其它注解

底层注解-@Configuration详解

- 基本使用
  - Full模式与Lite模式
  - 示例

```
/**
 * 1、配置类里面使用@Bean标注在方法上给容器注册组件，默认也是单实例的
 * 2、配置类本身也是组件
 * 3、proxyBeanMethods：代理bean的方法
 *      Full(proxyBeanMethods = true)（保证每个@Bean方法被调用多少次返回的组件都是单实例的）（默认）
 *      Lite(proxyBeanMethods = false)（每个@Bean方法被调用多少次返回的组件都是新创建的）
 */
@Configuration(proxyBeanMethods = false) //告诉SpringBoot这是一个配置类 == 配置文件
public class MyConfig {

    /**
     * Full:外部无论对配置类中的这个组件注册方法调用多少次获取的都是之前注册容器中的单实例对象
     * @return
     */
    @Bean //给容器中添加组件。以方法名作为组件的id。返回类型就是组件类型。返回的值，就是组件在容器中的实例
    public User user01(){
        User zhangsan = new User("zhangsan", 18);
        //user组件依赖了Pet组件
        zhangsan.setPet(tomcatPet());
        return zhangsan;
    }

    @Bean("tom")
    public Pet tomcatPet(){
        return new Pet("tomcat");
    }
}
```

@Configuration测试代码如下:

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan("com.atguigu.boot")
public class MainApplication {

    public static void main(String[] args) {
    //1、返回我们IOC容器
```

```
        ConfigurableApplicationContext run =
SpringApplication.run(MainApplication.class, args);

    //2、查看容器里面的组件
        String[] names = run.getBeanDefinitionNames();
        for (String name : names) {
            System.out.println(name);
        }

    //3、从容器中获取组件
        Pet tom01 = run.getBean("tom", Pet.class);
        Pet tom02 = run.getBean("tom", Pet.class);
        System.out.println("组件："+(tom01 == tom02));

    //4、com.atguigu.boot.config.MyConfig$$EnhancerBySpringCGLIB$$51f1e1ca@1654a892
        MyConfig bean = run.getBean(MyConfig.class);
        System.out.println(bean);

    //如果@Configuration(proxyBeanMethods = true)代理对象调用方法。SpringBoot总会检查这个
组件是否在容器中有。
        //保持组件单实例
        User user = bean.user01();
        User user1 = bean.user01();
        System.out.println(user == user1);

        User user01 = run.getBean("user01", User.class);
        Pet tom = run.getBean("tom", Pet.class);

        System.out.println("用户的宠物："+(user01.getPet() == tom));
    }
}
```

- 最佳实战
  - 配置 类组件之间**无依赖关系**用Lite模式加速容器启动过程，减少判断
  - 配置 类组件之间**有依赖关系**，方法会被调用得到之前单实例组件，用Full模式（默认）

底层注解-@Import导入组件

@Bean、@Component、@Controller、@Service、@Repository，它们是Spring的基本标签，在Spring Boot中并未改变它们原来的功能。

@Import({User.class, DBHelper.class})给容器中**自动创建(导入)出这两个类型的组件**、默认组件的名字就是全类名

```
@Import({User.class, DBHelper.class})
@Configuration(proxyBeanMethods = false) //告诉SpringBoot这是一个配置类 == 配置文件
public class MyConfig {
}
```

测试类：

```
//1、返回我们IOC容器
ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
args);

//...

//5、获取组件
String[] beanNamesForType = run.getBeanNamesForType(User.class);

for (String s : beanNamesForType) {
    System.out.println(s);
}

DBHelper bean1 = run.getBean(DBHelper.class);
System.out.println(bean1);
```

底层注解-@Conditional条件装配

条件装配：满足Conditional指定的条件，则进行组件注入

可以用在@Configuration注解声明的类上 或 @Bean注解声明的方法上

用@ConditionalOnMissingBean举例说明

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(name = "tom")//没有tom名字的Bean时，MyConfig类的Bean才能生效。
public class MyConfig {

    @Bean
    public User user01(){
        User zhangsan = new User("zhangsan", 18);
        zhangsan.setPet(tomcatPet());
        return zhangsan;
    }

    @Bean("tom22")
    public Pet tomcatPet(){
        return new Pet("tomcat");
    }
}

public static void main(String[] args) {
    //1、返回我们IOC容器
    ConfigurableApplicationContext run =
SpringApplication.run(MainApplication.class, args);

    //2、查看容器里面的组件
    String[] names = run.getBeanDefinitionNames();
    for (String name : names) {
        System.out.println(name);
    }
```

```
    boolean tom = run.containsBean("tom");
    System.out.println("容器中Tom组件："+tom);//false

    boolean user01 = run.containsBean("user01");
    System.out.println("容器中user01组件："+user01);//true

    boolean tom22 = run.containsBean("tom22");
    System.out.println("容器中tom22组件："+tom22);//true

}
```

底层注解-@ImportResource导入Spring配置文件

比如，公司使用bean.xml文件生成配置bean，然而你为了省事，想继续复用bean.xml，@ImportResource粉墨登场。

bean.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...">

    <bean id="haha" class="com.lun.boot.bean.User">
        <property name="name" value="zhangsan"></property>
        <property name="age" value="18"></property>
    </bean>

    <bean id="hehe" class="com.lun.boot.bean.Pet">
        <property name="name" value="tomcat"></property>
    </bean>
</beans>
```

使用方法：

```
@ImportResource("classpath:beans.xml")
@Configuration
public class MyConfig {
...
}
```

测试类：

```
public static void main(String[] args) {
    //1、返回我们IOC容器
    ConfigurableApplicationContext run =
SpringApplication.run(MainApplication.class, args);

    boolean haha = run.containsBean("haha");
    boolean hehe = run.containsBean("hehe");
    System.out.println("haha: "+haha);//true
    System.out.println("hehe: "+hehe);//true
}
```

Springboot简介

SpringBoot本身是为了加速Spring程序的开发的

- SpringBoot是**整合Spring技术栈**的一站式框架
- SpringBoot是**简化**Spring技术栈的快速开发脚手架

parent

parent自身具有很多个版本，每个**parent**版本中包含有几百个其他技术的版本号

你**无需关注这些技术间的版本冲突问题**，你只需要关注你用什么技术就行了，冲突问题由**parent**负责处理。

当开发者使用某些技术时，直接使用SpringBoot提供的**parent**就行了，由**parent帮助开发者统一的进行各种技术的版本管理**

引入的依赖不用指定版本。如若parent中没有收录依赖的版本号，则需要自己手动写依赖的版本号

starter

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

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

starter中定义了若干个具体依赖的坐标

引导类

```
@SpringBootApplication
public class Springboot0101QuickstartApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext ctx =
SpringApplication.run(Springboot0101QuickstartApplication.class, args);
        BookController bean = ctx.getBean(BookController.class);
        System.out.println("bean======>" + bean);
    }
}
```

SpringBoot程序启动还是创建了一个Spring容器对象。这个类在SpringBoot程序中是所有功能的入口，称这个类为**引导类**。

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

1. **SpringBoot工程提供引导类用来启动程序**
2. **SpringBoot工程启动后创建并初始化Spring容器**
3. **会扫描引导类所在包和子包创建bean对象，可以通过@ComponentScan注解指定要扫描的包**

内嵌tomcat

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

Springboot基础配置

属性配置

SpringBoot默认配置文件是application.properties

```
#关闭运行日志图表（banner）
spring.main.banner-mode=off
#设置运行日志的显示级别
logging.level.root=debug
```

- application.yml（**yml格式**）

```
server:
  port: 81
```

- application.yaml（yaml格式）

```
server:
  port: 82
```

yml和yaml文件格式就是一模一样的，只是文件后缀不同，所以可以合并成一种格式来看。

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

yaml文件

数据前面要加空格与冒号隔开

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

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

yaml数据读取

读取单一数据：

@Value(${一级属性名.二级属性名...})

读取全部数据：

```
@Autowired
private Environment env;
```

SpringBoot提供了一个对象，能够把所有的数据都封装到这一个对象中，这个对象叫做Environment

获取属性时，调用其方法getProperties（String）

读取对象数据：

使用**@ConfigurationProperties**注解绑定配置信息到封装类中

封装类需要定义为Spring管理的bean，否则无法进行属性注入

```
datasource:
  driver-class-name: xxx
  url: xxx
  username: xxx
  password: xxx
```

```
@Component
@ConfigurationProperties(prefix = "datasource")
public class DataSource{
    private String driverClassName;
    private String url;
    private String userName;
    private String password;
}
```

yaml文件中的数据引用

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

Springboot程序打包与运行

1. SpringBoot工程可以基于java环境下独立运行jar文件启动服务
2. SpringBoot工程通过执行maven命令package进行打包
3. 执行jar命令：java –jar 工程名.jar

配置高级-1

配置文件分类

- 类路径下配置文件（一直使用的是这个，也就是resources目录中的application.yml文件）

- 类路径下config目录下配置文件
- 程序jar包所在目录中配置文件
- 程序jar包所在目录中config目录下配置文件

4个文件的加载优先级顺序为

1. file： config/application.yml **【最高】**
2. file： application.yml
3. classpath：config/application.yml
4. classpath：application.yml **【最低】**

多层级配置文件间的属性采用**叠加并覆盖**的形式作用于程序

多环境开发

主配置文件application.yaml

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

application-pro.yaml

```yaml
server:
    port: 80
```

application-dev.yaml

```yaml
server:
    port: 81
```

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

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

加载使其生效，多个环境间使用逗号分隔

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

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

or

```yaml
spring:
    profiles:
        active: dev
        group:
            "dev": devDB,devRedis,devMVC
            "pro": proDB,proRedis,proMVC
            "test": testDB,testRedis,testMVC
#dev也会加载，且最先加载优先级最低
```

maven和SpringBoot同时设置多环境的话怎么搞?

SpringBoot应该听maven的。整个确认后下面就好做了。大体思想如下:

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

maven中设置多环境（使用属性方式区分环境）

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

SpringBoot中读取maven设置值

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

上面的@属性名@就是读取maven中配置的属性值的语法格式。

日志

添加日志

步骤①：添加日志记录操作

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

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

步骤②：设置日志输出级别

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

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

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

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

这么设置太简单粗暴了，日志系统通常都提供了细粒度的控制

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

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

还可以再设置更细粒度的控制

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

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

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

lombok

@Slf4j 添加注解

日志对象名为log

日志输出格式控制

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

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

日志文件

记录日志到文件中格式非常简单，设置日志文件名即可。

```yaml
logging:
    file:
        name: server.log
```

通常会每天记录日志文件，同时为了便于维护，还要限制每个日志文件的大小。下面给出日志文件的常用配置方式：

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

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

配置高级-2

@ConfigurationProperties

@ConfigurationProperties注解是写在类定义的上方，而第三方开发的bean源代码不是你自己书写的，你也不可能到源代码中去添加@ConfigurationProperties注解，这种问题该怎么解决呢？下面就来说说这个问题。

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

步骤①：使用@Bean注解定义第三方bean

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

步骤②：在yml中定义要绑定的属性，注意datasource此时全小写

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

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

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

@EnableConfigurationProperties

在一个业务系统中，哪些bean通过注解@ConfigurationProperties去绑定属性了呢？因为这个注解不仅可以写在类上，还可以写在方法上，所以找起来就比较麻烦了。

spring给我们提供了一个全新的注解，专门标注使用@ConfigurationProperties注解绑定属性的bean是哪些。这个注解叫做@EnableConfigurationProperties。

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

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

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

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

当使用@EnableConfigurationProperties注解时，spring会默认将其标注的类定义为bean，因此无需再次声明@Component注解了。

宽松绑定

在ServerConfig中的ipAddress属性名

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

可以与下面的配置属性名规则全兼容

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

也可以说，以上4种模式最终都可以匹配到ipAddress这个属性名。

在进行匹配时，配置中的名称要去掉中划线和下划线后，忽略大小写的情况下去与java代码中的属性名进行忽略大小写的等值匹配

springboot官方推荐使用烤肉串模式，也就是中划线模式。

即书写前缀只能用小写字母数字和中划线

以上规则仅针对springboot中@ConfigurationProperties注解进行属性绑定时有效，对@Value注解进行属性映射无效。

常用计量单位绑定

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

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

但是每个人都这个值的理解会产生不同，比如线上服务器完成一次主从备份，配置超时时间240，这个240如果单位是秒就是超时时间4分钟，如果单位是分钟就是超时时间4小时。

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

```java
@Component
@Data
@ConfigurationProperties(prefix = "servers")
public class ServerConfig {
    @DurationUnit(ChronoUnit.HOURS)
    private Duration serverTimeOut;
    @DataSizeUnit(DataUnit.MEGABYTES)
    private DataSize dataSize;
    //配置文件中配置这两个属性，用整型字面量，再用这两个注解指定单位
}
```

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

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

校验

目前我们在进行属性绑定时可以通过松散绑定规则在书写时放飞自我了，但是在书写时由于无法感知模型类中的数据类型，就会出现类型不匹配的问题，比如代码中需要int类型，配置中给了非法的数值，例如写一个“a"，这种数据肯定无法有效的绑定，还会引发错误。

步骤①：开启校验框架

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

步骤②：在需要开启校验功能的类上使用注解@Validated开启校验功能

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

步骤③：对具体的字段设置校验规则

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

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

测试

加载测试临时属性参数

在测试环境中创建一组临时属性，去覆盖我们源码中设定的属性，这样测试用例就相当于是一个独立的环境，能够独立测试

临时属性

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

```
//properties属性可以为当前测试用例添加临时的属性配置
@SpringBootTest(properties = {"test.prop=testValue1"})
public class PropertiesAndArgsTest {

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

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

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

临时参数

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

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

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

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

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

总结

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

加载测试专用配置

临时配置一些专用于测试环境的bean的需求

一个spring环境中可以设置若干个配置文件或配置类，若干个配置信息可以同时生效。现在我们的需求就是在测试环境中再添加一个配置类，然后启动测试环境时，生效此配置就行了

步骤①：在测试包test中创建专用的测试环境配置类

```
@Configuration
public class MsgConfig {
    @Bean
    public String msg(){
        return "bean msg";
    }
}
```

上述配置仅用于演示当前实验效果，实际开发可不能这么注入String类型的数据

步骤②：在启动测试环境时，导入测试环境专用的配置类，使用@Import注解即可实现

```
@SpringBootTest
@Import({MsgConfig.class})
public class ConfigurationTest {

    @Autowired
    private String msg;

    @Test
    void testConfiguration(){
        System.out.println(msg);
    }
}
/*
@Import只能用在类上 ，@Import通过快速导入的方式实现把实例加入spring的IOC容器中

加入IOC容器的方式有很多种，@Import注解就相对很牛皮了，@Import注解可以用于导入第三方包 ，当然@Bean注解也可以，但是@Import注解快速导入的方式更加便捷
*/
```

到这里就通过@Import属性实现了基于开发环境的配置基础上，对配置进行测试环境的追加操作，实现了1+1的配置环境效果。这样我们就可以实现每一个不同的测试用例加载不同的bean的效果，丰富测试用例的编写，同时不影响开发环境的配置。

总结

1. **定义测试环境专用的配置类，然后通过@Import注解在具体的测试中**导入临时的配置，例如测试用例，方便测试过程，且上述配置不影响其他的测试类环境

数据层测试回滚

在打包的阶段由于test生命周期属于必须被运行的生命周期，如果跳过会给系统带来极高的安全隐患，**所以测试用例必须执行。**但是新的问题就呈现了，**测试用例如果测试时产生了事务提交就会在测试过程中对数据库数据产生影响，进而产生垃圾数据。这个过程不是我们希望发生的**

在原始测试用例中添加注解@Transactional即可实现当前测试用例的事务不提交。当程序运行后，只要注解@Transactional出现的位置存在注解@SpringBootTest，springboot就会认为这是一个测试程序，无需提交事务，所以也就可以避免事务的提交。

```
@SpringBootTest
@Transactional
@Rollback(true)
public class DaoTest {
    @Autowired
    private BookService bookService;

    @Test
    void testSave(){
        Book book = new Book();
        book.setName("springboot3");
        book.setType("springboot3");
        book.setDescription("springboot3");

        bookService.save(book);
    }
}
```

如果开发者想提交事务，也可以，再添加一个@RollBack的注解，设置回滚状态为false即可正常提交事务

第三方技术整合

任务

任务系统指的是定时任务

Quartz

Quartz技术是一个比较成熟的定时任务框架，怎么说呢？有点繁琐，用过的都知道，配置略微复杂。springboot对其进行整合后，简化了一系列的配置，将很多配置采用默认设置，这样开发阶段就简化了很多。再学习springboot整合Quartz前先普及几个Quartz的概念。

- 工作（Job）：用于定义具体执行的工作
- 工作明细（JobDetail）：用于描述定时工作相关的信息
- 触发器（Trigger）：描述了工作明细与调度器的对应关系
- 调度器（Scheduler）：用于描述触发工作的执行规则，通常使用cron表达式定义规则

步骤①：导入springboot整合Quartz的starter

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

步骤②：定义任务Bean，按照Quartz的开发规范制作，继承QuartzJobBean

```
public class MyQuartz extends QuartzJobBean {
    @Override
    protected void executeInternal(JobExecutionContext context) throws
JobExecutionException {
        System.out.println("quartz task run...");
    }
}
```

步骤③：创建Quartz配置类，定义工作明细（JobDetail）与触发器的（Trigger）bean

```
@Configuration
public class QuartzConfig {
    @Bean
    public JobDetail printJobDetail(){
        //绑定具体的工作
        return JobBuilder.newJob(MyQuartz.class).storeDurably().build();
    }
    @Bean
    public Trigger printJobTrigger(){
        //指定了任务的执行时间
        ScheduleBuilder schedBuilder = CronScheduleBuilder.cronSchedule("0/5 * * * *
?");
        //绑定对应的工作明细
        return
TriggerBuilder.newTrigger().forJob(printJobDetail()).withSchedule(schedBuilder).buil
d();
    }
}
```

工作明细中要设置对应的具体工作，使用newJob()操作传入对应的工作任务类型即可。

触发器需要绑定任务，使用forJob()操作传入绑定的工作明细对象。此处可以为工作明细设置名称然后使用名称绑定，也可以直接调用对应方法绑定。触发器中最核心的规则是执行时间，此处使用调度器定义执行时间，执行时间描述方式使用的是cron表达式。

Task

Quartz将其中的对象划分粒度过细，导致开发的时候有点繁琐，spring针对上述规则进行了简化，开发了自己的任务管理组件——Task

步骤①：开启定时任务功能，在引导类上开启定时任务功能的开关，使用注解@EnableScheduling

```
@SpringBootApplication
//开启定时任务功能
@EnableScheduling
public class Springboot22TaskApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot22TaskApplication.class, args);
    }
}
```

步骤②：定义Bean，在对应要定时执行的操作上方，使用注解@Scheduled定义执行的时间，执行时间的描述方式还是cron表达式

```
@Component
public class MyBean {
    @Scheduled(cron = "0/1 * * * * ?")
    public void print(){
        System.out.println(Thread.currentThread().getName()+" :spring task run...");
    }
}
```

完事，这就完成了定时任务的配置。总体感觉其实什么东西都没少，只不过没有将所有的信息都抽取成bean，而是直接使用注解绑定定时执行任务的事情而已。

如何想对定时任务进行相关配置，可以通过配置文件进行

```
spring:
  task:
    scheduling:
      pool:
        size: 1                          # 任务调度线程池大小 默认 1
      thread-name-prefix: ssm_           # 调度线程名称前缀 默认 scheduling-
        shutdown:
          await-termination: false       # 线程池关闭时等待所有任务完成
          await-termination-period: 10s # 调度线程关闭前最大等待时间，确保最后一定关闭
```

监控

对软件的运行情况进行监督

监控服务状态是否处理宕机状态

监控服务运行指标

监控程序运行日志

管理服务状态

监控程序必须具有主动发起请求获取被监控服务信息的能力。

在被监控程序启动时上报监控程序，告诉监控程序你可以监控我了。看来需要在被监控程序端做主动上报的操作，这就要求被监控程序中配置对应的监控程序是谁

被监控程序可以提供各种各样的指标数据给监控程序看，但是每一个指标都代表着公司的机密信息，并不是所有的指标都可以给任何人看的，乃至运维人员，所以对被监控指标的是否开放出来给监控系统看，也需要做详细的设定。

总结

1. 监控是一个非常重要的工作，是保障程序正常运行的基础手段
2. 监控的过程通过一个监控程序进行，它汇总所有被监控的程序的信息集中统一展示
3. **被监控程序需要主动上报自己被监控，同时要设置哪些指标被监控**

可视化监控平台

Spring Boot Admin

包含有客户端和服务端两部分，而监控平台指的就是服务端。我们做的程序如果需要被监控，将我们做的程序制作成客户端

服务端开发

步骤①：导入springboot admin对应的starter，版本与当前使用的springboot版本保持一致，并将其配置成web工程

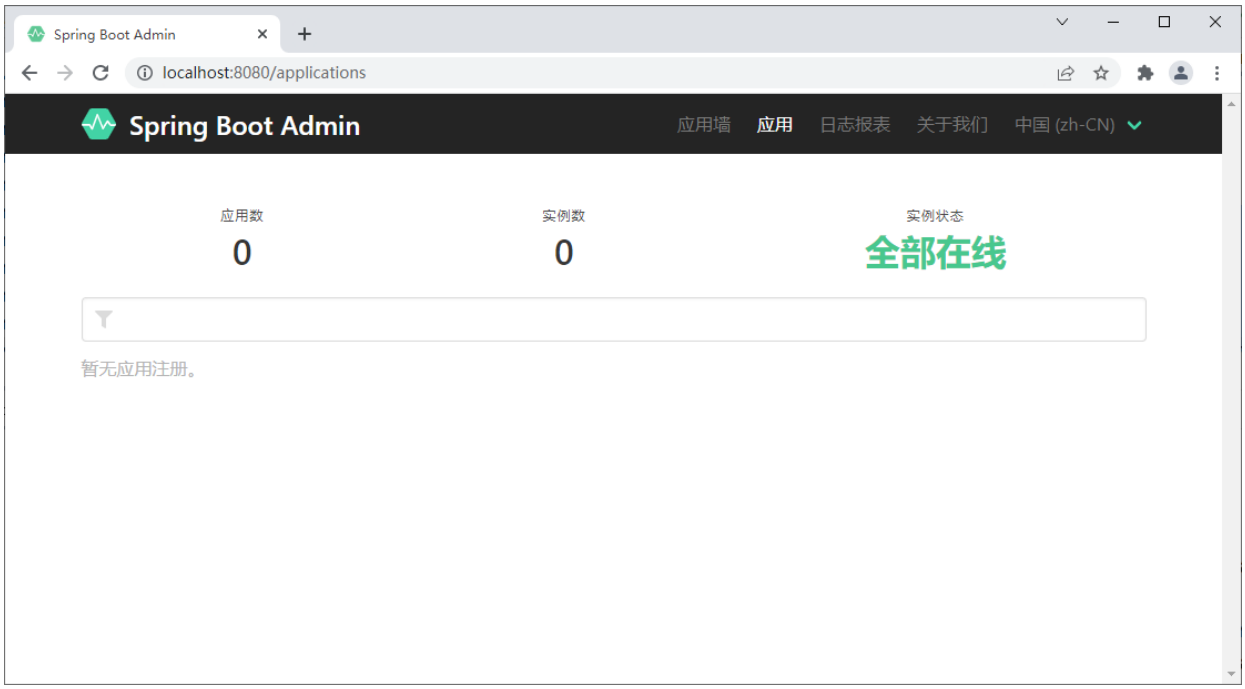
```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>2.5.4</version>
</dependency>

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

上述过程可以通过创建项目时使用勾选的形式完成。

步骤②：在引导类上添加注解@EnableAdminServer，声明当前应用启动后作为SpringBootAdmin的服务器使用

```
@SpringBootApplication
@EnableAdminServer
public class Springboot25AdminServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot25AdminServerApplication.class, args);
    }
}
```

做到这里，这个服务器就开发好了，启动后就可以访问当前程序了，界面如下。



客户端开发

客户端程序开发其实和服务端开发思路基本相似，多了一些配置而已。

步骤①：导入springboot admin对应的starter，版本与当前使用的springboot版本保持一致，并将其配置成web工程

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.5.4</version>
</dependency>

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

上述过程也可以通过创建项目时使用勾选的形式完成，不过一定要小心，端口配置成不一样的，否则会冲突。

步骤②：**设置当前客户端将信息上传到哪个服务器上**，通过yml文件配置

```
spring:
  boot:
    admin:
      client:
        url: http://localhost:8080
```

做到这里，这个客户端就可以启动了。启动后再次访问服务端程序

由于当前没有设置开放哪些信息给监控服务器，所以目前看不到什么有效的信息。下面需要做两组配置就可以看到信息了。

1. 开放指定信息给服务器看
2. 允许服务器以HTTP请求的方式获取对应的信息

配置如下：

```yaml
server:
  port: 80
spring:
  boot:
    admin:
      client:
        url: http://localhost:8080
management:
  endpoint:
    health:
      show-details: always
  endpoints:
    web:
      exposure:
        include: "*"
```

但是即便如此我们看到健康信息中也没什么内容，原因在于健康信息中有一些信息描述了你当前应用使用了什么技术等信息，如果无脑的对外暴露功能会有安全隐患。**通过配置就可以开放所有的健康信息明细查看了。**

```yaml
management:
  endpoint:
    health:
      show-details: always
```

目前**除了健康信息，其他信息都查阅不了。原因在于其他12种信息是默认不提供给服务器通过HTTP请求查阅的**，所以需要开启查阅的内容项，**使用*表示查阅全部。记得带引号。**

```yaml
endpoints:
  web:
    exposure:
      include: "*"
```

配置后再刷新服务器页面，就可以看到所有的信息了。

总结

1. **开发监控服务端需要导入坐标，然后在引导类上添加注解@EnableAdminServer，并将其配置成web程序即可**
2. **开发被监控的客户端需要导入坐标，然后配置服务端服务器地址，并做开放指标的设定即可**
3. **在监控平台中可以查阅到各种各样被监控的指标，前提是客户端开放了被监控的指标**

web场景-静态资源规则与定制化

静态资源目录

只要静态资源放在类路径下： called `/static` (or `/public` or `/resources` or `/META-INF/resources`

访问： 当前项目根路径/ + 静态资源名

原理： 静态映射/**。

请求进来，先去找Controller看能不能处理。不能处理的所有请求又都交给静态资源处理器。静态资源也找不到则响应404页面。

也可以改变默认的静态资源路径，`/static`， `/public`,`/resources`, `/META-INF/resources`失效

```yaml
resources:
  static-locations: [classpath:/haha/]
```

静态资源访问前缀

```yaml
spring:
  mvc:
    static-path-pattern: /res/**
```

当前项目 + static-path-pattern + 静态资源名 = 静态资源文件夹下找

欢迎页支持

- 静态资源路径下 index.html。
  - 可以配置静态资源路径
  - 但是不可以配置静态资源的访问前缀。否则导致 index.html不能被默认访问

```yaml
spring:
#  mvc:
#    static-path-pattern: /res/**   这个会导致welcome page功能失效
  resources:
    static-locations: [classpath:/haha/]
```

- controller能处理/index。

自定义Favicon

指网页标签上的小图标。

favicon.ico 放在静态资源目录下即可。

```yaml
spring:
#  mvc:
#    static-path-pattern: /res/**   这个会导致 Favicon 功能失效
```

拦截器-登录检查与静态资源放行

1. 编写一个拦截器实现 `HandlerInterceptor` 接口
2. 拦截器注册到容器中（实现 `WebMvcConfigurer` 的 `addInterceptors()` ）
3. 指定拦截规则（注意，如果是拦截所有，静态资源也会被拦截】

编写一个实现 `HandlerInterceptor` 接口的拦截器：

```
@Slf4j
public class LoginInterceptor implements HandlerInterceptor {

    /**
     * 目标方法执行之前
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {

        String requestURI = request.getRequestURI();
        log.info("preHandle拦截的请求路径是{}",requestURI);

        //登录检查逻辑
        HttpSession session = request.getSession();

        Object loginUser = session.getAttribute("loginUser");

        if(loginUser != null){
            //放行
            return true;
        }

        //拦截住。未登录。跳转到登录页
        request.setAttribute("msg","请先登录");
//        re.sendRedirect("/");
        request.getRequestDispatcher("/").forward(request,response);
        return false;
    }

    /**
     * 目标方法执行完成以后
     */
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) throws Exception {
        log.info("postHandle执行{}",modelAndView);
    }

    /**
     * 页面渲染以后
     */
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
```

```
        log.info("afterCompletion执行异常{}",ex);
    }
}
```

拦截器注册到容器中 && 指定拦截规则：

```
@Configuration
public class AdminWebConfig implements WebMvcConfigurer{
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoginInterceptor())//拦截器注册到容器中
                .addPathPatterns("/**")  //所有请求都被拦截包括静态资源

.excludePathPatterns("/","/login","/css/**","/fonts/**","/images/**",
                        "/js/**","/aa/**"); //放行的请求
}
```

文件上传-单文件与多文件上传的使用

- 控制层代码

```
@Slf4j
@Controller
public class FormTestController {

    @GetMapping("/form_layouts")
    public String form_layouts(){
        return "form/form_layouts";
    }

    @PostMapping("/upload")
    public String upload(@RequestParam("email") String email,
                         @RequestParam("username") String username,
                         @RequestPart("headerImg") MultipartFile headerImg,
                         @RequestPart("photos") MultipartFile[] photos) throws
IOException {

        log.info("上传的信息：email={}，username={}，headerImg={}，photos={}",
                 email,username,headerImg.getSize(),photos.length);

        if(!headerImg.isEmpty()){
            //保存到文件服务器，OSS服务器
            String originalFilename = headerImg.getOriginalFilename();
            headerImg.transferTo(new File("H:\\cache\\"+originalFilename));
        }

        if(photos.length > 0){
            for (MultipartFile photo : photos) {
                if(!photo.isEmpty()){
```

```
                    String originalFilename = photo.getOriginalFilename();
                    photo.transferTo(new File("H:\\cache\\"+originalFilename));
                }
            }
        }


        return "main";
    }
}
```

文件上传相关的配置类：

- `org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration`
- `org.springframework.boot.autoconfigure.web.servlet.MultipartProperties`

文件大小相关配置项：

```
spring.servlet.multipart.max-file-size=10MB
spring.servlet.multipart.max-request-size=100MB
```

单元测试-JUnit5

Spring Boot 2.2.0 版本开始引入 JUnit 5 作为单元测试默认库

JUnit 5官方文档

作为最新版本的JUnit框架，JUnit5与之前版本的JUnit框架有很大的不同。由三个不同子项目的几个不同模块组成。

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

- **JUnit Platform**: Junit Platform是在JVM上启动测试框架的基础，不仅支持Junit自制的测试引擎，其他测试引擎也都可以接入。
- **JUnit Jupiter**: JUnit Jupiter提供了JUnit5的新的编程模型，是JUnit5新特性的核心。内部包含了一个**测试引擎**，用于在Junit Platform上运行。
- **JUnit Vintage**: 由于JUint已经发展多年，为了照顾老的项目，JUnit Vintage提供了兼容JUnit4.x，JUnit3.x的测试引擎。

注意：

- SpringBoot 2.4 以上版本移除了默认对 Vintage 的依赖。如果需要兼容JUnit4需要自行引入（不能使用JUnit4的功能 @Test）
- JUnit 5's Vintage已经从`spring-boot-starter-test`从移除。如果需要继续兼容Junit4需要自行引入Vintage依赖：

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

- 使用添加Unit 5，添加对应的starter：

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

- Spring的JUnit 5的基本单元测试模板（Spring的JUnit4的是`@SpringBootTest`+`@RunWith(SpringRunner.class)`）：

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;//注意不是org.junit.Test（这是JUnit4版本的）
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class SpringBootApplicationTests {

    @Autowired
    private Component component;

    @Test
    //@Transactional 标注后连接数据库有回滚功能
    public void contextLoads() {
        Assertions.assertEquals(5, component.getFive());
    }
}
```

单元测试-常用测试注解

- **@Test**：表示方法是测试方法。但是与JUnit4的@Test不同，他的职责非常单一不能声明任何属性，拓展的测试将会由Jupiter提供额外测试
- **@ParameterizedTest**：表示方法是参数化测试。
- **@RepeatedTest**：表示方法可重复执行。
- **@DisplayName**：为测试类或者测试方法设置展示名称。

- **@BeforeEach**：表示在**每个**单元测试**之前**执行。
- **@AfterEach**：表示在**每个**单元测试**之后**执行。
- **@BeforeAll**：表示在**所有**单元测试**之前**执行。
- **@AfterAll**：表示在**所有**单元测试**之后**执行。
- **@Tag**：表示单元测试类别，类似于JUnit4中的@Categories。
- **@Disabled**：表示测试类或测试方法不执行，类似于JUnit4中的@Ignore。
- **@Timeout**：表示测试方法运行如果超过了指定时间将会返回错误。
- **@ExtendWith**：为测试类或测试方法提供扩展类引用。

单元测试-断言机制