**1. Features**

SC专注于为经典案例提供良好的开箱即用体验，以及提供扩展机制用于覆盖其它案例。

* 分布式/版本化配置
* 服务注册和发现
* 路由
* 服务与服务之间的调用
* 负载均衡
* 断路器
* 分布式消息传递

# Part I. Cloud Native Applications

Cloud Native 是一种开发风格，它鼓励在持续交付和价值驱动开发领域采用最佳实践。持续交付最佳实践等相关科目参见[12-factor Apps](http://12factor.net/)（<https://12factor.net/zh_cn/），12-factor-apps> 主要阐述了开发实践应该与交付运营目标相一致，比如通过声明式编程、管理、监控。SC以多种方式促进这种开发风格，起点是一组易用的分布式组件。

许多特性都已被Spring boot覆盖，SC基于Spring boot构建。SC提供了两个库（SC context / SC commons）包含了其它一些特性。SC context提供应用上下文服务（引导上下文，加密，刷新范围和环境端点）。SC commons是一组抽象类和常用类，用于对不同具体实现的抽象（SC Netflix vs SC Consul）。

如果使用SUN JDK过程中遇到“Illegal key size”，请安装“java加密扩展”。

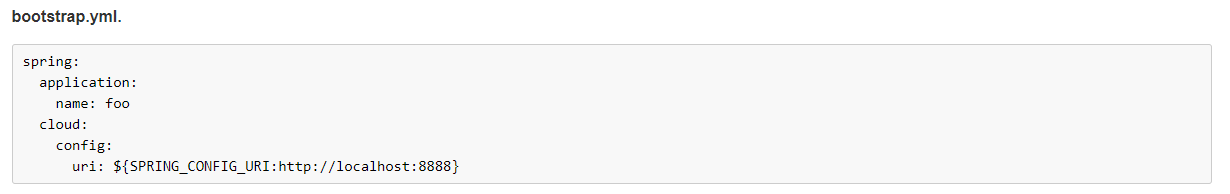
## 2. Spring Cloud Context: Application Context Services

Spring Boot对于如何使用Spring构建应用程序有一些约定看法：例如约定了常规文件位置，以及用于常见管理和监视任务的端点。Spring Cloud建立在此之上，并添加了一些系统中所有组件都要使用或偶尔需要的功能。

## 2.1 The Bootstrap Application Context

SC Application通过创建一个Bootstrap Context来运转，这个上下文是SC Application的父上下文。初始化的时候，Bootstrap Context负责从外部源中加载配置属性，并解密本地配置文件中的属性。这两个上下文共享一个Environment，这个Environment是任何Spring application的外部属性的来源。Bootstrap属性有高优先级，默认情况下，它们不会被本地配置覆盖。

Bootstrap Context与application context在定位外部配置约定不同，因此使用bootstrap.yml来作为bootstrap context的配置文件，而不是用application.yml，保持两个上下午在外部配置上的良好分离。



如果需要从SC config中获取应用特定配置的话，那么推荐设置spring.application.name (in bootstrap.yml or application.yml)。

您可以通过设置spring.cloud.bootstrap.enabled=false（例如在系统属性中）来完全禁用引导过程。

## 2.2 Application Context Hierarchies

如果你通过SpringApplication或者SpringApplicationBuilder创建一个Application Context,那么会创建Bootstrap Context作为 Application Context的父上下文。在Spring里有个特性，子上下文会继承父上下文的属性源和配置文件 ，所以main application context 相对于没有使用Spring Cloud Config的application context，会新增额外的属性源。额外的属性源有：

“bootstrap” : 如果在Bootstrap Context扫描到PropertySourceLocator（属性源定位）并且有属性，那么会添加到CompositePropertySource（混合属性源）。Spirng Cloud Config就是通过这种方式来添加的属性的，有关如何自定义此属性源的内容的[说明](https://springcloud.cc/spring-cloud-dalston.html#customizing-bootstrap-property-sources)，请参阅[下文](https://springcloud.cc/spring-cloud-dalston.html#customizing-bootstrap-property-sources)。

“applicationConfig: [classpath:bootstrap.yml]” (and friends if Spring profiles are active): 如果项目存在bootstrap.yml，里面的属性用来配置Bootstrap Context，这些属性也添加到了子上下文通过子上下文继承父上下文，它比application.yml优先级低，同时也比其它被添加到子上下文中用于创建Spring boot application的属性源优先级低。下文有介绍。

由于属性源的排序规则，“引导”条目优先，但请注意，这些条目不包含来自bootstrap.yml的任何数据，它具有非常低的优先级，但可用于设置默认值。

您可以通过简单地设置您创建的ApplicationContext的父上下文来扩展上下文层次结构，例如使用spirng自己的接口，或使用SpringApplicationBuilder方便方法（parent()，child()和sibling(兄弟姐妹））。Bootstrap Context将是您创建自己的最高级上下文的父级。层次结构中的每个上下文都将有自己的“引导”属性源（可能为空），以避免无意中将父上下文的属性带到子上下文。层次结构中的每个上下文（原则上）也可以具有不同的spring.application.name，因此如果存在配置服务器，那么这些上下文可以拥有不同的远程属性源。普通的Spring应用程序上下文行为规则适用于属性解析：子上下文的属性会覆盖父上下文的属性，如果属性名或属性源名相同（如果属性源名相同，父上下文属性源不会添加到子中）。

请注意，SpringApplicationBuilder允许您在整个层次结构中共享Environment，但这不是默认值。因此，兄弟上下文不需要具有相同profiles或属性源，尽管它们与父母共享共同点。

## 2.3 Changing the Location of Bootstrap Properties

Bootstrap.xml位置可以使用spring.cloud.bootstrap.name (default "bootstrap") 或者spring.cloud.bootstrap.location (default empty)来自定义，例如，在系统属性中。这些属性的行为与spring.config.\*类似，实际上他们用来通过在其environment中设置这些属性来构建bootstrap appliacationcontext。如果有一个激活的Profiles(spring.profiles.active 或者 通过environment api)，那么对应的bootstrap-env也会被加载，就想常规的spring boot 应用程序。

## 2.4 Overriding the Values of Remote Properties

通过bootstrap contex添加到应用程序的属性源通常是“远程”（例如从配置服务器），并且默认情况下，不能被本地属性覆盖，除了命令行属性。如果你要允许application使用自己的系统属性或配置文件覆盖远程属性，则远程属性源必须通过设置spring.cloud.config.allowOverride=true（在本地设置本身不起作用）授予权限。一旦设置了该标志，就会有一些更精细的设置来控制远程属性与系统属性、应用程序本地配置的优先级：spring.cloud.config.overrideNone=true覆盖任何本地属性源，spring.cloud.config.overrideSystemProperties=false那么只有系统属性和env var能覆盖远程设置，本地配置文件无法覆盖远程。

## 2.5 Customizing the Bootstrap Configuration

Bootstrap context可以被训练做任何你喜欢的事通过在/META-INF/spring.factories（spring cloud context包中）中org.springframework.cloud.bootstrap.BootstrapConfiguration key后面添加条目。Key对应的值为一个用逗号分隔用来创建context的Spring @Configuration类列表。你可以在此处创建要用于自动装配的主应用程序上下文的任何bean，自动装配与标注了@Beans的ApplicationContextInitializer子类有特殊约定。你可以通过@Order来控制启动顺序(the default order is "last") 。

警告：当你添加自定义BootstrapConfiguration时，请不要被错误扫描的到main application context中，main application context不需要它们。要给boot configuration类使用单独的包，这个包不能被@ComponentScan or @SpringBootApplication注解包含。

bootstrap进程通过将初始化器注入SpringApplication实例后结束(即正常的SpringBoot启动顺序,无论是作为独立程序运行还是部署在应用程序服务器中).首先用spring.factories配置的类创建一个bootstrap contet，然后创建@Beans注解的ApplicationContextInitializer子类，这些类都会在main SpringApplication启动前添加到其中。

## 2.6 Customizing the Bootstrap Property Sources

在bootstrap 进程中默认添加的额外的配置源是Config Server,但是你通过添加PropertySourceLocator的beans到bootstrap context中(通过spring.factories),添加额外的资源.你可以使用这个从不同的服务器插入额外的配置,或者从数据库   
example

*@Configuration*

**public** **class** CustomPropertySourceLocator **implements** PropertySourceLocator {

*@Override*

**public** PropertySource<?> locate(Environment environment) {

**return** **new** MapPropertySource("customProperty",

Collections.<String, Object>singletonMap("property.from.sample.custom.source", "worked as intended"));

}

}

传入的Environment是要创建的ApplicationContext的Environment，即为我们提供额外的属性来源的。它将已经具有正常的Spring Boot提供的资源来源，因此您可以使用它们来定位特定于此Environment的属性源（例如通过将其绑定在spring.application.name上，如在默认情况下所做的那样Config Server属性源定位器）。

如果你创建的jar里面有这个类，然后新增META-INF/spring.factories，包含以下配置：

org.springframework.cloud.bootstrap.BootstrapConfiguration=sample.custom.CustomPropertySourceLocator，那么“customProperty”PropertySource将显示在包含该jar的任何应用程序中。

## 2.7 Environment Changes

Application监听EnvironmentChangeEvent，并对变化做出若干种标准反应（用户可以以常规方式添加ApplicationListeners）。当观察到EnvironmentChangeEvent时，那就意味着将有一个已更改的键值对列表，Application将使用这个列表去做：

- 重新绑定context中任何一个@ConfigurationProperties bean   
- 设置任何属性的日志级别logging.level.\*

请注意，config client默认情况下不会通过轮询查找Environment中的更改，通常我们不建议使用轮询方法来检测更改（尽管可以使用@Scheduled注释进行设置）。如果您有一个横向扩展的application，那么最好将EnvironmentChangeEvent广播到所有实例，而不是让它们轮询更改（例如使用[Spring Cloud总线](https://github.com/spring-cloud/spring-cloud-bus)）。

EnvironmentChangeEvent覆盖一大类刷新用例，只要您真想更改Environment并发布事件（这些API是公开的，是Spring core的一部分）。您可以通过访问/configprops端点（普通Spring Boot执行器功能）来验证更改是否绑定到@ConfigurationProperties bean。例如，DataSource可以在运行时更改其maxPoolSize（由Spring Boot创建的默认DataSource是一个@ConfigurationProperties bean），并且动态增加容量。重新绑定@ConfigurationProperties不会覆盖另一大类用例，您需要更多的控制刷新，并且您需要更改的响应在整个ApplicationContext上是原子的。为了解决这些担忧，我们有@RefreshScope。

## 2.8 Refresh Scope

当配置更改时，标有@RefreshScope的Spring @Bean将会被特殊处理。这解决了有状态的bean只能在初始化时注入配置的问题。例如，如果datatsource在URL变更时有打开的连接，那么我们可能希望这些连接的持有者能够完成正在做的工作。此后当从连接池中获取连接时，拿到都是新URL的连接。

Refresh scope beans当被使用时（即调用一个方法）会延迟生效，并且作用域呈现的是初始值的缓存。要强制bean在下一个方法调用重新初始化，您只需要使其缓存条目无效。

RefreshScope是context中的一个bean，它有一个公共方法refreshAll()来清除目标缓存中的范围内的所有bean。还有一个refresh(String)方法可以按名称刷新单个bean。此功能在/refresh端点（通过HTTP或JMX）中公开。

注意：@RefreshScope（技术上）在@Configuration类上工作，但可能会导致令人惊讶的行为：例如，这并不意味着该类中定义的所有@Beans本身都是@RefreshScope。具体来说，任何依赖这些bean的东西都不能依赖于它们在刷新启动时被更新，除非它本身在@RefreshScope（在其中它将会被重新构建，而且它的依赖也会被重新注入，在此处它将会在刷新配置时被重新初始化）。

## 2.9 Encryption and Decryption

属性值加密。（暂不翻译）

## 2.10 Endpoints

对于Spring Boot actuator，还有一些额外的管理端点：

- POST 到/env接口去更新Environment并重新绑定@ConfigurationProperties和log级别   
- /refresh用于重新加载bootstrap context并刷新@RefreshScope标注的bean   
- /restart关闭ApplicationContext并重新启动它（默认情况下禁用）   
- /pause和/resume调用ApplicationContext的生命周期方法stop()和start()

## 3. Spring Cloud Commons: Common Abstractions

将所有Spring Cloud客户端都要引用的模式诸如服务发现、负载平衡和断路器等封装成一个通用的抽象层，与具体实现分离（例如通过Eureka或Consul发现）。

## 3.1 @EnableDiscoveryClient

[Commons提供@EnableDiscoveryClient注解。通过META-INF/spring.factories](mailto:Commons提供@EnableDiscoveryClient注解。通过META-INF/spring.factories)可以查找DiscoveryClient接口的实现。可以在spring.factories文件的org.springframework.cloud.client.discovery.EnableDiscoveryClient中添加新Discovery Client实现。DiscoveryClient实现的示例是[Spring Cloud Netflix Eureka](http://cloud.spring.io/spring-cloud-netflix/)，[Spring Cloud Consul发现](http://cloud.spring.io/spring-cloud-consul/)和[Spring Cloud Zookeeper发现](http://cloud.spring.io/spring-cloud-zookeeper/)。

默认情况下，DiscoveryClient实现将在远程discover server中自动注册Spring Boot server。可以通过在@EnableDiscoveryClient中设置autoRegister=false来禁用此功能。

警告：@EnableDiscoveryClient注解非必须，只要classpath包含discoveryclient实现就能自动注册

### 3.1.1 Health Indicator

Commons新增了一个spring boot HealthIndicator实现类DiscoveryHealthIndicator用于监测DiscoverClient，通过设置spring.cloud.discovery.client.composite-indicator.enabled=false禁用复合HealthIndicator。一个基于 DiscoveryClient 通用的 HealthIndicator（DiscoveryClientHealthIndicator） 是自动配置。为了禁用它，可以设置 spring.cloud.discovery.client.health-indicator.enabled=false.。为了禁用 DiscoveryClientHealthIndicator的描述域可以设置 spring.cloud.discovery.client.health-indicator.include-description=false ，otherwise it can bubble up as the description of the rolled up HealthIndicator（）

## 3.2 ServiceRegistry

Commons现在提供了一个ServiceRegistry接口，它提供了诸如register(Registration)和deregister(Registration)之类的方法，允许您提供定制的注册服务。Registration是一个marker interface。

每一个ServiceRegistry的实现有他自己的Registry实现

* ZookeeperRegistration used with ZookeeperServiceRegistry
* EurekaRegistration used with EurekaServiceRegistry
* ConsulRegistration used with ConsulServiceRegistry

如果你使用ServiceRegistry接口，你需要将正确的Registry实现传入相应的ServiceRegistry实现中。

### 3.2.1 ServiceRegistry Auto-Registration

默认情况下，ServiceRegistry实现将自动注册正在运行的服务。要禁用该行为，有两种方法。您可以设置@EnableDiscoveryClient(autoRegister=false)永久禁用自动注册。您还可以设置spring.cloud.service-registry.auto-registration.enabled=false以通过配置禁用该行为。

### 3.2.2 Service Registry Actuator Endpoint

Commons提供 /service-registry/ actuator 断点。该端点依赖于Spring context的Registration bean。GET调用/service-registry/instance-status将返回Registration的状态。POST调用会更改Registration的状态。请参阅您正在使用的ServiceRegistry实现的文档，查看哪些值可以修改以及返回值得含义。

## 3.3 Spring RestTemplate as a Load Balancer Client

使用Ribbon会自动配置RestTemplate.通过@LoadBalanced和@Bean可以创建一个负载均衡的RestTemplate

警告：RestTemplate不再通过自动配置创建。它必须被独立的应用创建。

*@Configuration*

**public** **class** MyConfiguration {

*@LoadBalanced*

*@Bean*

RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

}

**public** **class** MyClass {

*@Autowired*

**private** RestTemplate restTemplate;

**public** String doOtherStuff() {

String results = restTemplate.getForObject("http://stores/stores", String.**class**);

**return** results;

}

}

URI需要使用虚拟主机名（即服务名称，而不是主机名）。Ribbon客户端会创建完整的物理地址。有关如何设置RestTemplate的详细信息，请参阅 [RibbonAutoConfiguration](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/ribbon/RibbonAutoConfiguration.java)。

### 3.3.1 Retrying Failed Requests

负载平衡RestTemplate可以配置为重试失败的请求。默认情况下，该逻辑被禁用，您可以通过将[Spring Retry](https://github.com/spring-projects/spring-retry)（spring 子项目）添加到application’s classth中来启用它。负载平衡RestTemplate将会遵守Ribbon配置中的重试规则。如果application’s classth中有Spring Retry，可以通过spring.cloud.loadbalancer.retry.enabled=false禁用重试逻辑。您可以使用的属性有client.ribbon.MaxAutoRetries，client.ribbon.MaxAutoRetriesNextServer和client.ribbon.OkToRetryOnAllOperations。请参阅[Ribbon文档](https://github.com/Netflix/ribbon/wiki/Getting-Started#the-properties-file-sample-clientproperties) ，了解属性的具体内容。

如果你想在重新机制中实现一个BackOffPolicy ，你需要创建一个LoadBalancedBackOffPolicyFactory，并返回指定service对应的BackOffPolicy 。

*@Configuration*

**public** **class** MyConfiguration {

*@Bean*

LoadBalancedBackOffPolicyFactory backOffPolciyFactory() {

**return** **new** LoadBalancedBackOffPolicyFactory() {

*@Override*

**public** BackOffPolicy createBackOffPolicy(String service) {

**return** **new** ExponentialBackOffPolicy();

}

};

}

}

警告：上述示例中的client应替换为您的Ribbon客户端名称。？？？？哪儿有client

如果你想在重新机制中添加一个或多个RetryListener  ，你需要创建一个LoadBalancedRetryListenerFactory ，并返回指定service对应的RetryListener 数组。

*@Configuration*

**public** **class** MyConfiguration {

*@Bean*

LoadBalancedRetryListenerFactory retryListenerFactory() {

**return** **new** LoadBalancedRetryListenerFactory() {

*@Override*

**public** RetryListener[] createRetryListeners(String service) {

**return** **new** RetryListener[]{**new** RetryListener() {

*@Override*

**public** <T, E **extends** Throwable> **boolean** open(RetryContext context, RetryCallback<T, E> callback) {

*//TODO Do you business...*

**return** true;

}

*@Override*

**public** <T, E **extends** Throwable> **void** close(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {

*//TODO Do you business...*

}

*@Override*

**public** <T, E **extends** Throwable> **void** onError(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {

*//TODO Do you business...*

}

}};

}

};

}

}

## 3.4 Multiple RestTemplate objects

如果你想要一个没有负载平衡的RestTemplate，创建一个RestTemplate bean并把它当做一般类注入。如果需要负载均衡RestTemplate，那么创建bean时加@LoadBalanced注解。

重要：注意下面例子中常规 RestTemplate 上 @Primary 注解，用来消除不合格的 @Autowired 注入。

*@Configuration*

**public** **class** MyConfiguration {

*@LoadBalanced*

*@Bean*

RestTemplate loadBalanced() {

**return** **new** RestTemplate();

}

*@Primary*

*@Bean*

RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

}

**public** **class** MyClass {

*@Autowired*

**private** RestTemplate restTemplate;

*@Autowired*

*@LoadBalanced*

**private** RestTemplate loadBalanced;

**public** String doOtherStuff() {

**return** loadBalanced.getForObject("http://stores/stores", String.**class**);

}

**public** String doStuff() {

**return** restTemplate.getForObject("http://example.com", String.**class**);

}

}

**提示**，如果你遇到像这样的错误 java.lang.IllegalArgumentException: Can not set org.springframework.web.client.RestTemplate field com.my.app.Foo.restTemplate to com.sun.proxy.$Proxy89，那么可以尝试注入 RestOperations替换RestTemplate 或设置 spring.aop.proxyTargetClass=true。

## 3.5 Ignore Network Interfaces

有时，忽略某些网络接口很好用，这样可以将其从服务发现注册中排除（例如，在Docker容器中运行）。可以设置正则表达式的列表，这将导致所需的网络接口被忽略。以下配置将忽略“docker0”接口和以“veth”开头的所有接口。

**application.yml.**

spring:

cloud:

inetutils:

ignoredInterfaces:

- docker0

- veth.\*

您还可以强制只允许指定的网络地址

**bootstrap.yml.**

spring:

cloud:

inetutils:

preferredNetworks:

- 192.168

- 10.0

您也可以强制仅使用站点本地地址。有关更多详细信息，请参阅[Inet4Address.html.isSiteLocalAddress（））](https://docs.oracle.com/javase/8/docs/api/java/net/Inet4Address.html#isSiteLocalAddress--)是什么是站点本地地址。

**application.yml.**

spring:

cloud:

inetutils:

useOnlySiteLocalInterfaces: true

## 3.6 HTTP Client Factories

Spring Cloud Commons 提供 bean 来创建 Apache HTTP 客户端 (ApacheHttpClientFactory) 和 OK HTTP 客户端 (OkHttpClientFactory)。OkHttpClientFactory 只在 OK HTTP jar 存在于 classpath 时，才会创建。此外，Spring Cloud Commons 提供 bean 来创建 两种客户端的链接管理，ApacheHttpClientConnectionManagerFactory 为 Apache HTTP 客户端创建管理，OkHttpClientConnectionPoolFactory 为 OK HTTP 客户端创建管理。你可以为这些 bean 提供你自己的实现，如果你希望自定义 HTTP 客户端在下游项目中如何被创建。你也可以禁用这些 bean 创造，通过设置 spring.cloud.httpclientfactories.apache.enabled 或 spring.cloud.httpclientfactories.ok.enabled 为 false。