processedRequest);

            //HandlerMapping：处理器映射。/xxx->>xxxx
    ...
}
```

`getHandler()`方法如下：

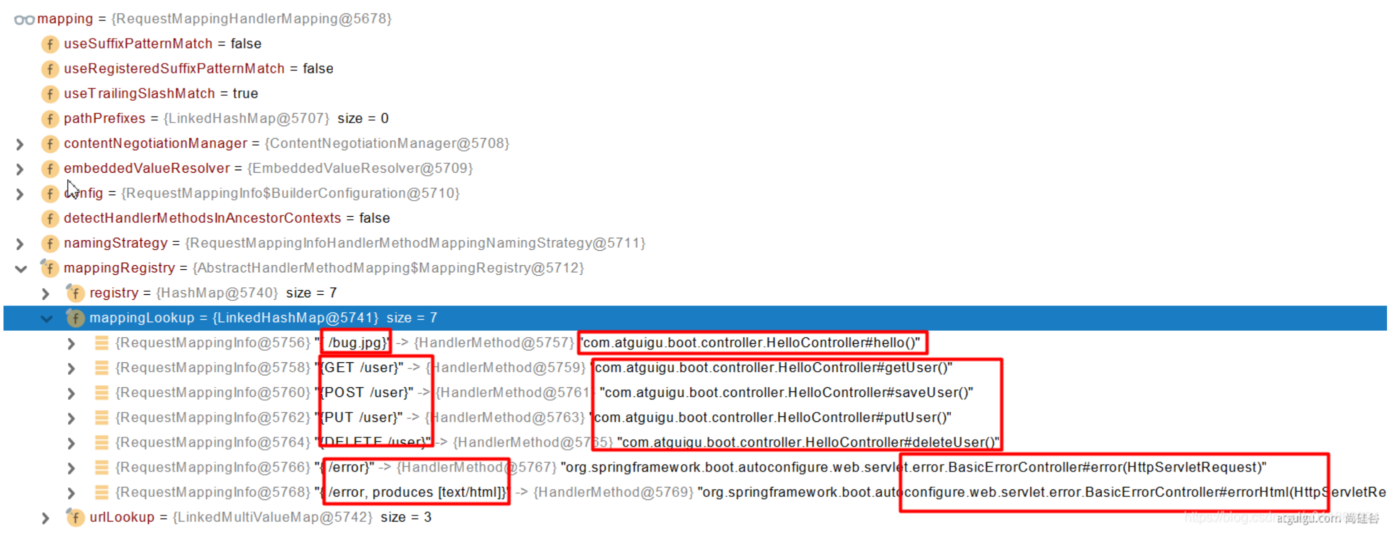
```java
@Nullable
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    if (this.handlerMappings != null) {
        for (HandlerMapping mapping : this.handlerMappings) {
            HandlerExecutionChain handler = mapping.getHandler(request);
            if (handler != null) {
                return handler;
            }
        }
    }
    return null;
}
```

`this.handlerMappings`在Debug模式下展现的内容：



```
this.handlerMappings = {ArrayList@5614}  size = 5
    0 = {RequestMappingHandlerMapping@5678}
    1 = {WelcomePageHandlerMapping@5679}
    2 = {BeanNameUrlHandlerMapping@5680}
    3 = {RouterFunctionMapping@5681}
    4 = {SimpleUrlHandlerMapping@5682}
```

其中，保存了所有`@RequestMapping` 和`handler`的映射规则。



```
mapping = {RequestMappingHandlerMapping@5678}
    useSuffixPatternMatch = false
    useRegisteredSuffixPatternMatch = false
    useTrailingSlashMatch = true
    pathPrefixes = {LinkedHashMap@5707}  size = 0
  > contentNegotiationManager = {ContentNegotiationManager@5708}
  > embeddedValueResolver = {EmbeddedValueResolver@5709}
  > config = {RequestMappingInfo$BuilderConfiguration@5710}
    detectHandlerMethodsInAncestorContexts = false
  > namingStrategy = {RequestMappingInfoHandlerMethodMappingNamingStrategy@5711}
  v mappingRegistry = {AbstractHandlerMethodMapping$MappingRegistry@5712}
    > registry = {HashMap@5740}  size = 7
    v mappingLookup = {LinkedHashMap@5741}  size = 7
      > {RequestMappingInfo@5756} "{ /bug.jpg}" -> {HandlerMethod@5757} "com.atguigu.boot.controller.HelloController#hello()"
      > {RequestMappingInfo@5758} "{GET /user}" -> {HandlerMethod@5759} "com.atguigu.boot.controller.HelloController#getUser()"
      > {RequestMappingInfo@5760} "{POST /user}" -> {HandlerMethod@5761} "com.atguigu.boot.controller.HelloController#saveUser()"
      > {RequestMappingInfo@5762} "{PUT /user}" -> {HandlerMethod@5763} "com.atguigu.boot.controller.HelloController#putUser()"
      > {RequestMappingInfo@5764} "{DELETE /user}" -> {HandlerMethod@5765} "com.atguigu.boot.controller.HelloController#deleteUser()"
      > {RequestMappingInfo@5766} "{ /error}" -> {HandlerMethod@5767} "org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController#error(HttpServletRequest)"
      > {RequestMappingInfo@5768} "{ /error, produces [text/html]}" -> {HandlerMethod@5769} "org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController#errorHtml(HttpServletRe
    > urlLookup = {LinkedMultiValueMap@5742}  size = 3
```

所有的请求映射都在HandlerMapping中：

- SpringBoot自动配置欢迎页的 `WelcomePageHandlerMapping` 。访问 /能访问到index.html；
- SpringBoot自动配置了默认的 `RequestMappingHandlerMapping`
- 请求进来，挨个尝试所有的 `HandlerMapping` 看是否有请求信息。
  - 如果有就找到这个请求对应的handler
  - 如果没有就是下一个 `HandlerMapping`
- 我们需要一些自定义的映射处理，我们也可以自己给容器中放**HandlerMapping**。未来可以自定义**HandlerMapping**

IDEA快捷键：

- Ctrl + Alt + U : 以UML的类图展现类有哪些继承类，派生类以及实现哪些接口。
- Crtl + Alt + Shift + U : 同上，区别在于上条快捷键结果在新页展现，而本条快捷键结果在弹窗展现。
- Ctrl + H : 以树形方式展现类层次结构图。

29、请求处理-常用参数注解使用

本部分旨在说明，写 `Controller` 方法的时候，参数栏可以使用哪些参数。

- `Servlet API`
  - WebRequest
  - ServletRequest
  - MultipartRequest
  - HttpSession
  - javax.servlet.http.PushBuilder
  - Principal
  - InputStream
  - Reader
  - HttpMethod
  - Locale
  - TimeZone
  - ZoneId
- 复杂参数
  - Map

    - Errors、BindingResult
    - Model
    - RedirectAttributes
    - ServletResponse
    - SessionStatus
    - UriComponentsBuilder
    - ServletUriComponentsBuilder
- 注解
    - `@PathVariable` 路径变量

```java
@RequestMapping("/car/{id}")
public Map<String, Object> getCar(@PathVariable("id") Integer id){
    Map<String, Object> map = new HashMap<>();
    map.put("id", id);
    return map;
}
```

    - `@RequestHeader` 获取请求头
    - `@RequestParam` 获取请求参数（指问号后的参数，url?a=1&b=2）

```java
@RequestMapping("/bug.jpg")
public String hello(@RequestParam("username") String name){ // 从request中问号的参数后面将参数拿出来
    return name;
}
```

    - `@CookieValue` 获取Cookie值
    - `@RequestAttribute` 获取request域属性
    - `@RequestBody` 获取请求体[POST]
    - `@MatrixVariable` 矩阵变量
    - `@ModelAttribute`

```java
@RestController
public class ParameterTestController {
    //  car/2/owner/zhangsan
    @GetMapping("/car/{id}/owner/{username}")
    public Map<String,Object> getCar(@PathVariable("id") Integer id,
                                     @PathVariable("username") String name,
                                     @PathVariable Map<String,String> pv,
                                     @RequestHeader("User-Agent") String userAgent,
                                     @RequestHeader Map<String,String> header,
                                     @RequestParam("age") Integer age,
                                     @RequestParam("inters") List<String> inters,
                                     @RequestParam Map<String,String> params,
                                     @CookieValue("_ga") String _ga,
                                     @CookieValue("_ga") Cookie cookie){

        Map<String,Object> map = new HashMap<>();
//        map.put("id",id);
//        map.put("name",name);
//        map.put("pv",pv);
//        map.put("userAgent",userAgent);
//        map.put("headers",header);
```

```java
        map.put("age",age);
        map.put("inters",inters);
        map.put("params",params);
        map.put("_ga",_ga);
        System.out.println(cookie.getName()+"===>"+cookie.getValue());
        return map;
    }

    @PostMapping("/save")
    public Map postMethod(@RequestBody String content){
        Map<String,Object> map = new HashMap<>();
        map.put("content",content);
        return map;
    }
}
```

- 自定义对象参数：可以自动类型转换域格式化，可以级联封装

30、请求处理-@RequestAttribute

请求属性，是用来获取request域中保存的属性的。

用例：

```java
@Controller
public class RequestController {

    @GetMapping("/goto")
    public String goToPage(HttpServletRequest request){

        request.setAttribute("msg","成功了...");
        request.setAttribute("code",200);
        return "forward:/success";  //转发到  /success请求
    }

    @GetMapping("/params")
    public String testParam(Map<String,Object> map,
                            Model model,
                            HttpServletRequest request,
                            HttpServletResponse response){
        map.put("hello","world666");
        model.addAttribute("world","hello666");
        request.setAttribute("message","HelloWorld");

        Cookie cookie = new Cookie("c1","v1");
        response.addCookie(cookie);
        return "forward:/success";
    }

    ///<-----------------主角@RequestAttribute在这个方法
    @ResponseBody
    @GetMapping("/success")
    public Map success(@RequestAttribute(value = "msg",required = false) String msg,
                       @RequestAttribute(value = "code",required = false)Integer code,
                       HttpServletRequest request){
        Object msg1 = request.getAttribute("msg");

        Map<String,Object> map = new HashMap<>();
        Object hello = request.getAttribute("hello");
        Object world = request.getAttribute("world");
        Object message = request.getAttribute("message");

        map.put("reqMethod_msg",msg1);
        map.put("annotation_msg",msg);
        map.put("hello",hello);
        map.put("world",world);
        map.put("message",message);

        return map;
    }
}
```

31、请求处理-@MatrixVariable与UrlPathHelper

1. 语法： 请求路径：`/cars/sell;low=34;brand=byd,audi,yd`
2. SpringBoot默认是禁用了矩阵变量的功能
    - 手动开启：原理。对于路径的处理。UrlPathHelper的removeSemicolonContent设置为false，让其支持矩阵变量的。
3. 矩阵变量**必须**有url路径变量才能被解析

手动开启矩阵变量：

- 实现`WebMvcConfigurer`接口：

```java
@Configuration(proxyBeanMethods = false)
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {

        UrlPathHelper urlPathHelper = new UrlPathHelper();
        // 不移除；后面的内容。矩阵变量功能就可以生效
        urlPathHelper.setRemoveSemicolonContent(false);
        configurer.setUrlPathHelper(urlPathHelper);
    }
}
```

- 创建返回`WebMvcConfigurer`Bean：

```java
@Configuration(proxyBeanMethods = false)
public class WebConfig{
    @Bean
    public WebMvcConfigurer webMvcConfigurer(){
        return new WebMvcConfigurer() {
                        @Override
            public void configurePathMatch(PathMatchConfigurer configurer) {
                UrlPathHelper urlPathHelper = new UrlPathHelper();
                // 不移除；后面的内容。矩阵变量功能就可以生效
                urlPathHelper.setRemoveSemicolonContent(false);
                configurer.setUrlPathHelper(urlPathHelper);
            }
        }
    }
}
```

`@MatrixVariable`**的用例**

```java
@RestController
public class ParameterTestController {

    ///cars/sell;low=34;brand=byd,audi,yd
    @GetMapping("/cars/{path}")
    public Map carsSell(@MatrixVariable("low") Integer low, // 矩阵变量
                        @MatrixVariable("brand") List<String> brand, // 矩阵变量
                        @PathVariable("path") String path){
        Map<String,Object> map = new HashMap<>();

        map.put("low",low);
        map.put("brand",brand);
        map.put("path",path);
        return map;
    }

    // /boss/1;age=20/2;age=10

    @GetMapping("/boss/{bossId}/{empId}")
    public Map boss(@MatrixVariable(value = "age",pathVar = "bossId") Integer bossAge,
                    @MatrixVariable(value = "age",pathVar = "empId") Integer empAge){
        Map<String,Object> map = new HashMap<>();

        map.put("bossAge",bossAge);
        map.put("empAge",empAge);
        return map;

    }

}
```

32、请求处理-【源码分析】-各种类型参数解析原理

这要从`DispatcherServlet`开始说起：

```java
public class DispatcherServlet extends FrameworkServlet {

    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        boolean multipartRequestParsed = false;

        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

        try {
            ModelAndView mv = null;
            Exception dispatchException = null;

            try {
                processedRequest = checkMultipart(request);
                multipartRequestParsed = (processedRequest != request);

                // Determine handler for the current request.
                mappedHandler = getHandler(processedRequest);
                if (mappedHandler == null) {
                    noHandlerFound(processedRequest, response);
                    return;
                }

                // Determine handler adapter for the current request.
                HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
                ...
```

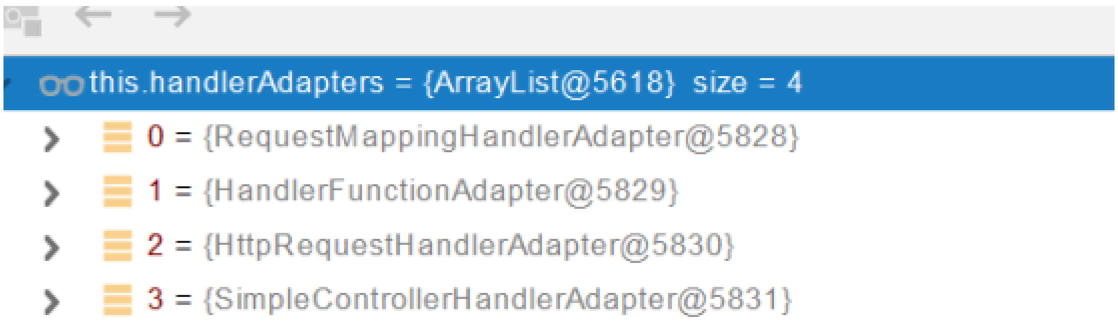
- `HandlerMapping`中找到能处理请求的`Handler`（Controller的某个方法）。
- 为当前Handler 找一个适配器 `HandlerAdapter`，用的最多的是**RequestMappingHandlerAdapter**。
- 适配器执行目标方法并确定方法参数的每一个值。

一、HandlerAdapter

默认会加载所有`HandlerAdapter`

```java
public class DispatcherServlet extends FrameworkServlet {

    /** Detect all HandlerAdapters or just expect "handlerAdapter" bean?. */
    private boolean detectAllHandlerAdapters = true;

    ...

    private void initHandlerAdapters(ApplicationContext context) {
        this.handlerAdapters = null;

        if (this.detectAllHandlerAdapters) {
            // Find all HandlerAdapters in the ApplicationContext, including ancestor contexts.
            Map<String, HandlerAdapter> matchingBeans =
                BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerAdapter.class, true, false);
            if (!matchingBeans.isEmpty()) {
                this.handlerAdapters = new ArrayList<>(matchingBeans.values());
                // We keep HandlerAdapters in sorted order.
                AnnotationAwareOrderComparator.sort(this.handlerAdapters);
            }
        }
      ...
```

有这些`HandlerAdapter`:


```
this.handlerAdapters = {ArrayList@5618} size = 4
  0 = {RequestMappingHandlerAdapter@5828}
  1 = {HandlerFunctionAdapter@5829}
  2 = {HttpRequestHandlerAdapter@5830}
  3 = {SimpleControllerHandlerAdapter@5831}
```

0. 支持方法上标注`@RequestMapping`
1. 支持函数式编程的
2. ...
3. ...

二、执行目标方法

```java
public class DispatcherServlet extends FrameworkServlet {

    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        ModelAndView mv = null;

        ...

        // Determine handler for the current request.
        mappedHandler = getHandler(processedRequest);
        if (mappedHandler == null) {
            noHandlerFound(processedRequest, response);
            return;
        }

        // Determine handler adapter for the current request.
        HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

        ...
        //本节重点
        // Actually invoke the handler.
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
```

`HandlerAdapter`接口实现类`RequestMappingHandlerAdapter`（主要用来处理`@RequestMapping`）

```java
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {

    ...

    //AbstractHandlerMethodAdapter类的方法，RequestMappingHandlerAdapter继承AbstractHandlerMethodAdapter
    public final ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
        throws Exception {

        return handleInternal(request, response, (HandlerMethod) handler);
    }

    @Override
    protected ModelAndView handleInternal(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
        ModelAndView mav;
        //handleInternal的核心
        mav = invokeHandlerMethod(request, response, handlerMethod);//解释看下节
        //...
        return mav;
    }
}
```

三、参数解析器

确定将要执行的目标方法的每一个参数的值是什么;【简言之解析Controller方法参数的工具】

SpringMVC目标方法能写多少种参数类型。取决于**参数解析器argumentResolvers**。

```java
@Nullable
protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
                                            HttpServletResponse response, HandlerMethod
handlerMethod) throws Exception {

    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    try {
        WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);
        ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);

        ServletInvocableHandlerMethod invocableMethod =
createInvocableHandlerMethod(handlerMethod);
        if (this.argumentResolvers != null) {//<-----关注点
            invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
        }

        ...
```

`this.argumentResolvers` 在 `afterPropertiesSet()` 方法内初始化

```java
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {

    @Nullable
    private HandlerMethodArgumentResolverComposite argumentResolvers;

    @Override
    public void afterPropertiesSet() {
        ...
        if (this.argumentResolvers == null) {//初始化argumentResolvers
            List<HandlerMethodArgumentResolver> resolvers = getDefaultArgumentResolvers();
            this.argumentResolvers = new
HandlerMethodArgumentResolverComposite().addResolvers(resolvers);
        }
        ...
    }

    //初始化了一堆的实现HandlerMethodArgumentResolver接口的
    private List<HandlerMethodArgumentResolver> getDefaultArgumentResolvers() {
        List<HandlerMethodArgumentResolver> resolvers = new ArrayList<>(30);

        // Annotation-based argument resolution
        resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), false));
        resolvers.add(new RequestParamMapMethodArgumentResolver());
        resolvers.add(new PathVariableMethodArgumentResolver());
        resolvers.add(new PathVariableMapMethodArgumentResolver());
        resolvers.add(new MatrixVariableMethodArgumentResolver());
        resolvers.add(new MatrixVariableMapMethodArgumentResolver());
        resolvers.add(new ServletModelAttributeMethodProcessor(false));
        resolvers.add(new RequestResponseBodyMethodProcessor(getMessageConverters(),
this.requestResponseBodyAdvice));
        resolvers.add(new RequestPartMethodArgumentResolver(getMessageConverters(),
this.requestResponseBodyAdvice));
        resolvers.add(new RequestHeaderMethodArgumentResolver(getBeanFactory()));
        resolvers.add(new RequestHeaderMapMethodArgumentResolver());
        resolvers.add(new ServletCookieValueMethodArgumentResolver(getBeanFactory()));
        resolvers.add(new ExpressionValueMethodArgumentResolver(getBeanFactory()));
        resolvers.add(new SessionAttributeMethodArgumentResolver());
        resolvers.add(new RequestAttributeMethodArgumentResolver());

        // Type-based argument resolution
        resolvers.add(new ServletRequestMethodArgumentResolver());
        resolvers.add(new ServletResponseMethodArgumentResolver());
        resolvers.add(new HttpEntityMethodProcessor(getMessageConverters(),
this.requestResponseBodyAdvice));
        resolvers.add(new RedirectAttributesMethodArgumentResolver());
        resolvers.add(new ModelMethodProcessor());
        resolvers.add(new MapMethodProcessor());
        resolvers.add(new ErrorsMethodArgumentResolver());
        resolvers.add(new SessionStatusMethodArgumentResolver());
        resolvers.add(new UriComponentsBuilderMethodArgumentResolver());
        if (KotlinDetector.isKotlinPresent()) {
            resolvers.add(new ContinuationHandlerMethodArgumentResolver());
        }

        // Custom arguments
        if (getCustomArgumentResolvers() != null) {
            resolvers.addAll(getCustomArgumentResolvers());
        }

        // Catch-all
        resolvers.add(new PrincipalMethodArgumentResolver());
        resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), true));
        resolvers.add(new ServletModelAttributeMethodProcessor(true));

        return resolvers;
    }

}
```

`HandlerMethodArgumentResolverComposite`类如下：（众多**参数解析器argumentResolvers**的包装类）。

```java
public class HandlerMethodArgumentResolverComposite implements
HandlerMethodArgumentResolver {

    private final List<HandlerMethodArgumentResolver> argumentResolvers = new ArrayList<>
();

    ...

    public HandlerMethodArgumentResolverComposite addResolvers(
            @Nullable HandlerMethodArgumentResolver... resolvers) {

        if (resolvers != null) {
            Collections.addAll(this.argumentResolvers, resolvers);
        }
        return this;
    }

    ...
}
```

我们看看`HandlerMethodArgumentResolver`的源码:

```
public interface HandlerMethodArgumentResolver {

    //当前解析器是否支持解析这种参数
    boolean supportsParameter(MethodParameter parameter);

    @Nullable//如果支持，就调用 resolveArgument
    Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
            NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception;

}

```

四、返回值处理器

ValueHandler

```
@Nullable
protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
                                           HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    try {
        WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);
        ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);

        ServletInvocableHandlerMethod invocableMethod = createInvocableHandlerMethod(handlerMethod);
        if (this.argumentResolvers != null) {
            invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
        }
        if (this.returnValueHandlers != null) {//<---关注点
            invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
        }
     ...

```

`this.returnValueHandlers`在`afterPropertiesSet()`方法内初始化

```
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {

    @Nullable
```

```java
    private HandlerMethodReturnValueHandlerComposite returnValueHandlers;

    @Override
    public void afterPropertiesSet() {

        ...

        if (this.returnValueHandlers == null) {
            List<HandlerMethodReturnValueHandler> handlers = getDefaultReturnValueHandlers();
            this.returnValueHandlers = new HandlerMethodReturnValueHandlerComposite().addHandlers(handlers);
        }
    }

    //初始化了一堆的实现HandlerMethodReturnValueHandler接口的
    private List<HandlerMethodReturnValueHandler> getDefaultReturnValueHandlers() {
        List<HandlerMethodReturnValueHandler> handlers = new ArrayList<>(20);

        // Single-purpose return value types
        handlers.add(new ModelAndViewMethodReturnValueHandler());
        handlers.add(new ModelMethodProcessor());
        handlers.add(new ViewMethodReturnValueHandler());
        handlers.add(new ResponseBodyEmitterReturnValueHandler(getMessageConverters(),
                this.reactiveAdapterRegistry, this.taskExecutor, this.contentNegotiationManager));
        handlers.add(new StreamingResponseBodyReturnValueHandler());
        handlers.add(new HttpEntityMethodProcessor(getMessageConverters(),
                this.contentNegotiationManager, this.requestResponseBodyAdvice));
        handlers.add(new HttpHeadersReturnValueHandler());
        handlers.add(new CallableMethodReturnValueHandler());
        handlers.add(new DeferredResultMethodReturnValueHandler());
        handlers.add(new AsyncTaskMethodReturnValueHandler(this.beanFactory));

        // Annotation-based return value types
        handlers.add(new ServletModelAttributeMethodProcessor(false));
        handlers.add(new RequestResponseBodyMethodProcessor(getMessageConverters(),
                this.contentNegotiationManager, this.requestResponseBodyAdvice));

        // Multi-purpose return value types
        handlers.add(new ViewNameMethodReturnValueHandler());
        handlers.add(new MapMethodProcessor());

        // Custom return value types
        if (getCustomReturnValueHandlers() != null) {
            handlers.addAll(getCustomReturnValueHandlers());
        }

        // Catch-all
        if (!CollectionUtils.isEmpty(getModelAndViewResolvers())) {
            handlers.add(new ModelAndViewResolverMethodReturnValueHandler(getModelAndViewResolvers()));
        }
        else {
            handlers.add(new ServletModelAttributeMethodProcessor(true));
        }

        return handlers;
    }
}
```

`HandlerMethodReturnValueHandlerComposite`类如下：

```java
public class HandlerMethodReturnValueHandlerComposite implements 
HandlerMethodReturnValueHandler {

    private final List<HandlerMethodReturnValueHandler> returnValueHandlers = new 
ArrayList<>();

    ...

    public HandlerMethodReturnValueHandlerComposite addHandlers(
            @Nullable List<? extends HandlerMethodReturnValueHandler> handlers) {

        if (handlers != null) {
            this.returnValueHandlers.addAll(handlers);
        }
        return this;
    }

}
```

`HandlerMethodReturnValueHandler`接口：

```java
public interface HandlerMethodReturnValueHandler {

    boolean supportsReturnType(MethodParameter returnType);

    void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
            ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws 
Exception;

}
```

五、回顾执行目标方法

```java
public class DispatcherServlet extends FrameworkServlet {
    ...
    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) 
throws Exception {
        ModelAndView mv = null;
        ...
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
```

`RequestMappingHandlerAdapter`的`handle()`方法：

```java
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {

    ...

    //AbstractHandlerMethodAdapter类的方法，RequestMappingHandlerAdapter继承
AbstractHandlerMethodAdapter
    public final ModelAndView handle(HttpServletRequest request, HttpServletResponse 
response, Object handler)
        throws Exception {

            return handleInternal(request, response, (HandlerMethod) handler);
    }

    @Override
    protected ModelAndView handleInternal(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
        ModelAndView mav;
        //handleInternal的核心
        mav = invokeHandlerMethod(request, response, handlerMethod);//解释看下节
        //...
        return mav;
    }
}
```

`RequestMappingHandlerAdapter`的`invokeHandlerMethod()`方法:

```java
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {

    protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

        ServletWebRequest webRequest = new ServletWebRequest(request, response);
        try {
            ...

            ServletInvocableHandlerMethod invocableMethod =
createInvocableHandlerMethod(handlerMethod);
            if (this.argumentResolvers != null) {
                invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
            }
            if (this.returnValueHandlers != null) {

invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
            }
            ...

            //关注点: 执行目标方法
            invocableMethod.invokeAndHandle(webRequest, mavContainer);
            if (asyncManager.isConcurrentHandlingStarted()) {
                return null;
            }

            return getModelAndView(mavContainer, modelFactory, webRequest);
        }
        finally {
            webRequest.requestCompleted();
        }
    }
```

`invokeAndHandle()`方法如下:

```java
public class ServletInvocableHandlerMethod extends InvocableHandlerMethod {

    public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer
mavContainer,
            Object... providedArgs) throws Exception {

```

```
        Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);

        ...

        try {
            //returnValue存储起来
            this.returnValueHandlers.handleReturnValue(
                    returnValue, getReturnValueType(returnValue), mavContainer,
webRequest);
        }
        catch (Exception ex) {
            ...
        }
    }

    @Nullable//InvocableHandlerMethod类的，ServletInvocableHandlerMethod类继承
InvocableHandlerMethod类
    public Object invokeForRequest(NativeWebRequest request, @Nullable
ModelAndViewContainer mavContainer,
            Object... providedArgs) throws Exception {

        ////获取方法的参数值
        Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);

        ...

        return doInvoke(args);
    }

    @Nullable
    protected Object doInvoke(Object... args) throws Exception {
        Method method = getBridgedMethod();//@RequestMapping的方法
        ReflectionUtils.makeAccessible(method);
        try {
            if (KotlinDetector.isSuspendingFunction(method)) {
                return CoroutinesUtils.invokeSuspendingFunction(method, getBean(), args);
            }
            //通过反射调用
            return method.invoke(getBean(), args);//getBean()指@RequestMapping的方法所在类的
对象。
        }
        catch (IllegalArgumentException ex) {
            ...
        }
        catch (InvocationTargetException ex) {
            ...
        }
    }

}
```

六、如何确定目标方法每一个参数的值

重点分析`ServletInvocableHandlerMethod`的`getMethodArgumentValues`方法

```
public class ServletInvocableHandlerMethod extends InvocableHandlerMethod {
    ...
```

```java

    @Nullable//InvocableHandlerMethod类的，ServletInvocableHandlerMethod类继承InvocableHandlerMethod类
    public Object invokeForRequest(NativeWebRequest request, @Nullable ModelAndViewContainer mavContainer,
            Object... providedArgs) throws Exception {

        ////获取方法的参数值
        Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);

        ...

        return doInvoke(args);
    }

    //本节重点，获取方法的参数值
    protected Object[] getMethodArgumentValues(NativeWebRequest request, @Nullable ModelAndViewContainer mavContainer,
            Object... providedArgs) throws Exception {

        MethodParameter[] parameters = getMethodParameters();
        if (ObjectUtils.isEmpty(parameters)) {
            return EMPTY_ARGS;
        }

        Object[] args = new Object[parameters.length];
        for (int i = 0; i < parameters.length; i++) {
            MethodParameter parameter = parameters[i];
            parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);
            args[i] = findProvidedArgument(parameter, providedArgs);
            if (args[i] != null) {
                continue;
            }
            //查看resolvers是否有支持
            if (!this.resolvers.supportsParameter(parameter)) {
                throw new IllegalStateException(formatArgumentError(parameter, "No suitable resolver"));
            }
            try {
                //支持的话就开始解析吧
                args[i] = this.resolvers.resolveArgument(parameter, mavContainer, request, this.dataBinderFactory);
            }
            catch (Exception ex) {
                ....
            }
        }
        return args;
    }

}
```

`this.resolvers`的类型为`HandlerMethodArgumentResolverComposite`（在参数解析器章节提及）

```java
public class HandlerMethodArgumentResolverComposite implements HandlerMethodArgumentResolver {

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        return getArgumentResolver(parameter) != null;
    }

    @Override
    @Nullable
    public Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
            NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {

        HandlerMethodArgumentResolver resolver = getArgumentResolver(parameter);
        if (resolver == null) {
            throw new IllegalArgumentException("Unsupported parameter type [" +
                    parameter.getParameterType().getName() + "]. supportsParameter should be called first.");
        }
        return resolver.resolveArgument(parameter, mavContainer, webRequest, binderFactory);
    }


    @Nullable
    private HandlerMethodArgumentResolver getArgumentResolver(MethodParameter parameter) {
        HandlerMethodArgumentResolver result = this.argumentResolverCache.get(parameter);
        if (result == null) {
            //挨个判断所有参数解析器那个支持解析这个参数
            for (HandlerMethodArgumentResolver resolver : this.argumentResolvers) {
                if (resolver.supportsParameter(parameter)) {
                    result = resolver;
                    this.argumentResolverCache.put(parameter, result);//找到了，resolver就缓存起来，方便稍后resolveArgument()方法使用
                    break;
                }
            }
        }
        return result;
    }
}
```

小结

本节描述，一个请求发送到DispatcherServlet后的具体处理流程，也就是SpringMVC的主要原理。

本节内容较多且硬核，对日后编程很有帮助，需耐心对待。

可以运行一个示例，打断点，在Debug模式下，查看程序流程。

33、请求处理-【源码分析】-Servlet API参数解析原理

- WebRequest
- ServletRequest
- MultipartRequest
- HttpSession
- javax.servlet.http.PushBuilder
- Principal
- InputStream

- Reader
- HttpMethod
- Locale
- TimeZone
- ZoneId

ServletRequestMethodArgumentResolver用来处理以上的参数

```java
public class ServletRequestMethodArgumentResolver implements HandlerMethodArgumentResolver {

    @Nullable
    private static Class<?> pushBuilder;

    static {
        try {
            pushBuilder = ClassUtils.forName("javax.servlet.http.PushBuilder",
                    ServletRequestMethodArgumentResolver.class.getClassLoader());
        }
        catch (ClassNotFoundException ex) {
            // Servlet 4.0 PushBuilder not found - not supported for injection
            pushBuilder = null;
        }
    }


    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        Class<?> paramType = parameter.getParameterType();
        return (WebRequest.class.isAssignableFrom(paramType) ||
                ServletRequest.class.isAssignableFrom(paramType) ||
                MultipartRequest.class.isAssignableFrom(paramType) ||
                HttpSession.class.isAssignableFrom(paramType) ||
                (pushBuilder != null && pushBuilder.isAssignableFrom(paramType)) ||
                (Principal.class.isAssignableFrom(paramType) && !parameter.hasParameterAnnotations()) ||
                InputStream.class.isAssignableFrom(paramType) ||
                Reader.class.isAssignableFrom(paramType) ||
                HttpMethod.class == paramType ||
                Locale.class == paramType ||
                TimeZone.class == paramType ||
                ZoneId.class == paramType);
    }

    @Override
    public Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
            NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {

        Class<?> paramType = parameter.getParameterType();

        // WebRequest / NativeWebRequest / ServletWebRequest
        if (WebRequest.class.isAssignableFrom(paramType)) {
            if (!paramType.isInstance(webRequest)) {
                throw new IllegalStateException(
                        "Current request is not of type [" + paramType.getName() + "]: " + webRequest);
            }
            return webRequest;
        }

        // ServletRequest / HttpServletRequest / MultipartRequest / MultipartHttpServletRequest
        if (ServletRequest.class.isAssignableFrom(paramType) || MultipartRequest.class.isAssignableFrom(paramType)) {
            return resolveNativeRequest(webRequest, paramType);
        }

        // HttpServletRequest required for all further argument types
        return resolveArgument(paramType, resolveNativeRequest(webRequest, HttpServletRequest.class));
    }

    private <T> T resolveNativeRequest(NativeWebRequest webRequest, Class<T> requiredType) {
        T nativeRequest = webRequest.getNativeRequest(requiredType);
        if (nativeRequest == null) {
            throw new IllegalStateException(
                    "Current request is not of type [" + requiredType.getName() + "]: " + webRequest);
        }
        return nativeRequest;
    }

    @Nullable
    private Object resolveArgument(Class<?> paramType, HttpServletRequest request) throws IOException {
        if (HttpSession.class.isAssignableFrom(paramType)) {
            HttpSession session = request.getSession();
            if (session != null && !paramType.isInstance(session)) {
                throw new IllegalStateException(
                        "Current session is not of type [" + paramType.getName() + "]: " + session);
            }
            return session;
        }
        else if (pushBuilder != null && pushBuilder.isAssignableFrom(paramType)) {
            return PushBuilderDelegate.resolvePushBuilder(request, paramType);
        }
        else if (InputStream.class.isAssignableFrom(paramType)) {
            InputStream inputStream = request.getInputStream();
            if (inputStream != null && !paramType.isInstance(inputStream)) {
                throw new IllegalStateException(
                        "Request input stream is not of type [" + paramType.getName() + "]: " + inputStream);
            }
            return inputStream;
        }
        else if (Reader.class.isAssignableFrom(paramType)) {
            Reader reader = request.getReader();
            if (reader != null && !paramType.isInstance(reader)) {
                throw new IllegalStateException(
                        "Request body reader is not of type [" + paramType.getName() + "]: " + reader);
            }
            return reader;
        }
        else if (Principal.class.isAssignableFrom(paramType)) {
            Principal userPrincipal = request.getUserPrincipal();
            if (userPrincipal != null && !paramType.isInstance(userPrincipal)) {
                throw new IllegalStateException(
                        "Current user principal is not of type [" + paramType.getName() + "]: " + userPrincipal);
            }
            return userPrincipal;
        }
        else if (HttpMethod.class == paramType) {
            return HttpMethod.resolve(request.getMethod());
        }
        else if (Locale.class == paramType) {
            return RequestContextUtils.getLocale(request);
        }
        else if (TimeZone.class == paramType) {
            TimeZone timeZone = RequestContextUtils.getTimeZone(request);
            return (timeZone != null ? timeZone : TimeZone.getDefault());
        }
        else if (ZoneId.class == paramType) {
            TimeZone timeZone = RequestContextUtils.getTimeZone(request);
            return (timeZone != null ? timeZone.toZoneId() : ZoneId.systemDefault());
        }

        // Should never happen...
        throw new UnsupportedOperationException("Unknown parameter type: " + paramType.getName());
    }


    /**
     * Inner class to avoid a hard dependency on Servlet API 4.0 at runtime.
     */
    private static class PushBuilderDelegate {

        @Nullable
        public static Object resolvePushBuilder(HttpServletRequest request, Class<?> paramType) {
            PushBuilder pushBuilder = request.newPushBuilder();
            if (pushBuilder != null && !paramType.isInstance(pushBuilder)) {
                throw new IllegalStateException(
                        "Current push builder is not of type [" + paramType.getName() + "]: " + pushBuilder);
            }
            return pushBuilder;

        }
    }
}
```

用例：

```java
@Controller
public class RequestController {

    @GetMapping("/goto")
    public String goToPage(HttpServletRequest request){

        request.setAttribute("msg","成功了...");
        request.setAttribute("code",200);
        return "forward:/success";  //转发到  /success请求
    }
}
```

34、请求处理-【源码分析】-Model、Map原理

复杂参数：

- **Map（map里面的数据会被放在request的请求域request.setAttribute中）**
- **Model（model里面的数据会被放在request的请求域request.setAttribute中）**
- Errors/BindingResult
- **RedirectAttributes（重定向携带数据）**
- **ServletResponse（response响应）**
- SessionStatus
- UriComponentsBuilder
- ServletUriComponentsBuilder

用例：

```java
@GetMapping("/params")
public String testParam(Map<String,Object> map,
                        Model model,
                        HttpServletRequest request,
                        HttpServletResponse response){
    //下面三位都是可以给request域中放数据
    map.put("hello","world666");
    model.addAttribute("world","hello666");
    request.setAttribute("message","HelloWorld");

    Cookie cookie = new Cookie("c1","v1");
    response.addCookie(cookie);
    return "forward:/success";
}

@ResponseBody
@GetMapping("/success")
public Map success(@RequestAttribute(value = "msg",required = false) String msg, // 设置
required=false的作用：说明这两个参数不是必须的
                   @RequestAttribute(value = "code",required = false)Integer code,
                   HttpServletRequest request){
    Object msg1 = request.getAttribute("msg");

    Map<String,Object> map = new HashMap<>();
    Object hello = request.getAttribute("hello"); //得出testParam方法赋予的值 world666
    Object world = request.getAttribute("world"); //得出testParam方法赋予的值 hello666
    Object message = request.getAttribute("message"); //得出testParam方法赋予的值 HelloWorld

    map.put("reqMethod_msg",msg1);
    map.put("annotation_msg",msg);
    map.put("hello",hello);
    map.put("world",world);
    map.put("message",message);

    return map;
}
```

- `Map<String,Object> map`
- `Model model`
- `HttpServletRequest request`

上面三位都是可以给request域中放数据，用`request.getAttribute()`获取

接下来我们看看，`Map<String,Object> map`与`Model model`用什么参数处理器。

`Map<String,Object> map`参数用`MapMethodProcessor`处理：

```java
public class MapMethodProcessor implements HandlerMethodArgumentResolver, HandlerMethodReturnValueHandler {

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        return (Map.class.isAssignableFrom(parameter.getParameterType()) &&
                parameter.getParameterAnnotations().length == 0);
    }

    @Override
    @Nullable
    public Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
            NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {

        Assert.state(mavContainer != null, "ModelAndViewContainer is required for model exposure");
        return mavContainer.getModel();
    }

    ...

}
```

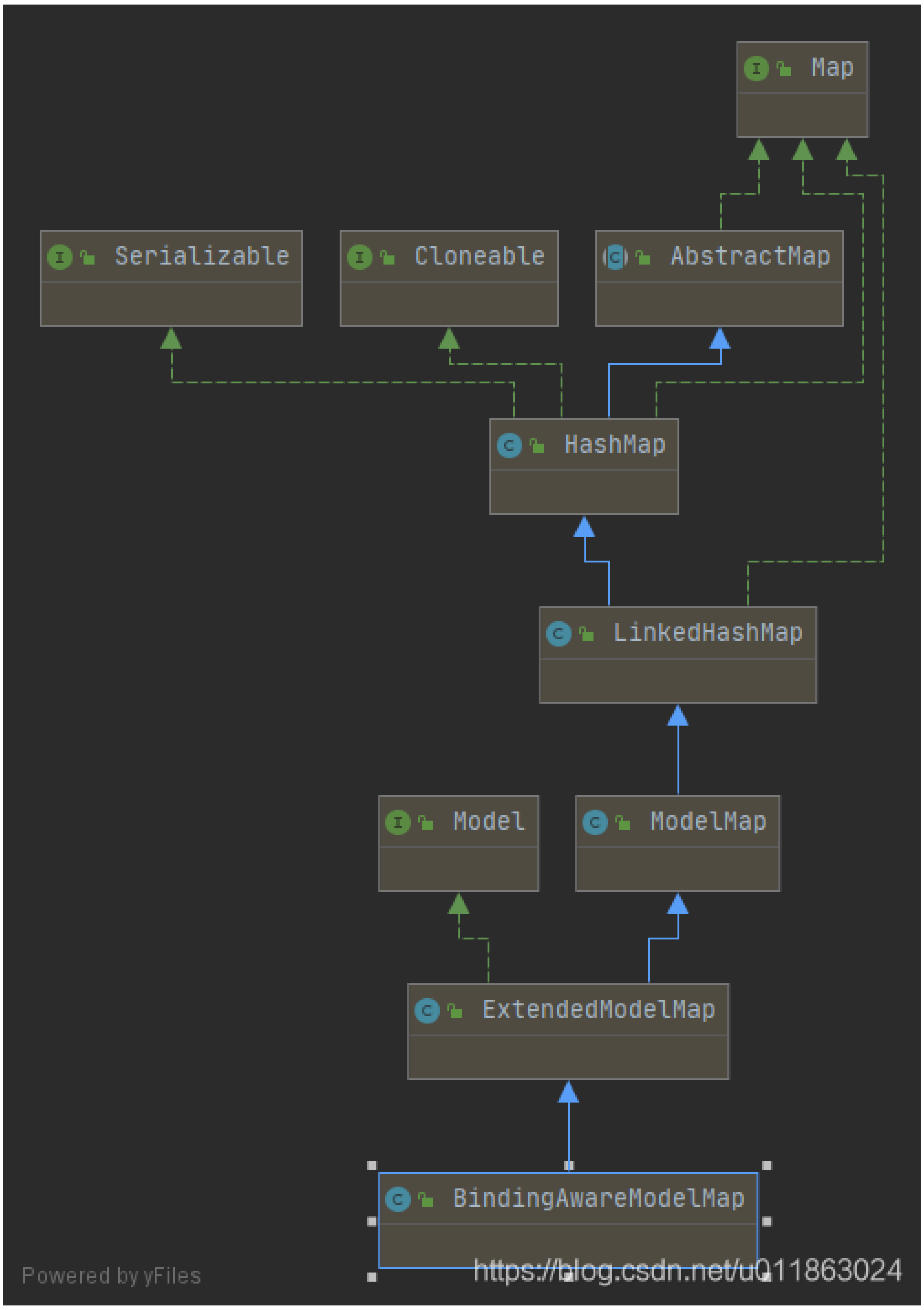
`mavContainer.getModel()`如下：

```java
public class ModelAndViewContainer {

    ...

    private final ModelMap defaultModel = new BindingAwareModelMap();

    @Nullable
    private ModelMap redirectModel;

    ...

    public ModelMap getModel() {
        if (useDefaultModel()) {
            return this.defaultModel;
        }
        else {
            if (this.redirectModel == null) {
                this.redirectModel = new ModelMap();
            }
            return this.redirectModel;
        }
    }

    private boolean useDefaultModel() {
        return (!this.redirectModelScenario || (this.redirectModel == null && 
!this.ignoreDefaultModelOnRedirect));
    }
    ...

}
```

`Model model`用`ModelMethodProcessor`处理:

```java
public class ModelMethodProcessor implements HandlerMethodArgumentResolver, 
HandlerMethodReturnValueHandler {

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        return Model.class.isAssignableFrom(parameter.getParameterType());
    }

    @Override
    @Nullable
    public Object resolveArgument(MethodParameter parameter, @Nullable 
ModelAndViewContainer mavContainer,
            NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) 
throws Exception {

        Assert.state(mavContainer != null, "ModelAndViewContainer is required for model 
exposure");
        return mavContainer.getModel();
    }
    ...
}
```

`return mavContainer.getModel();`这跟`MapMethodProcessor`的一致



`Model`也是另一种意义的`Map`。

接下来看看`Map<String,Object> map`与`Model model`值是如何做到用`request.getAttribute()`获取的。

众所周知，所有的数据都放在 **ModelAndView**包含要去的页面地址View，还包含Model数据。

先看**ModelAndView**接下来是如何处理的?

```java
public class DispatcherServlet extends FrameworkServlet {

    ...

    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        ...

        try {
            ModelAndView mv = null;

            ...

            // Actually invoke the handler.
            mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

            ...

            }
            catch (Exception ex) {
                dispatchException = ex;
            }
            catch (Throwable err) {
                // As of 4.3, we're processing Errors thrown from handler methods as well,
                // making them available for @ExceptionHandler methods and other scenarios.
                dispatchException = new NestedServletException("Handler dispatch failed", err);
            }
            //处理分发结果
            processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
        }
        ...

    }

    private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
            @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,
            @Nullable Exception exception) throws Exception {
        ...

        // Did the handler return a view to render?
        if (mv != null && !mv.wasCleared()) {
            render(mv, request, response);
            ...
        }
        ...
    }

    protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) throws Exception {
        ...

        View view;
        String viewName = mv.getViewName();
        if (viewName != null) {
            // We need to resolve the view name.
            view = resolveViewName(viewName, mv.getModelInternal(), locale, request);
            if (view == null) {
                throw new ServletException("Could not resolve view with name '" + mv.getViewName() +
                        "' in servlet with name '" + getServletName() + "'");
            }
        }
        else {
            // No need to lookup: the ModelAndView object contains the actual View object.
            view = mv.getView();
            if (view == null) {
                throw new ServletException("ModelAndView [" + mv + "] neither contains a 
view name nor a " +
                        "View object in servlet with name '" + getServletName() + "'");
            }
        }
        view.render(mv.getModelInternal(), request, response);

        ...
    }

}
```

在Debug模式下，`view`属为`InternalResourceView`类。

```
public class InternalResourceView extends AbstractUrlBasedView {

    @Override//该方法在AbstractView，AbstractUrlBasedView继承了AbstractView
    public void render(@Nullable Map<String, ?> model, HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        ...

        Map<String, Object> mergedModel = createMergedOutputModel(model, request, 
response);
        prepareResponse(request, response);

        //看下一个方法实现
        renderMergedOutputModel(mergedModel, getRequestToExpose(request), response);
    }

    @Override
    protected void renderMergedOutputModel(
            Map<String, Object> model, HttpServletRequest request, HttpServletResponse 
response) throws Exception {

        // Expose the model object as request attributes.
        // 暴露模型作为请求域属性
        exposeModelAsRequestAttributes(model, request);//<---重点

        // Expose helpers as request attributes, if any.
        exposeHelpers(request);

        // Determine the path for the request dispatcher.
        String dispatcherPath = prepareForRendering(request, response);

        // Obtain a RequestDispatcher for the target resource (typically a JSP).
        RequestDispatcher rd = getRequestDispatcher(request, dispatcherPath);

        ...
    }

    //该方法在AbstractView，AbstractUrlBasedView继承了AbstractView
    protected void exposeModelAsRequestAttributes(Map<String, Object> model,
            HttpServletRequest request) throws Exception {

        model.forEach((name, value) -> {
            if (value != null) {
                request.setAttribute(name, value);
            }
            else {
                request.removeAttribute(name);
            }
        });
    }

}
```

`exposeModelAsRequestAttributes`方法看出，`Map<String,Object> map`，`Model model`这两种类型数据可以给request域中放数据，用`request.getAttribute()`获取。

小结：

Model和Map类型的数据本质上是一样的，在底层实现上都共同作为一个类似Map的对象，最后保存到作用域的时候都是通过`request.setAttribute`方法实现的。

35、请求处理-【源码分析】-自定义参数绑定原理

小结：找到ServletModelAttributeMethodProcessor对参数进行解析，并判断参数是不是"简单类型"（枚举、字符串、数字、日期等），

```java
@RestController
public class ParameterTestController {

    /**
     * 数据绑定：页面提交的请求数据（GET、POST）都可以和对象属性进行绑定
     * @param person
     * @return
     */
    @PostMapping("/saveuser")
    public Person saveuser(Person person){
        return person;
    }
}
```

```java
/**
 *     姓名： <input name="userName"/> <br/>
 *     年龄： <input name="age"/> <br/>
 *     生日： <input name="birth"/> <br/>
 *     宠物姓名：<input name="pet.name"/><br/>
 *     宠物年龄：<input name="pet.age"/>
 */
@Data
public class Person {

    private String userName;
    private Integer age;
    private Date birth;
    private Pet pet;

}

@Data
public class Pet {

    private String name;
    private String age;

}
```

封装过程用到`ServletModelAttributeMethodProcessor`

```java
public class ServletModelAttributeMethodProcessor extends ModelAttributeMethodProcessor {

    @Override//本方法在ModelAttributeMethodProcessor类，
    public boolean supportsParameter(MethodParameter parameter) {
        return (parameter.hasParameterAnnotation(ModelAttribute.class) ||
                (this.annotationNotRequired && !BeanUtils.isSimpleProperty(parameter.getParameterType())));
    }

    @Override
    @Nullable//本方法在ModelAttributeMethodProcessor类，
    public final Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
            NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {

        ...

        String name = ModelFactory.getNameForParameter(parameter);
        ModelAttribute ann = parameter.getParameterAnnotation(ModelAttribute.class);
        if (ann != null) {
            mavContainer.setBinding(name, ann.binding());
        }

        Object attribute = null;
        BindingResult bindingResult = null;

        if (mavContainer.containsAttribute(name)) {
            attribute = mavContainer.getModel().get(name);
        }
        else {
            // Create attribute instance
            try {
                attribute = createAttribute(name, parameter, binderFactory, webRequest);
            }
            catch (BindException ex) {
                ...
            }
        }

        if (bindingResult == null) {
            // Bean property binding and validation;
            // skipped in case of binding failure on construction.
            WebDataBinder binder = binderFactory.createBinder(webRequest, attribute, name);
            if (binder.getTarget() != null) {
                if (!mavContainer.isBindingDisabled(name)) {
                    //web数据绑定器，将请求参数的值绑定到指定的JavaBean里面**
                    bindRequestParameters(binder, webRequest);
                }
                validateIfApplicable(binder, parameter);
                if (binder.getBindingResult().hasErrors() &&
isBindExceptionRequired(binder, parameter)) {
                    throw new BindException(binder.getBindingResult());
                }
            }
            // Value type adaptation, also covering java.util.Optional
            if (!parameter.getParameterType().isInstance(attribute)) {
                attribute = binder.convertIfNecessary(binder.getTarget(),
parameter.getParameterType(), parameter);
            }
            bindingResult = binder.getBindingResult();
        }

        // Add resolved attribute and BindingResult at the end of the model
        Map<String, Object> bindingResultModel = bindingResult.getModel();
        mavContainer.removeAttributes(bindingResultModel);
        mavContainer.addAllAttributes(bindingResultModel);

        return attribute;
    }
}
```

WebDataBinder 利用它里面的 Converters 将请求数据转成指定的数据类型。再次封装到JavaBean中

在过程当中，用到GenericConversionService：在设置每一个值的时候，找它里面的所有converter那个可以将这个数据类型（request带来参数的字符串）转换到指定的类型

36、请求处理-【源码分析】-自定义Converter原理

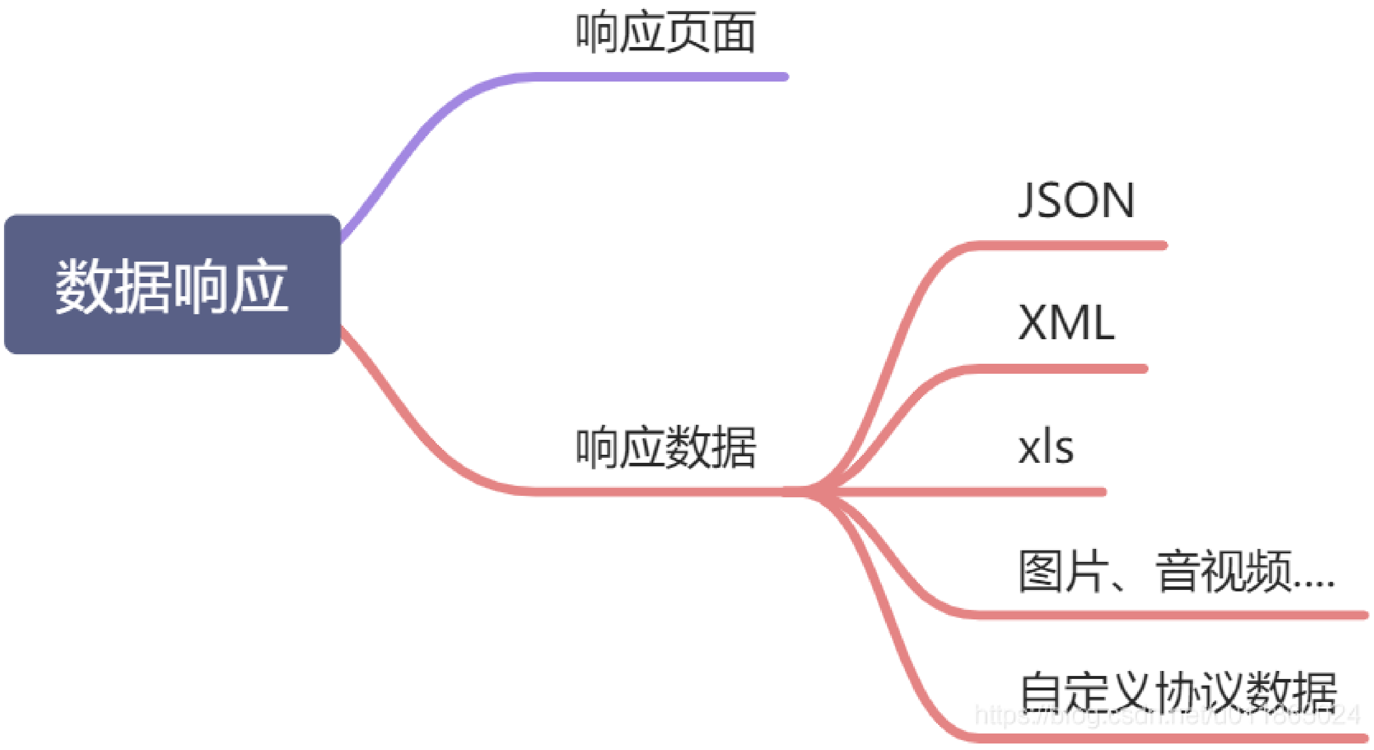
未来我们可以给WebDataBinder里面放自己的Converter;

下面演示将字符串`“啊猫,3”`转换成`Pet`对象。

```
    //1、WebMvcConfigurer定制化SpringMVC的功能
    @Bean
    public WebMvcConfigurer webMvcConfigurer(){
        return new WebMvcConfigurer() {

            @Override
            public void addFormatters(FormatterRegistry registry) {
                registry.addConverter(new Converter<String, Pet>() {

                    @Override
                    public Pet convert(String source) {
                        // 啊猫,3
                        if(!StringUtils.isEmpty(source)){
                            Pet pet = new Pet();
                            String[] split = source.split(",");
                            pet.setName(split[0]);
                            pet.setAge(Integer.parseInt(split[1]));
                            return pet;
                        }
                        return null;
                    }
                });
            }
        };
    }
```

37、响应处理-【源码分析】-ReturnValueHandler原理

- 返回值处理器判断是否支持这种类型返回值
- 返回值处理器调用handleReturnValue进行处理



假设给前端自动返回json数据，需要引入相关的依赖

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

<!-- web场景自动引入了json场景 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-json</artifactId>
    <version>2.3.4.RELEASE</version>
    <scope>compile</scope>
</dependency>
```

控制层代码如下：

```java
@Controller
public class ResponseTestController {

    @ResponseBody  //利用返回值处理器里面的消息转换器进行处理
    @GetMapping(value = "/test/person")
    public Person getPerson(){
        Person person = new Person();
        person.setAge(28);
        person.setBirth(new Date());
        person.setUserName("zhangsan");
        return person;
    }

}
```

32、请求处理-【源码分析】-各种类型参数解析原理 - 返回值处理器有讨论**ReturnValueHandler**。现在直接看看重点：

```java
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {

    ...

    @Nullable
    protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

        ServletWebRequest webRequest = new ServletWebRequest(request, response);
        try {

            ...

            ServletInvocableHandlerMethod invocableMethod =
createInvocableHandlerMethod(handlerMethod);

            if (this.argumentResolvers != null) {
                invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
            }
            if (this.returnValueHandlers != null) {//<----关注点

invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
            }

            ...

            invocableMethod.invokeAndHandle(webRequest, mavContainer);//看下块代码
            if (asyncManager.isConcurrentHandlingStarted()) {
                return null;
            }

            return getModelAndView(mavContainer, modelFactory, webRequest);
        }
        finally {
            webRequest.requestCompleted();
        }
    }
```

```java
public class ServletInvocableHandlerMethod extends InvocableHandlerMethod {

    public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,
            Object... providedArgs) throws Exception {

        Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);

        ...

        try {
            //看下块代码
            this.returnValueHandlers.handleReturnValue(
                    returnValue, getReturnValueType(returnValue), mavContainer, webRequest);
        }
        catch (Exception ex) {
            ...
        }
    }
```

```java
public class HandlerMethodReturnValueHandlerComposite implements HandlerMethodReturnValueHandler {

    ...

    @Override
    public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
            ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception {

        //selectHandler()实现在下面
        HandlerMethodReturnValueHandler handler = selectHandler(returnValue, returnType);
        if (handler == null) {
            throw new IllegalArgumentException("Unknown return value type: " + returnType.getParameterType().getName());
        }
        //开始处理
        handler.handleReturnValue(returnValue, returnType, mavContainer, webRequest);
    }

    @Nullable
    private HandlerMethodReturnValueHandler selectHandler(@Nullable Object value, MethodParameter returnType) {
        boolean isAsyncValue = isAsyncReturnValue(value, returnType);
        for (HandlerMethodReturnValueHandler handler : this.returnValueHandlers) {
            if (isAsyncValue && !(handler instanceof AsyncHandlerMethodReturnValueHandler)) {
                continue;
            }
            if (handler.supportsReturnType(returnType)) {
                return handler;
            }
        }
        return null;
    }

```

`@ResponseBody` 注解，即 `RequestResponseBodyMethodProcessor`，它实现 `HandlerMethodReturnValueHandler` 接口

```java
public class RequestResponseBodyMethodProcessor extends AbstractMessageConverterMethodProcessor {

    ...

    @Override
    public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
            ModelAndViewContainer mavContainer, NativeWebRequest webRequest)
            throws IOException, HttpMediaTypeNotAcceptableException, HttpMessageNotWritableException {

        mavContainer.setRequestHandled(true);
        ServletServerHttpRequest inputMessage = createInputMessage(webRequest);
        ServletServerHttpResponse outputMessage = createOutputMessage(webRequest);

        // 使用消息转换器进行写出操作，本方法下一章节介绍：
        // Try even with null return value. ResponseBodyAdvice could get involved.
        writeWithMessageConverters(returnValue, returnType, inputMessage, outputMessage);
    }

}
```

- springMVC支持哪些返回值?
  - ModelAndView
  - Model
  - View
  - ResponseEntity
  - ResponseBodyEmitter
  - StreamingResponseBody
  - HttpEntity
  - HttpHeaders
  - Callable
  - DeferredResult
  - ListenableFuture
  - CompletionStage
  - WebAsyncTask
  - @ModelAttribute
  - @ResponseBody

38、响应处理-【源码分析】-HTTPMessageConverter原理（消息转换器）

消息转换器：看是否支持将此class类型的对象，转为MediaType类型的数据。

- 一些默认的MessageConverter

  - ByteArrayHttpMessageConverter：支持Byte
  - StringHttpMessageConverter：支持String（utf-8）
  - StringHttpMessageConverter：支持String（ISO-8859-1）
  - ResourceHttpMessageConverter：支持Resource
  - ResourceRegoinHttpMessageConverter：支持ResourceRegion
  - SourceHttpMessageConverter：支持Source
  - AllEncompassingFormHttpMessageConverter：：支持MultiValueMap
  - MappingJackson2HttpMessageConverter：全部支持
  - MappingJackson2HttpMessageConverter：全部支持
  - Jaxb2RootElementHttpMessageConverter：支持注解方式xml处理的。

返回值处理器`ReturnValueHandler`原理：

1. 返回值处理器判断是否支持这种类型返回值 `supportsReturnType`
2. 返回值处理器调用 `handleReturnValue` 进行处理
3. `RequestResponseBodyMethodProcessor` 可以处理**返回值标了** `@ResponseBody` **注解**的。
   - 利用 `MessageConverters` 进行处理 将数据写为json
     1. 内容协商（浏览器默认会以请求头的方式告诉服务器他能接受什么样的内容类型）
     2. 服务器最终根据自己自身的能力，决定服务器能生产出什么样内容类型的数据，
     3. SpringMVC会挨个遍历所有容器底层的 `HttpMessageConverter` ，看谁能处理？
        1. 得到`MappingJackson2HttpMessageConverter`可以将对象写为json
        2. 利用`MappingJackson2HttpMessageConverter`将对象转为json再写出去。

```java

//RequestResponseBodyMethodProcessor继承这类
public abstract class AbstractMessageConverterMethodProcessor extends AbstractMessageConverterMethodArgumentResolver
		implements HandlerMethodReturnValueHandler {

    ...

    //承接上一节内容
    protected <T> void writeWithMessageConverters(@Nullable T value, MethodParameter returnType,
                ServletServerHttpRequest inputMessage, ServletServerHttpResponse outputMessage)
                throws IOException, HttpMediaTypeNotAcceptableException, HttpMessageNotWritableException {

            Object body;
            Class<?> valueType;
            Type targetType;

            if (value instanceof CharSequence) {
                body = value.toString();
                valueType = String.class;
                targetType = String.class;
            }
            else {
                body = value;
                valueType = getReturnValueType(body, returnType);
                targetType = GenericTypeResolver.resolveType(getGenericType(returnType), returnType.getContainingClass());
            }

            ...

            //内容协商（浏览器默认会以请求头(参数Accept)的方式告诉服务器他能接受什么样的内容类型）
            MediaType selectedMediaType = null;
            MediaType contentType = outputMessage.getHeaders().getContentType();
            boolean isContentTypePreset = contentType != null && contentType.isConcrete();
            if (isContentTypePreset) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Found 'Content-Type:" + contentType + "' in response");
                }
                selectedMediaType = contentType;
            }
            else {
                HttpServletRequest request = inputMessage.getServletRequest();
                List<MediaType> acceptableTypes = getAcceptableMediaTypes(request);
                //服务器最终根据自己自身的能力，决定服务器能生产出什么样内容类型的数据
                List<MediaType> producibleTypes = getProducibleMediaTypes(request, valueType, targetType);

                if (body != null && producibleTypes.isEmpty()) {
                    throw new HttpMessageNotWritableException(
                            "No converter found for return value of type: " + valueType);
                }
                List<MediaType> mediaTypesToUse = new ArrayList<>();
                for (MediaType requestedType : acceptableTypes) {
                    for (MediaType producibleType : producibleTypes) {
                        if (requestedType.isCompatibleWith(producibleType)) {
                            mediaTypesToUse.add(getMostSpecificMediaType(requestedType, producibleType));
                        }
                    }
                }
                if (mediaTypesToUse.isEmpty()) {
                    if (body != null) {
                        throw new HttpMediaTypeNotAcceptableException(producibleTypes);
                    }
                    if (logger.isDebugEnabled()) {
                        logger.debug("No match for " + acceptableTypes + ", supported: " + producibleTypes);
                    }
                    return;
                }

                MediaType.sortBySpecificityAndQuality(mediaTypesToUse);

                //选择一个MediaType
                for (MediaType mediaType : mediaTypesToUse) {
                    if (mediaType.isConcrete()) {
                        selectedMediaType = mediaType;
                        break;
                    }
                    else if (mediaType.isPresentIn(ALL_APPLICATION_MEDIA_TYPES)) {
                        selectedMediaType = MediaType.APPLICATION_OCTET_STREAM;
                        break;
                    }
                }

                if (logger.isDebugEnabled()) {
                    logger.debug("Using '" + selectedMediaType + "', given " +
                            acceptableTypes + " and supported " + producibleTypes);
                }
            }


            if (selectedMediaType != null) {
                selectedMediaType = selectedMediaType.removeQualityValue();
                //本节主角：HttpMessageConverter
                for (HttpMessageConverter<?> converter : this.messageConverters) {
                    GenericHttpMessageConverter genericConverter = (converter instanceof GenericHttpMessageConverter ?
                            (GenericHttpMessageConverter<?>) converter : null);

                    //判断是否可写
                    if (genericConverter != null ?
                            ((GenericHttpMessageConverter) converter).canWrite(targetType, valueType, selectedMediaType) :
                            converter.canWrite(valueType, selectedMediaType)) {
                        body = getAdvice().beforeBodyWrite(body, returnType, selectedMediaType,
                                (Class<? extends HttpMessageConverter<?>>) converter.getClass(),
                                inputMessage, outputMessage);
                        if (body != null) {
                            Object theBody = body;
                            LogFormatUtils.traceDebug(logger, traceOn ->
                                    "Writing [" + LogFormatUtils.formatValue(theBody, !traceOn) + "]");
                            addContentDispositionHeader(inputMessage, outputMessage);
                            //开始写入
                            if (genericConverter != null) {
                                genericConverter.write(body, targetType, selectedMediaType, outputMessage);
                            }
                            else {
                                ((HttpMessageConverter) converter).write(body, selectedMediaType, outputMessage);
                            }
                        }
                        else {
                            if (logger.isDebugEnabled()) {
                                logger.debug("Nothing to write: null body");
                            }
                        }
                        return;
                    }
                }
            }
            ...
        }
```

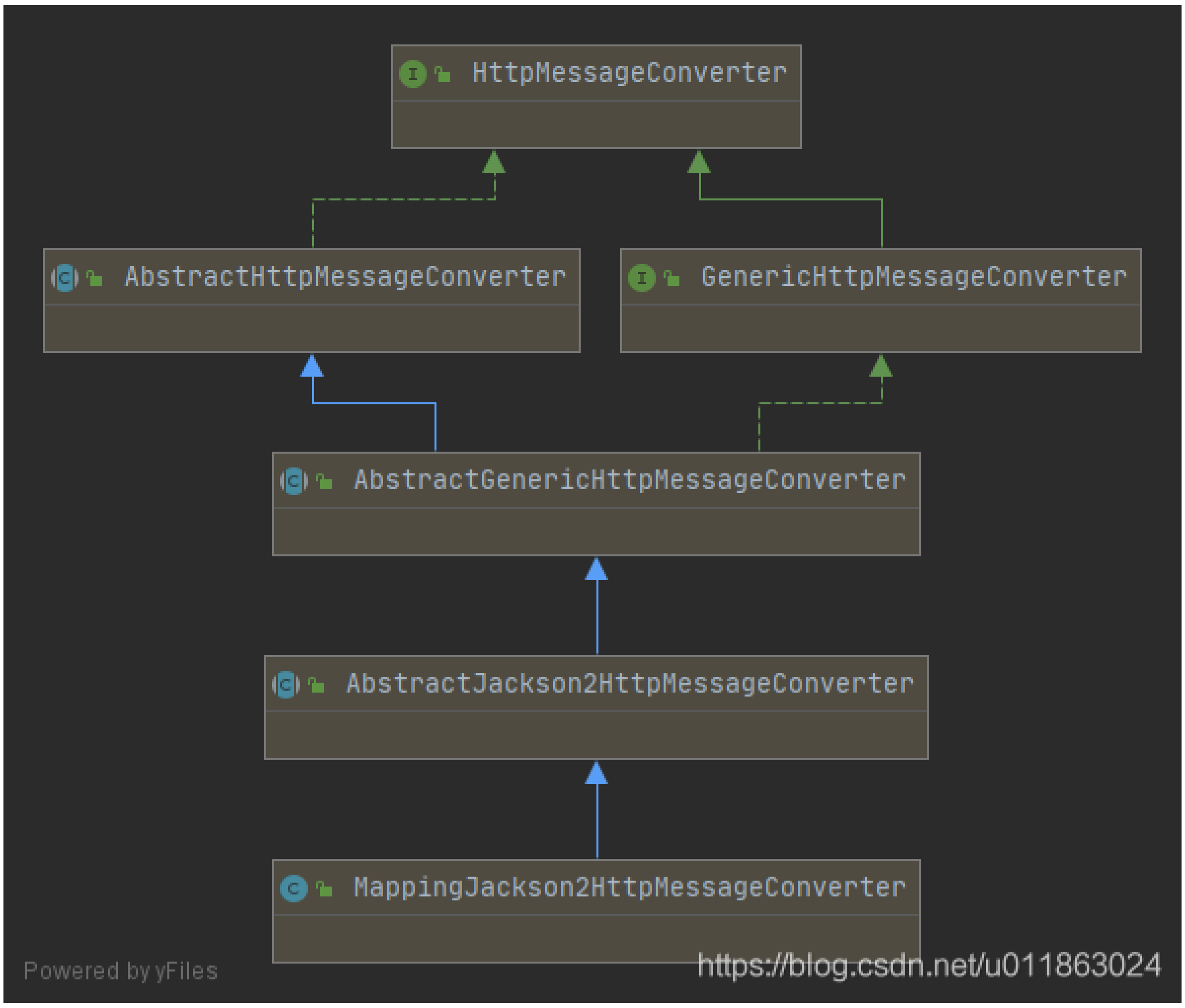
`HTTPMessageConverter`接口：

```java
/**
 * Strategy interface for converting from and to HTTP requests and responses.
 */
public interface HttpMessageConverter<T> {

    /**
```

```java
     * Indicates whether the given class can be read by this converter.
     */
    boolean canRead(Class<?> clazz, @Nullable MediaType mediaType);

    /**
     * Indicates whether the given class can be written by this converter.
     */
    boolean canWrite(Class<?> clazz, @Nullable MediaType mediaType);

    /**
     * Return the list of {@link MediaType} objects supported by this converter.
     */
    List<MediaType> getSupportedMediaTypes();

    /**
     * Read an object of the given type from the given input message, and returns it.
     */
    T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
            throws IOException, HttpMessageNotReadableException;

    /**
     * Write an given object to the given output message.
     */
    void write(T t, @Nullable MediaType contentType, HttpOutputMessage outputMessage)
            throws IOException, HttpMessageNotWritableException;

}

```

`HttpMessageConverter`: 看是否支持将 此 `Class`类型的对象，转为`MediaType`类型的数据。

例子：`Person`对象转为JSON，或者 JSON转为`Person`，这将用到`MappingJackson2HttpMessageConverter`


HttpMessageConverter
AbstractHttpMessageConverter
GenericHttpMessageConverter
AbstractGenericHttpMessageConverter
AbstractJackson2HttpMessageConverter
MappingJackson2HttpMessageConverter
Powered by yFiles
https://blog.csdn.net/u011863024

```
public class MappingJackson2HttpMessageConverter extends
AbstractJackson2HttpMessageConverter {
    ...
}
```

关于`MappingJackson2HttpMessageConverter`的实例化请看下节。

关于HttpMessageConverters的初始化

`DispatcherServlet`的初始化时会调用`initHandlerAdapters(ApplicationContext context)`

```
public class DispatcherServlet extends FrameworkServlet {

    ...

    private void initHandlerAdapters(ApplicationContext context) {
        this.handlerAdapters = null;

        if (this.detectAllHandlerAdapters) {
            // Find all HandlerAdapters in the ApplicationContext, including ancestor
contexts.
            Map<String, HandlerAdapter> matchingBeans =
                    BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
HandlerAdapter.class, true, false);
            if (!matchingBeans.isEmpty()) {
                this.handlerAdapters = new ArrayList<>(matchingBeans.values());
                // We keep HandlerAdapters in sorted order.
                AnnotationAwareOrderComparator.sort(this.handlerAdapters);
            }
        }
    ...
```

上述代码会加载`ApplicationContext`的所有`HandlerAdapter`，用来处理`@RequestMapping`的`RequestMappingHandlerAdapter`实现`HandlerAdapter`接口，`RequestMappingHandlerAdapter`也被实例化。

```java
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {

    ...

    private List<HttpMessageConverter<?>> messageConverters;

    ...

    public RequestMappingHandlerAdapter() {
        this.messageConverters = new ArrayList<>(4);
        this.messageConverters.add(new ByteArrayHttpMessageConverter());
        this.messageConverters.add(new StringHttpMessageConverter());
        if (!shouldIgnoreXml) {
            try {
                this.messageConverters.add(new SourceHttpMessageConverter<>());
            }
            catch (Error err) {
                // Ignore when no TransformerFactory implementation is available
            }
        }
        this.messageConverters.add(new AllEncompassingFormHttpMessageConverter());
    }
```

在构造器中看到**一堆**`HttpMessageConverter`。接着，重点查看`AllEncompassingFormHttpMessageConverter`类：

```java
public class AllEncompassingFormHttpMessageConverter extends FormHttpMessageConverter {

    /**
     * Boolean flag controlled by a {@code spring.xml.ignore} system property that
instructs Spring to
     * ignore XML, i.e. to not initialize the XML-related infrastructure.
     * <p>The default is "false".
     */
    private static final boolean shouldIgnoreXml =
SpringProperties.getFlag("spring.xml.ignore");

    private static final boolean jaxb2Present;

    private static final boolean jackson2Present;

    private static final boolean jackson2XmlPresent;

    private static final boolean jackson2SmilePresent;

    private static final boolean gsonPresent;

    private static final boolean jsonbPresent;

    private static final boolean kotlinSerializationJsonPresent;

    static {
        ClassLoader classLoader = AllEncompassingFormHttpMessageConverter.class.getClassLoader();
        jaxb2Present = ClassUtils.isPresent("javax.xml.bind.Binder", classLoader);
        jackson2Present = ClassUtils.isPresent("com.fasterxml.jackson.databind.ObjectMapper", classLoader) &&
                        ClassUtils.isPresent("com.fasterxml.jackson.core.JsonGenerator", classLoader);
        jackson2XmlPresent = ClassUtils.isPresent("com.fasterxml.jackson.dataformat.xml.XmlMapper", classLoader);
        jackson2SmilePresent = ClassUtils.isPresent("com.fasterxml.jackson.dataformat.smile.SmileFactory", classLoader);
        gsonPresent = ClassUtils.isPresent("com.google.gson.Gson", classLoader);
        jsonbPresent = ClassUtils.isPresent("javax.json.bind.Jsonb", classLoader);
        kotlinSerializationJsonPresent = ClassUtils.isPresent("kotlinx.serialization.json.Json", classLoader);
    }


    public AllEncompassingFormHttpMessageConverter() {
        if (!shouldIgnoreXml) {
            try {
                addPartConverter(new SourceHttpMessageConverter<>());
            }
            catch (Error err) {
                // Ignore when no TransformerFactory implementation is available
            }

            if (jaxb2Present && !jackson2XmlPresent) {
                addPartConverter(new Jaxb2RootElementHttpMessageConverter());
            }
        }

        if (jackson2Present) {
            addPartConverter(new MappingJackson2HttpMessageConverter());//<----重点看这里
        }
        else if (gsonPresent) {
            addPartConverter(new GsonHttpMessageConverter());
        }
        else if (jsonbPresent) {
            addPartConverter(new JsonbHttpMessageConverter());
        }
        else if (kotlinSerializationJsonPresent) {
            addPartConverter(new KotlinSerializationJsonHttpMessageConverter());
        }

        if (jackson2XmlPresent && !shouldIgnoreXml) {
            addPartConverter(new MappingJackson2XmlHttpMessageConverter());
        }

        if (jackson2SmilePresent) {
            addPartConverter(new MappingJackson2SmileHttpMessageConverter());
        }
    }

}

public class FormHttpMessageConverter implements HttpMessageConverter<MultiValueMap<String, ?>> {

    ...

    private List<HttpMessageConverter<?>> partConverters = new ArrayList<>();

    ...

    public void addPartConverter(HttpMessageConverter<?> partConverter) {
        Assert.notNull(partConverter, "'partConverter' must not be null");
        this.partConverters.add(partConverter);
    }

    ...
}

```

在`AllEncompassingFormHttpMessageConverter`类构造器看到`MappingJackson2HttpMessageConverter`类的实例化，`AllEncompassingFormHttpMessageConverter`**包含**`MappingJackson2HttpMessageConverter`。

`ReturnValueHandler`是怎么与`MappingJackson2HttpMessageConverter`关联起来？请看下节。

ReturnValueHandler与MappingJackson2HttpMessageConverter关联

再次回顾`RequestMappingHandlerAdapter`

```java
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {

    ...
    @Nullable
    private HandlerMethodReturnValueHandlerComposite returnValueHandlers;//我们关注的returnValueHandlers


    @Override
    @Nullable//本方法在AbstractHandlerMethodAdapter
    public final ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {

        return handleInternal(request, response, (HandlerMethod) handler);
    }

    @Override
    protected ModelAndView handleInternal(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
        ModelAndView mav;
        ...
        mav = invokeHandlerMethod(request, response, handlerMethod);
        ...
        return mav;
    }

    @Nullable
    protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

        ServletWebRequest webRequest = new ServletWebRequest(request, response);
        try {
            WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);
            ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);

            ServletInvocableHandlerMethod invocableMethod = createInvocableHandlerMethod(handlerMethod);
            if (this.argumentResolvers != null) {
                invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
            }
            if (this.returnValueHandlers != null) {//<---我们关注的returnValueHandlers
                invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
            }

            ...

            invocableMethod.invokeAndHandle(webRequest, mavContainer);
            if (asyncManager.isConcurrentHandlingStarted()) {
                return null;
            }

            return getModelAndView(mavContainer, modelFactory, webRequest);
        }
        finally {
            webRequest.requestCompleted();
        }
    }

    @Override
    public void afterPropertiesSet() {
        // Do this first, it may add ResponseBody advice beans

        ...

        if (this.returnValueHandlers == null) {//赋值returnValueHandlers
            List<HandlerMethodReturnValueHandler> handlers = getDefaultReturnValueHandlers();
            this.returnValueHandlers = new HandlerMethodReturnValueHandlerComposite().addHandlers(handlers);
        }
    }

    private List<HandlerMethodReturnValueHandler> getDefaultReturnValueHandlers() {
        List<HandlerMethodReturnValueHandler> handlers = new ArrayList<>(20);

        ...
        // Annotation-based return value types
        //这里就是 ReturnValueHandler与 MappingJackson2HttpMessageConverter关联 的关键点
        handlers.add(new RequestResponseBodyMethodProcessor(getMessageConverters(),//<---MessageConverters也就传参传进来的
                this.contentNegotiationManager, this.requestResponseBodyAdvice));//
        ...

        return handlers;
    }

    //------

    public List<HttpMessageConverter<?>> getMessageConverters() {
        return this.messageConverters;
    }

    //RequestMappingHandlerAdapter构造器已初始化部分messageConverters
    public RequestMappingHandlerAdapter() {
        this.messageConverters = new ArrayList<>(4);
        this.messageConverters.add(new ByteArrayHttpMessageConverter());
        this.messageConverters.add(new StringHttpMessageConverter());
        if (!shouldIgnoreXml) {
            try {
                this.messageConverters.add(new SourceHttpMessageConverter<>());
            }
            catch (Error err) {
                // Ignore when no TransformerFactory implementation is available
            }
        }
        this.messageConverters.add(new AllEncompassingFormHttpMessageConverter());
    }

    ...

}
```

应用中`WebMvcAutoConfiguration`（底层是`WebMvcConfigurationSupport`实现）传入更多`messageConverters`，其中就包含`MappingJackson2HttpMessageConverter`。

39、响应处理-【源码分析】-内容协商原理

根据客户端接收能力不同，返回不同媒体类型的数据。

一、引入XML依赖：【支持xml的依赖】

```xml
 <dependency>
     <groupId>com.fasterxml.jackson.dataformat</groupId>
     <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

可用Postman软件分别测试返回json和xml：只需要改变请求头中Accept字段（application/json、application/xml）。

Http协议中规定的，Accept字段告诉服务器本客户端可以接收的数据类型。

二、内容协商原理：

1. 判断当前响应头中是否已经有确定的媒体类型`MediaType`。【看看之前有没有处理过类似的请求，如果有的话按照之前的配置进行返回】
2. 获取客户端（PostMan、浏览器）支持接收的内容类型。（获取客户端Accept请求头字段application/xml）
    - `contentNegotiationManager` 内容协商管理器默认使用基于请求头的策略
    - `HeaderContentNegotiationStrategy` 确定客户端可以接收的内容类型
3. 遍历循环所有当前系统的 `MessageConverter`，看谁支持操作这个对象（Person）
4. 找到支持操作Person的converter，把converter支持的媒体类型统计出来。
5. 客户端需要application/xml，服务端有10种MediaType（也就是通过上面Converter可以获得的类型）
6. 进行内容协商的最佳匹配媒体类型，看看服务器能产生的MediaType能够与哪些产生最佳匹配。
7. 用 支持 将对象转为 最佳匹配媒体类型 的converter。调用它进行转化 。

简单来说，就是先找哪个MessageConverter能够处理数据，然后再找哪个MessageConverter能够将数据转换为客户端指定的那几种接收类型，最后将所有符合条件的MessageConverter进行排序，找到最佳的匹配方式。

```java

//RequestResponseBodyMethodProcessor继承这类
public abstract class AbstractMessageConverterMethodProcessor extends AbstractMessageConverterMethodArgumentResolver
        implements HandlerMethodReturnValueHandler {

    ...

    //跟上一节的代码一致
    protected <T> void writeWithMessageConverters(@Nullable T value, MethodParameter returnType,
                ServletServerHttpRequest inputMessage, ServletServerHttpResponse outputMessage)
                throws IOException, HttpMediaTypeNotAcceptableException, HttpMessageNotWritableException {

            Object body;
            Class<?> valueType;
            Type targetType;

            if (value instanceof CharSequence) {
                body = value.toString();
                valueType = String.class;
                targetType = String.class;
            }
            else {
                body = value;
                valueType = getReturnValueType(body, returnType);
                targetType = GenericTypeResolver.resolveType(getGenericType(returnType), returnType.getContainingClass());
            }

            ...

            //本节重点
            //内容协商（浏览器默认会以请求头(参数Accept)的方式告诉服务器他能接受什么样的内容类型）
            MediaType selectedMediaType = null;
            MediaType contentType = outputMessage.getHeaders().getContentType();
            boolean isContentTypePreset = contentType != null && contentType.isConcrete();
            if (isContentTypePreset) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Found 'Content-Type:" + contentType + "' in response");
                }
                selectedMediaType = contentType;
            }
            else {
                HttpServletRequest request = inputMessage.getServletRequest();
                List<MediaType> acceptableTypes = getAcceptableMediaTypes(request);
                //服务器最终根据自己自身的能力，决定服务器能生产出什么样内容类型的数据
                List<MediaType> producibleTypes = getProducibleMediaTypes(request, valueType, targetType);

                if (body != null && producibleTypes.isEmpty()) {
                    throw new HttpMessageNotWritableException(
                            "No converter found for return value of type: " + valueType);
                }
                List<MediaType> mediaTypesToUse = new ArrayList<>();
                for (MediaType requestedType : acceptableTypes) {
                    for (MediaType producibleType : producibleTypes) {
                        if (requestedType.isCompatibleWith(producibleType)) {
                            mediaTypesToUse.add(getMostSpecificMediaType(requestedType, producibleType));
                        }
                    }
                }
                if (mediaTypesToUse.isEmpty()) {
                    if (body != null) {
                        throw new HttpMediaTypeNotAcceptableException(producibleTypes);
                    }
                    if (logger.isDebugEnabled()) {
                        logger.debug("No match for " + acceptableTypes + ", supported: " + producibleTypes);
                    }
                    return;
                }

                MediaType.sortBySpecificityAndQuality(mediaTypesToUse);

                //选择一个MediaType
                for (MediaType mediaType : mediaTypesToUse) {
                    if (mediaType.isConcrete()) {
                        selectedMediaType = mediaType;
                        break;
                    }
                    else if (mediaType.isPresentIn(ALL_APPLICATION_MEDIA_TYPES)) {
                        selectedMediaType = MediaType.APPLICATION_OCTET_STREAM;
                        break;
                    }
                }

                if (logger.isDebugEnabled()) {
                    logger.debug("Using '" + selectedMediaType + "', given " +
                            acceptableTypes + " and supported " + producibleTypes);
                }
            }


            if (selectedMediaType != null) {
                selectedMediaType = selectedMediaType.removeQualityValue();
                //本节主角：HttpMessageConverter
                for (HttpMessageConverter<?> converter : this.messageConverters) {
                    GenericHttpMessageConverter genericConverter = (converter instanceof GenericHttpMessageConverter ?
                            (GenericHttpMessageConverter<?>) converter : null);

                    //判断是否可写
                    if (genericConverter != null ?
                            ((GenericHttpMessageConverter) converter).canWrite(targetType, valueType, selectedMediaType) :
                            converter.canWrite(valueType, selectedMediaType)) {
                        body = getAdvice().beforeBodyWrite(body, returnType, selectedMediaType,
                                (Class<? extends HttpMessageConverter<?>>) converter.getClass(),
                                inputMessage, outputMessage);
                        if (body != null) {
                            Object theBody = body;
                            LogFormatUtils.traceDebug(logger, traceOn ->
                                    "Writing [" + LogFormatUtils.formatValue(theBody, !traceOn) + "]");
                            addContentDispositionHeader(inputMessage, outputMessage);
                            //开始写入
                            if (genericConverter != null) {
                                genericConverter.write(body, targetType, selectedMediaType, outputMessage);
                            }
                            else {
                                ((HttpMessageConverter) converter).write(body, selectedMediaType, outputMessage);
                            }
                        }
                        else {
                            if (logger.isDebugEnabled()) {
                                logger.debug("Nothing to write: null body");
                            }
                        }
                        return;
                    }
                }
            }
            ...
        }
```

40、响应处理-【源码分析】-基于请求参数的内容协商原理

一、基于请求头方式的内容协商

上一节内容协商原理的第二步：

获取客户端（PostMan、浏览器）支持接收的内容类型。（获取客户端Accept请求头字段application/xml）

- `contentNegotiationManager` 内容协商管理器 默认使用**基于请求头的策略**
- `HeaderContentNegotiationStrategy` 确定客户端可以接收的内容类型

```
//RequestResponseBodyMethodProcessor继承这类
public abstract class AbstractMessageConverterMethodProcessor extends AbstractMessageConverterMethodArgumentResolver
        implements HandlerMethodReturnValueHandler {

    ...

    //跟上一节的代码一致
    protected <T> void writeWithMessageConverters(@Nullable T value, MethodParameter returnType,
                ServletServerHttpRequest inputMessage, ServletServerHttpResponse outputMessage)
                throws IOException, HttpMediaTypeNotAcceptableException, HttpMessageNotWritableException {

```

```java
            Object body;
            Class<?> valueType;
            Type targetType;

            ...

                    //本节重点
            //内容协商（浏览器默认会以请求头(参数Accept)的方式告诉服务器他能接受什么样的内容类型）
            MediaType selectedMediaType = null;
            MediaType contentType = outputMessage.getHeaders().getContentType();
            boolean isContentTypePreset = contentType != null && contentType.isConcrete();
            if (isContentTypePreset) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Found 'Content-Type:" + contentType + "' in response");
                }
                selectedMediaType = contentType;
            }
            else {
                HttpServletRequest request = inputMessage.getServletRequest();
                List<MediaType> acceptableTypes = getAcceptableMediaTypes(request);
                //服务器最终根据自己自身的能力，决定服务器能生产出什么样内容类型的数据
                List<MediaType> producibleTypes = getProducibleMediaTypes(request, valueType, targetType);
            ...

    }

    //在AbstractMessageConverterMethodArgumentResolver类内
    private List<MediaType> getAcceptableMediaTypes(HttpServletRequest request)
            throws HttpMediaTypeNotAcceptableException {

        //内容协商管理器 默认使用基于请求头的策略
        return this.contentNegotiationManager.resolveMediaTypes(new ServletWebRequest(request));
    }

}
```

```java
public class ContentNegotiationManager implements ContentNegotiationStrategy, MediaTypeFileExtensionResolver {

    ...

    public ContentNegotiationManager() {
        this(new HeaderContentNegotiationStrategy());//内容协商管理器 默认使用基于请求头的策略
    }

    @Override
    public List<MediaType> resolveMediaTypes(NativeWebRequest request) throws HttpMediaTypeNotAcceptableException {
        for (ContentNegotiationStrategy strategy : this.strategies) {
            List<MediaType> mediaTypes = strategy.resolveMediaTypes(request);
            if (mediaTypes.equals(MEDIA_TYPE_ALL_LIST)) {
                continue;
            }
            return mediaTypes;
        }
        return MEDIA_TYPE_ALL_LIST;
    }
    ...

}
```

```java
//基于请求头的策略
public class HeaderContentNegotiationStrategy implements ContentNegotiationStrategy {

    /**
     * {@inheritDoc}
     * @throws HttpMediaTypeNotAcceptableException if the 'Accept' header cannot be parsed
     */
    @Override
    public List<MediaType> resolveMediaTypes(NativeWebRequest request)
            throws HttpMediaTypeNotAcceptableException {

        String[] headerValueArray = request.getHeaderValues(HttpHeaders.ACCEPT);
        if (headerValueArray == null) {
            return MEDIA_TYPE_ALL_LIST;
        }

        List<String> headerValues = Arrays.asList(headerValueArray);
        try {
            List<MediaType> mediaTypes = MediaType.parseMediaTypes(headerValues);
            MediaType.sortBySpecificityAndQuality(mediaTypes);
            return !CollectionUtils.isEmpty(mediaTypes) ? mediaTypes :
MEDIA_TYPE_ALL_LIST;
        }
        catch (InvalidMediaTypeException ex) {
            throw new HttpMediaTypeNotAcceptableException(
                    "Could not parse 'Accept' header " + headerValues + ": " +
ex.getMessage());
        }
    }

}
```

二、基于浏览器参数方式的内容协商

为了方便内容协商，开启基于请求参数的内容协商功能。开启浏览器参数内容协商的本质就是，在浏览器的参数中加入一个format参数，指明客户端想要的数据格式。比如：`localhost:8080/test/person?format=json`

```yaml
spring:
  mvc:
    contentnegotiation:
      favor-parameter: true  #开启请求参数内容协商模式
```

内容协商管理器，就会多了一个`ParameterContentNegotiationStrategy`（由Spring容器注入）

```java
public class ParameterContentNegotiationStrategy extends
AbstractMappingContentNegotiationStrategy {

    private String parameterName = "format";//

    /**
     * Create an instance with the given map of file extensions and media types.
     */
    public ParameterContentNegotiationStrategy(Map<String, MediaType> mediaTypes) {
        super(mediaTypes);
    }


    /**
     * Set the name of the parameter to use to determine requested media types.
     * <p>By default this is set to {@code "format"}.
     */
    public void setParameterName(String parameterName) {
        Assert.notNull(parameterName, "'parameterName' is required");
        this.parameterName = parameterName;
    }

    public String getParameterName() {
        return this.parameterName;
    }


    @Override
    @Nullable
    protected String getMediaTypeKey(NativeWebRequest request) {
        return request.getParameter(getParameterName());
    }

    //---以下方法在AbstractMappingContentNegotiationStrategy类

    @Override
    public List<MediaType> resolveMediaTypes(NativeWebRequest webRequest)
            throws HttpMediaTypeNotAcceptableException {

        return resolveMediaTypeKey(webRequest, getMediaTypeKey(webRequest));
    }

    /**
     * An alternative to {@link #resolveMediaTypes(NativeWebRequest)} that accepts
     * an already extracted key.
     * @since 3.2.16
     */
    public List<MediaType> resolveMediaTypeKey(NativeWebRequest webRequest, @Nullable
String key)
            throws HttpMediaTypeNotAcceptableException {

        if (StringUtils.hasText(key)) {
            MediaType mediaType = lookupMediaType(key);
            if (mediaType != null) {
                handleMatch(key, mediaType);
                return Collections.singletonList(mediaType);
            }
            mediaType = handleNoMatch(webRequest, key);
            if (mediaType != null) {
                addMapping(key, mediaType);
                return Collections.singletonList(mediaType);
            }
        }
        return MEDIA_TYPE_ALL_LIST;
    }


}
```

然后，浏览器地址输入带format参数的URL：

```
http://localhost:8080/test/person?format=json
或
http://localhost:8080/test/person?format=xml
```

这样，后端会根据参数format的值，返回对应json或xml格式的数据。

基于参数的内容协商和基于请求头的内容协商同时生效，基于参数（请求参数format）的优先级更高。

内容协商的本质就是找到合适的MessageConverter，一边是能够处理响应数据，一边是能够转换成客户端需要的固定格式。

41、响应处理-【源码分析】-自定义MessageConverter

实现多协议数据兼容。json、xml、x-guigu（这个是自创的）

1. `@ResponseBody` 响应数据出去 调用 `RequestResponseBodyMethodProcessor` 处理
2. Processor 处理方法返回值。通过 `MessageConverter` 处理
3. 所有 `MessageConverter` 合起来可以支持各种媒体类型数据的操作（读、写）
4. 内容协商找到最终的 `messageConverter`

SpringMVC的什么功能，只需创建`WebMvcConfigurer`并进行修改。

```java
@Configuration(proxyBeanMethods = false)
public class WebConfig {
    @Bean
    public WebMvcConfigurer webMvcConfigurer(){
        return new WebMvcConfigurer() {

            @Override
            public void extendMessageConverters(List<HttpMessageConverter<?>> converters)
{
                converters.add(new GuiguMessageConverter());
            }
        }
    }
}
```

```java

/**
 * 自定义的Converter
 */
public class GuiguMessageConverter implements HttpMessageConverter<Person> {

    @Override
    public boolean canRead(Class<?> clazz, MediaType mediaType) {
        return false;
    }

    @Override
    public boolean canWrite(Class<?> clazz, MediaType mediaType) {
        return clazz.isAssignableFrom(Person.class);
    }

    /**
     * 服务器要统计所有MessageConverter都能写出哪些内容类型
     *
     * application/x-guigu
     * @return
     */
    @Override
    public List<MediaType> getSupportedMediaTypes() {
        return MediaType.parseMediaTypes("application/x-guigu");
    }

    @Override
    public Person read(Class<? extends Person> clazz, HttpInputMessage inputMessage)
throws IOException, HttpMessageNotReadableException {
        return null;
    }

    @Override
    public void write(Person person, MediaType contentType, HttpOutputMessage
outputMessage) throws IOException, HttpMessageNotWritableException {
        //自定义协议数据的写出
        String data = person.getUserName()+";"+person.getAge()+";"+person.getBirth();


        //写出去
        OutputStream body = outputMessage.getBody();
        body.write(data.getBytes());
    }
}
```

```java
import java.util.Date;

@Controller
public class ResponseTestController {

    /**
     * 1、浏览器发请求直接返回 xml    [application/xml]        jacksonXmlConverter
     * 2、如果是ajax请求 返回 json   [application/json]      jacksonJsonConverter
     * 3、如果硅谷app发请求，返回自定义协议数据  [appliaction/x-guigu]   xxxxConverter
     *          属性值1;属性值2;
     *
     * 步骤：
     * 1、添加自定义的MessageConverter进系统底层
     * 2、系统底层就会统计出所有MessageConverter能操作哪些类型
     * 3、客户端内容协商 [guigu--->guigu]
     *
     * 作业：如何以参数的方式进行内容协商
     * @return
     */
    @ResponseBody  //利用返回值处理器里面的消息转换器进行处理
    @GetMapping(value = "/test/person")
    public Person getPerson(){
        Person person = new Person();
        person.setAge(28);
        person.setBirth(new Date());
        person.setUserName("zhangsan");
        return person;
    }

}
```

用Postman发送`/test/person`（请求头`Accept:application/x-guigu`），将返回自定义协议数据的写出。

42、响应处理-【源码分析】-浏览器与PostMan内容协商完全适配

假设你想基于自定义请求参数的自定义内容协商功能。

- SpringBoot中默认的基于参数的内容协商方式只有两种，分别是xml和json。

换句话，在地址栏输入`http://localhost:8080/test/person?format=gg`返回数据，跟`http://localhost:8080/test/person`且请求头参数`Accept:application/x-guigu`的返回自定义协议数据的一致。

```java
@Configuration(proxyBeanMethods = false)
public class WebConfig /*implements WebMvcConfigurer*/ {
    //1、WebMvcConfigurer定制化SpringMVC的功能
    @Bean
    public WebMvcConfigurer webMvcConfigurer(){
        return new WebMvcConfigurer() {
            /**
             * 自定义内容协商策略
             * @param configurer
             */
            @Override
            public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
                //Map<String, MediaType> mediaTypes
                Map<String, MediaType> mediaTypes = new HashMap<>();
                mediaTypes.put("json",MediaType.APPLICATION_JSON);
                mediaTypes.put("xml",MediaType.APPLICATION_XML);
                //自定义媒体类型
                mediaTypes.put("gg",MediaType.parseMediaType("application/x-guigu"));
                //指定支持解析哪些参数对应的哪些媒体类型，只配置上面的会导致请求头的配置策略失效
                ParameterContentNegotiationStrategy parameterStrategy = new ParameterContentNegotiationStrategy(mediaTypes);
                //还需添加请求头处理策略，否则accept:application/json、application/xml则会失效，自定义的处理器只具有配置过的处理策略
                HeaderContentNegotiationStrategy headeStrategy = new HeaderContentNegotiationStrategy();
               // 将请求头的处理策略也添加进入
                configurer.strategies(Arrays.asList(parameterStrategy, headeStrategy));
            }
        }
    }

    ...

}
```

日后开发要注意，**有可能我们添加的自定义的功能会覆盖默认很多功能，导致一些默认的功能失效。**

43、视图解析-Thymeleaf初体验

> **Thymeleaf** is a modern server-side Java template engine for both web and standalone environments.
>
> Thymeleaf's main goal is to bring elegant *natural templates* to your development workflow — HTML that can be correctly displayed in browsers and also work as static prototypes, allowing for stronger collaboration in development teams.
>
> With modules for Spring Framework, a host of integrations with your favourite tools, and the ability to plug in your own functionality, Thymeleaf is ideal for modern-day HTML5 JVM web development — although there is much more it can do.——Link

Thymeleaf官方文档

thymeleaf使用

引入Starter

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

自动配置好了thymeleaf

```java
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(ThymeleafProperties.class)
@ConditionalOnClass({ TemplateMode.class, SpringTemplateEngine.class })
@AutoConfigureAfter({ WebMvcAutoConfiguration.class, WebFluxAutoConfiguration.class })
public class ThymeleafAutoConfiguration {
    ...
}
```

自动配好的策略

1. 所有thymeleaf的配置值都在 ThymeleafProperties
2. 配置好了 **SpringTemplateEngine**
3. 配好了 **ThymeleafViewResolver**
4. 我们只需要直接开发页面

```java
public static final String DEFAULT_PREFIX = "classpath:/templates/";  //模板放置处
public static final String DEFAULT_SUFFIX = ".html"; //文件的后缀名
```

编写一个控制层：

```java
@Controller
public class ViewTestController {
    @GetMapping("/hello")
    public String hello(Model model){
        //model中的数据会被放在请求域中 request.setAttribute("a",aa)
        model.addAttribute("msg","一定要大力发展工业文化");
        model.addAttribute("link","http://www.baidu.com");
        return "success";
    }
}
```

`/templates/success.html`：

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1 th:text="${msg}">nice</h1>
<h2>
    <a href="www.baidu.com" th:href="${link}">去百度</a>  <br/>
    <a href="www.google.com" th:href="@{/link}">去百度</a>
</h2>
</body>
</html>
```

```yaml
server:
  servlet:
    context-path: /app #设置应用名
```

这个设置后，URL要插入`/app`，如`http://localhost:8080/app/hello.html`。

基本语法

表达式

表达式名字	语法	用途
变量取值	${...}	获取请求域、session域、对象等值
选择变量	*{...}	获取上下文对象值
消息	#{...}	获取国际化等值
链接	@{...}	生成链接
片段表达式	~{...}	jsp:include 作用，引入公共页面片段

字面量

- 文本值: **'one text' , 'Another one!' ,…**
- 数字: **0 , 34 , 3.0 , 12.3 ,…**
- 布尔值: **true , false**
- 空值: **null**
- 变量： one，two，.... 变量不能有空格

文本操作

- 字符串拼接: **+**
- 变量替换: **|The name is ${name}|**

数学运算

- 运算符: + , - , * , / , %

布尔运算

- 运算符: **and , or**
- 一元运算: **! , not**

比较运算

- 比较: **> , < , >= , <= (gt , lt , ge , le)**
- 等式: **== , != (eq , ne)**

条件运算

- If-then: **(if) ? (then)**
- If-then-else: **(if) ? (then) : (else)**
- Default: (value) **?: (defaultvalue)**

特殊操作

- 无操作: _

设置属性值-th:attr

- 设置单个值

```
<form action="subscribe.html" th:attr="action=@{/subscribe}">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe!" th:attr="value=#{subscribe.submit}"/>
  </fieldset>
</form>
```

- 设置多个值

```
<img src="../../images/gtvglogo.png"
     th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />
```

[官方文档 - 5 Setting Attribute Values]

迭代

```
<tr th:each="prod : ${prods}">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
</tr>
```

```
<tr th:each="prod,iterStat : ${prods}" th:class="${iterStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
</tr>
```

条件运算

```
<a href="comments.html"
    th:href="@{/product/comments(prodId=${prod.id})}"
    th:if="${not #lists.isEmpty(prod.comments)}">view</a>
```

```
<div th:switch="${user.role}">
      <p th:case="'admin'">User is an administrator</p>
      <p th:case="#{roles.manager}">User is a manager</p>
      <p th:case="*">User is some other thing</p>
</div>
```

属性优先级

Order	Feature	Attributes
1	Fragment inclusion	`th:insert` `th:replace`
2	Fragment iteration	`th:each`
3	Conditional evaluation	`th:if` `th:unless` `th:switch` `th:case`
4	Local variable definition	`th:object` `th:with`
5	General attribute modification	`th:attr` `th:attrprepend` `th:attrappend`
6	Specific attribute modification	`th:value` `th:href` `th:src` `...`
7	Text (tag body modification)	`th:text` `th:utext`
8	Fragment specification	`th:fragment`
9	Fragment removal	`th:remove`

官方文档 - 10 Attribute Precedence

44、web实验-后台管理系统基本功能

项目创建

使用IDEA的Spring Initializr。

- thymeleaf、
- web-starter、
- devtools、
- lombok

登陆页面

- `/static` 放置 css，js等静态资源
- `/templates/login.html` 登录页

```
<html lang="en" xmlns:th="http://www.thymeleaf.org"><!-- 要加这玩意thymeleaf才能用 -->

<form class="form-signin" action="index.html" method="post" th:action="@{/login}">
```

```

    ...

    <!-- 消息提醒 -->
    <label style="color: red" th:text="${msg}"></label>

    <input type="text" name="userName" class="form-control" placeholder="User ID" autofocus>
    <input type="password" name="password" class="form-control" placeholder="Password">

    <button class="btn btn-lg btn-login btn-block" type="submit">
        <i class="fa fa-check"></i>
    </button>

    ...

</form>
```

- `/templates/main.html` 主页

thymeleaf内联写法:

```
<p>Hello, [[${session.user.name}]]!</p>
```

登录控制层

```
@Controller
public class IndexController {
    /**
     * 来登录页
     * @return
     */
    @GetMapping(value = {"/","/login"})
    public String loginPage(){

        return "login";
    }

    @PostMapping("/login")
    public String main(User user, HttpSession session, Model model){ //RedirectAttributes

        if(StringUtils.hasLength(user.getUserName()) && "123456".equals(user.getPassword())){
            //把登陆成功的用户保存起来
            session.setAttribute("loginUser",user);
            //登录成功重定向到main.html;  重定向防止表单重复提交
            return "redirect:/main.html";
        }else {
            model.addAttribute("msg","账号密码错误");
            //回到登录页面
            return "login";
        }
    }

     /**
     * 去main页面
     * @return
     */
```

```java
    @GetMapping("/main.html")
    public String mainPage(HttpSession session, Model model){

        //最好用拦截器,过滤器
        Object loginUser = session.getAttribute("loginUser");
        if(loginUser != null){
            return "main";
        }else {
            //session过期，没有登陆过
            //回到登录页面
            model.addAttribute("msg","请重新登录");
            return "login";
        }
    }

}
```

模型

```java
@AllArgsConstructor
@NoArgsConstructor
@Data
public class User {
    private String userName;
    private String password;
}
```

SpringBoot【二】

45、web实验-抽取公共页面

[官方文档 - Template Layout]

- 公共页面 `/templates/common.html`

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"><!--注意要添加xmlns:th才能添加thymeleaf的标签-->
<head th:fragment="commonheader">
    <!--common-->
    <link href="css/style.css" th:href="@{/css/style.css}" rel="stylesheet">
    <link href="css/style-responsive.css" th:href="@{/css/style-responsive.css}" rel="stylesheet">
    ...
</head>
<body>
<!-- left side start-->
<div id="leftmenu" class="left-side sticky-left-side">
    ...

    <div class="left-side-inner">
        ...

        <!--sidebar nav start-->
        <ul class="nav nav-pills nav-stacked custom-nav">
            <li><a th:href="@{/main.html}"><i class="fa fa-home"></i> <span>Dashboard</span></a></li>
            ...
            <li class="menu-list nav-active"><a href="#"><i class="fa fa-th-list"></i> <span>Data Tables</span></a>
                <ul class="sub-menu-list">
                    <li><a th:href="@{/basic_table}"> Basic Table</a></li>
                    <li><a th:href="@{/dynamic_table}"> Advanced Table</a></li>
                    <li><a th:href="@{/responsive_table}"> Responsive Table</a></li>
                    <li><a th:href="@{/editable_table}"> Edit Table</a></li>
                </ul>
            </li>
            ...
        </ul>
        <!--sidebar nav end-->
    </div>
</div>
<!-- left side end-->


<!-- header section start-->
<div th:fragment="headermenu" class="header-section">

    <!--toggle button start-->
    <a class="toggle-btn"><i class="fa fa-bars"></i></a>
    <!--toggle button end-->
    ...

```

```html
</div>
<!-- header section end-->

<div id="commonscript">
    <!-- Placed js at the end of the document so the pages load faster -->
    <script th:src="@{/js/jquery-1.10.2.min.js}"></script>
    <script th:src="@{/js/jquery-ui-1.9.2.custom.min.js}"></script>
    <script th:src="@{/js/jquery-migrate-1.2.1.min.js}"></script>
    <script th:src="@{/js/bootstrap.min.js}"></script>
    <script th:src="@{/js/modernizr.min.js}"></script>
    <script th:src="@{/js/jquery.nicescroll.js}"></script>
    <!--common scripts for all pages-->
    <script th:src="@{/js/scripts.js}"></script>
</div>
</body>
</html>
```

- `/templates/table/basic_table.html`

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0">
  <meta name="description" content="">
  <meta name="author" content="ThemeBucket">
  <link rel="shortcut icon" href="#" type="image/png">

  <title>Basic Table</title>
    <div th:include="common :: commonheader"> </div><!--将common.html的代码段 插进来-->
</head>

<body class="sticky-header">

<section>
<div th:replace="common :: #leftmenu"></div>

    <!-- main content start-->
    <div class="main-content" >

        <div th:replace="common :: headermenu"></div>
        ...
    </div>
    <!-- main content end-->
</section>

<!-- Placed js at the end of the document so the pages load faster -->
<div th:replace="common :: #commonscript"></div>


</body>
</html>

```

Difference between `th:insert` and `th:replace` (and `th:include`)

46、web实验-遍历数据与页面bug修改

控制层代码：

```java
@GetMapping("/dynamic_table")
public String dynamic_table(Model model){
    //表格内容的遍历
    List<User> users = Arrays.asList(new User("zhangsan", "123456"),
                                     new User("lisi", "123444"),
                                     new User("haha", "aaaaa"),
                                     new User("hehe ", "aaddd"));
    model.addAttribute("users",users);

    return "table/dynamic_table";
}
```

页面代码：

```html
<table class="display table table-bordered" id="hidden-table-info">
    <thead>
        <tr>
            <th>#</th>
            <th>用户名</th>
            <th>密码</th>
        </tr>
    </thead>
    <tbody>
        <tr class="gradeX" th:each="user,stats:${users}">
            <td th:text="${stats.count}">Trident</td>
            <td th:text="${user.userName}">Internet</td>
            <td >[[${user.password}]]</td>
        </tr>
    </tbody>
</table>
```

47、视图解析-【源码分析】-视图解析器与视图

视图解析原理流程：

1. 目标方法处理的过程中（阅读`DispatcherServlet`源码），所有数据都会被放在 `ModelAndViewContainer` 里面，其中包括数据和视图地址。
2. 方法的参数是一个自定义类型对象（从请求参数中确定的），把它重新放在 `ModelAndViewContainer` 。
3. 任何目标方法执行完成以后都会返回`ModelAndView`（数据和视图地址）。
4. `processDispatchResult()`方法处理派发结果（页面改如何响应）
    - `render(mv, request, response);` 进行页面渲染逻辑
        - 根据方法的`String`返回值得到 `View` 对象【定义了页面的渲染逻辑】
        1. 所有的视图解析器尝试是否能根据当前返回值得到`View`对象

```
// 五个视图解析器
ContentNegotiatingViewResolver
BeanNameViewResolver
ThymeleafViewResolver
ViewResolverComposite
InternalResourceViewResolver
```

2. 得到了`r edirect:/main.html --> Thymeleaf new RedirectView()`。
3. `ContentNegotiationViewResolver` 里面包含了下面所有的视图解析器，内部还是利用下面所有视图解析器得到视图对象。
4. `view.render(mv.getModelInternal(), request, response);` 视图对象调用自定义的render进行页面渲染工作。
    - `RedirectView` 如何渲染【重定向到一个页面】
        - 获取目标url地址
        - `response.sendRedirect(encodedURL);`

视图解析：根据前缀内容找到合适的视图解析器

- 返回值以 `forward:` 开始： `new InternalResourceView(forwardUrl);` ——转发 `request.getRequestDispatcher(path).forward(request, response);`
- 返回值以 `redirect:` 开始： `new RedirectView()` ——render就是重定向
- 返回值是普通字符串： `new ThymeleafView()`——自己创建一个视图，并进行渲染

48、拦截器-登录检查与静态资源放行

> preHandler：在处理请求之前需要做的逻辑
>
> postHandler：在返回视图页面之前需要做的逻辑
>
> afterCompletion：整个请求完成后（页面渲染完成后），需要执行的逻辑

1. 编写一个拦截器实现`HandlerInterceptor`接口
2. 拦截器注册到容器中（实现`WebMvcConfigurer`的`addInterceptors()`）
3. 指定拦截规则（注意，如果是拦截所有，静态资源也会被拦截）【`excludePathPatterns`放行静态资源】

编写一个实现`HandlerInterceptor`接口的拦截器：

```java
@Slf4j
public class LoginInterceptor implements HandlerInterceptor {

    /**
     * 目标方法执行之前
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

        String requestURI = request.getRequestURI();
        log.info("preHandle拦截的请求路径是{}",requestURI);

        //登录检查逻辑
        HttpSession session = request.getSession();

        Object loginUser = session.getAttribute("loginUser");

        if(loginUser != null){
            //放行
            return true;
        }

        //拦截住。未登录。跳转到登录页
        request.setAttribute("msg","请先登录");
//        re.sendRedirect("/");
        request.getRequestDispatcher("/").forward(request,response);
        return false;
    }

    /**
     * 目标方法执行完成以后
     */
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        log.info("postHandle执行{}",modelAndView);
    }

    /**
     * 页面渲染以后
     */
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        log.info("afterCompletion执行异常{}",ex);
    }
}
```

拦截器注册到容器中 && 指定拦截规则：

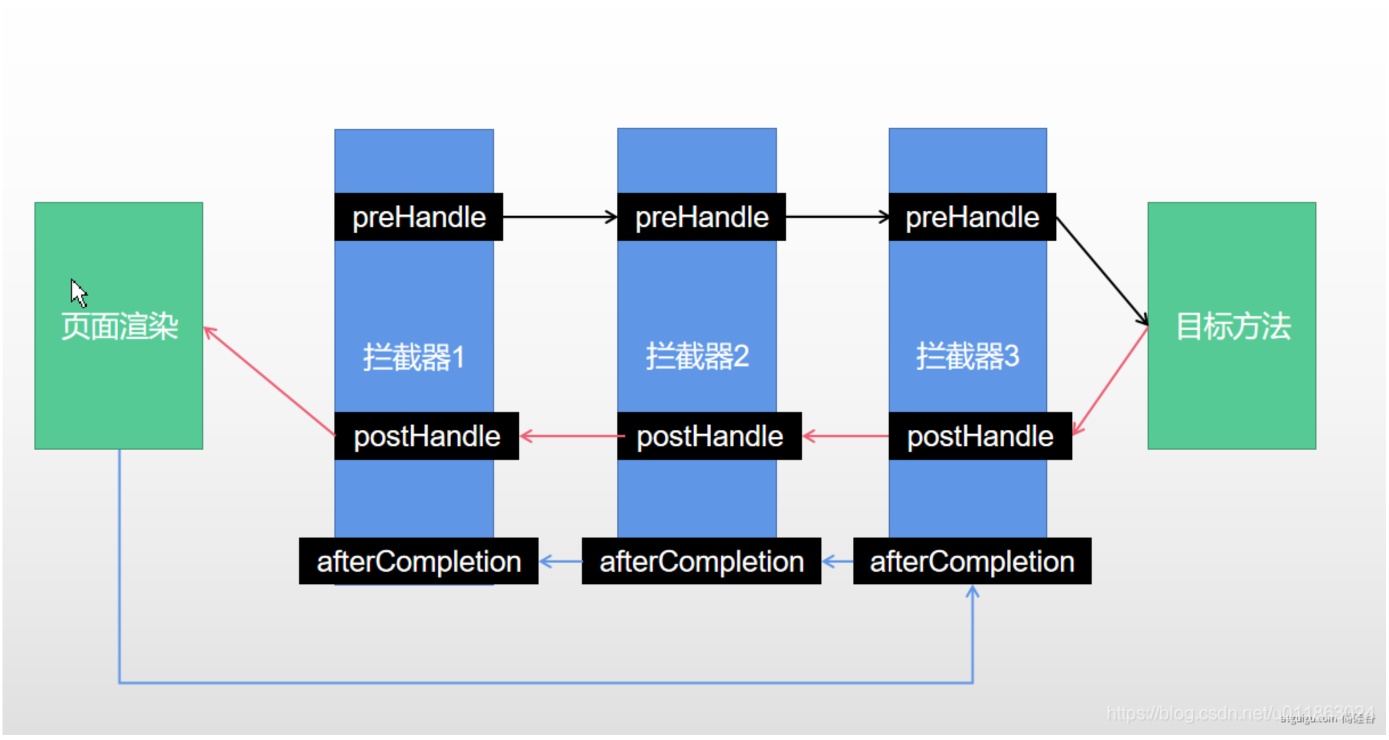
```java
@Configuration
public class AdminWebConfig implements WebMvcConfigurer{
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoginInterceptor())//拦截器注册到容器中
                .addPathPatterns("/**")  //所有请求都被拦截包括静态资源
                .excludePathPatterns("/","/login","/css/**","/fonts/**","/images/**",
                        "/js/**","/aa/**"); //放行的请求
}
```

49、拦截器-【源码分析】-拦截器的执行时机和原理

1. 根据当前请求，找到`HandlerExecutionChain`（可以处理请求的handler以及handler的所有拦截器）
2. 先来顺序执行所有拦截器的 `preHandle()`方法。

   - 如果当前拦截器`preHandle()`返回为`true`。则执行下一个拦截器的`preHandle()`
   - 如果当前拦截器返回为`false`。直接倒序执行所有已经执行了的拦截器的 `afterCompletion();`。
3. 如果任何一个拦截器返回`false`，直接跳出不执行目标方法。
4. 所有拦截器都返回`true`，才执行目标方法。
5. 倒序执行所有拦截器的`postHandle()`方法。
6. 前面的步骤有任何异常都会直接倒序触发 `afterCompletion()`。
7. 页面成功渲染完成以后，也会倒序触发 `afterCompletion()`。

小结：正序preHandle，逆序postHandle和afterCompletion



`DispatcherServlet`中涉及到`HandlerInterceptor`的地方：

```java
public class DispatcherServlet extends FrameworkServlet {

    ...

    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) 
throws Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        boolean multipartRequestParsed = false;

        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

        try {
            ModelAndView mv = null;
            Exception dispatchException = null;

                ...

                //该方法内调用HandlerInterceptor的preHandle()
                if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                    return;
                }

                // Actually invoke the handler.
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

                ...
                //该方法内调用HandlerInterceptor的postHandle()
                mappedHandler.applyPostHandle(processedRequest, response, mv);
            }
            processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
        }
        catch (Exception ex) {
            //该方法内调用HandlerInterceptor接口的afterCompletion方法
            triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
        }
        catch (Throwable err) {
            //该方法内调用HandlerInterceptor接口的afterCompletion方法
            triggerAfterCompletion(processedRequest, response, mappedHandler,
                    new NestedServletException("Handler processing failed", err));
        }
        finally {
            ...
        }
    }

    private void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse
response,
            @Nullable HandlerExecutionChain mappedHandler, Exception ex) throws Exception
{

        if (mappedHandler != null) {
            //该方法内调用HandlerInterceptor接口的afterCompletion方法
            mappedHandler.triggerAfterCompletion(request, response, ex);
        }
        throw ex;
    }

    private void processDispatchResult(HttpServletRequest request, HttpServletResponse
response,
            @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,
            @Nullable Exception exception) throws Exception {

        ...

        if (mappedHandler != null) {
            //该方法内调用HandlerInterceptor接口的afterCompletion方法
            // Exception (if any) is already handled..
            mappedHandler.triggerAfterCompletion(request, response, null);
        }
    }


}
```

```java
public class HandlerExecutionChain {

    ...

    boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response)
throws Exception {
        for (int i = 0; i < this.interceptorList.size(); i++) {
            HandlerInterceptor interceptor = this.interceptorList.get(i);
            //HandlerInterceptor的preHandle方法
            if (!interceptor.preHandle(request, response, this.handler)) {

                triggerAfterCompletion(request, response, null);
                return false;
            }
            this.interceptorIndex = i;
        }
        return true;
    }

    void applyPostHandle(HttpServletRequest request, HttpServletResponse response, @Nullable ModelAndView mv)
            throws Exception {

        for (int i = this.interceptorList.size() - 1; i >= 0; i--) {
            HandlerInterceptor interceptor = this.interceptorList.get(i);

            //HandlerInterceptor接口的postHandle方法
            interceptor.postHandle(request, response, this.handler, mv);
        }
    }

    void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, @Nullable Exception ex) {
        for (int i = this.interceptorIndex; i >= 0; i--) {
            HandlerInterceptor interceptor = this.interceptorList.get(i);
            try {
                //HandlerInterceptor接口的afterCompletion方法
                interceptor.afterCompletion(request, response, this.handler, ex);
            }
            catch (Throwable ex2) {
                logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);
            }
        }
    }


}

```

50、文件上传-单文件与多文件上传的使用

- 页面代码 `/static/form/form_layouts.html`

```html
<form role="form" th:action="@{/upload}" method="post" enctype="multipart/form-data">
    <div class="form-group">
        <label for="exampleInputEmail1">邮箱</label>
        <input type="email" name="email" class="form-control" id="exampleInputEmail1" placeholder="Enter email">
    </div>

    <div class="form-group">
        <label for="exampleInputPassword1">名字</label>
        <input type="text" name="username" class="form-control" id="exampleInputPassword1" placeholder="Password">
    </div>

    <div class="form-group">
        <label for="exampleInputFile">头像</label>
        <input type="file" name="headerImg" id="exampleInputFile">
    </div>

    <div class="form-group">
        <label for="exampleInputFile">生活照</label>
        <input type="file" name="photos" multiple>
    </div>

    <div class="checkbox">
        <label>
            <input type="checkbox"> Check me out
        </label>
    </div>
    <button type="submit" class="btn btn-primary">提交</button>
</form>
```

- 控制层代码

```java
@Slf4j
@Controller
public class FormTestController {

    @GetMapping("/form_layouts")
    public String form_layouts(){
        return "form/form_layouts";
    }

    @PostMapping("/upload")
    public String upload(@RequestParam("email") String email,
                         @RequestParam("username") String username,
                         @RequestPart("headerImg") MultipartFile headerImg,
                         @RequestPart("photos") MultipartFile[] photos) throws IOException {

        log.info("上传的信息：email={}，username={}，headerImg={}，photos={}",
                 email,username,headerImg.getSize(),photos.length);

        if(!headerImg.isEmpty()){
            //保存到文件服务器，OSS服务器
            String originalFilename = headerImg.getOriginalFilename();
            headerImg.transferTo(new File("H:\\cache\\"+originalFilename));
        }

        if(photos.length > 0){
            for (MultipartFile photo : photos) {
                if(!photo.isEmpty()){
                    String originalFilename = photo.getOriginalFilename();
                    photo.transferTo(new File("H:\\cache\\"+originalFilename));
                }
            }
        }


        return "main";
    }
}
```

文件上传相关的配置类：

- `org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration`
- `org.springframework.boot.autoconfigure.web.servlet.MultipartProperties`

文件大小相关配置项：

```properties
# 单个文件最大大小10MB
spring.servlet.multipart.max-file-size=10MB
# 整个请求内的文件最大大小不能超过100MB
spring.servlet.multipart.max-request-size=100MB
```

51、文件上传-【源码流程】文件上传参数解析器

文件上传相关的自动配置类`MultipartAutoConfiguration`有创建文件上传参数解析器`StandardServletMultipartResolver`。

```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ Servlet.class, StandardServletMultipartResolver.class, MultipartConfigElement.class })
@ConditionalOnProperty(prefix = "spring.servlet.multipart", name = "enabled", matchIfMissing = true)
@ConditionalOnWebApplication(type = Type.SERVLET)
@EnableConfigurationProperties(MultipartProperties.class)
public class MultipartAutoConfiguration {

    private final MultipartProperties multipartProperties;

    public MultipartAutoConfiguration(MultipartProperties multipartProperties) {
        this.multipartProperties = multipartProperties;
    }

    @Bean
    @ConditionalOnMissingBean({ MultipartConfigElement.class, CommonsMultipartResolver.class })
    public MultipartConfigElement multipartConfigElement() {
        return this.multipartProperties.createMultipartConfig();
    }

    @Bean(name = DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)
    @ConditionalOnMissingBean(MultipartResolver.class)
    public StandardServletMultipartResolver multipartResolver() {
        //配置好文件上传解析器
        StandardServletMultipartResolver multipartResolver = new StandardServletMultipartResolver();
        multipartResolver.setResolveLazily(this.multipartProperties.isResolveLazily());
        return multipartResolver;
    }

}
```

```java
//文件上传解析器
public class StandardServletMultipartResolver implements MultipartResolver {

    private boolean resolveLazily = false;

    public void setResolveLazily(boolean resolveLazily) {
        this.resolveLazily = resolveLazily;
    }


    @Override
    public boolean isMultipart(HttpServletRequest request) {
        return StringUtils.startsWithIgnoreCase(request.getContentType(), "multipart/");
    }

    @Override
    public MultipartHttpServletRequest resolveMultipart(HttpServletRequest request) throws MultipartException {
        return new StandardMultipartHttpServletRequest(request, this.resolveLazily);
    }

    @Override
    public void cleanupMultipart(MultipartHttpServletRequest request) {
        if (!(request instanceof AbstractMultipartHttpServletRequest) ||
                ((AbstractMultipartHttpServletRequest) request).isResolved()) {
            // To be on the safe side: explicitly delete the parts,
            // but only actual file parts (for Resin compatibility)
            try {
                for (Part part : request.getParts()) {
                    if (request.getFile(part.getName()) != null) {
                        part.delete();
                    }
                }
            }
            catch (Throwable ex) {
                LogFactory.getLog(getClass()).warn("Failed to perform cleanup of multipart items", ex);
            }
        }
    }

}
```

```java
public class DispatcherServlet extends FrameworkServlet {

    @Nullable
    private MultipartResolver multipartResolver;

    private void initMultipartResolver(ApplicationContext context) {
        ...

        //这个就是配置类配置的StandardServletMultipartResolver文件上传解析器
        this.multipartResolver = context.getBean(MULTIPART_RESOLVER_BEAN_NAME, MultipartResolver.class);
        ...
    }

    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        boolean multipartRequestParsed = false;//最后finally的回收flag
        ...
        try {
            ModelAndView mv = null;
            Exception dispatchException = null;

            try {
                //做预处理,如果有上传文件 就new StandardMultipartHttpServletRequest包装类
                processedRequest = checkMultipart(request);
                multipartRequestParsed = (processedRequest != request);
                // Determine handler for the current request.
                mappedHandler = getHandler(processedRequest);

                ...

                // Determine handler adapter for the current request.
                HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

                ...

                // Actually invoke the handler.
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

            }
            ....

        finally {

            ...

            if (multipartRequestParsed) {
                cleanupMultipart(processedRequest);
            }
        }
    }

    protected HttpServletRequest checkMultipart(HttpServletRequest request) throws MultipartException {
        if (this.multipartResolver != null && this.multipartResolver.isMultipart(request)) {
            ...
            return this.multipartResolver.resolveMultipart(request);
            ...
        }
    }

    protected void cleanupMultipart(HttpServletRequest request) {
        if (this.multipartResolver != null) {
            MultipartHttpServletRequest multipartRequest =
                    WebUtils.getNativeRequest(request, MultipartHttpServletRequest.class);
            if (multipartRequest != null) {
                this.multipartResolver.cleanupMultipart(multipartRequest);
            }
        }
    }
}
```

`mv = ha.handle(processedRequest, response, mappedHandler.getHandler());`跳到以下的类

```java
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
        implements BeanFactoryAware, InitializingBean {
    @Override
    protected ModelAndView handleInternal(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
        ModelAndView mav;
        ...
        mav = invokeHandlerMethod(request, response, handlerMethod);
        ...
        return mav;
    }

    @Nullable
    protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

        ServletWebRequest webRequest = new ServletWebRequest(request, response);
        try {
            WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);
            ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);

            ServletInvocableHandlerMethod invocableMethod =
createInvocableHandlerMethod(handlerMethod);
            if (this.argumentResolvers != null) {//关注点
                invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
            }
            ...
            invocableMethod.invokeAndHandle(webRequest, mavContainer);
            ...

            return getModelAndView(mavContainer, modelFactory, webRequest);
        }
        finally {
            webRequest.requestCompleted();
        }
    }

}
```

`this.argumentResolvers`其中主角类`RequestPartMethodArgumentResolver`用来生成

```java
public class ServletInvocableHandlerMethod extends InvocableHandlerMethod {

    ...
    public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer
mavContainer,
            Object... providedArgs) throws Exception {
        Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);
        ...
    }

    @Nullable
    public Object invokeForRequest(NativeWebRequest request, @Nullable
ModelAndViewContainer mavContainer,
            Object... providedArgs) throws Exception {

        Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
        ...
        return doInvoke(args);//反射调用
    }

    @Nullable
    protected Object doInvoke(Object... args) throws Exception {
        Method method = getBridgedMethod();
        ReflectionUtils.makeAccessible(method);
        return method.invoke(getBean(), args);
        ...
    }

    //处理得出multipart参数，准备稍后的反射调用（@PostMapping标记的上传方法）
    protected Object[] getMethodArgumentValues(NativeWebRequest request, @Nullable
ModelAndViewContainer mavContainer,
            Object... providedArgs) throws Exception {

        MethodParameter[] parameters = getMethodParameters();
        ...
        Object[] args = new Object[parameters.length];
        for (int i = 0; i < parameters.length; i++) {
            MethodParameter parameter = parameters[i];
            parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);
            args[i] = findProvidedArgument(parameter, providedArgs);
            if (args[i] != null) {
                continue;
            }
            //关注点1
            if (!this.resolvers.supportsParameter(parameter)) {
                throw new IllegalStateException(formatArgumentError(parameter, "No
suitable resolver"));
            }
            try {
                //关注点2
                args[i] = this.resolvers.resolveArgument(parameter, mavContainer, request,
this.dataBinderFactory);
            }
            catch (Exception ex) {
                ...
            }
        }
        return args;
    }

}
```

```java
public class RequestPartMethodArgumentResolver extends
AbstractMessageConverterMethodArgumentResolver {

    //对应上面代码关注点1
    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        //标注@RequestPart的参数
        if (parameter.hasParameterAnnotation(RequestPart.class)) {
            return true;
        }
        else {
            if (parameter.hasParameterAnnotation(RequestParam.class)) {
                return false;
            }
            return
MultipartResolutionDelegate.isMultipartArgument(parameter.nestedIfOptional());
        }
    }

    //对应上面代码关注点2
    @Override
    @Nullable
    public Object resolveArgument(MethodParameter parameter, @Nullable
ModelAndViewContainer mavContainer,
            NativeWebRequest request, @Nullable WebDataBinderFactory binderFactory) throws
Exception {

        HttpServletRequest servletRequest =
request.getNativeRequest(HttpServletRequest.class);
        Assert.state(servletRequest != null, "No HttpServletRequest");

        RequestPart requestPart = parameter.getParameterAnnotation(RequestPart.class);
        boolean isRequired = ((requestPart == null || requestPart.required()) &&
!parameter.isOptional());

        String name = getPartName(parameter, requestPart);
        parameter = parameter.nestedIfOptional();
        Object arg = null;

        //封装成MultipartFile类型的对象作参数
        Object mpArg = MultipartResolutionDelegate.resolveMultipartArgument(name,
parameter, servletRequest);
        if (mpArg != MultipartResolutionDelegate.UNRESOLVABLE) {
            arg = mpArg;
        }

        ...

        return adaptArgumentIfNecessary(arg, parameter);
    }
}
```

```java
public final class MultipartResolutionDelegate {
    ...

    @Nullable
    public static Object resolveMultipartArgument(String name, MethodParameter parameter,
HttpServletRequest request)
            throws Exception {
```

```java

        MultipartHttpServletRequest multipartRequest =
                WebUtils.getNativeRequest(request, MultipartHttpServletRequest.class);
        boolean isMultipart = (multipartRequest != null || isMultipartContent(request));

        if (MultipartFile.class == parameter.getNestedParameterType()) {
            if (!isMultipart) {
                return null;
            }
            if (multipartRequest == null) {
                multipartRequest = new StandardMultipartHttpServletRequest(request);
            }
            return multipartRequest.getFile(name);
        }
        else if (isMultipartFileCollection(parameter)) {
            if (!isMultipart) {
                return null;
            }
            if (multipartRequest == null) {
                multipartRequest = new StandardMultipartHttpServletRequest(request);
            }
            List<MultipartFile> files = multipartRequest.getFiles(name);
            return (!files.isEmpty() ? files : null);
        }
        else if (isMultipartFileArray(parameter)) {
            if (!isMultipart) {
                return null;
            }
            if (multipartRequest == null) {
                multipartRequest = new StandardMultipartHttpServletRequest(request);
            }
            List<MultipartFile> files = multipartRequest.getFiles(name);
            return (!files.isEmpty() ? files.toArray(new MultipartFile[0]) : null);
        }
        else if (Part.class == parameter.getNestedParameterType()) {
            if (!isMultipart) {
                return null;
            }
            return request.getPart(name);
        }
        else if (isPartCollection(parameter)) {
            if (!isMultipart) {
                return null;
            }
            List<Part> parts = resolvePartList(request, name);
            return (!parts.isEmpty() ? parts : null);
        }
        else if (isPartArray(parameter)) {
            if (!isMultipart) {
                return null;
            }
            List<Part> parts = resolvePartList(request, name);
            return (!parts.isEmpty() ? parts.toArray(new Part[0]) : null);
        }
        else {
            return UNRESOLVABLE;
        }
    }

    ...
```

```

}
```

52、错误处理-SpringBoot默认错误处理机制

[Spring Boot官方文档 - Error Handling]

默认规则：

- 默认情况下，Spring Boot提供`/error`处理所有错误的映射
- 机器客户端，它将生成JSON响应，其中包含错误，HTTP状态和异常消息的详细信息。对于浏览器客户端，响应一个" whitelabel"错误视图，以HTML格式呈现相同的数据

```json
{
  "timestamp": "2020-11-22T05:53:28.416+00:00",
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/asadada"
}
```

- 要对其进行自定义，添加`View`解析为`error`
- 要完全替换默认行为，可以实现 `ErrorController`并注册该类型的Bean定义，或添加`ErrorAttributes类型的组件`以使用现有机制但替换其内容。
- `/templates/error/`下的4xx，5xx页面会被自动解析

  ```
  |----templates
      |----error
          |----404.html
          |----5xx.html // 代表所有以5开头的都显示这个页面
  ```

> 定制错误处理逻辑：
>
> - 自定义错误页
> - @ControllerAdvice + @ExceptionHandler处理异常
> - 实现HandlerExceptionResolver处理异常

53、错误处理-【源码分析】底层组件功能分析

- `ErrorMvcAutoConfiguration`自动配置异常处理规则
- **容器中的组件**：类型：`DefaultErrorAttributes` -> id：`errorAttributes`
- `public class DefaultErrorAttributes implements ErrorAttributes, HandlerExceptionResolver`
  - `DefaultErrorAttributes`：定义错误页面中可以包含数据（异常明细，堆栈信息等）。
- **容器中的组件**：类型：`BasicErrorController` --> id：`basicErrorController`（json+白页 适配响应）
- **处理默认 `/error` 路径的请求**，页面响应 `new ModelAndView("error", model);`

    - 容器中有组件 `View`->id是error；（响应默认错误页）
    - 容器中放组件 `BeanNameViewResolver`（视图解析器）；按照返回的视图名作为组件的id去容器中找`View`对象。
- **容器中的组件**：类型：`DefaultErrorViewResolver` -> id：`conventionErrorViewResolver`
- **如果发生异常错误，会以HTTP的状态码 作为视图页地址（viewName），找到真正的页面**（主要作用）。
    - error/404、5xx.html
    - 如果想要返回页面，就会找error视图（`StaticView`默认是一个白页）。

54、错误处理-【源码流程】异常处理流程

> 执行目标方法，目标方法运行期间会有任何异常都会被catch，而且标志当前请求结束；并且产生dispatchException
>
> 进入视图解析流程：`processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException)`
>
> `mv=processHandlerException`；处理Handler发生的异常，处理完成后返回ModelAndView
>
> > 遍历所有的handlerExceptionResolvers，看谁能处理当前异常
> >
> > 默认的异常处理器DefaultErrorAttributes先来处理异常，把异常信息保存到request并返回null
> >
> > 默认没有任何解析器能够处理异常，所以异常会被抛出
> >
> > > 如果没有任何人能够处理，最终底层就会发送/error请求，会被底层的BasicErrorController处理
> > >
> > > 解析错误试图：遍历所有的ErrorViewResolver对error视图进行处理（只有一个叫`DefaultErrorViewResolver`）

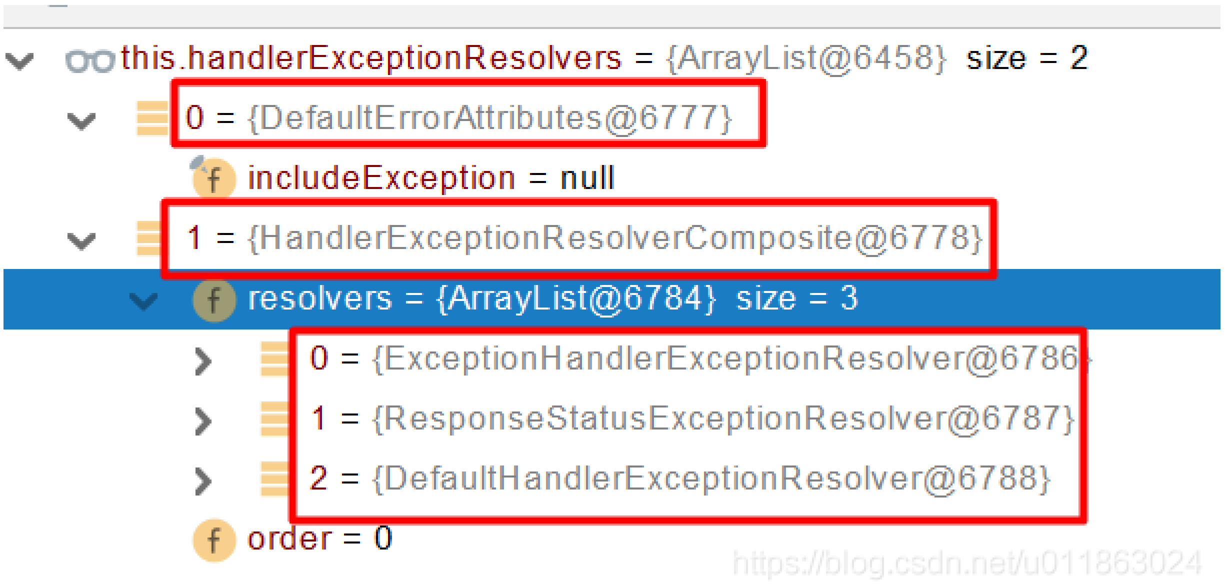
譬如写一个会抛出异常的控制层：

```java
@Slf4j
@RestController
public class HelloController {

    @RequestMapping("/hello")
    public String handle01(){

        int i = 1 / 0;//将会抛出ArithmeticException

        log.info("Hello, Spring Boot 2!");
        return "Hello, Spring Boot 2!";
    }
}
```

当浏览器发出`/hello`请求，`DispatcherServlet`的`doDispatch()`的`mv = ha.handle(processedRequest, response, mappedHandler.getHandler());`将会抛出`ArithmeticException`。

```java
public class DispatcherServlet extends FrameworkServlet {
    ...
    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        ...
                // Actually invoke the handler.
                //将会抛出ArithmeticException
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

                applyDefaultViewName(processedRequest, mv);
                mappedHandler.applyPostHandle(processedRequest, response, mv);
            }
            catch (Exception ex) {
                //将会捕捉ArithmeticException
                dispatchException = ex;
            }
            catch (Throwable err) {
                ...
            }
            //捕捉后，继续运行
            processDispatchResult(processedRequest, response, mappedHandler, mv, 
dispatchException);
        }
        catch (Exception ex) {
            triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
        }
        catch (Throwable err) {
            triggerAfterCompletion(processedRequest, response, mappedHandler,
                    new NestedServletException("Handler processing failed", err));
        }
        finally {
            ...
        }
    }

    private void processDispatchResult(HttpServletRequest request, HttpServletResponse 
response,
            @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,
            @Nullable Exception exception) throws Exception {

        boolean errorView = false;

        if (exception != null) {
            if (exception instanceof ModelAndViewDefiningException) {
                ...
            }
            else {
                Object handler = (mappedHandler != null ? mappedHandler.getHandler() : 
null);
                //ArithmeticException将在这处理
                mv = processHandlerException(request, response, handler, exception);
                errorView = (mv != null);
            }
        }
        ...
    }

    protected ModelAndView processHandlerException(HttpServletRequest request, 
HttpServletResponse response,
            @Nullable Object handler, Exception ex) throws Exception {

        // Success and error responses may use different content types
        request.removeAttribute(HandlerMapping.PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE);

        // Check registered HandlerExceptionResolvers...
        ModelAndView exMv = null;
        if (this.handlerExceptionResolvers != null) {
            //遍历所有的 handlerExceptionResolvers，看谁能处理当前异常HandlerExceptionResolver处理器异常解析器
            for (HandlerExceptionResolver resolver : this.handlerExceptionResolvers) {
                exMv = resolver.resolveException(request, response, handler, ex);
                if (exMv != null) {
                    break;
                }
            }
        }
        ...

        //若只有系统的自带的异常解析器（没有自定义的），异常还是会抛出
        throw ex;
    }

}
```

系统自带的**异常解析器**：



```
this.handlerExceptionResolvers = {ArrayList@6458}  size = 2
    0 = {DefaultErrorAttributes@6777}
        includeException = null
    1 = {HandlerExceptionResolverComposite@6778}
        resolvers = {ArrayList@6784}  size = 3
            0 = {ExceptionHandlerExceptionResolver@6786}
            1 = {ResponseStatusExceptionResolver@6787}
            2 = {DefaultHandlerExceptionResolver@6788}
        order = 0
```

- `DefaultErrorAttributes`先来处理异常，它主要功能把异常信息保存到request域，并且返回null。

```java
public class DefaultErrorAttributes implements ErrorAttributes, HandlerExceptionResolver, Ordered {
    ...
    public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {
        this.storeErrorAttributes(request, ex);
        return null;
    }

    private void storeErrorAttributes(HttpServletRequest request, Exception ex) {
        request.setAttribute(ERROR_ATTRIBUTE, ex);//把异常信息保存到request域
    }
    ...

}
```

- 默认没有任何解析器（上图的`HandlerExceptionResolverComposite`）能处理异常，所以最后异常会被抛出。
- 最终底层就会转发`/error` 请求。会被底层的`BasicErrorController`处理。

```java
@Controller
```

```java
@RequestMapping("${server.error.path:${error.path:/error}}")
public class BasicErrorController extends AbstractErrorController {

    @RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
    public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response) {
        HttpStatus status = getStatus(request);
        Map<String, Object> model = Collections
              .unmodifiableMap(getErrorAttributes(request, getErrorAttributeOptions(request, MediaType.TEXT_HTML)));
        response.setStatus(status.value());
        ModelAndView modelAndView = resolveErrorView(request, response, status, model);
        //如果/template/error内没有4**.html或5**.html，
        //modelAndView为空，最终还是返回viewName为error的modelAndView
        return (modelAndView != null) ? modelAndView : new ModelAndView("error", model);
    }

    ...
}
```

```java
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {

    ...

    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        ...
        // Actually invoke the handler.
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
        ...
        //渲染页面
        processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
        ...
    }

    private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
            @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,
            @Nullable Exception exception) throws Exception {

        boolean errorView = false;
        ...
        // Did the handler return a view to render?
        if (mv != null && !mv.wasCleared()) {
            render(mv, request, response);
            if (errorView) {
                WebUtils.clearErrorRequestAttributes(request);
            }
        }
        ...
    }

    protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) throws Exception {
        ...

        View view;
        String viewName = mv.getViewName();
        if (viewName != null) {
            // We need to resolve the view name.
            //找出合适error的View，如果/template/error内没有4**.html或5**.html，
            //将会返回默认异常页面ErrorMvcAutoConfiguration.StaticView
            //这里按需深究代码吧！
            view = resolveViewName(viewName, mv.getModelInternal(), locale, request);
            ...
        }
        ...
        try {
            if (mv.getStatus() != null) {
                response.setStatus(mv.getStatus().value());
            }
            //看下面代码块的StaticView的render块
            view.render(mv.getModelInternal(), request, response);
        }
        catch (Exception ex) {
            ...
        }
    }

}
```

```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class })
// Load before the main WebMvcAutoConfiguration so that the error View is available
@AutoConfigureBefore(WebMvcAutoConfiguration.class)
@EnableConfigurationProperties({ ServerProperties.class, ResourceProperties.class,
WebMvcProperties.class })
public class ErrorMvcAutoConfiguration {

    ...

    @Configuration(proxyBeanMethods = false)
    @ConditionalOnProperty(prefix = "server.error.whitelabel", name = "enabled",
matchIfMissing = true)
    @Conditional(ErrorTemplateMissingCondition.class)
    protected static class WhitelabelErrorViewConfiguration {

        //将创建一个名为error的系统默认异常页面View的Bean
        private final StaticView defaultErrorView = new StaticView();

        @Bean(name = "error")
        @ConditionalOnMissingBean(name = "error")
        public View defaultErrorView() {
            return this.defaultErrorView;
        }

        // If the user adds @EnableWebMvc then the bean name view resolver from
        // WebMvcAutoConfiguration disappears, so add it back in to avoid disappointment.
        @Bean
        @ConditionalOnMissingBean
        public BeanNameViewResolver beanNameViewResolver() {
            BeanNameViewResolver resolver = new BeanNameViewResolver();
            resolver.setOrder(Ordered.LOWEST_PRECEDENCE - 10);
            return resolver;
        }

    }


    private static class StaticView implements View {

        private static final MediaType TEXT_HTML_UTF8 = new MediaType("text", "html", StandardCharsets.UTF_8);

        private static final Log logger = LogFactory.getLog(StaticView.class);

        @Override
        public void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response)
                throws Exception {
            if (response.isCommitted()) {
                String message = getMessage(model);
                logger.error(message);
                return;
            }
            response.setContentType(TEXT_HTML_UTF8.toString());
            StringBuilder builder = new StringBuilder();
            Object timestamp = model.get("timestamp");
            Object message = model.get("message");
            Object trace = model.get("trace");
            if (response.getContentType() == null) {
                response.setContentType(getContentType());
            }
            //系统默认异常页面html代码
            builder.append("<html><body><h1>Whitelabel Error Page</h1>").append(
                    "<p>This application has no explicit mapping for /error, so you are seeing this as a fallback.</p>")
                    .append("<div id='created'>").append(timestamp).append("</div>")
                    .append("<div>There was an unexpected error (type=").append(htmlEscape(model.get("error")))
                    .append(", status=").append(htmlEscape(model.get("status"))).append(").</div>");
            if (message != null) {
                builder.append("<div>").append(htmlEscape(message)).append("</div>");
            }
            if (trace != null) {
                builder.append("<div style='white-space:pre-wrap;'>").append(htmlEscape(trace)).append("</div>");
            }
            builder.append("</body></html>");
            response.getWriter().append(builder.toString());
        }

        private String htmlEscape(Object input) {
            return (input != null) ? HtmlUtils.htmlEscape(input.toString()) : null;
        }

        private String getMessage(Map<String, ?> model) {
            Object path = model.get("path");
            String message = "Cannot render error page for request [" + path + "]";
            if (model.get("message") != null) {
                message += " and exception [" + model.get("message") + "]";
            }
            message += " as the response has already been committed.";
            message += " As a result, the response may have the wrong status code.";
            return message;
        }

        @Override
        public String getContentType() {
            return "text/html";
        }

    }
}
```

55、错误处理-【源码流程】几种异常处理原理

- 第一种异常处理的方式
    - 自定义错误页
        - error/404.html   error/5xx.html；有精确的错误状态码页面就匹配精确，没有就找 4xx.html；如果都没有就触发白页
- `@ControllerAdvice`+`@ExceptionHandler`处理全局异常；底层是 `ExceptionHandlerExceptionResolver` 支持的

```java
@Slf4j
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler({ArithmeticException.class,NullPointerException.class})  //处理异常
    public String handleArithException(Exception e){

        log.error("异常是：{}",e);
        return "login"; //视图地址
    }
}
```

- 第二种异常处理的方式
    - `@ResponseStatus`+自定义异常； 底层是 `ResponseStatusExceptionResolver` ，把`responseStatus`注解的信息底层调用 `response.sendError(statusCode, resolvedReason)`，tomcat发送的`/error`

```java
@ResponseStatus(value= HttpStatus.FORBIDDEN,reason = "用户数量太多") // 这个异常可以返回一个状态码
public class UserTooManyException extends RuntimeException {

    public  UserTooManyException(){

    }
    public  UserTooManyException(String message){
        super(message);
    }
}
```

```java
@Controller
public class TableController {

    @GetMapping("/dynamic_table")
    public String dynamic_table(@RequestParam(value="pn",defaultValue = "1") Integer pn,Model model){
        //表格内容的遍历
         List<User> users = Arrays.asList(new User("zhangsan", "123456"),
                new User("lisi", "123444"),
                new User("haha", "aaaaa"),
                new User("hehe ", "aaddd"));
        model.addAttribute("users",users);

        if(users.size()>3){
            throw new UserTooManyException();//抛出自定义异常
        }
        return "table/dynamic_table";
    }

}
```

- 第三种异常处理的方式
  - Spring自家异常如 `org.springframework.web.bind.MissingServletRequestParameterException` , `DefaultHandlerExceptionResolver` 处理Spring自家异常。
  - 此次请求立即结束，转到默认错误页：`response.sendError(HttpServletResponse.SC_BAD_REQUEST/*400*/, ex.getMessage());`
- 第四种异常处理的方式
  - 自定义实现 `HandlerExceptionResolver` 处理异常；可以作为默认的全局异常处理规则

```java
@Order(value= Ordered.HIGHEST_PRECEDENCE)  //优先级，数字越小优先级越高
@Component
public class CustomerHandlerExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest request,
                                         HttpServletResponse response,
                                         Object handler, Exception ex) {

        try {
            response.sendError(511,"我喜欢的错误");
        } catch (IOException e) {
            e.printStackTrace();
        }
        return new ModelAndView();
    }
}
```

- 第五种异常处理的方式
    - `ErrorViewResolver` 实现自定义处理异常
        - `response.sendError()`，error请求就会转给controller。
        - 你的异常没有任何人能处理，tomcat底层调用`response.sendError()`，error请求就会转给controller。
        - `basicErrorController` 要去的页面地址是 `ErrorViewResolver` 。

```java
@Controller
@RequestMapping("${server.error.path:${error.path:/error}}")
public class BasicErrorController extends AbstractErrorController {
    @RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
    public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse
response) {
        HttpStatus status = getStatus(request);
        Map<String, Object> model = Collections
                .unmodifiableMap(getErrorAttributes(request,
getErrorAttributeOptions(request, MediaType.TEXT_HTML)));
        response.setStatus(status.value());
        ModelAndView modelAndView = resolveErrorView(request, response, status, model);
        return (modelAndView != null) ? modelAndView : new ModelAndView("error", model);
    }

    protected ModelAndView resolveErrorView(HttpServletRequest request,
HttpServletResponse response, HttpStatus status,
            Map<String, Object> model) {
        //这里用到ErrorViewResolver接口
        for (ErrorViewResolver resolver : this.errorViewResolvers) {
            ModelAndView modelAndView = resolver.resolveErrorView(request, status, model);
            if (modelAndView != null) {
                return modelAndView;
            }
        }
        return null;
    }
    ...
}
```

```java
@FunctionalInterface
public interface ErrorViewResolver {
    ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status,
Map<String, Object> model);
}
```

56、原生组件注入-原生注解与Spring方式注入

原生组件：Servlets，Filters，Listeners

一、使用原生的注解

自定义组件，实现对应的接口，在主方法中加入扫描注解；

```java
@WebServlet(urlPatterns = "/my")
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        resp.getWriter().write("66666");
    }
}
```

```java
@Slf4j
@WebFilter(urlPatterns={"/css/*","/images/*"}) //my
public class MyFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        log.info("MyFilter初始化完成");
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) throws IOException, ServletException {
        log.info("MyFilter工作");
        chain.doFilter(request,response);
    }

    @Override
    public void destroy() {
        log.info("MyFilter销毁");
    }
}
```

```java
@Slf4j
@WebListener
public class MyServletContextListener implements ServletContextListener {


    @Override
    public void contextInitialized(ServletContextEvent sce) {
        log.info("MySwervletContextListener监听到项目初始化完成");
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        log.info("MySwervletContextListener监听到项目销毁");
    }
}
```

最后还要在主启动类添加注解 `@ServletComponentScan`，表明Servlet扫描

```java
@ServletComponentScan(basePackages = "com.lun")//
@SpringBootApplication(exclude = RedisAutoConfiguration.class)
public class Boot05WebAdminApplication {
    public static void main(String[] args) {
        SpringApplication.run(Boot05WebAdminApplication.class, args);
    }
}
```

二、Spring方式注入——写一个配置类

`ServletRegistrationBean`, `FilterRegistrationBean`, and `ServletListenerRegistrationBean`

```java
@Configuration(proxyBeanMethods = true) // true是单例
public class MyRegistConfig {

    @Bean
    public ServletRegistrationBean myServlet(){
        MyServlet myServlet = new MyServlet();

        return new ServletRegistrationBean(myServlet,"/my","/my02");
    }


    @Bean
    public FilterRegistrationBean myFilter(){

        MyFilter myFilter = new MyFilter();
//        return new FilterRegistrationBean(myFilter,myServlet());
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(myFilter);
        filterRegistrationBean.setUrlPatterns(Arrays.asList("/my","/css/*"));
        return filterRegistrationBean;
    }

    @Bean
    public ServletListenerRegistrationBean myListener(){
        MySwervletContextListener mySwervletContextListener = new MySwervletContextListener();
        return new ServletListenerRegistrationBean(mySwervletContextListener);
    }
}
```

57、原生组件注入-【源码分析】DispatcherServlet注入原理

> DispatchServlet如何注入进来
>
> - 容器中自动配置了DispatcherServlet属性绑定到WebMvcProperties；对应的配置文件项是spring.mvc
> - 通过`ServletRegistrationBean<DispatcherServlet>`把DispatcherServlet配置进来

`org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration`配置类

```java
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration(proxyBeanMethods = false)
```

```java
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass(DispatcherServlet.class)
@AutoConfigureAfter(ServletWebServerFactoryAutoConfiguration.class)
public class DispatcherServletAutoConfiguration {

    /*
     * The bean name for a DispatcherServlet that will be mapped to the root URL "/"
     */
    public static final String DEFAULT_DISPATCHER_SERVLET_BEAN_NAME = "dispatcherServlet";

    /*
     * The bean name for a ServletRegistrationBean for the DispatcherServlet "/"
     */
    public static final String DEFAULT_DISPATCHER_SERVLET_REGISTRATION_BEAN_NAME = "dispatcherServletRegistration";

    @Configuration(proxyBeanMethods = false)
    @Conditional(DefaultDispatcherServletCondition.class)
    @ConditionalOnClass(ServletRegistration.class)
    @EnableConfigurationProperties(WebMvcProperties.class)
    protected static class DispatcherServletConfiguration {
        //创建DispatcherServlet类的Bean
        @Bean(name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
        public DispatcherServlet dispatcherServlet(WebMvcProperties webMvcProperties) {
            DispatcherServlet dispatcherServlet = new DispatcherServlet();
            dispatcherServlet.setDispatchOptionsRequest(webMvcProperties.isDispatchOptionsRequest());
            dispatcherServlet.setDispatchTraceRequest(webMvcProperties.isDispatchTraceRequest());
            dispatcherServlet.setThrowExceptionIfNoHandlerFound(webMvcProperties.isThrowExceptionIfNoHandlerFound());
            dispatcherServlet.setPublishEvents(webMvcProperties.isPublishRequestHandledEvents());
            dispatcherServlet.setEnableLoggingRequestDetails(webMvcProperties.isLogRequestDetails());
            return dispatcherServlet;
        }

        @Bean
        @ConditionalOnBean(MultipartResolver.class)
        @ConditionalOnMissingBean(name = DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)
        public MultipartResolver multipartResolver(MultipartResolver resolver) {
            // Detect if the user has created a MultipartResolver but named it incorrectly
            return resolver;
        }

    }

    @Configuration(proxyBeanMethods = false)
    @Conditional(DispatcherServletRegistrationCondition.class)
    @ConditionalOnClass(ServletRegistration.class)
    @EnableConfigurationProperties(WebMvcProperties.class)
    @Import(DispatcherServletConfiguration.class)
    protected static class DispatcherServletRegistrationConfiguration {
        //注册DispatcherServlet类
        @Bean(name = DEFAULT_DISPATCHER_SERVLET_REGISTRATION_BEAN_NAME)
        @ConditionalOnBean(value = DispatcherServlet.class, name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
        public DispatcherServletRegistrationBean
dispatcherServletRegistration(DispatcherServlet dispatcherServlet,
                WebMvcProperties webMvcProperties, ObjectProvider<MultipartConfigElement>
multipartConfig) {
            DispatcherServletRegistrationBean registration = new
DispatcherServletRegistrationBean(dispatcherServlet,
                    webMvcProperties.getServlet().getPath());
            registration.setName(DEFAULT_DISPATCHER_SERVLET_BEAN_NAME);

registration.setLoadOnStartup(webMvcProperties.getServlet().getLoadOnStartup());
            multipartConfig.ifAvailable(registration::setMultipartConfig);
            return registration;
        }
    }

    ...

}
```

`DispatcherServlet` 默认映射的是 `/` 路径，可以通过在配置文件修改 `spring.mvc.servlet.path=/mvc`。

Tomcat的servlet：

多个Servlet都能处理到同一个路径，精确匹配优先原则；

58、嵌入式Servlet容器-【源码分析】切换web服务器与定制化

- 默认支持的WebServer
  - `Tomcat`, `Jetty`, or `Undertow`。
  - `ServletWebServerApplicationContext` 容器启动寻找 `ServletWebServerFactory` 并引导创建服务器。
- 原理
  - SpringBoot应用启动发现当前是Web应用，web场景包-导入tomcat。
  - web应用会创建一个web版的IOC容器 `ServletWebServerApplicationContext` 。
  - `ServletWebServerApplicationContext` 启动的时候寻找 `ServletWebServerFactory` （Servlet 的web服务器工厂——>Servlet 的web服务器）。
  - SpringBoot底层默认有很多的WebServer工厂（`ServletWebServerFactoryConfiguration`内创建Bean），如：
    - `TomcatServletWebServerFactory`
    - `JettyServletWebServerFactory`
    - `UndertowServletWebServerFactory`
  - 底层直接会有一个自动配置类 `ServletWebServerFactoryAutoConfiguration`。
  - `ServletWebServerFactoryAutoConfiguration` 导入了 `ServletWebServerFactoryConfiguration`（配置类）。
  - `ServletWebServerFactoryConfiguration` 根据动态判断系统中到底导入了那个Web服务器的包。（默认是web-starter导入tomcat包），容器中就有 `TomcatServletWebServerFactory`
  - `TomcatServletWebServerFactory` 创建出Tomcat服务器并启动；`TomcatWebServer` 的构造器拥有初始化方法initialize——`this.tomcat.start();`

    - 内嵌服务器，与以前手动把启动服务器相比，改成现在使用代码启动（tomcat核心jar包存在）。

Spring Boot默认使用Tomcat服务器，若需更改其他服务器，则修改工程pom.xml：

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

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

```

定制Servlet容器

- 实现 `WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>`
- 把配置文件的值和 `ServletWebServerFactory` 进行绑定
- 修改配置文件 `server.xxx`
- 直接自定义 `ConfigurableServletWebServerFactory`

`xxxxxCustomizer`：定制化器，可以改变xxxx的默认规则

```java
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory server) {
        server.setPort(9000);
    }

}
```

59、定制化原理-SpringBoot定制化组件的几种方式（小结）

一、定制化的常见方式

- 修改配置文件
- `xxxxxCustomizer`
- 编写自定义的配置类 `xxxConfiguration` + `@Bean`替换、增加容器中默认组件，视图解析器
- **Web应用 编写一个配置类实现 `WebMvcConfigurer` 即可定制化web功能 + `@Bean`给容器中再扩展一些组件【常用的】**

```java
@Configuration
public class AdminWebConfig implements WebMvcConfigurer{

}
```

- `@EnableWebMvc` + `WebMvcConfigurer` — `@Bean` 可以全面接管SpringMVC，所有规则全部自己重新配置； 实现定制和扩展功能（**高级功能，初学者退避三舍**）。使用了`@EnableWebMvc`会导致`WebMvcAutoConfiguration`失效。
    - 原理：
        1. `WebMvcAutoConfiguration`是默认的SpringMVC的自动配置功能类，如静态资源、欢迎页等。
        2. 一旦使用 `@EnableWebMvc` ，会`@Import(DelegatingWebMvcConfiguration.class)`。
        3. `DelegatingWebMvcConfiguration`的作用，只保证SpringMVC最基本的使用
            - 把所有系统中的`WebMvcConfigurer`拿过来，所有功能的定制都是这些`WebMvcConfigurer`合起来一起生效。
            - 自动配置了一些非常底层的组件，如`RequestMappingHandlerMapping`，这些组件依赖的组件都是从容器中获取如。
            - `public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport`。
        4. `WebMvcAutoConfiguration`里面的配置要能生效必须 `@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)`。【也就是没有这个类型才会加载，而使用了 `@EnableWebMvc` ，会`@Import(DelegatingWebMvcConfiguration.class)`。】
        5. @EnableWebMvc 导致了WebMvcAutoConfiguration 没有生效。

二、原理分析套路

导入一个场景Starter

由Starter导入自动配置类`XXXXAutoConfiguration`

通过自动配置类将各种组件导入，并根据`xxxProperties`文件进行配置

【所以对于基本开发需求来说，只需要导入场景的`Starter`并修改`xxxProperties`配置文件】

60、数据访问-数据库场景的自动配置分析与整合测试

一、导入JDBC场景

- 先引入JDBC
- 再导入数据库驱动

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

接着导入数据库驱动包（MySQL为例）。

```xml
<!--默认版本：-->
<mysql.version>8.0.22</mysql.version>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <!--直接导入确定的5.1.49<version>5.1.49</version>-->
</dependency>

<!--
想要修改版本
1、直接依赖引入具体版本（maven的就近依赖原则）
2、重新声明版本（maven的属性的就近优先原则）
-->
<properties>
    <java.version>1.8</java.version>
    <mysql.version>5.1.49</mysql.version>
</properties>
```

二、相关数据源配置类

- `DataSourceAutoConfiguration` ： 数据源的自动配置。
  - 修改数据源相关的配置：`spring.datasource`。
  - **数据库连接池的配置，是自己容器中没有DataSource才自动配置的。**
  - 底层配置好的连接池是：`HikariDataSource`。
- `DataSourceTransactionManagerAutoConfiguration`： 事务管理器的自动配置。
- `JdbcTemplateAutoConfiguration`： `JdbcTemplate`的自动配置，可以来对数据库进行CRUD。
  - 可以修改前缀为`spring.jdbc`的配置项来修改`JdbcTemplate`。
  - `@Bean @Primary JdbcTemplate`：Spring容器中有这个`JdbcTemplate`组件，使用`@Autowired`自动装配。
- `JndiDataSourceAutoConfiguration`： JNDI的自动配置。
- `XADataSourceAutoConfiguration`： 分布式事务相关的。

三、修改配置项

```yaml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/db_account
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
```

四、单元测试数据源

```java
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.jdbc.core.JdbcTemplate;

@SpringBootTest
class Boot05WebAdminApplicationTests {
    @Autowired
    JdbcTemplate jdbcTemplate;
    @Test//用@org.junit.Test会报空指针异常，可能跟JUnit新版本有关
    void contextLoads() {
//        jdbcTemplate.queryForObject("select * from account_tbl")
//        jdbcTemplate.queryForList("select * from account_tbl",)
        Long aLong = jdbcTemplate.queryForObject("select count(*) from account_tbl", Long.class);
        log.info("记录总数：{}",aLong);
    }
}
```

61、数据访问——自定义方式整合`druid`数据源

一、Druid是什么？

它是数据库连接池，它能够提供强大的监控和扩展功能。

Spring Boot整合第三方技术的两种方式：

- 自定义
- 找`starter`场景

二、自定义方式

添加依赖：`pom.xml`

```xml
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.17</version>
</dependency>
```

配置Druid数据源：

```java
@Configuration
public class MyConfig {
    @Bean
    @ConfigurationProperties("spring.datasource") //复用配置文件的数据源配置，使用
spring.datasource中的配置信息进行配置
    public DataSource dataSource() throws SQLException {
        DruidDataSource druidDataSource = new DruidDataSource();
//        druidDataSource.setUrl();
//        druidDataSource.setUsername();
//        druidDataSource.setPassword();
        return druidDataSource;
    }
}
```

配置Druid的监控页功能：

- Druid内置提供了一个`StatViewServlet`用于展示Druid的统计信息。这个`StatViewServlet`的用途包括：
  - 提供监控信息展示的html页面
  - 提供监控信息的JSON API
- Druid内置提供一个`StatFilter`，用于统计监控信息。
- `WebStatFilter`用于采集web-jdbc关联监控的数据，如SQL监控、URI监控。
- Druid提供了`WallFilter`，它是基于SQL语义分析来实现防御SQL注入攻击的。

```java
@Configuration
public class MyConfig {
    @Bean
    @ConfigurationProperties("spring.datasource")
    public DataSource dataSource() throws SQLException {
        DruidDataSource druidDataSource = new DruidDataSource();
        //加入监控和防火墙功能功能
        druidDataSource.setFilters("stat,wall");
        return druidDataSource;
    }

    /**
     * 配置 druid的监控页功能
     * @return
     */
    @Bean
    public ServletRegistrationBean statViewServlet(){
        StatViewServlet statViewServlet = new StatViewServlet();
        ServletRegistrationBean<StatViewServlet> registrationBean =
            new ServletRegistrationBean<>(statViewServlet, "/druid/*");
        //监控页账号密码：
        registrationBean.addInitParameter("loginUsername","admin");
        registrationBean.addInitParameter("loginPassword","123456");
        return registrationBean;
    }

     /**
     * WebStatFilter 用于采集web-jdbc关联监控的数据。
     */
    @Bean
    public FilterRegistrationBean webStatFilter(){
        WebStatFilter webStatFilter = new WebStatFilter();
        FilterRegistrationBean<WebStatFilter> filterRegistrationBean = new
FilterRegistrationBean<>(webStatFilter);
```

```
        filterRegistrationBean.setUrlPatterns(Arrays.asList("/*"));

 filterRegistrationBean.addInitParameter("exclusions","*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*");
        return filterRegistrationBean;
    }

}
```

62、数据访问——druid数据源starter整合方式

【回顾】starter —— 自动配置类 —— 配置文件 —— 完成配置

引入依赖：

```xml
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.17</version>
</dependency>
```

分析自动配置：

- 扩展配置项 `spring.datasource.druid`
- 自动配置类 `DruidDataSourceAutoConfigure`
- `DruidSpringAopConfiguration.class`，监控SpringBean的组件；配置项：`spring.datasource.druid.aop-patterns`
- `DruidStatViewServletConfiguration.class`，监控页的配置。`spring.datasource.druid.stat-view-servlet` 默认开启。
- `DruidWebStatFilterConfiguration.class`，web监控配置。`spring.datasource.druid.web-stat-filter` 默认开启。
- `DruidFilterConfiguration.class` 所有 `Druid` 的 `filter` 的配置：

```java
private static final String FILTER_STAT_PREFIX = "spring.datasource.druid.filter.stat";
private static final String FILTER_CONFIG_PREFIX = "spring.datasource.druid.filter.config";
private static final String FILTER_ENCODING_PREFIX = "spring.datasource.druid.filter.encoding";
private static final String FILTER_SLF4J_PREFIX = "spring.datasource.druid.filter.slf4j";
private static final String FILTER_LOG4J_PREFIX = "spring.datasource.druid.filter.log4j";
private static final String FILTER_LOG4J2_PREFIX = "spring.datasource.druid.filter.log4j2";
private static final String FILTER_COMMONS_LOG_PREFIX = "spring.datasource.druid.filter.commons-log";
private static final String FILTER_WALL_PREFIX = "spring.datasource.druid.filter.wall";
```

配置示例：

```yaml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/db_account
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver

    druid:
      aop-patterns: com.atguigu.admin.*  #监控SpringBean
      filters: stat,wall     # 底层开启功能，stat（sql监控），wall（防火墙）

      stat-view-servlet:   # 配置监控页功能
        enabled: true      # 将监控功能开启
        login-username: admin  # 对监控页面配置账号密码
        login-password: admin  # 对监控页面配置账号密码
        resetEnable: false # 重置按钮

      web-stat-filter:  # 监控web
        enabled: true
        urlPattern: /*
        exclusions: '*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*'

      filter:
        stat:    # 对上面filters里面的stat的详细配置
          slow-sql-millis: 1000
          logSlowSql: true
          enabled: true
        wall:
          enabled: true
          config:
            drop-table-allow: false
```

63、数据访问-整合MyBatis-配置版

starter的命名方式：

1. SpringBoot官方的Starter： `spring-boot-starter-*` 【Spring官方的一般都是以Spring-boot-starter开头】
2. 第三方的： `*-spring-boot-starter` 【第三方的都是以第三方名称开始】

引入starter依赖：

```xml
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.4</version>
</dependency>
```

配置模式：

- 全局配置文件
- `SqlSessionFactory`：自动配置好了
- `SqlSession`：自动配置了`SqlSessionTemplate` 组合了`SqlSession`
- `@Import(AutoConfiguredMapperScannerRegistrar.class)`
- `Mapper`： 只要我们写的操作MyBatis的接口标注了`@Mapper`就会被自动扫描进来

```java
@EnableConfigurationProperties(MybatisProperties.class) ： MyBatis配置项绑定类。
@AutoConfigureAfter({ DataSourceAutoConfiguration.class, MybatisLanguageDriverAutoConfiguration.class })
public class MybatisAutoConfiguration{
    ...
}

@ConfigurationProperties(prefix = "mybatis")
public class MybatisProperties{
    ...
}
```

配置文件：【myBatis需要自己单独的配置文件】

```yaml
spring:
  datasource:
    username: root
    password: 1234
    url: jdbc:mysql://localhost:3306/my
    driver-class-name: com.mysql.jdbc.Driver

# 配置mybatis规则
mybatis:
  config-location: classpath:mybatis/mybatis-config.xml  #全局配置文件位置
  mapper-locations: classpath:mybatis/*.xml  #sql映射文件位置
```

mybatis-config.xml：

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 由于Spring Boot自动配置缘故，此处不必配置，只用来做做样。-->
</configuration>
```

Mapper接口：

```xml
<?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.lun.boot.mapper.UserMapper">
    <select id="getUser" resultType="com.lun.boot.bean.User">
        select * from user where id=#{id}
    </select>
</mapper>
```

```java
import com.lun.boot.bean.User;
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface UserMapper {
    public User getUser(Integer id);
}
```

POJO：

```java
public class User {
    private Integer id;
    private String name;

    //getters and setters...
}
```

DB：

```sql
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4;
```

Service：

```java
@Service
public class UserService {

    @Autowired
    private UserMapper userMapper;//IDEA下标红线，可忽视这红线

    public User getUser(Integer id){
        return userMapper.getUser(id);
    }

}
```

Controller：

```java
@Controller
public class UserController {

    @Autowired
    private UserService userService;

    @ResponseBody
    @GetMapping("/user/{id}")
    public User getUser(@PathVariable("id") Integer id){
        return userService.getUser(id);
    }

}
```

配置`private Configuration configuration;`也就是配置`mybatis.configuration`相关的，就是相当于改mybatis全局配置文件中的值。（也就是说配置了`mybatis.configuration`，就不需配置mybatis全局配置文件了）

```yaml
# 配置mybatis规则
mybatis:
  mapper-locations: classpath:mybatis/mapper/*.xml
  # 可以不写全局配置文件，所有全局配置文件的配置都放在configuration配置项中了。
  # config-location: classpath:mybatis/mybatis-config.xml
  configuration:
    map-underscore-to-camel-case: true

    # 配置mybatis规则
mybatis:
  config-location: classpath:mybatis/mybatis-config.xml  #全局配置文件位置
  mapper-locations: classpath:mybatis/*.xml  #sql映射文件位置
```

小结

- 导入MyBatis官方Starter。
- 编写Mapper接口，需`@Mapper`注解。
- 编写SQL映射文件`.xml`并绑定Mapper接口。
- 在`application.yaml`中指定Mapper配置文件的所处位置

  ```yaml
  # 配置mybatis规则
  mybatis:
    config-location: classpath:mybatis/mybatis-config.xml  #全局配置文件位置
    mapper-locations: classpath:mybatis/*.xml  #mapper.xml映射文件位置
  ```

- 以及指定全局配置文件的信息 （建议：**配置在**`mybatis.configuration`）。【也可以不写这个配置文件，直接在yaml中的mybatis的configuration中进行配置】

  ```xml
  <?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
  <configuration>
      <!-- 由于Spring Boot自动配置缘故，此处不必配置，只用来做做样。-->
  </configuration>
  ```

64、数据访问-整合MyBatis-注解配置混合版

注解与配置混合搭配，干活不累：

```java
@Mapper
public interface UserMapper {
    public User getUser(Integer id);

    @Select("select * from user where id=#{id}")
    public User getUser2(Integer id);

    // xml方式
    public void saveUser(User user);

```

```java
    @Insert("insert into user(`name`) values(#{name})")
    @Options(useGeneratedKeys = true, keyProperty = "id")
    public void saveUser2(User user);

}

```

```xml
<?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.lun.boot.mapper.UserMapper">

    <select id="getUser" resultType="com.lun.boot.bean.User">
        select * from user where id=#{id}
    </select>

    <insert id="saveUser" useGeneratedKeys="true" keyProperty="id">
        insert into user(`name`) values(#{name})
    </insert>

</mapper>
```

- 简单DAO方法就写在注解上。复杂的就写在配置文件里。
- 使用`@MapperScan("com.lun.boot.mapper")` 简化，Mapper接口就可以不用标注`@Mapper`注解。

```java
@MapperScan("com.lun.boot.mapper")
@SpringBootApplication
public class MainApplication {
    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class, args);
    }
}
```

> 小结：
>
> - 引入mybatis-starter
> - 配置application.yaml中，指定mapper.xml的位置
> - 编写Mapper接口并加上`@Mapper`，对应的SQL语句可以采用注解方式也可以采用xml文件的形式
> - 也可以在SpringBoot主程序入口处加上`@MapperScan("com.atguigu.admin.mapper")`简化，这样其他的接口就都可以不使用`@Mapper`注解了。
>
> ```java
> @MapperScan("com.atguigu.admin.mapper")
> @ServletComponentScan(basePackages = "com.atguigu.admin")
> public class BootWebApplication{
>     public static void main(String[] args){
>         ...
>     }
> }
> ```

65、数据访问-整合MyBatisPlus操作数据库

一、MyBatisPlus是什么

MyBatis-Plus（简称 MP）是一个 MyBatis的增强工具，在 MyBatis 的基础上只做增强不做改变，为简化开发、提高效率而生。

添加依赖：

```xml
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.1</version>
</dependency>
```

- 添加依赖后自动导入了`MybatisPlusAutoConfiguration`配置类，`MybatisPlusProperties`配置项绑定。
- `SqlSessionFactory`自动配置好，底层是容器中默认的数据源。
- `mapperLocations`自动配置好的，有默认值`classpath*:/mapper/**/*.xml`，这表示任意包的类路径下的所有mapper文件夹下任意路径下的所有xml都是sql映射文件。 建议以后sql映射文件放在 mapper下。
- 容器中也自动配置好了`SqlSessionTemplate`。
- `@Mapper` 标注的接口也会被自动扫描，建议直接 `@MapperScan("com.lun.boot.mapper")` 批量扫描。
- MyBatisPlus**优点**之一：只需要我们的Mapper继承MyBatisPlus的`BaseMapper` 就可以拥有CRUD能力，减轻开发工作。

```java
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.lun.hellomybatisplus.model.User;

public interface UserMapper extends BaseMapper<User> {
    // 基本的BaseMapper中包含了CRUD的能力
    // 开发的时候加上一些其他的功能就可以了
}
```

66、数据访问-CRUD实验-数据列表展示

使用MyBatis Plus提供的`IService`，`ServiceImpl`，减轻Service层开发工作。

```java
import com.lun.hellomybatisplus.model.User;
import com.lun.hellomybatisplus.mapper.UserMapper;
import com.lun.hellomybatisplus.service.UserService;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class UserServiceImpl extends ServiceImpl<UserMapper,User> implements UserService {
    //此处故意为空，因为大部分方法已经通过extends被实现了
}
```

与下一节联合在一起

67、数据访问-CRUD实验-分页数据展示

与下一节联合在一起

68、数据访问-CRUD实验-删除用户完成

添加分页插件：

```java
@Configuration
public class MyBatisConfig {


    /**
     * MybatisPlusInterceptor
     * @return
     */
    @Bean
    public MybatisPlusInterceptor paginationInterceptor() {
        MybatisPlusInterceptor mybatisPlusInterceptor = new MybatisPlusInterceptor();
        // 设置请求的页面大于最大页后操作， true调回到首页，false 继续请求  默认false
        // paginationInterceptor.setOverflow(false);
        // 设置最大单页限制数量，默认 500 条，-1 不受限制
        // paginationInterceptor.setLimit(500);
        // 开启 count 的 join 优化,只针对部分 left join

        //这是分页拦截器
        PaginationInnerInterceptor paginationInnerInterceptor = new
PaginationInnerInterceptor();
        paginationInnerInterceptor.setOverflow(true);
        paginationInnerInterceptor.setMaxLimit(500L);
        mybatisPlusInterceptor.addInnerInterceptor(paginationInnerInterceptor);

        return mybatisPlusInterceptor;
    }
}
```

```html
<table class="display table table-bordered table-striped" id="dynamic-table">
    <thead>
        <tr>
            <th>#</th>
```

```html
                <th>name</th>
                <th>age</th>
                <th>email</th>
                <th>操作</th>
            </tr>
        </thead>
        <tbody>
            <tr class="gradeX" th:each="user: ${users.records}">
                <td th:text="${user.id}"></td>
                <td>[[${user.name}]]</td>
                <td th:text="${user.age}">Win 95+</td>
                <td th:text="${user.email}">4</td>
                <td>
                    <a th:href="@{/user/delete/{id}(id=${user.id},pn=${users.current})}"
                       class="btn btn-danger btn-sm" type="button">删除</a>
                </td>
            </tr>
        </tfoot>
</table>

<div class="row-fluid">
    <div class="span6">
        <div class="dataTables_info" id="dynamic-table_info">
            当前第[[${users.current}]]页  总计 [[${users.pages}]]页  共[[${users.total}]]条记录
        </div>
    </div>
    <div class="span6">
        <div class="dataTables_paginate paging_bootstrap pagination">
            <ul>
                <li class="prev disabled"><a href="#">← 前一页</a></li>
                <li th:class="${num == users.current?'active':''}"
                    th:each="num:${#numbers.sequence(1,users.pages)}" >
                    <a th:href="@{/dynamic_table(pn=${num})}">[[${num}]]</a>
                </li>
                <li class="next disabled"><a href="#">下一页 → </a></li>
            </ul>
        </div>
    </div>
</div>

```

`#numbers`表示methods for formatting numeric objects.link

```java
@GetMapping("/user/delete/{id}")
public String deleteUser(@PathVariable("id") Long id,
                         @RequestParam(value = "pn",defaultValue = "1")Integer pn,
                         RedirectAttributes ra){

    userService.removeById(id);

    ra.addAttribute("pn",pn);
    return "redirect:/dynamic_table";
}

@GetMapping("/dynamic_table")
```

```java
public String dynamic_table(@RequestParam(value="pn",defaultValue = "1") Integer pn,Model model){
    //表格内容的遍历

    //从数据库中查出user表中的用户进行展示

    //构造分页参数
    Page<User> page = new Page<>(pn, 2);
    //调用page进行分页
    Page<User> userPage = userService.page(page, null);

    model.addAttribute("users",userPage);

    return "table/dynamic_table";
}
```

69、数据访问-准备阿里云Redis环境

添加依赖：

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

<!--导入jedis-->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```

- `RedisAutoConfiguration`自动配置类，RedisProperties 属性类 --> spring.redis.xxx是对redis的配置。
- 连接工厂`LettuceConnectionConfiguration`、`JedisConnectionConfiguration`是准备好的。
- 自动注入了`RedisTemplate<Object, Object>`，`xxxTemplate`。
- 自动注入了`StringRedisTemplate`，key，value都是String
- 底层只要我们使用`StringRedisTemplate`、`RedisTemplate`就可以操作Redis。

外网Redis环境搭建：

1. 阿里云按量付费Redis，其中选择**经典网络**。
2. 申请Redis的公网连接地址。
3. 修改白名单，允许`0.0.0.0/0`访问。

70、数据访问-Redis操作与统计小实验

相关Redis配置：

```yaml
spring:
  redis:
```

```
#   url: redis://lfy:Lfy123456@r-bp1nc7reqesxisgxpipd.redis.rds.aliyuncs.com:6379
    host: r-bp1nc7reqesxisgxpipd.redis.rds.aliyuncs.com
    port: 6379
    password: lfy:Lfy123456
    client-type: jedis
    jedis:
      pool:
        max-active: 10
#   lettuce:# 另一个用来连接redis的java框架
#      pool:
#        max-active: 10
#        min-idle: 5
```

测试Redis连接：

```
@SpringBootTest
public class Boot05WebAdminApplicationTests {

    @Autowired
    StringRedisTemplate redisTemplate;


    @Autowired
    RedisConnectionFactory redisConnectionFactory;

    @Test
    void testRedis(){
        ValueOperations<String, String> operations = redisTemplate.opsForValue();

        operations.set("hello","world");

        String hello = operations.get("hello");
        System.out.println(hello);

        System.out.println(redisConnectionFactory.getClass());
    }

}
```

Redis Desktop Manager：可视化Redis管理软件。

URL统计拦截器：

```
@Component
public class RedisUrlCountInterceptor implements HandlerInterceptor {

    @Autowired
    StringRedisTemplate redisTemplate;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        String uri = request.getRequestURI();

        //默认每次访问当前uri就会计数+1
        redisTemplate.opsForValue().increment(uri);

        return true;
    }
}
```

注册URL统计拦截器:

```java
@Configuration
public class AdminWebConfig implements WebMvcConfigurer{

    @Autowired
    RedisUrlCountInterceptor redisUrlCountInterceptor;


    @Override
    public void addInterceptors(InterceptorRegistry registry) {

        registry.addInterceptor(redisUrlCountInterceptor)
                .addPathPatterns("/**")
                .excludePathPatterns("/","/login","/css/**","/fonts/**","/images/**",
                        "/js/**","/aa/**");
    }
}
```

Filter、Interceptor 几乎拥有相同的功能?

- Filter是Servlet定义的原生组件，它的好处是脱离Spring应用也能使用。
- Interceptor是Spring定义的接口，可以使用Spring的自动装配等功能。

调用Redis内的统计数据:

```java
@Slf4j
@Controller
public class IndexController {

    @Autowired
    StringRedisTemplate redisTemplate;

    @GetMapping("/main.html")
    public String mainPage(HttpSession session,Model model){

        log.info("当前方法是：{}","mainPage");

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

        String s = opsForValue.get("/main.html");
        String s1 = opsForValue.get("/sql");

        model.addAttribute("mainCount",s);
        model.addAttribute("sqlCount",s1);

        return "main";
    }
}
```

71、单元测试-JUnit5简介

Spring Boot 2.2.0 版本开始引入 JUnit 5 作为单元测试默认库

JUnit 5官方文档

作为最新版本的JUnit框架，JUnit5与之前版本的JUnit框架有很大的不同。由三个不同子项目的几个不同模块组成。

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

- **JUnit Platform**: Junit Platform是在JVM上启动测试框架的基础，不仅支持Junit自制的测试引擎，其他测试引擎也都可以接入。
- **JUnit Jupiter**: JUnit Jupiter提供了JUnit5的新的编程模型，是JUnit5新特性的核心。内部包含了一个**测试引擎**，用于在Junit Platform上运行。
- **JUnit Vintage**: 由于JUint已经发展多年，为了照顾老的项目，JUnit Vintage提供了兼容JUnit4.x，JUnit3.x的测试引擎。

注意：

- SpringBoot 2.4 以上版本移除了默认对 Vintage 的依赖。如果需要兼容JUnit4需要自行引入（不能使用JUnit4的功能 @Test）
- JUnit 5's Vintage已经从`spring-boot-starter-test`从移除。如果需要继续兼容Junit4需要自行引入Vintage依赖：

```xml
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

- 使用添加JUnit 5，添加对应的starter：

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

- Spring的JUnit 5的基本单元测试模板（Spring的JUnit4的是`@SpringBootTest`+`@RunWith(SpringRunner.class)`）：

```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;//注意不是org.junit.Test（这是JUnit4版本的）
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class SpringBootApplicationTests {

    @Autowired
    private Component component;

    @Test
    //@Transactional 标注后连接数据库有回滚功能
    public void contextLoads() {
        Assertions.assertEquals(5, component.getFive());
    }
}
```

72、单元测试-常用测试注解

[官方文档 - Annotations]

- **@Test**：表示方法是测试方法。但是与JUnit4的@Test不同，他的职责非常单一不能声明任何属性，拓展的测试将会由Jupiter提供额外测试
- **@ParameterizedTest**：表示方法是参数化测试。
- **@RepeatedTest**：表示方法可重复执行。
- **@DisplayName**：为测试类或者测试方法设置展示名称。
- **@BeforeEach**：表示在**每个**单元测试**之前**执行。
- **@AfterEach**：表示在**每个**单元测试**之后**执行。
- **@BeforeAll**：表示在**所有**单元测试**之前**执行。【静态】
- **@AfterAll**：表示在**所有**单元测试**之后**执行。【静态】
- **@Tag**：表示单元测试类别，类似于JUnit4中的@Categories。
- **@Disabled**：表示测试类或测试方法不执行，类似于JUnit4中的@Ignore。
- **@Timeout**：表示测试方法运行如果超过了指定时间将会返回错误。
- **@ExtendWith**：为测试类或测试方法提供扩展类引用。
- **@Transaction**：事务测试并自动回滚
- **@RepeatedTest(重复次数)**：多次测试

```java
import org.junit.jupiter.api.*;

@DisplayName("junit5功能测试类")
public class Junit5Test {


    @DisplayName("测试displayname注解")
    @Test
    void testDisplayName() {
        System.out.println(1);
        System.out.println(jdbcTemplate);
    }

    @ParameterizedTest
    @ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
    void palindromes(String candidate) {
        assertTrue(StringUtils.isPalindrome(candidate));
    }


    @Disabled
    @DisplayName("测试方法2")
    @Test
    void test2() {
        System.out.println(2);
    }

    @RepeatedTest(5)
    @Test
    void test3() {
        System.out.println(5);
    }

    /**
     * 规定方法超时时间。超出时间测试出异常
     *
     * @throws InterruptedException
     */
    @Timeout(value = 500, unit = TimeUnit.MILLISECONDS)
    @Test
    void testTimeout() throws InterruptedException {
        Thread.sleep(600);
    }


    @BeforeEach
    void testBeforeEach() {
        System.out.println("测试就要开始了...");
    }

    @AfterEach
    void testAfterEach() {
        System.out.println("测试结束了...");
    }

    @BeforeAll
    static void testBeforeAll() {
        System.out.println("所有测试就要开始了...");
    }

    @AfterAll
    static void testAfterAll() {
        System.out.println("所有测试以及结束了...");

    }

}
```

73、单元测试-断言机制

【前面的断言失败了，后面的代码根本就不会运行】

断言Assertion是测试方法中的核心部分，用来对测试需要满足的条件进行验证。这些断言方法都是org.junit.jupiter.api.Assertions的静态方法。检查业务逻辑返回的数据是否合理。所有的测试运行结束以后，会有一个详细的测试报告。

JUnit 5 内置的断言可以分成如下几个类别：

简单断言

用来对单个值进行简单的验证。如：

`assertXxxx(Param A, Param B, Param C): 参数A和参数B如果不满足测试断言就打印参数C中的字符串`

方法	说明
assertEquals	判断两个对象或两个原始类型是否相等
assertNotEquals	判断两个对象或两个原始类型是否不相等
assertSame	判断两个对象引用是否指向同一个对象
assertNotSame	判断两个对象引用是否指向不同的对象
assertTrue	判断给定的布尔值是否为 true
assertFalse	判断给定的布尔值是否为 false
assertNull	判断给定的对象引用是否为 null
assertNotNull	判断给定的对象引用是否不为 null

```java
@Test
@DisplayName("simple assertion")
public void simple() {
     assertEquals(3, 1 + 2, "simple math");
     assertNotEquals(3, 1 + 1);

     assertNotSame(new Object(), new Object());
     Object obj = new Object();
     assertSame(obj, obj);

     assertFalse(1 > 2);
     assertTrue(1 < 2);

     assertNull(null);
     assertNotNull(new Object());
}
```

数组断言

通过 assertArrayEquals 方法来判断两个对象或原始类型的数组是否相等。

```java
@Test
@DisplayName("array assertion")
public void array() {
    assertArrayEquals(new int[]{1, 2}, new int[] {1, 2});
}
```

组合断言

`assertAll()` 方法接受多个 `org.junit.jupiter.api.Executable` 函数式接口的实例作为要验证的断言，可以通过 lambda 表达式很容易的提供这些断言。

```java
@Test
@DisplayName("assert all")
public void all() {
 assertAll("Math",
    () -> assertEquals(2, 1 + 1),
    () -> assertTrue(1 > 0)
 );
}
```

异常断言

在JUnit4时期，想要测试方法的异常情况时，需要用`@Rule`注解的`ExpectedException`变量还是比较麻烦的。而JUnit5提供了一种新的断言方式`Assertions.assertThrows()`，配合函数式编程就可以进行使用。

```java
@Test
@DisplayName("异常测试")
public void exceptionTest() {
    ArithmeticException exception = Assertions.assertThrows(
           //扔出断言异常
            ArithmeticException.class, () -> System.out.println(1 % 0));
}
```

超时断言

JUnit5还提供了Assertions.assertTimeout()为测试方法设置了超时时间。

```java
@Test
@DisplayName("超时测试")
public void timeoutTest() {
    //如果测试方法时间超过1s将会异常
    Assertions.assertTimeout(Duration.ofMillis(1000), () -> Thread.sleep(500));
}
```

快速失败

通过 fail 方法直接使得测试失败。

```java
@Test
@DisplayName("fail")
public void shouldFail() {
    fail("This should fail");
}
```

74、单元测试-前置条件

Unit 5 中的前置条件（assumptions【假设】）类似于断言，不同之处在于不满足的**断言assertions**会使得测试方法失败，而**不满足的前置条件只会使得测试方法的执行终止。**

前置条件可以看成是测试方法执行的前提，当该前提不满足时，就没有继续执行的必要。

```java
@DisplayName("前置条件")
public class AssumptionsTest {
    private final String environment = "DEV";
    @Test
    @DisplayName("simple")
    public void simpleAssume() {
        assumeTrue(Objects.equals(this.environment, "DEV"));
        assumeFalse(() -> Objects.equals(this.environment, "PROD"));
    }
    @Test
    @DisplayName("assume then do")
    public void assumeThenDo() {
        assumingThat(
            Objects.equals(this.environment, "DEV"),
            () -> System.out.println("In DEV")
        );
    }
}
```

`assumeTrue` 和 `assumFalse` 确保给定的条件为 `true` 或 `false`，不满足条件会使得测试执行终止。

`assumingThat` 的参数是表示条件的布尔值和对应的 Executable 接口的实现对象。只有条件满足时，`Executable` 对象才会被执行；当条件不满足时，测试执行并不会终止。

75、单元测试-嵌套测试

JUnit 5 可以通过 Java 中的内部类和 `@Nested` 注解实现嵌套测试，从而可以更好的把相关的测试方法组织在一起。在内部类中可以使用 `@BeforeEach` 和 `@AfterEach` 注解，而且嵌套的层次没有限制。

```java
@DisplayName("A stack")
class TestingAStackDemo {

    Stack<Object> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested // 写在嵌套测试中的测试内容，外层的@Test方法不能驱动内层的BeforeEach、BeforeAll方法
    @DisplayName("when new")
    class WhenNew {

        @BeforeEach // 内层的方法可以驱动外层的BeforeEach
        void createNewStack() {
            stack = new Stack<>();
        }

        @Test
        @DisplayName("is empty")
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }

        @Test
        @DisplayName("throws EmptyStackException when popped")
        void throwsExceptionWhenPopped() {
            assertThrows(EmptyStackException.class, stack::pop);
        }

        @Test
        @DisplayName("throws EmptyStackException when peeked")
        void throwsExceptionWhenPeeked() {
            assertThrows(EmptyStackException.class, stack::peek);
        }

        @Nested
        @DisplayName("after pushing an element")
        class AfterPushing {

            String anElement = "an element";

            @BeforeEach
            void pushAnElement() {
                stack.push(anElement);
            }

            @Test
            @DisplayName("it is no longer empty") // 内层的方法可以驱动外层的BeforeEach
            void isNotEmpty() {
                assertFalse(stack.isEmpty());
            }

            @Test
            @DisplayName("returns the element when popped and is empty")
            void returnElementWhenPopped() {
                assertEquals(anElement, stack.pop());
                assertTrue(stack.isEmpty());
            }

            @Test
            @DisplayName("returns the element when peeked but remains not empty")
            void returnElementWhenPeeked() {
                assertEquals(anElement, stack.peek());
                assertFalse(stack.isEmpty());
            }
        }
    }
}
```

76、单元测试-参数化测试

参数化测试是JUnit5很重要的一个新特性，它使得用不同的参数多次运行测试成为了可能，也为我们的单元测试带来许多便利。

利用@ValueSource等注解，指定入参，我们将可以使用不同的参数进行多次单元测试，而不需要每新增一个参数就新增一个单元测试，省去了很多冗余代码。

利用**@ValueSource**等注解，指定入参，我们将可以使用不同的参数进行多次单元测试，而不需要每新增一个参数就新增一个单元测试，省去了很多冗余代码。

- **@ValueSource**: 为参数化测试指定入参来源，支持八大基础类以及String类型,Class类型
- **@NullSource**: 表示为参数化测试提供一个null的入参
- **@EnumSource**: 表示为参数化测试提供一个枚举入参
- **@CsvFileSource**：表示读取指定CSV文件内容作为参数化测试入参
- **@MethodSource**：表示读取指定方法的返回值作为参数化测试入参(注意方法返回需要是一个流)

当然如果参数化测试仅仅只能做到指定普通的入参还达不到让我觉得惊艳的地步。让我真正感到他的强大之处的地方在于他可以支持外部的各类入参。如:CSV,YML,JSON 文件甚至方法的返回值也可以作为入参。只需要去实现`ArgumentsProvider`接口，任何外部文件都可以作为它的入参。

```java
@ParameterizedTest
@ValueSource(strings = {"one", "two", "three"})
@DisplayName("参数化测试1")
public void parameterizedTest1(String string) {
    System.out.println(string);
    Assertions.assertTrue(StringUtils.isNotBlank(string));
}


@ParameterizedTest
@MethodSource("method")    //指定方法名
@DisplayName("方法来源参数")
public void testWithExplicitLocalMethodSource(String name) {
    System.out.println(name);
    Assertions.assertNotNull(name);
}

static Stream<String> method() { // 方法返回的必须是流，而且是静态的
    return Stream.of("apple", "banana");
}
```

迁移指南

在进行迁移的时候需要注意如下的变化：

- 注解在 `org.junit.jupiter.api` 包中，断言在 `org.junit.jupiter.api.Assertions` 类中，前置条件在 `org.junit.jupiter.api.Assumptions` 类中。
- 把 `@Before` 和 `@After` 替换成 `@BeforeEach` 和 `@AfterEach`。
- 把 `@BeforeClass` 和 `@AfterClass` 替换成 `@BeforeAll` 和@AfterAll。
- 把 `@Ignore` 替换成 `@Disabled`。
- 把 `@Category` 替换成 `@Tag`。
- 把 `@RunWith`、`@Rule` 和 `@ClassRule` 替换成 `@ExtendWith`。

77、指标监控-SpringBoot Actuator与Endpoint

使用监控器：【引入依赖】【暴露Endpoint】

未来每一个微服务在云上部署以后，我们都需要对其进行监控、追踪、审计、控制等。SpringBoot就抽取了Actuator场景，使得我们每个微服务快速引用即可获得生产级别的应用监控、审计等功能。

1.x与2.x的不同：

- SpringBoot Actuator 1.x
    - 支持SpringMVC
    - 基于继承方式进行扩展
    - 层级Metrics配置
    - 自定义Metrics收集
    - 默认较少的安全策略
- SpringBoot Actuator 2.x
    - 支持SpringMVC、JAX-RS以及Webflux
    - 注解驱动进行扩展
    - 层级&名称空间Metrics
    - 底层使用MicroMeter，强大、便捷默认丰富的安全策略

如何使用

- 添加依赖：

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

- 访问`http://localhost:8080/actuator/**`。
- 暴露所有监控信息为HTTP。

```yaml
management:
  endpoints:
    enabled-by-default: true #暴露所有端点信息
    web:
      exposure:
        include: '*'  #以web方式暴露
```

78、指标监控-常使用的端点及开启与禁用

一、常使用的端点

ID	描述
`auditevents`	暴露当前应用程序的审核事件信息。需要一个`AuditEventRepository组件`。
`beans`	显示应用程序中所有Spring Bean的完整列表。
`caches`	暴露可用的缓存。
`conditions`	显示自动配置的所有条件信息，包括匹配或不匹配的原因。
`configprops`	显示所有`@ConfigurationProperties`。
`env`	暴露Spring的属性`ConfigurableEnvironment`
`flyway`	显示已应用的所有Flyway数据库迁移。 需要一个或多个`Flyway`组件。
`health`	显示应用程序运行状况信息。
`httptrace`	显示HTTP跟踪信息（默认情况下，最近100个HTTP请求-响应）。需要一个`HttpTraceRepository`组件。
`info`	显示应用程序信息。
`integrationgraph`	显示Spring `integrationgraph` 。需要依赖`spring-integration-core`。
`loggers`	显示和修改应用程序中日志的配置。
`liquibase`	显示已应用的所有Liquibase数据库迁移。需要一个或多个`Liquibase`组件。
`metrics`	显示当前应用程序的"指标"信息。
`mappings`	显示所有`@RequestMapping`路径列表。
`scheduledtasks`	显示应用程序中的计划任务。
`sessions`	允许从Spring Session支持的会话存储中检索和删除用户会话。需要使用Spring Session的基于Servlet的Web应用程序。
`shutdown`	使应用程序正常关闭。默认禁用。
`startup`	显示由`ApplicationStartup`收集的启动步骤数据。需要使用`SpringApplication`进行配置`BufferingApplicationStartup`。
`threaddump`	执行线程转储。

如果您的应用程序是Web应用程序（Spring MVC，Spring WebFlux或Jersey），则可以使用以下附加端点：

ID	描述
`heapdump`	返回`hprof`堆转储文件。
`jolokia`	通过HTTP暴露JMX bean（需要引入Jolokia，不适用于WebFlux）。需要引入依赖`jolokia-core`。
`logfile`	返回日志文件的内容（如果已设置`logging.file.name`或`logging.file.path`属性）。支持使用HTTP `Range`标头来检索部分日志文件的内容。
`prometheus`	以Prometheus服务器可以抓取的格式公开指标。需要依赖`micrometer-registry-prometheus`。

其中最常用的Endpoint：

- **Health：监控状况**
- **Metrics：运行时指标**
- **Loggers：日志记录**

二、Health Endpoint

健康检查端点，我们一般用于在云平台，平台会定时的检查应用的健康状况，我们就需要Health Endpoint可以为平台返回当前应用的一系列组件健康状况的集合。

重要的几点：

- health endpoint返回的结果，应该是一系列健康检查后的一个汇总报告。
- 很多的健康检查默认已经自动配置好了，比如：数据库、redis等。
- 可以很容易的添加自定义的健康检查机制。

三、Metrics Endpoint

提供详细的、层级的、空间指标信息，这些信息可以被pull（主动推送）或者push（被动获取）方式得到：

- 通过Metrics对接多种监控系统。
- 简化核心Metrics开发。
- 添加自定义Metrics或者扩展已有Metrics。

四、开启与禁用Endpoints

- 默认所有的Endpoint除过shutdown都是开启的。
- 需要开启或者禁用某个Endpoint。配置模式为 `management.endpoint.<endpointName>.enabled = true`

```yaml
management:
  endpoint:
    beans:
      enabled: true
```

- 或者禁用所有的Endpoint然后手动开启指定的Endpoint。

```yaml
management:
  endpoints:
    enabled-by-default: false
  endpoint:
    beans:
      enabled: true
    health:
      enabled: true
```

五、暴露Endpoints

支持的暴露方式

- HTTP：默认只暴露health和info。
- JMX：默认暴露所有Endpoint。
- 除过health和info，剩下的Endpoint都应该进行保护访问。如果引入Spring Security，则会默认配置安全访问规则。

ID	JMX	Web
`auditevents`	Yes	No
`beans`	Yes	No
`caches`	Yes	No
`conditions`	Yes	No
`configprops`	Yes	No
`env`	Yes	No
`flyway`	Yes	No
`health`	Yes	Yes
`heapdump`	N/A	No
`httptrace`	Yes	No
`info`	Yes	Yes
`integrationgraph`	Yes	No
`jolokia`	N/A	No
`logfile`	N/A	No
`loggers`	Yes	No
`liquibase`	Yes	No
`metrics`	Yes	No
`mappings`	Yes	No
`prometheus`	N/A	No
`scheduledtasks`	Yes	No
`sessions`	Yes	No
`shutdown`	Yes	No
`startup`	Yes	No
`threaddump`	Yes	No

若要更改公开的Endpoint，请配置以下的包含和排除属性：

Property	Default
`management.endpoints.jmx.exposure.exclude`	
`management.endpoints.jmx.exposure.include`	`*`
`management.endpoints.web.exposure.exclude`	
`management.endpoints.web.exposure.include`	`info, health`

79、指标监控-定制Endpoint

一、定制 Health 信息的两种方法

```yaml
management:
    health:
      enabled: true
      show-details: always #总是显示详细信息。可显示每个模块的状态信息
```

【实现接口】通过实现`HealthIndicator`接口

```java
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }

}

/*
构建Health
Health build = Health.down()
                .withDetail("msg", "error service")
                .withDetail("code", "500")
                .withException(new RuntimeException())
                .build();
*/
```

【继承抽象类】通过继承`AbstractHealthIndicator`抽象类

```java
@Component
public class MyComHealthIndicator extends AbstractHealthIndicator {

    /**
     * 真实的检查方法
     * @param builder
     * @throws Exception
     */
    @Override
    protected void doHealthCheck(Health.Builder builder) throws Exception {
        //mongodb。   获取连接进行测试
        Map<String,Object> map = new HashMap<>();
        // 检查完成
        if(1 == 2){
//              builder.up(); //健康
            builder.status(Status.UP);
            map.put("count",1);
            map.put("ms",100);
        }else {
//              builder.down();
            builder.status(Status.OUT_OF_SERVICE);
            map.put("err","连接超时");
            map.put("ms",3000);
        }


        builder.withDetail("code",100)
                .withDetails(map);

    }
}
```

二、定制info信息

常用两种方式：

- 编写配置文件

```yaml
info:
  appName: boot-admin
  version: 2.0.1
  mavenProjectName: @project.artifactId@  #使用@@可以获取maven的pom文件值
  mavenProjectVersion: @project.version@
```

- 编写InfoContributor

```java
import java.util.Collections;

import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class ExampleInfoContributor implements InfoContributor {

    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("example",
                Collections.singletonMap("key", "value"));
    }

}
```

会输出以上方式返回的所有info信息

三、定制Metrics信息

增加定制Metrics：

```
class MyService{
    Counter counter;
    public MyService(MeterRegistry meterRegistry){
         counter = meterRegistry.counter("myservice.method.running.counter");
    }

    public void hello() {
        counter.increment();
    }
}
```

```
//也可以使用下面的方式
@Bean
MeterBinder queueSize(Queue queue) {
    return (registry) -> Gauge.builder("queueSize", queue::size).register(registry);
}
```

四、定制Endpoint

```
@Component
@Endpoint(id = "container")
public class DockerEndpoint {

    @ReadOperation
    public Map getDockerInfo(){
        return Collections.singletonMap("info","docker started...");
    }

    @WriteOperation
    private void restartDocker(){
        System.out.println("docker restarted....");
    }

}
```

场景：

- 开发ReadinessEndpoint来管理程序是否就绪。
- 开发LivenessEndpoint来管理程序是否存活。

80、指标监控-Boot Admin Server可视化

可视化指标监控

> What is Spring Boot Admin?

> codecentric’s Spring Boot Admin is a community project to manage and monitor your Spring Boot ® applications. The applications register with our Spring Boot Admin Client (via HTTP) or are discovered using Spring Cloud ® (e.g. Eureka, Consul). The UI is just a Vue.js application on top of the Spring Boot Actuator endpoints.

一、引入依赖pom.xml

```xml
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

在配置文件中配置选项application.properties

```
spring.boot.admin.client.url=http://localhost:8080
management.endpoints.web.exposure.include=*
```

二、添加对应的注解

```java
@Configuration
@EnableAutoConfiguration
@EnableAdminServer
public class SpringBootAdminApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootAdminApplication.class, args);
    }
}
```

81、高级特性-Profile环境切换

为了方便多环境适配，Spring Boot简化了profile功能。可以通过该功能选定某一个环境运行项目。

- 默认配置文件`application.yaml`任何时候都会加载。
- 指定环境配置文件`application-{env}.yaml`，`env`通常替代为`test`，
- 激活指定环境
  - 配置文件激活：`spring.profiles.active=prod`
  - 命令行激活：`java -jar xxx.jar --spring.profiles.active=prod  --person.name=haha`（修改配置文件的任意值，**命令行优先**）
- 默认配置与环境配置同时生效
- 同名配置项，profile配置优先

@Profile条件装配功能

```java
@Data
@Component
@ConfigurationProperties("person")//在配置文件中配置
public class Person{
    private String name;
    private Integer age;
}
```

application.properties

```yaml
person:
  name: lun
  age: 8
```

```java
public interface Person {

   String getName();
   Integer getAge();

}

@Profile("test")//加载application-test.yaml里的
@Component
@ConfigurationProperties("person")
@Data
public class Worker implements Person {

    private String name;
    private Integer age;
}

@Profile(value = {"prod","default"})//加载application-prod.yaml里的
@Component
@ConfigurationProperties("person")
@Data
public class Boss implements Person {

    private String name;
    private Integer age;

```

application-test.yaml

```yaml
person:
  name: test-张三

server:
  port: 7000
```

application-prod.yaml

```yaml
person:
  name: prod-张三

server:
  port: 8000
```

application.properties

```properties
# 激活prod配置文件
spring.profiles.active=prod
```

```java
@Autowired
private Person person;

@GetMapping("/")
public String hello(){
    //激活了prod，则返回Boss；激活了test，则返回Worker
    return person.getClass().toString();
}
```

@Profile还可以修饰在方法上：

```java
class Color {
}

@Configuration
public class MyConfig {

    @Profile("prod")
    @Bean
    public Color red(){
        return new Color();
    }

    @Profile("test")
    @Bean
    public Color green(){
        return new Color();
    }
}
```

可以激活一组：

```properties
spring.profiles.active=production

spring.profiles.group.production[0]=proddb
spring.profiles.group.production[1]=prodmq
```

82、高级特性-配置加载优先级

外部化配置

Spring Boot uses a very particular `PropertySource` order that is designed to allow sensible overriding of values. Properties are considered in the following order (with values from lower items overriding earlier ones)（1优先级最低，14优先级最高）：

1. Default properties (specified by setting `SpringApplication.setDefaultProperties`).
2. [`@PropertySource`]() annotations on your `@Configuration` classes. Please note that such property sources are not added to the `Environment` until the application context is being refreshed. This is too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.
3. Config data (such as `application.properties` files)
4. A `RandomValuePropertySource` that has properties only in `random.*`.
5. OS environment variables.
6. Java System properties (`System.getProperties()`).
7. JNDI attributes from `java:comp/env`.
8. `ServletContext` init parameters.
9. `ServletConfig` init parameters.
10. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. `properties` attribute on your tests. Available on [`@SpringBootTest`]() and the [test annotations for testing a particular slice of your application]().
13. [`@TestPropertySource`]() annotations on your tests.
14. [Devtools global settings properties]() in the `$HOME/.config/spring-boot` directory when devtools is active.

```java
import org.springframework.stereotype.*;
import org.springframework.beans.factory.annotation.*;

@Component
public class MyBean {

    @Value("${name}")//以这种方式可以获得配置值
    private String name;

    // ...

}
```

- 外部配置源
  - Java属性文件。**.properties**
  - YAML文件。
  - 环境变量。
  - 命令行参数。
- 配置文件查找位置【后面的会覆盖前面的内容】
  1. classpath 根路径。
  2. classpath 根路径下config目录。
  3. jar包当前目录。
  4. jar包当前目录的config目录。
  5. /config子目录的直接子目录。
- 配置文件加载顺序：【后面的会覆盖前面的内容】
  1. 当前jar包内部的`application.properties`和`application.yml`。
  2. 当前jar包内部的`application-{profile}.properties` 和 `application-{profile}.yml`。

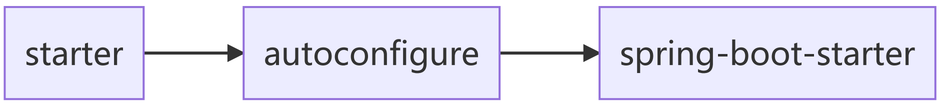
3. 引用的外部jar包的`application.properties`和`application.yml`。
4. 引用的外部jar包的`application-{profile}.properties`和`application-{profile}.yml`。

- 指定环境优先，外部优先，后面的可以覆盖前面的同名配置项。

83、高级特性-自定义starter细节

一、starter启动原理

- starter的pom.xml引入autoconfigure依赖


$$\begin{tikzcd} \boxed{\text{starter}} \arrow[r] & \boxed{\text{autoconfigure}} \arrow[r] & \boxed{\text{spring-boot-starter}} \end{tikzcd}$$

- autoconfigure包中配置使用`META-INF/spring.factories`中`EnableAutoConfiguration`的值，使得项目启动加载指定的自动配置类
- 编写自动配置类 `xxxAutoConfiguration` -> `xxxxProperties`
-
  - `@Configuration`
  - `@Conditional`
  - `@EnableConfigurationProperties`
  - `@Bean`
  - ......
- 引入starter --- `xxxAutoConfiguration` --- 容器中放入组件 ---- `绑定xxxProperties` ---- 配置项

自定义starter

- 目标：创建`HelloService`的自定义starter。
- 创建两个工程，分别命名为`hello-spring-boot-starter`（普通Maven工程），`hello-spring-boot-starter-autoconfigure`（需用用到Spring Initializr创建的Maven工程）。
- `hello-spring-boot-starter`无需编写什么代码，只需让该工程引入`hello-spring-boot-starter-autoconfigure`依赖：

  【hello-spring-boot-starter的pom.xml】

```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>

    <groupId>com.lun</groupId>
    <artifactId>hello-spring-boot-starter</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>com.lun</groupId>
            <artifactId>hello-spring-boot-starter-autoconfigure</artifactId>
            <version>1.0.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

</project>
```

- `hello-spring-boot-starter-autoconfigure`的pom.xml如下：

```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>2.4.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.lun</groupId>
    <artifactId>hello-spring-boot-starter-autoconfigure</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>hello-spring-boot-starter-autoconfigure</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</artifactId>
        </dependency>
    </dependencies>
</project>
```

- 创建4个文件：
    - `com/lun/hello/auto/HelloServiceAutoConfiguration`
    - `com/lun/hello/bean/HelloProperties`
    - `com/lun/hello/service/HelloService`
    - `src/main/resources/META-INF/spring.factories`

```java
import com.lun.hello.bean.HelloProperties;
import com.lun.hello.service.HelloService;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@ConditionalOnMissingBean(HelloService.class)
@EnableConfigurationProperties(HelloProperties.class)//默认HelloProperties放在容器中
public class HelloServiceAutoConfiguration {

    @Bean
    public HelloService helloService(){
        return new HelloService();
    }

}
```

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

@ConfigurationProperties("hello")
public class HelloProperties {
    private String prefix;
    private String suffix;

    public String getPrefix() {
        return prefix;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public String getSuffix() {
        return suffix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
}

```

```java
import com.lun.hello.bean.HelloProperties;
import org.springframework.beans.factory.annotation.Autowired;


/**
 * 默认不要放在容器中
 */
public class HelloService {

    @Autowired
    private HelloProperties helloProperties;

    public String sayHello(String userName){
        return helloProperties.getPrefix() + ": " + userName + " > " +
helloProperties.getSuffix();
    }
}
```

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.lun.hello.auto.HelloServiceAutoConfiguration
```

- 用maven插件，将两工程install到本地。
- 接下来，测试使用自定义starter，用Spring Initializr创建名为`hello-spring-boot-starter-test`工程，引入`hello-spring-boot-starter`依赖，其pom.xml如下：

```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"
```

```xml
         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>2.4.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.lun</groupId>
    <artifactId>hello-spring-boot-starter-test</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>hello-spring-boot-starter-test</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</artifactId>
        </dependency>

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

        <!-- 引入`hello-spring-boot-starter`依赖 -->
        <dependency>
            <groupId>com.lun</groupId>
            <artifactId>hello-spring-boot-starter</artifactId>
            <version>1.0.0-SNAPSHOT</version>
        </dependency>

    </dependencies>

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

</project>

```

- 添加配置文件`application.properties`：

```properties
hello.prefix=hello
hello.suffix=666
```

- 添加单元测试类：

```java
import com.lun.hello.service.HelloService;//来自自定义starter
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class HelloSpringBootStarterTestApplicationTests {

    @Autowired
    private HelloService helloService;

    @Test
    void contextLoads() {
        // System.out.println(helloService.sayHello("lun"));
        Assertions.assertEquals("hello: lun > 666", helloService.sayHello("lun"));
    }

}
```

84、原理解析-SpringApplication创建初始化流程

SpringBoot启动过程

Spring Boot应用的启动类:

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

@SpringBootApplication
public class HelloSpringBootStarterTestApplication {

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

}
```

```java
public class SpringApplication {

    ...

    public static ConfigurableApplicationContext run(Class<?> primarySource, String... args) {
        return run(new Class<?>[] { primarySource }, args);
    }

    public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args) {
        return new SpringApplication(primarySources).run(args);
    }

    //先看看new SpringApplication(primarySources),下一节再看看run()
    public SpringApplication(Class<?>... primarySources) {
        this(null, primarySources);
    }

    public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
        this.resourceLoader = resourceLoader;
        Assert.notNull(primarySources, "PrimarySources must not be null");
        this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
        //WebApplicationType是枚举类，有NONE,SERVLET,REACTIVE,下行webApplicationType是SERVLET
        this.webApplicationType = WebApplicationType.deduceFromClasspath();

        //初始启动引导器，去spring.factories文件中找org.springframework.boot.Bootstrapper，但我找不到实现Bootstrapper接口的类
        this.bootstrappers = new ArrayList<>(getSpringFactoriesInstances(Bootstrapper.class));

        //去spring.factories找 ApplicationContextInitializer
        setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));

        //去spring.factories找 ApplicationListener
        setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));

        this.mainApplicationClass = deduceMainApplicationClass();
    }

    private Class<?> deduceMainApplicationClass() {
        try {
            StackTraceElement[] stackTrace = new RuntimeException().getStackTrace();
            for (StackTraceElement stackTraceElement : stackTrace) {
                if ("main".equals(stackTraceElement.getMethodName())) {
                    return Class.forName(stackTraceElement.getClassName());
                }
            }
        }
        catch (ClassNotFoundException ex) {
            // Swallow and continue
        }
        return null;
    }

    ...

}
```

spring.factories:

```
...

# Application Context Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer,\
org.springframework.boot.context.ContextIdApplicationContextInitializer,\
org.springframework.boot.context.config.DelegatingApplicationContextInitializer,\
org.springframework.boot.rsocket.context.RSocketPortInfoApplicationContextInitializer,\
org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer

# Application Listeners
```

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.ClearCachesApplicationListener,\
org.springframework.boot.builder.ParentContextCloserApplicationListener,\
org.springframework.boot.context.FileEncodingApplicationListener,\
org.springframework.boot.context.config.AnsiOutputApplicationListener,\
org.springframework.boot.context.config.DelegatingApplicationListener,\
org.springframework.boot.context.logging.LoggingApplicationListener,\
org.springframework.boot.env.EnvironmentPostProcessorApplicationListener,\
org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener

...

```

85、原理解析-SpringBoot完整启动过程

继续上一节，接着讨论`return new SpringApplication(primarySources).run(args)`的`run`方法

```java
public class SpringApplication {

    ...

    public ConfigurableApplicationContext run(String... args) {
        StopWatch stopWatch = new StopWatch();//开始计时器
        stopWatch.start();//开始计时

        //1.
        //创建引导上下文（Context环境）createBootstrapContext()
        //获取到所有之前的 bootstrappers 挨个执行 intitialize() 来完成对引导启动器上下文环境设置
        DefaultBootstrapContext bootstrapContext = createBootstrapContext();

        //2.到最后该方法会返回这context
        ConfigurableApplicationContext context = null;

        //3.让当前应用进入headless模式
        configureHeadlessProperty();

        //4.获取所有 RunListener（运行监听器）,为了方便所有Listener进行事件感知
        SpringApplicationRunListeners listeners = getRunListeners(args);

        //5. 遍历 SpringApplicationRunListener 调用 starting 方法;
        // 相当于通知所有感兴趣系统正在启动过程的人，项目正在 starting。
        listeners.starting(bootstrapContext, this.mainApplicationClass);
        try {
            //6.保存命令行参数 ApplicationArguments
            ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);

            //7.准备环境
            ConfigurableEnvironment environment = prepareEnvironment(listeners, bootstrapContext, applicationArguments);
            configureIgnoreBeanInfo(environment);

            /*打印标志
              .   ____          _            __ _ _
```

```java
             /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
            ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
             \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
              '  |____| .__|_| |_|_| |_\__, | / / / /
             =========|_|==============|___/=/_/_/_/
             :: Spring Boot ::                (v2.4.2)
            */
            Banner printedBanner = printBanner(environment);

            // 创建IOC容器（createApplicationContext（））
            // 根据项目类型webApplicationType（NONE,SERVLET,REACTIVE）创建容器，
            // 当前会创建 AnnotationConfigServletWebServerApplicationContext
            context = createApplicationContext();
            context.setApplicationStartup(this.applicationStartup);

            //8.准备ApplicationContext IOC容器的基本信息
            prepareContext(bootstrapContext, context, environment, listeners, applicationArguments, printedBanner);
            //9.刷新IOC容器,创建容器中的所有组件,Spring框架的内容
            refreshContext(context);
            //该方法没内容，大概为将来填入
            afterRefresh(context, applicationArguments);
            stopWatch.stop();//停止计时
            if (this.logStartupInfo) {//this.logStartupInfo默认是true
                new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);
            }
            //10.
            listeners.started(context);

            //11.调用所有runners
            callRunners(context, applicationArguments);
        }
        catch (Throwable ex) {
            //13.
            handleRunFailure(context, ex, listeners);
            throw new IllegalStateException(ex);
        }

        try {
            //12.
            listeners.running(context);
        }
        catch (Throwable ex) {
            //13.
            handleRunFailure(context, ex, null);
            throw new IllegalStateException(ex);
        }
        return context;
    }

    //1.
    private DefaultBootstrapContext createBootstrapContext() {
        DefaultBootstrapContext bootstrapContext = new DefaultBootstrapContext();
        this.bootstrappers.forEach((initializer) -> initializer.intitialize(bootstrapContext));
        return bootstrapContext;
    }

    //3.
    private void configureHeadlessProperty() {
        //this.headless默认为true
        System.setProperty(SYSTEM_PROPERTY_JAVA_AWT_HEADLESS,
                System.getProperty(SYSTEM_PROPERTY_JAVA_AWT_HEADLESS,
Boolean.toString(this.headless)));
    }

    private static final String SYSTEM_PROPERTY_JAVA_AWT_HEADLESS = "java.awt.headless";

    //4.
    private SpringApplicationRunListeners getRunListeners(String[] args) {
        Class<?>[] types = new Class<?>[] { SpringApplication.class, String[].class };
        //getSpringFactoriesInstances 去 spring.factories 找 SpringApplicationRunListener
        return new SpringApplicationRunListeners(logger,
                getSpringFactoriesInstances(SpringApplicationRunListener.class, types,
this, args),
                this.applicationStartup);
    }

    //7.准备环境
    private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners
listeners,
            DefaultBootstrapContext bootstrapContext, ApplicationArguments
applicationArguments) {
        // Create and configure the environment
        //返回或者创建基础环境信息对象，如：StandardServletEnvironment,
StandardReactiveWebEnvironment
        ConfigurableEnvironment environment = getOrCreateEnvironment();
        //配置环境信息对象,读取所有的配置源的配置属性值。
        configureEnvironment(environment, applicationArguments.getSourceArgs());
        //绑定环境信息
        ConfigurationPropertySources.attach(environment);
        //7.1 通知所有的监听器当前环境准备完成
        listeners.environmentPrepared(bootstrapContext, environment);
        DefaultPropertiesPropertySource.moveToEnd(environment);
        configureAdditionalProfiles(environment);
        bindToSpringApplication(environment);
        if (!this.isCustomEnvironment) {
            environment = new
EnvironmentConverter(getClassLoader()).convertEnvironmentIfNecessary(environment,
                    deduceEnvironmentClass());
        }
        ConfigurationPropertySources.attach(environment);
        return environment;
    }

    //8.
    private void prepareContext(DefaultBootstrapContext bootstrapContext,
ConfigurableApplicationContext context,
            ConfigurableEnvironment environment, SpringApplicationRunListeners listeners,
            ApplicationArguments applicationArguments, Banner printedBanner) {
        //保存环境信息
        context.setEnvironment(environment);
        //IOC容器的后置处理流程
        postProcessApplicationContext(context);
        //应用初始化器
        applyInitializers(context);
        //8.1 遍历所有的 listener 调用 contextPrepared。
        //EventPublishRunListenr通知所有的监听器contextPrepared
        listeners.contextPrepared(context);
        bootstrapContext.close(context);
        if (this.logStartupInfo) {
            logStartupInfo(context.getParent() == null);
            logStartupProfileInfo(context);
        }
        // Add boot specific singleton beans
        ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
        beanFactory.registerSingleton("springApplicationArguments",
applicationArguments);
        if (printedBanner != null) {
            beanFactory.registerSingleton("springBootBanner", printedBanner);
        }
        if (beanFactory instanceof DefaultListableBeanFactory) {
            ((DefaultListableBeanFactory) beanFactory)

.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
        }
        if (this.lazyInitialization) {
            context.addBeanFactoryPostProcessor(new
LazyInitializationBeanFactoryPostProcessor());
        }
        // Load the sources
        Set<Object> sources = getAllSources();
        Assert.notEmpty(sources, "Sources must not be empty");
        load(context, sources.toArray(new Object[0]));
        //8.2
        listeners.contextLoaded(context);
    }

    //11.调用所有runners
    private void callRunners(ApplicationContext context, ApplicationArguments args) {
        List<Object> runners = new ArrayList<>();

        //获取容器中的 ApplicationRunner
        runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
        //获取容器中的  CommandLineRunner
        runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
        //合并所有runner并且按照@Order进行排序
        AnnotationAwareOrderComparator.sort(runners);
        //遍历所有的runner。调用 run 方法
        for (Object runner : new LinkedHashSet<>(runners)) {
            if (runner instanceof ApplicationRunner) {
                callRunner((ApplicationRunner) runner, args);
            }
            if (runner instanceof CommandLineRunner) {
                callRunner((CommandLineRunner) runner, args);
            }
        }
    }

    //13.
    private void handleRunFailure(ConfigurableApplicationContext context, Throwable
exception,
            SpringApplicationRunListeners listeners) {
        try {
            try {
                handleExitCode(context, exception);
                if (listeners != null) {
                    //14.
                    listeners.failed(context, exception);
                }
            }
            finally {
                reportFailure(getExceptionReporters(context), exception);
                if (context != null) {
                    context.close();
                }
            }
        }
        catch (Exception ex) {
            logger.warn("Unable to close ApplicationContext", ex);
        }
        ReflectionUtils.rethrowRuntimeException(exception);
    }

    ...
}
```

```java
//2. new SpringApplication(primarySources).run(args) 最后返回的接口类型
public interface ConfigurableApplicationContext extends ApplicationContext, Lifecycle, Closeable {
    String CONFIG_LOCATION_DELIMITERS = ",; \t\n";
    String CONVERSION_SERVICE_BEAN_NAME = "conversionService";
    String LOAD_TIME_WEAVER_BEAN_NAME = "loadTimeWeaver";
    String ENVIRONMENT_BEAN_NAME = "environment";
    String SYSTEM_PROPERTIES_BEAN_NAME = "systemProperties";
    String SYSTEM_ENVIRONMENT_BEAN_NAME = "systemEnvironment";
    String APPLICATION_STARTUP_BEAN_NAME = "applicationStartup";
    String SHUTDOWN_HOOK_THREAD_NAME = "SpringContextShutdownHook";

    void setId(String var1);

    void setParent(@Nullable ApplicationContext var1);

    void setEnvironment(ConfigurableEnvironment var1);

    ConfigurableEnvironment getEnvironment();

    void setApplicationStartup(ApplicationStartup var1);

    ApplicationStartup getApplicationStartup();

    void addBeanFactoryPostProcessor(BeanFactoryPostProcessor var1);

    void addApplicationListener(ApplicationListener<?> var1);

    void setClassLoader(ClassLoader var1);

    void addProtocolResolver(ProtocolResolver var1);

    void refresh() throws BeansException, IllegalStateException;

    void registerShutdownHook();

    void close();

    boolean isActive();

    ConfigurableListableBeanFactory getBeanFactory() throws IllegalStateException;
}
```

```properties
#4.
#spring.factories
# Run Listeners
org.springframework.boot.SpringApplicationRunListener=\
org.springframework.boot.context.event.EventPublishingRunListener
```

```java
class SpringApplicationRunListeners {

    private final Log log;

    private final List<SpringApplicationRunListener> listeners;

    private final ApplicationStartup applicationStartup;

    SpringApplicationRunListeners(Log log, Collection<? extends SpringApplicationRunListener> listeners,
            ApplicationStartup applicationStartup) {
        this.log = log;
        this.listeners = new ArrayList<>(listeners);
        this.applicationStartup = applicationStartup;
    }

    //5.遍历 SpringApplicationRunListener 调用 starting 方法;
    //相当于通知所有感兴趣系统正在启动过程的人，项目正在 starting。
    void starting(ConfigurableBootstrapContext bootstrapContext, Class<?> mainApplicationClass) {
        doWithListeners("spring.boot.application.starting", (listener) -> listener.starting(bootstrapContext),
                (step) -> {
                    if (mainApplicationClass != null) {
                        step.tag("mainApplicationClass", mainApplicationClass.getName());
                    }
                });
    }

    //7.1
    void environmentPrepared(ConfigurableBootstrapContext bootstrapContext, ConfigurableEnvironment environment) {
        doWithListeners("spring.boot.application.environment-prepared",
                (listener) -> listener.environmentPrepared(bootstrapContext, environment));
    }

    //8.1
    void contextPrepared(ConfigurableApplicationContext context) {
        doWithListeners("spring.boot.application.context-prepared", (listener) -> listener.contextPrepared(context));
    }

    //8.2
    void contextLoaded(ConfigurableApplicationContext context) {
        doWithListeners("spring.boot.application.context-loaded", (listener) -> listener.contextLoaded(context));
    }


    //10.
    void started(ConfigurableApplicationContext context) {
        doWithListeners("spring.boot.application.started", (listener) -> listener.started(context));
    }

    //12.
    void running(ConfigurableApplicationContext context) {
        doWithListeners("spring.boot.application.running", (listener) -> listener.running(context));
    }

    //14.
    void failed(ConfigurableApplicationContext context, Throwable exception) {
        doWithListeners("spring.boot.application.failed",
                (listener) -> callFailedListener(listener, context, exception), (step) -> {
                    step.tag("exception", exception.getClass().toString());
                    step.tag("message", exception.getMessage());
                });
    }

    private void doWithListeners(String stepName, Consumer<SpringApplicationRunListener> listenerAction,
            Consumer<StartupStep> stepAction) {
        StartupStep step = this.applicationStartup.start(stepName);
        this.listeners.forEach(listenerAction);
        if (stepAction != null) {
            stepAction.accept(step);
        }
        step.end();
    }

    ...

}
```

86、原理解析-自定义事件监听组件

`MyApplicationContextInitializer.java`

```java
import org.springframework.context.ApplicationContextInitializer;
import org.springframework.context.ConfigurableApplicationContext;

public class MyApplicationContextInitializer implements ApplicationContextInitializer {
    @Override
    public void initialize(ConfigurableApplicationContext applicationContext) {
        System.out.println("MyApplicationContextInitializer ....initialize.... ");
    }
}
```

`MyApplicationListener.java`

```java
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

public class MyApplicationListener implements ApplicationListener {
    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        System.out.println("MyApplicationListener.....onApplicationEvent...");
    }
}
```

`MyApplicationRunner.java`

```java
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;


@Order(1)
@Component//放入容器
public class MyApplicationRunner implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("MyApplicationRunner...run...");
    }
}
```

`MyCommandLineRunner.java`

```java
import org.springframework.boot.CommandLineRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
/**
 * 应用启动做一个一次性事情
 */
@Order(2)
@Component//放入容器
public class MyCommandLineRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("MyCommandLineRunner....run....");
    }
}
```

`MySpringApplicationRunListener.java`

```java
import org.springframework.boot.ConfigurableBootstrapContext;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.SpringApplicationRunListener;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.core.env.ConfigurableEnvironment;

```

```java
public class MySpringApplicationRunListener implements SpringApplicationRunListener {

    private SpringApplication application;
    public MySpringApplicationRunListener(SpringApplication application, String[] args){
        this.application = application;
    }

    @Override
    public void starting(ConfigurableBootstrapContext bootstrapContext) {
        System.out.println("MySpringApplicationRunListener....starting....");

    }


    @Override
    public void environmentPrepared(ConfigurableBootstrapContext bootstrapContext, ConfigurableEnvironment environment) {
        System.out.println("MySpringApplicationRunListener....environmentPrepared....");
    }


    @Override
    public void contextPrepared(ConfigurableApplicationContext context) {
        System.out.println("MySpringApplicationRunListener....contextPrepared....");

    }

    @Override
    public void contextLoaded(ConfigurableApplicationContext context) {
        System.out.println("MySpringApplicationRunListener....contextLoaded....");
    }

    @Override
    public void started(ConfigurableApplicationContext context) {
        System.out.println("MySpringApplicationRunListener....started....");
    }

    @Override
    public void running(ConfigurableApplicationContext context) {
        System.out.println("MySpringApplicationRunListener....running....");
    }

    @Override
    public void failed(ConfigurableApplicationContext context, Throwable exception) {
        System.out.println("MySpringApplicationRunListener....failed....");
    }
}
```

注册`MyApplicationContextInitializer`，`MyApplicationListener`，`MySpringApplicationRunListener`:

`resources / META-INF / spring.factories`:

```
org.springframework.context.ApplicationContextInitializer=\
  com.lun.boot.listener.MyApplicationContextInitializer

org.springframework.context.ApplicationListener=\
  com.lun.boot.listener.MyApplicationListener

org.springframework.boot.SpringApplicationRunListener=\
  com.lun.boot.listener.MySpringApplicationRunListener
```