

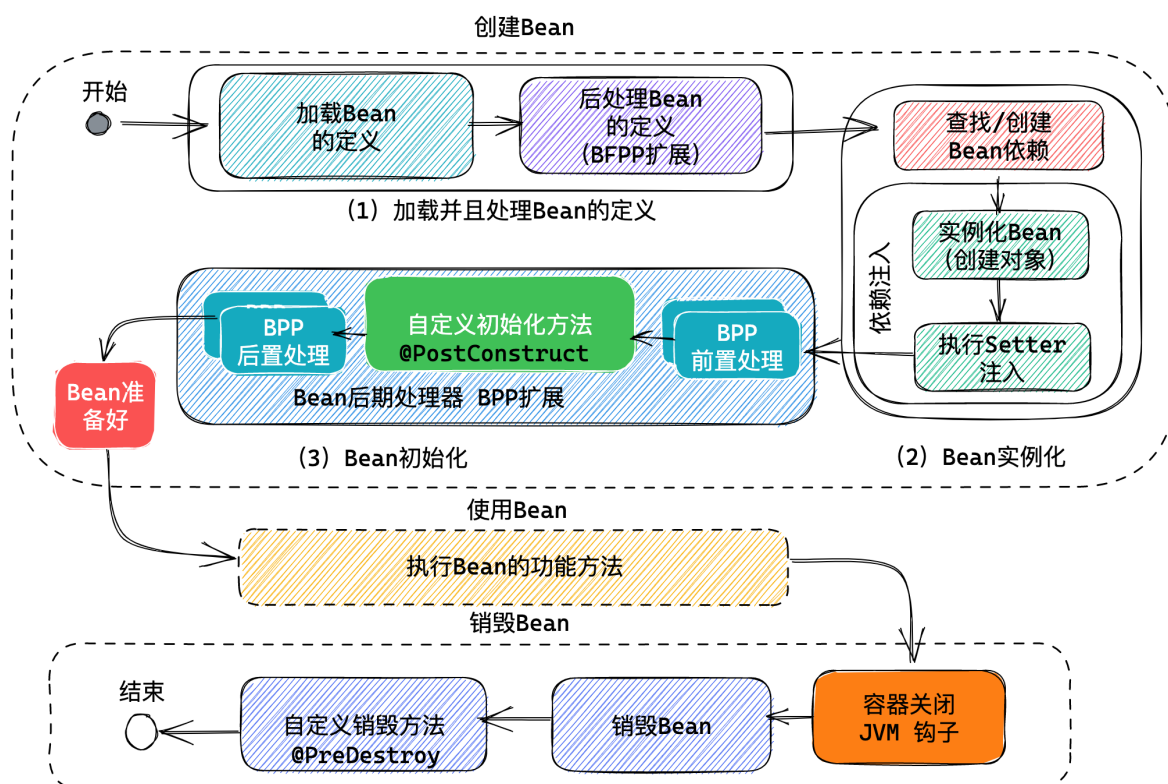
Spring 生命周期 和 Spring Boot

今日内容:

1. Spring Bean 生命周期管理，也就是Spring中Bean的创建、使用以及销毁的过程，经典面试题目！
2. Spring Boot 以及自动配置原理，也是经典面试题目

生命周期管理 life-cycle

本次课程的目标就是研究Spring的完整Bean生命周期管理过程。



自定义的初始化和销毁方法

Spring 为了方便用户扩展功能，提供了在Bean生命周期管理过程中自动调用的声明周期管理方法功能：

- **@PostConstruct**: 在对象实例化后，Bean初始化过程中调用用户自定义的方法
 - 标注在方法上，要求该方法无参且返回值为void
 - 可以用于初始化: 如初始化缓存，初始化数据库连接
- **@PreDestroy** 在销毁Bean时候，执行用户自定义的方法
 - 标注在方法上，要求该方法无参且返回值为void
 - 清空缓存，释放资源
 - 关闭容器时候会自动调用
 - Spring会在JVM上挂“钩子”，关闭JVM时候，钩子会自动调用Spring容器的关闭方法, Spring容器关闭时候，会自动实现 **@PreDestroy** 标注的方法。
 - 这个方法不是绝对可靠，直接关闭进程时候不会执行
 - 只有单例对象，销毁时候才会执行销毁方法

@PreDestroy 的执行原理：Spring 在虚拟机上挂了关闭钩子，虚拟机关闭时候会自动执行钩子，Spring的钩子会关闭Spring容器，关闭容器时候会自动执行@PreDestroy 标注的方法。

案例：

```
/**
 * 标签服务
 * concurrent 并发
 */
@Component
public class TagService {
    Logger logger = LoggerFactory.getLogger(TagService.class);
    /**
     * 本地缓存，缓存了标签信息
     */
    private final CopyOnWriteArrayList<String> tags = new CopyOnWriteArrayList<>();

    @PostConstruct //创建对象以后调用
    public void initTags() {
        tags.add("应季");
        tags.add("爆款");
        tags.add("进口");
        tags.add("生鲜");
        tags.add("健身");
        logger.debug("初始化标签{}", tags);
    }

    public List<String> getTags() {
        return tags;
    }

    @PreDestroy //销毁对象之前调用
    public void destroy() {
        logger.debug("清空标签{}", tags);
        tags.clear();
    }
}
```

测试案例：

```
@SpringBootTest
public class TagServiceTests {

    Logger logger = LoggerFactory.getLogger(TagServiceTests.class);

    @Autowired
    TagService tagService;

    @Test
    void tests(){
        List<String> tags = tagService.getTags();
        tags.forEach(tag->logger.debug("{} ", tag));
    }
}
```

关闭钩子案例:

```
package cn.tedu.spring.hook;

public class ShutdownHookDemo {
    public static void main(String[] args) throws Exception{
        /*
         * runtime 运行时
         * total 总数
         * Memory 记忆, 内存, 记忆体
         * shutdown 关机
         * Hook 钩子
         * runtime 代表正在运行的虚拟机
         * 可以通过runtime获取当前虚拟机的参数
         */
        Runtime runtime = Runtime.getRuntime();
        //获取当前JVM总内存数量
        long bytes = runtime.totalMemory();
        System.out.println("totalMemory:" + bytes);
        //给系统挂关闭钩子
        runtime.addShutdownHook(new DemoHook());
        System.out.println("挂完了");
        Thread.sleep(2000);
    }
}

class DemoHook extends Thread{
    @Override
    public void run() {
        System.out.println("执行钩子了!");
    }
}
```

在Spring中挂钩子:

```
package cn.tedu.spring.hook;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

/**
 * 在当前虚拟机上挂关闭钩子
 */
@Component
public class HockBean extends Thread {

    Logger logger = LoggerFactory.getLogger(HockBean.class);

    @PostConstruct
    void addHock(){
        Runtime.getRuntime().addShutdownHook(this);
    }
}
```

```

        logger.debug("在当前虚拟机上挂关闭钩子");
    }

    @Override
    public void run() {
        logger.debug("系统关闭! ");
    }
}

```

@Bean Bean的生命周期管理方法

- @Bean 注解使用属性设置销毁(destroy)方法
 - @Bean(initMethod="init", destroyMethod="destroy")
- 可以根据实际业务需要执行这两个方法

案例：Bean 组件

```

public class CategoryService {
    Logger logger = LoggerFactory.getLogger(CategoryService.class);
    /**
     * 分类缓存
     */
    private CopyOnWriteArrayList<Category> categoryList;

    public void init() {
        categoryList = new CopyOnWriteArrayList<>();
        categoryList.add(new Category("1", "家电"));
        categoryList.add(new Category("2", "食品"));
        categoryList.add(new Category("2", "服装"));
        logger.debug("初始化分类{}", categoryList);
    }

    public List<Category> getCategoryList() {
        return categoryList;
    }

    public void destroy(){
        logger.debug("销毁 {}", categoryList);
        categoryList.clear();
    }
}

```

配置类：

```

@Configuration
public class ServiceConfig {

    /**
     * @Bean 的属性调用生命周期管理方法
     * initMethod 初始化方法 等同与 @PostConstruct
     * destroyMethod 销毁方法 等同与 @PreDestroy
     * initMethod="初始化方法名"
     * destroyMethod="销毁方法名"
     */
}

```

```

@Bean(initMethod = "init", destroyMethod = "destroy")
public CategoryService categoryService() {
    return new CategoryService();
}
}

```

测试类:

```

package cn.tedu.spring;

import cn.tedu.spring.entity.Category;
import cn.tedu.spring.service.CategoryService;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
public class CategoryServiceTests {

    Logger logger = LoggerFactory.getLogger(CategoryServiceTests.class);

    @Autowired
    CategoryService categoryService;

    @Test
    void test(){
        List<Category> list = categoryService.getCategoryList();
        list.forEach(category->logger.debug("{} ", category));
    }
}

```

关于 在Bean上标注了@Scope("prototype")

- 是“原型”范围，会创建多个实例，每次使用bean时候都会创建一个新的Bean对象
 - getBean 和 注入Bean
- 如果不使用Bean，则不创建对象
- 创建对象时候会自动调用 @PostConstruct 方法
- 多个实例时候因为对象太多，Spring将不再管理销毁方法，也就关闭Spring时候不会调用 @PreDestroy方法
- 请自行销毁对象（设置引用为空），由GC销毁

案例:

```

package cn.tedu.spring.service;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

```

```

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

@Component
@Scope("prototype")
public class NameService {
    Logger logger = LoggerFactory.getLogger(NameService.class);
    /**
     * 本地缓存，缓存了名字信息
     */
    private CopyOnWriteArrayList<String> names = new CopyOnWriteArrayList<>();

    @PostConstruct
    public void init(){
        names.add("Tom");
        names.add("Jerry");
        names.add("Andy");
        logger.debug("初始化 {}", names);
    }

    public List<String> getNames() {
        return names;
    }

    /**
     * Spring不会调用“Prototype”组件的销毁方法
     */
    @PreDestroy
    public void destroy(){
        logger.debug("销毁 {}", names);
        names.clear();
    }
}

```

测试:

```

package cn.tedu.spring;

import cn.tedu.spring.service.NameService;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
public class NameServiceTests {

```

```

Logger logger = LoggerFactory.getLogger(NameServiceTests.class);

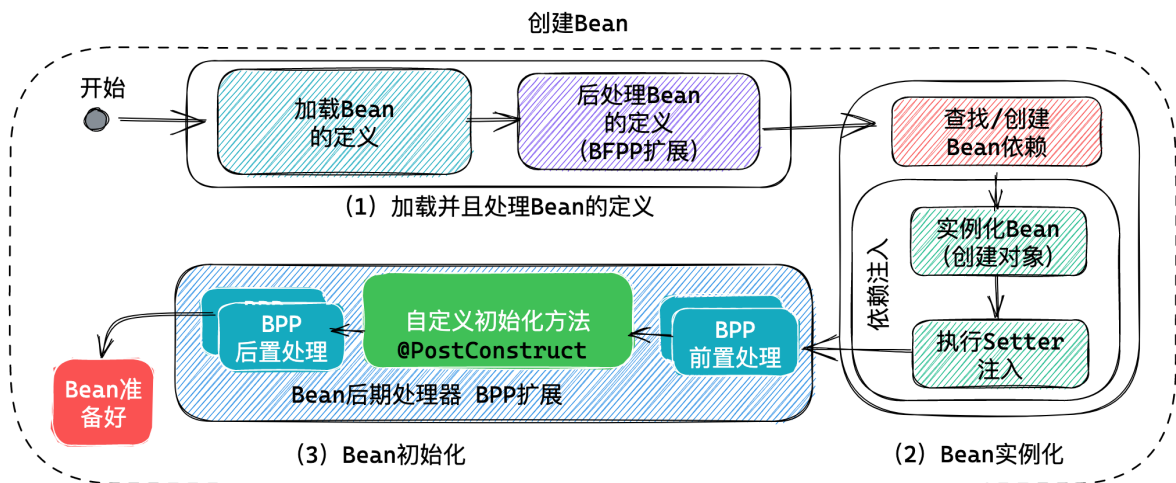
@Autowired
NameService nameService;

@Test
void tests(){
    List<String> names = nameService.getNames();
    names.forEach(name->logger.debug("{} ", name));
}
}

```

Bean的创建步骤

Bean的创建有3大步



1. 加载Bean的定义，加载Bean定义之后，并不会立即创建Bean对象！

1. Spring 启动后，根据配置文件加载Bean的定义，包括处理@Bean 和 @Component 注解
2. 找到Bean定义后，将转换应用于Bean定义，也就是可以进一步修改处理Bean定义
3. 处理Bean定义时候调用了一系列实现 BeanFactoryPostProcessor 接口的对象。
4. 可以自行扩展 BeanFactoryPostProcessor 接口，参与Bean后期处理功能。
5. 其中包括处理 @PropertySource、@Value 的 PropertySourcesPlaceholderConfigurer
 - 参考：<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/PropertySourcesPlaceholderConfigurer.html>
6. 其中包括处理 @Configuration 的 ConfigurationClassPostProcessor
 - 参考：<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/ConfigurationClassPostProcessor.html>

2. 实例化Bean对象，这一步才开始创建Bean对象

1. 首先查找Bean的依赖关系，解决创建Bean的先后次序问题
2. 实例化Bean对象，也就行创建Bean对象，这里包含构造器注入过程。
3. 然后进行属性注入

3. Bean的初始化（这里是指对创建Bean以后的处理过程）

1. Bean的初始化由一系列的BeanPostProcessor对象完成
2. 先执行 BPP 的前置处理方法

3. 然后执行Bean的自定义初始化方法
4. 再执行BPP的后置处理方法
5. 这个步骤可以干预扩展，可以自行实现BeanPostProcessor
6. AOP代理就在在这一步添加的DefaultAdvisorAutoProxyCreator
AspectJAwareAdvisorAutoProxyCreator
7. AspectJ 的注解在AnnotationAwareAspectJAutoProxyCreator 处理 是一个BPP实现类!
 - 参考: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/aop/framework/autoproxy/AbstractAdvisorAutoProxyCreator.html>

BeanFactoryPostProcessor 案例:

```
/**
 * BeanFactoryPostProcessor Bean工厂后期处理器，
 * 其中使用一个方法 postProcessBeanFactory Bean工厂后置处理
 * 在加载了Bean定义信息以后，对Bean定义进行后续转换处理。
 * 原则上不建议自行处理
 * configurableListableBeanFactory 对象中封装了全部的 Bean定义信息
 */
@Component
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
    Logger logger = LoggerFactory.getLogger(MyBeanFactoryPostProcessor.class);

    /**
     * postProcessBeanFactory 在Spring加载全部Bean定义以后执行
     * @param configurableListableBeanFactory 这个对象封装了全部Bean定义信息
     * @throws BeansException bean定义加载失败抛出异常，如果抛出异常，则Spring初始化失败!
     */
    @Override
    public void postProcessBeanFactory(
        ConfigurableListableBeanFactory configurableListableBeanFactory)
        throws BeansException {
        //获取全部的Bean定义， getBeanDefinitionNames获取Bean定义的名称
        String[] beanNames =
            configurableListableBeanFactory.getBeanDefinitionNames();
        for (String name : beanNames) {
            logger.debug("bean name: {}", name);
        }
    }
}
```

BeanPostProcessor 案例:

```
/**
 * Post 后期
 * Process 处理
 * Processor 处理器
 * Before 之前
 * After 之后
 * Initialization: 初始化， 这里是指Bean的初始化过程
 * BeanPostProcessor Bean后期处理器(BPP)， 在创建了Bean对象之后对Bean进行处理
 * 其中包括两个方法
```



```

* postProcessBeforeInitialization: bean后期处理，在初始化之前执行
* postProcessAfterInitialization: bean后期处理，在初始化之后执行
*/
@Component
public class MyBeanPostProcessor implements BeanPostProcessor {
    Logger logger = LoggerFactory.getLogger(MyBeanPostProcessor.class);
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        //创建了一个bean以后，执行初始化方法之前，执行
        //bean 就是刚刚创建的Bean对象， beanName就是这个Bean ID
        logger.debug("初始化前置处理 {} {}", beanName, bean);
        //方法务必返回 bean 对象，否则会干扰 Bean初始化方法
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        //在每个Bean初始化以后执行后置处理方法
        logger.debug("初始化后置处理 {} {}", beanName, bean);
        //方法务必返回 bean 对象，否则会干扰Bean的使用
        return bean;
    }
}

```

添加AOP切面后会自动创建代理：

```

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

```

```

@Component
@Aspect
public class DemoAspect {
    Logger logger = LoggerFactory.getLogger(DemoAspect.class);

    @Before("bean(categoryService)")
    public void test(){
        logger.debug("在方法之前执行");
    }
}

```

测试案例：

```

package cn.tedu.spring;

import cn.tedu.spring.service.CategoryService;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

```

```

@SpringBootTest
public class AopProxyTests {

    Logger logger = LoggerFactory.getLogger(AopProxyTests.class);

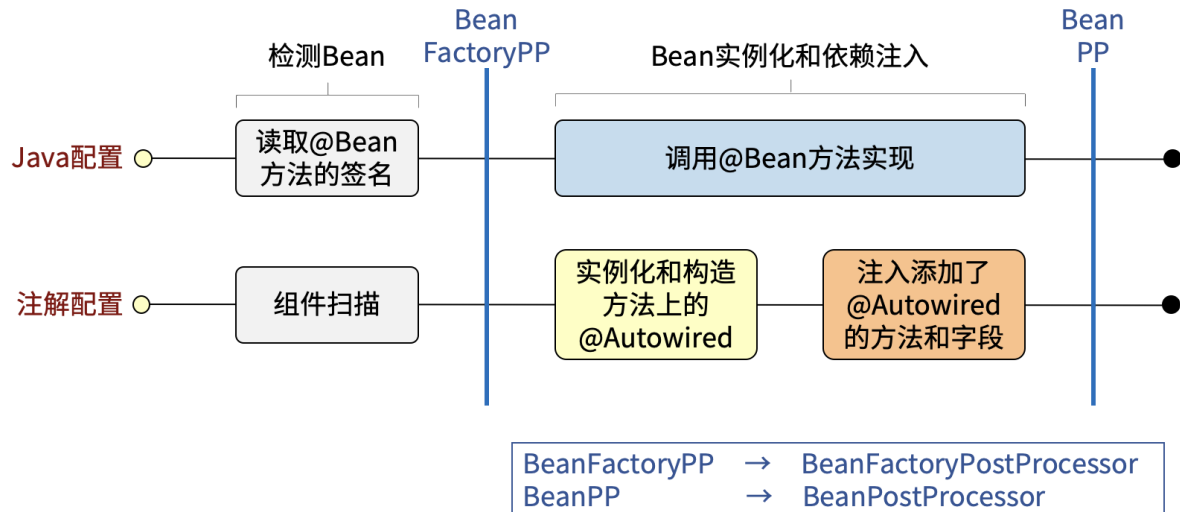
    @Autowired
    CategoryService categoryService;

    @Test
    void test(){
        /*
         * 添加AOP以后，获得categoryService对象的代理对象
         * 这个代理对象 就是 spring 在初始化对象时候
         * 执行AspectJAwareAdvisorAutoProxyCreator
         * 创建的代理对象
         */
        logger.debug("{} ", categoryService.getClass().getName());
    }
}

```

@Bean 和 @Component

@Bean 和 @Component 注解处理方式都是一样的：



Spring Boot

案例在 spring-boot 模块

独立使用Spring的问题

- 搭建一个Spring框架项目，创建maven项目，然后在项目中倒入spring框架用到的依赖
 - spring-core-xx
 - spring-context-xx
 - spring-aop-xx
 -

- 在Spring框架中导入依赖时是需要指定版本号的，此时就可能会产生版本不兼容问题。 eg:导入SpringMVC的依赖，导入mybatis的依赖，
- 框架之间的整合问题:SSM SpringMVC Spring Mybatis，必须导入整合的依赖

什么是Spring Boot

Spring Boot 好处：

- Spring Boot帮助你创建可以运行的独立的、基于Spring的生产级应用程序。
- 对Spring平台和第三方库采用Starter依赖，这样你就能以最少的代码开始工作。
- 大多数Spring Boot应用程序只需要很少的Spring配置, 开箱即用。
- 你可以使用Spring Boot来创建Java应用程序，这些应用程序可以通过使用java -jar或更传统的war部署来启动。
- 我们还提供一个运行 "spring scripts "的命令行工具。

Spring Boot的主要目标是。

- 为所有的Spring开发提供一个从根本上更快、更广泛的入门体验。
- **开箱即用**，但当需求开始偏离默认值时，迅速配置。
- 提供一系列大类项目常见的非功能特性（如嵌入式服务器、安全、度量、健康检查和外部化配置）。

Spring Boot功能

Spring Boot 提供了四大功能。（还记得Spring提供的两大功能么？）

- 依赖管理
- 自动配置
- 打包
- 热部署

面试题：Spring 和SpringBoot的区别：

- Spring（Spring Framework）是Spring全家桶的基石，其核心功能是 IOC/DI、AOP
- Spring Boot 在Spring的基础上提供了开箱即用的功能，四大功能：依赖管理、自动配置、打包、热部署

依赖管理

搭建Spring环境，存在的问题:1. 导入的依赖项非常多 2. 版本不兼容问题

- **Spring Boot** 父级POM，内部使用 **dependencyManagement** 管理了常用组件，解决版本兼容问题
 - start.spring.io 脚手架使用的是 parent 方式
 - start.aliyun.com 脚手架使用的是 dependencyManagement 方式
- 各种 starter 解决依赖包导入问题
 - spring-boot-starter 解决16个jar
 - spring-boot-starter-test 解决测试相关的jar包
 - 举几个例子:
 - spring-boot-starter-jdbc
 - spring-boot-starter-data-jpa
 - spring-boot-starter-web
 - spring-boot-starter-batch

Spring Boot 依赖管理：1 Spring Boot 父级项目提供了依赖管理，2 Spring Boot项目通过XXX-starter自动依赖各种包。

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

自动配置

Enable: 允许

Auto: 自动 汽车

Configuration: 配置

SpringBoot 提供了自动配置功能

- 如果需要在Spring项目使用自动配置，需要在配置类上使用**@EnableAutoConfiguration**
- SpringBoot提供了强大的组合注解 @SpringBootApplication，它的元注解包括：
 - @EnableAutoConfiguration
 - @ComponentScan
 - @SpringBootConfiguration（继承于@Configuration）
- 在SpringBoot启动类中标注@SpringBootApplication就开启了自动配置功能

@SpringBootApplication 的元注解包括 @EnableAutoConfiguration

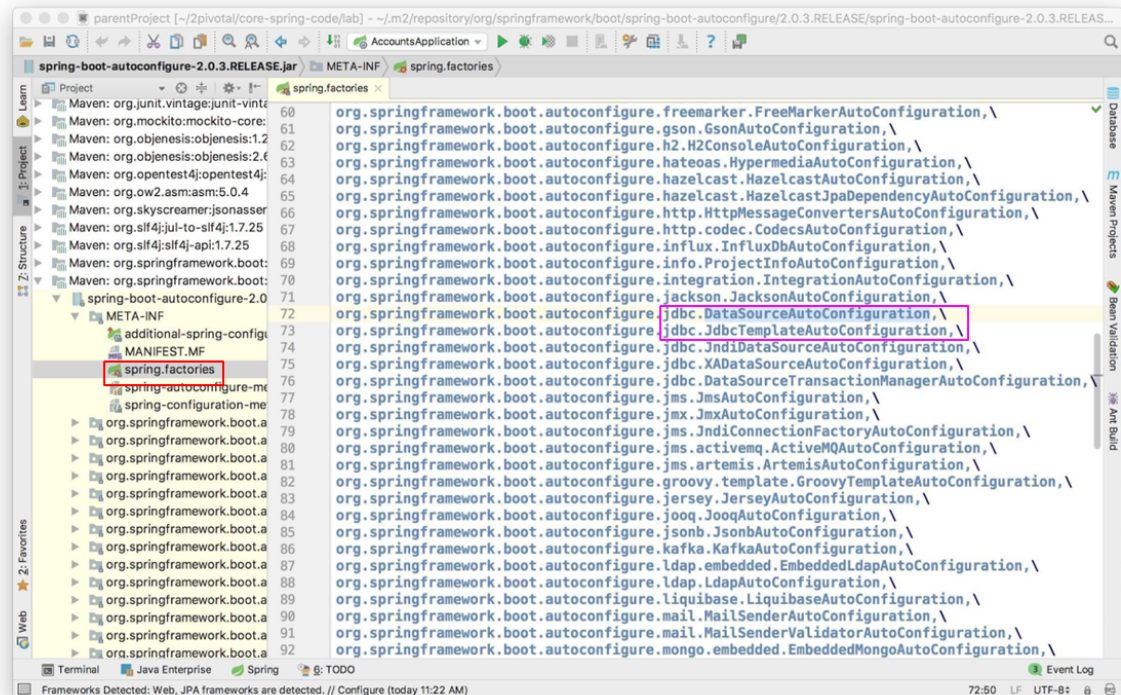
```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
        type = FilterType.CUSTOM,
        classes = {TypeExcludeFilter.class}
    )}, @Filter(
        type = FilterType.CUSTOM,
        classes = {AutoConfigurationExcludeFilter.class}
    )}
)
public @interface SpringBootApplication {
}
```

@EnableAutoConfiguration是如何工作的：

- @EnableAutoConfiguration会读取工厂配置

- 从jar文件中读取spring-boot-autoconfigure/META-INF/spring.factories
- 找到 标注@Configuration 的自动配置类
- 按照自动配置类中的注解完成自动配置

自动配置类实例



什么是自动配置类

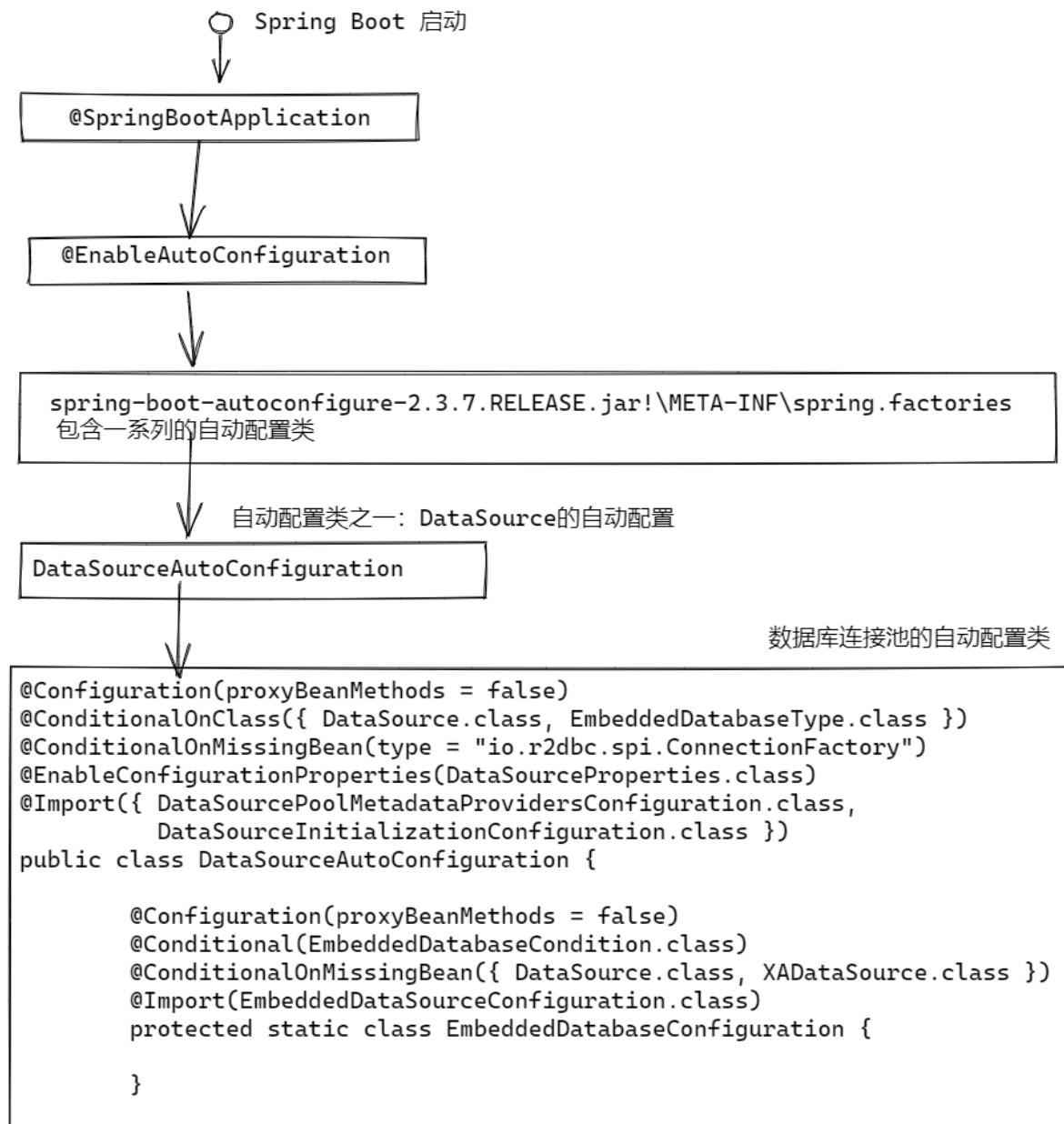
预先写好的配置类

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@ConditionalOnMissingBean(type = "io.r2dbc.spi.ConnectionFactory")
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ DataSourcePoolMetadataProvidersConfiguration.class,
DataSourceInitializationConfiguration.class })
public class DataSourceAutoConfiguration {

    @Configuration(proxyBeanMethods = false)
    @Conditional(EmbeddedDatabaseCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import(EmbeddedDataSourceConfiguration.class)
    protected static class EmbeddedDatabaseConfiguration {

    }
}
```

自动配置原理：



1. 如果配置数据源，则使用用户配置的数据源
2. 如果没有数据源，则使用Spring 提供的数据源
3. 如果没有配置数据库参数，则使用嵌入式数据库
4. 如果导入了SpringJDBC，没有任何数据库参数，则出现启动错误

Conditional: 条件

Missing: 缺少

@Conditional注解，是系列注解 @ConditionalXXX

- 允许条件性的创建Bean
 - 仅当其它Bean存在（或不存在）时创建 Bean 对象

@ConditionalOnBean(Worker.class) 在存在了Worker类型的Bean的时候，创建当前的Bean对象

@ConditionalOnMissingBean(Tool.class) 在缺少Tool类型的 Bean时候，创建当前的Bean

例子：存在 Worker.class 就创建对象 Saw

@Component

```

/**
 * 如果存在 worker 类型的对象时候，就创建电锯对象
 */
@ConditionalOnBean(Worker.class)
public class Saw implements Tool {
    Logger logger = LoggerFactory.getLogger(Saw.class);

    public Saw(){
        logger.debug("创建电锯");
    }
    @Override
    public String toString() {
        return "寒冰锯";
    }
}

```

例子：不存在Tool类型的Bean就创建Axe，使用类型比Bean Name更加方便

```

@Component
/**
 * 如果有工人类型Bean就可以创建 斧子Bean
 * 如果缺少Tool类型的Bean就创建(忽略掉当前的斧子) 斧子Bean
 */
@ConditionalOnMissingBean(name = "saw")
@ConditionalOnBean(Worker.class)
public class Axe implements Tool {
    Logger logger = LoggerFactory.getLogger(Axe.class);

    public Axe(){
        logger.debug("创建斧子");
    }

    @Override
    public String toString() {
        return "开天斧";
    }
}

```

例子：工人：

```

@Component
public class Worker {
    Logger logger = LoggerFactory.getLogger(Worker.class);
    private String name = "光头强";

    @Autowired
    private Tool tool;

    public Worker() {
        logger.debug("创建工人");
    }

    public void work(){
        logger.debug("{}使用{}砍树", name, tool);
    }
}

```

```
}  
}
```

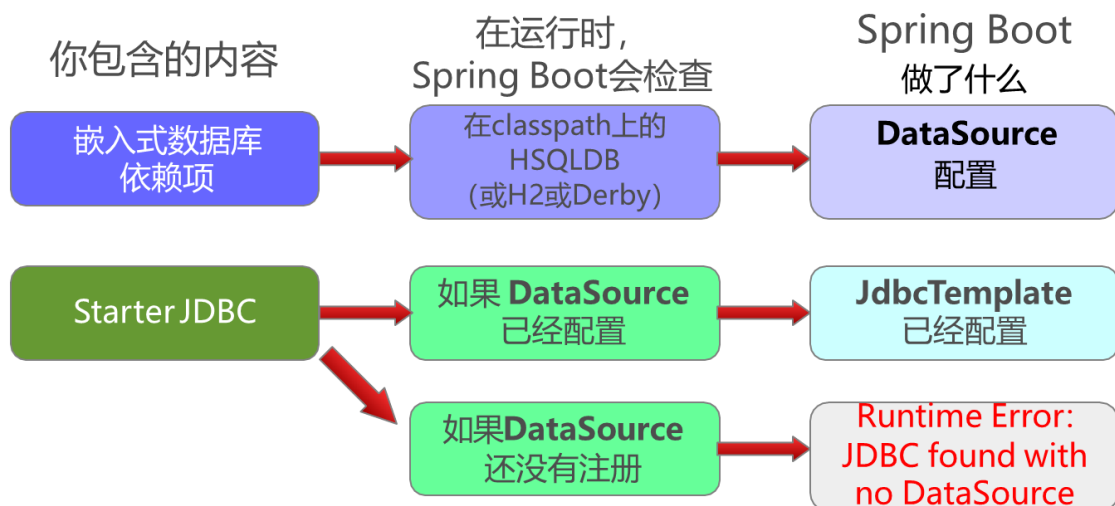
测试类:

```
@SpringBootTest  
public class WorkerTests {  
    Logger logger = LoggerFactory.getLogger(WorkerTests.class);  
  
    @Autowired  
    Worker worker;  
  
    @Test  
    void tests(){  
        logger.debug("{} ", worker);  
    }  
  
    @Test  
    void work(){  
        worker.work();  
    }  
}
```

自动配置数据库链接的例子:

添加 Spring boot jdbc 和 Derby 就会自动配置:

自动配置示例: DataSource, JdbcTemplate



案例, Spring Boot项目添加Derby和Spring Jdbc依赖就会自动配置数据源和JdbcTemplate:


```
<dependencies>
  <dependency>
    <groupId>org.apache.derby</groupId>
    <artifactId>derby</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
  </dependency>
</dependencies>
```

测试:

```
@SpringBootTest
public class DataSourceTests {
    Logger logger = LoggerFactory.getLogger(DataSourceTests.class);

    @Test
    void test(){
        logger.debug("测试");
    }

    @Autowired
    JdbcTemplate jdbcTemplate;

    @Test
    void testJdbcTemplate(){
        logger.debug("{} ", jdbcTemplate);
    }

    @Autowired
    DataSource dataSource;
    @Test
    void driver() throws SQLException {
        logger.debug("{} ", dataSource.getConnection()
            .getMetaData().getDriverName());
    }

    @Test
    void testDataSource(){
        logger.debug("{} ", dataSource.getClass().getName());
    }
}
```

添加MySQL驱动和配置后，就自动更换了数据源连接池：

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

```
# 配置MySQL驱动， 就会自动替换Derby数据库配置
spring.datasource.url=jdbc:mysql://localhost:3306/mysql?
characterEncoding=utf8&useSSL=false&serverTimezone=Asia/Shanghai&rewriteBatchedS
tatements=true
spring.datasource.username=root
spring.datasource.password=root
```

手动配置了数据源，就会自动替换默认数据源：

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.2.12</version>
</dependency>
```

```
@Configuration
public class DataSourceConfig {
    //获取 application.properties 中的配置信息
    @Value("${spring.datasource.url}")
    String url;
    @Value("${spring.datasource.username}")
    String username;
    @Value("${spring.datasource.password}")
    String password;

    @Bean
    public DataSource dataSource(){
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        return dataSource;
    }
}
```

SpringBoot中自定义Bean配置和自动配置的顺序：

- 在定义的Bean显式的创建之后处理自动配置类，你定义的Bean总是优先于自动配置
- 问题：若自己在配置文件中已经配置了数据源，则SpringBoot是否还自动配置数据源

因为优先处理 用户自定义配置，再处理自动配置，可以使用自定义配置覆盖自动配置！！

覆盖配置 也就是修改自动配置

- Spring Boot的设计是为了让覆盖更简单
- 有几种选项
 1. 设置一些Spring Boot的属性（application.properties）
 1. 例如：添加MySQL属性参数，则不会自动配置内嵌数据库
 2. 自己显式的定义Bean，则Spring Boot不会再创建自己的Bean对象
 1. 例如：添加Druid数据源，则Spring不再提供数据源
 3. 显式禁用一些自动配置
 4. 更换依赖项

1 设置SpringBoot的一些属性

- 例如：外置数据源配置属性，可以覆盖SpringBoot默认数据源配置，比如自动更换为MySQL数据库

2 自己显式的定义Bean，创建自己的数据源对象，Spring Boot 就不会自己创建数据源对象了

3 显式禁用一些自动配置

- `@EnableAutoConfiguration(exclude=DataSourceAutoConfiguration.class)`
- `spring.autoconfigure.exclude=\norg.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`

4 显式替代依赖项

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

打包

Fat: 肥、胖

Fat jar: 胖jar

Spring Boot 提供了Fat jar 打包方式:

- 将全部的依赖项和配置、Java类等打包到一个jar文件，包含内嵌Web服务器。
- 只需要一个命令就能部署启动：关闭时候使用 Ctrl+C

```
java -jar xxxx.jar
```

- 文件扩展名，可以是jar或者war都可以
- 同时也提供传统部署jar（瘦jar）没有包含依赖项，可以部署到Tomcat中

```
java -jar spring-boot-0.0.1-SNAPSHOT.jar
```

热部署（了解）

开发过程中需要多次调试，经常重启服务器，SpringBoot提供了开发工具，可以实现热部署，不关闭服务器，自动部署Java类等资源。

添加一个依赖就可以了。

```
<!-- Spring Boot 开发工具， 可以实现热部署功能 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

更容易开发Spring Boot项目

- 自动重启：当一个类改变时（重新编译）
- 支持从IDE远程执行应用程序、全局开发工具设置的附加功能

注意：IDEA中 Spring 热部署工具， 经常失效！