

SpringBoot

文档更新日志

版本	更新日期	操作	描述
v1.0	2021/11/14	A	基础篇
v1.0.1	2021/11/30	U	更新基础篇错别字若干，不涉及内容变更
v2.0	2021/12/01	A	运维实用篇
V3.0	2022/2/21	A	开发实用篇
V4.0	2022/3/29	A	原理篇

前言

很荣幸有机会能以这样的形式和互联网上的各位小伙伴一起学习交流技术课程，这次给大家带来的是Spring家族中比较重要的一门技术课程——SpringBoot。一句话介绍这个技术，应该怎么文件说呢？现在如果开发Spring程序不用SpringBoot那就是给自己过不去，SpringBoot为我们开发Spring程序提供了太多的帮助，在此借这个机会给大家分享这门课程，希望各位小伙伴学有所得，学有所用，学有所成。

正如上面提到的，这门技术是用来加速开发Spring程序的，因此学习这门技术是有一定的门槛的。你可以理解为你现在是一门传统的手工艺人，现在工艺升级，可以加速你的生产制作过程，但是前提是你要会原始工艺，然后才能学习新的工艺。嗯，怎么说呢？有一定的门槛，至少Spring怎么回事，与Spring配合在一起工作的一堆技术又是怎么回事，这些搞明白才能来看这个技术，不然就只能学个皮毛，或者学着学着就开始因为其他技术不太过关，然后就学不下去了，然后，就没有然后了，果断弃坑了。不管怎么说，既来之则安之，加油学习吧，投资自己肯定是没毛病的。

课程内容说明

SpringBoot这门技术课程所包含的技术点其实并不是很多，但是围绕着SpringBoot的周边知识，也就是SpringBoot整合其他技术，这样的知识量很大，例如SpringBoot整合MyBatis等等。因此为了能够将本课程制作的能够适应于各个层面的学习者进行学习，本套课程会针对小白，初学者，开发者三种不同的人群来设计全套课程。具体这三种人群如何划分，就按照我的描述形式来分吧，各位小伙伴可以对号入座，每种人群看课程的起始位置略有差别。

学习者	归类方式
小白	完全没有用过SpringBoot技术
初学者	能使用SpringBoot技术完成基础的SSM整合
开发者	能使用SpringBoot技术实现常见的技术整合工作

简单说就是你能用SpringBoot做多少东西，一点不会就是小白，会一点就是初学者，大部分都会就是开发者。其实这个划分也不用过于纠结，这个划分仅仅是为了帮助你对本技术课程所包含的阶段模块划分做一个清晰认知，因为本课程中会将SpringBoot技术划分成4个单元，每个单元是针对不同的学习者准备的。

学习者	课程单元
小白	基础篇
初学者	应用篇 (运维实用篇 & 开发实用篇)
开发者	原理篇

看完这个划分你就应该有这么个概念，我没有用过SpringBoot技术，所以从基础篇开始学习；或者我会一点SpringBoot技术，那我从实用篇开始学就好了，就是这个意思。

每个课程单元内容设置不同，目标也不一样，作为学习者如果想达成最佳的学习效果，最好明确自己的学习目标再进行学习，这样目标明确，学习的时候能够更轻松，你就不会在学习的时候纠结如下的问题了。比如学着**基础篇**在想，这个东西是个什么原理啊？这个东西是这么用的，那个东西该怎么用啊？因为原理性的内容统一放置到了**原理篇**讲解了，应用相关的内容统一放到**应用篇**里面讲解，你在**基础篇**阶段纠结也没有用，这一部分不讲这些知识，在**基础篇**先把SpringBoot的基础使用掌握完再说后面的知识吧。

此外还有一点需要说明的是，目前SpringBoot技术发展速度很快，更新速度也很快，因此后续还会对本套课程进行持续更新，特此在三个课程单元的基础上追加一个**番外篇**。番外篇的设置为了解决如下问题：

- 持续更新SpringBoot后续发展出现的新技术
- 讲解部分知识点规模较大的支线知识（例如WebFlux）
- 扩展非实用性知识，扩展学习者视野

每一个课程单元的学习目标如下，请各位查收，在学习的过程中可以阶段性的给自己提个问题，下面列出来的这些学习目标你是否达成了，可以检验你的学习成果。

课程单元	学习目标
基础篇	能够创建SpringBoot工程 基于SpringBoot实现ssm/ssmp整合
应用篇	能够掌握SpringBoot程序多环境开发 能够基于Linux系统发布SpringBoot工程 能够解决线上灵活配置SpringBoot工程的需求 能够基于SpringBoot整合任意第三方技术
原理篇	掌握SpringBoot内部工作流程 理解SpringBoot整合第三方技术的原理 实现自定义开发整合第三方技术的组件
番外篇	掌握SpringBoot整合非常见的第三方技术 掌握相同领域的更多的解决方案，并提升同领域方案设计能力

整体课程包含的内容就是这些啦，要想完成前面这些内容的学习，顺利的达成学习目标，有些东西还是要提前和大家说清楚的。SpringBoot课程不像是Java基础，不管你有没有基础，都可以听一听，这个课程还真不行，需要一定的前置知识。下面给大家列表一些前置知识，如果还有不太会的，需要想办法快速补救一下。

课程前置知识说明

课程单元	前置知识	要求
基础篇	Java基础语法	面向对象，封装，继承，多态，类与接口，集合，IO，网络编程等
基础篇	Spring与SpringMVC	知道Spring是用来管理bean，能够基于Restful实现页面请求交互功能
基础篇	Mybatis与Mybatis-Plus	基于Mybatis和MybatisPlus能够开发出包含基础CRUD功能的标准Dao模块
基础篇	数据库MySQL	能够读懂基础CRUD功能的SQL语句
基础篇	服务器	知道服务器与web工程的关系，熟悉web服务器的基础配置
基础篇	maven	知道maven的依赖关系，知道什么是依赖范围，依赖传递，排除依赖，可选依赖，继承
基础篇	web技术（含vue，ElementUI）	知道vue如何发送ajax请求，如何获取响应数据，如何进行数据模型双向绑定
应用篇	Linux(CenterOS7)	熟悉常用的Linux基础指令，熟悉Linux系统目录结构
应用篇	实用开发技术	缓存：Redis、MongoDB、..... 消息中间件：RocketMq、RabbitMq、.....
原理篇	Spring	了解Spring加载bean的各种方式 知道Spring容器底层工作原理，能够阅读简单的Spring底层源码

看着略微有点多，其实还好吧，如果个别技术真的不会，在学习课程的时候多用心听就好，基础篇是可以跟着学下来了，后面的实用篇和原理篇就比较难了。比如我要在Linux系统下操作，命令我就直接使用了，然后你看不懂可能学习起来就比较心累了。

课程安排就说到这里了，下面进入到SpringBoot基础篇的学习

SpringBoot基础篇

在基础篇中，我给学习者的定位是先上手，能够使用SpringBoot搭建基于SpringBoot的web项目开发，所以内容设置较少，主要包含如下内容：

- SpringBoot快速入门
- SpringBoot基础配置
- 基于SpringBoot整合SSMP

JC-1.快速上手SpringBoot

学习任意一项技术，首先要知道这个技术的作用是什么，不然学完以后，你都不知道什么时候使用这个技术，也就是技术对应的应用场景。SpringBoot技术由Pivotal团队研发制作，功能的话简单概括就是加速Spring程序的开发，这个加速要从如下两个方面来说

- Spring程序初始搭建过程
- Spring程序的开发过程

通过上面两个方面的定位，我们可以产生两个模糊的概念：

1. SpringBoot开发团队认为原始的Spring程序初始搭建的时候可能有些繁琐，这个过程是可以简化的，那原始的Spring程序初始搭建过程都包含哪些东西了呢？为什么觉得繁琐呢？最基本的Spring程序至少有一个配置文件或配置类，用来描述Spring的配置信息，莫非这个文件都可以不写？此外现在企业级开发使用Spring大部分情况下是做web开发，如果做web开发的话，还要在加载web环境时加载时加载指定的spring配置，这都是最基本的需求了，不写的话怎么知道加载哪个配置文件/配置类呢？那换了SpringBoot技术以后呢，这些还要写吗？谜底稍后揭晓，先卖个关子
2. SpringBoot开发团队认为原始的Spring程序开发的过程也有些繁琐，这个过程仍然可以简化。开发过程无外乎使用什么技术，导入对应的jar包（或坐标）然后将这个技术的核心对象交给Spring容器管理，也就是配置成Spring容器管控的bean就可以了。这都是基本操作啊，难道这些东西SpringBoot也能帮我们简化？

带着上面这些疑问我们就着手第一个SpringBoot程序的开发了，看看到底使用SpringBoot技术能简化开发到什么程度。

温馨提示

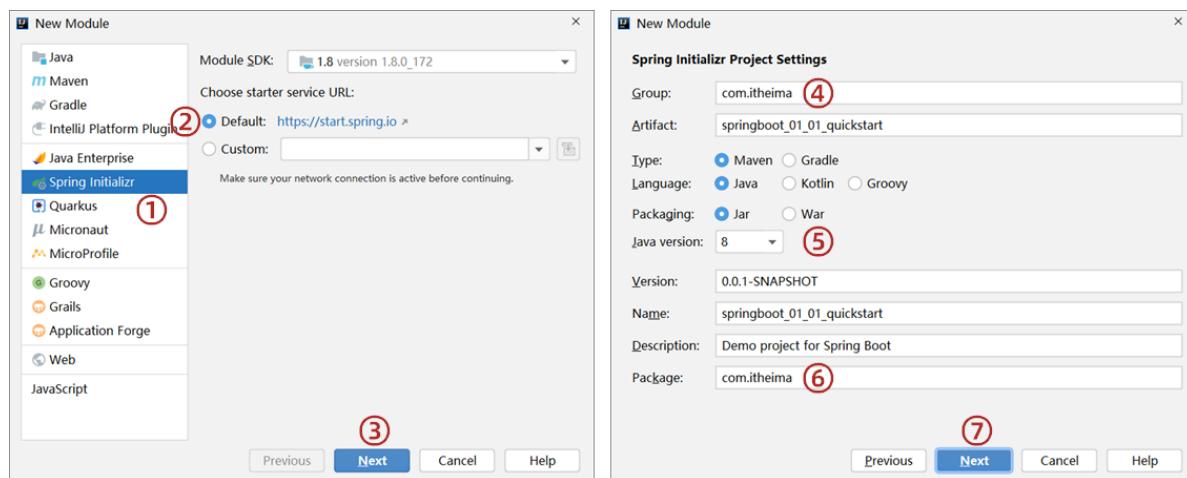
如果对Spring程序的基础开发不太懂的小伙伴，看到这里可以弃坑了，下面的内容学习需要具备Spring技术的知识，硬着头皮学不下去的。

JC-1-1.SpringBoot入门程序制作（一）

下面让我们开始做第一个SpringBoot程序吧，本课程基于Idea2020.3版本制作，使用的Maven版本为3.6.1，JDK版本为1.8。如果你的环境和上述环境不同，可能在操作界面和操作过程中略有不同，只要软件匹配兼容即可（说到这个Idea和Maven，它们两个还真不是什么版本都能搭到一起的，说多了都是泪啊）。

下面使用SpringBoot技术快速构建一个SpringMVC的程序，通过这个过程体会**简化**二字的含义。

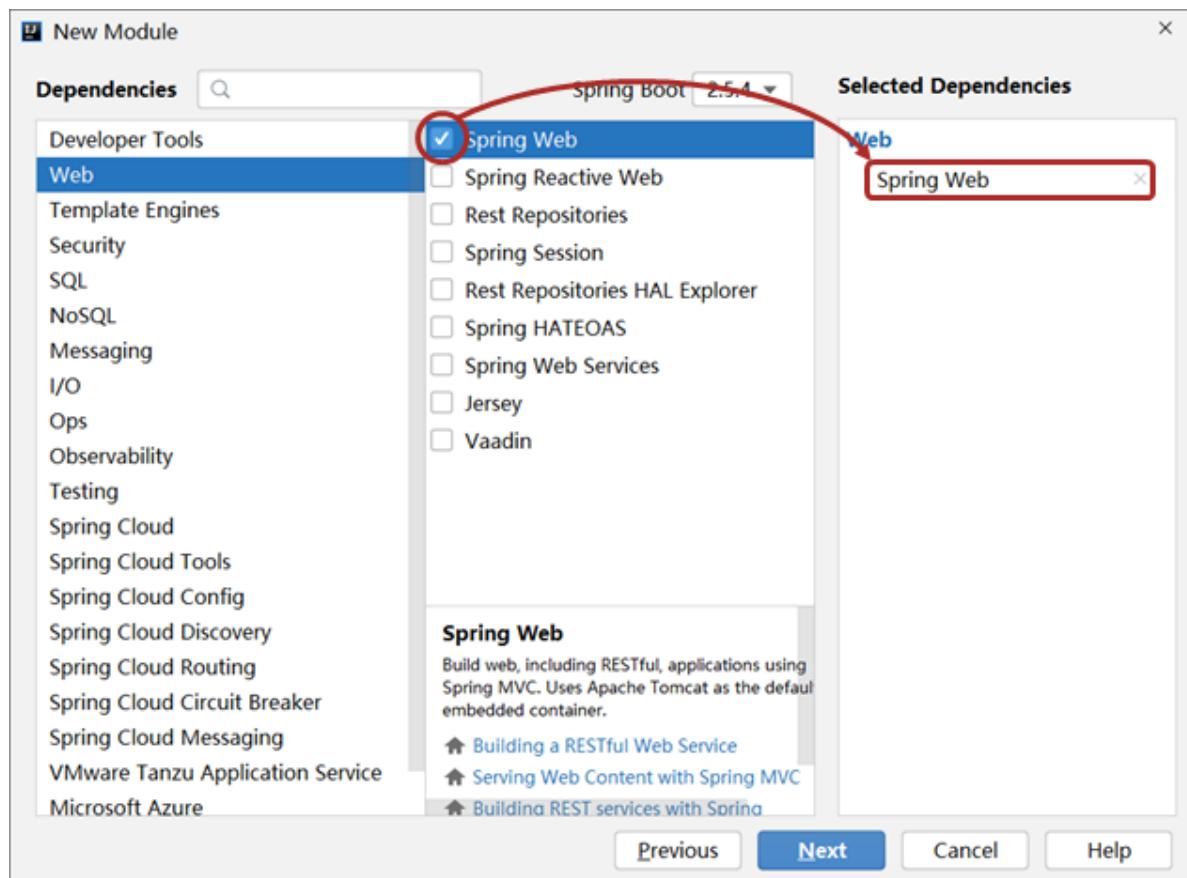
步骤①：创建新模块，选择Spring Initializr，并配置模块相关基础信息



特别关注：第3步点击Next时，Idea需要联网状态才可以进入到后面那一页，如果不能正常联网，就无法正确到达右面那个设置页了，会一直联网转转转。

特别关注：第5步选择java版本和你计算机上安装的JDK版本匹配即可，但是最低要求为JDK8或以上版本，推荐使用8或11。

步骤②：选择当前模块需要使用的技术集



按照要求，左侧选择web，然后在中间选择Spring Web即可，选完右侧就出现了新的内容项，这就表示勾选成功了。

关注：此处选择的SpringBoot的版本使用默认的就可以了，需要说一点，SpringBoot的版本升级速度很快，可能昨天创建工程的时候默认版本是2.5.4，今天再创建工程默认版本就变成2.5.5了，差别不大，无需过于纠结，并且还可以到配置文件中修改对应的版本。

步骤③：开发控制器类

```
//Rest模式  
 @RestController  
 @RequestMapping("/books")  
 public class BookController {  
     @GetMapping  
     public String getById(){  
         System.out.println("springboot is running...");  
         return "springboot is running...";  
     }  
 }
```

入门案例制作的SpringMVC的控制器基于Rest风格开发，当然此处使用原始格式制作SpringMVC的程序也是没有问题的，上例中的@RestController与@GetMapping注解是基于Restful开发的典型注解。

关注：做到这里SpringBoot程序的最基础的开发已经做完了，现在就可以正常的运行Spring程序了。可能有些小伙伴会有疑惑，Tomcat服务器没有配置，Spring也没有配置，什么都没有配置这就能用吗？这就是SpringBoot技术的强大之处。关于内部工作流程后面再说，先专心学习开发过程。

步骤④：运行自动生成的Application类

使用带main方法的java程序的运行形式来运行程序，运行完毕后，控制台输出上述信息。

不难看出，运行的信息中包含了8080的端口，Tomcat这种熟悉的字样，难道这里启动了Tomcat服务器？是的，这里已经启动了。那服务器没有配置，哪里来的呢？后面再说。现在你就可以通过浏览器访问请求的路径，测试功能是否工作正常了。

访问路径: <http://localhost:8080/books>

是不是感觉很神奇？当前效果其实依赖的底层逻辑还是很复杂的，但是从开发者角度来看，目前只有两个文件展现到了开发者面前。

- pom.xml

这是maven的配置文件，描述了当前工程构建时相应的配置信息。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
```

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.5.4</version>
</parent>

<groupId>com.itheima</groupId>
<artifactId>springboot_01_01_quickstart</artifactId>
<version>0.0.1-SNAPSHOT</version>

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

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

```

配置中有两个信息需要关注，一个是parent，也就是当前工程继承了另外一个工程，干什么用的后面再说，还有依赖坐标，干什么用的后面再说。

- Application类

```

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

```

这个类功能很简单，就一句代码，前面运行程序就是运行的这个类。

到这里我们可以大胆推测一下，如果上面这两个文件没有的话，SpringBoot肯定没法玩，看来核心就是这两个文件了。由于是制作第一个SpringBoot程序，先不要关注这两个文件的功能，后面详细讲解内部工作流程。

通过上面的制作，我们不难发现，SpringBoot程序简直太好写了，几乎什么都没写，功能就有了，这也是SpringBoot技术为什么现在这么火的原因，和Spring程序相比，SpringBoot程序在开发的过程中各个层面均具有优势。

类配置文件	Spring	SpringBoot
pom文件中的坐标	手工添加	勾选添加
web3.0配置类	手工制作	无
Spring/SpringMVC配置类	手工制作	无
控制器	手工制作	手工制作

一句话总结一下就是**能少写就少写，能不写就不写**，这就是SpringBoot技术给我们带来的好处，行了，现在你就可以动手做一做SpringBoot程序了，看看效果如何，是否真的帮助你简化开发了。

总结

1. 开发SpringBoot程序在Idea工具中基于联网的前提下可以根据向导快速制作
2. SpringBoot程序需要依赖JDK，版本要求最低为JDK8
3. SpringBoot程序中需要使用某种功能时可以通过勾选的形式选择技术，也可以手工添加对应的要使用的技术（后期讲解）
4. 运行SpringBoot程序通过运行Application程序进行

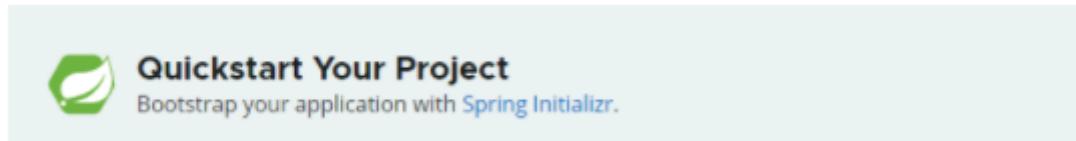
思考

前面制作的时候说过，这个过程必须联网才可以进行，但是有些时候你会遇到一些莫名其妙的问题，比如基于Idea开发时，你会发现你配置了一些坐标，然后Maven下载对应东西的时候死慢死慢的，甚至还会失败。其实这种现象和Idea这款IDE工具有关，万一Idea不能正常访问网络的话，我们是不是就无法制作SpringBoot程序了呢？咱们下一节再说。

JC-1-2.SpringBoot入门程序制作（二）

如果Idea不能正常联网，这个SpringBoot程序就无法制作了吗？开玩笑，世上IDE工具千千万，难道SpringBoot技术还必须基于Idea来做了？这是不可能的。开发SpringBoot程序可以不基于IDE工具进行，在SpringBoot官网中可以直接创建SpringBoot程序。

SpringBoot官网和Spring的官网是在一起的，都是 spring.io。你可以通过项目一级一级的找到SpringBoot技术的介绍页，然后在页面中间部位找到如下内容



步骤①：点击**Spring Initializr**后进入到创建SpringBoot程序界面，接下来就是输入信息的过程，和在Idea中制作是一样的，只是界面发生了变化，根据自己的要求，在左侧选择对应信息和输入对应的信息。

Project
 Maven Project Gradle Project Language
 Java Kotlin Groovy

Spring Boot
 2.6.0 (SNAPSHOT) 2.6.0 (M2) 2.5.5 (SNAPSHOT) 2.5.4
 2.4.11 (SNAPSHOT) 2.4.10

Project Metadata
Group: com.theima
Artifact: springboot_01_02_quickstart
Name: springboot_01_02_quickstart
Description: Demo project for Spring Boot
Package name: com.theima
Packaging: Jar War
Java: 16 11 8

Dependencies
Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

ADD DEPENDENCIES... CTRL + B

GENERATE CTRL + ⌘ EXPLORE CTRL + SPACE SHARE...

步骤②：右侧的**ADD DEPENDENCIES**用于选择使用何种技术，和之前勾选的Spring WEB是在做同一件事，仅仅是界面不同而已，点击后打开网页版的技术选择界面。

Web, Security, JPA, Actuator, Devtools... Press Ctrl for multiple adds

DEVELOPER TOOLS

Spring Native [Experimental]
Incubating support for compiling Spring applications to native executables using the GraalVM native-image compiler.

Spring Boot DevTools
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Lombok
Java annotation library which helps to reduce boilerplate code.

Spring Configuration Processor
Generate metadata for developers to offer contextual help and "code completion" when working with custom configuration keys (ex.application.properties/.yml files).

WEB

Spring Web
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Reactive Web
Build reactive web applications with Spring WebFlux and Netty.

Rest Repositories
Exposing Spring Data repositories over REST via Spring Data REST.

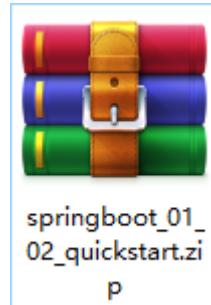
Spring Session
Provides an API and implementations for managing user session information.

Rest Repositories HAL Explorer
Browsing Spring Data REST repositories in your browser.

步骤③：所有信息设置完毕后，点击下面左侧**GENERATE**按钮，生成一个文件包。



步骤④：保存后得到一个压缩文件，这个文件就是创建的SpringBoot工程

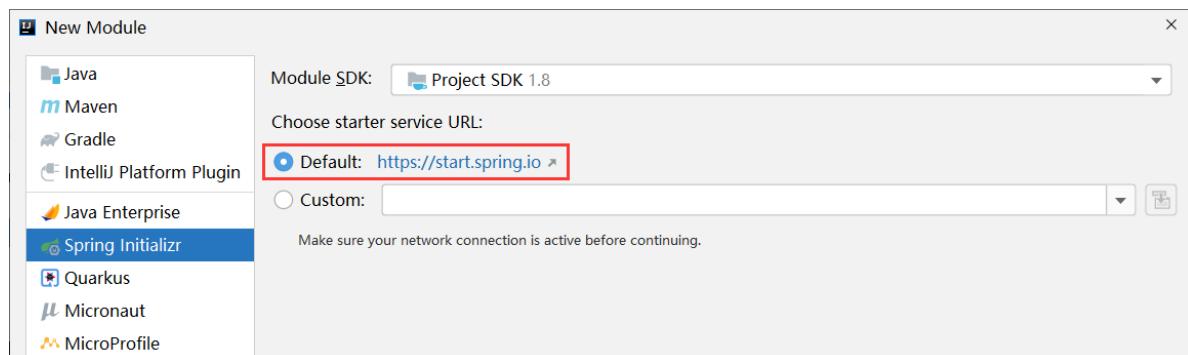


步骤⑤：解压缩此文件得到工程目录，在Idea中导入即可直接使用，和之前在Idea环境下根据向导创建的工程完全一样，你可以创建一个Controller测试一下当前工程是否可用。

温馨提示

做到这里其实可以透漏一个小秘密，Idea工具中创建SpringBoot工程其实连接的就是SpringBoot的官网，还句话说这种方式和第一种方式是一模一样的，只不过Idea把界面给整合了一下，读取Spring官网信息，然后展示到Idea界面中而已，可以通过如下信息比对一下

Idea中创建工程时默认选项



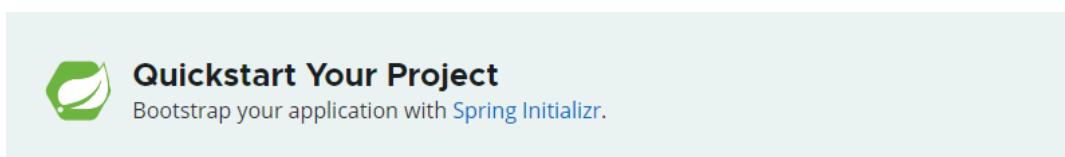
SpringBoot官网创建工程时对应的地址

看看SpringBoot官网创建工程的URL地址，是不是和Idea中使用的URL地址是一样的？

The screenshot shows a browser window with the Spring Initializr URL (<https://start.spring.io>) in the address bar. The page itself displays the Spring Initializr logo and navigation options for creating a Maven or Gradle project in Java, Kotlin, or Groovy.

总结

1. 打开SpringBoot官网，选择Quickstart Your Project中的Spring Initializr。



2. 创建工程。

The screenshot shows the Spring Initializr web interface. At the top, there are sections for 'Project' (Maven Project selected), 'Language' (Java selected), and 'Dependencies' (Spring Web selected). Under 'Project Metadata', fields include Group (com.theima), Artifact (springboot_01_02_quickstart), Name (springboot_01_02_quickstart), Description (Demo project for Spring Boot), Package name (com.theima), and Packaging (Jar selected). Java version 8 is chosen. At the bottom, there are three buttons: 'GENERATE' (CTRL + ⌘), 'EXPLORE' (CTRL + SPACE), and 'SHARE...'. A tooltip for 'Spring Web' indicates it's a build web application using Spring MVC with Apache Tomcat.

3. 保存项目文件。

GENERATE CTRL + ⌘

EXPLORE CTRL + SPACE

SHARE...

4. 解压项目，通过IDE导入项目后进行编辑使用。

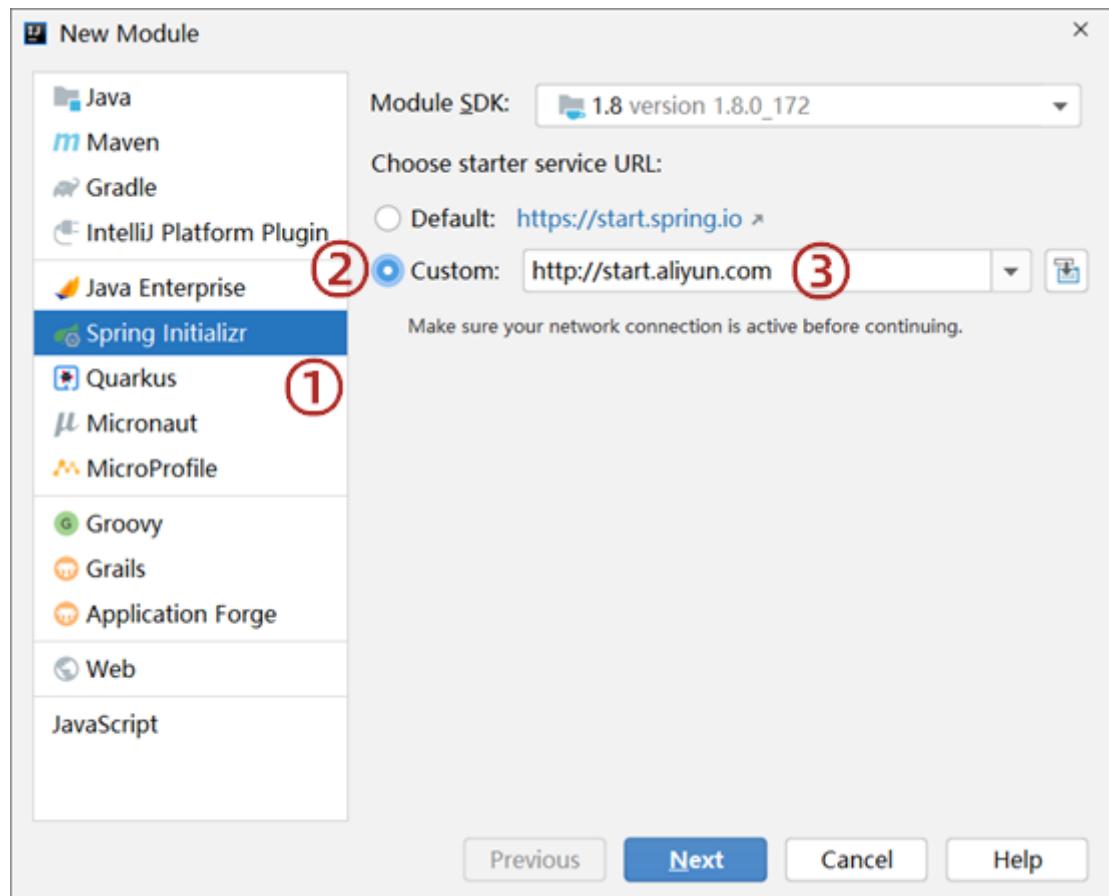
思考

现在创建工程靠的是访问国外的Spring主站，但是互联网信息的访问是可以被约束的，如果一天这个网站你在国内无法访问了，那前面这两种方式就无法创建SpringBoot工程了，这时候又该怎么解决这个问题呢？咱们下一节再说。

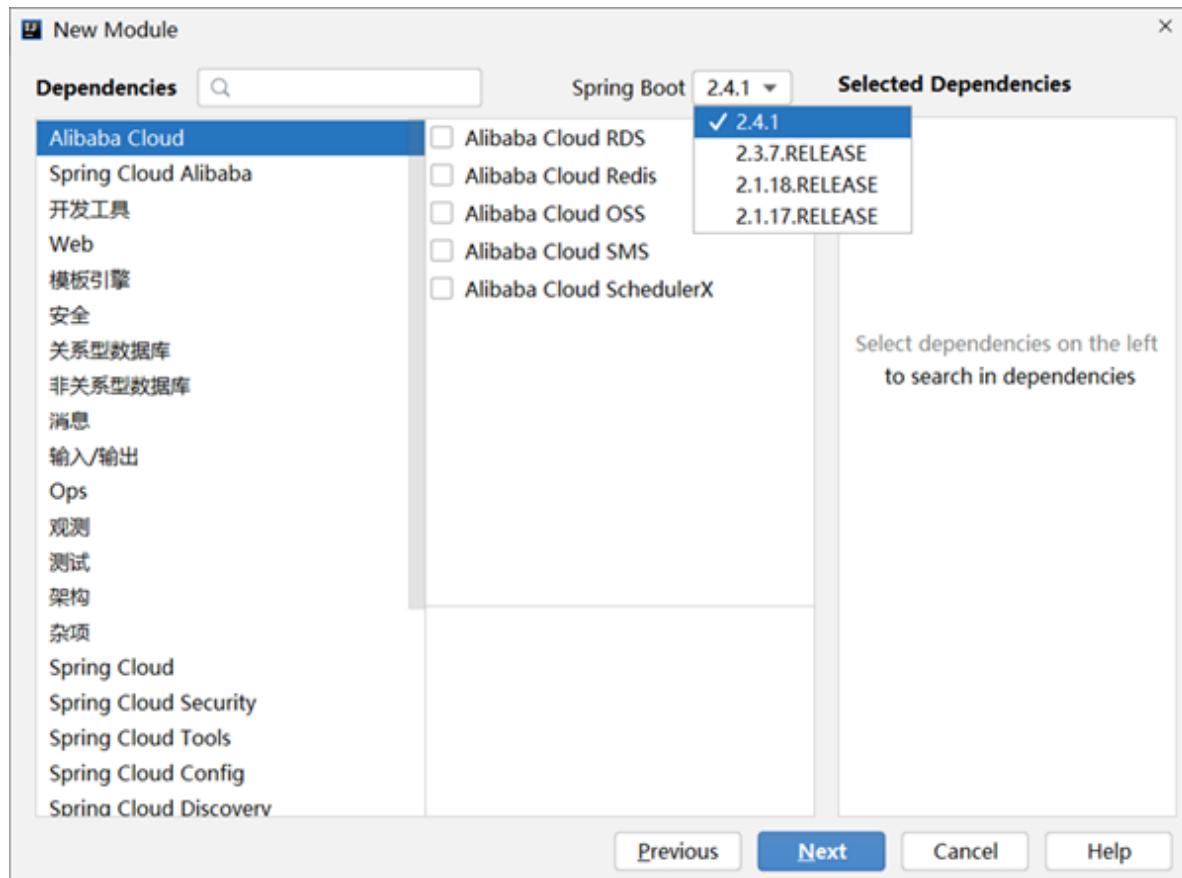
JC-1-3.SpringBoot入门程序制作（三）

前面提到网站如果被限制访问了，该怎么办？开动脑筋想一想，不管是方式一还是方式二其实走的都是同一个路线，就是通过SpringBoot官网创建SpringBoot工程，假如国内有这么一个网站也能提供这样的功能，是不是就解决了呢？必然的嘛，新的问题又来了，国内有提供这样功能的网站吗？还真有，阿里提供了一个，下面问题就简单了，网址告诉我们就OK了，没错，就是这样。

创建工程时，切换选择starter服务路径，然后手工输入阿里云地址即可，地址：<http://start.aliyun.com>或<https://start.aliyun.com>



阿里为了便自己公司开发使用，特此在依赖坐标中添加了一些阿里自主的技术，也是为了推广自己的技术吧，所以在依赖选择列表中，你有了更多的选择。此外，阿里提供的地址更符合国内开发者的使用习惯，里面有一些SpringBoot官网上没有给出的坐标，大家可以好好看一看。



不过有一点需要说清楚，阿里云地址默认创建的SpringBoot工程版本是**2.4.1**，所以如果你想更换其他的版本，创建项目后在pom文件中手工修改即可，别忘了刷新一下，加载新版本信息。

注意：阿里云提供的工程创建地址初始化完毕后和使用SpringBoot官网创建出来的工程略有区别，主要是在配置文件的形式上有区别，这个信息在后面讲解SpringBoot程序的执行流程时给大家揭晓。

总结

1. 选择start来源为自定义URL
2. 输入阿里云starter地址
3. 创建项目

思考

做到这里我们已经有了三种方式创建SpringBoot工程，但是每种方式都要求你必须能上网才能创建工程。假如有一天，你加入了一个保密级别比较高的项目组，整个项目组没有外网，这个事情是不是就不能做了呢？咱们下一节再说。

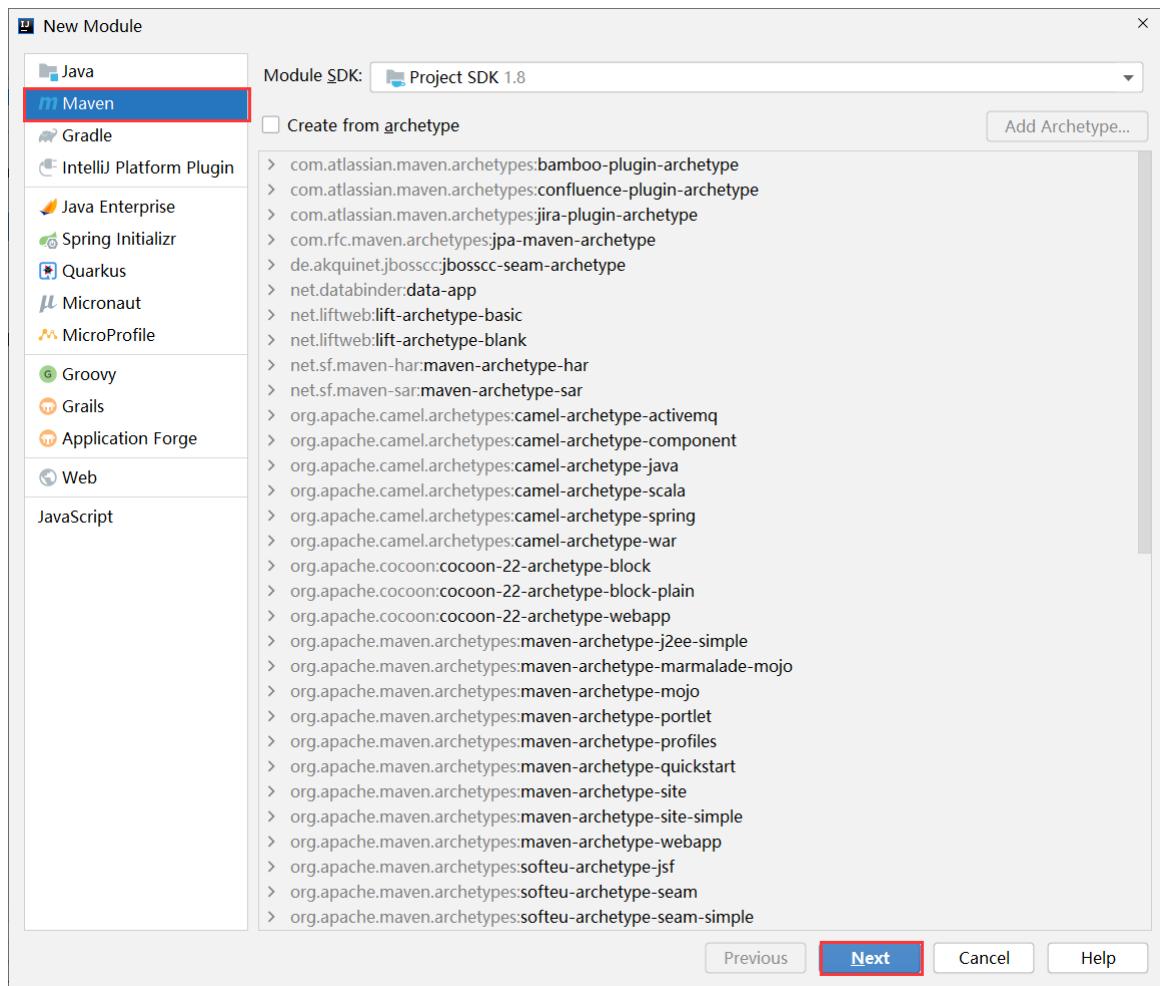
JC-1-4.SpringBoot入门程序制作（四）

不能上网，还想创建SpringBoot工程，能不能做呢？能做，但是你要先问问自己联网和不联网到底差别是什么？这个差别找到以后，你就发现，你把联网要干的事情都提前准备好，就无需联网了。

联网做什么呢？首先SpringBoot工程也是基于Maven构建的，而Maven工程中如果加载一些工程需要使用又不存在的东西时，就要联网去下载。其实SpringBoot工程创建的时候就是要去下载一些必要的组件。如果把这些东西提前准备好呢？是的，就是这样。

下面就手工创建一个SpringBoot工程，如果需要使用的东西提前保障在maven仓库中存在，整个过程就可以不依赖联网环境了。不过咱们已经用3种方式创建了SprongBoot工程了，所以下面也没什么东西需要下载了。

步骤①：创建工程时，选择创建普通Maven工程。



步骤②：参照标准SpringBoot工程的pom文件，书写自己的pom文件即可。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.4</version>
    </parent>

    <groupId>com.itheima</groupId>
    <artifactId>springboot_01_04_quickstart</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

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

```
</dependency>
</dependencies>

</project>
```

用什么写什么，不用的都可以不写。当然，现在小伙伴们可能还不知道用什么和不用什么，最简单的就是复制粘贴了，随着后面的学习，你就知道哪些可以省略了。此处我删减了一些目前不是必须的东西，一样能用。核心的内容有两条，一个是继承了一个父工程，另外添加了一个依赖。

步骤③：之前运行SpringBoot工程需要一个类，这个缺不了，自己手写一个就行了，建议按照之前的目录结构来创建，先别玩花样，先学走后学跑。类名可以自定义，关联的名称同步修改即可。

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

关注：类上面的注解@SpringBootApplication千万别丢了，这个是核心，后面再介绍。

关注：类名可以自定义，只要保障下面代码中使用的类名和你自己定义的名称一样即可，也就是run方法中的那个class对应的名字。

步骤④：下面就可以自己创建一个Controller测试一下是否能用了，和之前没有差别的。

看到这里其实应该能够想明白了，通过向导或者网站创建的SpringBoot工程其实就是帮你写了一些代码，而现在是自己手写，写的内容都一样，仅此而已。

温馨提示

如果你的计算机上从来没有创建成功过SpringBoot工程，自然也就没有下载过SpringBoot对应的坐标相关的资源，那用手写创建的方式在不联网的情况下肯定该是不能用的。所谓手写，其实就是自己写别人帮你生成的东西，但是引用的坐标对应的资源必须保障maven仓库里面有才行，如果没有，还是要去下载的。

总结

1. 创建普通Maven工程
2. 继承spring-boot-starter-parent
3. 添加依赖spring-boot-starter-web
4. 制作引导类Application

到这里已经学习了4种创建SpringBoot工程的方式，其实本质是一样的，都是根据SpringBoot工程的文件格式要求，通过不同时方式生成或者手写得到对应的文件，效果完全一样。

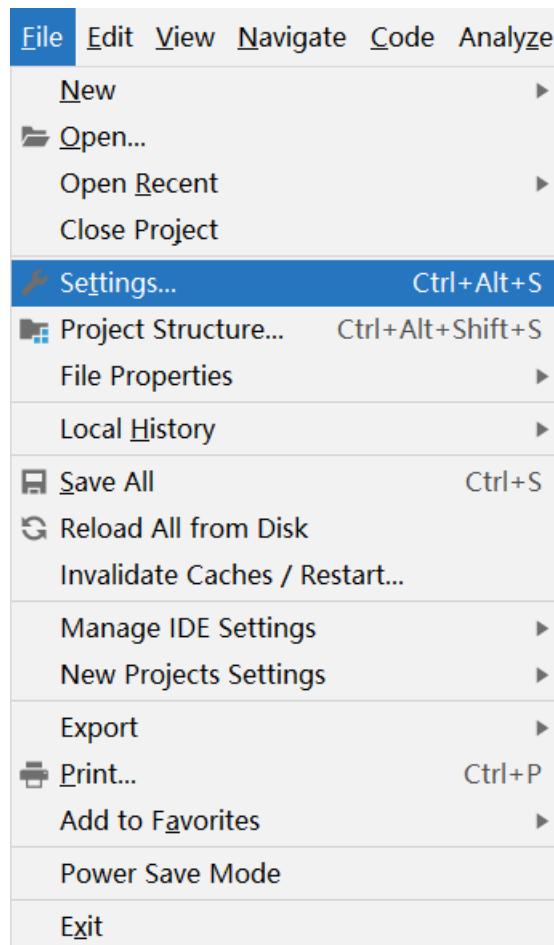
教你一招：在Idea中隐藏指定文件/文件夹

创建SpringBoot工程时，使用SpringBoot向导也好，阿里云也罢，其实都是为了一个目的，得到一个标准的SpringBoot工程文件结构。这个时候就有新的问题出现了，标准的工程结构中包含了一些未知的文件夹，在开发的时候看起来特别别扭，这一节就来说说这些文件怎么处理。

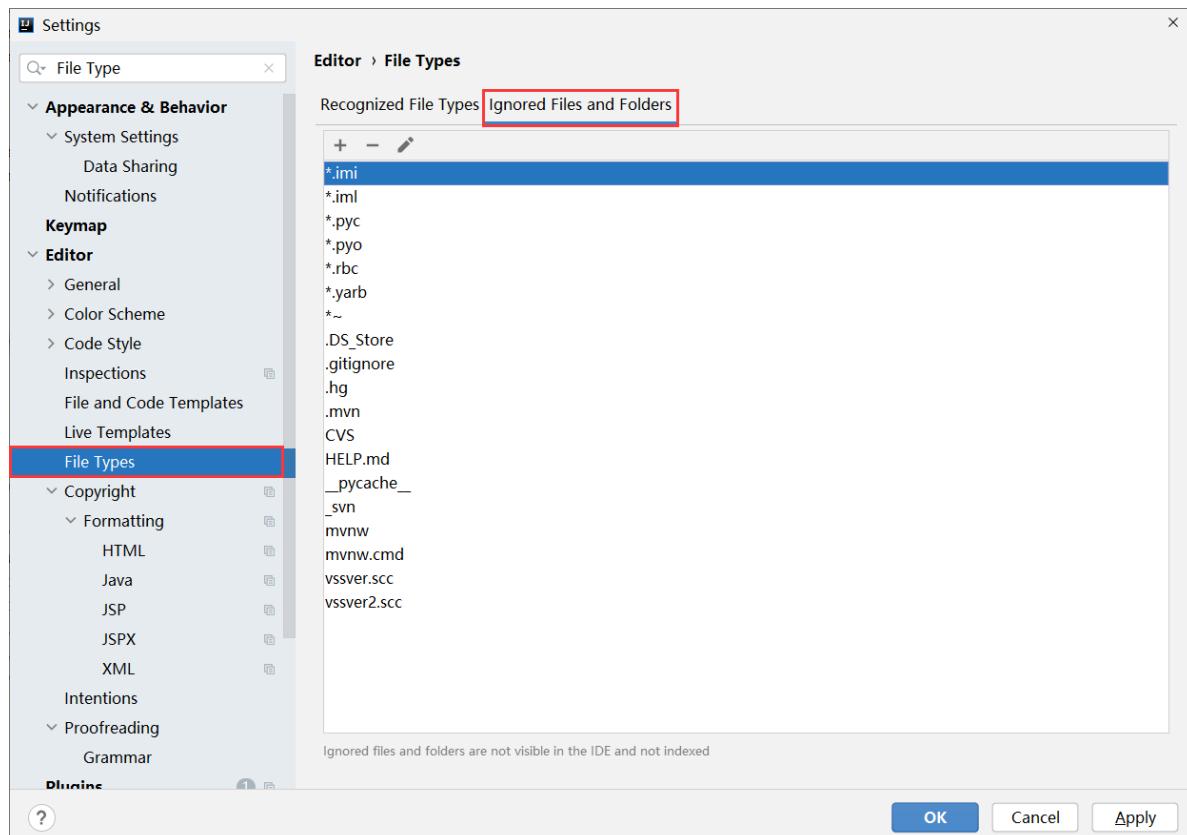
处理方案无外乎两种，如果你对每一个文件/目录足够了解，有用的留着，没有用的完全可以删除掉。或者不删除，但是看着别扭，就设置文件为看不到就行了。删除不说了，选中后直接Delete掉就好了，这一节说说如何隐藏指定的文件或文件夹信息。

既然是在Idea下做隐藏功能，肯定隶属于Idea的设置，设置方式如下。

步骤①：打开设置，【Files】→【Settings】。



步骤②：打开文件类型设置界面后，【Editor】→【File Types】→【Ignored Files and Folders】，忽略文件或文件夹显示。



步骤③：添加你要隐藏的文件名称或文件夹名称，可以使用*号通配符，表示任意，设置完毕即可。

到这里就做完了，其实就是Idea的一个小功能

总结

1. Idea中隐藏指定文件或指定类型文件

1. 【Files】→【Settings】
2. 【Editor】→【File Types】→【Ignored Files and Folders】
3. 输入要隐藏的名称，支持*号通配符
4. 回车确认添加

JC-1-5.SpringBoot简介

入门案例做完了，这个时候回忆一下咱们之前说的SpringBoot的功能是什么还记得吗？加速Spring程序的开发，现在是否深有体会？再来看SpringBoot技术的设计初衷就很容易理解了。

SpringBoot是由Pivotal团队提供的全新框架，其设计目的是用来**简化Spring应用的初始搭建以及开发过程。**

都简化了哪些东西呢？其实就是针对原始的Spring程序制作的两个方面进行了简化：

- Spring程序缺点
 - 依赖设置繁琐
 - 以前写Spring程序，使用的技术都要自己一个一个的写，现在不需要了，如果做过原始SpringMVC程序的小伙伴应该知道，写SpringMVC程序，最基础的spring-web和spring-webmvc这两个坐标是必须的，就这还不包含你用json啊等等这些坐标，现在呢？一个坐标搞定了。
 - 配置繁琐
 - 以前写配置类或者配置文件，然后用什么东西就要自己写加载bean这些东西，现在呢？什么都没写，照样能用。

回顾

通过上面两个方面的定位，我们可以产生两个模糊的概念：

1. SpringBoot开发团队认为原始的Spring程序初始搭建的时候可能有些繁琐，这个过程是可以简化的，那原始的Spring程序初始搭建过程都包含哪些东西了呢？为什么觉得繁琐呢？最基本的Spring程序至少有一个配置文件或配置类，用来描述Spring的配置信息，莫非这个文件都可以不写？此外现在企业级开发使用Spring大部分情况下是做web开发，如果做web开发的话，还要在加载web环境时加载时加载指定的spring配置，这都是最基本的需求了，不写的话怎么知道加载哪个配置文件/配置类呢？那换了SpringBoot技术以后呢，这些还要写吗？谜底稍后揭晓，先卖个关子
2. SpringBoot开发团队认为原始的Spring程序开发的过程也有些繁琐，这个过程仍然可以简化。开发过程无外乎使用什么技术，导入对应的jar包（或坐标）然后将这个技术的核心对象交给Spring容器管理，也就是配置成Spring容器管控的bean就可以了。这都是基本操作啊，难道这些东西SpringBoot也能帮我们简化？

再来看看前面提出的两个问题，已经有答案了，都简化了，都不用写了，这就是SpringBoot给我们带来的好处。这些简化操作在SpringBoot中有专业的用语，也是SpringBoot程序的核心功能及优点：

- 起步依赖（简化依赖配置）
 - 依赖配置的书写简化就是靠这个起步依赖达成的。
- 自动配置（简化常用工程相关配置）

- 配置过于繁琐，使用自动配置就可以做相应的简化，但是内部还是很复杂的，后面具体展开说。
- 辅助功能（内置服务器，……）
 - 除了上面的功能，其实SpringBoot程序还有其他的一些优势，比如我们没有配置Tomcat服务器，但是能正常运行，这是SpringBoot入门程序中一个可以感知到的功能，也是SpringBoot的辅助功能之一。一个辅助功能都能做的这么6，太牛了。

下面结合入门程序来说说这些简化操作都在哪些方面进行体现的，一共分为4个方面

- parent
- starter
- 引导类
- 内嵌tomcat

parent

SpringBoot关注到开发者在进行开发时，往往对依赖版本的选择具有固定的搭配格式，并且这些依赖版本的选择还不能乱搭配。比如A技术的2.0版，在与B技术进行配合使用时，与B技术的3.5版可以合作在一起工作，但是和B技术的3.7版合作开发使用时就有冲突。其实很多开发者都一直想做一件事情，就是将各种各样的技术配合使用的常见依赖版本进行收集整理，制作出了最合理的依赖版本配置方案，这样使用起来就方便多了。

SpringBoot一看这种情况so easy啊，于是将所有的技术版本的常见使用方案都给开发者整理了出来，以后开发者使用时直接用它提供的版本方案，就不用担心冲突问题了，相当于SpringBoot做了无数个技术版本搭配的列表，这个技术搭配列表的名字叫做**parent**。

parent自身具有很多个版本，每个**parent**版本中包含有几百个其他技术的版本号，不同的parent间使用的各种技术的版本号有可能会发生变化。当开发者使用某些技术时，直接使用SpringBoot提供的**parent**就行了，由**parent**帮助开发者统一的进行各种技术的版本管理。

比如你现在要使用Spring配合MyBatis开发，没有parent之前怎么做呢？选个Spring的版本，再选个MyBatis的版本，再把这些技术使用时关联的其他技术的版本逐一确定下来。当你Spring的版本发生变化需要切换时，你的MyBatis版本有可能也要跟着切换，关联技术呢？可能都要切换，而且切换后还可能出现其他问题。现在这一切工作都可以交给parent来做了。你无需关注这些技术间的版本冲突问题，你只需要关注你用什么技术就行了，冲突问题由**parent**负责处理。

有人可能会提出来，万一**parent**给我导入了一些我不想使用的依赖怎么办？记清楚，这一点很关键，**parent**仅仅帮我们进行版本管理，它不负责帮你导入坐标，说白了用什么还是你自己定，只不过版本不需要你管理了。整体上来说，**使用parent可以帮助开发者进行版本的统一管理**。

关注：parent定义出来以后，并不是直接使用的，仅仅给了开发者一个说明书，但是并没有实际使用，这个一定要确认清楚。

那SpringBoot又是如何做到这一点的呢？可以查阅SpringBoot的配置源码，看到这些定义。

- 项目中的pom.xml中继承了一个坐标

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.5.4</version>
</parent>
```

- 打开后可以查阅到其中又继承了一个坐标

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.5.4</version>
</parent>
```

- 这个坐标中定义了两组信息

第一组是各式各样的依赖版本号属性，下面列出依赖版本属性的局部，可以看的出来，定义了若干个技术的依赖版本号。

```
<properties>
  <activemq.version>5.16.3</activemq.version>
  <aspectj.version>1.9.7</aspectj.version>
  <assertj.version>3.19.0</assertj.version>
  <commons-codec.version>1.15</commons-codec.version>
  <commons-dbcp2.version>2.8.0</commons-dbcp2.version>
  <commons-lang3.version>3.12.0</commons-lang3.version>
  <commons-pool.version>1.6</commons-pool.version>
  <commons-pool2.version>2.9.0</commons-pool2.version>
  <h2.version>1.4.200</h2.version>
  <hibernate.version>5.4.32.Final</hibernate.version>
  <hibernate-validator.version>6.2.0.Final</hibernate-validator.version>
  <httpclient.version>4.5.13</httpclient.version>
  <jackson-bom.version>2.12.4</jackson-bom.version>
  <javax-jms.version>2.0.1</javax-jms.version>
  <javax-json.version>1.1.4</javax-json.version>
  <javax-websocket.version>1.1</javax-websocket.version>
  <jetty-e1.version>9.0.48</jetty-e1.version>
  <junit.version>4.13.2</junit.version>
</properties>
```

第二组是各式各样的依赖坐标信息，可以看出依赖坐标定义中没有具体的依赖版本号，而是引用了第一组信息中定义的依赖版本属性值.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

关注: 上面的依赖坐标定义是出现在标签中的，是对引用坐标的依赖管理，并不是实际使用的坐标。因此当你的项目中继承了这组parent信息后，在不使用对应坐标的情况下，前面的这组定义是不会具体导入某个依赖的。

关注: 因为在maven中继承机会只有一次，上述继承的格式还可以切换成导入的形式进行，并且在阿里云的starter创建工程时就使用了此种形式。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>${spring-boot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

总结

1. 开发SpringBoot程序要继承spring-boot-starter-parent
2. spring-boot-starter-parent中定义了若干个依赖管理
3. 继承parent模块可以避免多个依赖使用相同技术时出现依赖版本冲突
4. 继承parent的形式也可以采用引入依赖的形式实现效果

思考

parent中定义了若干个依赖版本管理，但是也没有使用，那这个设定也就不生效啊，究竟谁在使用这些定义呢？

starter

SpringBoot关注到实际开发时，开发者对于依赖坐标的使用往往都有一些固定的组合方式，比如使用spring-webmvc就一定要使用spring-web。每次都要固定搭配着写，非常繁琐，而且格式固定，没有任何技术含量。

SpringBoot一看这种情况，看来需要给开发者带来一些帮助了。安排，把所有的技术使用的固定搭配格式都给开发出来，以后你用某个技术，就不用每次写一堆依赖了，还容易写错，我给你做一个东西，代表一堆东西，开发者使用的时候，直接用我做好的这个东西就好了，对于这样的固定技术搭配，SpringBoot给它起了个名字叫做**starter**。

starter定义了使用某种技术时对于依赖的固定搭配格式，也是一种最佳解决方案，**使用starter可以帮助开发者减少依赖配置。**

这个东西其实在入门案例里面已经使用过了，入门案例中的web功能就是使用这种方式添加依赖的。可以查阅SpringBoot的配置源码，看到这些定义。

- 项目中的pom.xml定义了使用SpringMVC技术，但是并没有写SpringMVC的坐标，而是添加了一个名字中包含starter的依赖

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

- 在spring-boot-starter-web中又定义了若干个具体依赖的坐标

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <version>2.5.4</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-json</artifactId>
        <version>2.5.4</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <version>2.5.4</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>5.3.9</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.9</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

之前提到过开发SpringMVC程序需要导入spring-webmvc的坐标和spring整合web开发的坐标，就是上面这组坐标中的最后两个了。

但是我们发现除了这两个坐标，还有其他的坐标。比如第二个，叫做spring-boot-starter-json。看名称就知道，这个是与json有关的坐标了，但是看名字发现和最后两个又不太一样，它的名字中也有starter，打开看看里面有什么？

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <version>2.5.4</version>
        <scope>compile</scope>
    </dependency>
```

```

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.3.9</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.4</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jdk8</artifactId>
    <version>2.12.4</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
    <version>2.12.4</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.module</groupId>
    <artifactId>jackson-module-parameter-names</artifactId>
    <version>2.12.4</version>
    <scope>compile</scope>
</dependency>
</dependencies>

```

我们可以发现，这个starter中又包含了若干个坐标，其实就是使用SpringMVC开发通常都会使用到Json，使用json又离不开这里面定义的这些坐标，看来还真是方便，SpringBoot把我们开发中使用的东西能用到的都给提前做好了。你仔细看完会发现，里面有一些你没用过的。的确会出现这种过量导入的可能性，没关系，可以通过maven中的排除依赖剔除掉一部分。不过你不管它也没事，大不了就是过量导入呗。

到这里基本上得到了一个信息，使用starter可以帮开发者快速配置依赖关系。以前写依赖3个坐标的，现在写导入一个就搞定了，就是加速依赖配置的。

starter与parent的区别

朦朦胧胧中感觉starter与parent好像都是帮助我们简化配置的，但是功能又不一样，梳理一下。

starter是一个坐标中定了若干个坐标，以前写多个的，现在写一个，**是用来减少依赖配置的书写量的。**

parent是定义了几百个依赖版本号，以前写依赖需要自己手工控制版本，现在由SpringBoot统一管理，这样就不存在版本冲突了，**是用来减少依赖冲突的。**

实际开发应用方式

- 实际开发中如果需要用什么技术，先去找有没有这个技术对应的starter
 - 如果有对应的starter，直接写starter，而且无需指定版本，版本由parent提供
 - 如果没有对应的starter，手写坐标即可

- 实际开发中如果发现坐标出现了冲突现象，确认你要使用的可行的版本号，使用手工书写的方式添加对应依赖，覆盖SpringBoot提供给我们的配置管理
 - 方式一：直接写坐标
 - 方式二：覆盖中定义的版本号，就是下面这堆东西了，哪个冲突了覆盖哪个就OK了

```
<properties>
    <activemq.version>5.16.3</activemq.version>
    <aspectj.version>1.9.7</aspectj.version>
    <assertj.version>3.19.0</assertj.version>
    <commons-codec.version>1.15</commons-codec.version>
    <commons-dbcp2.version>2.8.0</commons-dbcp2.version>
    <commons-lang3.version>3.12.0</commons-lang3.version>
    <commons-pool.version>1.6</commons-pool.version>
    <commons-pool2.version>2.9.0</commons-pool2.version>
    <h2.version>1.4.200</h2.version>
    <hibernate.version>5.4.32.Final</hibernate.version>
    <hibernate-validator.version>6.2.0.Final</hibernate-validator.version>
    <httpclient.version>4.5.13</httpclient.version>
    <jackson-bom.version>2.12.4</jackson-bom.version>
    <javax-jms.version>2.0.1</javax-jms.version>
    <javax-json.version>1.1.4</javax-json.version>
    <javax-websocket.version>1.1</javax-websocket.version>
    <jetty-el.version>9.0.48</jetty-el.version>
    <junit.version>4.13.2</junit.version>
</properties>
```

温馨提示

SpringBoot官方给出了好多个starter的定义，方便我们使用，而且名称都是如下格式

命名规则：`spring-boot-starter-技术名称`

所以后期见了`spring-boot-starter-aaa`这样的名字，这就是SpringBoot官方给出的starter定义。那非官方定义的也有吗？有的，具体命名方式到整合技术的章节再说。

总结

- 开发SpringBoot程序需要导入坐标时通常导入对应的starter
- 每个不同的starter根据功能不同，通常包含多个依赖坐标
- 使用starter可以实现快速配置的效果，达到简化配置的目的

引导类

配置说完了，我们发现SpringBoot确实帮助我们减少了很多配置工作，下面说一下程序是如何运行的。目前程序运行的入口就是SpringBoot工程创建时自带的那个类，也就是带有`main`方法的那个类，运行这个类就可以启动SpringBoot工程的运行。

```
@SpringBootApplication  
public class Springboot0101QuickstartApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(Springboot0101QuickstartApplication.class, args);  
    }  
}
```

SpringBoot本身是为了加速Spring程序的开发的，而Spring程序运行的基础是需要创建Spring容器对象（IoC容器）并将所有的对象放置到Spring容器中管理，也就是一个一个的Bean。现在改用SpringBoot加速开发Spring程序，这个容器还在吗？这个疑问不用说，一定在。其实当前这个类运行后就会产生一个Spring容器对象，并且可以将这个对象保存起来，通过容器对象直接操作Bean。

```
@SpringBootApplication  
public class QuickstartApplication {  
    public static void main(String[] args) {  
        ConfigurableApplicationContext ctx =  
        SpringApplication.run(QuickstartApplication.class, args);  
        BookController bean = ctx.getBean(BookController.class);  
        System.out.println("bean=====》" + bean);  
    }  
}
```

通过上述操作不难看出，其实SpringBoot程序启动还是创建了一个Spring容器对象。当前运行的这个类在SpringBoot程序中是所有功能的入口，称为**引导类**。

作为一个引导类最典型的特征就是当前类上方声明了一个注解**@SpringBootApplication**。

总结

1. SpringBoot工程提供引导类用来启动程序
2. SpringBoot工程启动后创建并初始化Spring容器

思考

程序现在已经运行了，通过引导类的main方法运行了起来。但是运行java程序不应该是执行完就结束了么？但是我们现在明显是启动了一个web服务器啊，不然网页怎么能正常访问呢？这个服务器是在哪里写的呢？

内嵌tomcat

当前我们做的SpringBoot入门案例勾选了Spring-web的功能，并且导入了对应的starter。

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

SpringBoot发现，既然你要做web程序，肯定离不开使用web服务器，这样吧，帮人帮到底，送佛送到西，我帮你搞一个web服务器，你要愿意用的，直接使用就好了。SpringBoot又琢磨，提供一种服务器万一不满足开发者需要呢？干脆我再多给你几种选择，你随便切换。万一你不想用我给你提供的，也行，你可以自己搞。

由于这个功能不属于程序的主体功能，可用可不用，于是乎SpringBoot将其定位成辅助功能，别小看这么一个辅助功能，它可是帮我们开发者又减少了好多的设置性工作。

下面就围绕着这个内置的web服务器，也可以说是内置的tomcat服务器来研究几个问题：

1. 这个服务器在什么位置定义的
2. 这个服务器是怎么运行的
3. 这个服务器如果想换怎么换？虽然这个需求很垃圾，搞得开发者会好多web服务器一样，用别人提供的好的不香么？非要自己折腾

内嵌Tomcat定义位置

说到定义的位置，我们就想，如果我们不开发web程序，用的着web服务器吗？肯定用不着啊。那如果这个东西被加入到你的程序中，伴随着什么技术进来的呢？肯定是web相关的功能啊，没错，就是前面导入的web相关的starter做的这件事。

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

打开web对应的starter查看导入了哪些东西。

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <version>2.5.4</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-json</artifactId>
        <version>2.5.4</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <version>2.5.4</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>5.3.9</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.9</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

第三个依赖就是tomcat对应的东西了，居然也是一个starter，再打开看看。

```
<dependencies>
    <dependency>
        <groupId>jakarta.annotation</groupId>
        <artifactId>jakarta.annotation-api</artifactId>
        <version>1.3.5</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-core</artifactId>
        <version>9.0.52</version>
        <scope>compile</scope>
        <exclusions>
            <exclusion>
                <artifactId>tomcat-annotations-api</artifactId>
                <groupId>org.apache.tomcat</groupId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-el</artifactId>
        <version>9.0.52</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-websocket</artifactId>
        <version>9.0.52</version>
        <scope>compile</scope>
        <exclusions>
            <exclusion>
                <artifactId>tomcat-annotations-api</artifactId>
                <groupId>org.apache.tomcat</groupId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

这里面有一个核心的坐标，tomcat-embed-core，叫做tomcat内嵌核心。就是这个东西把tomcat功能引入到了我们的程序中的。目前解决了第一个问题，找到根儿了，谁把tomcat引入到程序中的？spring-boot-starter-web中的spring-boot-starter-tomcat做的。之所以你感觉很奇妙的原因就是，这个东西是默认加入到程序中了，所以感觉很神奇，居然什么都不做，就有了web服务器对应的功能。再来说明第二个问题，这个服务器是怎么运行的。

内嵌Tomcat运行原理

Tomcat服务器是一款软件，而且是一款使用java语言开发的软件，熟悉tomcat的话应该知道tomcat安装目录中保存有很多jar文件。

下面的问题来了，既然是使用java语言开发的，运行的时候肯定符合java程序运行的原理，java程序运行靠的是什么？对象呀，一切皆对象，万物皆对象。那tomcat运行起来呢？也是对象啊。

如果是对象，那Spring容器是用来管理对象的，这个对象能交给Spring容器管理吗？把吗去掉，是个对象都可以交给Spring容器管理，行了，这下通了，tomcat服务器运行其实是以对象的形式在Spring容器中运行的。怪不得我们没有安装这个tomcat但是还能用，闹了白天这东西最后是以一个对象的形式存在，保存在Spring容器中悄悄运行的。具体运行的是什么呢？其实就是上前面提到的那个tomcat内嵌核心。

```
<dependencies>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-core</artifactId>
        <version>9.0.52</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

那既然是个对象，如果把这个对象从Spring容器中去掉是不是就没有web服务器的功能呢？是这样的，通过依赖排除可以去掉这个web服务器功能。

```
<dependencies>
    <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>
</dependencies>
```

上面对web-starter做了一个操作，使用maven的排除依赖去掉了使用tomcat的starter。这下好了，容器中肯定没有这个对象了，重新启动程序可以观察到程序运行了，但是并没有像之前那样运行后是一个一直运行的服务，而是直接停掉了，就是这个原因。

更换内嵌Tomcat

那根据上面的操作我们思考是否可以换个服务器呢？必须的嘛。根据SpringBoot的工作机制，用什么技术，加入什么依赖就行了。SpringBoot提供了3款内置的服务器：

- tomcat(默认)：apache出品，粉丝多，应用面广，负载了若干较重的组件
- jetty：更轻量级，负载性能远不及tomcat
- undertow：负载性能勉强跑赢tomcat

想用哪个，加个坐标就OK。前提是把tomcat排除掉，因为tomcat是默认加载的。

```
<dependencies>
    <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>
</dependencies>
```

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

现在就已经成功替换了web服务器，核心思想就是用什么加入对应坐标就可以了。如果有starter，优先使用starter。

总结

1. 内嵌Tomcat服务器是SpringBoot辅助功能之一
2. 内嵌Tomcat工作原理是将Tomcat服务器作为对象运行，并将该对象交给Spring容器管理
3. 变更内嵌服务器思想是去除现有服务器，添加全新的服务器

到这里第一章快速上手SpringBoot就结束了，这一章我们学习了两大块知识

1. 使用了4种方式制作了SpringBoot的入门程序，不管是哪一种，其实内部都是一模一样的
2. 学习了入门程序的工作流程，知道什么是parent，什么是starter，这两个东西是怎么配合工作的，以及我们的程序为什么启动起来是一个tomcat服务器等等

第一章到这里就结束了，再往下学习就要去基于会创建SpringBoot工程的基础上，研究SpringBoot工程的具体细节了。

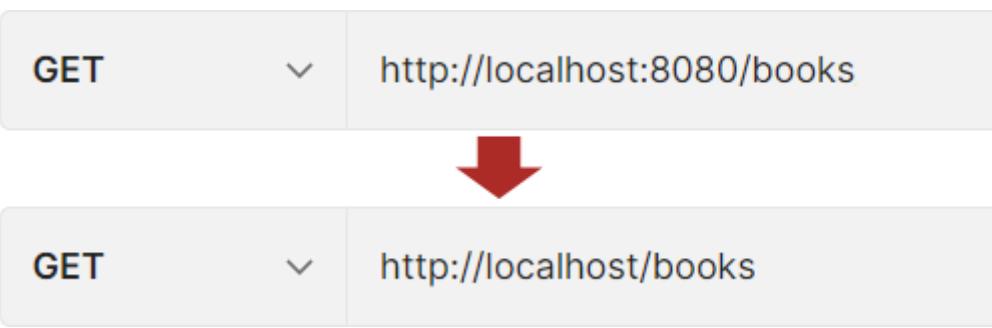
JC-2.SpringBoot基础配置

入门案例做完了，下面就要研究SpringBoot的用法了。通过入门案例，各位小伙伴能够感知到一个信息，SpringBoot没有具体的功能，它是辅助加快Spring程序的开发效率的。我们发现，现在几乎不用做任何配置功能就有了，确实很好用。但是仔细想想，没有做配置意味着什么？意味着配置已经做好了，不用你自己写了。但是新的问题又来了，如果不想用已经写好的默认配置，该如何干预呢？这就是这一章咱们要研究的问题。

如果想修改默认的配置，这个信息应该写在什么位置呢？目前我们接触的入门案例中一共有3个文件，第一是pom.xml文件，设置项目的依赖，这个没什么好研究的，相关的高级内容咱们到原理篇再说，第二是引导类，这个是执行SpringBoot程序的入口，也不像是做功能配置的地方，其实还有一个信息，就是在resources目录下面有一个空白的文件，叫做application.properties。一看就是个配置文件，咱们这一章就来说说配置文件怎么写，能写什么，怎么覆盖SpringBoot的默认配置修改成自己的配置。

JC-2-1.属性配置

SpringBoot通过配置文件application.properties就可以修改默认的配置，那咱们就先找个简单的配置下手，当前访问tomcat的默认端口是8080，好熟悉的味道，但是不便于书写，我们先改成80，通过这个操作来熟悉一下SpringBoot的配置格式是什么样的。



那该如何写呢？properties格式的文件书写规范是key=value

`name=itheima`

这个格式肯定是不能颠覆的，那就尝试性的写就行了，改端口，写port。当你输入port后，神奇的事情就发生了，这玩意儿带提示，太好了。

Property	Type
<code>p server.port=8080 (Server HTTP port)</code>	Integer
<code>p spring.data.cassandra.port=9042 (Port to u...</code>	Integer
<code>p spring.data.mongodb.port (Mongo server por...</code>	Integer
<code>p spring.integration.rsocket.client.port (TC...</code>	Integer
<code>p spring.ldap.embedded.port=0 (Embedded LDAP...</code>	Integer
<code>p spring.mail.port (SMTP server port)</code>	Integer
<code>p spring.rabbitmq.port (RabbitMQ port)</code>	Integer
<code>p spring.redis.port=6379 (Redis server port)</code>	Integer
<code>p spring.rsocket.server.port (Server port)</code>	Integer
<code>p spring.sendgrid.proxy.port (SendGrid proxy...</code>	Integer
<code>p server.tomcat.remoteip.port-header=X-Forwar...</code>	String
<code>p spring.config.import / Tomcat addition</code>	List<String>

根据提示敲回车，输入80端口，搞定。

`server.port=80`

下面就可以直接运行程序，测试效果了。

我们惊奇的发现SpringBoot这玩意儿狠啊，以前修改端口在哪里改？tomcat服务器的配置文件中改，现在呢？SpringBoot专用的配置文件中改，是不是意味着以后所有的配置都可以写在这一个文件中呢？是的，简化开发者配置的书写位置，集中管理。妙啊，妈妈再也不用担心我找不到配置文件了。

其实到这里我们应该得到如下三个信息：

1. SpringBoot程序可以在application.properties文件中进行属性配置
2. application.properties文件中只要输入要配置的属性关键字就可以根据提示进行设置
3. SpringBoot将配置信息集中在一个文件中写，不管是服务器的配置，还是数据库的配置，总之都写在一起，逃离一个项目十几种配置文件格式的尴尬局面

总结

1. SpringBoot默认配置文件是application.properties

做完了端口的配置，趁热打铁，再做几个配置，目前项目启动时会显示一些日志信息，就来改一改这里面的一些设置。

关闭运行日志图表 (banner)

```
spring.main.banner-mode=off
```

设置banner图片

```
spring.banner.image.location=xxx.png
```

设置运行日志的显示级别

```
logging.level.root=debug
```

你会发现，现在这么搞配置太爽了，以前你做配置怎么做？不同的技术有自己专用的配置文件，文件不同格式也不统一，现在呢？不用东奔西走的找配置文件写配置了，统一格式了，这就是大秦帝国啊，统一六国。SpringBoot比大秦狠，因为未来出现的技术还没出现呢，但是现在已经确认了，配置都写这个文件里面。

我们现在配置了3个信息，但是又有新的问题了。这个配置是随便写的吗？什么都能配？有没有一个东西显示所有能配置的项呢？此外这个配置和什么东西有关呢？会不会因为我写了什么东西以后才可以写什么配置呢？比如我现在没有写数据库相关的东西，能否配置数据呢？一个一个来，先说第一个问题，都能配置什么。

打开SpringBoot的官网，找到SpringBoot官方文档，打开查看附录中的Application Properties就可以获取到对应的配置项了，网址奉上：<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#application-properties>

能写什么的问题解决了，再来说第二个问题，这个配置项和什么有关。在pom中注释掉导入的spring-boot-starter-web，然后刷新工程，你会发现配置的提示消失了。闹了半天是设定使用了什么技术才能做什么配置。也合理，不然没有使用对应技术，配了也是白配。

温馨提示

所有的starter中都会依赖下面这个starter，叫做spring-boot-starter。这个starter是所有的SpringBoot的starter的基础依赖，里面定义了SpringBoot相关的基础配置，关于这个starter我们到开发应用篇和原理篇中再深入讲解。

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

总结

1. SpringBoot中导入对应starter后，提供对应配置属性
2. 书写SpringBoot配置采用关键字+提示形式书写

JC-2-2.配置文件分类

现在已经能够进行SpringBoot相关的配置了，但是properties格式的配置写起来总是觉得看着不舒服，所以就期望存在一种书写起来更简便的配置格式提供给开发者使用。有吗？还真有，SpringBoot除了支持properties格式的配置文件，还支持另外两种格式的配置文件。三种配置文件格式分别如下：

- properties格式
- yml格式
- yaml格式

一看到全新的文件格式，各位小伙伴肯定想，这下又要学习新的语法格式了。怎么说呢？从知识角度来说，要学，从开发角度来说，不用学。为什么呢？因为SpringBoot的配置在Idea工具下有提示啊，跟着提示走就行了。下面列举三种不同文件格式配置相同的属性范例，先了解一下。

- application.properties (properties格式)

```
server.port=80
```

- application.yml (yml格式)

```
server:  
  port: 81
```

- application.yaml (yaml格式)

```
server:  
  port: 82
```

仔细看会发现yml格式和yaml格式除了文件名后缀不一样，格式完全一样，是这样的，yml和yaml文件格式就是一模一样的，只是文件后缀不同，所以可以合并成一种格式来看。那对于这三种格式来说，以后用哪一种比较多呢？记清楚，以后基本上都是用yml格式的，本课程后面的所有知识都是基于yml格式来制作的，以后在企业开发过程中用这个格式的机会也最多，一定要重点掌握。

总结

1. SpringBoot提供了3种配置文件的格式

- properties (传统格式/默认格式)
- **yml** (主流格式)
- yaml

思考

现在我们已经知道使用三种格式都可以做配置了，好奇宝宝们就有新的灵魂拷问了，万一我三个都写了，他们三个谁说了算呢？打一架吗？

配置文件优先级

其实三个文件如果共存的话，谁生效说的就是配置文件加载的优先级别。先说一点，虽然以后这种情况很少出现，但是这个知识还是可以学习一下的。我们就让三个配置文件书写同样的信息，比如都配置端口，然后我们让每个文件配置的端口号都不一样，最后启动程序后看启动端口是多少就知道谁的加载优先级比较高了。

- application.properties (properties格式)

```
server.port=80
```

- application.yml (yml格式)

```
server:  
  port: 81
```

- application.yaml (yaml格式)

```
server:  
  port: 82
```

启动后发现目前的启动端口为80，把80对应的文件删除掉，然后再启动，现在端口又改成了81。现在我们就已经知道了3个文件的加载优先顺序是什么。

```
application.properties > application.yml > application.yaml
```

虽然得到了一个知识结论，但是我们实际开发的时候还是要看最终的效果为准。也就是你要的最终效果是什么自己是明确的，上述结论只能帮助你分析结论产生的原因。这个知识了解一下就行了，因为以后同时写多种配置文件格式的情况实在是较少。

最后我们把配置文件内容给修改一下

- application.properties (properties格式)

```
server.port=80  
spring.main.banner-mode=off
```

- application.yml (yml格式)

```
server:  
  port: 81  
logging:  
  level:  
    root: debug
```

- application.yaml (yaml格式)

```
server:  
  port: 82
```

我们发现不仅端口生效了，最终显示80，同时其他两条配置也生效了，看来每个配置文件中的项都会生效，只不过如果多个配置文件中有相同类型的配置会优先级高的文件覆盖优先级的文件中的配置。如果配置项不同的话，所有的配置项都会生效。

总结

1. 配置文件间的加载优先级 properties (最高) > yml > yaml (最低)
2. 不同配置文件中相同配置按照加载优先级相互覆盖，不同配置文件中不同配置全部保留

教你一招：自动提示功能消失解决方案

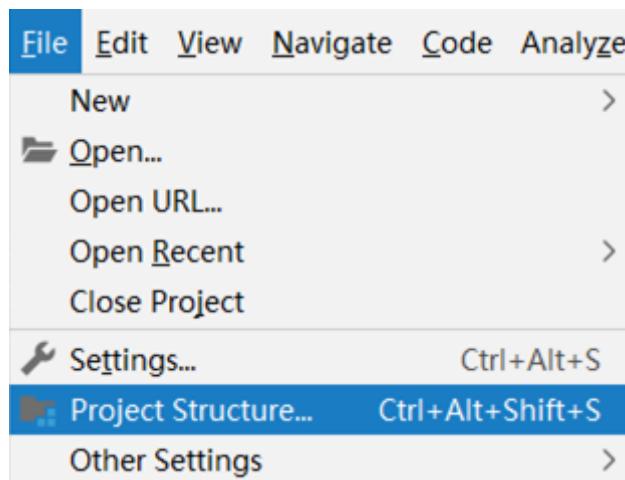
在做程序的过程中，可能有些小伙伴会基于各种各样的原因导致配置文件中没有提示，这个确实很让人头疼，所以下面给大家说一下如果自动提示功能消失了怎么解决。

先要明确一个核心，就是自动提示功能不是SpringBoot技术给我们提供的，是我们在Idea工具下编程，这个编程工具给我们提供的。明白了这一点后，再来说为什么会出现这种现象。其实这个自动提示功能消失的原因还是蛮多的，如果想解决这个问题，就要知道为什么会消失，大体原因有如下2种：

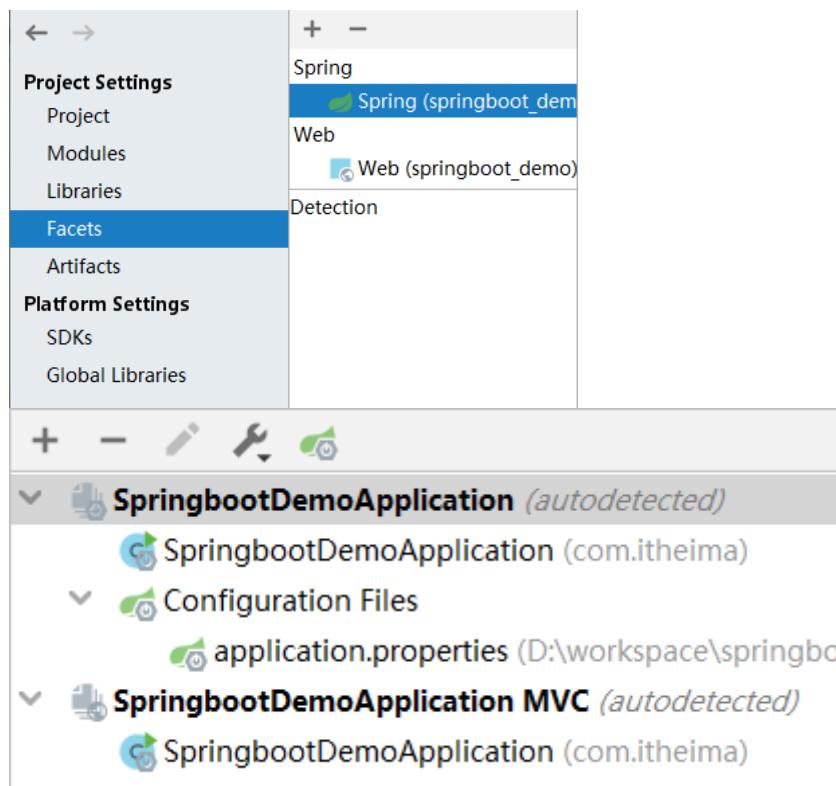
1. Idea认为你现在写配置的文件不是个配置文件，所以拒绝给你提供提示功能
2. Idea认定你是合理的配置文件，但是Idea加载不到对应的提示信息

这里我们主要解决第一个现象，第二种现象到原理篇再讲解。第一种现象的解决方式如下：

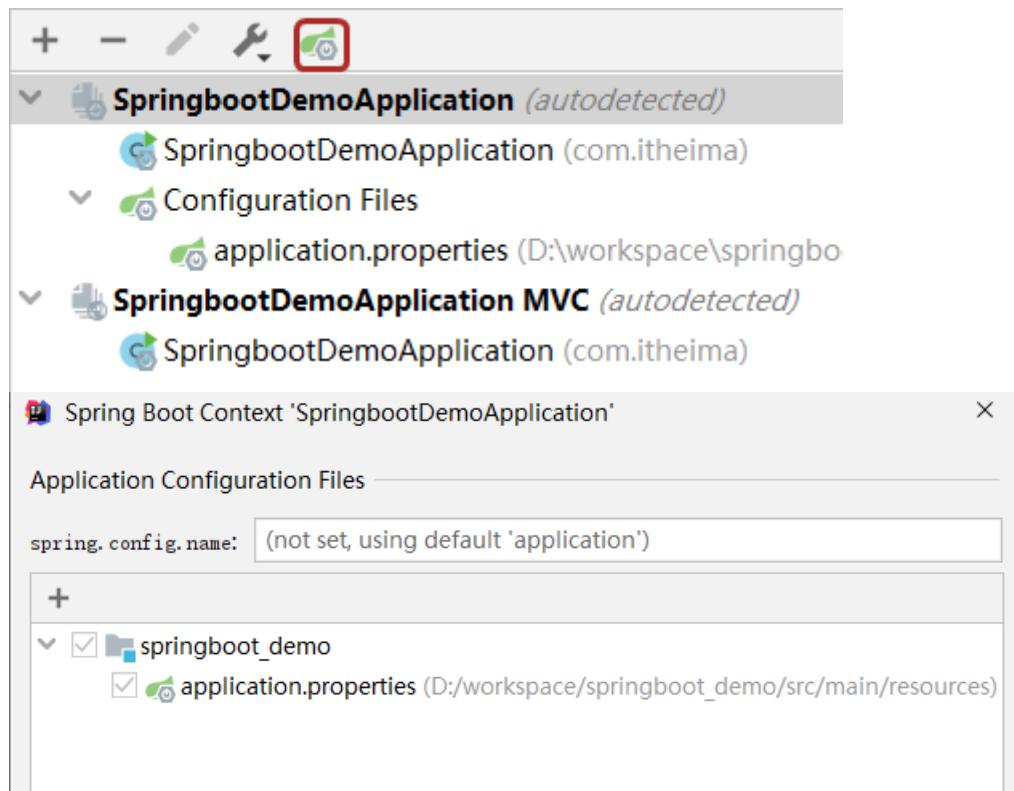
步骤①：打开设置，【Files】→【Project Structure...】



步骤②：在弹出窗口中左侧选择【Facets】，右侧选中Spring路径下对应的模块名称，也就是你自动提示功能消失的那个模块



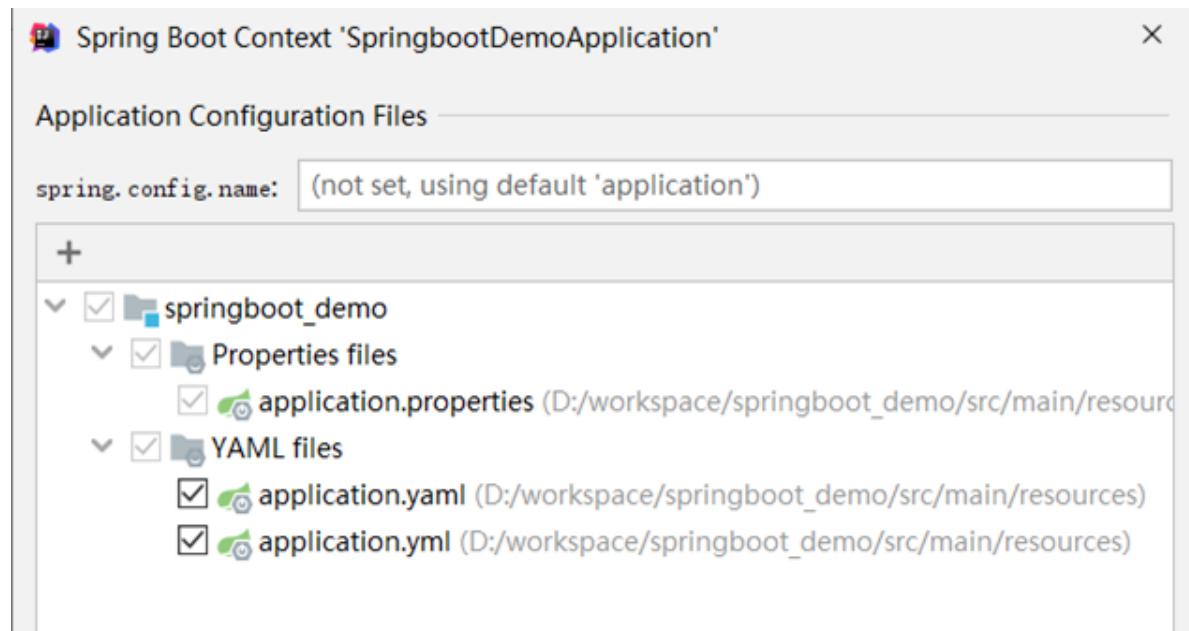
步骤③：点击Customize Spring Boot按钮，此时可以看到当前模块对应的配置文件是哪些了。如果没有你想要称为配置文件的文件格式，就有可能无法弹出提示



步骤④：选择添加配置文件，然后选中要作为配置文件的具体文件就OK了

The screenshot shows the 'Choose Custom Configuration Files' dialog from IntelliJ IDEA. The path 'D:\workspace\springboot_demo\src\main\resources\application.yaml' is shown in the top bar. The main area displays a file tree with several folders like '.mvn', 'src', 'main', 'java', 'resources', 'static', 'templates', and 'test'. Two files are visible in the 'resources' folder: 'application.yaml' and 'application.yml'. The 'application.yaml' file is highlighted with a blue selection bar.

到这里就做完了，其实就是一个小功能



总结

1. 指定SpringBoot配置文件

- Setting → Project Structure → Facets
- 选中对应项目/工程
- Customize Spring Boot
- 选择配置文件

JC-2-3.yaml文件

SpringBoot的配置以后主要使用yaml结尾的这种文件格式，并且在书写时可以通过提示的形式加载正确的格式。但是这种文件还是有严格的书写格式要求的。下面就来说一下具体的语法格式。

YAML (YAML Ain't Markup Language)，一种数据序列化格式。具有容易阅读、容易与脚本语言交互、以数据为核心，重数据轻格式的特点。常见的文件扩展名有两种：

- .yml格式（主流）
- .yaml格式

具体的语法格式要求如下：

1. 大小写敏感
2. 属性层级关系使用多行描述，**每行结尾使用冒号结束**
3. 使用缩进表示层级关系，同层级左侧对齐，只允许使用空格（不允许使用Tab键）
4. 属性值前面添加空格（属性名与属性值之间使用冒号+空格作为分隔）
5. #号 表示注释

上述规则不要死记硬背，按照书写习惯慢慢适应，并且在Idea下由于具有提示功能，慢慢适应着写格式就行了。核心的一条规则要记住，**数据前面要加空格与冒号隔开**。

下面列出常见的数据书写格式，熟悉一下

```

boolean: TRUE          #TRUE, true, True, FALSE, false, False均可
float: 3.14            #6.8523015e+5 #支持科学计数法
int: 123               #0b1010_0111_0100_1010_1110 #支持二进制、八
进制、十六进制
null: ~                #使用~表示null
string: HelloWorld     #字符串可以直接书写
string2: "Hello World" #可以使用双引号包裹特殊字符
date: 2018-02-17       #日期必须使用yyyy-MM-dd格式
datetime: 2018-02-17T15:02:31+08:00 #时间和日期之间使用T连接，最后使用+代表时区

```

此外，yaml格式中也可以表示数组，在属性名书写位置的下方使用减号作为数据开始符号，每行书
写一个数据，减号与数据间空格分隔。

```

subject:
  - Java
  - 前端
  - 大数据
enterprise:
  name: itcast
  age: 16
  subject:
    - Java
    - 前端
    - 大数据
likes: [王者荣耀,刺激战场]      #数组书写缩略格式
users:                           #对象数组格式一
  - 
    name: Tom
    age: 4
  - 
    name: Jerry
    age: 5
users:                           #对象数组格式二
  -
    name: Tom
    age: 4
  -
    name: Jerry
    age: 5
users2: [ { name:Tom , age:4 } , { name:Jerry , age:5 } ] #对象数组缩略格式

```

总结

1. yaml语法规则

- 大小写敏感
- 属性层级关系使用多行描述，每行结尾使用冒号结束
- 使用缩进表示层级关系，同层级左侧对齐，只允许使用空格（不允许使用Tab键）
- 属性值前面添加空格（属性名与属性值之间使用冒号+空格作为分隔）
- #号 表示注释

2. 注意属性名冒号后面与数据之间有一个空格

3. 字面值、对象数据格式、数组数据格式

思考

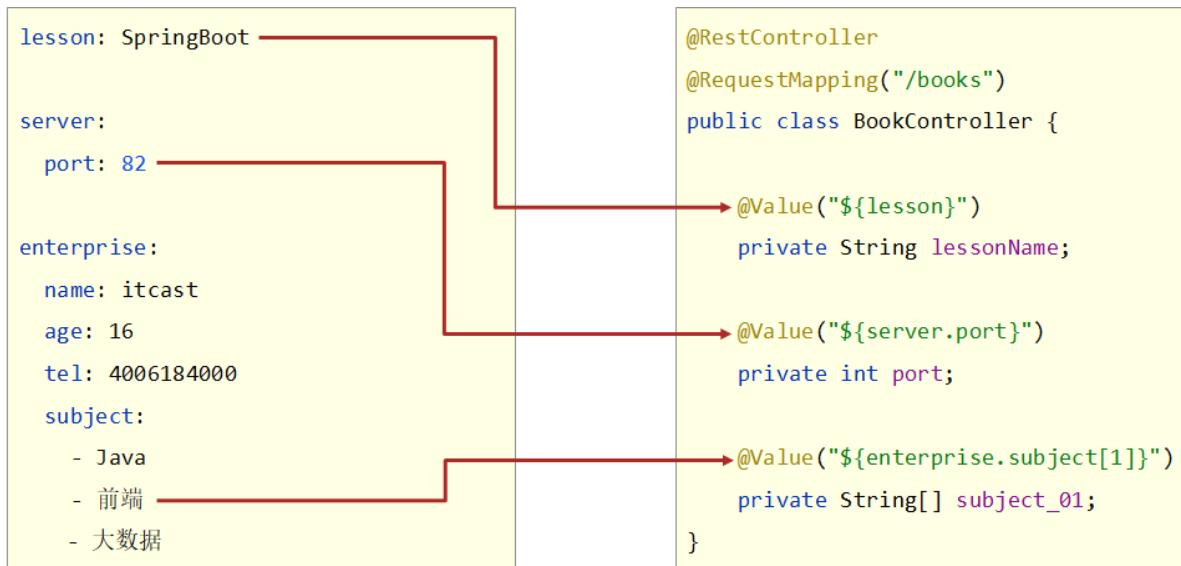
现在我们已经知道了yaml具有严格的数据格式要求，并且已经可以正确的书写yaml文件了，那这些文件书写后其实是在定义一些数据。这些数据是给谁用的呢？大部分是SpringBoot框架内部使用，但是如果我们要配置一些数据自己使用，能不能用呢？答案是可以的，那如何读取yaml文件中的数据呢？咱们下一节再说。

JC-2-4.yaml数据读取

对于yaml文件中的数据，其实你就可以想象成这就是一个小型的数据库，里面保存有若干数据，每个数据都有一个独立的名字，如果你想读取里面的数据，肯定是支持的，下面就介绍3种读取数据的方式。

读取单一数据

yaml中保存的单个数据，可以使用Spring中的注解@Value读取单个数据，属性名引用方式：**\$一级属性名.二级属性名.....}**



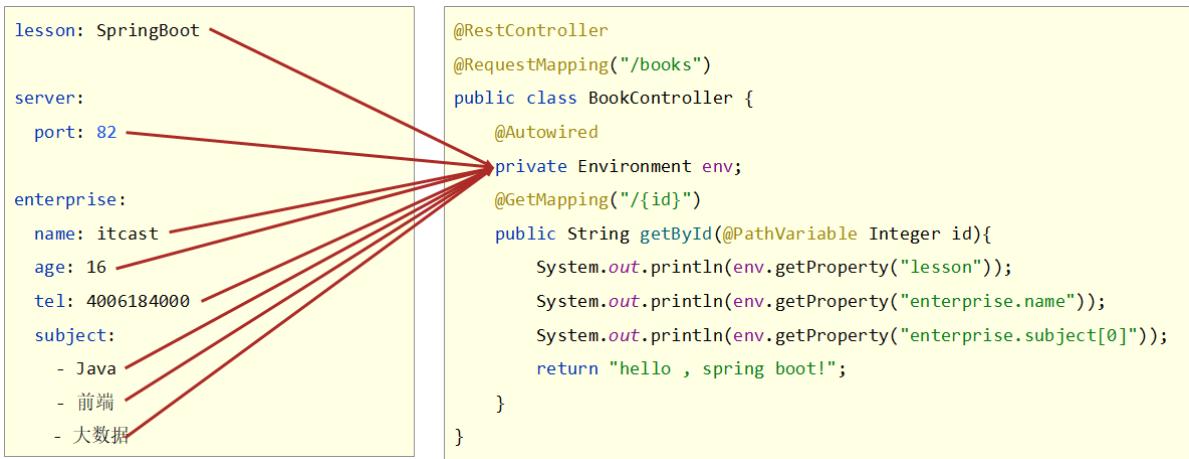
记得使用@Value注解时，要将该注解写在某一个指定的Spring管控的bean的属性名上方，这样当bean进行初始化时候就可以读取到对应的单一数据了。

总结

1. 使用@Value配合SpEL读取单个数据
2. 如果数据存在多层级，依次书写层级名称即可

读取全部数据

读取单一数据可以解决读取数据的问题，但是如果定义的数据量过大，这么一个一个书写肯定会累死人的，SpringBoot提供了一个对象，能够把所有的数据都封装到这一个对象中，这个对象叫做**Environment**（这个对象是spring提供给我们的，但是必须加上@Autowired注解在上面），使用自动装配注解可以将所有的yaml数据封装到这个对象中



数据封装到了Environment对象中，获取属性时，通过Environment的接口操作进行，具体方法是getProperties (String)，参数填写属性名即可

总结

1. 使用Environment对象封装全部配置信息
2. 使用@Autowired自动装配数据到Environment对象中

读取对象数据

单一数据读取书写比较繁琐，全数据读取封装的太厉害了，每次拿数据还要一个一个的getProperties ()，总之用起来都不是很舒服。由于Java是一个面向对象的语言，很多情况下，我们会将一组数据封装成一个对象。SpringBoot也提供了可以将一组yaml对象数据封装一个Java对象的操作

首先定义一个对象，并将该对象纳入Spring管控的范围，也就是定义成一个bean，然后使用注解@ConfigurationProperties指定该对象加载哪一组yaml中配置的信息。



这个@ConfigurationProperties必须告诉他加载的数据前缀是什么，这样指定前缀下的所有属性就封装到这个对象中。记得数据属性名要与对象的变量名一一对应啊，不然没法封装。其实以后如果你要定义一组数据自己使用，就可以先写一个对象，然后定义好属性，下面到配置中根据这个格式书写即可。

温馨提示

细心的小伙伴会发现一个问题，自定义的这种数据在yaml文件中书写时没有弹出提示，咱们到原理篇再揭秘如何弹出提示。

总结

1. 使用@ConfigurationProperties注解绑定配置信息到封装类中
2. 封装类需要定义为Spring管理的bean，否则无法进行属性注入

yaml文件中的数据引用

如果你在书写yaml数据时，经常出现如下现象，比如很多个文件都具有相同的目录前缀

```
center:  
  dataDir: /usr/local/fire/data  
  tmpDir: /usr/local/fire/tmp  
  logDir: /usr/local/fire/log  
  msgDir: /usr/local/fire/msgDir
```

或者

```
center:  
  dataDir: D:/usr/local/fire/data  
  tmpDir: D:/usr/local/fire/tmp  
  logDir: D:/usr/local/fire/log  
  msgDir: D:/usr/local/fire/msgDir
```

这个时候你可以使用引用格式来定义数据，其实就是搞了个变量名，然后引用变量了，格式如下：

```
baseDir: /usr/local/fire  
center:  
  dataDir: ${baseDir}/data  
  tmpDir: ${baseDir}/tmp  
  logDir: ${baseDir}/log  
  msgDir: ${baseDir}/msgDir
```

还有一个注意事项，在书写字符串时，如果需要使用转义字符，需要将数据字符串使用双引号包裹起来

```
lesson: "Spring\tboot\nlesson" #注意这个的双引号
```

总结

1. 在配置文件中可以使用\${属性名}方式引用属性值
2. 如果属性中出现特殊字符，可以使用双引号包裹起来作为字符解析

到这里有关yaml文件的基础使用就先告一段落，实用篇中再继续研究更深入的内容。

JC-3.基于SpringBoot实现SSMP整合

重头戏来了，SpringBoot之所以好用，就是它能方便快捷的整合其他技术，这一部分咱们就来聊聊一些技术的整合方式，通过这一章的学习，大家能够感受到SpringBoot到底有多酷炫。这一章咱们学习如下技术的整合方式

- 整合JUnit
- 整合MyBatis

- 整合MyBatis-Plus
- 整合Druid

上面这些技术都整合完毕后，我们做一个小案例，也算是学有所用吧。涉及的技术比较多，综合运用一下。

JC-3-1.整合JUnit

SpringBoot技术的定位用于简化开发，再具体点是简化Spring程序的开发。所以在整合任意技术的时候，如果你想直观感触到简化的效果，你必须先知道使用非SpringBoot技术时对应的整合是如何做的，然后再看基于SpringBoot的整合是如何做的，才能比对出来简化在了哪里。

我们先来看一下不使用SpringBoot技术时，Spring整合JUnit的制作方式

```
//加载spring整合junit专用的类运行器
@RunWith(SpringJUnit4ClassRunner.class)
//指定对应的配置信息
@ContextConfiguration(classes = SpringConfig.class)
public class AccountServiceTestCase {
    //注入你要测试的对象
    @Autowired
    private AccountService accountService;
    @Test
    public void test GetById() {
        //执行要测试的对象对应的方法
        System.out.println(accountService.findById(2));
    }
}
```

其中核心代码是前两个注解，第一个注解@RunWith是设置Spring专用的测试类运行器，简单说就是Spring程序执行程序有自己的一套独立的运行程序的方式，不能使用JUnit提供的类运行方式了，必须指定一下，但是格式是固定的，琢磨一下，**每次都指定一样的东西，这个东西写起来没有技术含量啊**，第二个注解@ContextConfiguration是用来设置Spring核心配置文件或配置类的，简单说就是加载Spring的环境你要告诉Spring具体的环境配置是在哪里写的，虽然每次加载的文件都有可能不同，但是仔细想想，如果文件名是固定的，这个貌似也是一个固定格式。既然**有可能是固定格式，那就有可能每次都写一样的东西，也是一个没有技术含量的内容书写**

SpringBoot就抓住上述两条没有技术含量的内容书写进行开发简化，能走默认值的走默认值，能不写的就不写，具体格式如下

```

@SpringBootTest
class Springboot04JunitApplicationTests {
    //注入你要测试的对象
    @Autowired
    private BookDao bookDao;
    @Test
    void contextLoads() {
        //执行要测试的对象对应的方法
        bookDao.save();
        System.out.println("two...");
    }
}

```

看看这次简化成什么样了，一个注解就搞定了，而且还没有参数，再体会SpringBoot整合其他技术的优势在哪里，就两个字——**简化**。使用一个注解@SpringBootTest替换了前面两个注解。至于内部是怎么回事？和之前一样，只不过都走默认值。

这个时候有人就问了，你加载的配置类或者配置文件是哪一个？就是我们前面启动程序使用的引导类。如果想手工指定引导类有两种方式，第一种方式使用属性的形式进行，在注解@SpringBootTest中添加classes属性指定配置类

```

@SpringBootTest(classes = Springboot04JunitApplication.class)
class Springboot04JunitApplicationTests {
    //注入你要测试的对象
    @Autowired
    private BookDao bookDao;
    @Test
    void contextLoads() {
        //执行要测试的对象对应的方法
        bookDao.save();
        System.out.println("two...");
    }
}

```

第二种方式回归原始配置方式，仍然使用@ContextConfiguration注解进行，效果是一样的

```

@SpringBootTest
@ContextConfiguration(classes = Springboot04JunitApplication.class)
class Springboot04JunitApplicationTests {
    //注入你要测试的对象
    @Autowired
    private BookDao bookDao;
    @Test
    void contextLoads() {
        //执行要测试的对象对应的方法
        bookDao.save();
        System.out.println("two...");
    }
}

```

温馨提示

使用SpringBoot整合JUnit需要保障导入test对应的starter，由于初始化项目时此项是默认导入的，所以此处没有提及，其实和之前学习的内容一样，用什么技术导入对应的starter即可。

总结

1. 导入测试对应的starter
2. 测试类使用@SpringBootTest修饰
3. 使用自动装配的形式添加要测试的对象
4. 测试类如果存在于引导类所在包或子包中无需指定引导类
5. 测试类如果不存在于引导类所在的包或子包中需要通过classes属性指定引导类

JC-3-2.整合MyBatis

整合完JUnit下面再来说一下整合MyBatis，这个技术是大部分公司都要使用的技术，务必掌握。如果对Spring整合MyBatis不熟悉的小伙伴好好复习一下，下面列举出原始整合的全部内容，以配置类的形式为例进行

- 导入坐标，MyBatis坐标不能少，Spring整合MyBatis还有自己专用的坐标，此外Spring进行数据库操作的jdbc坐标是必须的，剩下还有mysql驱动坐标，本例中使用了Druid数据源，这个倒是可以不要

```
<dependencies>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.16</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.6</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <!--1.导入mybatis与spring整合的jar包-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>1.3.0</version>
    </dependency>
    <!--导入spring操作数据库必选的包-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.10.RELEASE</version>
    </dependency>
</dependencies>
```

- Spring核心配置

```
@Configuration  
@ComponentScan("com.itheima")  
@PropertySource("jdbc.properties")  
public class SpringConfig {  
}
```

- MyBatis要交给Spring接管的bean

```
//定义mybatis专用的配置类  
@Configuration  
public class MyBatisConfig {  
    // 定义创建SqlSessionFactory对应的bean  
    @Bean  
    public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){  
        //SqlSessionFactoryBean是由mybatis-spring包提供的，专用于整合用的对象  
        SqlSessionFactoryBean sfb = new SqlSessionFactoryBean();  
        //设置数据源替代原始配置中的environments的配置  
        sfb.setDataSource(dataSource);  
        //设置类型别名替代原始配置中的typeAliases的配置  
        sfb.setTypeAliasesPackage("com.itheima.domain");  
        return sfb;  
    }  
    // 定义加载所有的映射配置  
    @Bean  
    public MapperScannerConfigurer mapperScannerConfigurer(){  
        MapperScannerConfigurer msc = new MapperScannerConfigurer();  
        msc.setBasePackage("com.itheima.dao");  
        return msc;  
    }  
}
```

- 数据源对应的bean，此处使用Druid数据源

```
@Configuration  
public class JdbcConfig {  
    @Value("${jdbc.driver}")  
    private String driver;  
    @Value("${jdbc.url}")  
    private String url;  
    @Value("${jdbc.username}")  
    private String userName;  
    @Value("${jdbc.password}")  
    private String password;  
  
    @Bean("dataSource")  
    public DataSource dataSource(){  
        DruidDataSource ds = new DruidDataSource();  
        ds.setDriverClassName(driver);  
        ds.setUrl(url);  
        ds.setUsername(userName);  
        ds.setPassword(password);  
        return ds;  
    }  
}
```

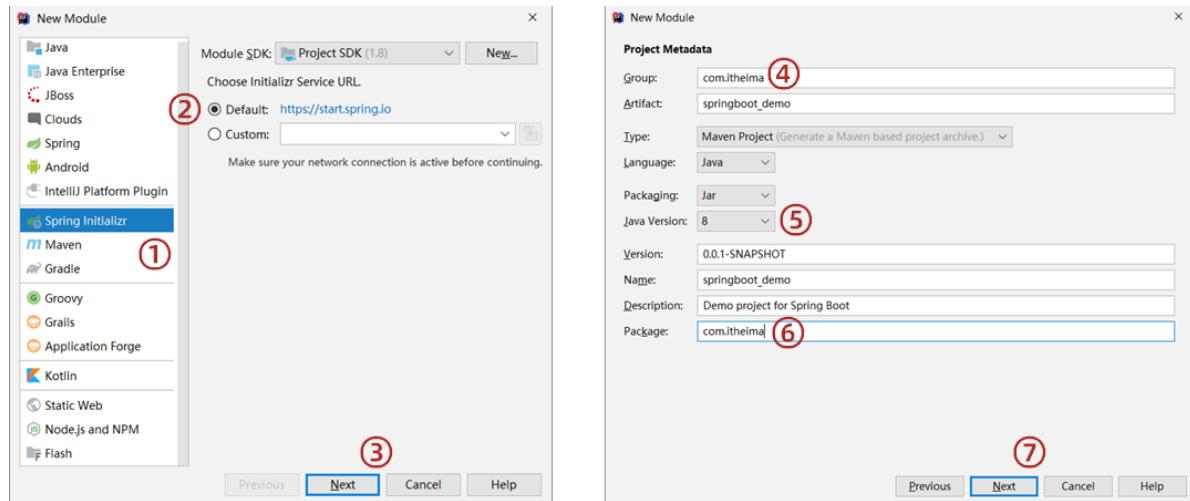
```
}
```

- 数据库连接信息 (properties格式)

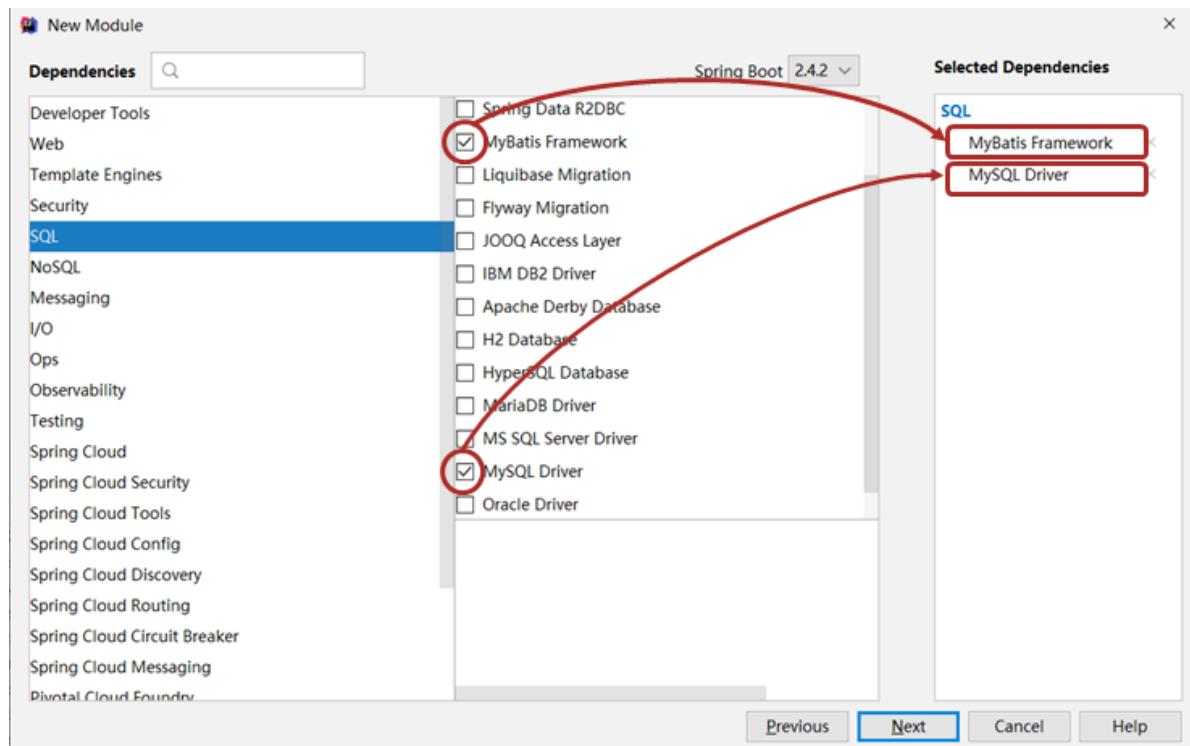
```
jdbc.driver=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/spring_db?useSSL=false  
jdbc.username=root  
jdbc.password=root
```

上述格式基本上是最简格式了，要写的东西还真不少。下面看看SpringBoot整合MyBatis格式

步骤①：创建模块



步骤②：勾选要使用的技术，MyBatis，由于要操作数据库，还要勾选对应数据库



或者手工导入对应技术的starter，和对应数据库的坐标

```
<dependencies>  
    <!--1. 导入对应的starter-->  
    <dependency>  
        <groupId>org.mybatis.spring.boot</groupId>
```

```
<artifactId>mybatis-spring-boot-starter</artifactId>
<version>2.2.0</version>
</dependency>

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

步骤③：配置数据源相关信息，没有这个信息你连接哪个数据库都不知道

```
#2. 配置相关信息
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db
    username: root
    password: root
```

结束了，就这么多了。有人就很纳闷，这就结束了？对，这就结束了，SpringBoot把配置中所有可能出现的通用配置都简化了。下面写一个MyBatis程序运行需要的Dao（或者Mapper）就可以运行了

实体类

```
public class Book {
    private Integer id;
    private String type;
    private String name;
    private String description;
}
```

映射接口（Dao）

```
@Mapper
public interface BookDao {
    @Select("select * from tbl_book where id = #{id}")
    public Book getById(Integer id);
}
```

测试类

```
@SpringBootTest
class Springboot05MybatisApplicationTests {
    @Autowired
    private BookDao bookDao;
    @Test
    void contextLoads() {
        System.out.println(bookDao.getById(1));
    }
}
```

完美，开发从此变的就这么简单。再体会一下SpringBoot如何进行第三方技术整合的，是不是很优秀？具体内部的原理到原理篇再展开讲解

注意：当前使用的SpringBoot版本是2.5.4，对应的坐标设置中Mysql驱动使用的是8x版本。使用SpringBoot2.4.3（不含）之前版本会出现一个小BUG，就是MySQL驱动升级到8以后要求强制配置时区，如果不设置会出问题。解决方案很简单，驱动url上面添加上对应设置就行了

#2. 配置相关信息

```
spring:  
  datasource:  
    driver-class-name: com.mysql.cj.jdbc.Driver  
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC  
    username: root  
    password: root
```

这里设置的UTC是全球标准时间，你也可以理解为是英国时间，中国处在东八区，需要在这个基础上加上8小时，这样才能和中国地区的时间对应的，也可以修改配置为Asia/Shanghai，**同样**可以解决这个问题。

#2. 配置相关信息

```
spring:  
  datasource:  
    driver-class-name: com.mysql.cj.jdbc.Driver  
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=Asia/Shanghai  
    username: root  
    password: root
```

如果不想每次都设置这个东西，也可以去修改mysql中的配置文件mysql.ini，在mysqld项中添加default-time-zone=+8:00也可以解决这个问题。其实方式方法很多，这里就说这么多吧。

此外在运行程序时还会给出一个提示，说数据库驱动过时的警告，根据提示修改配置即可，弃用**com.mysql.jdbc.Driver**，换用**com.mysql.cj.jdbc.Driver**。前面的例子中已经更换了驱动了，在此说明一下。

```
Loading class `com.mysql.jdbc.Driver'. This is deprecated. The new driver class  
is `com.mysql.cj.jdbc.Driver'. The driver is automatically registered via the SPI  
and manual loading of the driver class is generally unnecessary.
```

总结

1. 整合操作需要勾选MyBatis技术，也就是导入MyBatis对应的starter
2. 数据库连接相关信息转换成配置
3. 数据库SQL映射需要添加@Mapper被容器识别到
4. MySQL 8.X驱动强制要求设置时区
 - 修改url，添加serverTimezone设定
 - 修改MySQL数据库配置
5. 驱动类过时，提醒更换为com.mysql.cj.jdbc.Driver

JC-3-3.整合MyBatis-Plus

做完了两种技术的整合了，各位小伙伴要学会总结，我们做这个整合究竟哪些是核心？总结下来就两句话

- 导入对应技术的starter坐标
- 根据对应技术的要求做配置

虽然看起来有点虚，但是确实是这个理儿，下面趁热打铁，再换一个技术，看看是不是上面这两步。

接下来在MyBatis的基础上再升级一下，整合MyBatisPlus（简称MP），国人开发的技术，符合中国人开发习惯，谁用谁知道。来吧，一起做整合

步骤①：导入对应的starter

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.3</version>
</dependency>
```

关于这个坐标，此处要说明一点，之前我们看的starter都是spring-boot-starter-? ? ?，也就是说都是下面的格式

```
Spring-boot-start-***
```

而MyBatis与MyBatisPlus这两个坐标的名字书写比较特殊，是第三方技术名称在前，boot和starter在后。此处简单提一下命名规范，后期原理篇会再详细讲解

starter 所属	命名规则	示例
官方提供	spring-boot-starter-技术名称	spring-boot-starter-web spring-boot-starter-test
第三方提供	第三方技术名称-spring-boot-starter	mybatis-spring-boot-starter druid-spring-boot-starter
第三方提供	第三方技术名称-boot-starter（第三方技术名称过长，简化命名）	mybatis-plus-boot-starter

温馨提示

有些小伙伴在创建项目时想通过勾选的形式找到这个名字，别翻了，没有。截止目前，SpringBoot官网还未收录此坐标，而我们Idea创建模块时读取的是SpringBoot官网的Spring Initializr，所以也没有。如果换用阿里云的url创建项目可以找到对应的坐标。

步骤②：配置数据源相关信息

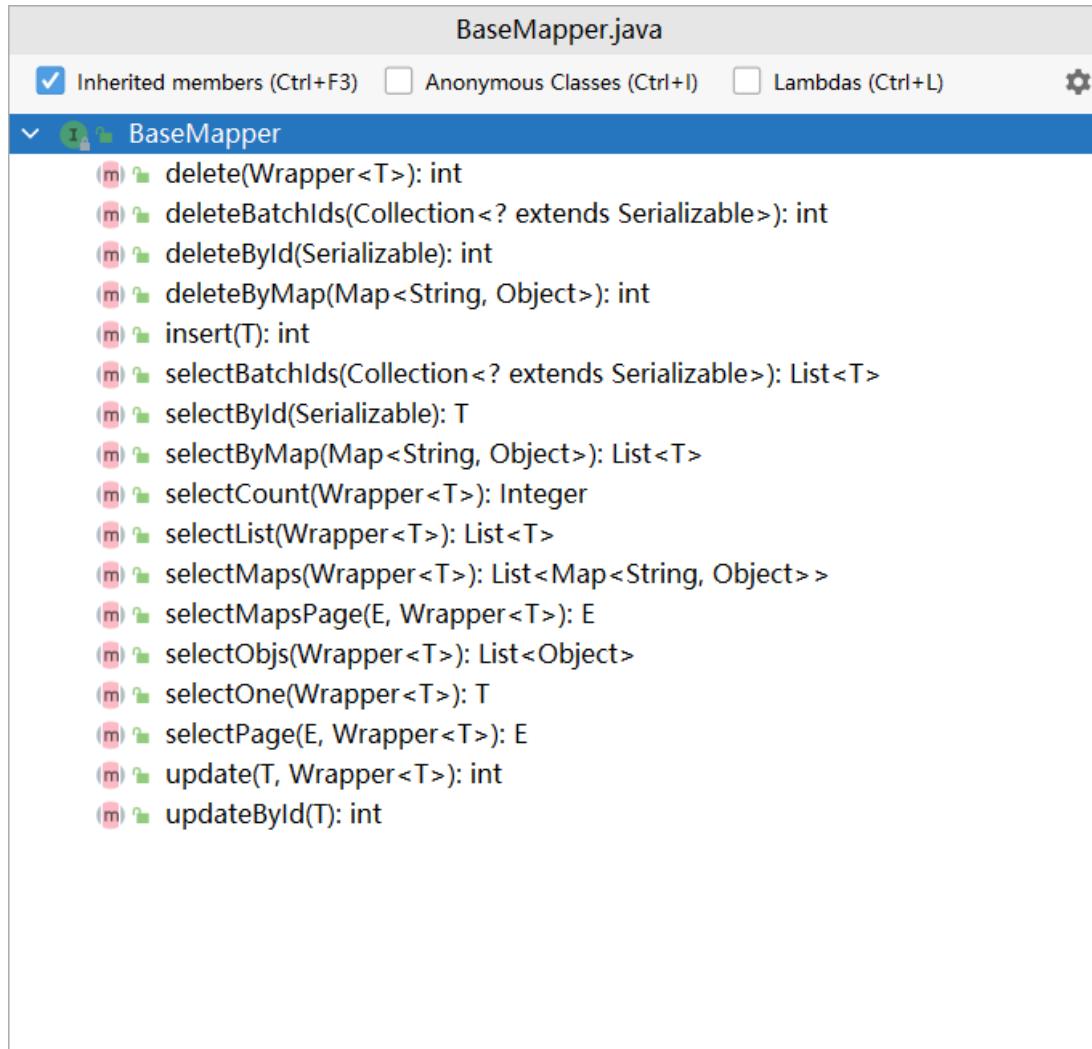
```
#2.配置相关信息
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db
    username: root
    password: root
```

没了，就这么多了，剩下的就是写MyBaitsPlus的程序了

映射接口 (Dao)

```
@Mapper
public interface BookDao extends BaseMapper<Book> {
}
```

核心在于Dao接口继承了一个BaseMapper的接口，这个接口中帮助开发者预定了若干个常用的API接口，简化了通用API接口的开发工作。



下面就可以写一个测试类进行测试了，此处省略。

温馨提示

默认数据库名和实体类名是一致的(因为mp不需要我们自己写sql了，之前的mybatis是我们自己写的sql)

但是就比如你数据库表为tbl_book，但是你的实体类名脚book，所以你就需要加一个前缀，下面就是加前缀的过程

目前数据库的表名定义规则是tbl_模块名称，为了能和实体类相对应，需要做一个配置，相关知识各位小伙伴可以到MyBatisPlus课程中去学习，此处仅给出解决方案。配置application.yml文件，添加如下配置即可，设置所有表名的通用前缀名

```
mybatis-plus:  
  global-config:  
    db-config:  
      table-prefix: tb1_ #设置所有表的通用前缀名称为tb1_
```

总结

1. 手工添加MyBatis-Plus对应的starter
2. 数据层接口使用BaseMapper简化开发
3. 需要使用的第三方技术无法通过勾选确定时，需要手工添加坐标

JC-3-4.整合Druid

使用SpringBoot整合了3个技术了，发现套路基本相同，导入对应的starter，然后做配置，各位小伙伴需要一直强化这套思想。下面再整合一个技术，继续深入强化此思想。

前面整合MyBatis和MyBatisPlus的时候，使用的数据源对象都是SpringBoot默认的数据源对象，下面我们手工控制一下，自己指定了一个数据源对象，Druid。

在没有指定数据源时，我们的配置如下：

```
#2.配置相关信息  
spring:  
  datasource:  
    driver-class-name: com.mysql.cj.jdbc.Driver  
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=Asia/Shanghai  
    username: root  
    password: root
```

此时虽然没有指定数据源，但是根据SpringBoot的德行，肯定帮我们选了一个它认为最好的数据源对象，这就是HiKari。通过启动日志可以查看到对应的的身影。

```
2021-11-29 09:39:15.202 INFO 12260 --- [           main]  
com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Starting...  
2021-11-29 09:39:15.208 WARN 12260 --- [           main]  
com.zaxxer.hikari.util.DriverDataSource : Registered driver with  
driverClassName=com.mysql.jdbc.Driver was not found, trying direct  
instantiation.  
2021-11-29 09:39:15.551 INFO 12260 --- [           main]  
com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Start completed.
```

上述信息中每一行都有HiKari的身影，如果需要更换数据源，其实只需要两步即可。

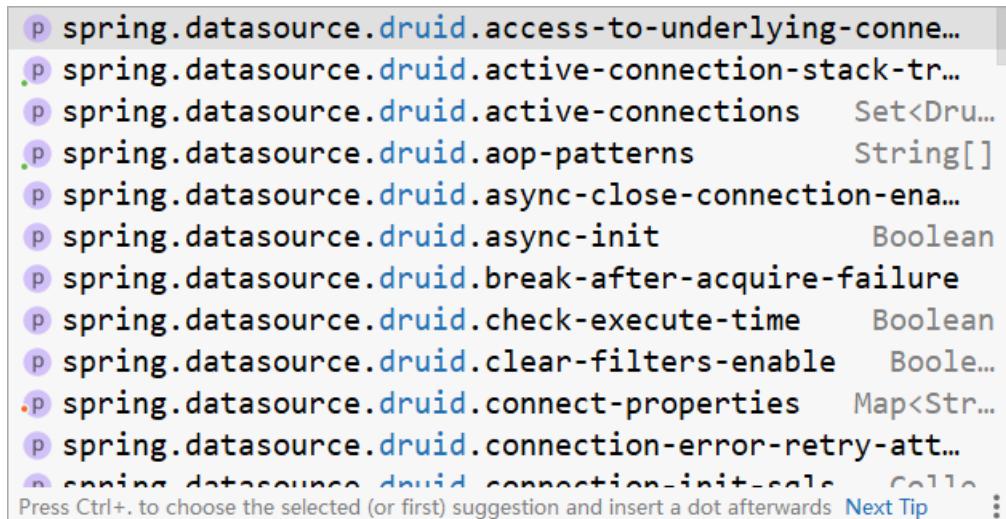
步骤①：导入对应的starter

```
<dependencies>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.2.6</version>
    </dependency>
</dependencies>
```

步骤②：修改配置

```
spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
      username: root
      password: root
```

注意观察，配置项中，在datasource下面并不是直接配置url这些属性的，而是先配置了一个druid节点，然后再配置的url这些东西。言外之意，url这个属性是druid下面的属性，那你能想到什么？除了这4个常规配置外，还有druid专用的其他配置。通过提示功能可以打开druid相关的配置查阅



与druid相关的配置超过200条以上，这就告诉你，如果想做druid相关的配置，使用这种格式就可以了，这里就不展开描述了，太多了。

这是我们做的第4个技术的整合方案，还是那两句话：**导入对应starter，使用对应配置**。没了，SpringBoot整合其他技术就这么简单粗暴。

总结

1. 整合Druid需要导入Druid对应的starter
2. 根据Druid提供的配置方式进行配置
3. 整合第三方技术通用方式
 - 导入对应的starter
 - 根据提供的配置格式，配置非默认值对应的配置项

JC-3-5.SSMP整合综合案例

SpringBoot能够整合的技术太多太多了，对于初学者来说慢慢来，一点点掌握。前面咱们做了4个整合了，下面就通过一个稍微综合一点的案例，将所有知识贯穿起来，同时做一个小功能，体会一下。不过有言在先，这个案例制作的时候，你可能会有这种感觉，说好的SpringBoot整合其他技术的案例，为什么感觉SpringBoot整合其他技术的身影不多呢？因为这东西书写太简单了，简单到瞬间写完，大量的时间做的不是这些整合工作。

先看一下这个案例的最终效果

主页面

The screenshot shows a table of books with the following data:

序号	图书类别	图书名称	描述	操作
1	计算机理论	Spring实战 第3版	Spring入门经典教程,深入理解Spring原理技术内幕	[编辑] [删除]
2	计算机理论	Spring 5核心技术与30个亲手实践	十年沉淀之作,手写Spring精华思想	[编辑] [删除]
3	计算机理论	Spring 5 设计模式	深入Spring源码剖析Spring源码中蕴含的10大设计模式	[编辑] [删除]
4	计算机理论	Spring MVC+MyBatis开发从入门到项目实战	全方位解析面向Web应用的经典架构,带你成为Spring MVC开发高手	[编辑] [删除]
5	计算机理论	轻量级Java Web企业应用实战	源码级剖析Spring框架,适合已掌握Java基础的读者	[编辑] [删除]
6	计算机理论	Java核心技术 番茄知识 (原书第11版)	Core Java 第11版, Jolt大奖获奖作品,针对Java 5.0、10、11全面更新	[编辑] [删除]
7	计算机理论	深入理解Java虚拟机	5个维度全面剖析JVM, 大厂面试知识点全覆盖	[编辑] [删除]
8	计算机理论	Java编程思想 (第4版)	Java学习必读经典,课堂必备书!赢得了全球程序员的广泛赞誉	[编辑] [删除]
9	计算机理论	零基础学Java (全彩版)	零基础自学编程的入门图书,由浅入深,讲解Java语言的编程思想和核心技术	[编辑] [删除]
10	市场营销	直接抵达这么快:主播高效沟通实战指南	李子柒、李佳琦、薇娅成长为网红的秘密都在书中	[编辑] [删除]

共 15 条 < 1 2 > 前往 1 页

添加

The screenshot shows a modal dialog titled "新增图书" (Add Book) with the following fields:

* 图书类别	* 图书名称
描述	

Buttons: 取消 (Cancel), 确定 (Confirm)

The background table shows the same book list as the previous screenshot.

共 15 条 < 1 2 > 前往 1 页

删除

The screenshot shows a confirmation dialog titled "提示" (Prompt) with the message: "此操作永久删除当前信息, 是否继续?" (This operation will permanently delete the current information, do you want to continue?). Buttons: 取消 (Cancel), 确定 (Confirm).

The background table shows the same book list as the previous screenshots.

共 15 条 < 1 2 > 前往 1 页

修改

The screenshot shows a book management interface with a modal dialog titled "编辑检查项" (Edit Check Item). The dialog contains fields for "图书类别" (Book Category) set to "计算机理论", "图书名称" (Book Name) set to "Spring实战 第5版", and "描述" (Description) set to "Spring入门经典教程, 深入理解Spring原理技术内幕". There are "取消" (Cancel) and "确定" (Confirm) buttons at the bottom. The background table lists books with columns: 序号 (Index), 图书类别 (Category), 图书名称 (Name), 描述 (Description), and 操作 (Operations). The table has 15 rows.

分页

The screenshot shows a paginated list of books. The table has columns: 序号 (Index), 图书类别 (Category), 图书名称 (Name), 描述 (Description), and 操作 (Operations). The table has 15 rows. The page navigation at the bottom shows "共 15 条 < 1 2 > 前往 1 页".

条件查询

The screenshot shows a search results page with a filter bar at the top containing "图书类别" (Category) and "Spring". The table has columns: 序号 (Index), 图书类别 (Category), 图书名称 (Name), 描述 (Description), and 操作 (Operations). The table has 4 rows. The page navigation at the bottom shows "共 4 条 < 1 > 前往 1 页".

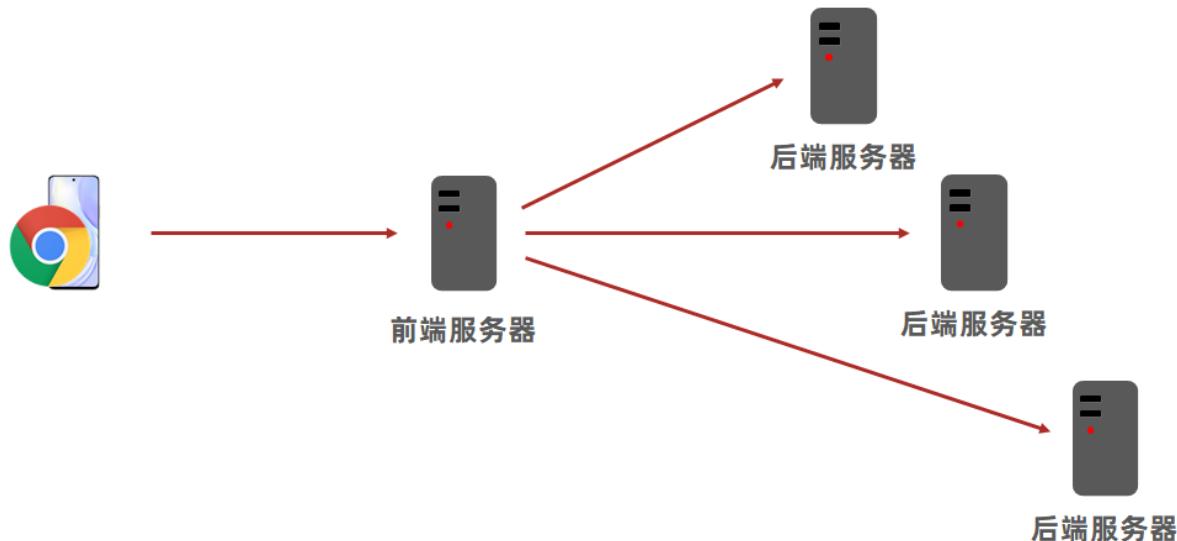
整体案例中需要采用的技术如下，先了解一下，做到哪一个说哪一个

1. 实体类开发——使用Lombok快速制作实体类
2. Dao开发——整合MyBatisPlus，制作数据层测试
3. Service开发——基于MyBatisPlus进行增量开发，制作业务层测试类
4. Controller开发——基于Restful开发，使用PostMan测试接口功能
5. Controller开发——前后端开发协议制作
6. 页面开发——基于VUE+ElementUI制作，前后端联调，页面数据处理，页面消息处理
 - 列表
 - 新增
 - 修改
 - 删除
 - 分页
 - 查询
7. 项目异常处理
8. 按条件查询——页面功能调整、Controller修正功能、Service修正功能

可以看的出来，东西还是很多的，希望通过这个案例，各位小伙伴能够完成基础开发的技能训练。整体开发过程采用做一层测一层的形式进行，过程完整，战线较长，希望各位能跟紧进度，完成这个小案例的制作。

0.模块创建

对于这个案例如果按照企业开发的形式进行应该制作后台微服务，前后端分离的开发。



我知道这个对初学的小伙伴要求太高了，咱们简化一下。后台做单体服务器，前端不使用前后端分离的制作了。



一个服务器即充当后台服务调用，又负责前端页面展示，降低学习的门槛。

下面我们创建一个新的模块，加载要使用的技术对应的starter，修改配置文件格式为yml格式，并把web访问端口先设置成80。

pom.xml

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

application.yml

```
server:
  port: 80
```

1.实体类开发

本案例对应的模块表结构如下：

```
-- -----
-- Table structure for `tbl_book`
-- -----
DROP TABLE IF EXISTS `tbl_book`;
CREATE TABLE `tbl_book` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `type` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `name` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `description` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 51 CHARACTER SET = utf8 COLLATE =
utf8_general_ci ROW_FORMAT = Dynamic;

-- -----
-- Records of `tbl_book`
-- -----
INSERT INTO `tbl_book` VALUES (1, '计算机理论', 'Spring实战 第5版', 'Spring入门经典教程, 深入理解Spring原理技术内幕');
INSERT INTO `tbl_book` VALUES (2, '计算机理论', 'Spring 5核心原理与30个类手写实战', '十年沉淀之作, 手写Spring精华思想');
INSERT INTO `tbl_book` VALUES (3, '计算机理论', 'Spring 5 设计模式', '深入Spring源码剖析Spring源码中蕴含的10大设计模式');
INSERT INTO `tbl_book` VALUES (4, '计算机理论', 'Spring MVC+MyBatis开发从入门到项目实战', '全方位解析面向web应用的轻量级框架, 带你成为Spring MVC开发高手');
INSERT INTO `tbl_book` VALUES (5, '计算机理论', '轻量级Java web企业应用实战', '源码级剖析Spring框架, 适合已掌握Java基础的读者');
INSERT INTO `tbl_book` VALUES (6, '计算机理论', 'Java核心技术 卷I 基础知识 (原书第11版)', 'Core Java 第11版, Jolt大奖获奖作品, 针对Java SE9、10、11全面更新');
```

```
INSERT INTO `tbl_book` VALUES (7, '计算机理论', '深入理解Java虚拟机', '5个维度全面剖析JVM, 大厂面试知识点全覆盖');
INSERT INTO `tbl_book` VALUES (8, '计算机理论', 'Java编程思想(第4版)', 'Java学习必读经典, 殿堂级著作! 赢得了全球程序员的广泛赞誉');
INSERT INTO `tbl_book` VALUES (9, '计算机理论', '零基础学Java(全彩版)', '零基础自学编程的入门图书, 由浅入深, 详解Java语言的编程思想和核心技术');
INSERT INTO `tbl_book` VALUES (10, '市场营销', '直播就该这么做: 主播高效沟通实战指南', '李子柒、李佳琦、薇娅成长为网红的秘密都在书中');
INSERT INTO `tbl_book` VALUES (11, '市场营销', '直播销讲实战一本通', '和秋叶一起学系列网络营销书籍');
INSERT INTO `tbl_book` VALUES (12, '市场营销', '直播带货: 淘宝、天猫直播从新手到高手', '一本教你如何玩转直播的书, 10堂课轻松实现带货月入3W+');
```

根据上述表结构，制作对应的实体类

实体类

```
public class Book {
    private Integer id;
    private String type;
    private String name;
    private String description;
}
```

实体类的开发可以自动通过工具手工生成get/set方法，然后覆盖toString()方法，方便调试，等等。不过这一套操作书写很繁琐，有对应的工具可以帮助我们简化开发，介绍一个小工具，lombok。

Lombok，一个Java类库，提供了一组注解，简化POJO实体类开发，SpringBoot目前默认集成了lombok技术，并提供了对应的版本控制，所以只需要提供对应的坐标即可，在pom.xml中添加lombok的坐标。

```
<dependencies>
    <!--lombok-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
</dependencies>
```

使用lombok可以通过一个注解@Data完成一个实体类对应的getter, setter, toString, equals, hashCode等操作的快速添加

```
import lombok.Data;
@Data
public class Book {
    private Integer id;
    private String type;
    private String name;
    private String description;
}
```

到这里实体类就做好了，是不是比不使用lombok简化好多，这种工具在Java开发中还有N多，后面遇到了能用的实用开发技术时，在不增加各位小伙伴大量的学习时间的情况下，尽量多给大家介绍一些。

总结

1. 实体类制作
2. 使用lombok简化开发
 - 导入lombok无需指定版本，由SpringBoot提供版本
 - @Data注解

2.数据层开发——基础CRUD

数据层开发本次使用MyBatisPlus技术，数据源使用前面学习的Druid，学都学了都用上。

步骤①：导入MyBatisPlus与Druid对应的starter，当然mysql的驱动不能少

```
<dependencies>
    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>mybatis-plus-boot-starter</artifactId>
        <version>3.4.3</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.2.6</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

步骤②：配置数据库连接相关的数据源配置

```
server:
  port: 80

spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
      username: root
      password: root
```

步骤③：使用MyBatisPlus的标准通用接口BaseMapper加速开发，别忘了@Mapper和泛型的指定

```
@Mapper
public interface BookDao extends BaseMapper<Book> {
}
```

步骤④：制作测试类测试结果，这个测试类制作是个好习惯，不过在企业开发中往往都为加速开发跳过此步，且行且珍惜吧

```
package com.haohao.dao;

import com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.core.metadata.IPage;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.itheima.domain.Book;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class BookDaoTestCase {

    @Autowired
    private BookDao bookDao;

    @Test
    void test GetById() {
        System.out.println(bookDao.selectById(1));
    }

    @Test
    void test Save() {
        Book book = new Book();
        book.setType("测试数据123");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookDao.insert(book);
    }

    @Test
    void test Update() {
        Book book = new Book();
        book.setId(17);
        book.setType("测试数据abcdefg");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookDao.updateById(book);
    }

    @Test
    void test Delete() {
        bookDao.deleteById(16);
    }

    @Test
```

```
void testGetAll(){
    bookDao.selectList(null);
}
```

温馨提示

MyBatisPlus技术默认的主键生成策略为雪花算法，生成的主键ID长度较大，和目前的数据库设定规则不相符，需要配置一下使MyBatisPlus使用数据库的主键生成策略，方式嘛还是老一套，做配置。在application.yml中添加对应配置即可，具体如下

```
server:
  port: 80

spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
      username: root
      password: root

  mybatis-plus:
    global-config:
      db-config:
        table-prefix: tbl_          #设置表名通用前缀
        id-type: auto              #设置主键id字段的生成策略为参照数据库设定的策略，当前数据
                                    #库设置id生成策略为自增
```

查看MyBatisPlus运行日志

在进行数据层测试的时候，因为基础的CRUD操作均由MyBatisPlus给我们提供了，所以就出现了一个局面，开发者不需要书写SQL语句了，这样程序运行的时候总有一种感觉，一切的一切都是黑盒的，作为开发者我们啥也不知道就完了。如果程序正常运行还好，如果报错了，这个时候就很崩溃，你甚至都不知道从何下手，因为传递参数、封装SQL语句这些操作完全不是你开发出来的，所以查看执行期运行的SQL语句就成为当务之急。

SpringBoot整合MyBatisPlus的时候充分考虑到了这点，通过配置的形式就可以查阅执行期SQL语句，配置如下

```
mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_
      id-type: auto
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
```

再来看运行结果，此时就显示了运行期执行SQL的情况。

```
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@2c9a6717] was
not registered for synchronization because synchronization is not active
```

```
JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@6ca30b8a] will not be managed  
by Spring  
==> Preparing: SELECT id,type,name,description FROM tbl_book  
==> Parameters:  
<== Columns: id, type, name, description  
<== Row: 1, 计算机理论, Spring实战 第5版, Spring入门经典教程, 深入理解Spring原理  
技术内幕  
<== Row: 2, 计算机理论, Spring 5核心原理与30个类手写实战, 十年沉淀之作, 手写Spring  
精华思想  
<== Row: 3, 计算机理论, Spring 5 设计模式, 深入Spring源码剖析spring源码中蕴含的10  
大设计模式  
<== Row: 4, 计算机理论, Spring MVC+MyBatis开发从入门到项目实战, 全方位解析面向web  
应用的轻量级框架, 带你成为Spring MVC开发高手  
<== Row: 5, 计算机理论, 轻量级Java web企业应用实战, 源码级剖析Spring框架, 适合已掌  
握Java基础的读者  
<== Row: 6, 计算机理论, Java核心技术 卷I 基础知识 (原书第11版), Core Java 第11  
版, Jolt大奖获奖作品, 针对Java SE9、10、11全面更新  
<== Row: 7, 计算机理论, 深入理解Java虚拟机, 5个维度全面剖析JVM, 大厂面试知识点全覆盖  
<== Row: 8, 计算机理论, Java编程思想 (第4版), Java学习必读经典, 殿堂级著作! 赢得了  
全球程序员的广泛赞誉  
<== Row: 9, 计算机理论, 零基础学Java (全彩版), 零基础自学编程的入门图书, 由浅入深,  
详解Java语言的编程思想和核心技术  
<== Row: 10, 市场营销, 直播就该这么做: 主播高效沟通实战指南, 李子柒、李佳琦、薇娅成长  
为网红的秘密都在书中  
<== Row: 11, 市场营销, 直播销讲实战一本通, 和秋叶一起学系列网络营销书籍  
<== Row: 12, 市场营销, 直播带货: 淘宝、天猫直播从新手到高手, 一本教你如何玩转直播的  
书, 10堂课轻松实现带货月入3W+  
<== Row: 13, 测试类型, 测试数据, 测试描述数据  
<== Row: 14, 测试数据update, 测试数据update, 测试数据update  
<== Row: 15, -----, 测试数据123, 测试数据123  
<== Total: 15
```

其中清晰的标注了当前执行的SQL语句是什么，携带了什么参数，对应的执行结果是什么，所有信息应有尽有。

此处设置的是日志的显示形式，当前配置的是控制台输出，当然还可以由更多的选择，根据需求切换即可

```
mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_
  configuration:
    log-impl: | |
      c StdOutImpl (org.apache.ibatis.logging.stdout)
      c JakartaCommonsLoggingImpl (org.apache.ibatis.log...
      c Jdk14LoggingImpl (org.apache.ibatis.logging.jdk1...
      c Log4j2AbstractLoggerImpl (org.apache.ibatis.logg...
      c Log4j2Impl (org.apache.ibatis.logging.log4j2)
      c Log4j2LoggerImpl (org.apache.ibatis.logging.log4...
      c Log4jImpl (org.apache.ibatis.logging.log4j)
      c NoLoggingImpl (org.apache.ibatis.logging.nologgi...
      c Slf4jImpl (org.apache.ibatis.logging.slf4j)
      c o Slf4jLocationAwareLoggerImpl (org.apache.ibatis...
      c o Slf4jLoggerImpl (org.apache.ibatis.logging.slf4j)
Press Ctrl+ to choose the selected (or first) suggestion and insert a dot afterwards Next Tip ☰ ::
```

总结

1. 手工导入starter坐标（2个），mysql驱动（1个）
2. 配置数据源与MyBatisPlus对应的配置
3. 开发Dao接口（继承BaseMapper）
4. 制作测试类测试Dao功能是否有效
5. 使用配置方式开启日志，设置日志输出方式为标准输出即可查阅SQL执行日志

3.数据层开发——分页功能制作

前面仅仅是使用了MyBatisPlus提供的基础CRUD功能，实际上MyBatisPlus给我们提供了几乎所有基础操作，这一节说一下如何实现数据库端的分页操作。

MyBatisPlus提供的分页操作API如下：

```
@Test
void testGetPage(){
  IPage page = new Page(2,5);
  bookDao.selectPage(page, null);
  System.out.println(page.getCurrent());
  System.out.println(page.getSize());
  System.out.println(page.getTotal());
  System.out.println(page.getPages());
  System.out.println(page.getRecords());
}
```

其中selectPage方法需要传入一个封装分页数据的对象，可以通过new的形式创建这个对象，当然这个对象也是MyBatisPlus提供的，别选错了。创建此对象时需要指定两个分页的基本数据

- 当前显示第几页
- 每页显示几条数据

可以通过创建Page对象时利用构造方法初始化这两个数据。

```
IPage page = new Page(2,5);
```

将该对象传入到查询方法selectPage后，可以得到查询结果，但是我们会发现当前操作查询结果返回值仍然是一个IPage对象，这又是怎么回事？

```
IPage page = bookDao.selectPage(page, null);
```

原来这个IPage对象中封装了若干个数据，而查询的结果作为IPage对象封装的一个数据存在的，可以理解为查询结果得到后，又塞到了这个IPage对象中，其实还是为了高度的封装，一个IPage描述了分页所有的信息。下面5个操作就是IPage对象中封装的所有信息了。

```
@Test
void testGetPage(){
    IPage page = new Page(2,5);
    bookDao.selectPage(page, null);
    System.out.println(page.getCurrent());           //当前页码值
    System.out.println(page.getSize());             //每页显示数
    System.out.println(page.getTotal());            //数据总量
    System.out.println(page.getPages());            //总页数
    System.out.println(page.getRecords());          //详细数据
}
```

到这里就知道这些数据如何获取了，但是当你去执行这个操作时，你会发现并不像我们分析的这样，实际上这个分页功能当前是无效的。为什么这样呢？这个要源于MyBatisPlus的内部机制。

对于MySQL的分页操作使用limit关键字进行，而并不是所有的数据库都使用limit关键字实现的，这个时候MyBatisPlus为了制作的兼容性强，将分页操作设置为基础查询操作的升级版，你可以理解为iPhone6与iPhone6S-PLUS的关系。

基础操作中有查询全部的功能，而在这个基础上只需要升级一下（PLUS）就可以得到分页操作。所以MyBatisPlus将分页操作做成了一个开关，你用分页功能就把开关开启，不用就不需要开启这个开关。而我们现在没有开启这个开关，所以分页操作是没有的。这个开关是通过MyBatisPlus的拦截器的形式存在的，其中的原理这里不分析了，有兴趣的小伙伴可以学习MyBatisPlus这门课程进行详细了解。具体设置方式如下：

定义MyBatisPlus拦截器并将其设置为Spring管控的bean

```
@Configuration
public class MPConfig {
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        //配置分页插件
        interceptor.addInnerInterceptor(new PaginationInnerInterceptor());
        return interceptor;
    }
}
```

上述代码第一行是创建MyBatisPlus的拦截器栈，这个时候拦截器栈中没有具体的拦截器，第二行是初始化了分页拦截器，并添加到拦截器栈中。如果后期开发其他功能，需要添加全新的拦截器，按照第二行的格式继续add进去新的拦截器就可以了。

总结

1. 使用IPage封装分页数据
2. 分页操作依赖MyBatisPlus分页拦截器实现功能
3. 借助MyBatisPlus日志查阅执行SQL语句

4. 数据层开发——条件查询功能制作

除了分页功能，MyBatisPlus还提供有强大的条件查询功能。以往我们写条件查询要自己动态拼写复杂的SQL语句，现在简单了，MyBatisPlus将这些操作都制作成API接口，调用一个又一个的方法就可以实现各种条件的拼装。这里给大家普及一下基本格式，详细的操作还是到MyBatisPlus的课程中查阅吧。

下面的操作就是执行一个模糊匹配对应的操作，由like条件书写变为了like方法的调用。

```
@Test  
void testGetBy(){  
    QueryWrapper<Book> qw = new QueryWrapper<>();  
    qw.like("name", "Spring");  
    bookDao.selectList(qw);  
}
```

其中第一句QueryWrapper对象是一个用于封装查询条件的对象，该对象可以动态使用API调用的方法添加条件，最终转化成对应的SQL语句。第二句就是一个条件了，需要什么条件，使用QueryWrapper对象直接调用对应操作即可。比如做大于小于关系，就可以使用lt或gt方法，等于使用eq方法，等等，此处不做更多的解释了。

这组API使用还是比较简单的，但是关于属性字段名的书写存在着安全隐患，比如查询字段name，当前是以字符串的形态书写的，万一写错，编译器还没有办法发现，只能将问题抛到运行器通过异常堆栈告诉开发者，不太友好。

MyBatisPlus针对字段检查进行了功能升级，全面支持Lambda表达式，就有了下面这组API。由QueryWrapper对象升级为LambdaQueryWrapper对象，这下就避免了上述问题的出现。

```
@Test  
void testGetBy2(){  
    String name = "1";  
    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();  
    lqw.like(Book::getName, name);  
    bookDao.selectList(lqw);  
}
```

为了便于开发者动态拼写SQL，防止将null数据作为条件使用，MyBatisPlus还提供了动态拼装SQL的快捷书写方式。

```
@Test
void testGetBy2() {
    String name = "1";
    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();
    //if(name != null) lqw.like(Book::getName,name);           //方式一：JAVA代码控制
    lqw.like(name != null,Book::getName,name);                //方式二：API接口提供控制开关
    bookDao.selectList(lqw);
}
```

其实就是个格式，没有区别。关于MyBatisPlus的基础操作就说到这里吧，如果这一块知识不太熟悉的小伙伴建议还是完整的学习一下MyBatisPlus的知识吧，这里只是蜻蜓点水的用了几个操作而已。

总结

1. 使用QueryWrapper对象封装查询条件
2. 推荐使用LambdaQueryWrapper对象
3. 所有查询操作封装成方法调用
4. 查询条件支持动态条件拼装

5.业务层开发

数据层开发告一段落，下面进行业务层开发，其实标准业务层开发很多初学者认为就是调用数据层，怎么说呢？这个理解是没有大问题的，更精准的说法应该是**组织业务逻辑功能，并根据业务需求，对数据持久层发起调用**。有什么差别呢？目标是为了组织出符合需求的业务逻辑功能，至于调不调用数据层还真不好说，有需求就调用，没有需求就不调用。

一个常识性的知识普及一下，业务层的方法名定义一定要与业务有关，例如登录操作

```
login(String username, String password);
```

而数据层的方法名定义一定与业务无关，是一定，不是可能，也不是有可能，例如根据用户名密码查询

```
selectByUserNameAndPassword(String username, String password);
```

我们在开发的时候是可以根据完成的工作不同划分成不同职能的开发团队的。比如一个哥们制作数据层，他就可以不知道业务是什么样子，拿到的需求文档要求可能是这样的

```
接口：传入用户名与密码字段，查询出对应结果，结果是单条数据
接口：传入ID字段，查询出对应结果，结果是单条数据
接口：传入离职字段，查询出对应结果，结果是多条数据
```

但是进行业务功能开发的哥们，拿到的需求文档要求差别就很大

```
接口：传入用户名与密码字段，对用户名字段做长度校验，4-15位，对密码字段做长度校验，8到24位，对密码字段做特殊字符校验，不允许存在空格，查询结果为对象。如果为null，返回BusinessException，封装消息码INFO_LOGON_USERNAME_PASSWORD_ERROR
```

你比较一下，能是一回事吗？差别太大了，所以说业务层方法定义与数据层方法定义差异化很大，只不过有些入门级的开发者手懒或者没有使用过公司相关的ISO标准化文档而已。

多余的话不说了，咱们做案例就简单制作了，业务层接口定义如下：

```
public interface BookService {  
    Boolean save(Book book);  
    Boolean update(Book book);  
    Boolean delete(Integer id);  
    Book getById(Integer id);  
    List<Book> getAll();  
    IPage<Book> getPage(int currentPage, int pagesize);  
}
```

业务层实现类如下，转调数据层即可：

```
@Service  
public class BookServiceImpl implements BookService {  
  
    @Autowired  
    private BookDao bookDao;  
  
    @Override  
    public Boolean save(Book book) {  
        return bookDao.insert(book) > 0;  
    }  
  
    @Override  
    public Boolean update(Book book) {  
        return bookDao.updateById(book) > 0;  
    }  
  
    @Override  
    public Boolean delete(Integer id) {  
        return bookDao.deleteById(id) > 0;  
    }  
  
    @Override  
    public Book getById(Integer id) {  
        return bookDao.selectById(id);  
    }  
  
    @Override  
    public List<Book> getAll() {  
        return bookDao.selectList(null);  
    }  
  
    @Override  
    public IPage<Book> getPage(int currentPage, int pagesize) {  
        IPage page = new Page(currentPage, pageSize);  
        bookDao.selectPage(page, null);  
        return page;  
    }  
}
```

别忘了对业务层接口进行测试，测试类如下：

```
@SpringBootTest
public class BookServiceTest {
    @Autowired
    private IBookService bookService;

    @Test
    void test GetById() {
        System.out.println(bookService.getById(4));
    }

    @Test
    void testSave() {
        Book book = new Book();
        book.setType("测试数据123");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookService.save(book);
    }

    @Test
    void testUpdate() {
        Book book = new Book();
        book.setId(17);
        book.setType("-----");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookService.updateById(book);
    }

    @Test
    void testDelete() {
        bookService.removeById(18);
    }

    @Test
    void test GetAll() {
        bookService.list();
    }

    @Test
    void testGetPage() {
        IPage<Book> page = new Page<Book>(2, 5);
        bookService.page(page);
        System.out.println(page.getCurrent());
        System.out.println(page.getSize());
        System.out.println(page.getTotal());
        System.out.println(page.getPages());
        System.out.println(page.getRecords());
    }
}
```

总结

1. Service接口名称定义成业务名称，并与Dao接口名称进行区分
2. 制作测试类测试Service功能是否有效

业务层快速开发

其实MyBatisPlus技术不仅提供了数据层快速开发方案，业务层MyBatisPlus也给了一个通用接口，个人观点不推荐使用，凑合能用吧，其实就是一个封装+继承的思想，代码给出，实际开发慎用。

业务层接口快速开发

```
public interface IBookService extends IService<Book> {  
    //添加非通用操作API接口  
}
```

业务层接口实现类快速开发，关注继承的类需要传入两个泛型，一个是数据层接口，另一个是实体类。

```
@Service  
public class BookServiceImpl extends ServiceImpl<BookDao, Book> implements  
IBookService {  
    @Autowired  
    private BookDao bookDao;  
    //添加非通用操作API  
}
```

如果感觉MyBatisPlus提供的功能不足以支撑你的使用需要（其实是一定不能支撑的，因为需求不可能是通用的），在原始接口基础上接着定义新的API接口就行了，此处不说太多了，就是自定义自己的操作了，但是不要和已有的API接口名冲突即可。

总结

1. 使用通用接口 (IService) 快速开发Service
2. 使用通用实现类 (ServiceImpl<M,T>) 快速开发ServiceImpl
3. 可以在通用接口基础上做功能重载或功能追加
4. 注意重载时不要覆盖原始操作，避免原始提供的功能丢失

6.表现层开发

终于做到表现层了，做了这么多都是基础工作。其实你现在回头看看，哪里还有什么SpringBoot的影子？前面1,2步就搞完了。继续完成表现层制作吧，咱们表现层的开发使用基于Restful的表现层接口开发，功能测试通过Postman工具进行。

表现层接口如下：

```
@RestController  
@RequestMapping("/books")  
public class BookController2 {  
  
    @Autowired  
    private IBookService bookService;  
  
    @GetMapping  
    public List<Book> getAll(){  
        return bookService.list();
```

```

}

@PostMapping
public Boolean save(@RequestBody Book book){
    return bookService.save(book);
}

@PutMapping
public Boolean update(@RequestBody Book book){
    return bookService.modify(book);
}

@DeleteMapping("{id}")
public Boolean delete(@PathVariable Integer id){
    return bookService.delete(id);
}

@GetMapping("{id}")
public Book getById(@PathVariable Integer id){
    return bookService.getById(id);
}

@GetMapping("{currentPage}/{pageSize}")
public IPage<Book> getPage(@PathVariable int currentPage,@PathVariable int
pageSize){
    return bookService.getPage(currentPage,pageSize, null);
}
}

```

在使用Postman测试时关注提交类型，对应上即可，不然就会报405的错误码了。

普通GET请求

The screenshot shows the Postman interface with the following details:

- Request URL:** GET localhost/books
- Method:** GET
- Headers:** None
- Body:** None
- Query Params:** None
- Response:**
 - Status: 200 OK
 - Time: 312 ms
 - Size: 2.37 KB
 - Description: Spring入门经典教程，深入理解Spring原理技术内幕
 - Content (Pretty):

```

1 [
2   {
3     "id": 1,
4     "type": "计算机理论",
5     "name": "Spring实战 第5版",
6     "description": "Spring入门经典教程，深入理解Spring原理技术内幕"
7   }
]

```

PUT请求传递json数据，后台实用@RequestBody接收数据

PUT localhost/books

localhost/books

Body (JSON)

```

1 {
2   "id":15,
3   "name": "测试数据update",
4   "type": "测试数据update",
5   "description": "测试数据update"
6 }

```

Status: 200 OK Time: 107 ms Size: 168 B Save Response

GET请求传递路径变量，后台实用@PathVariable接收数据

GET localhost/books/1/5

localhost/books/1/5

Params (Query Params)

KEY	VALUE	DESCRIPTION	Bulk Edit
Key	Value	Description	

Status: 200 OK Time: 347 ms Size: 959 B Save Response

```

1 [
2   {
3     "id": 1,
4     "type": "计算机理论",
5     "name": "Spring实战 第5版",
6     "description": "Spring入门经典教程，深入理解Spring原理技术内幕"
7   },
8   {
9     "id": 2,
10    "type": "计算机理论",
11   }
]

```

总结

1. 基于Restful制作表现层接口

- 新增: POST
- 删除: DELETE
- 修改: PUT
- 查询: GET

2. 接收参数

- 实体数据: @RequestBody
- 路径变量: @PathVariable

7. 表现层消息一致性处理

目前我们通过Postman测试后业务层接口功能是通的，但是这样的结果给到前端开发者会出现一个小问题。不同的操作结果所展示的数据格式差异化严重。

增删改操作结果

true

查询单个数据操作结果

```
{  
    "id": 1,  
    "type": "计算机理论",  
    "name": "Spring实战 第5版",  
    "description": "Spring入门经典教程"  
}
```

查询全部数据操作结果

```
[  
    {  
        "id": 1,  
        "type": "计算机理论",  
        "name": "Spring实战 第5版",  
        "description": "Spring入门经典教程"  
    },  
    {  
        "id": 2,  
        "type": "计算机理论",  
        "name": "Spring 5核心原理与30个类手写实战",  
        "description": "十年沉淀之作"  
    }  
]
```

每种不同操作返回的数据格式都不一样，而且还不知道以后还会有什么格式，这样的结果让前端人员看了是很容易让人崩溃的，必须将所有操作的操作结果数据格式统一起来，需要设计表现层返回结果的模型类，用于后端与前端进行数据格式统一，也称为**前后端数据协议**

```
@Data  
public class R {  
    private Boolean flag;  
    private Object data;  
}
```

其中flag用于标识操作是否成功，data用于封装操作数据，现在的数据格式就变了

```
{  
    "flag": true,  
    "data":{  
        "id": 1,  
        "type": "计算机理论",  
        "name": "Spring实战 第5版",  
        "description": "Spring入门经典教程"  
    }  
}
```

表现层开发格式也需要转换一下

```

@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;
    @PostMapping
    public R save(@RequestBody Book book){
        Boolean flag = bookService.insert(book);
        return new R(flag);
    }
    @PutMapping
    public R update(@RequestBody Book book){
        Boolean flag = bookService.modify(book);
        return new R(flag);
    }
}

```

```

@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;
    @DeleteMapping("/{id}")
    public R delete(@PathVariable Integer id){
        Boolean flag = bookService.delete(id);
        return new R(flag);
    }
    @GetMapping("/{id}")
    public R getById(@PathVariable Integer id){
        Book book = bookService.getById(id);
        return new R(true,book);
    }
}

```

```

@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;
    @GetMapping
    public R getAll(){
        List<Book> bookList = bookService.list();
        return new R(true ,bookList);
    }
    @GetMapping("/{currentPage}/{pageSize}")
    public R getAll(@PathVariable Integer currentPage,@PathVariable Integer pageSize){
        IPage<Book> page = bookService.getPage(currentPage, pageSize);
        return new R(true,page);
    }
}

```

结果这么一折腾，全格式统一，现在后端发送给前端的数据格式就统一了，免去了不少前端解析数据的烦恼。

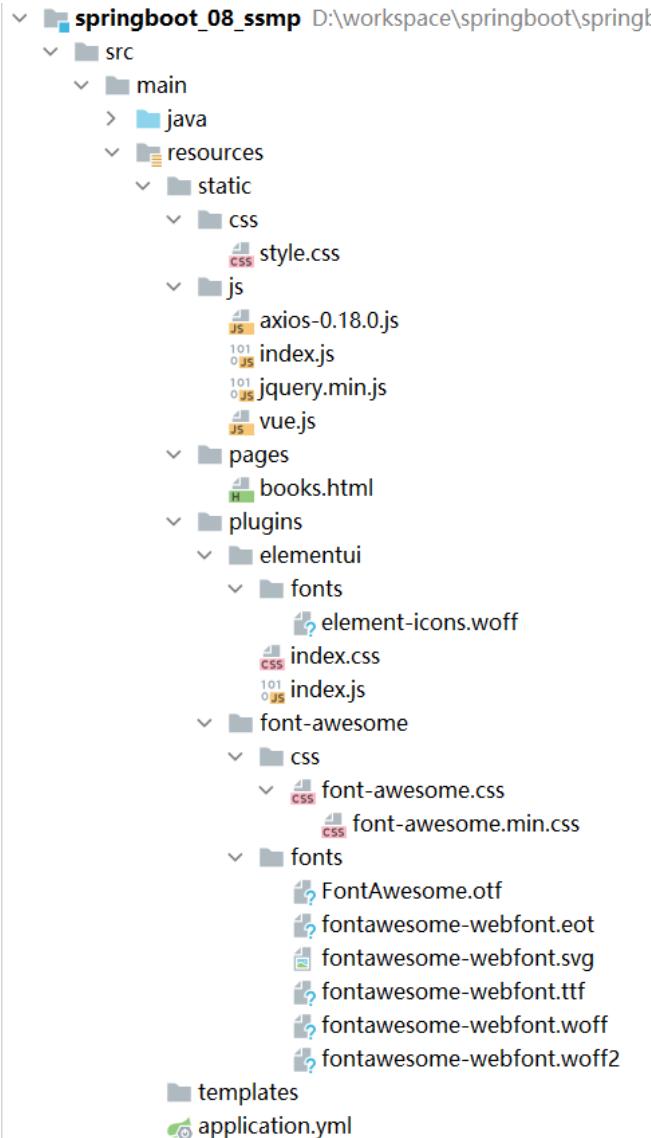
总结

1. 设计统一的返回值结果类型便于前端开发读取数据
2. 返回值结果类型可以根据需求自行设定，没有固定格式
3. 返回值结果模型类用于后端与前端进行数据格式统一，也称为前后端数据协议

8.前后端联通性测试

后端的表现层接口开发完毕，就可以进行前端的开发了。

将前端人员开发的页面保存到resources目录下的static目录中，建议执行maven的clean生命周期，避免缓存的问题出现。



在进行具体的功能开发之前，先做联通性的测试，通过页面发送异步提交（axios），这一步调试通过后再进行进一步的功能开发。

```
//列表
getAll() {
    axios.get("/books").then((res)=>{
        console.log(res.data);
    });
},
```

只要后台代码能够正常工作，前端能够在日志中接收到数据，就证明前后端是通的，也就可以进行下一步的功能开发了。

总结

1. 单体项目中页面放置在resources/static目录下
2. created钩子函数用于初始化页面时发起调用
3. 页面使用axios发送异步请求获取数据后确认前端是否联通

9. 页面基础功能开发

F-1. 列表功能（非分页版）

列表功能主要操作就是加载完数据，将数据展示到页面上，此处要利用VUE的数据模型绑定，发送请求得到数据，然后页面上读取指定数据即可。

页面数据模型定义

```
data: {  
    dataList: [], //当前页要展示的列表数据  
    ...  
},
```

异步请求获取数据

```
//列表  
getAll() {  
    axios.get("/books").then((res) => {  
        this.dataList = res.data.data;  
    });  
},
```

这样在页面加载时就可以获取到数据，并且由VUE将数据展示到页面上了。

总结：

1. 将查询数据返回到页面，利用前端数据绑定进行数据展示

F-2. 添加功能

添加功能用于收集数据的表单是通过一个弹窗展示的，因此在添加操作前首先要进行弹窗的展示，添加后隐藏弹窗即可。因为这个弹窗一直存在，因此当页面加载时首先设置这个弹窗为不可显示状态，需要展示，切换状态即可。

默认状态

```
data: {  
    dialogFormVisible: false, //添加表单是否可见  
    ...  
},
```

切换为显示状态

```
//弹出添加窗口  
handleCreate() {  
    this.dialogFormVisible = true;  
},
```

由于每次添加数据都是使用同一个弹窗录入数据，所以每次操作的痕迹将在下一次操作时展示出来，需要在每次操作之前清理掉上次操作的痕迹。

定义清理数据操作

```
//重置表单
resetForm() {
    this.formData = {};
},
}
```

切换弹窗状态时清理数据

```
//弹出添加窗口
handleCreate() {
    this.dialogFormVisible = true;
    this.resetForm();
},
}
```

至此准备工作完成，下面就要调用后台完成添加操作了。

添加操作

```
//添加
handleAdd () {
    //发送异步请求
    //这里的第二个参数this.formData就是组件中填的数据，至于为什么直接把第二个参数放到请求体
    //里面了，这个是axios的知识。。。
    axios.post("/books",this.formData).then((res)=>{
        //如果操作成功，关闭弹层，显示数据
        if(res.data.flag){
            this.dialogFormVisible = false;
            this.$message.success("添加成功");
        }else {
            this.$message.error("添加失败");
        }
    }).finally(()=>{
        this.getAll();
    });
},
}
```

1. 将要保存的数据传递到后台，通过post请求的第二个参数传递json数据到后台
2. 根据返回的操作结果决定下一步操作
 - 如何是true就关闭添加窗口，显示添加成功的消息
 - 如果是false保留添加窗口，显示添加失败的消息
3. 无论添加是否成功，页面均进行刷新，动态加载数据（对getAll操作发起调用）

取消添加操作

```
//取消
cancel(){
    this.dialogFormVisible = false;
    this.$message.info("操作取消");
},
}
```

总结

1. 请求方式使用POST调用后台对应操作
2. 添加操作结束后动态刷新页面加载数据
3. 根据操作结果不同，显示对应的提示信息
4. 弹出添加Div时清除表单数据

F-3.删除功能

模仿添加操作制作删除功能，差别之处在于删除操作仅传递一个待删除的数据id到后台即可。

删除操作

```
// 删除
handleDelete(row) {
  axios.delete("/books/" + row.id).then((res) => {
    if(res.data.flag){
      this.$message.success("删除成功");
    }else{
      this.$message.error("删除失败");
    }
  }).finally(() => {
    this.getAll();
  });
},
```

删除操作提示信息

```
// 删除
handleDelete(row) {
  //1.弹出提示框
  this.$confirm("此操作永久删除当前数据，是否继续？", "提示", {
    type: 'info'
  }).then(() => {
    //2.做删除业务
    axios.delete("/books/" + row.id).then((res) => {
      if(res.data.flag){
        this.$message.success("删除成功");
      }else{
        this.$message.error("删除失败");
      }
    }).finally(() => {
      this.getAll();
    });
  }).catch(() => {
    //3.取消删除
    this.$message.info("取消删除操作");
  });
},
```

总结

1. 请求方式使用Delete调用后台对应操作
2. 删除操作需要传递当前行数据对应的id值到后台
3. 删除操作结束后动态刷新页面加载数据

4. 根据操作结果不同，显示对应的提示信息
5. 删除操作前弹出提示框避免误操作

F-4.修改功能

修改功能可以说是列表功能、删除功能与添加功能的合体。几个相似点如下：

1. 页面也需要有一个弹窗用来加载修改的数据，这一点与添加相同，都是要弹窗
2. 弹出窗口中要加载待修改的数据，而数据需要通过查询得到，这一点与查询全部相同，都是要查数据
3. 查询操作需要将要修改的数据id发送到后台，这一点与删除相同，都是传递id到后台
4. 查询得到数据后需要展示到弹窗中，这一点与查询全部相同，都是要通过数据模型绑定展示数据
5. 修改数据时需要将被修改的数据传递到后台，这一点与添加相同，都是要传递数据

所以整体上来看，修改功能就是前面几个功能的大合体

查询并展示数据

```
//弹出编辑窗口
handleUpdate(row) {
  axios.get("/books/" + row.id).then((res) => {
    if(res.data.flag){
      //展示弹层，加载数据
      this.formData = res.data.data;
      this.dialogFormVisible4Edit = true;
    }else{
      this.$message.error("数据同步失败，自动刷新");
    }
  });
},

```

修改操作

```
//修改
handleEdit() {
  axios.put("/books", this.formData).then((res) => {
    //如果操作成功，关闭弹层并刷新页面
    if(res.data.flag){
      this.dialogFormVisible4Edit = false;
      this.$message.success("修改成功");
    }else {
      this.$message.error("修改失败，请重试");
    }
  }).finally(() => {
    this.getAll();
  });
},

```

总结

1. 加载要修改数据通过传递当前行数据对应的id值到后台查询数据（同删除与查询全部）
2. 利用前端双向数据绑定将查询到的数据进行回显（同查询全部）
3. 请求方式使用PUT调用后台对应操作（同新增传递数据）

4. 修改操作结束后动态刷新页面加载数据（同新增）
5. 根据操作结果不同，显示对应的提示信息（同新增）

10.业务消息一致性处理

目前的功能制作基本上达成了正常使用的情况，什么叫正常使用呢？也就是这个程序不出BUG，如果我们搞一个BUG出来，你会发现程序马上崩溃掉。比如后台手工抛出一个异常，看看前端接收到的数据什么样子。

```
{  
    "timestamp": "2021-09-15T03:27:31.038+00:00",  
    "status": 500,  
    "error": "Internal Server Error",  
    "path": "/books"  
}
```

面对这种情况，前端的同学又不会了，这又是什么格式？怎么和之前的格式不一样？

```
{  
    "flag": true,  
    "data":{  
        "id": 1,  
        "type": "计算机理论",  
        "name": "Spring实战 第5版",  
        "description": "Spring入门经典教程"  
    }  
}
```

看来不仅要对正确的操作数据格式做处理，还要对错误的操作数据格式做同样的格式处理。

首先在当前的数据结果中添加消息字段，用来兼容后台出现的操作消息。

```
@Data  
public class R{  
    private Boolean flag;  
    private Object data;  
    private String msg;      //用于封装消息  
}
```

后台代码也要根据情况做处理，当前是模拟的错误。

```
@PostMapping  
public R save(@RequestBody Book book) throws IOException {  
    Boolean flag = bookService.insert(book);  
    return new R(flag , flag ? "添加成功^_^" : "添加失败-_-!");  
}
```

然后在表现层做统一的异常处理，使用SpringMVC提供的异常处理器做统一的异常处理。

```

//使用SpringMVC提供的异常处理器做统一的异常处理。
@RestControllerAdvice
public class ProjectExceptionAdvice {
    @ExceptionHandler(Exception.class)
    public R doOtherException(Exception ex){
        //记录日志
        //发送消息给运维
        //发送邮件给开发人员，ex对象发送给开发人员
        ex.printStackTrace();
        //发送错误消息
        return new R("系统错误，请稍后再试！");
    }
}

```

页面上得到数据后，先判定是否有后台传递过来的消息，标志就是当前操作是否成功，如果返回操作结果false，就读取后台传递的消息。

```

//添加
handleAdd () {
    //发送ajax请求
    axios.post("/books",this.formData).then((res)=>{
        //如果操作成功，关闭弹层，显示数据
        if(res.data.flag){
            this.dialogFormVisible = false;
            this.$message.success("添加成功");
        }else {
            this.$message.error(res.data.msg);           //消息来自于后台传递过来，而非固定内容
        }
    }).finally(()=>{
        this.getAll();
    });
},

```

总结

1. 使用注解@RestControllerAdvice定义SpringMVC异常处理器用来处理异常的
2. 异常处理器必须被扫描加载，否则无法生效
3. 表现层返回结果的模型类中添加消息属性用来传递消息到页面

11. 页面功能开发

F-5. 分页功能

分页功能的制作用于替换前面的查询全部，其中要使用到elementUI提供的分页组件。

```
<!--分页组件-->
<div class="pagination-container">
  <el-pagination
    class="pagiantion"
    @current-change="handleCurrentChange"
    :current-page="pagination.currentPage"
    :page-size="pagination.pageSize"
    layout="total, prev, pager, next, jumper"
    :total="pagination.total">
  </el-pagination>
</div>
```

为了配合分页组件，封装分页对应的数据模型。

```
data: {
  pagination: {
    //分页相关模型数据
    currentPage: 1, //当前页码
    pageSize: 10, //每页显示的记录数
    total: 0, //总记录数
  }
},
```

修改查询全部功能为分页查询，通过路径变量传递页码信息参数。

```
getAll() {
  axios.get("/books/" + this.pagination.currentPage + "/" + this.pagination.pageSize).then((res) => {
    });
},
```

后台提供对应的分页功能。

```
@GetMapping("/{currentPage}/{pageSize}")
public R getAll(@PathVariable Integer currentPage, @PathVariable Integer pageSize) {
  IPage<Book> pageBook = bookService.getPage(currentPage, pageSize);
  return new R(null != pageBook, pageBook);
}
```

页面根据分页操作结果读取对应数据，并进行数据模型绑定。

```

getAll() {

    axios.get("/books/" + this.pagination.currentPage + "/" + this.pagination.pageSize).then((res) => {
        this.pagination.total = res.data.data.total;
        this.pagination.currentPage = res.data.data.current;
        this.pagination.pageSize = res.data.data.size;
        this.dataList = res.data.data.records;
    });
},

```

对切换页码操作设置调用当前分页操作。

```

//切换页码
handleCurrentChange(currentPage) {
    this.pagination.currentPage = currentPage;
    this.getAll();
},

```

总结

1. 使用el分页组件
2. 定义分页组件绑定的数据模型
3. 异步调用获取分页数据
4. 分页数据页面回显

F-6.删除功能维护

由于使用了分页功能，当最后一页只有一条数据时，删除操作就会出现BUG，最后一页无数据但是独立展示，对分页查询功能进行后台功能维护，如果当前页码值大于最大页码值，重新执行查询。其实这个问题解决方案很多，这里给出比较简单的一种处理方案。

```

@GetMapping("{currentPage}/{pageSize}")
public R getPage(@PathVariable int currentPage,@PathVariable int pageSize){
    IPage<Book> page = bookService.getPage(currentPage, pageSize);
    //如果当前页码值大于了总页码值，那么重新执行查询操作，使用最大页码值作为当前页码值
    if( currentPage > page.getPages()){
        page = bookService.getPage((int)page.getPages(), pageSize);
    }
    return new R(true, page);
}

```

F-7.条件查询功能

最后一个功能来做条件查询，其实条件查询可以理解为分页查询的时候除了携带分页数据再多带几个数据的查询。这些多带的数据就是查询条件。比较一下不带条件的分页查询与带条件的分页查询差别之处，这个功能就好做了

- 页面封装的数据：带不带条件影响的仅仅是一次性传递到后台的数据总量，由传递2个分页相关数据转换成2个分页数据加若干个条件

- 后台查询功能：查询时由不带条件，转换成带条件，反正不带条件的时候查询条件对象使用的是null，现在换成具体条件，差别不大
 - 查询结果：不管带不带条件，出来的数据只是有数量上的差别，其他都差别，这个可以忽略
- 经过上述分析，看来需要在页面发送请求的格式方面做一定的修改，后台的调用数据层操作时发送修改，其他没有区别。

页面发送请求时，两个分页数据仍然使用路径变量，其他条件采用动态拼装url参数的形式传递。

页面封装查询条件字段

```
pagination: {
  //分页相关模型数据
  currentPage: 1,      //当前页码
  pageSize: 10,        //每页显示的记录数
  total: 0,            //总记录数
  name: "",            //图书名称
  type: "",            //图书类别
  description: ""     //图书描述
},
```

页面添加查询条件字段对应的数据模型绑定名称

```
<div class="filter-container">
  <el-input placeholder="图书类别" v-model="pagination.type" class="filter-item"/>
  <el-input placeholder="图书名称" v-model="pagination.name" class="filter-item"/>
  <el-input placeholder="图书描述" v-model="pagination.description" class="filter-item"/>
  <el-button @click="getAll()" class="defaultBut">查询</el-button>
  <el-button type="primary" class="butT" @click="handleCreate()">新建</el-button>
</div>
```

将查询条件组织成url参数，添加到请求url地址中，这里可以借助其他类库快速开发，当前使用手工形式拼接，降低学习要求

```
getAll() {
  //1. 获取查询条件，拼接查询条件
  //注意这第一个，很有说法，这个q没啥用，只是为了统一后面的参数格式
  param = "?q";
  param += "&name=" + this.pagination.name;
  param += "&type=" + this.pagination.type;
  param += "&description=" + this.pagination.description;
  console.log("-----" + param);

  axios.get("/books/" + this.pagination.currentPage + "/" + this.pagination.pageSize + param).then((res) => {
    this.dataList = res.data.data.records;
  });
},
```

后台代码中定义实体类封查询条件

```

@GetMapping("{currentPage}/{pageSize}")
public R getAll(@PathVariable int currentPage,@PathVariable int
pageSize,Book book) {
    System.out.println("参数===== "+book);
    IPage<Book> pageBook = bookService.getPage(currentPage,pageSize);
    return new R(null != pageBook ,pageBook);
}

```

对应业务层接口与实现类进行修正

```

public interface IBookService extends IService<Book> {
    IPage<Book> getPage(Integer currentPage,Integer pageSize,Book
queryBook);
}

```

```

@Service
public class BookServiceImpl2 extends ServiceImpl<BookDao,Book> implements
IBookService {
    public IPage<Book> getPage(Integer currentPage,Integer pagesize,Book
queryBook){
        IPage page = new Page(currentPage,pageSize);
        LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

        lqw.like(StringUtils.isNotEmpty(queryBook.getName()),Book::getName,queryBook.ge
tName());

        lqw.like(StringUtils.isNotEmpty(queryBook.getType()),Book::getType,queryBook.ge
tType());

        lqw.like(StringUtils.isNotEmpty(queryBook.getDescription()),Book::getDescription,
queryBook.getDescription());
        return bookDao.selectPage(page,lqw);
    }
}

```

页面回显数据

```

getAll() {
    //1. 获取查询条件,拼接查询条件
    param = "?name="+this.pagination.name;
    param += "&type="+this.pagination.type;
    param += "&description="+this.pagination.description;
    console.log("-----"+ param);

    axios.get("/books/"+this.pagination.currentPage+"/"+this.pagination.pageSize+param)
        .then((res) => {
            this.pagination.total = res.data.data.total;
            this.pagination.currentPage = res.data.data.current;
            this.pagination.pageSize = res.data.data.size;
            this.dataList = res.data.data.records;
        });
},

```

总结

1. 定义查询条件数据模型（当前封装到分页数据模型中）
2. 异步调用分页功能并通过请求参数传递数据到后台

基础篇完结

基础篇到这里就全部结束了，在基础篇中带着大家学习了如何创建一个SpringBoot工程，然后学习了SpringBoot的基础配置语法格式，接下来对常见的市面上的实用技术做了整合，最后通过一个小的案例对前面学习的内容做了一个综合应用。整体来说就是一个最基本的入门，关于SpringBoot的实际开发其实接触的还是很少的，我们到实用篇和原理篇中继续吧，各位小伙伴，加油学习，再见。

SpringBoot运维实用篇

基础篇发布以后，看到了很多小伙伴在网上的留言，也帮助超过100位小伙伴解决了一些遇到的问题，并且已经发现了部分问题具有典型性，预计将有些问题在后面篇章的合适位置添加到本套课程中，作为解决方案提供给大家。

从此刻开始，咱们就要进入到实用篇的学习了。实用篇是在基础篇的根基之上，补全SpringBoot的知识图谱。比如在基础篇中只给大家讲了yaml的语法格式，但是具体写yaml文件的时候还有很多实用开发过程中的坑，这些在实用篇中都要进行学习。

实用篇共分为两块内容，分别是运维实用篇和开发实用篇。其实划分的标准是我自己制定的，因为这里面的知识有一些还是比较散的，做两个阶段的划分是为了更好的将同类知识点进行归类，帮助学习者找到知识之间的关联性，这样有助于知识的记忆存储转换，经过一系列的知识反复出现与强化练习，将临时记忆转换成永久性记忆。做课程嘛，不能仅以讲完为目标，要以学习者的学习收获为目标，这也是我这么多年教学秉承的基本理念。

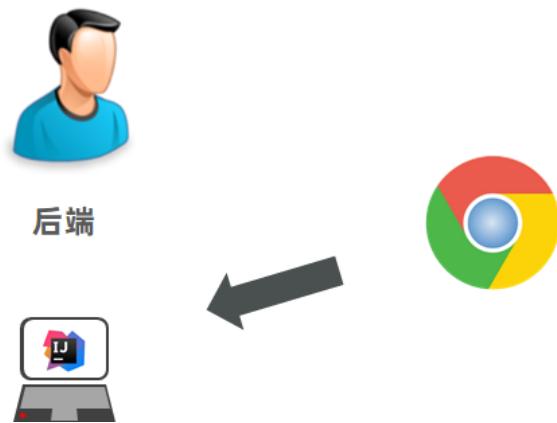
下面就从运维实用篇开始讲，在运维实用篇中，我给学习者的定位是玩转配置，为开发实用篇中做各种技术的整合做好准备工作。与开发实用篇相比，运维实用篇的内容显得略微单薄，并且有部分知识模块在运维实用篇和开发实用篇中都要讲一部分，这些内容都后置到开发实用篇中了。废话不说了，先看看运维实用篇中都包含哪些内容：

- SpringBoot程序的打包与运行
- 配置高级
- 多环境开发
- 日志

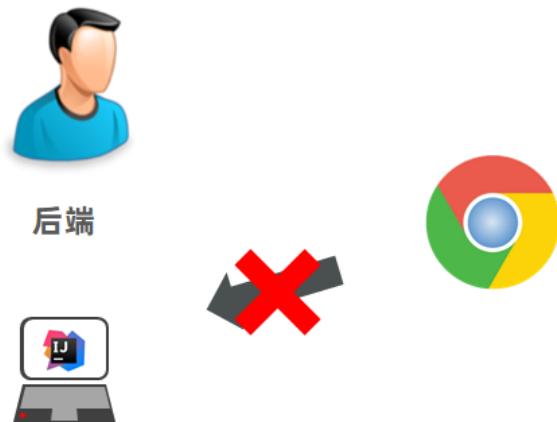
下面开启第一部分SpringBoot程序打包与运行的学习

YW-1.SpringBoot程序的打包与运行

刚开始做开发学习的小伙伴可能在有一个知识上面有错误的认知，我们天天写程序是在Idea下写的，运行也是在Idea下运行的。



但是实际开发完成后，我们的项目是不可能运行在自己的电脑上的。



我们以后制作的程序是运行在专用的服务器上的，简单说就是将你做的程序放在一台独立运行的电脑上，这台电脑要比你开发使用的计算机更专业，并且安全等级各个方面要远超过你现在的电脑。



那我们的程序如何放置在这台专用的电脑上呢，这就要将我们的程序先组织成一个文件，然后将这个文件传输到这台服务器上。这里面就存在两个过程，一个是打包的过程，另一个是运行的过程。

温馨提示

企业项目上线为了保障环境适配性会采用下面流程发布项目，这里不讨论此过程。

1. 开发部门使用Git、SVN等版本控制工具上传工程到版本服务器
2. 服务器使用版本控制工具下载工程
3. 服务器上使用Maven工具在当前真机环境下重新构建项目
4. 启动服务

继续说我们的打包和运行过程。所谓打包指将程序转换成一个可执行的文件，所谓运行指不依赖开发环境执行打包产生的文件。上述两个操作都有对应的命令可以快速执行。

程序打包

SpringBoot程序是基于Maven创建的，在Maven中提供有打包的指令，叫做package。本操作可以在Idea环境下执行。

```
mvn package
```

打包后会产生一个与工程名类似的jar文件，其名称是由模块名+版本号+.jar组成的。

程序运行

程序包打好以后，就可以直接执行了。在程序包所在路径下，执行指令。

```
java -jar 工程包名.jar
```

执行程序打包指令后，程序正常运行，与在Idea下执行程序没有区别。

特别关注：如果你的计算机中没有安装java的jdk环境，是无法正确执行上述操作的，因为程序执行使用的是java指令。

特别关注：在使用向导创建SpringBoot工程时，pom.xml文件中会有如下配置，这一段配置千万不能删除，否则打包后无法正常执行程序。

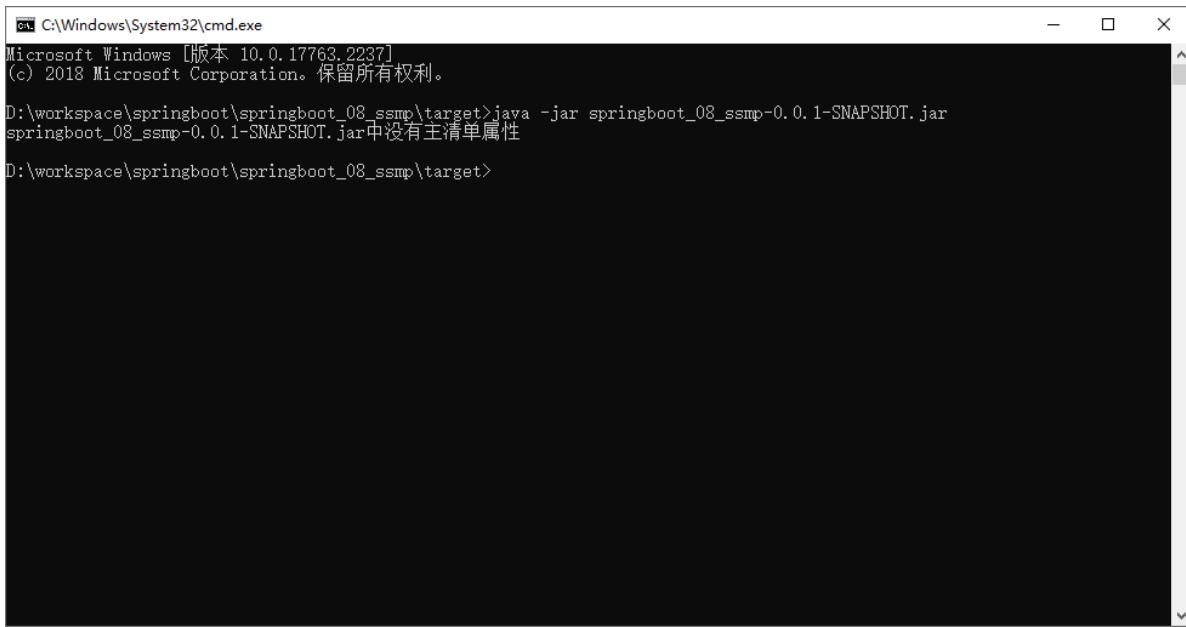
```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

总结

1. SpringBoot工程可以基于java环境下独立运行jar文件启动服务
2. SpringBoot工程执行mvn命令package进行打包
3. 执行jar命令：java -jar 工程名.jar

SpringBoot程序打包失败处理

有些小伙伴打包以后执行会出现一些问题，导致程序无法正常执行，例如下面的现象



```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.17763.2237]
(c) 2018 Microsoft Corporation. 保留所有权利。
D:\workspace\springboot\springboot_08_ssmp\target>java -jar springboot_08_ssmp-0.0.1-SNAPSHOT.jar
springboot_08_ssmp-0.0.1-SNAPSHOT.jar中没有主清单属性
D:\workspace\springboot\springboot_08_ssmp\target>
```

要想搞清楚这个问题就要说说jar文件的工作机制了，知道了这个东西就知道如何避免此类问题的发生了。

搞java开发平时会接触很多jar包，比如mysql的驱动jar包，而上面我们打包程序后得到的也是一个jar文件。这个时候如果你使用上面的java -jar指令去执行mysql的驱动jar包就会出现上述不可执行的现象，而我们的SpringBoot项目为什么能执行呢？其实是因为打包方式不一样。

在SpringBoot工程的pom.xml中有下面这组配置，这组配置决定了打包出来的程序包是否可以执行。

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

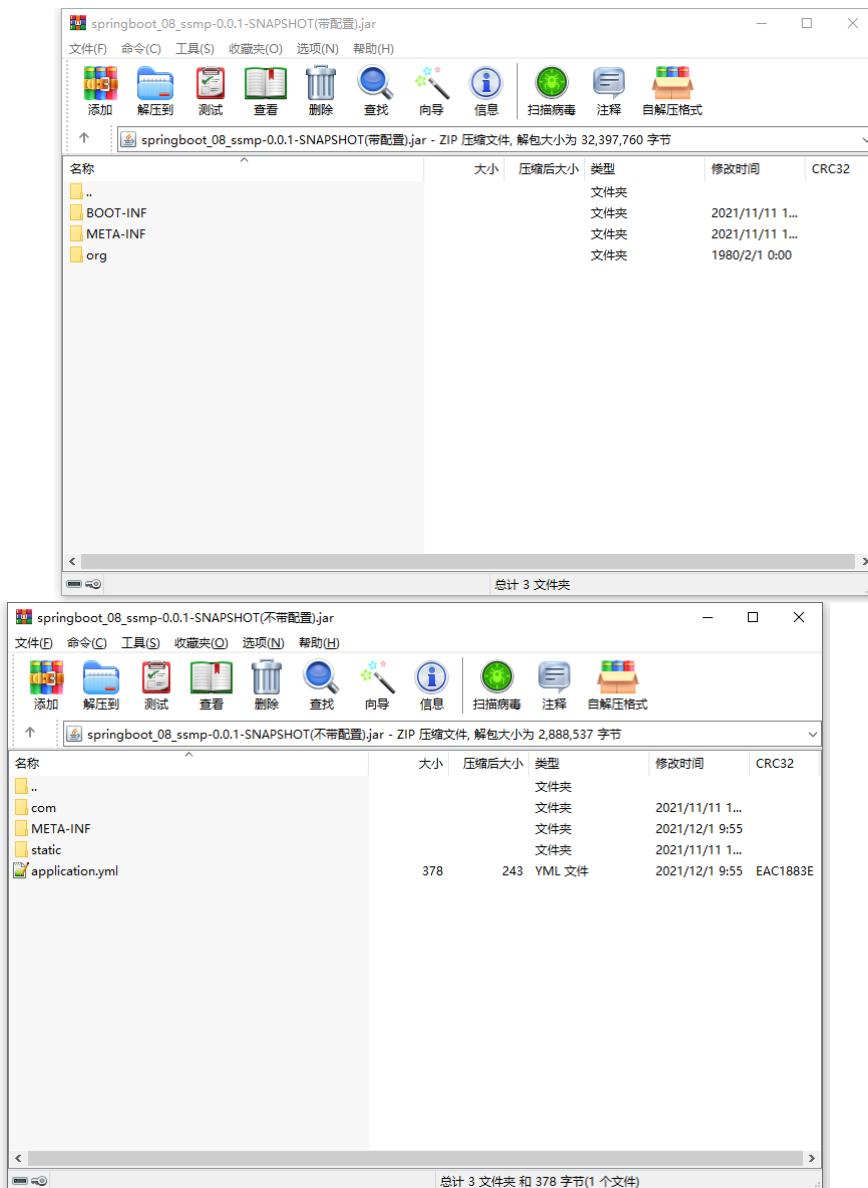
我们分别开启这段配置和注释掉这段配置分别执行两次打包，然后观察两次打包后的程序包的差别，共有3处比较明显的特征

- 打包后文件的大小不同
- 打包后所包含的内容不同
- 打包程序中个别文件内容不同

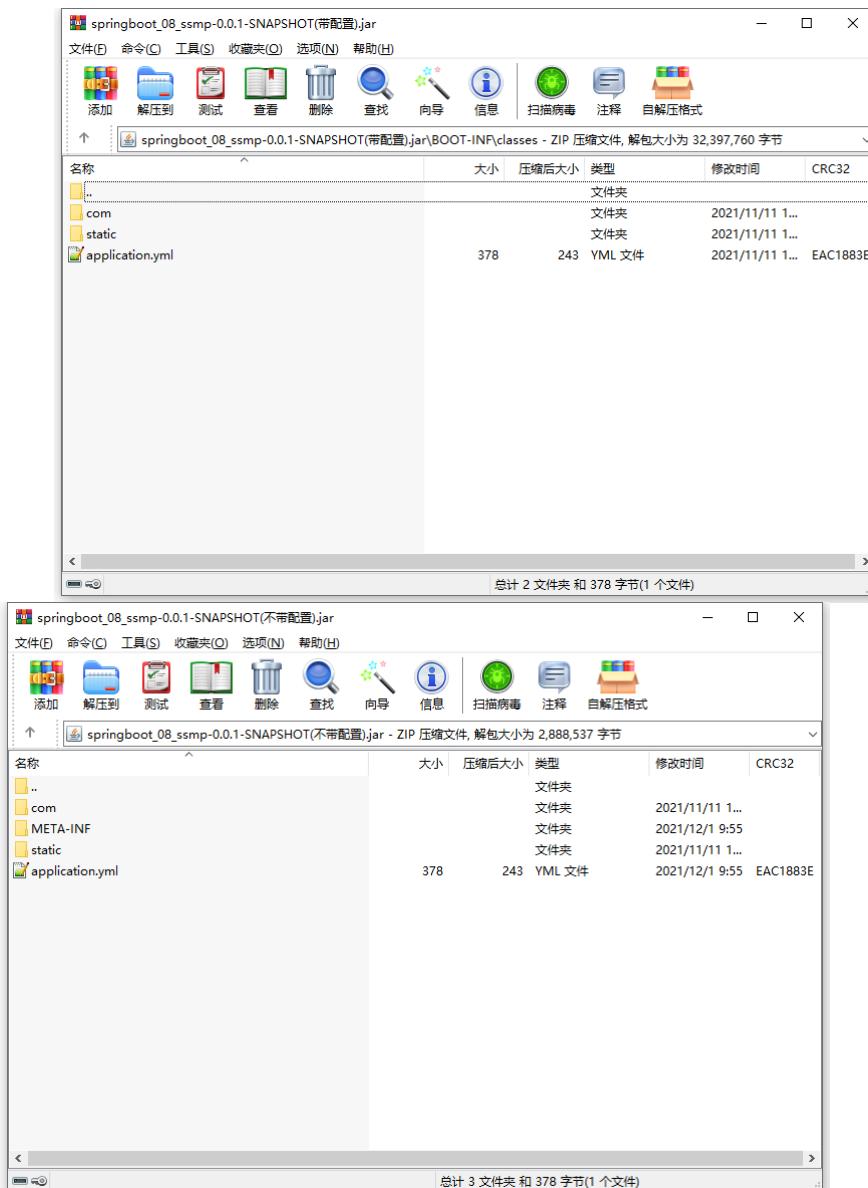
先看第一个现象，文件大小不同。带有配置时打包生成的程序包大小如下：

 springboot_08_ssmp-0.0.1-SNAPSHOT(不带配置).jar	1,056 KB
 springboot_08_ssmp-0.0.1-SNAPSHOT(带配置).jar	29,759 KB

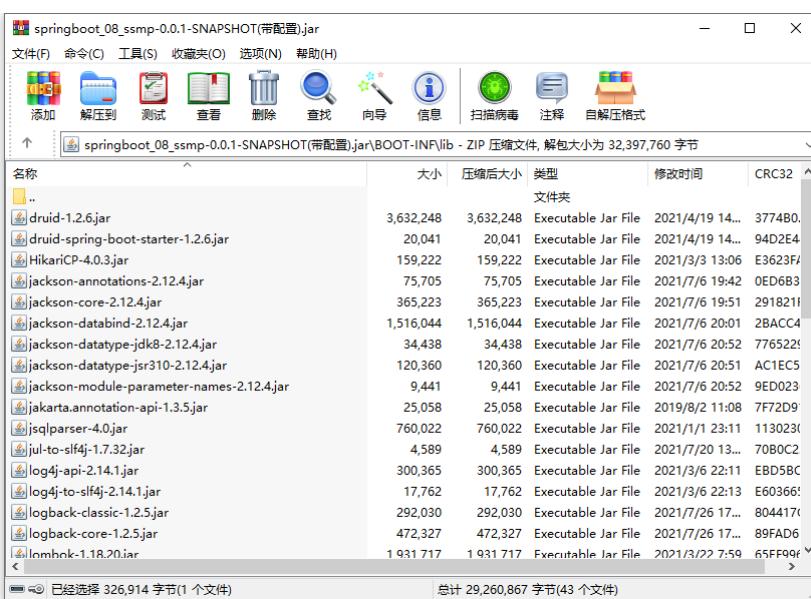
不难看出，带有配置的程序包体积比不带配置的大了30倍，那这里面都有什么呢？能差这么多？下面看看里面的内容有什么区别。



我们发现内容也完全不一样，仅有一个目录是一样的，叫做META-INF。打开容量大的程序包中的BOOT-INF目录下的classes目录，我们发现其中的内容居然和容量小的程序包中的内容完全一样。



原来大的程序包中除了包含小的程序包中的内容，还有别的东西。都有什么呢？回到BOOT-INF目录下，打开lib目录，里面显示了很多个jar文件。



仔细翻阅不难发现，这些jar文件都是我们制作这个工程时导入的坐标对应的文件。大概可以想明白了，SpringBoot程序为了让自己打包生成的程序可以独立运行，不仅将项目中自己开发的内容进行了打包，还把当前工程运行需要使用的jar包全部打包进来了。为什么这样做呢？就是为了可以独立运行。不依赖程序包外部的任何资源可以独立运行当前程序。这也是为什么大的程序包容量是小的程序包容量的30倍的主要原因。

再看看大程序包还有什么不同之处，在最外层目录包含一个org目录，进入此目录，目录名是org\springframework\boot\loader，在里面可以找到一个**JarLauncher.class**的文件，先记得这个文件。再看这套目录名，明显是一个Spring的目录名，为什么要把Spring框架的东西打包到这个程序包中呢？不清楚。

回到两个程序包的最外层目录，查看名称相同的文件夹META-INF下都有一个叫做MANIFEST.MF的文件，但是大小不同，打开文件，比较内容区别

- 小容量文件的MANIFEST.MF

```
Manifest-Version: 1.0
Implementation-Title: springboot_08_ssmp
Implementation-Version: 0.0.1-SNAPSHOT
Build-Jdk-Spec: 1.8
Created-By: Maven Jar Plugin 3.2.0
```

- 大容量文件的MANIFEST.MF

```
Manifest-Version: 1.0
Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx
Implementation-Title: springboot_08_ssmp
Implementation-Version: 0.0.1-SNAPSHOT
Spring-Boot-Layers-Index: BOOT-INF/layers.idx
Start-Class: com.itheima.SSMPApplication
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Build-Jdk-Spec: 1.8
Spring-Boot-Version: 2.5.4
Created-By: Maven Jar Plugin 3.2.0
Main-Class: org.springframework.boot.loader.JarLauncher
```

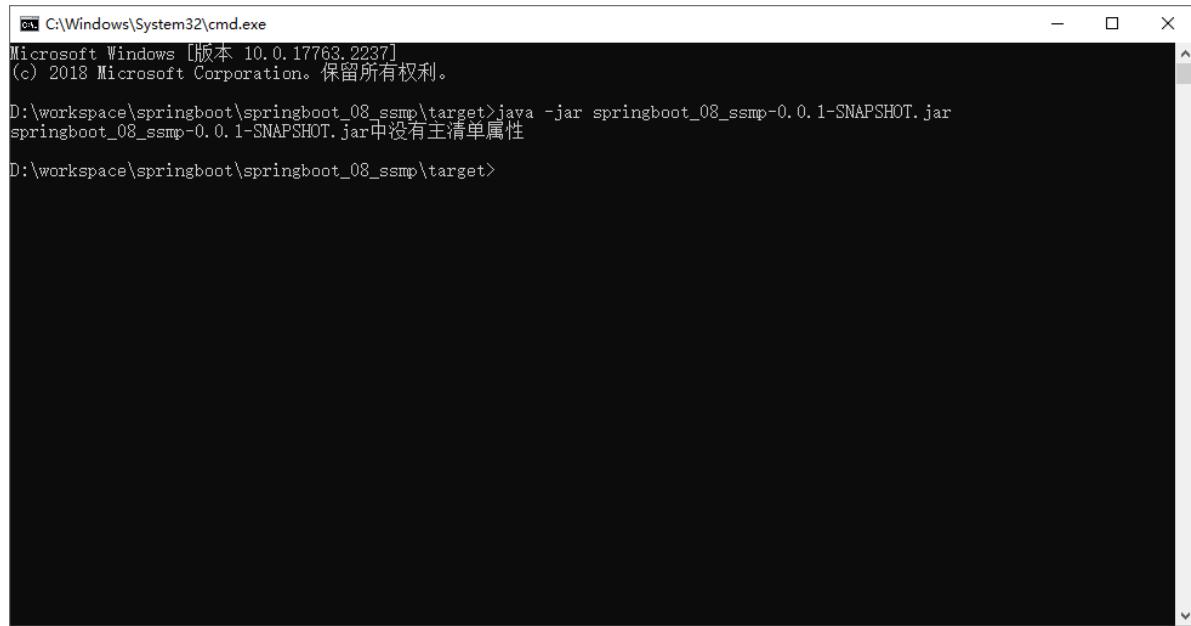
大文件中明显比小文件中多了几行信息，其中最后一行信息是Main-Class:
org.springframework.boot.loader.**JarLauncher**。这句话什么意思呢？如果使用java -jar执行此程序包，将执行Main-Class属性配置的类，这个类恰巧就是前面看到的那个文件。原来SpringBoot打包程序中出现Spring框架的东西是为这里服务的。而这个org.springframework.boot.loader.**JarLauncher**类内部要查找Start-Class属性中配置的类，并执行对应的类。这个属性在当前配置中也存在，对应的就是我们的引导类类名。

现在这组设定的作用就搞清楚了

1. SpringBoot程序添加配置后会打出一个特殊的包，包含Spring框架部分功能，原始工程内容，原始工程依赖的jar包
2. 首先读取MANIFEST.MF文件中的Main-Class属性，用来标记执行java -jar命令后运行的类
3. JarLauncher类执行时会找到Start-Class属性，也就是启动类类名
4. 运行启动类时会运行当前工程的内容
5. 运行当前工程时会使用依赖的jar包，从lib目录中查找

看来SpringBoot打出来了包为了能够独立运行，简直是煞费苦心，将所有需要使用的资源全部都添加到了这个包里。这就是为什么这个jar包能独立运行的原因。

再来看之前的报错信息：



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The window contains the following text:

```
Microsoft Windows [版本 10.0.17763.2237]
(c) 2018 Microsoft Corporation. 保留所有权利。

D:\workspace\springboot\springboot_08_ssmp\target>java -jar springboot_08_ssmp-0.0.1-SNAPSHOT.jar
springboot_08_ssmp-0.0.1-SNAPSHOT.jar中没有主清单属性

D:\workspace\springboot\springboot_08_ssmp\target>
```

由于打包时没有使用那段配置，结果打包后形成了一个普通的jar包，在MANIFEST.MF文件中也就没有了Main-Class对应的属性了，所以运行时提示找不到主清单属性，这就是报错的原因。

上述内容搞清楚对我们编程意义并不大，但是对各位小伙伴理清楚SpringBoot工程独立运行的机制是有帮助的。其实整体过程主要是带着大家分析，如果以后遇到了类似的问题，多给自己提问，多问一个为什么，兴趣自己就可以独立解决问题了。

总结

1. spring-boot-maven-plugin插件用于将当前程序打包成一个可以独立运行的程序包

命令行启动常见问题及解决方案

各位小伙伴在DOS环境下启动SpringBoot工程时，可能会遇到端口占用的问题。给大家一组命令，不用深入学习，备用吧。

```
# 查询端口
netstat -ano
# 查询指定端口
netstat -ano | findstr "端口号"
# 根据进程PID查询进程名称
tasklist | findstr "进程PID号"
# 根据PID杀死任务
taskkill /F /PID "进程PID号"
# 根据进程名称杀死任务
taskkill -f -t -im "进程名称"
```

关于打包与运行程序其实还有一系列的配置和参数，下面的内容中遇到再说，这里先开个头，知道如何打包和运行程序。

SpringBoot项目快速启动（Linux版）

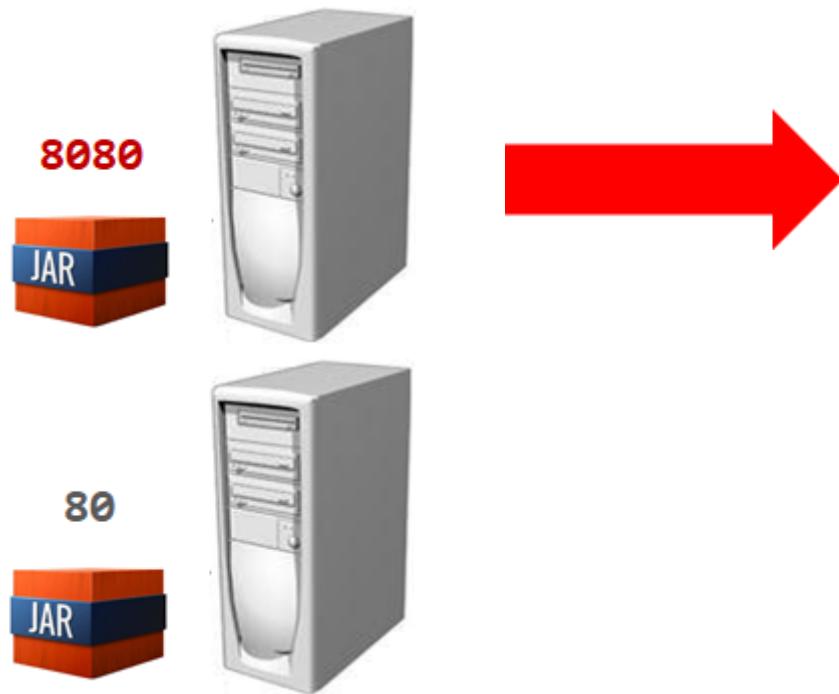
其实对于Linux系统下的程序运行与Windows系统下的程序运行差别不大，命令还是那组命令，只不过各位小伙伴可能对Linux指令不太熟悉，结果就会导致各种各样的问题发生。比如防火墙如何关闭，IP地址如何查询，JDK如何安装等等。这里不作为重点内容给大家普及了，了解一下整体过程就行了。

YW-2.配置高级

关于配置在基础篇讲过一部分，基础篇的配置总体上来说就是让各位小伙伴掌握配置的格式。比如配置文件如何写啊，写好的数据如何读取啊，都是基础的语法级知识。在实用篇中就要集中在配置的应用这个方面了，下面就开始配置高级相关内容的第一部分学习，为什么说第一部，因为在开发实用篇中还有对应的配置高级知识要进行学习。

YW-2-1.临时属性设置

目前我们的程序包打好了，可以发布了。但是程序包打好以后，里面的配置都已经是固定的了，比如配置了服务器的端口是8080。如果我要启动项目，发现当前我的服务器上已经有应用启动起来并且占用了8080端口，这个时候就尴尬了。难道要重新把打包好的程序修改一下吗？比如我要把打包好的程序启动端口改成80。



SpringBoot提供了灵活的配置方式，如果你发现你的项目中有个别属性需要重新配置，可以使用临时属性的方式快速修改某些配置。方法也特别简单，在启动的时候添加上对应参数就可以了。

```
java -jar springboot.jar --server.port=80
```

上面的命令是启动SpringBoot程序包的命令，在命令输入完毕后，空一格，然后输入两个-号。下面按照属性名=属性值的形式添加对应参数就可以了。记得，这里的格式不是yaml中的书写格式，当属性存在多级名称时，中间使用点分隔，和properties文件中的属性格式完全相同。

如果你发现要修改的属性不止一个，可以按照上述格式继续写，属性与属性之间使用空格分隔。

```
java -jar springboot.jar --server.port=80 --logging.level.root=debug
```

属性加载优先级

现在我们的程序配置受两个地方控制了，第一配置文件，第二临时属性。并且我们发现临时属性的加载优先级要高于配置文件的。那是否还有其他的配置方式呢？其实是有的，而且不少，打开官方文档中对应的内容，就可以查看配置读取的优先顺序。地址奉上：<https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-external-config>

1. Default properties (specified by setting `SpringApplication.setDefaultProperties()`).
2. `@PropertySource` annotations on your `@Configuration` classes. Please note that such property sources are not added to the `Environment` until the application context is being refreshed. This is too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.
3. Config data (such as `application.properties` files)
4. A `RandomValuePropertySource` that has properties only in `random.*`.
5. OS environment variables.
6. Java System properties (`System.getProperties()`).
7. JNDI attributes from `java:comp/env`.
8. `ServletContext` init parameters.
9. `ServletConfig` init parameters.
10. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. `properties` attribute on your tests. Available on `@SpringBootTest` and the test annotations for testing a particular slice of your application.
13. `@TestPropertySource` annotations on your tests.
14. Devtools global settings properties in the `$HOME/.config/spring-boot` directory when devtools is active.

我们可以看到，居然有14种配置的位置，而我们现在使用的是这里面的2个。第3条Config data说的就是使用配置文件，第11条Command line arguments说的就是使用命令行临时参数。而这14种配置的顺序就是SpringBoot加载配置的顺序，言外之意，命令行临时属性比配置文件的加载优先级高，所以这个列表上面的优先级低，下面的优先级高。其实这个东西不用背的，你就记得一点，你最终要什么效果，你自己是知道的，不管这个顺序是怎么个高低排序，开发时一定要配置成你要的顺序为准。这个顺序只是在你想不明白问题的时候帮助你分析罢了。

比如你现在加载了一个`user.name`属性。结果你发现出来的结果和你想的不一样，那肯定是别的优先级比你高的属性覆盖你的配置属性了，那你就可以看着这个顺序挨个排查。哪个位置有可能覆盖了你的属性。

我在课程评论区看到小伙伴学习基础篇的时候问这个问题了，就是这个原因造成的。在yaml中配置了`user.name`属性值，然后读取出来的时候居然不是自己的配置值，因为在系统属性中有一个属性叫做`user.name`，两个相互冲突了。而系统属性的加载优先顺序在上面这个列表中是5号，高于3号，所以SpringBoot最终会加载系统配置属性`user.name`。

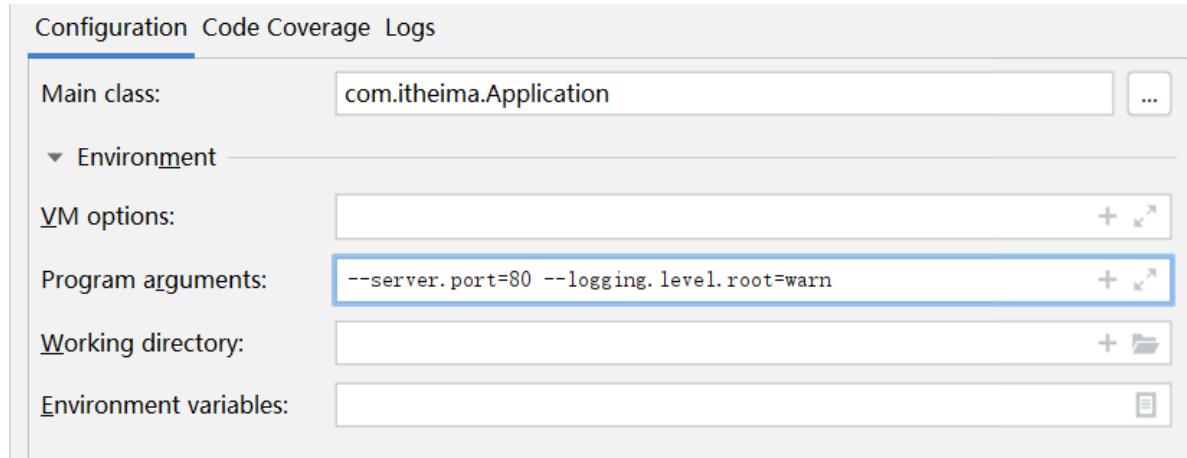
总结

1. 使用jar命令启动SpringBoot工程时可以使用临时属性替换配置文件中的属性
2. 临时属性添加方式：`java -jar 工程名.jar --属性名=值`
3. 多个临时属性之间使用空格分隔
4. 临时属性必须是当前boot工程支持的属性，否则设置无效

开发环境中使用临时属性

临时使用目前是有了，但是上线的时候通过命令行输入的临时属性必须是正确的啊，那这些属性配置值我们必须在开发环境中测试好才行。下面说一下开发环境中如何使用临时属性，其实就是在Idea界面下如何操作了。

打开SpringBoot引导类的运行界面，在里面找到配置项。其中Program arguments对应的位置就是添加临时属性的，可以加几个试试效果。



做到这里其实可以产生一个思考了，如果对java编程熟悉的小伙伴应该知道，我们运行main方法的时候，如果想使用main方法的参数，也就是下面的args参数，就是在上面这个位置添加的参数。

```
public static void main(String[] args) {  
}
```

原来是这样，通过这个args就可以获取到参数。再来看我们的引导类是如何书写的

```
public static void main(String[] args) {  
    SpringApplication.run(SSMPApplication.class, args);  
}
```

这个args参数居然传递给了run方法，看来在Idea中配置的临时参数就是通过这个位置传递到我们的程序中的。言外之意，这里如果不用这个args是不是就断开了外部传递临时属性的入口呢？是这样的，我们可以使用下面的调用方式，这样外部临时属性就无法进入到SpringBoot程序中了。

```
public static void main(String[] args) {  
    SpringApplication.run(SSMPApplication.class);  
}
```

或者还可以使用如下格式来玩这个操作，就是将配置不写在配置文件中，直接写成一个字符串数组，传递给程序入口。当然，这种做法并没有什么实际开发意义。

```
public static void main(String[] args) {  
    String[] arg = new String[1];  
    arg[0] = "--server.port=8082";  
    SpringApplication.run(SSMPApplication.class, arg);  
}
```

总结

- 启动SpringBoot程序时，可以选择是否使用命令行属性为SpringBoot程序传递启动属性

思考

现在使用临时属性可以在启动项目前临时更改配置了，但是新的问题又出来了。临时属性好用是好用，就是写的多了会很麻烦。比如我现在有个需求，上线的时候使用临时属性配置20个值，这下可麻烦了，能不能搞得简单点，集中管理一下呢？比如说搞个文件，加载指定文件？还真可以。怎么做呢？咱们下一节再说。

YW-2-2.配置文件分类

SpringBoot提供了配置文件和临时属性的方式来对程序进行配置。前面一直说的是临时属性，这一节要说说配置文件了。其实这个配置文件我们一直在使用，只不过我们用的是SpringBoot提供的4级配置文件中的其中一个级别。4个级别分别是：

- 类路径下配置文件（一直使用的是这个，也就是resources目录中的application.yml文件）
- 类路径下config目录下配置文件
- 程序包所在目录中配置文件
- 程序包所在目录中config目录下配置文件

类路径就是这种

名称	修改日期	类型	大小
.idea	2021/11/4 10:48	文件夹	
springboot_0x_xxxxxxxxxxxxxxxxxxxxxxx	2021/11/1 11:28	文件夹	
springboot_01_01_quickstart	2021/10/29 13:54	文件夹	
springboot_01_02_quickstart	2021/10/29 13:54	文件夹	
springboot_01_03_quickstart	2021/10/29 13:54	文件夹	
springboot_01_04_quickstart	2021/10/29 13:54	文件夹	
springboot_02_base_configuration	2021/10/29 13:54	文件夹	
springboot_03_yaml	2021/10/29 13:54	文件夹	
springboot_04_junit	2021/10/29 13:54	文件夹	
springboot_05_mybatis	2021/10/29 13:54	文件夹	
springboot_06_mybatis_plus	2021/10/29 13:54	文件夹	
springboot_07_druid	2021/10/29 13:54	文件夹	
springboot_08_ssmp	2021/11/4 10:53	文件夹	
application.yml	2021/11/4 10:56	YML 文件	1 KB
springboot_08_ssmp-0.0.1-SNAPSHOT...	2021/11/4 10:53	Executable Jar File	29,759 KB

程序包就是这种

springboot_05_mybatis	D:\workspace\springboot\springboot_05_mybatis
springboot_06_mybatis_plus	D:\workspace\springboot\springboot_06_mybatis_plus
springboot_07_druid	D:\workspace\springboot\springboot_07_druid
springboot_08_ssmp	D:\workspace\springboot\springboot_08_ssmp
SRC	
main	
java	
com	
itheima	
config	
controller	
dao	
domain	
service	
SSMPApplication	
resources	
config	
application.yml	
static	
templates	
application.yml	
test	
target	

好复杂，一个一个说。其实上述4种文件是提供给你了4种配置文件书写的位置，功能都是一样的，都是做配置的。那大家关心的就是差别了，没错，就是因为位置不同，产生了差异。总体上来说，4种配置文件如果都存在的话，有一个优先级的问题，说白了就是加入4个文件我都有，里面都有一样的配置，谁生效的问题。上面4个文件的加载优先顺序为

- file : config/application.yml 【最高】
- file : application.yml

3. classpath: config/application.yml
4. classpath: application.yml 【最低】

那为什么设计这种多种呢？说一个最典型的应用吧。

- 场景A：你作为一个开发者，你做程序的时候为了方便自己写代码，配置的数据库肯定是连接你自己本机的，咱们使用4这个级别，也就是之前一直用的application.yml。
- 场景B：现在项目开发到了一个阶段，要联调测试了，连接的数据库是测试服务器的数据库，肯定要换一组配置吧。你可以选择把你之前的文件中的内容都改了，目前还不麻烦。
- 场景C：测试完了，一切OK。你继续写你的代码，你发现你原来写的配置文件被改成测试服务器的内容了，你要再改回来。现在明白了不？场景B中把你的内容都改掉了，你现在要重新改回来，以后呢？改来改去吗？

解决方案很简单，用上面的3这个级别的配置文件就可以快速解决这个问题，再写一个配置就行了。两个配置文件共存，因为config目录中的配置加载优先级比你的高，所以配置项如果和级别4里面的内容相同就覆盖了，这样是不是很简单？

级别1和2什么时候使用呢？程序打包以后就要用这个级别了，管你程序里面配置写的是什么？我的级别高，可以轻松覆盖你，就不用考虑这些配置冲突的问题了。

总结

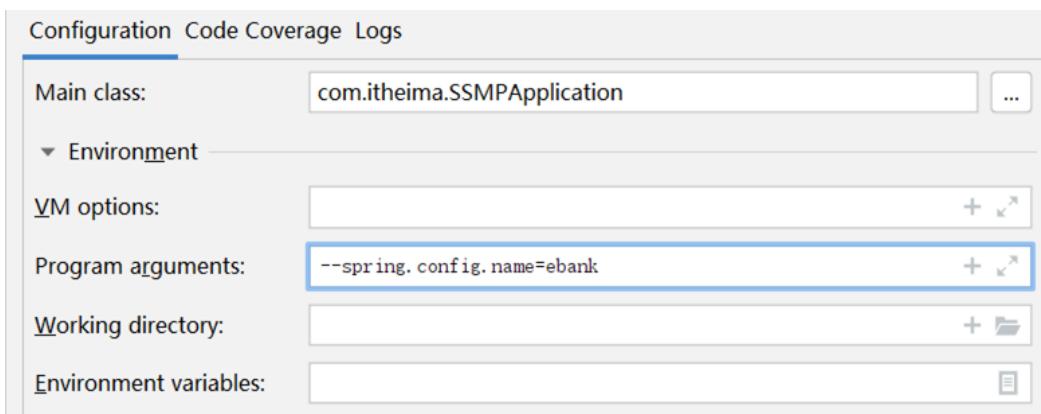
1. 配置文件分为4种
 - 项目类路径配置文件：服务于开发人员本机开发与测试
 - 项目类路径config目录中配置文件：服务于项目经理整体调控
 - 工程路径配置文件：服务于运维人员配置涉密线上环境
 - 工程路径config目录中配置文件：服务于运维经理整体调控
2. 多层级配置文件间的属性采用叠加并覆盖的形式作用于程序

YW-2-3.自定义配置文件

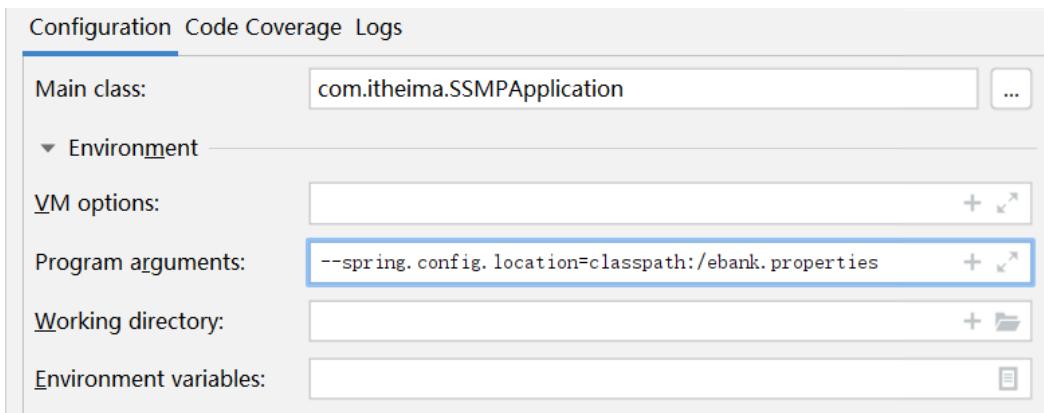
之前咱们做配置使用的配置文件都是application.yml，其实这个文件也是可以改名字的，这样方便维护。比如我2020年4月1日搞活动，走了一组配置，2020年5月1日活动取消，恢复原始配置，这个时候只需要重新更换一下配置文件就可以了。但是你总不能在原始配置文件上修改吧，不然做完活动以后，活动的配置就留不下来了，不利于维护。

自定义配置文件方式有如下两种：

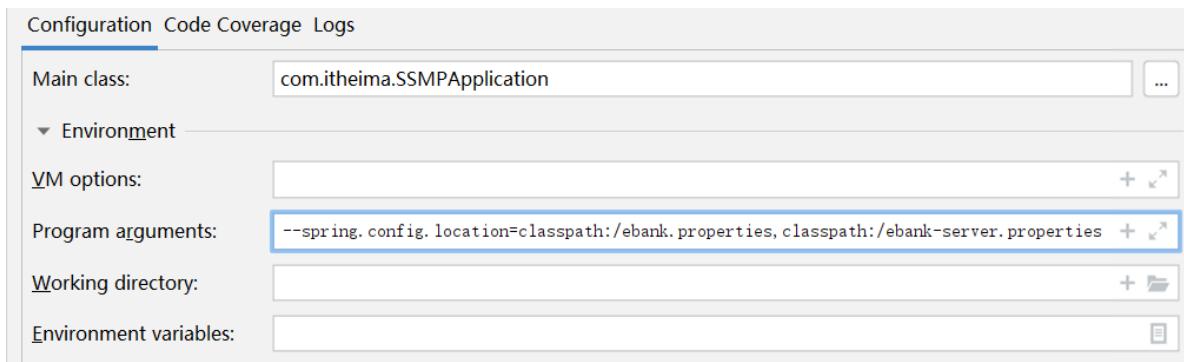
方式一：使用临时属性设置配置文件名，注意仅仅是名称，不要带扩展名



方式二：使用临时属性设置配置文件路径，这个是全路径名



也可以设置加载多个配置文件



使用的属性一个是spring.config.name，另一个是spring.config.location，这个一定要区别清楚。

温馨提示

我们现在研究的都是SpringBoot单体项目，就是单服务器版本。其实企业开发现在更多的是使用基于SpringCloud技术的多服务器项目。这种配置方式和我们现在学习的完全不一样，所有的服务器将不再设置自己的配置文件，而是通过配置中心获取配置，动态加载配置信息。为什么这样做？集中管理。这里不再说这些了，后面再讲这些东西。

总结

1. 配置文件可以修改名称，通过启动参数设定
2. 配置文件可以修改路径，通过启动参数设定
3. 微服务开发中配置文件通过配置中心进行设置

YW-3.多环境开发

讲的内容距离线上开发越来越近了，下面说一说多环境开发问题。

什么是多环境？其实就是说你的电脑上写的程序最终要放到别人的服务器上去运行。每个计算机环境不一样，这就是多环境。常见的多环境开发主要兼顾3种环境设置，开发环境——自己用的，测试环境——自己公司用的，生产环境——甲方爸爸用的。因为这是绝对不同的三台电脑，所以环境肯定有所不同，比如连接的数据库不一样，设置的访问端口不一样等等。



生产环境



开发环境



测试环境

```
jdbc:
  url: jdbc:mysql://21.49.35.241:3306/ccb
  user: ccb_admin
  password: 8Fm#_!@aSdm93]4k
```

```
jdbc:
  url: jdbc:mysql://127.0.0.1:4092/ccb_svms
  user: root
  password: root
```

```
jdbc:
  url: jdbc:mysql://21.49.27.66:4092/ccb_svms
  user: ccb_admin_test
  password: noBUGnoBugnoBUG
```

YW-3-1.多环境开发 (yaml单一文件版)

那什么是多环境开发？就是针对不同的环境设置不同的配置属性即可。比如你自己开发时，配置你的端口如下：

```
server:
  port: 80
```

如何想设计两组环境呢？中间使用三个减号分隔开

```
server:
  port: 80
---
server:
  port: 81
```

如何区分两种环境呢？起名字呗

```
spring:
  profiles: pro
server:
  port: 80
---
spring:
  profiles: dev
server:
  port: 81
```

那用哪一个呢？设置默认启动哪个就可以了

```
spring:
  profiles:
    active: pro      # 启动pro
---
spring:
  profiles: pro
server:
  port: 80
---
spring:
  profiles: dev
server:
  port: 81
```

就这么简单，再多来一组环境也OK

```
spring:
  profiles:
    active: pro      # 启动pro
---
spring:
  profiles: pro
server:
  port: 80
---
spring:
  profiles: dev
server:
  port: 81
---
spring:
  profiles: test
server:
  port: 82
```

其中关于环境名称定义上述格式是过时格式，标准格式如下

```
spring:
  config:
    activate:
      on-profile: pro
```

总结

1. 多环境开发需要设置若干种常用环境，例如开发、生产、测试环境
2. yaml格式中设置多环境使用---区分环境设置边界
3. 每种环境的区别在于加载的配置属性不同
4. 启用某种环境时需要指定启动时使用该环境

YW-3-2.多环境开发 (yaml多文件版)

将所有的配置都放在一个配置文件中，尤其是每一个配置应用场景都不一样，这显然不合理，于是就有了将一个配置文件拆分成多个配置文件的想法。拆分后，每个配置文件中写自己的配置，主配置文件中写清楚用哪一个配置文件就好了。

主配置文件

```
spring:  
  profiles:  
    active: pro      # 启动pro
```

application-pro.yaml

```
server:  
  port: 80
```

application-dev.yaml

```
server:  
  port: 81
```

文件的命名规则为：application-环境名.yml。

在配置文件中，如果某些配置项所有环境都一样，可以将这些项写入到主配置中，只有哪些有区别的项才写入到环境配置文件中。

- **主配置文件**中设置公共配置（全局）
- **环境分类配置文件**中常用于设置冲突属性（局部）

总结

1. 可以使用独立配置文件定义环境属性
2. 独立配置文件便于线上系统维护更新并保障系统安全性

YW-3-3.多环境开发 (properties多文件版)

SpringBoot最早期提供的配置文件格式是properties格式的，这种格式的多环境配置也了解一下吧。

主配置文件

```
spring.profiles.active=pro
```

环境配置文件：

application-pro.properties

```
server.port=80
```

application-dev.properties

```
server.port=81
```

文件的命名规则为：application-环境名.properties。

总结：

properties文件多环境配置仅支持多文件格式

YW-3-4.多环境开发独立配置文件书写技巧

作为程序员在搞配置的时候往往处于一种分久必合合久必分的局面。开始先写一起，后来为了方便维护就拆分。对于多环境开发也是如此，下面给大家说一下如何基于多环境开发做配置独立管理，务必掌握。

准备工作

将所有的配置根据功能对配置文件中的信息进行拆分，并制作成独立的配置文件，命名规则如下

- application-devDB.yml
- application-devRedis.yml
- application-devMVC.yml

使用

使用include属性在激活指定环境的情况下，同时对多个环境进行加载使其生效，多个环境间使用逗号分隔

```
spring:  
  profiles:  
    active: dev  
    include: devDB,devRedis,devMVC
```

比较一下，现在相当于加载dev配置时，再加载对应的3组配置，从结构上就很清晰，用了什么，对应的名称是什么

注意

当主环境dev与其他环境有相同属性时，主环境属性生效；其他环境中有相同属性时，最后加载的环境属性生效

改良

但是上面的设置也有一个问题，比如我要切换dev环境为pro时，include也要修改。因为include属性只能使用一次，这就比较麻烦了。SpringBoot从2.4版开始使用group属性替代include属性，降低了配置书写量。简单说就是我先写好，你爱用哪个用哪个。

```
spring:  
  profiles:  
    active: dev  
    group:  
      "dev": devDB,devRedis,devMVC  
      "pro": proDB,proRedis,proMVC  
      "test": testDB,testRedis,testMVC
```

现在再来看，如果切换dev到pro，只需要改一下是不是就结束了？完美！

总结

1. 多环境开发使用 group属性设置配置文件分组，便于线上维护管理

YW-3-5.多环境开发控制

多环境开发到这里基本上说完了，最后说一个冲突问题。就是maven和SpringBoot同时设置多环境的话怎么搞。

要想处理这个冲突问题，你要先理清一个关系，究竟谁在多环境开发中其主导地位。也就是说如果现在都设置了多环境，谁的应该是保留下来的，另一个应该遵从相同的设置。

maven是做什么的？项目构建管理的，最终生成代码包的，SpringBoot是什么的？简化开发的。简化，又不是其主导作用。最终还是要靠maven来管理整个工程，所以SpringBoot应该听maven的。整个确认后下面就好做了。大体思想如下：

- 先在maven环境中设置用什么具体的环境
- 在SpringBoot中读取maven设置的环境即可

maven中设置多环境（使用属性方式区分环境）

```
<profiles>
  <profile>
    <id>env_dev</id>
    <properties>
      <profile.active>dev</profile.active>
    </properties>
    <activation>
      <activeByDefault>true</activeByDefault>      <!--默认启动环境-->
    </activation>
  </profile>
  <profile>
    <id>env_pro</id>
    <properties>
      <profile.active>pro</profile.active>
    </properties>
  </profile>
</profiles>
```

SpringBoot中读取maven设置值

```
spring:
  profiles:
    active: @profile.active@
```

上面的@属性名@就是读取maven中配置的属性值的语法格式。

总结

1. 当Maven与SpringBoot同时对多环境进行控制时，以Maven为主，SpringBoot使用@..@占位符读取Maven对应的配置属性值
2. 基于SpringBoot读取Maven配置属性的前提下，如果在Idea下测试工程时pom.xml每次更新需要手动compile方可生效

YW-4.日志

运维篇最后一部分我们来聊聊日志，日志大家不陌生，简单介绍一下。日志其实就是记录程序日常运行的信息，主要作用如下：

- 编程期调试代码
- 运营期记录信息
- 记录日常运营重要信息（峰值流量、平均响应时长……）
- 记录应用报错信息（错误堆栈）
- 记录运维过程数据（扩容、宕机、报警……）

或许各位小伙伴并不习惯于使用日志，没关系，慢慢多用，习惯就好。想进大厂，这是最基本的，别去面试的时候说没用过，完了，没机会了。

YW-4-1.代码中使用日志工具记录日志

日志的使用格式非常固定，直接上操作步骤：

步骤①：添加日志记录操作

```
@RestController
@RequestMapping("/books")
public class BookController extends BaseClass{
    private static final Logger log =
LoggerFactory.getLogger(BookController.class);
    @GetMapping
    public String getById(){
        log.debug("debug...");
        log.info("info...");
        log.warn("warn...");
        log.error("error...");
        return "springboot is running...2";
    }
}
```

上述代码中log对象就是用来记录日志的对象，下面的log.debug, log.info这些操作就是写日志的API了。

步骤②：设置日志输出级别

日志设置好以后可以根据设置选择哪些参与记录。这里是根据日志的级别来设置的。日志的级别分为6种，分别是：

- TRACE：运行堆栈信息，使用率低
- DEBUG：程序员调试代码使用
- INFO：记录运维过程数据
- WARN：记录运维过程报警数据
- ERROR：记录错误堆栈信息
- FATAL：灾难信息，合并计入ERROR

一般情况下，开发时候使用DEBUG，上线后使用INFO，运维信息记录使用WARN即可。下面就设置一下日志级别：

```
# 开启debug模式，输出调试信息，常用于检查系统运行状况
debug: true
```

这么设置太简单粗暴了，日志系统通常都提供了细粒度的控制

```
# 开启debug模式，输出调试信息，常用于检查系统运行状况
debug: true
```

```
# 设置日志级别，root表示根节点，即整体应用日志级别
logging:
  level:
    root: debug
```

还可以再设置更细粒度的控制

步骤③：设置日志组，控制**指定包**对应的日志输出级别，也可以直接控制**指定包**对应的日志输出级别

```
logging:
  # 设置日志组
  group:
    # 自定义组名，设置当前组中所包含的包
    ebank: com.itheima.controller,com.itheima.service...等等包
  level:
    # 为对应组设置日志级别
    ebank: debug
    # 为对包设置日志级别
    com.itheima.controller: debug
```

说白了就是总体设置一下，每个包设置一下，如果感觉设置的麻烦，就先把包分个组，对组设置，没了，就这些。

总结

1. 日志用于记录开发调试与运维过程消息
2. 日志的级别共6种，通常使用4种即可，分别是DEBUG, INFO, WARN, ERROR
3. 可以通过日志组或代码包的形式进行日志显示级别的控制

教你一招：优化日志对象创建代码

写代码的时候每个类都要写创建日志记录对象，这个可以优化一下，使用前面用过的lombok技术给我们提供的工具类即可。

```
@RestController
@RequestMapping("/books")
public class BookController extends BaseClass{
    private static final Logger log =
    LoggerFactory.getLogger(BookController.class); //这一句可以不写了
}
```

导入lombok后使用注解搞定，日志对象名为log

```

@Slf4j      //这个注解替代了下面那一行
@RestController
@RequestMapping("/books")
public class BookController {
    private static final Logger log =
        LoggerFactory.getLogger(BookController.class); //这一句可以不写了
}

```

总结

1. 基于lombok提供的@Slf4j注解为类快速添加日志对象

YW-4-2.日志输出格式控制

日志已经能够记录了，但是目前记录的格式是SpringBoot给我们提供的，如果想自定义控制就需要自己设置了。先分析一下当前日志的记录格式。

```

2021-11-02 12:25:39.392 INFO 2336 --- [           main] com.itheima.springboot10LogApplication : Starting Springboot10LogApplication using Java 1.8.0_172 or
2021-11-02 12:25:39.395 INFO 2336 --- [           main] com.itheima.springboot10LogApplication : No active profile set, falling back to default profiles: de
2021-11-02 12:25:40.065 INFO 2336 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-11-02 12:25:40.071 INFO 2336 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-02 12:25:40.071 INFO 2336 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.54]
2021-11-02 12:25:40.113 INFO 2336 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-02 12:25:40.113 INFO 2336 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 671
2021-11-02 12:25:40.326 INFO 2336 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-02 12:25:40.334 INFO 2336 --- [           main] com.itheima.springboot10LogApplication : Started Springboot10LogApplication in 1.281 seconds (JVM ru

```

时间 级别 PID 所属线程 所属类/接口名 日志信息

对于单条日志信息来说，日期，触发位置，记录信息是最核心的信息。级别用于做筛选过滤，PID与线程名用于做精准分析。了解这些信息后就可以DIY日志格式了。本课程不做详细的研究，有兴趣的小伙伴可以学习相关的知识。下面给出课程中模拟的官方日志模板的书写格式，便于大家学习。

```

logging:
  pattern:
    console: "%d %clr(%p) --- [%16t] %clr(%-40.40c){cyan} : %m %n"

```

总结

1. 日志输出格式设置规则

YW-4-3.日志文件

日志信息显示，记录已经控制住了，下面就要说一下日志的转存了。日志不能仅显示在控制台上，要把日志记录到文件中，方便后期维护查阅。

对于日志文件的使用存在各种各样的策略，例如每日记录，分类记录，报警后记录等。这里主要研究日志文件如何记录。

记录日志到文件中格式非常简单，设置日志文件名即可。

```

logging:
  file:
    name: server.log

```

虽然使用上述格式可以将日志记录下来了，但是面对线上的复杂情况，一个文件记录肯定是不能够满足运维要求的，通常会每天记录日志文件，同时为了便于维护，还要限制每个日志文件的大小。下面给出日志文件的常用配置方式：

```
logging:  
  logback:  
    rollingpolicy:  
      max-file-size: 3KB  
      file-name-pattern: server.%d{yyyy-MM-dd}.%i.log
```

以上格式是基于logback日志技术设置每日日志文件的设置格式，要求容量到达3KB以后就转存信息到第二个文件中。文件命名规则中的%d标识日期，%i是一个递增变量，用于区分日志文件，这个变量不用我们自己定义！。

总结

1. 日志记录到文件
2. 日志文件名称格式设置

运维实用篇完结

运维实用篇到这里就要先告一段落了，为什么不说结束呢？因为运维篇中还有一些知识，但是现在讲解过于分散了。所以要把这些知识与开发实用篇的知识结合在一起讲，也是本课程的教学设计的体现。

在整体运维实用篇中带着大家学习了4块内容，首先学习了如何运行SpringBoot程序，也就是程序的打包与运行，接下来对配置进行了升级学习，不再局限在配置文件中进行设置，通过临时属性，外部配置文件对项目的配置进行管控。在多环境开发中给大家介绍了多种多环境开发的格式，其实掌握一种即可，此外还给大家讲了多环境开发的一些技巧以及与maven的冲突解决方案。最后给大家介绍了日志系统，老实说日志这里讲的相当的潦草，因为大部分日志相关的知识都不应该在这门课中学习，这里只是告诉大家如何整合实用而已。

看了各位小伙伴的评论，知道你们再催更，我也在加油，一起努力吧，实用开发篇再会。实用开发篇会提高更新频度，不全部做完给大家更新了，我先把做好的一部分开放出来，随后做完一点就更新一点，额，好吧，就说到这里吧。

SpringBoot开发实用篇

怀着忐忑的心情，开始了开发实用篇文档的编写。为什么忐忑？特喵的债欠的太多，不知道从何写起。哎，不煽情了，开工。

运维实用篇完结以后，开发实用篇采用日更新的形式发布给各位小伙伴，基本上是每天一集，目前已经发布完毕。看评论区，好多小伙伴在求文档，所以赶紧来补文档，加班加点把开发实用篇的文档创出来。

开发实用篇中因为牵扯到SpringBoot整合各种各样的技术，由于不是每个小伙伴对各种技术都有所掌握，所以在整合每一个技术之前，都会做一个快速的普及，这样的话内容整个开发实用篇所包含的内容就会比较多。各位小伙伴在学习的时候，如果对某一个技术不是很清楚，可以先跳过对应章节，或者先补充一下技术知识，然后再来看对应的课程。开发实用篇具体包含的内容如下：

- 热部署
- 配置高级
- 测试
- 数据层解决方案
- 整合第三方技术
- 监控

看目录感觉内容量并不是很大，但是在数据层解决方案和整合第三方技术中包含了大量的知识，一点一点慢慢学吧。下面开启第一部分热部署相关知识的学习

KF-1.热部署

什么是热部署？简单说就是你程序改了，现在要重新启动服务器，嫌麻烦？不用重启，服务器会自己悄悄的把更新后的程序给重新加载一遍，这就是热部署。

热部署的功能是如何实现的呢？这就要分两种情况来说了，非springboot工程和springboot工程的热部署实现方式完全不一样。先说一下原始的非springboot项目是如何实现热部署的。

非springboot项目热部署实现原理

开发非springboot项目时，我们要制作一个web工程并通过tomcat启动，通常需要先安装tomcat服务器到磁盘中，开发的程序配置发布到安装的tomcat服务器上。如果想实现热部署的效果，这种情况其实有两种做法，一种是在tomcat服务器的配置文件中进行配置，这种做法与你使用什么IDE工具无关，不管你使用eclipse还是idea都行。还有一种做法是通过IDE工具进行配置，比如在idea工具中进行设置，这种形式需要依赖IDE工具，每款IDE工具不同，对应的配置也不太一样。但是核心思想是一样的，就是使用服务器去监控其中加载的应用，发现产生了变化就重新加载一次。

上面所说的非springboot项目实现热部署看上去是一个非常简单的过程，几乎每个小伙伴都能自己写出来。如果你不会写，我给你个最简单的思路，但是实际设计要比这复杂一些。例如启动一个定时任务，任务启动时记录每个文件的大小，以后每5秒比对一下每个文件的大小是否有改变，或者是否有新文件。如果没有改变，放行，如果有改变，刷新当前记录的文件信息，然后重新启动服务器，这就可以实现热部署了。当然，这个过程肯定不能这么做，比如我把一个打印输出的字符串"abc"改成"cba"，比对大小是没有变化的，但是内容确实变了，所以这么做肯定不行，只是给大家打个比方，而且重启服务器这就是冷启动了，不能算热部署，领会精神吧。

看上去这个过程也没多复杂，在springboot项目中难道还有其他的弯弯绕吗？还真有。

springboot项目热部署实现原理

基于springboot开发的web工程其实有一个显著的特征，就是tomcat服务器内置了，还记得内嵌服务器吗？服务器是以一个对象的形式在spring容器中运行的。本来我们期望于tomcat服务器加载程序后由tomcat服务器盯着程序，你变化后我就重新启动重新加载，但是现在tomcat和我们的程序是平级的了，都是spring容器中的组件，这下就麻烦了，缺乏了一个直接的管理权，那该怎么做呢？简单，再搞一个程序X在spring容器中盯着你原始开发的程序A不就行了吗？确实，搞一个盯着程序A的程序X就行了，如果你自己开发的程序A变化了，那么程序X就命令tomcat容器重新加载程序A就OK了。并且这样做有一个好处，spring容器中东西不用全部重新加载一遍，只需要重新加载你开发的程序那一部分就可以了，这下效率又高了，挺好。

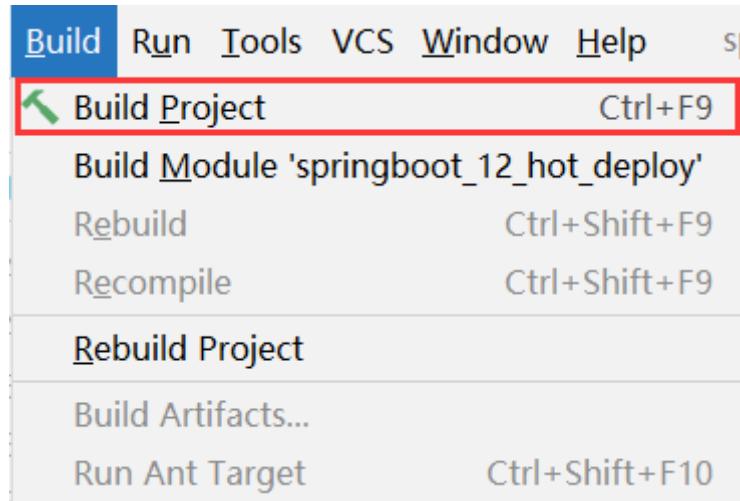
下面就说说，怎么搞出来这么一个程序X，肯定不是我们自己手写了，springboot早就做好了，搞一个坐标导入进去就行了。

KF-1-1.手动启动热部署

步骤①：导入开发者工具对应的坐标

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

步骤②：构建项目，可以使用快捷键激活此功能（每次修改都需要重新build项目）



对应的快捷键一定要记得

<CTR>L+<F9>

以上过程就实现了springboot工程的热部署，是不是挺简单的。不过这里需要把底层的工作工程给普及一下。

重启与重载

一个springboot项目在运行时实际上是分两个过程进行的，根据加载的东西不同，划分成base类加载器与restart类加载器。

- base类加载器：用来加载jar包中的类，jar包中的类和配置文件由于不会发生变化，因此不管加载多少次，加载的内容不会发生变化（reload重载）
- restart类加载器：用来加载开发者自己开发的类、配置文件、页面等信息，这一类文件受开发者影响（restart重启）

当springboot项目启动时（既有restart又有reload），**base类加载器**执行，加载jar包中的信息后，**restart类加载器**执行，加载开发者制作的内容。

当执行构建项目后，由于jar中的信息不会变化，因此base类加载器无需再次执行，所以仅仅运行restart类加载即可，也就是将开发者自己制作的内容重新加载就行了，这就完成了一次热部署的过程，也可以说热部署的过程实际上是重新加载restart类加载器中的信息。

总结

1. 使用开发者工具可以为当前项目开启热部署功能
2. 使用构建项目操作对工程进行热部署

思考

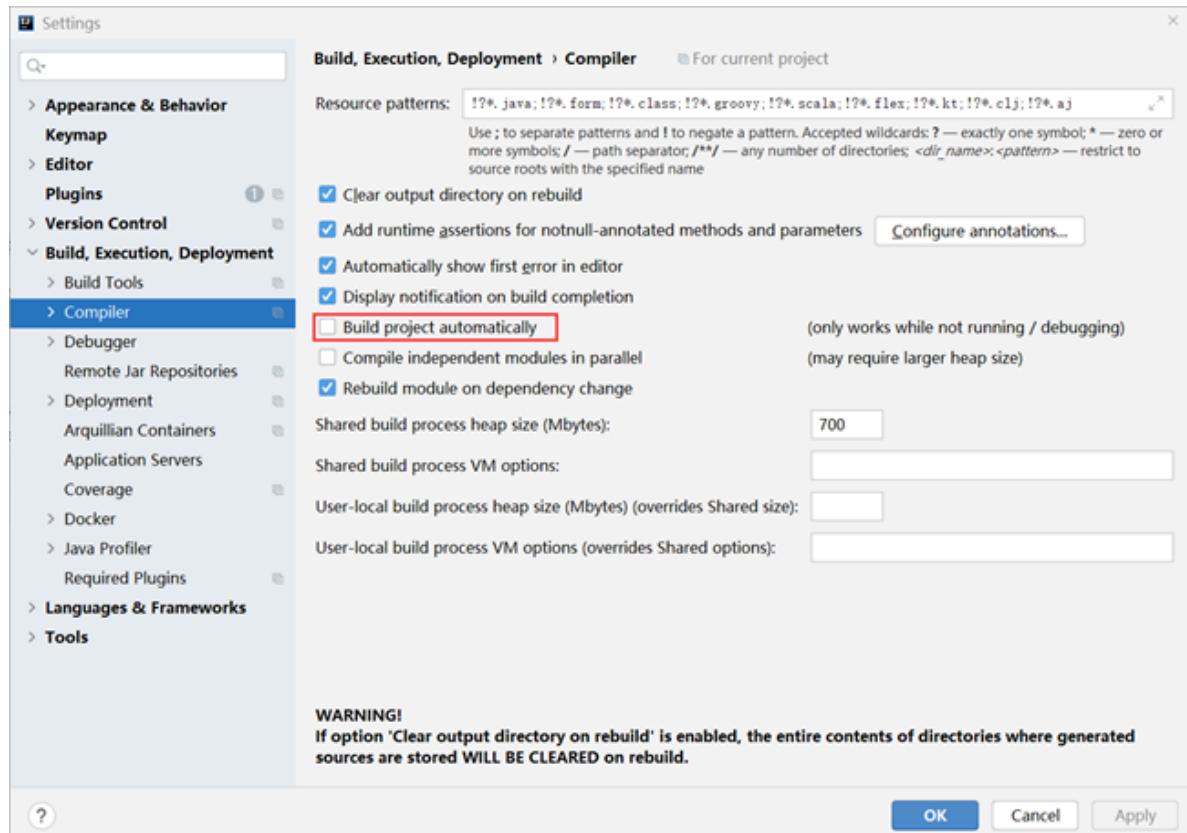
上述过程每次进行热部署都需要开发者手工操作，不管是点击按钮还是快捷键都需要开发者手工执行。这种操作的应用场景主要是在开发调试期，并且调试的代码处于不同的文件中，比如服务器启动了，我需要改4个文件中的内容，然后重启，等4个文件都改完了再执行热部署，使用一个快捷键就OK了。但是如果现在开发者要修改的内容就只有一个文件中的少量代码，这个时候代码修改完毕如果能够让程序自己执行热部署功能，就可以减少开发者的操作，也就是自动进行热部署，能这么做吗？是可以的。咱们下一节再说。

KF-1-2. 自动启动热部署

自动热部署其实就是一个开关，打开这个开关后，IDE工具就可以自动热部署。因此这个操作和IDE工具有关，以下以idea为例设置idea中启动热部署

步骤①：设置自动构建项目

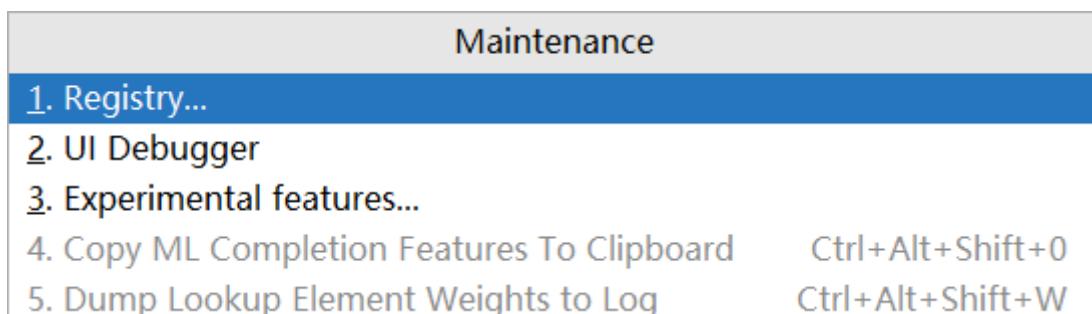
打开【File】，选择【settings...】，在面板左侧的菜单中找到【Compile】选项，然后勾选【Build project automatically】，意思是自动构建项目



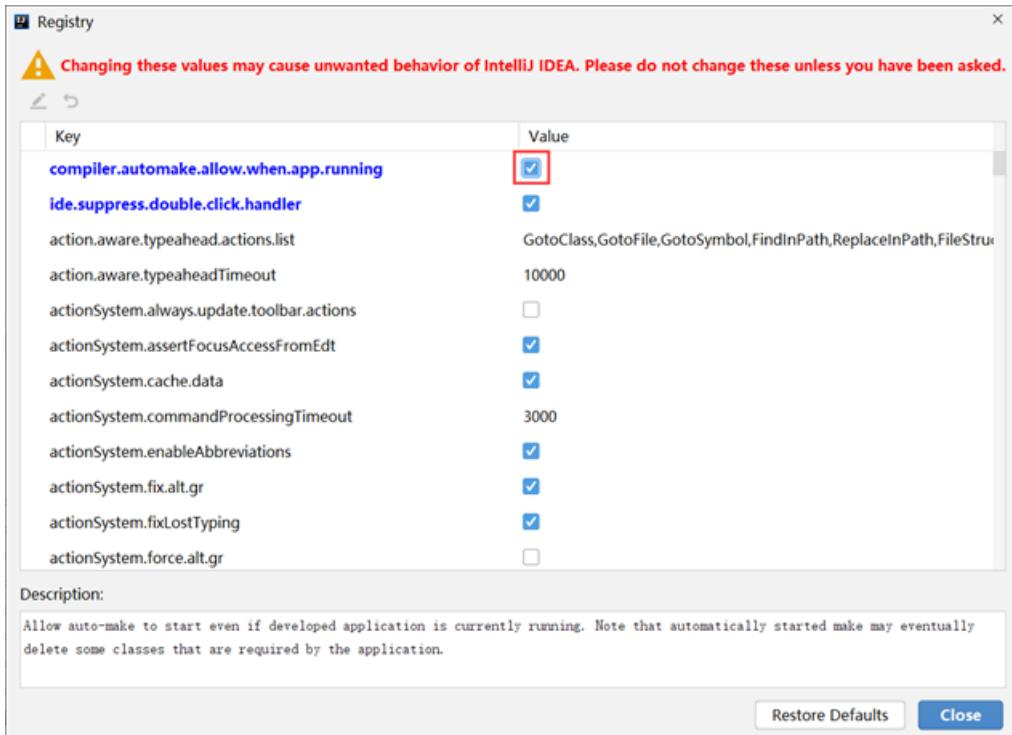
自动构建项目选项勾选后

步骤②：允许在程序运行时进行自动构建

使用快捷键【Ctrl】+【Alt】+【Shift】+【/】打开维护面板，选择第1项【Registry...】



在选项中搜索complete，然后勾选对应项即可



这样程序在运行的时候就可以进行自动构建了，实现了热部署的效果。

关注：如果你每敲一个字母，服务器就重新构建一次，这未免有点太频繁了，所以idea设置当**idea工具失去焦点5秒后进行热部署**（这个思想好nb，因为失去焦点说明你去做测试了）。其实质是你从idea工具中切换到其他工具时进行热部署，比如改完程序需要到浏览器上去调试，这个时候idea就自动进行热部署操作。

总结

1. 自动热部署要开启自动构建项目
2. 自动热部署要开启在程序运行时自动构建项目

思考

现在已经实现了热部署了，但是到企业开发的时候你会发现，为了便于管理，在你的程序目录中除了有代码，还有可能有文档，如果你修改了一下文档，这个时候会进行热部署吗？不管是否进行热部署，这个过程我们需要自己控制才比较合理，那这个东西能控制吗？咱们下一节再说。

KF-1-3. 参与热部署监控的文件范围配置

通过修改项目中的文件，你可以发现其实并不是所有的文件修改都会激活热部署的，原因在于在开发者工具中有一组配置，当满足了配置中的条件后，才会启动热部署，配置中默认不参与热部署的目录信息如下

- /META-INF/maven
- /META-INF/resources
- /resources
- /static
- /public
- /templates

以上目录中的文件如果发生变化，是不参与热部署的。如果想修改配置，可以通过application.yml文件进行设定哪些文件不参与热部署操作

```
spring:  
  devtools:  
    restart:  
      # 设置不参与热部署的文件或文件夹  
      exclude: static/**,public/**,config/application.yml
```

总结

1. 通过配置可以修改不参与热部署的文件或目录

思考

热部署功能是一个典型的开发阶段使用的功能，到了线上环境运行程序时，这个功能就没有意义了。能否关闭热部署功能呢？咱们下一节再说。

KF-1-4.关闭热部署

线上环境运行时是不可能使用热部署功能的，所以需要强制关闭此功能，通过配置可以关闭此功能。

```
spring:  
  devtools:  
    restart:  
      enabled: false
```

如果当心配置文件层级过多导致相符覆盖最终引起配置失效，可以提高配置的层级，在更高层级中配置关闭热部署。例如在启动容器前通过系统属性设置关闭热部署功能。

```
@SpringBootApplication  
public class SSMPApplication {  
    public static void main(String[] args) {  
        System.setProperty("spring.devtools.restart.enabled","false");  
        SpringApplication.run(SSMPApplication.class);  
    }  
}
```

其实上述担心略微有点多余，因为线上环境的维护是不可能出现修改代码的操作的，这么做唯一的作用是降低资源消耗，毕竟那双盯着你项目是不是产生变化的眼睛只要闭上了，就不具有热部署功能了，这个开关的作用就是禁用对应功能。

总结

1. 通过配置可以关闭热部署功能降低线上程序的资源消耗

KF-2.配置高级

进入开发实用篇第二章内容，配置高级，其实配置在基础篇讲了一部分，在运维实用篇讲了一部分，这里还要讲，讲的东西有什么区别呢？距离开发过程越来越接近，解决的问题也越来越靠近线上环境，下面就开启本章的学习。

KF-2-1.@ConfigurationProperties

在基础篇学习了@ConfigurationProperties注解，此注解的作用是用来为bean绑定属性的。开发者可以在yml配置文件中以对象的格式添加若干属性

```
servers:  
  ip-address: 192.168.0.1  
  port: 2345  
  timeout: -1
```

然后再开发一个用来封装数据的实体类，注意要提供属性对应的setter方法

```
@Component  
@Data  
public class ServerConfig {  
    private String ipAddress;  
    private int port;  
    private long timeout;  
}
```

使用@ConfigurationProperties注解就可以将配置中的属性值关联到开发的模型类上

```
@Component  
@Data  
@ConfigurationProperties(prefix = "servers")  
public class ServerConfig {  
    private String ipAddress;  
    private int port;  
    private long timeout;  
}
```

这样加载对应bean的时候就可以直接加载配置属性值了。但是目前我们学的都是给自定义的bean使用这种形式加载属性值，如果是第三方的bean呢？能不能用这种形式加载属性值呢？为什么会提出这个疑问？原因就在于当前@ConfigurationProperties注解是写在类定义的上方，而第三方开发的bean源代码不是你自己书写的，你也不可能到源代码中去添加@ConfigurationProperties注解，这种问题该怎么解决呢？下面就来说说这个问题。

使用@ConfigurationProperties注解其实可以为第三方bean加载属性，格式特殊一点而已。

步骤①： 使用@Bean注解定义第三方bean

```
@Bean  
public DruidDataSource datasource(){  
    DruidDataSource ds = new DruidDataSource();  
    return ds;  
}
```

步骤②： 在yml中定义要绑定的属性，注意datasource此时全小写

```
datasource:  
  driverClassName: com.mysql.jdbc.Driver
```

步骤③：使用@ConfigurationProperties注解为第三方bean进行属性绑定，注意前缀是全小写的datasource

```
@Bean  
@ConfigurationProperties(prefix = "datasource")  
public DruidDataSource datasource(){  
    DruidDataSource ds = new DruidDataSource();  
    return ds;  
}
```

操作方式完全一样，只不过@ConfigurationProperties注解不仅能添加到类上，还可以添加到方法上，添加到类上是为spring容器管理的当前类的对象绑定属性，添加到方法上是为spring容器管理的当前方法的返回值对象绑定属性，其实本质上都一样。

做到这其实就出现了一个新的问题，目前我们定义bean不是通过类注解定义就是通过@Bean定义，使用@ConfigurationProperties注解可以为bean进行属性绑定，那在一个业务系统中，哪些bean通过注解@ConfigurationProperties去绑定属性了呢？因为这个注解不仅可以写在类上，还可以写在方法上，所以找起来就比较麻烦了。为了解决这个问题，spring给我们提供了一个全新的注解，专门标注使用@ConfigurationProperties注解绑定属性的bean是哪些。这个注解叫做@EnableConfigurationProperties。具体如何使用呢？

步骤①：在配置类上开启@EnableConfigurationProperties注解，并标注要使用@ConfigurationProperties注解绑定属性的类

```
@SpringBootApplication  
@EnableConfigurationProperties(ServerConfig.class)  
public class Springboot13ConfigurationApplication {  
}
```

步骤②：在对应的类上直接使用@ConfigurationProperties进行属性绑定

```
@Data  
@ConfigurationProperties(prefix = "servers")  
public class ServerConfig {  
    private String ipAddress;  
    private int port;  
    private long timeout;  
}
```

有人感觉这没区别啊？注意观察，现在绑定属性的ServerConfig类并没有声明@Component注解。当使用@EnableConfigurationProperties注解时，spring会默认将其标注的类定义为bean，因此无需再次声明@Component注解了。

最后再说一个小技巧，使用@ConfigurationProperties注解时，会出现一个提示信息

 Spring Boot Configuration Annotation Processor not configured [Open Documentation...](#) 

出现这个提示后只需要添加一个坐标此提醒就消失了

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

总结

1. 使用@ConfigurationProperties可以为使用@Bean声明的第三方bean绑定属性
2. 当使用@EnableConfigurationProperties声明进行属性绑定的bean后，无需使用@Component注解再次进行bean声明

KF-2-2. 宽松绑定/松散绑定

在进行属性绑定时，可能会遇到如下情况，为了进行标准命名，开发者会将属性名严格按照驼峰命名法书写，在yml配置文件中将datasource修改为dataSource，如下：

```
dataSource:  
  driverClassName: com.mysql.jdbc.Driver
```

此时程序可以正常运行，然后又将代码中的前缀datasource修改为dataSource，如下：

```
@Bean  
@ConfigurationProperties(prefix = "dataSource")  
public DruidDataSource datasource(){  
    DruidDataSource ds = new DruidDataSource();  
    return ds;  
}
```

此时就发生了编译错误，而且并不是idea工具导致的，运行后依然会出现问题，配置属性名dataSource是无效的

```
Configuration property name 'dataSource' is not valid:  
  
  Invalid characters: 's'  
  Bean: datasource  
  Reason: Canonical names should be kebab-case ('-' separated), lowercase  
alpha-numeric characters and must start with a letter  
  
  Action:  
  Modify 'dataSource' so that it conforms to the canonical names requirements.
```

为什么会出现这种问题，这就要来说一说springboot进行属性绑定时的一个重要知识点了，有关属性名称的宽松绑定，也可以称为宽松绑定。

什么是宽松绑定？实际上是springboot进行编程时人性化设计的一种体现，即配置文件中的命名格式与变量名的命名格式可以进行格式上的最大化兼容。兼容到什么程度呢？几乎主流的命名格式都支持，例如：

在ServerConfig中的ipAddress属性名

```
@Component  
@Data  
@ConfigurationProperties(prefix = "servers")  
public class ServerConfig {  
    private String ipAddress;  
}
```

可以与下面的配置属性名规则全兼容

```
servers:  
  ipAddress: 192.168.0.2      # 驼峰模式  
  ip_address: 192.168.0.2     # 下划线模式  
  ip-address: 192.168.0.2     # 烤肉串模式  
  IP_ADDRESS: 192.168.0.2     # 常量模式
```

也可以说，以上4种模式最终都可以匹配到ipAddress这个属性名。为什么这样呢？原因就是在进行匹配时，配置中的名称要去掉中划线和下划线后，忽略大小写的情况下与java代码中的属性名进行忽略大小写的等值匹配，以上4种命名去掉下划线中划线忽略大小写后都是一个词ipaddress，java代码中的属性名忽略大小写后也是ipaddress，这样就可以进行等值匹配了，这就是为什么这4种格式都能匹配成功的原因。不过springboot官方推荐使用烤肉串模式，也就是中划线模式。

到这里我们掌握了一个知识点，就是命名的规范问题。再来看开始出现的编程错误信息

```
Configuration property name 'dataSource' is not valid:  
  
  Invalid characters: 'S'  
  Bean: datasource  
  Reason: Canonical names should be kebab-case ('-' separated), lowercase  
  alpha-numeric characters and must start with a letter  
  
  Action:  
  Modify 'dataSource' so that it conforms to the canonical names requirements.
```

其中Reason描述了报错的原因，规范的名称应该是烤肉串(kebab)模式(case)，即使用-分隔，使用小写字母数字作为标准字符，且必须以字母开头。然后再看我们写的名称dataSource，就不满足上述要求。闹了半天，在书写前缀时，这个词不是随意支持的，必须使用上述标准。编程写了这么久，基本上编程习惯都养成了，到这里又被springboot教育了，没辙，谁让人家东西好用呢，按照人家的要求写吧。

最后说一句，以上规则仅针对springboot中@ConfigurationProperties注解进行属性绑定时有效，对@Value注解进行属性映射无效。有人说，那我不用你不就行了？不用，你小看springboot的推广能力了，到原理篇我们看源码时，你会发现内部全是这玩意儿，算了，拿人手短吃人嘴短，认怂吧。

总结

1. @ConfigurationProperties绑定属性时支持属性名宽松绑定，这个宽松体现在属性名的命名规则上
2. @Value注解不支持松散绑定规则
3. 绑定前缀名推荐采用烤肉串命名规则，即使用中划线做分隔符

KF-2-3.常用计量单位绑定

在前面的配置中，我们书写了如下配置值，其中第三项超时时间timeout描述了服务器操作超时时间，当前值是-1表示永不超时。

```
servers:  
  ip-address: 192.168.0.1  
  port: 2345  
  timeout: -1
```

但是每个人都这个值的理解会产生不同，比如线上服务器完成一次主从备份，配置超时时间240，这个240如果单位是秒就是超时时间4分钟，如果单位是分钟就是超时时间4小时。面对一次线上服务器的主从备份，设置4分钟，简直是开玩笑，别说拷贝过程，备份之前的压缩过程4分钟也搞不定，这个时候问题就来了，怎么解决这个误会？

除了加强约定之外，springboot充分利用了JDK8中提供的全新的用来表示计量单位的新数据类型，从根本上解决这个问题。以下模型类中添加了两个JDK8中新增的类，分别是Duration和DataSize

```
@Component
@Data
@ConfigurationProperties(prefix = "servers")
public class ServerConfig {
    @DurationUnit(ChronoUnit.HOURS)
    private Duration serverTimeout;
    @DataSizeUnit(DataUnit.MEGABYTES)
    private DataSize dataSize;
}
```

Duration: 表示时间间隔，可以通过@DurationUnit注解描述时间单位，例如上例中描述的单位为小时(ChronoUnit.HOURS)

DataSize: 表示存储空间，可以通过@DataSizeUnit注解描述存储空间单位，例如上例中描述的单位为MB (DataUnit.MEGABYTES)

使用上述两个单位就可以有效避免因沟通不同步或文档不健全导致的信息不对称问题，从根本上解决了问题，避免产生误读。

Druation常用单位如下：

DAYS	ChronoUnit
SECONDS	ChronoUnit
MINUTES	ChronoUnit
HALF_DAYS	ChronoUnit
CENTURIES	ChronoUnit
DECADES	ChronoUnit
ERAS	ChronoUnit
FOREVER	ChronoUnit
HOURS	ChronoUnit
MICROS	ChronoUnit
MILLENNIA	ChronoUnit
MILLIS	ChronoUnit
MONTHS	ChronoUnit
NANOS	ChronoUnit
WEEKS	ChronoUnit
YEARS	ChronoUnit

Press Ctrl+, to choose the selectedit afterwards Next Tip

DataSize常用单位如下：

MEGABYTES	DataUnit
BYTES	DataUnit
GIGABYTES	DataUnit
KILOBYTES	DataUnit
TERABYTES	DataUnit

Press Enter to insert, Tab to replace

KF-2-4.校验

目前我们在进行属性绑定时可以通过松散绑定规则在书写时放飞自我了，但是在书写时由于无法感知模型类中的数据类型，就会出现类型不匹配的问题，比如代码中需要int类型，配置中给了非法的数据值，例如写一个“a”，这种数据肯定无法有效的绑定，还会引发错误。SpringBoot给出了强大的数据校验功能，可以有效的避免此类问题的发生。在JAVAEE的JSR303规范中给出了具体的数据校验标准，开发者可以根据自己的需要选择对应的校验框架，此处使用Hibernate提供的校验框架来作为实现进行数据校验。书写应用格式非常固定，话不多说，直接上步骤

步骤①：开启校验框架

```
<!--1.导入JSR303规范-->
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
</dependency>
<!--使用hibernate框架提供的校验器做实现-->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
```

步骤②：在需要开启校验功能的类上使用注解@Validated开启校验功能

```
@Component
@Data
@ConfigurationProperties(prefix = "servers")
//开启对当前bean的属性注入校验
@Validated
public class ServerConfig {
```

步骤③：对具体的字段设置校验规则

```
@Component
@Data
@ConfigurationProperties(prefix = "servers")
//开启对当前bean的属性注入校验
@Validated
public class ServerConfig {
    //设置具体的规则
    @Max(value = 8888,message = "最大值不能超过8888")
    @Min(value = 202,message = "最小值不能低于202")
    private int port;
}
```

通过设置数据格式校验，就可以有效避免非法数据加载，其实使用起来还是挺轻松的，基本上就是一个格式。

总结

1. 开启Bean属性校验功能一共3步：导入JSR303与Hibernate校验框架坐标、使用@Validated注解启用校验功能、使用具体校验规则规范数据校验格式

KF-2-5.数据类型转换

有关spring属性注入的问题到这里基本上就讲完了，但是最近一名开发者向我咨询了一个问题，我觉得需要给各位学习者分享一下。在学习阶段其实我们遇到的问题往往复杂度比较低，单一性比较强，但是到了线上开发时，都是综合性的问题，而这个开发者遇到的问题就是由于bean的属性注入引发的灾难。

先把问题描述一下，这位开发者连接数据库正常操作，但是运行程序后显示的信息是密码错误。

```
java.sql.SQLException: Access denied for user 'root'@'localhost' (using password: YES)
```

其实看到这个报错，几乎所有的学习者都能分辨出来，这是用户名和密码不匹配，就就是密码输入错了，但是问题就在于密码并没有输入错误，这就比较讨厌了。给的报错信息无法帮助你有效的分析问题，甚至会给你带到沟里。如果是初学者，估计这会心态就崩了，我密码没错啊，你怎么能说我有错误呢？来看看用户名密码的配置是如何写的：

```
spring:  
  datasource:  
    driver-class-name: com.mysql.cj.jdbc.Driver  
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC  
    username: root  
    password: 0127
```

这名开发者的生日是1月27日，所以密码就使用了0127，其实问题就出在这里了。

之前在基础篇讲属性注入时，提到过类型相关的知识，在整数相关知识中有这么一句话，**支持二进制，八进制，十六进制**

```
boolean: TRUE          #TRUE, true, True, FALSE, false, False均可  
float: 3.14           #6.8523015e+5 #支持科学计数法  
int: 123              #0b1010_0111_0100_1010_1110      #支持二进制、八进制、十六进制  
null: ~               #使用~表示null  
string: HelloWorld    #字符串可以直接书写  
string2: "Hello World" #可以使用双引号包裹特殊字符  
date: 2018-02-17       #日期必须使用yyyy-MM-dd格式  
datetime: 2018-02-17T15:02:31+08:00 #时间和日期之间使用T连接，最后使用+代表时区
```

这个问题就处在这里了，因为0127在开发者眼中是一个字符串“0127”，但是在springboot看来，这就是一个数字，而且是一个八进制的数字。当后台使用String类型接收数据时，如果配置文件中配置了一个整数值，他是先安装整数进行处理，读取后再转换成字符串。巧了，0127撞上了八进制的格式，所以最终以十进制数字87的结果存在了。

这里提两个注意点，第一，字符串标准书写加上引号包裹，养成习惯，第二，遇到0开头的数据多注意吧。

总结

1. yaml文件中对于数字的定义支持进制书写格式，如需使用字符串请使用引号明确标注

KF-3.测试

说完bean配置相关的内容，下面要对前面讲过的一个知识做加强了，测试。测试是保障程序正确性的唯一屏障，在企业级开发中更是不可缺少，但是由于测试代码往往不产生实际效益，所以一些小型公司并不是很关注，导致一些开发者从小型公司进入中大型公司后，往往这一块比较短板，所以还是要拿出来把这一块知识好好说说，做一名专业的开发人员。

KF-3-1.加载测试专用属性

测试过程本身并不是一个复杂的过程，但是很多情况下测试时需要模拟一些线上情况，或者模拟一些特殊情况。如果当前环境按照线上环境已经设定好了，例如是下面的配置

```
env:  
  maxMemory: 32GB  
  minMemory: 16GB
```

但是你现在想测试对应的兼容性，需要测试如下配置

```
env:  
  maxMemory: 16GB  
  minMemory: 8GB
```

这个时候我们能不能每次测试的时候都去修改源码application.yml中的配置进行测试呢？显然是不行的。每次测试前改过来，每次测试后改回去，这太麻烦了。于是我们就想，需要在测试环境中创建一组临时属性，去覆盖我们源码中设定的属性，这样测试用例就相当于是一个独立的环境，能够独立测试，这样就方便多了。

临时属性

springboot已经为我们开发者早就想好了这种问题该如何解决，并且提供了对应的功能入口。在测试用例程序中，通过对注解@SpringBootTest添加属性来模拟临时属性，具体如下：

```
//properties属性可以为当前测试用例添加临时的属性配置  
@SpringBootTest(properties = {"test.prop=testValue1"})  
public class PropertiesAndArgsTest {  
  
    @Value("${test.prop}")  
    private String msg;  
  
    @Test  
    void testProperties(){  
        System.out.println(msg);  
    }  
}
```

使用注解@SpringBootTest的properties属性就可以为当前测试用例添加临时的属性，覆盖源码配置文件中对应的属性值进行测试。

临时参数

除了上述这种情况，在前面讲解使用命令行启动springboot程序时讲过，通过命令行参数也可以设置属性值。而且线上启动程序时，通常都会添加一些专用的配置信息。作为运维人员他们才不懂java，更不懂这些配置的信息具体格式该怎么写，那如果我们作为开发者提供了对应的书写内容后，能否提前测试一下这些配置信息是否有效呢？当时是可以的，还是通过注解@SpringBootTest的另一个属性进来

行设定。

```
//args属性可以为当前测试用例添加临时的命令行参数
@SpringBootTest(args={"--test.prop=testValue2"})
public class PropertiesAndArgsTest {

    @Value("${test.prop}")
    private String msg;

    @Test
    void testProperties(){
        System.out.println(msg);
    }
}
```

使用注解@SpringBootTest的args属性就可以为当前测试用例模拟命令行参数并进行测试。

说到这里，好奇宝宝们肯定就有新问题了，如果两者共存呢？其实如果思考一下配置属性与命令行参数的加载优先级，这个结果就不言而喻了。在属性加载的优先级设定中，有明确的优先级设定顺序，还记得下面这个顺序吗？

1. Default properties (specified by setting `SpringApplication.setDefaultProperties()`).
2. `@PropertySource` annotations on your `@Configuration` classes. Please note that such property sources are not added to the `Environment` until the application context is being refreshed. This is too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.
3. Config data (such as `application.properties` files)
4. A `RandomValuePropertySource` that has properties only in `random.*`.
5. OS environment variables.
6. Java System properties (`System.getProperties()`).
7. JNDI attributes from `java:comp/env`.
8. `ServletContext` init parameters.
9. `ServletConfig` init parameters.
10. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. `properties` attribute on your tests. Available on `@SpringBootTest` and the `test` annotations for testing a particular slice of your application.
13. `@TestPropertySource` annotations on your tests.
14. Devtools global settings properties in the `$HOME/.config/spring-boot` directory when devtools is active.

在这个属性加载优先级的顺序中，明确规定了命令行参数的优先级排序是11，而配置属性的优先级是3，结果不言而喻了，args属性配置优先于properties属性配置加载。

到这里我们就掌握了如果在测试用例中去模拟临时属性的设定。

总结

1. 加载测试临时属性可以通过注解@SpringBootTest的properties和args属性进行设定，此设定应用范围仅适用于当前测试用例

思考

应用于测试环境的临时属性解决了，如果想在测试的时候临时加载一些bean能不做呢？也就是说我测试时，想搞一些独立的bean出来，专门应用于测试环境，能否实现呢？咱们下一节再讲。

KF-3-2.加载测试专用配置

上一节提出了临时配置一些专用于测试环境的bean的需求，这一节我们就来解决这个问题。

学习过Spring的知识，我们都知道，其实一个spring环境中可以设置若干个配置文件或配置类，若干个配置信息可以同时生效。现在我们的需求就是在测试环境中再添加一个配置类，然后启动测试环境时，生效此配置就行了。其实做法和spring环境中加载多个配置信息的方式完全一样。具体操作步骤如下：

步骤①：在测试包test中创建专用的测试环境配置类

```
@Configuration  
public class MsgConfig {  
    @Bean  
    public String msg(){  
        return "bean msg";  
    }  
}
```

上述配置仅用于演示当前实验效果，实际开发可不能这么注入String类型的数据

步骤②：在启动测试环境时，导入测试环境专用的配置类，使用@Import注解即可实现

```
@SpringBootTest  
@Import({MsgConfig.class})  
public class ConfigurationTest {  
  
    @Autowired  
    private String msg;  
  
    @Test  
    void testConfiguration(){  
        System.out.println(msg);  
    }  
}
```

到这里就通过@Import属性实现了基于开发环境的配置基础上，对配置进行测试环境的追加操作，实现了1+1的配置环境效果。这样我们就可以实现每一个不同的测试用例加载不同的bean的效果，丰富测试用例的编写，同时不影响开发环境的配置。

总结

1. 定义测试环境专用的配置类，然后通过@Import注解在具体的测试中导入临时的配置，例如测试用例，方便测试过程，且上述配置不影响其他的测试类环境

思考

当前我们已经可以实现业务层和数据层的测试，并且通过临时配置，控制每个测试用例加载不同的测试数据。但是实际企业开发不仅要保障业务层与数据层的功能安全有效，也要保障表现层的功能正常。但是我们目的对表现层的测试都是通过postman手工测试的，并没有在打包过程中体现表现层功能被测试通过。能否在测试用例中对表现层进行功能测试呢？还真可以，咱们下一节再讲。

KF-3-3.Web环境模拟测试

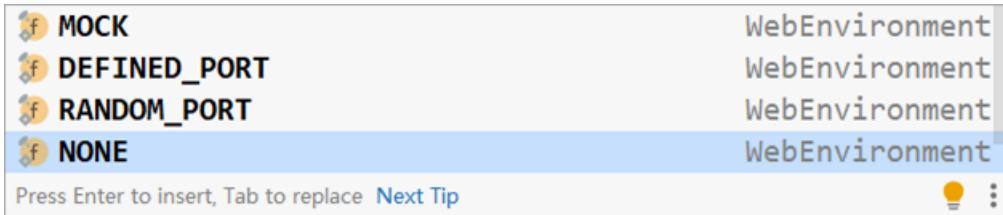
在测试中对表现层功能进行测试需要一个基础和一个功能。所谓的一个基础是运行测试程序时，必须启动web环境，不然没法测试web功能。一个功能是必须在测试程序中具备发送web请求的能力，不然无法实现web功能的测试。所以在测试用例中测试表现层接口这项工作就转换成了两件事，一，如何在测试类中启动web测试，二，如何在测试类中发送web请求。下面一件事一件事进行，先说第一个

测试类中启动web环境

每一个springboot的测试类上方都会标准@SpringBootTest注解，而注解带有一个属性，叫做webEnvironment。通过该属性就可以设置在测试用例中启动web环境，具体如下：

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class WebTest {
}
```

测试类中启动web环境时，可以指定启动的Web环境对应的端口，springboot提供了4种设置值，分别如下：



- MOCK：根据当前设置确认是否启动web环境，例如使用了Servlet的API就启动web环境，属于适配性的配置
- DEFINED_PORT：使用自定义的端口作为web服务器端口
- RANDOM_PORT：使用随机端口作为web服务器端口
- NONE：不启动web环境

通过上述配置，现在启动测试程序时就可以正常启用web环境了，建议大家测试时使用RANDOM_PORT，避免代码中因为写死设定引发线上功能打包测试时由于端口冲突导致意外现象的出现。就是说你程序中写了用8080端口，结果线上环境8080端口被占用了，结果你代码中所有写的东西都要改，这就是写死代码的代价。现在你用随机端口就可以测试出来你有没有这种问题的隐患了。

测试环境中的web环境已经搭建好了，下面就可以来解决第二个问题了，如何在程序代码中发送web请求。

测试类中发送请求

对于测试类中发送请求，其实java的API就提供对应的功能，只不过平时各位小伙伴接触的比较少，所以较为陌生。springboot为了便于开发者进行对应的功能开发，对其又进行了包装，简化了开发步骤，具体操作如下：

步骤①：在测试类中开启web虚拟调用功能，通过注解@AutoConfigureMockMvc实现此功能的开启

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
//开启虚拟MVC调用
@AutoConfigureMockMvc
public class WebTest {
}
```

步骤②：定义发起虚拟调用的对象MockMVC，通过自动装配的形式初始化对象

```

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
//开启虚拟MVC调用
@AutoConfigureMockMvc
public class webTest {

    @Test
    void testweb(@Autowired MockMvc mvc) {
    }
}

```

步骤③：创建一个虚拟请求对象，封装请求的路径，并使用MockMVC对象发送对应请求

```

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
//开启虚拟MVC调用
@AutoConfigureMockMvc
public class webTest {

    @Test
    void testweb(@Autowired MockMvc mvc) throws Exception {
        //http://localhost:8080/books
        //创建虚拟请求，当前访问/books
        MockHttpServletRequestBuilder builder =
        MockMvcBuilders.get("/books");
        //执行对应的请求
        mvc.perform(builder);
    }
}

```

执行测试程序，现在就可以正常的发送/books对应的请求了，注意访问路径不要写<http://localhost:8080/books>，因为前面的服务器IP地址和端口使用的是当前虚拟的web环境，无需指定，仅指定请求的具体路径即可。

总结

- 在测试类中测试web层接口要保障测试类启动时启动web容器，使用@SpringBootTest注解的webEnvironment属性可以虚拟web环境用于测试
- 为测试方法注入MockMvc对象，通过MockMvc对象可以发送虚拟请求，模拟web请求调用过程

思考

目前已经成功的发送了请求，但是还没有起到测试的效果，测试过程必须出现预计值与真实值的比对结果才能确认测试结果是否通过，虚拟请求中能对哪些请求结果进行比对呢？咱们下一节再讲。

web环境请求结果比对

上一节已经在测试用例中成功的模拟出了web环境，并成功的发送了web请求，本节就来解决发送请求后如何比对发送结果的问题。其实发完请求得到的信息只有一种，就是响应对象。至于响应对象中包含什么，就可以比对什么。常见的比对内容如下：

- 响应状态匹配

```

@Test
void testStatus(@Autowired MockMvc mvc) throws Exception {
    MockHttpServletResponseBuilder builder =
    MockMvcBuilders.get("/books");
    ResultActions action = mvc.perform(builder);
    //设定预期值 与真实值进行比较，成功测试通过，失败测试失败
    //定义本次调用的预期值
    StatusResultMatchers status = MockMvcResultMatchers.status();
    //预计本次调用时成功的：状态200
    ResultMatcher ok = statusisOk();
    //添加预计值到本次调用过程中进行匹配
    action.andExpect(ok);
}

```

- 响应体匹配（非json数据格式）

```

@Test
void testBody(@Autowired MockMvc mvc) throws Exception {
    MockHttpServletResponseBuilder builder =
    MockMvcBuilders.get("/books");
    ResultActions action = mvc.perform(builder);
    //设定预期值 与真实值进行比较，成功测试通过，失败测试失败
    //定义本次调用的预期值
    ContentResultMatchers content = MockMvcResultMatchers.content();
    ResultMatcher result = content.string("springboot2");
    //添加预计值到本次调用过程中进行匹配
    action.andExpect(result);
}

```

- 响应体匹配（json数据格式，开发中的主流使用方式）

```

@Test
void testJson(@Autowired MockMvc mvc) throws Exception {
    MockHttpServletResponseBuilder builder =
    MockMvcBuilders.get("/books");
    ResultActions action = mvc.perform(builder);
    //设定预期值 与真实值进行比较，成功测试通过，失败测试失败
    //定义本次调用的预期值
    ContentResultMatchers content = MockMvcResultMatchers.content();
    ResultMatcher result = content.json(
    "{\"id\":1,\"name\":\"springboot2\",\"type\":\"springboot\"}");
    //添加预计值到本次调用过程中进行匹配
    action.andExpect(result);
}

```

- 响应头信息匹配

```

@Test
void testContentType(@Autowired MockMvc mvc) throws Exception {
    MockHttpServletRequestBuilder builder =
    MockMvcBuilders.get("/books");
    ResultActions action = mvc.perform(builder);
    //设定预期值 与真实值进行比较，成功测试通过，失败测试失败
    //定义本次调用的预期值
    HeaderResultMatchers header = MockMvcResultMatchers.header();
    ResultMatcher contentType = header.string("Content-Type",
"application/json");
    //添加预计值到本次调用过程中进行匹配
    action.andExpect(contentType);
}

```

基本上齐了，头信息，正文信息，状态信息都有了，就可以组合出一个完美的响应结果比对结果了。以下范例就是三种信息同时进行匹配校验，也是一个完整的信息匹配过程。

```

@Test
void test GetById(@Autowired MockMvc mvc) throws Exception {
    MockHttpServletRequestBuilder builder =
    MockMvcBuilders.get("/books");
    ResultActions action = mvc.perform(builder);

    StatusResultMatchers status = MockMvcResultMatchers.status();
    ResultMatcher ok = status.isok();
    action.andExpect(ok);

    HeaderResultMatchers header = MockMvcResultMatchers.header();
    ResultMatcher contentType = header.string("Content-Type",
"application/json");
    action.andExpect(contentType);

    ContentResultMatchers content = MockMvcResultMatchers.content();
    ResultMatcher result = content.json(
 "{\"id\":1,\"name\":\"springboot\",\"type\":\"springboot\"}");
    action.andExpect(result);
}

```

总结

1. web虚拟调用可以对本地虚拟请求的返回响应信息进行比对，分为响应头信息比对、响应体信息比对、响应状态信息比对

KF-3-4.数据层测试回滚

当前我们的测试程序可以完美的进行表现层、业务层、数据层接口对应的功能测试了，但是测试用例开发完成后，在打包的阶段由于test生命周期属于必须被运行的生命周期，如果跳过会给系统带来极高的安全隐患，所以测试用例必须执行。但是新的问题就呈现了，测试用例如果测试时产生了事务提交就会在测试过程中对数据库数据产生影响，进而产生垃圾数据。这个过程不是我们希望发生的，作为开发者测试用例该运行运行，但是过程中产生的数据不要在我的系统中留痕，这样该如何处理呢？

springboot早就为开发者想到了这个问题，并且针对此问题给出了最简解决方案，在原始测试用例中添加注解@Transactional即可实现当前测试用例的事务不提交。当程序运行后，只要注解@Transactional出现的位置存在注解@SpringBootTest，springboot就会认为这是一个测试程序，无需提交事务，所以也就可以避免事务的提交。

```
@SpringBootTest
@Transactional
@Rollback(true)
public class DaoTest {
    @Autowired
    private BookService bookService;

    @Test
    void testSave(){
        Book book = new Book();
        book.setName("springboot3");
        book.setType("springboot3");
        book.setDescription("springboot3");

        bookService.save(book);
    }
}
```

如果开发者想提交事务，也可以，再添加一个@RollBack的注解，设置回滚状态为false即可正常提交事务，是不是很方便？springboot在辅助开发者日常工作这一块展现出了惊人的能力，实在太贴心了。

总结

1. 在springboot的测试类中通过添加注解@Transactional来阻止测试用例提交事务
2. 通过注解@Rollback控制springboot测试类执行结果是否提交事务，需要配合注解@Transactional使用

思考

当前测试程序已经近乎完美了，但是由于测试用例中书写的测试数据属于固定数据，往往失去了测试的意义，开发者可以针对测试用例进行针对性开发，这样就有可能出现测试用例不能完美呈现业务逻辑代码是否真实有效的达成业务目标的现象，解决方案其实很容易想，测试用例的数据只要随机产生就可以了，能实现吗？咱们下一节再讲。

KF-3-5. 测试用例数据设定

对于测试用例的数据固定书写肯定是不合理的，springboot提供了在配置中使用随机值的机制，确保每次运行程序加载的数据都是随机的。具体如下：

```
 testcase:
  book:
    id: ${random.int}
    id2: ${random.int(10)}
    type: ${random.int!5,10!}
    name: ${random.value}
    uuid: ${random.uuid}
    publishTime: ${random.long}
```

当前配置就可以在每次运行程序时创建一组随机数据，避免每次运行时数据都是固定值的尴尬现象发生，有助于测试功能的进行。数据的加载按照之前加载数据的形式，使用@ConfigurationProperties注解即可

```
@Component
@Data
@ConfigurationProperties(prefix = "testcase.book")
public class BookCase {
    private int id;
    private int id2;
    private int type;
    private String name;
    private String uuid;
    private long publishTime;
}
```

对于随机值的产生，还有一些小的限定规则，比如产生的数值性数据可以设置范围等，具体如下：

```
testcast:
book:
  id: ${random.int}          # 随机整数
  id2: ${random.int(10)}     # 10以内随机数
  type: ${random.int(10,20)} # 10到20随机数
  uuid: ${random.uuid}       # 随机uuid
  name: ${random.value}      # 随机字符串,MD5字符串,32位
  publishTime: ${random.long} # 随机整数(Long范围)
```

- \${random.int}表示随机整数
- \${random.int(10)}表示10以内的随机数
- \${random.int(10,20)}表示10到20的随机数
- 其中()可以是任意字符，例如[]，!!均可

总结

1. 使用随机数据可以替换测试用例中书写的固定数据，提高测试用例中的测试数据有效性

KF-4.数据层解决方案

开发实用篇前三章基本上是开胃菜，从第四章开始，开发实用篇进入到了噩梦难度了，从这里开始，不再是单纯的在springboot内部搞事情了，要涉及到很多相关知识。本章节主要内容都是和数据存储与读取相关，前期学习的知识与数据层有关的技术基本上都围绕在数据库这个层面上，所以本章要讲的第一个大的分支就是SQL解决方案相关的内容，除此之外，数据的来源还可以是非SQL技术相关的数据操作，因此第二部分围绕着NOSQL解决方案讲解。至于什么是NOSQL解决方案，讲到了再说吧。下面就开始从SQL解决方案说起。

KF-4-1.SQL

回忆一下之前做SSMP整合的时候数据层解决方案涉及到了哪些技术？MySQL数据库与MyBatisPlus框架，后面又学了Druid数据源的配置，所以现在数据层解决方案可以说是Mysql+Druid+MyBatisPlus。而三个技术分别对应了数据层操作的三个层面：

- 数据源技术：Druid
- 持久化技术：MyBatisPlus
- 数据库技术：MySQL

下面的研究就分为三个层面进行研究，对应上面列出的三个方面，咱们就从第一个数据源技术开始说起。

数据源技术

目前我们使用的数据源技术是Druid，运行时可以在日志中看到对应的数据源初始化信息，具体如下：

```
INFO 28600 --- [           main] c.a.d.s.b.a.DruidDataSourceAutoConfigure : Init
DruidDataSource
INFO 28600 --- [           main] com.alibaba.druid.pool.DruidDataSource  :
{dataSource-1} initied
```

如果不使用Druid数据源，程序运行后是什么样子呢？是独立的数据库连接对象还是有其他的连接池技术支持呢？将Druid技术对应的starter去掉再次运行程序可以在日志中找到如下初始化信息：

```
INFO 31820 --- [           main] com.zaxxer.hikari.HikariDataSource  :
HikariPool-1 - Starting...
INFO 31820 --- [           main] com.zaxxer.hikari.HikariDataSource  :
HikariPool-1 - Start completed.
```

虽然没有DruidDataSource相关的信息了，但是我们发现日志中有HikariDataSource这个信息，就算不懂这是个什么技术，看名字也能看出来，以DataSource结尾的名称，这一定是一个数据源技术。我们又没有手工添加这个技术，这个技术哪里来的呢？这就是这一节要讲的知识，springboot内嵌数据源。

数据层技术是每一个企业级应用程序都会用到的，而其中必定会进行数据库连接的管理。springboot根据开发者的习惯出发，开发者提供了数据源技术，就用你提供的，开发者没有提供，那总不能手工管理一个一个的数据库连接对象啊，怎么办？我给你一个默认的就好了，这样省心又省事，大家都方便。

springboot提供了3款内嵌数据源技术，分别如下：

- HikariCP
- Tomcat提供DataSource
- Commons DBCP

第一种，HikariCP，这是springboot官方推荐的数据源技术，作为默认内置数据源使用。啥意思？你不配置数据源，那就用这个。

第二种，Tomcat提供的DataSource，如果不想用HikariCP，并且使用tomcat作为web服务器进行web程序的开发，使用这个。为什么是Tomcat，不是其他web服务器呢？因为web技术导入starter后，默认使用内嵌tomcat，既然都是默认使用的技术了，那就一用到底，数据源也用它的。有人就提出怎么才能不使用HikariCP用tomcat提供的默认数据源对象呢？把HikariCP技术的坐标排除掉就OK了。

第三种，DBCP，这个使用的条件就更苛刻了，既不使用HikariCP也不使用tomcat的DataSource时，默认给你用这个。

springboot这心操的，也是稀碎啊，就怕你自己管不好连接对象，给你一顿推荐，真是开发界的最强辅助。既然都给你奶上了，那就受用吧，怎么配置使用这些东西呢？之前我们配置druid时使用druid的starter对应的配置如下：

```
spring:  
  datasource:  
    druid:  
      url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC  
      driver-class-name: com.mysql.cj.jdbc.Driver  
      username: root  
      password: root
```

换成是默认的数据源HikariCP后，直接吧druid删掉就行了，如下：

```
spring:  
  datasource:  
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC  
    driver-class-name: com.mysql.cj.jdbc.Driver  
    username: root  
    password: root
```

当然，也可以写上是对hikari做的配置，但是url地址要单独配置，如下：

```
spring:  
  datasource:  
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC  
    hikari:  
      driver-class-name: com.mysql.cj.jdbc.Driver  
      username: root  
      password: root
```

这就是配置hikari数据源的方式。如果想对hikari做进一步的配置，可以继续配置其独立的属性。例如：

```
spring:  
  datasource:  
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC  
    hikari:  
      driver-class-name: com.mysql.cj.jdbc.Driver  
      username: root  
      password: root  
      maximum-pool-size: 50
```

如果不想使用hikari数据源，使用tomcat的数据源或者DBCP配置格式也是一样的。学习到这里，以后我们做数据层时，数据源对象的选择就不再是单一的使用druid数据源技术了，可以根据需要自行选择。

总结

1. springboot技术提供了3种内置的数据源技术，分别是Hikari、tomcat内置数据源、DBCP

持久化技术

说完数据源解决方案，再来说一下持久化解决方案。springboot充分发挥其最强辅助的特征，给开发者提供了一套现成的数据层技术，叫做JdbcTemplate。其实这个技术不能说是springboot提供的，因为不使用springboot技术，一样能使用它，谁提供的呢？spring技术提供的，所以在springboot技术范畴中，这个技术也是存在的，毕竟springboot技术是加速spring程序开发而创建的。

这个技术其实就是回归到jdbc最原始的编程形式来进行数据层的开发，下面直接上操作步骤：

步骤①：导入jdbc对应的坐标，记得是starter

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

步骤②：自动装配JdbcTemplate对象

```
@SpringBootTest
class Springboot15SqlApplicationTests {
    @Test
    void testJdbcTemplate(@Autowired JdbcTemplate jdbcTemplate) {
    }
}
```

步骤③：使用JdbcTemplate实现查询操作（非实体类封装数据的查询操作）

```
@Test
void testJdbcTemplate(@Autowired JdbcTemplate jdbcTemplate) {
    String sql = "select * from tbl_book";
    List<Map<String, Object>> maps = jdbcTemplate.queryForList(sql);
    System.out.println(maps);
}
```

步骤④：使用JdbcTemplate实现查询操作（实体类封装数据的查询操作）

```
@Test
void testJdbcTemplate(@Autowired JdbcTemplate jdbcTemplate) {

    String sql = "select * from tbl_book";
    RowMapper<Book> rm = new RowMapper<Book>() {
        @Override
        public Book mapRow(ResultSet rs, int rowNum) throws SQLException {
            Book temp = new Book();
            temp.setId(rs.getInt("id"));
            temp.setName(rs.getString("name"));
            temp.setType(rs.getString("type"));
            temp.setDescription(rs.getString("description"));
            return temp;
        }
    };
    List<Book> list = jdbcTemplate.query(sql, rm);
    System.out.println(list);
}
```

步骤⑤：使用JdbcTemplate实现增删改操作

```
@Test  
void testJdbcTemplateSave(@Autowired JdbcTemplate jdbcTemplate){  
    String sql = "insert into tb1_book  
values(3,'springboot1','springboot2','springboot3')";  
    jdbcTemplate.update(sql);  
}
```

如果想对JdbcTemplate对象进行相关配置，可以在yml文件中进行设定，具体如下：

```
spring:  
  jdbc:  
    template:  
      query-timeout: -1    # 查询超时时间  
      max-rows: 500        # 最大行数  
      fetch-size: -1      # 缓存行数
```

总结

1. SpringBoot内置JdbcTemplate持久化解决方案
2. 使用JdbcTemplate需要导入spring-boot-starter-jdbc的坐标

数据库技术

截止到目前，springboot给开发者提供了内置的数据源解决方案和持久化解决方案，在数据层解决方案三件套中还剩下一个数据库，莫非springboot也提供有内置的解决方案？还真有，还是一个，三个，这一节就来说说内置的数据库解决方案。

springboot提供了3款内置的数据库，分别是

- H2
- HSQL
- Derby

以上三款数据库除了可以独立安装之外，还可以像是tomcat服务器一样，采用内嵌的形式运行在spirngboot容器中。内嵌在容器中运行，那必须是java对象啊，对，这三款数据库底层都是使用java语言开发的。

我们一直使用MySQL数据库就挺好的，为什么有需求用这个呢？原因就在于这三个数据库都可以采用内嵌容器的形式运行，在应用程序运行后，如果我们进行测试工作，此时测试的数据无需存储在磁盘上，但是又要测试使用，内嵌数据库就方便了，运行在内存中，该测试测试，该运行运行，等服务器关闭后，一切烟消云散，多好，省得你维护外部数据库了。这也是内嵌数据库的最大优点，方便进行功能测试。

下面以H2数据库为例讲解如何使用这些内嵌数据库，操作步骤也非常简单，简单才好用嘛

步骤①：导入H2数据库对应的坐标，一共2个

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

步骤②：将工程设置为web工程，启动工程时启动H2数据库

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

步骤③：通过配置开启H2数据库控制台访问程序，也可以使用其他的数据库连接软件操作

```
spring:
  h2:
    console:
      enabled: true
      path: /h2
```

web端访问路径localhost/h2，访问密码123456，如果访问失败，先配置下列数据源，启动程序运行后再次访问/h2路径就可以正常访问了

```
datasource:
  url: jdbc:h2:~/test
  hikari:
    driver-class-name: org.h2.Driver
    username: sa
    password: 123456
```

步骤④：使用JdbcTemplate或MyBatisPlus技术操作数据库

(略)

其实我们只是换了一个数据库而已，其他的东西都不受影响。一个重要提醒，别忘了，上线时，把内存级数据库关闭，采用MySQL数据库作为数据持久化方案，关闭方式就是设置enabled属性为false即可。

总结

1. H2内嵌式数据库启动方式，添加坐标，添加配置
2. H2数据库线上运行时请务必关闭

到这里SQL相关的数据层解决方案就讲完了，现在的可选技术就丰富的多了。

- 数据源技术：Druid、Hikari、tomcat DataSource、DBCP
- 持久化技术：MyBatisPlus、MyBatis、JdbcTemplate
- 数据库技术：MySQL、H2、HSQL、Derby

现在开发程序时就可以在以上技术中任选一种组织成一套数据库解决方案了。

KF-4-2.NoSQL

SQL数据层解决方案说完了，下面来说说NoSQL数据层解决方案。这个NoSQL是什么意思呢？从字面来看，No表示否定，NoSQL就是非关系型数据库解决方案，意思就是数据该存该取，只是这些数据不放在关系型数据库中了，那放在哪里？自然是一些能够存储数据的其他相关技术中了，比如Redis等。本节讲解的内容就是springboot如何整合这些技术，在springboot官方文档中提供了10种相关技术的整合方案，我们将讲解国内市场中最流行的几款NoSQL数据库整合方案，分别是Redis、MongoDB、ES。

因为每个小伙伴学习这门课程的时候起点不同，为了便于各位学习者更好的学习，每种技术在讲解整合前都会先讲一下安装和基本使用，然后再讲整合。如果对某个技术比较熟悉的小伙伴可以直接跳过安装的学习过程，直接看整合方案即可。此外上述这些技术最佳使用方案都是在Linux服务器上部署，但是考虑到各位小伙伴的学习起点差异过大，所以下面的课程都是以Windows平台作为安装基础讲解，如果想看Linux版软件安装，可以再找到对应技术的学习文档查阅学习。

SpringBoot整合Redis

Redis是一款采用key-value数据存储格式的内存级NoSQL数据库，重点关注数据存储格式，是key-value格式，也就是键值对的存储形式。与MySQL数据库不同，MySQL数据库有表、有字段、有记录，Redis没有这些东西，就是一个名称对应一个值，并且数据以存储在内存中使用为主。什么叫以存储在内存中为主？其实Redis有它的数据持久化方案，分别是RDB和AOF，但是Redis自身并不是为了数据持久化而生的，主要是在内存中保存数据，加速数据访问的，所以说是一款内存级数据库。

Redis支持多种数据存储格式，比如可以直接存字符串，也可以存一个map集合，list集合，后面会涉及到一些不同格式的数据操作，这个需要先学习一下才能进行整合，所以在基本操作中会介绍一些相关操作。下面就先安装，再操作，最后说整合

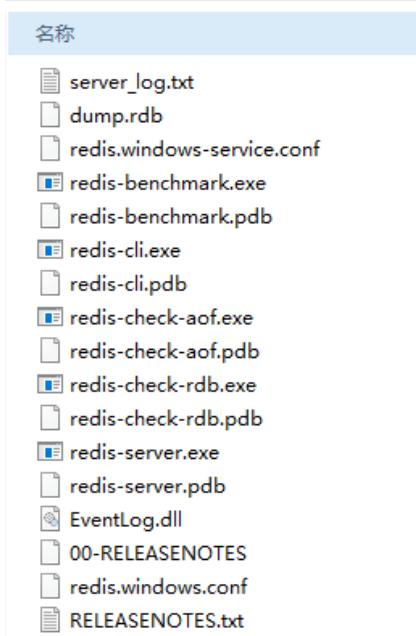
安装

windows版安装包下载地址：<https://github.com/tporadowski/redis/releases>

下载的安装包有两种形式，一种是一键安装的msi文件，还有一种是解压缩就能使用的zip文件，哪种形式都行，这里就不介绍安装过程了，本课程采用的是msi一键安装的msi文件进行安装的。

啥是msi，其实就是一个文件安装包，不仅安装软件，还帮你把安装软件时需要的功能关联在一起，打包操作。比如安装序列、创建和设置安装路径、设置系统依赖项、默认设定安装选项和控制安装过程的属性。说简单点就是一站式服务，安装过程一条龙操作一气呵成，就是为小白用户提供的软件安装程序。

安装完毕后会得到如下文件，其中有两个文件对应两个命令，是启动Redis的核心命令，需要再CMD命令行模式执行。



启动服务器

```
redis-server.exe redis.windows.conf
```

初学者无需调整服务器对外服务端口， 默认6379。

启动客户端

```
redis-cli.exe
```

如果启动redis服务器失败，可以先启动客户端，然后执行shutdown操作后退出，此时redis服务器就可以正常执行了。

基本操作

服务器启动后，使用客户端就可以连接服务器，类似于启动完MySQL数据库，然后启动SQL命令行操作数据库。

放置一个字符串数据到redis中，先为数据定义一个名称，比如name,age等，然后使用命令set设置数据到redis服务器中即可

```
set name itheima
set age 12
```

从redis中取出已经放入的数据，根据名称取，就可以得到对应数据。如果没有对应数据就会得到(nil)

```
get name
get age
```

以上使用的数据存储是一个名称对应一个值，如果要维护的数据过多，可以使用别的数据存储结构。例如hash，它是一种一个名称下可以存储多个数据的存储模型，并且每个数据也可以有自己的二级存储名称。向hash结构中存储数据格式如下：

```
hset a a1 aa1      #对外key名称是a，在名称为a的存储模型中，a1这个key中保存了数据aa1  
hset a a2 aa2
```

获取hash结构中的数据命令如下

```
hget a a1          #得到aa1  
hget a a2          #得到aa2
```

有关redis的基础操作就普及到这里，需要全面掌握redis技术，请参看相关教程学习。

整合

在进行整合之前先梳理一下整合的思想，springboot整合任何技术其实就是在springboot中使用对应技术的API。如果两个技术没有交集，就不存在整合的概念了。所谓整合其实就是使用springboot技术去管理其他技术，几个问题是躲不掉的。

第一，需要先导入对应技术的坐标，而整合之后，这些坐标都有了一些变化

第二，任何技术通常都会有一些相关的设置信息，整合之后，这些信息如何写，写在哪是一个问题

第三，没有整合之前操作如果是模式A的话，整合之后如果没有给开发者带来一些便捷操作，那整合将毫无意义，所以整合后操作肯定要简化一些，那对应的操作方式自然也有所不同

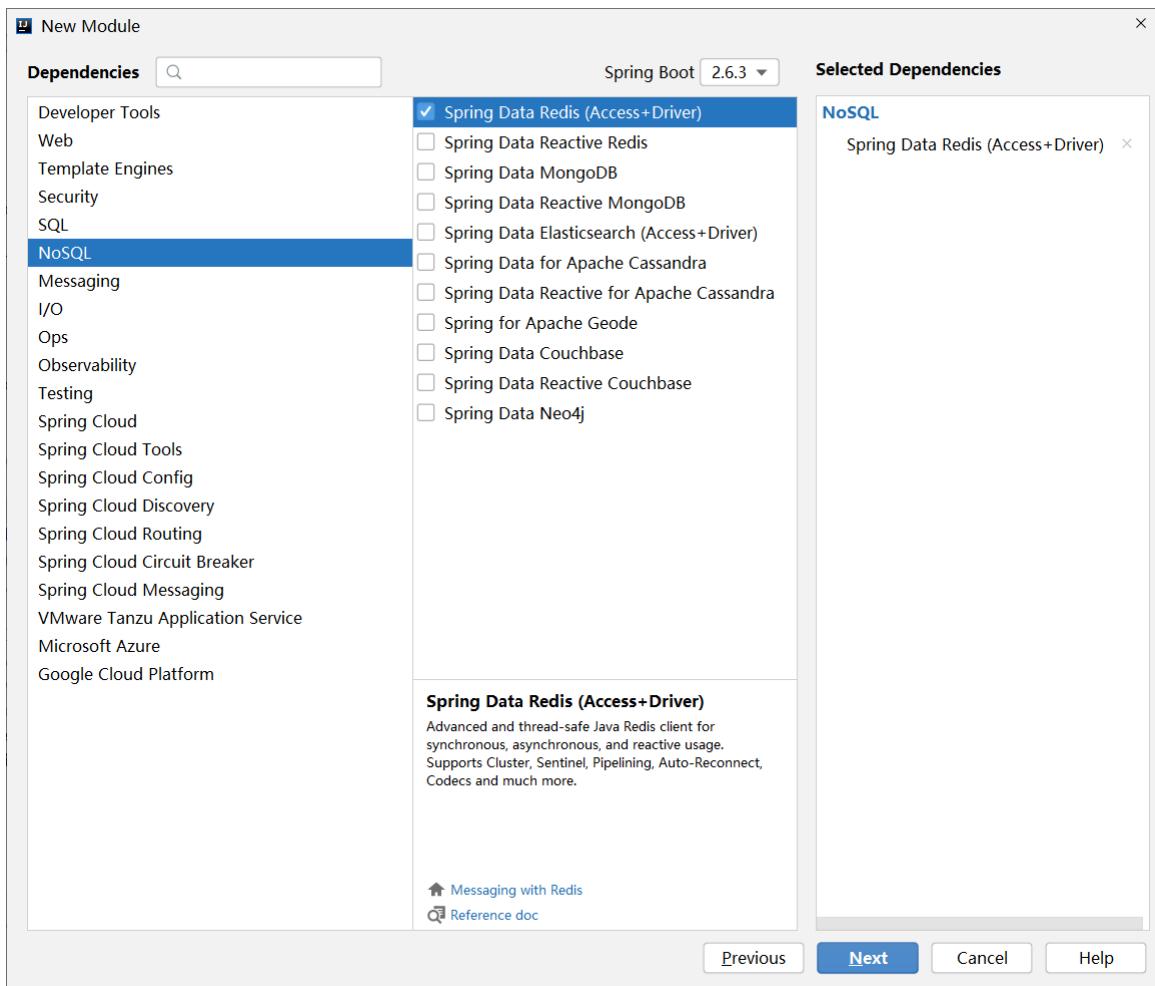
按照上面的三个问题去思考springboot整合所有技术是一种通用思想，在整合的过程中会逐步摸索出整合的套路，而且适用性非常强，经过若干种技术的整合后基本上可以总结出一套固定思维。

下面就开始springboot整合redis，操作步骤如下：

步骤①：导入springboot整合redis的starter坐标

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

上述坐标可以在创建模块的时候通过勾选的形式进行选择，归属NoSQL分类中



步骤②：进行基础配置

```
spring:  
  redis:  
    host: localhost  
    port: 6379
```

操作redis，最基本的信息就是操作哪一台redis服务器，所以服务器地址属于基础配置信息，不可缺少。但是即便你不配置，目前也是可以用的。因为以上两组信息都有默认配置，刚好就是上述配置值。

步骤③：使用springboot整合redis的专用客户端接口操作，此处使用的是RedisTemplate

```
@SpringBootTest  
class Springboot16RedisApplicationTests {  
    @Autowired  
    private RedisTemplate redisTemplate;  
    @Test  
    void set() {  
        ValueOperations ops = redisTemplate.opsForValue();  
        ops.set("age", 41);  
    }  
    @Test  
    void get() {  
        ValueOperations ops = redisTemplate.opsForValue();  
        Object age = ops.get("name");  
        System.out.println(age);  
    }  
}
```

```

    @Test
    void hset() {
        HashOperations ops = redisTemplate.opsForHash();
        ops.put("info", "b", "bb");
    }
    @Test
    void hget() {
        HashOperations ops = redisTemplate.opsForHash();
        Object val = ops.get("info", "b");
        System.out.println(val);
    }
}

```

在操作redis时，需要先确认操作何种数据，根据数据种类得到操作接口。例如使用opsForValue()获取string类型的数据操作接口，使用opsForHash()获取hash类型的数据操作接口，剩下的就是调用对应api操作了。各种类型的数据操作接口如下：



总结

1. springboot整合redis步骤

1. 导入springboot整合redis的starter坐标
2. 进行基础配置
3. 使用springboot整合redis的专用客户端接口RedisTemplate操作

StringRedisTemplate

由于redis内部不提供java对象的存储格式，因此当操作的数据以对象的形式存在时，会进行转码，转换成字符串格式后进行操作。为了方便开发者使用基于字符串为数据的操作，springboot整合redis时提供了专用的API接口StringRedisTemplate，你可以理解为这是RedisTemplate的一种指定数据泛型的操作API。

```
@SpringBootTest
public class StringRedisTemplateTest {
    @Autowired
    private StringRedisTemplate stringRedisTemplate;
    @Test
    void get(){
        ValueOperations<String, String> ops = stringRedisTemplate.opsForValue();
        String name = ops.get("name");
        System.out.println(name);
    }
}
```

redis客户端选择

springboot整合redis技术提供了多种客户端兼容模式， 默认提供的是lettuce客户端技术， 也可以根据需要切换成指定客户端技术， 例如jedis客户端技术， 切换成jedis客户端技术操作步骤如下：

步骤①：导入jedis坐标

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```

jedis坐标受springboot管理， 无需提供版本号

步骤②：配置客户端技术类型， 设置为jedis

```
spring:
  redis:
    host: localhost
    port: 6379
    client-type: jedis
```

步骤③：根据需要设置对应的配置

```
spring:
  redis:
    host: localhost
    port: 6379
    client-type: jedis
    lettuce:
      pool:
        max-active: 16
  jedis:
    pool:
      max-active: 16
```

lettuce与jedis区别

- jedis连接Redis服务器是直连模式，当多线程模式下使用jedis会存在线程安全问题，解决方案可以通过配置连接池使每个连接专用，这样整体性能就大受影响
- lettus基于Netty框架进行与Redis服务器连接，底层设计中采用StatefulRedisConnection。StatefulRedisConnection自身是线程安全的，可以保障并发访问安全问题，所以一个连接可以被多线程复用。当然lettus也支持多连接实例一起工作

总结

1. springboot整合redis提供了StringRedisTemplate对象，以字符串的数据格式操作redis
2. 如果需要切换redis客户端实现技术，可以通过配置的形式进行

SpringBoot整合MongoDB

使用Redis技术可以有效的提高数据访问速度，但是由于Redis的数据格式单一性，无法操作结构化数据，当操作对象型的数据时，Redis就显得捉襟见肘。在保障访问速度的情况下，如果想操作结构化数据，看来Redis无法满足要求了，此时需要使用全新的数据存储结束来解决此问题，本节讲解springboot如何整合MongoDB技术。

MongoDB是一个开源、高性能、无模式的文档型数据库，它是NoSQL数据库产品中的一种，是最像关系型数据库的非关系型数据库。

上述描述中几个词，其中对于我们最陌生的词是无模式的。什么叫无模式呢？简单说就是作为一款数据库，没有固定的数据存储结构，第一条数据可能有A、B、C一共3个字段，第二条数据可能有D、E、F也是3个字段，第三条数据可能是A、C、E3个字段，也就是说数据的结构不固定，这就是无模式。有人会说这有什么用啊？灵活，随时变更，不受约束。基于上述特点，MongoDB的应用面也会产生一些变化。以下列出了一些可以使用MongoDB作为数据存储的场景，但是并不是必须使用MongoDB的场景：

- 淘宝用户数据
 - 存储位置：数据库
 - 特征：永久性存储，修改频度极低
- 游戏装备数据、游戏道具数据
 - 存储位置：数据库、Mongodb
 - 特征：永久性存储与临时存储相结合、修改频度较高
- 直播数据、打赏数据、粉丝数据
 - 存储位置：数据库、Mongodb
 - 特征：永久性存储与临时存储相结合，修改频度极高
- 物联网数据
 - 存储位置：Mongodb
 - 特征：临时存储，修改频度飞速

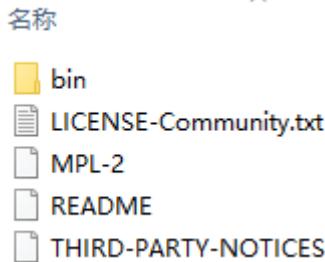
快速了解一下MongoDB，下面直接开始我们的学习，老规矩，先安装，再操作，最后说整合

安装

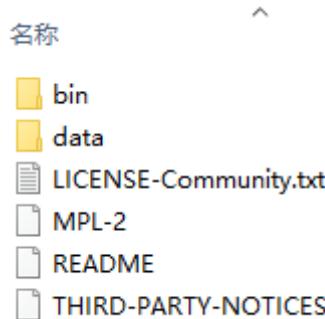
windows版安装包下载地址：<https://www.mongodb.com/try/download>

下载的安装包也有两种形式，一种是一键安装的msi文件，还有一种是解压缩就能使用的zip文件，哪种形式都行，本课程采用解压缩zip文件进行安装。

解压缩完毕后会得到如下文件，其中bin目录包含了所有mongodb的可执行命令



mongodb在运行时需要指定一个数据存储的目录，所以创建一个数据存储目录，通常放置在安装目录中，此处创建data的目录用来存储数据，具体如下



如果在安装的过程中出现了如下警告信息，就是告诉你，你当前的操作系统缺少了一些系统文件，这个不用担心。



根据下列方案即可解决，在浏览器中搜索提示缺少的名称对应的文件，并下载，将下载的文件拷贝到windows安装目录的system32目录下，然后在命令行中执行regsvr32命令注册此文件。根据下载的文件名不同，执行命令前更改对应名称。

```
regsvr32 vcruntime140_1.dll
```

启动服务器

```
mongod --dbpath=..\data\db
```

启动服务器时需要指定数据存储位置，通过参数--dbpath进行设置，可以根据需要自行设置数据存储路径。默认服务端口27017。

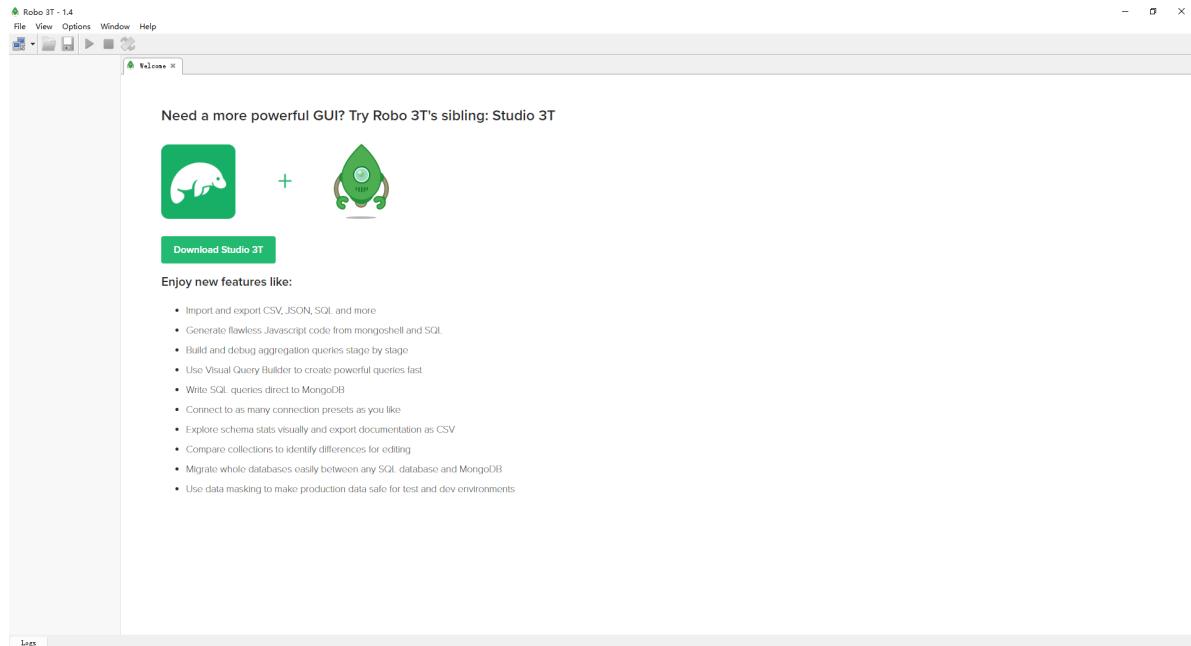
启动客户端

```
mongo --host=127.0.0.1 --port=27017
```

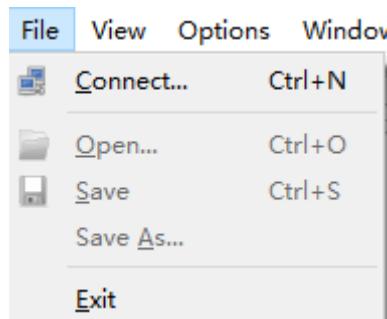
基本操作

MongoDB虽然是一款数据库，但是它的操作并不是使用SQL语句进行的，因此操作方式各位小伙伴可能比较陌生，好在有一些类似于Navicat的数据库客户端软件，能够便捷的操作MongoDB，先安装一个客户端，再来操作MongoDB。

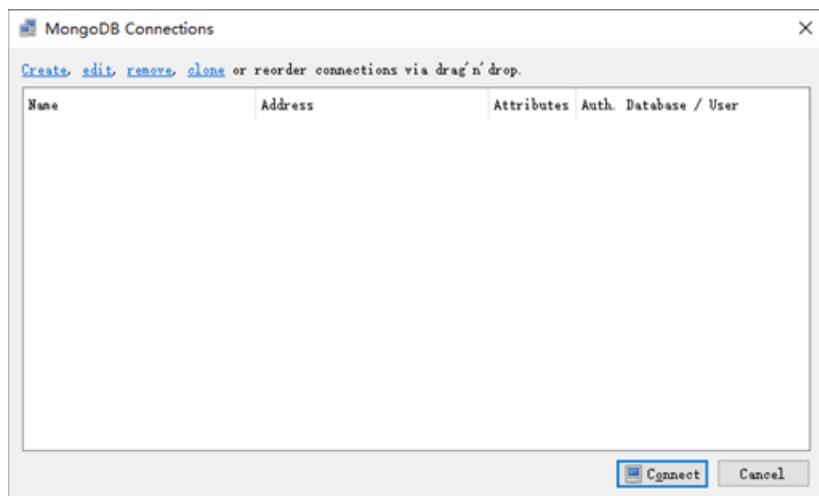
同类型的软件较多，本次安装的软件时Robo3t，Robo3t是一款绿色软件，无需安装，解压缩即可。解压缩完毕后进入安装目录双击robot3t.exe即可使用。



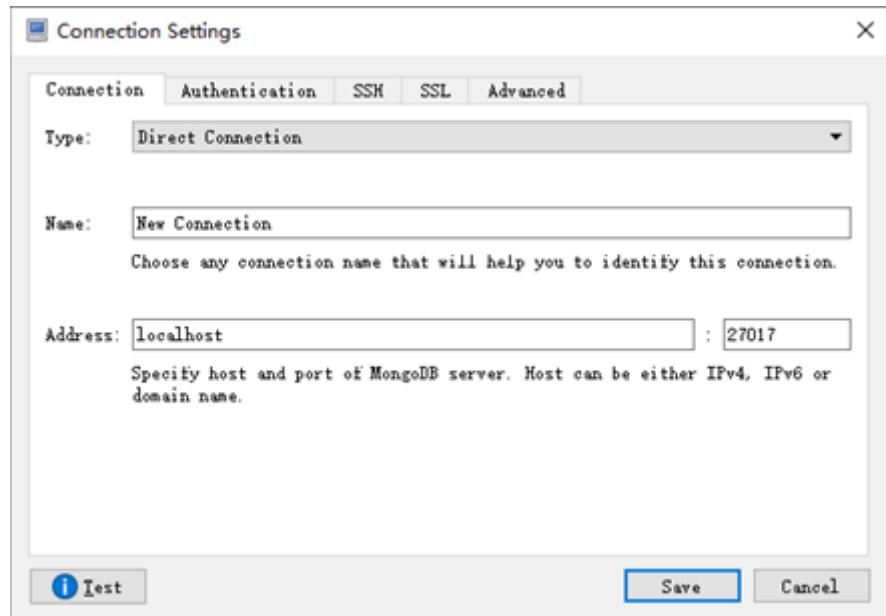
打开软件首先要连接MongoDB服务器，选择【File】菜单，选择【Connect...】



进入连接管理界面后，选择左上角的【Create】链接，创建新的连接设置



如果输入设置值即可连接（默认不修改即可连接本机27017端口）



连接成功后在命令输入区域输入命令即可操作MongoDB。

创建数据库：在左侧菜单中使用右键创建，输入数据库名称即可

创建集合：在Collections上使用右键创建，输入集合名称即可，集合等同于数据库中的表的作用

新增文档：（文档是一种类似json格式的数据，初学者可以先把数据理解为就是json数据）

```
db.集合名称.insert/save/insertOne(文档)
```

删除文档：

```
db.集合名称.remove(条件)
```

修改文档：

```
db.集合名称.update(条件, {操作种类: {文档}})
```

查询文档：

基础查询

查询全部:	<code>db.集合.find();</code>	
查第一条:	<code>db.集合.findOne()</code>	
查询指定数量文档:	<code>db.集合.find().limit(10)</code>	//查10条文档
跳过指定数量文档:	<code>db.集合.find().skip(20)</code>	//跳过20条文档
统计:	<code>db.集合.count()</code>	
排序:	<code>db.集合.sort({age:1})</code>	//按age升序排序
投影:	<code>db.集合名称.find(条件, {name:1, age:1})</code>	//仅保留name与age域

条件查询

基本格式:	<code>db.集合.find({条件})</code>	
模糊查询:	<code>db.集合.find({域名:/正则表达式/})</code>	//等同SQL中的like，比
Like强大，可以执行正则所有规则		
条件比较运算:	<code>db.集合.find({域名:{\$gt:值}})</code>	//等同SQL中的数值比较操
作，例如: name>18		
包含查询:	<code>db.集合.find({域名:{\$in:[值1, 值2]}})</code>	//等同于SQL中的in
条件连接查询:	<code>db.集合.find({\$and:[{条件1},{条件2}]})</code>	//等同于SQL中的and、or

有关MongoDB的基础操作就普及到这里，需要全面掌握MongoDB技术，请参看相关教程学习。

整合

使用springboot整合MongoDB该如何进行呢？其实springboot为什么使用的开发者这么多，就是因为他的套路几乎完全一样。导入坐标，做配置，使用API接口操作。整合Redis如此，整合MongoDB同样如此。

第一，先导入对应技术的整合starter坐标

第二，配置必要信息

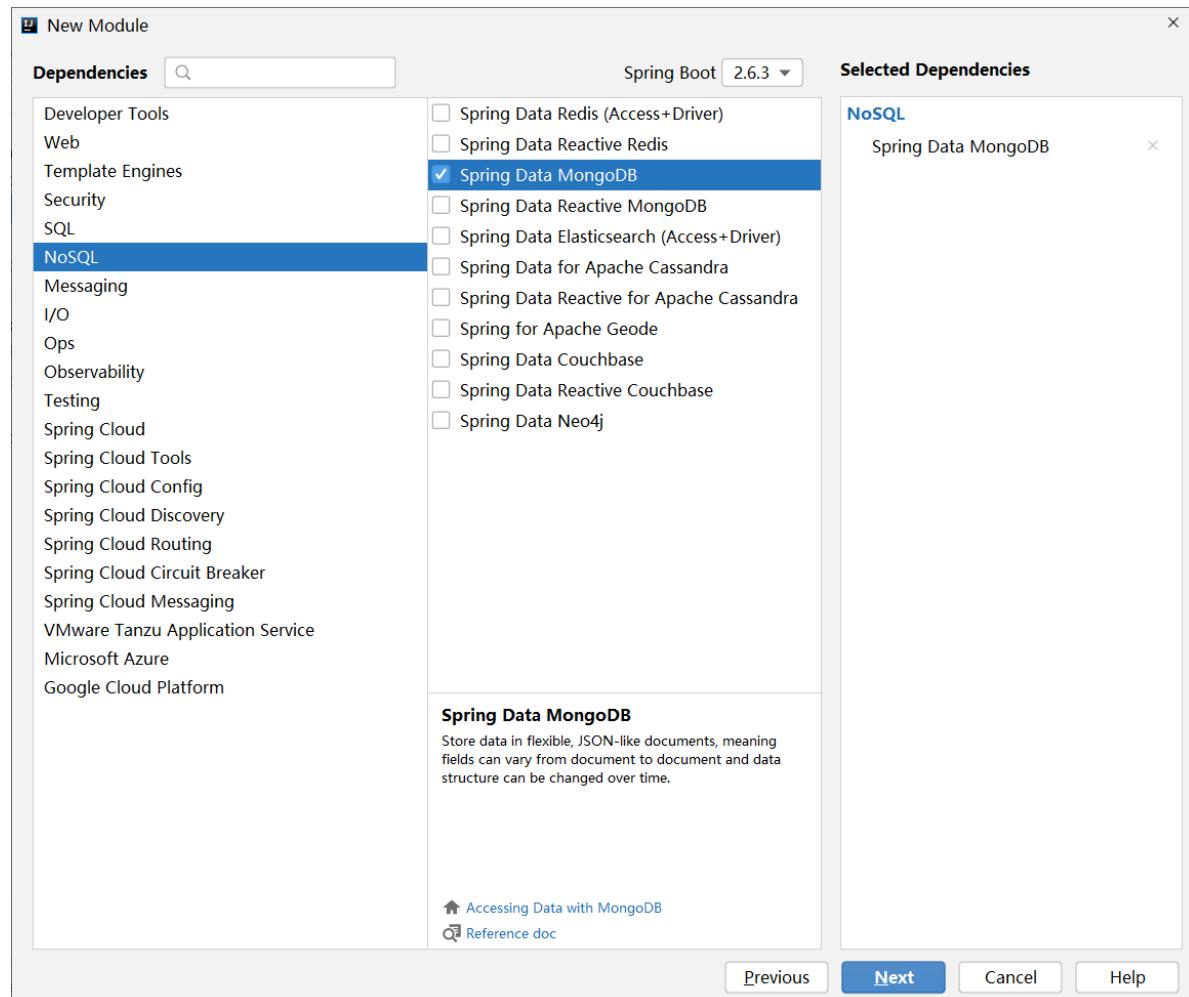
第三，使用提供的API操作即可

下面就开始springboot整合MongoDB，操作步骤如下：

步骤①：导入springboot整合MongoDB的starter坐标

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

上述坐标也可以在创建模块的时候通过勾选的形式进行选择，同样归属NoSQL分类中



步骤②：进行基础配置

```
spring:  
  data:  
    mongodb:  
      uri: mongodb://localhost/itheima
```

操作MongoDB需要的配置与操作redis一样，最基本的信息都是操作哪一台服务器，区别就是连接的服务器IP地址和端口不同，书写格式不同而已。

步骤③：使用springboot整合MongoDB的专用客户端接口MongoTemplate来进行操作

```
@SpringBootTest  
class Springboot17MongodbApplicationTests {  
  @Autowired  
  private MongoTemplate mongoTemplate;  
  @Test  
  void contextLoads() {  
    Book book = new Book();  
    book.setId(2);  
    book.setName("springboot2");  
    book.setType("springboot2");  
    book.setDescription("springboot2");  
    mongoTemplate.save(book);  
  }  
  @Test  
  void find() {  
    List<Book> all = mongoTemplate.findAll(Book.class);  
    System.out.println(all);  
  }  
}
```

整合工作到这里就做完了，感觉既熟悉也陌生。熟悉的是这个套路，三板斧，就这三招，导坐标做配置用API操作，陌生的是这个技术，里面具体的操作API可能会不熟悉，有关springboot整合MongoDB我们就讲到这里。有兴趣可以继续学习MongoDB的操作，然后再来这里通过编程的形式操作MongoDB。

总结

1. springboot整合MongoDB步骤
 1. 导入springboot整合MongoDB的starter坐标
 2. 进行基础配置
 3. 使用springboot整合MongoDB的专用客户端接口MongoTemplate操作

SpringBoot整合ES

NoSQL解决方案已经讲完了两种技术的整合了，Redis可以使用内存加载数据并实现数据快速访问，MongoDB可以在内存中存储类似对象的数据并实现数据的快速访问，在企业级开发中对于速度的追求是永无止境的。下面要讲的内容也是一款NoSQL解决方案，只不过他的作用不是为了直接加速数据的读写，而是加速数据的查询的，叫做ES技术。

ES (Elasticsearch) 是一个分布式全文搜索引擎，重点是全文搜索。

那什么是全文搜索呢？比如用户要买一本书，以Java为关键字进行搜索，不管是书名中还是书的介绍中，甚至是书的作者名字，只要包含Java就作为查询结果返回给用户查看，上述过程就使用了全文搜索技术。搜索的条件不再是仅用于对某一个字段进行比对，而是在一条数据中使用搜索条件去比对更多的字段，只要能匹配上就列入查询结果，这就是全文搜索的目的。而ES技术就是一种可以实现上述效果的技术。

要实现全文搜索的效果，不可能使用数据库中like操作去进行比对，这种效率太低了。ES设计了一种全新的思想，来实现全文搜索。具体操作过程如下：

1. 将被查询的字段的数据全部文本信息进行查分，分成若干个词
 - 例如“中华人民共和国”就会被拆分成三个词，分别是“中华”、“人民”、“共和国”，此过程有专业术语叫做分词。分词的策略不同，分出的效果不一样，不同的分词策略称为分词器。
2. 将分词得到的结果存储起来，对应每条数据的id
 - 例如id为1的数据中名称这一项的值是“中华人民共和国”，那么分词结束后，就会出现“中华”对应id为1，“人民”对应id为1，“共和国”对应id为1
 - 例如id为2的数据中名称这一项的值是“人民代表大会”，那么分词结束后，就会出现“人民”对应id为2，“代表”对应id为2，“大会”对应id为2
 - 此时就会出现如下对应结果，按照上述形式可以对所有文档进行分词。需要注意分词的过程不仅是仅对一个字段进行，而是对每一个参与查询的字段都执行，最终结果汇总到一个表格中

分词结果关键字	对应id
中华	1
人民	1,2
共和国	1
代表	2
大会	2

3. 当进行查询时，如果输入“人民”作为查询条件，可以通过上述表格数据进行比对，得到id值1,2，然后根据id值就可以得到查询的结果数据了。

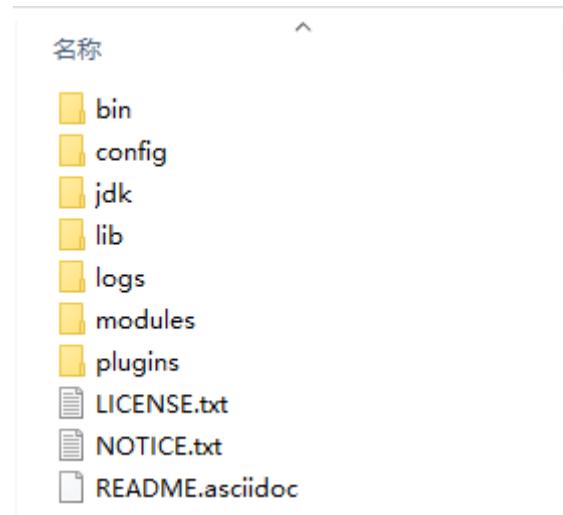
上述过程中分词结果关键字内容每一个都不相同，作用有点类似于数据库中的索引，是用来加速数据查询的。但是数据库中的索引是对某一个字段进行添加索引，而这里的分词结果关键字不是一个完整的字段值，只是一个字段中的其中的一部分内容。并且索引使用时是根据索引内容查找整条数据，全文搜索中的分词结果关键字查询后得到的并不是整条的数据，而是数据的id，要想获得具体数据还要再次查询，因此这里为这种分词结果关键字起了一个全新的名称，叫做**倒排索引**。

通过上述内容的学习，发现使用ES其实准备工作还是挺多的，必须先建立文档的倒排索引，然后才能继续使用。快速了解一下ES的工作原理，下面直接开始我们的学习，老规矩，先安装，再操作，最后说整合。

安装

windows版安装包下载地址：<https://www.elastic.co/cn/downloads/elasticsearch>

下载的安装包是解压缩就能使用的zip文件，解压缩完毕后会得到如下文件



- bin目录：包含所有的可执行命令
- config目录：包含ES服务器使用的配置文件
- jdk目录：此目录中包含了一个完整的jdk工具包，版本17，当ES升级时，使用最新版本的jdk确保不会出现版本支持性不足的问题
- lib目录：包含ES运行的依赖jar文件
- logs目录：包含ES运行后产生的所有日志文件
- modules目录：包含ES软件中所有的功能模块，也是一个一个的jar包。和jar目录不同，jar目录是ES运行期间依赖的jar包，modules是ES软件自己的功能jar包
- plugins目录：包含ES软件安装的插件，默认为空

启动服务器

```
elasticsearch.bat
```

双击elasticsearch.bat文件即可启动ES服务器，默认服务端口9200。通过浏览器访问<http://localhost:9200>看到如下信息视为ES服务器正常启动

```
{
  "name" : "CZBK-*****",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "j137DSSwTPG8U4Yb-0T1Mg",
  "version" : {
    "number" : "7.16.2",
    "build_flavor" : "default",
    "build_type" : "zip",
    "build_hash" : "2b937c44140b6559905130a8650c64dbd0879cfb",
    "build_date" : "2021-12-18T19:42:46.604893745Z",
    "build_snapshot" : false,
    "lucene_version" : "8.10.1",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

基本操作

ES中保存有我们要查询的数据，只不过格式和数据库存储数据格式不同而已。在ES中我们要先创建倒排索引，这个索引的功能又点类似于数据库的表，然后将数据添加到倒排索引中，添加的数据称为文档。所以要进行ES的操作要先创建索引，再添加文档，这样才能进行后续的查询操作。

要操作ES可以通过Rest风格的请求来进行，也就是说发送一个请求就可以执行一个操作。比如新建索引，删除索引这些操作都可以使用发送请求的形式来进行。

- 创建索引，books是索引名称，下同

PUT请求 <http://localhost:9200/books>

发送请求后，看到如下信息即索引创建成功

```
{  
    "acknowledged": true,  
    "shards_acknowledged": true,  
    "index": "books"  
}
```

重复创建已经存在的索引会出现错误信息，reason属性中描述错误原因

```
{  
    "error": {  
        "root_cause": [  
            {  
                "type": "resource_already_exists_exception",  
                "reason": "index [books/VgC_XMVAQmedaiBNsgO2-w] already  
exists",  
                "index_uuid": "VgC_XMVAQmedaiBNsgO2-w",  
                "index": "books"  
            }  
        ],  
        "type": "resource_already_exists_exception",  
        "reason": "index [books/VgC_XMVAQmedaiBNsgO2-w] already exists",  
        "# books索引已经存在"  
        "index_uuid": "VgC_XMVAQmedaiBNsgO2-w",  
        "index": "book"  
    },  
    "status": 400  
}
```

- 查询索引

GET请求 <http://localhost:9200/books>

查询索引得到索引相关信息，如下

```
{  
    "book": {  
        "aliases": {},  
        "mappings": {}  
    }  
}
```

```

    "settings": {
        "index": {
            "routing": {
                "allocation": {
                    "include": {
                        "_tier_preference": "data_content"
                    }
                }
            },
            "number_of_shards": "1",
            "provided_name": "books",
            "creation_date": "1645768584849",
            "number_of_replicas": "1",
            "uuid": "VgC_XMVAQmedaiBNSgo2-w",
            "version": {
                "created": "7160299"
            }
        }
    }
}

```

如果查询了不存在的索引，会返回错误信息，例如查询名称为book的索引后信息如下

```

{
    "error": {
        "root_cause": [
            {
                "type": "index_not_found_exception",
                "reason": "no such index [book]",
                "resource.type": "index_or_alias",
                "resource.id": "book",
                "index_uuid": "_na_",
                "index": "book"
            }
        ],
        "type": "index_not_found_exception",
        "reason": "no such index [book]",          # 没有book索引
        "resource.type": "index_or_alias",
        "resource.id": "book",
        "index_uuid": "_na_",
        "index": "book"
    },
    "status": 404
}

```

- 删除索引

DELETE请求 <http://localhost:9200/books>

删除所有后，给出删除结果

```
{  
    "acknowledged": true  
}
```

如果重复删除，会给出错误信息，同样在reason属性中描述具体的错误原因

```
{  
    "error": {  
        "root_cause": [  
            {  
                "type": "index_not_found_exception",  
                "reason": "no such index [books]",  
                "resource.type": "index_or_alias",  
                "resource.id": "book",  
                "index_uuid": "_na_",  
                "index": "book"  
            }  
,  
            {  
                "type": "index_not_found_exception",  
                "reason": "no such index [books]",      # 没有books索引  
                "resource.type": "index_or_alias",  
                "resource.id": "book",  
                "index_uuid": "_na_",  
                "index": "book"  
            },  
        ],  
        "status": 404  
    }  
}
```

- 创建索引并指定分词器

前面创建的索引是未指定分词器的，可以在创建索引时添加请求参数，设置分词器。目前国内较为流行的分词器是IK分词器，使用前先在下对应的分词器，然后使用。IK分词器下载地址：<https://github.com/medcl/elasticsearch-analysis-ik/releases>

分词器下载后解压到ES安装目录的plugins目录中即可，安装分词器后需要重新启动ES服务器。使用IK分词器创建索引格式：

PUT请求 <http://localhost:9200/books>

请求参数如下（注意是json格式的参数）

```
{  
    "mappings":{  
        "#定义mappings属性，替换创建索引时对应的mappings属性"  
        "properties":{  
            "id":{  
                "#定义索引中包含的属性设置"  
                "type": "keyword"          #设置索引中包含id属性  
            },  
            "name":{  
                "#设置索引中包含name属性"  
                "type": "text",           #当前属性可以被直接搜索  
                "analyzer": "ik_max_word",  
                "copy_to": "all"           #当前属性是文本信息，参与分词  
            },  
            "type":{  
                "#使用IK分词器进行分词"  
                "type": "keyword"         #分词结果拷贝到all属性中  
            },  
        },  
    },  
}
```

```

    "description": {
        "type": "text",
        "analyzer": "ik_max_word",
        "copy_to": "all"
    },
    "all": { #定义属性，用来描述多个字段的分词结果集合，当前属性可以参与查询
        "type": "text",
        "analyzer": "ik_max_word"
    }
}
}
}
}

```

创建完毕后返回结果和不使用分词器创建索引的结果是一样的，此时可以通过查看索引信息观察到添加的请求参数mappings已经进入到了索引属性中

```

{
    "books": {
        "aliases": {},
        "mappings": { #mappings属性已经被替换
            "properties": {
                "all": {
                    "type": "text",
                    "analyzer": "ik_max_word"
                },
                "description": {
                    "type": "text",
                    "copy_to": [
                        "all"
                    ],
                    "analyzer": "ik_max_word"
                },
                "id": {
                    "type": "keyword"
                },
                "name": {
                    "type": "text",
                    "copy_to": [
                        "all"
                    ],
                    "analyzer": "ik_max_word"
                },
                "type": {
                    "type": "keyword"
                }
            }
        },
        "settings": {
            "index": {
                "routing": {
                    "allocation": {
                        "include": {
                            "_tier_preference": "data_content"
                        }
                    }
                }
            }
        }
    }
}
```

```
        }
    },
    "number_of_shards": "1",
    "provided_name": "books",
    "creation_date": "1645769809521",
    "number_of_replicas": "1",
    "uuid": "DohYKvr_SZ04KRGmbZYmTQ",
    "version": {
        "created": "7160299"
    }
}
}
```

目前我们已经有了索引了，但是索引中还没有数据，所以要先添加数据，ES中称数据为文档，下面进行文档操作。

- 添加文档，有三种方式

```
POST请求 http://localhost:9200/books/_doc          #使用系统生成id  
POST请求 http://localhost:9200/books/_create/1      #使用指定id  
POST请求 http://localhost:9200/books/_doc/1          #使用指定id, 不存在创建, 存在更新(版本递增)
```

文档通过请求参数传递，数据格式json

```
{  
    "name": "springboot",  
    "type": "springboot",  
    "description": "springboot"  
}
```

- 查询文档

```
GET请求 http://localhost:9200/books/_doc/1          #查询单个文档  
GET请求 http://localhost:9200/books/_search          #查询全部文档
```

- 条件查询

```
GET请求 http://localhost:9200/books/_search?q=name:springboot # q=查询属性名:查询属性值
```

- 删除文档

```
DELETE请求 http://localhost:9200/books/_doc/1
```

- 修改文档（全量更新）

PUT请求 http://localhost:9200/books/_doc/1

文档通过请求参数传递，数据格式json

```
{  
    "name": "springboot",  
    "type": "springboot",  
    "description": "springboot"  
}
```

- 修改文档 (部分更新)

POST请求 http://localhost:9200/books/_update/1

文档通过请求参数传递，数据格式json

```
{  
    "doc": {  
        "name": "springboot"      #部分更新并不是对原始文档进行更新，而是对原始文档对象  
    }                          #仅更新提供的属性值，未提供的属性值不参与更新操作  
}
```

整合

使用springboot整合ES该如何进行呢？老规矩，导入坐标，做配置，使用API接口操作。整合Redis如此，整合MongoDB如此，整合ES依然如此。太没有新意了，其实不是没有新意，这就是springboot的强大之处，所有东西都做成相同规则，对开发者来说非常友好。

下面就开始springboot整合ES，操作步骤如下：

步骤①：导入springboot整合ES的starter坐标

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

步骤②：进行基础配置

```
spring:  
  elasticsearch:  
    rest:  
      uris: http://localhost:9200
```

配置ES服务器地址，端口9200

步骤③：使用springboot整合ES的专用客户端接口ElasticsearchRestTemplate来进行操作

```
@SpringBootTest  
class Springboot18EsApplicationTests {  
    @Autowired  
    private ElasticsearchRestTemplate template;  
}
```

上述操作形式是ES早期的操作方式，使用的客户端被称为Low Level Client，这种客户端操作方式性能方面略显不足，于是ES开发了全新的客户端操作方式，称为High Level Client。高级别客户端与ES版本同步更新，但是springboot最初整合ES的时候使用的是低级别客户端，所以企业开发需要更换成高级别的客户端模式。

下面使用高级别客户端方式进行springboot整合ES，操作步骤如下：

步骤①：导入springboot整合ES高级别客户端的坐标，此种形式目前没有对应的starter

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>
</dependency>
```

步骤②：使用编程的形式设置连接的ES服务器，并获取客户端对象

```
@SpringBootTest
class Springboot18EsApplicationTests {
    private RestHighLevelClient client;

    @Test
    void testCreateClient() throws IOException {
        HttpHost host = HttpHost.create("http://localhost:9200");
        RestClientBuilder builder = RestClient.builder(host);
        client = new RestHighLevelClient(builder);

        client.close();
    }
}
```

配置ES服务器地址与端口9200，记得客户端使用完毕需要手工关闭。由于当前客户端是手工维护的，因此不能通过自动装配的形式加载对象。

步骤③：使用客户端对象操作ES，例如创建索引

```
@SpringBootTest
class Springboot18EsApplicationTests {
    private RestHighLevelClient client;

    @Test
    void testCreateIndex() throws IOException {
        HttpHost host = HttpHost.create("http://localhost:9200");
        RestClientBuilder builder = RestClient.builder(host);
        client = new RestHighLevelClient(builder);

        CreateIndexRequest request = new CreateIndexRequest("books");
        client.indices().create(request, RequestOptions.DEFAULT);

        client.close();
    }
}
```

高级别客户端操作是通过发送请求的方式完成所有操作的，ES针对各种不同的操作，设定了各式各样的请求对象，上例中创建索引的对象是CreateIndexRequest，其他操作也会有自己专用的Request对象。

当前操作我们发现，无论进行ES何种操作，第一步永远是获取RestHighLevelClient对象，最后一步永远是关闭该对象的连接。在测试中可以使用测试类的特性去帮助开发者一次性的完成上述操作，但是在业务书写时，还需要自行管理。将上述代码格式转换成使用测试类的初始化方法和销毁方法进行客户端对象的维护。

```
@SpringBootTest
class Springboot18EsApplicationTests {
    @BeforeEach      //在测试类中每个操作运行前运行的方法
    void setup() {
        HttpHost host = HttpHost.create("http://localhost:9200");
        RestClientBuilder builder = RestClient.builder(host);
        client = new RestHighLevelClient(builder);
    }

    @AfterEach      //在测试类中每个操作运行后运行的方法
    void tearDown() throws IOException {
        client.close();
    }

    private RestHighLevelClient client;

    @Test
    void testCreateIndex() throws IOException {
        CreateIndexRequest request = new CreateIndexRequest("books");
        client.indices().create(request, RequestOptions.DEFAULT);
    }
}
```

现在的书写简化了很多，也更合理。下面使用上述模式将所有的ES操作执行一遍，测试结果

创建索引（IK分词器）：

```
@Test
void testCreateIndexByIK() throws IOException {
    CreateIndexRequest request = new CreateIndexRequest("books");
    String json = "{\n" +
        "    \"mappings\":{\n" +
        "        \"properties\":{\n" +
        "            \"id\":{\n" +
        "                \"type\":\"keyword\"\n" +
        "            },\n" +
        "            \"name\":{\n" +
        "                \"type\":\"text\",\n" +
        "                \"analyzer\":\"ik_max_word\",\n" +
        "                \"copy_to\":\"all\"\n" +
        "            },\n" +
        "            \"type\":{\n" +
        "                \"type\":\"keyword\"\n" +
        "            },\n" +
        "            \"description\":{\n" +
        "                \"type\":\"text\",\n" +
        "                \"analyzer\":\"ik_max_word\",\n" +
        "                \"copy_to\":\"all\"\n" +
        "            }\n" +
    "
```

```

        "      \\"all\\":{\n" +
        "          \\"type\\":\\"text\\",\n" +
        "          \\"analyzer\\":\\"ik_max_word\\\"\n" +
        "      }\n" +
        "  }\n" +
    "}";
//设置请求中的参数
request.source(json, XContentType.JSON);
client.indices().create(request, RequestOptions.DEFAULT);
}

```

IK分词器是通过请求参数的形式进行设置的，设置请求参数使用request对象中的source方法进行设置，至于参数是什么，取决于你的操作种类。当请求中需要参数时，均可使用当前形式进行参数设置。

添加文档：

```

@Test
//添加文档
void testCreateDoc() throws IOException {
    Book book = bookDao.selectById(1);
    IndexRequest request = new
IndexRequest("books").id(book.getId().toString());
    String json = JSON.toJSONString(book);
    request.source(json,XContentType.JSON);
    client.index(request,RequestOptions.DEFAULT);
}

```

添加文档使用的请求对象是IndexRequest，与创建索引使用的请求对象不同。

批量添加文档：

```

@Test
//批量添加文档
void testCreateDocAll() throws IOException {
    List<Book> bookList = bookDao.selectList(null);
    BulkRequest bulk = new BulkRequest();
    for (Book book : bookList) {
        IndexRequest request = new
IndexRequest("books").id(book.getId().toString());
        String json = JSON.toJSONString(book);
        request.source(json,XContentType.JSON);
        bulk.add(request);
    }
    client.bulk(bulk,RequestOptions.DEFAULT);
}

```

批量做时，先创建一个BulkRequest的对象，可以将该对象理解为是一个保存request对象的容器，将所有的请求都初始化好后，添加到BulkRequest对象中，再使用BulkRequest对象的bulk方法，一次性执行完毕。

按id查询文档：

```

@Test
//按id查询
void testGet() throws IOException {
    GetRequest request = new GetRequest("books", "1");
    GetResponse response = client.get(request, RequestOptions.DEFAULT);
    String json = response.getSourceAsString();
    System.out.println(json);
}

```

根据id查询文档使用的请求对象是GetRequest。

按条件查询文档：

```

@Test
//按条件查询
void testSearch() throws IOException {
    SearchRequest request = new SearchRequest("books");

    SearchSourceBuilder builder = new SearchSourceBuilder();
    builder.query(QueryBuilders.termQuery("all", "spring"));
    request.source(builder);

    SearchResponse response = client.search(request, RequestOptions.DEFAULT);
    SearchHits hits = response.getHits();
    for (SearchHit hit : hits) {
        String source = hit.getSourceAsString();
        //System.out.println(source);
        Book book = JSON.parseObject(source, Book.class);
        System.out.println(book);
    }
}

```

按条件查询文档使用的请求对象是SearchRequest，查询时调用SearchRequest对象的termQuery方法，需要给出查询属性名，此处支持使用合并字段，也就是前面定义索引属性时添加的all属性。

springboot整合ES的操作到这里就说完了，与前期进行springboot整合redis和mongodb的差别还是蛮大的，主要原始就是我们没有使用springboot整合ES的客户端对象。至于操作，由于ES操作种类过多，所以显得操作略微有点复杂。有关springboot整合ES就先学习到这里吧。

总结

1. springboot整合ES步骤

1. 导入springboot整合ES的High Level Client坐标
2. 手工管理客户端对象，包括初始化和关闭操作
3. 使用High Level Client根据操作的种类不同，选择不同的Request对象完成对应操作

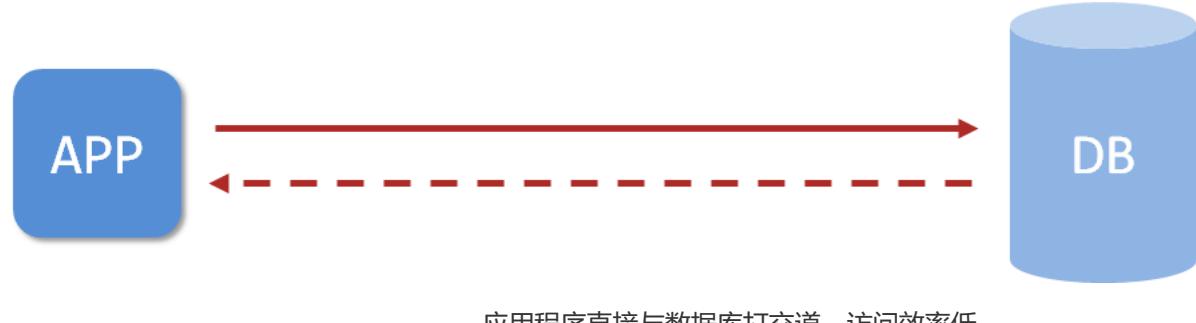
KF-5.整合第三方技术

通过第四章的学习，我们领略到了springboot在整合第三方技术时强大的一致性，在第五章中我们要使用springboot继续整合各种各样的第三方技术，通过本章的学习，可以将之前学习的springboot整合第三方技术的思想贯彻到底，还是那三板斧。导坐标、做配置、调API。

springboot能够整合的技术实在是太多了，可以说是万物皆可整。本章将从企业级开发中常用的一些技术作为出发点，对各种各样的技术进行整合。

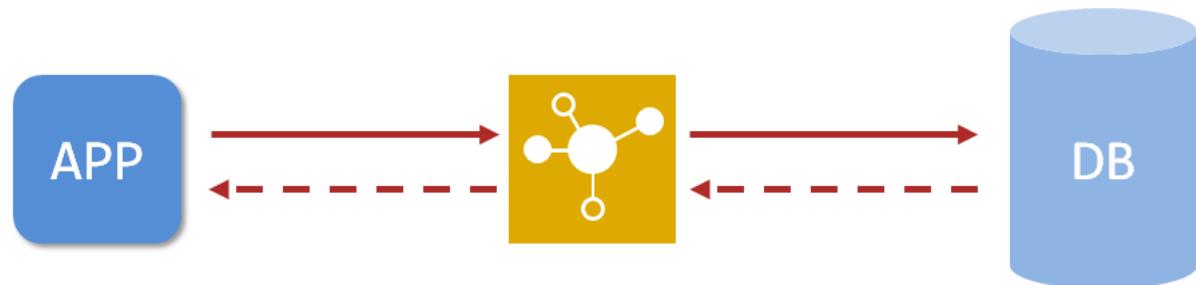
KF-5-1.缓存

企业级应用主要作用是信息处理，当需要读取数据时，由于受限于数据库的访问效率，导致整体系统性能偏低。



应用程序直接与数据库打交道，访问效率低

为了改善上述现象，开发者通常会在应用程序与数据库之间建立一种临时的数据存储机制，该区域中的数据在内存中保存，读写速度较快，可以有效解决数据库访问效率低下的问题。这一块临时存储数据的区域就是缓存。



使用缓存后，应用程序与缓存打交道，缓存与数据库打交道，数据访问效率提高

缓存是什么？缓存是一种介于数据永久存储介质与应用程序之间的数据临时存储介质，使用缓存可以有效的减少低速数据读取过程的次数（例如磁盘IO），提高系统性能。此外缓存不仅可以用于提高永久性存储介质的数据读取效率，还可以提供临时的数据存储空间。而springboot提供了对市面上几乎所有的缓存技术进行整合的方案，下面就一起开启springboot整合缓存之旅。

SpringBoot内置缓存解决方案

springboot技术提供有**内置**（这个就是spring-cache技术，在瑞吉外卖里面我详细的记录过）的缓存解决方案，可以帮助开发者快速开启缓存技术，并使用缓存技术进行数据的快速操作，例如读取缓存数据和写入数据到缓存。

步骤①：导入springboot提供的缓存技术对应的starter

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

步骤②：启用缓存，在引导类上方标注注解@EnableCaching配置springboot程序中可以使用缓存

```
@SpringBootApplication
//开启缓存功能
@EnableCaching
public class Springboot19CacheApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot19CacheApplication.class, args);
    }
}
```

步骤③：设置操作的数据是否使用缓存

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;

    @Cacheable(value="cacheSpace", key="#id")
    public Book getById(Integer id) {
        return bookDao.selectById(id);
    }
}
```

在业务方法上面使用注解@Cacheable声明当前方法的返回值放入缓存中，其中要指定缓存的存储位置，以及缓存中保存当前方法返回值对应的名称。上例中value属性描述缓存的存储位置，可以理解为是一个存储空间名，key属性描述了缓存中保存数据的名称，使用#id读取形参中的id值作为缓存名称。

使用@Cacheable注解后，执行当前操作，如果发现对应名称在缓存中没有数据，就正常读取数据，然后放入缓存；如果对应名称在缓存中有数据，就终止当前业务方法执行，直接返回缓存中的数据。

手机验证码案例

为了便于下面演示各种各样的缓存技术，我们创建一个手机验证码的案例环境，模拟使用缓存保存手机验证码的过程。

手机验证码案例需求如下：

- 输入手机号获取验证码，组织文档以短信形式发送给用户（页面模拟）
- 输入手机号和验证码验证结果

为了描述上述操作，我们制作两个表现层接口，一个用来模拟发送短信的过程，其实就是根据用户提供的手机号生成一个验证码，然后放入缓存，另一个用来模拟验证码校验的过程，其实就是使用传入的手机号和验证码进行匹配，并返回最终匹配结果。下面直接制作本案例的模拟代码，先以上例中springboot提供的内置缓存技术来完成当前案例的制作。

步骤①：导入springboot提供的缓存技术对应的starter

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

步骤②：启用缓存，在引导类上方标注注解@EnableCaching配置springboot程序中可以使用缓存

```
@SpringBootApplication  
//开启缓存功能  
@EnableCaching  
public class Springboot19CacheApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(Springboot19CacheApplication.class, args);  
    }  
}
```

步骤③：定义验证码对应的实体类，封装手机号与验证码两个属性

```
@Data  
public class SMSCode {  
    private String tele;  
    private String code;  
}
```

步骤④：定义验证码功能的业务层接口与实现类

```
public interface SMSCodeService {  
    public String sendCodeToSMS(String tele);  
    public boolean checkCode(SMSCode smsCode);  
}  
  
@Service  
public class SMSCodeServiceImpl implements SMSCodeService {  
    @Autowired  
    private Codeutils codeutils;  
  
    @CachePut(value = "smsCode", key = "#tele")  
    public String sendCodeToSMS(String tele) {  
        String code = codeutils.generator(tele);  
        return code;  
    }  
  
    public boolean checkCode(SMSCode smsCode) {  
        //取出内存中的验证码与传递过来的验证码比对，如果相同，返回true  
        String code = smsCode.getCode();  
        //huo'd  
        String cacheCode = codeutils.get(smsCode.getTele());  
        return code.equals(cacheCode);  
    }  
}
```

获取验证码后，当验证码失效时必须重新获取验证码，因此在获取验证码的功能上不能使用@Cacheable注解，@Cacheable注解是缓存中没有值则放入值，缓存中有值则取值。此处的功能仅仅是生成验证码并放入缓存，并不具有从缓存中取值的功能，因此不能使用@Cacheable注解，应该使用仅具有向缓存中保存数据的功能，使用@CachePut注解即可。

对于校验验证码的功能建议放入工具类中进行。

步骤⑤：定义验证码的生成策略与根据手机号读取验证码的功能

```

@Component
public class Codeutils {
    //用来补零，但是我感觉没必要，直接生成string就行
    private String [] patch = {"000000", "00000", "0000", "000", "00", "0", ""};

    //生成验证码（但是我感觉不用那么麻烦...）
    public String generator(String tele){
        int hash = tele.hashCode();
        int encryption = 20206666;
        long result = hash ^ encryption;
        long nowTime = System.currentTimeMillis();
        result = result ^ nowTime;
        long code = result % 1000000;
        code = code < 0 ? -code : code;
        String codeStr = code + "";
        int len = codeStr.length();
        return patch[len] + codeStr;
    }

    //得到缓存中的code
    @Cacheable(value = "smsCode", key="#tele")
    public String get(String tele){
        //如果有需求需要取到缓存中的数据，这里直接返回null就行
        return null;
    }
}

```

步骤⑥： 定义验证码功能的web层接口，一个方法用于提供手机号获取验证码，一个方法用于提供手机号和验证码进行校验

```

@RestController
@RequestMapping("/sms")
public class SMSCodeController {
    @Autowired
    private SMSCodeService smsCodeService;

    @GetMapping
    public String getCode(String tele){
        String code = smsCodeService.sendCodeToSMS(tele);
        return code;
    }

    @PostMapping
    public boolean checkCode(SMSCode smsCode){
        return smsCodeService.checkCode(smsCode);
    }
}

```

SpringBoot整合Ehcache缓存

手机验证码的案例已经完成了，下面就开始springboot整合各种各样的缓存技术，第一个整合Ehcache技术。Ehcache是一种缓存技术，使用springboot整合Ehcache其实只是变更一下缓存技术的实现方式，话不多说，直接开整

步骤①：导入Ehcache的坐标

```
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
</dependency>
```

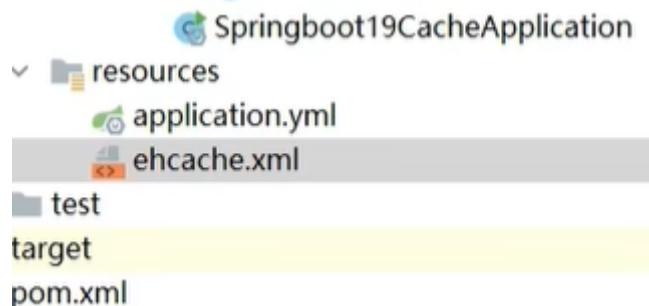
此处为什么不是导入Ehcache的starter，而是导入技术坐标呢？其实springboot整合缓存技术做的是通用格式，不管你整合哪种缓存技术，只是实现变化了，操作方式一样。这也体现出springboot技术的优点，统一同类技术的整合方式。

步骤②：配置缓存技术实现使用Ehcache

```
spring:
  cache:
    type: ehcache
    ehcache:
      config: ehcache.xml
```

配置缓存的类型type为ehcache，此处需要说明一下，当前springboot可以整合的缓存技术中包含有ehcach，所以可以这样书写。其实这个type不可以随便写的，不是随便写一个名称就可以整合的。

由于ehcache的配置有独立的配置文件格式，因此还需要指定ehcache的配置文件，以便于读取相应配置



```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
  updateCheck="false">
  <diskStore path="D:\ehcache" />

  <!--默认缓存策略 -->
  <!-- external: 是否永久存在，设置为true则不会被清除，此时与timeout冲突，通常设置为false-->
  <!-- diskPersistent: 是否启用磁盘持久化-->
  <!-- maxElementsInMemory: 最大缓存数量-->
  <!-- overflowToDisk: 超过最大缓存数量是否持久化到磁盘-->
  <!-- timeToIdleSeconds: 最大不活动间隔，设置过长缓存容易溢出，设置过短无效果，可用于记录时效性数据，例如验证码-->
```

```

<!-- timeToLiveSeconds: 最大存活时间-->
<!-- memoryStoreEvictionPolicy: 缓存清除策略-->
<defaultCache
    eternal="false"
    diskPersistent="false"
    maxElementsInMemory="1000"
    overflowToDisk="false"
    timeToIdleSeconds="60"
    timeToLiveSeconds="60"
    memoryStoreEvictionPolicy="LRU" />

<!--注意写这个，这个的含义是和上面代码中的smsCode相对应的，如果没有下面这个标签会报错的-->
>
<cache
    name="smsCode"
    eternal="false"
    diskPersistent="false"
    maxElementsInMemory="1000"
    overflowToDisk="false"
    timeToIdleSeconds="10"
    timeToLiveSeconds="10"
    memoryStoreEvictionPolicy="LRU" />
</ehcache>

```

注意前面的案例中，设置了数据保存的位置是smsCode

```

@CachePut(value = "smsCode", key = "#tele")
public String sendCodeToSMS(String tele) {
    String code = codeutils.generator(tele);
    return code;
}

```

这个设定需要保障ehcache中有一个缓存空间名称叫做smsCode的配置，前后要统一。在企业开发过程中，通过设置不同名称的cache来设定不同的缓存策略，应用于不同的缓存数据。

到这里springboot整合Ehcache就做完了，可以发现一点，原始代码没有任何修改，仅仅是加了一组配置就可以变更缓存供应商了，这也是springboot提供了统一的缓存操作接口的优势，变更实现并不影响原始代码的书写。

总结

1. springboot使用Ehcache作为缓存实现需要导入Ehcache的坐标
2. 修改设置，配置缓存供应商为ehcache，并提供对应的缓存配置文件

SpringBoot整合Redis缓存

上节使用Ehcache替换了springboot内置的缓存技术，其实springboot支持的缓存技术还很多，下面使用redis技术作为缓存解决方案来实现手机验证码案例。

比对使用Ehcache的过程，加坐标，改缓存实现类型为ehcache，做Ehcache的配置。如果还想用redis做缓存呢？一模一样，加坐标，改缓存实现类型为redis，做redis的配置。差别之处只有一点，redis的配置可以在yml文件中直接进行配置，无需制作独立的配置文件。

步骤①：导入redis的坐标

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

步骤②：配置缓存技术实现使用redis

```
spring:
  redis:
    host: localhost
    port: 6379
  cache:
    type: redis
```

如果需要对redis作为缓存进行配置，注意不是对原始的redis进行配置，而是配置redis作为缓存使用相关的配置，隶属于spring.cache.redis节点下，注意不要写错位置了。

```
spring:
  redis:
    host: localhost
    port: 6379
  cache:
    type: redis
    redis:
      use-key-prefix: false
      key-prefix: sms_
      cache-null-values: false
      time-to-live: 10s
```

总结

1. springboot使用redis作为缓存实现需要导入redis的坐标
2. 修改设置，配置缓存供应商为redis，并提供对应的缓存配置

SpringBoot整合Memcached缓存

目前我们已经掌握了3种缓存解决方案的配置形式，分别是springboot内置缓存，ehcache和redis，本节研究一下国内比较流行的一款缓存memcached。

按照之前的套路，其实变更缓存并不繁琐，但是springboot并没有支持使用memcached作为其缓存解决方案，也就是说在type属性中没有memcached的配置选项，这里就需要变更一下处理方式了。在整合之前先安装memcached。

安装

windows版安装包下载地址：<https://www.runoob.com/memcached/window-install-memcached.html>

下载的安装包是解压缩就能使用的zip文件，解压缩完毕后会得到如下文件



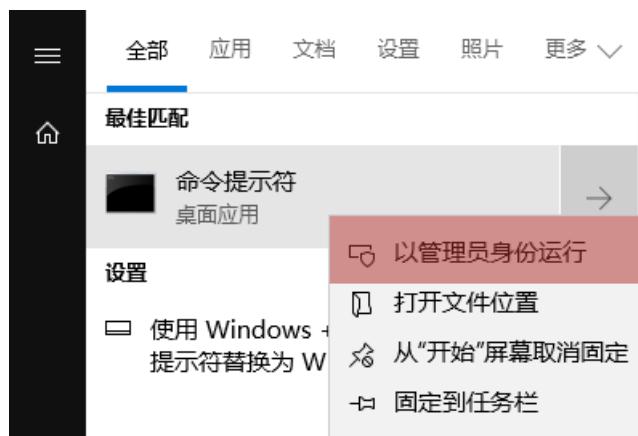
可执行文件只有一个memcached.exe，使用该文件可以将memcached作为系统服务启动，执行此文件时会出现报错信息，如下：

```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.17763.2565]
(c) 2018 Microsoft Corporation。保留所有权利。

D:\soft\memcached>memcached.exe -d install
failed to install service or service already installed

D:\soft\memcached>
```

此处出现问题的原因是注册系统服务时需要使用管理员权限，当前账号权限不足导致安装服务失败，切换管理员账号权限启动命令行



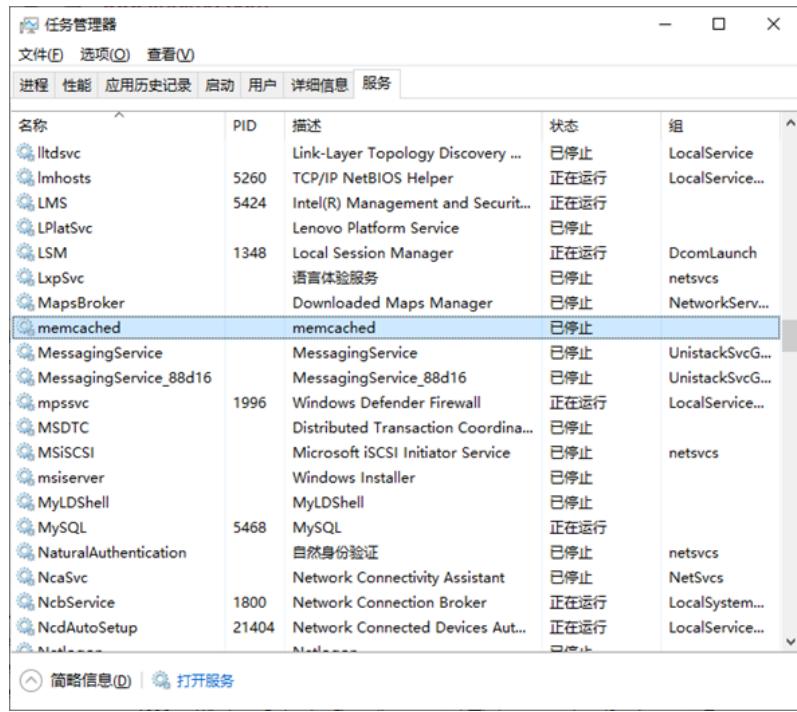
然后再次执行安装服务的命令即可，如下：

```
memcached.exe -d install
```

服务安装完毕后可以使用命令启动和停止服务，如下：

```
memcached.exe -d start      # 启动服务
memcached.exe -d stop       # 停止服务
```

也可以在任务管理器中进行服务状态的切换



变更缓存为Memcached

由于memcached未被springboot收录为缓存解决方案，因此使用memcached需要通过手工硬编码的方式来使用，于是前面的套路都不适用了，需要自己写了。

memcached目前提供有三种客户端技术，分别是Memcached Client for Java、SpyMemcached和Xmemcached，其中性能指标各方面最好的客户端是Xmemcached，本次整合就使用这个作为客户端实现技术了。下面开始使用Xmemcached

步骤①：导入xmemcached的坐标

```
<dependency>
    <groupId>com.googlecode.xmemcached</groupId>
    <artifactId>xmemcached</artifactId>
    <version>2.4.7</version>
</dependency>
```

步骤②：（因为springboot没有提供对memcached的整合，所以这里我们直接不编写配置文件，直接写配置类）配置memcached，制作memcached的配置类

```
@Configuration
public class XMemcachedConfig {
    @Bean
    public MemcachedClient getMemcachedClient() throws IOException {
        MemcachedClientBuilder memcachedClientBuilder = new
XMemcachedClientBuilder("localhost:11211");
        MemcachedClient memcachedClient = memcachedClientBuilder.build();
        return memcachedClient;
    }
}
```

memcached默认对外服务端口11211。

步骤③：使用xmemcached客户端操作缓存，注入MemcachedClient对象

```

@Service
public class SMSCodeServiceImpl implements SMSCodeService {
    @Autowired
    private Codeutils codeutils;
    @Autowired
    private MemcachedClient memcachedClient;

    public String sendCodeToSMS(String tele) {
        String code = codeutils.generator(tele);
        try {
            memcachedClient.set(tele, 10, code);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return code;
    }

    public boolean checkCode(SMSCode smsCode) {
        String code = null;
        try {
            code = memcachedClient.get(smsCode.getTele()).toString();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return smsCode.getCode().equals(code);
    }
}

```

设置值到缓存中使用set操作，取值使用get操作，其实更符合我们开发者的习惯。

上述代码中对于服务器的配置使用硬编码写死到了代码中，将此数据提取出来，做成独立的配置属性。

定义配置属性

以下过程采用前期学习的属性配置方式进行，当前操作有助于理解原理篇中的很多知识。

- 定义配置类，加载必要的配置属性，读取配置文件中memcached节点信息

```

@Component
@ConfigurationProperties(prefix = "memcached")
@Data
public class XMemcachedProperties {
    private String servers;
    private int poolsize;
    private long opTimeout;
}

```

- 定义memcached节点信息

```

memcached:
  servers: localhost:11211
  poolsize: 10
  opTimeout: 3000

```

- 在memcached配置类中加载信息

```

@Configuration
public class XMemcachedConfig {
    @Autowired
    private XMemcachedProperties props;
    @Bean
    public MemcachedClient getMemcachedClient() throws IOException {
        MemcachedClientBuilder memcachedClientBuilder = new
XMemcachedClientBuilder(props.getServers());
        memcachedClientBuilder.setConnectionPoolSize(props.getPoolSize());
        memcachedClientBuilder.setOpTimeout(props.getOpTimeout());
        MemcachedClient memcachedClient = memcachedClientBuilder.build();
        return memcachedClient;
    }
}

```

总结

1. memcached安装后需要启动对应服务才可以对外提供缓存功能，安装memcached服务需要基于windows系统管理员权限
2. 由于springboot没有提供对memcached的缓存整合方案，需要采用手工编码的形式创建xmemcached客户端操作缓存
3. 导入xmemcached坐标后，创建memcached配置类，注册MemcachedClient对应的bean，用于操作缓存
4. 初始化MemcachedClient对象所需要使用的属性可以通过自定义配置属性类的形式加载

思考

到这里已经完成了三种缓存的整合，其中redis和mongodb需要安装独立的服务器，连接时需要输入对应的服务器地址，这种是远程缓存，Ehcache是一个典型的内存级缓存，因为它什么也不用安装，启动后导入jar包就有缓存功能了。这个时候就要问了，能不能这两种缓存一起用呢？咱们下节再说。

SpringBoot整合jetcache缓存

目前我们使用的缓存都是要么A要么B，能不能AB一起用呢？这一节就解决这个问题。springboot针对缓存的整合仅仅停留在用缓存上面，如果缓存自身不支持同时支持AB一起用，springboot也没办法，所以要想解决AB缓存一起用的问题，就必须找一款缓存能够支持AB两种缓存一起用，有这种缓存吗？还真有，阿里出品，jetcache。

jetcache严格意义上来说，并不是一个缓存解决方案，只能说他算是一个缓存框架，然后把别的缓存放到jetcache中管理，这样就可以支持AB缓存一起用了。并且jetcache参考了springboot整合缓存的思想，整体技术使用方式和springboot的缓存解决方案思想非常类似。下面咱们就先把jetcache用起来，然后再谈它里面的一些小的功能。

做之前要先明确一下，jetcache并不是随便拿两个缓存都能拼到一起去的。目前jetcache支持的缓存方案本地缓存支持两种，远程缓存支持两种，分别如下：

- 本地缓存 (Local)
 - LinkedHashMap
 - Caffeine
- 远程缓存 (Remote)
 - Redis

- Tair

其实也有人问我，为什么jetcache只支持2+2这么4款缓存呢？阿里研发这个技术其实主要是为了满足自身的使用需要。最初肯定只有1+1种，逐步变化成2+2种。下面就以LinkedHashMap+Redis的方案实现本地与远程缓存方案同时使用。

纯远程方案

步骤①：导入springboot整合jetcache对应的坐标starter，当前坐标默认使用的远程方案是redis

```
<dependency>
    <groupId>com.alicp.jetcache</groupId>
    <artifactId>jetcache-starter-redis</artifactId>
    <version>2.6.2</version>
</dependency>
```

步骤②：远程方案基本配置

```
jetcache:
  remote:
    default:
      type: redis
      host: localhost
      port: 6379
      poolConfig:
        maxTotal: 50
```

其中poolConfig是必配项，否则会报错

步骤③：启用缓存，在引导类上方标注注解@EnableCreateCacheAnnotation配置springboot程序中可以使用注解的形式创建缓存

```
@SpringBootApplication
//jetcache启用缓存的主开关
@EnableCreateCacheAnnotation
public class Springboot20JetCacheApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot20JetCacheApplication.class, args);
    }
}
```

步骤④：创建缓存对象Cache，并使用注解@CreateCache标记当前缓存的信息，然后使用Cache对象的API操作缓存，put写缓存，get读缓存。

```
@Service
public class SMSCodeServiceImpl implements SMSCodeService {
    @Autowired
    private Codeutils codeutils;

    @CreateCache(name="jetCache_", expire = 10, timeUnit = TimeUnit.SECONDS)
    private Cache<String ,String> jetCache;

    public String sendCodeToSMS(String tele) {
        String code = codeutils.generator(tele);
```

```

        jetCache.put(tele, code);
        return code;
    }

    public boolean checkCode(SMSCode smsCode) {
        String code = jetCache.get(smsCode.getTele());
        return smsCode.getCode().equals(code);
    }
}

```

通过上述jetcache使用远程方案连接redis可以看出，jetcache操作缓存时的接口操作更符合开发者习惯，使用缓存就先获取缓存对象Cache，放数据进去就是put，取数据出来就是get，更加简单易懂。并且jetcache操作缓存时，可以为某个缓存对象设置过期时间，将同类型的数据放入缓存中，方便有效周期的管理。

上述方案中使用的是配置中定义的default缓存，其实这个default是个名字，可以随便写，也可以随便加。例如再添加一种缓存解决方案，参照如下配置进行：

```

jetcache:
  remote:
    default:
      type: redis
      host: localhost
      port: 6379
      poolConfig:
        maxTotal: 50
    sms:
      type: redis
      host: localhost
      port: 6379
      poolConfig:
        maxTotal: 50

```

如果想使用名称是sms的缓存，需要再创建缓存时指定参数area，声明使用对应缓存即可

```

@Service
public class SMScodeServiceImpl implements SMSCodeService {
    @Autowired
    private Codeutils codeUtils;

    @CreateCache(area="sms",name="jetCache_",expire = 10,timeUnit =
TimeUnit.SECONDS)
    private Cache<String ,String> jetCache;

    public String sendCodeToSMS(String tele) {
        String code = codeUtils.generator(tele);
        jetCache.put(tele,code);
        return code;
    }

    public boolean checkCode(SMSCode smsCode) {
        String code = jetCache.get(smsCode.getTele());
        return smsCode.getCode().equals(code);
    }
}

```

```
}
```

纯本地方案

远程方案中，配置中使用remote表示远程，换成local就是本地，只不过类型不一样而已。

步骤①：导入springboot整合jetcache对应的坐标starter

```
<dependency>
    <groupId>com.alicp.jetcache</groupId>
    <artifactId>jetcache-starter-redis</artifactId>
    <version>2.6.2</version>
</dependency>
```

步骤②：本地缓存基本配置

```
jetcache:
  local:
    default:
      type: linkedhashmap
      keyConvertor: fastjson
```

为了加速数据获取时key的匹配速度，jetcache要求指定key的类型转换器。简单说就是，如果你给了一个Object作为key的话，我先用key的类型转换器给转换成字符串，然后再保存。等到获取数据时，仍然是先使用给定的Object转换成字符串，然后根据字符串匹配。由于jetcache是阿里的技术，这里推荐key的类型转换器使用阿里的fastjson。

步骤③：启用缓存

```
@SpringBootApplication
//jetcache启用缓存的主开关
@EnableCreateCacheAnnotation
public class Springboot20JetCacheApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot20JetCacheApplication.class, args);
    }
}
```

步骤④：创建缓存对象Cache时，标注当前使用本地缓存

```
@Service
public class SmscodeServiceImpl implements SmscodeService {
    @CreateCache(name = "jetCache_", expire = 1000, timeUnit =
TimeUnit.SECONDS, cacheType = CacheType.LOCAL)
    private Cache<String ,String> jetCache;

    public String sendCodeToSMS(String tele) {
        String code = codeutils.generator(tele);
        jetCache.put(tele,code);
        return code;
    }

    public boolean checkCode(SMSCode smsCode) {
        String code = jetCache.get(smsCode.getTele());
```

```
        return smsCode.getCode().equals(code);
    }
}
```

cacheType控制当前缓存使用本地缓存还是远程缓存，配置cacheType=CacheType.LOCAL即使用本地缓存。

本地+远程方案

本地和远程方法都有了，两种方案一起使用如何配置呢？其实就是将两种配置合并到一起就可以了。

```
jetcache:
  local:
    default:
      type: linkedhashmap
      keyConvertor: fastjson
  remote:
    default:
      type: redis
      host: localhost
      port: 6379
      poolConfig:
        maxTotal: 50
  sms:
    type: redis
    host: localhost
    port: 6379
    poolConfig:
      maxTotal: 50
```

在创建缓存的时候，配置cacheType为BOTH即则本地缓存与远程缓存同时使用。

```
@Service
public class SMSCodeServiceImpl implements SMSCodeService {
    @CreateCache(name="jetCache_", expire = 1000, timeUnit =
TimeUnit.SECONDS, cacheType = CacheType.BOTH)
    private Cache<String ,String> jetCache;
}
```

cacheType如果不进行配置，默认值是REMOTE，即仅使用远程缓存方案。关于jetcache的配置，参考以下信息

属性	默认值	说明
jetcache.statIntervalMinutes	0	统计间隔, 0表示不统计
jetcache.hiddenPackages	无	自动生成name时, 隐藏指定的包名前缀
jetcache.[local remote].\${area}.type	无	缓存类型, 本地支持 linkedhashmap、caffeine, 远程 支持redis、tair
jetcache.[local remote].\${area}.keyConvertor	无	key转换器, 当前仅支持fastjson
jetcache.[local remote].\${area}.valueEncoder	java	仅remote类型的缓存需要指定, 可 选java和kryo
jetcache.[local remote].\${area}.valueDecoder	java	仅remote类型的缓存需要指定, 可 选java和kryo
jetcache.[local remote].\${area}.limit	100	仅local类型的缓存需要指定, 缓存 实例最大元素数
jetcache. [local remote].\${area}.expireAfterWriteInMillis	无穷大	默认过期时间, 毫秒单位
jetcache.local.\${area}.expireAfterAccessInMillis	0	仅local类型的缓存有效, 毫秒单 位, 最大不活动间隔

以上方案仅支持手工控制缓存, 但是springcache方案中的方法缓存特别好用, 给一个方法添加一个注解, 方法就会自动使用缓存。jetcache也提供了对应的功能, 即方法缓存。

方法缓存

jetcache提供了方法缓存方案, 只不过名称变更了而已。在对应的操作接口上方使用注解@Cached即可

步骤①: 导入springboot整合jetcache对应的坐标starter

```
<dependency>
    <groupId>com.alicp.jetcache</groupId>
    <artifactId>jetcache-starter-redis</artifactId>
    <version>2.6.2</version>
</dependency>
```

步骤②：配置缓存

```
jetcache:
  local:
    default:
      type: linkedhashmap
      keyConvertor: fastjson
  remote:
    default:
      type: redis
      host: localhost
      port: 6379
      keyConvertor: fastjson
      valueEncode: java
      valueDecode: java
      poolConfig:
        maxTotal: 50
  sms:
    type: redis
    host: localhost
    port: 6379
    poolConfig:
      maxTotal: 50
```

由于redis缓存中不支持保存对象，因此需要对redis设置当Object类型数据进入到redis中时如何进行类型转换。需要配置keyConvertor表示key的类型转换方式，同时标注value的转换类型方式，值进入redis时是java类型，标注valueEncode为java，值从redis中读取时转换成java，标注valueDecode为java。

注意，为了实现Object类型的值进出redis，需要保障进出redis的Object类型的数据必须实现序列化接口。

```
@Data
public class Book implements Serializable {
    private Integer id;
    private String type;
    private String name;
    private String description;
}
```

步骤③：启用缓存时开启方法缓存功能，并配置basePackages，说明在哪些包中开启方法缓存

```
@SpringBootApplication
//jetcache启用缓存的主开关
@EnableCreateCacheAnnotation
//开启方法注解缓存
@EnableMethodCache(basePackages = "com.itheima")
public class Springboot20JetCacheApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot20JetCacheApplication.class, args);
    }
}
```

步骤④：使用注解@Cached标注当前方法使用缓存

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;

    @Override
    @Cached(name="book_",key="#id",expire = 3600,cacheType = CacheType.REMOTE)
    public Book getById(Integer id) {
        return bookDao.selectById(id);
    }
}
```

远程方案的数据同步

由于远程方案中redis保存的数据可以被多个客户端共享，这就存在了数据同步问题。jetcache提供了3个注解决此问题，分别在更新、删除操作时同步缓存数据，和读取缓存时定时刷新数据

更新缓存

```
@CacheUpdate(name="book_",key="#book.id",value="#book")
public boolean update(Book book) {
    return bookDao.updateById(book) > 0;
}
```

删除缓存

```
@CacheInvalidate(name="book_",key = "#id")
public boolean delete(Integer id) {
    return bookDao.deleteById(id) > 0;
}
```

定时刷新缓存

```
@Cached(name="book_",key="#id",expire = 3600,cacheType = CacheType.REMOTE)
@CacheRefresh(refresh = 5)
public Book getById(Integer id) {
    return bookDao.selectById(id);
}
```

数据报表

jetcache还提供有简单的数据报表功能，帮助开发者快速查看缓存命中信息，只需要添加一个配置即可

```
jetcache:  
    statIntervalMinutes: 1
```

设置后，每1分钟在控制台输出缓存数据命中信息

```
[DefaultExecutor] c.alicp.jetcache.support.StatInfoLogger : jetcache stat from  
2022-02-28 09:32:15,892 to 2022-02-28 09:33:00,003  
cache | qps | rate | get | hit | fail | expire | avgLoadTime |  
maxLoadTime  
-----+-----+-----+-----+-----+-----+-----+-----+  
book_ | 0.66 | 75.86% | 29 | 22 | 0 | 0 | 28.0 |  
188  
-----+-----+-----+-----+-----+-----+-----+  
-----
```

总结

1. jetcache是一个类似于springcache的缓存解决方案，自身不具有缓存功能，它提供有本地缓存与远程缓存多级共同使用的缓存解决方案
2. jetcache提供的缓存解决方案受限于目前支持的方案，本地缓存支持两种，远程缓存支持两种
3. 注意数据进入远程缓存时的类型转换问题
4. jetcache提供方法缓存，并提供了对应的缓存更新与刷新功能
5. jetcache提供有简单的缓存信息命中报表方便开发者即时监控缓存数据命中情况

思考

jetcache解决了前期使用缓存方案单一的问题，但是仍然不能灵活的选择缓存进行搭配使用，是否存在一种技术可以灵活的搭配各种各样的缓存使用呢？有，咱们下一节再讲。

SpringBoot整合j2cache缓存

jetcache可以在限定范围内构建多级缓存，但是灵活性不足，不能随意搭配缓存，本节介绍一种可以随意搭配缓存解决方案的缓存整合框架，j2cache。下面就来讲解如何使用这种缓存框架，以Ehcache与redis整合为例：

步骤①：导入j2cache、redis、ehcache坐标

```
<dependency>  
    <groupId>net.oschina.j2cache</groupId>  
    <artifactId>j2cache-core</artifactId>  
    <version>2.8.4-release</version>  
</dependency>  
<dependency>  
    <groupId>net.oschina.j2cache</groupId>  
    <artifactId>j2cache-spring-boot2-starter</artifactId>  
    <version>2.8.0-release</version>  
</dependency>  
<dependency>  
    <groupId>net.sf.ehcache</groupId>
```

```
<artifactId>ehcache</artifactId>
</dependency>
```

j2cache的starter中默认包含了redis坐标，官方推荐使用redis作为二级缓存，因此此处无需导入redis坐标

步骤②：配置一级与二级缓存，并配置一二级缓存间数据传递方式，配置书写在名称为j2cache.properties的文件中。如果使用ehcache还需要单独添加ehcache的配置文件

```
# 1级缓存
j2cache.L1.provider_class = ehcache
ehcache.configxml = ehcache.xml

# 2级缓存
j2cache.L2.provider_class =
net.oschina.j2cache.cache.support.redis.SpringRedisProvider
j2cache.L2.config_section = redis
redis.hosts = localhost:6379

# 1级缓存中的数据如何到达二级缓存
j2cache.broadcast =
net.oschina.j2cache.cache.support.redis.SpringRedisPubSubPolicy
```

此处配置不能乱配置，需要参照官方给出的配置说明进行。例如1级供应商选择ehcache，供应商名称仅仅是一个ehcache，但是2级供应商选择redis时要写专用的Spring整合Redis的供应商类名SpringRedisProvider，而且这个名称并不是所有的redis包中能提供的，也不是spring包中提供的。因此配置j2cache必须参照官方文档配置，而且还要去找专用的整合包，导入对应坐标才可以使用。

一级与二级缓存最重要的一个配置就是两者之间的数据沟通方式，此类配置也不是随意配置的，并且不同的缓存解决方案提供的数据沟通方式差异化很大，需要查询官方文档进行设置。

步骤③：使用缓存

```
@Service
public class SMSCodeServiceImpl implements SMSCodeService {
    @Autowired
    private Codeutils codeutils;

    @Autowired
    private Cachechannel cachechannel;

    public String sendCodeToSMS(String tele) {
        String code = codeutils.generator(tele);
        cacheChannel.set("sms", tele, code);
        return code;
    }

    public boolean checkCode(SMSCode smsCode) {
        String code = cacheChannel.get("sms", smsCode.getTele()).asString();
        return smsCode.getCode().equals(code);
    }
}
```

j2cache的使用和jetcache比较类似，但是无需开启使用的开关，直接定义缓存对象即可使用，缓存对象名CacheChannel。

j2cache的使用不复杂，配置是j2cache的核心，毕竟是一个整合型的缓存框架。缓存相关的配置过多，可以查阅j2cache-core核心包中的j2cache.properties文件中的说明。如下：

```
#J2Cache configuration
#####
# Cache Broadcast Method
# values:
# jgroups -> use jgroups's multicast
# redis -> use redis publish/subscribe mechanism (using jedis)
# lettuce -> use redis publish/subscribe mechanism (using lettuce, Recommend)
# rabbitmq -> use RabbitMQ publisher/consumer mechanism
# rocketmq -> use RocketMQ publisher/consumer mechanism
# none -> don't notify the other nodes in cluster
# xx.xxxx.xxxx.Xxxxx your own cache broadcast policy classname that implement
net.oschina.j2cache.cluster.ClusterPolicy
#####
j2cache.broadcast = redis

# jgroups properties
jgroups.channel.name = j2cache
jgroups.configXml = /network.xml

# RabbitMQ properties
rabbitmq.exchange = j2cache
rabbitmq.host = localhost
rabbitmq.port = 5672
rabbitmq.username = guest
rabbitmq.password = guest

# RocketMQ properties
rocketmq.name = j2cache
rocketmq.topic = j2cache
# use ; to split multi hosts
rocketmq.hosts = 127.0.0.1:9876

#####
# Level 1&2 provider
# values:
# none -> disable this level cache
# ehcache -> use ehcache2 as level 1 cache
# ehcache3 -> use ehcache3 as level 1 cache
# caffeine -> use caffeine as level 1 cache(only in memory)
# redis -> use redis as level 2 cache (using jedis)
# lettuce -> use redis as level 2 cache (using lettuce)
# readonly-redis -> use redis as level 2 cache ,but never write data to it. if
use this provider, you must uncomment `j2cache.L2.config_section` to make the
redis configurations available.
# memcached -> use memcached as level 2 cache (xmempached),
# [classname] -> use custom provider
#####

j2cache.L1.provider_class = caffeine
```

```
j2cache.L2.provider_class = redis

# When L2 provider isn't `redis`, using `L2.config_section = redis` to read redis
configurations
# j2cache.L2.config_section = redis

# Enable/Disable ttl in redis cache data (if disabled, the object in redis will
never expire, default:true)
# NOTICE: redis hash mode (redis.storage = hash) do not support this feature)
j2cache.sync_ttl_to_redis = true

# whether to cache null objects by default (default false)
j2cache.default_cache_null_object = true

#####
# Cache serialization Provider
# values:
# fst -> using fast-serialization (recommend)
# kryo -> using kryo serialization
# json -> using fst's json serialization (testing)
# fastjson -> using fastjson serialization (embed non-static class not support)
# java -> java standard
# fse -> using fse serialization
# [classname implements Serializer]
#####

j2cache.serialization = json
#json.map.person = net.oschina.j2cache.demo.Person

#####
# Ehcache configuration
#####

# ehcache.configxml = /ehcache.xml

# ehcache3.configxml = /ehcache3.xml
# ehcache3.defaultHeapSize = 1000

#####
# Caffeine configuration
# caffeine.region.[name] = size, xxxx[s|m|h|d]
#
#####
caffeine.properties = /caffeine.properties

#####
# Redis connection configuration
#####

#####
# Redis Cluster Mode
#
# single -> single redis server
# sentinel -> master-slaves servers
# cluster -> cluster servers (数据库配置无效, 使用 database = 0)
```

```
# sharded -> sharded servers (密码、数据库必须在 hosts 中指定，且连接池配置无效；  
redis://user:password@127.0.0.1:6379/0)  
#  
#####  
  
redis.mode = single  
  
#redis storage mode (generic|hash)  
redis.storage = generic  
  
## redis pub/sub channel name  
redis.channel = j2cache  
## redis pub/sub server (using redis.hosts when empty)  
redis.channel.host =  
  
#cluster name just for sharded  
redis.cluster_name = j2cache  
  
## redis cache namespace optional, default[empty]  
redis.namespace =  
  
## redis command scan parameter count, default[1000]  
#redis.scanCount = 1000  
  
## connection  
# Separate multiple redis nodes with commas, such as  
192.168.0.10:6379,192.168.0.11:6379,192.168.0.12:6379  
  
redis.hosts = 127.0.0.1:6379  
redis.timeout = 2000  
redis.password =  
redis.database = 0  
redis.ssl = false  
  
## redis pool properties  
redis.maxTotal = 100  
redis.maxIdle = 10  
redis.maxWaitMillis = 5000  
redis.minEvictableIdleTimeMillis = 60000  
redis.minIdle = 1  
redis.numTestsPerEvictionRun = 10  
redis.lifo = false  
redis.softMinEvictableIdleTimeMillis = 10  
redis.testOnBorrow = true  
redis.testOnReturn = false  
redis.testWhileIdle = true  
redis.timeBetweenEvictionRunsMillis = 300000  
redis.blockWhenExhausted = false  
redis.jmxEnabled = false  
  
#####  
# Lettuce scheme  
#  
# redis -> single redis server  
# rediss -> single redis server with ssl
```

```

# redis-sentinel -> redis sentinel
# redis-cluster -> cluster servers
#
#####
##### Lettuce Mode #####
#
# single -> single redis server
# sentinel -> master-slaves servers
# cluster -> cluster servers (数据库配置无效, 使用 database = 0)
# sharded -> sharded servers (密码、数据库必须在 hosts 中指定, 且连接池配置无效 ;
redis://user:password@127.0.0.1:6379/0)
#
#####

## redis command scan parameter count, default[1000]
#lettuce.scanCount = 1000
lettuce.mode = single
lettuce.namespace =
lettuce.storage = hash
lettuce.channel = j2cache
lettuce.scheme = redis
lettuce.hosts = 127.0.0.1:6379
lettuce.password =
lettuce.database = 0
lettuce.sentinelMasterId =
lettuce.maxTotal = 100
lettuce.maxIdle = 10
lettuce.minIdle = 10
# timeout in milliseconds
lettuce.timeout = 10000
# redis cluster topology refresh interval in milliseconds
lettuce.clusterTopologyRefresh = 3000

#####
# memcached server configurations
# refer to https://gitee.com/mirrors/XMemcached
#####

memcached.servers = 127.0.0.1:11211
memcached.username =
memcached.password =
memcached.connectionPoolSize = 10
memcached.connectTimeout = 1000
memcached.failureMode = false
memcached.healSessionInterval = 1000
memcached.maxQueuedNoReplyOperations = 100
memcached.opTimeout = 100
memcached.sanitizeKeys = false

```

总结

1. j2cache是一个缓存框架，自身不具有缓存功能，它提供多种缓存整合在一起使用的方案
2. j2cache需要通过复杂的配置设置各级缓存，以及缓存之间数据交换的方式

KF-5-2.任务

springboot整合第三方技术第二部分我们来说说任务系统，其实这里说的任务系统指的是定时任务。定时任务是企业级开发中必不可少的组成部分，诸如长周期业务数据的计算，例如年度报表，诸如系统脏数据的处理，再比如系统性能监控报告，还有抢购类活动的商品上架，这些都离不开定时任务。本节将介绍两种不同的定时任务技术。

Quartz

Quartz技术是一个比较成熟的定时任务框架，怎么说呢？有点繁琐，用过的都知道，配置略微复杂。springboot对其进行整合后，简化了一系列的配置，将很多配置采用默认设置，这样开发阶段就简化了很多。再学习springboot整合Quartz前先普及几个Quartz的概念。

- 工作 (Job)：用于定义具体执行的工作
- 工作明细 (JobDetail)：用于描述定时工作相关的信息
- 触发器 (Trigger)：描述了工作明细与调度器的对应关系
- 调度器 (Scheduler)：用于描述触发工作的执行规则，通常使用cron表达式定义规则

简单说就是你定时干什么事情，这就是工作，工作不可能就是一个简单的方法，还要设置一些明细信息。工作啥时候执行，设置一个调度器，可以简单理解成设置一个工作执行的时间。工作和调度都是独立定义的，它们两个怎么配合到一起呢？用触发器。完了，就这么多。下面开始springboot整合Quartz。

步骤①：导入springboot整合Quartz的starter

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

步骤②：定义任务Bean，按照Quartz的开发规范制作，继承QuartzJobBean

```
public class MyQuartz extends QuartzJobBean {
    @Override
    protected void executeInternal(JobExecutionContext context) throws
JobExecutionException {
        System.out.println("quartz task run...");
    }
}
```

步骤③：创建Quartz配置类，定义工作明细 (JobDetail) 与触发器的 (Trigger) bean

```
@Configuration
public class QuartzConfig {
    @Bean
    public JobDetail printJobDetail(){
        //绑定具体的工作
        return JobBuilder.newJob(MyQuartz.class).storeDurably().build();
    }
}
```

```

@Bean
public Trigger printJobTrigger(){
    ScheduleBuilder schedBuilder = CronScheduleBuilder.cronSchedule("0/5 * * * * ?");
    //绑定对应的工作明细
    return
TriggerBuilder.newTrigger().forJob(printJobDetail()).withSchedule(schedBuilder).
build();
}
}

```

工作明细中要设置对应的具体工作，使用newJob()操作传入对应的工作任务类型即可。

触发器需要绑定任务，使用forJob()操作传入绑定的工作明细对象。此处可以为工作明细设置名称然后使用名称绑定，也可以直接调用对应方法绑定。触发器中最核心的规则是执行时间，此处使用调度器定义执行时间，执行时间描述方式使用的是cron表达式。有关cron表达式的规则，各位小伙伴可以去参考相关课程学习，略微复杂，而且格式不能乱设置，不是写个格式就能用的，写不好就会出现冲突问题。

总结

1. springboot整合Quartz就是将Quartz对应的核心对象交给spring容器管理，包含两个对象，JobDetail和Trigger对象
2. JobDetail对象描述的是工作的执行信息，需要绑定一个QuartzJobBean类型的对象
3. Trigger对象定义了一个触发器，需要为其指定绑定的JobDetail是哪个，同时要设置执行周期调度器

思考

上面的操作看上去不多，但是Quartz将其中的对象划分粒度过细，导致开发的时候有点繁琐，spring针对上述规则进行了简化，开发了自己的任务管理组件——Task，如何用呢？咱们下节再说。

Spring Task

spring根据定时任务的特征，将定时任务的开发简化到了极致。怎么说呢？要做定时任务总要告诉容器有这功能吧，然后定时执行什么任务直接告诉对应的bean什么时间执行就行了，就这么简单，一起来看怎么做

步骤①：开启定时任务功能，在引导类上开启定时任务功能的开关，使用注解@EnableScheduling

```

@SpringBootApplication
//开启定时任务功能
@EnableScheduling
public class Springboot22TaskApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot22TaskApplication.class, args);
    }
}

```

步骤②：定义Bean，在对应要定时执行的操作上方，使用注解@Scheduled定义执行的时间，执行时间的描述方式还是cron表达式

```

@Component
public class MyBean {
    @Scheduled(cron = "0/1 * * * * ?")
    public void print(){
        System.out.println(Thread.currentThread().getName()+" :spring task
run...");
    }
}

```

完事，这就完成了定时任务的配置。总体感觉其实什么东西都没少，只不过没有将所有的信息都抽取成bean，而是直接使用注解绑定定时执行任务的事情而已。

如何想对定时任务进行相关配置，可以通过配置文件进行

```

spring:
  task:
    scheduling:
      pool:
        size: 1          # 任务调度线程池大小 默认 1
        thread-name-prefix: ssm_      # 调度线程名称前缀 默认 scheduling-
      shutdown:
        await-termination: false    # 线程池关闭时等待所有任务完成
        await-termination-period: 10s # 调度线程关闭前最大等待时间，确保最后一定关闭

```

总结

1. spring task需要使用注解@EnableScheduling开启定时任务功能
2. 为定时执行的任务设置执行周期，描述方式cron表达式

KF-5-3.邮件

springboot整合第三方技术第三部分我们来说说邮件系统，发邮件是java程序的基本操作，springboot整合javamail其实就是简化开发。不熟悉邮件的小伙伴可以先学习完javamail的基础操作，再来看这一部分内容才能感触到springboot整合javamail究竟简化了哪些操作。简化的多码？其实不多，差别不大，只是还个格式而已。

学习邮件发送之前先了解3个概念，这些概念规范了邮件操作过程中的标准。

- SMTP (Simple Mail Transfer Protocol)：简单邮件传输协议，用于**发送**电子邮件的传输协议
- POP3 (Post Office Protocol - Version 3)：用于**接收**电子邮件的标准协议
- IMAP (Internet Mail Access Protocol)：互联网消息协议，是POP3的替代协议

简单说就是SMTP是发邮件的标准，POP3是收邮件的标准，IMAP是对POP3的升级。我们制作程序中操作邮件，通常是发邮件，所以SMTP是使用的重点，收邮件大部分都是通过邮件客户端完成，所以开发收邮件的代码极少。除非你要读取邮件内容，然后解析，做邮件功能的统一处理。例如HR的邮箱收到求职者的简历，可以读取后统一处理。但是为什么不制作独立的投递简历的系统呢？所以说，好奇怪的需求，因为要想收邮件就要规范发邮件的人的书写格式，这个未免有点强人所难，并且极易收到外部攻击，你不可能使用白名单来收邮件。如果能使用白名单来收邮件然后解析邮件，还不如开发个系统给白名单中的人专用呢，更安全，总之就是鸡肋了。下面就开始学习springboot如何整合javamail发送邮件。

发送简单邮件

步骤①：导入springboot整合javamail的starter

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

步骤②：配置邮箱的登录信息

```
spring:
  mail:
    #126是邮箱
    host: smtp.126.com
    username: test@126.com
    password: test
```

java程序仅用于发送邮件，邮件的功能还是邮件供应商提供的，所以这里是用别人的邮件服务，要配置对应信息。

host配置的是提供邮件服务的主机协议，当前程序仅用于发送邮件，因此配置的是smtp的协议。

下面教你如何获得password

password并不是邮箱账号的登录密码，是邮件供应商提供的一个加密后的密码，也是为了保障系统安全性。不然外部人员通过地址访问下载了配置文件，直接获取到了邮件密码就会有极大的安全隐患。有关该密码的获取每个邮件供应商提供的方式都不一样，此处略过。可以到[邮件供应商](#)的设置页面找POP3或IMAP这些关键词找到对应的获取位置。下例仅供参考：

下面是qq邮箱的



步骤③：使用JavaMailSender接口发送邮件

```
@Service
public class SendMailServiceImpl implements SendMailService {
    @Autowired
    private JavaMailSender javaMailSender;

    //发送人
    private String from = "test@qq.com";
    //接收人
    private String to = "test@126.com";
    //标题
    private String subject = "测试邮件";
```

```

//正文
private String context = "测试邮件正文内容";

@Override
public void sendMail() {
    SimpleMailMessage message = new SimpleMailMessage();
    message.setFrom(from+"(小甜甜)");
    message.setTo(to);
    message.setSubject(subject);
    message.setText(context);
    javaMailSender.send(message);
}
}

```

将发送邮件的必要信息（发件人、收件人、标题、正文）封装到SimpleMailMessage对象中，可以根据规则设置发送人昵称等。

发送多组件邮件（附件、复杂正文）

发送简单邮件仅需要提供对应的4个基本信息就可以了，如果想发送复杂的邮件，需要更换邮件对象。使用MimeMessage可以发送特殊的邮件。

发送网页正文邮件

```

@Service
public class SendMailServiceImp2 implements SendMailService {
    @Autowired
    private JavaMailSender javaMailSender;

    //发送人
    private String from = "test@qq.com";
    //接收人
    private String to = "test@126.com";
    //标题
    private String subject = "测试邮件";
    //正文
    private String context = "<img src='ABC.JPG' /><a href='https://www.itcast.cn'>点开有惊喜</a>";

    public void sendMail() {
        try {
            MimeMessage message = javaMailSender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(message);
            helper.setFrom(to+"(小甜甜)");
            helper.setTo(from);
            helper.setSubject(subject);
            helper.setText(context, true);           //此处设置正文支持html解析

            javaMailSender.send(message);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

发送带有附件的邮件

```
@Service
public class SendMailServiceImpl implements SendMailService {
    @Autowired
    private JavaMailSender javaMailSender;

    //发送人
    private String from = "test@qq.com";
    //接收人
    private String to = "test@126.com";
    //标题
    private String subject = "测试邮件";
    //正文
    private String context = "测试邮件正文";

    public void sendMail() {
        try {
            MimeMessage message = javaMailSender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(message, true);
            //此处设置支持附件
            helper.setFrom(to + "(小甜甜)");
            helper.setTo(from);
            helper.setSubject(subject);
            helper.setText(context);

            //添加附件
            File f1 = new File("springboot_23_mail-0.0.1-SNAPSHOT.jar");
            File f2 = new File("resources\\logo.png");

            helper.addAttachment(f1.getName(), f1);
            helper.addAttachment("最靠谱的培训结构.png", f2);

            javaMailSender.send(message);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

总结

1. springboot整合javamail其实就简化了发送邮件的客户端对象JavaMailSender的初始化过程，通过配置的形式加载信息简化开发过程

KF-5-4.消息

springboot整合第三方技术最后一部分我们来说说消息中间件，首先先介绍一下消息的应用。

消息的概念

从广义角度来说，消息其实就是信息，但是和信息又有所不同。信息通常被定义为一组数据，而消息除了具有数据的特征之外，还有消息的来源与接收的概念。通常发送消息的一方称为消息的生产者，接收消息的一方称为消息的消费者。这样比较后，发现其实消息和信息差别还是很大的。

为什么要设置生产者和消费者呢？这就是要说到消息的意义了。信息通常就是一组数据，但是消息由于有了生产者和消费者，就出现了消息中所包含的信息可以被二次解读，生产者发送消息，可以理解为生产者发送了一个信息，也可以理解为生产者发送了一个命令；消费者接收消息，可以理解为消费者得到了一个信息，也可以理解为消费者得到了一个命令。对比一下我们会发现信息是一个基本数据，而命令则可以关联下一个行为动作，这样就可以理解为基于接收的消息相当于得到了一个行为动作，使用这些行为动作就可以组织成一个业务逻辑，进行进一步的操作。总的来说，消息其实也是一组信息，只是为其赋予了全新的含义，因为有了消息的流动，并且是有方向性的流动，带来了基于流动的行为产生的全新解读。开发者就可以基于消息的这种特殊解，将其换成代码中的指令。

对于消息的理解，初学者总认为消息内部的数据非常复杂，这是一个误区。比如我发送了一个消息，要求接受者翻译发送过去的内容。初学者会认为消息中会包含被翻译的文字，已经本次操作要执行翻译操作而不是打印操作。其实这种现象有点过度解读了，发送的消息中仅仅包含被翻译的文字，但是可以通过控制不同的人接收此消息来确认要做的事情。例如发送被翻译的文字仅到A程序，而A程序只能进行翻译操作，这样就可以发送简单的信息完成复杂的业务了，是通过接收消息的主体不同，进而执行不同的操作，而不会在消息内部定义数据的操作行为，当然如果开发者希望消息中包含操作种类信息也是可以的，只是提出消息的内容可以更简单，更单一。

对于消息的生产者与消费者的工作模式，还可以将消息划分成两种模式，同步消费与异步消息。

所谓同步消息就是生产者发送完消息，等待消费者处理，消费者处理完将结果告知生产者，然后生产者继续向下执行业务。这种模式过于卡生产者的业务执行连续性，在现在的企业级开发中，上述这种业务场景通常不会采用消息的形式进行处理。

所谓异步消息就是生产者发送完消息，无需等待消费者处理完毕，生产者继续向下执行其他动作。比如生产者发送了一个日志信息给日志系统，发送过去以后生产者就向下做其他事情了，无需关注日志系统的执行结果。日志系统根据接收到的日志信息继续进行业务执行，是单纯的记录日志，还是记录日志并报警，这些和生产者无关，这样生产的业务执行效率就会大幅度提升。并且可以通过添加多个消费者来处理同一个生产者发送的消息来提高系统的高并发性，改善系统工作效率，提高用户体验。一旦某一个消费者由于各种问题宕机了，也不会对业务产生影响，提高了系统的高可用性。

以上简单的介绍了一下消息这种工作模式存在的意义，希望对各位学习者有所帮助。

Java处理消息的标准规范

目前企业级开发中广泛使用的消息处理技术共三大类，具体如下：

- JMS
- AMQP
- MQTT

为什么是三大类，而不是三个技术呢？因为这些都是**规范**，就想 JDBC 技术，是个规范，开发针对规范开发，运行还要靠实现类，例如 MySQL 提供了 JDBC 的实现，最终运行靠的还是实现。并且这三类规范都是针对异步消息进行处理的，也符合消息的设计本质，处理异步的业务。对以上三种消息规范做一下普及

JMS

JMS (Java Message Service) ,这是一个规范，作用等同于JDBC规范，提供了与消息服务相关的API接口。

JMS消息模型

JMS规范中规范了消息有两种模型。分别是**点对点模型**和**发布订阅模型**。

点对点模型：peer-2-peer，生产者会将消息发送到一个保存消息的容器中，通常使用队列模型，使用队列保存消息。一个队列的消息只能被一个消费者消费，或未被及时消费导致超时。这种模型下，生产者和消费者是一对一绑定的。

发布订阅模型：publish-subscribe，生产者将消息发送到一个保存消息的容器中，也是使用队列模型来保存。但是消息可以被多个消费者消费，生产者和消费者完全独立，相互不需要感知对方的存在。

以上这种分类是从消息的生产和消费过程来进行区分，针对消息所包含的信息不同，还可以进行不同类别的划分。

JMS消息种类

根据消息中包含的数据种类划分，可以将消息划分成6种消息。

- TextMessage
- MapMessage
- BytesMessage
- StreamMessage
- ObjectMessage
- Message (只有消息头和属性)

JMS主张不同种类的消息，消费方式不同，可以根据使用需要选择不同种类的消息。但是这一点也成为其诟病之处，后面再说。整体上来说，JMS就是典型的保守派，什么都按照J2EE的规范来，做一套规范，定义若干个标准，每个标准下又提供一大批API。目前对JMS规范实现的消息中间件技术还是挺多的，毕竟是皇家御用，肯定有人舔，例如ActiveMQ、Redis、HornetMQ。但是也有一些不太规范的实现，参考JMS的标准设计，但是又不完全满足其规范，例如：RabbitMQ、RocketMQ。

AMQP

JMS的问世为消息中间件提供了很强大的规范性支撑，但是使用的过程中就开始被人诟病，比如JMS设置的极其复杂的多种类消息处理机制。本来分门别类处理挺好的，为什么会被诟病呢？原因就在于JMS的设计是J2EE规范，站在Java开发的角度思考问题。但是现实往往是复杂度很高的。比如我有一个.NET开发的系统A，有一个Java开发的系统B，现在要从A系统给B系统发业务消息，结果两边数据格式不统一，没法操作。JMS不是可以统一数据格式吗？提供了6种数据种类，总有一款适合你啊。NO，一个都不能用。因为A系统的底层语言不是Java语言开发的，根本不支持那些对象。这就意味着如果想使用现有的业务系统A继续开发已经不可能了，必须推翻重新做使用Java语言开发的A系统。

这时候有人就提出说，你搞那么复杂，整那么多种类干什么？找一种大家都支持的消息数据类型不就解决这个跨平台的问题了吗？大家一想，对啊，于是AMQP孕育而生。

单从上面的说明中其实可以明确感知到，AMQP的出现解决的是消息传递时使用的消息种类的问题，化繁为简，但是其并没有完全推翻JMS的操作API，所以说AMQP仅仅是一种协议，规范了数据传输的格式而已。

AMQP (advanced message queuing protocol) : 一种协议 (高级消息队列协议，也是消息代理规范)，规范了网络交换的数据格式，兼容JMS操作。

优点

具有跨平台性，服务器供应商，生产者，消费者可以使用不同的语言来实现

JMS消息种类

AMQP消息种类：byte[]

AMQP在JMS的消息模型基础上又进行了进一步的扩展，除了点对点和发布订阅的模型，开发了几种全新的消息模型，适应各种各样的消息发送。

AMQP消息模型

- direct exchange
- fanout exchange
- topic exchange
- headers exchange
- system exchange

目前实现了AMQP协议的消息中间件技术也很多，而且都是较为流行的技术，例如：RabbitMQ、StormMQ、RocketMQ

MQTT

MQTT (Message Queueing Telemetry Transport) 消息队列遥测传输，专为小设备设计，是物联网 (IOT) 生态系统中主要成分之一。由于与JavaEE企业级开发没有交集，此处不作过多的说明。

除了上述3种J2EE企业级应用中广泛使用的三种异步消息传递技术，还有一种技术也不能忽略，Kafka。

Kafka

Kafka，一种高吞吐量的分布式发布订阅消息系统，提供实时消息功能。Kafka技术并不是作为消息中间件为主要功能的产品，但是其拥有发布订阅的工作模式，也可以充当消息中间件来使用，而且目前企业级开发中其身影也不少见。

本节内容讲围绕着上述内容中的几种实现方案讲解springboot整合各种各样的消息中间件。由于各种消息中间件必须先安装再使用，下面的内容采用Windows系统安装，降低各位学习者的学习难度，基本套路和之前学习NoSQL解决方案一样，先安装再整合。

购物订单发送手机短信案例

为了便于下面演示各种各样的消息中间件技术，我们创建一个购物过程生成订单时为用户发送短信的案例环境，模拟使用消息中间件实现发送手机短信的过程。

手机验证码案例需求如下：

- 执行下单业务时（模拟此过程），调用消息服务，将要发送短信的订单id传递给消息中间件
- 消息处理服务接收到要发送的订单id后输出订单id（模拟发短信）

由于不涉及数据读写，仅开发业务层与表现层，其中短信处理的业务代码独立开发，代码如下：

订单业务

业务层接口

```
public interface OrderService {  
    void order(String id);  
}
```

模拟传入订单id，执行下订单业务，参数为虚拟设定，实际应为订单对应的实体类

业务层实现

```

@Service
public class OrderServiceImpl implements OrderService {
    @Autowired
    private MessageService messageService;

    @Override
    public void order(String id) {
        //一系列操作，包含各种服务调用，处理各种业务
        System.out.println("订单处理开始");
        //短信消息处理
        messageService.sendMessage(id);
        System.out.println("订单处理结束");
        System.out.println();
    }
}

```

业务层转调短信处理的服务MessageService

表现层服务

```

@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    private OrderService orderService;

    @PostMapping("{id}")
    public void order(@PathVariable String id){
        orderService.order(id);
    }
}

```

表现层对外开发接口，传入订单id即可（模拟）

短信处理业务

业务层接口

```

public interface MessageService {
    void sendMessage(String id);
    String doMessage();
}

```

短信处理业务层接口提供两个操作，发送要处理的订单id到消息中间件，另一个操作目前暂且设计成处理消息，实际消息的处理过程不应该是手动执行，应该是自动执行，到具体实现时再进行设计

业务层实现

```

@Service
public class MessageServiceImpl implements MessageService {
    private ArrayList<String> msgList = new ArrayList<String>();

    @Override

```

```

public void sendMessage(String id) {
    System.out.println("待发送短信的订单已纳入处理队列, id: "+id);
    msgList.add(id);
}

@Override
public String doMessage() {
    String id = msgList.remove(0);
    System.out.println("已完成短信发送业务, id: "+id);
    return id;
}
}

```

短信处理业务层实现中使用集合先模拟消息队列，观察效果

表现层服务

```

@RestController
@RequestMapping("/msgs")
public class MessageController {

    @Autowired
    private MessageService messageService;

    @GetMapping
    public String doMessage(){
        String id = messageService.doMessage();
        return id;
    }
}

```

短信处理表现层接口暂且开发出一个处理消息的入口，但是此业务是对应业务层中设计的模拟接口，实际业务不需要设计此接口。

下面开启springboot整合各种各样的消息中间件，从严格满足JMS规范的ActiveMQ开始

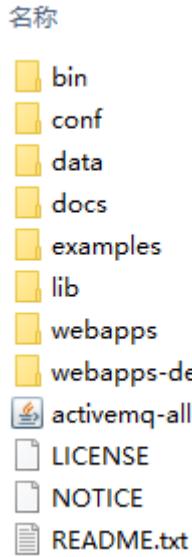
SpringBoot整合ActiveMQ

ActiveMQ是MQ产品中的元老级产品，早期标准MQ产品之一，在AMQP协议没有出现之前，占据了消息中间件市场的绝大部分份额，后期因为AMQP系列产品的出现，迅速走弱，目前仅在一些线上运行的产品中出现，新产品开发较少采用。

安装

windows版安装包下载地址：<https://activemq.apache.org/components/classic/download/>

下载的安装包是解压缩就能使用的zip文件，解压缩完毕后会得到如下文件



启动服务器

activemq.bat

运行bin目录下的win32或win64目录下的activemq.bat命令即可，根据自己的操作系统选择即可，
默认对外服务端口61616。

访问web管理服务

ActiveMQ启动后会启动一个Web控制台服务，可以通过该服务管理ActiveMQ。

http://127.0.0.1:8161/

web管理服务默认端口8161，访问后可以打开ActiveMQ的管理界面，如下：



首先输入访问用户名和密码，初始化用户名和密码相同，均为：admin，成功登录后进入管理后台
界面，如下：

The screenshot shows the Apache ActiveMQ management console interface. At the top, there's a header with the ActiveMQ logo and the Apache Software Foundation logo. The main content area has a "Welcome!" message and a "Broker" table showing system information like Name (localhost), Version (5.16.3), ID (ID:CZBK-20210302VL-10434-1646035669595-0:1), Uptime (1 minute), and memory usage (Store percent used 0, Memory percent used 0, Temp percent used 0). On the right side, there's a sidebar with links for Queue Views, Topic Views, Subscribers Views, and Useful Links. The footer includes copyright information and a link to the Apache Software Foundation.

看到上述界面视为启动ActiveMQ服务成功。

启动失败

在ActiveMQ启动时要占用多个端口，以下为正常启动信息：

```
wrapper | --> Wrapper Started as Console
wrapper | Launching a JVM...
jvm 1 | Wrapper (Version 3.2.3) http://wrapper.tanukisoftware.org
jvm 1 | Copyright 1999-2006 Tanuki Software, Inc. All Rights Reserved.
jvm 1 |
jvm 1 | Java Runtime: oracle corporation 1.8.0_172 D:\soft\jdk1.8.0_172\jre
jvm 1 | Heap sizes: current=249344k free=235037k max=932352k
jvm 1 | JVM args: -Dactivemq.home=.../... -Dactivemq.base=.../... -
Djavax.net.ssl.keyStorePassword=password -
Djavax.net.ssl.trustStorePassword=password -
Djavax.net.ssl.keyStore=.../..../conf/broker.ks -
Djavax.net.ssl.trustStore=.../..../conf/broker.ts -Dcom.sun.management.jmxremote -
Dorg.apache.activemq.UseDedicatedTaskRunner=true -
Djava.util.logging.config.file=logging.properties -Dactivemq.conf=.../..../conf -
Dactivemq.data=.../..../data -
Djava.security.auth.login.config=.../..../conf/login.config -Xmx1024m -
Djava.library.path=.../..../bin/win64 -Dwrapper.key=7ySrCD75XhLCpLjd -
Dwrapper.port=32000 -Dwrapper.jvm.port.min=31000 -Dwrapper.jvm.port.max=31999 -
Dwrapper.pid=9364 -Dwrapper.version=3.2.3 -Dwrapper.native_library=wrapper -
Dwrapper.cpu.timeout=10 -Dwrapper.jvmid=1
jvm 1 | Extensions classpath:
jvm 1 |
[...\\..\\lib,...\\..\\lib\\camel,...\\..\\lib\\optional,...\\..\\lib\\web,...\\..\\lib\\extra]
jvm 1 | ACTIVEMQ_HOME: ...\\..
jvm 1 | ACTIVEMQ_BASE: ...\\..
jvm 1 | ACTIVEMQ_CONF: ...\\..\\conf
jvm 1 | ACTIVEMQ_DATA: ...\\..\\data
jvm 1 | Loading message broker from: xbean:activemq.xml
jvm 1 | INFO | Refreshing
org.apache.activemq.xbean.XBeanBrokerFactory$1@5f3ebfe0: startup date [Mon Feb
28 16:07:48 CST 2022]; root of context hierarchy
jvm 1 | INFO | Using Persistence Adapter:
KahaDBPersistenceAdapter[D:\\soft\\activemq\\bin\\win64\\..\\..\\data\\kahadb]
jvm 1 | INFO | KahaDB is version 7
jvm 1 | INFO | PLListStore:
[D:\\soft\\activemq\\bin\\win64\\..\\..\\data\\localhost\\tmp_storage] started
jvm 1 | INFO | Apache ActiveMQ 5.16.3 (localhost, ID:CZBK-20210302VL-10434-
1646035669595-0:1) is starting
jvm 1 | INFO | Listening for connections at: tcp://CZBK-20210302VL:61616?
maximumConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector openwire started
jvm 1 | INFO | Listening for connections at: amqp://CZBK-20210302VL:5672?
maximumConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector amqp started
jvm 1 | INFO | Listening for connections at: stomp://CZBK-20210302VL:61613?
maximumConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector stomp started
jvm 1 | INFO | Listening for connections at: mqtt://CZBK-20210302VL:1883?
maximumConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector mqtt started
jvm 1 | INFO | Starting Jetty server
```

```

jvm 1 | INFO | Creating Jetty connector
jvm 1 | WARN |
ServletContext@o.e.j.s.ServletContextHandler@7350746f{/null,STARTING} has
uncovered http methods for path: /
jvm 1 | INFO | Listening for connections at ws://CZBK-20210302VL:61614?
maximumConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector ws started
jvm 1 | INFO | Apache ActiveMQ 5.16.3 (localhost, ID:CZBK-20210302VL-10434-
1646035669595-0:1) started
jvm 1 | INFO | For help or more information please see:
http://activemq.apache.org
jvm 1 | WARN | Store limit is 102400 mb (current store usage is 0 mb). The
data directory: D:\soft\activemq\bin\win64\..\..\data\kahadb only has 68936 mb
of usable space. - resetting to maximum available disk space: 68936 mb
jvm 1 | INFO | ActiveMQ WebConsole available at http://127.0.0.1:8161/
jvm 1 | INFO | ActiveMQ Jolokia REST API available at
http://127.0.0.1:8161/api/jolokia/

```

其中占用的端口有：61616、5672、61613、1883、61614，如果启动失败，请先管理对应端口即可。以下就是某个端口占用的报错信息，可以从抛出异常的位置看出，启动5672端口时端口被占用，显示java.net.BindException: Address already in use: JVM_Bind。Windows系统中终止端口运行的操作参看[【命令行启动常见问题及解决方案】](#)

```

wrapper | --> Wrapper Started as Console
wrapper | Launching a JVM...
jvm 1 | Wrapper (Version 3.2.3) http://wrapper.tanukisoftware.org
jvm 1 | Copyright 1999-2006 Tanuki Software, Inc. All Rights Reserved.
jvm 1 |
jvm 1 | Java Runtime: Oracle Corporation 1.8.0_172 D:\soft\jdk1.8.0_172\jre
jvm 1 | Heap sizes: current=249344k free=235038k max=932352k
jvm 1 | JVM args: -Dactivemq.home=.../... -Dactivemq.base=.../... -
Djavax.net.ssl.keyStorePassword=password -
Djavax.net.ssl.trustStorePassword=password -
Djavax.net.ssl.keyStore=.../.../conf/broker.ks -
Djavax.net.ssl.trustStore=.../.../conf/broker.ts -Dcom.sun.management.jmxremote -
Dorg.apache.activemq.UseDedicatedTaskRunner=true -
Djava.util.logging.config.file=logging.properties -Dactivemq.conf=.../.../conf -
Dactivemq.data=.../.../data -
Djava.security.auth.login.config=.../.../conf/login.config -Xmx1024m -
Djava.library.path=.../.../bin/win64 -Dwrapper.key=QPJoy9ZoXewmmwTS -
Dwrapper.port=32000 -Dwrapper.jvm.port.min=31000 -Dwrapper.jvm.port.max=31999 -
Dwrapper.pid=14836 -Dwrapper.version=3.2.3 -Dwrapper.native_library=wrapper -
Dwrapper.cpu.timeout=10 -Dwrapper.jvmid=1
jvm 1 | Extensions classpath:
jvm 1 |
[...\\..\\lib,...\\..\\lib\\camel,...\\..\\lib\\optional,...\\..\\lib\\web,...\\..\\lib\\extra]
jvm 1 | ACTIVEMQ_HOME: ...\\..
jvm 1 | ACTIVEMQ_BASE: ...\\..
jvm 1 | ACTIVEMQ_CONF: ...\\..\\conf
jvm 1 | ACTIVEMQ_DATA: ...\\..\\data
jvm 1 | Loading message broker from: xbean:activemq.xml
jvm 1 | INFO | Refreshing
org.apache.activemq.xbean.XBeanBrokerFactory$1@2c9392f5: startup date [Mon Feb
28 16:06:16 CST 2022]; root of context hierarchy

```

```
jvm 1 |  INFO | Using Persistence Adapter:  
KahaDBPersistenceAdapter[D:\soft\activemq\bin\win64\..\..\data\kahadb]  
jvm 1 |  INFO | KahaDB is version 7  
jvm 1 |  INFO | PLListStore:  
[D:\soft\activemq\bin\win64\..\..\data\localhost\tmp_storage] started  
jvm 1 |  INFO | Apache ActiveMQ 5.16.3 (localhost, ID:CZBK-20210302VL-10257-  
1646035577620-0:1) is starting  
jvm 1 |  INFO | Listening for connections at: tcp://CZBK-20210302VL:61616?  
maximumConnections=1000&wireFormat.maxFrameSize=104857600  
jvm 1 |  INFO | Connector openwire started  
jvm 1 |  ERROR | Failed to start Apache ActiveMQ (localhost, ID:CZBK-  
20210302VL-10257-1646035577620-0:1)  
jvm 1 | java.io.IOException: Transport Connector could not be registered in  
JMX: java.io.IOException: Failed to bind to server socket: amqp://0.0.0.0:5672?  
maximumConnections=1000&wireFormat.maxFrameSize=104857600 due to:  
java.net.BindException: Address already in use: JVM_Bind  
jvm 1 |      at  
org.apache.activemq.util.IOExceptionSupport.create(IOExceptionSupport.java:28)  
jvm 1 |      at  
org.apache.activemq.broker.BrokerService.registerConnectorMBean(BrokerService.ja  
va:2288)  
jvm 1 |      at  
org.apache.activemq.broker.BrokerService.startTransportConnector(BrokerService.j  
ava:2769)  
jvm 1 |      at  
org.apache.activemq.broker.BrokerService.startAllConnectors(BrokerService.java:2  
665)  
jvm 1 |      at  
org.apache.activemq.broker.BrokerService.doStartBroker(BrokerService.java:780)  
jvm 1 |      at  
org.apache.activemq.broker.BrokerService.startBroker(BrokerService.java:742)  
jvm 1 |      at  
org.apache.activemq.broker.BrokerService.start(BrokerService.java:645)  
jvm 1 |      at  
org.apache.activemq.xbean.XBeanBrokerService.afterPropertiesSet(XBeanBrokerServ  
ice.java:73)  
jvm 1 |      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
jvm 1 |      at  
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  
jvm 1 |      at  
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav  
a:43)  
jvm 1 |      at java.lang.reflect.Method.invoke(Method.java:498)  
jvm 1 |      at  
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.inv  
okeCustomInitMethod(AbstractAutowireCapableBeanFactory.java:1748)  
jvm 1 |      at  
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.inv  
okeInitMethods(AbstractAutowireCapableBeanFactory.java:1685)  
jvm 1 |      at  
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.ini  
tializeBean(AbstractAutowireCapableBeanFactory.java:1615)  
jvm 1 |      at  
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doc  
reateBean(AbstractAutowireCapableBeanFactory.java:553)
```

```
jvm 1 |      at
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java:481)
jvm 1 |      at
org.springframework.beans.factory.support.AbstractBeanFactory$1.getObject(AbstractBeanFactory.java:312)
jvm 1 |      at
org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:230)
jvm 1 |      at
org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:308)
jvm 1 |      at
org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:197)
jvm 1 |      at
org.springframework.beans.factory.support.DefaultListableBeanFactory.preInstantiatesSingletons(DefaultListableBeanFactory.java:756)
jvm 1 |      at
org.springframework.context.support.AbstractApplicationContext.finishBeanFactoryInitialization(AbstractApplicationContext.java:867)
jvm 1 |      at
org.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicationContext.java:542)
jvm 1 |      at
org.apache.xbean.spring.context.ResourceXmlApplicationContext.<init>(ResourceXmlApplicationContext.java:64)
jvm 1 |      at
org.apache.xbean.spring.context.ResourceXmlApplicationContext.<init>(ResourceXmlApplicationContext.java:52)
jvm 1 |      at org.apache.activemq.xbean.XBeanBrokerFactory$1.<init>(XBeanBrokerFactory.java:104)
jvm 1 |      at
org.apache.activemq.xbean.XBeanBrokerFactory.createApplicationContext(XBeanBrokerFactory.java:104)
jvm 1 |      at
org.apache.activemq.xbean.XBeanBrokerFactory.createBroker(XBeanBrokerFactory.java:67)
jvm 1 |      at
org.apache.activemq.broker.BrokerFactory.createBroker(BrokerFactory.java:71)
jvm 1 |      at
org.apache.activemq.broker.BrokerFactory.createBroker(BrokerFactory.java:54)
jvm 1 |      at
org.apache.activemq.console.command.StartCommand.runTask(StartCommand.java:87)
jvm 1 |      at
org.apache.activemq.console.command.AbstractCommand.execute(AbstractCommand.java:63)
jvm 1 |      at
org.apache.activemq.console.command.ShellCommand.runTask(ShellCommand.java:154)
jvm 1 |      at
org.apache.activemq.console.command.AbstractCommand.execute(AbstractCommand.java:63)
jvm 1 |      at
org.apache.activemq.console.command.ShellCommand.main(ShellCommand.java:104)
jvm 1 |      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

```
jvm 1 |      at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
jvm 1 |      at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
jvm 1 |      at java.lang.reflect.Method.invoke(Method.java:498)
jvm 1 |      at org.apache.activemq.console.Main.runTaskClass(Main.java:262)
jvm 1 |      at org.apache.activemq.console.Main.main(Main.java:115)
jvm 1 |      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
jvm 1 |      at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
jvm 1 |      at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
jvm 1 |      at java.lang.reflect.Method.invoke(Method.java:498)
jvm 1 |      at
org.tanukisoftware.wrapper.WrapperSimpleApp.run(WrapperSimpleApp.java:240)
jvm 1 |      at java.lang.Thread.run(Thread.java:748)
jvm 1 | Caused by: java.io.IOException: Failed to bind to server socket:
amqp://0.0.0.0:5672?maximumConnections=1000&wireFormat.maxFrameSize=104857600
due to: java.net.BindException: Address already in use: JVM_Bind
jvm 1 |      at
org.apache.activemq.util.IOExceptionSupport.create(IOExceptionSupport.java:34)
jvm 1 |      at
org.apache.activemq.transport.tcp.TcpTransportServer.bind(TcpTransportServer.java:146)
jvm 1 |      at
org.apache.activemq.transport.tcp.TcpTransportFactory.doBind(TcpTransportFactory.java:62)
jvm 1 |      at
org.apache.activemq.transport.TransportFactorySupport.bind(TransportFactorySupport.java:40)
jvm 1 |      at
org.apache.activemq.broker.TransportConnector.createTransportServer(TransportConnector.java:335)
jvm 1 |      at
org.apache.activemq.broker.TransportConnector.getServer(TransportConnector.java:145)
jvm 1 |      at
org.apache.activemq.broker.TransportConnector.asManagedConnector(TransportConnector.java:110)
jvm 1 |      at
org.apache.activemq.broker.BrokerService.registerConnectorMBean(BrokerService.java:2283)
jvm 1 |      ... 46 more
jvm 1 | Caused by: java.net.BindException: Address already in use: JVM_Bind
jvm 1 |      at java.net.DualStackPlainSocketImpl.bind0(Native Method)
jvm 1 |      at
java.net.DualStackPlainSocketImpl.socketBind(DualStackPlainSocketImpl.java:106)
jvm 1 |      at
java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocketImpl.java:387)
jvm 1 |      at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:190)
jvm 1 |      at java.net.ServerSocket.bind(ServerSocket.java:375)
jvm 1 |      at java.net.ServerSocket.<init>(ServerSocket.java:237)
```

```
jvm 1 |      at
javax.net.DefaultServerSocketFactory.createServerSocket(ServerSocketFactory.java
:231)
jvm 1 |      at
org.apache.activemq.transport.tcp.TcpTransportServer.bind(TcpTransportServer.jav
a:143)
jvm 1 |      ... 52 more
jvm 1 |  INFO | Apache ActiveMQ 5.16.3 (localhost, ID:CBZK-20210302VL-10257-
1646035577620-0:1) is shutting down
jvm 1 |  INFO | socketQueue interrupted - stopping
jvm 1 |  INFO | Connector openwire stopped
jvm 1 |  INFO | Could not accept connection during shutdown : null (null)
jvm 1 |  INFO | Connector amqp stopped
jvm 1 |  INFO | Connector stomp stopped
jvm 1 |  INFO | Connector mqtt stopped
jvm 1 |  INFO | Connector ws stopped
jvm 1 |  INFO | PListStore:
[D:\soft\activemq\bin\win64\..\..\data\localhost\tmp_storage] stopped
jvm 1 |  INFO | Stopping async queue tasks
jvm 1 |  INFO | Stopping async topic tasks
jvm 1 |  INFO | Stopped Kahadb
jvm 1 |  INFO | Apache ActiveMQ 5.16.3 (localhost, ID:CBZK-20210302VL-10257-
1646035577620-0:1) uptime 0.426 seconds
jvm 1 |  INFO | Apache ActiveMQ 5.16.3 (localhost, ID:CBZK-20210302VL-10257-
1646035577620-0:1) is shutdown
jvm 1 |  INFO | Closing
org.apache.activemq.xbean.XBeanBrokerFactory$1@2c9392f5: startup date [Mon Feb
28 16:06:16 CST 2022]; root of context hierarchy
jvm 1 |  WARN | Exception encountered during context initialization -
canceling refresh attempt:
org.springframework.beans.factory.BeanCreationException: Error creating bean
with name 'org.apache.activemq.xbean.XBeanBrokerService#0' defined in class path
resource [activemq.xml]: Invocation of init method failed; nested exception is
java.io.IOException: Transport Connector could not be registered in JMX:
java.io.IOException: Failed to bind to server socket: amqp://0.0.0.0:5672?
maximumConnections=1000&wireFormat.maxFrameSize=104857600 due to:
java.net.BindException: Address already in use: JVM_Bind
jvm 1 |  ERROR: java.lang.RuntimeException: Failed to execute start task.
Reason: java.lang.IllegalStateException: BeanFactory not initialized or already
closed - call 'refresh' before accessing beans via the ApplicationContext
jvm 1 |  java.lang.RuntimeException: Failed to execute start task. Reason:
java.lang.IllegalStateException: BeanFactory not initialized or already closed -
call 'refresh' before accessing beans via the ApplicationContext
jvm 1 |      at
org.apache.activemq.console.command.StartCommand.runTask(StartCommand.java:91)
jvm 1 |      at
org.apache.activemq.console.command.AbstractCommand.execute(AbstractCommand.java
:63)
jvm 1 |      at
org.apache.activemq.console.command.ShellCommand.runTask(ShellCommand.java:154)
jvm 1 |      at
org.apache.activemq.console.command.AbstractCommand.execute(AbstractCommand.java
:63)
jvm 1 |      at
org.apache.activemq.console.command.ShellCommand.main(ShellCommand.java:104)
```

```
jvm 1 |      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
jvm 1 |      at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
jvm 1 |      at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav
a:43)
jvm 1 |      at java.lang.reflect.Method.invoke(Method.java:498)
jvm 1 |      at org.apache.activemq.console.Main.runTaskClass(Main.java:262)
jvm 1 |      at org.apache.activemq.console.Main.main(Main.java:115)
jvm 1 |      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
jvm 1 |      at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
jvm 1 |      at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav
a:43)
jvm 1 |      at java.lang.reflect.Method.invoke(Method.java:498)
jvm 1 |      at
org.tanukisoftware.wrapper.WrapperSimpleApp.run(WrapperSimpleApp.java:240)
jvm 1 |      at java.lang.Thread.run(Thread.java:748)
jvm 1 | Caused by: java.lang.IllegalStateException: BeanFactory not
initialized or already closed - call 'refresh' before accessing beans via the
ApplicationContext
jvm 1 |      at
org.springframework.context.support.AbstractRefreshableApplicationContext.getBean
Factory(AbstractRefreshableApplicationContext.java:164)
jvm 1 |      at
org.springframework.context.support.AbstractApplicationContext.destroyBeans(Abst
ractApplicationContext.java:1034)
jvm 1 |      at
org.springframework.context.support.AbstractApplicationContext.refresh(AbstractA
pplicationContext.java:555)
jvm 1 |      at
org.apache.xmlbeans.spring.context.ResourceXmlApplicationContext.<init>
(ResourceXmlApplicationContext.java:64)
jvm 1 |      at
org.apache.xmlbeans.spring.context.ResourceXmlApplicationContext.<init>
(ResourceXmlApplicationContext.java:52)
jvm 1 |      at org.apache.activemq.xbean.XBeanBrokerFactory$1.<init>
(XBeanBrokerFactory.java:104)
jvm 1 |      at
org.apache.activemq.xbean.XBeanBrokerFactory.createApplicationContext(XBeanBroke
rFactory.java:104)
jvm 1 |      at
org.apache.activemq.xbean.XBeanBrokerFactory.createBroker(XBeanBrokerFactory.jav
a:67)
jvm 1 |      at
org.apache.activemq.broker.BrokerFactory.createBroker(BrokerFactory.java:71)
jvm 1 |      at
org.apache.activemq.broker.BrokerFactory.createBroker(BrokerFactory.java:54)
jvm 1 |      at
org.apache.activemq.console.command.StartCommand.runTask(StartCommand.java:87)
jvm 1 |      ... 16 more
jvm 1 | ERROR: java.lang.IllegalStateException: BeanFactory not initialized
or already closed - call 'refresh' before accessing beans via the
ApplicationContext
```

```
jvm 1 | java.lang.IllegalStateException: BeanFactory not initialized or
already closed - call 'refresh' before accessing beans via the
ApplicationContext
jvm 1 |     at
org.springframework.context.support.AbstractRefreshableApplicationContext.getBeanFactory(AbstractRefreshableApplicationContext.java:164)
jvm 1 |     at
org.springframework.context.support.AbstractApplicationContext.destroyBeans(AbstractApplicationContext.java:1034)
jvm 1 |     at
org.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicationContext.java:555)
jvm 1 |     at
org.apache.xbean.spring.context.ResourceXmlApplicationContext.<init>
(ResourceXmlApplicationContext.java:64)
jvm 1 |     at
org.apache.xbean.spring.context.ResourceXmlApplicationContext.<init>
(ResourceXmlApplicationContext.java:52)
jvm 1 |     at org.apache.activemq.xbean.XBeanBrokerFactory$1.<init>
(XBeanBrokerFactory.java:104)
jvm 1 |     at
org.apache.activemq.xbean.XBeanBrokerFactory.createApplicationContext(XBeanBrokerFactory.java:104)
jvm 1 |     at
org.apache.activemq.xbean.XBeanBrokerFactory.createBroker(XBeanBrokerFactory.java:67)
jvm 1 |     at
org.apache.activemq.broker.BrokerFactory.createBroker(BrokerFactory.java:71)
jvm 1 |     at
org.apache.activemq.broker.BrokerFactory.createBroker(BrokerFactory.java:54)
jvm 1 |     at
org.apache.activemq.console.command.StartCommand.runTask(StartCommand.java:87)
jvm 1 |     at
org.apache.activemq.console.command.AbstractCommand.execute(AbstractCommand.java:63)
jvm 1 |     at
org.apache.activemq.console.command.ShellCommand.runTask(ShellCommand.java:154)
jvm 1 |     at
org.apache.activemq.console.command.AbstractCommand.execute(AbstractCommand.java:63)
jvm 1 |     at
org.apache.activemq.console.command.ShellCommand.main(ShellCommand.java:104)
jvm 1 |     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
jvm 1 |     at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
jvm 1 |     at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
jvm 1 |     at java.lang.reflect.Method.invoke(Method.java:498)
jvm 1 |     at org.apache.activemq.console.Main.runTaskClass(Main.java:262)
jvm 1 |     at org.apache.activemq.console.Main.main(Main.java:115)
jvm 1 |     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
jvm 1 |     at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
```

```
jvm 1 |      at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.jav
a:43)
jvm 1 |      at java.lang.reflect.Method.invoke(Method.java:498)
jvm 1 |      at
org.tanukisoftware.wrapper.WrapperSimpleApp.run(wrapperSimpleApp.java:240)
jvm 1 |      at java.lang.Thread.run(Thread.java:748)
wrapper | <-- Wrapper Stopped
请按任意键继续. . .
```

整合

做了这么多springboot整合第三方技术，已经摸到门路了，加坐标，做配置，调接口，直接开工

步骤①：导入springboot整合ActiveMQ的starter

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

步骤②：配置ActiveMQ的服务器地址

```
spring:
  activemq:
    broker-url: tcp://localhost:61616
```

步骤③：使用JmsMessagingTemplate操作ActiveMQ

```
@Service
public class MessageServiceActivemqImpl implements MessageService {
    @Autowired
    private JmsMessagingTemplate messagingTemplate;

    @Override
    public void sendMessage(String id) {
        System.out.println("待发送短信的订单已纳入处理队列, id: " + id);
        messagingTemplate.convertAndSend("order.queue.id", id);
    }

    @Override
    public String doMessage() {
        String id =
messagingTemplate.receiveAndConvert("order.queue.id", String.class);
        System.out.println("已完成短信发送业务, id: " + id);
        return id;
    }
}
```

发送消息需要先将消息的类型转换成字符串，然后再发送，所以是convertAndSend，定义消息发送的位置，和具体的消息内容，此处使用id作为消息内容。

接收消息需要先将消息接收到，然后再转换成指定的数据类型，所以是receiveAndConvert，接收消息除了提供读取的位置，还要给出转换后的数据的具体类型。

步骤④：使用消息监听器在服务器启动后，监听指定位置，当消息出现后，立即消费消息

```
@Component
public class MessageListener {
    @JmsListener(destination = "order.queue.id")
    //这个注解可以把这个处理结果的返回值传递到新的消息队列
    @SendTo("order.other.queue.id")
    public String receive(String id){
        System.out.println("已完成短信发送业务, id: "+id);
        return "new:"+id;
    }
}
```

使用注解@JmsListener定义当前方法监听ActiveMQ中指定名称的消息队列。

如果当前消息队列处理完还需要继续向下传递当前消息到另一个队列中使用注解@SendTo即可，这样即可构造连续执行的顺序消息队列。

步骤⑤：切换消息模型由点对点模型到发布订阅模型，修改jms配置即可

```
spring:
  activemq:
    broker-url: tcp://localhost:61616
  jms:
    pub-sub-domain: true
```

pub-sub-domain默认值为false，即点对点模型，修改为true后就是发布订阅模型。

总结

1. springboot整合ActiveMQ提供了JmsMessagingTemplate对象作为客户端操作消息队列
2. 操作ActiveMQ需要配置ActiveMQ服务器地址，默认端口61616
3. 企业开发时通常使用监听器来处理消息队列中的消息，设置监听器使用注解@JmsListener
4. 配置jms的pub-sub-domain属性可以在点对点模型和发布订阅模型间切换消息模型

SpringBoot整合RabbitMQ

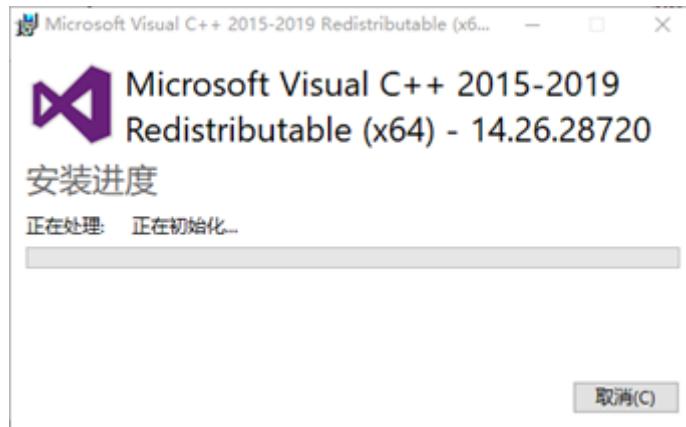
RabbitMQ是MQ产品中的目前较为流行的产品之一，它遵从AMQP协议。RabbitMQ的底层实现语言使用的是Erlang，所以安装RabbitMQ需要先安装Erlang。

Erlang安装

windows版安装包下载地址：<https://www.erlang.org/downloads>

下载完毕后得到exe安装文件，一键傻瓜式安装，安装完毕需要重启，需要重启，需要重启。

安装的过程中可能会出现依赖Windows组件的提示，根据提示下载安装即可，都是自动执行的，如下：



Erlang安装后需要配置环境变量，否则RabbitMQ将无法找到安装的Erlang。需要配置项如下，作用等同JDK配置环境变量的作用。

- ERLANG_HOME
- PATH

安装

windows版安装包下载地址：<https://rabbitmq.com/install-windows.html>

下载完毕后得到exe安装文件，一键傻瓜式安装，安装完毕后会得到如下文件

名称	修改日期
escript	2022/2/9 15:20
etc	2022/2/9 15:20
plugins	2022/2/9 15:20
sbin	2022/2/9 15:20
INSTALL.txt	2022/1/19 7:14
LICENSE.txt	2022/1/19 7:14
LICENSE-APACHE2.txt	2022/1/19 7:14
LICENSE-APACHE2-excanvas.txt	2022/1/19 7:14
LICENSE-APACHE2-ExplorerCanvas.txt	2022/1/19 7:14
LICENSE-APL2-Stomp-WebSocket.txt	2022/1/19 7:14
LICENSE-BSD-base64js.txt	2022/1/19 7:14
LICENSE-BSD-recon.txt	2022/1/19 7:14
LICENSE-erlcloud.txt	2022/1/19 7:14
LICENSE-htcpc_aws.txt	2022/1/19 7:14
LICENSE-ISC-cowboy.txt	2022/1/19 7:14
LICENSE-MIT-EJS.txt	2022/1/19 7:14
LICENSE-MIT-EJS10.txt	2022/1/19 7:14
LICENSE-MIT-Erlware-Commons.txt	2022/1/19 7:14
LICENSE-MIT-Flot.txt	2022/1/19 7:14
LICENSE-MIT-jQuery.txt	2022/1/19 7:14
LICENSE-MIT-jQuery164.txt	2022/1/19 7:14
LICENSE-MIT-Mochi.txt	2022/1/19 7:14
LICENSE-MIT-Sammy.txt	2022/1/19 7:14
LICENSE-MIT-Sammy060.txt	2022/1/19 7:14
LICENSE-MPL.txt	2022/1/19 7:14
LICENSE-MPL2.txt	2022/1/19 7:14
LICENSE-MPL-RabbitMQ.txt	2022/1/19 7:14
LICENSE-rabbitmq_aws.txt	2022/1/19 7:14
readme-service.txt	2022/1/19 7:14

启动服务器

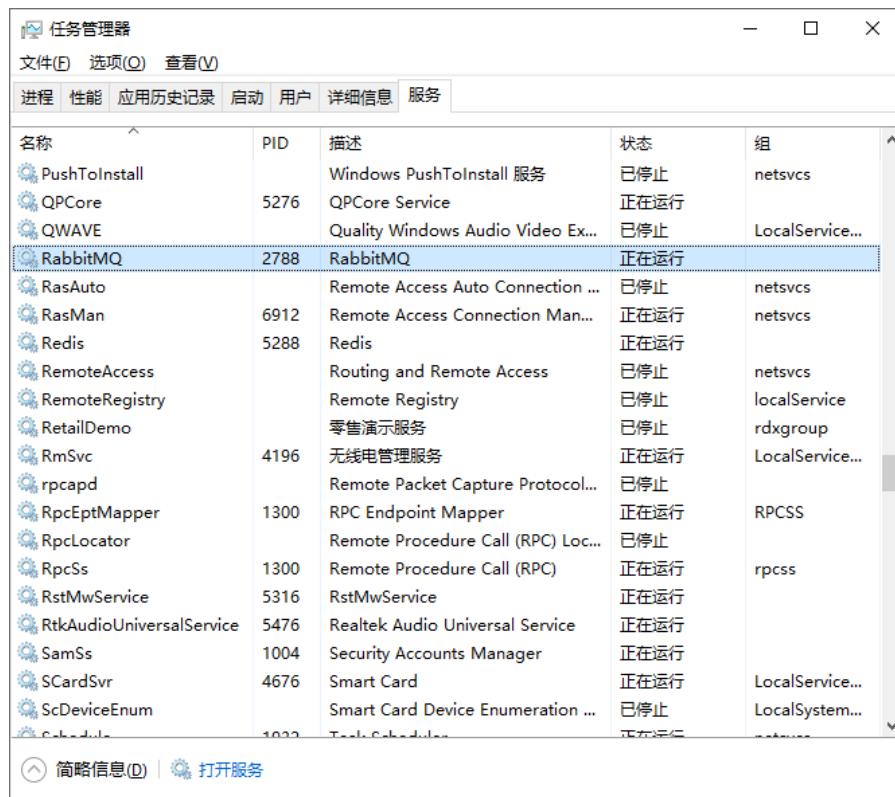
```
rabbitmq-service.bat start      # 启动服务  
rabbitmq-service.bat stop       # 停止服务  
rabbitmqctl status             # 查看服务状态
```

运行sbin目录下的rabbitmq-service.bat命令即可，start参数表示启动，stop参数表示退出，默认对外服务端口5672。

注意：启动rabbitmq的过程实际上是开启rabbitmq对应的系统服务，需要管理员权限方可执行。

说明：有没有感觉5672的服务端口很熟悉？activemq与rabbitmq有一个端口冲突问题，学习阶段无论操作哪一个？请确保另一个处于关闭状态。

说明：不喜欢命令行的小伙伴可以使用任务管理器中的服务页，找到RabbitMQ服务，使用鼠标右键菜单控制服务的启停。



访问web管理服务

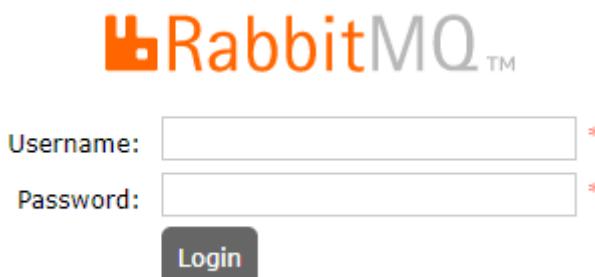
RabbitMQ也提供有web控制台服务，但是此功能是一个插件，需要先启用才可以使用。

```
rabbitmq-plugins.bat list          # 查看当前所有插件的运行状态  
rabbitmq-plugins.bat enable rabbitmq_management    # 启动rabbitmq_management插件
```

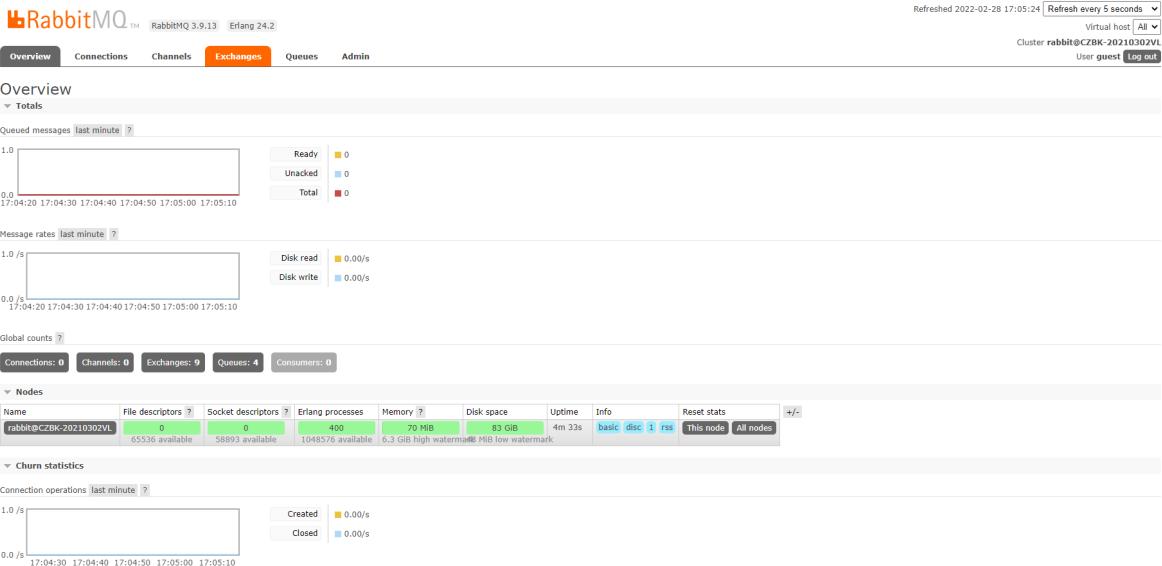
启动插件后可以在插件运行状态中查看是否运行，运行后通过浏览器即可打开服务后台管理界面

```
http://localhost:15672
```

web管理服务默认端口15672，访问后可以打开RabbitMQ的管理界面，如下：



首先输入访问用户名和密码，初始化用户名和密码相同，均为：guest，成功登录后进入管理后台界面，如下：



整合(direct模型)

RabbitMQ满足AMQP协议，因此不同的消息模型对应的制作不同，先使用最简单的direct模型开发。

步骤①：导入springboot整合amqp的starter， amqp协议默认实现为rabbitmq方案

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

步骤②：配置RabbitMQ的服务器地址

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
```

步骤③：初始化直连模式系统设置

由于RabbitMQ不同模型要使用不同的交换机，因此需要先初始化RabbitMQ相关的对象，例如队列，交换机等

```
@Configuration
public class RabbitConfigDirect {
    @Bean
    public Queue directQueue(){
        return new Queue("direct_queue");
    }
    @Bean
    public Queue directQueue2(){
        return new Queue("direct_queue2");
    }
    @Bean
    public DirectExchange directExchange(){
        return new DirectExchange("directExchange");
    }
}
```

```

@Bean
public Binding bindingDirect(){
    return
BindingBuilder.bind(directQueue()).to(directExchange()).with("direct");
}

@Bean
public Binding bindingDirect2(){
    return
BindingBuilder.bind(directQueue2()).to(directExchange()).with("direct2");
}

```

队列Queue与直连交换机DirectExchange创建后，还需要绑定他们之间的关系Binding，这样就可以通过交换机操作对应队列。

步骤④：使用AmqpTemplate操作RabbitMQ

```

@Service
public class MessageServiceRabbitmqDirectImpl implements MessageService {
    @Autowired
    private AmqpTemplate amqpTemplate;

    @Override
    public void sendMessage(String id) {
        System.out.println("待发送短信的订单已纳入处理队列(rabbitmq direct),
id: "+id);
        amqpTemplate.convertAndSend("directExchange", "direct", id);
    }
}

```

amqp协议中的操作API接口名称看上去和jms规范的操作API接口很相似，但是传递参数差异很大。

步骤⑤：使用消息监听器在服务器启动后，监听指定位置，当消息出现后，立即消费消息

```

@Component
public class MessageListener {
    @RabbitListener(queues = "direct_queue")
    public void receive(String id){
        System.out.println("已完成短信发送业务(rabbitmq direct), id: "+id);
    }
}

```

使用注解@RabbitListener定义当前方法监听RabbitMQ中指定名称的消息队列。

整合(topic模型)

步骤①：同上

步骤②：同上

步骤③：初始化主题模式系统设置

```

@Configuration
public class RabbitConfigTopic {
    @Bean

```

```

public Queue topicQueue(){
    return new Queue("topic_queue");
}

@Bean
public Queue topicQueue2(){
    return new Queue("topic_queue2");
}

@Bean
public TopicExchange topicExchange(){
    return new TopicExchange("topicExchange");
}

@Bean
public Binding bindingTopic(){
    return
BindingBuilder.bind(topicQueue()).to(topicExchange()).with("topic.*.id");
}

@Bean
public Binding bindingTopic2(){
    return
BindingBuilder.bind(topicQueue2()).to(topicExchange()).with("topic.orders.*");
}

```

主题模式支持routingKey匹配模式，*表示匹配一个单词，#表示匹配任意内容，这样就可以通过主题交换机将消息分发到不同的队列中，详细内容请参看RabbitMQ系列课程。

匹配键	topic.*.*	topic.#
topic.order.id	true	true
order.topic.id	false	false
topic.sm.order.id	false	true
topic.sm.id	false	true
topic.id.order	true	true
topic.id	false	true
topic.order	false	true

步骤④：使用AmqpTemplate操作RabbitMQ

```

@Service
public class MessageServiceRabbitmqTopicImpl implements MessageService {
    @Autowired
    private AmqpTemplate amqpTemplate;

    @Override
    public void sendMessage(String id) {
        System.out.println("待发送短信的订单已纳入处理队列(rabbitmq topic), id: " + id);
        amqpTemplate.convertAndSend("topicExchange", "topic.orders.id", id);
    }
}

```

发送消息后，根据当前提供的routingKey与绑定交换机时设定的routingKey进行匹配，规则匹配成功消息才会进入到对应的队列中。

步骤⑤：使用消息监听器在服务器启动后，监听指定队列

```

@Component
public class MessageListener {
    @RabbitListener(queues = "topic_queue")
    public void receive(String id){
        System.out.println("已完成短信发送业务(rabbitmq topic 1), id: " + id);
    }
    @RabbitListener(queues = "topic_queue2")
    public void receive2(String id){
        System.out.println("已完成短信发送业务(rabbitmq topic 22222222), id: " + id);
    }
}

```

使用注解@RabbitListener定义当前方法监听RabbitMQ中指定名称的消息队列。

总结

1. springboot整合RabbitMQ提供了AmqpTemplate对象作为客户端操作消息队列
2. 操作ActiveMQ需要配置ActiveMQ服务器地址，默认端口5672
3. 企业开发时通常使用监听器来处理消息队列中的消息，设置监听器使用注解@RabbitListener
4. RabbitMQ有5种消息模型，使用的队列相同，但是交换机不同。交换机不同，对应的消息进入的策略也不同

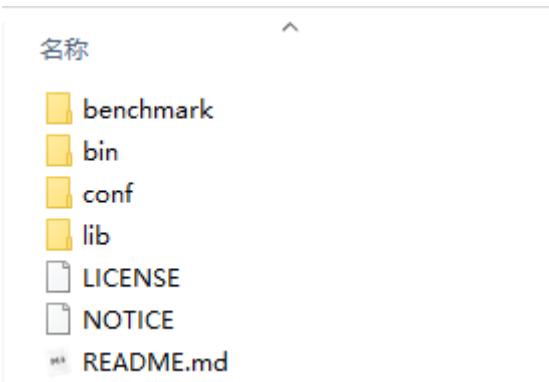
SpringBoot整合RocketMQ

RocketMQ由阿里研发，后捐赠给apache基金会，目前是apache基金会顶级项目之一，也是目前市面上的MQ产品中较为流行的产品之一，它遵从AMQP协议。

安装

windows版安装包下载地址：<https://rocketmq.apache.org/>

下载完毕后得到zip压缩文件，解压缩即可使用，解压后得到如下文件



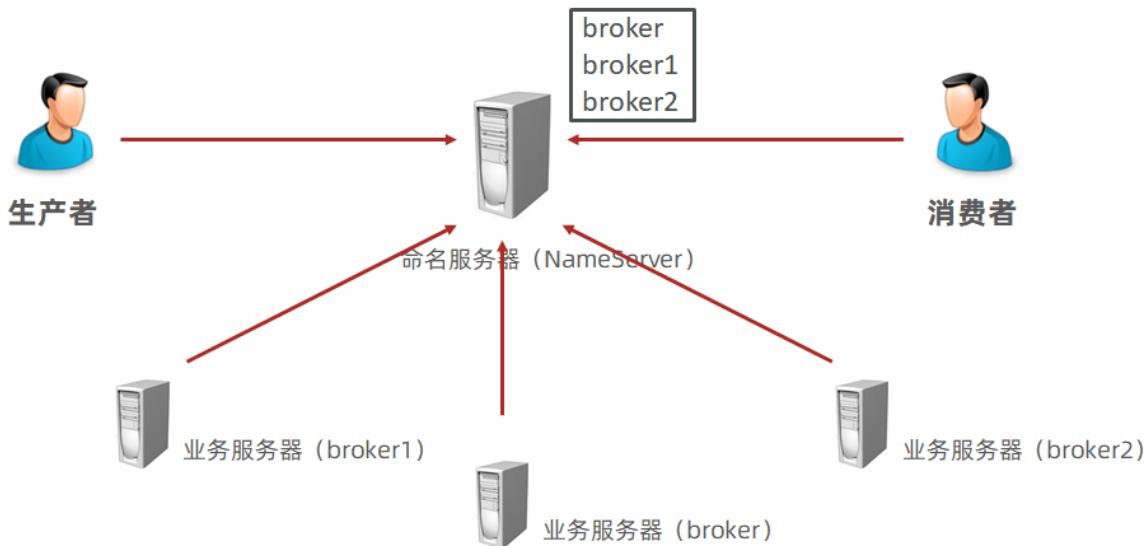
RocketMQ安装后需要配置环境变量，具体如下：

- ROCKETMQ_HOME
- PATH
- NAMESRV_ADDR (建议) : 127.0.0.1:9876

关于NAMESRV_ADDR对于初学者来说建议配置此项，也可以通过命令设置对应值，操作略显繁琐，建议配置。系统学习RocketMQ知识后即可灵活控制该项。

RocketMQ工作模式

在RocketMQ中，处理业务的服务器称为broker，生产者与消费者不是直接与broker联系的，而是通过命名服务器进行通信。broker启动后会通知命名服务器自己已经上线，这样命名服务器中就保存有所有的broker信息。当生产者与消费者需要连接broker时，通过命名服务器找到对应的处理业务的broker，因此命名服务器在整套结构中起到一个信息中心的作用。并且broker启动前必须保障命名服务器先启动。



启动服务器

```
mqnamesrv      # 启动命名服务器
mqbroker       # 启动broker
```

运行bin目录下的mqnamesrv命令即可启动命名服务器，默认对外服务端口9876。

运行bin目录下的mqbroker命令即可启动broker服务器，如果环境变量中没有设置NAMESRV_ADDR则需要在运行mqbroker指令前通过set指令设置NAMESRV_ADDR的值，并且每次开启均需要设置此项。

测试服务器启动状态

RocketMQ提供有一套测试服务器功能的测试程序，运行bin目录下的tools命令即可使用。

```
tools org.apache.rocketmq.example.quickstart.Producer      # 生产消息  
tools org.apache.rocketmq.example.quickstart.Consumer    # 消费消息
```

整合（异步消息）

步骤①：导入springboot整合RocketMQ的starter，此坐标不由springboot维护版本

```
<dependency>  
  <groupId>org.apache.rocketmq</groupId>  
  <artifactId>rocketmq-spring-boot-starter</artifactId>  
  <version>2.2.1</version>  
</dependency>
```

步骤②：配置RocketMQ的服务器地址

```
rocketmq:  
  name-server: localhost:9876  
  producer:  
    group: group_rocketmq #自定义的名称
```

设置默认的生产者消费者所属组group。

步骤③：使用RocketMQTemplate操作RocketMQ

```
@Service  
public class MessageServiceRocketmqImpl implements MessageService {  
    @Autowired  
    private RocketMQTemplate rocketMQTemplate;  
  
    @Override  
    public void sendMessage(String id) {  
        System.out.println("待发送短信的订单已纳入处理队列(rocketmq), id: " + id);  
        SendCallback callback = new SendCallback() {  
            @Override  
            public void onSuccess(SendResult sendResult) {  
                System.out.println("消息发送成功");  
            }  
            @Override  
            public void onException(Throwable e) {  
                System.out.println("消息发送失败!!!!");  
            }  
        };  
        rocketMQTemplate.asyncSend("order_id", id, callback);  
    }  
}
```

使用asyncSend方法发送异步消息。

步骤④：使用消息监听器在服务器启动后，监听指定位置，当消息出现后，立即消费消息

```
@Component
@RocketMQMessageListener(topic = "order_id", consumerGroup = "group_rocketmq")
public class MessageListener implements RocketMQListener<String> {
    @Override
    public void onMessage(String id) {
        System.out.println("已完成短信发送业务(rocketmq), id: "+id);
    }
}
```

RocketMQ的监听器必须按照标准格式开发，实现RocketMQListener接口，泛型为消息类型。

使用注解@RocketMQMessageListener定义当前类监听RabbitMQ中指定组、指定名称的消息队列。

总结

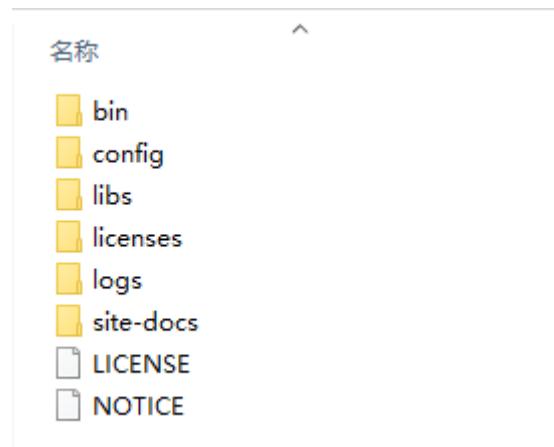
1. springboot整合RocketMQ使用RocketMQTemplate对象作为客户端操作消息队列
2. 操作RocketMQ需要配置RocketMQ服务器地址，默认端口9876
3. 企业开发时通常使用监听器来处理消息队列中的消息，设置监听器使用注解
 @RocketMQMessageListener

SpringBoot整合Kafka

安装

windows版安装包下载地址：<https://kafka.apache.org/downloads>

下载完毕后得到tgz压缩文件，使用解压缩软件解压缩即可使用，解压后得到如下文件



建议使用windows版2.8.1版本。

启动服务器

kafka服务器的功能相当于RocketMQ中的broker，kafka运行还需要一个类似于命名服务器的服务。在kafka安装目录中自带一个类似于命名服务器的工具，叫做zookeeper，它的作用是注册中心，相关知识请到对应课程中学习。

```
zookeeper-server-start.bat ..\..\config\zookeeper.properties      # 启动zookeeper
kafka-server-start.bat ..\..\config\server.properties           # 启动kafka
```

运行bin目录下的windows目录下的zookeeper-server-start命令即可启动注册中心， 默认对外服务端口2181。

运行bin目录下的windows目录下的kafka-server-start命令即可启动kafka服务器， 默认对外服务端口9092。

创建主题

和之前操作其他MQ产品相似， kafka也是基于主题操作， 操作之前需要先初始化topic。

```
# 创建topic
kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --
partitions 1 --topic itheima
# 查询topic
kafka-topics.bat --zookeeper 127.0.0.1:2181 --list
# 删除topic
kafka-topics.bat --delete --zookeeper localhost:2181 --topic itheima
```

测试服务器启动状态

Kafka提供有一套测试服务器功能的测试程序， 运行bin目录下的windows目录下的命令即可使用。

```
kafka-console-producer.bat --broker-list localhost:9092 --topic itheima
# 测试生产消息
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic itheima --
from-beginning # 测试消息消费
```

整合

步骤①： 导入springboot整合Kafka的starter， 此坐标由springboot维护版本

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

步骤②： 配置Kafka的服务器地址

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: order
```

设置默认的生产者消费者所属组id。

步骤③： 使用KafkaTemplate操作Kafka

```

@Service
public class MessageServiceKafkaImpl implements MessageService {
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    @Override
    public void sendMessage(String id) {
        System.out.println("待发送短信的订单已纳入处理队列(kafka), id: " + id);
        kafkaTemplate.send("itheima2022", id);
    }
}

```

使用send方法发送消息，需要传入topic名称。

步骤④： 使用消息监听器在服务器启动后，监听指定位置，当消息出现后，立即消费消息

```

@Component
public class MessageListener {
    @KafkaListener(topics = "itheima2022")
    public void onMessage(ConsumerRecord<String, String> record) {
        System.out.println("已完成短信发送业务(kafka), id: " + record.value());
    }
}

```

使用注解@KafkaListener定义当前方法监听Kafka中指定topic的消息，接收到的消息封装在对象ConsumerRecord中，获取数据从ConsumerRecord对象中获取即可。

总结

1. springboot整合Kafka使用KafkaTemplate对象作为客户端操作消息队列
2. 操作Kafka需要配置Kafka服务器地址，默认端口9092
3. 企业开发时通常使用监听器来处理消息队列中的消息，设置监听器使用注解@KafkaListener。接收消息保存在形参ConsumerRecord对象中

KF-6.监控

在说监控之前，需要回顾一下软件业的发展史。最早的软件完成一些非常简单的功能，代码不多，错误也少。随着软件功能的逐步完善，软件的功能变得越来越复杂，功能不能得到有效的保障，这个阶段出现了针对软件功能的检测，也就是软件测试。伴随着计算机操作系统的逐步升级，软件的运行状态也变得开始让人捉摸不透，出现了不稳定的状况。伴随着计算机网络的发展，程序也从单机状态切换成基于计算机网络的程序，应用于网络的程序开始出现，由于网络的不稳定性，程序的运行状态让使用者更加堪忧。互联网的出现彻底打破了软件的思维模式，随之而来的互联网软件就更加凸显出应对各种各样复杂的网络情况之下的弱小。计算机软件的运行状况已经成为了软件运行的一个大话题，针对软件的运行状况就出现了全新的思维，建立起了初代的软件运行状态监控。

什么是监控？就是通过软件的方式展示另一个软件的运行情况，运行的情况则通过各种各样的指标数据反馈给监控人员。例如网络是否顺畅、服务器是否在运行、程序的功能是否能够整百分百运行成功，内存是否够用，等等等。

本章要讲解的监控就是对软件的运行情况进行监督，但是springboot程序与非springboot程序的差异还是很大的，为了方便监控软件的开发，springboot提供了一套功能接口，为开发者加速开发过程。

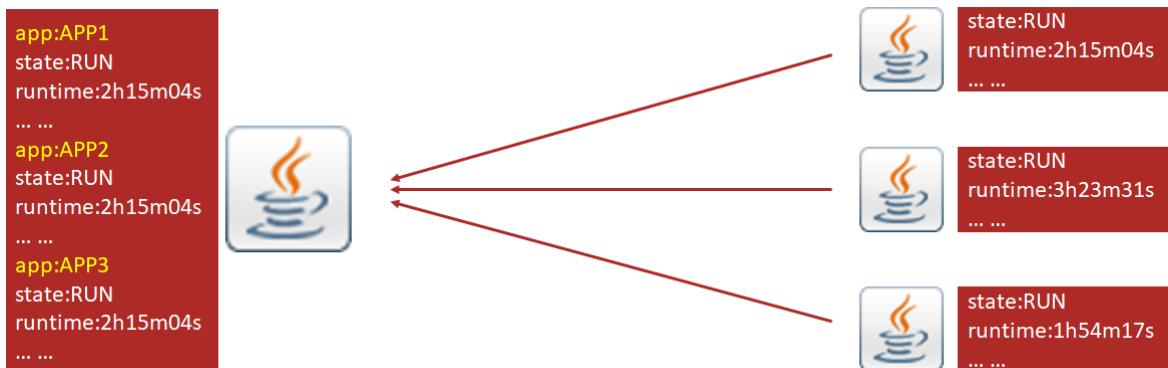
KF-6-1.监控的意义

对于现代的互联网程序来说，规模越来越大，功能越来越复杂，还要追求更好的客户体验，因此要监控的信息量也就比较大了。由于现在的互联网程序大部分都是基于微服务的程序，一个程序的运行需要若干个服务来保障，因此第一个要监控的指标就是服务是否正常运行，也就是**监控服务状态是否处理宕机状态**。一旦发现某个服务宕机了，必须马上给出对应的解决方案，避免整体应用功能受影响。其次，由于互联网程序服务的客户量是巨大的，当客户的请求在短时间内集中达到服务器后，就会出现各种程序运行指标的波动。比如内存占用严重，请求无法及时响应处理等，这就是第二个要监控的重要指标，**监控服务运行指标**。虽然软件是对外提供用户的访问需求，完成对应功能的，但是后台的运行是否平稳，是否出现了不影响客户使用的功能隐患，这些也是要密切监控的，此时就需要在不停机的情况下，监控系统运行情况，日志是一个不错的手段。如果在众多日志中找到开发者或运维人员所关注的日志信息，简单快速有效的过滤出要看的日志也是监控系统需要考虑的问题，这就是第三个要监控的指标，**监控程序运行日志**。虽然我们期望程序一直平稳运行，但是由于突发情况的出现，例如服务器被攻击、服务器内存溢出等情况造成了服务器宕机，此时当前服务不能满足使用需要，就要将其重启甚至关闭，如果快速控制服务器的启停也是程序运行过程中不可回避的问题，这就是第四个监控项，**管理服务状态**。以上这些仅仅是从大的方面来思考监控这个问题，还有很多的细节点，例如上线了一个新功能，定时提醒用户续费，这种功能不是上线后马上就运行的，但是当前功能是否真的启动，如果快速的查询到这个功能已经开启，这也是监控中要解决的问题，等等。看来监控真的是一项非常重要的工作。

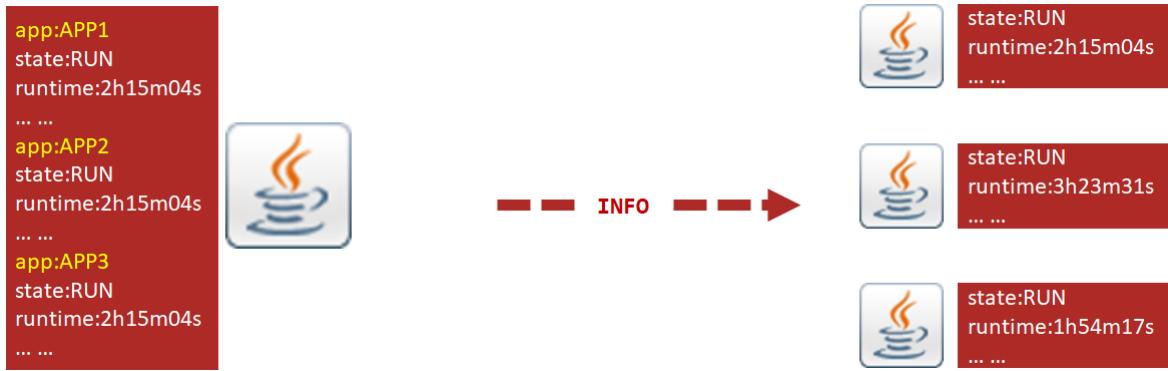
通过上述描述，可以看出监控很重要。那具体的监控要如何开展呢？还要从实际的程序运行角度出发。比如现在有3个服务支撑着一个程序的运行，每个服务都有自己的运行状态。



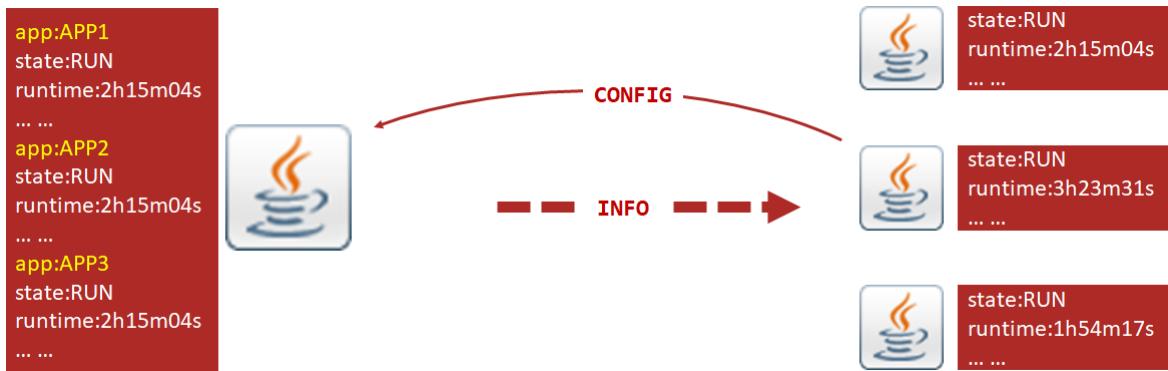
此时被监控的信息就要在三个不同的程序中去查询并展示，但是三个服务是服务于一个程序的运行的，如果不能合并到一个平台上展示，监控工作量巨大，而且信息对称性差，要不停的在三个监控端查看数据。如果将业务放大成30个，300个，3000个呢？看来必须有一个单独的平台，将多个被监控的服务对应的监控指标信息汇总在一起，这样更利于监控工作的开展。



新的程序专门用来监控，新的问题就出现了，是被监控程序主动上报信息还是监控程序主动获取信息？如果监控程序不能主动获取信息，这就意味着监控程序有可能看到的是很久之前被监控程序上报的信息，万一被监控程序宕机了，监控程序就无法区分究竟是好久没法信息了，还是已经下线了。所以监控程序必须具有主动发起请求获取被监控服务信息的能力。



如果监控程序要监控服务时，主动获取对方的信息。那监控程序如何知道哪些程序被自己监控呢？不可能在监控程序中设置我监控谁，这样互联网上的所有程序岂不是都可以被监控到，这样的话信息安全将无法得到保障。合理地做法只能是在被监控程序启动时上报监控程序，告诉监控程序你可以监控我了。看来需要在被监控程序端做主动上报的操作，这就要求被监控程序中配置对应的监控程序是谁。



被监控程序可以提供各种各样的指标数据给监控程序看，但是每一个指标都代表着公司的机密信息，并不是所有的指标都可以给任何人看的，乃至运维人员，所以对被监控指标的是否开放出来给监控系统看，也需要做详细的设定。

以上描述的整个过程就是一个监控系统的基本流程。

总结

1. 监控是一个非常重要的工作，是保障程序正常运行的基础手段
2. 监控的过程通过一个监控程序进行，它汇总所有被监控的程序的信息集中统一展示
3. 被监控程序需要主动上报自己被监控，同时要设置哪些指标被监控

思考

下面就要开始做监控了，新的问题就来了，监控程序怎么做呢？难道要自己写吗？肯定是不现实的，如何进行监控，咱们下节再讲。

KF-6-2. 可视化监控平台

springboot抽取了大部分监控系统的常用指标，提出了监控的总思想。然后就有好心的同志根据监控的总思想，制作了一个通用性很强的监控系统，因为是基于springboot监控的核心思想制作的，所以这个程序被命名为**Spring Boot Admin**。

Spring Boot Admin，这是一个开源社区项目，用于管理和监控SpringBoot应用程序。这个项目中包含有客户端和服务端两部分，而监控平台指的就是服务端。我们做的程序如果需要被监控，将我们做的程序制作成客户端，然后配置服务端地址后，服务端就可以通过HTTP请求的方式从客户端获取对应的信息，并通过UI界面展示对应信息。

下面就来开发这套监控程序，先制作服务端，其实服务端可以理解为是一个web程序，收到一些信息后展示这些信息。

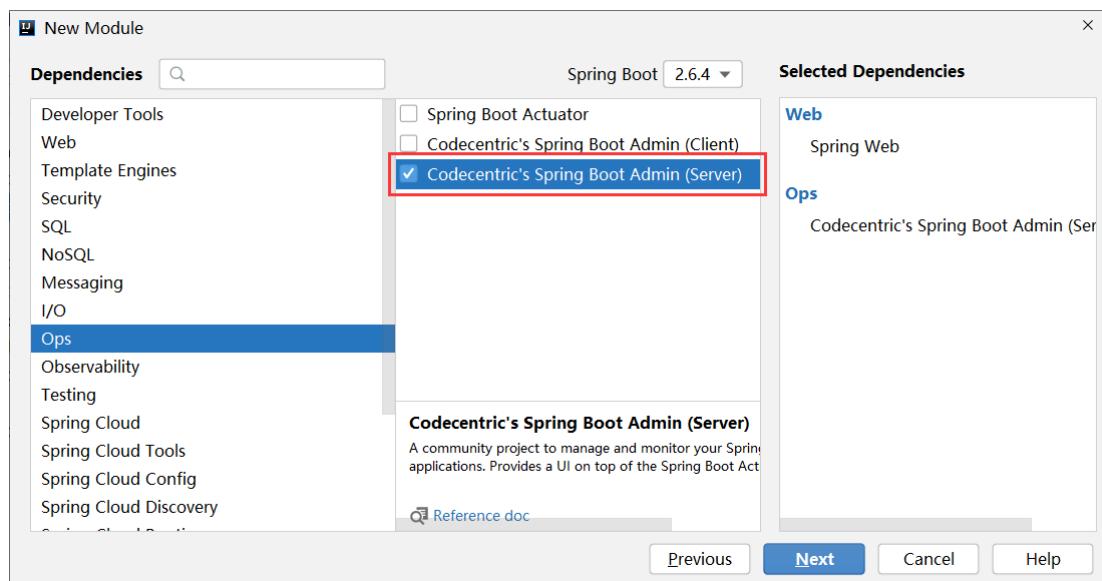
服务端开发

步骤①：导入springboot admin对应的starter，版本与当前使用的springboot版本保持一致，并将其配置成web工程

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>2.5.4</version>
</dependency>

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

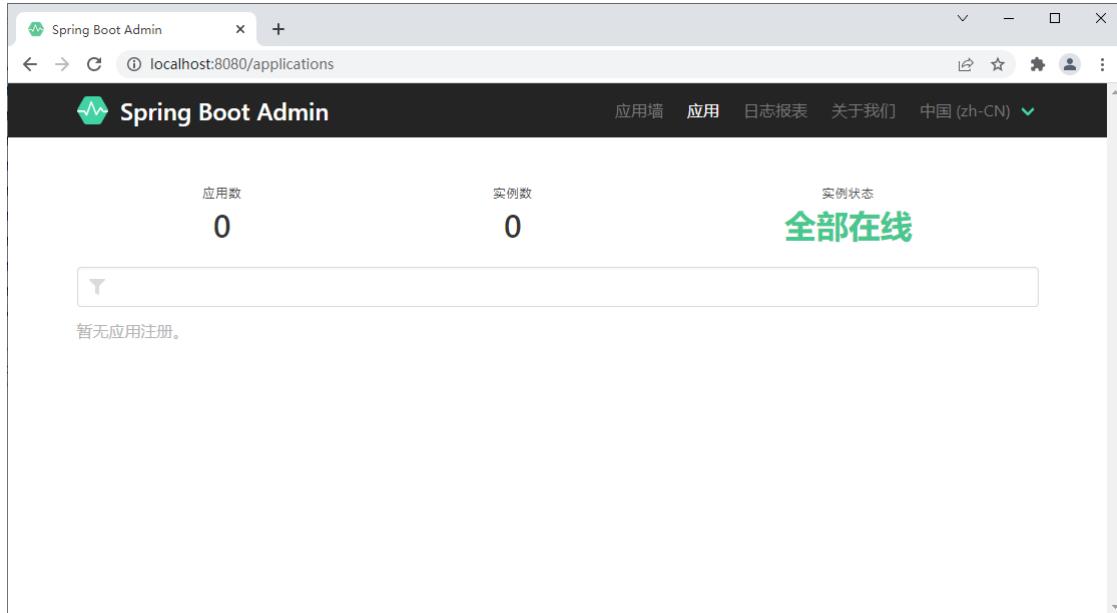
上述过程可以通过创建项目时使用勾选的形式完成。



步骤②：在引导类上添加注解@EnableAdminServer，声明当前应用启动后作为SpringBootAdmin的服务器使用

```
@SpringBootApplication
@EnableAdminServer
public class Springboot25AdminServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot25AdminServerApplication.class, args);
    }
}
```

做到这里，这个服务器就开发好了，启动后就可以访问当前程序了，界面如下。



由于目前没有启动任何被监控的程序，所以里面什么信息都没有。下面制作一个被监控的客户端程序。

客户端开发

客户端程序开发其实和服务端开发思路基本相似，多了一些配置而已。

步骤①：导入springboot admin对应的starter，版本与当前使用的springboot版本保持一致，并将其配置成web工程

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.5.4</version>
</dependency>

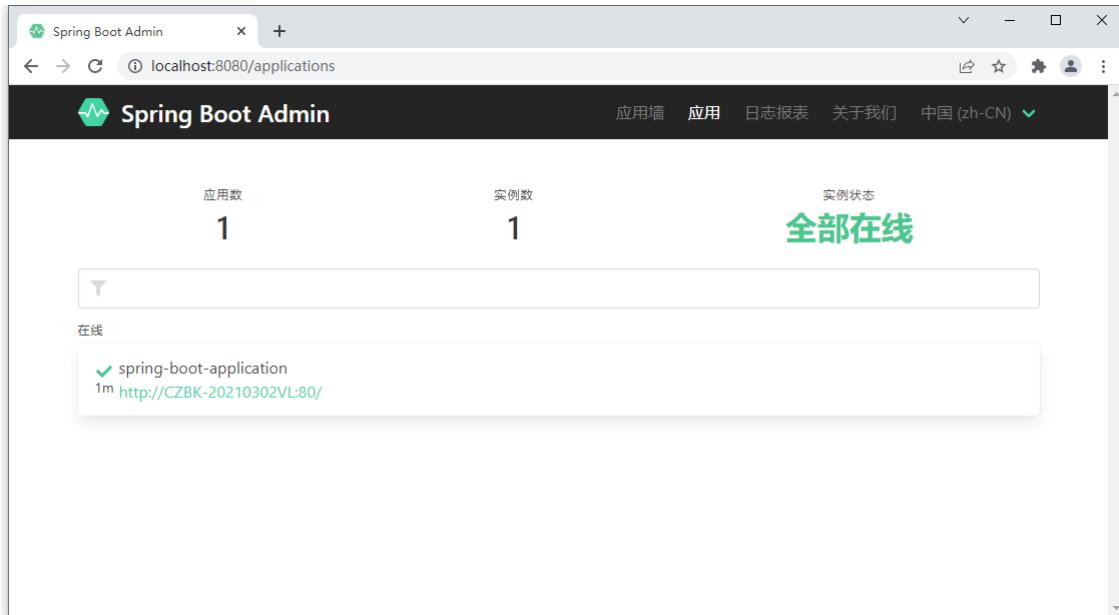
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

上述过程也可以通过创建项目时使用勾选的形式完成，不过一定要小心，端口配置成不一样的，否则会冲突。

步骤②：设置当前客户端将信息上传到哪个服务器上，通过yml文件配置

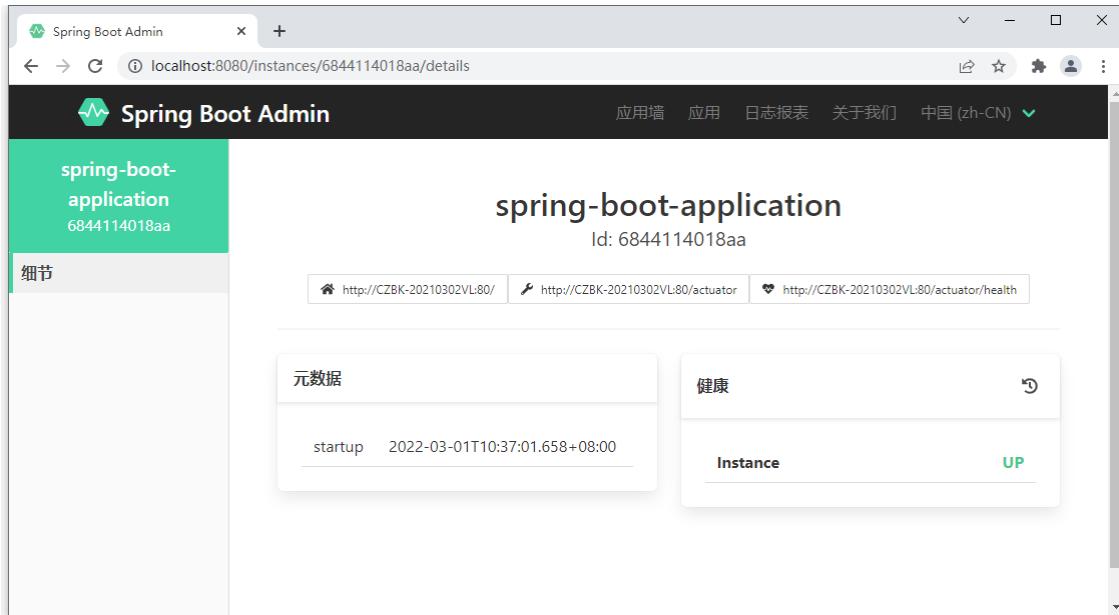
```
spring:  
  boot:  
    admin:  
      client:  
        url: http://localhost:8080
```

做到这里，这个客户端就可以启动了。启动后再次访问服务端程序，界面如下。



The screenshot shows the Spring Boot Admin interface at <http://localhost:8080/applications>. At the top, there are three summary statistics: 应用数 (1), 实例数 (1), and 实例状态 (全部在线). Below this, a section titled '在线' lists a single application instance: 'spring-boot-application' with ID '6844114018aa'. The URL for this instance is <http://CZBK-20210302VL:80/>.

可以看到，当前监控了1个程序，点击进去查看详细信息。



The screenshot shows the detailed view for the 'spring-boot-application' instance. The left sidebar shows the application name and ID. The main area displays the application name 'spring-boot-application' and ID '6844114018aa'. Below this, there are two tabs: '元数据' (Metadata) and '健康' (Health). The '元数据' tab shows the startup timestamp: 'startup 2022-03-01T10:37:01.658+08:00'. The '健康' tab shows the status as 'UP'. Navigation links for the application are provided at the top: <http://CZBK-20210302VL:80/>, <http://CZBK-20210302VL:80/actuator>, and <http://CZBK-20210302VL:80/actuator/health>.

由于当前没有设置开放哪些信息给监控服务器，所以目前看不到什么有效的信息。下面需要做两组配置就可以看到信息了。

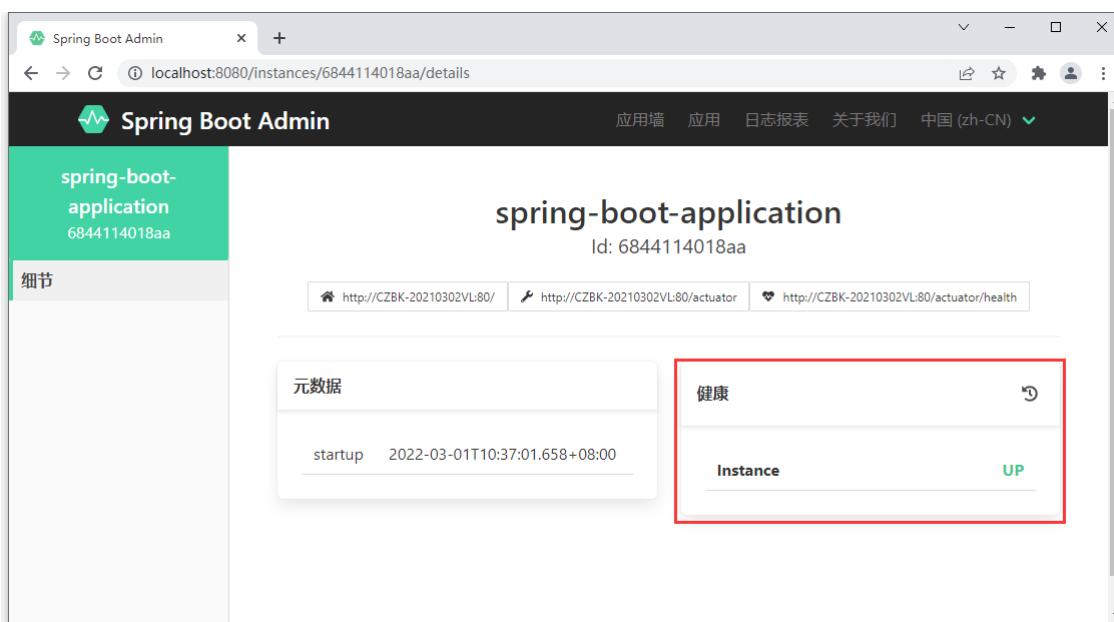
1. 开放指定信息给服务器看
2. 允许服务器以HTTP请求的方式获取对应的信息

配置如下：

```
server:  
  port: 80  
spring:
```

```
boot:
  admin:
    client:
      url: http://localhost:8080
management:
  endpoint:
    health:
      show-details: always
endpoints:
  web:
    exposure:
      include: "*"
```

上述配置对于初学者来说比较容易混淆。简单解释一下，到下一节再做具体的讲解。
springbootadmin的客户端默认开放了13组信息给服务器，但是这些信息除了一个之外，其他的信息都不让通过HTTP请求查看。所以你看到的信息基本上就没什么内容了，只能看到一个内容，就是下面的健康信息。



但是即便如此我们看到健康信息中也没什么内容，原因在于健康信息中有一些信息描述了你当前应用使用了什么技术等信息，如果无脑的对外暴露功能会有安全隐患。通过配置就可以开放所有的健康信息明细查看了。

```
management:
  endpoint:
    health:
      show-details: always
```

健康明细信息如下：

The screenshot shows the Spring Boot Admin interface for a Spring Boot application with ID 6844114018aa. The left sidebar lists sections like '细节' (Details), '性能' (Performance), '环境' (Environment), '类' (Classes), '配置属性' (Config Properties), '计划任务' (Scheduled Tasks), '日志配置' (Log Configuration), 'JVM', '映射' (Mappings), and '缓存' (Caches). The main content area displays the application's name, ID, and startup timestamp. It also shows a '健康' (Health) section with a table for disk space and a 'ping' status.

目前除了健康信息，其他信息都查阅不了。原因在于其他12种信息是默认不提供给服务器通过HTTP请求查阅的，所以需要开启查阅的内容项，使用*表示查阅全部。记得带引号。

```
endpoints:  
  web:  
    exposure:  
      include: "*"
```

配置后再刷新服务器页面，就可以看到所有的信息了。

The screenshot shows the same Spring Boot Admin interface after configuration. The left sidebar now includes all the listed sections: '细节', '性能', '环境', '类', '配置属性', '计划任务', '日志配置', 'JVM', '映射', and '缓存'. The main content area shows the application details and health status, with the '信息' (Information) section indicating '未提供任何信息' (No information provided).

以上界面中展示的信息量就非常大了，包含了13组信息，有性能指标监控，加载的bean列表，加载的系统属性，日志的显示控制等等。

配置多个客户端

可以通过配置客户端的方式在其他的springboot程序中添加客户端坐标，这样当前服务器就可以监控多个客户端程序了。每个客户端展示不同的监控信息。

The screenshot shows the Spring Boot Admin application running at localhost:8080. At the top, it displays "应用数 1" and "实例数 2". Below this, the status is shown as "全部在线" (All Online). The main content area is titled "spring-boot-application" and shows two instances listed under the "在线" (Online) section. Each instance has a green checkmark icon and its URL: "http://CZBK-20210302VL:80/" and "http://CZBK-20210302VL:81/". The URLs also include a long string of characters.

进入监控面板，如果你加载的应用具有功能，在监控面板中可以看到3组信息展示的与之前加载的空工程不一样。

- 类加载面板中可以查阅到开发者自定义的类，如左图

The screenshots show the 'beans' section of the Spring Boot Admin interface. The left sidebar lists various monitoring categories: Insights, 日志配置 (Log Configuration), JVM, 映射 (Mappings), and 缓存 (Cache). The right panel displays a table of beans with their names, descriptions, and scopes (singleton).

bean	细节	scope
MPConfig	com.theima.config.MPConfig\$\$EnhancerBySpringCGLIB\$\$20f63994	singleton
SSMApplication	com.theima.SSMApplication\$\$EnhancerBySpringCGLIB\$\$c5542e70	singleton
applicationAvailability	org.springframework.boot.availability.ApplicationAvailabilityBean	singleton
applicationFactory	de.codecentric.boot.admin.client.registration.ServletApplicationFactory	singleton
applicationTaskExecutor	org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor	singleton
basicErrorController	org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController	singleton

- 映射中可以查阅到当前应用配置的所有请求

The screenshots show the 'mappings' section of the Spring Boot Admin interface. The left sidebar lists monitoring categories: Insights, 日志配置 (Log Configuration), JVM, 映射 (Mappings), and 缓存 (Cache). The right panel displays a table of request mappings with their paths, methods, and handling programs.

path	方法	处理程序
/books	GET	com.theima.controller.BookController#getAll()
/books/{id}	DELETE	com.theima.controller.BookController#delete(Integer)
/books/{currentPage}/{pageSize}	GET	com.theima.controller.BookController#getPage(Integer, int, Book)
/books	PUT	com.theima.controller.BookController#update(Book)
/webjars/**		ResourceHttpRequestHandler [Classpath [META-INF/resources/webjars/]]
/**		ResourceHttpRequestHandler [Classpath [META-INF/resources/], Classpath [resources/]]

- 性能指标中可以查阅当前应用独有的请求路径统计数据

The image contains two side-by-side screenshots of the Spring Boot Admin application monitoring interface.

Screenshot 1 (Top): Shows the metrics for the application `spring-boot-application 44f452bc95bd`. The left sidebar has '性能' (Performance) selected. The main panel shows a search bar with filters: `http.server.requests`, `exception`, `method`, `uri /books/{currentPage}/{pageSize}`, `outcome`, and `status`. Below the search bar is a table with columns: `http.server.requests`, `COUNT`, `TOTAL_TIME`, and `MAX`. A single row is shown: `uri:/books/{currentPage}/{pageSize}` with COUNT: 1, TOTAL_TIME: 0.2609144, and MAX: 0. A red box highlights this row.

Screenshot 2 (Bottom): Shows the metrics for the application `spring-boot-application 6844114018aa`. The left sidebar has '性能' (Performance) selected. The main panel shows a search bar with filters: `http.server.requests`, `exception`, `method`, `uri`, `outcome`, and `status`. The `uri` dropdown is expanded, showing a list of endpoints: `/actuator/metrics/{requiredMetricName}`, `/actuator`, `/actuator/beans`, `/actuator/health`, `/actuator/info`, `/actuator/env/{toMatch}`, `/actuator/mappings`, and `/actuator/metrics`. A red box highlights the `uri` dropdown.

总结

1. 开发监控服务端需要导入坐标，然后在引导类上添加注解`@EnableAdminServer`，并将其配置成 web程序即可
2. 开发被监控的客户端需要导入坐标，然后配置服务端服务器地址，并做开放指标的设定即可
3. 在监控平台中可以查阅到各种各样被监控的指标，前提是客户端开放了被监控的指标

思考

之前说过，服务端要想监控客户端，需要主动的获取到对应信息并展示出来。但是目前我们并没有在客户端开发任何新的功能，但是服务端确可以获取监控信息，谁帮我们做的这些功能呢？咱们下一节再讲。

KF-6-3. 监控原理

通过查阅监控中的映射指标，可以看到当前系统中可以运行的所有请求路径，其中大部分路径以`/actuator`开头

The screenshot shows the Spring Boot Admin application page. On the left, there's a sidebar with tabs: Insights, 日志配置 (Log Configuration), JVM, 映射 (Mappings), and 缓存 (Cache). The 'Mappings' tab is selected. In the main area, there's a table listing various actuator endpoints and their details:

方法	POST
提交内容类型	application/vnd.spring-boot.actuator.v3+json, application/vnd.spring-boot.actuator.v2+json, application/json
处理程序	Actuator web endpoint 'loggers-name'
方法	GET
返回内容类型	application/vnd.spring-boot.actuator.v3+json, application/vnd.spring-boot.actuator.v2+json, application/json
处理程序	Actuator root web endpoint
方法	GET
返回内容类型	application/vnd.spring-boot.actuator.v3+json, application/vnd.spring-boot.actuator.v2+json, application/json
处理程序	Actuator web endpoint 'loggers-name'
方法	GET

首先这些请求路径不是开发者自己编写的，其次这个路径代表什么含义呢？既然这个路径可以访问，就可以通过浏览器发送该请求看看究竟可以得到什么信息。

The screenshot shows the Postman application interface. On the left, there's a sidebar with sections: Collections, APIs, Environments, Mock Servers, Monitors, and History. The 'Collections' section is expanded, showing a folder named 'springmvc' which contains several items like 'ES', 'REST', 'SpringBoot', etc. The main workspace shows a 'GET' request to 'http://localhost:81/actuator'. The 'Body' tab is selected, displaying the JSON response from the server. The response is a JSON object with '_links' and other fields.

```

{
  "_links": {
    "self": {
      "href": "http://localhost:81/actuator",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:81/actuator/beans",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:81/actuator/caches/{cache}",
      "templated": true
    },
    "caches": {
      "href": "http://localhost:81/actuator/caches",
      "templated": false
    }
  }
}

```

通过发送请求，可以得到一组json信息，如下

```
{
  "_links": {
    "self": {
      "href": "http://localhost:81/actuator",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:81/actuator/beans",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:81/actuator/caches/{cache}",
      "templated": true
    }
  }
}
```

```
},
"caches": {
    "href": "http://localhost:81/actuator/caches",
    "templated": false
},
"health": {
    "href": "http://localhost:81/actuator/health",
    "templated": false
},
"health-path": {
    "href": "http://localhost:81/actuator/health/{*path}",
    "templated": true
},
"info": {
    "href": "http://localhost:81/actuator/info",
    "templated": false
},
"conditions": {
    "href": "http://localhost:81/actuator/conditions",
    "templated": false
},
"shutdown": {
    "href": "http://localhost:81/actuator/shutdown",
    "templated": false
},
"configprops": {
    "href": "http://localhost:81/actuator/configprops",
    "templated": false
},
"configprops-prefix": {
    "href": "http://localhost:81/actuator/configprops/{prefix}",
    "templated": true
},
"env": {
    "href": "http://localhost:81/actuator/env",
    "templated": false
},
"env-toMatch": {
    "href": "http://localhost:81/actuator/env/{toMatch}",
    "templated": true
},
"loggers": {
    "href": "http://localhost:81/actuator/loggers",
    "templated": false
},
"loggers-name": {
    "href": "http://localhost:81/actuator/loggers/{name}",
    "templated": true
},
"heapdump": {
    "href": "http://localhost:81/actuator/heapdump",
    "templated": false
},
"threaddump": {
    "href": "http://localhost:81/actuator/threaddump",
```

```
        "templated": false
    },
    "metrics-requiredMetricName": {
        "href": "http://localhost:81/actuator/metrics/{requiredMetricName}",
        "templated": true
    },
    "metrics": {
        "href": "http://localhost:81/actuator/metrics",
        "templated": false
    },
    "scheduledtasks": {
        "href": "http://localhost:81/actuator/scheduledtasks",
        "templated": false
    },
    "mappings": {
        "href": "http://localhost:81/actuator/mappings",
        "templated": false
    }
}
}
```

其中每一组数据都有一个请求路径，而在这里请求路径中有之前看到过的health，发送此请求又得到了一组信息

```
{
    "status": "UP",
    "components": {
        "diskSpace": {
            "status": "UP",
            "details": {
                "total": 297042808832,
                "free": 72284409856,
                "threshold": 10485760,
                "exists": true
            }
        },
        "ping": {
            "status": "UP"
        }
    }
}
```

当前信息与监控面板中的数据存在着对应关系

The screenshot shows the Spring Boot Admin web interface. On the left, a sidebar titled 'spring-boot-application' displays various monitoring tabs like Insights, Details, Performance, Environment, Classes, Configuration, Scheduled Tasks, Log Configuration, JVM, Mappings, and Caches. The 'Details' tab is selected. In the main content area, there are three tabs at the top: 'http://CZBK-20210302VL:80/' (disabled), 'http://CZBK-20210302VL:80/actuator' (selected), and 'http://CZBK-20210302VL:80/actuator/health'. Below these tabs, there are two sections: 'Information' (显示 '未提供任何信息。') and 'Metadata' (显示 'startup 2022-03-01T11:03:08.028+08:00') in a greyed-out state. To the right, a red box highlights the 'Health' section, which contains a table with two rows:

Instance	UP
diskSpace	UP
total	297 GB
free	72.3 GB
threshold	10.5 MB
exists	true

Below this table is another row labeled 'ping' with 'UP' status.

原来监控中显示的信息实际上是通过发送请求后得到json数据，然后展示出来。按照上述操作，可以发送更多的以/actuator开头的链接地址，获取更多的数据，这些数据汇总到一起组成了监控平台显示的所有数据。

到这里我们得到了一个核心信息，监控平台中显示的信息实际上是通过对被监控的应用发送请求得到的。那这些请求谁开发的呢？打开被监控应用的pom文件，其中导入了springboot admin的对应的client，在这个资源中导入了一个名称叫做actuator的包。被监控的应用之所以可以对外提供上述请求路径，就是因为添加了这个包。

The screenshot shows a Maven dependency tree. The root node is 'springboot_26_admin_client'. It has three direct dependencies: 'Lifecycle', 'Plugins', and 'Dependencies'. The 'Dependencies' node is expanded, showing a list of dependencies. One dependency, 'de.codecentric:spring-boot-admin-starter-client:2.5.4', is highlighted with a blue bar. Underneath it, the 'org.springframework.boot:spring-boot-starter-actuator:2.5.4' dependency is also highlighted with a red box. Other dependencies listed include 'org.springframework.boot:spring-boot-starter-web:2.5.4' and 'org.springframework.boot:spring-boot-starter-test:2.5.4 (test)'.

这个actuator是什么呢？这就是本节要讲的核心内容，监控的端点。

Actuator，可以称为端点，描述了一组监控信息，SpringBootAdmin提供了多个内置端点，通过访问端点就可以获取对应的监控信息，也可以根据需要自定义端点信息。通过发送请求路劲/actuator可以访问应用所有端点信息，如果端点中还有明细信息可以发送请求/actuator/端点名称来获取详细信息。以下列出了所有端点信息说明：

ID	描述	默认启用
auditevents	暴露当前应用程序的审计事件信息。	是
beans	显示应用程序中所有 Spring bean 的完整列表。	是
caches	暴露可用的缓存。	是
conditions	显示在配置和自动配置类上评估的条件以及它们匹配或不匹配的原因。	是
configprops	显示所有 @ConfigurationProperties 的校对清单。	是
env	暴露 Spring ConfigurableEnvironment 中的属性。	是
flyway	显示已应用的 Flyway 数据库迁移。	是
health	显示应用程序健康信息	是
httptrace	显示 HTTP 追踪信息（默认情况下，最后 100 个 HTTP 请求/响应交换）。	是
info	显示应用程序信息。	是
integrationgraph	显示 Spring Integration 图。	是
loggers	显示和修改应用程序中日志记录器的配置。	是
liquibase	显示已应用的 Liquibase 数据库迁移。	是
metrics	显示当前应用程序的指标度量信息。	是
mappings	显示所有 @RequestMapping 路径的整理清单。	是
scheduledtasks	显示应用程序中的调度任务。	是
sessions	允许从 Spring Session 支持的会话存储中检索和删除用户会话。当使用 Spring Session 的响应式 Web 应用程序支持时不可用。	是
shutdown	正常关闭应用程序。	否
threaddump	执行线程 dump。	是
headdump	返回一个 hprof 堆 dump 文件。	是

ID	描述	默认启用
jolokia	通过 HTTP 暴露 JMX bean (当 Jolokia 在 classpath 上时, 不适用于 WebFlux)。	是
logfile	返回日志文件的内容 (如果已设置 logging.file 或 logging.path 属性)。支持使用 HTTP Range 头来检索部分日志文件的内容。	是
prometheus	以可以由 Prometheus 服务器抓取的格式暴露指标。	是

上述端点每一项代表被监控的指标, 如果对外开放则监控平台可以查询到对应的端点信息, 如果未开放则无法查询对应的端点信息。通过配置可以设置端点是否对外开放功能。使用enable属性控制端点是否对外开放。其中health端点为默认端点, 不能关闭。

```
management:
  endpoint:
    health: # 端点名称
      show-details: always
    info: # 端点名称
    enabled: true # 是否开放
```

为了方便开发者快速配置端点, springboot admin设置了13个较为常用的端点作为默认开放的端点, 如果需要控制默认开放的端点的开放状态, 可以通过配置设置, 如下:

```
management:
  endpoints:
    enabled-by-default: true # 是否开启默认端点, 默认值true
```

上述端点开启后, 就可以通过端点对应的路径查看对应的信息了。但是此时还不能通过HTTP请求查询此信息, 还需要开启通过HTTP请求查询的端点名称, 使用“*”可以简化配置成开放所有端点的WEB端HTTP请求权限。

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

整体上来说, 对于端点的配置有两组信息, 一组是endpoints开头的, 对所有端点进行配置, 一组是endpoint开头的, 对具体端点进行配置。

```
management:
  endpoint:      # 具体端点的配置
  health:
    show-details: always
  info:
    enabled: true
  endpoints:      # 全部端点的配置
  web:
    exposure:
      include: "*"
  enabled-by-default: true
```

总结

1. 被监控客户端通过添加actuator的坐标可以对外提供被访问的端点功能
2. 端点功能的开放与关闭可以通过配置进行控制
3. web端默认无法获取所有端点信息，通过配置开放端点功能

KF-6-4.自定义监控指标

端点描述了被监控的信息，除了系统默认的指标，还可以自行添加显示的指标，下面就通过3种不同的端点的指标自定义方式来学习端点信息的二次开发。

INFO端点

info端点描述了当前应用的基本信息，可以通过两种形式快速配置info端点的信息

- 配置形式

在yml文件中通过设置info节点的信息就可以快速配置端点信息

```
info:
  appName: @project.artifactId@
  version: @project.version@
  company: 传智教育
  author: itheima
```

配置完毕后，对应信息显示在监控平台上

The screenshot shows the Spring Boot Admin web interface. On the left, a sidebar menu includes 'Insights' (selected), '细节', '性能', '环境', '类', '配置属性', '计划任务', '日志配置', 'JVM', '映射', and '缓存'. The main content area is titled 'spring-boot-application' with ID '44f452bc95bd'. It displays three tabs: '信息' (highlighted with a red box), '健康', and '元数据'. The '信息' tab contains the following data:

appName	springboot_08_ssmp
version	0.0.1-SNAPSHOT
company	传智教育
author	itheima

The '健康' tab shows the following status:

Instance	UP
db	UP
database	MySQL
validationQuery	isValid()
diskSpace	UP
total	297 GB

也可以通过请求端点信息路径获取对应json信息

The screenshot shows a POSTMAN API client. The request method is 'GET', the URL is 'http://localhost:81/actuator/info', and the 'Send' button is highlighted. The 'Body' tab is selected, showing the JSON response:

```

1 {
2   "appName": "springboot_08_ssmp",
3   "version": "0.0.1-SNAPSHOT",
4   "company": "传智教育",
5   "author": "itheima"
6 }

```

- 编程形式

通过配置的形式只能添加固定的数据，如果需要动态数据还可以通过配置bean的方式为info端点添加信息，此信息与配置信息共存

```

@Component
public class InfoConfig implements InfoContributor {
    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("runTime", System.currentTimeMillis());           //添加单个信息
        Map infoMap = new HashMap();
        infoMap.put("buildTime", "2006");
        builder.withDetails(infoMap);                                         //添加一组信息
    }
}

```

Health端点

health端点描述当前应用的运行健康指标，即应用的运行是否成功。通过编程的形式可以扩展指标信息。

```
@Component
```

```

public class HealthConfig extends AbstractHealthIndicator {
    @Override
    protected void doHealthCheck(Health.Builder builder) throws Exception {
        boolean condition = true;
        if(condition) {
            builder.status(Status.UP); //设置运行状态为启动状态
            builder.withDetail("runTime", System.currentTimeMillis());
            Map infoMap = new HashMap();
            infoMap.put("buildTime", "2006");
            builder.withDetails(infoMap);
        }else{
            builder.status(Status.OUT_OF_SERVICE); //设置运行状态为不在服务状态
            builder.withDetail("上线了吗?", "你做梦");
        }
    }
}

```

当任意一个组件状态不为UP时，整体应用对外服务状态为非UP状态。

The screenshot shows the Spring Boot Admin web interface at localhost:8080/instances/44f452bc95bd/details. The left sidebar has a red header for the application 'spring-boot-application' and a green 'Details' tab selected. The main area is divided into two sections: 'Information' and 'Health'. The 'Information' section contains details like appName (springboot_08_ssmp), version (0.0.1-SNAPSHOT), company (传智教育), author (itheima), runTime (1646128030386), and buildTime (2006). The 'Health' section shows the overall status as 'OUT_OF_SERVICE'. It lists components: 'db' (UP, MySQL, validationQuery isValid()), 'diskSpace' (UP, total 297 GB, free 72.3 GB, threshold 10.5 MB, exists true), and 'healthConfig' (OUT_OF_SERVICE, 上线了吗? 你做梦). Other components listed as UP are 'ping' and 'redis'.

Metrics端点

metrics端点描述了性能指标，除了系统自带的监控性能指标，还可以自定义性能指标。

```

@Service
public class BookServiceImpl extends ServiceImpl<BookDao, Book> implements
IBookService {
    @Autowired
    private BookDao bookDao;

    private Counter counter;
}

```

```

public BookServiceImpl(MeterRegistry meterRegistry){
    counter = meterRegistry.counter("用户付费操作次数: ");
}

@Override
public boolean delete(Integer id) {
    //每次执行删除业务等同于执行了付费业务
    counter.increment();
    return bookDao.deleteById(id) > 0;
}
}

```

在性能指标中就出现了自定义的性能指标监控项

The screenshot shows the Spring Boot Admin web interface at `localhost:8080/instances/44f452bc95bd/metrics`. On the left, there's a sidebar with various monitoring sections like Insights, Details, and Performance. The 'Performance' section is currently selected. In the main area, there's a dropdown menu labeled '用户付费操作次数:' (User Pay Operation Count) which is expanded to show a list of metrics. The list includes standard JVM and Tomcat metrics, and at the bottom, it shows the custom metric '用户付费操作次数:' again, which is highlighted with a red box. A green button labeled '添加指标' (Add Metric) is visible to the right of the dropdown.

自定义端点

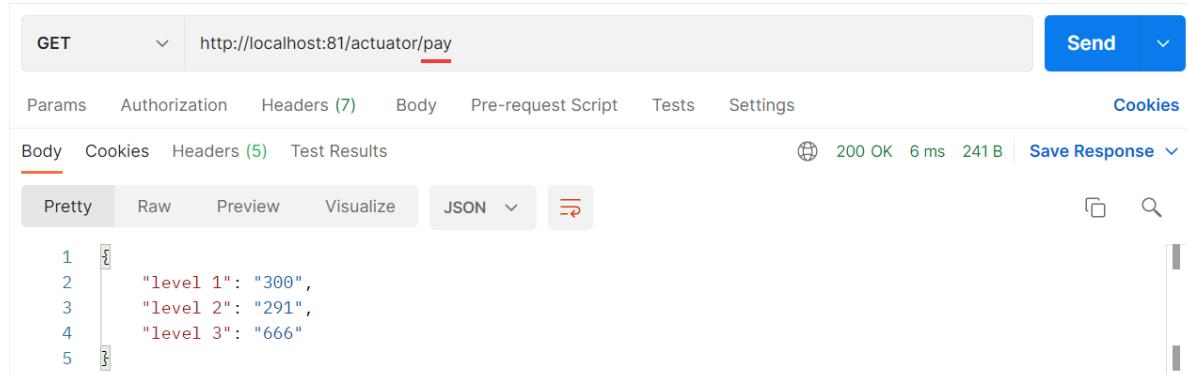
可以根据业务需要自定义端点，方便业务监控

```

@Component
@Endpoint(id="pay", enableByDefault = true)
public class PayEndpoint {
    @ReadOperation
    public Object getPay(){
        Map payMap = new HashMap();
        payMap.put("level 1", "300");
        payMap.put("level 2", "291");
        payMap.put("level 3", "666");
        return payMap;
    }
}

```

由于此端点数据spring boot admin无法预知该如何展示，所以通过界面无法看到此数据，通过HTTP请求路径可以获取到当前端点的信息，但是需要先开启当前端点对外功能，或者设置当前端点为默认开发的端点。



```
1 {  
2   "level 1": "300",  
3   "level 2": "291",  
4   "level 3": "666"  
5 }
```

总结

1. 端点的指标可以自定义，但是每种不同的指标根据其功能不同，自定义方式不同
2. info端点通过配置和编程的方式都可以添加端点指标
3. health端点通过编程的方式添加端点指标，需要注意要为对应指标添加启动状态的逻辑设定
4. metrics指标通过在业务中添加监控操作设置指标
5. 可以自定义端点添加更多的指标

开发实用篇完结

开发实用篇到这里就暂时完结了，在开发实用篇中我们讲解了大量的第三方技术的整合方案，选择的方案都是市面上比较流行的常用方案，还有一些国内流行度较低的方案目前还没讲，留到番外篇中慢慢讲吧。

整体开发实用篇中讲解的内容可以分为两大类知识：实用性知识与经验性知识。

实用性知识就是新知识了，springboot整合各种技术，每种技术整合中都有一些特殊操作，整体来说其实就是三句话。加坐标做配置调接口。经验性知识是对前面两篇中出现的一些知识的补充，在学习基础篇时如果将精力放在这些东西上就有点学偏了，容易钻牛角尖，放到实用开发篇中结合实际开发说一些不常见的但是对系统功能又危害的操作解决方案，提升理解。

开发实用篇做到这里就告一段落，下面就要着手准备原理篇了。市面上很多课程原理篇讲的过于高深莫测，在新手还没明白123的时候就开始讲微积分了，着实让人看了着急。至于原理篇我讲成什么样子？一起期待吧。

SpringBoot原理篇

在学习前面三篇的时候，好多小伙伴一直在B站评论区嚷嚷着期待原理篇，今天可以正式的宣布了，他来了他来了他脚踏祥云进来了（此处请自行脑补BGM）。

其实从本人的角度出发，看了这么多学习java的小伙伴的学习过程，个人观点，不建议小伙伴过早的去研究技术的原理。原因有二：一，**先应用熟练，培养技术应用的条件反射**，然后再学原理。大把的学习者天天还纠结于这里少写一个这，那里少写一个那，程序都跑不下去，要啥原理，要啥自行车。这里要说一句啊，懂不懂啥意思那不叫原理，原理是抽象到顶层设计层面的东西。知道为什么写这句话，知道错误的原因和懂原理是两码事。二，**原理真不是看源码**，源码只能称作原理的落地实现方式，当好

的落地实现方式出现后，就会有新旧版本的迭代，底层实现方式也会伴随着更新升级。但是原理不变，只是找到了更好的实现最初目标的路径。一个好的课程，一位好的老师，不会用若干行云里雾里的源代码把学习者带到沟里，然后爬不出来，深陷泥潭。一边沮丧的看着源码，一边舔着老师奉其为大神，这就叫不干人事。原理就应该使用最通俗易懂的语言，把设计思想讲出来，至于看源码，只是因为目前的技术原创人员只想到了当前这种最笨的设计方案，还没有更好的。比如spring程序，写起来很费劲，springboot出来以后就简单轻松了很多，实现方案变了，原理不变。但凡你想通过下面的课程学习去读懂若干行代码，然后特别装逼的告诉自己，我懂原理了。我只能告诉你，你选了一条成本最高的路线，看源码仅仅是验证原理，源码仅对应程序流程，不对应原理。原理是思想级的，不是代码级的，原理是原本的道理。

springboot技术本身就是为了加速spring程序的开发的，可以大胆的说，springboot技术没有自己的原理层面的设计，仅仅是实现方案进行了改进。将springboot定位成工具，你就不会去想方设法的学习其原理了。就像是将木头分割成若干份，我们可以用斧子，用锯子，用刀，用火烧或者一脚踹断它，这些都是方式方法，而究其本质底层原理是植物纤维的组织方式，研究完这个，你再看前述的各种工具，都是基于这个原理在说如何变更破坏这种植物纤维的方式。所以不要一张嘴说了若干种技术，然后告诉自己，这就是springboot的原理。没有的事，springboot作为一款工具，压根就没有原理。我们下面要学习的其实就是springboot程序的工作流程。

下面就开始学习原理篇，因为没有想出来特别好的名字，所以还是先称作原理篇吧。原理篇中包含如下内容：

- 自动配置工作流程
- 自定义starter开发
- springboot程序启动流程

下面开启第一部分自动配置工作流程的学习

YL-1.自动配置工作流程

自动配置是springboot技术非常好用的核心因素，前面学习了这么多种技术的整合，每一个都离不开自动配置。不过在学习自动配置的时候，需要你对spring容器如何进行bean管理的过程非常熟悉才行，所以这里需要先复习一下有关spring技术中bean加载相关的知识。方式方法很多，逐一快速复习一下，查漏补缺。不过这里需要声明一点，这里列出的bean的加载方式仅仅应用于后面课程的学习，并不是所有的spring加载bean的方式。跟着我的步伐一种一种的复习，他们这些方案之间有千丝万缕的关系，顺着看完，你就懂自动配置是怎么回事了。

YL-1-1.bean的加载方式

关于bean的加载方式，spring提供了各种各样的形式。因为spring管理bean整体上来说就是由spring维护对象的生命周期，所以bean的加载可以从大的方面划分成2种形式。已知类并交给spring管理，和已知类名并交给spring管理。有什么区别？一个给.class，一个给类名字符串。内部其实都一样，都是通过spring的BeanDefinition对象初始化spring的bean。如果前面这句话看起来有障碍，可以去复习一下spring的相关知识。B站中有我尊敬的满一航老师录制的spring高级课程，链接地址如下，欢迎大家捧场，记得一键三连哦。

<https://www.bilibili.com/video/BV1P44y1N7QG>

方式一：配置文件+<bean/>标签

最高端的食材往往只需要最简单的烹饪方法，搞错了，再来。最初级的bean的加载方式其实可以直击spring管控bean的核心思想，就是提供类名，然后spring就可以管理了。所以第一种方式就是给出bean的类名，至于内部嘛就是反射机制加载成class，然后，就没有然后了，拿到了class你就可以搞定一切了。如果这句话听不太懂，请这些小盆友转战java基础高级部分复习一下反射相关知识。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--xml方式声明自己开发的bean-->
    <bean id="cat" class="Cat"/>
    <bean class="Dog"/>

    <!--xml方式声明第三方开发的bean-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"/>
    <bean class="com.alibaba.druid.pool.DruidDataSource"/>
    <bean class="com.alibaba.druid.pool.DruidDataSource"/>
</beans>
```

方式二：配置文件扫描+注解定义bean

由于方式一种需要将spring管控的bean全部写在xml文件中，对于程序员来说非常不友好，所以就有了第二种方式。哪一个类要受到spring管控加载成bean，就在这个类的上面加一个注解，还可以顺带起一个bean的名字（id）。这里可以使用的注解有@Component以及三个衍生注解@Service、@Controller、@Repository。

```
@Component("tom")//tom是id
public class Cat {
```

```
@Service
public class Mouse {
```

当然，由于我们无法在第三方提供的技术源代码中去添加上述4个注解，因此当你需要加载第三方开发的bean的时候可以使用下列方式定义注解式的bean。`@Bean`定义在一个方法上方，当前方法的返回值就可以交给spring管控，记得这个方法所在的类一定要定义在@Component修饰的类中，有人会说不是@Configuration吗？建议把spring注解开发相关课程学习一下，就不会有这个疑问了。

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

上面提供的仅仅是bean的声明，spring并没有感知到这些东西，像极了上课积极回答问题的你，手举的非常高，可惜老师都没有往你的方向看上一眼。想让spring感知到这些积极的小伙伴，必须设置spring去检查这些类，看他们是否贴标签，想当积极分子。可以通过下列xml配置设置spring去检查哪些包，发现定了对应注解，就将对应的类纳入spring管控范围，声明成bean。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd
       ">
    <!--指定扫描加载bean的位置-->
    <context:component-scan base-package="com.itheima.bean,com.itheima.config"/>
</beans>
```

方式二声明bean的方式是目前企业中较为常见的bean的声明方式，但是也有缺点。方式一中，通过一个配置文件，你可以查阅当前spring环境中定义了多少个或者说多少种bean，但是方式二没有任何一个地方可以查阅整体信息，只有当程序运行起来才能感知到加载了多少个bean。

方式三：注解方式声明配置类

方式二已经完美的简化了bean的声明，以后再也不用写茫茫多的配置信息了。仔细观察xml配置文件，会发现这个文件中只剩了扫描包这句话，于是就有人提出，使用java类替换掉这种固定格式的配置，所以下面这种格式就出现了。严格意义上讲不能算全新的方式，但是由于此种开发形式是企业级开发中的主流形式，所以单独独立出来做成一种方式。嗯……，怎么说呢？方式二和方式三其实差别还是挺大的，番外篇找个时间再聊吧。

定义一个类并使用@ComponentScan替代原始xml配置中的包扫描这个动作，其实功能基本相同。为什么说基本，还是有差别的。先卖个关子吧，番外篇再聊。

```
@ComponentScan({"com.itheima.bean", "com.itheima.config"})
public class SpringConfig3 {
    @Bean
    public DogFactoryBean dog(){
        return new DogFactoryBean();
    }
}
```

使用FactoryBean接口

补充一个小知识，spring提供了一个接口FactoryBean，也可以用于声明bean，只不过实现了FactoryBean接口的类造出来的对象不是当前类的对象，而是FactoryBean接口泛型指定类型的对象。如下列，造出来的bean并不是DogFactoryBean，而是Dog。有什么用呢？可以在对象初始化前做一些事情，下例中的注释位置就是让你自己去扩展要做的其他事情的。

```
public class DogFactoryBean implements FactoryBean<Dog> {
    @Override
    public Dog getObject() throws Exception {
        Dog d = new Dog();
    }
}
```

```

//.....
    return d;
}
@Override
public Class<?> getObjectType() {
    return Dog.class;
}
@Override
public boolean isSingleton() {
    return true;
}
}

```

有人说，注释中的代码写入Dog的构造方法不就行了吗？干嘛这么费劲转一圈，还写个类，还要实现接口，多麻烦啊。还真不一样，你可以理解为Dog是一个抽象后剥离的特别干净的模型，但是**实际使用的时候必须进行一系列的初始化（需要进行设置xxx）动作**。只不过根据情况不同，初始化动作不同而已。如果写入Dog，或许初始化动作A当前并不能满足你的需要，这个时候你就要做一个DogB的方案了。然后，就没有然后了，你就要做两个Dog类。当时使用FactoryBean接口就可以完美解决这个问题。

通常实现了FactoryBean接口的类使用@Bean的形式进行加载，当然你也可以使用@Component去声明DogFactoryBean，只要被扫描加载到即可，但是这种格式加载总觉得怪怪的，指向性不是很明确。

```

@ComponentScan({"com.itheima.bean", "com.itheima.config"})
public class SpringConfig3 {
    @Bean
    public DogFactoryBean dog(){
        return new DogFactoryBean();
    }
}

```

注解格式导入XML格式配置的bean

再补充一个小知识，由于早起开发的系统大部分都是采用xml的形式配置bean，现在的企业级开发基本上不用这种模式了。但是如果你特别幸运，需要基于之前的系统进行二次开发，这就尴尬了。新开发的用注解格式，之前开发的是xml格式。这个时候可不是让你选择用哪种模式的，而是两种要同时使用。spring提供了一个注解可以解决这个问题，@ImportResource，在配置类上直接写上要被融合的xml配置文件名即可，算的上一种兼容性解决方案，没啥实际意义。

```

@Configuration
@ImportResource("applicationContext1.xml")
public class SpringConfig32 {
}

```

proxyBeanMethods属性

前面的例子中用到了@Configuration这个注解，当我们使用AnnotationConfigApplicationContext加载配置类的时候，配置类可以不添加这个注解。但是这个注解有一个更加强大的功能，它可以保障配置类中使用方法创建的bean的唯一性。为@Configuration注解设置proxyBeanMethods属性值为true即可，由于此属性默认值为true，所以很少看见明确书写的，除非想放弃此功能。

```
@Configuration(proxyBeanMethods = true)
public class SpringConfig33 {
    @Bean
    public Cat cat(){
        return new Cat();
    }
}
```

下面通过容器再调用上面的cat方法时，得到的就是同一个对象了。注意，必须使用spring容器对象调用此方法才有保持bean唯一性的特性。此特性在很多底层源码中有应用，前面讲MQ时，也应用了此特性，只不过当前没有解释而已。这里算是填个坑吧。

```
public class App33 {
    public static void main(String[] args) {
        ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig33.class);
        String[] names = ctx.getBeanDefinitionNames();
        for (String name : names) {
            System.out.println(name);
        }
        System.out.println("-----");
        SpringConfig33 springConfig33 = ctx.getBean("springConfig33",
SpringConfig33.class);
        System.out.println(springConfig33.cat());
        System.out.println(springConfig33.cat());
        System.out.println(springConfig33.cat());
    }
}
```

方式四：使用@Import注解注入bean

使用扫描的方式加载bean是企业级开发中常见的bean的加载方式，但是由于扫描的时候不仅可以加载到你要的东西，还有可能加载到各种各样的乱七八糟的东西，万一没有控制好得不偿失了。

有人就会奇怪，会有什么问题呢？比如你扫描了com.itheima.service包，后来因为业务需要，又扫描了com.itheima.dao包，你发现com.itheima包下面只有service和dao这两个包，这就简单了，直接扫描com.itheima就行了。但是万万没想到，十天后你加入了一个外部依赖包，里面也有com.itheima包，这下就热闹了，该来的不该来的全来了。

所以我们需要一种精准制导的加载方式，使用@Import注解就可以解决你的问题。它可以加载所有的一切，只需要在注解的参数中写上加载的类对应的.class即可。有人就会觉得，还要自己手写，多麻烦，不如扫描好用。对呀，但是他可以指定加载啊，好的命名规范配合@ComponentScan可以解决很多问题，但是@Import注解拥有其重要的应用场景。有没有想过假如你要加载的bean没有使用@Component修饰呢？这下就无解了，而@Import就无需考虑这个问题。

```
@Import({Dog.class,DbConfig.class})
public class SpringConfig4 {
```

使用@Import注解注入配置类

除了加载bean，还可以使用@Import注解加载配置类。配置类里面的@Bean注解定义的bean也会被注册到spring容器中。其实本质上是一样的，不解释太多了。

```
@Import(DbConfig.class)
public class SpringConfig4 {  
}
```

方式五：编程形式注册bean

前面介绍的加载bean的方式都是在容器启动阶段完成bean的加载，下面这种方式就比较特殊了，可以在容器初始化完成后手动加载bean。通过这种方式可以实现编程式控制bean的加载。

```
public class App5 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
        //上下文容器对象已经初始化完毕后，手工加载bean
        ctx.register(Mouse.class);
    }
}
```

其实这种方式坑还是挺多的，比如容器中已经有了某种类型的bean，再加载会不会（会覆盖，只会留下最后一个）覆盖呢？这都是要思考和关注的问题。新手慎用。

```
public class App5 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
        //上下文容器对象已经初始化完毕后，手工加载bean
        ctx.registerBean("tom", Cat.class, 0);
        ctx.registerBean("tom", Cat.class, 1);
        ctx.registerBean("tom", Cat.class, 2);
        System.out.println(ctx.getBean(Cat.class));
    }
}
```

方式六：导入实现了ImportSelector接口的类

在方式五种，我们感受了bean的加载可以进行编程化的控制，添加if语句就可以实现bean的加载控制了。但是毕竟是在容器初始化后实现bean的加载控制，那是否可以在容器初始化过程中进行控制呢？答案是必须的。实现ImportSelector接口的类可以设置加载的bean的全路径类名，记得一点，只要能编程就能判定，能判定意味着可以控制程序的运行走向，进而控制一切。

现在又多了一种控制bean加载的方式，或者说是选择bean的方式。

```

public class MyImportSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata metadata) {
        //各种条件的判定，判定完毕后，决定是否装载指定的bean
        boolean flag =
metadata.hasAnnotation("org.springframework.context.annotation.Configuration");
        if(flag){
            return new String[]{"com.itheima.bean.Dog"};
        }
        return new String[]{"com.itheima.bean.Cat"};
    }
}

```

方式七：导入实现了ImportBeanDefinitionRegistrar接口的类

方式六中提供了给定类全路径类名控制bean加载的形式，如果对spring的bean的加载原理比较熟悉的小伙伴知道，其实bean的加载不是一个简简单单的对象，spring中定义了一个叫做BeanDefinition的东西，它才是控制bean初始化加载的核心。BeanDefinition接口中给出了若干种方法，可以控制bean的相关属性。说个最简单的，创建的对象是单例还是非单例，在BeanDefinition中定义了scope属性就可以控制这个。如果你感觉方式六没有给你开放出足够的对bean的控制操作，那么方式七你值得拥有。我们可以通过定义一个类，然后实现ImportBeanDefinitionRegistrar接口的方式定义bean，并且还可以让你对bean的初始化进行更加细粒度的控制，不过对于新手并不是很友好。忽然给你开放了若干个操作，还真不知道如何下手。

```

public class MyRegistrar implements ImportBeanDefinitionRegistrar {
    //这里面也能取到元数据（AnnotationMetadata）
    //也可以根据注册器注册beanDefinition
    @Override
    public void registerBeanDefinitions(AnnotationMetadata metadata,
BeanDefinitionRegistry registry) {
        BeanDefinition beanDefinition =
BeanDefinitionBuilder.rootBeanDefinition(BookServiceImp12.class).getBeanDefinition();
        //这里可以做beanDefinition的设定
        beanDefinition.setScope();
        //第一个参数就是id
        registry.registerBeanDefinition("bookServiceid",beanDefinition);
    }
}

```

方式八：导入实现了BeanDefinitionRegistryPostProcessor接口的类

上述七种方式都是在容器初始化过程中进行bean的加载或者声明，但是这里有一个bug。这么多种方式，它们之间如果有冲突怎么办？谁能有最终裁定权？这是个好问题，当某种类型的bean被接二连三的使用各种方式加载后，在你对所有加载方式的加载顺序没有完全理解清晰之前，你还真不知道最后谁说了算。即便你理清楚了，保不齐和你一起开发的猪队友又添加了一个bean，得嘞，这下就热闹了。

spring挥舞它仲裁者的大刀来了一个致命一击，都别哔哔了，我说了算，
BeanDefinitionRegistryPostProcessor，看名字知道，BeanDefinition意思是bean定义，Registry注册的意思，Post后置，Processor处理器，全称bean定义后处理器，干啥的？在所有bean注册都折腾完后，它把最后一道关，说白了，它说了算，这下消停了，它是最后一个运行的。

```

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

```

总体上来说，上面介绍了各种各样的bean的注册加载初始化方式，脑子里建立个概念吧，方式很多，spring源码中大量运用各种方式。复习的内容就先说到这里。

总结

1. bean的定义由前期xml配置逐步演化成注解配置，本质是一样的，都是通过反射机制加载类名后创建对象，对象就是spring管控的bean
2. @Import注解可以指定加载某一个类作为spring管控的bean，如果被加载的类中还具有@Bean相关的定义，会被一同加载
3. spring开放出了若干种可编程控制的bean的初始化方式，通过分支语句由固定的加载bean转成了可以选择bean是否加载或者选择加载哪一种bean

YL-1-2.bean的加载控制

我想向大家说明一下最近的考核情况。我们对实验室成员进行了考核，有些人的表现让学长学姐们很惊讶，但是有些人未能通过考核。我们认为他们在某些方面需要进一步提高，以更好地适应未来的工作，目前已经跟有关人员打了电话。我们希望每个人都能够理解和支持这个决定。谢谢大家的合作和理解。

前面复习bean的加载时，提出了有关加载控制的方式，其中手工注册bean，ImportSelector接口，ImportBeanDefinitionRegistrar接口，BeanDefinitionRegistryPostProcessor接口都可以控制bean的加载，这一节就来说说这些加载控制。

企业级开发中不可能在spring容器中进行bean的饱和式加载的。什么是饱和式加载，就是不管用不用，全部加载。比如jdk中有两万个类，那就加载两万个bean，显然是不合理的，因为你压根就不会使用其中大部分的bean。那合理的加载方式是什么？肯定是必要性加载，就是用什么加载什么。继续思考，加载哪些bean通常受什么影响呢？最容易想的就是你要用什么技术，就加载对应的bean。用什么技术意味着什么？就是加载对应技术的类。所以在spring容器中，通过判定是否加载了某个类来控制某些bean的加载是一种常见操作。下例给出了对应的代码实现，其实思想很简单，先判断一个类的全路径名是否能够成功加载，加载成功说明有这个类，那就干某项具体的工作，否则就干别的工作。

```

public class MyImportSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        try {
            Class<?> clazz = Class.forName("com.itheima.bean.Mouse");
            if(clazz != null) {

```

```

        return new String[]{"com.itheima.bean.Cat"};
    }
} catch (ClassNotFoundException e) {
//      e.printStackTrace();
    return new String[0];
}
return null;
}
}

```

通过上述的分析，可以看到此类操作将成为企业级开发中的常见操作，于是springboot将把这些常用操作给我们做了一次封装。这种逻辑判定你开发者就别搞了，我springboot信不过你这种新手开发者，我给你封装一下，做几个注解，你填参数吧，耶，happy。

下例使用@ConditionalOnClass注解实现了当虚拟机中加载了com.itheima.bean.Wolf类时加载对应的bean。比较一下上面的代码和下面的代码，有没有感觉很清爽。其实此类注解还有很多。

```

@Bean
@ConditionalOnClass(name = "com.itheima.bean.Wolf")
public Cat tom(){
    return new Cat();
}

```

@ConditionalOnMissingClass注解控制虚拟机中没有加载指定的类才加载对应的bean。

```

@Bean
@ConditionalOnMissingClass("com.itheima.bean.Dog")
public Cat tom(){
    return new Cat();
}

```

这种条件还可以做并且的逻辑关系，写2个就是2个条件都成立，写多个就是多个条件都成立。

```

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

```

除了判定是否加载类，还可以对当前容器类型做判定，下例是判定当前容器环境是否是web环境。

```

@Bean
@ConditionalOnWebApplication
public Cat tom(){
    return new Cat();
}

```

下面是判定容器环境是否是非web环境。

```
@Bean  
@ConditionalOnNotWebApplication  
public Cat tom(){  
    return new Cat();  
}
```

当然还可以判定是否加载了指定名称的bean，这种有什么用呢？太有用了。比如当前容器中已经提供了JdbcTemplate对应的bean，你还需要再加载一个全新的JdbcTemplate的bean吗？没有必要了嘛。Spring说，如果你自己写的话，我就不帮你操这份心了，如果你没写，我再给你提供。自适应，自适应，明白？没有的话就提供给你，有的话就用你自己的，是不是很帅？

```
@Bean  
@ConditionalOnBean(name="jerry")  
public Cat tom(){  
    return new Cat();  
}
```

以下就是判定当前是否加载了mysql的驱动类，如果加载了，我就给你搞一个Druid的数据源对象出来，完美！

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

其中SpringBoot的bean加载控制注解还有很多，这里就不一一列举了，最常用的判定条件就是根据类是否加载来进行控制。

总结

1. SpringBoot定义了若干种控制bean加载的条件设置注解，由Spring固定加载bean变成了可以根据情况选择性的加载bean

YL-1-3.bean的依赖属性配置管理

bean的加载及加载控制已经搞完了，下面研究一下bean内部的事情。bean在运行的时候，实现对应该的业务逻辑时有可能需要开发者提供一些设置值，那就是属性了。如果使用构造方法将参数固定，灵活性不足，这个时候就可以使用前期学习的bean的属性配置相关的知识进行灵活的配置了。先通过yml配置文件，设置bean运行需要使用的配置信息。

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

然后定义一个封装属性的专用类，加载配置属性，读取对应前缀相关的属性值。

```
@ConfigurationProperties(prefix = "cartoon")
@Data
public class CartoonProperties {
    private Cat cat;
    private Mouse mouse;
}
```

最后在使用的位置注入对应的配置即可。

```
@EnableConfigurationProperties(CartoonProperties.class)
public class CartoonCatAndMouse{
    @Autowired
    private CartoonProperties cartoonProperties;
}
```

建议在业务类上使用@EnableConfigurationProperties声明bean，这样在不使用这个类的时候，也不会无故加载专用的属性配置类CartoonProperties，减少spring管控的资源数量。

总结

1. bean的运行如果需要外部设置值，建议将设置值封装成专用的属性类*** Properties
2. 设置属性类加载指定前缀的配置信息
3. 在需要使用属性类的位置通过注解@EnableConfigurationProperties加载bean，而不要直接在属性配置类上定义bean，减少资源加载的数量，因需加载而不要饱和式加载。

YL-1-4. 自动配置原理（工作流程）

经过前面的知识复习，下面终于进入到了本章核心内容的学习，自动配置原理。原理谈不上，就是自动配置的工作流程。

啥叫自动配置呢？简单说就是springboot根据我们开发者的行为猜测你要做什么事情，然后把你要用的bean都给你准备好。听上去是不是很神奇？其实非常简单，前面复习的东西都已经讲完了。

springboot咋做到的呢？就是看你导入了什么类，就知道你想干什么了。然后把你有可能要用的bean（注意是有可能）都给你加载好，你直接使用就行了，springboot把所需要的一切工作都做完了。

自动配置的意义就是加速开发效率，将开发者使用某种技术时需要使用的bean根据情况提前加载好，实现自动配置的效果。当然，开发者有可能需要提供必要的参数，比如你要用mysql技术，导入了mysql的坐标，springboot就知道了你要做数据库操作，一系列的数据库操作相关的bean都给你提前声明好，但是你要告诉springboot你到底用哪一个数据库，像什么IP地址啊，端口啊，你不告诉springboot，springboot就无法帮你把自动配置相关的工作做完。

而这种思想其实就是在日常的开发过程中根据开发者的习惯慢慢抽取得到了。整体过程分为2个阶段：

阶段一：准备阶段

1. springboot的开发人员先大量收集Spring开发者的编程习惯，整理开发过程每一个程序经常使用的技术列表，形成一个**技术集A**
2. 收集常用技术(**技术集A**)的使用参数，不管你用什么常用设置，我用什么常用设置，统统收集起来整理一下，得到开发过程中每一个技术的常用设置，形成每一个技术对应的**设置集B**

阶段二：加载阶段

3. springboot初始化Spring容器基础环境，读取用户的配置信息，加载用户自定义的bean和导入的其他坐标，形成**初始化环境**
4. springboot将**技术集A**包含的所有技术在SpringBoot启动时默认全部加载，这时肯定加载的东西有一些是无效的，没有用的
5. springboot会对**技术集A**中每一个技术约定出启动这个技术对应的条件，并设置成按条件加载，由于开发者导入了一些bean和其他坐标，也就是与**初始化环境**，这个时候就可以根据这个**初始化环境**与springboot的**技术集A**进行比对了，哪个匹配上加载哪个
6. 因为有些技术不做配置就无法工作，所以springboot开始对**设置集B**下手了。它统计出各个国家各个行业的开发者使用某个技术时最常用的设置是什么，然后把这些设置作为默认值直接设置好，并告诉开发者当前设置我已经给你搞了一套，你要用可以直接用，这样可以减少开发者配置参数的工作量
7. 但是默认配置不一定能解决问题，于是springboot开放修改**设置集B**的接口，可以由开发者根据需要决定是否覆盖默认配置

以上这些仅仅是一个思想，落地到代码实现阶段就要好好思考一下怎么实现了。假定我们想自己实现自动配置的功能，都要做哪些工作呢？

- 首先指定一个技术X，我们打算让技术X具备自动配置的功能，这个技术X可以是任意功能，这个技术隶属于上面描述的**技术集A**

```
public class CartoonCatAndMouse{  
}
```

- 然后找出技术X使用过程中的常用配置Y，这个配置隶属于上面表述的**设置集B**

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

- 将常用配置Y设计出对应的yml配置书写格式，然后定义一个属性类封装对应的配置属性，这个过程其实就是上一节咱们做的bean的依赖属性管理，一模一样

```
@ConfigurationProperties(prefix = "cartoon")  
@Data  
public class CartoonProperties {  
    private Cat cat;  
    private Mouse mouse;  
}
```

- 最后做一个配置类，当这个类加载的时候就可以初始化对应的功能bean，并且可以加载到对应的配置

```
@EnableConfigurationProperties(CartoonProperties.class)
public class CartoonCatAndMouse implements ApplicationContextAware {
    private CartoonProperties cartoonProperties;
}
```

- 当然，你也可以为当前自动配置类设置上激活条件，例如使用@ConditionOn***为其设置加载条件

```
@ConditionalOnClass(name="org.springframework.data.redis.core.RedisOperations")
@EnableConfigurationProperties(CartoonProperties.class)
public class CartoonCatAndMouse implements ApplicationContextAware {
    private CartoonProperties cartoonProperties;
}
```

做到这里都已经做完了，但是遇到了一个全新的问题，如何让springboot启动的时候去加载这个类呢？如果不加载的话，我们做的条件判定，做的属性加载这些全部都失效了。springboot为我们开放了一个配置入口，在配置目录中创建META-INF目录，并创建spring.factories文件，在其中添加设置，说明哪些类要启动自动配置就可以了。

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

其实这个文件就做了一件事，通过这种配置的方式加载了指定的类。转了一圈，就是个普通的bean的加载，和最初使用xml格式加载bean几乎没有区别，格式变了而已。那自动配置的核心究竟是什么呢？自动配置其实是一个小的生态，可以按照如下思想理解：

1. 自动配置从根本上来说就是一个bean的加载
2. 通过bean加载条件的控制给开发者一种感觉，自动配置是自适应的，可以根据情况自己判定，但实际上就是最普通的分支语句的应用，这是蒙蔽我们双眼的第一层面纱
3. 使用bean的时候，如果不设置属性，就有默认值，如果不想用默认值，就可以自己设置，也就是可以修改部分或者全部参数，感觉这个过程好屌，也是一种自适应的形式，其实还是需要使用分支语句来做判断的，这是蒙蔽我们双眼的第二层面纱
4. springboot技术提前将大量开发者有可能使用的技术提前做好了，条件也写好了，用的时候你导入了一个坐标，对应技术就可以使用了，其实就是提前帮我们把spring.factories文件写好了，这是蒙蔽我们双眼的第三层面纱

你在不知道自动配置这个知识的情况下，经过上面这一二三，你当然觉得自动配置是一种特别牛的技术，但是一窥究竟后发现，也就那么回事。而且现在springboot程序启动时，在后台偷偷的做了这么多次检测，这么多种情况判定，不用问了，效率一定是非常低的，毕竟它要检测100余种技术是否在你程序中使用。

以上内容是自动配置的工作流程。

总结

1. springboot启动时先加载spring.factories文件中的org.springframework.boot.autoconfigure.EnableAutoConfiguration配置项，将其中配置的所有类都加载成bean
2. 在加载bean的时候，bean对应的类定义上都设置有加载条件，因此有可能加载成功，也可能条件检测失败不加载bean
3. 对于可以正常加载成bean的类，通常会通过@EnableConfigurationProperties注解初始化对应的配置属性类并加载对应的配置

4. 配置属性类上通常会通过@ConfigurationProperties加载指定前缀的配置，当然这些配置通常都有默认值。如果没有默认值，就强制你必须配置后使用了

1. 先开发若干种技术的标准实现
2. SpringBoot启动时加载所有的技术实现对应的自动配置类
3. 检测每个配置类的加载条件是否满足并进行对应的初始化
4. 切记是先加载所有的外部资源，然后根据外部资源进行条件比对

YL-1-5. 变更自动配置

知道了自动配置的执行过程，下面就可以根据这个自动配置的流程做一些高级定制了。例如系统默认会加载100多种自动配置的技术，如果我们先手工干预此工程，禁用自动配置是否可行呢？答案一定是可以的。方式还挺多：

方式一：通过yaml配置设置排除指定的自动配置类

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

方式二：通过注解参数排除自动配置类

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

方式三：排除坐标（应用面较窄）

如果当前自动配置中包含有更多的自动配置功能，也就是一个套娃的效果。此时可以通过检测条件的控制来管理自动配置是否启动。例如web程序启动时会自动启动tomcat服务器，可以通过排除坐标的方式，让加载tomcat服务器的条件失效。不过需要提醒一点，你把tomcat排除掉，记得再加一种可以运行的服务器。

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

```
</dependency>  
</dependencies>
```

总结

1. springboot的自动配置并不是必然运行的，可以通过配置的形式干预是否启用对应的自动配置功能

YL-2.自定义starter开发

自动配置学习完后，我们就可以基于自动配置的特性，开发springboot技术中最引以为傲的功能了，starter。其实通过前期学习，我们发现用什么技术直接导入对应的starter，然后就实现了springboot整合对应技术，再加上一些简单的配置，就可以直接使用了。这种设计方式对开发者非常友好，本章就通过一个案例的制作，开发自定义starter来实现自定义功能的快捷添加。

YL-2-1.案例：记录系统访客独立IP访问次数

本案例的功能是统计网站独立IP访问次数的功能，并将访问信息在后台持续输出。整体功能是在后台每10秒输出一次监控信息（格式：IP+访问次数），当用户访问网站时，对用户的访问行为进行统计。

例如：张三访问网站功能15次，IP地址：192.168.0.135，李四访问网站功能20次，IP地址：61.129.65.248。那么在网站后台就输出如下监控信息，此信息每10秒刷新一次。

```
IP访问监控  
+----ip-address---+--num--+  
| 192.168.0.135 | 15 |  
| 61.129.65.248 | 20 |  
+-----+-----+
```

在进行具体制作之前，先对功能做具体的分析

1. 数据记录在什么位置

最终记录的数据是一个字符串（IP地址）对应一个数字（访问次数），此处可以选择的数据存储模型可以使用java提供的map模型，也就是key-value的键值对模型，或者具有key-value键值对模型的存储技术，例如redis技术。本案例使用map作为实现方案，有兴趣的小伙伴可以使用redis作为解决方案。

1. 数据记录在什么位置
2. 统计功能运行位置，因为每次web请求都需要进行统计，因此使用拦截器会是比较好的方案，本案例使用拦截器来实现。不过在制作初期，先使用调用的形式进行测试，等功能完成了，再改成拦截器的实现方案。
3. 为了提升统计数据展示的灵活度，为统计功能添加配置项。输出频度，输出的数据格式，统计数据的显示模式均可以通过配置实现调整。
 - 输出频度，默认10秒
 - 数据特征：累计数据 / 阶段数据，默认累计数据
 - 输出格式：详细模式 / 极简模式

在下面的制作中，分成若干个步骤实现。先完成最基本的统计功能的制作，然后开发出统计报表，接下来把所有的配置都设置好，最后将拦截器功能实现，整体功能就做完了。

YL-2-2.IP计数业务功能开发（自定义starter）

本功能最终要实现的效果是在现有的项目中导入一个starter，对应的功能就添加上了，删除掉对应的starter，功能就消失了，要求功能要与原始项目完全解耦。因此需要开发一个独立的模块，制作对应功能。

步骤一：创建全新的模块，定义业务功能类

功能类的制作并不复杂，定义一个业务类，声明一个Map对象，用于记录ip访问次数，key是ip地址，value是访问次数

```
public class IpCountService {  
    private Map<String, Integer> ipCountMap = new HashMap<String, Integer>();  
}
```

有些小伙伴可能会有疑问，不设置成静态的，如何在每次请求时进行数据共享呢？记得，当前类加载成bean以后是一个单例对象，对象都是单例的，哪里存在多个对象共享变量的问题。

步骤二：制作统计功能

制作统计操作对应的方法，每次访问后对应ip的记录次数+1。需要分情况处理，如果当前没有对应ip的数据，新增一条数据，否则就修改对应key的值+1即可

```
public class IpCountService {  
    private Map<String, Integer> ipCountMap = new HashMap<String, Integer>();  
    public void count(){  
        //每次调用当前操作，就记录当前访问的IP，然后累加访问次数  
        //1. 获取当前操作的IP地址  
        String ip = null;  
        //2. 根据IP地址从Map取值，并递增  
        Integer count = ipCountMap.get(ip);  
        if(count == null){  
            ipCountMap.put(ip, 1);  
        }else{  
            ipCountMap.put(ip, count + 1);  
        }  
    }  
}
```

因为当前功能最终导入到其他项目中进行，而导入当前功能的项目是一个web项目，可以从容器中直接获取请求对象，因此获取IP地址的操作可以通过自动装配得到请求对象，然后获取对应的访问IP地址。

```
public class IpCountService {  
    private Map<String, Integer> ipCountMap = new HashMap<String, Integer>();  
    @Autowired  
    //当前的request对象的注入工作由使用当前starter的工程提供自动装配  
    private HttpServletRequest httpServletRequest;  
    public void count(){  
        //每次调用当前操作，就记录当前访问的IP，然后累加访问次数  
        //1. 获取当前操作的IP地址  
        String ip = httpServletRequest.getRemoteAddr();  
        //2. 根据IP地址从Map取值，并递增  
        Integer count = ipCountMap.get(ip);  
    }  
}
```

```
        if(count == null){
            ipCountMap.put(ip,1);
        }else{
            ipCountMap.put(ip,count + 1);
        }
    }
}
```

步骤三：定义自动配置类

我们需要做到的效果是导入当前模块即开启此功能，因此使用自动配置实现功能的自动装载，需要开发自动配置类在启动项目时加载当前功能。

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

自动配置类需要在spring.factories文件中做配置方可自动运行。

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=cn.itcast.autoconfig.IpAutoConfiguration
```

步骤四：在原始项目中模拟调用，测试功能

原始调用项目中导入当前开发的starter

```
<dependency>
    <groupId>cn.itcast</groupId>
    <artifactId>ip_spring_boot_starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

推荐选择调用方便的功能做测试，推荐使用分页操作，当然也可以换其他功能位置进行测试。

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IpCountService ipCountService;
    @GetMapping("{currentPage}/{pageSize}")
    public R getPage(@PathVariable int currentPage,@PathVariable int
    pageSize,Book book){
        ipCountService.count();
        IPage<Book> page = bookService.getPage(currentPage, pagesize,book);
        if( currentPage > page.getPages()){
            page = bookService.getPage((int)page.getPages(), pageSize,book);
        }
        return new R(true, page);
    }
}
```

温馨提示

由于当前制作的功能需要在对应的调用位置进行坐标导入，因此必须保障仓库中具有当前开发的功能，所以每次原始代码修改后，需要重新编译并安装到仓库中。为防止问题出现，建议每次安装之前先clean然后install，保障资源进行了更新。切记切记！！

当前效果

每次调用分页操作后，可以在控制台输出当前访问的IP地址，此功能可以在count操作中添加日志或者输出语句进行测试。

YL-2-3.定时任务报表开发

当前已经实现了在业务功能类中记录访问数据，但是还没有输出监控的信息到控制台。由于监控信息需要每10秒输出1次，因此需要使用定时器功能。可以选取第三方技术Quartz实现，也可以选择Spring内置的task来完成此功能，此处选用Spring的task作为实现方案。

步骤一：开启定时任务功能

定时任务功能开启需要在当前功能的总配置中设置，结合现有业务设定，比较合理的位置是设置在自动配置类上。加载自动配置类即启用定时任务功能。

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

步骤二：制作显示统计数据功能

定义显示统计功能的操作print()，并设置定时任务，当前设置每5秒运行一次统计数据。

```
public class IpCountService {  
    private Map<String, Integer> ipCountMap = new HashMap<String, Integer>();  
    @Scheduled(cron = "0/5 * * * * ?")  
    public void print(){  
        System.out.println("          IP访问监控");  
        System.out.println("-----ip-address-----+num--+");  
        for (Map.Entry<String, Integer> entry : ipCountMap.entrySet()) {  
            String key = entry.getKey();  
            Integer value = entry.getValue();  
            System.out.println(String.format("|%18s |%5d |",key,value));  
        }  
        System.out.println("-----+-----+-----+");  
    }  
}
```

其中关于统计报表的显示信息拼接可以使用各种形式进行，此处使用String类中的格式化字符串操作进行，学习者可以根据自己的喜好调整实现方案。

温馨提示

每次运行效果之前先clean然后install，切记切记！！

当前效果

每次调用分页操作后，可以在控制台看到统计数据，至此基础功能已经开发完毕。

YL-2-4. 使用属性配置设置功能参数

由于当前报表显示的信息格式固定，为提高报表信息显示的灵活性，需要通过yml文件设置参数，控制报表的显示格式。

步骤一：定义参数格式

设置3个属性，分别用来控制显示周期（cycle），阶段数据是否清空（cycleReset），数据显示格式（model）

```
tools:  
  ip:  
    cycle: 10  
    cycleReset: false  
    model: "detail"
```

步骤二：定义封装参数的属性类，读取配置参数

为防止项目组定义的参数种类过多，产生冲突，通常设置属性前缀会至少使用两级属性作为前缀进行区分。

日志输出模式是在若干个类别选项中选择某一项，对于此种分类性数据建议制作枚举定义分类数据，当然使用字符串也可以。

```
@ConfigurationProperties(prefix = "tools.ip")  
public class IpProperties {  
    /**  
     * 日志显示周期  
     */  
    private Long cycle = 5L;  
    /**  
     * 是否周期内重置数据  
     */  
    private Boolean cycleReset = false;  
    /**  
     * 日志输出模式 detail: 详细模式 simple: 极简模式  
     */  
    private String model = LogModel.DETAIL.value;  
    public enum LogModel{  
        DETAIL("detail"),  
        SIMPLE("simple");  
        private String value;  
        LogModel(String value) {  
            this.value = value;  
        }  
        public String getValue() {  
            return value;  
        }  
    }
```

```
    }  
}
```

步骤三：加载属性类

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

步骤四：应用配置属性

在应用配置属性的功能类中，使用自动装配加载对应的配置bean，然后使用配置信息做分支处理。

注意：清除数据的功能一定要在输出后运行，否则每次查阅的数据均为空白数据。

```
public class IpCountService {  
    private Map<String, Integer> ipCountMap = new HashMap<String, Integer>();  
    @Autowired  
    private IpProperties ipProperties;  
    @Scheduled(cron = "0/5 * * * * ?")  
    public void print(){  
  
        if(ipProperties.getModel().equals(IpProperties.LogModel.DETAIL.getValue())){  
            System.out.println("          IP访问监控");  
            System.out.println("-----ip-address-----+num--+");  
            for (Map.Entry<String, Integer> entry : ipCountMap.entrySet()) {  
                String key = entry.getKey();  
                Integer value = entry.getValue();  
                System.out.println(string.format("|%18s |%5d |",key,value));  
            }  
            System.out.println("-----+-----+-----+");  
        }else  
        if(ipProperties.getModel().equals(IpProperties.LogModel.SIMPLE.getValue())){  
            System.out.println("          IP访问监控");  
            System.out.println("-----ip-address-----+");  
            for (String key: ipCountMap.keySet()) {  
                System.out.println(string.format("|%18s |",key));  
            }  
            System.out.println("-----+-----+");  
        }  
        //阶段内统计数据归零  
        if(ipProperties.getCycleReset()){  
            ipCountMap.clear();  
        }  
    }  
}
```

温馨提示

每次运行效果之前先clean然后install，切记切记！！

当前效果

在web程序端可以通过控制yml文件中的配置参数对统计信息进行格式控制。但是数据显示周期还未进行控制。

YL-2-5.使用属性配置设置定时器参数

在使用属性配置中的显示周期数据时，遇到了一些问题。由于无法在@Scheduled注解上直接使用配置数据，改用曲线救国的方针，放弃使用@EnableConfigurationProperties注解对应的功能，改成最原始的bean定义格式。

步骤一：@Scheduled注解使用#{}读取bean属性值

此处读取bean名称为ipProperties的bean的cycle属性值

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

步骤二：属性类定义bean并指定bean的访问名称

如果此处不设置bean的访问名称，spring会使用自己的命名生成器生成bean的长名称，无法实现属性的读取

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

步骤三：弃用@EnableConfigurationProperties注解对应的功能，改为导入bean的形式加载配置属性类

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

温馨提示

每次运行效果之前先clean然后install，切记切记！！

当前效果

在web程序端可以通过控制yml文件中的配置参数对统计信息的显示周期进行控制

YL-2-6.拦截器开发

基础功能基本上已经完成了制作，下面进行拦截器的开发。开发时先在web工程中制作，然后将所有功能挪入starter模块中

步骤一：开发拦截器

使用自动装配加载统计功能的业务类，并在拦截器中调用对应功能

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

步骤二：配置拦截器

配置mvc拦截器，设置拦截对应的请求路径。此处拦截所有请求，用户可以根据使用需要设置要拦截的请求。甚至可以在此处加载IpCountProperties中的属性，通过配置设置拦截器拦截的请求。

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

温馨提示

每次运行效果之前先clean然后install，切记切记！！

当前效果

在web程序端导入对应的starter后功能开启，去掉坐标后功能消失，实现自定义starter的效果。

到此当前案例全部完成，自定义stater的开发其实在第一轮开发中就已经完成了，就是创建独立模块导出独立功能，需要使用的位置导入对应的starter即可。如果是在企业中开发，记得不仅需要将开发完成的starter模块install到自己的本地仓库中，开发完毕后还要deploy到私服上，否则别人就无法使用了。

YL-2-7.功能性完善——开启yml提示功能

我们在使用springboot的配置属性时，都可以看到提示，尤其是导入了对应的starter后，也会有对应的提示信息出现。但是现在我们的starter没有对应的提示功能，这种设定就非常的不友好，本节解决自定义starter功能如何开启配置提示的问题。

springboot提供有专用的工具实现此功能，仅需要导入下列坐标。

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

程序编译后，在META-INF目录中会生成对应的提示文件，然后拷贝生成出的文件到自己开发的META-INF目录中，并对其进行编辑。打开生成的文件，可以看到如下信息。其中groups属性定义了当前配置的提示信息总体描述，当前配置属于哪一个属性封装类，properties属性描述了当前配置中每一个属性的具体设置，包含名称、类型、描述、默认值等信息。hints属性默认是空白的，没有进行设置。hints属性可以参考springboot源码中的制作，设置当前属性封装类专用的提示信息，下例中为日志输出模式属性model设置了两种可选提示信息。

```
{
  "groups": [
    {
      "name": "tools.ip",
      "type": "cn.itcast.properties.IpProperties",
      "sourceType": "cn.itcast.properties.IpProperties"
    }
  ],
  "properties": [
    {
      "name": "tools.ip.cycle",
      "type": "java.lang.Long",
      "description": "日志显示周期",
      "sourceType": "cn.itcast.properties.IpProperties",
      "defaultValue": 5
    },
    {
      "name": "tools.ip.cycle-reset",
      "type": "java.lang.Boolean",
      "description": "是否周期内重置数据",
      "sourceType": "cn.itcast.properties.IpProperties",
      "defaultValue": false
    },
    {
      "name": "tools.ip.model",
      "type": "java.lang.String",
      "description": "日志输出模式 detail: 详细模式 simple: 极简模式",
      "sourceType": "cn.itcast.properties.IpProperties"
    }
  ],
  "hints": [
    {
      "name": "tools.ip.model",
      "type": "java.lang.String",
      "description": "日志输出模式 detail: 详细模式 simple: 极简模式",
      "sourceType": "cn.itcast.properties.IpProperties"
    }
  ]
}
```

```

    "values": [
      {
        "value": "detail",
        "description": "详细模式."
      },
      {
        "value": "simple",
        "description": "极简模式."
      }
    ]
  }
}

```

总结

1. 自定义starter其实就是做一个独立的功能模块，核心技术是利用自动配置的效果在加载模块后加载对应的功能
2. 通常会为自定义starter的自动配置功能添加足够的条件控制，而不会做成100%加载对功能的效果
3. 本例中使用map保存数据，如果换用redis方案，在starter开发模块中就要导入redis对应的starter
4. 对于配置属性务必开启提示功能，否则使用者无法感知配置应该如何书写

YL-3.SpringBoot程序启动流程解析

原理篇学习到这里即将结束，最后一章说一下springboot程序的启动流程。对于springboot技术来说，它用于加速spring程序的开发，核心本质还是spring程序的运行，所以于其说是springboot程序的启动流程，不如说是springboot对spring程序的启动流程做了哪些更改。

其实不管是springboot程序还是spring程序，启动过程本质上都是在做容器的初始化，并将对应的bean初始化出来放入容器。在spring环境中，每个bean的初始化都要开发者自己添加设置，但是切换成springboot程序后，自动配置功能的添加帮助开发者提前预设了很多bean的初始化过程，加上各种各样的参数设置，使得整体初始化过程显得略微复杂，但是核心本质还是在做一件事，初始化容器。作为开发者只要搞清楚springboot提供了哪些参数设置的环节，同时初始化容器的过程中都做了哪些事情就行了。

springboot初始化的参数根据参数的提供方，划分成如下3个大类，每个大类的参数又被封装了各种各样的对象，具体如下：

- 环境属性 (Environment)
- 系统配置 (spring.factories)
- 参数 (Arguments、application.properties)

以下通过代码流向介绍了springboot程序启动时每一环节做的具体事情。

```

Springboot30StartupApplication【10】-
>SpringApplication.run(Springboot30StartupApplication.class, args);
  SpringApplication【1332】->return run(new Class<?>[] { primarySource },
args);
  SpringApplication【1343】->return new
SpringApplication(primarySources).run(args);
  SpringApplication【1343】->SpringApplication(primarySources)
  # 加载各种配置信息，初始化各种配置对象
  SpringApplication【266】->this(null, primarySources);

```

```
    SpringApplication【280】->public
SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources)
    SpringApplication【281】->this.resourceLoader =
resourceLoader;
    # 初始化资源加载器
    SpringApplication【283】->this.primarySources = new
LinkedHashSet<>(Arrays.asList(primarySources));
    # 初始化配置类的类名信息（格式转换）
    SpringApplication【284】->this.webApplicationType =
webApplicationType.deduceFromClasspath();
    # 确认当前容器加载的类型
    SpringApplication【285】-
>this.bootstrapRegistryInitializers =
getBootstrapRegistryInitializersFromSpringFactories();
    # 获取系统配置引导信息
    SpringApplication【286】->setInitializers((Collection)
getSpringFactoriesInstances(ApplicationInitializer.class));
    # 获取ApplicationContextInitializer.class对应的实例
    SpringApplication【287】->setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
    # 初始化监听器，对初始化过程及运行过程进行干预
    SpringApplication【288】->this.mainApplicationClass =
deduceMainApplicationClass();
    # 初始化了引导类类名信息，备用
    SpringApplication【1343】->new
SpringApplication(primarySources).run(args)
    # 初始化容器，得到ApplicationContext对象
    SpringApplication【323】->Stopwatch stopwatch = new Stopwatch();
    # 设置计时器
    SpringApplication【324】->stopwatch.start();
    # 计时开始
    SpringApplication【325】->DefaultBootstrapContext
bootstrapContext = createBootstrapContext();
    # 系统引导信息对应的上下文对象
    SpringApplication【327】->configureHeadlessProperty();
    # 模拟输入输出信号，避免出现因缺少外设导致的信号传输失败，进而引发错误（模拟
显示器，键盘，鼠标...）
        java.awt.headless=true
    SpringApplication【328】->SpringApplicationRunListeners listeners
= getRunListeners(args);
    # 获取当前注册的所有监听器
    SpringApplication【329】->listeners.starting(bootstrapContext,
this.mainApplicationClass);
    # 监听器执行了对应的操作步骤
    SpringApplication【331】->ApplicationArguments
applicationArguments = new DefaultApplicationArguments(args);
    # 获取参数
    SpringApplication【333】->ConfigurableEnvironment environment =
prepareEnvironment(listeners, bootstrapContext, applicationArguments);
    # 将前期读取的数据加载成了一个环境对象，用来描述信息
    SpringApplication【333】->configureIgnoreBeanInfo(environment);
    # 做了一个配置，备用
    SpringApplication【334】->Banner printedBanner =
printBanner(environment);
    # 初始化logo
```

```

    SpringApplication【335】->context = createApplicationContext();
    # 创建容器对象，根据前期配置的容器类型进行判定并创建
    SpringApplication【363】-
>context.setApplicationStartup(this.applicationStartup);
    # 设置启动模式
    SpringApplication【337】->prepareContext(bootstrapContext,
context, environment, listeners, applicationArguments, printedBanner);
    # 对容器进行设置，参数来源于前期的设定
    SpringApplication【338】->refreshContext(context);
    # 刷新容器环境
    SpringApplication【339】->afterRefresh(context,
applicationArguments);
    # 刷新完毕后做后处理
    SpringApplication【340】->stopWatch.stop();
    # 计时结束
    SpringApplication【341】->if (this.logStartupInfo) {
        # 判定是否记录启动时间的日志
        SpringApplication【342】-> new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
stopwatch);
        # 创建日志对应的对象，输出日志信息，包含启动时间
        SpringApplication【344】->listeners.started(context);
        # 监听器执行了对应的操作步骤
        SpringApplication【345】->callRunners(context,
applicationArguments);
        # 调用运行器
        SpringApplication【353】->listeners.running(context);
        # 监听器执行了对应的操作步骤

```

上述过程描述了springboot程序启动过程中做的所有的事情，这个时候好奇宝宝们就会提出一个问题。如果想干预springboot的启动过程，比如自定义一个数据库环境检测的程序，该如何将这个过程加入springboot的启动流程呢？

遇到这样的问题，大部分技术是这样设计的，设计若干个标准接口，对应程序中的所有标准过程。当你想干预某个过程时，实现接口就行了。例如spring技术中bean的生命周期管理就是采用标准接口进行的。

```

public class Abc implements InitializingBean, DisposableBean {
    public void destroy() throws Exception {
        //销毁操作
    }
    public void afterPropertiesSet() throws Exception {
        //初始化操作
    }
}

```

springboot启动过程由于存在着大量的过程阶段，如果设计接口就要设计十余个标准接口，这样对开发者不友好，同时整体过程管理分散，十余个过程各自为政，管理难度大，过程过于松散。那springboot如何解决这个问题呢？它采用了一种最原始的设计模式来解决这个问题，这就是监听器模式，使用监听器来解决这个问题。

springboot将自身的启动过程比喻成一个大的事件，该事件是由若干个小的事件组成的。例如：

- org.springframework.boot.context.event.ApplicationStartingEvent

- 应用启动事件，在应用运行但未进行任何处理时，将发送 ApplicationStartingEvent
- org.springframework.boot.context.event.ApplicationEnvironmentPreparedEvent
 - 环境准备事件，当Environment被使用，且上下文创建之前，将发送 ApplicationEnvironmentPreparedEvent
- org.springframework.boot.context.event.ApplicationContextInitializedEvent
 - 上下文初始化事件
- org.springframework.boot.context.event.ApplicationPreparedEvent
 - 应用准备事件，在开始刷新之前，bean定义被加载之后发送 ApplicationPreparedEvent
- org.springframework.context.event.ContextRefreshedEvent
 - 上下文刷新事件
- org.springframework.boot.context.event.ApplicationStartedEvent
 - 应用启动完成事件，在上下文刷新之后且所有的应用和命令行运行器被调用之前发送 ApplicationStartedEvent
- org.springframework.boot.context.event.ApplicationReadyEvent
 - 应用准备就绪事件，在应用程序和命令行运行器被调用之后，将发出 ApplicationReadyEvent，用于通知应用已经准备处理请求
- org.springframework.context.event.ContextClosedEvent (上下文关闭事件，对应容器关闭)

上述列出的仅仅是部分事件，当应用启动后走到某一个过程点时，监听器监听到某个事件触发，就会执行对应的事件。除了系统内置的事件处理，用户还可以根据需要自定义开发当前事件触发时要做的其他动作。

```
//设定监听器，在应用启动开始事件时进行功能追加
public class MyListener implements ApplicationListener<ApplicationStartingEvent>
{
    public void onApplicationEvent(ApplicationStartingEvent event) {
        //自定义事件处理逻辑
    }
}
```

按照上述方案处理，用户就可以干预springboot启动过程的所有工作节点，设置自己的业务系统中独有的功能点。

总结

1. springboot启动流程是先初始化容器需要的各种配置，并加载成各种对象，初始化容器时读取这些对象，创建容器
2. 整体流程采用事件监听的机制进行过程控制，开发者可以根据需要自行扩展，添加对应的监听器绑定具体事件，就可以在事件触发位置执行开发者的业务代码

原理篇完结

原理篇到这里就要结束了，springboot2整套课程的基础篇、实用篇和原理篇就全部讲完了。至于后面的番外篇由于受B站视频上传总量不得超过200个视频的约束，番外篇的内容不会在当前课程中发布了，会重新定义一个课程继续发布，至于具体时间，暂时还无法给到各位小伙伴。

原理篇个人感觉略微有点偷懒，怎么说呢？学习原理篇需要的前置铺垫知识太多，比如最后一节讲到启动流程时，看到reflush方法时我就想现在在看这套课程的小伙伴是否真的懂这个过程呢？但是如果把这些东西都讲了，那估计要补充的知识就太多了，就是将spring的很多知识加入到这里面重新讲解了，会出现喧宾夺主的现象。很纠结，('・へ・`)

课程做到这里就要和各位小伙伴先say拜了，感谢各位小伙伴的支持，也欢迎各位小伙伴持续关注黑马程序员出品的各种视频教程。黑马程序员的每位老师做课程都是认真的，都是为了各位致力于IT研发事业的小伙伴能够学习之路上少遇沟沟坎坎，顺利到达成功的彼岸。

番外篇，さようなら！ 안녕히 계십시오！ ແລ້ວເຈອກັນ! До свидания ! خداحفظ !