

# 运行中的 Spring Boot



扫码试看/订阅  
《玩转 Spring 全家桶》

# 认识 Spring Boot 的各类 Actuator Endpoint

# Actuator

**目的**      管理比如可以通过Actuator shut down 应用程序，当然这个受保护的，默认不开启

- 监控并管理应用程序

## 访问方式

- HTTP
- JMX      用Jdk自带的JConsole或者JvirtualVM

## 依赖

- spring-boot-starter-actuator

# 一些常用 Endpoint

条件判断时候—Debug可以打印出条件计算结果

ID	说明	默认开启	默认 HTTP	默认 JMX
beans	显示容器中的 Bean 列表	Y	N	Y
caches	显示应用中的缓存	Y	N	Y
conditions	显示配置条件的计算情况	Y	N	Y
configprops	显示 @ConfigurationProperties 的信息	Y	N	Y
env	显示 ConfigurableEnvironment 中的属性	Y	N	Y
health	显示健康检查信息	Y	Y	Y
httptrace	显示 HTTP Trace 信息	Y	N	Y
info	显示设置好的应用信息	Y	Y	Y

# 一些常用 Endpoint

ID	说明	默认开启	默认 HTTP	默认 JMX
loggers	显示并更新日志配置	Y	N	Y
metrics	显示应用的度量信息	Y	N	Y
mappings	显示所有的 @RequestMapping 信息	Y	N	Y
scheduledtasks	显示应用的调度任务信息	Y	N	Y
shutdown	优雅地关闭应用程序	N	N	Y
threaddump	执行 Thread Dump	Y	N	Y
heapdump	返回 Heap Dump 文件，格式为 HPROF	Y	N	N/A
prometheus	返回可供 Prometheus 抓取的信息	Y	N	N/A

需要在pom里面加入相关依赖

# 如何访问 Actuator Endpoint

## HTTP 访问

- `/actuator/<id>`

## 端口与路径

- `management.server.address=`
- `management.server.port=` 发布端口和应用程序分离！
- `management.endpoints.web.base-path=/actuator`
- `management.endpoints.web.path-mapping.<id>=路径`

# 如何访问 Actuator Endpoint

## 开启 Endpoint

比如health.enabled=false关掉health

- `management.endpoint.<id>.enabled=true`
- `management.endpoints.enabled-by-default=false`

## 暴露 Endpoint

- `management.endpoints.jmx.exposure.exclude=`
- `management.endpoints.jmx.exposure.include=*`
- `management.endpoints.web.exposure.exclude=`
- `management.endpoints.web.exposure.include=info, health`



# 动手定制自己的 Health Indicator

增加自己指标，比如私有的设施

# Spring Boot 自带的 Health Indicator

## 目的

- 检查应用程序的运行状态

## 状态

- DOWN - 503
- OUT\_OF\_SERVICE - 503
- UP - 200
- UNKNOWN - 200

# Spring Boot 自带的 Health Indicator

## 机制

- 通过 HealthIndicatorRegistry 收集信息
- HealthIndicator 实现具体检查逻辑

## 配置项

- `management.health.defaults.enabled=true|false`
- `management.health.<id>.enabled=true`
- `management.endpoint.health.show-details=never|when-authorized|always`

# Spring Boot 自带的 Health Indicator

内置 HealthIndicator 清单			
CassandraHealthIndicator	ElasticsearchHealthIndicator	MongoHealthIndicator	SolrHealthIndicator
CouchbaseHealthIndicator	InfluxDbHealthIndicator	Neo4jHealthIndicator	
DiskSpaceHealthIndicator	JmsHealthIndicator	RabbitHealthIndicator	
DataSourceHealthIndicator	MailHealthIndicator	RedisHealthIndicator	

The screenshot shows an IDE interface with the following components:

- Project Structure (Left Sidebar):** A tree view showing a project with a 'health' package. The 'health' package contains several classes, including 'AbstractHealthIndicator', which is currently selected.
- Editor (Main Area):** The editor displays the decompiled source code for 'DataSourceHealthIndicator.class'. The code is as follows:

```
Decompiled .class file, bytecode version: 52.0 (Java 8)

1  + /.../
5
6  package org.springframework.boot.actuate.jdbc;
7
8  + import ...
25
26 public class DataSourceHealthIndicator extends AbstractHealthIndicator implements InitializingBean {
27     private static final String DEFAULT_QUERY = "SELECT 1";
28     private DataSource dataSource;
29     private String query;
30     private JdbcTemplate jdbcTemplate;
31
```
- Buttons (Top Right):** There are two buttons: 'Download Sources' and 'Choose Sources...'. A 'Reader Mode' button is also visible on the right side of the editor.

```
protected void doHealthCheck(Builder builder) throws Exception {
    if (this.dataSource == null) {
        builder.up().withDetail("database", "unknown");
    } else {
        this.doDataSourceHealthCheck(builder);
    }
}
```

# 自定义 Health Indicator

## 方法

- 实现 HealthIndicator 接口      或者extend一个AbstractHealthIndicator
- 根据自定义检查逻辑返回对应 Health 状态
- Health 中包含状态和详细描述信息



@Component

public class CoffeeIndicator implements HealthIndicator {

@Autowired

private CoffeeService coffeeService;

@Override

public Health health() {

    long count = coffeeService.getCoffeeCount();

    Health health;

    if (count > 0) {

        health = Health.up()

            .withDetail("count", count)

            .withDetail("message", "We have enough coffee.")

            .build();

    } else {

        health = Health.down()

            .withDetail("count", 0)

            .withDetail("message", "We are out of coffee.")

            .build();

    }

    return health;

}

}

**“Talk is cheap, show me the code.”**

*Chapter 10 / indicator-demo*



通过 Micrometer 获取运行数据

“Micrometer provides a simple facade over the instrumentation clients for the most popular **monitoring systems**, allowing you to instrument your JVM-based application code without vendor lock-in. Think SLF4J, but for metrics.”  
度量的门面

– *Micrometer* 官网

# 认识 Micrometer

## 特性

- 多维度度量
- 支持 Tag
- 预置大量探针
  - 缓存、类加载器、GC、CPU 利用率、线程池.....
- 与 Spring 深度整合

# 认识 Micrometer

## 支持多种监控系统

- Dimensional 多维度的
  - AppOptics, Atlas, Azure Monitor, Cloudwatch, Datadog, Datadog StatsD, Dynatrace, Elastic, Humio, Influx, KairosDB, New Relic, Prometheus, SignalFx, Sysdig StatsD, Telegraf StatsD, Wavefront
- Hierarchical 分层的
  - Graphite, Ganglia, JMX, Etsy StatsD

# 一些核心度量指标

## 核心接口

- Meter

## 内置实现

- Gauge, TimeGauge
- Timer, LongTaskTimer, FunctionTimer
- Counter, FunctionCounter
- DistributionSummary 95线, 99线

# Micrometer in Spring Boot 2.x

## 一些 URL

- `/actuator/metrics`
- `/actuator/prometheus` 把结果输出到prometheus

## 一些配置项

- `management.metrics.export.*`
- `management.metrics.tags.*`
- `management.metrics.enable.*`
- `management.metrics.distribution.*`
- `management.metrics.web.server.auto-time-requests`

# Micrometer in Spring Boot 2.x

## 核心度量项

- JVM、CPU、文件句柄数、日志、启动时间

## 其他度量项

- Spring MVC、Spring WebFlux
- Tomcat、Jersey JAX-RS
- RestTemplate、WebClient
- 缓存、数据源、Hibernate
- Kafka、RabbitMQ

# 自定义度量指标




三种方式：

- 通过 MeterRegistry 注册 Meter
- 提供 MeterBinder Bean 让 Spring Boot 自动绑定      实现MeterBinder接口
- 通过 MeterFilter 进行定制      度量指标进行过滤



**“Talk is cheap, show me the code.”**

*Chapter 10 / metrics-demo*

  
 @Service  
@Transactional  
 @Slf4j  
public class CoffeeOrderService implements MeterBinder {  
 @Autowired  
@Override  
public void bindTo(MeterRegistry meterRegistry) { this.orderCounter = meterRegistry.counter(name: "order.count"); }

通过 Spring Boot Admin 了解程序的运行状态

# Spring Boot Admin

## 目的

- 为 Spring Boot 应用程序提供一套管理界面

## 主要功能

- 集中展示应用程序 Actuator 相关的内容
- 变更通知

# 快速上手

## 服务端

- `de.codecentric:spring-boot-admin-starter-server:2.1.3`
- `@EnableAdminServer`

## 客户端

- `de.codecentric:spring-boot-admin-starter-client:2.1.3`
- 配置服务端及Endpoint
  - `spring.boot.admin.client.url=http://localhost:8080`
  - `management.endpoints.web.exposure.include=*`

# 安全控制

## 安全相关依赖

- `spring-boot-starter-security`

## 服务端配置

- `spring.security.user.name`
- `spring.security.user.password`

# 安全控制

## 客户端配置

- `spring.boot.admin.client.username`
- `spring.boot.admin.client.password`
- `spring.boot.admin.client.instance.metadata.user.name`
- `spring.boot.admin.client.instance.metadata.user.password`

**“Talk is cheap, show me the code.”**

*Chapter 10 / sba-server-demo & sba-client-demo*



# 如何定制 Web 容器的运行参数

# 内嵌 Web 容器

## 可选容器列表

- spring-boot-starter-tomcat
- spring-boot-starter-jetty
- spring-boot-starter-undertow
- spring-boot-starter-reactor-netty

# 修改容器配置

## 端口

- `server.port`
- `server.address`

## 压缩

- `server.compression.enabled`
- `server.compression.min-response-size`
- `server.compression.mime-types`

# 修改容器配置

## Tomcat 特定配置

- `server.tomcat.max-connections=10000`
- `server.tomcat.max-http-post-size=2MB`
- `server.tomcat.max-swallow-size=2MB`
- `server.tomcat.max-threads=200`
- `server.tomcat.min-spare-threads=10`

# 修改容器配置

## 错误处理

- `server.error.path=/error`
- `server.error.include-exception=false`
- `server.error.include-stacktrace=never`
- `server.error.whitelabel.enabled=true`

## 其他

- `server.use-forward-headers`
- `server.servlet.session.timeout`

# 修改容器配置

## 编程方式

- `WebServerFactoryCustomizer<T>`
  - `TomcatServletWebServerFactory`
  - `JettyServletWebServerFactory`
  - `UndertowServletWebServerFactory`

@Bean

```
public ServletWebServerFactory servletContainer() {
    TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory();
    tomcat.addAdditionalTomcatConnectors(createSslConnector());
    return tomcat;
}

private Connector createSslConnector() {
    Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
    Http11NioProtocol protocol = (Http11NioProtocol) connector.getProtocolHandler();
    try {
        File keystore = new ClassPathResource("keystore").getFile();
        File truststore = new ClassPathResource("keystore").getFile();
        connector.setScheme("https");
        connector.setSecure(true);
        connector.setPort(8443);
        protocol.setSSLEnabled(true);
        protocol.setKeystoreFile(keystore.getAbsolutePath());
        protocol.setKeystorePass("changeit");
        protocol.setTruststoreFile(truststore.getAbsolutePath());
        protocol.setTruststorePass("changeit");
        protocol.setKeyAlias("apitester");
        return connector;
    }
    catch (IOException ex) {
        throw new IllegalStateException("can't access keystore: [" + "keystore"
            + "] or truststore: [" + "keystore" + "]", ex);
    }
}
```

**“Talk is cheap, show me the code.”**

*Chapter 10 / tomcat-demo*



# 如何配置 HTTP/2 支持

## 配置 HTTPS 支持

### 通过参数进行配置

- `server.port=8443`
- `server.ssl.*`
  - `server.ssl.key-store`
  - `server.ssl.key-store-type`, JKS或者PKCS12
  - `server.ssl.key-store-password=secret`

# 生成证书文件

## 命令

- `keytool -genkey -alias 别名`  
    `-storetype 仓库类型 -keyalg 算法 -keysize 长度`  
    `-keystore 文件名 -validity 有效期`

## 说明

- 仓库类型，JKS、JCEKS、PKCS12 等
- 算法，RSA、DSA 等
- 长度，例如 2048

# 客户端 HTTPS 支持

## 配置 HttpClient ( $\geq 4.4$ )

- SSLContextBuilder 构造 SSLContext
- `setSSLHostnameVerifier(new NoopHostnameVerifier())`

## 配置 RequestFactory

- `HttpComponentsClientHttpRequestFactory`
  - `setHttpClient()`

**“Talk is cheap, show me the code.”**

*Chapter 10 / ssl-waiter-service & ssl-customer-service*

## 配置 HTTP/2 支持

### 前提条件

- Java  $\geq$  JDK 9
- Tomcat  $\geq$  9.0.0
- Spring Boot 不支持 h2c, 需要先配置 SSL

### 配置项

- `server.http2.enabled`

# 客户端 HTTP/2 支持

## HTTP 库选择

- OkHttpClient ( com.squareup.okhttp3:okhttp:3.14.0 )
- OkHttpClient

## RestTemplate 配置

- OkHttpClientHttpRequestFactory

**“Talk is cheap, show me the code.”**

*Chapter 10 / http2-waiter-service & http2-customer-service*



# 如何编写命令行运行的程序

# 关闭 Web 容器

## 控制依赖

- 不添加 Web 相关依赖

## 配置方式

- `spring.main.web-application-type=none`

# 关闭 Web 容器

## 编程方式

- `SpringApplication`
  - `setWebApplicationType()`
- `SpringApplicationBuilder`
  - `web()`
- 在调用 `SpringApplication` 的 `run()` 方法前设置 `WebApplicationType`

# 常用工具类

## 不同的 Runner

- ApplicationRunner
  - 参数是 ApplicationArguments
- CommandLineRunner
  - 参数是 String[]

## 返回码

- ExitCodeGenerator

**“Talk is cheap, show me the code.”**

*Chapter 10 / command-line-demo*

# 了解可执行 Jar 背后的秘密

# 认识可执行 Jar

## 其中包含

- Jar 描述, META-INF/MANIFEST.MF
- Spring Boot Loader, org/springframework/boot/loader
- 项目内容, BOOT-INF/classes
- 项目依赖, BOOT-INF/lib

## 其中不包含

- JDK / JRE

# 认识可执行 Jar

```
example.jar
|
+-META-INF
|   +-MANIFEST.MF
+-org
|   +-springframework
|       +-boot
|           +-loader
|               +-<spring boot loader classes>
+-BOOT-INF
    +-classes
    |   +-mycompany
    |       +-project
    |           +-YourClasses.class
    +-lib
        +-dependency1.jar
        +-dependency2.jar
```



# 如何找到程序的入口

## Jar 的启动类

- MANIFEST.MF
  - Main-Class: `org.springframework.boot.loader.JarLauncher`

## 项目的主类

- `@SpringApplication`
- MANIFEST.MF
  - Start-Class: `xxx.yyy.zzz`

## 再进一步：可直接运行的 Jar

### 如何创建可直接执行的 Jar

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <executable>true</executable>
  </configuration>
</plugin>
```

- 打包后的 Jar 可直接运行，无需 java 命令
- 可以在 .conf 的同名文件中配置参数

## 默认脚本中的一些配置项

配置项	说明	备注
CONF_FOLDER	放置 .conf 的目录位置	只能放环境变量中
JAVA_OPTS	JVM 启动时的参数	比如 JVM 的内存和 GC
RUN_ARGS	传给程序执行的参数	

**“Talk is cheap, show me the code.”**

*Chapter 10 / jar-demo*

# 如何将 Spring Boot 应用打包成 Docker 镜像

# 什么是 Docker 镜像

- 镜像是静态的只读模板
- 镜像中包含构建 Docker 容器的指令
- 镜像是分层的
- 通过 Dockerfile 来创建镜像

# Dockerfile

指令	作用	格式举例
FROM	基于哪个镜像	FROM <image>[:<tag>] [AS <name>]
LABEL	设置标签	LABEL maintainer="Geektime"
RUN	运行安装命令	RUN ["executable", "param1", "param2"]
CMD	容器启动时的命令	CMD ["executable","param1","param2"]
ENTRYPOINT	容器启动后的命令	ENTRYPOINT ["executable", "param1", "param2"]
VOLUME	挂载目录	VOLUME ["/data"]
EXPOSE	容器要监听的端口	EXPOSE <port> [<port>/<protocol>...]
ENV	设置环境变量	ENV <key> <value>
ADD	添加文件	ADD [--chown=<user>:<group>] <src>... <dest>
WORKDIR	设置运行的工作目录	WORKDIR /path/to/workdir
USER	设置运行的用户	USER <user>[:<group>]

# 通过 Maven 构建 Docker 镜像

## 准备工作

- 提供一个 Dockerfile
- 配置 dockerfile-maven-plugin 插件

## 执行构建

- `mvn package`
- `mvn dockerfile:build`

## 检查结果

- `docker images`



# dockerfile-maven-plugin

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>dockerfile-maven-plugin</artifactId>
  <version>${dockerfile-maven-version}</version>
  <executions>
    <execution>
      <id>default</id>
      <goals>
        <goal>build</goal>
        <goal>push</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <repository>spotify/foobar</repository>
    <tag>${project.version}</tag>
    <buildArgs>
      <JAR_FILE>${project.build.finalName}.jar</JAR_FILE>
    </buildArgs>
  </configuration>
</plugin>
```

**“Talk is cheap, show me the code.”**

*Chapter 10 / docker-demo*

# SpringBucks 实战项目进度小结

## 本章小结

- Spring Boot Actuator 的内容
- 如何监控运行中的 Spring Boot 应用程序
- 如何配置 Web 容器
- 如何开发命令行程序
- 可执行 Jar 包的原理
- 如何打包 Docker 镜像

# SpringBucks 进度小结

## **waiter-service**

- 增加了咖啡数量的健康检查
- 增加了订单数量的监控
- 增加了 HTTPS 和 HTTP/2 的支持

## **customer-service**

- 增加了 HTTPS 和 HTTP/2 的支持



扫码试看/订阅  
《玩转 Spring 全家桶》