红岩上课

目录

正则表达式

> 正则表达式是用于进行文本匹配的工具

一些例子：

> Hello

---匹配“Hello”

> \bHello\b.*\bWorld\b '\b'表示一个单词边界，'.' 匹配除换行符 \n 之外单字符, '*'匹配0次或1次

---匹配“Hello world”

非打印字符

字符	描述
\cx	匹配由x指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。注意 Unicode 正则表达式会匹配全角空格符。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。

- 字符转义要使用`\`获取字符的本身意义
-

特殊字符

特别字符	描述
$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 \$。
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单字符。要匹配 . ，请使用 \. 。
[	标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 {，请使用 \{。
|	指明两项之间的一个选择。要匹配 |，请使用 \\|。

- 反义：`<div[^>]+>`表明已div开头的标签（最大可能包）
- 分枝条件：0\d{2}-\d{8}|0\d{3}-\d{7} 匹配两种以连字号分隔的电话号码（010-12345678或0376-2233445）

限定符

字符	描述
*	匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。* 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 、 "does" 中的 "does" 、 "doxy" 中的 "do" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

> 贪婪：尽可能多的匹配字符 如`*` `{m,n}`

针对"axxxaxxxa"

例：`(a)(\w+)` :"\w+" 将匹配第一个 "a" 之后的所有字符 "xxxaxxxa"

例：`(a)(\w+)(a)` "\w+" 将匹配第一个 "a" 和最后一个 "a" 之间的所有字符 "xxxdxxx"。虽然 "\w+" 也能够匹配上最后一个 "a"，但是为了使整个表达式匹配成功，"\w+" 可以 "让出" 它本来能够匹配的最后一个 "a"

> 懒惰：尽可能少的匹配字符

例：`(a)(\w+?)`： "\w+?" 将尽可能少的匹配第一个 "a" 之后的字符，结果是："\w+?" 只匹配了一个 "x"

例：`(a)(\w+?)(a)` 为了让整个表达式匹配成功，"\w+?" 不得不匹配 "xxx" 才可以让后边的 "a" 匹配，从而使整个表达式匹配成功。因此，结果是："\w+?" 匹配 "xxx"：

定位符

字符	描述
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与 \n 或 \r 之前的位置匹配。
\b	匹配一个单词边界，即字与空格间的位置。
\B	非单词边界匹配。

选择（分组）

小括号()、中括号[]、大括号的区别?

> 小括号()：匹配小括号内的字符串,可以是一个，也可以是多个，是多选结构的

> 中括号[]：匹配字符组内的字符，在[]内的字符都是字符，不是元字符

例[^A-F0-3] 匹配 "A"~"F","0"~"3" 之外的任意一个字符

> 大括号{}：匹配次数，匹配在它之前表达式匹配出来的元素出现的次数

- 小括号会缓存匹配
- 使用非捕获元`?:` `?=` `?!` 消除缓存
- (?:)匹配不存储

```
(?<= (href=")).{1,200}(?=(">))
//匹配以(href=")开头的字符串，并且捕获(存储)到分组中
//匹配以(">)结尾的字符串，并且捕获(存储)到分组中
(?<=(?:href=")).{1,200}(?=(?:">))
//不捕获分组
```

- (?=)正向肯定预查
- (?!)正向否定预查

- (?#comment) 提供注释让人阅读

反向引用

- 对正则表达式中用括号选择的子匹配会缓存到缓存区
- 子匹配按从左往右顺序存储

举例：`\b(\w+)\b\s+\1\b`

查找相同单词的匹配项

应用

```
public static void main(String[] args) {
        String regex = "(?<=(href=\")).{1,200}(?=(\">))";
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher("<a href=\"www.baidu.com\">");
        while (matcher.find()){
            String str = matcher.group(0);
            System.out.println(matcher.group(1));
            System.out.println(str);
        }


        String string = "<a href=\"www.baidu.com\">";
        String sss = string.replaceAll(regex,"aaaaaaa");
        String[] strings = string.split(regex);


    }
```

[菜鸟教程：正则表达式 - 元字符]

[正则表达式30分钟入门教程]

自定义异常处理

单一类的异常

自定义异常类

```
@AllArgsConstructor
@NoArgsConstructor
@Data
public class MyException extends RuntimeException {          //继承异常类

    private int status;
    private String msg;
}
```

```
@ExceptionHandler(MyException.class)
     public MyExceptionResponse exceptionHandler() {
         ErrorResponse resp = new ErrorResponse();
         resp.setCode(300);
         resp.setMsg("exception-Handler");
         return resp;
     }
```

自定义异常返回类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ErrorResponse {
    private int status;
    private String msg;
}
```

全局异常

自定义异常类

```
public class MySelfException extends Exception {
     MySelfException(String message) {
        super(message);
    }
}
```

异常处理切面类

```
@ControllerAdvice                //作为全局异常处理的切面类
@ResponseBody
public class ExceptionHandlerAdvice {
```

```
    @ExceptionHandler(MySelfException.class)            //处理指定异常
    public ErrorResponse handleException(MySelfException e) {

        return  new ErrorResponse(415,e.getMessage());
    }
}
/*
@ControllerAdvice(annotations=RestController.class)                //指定controller
@ControllerAdvice(basePackages={"com.xxx","com.ooo"})           //指定扫描包
*/
```

定时任务

步骤

> 依次加载所有的实现Scheduled注解的类方法->将对应类型的定时器放入相应的“定时任务列表”中->执行相应的定时任务

```
@Overridepublic Object postProcessAfterInitialization(final Object bean, String
beanName) {
    //...
    for (Map.Entry> entry : annotatedMethods.entrySet()) {
        Method method = entry.getKey();
        for (Scheduled scheduled : entry.getValue())
        {
            processScheduled(scheduled, method, bean);
         }
    }
}
            return bean;
}
```

```
//每次执行定时任务结束后，会先设置下下次定时任务的执行时间，以此来确认下次任务的执行时间。
public void run() {
    boolean periodic = isPeriodic();
    if (!canRunInCurrentRunState(periodic))
        cancel(false);
    else if (!periodic)
        ScheduledFutureTask.super.run();
    else if (ScheduledFutureTask.super.runAndReset()) {
        setNextRunTime();
        reExecutePeriodic(outerTask);
```

```
    }
}
```

- 从上面的代码可以看出，如果多个定时任务定义的是同一个时间，那么也是顺序执行的，会根据程序加载Scheduled方法的先后来执行

核心配置

1. TriggerTask：动态定时任务。通过Trigger#nextExecutionTime 给定的触发上下文确定下一个执行时间。
2. CronTask：动态定时任务，TriggerTask子类。通过cron表达式确定的时间触发下一个任务执行。
3. IntervalTask：一定时间延迟之后，周期性执行的任务。
4. taskScheduler 如果为空，默认是ConcurrentTaskScheduler，并使用默认单线程的ScheduledExecutor。

@Schedule

> 静态执行定时任务，默认为单线程，任务的执行时机会受上一个任务执行时间的影响

步骤

- 标记配置类@Configuration
- 开启定时任务@EnableScheduling
- 添加定时任务@Schedule

多线程定时任务

```
@Component              //不同层使用不同注解注入
@EnableScheduling
@EnableAsync
public class MultithreadScheduleTask {

        @Async          //异步
        @Scheduled(fixedDelay = 1000)  //间隔1秒
        public void first() throws InterruptedException {
            System.out.println("第一个定时任务开始 : " + LocalDateTime.now().toLocalTime()
+ "\r\n线程 : " + Thread.currentThread().getName());
            System.out.println();
            Thread.sleep(1000 * 10);
        }

        @Async
        @Scheduled(fixedDelay = 2000)
        public void second() {
            System.out.println("第二个定时任务开始 : " + LocalDateTime.now().toLocalTime()
+ "\r\n线程 : " + Thread.currentThread().getName());
            System.out.println();
        }
```

```
    }
```

cron表达式

> 一个cron表达式有至少6个（也可能7个）有空格分隔的时间元素。按顺序依次为：
>
> - 秒（0~59）
> - 分钟（0~59）
> - 小时（0~23）
> - 天（0~31）
> - 月（0~11）
> - 星期（1~7 1=SUN 或 SUN，MON，TUE，WED，THU，FRI，SAT）
> - 年份（1970 - 2099）

- “*”字符代表所有可能的值
- “/”字符用来指定数值的增量
- “？”字符仅被用于天（月）和天（星期）两个子表达式，表示不指定值 当2个子表达式其中之一被指定了值以后，为了避免冲突，需要将另一个子表达式的值设为“？”
- “L” 字符仅被用于天（月）和天（星期）两个子表达式
- W 字符代表着平日(Mon-Fri)，并且仅能用于日域中。它用来指定离指定日的最近的一个平日。
- C：代表“Calendar”的意思。它的意思是计划所关联的日期，如果日期没有被关联，则相当于日历中所有日期

举例

> “0 0/5 14,18 * * ?” 在每天下午2点到2:55期间和下午6点到6:55期间的每5分钟触发

> “0 15 10 ? * 6L” 每月的最后一个星期五上午10:15触发

> “0 15 10 ? * 6L 2002-2005” 2002年至2005年的每月的最后一个星期五上午10:15触发

日志配置

- springboot的默认日志框架是Logback

```
#root 日志级别以WARN级别输出
logging.level.root=WARN
#springframework.web日志以DEBUG级别输出
logging.level.org.springframework.web=DEBUG
#相对路径
#logging.file=log/my.log
#把日志信息写入日志文件，会自动生成
logging.file=D:\\springbootLogs\\info.log
#配置控制台日志显示格式
logging.pattern.console=%d{yyyy/MM/dd-HH:mm:ss} [%thread] %-5level %logger- %msg%n
#配置文件中日志显示格式
logging.pattern.file=%d{yyyy/MM/dd-HH:mm:ss}  [%thread] %-5level %logger- %msg%n
```

```
private Logger logger = LoggerFactory.getLogger(this.getClass());
log.info("日志输出info");
```

- 这里使用 SLF4J日志框架

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="10 seconds">

    <contextName>logback</contextName>
    <!-- 彩色日志 -->
    <!-- 彩色日志依赖的渲染类 -->
    <conversionRule conversionWord="clr"
converterClass="org.springframework.boot.logging.logback.ColorConverter"/>
    <conversionRule conversionWord="wex"

 converterClass="org.springframework.boot.logging.logback.WhitespaceThrowableProxyConve
rter"/>
    <conversionRule conversionWord="wEx"

 converterClass="org.springframework.boot.logging.logback.ExtendedWhitespaceThrowablePr
oxyConverter"/>
    <!-- 彩色日志格式 -->
    <property name="CONSOLE_LOG_PATTERN"
              value="${CONSOLE_LOG_PATTERN:-%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint}
%clr(${LOG_LEVEL_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint}
%clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint}
%m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}}"/>


    <!-- 格式化输出：%date表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度 %msg：日
志消息，%n是换行符-->
    <property name="LOG_PATTERN" value="%date{HH:mm:ss.SSS} [%thread] %-5level
%logger{36} - %msg%n"/>


    <!-- 定义日志存储的路径，不要配置相对路径 -->
    <!--<property name="FILE_PATH" value="D:/Desktop/log/spring-log.%d{yyyy-MM-
dd}.%i.log"/>-->
    <!--<property name="FILE_PATH" value="/var/lib/tomcat8/logs/prize_log/spring-
log.%d{yyyy-MM-dd}.%i.log"/>-->
    <property name="FILE_PATH" value="${catalina.base}/logs/prize_log/spring-
log.%d{yyyy-MM-dd}.%i.log"/>

    <!-- 控制台输出日志 -->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <!-- 日志级别过滤INFO以下 -->
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>INFO</level>
        </filter>
        <encoder>
            <!-- 按照上面配置的LOG_PATTERN来打印日志 -->
```

```
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
        </encoder>
    </appender>

    <!--每天生成一个日志文件，保存30天的日志文件。rollingFile用来切分文件的 -->
    <appender name="rollingFile"
class="ch.qos.logback.core.rolling.RollingFileAppender">

        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

            <fileNamePattern>${FILE_PATH}</fileNamePattern>

            <!-- keep 15 days' worth of history -->
            <maxHistory>30</maxHistory>

            <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <!-- 日志文件的最大大小 -->
                <maxFileSize>2MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>

            <!-- 超出删除老文件 -->
            <totalSizeCap>1GB</totalSizeCap>
        </rollingPolicy>

        <encoder>
            <pattern>${LOG_PATTERN}</pattern>
        </encoder>
    </appender>

    <!-- project default level -->
    <logger name="net.sh.rgface.serive" level="ERROR"/>

    <!-- 日志输出级别 -->
    <root level="INFO">
        <appender-ref ref="console"/>
        <appender-ref ref="rollingFile"/>
    </root>
</configuration>
```

日志输出内容元素

- 时间日期：精确到毫秒
- 日志级别：ERROR, WARN, INFO, DEBUG or TRACE
- 进程ID
- 分隔符：--- 标识实际日志的开始
- 线程名：方括号括起来（可能会截断控制台输出）
- Logger名：通常使用源代码的类名
- 日志内容

日志输出级别

> TRANCE < DRBUG < INFO < WARN < ERROR <FATAL < OFF

指定 Logger 日志级别后对记录日志的影响矩阵表（*N* 为不输出、*Y* 为输出）：

日志级别	trace	debug	info	warn	error
ERROR	N	N	N	N	Y
WARN	N	N	N	Y	Y
INFO	N	N	Y	Y	Y
DEBUG	N	Y	Y	Y	Y
TRACE	Y	Y	Y	Y	Y