

SpringBoot3核心特性

快速入门

简介

前置知识

- Java17
- Spring、SpringMVC、MyBatis
- Maven、IDEA

环境要求

环境&工具	版本 (or later)
SpringBoot	3.0.5+
IDEA	2021.2.1+
Java	17+
Maven	3.5+
Tomcat	10.0+
Servlet	5.0+
GraalVM Community	22.3+
Native Build Tools	0.9.19+

SpringBoot是什么

SpringBoot 帮我们简单、快速地创建一个独立的、生产级别的 **Spring 应用** (说明: SpringBoot底层是 Spring)

大多数 SpringBoot 应用只需要编写少量配置即可快速整合 Spring 平台以及第三方技术

特性:

- 快速创建独立 Spring 应用
 - SSM: 导包、写配置、启动运行
- 直接嵌入Tomcat、Jetty or Undertow (无需部署 war 包) 【Servlet容器】
 - linux java tomcat mysql: war 放到 tomcat 的 webapps 下
 - jar: java环境; java -jar
- **重点:** 提供可选的starter, 简化应用整合
 - 场景启动器 (starter) : web、json、邮件、oss (对象存储) 、异步、定时任务、缓存...
 - 导包一堆, 控制好版本。
 - 为每一种场景准备了一个依赖; **web-starter, mybatis-starter**
- **重点:** 按需自动配置 Spring 以及 第三方库
 - 如果这些场景我要使用 (生效) 。这个场景的所有配置都会自动配置好。

- 约定大于配置：每个场景都有很多默认配置。
- 自定义：配置文件中修改配置项就可以
- 提供生产级特性：
 - 如监控指标、健康检查（k8s）、外部化配置
- 无代码生成、无xml

总结：简化开发，简化配置，简化整合，简化部署，简化监控，简化运维。

快速体验

场景：浏览器发送/hello请求，返回“Hello, Spring Boot 3！”

开发流程

1. 创建项目

maven项目

```
<!-- 所有springboot项目都必须继承自 spring-boot-starter-parent -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.5</version>
</parent>
```

2. 导入场景

场景启动器

```
<dependencies>
    <!--web开发的场景启动器 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

3. 主程序

```
@SpringBootApplication //这是一个SpringBoot应用
public class MainApplication {
    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class,args);
    }
}
```

4. 业务

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello(){
        return "Hello, Spring Boot 3!";
    }
}
```

5. 测试

默认启动访问： localhost:8080

6. 打包

```
<!-- SpringBoot应用打包插件-->
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

`mvn clean package` 把项目打成可执行的jar包

`java -jar demo.jar` 启动项目

特性小结

1. 简化整合

导入相关的场景，拥有相关功能。场景启动器

默认支持的所有场景：<https://docs.spring.io/spring-boot/docs/current/reference/html/using-build-systems.starter>

- 官方提供的场景：命名为： `spring-boot-starter-*`
- 第三方提供场景：命名为： `*-spring-boot-starter`

场景一导入，万物皆就绪

2. 简化开发

无需编写任何配置，直接开发业务

3. 简化配置

`application.properties`：

- 集中式管理配置。只需要修改这个文件就行。
- 配置基本都有默认值
- 能写的所有配置都在：<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties>

4. 简化部署

打包为可执行的jar包。

linux服务器上有java环境。

5. 简化运维

修改配置（外部放一个`application.properties`文件）、监控、健康检查...

Spring Initializr 创建向导

一键创建好整个项目结构



应用分析

依赖管理机制

思考：

1、为什么导入 `starter-web` 所有相关依赖都导入进来？

- 开发什么场景，导入什么 **场景启动器**。
- **maven依赖传递原则。A-B-C： A就拥有B和C**
- 导入 场景启动器。 场景启动器 自动把这个场景的所有核心依赖全部导入进来

2、为什么版本号都不用写？

- 每个boot项目都有一个父项目 `spring-boot-starter-parent`
- parent的父项目是 `spring-boot-dependencies`
- 父项目 **版本仲裁中心**，把所有常见的jar的依赖版本都声明好了。
- 比如：`mysql-connector-j`

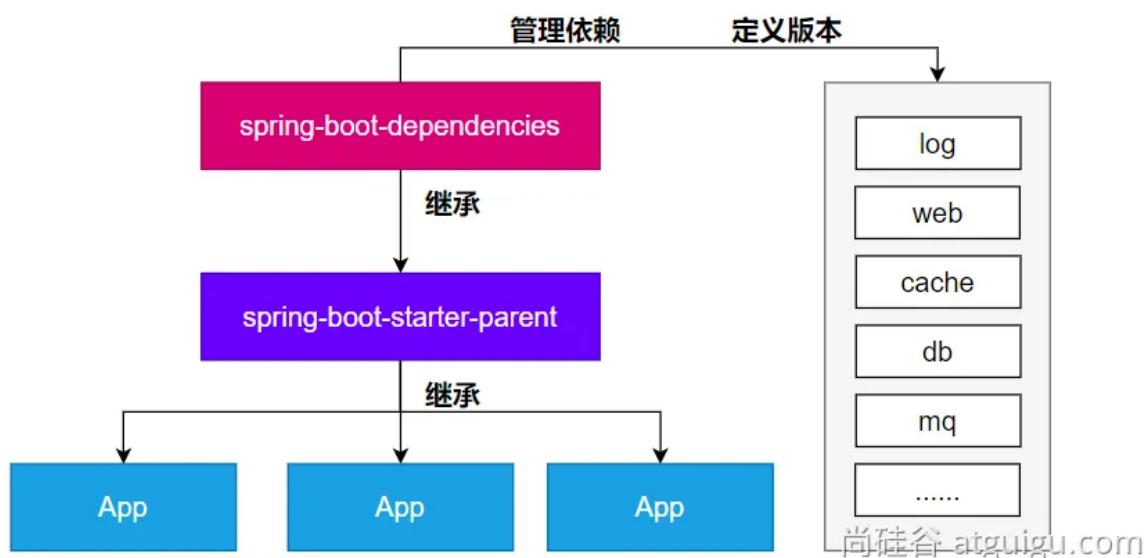
3、自定义版本号

- 利用maven的就近原则
 - 直接在当前项目 `properties` 标签中声明父项目用的版本属性的key
 - 直接在**导入依赖的时候声明版本**

4、第三方的jar包

- boot父项目没有管理的需要自行声明好

```
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.16</version>
</dependency>
```



自动配置机制

初步理解

- 自动配置的 Tomcat、SpringMVC 等
 - 导入场景，容器中就会自动配置好这个场景的核心组件。
 - 以前：DispatcherServlet、ViewResolver、CharacterEncodingFilter....
 - 现在：自动配置好的这些组件
 - 验证：容器中有了什么组件，就具有什么功能

```
public static void main(String[] args) {  
  
    //java10: 局部变量类型的自动推断  
    var ioc = SpringApplication.run(MainApplication.class, args);  
  
    //1、获取容器中所有组件的名字  
    String[] names = ioc.getBeanDefinitionNames();  
    //2、挨个遍历：  
    // dispatcherServlet、beanNameViewResolver、characterEncodingFilter、  
    multipartResolver  
    // SpringBoot把以前配置的核心组件现在都给我们自动配置好了。  
    for (String name : names) {  
        System.out.println(name);  
    }  
}
```

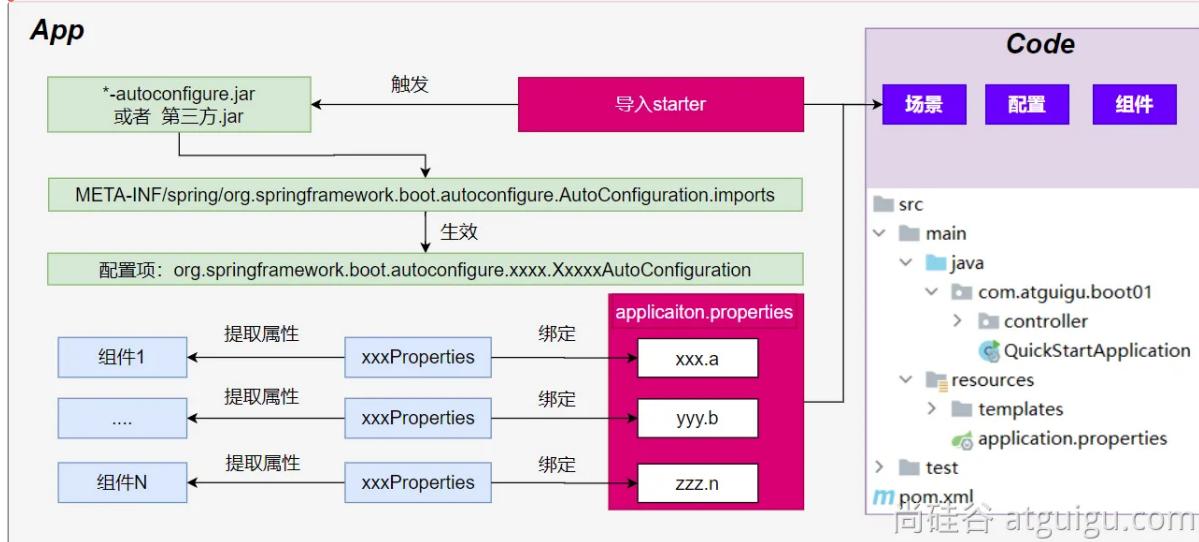
- 默认的包扫描规则
 - @SpringBootApplication 标注的类就是主程序类
 - SpringBoot只会扫描主程序所在的包及其下面的子包，自动的component-scan功能
 - 自定义扫描路径
 - @SpringBootApplication(scanBasePackages = "com.atguigu")
 - @ComponentScan("com.atguigu") 直接指定扫描的路径
- 配置默认值
 - 配置文件的所有配置项是和某个类的对象值进行一一绑定的。
 - 绑定了配置文件中每一项值的类：属性类。
 - 比如：
 - ServerProperties 绑定了所有Tomcat服务器有关的配置
 - MultipartProperties 绑定了所有文件上传相关的配置
 -参照[官方文档](#)：或者参照 绑定的 属性类。
- 按需加载自动配置
 - 导入场景 spring-boot-starter-web
 - 场景启动器除了会导入相关功能依赖，导入一个 spring-boot-starter，是所有 starter 的 starter，基础核心starter
 - spring-boot-starter 导入了一个包 spring-boot-autoconfigure。包里面都是各种场景的 AutoConfiguration 自动配置类
 - 虽然全场景的自动配置都在 spring-boot-autoconfigure 这个包，但是不是全都开启的。
 - 导入哪个场景就开启哪个自动配置

总结：导入场景启动器、触发 spring-boot-autoconfigure 这个包的自动配置生效、容器中就会具有相关场景的功能

完整流程

思考：

- 1、SpringBoot怎么实现导入一个 **starter**、写一些简单配置，应用就能跑起来，我们无需关心整合
- 2、为什么Tomcat的端口号可以配置在 `application.properties` 中，并且 Tomcat 能启动成功？
- 3、导入场景后哪些自动配置能生效？



自动配置流程细节梳理：

1. 导入 starter，导入了web开发场景

1. 场景启动器导入了相关场景的依赖： `starter-json`、`starter-tomcat`、`springmvc`
2. 每一个场景启动器都引入了一个 `spring-boot-starter`，核心场景启动器
3. 核心场景启动器引入了 `spring-boot-autoconfigure` 包。
4. `spring-boot-autoconfigure` 里面囊括了所有场景的所有配置
5. 只要这个包下所有类都能剩下，那么相当于SpringBoot官方写好的整合功能就生效了
6. SpringBoot默认扫描不到 `spring-boot-autoconfigure` 下写好的所有配置类（这些配置类给我们做了整合操作），默认只扫描主程序所在的包

2. 主程序： `@SpringBootApplication`

1. `@SpringBootApplication` 由三个注解组成 `@SpringBootConfiguration`、`@EnableAutoConfiguration`、`@ComponentScan`
2. SpringBoot默认只能扫描自己主程序所在的包及其下面的子包，扫描不到 `spring-boot-autoconfigure` 包中官方写好的配置类
3. `EnableAutoConfiguration`：SpringBoot开启自动配置的核心
 1. 是由 `@Import({AutoConfigurationImportSelector.class})` 提供功能：批量给容器导入组件
 2. SpringBoot启动会默认加载142个配置类
 3. 这142个配置类来自于 `spring-boot-autoconfigure` 下 `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 文件指定的
 4. 项目启动的时候利用 `@Import` 批量导入机制把 `autoconfigure` 包下的 142 的 `xxxAutoConfiguration` 类导入进来（自动配置类）
4. 按需生效：
 1. 并不是这142个自动配置类都能生效
 2. 每一个自动配置类，都有条件注解 `@ConditionalOnxxx`，只有条件成立，才能生效
3. `xxxAutoConfiguration` 自动配置类

1. 给容器中使用 @Bean 放一堆组件
 2. 每个自动配置类都可能有这个注解
 `@EnableConfigurationProperties({ServerProperties.class})`, 用来把配置文件中配的指定前缀的属性值封装到 `xxxProperties` 属性类中
 3. 以Tomcat为例, 把服务器的所有配置都是以 `server` 开头的。配置都封装到属性类中
 4. 给容器中放的所有组件的一些核心参数, 都来自于 `xxxProperties`, `xxxProperteis` 都是和配置文件绑定。
 5. 只需要改配置文件的值, 核心组件的底层参数都能修改
4. **写业务, 全程无需关心各种整合 (底层这些整合好了, 而且也生效了)**

核心流程总结:

1. 导入 `starter`, 就会导入 `autoconfigure` 包
2. `autoconfigure` 包里面有一个文件 `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 指定了所有启动要加载的自动配置类
3. `@EnableAutoConfiguration` 会自动把上面文件里面写的所有自动配置类都导入进来。
`xxxAutoConfiguration` 是有条件注解进行按需加载
4. `xxxAutoConfiguration` 给容器中导入一堆组件, 组件都是从 `xxxProperties` 中提取属性值
5. `xxxProperties` 又是和配置文件进行了绑定

效果: 导入 `starter`、修改配置文件, 就能修改底层行为

如何学好SpringBoot3

框架的框架、底层基于Spring。能调整每一个场景的底层行为。100%项目一定会用到底层自定义

1. 理解自动配置原理

1. 导入 `starter` => 生效 `xxxxAutoConfiguration` => 组件 => `xxxProperties` => 配置文件

2. 理解其他框架底层

1. 拦截器

3. 可以随时定制化任何组件

1. 配置文件

2. 自定义组件

- 普通开发: 导入 `starter`, `Controller`、`Service`、`Mapper`、偶尔修改配置文件
- 高级开发: 自定义组件、自定义配置、自定义 `starter`
- 核心:
 - 这个场景自动配置导入了哪些组件, 我们能不能 `Autowired` 进来使用
 - 能不能通过修改配置改变组件的一些默认参数
 - 需不需要自己完全定义这个组件
 - 场景定制化

最佳实战:

- 选场景, 导入到项目
 - 官方: `starter`
 - 第三方: 去仓库搜
- 写配置, 改配置文件关键项
 - 数据库参数 (连接地址、账号密码...)
- 分析这个场景给我们导入了哪些能用的组件

- 自动装配这些组件进行后续使用
- 不满意boot提供的自动配好的默认组件

- 定制化
 - 改配置
 - 自定义组件

整合redis:

- 选场景: `spring-boot-starter-data-redis`
 - 场景AutoConfiguration 就是这个场景的自动配置类
- 写配置:
 - 分析到这个场景的自动配置类开启了哪些属性绑定关系
 - `@EnableConfigurationProperties(RedisProperties.class)`
 - 修改redis相关的配置
- 分析组件:
 - 分析到 `RedisAutoConfiguration` 给容器中放了 `StringRedisTemplate`
 - 给业务代码中自动装配 `StringRedisTemplate`
- 定制化
 - 修改配置文件
 - 自定义组件,自己给容器中放一个 `StringRedisTemplate`

核心技能

常用注解

SpringBoot摒弃XML配置方式, 改为全注解驱动

组件注册

`@Configuration`、`@SpringBootConfiguration`
`@Bean`、`@Scope`
`@Controller`、`@Service`、`@Repository`、`@Component`
`@Import`
`@Componentscan`

步骤:

- `@Configuration` 编写一个配置类
- 在配置类中, 自定义方法给容器中注册组件。配合`@Bean`
- 或使用`@Import` 导入第三方的组件

条件注解

如果注解指定的**条件成立**, 则触发指定行为

- `@ConditionalOnXxx`
- `@ConditionalOnClass`: 如果类路径中存在这个类, 则触发指定行为
- `@ConditionalOnMissingClass`: 如果类路径中不存在这个类, 则触发指定行为
- `@ConditionalOnBean`: 如果容器中存在这个Bean (组件), 则触发指定行为
- `@ConditionalOnMissingBean`: 如果容器中不存在这个Bean (组件), 则触发指定行为

场景：

- 如果存在 `Fastsq1Exception` 这个类，给容器中放一个 `Cat` 组件，名 `cat01`，
- 否则，就给容器中放一个 `Dog` 组件，名 `dog01`
- 如果系统中有 `dog01` 这个组件，就给容器中放一个 `User` 组件，名 `zhangsan`
- 否则，就放一个 `User`，名叫 `lisi`

- `@ConditionalOnBean (value=组件类型, name=组件名字)`：判断容器中是否有这个类型的组件，并且名字是指定的值

```
@ConditionalOnRepositoryType (org.springframework.boot.autoconfigure.data)
@ConditionalOnDefaultWebSecurity
(org.springframework.boot.autoconfigure.security)
@ConditionalOnSingleCandidate
(org.springframework.boot.autoconfigure.condition)
@ConditionalOnWebApplication (org.springframework.boot.autoconfigure.condition)
@ConditionalOnWarDeployment (org.springframework.boot.autoconfigure.condition)
@ConditionalOnJndi (org.springframework.boot.autoconfigure.condition)
@ConditionalOnResource (org.springframework.boot.autoconfigure.condition)
@ConditionalOnExpression (org.springframework.boot.autoconfigure.condition)
@ConditionalOnClass (org.springframework.boot.autoconfigure.condition)
@ConditionalOnEnabledResourceChain (org.springframework.boot.autoconfigure.web)
@ConditionalOnMissingClass (org.springframework.boot.autoconfigure.condition)
@ConditionalOnNotWebApplication
(org.springframework.boot.autoconfigure.condition)
@ConditionalOnProperty (org.springframework.boot.autoconfigure.condition)
@ConditionalOnCloudPlatform (org.springframework.boot.autoconfigure.condition)
@ConditionalOnBean (org.springframework.boot.autoconfigure.condition)
@ConditionalOnMissingBean** (org.springframework.boot.autoconfigure.condition)
@ConditionalOnMissingFilterBean
(org.springframework.boot.autoconfigure.web.servlet)
@Profile (org.springframework.context.annotation)
@ConditionalOnInitializedRestarter (org.springframework.boot.devtools.restart)
@ConditionalOnGraphQLSchema (org.springframework.boot.autoconfigure.graphql)
@ConditionalOnJava (org.springframework.boot.autoconfigure.condition)
```

属性绑定

`@ConfigurationProperties`: 声明组件的属性和配置文件哪些前缀开始项进行绑定

`@EnableConfigurationProperties`: 快速注册注解:

- 场景：SpringBoot默认只扫描自己主程序所在的包。如果导入第三方包，即使组件上标注了 `@Component`、`@ConfigurationProperties` 注解，也没用。因为组件都扫描不进来，此时使用这个注解就可以快速进行属性绑定并把组件注册进容器

将容器中任意组件（Bean）的属性值和配置文件的配置项的值进行绑定

- 给容器中注册组件（`@Component`、`@Bean`）
- 使用 `@ConfigurationProperties` 声明组件和配置文件的哪些配置项进行绑定

更多注解参照：[Spring注解驱动开发【1-26集】](#)

YML配置文件

痛点：SpringBoot 集中化管理配置，`application.properties`

问题：配置多以后难阅读和修改，层级结构辨识度不高

YAML 是 "YAML Ain't a Markup Language" (YAML 不是一种标记语言)。在开发的这种语言时，YAML 的意思其实是："Yet Another Markup Language" (是另一种标记语言)。

- 设计目标，就是**方便人类读写**
- **层次分明**，更适合做配置文件
- 使用 `.yaml` 或 `.yml` 作为文件后缀

基本语法

- **大小写敏感**
- 使用**缩进表示层级关系**，`k: v`，使用空格分割`k,v`
- 缩进时不允许使用Tab键，只允许**使用空格**。换行
- 缩进的空格数目不重要，只要**相同层级**的元素**左侧对齐**即可
- **# 表示注释**，从这个字符一直到行尾，都会被解析器忽略。

支持的写法：

- **对象**：键值对的集合，如：映射 (map) / 哈希 (hash) / 字典 (dictionary)
- **数组**：一组按次序排列的值，如：序列 (sequence) / 列表 (list)
- **纯量**：单个的、不可再分的值，如：字符串、数字、bool、日期

实例

```
@Component
@ConfigurationProperties(prefix = "person") //和配置文件person前缀的所有配置进行绑定
@Data //自动生成JavaBean属性的getter/setter
//@NoArgsConstructor //自动生成无参构造器
//@AllArgsConstructor //自动生成全参构造器
public class Person {
    private String name;
    private Integer age;
    private Date birthDay;
    private Boolean like;
    private Child child; //嵌套对象
    private List<Dog> dogs; //数组（里面是对象）
    private Map<String,Cat> cats; //表示Map
}

@Data
public class Dog {
    private String name;
    private Integer age;
}

@Data
public class Child {
    private String name;
    private Integer age;
    private Date birthDay;
    private List<String> text; //数组
}
```

```
@Data  
public class Cat {  
    private String name;  
    private Integer age;  
}
```

properties表示方式

```
person.name=张三  
person.age=18  
person.birthDay=2010/10/12 12:12:12  
person.like=true  
person.child.name=李四  
person.child.age=12  
person.child.birthDay=2018/10/12  
person.child.text[0]=abc  
person.child.text[1]=def  
person.dogs[0].name=小黑  
person.dogs[0].age=3  
person.dogs[1].name=小白  
person.dogs[1].age=2  
person.cats.c1.name=小蓝  
person.cats.c1.age=3  
person.cats.c2.name=小灰  
person.cats.c2.age=2
```

yaml表示法

```
person:  
    name: 张三  
    age: 18  
    birthDay: 2010/10/10 12:12:12  
    like: true  
    child:  
        name: 李四  
        age: 20  
        birthDay: 2018/10/10  
        text: ["abc", "def"]  
    dogs:  
        - name: 小黑  
          age: 3  
        - name: 小白  
          age: 2  
    cats:  
        c1:  
            name: 小蓝  
            age: 3  
        c2: {name: 小绿, age: 2} #对象也可用{}表示
```

细节

- birthDay 推荐写为 birth-day
- 文本：
 - 单引号不会转义【\n 则为普通字符串显示】
 - 双引号会转义【\n会显示为换行符】
- 大文本
 - | 开头，大文本写在下层，保留文本格式，换行符正确显示

- >开头，大文本写在下层，折叠换行符
- 多文档合并
 - 使用 `--` 可以把多个yaml文档合并在一个文档中，每个文档区依然认为内容独立
3. 小技巧：lombok

简化JavaBean 开发。自动生成构造器、getter/setter、自动生成Builder模式等

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

使用 `@Data` 等注解

日志配置

规范：项目开发不要编写 `System.out.println()`，应该用日志记录信息

日志门面	日志实现
JCL (Jakarta Commons Logging)	Log4j
SLF4j (Simple Logging Facade for Java)	JUL (java.util.logging)
jboss-logging	Log4j2 Logback

尚硅谷 atguigu.com

感兴趣日志框架关系与起源可参考：<https://www.bilibili.com/video/BV1gW411W76m> 视频 21~27集

简介

1. Spring使用 `commons-logging` 作为内部日志，但底层日志实现是开放的。可对接其他日志框架。
 1. spring5及以后 `commons-logging` 被spring直接自己写了
2. 支持 `jui`, `log4j2`, `logback`。SpringBoot提供了默认的控制台输入配置，也可以配置输出为文件。
3. `Logback` 是默认使用的。
4. 虽然 **日志框架很多**，但是我们不用担心，使用SpringBoot的默认配置就能工作很好

- **SpringBoot怎么把日志默认配置好的**

1. 每个 `starter` 场景，都会导入一个核心场景 `spring-boot-starter`
2. 核心厂家引入了日志所用功能 `spring-boot-starter-logging`
3. 默认使用了 `Logback + slf4j` 组合作为默认底层日志
4. **日志是系统一启动就要用**，`xxxAutoConfiguration` 是系统启动好了以后放好的组件，后来用的。
5. 日志是利用**监听器机制**配置好的。`ApplicationListener`。
6. 日志所有的配置都可以通过修改配文件实现。以 `logging` 开始的所有配置

日志格式

```
2023-03-31T13:56:17.511+08:00 INFO 4944 --- [           main]
o.apache.catalina.core.StandardService   : Starting service [Tomcat]

2023-03-31T13:56:17.511+08:00 INFO 4944 --- [           main]
o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache
Tomcat/10.1.7]
```

默认输出格式：

- 时间和日期：毫秒级精度
- 日志级别： `ERROR`、`WARN`、`INFO`、`DEBUG`、or `TRACE`
- 进程ID
- ：消息分隔符
- 线程名：使用 `[]` 进行包含
- Logger名：通常是产生日志的类名
- 消息：日志记录的内容

注意：logback没有 `FATAL` 级别，对应的是 `ERROR`

- 默认值：参照：`spring-boot`包`additional-spring-configuration-metadata.json`文件
- 默认输出格式值：`%c\lr(%d${${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}})`
`{faint} %c\lr(${LOG_LEVEL_PATTERN:-%5p}) %c\lr(${PID:- }){magenta} %c\lr(---`
`{faint} %c\lr([%15.15t]) {faint} %c\lr(%-40.40logger{39}){cyan} %c\lr(:){faint}`
`%m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}`
- 可修改为：`'%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level [%thread] %logger{15} ===>`
`%msg%n'`

记录日志

```
Logger logger = LoggerFactory.getLogger(getClass());

// 或者使用Lombok的@Slf4j注解
```

日志级别

- 由低到高：`ALL, TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF`；
 - 只会打印指定级别及以上的日志
 - `ALL`：打印所有的日志
 - `TRACE`：追踪框架详细流程日志，一般不使用
 - `DEBUG`：开发调试细节
 - `INFO`：关键、感兴趣的日志
 - `WARN`：警告但不是错误的信息日志，比如：版本过时
 - `ERROR`：业务错误日志，比如出现各种异常
 - `FATAL`：致命错误日志，比如jvm系统崩溃
 - `OFF`：关闭所有日志记录
 - 不指定级别的所有类，都是用root指定的级别作为默认级别
 - SpringBoot日志默认值是`INFO`
1. 在 `application.properties/yaml` 中配置 `logging.level.<logger-name>=<level>` 指定日志级别

2. `level` 可取值范围: `TRACE, DEBUG, INFO, WARN, ERROR, FATAL, or OFF`, 定义在 `LogLevel` 类中
3. root的 `logging-name` 叫 `root`, 可以配置 `logging.level.root=warn`, 代表所有未指定日志级别都是用root的warn级别

日志分组

比较有用的技巧是:

将相关的 `logger` 分组在一起, 统一配置。SpringBoot 也支持。比如: Tomcat 相关的日志统一设置

```
logging.group.tomcat=org.apache.catalina,org.apache.coyote,org.apache.tomcat
logging.level.tomcat=trace
```

SpringBoot 预定义两个组

Name	Loggers
web	org.springframework.core.codec, org.springframework.http, org.springframework.web, org.springframework.boot.actuate.endpoint.web, org.springframework.boot.web.servlet.ServletContextInitializerBeans
sql	org.springframework.jdbc.core, org.hibernate.SQL, org.jooq.tools.LoggerListener

文件输出

SpringBoot 默认只把日志写在控制台, 如果想额外记录到文件, 可以在 `application.properties` 中添加 `logging.file.name` 或 `logging.file.path` 配置项。

logging.file.name	logging.file.path	示例	效果
未指定	未指定		仅控制台输出
指定	未指定	my.log	写入指定文件。可以加路径
未指定	指定	/var/log	写入指定目录, 文件名为spring.log
指定	指定		以logging.file.name为准

文档归档与滚动切割

归档: 每天的日志都单独存到一个文档中

切割: 每个文件10MB, 超过大小切割成另外一个文件

1. 每天的日志应该独立分割出来存档, 如果使用 `logback` (SpringBoot 默认整合), 可以通过 `application.properties/yaml` 文件指定日志滚动规则
2. 如果是其他日志系统, 需要自行配置 (添加 `log4j2.xml` 或 `log4j2-spring.xml`)
3. 支持的滚动规则设置如下:

配置项	描述
<code>logging.logback.rollingpolicy.fileNamePattern</code>	日志存档的文件名格式 (默认值: \${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz)
<code>logging.logback.rollingpolicy.cleanHistoryOnStart</code>	应用启动时是否清除以前存档 (默认值: false)
<code>logging.logback.rollingpolicy.maxFileSize</code>	存档前, 每个日志文件的最大大小 (默认值: 10MB)
<code>logging.logback.rollingpolicy.totalSizeCap</code>	日志文件被删除之前, 可以容纳的最大大小 (默认值: 0B)。设置1GB则磁盘存储超过1GB 日志后就会删除旧日志文件
<code>logging.logback.rollingpolicy.maxHistory</code>	日志文件保存的最大天数(默认值: 7).

自定义配置

通常我们配置 `application.properties` 就够了。当然也可以自定义。比如:

日志系统	自定义
Logback	<code>logback-spring.xml</code> , <code>logback-spring.groovy</code> , <code>logback.xml</code> , or <code>logback.groovy</code>
Log4j2	<code>log4j2-spring.xml</code> or <code>log4j2.xml</code>
JDK (Java Util Logging)	<code>logging.properties</code>

如果可能, 我们建议您在日志配置中使用 `-spring` 变量 (例如, `logback-spring.xml` 而不是 `logback.xml`)。如果您使用标准配置文件, `spring` 无法完全控制日志初始化。

最佳实践: 自己要写配置, 配置文件名加上 `xx-spring.xml`

切换日志组合

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

log4j2支持yaml和json格式的配置文件

格式	依赖	文件名
YAML	com.fasterxml.jackson.core:jackson-databind + com.fasterxml.jackson.dataformat:jackson-dataformat-yaml	log4j2.yaml + log4j2.yml
JSON	com.fasterxml.jackson.core:jackson-databind	log4j2.json + log4j2.jsn

最佳实战

- 导入任何第三方框架，先排除它的日志包，因为Boot底层控制好了日志
- 修改 `application.properties` 配置文件，就可以调整日志的所有行为。如果不够，可以编写日志框架自己的配置文件放在类路径下就行，比如 `logback-spring.xml`, `log4j2-spring.xml`
- 如需对接**专业日志系统**，也只需要把 `logback` 记录的日志灌倒 `kafka` 之类的中间件，这和SpringBoot没关系，都是日志框架自己的配置，**修改配置文件即可**
- 业务中使用slf4j-api记录日志。不要再 sout 了**

Web开发

SpringBoot的Web开发能力，由SpringMVC提供

WebMvc AutoConfiguration原理

生效条件

```
// 在这些自动配置之后
@AutoConfiguration(after = {
    DispatcherServletAutoConfiguration.class,
    TaskExecutionAutoConfiguration.class,
    ValidationAutoConfiguration.class })
// 如果是web应用就生效，类似SERVLET、REACTIVE 响应式web
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({
    Servlet.class,
    DispatcherServlet.class,
    webMvcConfigurer.class})
// 容器中没有这个Bean，才生效
@ConditionalOnMissingBean({WebMvcConfigurationSupport.class})
// 优先级
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@ImportRuntimeHints({WebResourcesRuntimeHints.class})
public class webMvcAutoConfiguration {}
```

效果

- 放了两个Filter:
 - `HiddenHttpMethodFilter`: 页面表单提交Rest请求 (GET、POST、PUT、DELETE)
 - `FormContentFilter`: 表单内容Filter, GET (数据放URL后面)、POST (数据放请求体) 可以携带数据，PUT、DELETE的请求体数据会被忽略
- 给容器中放了 `webMvcConfigurer` 组件，给SpringMVC添加各种定制功能
 - 所有的功能最终会和配置文件进行绑定

2. WebMvcProperties: `spring.mvc` 配置文件

3. WebProperties: `spring.web` 配置文件

```
@Configuration(proxyBeanMethods = false)
@Import(EnableWebMvcConfiguration.class) // 额外导入了其他配置
@EnableConfigurationProperties({ WebMvcProperties.class, WebProperties.class })
@Order(0)
public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer,
ServletContextAware {}
```

WebMvcConfigurer接口

提供了SpringMVC底层的所有组件入口



向硅谷 atguigu.com

静态资源规则源码

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    if (!this.resourceProperties.isAddMappings()) {
        logger.debug("Default resource handling disabled");
        return;
    }
    addResourceHandler(registry, this.mvcProperties.getWebjarsPathPattern(),
                      "classpath:/META-INF/resources/webjars/");
    addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(),
                      registration) -> {

        registration.addResourceLocations(this.resourceProperties.getStaticLocations());
        if (this.servletContext != null) {
            ServletContextResource resource = new
                    ServletContextResource(this.servletContext, SERVLET_LOCATION);
            registration.addResourceLocations(resource);
        }
    });
}
```

1. 规则一：访问： `/webjars/**` 路径就去 `classpath:/META-INF/resources/webjars/` 下找资源

1. maven导入依赖

2. 规则二：访问： `/**` 路径就去 静态资源默认的四个位置

1. `classpath:/META-INF/resources/",`

2. `classpath:/resources/`

3. `classpath:/static/`
 4. `classpath:/public/`
3. 规则三：静态资源默认都有缓存规则的设置
1. 所有缓存的设置，直接通过配置文件，`spring.web`
 2. `cachePeriod`: 缓存周期，多久不用找服务器更新的。默认没有，以秒为单位
 3. `cacheControl`: HTTP缓存控制；<https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Caching>
 4. `useLastModified`: 是否使用最后一次修改，配合HTTP Cache规则

如果浏览器访问了一个静态资源 `index.js` 如果服务这个资源没有发生变化，下次访问的时候就可以直接让浏览器用自己缓存中的东西

```
registration.setCachePeriod(getSeconds(this.resourceProperties.getCache().getPeriod()));
registration.setCacheControl(this.resourceProperties.getCache().getCachecontrol()
    .toHttpCacheControl());registration.setUseLastModified(this.resourceProperties.getCache().isUseLastModified());
```

EnableWebMvcConfiguration源码

```
// SpringBoot 给容器中放 EnableWebMvcConfiguration组件,
// 我们如果自己放了 WebMvcConfigurationSupport, Boot的 webMvcConfiguration都会失效
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(WebProperties.class)
public static class EnableWebMvcConfiguration extends
DelegatingWebMvcConfiguration implements ResourceLoaderAware {}
```

1. `HandlerMapping`：根据请求路径 `/a` 找那个handler能处理请求
 1. `welcomePageHandlerMapping`
 1. 访问 `/**` 路径下的所有请求，都在以前四个静态资源路径下找，欢迎页也一样。
 2. 找 `index.html`：只要静态资源的为只有一个 `index.html` 页面，项目启动默认访问

为什么容器中放一个WebMvcConfigure就能配置底层行为

- `WebMvcAutoConfiguration` 是一个自动配置类，它里面有一个 `EnableWebMvcConfiguration`
- `EnableWebMvcConfiguration` 继承与 `DelegatingWebMvcConfiguration`，这两个都生效
- `DelegatingWebMvcConfiguration` 利用 DI 把容器中所有 `WebMvcConfigurer` 注入进来
- 别人调用 `DelegatingWebMvcConfiguration` 的方法配置底层规则，而它调用所有 `WebMvcConfigurer` 的所有配置底层方法。

WebMvcConfigurationSupport

提供了很多的默认设置

判断系统中是否有相应的类：如果有，就加入相应的 `HttpMessageConverter`

```
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);
```

Web场景

自动配置

1. 整合web场景

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

2. 引入了 `autoconfigure` 功能

3. `@EnableAutoConfiguration` 注解使用了
`@Import(AutoConfigurationImportSelector.class)` 批量导入组件

4. 加载 META-INF

`INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`

5. 所有的自动配置类

```
org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration
org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCus
tomizerAutoConfiguration
#====以下是响应式web场景和现在的没关系=====
org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguratio
n
org.springframework.boot.autoconfigure.web.reactive.ReactiveMultipartAutoConfig
uration
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAut
oConfiguration
org.springframework.boot.autoconfigure.web.reactive.webFluxAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.WebSessionIdResolverAutoCon
figuration
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfi
guration
org.springframework.boot.autoconfigure.web.reactive.function.client.ClientHttpc
onnectionAutoConfiguration
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAu
toConfiguration
#=====以上没关系=====
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfi
guration
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoC
onfiguration
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfigurat
ion
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguratio
n
```

```
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration  
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration
```

6. 绑定了配置文件的一堆配置项
 1. SpringMVC的所有配置 `spring.mvc`
 2. Web场景通用配置 `spring.web`
 3. 文件上传配置 `spring.servlet.multipart`
 4. 服务器的配置 `server`：比如：编码方式

默认效果

默认配置：

1. 包含了 `ContentNegotiatingViewResolver` 和 `BeanNameViewResolver` 组件，方便视图解析
2. 默认的静态资源处理机制：静态资源放在 `static` 文件夹下即可直接访问
3. 自动注册了 `Convert`, `GenericConverter`, `Formatter` 组件，适配常见的数据类型转换和格式化需求
4. 支持 `HttpMessageConverters`，可以方便返回 `json` 等数据类型
5. 注册 `MessageCodesResolver`，方便国际化及错误消息处理
6. 支持静态 `index.html`
7. 自动使用 `ConfigurableWebBindingInitializer`，实现 消息处理、数据绑定、类型转化等功能

重要：

- 如果想保持 boot mvc 的默认配置，并且自定义更多的 mvc 配置，如：interceptors、formatters，view controllers等。可以使用 `@Configuration` 注解添加一个 `WebMvcConfigurer` 类型的配置类，并不要标注 `@EnableWebMvc`
- 如果想保持 boot mvc 的默认配置，但要自定义核心组件实例，比如：
`RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, 或
`ExceptionHandlerExceptionResolver`, 给容器中放一个 `webMvcRegistrations` 组件即可
- 如果向全面接管Spring MVC, `@Configuration` 标注一个配置类，并加上 `@EnableWebMvc` 注解，实现 `webMvcConfigure` 接口

静态资源

默认规则

静态资源映射

静态资源映射规则在 `WebMvcAutoConfiguration` 中进行了定义：

1. `/webjars/**` 的所有路径 资源都在 `classpath:/META-INF/resources/webjars/`
2. `/**` 的所有路径 资源都在 `classpath:/META-INF/resources/`、`classpath:/resources/`、`classpath:/static/`、`classpath:/public/`
3. 所有静态资源都定义了缓存规则。【浏览器访问过一次，就会缓存一段时间】，但此功能参数无默认值
4. 1. `period`: 缓存间隔。默认 0S；
2. `cacheControl`: 缓存控制。默认无；
3. `useLastModified`: 是否使用lastModified头。默认 false

静态资源缓存

如前面所述

1. 所有静态资源都定义了缓存规则。【浏览器访问过一次，就会缓存一段时间】，但此功能参数无默认值
2. 1. period：缓存间隔。默认 0S；
2. cacheControl：缓存控制。默认无；
3. useLastModified：是否使用lastModified头。默认 false；

欢迎页

欢迎页规则在 `webMvcAutoConfiguration` 中进行了定义：

1. 在静态资源目录下找 `index.html`
2. 没有就在 `templates` 下找 `index` 模板页

favicon

1. 在静态资源目录下找 `favicon.ico`

缓存实验

```
server.port=9000

#1、spring.web:
# 1.配置国际化的区域信息
# 2.静态资源策略(开启、处理链、缓存)

#开启静态资源映射规则
spring.web.resources.add-mappings=true

#设置缓存
#spring.web.resources.cache.period=3600
##缓存详细合并项控制，覆盖period配置：
## 浏览器第一次请求服务器，服务器告诉浏览器此资源缓存7200秒，7200秒以内的所有此资源访问不用发
给服务器请求，7200秒以后发请求给服务器
spring.web.resources.cache.cachecontrol.max-age=7200
#使用资源 last-modified 时间，来对比服务器和浏览器的资源是否相同没有变化。相同返回 304
spring.web.resources.cache.use-last-modified=true
```

自定义静态资源规则

自定义静态资源路径、自定义缓存规则

配置方式

`spring.mvc`：静态资源访问前缀路径
`spring.web`：

- 静态资源目录
- 静态资源缓存策略

```
#1、spring.web:
# 1.配置国际化的区域信息
# 2.静态资源策略(开启、处理链、缓存)
```

```

#开启静态资源映射规则
spring.web.resources.add-mappings=true

#设置缓存
spring.web.resources.cache.period=3600
##缓存详细合并项控制，覆盖period配置：
## 浏览器第一次请求服务器，服务器告诉浏览器此资源缓存7200秒，7200秒以内的所有此资源访问不用发
给服务器请求，7200秒以后发请求给服务器
spring.web.resources.cache.cachecontrol.max-age=7200
## 共享缓存
spring.web.resources.cache.cachecontrol.cache-public=true
#使用资源 last-modified 时间，来对比服务器和浏览器的资源是否相同没有变化。相同返回 304
spring.web.resources.cache.use-last-modified=true

#自定义静态资源文件夹位置
spring.web.resources.static-
locations=classpath:/a/,classpath:/b/,classpath:/static/

#2、spring.mvc
## 2.1. 自定义webjars路径前缀
spring.mvc.webjars-path-pattern=/wj/**
## 2.2. 静态资源访问路径前缀
spring.mvc.static-path-pattern=/static/**

```

代码方式

```

@Configuration //这是一个配置类
public class MyConfig implements WebMvcConfigurer {
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry){
        //保留以前规则
        //自己写新的规则。
        registry.addResourceHandler("/static/**")
            .addResourceLocations("classpath:/a/","classpath:/b/")
            .setCacheControl(CacheControl.maxAge(1180, TimeUnit.SECONDS));
    }
}

```

```

@Configuration // 这是一个配置类，给容器放一个WebMvcConfigurer组件，就能自定义底层
public class MyConfig implements WebMvcConfigurer {
    @Bean
    public WebMvcConfigurer webMvcConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addResourceHandlers(ResourceHandlerRegistry registry) {
                registry.addResourceHandler("/static/**")
                    .addResourceLocations("classpath:/static/")
                    .setCacheControl(CacheControl.maxAge(7200,
TimeUnit.SECONDS));
            }
        };
    }
}

```

路径匹配

Spring5.3 之后加入了更多的请求路径匹配的实现策略
以前只支持 `AntPathMatcher` 策略，现在提供了 `PathPatternParser` 策略。并且可以让我们指定到底使用哪种策略。

Ant风格路径用法

Ant 风格的路径模式语法具有以下规则：

- `*`：表示任意数量的字符。
- `?`：表示任意一个字符。
- `**`：表示任意数量的目录。
- `{}`：表示一个命名的模式占位符。
- `[]`：表示字符集合，例如`[a-z]`表示小写字母。

例如：

- `*.html` 匹配任意名称，扩展名为 `.html` 的文件。
- `/folder1/*/*.java` 匹配在 `folder1` 目录下的任意两级目录下的.java文件。
- `/folder2/**/*.jsp` 匹配在 `folder2` 目录下任意目录深度的.jsp文件。
- `/{type}/{id}.html` 匹配任意文件名为 `{id}.html`，在任意命名的 `{type}` 目录下的文件。

注意：Ant 风格的路径模式语法中的特殊字符需要转义，如：

- 要匹配文件路径中的星号，则需要转义为 `**`。
- 要匹配文件路径中的问号，则需要转义为 `\?\?`。

模式切换

AntPathMatcher 与 PathPatternParser

- `PathPatternParser` 在 jmh 基准测试下，有 6~8 倍吞吐量提升，降低 30%~40% 空间分配率
- `PathPatternParser` 兼容 `AntPathMatcher` 语法，并支持更多类型的路径模式
- `PathPatternParser` “`**`”多段匹配的支持仅允许在模式末尾使用

```
@GetMapping("/a*/b?/{p1:[a-f]+}")
public String hello(HttpServletRequest request,
                     @PathVariable("p1") String path) {
    log.info("路径变量p1: {}", path);
    //获取请求路径
    String uri = request.getRequestURI();
    return uri;
}
```

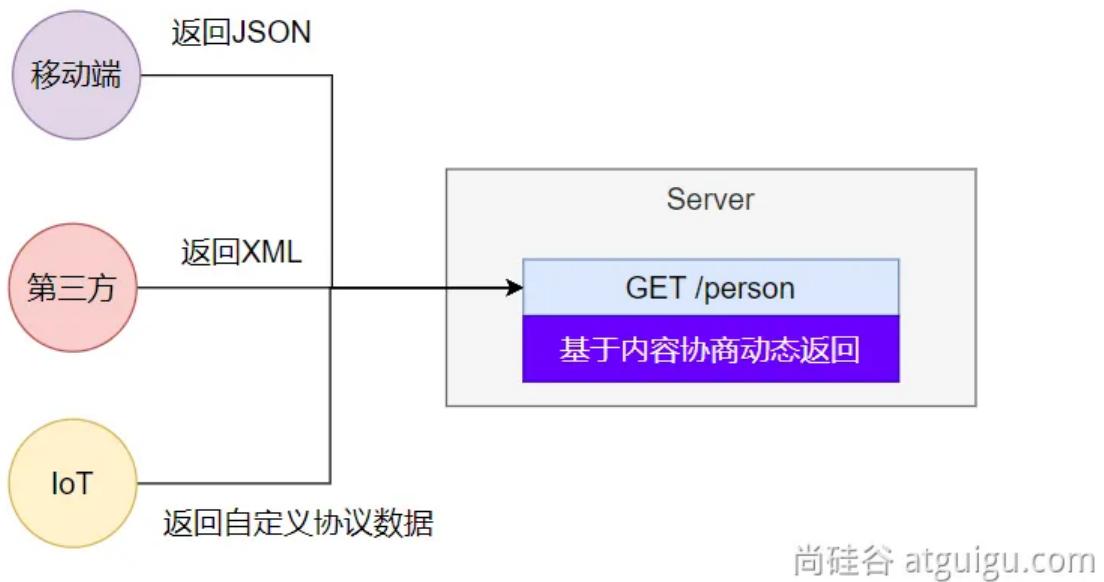
总结：

- 使用默认的路径匹配规则，是由 `PathPatternParser` 提供的
- 如果路径中间需要有 `**`，替换成 ant 风格路径

```
# 改变路径匹配策略:
# ant_path_matcher 老版策略;
# path_pattern_parser 新版策略;
spring.mvc.pathmatch.matching-strategy=ant_path_matcher
```

内容协商

一套系统适配多段数据返回



多端内容适配

默认规则

1. SpringBoot 多端内容适配

1. 基于请求头内容协商

1. 客户端向服务端发送请求，携带HTTP标准的Accept请求头
 1. Accept: `application/json`、`text/xml`、`text/yaml`

2. 服务端根据客户端请求头期望的数据类型进行动态返回

2. 基于请求参数内容协商

1. 发送请求 `GET/projects/spring-boot?format=json`
2. 匹配到 `GetMapping(*/projects/spring-boot*)`
3. 根据参数协商，优先返回json类型数据【需要开启参数匹配设置】
4. 发送请求 `GET/projects/spring-boot?format=xml` 优先返回xml类型数据

效果演示

请求同一个接口，可以返回json和xml不同格式数据

1. 引入支持写出xml内容依赖

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

2. 标注注解

```
@JacksonXmlRootElement // 可以写出为xml文档
@Data
public class Person {
    private Long id;
    private String userName;
    private String email;
    private Integer age;
}
```

3. 开启基于请求参数的内容协商

```
# 开启基于请求参数的内容协商功能。 默认参数名: format。 默认此功能不开启
spring.mvc.contentnegotiation.favor-parameter=true
# 指定内容协商时使用的参数名。默认是 format
spring.mvc.contentnegotiation.parameter-name=type
```

4. 效果

The screenshot shows two browser requests for the URL `localhost:8080/person?type=`:

- Top Request:** `localhost:8080/person?type=xml`. The response is an XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<Person>
    <id>1</id>
    <userName>zhangsan</userName>
    <password>123456</password>
    <email>123@1123</email>
    <age>18</age>
</Person>
```
- Bottom Request:** `localhost:8080/person?type=json`. The response is a JSON object:

```
{"id":1,"userName":"zhangsan","password":"123456","email":"123@1123","age":18}
```

配置协商规则与支持类型

1. 修改内容协商方式

```
# 使用参数进行内容协商
spring.mvc.contentnegotiation.favor-parameter=true
# 自定义参数名, 默认为format
spring.mvc.contentnegotiation.parameter-name=myparam
```

2. 大多数 MediaType 都是开箱即用的。也可以自定义内容类型, 如:

```
spring.mvc.contentnegotiation.media-types.yaml=text/yaml
```

自定义内容返回

增加yaml返回支持

添加依赖

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

把对象写出成YAML

```
public static void main(String[] args) throws JsonProcessingException {
    Person person = new Person();
    person.setUserName("zhangsan");
    person.setAge(18);
    person.setPassword("123456");
    person.setEmail("123@1123");
    person.setId(1);

    YAMLFactory factory = new
    YAMLFactory().disable(YAMLEncoder.Feature.WRITE_DOC_START_MARKER);
    ObjectMapper mapper = new ObjectMapper(factory);

    String s = mapper.writeValueAsString(person);
    System.out.println(s);
}
```

编写配置

```
#新增一种媒体类型
spring.mvc.contentnegotiation.media-types.yaml=text/yaml
```

增加 `HttpMessageConverter` 组件，专门负责把对象写出为yaml

```
@Configuration // 这是一个配置类，给容器放一个WebMvcConfigurer组件，就能自定义底层
public class MyConfig implements WebMvcConfigurer {
    @Bean
    public WebMvcConfigurer webMvcConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
                converters.add(new MyYamlHttpMessageConvert());
            }
        };
    }
}
```

思考：如何增加其他

- 配置媒体类型支持：
 - `spring.mvc.contentnegotiation.media-types.yaml=text/yaml`
- 编写对应的 `HttpMessageConverter`，要告诉Boot这个支持的媒体类型
 - 按照 `HttpMessageConvert` 的示例
- 把 `MessageConverter` 组件加入到底层

- 容器中放一个 `WebMvcConfigurer` 组件，并配置底层的 `MessageConverter`

HttpMessageConvert的示例

```

public class MyYamlHttpMessageConvert extends
AbstractHttpMessageConverter<Object> {

    private ObjectMapper objectMapper = null; // 把对象转成YAML

    public MyYamlHttpMessageConvert() {
        // 告诉SpringBoot,这个MessageConverter支持那种媒体类型 // 媒体类型
        super(new MediaType("text", "yaml", Charset.forName("UTF-8")));
    }

    YAMLFactory factory = new
    YAMLFactory().disable(YAMLEncoder.Feature.WRITE_DOC_START_MARKER);
    this.objectMapper = new ObjectMapper(factory);
}

@Override
protected boolean supports(Class<?> clazz) {
    // 只要是对象类型,不是基本类型
    return true;
}

@Override // @RequestBody
protected Object readInternal(Class<?> clazz, HttpInputMessage
inputMessage) throws IOException, HttpMessageNotReadableException {
    return null;
}

@Override // ResponseBody 把对象怎么写出去
protected void writeInternal(Object methodReturnValue, HttpOutputMessage
outputMessage) throws IOException, HttpMessageNotWritableException {

    // try-with写法,自动关流
    try (OutputStream os = outputMessage.getBody()) {
        this.objectMapper.writeValue(os, methodReturnValue);
    }
}
}

```

内容协商原理 - HttpMessageConverter

- `HttpMessageConverter` 怎么工作,何时工作?
- 定制 `HttpMessageConverter` 来实现多段内容协商
- 编写 `WebMvcConfigurer` 提供的 `configureMessageConverters` 底层, 修改底层的 `MessageConverter`

@ResponseBody 由 HttpMessageConverter 处理

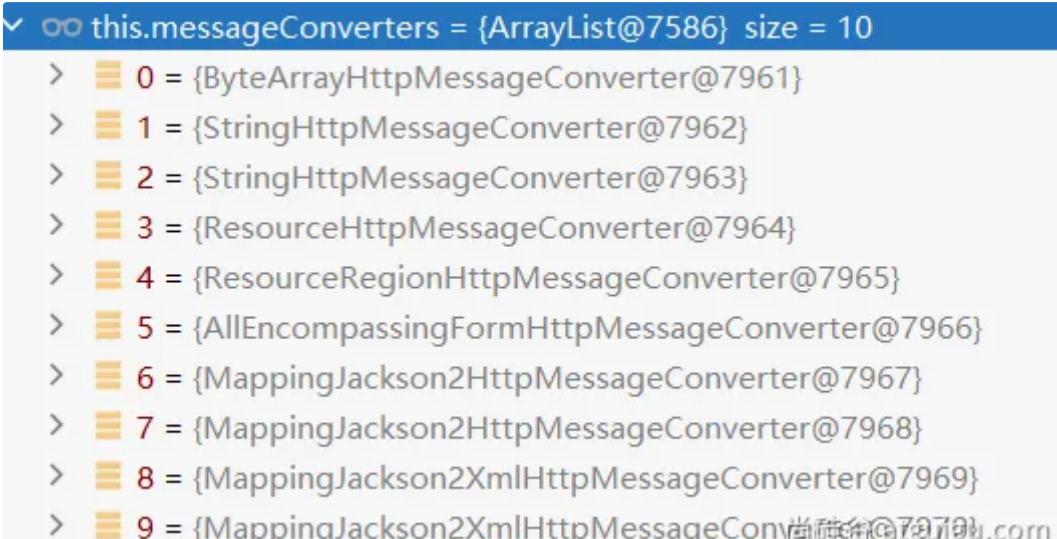
标注了 `ResponseBody` 的返回值将会由支持它的 `HttpMessageConverter` 写给浏览器

- 如果controller方法的返回值标注了 `@ResponseBody` 注解
 - 请求进来先来到 `DispatcherServlet` 的 `doDispatch()` 进行处理

2. 找到一个 `HandlerAdapter` 适配器，利用适配器执行目标方法
3. `RequestMappingHandlerAdapter` 来执行，调用 `invokeHandlerMethod()` 来执行目标方法
4. 目标方法执行之前，准备好两个东西
 1. `HandlerMethodArgumentResolver`：参数解析器，确定目标方法每个参数值
 2. `HandlerMethodReturnValueHandler`：返回值处理器，确定目标方法的返回值该怎么处理
5. `RequestMappingHandlerAdapter` 里面的 `invokeMethod()` 真正执行目标方法
6. 目标方法执行完成，会返回返回值对象
7. 找到一个合适的返回值处理器 `HandlerMethodReturnValueHandler`
8. 最终找到 `RequestResponseBodyMethodProcessor` 能处理 标注了 `@ResponseBody` 注解的方法
9. `RequestResponseBodyMethodProcessor` 调用 `writeWithMessageConverters`，利用 `MessageConverter` 把返回值写出去

上面解释：`@ResponseBody` 由 `HttpMessageConverter` 处理

2. `HttpMessageConverter` 会先进行内容协商

1. 遍历所有的 `MessageConverter` 看谁支持这种 **内容类型的数据**
2. 默认 `MessageConverter` 有以下
3. 

```
this.messageConverters = {ArrayList@7586} size = 10
  > 0 = {ByteArrayHttpMessageConverter@7961}
  > 1 = {StringHttpMessageConverter@7962}
  > 2 = {StringHttpMessageConverter@7963}
  > 3 = {ResourceHttpMessageConverter@7964}
  > 4 = {ResourceRegionHttpMessageConverter@7965}
  > 5 = {AllEncompassingFormHttpMessageConverter@7966}
  > 6 = {MappingJackson2HttpMessageConverter@7967}
  > 7 = {MappingJackson2HttpMessageConverter@7968}
  > 8 = {MappingJackson2XmlHttpMessageConverter@7969}
  > 9 = {MappingJackson2XmlHttpMessageConverter@7970}
```
4. 最终因为要 `json` 所以 `MappingJackson2HttpMessageConverter` 支持写出 `json`
5. `jackson` 用 `ObjectMapper` 把对象写出去

WebMvcAutoConfiguration 提供几种默认 `HttpMessageConverters`

- `EnableWebMvcConfiguration` 通过 `addDefaultHttpMessageConverters` 添加了默认的 `MessageConverter`；如下：
 - `ByteArrayHttpMessageConverter`：支持字节数据读写
 - `StringHttpMessageConverter`：支持字符串读写
 - `ResourceHttpMessageConverter`：支持资源读写
 - `ResourceRegionHttpMessageConverter`：支持分区资源写出
 - `AllEncompassingFormHttpMessageConverter`：支持表单xml/json读写
 - `MappingJackson2HttpMessageConverter`：支持请求响应体Json读写

默认8个：

```

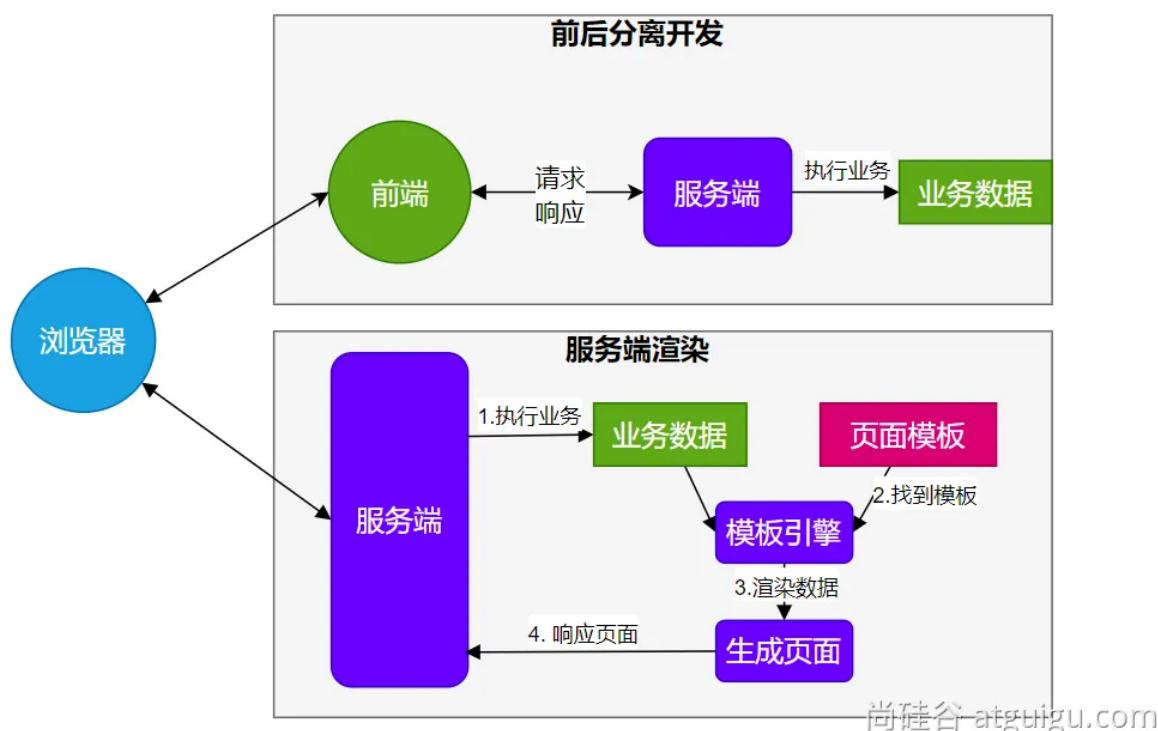
    > 0 = {ByteArrayHttpMessageConverter@7605}
    > 1 = {StringHttpMessageConverter@7606}
    > 2 = {StringHttpMessageConverter@7607}
    > 3 = {ResourceHttpMessageConverter@7608}
    > 4 = {ResourceRegionHttpMessageConverter@7609}
    > 5 = {AllEncompassingFormHttpMessageConverter@7610}
    > 6 = {MappingJackson2HttpMessageConverter@7611}
    > 7 = {MappingJackson2HttpMessageConverter@7612}

```

系统提供默认的MessageConverter 功能有限，仅用于json或者普通返回数据。额外增加新的内容协商功能，必须增加新的 [HttpMessageConverter](#)

模板引擎

由于SpringBoot使用了嵌入式 Servlet 容器，所以 JSP 默认是不能使用的。
如果需要使用服务端页面渲染，优先考虑使用模板引擎



模板引擎页面默认放在 [src/main/resources/templates](#)

SpringBoot 包含以下模板引擎的自动配置

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

Thymeleaf官网: <https://www.thymeleaf.org/>

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
th:href="@{/css/gtvg.css}" />
</head>
<body>
    <p th:text="#{home.welcome}">welcome to our grocery store!</p>
</body>
</html>

```

Thymeleaf 整合

```

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

```

自动配置原理

1. 开启了 `org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration` 自动配置
2. 属性绑定在 `ThymeleafProperties` 中，对应配置文件 `spring.thymeleaf` 内容
3. 所有的模板页面默认在 `classpath:/templates` 文件夹下
4. 默认效果
5. 1. 所有的模板页面在 `classpath:/templates/` 下面找
 2. 找后缀名为 `.html` 的页面

基础语法

核心用法

`th:xxx` : 动态渲染指定的 html 标签属性值、或者 th 指令（遍历、判断等）

- `th:text`: 标签体内文本值渲染
 - `th:utext`: 不会转义，显示为html原本的样子
- `th:属性`: 标签指定属性渲染
- `th:attr`: 标签任意属性渲染
- `th:if`: `th:each ...`: 其他th命令
- 例如:

```

<p th:text="${content}">原内容</p>
<a th:href="${url}">登录</a>


```

表达式：用来动态取值

- `{}: 变量取值；使用model共享给页面的值都直接用{}；`
- `@{}: url路径；`
- `#{}: 国际化消息`

- `~{}`：片段引用
- `*{}`：变量选择：需要配合th:object绑定对象

系统工具&内置对象：详细文档

- `param`：请求参数对象
- `session`：session对象
- `application`：application对象
- `#execInfo`：模板执行信息
- `#messages`：国际化消息
- `#uris`：uri/url工具
- `#conversions`：类型转换工具
- `#dates`：日期工具，是 `java.util.Date` 对象的工具类
- `#calendars`：类似`#dates`，只不过是 `java.util.Calendar` 对象的工具类
- `#temporals`：JDK8+ `java.time` API 工具类
- `#numbers`：数字操作工具
- `#strings`：字符串操作
- `#objects`：对象操作
- `#bools`：bool操作
- `#arrays`：array工具
- `#lists`：list工具
- `#sets`：set工具
- `#maps`：map工具
- `#aggregates`：集合聚合工具（sum、avg）
- `#ids`：id生成工具

语法示例

表达式：

- 变量取值： `${...}`
- url 取值： `@{...}`
- 国际化消息： `#{...}`
- 变量选择： `*{...}`
- 片段引用： `~{...}`

常见：

- 文本： `'one text' , 'another one!' ,...`
- 数字： `0, 34, 3.0, 12.3 ,...`
- 布尔： `true, false`
- null： `null`
- 变量名： `one, sometext, main ...`

文本操作：

- 拼串： `+`
- 文本替换： `| The name is ${name} |`

布尔操作：

- 二进制运算： `and, or`
- 取反： `!, not`

比较运算:

- 比较: >, <, <=, >= (gt, lt, ge, le)
- 等值运算: ==, != (eq, ne)

条件运算:

- if-then: (if)?(then)
- if-then-else: (if)?(then):(else)
- default: (value)?:(defaultValue)

特殊语法:

- 无操作: _

所有以上都可以嵌套组合

```
'User is of type ' + (${user.isAdmin()} ? 'Administrator' : (${user.type} ?: 'Unknown'))
```

属性设置

1. **th:href="@{/product/list}"**
2. **th:attr="class=\${active}"**
3. **th:attr="src=@{/images/gtvgllogo.png},title=\${logo},alt=#{logo}"**
4. **th:checked="\${user.active}"**

```
<p th:text="${content}">原内容</p>
<a th:href="${url}">登录</a>

```

遍历

语法: **th:each="元素名,迭代状态 : \${集合}"**

```
<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>
```

iterStat 有以下属性:

- index: 当前遍历元素的索引, 从0开始
- count: 当前遍历元素的索引, 从1开始
- size: 需要遍历元素的总数量
- current: 当前正在遍历的元素对象
- even/odd: 是否偶数/奇数行
- first: 是否第一个元素
- last: 是否最后一个元素

判断

th:if

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

th:switch

```
<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

- 片段
- 遍历
- 判断

```
<ul>  
    <li th:each="item : ${items}" th:text="${item.description}">Item description  
here...</li>  
</ul>
```

Order	Feature	Attributes
1	片段包含	th:insert th:replace
2	遍历	th:each
3	判断	th:if th:unless th:switch th:case
4	定义本地变量	th:object th:with
5	通用方式属性修改	th:attr th:attrprepend th:attrappend
6	指定属性修改	th:value th:href th:src ...
7	文本值	th:text th:utext
8	片段指定	th:fragment
9	片段移除	th:remove

行内写法

[[...]] or [(...)]

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

变量选择

```
<div th:object="${session.user}">
    <p>Name:
        <span th:text="*{firstName}">Sebastian</span>.
    </p>
    <p>Surname:
        <span th:text="*{lastName}">Pepper</span>.
    </p>
    <p>Nationality:
        <span th:text="*{nationality}">Saturn</span>.
    </p>
</div>
```

等同于：

```
<div>
    <p>Name:
        <span th:text="${session.user.firstName}">Sebastian</span>.
    </p>
    <p>Surname:
        <span th:text="${session.user.lastName}">Pepper</span>.
    </p>
    <p>Nationality:
        <span th:text="${session.user.nationality}">Saturn</span>.
    </p>
</div>
```

模板布局

- 定义模板： `th:fragment`
- 引用模板： `~{templatename::selector}`
- 插入模板： `th:insert`、`th:replace`

```
<footer th:fragment="copy">&copy; 2011 The Good Thymes Virtual Grocery</footer>

<body>
    <div th:insert="~{footer :: copy}"></div>
    <div th:replace="~{footer :: copy}"></div>
</body>
<body>
    结果：
    <body>
        <div>
            <footer>&copy; 2011 The Good Thymes Virtual Grocery</footer>
        </div>

        <footer>&copy; 2011 The Good Thymes Virtual Grocery</footer>
    </body>
</body>
```

devtools

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

修改页面后： **ctrl+F9** 刷新效果；

java代码的修改，如果 **devtools** 热启动了，可能会引起一些bug，难以排查

国际化

国际化的自动配置参照 **MessageSourceAutoConfiguration**

实现步骤

1. Spring Boot 在类路径根下查找 **messages** 资源绑定文件。文件名为：**messages.properties**
2. 多语言可以定义多个消息文件，命名为 **messages_区域代码.properties**。如
 1. **messages.properties**：默认
 2. **messages_zh_CN.properties**：中文环境
 3. **messages_en_US.properties**：英语环境
3. 在程序中可以自动注入 **MessageSource** 组件，获取国际化的配置项值
4. 在页面中可以使用表达式 **#{}** 获取国际化的配置项值

```
@Autowired //国际化取消息用的组件
MessageSource messageSource;
@GetMapping("/haha")
public String haha(HttpServletRequest request) {
    Locale locale = request.getLocale();
    // 利用代码的方法获取国际化配置文件中指定的配置项的值
    String login = messageSource.getMessage("login", null, locale);
    return login;
}
```

5. 通过 **spring.messages.basename=i18N.login** 可以更改国际化配置文件位置和名称

区域信息解析器

通过配置类来配置区域信息解析器

MyLocalResolver.class

```
@Configuration
public class MyLocalResolver implements LocaleResolver {
    // 解析请求
    @Override
    public Locale resolveLocale(HttpServletRequest request) {
        // 获取页面手动切换时传递的语言参数 language
        String language = request.getParameter("language");
        // 获取请求头中自动传递的语言参数Accept-language
        String header = request.getHeader("Accept-Language");

        Locale locale = null;
        // 如果手动切换参数不为空，根据参数手动进行语言切换，否则就根据默认请求头进行切换
        if (StringUtils.hasText(language)) {
            // 访问地址： login?language=zh_CN
            // split[0]=zh, split[1]=CN
            String[] split = language.split("_");
```

```

        locale = new Locale(split[0], split[1]);
    } else {
        // zh-CN,zh;q=0.9,en;q=0.8
        String[] splits = header.split(", ");
        String[] split = splits[0].split("-");
        locale = new Locale(splits[0], splits[1]);
    }
    return locale;
}

@Override
public void setLocale(HttpServletRequest request, HttpServletResponse
response, Locale locale) {

}

@Bean
public LocaleResolver localeResolver() {
    return new MyLocaleResolver();
}
}

```

i18N

login.properties
login_en_US.properties
login_zh_CN.properties

```

login.tip=请登录
login.username=请输入用户名
login.password=请输入密码
login.rememberme=记住我
login.button=登陆

```

```

login.tip=Please sign in
login.username=enter username
login.password=enter password
login.rememberme=Remember me
login.button=Login

```

```

login.tip=请登录
login.username=请输入用户名
login.password=请输入密码
login.rememberme=记住我
login.button=登陆

```

页面进行请求

```

<a th:href="@{/login(language='zh_CN')}" class="btn btn-sm">中文</a>
<a th:href="@{/login(language='en_us')}" class="btn btn-sm">英文</a>

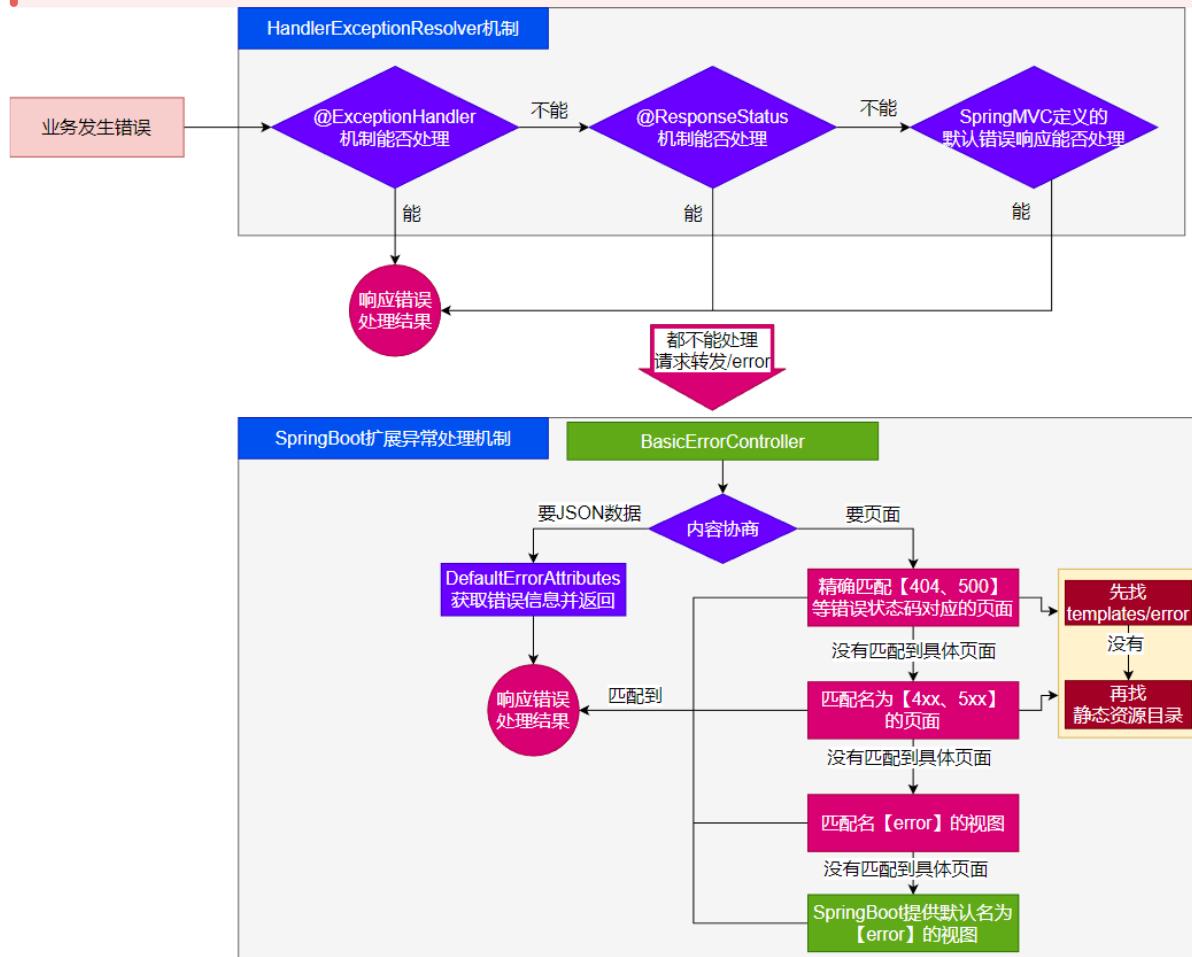
```

错误处理

默认机制

错误处理的自动配置都在 `ErrorMvcAutoConfiguration` 中，两大核心机构

1. SpringBoot 会自适应处理错误，响应页面或JSON数据
2. SpringMVC的错误处理机制依然保留，MVC处理不了，才会交给boot进行处理



1. 发生错误以后，转发给/error路径，SpringBoot在底层写好一个 `BasicErrorHandler` 的组件，专门处理这个请求

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

@RequestMapping // 返回 ResponseEntity, json
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
    HttpStatus status = getStatus(request);
    if (status == HttpStatus.NO_CONTENT) {
```

```

        return new ResponseEntity<>(status);
    }
    Map<String, Object> body = getErrorAttributes(request,
getErrorAttributeOptions(request, MediaType.ALL));
    return new ResponseEntity<>(body, status);
}

```

- 错误页面是这么解析到的

```

//1. 解析错误的视图地址
ModelAndView modelAndView = resolveErrorView(request, response, status, model);
//2. 如果解析不到错误页面的地址, 默认的错误页就是 error
return (modelAndView != null) ? modelAndView : new ModelAndView("error",
model);

```

容器中专门有一个错误视图解析器

```

@Bean
@ConditionalOnBean(DispatcherServlet.class)
@ConditionalOnMissingBean(ErrorViewResolver.class)
DefaultErrorViewResolver conventionErrorViewResolver() {
    return new DefaultErrorViewResolver(this.applicationContext,
this.resources);
}

```

SpringBoot解析错误页得默认规则

```

@Override
public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus
status, Map<String, Object> model) {
    ModelAndView modelAndView = resolve(String.valueOf(status.value()), model);
    if (modelAndView == null && SERIES_VIEWS.containsKey(status.series())) {
        modelAndView = resolve(SERIES_VIEWS.get(status.series()), model);
    }
    return modelAndView;
}

private ModelAndView resolve(String viewName, Map<String, Object> model) {
    String errorviewName = "error/" + viewName;
    TemplateAvailabilityProvider provider =
this.templateAvailabilityProviders.getProvider(errorviewName,

        this.applicationContext);
    if (provider != null) {
        return new ModelAndView(errorviewName, model);
    }
    return resolveResource(errorviewName, model);
}

private ModelAndView resolveResource(String viewName, Map<String, Object>
model) {
    for (String location : this.resources.getStaticLocations()) {
        try {
            Resource resource = this.applicationContext.getResource(location);
            resource = resource.createRelative(viewName + ".html");
            if (resource.exists()) {
                return new ModelAndView(new HtmlResourceView(resource), model);
            }
        }
        catch (Exception ex) {

```

```
        // Ignore
    }
}
return null;
}
```

容器中有一个默认得名为 error 的 view; 提供了默认白页功能

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

封装了JSON格式的错误信息

```
@Bean
@ConditionalOnMissingBean(value = ErrorAttributes.class, search =
SearchStrategy.CURRENT)
public DefaultErrorAttributes errorAttributes() {
    return new DefaultErrorAttributes();
}
```

规则:

1. 解析一个错误页
 1. 如果发生了500、404、503、403..这些错误
 1. 如果有模板引擎, 默认在 `classpath:/templates/error/精确码.html`
 2. 如果没有模板引擎, 在静态资源文件夹下找 `精确码.html`
 2. 如果匹配不到 `精确码.html` 这些精确的错误页, 就去找 `5xx.html`, `4xx.html` 模糊匹配
 1. 如果有模板引擎, 默认在 `classpath:/templates/error/5|4xx.html`
 2. 如果没有模板引擎, 在静态资源文件夹下找 `5|4xx.html`
2. 如果模板引擎路径 `templates` 下由 `error.html` 页面, 就直接渲染

自定义错误响应

1. 自定义json响应

使用`@ControllerAdvice + @ExceptionHandler`进行统一异常处理

```
// 全局异常处理器, 用于统一处理特定类型的异常
@ControllerAdvice
public class GlobalExceptionHandler {
    // 处理ArithmaticException类型的异常
    @ExceptionHandler(ArithmaticException.class)
    public String handleArithmaticException(ArithmaticException e) {
        // 返回异常处理结果, 这里是一个包含异常类型和消息的字符串
        return "ArithmaticException, message: " + e.getMessage();
    }
}
```

2. 自定义页面响应

根据boot的错误页面规则, 自定义页面模板

最佳实践

- 前后分离
 - 后台发生所有的错误，`@ControllerAdvice + @ExceptionHandler` 进行统一处理
- 服务端页面渲染
 - HTTP码错误
 - 给 `classpath:/templates/error/` 下面放一个精确码的错误页面。`500.html`, `404.html`
 - 给 `classpath:/templates/error/` 下面，放通用模糊匹配的错误码页面
 - 发生业务错误
 - 核心业务，每一种错误，都应该代码控制，跳转到自己定制的错误页面
 - 通用业务，`classpath:/templates/error/`

页面，JSON，可用的Model数据如下

```
▽ □ model = {Collections$UnmodifiableMap@6718}  size = 6
  > □ "timestamp" -> {Date@6738} "Sat Mar 16 21:52:14 CST 2024"
  > □ "status" -> {Integer@6740} 404
  > □ "error" -> "Not Found"
  > □ "trace" -> "org.springframework.web.servlet.resource.NoResourceNotFoundException: No static ... View
  > □ "message" -> "No static resource asfasdf."
  > □ "path" -> "/asfasdf"

Set value F2  Create renderer
```

嵌入式容器

嵌入式容器：管理、运行Servlet组件（Servlet、Filter、Listener）的环境，一般指服务器

自动配置原理

- SpringBoot 默认嵌入 Tomcat 作为 Servlet 容器
- 自动配置类是 `ServletWebServerFactoryAutoConfiguration`、`EmbeddedWebServerFactoryCustomizerAutoConfiguration`
- 自动配置类开始分析功能。`xxxAutoConfiguration`

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

1. `ServletWebServerFactoryAutoConfiguration` 自动配置了嵌入式容器场景
2. 绑定了 `ServerProperties` 配置类，所有和服务器有关配置 `server`
3. `ServletWebServerFactoryAutoConfiguration` 导入了嵌入式的三大服务器 `Tomcat`、`Jetty`、`Undertow`
 1. 导入 `Tomcat`、`Jetty`、`Undertow` 都有条件注解。系统中有这个类才行（也就是导了包）

2. 默认 Tomcat 配置生效。给容器中放 TomcatServletWebServerFactory
3. 都给容器中 ServletWebServerFactory 放了一个 web服务器工厂 (造web服务器的)
4. web服务器工厂都有一个功能， getWebServer 获取web服务器
5. TomcatServletWebServerFactory 创建了 tomcat
6. ServletWebServerFactory 什么时候会创建 webServer 出来
7. `ServletWebServerApplicationContext` ioc容器，启动的时候会调用创建web服务器
8. Spring容器刷新 (启动) 的时候，会预留一个时机，刷新子容器。onRefresh()
9. refresh() 容器刷新 十二大步的刷新子容器会调用 onRefresh()；

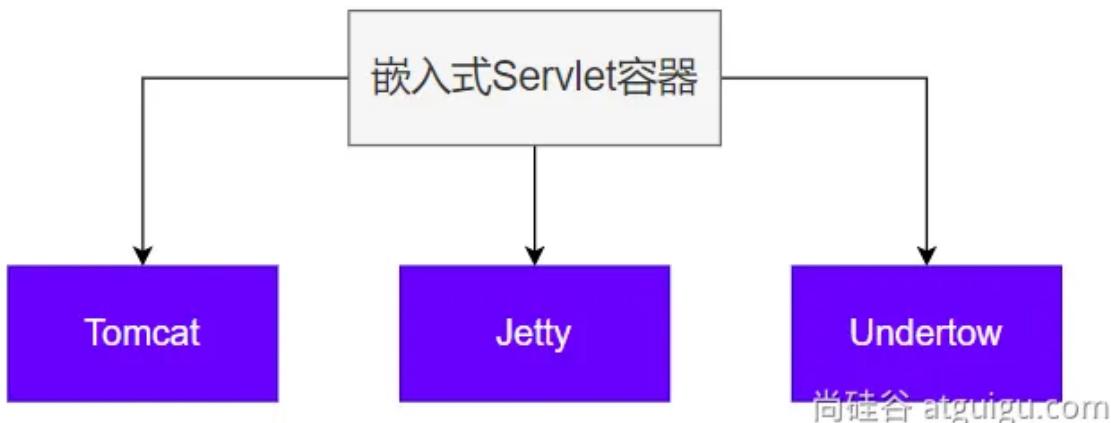
```

@Override
protected void onRefresh() {
    super.onRefresh();
    try {
        createWebServer();
    }
    catch (Throwable ex) {
        throw new ApplicationContextException("Unable to start web server",
ex);
    }
}

```

Web场景的Spring容器启动，在onRefresh的时候，会调用创建web服务器的方法。
Web服务器的创建是通过WebServerFactory搞定的。容器中又会根据导了什么包条件注解，启动相关的服务器配置，默认 EmbeddedTomcat 会给容器中放一个 TomcatServletWebServerFactory，导致项目启动，自动创建出Tomcat。

自定义



切换服务器

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <!-- Exclude the Tomcat dependency -->
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>

```

```
<!-- Use Jetty instead -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

最佳实践

用法：

- 修改 `server` 下的相关配置就可以修改服务器参数
- 通过给容器中放一个 `ServletWebServerFactory`，来禁用掉SpringBoot默认放的服务器工厂，实现自定义嵌入任意服务器

全面接管SpringMVC

- SpringBoot 默认配置好了 SpringMVC 的所有常见特性
- 如果我们需要全面接管SpringMVC的所有配置并禁用默认配置，仅需要编写一个 `WebMvcConfigurer` 配置类，并标注 `@EnableWebMvc` 即可
- 全手动模式
 - `@EnableWebMvc`：禁用默认配置
 - `WebMvcConfigurer` 组件：定义MVC的底层行为

WebMvcAutoConfiguration 到底自动配置了那些规则

SpringMVC自动配置场景给我们配置了如下所有默认行为

1. `WebMvcAutoConfiguration` web场景的自动配置类
 1. 支持 RESTful 的 filter: `HiddenHttpMethodFilter`
 2. 支持非 POST 请求，请求体携带数据: `FormContentFilter`
 3. 导入 `EnableWebMvcConfiguration`:
 1. `RequestMappingHandlerAdapter`
 2. `welcomePageHandlerMapping`: 欢迎页功能支持（模板引擎目录、静态资源目录放 `index.html`），项目访问 / 就默认展示这个页面
 3. `RequestMappingHandlerMapping`: 找每个请求由谁处理的映射关系
 4. `ExceptionHandlerExceptionResolver`: 默认的异常解析器
 5. `LocaleResolver`: 国际化解析器
 6. `ThemeResolver`: 主题解析器
 7. `FlashMapManager`: 临时数据共享
 8. `FormattingConversionService`: 数据格式化、类型转换
 9. `validator`: 数据校验 `JSP303` 提供的数据校验功能
 10. `webBindingInitializer`: 请求参数的封装与绑定
 11. `ContentNegotiationManager`: 内容协商管理器
 4. `WebMvcAutoConfigurationAdapter` 配置生效，它是一个 `WebConfigurer`，定义mvc底层组件
 1. 定义好 `WebMvcConfigurer` 底层组件默认功能；所有功能详见列表、
 2. 视图解析器: `InternalResourceViewResolver`
 3. 视图解析器: `BeanNameViewResolver`，视图名（controller方法的返回值字符串）就是组件名

4. 内容协商解析器: `ContentNegotiatingViewResolver`
5. 请求上下文过滤器: `RequestContextFilter`: 任意位置直接获取当前请求
6. 静态资源链规则: `ResourceChainCustomizerConfiguration`
7. `problemDetailsExceptionHandler`: 错误详情
 1. SpringMVC内部场景异常被它捕获:
8. 定义了MVC默认的底层行为: `WebMvcConfigurer`

@EnableWebMvc禁用默认行为

1. `@EnableWebMvc` 给容器中导入 `DelegatingWebMvcConfiguration` 组件,
他是 `WebMvcConfigurationSupport`
1. `WebMvcAutoConfiguration` 有一个核心的条件注解,
`@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)`, 容器中没有
`WebMvcConfigurationSupport`, `WebMvcAutoConfiguration` 才生效.
2. `@EnableWebMvc` 导入 `WebMvcConfigurationSupport` 导致 `WebMvcAutoConfiguration` 失效。导
致禁用了默认行为

- `@EnableWebMvc` 禁用了 Mvc 的自动配置
- `WebMvcConfigurer` 定义 SpringMVC 底层组件的功能类

WebMvcConfigurer 功能

定义扩展 SpringMVC 底层功能

提供方法	核心参数	功能	默认
addFormatters	FormatterRegistry	格式化器： 支持属性上 @NumberFormat 和 @DatetimeFormat 的数据类型转换	GenericConversionService
getValidator	无	数据校验： 校验 Controller 上使用 @Valid 标注的参数合法性。需要导入 starter-validator	无
addInterceptors	InterceptorRegistry	拦截器： 拦截收到的所有请求	无
configureContentNegotiation	ContentNegotiationConfigurer	内容协商： 支持多种数据格式返回。需要配合支持这种类型的 HttpMessageConverter	支持 json
configureMessageConverters	List<HttpMessageConverter<? >>	消息转换器： 标注 @ResponseBody 的返回值会利用 MessageConverter 直接写出去	8 个，支持 byte, string, multipart, resource, json
addViewControllers	ViewControllerRegistry	视图映射： 直接将请求路径与物理视图映射。用于无 Java 业务逻辑的直接视图页渲染	无 mvc:view-controller
configureViewResolvers	ViewResolverRegistry	视图解析器： 逻辑视图转为物理视图	ViewResolverComposite
addResourceHandlers	ResourceHandlerRegistry	静态资源处理： 静态资源路径映射、缓存控制	ResourceHandlerRegistry
configureDefaultServletHandling	DefaultServletHandlerConfigurer	默认 Servlet： 可以覆盖 Tomcat 的 DefaultServlet。让 DispatcherServlet 拦截/	无
configurePathMatch	PathMatchConfigurer	路径匹配： 自定义 URL 路径匹配。可以自动为所有路径加上指定前缀，比如 /api	无
configureAsyncSupport	AsyncSupportConfigurer	异步支持：	TaskExecutionAutoConfiguration
addCorsMappings	CorsRegistry	跨域：	无
addArgumentResolvers	List	参数解析器：	mvc 默认提供
addReturnValueHandlers	List	返回值解析器：	mvc 默认提供
configureHandlerExceptionResolvers	List	异常处理器：	默认 3 个 ExceptionHandlerExceptionResolver ResponseStatusExceptionResolver DefaultHandlerExceptionResolver
getMessageCodesResolver	无	消息码解析器： 国际化使用	无

最佳实践

SpringBoot 已经默认配置好了 Web 开发场景常用功能。我们直接使用即可。

三种方式

方式	用法	效果	
全自动	直接编写控制器逻辑		全部使用自动配置默认效果
手自一体	@Configuration + 配置 **WebMvcConfigurer** + 配置 WebMvcRegistrations	不要标注 @**EnableWebMvc**	保留自动配置效果 手动设置部分功能 定义 MVC 底层组件
全手动	@Configuration + 配置 **WebMvcConfigurer**	标注 @**EnableWebMvc**	禁用自动配置效果 全手动设置

总结：

给容器中写一个配置类 **@Configuration** 实现 **WebMvcConfigurer** 但是不要标注

@EnableWebMvc 注解，实现手自一体的效果。

两种模式

- 1、前后分离模式：`@RestController` 响应JSON数据
- 2、前后不分离模式：`@Controller + Thymeleaf模板引擎`

Web新特性

Problemdetails

RFC 7807: <https://www.rfc-editor.org/rfc/rfc7807>

错误信息返回新格式

原理

```
@Configuration(proxyBeanMethods = false)
//配置过一个属性 spring.mvc.problemdetails.enabled=true
@ConditionalOnProperty(prefix = "spring.mvc.problemdetails", name = "enabled",
havingValue = "true")
static class ProblemDetailsErrorHandlerConfiguration {
    @Bean
    @ConditionalOnMissingBean(ResponseEntityExceptionHandler.class)
    ProblemDetailsExceptionHandler problemDetailsExceptionHandler() {
        return new ProblemDetailsExceptionHandler();
    }
}
```

1. `ProblemDetailsExceptionHandler`是一个`@ControllerAdvice`集中处理系统异常
2. 处理以下异常。如果系统出现以下异常，会被SpringBoot支持以 RFC 7807 规范方式返回错误数据

```
@ExceptionHandler({
    HttpRequestMethodNotSupportedException.class, //请求方式不支持
    HttpMediaTypeNotSupportedException.class,
    HttpMediaTypeNotAcceptableException.class,
    MissingPathVariable.class,
    MissingServletRequestParameterException.class,
    MissingServletRequestPartException.class,
    ServletRequestBindingException.class,
    MethodArgumentNotValidException.class,
    NoHandlerFoundException.class,
    AsyncRequestTimeoutException.class,
    ErrorResponseException.class,
    ConversionNotSupportedException.class,
    TypeMismatchException.class,
    HttpResponseMessageNotReadableException.class,
    HttpResponseMessageNotWritableException.class,
    BindException.class
})
```

效果：

默认相应错误的 json。状态码 405

```
{  
    "timestamp": "2023-04-18T11:13:05.515+00:00",  
    "status": 405,  
    "error": "Method Not Allowed",  
    "trace": "org.springframework.web.HttpRequestMethodNotSupportedException:  
Request method 'POST' is not supported\r\n\tat .....略",  
    "message": "Method 'POST' is not supported.",  
    "path": "/list"  
}
```

开启ProblemDetails返回, 使用新的MediaType

Content-Type: application/problem+json + 额外扩展返回

```
{  
    "type": "about:blank",  
    "title": "Method Not Allowed",  
    "status": 405,  
    "detail": "Method 'POST' is not supported.",  
    "instance": "/list"  
}
```

函数式Web

SpringMVC 5.2 以后允许我们使用的函数式的方式, 定义Web的请求处理流程

函数式接口

Web请求处理的方式:

1. @Controller + @RequestMapping: 耦合式 (路由、业务耦合)
2. 函数式Web: 分离式 (路由、业务分离)

场景

场景: User RESTful - CRUD

- GET /user/1 获取1号用户
- GET /users 获取所有用户
- POST /user **请求体**携带JSON, 新增一个用户
- PUT /user/1 **请求体**携带JSON, 修改1号用户
- DELETE /user/1 **删除**1号用户

核心类

- RouterFunction
- RequestPredicate
- ServerRequest
- ServerResponse

示例

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import org.springframework.web.servlet.function.RequestPredicate;
import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.ServerResponse;

import static
org.springframework.web.servlet.function.RequestPredicates.accept;
import static org.springframework.web.servlet.function.RouterFunctions.route;

@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {

    private static final RequestPredicate ACCEPT_JSON =
accept(MediaType.APPLICATION_JSON);

    @Bean
    public RouterFunction<ServerResponse> routerFunction(MyUserHandler
userHandler) {
        return route()
            .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
            .GET("/{user}/customers", ACCEPT_JSON,
userHandler::getUserCustomers)
            .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
            .build();
    }

}
```

```
import org.springframework.stereotype.Component;
import org.springframework.web.function.ServerRequest;
import org.springframework.web.function.ServerResponse;

@Component
public class MyUserHandler {

    public ServerResponse getUser(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

    public ServerResponse getUserCustomers(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

    public ServerResponse deleteUser(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }
}
```

数据访问

整合SSM场景

SpringBoot 整合 Spring、SpringMVC、MyBatis 进行数据访问场景开发

创建SSM整合项目

```
<!-- https://mvnrepository.com/artifact/org.mybatis.spring.boot/mybatis-spring-boot-starter -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>3.0.1</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

配置数据源

```
spring.datasource.url=jdbc:mysql://192.168.200.100:3306/demo
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.type=com.zaxxer.hikari.HikariDataSource
```

配置MyBatis

```
#指定mapper映射文件位置
mybatis.mapper-locations=classpath:/mapper/*.xml
#参数项调整
mybatis.configuration.map-underscore-to-camel-case=true
```

CRUD编写

- 编写Bean
- 编写Mapper
- 使用 mybatisx 插件，快速生成MapperXML
- 测试CRUD

自动配置原理

SSM整合总结：

1. 导入 mybatis-spring-boot-starter
2. 配置数据源信息
3. 配置mybatis的 **mapper接口扫描** 与 **xml映射文件扫描**

4. 编写bean, mapper, 生成xml, 编写sql 进行crud。事务等操作依然和Spring中用法一样

5. 效果：

6. 1. 所有sql写在xml中

2. 所有 mybatis 配置 写在 application.properties 下面

- jdbc 场景的自动配置：
 - mybatis-spring-boot-starter 导入 spring-boot-starter-jdbc , jdbc是操作数据库的场景
 - Jdbc 场景的几个自动配置
 - org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
- ◦ ◦ 数据源的自动配置
 - 所有和数据源有关的配置都绑定在 DataSourceProperties
 - 默认使用 HikariDataSource
- ◦ ◦ org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration
- ◦ ◦ 给容器中放了 JdbcTemplate 操作数据库
- ◦ ◦ org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration
- org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration
- ◦ ◦ 基于XA二阶提交协议的分布式事务数据源
- ◦ ◦ org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration
- n
- ◦ ◦ 支持事务
- 具有的底层能力：数据源、 JdbcTemplate 、事务
- MyBatisAutoConfiguration : 配置了MyBatis的整合流程
 - mybatis-spring-boot-starter 导入 mybatis-spring-boot-autoconfigure (mybatis的自动配置包)
 - 默认加载两个自动配置类
 - org.mybatis.spring.boot.autoconfigure.MybatisLanguageDriverAutoConfiguration
 - org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration
 - 在数据源配置好之后才配置
 - 给容器中 SqlSessionFactory 组件。创建和数据库的一次会话
 - 给容器中 SqlSessionTemplate 组件。操作数据库
 - MyBatis的所有配置绑定在 MybatisProperties
 - 每个Mapper接口的代理对象是怎么创建放到容器中。详见@MapperScan原理：
 - 利用 @Import(MapperScannerRegistrar.class) 批量给容器中注册组件。解析指定的包路径里面的每一个类，为每一个Mapper接口类，创建Bean定义信息，注册到容器中。

如何分析哪个场景导入以后，开启了哪些自动配置类。

找： classpath:/META-

INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports 文件中配置的所有值，就是要开启的自动配置类，但是每个类可能有条件注解，基于条件注解判断哪个自动配置类生效了。

快速定位生效的配置

```
#开启调试模式，详细打印开启了哪些自动配置
debug=true
# Positive (生效的自动配置)    Negative (不生效的自动配置)
```

拓展其他数据源

1. Druid 数据源

暂不支持 SpringBoot3

- 导入 `druid-starter`
- 写配置
- 分析自动配置了哪些东西，怎么用

Druid官网：<https://github.com/alibaba/druid>

```
#数据源基本配置
spring.datasource.url=jdbc:mysql://192.168.200.100:3306/demo
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource

# 配置StatFilter监控
spring.datasource.druid.filter.stat.enabled=true
spring.datasource.druid.filter.stat.db-type=mysql
spring.datasource.druid.filter.stat.log-slow-sql=true
spring.datasource.druid.filter.stat.slow-sql-millis=2000
# 配置wallFilter防火墙
spring.datasource.druid.filter.wall.enabled=true
spring.datasource.druid.filter.wall.db-type=mysql
spring.datasource.druid.filter.wall.config.delete-allow=false
spring.datasource.druid.filter.wall.config.drop-table-allow=false
# 配置监控页，内置监控页面的首页是 /druid/index.html
spring.datasource.druid.stat-view-servlet.enabled=true
spring.datasource.druid.stat-view-servlet.login-username=admin
spring.datasource.druid.stat-view-servlet.login-password=admin
spring.datasource.druid.stat-view-servlet.allow=*

# 其他 Filter 配置不再演示
# 目前为以下 Filter 提供了配置支持，请参考文档或者根据IDE提示
# (spring.datasource.druid.filter.*)
# 进行配置。
# StatFilter
# WallFilter
# ConfigFilter
# EncodingConvertFilter
# Slf4jLogFilter
# Log4jFilter
# Log4j2Filter
# CommonsLogFilter
```

示例数据库

```
CREATE TABLE `t_user`  
(  
    `id`          BIGINT(20)    NOT NULL AUTO_INCREMENT COMMENT '编号',  
    `login_name`  VARCHAR(200)  NULL DEFAULT NULL COMMENT '用户名称' COLLATE  
    'utf8_general_ci',  
    `nick_name`   VARCHAR(200)  NULL DEFAULT NULL COMMENT '用户昵称' COLLATE  
    'utf8_general_ci',  
    `passwd`      VARCHAR(200)  NULL DEFAULT NULL COMMENT '用户密码' COLLATE  
    'utf8_general_ci',  
    PRIMARY KEY (`id`)  
);  
insert into t_user(login_name, nick_name, passwd) VALUES ('zhangsan','张  
三','123456');
```

基础特性

SpringApplication

自定义 banner

- 类路径添加 `banner.txt` 或设置 `spring.banner.location` 就可以定制banner
- 推荐网站：[Spring Boot banner 在线生成工具](#), 制作下载英文 `banner.txt`, 修改替换 `banner.txt` 文字实现自定义, 个性化启动 [banner-bootschool.net](#)

自定义 SpringApplication

```
@SpringBootApplication  
public class MyApplication {  
    public static void main(String[] args) {  
        // 1. SpringApplication: Boot应用的核心API入口  
        // SpringApplication.run(Boot306FeaturesApplication.class, args);  
  
        // 1. 自定义SpringApplication的底层设置  
        SpringApplication application = new  
        SpringApplication(Boot306FeaturesApplication.class);  
  
        // 程序化调整【SpringApplication的参数】  
        // application.setDefaultProperties();  
        application.setBannerMode(Banner.Mode.OFF);  
  
        // 【配置文件优先级高于程序化调整的优先级】  
  
        // 2. SpringApplication 进行启动  
        application.run(args);  
    }  
}
```

FluentBuilder API

```
// 以Builder方式构建 SpringApplication
new SpringApplicationBuilder()
    .main(Boot306FeaturesApplication.class)
    .sources(Boot306FeaturesApplication.class)
    .bannerMode(Banner.Mode.OFF)
    // .environment(null)
    // .listeners(null)
    .run(args);
```

Profiles

环境隔离能力；快速切换开发、测试、生产环境

步骤：

1. 标识环境：指定哪些组件、配置在那个环境生效
2. 切换环境：这个环境对应的所有组件和配置就应该生效

使用

指定环境

- Spring Profiles 提供一种隔离配置的方式，使其仅在特定环境生效
- 任何 `@Component`、`@Configuration` 或 `@ConfigurationProperties` 可以使用 `@Profile` 标记，来指定何时被加载。【容器中的组件都可以被 `@Profile` 标记】

环境激活

1. 配置激活指定环境；配置文件

```
spring.profiles.active=production,hsqldb
```

2. 也可以使用命令行激活。`--spring.profile.active=dev,hasqldb`

3. 还可以配置默认环境；不标注`@Profile`的组件永远都存在。

1. 以前默认环境叫`default`
2. `spring.profiles.default-test`

4. 推荐使用激活方式激活指定环境

环境包含

注意：

1. `spring.profiles.active` 和 `spring.profiles.default` 只能用到无 profile 的文件中，如果在 `application-dev.yml` 中编写就是无效的
2. 也可以额外添加生肖文件，而不是激活替换。比如：

```
spring.profiles.include[0]=common
spring.profiles.include[1]=local
```

最佳实战：

- 生效的环境 = 激活的环境/默认环境 + 包含的环境
- 项目里面这么用

- 基础的配置 mybatis、log、xxx：写到**包含环境中**
- 需要动态切换变化的 db、redis：写到**激活的环境中**

Profile分组

创建prod组，指定包含db和mq配置

```
spring.profiles.group.prod[0]=db  
spring.profiles.group.prod[1]=mq
```

使用--spring.profiles.active=prod，就会激活prod，db，mq配置文

Profile 配置文件

- application-{profile}.properties 可以作为**指定环境的配置文件**。
 - 激活这个环境，**配置**就会生效。最终生效的所有**配置**是
 - application.properties：主配置文件，任意时候都生效
 - application-{profile}.properties：指定环境配置文件，激活**指定环境**生效
- profile优先级 > application

外部化配置

场景：线上应用如何**快速修改配置，并应用最新配置**？

- SpringBoot使用**配置优先级 + 外部配置** 简化配置更新，简化运维
- 只需要给 jar 应用所在的文件夹放一个 **application.properties** 最新配置文件，重启项目就能自动应用最新配置

配置优先级

- Spring Boot 允许将**配置外部化**，以便可以在不同的环境中使用相同的应用程序代码。
- 我们可以使用各种**外部配置源**，包括Java Properties文件、YAML文件、**环境变量**和**命令行参数**。
- @value 可以获取值，也可以用 @ConfigurationProperties 将所有属性绑定到 **java object** 中
- **以下是 SpringBoot 属性源加载顺序**。**后面的会覆盖前面的值**。由低到高，高优先级配置覆盖低优先级
 - 默认属性 (通过 **SpringApplication.setDefaultProperties** 指定的)
 - @PropertySource 指定加载的配置 (需要写在 @Configuration 类上才可生效)
 - 配置文件 (**application.properties/yml** 等)
 - RandomValuePropertySource 支持的 random.* 配置 (如：@value("\${random.int}"))
 - OS 环境变量
 - Java 系统属性 (**System.getProperties()**)
 - JNDI 属性 (来自 **java:comp/env**)
 - ServletContext 初始化参数
 - ServletConfig 初始化参数
 - SPRING_APPLICATION_JSON 属性 (内置在环境变量或系统属性中的 JSON)
 - 命令行参数
 - 测试属性。 (@SpringBootTest 进行测试时指定的属性)
 - 测试类 @TestPropertySource 注解
 - Devtools 设置的全局属性。 (\${HOME/.config/spring-boot})

结论：配置可以写到很多位置，常见的优先级顺序

- 命令行 > 配置文件 > **springapplication**配置

- 配置文件优先级如下：(后面覆盖前面)

- jar 包内的 `application.properties/yml`
- jar 包内的 `application-{profile}.properties/yml`
- jar 包外的 `application.properties/yml`
- jar 包外的 `application-{profile}.properties/yml`

- 建议：用一种格式的配置文件。如果`.properties`和`.yml`同时存在，则`.properties`优先

结论：包外 > 包内； 同级情况：profile配置 > application配置

- 所有参数均可由命令行传入，使用`--参数项=参数值`，将会被添加到环境变量中，并优先于配置文件。

- *比如`java -jar app.jar --name="Spring"`，可以使用`@Value("${name}")`获取

演示场景：

- 包内：`application.properties server.port=8000`
- 包内：`application-dev.properties server.port=9000`
- 包外：`application.properties server.port=8001`
- 包外：`application-dev.properties server.port=9001`

启动端口：命令行 > 9001 > 8001 > 9000 > 8000

外部配置

- SpringBoot 应用启动时会自动寻找 `application.properties` 和 `application.yaml` 位置，进行加载。顺序如下：(后面覆盖前面)

1. 类路径：内部

1. 类根路径

2. 类下 `/config` 包

2. 当前路径（项目所在的位置）

1. 当前路径

2. 当前下 `/config` 子目录

3. `/config` 目录的直接子目录

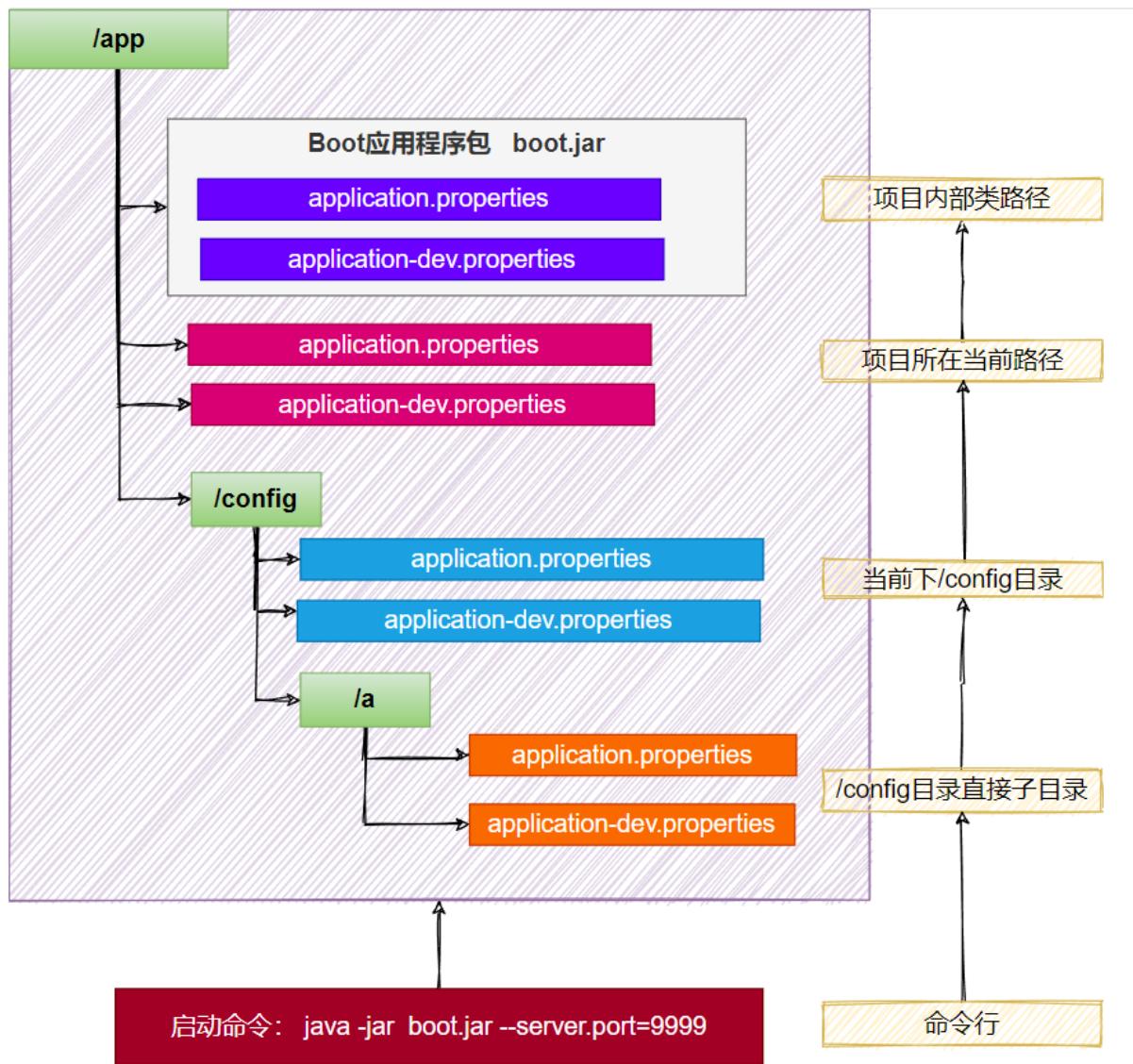
最终效果：优先级由高到低，前面覆盖后面

- 命令行 > 包外 config 直接子目录 > 包外 config 目录 > 包外 根目录 > 包内 目录

- 同级比较：

- profile 配置 > 默认配置

- properties 配置 > yaml 配置



规律：最外层的最优先。

- 命令行 > 所有
- 包外 > 包内
- config 目录 > 根目录
- profile > application

配置不同就都生效（互补），配置相同高优先级覆盖低优先级

导入配置

使用 `spring.config.import` 可以导入额外配置

```
spring.config.import=my.properties
my.property=value
```

无论以上写法的先后顺序，`my.properties` 的值总是优先于直接在文件中编写的 `my.property`。

属性占位符

配置文件中可以使用 `${name:default}` 形式取出之前配置过的值。

```
app.name=MyApp
app.description=${app.name} is a Spring Boot application written by
${username:Unknown}
```

单元测试-JUnit5

整合

- SpringBoot 提供一系列测试工具集及注解方便我们进行测试。
- `spring-boot-test` 提供核心测试能力, `spring-boot-test-autoconfigure` 提供测试的一些自动配置。
- 我们只需要导入 `spring-boot-starter-test` 即可整合测试

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

`spring-boot-starter-test` 默认提供了以下库供我们测试使用

- JUnit 5
- Spring Test
- AssertJ
- Hamcrest
- Mockito
- JSONassert
- JsonPath

测试

组件测试

直接 `@Autowired` 容器中的组件进行测试

注解

JUnit5的注解与JUnit4的注解有所变化

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations>

- `@Test` :表示方法是测试方法。但是与JUnit4的`@Test`不同，他的职责非常单一不能声明任何属性，拓展的测试将会由Jupiter提供额外测试
- `@ParameterizedTest` :表示方法是参数化测试，下方会有详细介绍
- `@RepeatedTest` :表示方法可重复执行，下方会有详细介绍
- `@DisplayName` :为测试类或者测试方法设置展示名称
- `@BeforeEach` :表示在每个单元测试之前执行
- `@AfterEach` :表示在每个单元测试之后执行
- `@BeforeAll` :表示在所有单元测试之前执行
- `@AfterAll` :表示在所有单元测试之后执行
- `@Tag` :表示单元测试类别，类似于JUnit4中的`@Categories`
- `@Disabled` :表示测试类或测试方法不执行，类似于JUnit4中的`@Ignore`
- `@Timeout` :表示测试方法运行如果超过了指定时间将会返回错误
- `@ExtendWith` :为测试类或测试方法提供扩展类引用

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
```

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @DisplayName("😱")
    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("z"));
        fail("test should have been aborted");
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }
}
```

断言

方法	说明
assertEquals	判断两个对象或两个原始类型是否相等
assertNotEquals	判断两个对象或两个原始类型是否不相等
assertSame	判断两个对象引用是否指向同一个对象
assertNotSame	判断两个对象引用是否指向不同的对象
assertTrue	判断给定的布尔值是否为 true
assertFalse	判断给定的布尔值是否为 false
assertNull	判断给定的对象引用是否为 null
assertNotNull	判断给定的对象引用是否不为 null
assertArrayEquals	数组断言
assertAll	组合断言
assertThrows	异常断言
assertTimeout	超时断言
fail	快速失败

嵌套测试

JUnit 5 可以通过 Java 中的内部类和@Nested 注解实现嵌套测试，从而可以更好的把相关的测试方法组织在一起。在内部类中可以使用@BeforeEach 和@AfterEach 注解，而且嵌套的层次没有限制。

```
@DisplayName("A stack")
class TestingAStackDemo {

    Stack<Object> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested
    @DisplayName("when new")
    class WhenNew {

        @BeforeEach
        void createNewStack() {
            stack = new Stack<>();
        }

        @Test
        @DisplayName("is empty")
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }
    }
}
```

```

    @Test
    @DisplayName("throws EmptyStackException when popped")
    void throwsExceptionWhenPopped() {
        assertThrows(EmptyStackException.class, stack::pop);
    }

    @Test
    @DisplayName("throws EmptyStackException when peeked")
    void throwsExceptionWhenPeeked() {
        assertThrows(EmptyStackException.class, stack::peek);
    }

    @Nested
    @DisplayName("after pushing an element")
    class AfterPushing {

        String anElement = "an element";

        @BeforeEach
        void pushAnElement() {
            stack.push(anElement);
        }

        @Test
        @DisplayName("it is no longer empty")
        void isNotEmpty() {
            assertFalse(stack.isEmpty());
        }

        @Test
        @DisplayName("returns the element when popped and is empty")
        void returnElementWhenPopped() {
            assertEquals(anElement, stack.pop());
            assertTrue(stack.isEmpty());
        }

        @Test
        @DisplayName("returns the element when peeked but remains not
empty")
        void returnElementWhenPeeked() {
            assertEquals(anElement, stack.peek());
            assertFalse(stack.isEmpty());
        }
    }
}

```

参数化测试

- 参数化测试是JUnit5很重要的一个新特性，它使得用不同的参数多次运行测试成为了可能，也为我们的单元测试带来许多便利。
- 利用**@ValueSource**等注解，指定入参，我们将可以使用不同的参数进行多次单元测试，而不需要每新增一个参数就新增一个单元测试，省去了很多冗余代码。

- **@ValueSource**: 为参数化测试指定入参来源，支持八大基础类以及String类型,Class类型
- **@NullSource**: 表示为参数化测试提供一个null的入参
- **@EnumSource**: 表示为参数化测试提供一个枚举入参
- **@CsvFileSource**: 表示读取指定CSV文件内容作为参数化测试入参

- **@MethodSource**: 表示读取指定方法的返回值作为参数化测试入参(注意方法返回需要是一个流)

```

@ParameterizedTest
@valueSource(strings = {"one", "two", "three"})
@DisplayName("参数化测试1")
public void parameterizedTest1(String string) {
    System.out.println(string);
    Assertions.assertTrue(StringUtils.isNotBlank(string));
}

@ParameterizedTest
@MethodSource("method")      //指定方法名
@DisplayName("方法来源参数")
public void testWithExplicitLocalMethodSource(String name) {
    System.out.println(name);
    Assertions.assertNotNull(name);
}

static Stream<String> method() {
    return Stream.of("apple", "banana");
}

```

核心原理

事件和监听器

生命周期监听

监听应用的生命周期

监听器-SpringApplicationRunListener

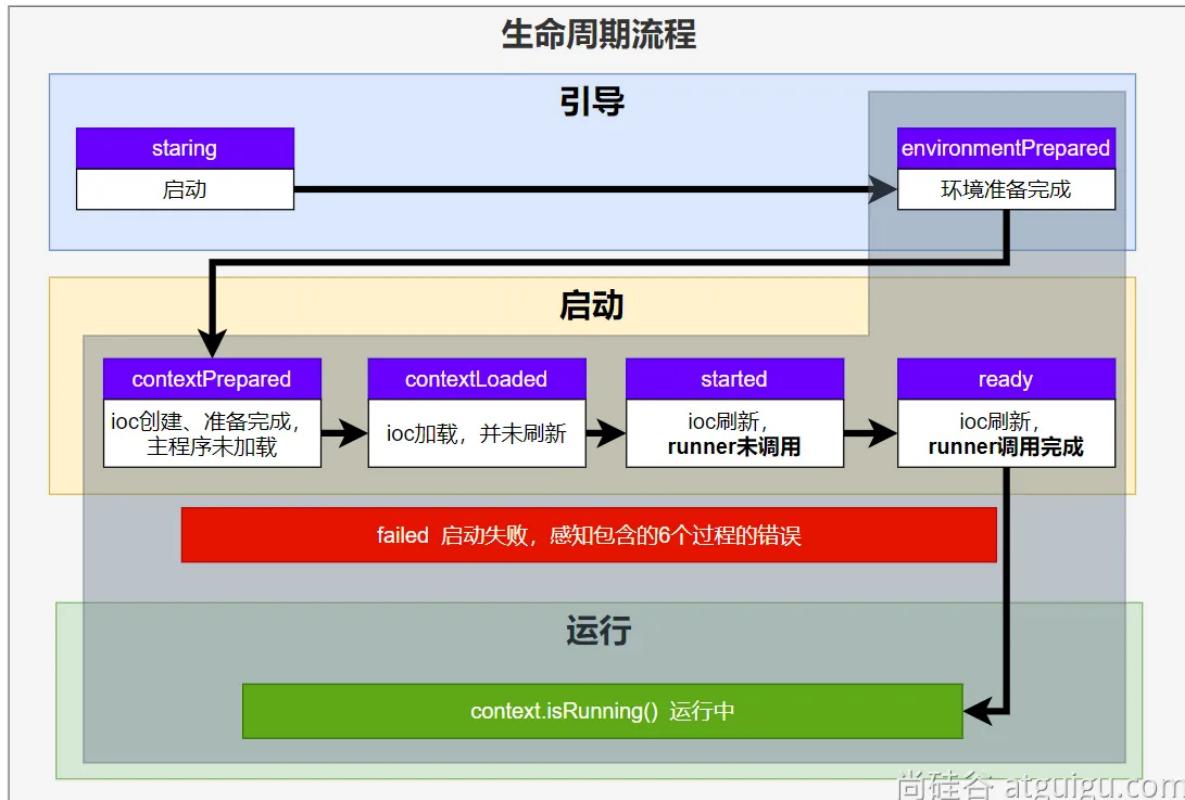
1. 自定义 `SpringApplicationRunListener` 来监听事件;
 1. 编写 `SpringApplicationRunListener` 实现类
 2. 在 `META-INF/spring.factories` 中配置
`org.springframework.boot.SpringApplicationRunListener=自己的Listener`, 还可以指定一个有参构造器, 接受两个参数 (`SpringApplication application, String[] args`)
 3. springboot 在 `spring-boot.jar` 中配置了默认的Listener, 如下
- ```

org.springframework.boot.SpringApplicationRunListener=\norg.springframework.boot.context.event.EventPublishingRunListener

```
- 监听器运行步骤
    - Listener先要从 `META-INF/spring.factories` 读到
    - 引导: 利用 `BootstrapContext` 引导整个项目启动
      - **starting**: 应用开始, `SpringApplication` 的run方法一调用, 只要有有了 `BootstrapContext` 就执行
      - **environmentPrepared**: 环境准备好 (把启动参数等绑定到环境变量中), 但是ioc还没有创建; 【调一次】
    - 启动:
      - **contextPrepared**: IOC容器创建并准备好, 但是sources (主配置类) 没加载。并关闭引导上下文; 组件都没创建 【调一次】

- **contextLoaded**: IOC容器加载。主配置类加载进去了。但是ioc容器还没刷新（我们的bean没创建）。
- =====截止以前，IOC容器里面还没造bean呢=====
- **started**: ioc容器刷新了（所有bean造好了），但是 runner 没调用。
- **ready**: ioc容器刷新了（所有bean造好了），所有 runner 调用完了。
- 运行
  - 以前步骤都正确执行，代表容器running。

## 生命周期全流程



## 事件触发时机

### 各种回调监听器

- **BootstrapRegistryInitializer**: 感知引导初始化
  - **META-INF/spring.factories**
  - 创建引导上下文 `bootstrapContext` 的时候触发
  - `application.addBootstrapRegistryInitializer();`
  - 场景：进行密钥校对授权
- **ApplicationContextInitializer**: 感知IOC容器初始化
  - **META-INF/spring.factories**
  - `application.addInitializers();`
- **ApplicationListener**: 感知全阶段：基于事件机制，感知事件。一旦到了那个阶段可以做别的事
  - `@Bean` 或 `@EventListener`：事件驱动
  - `SpringApplication.addListener(...)` 或 `SpringApplicationBuilder.listeners(...)`
  - **META-INF/spring.factories**

- SpringApplicationRunListener: 感知全阶段生命周期 + 各种阶段都能自定义操作; 功能更完善
  - META-INF/spring.factories
- ApplicationRunner: 感知特定节点: 感知应用就绪Ready。卡死应用, 就不会就绪
  - @Bean
- CommandLineRunner: 感知特定阶段: 感知应用就绪Ready。卡死应用, 就不会就绪
  - @Bean

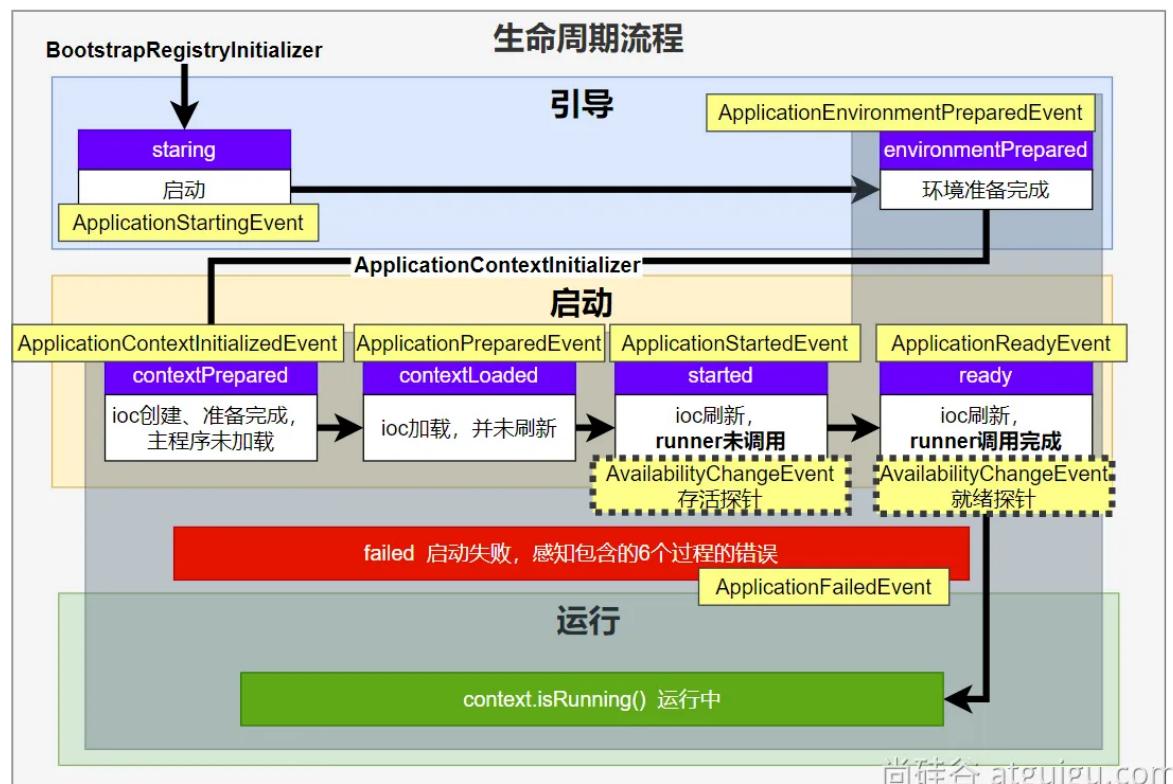
最佳实践:

- 如果项目启动前做事: `BootstrapRegistryInitializer`、`ApplicationContextInitializer`
- 如果想要在项目启动完成后做事: `ApplicationRunner`
- 如果要干涉生命周期做事: `SpringApplicationRunListener`
- 如果想要用事件机制: `ApplicationListener`

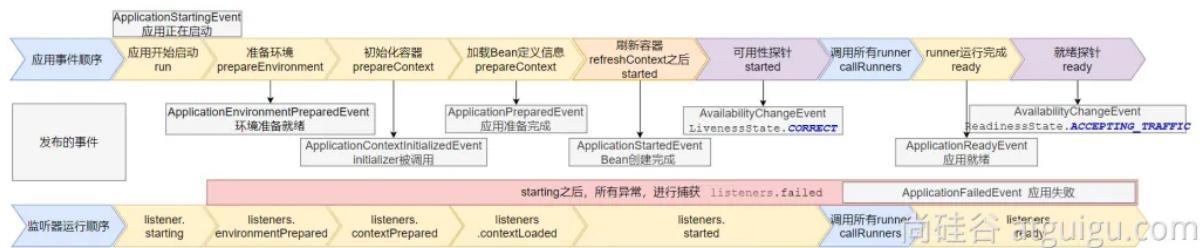
## 完整触发流程

9大事件 触发顺序&时机

1. `ApplicationStartingEvent`: 应用启动但未做任何事情, 除过注册listeners and initializers.
2. `ApplicationEnvironmentPreparedEvent`: Environment 准备好, 但context 未创建.
3. `ApplicationContextInitializedEvent`: ApplicationContext 准备好, ApplicationContextInitializers 调用, 但是任何bean未加载
4. `ApplicationPreparedEvent`: 容器刷新之前, bean定义信息加载
5. `ApplicationStartedEvent`: 容器刷新完成, runner未调用
- =====以下就开始插入了探针机制=====
6. `AvailabilityChangeEvent`: `LivenessState.CORRECT` 应用存活; **存活探针**
7. `ApplicationReadyEvent`: 任何runner被调用
8. `AvailabilityChangeEvent`: `ReadinessState.ACCEPTING_TRAFFIC` 就绪探针, 可以接请求
9. `ApplicationFailedEvent`: 启动出错



应用事件发送顺序如下:

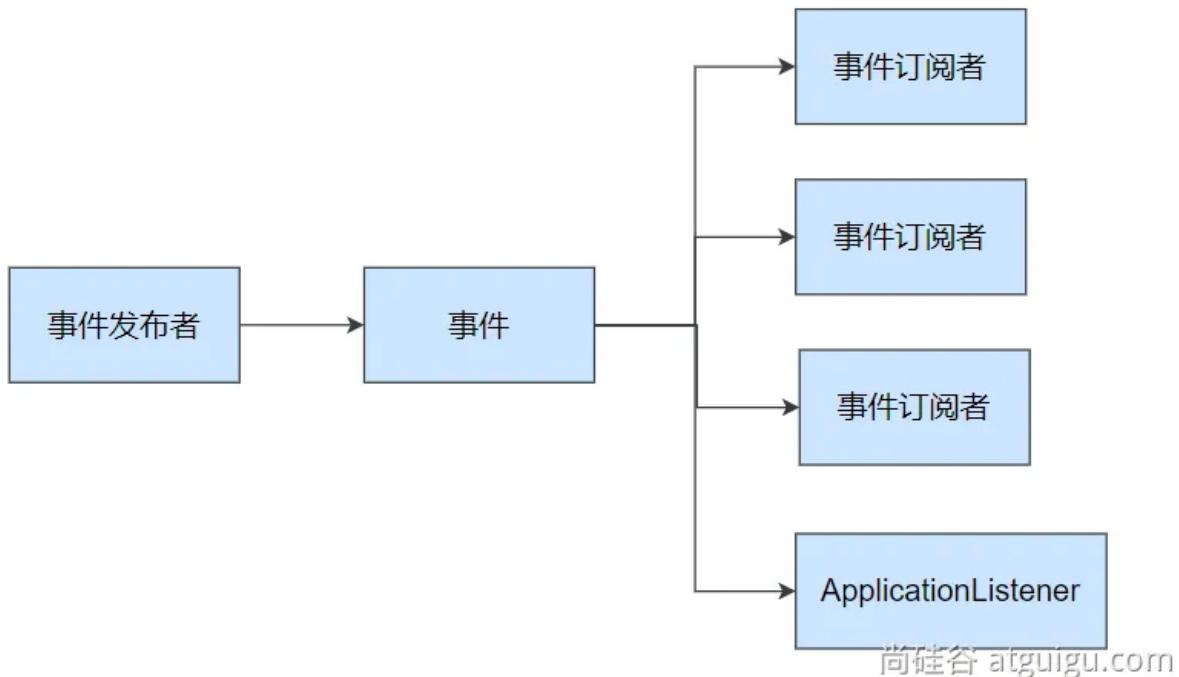
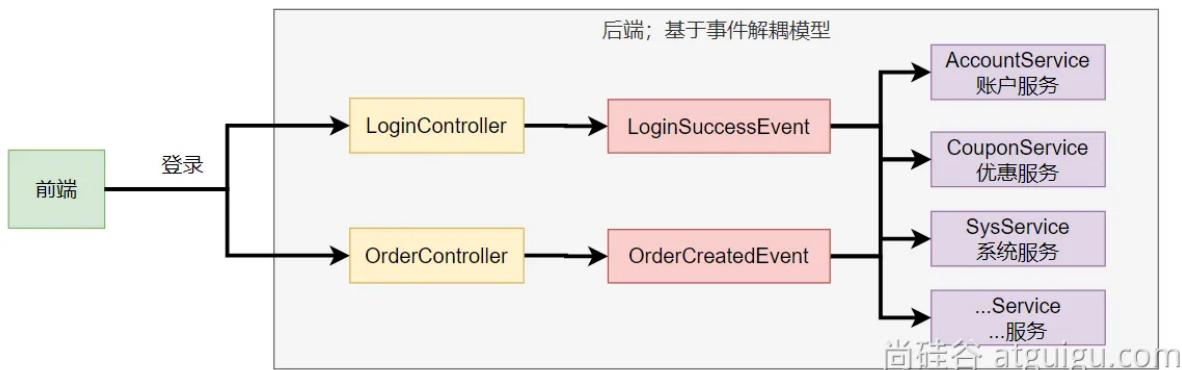


- 感知应用是否存活了：可能植物状态，虽然活着但是不能处理请求。
- 应用是否就绪了：能响应请求，说明确实活的比较好。

## SpringBoot 事件驱动开发

应用启动过程生命周期事件感知（9大事件）、应用运行中事件感知（无数种）

- 事件发布：ApplicationEventPublisherAware 或 注入：ApplicationEventMulticaster
- 事件监听：组件 + @EventListener



事件发布者

```
@Service
public class EventPublisher implements ApplicationEventPublisher {
 /**
 * ...
 */
}
```

```
* 底层发送事件用的组件，SpringBoot会通过ApplicationEventPublisherAware接口自动注入给我们
 */
ApplicationEventPublisher applicationEventPublisher;

/**
 * 所有事件都可以发
 * @param event
 */
public void sendEvent(ApplicationEvent event) {
 //调用底层API发送事件
 applicationEventPublisher.publishEvent(event);
}

/**
 * 会被自动调用，把真正发事件的底层组件给我们注入进来
 * @param applicationEventPublisher event publisher to be used by this
object
 */
@Override
public void setApplicationEventPublisher(ApplicationEventPublisher
applicationEventPublisher) {
 this.applicationEventPublisher = applicationEventPublisher;
}
}
```

### 事件订阅者

```
@Service
public class CouponService {

 @Order(1)
 @EventListener
 public void onEvent(LoginSuccessEvent loginSuccessEvent){
 System.out.println("===== CouponService =====感知到事
件"+loginSuccessEvent);
 UserEntity source = (UserEntity) loginSuccessEvent.getSource();
 sendCoupon(source.getUsername());
 }

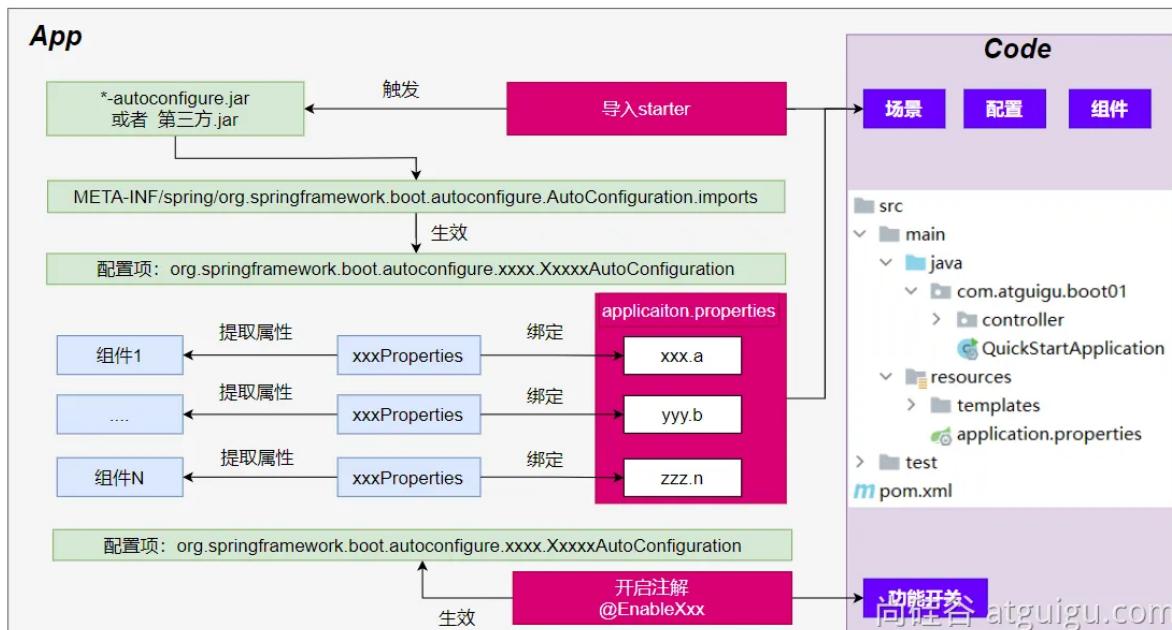
 public void sendCoupon(String username){
 System.out.println(username + " 随机得到了一张优惠券");
 }
}
```

## 自动配置原理

### 入门理解

应用关注的三大核心：场景、配置、组件

## 自动配置流程



1. 导入 `starter`
2. 依赖导入 `autoconfigure`
3. 寻找类路径下 `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 文件
4. 启动，加载所有 `自动配置类 xxxAutoConfiguration`
  1. 给容器中配置功能 `组件`
  2. `组件参数` 绑定到 `属性类` 中
  3. `属性类` 和 `配置文件` 前缀项绑定
  4. `@Conditional` 派生的条件注解 进行判断是否组件生效
5. 效果：
  1. 修改配置文件，修改底层参数
  2. 所有场景自动配置好直接使用
  3. 可以注入SpringBoot配置好的组件随时使用

## SPI机制

- Java中的SPI (Service Provider Interface) 是一种软件设计模式，用于在应用程序中动态地发现和加载组件。SPI的思想是，定义一个接口或抽象类，然后通过在classpath中定义实现该接口的类来实现对组件的动态发现和加载。
- SPI的主要目的是解决在应用程序中使用可插拔组件的问题。例如，一个应用程序可能需要使用不同的日志框架或数据库连接池，但是这些组件的选择可能取决于运行时的条件。通过使用SPI，应用程序可以在运行时发现并加载适当的组件，而无需在代码中硬编码这些组件的实现类。
- 在Java中，SPI的实现方式是通过在 `META-INF/services` 目录下创建一个以服务接口全限定名为名字的文件，文件中包含实现该服务接口的类的全限定名。当应用程序启动时，Java的SPI机制会自动扫描classpath中的这些文件，并根据文件中指定的类名来加载实现类。
- 通过使用SPI，应用程序可以实现更灵活、可扩展的架构，同时也可以避免硬编码依赖关系和增加代码的可维护性。

在SpringBoot中，`META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`

## 功能开关

- 自动配置：全部都配置好，什么都不用管。 自动批量导入
- ○ 项目一启动，spi文件中指定的所有都加载。
- `@Enablexxxx`：手动控制哪些功能的开启； 手动导入。
- ○ 开启xxx功能
  - 都是利用 `@Import` 把此功能要用的组件导入进去

## 进阶理解

### `@SpringBootApplication`

#### `@SpringBootConfiguration`

就是： `@Configuration`， 容器中的组件， 配置类。spring ioc启动就会加载创建这个类对象

#### `@EnableAutoConfiguration`: 开启自动配置

开启自动配置

#### `@AutoConfigurationPackage`: 扫描主程序包：加载自己的组件

- 利用 `@Import(AutoConfigurationPackages.Registrar.class)` 想要给容器中导入组件。
- 把主程序所在的包的所有组件导入进来。
- **为什么SpringBoot默认只扫描主程序所在的包及其子包**

#### `@Import(AutoConfigurationImportSelector.class)`: 加载所有自动配置类：加载starter导入的组件

```
List<String> configurations = ImportCandidates.load(AutoConfiguration.class,
getBeanClassLoader()).getCandidates();
```

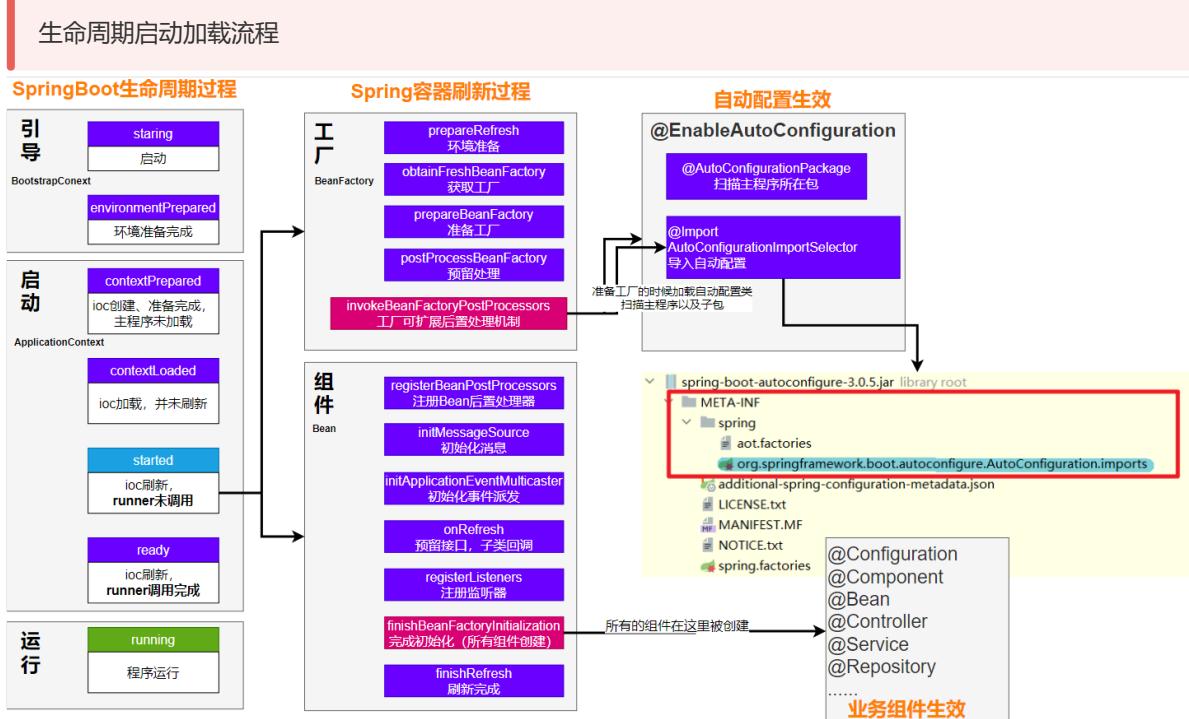
扫描SPI文件：`META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`

### `@ComponentScan`

组件扫描：排除一些组件（哪些不要）  
排除前面已经扫描进来的 **配置类**、和 **自动配置类**

```
@ComponentScan(excludeFilters = {
 @Filter(
 type = FilterType.CUSTOM,
 classes = TypeExcludeFilter.class
),
 @Filter(
 type = FilterType.CUSTOM,
 classes = AutoConfigurationExcludeFilter.class
) })
```

# 完整启动加载流程



## 自定义starter

场景：抽取聊天机器人场景，它可以打招呼。

效果：任何项目导入此 **starter** 都具有打招呼功能，并且问候语中的人名需要可以在配置文件中修改

- 创建 **自定义starter** 项目，引入 **spring-boot-starter** 基础依赖
- 编写模块功能，引入模块所有需要的依赖。
- 编写 **xxxAutoConfiguration** 自动配置类，帮其他项目导入这个模块需要的所有组件
- 编写配置文件 **META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports** 指定启动需要加载的自动配置
- 其他项目引入即可使用

## 业务代码

自定义配置有提示。导入以下依赖重启项目，再写配置文件就有提示

```
@ConfigurationProperties(prefix = "robot") //此属性类和配置文件指定前缀绑定
@Component
@Data
public class RobotProperties {

 private String name;
 private String age;
 private String email;
}
```

```
<!-- 导入配置处理器，配置文件自定义的properties配置都会有提示-->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-configuration-processor</artifactId>
 <optional>true</optional>
</dependency>
```

## 基本抽取

- 创建starter项目，把公共代码需要的所有依赖导入
- 把公共代码复制进来
- 自己写一个 `RobotAutoConfiguration`，给容器中导入这个场景需要的所有组件
  - 为什么这些组件默认不会扫描进去？
    - `starter`所在的包和 引入它的项目的主程序所在的包不是父子层级
- 别人引用这个 `starter`，直接导入这个 `RobotAutoConfiguration` 就能把这个场景的组件导入进来
- 功能生效。
- 测试编写配置文件

## 使用`@EnableXxx`机制

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Documented
@Import(RobotAutoConfiguration.class)
public @interface EnableRobot {}
```

别人引入 `starter` 需要使用 `@EnableRobot` 开启功能

## 完全自动配置

- 依赖SpringBoot的SPI机制
- `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 文件中编写好我们自动配置类的全类名即可
- 项目启动，自动加载我们的自动配置类

# SpringBoot3场景整合

## 环境准备

- 安装以下组件
  - docker
  - redis
  - kafka
  - prometheus
  - grafana

# Docker安装

```
sudo yum install -y yum-utils

sudo yum-config-manager \
 --add-repo \
 https://download.docker.com/linux/centos/docker-ce.repo

sudo yum install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin

sudo systemctl enable docker --now

#测试工作
docker ps
批量安装所有软件
docker compose
```

创建 `/prod` 文件夹，准备以下文件

## prometheus.yml

```
global:
 scrape_interval: 15s
 evaluation_interval: 15s

scrape_configs:
 - job_name: 'prometheus'
 static_configs:
 - targets: ['localhost:9090']

 - job_name: 'redis'
 static_configs:
 - targets: ['redis:6379']

 - job_name: 'kafka'
 static_configs:
 - targets: ['kafka:9092']
```

## docker-compose.yml

```
version: '3.9'

services:
 redis:
 image: redis:latest
 container_name: redis
 restart: always
 ports:
 - "6379:6379"
 networks:
 - backend

 zookeeper:
 image: bitnami/zookeeper:latest
 container_name: zookeeper
 restart: always
```

```
environment:
 ZOOKEEPER_CLIENT_PORT: 2181
 ZOOKEEPER_TICK_TIME: 2000
networks:
 - backend

kafka:
 image: bitnami/kafka:3.4.0
 container_name: kafka
 restart: always
 depends_on:
 - zookeeper
 ports:
 - "9092:9092"
environment:
 ALLOW_PLAINTEXT_LISTENER: yes
 KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
 KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
networks:
 - backend

kafka-ui:
 image: provectuslabs/kafka-ui:latest
 container_name: kafka-ui
 restart: always
 depends_on:
 - kafka
 ports:
 - "8080:8080"
environment:
 KAFKA_CLUSTERS_0_NAME: dev
 KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:9092
networks:
 - backend

prometheus:
 image: prom/prometheus:latest
 container_name: prometheus
 restart: always
 volumes:
 - ./prometheus.yml:/etc/prometheus/prometheus.yml
 ports:
 - "9090:9090"
 networks:
 - backend

grafana:
 image: grafana/grafana:latest
 container_name: grafana
 restart: always
 depends_on:
 - prometheus
 ports:
 - "3000:3000"
 networks:
 - backend

networks:
 backend:
 name: backend
```

# 启动环境

```
docker compose -f docker-compose.yml up -d
```

## 安装完后进行验证

- Redis: ip:6379; 使用可视化工具测试
- 使用浏览器直接访问
  - kafka: ip:9002
  - permetheus: ip:9090
  - grafana: ip:3000

# NoSQL

## Redis整合

### 场景整合

依赖导入

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

配置

```
spring.data.redis.host=ip
spring.data.redis.password=密码
```

测试

```
@Autowired
StringRedisTemplate redisTemplate;

@Test
void redisTest(){
 redisTemplate.opsForValue().set("a","1234");
 Assertions.assertEquals("1234",redisTemplate.opsForValue().get("a"));
}
```

## 自动配置原理

- META-**  
INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports 中导入了 RedisAutoConfiguration、RedisReactiveAutoConfiguration 和 RedisRepositoriesAutoConfiguration。所有属性绑定在 RedisProperties 中
- RedisReactiveAutoConfiguration** 属于响应式编程，不用管。  
**RedisRepositoriesAutoConfiguration** 属于 JPA 操作，也不用管
- RedisAutoConfiguration** 配置了以下组件
  - LettuceConnectionConfiguration**: 给容器中注入了连接工厂 LettuceConnectionFactory，和操作 redis 的客户端 DefaultClientResources
  - RedisTemplate<Object, Object>**: 可给 redis 中存储任意对象，会使用 jdk 默认序列化方式。

3. **StringRedisTemplate**: 给 redis 中存储字符串，如果要存对象，需要开发人员自己进行序列化。  
key-value都是字符串进行操作…

## 定制化

### 序列化机制

```
@Configuration
public class AppRedisConfiguration {

 /**
 * 允许Object类型的key-value，都可以被转为json进行存储。
 * @param redisConnectionFactory 自动配置好了连接工厂
 * @return
 */
 @Bean
 public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory) {
 RedisTemplate<Object, Object> template = new RedisTemplate<>();
 template.setConnectionFactory(redisConnectionFactory);
 //把对象转为json字符串的序列化工具
 template.setDefaultSerializer(new
GenericJackson2JsonRedisSerializer());
 return template;
 }
}
```

## redis客户端

RedisTemplate、StringRedisTemplate: 操作redis的工具类

- 要从redis的连接工厂获取链接才能操作redis
- **Redis客户端**
- ◦ Lettuce: 默认
- ◦ Jedis: 可以使用以下切换

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-redis</artifactId>
 <exclusions>
 <exclusion>
 <groupId>io.lettuce</groupId>
 <artifactId>lettuce-core</artifactId>
 </exclusion>
 </exclusions>
</dependency>

<!--切换 jedis 作为操作redis的底层客户端-->
<dependency>
 <groupId>redis.clients</groupId>
 <artifactId>jedis</artifactId>
</dependency>
```

## 配置参考

```
spring.data.redis.host=8.130.74.183
spring.data.redis.port=6379
#spring.data.redis.client-type=lettuce

#设置lettuce的底层参数
#spring.data.redis.lettuce.pool.enabled=true
#spring.data.redis.lettuce.pool.max-active=8

#配置jedis底层参数
spring.data.redis.client-type=jedis
spring.data.redis.jedis.pool.enabled=true
spring.data.redis.jedis.pool.max-active=8
```

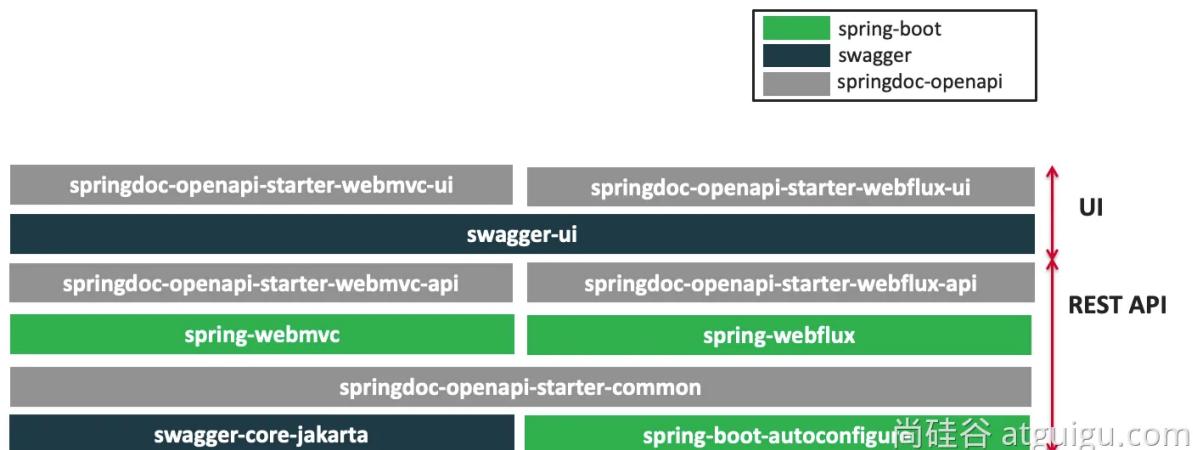
## 接口文档

### OpenAPI 3 和 Swagger

Swagger 可以快速生成**实时接口文档**，方便前后开发人员进行协调沟通。遵循 **OpenAPI** 规范。

文档：<https://springdoc.org/v2/>

### OpenAPI 3 架构



## 整合

导入场景

```
<dependency>
 <groupId>org.springdoc</groupId>
 <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
 <version>2.1.0</version>
</dependency>
```

配置

```
/api-docs endpoint custom path 默认 /v3/api-docs
springdoc.api-docs.path=/api-docs

swagger 相关配置在 springdoc.swagger-ui
swagger-ui custom path
springdoc.swagger-ui.path=/swagger-ui.html

springdoc.show-actuator=true
```

## 使用

### 常用注解

注解	标注位置	作用
@Tag	controller 类	标识 controller 作用
@Parameter	参数	标识参数作用
@Parameters	参数	参数多重说明
@Schema	model 层的 JavaBean	描述模型作用及每个属性
@Operation	方法	描述方法作用
@ApiResponse	方法	描述响应状态码等

### Docket配置

如果有多个Docket, 配置如下

```
@Bean
public GroupedOpenApi publicApi() {
 return GroupedOpenApi.builder()
 .group("springshop-public")
 .pathsToMatch("/public/**")
 .build();
}
@Bean
public GroupedOpenApi adminApi() {
 return GroupedOpenApi.builder()
 .group("springshop-admin")
 .pathsToMatch("/admin/**")
 .addMethodFilter(method -> method.isAnnotationPresent(Admin.class))
 .build();
}
```

如果只有一个Docket, 可以配置如下

```
springdoc.packagesToScan=package1, package2
springdoc.pathsToMatch=/v1, /api/balance/**
```

## OpenAPI配置

```
@Bean
public OpenAPI springShopOpenAPI() {
 return new OpenAPI()
 .info(new Info().title("SpringShop API")
 .description("Spring shop sample application")
 .version("v0.0.1")
 .license(new License().name("Apache
2.0")).url("http://springdoc.org")))
 .externalDocs(new ExternalDocumentation()
 .description("SpringShop Wiki Documentation")
 .url("https://springshop.wiki.github.org/docs"));
}
```

## Springfox迁移

### 注解变化

原注解	现注解	作用
@Api	@Tag	描述 Controller
@ApiIgnore	@Parameter(hidden = true) @Operation(hidden = true) @Hidden	描述忽略操作
@ApiImplicitParam	@Parameter	描述参数
@ApiImplicitParams	@Parameters	描述参数
@ApiModel	@Schema	描述对象
@ApiModelProperty(hidden = true)	@Schema(accessMode = READ_ONLY)	描述对象属性
@ApiModelProperty	@Schema	描述对象属性
@ApiOperation(value = "foo", notes = "bar")	@Operation(summary = "foo", description = "bar")	描述方法
@ApiParam	@Parameter	描述参数
@ApiResponse(code = 404, message = "foo")	@ApiResponse(responseCode = "404", description = "foo")	描述响应

## Docket配置

- 以前写法

```
@Bean
public Docket publicApi() {
 return new Docket(DocumentationType.SWAGGER_2)
 .select()

 .apis(RequestHandlerSelectors.basePackage("org.github.springshop.web.public"))
 .paths(PathSelectors.regex("/public.*"))
 .build()
```

```

 .groupName("springshop-public")
 .apiInfo(apiInfo());
 }

 @Bean
 public Docket adminApi() {
 return new Docket(DocumentationType.SWAGGER_2)
 .select()

 .apis(RequestHandlerSelectors.basePackage("org.github.springshop.web.admin"))
 .paths(PathSelectors.regex("/admin.*"))
 .apis(RequestHandlerSelectors.withMethodAnnotation(Admin.class))
 .build()
 .groupName("springshop-admin")
 .apiInfo(apiInfo());
 }
}

```

- 新写法

```

@Bean
public GroupedOpenApi publicApi() {
 return GroupedOpenApi.builder()
 .group("springshop-public")
 .pathsToMatch("/public/**")
 .build();
}

@Bean
public GroupedOpenApi adminApi() {
 return GroupedOpenApi.builder()
 .group("springshop-admin")
 .pathsToMatch("/admin/**")
 .addOpenApiMethodFilter(method ->
method.isAnnotationPresent(Admin.class))
 .build();
}

```

- 添加OpenAPI组件

```

@Bean
public OpenAPI springShopOpenAPI() {
 return new OpenAPI()
 .info(new Info().title("SpringShop API")
 .description("Spring shop sample application")
 .version("v0.0.1")
 .license(new License().name("Apache
2.0").url("http://springdoc.org")))
 .externalDocs(new ExternalDocumentation()
 .description("SpringShop wiki Documentation")
 .url("https://springshop.wiki.github.org/docs"));
}

```

# 远程调用

## WebClient

### 创建与配置

发请求:

- 请求方式: GET\POST\DELETE\xxxx
- 请求路径: /xxx
- 请求参数: aa=bb&cc=dd&xxx
- 请求头: aa=bb,cc=ddd
- 请求体:

创建 `webClient` 非常简单:

- `webClient.create()`
- `webClient.create(String baseUrl)`

还可以使用 `WebClient.builder()` 配置更多参数项:

- `uriBuilderFactory`: 自定义 `UriBuilderFactory` , 定义 `baseUrl`.
- `defaultUriVariables`: 默认 uri 变量.
- `defaultHeader`: 每个请求默认头.
- `defaultCookie`: 每个请求默认 cookie.
- `defaultRequest`: Consumer 自定义每个请求.
- `filter`: 过滤 client 发送的每个请求
- `exchangeStrategies`: HTTP 消息 reader/writer 自定义.
- `clientConnector`: HTTP client 库设置.

```
//获取响应完整信息
webClient client = webClient.create("https://example.org");
```

### 获取响应

`retrieve()` 方法用来声明如何提取响应数据。比如

```
//获取响应完整信息
webClient client = webClient.create("https://example.org");

Mono<ResponseEntity<Person>> result = client.get()
 .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
 .retrieve()
 .toEntity(Person.class);

//只获取body
webClient client = webClient.create("https://example.org");

Mono<Person> result = client.get()
 .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
 .retrieve()
 .bodyToMono(Person.class);

//stream数据
```

```

Flux<Quote> result = client.get()
 .uri("/person").accept(MediaType.TEXT_EVENT_STREAM)
 .retrieve()
 .bodyToFlux(Person.class);

//定义错误处理
Mono<Person> result = client.get()
 .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
 .retrieve()
 .onStatus(HttpStatus::is4xxClientError, response -> ...)
 .onStatus(HttpStatus::is5xxServerError, response -> ...)
 .bodyToMono(Person.class);

```

```

private static Mono<String> getStringMono(String city) {
 String key = "3ff0xxxxxxxxxxxxx465f6f0c105";
 // 远程调用和风天气的API
 // 1. 创建webClient
 // https://geoapi.qweather.com/v2/city/lookup?location=北京
 &key=3ff0xxxxxxxxxxxxx465f6f0c105
 webclient webClient =
 webClient.create("https://geoapi.qweather.com/v2/city/lookup");

 // 2. 定义发请求的行为
 Mono<String> mono = webClient.get()
 .uri(uriBuilder -> uriBuilder
 .queryParam("location", city)
 .queryParam("key", key)
 .build())
 .accept(MediaType.APPLICATION_JSON) // 返回JSON数据
 .retrieve()
 .bodyToMono(String.class);
 return mono;
}

```

## 定义请求体

```

//1、响应式-单个数据
Mono<Person> personMono = ... ;

Mono<Void> result = client.post()
 .uri("/persons/{id}", id)
 .contentType(MediaType.APPLICATION_JSON)
 .body(personMono, Person.class)
 .retrieve()
 .bodyToMono(Void.class);

//2、响应式-多个数据
Flux<Person> personFlux = ... ;

Mono<Void> result = client.post()
 .uri("/persons/{id}", id)
 .contentType(MediaType.APPLICATION_STREAM_JSON)
 .body(personFlux, Person.class)
 .retrieve()
 .bodyToMono(Void.class);

//3、普通对象
Person person = ... ;

```

```
Mono<Void> result = client.post()
 .uri("/persons/{id}", id)
 .contentType(MediaType.APPLICATION_JSON)
 .bodyValue(person)
 .retrieve()
 .bodyToMono(Void.class);
```

## HTTP Interface

Spring 允许我们通过定义接口的方式，给任意位置发送 http 请求，实现远程调用，可以用来简化 HTTP 远程访问。需要 **webFlux** 场景才可

### 导入依赖

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

### 定义接口

```
public interface WeatherInterface {

 /**
 * 获取天气信息
 *
 * @param city 城市名称
 * @param key 授权密钥
 * @return 包含天气信息的Mono对象
 */
 @GetExchange(url = "https://geoapi.qweather.com/v2/city/lookup", accept =
 "application/json")
 Mono<String> getWeather(@RequestParam("location") String city,
 @RequestParam("key") String key);
}
```

### 创建代理&测试

```
private static final String key = "3ff0167904294264bc451465f6f0c105";
public Mono<String> getWeather(String city) {
 // 1. 创建客户端
 webclient client = webClient.builder()
 .codecs(clientCodecConfigurer -> {
 clientCodecConfigurer
 .defaultCodecs()
 .maxInMemorySize(256 * 1024 * 1024);
 }).build();
 // 2、创建工厂
 HttpServiceProxyFactory factory =
 HttpServiceProxyFactory.builderFor(webClientAdapter.forClient(client)).build();

 // 3、获取代理对象
 weatherInterface weatherInterface =
 factory.createClient(weatherInterface.class);
```

```
 Mono<String> weather = weatherInterface.getWeather(city, key);
 return weather;
 }
```

## 抽取通用配置类

通用配置

WeatherInterface 接口

```
public interface WeatherInterface {
 @GetExchange(url = "https://geoapi.qweather.com/v2/city/lookup", accept =
 "application/json")
 Mono<String> getWeather(@RequestParam("location") String city,
 @RequestParam("key") String key);
}
```

WeatherConfiguration

```
@Configuration
public class WeatherConfiguration {
 @Bean
 HttpServiceProxyFactory httpServiceProxyFactory() {
 // 1. 创建客户端
 webClient client = webClient.builder()
 .codecs(clientCodecConfigurer -> {
 clientCodecConfigurer
 .defaultCodecs()
 .maxInMemorySize(256 * 1024 * 1024);
 }).build();
 // 2. 创建工厂
 HttpServiceProxyFactory factory =
 HttpServiceProxyFactory.builderFor(webClientAdapter.forClient(client)).build();
 return factory;
 }

 @Bean
 public WeatherInterface weatherInterface(HttpServiceProxyFactory
 httpServiceProxyFactory) {
 // 3. 获取代理对象
 WeatherInterface watherService =
 httpServiceProxyFactory.createClient(WeatherInterface.class);

 return watherService;
 }
}
```

WeatherService

```
public Mono<String> getWeather(String city) {
 Mono<String> weather = weatherInterface.getWeather(city, key);
 return weather;
}
```

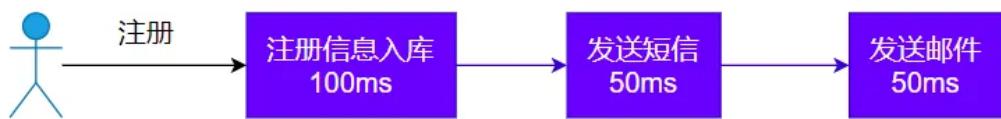
# 消息服务

<https://kafka.apache.org/documentation/>

## 消息队列-场景

### 异步

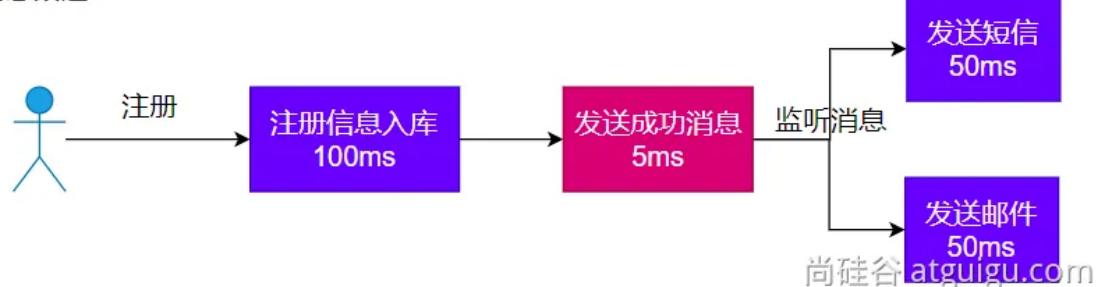
#### 场景1



#### 异步改造：



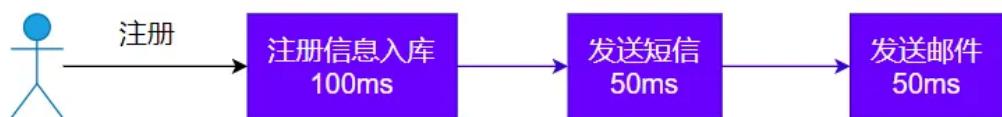
#### 消息改造：



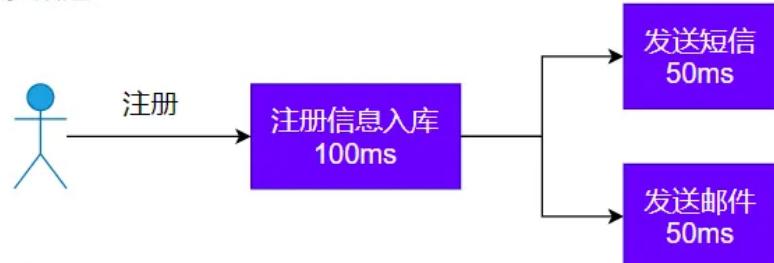
尚硅谷 [atguigu.com](http://atguigu.com)

## 解耦

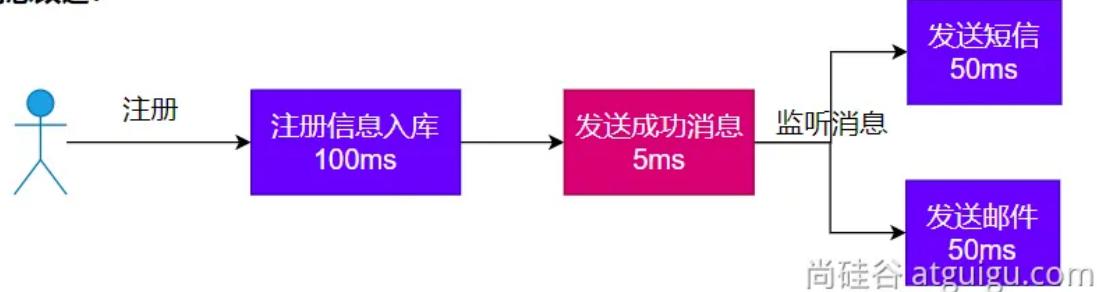
### 场景1



异步改造：



消息改造：



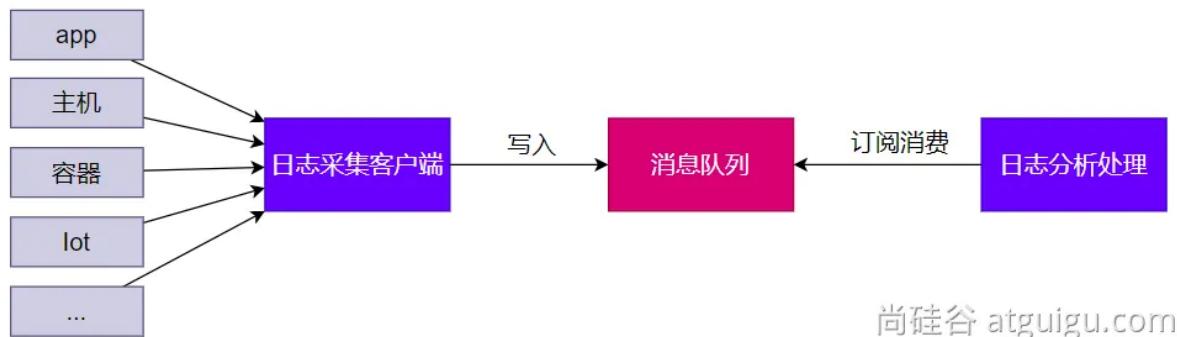
## 削峰

### 场景3



## 缓冲

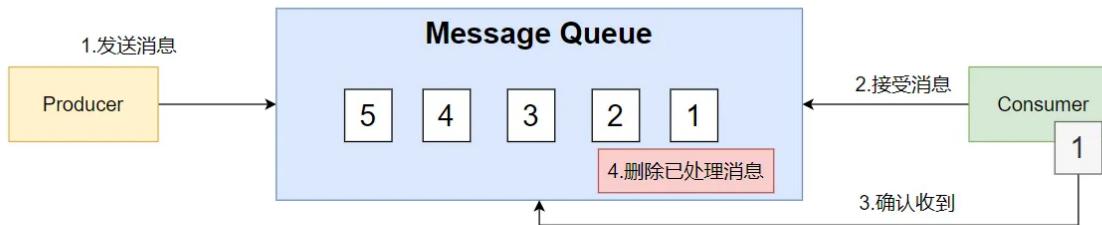
### 场景4



# 消息队列-Kafka

## 消息模式

### 消息点对点模式



### 消息发布订阅模式

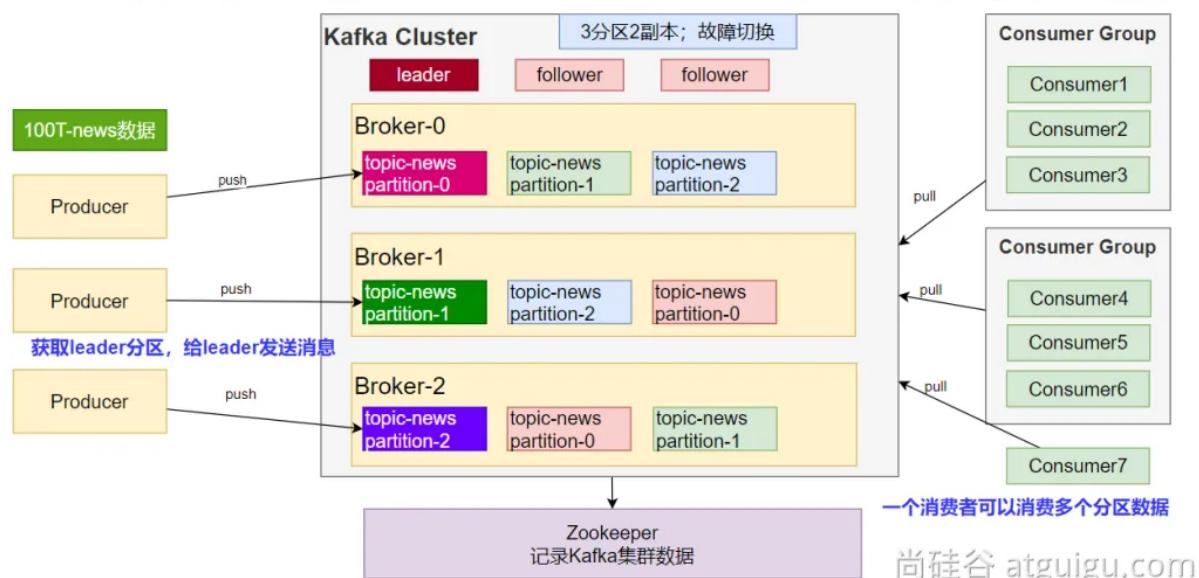


## Kafka工作原理

### Kafka原理

分区：海量数据分散存储  
副本：每个数据区都有备份

同一个消费者组里面的消费者是队列竞争模式  
不同消费者组里面的消费者是发布/订阅模式



## SpringBoot整合

参照：<https://docs.spring.io/spring-kafka/docs/current/reference/>

```
<dependency>
 <groupId>org.springframework.kafka</groupId>
 <artifactId>spring-kafka</artifactId>
</dependency>
```

配置

```
spring.kafka.bootstrap-servers=192.168.110.130:9092
```

修改 C:\Windows\System32\drivers\etc\hosts 文件，配置 192.168.110.130(自己的服务器IP地址)  
kafka

## 消息发送

```
@SpringBootTest
class Boot07KafkaApplicationTests {

 @Autowired
 KafkaTemplate kafkaTemplate;
 @Test
 void contextLoads() throws ExecutionException, InterruptedException {
 Stopwatch watch = new Stopwatch();
 watch.start();
 CompletableFuture[] futures = new CompletableFuture[10000];
 for (int i = 0; i < 10000; i++) {
 CompletableFuture send = kafkaTemplate.send("order",
"order.create."+i, "订单创建了: "+i);
 futures[i]=send;
 }
 CompletableFuture.allOf(futures).join();
 watch.stop();
 System.out.println("总耗时: "+watch.getTotalTimeMillis());
 }

}
```

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

 private final KafkaTemplate<String, String> kafkaTemplate;

 public MyBean(KafkaTemplate<String, String> kafkaTemplate) {
 this.kafkaTemplate = kafkaTemplate;
 }

 public void someMethod() {
 this.kafkaTemplate.send("someTopic", "Hello");
 }

}
```

## 消息监听

```
@Component
public class OrderMsgListener {

 @KafkaListener(topics = "order", groupId = "order-service")
 public void listen(ConsumerRecord record){
 System.out.println("收到消息: "+record); //可以监听到发给kafka的新消息，以前的
拿不到
 }

 @KafkaListener(groupId = "order-service-2", topicPartitions = {
 @TopicPartition(topic = "order", partitionOffsets = {
 @PartitionOffset(partition = "0", initialOffset = "0")
 })
 })
 public void listenAll(ConsumerRecord record){
 System.out.println("收到partition-0消息: "+record);
 }
}
```

## 参数配置

消费者

```
spring.kafka.consumer.value-
deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties[spring.json.value.default.type]=com.example.In
voice
spring.kafka.consumer.properties[spring.json.trusted.packages]=com.example.main
,com.example.another
```

生产者

```
spring.kafka.producer.value-
serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties[spring.json.add.type.headers]=false
```

## 自动配置原理

Kafka 自动配置在 `KafkaAutoConfiguration`

- 容器中放了 `KafkaTemplate` 可以进行消息收发
- 容器中放了 `KafkaAdmin` 可以进行 Kafka 的管理，比如创建 topic 等
- kafka 的配置在 `KafkaProperties` 中
- `@EnableKafka` 可以开启基于注解的模式

## Web安全

- Apache Shiro
- Spring Security
- 自研：Filter

# Spring Security

## 安全架构

### 认证：Authentication

Who are you?

登陆系统, 用户系统

### 授权：Authorization

what are you allowed to do?

权限管理, 用户授权

## 攻击防护

- XSS (Cross-site scripting)
- CSRF (Cross-site request forgery)
- CORS (Cross-Origin Resource Sharing)
- SQL注入

## 权限模型

### RBAC(Role Based Access Control)

- 用户 (t\_user)
  - id,username,password, xxx
    - 1,zhangsan
    - 2,lisi
- 用户\_角色 (t\_user\_role) 【N对N关系需要中间表】
  - zhangsan, admin
  - zhangsan,common\_user
  - lisi, hr
  - lisi, common\_user
- 角色 (t\_role)
  - id,role\_name
    - admin
    - hr
    - common\_user
- 角色\_权限(t\_role\_perm)
  - admin, 文件r
    - admin, 文件w

- admin, 文件执行
- admin, 订单query, create,xxx
- hr, 文件r
- 权限 (t\_permission)
- ◦ id,perm\_id
- 文件 r,w,x
- 订单 query,create,xxx

## ACL(Access Control List)

直接用户和权限挂钩

- 用户 (t\_user)
- ◦ zhangsan
  - lisi
- 用户\_权限(t\_user\_perm)
- ◦ zhangsan,文件 r
  - zhangsan,文件 x
  - zhangsan,订单 query
- 权限 (t\_permission)
- ◦ id,perm\_id
  - 文件 r,w,x
  - 订单 query,create,xxx

```
@Secured("文件 r")
public void readFile(){
 //读文件
}
```

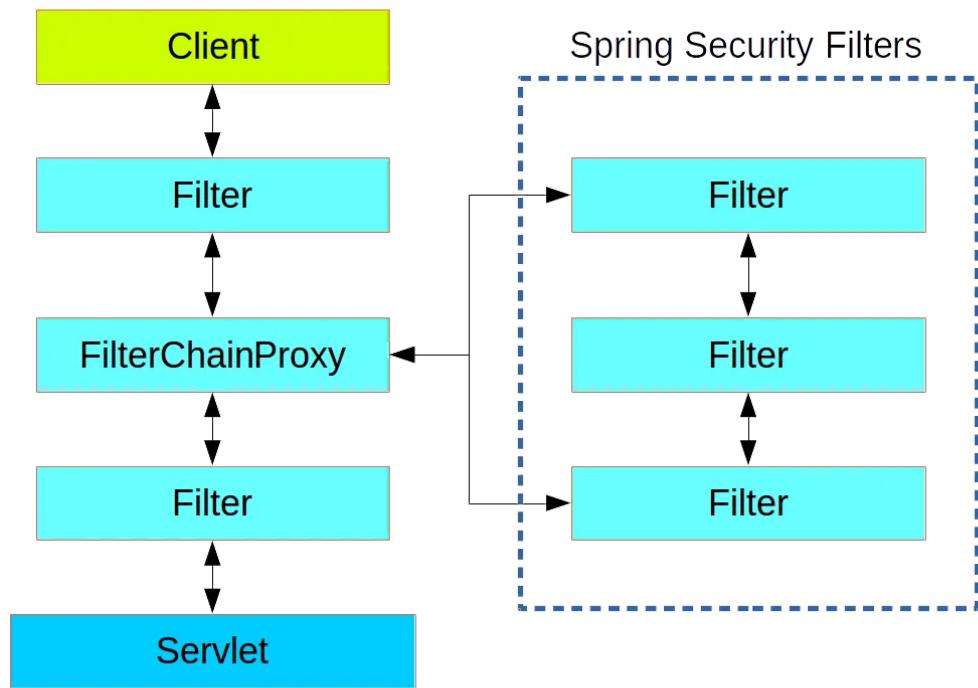
## Spring Security 原理

```
// security 场景的自动配置类
// 1. SecurityAutoConfiguration
// 2. SpringBootWebSecurityConfiguration
// 3. SecurityFilterAutoConfiguration

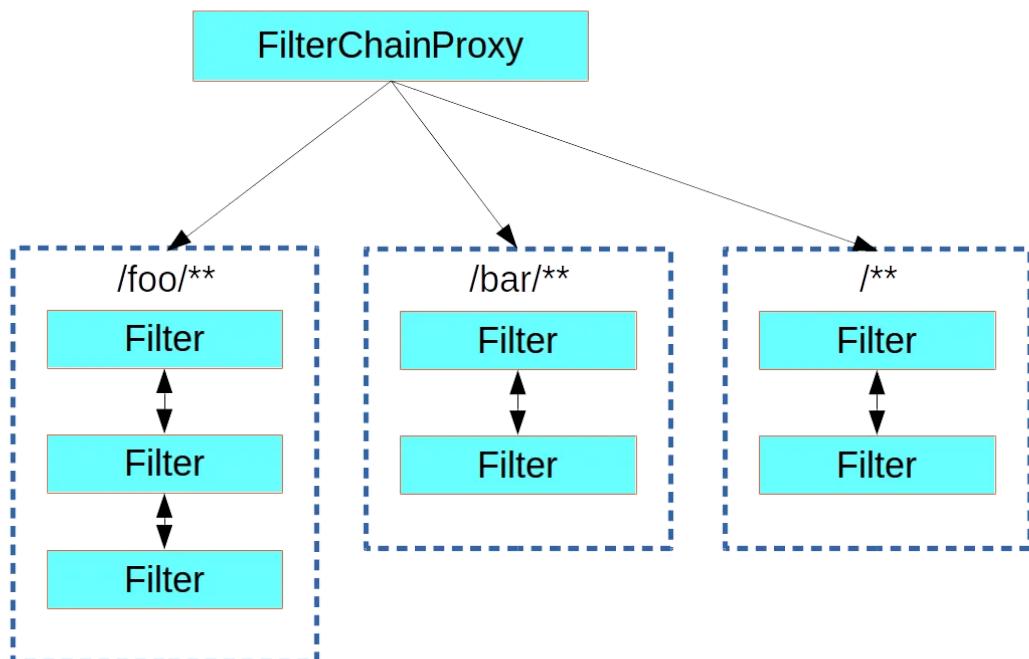
// security 的所有配置都在 SecurityProperties 中, 以spring.security开头
// 默认 SecurityFilterChain 组件配置:
// - 所有请求都需要认证
// - 开启表单登录, Spring Security 提供一个默认登录页, 未经登录的所有请求都需要登录
// - 使用 httpBasic 方式登录
// @EnableSecurity 生效:
// - webSecurityConfiguration 生效, web安全配置
// - HttpSecurityConfiguration 生效, HTTP安全配置
// - EnableGlobalAuthentication 全局认证生效
// - AuthenticationConfiguration 认证配置生效
```

## 过滤器链架构

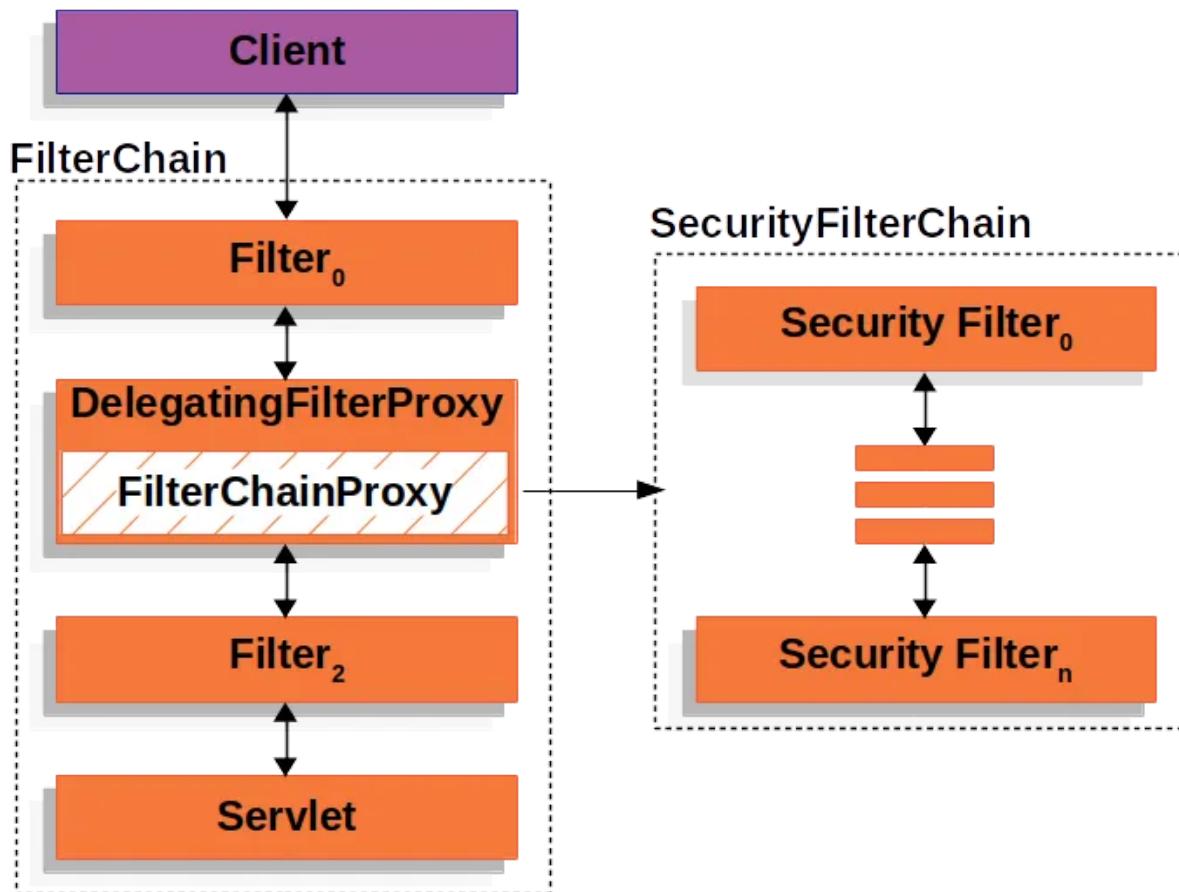
Spring Security利用 FilterChainProxy 封装一系列拦截器链，实现各种安全拦截功能  
Servlet三大组件：Servlet、Filter、Listener



## FilterChainProxy



## SecurityFilterChain



## 使用

### 引入依赖

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

### 配置类

```
/***
 * 1. 自定义授权规则: http.authorizeHttpRequests()
 * 2. 自定义登陆页: http.formLogin()
 * 3. 自定义用户信息查询规则: UserDetailsService
 * 4. 开启方法级别的精确权限控制: @EnableMethodSecurity +
 * @PreAuthorize("hasAuthority('file_write')")
 */

@EnableMethodSecurity
@Configuration
public class AppSecurityConfiguration {

 // 配置安全过滤器链
 @Bean
 SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
 {
 http.authorizeHttpRequests(
 registry -> {
```

```

 registry.requestMatchers("/*").permitAll()
 .anyRequest().authenticated();
 }

);

// 配置自定义登录页
http.formLogin(fromLogin -> {
 fromLogin.loginPage("/login")
 .permitAll();
});

return http.build();
}

// 配置用户详情查询
@Bean
UserDetailsService userDetailsService() {
 // 创建用户信息
 UserDetails zhangsan = User.withUsername("zhangsan")
 .password(passwordEncoder().encode("123456"))
 .roles("admin", "hr") // 角色
 .authorities("file_write", "file_read") // 权限
 .build();

 // 在内存中保存用户信息
 InMemoryUserDetailsManager manager = new
 InMemoryUserDetailsManager(zhangsan);

 return manager;
}

// 配置密码加密器
@Bean
PasswordEncoder passwordEncoder() {
 return new BCryptPasswordEncoder();
}
} // 分配权限

```

## HttpSecurity

通过 `ApplicationConfigurerAdapter` 继承 `WebSecurityConfigurerAdapter` 实现 `config` 方法，进行HTTP安全性的配置

```

@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends WebSecurityConfigurerAdapter
{

 // 配置HTTP安全性
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http.antMatcher("/match1/**") // 匹配以"/match1/"开头的所有路径
 .authorizeRequests() // 开始配置请求授权规则
 .antMatchers("/match1/user").hasRole("USER") // 针对"/match1/user"路径,
要求用户具有"USER"角色
 .antMatchers("/match1/spam").hasRole("SPAM") // 针对"/match1/spam"路径,
要求用户具有"SPAM"角色
 .anyRequest().isAuthenticated(); // 其他任何请求都需要经过认证
 }
}

```

```
}
```

## MethodSecurity

在方法下添加 `@Secured`、`@PreAuthorize`、`@PostAuthorize` 进行权限控制

```
@SpringBootApplication
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SampleSecureApplication {}

@Service
public class MyService {

 // 在MyService类中，用@Secured注解标记secure()方法，要求用户具有"ROLE_USER"角色才能访问
 @Secured("ROLE_USER")
 // 可以使用@PreAuthorize或@PostAuthorize注解来定义更复杂的方法级安全性规则
 // @PreAuthorize("hasAuthority('file_write')" | "hasRole('admin')")
 // @PostAuthorize("hasAuthority('file_read')" | "hasRole('admin')")
 public String secure() {
 return "Hello Security";
 }
}
```

核心

- `WebSecurityConfigurerAdapter`
- `@EnableGlobalMethodSecurity`: 开启全局方法安全配置
- ◦ `@Secured`
- `@PreAuthorize`
- `@PostAuthorize`
- `UserDetailsService`: 去数据库查询用户详细信息的service (用户基本信息、用户角色、用户权限)

## 可观测性

### SpringBoot Actuator

#### 实战

#### 场景引入

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

## 暴露指标

```
management:
 endpoints:
 enabled-by-default: true #暴露所有端点信息
 web:
 exposure:
 include: '*' #以web方式暴露
```

## 访问数据

- 访问 <http://localhost:8080/actuator>；展示出所有可以用的监控端点
- <http://localhost:8080/actuator/beans>
- <http://localhost:8080/actuator/configprops>
- <http://localhost:8080/actuator/metrics>
- <http://localhost:8080/actuator/metrics/jvm.gc.pause>
- <http://localhost:8080/actuator/endpointName/detailPath>

## Endpoint

### 常用端点

ID	描述
<code>auditevents</code>	暴露当前应用程序的审核事件信息。需要一个 <code>AuditEventRepository</code> 组件。
<code>beans</code>	显示应用程序中所有 Spring Bean 的完整列表。
<code>caches</code>	暴露可用的缓存。
<code>conditions</code>	显示自动配置的所有条件信息，包括匹配或不匹配的原因。
<code>configprops</code>	显示所有 <code>@ConfigurationProperties</code> 。
<code>env</code>	暴露 Spring 的属性 <code>ConfigurableEnvironment</code>
<code>flyway</code>	显示已应用的所有 Flyway 数据库迁移。需要一个或多个 <code>Flyway</code> 组件。
<code>health</code>	显示应用程序运行状况信息。
<code>httptrace</code>	显示 HTTP 跟踪信息（默认情况下，最近 100 个 HTTP 请求-响应）。需要一个 <code>HttpTraceRepository</code> 组件。
<code>info</code>	显示应用程序信息。
<code>integrationgraph</code>	显示 Spring <code>IntegrationGraph</code> 。需要依赖 <code>spring-integration-core</code> 。
<code>loggers</code>	显示和修改应用程序中日志的配置。
<code>liquibase</code>	显示已应用的所有 Liquibase 数据库迁移。需要一个或多个 <code>Liquibase</code> 组件。
<code>metrics</code>	显示当前应用程序的“指标”信息。
<code>mappings</code>	显示所有 <code>@RequestMapping</code> 路径列表。
<code>scheduledtasks</code>	显示应用程序中的计划任务。

ID	描述
sessions	允许从Spring Session支持的会话存储中检索和删除用户会话。需要使用Spring Session的基于Servlet的Web应用程序。
shutdown	使应用程序正常关闭。默认禁用。
startup	显示由 <a href="#">ApplicationStartup</a> 收集的启动步骤数据。需要使用 <a href="#">SpringApplication</a> 进行配置 <a href="#">BufferingApplicationStartup</a> 。
threaddump	执行线程转储。
heapdump	返回 <a href="#">hprof</a> 堆转储文件。
jolokia	通过HTTP暴露JMX bean (需要引入Jolokia, 不适用于WebFlux)。需要引入依赖 <a href="#">jolokia-core</a> 。
logfile	返回日志文件的内容 (如果已设置 <a href="#">logging.file.name</a> 或 <a href="#">logging.file.path</a> 属性)。支持使用HTTP Range标头来检索部分日志文件的内容。
prometheus	以Prometheus服务器可以抓取的格式公开指标。需要依赖 <a href="#">micrometer-registry-prometheus</a> 。

[threaddump](#)、[heapdump](#)、[metrics](#)

## 定制端点

- 健康监控：返回存活、死亡
- 指标监控：次数、率

### HealthEndpoint

定义了一个名为[MyHealthIndicator](#) 的健康指标 (Health Indicator) 类，实现了 Spring Boot 的 [HealthIndicator](#) 接口。在 [health](#) 方法中，根据执行 [check\(\)](#) 特定的健康检查操作后返回不同的 [Health](#) 状态。如果健康检查出现错误 ([errorCode](#) 不为 0)，则返回一个健康状态为 [down](#)，并附带错误信息；否则返回一个健康状态为 [up](#)

```

@Component
public class MyHealthIndicator implements HealthIndicator {
 @Override
 public Health health() {
 int errorCode = check(); // perform some specific health check
 if (errorCode != 0) {
 return Health.down().withDetail("Error Code", errorCode).build();
 }
 return Health.up().build();
 }
}

// 构建Health
// Health build = Health.down()
// .withDetail("msg", "error service")
// .withDetail("code", "500")
// .withException(new RuntimeException())
// .build();

```

`health.enabled: true` 表示启用健康监控功能，允许应用程序暴露健康检查端点。  
`health.show-details: always` 设置为 `always` 表示总是显示详细信息，无论是健康状态正常还是异常，都会显示每个模块的状态信息。这样可以更详细地了解每个模块的健康状态，有助于排查问题和监控系统的运行状况。

```
management:
 health:
 enabled: true
 show-details: always #总是显示详细信息。可显示每个模块的状态信息
```

自定义的健康指标（Health Indicator）类 `MyComHealthIndicator`，它继承了 Spring Boot 的 `AbstractHealthIndicator` 抽象类。在这个类中，重写了抽象方法 `doHealthCheck`，该方法用于执行实际的健康检查操作。

在 `doHealthCheck` 方法中，首先进行了一个模拟的健康检查操作，这里以检查 MongoDB 连接为例。根据检查结果，如果条件为假 (`if(1 == 2)`)，表示连接超时或出错，此时通过 `builder.status(Status.OUT_OF_SERVICE)` 将健康状态设置为 `OUT_OF_SERVICE`，同时添加了错误信息和耗时等详细信息到健康状态中。如果条件为真，表示连接正常，通过 `builder.status(Status.UP)` 将健康状态设置为 `UP`，并添加相应的详细信息。

```
@Component
public class MyComHealthIndicator extends AbstractHealthIndicator {
 /**
 * 真实的检查方法
 * @param builder
 * @throws Exception
 */
 @Override
 protected void doHealthCheck(Health.Builder builder) throws Exception {
 //mongodb。 获取连接进行测试
 Map<String, Object> map = new HashMap<>();
 // 检查完成
 if(1 == 2){
 // builder.up(); //健康
 builder.status(Status.UP);
 map.put("count", 1);
 map.put("ms", 100);
 }else {
 // builder.down();
 builder.status(Status.OUT_OF_SERVICE);
 map.put("err", "连接超时");
 map.put("ms", 3000);
 }

 builder.withDetail("code", 100)
 .withDetails(map);
 }
}
```

## MetricsEndpoint

当 `Controller` 调用 `Myservice` 中的 `hello` 时，就会增加 `counter` 的计数，此时访问 `/actuator/metrics/myservice.method.running.counter` 就会显现请求的次数

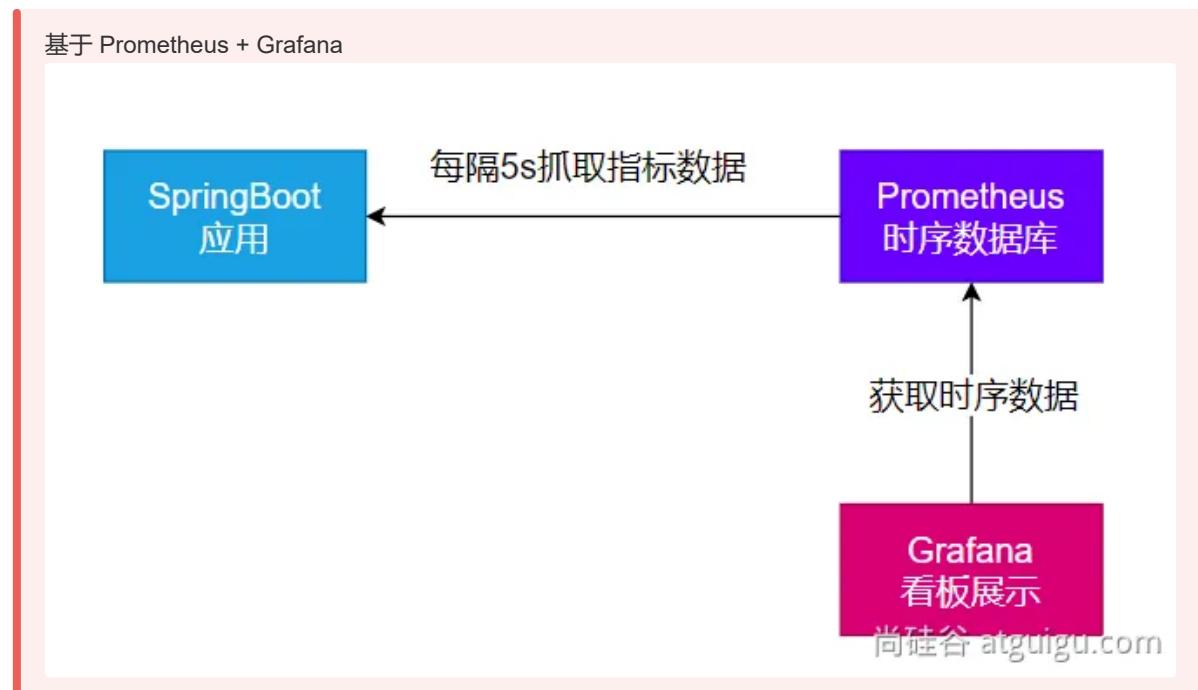
```

class Myservice{
 Counter counter;
 public MyService(MeterRegistry meterRegistry){
 counter = meterRegistry.counter("myservice.method.running.counter");
 }

 public void hello() {
 counter.increment();
 }
}

```

## 监控案例落地



## 安装 Prometheus + Grafana

```

#安装prometheus:时序数据库
docker run -p 9090:9090 -d \
-v /etc/prometheus \
prom/prometheus

#安装grafana; 默认账号密码 admin:admin
docker run -d --name=grafana -p 3000:3000 grafana/grafana

```

## 导入依赖

```

<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
 <groupId>io.micrometer</groupId>
 <artifactId>micrometer-registry-prometheus</artifactId>
 <version>1.10.6</version>
</dependency>

```

```
management:
 endpoints:
 web:
 exposure: #暴露所有监控的端点
 include: '*'
```

访问: <http://localhost:8000/actuator/prometheus> 验证, 返回 prometheus 格式的所有指标

在linux部署 Java 应用

```
#安装上传工具
yum install lrzs

#安装openjdk
下载openjdk17
wget https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.tar.gz

mkdir -p /opt/java
tar -xzf jdk-17_linux-x64_bin.tar.gz -C /opt/java/
sudo vi /etc/profile
#加入以下内容
export JAVA_HOME=/opt/java/jdk-17.0.7
export PATH=$PATH:$JAVA_HOME/bin

#环境变量生效
source /etc/profile

后台启动java应用
nohup java -jar boot3-14-actuator-0.0.1-SNAPSHOT.jar > output.log 2>&1 &
```

确认可以访问到部署在linux上的项目: <http://8.130.32.70:9999/actuator/prometheus>

## 配置 Prometheus 拉取数据

```
修改 prometheus.yml 配置文件
scrape_configs:
 - job_name: 'spring-boot-actuator-exporter'
 metrics_path: '/actuator/prometheus' #指定抓取的路径
 static_configs:
 - targets: ['192.168.200.1:8001']
 labels:
 nodename: 'app-demo'
```

## 配置 Grafana 监控面板

- 添加数据源 (Prometheus)
- 添加面板。可去 dashboard 市场找一个自己喜欢的面板, 也可以自己开发面板;[Dashboards | Grafana Labs](#)



## AOT

### AOT与JIT

AOT: Ahead-of-Time (提前编译) : 程序执行前, 全部被编译成机器码

JIT: Just in Time (即时编译) : 程序边编译, 边运行

编译:

- 源代码 (.c、.cpp、.go、.java ...) ==> 编译器 ==> 机器码

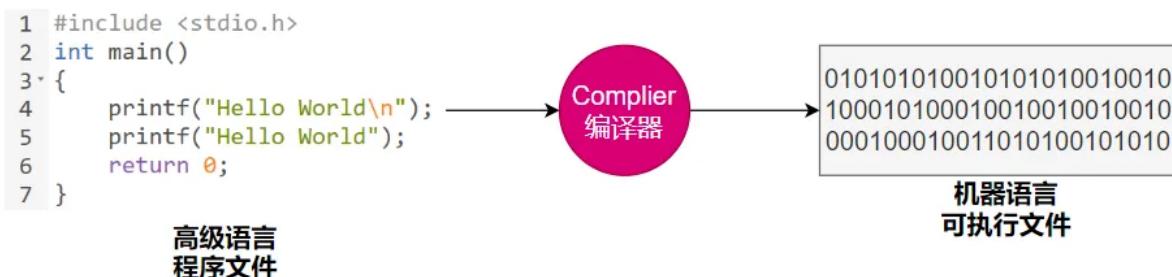
语言

- 编译型语言: 编译器
- 解释型语言: 解释器

## Complier 与 Interpreter

Java: 半编译半解释

<https://anycodes.cn/editor>



对比项	编译器	解释器
机器执行速度	快，因为源代码只需被转换一次	慢，因为每行代码都需要被解释执行
开发效率	慢，因为需要耗费大量时间编译	快，无需花费时间生成目标代码，更快的开发和测试
调试	难以调试编译器生成的目标代码	容易调试源代码，因为解释器一行一行地执行
可移植性（跨平台）	不同平台需要重新编译目标平台代码	同一份源码可以跨平台执行，因为每个平台会开发对应的解释器
学习难度	相对较高，需要了解源代码、编译器以及目标机器的知识	相对较低，无需了解机器的细节
错误检查	编译器可以在编译代码时检查错误	解释器只能在执行代码时检查错误
运行时增强	无	可以动态增强

## AOT 与 JIT 对比

	JIT	AOT
优点	1. 具备适时调整能力 2. 生成最优机器指令 3. 根据代码运行情况优化内存占用	1. 速度快，优化了运行时编译时间和内存消耗 2. 程序初期就能达到最高性能 3. 加快程序启动速度
缺点	1. 运行期间编译速度慢 2. 初始编译不能达到最高性能	1. 程序第一次编译占用时间长 2. 牺牲高级语言一些特性

在 OpenJDK 的官方 Wiki 上，介绍了 HotSpot 虚拟机一个相对比较全面的、**即时编译器 (JIT)** 中采用的**优化技术列表**。

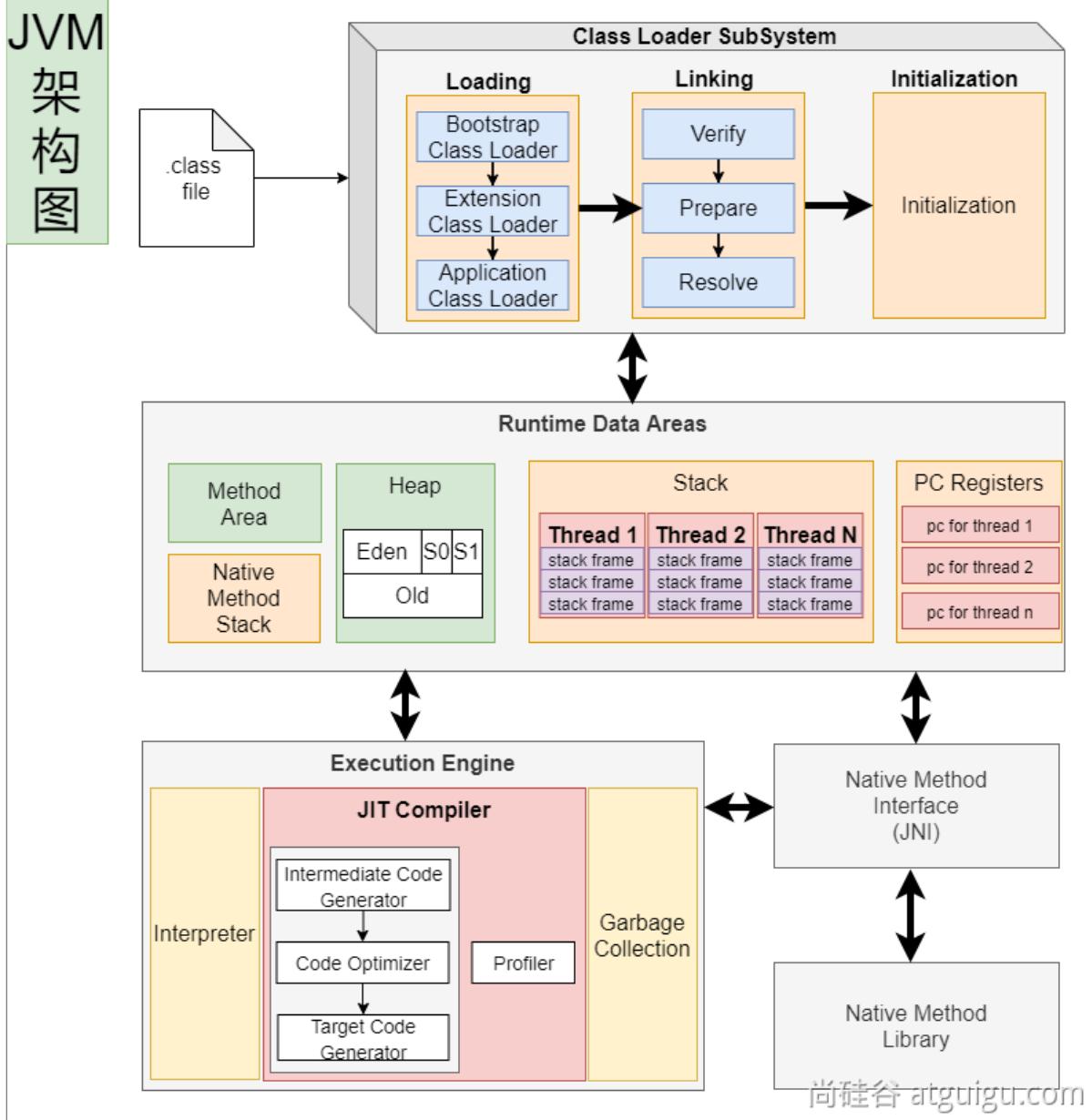
类 型	优 化 技 术
语言相关的优化技术 (Language-Specific Techniques)	锁膨胀 (Lock Coarsening)
	消除反射 (De-Reflection)
内存及代码位置变换 (Memory And Placement Transformation)	表达式提升 (Expression Hoisting)
	表达式下沉 (Expression Sinking)
	冗余存储消除 (Redundant Store Elimination)
	相邻存储合并 (Adjacent Store Fusion)
	交汇点分离 (Merge-Point Splitting)
	循环展开 (Loop Unrolling)
循环变换 (Loop Transformations)	循环剥离 (Loop Peeling)
	安全点消除 (Safepoint Elimination)
	迭代范围分离 (Iteration Range Splitting)
	范围检查消除 (Range Check Elimination)
	循环向量化 (Loop Vectorization) 尚硅谷 atguigu.com

全局代码调整 (Global Code Shaping)	内联 (Inlining)
	全局代码外提 (Global Code Motion)
	基于热度的代码布局 (Heat-Based Code Layout)
	Switch 调整 (Switch Balancing)
控制流图变换 (Control Flow Graph Transformation)	本地代码编排 (Local Code Scheduling)
	本地代码封包 (Local Code Bundling)
	延迟槽填充 (Delay Slot Filling)
	着色图寄存器分配 (Graph-Coloring Register Allocation)
	线性扫描寄存器分配 (Linear Scan Register Allocation)
	复写聚合 (Copy Coalescing)
	常量分裂 (Constant Splitting)
	复写移除 (Copy Removal)
	地址模式匹配 (Address Mode Matching)
	指令窥孔优化 (Instruction Peephole)
	基于确定有限状态机的代码生成 (DFA-based Code Generation)

可使用: -XX:+PrintCompilation 打印JIT编译信息

## JVM 架构

.java === .class === 机器码



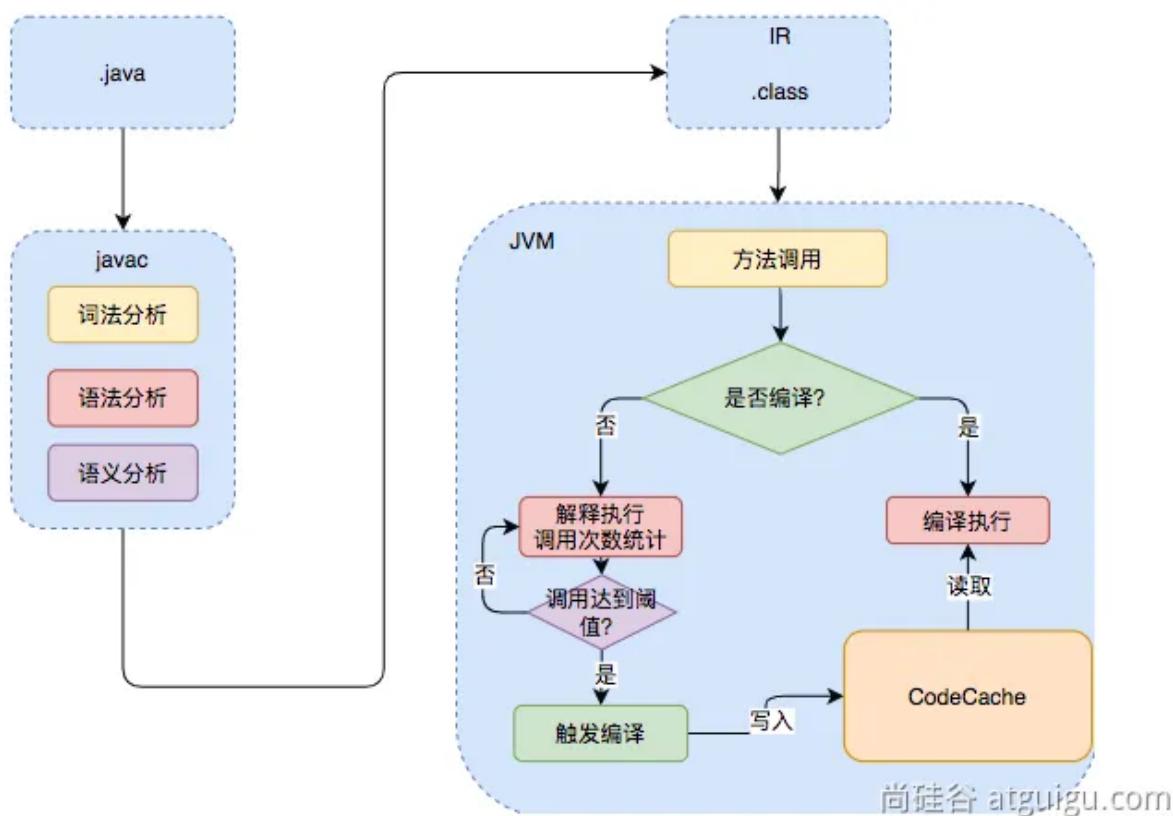
JVM: 既有解释器，又有编辑器（JIT：即时编译）；

## Java的执行过程

建议阅读：

- 美团技术：<https://tech.meituan.com/2020/10/22/java-jit-practice-in-meituan.html>
- openjdk官网：<https://wiki.openjdk.org/display/HotSpot/Compiler>

## 流程概要

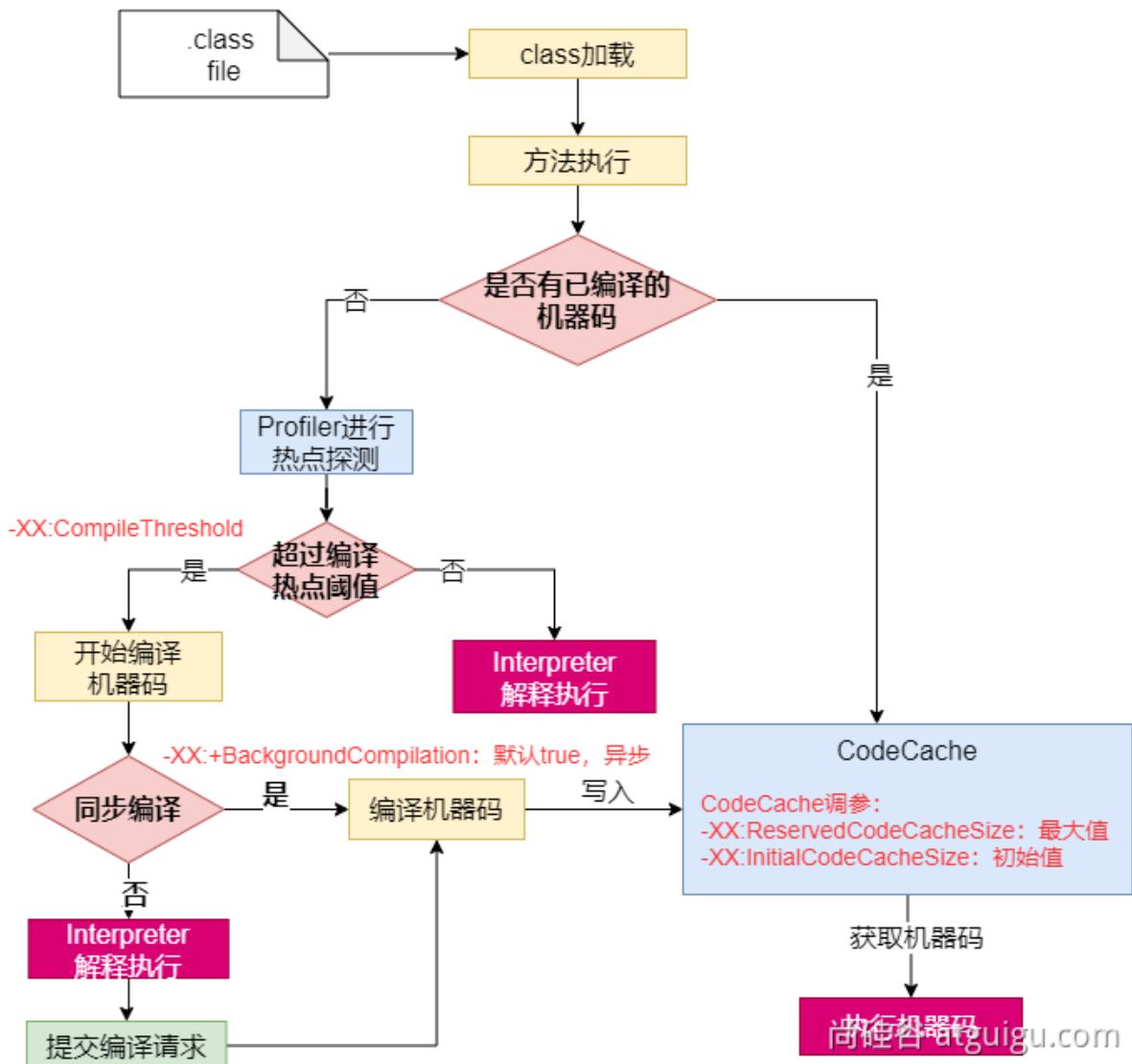


解释执行：

编译执行：

## 详细流程

热点代码：调用次数非常多的代码



## JVM编译器

JVM中集成了两种编译器，Client Compiler 和 Server Compiler；

- Client Compiler注重启动速度和局部的优化
- Server Compiler更加关注全局优化，性能更好，但由于会进行更多的全局分析，所以启动速度会慢。

Client Compiler：

- HotSpot VM带有一个Client Compiler **C1编译器**
- 这种编译器**启动速度快**，但是性能比较Server Compiler来说会差一些。
- 编译后的机器码执行效率没有C2的高

Server Compiler：

- Hotspot虚拟机中使用的Server Compiler有两种：**C2** 和 **Graal**。
- 在Hotspot VM中，默认的Server Compiler是**C2编译器**。

## 分层编译

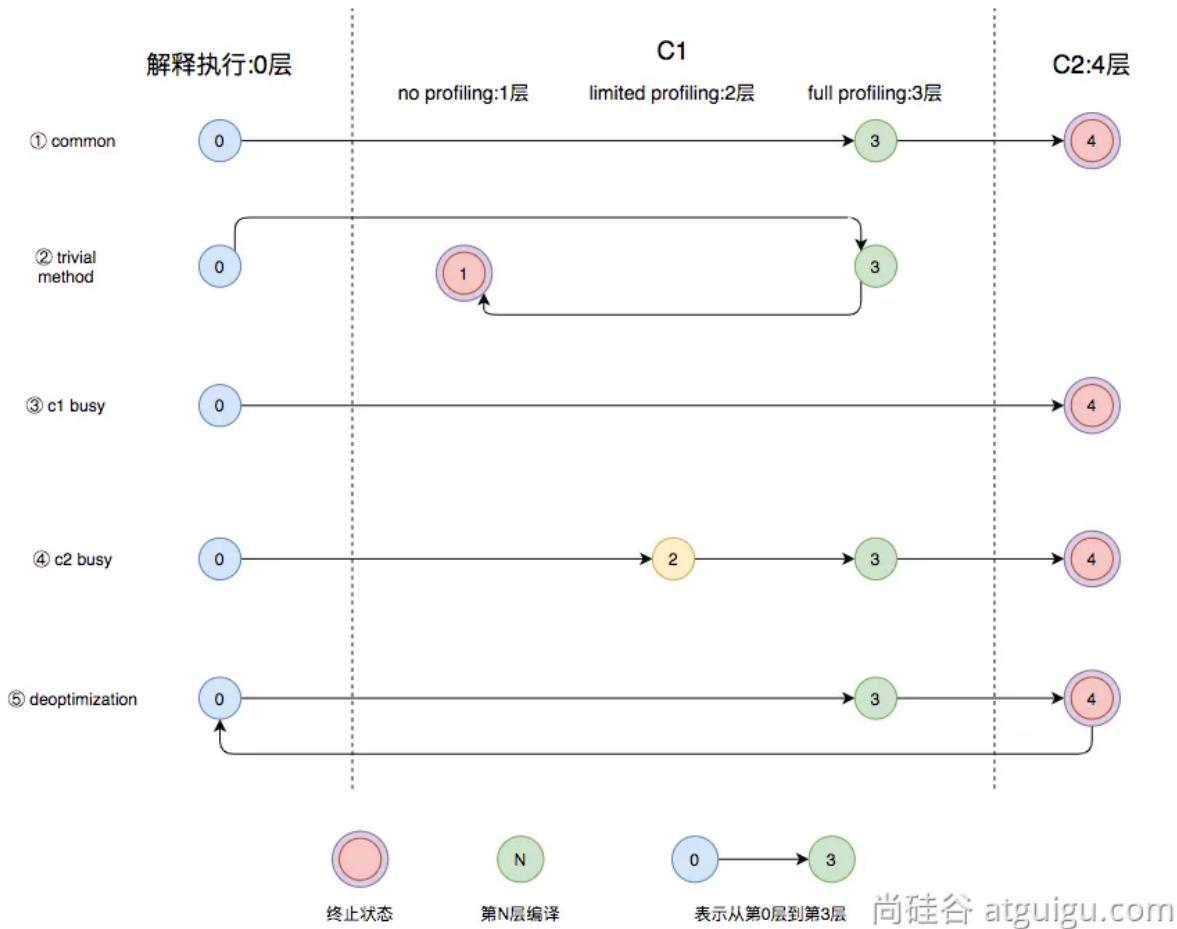
Java 7开始引入了分层编译(**Tiered Compiler**)的概念，它结合了**C1**和**C2**的优势，追求启动速度和峰值性能的一个平衡。分层编译将JVM的执行状态分为了五个层次。五个层级分别是：

- 解释执行。
- 执行不带profiling的C1代码。
- 执行仅带方法调用次数以及循环回边执行次数profiling的C1代码。

- 执行带所有profiling的C1代码。

- 执行C2代码。

**profiling**就是收集能够反映程序执行状态的数据。其中最基本的统计数据就是方法的调用次数，以及循环回边的执行次数。



- 图中第①条路径，代表编译的一般情况，**热点方法**从解释执行到被3层的C1编译，最后被4层的C2编译。
- 如果**方法比较小**（比如Java服务中常见的**getter/setter**方法），3层的profiling没有收集到有价值的数据，JVM就会断定该方法对于C1代码和C2代码的执行效率相同，就会执行图中第②条路径。在这种情况下，JVM会在3层编译之后，放弃进入C2编译，**直接选择用1层的C1编译运行**。
- 在**C1忙碌**的情况下，执行图中第③条路径，在解释执行过程中对程序进行**profiling**，根据信息直接由第4层的**C2编译**。
- 前文提到C1中的执行效率是**1层>2层>3层**，**第3层**一般要比**第2层**慢35%以上，所以在**C2忙碌**的情况下，执行图中第④条路径。这时方法会被2层的C1编译，然后再被3层的C1编译，以减少方法在**3层**的执行时间。
- 如果**编译器**做了一些**比较激进的优化**，比如分支预测，在实际运行时**发现预测出错**，这时就会进行**反优化**，重新进入**解释执行**，图中第⑤条执行路径代表的就是**反优化**。

总的来说，C1的编译速度更快，C2的编译质量更高，分层编译的不同编译路径，也就是JVM根据当前服务的运行情况来寻找当前服务的最佳平衡点的一个过程。从JDK 8开始，JVM默认开启分层编译。

**云原生**: Cloud Native; Java小改版;

最好的效果：

存在的问题：

- java应用如果用jar，解释执行，热点代码才编译成机器码；初始启动速度慢，初始处理请求数量少。
- 大型云平台，要求每一种应用都必须秒级启动。每个应用都要求效率高。

希望的效果：

- java应用也能提前被编译成**机器码**，随时**急速启动**，一启动就急速运行，最高性能
- 编译成机器码的好处：
  - 另外的服务器还需要安装Java环境
  - 编译成**机器码**的，可以在这个平台 Windows X64 **直接运行**。

原生镜像: native-image (机器码、本地镜像)

- 把应用打包成能适配本机平台的可执行文件 (机器码、本地镜像)

## GraalVM

<https://www.graalm.org/>

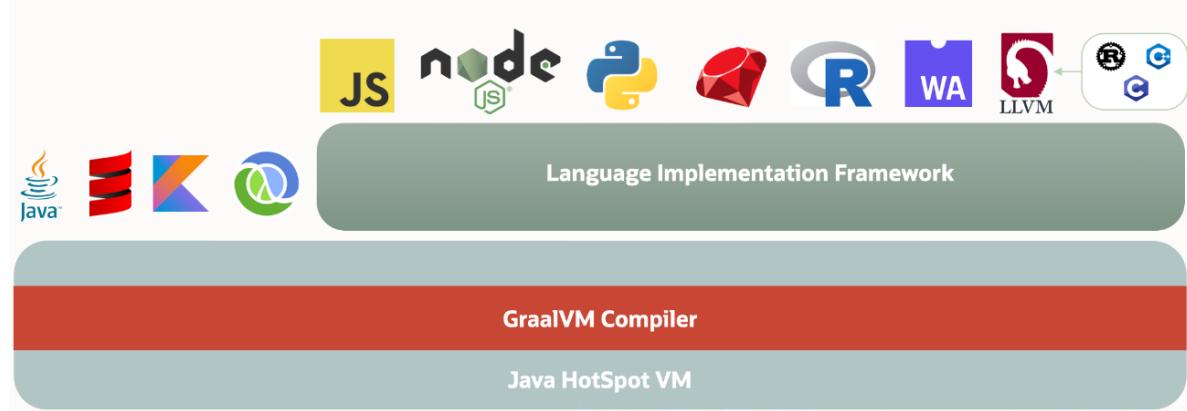
GraalVM是一个高性能的JDK，旨在加速用Java和其他JVM语言编写的**应用程序的执行**，同时还提供JavaScript、Python和许多其他流行语言的运行时。

GraalVM提供了**两种运行Java应用程序的方式**:

- 1. 在HotSpot JVM上使用**Graal即时 (JIT) 编译器**
- 2. 作为**预先编译 (AOT)** 的**本机可执行文件运行 (本地镜像)**。

GraalVM的多语言能力使得在单个应用程序中混合多种编程语言成为可能，同时消除了外部语言调用的成本。

## 架构



## 安装

跨平台提供原生镜像原理:

gu install native-image: GraalVM 提供 native-image 工具

VisualStudio  
提供C++集成环境

Windows平台

gcc  
glibc-devel  
zlib-devel

Linux平台

Xcode  
提供集成环境

Mac平台

GraalVM

# VisualStudio

<https://visualstudio.microsoft.com/zh-hans/free-developer-offers/>

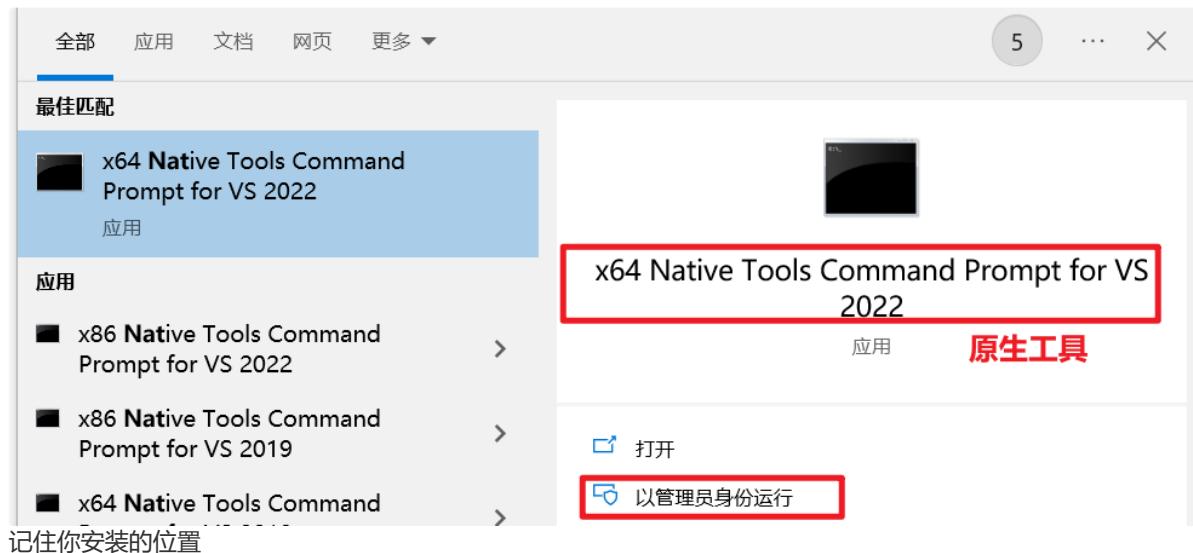
The screenshot shows the Visual Studio 2022 download page. It features two main sections: 'Community' (left) and 'Professional' (right). The 'Community' section is highlighted with a red border and contains a large '免费下载' (Free Download) button. The 'Professional' section contains a '免费试用' (Free Trial) button. To the right, a 'Visual Studio Installer' window is open, showing a progress bar for downloading the Visual Studio setup.

[发行说明](#) → [版本比较](#) → [如何脱机安装](#) →

The screenshot shows the 'Visual Studio Installer' window for the 'Community' edition. Under the '必选' (Must Have) category, the '使用 C++ 的桌面开发' (Desktop development with C++) and '通用 Windows 平台开发' (Universal Windows Platform development) components are selected. In the '游戏' (Games) category, '使用 Unity 的游戏开发' (Unity game development) is selected. On the right side, the '安装详细信息' (Detailed Information) pane lists various development areas like ASP.NET and Web development, Python development, and C++ for Linux and embedded systems.

## 别选中文

The screenshot shows the 'Visual Studio Installer' window for the 'Community' edition. Under the '语言包' (Language Pack) tab, the '英语' (English) checkbox is selected. The '安装详细信息' (Detailed Information) pane on the right lists various optional components and tools for the C++ development environment, such as MSVC v143, ATL, and AddressSanitizer.



记住你安装的位置

## GraalVM

### 安装

下载 GraalVM + native-image

### GraalVM Community Edition 22.3.2

ezzarghili released this 3 weeks ago · vm-22.3.2 · 78e9d6d

GraalVM is a high performance JDK distribution. It is designed to accelerate the execution of applications written in Java and other JVM languages for JavaScript, Python, LLVM-based languages such as C and C++, and a number of other popular languages. Additionally, Graal interoperability between programming languages and compiling Java applications ahead-of-time into native executables for faster startup overhead.

This JDK distribution of GraalVM Community includes the Java Development Kit with the GraalVM compiler.

The GraalVM environment can be extended with optionally available components such as Native Image, JavaScript runtime (GraalJS), Node.js, WebAssembly, LLVM runtime, LLVM Toolchain, Java on truffle, Java on Truffle LLVM Java library and VisualVM using the [GraalVM Updater](#).

The release notes can be found on the website: [https://www.graalm.org/latest/release-notes/22\\_3/#2232](https://www.graalm.org/latest/release-notes/22_3/#2232).

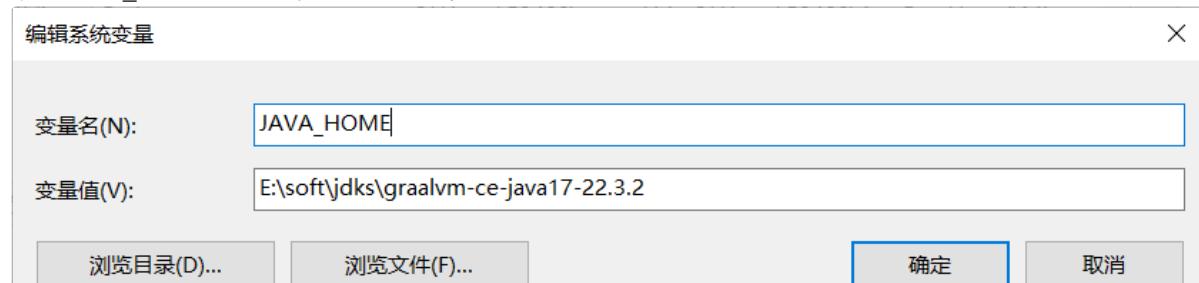
Here are the convenience links for the JDK base downloads of GraalVM:

Platform	Java 11	Java 17	
Linux (amd64)	<a href="#">download</a>	<a href="#">download</a>	<a href="#">instructions</a>
Linux (aarch64)	<a href="#">download</a>	<a href="#">download</a>	<a href="#">instructions</a>
macOS (amd64) +	<a href="#">download</a>	<a href="#">download</a>	<a href="#">instructions</a>
Windows (amd64)	<a href="#">download</a>	<a href="#">download</a>	<a href="#">instructions</a>

<a href="#">native-image-installable-svm-java17-darwin-amd64-22.3.2.jar.sha256</a>	64 Bytes	3 weeks ago
<a href="#">native-image-installable-svm-java17-linux-aarch64-22.3.2.jar</a>	28.6 MB	3 weeks ago
<a href="#">native-image-installable-svm-java17-linux-aarch64-22.3.2.jar.sha256</a>	64 Bytes	3 weeks ago
<a href="#">native-image-installable-svm-java17-linux-amd64-22.3.2.jar</a>	33 MB	3 weeks ago
<a href="#">native-image-installable-svm-java17-linux-amd64-22.3.2.jar.sha256</a>	64 Bytes	3 weeks ago
<a href="#">native-image-installable-svm-java17-windows-amd64-22.3.2.jar</a>	41.2 MB	3 weeks ago
<a href="#">native-image-installable-svm-java17-windows-amd64-22.3.2.jar.sha256</a>	64 Bytes	3 weeks ago
<a href="#">native-image-llvm-backend-installable-ce-java11-darwin-amd64-22.3.2.jar</a>	78.8 MB	3 weeks ago
<a href="#">native-image-llvm-backend-installable-ce-java11-darwin-amd64-22.3.2.jar.sha256</a>	64 Bytes	3 weeks ago
<a href="#">native-image-llvm-backend-installable-ce-java11-linux-aarch64-22.3.2.jar</a>	96 MB	3 weeks ago

## 配置

修改 JAVA\_HOME 与 Path, 指向新bin路径



在path中添加环境变量



验证JDK环境为GraalVM提供的即可：

```
C:\Users\lucki>java -version
openjdk version "17.0.3" 2022-04-19
OpenJDK Runtime Environment GraalVM CE 22.1.0 (build 17.0.3+7-jvmci-22.1-b06)
OpenJDK 64-Bit Server VM GraalVM CE 22.1.0 (build 17.0.3+7-jvmci-22.1-b06, mixed mode, sharing)
```

## 依赖

安装 native-image 依赖：

1. 网络环境好：参考：<https://www.graalvm.org/latest/reference-manual/native-image/#install-native-image>

```
gu install native-image
```

2. 网络不好，使用我们下载的离线jar: `native-image-xxx.jar` 文件

```
gu install --file native-image-installable-svm-java17-windows-amd64-22.3.2.jar
```

验证一下

```
native-image
```

## 测试

### 创建项目

- 1. 创建普通java项目。编写HelloWorld类；
- 使用 `mvn clean package` 进行打包
- 确认jar包是否可以执行 `java -jar xxx.jar`
- 可能需要给 `MANIFEST.MF` 添加 `Main-Class: 你的主类`

### 编译镜像

- 编译为原生镜像 (native-image) : 使用 `native-tools` 终端

```
管理员: x64 Native Tools Command Prompt for VS 2022
```

```
C:\Windows\System32>native-image --help
```

```
GraalVM Native Image (https://www.graalvm.org/native-image/)
```

```
This tool can ahead-of-time compile Java code to native executables.
```

```
Usage: native-image [options] class [imagename] [options]
 (to build an image for a class)
 or native-image [options] -jar jarfile [imagename] [options]
 (to build an image for a jar file)
 or native-image [options] -m <module>[/<mainclass>] [options]
 native-image [options] --module <module>[/<mainclass>] [options]
 (to build an image for a module)
```

where options include:

```
@argument files one or more argument files containing options
-cp <class search path of directories and zip/jar files>
-classpath <class search path of directories and zip/jar files>
--class-path <class search path of directories and zip/jar files>
A ; separated list of directories, JAR archives,
and ZIP archives to search for class files.
-p <module path>
```

```
#从入口开始，编译整个jar
native-image -cp xxxx.jar org.example.App -H:Name=xxx

#编译某个类【必须有main入口方法，否则无法编译】
native-image -cp .\classes org.example.App
```

## Linux平台测试

### 1. 安装gcc等环境

```
yum install lrzs
sudo yum install gcc glibc-devel zlib-devel
```

### 2. 下载安装配置Linux下的GraalVM、native-image

- 下载: <https://www.graalvm.org/downloads/>
- 安装: GraalVM、native-image
- 配置: JAVA环境变量为GraalVM

```
tar -zxf graalvm-ce-java17-linux-amd64-22.3.2.tar.gz -C /opt/java/
sudo vim /etc/profile
#修改以下内容
export JAVA_HOME=/opt/java/graalvm-ce-java17-22.3.2
export PATH=$PATH:$JAVA_HOME/bin

source /etc/profile
```

### 3. 安装native-image

```
gu install --file native-image-installable-svm-jar17-linux-amd64-22.3.2.jar
```

### 4. 使用native-image编译jar为原生程序

```
native-image -cp xxxx.jar org.example.App -H:Name=xxx
```

# SpringBoot整合

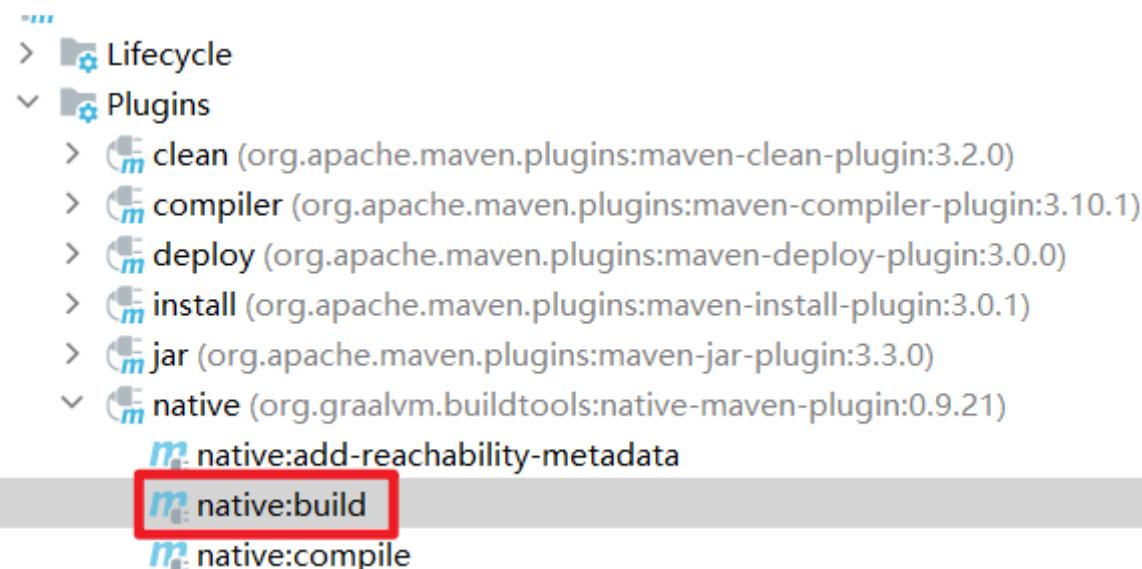
## 依赖导入

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

## 生产native-image

- 1、运行aot提前处理命令: `mvn springboot:process-aot`
- 2、运行native打包: `mvn -Pnative native:build`

```
推荐加上 -Pnative
mvn -Pnative native:build -f pom.xml
```



需要修改三个环境变量: `Path`、`INCLUDE`、`Lib`

- `Path`: 添加如下值
  - `C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.33.31629\bin\Hostx64\x64`
  - 你自己的安装位置
- 新建 `INCLUDE` 环境变量: 值为
  - 根据自己软件安装位置

```
◦ C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.33.31629\include;
C:\Program Files (x86)\Windows Kits\10\Include\10.0.19041.0\shared;C:\Program Files (x86)\Windows Kits\10\Include\10.0.19041.0\ucrt;
C:\Program Files (x86)\Windows Kits\10\Include\10.0.19041.0\um;
C:\Program Files (x86)\Windows Kits\10\Include\10.0.19041.0\winrt
```

- 新建 **Lib** 环境变量：值为

- 根据自己软件安装位置
- C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.33.31629\lib\x64;
C:\Program Files (x86)\Windows Kits\10\Lib\10.0.19041.0\um\x64;
C:\Program Files (x86)\Windows Kits\10\Lib\10.0.19041.0\ucrt\x64

## SpringBoot3响应式

### 前置知识

#### Lambda

Lambda表达式是Java8引入的一个重要特性

Lambda表达式可以被视为匿名函数

允许在需要函数的地方以更简洁的方式定义功能

```
(parameters) -> expression
(parameters) -> {statements;}
```

函数式接口：只要是函数式接口就可以用Lambda表达式简化

函数式接口：接口中有且只有一个未实现的方法，这个接口就叫函数式接口

**@FunctionalInterface**：检查注解，帮我们快速检查我们写的接口是否函数式接口

#### Function

##### 内置Function

BiConsumer	IntPredicate	ToDoubleBiFunction
BiFunction	IntSupplier	ToDoubleFunction
BinaryOperator	IntToDoubleFunction	ToIntBiFunction
BiPredicate	IntToLongFunction	ToIntFunction
BooleanSupplier	IntUnaryOperator	ToLongBiFunction
<b>Consumer</b>	LongBinaryOperator	ToLongFunction
DoubleBinaryOperator	LongConsumer	UnaryOperator
DoubleConsumer	LongFunction	
DoubleFunction	LongPredicate	
DoublePredicate	LongSupplier	
DoubleSupplier	LongToDoubleFunction	
DoubleToIntFunction	LongToIntFunction	
DoubleToLongFunction	LongUnaryOperator	
DoubleUnaryOperator	ObjDoubleConsumer	
<b>Function</b>	ObjIntConsumer	
IntBinaryOperator	ObjLongConsumer	
IntConsumer	<b>Predicate</b>	
IntFunction	Supplier	

函数式接口的出入参数定义：

1. 有入参，无出参【消费者】

```
Consumer<String> consumer = (a) -> {
 System.out.println(a);
}
consumer.accept("1");
```

## 2. 有入参，有出参【多功能函数】

```
Function<String, Integer> function = (x) -> {
 return Integer.parseInt(x);
}
```

## 3. 无入参，有出参

```
Supplier<String> supplier = () -> {
 return UUID.randomUUID().toString();
}
```

## 4. 无入参，无出参【纯同函数】

```
Runnable runnable = () -> System.out.println("aaa");
new Thread(runnable).start();
```

## 5. 断言

```
Predicate<Integer> predicate = (num) -> {
 // 判断是否是偶数，是则返回true
 return num % 2 == 0;
}
```

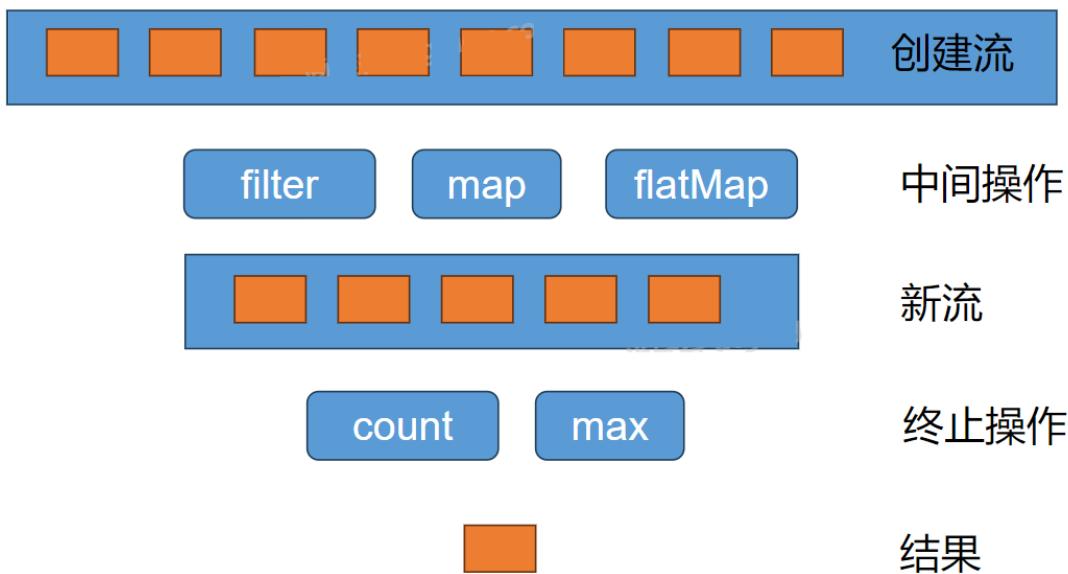
java.util.function包下的所有function定义:

- Consumer:消费者
- Supplier:提供者
- Function:多功能函数
- Predicate:断言

get/test/apply/accept调用的函数方法

## StreamAPI

声明式处理**集合数据**，包括：筛选、转换、组合等



最佳实践：以后凡是你写for循环处理数据的统一全部用StreamAPI进行替换；  
Stream所有数据和操作被组合成流管道流管道组成：

- 一个数据源（可以是一个数组、集合、生成器函数、I/O管道）
- 零或多个中间操作（将一个流变形为另一个流）
- 一个终止操作（产生最终结果）
- 流是惰性的；只有在启动最终操作时才会对源数据进行计算，而且只在需要时才会消耗源元素；

## 中间操作: Intermediate Operations

函数	描述
filter	根据指定条件过滤元素。
map	使用提供的函数转换流中的每个元素。映射, a 变成 b
mapToInt mapToLong mapToDouble	使用提供的函数将每个元素转换为整数/长整数/双精度浮点数。
flatMap	将每个元素转换为元素流, 然后将这些流合并为一个流。
flatMapToInt flatMapToLong flatMapToDouble	将每个元素转换为整数/长整数/双精度浮点数流, 然后将这些流合并为一个流。
mapMulti mapMultiToInt mapMultiToLong mapMultiToDouble	使用提供的函数将每个元素转换为多个元素, 并将它们合并为一个流。
parallel	并行执行流操作。
unordered	禁用遇到顺序保证, 可以在某些情况下提高性能。
onClose	注册在流关闭时调用的关闭处理程序。
sequential	强制流以顺序方式执行。
distinct	从流中删除重复元素。
sorted	根据提供的比较器对流的元素进行排序。
peek	允许在不消耗元素的情况下对流中的每个元素执行指定操作。
limit	将流的大小限制为指定的最大元素数量。
skip	跳过流中指定数量的元素。
takeWhile	从流的开头获取元素, 直到指定条件为假。
dropWhile	从流的开头丢弃元素, 直到指定条件为假。

## 终止操作

函数	描述
forEach	对流中的每个元素执行指定的操作。
forEachOrdered	对流中的每个元素按照遇到顺序执行指定的操作。
toArray	将流中的元素收集到一个数组中。
reduce	将流中的元素按顺序组合成一个值。
collect	将流中的元素收集到一个集合或容器中，例如列表、映射等。
toList	将流中的元素收集到一个列表中。
min	找到流中的最小元素。
max	找到流中的最大元素。
count	统计流中的元素数量。
anyMatch	判断流中是否至少有一个元素满足指定条件。
allMatch	判断流中是否所有元素都满足指定条件。
noneMatch	判断流中是否没有元素满足指定条件。
findFirst	找到流中的第一个元素。
findAny	找到流中的任意一个元素。
iterator	返回一个迭代器，用于遍历流中的元素。

## Reactive-Stream

Reactive Streams是JVM面向流的库的标准和规范

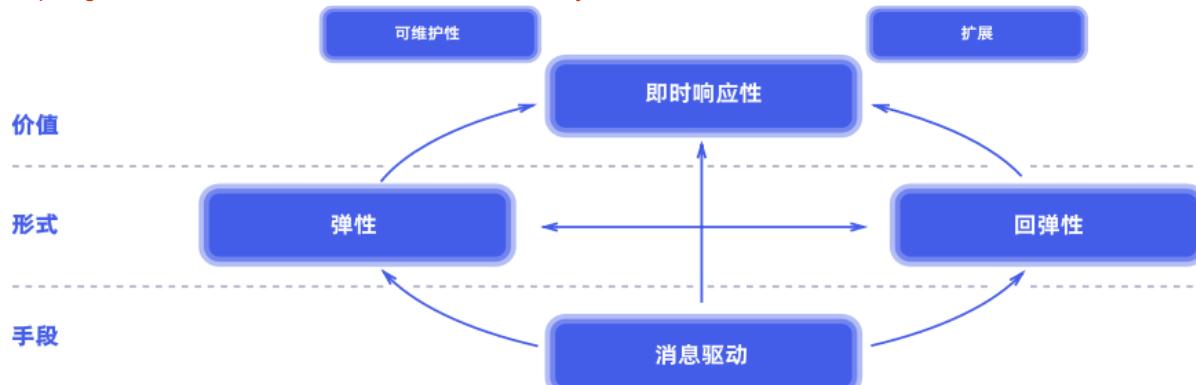
1. 处理可能无限数量的元素
2. 有序
3. 在组件之间异步传递元素
4. 强制性非阻塞，背压模式

响应式宣言：

<https://www.reactivemanifesto.org/zh-CNReactiveStream>

响应式官网：

<https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.4/README.md>



基于异步、消息驱动的全事件回调系统：响应式系统

API Components：

1. Publisher: 发布者；产生数据流

2. Subscriber: 订阅者； 消费数据流

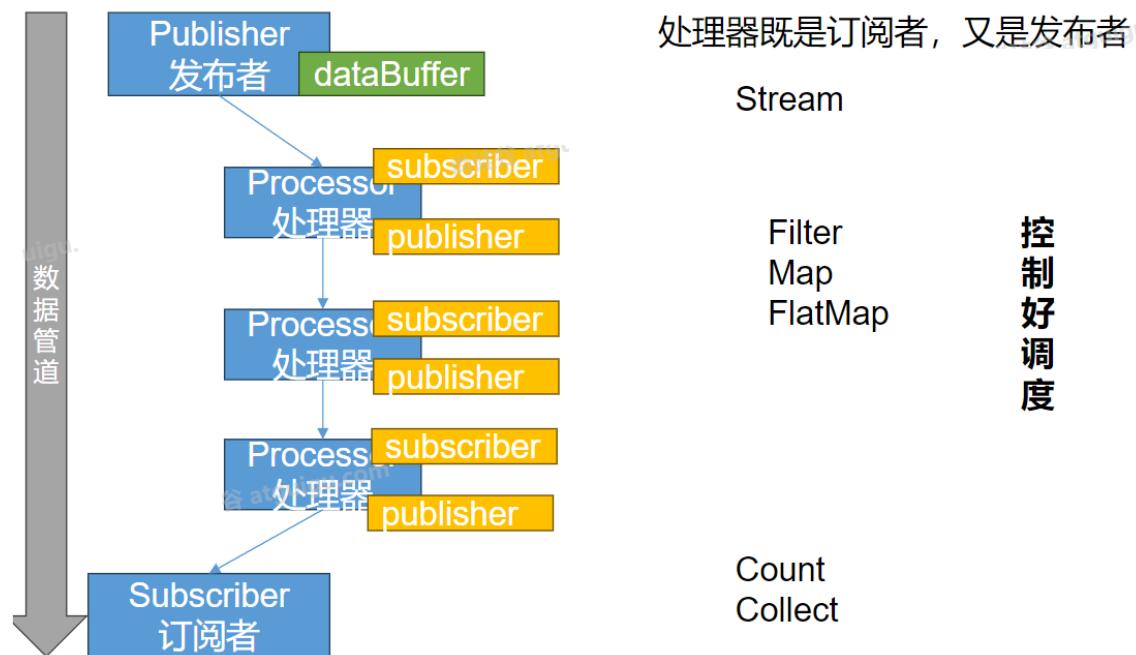
3. Subscription: 订阅关系；

- 订阅关系是发布者和订阅者之间的关键接口。订阅者通过订阅来表示对发布者产生的数据的兴趣。订阅者可以请求一定数量的元素，也可以取消订阅。

4. Processor: 处理器；

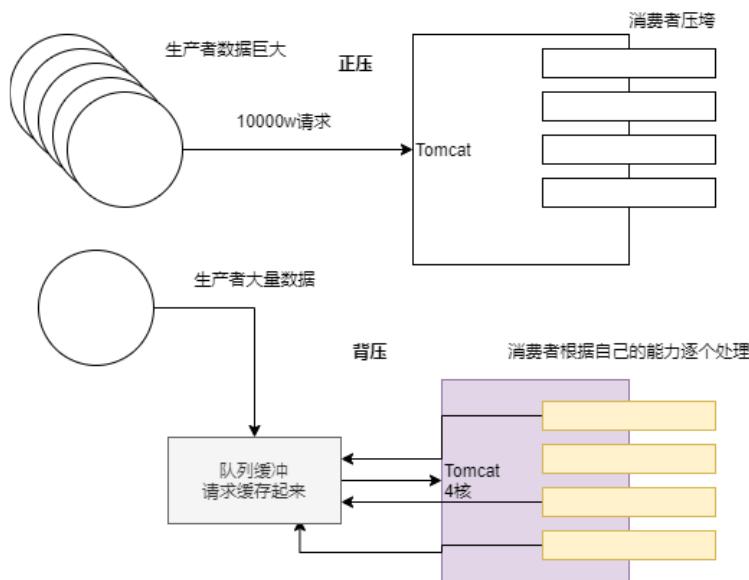
- 处理器是同时实现了发布者和订阅者接口的组件。它可以接收来自一个发布者的数据，进行处理，并将结果发布给下一个订阅者。处理器在Reactor中充当中间环节，代表一个处理阶段，允许你在数据流中进行转换、过滤和其他操作。

这种模型遵循Reactive Streams规范，确保了异步流的一致性和可靠性。



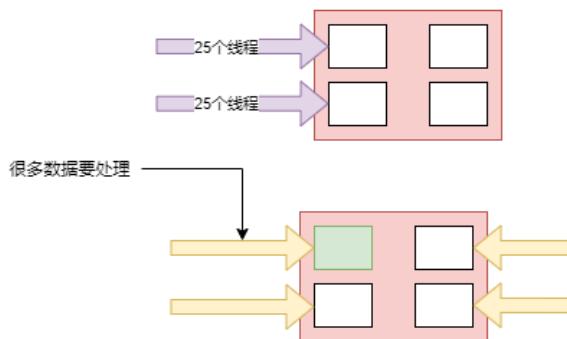
响应式编程：

- 1、底层：基于数据缓冲队列 + 消息驱动模型 + 异步回调机制(基于事件驱动)
- 2、编码：流式编程 + 链式调用 + 声明式API
- 3、效果：优雅全异步 + 消息实时处理 + 高吞吐量 + 占用少量资源



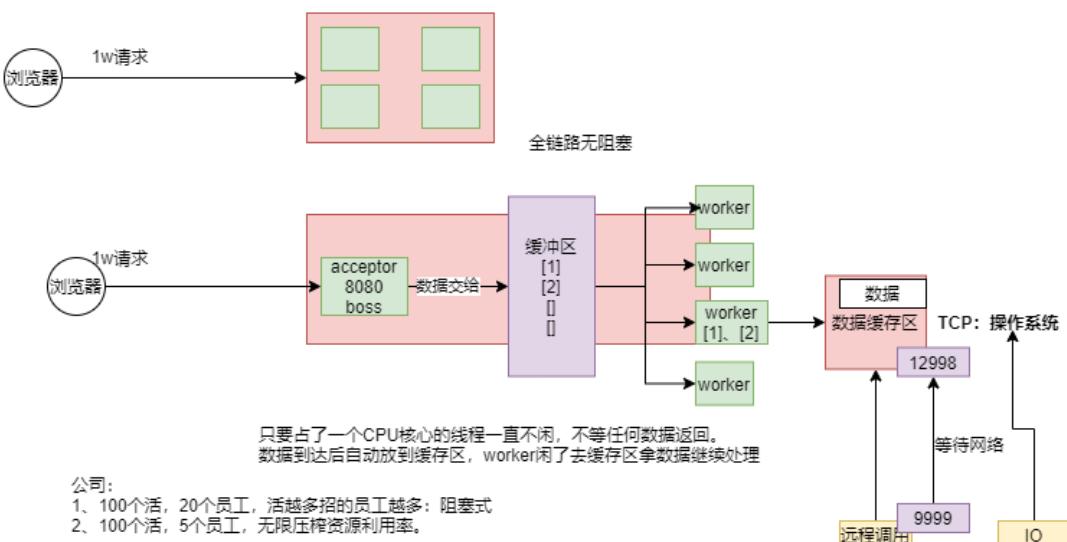
多线程：线程越多越好，还是越少越好？

100个线程：一个核心排队了很多线程，线程就要切换，切换保留现场（浪费内存，浪费时间）  
越多的线程只会产生激烈竞争



思路：让少量的线程一直忙，而不是让大量的线程一直切换等待

Tomcat：线程池200线程；开200个线程



目的：通过全异步的方式，加缓存区构建一个实时的数据流系统。  
Kafka、MQ能构建出大型分布式的响应式系统。  
缺本地化的消息系统解决方案：  
1）、让所有的异步线程能互相监听消息，处理消息，构建实时消息处理流

Reactive Stream





**痛点：**  
以前要做一个高并发系统：缓存、异步、队排好； 手动控制整个逻辑  
现在：全自动控制整个逻辑。只需要组装好数据处理流水线即可。

```

public class FlowDemo {

 // 定义流的中间操作处理器；只用写订阅者的接口
 static class MyProcessor extends SubmissionPublisher<String> implements
Flow.Processor<String, String> {
 private Flow.Subscription subscription;

 @Override
 // 当订阅开始时，此方法会被调用
 public void onSubscribe(Flow.Subscription subscription) {
 System.out.println("processor订阅绑定完成");
 this.subscription = subscription;
 subscription.request(1);
 }

 @Override // 当数据到达时，此回调会被触发
 public void onNext(String item) {
 System.out.println("processor接收到数据：" + item);
 item += "哈哈";
 submit(item); // 将数据发送出去
 // 处理完数据后,请求下一个数据
 subscription.request(1);
 }

 @Override // 当出现错误时，此方法会被调用
 public void onError(Throwable throwable) {

 }

 @Override // 当完成时，此方法会被调用
 public void onComplete() {

 }
 }

 public static void main(String[] args) throws InterruptedException {
 //1. 定义一个发布者
 SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

 //2. 定义多个中间操作，给每一个元素加一个哈哈前缀
 MyProcessor myProcessor = new MyProcessor();
 MyProcessor myProcessor1 = new MyProcessor();
 MyProcessor myProcessor2 = new MyProcessor();

 //3. 定义一个订阅者，订阅者感兴趣发布者的数据
 Flow.Subscriber<String> subscriber = new Flow.Subscriber<String>() {
 private Flow.Subscription subscription;

 @Override // 在订阅开始时，此方法会被调用
 public void onSubscribe(Flow.Subscription subscription) {

```

```

 System.out.println(Thread.currentThread() + " 订阅开始了," +
subscription);
 this.subscription = subscription;
 // 请求一个数据
 subscription.request(1);
 }

 @Override // 当下一个元素到达时，此方法会被调用
 public void onNext(String item) {
 System.out.println(Thread.currentThread() + " 收到数据：" + item);
 // 处理完数据后,请求下一个数据
 subscription.request(1);
 }

 @Override // 当出现错误时，此方法会被调用
 public void onError(Throwable throwable) {
 System.out.println("订阅出现异常：" + throwable);
 }

 @Override // 当完成时，此方法会被调用
 public void onComplete() {
 System.out.println("订阅完成了");
 }
};

// 绑定发布者和订阅者
publisher.subscribe(myProcessor); // 此时处理器相当于订阅者
myProcessor.subscribe(myProcessor1); // 此时处理器相当于订阅者
myProcessor1.subscribe(myProcessor2); // 此时处理器相当于订阅者
myProcessor2.subscribe(subscriber); // 此时处理器相当于发布者

for (int i = 0; i < 10; i++) {
 // 发布十条数据
 publisher.submit("p-" + i);
}

publisher.close();

Thread.sleep(20000);
}
}

```

## Reactor

Reactor是基于Reactive Streams的第四代响应式库规范，用于在JVM上构建非阻塞应用程序；<https://projectreactor.io>

1. 完全非阻塞的，并提供高效的需求管理。它直接与Java的功能API、CompletableFuture、Stream和Duration交互
2. Reactor提供了两个响应式和可组合的API，Flux[N]和Mono[0|1]
3. 适合微服务，提供基于netty背压机制的网络引擎(HTTP、TCP、UDP)

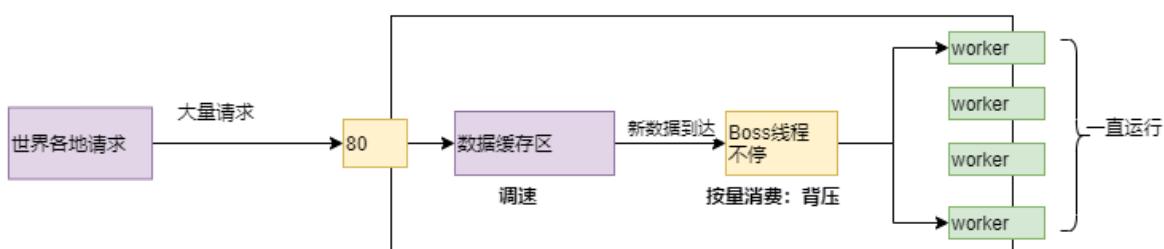
主要概念：

- 发布者 (Publisher)
- 订阅者 (Subscriber)
- 订阅关系 (Subscription)
- 处理器 (Processor)

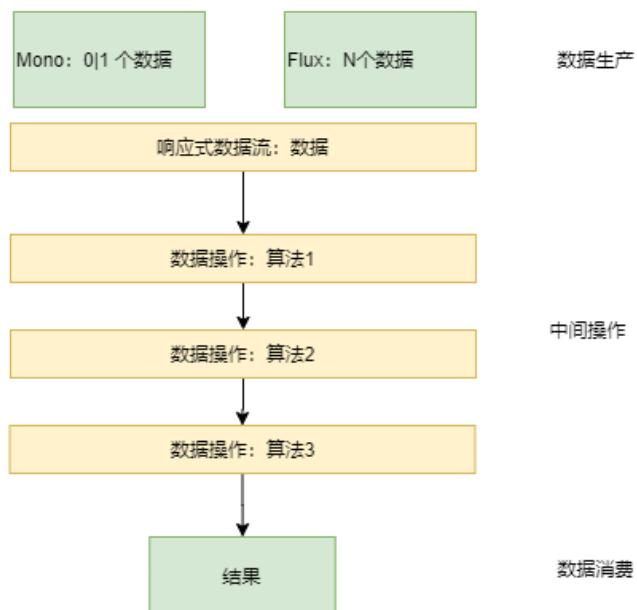
- 调度器 (Scheduler)
- 事件/信号 (event/signal)
- 序列/流 (sequence/stream)
- 元素 (element/item)
- 操作符 (operator)

## 非阻塞的原理：缓冲 + 回调

1. 开新线程不是解决问题的重点
2. 不要浪费时间去等待



少量线程一直运行 > 大量线程切换等待



1. 数据流： 数据源头
  2. 变化传播： 数据操作（中间操作）
  3. 异步编程模式： 底层控制异步
- 响应式编程

数据流：每个元素从流的源头，开始源源不断，自己往下滚动，  
onNext：当某个元素到达后，我们可以定义它的处理逻辑

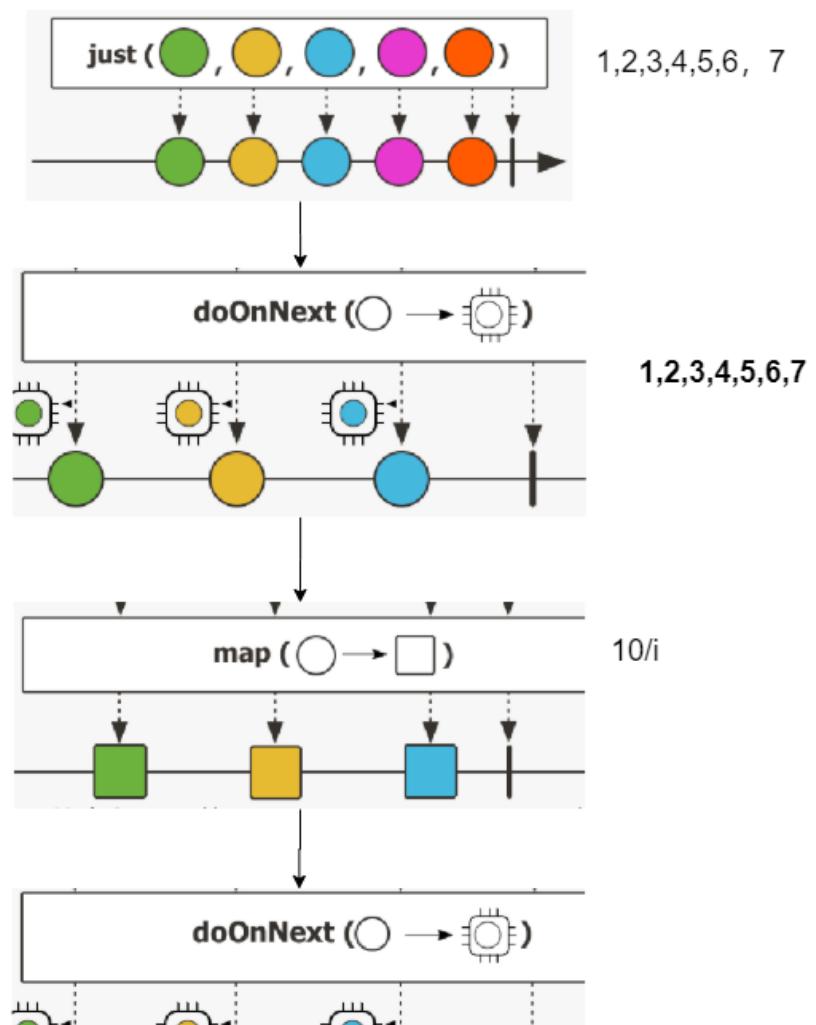
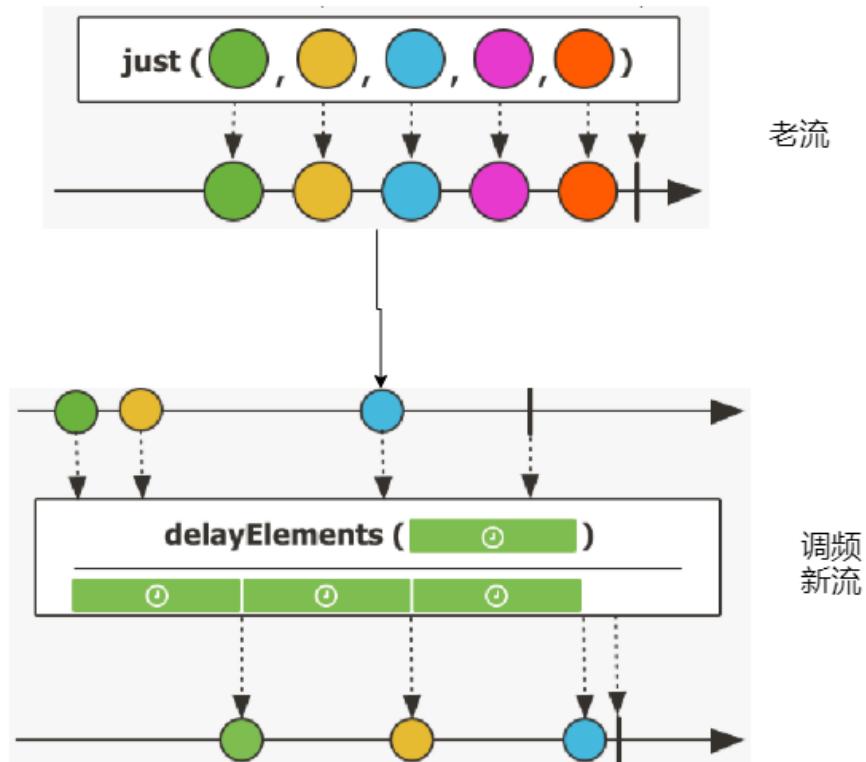
onComplete：处理结束；触发事件  
onError：异常结束；触发事件

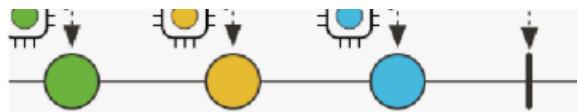
`onNext x 0..N [onError | onComplete]`

```
public void haha(List<Order> orders){
 //处理订单
 ...
}
```

```
//异常中断
//正常返回
}
```

流经过运算符计算后得到一个新流





10,5,3,2,2

## 快速上手

### 介绍

Reactor 是一个用于JVM的完全非阻塞的响应式编程框架，具备高效的需求管理（即对“背压（backpressure）”的控制）能力。它与 Java 8 函数式 API 直接集成，比如 `CompletableFuture`, `Stream`, 以及 `Duration`。它提供了异步序列 API `Flux`（用于[N]个元素）和 `Mono`（用于 [0|1]个元素），并完全遵循和实现了“响应式扩展规范”（Reactive Extensions Specification）。

Reactor 的 `reactor-ipc` 组件还支持非阻塞的进程间通信（inter-process communication, IPC）。Reactor IPC 为 HTTP（包括 Websockets）、TCP 和 UDP 提供了支持背压的网络引擎，从而适合应用于微服务架构。并且完整支持响应式编解码（reactive encoding and decoding）。

### 依赖

```
<dependencyManagement>
 <dependencies>
 <dependency>
 <groupId>io.projectreactor</groupId>
 <artifactId>reactor-bom</artifactId>
 <version>2023.0.0</version>
 <type>pom</type>
 <scope>import</scope>
 </dependency>
 </dependencies>
</dependencyManagement>
```

```
<dependencies>
 <dependency>
 <groupId>io.projectreactor</groupId>
 <artifactId>reactor-core</artifactId>
 </dependency>
 <dependency>
 <groupId>io.projectreactor</groupId>
 <artifactId>reactor-test</artifactId>
 <scope>test</scope>
 </dependency>
</dependencies>
```

## 响应式编程

响应式编程是一种关注于数据流（data streams）和变化传递（propagation of change）的异步编程方式。这意味着它可以用既有的编程语言表达静态（如数组）或动态（如事件源）的数据流。

了解历史：

- 在响应式编程方面，微软跨出了第一步，它在 .NET 生态中创建了响应式扩展库（Reactive Extensions library, Rx）。接着 RxJava 在 JVM 上实现了响应式编程。后来，在 JVM 平台出现了一套标准的响应式编程规范，它定义了一系列标准接口和交互规范。并整合到 Java 9 中（使用 Flow 类）。
- 响应式编程通常作为面向对象编程中的“观察者模式”（Observer design pattern）的一种扩展。响应式流（reactive streams）与“迭代子模式”（Iterator design pattern）也有相通之处，因为其中也有 Iterable-Iterator 这样的对应关系。主要的区别在于，Iterator 是基于“拉取”（pull）方式的，而响应式流是基于“推送”（push）方式的。

- 使用 iterator 是一种“命令式” (imperative) 编程范式，即使访问元素的方法是 Iterable 的唯一职责。关键在于，什么时候执行 next() 获取元素取决于开发者。在响应式流中，相对应的角色是 Publisher-Subscriber，但是 当有新的值到来的时候，却反过来由发布者 (Publisher) 通知订阅者 (Subscriber)，这种“推送”模式是响应式的关键。此外，对推送来的数据的操作是通过一种声明式 (declaratively) 而不是命令式 (imperatively) 的方式表达的：开发者通过描述“控制流程”来定义对数据流的处理逻辑。
- 除了数据推送，对错误处理 (error handling) 和完成 (completion) 信号的定义也很完善。一个 Publisher 可以推送新的值到它的 Subscriber (调用 onNext 方法)，同样也可以推送错误 (调用 onError 方法) 和完成 (调用 onComplete 方法) 信号。错误和完成信号都可以终止响应式流。可以用下边的表达式描述：

```
onNext x 0..N [onError | onComplete]
```

## 阻塞是对资源的浪费

现代应用需要应对大量的并发用户，而且即使现代硬件的处理能力飞速发展，软件性能仍然是关键因素。

广义来说我们有两种思路来提升程序性能：

1. **并行化 (parallelize)**：使用更多的线程和硬件资源。[异步]
2. 基于现有的资源来 提高执行效率。

通常，Java 开发者使用阻塞式 (blocking) 编写代码。这没有问题，在出现性能瓶颈后，我们可以增加处理线程，线程中同样是阻塞的代码。但是这种使用资源的方式会迅速面临 资源竞争和并发问题。

更糟糕的是，阻塞会浪费资源。具体来说，比如当一个程序面临延迟 (通常是 I/O 方面，比如数据库读写请求或网络调用)，所在线程需要进入 idle 状态等待数据，从而浪费资源。

所以，并行化方式并非银弹。这是挖掘硬件潜力的方式，但是却带来了复杂性，而且容易造成浪费。

## 异步可以解决问题吗

第二种思路——提高执行效率——可以解决资源浪费问题。通过编写 异步非阻塞 的代码，(任务发起异步调用后) 执行过程会切换到另一个 使用同样底层资源 的活跃任务，然后等 异步调用返回结果再去处理。

但是在 JVM 上如何编写异步代码呢？Java 提供了两种异步编程方式：

- **回调 (Callbacks)**：异步方法没有返回值，而是采用一个 callback 作为参数 (lambda 或匿名类)，当结果出来后回调这个 callback。常见的例子比如 Swings 的 EventListener。
- **Futures**：异步方法 立即 返回一个 Future，该异步方法要返回结果的是 T 类型，通过 Future 封装。这个结果并不是 立刻 可以拿到，而是等实际处理结束才可用。比如，ExecutorService 执行 Callable 任务时会返回 Future 对象。

这些技术够用吗？并非对于每个用例都是如此，两种方式都有局限性。

回调很难组合起来，因为很快就会导致代码难以理解和维护（即所谓的“回调地狱 (callback hell)”）。

考虑这样一种情景：

- 在用户界面上显示用户的5个收藏，或者如果没有任何收藏提供5个建议。
- 这需要3个服务（一个提供收藏的ID列表，第二个服务获取收藏内容，第三个提供建议内容）：

回调地狱 (Callback Hell) 的例子：

```
userService.getFavorites(userId, new Callback<List<String>>() {
 public void onSuccess(List<String> list) {
 if (list.isEmpty()) {
 suggestionService.getSuggestions(new Callback<List<Favorite>>() {
 public void onSuccess(List<Favorite> list) {
 uiutils.submitOnUiThread(() -> {
 list.stream()
 .limit(5)
 .forEach(uiList::show);
 });
 }
 });
 }
 }

 public void onError(Throwable error) {
 uiutils.errorPopup(error);
 }
}
```

```

 });
 } else {
 list.stream()
 .limit(5)
 .forEach(favId -> favoriteService.getDetails(favId,
 new Callback<Favorite>() {
 public void onSuccess(Favorite details) {
 uiUtils.submitOnUiThread(() -> uiList.show(details));
 }

 public void onError(Throwable error) {
 uiUtils.errorPopup(error);
 }
 }
));
 }
}

public void onError(Throwable error) {
 uiUtils.errorPopup(error);
}
);

```

Reactor改造后为：

```

userService.getFavorites(userId)
 .flatMap(favoriteService::getDetails)
 .switchIfEmpty(suggestionService.getSuggestions())
 .take(5)
 .publishOn(uiUtils.uiThreadScheduler())
 .subscribe(uiList::show, uiUtils::errorPopup);

```

如果你想确保“收藏的ID”的数据在800ms内获得（如果超时，从缓存中获取）呢？在基于回调的代码中，会比较复杂。但 Reactor 中就很简单，在处理链中增加一个 timeout 的操作符即可。

```

userService.getFavorites(userId)
 .timeout(Duration.ofMillis(800))
 .onErrorResume(cacheService.cachedFavoritesFor(userId))
 .flatMap(favoriteService::getDetails)
 .switchIfEmpty(suggestionService.getSuggestions())
 .take(5)
 .publishOn(uiUtils.uiThreadScheduler())
 .subscribe(uiList::show, uiUtils::errorPopup);

```

额外扩展：

Futures 比回调要好一点，但即使在 Java 8 引入了 `CompletableFuture`，它对于多个处理的组合仍不够好用。编排多个 Futures 是可行的，但却不易。此外，`Future` 还有一个问题：当对 `Future` 对象最终调用 `get()` 方法时，仍然会导致阻塞，并且缺乏对多个值以及更进一步对错误的处理。

考虑另外一个例子，我们首先得到 ID 的列表，然后通过它进一步获取到“对应的 name 和 statistics”为元素的列表，整个过程用异步方式来实现。

`CompletableFuture` 处理组合的例子

```

CompletableFuture<List<String>> result = ids.thenComposeAsync(l -> {
 Stream<CompletableFuture<String>> zip =
 l.stream()
 .map(i -> {
 CompletableFuture<String> nameTask = ifhName(i);
 CompletableFuture<Integer> statTask = ifhStat(i);

```

```

 return nameTask.thenCombineAsync(statTask, (name, stat) ->
 "Name " + name + " has stats " + stat);
 });
 List<CompletableFuture<String>> combinationList =
zip.collect(Collectors.toList());
 CompletableFuture<String>[] combinationArray = combinationList.toArray(new
CompletableFuture[combinationList.size()]);
}

CompletableFuture<Void> allDone =
CompletableFuture.allOf(combinationArray);
return allDone.thenApply(v -> {
 combinationList.stream()
 .map(CompletableFuture::join)
 .collect(Collectors.toList())
});
};

List<String> results = result.join();
assertThat(results).contains(
 "Name NameJoe has stats 103",
 "Name NameBart has stats 104",
 "Name NameHenry has stats 105",
 "Name NameNicole has stats 106",
 "Name NameABSLAJINFOAJINFOANFANSF has stats 121");

```

## 从命令式编程到响应式编程

类似 Reactor 这样的响应式库的目标就是要弥补上述“经典”的 JVM 异步方式所带来的不足，此外还会关注以下几个方面：

- 可编排性 (Composability) 以及 可读性 (Readability)
- 使用丰富的 操作符 来处理形如 流 的数据
- 在 订阅 (subscribe) 之前什么都不会发生
- 背压 (backpressure) 具体来说即 消费者能够反向告知生产者生产内容的速度的能力
- 高层次 (同时也是有高价值的) 的抽象，从而达到 并发无关 的效果

## 可编排性与可读性

可编排性，指的是编排多个异步任务的能力。比如我们将前一个任务的结果传递给后一个任务作为输入，或者将多个任务以分解再汇总 (fork-join) 的形式执行，或者将异步的任务作为离散的组件在系统中进行重用。

这种编排任务的能力与代码的可读性和可维护性是紧密相关的。随着异步处理任务数量和复杂度 的提高，编写和阅读代码都变得越来越困难。就像我们刚才看到的，回调模式是简单的，但是缺点是在复杂的处理逻辑中，回调中会层层嵌入回调，导致 回调地狱 (Callback Hell) 。你能猜到 (或有过这种痛苦经历)，这样的代码是难以阅读和分析的。

Reactor 提供了丰富的编排操作，从而代码直观反映了处理流程，并且所有的操作保持在同一层次 (尽量避免了嵌套) 。

## 就像装配流水线

你可以想象数据在响应式应用中的处理，就像流过一条装配流水线。Reactor 既是传送带，又是一个个的装配工或机器人。原材料从源头 (最初的 Publisher) 流出，最终被加工为成品，等待被推送到消费者 (或者说 Subscriber) 。

原材料会经过不同的中间处理过程，或者作为半成品与其他半成品进行组装。如果某处有齿轮卡住，或者某件产品的包装过程花费了太久时间，相应的工位就可以向上游发出信号来限制或停止发出原材料。

## 操作符 (Operators)

在 Reactor 中，操作符 (operator) 就像装配线中的工位（操作员或装配机器人）。**每一个操作符 对 Publisher 进行相应的处理，然后将 Publisher 包装为一个新的 Publisher。**就像一个链条，数据源自第一个 Publisher，然后顺链条而下，在每个环节进行相应的处理。最终，一个订阅者 (Subscriber) 终结这个过程。请记住，在订阅者 (Subscriber) 订阅 (subscribe) 到一个发布者 (Publisher) 之前，什么都不会发生。

理解了操作符会创建新的 Publisher 实例这一点，能够帮助你避免一个常见的问题，这种问题会让你觉得处理链上的某个操作符没有起作用。

虽然响应式流规范 (Reactive Streams specification) 没有规定任何操作符，类似 Reactor 这样的响应式库所带来的最大附加价值之一就是提供丰富的操作符。包括基础的转换操作，到过滤操作，甚至复杂的编排和错误处理操作。

## subscribe() 之前什么都不会发生

在 Reactor 中，当你创建了一条 Publisher 处理链，数据还不会开始生成。事实上，你是创建了一种抽象的对于异步处理流程的描述（从而方便重用和组装）。

当真正“订阅 (subscribe)”的时候，你需要将 Publisher 关联到一个 Subscriber 上，然后才会触发整个链的流动。这时候，Subscriber 会向上游发送一个 request 信号，一直到达源头的 Publisher。

## 背压

向上游传递信号这一点也被用于实现 **背压**，就像在装配线上，某个工位的处理速度如果慢于流水线速度，会对上游发送反馈信号一样。

在响应式流规范中实际定义的机制同刚才的类比非常接近：订阅者可以无限接受数据并让它的源头“满负荷”推送所有的数据，也可以通过使用 request 机制来告知源头它一次最多能够处理 n 个元素。

中间环节的操作也可以影响 request。想象一个能够将每10个元素分批打包的缓存 (buffer) 操作。如果订阅者请求一个元素，那么对于源头来说可以生成10个元素。此外预取策略也可以使用了，比如在订阅前预先生成元素。这样能够将“推送”模式转换为“推送+拉取”混合的模式，如果下游准备好了，可以从上游拉取 n 个元素；但是如果上游元素还没有准备好，下游还是要等待上游的推送。

## 热 (Hot) vs 冷 (Cold)

在 Rx 家族的响应式库中，响应式流分为“热”和“冷”两种类型，区别主要在于响应式流如何对订阅者进行响应：

- 一个“冷”的序列，指对于每一个 Subscriber，都会收到从头开始所有的数据。如果源头生成了一个 HTTP 请求，对于每一个订阅都会创建一个新的 HTTP 请求。
- 一个“热”的序列，指对于一个 Subscriber，只能获取从它开始订阅之后发出的数据。不过注意，有些“热”的响应式流可以缓存部分或全部历史数据。通常意义上来说，一个“热”的响应式流，甚至在即使没有订阅者接收数据的情况下，也可以发出数据（这一点同“Subscribe() 之前什么都不会发生”的规则有冲突）。

## 核心特性

### Mono 和 Flux

Mono: 0|1 数据流

Flux: N 数据流

响应式流: 元素(内容) + 信号(完成/异常)

```
public static void fluxMono() {
 Mono<Integer> mono = Mono.just(1).log();
 mono.subscribe();

 Flux.range(1, 7)
 .filter(i -> i > 3)
 .log()
 .map(i -> "hha" + i)
 //.log()
 .subscribe(System.out::println);
}
```

## subscribe

自定义流的信号感知回调

```
flux.subscribe(
 //流元素消费
 v -> System.out.println("v = " + v),
 //感知异常结束
 throwable -> System.out.println("throwable = " + throwable),
 //感知正常结束
 () -> System.out.println("流结束了...")
);
```

自定义消费者 BaseSubscriber

```
flux.subscribe(new BaseSubscriber<String>() {
 // 生命周期钩子1： 订阅关系绑定的时候触发
 @Override
 protected void hookOnSubscribe(Subscription subscription) {
 // 流被订阅的时候触发
 System.out.println("绑定了..." + subscription);

 // 找发布者要数据
 request(1); // 要1个数据
 // requestUnbounded(); // 要无限数据
 }

 @Override
 protected void hookOnNext(String value) {
 System.out.println("数据到达，正在处理：" + value);
 request(1); // 要1个数据
 }

 // hookOnComplete、hookOnError 二选一执行
 @Override
 protected void hookOnComplete() {
 System.out.println("流正常结束...");
 }

 @Override
 protected void hookOnError(Throwable throwable) {
 System.out.println("流异常..." + throwable);
 }
});
```

```

protected void hookOnCancel() {
 System.out.println("流被取消...");
}

@Override
protected void hookFinally(signalType type) {
 System.out.println("最终回调...一定会被执行");
}
);

```

## 流的取消

消费者调用cancel()取消流的订阅  
Disposable

```

Flux<String> flux = Flux.range(1, 10)
 .map(i -> {
 System.out.println("map..." + i);
 if(i==9) {
 i = 10/(9-i); //数学运算异常; doOnXXX
 }
 return "哈哈: " + i;
 });
// 流错误的时候，把错误吃掉，转为正常信号
// //流被订阅；默认订阅;
// flux.subscribe();
// //指定订阅规则： 正常消费者：只消费正常元素
// flux.subscribe(v-> System.out.println("v = " + v));

// flux.subscribe(
// //流元素消费
// v-> System.out.println("v = " + v),
// //感知异常结束
// throwable -> System.out.println("throwable = " + throwable),
// //感知正常结束
// ()-> System.out.println("流结束了...")
//);

// 流的生命周期钩子可以传播给订阅者。
// a() {
// data = b();
// }
flux.subscribe(new BaseSubscriber<String>() {
 // 生命周期钩子1： 订阅关系绑定的时候触发
 @Override
 protected void hookOnSubscribe(Subscription subscription) {
 // 流被订阅的时候触发
 System.out.println("绑定了..." + subscription);

 //找发布者要数据
 request(1); //要1个数据
 //requestUnbounded(); //要无限数据
 }

 @Override
 protected void hookOnNext(String value) {
 System.out.println("数据到达，正在处理: "+value);
 if(value.equals("哈哈: 5")){
 cancel(); //取消流
 }
 }
}

```

```

 request(1); //要1个数据
 }

 // hookOnComplete、hookOnError 二选一执行
 @Override
 protected void hookOnComplete() {
 System.out.println("流正常结束...");
 }

 @Override
 protected void hookOnError(Throwable throwable) {
 System.out.println("流异常..."+throwable);
 }

 @Override
 protected void hookOnCancel() {
 System.out.println("流被取消...");
 }

 @Override
 protected void hookFinally(SignalType type) {
 System.out.println("最终回调...一定会被执行");
 }
);

```

## 背压和请求重塑

### 背压(Backpressure)和请求重塑(Reshape Requests)

1. buffer: 缓冲,固定元素结束
2. bufferUntilChanged(): 给定条件结束,如果下一个判定值比起上一个发生了变化就开一个新的buffer保存

```

Flux<List<Integer>> flux = Flux.range(1, 10) //原始流10个
 //缓冲区: 缓冲3个元素: 消费一次最多可以拿到三个元素; 凑满数批量发给消费者
 .buffer(3)
 .log();
// 一次发一个, 一个一个发;
// 10元素, buffer(3); 消费者请求4次, 数据消费完成

```

2. limit: 限流

```

Flux.range(1, 1000)
 .log()
 //限流触发, 看上游是怎么限流获取数据的
 //一次预取100个元素; 第一次 request(100), 以后request(75)
 .limitRate(100)
 .subscribe();
// 75% 预取策略, limitRate(100)
// 第一次抓取100个数据, 如果75%的元素已经处理了,继续抓取新的75%的元素

```

## 以编程方式创建序列-Sink

Sink.next  
Sink.complete

## 同步环境

```
/**
 * Sink: 接收器、水槽、通道;
 * Source: 数据源、Sink、接受端
 */
// generate 方法用于生成一个数据流
public void generate() {
 // 使用 Flux.generate() 方法生成数据流
 Flux<Object> flux = Flux.generate(
 // 提供初始状态的函数
 () -> 0,
 // 生成元素的函数, 接收当前状态和 FluxSink 对象
 (state, sink) -> {
 // 向数据流中发送元素, 这里以状态值为例
 sink.next("值:" + state);
 // 如果状态值为 7, 则发送一个异常信号
 if (state == 7) {
 sink.error(new RuntimeException("我不喜欢7"));
 }
 // 如果状态值为 10, 则发送完成信号
 if (state == 10) {
 sink.complete();
 }
 // 返回新的状态值
 return state + 1;
 });

 // 订阅数据流, 并打印日志
 flux.log()
 // 在遇到错误时执行操作
 .doOnError(throwable -> {
 System.out.println("throwable:" + throwable);
 })
 // 订阅数据流
 .subscribe();
}
```

## 异步环境

```
// 定义一个 create 方法
public void create() {
 // 使用 Flux.create() 创建一个 Flux 数据流
 Flux.create(fluxSink -> {
 // 创建一个 MyListener 实例, 并传入 FluxSink 对象
 MyListener listener = new MyListener(fluxSink);
 // 模拟100个用户登录事件
 for (int i = 0; i < 100; i++) {
 // 调用 MyListener 中的 online 方法, 触发用户登录事件, 并传入用户名
 listener.online("张" + i);
 }
 })
 // 在数据流中增加日志
 .log()
 // 订阅数据流
 .subscribe();
}
// 自定义的监听器类 MyListener
class MyListener {
```

```

// FluxSink 对象，用于手动控制数据流
reactor.core.publisher.FluxSink<Object> sink;

// 构造方法，接受一个 FluxSink 对象
public MyListener(reactor.core.publisher.FluxSink<Object> sink) {
 this.sink = sink;
}

// 用户登录方法，触发用户登录事件，并将用户名传入数据流中
public void online(Object username) {
 // 打印用户登录信息
 System.out.println("用户登录了：" + username);
 // 将用户名传入数据流中
 sink.next(username);
}

```

## handle()

它是一个实例方法，意思是 它链接在现有源上（与常见运算符一样）。它存在 在 Mono 和 Flux。它接近于 generate，因为它使用 SynchronousSink 和只允许一个接一个的排放。但是， handle 可以用来生成一个每个源元素的任意值，可能跳过一些元素。

```

Flux.range(1, 10).
 handle((value, sink) -> {
 System.out.println("拿到的值：" + value);
 sink.next("张三-" + value);
 })
 .log()
 .subscribe();

```

## 自定义线程调度

响应式:响应式编程:全异步、消息、事件回调  
默认还是用当前线程，生成整个流、发布流、流操作  
Schedulers

```

public void thread() {
 Flux.range(1, 10)
 // 在那个线程池把这个流的数据和操作执行了
 // 改变发布者的线程
 .publishOn(Schedulers.boundedElastic())
 .log()
 // 改变订阅者的线程
 //.subscribeOn(Schedulers.single())
 .subscribe();

 // publishOn : 改变发布者线程
 // subscribeOn : 改变订阅者线程

 // 调度器: 线程池
 // 无执行上下文，当前线程运行所有操作
 Schedulers.immediate();
 // 选择固定的一个单线程
 Schedulers.single();
 // 有界，弹性调度，不是无线扩充的线程池
}

```

```
// 线程池中有10*CPU核心个线程; 队列默认100K, keepAliveTime: 60s
Schedulers.boundedElastic();
// 并发调度器
Schedulers.parallel();
// 自定义的调度器
// 参数:
// corePoolSize, 即使线程处于空闲状态, 线程池中保持的线程数量, 除非设置了
allowCoreThreadTimeOut
 // maximumPoolSize, 线程池允许的最大线程数量
 // keepAliveTime, 当线程数量大于核心时, 这是多余的空闲线程等待新任务之前终止的最大时间。
 // TimeUnit, 时间单位
 // capacity, 此队列将仅保存由execute方法提交的Runnable任务
 Schedulers.fromExecutor(new ThreadPoolExecutor(4, 8, 60, TimeUnit.SECONDS,
new LinkedBlockingQueue<>(1000)));
}
```

```
public void thread1(){
 Scheduler s = Schedulers.newParallel("parallel-scheduler", 4);

 final Flux<String> flux = Flux
 .range(1, 2)
 .map(i -> 10 + i)
 .log()
 .publishOn(s)
 .map(i -> "value " + i);

 //只要不指定线程池, 默认发布者用的线程就是订阅者的线程;
 new Thread(() -> flux.subscribe(System.out::println)).start();
}
```

## 错误处理

命令式编程: 常见的错误处理方式

1. Catch and return a static default value. **捕获异常返回一个静态默认值**

```
try {
 return doSomethingDangerous(10);
}
catch (Throwable error) {
 return "RECOVERED";
}
```

响应式编程实现  
onErrorReturn

1. 吃掉异常, 消费者无异常感知
2. 返回一个默认值
3. 流正常完成

```
Flux.just(10)
 .map(this::doSomethingDangerous)
 .onErrorReturn("RECOVERED");
```

2. Catch and execute an alternative path with a fallback method. **吃掉异常, 执行一个兜底方法**

```
List<Integer> numbers = List.of(1, 2, 0, 4);
List<String> results = new ArrayList<>();
```

```

for (int i : numbers) {
 try {
 results.add("100 / " + i + " = " + (100 / i));
 } catch (ArithmetricException e) {
 results.add("哈哈哈-error");
 }
}

for (String result : results) {
 System.out.println("v = " + result);
}

```

### OnErrorResume

1. 吃掉异常，消费者无异常感知
2. 调用一个兜底方法
3. 流正常完成

```

Flux.just(1, 2, 0, 4)
 .map(i -> "100 / " + i + " = " + (100 / i))
 // 调用一个兜底方法
 .onErrorResume(err -> Mono.just("哈哈哈-error"))
 .subscribe(v -> System.out.println("v = " + v),
 err -> System.out.println("err = " + err),
 () -> System.out.println("流结束了"));

```

### 3. Catch and dynamically compute a fallback value. 捕获并动态计算一个返回值

根据错误返回一个新值，进行动态计算

```

void test() {
 List<Integer> numbers = List.of(1, 2, 0, 4);
 List<String> results = new ArrayList<>();

 for (int i : numbers) {
 try {
 results.add("100 / " + i + " = " + (100 / i));
 } catch (ArithmetricException e) {
 results.add(error(e).block()); // 阻塞获取 Mono 的结果
 }
 }

 for (String result : results) {
 System.out.println("v = " + result);
 }
}

String error(Throwable throwable) {
 if (throwable instanceof NullPointerException) {
 // 处理 NullPointerException 的情况
 // 比如: return Mono.just("特殊错误处理");
 }
 return "error-" + throwable.getMessage();
}

```

### OnErrorResume

1. 吃掉异常，消费者有感知

2. 调用一个自定义方法

3. 流异常完成

```
void test() {
 Flux.just(1, 2, 0, 4)
 .map(i -> "100 / " + i + " = " + (100 / i))
 // 调用函数进行动态函数
 .onErrorResume(err -> error(err))
 .subscribe(v -> System.out.println("v = " + v),
 err -> System.out.println("err = " + err),
 () -> System.out.println("流结束了"));
}

Mono<String> error(Throwable throwable) {
 if (throwable.getClass() == NullPointerException.class) {
 // 处理 NullPointerException 的情况
 // 比如: return Mono.just("特殊错误处理");
 }
 return Mono.just("error-" + throwable.getMessage());
}
```

4. Catch, wrap to a `BusinessException`, and re-throw. **捕获并包装成一个业务异常，并重新抛出**

```
try {
 return callExternalService(k);
}
catch (Throwable error) {
 throw new BusinessException("oops, SLA exceeded", error);
}
```

包装重新抛出异常

1. 吃掉异常，消费者有感知

2. 抛新异常

3. 流异常完成

```
// .onErrorResume(err -> Flux.error(new BusinessException()))
// .onErrorMap(err -> new BusinessException()) 推荐
static void test() {
 Flux.just(1, 2, 0, 4)
 // 对每个元素执行除法运算，并将结果映射为字符串
 .map(i -> "100 / " + i + " = " + (100 / i))
 // 如果发生异常，将异常转换为 BusinessException
 .onErrorMap(err -> new BusinessException(err.getMessage() + "pong!"))
 // 如果希望在异常发生时返回另一个 Flux，则可以使用 onErrorResume 操作符
 // .onErrorResume(err -> Flux.error(new
BusinessException(err.getMessage() + "pong!")))
 // 订阅数据流，并分别处理正常元素、异常和流结束事件
 .subscribe(
 // 处理正常的数据项
 v -> System.out.println("v = " + v),
 // 处理异常情况
 err -> System.out.println("err = " + err),
 // 处理流结束事件
 () -> System.out.println("流结束了")
);
}

// 自定义的 BusinessException 类，继承自 RuntimeException
```

```
class BusinessException extends RuntimeException {
 public BusinessException(String message) {
 super(message);
 }
}
```

5. Catch, log an error-specific message, and re-throw.

捕获异常，记录特殊的错误日志，重新抛出

```
try {
 return callExternalService(k);
}
catch (RuntimeException error) {
 //make a record of the error
 log("uh oh, falling back, service failed for key " + k);
 throw error;
}
```

1. 异常被捕获，做自己的事情
2. 不影响异常继续顺着流水线传播
3. 不吃掉异常，只在异常发生的时候做一件事，消费者有感知

```
public static void main(String[] args) {
 // 创建一个 LongAdder 对象用于记录失败的次数
 LongAdder failureStat = new LongAdder();

 // 创建一个 Flux 数据流，包含一个字符串元素 "unknown"
 Flux<String> flux = Flux.just("unknown")
 // 对每个元素调用外部服务，并在出现错误时执行回退操作
 .flatMap(k -> callExternalService(k)
 .doOnError(e -> {
 // 如果调用外部服务发生错误，则增加 failurestat 的计数
 failureStat.increment();
 // 记录错误日志
 log("uh oh, falling back, service failed for key " + k);
 })
);
 }

 // 模拟调用外部服务的方法
 private static Flux<String> callExternalService(String key) {
 // 在这里进行实际的外部服务调用
 // 这里简单地返回一个包含错误信息的 Flux
 return Flux.error(new RuntimeException("External service failed for key: "
+ key));
 }

 // 模拟记录日志的方法
 private static void log(String message) {
 // 在这里执行日志记录操作
 System.out.println(message);
 }
}
```

6. Use the **finally** block to clean up resources or a Java 7 “try-with-resource” construct.

```

Stats stats = new Stats();
stats.startTimer();
try {
 doSomethingDangerous();
}
finally {
 stats.stopTimerAndRecordTiming();
}

```

```

// 创建一个 Stats 实例用于记录统计信息
Stats stats = new Stats();
// 创建一个 LongAdder 实例用于记录取消操作的次数
LongAdder statsCancel = new LongAdder();

// 创建一个 Flux 数据流，包含字符串 "foo" 和 "bar"
Flux<String> flux = Flux.just("foo", "bar")
 // 当订阅开始时，启动计时器
 .doOnSubscribe(s -> stats.startTimer())
 // 当数据流结束时，停止计时器并记录时间，并根据结束类型执行相应的操作
 .doFinally(type -> {
 stats.stopTimerAndRecordTiming();
 if (type == SignalType.CANCEL)
 // 如果结束类型是取消，则增加 statsCancel 的计数
 statsCancel.increment();
 })
 // 仅取第一个元素
 .take(1);

```

7 忽略当前异常，仅通知记录，继续推进

```

Flux.just(1, 2, 0, 4)
 // 对每个元素执行除法运算并构造字符串
 .map(i -> "100 / " + i + " = " + (100 / i))
 // 当出现异常时继续执行，并在控制台打印异常信息和当前元素值
 .onErrorContinue((err, val) -> {
 System.out.print("err = " + err);
 System.out.println(", val = " + val);
 System.out.println("发现" + val + "有问题了，继续执行其他的，我会记录这个问题！");
 })
 // 订阅数据流，并分别处理数据项和异常情况
 .subscribe(
 // 处理正常的数据项
 v -> System.out.println("v = " + v),
 // 处理异常情况
 err -> System.out.println("err = " + err)
);

```

## 常用操作

`filter`、`flatMap`、`concatMap`、`flatMapMany`、`transform`、`defaultIfEmpty`、`switchIfEmpty`、`concat`、`concatWith`、`merge`、`mergeWith`、`mergeSequential`、`zip`、`zipWith`...

- 常用操作
- 错误处理
- 超时与重试

- Flux.just(1)
 

```
.delayElements(Duration.ofSeconds(3))
.log()
// 超时，设置超过1秒就超时
.timeout(Duration.ofSeconds(1))
// 重试3次，把流从头到尾重新请求一次
.retry(3)
.map(i -> i + "haha")
.log()
.subscribe();
```

- **Sinks工具类**

- **单播**
- **多播**
- **重放**
- **背压**

- // Sinks: 接收器，数据管道，所有数据顺着这个管道往下走的
 

```
// 单播：只能绑定单个订阅者（消费者）
Sinks.many().unicast();
// 多播：可以绑定多个订阅者（消费者）
Sinks.many().multicast();
// 重放：可以绑定多个订阅者（消费者），并且可以重放历史数据，是否给后来的订阅者把之前的元素依然发给它
Sinks.many().replay();

// 单播/多播
/**
Sinks.Many<Object> many = sinks.many()
 // .unicast()
 .multicast()
 .onBackpressureBuffer();
*/
```
- // 创建一个 sinks.Many，可以发布多个数据项给订阅者
 

```
sinks.Many<Object> many = sinks.many()
 // 使用 replay() 方法，让发布者数据重放
 .replay()
 // 限制重放的数据项数量为3
 .limit(3);

// 创建一个新线程，向 sinks.Many 发布数据
new Thread(() -> {
 for (int i = 0; i < 10; i++) {
 // 尝试发布数据项
 many.tryEmitNext("a-" + i);
 try {
 // 线程休眠1秒
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 throw new RuntimeException(e);
 }
 }
}).start();

// 创建第一个订阅者，订阅 sinks.Many 的数据流
many.asFlux()
 .subscribe(v -> System.out.println("接收者1: " + v));

// 创建第二个订阅者，稍后再订阅 sinks.Many 的数据流
```

```

new Thread() -> {
 try {
 // 延迟5秒后订阅数据流
 Thread.sleep(5000);
 } catch (InterruptedException e) {
 throw new RuntimeException(e);
 }
 // 订阅 Sinks.Many 的数据流，并打印接收到的数据项
 many.asFlux()
 .subscribe(v -> System.out.println("接收者2: " + v));
}).start();

```

- 缓存

```

// 创建一个包含整数的数据流，从1开始，包含10个元素，每个元素之间延迟1秒
Flux<Integer> cache = Flux.range(1, 10)
 .delayElements(Duration.ofSeconds(1))
 // 不调缓存，就是缓存所有元素
 // 在此处调用缓存方法，以缓存数据流中的最近3个元素s
 .cache(3);
cache.subscribe();

new Thread() -> {
 try {
 // 延迟5秒，再看缓存元素
 Thread.sleep(5000);
 } catch (InterruptedException e) {
 throw new RuntimeException(e);
 }
 // 5秒后再次订阅数据流，打印缓存的元素
 cache.subscribe(v -> System.out.println("v: " + v));
}).start();
// 等待用户输入，以保持主线程运行
System.in.read();

```

- 阻塞式API

- block

```

List<Integer> block = Flux.just(1, 2, 4)
 .map(i -> i + 10)
 .collectList()
 // 也是一种订阅方式，BlockingMonoSubscriber
 .block();
System.out.println(block);

```

- Context-API：响应式中的ThreadLocal

- ThreadLocal机制失效

```

// ThreadLocal在响应式编程中不能用
// 响应式中，数据流期间共享数据，Context API: Context: 读写，ContextView: 只读

// 支持Context的中间操作
Flux.just(1, 2, 3)
 .transformDeferredContextual((flux, context) -> {
 System.out.println("val = " + flux);
 System.out.println("context = " + context);
 return flux.map(i -> i + "==" + context.get("prefix"));
})
// 上游能拿到下游的最近一次数据
// ThreadLocal共享了数据，上游的所有人能看到
// Context由下游传播给上游
 .contextWrite(Context.of("prefix", "哈哈"))

```

```

 .subscribe(v -> System.out.println("v = " + v));

```

- // 在 Reactor 中执行 PUT 请求并利用上下文传递数据的示例

```

// 定义 HTTP_CORRELATION_ID 常量作为上下文中的键
final String HTTP_CORRELATION_ID = "reactive.http.library.correlationId";

// 执行 PUT 请求的方法
Mono<Tuple2<Integer, Object>> doPut(String url, Mono<String> data) {
 // 将数据流和上下文项组合在一起
 Mono<Tuple2<String, Optional<Object>>> dataContext =
 data.zipWith(Mono.deferContextual(
 ctx -> Mono.just(ctx.getOrEmpty(HTTP_CORRELATION_ID))
));

 // 处理数据和上下文项，并返回结果状态码和消息的元组
 return dataContext.handle((val, sink) -> {
 if (val.getT2().isPresent()) {
 // 如果上下文项中包含了 HTTP_CORRELATION_ID 的值，则添加相应的头部信息
 sink.next("PUT <" + val.getT1() + "> sent to " + url + " with
header X-Correlation-ID=" + val.getT2().get());
 } else {
 // 如果上下文项中不包含 HTTP_CORRELATION_ID 的值，则发送简单的 PUT
 // 请求信息
 sink.next("PUT <" + val.getT1() + "> sent to " + url);
 }
 })
 .map(msg -> Tuples.of(200, msg)); // 返回状态码和消息的元组
}

// 测试方法，演示如何调用 doPut 方法，并设置 HTTP_CORRELATION_ID 的值
@Test
public void contextForLibraryReactivePut() {
 doPut("edu.lfsl.com", Mono.just("明朝那些事"))
 .contextWrite(Context.of(HTTP_CORRELATION_ID, "2-
askdjflaksjdfk")) // 设置上下文项的值
 .filter(t -> t.getT1() < 300) // 过滤出状态码小于 300 的结果
 .map(Tuple2::getT2) // 提取消息部分
 .log() // 打印日志
 .subscribe(); // 订阅流
}

```

#### • ParallelFlux:

##### ◦ 并发流

- // 创建一个从1到100的整数序列的Flux

```

Flux.range(1, 100)
 // 将序列分成每个包含10个元素的缓冲区
 .buffer(10)
 // 将流并行化，使用8个并行线程处理数据
 .parallel(8)
 // 在并行流中切换线程池，使用名为 "xx" 的新并行调度器
 .runOn(Schedulers.newParallel("xx"))
 // 打印日志，记录流中的元素
 .log()
 // 订阅流，启动数据处理
 .subscribe();

```

# WebFlux

- Reactor核心：HttpHandler 原生API
- DispatcherHandler原理
  - DispatcherHandler 组件分析
  - DispatcherHandler 请求处理流程
  - 返回结果处理
  - 异常处理
  - 视图解析
    - 重定向
    - Rendering
- 注解式 - Controller
  - 兼容老版本方式
  - 新版本变化
    - SSE
    - 文件上传
- 错误响应
  - @ExceptionHandler
    - ErrorResponse：自定义 错误响应
    - ProblemDetail：自定义PD返回
- WebFlux配置
  - @EnableWebFlux
  - WebFluxConfigurer

WebFlux: 底层完全基于 netty + reactor + springweb 完成一个全异步非阻塞的web响应式框架

底层：异步 + 消息队列（内存）+ 事件回调机制 = 整套系统

优点：能使用少量资源处理大量请求

## 组件对比

API功能	Servlet-阻塞式Web	WebFlux-响应式Web
前端控制器	DispatcherServlet	DispatcherHandler
处理器	Controller	WebHandler/Controller
请求、响应	<b>ServletRequest,</b> **ServletResponse**	<b>ServerWebExchange:</b> <b>ServerHttpRequest,</b> <b>ServerHttpResponse</b>
过滤器	Filter (HttpFilter)	WebFilter
异常处理器	HandlerExceptionResolver	DispatchExceptionHandler
Web配置	@EnableWebMvc	@EnableWebFlux
自定义配置	WebMvcConfigurer	WebFluxConfigurer
返回结果	任意	<b>Mono、 Flux、 任意</b>
发送REST请求	RestTemplate	WebClient

# 引入

底层基于Netty实现的Web容器与请求/响应处理机制

参照: <https://docs.spring.io/spring-framework/reference/6.0/web/webflux/reactive-spring.html>

```
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>3.1.11</version>
</parent>

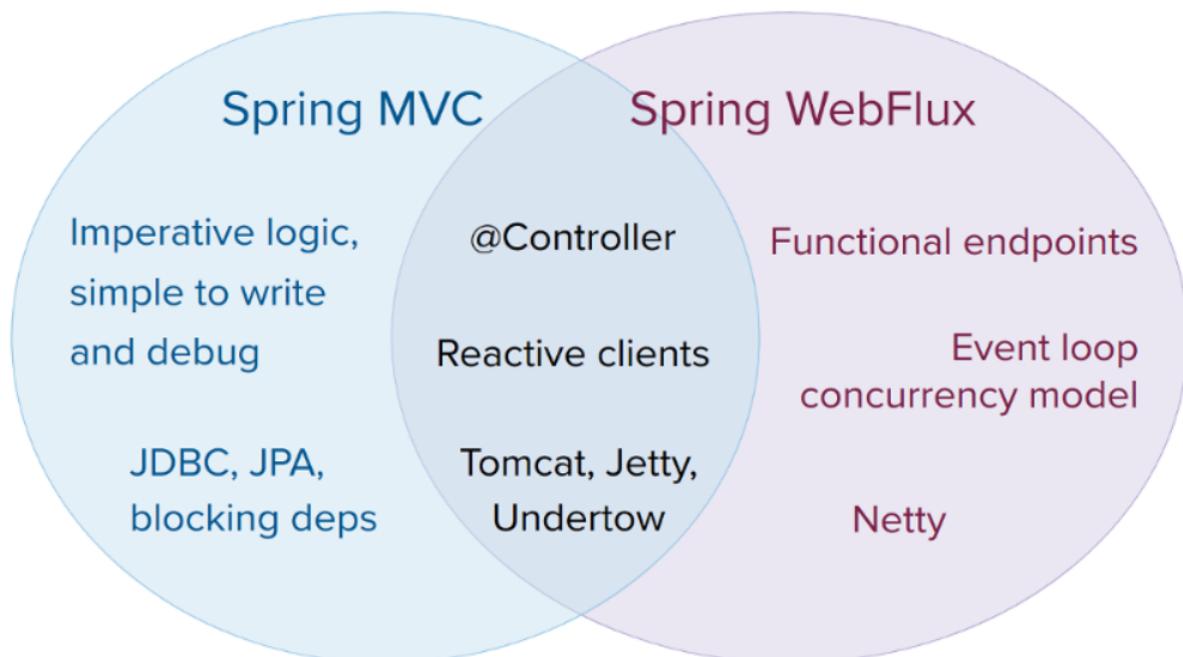
<dependencies>
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-webflux</artifactId>
 </dependency>
</dependencies>
```

Context响应式上下文数据传递; 由下游传播给上游;

以前: 浏览器 -> Controller -> Service -> Dao; 阻塞式进程

现在: Dao(数据源查询对象【数据发布者】) -> Service -> Controller -> 浏览器: 响应式

```
// 大数据流程: 从一个数据源拿到大量数据进行分析计算;
ProductVistorDao.loadData()
 .distinct()
 .map()
 .filter()
 .handle()
 .subscribe();
//加载最新的商品浏览数据
```



# Reactor Core

## HttpHandler/HttpServer

```
public static void main(String[] args) throws IOException {
 //快速自己编写一个能处理请求的服务器

 //1、创建一个能处理Http请求的处理器。 参数：请求、响应； 返回值：Mono<Void>：代表处理完成的信号
 HttpHandler handler = (ServerHttpRequest request,
 ServerHttpResponse response)->{
 URI uri = request.getURI();
 System.out.println(Thread.currentThread()+"请求进来: "+uri);
 // 编写请求处理的业务，给浏览器写一个内容URL + "Hello~!"
 //response.getHeaders(); // 响应头
 //response.getCookies(); // 响应Cookie
 //response.setStatusCode(null); // 响应状态码
 //response.bufferFactory(); // buffer工厂
 //response.writeWith(); // 把xxx写出去
 //response.setComplete(); // 响应完成

 //数据的发布者：Mono<DataBuffer>、Flux<DataBuffer>

 //创建 响应数据的 DataBuffer
 DataBufferFactory factory = response.bufferFactory();

 //数据Buffer
 DataBuffer buffer = factory.wrap(new String(uri.toString() + " ==> Hello!").getBytes());

 // 需要一个 DataBuffer 的发布者
 return response.writeWith(Mono.just(buffer));
 };

 //2、启动一个服务器，监听8080端口，接受数据，拿到数据交给 HttpHandler 进行请求处理
 ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(handler);

 //3、启动Netty服务器
 HttpServer.create()
 .host("localhost")
 .port(8080)
 .handle(adapter) //用指定的处理器处理请求
 .bindNow(); //现在就绑定

 System.out.println("服务器启动完成....监听8080，接受请求");
 System.in.read();
 System.out.println("服务器停止....");
}
```

## SSE

SSE (Server-Sent Events) 是一种基于HTTP的技术，用于服务器向客户端推送数据。它允许服务器持续地向客户端发送数据，而不需要客户端发出请求。通常用于实时更新网页内容，例如实时股票报价、即时通讯等。SSE基于简单的文本格式，易于实现和使用，并且在现代Web应用程序中具有广泛的应用。

SSE: 单工，请求过去以后，等待服务端源源不断的数据

WebSocket: 双工：连接建立后，可以任何交互；

```
// text/event-stream
// SSE测试
// 映射GET请求到路径"/sse"，并且指定返回类型为text/event-stream，即SSE格式的文本事件流
@GetMapping(value = "/sse", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<ServerSentEvent<String>> sse() {
 // 创建一个Flux序列，包含从1到10的数字
 return Flux.range(1, 10)
 // 将每个数字映射为一个ServerSentEvent对象
 .map(i -> {
 // 使用ServerSentEvent的Builder模式创建一个新的ServerSentEvent对象
 return ServerSentEvent.builder("ha-" + i)
 // 设置事件的ID为当前数字的字符串形式
 .id(i + "")
 // 设置事件的注释为"hei-" + 当前数字
 .comment("hei-" + i)
 // 设置事件的类型为"haha"
 .event("haha")
 // 构建ServerSentEvent对象
 .build();
 })
 // 设置每个事件之间的发送间隔为500毫秒
 .delayElements(Duration.ofMillis(500));
}
// 然后前端进行sse的请求
```

## DispatcherHandler

SpringMVC: DispatcherServlet  
SpringWebFlux: DispatcherHandler

### 1. 请求处理流程

- HandlerMapping: 请求映射处理器；保存每个请求由那个方法进行处理
- HandlerAdapter: 处理器适配器；反射执行目标方法
- HandlerResultHandler: 处理器结果处理器

SpringMVC: DispatcherServlet 有一个 doDispatch() 方法，来处理所有请求

WebFlux: DispatcherHandler 有一个 handle() 方法，来处理所有请求

```
public Mono<Void> handle(ServerWebExchange exchange) {
 if (this.handlerMappings == null) {
 return createNotFoundError();
 }
 if (CorsUtils.isPreFlightRequest(exchange.getRequest())) {
 return handlePreFlight(exchange);
 }
 //拿到所有的 handlerMappings
 return Flux.fromIterable(this.handlerMappings)
 //找每一个mapping看谁能处理请求
 .concatMap(mapping -> mapping.getHandler(exchange))
 //直接触发获取元素； 拿到流的第一个元素； 找到第一个能处理这个请求的handlerAdapter
 .next()
 //如果没拿到这个元素，则响应404错误；
 .switchIfEmpty(createNotFoundError())
 //异常处理，一旦前面发生异常，调用处理异常
 .onErrorResume(ex -> handleDispatchError(exchange, ex))
 //调用方法处理请求，得到响应结果
```

```
.flatMap(handler -> handleRequestWith(exchange, handler));
}
```

1. 请求和响应都封装在ServerWebExchange对象中，由handle方法进行处理
2. 如果没有任何的请求映射器；直接返回一个：创建一个未找到的错误；404，返回Mono.error；终结流

```
private <R> Mono<R> createNotFoundError() {
 Exception ex = new ResponseStatusException(HttpStatus.NOT_FOUND);
 return Mono.error(ex);
}
Mono.defer(() -> {
 Exception ex = new ResponseStatusException(HttpStatus.NOT_FOUND);
 return Mono.error(ex);
}); //有订阅者，且流被激活后就动态调用这个方法； 延迟加载;
```

3. 跨域工具、是否跨域请求，跨域请求检查是否复杂跨域，需要预检请求
4. Flux流式操作，先找到HandlerMapping，再获取handlerAdapter，再用Adapter处理请求，期间的错误由onErrorResume触发回调进行处理；

源码中的核心两个：

- **handleRequestWith**: 编写了handlerAdapter怎么处理请求
- **handleResult**: String、User、ServerSendEvent、Mono、Flux ...

## 注解开发

### 目标方法传参

<https://docs.spring.io/spring-framework/reference/6.0/web/webflux/controller/ann-methods/arguments.html>

Controller method argument	Description
ServerWebExchange	封装了请求和响应对象的对象; 自定义获取数据、自定义响应
ServerHttpRequest, ServerHttpResponse	请求、响应
WebSession	访问Session对象
java.security.Principal	
org.springframework.http.HttpMethod	请求方式
java.util.Locale	国际化
java.util.TimeZone + java.time.ZoneId	时区
@PathVariable	路径变量
@MatrixVariable	矩阵变量
@RequestParam	请求参数
@RequestHeader	请求头
@CookieValue	获取Cookie
@RequestBody	获取请求体, Post、文件上传
HttpEntity	封装后的请求对象
@RequestPart	获取文件上传的数据 multipart/form-data.
java.util.Map, org.springframework.ui.Model, and org.springframework.ui.ModelMap.	Map、Model、ModelMap
@ModelAttribute	
Errors, BindingResult	数据校验, 封装错误
SessionStatus + class-level @SessionAttributes	
UriComponentsBuilder	For preparing a URL relative to the current request's host, port, scheme, and context path. See <a href="#">URI Links</a> .
@SessionAttribute	
@RequestAttribute	转发请求的请求域数据
Any other argument	所有对象都能作为参数: 1、基本类型 , 等于标注@RequestParam 2、对象类型, 等于标注 @ModelAttribute

## 返回值写法

Controller method return value	Description
@ResponseBody	把响应数据写出去，如果是对象，可以自动转为json
HttpEntity, ResponseEntity	ResponseEntity：支持快捷自定义响应内容
HttpHeaders	没有响应内容，只有响应头
ErrorResponse	快速构建错误响应
ProblemDetail	错误响应格式
String	就是和以前的使用规则一样；forward: 转发到一个地址 redirect: 重定向到一个地址配合模板引擎
View	直接返回视图对象
java.util.Map, org.springframework.ui.Model	以前一样
@ModelAttribute	以前一样
Rendering	新版的页面跳转API；不能标注 @ResponseBody 注解
void	仅代表响应完成信号
Flux, Observable, or other reactive type	使用 text/event-stream 完成SSE效果
Other return values	未在上述列表的其他返回值，都会当成给页面的数据；

## 文件上传

<https://docs.spring.io/spring-framework/reference/6.0/web/webflux/controller/ann-methods/multipart-forms.html>

```
@Controller
public class FileuploadController {

 // 上传文件路径
 private static final String UPLOAD_DIR = "/path/to/upload/directory/";
 // 允许的文件类型
 private static final String[] ALLOWED_CONTENT_TYPES = new String[]{
 "image/jpeg", "image/png", "image/gif", "image/bmp", "image/webp",
 "image/tiff", "image/x-icon", "image/svg+xml"
 };

 @PostMapping("/form")
 public String handleFormUpload(
 @ModelAttribute From form,
 BindingResult errors,
 @RequestParam("file") MultipartFile file,
 Model model) {
 // 检查是否有上传的文件
 if (file.isEmpty()) {
 errors.rejectValue("file", "file.empty", "Please select a file to upload");
 return "form";
 }

 // 检查文件大小
 }
}
```

```

 if (file.getSize() > MAX_FILE_SIZE) {
 errors.rejectValue("file", "file.size", "File size exceeds maximum
allowed size");
 return "form";
 }
 // 检查文件类型
 String fileType = file.getContentType();
 boolean validContentType = isValidContentType(fileType);
 if (!validContentType) {
 errors.rejectValue("file", "file.type", "Invalid file type");
 return "form";
 }

 // 保存文件到服务器
 try {
 saveFile(file);
 } catch (IOException e) {
 errors.rejectValue("file", "file.save.error", "Failed to save
file");
 return "form";
 }

 // 处理表单中的其他字段（例如: name）
 String name = form.getName();
 form.setFile(file);
 // 进行其他处理...

 return "redirect:/success";
 }

 // 检查文件类型是否合法
 private boolean isValidContentType(String contentType) {
 for (String allowedType : ALLOWED_CONTENT_TYPES) {
 if (allowedType.equals(contentType)) {
 return true;
 }
 }
 return false;
 }

 // 保存文件到服务器
 private void saveFile(MultipartFile file) throws IOException {
 byte[] bytes = file.getBytes();
 Path path = Paths.get(UPLOAD_DIR + file.getOriginalFilename());
 Files.write(path, bytes);
 }
}

@Data
class Form {
 private String name;
 private MultipartFile file;
 // ...
}

```

现在：响应式

```

@RestController
public class FileuploadController {
 // 上传文件路径
}

```

```
private static final String UPLOAD_DIR = File.separator + "upload" +
File.separator +"directory" + File.separator;
// 允许的文件类型
private static final List<String> ALLOWED_CONTENT_TYPES = List.of(
 "image/jpeg", "image/png", "image/gif", "image/bmp", "image/webp",
"image/tiff", "image/x-icon", "image/svg+xml"
);

// 处理表单提交的方法
@PostMapping("/form")
public Mono<String> handleFormUpload(
 // 接收上传文件的Mono流
 @RequestPart("file") Mono<FilePart> filePartMono,
 // Spring MVC模型对象
 Model model) {
 return filePartMono.flatMap(filePart -> {
 // 获取文件名和文件大小
 String originalFilename = filePart.filename();
 long fileSize = filePart.headers().getContentLength();

 // 检查文件大小是否超过最大允许大小
 if (fileSize > MAX_FILE_SIZE) {
 return Mono.just("文件大小超过最大允许大小");
 }

 // 获取文件类型并检查是否合法
 String fileType = filePart.headers().getContentType().toString();
 if (!isValidContentType(fileType)) {
 return Mono.just("无效的文件类型");
 }

 return filePart.transferTo(Paths.get(UPLOAD_DIR +
originalFilename))
 .then(Mono.defer(() -> {
 // 处理表单中的其他字段（例如：name）
 String name = "示例名称"; // 例如：从表单中获取
 // 进行其他操作...
 return Mono.just("success");
 }))
 .onErrorResume(error -> Mono.just("文件保存失败"));
 });
}

// 检查文件类型是否合法
private boolean isValidContentType(String contentType) {
 return ALLOWED_CONTENT_TYPES.contains(contentType);
}
```

# 错误处理

```
@ExceptionHandler(ArithmeticException.class)
public String error(ArithmeticException exception){
 System.out.println("发生了数学运算异常"+exception);

 //返回这些进行错误处理;
 //ProblemDetail: 建造者: 声明式编程、链式调用
 //ErrorResponse :

 return "炸了，哈哈...";
}
```

## RequestContext

通常是指在Web应用程序中表示HTTP请求上下文的对象。它提供了关于当前HTTP请求的各种信息，例如请求的URL、请求参数、请求头部、请求方法等等。

```
@WebServlet("/example")
public class ExampleServlet extends HttpServlet {
 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
 // 获取请求的URL
 String requestUrl = request.getRequestURL().toString();
 System.out.println("Request URL: " + requestUrl);

 // 获取请求方法
 String requestMethod = request.getMethod();
 System.out.println("Request Method: " + requestMethod);

 // 获取请求参数
 String parameterValue = request.getParameter("paramName");
 System.out.println("Request Parameter Value: " + parameterValue);

 // 获取请求头部信息
 String userAgent = request.getHeader("User-Agent");
 System.out.println("User-Agent: " + userAgent);

 // 设置请求属性并在另一个Servlet中获取
 request.setAttribute("exampleAttribute", "exampleValue");

 // 转发请求到另一个Servlet
 request.getRequestDispatcher("/anotherServlet").forward(request,
 response);
 }
}
```

现在，响应式：

```
@Component
public class ExampleHandler {

 public Mono<ServerResponse> handleRequest(ServerRequest request) {
 // 获取请求的URL
 String requestUrl = request.uri().toString();
 System.out.println("Request URL: " + requestUrl);
```

```

 // 获取请求方法
 String requestMethod = request.method().name();
 System.out.println("Request Method: " + requestMethod);

 // 获取请求参数
 Mono<String> parameterValueMono =
request.queryParam("paramName").orElse(Mono.just("No parameter"));
 parameterValueMono.subscribe(parameterValue ->
System.out.println("Request Parameter Value: " + parameterValue));

 // 获取请求头部信息
 String userAgent = request.headers().firstHeader("User-Agent");
 System.out.println("User-Agent: " + userAgent);

 // 构建响应
 return ServerResponse.ok()
 .contentType(MediaType.TEXT_PLAIN)
 .bodyValue("Response from ExampleHandler");
}
}

```

## 自定义Flux配置

WebFluxConfigurer

容器中注入这个类型的组件，重写底层逻辑

```

@Configuration
public class MyWebConfiguration {
 //配置底层
 @Bean
 public WebFluxConfigurer webFluxConfigurer(){

 return new WebFluxConfigurer() {
 @Override
 public void addCorsMappings(CorsRegistry registry) {
 registry.addMapping("/**")
 .allowedHeaders("*")
 .allowedMethods("*")
 .allowedOrigins("localhost");
 }
 };
 }
}

```

## Filter

```

@Component
public class MyWebFilter implements WebFilter {
 @Override
 public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain)
 {
 ServerHttpRequest request = exchange.getRequest();
 ServerHttpResponse response = exchange.getResponse();

 System.out.println("请求处理放行到目标方法之前...");
 Mono<Void> filter = chain.filter(exchange); //放行
 }
}

```

```

 //流一旦经过某个操作就会变成新流
 Mono<Void> voidMono = filter.doOnError(err -> {
 System.out.println("目标方法异常以后...");
 }) // 目标方法发生异常后做事
 .doFinally(signalType -> {
 System.out.println("目标方法执行以后...");
 })// 目标方法执行之后

 //上面执行不花时间。
 return voidMono; //看清楚返回的是谁！！！
}
}

```

## R2DBC

Web、网络、IO (存储) 、中间件 (Redis、MySQL)  
应用开发：

- 网络
- **存储**: MySQL、Redis
- **Web**: Webflux
- 前端；后端：Controller -- Service -- Dao (r2dbc; mysql)

数据库：

- **导入驱动**；以前：JDBC (jdbc、各大驱动mysql-connector) ；现在：r2dbc (**r2dbc-spi**、各大驱动 r2dbc-mysql)
- **驱动**：
- 获取连接
- 发送SQL、执行
- 封装数据库返回结果
- r2dbc原生API: <https://r2dbc.io>
- boot整合spring data r2dbc: spring-boot-starter-data-r2dbc
- 三大组件：R2dbcRepository、R2dbcEntityTemplate 、DatabaseClient

## R2dbc

用法：

导入驱动：导入连接池 (r2dbc-pool)、导入启动(r2dbc-mysql)  
使用驱动提供的API操作

```

<dependency>
 <groupId>io.asyncer</groupId>
 <artifactId>r2dbc-mysql</artifactId>
 <version>1.0.5</version>
</dependency>

```

```

//0、MySQL配置
MysqlConnectionConfiguration configuration =
MysqlConnectionConfiguration.builder()
 .host("localhost")
 .port(3306)
 .username("root")
 .password("123456")

```

```

 .database("test")
 .build();

//1、获取连接工厂
MySqlConnectionFactory connectionFactory =
MySqlConnectionFactory.from(configuration);

//2、获取到连接，发送sql

// JDBC: Statement: 封装sql的
//3、数据发布者
Mono.from(connectionFactory.create())
 .flatMapMany(connection ->
 connection
 .createStatement("select * from t_author where id=?id and
name=?name")
 .bind("id",1L) //具名参数
 .bind("name","张三")
 .execute()
).flatMap(result -> {
 return result.map(readable -> {
 Long id = readable.get("id", Long.class);
 String name = readable.get("name", String.class);
 return new TAuthor(id, name);
 });
 })
 .subscribe(tAuthor -> System.out.println("tAuthor = " + tAuthor));

```

## Spring Data R2DBC

提升生产力方式的响应式数据库操作

### 整合

导入依赖

```

<!-- https://mvnrepository.com/artifact/io.asyncer/r2dbc-mysql -->
<dependency>
 <groupId>io.asyncer</groupId>
 <artifactId>r2dbc-mysql</artifactId>
 <version>1.0.5</version>
</dependency>
<!-- 响应式 Spring Data R2dbc-->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-r2dbc</artifactId>
</dependency>

```

编写配置

```

spring:
 r2dbc:
 password: 123456
 username: root
 url: r2dbc:mysql://localhost:3306/test
 name: test

```

# 数据库编程式组件

## 声明式接口: R2dbcRepository

```
@Repository
public interface AuthorRepositories extends R2dbcRepository<TAuthor, Long> {

 //默认继承了一堆CRUD方法; 像mybatis-plus

 //QBC: Query By Criteria
 //QBE: Query By Example

 //成为一个起名工程师 where id in () and name like ?
 //仅限单表复杂条件查询
 Flux<TAuthor> findAllByIdInAndNameLike(Collection<Long> id, String name);

 //多表复杂查询

 @Query("select * from t_author") //自定义query注解, 指定sql语句
 Flux<TAuthor> findHaha();

 // 1-1: 关联
 // 1-N: 关联
 //场景:
 // 1、一个图书有唯一作者; 1-1
 // 2、一个作者可以有很多图书; 1-N
}
```

## R2dbcEntityTemplate

```
@Test
void r2dbcEntityTemplate() throws IOException {
 // Query By Criteria: QBC

 // 创建查询条件
 Criteria criteria =
 Criteria.empty()
 .and("id")
 .is(1L)
 .and("name")
 .is("张三");

 // 创建查询对象
 Query query = Query.query(criteria);

 // 使用 r2dbcEntityTemplate 执行查询并订阅结果
 r2dbcEntityTemplate.select(query, TAuthor.class)
 .subscribe(Tauthor -> System.out.println("Tauthor: " +
 Tauthor));

 // 阻塞主线程, 等待结果
 System.in.read();
}
```

## DatabaseClient

1-1: 一个图书有唯一作者

```
databaseClient.sql("select a.name name, b.* from t_author a left join t_book b
on a.id=b.author_id where b.id = ?")
 // 绑定参数到查询中的占位符
 .bind(0, 1L)
 // 执行数据库查询并获取结果
 .fetch()
 // 获取所有结果
 .all()
 // 将查询结果映射为自定义对象
 .map(row -> {
 Long id = (Long) row.get("id");
 String title = row.get("title").toString();
 String name = row.get("name").toString();
 Instant publishTime = (Instant) row.get("publishTime");

 // 创建 TBook 对象并设置属性
 TBook tBook = new TBook();
 tBook.setId(id);
 tBook.setTitle(title);
 tBook.setAuthorId(id);
 tBook.setPublishTime(publishTime);

 // 创建 TAutor 对象并设置属性
 TAutor tAuthor = new TAutor();
 tAuthor.setId(id);
 tAuthor.setName(name);
 tBook.setAuthor(tAuthor);

 return tBook;
 })
 // 订阅并打印结果
 .subscribe(tBook -> System.out.println("tBook = " + tBook));
```

1-N: 一个作者可以有很多图书

```
@Test
void oneToN() throws IOException {

 // databaseClient.sql("select a.id aid,a.name,b.* from t_author a " +
 // "left join t_book b on a.id = b.author_id " +
 // "order by a.id")
 // .fetch()
 // .all(row -> {
 // })

 // 1~6
 // 1: false 2: false 3:false 4: true 8:true 5:false 6:false 7:false 8:true
 // 9:false 10:false
 // [1,2,3]
 // [4,8]
 // [5,6,7]
 // [8]
 // [9,10]
 // bufferUntilChanged:
 // 如果下一个判定值比起上一个发生了变化就开一个新buffer保存, 如果没有变化就保存到原buffer中
```

```

// Flux.just(1,2,3,4,8,5,6,7,8,9,10)
// .bufferUntilChanged(integer -> integer%4==0)
// .subscribe(list-> System.out.println("list = " + list));
//自带分组

// 数据库查询并根据作者ID进行分组处理
Flux<TAuthor> flux = databaseClient.sql("select a.id aid,a.name,b.* from
t_author a " +
 "left join t_book b on a.id = b.author_id " +
 "order by a.id")
// 执行数据库查询并获取结果
.fetch()
// 获取所有结果
.all()
// 根据作者ID进行分组
.bufferUntilChanged(rowMap ->
Long.parseLong(rowMap.get("aid").toString()))
// 将查询结果映射为自定义对象
.map(list -> {
 TAutor tAuthor = new TAutor();
 Map<String, Object> map = list.get(0);
 tAuthor.setId(Long.parseLong(map.get("aid").toString()));
 tAuthor.setName(map.get("name").toString());

 // 处理每个作者的书籍信息
 List<TBook> tBooks = list.stream()
 .map(ele -> {
 TBook tBook = new TBook();

 tBook.setId(Long.parseLong(ele.get("id").toString()));

 tBook.setAuthorId(Long.parseLong(ele.get("author_id").toString()));
 tBook.setTitle(ele.get("title").toString());
 return tBook;
 })
 .collect(Collectors.toList());

 tAuthor.setBooks(tBooks);
 return tAuthor;
 });

// 订阅并打印结果
flux.subscribe(tAuthor -> System.out.println("tAuthor = " + tAuthor));

System.in.read();
}

```

## 自定义Converter

会对以前的CRUD产生影响, 如果使用简单查询, 不需要进行Converter转换, 但是Converter依然会进行转换, 因为没有这个列, 就会报错

1. 使用新的VO + 新的Repository + 自定义类型转换器: 重新定义一个实体类和Repository, 使简单查询和需要Converter转换的复杂查询分开
2. 自定义类型转换器, 多写判断按条件, 兼容更多类型

实体类

```

@Table("t_book")
@Data
public class TBook {
 @Id
 private Long id;
 private String title;
 private Long authorId;
 private Instant publishTime;
 private TAuthor author; //唯一作者
}

```

自定义converter

```

// 读取数据库数据的时候,把row转成TBook
@ReadingConverter
public class BookConverter implements Converter<Row, TBook> {
 @Override
 public TBook convert(Row source) {
 // 如果源数据为空, 则返回null
 if (source == null) return null;

 // 从源数据中提取作者ID
 Long authorId = source.get("authorId", Long.class);

 // 创建 TBook 对象并设置属性
 TBook tBook = new TBook();
 tBook.setId(source.get("id", Long.class));
 tBook.setTitle(source.get("title", String.class));
 tBook.setAuthorId(authorId);
 tBook.setPublishTime(source.get("publish_time", Instant.class));

 // 如果结果集中包含作者名字, 则创建 TAuthor 对象并设置属性
 if (source.getMetadata().contains("name")) {
 TAuthor tAuthor = new TAuthor();
 tAuthor.setId(authorId);
 tAuthor.setName(source.get("name", String.class));
 tBook.setAuthor(tAuthor);
 }
 return tBook;
 }
}

```

使自定义converter生效

```

@EnableR2dbcRepositories //开启 R2dbc 仓库功能; jpa
@Configuration
public class R2dbcConfiguration {
 @Bean //替换容器中原来的
 @ConditionalOnMissingBean
 public R2dbcCustomConversions conversions() {
 //把我们的转换器加入进去; 效果新增了我们的 Converter
 return R2dbcCustomConversions.of(MySqlDialect.INSTANCE, new
BookConverter());
 }
}

```

自定义Converter<Row,Bean>方式

```

@EnableR2dbcRepositories //开启 R2dbc 仓库功能; jpa
@Configuration

```

```

public class R2dbcConfiguration {
 @Bean
 R2dbcCustomConversions r2dbcCustomConversions(){
 List<Converter<?, ?>> converters = new ArrayList<>();
 converters.add(new BookConverter());
 return R2dbcCustomConversions.of(MySqlDialect.INSTANCE, converters);
 }
}

//1-1: 结合自定义 Converter
// bookRepostory.hahaBook(1L)
// .subscribe(tBook -> System.out.println("tBook = " + tBook));

```

## 最佳实践

最佳实践: 提升生产效率的做法

- Spring Data R2DBC, 基础的CURD用 R2dbcRepository 提供好了
- 自定义复杂的SQL(单表): @Query
- 多表查询复杂结果集: DatabaseClient 自定义SQL及结果封装
  - @Query + 自定义 Converter 实现结果封装

经验:

- 1-1, 1-N 关联关系的封装都需要自定义结果集的方式
  - Spring Data R2DBC
    - 自定义Converter指定结果封装
    - DatabaseClient: 贴近底层的操作进行封装
  - MyBatis: 自定义 ResultMap 标签去来封装

RBAC SQL文件

```

-- 用户表
DROP TABLE IF EXISTS `t_user`;
CREATE TABLE `t_user`(
 `id` bigint(20) NOT NULL AUTO_INCREMENT,
 `username` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL COMMENT '用户名',
 `password` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL COMMENT '密码',
 `email` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL COMMENT '邮箱',
 `phone` char(11) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT NULL
COMMENT '电话',
 `create_time` datetime(0) NOT NULL COMMENT '创建时间',
 `update_time` datetime(0) NOT NULL COMMENT '更新时间',
 PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_0900_ai_ci
ROW_FORMAT = Dynamic;

-- 角色表
DROP TABLE IF EXISTS `t_roles`;
CREATE TABLE `t_roles`(
 `id` bigint(20) NOT NULL AUTO_INCREMENT,
 `name` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL COMMENT '角色名',
 `value` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL COMMENT '角色的英文名',
 `create_time` datetime(0) NOT NULL,
 `update_time` datetime(0) NOT NULL
)
```

```

 `update_time` datetime(0) NOT NULL,
 PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_0900_ai_ci
ROW_FORMAT = Dynamic;

-- 权限表（资源表）
DROP TABLE IF EXISTS `t_perm`;
CREATE TABLE `t_perm`(
 `id` bigint(20) NOT NULL AUTO_INCREMENT,
 `value` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL COMMENT '权限字段',
 `uri` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT NULL
COMMENT '资源路径',
 `description` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci
NOT NULL COMMENT '资源描述',
 `create_time` datetime(0) NOT NULL,
 `update_time` datetime(0) NOT NULL,
 PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_0900_ai_ci
ROW_FORMAT = Dynamic;

-- 用户角色关系表
DROP TABLE IF EXISTS `t_user_role`;
CREATE TABLE `t_user_role`(
 `id` bigint(20) NOT NULL AUTO_INCREMENT,
 `user_id` bigint(20) NOT NULL,
 `role_id` bigint(20) NOT NULL,
 `create_time` datetime(0) NOT NULL,
 `update_time` datetime(0) NOT NULL,
 PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_0900_ai_ci
ROW_FORMAT = Dynamic;

-- 角色权限关系表
DROP TABLE IF EXISTS `t_role_perm`;
CREATE TABLE `t_role_perm`(
 `id` bigint(20) NOT NULL AUTO_INCREMENT,
 `role_id` bigint(20) NOT NULL,
 `perm_id` bigint(20) NOT NULL,
 `create_time` datetime(0) NOT NULL,
 `update_time` datetime(0) NOT NULL,
 PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_0900_ai_ci
ROW_FORMAT = Dynamic;

-- 图书&作者表
CREATE TABLE `t_book`(
 `id` bigint(20) NOT NULL AUTO_INCREMENT,
 `title` varchar(255) NOT NULL,
 `author_id` bigint(20) NOT NULL,
 `publish_time` datetime(0) NOT NULL,
 PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_0900_ai_ci
ROW_FORMAT = Dynamic;

CREATE TABLE `t_author`(
 `id` bigint(20) NOT NULL AUTO_INCREMENT,
 `name` varchar(255) NOT NULL,
 PRIMARY KEY (`id`) USING BTREE
)

```

```
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_0900_ai_ci
ROW_FORMAT = Dynamic;
```

# SpringSecurity

SpringBoot + Webflux + Spring Data R2DBC + Spring Security

## 整合

```
<dependencies>
 <!-- https://mvnrepository.com/artifact/io.asyncer/r2dbc-mysql -->
 <dependency>
 <groupId>io.asyncer</groupId>
 <artifactId>r2dbc-mysql</artifactId>
 <version>1.0.5</version>
 </dependency>
 <!-- 响应式 Spring Data R2dbc-->
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-r2dbc</artifactId>
 </dependency>

 <!-- 响应式web -->
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-webflux</artifactId>
 </dependency>

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

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

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

# 开发

## 应用安全

- 防止攻击:
  - DDos、CSRF、XSS、SQL注入...
- 控制权限
  - 登录的用户能干什么。
  - 用户登录系统以后要控制住用户的所有行为，防止越权；
- 传输加密
  - https
  - X509
- 认证:
  - OAuth2.0
  - JWT

## RBAC权限模型

Role Based Access Control: 基于角色的访问控制

一个网站有很多**用户**: zhangsan

每个用户可以有很多**角色**:

一个角色可以关联很多**权限**:

一个人到底能干什么?

权限控制:

- 找到这个人，看他有哪些角色，每个角色能拥有哪些**权限**。这个人就拥有一堆的**角色**或者**权限**
- 这个人执行方法的时候，我们给方法规定好权限，由权限框架负责判断，这个人是否有指定的权限

所有权限框架:

- 让用户登录进来: **认证 (authenticate)** : 用账号密码、各种其他方式，先让用户进来
- 查询用户拥有的所有角色和权限: **授权 (authorize)** : 每个方法执行的时候，匹配角色或者权限来判定用户是否可以执行这个方法

## 认证

登录行为

1. 静态资源放行
2. 其他请求需要登陆

对springSecurity进行配置认证规则

```
@Configuration
@EnableReactiveMethodSecurity
public class AppSecurityConfiguration {

 @Autowired
 AppReactiveUserDetailsService appReactiveUserDetailsService;

 @Bean
 public SecurityWebFilterChain springSecurityWebFilterChain(ServerHttpSecurity http) {
 // 1. 定义那些请求需要认证，那些不需要
 }
}
```

```

http.authorizeExchange(authorize -> {
 // 2. 允许所有人都访问静态资源
 authorize.matchers(PathRequest.toStaticResources()
 .atCommonLocations())
 .permitAll();

 // 3. 剩下的所有请求都需要认证(登录)
 authorize.anyExchange()
 .authenticated();
});

// 开启默认的表单登录
http.formLogin(Customizer.withDefaults());

// 安全控制
http.csrf(csrfSpec -> {
 csrfSpec.disable();
});

// 配置认证规则: 如何去数据库查询用户
// Spring Security 底层使用 ReactiveAuthenticationManager 查询用户信息
// ReactiveAuthenticationManager 有一个实现是
UserDetailsRepositoryReactiveAuthenticationManager: 用户信息去数据库中查询
 // UDResponse 需要 ReactiveUserDetailsService
 // 我们只需要自己写一个 ReactiveUserDetailsService: 响应式的用户详情查询服务
 http.authenticationManager(new
UserDetailsRepositoryReactiveAuthenticationManager(appReactiveUserDetailsService
));

// 构建出安全配置
return http.build();
}
}
}

```

进入表单点击登录，最终Spring Security 框架会使用 ReactiveUserDetailsService 组件，按照表单提交的用户名去数据库查询这个**用户详情**（基本信息[账号、密码]，**角色**，**权限**）；  
把数据库中返回的**用户详情**中的密码 和 表单提交的密码进行比对。比对成功则登录成功；  
**实现ReactiveUserDetailsService中findByUsername接口,获取用户的详细信息, 提供给Spring Security进行身份验证和授权**

```

@Component
public class AppReactiveUserDetailsService implements ReactiveUserDetailsService {
 @Autowired
 DatabaseClient databaseClient;
 @Autowired
 PasswordEncoder passwordEncoder;

 // 通过用户名查找用户信息
 @Override
 public Mono<UserDetails> findByUsername(String username) {
 // 查询数据库中的用户信息
 Flux<UserDetails> userDetailsFlux = databaseClient.sql("select u.id,
u.username, u.password, r.id rid, r.name, r.value, p.id pid, p.value pvalue,
p.description " +
 "from t_user u " +
 "left join t_user_role ur on ur.user_id = u.id " +
 "left join t_roles r on ur.role_id = r.id " +
 "left join t_role_perm rp on rp.role_id = r.id " +
 "left join t_perm p on rp.perm_id = p.id " +

```

```

 "where u.username=?")
 .bind(0, username)
 .fetch()
 .all()
 // 对查询结果按用户名进行分组
 .bufferUntilChanged(rowMap ->
 rowMap.get("username").toString())
 .map(list -> {
 // 创建一个 TUser 对象
 TUser tUser = new Tuser();
 // 获取结果集的第一行数据，即用户基本信息
 Map<String, object> map = list.get(0);
 tUser.setId(Long.parseLong(map.get("id").toString()));
 tUser.setUsername(map.get("username").toString());
 tUser.setPassword(map.get("password").toString());

 // 处理用户的角色信息
 List<String> roles = list.stream()
 .map(ele -> ele.get("name").toString())
 .collect(Collectors.toList());
 tUser.setRoles(roles);

 // 处理用户的权限信息
 List<String> authorities = list.stream()
 .map(ele -> ele.get("pvalue").toString())
 .collect(Collectors.toList());
 tUser.setAuthorities(authorities);

 // 将 TUser 对象转换为 UserDetails 对象
 UserDetails userDetailsService = convertTUserToUserDetails(tUser);

 return userDetailsService;
 });
 // 将 Flux 转换为 Mono
 return Mono.from(userDetailsServiceFlux);
 }

 // 定义密码编码器 Bean
 @Bean
 PasswordEncoder passwordEncoder() {
 // 创建并返回一个委托给 DelegatingPasswordEncoder 的密码编码器
 PasswordEncoder encoder =
 PasswordEncoderFactories.createDelegatingPasswordEncoder();
 return encoder;
 }

 // 将 TUser 转换为 UserDetails 的逻辑
 private UserDetails convertTUserToUserDetails(TUser tUser) {
 // 将权限字符串列表转换为 SimpleGrantedAuthority 列表
 List<SimpleGrantedAuthority> authorities = tUser.getAuthorities()
 .stream()
 .map(SimpleGrantedAuthority::new)
 .collect(Collectors.toList());

 // 将角色字符串列表转换为 SimpleGrantedAuthority 列表，并在角色名前加上 "ROLE_"
 List<SimpleGrantedAuthority> roles = tUser.getRoles()
 .stream()
 .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
 .collect(Collectors.toList());

 // 将权限和角色合并成一个列表

```

```
 List<SimpleGrantedAuthority> combinedAuthorities = new ArrayList<>();
 combinedAuthorities.addAll(authorities);
 combinedAuthorities.addAll(roles);

 // 创建 UserDetails 对象
 UserDetails userDetails = User.builder()
 .username(tUser.getUsername())
 .password(tUser.getPassword())
 .authorities(combinedAuthorities)
 .build();

 return userDetails;
 }
}
```

## 授权

- 方法级别的授权
  - 需要在 `AppSecurityConfiguration` 中添加一个 `@EnableReactiveMethodSecurity` 注解, 可以实现方法级别的权限和角色的授权

```
@RestController
public class HelloController {
 // 角色 haha: ROLE_haha: 角色
 // 没有ROLE 前缀是权限

 // 只有角色为 ROLE_admin 的用户才可以进行访问
 @PreAuthorize("hasRole('ROLE_admin')")
 @GetMapping("/hello")
 public Mono<String> hello(){
 return Mono.just("hello world!");
 }

 //可以通过复杂的SpEL表达式来进行匹配
 // 只有权限为 view 的用户才可以访问
 @PreAuthorize("hasAuthority('view')")
 @GetMapping("/world")
 public Mono<String> world(){
 return Mono.just("world!!!!");
 }
}
```

官方实例:

<https://github.com/spring-projects/spring-security-samples/tree/main>