RefreshEndpoint 内部也调用了 ContextRefresher 的 refresh 方法。我们在 Config Client 客户端调用 RefreshEndpoint，也可以实现动态刷新（配置文件加上 management.endpoints.web.exposure.include=* 配置启用 RefreshEndpoint）。

RefreshEndpoint 对应的 ID 为 refresh，使用 curl 调用 curl -XPOST http://localhost:8081/actuator/refresh，返回["book.author"]，表示这个配置项对应的配置已经发生了改变。

试想一下：在分布式系统中，如果一个服务对应 100 个应用实例，这 100 个实例使用 Spring Cloud Config Client 加载配置，如果需要动态刷新配置，是否要在每一个实例中调用 RefreshEndpoint? 这显然是不合理的。Spring Cloud Bus（消息总线）能解决这个问题，其具体内容将在第 7 章介绍。

4.5　再谈配置动态刷新

我们在 4.3.3 节已介绍过 Spring Cloud 配置动态刷新的原理。

Spring Cloud 配置动态刷新本质上是事件的触发让客户端再一次从配置中心拉取最新的配置，然后配合@RefreshScope 重新刷新 Bean（重新刷新 Spring Bean 会重新读取一遍配置），以达到配置动态刷新的目的。这种方式会带来以下问题：

- Spring Cloud 提供的 Spring Cloud Config Server/Client 组件在分布式环境下需要配合 Spring Cloud Bus（消息总线），来达到多节点配置动态刷新的目的。因此，使用 Spring Cloud Config Server/Client 组件需要额外加上消息中间件组件。
- @RefreshScope 引起的 Spring Bean 刷新可能会与其他组件冲突。比如，spring-context 模块的 scheduling 调度功能在 Spring Bean 被刷新后会失去作用。

笔者认为配置管理更应该专注在配置上，而不应该有其他操作，比如需要引入消息中间件，或者需要重新刷新 Spring Bean 等操作。

Nacos 内部的配置动态刷新原理是通过客户端维护长轮询的任务，定时拉取发生变更的配置信息，然后将最新的数据推送给客户端 Listener 的持有者。下面来看一下这个长轮询的实现原理。

Nacos 配置所有的操作通过 NacosConfigService 完成，NacosConfigService 的构造函数内部会构造一个 ClientWorker，这个 ClientWorker 维护长轮询任务：

```
public NacosConfigService(Properties properties) throws NacosException {
```