

# 指标监控——Spring Boot Actuator&Admin

## 指标监控——Spring Boot Actuator&Admin

### 1、指标监控简介

### 2、SpringBoot Actuator

介绍：

实现：

Actuator Endpoint端点

### 3、Spring Boot Admin

3.1、快速使用：

3.2、通过注册中心集成客户端

3.3、安全防护

3.4、邮件通知

### 4、定制 Endpoint

1、定制 Health指标信息

2、定制info指标信息

3、定制Metrics指标信息

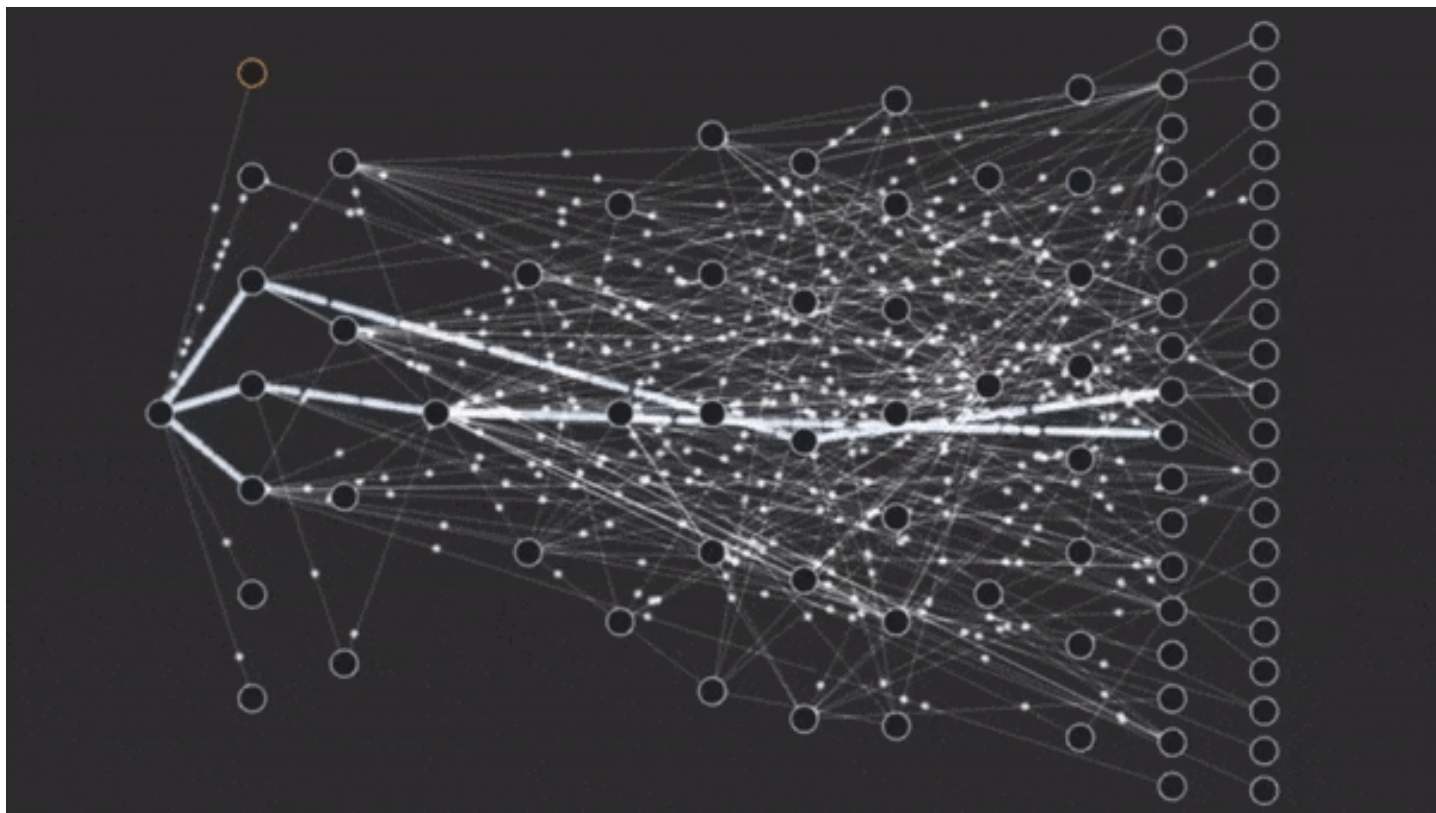
4、定制Endpoint

## 1、指标监控简介

干嘛的：

主要运用在微服务架构，所以我建议你先学微服务，否则可能get不到它的用处，只有大型的分布式系统才会用到指标监控...

Why：？



对于一个大型的几十个、几百个微服务构成的微服务架构系统，在线上时通常会遇到下面一些问题，比如：

1. 如何知道哪些服务出了问题，如何快速定位？（健康状况）
2. 如何统一监控各个微服务的性能指标（内存、jvm、并发数、线程池、Http 请求统计）
3. 如何统一管理各个微服务的日志？（切换线上日志等级，快速搜索日志...）
4. 如何优雅管理服务下线（正在运行的线程不发生中断）

So: 在这种大型分布式应用的环境下，我们如何能够快速发现问题、快速解决问题，必须要有监控平台、（链路追踪、日志）

## 2、SpringBoot Actuator

### 介绍：

SpringBoot自带监控功能Actuator，可以帮助实现对程序内部运行情况监控，比如监控状况、Bean加载情况、环境变量、日志信息、线程信息等

### 实现：

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-actuator</artifactId>
4 </dependency>
```

```
▼ org.springframework.boot:spring-boot-starter-actuator:2.4.0
  org.springframework.boot:spring-boot-starter:2.4.0 (omitted for duplicate)
  > org.springframework.boot:spring-boot-actuator-autoconfigure:2.4.0
  > io.micrometer:micrometer-core:1.6.1
```

- 引入场景
- 访问 [http://localhost:8080/actuator/\\*\\*](http://localhost:8080/actuator/**)
- 暴露所有监控信息为HTTP

```
1 management:
2   endpoints:
3     enabled-by-default: true # 默认开启所有监控端点
4   web:
5     base-path: /xushu    # 将"actuator"替换为"xushu" —安全措施
6     exposure:
7       include: '*' # 以web方式暴露所有端点
8
```

测试

<http://localhost:8080/actuator/beans>

<http://localhost:8080/actuator/configprops>

<http://localhost:8080/actuator/metrics>

<http://localhost:8080/actuator/metrics/jvm.gc.pause>

<http://localhost:8080/actuator/endpointName/detailPath>

• • • • •

**Actuator Endpoint**端点

JMX方式 可以通过 运行命令jconsole 查看

ID	说明	默认 HTTP	默认 JMX
beans	显示容器中的 Bean 列表	N	Y
caches	显示应用中的缓存	N	Y
conditions	显示配置条件的计算情况	N	Y
configprops	显示 @ConfigurationProperties 的信息	N	Y
env	显示 ConfigurableEnvironment 中的属性	N	Y
health	显示健康检查信息	Y	Y
httptrace	显示 HTTP Trace 信息	N	Y
info	显示设置好的应用信息	Y	Y
loggers	显示并更新日志配置	N	Y
metrics	显示应用的度量信息	N	Y
mappings	显示所有的 @RequestMapping 信息	N	Y
scheduledtasks	显示应用的调度任务信息	N	Y
shutdown	优雅地关闭应用程序	N	Y
threaddump	执行 Thread Dump	N	Y
heapdump	返回 Heap Dump 文件，格式为 HPROF	N	N/A
prometheus	返回可供 Prometheus 抓取的信息	N	N/A

## 常用端点说明

### Health: 监控状况

- 一个组件指标状态为Down则总状态信息Down，很多组件中都定制了Health子节点指标：比如jdbc、redis、等等

```

{
  status: "DOWN",
  components: {
    db: {
      status: "UP",
      details: {
        database: "MySQL",
        validationQuery: "isValid()"
      }
    },
    diskSpace: {
      status: "UP",
      details: {
        total: 107373129728,
        free: 30859857920,
        threshold: 10485760,
        exists: true
      }
    },
    ping: {
      status: "UP"
    },
    xuShu: {
      status: "DOWN",
      details: {
        error: "java.lang.ArithmeticException: / by zero"
      }
    }
  }
}

```

### shutdown:优雅关闭

- 注意需要web服务器的支持，做如下配置：
- 该端点必须通过POST请求或者JMX方式请求

```

1 server:
2   shutdown: graceful

```

### 测试代码：

请求/sayHi 时再请求/shutdown 端点， 会发现此时新请求无法进入， 正在请求的/sayHi 结束时系统才会关闭

```

1 @GetMapping("/sayHi")
2 public String sayHi(String username) throws InterruptedException {
3     // 线程长请求
4     Thread.sleep(5000);
5
6     System.out.println("success");
7
8     return "success";
9 }

```

### Metrics: 运行时指标

缺省情况下的指标信息：

- 如需查看某一指标可以在URL后面加上对应的指标名称
  - 如要查看http.server.requests指标：则访问actuator/metrics/http.server.requests
- 还可以传入tag过滤具体数据：/metrics/http.server.requests?tag=method:GET
  - 所有tag都可以作为参数（大小写敏感）

```

{
  - names: [
    "http.server.requests", http请求数量、时间
    "jvm.buffer.count",
    "jvm.buffer.memory.used", 缓冲
    "jvm.buffer.total.capacity",
    "jvm.classes.loaded",
    "jvm.classes.unloaded", 类加载
    "jvm.gc.live.data.size",
    "jvm.gc.max.data.size",
    "jvm.gc.memory.allocated", GC
    "jvm.gc.memory.promoted",
    "jvm.gc.pause",
    "jvm.memory.committed",
    "jvm.memory.max", 内存
    "jvm.memory.used",
    "jvm.threads.daemon",
    "jvm.threads.live",
    "jvm.threads.peak", 线程
    "jvm.threads.states",
    "logback.events",
    "process.cpu.usage",
    "process.start.time",
    "process.uptime", 处理器、
    "system.cpu.count",
    "system.cpu.usage",
    "tomcat.sessions.active.current",
    "tomcat.sessions.active.max",
    "tomcat.sessions.alive.max",
    "tomcat.sessions.created", Tomca
    "tomcat.sessions.expired",
    "tomcat.sessions.rejected"
  ]
}

```

#### Logfile: 远程日志跟踪

必须按如下配置设置日志添加到文件中，一旦设置会出现一个 /actuator/logfile 端点

```

- logfile: {
  href: "http://localhost:8081/xushu/logfile",
  templated: false
},

```

```

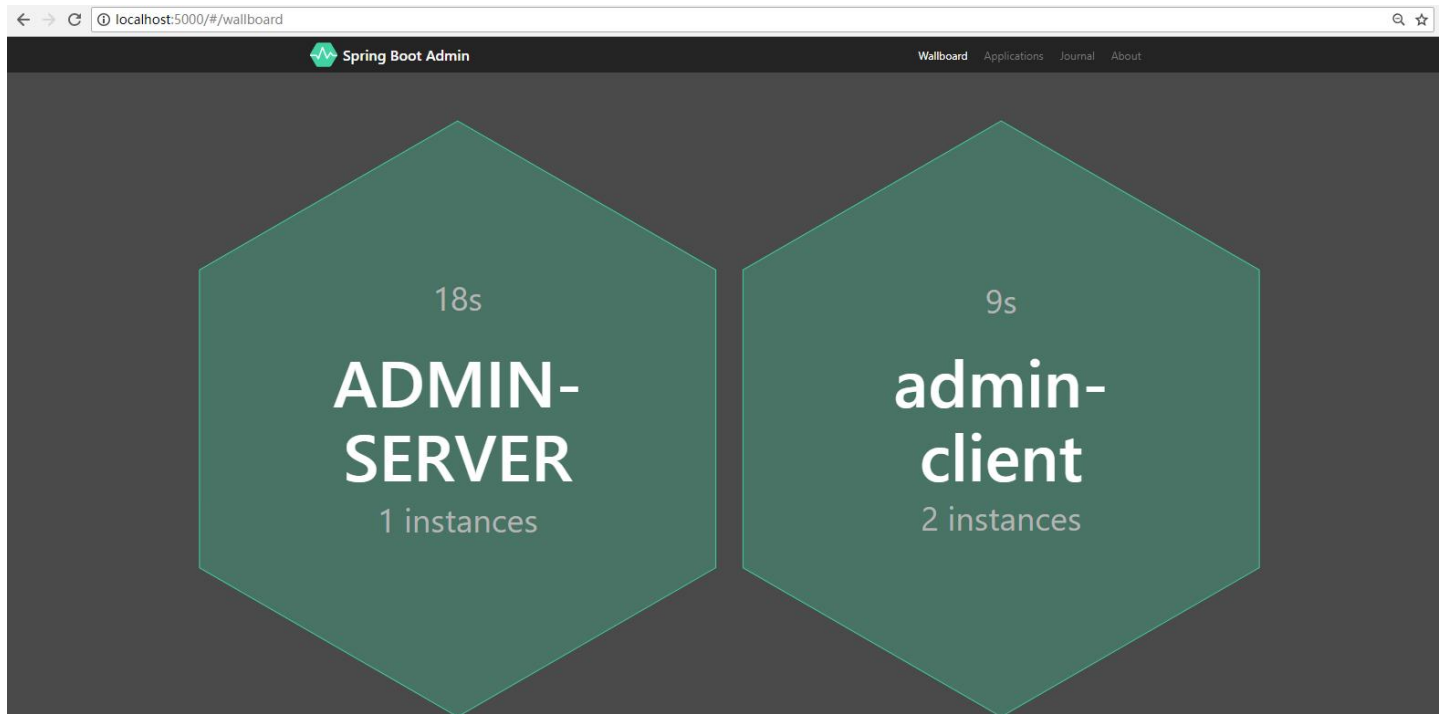
1 logging:
2   file:
3     name: D:/logs/xushu.log

```

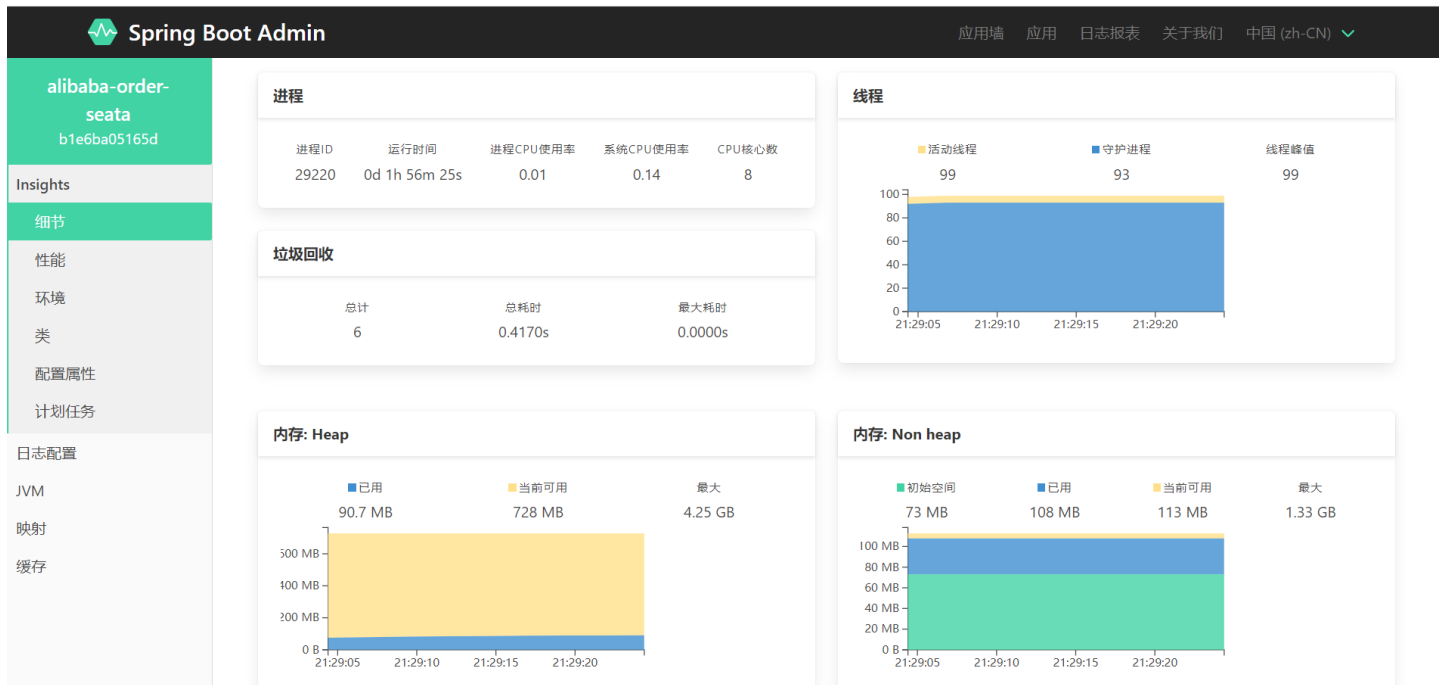
### 3、Spring Boot Admin

可视化监控平台，是一个基于 Spring Boot Actuator 端点之上的 Vue.js 应用程序。

<https://github.com/codecentric/spring-boot-admin>



- 绿色：健康
- 灰色：连接客户端健康信息超时（超过10s）
- 红色：就能看到具体异常信息



GitHub官方地址: <https://github.com/codecentric/spring-boot-admin>

### 3.1、快速使用:

#### 1、设置Spring Boot Admin Server

创建服务器并引入依赖，如一个springboot项目

版本建议: Spring Boot 2.x=Spring Boot Admin 2.x (比如Spring Boot 2.3.x 可以用Spring Boot Admin 2.3.x)

```
1 <dependency>
2   <groupId>de.codecentric</groupId>
3   <artifactId>spring-boot-admin-starter-server</artifactId>
```

```

4     <version>2.4.0-SNAPSHOT</version>
5 </dependency>
6 <dependency>
7     <groupId>org.springframework.boot</groupId>
8     <artifactId>spring-boot-starter-web</artifactId>
9 </dependency>

```

通过添加@EnableAdminServer到配置中来引入Spring Boot Admin Server配置：

```

1 @SpringBootApplication
2 @EnableAdminServer
3 public class Boot05AdminserverApplication {
4     public static void main(String[] args) {
5         SpringApplication.run(Boot05AdminserverApplication.class, args);
6     }
7
8 }

```

注册客户端应用程序并引入依赖，如一个springboot项目

```

1 <dependency>
2     <groupId>de.codecentric</groupId>
3     <artifactId>spring-boot-admin-starter-client</artifactId>
4     <version>2.3.1</version>
5 </dependency>

```

配置Spring Boot Admin Server的URL，客户端连接Spring Boot Admin Server的地址

```

1 spring.boot.admin.client.url=http://localhost:8080
2 management.endpoints.web.exposure.include=*

```

若连接不上，可能是地址使用计算机名称作为地址，可以改变使用ip注册

## alibaba-order-seata

Id: 1fd0c03a1fe3

🏠 <http://DESKTOP-FGTQI9C.mshome.net:8072/>

🔧 <http://DESKTOP-FGTQI9C.mshome.net:8072/actuator>

❤️ <http://DESKTOP-FGTQI9C.mshome.net:8072/actuator/health>

### 元数据

属性	值
startup	2021-09-16T19:27:43.373+08:00

### 健康

#### Instance

OFFLINE

exception	org.springframework.web.reactive.function.client.WebClientRequestException
message	failed to resolve 'DESKTOP-FGTQI9C.mshome.net' after 2 queries ; nested exception is java.net.UnknownHostException: failed to resolve 'DESKTOP-FGTQI9C.mshome.net' after 2 queries

```

1 spring:
2 boot:

```



```
3   admin:
4     client:
5       url: http://localhost:8080
6       instance:
7         prefer-ip: true # 使用ip注册进来
8   application:
9     name: boot-05-web-admin # 客户端名称
```

访问 服务器的根目录如http://localhost:8080/ 即可浏览

## 3.2、通过注册中心集成客户端

如果你有成百上千个微服务，这样配置未免太麻烦。如果您已经为您的应用程序使用了 Spring Cloud (Alibaba-nacos) Discovery，那么您就不需要 SBA 客户端。只需在 Spring Boot Admin Server 中添加一个 DiscoveryClient，剩下的工作由我们的 AutoConfiguration 完成。

下面的步骤使用 Nacos，但是也支持其他 Spring Discovery实现。

### • SBA服务端：

#### 1. 添加依赖

```
1 <!--nacos-服务注册发现-->
2 <dependency>
3   <groupId>com.alibaba.cloud</groupId>
4   <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
5 </dependency>
```

#### 2. 配置Nacos

```
1 spring:
2   cloud:
3     nacos:
4       discovery:
5         server-addr: 127.0.0.1:8848
6         username: nacos
7         password: nacos
8   application:
9     name: spring-boot-admin-server
```

```
1 # 也会将SBA服务配置为客户端， 所以也可以配置自己的endpoint规则(可选)
2 management:
3   endpoints:
4     web:
5       exposure:
6         include: '*'
7   endpoint:
8     health:
9       show-details: always
```

- **SBA客户端:**

1. 依赖

- a. 只需添加actuator即可

```
1
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-actuator</artifactId>
5     </dependency>
```

2. 配置

- a. 如需公开更多端点: (不配置该选项则只显示 (health和info端点))

```
1 management:
2     endpoints:
3         web:
4             exposure:
5                 include: '*'
```

### 3.3、安全防护

- **SBA服务端安全:**

spring-boot-admin-server-ui 提供了一个登录页面和一个注销按钮、可以结合SpringSecurity解决身份验证和授权。

Spring Security 配置如下:

依赖:

```
1
2 <!--安全问题-->
3 <dependency>
4     <groupId>org.springframework.boot</groupId>
5     <artifactId>spring-boot-starter-security</artifactId>
6 </dependency>
```

配置类:

```
1 @Configuration(proxyBeanMethods = false)
2 public class SecuritySecureConfig extends WebSecurityConfigurerAdapter {
3
4     private final AdminServerProperties adminServer;
5
6     private final SecurityProperties security;
7
8     public SecuritySecureConfig(AdminServerProperties adminServer, SecurityProperties security) {
9         this.adminServer = adminServer;
10        this.security = security;
```

```

11 }
12
13 @Override
14 protected void configure(HttpSecurity http) throws Exception {
15     SavedRequestAwareAuthenticationSuccessHandler successHandler = new SavedRequestAwareAuthenticationSuccessHandler();
16     successHandler.setTargetUrlParameter("redirectTo");
17     successHandler.setDefaultTargetUrl(this.adminServer.path("/"));
18
19     http.authorizeRequests(
20         (authorizeRequests) -> authorizeRequests.antMatchers(this.adminServer.path("/assets/**")).permitAll()
21             .antMatchers(this.adminServer.path("/actuator/info")).permitAll()
22             .antMatchers(this.adminServer.path("/actuator/health")).permitAll()
23             .antMatchers(this.adminServer.path("/login")).permitAll().anyRequest().authenticated()
24     ).formLogin(
25         (formLogin) -> formLogin.loginPage(this.adminServer.path("/login")).successHandler(successHandler).and(
26             .logout((logout) -> logout.logoutUrl(this.adminServer.path("/logout"))).httpBasic(Customizer.withDefaults())
27             .csrf((csrf) -> csrf.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
28                 .ignoringRequestMatchers(
29                     new AntPathRequestMatcher(this.adminServer.path("/instances"),
30                         HttpMethod.POST.toString()),
31                     new AntPathRequestMatcher(this.adminServer.path("/instances/*"),
32                         HttpMethod.DELETE.toString()),
33                     new AntPathRequestMatcher(this.adminServer.path("/actuator/**"))
34                 ))
35             .rememberMe((rememberMe) -> rememberMe.key(UUID.randomUUID().toString()).tokenValiditySeconds(1209600))
36     )
37
38     // Required to provide UserDetailsService for "remember functionality"
39     @Override
40     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
41         auth.inMemoryAuthentication().withUser(security.getUser().getName())
42             .password("{noop}" + security.getUser().getPassword()).roles("USER");
43     }
44
45 }

```

配置信息:

```

1 spring:
2   security:
3     user:
4       name: xushu
5       password: xushu

```

运行提示登录: 输入配置的name/password即可



## Spring Boot Admin

☐ 记住用户

### • SBA客户端:

如果服务端**不是通过注册中心**进行获取客户端，需要单独为客户端配置，因为不配置客户端无法向服务端注册应用（没有配置不会报错，只是在监控台看不到应用，也就是没有注册进去）

```
1 spring.boot.admin.client:
2   username: xushu
3   password: xushu
```

### 3.4、邮件通知

如果服务下线，会进行邮件通知

在spring boot admin 服务端修改

pom新增

```
1
2   <!-- 邮件通知 -->
3   <dependency>
4       <groupId>org.springframework.boot</groupId>
5       <artifactId>spring-boot-starter-mail</artifactId>
6   </dependency>
```

配置文件application.yml加入：

```
1 spring:
2   mail:
3     # 发件人使用的qq邮箱服务
4     host: smtp.qq.com
5     username: tulingxushu@foxmail.com
6     # 授权码，不是密码，在qq邮箱设置-账号里面有生成授权码
7     password: bktymeooyuapggbe
8   boot:
9     admin:
10      notify:
11        mail:
12          # 收件人，多个中间用,分隔
13          to: tulingxushu@foxmail.com
14          # 发件人
15          from: tulingxushu@foxmail.com
```

设置--->账户---



重启服务，然后将一个客户端关闭

product-service (32141eb704c4) is OFFLINE

Instance 32141eb704c4 changed status from UP to OFFLINE

Status Details

exception  
io.netty.channel.AbstractChannel\$AnnotatedConnectException  
message  
Connection refused: no further information: /192.168.65.106:8023

Registration

Service Url <http://192.168.65.106:8023>  
Health Url <http://192.168.65.106:8023/actuator/health>  
Management Url <http://192.168.65.106:8023/actuator>

4、定制 Endpoint

1、定制 Health指标信息

```

2 public class MyHealthIndicator implements HealthIndicator {
3
4     @Override
5     public Health health() {
6         int errorCode = check(); // perform some specific health check
7         if (errorCode != 0) {
8             return Health.down().withDetail("Error Code", errorCode).build();
9         }
10        return Health.up().build();
11    }
12
13 }
14

```

## 构建Health

```

1 Health build = Health.down()
2         .withDetail("msg", "error service")
3         .withDetail("code", "500")
4         .withException(new RuntimeException())
5         .build();
6 management:
7     health:
8         enabled: true
9         show-details: always #总是显示详细信息。可显示每个模块的状态信息
10

```

```

1 @Component
2 public class MyComHealthIndicator extends AbstractHealthIndicator {
3
4     //真实的检查方法
5     @Override
6     protected void doHealthCheck(Health.Builder builder) throws Exception {
7         //mongodb 获取连接进行测试
8         Map<String, Object> map = new HashMap<>();
9         if (1 == 1){
10 //            builder.up();//健康
11            builder.status(Status.UP);
12            map.put("count", 1);
13            map.put("ms", 100);
14        }else {
15 //            builder.down();//不健康
16            builder.status(Status.DOWN);
17            map.put("error", "连接超时");
18            map.put("ms", 3000);
19        }
20
21        builder.withDetail("code", 100).withDetails(map);
22

```

```
22     }
23 }
```

## 2、定制info指标信息

常用两种方式

### 1、编写配置文件

```
1 info:
2   appName: boot-admin
3   version: 2.0.1
4   mavenProjectName: @project.artifactId@ # 使用@@可以获取maven的pom文件值
5   mavenProjectVersion: @project.version@
```

### 2、编写InfoContributor

```
1 @Component
2 public class AppInfo implements InfoContributor {
3
4     @Override
5     public void contribute(Info.Builder builder) {
6         builder.withDetail("msg", "hello")
7                 .withDetail("hello", "world")
8                 .withDetails(Collections.singletonMap("world", "6661121"));
9     }
10 }
```

<http://localhost:8080/actuator/info> 会输出以上方式返回的所有info信息

## 3、定制Metrics指标信息

Spring Boot Actuator 为 [Micrometer](#) 提供了依赖管理和自动配置功能，Micrometer 是一个应用指标 facade（面门），支持多种监控系统，包括：

AppOptics, Azure Monitor, Netflix Atlas, CloudWatch, Datadog, Dynatrace, Elastic, Ganglia, Graphite, Humio, Influx/Telegraf, JMX, KairosDB, New Relic, Prometheus, SignalFx, Google Stackdriver, StatsD, 和 Wavefront.

Micrometer作用：

- 提供一些列api供我们操作指标
- 提供了缓存、类加载器、GC、jvm内存\cpu 利用率 线程...指标 能够开箱即用
- 已经融入springboot Actuator

<https://docs.spring.io/spring-boot/docs/2.3.12.RELEASE/reference/html/production-ready-features.html#production-ready-metrics-meter>

JVM度量，报告利用率(JVM metrics, report utilization of):

- 各种内存和缓冲池(Various memory and buffer pools)
- 与垃圾收集有关的统计数据(Statistics related to garbage collection)
- 线程利用率(Threads utilization)
- 加载/卸载的类数(Number of classes loaded/unloaded)
- CPU 指标-CPU metrics

- 文件描述符度量-File descriptor metrics
  - Kafka 的消费者、生产者和流量指标-Kafka consumer, producer, and streams metrics
  - Log4j2度量: 记录每个level记录在 Log4j2中的事件数量-Log4j2 metrics: record the number of events logged to Log4j2 at each level
  - Logback度量: 记录每个级别登录到 Logback 的事件数量 —Logback metrics: record the number of events logged to Logback at each level
  - 正常运行时间指标: 报告正常运行时间指标和表示应用程序绝对启动时间的固定指标—Uptime metrics: report a gauge for uptime and a fixed gauge representing the application's absolute start time
  - Tomcat 指标—Tomcat metrics (`server.tomcat.mbeanregistry.enabled` must be set to `true` for all Tomcat metrics to be registered)
  - Spring整合指标 [Spring Integration](#) metrics
- Spring Integration 度量啊

## 2、增加定制Metrics

### 定制计量方法:

#### • Counter

`Counter`是一种比较简单的Meter，它是一种单值的度量类型，或者说是一个单值计数器。

#### 使用场景:

`Counter`的作用是记录XXX的总量或者计数值，**适用于一些增长类型的统计**，例如下单、支付次数、Http请求总量记录等等，

```
1 //记录下单总数
2 Metrics.counter("order.count", "order.channel", order.getChannel()).increment();
```

#### • Timer

Timer(计时器)适用于记录耗时比较短的事件的执行时间，通过时间分布展示事件的序列和发生频率。

#### 使用场景:

根据个人经验和实践，总结如下：

- 1、记录指定方法的执行时间用于展示。
- 2、记录一些任务的执行时间，从而确定某些数据来源的速率，例如消息队列消息的消费速率等。

```
1 @Around(value = "execution(* com.tuling.service.*Service.*(..))")
2     public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
3         Signature signature = joinPoint.getSignature();
4         MethodSignature methodSignature = (MethodSignature) signature;
5         Method method = methodSignature.getMethod();
6         Timer timer = Metrics.timer("method.cost.time", "method.name", method.getName());
7         ThrowableHolder holder = new ThrowableHolder();
8         Object result = timer.recordCallable(() -> {
9             try {
10                 return joinPoint.proceed();
11             } catch (Throwable e) {
12                 holder.throwable = e;
13             }
14             return null;
15         });
16         if (null != holder.throwable) {
17             throw holder.throwable;
18         }
19     }
```



```
18         return result;
19     }
20 }
```

## • Gauge

Gauge(仪表)是获取当前度量记录值的句柄，也就是它表示一个可以任意上下浮动的单数值度量Meter。

### 使用场景：

根据个人经验和实践，总结如下：

- 1、有自然(物理)上界的浮动值的监测，例如物理内存、集合、映射、数值等。
- 2、有逻辑上界的浮动值的监测，例如积压的消息、(线程池中)积压的任务等，其实本质也是集合或者映射的监测。

```
1 Metrics.gauge("message.gauge", Collection::size);
```

## • Summary

Summary(摘要)主要用于跟踪事件的分布，在Micrometer中，对应的类是DistributionSummary(分发摘要)。它的使用方式和Timer十分相似，但是它的记录值并不依赖于时间单位。

### 使用场景：

根据个人经验和实践，总结如下：

- 1、不依赖于时间单位的记录值的测量，例如服务器有效负载值，缓存的命中率等。

```
1 DistributionSummary summary = Metrics.summary("test.summary");
2 summary.record(1);
```

其他定制方式：

### 方式1

```
1 class MyService{
2     Counter counter;
3
4     public CityServiceImpl(MeterRegistry meterRegistry){
5         // http://localhost:8080/actuator/metrics/cityService.saveCity.count
6         counter = meterRegistry.counter("cityService.saveCity.count");
7     }
8
9     public void saveCity(City city) {
10         counter.increment();
11         cityMapper.insert(city);
12     }
13 }
```

### 方式2：

```
1 @Bean
2 MeterBinder queueSize(Queue queue) {
3     return (registry) -> Gauge.builder("queueSize", queue::size).register(registry);
4 }
5
```

## 4、定制Endpoint

```
1 @Component
2 @Endpoint(id = "myservice")
3 public class MyServiceEndPoint {
4
5     @ReadOperation
6     public Map getDockerInfo(){
7         //端点的读操作 http://localhost:8080/actuator/myservice
8         return Collections.singletonMap("dockerInfo","docker stated...");
9     }
10
11     @WriteOperation
12     public void stopDocker(){
13         System.out.println("docker stopped...");
14     }
15 }
16
```

场景：开发ReadinessEndpoint来管理程序是否就绪，或者LivenessEndpoint来管理程序是否存活；