

第 5-2 课：使用 Spring Boot Actuator 监控应用

微服务的特点决定了功能模块的部署是分布式的，大部分功能模块都是运行在不同的机器上，彼此通过服务调用进行交互，前后台的业务流会经过很多个微服务的处理和传递，出现异常如何快速定位便成为了一个问题，在这种框架下微服务的监控显得尤为重要。

Spring Boot 是一个自带监控的开源框架，组件 Spring Boot Actuator 负责监控应用的各项静态和动态的变量。项目结合 Spring Boot Actuator 的使用，便可轻松对 Spring Boot 应用监控治理，Spring Boot 的 Actuator 提供了很多生产级的特性，比如监控和度量 Spring Boot 应用程序，这些特性可以通过众多 REST 接口、远程 Shell 和 JMX 获得。

Actuator 监控

Spring Boot 使用“习惯优于配置的理念”，采用包扫描和自动化配置的机制来加载依赖 jar 中的 Spring Bean，不需要任何 XML 配置，就可以实现 Spring 的所有配置。虽然这样做能让代码变得非常简洁，但是整个应用的实例创建和依赖关系等信息都被离散到了各个配置类的注解上，这使得我们分析整个应用中资源和实例的各种关系变得非常的困难。

只需要在项目中添加 spring-boot-starter-actuator，就自动启用了监控功能。

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

添加 spring-boot-starter-web 主要是为了保持应用运行状态。

Spring Boot Actuator 是 Spring Boot 提供的对应用系统的检查和监控的集成功能，可以查看应用配置的详细信息，例如自动化配置信息、创建的 Spring beans 以及一些环境属性等。

Actuator 的 REST 接口

Actuator 监控分成两类：原生端点和用户自定义端点。自定义端点主要是指扩展性，用户可以根据自己的实际应用，定义一些比较关心的指标，在运行期进行监控。

原生端点是在应用程序里提供众多 Web 接口，通过它们了解应用程序运行时的内部状况，原生端点又可以分成三类：

- 应用配置类，可以查看应用在运行期的静态信息，例如自动配置信息、加载的 springbean 信息、yml 文件配置信息、环境信息、请求映射信息；
- 度量指标类，主要是运行期的动态信息，如堆栈、请求连、一些健康指标、metrics 信息等；
- 操作控制类，主要是指 shutdown，用户可以发送一个请求将应用的监控功能关闭。

Actuator 提供了 13 个接口，具体如下表所示。

HTTP 方法	路径	描述
GET	/auditevents	显示应用暴露的审计事件（如认证进入、订单失败）
GET	/beans	描述应用程序上下文里全部的 Bean 以及它们的关系
GET	/conditions	就是 1.0 的 /autoconfig，提供一份自动配置生效的条件情况，记录哪些自动配置条件通过了，哪些没通过
GET	/configprops	描述配置属性（包含默认值）如何注入 Bean
GET	/env	获取全部环境属性
GET	/env/{name}	根据名称获取特定的环境属性值
GET	/flyway	提供一份 Flyway 数据库迁移信息
GET	/liquibase	显示 Liquibase 数据库迁移的纤细信息
GET	/health	报告应用程序的健康指标，这些值由 HealthIndicator 的实现类提供
GET	/heapdump	dump 一份应用的 JVM 堆信息
GET	/httptrace	显示 HTTP 足迹，最近 100 个 HTTP request/response
GET	/info	获取应用程序的定制信息，这些信息由 info 打头的属性提供
GET	/logfile	返回 log file 中的内容（如果 logging.file 或者 logging.path 被设置）
GET	/loggers	显示和修改配置的 loggers
GET	/metrics	报告各种应用程序度量信息，比如内存用量和 HTTP 请求计数
GET	/metrics/{name}	报告指定名称的应用程序度量值
GET	/scheduledtasks	展示应用中的定时任务信息
GET	/sessions	如果我们使用了 Spring Session 展示应用中的 HTTP Sessions 信息
POST	/shutdown	关闭应用程序，要求 endpoints.shutdown.enabled 设置为 true
GET	/mappings	描述全部的 URI 路径，以及它们和控制器（包含 Actuator 端点）的映射关系
GET	/threaddump	获取线程活动的快照

命令详解

在 Spring Boot 2.x 中为了安全期间，Actuator 只开放了两个端点 /actuator/health 和 /actuator/info，可以在配置文件中设置打开。

可以打开所有的监控点：

```
management.endpoints.web.exposure.include=*
```

也可以选择打开部分：

```
management.endpoints.web.exposure.exclude=beans,trace
```

Actuator 默认所有的监控点路径都在 /actuator/*，当然如果有需要这个路径也支持定制。

```
management.endpoints.web.base-path=/manage
```

设置完重启后，再次访问地址就会变成 /manage/*。

Actuator 几乎监控了应用涉及的方方面面，我们重点讲述一些经常在项目中常用的命令。

health

health 主要用来检查应用的运行状态，这是我们使用最高频的一个监控点，通常使用此接口提醒我们应用实例的运行状态，以及应用不“健康”的原因，如数据库连接、磁盘空间不够等。

默认情况下 health 的状态是开放的，添加依赖后启动项目，访问：<http://localhost:8080/actuator/health> 即可看到应用的状态。

```
{
  "status" : "UP"
}
```

默认情况下，最终的 Spring Boot 应用的状态是由 HealthAggregator 汇总而成的，汇总的算法是：

- 设置状态码顺序为 setStatusOrder(Status.DOWN, Status.OUT_OF_SERVICE, Status.UP, Status.UNKNOWN);
- 过滤掉不能识别的状态码
- 如果无任何状态码，整个 Spring Boot 应用的状态是 UNKNOWN
- 将所有收集到的状态码按照 1 中的顺序排序
- 返回有序状态码序列中的第一个状态码，作为整个 Spring Boot 应用的状态

health 通过合并几个健康指数检查应用的健康情况。Spring Boot Actuator 有几个预定义的健康指标比如 DataSourceHealthIndicator、DiskSpaceHealthIndicator、MongoHealthIndicator、RedisHealthIndicator 等，它使用这些健康指标作为健康检查的一部分。

举个例子，如果你的应用使用 Redis，RedisHealthIndicator 将被当作检查的一部分；如果使用 MongoDB，

那么 MongoHealthIndicator 将被当作检查的一部分。

可以在配置文件中关闭特定的健康检查指标，比如关闭 Redis 的健康检查：

```
management.health.redis.enabled=false
```

默认所有的这些健康指标被当作健康检查的一部分。

详细的健康检查信息

默认只是展示了简单的 UP 和 DOWN 状态，为了查询更详细的监控指标信息，可以在配置文件中添加以下信息：

```
management.endpoint.health.show-details=always
```

重启后再次访问网址 <http://localhost:8080/actuator/health>，返回信息如下：

```
{
  "status": "UP",
  "diskSpace": {
    "status": "UP",
    "total": 209715195904,
    "free": 183253909504,
    "threshold": 10485760
  }
}
```

可以看到 HealthEndPoint 给我们提供默认的监控结果，包含磁盘空间描述总磁盘空间，剩余的磁盘空间和最小阈值。

其实看 Spring Boot-actuator 源码，会发现 HealthEndPoint 提供的信息不仅限于此，在 org.springframework.boot.actuate.health 包下会发现 ElasticsearchHealthIndicator、RedisHealthIndicator、RabbitHealthIndicator 等。

info

info 是我们自己在配置文件中以 info 开头的配置信息，比如在示例项目中的配置是：

```
info.app.name=spring-boot-actuator
info.app.version= 1.0.0
info.app.test= test
```

启动示例项目，访问 <http://localhost:8080/actuator/info> 返回部分信息如下：

```
{
  "app": {
    "name": "spring-boot-actuator",
    "version": "1.0.0",
    "test": "test"
  }
}
```

beans

根据示例就可以看出，展示了 bean 的别名、类型、是否单例、类的地址、依赖等信息。

启动示例项目，访问网址 <http://localhost:8080/actuator/beans> 返回部分信息如下：

```
[
  {
    "context": "application:8080:management",
    "parent": "application:8080",
    "beans": [
      {
        "bean": "embeddedServletContainerFactory",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedSe
rvletContainerFactory",
        "resource": "null",
        "dependencies": [

        ]
      },
      {
        "bean": "endpointWebMvcChildContextConfiguration",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.boot.actuate.autoconfigure.EndpointWebMvcChil
dContextConfiguration$$EnhancerBySpringCGLIB$$a4a10f9d",
        "resource": "null",
        "dependencies": [

        ]
      }
    ]
  }
]
```

conditions

Spring Boot 的自动配置功能非常便利，但有时候也意味着出问题比较难找出具体的原因。使用 conditions 可以在应用运行时查看代码了解某个配置在什么条件下生效，或者某个自动配置为什么没有生效。

启动示例项目，访问网址 <http://localhost:8080/actuator/conditions> 返回部分信息如下：

```
{
  "positiveMatches": {
    "DevToolsDataSourceAutoConfiguration": {
      "notMatched": [
        {
          "condition": "DevToolsDataSourceAutoConfiguration.DevToolsDataSourceCondition",
          "message": "DevTools DataSource Condition did not find a single DataSource bean"
        }
      ],
      "matched": [ ]
    },
    "RemoteDevToolsAutoConfiguration": {
      "notMatched": [
        {
          "condition": "OnPropertyCondition",
          "message": "@ConditionalOnProperty (spring.devtools.remote.secret) did not find property 'secret'"
        }
      ],
      "matched": [
        {
          "condition": "OnClassCondition",
          "message": "@ConditionalOnClass found required classes 'javax.servlet.Filter', 'org.springframework.http.server.ServerHttpRequest'; @ConditionalOnMissingClass did not find unwanted class"
        }
      ]
    }
  }
}
```

configprops

查看配置文件中设置的属性内容以及一些配置属性的默认值。

启动示例项目，访问网址 <http://localhost:8080/actuator/configprops> 返回部分信息如下：

```

{
  ...
  "environmentEndpoint": {
    "prefix": "endpoints.env",
    "properties": {
      "id": "env",
      "sensitive": true,
      "enabled": true
    }
  },
  "spring.http.multipart-org.springframework.boot.autoconfigure.web.MultipartProperties": {
    "prefix": "spring.http.multipart",
    "properties": {
      "maxRequestSize": "10MB",
      "fileSizeThreshold": "0",
      "location": null,
      "maxFileSize": "1MB",
      "enabled": true,
      "resolveLazily": false
    }
  },
  "infoEndpoint": {
    "prefix": "endpoints.info",
    "properties": {
      "id": "info",
      "sensitive": false,
      "enabled": true
    }
  }
  ...
}

```

env

展示了系统环境变量的配置信息，包括使用的环境变量、JVM 属性、命令行参数、项目使用的 jar 包等信息。和 configprops 不同的是，configprops 关注于配置信息，env 关注运行环境信息。

启动示例项目，访问网址 <http://localhost:8080/actuator/env> 返回部分信息如下：


```
{
  "profiles": [

  ],
  "server.ports": {
    "local.management.port": 8088,
    "local.server.port": 8080
  },
  "servletContextInitParams": {

  },
  "systemProperties": {
    "com.sun.management.jmxremote.authenticate": "false",
    "java.runtime.name": "Java(TM) SE Runtime Environment",
    "spring.output.ansi.enabled": "always",
    "sun.boot.library.path": "C:\\Program Files\\Java\\jdk1.8.0_101\\jre\\bin",
    "java.vm.version": "25.101-b13",
    "java.vm.vendor": "Oracle Corporation",
    "java.vendor.url": "http://java.oracle.com/",
    "java.rmi.server.randomIDs": "true",
    "path.separator": ";",
    "java.vm.name": "Java HotSpot(TM) 64-Bit Server VM",
    "file.encoding.pkg": "sun.io",
    "user.country": "CN",
    "user.script": "",
    "sun.java.launcher": "SUN_STANDARD",
    "sun.os.patch.level": "",
    "PID": "5268",
    "com.sun.management.jmxremote.port": "60093",
    "java.vm.specification.name": "Java Virtual Machine Spe
```

为了避免敏感信息暴露到 /env 里，所有名为 password、secret、key（或者名字中最后一段是这些）的属性在 /env 里都会加上“*”。举个例子，如果有一个属性名字是 database.password，那么它在 /env 中的显示效果是这样的：

```
"database.password": "*****"
```

/env/{name} 用法

就是 env 的扩展可以获取指定配置信息，比如 `http://localhost:8080/actuator/env/java.vm.version`，返回 `{"java.vm.version":"25.101-b13"}`。

heapdump

返回一个 GZip 压缩的 JVM 堆 dump。

启动示例项目，访问网址 <http://localhost:8080/actuator/heapdump> 会自动生成一个 JVM 的堆文件 heapdump，我们可以使用 JDK 自带的 JVM 监控工具 VisualVM 打开此文件查看内存快照。类似如下图：

与另一个堆转储进行比较			
类名	实例...	实例数	大小
char[]		... (15.6%)	... (33.4%)
java.lang.String		... (15.2%)	... (5.5%)
java.util.concurrent.ConcurrentHashMap\$Node		... (5.4%)	... (3.1%)
java.lang.Class[]		... (4.9%)	... (2%)
java.util.LinkedHashMap\$Entry		... (4.2%)	... (3.2%)
java.lang.reflect.Method		... (3.8%)	... (7.2%)
java.lang.Object		... (3.5%)	... (0.7%)
java.util.HashMap\$Node		... (3.4%)	... (1.9%)
java.lang.Object[]		... (2.9%)	... (3.6%)
java.util.LinkedHashMap		... (2%)	... (2.1%)
java.util.HashMap\$Node[]		... (1.9%)	... (3.6%)
java.util.ArrayList		... (1.2%)	... (0.5%)
java.lang.reflect.Field		... (1.1%)	... (1.7%)
byte[]		... (1%)	... (8%)
int[]		... (1%)	... (0.9%)
java.util.LinkedList\$Node		... (0.8%)	... (0.4%)
java.lang.ref.SoftReference		... (0.8%)	... (0.6%)
java.util.LinkedHashSet		... (0.8%)	... (0.2%)
java.lang.reflect.Constructor		... (0.7%)	... (1.1%)
java.lang.String[]		... (0.7%)	... (0.7%)
org.springframework.core.annotation.AnnotationUtils\$DefaultValueHolder		... (0.6%)	... (0.2%)
java.lang.Class\$ReflectionData		... (0.6%)	... (0.7%)
java.util.LinkedHashMap\$LinkedEntrySet		... (0.6%)	... (0.2%)
java.lang.reflect.Method[]		... (0.5%)	... (0.7%)

httptrace

该端点用来返回基本的 HTTP 跟踪信息。默认情况下，跟踪信息的存储采用 `org.springframework.boot.actuate.trace.InMemoryTraceRepository` 实现的内存方式，始终保留最近的 100 条请求记录。

启动示例项目，访问网址 <http://localhost:8080/actuator/httptrace>，返回信息如下：

```
{
  "traces": [
    {
      "timestamp": "2018-11-21T12:42:25.333Z",
      "principal": null,
      "session": null,
      "request": {
        "method": "GET",
        "uri": "http://localhost:8080/actuator/heapdump",
        "headers": {
          "cookie": [
            "Hm_lvt_0fb30c642c5f6453f17d881f529a1141=1513076406,1514961720,1515649377; Hm_lvt_6d8e8bb59814010152d98507a18ad229=1515247964,1515296008,1515672972,1516086283; UM_distinctid=1647364371ef6-003ab9d0469ea5-b7a103e-100200-1647364371f104; CNZZDATA1260945749=232252692-1513233181-%7C1537492730"
          ],
          "accept-language": [
            "zh-CN,zh;q=0.9"
          ]
        }
      }
    }
  ]
}
```

```

        "upgrade-insecure-requests": [
            "1"
        ],
        "host": [
            "localhost:8080"
        ],
        "connection": [
            "keep-alive"
        ],
        "accept-encoding": [
            "gzip, deflate, br"
        ],
        "accept": [
            "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8"
        ],
        "user-agent": [
            "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36"
        ]
    },
    "remoteAddress": null
},
"response": {
    "status": 200,
    "headers": {
        "Accept-Ranges": [
            "bytes"
        ],
        "Content-Length": [
            "39454385"
        ],
        "Date": [
            "Wed, 21 Nov 2018 12:42:25 GMT"
        ],
        "Content-Type": [
            "application/octet-stream"
        ]
    }
},
"timeTaken": 1380
},
{
    ...
},
...
]
}

```

记录了请求的整个过程的详细信息。

metrics

最重要的监控内容之一，主要监控了 JVM 内容使用、GC 情况、类加载信息等。

启动示例项目，访问网址 <http://localhost:8080/actuator/metrics> 返回部分信息如下：

```
{
  "mem": 337132,
  "mem.free": 183380,
  "processors": 4,
  "instance.uptime": 254552,
  "uptime": 259702,
  "systemload.average": -1.0,
  "heap.committed": 292864,
  "heap.init": 129024,
  "heap.used": 109483,
  "heap": 1827840,
  "nonheap.committed": 45248,
  "nonheap.init": 2496,
  "nonheap.used": 44269,
  "nonheap": 0,
  "threads.peak": 63,
  "threads.daemon": 43,
  "threads.totalStarted": 83,
  "threads": 46,
  "classes": 6357,
  "classes.loaded": 6357,
  "classes.unloaded": 0,
  "gc.ps_scavenge.count": 8,
  "gc.ps_scavenge.time": 99,
  "gc.ps_marksweep.count": 1,
  "gc.ps_marksweep.time": 43,
  "httpsessions.max": -1,
  "httpsessions.active": 0
}
```

对 /metrics 接口提供的信息进行简单分类如下表：

分类	前缀	报告内容
垃圾收集器	gc.*	已经发生过的垃圾收集次数，以及垃圾收集所耗费的时间，适用于标记—清理垃圾收集器和并行垃圾收集器（数据源自 <code>java.lang.management.GarbageCollectorMXBean</code> ）
内存	mem.*	分配给应用程序的内存数量和空闲的内存数量（数据源自 <code>java.lang.Runtime</code> ）
堆	heap.*	当前内存用量（数据源自 <code>java.lang.management.MemoryUsage</code> ）
类加载器	classes.*	JVM 类加载器加载与卸载的类的数量（数据源自 <code>java.lang.management.ClassLoadingMXBean</code> ）
系统	processors、instance.uptime、uptime、systemload.average	系统信息，如处理器数量（数据源自 <code>java.lang.Runtime</code> 、运行时间（数据源自 <code>java.lang.management.RuntimeMXBean</code> ）、平均负载（数据源自 <code>java.lang.management.OperatingSystemMXBean</code> ）
线程池	thread.*	线程、守护线程的数量，以及 JVM 启动后的线程数量峰值（数据源自 <code>java.lang.management.ThreadMXBean</code> ）
数据源	datasource.*	数据源连接的数量（源自数据源的元数据，仅当 Spring 应用程序上下文里存在 <code>DataSource Bean</code> 的时候才会有这个信息）
Tomcat 会话	httpsessions.*	Tomcat 的活跃会话数和最大会话数（数据源自嵌入式 Tomcat 的 <code>Bean</code> ，仅在使用嵌入式 Tomcat 服务器运行应用程序时才有这个信息）
HTTP	counter.status._、gauge.response._	多种应用程序服务 HTTP 请求的度量值与计数器

解释说明

- 请注意，这里的一些度量值，比如数据源和 Tomcat 会话，仅在应用程序中运行特定组件时才有数据，还可以注册自己的度量信息。
- HTTP 的计数器和度量值需要做一点说明，`counter.status` 后的值是 HTTP 状态码，随后是所请求的路径。举个例子，`counter.status.200.metrics` 表明 `/metrics` 端点返回 200(OK) 状态码的次数。
- HTTP 的度量信息在结构上也差不多，却在报告另一类信息。它们全部以 `gauge.response` 开头，表明这是 HTTP 响应的度量信息，前缀后是对应的路径，度量值是以毫秒为单位的时间，反映了最近处理该路径请求的耗时。
- 这里还有几个特殊的值需要注意，`root` 路径指向的是根路径或 `/`，`star-star` 代表了那些 Spring 认为是静态资源的路径，包括图片、JavaScript 和样式表，其中还包含了那些找不到的资源，这就是为什么经常会看到 `counter.status.404.star-star`，因为返回了 HTTP 404 (NOT FOUND) 状态的请求数。
- `/metrics` 接口会返回所有的可用度量值，但你也可能只对某个值感兴趣。要获取单个值，请求时可以在

URL 后加上对应的键名。例如，要查看空闲内存大小，可以向 /metrics/mem.free 发一个GET请求，比如访问网址 <http://localhost:8080/actuator/metrics/mem.free>，返回：{"mem.free":178123}。

shutdown

开启接口优雅关闭 Spring Boot 应用，要使用这个功能首先需要在配置文件中开启：

```
management.endpoint.shutdown.enabled=true
```

配置完成之后，启动示例项目，使用 curl 模拟 post 请求访问 shutdown 接口。

shutdown 接口默认只支持 post 请求。

```
curl -X POST "http://localhost:8080/actuator/shutdown"
{
  "message": "Shutting down, bye..."
}
```

此时会发现应用已经被关闭。

mappings

描述全部的 URI 路径，以及它们和控制器的映射关系。

启动示例项目，访问网址 <http://localhost:8080/actuator/mappings> 返回部分信息如下：

```
{
  "**/favicon.ico": {
    "bean": "faviconHandlerMapping"
  },
  "{[/hello]}": {
    "bean": "requestMappingHandlerMapping",
    "method": "public java.lang.String com.neo.controller.HelloController.index()"
  },
  "{[/error]}": {
    "bean": "requestMappingHandlerMapping",
    "method": "public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)"
  }
}
```

threaddump

/threaddump 接口会生成当前线程活动的快照，这个功能非常好，方便我们在日常定位问题的时候查看线程

的情况，主要展示了线程名、线程 ID、线程的状态、是否等待锁资源等信息。

启动示例项目，访问网址 <http://localhost:8080/actuator/threaddump> 返回部分信息如下：

```
[
  {
    "threadName": "http-nio-8088-exec-6",
    "threadId": 49,
    "blockedTime": -1,
    "blockedCount": 0,
    "waitedTime": -1,
    "waitedCount": 2,
    "lockName": "java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionOb
ject@1630a501",
    "lockOwnerId": -1,
    "lockOwnerName": null,
    "inNative": false,
    "suspended": false,
    "threadState": "WAITING",
    "stackTrace": [
      {
        "methodName": "park",
        "fileName": "Unsafe.java",
        "lineNumber": -2,
        "className": "sun.misc.Unsafe",
        "nativeMethod": true
      },
      ...
      {
        "methodName": "run",
        "fileName": "TaskThread.java",
        "lineNumber": 61,
        "className": "org.apache.tomcat.util.threads.TaskThread$WrappingRunnable",
        "nativeMethod": false
      }
      ...
    ],
    "lockInfo": {
      "className": "java.util.concurrent.locks.AbstractQueuedSynchronizer$Conditio
nObject",
      "identityHashCode": 372286721
    }
  }
  ...
]
```

生产出现问题的时候，可以通过应用的线程快照来检测应用正在执行的任务。

总结

经过这节课实践我们发现 Spring Boot Actuator 作为 Spring Boot 自带的监控组件非常强大，它可以监控和管理 Spring Boot 应用，如健康检查、审计、统计和 HTTP 追踪等，所有的这些特性可以通过 JMX 或者 HTTP endpoints 来获得。使用 Spring Boot Actuator 之后，可以让我们通过很简单的接口或者命令来查询应用的一切运行状态，从而达到对应用的运行情况了如指掌。

[点击这里下载源码。](#)