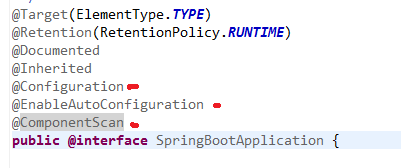
# 注解

## @SpringBootApplication

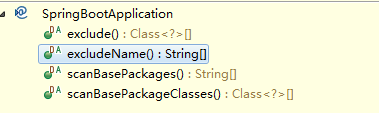
|  |
| --- |
| The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes. |

|  |
| --- |
| @SpringBootApplication also provides aliases to customize the attributes of @EnableAutoConfiguration and @ComponentScan. |

SpringBootApplication继承了Configuration、ComponentScan、EnableAutoConfiguration 见下图：



SpringBootApplication定义



SpringBootApplication提供的属性

## @EnableAutoConfiguration

|  |
| --- |
| You should only ever add one @EnableAutoConfiguration annotation. We generally recommend that you add it to your primary @Configurationclass. |

添加了该注解的bean所在的package,spring会作为扫描包进行扫描

## @Import

用于引入Configuration Class,并且通过该注解引入的配置bean不需要添加@Configuration注解

## @ImportResource

用于引入XML/groovy配置文件

## @ConfigurationProperties

该注解用于代替@Value注解,当一个类中有多个参数需要注入式,该注解比较便捷

注意:如果使用该注解需要两个条件,

1、使用@EnableConfigurationProperties注解开启该功能

2、将添加该注解的类纳入spring管理，有多种方式可以实现将bean纳入spring管理

* 1. 添加component注解
  2. 使用@Bean注解
  3. 在xml中定义bean
  4. 使用@EnableConfigurationProperties的属性引入该类（仅适用此处讨论的注解）

## @PropertySource

该注解用于代替PropertyPlaceholderConfigurer,使用注解的形式引入资源文件

# 自动配置auto-configuration

## auto-configuration

|  |
| --- |
| Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added |

## 替换auto-configuration

|  |
| --- |
| Auto-configuration is noninvasive(非侵略性的), at any point you can start to define your own configuration to replace specific parts of the auto-configuration. For example, if you add your own DataSource bean, the default embedded database support will back away.  If you need to find out what auto-configuration is currently being applied, and why, start your application with the --debug switch. This will enable debug logs for a selection of core loggers and log an auto-configuration report to the console. |

## 使指定的自动配置失效

|  |
| --- |
| If you find that specific auto-configure classes are being applied that you don’t want, you can use the exclude attribute of @EnableAutoConfiguration to disable them.  import org.springframework.boot.autoconfigure.\*;  import org.springframework.boot.autoconfigure.jdbc.\*;  import org.springframework.context.annotation.\*;  *@Configuration*  *@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})*  public class MyConfiguration {  }  //通过注解属性使指定配置失效  If the class is not on the classpath, you can use the excludeName attribute of the annotation and specify the fully qualified name instead.  //通过配置文件使指定配置失效  Finally, you can also control the list of auto-configuration classes to exclude via the spring.autoconfigure.exclude property. |

# 依赖管理

## Inheriting the starter parent

**step 1:**

继承spring-boot-starter-parent

To configure your project to inherit from the  spring-boot-starter-parent simply set the parent:

|  |
| --- |
| <!-- Inherit defaults from Spring Boot -->  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.3.RELEASE</version>  </parent> |

**step 2:**

添加需要的starter

## Using Spring Boot without the parent POM

使用依赖管理标签添加spring-boot-dependencies,并指定scope为import

注意:spring-boot-dependencies中不包含maven插件

|  |
| --- |
| <dependencyManagement>  <dependencies>  <dependency>  <!-- Import dependency management from Spring Boot -->  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>1.4.3.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

使用dependencyManagement时,如果向指定某个jar包使用指定的版本,需要在spring-boot-dependencies之前指定,如指定spring-data的版本

|  |
| --- |
| <dependencyManagement>  <dependencies>  <!-- Override Spring Data release train provided by Spring Boot -->  <dependency>  <groupId>org.springframework.data</groupId>  <artifactId>spring-data-releasetrain</artifactId>  <version>Fowler-SR2</version>  <scope>import</scope>  <type>pom</type>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>1.4.3.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

# 注入

## 构造注入

构造注入的一个好处是可以给final类型的变量注入

using constructor injection allows the riskAssessor field to be marked as final, indicating that it cannot be subsequently changed

|  |
| --- |
| package com.example.service;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  *@Service*  public class DatabaseAccountService implements AccountService {  private final RiskAssessor riskAssessor;  *@Autowired*  public DatabaseAccountService(RiskAssessor riskAssessor) {  this.riskAssessor = riskAssessor;  }  // ...  } |

## 属性注入

使用@Autowired根据类型注入spring中管理的其他bean

使用@Value(在属性上使用)或@ ConfigurationProperties(用在类上)将配置文件中的属性注入到类中

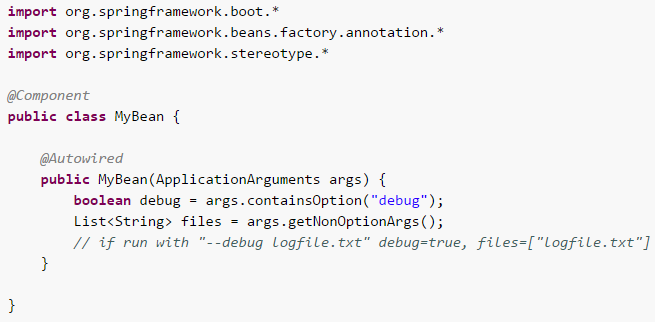
# SpringApplication

## 构造方法

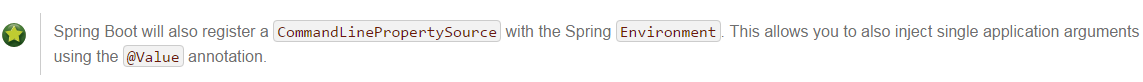
SpringApplication的构造方法可以接收一个@Configuration Class或者XML地址.或者需要扫描的包

## arguments

在程序中可以通过注入一个ApplicationArguments对象访问SpringApplication.run(..)方法中传入的参数



|  |
| --- |
| Spring Boot will also register a CommandLinePropertySource with the Spring Environment. This allows you to also inject single application arguments using the @Value annotation. |



## Application events and listeners

除了传统的spring框架中的事件,SpringApplication会发送一些额外的事件,按先后顺序依次为:

* An ApplicationStartedEvent is sent at the start of a run, but before any processing except the registration of listeners and initializers.

用户自定义的该事件的Listener,如果在其中打印日志,那么日志将会被屏蔽,因为在ApplicationStartedEvent和ApplicationEnvironmentPreparedEvent事件之间SpringBoot会对日志系统根据用户在配置文件中的配置进行配置

* An ApplicationEnvironmentPreparedEvent is sent when the Environment to be used in the context is known, but before the context is created.
* An ApplicationPreparedEvent is sent just before the refresh is started, but after bean definitions have been loaded.
* An ApplicationReadyEvent is sent after the refresh and any related callbacks have been processed to indicate the application is ready to service requests.
* An ApplicationFailedEvent is sent if there is an exception on startup.

## ApplicationRunner 和 CommandLineRunner

这两个组件的功能类似于监听器,在SpringApplication.run(...)方法完成之前调用实现了这两个接口的bean中的run方法

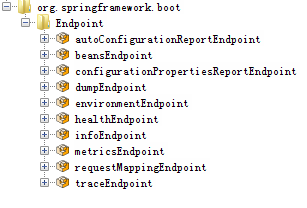
ApplicationRunner的run方法有一个ApplicationArguments参数

CommandLineRunner的run方法有一个String数组,该数组的值即为命令行输入的参数

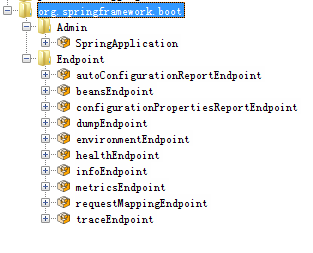
如果存在多个runner,可以通过实现order接口或使用@Order注解指定多个runner的执行顺序

## Admin Feature

通过指定spring.application.admin.enabled属性,可以暴露[SpringApplicationAdminMXBean](https://github.com/spring-projects/spring-boot/tree/v1.4.3.RELEASE/spring-boot/src/main/java/org/springframework/boot/admin/SpringApplicationAdminMXBean.java),这样管理员就可以通过远程JMX管理application了



不启用admin future的MBean



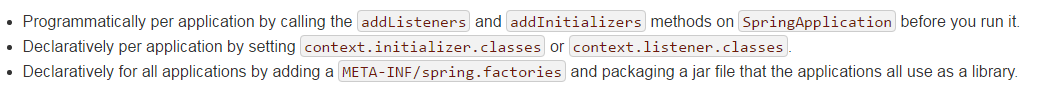
启用admin future后的MBean

## 定制ApplicationContext

可以通过ApplicationListeners和ApplicationContextInitializers定制化ApplicationContext

|  |
| --- |
| A SpringApplication has ApplicationListeners and ApplicationContextInitializers that are used to apply customizations to the context or environment. Spring Boot loads a number of such customizations for use internally from META-INF/spring.factories. |

将ApplicationListeners和ApplicationContextInitiallizers配置到spring的方式有一下几种:



## EnvironmentPostProcessor

You can customize the Environment before the application context is **refreshed** using EnvironmentPostProcessor. Each implementation should be registered in META-INF/spring.factories:

org.springframework.boot.env.EnvironmentPostProcessor=com.example.YourEnvironmentPostProcessor

# configuration

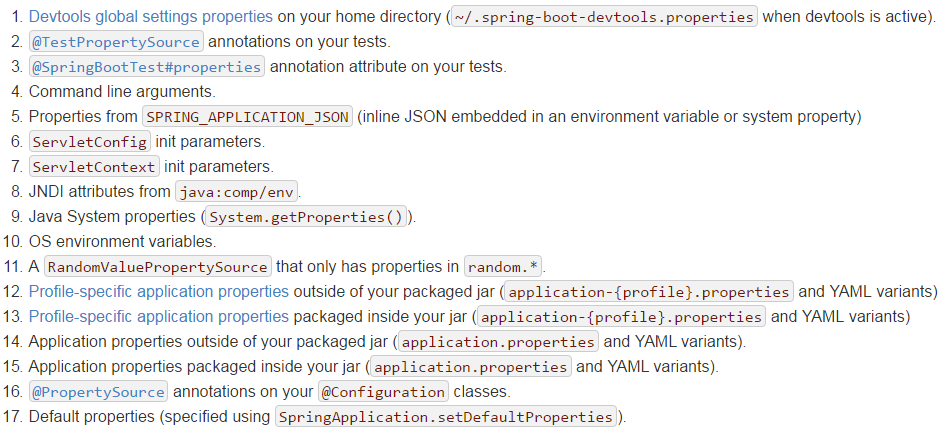
可以通过外置配置文件

## spring boot支持的属性列表

可以在spring boot官方文档中的Appendices中的common application properties中可以找到

## 配置的方式及优先级

通过下列方式配置的属性,1-17,数字越小优先级越高



## 通过json传递参数

* **设置环境变量**

通过添加SPRING\_APPLICATION\_JSON到环境变量来指定要传递的json串

* **设置系统参数**

通过-D设置system variable这种方式与System.setProperties()方法作用一致



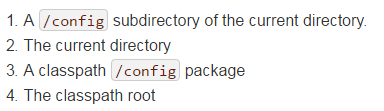
* **设置命令行参数**

通过给main函数传递参数指定json



## 默认查找配置文件的位置

spring会在一下四个位置查找application.properties或指定的profile文件



## 指定配置文件名称和存放位置

spring.config.name 指定配置文件的名称

spring.config.location 指定配置文件存放路径

这两个属性需要在OS env, system property or command line argument中设置

因为这两个属性被加载的时机很早.如果通过环境变量进行配置,可以设置SPRING\_CONFIG\_NAME代替spring.config.name

即使设置了这两个参数,系统还是回去默认的位置查找配置文件,不过自定义的配置文件中的属性会覆盖默认位置中设置的属性

如果指定了location,那么默认位置的配置文件也会起作用,指定位置的配置文件中的属性比默认位置配置文件中的属性优先级高

如果指定了config.name,那么application.properties就失效了

## 配置文件的优先级

* 指定location处的配置比默认位置处的配置优先级高
* profile的优先级比不带profile的配置的优先级高
* 默认位置处包含profile的配置比指定location处不带profile的配置的优先级高

## 配置加载

PropertySourceLoader负责从配置文件中解析出这些配置信息,然后EnvironmentPostProcessor负责向Environment中添加配置信息

**具体过程如下:**

* 加载配置的源头在监听器,通过SpringApplicationRunListener来控制spring boot中特有事件的触发
* ConfigFileApplicationListener监听ApplicationEnvironmentPreparedEvent事件
* 当监听到该事件后先加载所有的EnvironmentPostProcessor实现类(通过SpringFactoriesLoader加载所有META-INFO目录下spring.factories文件中的EnvironmentPostProcessor实现类实现并将自己也作为实现类加到实现类列表中)
* 调用实现类列表中所有对象的postProcessEnvironment方法,其中ConfigFileApplicationListener的postProcessEnvironment方法会使用SpringFactoriesLoader查找spring.factories文件中所有PropertySourceLoader实现类
* PropertySourceLoader负责从配置文件中加载配置信息

## profile-specific properties

In addition to application.properties files, profile-specific properties can also be defined using the naming convention application-{profile}.properties. The Environment has a set of default profiles (by default [default]) which are used if no active profiles are set.

Profile-specific properties are loaded from the same locations as standard application.properties, with profile-specific files always overriding the non-specific ones irrespective of whether the profile-specific files are inside or outside your packaged jar.

profile-specific文件优先级比non-specific文件高,不管profile-specific 文件是在jar包外还是在jar包内

if several profiles are specified ,a last wins strategy applies.

如果多个profile被激活,那么后激活的profile将覆盖之前激活的profile中的配置.

## Placeholders in properties

The value in application.properties are filtered through the existing Environment when they are used to you can refer back to previously defined values(e.g. from System properties).

application.properties中可以引用在Environment中已经存在的变量,比如系统变量

|  |
| --- |
| app.name=MyApp  app.description=${app.name} is a Spring Boot Application |

## 使用yaml配置spring boot

如果想使用yaml作为spring boot的配置文件,只需要在classpath下引入SnakeYAML的依赖,spring-boot-starter中自动引入了该依赖

yaml文件和properties文件对于文件名和文件位置的配置方式相同,也可以指定profile-specificed文件

yaml有一些properties不支持的功能:在配置文件内指定profile配置

## 在配置文件中使用随机值

RandomValuePropertySource用来注入随机值,在配置文件中使用${random.\*}取随机值

|  |
| --- |
| my.secret=${random.value}  my.number=${random.int}  my.bignumber=${random.long}  my.uuid=${random.uuid}  my.number.less.than.ten=${random.int(10)}  my.number.in.range=${random.int[1024,65536]} |

## 在配置文件中配置集合

# 内置容器

spring boot通过内置容器的方式提供web服务,如果使用内置容器只需要将内置容器的相关依赖添加到classpath下即可.

spring默认使用tomcat作为内置容器.

## Servlet Context Initialization

如果想在spring boot项目中影响servlet context的初始化过程,你只需要向spring注册一个org.springframework.boot.context.embedded.ServletContextInitializer接口的实现类,该类只有一个方法onStartup(),在该方法中可以操作ServletContext

## 注册Servlets,Filters和Listeners

可以通过扫描的方式注册Servlet,Filter和Listener组件,只要两步

1. 为Servlet,Filter,Listener添加相应的注解(@WebServlet、@WebFilter、@WebListener)
2. 使用@ServletComponentScan注解指定扫描的包

注意：@ServletComponentScan不会对独立运行的容器起作用，如单独启动tomcat运行war包

## 定制内置容器

定制内置容器有三种途径：

1. 配置文件(如application.properties)
2. EmbeddedServletContainerCustomizer
3. ConfigurableEmbeddedServletContainer

### 配置文件

在application.properties中可以进行一些基本的设置,例如server.port,server.address

server.session.persistence server.session.timeout 等

### EmbeddedServletContainerCustomizer

向spring注册一个EmbeddedServletContainerCustomizer实现类组件,在customize方法中可以对container进行操作

|  |
| --- |
| @Component  **public** **class** CustomizationBean **implements** EmbeddedServletContainerCustomizer {  @Override  **public** **void** customize(ConfigurableEmbeddedServletContainer container) {  container.setPort(8070);  }  } |

### EmbeddedServletContainerFactory

在spring容器中可以注册EmbeddedServletContainerFactory实现类,在spring boot启动时会自动覆盖默认的EmbeddedServletContainerFactory.EmbeddedServletContainerFactory只有一个方法

|  |
| --- |
| @Bean  **public** EmbeddedServletContainerFactory servletContainer(){  TomcatEmbeddedServletContainerFactory factory = **new** TomcatEmbeddedServletContainerFactory();  factory.setPort(8060);  factory.setSessionTimeout(30, TimeUnit.***MINUTES***);  **return** factory;  } |

# DataSource

有两个比较关键的类**DataSourceAutoConfiguration**和**DataSourceConfiguration**

spring boot中DataSource的配置工作在**DataSourceAutoConfiguration**中进行

在**DataSourceConfiguration**有一些内部类如Tomcat,DBCP等,这些类负责自动创建响应的连接池,并识别spring.datasource.tomct.\*或spring.datasource.dbcp.\*等配置

|  |
| --- |
| **//这个conditional的作用就是查看有没有创建pooledDataSource的条件,如果有EmbeddedDatabaseConfiguration就不起作用,如果没有查看有没有创建内存数据库的条件,比如有没有内存数据库所需要的依赖**  @Conditional(EmbeddedDatabaseCondition.**class**)  @ConditionalOnMissingBean({ DataSource.**class**, XADataSource.**class** })  @Import(EmbeddedDataSourceConfiguration.**class**)  **protected** **static** **class** EmbeddedDatabaseConfiguration {  }  @Configuration  **//这个conditional的作用就是查看有没有创建PooledDataSource所需要的资源**  @Conditional(PooledDataSourceCondition.**class**)  @ConditionalOnMissingBean({ DataSource.**class**, XADataSource.**class** })  @Import({ DataSourceConfiguration.Tomcat.**class**, DataSourceConfiguration.Hikari.**class**, DataSourceConfiguration.Dbcp.**class**, DataSourceConfiguration.Dbcp2.**class**,  DataSourceConfiguration.Generic.**class** })  **protected** **static** **class** PooledDataSourceConfiguration {  } |

## 嵌入式数据库

判断是否使用嵌入式数据库的逻辑是在**DataSourceAutoConfiguration.EmbeddedDatabaseCondition**中定义的,在该类中的getMatchOutcome方法中先判断是否有使用PooledDataSource的条件,如果没有在查看有哪些可以使用的EmbeddedDatabase,如果没有就不创建

|  |
| --- |
| @Override  **public** ConditionOutcome getMatchOutcome(ConditionContext context,  AnnotatedTypeMetadata metadata) {  ConditionMessage.Builder message = ConditionMessage  .*forCondition*("EmbeddedDataSource");  **if** (anyMatches(context, metadata, **this**.pooledCondition)) {  **return** ConditionOutcome.*noMatch*(message  .foundExactly("supported pooled data source"));  }  EmbeddedDatabaseType type = **EmbeddedDatabaseConnection**  **.*get*(context.getClassLoader()).getType();**  **if** (type == **null**) {  **return** ConditionOutcome.*noMatch*(message  .didNotFind("embedded database").atAll());  }  **return** ConditionOutcome.*match*(message.found("embedded database")  .items(type));  } |

查看有哪些可用的EmbeddedDatabase的逻辑是在**EmbeddedDatabaseConnection**中定义的

该类中也定义了支持哪些内存数据库,以及内存数据库使用的优先级

* spring boot支持嵌入式数据库H2,HSQL和Derby,配置起来很简单,只需要引入你想用的数据库相关的依赖即可,不需要配置任何东西,地址也不需要.
* 如果测试的时候在多个项目中使用嵌入式数据库,那么默认情况下这几个项目将共用一个数据库实例,如果不同的项目需要不同的数据库实例,那么需要设置spring.datasource.generate-unique-name为true
* 注意:不管是处于什么原因,如果你为嵌入式数据库配置了地址,那么,你需要将数据库的自动关闭功能关掉,这样spring boot才能控制什么时候关闭该数据库

## 生产数据库连接池

* 在生产环境下,spring boot会自动创建数据库连接池,如果在classpath下包含多个数据库连接池产品的依赖,那么spring boot选择连接池的先后循序是:
* Tomcat-jdbc-->HikariCP-->DBCP-->DBCP2
* 如果你想手动指定使用某个连接池产品,可以通过spring.datasource.type设置需要使用的连接池的Class全路径名
* 如果使用spring-boot-starter-jdbc或者spring-boot-starter-data-jpa, 'starters'将会自动引入tomcat-jdbc的依赖
* 一般不需要指定driver-class-name,因为spring会根据url自动判断

## 数据库初始化

在一些demo级别的项目,使用代码初始化数据库是一个不错的选择,初始化数据库有两种常见的途径:

### 使用hibernate的初始化功能

hibernate在启动的时候会自动加载classpath根目录下import.sql文件,这是hibernate的特性和spring boot无关,但是在spring boot下如果想要使用这一特性,需要设置spring.jpa.hibernate.ddl-auto为create或者create-drop.默认情况下,spring boot会检测使用的数据库是内置的数据库还是外置数据库,如果是内置数据库,那么,spring.jpa.hibernate.ddl-auto为create-drop否则为none

spring.jpa.hibernate.ddl-auto的可选值为none, validate, update, create, create-drop

### 使用spring jdbc的初始化功能

spring jdbc有初始化数据源的特性,spring boot默认开启该功能,默认加载classpath根目录下的schema.sql和data.sql(除此之外还有schema-{platform}.sql和data-{platform}.sql).如果向改变spring jdbc查找sql脚本的位置可以使用spring.datasource.schema来指定脚本存放位置.

初始化功能默认是开启的,如果向关闭初始化功能可以使用spring.datasource.initialize=false

# 静态内容

## 默认静态资源路径

默认情况下spring boot会从/static(或者/public或/resources或/META-INF/resources)目录下查找静态资源

你可以通过spring.resources.staticLocations来配置默认的静态资源查找目录

如果你在上面提到的目录下放置一个index.html,那么这个页面会成为欢迎页面

## 添加静态资源和路径的映射关系

有时候我们可能希望某个访问路径下的请求和我们的某个目录映射到一块,比如/mystatic/\*路径下的资源都去classpath:/mystatic/路径下去找,那么我们可以自定义一个

WebMvcConfigurerAdapter的实现类重写addResourceHandlers方法,代码如下:

|  |
| --- |
| @Override  **public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {  registry.addResourceHandler("/mystatic/\*\*")  .addResourceLocations("classpath:/mystatic/");  } |

这种是和默认配置累加的，现在默认配置也可以使用，但resourceHanler不能写成“/\*\*”，这是默认配置，如果自定义了就会覆盖，非要写成这样也可以，把默认路径也添加到location中就行，addResourceLocation是动参，可以这样写：

|  |
| --- |
| registry.addResourceHandler("/\*\*").addResourceLocations("classpath:/mystatic/","classpath:/resources/","classpath:/META-INF/resources/","classpath:/static/","classpath:/public"); |

# 日志管理

Spring Boot在所有内部日志中使用[**Commons Logging**](http://commons.apache.org/proper/commons-logging/)，但是默认配置也提供了对常用日志的支持，如：[**Java Util Logging**](http://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html)，[**Log4J**](http://logging.apache.org/log4j/),[**Log4J2**](http://logging.apache.org/log4j/)和[**Logback**](http://logback.qos.ch/)。每种Logger都可以通过配置使用控制台或者文件输出日志内容。

## 控制台输出

在Spring Boot中默认配置了ERROR、WARN和INFO级别的日志输出到控制台。

我们可以通过两种方式切换至DEBUG级别：

在运行命令后加入--debug标志，如： java -jar myapp.jar --debug

在application.properties中配置debug=true，该属性置为true的时候，核心Logger（包含嵌入式容器、hibernate、spring）会输出更多内容，但是你自己应用的日志并不会输出为DEBUG级别。

## 文件输出

Spring Boot默认配置只会输出到控制台，并不会记录到文件中，若要增加文件输出，需要在application.properties中配置logging.file或logging.path属性。

**logging.file**，设置文件，可以是绝对路径，也可以是相对路径。如：logging.file=D:\\log\\my.log

**logging.path**，设置目录，会在该目录下创建spring.log文件，并写入日志内容，如：logging.path=D:\\log ，就会在log目录下创建spring.log文件。

**以上二选其一即可。**

**日志文件会在10Mb大小的时候被截断，产生新的日志文件，默认级别为：INFO。**

## 级别控制

在Spring Boot中只需要在application.properties中进行配置完成日志记录的级别控制。

配置格式：logging.level.\*=LEVEL

logging.level：日志级别控制前缀，\*为包名或Logger名

LEVEL：选项TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF

举例：

logging.level.com.tk=DEBUG：com.tk包下所有class以DEBUG级别输出

logging.level.root=WARN：root日志以WARN级别输出

## 自定义配置

根据不同的日志系统，你可以按如下规则组织配置文件名，就能被正确加载：

* Logback： logback.xml
* Log4j： log4j.properties, log4j.xml
* Log4j2： log4j2.xml
* JDK (Java Util Logging)： logging.properties

自己配置了以后所有的日志格式和日志内容都会依据配置文件。