Spring Boot

原理篇

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

目录

Contents

自动配置

- bean加载方式（复习）
- bean加载控制（复习）
- bean依赖属性配置
- 自动配置原理
- 变更自动配置

bean的加载方式（一）

- XML方式声明bean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--声明自定义bean-->
    <bean id="bookService"
          class="com.itheima.service.impl.BookServiceImpl"
          scope="singleton"/>
    <!--声明第三方开发bean-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"/>
</beans>
```

bean的加载方式（二）

- XML+注解方式声明bean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       https://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">
    <context:component-scan base-package="com.itheima"/>
</beans>
```

bean的加载方式（二）

- 使用@Component及其衍生注解@Controller 、@Service、@Repository定义bean

```
@Service
public class BookServiceImpl implements BookService {
}
```

- 使用@Bean定义第三方bean，并将所在类定义为配置类或Bean

```
@Component
public class DbConfig {
    @Bean
    public DruidDataSource getDataSource(){
        DruidDataSource ds = new DruidDataSource();
        return ds;
    }
}
```

bean的加载方式（三）

- 注解方式声明配置类

```
@Configuration
@ComponentScan("com.itheima")
public class SpringConfig {
    @Bean
    public DruidDataSource getDataSource(){
        DruidDataSource ds = new DruidDataSource();
        return ds;
    }
}
```

  - @Configuration配置项如果不用于被扫描可以省略

bean的加载方式——扩展1

- 初始化实现FactoryBean接口的类，实现对bean加载到容器之前的批处理操作

```
public class BookFactoryBean implements FactoryBean<Book> {
    public Book getObject() throws Exception {
        Book book = new Book();
        // 进行book对象相关的初始化工作
        return book;
    }
    public Class<?> getObjectType() {
        return Book.class;
    }
}
```

```
public class SpringConfig8 {
    @Bean
    public BookFactoryBean book(){
        return new BookFactoryBean();
    }
}
```

bean的加载方式——扩展2

- 加载配置类并加载配置文件（系统迁移）

```
@Configuration
@ComponentScan("com.itheima")
@ImportResource("applicationContext-config.xml")
public class SpringConfig2 {
}
```

bean的加载方式——扩展3

- 使用proxyBeanMethods=true可以保障调用此方法得到的对象是从容器中获取的而不是重新创建的

```
@Configuration(proxyBeanMethods = false)
public class SpringConfig3 {
    @Bean
    public Book book(){
        System.out.println("book init ...");
        return new Book();
    }
}
```

```
public class AppObject {
    public static void main(String[] args) {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(Config.class);
        SpringConfig3 config = ctx.getBean("Config", Config.class);
        config.book();
        config.book();
    }
}
```

bean的加载方式（四）

- 使用@Import注解导入要注入的bean对应的字节码

```
@Import(Dog.class)
public class SpringConfig5 {
}
```

- 被导入的bean无需使用注解声明为bean

```
public class Dog {
}
```

  - 此形式可以有效的降低源代码与Spring技术的耦合度，在spring技术底层及诸多框架的整合中大量使用

bean的加载方式——扩展4

- 使用@Import注解导入配置类

```
@Import(DbConfig.class)
public class SpringConfig {
}
```

bean的加载方式（五）

- 使用上下文对象在容器初始化完毕后注入bean

```
public class AppImport {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext(SpringConfig5.class);
        ctx.register(Cat.class);
        String[] names = ctx.getBeanDefinitionNames();
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

bean的加载方式（六）

- 导入实现了ImportSelector接口的类，实现对导入源的编程式处理

```
public class MyImportSelector implements ImportSelector {
    public String[] selectImports(AnnotationMetadata metadata) {
        boolean flag = metadata.hasAnnotation("org.springframework.context.annotation.Import");
        if(flag){
            return new String[]{"com.itheima.domain.Dog"};
        }
        return new String[]{"com.itheima.domain.Cat"};
    }
}
```

bean的加载方式（七）

- 导入实现了ImportBeanDefinitionRegistrar接口的类，通过BeanDefinition的注册器注册实名bean，实现对容器中bean的裁定，例如对现有bean的覆盖，进而达成不修改源代码的情况下更换实现的效果

```
public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
                                        BeanDefinitionRegistry registry) {
        BeanDefinition beanDefinition = BeanDefinitionBuilder
                                        .rootBeanDefinition(BookServiceImpl2.class)
                                        .getBeanDefinition();
        registry.registerBeanDefinition("bookService", beanDefinition);
    }
}
```

bean的加载方式（八）

- 导入实现了BeanDefinitionRegistryPostProcessor接口的类，通过BeanDefinition的注册器注册实名bean，实现对容器中bean的最终裁定

```
public class MyPostProcessor implements BeanDefinitionRegistryPostProcessor {
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {
        BeanDefinition beanDefinition = BeanDefinitionBuilder
                                        .rootBeanDefinition(BookServiceImpl4.class)
                                        .getBeanDefinition();
        registry.registerBeanDefinition("bookService", beanDefinition);
    }
}
```

1. xml+<bean/>
2. xml:context+注解（@Component+4个@Bean）
3. 配置类+扫描+注解（@Component+4个@Bean）
   - @Bean定义FactoryBean接口
   - @ImportResource
   - @Configuration注解的proxyBeanMethods属性
4. @Import导入bean的类
   - @Import导入配置类
5. AnnotationConfigApplicationContext调用register方法
6. @Import导入ImportSelector接口
7. @Import导入ImportBeanDefinitionRegistrar接口
8. @Import导入BeanDefinitionRegistryPostProcessor接口

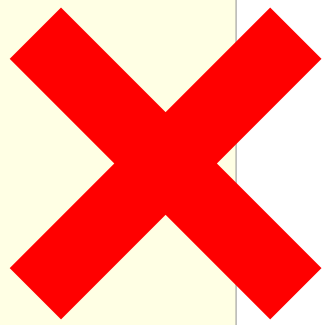
bean的加载控制

- bean的加载控制指根据特定情况对bean进行选择性加载以达到适用于项目的目标。

bean的加载方式（一）

- XML方式声明bean

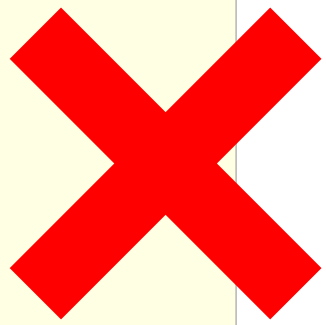
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--声明自定义bean-->
    <bean id="bookService"
          class="com.itheima.service.impl.BookServiceImpl"
          scope="singleton"/>
    <!--声明第三方开发bean-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"/>
</beans>
```



bean的加载方式（二）

- XML+注解方式声明bean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       https://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">
    <context:component-scan base-package="com.itheima"/>
</beans>
```



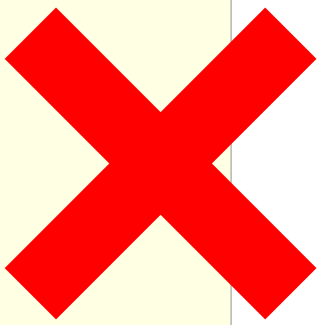
bean的加载方式（二）

- 使用@Component及其衍生注解@Controller 、@Service、@Repository定义bean

```
@Service
public class BookServiceImpl implements BookService {
}
```

- 使用@Bean定义第三方bean，并将所在类定义为配置类或Bean

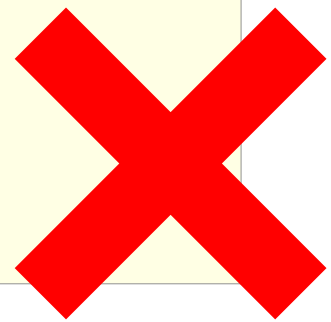
```
@Component
public class DbConfig {
    @Bean
    public DruidDataSource getDataSource(){
        DruidDataSource ds = new DruidDataSource();
        return ds;
    }
}
```



bean的加载方式（三）

- 注解方式声明配置类

```
@Configuration
@ComponentScan("com.itheima")
public class SpringConfig {
    @Bean
    public DruidDataSource getDataSource(){
        DruidDataSource ds = new DruidDataSource();
        return ds;
    }
}
```



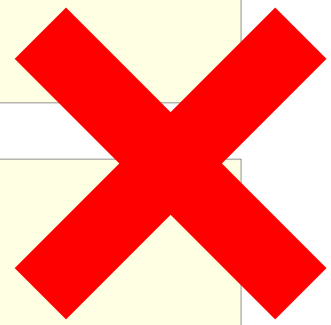
- @Configuration配置项如果不用于被扫描可以省略

bean的加载方式——扩展1

- 初始化实现FactoryBean接口的类，实现对bean加载到容器之前的批处理操作

```
public class BookFactoryBean implements FactoryBean<Book> {
    public Book getObject() throws Exception {
        Book book = new Book();
        // 进行book对象相关的初始化工作
        return book;
    }
    public Class<?> getObjectType() {
        return Book.class;
    }
}
```

```
public class SpringConfig8 {
    @Bean
    public BookFactoryBean book(){
        return new BookFactoryBean();
    }
}
```



bean的加载方式——扩展2

- 加载配置类并加载配置文件（系统迁移）

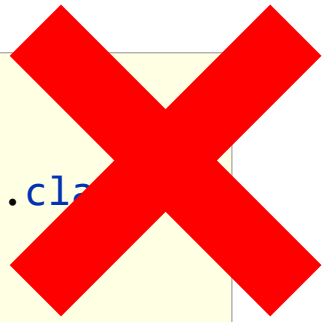
```
@Configuration
@ComponentScan("com.itheima")
@ImportResource("applicationContext-config.xml")
public class SpringConfig2 {
}
```

bean的加载方式——扩展3

- 使用proxyBeanMethods=true可以保障调用此方法得到的对象是从容器中获取的而不是重新创建的

```
@Configuration(proxyBeanMethods = false)
public class SpringConfig3 {
    @Bean
    public Book book(){
        System.out.println("book init ...");
        return new Book();
    }
}
```

```
public class AppObject {
    public static void main(String[] args) {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(Config.cla
        SpringConfig3 config = ctx.getBean("Config", Config.class);
        config.book();
        config.book();
    }
}
```



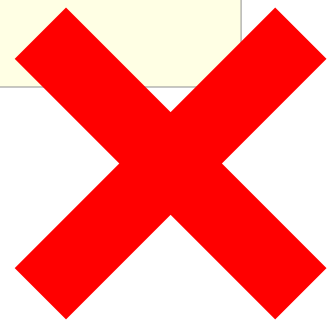
bean的加载方式（四）

- 使用@Import注解导入要注入的bean对应的字节码

```
@Import(Dog.class)
public class SpringConfig5 {
}
```

- 被导入的bean无需使用注解声明为bean

```
public class Dog {
}
```



  - 此形式可以有效的降低源代码与Spring技术的耦合度，在spring技术底层及诸多框架的整合中大量使用

bean的加载方式——扩展4

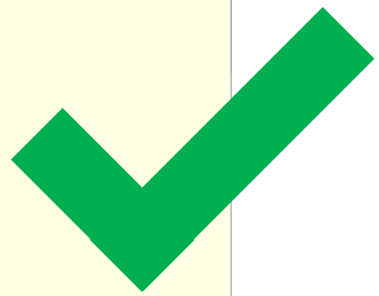
- 使用@Import注解导入配置类

```
@Import(DbConfig.class)
public class SpringConfig {
}
```

bean的加载方式（五）

- 使用上下文对象在容器初始化完毕后注入bean

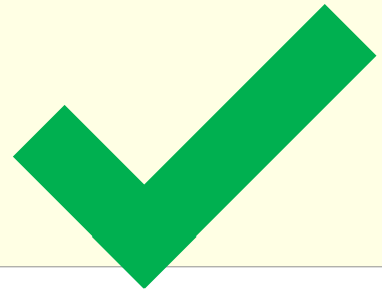
```
public class AppImport {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext(SpringConfig5.class);
        ctx.register(Cat.class);
        String[] names = ctx.getBeanDefinitionNames();
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```



bean的加载方式（六）

- 导入实现了ImportSelector接口的类，实现对导入源的编程式处理

```
public class MyImportSelector implements ImportSelector {
    public String[] selectImports(AnnotationMetadata metadata) {
        boolean flag = metadata.hasAnnotation("org.springframework.context.annotation.Import");
        if(flag){
            return new String[]{"com.itheima.domain.Dog"};
        }
        return new String[]{"com.itheima.domain.Cat"};
    }
}
```



bean的加载方式（七）

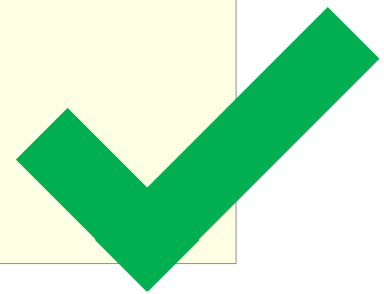
- 导入实现了ImportBeanDefinitionRegistrar接口的类，通过BeanDefinition的注册器注册实名bean，实现对容器中bean的裁定，例如对现有bean的覆盖，进而达成不修改源代码的情况下更换实现的效果

```
public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
                                        BeanDefinitionRegistry registry) {
        BeanDefinition beanDefinition = BeanDefinitionBuilder
                                        .rootBeanDefinition(BookServiceImpl2.class)
                                        .getBeanDefinition();
        registry.registerBeanDefinition("bookService", beanDefinition);
    }
}
```

bean的加载方式（八）

- 导入实现了BeanDefinitionRegistryPostProcessor接口的类，通过BeanDefinition的注册器注册实名bean，实现对容器中bean的最终裁定

```
public class MyPostProcessor implements BeanDefinitionRegistryPostProcessor {
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {
        BeanDefinition beanDefinition = BeanDefinitionBuilder
                                        .rootBeanDefinition(BookServiceImpl4.class)
                                        .getBeanDefinition();
        registry.registerBeanDefinition("bookService", beanDefinition);
    }
}
```



bean的加载控制

- xml+<bean/>
- xml:context+注解（@Component+4个@Bean）
- 配置类+扫描+注解（@Component+4个@Bean）
  - @Bean定义FactoryBean接口
  - @ImportResource
  - @Configuration注解的proxyBeanMethods属性
- @Import导入bean的类
- @Import导入配置类
- AnnotationConfigApplicationContext调用register方法
- @Import导入ImportSelector接口
- @Import导入ImportBeanDefinitionRegistrar接口
- @Import导入BeanDefinitionRegistryPostProcessor接口

bean的加载控制

- xml+<bean/>
- xml:context+注解（@Component+4个@Bean）
- 配置类+扫描+注解（@Component+4个@Bean）
  - @Bean定义FactoryBean接口
  - @ImportResource
  - @Configuration注解的proxyBeanMethods属性
- @Import导入bean的类
- @Import导入配置类
- **AnnotationConfigApplicationContext调用register方法**
- **@Import导入ImportSelector接口**
- **@Import导入ImportBeanDefinitionRegistrar接口**
- **@Import导入BeanDefinitionRegistryPostProcessor接口**

bean的加载控制

- 根据任意条件确认是否加载bean

```
public class MyImportSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        try {
            Class<?> clazz = Class.forName("com.itheima.ebean.Mouse");
            if(clazz != null) {
                return new String[]{"com.itheima.bean.Cat"};
            }
        } catch (ClassNotFoundException e) {
            return new String[0];
        }
        return null;
    }
}
```

bean的加载控制

- 使用@Conditional注解的派生注解设置各种组合条件控制bean的加载

bean的加载控制

- 匹配指定类

```
public class SpringConfig {
    @Bean
    @ConditionalOnClass(Mouse.class)
    public Cat tom(){
        return new Cat();
    }
}
```

bean的加载控制

- 未匹配指定类

```
public class SpringConfig {
    @Bean
    @ConditionalOnClass(Mouse.class)
    @ConditionalOnMissingClass("com.itheima.bean.Wolf")
    public Cat tom(){
        return new Cat();
    }
}
```

bean的加载控制

- 匹配指定类型的bean

```
@Import(Mouse.class)
public class SpringConfig {
    @Bean
    @ConditionalOnBean(Mouse.class)
    public Cat tom(){
        return new Cat();
    }
}
```

bean的加载控制

- 匹配指定名称的bean

```
@Import(Mouse.class)
public class SpringConfig {
    @Bean
    @ConditionalOnBean(name="com.itheima.bean.Mouse")
    public Cat tom(){
        return new Cat();
    }
}
```

bean的加载控制

- 匹配指定名称的bean

```
@Import(Mouse.class)
public class SpringConfig {
    @Bean
    @ConditionalOnBean(name="jerry")
    public Cat tom(){
        return new Cat();
    }
}
```

bean的加载控制

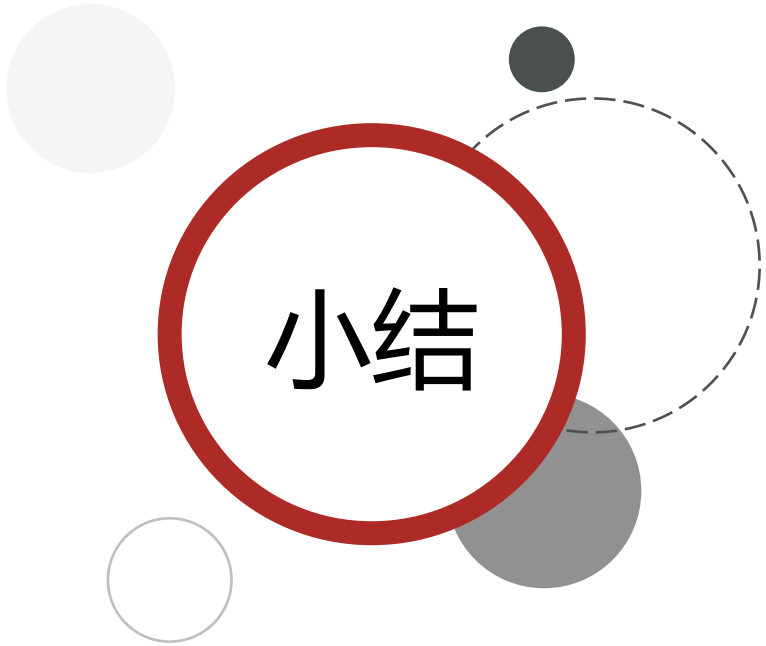
- 匹配指定环境

```
@Configuration
@Import(Mouse.class)
public class MyConfig {
    @Bean
    @ConditionalOnClass(Mouse.class)
    @ConditionalOnMissingClass("com.itheima.bean.Dog")
    @ConditionalOnNotWebApplication
    public Cat tom(){
        return new Cat();
    }
}
```

bean的加载控制

- 匹配指定环境

```
public class SpringConfig {
    @Bean
    @ConditionalOnClass(name = "com.mysql.jdbc.Driver")
    public DruidDataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        return ds;
    }
}
```



1. 使用@ConditionalOn***注解为bean的加载设置条件

bean依赖的属性配置

- 将业务功能bean运行需要的资源抽取成独立的属性类（******Properties），设置读取配置文件信息

```
@ConfigurationProperties(prefix = "cartoon")
public class CartoonProperties {
    private Cat cat;
    private Mouse mouse;
    public Cat getCat() {
        return cat;
    }
    public void setCat(Cat cat) {
        this.cat = cat;
    }
    public Mouse getMouse() {
        return mouse;
    }
    public void setMouse(Mouse mouse) {
        this.mouse = mouse;
    }
}
```

bean依赖的属性配置

- 配置文件中使用固定格式为属性类注入数据

```
cartoon:
  cat:
    name: "图多盖洛"
    age: 5
  mouse:
    name: "泰菲"
    age: 1
```

bean依赖的属性配置

- 定义业务功能bean，通常使用@Import导入，解耦强制加载bean

```
@Component
public class CartoonCatAndMouse {
    private Cat cat;
    private Mouse mouse;
    public void play(){
        System.out.println(cat.getAge()+"岁的"+cat.getName()+
                "与"+mouse.getAge()+"岁的"+mouse.getName()+"打起来了");
    }
}
```

bean依赖的属性配置

- 使用@EnableConfigurationProperties注解设定使用属性类时加载bean

```
@Component
@EnableConfigurationProperties(CartoonProperties.class)
public class CartoonCatAndMouse {
    private CartoonProperties cartoonProperties;
    public CartoonCatAndMouse(CartoonProperties cartoonProperties){
        this.cartoonProperties = cartoonProperties;
        cat = new Cat();
        cat.setName(cartoonProperties.getCat()!=null &&
                StringUtils.hasText(cartoonProperties.getCat().getName())?
                cartoonProperties.getCat().getName():"tom");
}
```

1．业务bean的属性可以为其设定默认值

2．当需要设置时通过配置文件传递属性

3．业务bean应尽量避免设置强制加载，而是根据需要导入后加载，降低spring容器管理bean的强度

自动配置原理

懒

自动配置原理

1. 收集Spring开发者的编程习惯，整理开发过程使用的常用技术列表—>(**技术集A**)
2. 收集常用技术(技术集A)的使用参数，整理开发过程中每个技术的常用设置列表—>(**设置集B**)
3. 初始化SpringBoot基础环境，加载用户自定义的bean和导入的其他坐标，形成**初始化环境**
4. 将**技术集A**包含的所有技术都定义出来，在Spring/SpringBoot启动时默认全部加载
5. 将**技术集A**中具有使用条件的技术约定出来，设置成按条件加载，由开发者决定是否使用该技术（与**初始化环境**比对）
6. 将**设置集B**作为默认配置加载（约定大于配置），减少开发者配置工作量
7. 开放**设置集B**的配置覆盖接口，由开发者根据自身需要决定是否覆盖默认配置

自动配置原理

1. 收集Spring开发者的编程习惯，整理开发过程使用的常用技术列表—>(**技术集A**)
2. 收集常用技术(技术集A)的使用参数，整理开发过程中每个技术的常用设置列表—>(**设置集B**)
3. 初始化SpringBoot基础环境，加载用户自定义的bean和导入的其他坐标，形成**初始化环境**
4. 将**技术集A**包含的所有技术都定义出来，在Spring/SpringBoot启动时默认全部加载
5. 将**技术集A**中具有使用条件的技术约定出来，设置成按条件加载，由开发者决定是否使用该技术（与**初始化环境**比对）
6. 将**设置集B**作为默认配置加载（约定大于配置），减少开发者配置工作量
7.

```
public final class SpringFactoriesLoader {
    public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";
}
```

自动配置原理

1. 收集Spring开发者的编程习惯，整理开发过程使用的常用技术列表—>(**技术集A**)
2. 收集常用技术(技术集A)的使用参数，整理开发过程中每个技术的常用设置列表—>(**设置集B**)
3. 初始化SpringBoot基础环境，加载用户自定义的bean和导入的其他坐标，形成**初始化环境**
4. 将**技术集A**包含的所有技术都定义出来，在Spring/SpringBoot启动时默认全部加载
5. 将**技术集A**中具有使用条件的技术约定出来，设置成按条件加载，由开发者决定是否使用该技术（与**初始化环境**比对）
6. 将**设置集B**作为默认配置加载（约定大于配置），减少开发者配置工作量
7.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>2.5.4</version>
</dependency>
```

自动配置原理

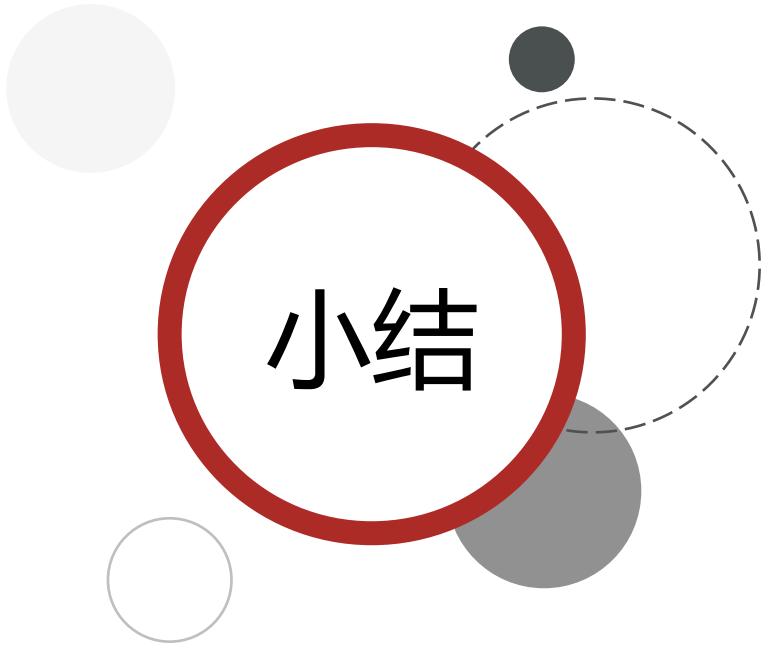
1. 收集Spring开发者的编程习惯，整理开发过程使用的常用技术列表—>(**技术集A**)
2. 收集常用技术(技术集A)的使用参数，整理开发过程中每个技术的常用设置列表—>(**设置集B**)
3. 初始化SpringBoot基础环境，加载用户自定义的bean和导入的其他坐标，形成**初始化环境**
4. 将**技术集A**包含的所有技术都定义出来，在Spring/SpringBoot启动时默认全部加载
5. 将**技术集A**中具有使用条件的技术约定出来，设置成按条件加载，由开发者决定是否使用该技术（与**初始化环境**比对）
6. 将**设置集B**作为默认配置加载（约定大于配置），减少开发者配置工作量
7.

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(RedisOperations.class)
@EnableConfigurationProperties(RedisProperties.class)
@Import({ LettuceConnectionConfiguration.class, JedisConnectionConfiguration.class })
public class RedisAutoConfiguration {
}
```

自动配置原理

1. 收集Spring开发者的编程习惯，整理开发过程使用的常用技术列表—>(**技术集A**)
2. 收集常用技术(技术集A)的使用参数，整理开发过程中每个技术的常用设置列表—>(**设置集B**)
3. 初始化SpringBoot基础环境，加载用户自定义的bean和导入的其他坐标，形成**初始化环境**
4. 将**技术集A**包含的所有技术都定义出来，在Spring/SpringBoot启动时默认全部加载
5. 将**技术集A**中具有使用条件的技术约定出来，设置成按条件加载，由开发者决定是否使用该技术（与**初始化环境**比对）
6. 将**设置集B**作为默认配置加载（约定大于配置），减少开发者配置工作量
7.

```java
@ConfigurationProperties(prefix = "spring.redis")
public class RedisProperties {
    private String url;
    private String host = "localhost";
    private int port = 6379;
}
```



1．先开发若干种技术的标准实现

2．SpringBoot启动时加载所有的技术实现对应的自动配置类

3．检测每个配置类的加载条件是否满足并进行对应的初始化

4．切记是先加载所有的外部资源，然后根据外部资源进行条件比对

变更自动配置

- 自定义自动配置（META-INF/spring.factories）

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.itheima.bean.CartoonCatAndMouse
```

- 控制SpringBoot内置自动配置类加载

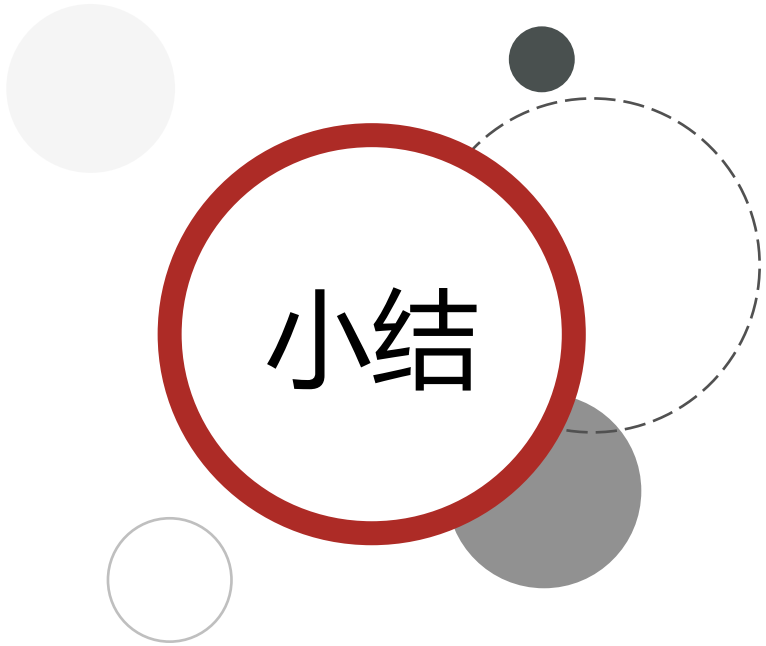
```
spring:
  autoconfigure:
    exclude:
      - org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration
      - org.springframework.boot.autoconfigure.context.LifecycleAutoConfiguration
```

```
@EnableAutoConfiguration(excludeName = "",exclude = {})
```

变更自动配置

- 变更自动配置：去除tomcat自动配置（条件激活），添加jetty自动配置（条件激活）

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <!--web起步依赖环境中，排除Tomcat起步依赖，匹配自动配置条件-->
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <!--添加Jetty起步依赖，匹配自动配置条件-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
    </dependency>
</dependencies>
```

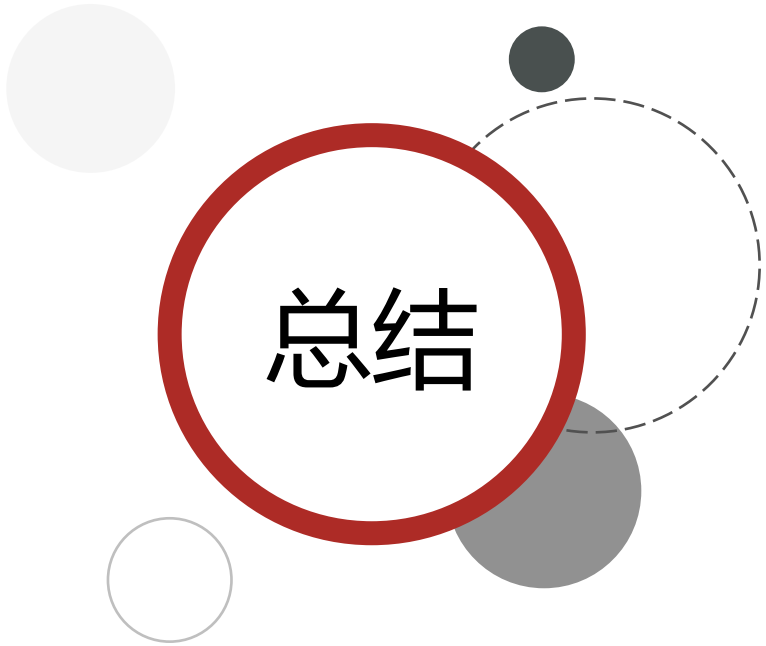


1．通过配置文件exclude属性排除自动配置

2．通过注解@EnableAutoConfiguration属性排除自动配置项

3．启用自动配置只需要满足自动配置条件即可

4．可以根据需求开发自定义自动配置项



1. bean加载方式（8+）
2. bean加载控制（编程 & 注解）
3. bean依赖属性配置（Properties）
4. 自动配置原理
5. 变更系统自动配置（配置文件、注解属性）
6. 添加自定义自动配置（META-INF/spring.factories）

自定义starter

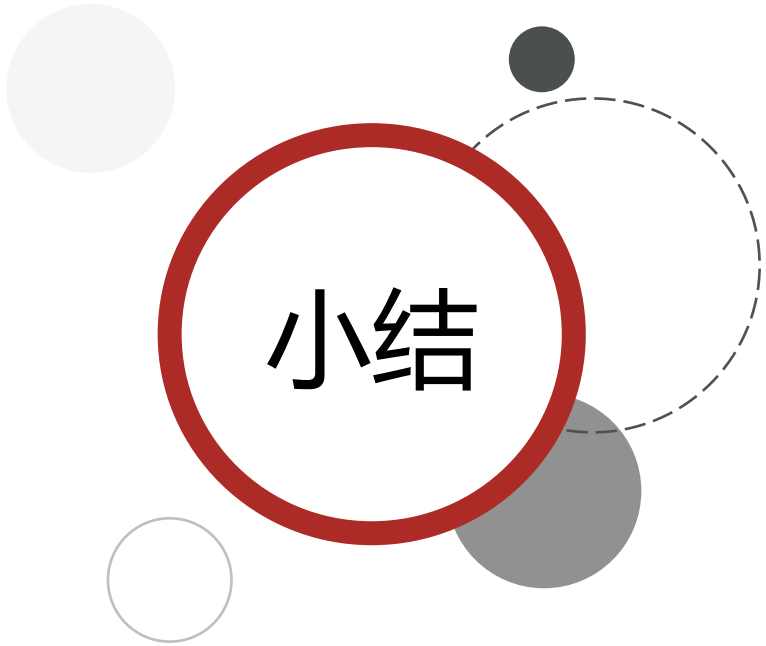
- 案例：统计独立IP访问次数
- 自定义starter
- 辅助功能开发

案例：记录系统访客独立IP访问次数

1. 每次访问网站行为均进行统计
2. 后台每10秒输出一次监控信息（格式：IP+访问次数）

案例：记录系统访客独立IP访问次数（需求分析）

1. 数据记录位置：Map / Redis
2. 功能触发位置：每次web请求（拦截器）
    ① 步骤一：降低难度，主动调用，仅统计单一操作访问次数（例如查询）
    ② 步骤二：开发拦截器
3. 业务参数（配置项）
    ① 输出频度，默认10秒
    ② 数据特征：累计数据 / 阶段数据，默认累计数据
    ③ 输出格式：详细模式 / 极简模式
4. 校验环境，设置加载条件



1. 案例：记录系统访客独立IP访问次数

自定义starter

- 业务功能开发

```
public class IpCountService {
    //计数集合
    private Map<String,Integer> ipCountMap = new HashMap<String,Integer>();
    @Autowired
    private HttpServletRequest request;
    public void count(){
        String ipAddress = request.getRemoteAddr();
        if(ipCountMap.containsKey(ipAddress)){
            ipCountMap.put(ipAddress,ipCountMap.get(ipAddress) + 1);
        }else{
            ipCountMap.put(ipAddress,1);
        }
        System.out.println("IP:"+ipAddress);
    }
}
```

自定义starter

- 自动配置类

```
public class IpAutoConfiguration {
    @Bean
    public IpCountService ipCountService(){
        return new IpCountService();
    }
}
```

自定义starter

- 配置

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.itheima.ip.autoconfigure.IpAutoConfiguration
```

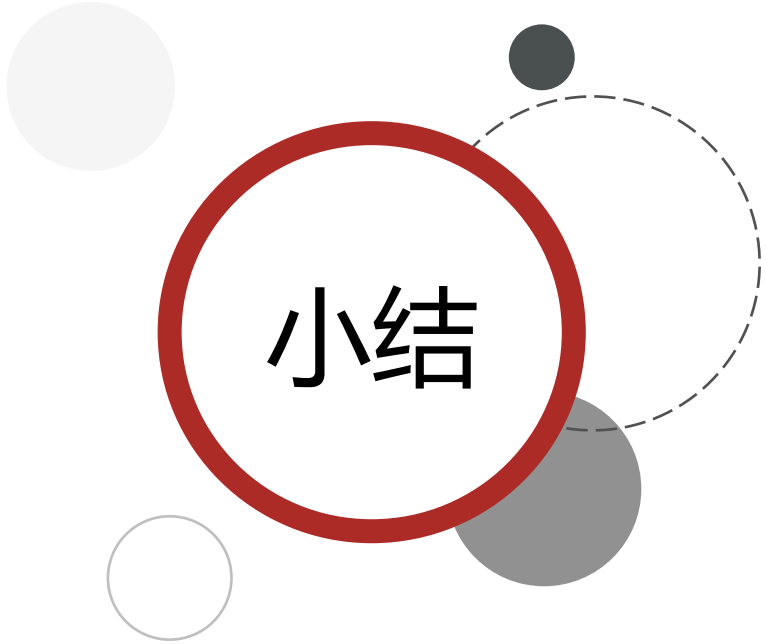
自定义starter

- 模拟调用（非最终版）

```
@RestController
@RequestMapping("/books")
public class BookController {

    @Autowired
    private IpCountService ipCountService;

    @GetMapping("{currentPage}/{pageSize}")
    public R getPage(@PathVariable int currentPage,@PathVariable int pageSize,Book book){
        // TODO 追加ip访问统计
        ipCountService.count();
        IPage<Book> page = bookService.getPage(currentPage, pageSize,book);
        if( currentPage > page.getPages()){
            page = bookService.getPage((int)page.getPages(), pageSize,book);
        }
        return new R(true, page);
    }
}
```



1．使用自动配置加载业务功能

2．切记使用之前先clean后install安装到maven仓库，确保资源更新

自定义starter

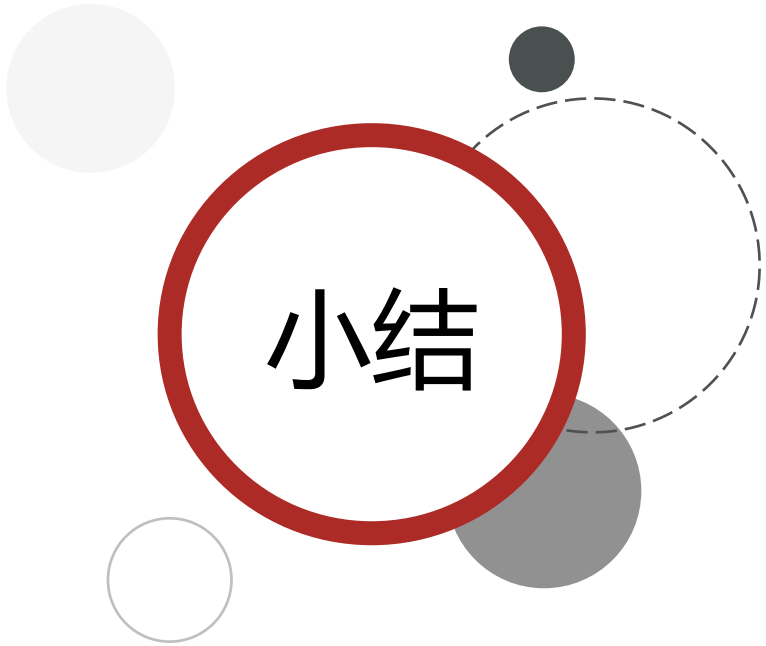
- 开启定时任务功能

```
@EnableScheduling
public class IpAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public IpCountService ipCountService(){
        return new IpCountService();
    }
}
```

自定义starter

- 设置定时任务

```
public class IpCountService {
    //计数集合
    private Map<String,Integer> ipCountMap = new HashMap<String,Integer>();
    @Scheduled(cron = "0/10 * * * * ?")
    public void print(){
        System.out.println("        IP访问监控");
        System.out.println("+-----ip-address-----+--num--+");
        for(Map.Entry<String,Integer> info :ipCountMap.entrySet()){
            String key = info.getKey();
            Integer count = info.getValue();
            String lineInfo = String.format("|%18s  |%6d |",key,count);
            System.out.println(lineInfo);
        }
        System.out.println("+--------------------+-------+");
    }
}
```



1. 完成业务功能定时显示报表

2. String.format()

自定义starter

- 定义属性类，加载对应属性

```
@ConfigurationProperties(prefix = "tools.ip")
public class IpProperties {
    /**  日志显示周期  */
    private long cycle = 10L;
    /**  是否周期内重置数据  */
    private Boolean cycleReset = false;
    /** 日志输出模式  detail:明细模式  simple:极简模式  */
    private String model = LogModel.DETAIL.value;
    public enum LogModel {
        DETAIL("detail"),
        SIMPLE("simple");
        private String value;
        private LogModel(String value) { this.value = value; }
        public String getValue() { return value; }
    }
}
```

自定义starter

- 设置加载Properties类为bean

```
@EnableConfigurationProperties(IpProperties.class)
@EnableScheduling
public class IpAutoConfiguration {
    @Bean
    public IpCountService ipCountService(){
        return new IpCountService();
    }
}
```

自定义starter

- 根据配置切换设置

```
public class IpCountService {
    @Autowired
    private IpProperties ipProperties;
    @Scheduled(cron = "0/10 * * * * ?")
    public void print(){
        //模式切换
        if(ipProperties.getMode().equals(IpProperties.LogModel.DETAIL.getValue())){
            //明细模式
        }else if(ipProperties.getMode().equals(IpProperties.LogModel.SIMPLE.getValue())){
            //极简模式
        }
        //周期内重置数据
        if(ipProperties.getCycleReset()){
            ipCountMap.clear();
        }
    }
}
```

自定义starter

- 明细模式报表模板

```
//明细模式
System.out.println("          IP访问监控");
System.out.println("+-----ip-address-----+--num--+");
for(Map.Entry<String,Integer> info :ipCountMap.entrySet()){
    String lineInfo = String.format("|%18s  |%6d |", info.getKey(), info.getValue());
    System.out.println(lineInfo);
}
System.out.println("+--------------------+-------+");
```

自定义starter

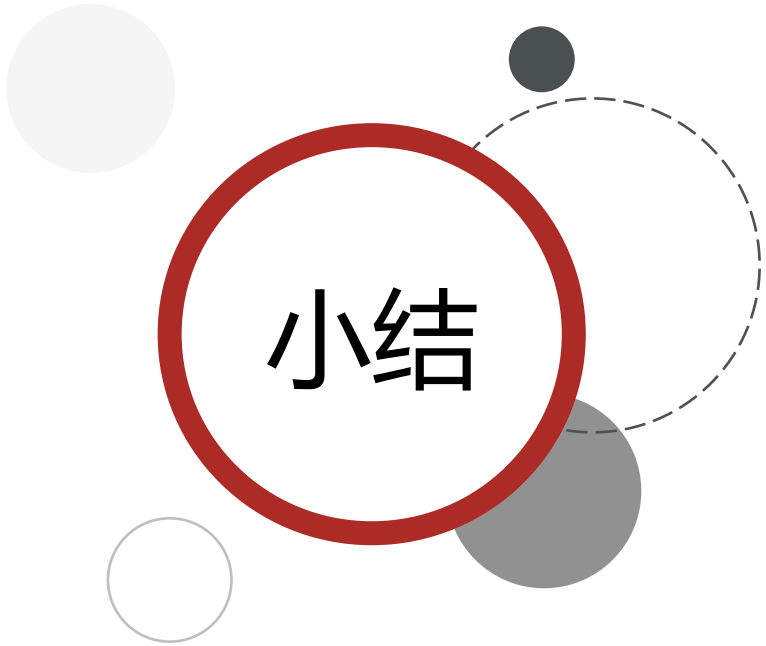
- 极简模式报表模板

```
//极简模式
System.out.println("         IP访问监控");
System.out.println("+-----ip-address-----+");
for(Map.Entry<String,Integer> info :ipCountMap.entrySet()){
    String lineInfo = String.format("|%18s  |", info.getKey());
    System.out.println(lineInfo);
}
System.out.println("+--------------------+");
```

自定义starter

- 配置信息

```
tools:
  ip:
    cycle-reset: false
    mode: detail
```



1．使用属性修改自动配置加载的设置值

自定义starter

- 配置信息

```
tools:
  ip:
    cycle: 2
    cycle-reset: false
    mode: detail
```

自定义starter

- 自定义bean名称

```
@Component("ipProperties")
@ConfigurationProperties(prefix = "tools.ip")
public class IpProperties {
}
```

自定义starter

- 放弃配置属性创建bean方式，改为手工控制

```
//@EnableConfigurationProperties(IpProperties.class)
@EnableScheduling
@Import(IpProperties.class)
public class IpAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public IpCountService ipCountService(){
        return new IpCountService();
    }
}
```

自定义starter

- 使用#{beanName.attrName}读取bean的属性

```
@Scheduled(cron = "0/#{ipProperties.cycle} * * * * ?")
public void print(){
}
```

1．配置调整

自定义starter

- 自定义拦截器

```
public class IpInterceptor implements HandlerInterceptor {
    @Autowired
    private IpCountService ipCountService;
    @Override
    public boolean preHandle(    HttpServletRequest request,
                                 HttpServletResponse response,
                                 Object handler) throws Exception {
        ipCountService.count();
        return true;
    }
}
```

自定义starter

- 设置核心配置类，加载拦截器

```
@Configuration
public class SpringMvcConfig implements WebMvcConfigurer {
    @Bean
    public IpInterceptor ipInterceptor(){
        return new IpInterceptor();
    }
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(ipInterceptor()).addPathPatterns("/**");
    }
}
```

1．拦截器开发

辅助功能开发

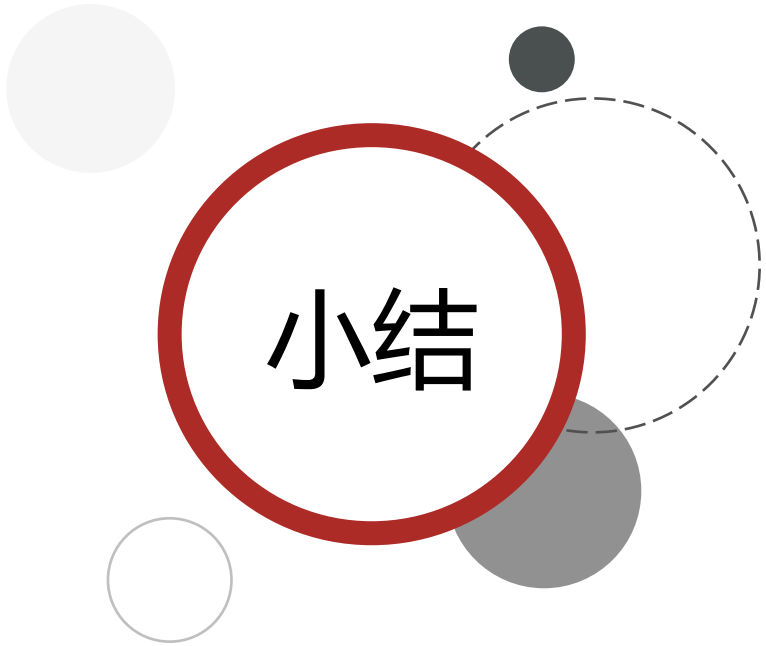
- 导入配置处理器坐标

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

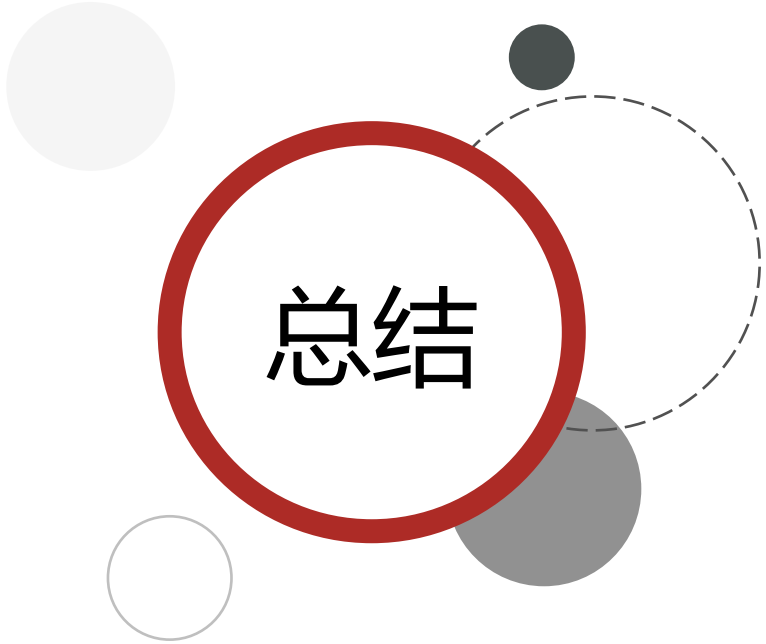
辅助功能开发

- 进行自定义提示功能开发

```
"hints": [
  {
    "name": "tools.ip.mode",
    "values": [
      {
        "value": "detail",
        "description": "明细模式."
      },
      {
        "value": "simple",
        "description": "极简模式."
      }
    ]
  }
]
```



1．yml提示功能开发



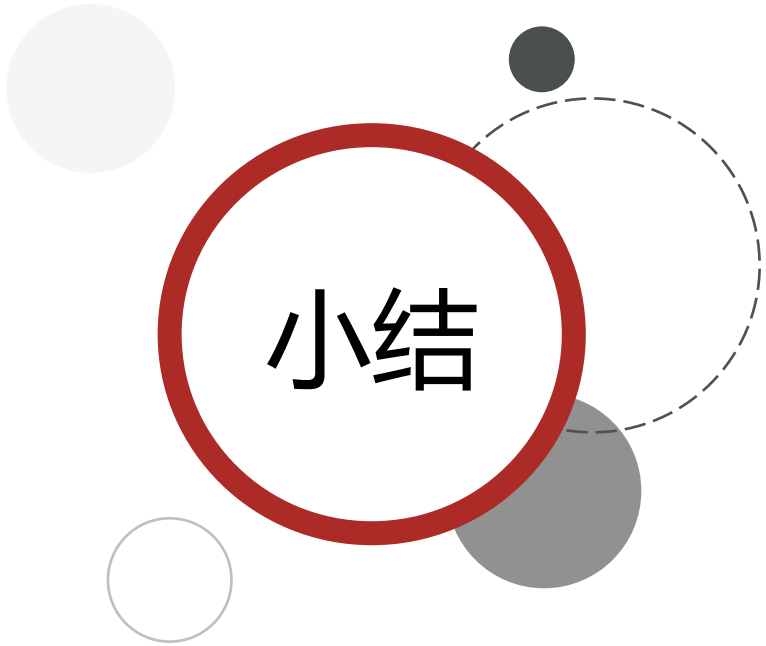
1. 案例：统计独立IP访问次数
2. 自定义starter
3. yml提示功能

核心原理

- SpringBoot启动流程
- 容器类型选择
- 监听器

SpringBoot启动流程

1. 初始化各种属性，加载成对象
   - 读取环境属性（Environment）
   - 系统配置（spring.factories）
   - 参数（Arguments、application.properties）
2. 创建Spring容器对象ApplicationContext，加载各种配置
3. 在容器创建前，通过监听器机制，应对不同阶段加载数据、更新数据的需求
4. 容器初始化过程中追加各种功能，例如统计时间、输出日志等



1．初始化数据

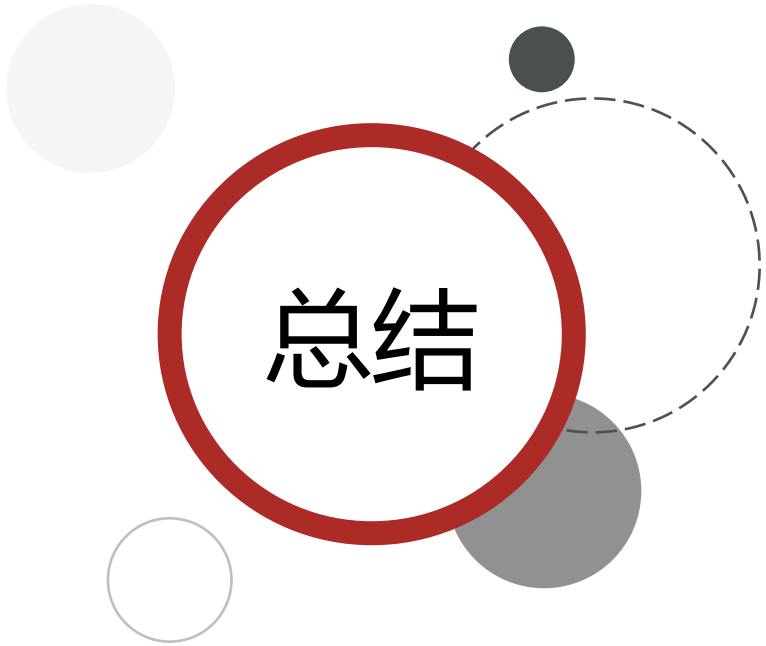
2．创建容器

SpringBoot启动流程

1. 初始化各种属性，加载成对象
   - 读取环境属性（Environment）
   - 系统配置（spring.factories）
   - 参数（Arguments、application.properties）
2. 创建Spring容器对象ApplicationContext，加载各种配置
3. 在容器创建前，通过监听器机制，应对不同阶段加载数据、更新数据的需求
4. 容器初始化过程中追加各种功能，例如统计时间、输出日志等

监听器类型

1. 在应用运行但未进行任何处理时，将发送 ApplicationStartingEvent。
2. 当Environment被使用，且上下文创建之前，将发送 ApplicationEnvironmentPreparedEvent。
3. 在开始刷新之前，bean定义被加载之后发送 ApplicationPreparedEvent。
4. 在上下文刷新之后且所有的应用和命令行运行器被调用之前发送 ApplicationStartedEvent。
5. 在应用程序和命令行运行器被调用之后，将发出 ApplicationReadyEvent，用于通知应用已经准备处理请求。
6. 启动时发生异常，将发送 ApplicationFailedEvent。



1．理解过程有助于思考

传智教育旗下高端IT教育品牌