拓薪教育
WWW.TXJAVA.CN

讲师：任亮

1 Spring Boot 基础篇

1.1Spring Boot 简介

Spring Boot 是所有基于 Spring 开发的项目的起点。SpringBoot 其实不是什么新的框架，它默认配置了很多框架的使用方式，就像 maven 整合了所有的 jar 包，spring boot 整合了很多的技术，提供 了 JavaEE 的大整合。

1.1.1 springboot 的学习铺垫和要求

1. 对 spring 的 IOC 有深入的理解，springboot 是基于 spring 的。
2. 深入掌握 spring 的注解开发。传统老程序员对注解认识偏少。
3. 最好有 ssm 项目的开发经验，因为 springboot 是整合了 javaEE 的技术。
4. 有较好的源码学习经验。
5. 有 idea 和 eclipse 的使用经验，本次课程使用 idea。
6. 在国内 IT 公司是一个主要的招聘要求。

1.1.2 、SpringBoot 主要特性

spring 官方的网站：https://spring.io/

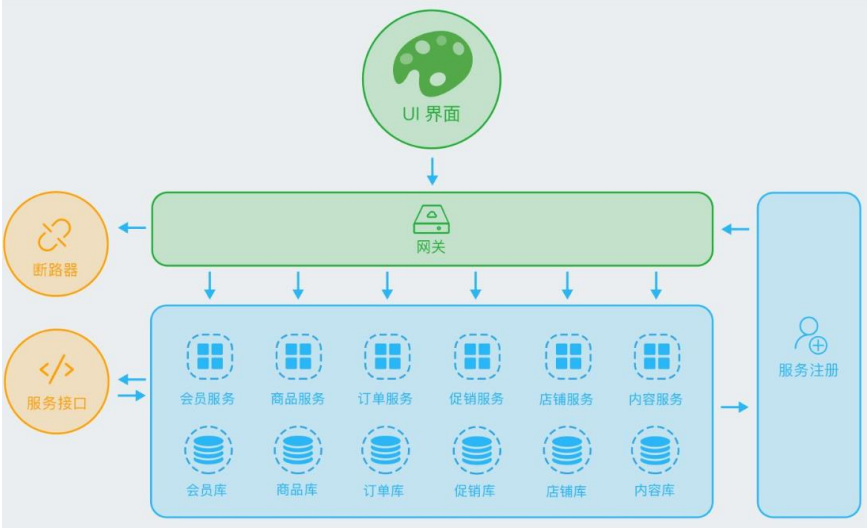
1、 SpringBoot Starter：他将常用的依赖分组进行了整合，将其合并到一个依赖中，这样就可以一次性添加到项目的 Maven 或 Gradle 构建中；

2、 使编码变得简单，SpringBoot 采用 JavaConfig 的方式对 Spring 进行配置，并且提供了大量的注解，极大的提高了工作效率。

3、 自动配置：SpringBoot 的自动配置特性利用了 Spring 对条件化配置的支持，合理地推测应用所需的 bean 并自动化配置他们；

4、 使部署变得简单，SpringBoot 内置了三种 Servlet 容器，Tomcat，Jetty,undertow.我们只需要一个 Java 的运行环境就可以跑 SpringBoot 的项目了，SpringBoot 的项目可以打成一个 jar 包。

5、 现在流行微服务与分布式系统，springboot 就是一个非常好的微服务开发框架，你可以使用它快速的搭建起一个系统。同时，你也可以使用 spring cloud（Spring Cloud 是一个基于 Spring Boot 实现的云应用开发工具）来搭建一个分布式的架构。



1.1.3 springboot 缺点

1. 将现有或传统的 Spring Framework 项目转换为 Spring Boot 应用程序是一个非常困难和耗时的过程。它仅适用于全新 Spring 项目。

2. 使用简单，学习成本高，精通难。

1.2 环境准备

jdk1.8：Spring Boot 推荐 jdk1.8 及以上；

maven3.x：maven 3.3 以上版本；

IntelliJIDEA2019，不要使用 2017

SpringBoot 2.3.0.RELEASE

1.2.1 MAVEN 设置

1. 配置阿里云镜像

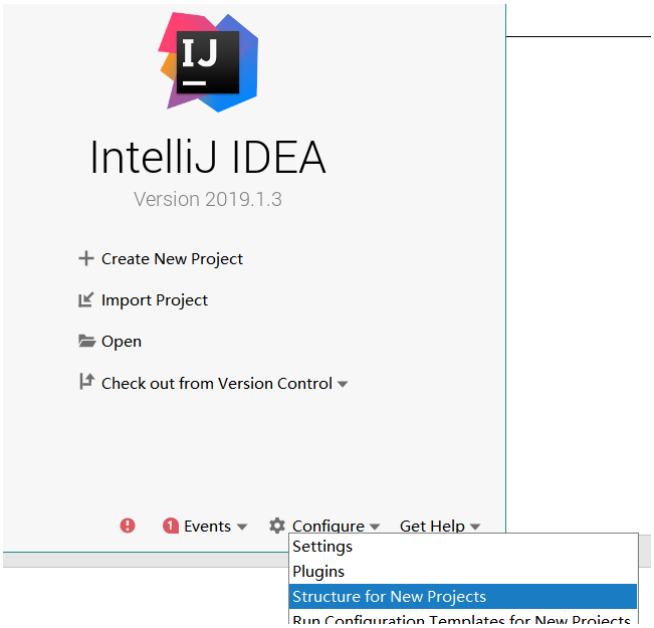
```
<mirror>
        <id>nexus-aliyun</id>
        <mirrorOf>central</mirrorOf>
        <name>Nexus aliyun</name>
        <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

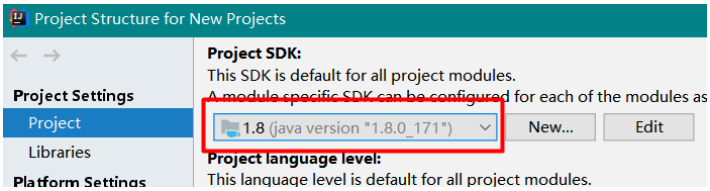
2. 给 maven 的 settings.xml 配置文件的 profiles 标签添加

```
<profile>
        <id>jdk-1.8</id>
        <activation>
            <activeByDefault>true</activeByDefault>
                <jdk>1.8</jdk>
        </activation>
        <properties>
            <maven.compiler.source>1.8</maven.compiler.source>
            <maven.compiler.target>1.8</maven.compiler.target>
            <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
        </properties>
    </profile>
```

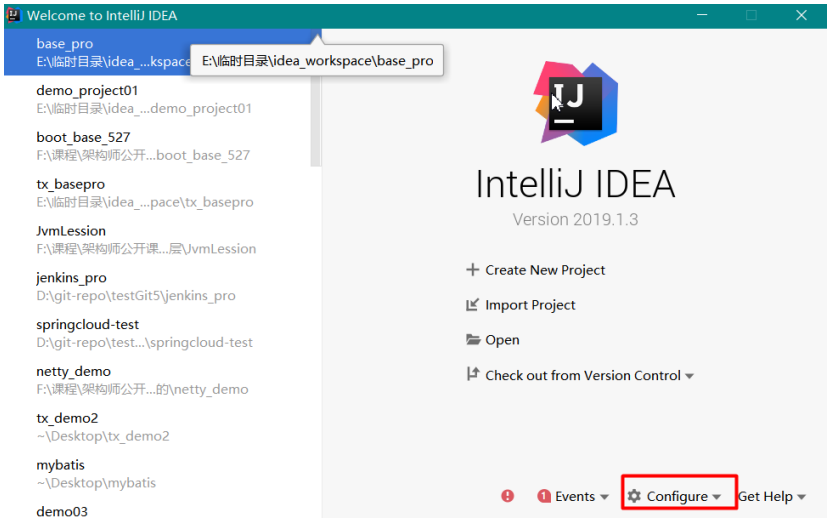
1.2.2 IDEA 设置

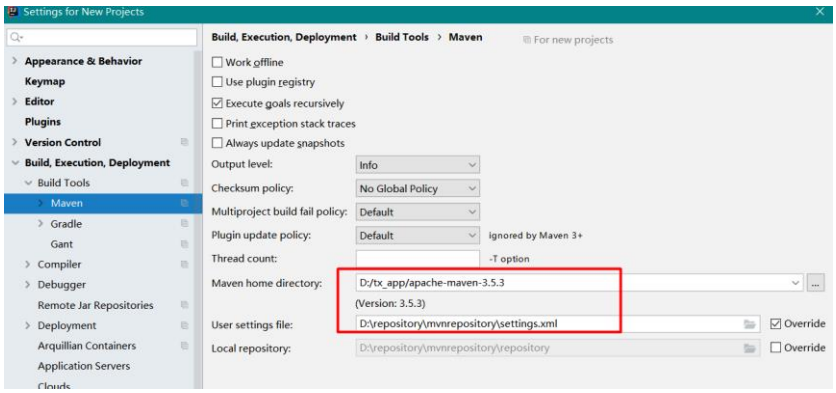
1. 指定 jdk 环境





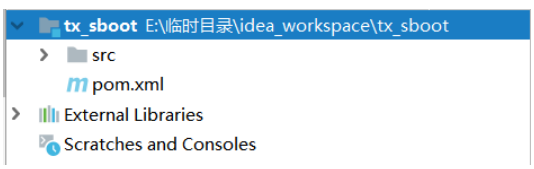
2. 指定 maven 环境





1.3 Spring Boot HelloWorld

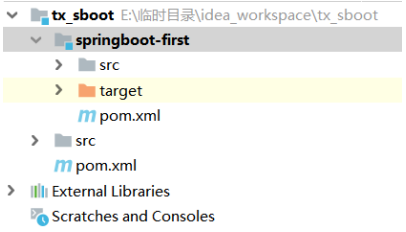
1.3.1 创建一个 maven 父工程 tx_sboot（pom）



pom.xml

```xml
<?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>

    <groupId>cn.tx.springboot</groupId>
    <artifactId>tx_sboot</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>



    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.0.RELEASE</version>
    </parent>



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

1.3.2 在父工程下创建 springboot-first（jar）



1.3.3 创建测试 Controller

```
@RestController
public class TestController {


    @RequestMapping("hello")
    public String hello(){
        return "hello";
    }
}
```

1.3.4 创建一个 springboot 启动类

```
@SpringBootApplication
public class FirstSpringApplication {


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

启动并且测试：http://localhost:8080/hello
成功。

1.3.5 在父工程 tx_sboot 中加入构建依赖

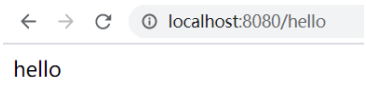
```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

我们可以把 springboot 工程打成可执行的 jar

```
mvn -f springboot-first clean package
```

打完 jar 包后，我们切入到对应的 jar 包里面执行

```
E:\临时目录\idea_workspace\tx_sboot\springboot-first\target>java -jar springboot-first-1.0-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.0.RELEASE)

2020-06-10 11:32:28.637  INFO 20600 --- [           main] cn.tx.sboot.FirstSpringApplication       : Starting FirstSprin
gApplication v1.0-SNAPSHOT on MACHENI-B936LIA with PID 20600 (E:\临时目录\idea_workspace\tx_sboot\springboot-first\targe
t\springboot-first-1.0-SNAPSHOT.jar started by rls1180506 in E:\临时目录\idea_workspace\tx_sboot\springboot-first\target
)
2020-06-10 11:32:28.640  INFO 20600 --- [           main] cn.tx.sboot.FirstSpringApplication       : No active profile s
et, falling back to default profiles: default
2020-06-10 11:32:30.201  INFO 20600 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized
```



hello

1.3.6 第一个例子做完的疑问。

1. starter 是什么？我们何如去使用这些 starter？
2. Tomcat 的内嵌是如何完成
3. 使用了 web 对应的 starter，springmvc 是如何自动装配？
4. 我们如何来配置我们自定义话的相关内容。

1.4 默认扫描器 basepackage

springboot 的主启动类所在的 package 就是扫描器的 basepackage
如图所示 com.example.myapplication 就是我们的扫描器中 basepackge

```
com
 +- example
     +- myapplication
         +- Application.java
         |
         +- customer
         |   +- Customer.java
         |   +- CustomerController.java
         |   +- CustomerService.java
         |   +- CustomerRepository.java
         |
         +- order
             +- Order.java
             +- OrderController.java
             +- OrderService.java
             +- OrderRepository.java
```

源码解析

@AutoConfigurationPackage

自动配置包负责 basepackge 的注册

@AutoConfigurationPackage 内部使用@Import 来做 bean 的定义的注册

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {

    /**
     * Base packages that should be registered with {@link
     * <p>
```

让我们进入 AutoConfigurationPackages.Registrar，通过 register 的调用来注册 basepackage 的 bean 定义的

```
static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {

    @Override
    public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {
        register(registry, new PackageImports(metadata).getPackageNames().toArray(new String[0]));
    }

    @Override
    public Set<Object> determineImports(AnnotationMetadata metadata) {
        return Collections.singleton(new PackageImports(metadata));
    }

}
```

进入到 PackageImports，获得 basepackge 设置给 packageNames

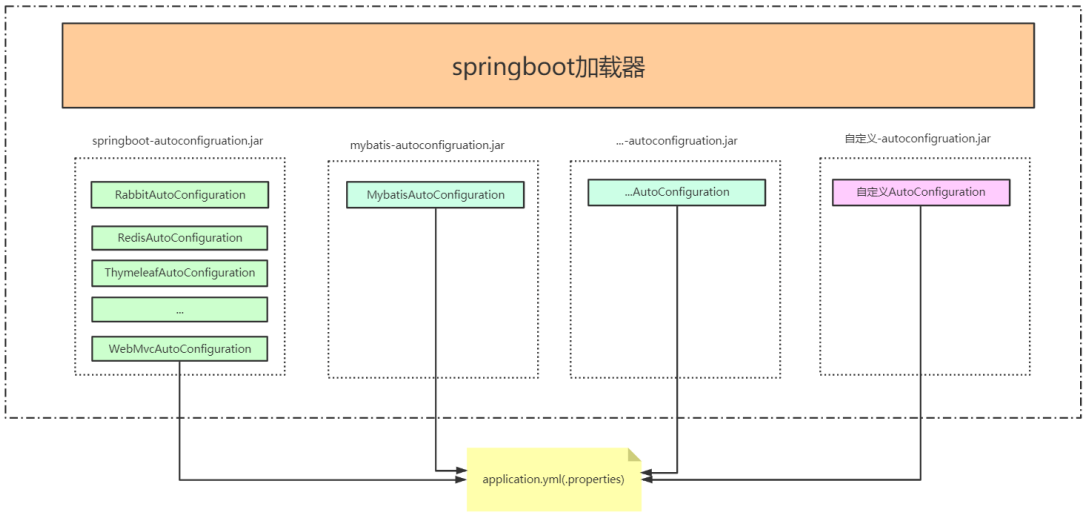
```
private static final class PackageImports {

    private final List<String> packageNames;

    PackageImports(AnnotationMetadata metadata) {
        AnnotationAttributes attributes = AnnotationAttributes
                .fromMap(metadata.getAnnotationAttributes(AutoConfigurationPackage.class.getName(), classValuesAsString: false));
        List<String> packageNames = new ArrayList<>();
        for (String basePackage : attributes.getStringArray( attributeName: "basePackages")) {
            packageNames.add(basePackage);
        }
        for (Class<?> basePackageClass : attributes.getClassArray( attributeName: "basePackageClasses")) {
            packageNames.add(basePackageClass.getPackage().getName());
        }
        if (packageNames.isEmpty()) {
            packageNames.add(ClassUtils.getPackageName(metadata.getClassName()));
        }
        this.packageNames = Collections.unmodifiableList(packageNames);
    }
```

回到上一层进入到 register，创建 bean 的定义并且把 packageNames 设置给 bean 定义然后把 bean 定义的做注册。

```
93  @       public static void register(BeanDefinitionRegistry registry, String... packageNames) {
94  ●           if (registry.containsBeanDefinition(BEAN)) {
95                  BeanDefinition beanDefinition = registry.getBeanDefinition(BEAN);
96                  ConstructorArgumentValues constructorArguments = beanDefinition.getConstructorArgumentValues();
97                  constructorArguments.addIndexedArgumentValue( index: 0, addBasePackages(constructorArguments, packageNames));
98              }
99              else {
00                  GenericBeanDefinition beanDefinition = new GenericBeanDefinition();
01                  beanDefinition.setBeanClass(BasePackages.class);
02                  beanDefinition.getConstructorArgumentValues().addIndexedArgumentValue( index: 0, packageNames);
03                  beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
04                  registry.registerBeanDefinition(BEAN, beanDefinition);
05              }
06      }
```

1.5 自动配置浅析

springboot 扫描当前 classpath 下所有的 jar 包，筛选出来 EnableAutoConfiguration 下的所有自动配置类注入到 spring 容器中，完成自动的 bean 的配置。



2 热部署

在实际开发过程中，每次修改代码就得将项目重启，重新部署，对于一些大型应用来说，重启时间需要花费大量的时间成本。对于一个后端开发者来说，重启过程确实很难受啊。在 Java 开发领域，热部署一直是一个难以解决的问题，目前的 Java 虚拟机只能实现方法体的修改热部署，对于整个类的结构修改，仍然需要重启虚拟机，对类重新加载才能完成更新操作。下面我们就看看对于简单的类修改的热部署怎么实现。

2.1 原理

深层原理是使用了两个 ClassLoader，一个 Classloader 加载那些不会改变的类（第三方 Jar 包），另一个 ClassLoader 加载会更改的类，称为 restart ClassLoader,这样在有代码更改的时候，原来的 restart ClassLoader 被丢弃，重新创建一个 restart ClassLoader，由于需要加载的类相比较少，所以实现了较快的重启时间。

2.2 devtools 工具包

devtools 会监听 classpath 下的文件变动，并且会立即重启应用（发生在保存时机），java 类文件热部署（类文件修改后不会立即生效），实现对属性文件的热部署。

devtools 可以实现页面热部署（页面修改后会立即生效，这个可以直接在 application.properties 文件中配置 spring.thymeleaf.cache=false 来实现，后面讲到）。

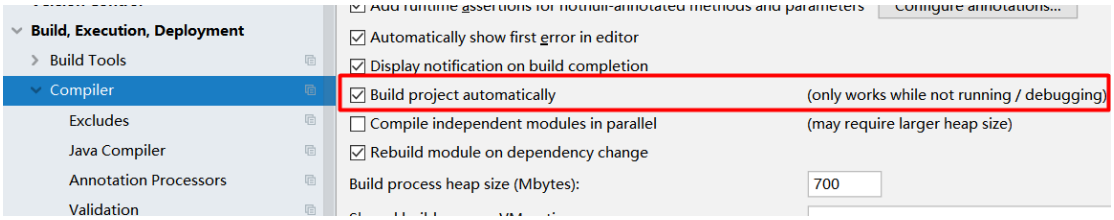
2.3idea 的工具设置

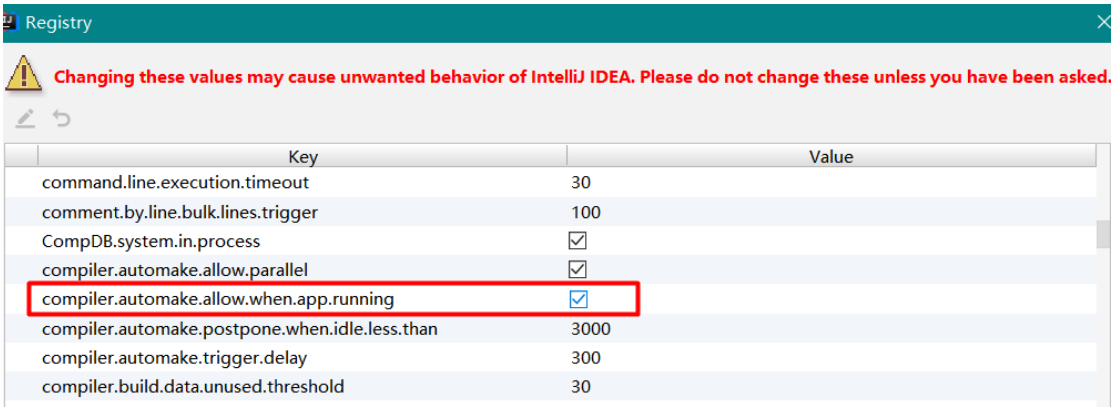
当我们修改了类文件后，idea 不会自动编译，需要通过 ctrl+F9 来触发，

如果想要自动生效得修改 idea 设置，该功能按着个人的喜好来设置，修改类后，当我们窗口切换时候可以看到热部署的发生

（1）File-Settings-Compiler-Build Project automatically

（2）ctrl + shift + alt + / ,选择 Registry,勾上 Compiler autoMake allow when app running





2.4 热部署的排除

默认情况下，/META-INF/maven，/META-INF/resources，/resources，/static，/templates，/public 这些文件夹下的文件修改不会使应用重启，但是会重新加载（devtools 内嵌了一个 LiveReload server，当资源发生改变时，浏览器刷新）。

1. 我们在 resources/static 目录下创建 tx.js 文件每次发生修改后的并不重启，而是采用 livereload 的方式。
2. 同时我们可以根据自己的意愿来设置想要排除的资源

```
spring.devtools.restart.exclude=static/**,public/**
```

3 boot 的属性配置文件

3.1配置文件位置

springboot 启动会扫描以下位置的 application.properties 或者 application.yml 文件作为 Spring boot 的默认配置文件

–file:./config/

–file:./ 项目的跟路径，如果当前的项目有父工程，配置文件要放在父工程 的根路径

–classpath:/config/

–classpath:/

优先级由高到底，高优先级的配置会覆盖低优先级的配置；

SpringBoot 会从这四个位置全部加载主配置文件；互补配置；

如果我们的配置文件名字不叫 application.properties 或者 application.yml，可以通过以下参数来指定配置文件的名字，myproject 是配置文件名

```
$ java -jar myproject.jar --spring.config.name=myproject
```

我们同时也可以指定其他位置的配置文件来生效

```
$ java -jar myproject.jar
--spring.config.location=classpath:/default.properties,classpath:/override.properties
```

3.2 配置文件

3.2.1 yaml

yml 是 YAML（YAML Ain't Markup Language）语言的文件，以数据为中心，比 properties、xml 等更适合做配置文件

- yml 和 xml 相比，少了一些结构化的代码，使数据更直接，一目了然。
- 相比 properties 文件更简洁

3.2.2 yaml 语法

以空格的缩进程度来控制层级关系。空格的个数并不重要，只要左边空格对齐则视为同一个层级。且大小写敏感。支持字面值，对象，数组三种数据结构，也支持复合结构。

- 字面值：字符串，布尔类型，数值，日期。字符串默认不加引号，单引号会转义特殊字符。日期格式支持 yyyy/MM/dd HH:mm:ss
- 对象：由键值对组成，形如 key:(空格)value 的数据组成。冒号后面的空格是必须要有的，每组键值对占用一行，且缩进的程度要一致，也可以使用行内写法：{k1: v1, ...kn: vn}
- 数组：由形如 -(空格)value 的数据组成。短横线后面的空格是必须要有的，每组数据占用一行，且缩进的程度要一致，也可以使用行内写法： [1,2,...n]
- 复合结构：上面三种数据结构任意组合

3.2.3 yaml 的运用

创建一个 Spring Boot 的全局配置文件 application.yml，配置属性参数。主要有字符串，带特殊字符的字符串，布尔类型，数值，集合，行内集合，行内对象，集合对象这几种常用的数据格式。

```yaml
yaml:
  str: 字符串可以不加引号
  specialStr: "双引号直接输出\n 特殊字符"
  specialStr2: '单引号可以转义\n 特殊字符'
  flag: false
  num: 666
  Dnum: 88.88
  list:
    - one
    - two
    - three
  set: [1,2,2,3]
  map: {k1: v1, k2: v2}
  positions:
    - name: txjava
      salary: 15000.00
    - name: liangge
      salary: 18888.88
```

创建实体类 YamlEntity.java 获取配置文件中的属性值，通过注解@ConfigurationProperties 获取配置文件中的指定值并注入到实体类中。

```java
@Component
@ConfigurationProperties(prefix = "yaml")
public class YamlEntity {

    // 字面值，字符串，布尔，数值
    private String str; // 普通字符串
    private String specialStr; // 转义特殊字符串
    private String specialStr2;// 输出特殊字符串
    private Boolean flag;   // 布尔类型
    private Integer num;    // 整数
    private Double dNum;    // 小数

    // 数组，List 和 Set，两种写法： 第一种：-空格 value，每个值占一行，需缩进对齐；第二种：[1,2,...n] 行内
```

```
写法
    private List<Object> list;  // list 可重复集合
    private Set<Object> set;    // set 不可重复集合

    //Map 和实体类，两种写法：第一种：key 空格 value，每个值占一行，需缩进对齐；第二种：{key: value,....} 行内写法
    private Map<String, Object> map; // Map K-V
    private List<Position> positions;  // 复合结构，集合对象

    // 省略 getter，setter，toString 方法
```

3.2.4 总结

- 字符串可以不加引号，若加双引号则输出特殊字符，若不加或加单引号则转义特殊字符；
- 数组类型，短横线后面要有空格；对象类型，冒号后面要有空格；
- YAML 是以空格缩进的程度来控制层级关系，但不能用 tab 键代替空格，大小写敏感；
- yaml 的缺点是可读性比较差

3.3 属性绑定

前面给大家讲了 yaml 的语法和绑定注入给实体类，那我们平时在工作中多数是在通过实体类来写 yaml 的配置。属性的绑定我们必须要提供 set 方法

提供一段来自官网的代码：

```
@Component
@ConfigurationProperties("acme")
public class AcmeProperties {

    private boolean enabled;

    private InetAddress remoteAddress;

    private final Security security = new Security();

    public boolean isEnabled() {
        return enabled;
    }
```

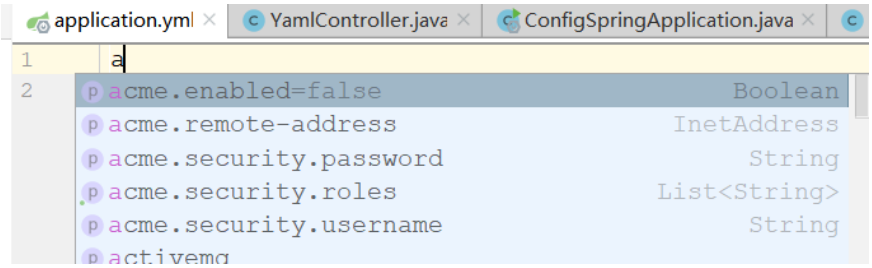
```
public void setEnabled(boolean enabled) {
    this.enabled = enabled;
}

public InetAddress getRemoteAddress() {
    return remoteAddress;
}

public void setRemoteAddress(InetAddress remoteAddress) {
    this.remoteAddress = remoteAddress;
}

public Security getSecurity() {
    return security;
}

public static class Security {

    private String username;

    private String password;

    private List<String> roles = new ArrayList<>(Collections.singleton("USER"));

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public List<String> getRoles() {
        return roles;
    }
```

```
        public void setRoles(List<String> roles) {
            this.roles = roles;
        }
    }
}
```

为了让当前的实体类能在配置文件中有对应的提示，我们需要引入如下的依赖，

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

加完依赖后通过 Ctrl+F9 来使之生效。



在配置文件中加入

```
acme:
  remote-address: 192.168.0.108
  enabled: false
  security:
    username: txjava
    password: 123
    roles:
      - manager
      - coder
      - tester
```

然后我们测试绑定的情况。

在属性绑定的方式里，我们是通过 set 方法来完成的，我们可以借助 Lombok 来给我们带来方便。

我们在父工程中引入 Lombok 的依赖：

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.20</version>
</dependency>
```

修改属性类：加上@Data 注解，然后测试结果相同。

```
@Data
@Component
@ConfigurationProperties("acme")
public class AcmeProperties {

    private boolean enabled;

    private InetAddress remoteAddress;

    private final Security security = new Security();


    @Data
    public static class Security {

        private String username;

        private String password;

        private List<String> roles = new ArrayList<>(Collections.singleton("USER"));


    }
}
```

3.4构造器绑定

让我们采用构造器的方式来定义

```
@ConfigurationProperties("acme")
@ConstructorBinding
public class AcmeProperties {

    private boolean enabled;

    private InetAddress remoteAddress;

    private final Security security ;

    public AcmeProperties(boolean enabled, InetAddress remoteAddress, Security security) {
        this.enabled = enabled;
        this.remoteAddress = remoteAddress;
```

```
        this.security = security;
    }

    public boolean isEnabled() {
        return enabled;
    }

    public InetAddress getRemoteAddress() {
        return remoteAddress;
    }

    public Security getSecurity() {
        return security;
    }

    public static class Security {

        private String username;

        private String password;

        private List<String> roles;

        public Security(String username, String password,
                @DefaultValue("USER") List<String> roles) {
            this.username = username;
            this.password = password;
            this.roles = roles;
        }

        public String getUsername() {
            return username;
        }

        public String getPassword() {
            return password;
        }

        public List<String> getRoles() {
            return roles;
        }
    }
}
```

要使用构造函数绑定,必须使用@EnableConfigurationProperties 或配置属性扫描启用类。不能对由常规 Spring 机制创建的 Bean 使用构造函数绑定（例如@Component Bean、通过@Bean 方法创建的 Bean 或使用@Import 加载的 Bean）

测试的 Controller

```
@RestController
@EnableConfigurationProperties(AcmeProperties.class)
public class YamlController {


    @Autowired
    private AcmeProperties acmeProperties;


    @RequestMapping("yaml")
    public AcmeProperties yaml(){
        System.out.println(acmeProperties);
        return acmeProperties;
    }
}
```

测试成功。在属性绑定的案例中我们同样也可以使用@EnableConfigurationProperties，此时不需要提供@Component

如果一个配置类只配置@ConfigurationProperties 注解，而没有使用@Component，那么在 IOC 容器中是获取不到 properties 配置文件转化的 bean。说白了 @EnableConfigurationProperties 相当于把使用 @ConfigurationProperties 的类进行了启用注入。

在之前的版本我们都是使用@Configuration 来进行作为配置类，从 SpringBoot2.2.1.RELEASE 版本开始我们不再需要添加@Configuration。
我们可以在扫描范围的 bean 的内部之间定义 bean 如：

```
@RestController
public class YamlController {
    @Bean
    public Dep getDep(){
        return new Dep();
    }


    @Autowired
    private AcmeProperties acmeProperties;
}
```

3.5第三方组件注入

除了使用@ConfigurationProperties 注释类之外，还可以在 public@Bean 方法上使用它。如果要将属性绑定到不在您控制范围内的第三方组件

依然采用之前的案例的 yaml 配置

创建一个其他组件类

```
@Data
public class AnotherComponent {

    private boolean enabled;

    private InetAddress remoteAddress;
}
```

创建 MyService

```
@Component
public class MyService {

    @ConfigurationProperties("acme")
    @Bean
    public AnotherComponent anotherComponent(){
        return new AnotherComponent();
    }
}
```

我们通过测试可以获得 AnotherComponent 组件的实例对象。

3.6松散绑定

Spring Boot 使用一些宽松的规则将环境属性绑定到@ConfigurationProperties bean，因此环境属性名和 bean 属性名之间不需要完全匹配。

例如属性类：

```
@Data
@Component
@ConfigurationProperties("acme.my-person.person")
public class OwnerProperties {

    private String firstName;

}
```

配置文件：

```
acme:
  my-person:
    person:
      first-name: 泰森
```

属性文件中配置	说明
acme.my-project.person.first-name	羊肉串模式 case，推荐使用
acme.myProject.person.firstName	标准驼峰模式
acme.my_project.person.first_name	下划线模式
ACME_MYPROJECT_PERSON_FIRSTNAME	大写下划线，如果使用系统环境时候推荐使用

3.7 @ConfigurationProperties 校验

每当使用 Spring 的@Validated 注释对@ConfigurationProperties 类进行注释时，Spring Boot 就会尝试验证它们。你可以用 JSR-303 javax.validation 直接在配置类上的约束注释。为此，请确保类路径上有一个兼容的 JSR-303 实现，此处我们用的是 hibernate 的实现，然后将约束注释添加到字段中

1. 引入依赖

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.2.0.Final</version>
</dependency>
```

2. 在属性类上加入注解

```
@Data
@Component
@ConfigurationProperties("acme.my-person.person")
@Validated //spring 提供的注解
```

```java
public class OwnerProperties {



    @NotNull   //javax.validation.constraints 提供
    private String firstName;

    @Max(35)
    private int age;

    @Email
    private String email;

}
```

3. 配置文件

```yaml
acme:
  my-person:
    person:
      FIRST_name: 泰森
      age: 34
      email: aaa
```

启动主启动类的时候，会自动发生校验。

Feature	@ConfigurationProperties	@Value
松散绑定	Yes	Limit
元数据支持	Yes	No
SpEL 表达式	No	Yes
复杂类型绑定	Yes	No
校验	Yes	No
应用场景	Boot 里面属性多个绑定	单个属性的绑定

3.8 @ConfigurationProperties vs. @Value

1. 松散绑定在@value 是被限制的

如果您确实想使用@Value，建议引用属性名（kebab case 只使用小写字母，既是羊肉串模式）。这允许 Spring Boot 使用与放松 binding@ConfigurationProperties 时相同的逻辑。例如，@Value（“${demo.item-price}）将匹配 demo.item-price 和 demo.itemPrice， 其他模式不能匹配。

2. 元数据支持
   我们在@ConfigurationProperties 方式可以生成元数据，目的是给我们提供提示和属性的描述。但是在@value 里面是没有的。@Value 适合单个的属性注入
3. spEL 在@ConfigurationProperties 中是不能支持的。在@Value 中可以支持
   如：

```
@Value("#{12*3}")
private int age;
```

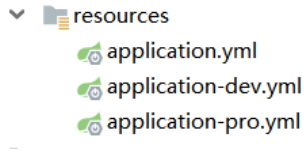
4. @Value 复杂类型不可注入，会有启动报错。

3.9 Profile

我们可以通过多样性的文档来解决多环境的需求。在一个 yml 中我们可以把文档划分成多个块

```
acme:
  enabled: true
  remote-address: 192.168.0.108
  spring:
    profiles: default
---
spring:
  profiles: development
acme:
  enabled: true
  remote-address: 192.168.0.109

---
spring:
  profiles: production
acme:
  enabled: true
  remote-address: 192.168.0.110
```

在启动的时候我们通过 **spring.profiles.active: development** 来指定开启哪个 profile
我们也可以采用多个文件来做。
我们创建 application-dev.yml, application-pro.yml
application-{profile}.xml



属性类：

```
@Data
@Component
@ConfigurationProperties("acme")
public class AcmeProperties {

    private boolean enabled;

    private InetAddress remoteAddress;

    private String host;

}
```

我们可以通过 profile 的指定来启用指定的文件。

application-dev.yml

```
acme:
  enabled: true
  remote-address: 192.168.0.109
  host: ${acme.remote-address}:8080
```

application-pro.yml

```
acme:
  enabled: true
  remote-address: 192.168.0.110
  host: ${acme.remote-address}:8080
```

我们依然在启动的时候我们通过 **spring.profiles.active: development** 来指定。

我们可以通过${}方式获取当前文档中配置和 jvm 的参数中的配置值。

4 springboot 自动配置解读

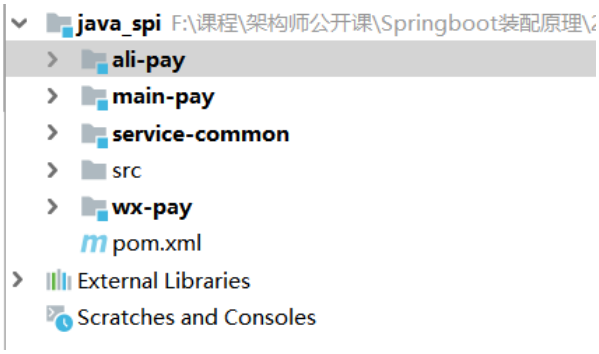
4.1java 中的 spi

SPI 的全名为 Service Provider Interface.大多数开发人员可能不熟悉，因为这个是针对厂商或者插件的。在 java.util.ServiceLoader 的文档里有比较详细的介绍。

简单的总结下 java SPI 机制的思想。我们系统里抽象的各个模块，往往有很多不同的实现方案。面向的对象的设计里，我们一般推荐模块之间基于接口编程，模块之间不对实现类进行硬编码。一旦代码里涉及具体的实现类，就违反了可拔插的原则，如果需要替换一种实现，就需要修改代码。为了实现在模块装配的时候能不在程序里动态指明，这就需要一种服务发现机制。

java SPI 就是提供这样的一个机制：为某个接口寻找服务实现的机制。有点类似 IOC 的思想，就是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要。

详情参考用例代码



Java SPI 规范

要使用 Java SPI，需要遵循如下约定：

1、当服务提供者提供了接口的一种具体实现后，在 jar 包的 META-INF/services 目录下创建一个以“接口全路径名”为命名的文件，内容为实现类的全限定名；

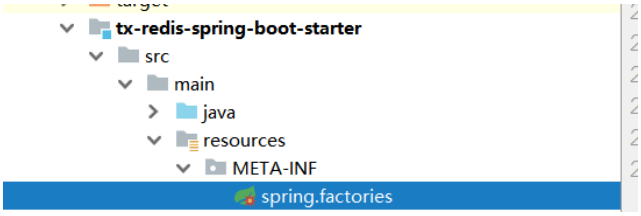
2、接口实现类所在的 jar 包放在主程序的 classpath 中；

3、主程序通过 java.util.ServiceLoder 动态装载实现模块，它通过扫描 META-INF/services 目录下的配置文件找到实现类的全限定名，把类加载到 JVM；

4、SPI 的实现类必须携带一个不带参数的构造方法；

4.2 Spring Boot 中的 SPI 机制

在 Spring 中也有一种类似与 Java SPI 的加载机制。它在 META-INF/spring.factories 文件中配置接口的实现类名称，然后在程序中读取这些配置文件并实例化。
这种自定义的 SPI 机制是 Spring Boot Starter 实现的基础。



4.3源码的引入

在 GitHub 上下载源码，解压导入到 idea
clean install -DskipTests -Pfast

4. 4 Spring Factories 实现原理

spring-core 包里定义了 SpringFactoriesLoader 类，这个类实现了检索 META-INF/spring.factories 文件，并获取指定接口的配置的功能。在这个类中定义了两个对外的方法：

```
public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable
ClassLoader classLoader) {
//获得接口名字
   String factoryClassName = factoryClass.getName();
//获得所有配置类，并且根据接口名字来获得
   return loadSpringFactories(classLoader).getOrDefault(factoryClassName,
```

```
Collections.emptyList());
}
```

```
private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader) {
//从缓存中获得 spring.factories 的全量信息
   MultiValueMap<String, String> result = (MultiValueMap)cache.get(classLoader);
   if (result != null) {
      return result;
   } else {
      try {
         //在 classpath 下的所有 jar 包中查找 META-INF/spring.factories 文件
         Enumeration<URL> urls = classLoader != null ? classLoader.getResources("META-INF/spring.factories") :
ClassLoader.getSystemResources("META-INF/spring.factories");
         //定义存储全量工厂类的 map
         LinkedMultiValueMap result = new LinkedMultiValueMap();
         //遍历 urls
         while(urls.hasMoreElements()) {
            URL url = (URL)urls.nextElement();
            UrlResource resource = new UrlResource(url);
         //加载属性集和
            Properties properties = PropertiesLoaderUtils.loadProperties(resource);
            Iterator var6 = properties.entrySet().iterator();
            //遍历属性键值对的键
            while(var6.hasNext()) {
               Entry<?, ?> entry = (Entry)var6.next();
               //获得 key 接口
               String factoryClassName = ((String)entry.getKey()).trim();
               String[] var9 = StringUtils.commaDelimitedListToStringArray((String)entry.getValue());
               int var10 = var9.length;
                //切分并且遍历接口实现类，加入结果集
               for(int var11 = 0; var11 < var10; ++var11) {
                  String factoryName = var9[var11];
                  result.add(factoryClassName, factoryName.trim());
               }
            }
         }

         cache.put(classLoader, result);
         return result;
      } catch (IOException var13) {
```

```
            throw new IllegalArgumentException("Unable to load factories from location
[META-INF/spring.factories]", var13);
        }
    }
}
```

从代码中我们可以知道，在这个方法中会遍历整个 ClassLoader 中所有 jar 包下的 spring.factories 文件。也就是说我们可以在自己的 jar 中配置 spring.factories 文件，不会影响到其它地方的配置，也不会被别人的配置覆盖。

spring.factories 的是通过 Properties 解析得到的，所以我们在写文件中的内容都是安装下面这种方式配置的：

```
com.xxx.interface=com.xxx.classname
```

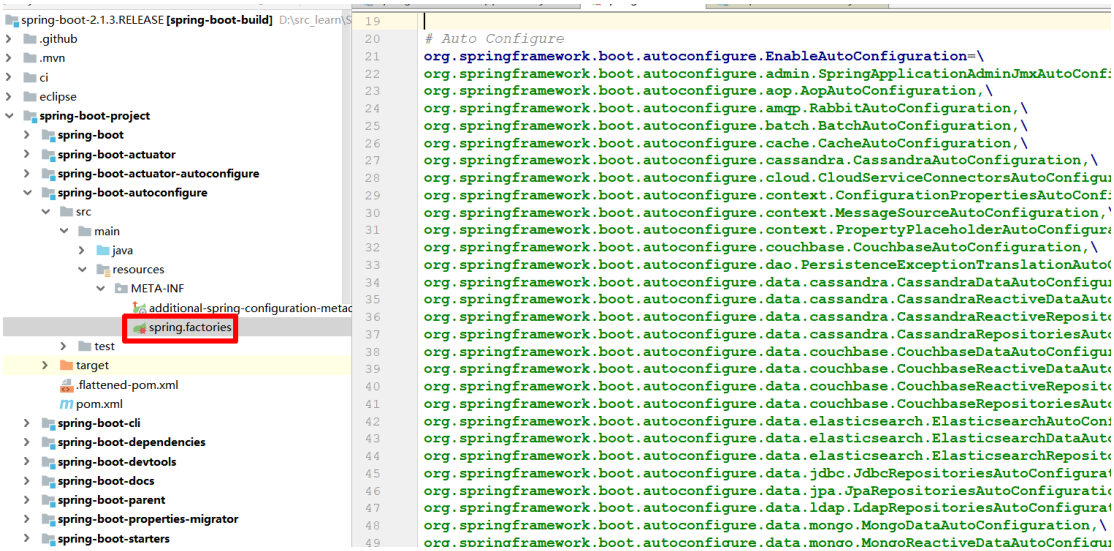
如果一个接口希望配置多个实现类，可以使用’,’进行分割。

在 Spring Boot 的很多包中都能够找到 spring.factories 文件，下面就是 spring-boot 包中的 spring.factories 文件

在 Spring Boot 中，使用的最多的就是 starter。starter 可以理解为一个可拔插式的插件，例如，你想使用 JDBC 插件，那么可以使用 spring-boot-starter-jdbc；如果想使用 MongoDB，可以使用 spring-boot-starter-data-mongodb。
初学的同学可能会说：如果我要使用 MongoDB，我直接引入驱动 jar 包就行了，何必要引入 starter 包？starter 和普通 jar 包的区别在于，它能够实现自动配置，和 Spring Boot 无缝衔接，从而节省我们大量开发时间。

4.5 自动配置类原理

我们可以发现在 spring-boot-autoconfigure 中的 spring.factories 里面保存着 springboot 的默认提供的自动配置类。



让我们看在哪里去创建这些类的。我们可以关注@springbootApplication 注解，在 boot 启动类的 bean 定义被加载的会执行当前的注解。

```
@SpringBootConfiguration //次注解等同于@Configuration
@EnableAutoConfiguration //重要！！！启用自动配置注解，点进去深入理解
@ComponentScan(excludeFilters = {//排除过滤器本身
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {

    /**
     * Exclude specific auto-configuration classes such that they will never be applied.
     * @return the classes to exclude
     */
    @AliasFor(annotation = EnableAutoConfiguration.class)
    Class<?>[] exclude() default {};
```

进入到@EnableAutoConfiguration 注解

```
@AutoConfigurationPackage//设置自动配置的扫描包
@Import(AutoConfigurationImportSelector.class)//自动配置类的引入选择器
public @interface EnableAutoConfiguration {

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguratic

    /**
     * Exclude specific auto-configuration classes such that they will nev
     * @return the classes to exclude
     */
    Class<?>[] exclude() default {};

    /**
```

@AutoConfigurationImportSelector 是引入自动配置类的位置。

```
protected AutoConfigurationEntry getAutoConfigurationEntry(
      AutoConfigurationMetadata autoConfigurationMetadata,
      AnnotationMetadata annotationMetadata) {
   if (!isEnabled(annotationMetadata)) {
      return EMPTY_ENTRY;
   }
```

```
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    //获得所有的自动配置类
    List<String> configurations = getCandidateConfigurations(annotationMetadata,
          attributes);
    //排除重复
    configurations = removeDuplicates(configurations);
    //排除手动设置的重复
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    //移除排除的自动配置类
    configurations.removeAll(exclusions);
    //过滤掉没有引入的自动配置类
    configurations = filter(configurations, autoConfigurationMetadata);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
}
```

5 HttpEncodingAutoConfiguration

5.1HTTP 编码自动配置类概览

以 HttpEncodingAutoConfiguration（Http 编码自动配置）为例解释自动配置原理；

```
@Configuration //表示这是一个配置类，以前编写的配置文件一样，也可以给容器中添加组件
@EnableConfigurationProperties(HttpEncodingProperties.class) //启动指定类的
ConfigurationProperties 功能；将配置文件中对应的值和 HttpEncodingProperties 绑定起来；并把
      HttpEncodingProperties 加入到 ioc 容器中
@ConditionalOnWebApplication //Spring 底层@Conditional 注解（Spring 注解版），根据不同的条件，如果
满足指定的条件，整个配置类里面的配置就会生效； 判断当前应用是否是 web 应用，如果是，当前配置类生效
@ConditionalOnClass(CharacterEncodingFilter.class) //判断当前项目有没有这个类
CharacterEncodingFilter；SpringMVC 中进行乱码解决的过滤器；
@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing =
      true) //判断配置文件中是否存在某个配置 spring.http.encoding.enabled；如果不存在，判断也是成立的
//即使我们配置文件中不配置 pring.http.encoding.enabled=true，也是默认生效的；
public class HttpEncodingAutoConfiguration {
    //他已经和 SpringBoot 的配置文件映射了
    private final HttpEncodingProperties properties;
    //只有一个有参构造器的情况下，参数的值就会从容器中拿
    public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {
```

```
        this.properties = properties;
    }
    @Bean //给容器中添加一个组件，这个组件的某些值需要从 properties 中获取
    @ConditionalOnMissingBean(CharacterEncodingFilter.class) //判断容器没有这个组件？
    public CharacterEncodingFilter characterEncodingFilter() {
        CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
        filter.setEncoding(this.properties.getCharset().name());
        filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));
        filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));
        return filter;
    }
```

5.2条件判断

1. @Conditional 派生注解（Spring 注解版原生的@Conditional 作用）

   作用：必须是@Conditional 指定的条件成立，才给容器中添加组件，配置配里面的所有内容才生效；

@Conditional 扩展注解	作用（判断是否满足当前指定条件）
@ConditionalOnJava	系统的 java 版本是否符合要求
@ConditionalOnBean	容器中存在指定 Bean；
@ConditionalOnMissingBean	容器中不存在指定 Bean；
@ConditionalOnExpression	满足 SpEL 表达式指定
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissingClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的 Bean，或者这个 Bean 是首选 Bean
@ConditionalOnProperty	系统中指定的属性是否有指定的值
@ConditionalOnResource	类路径下是否存在指定资源文件
@ConditionalOnWebApplication	当前是 web 环境
@ConditionalOnNotWebApplication	当前不是 web 环境
@ConditionalOnJndi	JNDI 存在指定项

自动配置类必须在一定的条件下才能生效；
我们怎么知道哪些自动配置类生效；

```
@ConfigurationProperties(prefix = "spring.http")
public class HttpProperties {
```

```
   /**
    * Whether logging of (potentially sensitive) request details at DEBUG and TRACE
level
    * is allowed.
    */
   private boolean logRequestDetails;
```

精髓：
1）、SpringBoot 启动会加载大量的自动配置类
2）、我们看我们需要的功能有没有 SpringBoot 默认写好的自动配置类；
3）、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件有，我们就不需要再来配置了）
4）、给容器中自动配置类添加组件的时候，会从 properties 类中获取某些属性。我们就可以在配置文件中指定这

6 Springboot 数据元自动配置

6.1数据源自动管理

引入 jdbc 的依赖和 springboot 的应用场景

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

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

让我们使用 yaml 方式配置，创建 application.yaml
在默认情况下， 数据库连接可以使用 DataSource 池进行自动配置

- 默认 Hikari 可用， Springboot 将使用它。

我们可以自己指定数据源配置，通过 type 来选取使用哪种数据源

```yaml
spring:
  datasource:
    username: root
    password: root
    url: jdbc:mysql://localhost:3306/boot_demo
    driver-class-name: com.mysql.jdbc.Driver
    type: com.zaxxer.hikari.HikariDataSource
   # type: org.apache.commons.dbcp2.BasicDataSource
```

6.2 数据源自动配置原理

在数据源自动配置类里面我们可以看到默认支持的数据源类型

```java
@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ DataSourcePoolMetadataProvidersConfiguration.class,
        DataSourceInitializationConfiguration.class })
public class DataSourceAutoConfiguration {

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

    }

    @Configuration
    @Conditional(PooledDataSourceCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import({ DataSourceConfiguration.Hikari.class, DataSourceConfiguration.Tomcat.class,
            DataSourceConfiguration.Dbcp2.class, DataSourceConfiguration.Generic.class,
            DataSourceJmxConfiguration.class })
    protected static class PooledDataSourceConfiguration {

    }
```

我们可以看到三种数据源的配置

```java
/**
 * Hikari DataSource configuration.
 */
@ConditionalOnClass(HikariDataSource.class)
@ConditionalOnMissingBean(DataSource.class)
@ConditionalOnProperty(name = "spring.datasource.type", havingValue = "com.zaxxer.hikari.HikariDataSource", matchIfMissing = true)
static class Hikari {

    @Bean
    @ConfigurationProperties(prefix = "spring.datasource.hikari")
    public HikariDataSource dataSource(DataSourceProperties properties) {
        HikariDataSource dataSource = createDataSource(properties,
                HikariDataSource.class);
        if (StringUtils.hasText(properties.getName())) {
            dataSource.setPoolName(properties.getName());
        }
        return dataSource;
    }

}
```

点开 starter-jdbc 我们可以看到 Hikari 是默认的数据源

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

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
</dependency>
```

6.3 配置 druid 数据源

引入 druid 的依赖

```
<dependency>
   <groupId>com.alibaba</groupId>
   <artifactId>druid</artifactId>
   <version>1.0.9</version>
</dependency>
<dependency>
   <groupId>log4j</groupId>
   <artifactId>log4j</artifactId>
   <version>1.2.15</version>
</dependency>
```

修改

spring.datasource.type=com.alibaba.druid.pool.DruidDataSource

在 application.yaml 中加入

```
spring:
  datasource:
    username: root
    password: root
    url: jdbc:mysql://localhost:3306/boot_demo
    driver-class-name: com.mysql.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
```

```
    poolPreparedStatements: true
    filters: stat,wall,log4j
    maxPoolPreparedStatementPerConnectionSize: 20
    useGlobalDataSourceStat: true
    connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
```

创建数据源注册类

```
@Configuration
public class DruidConfig {

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

7 jdbcTemplate 自动配置

在数据源建表

```
SET FOREIGN_KEY_CHECKS=0;

-- ----------------------------
-- Table structure for tx_user
-- ----------------------------
DROP TABLE IF EXISTS `tx_user`;
CREATE TABLE `tx_user` (
  `username` varchar(10) DEFAULT NULL,
  `userId` int(10) NOT NULL,
  `password` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`userId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

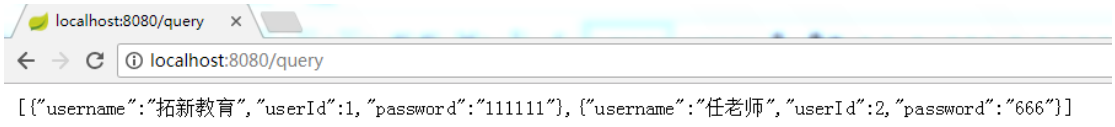
userId	username	password
1	拓新教育	666
2	任老师	666

创建 Controller

```
@Controller
public class TestController {



    @Autowired
    JdbcTemplate jdbcTemplate;


    @ResponseBody
    @RequestMapping("/query")
    public List<Map<String, Object>> query(){
        List<Map<String, Object>> maps = jdbcTemplate.queryForList("SELECT * FROM
tx_user");
        return maps;
    }


}
```

启动 springboot 访问

http://localhost:8080/query



Springboot 中提供了 JdbcTemplateAutoConfiguration 的自动配置

```
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
```

JdbcTemplateAutoConfiguration 源码：

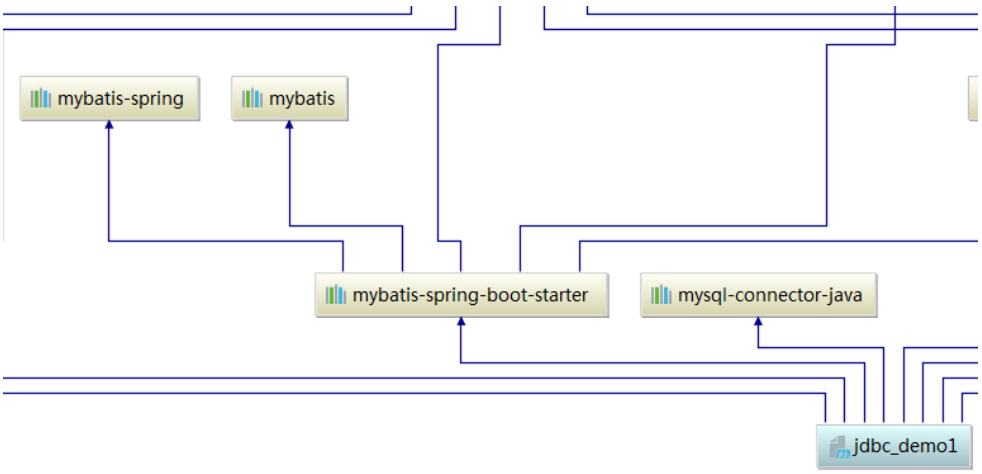
```
@Configuration
static class JdbcTemplateConfiguration {
    private final DataSource dataSource;
    private final JdbcProperties properties;

    JdbcTemplateConfiguration(DataSource dataSource, JdbcProperties properties) {
        this.dataSource = dataSource;
        this.properties = properties;
    }

    @Bean
    @Primary
    @ConditionalOnMissingBean({JdbcOperations.class})
    public JdbcTemplate jdbcTemplate() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(this.dataSource);
        Template template = this.properties.getTemplate();
        jdbcTemplate.setFetchSize(template.getFetchSize());
        jdbcTemplate.setMaxRows(template.getMaxRows());
        if (template.getQueryTimeout() != null) {
            jdbcTemplate.setQueryTimeout((int)template.getQueryTimeout().getSeconds());
        }

        return jdbcTemplate;
    }
}
```

注意：url后面加上时区useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC

8 mybatis 自动配置

8.1 Springboot 整合 mybatis 注解版

```
<dependency>
   <groupId>org.mybatis.spring.boot</groupId>
   <artifactId>mybatis-spring-boot-starter</artifactId>
   <version>1.3.1</version>
</dependency>
```



步骤：

1）、配置数据源相关属性（见上一节Druid）

2）、给数据库建表

3）、创建 JavaBean

```
public class TxPerson {


    private int pid;
```

```
    private String pname;

    private String addr;

    private int gender;

    private Date birth;
```

4）创建 Mapper

```
@Mapper
public interface TxPersonMapper {



    @Select("select * from tx_person")
    public List<TxPerson> getPersons();



    @Select("select * from tx_person t where t.pid = #{id}")
    public TxPerson getPersonById(int id);

    @Options(useGeneratedKeys =true, keyProperty = "pid")
    @Insert("insert into tx_person(pid, pname, addr,gender, birth)" +
           " values(#{pid}, #{pname}, #{addr},#{gender}, #{birth})")
    public void insert(TxPerson person);

    @Delete("delete from tx_person where pid = #{id}")
    public void update(int id);

}
```

单元测试

```
@Autowired
TxPersonMapper txPersonMapper;

@Test
public void contextLoads() throws SQLException {

    DataSource bean = (DataSource) context.getBean( s: "dataSource");
    System.out.println(bean);
}

@Test
public void testMybatis() throws SQLException {
    TxPerson p = txPersonMapper.getPersonById(1);
    System.out.println(p);

}
```

解决驼峰模式和数据库中下划线不能映射的问题。

pid	pname	p_addr	gender	birth
1	张三	北京	1	2018-06-14

```
public class TxPerson {

    private int pid;

    private String pname;

    private String pAddr;

    private int gender;

    private Date birth;
```

```
@Configuration
public class MybatisConfig {

    @Bean
    public ConfigurationCustomizer getCustomizer(){
        return new ConfigurationCustomizer() {
            @Override
            public void customize(org.apache.ibatis.session.Configuration configuration) {
                configuration.setMapUnderscoreToCamelCase(true);
            }
        };
    }
}
```

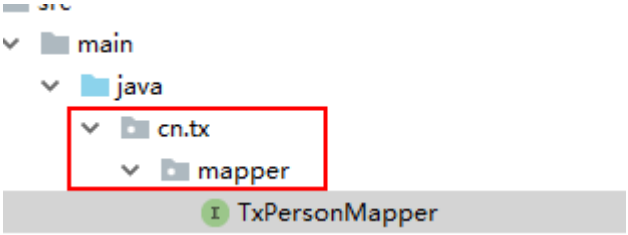
查询结果

```
TxPerson{pid=1, pname='张三', pAddr='北京', gender=1, birth=Thu Jun 14 00:00:00 CST 2018}
```

我们同样可以在 mybatis 的接口上不加@Mapper 注解，通过扫描器注解来扫描

```
@MapperScan("cn.tx.mapper")
@Configuration
public class MybatisConfig {

    @Bean
    public ConfigurationCustomizer getCustomizer(){
        return new ConfigurationCustomizer() {
            @Override
            public void customize(org.apache.ibatis.session.Configuration
                configuration.setMapUnderscoreToCamelCase(true);
            }
        };
    }
}
```

Mapper 接口存放在 cn.tx.mapper 下



8.2 Springboot 整合 mybatis 配置文件

创建 sqlMapConfig.xml 配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>
</configuration>
```

创建映射文件 PersonMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
        <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper
3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="cn.tx.mapper.TxPersonMapper">
    <select id="getPersons" resultType="TxPerson">
        select * from tx_person
```

```
    </select>
</mapper>
```

在 application.yaml 中配置 mybatis 的信息

```
mybatis:
  mapper-locations: classpath:mybatis/mapper/*.xml
  type-aliases-package: cn.tx.springboot.jdbc_demo1
```

@ConditionalOnSingleCandidate 类似于@ConditionalOnBean

但只有在确定了给定 bean 类的单个候选项时才会加载 bean。

@ConditionalOnMissingBean 当给定的 Bean 不存在时返回 true

各类型间是 or 的关系

@ConditionalOnBean 与上面相反，要求 bean 存在

@ConditionalOnMissingClass 当给定的类名在类路径上不存在时返回 true

各类型间是 and 的关系

@ConditionalOnClass 与上面相反，要求类存在

@ConditionalOnExpression Spel 表达式成立，返回 true

@ConditionalOnJava 运行时的 Java 版本号包含给定的版本号，返回 true

@ConditionalOnProperty 属性匹配条件满足，返回 true

@ConditionalOnWebApplication web 环境存在时，返回 true

@ConditionalOnNotWebApplication web 环境不存在时，返回 true

@ConditionalOnResource 指定的资源存在时，返回 true