

6. SpringBoot 配置文件

本节目标

1. 学习SpringBoot配置文件的格式以及对应的语法
2. 了解两个配置文件格式的差异

1. 配置文件作用

计算机上有数以千计的配置文件, 我们使用的绝大多数软件, 比如浏览器, 微信, Idea, 甚至电脑, 手机, 都离不开配置文件. 我们可能永远不会直接与其中的大部分文件打交道, 但它们确实以不同的形式散落在我们的计算机上, 比如C:\Users, C:\Windows文件夹, 以及各种 *.config, *.xml 文件

配置文件主要是为了解决硬编码带来的问题, 把可能会发生改变的信息, 放在一个集中的地方, 当我们启动某个程序时, 应用程序从配置文件中读取数据, 并加载运行.

硬编码是将数据直接嵌入到程序或其他可执行对象的源代码中, 也就是我们常说的"代码写死".

比如手机字体大小

如果采用硬编码的方式, 就直接在程序中指定字体大小, 所有的用户使用的都是同一个字体大小

但是不同的用户有不同的偏好, 我们可以把手机字体的大小放在配置文件中, 当程序启动时, 读取配置, 以用户设置的字体大小来显示.

使用配置文件, 可以使程序完成用户和应用程序的交互, 或者应用程序与其他应用程序的交互

SpringBoot配置文件

SpringBoot支持并定义了配置文件的格式, 也在另一个层面达到了规范其他框架集成到SpringBoot的目的.

很多项目或者框架的配置信息也放在配置文件中, 比如:

- 项目的启动端口
- 数据库的连接信息(包含用户名和密码的设置)
- 第三方系统的调用密钥等信息
- 用于发现和定位问题的普通日志和异常日志等.

项目的启动端口

SpringBoot内置了Tomcat服务器, 默认端口号是8080, 但是用户电脑上8080端口号有可能就被其他应用程序占用了, 所以SpringBoot需要支持让用户自定义端口号

数据库连接信息

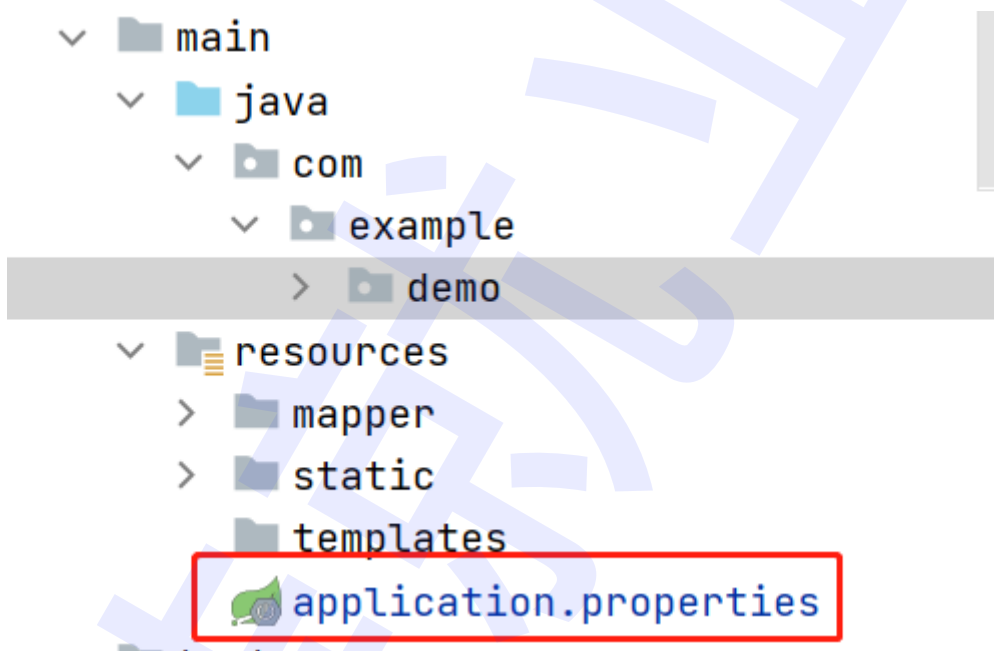
为了方便简单的访问数据库, 出现了一些持久层框架, 其实就是对JDBC进行了更深层次的封装. 让用户通过简单几行代码就可完成数据库的访问. 但是不同的应用程序访问的数据库不同, 这些持久层框架就需要支持用户可以自定义配置数据库的连接信息.

2. 配置文件快速入手

我们在前面讲了Tomcat 默认端口号是8080, 所以我们程序访问时的端口号也是8080

但是如果8080端口号已经被其他进程使用了呢? 我们可以通过配置文件来修改服务的端口号

SpringBoot 在创建项目时, 就已经帮我们创建了配置文件



修改 application.properties 文件

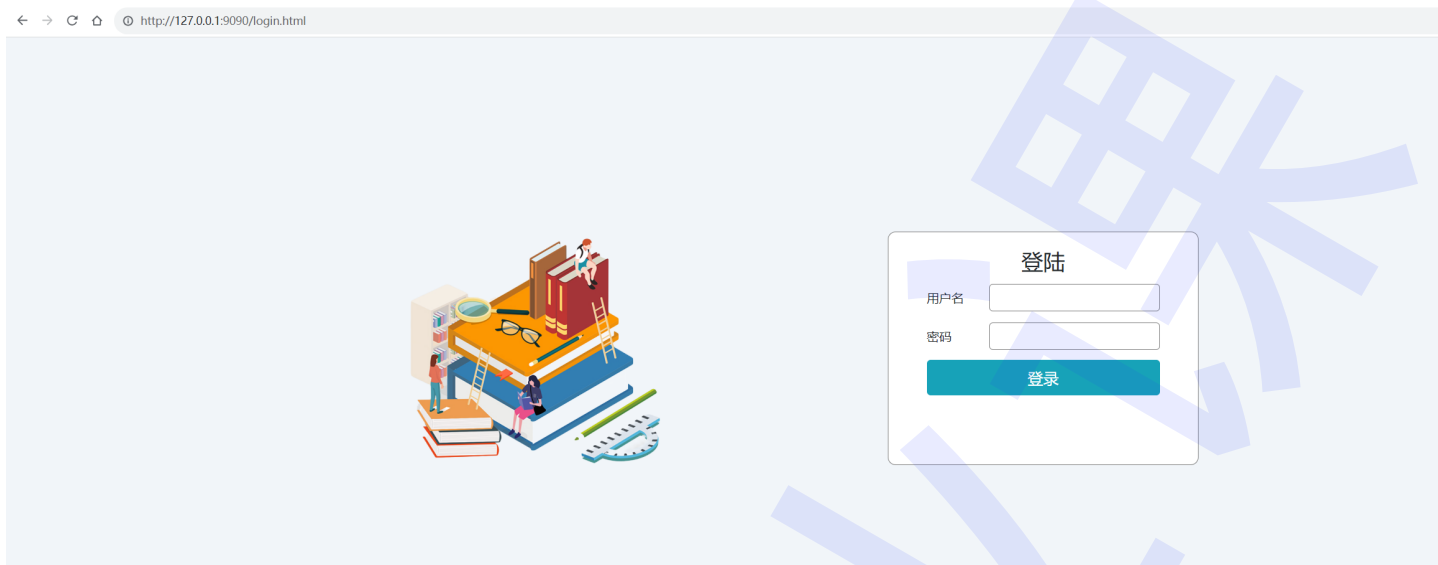
```
1 server.port=9090
```

重新运行程序, 观察日志

```
17:46:01.240 INFO 26068 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 9090 (http)
17:46:01.251 INFO 26068 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
17:46:01.251 INFO 26068 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.79]
17:46:01.403 INFO 26068 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
17:46:01.403 INFO 26068 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1545 ms
17:46:02.712 INFO 26068 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9090 (http) with context path ''
17:46:02.724 INFO 26068 --- [main] com.example.demo.DemoApplication : Started DemoApplication in 3.247 seconds (JVM running for 3.632)
```

显示Tomcat启动端口号为9090

访问程序: <http://127.0.0.1:9090/login.html>



此时: <http://127.0.0.1:808/login.html> 就不能再访问了

3. 配置文件的格式

Spring Boot 配置文件有以下三种:

- application.properties
- application.yml
- application.yaml

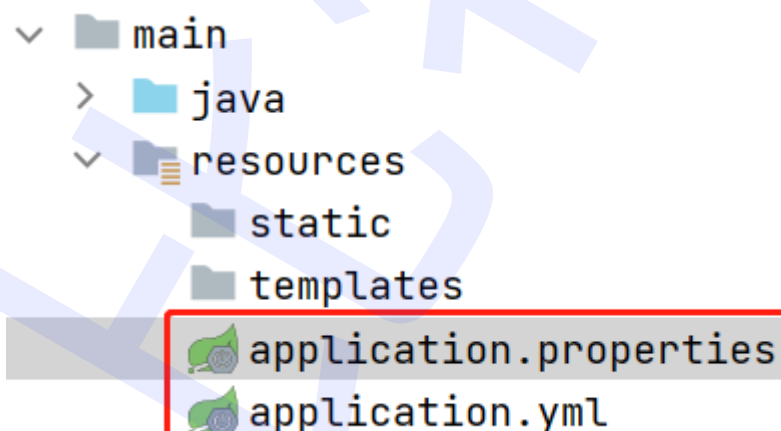
yml 为yaml的简写, 实际开发中出现频率最高. **yaml 和 yml 的使用方式一样, 课堂中只讲 yml 文件的使用**

当应用程序启动时, Spring Boot 会自动从 classpath 路径找到并加载

`application.properties` 和 `application.yaml` 或者 `application.yml` 文件.

也可以通过 `spring.config.name` 指定文件路径和名称, 参考 <https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config.files>

如下图所示:



类似商品的包装一样, 有新老两款包装. properties 类型的配置文件就属于老款包装, 也是创建 Spring Boot 项目时默认的文件格式 (主要是因为仓库里还有库存), 而 yaml 属于新版包装, 如果用户了解情况直接指定要新款包装, 那么就直接发给他

特殊说明

1. 理论上讲 `.properties` 和 `.yaml` 可以并存在于一个项目中, 当 `.properties` 和 `.yaml` 并存时, 两个配置都会加载. 如果配置文件内容有冲突, 则以 `.properties` 为主, 也就是 `.properties` 优先级更高.
2. 虽然理论上来讲 `.properties` 可以和 `.yaml` 共存, 但实际的业务当中, 我们通常会采取一种统一的配置文件格式, 这样可以更好的维护(降低故障率).

4. properties 配置文件说明

properties 配置文件是最早期的配置文件格式, 也是创建 SpringBoot 项目默认的配置文件

4.1 properties 基本语法

properties 是以键值的形式配置的, key 和 value 之间是以 "=" 连接的, 如:

```
1 # 配置项目端口号
2 server.port=8080
3 #配置数据库连接信息
4 spring.datasource.url=jdbc:mysql://127.0.0.1:3306/testdb?
  characterEncoding=utf8&useSSL=false
5 spring.datasource.username=root
6 spring.datasource.password=root
```

PS: 小技巧: 配置文件中使用 “#” 来添加注释信息。

咱们当前先学习语法和使用, 更多配置信息随着课堂练习再学习. 感兴趣的也可以参考官网: [Spring Boot配置](#)

4.2 读取配置文件

如果在项目中, 想要主动的读取配置文件中的内容, 可以使用 `@Value` 注解来实现。

`@Value` 注解使用 " `${}` " 的格式读取, 如下代码所示:

properties 配置如下:

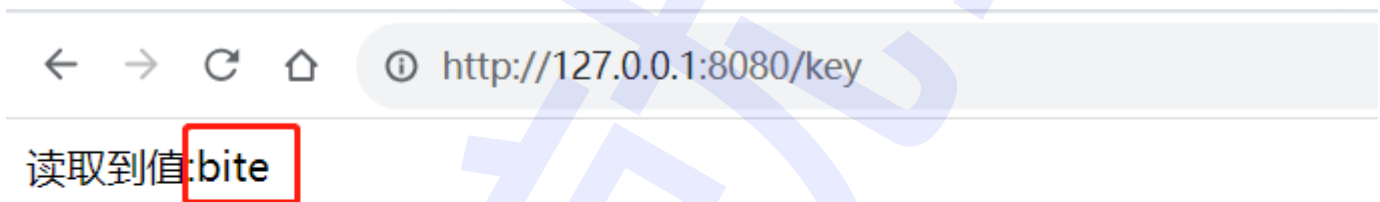
```
1 mykey.key1 = bite
```

```

1 import org.springframework.beans.factory.annotation.Value;
2 import org.springframework.web.bind.annotation.RequestMapping;
3 import org.springframework.web.bind.annotation.RestController;
4
5 @RestController
6 public class PropertiesController {
7     @Value("${mykey.key1}")
8     private String key1;
9
10    @RequestMapping("/key")
11    public String key(){
12        return "读取到值:"+key1;
13    }
14 }

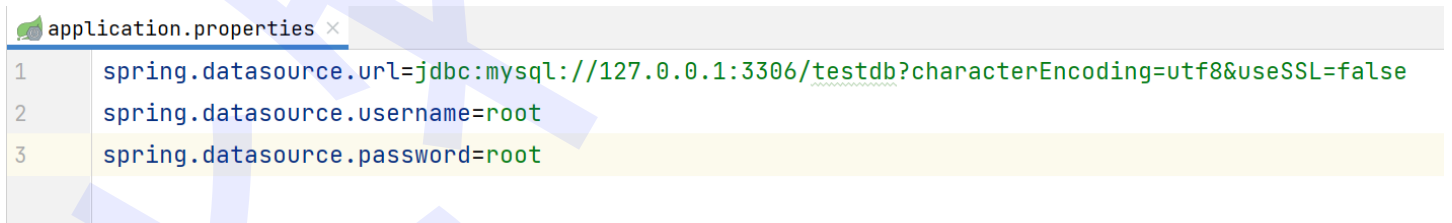
```

最终执行效果：

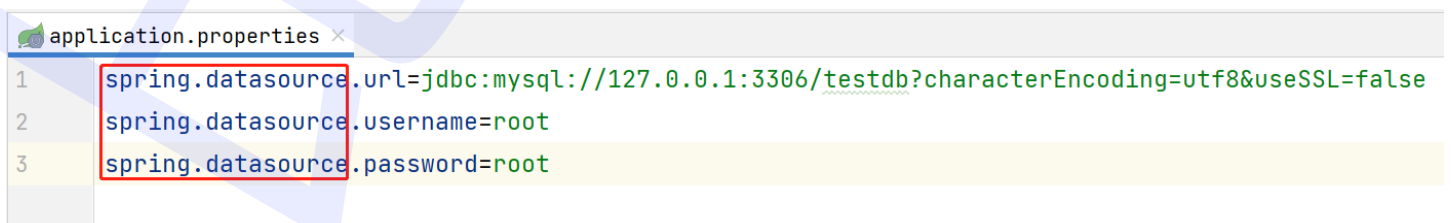


4.3 properties 缺点分析

properties 配置是以 key-value 的形式配置的，如下图所示：



从上述配置key看出，properties 配置文件中会有很多的冗余的信息，比如这些：



想要解决这个问题，就可以使用 yml 配置文件的格式化了。

5. yml 配置文件说明

yml 是 YAML 是缩写，它的全称 Yet Another Markup Language 翻译成中文就是“另一种标记语言。

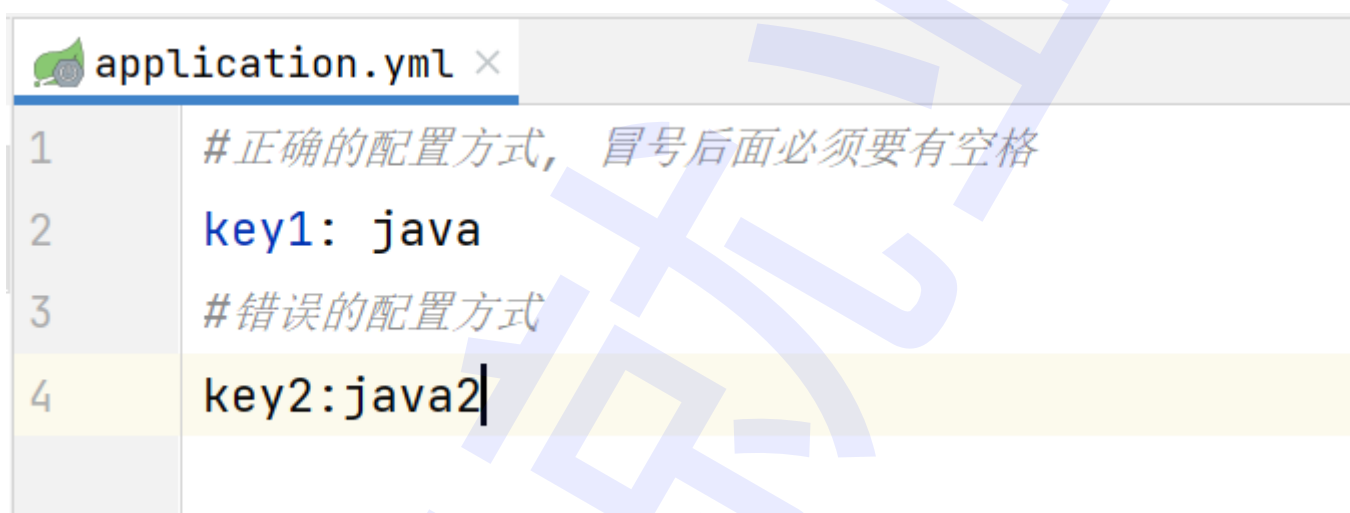
我们先来学习 yml 的语法

5.1 yml 基本语法

yml 是树形结构的配置文件，它的基础语法是 "key: value"。

key 和 value 之间使用英文冒号加空格的方式组成，**空格不可省略**

基础语法如下：



```
1 #正确的配置方式，冒号后面必须要有空格
2 key1: java
3 #错误的配置方式
4 key2:java2
```

第一项的配置为正确的，key 也是高亮显示的。第二项没有空格是错误的 Usage，第二项的 key 也没有高亮显示

使用 yml 连接数据库

yml 使用示例：

```
1 spring:
2   datasource:
3     url: jdbc:mysql://127.0.0.0:3306/dbname?characterEncoding=utf8&useSSL=false
4     username: root
5     password: root
```

yml 和 properties 连接数据库的配置对比

```
application.yml x application.properties x
1 spring.datasource.url=jdbc:mysql://127.0.0.1:3306/testdb?characterEncoding=utf8&useSSL=false
2 spring.datasource.username=root
3 spring.datasource.password=root
```

```
application.yml x application.properties x
1 spring:
2   datasource:
3     url: jdbc:mysql://127.0.0.0:3306/dbname?characterEncoding=utf8&useSSL=false
4     username: root
5     password: root
```

5.2 yml 使用进阶

5.2.1 yml 配置不同类型及 null

```
1 # 字符串
2 string.value: Hello
3
4 # 布尔值, true或false
5 boolean.value: true
6 boolean.value1: false
7
8 # 整数
9 int.value: 10
10
11 # 浮点数
12 float.value: 3.14159
13
14 # Null, ~代表null
15 null.value: ~
16
17 # "" 空字符串
18 #, 直接后面什么都不加就可以了, 但这种方式不直观, 更多的表示是使用引号括起来
19 empty.value: ''
20
```

5.2.1.1 yml 配置读取

yaml 读取配置的方式和 properties 相同, 使用 @Value 注解即可, 实现代码如下:

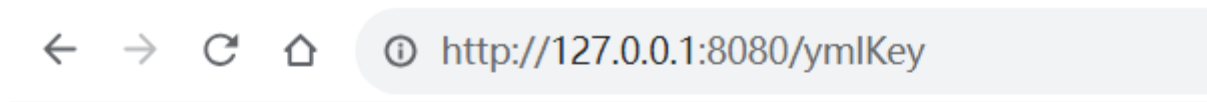
yaml配置:

```
1 string:
2   hello: bite
```

```
1 @RestController
2 public class ReadYml {
3     @Value("${string.hello}")
4     private String hello;
5
6     @RequestMapping("/yamlKey")
7     public String key(){
8         return "读取到值:"+hello;
9     }
10 }
```

访问: <http://127.0.0.1:8080/ymlKey>

运行结果:



← → ↻ 🏠 ⓘ http://127.0.0.1:8080/ymlKey

读取到值:bite

5.2.1.2 注意事项：value 值加单双引号

字符串默认不用加上单引号或者双引号，如果加英文的单双引号可以表示特殊的含义。
尝试在 application.yml 中配置如下信息：

```
1 string:
2   str1: Hello \n Spring Boot.
3   str2: 'Hello \n Spring Boot.'
4   str3: "Hello \n Spring Boot."
```

读取程序实现代码如下：

```
1 @RestController
2 public class ReadYml {
3     @Value("${string.str1}")
4     private String str1;
```



```

5     @Value("${string.str2}")
6     private String str2;
7     @Value("${string.str3}")
8     private String str3;
9
10    @RequestMapping("/yaml")
11    public String readYml(){
12        System.out.println(str1);
13        System.out.println(str2);
14        System.out.println(str3);
15        return "yaml";
16    }
17 }

```

以上程序的执行结果如下图所示：

```

Hello \n Spring Boot.
Hello \n Spring Boot.
Hello
    Spring Boot.

```

从上述结果可以看出：

- 字符串默认不用加上单引号或者双引号。
- 单引号会转义特殊字符，使其失去特殊功能，始终是一个普通的字符串。
- 双引号不会转义字符串里面的特殊字符，特殊字符会表示本身的含义。

此处的转义理解起来会有些拗口，\n 本意表示的是换行

使用单引号会转义，就是说，\n 不再表示换行了，而是表示一个普通的字符串

使用双引号不会转义，表示\n 表示的是它本身的含义，就是换行

JavaEE的学习重在理解和使用，不纠结概念和描述

5.2.2 配置对象

我们还可以在 yml 中配置对象，如下配置：

```

1 student:
2   id: 1
3   name: Java

```

```
4     age: 18
```

或者是使用行内写法（与上面的写法作用一致）：

```
1 student: {id: 1,name: Java,age: 18}
```

这个时候就不能用 `@Value` 来读取配置中的对象了，此时要使用另一个注解

`@ConfigurationProperties` 来读取，具体实现如下：

```
1 import lombok.Data;
2 import org.springframework.boot.context.properties.ConfigurationProperties;
3 import org.springframework.stereotype.Component;
4
5 @ConfigurationProperties(prefix = "student")
6 @Component
7 @Data
8 public class Student {
9     private int id;
10    private String name;
11    private int age;
12 }
```

调用类的实现如下：

```
1 @RestController
2 public class StudentController {
3     @Autowired
4     private Student student;
5
6     @RequestMapping("/readStudent")
7     public String readStudent(){
8         return student.toString();
9     }
10 }
```

访问 <http://127.0.0.1:8080/readStudent> 运行结果如下：

Student(id=1, name=Java, age=18)

5.2.3 配置集合

配置文件也可以配置 list 集合，如下所示：

```
1 dbtypes:
2   name:
3     - mysql
4     - sqlserver
5     - db2
```

集合的读取和对象一样，也是使用 `@ConfigurationProperties` 来读取的，具体实现如下：

```
1 @Component
2 @ConfigurationProperties("dbtypes")
3 @Data
4 public class ListConfig {
5     private List<String> name;
6 }
```

访问集合的实现如下：

```
1 @RestController
2 public class ReadYml2 {
3     @Autowired
4     private ListConfig listConfig;
5
6     @RequestMapping("/readList")
7     public String readList(){
8         return listConfig.toString();
9     }
10 }
```

5.2.4 配置Map

配置文件也可以配置 map，如下所示：

```
1 maptypes:
2   map:
3     k1: kk1
4     k2: kk2
5     k3: kk3
```

或者是使用行内写法(与上面的写法作用一致):

```
1 maptypes: {map: {k1: kk1,k2: kk2, k3: kk3}}
```

Map的读取和对象一样，也是使用 `@ConfigurationProperties` 来读取的，具体实现如下：

```
1 @Component
2 @ConfigurationProperties("maptypes")
3 @Data
4 public class MapConfig {
5     private HashMap<String,String> map;
6 }
```

打印类的实现如下：

```
1 @RestController
2 public class ReadYml2 {
3     @Autowired
4     private MapConfig mapConfig;
5
6     @RequestMapping("/readMap")
7     public String readStudent(){
8         return mapConfig.toString();
9     }
10 }
```

5.3 yml优缺点

优点:

1. 可读性高，写法简单，易于理解
2. 支持更多的数据类型，可以简单表达对象，数组，List，Map等数据形态。

3. 支持更多的编程语言, 不止是Java中可以使用, 在Golang, Python, Ruby, JavaScript中也可以使用

缺点:

1. 不适合写复杂的配置文件

比如properties格式如下

```
1 keycloak.realm = demo
2 keycloak.resource = fm-cache-cloud
3 keycloak.credentials.secret = d4589683-0ce7-4982-bcd3
4 keycloak.security[0].authRoles[0]= user
5 keycloak.security[0].collections[0].name = ssologinurl
6 keycloak.security[0].collections[0].patterns[0] = /login/*
7
```

转换为yaml

```
1 keycloak:
2   realm: demo
3   resource: fm-cache-cloud
4   credentials:
5     secret: d4589683-0ce7-4982-bcd3
6   security:
7     - authRoles:
8       - user
9     collections:
10      - name: ssologinurl
11        patterns:
12          - /login/*
```

转换的过程也比较花费精力, 如果配置更复杂一点, 可读性会更差, 代码也会更难写

2. 对格式有较强的要求(一个空格可能会引起一场血案)

6. 综合性练习

6.1 验证码案例

随着安全性的要求越来越高, 目前项目中很多都使用了验证码, 验证码的形式也是多种多样, 更复杂的图形验证码和行为验证码已经成为了更流行的趋势.



验证码的实现方式很多,可以前端实现,也可以后端实现.网上也有比较多的插件或者工具包可以使用,咱们选择使用Hutool提供的小工具来实现

6.1.1 需求

界面如下图所示

1. 页面生成验证码
2. 输入验证码, 点击提交, 验证用户输入验证码是否正确, 正确则进行页面跳转

输入验证码

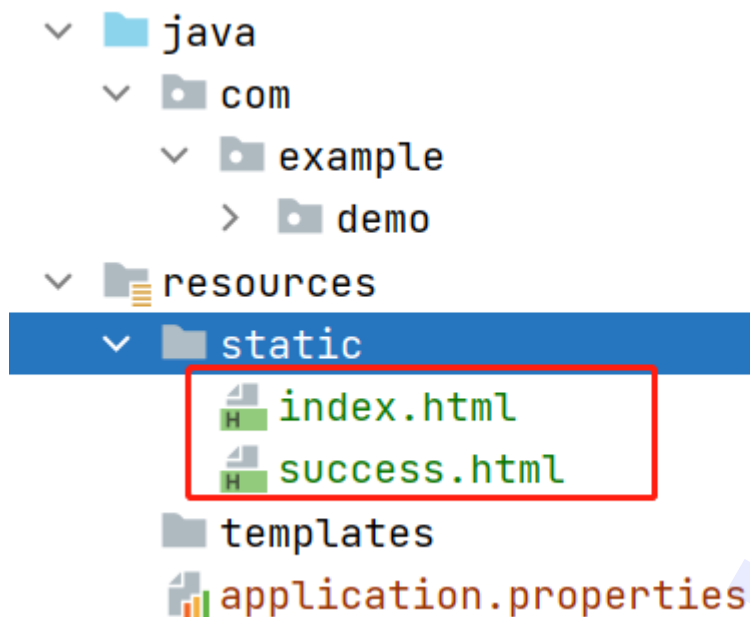


提交

6.1.2 准备工作

创建项目, 引入SpringMVC的依赖包, 把前端页面放在项目中(课件中提供)

码云地址: [JavaEE进阶课程资料包](#)



6.1.3 约定前后端交互接口

需求分析

后端需要提供两个服务

1. 生成验证码, 并返回验证码
2. 校验验证码是否正确: 校验验证码是否正确.

接口定义

1. 生成验证码

请求:

1 请求URL: `/captcha/getCaptcha`

响应: 验证码图片内容

浏览器给服务器发送一个 `/captcha/getCaptcha` 这样的请求, 服务器返回一个图片, 浏览器显示在页面上

2. 校验验证码是否正确

请求: `/captcha/check`

- 1 请求URL: `/captcha/check`
- 2
- 3 请求参数: `captcha=xn8d`

captcha : 用户输入的验证码

响应:

```
1 true
```

根据用户输入的验证码, 校验验证码是否正确. true: 验证成功. false: 验证失败.

6.1.4 Hutool工具介绍

咱们课程中验证码的实现, 使用Hutool提供的小工具来实现

Hutool是一个Java工具包类库, 对文件、流、加密解密、转码、正则、线程、XML等JDK方法进行封装, 组成各种Util工具类.

Hutool是一个小而全的Java工具类库, 通过静态方法封装, 降低相关API的学习成本, 提高工作效率, 使Java拥有函数式语言般的优雅, 让Java语言也可以"甜甜的".



Hutool官网: <https://hutool.cn/>

Hutool参考文档: <https://hutool.cn/docs/#/>

Hutool源码: <https://github.com/dromara/hutool>

6.1.5 实现服务器端代码

1. 引入依赖

```
1 <dependency>
```



```
2 <groupId>cn.hutool</groupId>
3 <artifactId>hutool-captcha</artifactId>
4 <version>5.8.22</version>
5 </dependency>
```

2. 实现验证码

根据API生成验证码, 并进行测试

```
1
2 @RequestMapping("/captcha")
3 @RestController
4 public class HuToolCaptchaController {
5
6     @RequestMapping("/getCaptcha")
7     public void getCaptcha(HttpServletResponse response){
8         //定义图形验证码的长和宽
9         LineCaptcha lineCaptcha = CaptchaUtil.createLineCaptcha(200,
10         100,4,150);
11
12         //图形验证码写出, 可以写出到文件, 也可以写出到流
13         try {
14             lineCaptcha.write(response.getOutputStream());
15             //输出code
16             System.out.println("生成的验证码:"+lineCaptcha.getCode());
17         } catch (IOException e) {
18             throw new RuntimeException(e);
19         }
20 }
```

对程序进行调整

- 1) 把配置项挪到配置文件中
- 2) 把生成的验证码存储在Session中, 校验时使用

配置项

```
1 captcha:
2   width: 100
3   height: 40
4   session:
5     key: CAPTCHA_SESSION_KEY
6     date: KAPTCHA_SESSION_DATE
```

验证码配置项对应的Java对象

```
1 @Data
2 @Component
3 @ConfigurationProperties(prefix = "captcha")
4 public class CaptchaProperties {
5     private Integer width;
6     private Integer height;
7     private Session session;
8
9     @Data
10    public static class Session {
11        private String key;
12        private String date;
13    }
14 }
```

调整Controller代码

```
1 @RequestMapping("/captcha")
2 @RestController
3 public class HuToolCaptchaController {
4     @Autowired
5     private CaptchaProperties captchaProperties;
6
7     @RequestMapping("/getCaptcha")
8     public void getCode(HttpSession session, HttpServletResponse response) {
9         //定义图形验证码的长和宽
10        LineCaptcha lineCaptcha =
11            CaptchaUtil.createLineCaptcha(captchaProperties.getWidth(),
12                captchaProperties.getHeight());
13        response.setContentType("image/jpeg");
14        //禁止使用缓存
15        response.setHeader("Pragma", "No-cache");
16        try {
17            // 输出到页面
18            lineCaptcha.write(response.getOutputStream());
19            //存储在Session中
20            session.setAttribute(captchaProperties.getSession().getKey(),
21                lineCaptcha.getCode());
22            session.setAttribute(captchaProperties.getSession().getDate(), new
23                Date());
24            // 打印日志
```

```

22         System.out.println("生成的验证码:"+lineCaptcha.getCode());
23         // 关闭流
24         response.getOutputStream().close();
25     } catch (IOException e) {
26         e.printStackTrace();
27     }
28 }

```

启动项目, 访问<http://127.0.0.1:8080/captcha/getCaptcha>, 显示验证码



3. 校验验证码

```

1 private final static long VALID_MILLIS_TIME = 60 * 1000;
2
3 @RequestMapping("/check")
4 public boolean checkHomeCaptcha(String captcha, HttpSession session) {
5
6     if (!StringUtils.hasLength(captcha)) {
7         return false;
8     }
9     String savedCaptcha = (String)
10 session.getAttribute(captchaProperties.getSession().getKey());
11     Date sessionDate = (Date)
12 session.getAttribute(captchaProperties.getSession().getDate());
13     if (captcha.equalsIgnoreCase(savedCaptcha)) {
14         if (sessionDate == null
15             || System.currentTimeMillis() - sessionDate.getTime() <
16             VALID_MILLIS_TIME) {
17             return true;
18         }
19         return false;
20     }
21     return false;
22 }

```

比对Session中存储的验证码是否和用户输入的一致

如果一致,并且时间在一分钟以为就认为成功

6.1.6 调整前端页面代码

修改 index.html

1. 补充ajax代码, 点击提交按钮, 发送请求去服务端进行校验

```
1 $("#checkCaptcha").click(function () {
2     $.ajax({
3         url: "/captcha/check",
4         type: "post",
5         data: { captcha: $("#inputCaptcha").val() },
6         success: function (result) {
7             if (result) {
8                 location.href = "success.html";
9             } else {
10                alert("验证码错误");
11                $("#inputCaptcha").val("");
12            }
13        }
14    });
15 });
```

6.1.7 运行测试

通过 URL <http://127.0.0.1:8080/index.html> 访问服务

输入验证码



输入验证码, 验证成功

← → ↻ 🏠 ⓘ <http://127.0.0.1:8080/success.html>

验证成功

7. 总结

1. properties 是以 key=value 的形式配置的键值类型的配置文件, yaml 使用的是树形配置方式.
2. 读取配置文件内容, 使用 `@Value` 注解, 注解内使用 "`${}`" 的格式读取.
3. yaml 层级之间使用换行缩进的方式配置, key 和 value 之间使用 ":" (英文冒号) 加空格的方式设置, 并且空格不可省略.
4. properties 为早期并且默认的配置文件格式, 其配置存在一定的冗余数据, 使用 yaml 可以很好的解决数据冗余的问题, 但不适合复杂配置.
5. yaml 可以和 properties 共存, 但一个项目中建议使用一种配置类型文件.