

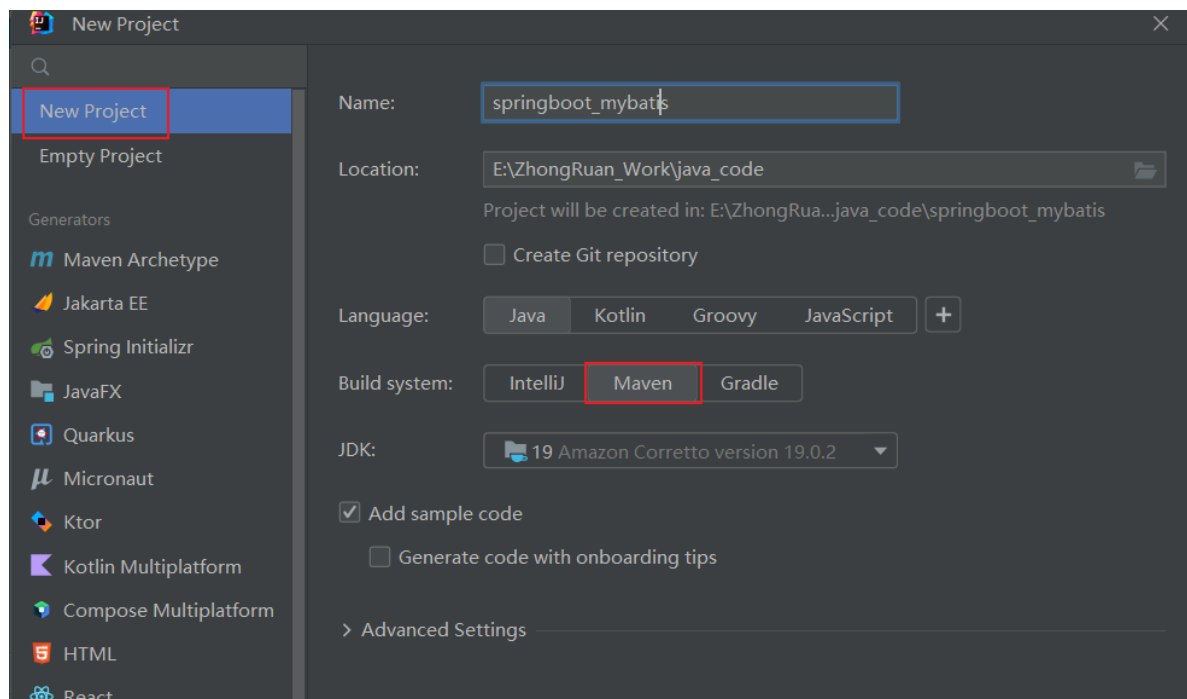
前言

此处记录学习过程，目前仍在学习过程中，这些学习过程的记录希望能对其他人也起到帮助

SpringBoot

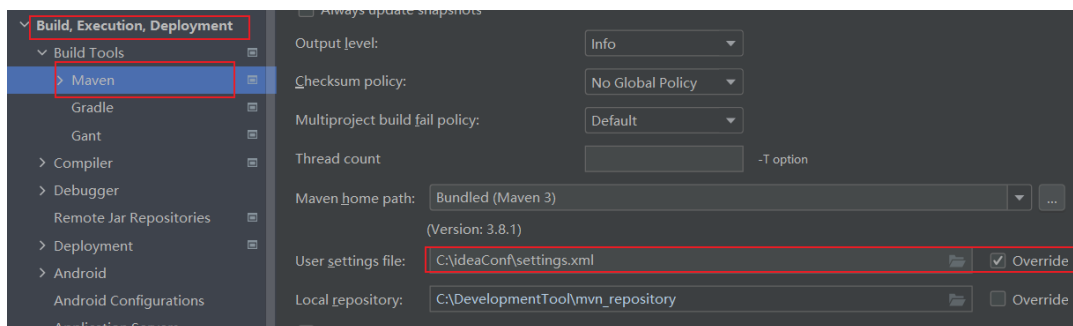
一.SpringBoot

1.项目创建



当网络不好时或者其他问题，Spring Initializr的SpringBoot项目创建方式会失败。这里我采用Maven的形式创建，当U盘拷贝了所需的Maven库后，使用该Maven库，在其他电脑上也可以快捷创建SpringBoot项目。

- Maven配置



在idea左上角File中的Settings中可以更改如上Maven配置，settings.xml中配置了mvn_repository的位置，所以下方勾选框没有勾选。settings.xml中可修改位置。

```

46 <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
47     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaL
48     <!-- localRepository
49         | The path to the local repository maven will use to store artif
50         |
51         | Default: ${user.home}/.m2/repository
52     <localRepository>/path/to/local/repo</localRepository>
53     -->
54     <localRepository>C:\DevelopmentTool\mvn_repository</localRepositio
55     <!-- interactiveMode
56         | This will determine whether maven prompts you when it needs in
57         | maven will use a sensible default value, perhaps based on some
58         | the parameter in question.
59         |

```

2.pom.xml基础配置

- spring-boot-starter-parent:父项目（<parent></parent>）依赖指定，基于SpringBoot的项目需要指定。
- 项目坐标信息与打包方式

```

<groupId>org.example</groupId>
<artifactId>springboot_mybatis</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

```

- spring-boot-starter-web：构建一个 Web 应用程序时，这个依赖项将提供 Spring MVC 和内嵌的 Tomcat 服务器等相关支持。
- junit：单元测试框架之一。
- lombok：Lombok 是一个 Java 库，它通过注解来简化 Java 代码的编写。它提供了一系列的注解，用于自动生成 Java 类中的常见方法，如 getter、setter、equals、hashCode 等，以及简化代码中的样板代码。

添加如上信息至pom.xml文件中，项目坐标信息与打包方式不用更改。添加后结果如下。

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <parent>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-parent</artifactId>
10        <version>2.6.1</version>
11    </parent>
12
13    <groupId>org.example</groupId>
14    <artifactId>springboot_mybatis</artifactId>
15    <version>1.0-SNAPSHOT</version>
16    <packaging>jar</packaging>
17
18    <properties>
19        <maven.compiler.source>19</maven.compiler.source>
20        <maven.compiler.target>19</maven.compiler.target>

```

```

21     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
22 </properties>
23
24 <dependencies>
25     <dependency>
26         <groupId>org.springframework.boot</groupId>
27         <artifactId>spring-boot-starter-web</artifactId>
28     </dependency>
29
30     <dependency>
31         <groupId>junit</groupId>
32         <artifactId>junit</artifactId>
33         <version>3.8.1</version>
34         <scope>test</scope>
35     </dependency>
36
37     <dependency>
38         <groupId>org.projectlombok</groupId>
39         <artifactId>lombok</artifactId>
40     </dependency>
41 </dependencies>
42 </project>

```

Maven配置之后，需要刷新一下。在ideal右侧。

```

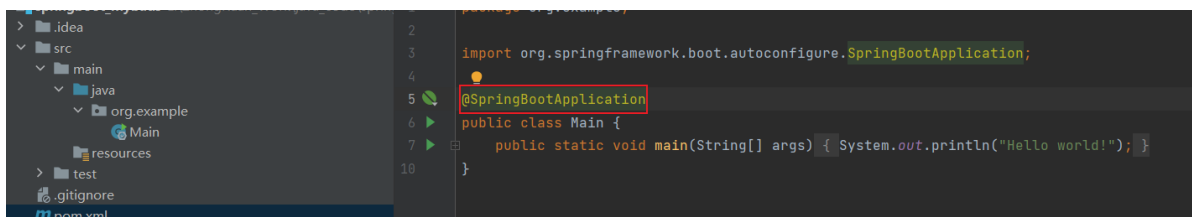
46 <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
47     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaL
48 <!-- localRepository
49     | The path to the local repository maven will use to store artif
50     |
51     | Default: ${user.home}/.m2/repository
52 <localRepository>/path/to/local/repo</localRepository>
53 -->
54 <localRepository>C:\DevelopmentTool\mvn_repository</localRepositio
55 <!-- interactiveMode
56     | This will determine whether maven prompts you when it needs in
57     | maven will use a sensible default value, perhaps based on some
58     | the parameter in question.
59     |

```

3.主应用程序

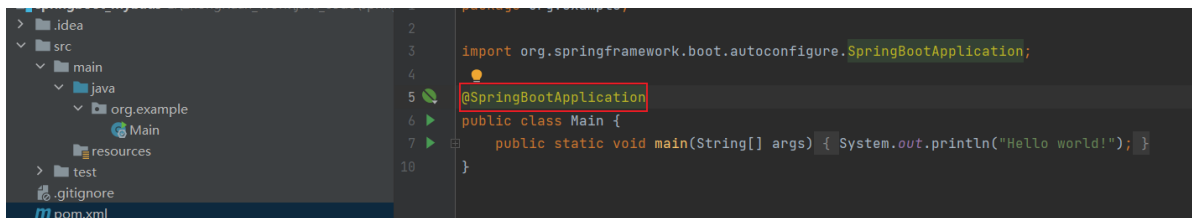
在步骤2web依赖导入后，添加注解@SpringBootApplication

- @SpringBootApplication：用于标识主应用程序类。可以让 Spring Boot 应用程序自动扫描并加载所有组件，并启用自动配置，从而简化应用程序的配置工作。



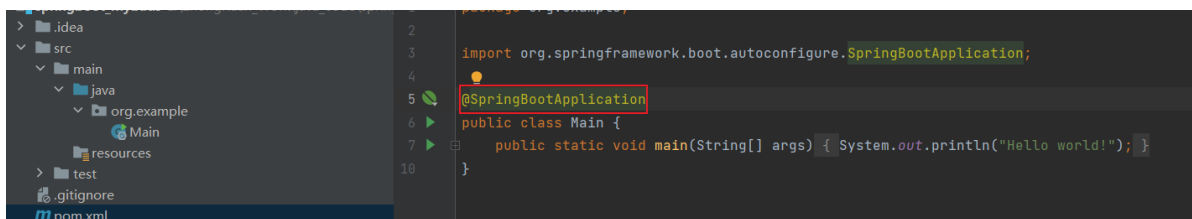
考虑到之后的操作，我将“Main”改名为“BootMyBatisApplication”（改名起标识作用，更美观？）。点击左侧项目App右键，然后点击Refactor-》Rename进行修改。

修改代码如下



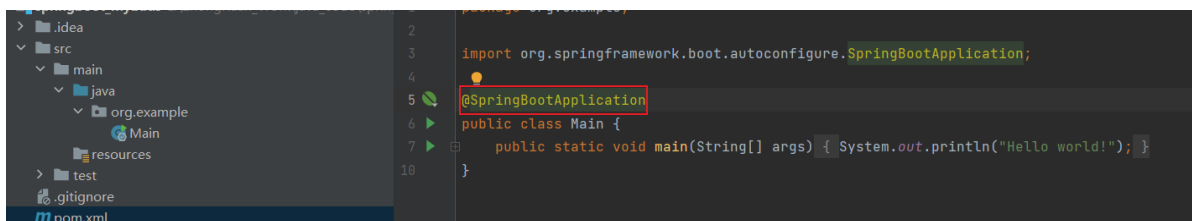
```
1 package org.example;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class BootMyBatisApplication {
8     public static void main(String[] args) {
9         SpringApplication.run(BootMyBatisApplication.class, args);
10    }
11 }
```

然后点击代码左侧绿色启动按钮。下方控制台呈现结果如下。



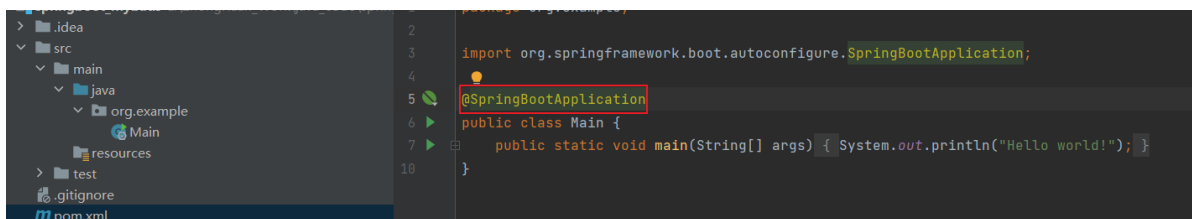
SpringBoot项目默认使用8080端口，可以在resources文件下的application.yml文件中进行配置（无该文件就先创建，application.properties与application.yml配置方式略有不同，本篇使用yml进行配置）。

无报错，SpringBoot项目启动，打开浏览器，在网址栏输入localhost:8080,由于现在没有设置页面，所以呈现默认的错误页面。



4.自动装配原理

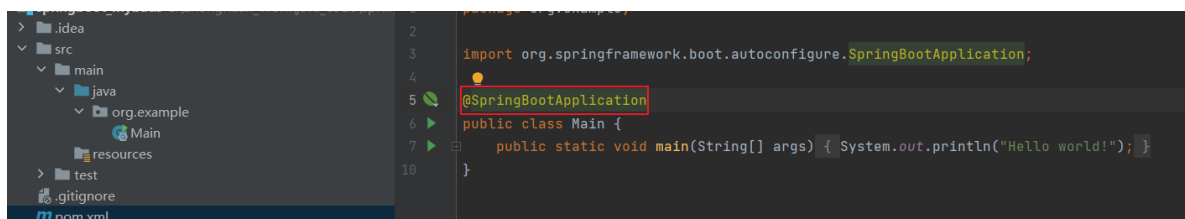
按住Ctrl点击@SpringBootApplication，可以查看它的代码构成。



- @Target(): 注解修饰内容
 - ElementType.TYPE: 类型
 - ElementType.FIELD: 属性
 - ElementType.METHOD: 方法
- @Retention() ; 定义 注解存活的周期

- RetentionPolicy.SOURCE: 只活在源码
 - RetentionPolicy.CLASS: 编译之后 字节码还有。运行时没有
 - RetentionPolicy.RUNTIME: 运行时也有。
- @Documented
 - @Documented 注解表示被它修饰的注解应该被 javadoc 工具记录。也就是说，如果我们自定义了一个注解，并且在这个注解上添加了 @Documented 注解，那么使用 javadoc 生成文档时，这个注解的信息将被包含在文档中，让用户能够看到。
- @Inherited:
 - @Inherited 注解表示被它修饰的注解可以被继承。通常情况下，Java 中的注解不具有继承性，即使一个类继承了另一个类，它也不会继承父类上的注解。但是如果父类的注解被 @Inherited 修饰，那么子类会继承父类的这个注解。
- @SpringBootConfiguration: 声明为配置文件，内部包含 @Configuration: 当前类实例化，放入容器
- @EnableAutoConfiguration: Spring Boot 在启动时会扫描 classpath 下的各种配置文件、jar 包和依赖，以寻找符合条件的配置类，然后根据找到的配置类以及当前环境的条件，自动配置应用程序的各种功能，例如数据源、JPA、Web MVC、安全性等。这将减少手动配置的工作量。

@EnableAutoConfiguration核心内容

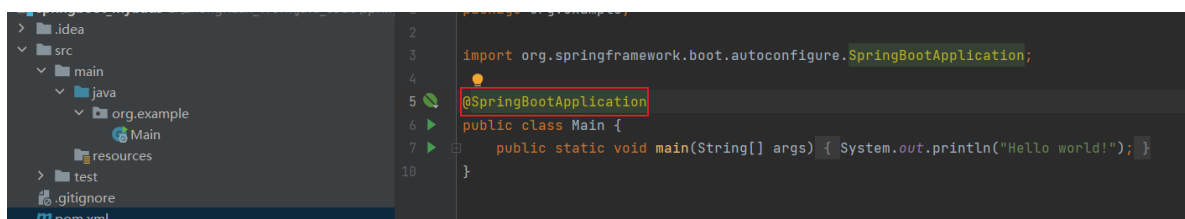


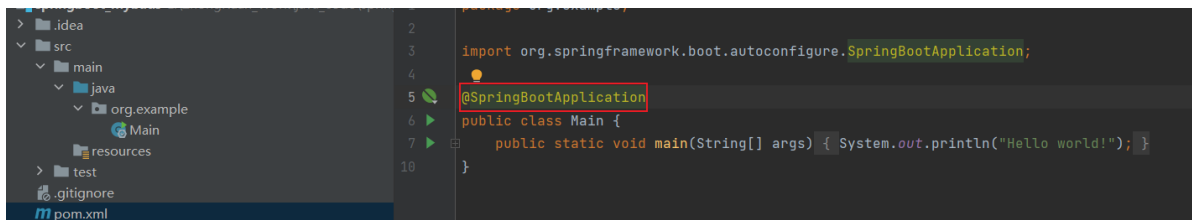
- @ComponentScan: 组件的扫描。扫描包: 当前类所有的包下的 所有类
- 这三个注解是自动装配的核心，它们的联动过程如下：
- @SpringBootConfiguration 标识这是一个 Spring Boot 的配置类，可以用来定义其他配置类或者 Bean。
 - @EnableAutoConfiguration 启用了 Spring Boot 的自动配置机制，根据 classpath 下的依赖和条件，自动配置应用程序的各种功能。
 - @ComponentScan 会扫描指定的包及其子包，寻找标有 @Component、@Service、@Repository 等注解的类，并注册为 Spring Bean。这样，被 @EnableAutoConfiguration 自动配置的类，以及开发者自定义的类，都会被扫描并注册为 Spring Bean。

4.1 自定义配置类

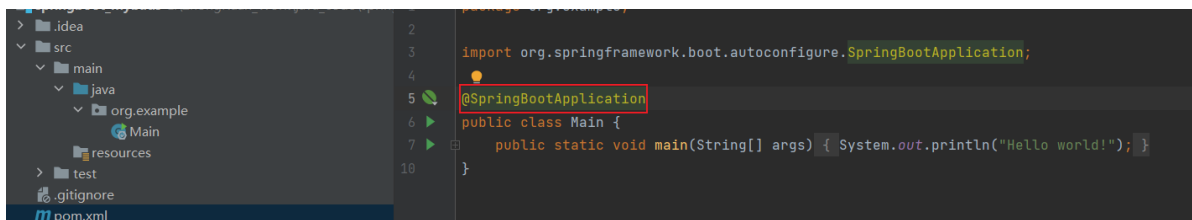
该部分内容不影响后续内容，可以不用做。

右键项目，创建Module

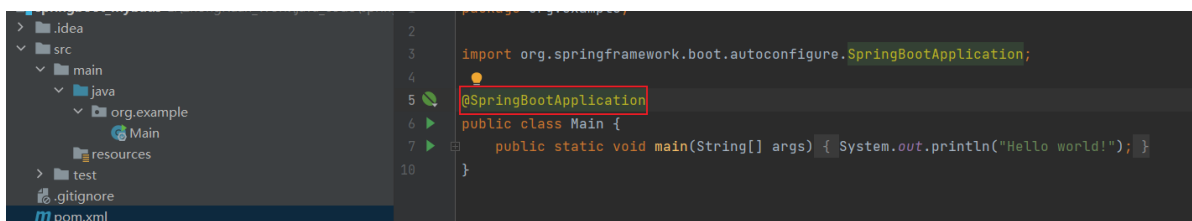




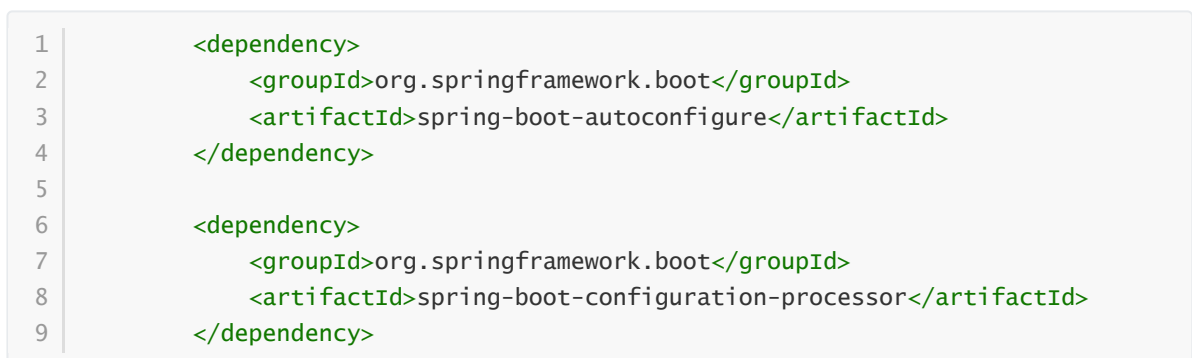
首先修改MyConfig-spring-boot-start项目中pom.xml内容，此处指示父类为外部的springboot_mabtis，这两者不能有关系，修改如下。



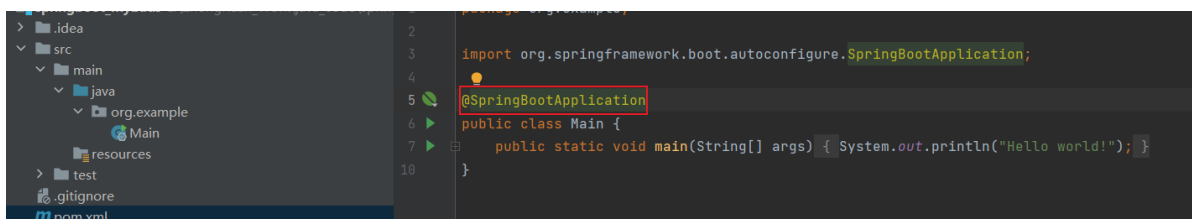
然后标识为org.example包下



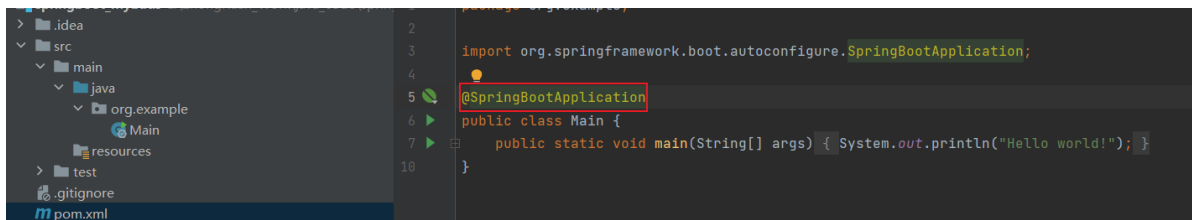
接着添加自动装配的依赖



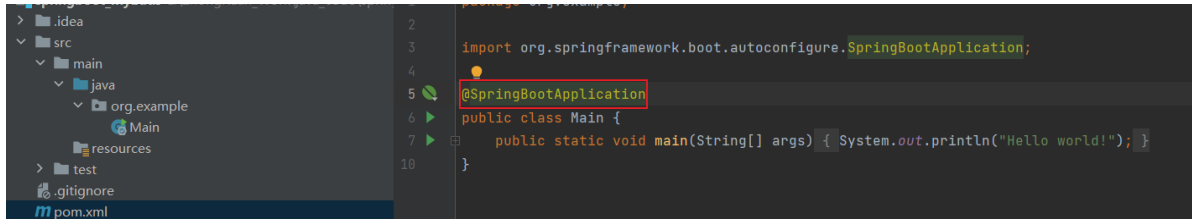
打开ideal右边栏Maven，进行打包处理，先双击clean，再双击package。



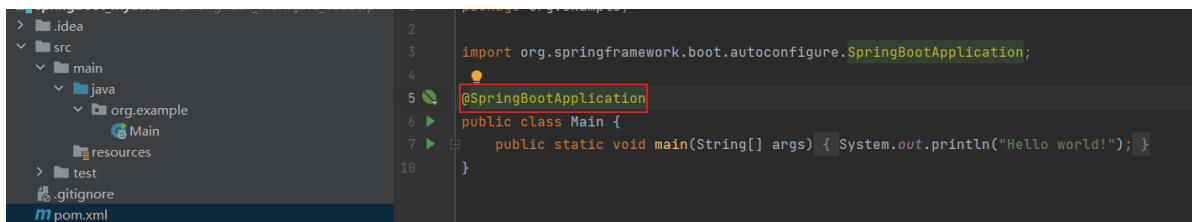
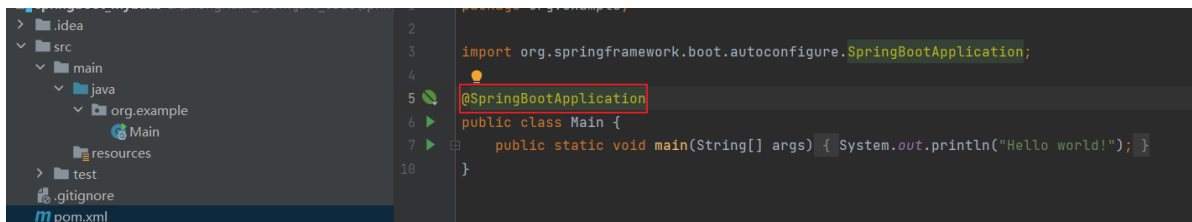
执行成功后，项目目录下生成打包好的jar文件。



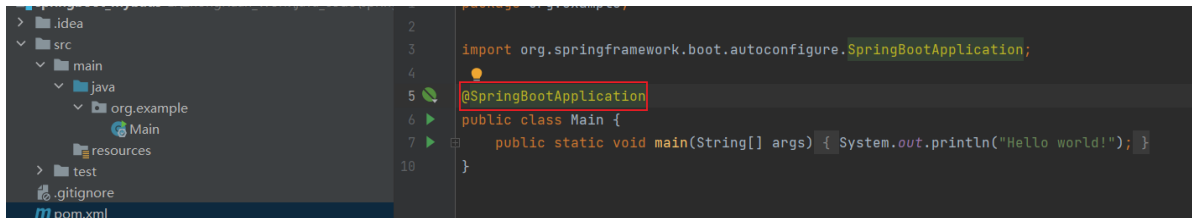
右键选择在文件中打开位置。



解压该文件后，复制META-INF文件，粘贴到MyConfig项目下的java目录下，并删除META-INF文件里的maven.org文件夹。



创建配置文件spring.factories，在META-INF文件下



在org.example文件下创建新类MyConfiguration，类内容如下

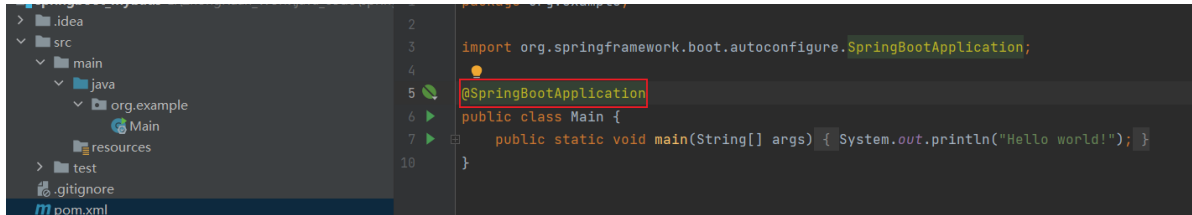
```
1 package org.example;
2
3 import org.springframework.context.annotation.Configuration;
4
5 @Configuration
6 public class MyConfiguration {
7     public MyConfiguration(){
8         System.out.println("我的配置工程");
9     }
10 }
11
```

然后去进行spring.factories文件的配置，关联MyConfiguration，配置内容如下。（\是换行符）

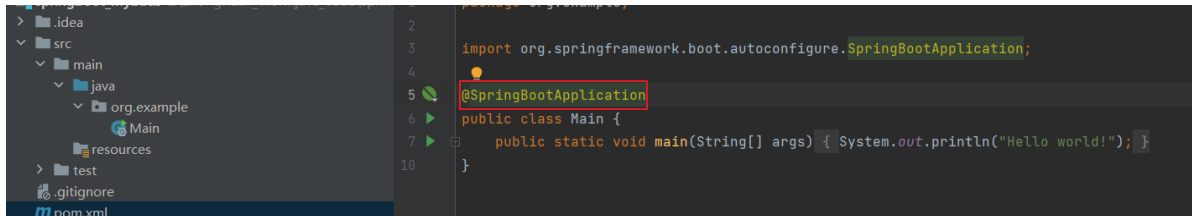
```
1 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
2 org.example.MyConfiguration
```

在MyConfig项目的pom.xml添加内容，之后再**打包**一次。

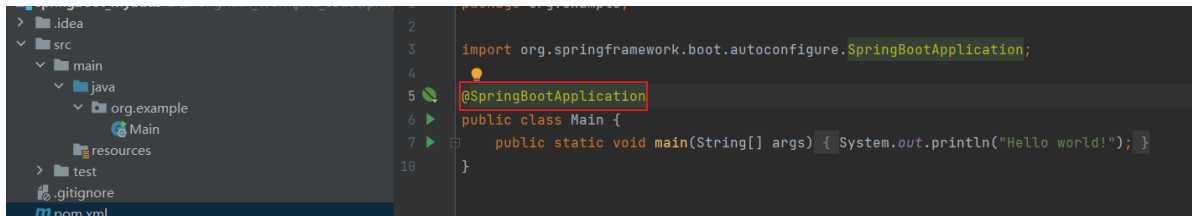
```
1      <build>
2          <resources>
3              <resource>
4                  <directory>src/main/java</directory>
5                  <includes>
6                      <include>META-INF/*</include>
7                  </includes>
8              </resource>
9          </resources>
10     </build>
```



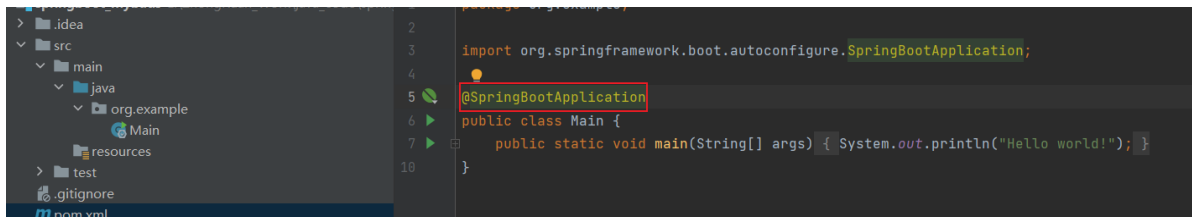
打包完成后，双击install，发布至本地仓库。



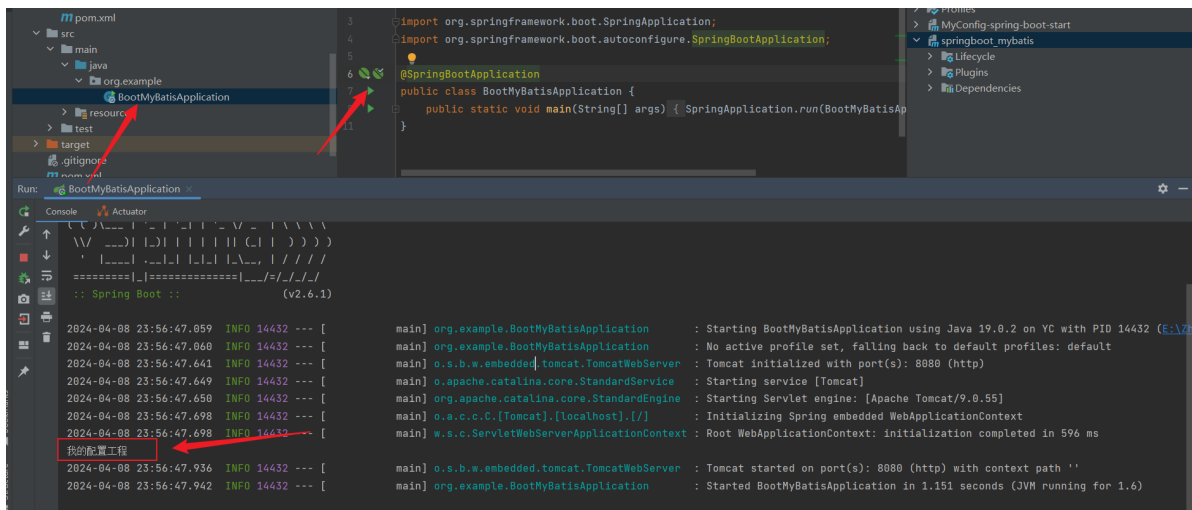
最后，来到外部项目的pom.xml文件，添加我们自定义的依赖。



删除外部项目的pom.xml文件中的该内容。



最后启动外部项目，打印出“我的配置过程”，配置成功。



4.2自定义properties

该部分内容不影响后续内容，可以不用做。

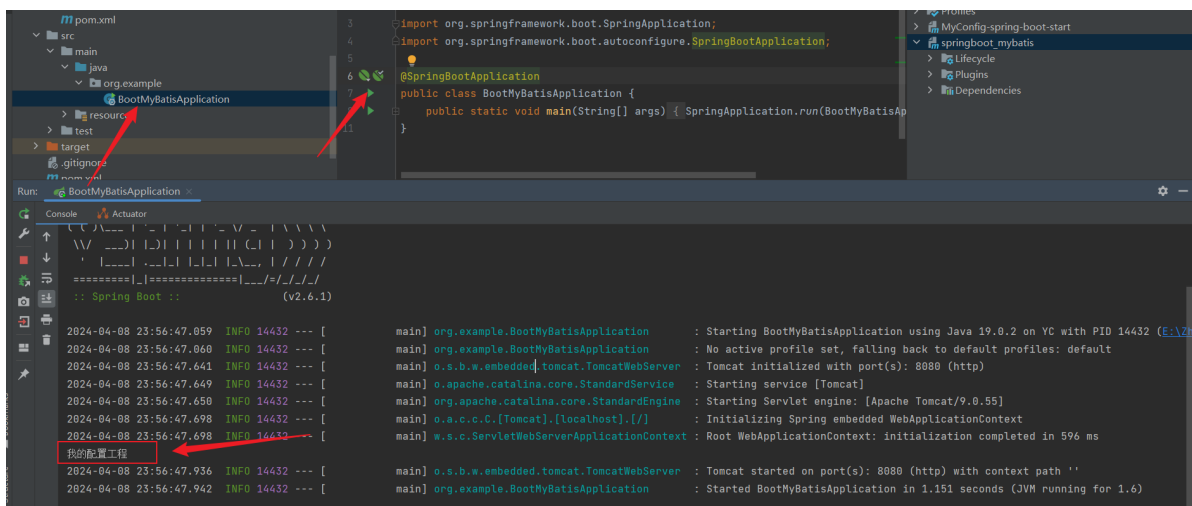
自定义 properties 文件通常用于存储应用程序的配置信息，这些信息可以在应用程序运行时被读取。Properties 文件通常是键值对的形式，键值对可以描述各种配置选项，例如数据库连接信息、日志配置、应用程序设置等等。

使用自定义 properties 文件的好处包括：

1. **分离配置信息**: 将配置信息从代码中分离出来，以便在不同的环境中进行配置。例如，在开发、测试和生产环境中，可以使用不同的 properties 文件来配置相应环境的设置。
2. **易于管理**: 将配置信息集中到一个文件中，使得配置管理更加方便。通过修改 properties 文件，可以轻松地更改配置而无需修改源代码。
3. **灵活性**: 可以动态地加载和修改 properties 文件，从而实现灵活的配置。这使得应用程序可以在不重启的情况下适应不同的配置。
4. **安全性**: 某些敏感信息可以放在 properties 文件中，例如密码或 API 密钥。这些信息可以通过加密或其他安全措施进行保护，而不需要硬编码在代码中。
5. **国际化和本地化**: 可以使用 properties 文件来存储与语言和地区相关的文本和其他资源，以实现国际化和本地化。

先将yml文件更改后缀名为properties，文件内配置以下内容

- 1 logging.charset.console=utf-8
- 2 spring.aop.proxy-target-class=false



- `logging.charset.console=utf-8``: 这个配置项指定了控制台输出日志时所使用的字符集。在这个例子中, 设置为 UTF-8, 这意味着日志信息将使用 UTF-8 编码输出到控制台, 以支持更广泛的字符集和特殊字符。
- `spring.aop.proxy-target-class=false``: 这个配置项用于控制 Spring AOP 是否使用基于类的代理 (CGLIB) 或基于接口的代理 (JDK 动态代理)。当设置为 `false` 时, Spring AOP 将尽可能使用基于接口的代理。这个配置项通常用于调优应用程序的性能和行为。

在内部类MyConfig中创建两个类

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

```

```

2024-04-08 23:56:47.059 INFO 14432 --- [main] org.example.BootMyBatisApplication : Starting BootMyBatisApplication using Java 19.0.2 on YC with PID 14432 (E:\A\2\
2024-04-08 23:56:47.060 INFO 14432 --- [main] org.example.BootMyBatisApplication : No active profile set, falling back to default profiles: default
2024-04-08 23:56:47.641 INFO 14432 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2024-04-08 23:56:47.649 INFO 14432 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-08 23:56:47.650 INFO 14432 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.55]
2024-04-08 23:56:47.658 INFO 14432 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-08 23:56:47.698 INFO 14432 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 596 ms
2024-04-08 23:56:47.936 INFO 14432 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2024-04-08 23:56:47.942 INFO 14432 --- [main] org.example.BootMyBatisApplication : Started BootMyBatisApplication in 1.151 seconds (JVM running for 1.6)

```

MyProp类, 写出属性, 通过生成器Generate生成Getter和Setter方法

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

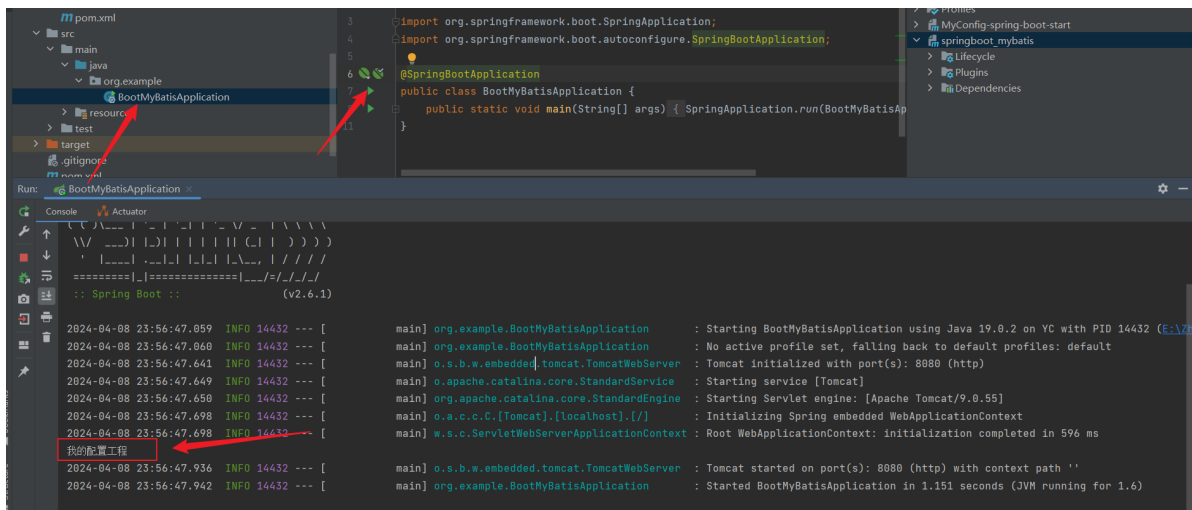
```

```

2024-04-08 23:56:47.059 INFO 14432 --- [main] org.example.BootMyBatisApplication : Starting BootMyBatisApplication using Java 19.0.2 on YC with PID 14432 (E:\A\2\
2024-04-08 23:56:47.060 INFO 14432 --- [main] org.example.BootMyBatisApplication : No active profile set, falling back to default profiles: default
2024-04-08 23:56:47.641 INFO 14432 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2024-04-08 23:56:47.649 INFO 14432 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-08 23:56:47.650 INFO 14432 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.55]
2024-04-08 23:56:47.658 INFO 14432 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-08 23:56:47.698 INFO 14432 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 596 ms
2024-04-08 23:56:47.936 INFO 14432 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2024-04-08 23:56:47.942 INFO 14432 --- [main] org.example.BootMyBatisApplication : Started BootMyBatisApplication in 1.151 seconds (JVM running for 1.6)

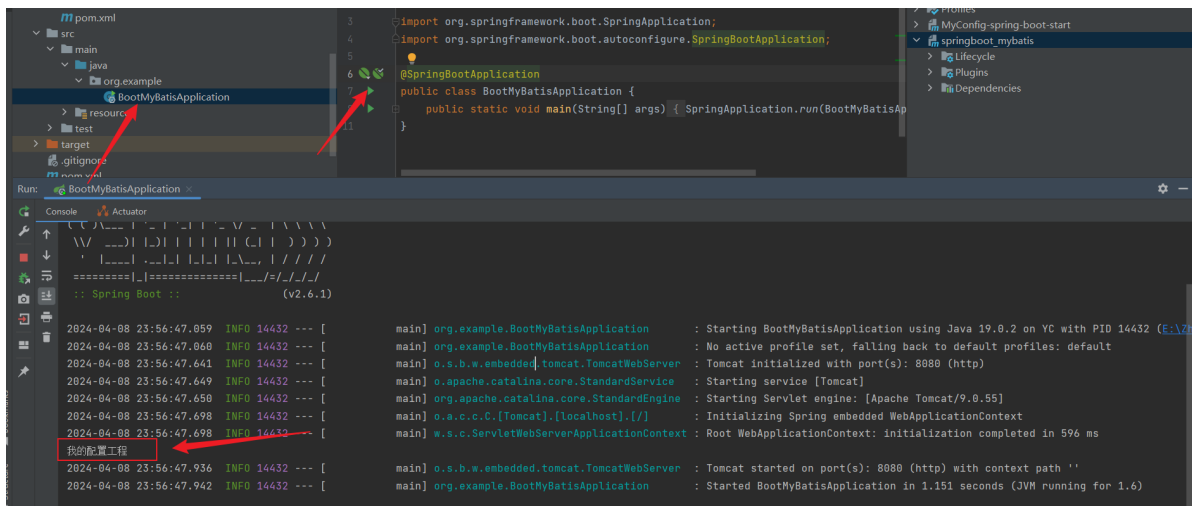
```

- `@ConfigurationProperties(prefix = "com.student")` 注解通常用于将配置文件中以特定前缀开头的属性值映射到Java对象的属性上
此处我将前缀设置为`com.student`, 那么取数据时, 形式为`com.student.name=?`
Student类, 生成属性的Getter和Setter方法。



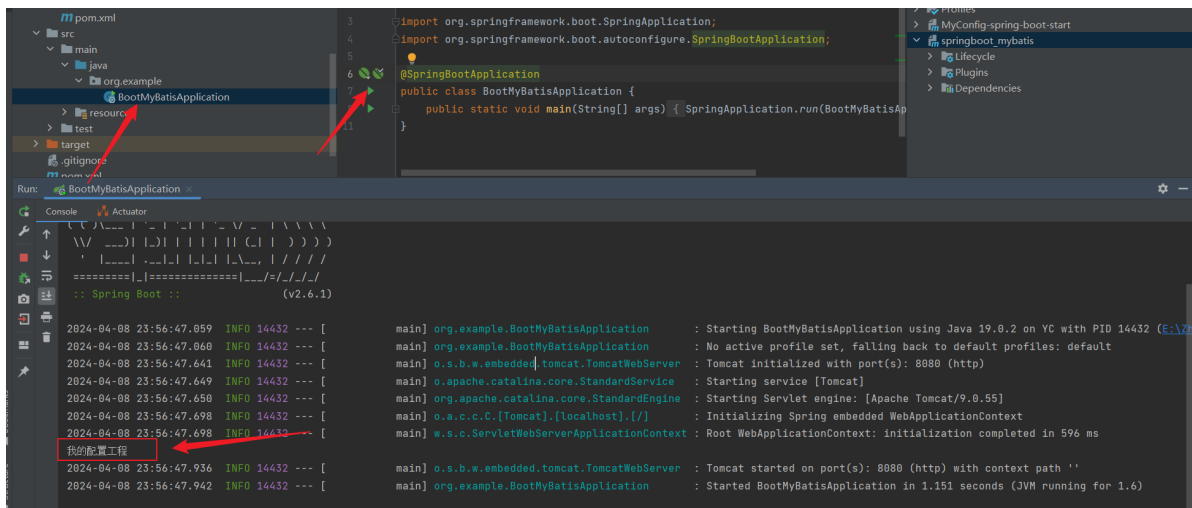
修改MyConfiguration类中内容

```
1 @Configuration
2 @EnableConfigurationProperties(value = MyProp.class)
3 public class MyConfiguration {
4     public MyConfiguration(){
5         System.out.println("我的配置工程");
6     }
7
8     @Bean
9     public Student initStudent(MyProp myProp){
10         return new Student(myProp.getName(), myProp.getAge(), 0);
11     }
12 }
```



完成后，重新打包并发布MyConfig项目。

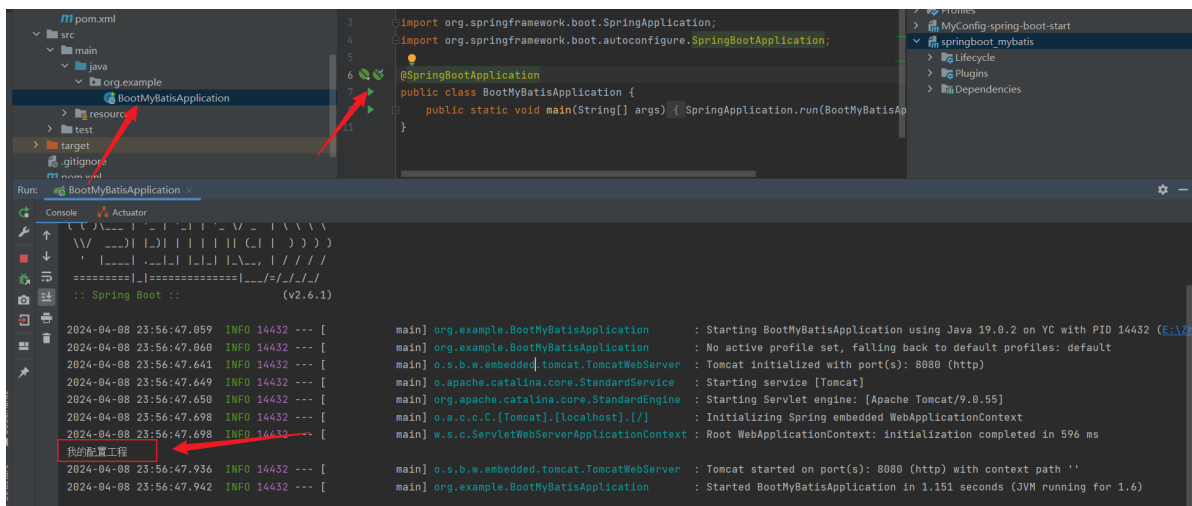
在application.yml中设置学生信息



编写测试类前，在外部项目的pom.xml中添加测试依赖

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-test</artifactId>
4   <scope>test</scope>
5 </dependency>
```

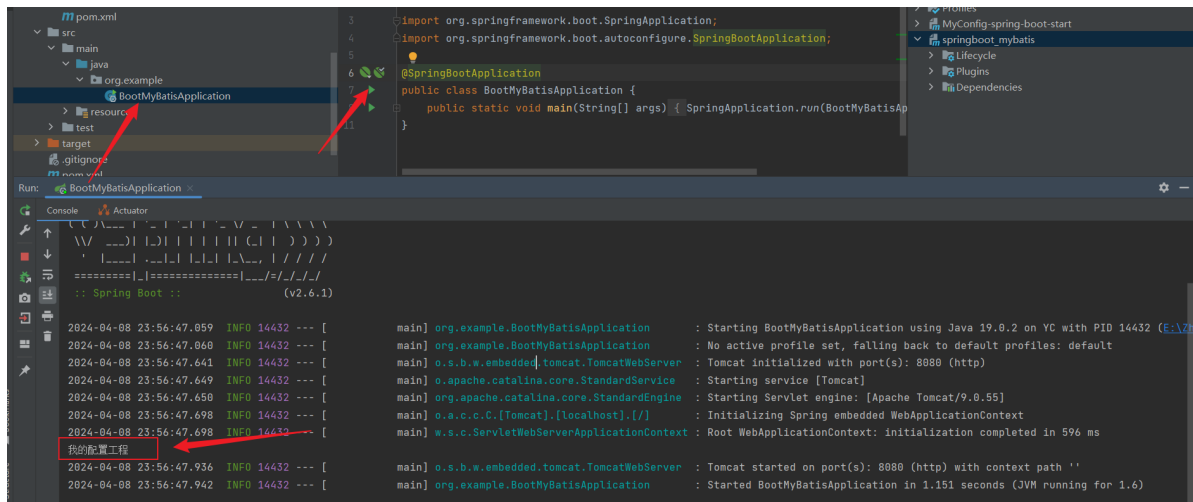
创建测试类，包名一定要对应。



测试方法如下

```
1 package org.example;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.boot.test.context.SpringBootTest;
5 import org.springframework.beans.factory.annotation.Autowired;
6
7 @SpringBootTest
8 public class BootApplicationTests {
9     @Autowired
10     Student student;
11
12     @Test
13     public void test1(){
14         System.out.println(student.getName() + "-0-0--" + student.getAge());
15     }
16 }
```

运行测试方法后，结果应显示设置的学生信息。



5.IOC注解

IOC (Inversion of Control, 控制反转) 是一种软件设计原则，它指的是将组件之间的控制权从代码本身转移到外部容器或框架中。在传统的程序设计中，通常由程序员在代码中直接创建并控制对象之间的关系和生命周期。而在IOC 模式中，这种控制权被颠倒过来了，即由容器负责创建和管理对象之间的依赖关系。

IOC 的核心思想是：**将对象之间的依赖关系交由容器来管理，而不是由对象自己来创建和管理依赖。**

IOC (Inversion of Control, 控制反转) 注解是用于实现依赖注入 (Dependency Injection, DI) 的一种方式。在 Spring 框架中，IOC 注解通常用于告诉 Spring 容器如何管理 Bean 的依赖关系。

Spring 框架提供了多种 IOC 注解，其中最常用的包括：

1. **@Autowired**: 用于自动装配 Bean 的依赖关系。通过在字段、构造方法、Setter 方法或者自定义方法上标注 **@Autowired** 注解，Spring 容器会自动将匹配类型的 Bean 注入到标注了 **@Autowired** 的地方。
2. **@Qualifier**: 与 **@Autowired** 注解一起使用，用于指定要注入的具体 Bean 的名称。当存在多个匹配类型的 Bean 时，可以通过 **@Qualifier** 注解来选择正确的 Bean 进行注入。
3. **@Resource**: 与 **@Autowired** 类似，用于标注需要注入的依赖关系。不同的是，**@Resource** 注解可以指定要注入的 Bean 的名称或者类型。
4. **@Inject**: 与 **@Autowired** 注解类似，也是用于自动装配 Bean 的依赖关系。不同的是，**@Inject** 注解是由 Java 规范定义的，而 **@Autowired** 是 Spring 框架提供的。

```

1 package org.example;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.boot.CommandLineRunner;
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7 import org.springframework.context.ApplicationContext;
8
9 @SpringBootApplication
10 public class BootMyBatisApplication implements CommandLineRunner {
11     public static void main(String[] args) {
12         SpringApplication.run(BootMyBatisApplication.class, args);

```

```

13     }
14
15     @Override
16     public String toString() {
17         return super.toString();
18     }
19
20     @Autowired
21     ApplicationContext applicationContext;
22
23
24     @Override
25     public void run(String... args) throws Exception {
26         String[] beannames = applicationContext.getBeanDefinitionNames();
27         for (String name: beannames){
28             // 根据名字获取Bean对象
29             Object o = applicationContext.getBean(name);
30             System.out.println(name + "@@" + o);
31         }
32     }
33 }

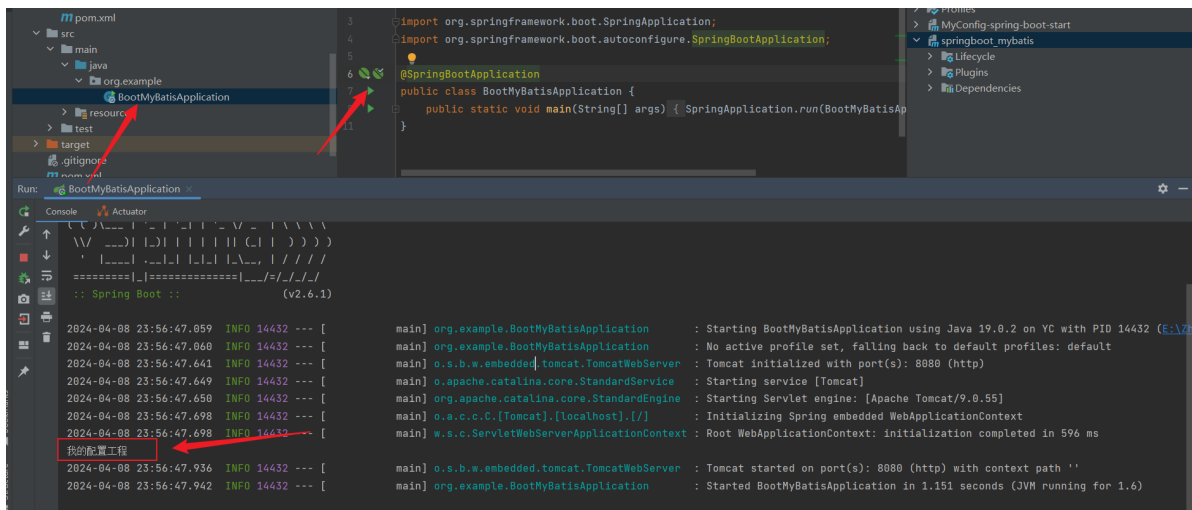
```

在这个例子中，我让启动类去实现CommandLineRunner接口，这个接口会生成run方法，在SpringBoot成功启动时，将会回调这个方法。容器通过@Autowired自动注入ApplicationContext（Bean对象）。在run方法中通过getBeanDefinitionNames()获取在容器中注册的所有Bean的名称，再创建对象，通过名称获取Bean对象，并打印。

打印结果如下，以@@作为分隔符，前面是Bean名称，后面是具体的Bean对象。

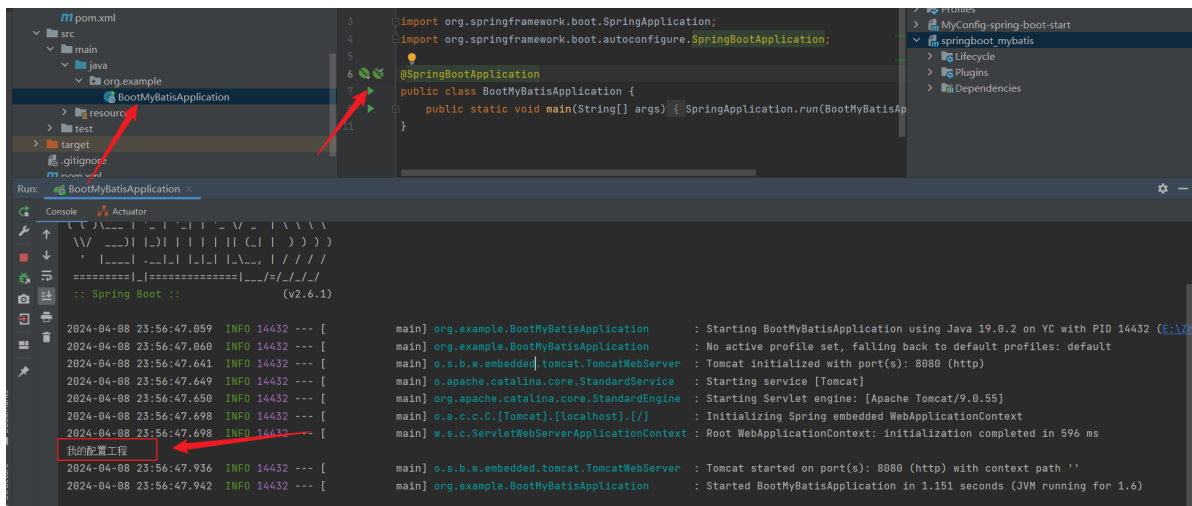
- Bean 名称列表 (beannames):
 - 包含了应用程序上下文中所有注册的 Bean 的名称。
 - 是一个字符串数组，每个元素都是一个唯一的 Bean 名称。
 - 通过这个列表，您可以获知应用程序上下文中存在哪些 Bean，但您无法直接访问 Bean 的具体实例。
- 具体的 Bean 对象 (o):
 - 通过 Bean 名称从应用程序上下文中获取到的具体的 Bean 实例。
 - 是一个对象实例，代表了应用程序中某个具体的组件或者服务。
 - 通过这个对象，您可以访问和操作 Bean 的属性和方法，进行各种操作，比如调用方法、设置属性等。

这两者的区别在于，Bean 名称列表只是一个存储了所有 Bean 名称的列表，而具体的 Bean 对象是根据名称从容器中获取到的实际的 Bean 实例。通过 Bean 名称列表，可以了解应用程序中存在哪些 Bean，而通过具体的 Bean 对象，您可以直接操作和使用这些 Bean。

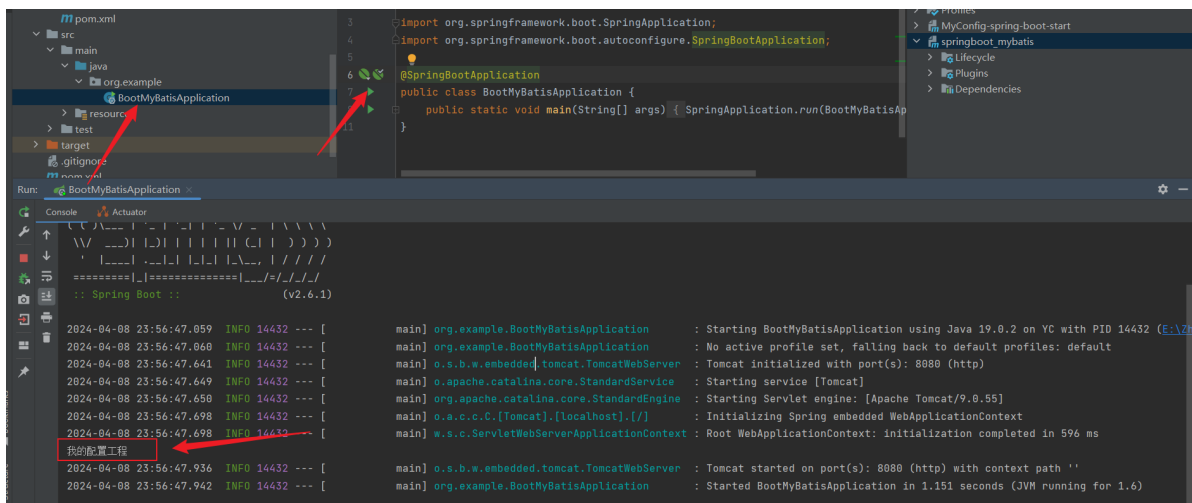


比如这个，比较明显的区别Spring容器中注册的Bean对象taskExecutorBuilder，它具体的实例是org.springframework.boot.task.TaskExecutorBuilder@33634f04。

按住Ctrl点击getBean，查看底层代码。接口BeanFactory就是一个管理和控制 Bean 对象的容器，负责创建、获取和管理 Bean 实例。



再继续向下查看，会看到Map类型的beans，Map（映射）是一种抽象数据类型，用于存储键值对（key-value pairs）。每个键都唯一地映射到一个值。Map 提供了基于键的快速查找操作，可以根据给定的键获取相应的值。



```

1 public class Mapintroduce {
2     public static void main(String[] args) {
3         // Node数组16, 所有节点达到0.75的时候扩容 每次自增一倍
4         // static final float DEFAULT_LOAD_FACTOR = 0.75f;
5         // Node存对象 存为单向链表
6         // Node存对象到8个时, 变成红黑树
7         // 红黑树节点个数到6时, 转置为单向链表
8         Map<String, String> map = new HashMap<>();
9         // key值唯一, key相同时, 覆盖value
10        map.put("010", "北京");
11        map.put("011", "上海");
12        map.put("011", "广州");
13
14        map.forEach((k, v) -> {
15            System.out.println(k + "@" + v);
16        });
17    }
18 }

```

Map<KeyType,ValueType>,定义变量时, 定义变量类型。

HashMap 的 put 方法用于向 HashMap 中添加键值对。当调用 `map.put("010", "北京")` 时, 它会计算键 "010" 的哈希值, 并将键值对存储在相应的位置上。如果在存储过程中, 发现该位置已经存在键值对, HashMap 会根据具体的情况选择覆盖已有值或者以链表或红黑树的形式存储新值。

- 哈希值 (Hash Value), 也称为散列值或摘要, 是将任意长度的输入数据通过哈希函数计算得到的固定长度的输出值。哈希值是一种对数据的高度压缩和唯一标识的表示方式, 通常用于快速地比较大量数据的唯一性。

哈希值具有以下特点:

1. **固定长度**: 无论输入数据的长度是多少, 哈希函数都会将其转换为固定长度的哈希值。这意味着无论输入数据的大小如何, 哈希值的长度始终保持不变。
2. **唯一性**: 理想情况下, 不同的输入数据应该产生不同的哈希值。这样可以通过比较哈希值来确定数据是否相同, 从而实现快速的查找和比较。
3. **散列性**: 哈希函数应该能够将输入数据均匀地分布到哈希空间中, 使得哈希值的计算结果具有良好的散列性。这样可以最大程度地避免哈希冲突, 即不同的输入数据产生相同的哈希值。

哈希值在计算机领域中有广泛的应用, 例如:

- 数据库中的索引: 数据库索引通常使用哈希值来快速查找数据。
- 数据校验: 哈希值可以用于校验数据的完整性, 例如文件的校验和。
- 加密和安全: 哈希函数在密码学中也有重要的应用, 例如密码存储和数字签名等。

查看HashMap底层代码可以看到设定了阈值0.75f。

```

1 static final float DEFAULT_LOAD_FACTOR = 0.75f;

```

- **HashMap 的扩容机制**: HashMap 在内部使用数组存储键值对, 当存储的键值对数量达到一定阈值时, HashMap 会进行扩容操作。HashMap 的默认加载因子是 0.75f, 表示当 HashMap 中的实际元素数量超过数组长度的 75% 时, 就会触发扩容。每次扩容都会将数组长度扩大一倍, 并且重新计算每个键值对的存储位置

6.DI注解

依赖注入（Dependency Injection，DI）是一种设计模式，它用于解耦组件之间的依赖关系。在Java中，Spring框架提供了一系列的注解来实现依赖注入。以下是一些常用的DI注解：

1. @Bean：

@Bean 注解通常用于配置类中的方法上，通过 @Bean 注解标注声明的一个Bean对象。当Spring启动时，会自动扫描配置类，并将标注了 @Bean 注解的方法返回的对象注册到容器中。

2. @Autowired：

@Autowired 注解用于自动装配Bean，它可以标注在构造方法、成员变量、方法或者参数上。Spring会自动在容器中查找匹配的Bean，并将其注入到标记了 @Autowired 的地方。

3. @Qualifier：

@Qualifier 注解通常和 @Autowired 注解一起使用，用于指定要注入的Bean的限定符。当容器中存在多个匹配的Bean时，可以通过 @Qualifier 来指定具体要注入的Bean。

4. @Resource：

@Resource 注解是Java EE提供的依赖注入注解，它可以用于注入由 @Autowired 和 @Qualifier 注解注入的Bean，也可以用于注入其他资源，比如JNDI资源。

5. @Inject：

@Inject 注解是Java规范中定义的依赖注入注解，它与 @Autowired 注解类似，用于标注要注入的Bean。@Inject 注解提供了更多的灵活性，支持在构造方法、成员变量、方法以及参数上进行注解。

6. @Value：

@Value 注解用于从配置文件中读取属性值，并将其注入到Bean中。可以通过 \${} 语法从配置文件中引用属性值，也可以通过 #{} 语法引用Spring的表达式语言（SpEL）。

7.单例模式

SpringBoot中的所有Bean默认是单实例。

单例模式是一种设计模式，它确保某个类只能创建一个实例，并提供一个全局访问点来获取该实例。在单例模式中，类的构造方法是私有的，所以外部无法直接实例化该类，而是通过类的静态方法来获取实例。

单例模式的主要特点包括：

1. **私有构造方法**：单例类的构造方法被私有化，以防止外部代码直接实例化该类。
2. **静态方法获取实例**：单例类通常提供一个静态方法来返回类的唯一实例。这个静态方法会在首次调用时创建实例，并在后续调用时返回相同的实例。
3. **全局访问点**：通过单例类提供的静态方法来获取实例，确保在整个应用程序中只有一个实例存在。
4. **延迟加载**：有些单例实现会延迟实例化，即只有在首次调用时才创建实例，而不是在类加载时就创建实例。

单例模式的应用场景包括：

- 线程池
- 配置文件管理器
- 日志记录器
- 数据库连接池
- 缓存管理器

使用单例模式可以确保某些资源在整个应用程序中只有一个实例，从而避免了资源的重复创建和消耗，提高了系统的性能和效率。然而，需要注意单例模式可能会导致全局变量的存在，增加了代码的耦合性，因此需要谨慎使用。

"懒汉式" (Lazy Initialization) 和"饿汉式" (Eager Initialization) 是两种常见的单例模式实现方式，它们在实例化单例对象的时机上有所不同。

1. 懒汉式 (Lazy Initialization) :

- 懒汉式是指在首次调用获取单例实例的方法时才进行实例化。在多线程环境下，懒汉式需要考虑线程安全性，以避免多个线程同时调用获取实例方法时创建多个实例的问题。通常采用双重检查锁 (Double-Checked Locking) 等方式来实现线程安全的懒加载单例模式。

2. 饿汉式 (Eager Initialization) :

- 饿汉式是指在类加载时就进行实例化，无论是否会使用该实例。在类加载时，单例对象就已经被创建，因此不存在多线程下的线程安全问题。但是这也意味着在应用程序启动时，就会创建单例实例，可能会占用一定的系统资源，且可能不会被立即使用。

简而言之，懒汉式是一种懒加载的单例模式，只有在需要时才进行实例化，而饿汉式则是一种提前初始化的单例模式，在类加载时就创建实例。

```
1  // 单例模式
2  public class SingletonPattern {
3
4  // 饿汉式
5      private SingletonPattern(){
6
7      }
8
9      private static SingletonPattern singletonPattern;
10
11 // 懒汉式
12 // synchronized :线程同步锁
13 public static synchronized SingletonPattern newInstance(){
14     if (singletonPattern == null){
15         singletonPattern = new SingletonPattern();
16     }
17     return singletonPattern;
18 }
19
20 public static void main(String[] args) {
21 // 输出false
22     SingletonPattern s1 = new SingletonPattern();
23     SingletonPattern s2 = new SingletonPattern();
24     System.out.println(s1 == s2);
25
26 // 输出true
27     SingletonPattern s3 = SingletonPattern.newInstance();
28     SingletonPattern s4 = SingletonPattern.newInstance();
29     System.out.println(s3 == s4);
30 }
31 }
```

比如对于两个对象的比较判断 (`s1 == s2`)，因为都通过new方法，开辟了新的空间，地址发生变化，地址不同判断为false，但是我希望它们是同一个地址，就可以使用懒汉式，使用时再实例化。

对于单实例可以使用@Scope注解，@Scope 是 Spring 框架中用于指定 Bean 的作用域的注解。

@Scope 注解常用的作用域包括：

1. **singleton**：单例作用域，表示在整个应用程序中只创建一个 Bean 实例，默认值。
2. **prototype**：原型作用域，每次注入或获取 Bean 时都会创建一个新的 Bean 实例。
3. **request**：每个 HTTP 请求都会创建一个新的 Bean 实例，适用于 Web 应用中。
4. **session**：每个 HTTP Session 都会创建一个新的 Bean 实例，适用于 Web 应用中。
5. **application**：在 ServletContext 中存活的时间，整个应用程序生命周期内只创建一个 Bean 实例。

```
1 package org.example.SingletonPattern;
2
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.context.annotation.Scope;
5
6 @Configuration
7 // 参数prototype使得单实例创建时会是新的Bean
8 @Scope("prototype")
9 public class Test2 {
10
11 }
12
```

8.1 面向切面编程

8.1 什么是面向切面编程（AOP）？

- 面向切面编程（Aspect-Oriented Programming, AOP）是一种软件开发的编程范式，旨在解决横切关注点（Cross-Cutting Concerns）的问题。
- 横切关注点是指那些跨越多个模块、方法或对象的功能，如日志记录、事务管理、安全检查等。

8.2 AOP 的核心概念

Spring AOP官方文档：<https://docs.spring.io/spring-framework/reference/core/aop/introduction-defn.html>

- **切面（Aspect）**：用于封装横切关注点的模块化单元，定义了在哪里、何时以及如何应用横切关注点的逻辑。
- **连接点（Join Point）**：在程序执行过程中可以应用切面的点，通常是方法调用或异常抛出的位置。在 Spring AOP 中，连接点始终代表方法执行。
- **切点（Point cut）**：切点是用来描述连接点的规则或者表达式，以确定哪些连接点应该被拦截和应用切面逻辑。切点实际上是连接点的过滤器，它定义了程序执行中哪些连接点应该被关注。切点描述了应该在何处应用横切关注点，而连接点是实际的执行点。连接点是切点匹配的结果，表示程序执行中的具体位置。所以，切点描述了应该拦截的连接点，而连接点是切点匹配的结果，是程序执行中的具体位置。
- **通知（Advice）**：切面在连接点处执行的动作，包括前置通知、后置通知、返回通知、异常通知和环绕通知等。
- **织入（Weaving）**：将切面的逻辑应用到目标对象的过程，可以在编译期、类加载期、运行时等不同的阶段进行。

8.2 AOP 的核心概念理解

1. **切面 (Aspect)** : 切面是通知和切点的结合, 它包含了一组通知和切点。切面定义了在哪里以及何时应该执行通知。比如, 可能有一个日志切面, 其中包含了在特定切点上执行的日志记录通知。
2. **切点 (Point cut)** : 切点就像是在道路上设置的路标或者标志, 告诉你在哪些地点应该停下来做某些事情。比如设置了一个标志, 表示在所有的红色路口停下来做记录。这个标志就是一个切点, 它告诉你在哪些连接点上应用你的日志记录。
3. **连接点 (Join Point)** : 连接点就像是程序中的特定地点, 比如方法调用、异常抛出等。想象一下程序是一条道路, 连接点就是你能够停下来做一些事情的具体地点, 比如路口、加油站等。
4. **通知 (Advice)** : 通知是在特定切点上执行的操作, 它定义了连接点上要执行的逻辑。通知是横切关注点的实际实现, 可以在连接点之前、之后或者环绕执行。
5. **织入 (Weaving)** : 织入是把切面应用到目标对象上并创建代理对象的过程。这意味着在程序运行时, AOP 框架将切面中的通知织入到应用程序的目标对象中, 以实现横切关注点的功能。

我们要使用的操作或逻辑 (通知 (Advice)), 执行这个操作或逻辑的条件 (切点 (Point cut)), 使用这个条件匹配的对象 (连接点 (Join point)), 以上封装为一个模块单元 (切面 (Aspect)), 将这个模块单元使用在别处的过程 (织入 (Weaving))

8.3 AOP 的实现方式

- **基于代理的 AOP**: 通过动态代理机制在运行时生成代理对象, 并将切面的逻辑织入到目标对象的方法调用中。
- **基于字节码的 AOP**: 通过字节码增强技术在编译期或类加载期修改目标类的字节码, 实现切面的织入。

8.4 AOP 的应用场景

- **日志记录**: 记录方法的调用信息、参数和返回值等。
- **事务管理**: 管理数据库事务的开始、提交和回滚等。
- **权限控制**: 对用户请求进行权限验证和授权处理。
- **性能监控**: 监控方法的执行时间、资源消耗等。
- **异常处理**: 捕获和处理方法执行过程中的异常。

8.5 AOP 的优点

- **解耦和模块化**: 将横切关注点从核心业务逻辑中分离出来, 提高了代码的模块化和可维护性。
- **重用性**: 将横切关注点封装成切面后, 可以在多个模块中共享和重用。
- **集中管理**: 统一管理横切关注点的逻辑, 降低了重复代码的数量和复杂度。
- **横向功能扩展**: 可以在不修改原有代码的情况下, 通过增加新的切面来实现对系统功能的扩展。

8.6 AOP 的实现框架

- **Spring AOP**: 基于代理的 AOP 实现, 集成在 Spring 框架中, 提供了丰富的功能和灵活的配置方式。
- **AspectJ**: 基于字节码的 AOP 实现, 提供了更强大的切面编程功能, 但相对于 Spring AOP 更复杂。

8.7 AOP 的应用示例

首先在pom文件导入AOP依赖

```
1      <!-- AOP依赖 -->
2      <dependency>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter-aop</artifactId>
5      </dependency>
```

创建AOP类，实现日志功能测试。

```
1 package org.example.AOP;
2
3 import org.aspectj.lang.annotation.*;
4 import org.springframework.stereotype.Component;
5
6 // 切面
7 @Aspect
8 @Component
9 public class LoggingAspect {
10
11     // 通知
12     // execution(* org.example.service.*.*(..)) -> 切点的定义式，定义了匹配的连接点
13     @Before("execution(* org.example.service.*.*(..))")
14     public void logBefore() {
15         System.out.println("Method execution is about to start.");
16     }
17
18     @AfterReturning(pointcut = "execution(* org.example.service.*.*(..))",
19     returning = "result")
20     public void logAfterReturning(Object result) {
21         System.out.println("Method executed successfully. Result: " +
22     result);
23     }
24
25     @AfterThrowing(pointcut = "execution(* org.example.service.*.*(..))",
26     throwing = "exception")
27     public void logAfterThrowing(Exception exception) {
28         System.out.println("Method threw an exception: " +
29     exception.getMessage());
30     }
31 }
```

在面向切面编程中，有几个常用的注解，它们用于定义切面、切点和通知等方面：

1. **@Aspect**：用于定义切面的注解，通常与其他通知注解一起使用。
2. **@Pointcut**：用于定义切点的注解，指定一组连接点，通常与其他通知注解一起使用。
3. **@Before**：前置通知，表示在目标方法执行之前执行的通知。
4. **@AfterReturning**：后置通知，表示在目标方法正常返回后执行的通知。
5. **@AfterThrowing**：异常通知，表示在目标方法抛出异常后执行的通知。
6. **@After**：最终通知，表示无论目标方法是否正常返回或抛出异常都会执行的通知。
7. **@Around**：环绕通知，表示在目标方法执行前后都执行的通知，可以控制目标方法的执行流程。

在测试类中测试方法

```
1  /*测试AOP日志记录*/
2      @Autowired
3      MyService myService;
4
5      @Test
6      public void testLoggingAspect() {
7          String result = myService.doSomething();
8          System.out.println("现在的result = " + result);
9          assertEquals("我要干饭", result) ;
10     }
```

测试结果如下

```
2024-05-05 23:05:15.935 INFO 7364 --- [          main] org.example.BootApplicationTests : Starting BootApplicationTests using Java
2024-05-05 23:05:15.936 INFO 7364 --- [          main] org.example.BootApplicationTests : No active profile set, falling back to de
我的配置工程
2024-05-05 23:05:17.607 INFO 7364 --- [          | main] org.example.BootApplicationTests : Started BootApplicationTests in 1.937 sec
Method execution is about to start.
Method executed successfully. Result: 我要干饭
现在的result = 我要干饭
```

调用service包下任何方法的前置通知，成功打印：**Method execution is about to start.**。接着是调用方法执行成功的后置通知，成功打印，并返回服务里的result的值：**Method executed successfully. Result: 我要干饭。**

测试异常通知，如果方法抛出异常。异常通知打印异常信息

```
Method threw an exception: 抛出一个异常，触发异常通知! result_test值发生变更!

java.lang.RuntimeException: 抛出一个异常，触发异常通知! result_test值发生变更!

at org.example.service.impl.MyServiceImpl.doSomething(MyServiceImpl.java:19) <2 internal lines>
at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:344)
at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethodInvocation.java:189)
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:166)
at org.springframework.aop.aspectj.AspectJAfterThrowingAdvice.invoke(AspectJAfterThrowingAdvice.java:64)
```

9.代理模式

当谈到代理模式时，我们通常指的是结构型设计模式中的一种，用于控制对其他对象的访问。代理模式通过创建一个代理对象来控制对原始对象的访问，从而可以在访问对象时添加额外的功能。

代理模式的主要参与者包括：

1. **抽象主题 (Subject)**：定义了代理和真实主题（被代理对象）的共同接口，客户端通过抽象主题来访问真实主题。
2. **真实主题 (Real Subject)**：实现了抽象主题接口，是被代理的对象，也就是真正执行业务逻辑的对象。
3. **代理 (Proxy)**：实现了抽象主题接口，并保存了对真实主题的引用。代理对象控制对真实主题的访问，并在访问真实主题时可以添加额外的功能。

代理模式可以分为以下几种类型：

1. **静态代理**：在编译时已经确定代理关系的代理方式。代理对象直接实现了与真实主题相同的接口，并在调用真实主题方法前后执行额外的逻辑。
2. **动态代理**：在运行时动态创建代理对象的代理方式。代理对象不需要实现与真实主题相同的接口，而是通过Java动态代理机制（`java.lang.reflect.Proxy`类）或者第三方库（如CGLIB）来动态生成代理对象。

3. **远程代理**：用于控制访问远程对象的代理方式。远程代理在客户端和服务端之间建立一个通信连接，并通过网络传输数据来访问远程对象。
4. **虚拟代理**：延迟加载真实主题的代理方式。虚拟代理在访问真实主题之前先进行一些预处理，例如加载大型资源或者计算高昂的开销。
5. **保护代理**：控制对真实主题的访问权限的代理方式。保护代理根据访问者的权限来决定是否允许访问真实主题。

代理模式的优点包括：

- **降低耦合度**：代理模式可以在客户端和真实主题之间建立一层间接层，降低了客户端与真实主题之间的直接依赖关系。
- **增强扩展性**：代理模式使得可以在不修改真实主题的情况下扩展其功能，只需要修改代理即可。
- **实现懒加载**：某些代理模式（如虚拟代理）可以实现懒加载，延迟加载真实主题的创建和初始化。

代理模式的缺点包括：

- **增加复杂度**：引入代理对象增加了系统的复杂度，需要额外的代码来管理代理对象的创建和使用。
- **性能损耗**：在使用动态代理时，代理对象的创建和调用可能会引入一定的性能损耗。
- **可能引入单点故障**：在使用远程代理时，代理服务器可能成为系统的单点故障，导致系统的可靠性降低。

接下来我们仅讨论静态加载与动态加载。代码示例在proxy包下

9.1 静态加载

特点：

- 在编译时已经确定代理关系，代理类和真实主题（被代理对象）之间的关系在编译期就已经确定。
- 代理对象直接实现了与真实主题相同的接口，或者继承了真实主题相同的父类。

实现方式：

1. 定义一个抽象主题接口（Subject），包含真实主题和代理类共同的行为方法。
2. 定义一个真实主题类（RealSubject），实现抽象主题接口，是被代理的对象。
3. 定义一个代理类（Proxy），实现抽象主题接口，内部保存对真实主题的引用，并在调用真实主题方法前后执行额外的逻辑。

优点：

- 易于理解和实现，静态代理的设计思路简单明了。
- 编译时确定代理关系，可以提前做一些静态优化和检查。

缺点：

- 每个代理类只能代理一个真实主题对象，如果需要代理多个真实主题对象，就需要创建多个代理类，导致类的数量增加。

静态代理适用于代理对象数量有限且在编译期间已经确定的情况。

9.2 动态加载

特点：

- 在运行时动态生成代理对象，不需要提前知道代理关系。
- 代理对象不需要实现与真实主题相同的接口，而是通过 Java 动态代理机制或者第三方库来动态生成代理对象。

实现方式：

- 1.使用Java动态代理机制：通过 `java.lang.reflect.Proxy` 类和 `InvocationHandler` 接口来创建代理对象。
- 2.使用第三方库，如CGLIB：通过继承或者代理目标类来创建代理对象。

优点：

- 不需要手动创建代理类，可以节省大量的编码工作。
- 可以动态地为目标对象添加额外的功能，不需要修改目标对象的源代码。

缺点：

- 动态代理的实现相对复杂，需要了解Java反射机制和动态代理的工作原理。
- 运行时生成代理对象可能会带来一定的性能开销。

动态代理适用于代理对象数量较多或者代理关系在运行时才能确定的情况。

10.Spring MVC

Spring MVC (Model-View-Controller) 是基于Spring框架的一种轻量级的Web框架，用于构建Web应用程序。它采用了经典的MVC设计模式，将应用程序分为模型 (Model)、视图 (View) 和控制器 (Controller) 三个部分，以提高应用程序的模块化、可维护性和可扩展性。

- 模型 (Model)：

模型代表应用程序中的业务逻辑和数据，负责处理应用程序的数据操作、业务逻辑和状态管理。在Spring MVC中，模型通常表示为POJO (Plain Old Java Object) 类，它们可以是普通的Java类、实体类、DTO (Data Transfer Object) 等。

- 视图 (View)：

视图是用户界面的呈现方式，负责将模型中的数据呈现给用户。在Spring MVC中，视图通常表示为JSP (JavaServer Pages)、Thymeleaf模板、FreeMarker模板等。视图负责将模型中的数据动态地渲染为HTML等格式，并呈现给用户。

- 控制器 (Controller)：

控制器负责接收用户的请求、处理用户输入、调用模型中的业务逻辑、准备模型数据、选择合适的视图并将模型数据传递给视图。在Spring MVC中，控制器通常表示为带有 `@Controller` 注解的类，其中的方法用于处理不同的URL请求。

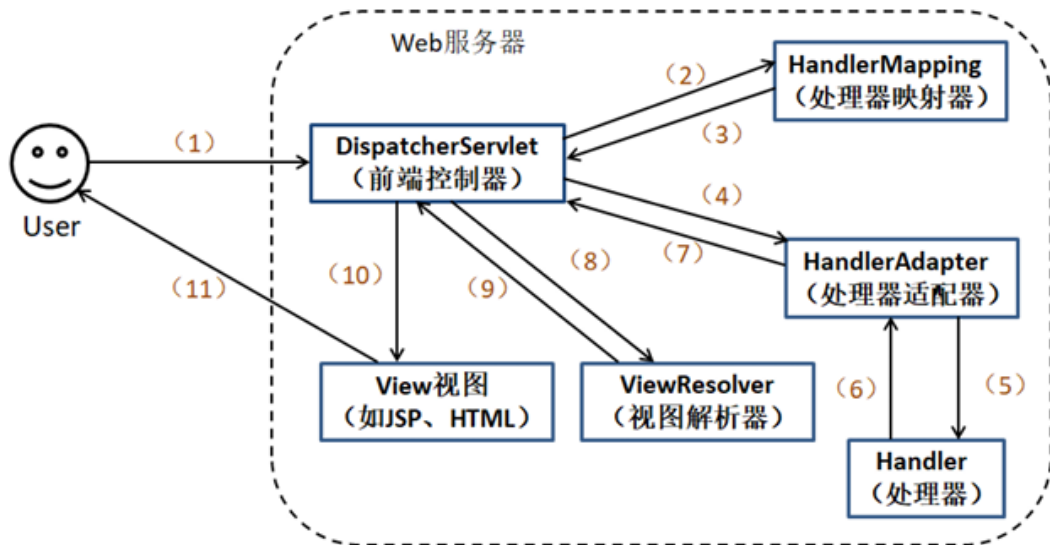
控制层框架：嵌入式的tomcat spring-web spring-webmvc

Spring MVC的工作流程：

1. **客户端发起请求**：用户在浏览器中输入URL或者点击链接，发送HTTP请求到服务器。
2. **前端控制器 (DispatcherServlet) 接收请求**：Spring MVC应用程序的前端控制器 (DispatcherServlet) 接收到请求，并将其转发给合适的处理器 (Controller)。
3. **处理器 (Controller) 处理请求**：处理器 (Controller) 接收到请求后，根据请求的URL映射到相应的处理方法，并执行业务逻辑，准备模型数据。
4. **处理器返回模型和视图**：处理器将模型数据传递给视图，选择合适的视图进行渲染。
5. **视图呈现页面**：视图接收到模型数据后，将其渲染为HTML等格式，并将结果返回给客户端 (浏览器)。
6. **客户端接收响应**：客户端 (浏览器) 接收到响应后，渲染页面并呈现给用户。

Spring MVC的工作原理

官方文档: <https://docs.spring.io/spring-framework/reference/index.html>



这图找不到哪里薅过来的了(๑' ̄ `๑)

1. 用户通过客户端向服务器发送请求，请求被前端控制器 DispatcherServlet 所拦截；

- 什么是请求：

<http://localhost:8080/hello.action?sname=tom>

http:// 协议

localhost: : ip 主机地址

: 8080 端口---对应程序

<http://localhost:8080/> 请求根目录

hello.action : 请求

?sname=tom&age=10; get 请求的参数列表

- DispatcherServlet: 调度员

- 一个请求 映射到 某个类的某个方法上

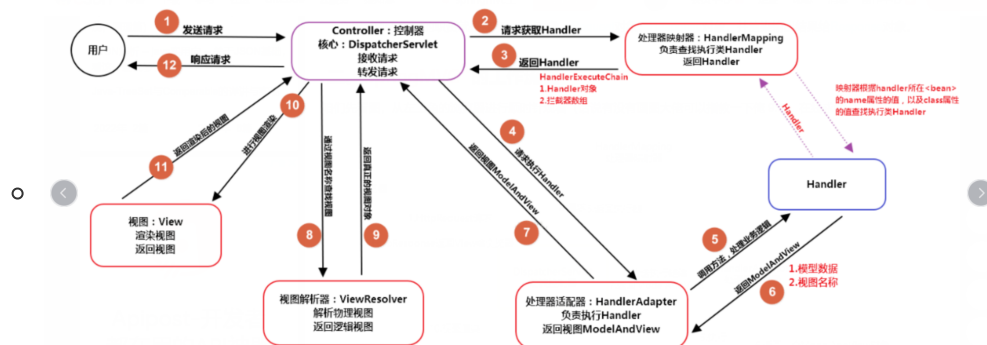
2. 前端控制器请求 HandlerMapping 查找 Handler 【可以根据 xml 配置、注解进行查找】

- HandlerMapping: 接口

- 找到: 具体类是 RequestMappingHandlerMapping
 - @RequestMapping 注解 查找 controller 中的方法，这个方法就是 Handler
 - 根据请求 `hello.action` 找到 HelloController 中 hello 方法
- > handler

3. 处理器映射器根据前端控制器发来的用户的请求找到对应符合的控制器 (Handler), 并且将其封装成处理器执行链, 返回给前端控制器。

- HandlerExecuteChain: 处理器执行链: 拦截器的位置
- 大佬的工作流程图片: [SpringMVC 工作流程 \(超级详细版\) - CSDN 博客](#)



CSDN @小强要亮头

4. 处理器适配器接收到来自前端控制器的执行链后，找到对应执行此执行链的处理器适配器（HandlerAdapter）来调用的具体的控制器（就是说其对应的方法或者逻辑）

◦ requestMappingHandlerAdapter:

- 国际化
- 参数类型转换
- handler方法入参

5. 控制器执行完成后，会返回一个ModelAndView对象给处理器适配器。

- 没有 ModelAndView了。 因为使用@ResponseBody
- @ResponseBody: 把返回对象按 json 解析，不走视图解析器了，下面的步骤直接跳过，到最后一步。把json返回给 请求的响应：response。

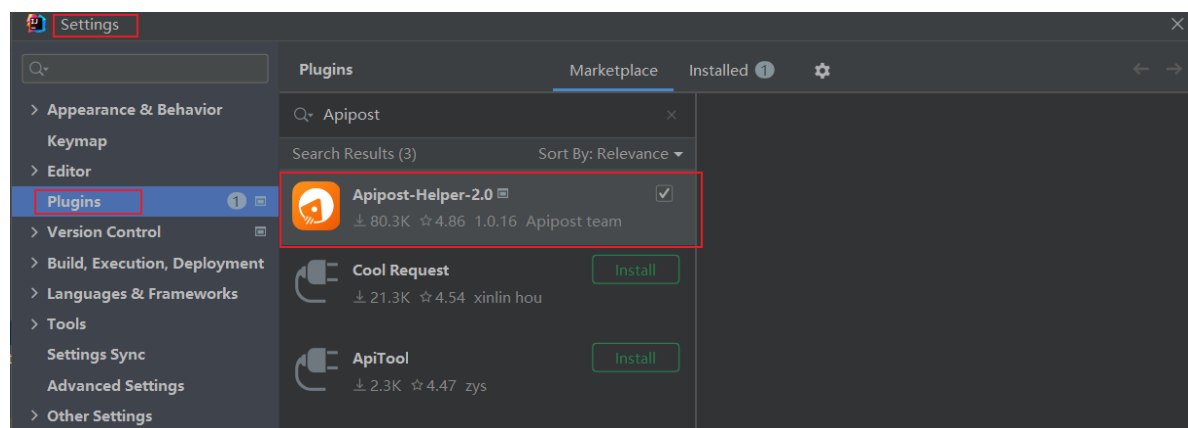
Spring MVC的特点:

- **灵活性**: Spring MVC提供了灵活的URL映射和控制器配置，可以根据业务需求进行定制。
- **松耦合**: Spring MVC采用了松耦合的设计方式，控制器、模型和视图之间的依赖关系松散，提高了代码的灵活性和可维护性。
- **可扩展性**: Spring MVC提供了丰富的拦截器、视图解析器等扩展点，可以轻松地扩展和定制框架的功能。
- **与Spring框架集成**: Spring MVC与Spring框架紧密集成，可以充分利用Spring框架提供的依赖注入、AOP等特性，简化了开发工作。

示例代码

为方便向发送各种请求（网页地址栏只可以模拟get请求），我们使用apipost，在ideal中有这个应用的插件，官网网站：[Apipost-API 文档、设计、调试、自动化测试一体化协作平台](#)。

通过ideal的设置Setting中的Plugins下载插件Apipost，它将会自动检测项目中的接口，方便发送请求。



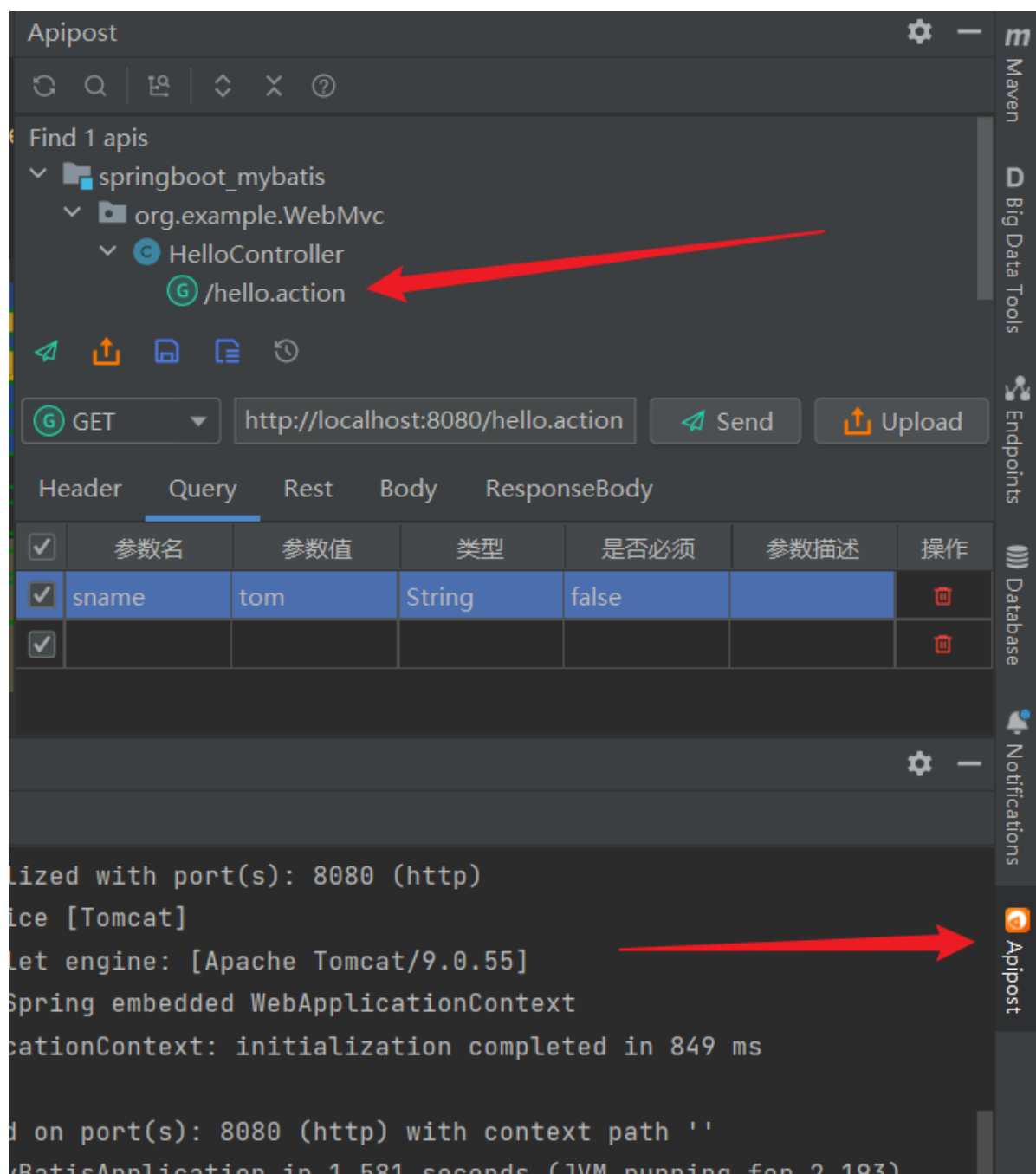
在项目的pom文件中导入web依赖

```
1 <!-- web页面依赖 -->
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-web</artifactId>
5     </dependency>
```

创建WebMvc包，包下创建一个控制器HelloController类来接受http请求（相关注解稍后说明，先使用一个简单例子），然后启动SpringBoot。

```
1 package org.example.webMvc;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.ResponseBody;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RestController
8 public class HelloController {
9     // mvc注解,匹配get请求
10    @GetMapping("hello.action")
11    public String hello(String sname){
12        return "hello" + sname;
13    }
14 }
```

此时，Api插件已经检查到接口了，手动填写sname参数为tom，点击发送，返回hellotom。



- @RestController:

组合注解，由@Responsbody和@Controller组合而成。

返回数据自动序列化：使用 @RestController 注解的控制器方法返回的数据会自动序列化为 JSON 或 XML 格式，并写入 HTTP 响应体中。这样，客户端可以直接通过 HTTP 请求获取到数据，而不需要进行额外的解析。

- @Controller:

标识一个类是控制器，负责处理客户端的 HTTP 请求并返回相应的视图或数据。通常，被 @controller 注解标识的类中的方法被称为处理器方法（Handler Method）。（视图解析：控制器方法通常返回一个视图名，Spring MVC 会将该视图名解析为实际的视图，然后将其渲染并返回给客户端。）

- @Responsbody:

用于指示方法返回的数据直接写入 HTTP 响应体中，而不是解析为视图。

- @GetMapping

参数设置访问路径，@GetMapping("hello.action")，当页面访问/hello.action这个路径时，调用hello方法。

RestFul请求风格

REST即表述性状态传递（英文：Representational State Transfer，简称REST）是Roy Fielding博士在 2000年他的博士论文中提出的一种 **软件架构风格**。

它是一种针对 网络应用的设计和 开发方式，可以降低开发的复杂性，提高系统的可伸缩性。在三种主流的 Web服务实现方案中，因为REST模式的Web服务与复杂的 SOAP和 XML-RPC对比来讲明显的更加简洁，越来越多的web服务开始采用REST风格设计和实现。例如， Amazon .com提供接近REST风格的Web服务进行图书查找； 雅虎提供的Web服务也是REST风格的。（来源于百度）具体内容如下

- Get:请求用于查询
- Post:请求用于新增
- Put:用于修改
- Delete: 用于删除
- option: 用于确认 返回结果应该统一
- code: 状态 data: 返回数据
- message: 返回信

以下注解都是RestFul风格的：GetMapping, PostMapping, PutMapping, deleteMapping, RestController

核心概念与特点：

1. 资源 (Resources)：

- 在 RESTful 架构中，所有的内容都被视为资源。资源可以是任何事物，例如用户、商品、文章等。每个资源都有一个唯一的标识符，通常是 URI（统一资源标识符）。

2. 表现层 (Representation)：

- 资源的表现形式是资源在不同表示形式之间的转换，例如 JSON、XML、HTML 等。客户端和服务端之间通过这些表现形式进行通信。

3. 状态转移 (State Transfer)：

- RESTful 服务通过使用标准的 HTTP 方法（GET、POST、PUT、DELETE 等）来操作资源，这些方法对应了资源的不同状态转移。
- GET：获取资源的表现形式。
- POST：创建新资源。
- PUT：更新已有资源。
- DELETE：删除资源。

4. 无状态 (Stateless)：

- RESTful 架构是无状态的，即每个请求都应该包含所有必要的信息，服务器不应该保存任何会话状态。这样可以使得系统更具可伸缩性和可靠性。

5. 统一接口 (Uniform Interface)：

- 统一接口是 RESTful 架构的核心原则之一，它定义了客户端和服务端之间通信的标准方式。这种统一接口包括以下几个方面：
 - 资源的标识符 (URI)：每个资源都有一个唯一的标识符。
 - 使用标准的 HTTP 方法：GET、POST、PUT、DELETE 等来对资源进行操作。
 - 使用标准的媒体类型（如 JSON、XML）来表示资源的表现形式。
 - 超媒体作为应用状态的引擎 (HATEOAS, Hypermedia As The Engine Of Application State)：通过响应中提供链接来描述资源之间的关系，使客户端能够动态地发现和导航资源。

多种入参形式

在WebMvc包下

1. 参数同名

```
1 @PostMapping("login1.action")
2     public HashMap login(@RequestParam("username") String uname, String
password) {
3         System.out.println(uname);
4         System.out.println(password);
5         HashMap map = new HashMap();
6         map.put("code", "20000");
7         map.put("message", "登录请求成功! ");
8         return map;
9     }
```



- `@PostMapping` 是Spring框架中用于将HTTP POST请求映射到特定处理器方法的注解之一。它是 `@RequestMapping` 的一个特例，专门用于处理HTTP POST请求。使用 `@PostMapping` 注解可以将一个处理器方法映射到指定的URL路径，并指定该方法应该处理的HTTP POST请求。
- `@RequestParam` 是Spring框架中用于从HTTP请求中获取请求参数的注解之一。它可以用于将请求参数映射到处理器方法的参数上。

使用 `@RequestParam` 注解，您可以指定处理器方法参数的名称，以及是否需要这些参数是必需的。如果请求中没有指定这些参数，可以提供默认值或者将参数设置为可选的。

以下是 `@RequestParam` 注解的一些常用属性：

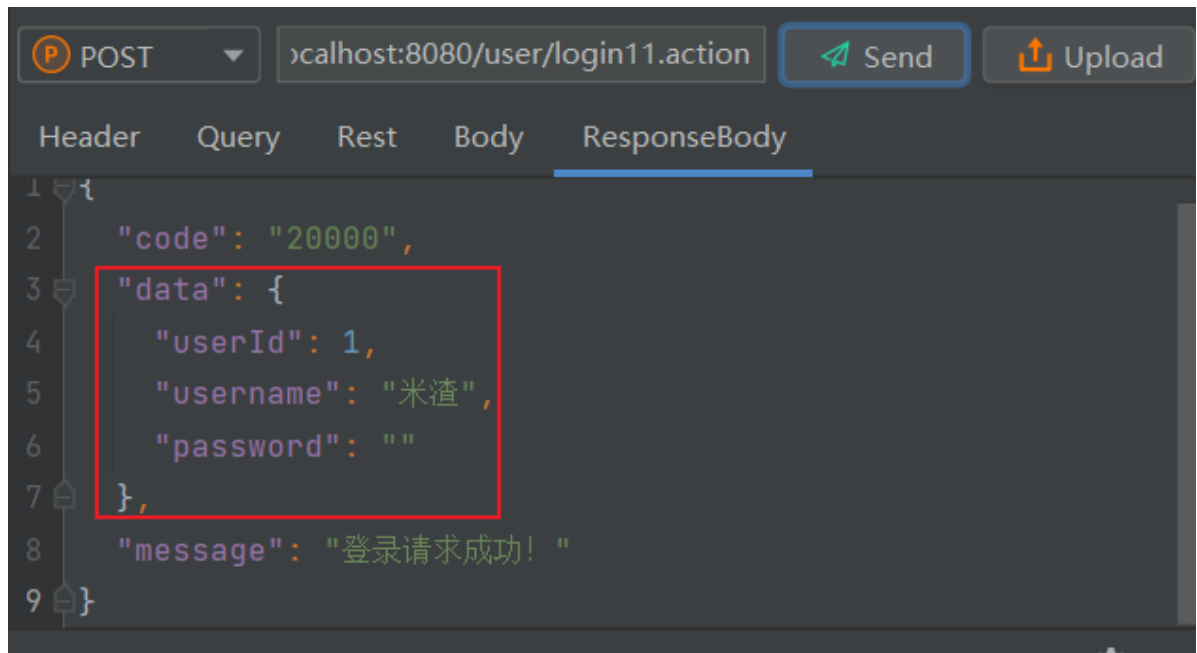
- `name`：指定请求参数的名称。如果不指定名称，默认情况下将使用处理器方法参数的名称作为请求参数的名称。
- `value`：与 `name` 属性相同，用于指定请求参数的名称。
- `required`：指定请求参数是否是必需的。默认值为 `true`，表示该参数是必需的。如果设置为 `false`，则表示该参数是可选的，如果请求中没有提供该参数，Spring 将使用 `null` 值注入到处理器方法的参数中。
- `defaultValue`：指定请求参数的默认值。如果请求中没有提供该参数，或者参数值为 `null`，则使用默认值作为参数值。

`@RequestParam("username") String uname, String password` 这句话是指定请求参数为 `username`，并将其映射到方法参数 `uname` 上，而 `String password` 没有被指定，是一个可选参数，如果请求中没有对应的参数，则值为 `null`。

使用万能的HashMap对象来存储键值，构成一个简单的响应对象。

2.模板数据

```
1 @PostMapping("login11.action")
2     public HashMap login2(UserInfo userInfo) {
3         System.out.println(userInfo.getUsername() + "--login11--" +
4             userInfo.getPassword());
5         HashMap map = new HashMap();
6         map.put("code", "20000");
7         map.put("message", "登录请求成功! ");
8         map.put("data", userInfo);
9         return map;
10    }
```



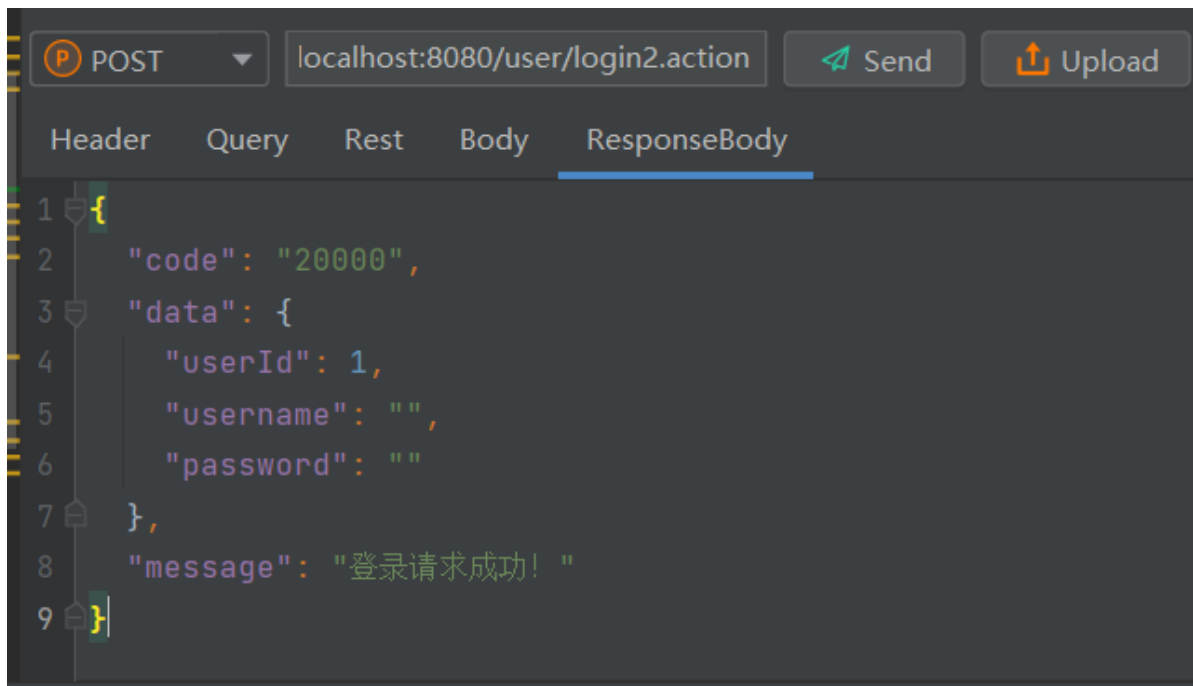
创建数据模型UserInfo，放置在vo包下（这个包名下一般放置数据模型），设置一些属性并生成get和set方法。

在客户端发送POST请求时，请求体中应该包含的参数名为 `username` 和 `password`。Spring框架会尝试从请求中获取这些参数的值，并将它们设置到 `UserInfo` 对象的对应属性中。然后，Spring将会将这个 `UserInfo` 对象作为方法的参数传递给 `login2()` 方法。

这种方式简化了方法的参数处理，不需要手动解析请求参数，而是直接使用数据模型对象来表示请求参数。

3.json形式入参

```
1 @PostMapping("login2.action")
2     public HashMap login3(@RequestBody UserInfo userInfo) {
3         HashMap map = new HashMap();
4         map.put("code", "20000");
5         map.put("message", "登录请求成功! ");
6         map.put("data", userInfo);
7         return map;
8     }
```



```
POST localhost:8080/user/login2.action
{
  "code": "20000",
  "data": {
    "userId": 1,
    "username": "",
    "password": ""
  },
  "message": "登录请求成功!"
}
```

`login3()` 方法接收一个 `UserInfo` 对象作为参数，该对象使用 `@RequestBody` 注解修饰。当客户端向 `/login2.action` 路径发送 POST 请求时，请求体中的数据会被自动映射到 `UserInfo` 对象中，并作为参数传递给 `login3()` 方法。

需要注意的是，使用 `@RequestBody` 注解时，请求体中的数据必须符合 `UserInfo` 对象的结构，否则会导致数据绑定失败。此外，Spring 默认使用的是 `HttpMessageConverter` 来处理请求体中的数据，因此需要确保配置了合适的消息转换器来处理请求体中的数据格式。通常情况下，Spring Boot 会自动配置好所需的消息转换器，因此大部分情况下无需手动配置。

使用 `@RequestBody` 注解接收请求体中的数据，并在方法中处理后返回的数据通常会以 JSON 格式返回给客户端。这是因为 Spring MVC 默认使用 Jackson 库来处理对象的序列化和反序列化，而 Jackson 库会将对象转换为 JSON 格式。因此，如果方法的返回类型是对象类型，并且使用了 `@RequestBody` 注解，那么返回的数据将会以 JSON 格式返回给客户端。

注意：使用 `@RequestBody` 注解时，数据通常作为请求体的一部分，而不是 URL 的查询参数，因此在此 URL 中不会显示具体的参数名称（apipost 中 Query 部分不会显示具体参数名称）。相反，不使用 `@RequestBody` 注解时，数据通常作为 URL 的查询参数，因此在 URL 中会显示具体的参数名称。

补充：

`HttpMessageConverter` 是 Spring MVC 中用于处理 HTTP 请求和响应消息的接口。它的作用是将请求体中的数据转换成对象，并将方法返回的对象转换成响应消息的格式。`HttpMessageConverter` 接口定义了两个主要的方法：`read()` 和 `write()`。

1. `read()` 方法用于将请求体中的数据转换成 Java 对象。当客户端发送 HTTP 请求时，Spring MVC 将会根据请求的内容类型选择合适的 `HttpMessageConverter` 来处理请求体中的数据，并将其转换成方法参数所需的对象。
2. `write()` 方法用于将 Java 对象转换成响应消息的格式。当处理器方法返回一个对象时，Spring MVC 将会根据响应的内容类型选择合适的 `HttpMessageConverter` 来将方法返回的对象转换成相应的格式，例如 JSON、XML 等。

Spring MVC 中默认提供了许多实现了 `HttpMessageConverter` 接口的类，用于处理不同类型的数据格式，包括：

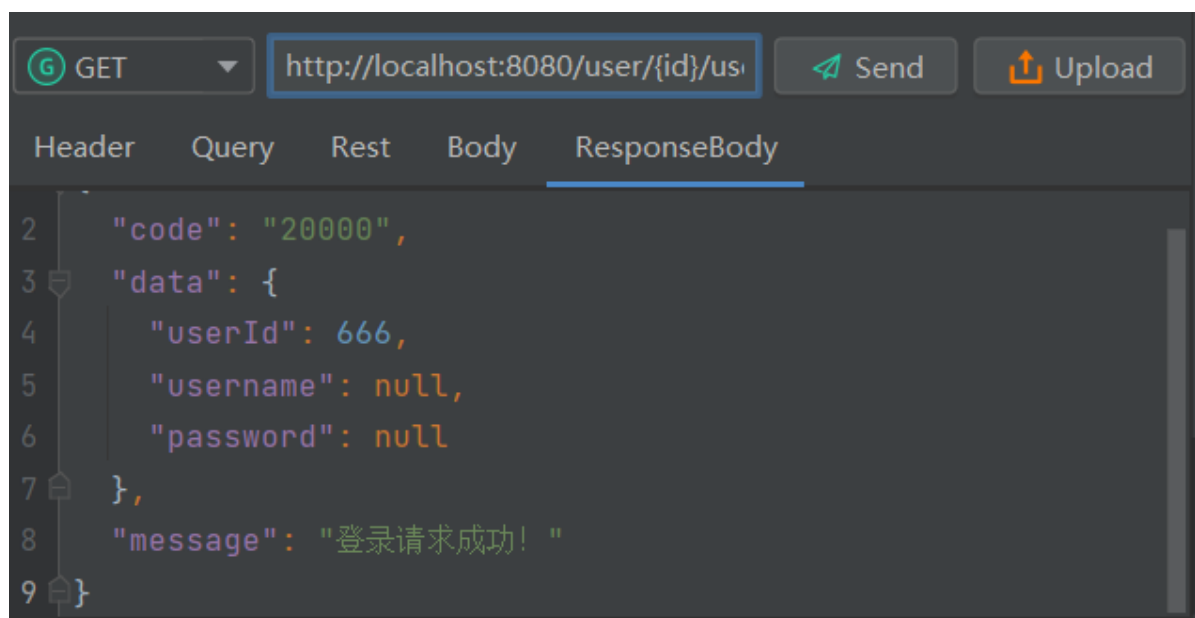
- `MappingJackson2HttpMessageConverter`：用于处理 JSON 格式的数据。
- `StringHttpMessageConverter`：用于处理字符串数据。
- `ByteArrayHttpMessageConverter`：用于处理字节数组数据。

- `FormHttpMessageConverter`：用于处理表单数据。

在 Spring Boot 应用中，通常会自动配置好所需的 `HttpMessageConverter`，因此大部分情况下无需手动配置。但如果需要自定义消息转换器，可以通过配置 `WebMvcConfigurer` 或 `WebMvcConfigurerAdapter` 来实现。

4. 路径变量形式入参

```
1  @GetMapping("/{id}/userId")
2      public HashMap getById(@PathVariable("id") int id) {
3      UserInfo ui = new UserInfo();
4      ui.setUserId(id);
5
6      HashMap map = new HashMap();
7      map.put("code", "20000");
8      map.put("message", "登录请求成功! ");
9      map.put("data", ui);
10     return map;
11 }
```



`{id}` 是路径变量，表示在请求路径中会有一个名为 `id` 的占位符，该占位符会被具体的值替代。
`@PathVariable("id")` 注解表示将请求路径中的 `id` 路径变量映射到方法参数 `id` 上。

这个方法的入参形式是使用了路径变量的形式，通过请求路径中的 `id` 参数来传递值。例如，对于请求 `GET /{id}/userId`，如果请求路径是 `/123/userId`，那么 `id` 的值将会被映射为 `123`，然后将这个值传递给方法中的 `getById()` 方法的参数 `id`。

在方法中，首先创建了一个 `UserInfo` 对象，并设置了 `userId` 属性为传入的 `id` 值。然后，创建了一个 `HashMap` 对象，将一些信息放入其中，并将这个 `HashMap` 对象作为方法的返回值返回。

数据库相关操作

一.JDBC: java链接数据库技术

1.引入JDBC数据库驱动

```
1 <!--mysql8-->
2 <dependency>
3 <groupId>mysql</groupId>
4 <artifactId>mysql-connector-java</artifactId>
5 <version>8.0.20</version>
6 </dependency>
7
8 <!--mysql5-->
9 <dependency>
10 <groupId>mysql</groupId>
11 <artifactId>mysql-connector-java</artifactId>
12 <version>5.1.48</version>
13 </dependency>
```

在使用前，首先查看数据库所在版本

```
1 | SELECT VERSION();
```

```
mysql> SELECT VERSION();
+-----+
| VERSION() |
+-----+
| 8.0.20    |
+-----+
1 row in set (0.00 sec)

mysql> _
```

2.查询操作五个步骤

- 1.加载驱动： 实际上可以省略，一个驱动类
- 2.创建链接：
 - 1.url: 协议 ip, port, 库
 - 2.用户名
 - 3.密码
- 3.创建执行器
 1. Statement, PreparedStatement, CallableStatement
 - PreparedStatement: 预处理执行器： 可以设置？ 作为占位符
- 4.执行sql, 处理结果
 1. ResultSet: 结果集
 2. next(); 判断是否有下一条记录，如果有就指向下一条记录。
 3. getXXX(int | String); 获取int第几列数据， String根据列名获取数据
- 5.关闭资源

测试代码位于JDBCtest包下。以下是部分讲解

```
1 /**
2  *首先，使用 Class.forName() 方法加载 MySQL 数据库的 JDBC 驱动程序。
3  *"com.mysql.cj.jdbc.Driver" 是 MySQL Connector/J 提供的 JDBC 驱动程序类的完全限定名。
4  *加载驱动程序是为了使得 Java 应用程序能够与 MySQL 数据库进行通信。
```

```

5  */
6
7  Class.forName("com.mysql.cj.jdbc.Driver");
8      // 2.创建链接
9
10 /*
11 *创建数据库连接：通过 DriverManager.getConnection() 方法创建与 MySQL 数据库的连接。
  该方法的参数包括：
12
13 数据库 URL：指定要连接的 MySQL 数据库的 URL。在这
  里，"jdbc:mysql://192.168.10.141:3306/myemployees1" 是一个 URL 格式的字符串，其
  中包括了 MySQL 服务器的 IP 地址（192.168.10.141）、端口号（3306）以及要连接的数据库
  名称（myemployees1）。
14
15 用户名和密码：用于验证连接的用户名和密码。在这里，用户名是 "root"，密码是 "123456"。
16
17 */
18 conn =
  DriverManager.getConnection("jdbc:mysql://192.168.10.141:3306/myemployees1",
    "root", "123456");

```

```

1  // 3.创建执行器
2
3  StringBuilder sb = new StringBuilder();
4
5  // SQL语法
6  sb.append("\n" +
7  "SELECT e.employee_id,\n" +
8  "      e.first_name,\n" +
9  "      e.last_name,\n" +
10 "      e.email,\n" +
11 "      e.phone_number,\n" +
12 "      e.job_id,\n" +
13 "      e.salary,\n" +
14 "      e.commission_pct,\n" +
15 "      e.manager_id,\n" +
16 "      e.department_id,\n" +
17 "      e.hiredate\n" +
18 "FROM employees1 e");

```

- 创建一个 `StringBuilder` 对象，用于动态构建字符串。`StringBuilder` 是 Java 中用于处理字符串的一个类，它允许动态地构建和修改字符串，而不会产生大量的临时对象，从而提高了性能。

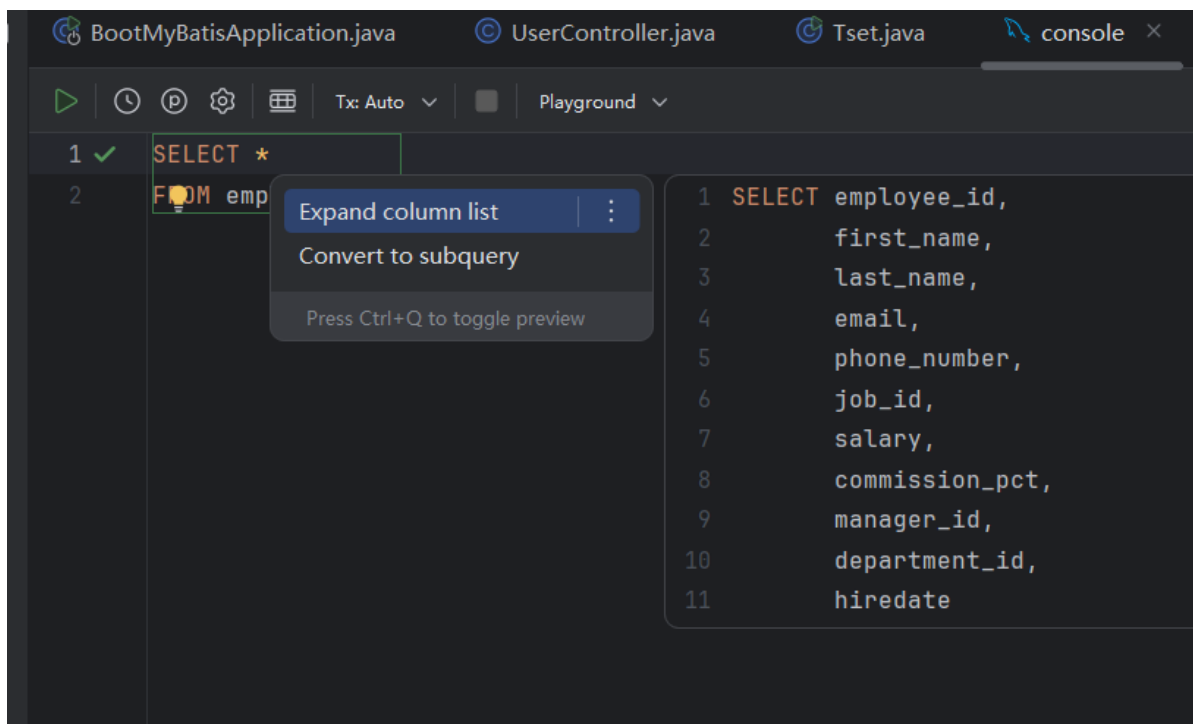
具体来说：

1. **StringBuilder**： `StringBuilder` 是 Java 中的一个类，位于 `java.lang` 包下。它提供了一系列的方法来操作字符串，包括添加、插入、替换、删除等操作。
2. **new StringBuilder()**： 通过 `new` 关键字创建了一个 `StringBuilder` 对象的实例。在括号中没有传入任何参数，表示创建一个初始容量为 16 个字符的 `StringBuilder` 对象。如果需要，`StringBuilder` 的容量会自动增长以容纳更多的字符。

通过这行代码，我们创建了一个空的 `StringBuilder` 对象，可以随时向其中添加需要的字符串内容，用于构建 SQL 查询语句、HTML 页面、日志信息等。

对于这条 SQL 语法，有着快速的写法。

1. 首先连接数据库，新建查询，通过*可以快速填写各字段



2. 写出 `sb.append("")`，然后在双引号中执行粘贴即可。

```
1 // 创建执行器Statement，此处使用PreparedStatement
2 PreparedStatement ps = conn.prepareStatement(sb.toString())
```

- `PreparedStatement` 是 `Statement` 的子接口，它可以执行预编译的 SQL 语句，并且支持参数化查询，以便安全地防止 SQL 注入攻击。

具体解释如下：

1. **连接对象 conn**：在前面的代码中已经通过 `DriverManager.getConnection()` 方法建立了与数据库的连接，现在我们将使用这个连接对象来创建 `PreparedStatement`。
2. **sb.toString()**：这里的 `sb` 是一个 `StringBuilder` 对象，它用于构建 SQL 语句。`.toString()` 方法将 `StringBuilder` 对象转换为一个字符串，该字符串包含了我们要执行的 SQL 查询或更新语句。
3. **ps**：这是一个 `PreparedStatement` 对象，它表示一个已经预编译好的 SQL 语句。通过调用连接对象的 `prepareStatement()` 方法并传入 SQL 语句，我们可以创建一个 `PreparedStatement` 对象。

这行代码的作用是根据给定的 SQL 语句创建一个 `PreparedStatement` 对象，以便后续可以向其中填充参数并执行查询或更新操作。

```
1 // 4. 执行SQL
2     ResultSet rs = ps.executeQuery();
3     while (rs.next()) {
4         int employee_id = rs.getInt("employee_id");
5         String first_name = rs.getString("first_name");
6         String last_name = rs.getString("last_name");
7         System.out.println(rs.getInt(1) + "\t" + first_name + "\t" +
8         last_name);
9     }
```

第五步，使用了数据库连接之后，一定要记得关闭。

```

1 finally {
2     if (conn != null) {
3         try {
4             // 5.释放资源
5             conn.close();
6         } catch (SQLException e){
7             throw new RuntimeException(e);
8         }
9     }
}

```

3.插入操作

对于另外的SQL操作，我们更改第3步的SQL语法就好了。比如插入操作，我们把查询语句修改为。ideal提供的数据库管理工具，没有提示各字段，我使用了DataGrip。（SQL语法务必在数据库管理工具上运行成功了，再拿过来）

```

1 INSERT INTO employees1 (first_name, last_name, email, phone_number, job_id,
2 salary, commission_pct, manager_id, department_id, hiredate)
3 values ('张', '三', 'zhangsan.qq.com', '12321312', 'AD_PRES', 4000, null, 100,
4 90, STR_TO_DATE('2024-6-01', '%Y-%m-%d'));

```

```

1 use myemployees1;
2
3 INSERT INTO employees1 (first_name, last_name, email, phone_number, job_id, salary, commission_pct, manager_id, department_id, hiredate)
4 values ('张', '三', 'zhangsan.qq.com', '12321312', 'AD_PRES', 4000, null, 100, 90, STR_TO_DATE('2024-6-01', '%Y-%m-%d'));
5
6 SELECT *
7 from employees1
8 where first_name='张';
9
10 delete from employees1
11 where first_name='张';

```

这句SQL可以预编译，正如前面所说，PreparedStatement: 预处理执行器：可以设置？作为占位符。所以这句SQL语法，可以写成如下形式。

```

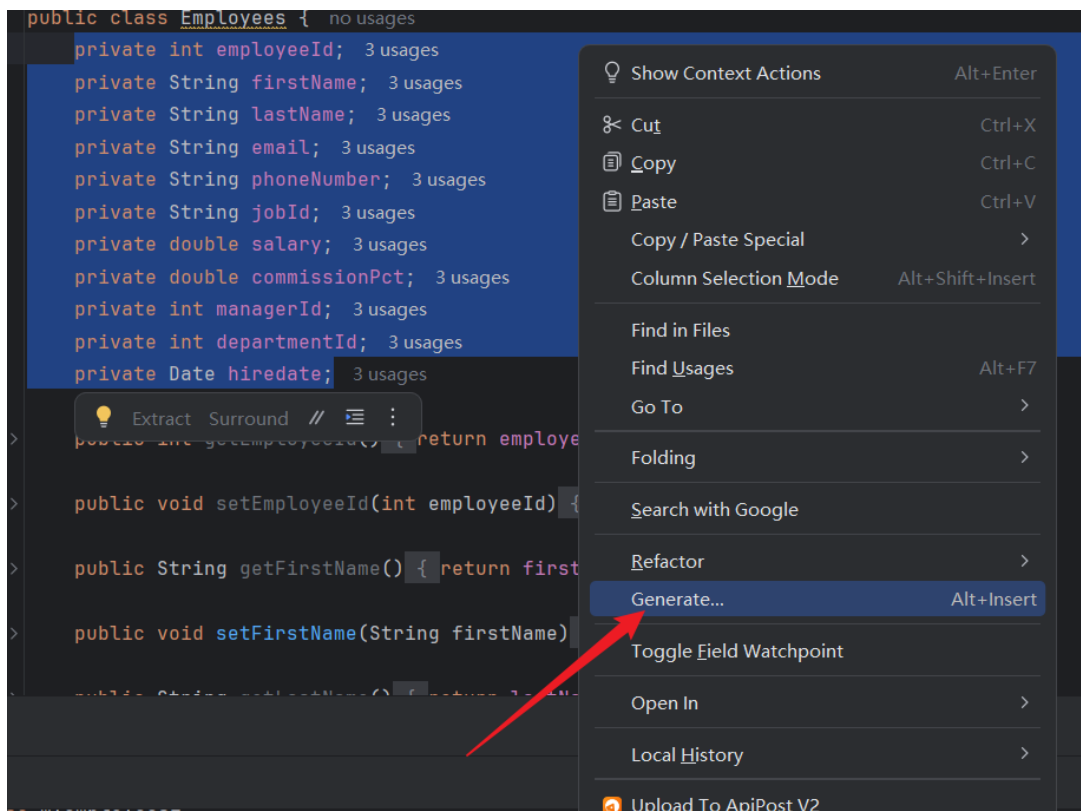
1 INSERT INTO employees1 (first_name, last_name, email, phone_number, job_id,
2 salary, commission_pct, manager_id, department_id, hiredate)
3 values (?, ?, ?, '12321312', 'AD_PRES', 4000, null, 100, ?, ?));

```

？号占位，那么这些字段对应的数据怎么办呢？接下来做一个网页信息模拟。？所代表的信息需要在网页填写。填写的这些信息怎么拿到呢？我们为这些数据创建一个数据模型Employees，用来存放这些数据。

数据模型中包含employees表的所有字段，字段employeeId是自增主键，在SQL语句中可不写。使用ideal的生成器生成这些属性的get, set, toString方法。

- 生成器使用：选中这些属性，然后鼠标右键选择Generate，在弹出的界面中，选择要生成的方法，然后选择为哪些属性生成方法（按住键盘Ctrl键，可以多选属性）



```
1 private int employeeId;  
2 private String firstName;  
3 private String lastName;  
4 private String email;  
5 private String phoneNumber;  
6 private String jobId;  
7 private double salary;  
8 private double commissionPct;  
9 private int managerId;  
10 private int departmentId;  
11 private Date hiredate;
```

接着模拟网页添加信息，创建Employee对象。实际上所有字段都应该用？占位，然后设置填写，但是我懒，写5个试试吧。

```
1 // 模拟网页添加信息  
2 Employees empcondition = new Employees();  
3 empcondition.setFirstName("张");  
4 empcondition.setLastName("三");  
5 empcondition.setEmail("dsad22333@163.com");  
6 empcondition.setDepartmentId(90);  
7 empcondition.setHiredate(new SimpleDateFormat("yyyy-MM-dd").parse("2023-10-09"));
```

然后从数据模型中获取数据，设置？的值。对于时间，setDate默认使用的是SQL的时间，所以我们这里需要转换一下。参数index的位置，需要填写是第几个占位符，比如第一个占位符是1，所以index从1开始自增

```

1 | int index = 1;
2 | ps.setString(index++, empcondition.getFirstName());
3 | ps.setString(index++, empcondition.getLastName());
4 | ps.setString(index++, empcondition.getEmail());
5 | ps.setInt(index++, empcondition.getDepartmentId());
6 | ps.setDate(index++, new java.sql.Date(empcondition.getHiredate().getTime()));

```

最后，来到第4步，之前执行的是查询操作，这里是添加数据，使用更新操作。返回更新的条数。

```

1 | int rows = ps.executeUpdate();
2 | System.out.println(rows);

```

对于更新操作而已，可以视为一个事务，事务可以开启回滚。在模拟网页添加信息这步前添加下面代码，关闭自动提交。

```

1 | // 开启事务
2 | conn.setAutoCommit(false);

```

在执行SQL步骤后，对提交做一个判断，如果更新条数是1才提交，否则，回滚。

```

1 | if (rows == 1) {
2 |         conn.commit();
3 |     } else {
4 |         conn.rollback();
5 |     }

```

二.Druid

三.MyBatis

1.为什么使用MyBatis?

MyBatis 是一个优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以通过简单的 XML 或注解用于配置和原始映射，将接口和 Java 的 POJOs（Plain Old Java Objects）映射成数据库中的记录。

所以，使用MyBatis是因为它能够通过简单的配置文件和注解，将 SQL 语句与 Java 对象进行映射，简化了数据持久层的开发。比传统的 JDBC 更加灵活和方便，适用于需要对 SQL 有更多控制和自定义的场景。

s核心功能：

- **SQL 映射配置文件**：MyBatis 通过 XML 文件或注解来配置和映射 SQL 语句。
- **持久层对象映射**：通过映射文件，可以将 Java 对象与数据库中的记录进行映射。
- **动态 SQL**：支持动态生成 SQL 语句，减少硬编码 SQL 语句的麻烦。
- **参数化 SQL**：支持使用参数化的 SQL 语句，增强 SQL 的可复用性和可维护性。
- **缓存机制**：MyBatis 提供了一级缓存和二级缓存机制，提高数据库查询的效率。

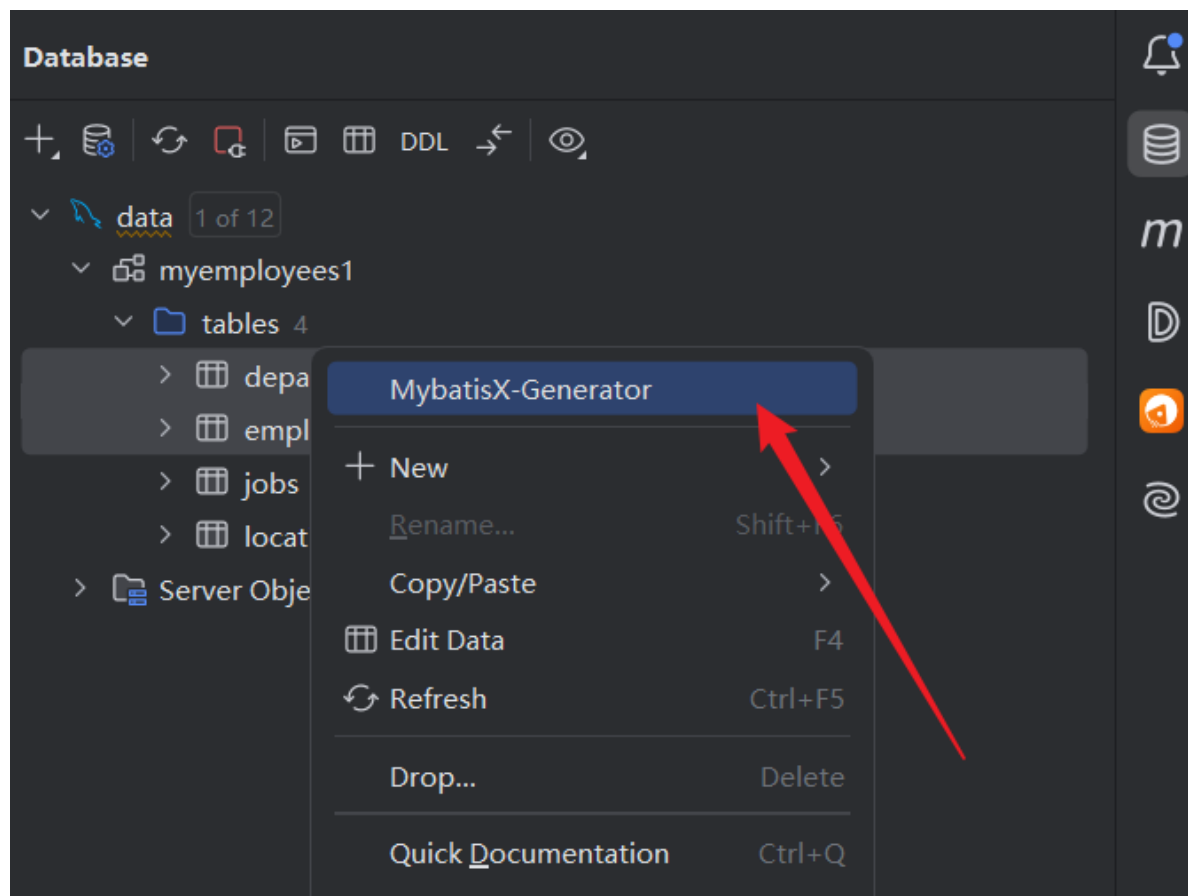
工作原理：

1. **读取配置文件**：MyBatis 启动时读取全局配置文件 `mybatis-config.xml`。
2. **创建 SqlSessionFactory**：通过配置文件创建 SqlSessionFactory 实例。

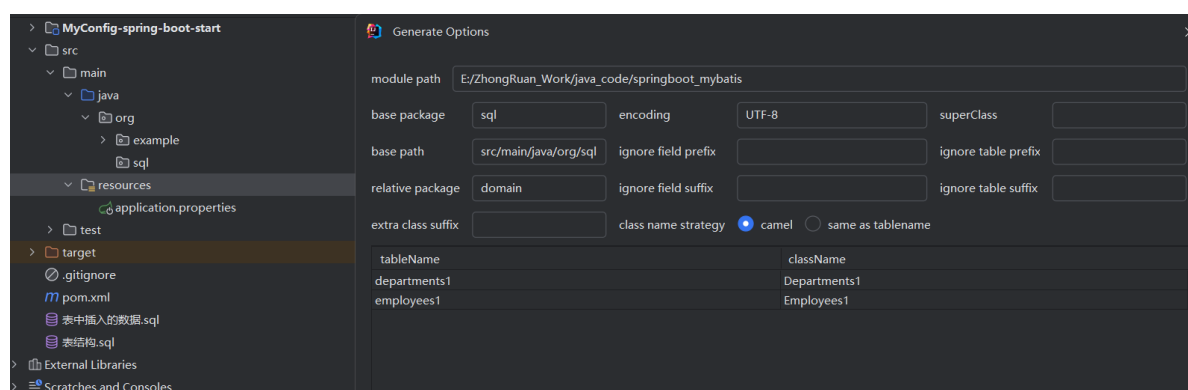
3. **获取 SqlSession**：通过 SqlSessionFactory 获取 SqlSession 实例。
4. **执行 SQL 语句**：使用 SqlSession 调用配置的 SQL 语句进行数据库操作。
5. **处理结果**：将 SQL 执行结果映射到 Java 对象并返回给调用者。

2. MyBatis的使用

我们可以使用插件MyBatisX来快速生成。在IDEA的插件市场下载好MyBatisX之后，在IDEA中选中数据库中的表，右键出现一个新的选项。



生成器界面如下。



以下为将要使用的各个选项说明

- module path: 设置基础路径。
- base package: 设置包路径。（此处我先创建了包sql，并设置在这个包下生成）
- relative package: 子包路径。（方便管理项目结构，一般设置以下名字）
 - domain:
 - 含义：表示领域模型包，通常包含与数据库表对应的实体类（Entity Classes）。

- **用途**：用来存放业务领域对象，例如 `User`、`Product` 等与数据库表对应的类。
- 生成的实体类会放在 `domain` 包下。
- **pojo** (Plain Old Java Object) :
 - **含义**：表示简单的 Java 对象包，通常包含没有业务逻辑的纯数据对象。
 - **用途**：用来存放简单的 Java 对象，这些对象主要用于数据传输和存储。
 - 生成的简单数据对象会放在 `pojo` 包下
- **dao** (Data Access Object) :
 - **含义**：表示数据访问对象包，通常包含与数据库操作相关的接口和实现类。
 - **示例**：`com.example.project.dao`
 - **用途**：用来存放数据访问层代码，例如 Mapper 接口、数据库操作方法。
 - 生成的数据访问接口和类会放在 `dao` 包下

疑问

此处是学习过程中遇到过的疑惑。

1.类，对象，Bean的关系？

首先，在 Java 中，类是对象的模板或者蓝图，它描述了对应的属性和方法。当程序运行时，类的实例化过程就是创建对象的过程，通过关键字 `new` 可以实例化一个类，创建对象的实例。

而在 Spring 中，Bean 是 Spring IoC 容器中管理的对象，它们由容器负责创建、装配和管理。Spring IoC 容器负责实例化和 Bean，使得开发者无需手动创建对象，而是由容器负责创建和管理对象的生命周期。

在 Spring 中，我们使用 `@Bean` 注解或者在 `@Component`、`@Service`、`@Repository` 等注解标记的类来声明 Bean。这些注解标记的类在 Spring IoC 容器启动时会被扫描到，并且相应的 Bean 会被创建并加入容器管理。这样，这些被注解标记的类就变成了 Spring 容器中的 Bean，可以通过容器来获取和使用。

所以，虽然在 Java 中类是对象的模板，但在 Spring 中，我们可以使用注解将类声明为 Bean，并由 Spring IoC 容器来实例化和 Bean。这样，类就变成了 Spring 中管理的对象，也就是 Bean。

```
1 | @Autowired
2 | private UserService userService;
```

比如有一个类 `UserService` 使用了 `@Bean` 注解，我们不需要手动去实例化，使用 `@Autowired` 注解，Spring IoC 容器会自动将 `userServiceImpl` 的实例注入到 `userService` 属性中。

2.什么叫做应用程序上下文？

应用程序的上下文指的是 Spring 容器中管理的所有 Bean、配置信息以及容器的状态。在 Spring 框架中，应用程序的上下文是一个对象，它管理着应用程序中的所有组件、依赖关系和配置。

应用程序的上下文包括以下内容：

1. **Bean 定义**：定义了应用程序中的所有 Bean，包括服务类、控制器、DAO、配置类等。
2. **Bean 实例**：容器实例化了所有的 Bean，并维护着它们的生命周期。
3. **依赖注入**：容器负责注入 Bean 之间的依赖关系，确保它们能够正确地协作。
4. **配置信息**：包括应用程序的配置文件、注解配置以及外部配置等。
5. **运行时状态**：容器维护了应用程序运行时的状态信息，如活动的线程、事务状态等。

Spring 容器提供了不同类型的上下文，最常见的包括：

- **应用程序上下文 (ApplicationContext)**：负责加载应用程序的所有 Bean 定义，并管理这些 Bean 的生命周期。它是 Spring 应用程序的核心接口，提供了丰富的功能，如国际化支持、事件处理、Bean 生命周期管理等。
- **Web 应用程序上下文 (WebApplicationContext)**：是应用程序上下文的一个扩展，专门用于 Web 应用程序。它除了具备应用程序上下文的所有功能外，还提供了与 Web 相关的功能，如处理 HTTP 请求、管理会话、处理表单提交等。
- **其他类型的上下文**：还有一些其他类型的上下文，如测试上下文 (TestContext)、消息上下文 (MessageContext) 等，它们用于特定的场景或功能。

总的来说，应用程序的上下文是 Spring 容器中管理的所有组件、依赖关系和配置的集合，它提供了一个统一的管理和访问接口，帮助开发者管理和控制应用程序的运行。

3.异常处理和异常通知有什么区别？

异常通知不是用来捕获和处理异常的，而是用来在方法抛出异常时执行的一种行为。通常，异常通知用于捕获方法执行过程中抛出的异常，并记录相关信息，比如将异常信息记录到日志中。通知可以在方法抛出异常后立即执行，而无需在每个可能抛出异常的地方都编写处理逻辑。

4.@RequestBody和@ResponseBody注解的不同

`@RequestBody` 和 `@ResponseBody` 是 Spring MVC 中用于处理 HTTP 请求和响应的注解，它们用于不同的地方，具有不同的作用：

1. @RequestBody:

- `@RequestBody` 注解通常用于方法参数上，用于指示 Spring MVC 将请求体中的数据转换成对应的 Java 对象，并作为方法的参数传递进来。通常情况下，请求体中的数据是以 JSON、XML 等格式进行提交的。
- 例如，`@RequestBody` 注解可以用于处理 POST 请求中的 JSON 数据，将 JSON 数据自动转换成 Java 对象。
- 使用 `@RequestBody` 注解后，Spring MVC 将会根据请求的内容类型选择合适的 `HttpMessageConverter` 来处理请求体中的数据。

2. @ResponseBody:

- `@ResponseBody` 注解通常用于方法上，用于指示 Spring MVC 将方法的返回值直接作为响应消息的主体返回给客户端。
- 例如，`@ResponseBody` 注解可以用于返回 JSON 格式的数据，将 Java 对象自动转换成 JSON 格式，并作为响应的主体返回给客户端。
- 使用 `@ResponseBody` 注解后，Spring MVC 将会根据方法返回值的类型选择合适的 `HttpMessageConverter` 来将返回值转换成响应消息的格式。

因此，`@RequestBody` 和 `@ResponseBody` 注解用于不同的地方，分别处理请求和响应中的数据。`@RequestBody` 处理请求体中的数据，将其转换成 Java 对象；`@ResponseBody` 处理方法的返回值，将其转换成响应消息的格式。

5.url,uri是什么？

URI (Uniform Resource Identifier, 统一资源标识符)

- **定义**：URI 是一种用于标识资源的字符串。URI 可以是一个 URL，也可以是一个 URN (Uniform Resource Name, 统一资源名称)。
- **组成部分**：

1. **方案 (Scheme)**：例如 `http`、`https`、`ftp`、`mailto` 等。

2. 层级部分：

- **授权 (Authority)** (可选)：例如 `user:password@host:port`。
- **路径 (Path)**：例如 `/path/to/resource`。

3. 查询 (Query) (可选)：例如 `?key=value`。

4. 片段 (Fragment) (可选)：例如 `#section1`。

• 示例：

- `http://example.com/path/to/resource?key=value#section1`
- `mailto:user@example.com`

- **特点：**URI 是一个抽象的概念，它可以用来标识任何资源（包括 URL 和 URN）。
-

URL (Uniform Resource Locator, 统一资源定位符)

- **定义：**URL 是一种特定类型的 URI，用于通过描述访问资源的方法和路径来定位资源。

• 组成部分：

1. 方案 (Scheme)：例如 `http`、`https`、`ftp`。

2. 授权 (Authority)：

1. 用户信息 (User Info) (可选)：例如 `user:password`。

2. 主机 (Host)：例如 `example.com`。

3. 端口 (Port) (可选)：例如 `:8080`。

3. 路径 (Path)：例如 `/path/to/resource`。

4. 查询 (Query) (可选)：例如 `?key=value`。

5. 片段 (Fragment) (可选)：例如 `#section1`。

• 示例：

- `http://example.com/path/to/resource?key=value#section1`
- `ftp://user:password@example.com:21/path`

- **特点：**URL 不仅标识了资源，还提供了访问资源的方法和路径。
-

主要区别

1. **范围：**URL 是 URI 的一个子集。所有的 URL 都是 URI，但不是所有的 URI 都是 URL。
 2. **功能：**URI 可以仅仅标识资源，而 URL 除了标识资源外，还提供了如何访问该资源的信息。
 3. **表达方式：**URL 必须包含访问资源的协议（如 `http`、`ftp` 等）和路径，而 URI 可能只包含资源的名称或标识符。
-

URN (Uniform Resource Name, 统一资源名称)

- **定义：**URN 是一种 URI，用于通过名称标识资源，而不提供其位置或访问方法。

• 组成部分：

1. 方案 (Scheme)：通常是 `urn`。

2. 命名空间标识符 (Namespace Identifier, NID)：例如 `isbn`。

3. 命名空间具体字符串 (Namespace Specific String, NSS)：例如 `0451450523`。

• 示例：

- `urn:isbn:0451450523` (标识一本书的 ISBN 编号)

- `urn:ietf:rfc:2141` (标识一个 IETF RFC 文档)

- **特点：**URN 仅仅用来标识资源，不包含访问资源的方法或路径信息。
-

总结

- **URI：**统一资源标识符，可以是 URL 或 URN，用于标识资源。
- **URL：**统一资源定位符，是 URI 的一种，用于描述如何访问资源。
- **URN：**统一资源名称，是 URI 的一种，用于通过名称标识资源。