Spring基础

核心注解：SpringbootApplication

@Documented：用于指示编译器在生成Java文档（Javadoc）时，将被注解的元素和注解信息包含在文档中。当你使用 @Documented 注解一个自定义注解时，使用了该自定义注解的元素（类、方法、字段等）的Javadoc中将包含该自定义注解的信息。
@Inherited：用于指示一个自定义注解是否应该被子类继承。当一个使用了 @Inherited 注解的注解被应用到一个类上时，如果这个类的子类没有显式地应用其他相同注解，那么子类将继承父类的该注解。
@SpringBootConfiguration：以下
@EnableAutoConfiguration：以下

@SpringBootConfiguration

1. **定义 Bean：** 在配置类中，通过在方法上添加 `@Bean` 注解，可以将方法返回的对象注册为 Spring 容器中的一个 bean。这样的 bean 将由 Spring 管理其生命周期，并可以在应用程序的其他地方进行依赖注入。
2. **依赖注入：** 使用 `@Autowired`、`@Resource` 或 `@Inject` 等注解，你可以在其他类中将配置类中定义的 bean 注入到属性、构造函数或方法中，实现依赖注入。
3. **条件化配置：** 配置类可以使用 `@Conditional` 注解来根据特定条件决定是否创建某个 bean，这样可以根据不同的环境或配置灵活地定义 bean。
4. **外部属性配置：** 通过在配置类上添加 `@PropertySource` 注解，可以从外部属性文件中读取属性值，然后将其作为 bean 的属性值。
5. **AOP配置：** 在配置类中使用 `@EnableAspectJAutoProxy` 注解启用 Spring AOP，从而可以通过 `@Aspect` 注解定义切面和通知。
6. **组件扫描：** 使用 `@ComponentScan` 注解，可以指定 Spring 在配置类所在的包及其子包中进行组件扫描，自动将带有 `@Component` 注解的类注册为 bean。
7. **事件发布与监听：** 配置类可以通过实现 `ApplicationEventPublisher` 接口来发布事件，也可以使用 `@EventListener` 注解在方法上监听事件并做出相应的处理。
8. **事务管理：** 使用 `@EnableTransactionManagement` 注解来启用 Spring 的事务管理功能，然后在方法上使用 `@Transactional` 注解声明事务。

@EnableAutoConfiguration

1. **自动配置：** 该注解通过扫描 classpath 下的依赖，根据类路径上的 jar 包和项目的依赖信息，自动配置 Spring 应用程序所需的 bean。Spring Boot 根据约定和默认配置，为你的应用程序自动配置各种 Spring 和第三方库的 bean，以避免手动配置。
2. **条件化配置：** Spring Boot 使用条件化配置（`@Conditional` 注解）来根据不同的条件决定是否应用特定的配置。这样可以根据运行时的环境、配置、或 classpath 上的类是否存在等条件，动态地决定是否启用某个配置。
3. **自动扫描：** `@EnableAutoConfiguration` 会自动扫描 Spring Boot 应用程序主类所在包及其子包中的组件，将带有 `@Component` 及相关注解的类注册为 Spring 的 bean。
4. **自动配置优先级：** Spring Boot 的自动配置遵循优先级的规则。即，如果你在应用程序代码中自定义了某个 bean，那么 Spring Boot 自动配置将会被覆盖。这使得你可以轻松地根据需要进行个性化定制。

5. **快速启动：** 通过自动配置，Spring Boot 可以极大地减少配置的工作量，让你更快速地启动和开发应用程序。

Spring特点

轻量：Spring 是轻量的，基本的版本大约 2MB

控制反转：Spring 通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们

面向切面的编程(AOP)：Spring 支持面向切面的编程，并且把应用业务逻辑和系统服务分开

容器：Spring 包含并管理应用中对象的生命周期和配置 MVC 框架：Spring 的 WEB 框架是个精心设计的框架，是 Web 框架的一个很好的替代品

事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）

异常处理：Spring 提供方便的 API 把具体技术相关的异常（比如由JDBC，Hibernate or JDO 抛出的）转化为一致的 unchecked 异常

Spring基本模块

Core module

Bean module

Context module

Expression Language module

JDBC module

ORM module

OXM module

Java Messaging Service(JMS) module

Transaction module

Web module

Web-Servlet module

Web-Struts module

Web-Portlet module

Springboot

特点

减少开发，测试时间和努力。

使用 JavaConfig 有助于避免使用 XML。

避免大量的 Maven 导入和各种版本冲突。

提供意见发展方法。

通过提供默认值快速开始开发。

没有单独的 Web 服务器需要。

这意味着你不再需要启动 Tomcat，Glassfish 或其他任何东 西。

需要更少的配置 因为没有 web.xml 文件。

只需添加用@ Configuration 注释的类，然后添加 用@Bean 注释的方法，Spring 将自动加载对象并像以前一样对其进行管理。

您甚至可以将 @Autowired 添加到 bean 方法中，以使 Spring 自动装入需要的依赖关系中。

基于环境的配置 使用这些属性，您可以将您正在使用的环境传递到应用程序：- Dspring.profiles.active = {enviornment}。在加载主应用程序属性文件后，Spring 将在 （application{environment} .properties）中加载后续的应用程序属性文件。

Spring 框架中都用到了哪些设计模式?

简单工厂模式

实现方式：

BeanFactory。Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得Bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

实质：

由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

实现原理：

bean容器的启动阶段：

- 读取bean的xml配置文件,将bean元素分别转换成一个BeanDefinition对象。
- 然后通过BeanDefinitionRegistry将这些bean注册到beanFactory中，保存在它的一个ConcurrentHashMap中。
- 将BeanDefinition注册到了beanFactory之后，在这里Spring为我们提供了一个扩展的切口，允许我们通过实现接口BeanFactoryPostProcessor 在此处来插入我们定义的代码。典型的例子就是：PropertyPlaceholderConfigurer，我们一般在配置数据库的dataSource时使用到的占位符的值，就是它注入进去的。

容器中bean的实例化阶段：

实例化阶段主要是通过反射或者CGLIB对bean进行实例化，在这个阶段Spring又给我们暴露了很多的扩展点：

- **各种的Aware接口**，比如 BeanFactoryAware，对于实现了这些Aware接口的bean，在实例化bean时Spring会帮我们注入对应的BeanFactory的实例。
- **BeanPostProcessor接口**，实现了BeanPostProcessor接口的bean，在实例化bean时Spring会帮我们调用接口中的方法。

- **InitializingBean接口**，实现了InitializingBean接口的bean，在实例化bean时Spring会帮我们调用接口中的方法。
- **DisposableBean接口**，实现了BeanPostProcessor接口的bean，在该bean死亡时Spring会帮我们调用接口中的方法。

工厂方法

实现方式：

FactoryBean接口。

实现原理：

实现了FactoryBean接口的bean是一类叫做factory的bean。其特点是，spring会在使用getBean()调用获得该bean时，会自动调用该bean的getObject()方法，所以返回的不是factory这个bean，而是这个bean.getOjbect()方法的返回值。

例子：

典型的例子有spring与mybatis的结合。

单例模式

Spring依赖注入Bean实例默认是单例的。

Spring的依赖注入（包括lazy-init方式）都是发生在AbstractBeanFactory的getBean里。getBean的doGetBean方法调用getSingleton进行bean的创建。

分析getSingleton()方法

适配器模式

实现方式：

SpringMVC中的适配器HandlerAdatper。

实现原理：

HandlerAdatper根据Handler规则执行不同的Handler。

实现过程：

DispatcherServlet根据HandlerMapping返回的handler，向HandlerAdatper发起请求，处理Handler。

HandlerAdapter根据规则找到对应的Handler并让其执行，执行完毕后Handler会向HandlerAdapter返回一个ModelAndView，最后由HandlerAdapter向DispatchServelet返回一个ModelAndView。

实现意义：

HandlerAdatper使得Handler的扩展变得容易，只需要增加一个新的Handler和一个对应的HandlerAdapter即可。

因此Spring定义了一个适配接口，使得每一种Controller有一种对应的适配器实现类，让适配器代替controller执行相应的方法。这样在扩展Controller时，只需要增加一个适配器类就完成了SpringMVC的扩展了。

装饰器模式

实现方式：

Spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。

实质：

动态地给一个对象添加一些额外的职责。

就增加功能来说，Decorator模式相比生成子类更为灵活。

代理模式

实现方式：

AOP底层，就是动态代理模式的实现。

动态代理：

在内存中构建的，不需要手动编写代理类

静态代理：

需要手工编写代理类，代理类引用被代理对象。

实现原理：

切面在应用运行的时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象创建动态的创建一个代理对象。SpringAOP就是以这种方式织入切面的。

织入：把切面应用到目标对象并创建新的代理对象的过程。

观察者模式

实现方式：

spring的事件驱动模型使用的是 观察者模式 ， Spring中Observer模式常用的地方是listener的实现。

具体实现：

事件机制的实现需要三个部分,事件源,事件,事件监听器

ApplicationEvent抽象类[事件]

继承自jdk的EventObject,所有的事件都需要继承ApplicationEvent,并且通过构造器参数source得到事件源.

该类的实现类ApplicationContextEvent表示ApplicaitonContext的容器事件.

Spring由哪些模块组成?

- Spring Core（Spring核心）：这是Spring框架的核心部分，包含IoC（Inversion of Control）容器，它管理应用程序中的对象和它们之间的依赖关系。Spring Core还提供了一系列工具类和配置文件，用于简化开发过程。
- Spring AOP（Aspect-Oriented Programming）：这个模块支持面向切面编程，允许通过切面（Aspects）来实现横切关注点的功能，如日志记录、安全性等。AOP可以帮助将应用程序的关注点分离，提高代码的可重用性和可维护性。
- Spring Data：Spring Data模块简化了与数据访问相关的开发工作。它为许多数据存储技术（如关系型数据库、NoSQL数据库等）提供了通用的数据访问接口，同时还提供了更高级别的查询功能和数据操作。
- Spring JDBC（Java Database Connectivity）：这是Spring框架的一个子模块，提供了对JDBC的封装和简化，使得在Java应用程序中更容易地访问和操作数据库。
- Spring ORM（Object-Relational Mapping）：Spring ORM模块提供了对对象关系映射框架的集成支持，如Hibernate、JPA等，使得在应用程序中使用对象而不是原始SQL来操作数据库变得更容易。
- Spring Web：这个模块提供了构建Web应用程序的支持，包括Web MVC框架，用于开发基于模型-视图-控制器（MVC）的Web应用程序。
- Spring WebFlux：这是Spring框架的响应式编程模块，支持构建异步非阻塞的Web应用程序。它基于Reactor库，允许处理高并发和高负载的Web请求。
- Spring Security：Spring Security是用于保护应用程序的安全性的模块。它提供了身份验证、授权、加密等功能，帮助保护应用程序免受常见的安全威胁。
- Spring Test：这个模块提供了对Spring应用程序进行单元测试和集成测试的支持，包括使用JUnit或TestNG进行测试的工具。

Spring IoC 的实现机制

1. Bean定义：Bean定义是Spring IoC容器中的一个元数据，它描述了要创建的Bean的信息，包括Bean的类型、依赖关系、作用域等。通常，Bean定义可以通过XML配置、Java注解或Java代码来定义。
2. 容器：IoC容器是Spring框架的核心部分，负责管理Bean的生命周期和依赖关系。Spring提供了不同类型的IoC容器，包括XmlBeanFactory（过时）、ApplicationContext等。
3. Bean实例化：当应用程序启动时，IoC容器会根据Bean定义创建相应的Bean实例。这通常是通过Java的反射机制来实现的，IoC容器会根据Bean定义中指定的类路径来实例化Bean。
4. 依赖注入：在创建Bean实例时，IoC容器会解析Bean定义中的依赖关系，并将依赖的Bean注入到目标Bean中。这样，Bean之间的依赖关系由容器来管理，而不是由开发者手动管理。
5. 生命周期管理：IoC容器负责管理Bean的生命周期，包括初始化、销毁等阶段。可以通过在Bean中实现特定的接口或使用注解来定义初始化和销毁的回调方法。
6. 延迟加载：Spring允许将Bean的创建延迟到第一次使用时。这样可以提高应用程序的启动性能，只有在需要时才会实例化Bean。
7. 作用域：Spring提供了不同的Bean作用域，包括Singleton（单例，默认）、Prototype（原型）、Request、Session等。开发者可以根据需要选择合适的作用域来管理Bean的生命周期。

Prototype和Singleton作用域

Singleton作用域：

- 默认作用域：当没有指定作用域时，默认为Singleton作用域。
- 单例模式：在Singleton作用域下，容器只会创建一个Bean实例，并将其放入容器中。后续对该Bean的请求都会返回同一个实例。
- 生命周期：Bean在容器启动时被创建，并在容器关闭时销毁。

Prototype作用域：

- 原型模式：在Prototype作用域下，容器每次请求都会创建一个新的Bean实例，即每次请求都会返回一个不同的实例。
- 生命周期：容器创建Bean实例后，不负责后续的生命周期管理，需要由开发者手动管理Bean的销毁。

BeanFactory和ApplicationContext

- ApplicationContext继承了BeanFactory
- ApplicationContext提供**生命周期、注解扫描、AOP、事务监听以及Web应用支持**
- BeanFactory主要提供**实例化、依赖注入（DI）、作用域和延迟加载**
- BeanFactory不足之处：不会调用BeanFactory后置处理器，不会主动添加Bean后置处理器，不会主动初始化单例

ApplicationContext方法

- getResource：资源配置项
- getEnvironment：环境配置
- publishEvent：事件发布，主要用于组件事件之间的解耦

ApplicationContext

- ClassPatgXmlApplicationContext：基于classpath下的xml文件来创建容器
- FileSystemXmlApplicationContext：基于磁盘路径下的xml文件来创建容器
    - XmlBeanDefinitionReader：用于读取xml内部信息
- AnnotationConfigApplicationContext：基于注解来创建管理Bean的容器
- AnnotationConfigServletWebSeverApplicationContext：基于注解创建，可以用于Web环境配置，其中需要在WebConfig中配置以下Bean：
    - ServletWebSeverFactory：返回一个Tomcat Web服务器
    - DispatchServlet：返回一个Servlet前置处理器
    - DispatchServletRegistrationBean：将前置处理器注册到Web服务器当中
- GenericApplicationContext：是一个赶紧的容器，排除前、后置处理器的同时，还可以直接使用registerBean方法注入Bean

BeanFactory

- 给BeanFactory添加一些后置处理器，可以扫描配置类中的Bean（主要是用于补充对Bean的定义）
- BeanFactory加载Bean一般是懒汉式加载，使用preInstantiateSingletons可以在初始化时就加载Bean
- 排序加载注解机制，优先度数值小的先解析
- 使用doResolverDependecy进行依赖解析

Bean生命周期

1. 实例化前置处理方法
2. 构造方法（实例化方法）
3. 实例化后置处理方法
4. 依赖注入，属性复制
5. preProcessor初始化前置处理器进行前置方法调用
6. 初始化方法调用
7. postProcessor初始化后置处理器进行后置方法调用
8. 使用Bean
9. 销毁之前方法调用
10. 销毁Bean

Bean后置处理器

- AutowiredAnnotationBeanPostProcessor：自动装配注解后置处理器，包括@Autowired、@Value
    - 使用inject方法完成依赖注入
    - @Value注解，${property}表示配置项，#{obj.property}表示某一个bean的属性
- CommonAnnotationBeanPostProcessor：处理@Resource、@PostConstruct、@PreDestory
    - @Resource默认根据byName进行注入，@Autowired默认根据byType进行注入
- ConfigurationProperties：负责配置属性后置处理（Config）

BeanFactory后置处理器

- ConfigurationClassPostProcessor：处理@ComponentScan（组件扫描）、@Bean、@Import、@ImportResource注解
    - ComponentScan主要用于扫描basePackages下的类
    - 具体流程：根据注解元数据AnnotationMetadata来解析注解信息，根据注解信息来注入Bean
- MapperScannerConfigurer：处理MapperScanner（数据映射扫描），并需要指定被扫描的包名
- MapperPostProcessor：处理Mapper的后置处理器
    - 在MapperFactoryBean中注入sqlSessionFactory

Autowired注解

- 不建议使用Autowired注解，原因是自动注入可能会使系统的耦合度增加，可以使用Setter或者构造函数注入，但是需要在Config或者xml配置中去配置bean

如何将Mybatis注入到Spring当中?

1. 配置数据源：在 Spring 配置文件中配置数据源，可以使用 Spring 提供的数据源或自定义数据源。
2. 配置 SqlSessionFactoryBean：配置 MyBatis 的 SqlSessionFactoryBean，它是 MyBatis 的核心配置，用于创建 SqlSessionFactory 实例。设置数据源和 MyBatis 的配置文件路径。
3. 配置 MapperScannerConfigurer：配置 MapperScannerConfigurer，用于扫描指定包下的 Mapper 接口，并将其注册为 MyBatis 的 Mapper 接口。
4. 编写 Mapper 接口：定义 Mapper 接口，其中定义 SQL 查询或更新的方法，方法的名称和参数与实际的 SQL 语句相对应。
5. 编写 MyBatis 映射文件：在映射文件中，定义 SQL 语句和结果映射关系。Mapper 接口中的方法名与映射文件中的 SQL 语句的 id 对应，SQL 语句的参数和返回值与方法的参数和返回值相对应。

AOP的实现方法

1. AspectJ：AspectJ 是一种 AOP 框架，它是 Java 语言的扩展，提供了更丰富和更灵活的 AOP 支持。AspectJ 可以通过编译时织入、类加载时织入或运行时织入等方式实现 AOP。
2. Spring AOP：Spring AOP 是 Spring 框架的一部分，提供了基于代理的轻量级 AOP 支持。它主要通过 JDK 动态代理和 CGLIB 代理来实现，可以在方法调用前、后、抛出异常时等切入横切关注点。
3. 在类加载阶段通过agent代理

JDK代理

```
public class JDKDynamicProxy implements InvocationHandler {

    private final Object target;

    public JDKDynamicProxy(Object target) {
        this.target = target;
    }

    public Object getProxy() {
        return Proxy.newProxyInstance(
                this.target.getClass().getClassLoader(),
                this.target.getClass().getInterfaces(),
                this
        );
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 在方法执行前进行增强
        System.out.println("Before method execution");
        Object result = method.invoke(target, args); // 调用目标对象的方法
        // 在方法执行后进行增强
```

```
        System.out.println("After method execution");
        return result;
    }
}
```

CGLIB代理

```
public class CGLIBProxy implements MethodInterceptor {

    private Object target;

    public CGLIBProxy(Object target) {
        this.target = target;
    }

    public Object getProxy() {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(target.getClass());
        enhancer.setCallback(this);
        return enhancer.create();
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy)
throws Throwable {
        // 在方法执行前进行增强
        System.out.println("Before method execution");
        Object result = proxy.invokeSuper(obj, args); // 调用目标对象的方法
        // 在方法执行后进行增强
        System.out.println("After method execution");
        return result;
    }
}
```

JDK代理和CGLIB代理的区别

JDK 代理（基于接口的代理）：

- JDK 代理是基于接口的动态代理技术，要求目标对象必须实现接口。
- JDK 代理通过 `java.lang.reflect.Proxy` 类来创建代理对象。它只能代理接口中声明的方法。
- JDK 代理生成的代理类是通过反射机制创建的，性能较 CGLIB 稍差。但由于只能代理接口方法，适用于基于接口的代理场景。

CGLIB 代理（基于类的代理）：

- CGLIB 代理是基于类的动态代理技术，可以代理目标类的方法，包括不实现接口的类。
- CGLIB 代理通过生成目标类的子类来创建代理对象，因此目标类不能是 final 类，也不能被 final 方法修饰。
- CGLIB 代理生成的代理类是通过继承目标类的子类实现的，性能通常比 JDK 代理更好，但生成过程更复杂。
- 由于 CGLIB 代理是基于类的，因此适用于不实现接口或无法修改目标类的情况。

SpringSecurity执行流程

认证技术

基于表单的认证

基于JWT的认证

Http Basic认证

Http Digest认证

Oauth认证，单点登录

认证流程

定义加密器Bean

创建用户信息表

RoleInfi权限表

自定义UserDetail

实现UserDetailsService

编写Controller

编写认证服务

配置控制访问

循环依赖解决方案

1. **重构代码：** 重构是解决循环依赖的有效方法之一。重新设计模块或组件之间的关系，将相互引用的部分拆分成更小的模块，避免直接的循环依赖。
2. **接口隔离：** 如果循环依赖发生在接口或抽象类之间，可以考虑使用接口隔离原则，将接口细化成更小的接口，避免多个模块在同一个接口上相互依赖。
3. **依赖注入：** 使用依赖注入（Dependency Injection）可以将依赖关系从代码中解耦，从而避免循环依赖。通过依赖注入容器来管理依赖关系，可以更灵活地控制模块之间的依赖。
4. **中介者模式：** 中介者模式可以将多个模块之间的交互逻辑集中在一个中介者对象中，避免直接的相互引用。这样可以降低模块之间的耦合性，从而减少循环依赖的发生。
5. **延迟初始化：** 在一些情况下，可以使用延迟初始化来避免循环依赖。即将初始化过程推迟到需要使用某个模块时再进行，以减少初始化过程中的依赖问题。
6. **模块分割和顺序控制：** 如果循环依赖无法避免，可以考虑将模块进行分割，将具有循环依赖的部分放到一个单独的模块中，并通过控制模块的初始化顺序来避免循环依赖问题。

SpringMVC什么周期

1. **请求到达 DispatcherServlet：** 客户端发起一个 HTTP 请求，请求被 DispatcherServlet（前端控制器）接收。
2. **HandlerMapping 查找处理器：** DispatcherServlet 使用 HandlerMapping 来查找适合处理该请求的处理器（Controller）。
3. **处理器执行：** 找到匹配的处理器后，DispatcherServlet 会调用处理器的适当方法来执行业务逻辑。
4. **处理器适配器调用处理器：** 处理器适配器（HandlerAdapter）会处理处理器的调用，将请求和处理器进行适配，确保正确的处理方法被调用。
5. **处理器执行：** 处理器适配器会调用实际的处理器方法，执行业务逻辑，返回一个 ModelAndView 对象。
6. **ModelAndView 创建：** 处理器方法返回的 ModelAndView 包含模型数据和要返回的视图名。
7. **ViewResolver 解析视图名：** DispatcherServlet 使用 ViewResolver 来解析视图名，找到对应的视图对象。
8. **视图渲染：** 视图对象会根据模型数据渲染页面内容，生成最终的 HTML 输出。
9. **响应返回客户端：** DispatcherServlet 将生成的 HTML 页面作为响应返回给客户端，完成请求的处理。

JPA

@Entity：注解用于标注一个类是一个实体

@Id：注解标志实体类的主键字段

瞬态、持久、托管

1. **瞬态（Transient）状态：** 瞬态状态指的是实体对象刚刚被创建，尚未和持久化上下文建立关联，也没有与数据库进行交互。在这个状态下，实体对象不受 JPA 管理，对其进行的任何更改都不会反映到数据库中。
2. **持久（Persistent）状态：** 持久状态指的是实体对象与持久化上下文建立了关联，并且在数据库中有对应的记录。当实体对象从瞬态状态转换为持久状态时，JPA 将跟踪它的状态变化，并且对实体对象的更改会被同步到数据库中。持久状态的实体对象可以通过 JPA 提供的 EntityManager 进行数据库操作。
3. **脱管（Detached）状态：** 脱管状态指的是实体对象曾经处于持久状态，但是与持久化上下文的关联已经断开。这可能是因为实体对象所在的持久化上下文被关闭，或者实体对象被显式地从上下文中移除。在脱管状态下，实体对象仍然代表一个数据库记录，但是对其进行的更改不会自动同步到数据库。如果需要将脱管状态的对象重新变为持久状态，可以使用 EntityManager 的 merge 方法将对象重新关联到持久化上下文。

Springboot的启动流程

1. **加载配置：** Spring Boot 应用启动时会加载各种配置文件，包括应用的配置文件（如 `application.properties` 或 `application.yml`）、外部配置（如环境变量）等。这些配置文件会覆盖 Spring Boot 默认的配置，从而定制化应用的行为。
2. **初始化ApplicationContext：** Spring Boot 应用启动后会初始化应用上下文（ApplicationContext），这是 Spring 框架的核心容器，管理着各个 Bean（组件）的生命周期和依赖关系。
3. **执行自动配置：** Spring Boot 会根据应用的依赖和配置，自动进行配置。这包括数据库连接、缓存配置、Web 服务器设置等。自动配置可以大大减少开发者的配置工作，让应用快速上手。

4. **扫描组件：** Spring Boot 会扫描应用中的各个组件，将它们注册到应用上下文中，以供后续使用。
5. **启动内嵌的 Web 服务器：** 如果应用是一个 Web 应用，Spring Boot 会启动内嵌的 Web 服务器，如 Tomcat、Jetty 或 Undertow。这使得应用可以独立运行，无需额外的外部 Web 服务器。
6. **执行命令行运行器和应用启动器：** Spring Boot 允许在应用启动后执行一些特定的代码，可以通过实现 `CommandLineRunner` 接口来实现。同时，应用启动类的 `main` 方法会被执行，整个应用正式启动。
7. **处理请求：** 如果应用是一个 Web 应用，内嵌的 Web 服务器开始监听请求。当收到请求后，Spring Boot 会根据 URL 映射、Controller 等进行请求处理，返回相应的响应。