简介

SpringBoot 帮我们简单、快速地创建一个独立的、生产级别的 Spring 应用；

特性：

- 快速创建独立 Spring 应用
- 直接嵌入Tomcat、Jetty or Undertow
- 提供可选的 starter，简化应用整合
- 按需自动配置 Spring 以及 第三方库
- 提供生产级特性：如 监控指标、健康检查、外部化配置等
- 无代码生成、无xml； 都是基于自动配置技术

场景启动器

SpringBoot场景启动器：官方写的启动器命名：`spring-boot-starter-*`。第三方启动器命名：`*-spring-boot-starter`

> **作用：**场景启动器负责把当前场景需要用的jar包都导入进来
>
> 每个场景启动器都有一个基础依赖：spring-boot-starter

依赖管理

为什么项目依赖不需要写版本号？---maven父子继承，父项目可以锁定版本。

父项目不管理的包都需要添加版本号

*自动配置

基本理解

- 自动配置

  导入场景，容器中就会自动配置好这个场景的核心组件。
  如 Tomcat、SpringMVC、DataSource 等
  不喜欢的组件可以自行配置进行替换。

- 默认的包扫描规则
  SpringBoot只会扫描主程序所在的包及其下面的子包

- 配置默认值
  配置文件的所有配置项 是和 某个类的对象值进行一一绑定的。

  很多配置即使不写都有默认值，如：端口号，字符编码等
  默认能写的所有配置项：https://docs.spring.io/spring-boot/appendix/application-properties/index.html

完整流程

核心流程总结：
1: 导入 starter，就会导入autoconfigure 包。

2: autoconfigure 包里面 有一个文件 META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports,里面指定的所有启动要加载的自动配置类(xxxAutoConfiguration)

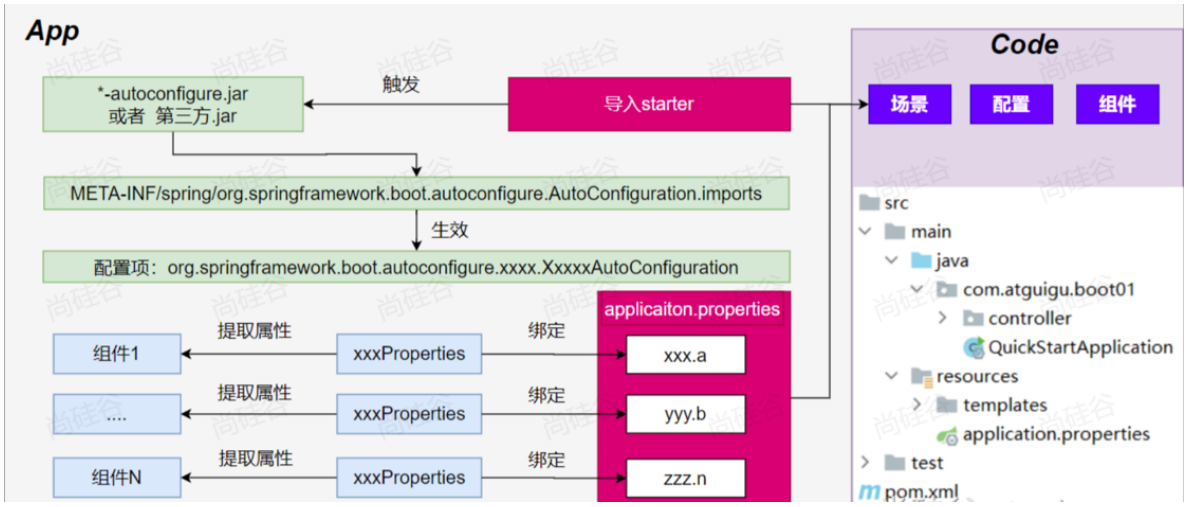
3: xxxAutoConfiguration 基于@Conditional给容器中导入一堆组件

4: 组件都是从 xxxProperties 中提取属性值

5: xxxProperties 又是和配置文件进行了绑定

效果：

导入starter、修改配置文件，就能修改底层行为。



基础使用

属性绑定

将容器中任意组件的属性值和配置文件的配置项的值进行绑定

1、给容器中注册组件（@Component、@Bean）
2、使用 @ConfigurationProperties 声明组件和配置文件的哪些配置项进行绑定

```
//读取前缀为dog的配置文件属性
@ConfigurationProperties(prefix = "dog")
@Component
@Data
public class DogProperties {
    private String name;
    private Integer age;
    private String gender;

}
```

properties:

```
dog.name=旺财
dog.age=2
dog.gender=男
```

banner设置

https://www.bootschool.net/ascii

properties里配置：`spring.banner.location=classpath:banner.txt`即可自定义banner

启动spring应用的其他方式

```
//创建SpringApplication对象
SpringApplication springApplication = new
SpringApplication(Springboot01DemoApplication.class);
//启动
springApplication.run(args);
```

在中间可以插入一些设置

```
//创建SpringApplication对象
SpringApplication springApplication = new
SpringApplication(Springboot01DemoApplication.class);

//关闭banner
//springApplication.setBannerMode(Banner.Mode.OFF);
//设置监听器
//springApplication.setListeners();
//设置环境
//springApplication.setEnvironment();

//启动
springApplication.run(args);
```

日志

springboot默认使用slf4j+logback

日志记录

在类上加上@Slf4j注解，会给我们一个日志对象log

然后就可以加日志:

```
//记录日志
log.debug("调试日志");
log.error("错误日志");
log.info("信息日志");
```

根据级别记录日志

级别由低到高: ALL=>TRACE=>DEBUG=>INFO=>WARN=>ERROR=>OFF

由低到高越来越不详细

日志有一个默认级别（INFO），只会打印这个级别 (包括) 之上的所有信息

调整默认级别:

```
logging.level.root=debug
```

也可以只调整某个包下的默认级别:

```
logging.level.全包名=info
```

日志分组

```
logging.group.biz=包名，包名

logging.level.biz=级别
```

SpringBoot 预定义两个组:

组名	范围
web	org.springframework.core.codec, org.springframework.http, org.springframework.web, org.springframework.boot.actuate.endpoint.web, org.springframework.boot.web.servlet.ServletContextInitializerBeans
sql	org.springframework.jdbc.core, org.hibernate.SQL, org.jooq.tools.LoggerListener

*输出到文件

```
# 当前项目的根文件夹下生成一个指定名字的日志文件
logging.file.name=boot.log
```

归档与切割

归档：每天的日志单独存到一个文档中。
切割：每个文件10MB，超过大小切割成另外一个文件。

默认滚动切割与归档规则如下：

配置项	描述
logging.**logback**.rollingpolicy.file-name-pattern	日志存档的文件名格式 默认值：${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz
logging.**logback**.rollingpolicy.clean-history-on-start	应用启动时是否清除以前存档；默认值：false
logging.**logback**.rollingpolicy.max-file-size	每个日志文件的最大大小；默认值：10MB
logging.**logback**.rollingpolicy.total-size-cap	日志文件被删除之前，可以容纳的最大大小（默认值：0B）。设置1GB则磁盘存储超过 1GB 日志后就会删除旧日志文件
logging.**logback**.rollingpolicy.max-history	日志文件保存的最大天数；默认值：7

补充:{}表示占位符

自定义配置

日志系统	自定义
Logback	logback-spring.xml / logback.xml
Log4j2	log4j2-spring.xml / log4j2.xml
JDK (Java Util Logging)	logging.properties

切换日志系统

在本模块的pom.xml中导入spring-boot-starter依赖并在其中用exclusion标签排除原本的日志系统。
之后在`<dependencies>`导入另外的日志系统。

总结

我们用日志：

1.配置（日志输出到文件、打印日志级别）

2.不同的时候选择合适的级别进行日志记录

3.用日志取代`sout`

进阶

profiles环境隔离

环境隔离：

1.定义环境：dev、test、prod等

2.定义这个环境下生效哪些组件或者哪些配置

1）、生效哪些组件：给组件@profile

2）、生效那些配置：application-dev.properties

3.激活环境，环境下的组件和配置就会生效

1）、application.properties中配置spring.profiles.active=dev

2）、命令行：java -jar xxx.jar --spring.profiles.active=dev

所以以后生效的配置=默认配置+激活的配置(spring.profiles.active)+包含的配置(spring.profiles.include)

profiles也可以分组

`spring.profiles.group.组名[索引]=环境`。在激活时，激活组名

外部化配置

打包完的jar包如果要修改配置，如果在内部修改，就需要重新打包，很麻烦。可以采用**外部化配置**

优先级：

jar包中的配置<jar包同源路径下的配置文件<同源路径下的config（一定是这个名字）文件夹里的配置<config里的任意名字的文件夹里的配置文件<命令行

所以配置文件，外部优先，激活优先。外部的默认和内部的激活同时配置了某个属性，用内部激活的。

单元测试断言机制

目前的单元测试在有异常时才会被判定为失败。

但是我们需要的是测试结果不符合业务，就算测试失败。

断言机制就是对测试结果进行预测，如果测试结果与预测不符，就判定失败。

使用：

```
Assertions.断言方法（）
```

方法	说明
assertEquals	判断两个对象或两个原始类型是否相等
assertNotEquals	判断两个对象或两个原始类型是否不相等
assertSame	判断两个对象引用是否指向同一个对象
assertNotSame	判断两个对象引用是否指向不同的对象
assertTrue	判断给定的布尔值是否为 true
assertFalse	判断给定的布尔值是否为 false
assertNull	判断给定的对象引用是否为 null
assertNotNull	判断给定的对象引用是否不为 null
assertArrayEquals	数组断言
assertAll	组合断言
assertThrows	异常断言
assertTimeout	超时断言
fail	快速失败

自定义starter

一般将配置文件中的应用名改为 *-spring-boot-starter形式

自定义starter是不需要主程序的

我们需要定义一个自动配置类，将所有组件导入容器。并且在resources目录下定义META-INF.spring包，创建一个org.springframework.boot.autoconfigure.AutoConfiguration.imports文件。把自动配置类的全类名写入。

这样，别人在引入starter依赖就不用进行操作就可以使用组件了。

当然一些组件的属性也可能需要别人自己配置（类似于jdbc的username等）

需要掌握的源码

1.springboot自动配置原理
2.springmvc　dispatcherservlet流程
3.spring　ioc容器三级缓存机制
4.spring 事务原理（transactionManager、transactioninterceptor）