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

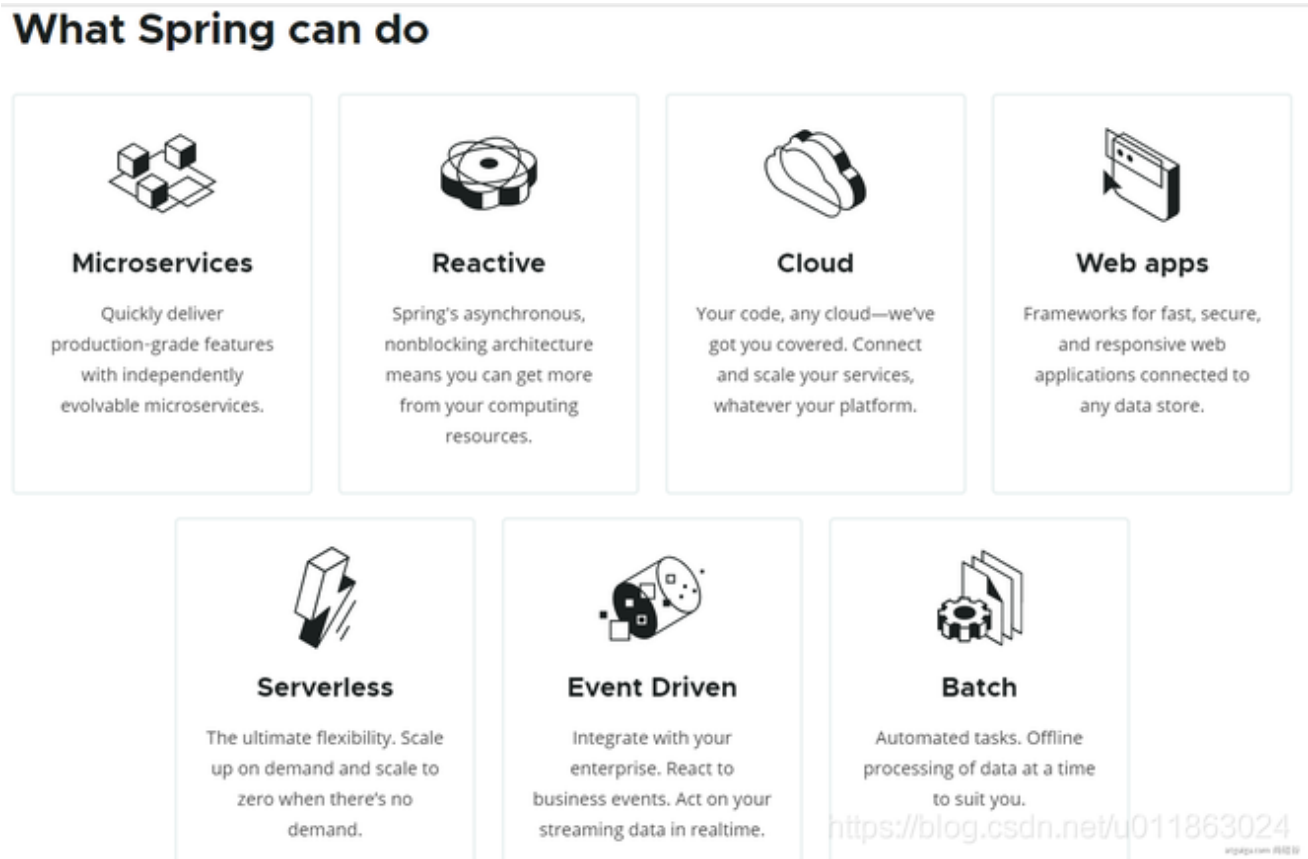
- 学习要求 -熟悉Spring基础 -熟悉Maven使用
- 环境要求
  - Java8及以上
  - Maven 3.3及以上
- 学习资料
  - Spring Boot官网
  - Spring Boot官方文档
  - 本课程文档地址
  - 视频地址1、视频地址2
  - 源码地址

02、基础入门-Spring生态圈

Spring官网

Spring能做什么

Spring的能力



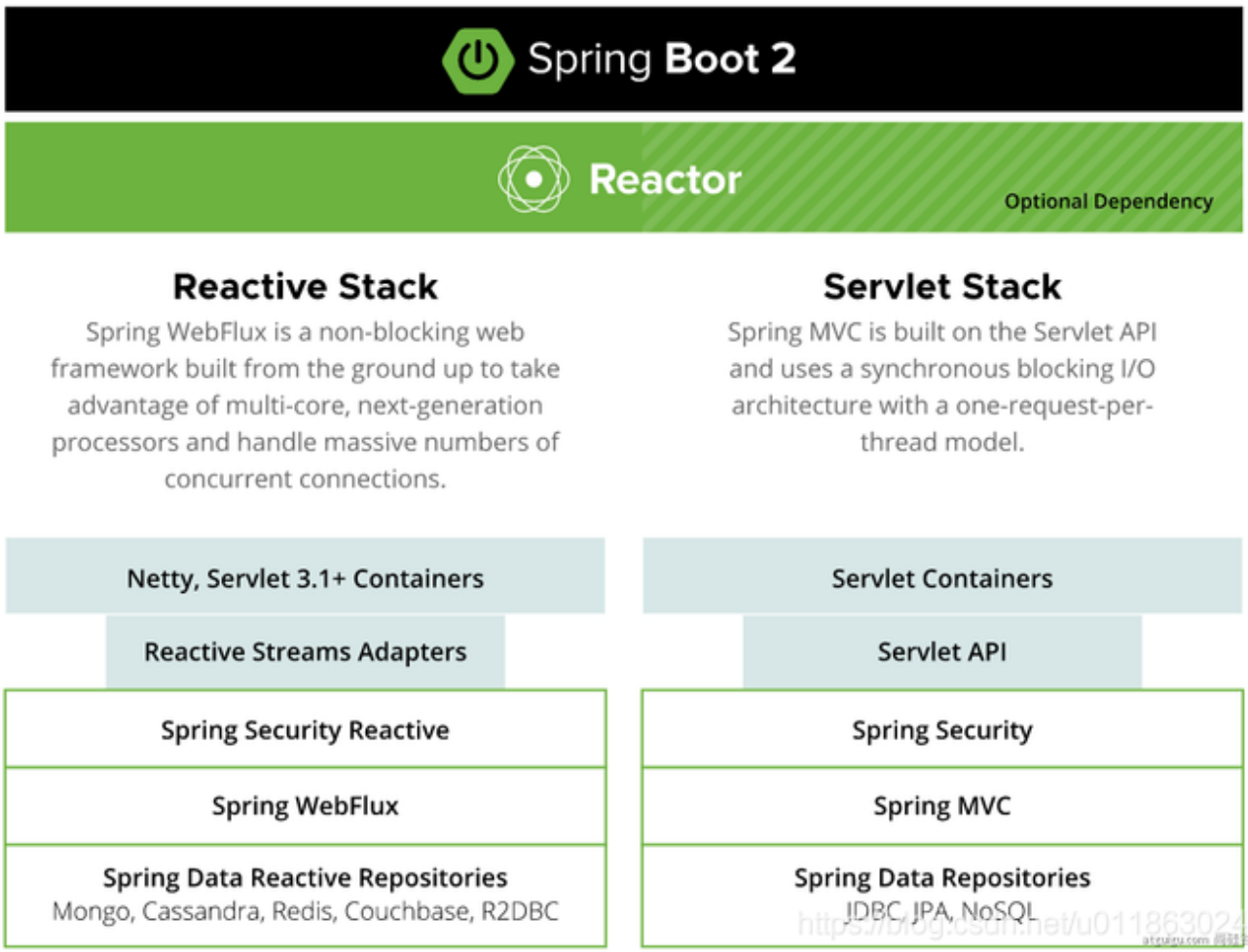
Spring的生态

覆盖了：

- 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
- .....

Spring5重大升级

- 响应式编程



- 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。

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

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

分布式

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

云原生

原生应用如何上云。 Cloud Native

上云的困难

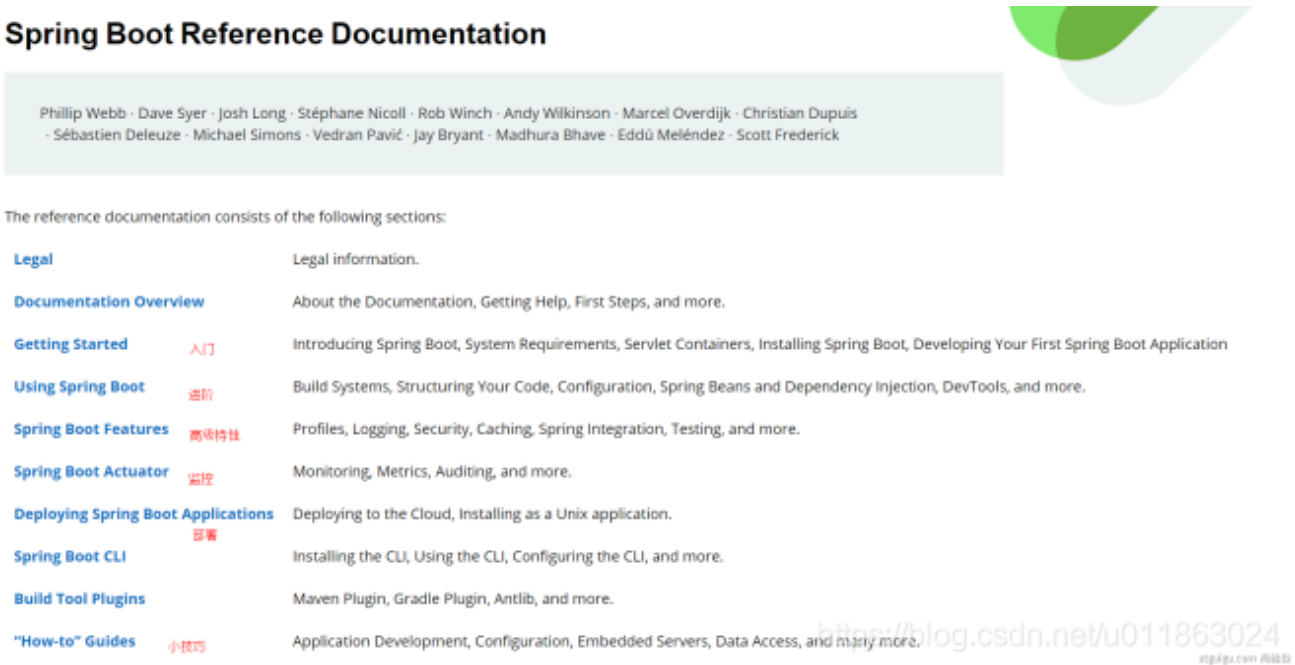
- 服务自愈
- 弹性伸缩
- 服务隔离
- 自动化部署
- 灰度发布
- 流量治理
- ……

上云的解决

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

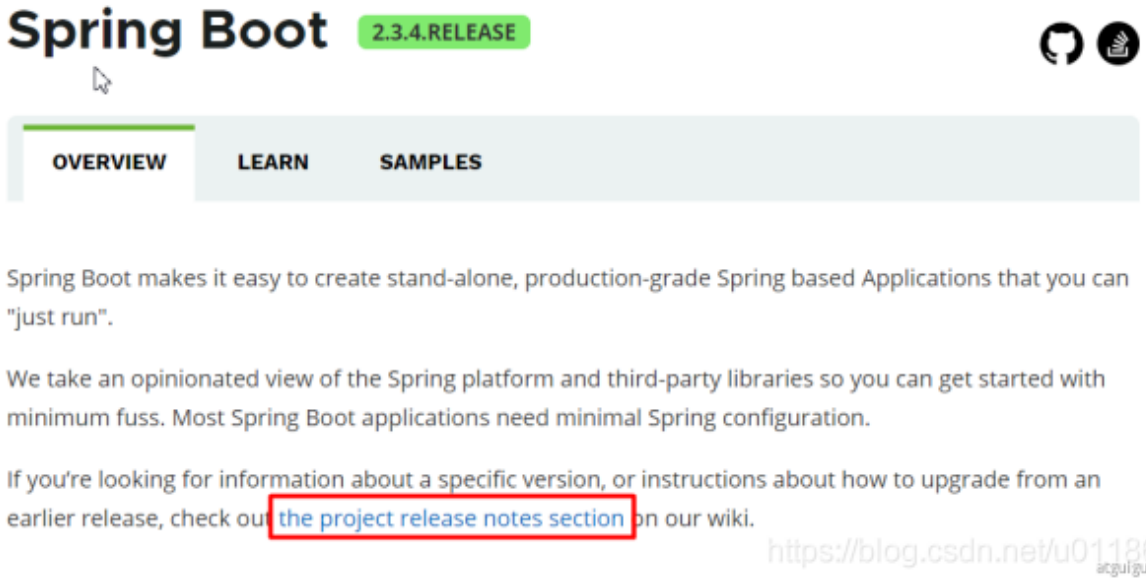
The reference documentation consists of the following sections:

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

OVERVIEW LEARN SAMPLES

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-HelloWorld

系统要求

- Java 8
- Maven 3.3+
- IntelliJ IDEA 2019.1.2

Maven配置文件

新添内容：

```
<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>
```

HelloWorld项目

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

创建maven工程

引入依赖

```
<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>
```

创建主程序

```
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);
    }
}
```

编写业务

```
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!";
    }
}
```

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!`。

设置配置

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

```
# 设置端口号

server.port=8888
```

更多配置信息

打包部署

在pom.xml添加

```
<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。

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

```
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
    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。

```
所有场景启动器最底层的依赖
<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。

```
<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

```
<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
    - 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

```
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);
    }
}
```

- 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
  - 无需以前的包扫描配置
  - 想要改变扫描路径
    - @SpringBootApplication(scanBasePackages="com.lun")
    - @ComponentScan 指定扫描路径

```
@SpringBootApplication
等同于
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan("com.lun")
```

- 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
  - 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

```
/**
 * 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:

```
@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
  - 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）

> lite 英 [laɪt] 美 [laɪt]
> adj. 低热量的，清淡的(light的一种拼写方法);类似...的劣质品

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

```
@Import({User.class, DBHelper.class})
@Configuration(proxyBeanMethods = false) //告诉SpringBoot这是一个配置类 == 配置文件
public class MyConfig {
}
```

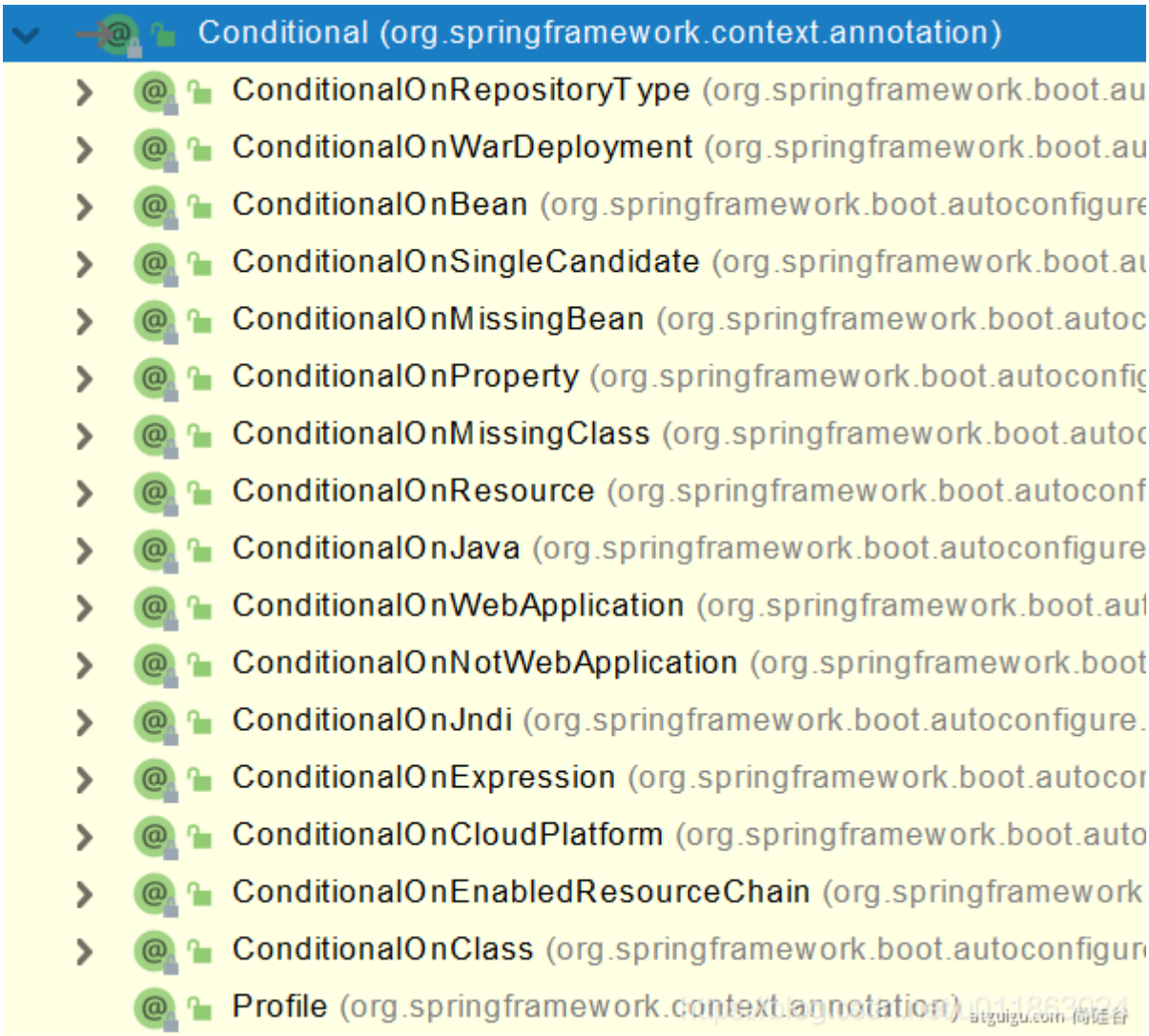
测试类：

```


//1、返回我们IOC容器
ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class, args);

//...

//5、获取组件
String[] beanNamesForType = run.getBeanNamesForType(User.class);

for (String s : beanNamesForType) {
    System.out.println(s);
}
```

```
DBHelper bean1 = run.getBean(DBHelper.class);
System.out.println(bean1);
```

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



用@ConditionalOnMissingBean举例说明

```
@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

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 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：

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

测试类：

```
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

传统方法：

```
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。
         }
     }
 }
```

Spring Boot一种配置配置绑定：

@ConfigurationProperties + @Component

假设有配置文件application.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

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

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

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

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

13、自动配置【源码分析】-自动包规则原理

Spring Boot应用的启动类：

```
@SpringBootApplication
public class MainApplication {

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

}
```

分析下`@SpringBootApplication`

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

重点分析`@SpringBootConfiguration`，`@EnableAutoConfiguration`，`@ComponentScan`。

@SpringBootConfiguration

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

`@Configuration`代表当前是一个配置类。

@ComponentScan

指定扫描哪些Spring注解。

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

@EnableAutoConfiguration

```
@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。

```
@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`


```
# 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：

```
@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:

```
@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。**

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 ----> application.properties

16、最佳实践-SpringBoot应用如何编写

- 引入场景依赖
    - 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：

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

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

```
@NoArgsConstructor
//@AllArgsConstructor
@Data
@ToString
@EqualsAndHashCode
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

```
@Slf4j
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String handle01(@RequestParam("name") String name){
        log.info("请求进来了....");
        return "Hello, Spring Boot 2!"+"你好: "+name;
    }
}
```

18、最佳实践-dev-tools

> Spring Boot includes an additional set of tools that can make the application development experience a little more pleasant. The `spring-boot-devtools` module can be included in any project to provide additional development-time features.——link
>
> Applications that use `spring-boot-devtools` automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE, as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a directory is monitored for changes. Note that certain resources, such as static assets and view templates, do not need to restart the application.——link
>
> Triggering a restart
>
> As DevTools monitors classpath resources, the only way to trigger a restart is to update the classpath. The way in which you cause the classpath to be updated depends on the IDE that you are using:
>
> - In Eclipse, saving a modified file causes the classpath to be updated and triggers a restart.
> - In IntelliJ IDEA, building the project (`Build -> Build Project`)(shortcut: Ctrl+F9) has the same effect.

添加依赖：

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

在IDEA中，项目或者页面修改以后：Ctrl+F9。

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。

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。

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

数据类型

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

```
k: v
```

- 对象：键值对的集合。map、hash、set、object

```
#行内写法:

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

```
#行内写法:

k: [v1,v2,v3]

#或者

k:
 - v1
 - v2
 - v3
```

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
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}]
```

21、配置文件-自定义类绑定的配置提示

> You can easily generate your own configuration metadata file from items annotated with `@ConfigurationProperties` by using the `spring-boot-configuration-processor` jar. The jar includes a Java annotation processor which is invoked as your project is compiled.——link

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：

```
<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

Spring Boot provides auto-configuration for Spring MVC that **works well with most applications.(大多场景我们都无需自定义配置)**

The auto-configuration adds the following features on top of Spring's defaults:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
  - 内容协商视图解析器和BeanName视图解析器
- Support for serving static resources, including support for WebJars (covered later in this document)).
  - 静态资源（包括webjars）
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
  - 自动注册 `Converter, GenericConverter, Formatter`
- Support for `HttpMessageConverters` (covered later in this document).
  - 支持 `HttpMessageConverters` （后来我们配合内容协商理解原理）
- Automatic registration of `MessageCodesResolver` (covered later in this document).
  - 自动注册 `MessageCodesResolver` （国际化用）

- Static `index.html` support.
  - 静态index.html 页支持
- Custom `Favicon` support (covered later in this document).
  - 自定义 `Favicon`
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered later in this document).
  - 自动使用 `ConfigurableWebBindingInitializer` ，（DataBinder负责将请求数据绑定到JavaBean上）

> 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

静态资源目录

只要静态资源放在类路径下： called `/static` (or `/public` or `/resources` or `/META-INF/resources`

访问： 当前项目根路径/ + 静态资源名

原理： 静态映射/**。

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。

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

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

静态资源访问前缀

```yaml
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

webjar

可用jar方式添加css，js等资源文件，

https://www.webjars.org/

例如，添加jquery

```
<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

官方文档

欢迎页支持

- 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

```
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。

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。

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

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

```
@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**

```
@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

```
////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();
}
```

- ResourceProperties resourceProperties；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
- HttpMessageConverters 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

```
...
public class WebMvcAutoConfiguration {
    ...
    public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration
implements ResourceLoaderAware {
        ...
        @Override
        protected void addResourceHandlers(ResourceHandlerRegistry registry) {
            super.addResourceHandlers(registry);
            if (!this.resourceProperties.isAddMappings()) {
                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。

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

静态资源规则：

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

    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

```
...
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：

```
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

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

```
<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;
    }

    @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

```
@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 {

    ...

    @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：

```
@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`。

```
<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

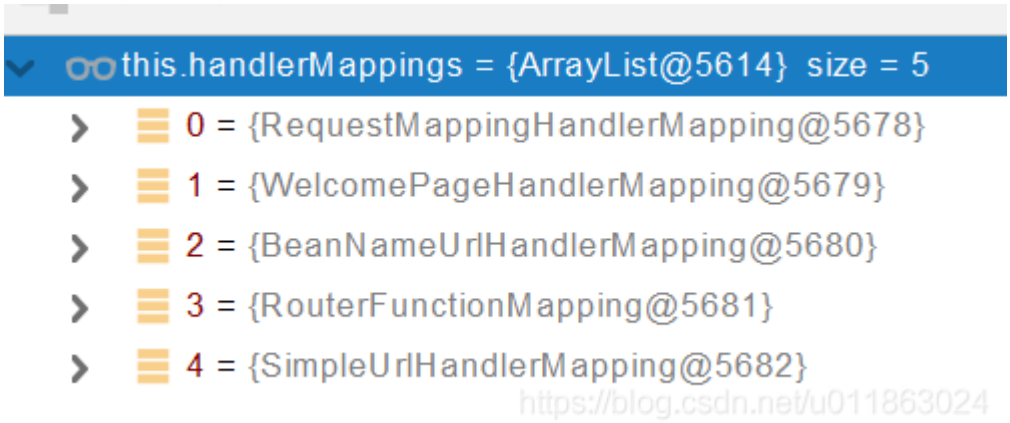
SpringMVC功能分析都从 `org.springframework.web.servlet.DispatcherServlet` -> `doDispatch()`

```
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：

```
@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：



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

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**。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

注解：

- `@PathVariable` 路径变量
- `@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）
- `@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`

使用用例：

```
@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);
        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;
    }
}
```

30、请求处理-@RequestAttribute

用例：

```
@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：

```
@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：

```
@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

```
@RestController
public class ParameterTestController {

    ///cars/sell;low=34;brand=byd,audi,yd
    @GetMapping("/cars/{path}")
    public Map carsSell(@MatrixVariable("low") Integer low,
                        @MatrixVariable("brand") List<String> brand,
                        @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：

```
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());
                ...
```

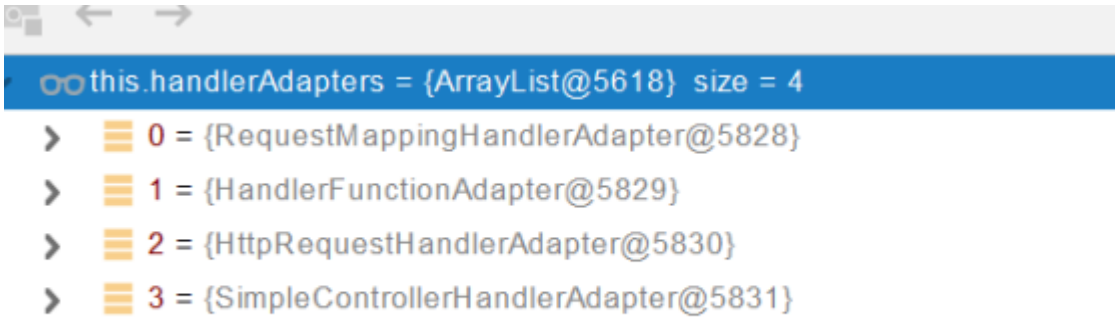
- `HandlerMapping`中找到能处理请求的`Handler`（Controller.method()）。
- 为当前Handler 找一个适配器`HandlerAdapter`，用的最多的是**RequestMappingHandlerAdapter**。
- 适配器执行目标方法并确定方法参数的每一个值。

HandlerAdapter

默认会加载所有`HandlerAdapter`

```
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`：



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

执行目标方法

```
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`）

```
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;
    }
}
```

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;

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

```
@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

```
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）。

```
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;

}
```

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
    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：

```
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：

```
public interface HandlerMethodReturnValueHandler {

    boolean supportsReturnType(MethodParameter returnType);

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

}
```

回顾执行目标方法

```
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：

```
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：

```
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：

```
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) {
                ...
            }
        }

}
```

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 {
    ...

    @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）

```
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

```
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：

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

用例：

```
@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,
                   @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：

```
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：

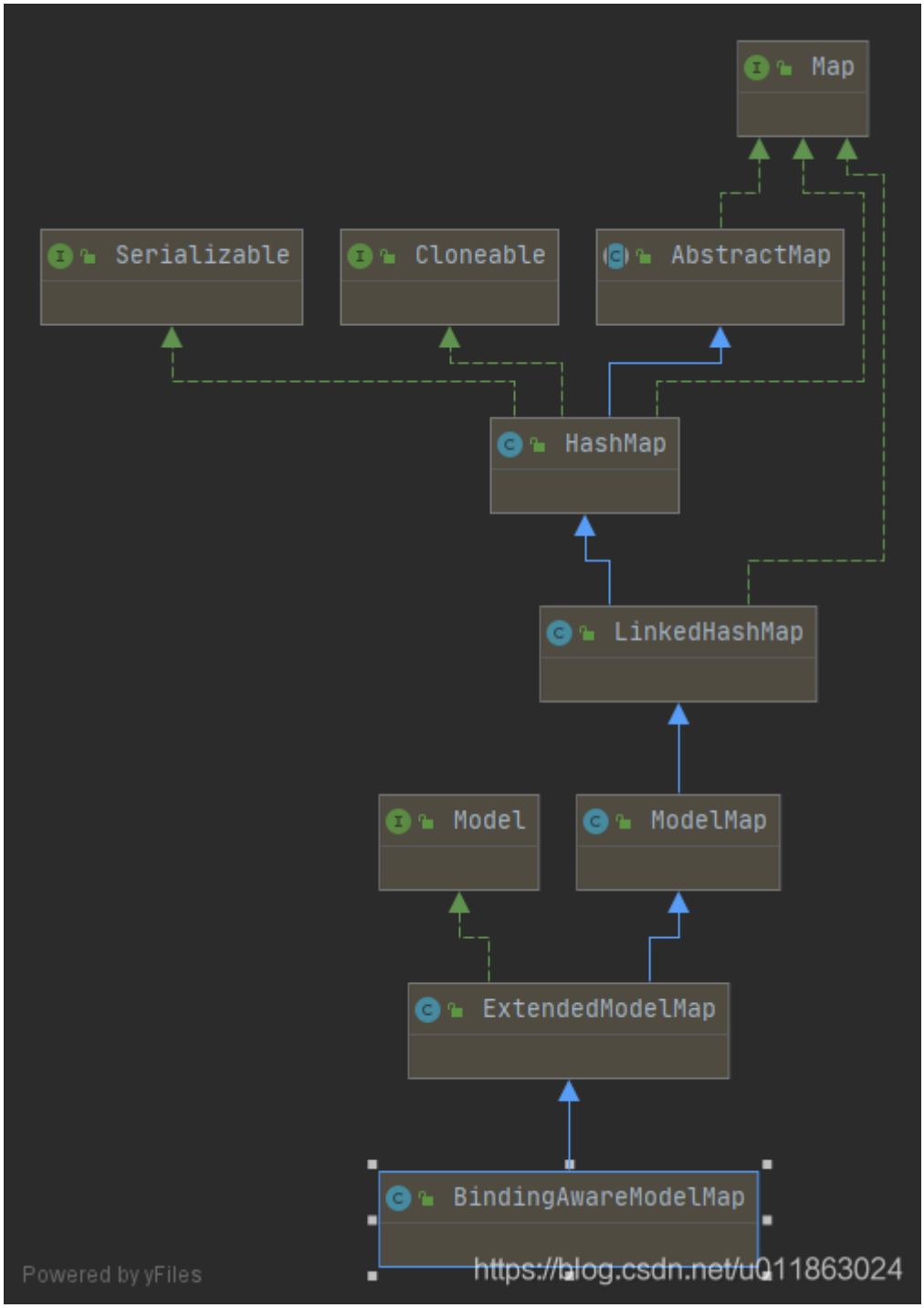
```
public class ModelAndViewContainer {

    ...

    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：

```
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?

```
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。

35、请求处理-【源码分析】-自定义参数绑定原理

```
@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;
    }
}
```

```
/**
 *      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`

```
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

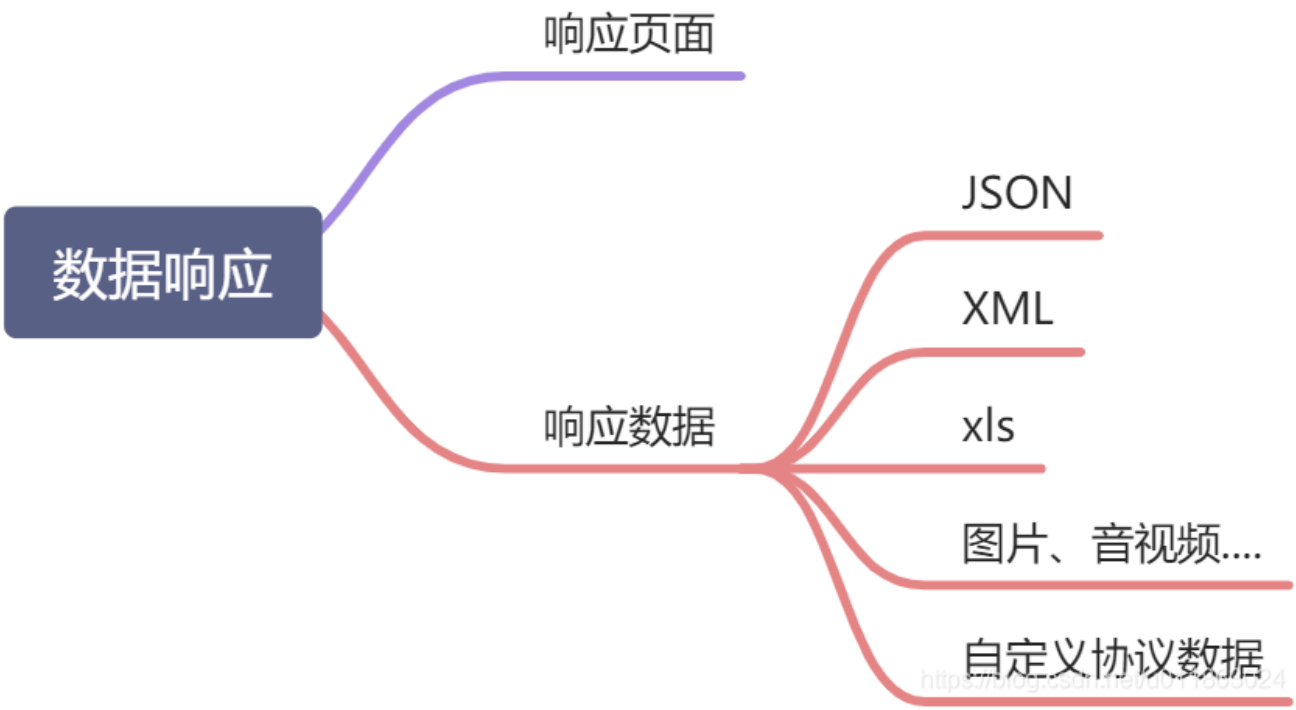
未来我们可以给WebDataBinder里面放自己的Converter；

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



假设给前端自动返回json数据，需要引入相关的依赖

```
<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：

```
@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：

```
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();
        }
    }
```

```
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) {
            ...
        }
    }
```

```
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

```
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);
    }

}
```

38、响应处理-【源码分析】-HTTPMessageConverter原理

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。

```
//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;

                    }
```

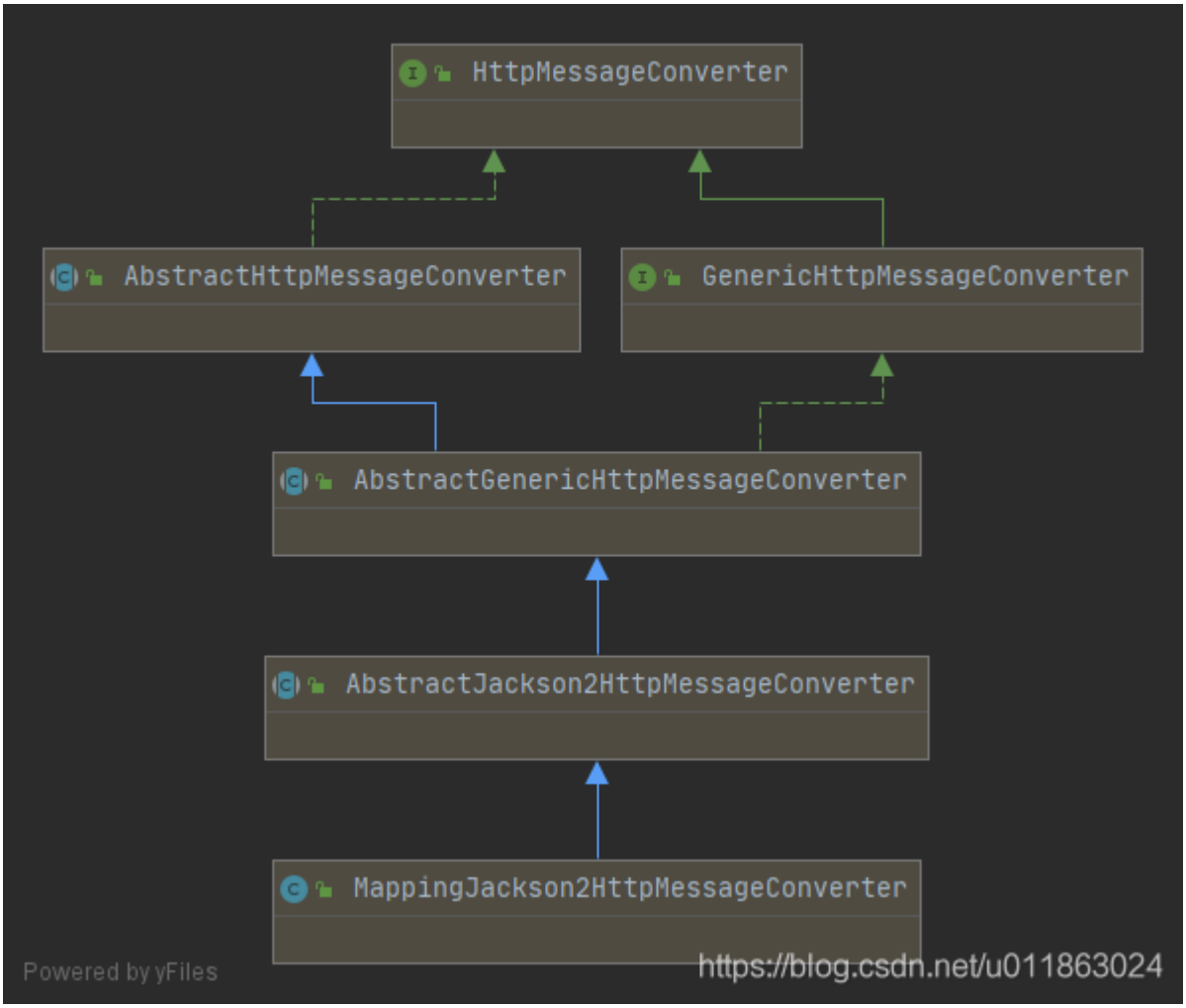
```
                }

                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：

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

    /**
     * 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`




```
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。

```
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：

```
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`

```
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。

引入XML依赖：

```
 <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。

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`。
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）（这一步在下一节有详细介绍）
   - `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。
6. 进行内容协商的最佳匹配媒体类型
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 。

```
//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

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 {

        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));
    }

}
```

```
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;
    }
    ...

}
```

```
//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());
        }
    }

}
```

开启浏览器参数方式内容协商功能

为了方便内容协商，开启基于请求参数的内容协商功能。

```
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）

```
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。

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，一个入口给容器中添加一个 `WebMvcConfigurer`

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

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

```


/**
 * 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());
    }
}
```

```
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。

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。

```
@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
                ParameterContentNegotiationStrategy parameterStrategy = new
ParameterContentNegotiationStrategy(mediaTypes);
//                parameterStrategy.setParameterName("ff");

                //还需添加请求头处理策略，否则accept:application/json、application/xml则会失效
                HeaderContentNegotiationStrategy headeStrategy = new
HeaderContentNegotiationStrategy();

                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

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

自动配置好了thymeleaf

```
@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

```
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：

```
@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`：

```
<!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
     */
    @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

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