springboot

1.日志

1.1日志框架简介

市面上的日志框架；

JUL、JCL、Jboss-logging、logback、log4j、log4j2、slf4j....

日志门面 （日志的抽象层）	日志实现
~~JCL（Jakarta Commons Logging）~~ SLF4j（Simple Logging Facade for Java） **~~jboss-logging~~**	Log4j JUL（java.util.logging） Log4j2 **Logback**

左边选一个门面（抽象层）、右边来选一个实现；

日志门面： SLF4J；

日志实现：Logback；

SpringBoot：底层是Spring框架，Spring框架默认是用JCL；‘

SpringBoot选用 SLF4j和logback；

1.2测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBoot03LoggingApplicationTests {

    //记录器
    Logger logger = LoggerFactory.getLogger(getClass());
    @Test
    public void contextLoads() {
        //System.out.println();

        //日志的级别；
        //由低到高   trace<debug<info<warn<error
        //可以调整输出的日志级别；日志就只会在这个级别以以后的高级别生效
        logger.trace("这是trace日志...");
        logger.debug("这是debug日志...");
        logger.info("这是info日志...");
        logger.warn("这是warn日志...");
        logger.error("这是error日志...");


    }

}
```

1.3调整日志级别

```
logging.level.com.atguigu=trace
```

1.4日志的输出位置

spring-boot-starter-log4j2	Starter for using Log4j2 for logging. An alternative to spring-boot-starter-logging
spring-boot-starter-logging	Starter for logging using Logback. Default logging starter

logging.file	logging.path	Example	Description
(none)	(none)		只在控制台输出
指定文件名	(none)	my.log	输出日志到my.log文件
(none)	指定目录	/var/log	输出到指定目录的 spring.log 文件中

```
#logging.path=
# 不指定路径在当前项目下生成springboot.log日志
# 可以指定完整的路径；
#logging.file=G:/springboot.log
# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹；使用 spring.log 作为默认文件
logging.path=/spring/log
```

1.5日志的输出格式

```
logging.level.com.atguigu=trace


#logging.path=
# 不指定路径在当前项目下生成springboot.log日志
# 可以指定完整的路径；
#logging.file=G:/springboot.log
# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹；使用 spring.log 作为默认文件
logging.path=/spring/log

#  在控制台输出的日志的格式
logging.pattern.console=%d{yyyy-MM-dd} [%thread] %-5level %logger{50} - %msg%n
# 指定文件中日志输出的格式
logging.pattern.file=%d{yyyy-MM-dd} === [%thread] === %-5level === %logger{50} ==== %msg%n
```

1.6指定日志文件

给类路径下放上每个日志框架自己的配置文件即可；SpringBoot就不使用他默认配置的了

Logging System	Customization
Logback	`logback-spring.xml`, `logback-spring.groovy`, `logback.xml` or `logback.groovy`
Log4j2	`log4j2-spring.xml` or `log4j2.xml`
JDK (Java Util Logging)	`logging.properties`

logback.xml：直接就被日志框架识别了；

logback-spring.xml：日志框架就不直接加载日志的配置项，由SpringBoot解析日志配置，可以使用SpringBoot的高级Profile功能

```
<springProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
    可以指定某段配置只在某个环境下生效
</springProfile>

```

logback-spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration  scan="true" scanPeriod="10 seconds">
    <!-- 日志级别从低到高分为TRACE < DEBUG < INFO < WARN < ERROR < FATAL，如果设置为WARN，则低于WARN的信息都不会输出 -->
    <!-- scan:当此属性设置为true时，配置文件如果发生改变，将会被重新加载，默认值为true -->
    <!-- scanPeriod:设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认单位是毫秒。当scan为true时，此属性生效。默认的时间间隔为1分钟。 -->
    <!-- debug:当此属性设置为true时，将打印出logback内部日志信息，实时查看logback运行状态。默认值为false。 -->
```

```
    <contextName>logback</contextName>
    <!-- name的值是变量的名称，value的值时变量定义的值。通过定义的值会被插入到logger上下文中。定义变量
后，可以使“${}”来使用变量。 -->
    <property name="log.path" value="D:/guli_1010/edu" />

    <!-- 彩色日志 -->
    <!-- 配置格式变量：CONSOLE_LOG_PATTERN 彩色日志格式 -->
    <!-- magenta:洋红 -->
    <!-- boldMagenta:粗红-->
    <!-- cyan:青色 -->
    <!-- white:白色 -->
    <!-- magenta:洋红 -->
    <property name="CONSOLE_LOG_PATTERN"
              value="%yellow(%date{yyyy-MM-dd HH:mm:ss}) |%highlight(%-5level) |%blue(%thread)
|%blue(%file:%line) |%green(%logger) |%cyan(%msg%n)"/>


    <!--输出到控制台-->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <!--此日志appender是为开发使用，只配置最底级别，控制台输出的日志级别是大于或等于此级别的日志信
息-->
        <!-- 例如：如果此处配置了INFO级别，则后面其他位置即使配置了DEBUG级别的日志，也不会被输出 -->
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>INFO</level>
        </filter>
        <encoder>
            <Pattern>${CONSOLE_LOG_PATTERN}</Pattern>
            <!-- 设置字符集 -->
            <charset>UTF-8</charset>
        </encoder>
    </appender>


    <!--输出到文件-->

    <!-- 时间滚动输出 level为 INFO 日志 -->
    <appender name="INFO_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!-- 正在记录的日志文件的路径及文件名 -->
        <file>${log.path}/log_info.log</file>
        <!--日志文件输出格式-->
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} -
%msg%n</pattern>
            <charset>UTF-8</charset>
        </encoder>
        <!-- 日志记录器的滚动策略，按日期，按大小记录 -->
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- 每天日志归档路径以及格式 -->
            <fileNamePattern>${log.path}/info/log-info-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
            <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <maxFileSize>100MB</maxFileSize>

            </timeBasedFileNamingAndTriggeringPolicy>
```

```
            <!--日志文件保留天数-->
            <maxHistory>15</maxHistory>
        </rollingPolicy>
        <!-- 此日志文件只记录info级别的 -->
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>INFO</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
    </appender>

    <!-- 时间滚动输出 level为 WARN 日志 -->
    <appender name="WARN_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!-- 正在记录的日志文件的路径及文件名 -->
        <file>${log.path}/log_warn.log</file>
        <!--日志文件输出格式-->
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} -
%msg%n</pattern>
            <charset>UTF-8</charset> <!-- 此处设置字符集 -->
        </encoder>
        <!-- 日志记录器的滚动策略，按日期，按大小记录 -->
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${log.path}/warn/log-warn-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
            <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <maxFileSize>100MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
            <!--日志文件保留天数-->
            <maxHistory>15</maxHistory>
        </rollingPolicy>
        <!-- 此日志文件只记录warn级别的 -->
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>warn</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
    </appender>


    <!-- 时间滚动输出 level为 ERROR 日志 -->
    <appender name="ERROR_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!-- 正在记录的日志文件的路径及文件名 -->
        <file>${log.path}/log_error.log</file>
        <!--日志文件输出格式-->
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} -
%msg%n</pattern>
            <charset>UTF-8</charset> <!-- 此处设置字符集 -->
        </encoder>
        <!-- 日志记录器的滚动策略，按日期，按大小记录 -->
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

            <fileNamePattern>${log.path}/error/log-error-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
```

```
            <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <maxFileSize>100MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
            <!--日志文件保留天数-->
            <maxHistory>15</maxHistory>
        </rollingPolicy>
        <!-- 此日志文件只记录ERROR级别的 -->
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>ERROR</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
    </appender>

    <!--
        <logger>用来设置某一个包或者具体的某一个类的日志打印级别、以及指定<appender>。
        <logger>仅有一个name属性,
        一个可选的level和一个可选的addtivity属性。
        name:用来指定受此logger约束的某一个包或者具体的某一个类。
        level:用来设置打印级别，大小写无关：TRACE，DEBUG，INFO，WARN，ERROR，ALL 和 OFF,
              如果未设置此属性，那么当前logger将会继承上级的级别。
    -->
    <!--
        使用mybatis的时候，sql语句是debug下才会打印，而这里我们只配置了info，所以想要查看sql语句的话，
有以下两种操作：
        第一种把<root level="INFO">改成<root level="DEBUG">这样就会打印sql，不过这样日志那边会出现很
多其他消息
        第二种就是单独给mapper下目录配置DEBUG模式，代码如下，这样配置sql语句会打印，其他还是正常DEBUG级
别：
     -->
    <!--开发环境:打印控制台-->
    <springProfile name="dev">
        <!--可以输出项目中的debug日志，包括mybatis的sql日志-->
        <logger name="com.guli" level="INFO" />

        <!--
            root节点是必选节点，用来指定最基础的日志输出级别，只有一个level属性
            level:用来设置打印级别，大小写无关：TRACE，DEBUG，INFO，WARN，ERROR，ALL 和 OFF，默认是
DEBUG
            可以包含零个或多个appender元素。
        -->
        <root level="INFO">
            <appender-ref ref="CONSOLE" />
            <appender-ref ref="INFO_FILE" />
            <appender-ref ref="WARN_FILE" />
            <appender-ref ref="ERROR_FILE" />
        </root>
    </springProfile>


    <!--生产环境:输出到文件-->

    <springProfile name="pro">
```

```
        <root level="INFO">
            <appender-ref ref="CONSOLE" />
            <appender-ref ref="DEBUG_FILE" />
            <appender-ref ref="INFO_FILE" />
            <appender-ref ref="ERROR_FILE" />
            <appender-ref ref="WARN_FILE" />
        </root>
    </springProfile>

</configuration>
```

2.异常处理

2.1创建自定义异常类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class GuliException extends RuntimeException {

    @ApiModelProperty(value = "状态码")
    private Integer code;

    private String msg;

}
```

2.2业务中需要的位置抛出GuliException

```
try {
    int a = 10/0;
}catch(Exception e) {
    throw new GuliException(20001,"出现自定义异常");
}
```

2.3添加异常处理方法 GlobalExceptionHandler.java中添加

```
@ExceptionHandler(GuliException.class)
@ResponseBody
public R error(GuliException e){
    e.printStackTrace();
    return R.error().message(e.getMsg()).code(e.getCode());
}
```

```
/**
 * 统一异常处理类
 */
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    @ResponseBody
    public R error(Exception e){
        e.printStackTrace();
        return R.error();
    }
}
```

2.4自定义错误页面(在templates下创建error.html)

3.快速简单入门

3.1创建一个maven工程

3.2导入spring boot相关的依赖

```
  <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.6.RELEASE</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
```

3.3编写一个主程序；启动Spring Boot应用

```
/**
 *  @SpringBootApplication 来标注一个主程序类，说明这是一个Spring Boot应用
 */
@SpringBootApplication
public class HelloWorldMainApplication {

    public static void main(String[] args) {

        // Spring应用启动起来
        SpringApplication.run(HelloWorldMainApplication.class,args);
    }
}
```

3.4编写相关的Controller、Service

```
@Controller
public class HelloController {

    @ResponseBody
    @RequestMapping("/hello")
    public String hello(){
        return "Hello World!";
    }
}
```

3.5运行主程序测试

3.6简化部署

```
<!-- 这个插件，可以将应用打包成一个可执行的jar包；-->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
```

4.配置文件

4.1配置文件值注入

配置文件

```
person:
    lastName: hello
    age: 18
    boss: false
    birth: 2017/12/12
    maps: {k1: v1,k2: 12}
    lists:
      - lisi
      - zhaoliu
    dog:
      name: 小狗
      age: 12
```

javaBean

```
/**
 * 将配置文件中配置的每一个属性的值，映射到这个组件中
 * @ConfigurationProperties：告诉SpringBoot将本类中的所有属性和配置文件中相关的配置进行绑定；
 *      prefix = "person"：配置文件中哪个下面的所有属性进行一一映射
 *
 * 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；
 *
 */
@Component
@ConfigurationProperties(prefix = "person")
public class Person {

    private String lastName;
    private Integer age;
    private Boolean boss;
    private Date birth;

    private Map<String,Object> maps;
    private List<Object> lists;
    private Dog dog;
```

4.2@Value获取值和@ConfigurationProperties获取值比较

	@ConfigurationProperties	@Value
功能	批量注入配置文件中的属性	一个个指定
松散绑定（松散语法）	支持	不支持
SpEL	不支持	支持
JSR303数据校验	支持	不支持
复杂类型封装	支持	不支持

配置文件yml还是properties他们都能获取到值；

如果说，我们只是在某个业务逻辑中需要获取一下配置文件中的某项值，使用@Value；

如果说，我们专门编写了一个javaBean来和配置文件进行映射，我们就直接使用@ConfigurationProperties；

4.3配置文件注入值数据校验

```
@Component
@ConfigurationProperties(prefix = "person")
@Validated
public class Person {

    /**
     * <bean class="Person">
     *      <property name="lastName" value="字面量/${key}从环境变量、配置文件中获取值/#{SpEL}">
</property>
     * <bean/>
     */

   //lastName必须是邮箱格式
    @Email
    //@Value("${person.last-name}")
    private String lastName;
    //@Value("#{11*2}")
    private Integer age;
    //@Value("true")
    private Boolean boss;

    private Date birth;
    private Map<String,Object> maps;
    private List<Object> lists;
    private Dog dog;
```

4.4@PropertySource&@ImportResource&@Bean

4.4.1@PropertySource：加载指定的配置文件；

```
/**
```

```
 * 将配置文件中配置的每一个属性的值，映射到这个组件中
 * @ConfigurationProperties：告诉SpringBoot将本类中的所有属性和配置文件中相关的配置进行绑定；
 *      prefix = "person"：配置文件中哪个下面的所有属性进行一一映射
 *
 * 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；
 *  @ConfigurationProperties(prefix = "person")默认从全局配置文件中获取值；
 *
 */
@PropertySource(value = {"classpath:person.properties"})
@Component
@ConfigurationProperties(prefix = "person")
//@Validated
public class Person {

    /**
     * <bean class="Person">
     *      <property name="lastName" value="字面量/${key}从环境变量、配置文件中获取值/#{SpEL}">
</property>
     * <bean/>
     */

   //lastName必须是邮箱格式
   // @Email
    //@Value("${person.last-name}")
    private String lastName;
    //@Value("#{11*2}")
    private Integer age;
    //@Value("true")
    private Boolean boss;
```

4.4.2导入Spring的配置文件，让配置文件里面的内容生效；

@ImportResource（了解）

SpringBoot推荐给容器中添加组件的方式，使用全注解的方式（推荐）

1、配置类@Configuration------>Spring配置文件

2、使用@Bean给容器中添加组件

```
/**
 * @Configuration：指明当前类是一个配置类；就是来替代之前的Spring配置文件
 *
 * 在配置文件中用<bean><bean/>标签添加组件
 *
 */
@Configuration
public class MyAppConfig {

    //将方法的返回值添加到容器中；容器中这个组件默认的id就是方法名
    @Bean

    public HelloService helloService02(){
```

```
        System.out.println("配置类@Bean给容器中添加组件了...");
        return new HelloService();
    }
}
```

4.5配置文件占位符

1、随机数

```
${random.value}、${random.int}、${random.long}
${random.int(10)}、${random.int[1024,65536]}
```

2、占位符获取之前配置的值，如果没有可以是用:指定默认值

```
person.last-name=张三${random.uuid}
person.age=${random.int}
person.birth=2017/12/15
person.boss=false
person.maps.k1=v1
person.maps.k2=14
person.lists=a,b,c
person.dog.name=${person.hello:hello}_dog
person.dog.age=15
```

4.6Profile

1、多Profile文件

我们在主配置文件编写的时候，文件名可以是 application-{profile}.properties/yml

默认使用application.properties的配置；

2、yml支持多文档块方式（了解）

```


server:
  port: 8081
spring:
  profiles:
    active: prod

---
server:
  port: 8083
spring:
  profiles: dev



---
```

```
server:
  port: 8084
spring:
  profiles: prod  #指定属于哪个环境
```

3、激活指定profile

1、在配置文件中指定 spring.profiles.active=dev

```
2、命令行：

    java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev；

    可以直接在测试的时候，配置传入命令行参数

3、虚拟机参数；

    -Dspring.profiles.active=dev
```

4.7配置文件加载位置(了解)

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:./

–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；

==我们还可以通过spring.config.location来改变默认的配置文件位置==

项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；指定配置文件和默认加载的这些配置文件共同起作用形成互补配置；

java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --spring.config.location=G:/application.properties

4.8自动配置原理（重点）

1.SpringBoot启动的时候加载主配置类，开启了自动配置功能 ==@EnableAutoConfiguration==

2.@EnableAutoConfiguration 作用：

- 利用EnableAutoConfigurationImportSelector给容器中导入一些组件
- 可以查看selectImports()方法的内容；
- List configurations = getCandidateConfigurations(annotationMetadata, attributes);获取候选的配置

```
SpringFactoriesLoader.loadFactoryNames()
扫描所有jar包类路径下  META-INF/spring.factories
把扫描到的这些文件的内容包装成properties对象
从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器中
```

将 类路径下 META-INF/spring.factories 里面配置的所有EnableAutoConfiguration的值加入到了容器中；

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfigura
tion,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.ldap.LdapDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\
org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\
org.springframework.boot.autoconfigure.mobile.DeviceResolverAutoConfiguration,\
org.springframework.boot.autoconfigure.mobile.DeviceDelegatingViewResolverAutoConfiguration,\
org.springframework.boot.autoconfigure.mobile.SitePreferenceAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.reactor.ReactorAutoConfiguration,\
org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.SecurityFilterAutoConfiguration,\
org.springframework.boot.autoconfigure.security.FallbackWebSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.OAuth2AutoConfiguration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.social.SocialWebAutoConfiguration,\
org.springframework.boot.autoconfigure.social.FacebookAutoConfiguration,\
org.springframework.boot.autoconfigure.social.LinkedInAutoConfiguration,\
org.springframework.boot.autoconfigure.social.TwitterAutoConfiguration,\
org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration,\
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration,\
org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration,\
org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.web.WebClientAutoConfiguration,\
org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration,\

org.springframework.boot.autoconfigure.websocket.WebSocketMessagingAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration
```

每一个这样的 xxxAutoConfiguration类都是容器中的一个组件，都加入到容器中；用他们来做自动配置；

每一个自动配置类进行自动配置功能

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;
    }
```

根据当前不同的条件判断，决定这个配置类是否生效？

一但这个配置类生效；这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的properties类中获取的，这些类里面的每一个属性又是和配置文件绑定的；

所有在配置文件中能配置的属性都是在xxxxProperties类中封装者'；配置文件能配置什么就可以参照某个功能对应的这个属性类

```
@ConfigurationProperties(prefix = "spring.http.encoding")  //从配置文件中获取指定的值和bean的属性进行绑定
public class HttpEncodingProperties {

   public static final Charset DEFAULT_CHARSET = Charset.forName("UTF-8");
```

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些属性的值；

xxxxAutoConfigurartion：自动配置类；

给容器中添加组件

xxxxProperties:封装配置文件中相关属性

细节

@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；

==我们可以通过启用 debug=true属性；来让控制台打印自动配置报告==，这样我们就可以很方便的知道哪些自动配置类生效；

```
=========================
AUTO-CONFIGURATION REPORT
=========================


Positive matches:（自动配置类启用的）
-----------------

   DispatcherServletAutoConfiguration matched:
      - @ConditionalOnClass found required class
'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass did not find
unwanted class (OnClassCondition)
      - @ConditionalOnWebApplication (required) found StandardServletEnvironment
(OnWebApplicationCondition)


Negative matches:（没有启动，没有匹配成功的自动配置类）
-----------------

   ActiveMQAutoConfiguration:
      Did not match:
         - @ConditionalOnClass did not find required classes 'javax.jms.ConnectionFactory',
'org.apache.activemq.ActiveMQConnectionFactory' (OnClassCondition)

   AopAutoConfiguration:
      Did not match:
         - @ConditionalOnClass did not find required classes
'org.aspectj.lang.annotation.Aspect', 'org.aspectj.lang.reflect.Advice' (OnClassCondition)
```

启动器

- 说白了就是spring boot的启动场景
- 比如spring-boot-starter-web,就会帮我们自动导入web环境所有的依赖
- springboot会将所有的功能场景变成一个个启动器
- 我们要什么功能，引入相应的启动器就可以了

自动配置过程

1. @SpringBootApplication
2. @EnableAutoConfiguration
3. @Import(AutoConfigurationImportSelector.class)
4. selectImports（）
5. getAutoConfigurationEntry（）
6. getCandidateConfigurations（）
7. loadFactoryNames（）

8. loadSpringFactories（）
9. getResources("META-INF/spring.factories")
10. **简单点说，就是加载spring.factories文件中的自动配置类，这些自动配置类根据引入的jar包等条件判断是否起作用，引入相应的组件到容器中**

5.springboot和缓存

5.1JSR107

一、JSR107

Java Caching定义了5个核心接口，分别是**CachingProvider**, **CacheManager**, **Cache**, **Entry** 和 **Expiry**。

- **CachingProvider**定义了创建、配置、获取、管理和控制多个**CacheManager**。一个应用可以在运行期访问多个CachingProvider。
- **CacheManager**定义了创建、配置、获取、管理和控制多个唯一命名的**Cache**，这些Cache存在于CacheManager的上下文中。一个CacheManager仅被一个CachingProvider所拥有。
- **Cache**是一个类似Map的数据结构并临时存储以Key为索引的值。一个Cache仅被一个CacheManager所拥有。
- **Entry**是一个存储在Cache中的key-value对。
- **Expiry** 每一个存储在Cache中的条目有一个定义的有效期。一旦超过这个时间，条目为过期的状态。一旦过期，条目将不可访问、更新和删除。缓存有效期可以通过ExpiryPolicy设置。

5.2Spring缓存抽象

5.2.1几个重要概念&缓存注解

Cache	**缓存接口，定义缓存操作。实现有：RedisCache、EhCacheCache、ConcurrentMapCache等**
CacheManager	**缓存管理器，管理各种缓存（Cache）组件**
@Cacheable	**主要针对方法配置，能够根据方法的请求参数对其结果进行缓存**
@CacheEvict	**清空缓存**
@CachePut	**保证方法被调用，又希望结果被缓存。**
@EnableCaching	**开启基于注解的缓存**
keyGenerator	**缓存数据时key生成策略**
serialize	**缓存数据时value序列化策略**

@Cacheable/@CachePut/@CacheEvict 主要的参数		
value	缓存的名称，在 spring 配置文件中定义，必须指定至少一个	例如： @Cacheable(value="mycache") 或者 @Cacheable(value={"cache1","cache2"}
key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合	例如： @Cacheable(value="testcache",key="#userName")
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存/清除缓存，在调用方法之前之后都能判断	例如： @Cacheable(value="testcache",condition="#userName.length()>2")
allEntries (**@CacheEvict**)	是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存	例如： @CachEvict(value="testcache",allEntries=true)
beforeInvocation **(@CacheEvict)**	是否在方法执行前就清空，缺省为 false，如果指定为 true，则在方法还没有执行的时候就清空缓存，缺省情况下，如果方法执行抛出异常，则不会清空缓存	例如： @CachEvict(value="testcache", beforeInvocation=true)
unless **(@CachePut)** **(@Cacheable)**	用于否决缓存的，不像condition，该表达式只在方法执行之后判断，此时可以拿到返回值result进行判断。条件为true不会缓存，fasle才缓存	例如： @Cacheable(value="testcache",unless="#result == null")

Cache SpEL available metadata

名字	位置	描述	示例
methodName	root object	当前被调用的方法名	#root.methodName
method	root object	当前被调用的方法	#root.method.name
target	root object	当前被调用的目标对象	#root.target
targetClass	root object	当前被调用的目标对象类	#root.targetClass
args	root object	当前被调用的方法的参数列表	#root.args[0]
caches	root object	当前方法调用使用的缓存列表（如@Cacheable(value={"cache1", "cache2"})），则有两个cache	#root.caches[0].name
argument name	evaluation context	方法参数的名字. 可以直接 #参数名 ，也可以使用 #p0或#a0 的形式，0代表参数的索引；	#iban 、 #a0 、 #p0
result	evaluation context	方法执行后的返回值（仅当方法执行之后的判断有效，如 'unless'，'cache put'的表达式 'cache evict'的表达式 beforeInvocation=false）	#result

5.3缓存使用

- 1、引入spring-boot-starter-cache模块
- 2、@EnableCaching开启缓存
- 3、使用缓存注解
- 4、切换为其他缓存

```
@Cacheable(value = "mm",key = "#id")
public EduTest query(String id) {
    System.out.println("查询数据库");
   return eduTestDao.selectById(id);
}

@CachePut(value = "mm",key = "#result.id")
public EduTest updata() {
    System.out.println("updata***************");
    EduTest eduTest = new EduTest();
    eduTest.setId("1259769163578511362");
    eduTest.setName("朱四亿");
    int i = eduTestDao.updateById(eduTest);
    EduTest eduTest1 = eduTestDao.selectById( serializable: "1259769163578511362");
    return eduTest1;
}
```

```
String[] value() default {}
String[] cacheNames() default {}
String key() default ""
String keyGenerator() default ""
String cacheManager() default ""
String cacheResolver() default ""
String condition() default ""
boolean allEntries() default false
boolean beforeInvocation() default false

@CacheEvict()
public EduTest updata() {
```

@Cacheing 组合注解

```
Cacheable[] cacheable() default {}
CachePut[] put() default {}
CacheEvict[] evict() default {}

@Caching()
public EduTest updata() {
```

@CacheConfig(处理缓存的公共配置)

```
String[] cacheNames() default {}
String keyGenerator() default ""
String cacheManager() default ""
String cacheResolver() default ""

@CacheConfig()
public class EduTestServiceImpl {
```

5.4缓存整合redis

1.安装redis

2.引入redis启动器

3.配置文件中添加

```
spring.redis.host=118.24.44.169
```

4.redis操作

```java
@Autowired
StringRedisTemplate stringRedisTemplate;  //操作k-v都是字符串的

@Autowired
RedisTemplate redisTemplate;  //k-v都是对象的


/**
 * Redis常见的五大数据类型
 *  String（字符串）、List（列表）、Set（集合）、Hash（散列）、ZSet（有序集合）
 *  stringRedisTemplate.opsForValue()[String（字符串）]
 *  stringRedisTemplate.opsForList()[List（列表）]
 *  stringRedisTemplate.opsForSet()[Set（集合）]
 *  stringRedisTemplate.opsForHash()[Hash（散列）]
 *  stringRedisTemplate.opsForZSet()[ZSet（有序集合）]
 */
@Test
public void test01(){
    //给redis中保存
    stringRedisTemplate.opsForValue().append( k: "msg", s: "hello");
}
```

```java
@Test
public void test01(){
    //给redis中保存数据
    //stringRedisTemplate.opsForValue().append("msg","hello");
    String msg = stringRedisTemplate.opsForValue().get("msg");
    System.out.println(msg);

    stringRedisTemplate.opsForList().leftPush("mylist","1");
    stringRedisTemplate.opsForList().leftPush("mylist","2");
}
```

```java
@Configuration
public class MyRedis {


/*
        redisTemplate 默认使用JDK的序列化机制，存储二进制字节码，所以自定义序列化类
     @param redisConnectionFactory redis连接工厂类
    @return RedisTemplate*/


    @Bean(name = "redisTemplate")
    public RedisTemplate<String,Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory){

        RedisTemplate<String,Object> redisTemplate = new RedisTemplate<>();

        redisTemplate.setConnectionFactory(redisConnectionFactory);

        redisTemplate.setKeySerializer(keySerializer());
```

```
        redisTemplate.setHashKeySerializer(keySerializer());
        redisTemplate.setValueSerializer(valueSerializer());
        redisTemplate.setHashValueSerializer(valueSerializer());
        return redisTemplate;
    }

    @Primary
    @Bean
    public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory){
        //缓存配置对象
        RedisCacheConfiguration redisCacheConfiguration =
RedisCacheConfiguration.defaultCacheConfig();

        redisCacheConfiguration = redisCacheConfiguration.entryTtl(Duration.ofMinutes(30L)) //设
置缓存的默认超时时间：30分钟
                .disableCachingNullValues()             //如果是空值，不缓存

.serializeKeysWith(RedisSerializationContext.SerializationPair.fromSerializer(keySerializer()))
        //设置key序列化器

.serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer((valueSerializer
()))); //设置value序列化器

        return RedisCacheManager
                .builder(RedisCacheWriter.nonLockingRedisCacheWriter(redisConnectionFactory))
                .cacheDefaults(redisCacheConfiguration).build();
    }


    private RedisSerializer<String> keySerializer() {
        return new StringRedisSerializer();
    }

    private RedisSerializer<Object> valueSerializer() {
        return new GenericJackson2JsonRedisSerializer();
    }
}
```

6.springboot,异步任务，定时任务，邮件任务

6.1异步任务

在Java应用中，绝大多数情况下都是通过同步的方式来实现交互处理的；但是在处理与第三方系统交互的时候，容易造成响应迟缓的情况，之前大部分都是使用多线程来完成此类任务，其实，在Spring 3.x之后，就已经内置了@Async来完美解决这个问题。

两个注解：

@EnableAysnc、@Aysnc

快速相应，加@Async的方法，spring会开启另一个线程执行

```
@SpringBootApplication
@EnableAsync
public class SpringbootTaskApplication {

    public static void main(String[] args) { SpringApplication.run(SpringbootTaskApplication.

}
```

```
@Service
public class AsyncService {


    @Async
    public void hello() {

        try {
            Thread.sleep( millis: 3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("数据处理中");
    }
}
```

6.2定时任务（重点）

项目开发中经常需要执行一些定时任务，比如需要在每天凌晨时候，分析一次前一天的日志信息。Spring为我们提供了异步执行任务调度的方式，提供 TaskExecutor 、TaskScheduler 接口。

两个注解：@EnableScheduling、@Scheduled

cron表达式：

字段	允许值	允许的特殊字符
秒	0-59	, - * /
分	0-59	, - * /
小时	0-23	, - * /
日期	1-31	, - * ? / L W C
月份	1-12	, - * /
星期	0-7或SUN-SAT 0,7是SUN	, - * ? / L C #

特殊字符	代表含义
,	枚举
-	区间
*	任意
/	步长
?	日/星期冲突匹配
L	最后
W	工作日
C	和calendar联系后计算过的值
#	星期，4#2，第2个星期四

```
@SpringBootApplication
@EnableAsync
@EnableScheduling
public class SpringbootTaskApplication {

    public static void main(String[] args) { SpringApplication.run(SpringbootTaskApplication

}
```

```
@Service
public class SchedulerService {

    @Scheduled(cron = "49 29 17 * * ? ")
    public void scheduler() {
        System.out.println("hello");
    }
}
```

```
/**
 * second(秒), minute（分）, hour（时）, day of month（日）, month（月）, day of week（周几）.
 * 0 * * * * MON-FRI
 *  【0 0/5 14,18 * * ?】 每天14点整，和18点整，每隔5分钟执行一次
 *  【0 15 10 ? * 1-6】 每个月的周一至周六10:15分执行一次
 *  【0 0 2 ? * 6L】每个月的最后一个周六凌晨2点执行一次
 *  【0 0 2 LW * ?】每个月的最后一个工作日凌晨2点执行一次
 *  【0 0 2-4 ? * 1#1】每个月的第一个周一凌晨2点到4点期间，每个整点都执行一次；
 */
// @Scheduled(cron = "0 * * * * MON-SAT")
 //@Scheduled(cron = "0,1,2,3,4 * * * * MON-SAT")
// @Scheduled(cron = "0-4 * * * * MON-SAT")
 @Scheduled(cron = "0/4 * * * * MON-SAT")  //每4秒执行一次
```

6.3邮件任务

- 邮件发送需要引入spring-boot-starter-mail
- Spring Boot 自动配置MailSenderAutoConfiguration
- 定义MailProperties内容，配置在application.yml中
- 自动装配JavaMailSender
- 测试邮件发送

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

```
spring.mail.username=534096094@qq.com
spring.mail.password=gtstkoszjelabijb
spring.mail.host=smtp.qq.com
spring.mail.properties.mail.smtp.ssl.enable=true
```

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class Springboot04TaskApplicationTests {

    @Autowired
    JavaMailSenderImpl mailSender;

    @Test
    public void contextLoads() {
        SimpleMailMessage message = new SimpleMailMessage();
        //邮件设置
        message.setSubject("通知-今晚开会");
        message.setText("今晚7:30开会");

        message.setTo("17512080612@163.com");
        message.setFrom("534096094@qq.com");

        mailSender.send(message);
    }

}
```

```java
@Test
public void test02() throws  Exception{
    //1、创建一个复杂的消息邮件
    MimeMessage mimeMessage = mailSender.createMimeMessage();
    MimeMessageHelper helper = new MimeMessageHelper(mimeMessage,  multipart: true);

    //邮件设置
    helper.setSubject("通知-今晚开会");
    helper.setText("<b style='color:red'>今天 7:30 开会</b>");

    helper.setTo("17512080612@163.com");
    helper.setFrom("534096094@qq.com");

    //上传文件
    helper.addAttachment( attachmentFilename: "1.jpg",new File( pathname: "1"));
    helper.addAttachment( attachmentFilename: "2.jpg",new File( pathname: ""));

    mailSender.send(mimeMessage);

}
```

7.Spring Boot与开发热部署（了解）

在开发中我们修改一个Java文件后想看到效果不得不重启应用，这导致大量时间花费，我们希望不重启应用的情况下，程序可以自动部署（热部署）。有以下四种情况，如何能实现热部署。

- 1、模板引擎
  - 在Spring Boot中开发情况下禁用模板引擎的cache
  - 页面模板改变ctrl+F9可以重新编译当前页面并生效

2、Spring Loaded

Spring官方提供的热部署程序，实现修改类文件的热部署

- 下载Spring Loaded（项目地址https://github.com/spring-projects/spring-loaded）
- 添加运行时参数；

-javaagent:C:/springloaded-1.2.5.RELEASE.jar –noverify

3、JRebel

- 收费的一个热部署软件
- 安装插件使用即可

4、Spring Boot Devtools（推荐）

– 引入依赖

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

– IDEA使用ctrl+F9

– 或做一些小调整

Intellij IEDA和Eclipse不同，Eclipse设置了自动编译之后，修改类它会自动编译，而IDEA在非RUN或DEBUG情况下才会自动编译（前提是你已经设置了Auto-Compile）。

- 设置自动编译（settings-compiler-make project automatically）
- ctrl+shift+alt+/（maintenance）
- 勾选compiler.automake.allow.when.app.running

需要按ctrl+F9从新加载

8.Spring Boot与监控管理

58965457570

```
#配置开启其他端点的URI
#开启所有的端点访问：*
#指定开启端点访问：
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=
```

开启所有端点

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

springboot admin(可视化管理)

2 使用可视化监控应用 Spring Boot Admin

2.1 使用步骤

Spring Boot Admin 的使用是需要建立服务端与客户端。
服务端：独立的项目，会将搜集到的数据在自己的图形界面中展示。
客户端：需要监控的项目。
对应关系：一个服务端可以监控多个客户端。

2.1.1 搭建服务端

2.1.1.1 创建项目

2.1.1.2 修改 POM 文件

注意：目前在 Spring Boot Admin Starter Server2.1.6 版本中不支持 Spring Boot2.2.x 版本。

服务端

1.引入依赖，web启动器，修改spring boot版本至2.1.x

```
<!-- https://mvnrepository.com/artifact/de.codecentric/spring-boot-admin-starter-server -->
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>2.1.6</version>
</dependency>
```

2.修改端口

```
server.port=9090
```

3.在启动类添加注解

```
@SpringBootApplication
@EnableAdminServer
public class SpringbootAdminServerApplication {

    public static void main(String[] args) { SpringApplication.run(SpringbootAdminServe

}
```

http://localhost:9090/

客户端

1.引入依赖

```
<!-- https://mvnrepository.com/artifact/de.codecentric/spring-boot-admin-starter-client -->
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.1.6</version>
</dependency>
```

2.配置地址（spring.boot.admin.client.url: ）

9.springboot服务器端数据校验

无需引入依赖，web启动器已整合

9.1对实体对象校验

```
public class Users {
    @NotNull
    private  Integer userid;
    @NotBlank
    private String username;
    @NotBlank
    @NotEmpty
    private String usersex;

    public Integer getUserid() {
        return userid;
    }

    public void setUserid(Integer userid) {
        this.userid = userid;
```

@NotNull:对基本数据类型的对象类型做非空校验（Long,Integer,可以，int等不可以）

@NotBlank:对字符串类型做非空校验

@NotEmpty:对集合类型做非空校验

```
@RequestMapping("/addUser")
public String addUser(@VUsers users){
    Validated (org.springframework.validation.annotation)
```

9.2对controller进行数据校验

@Validated:对对象做校验

```java
public String addUser(@Validated Users users,BindingResult result)
    if(result.hasErrors()){
        List<ObjectError> list = result.getAllErrors();
        for(ObjectError err:list){
            FieldError fieldError = (FieldError) err;
            String fieldName = fieldError.getField();
            String msg = fieldError.getDefaultMessage();
            System.out.println(fieldName+"\t"+msg);
        }
    }
    System.out.println(users);
    return "ok";
}
```

9.3自定义错误信息

```java
public class Users {
    @NotNull(message = "用户ID不能为空")
    private  Integer userid;
    @NotBlank(message = "用户姓名不能为空")
    private String username;
    @NotBlank(message = "用户性别不能为空")
    private String usersex;
```

9.4其他校验规则

1.2.6 其他校验规则

@NotNull: 判断基本数据类型的对象类型是否为 null

@NotBlank: 判断字符串是否为 null 或者是空串(去掉首尾空格)。

@NotEmpty: 判断集合是否为空。

@Length: 判断字符的长度(最大或者最小)

@Min: 判断数值最小值

@Max: 判断数值最大值

@Email: 判断邮箱是否合法

9.5controller中其他参数校验

在controller上加@Validator

在参数上加校验规则，比如@NotBlank

10.springboot整合servlet三大组件

10.1servlet

```
@WebServlet(name = "FirstServlet",urlPatterns = "/servlet")
public class FirstServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        System.out.println("firstservlet-------------");

    }
}
```

在启动类添加@ServletComponentScan

```
@SpringBootApplication
@ServletComponentScan
public class SpringbootSpringmvcApplication {

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

}
```

10.2filter

```
@WebServlet(name = "FirstServlet",urlPatterns = "/servlet")
public class FirstServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        System.out.println("firstservlet-------------");

    }
}
```

10.3listener

```
@WebListener
public class FirstListener implements ServletContextListener {
```

```
    //servlet初始化调用
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("ServletContextListener----------init");
    }
    //servlet销毁调用
    @Override
    public void contextDestroyed(ServletContextEvent sce) {

    }
}
```

11.springboot 与消息（RabbitMQ）

1.引入web和rabbitMQ的启动器

2.配置

3.测试

```java
@Autowired
RabbitTemplate rabbitTemplate;

/**
 * 1、单播（点对点）
 */
@Test
public void contextLoads() {
    //Message需要自己构造一个;定义消息体内容和消息头
    //rabbitTemplate.send(exchage,routeKey,message);

    //object默认当成消息体，只需要传入要发送的对象，自动序列化发送给rabbitmq;
    //rabbitTemplate.convertAndSend(exchage,routeKey,object);
    Map<String,Object> map = new HashMap<>();
    map.put("msg","这是第一个消息");
    map.put("data", Arrays.asList("helloworld",123,true));
    rabbitTemplate.convertAndSend( exchange: "exchange.direct", routingKey: "atguigu.news",map);

}
```

4.通过序列化方式，将数据转为json发送和接受

```java
@Configuration
public class MyAMQPConfig {

    @Bean
    public MessageConverter messageConverter(){
        return new Jackson2JsonMessageConverter();
    }
}
```

5.

```java
//接受数据,如何将数据自动的转为json发送出去
@Test
public void receive(){
    Object o = rabbitTemplate.receiveAndConvert( queueName: "atguigu.news");
    System.out.println(o.getClass());
    System.out.println(o);
}
```

6.

```java
 */
@EnableRabbit  //开启基于注解的RabbitMQ模式
@SpringBootApplication
public class Springboot02AmqpApplication {

    public static void main(String[] args) { SpringApplication.run(Springboot0
}
```

7.监听

```
@RabbitListener(queues = "atguigu.news")
public void receive(Book book){
    System.out.println("收到消息: "+book);
}

@RabbitListener(queues = "atguigu")
public void receive02(Message message){
    System.out.println(message.getBody());
    System.out.println(message.getMessageProperties());
}
```

8.通过代码创建quenue和exchange(amqpAdmin)

```
@Autowired
AmqpAdmin amqpAdmin;

@Test
public void createExchange(){

    amqpAdmin.declareExchange(new DirectExchange("amqpadmin.exchange"));
    System.out.println("创建完成");

    amqpAdmin.declareQueue(new Queue("amqpadmin.queue",true));
    //创建绑定规则

    amqpAdmin.declareBinding(new Binding( destination: "amqpadmin.queue", Binding.DestinationTy
}
```

12.springboot整合elasticsearch

1.docker安装elasticsearch

简单入门

postman测试

2.增（**put请求**）

www.allweing.cn:9200/megacorp/employee/1

megacorp:索引（数据库）

employee:类型 （表）

文档: （每行数据）

```
{
    "first_name" : "John",
    "last_name" :  "Smith",
    "age" :        25,
    "about" :      "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}
```

3.删（**delete请求**）

```
www.allweing.cn:9200/megacorp/employee/1
```

4.改（**put请求**）同增

5.查(**get请求**)

```
www.allweing.cn:9200/megacorp/employee/1
```

6查找所有

```
www.allweing.cn:9200/megacorp/employee/_search
```

7.有条件简单查询

```
www.allweing.cn:9200/megacorp/employee/_search?age=25
```

8有条件复杂查询（post请求）

```
post请求   www.allweing.cn:9200/megacorp/employee/_search

请求体
{
    "query" : {
        "match" : {
            "last_name" : "Smith"
        }
    }
}
```

9.更复杂的条件查询

```
post请求   www.allweing.cn:9200/megacorp/employee/_search

请求体
{
    "query" : {
        "bool": {
            "must": {
                "match" : {
```

```
                    "last_name" : "smith"
                }
            },
            "filter": {
                "range" : {
                    "age" : { "gt" : 30 }
                }
            }
        }
    }
}
```

等等等等。。。

spring-data-elasticsearch

1.引入相关依赖web elasticsearch启动器

2.

```
spring.data.elasticsearch.cluster-name=elasticsearch
spring.data.elasticsearch.cluster-nodes=118.24.44.169:9300
```

（可能出现连接超时，修改版本适配，spring-data-elasticsearch 和elasticsearch ）

Spring Data Release Train	Spring Data Elasticsearch	Elasticsearch	Spring Boot
Neumann[1]	4.0.x[1]	7.6.2	2.3.x[1]
Moore	3.2.x	6.8.6	2.2.x
Lovelace	3.1.x	6.2.2	2.1.x
Kay[2]	3.0.x[2]	5.5.0	2.0.x[2]
Ingalls[2]	2.1.x[2]	2.4.0	1.5.x[2]

3

Book:数据实体类，internet:实体类主键

```
public interface BookRepository extends ElasticsearchRepository<Book,Integer> {
}
```

T extends Object
ID extends Serializable

4.

```java
@Document(indexName = "atguigu",type = "book")
public class Book {
    private Integer id;
    private String bookName;
    private String author;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getBookName() {
        return bookName;
    }

    public void setBookName(String bookName) {
        this.bookName = bookName;
    }

    public String getAuthor() {
        return author;
    }
```

5.

```java
@Autowired
BookRepository bookRepository;

@Test
public void test02(){
    Book book = new Book();
    book.setId(1);
    book.setBookName("西游记");
    book.setAuthor("吴承恩");
    bookRepository.index(book);

}
```

13.springboot整合springsecurity(安全框架)

14.springboot自定义启动器

14.1创建启动工程

allweing-spring-boot-starter（只做打包发布引入功能）

引入自动配置工程

```
<dependencies>
        <dependency>
            <groupId>cn.allweing.starter</groupId>
            <artifactId>allweing-spring-boot-starter-autoconfiguration</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
    </dependencies>
```

14.2创建自动配置工程

引入依赖

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <groupId>cn.allweing.starter</groupId>
    <artifactId>allweing-spring-boot-starter-autoconfiguration</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <!--所有starter基本配置-->
        <dependency>
```

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

编写业务类

```
public class HelloService {


    HelloProperties helloProperties;

    public HelloProperties getHelloProperties() {
        return helloProperties;
    }

    public void setHelloProperties(HelloProperties helloProperties) {
        this.helloProperties = helloProperties;
    }
        //主要方法
    public String getName(String name) {
        return helloProperties.getPrefix()+name+helloProperties.getSuffix();
    }
}
```

编写属性读取类

```
@ConfigurationProperties(prefix = "cn.allweing")
public class HelloProperties {
   private String prefix;
   private String suffix;

    public String getPrefix() {
        return prefix;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public String getSuffix() {
        return suffix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
}
```

编写自动配置类

```
@Configuration
@ConditionalOnWebApplication
@EnableConfigurationProperties(HelloProperties.class)
public class HelloServiceAutoConfiguration {

    @Resource
    HelloProperties helloProperties;

    @Bean
    public HelloService helloService() {
        HelloService helloService = new HelloService();
        helloService.setHelloProperties(helloProperties);
        return helloService;
    }
}
```

在resources文件夹下创建META-INf文件夹

创建文件spring.factories

引入自动配置类

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  cn.allweing.starter.HelloServiceAutoConfiguration
```

14.3打包安装应用

先安装自动配置工程，在安装启动工程（install）

在其他工程中引入

```
        <exclusions>
            <exclusion>
                <groupId>org.junit.vintage</groupId>
                <artifactId>junit-vintage-engine</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>cn.allweing.starter</groupId>
        <artifactId>allweing-spring-boot-starter</artifactId>
        <version>1.0.1.SNAPSHOT</version>
    </dependency>
/dependencies>
```

配置文件

```
1
2   cn.allweing.prefix=prefix
3   cn.allweing.suffix=suffix
4
```

应用

```
    }

    @Autowired
    HelloService helloService;

    @GetMapping("test04")
    public String test04() { return helloService.getName("zzzgy"); }
}
```