Spring Cloud Config

目录

* [快速开始](https://springcloud.cc/spring-cloud-config.html#_quick_start)
  + [客户端使用](https://springcloud.cc/spring-cloud-config.html#_client_side_usage)
* [Spring Cloud Config服务器](https://springcloud.cc/spring-cloud-config.html#_spring_cloud_config_server)
  + [环境库](https://springcloud.cc/spring-cloud-config.html#_environment_repository)
  + [健康指标](https://springcloud.cc/spring-cloud-config.html#_health_indicator)
  + [安全](https://springcloud.cc/spring-cloud-config.html#_security)
  + [加密和解密](https://springcloud.cc/spring-cloud-config.html#_encryption_and_decryption)
  + [密钥管理](https://springcloud.cc/spring-cloud-config.html#_key_management)
  + [创建用于测试的密钥库](https://springcloud.cc/spring-cloud-config.html#_creating_a_key_store_for_testing)
  + [使用多个键和键旋转](https://springcloud.cc/spring-cloud-config.html#_using_multiple_keys_and_key_rotation)
  + [提供加密属性](https://springcloud.cc/spring-cloud-config.html#_serving_encrypted_properties)
* [服务替代格式](https://springcloud.cc/spring-cloud-config.html#_serving_alternative_formats)
* [服务纯文本](https://springcloud.cc/spring-cloud-config.html#_serving_plain_text)
* [嵌入配置服务器](https://springcloud.cc/spring-cloud-config.html#_embedding_the_config_server)
* [推送通知和Spring Cloud Bus](https://springcloud.cc/spring-cloud-config.html#_push_notifications_and_spring_cloud_bus)
* [Spring Cloud Config客户端](https://springcloud.cc/spring-cloud-config.html#_spring_cloud_config_client)
  + [配置第一引导](https://springcloud.cc/spring-cloud-config.html#config-first-bootstrap)
  + [发现第一个引导](https://springcloud.cc/spring-cloud-config.html#discovery-first-bootstrap)
  + [配置客户端快速失败](https://springcloud.cc/spring-cloud-config.html#config-client-fail-fast)
  + [配置客户端重试](https://springcloud.cc/spring-cloud-config.html#config-client-retry)
  + [查找远程配置资源](https://springcloud.cc/spring-cloud-config.html#_locating_remote_configuration_resources)
  + [安全](https://springcloud.cc/spring-cloud-config.html#_security_2)
  + [Vault](https://springcloud.cc/spring-cloud-config.html#_vault_2)

**1.3.0.RELEASE**

Spring Cloud Config为分布式系统中的外部配置提供服务器和客户端支持。使用Config Server，您可以在所有环境中管理应用程序的外部属性。客户端和服务器上的概念映射与Spring Environment和PropertySource抽象相同，因此它们与Spring应用程序非常契合，但可以与任何以任何语言运行的应用程序一起使用。随着应用程序通过从开发人员到测试和生产的部署流程，您可以管理这些环境之间的配置，并确定应用程序具有迁移时需要运行的一切。服务器存储后端的默认实现使用git，因此它轻松支持标签版本的配置环境，以及可以访问用于管理内容的各种工具。可以轻松添加替代实现，并使用Spring配置将其插入。

快速开始

启动服务器：

$ cd spring-cloud-config-server

$ ../mvnw spring-boot:run

该服务器是一个Spring Boot应用程序，所以你可以从IDE运行它，而不是喜欢（主类是ConfigServerApplication）。然后尝试一个客户端：

$ curl localhost:8888/foo/development

{"name":"development","label":"master","propertySources":[

{"name":"https://github.com/scratches/config-repo/foo-development.properties","source":{"bar":"spam"}},

{"name":"https://github.com/scratches/config-repo/foo.properties","source":{"foo":"bar"}}

]}

定位资源的默认策略是克隆一个git仓库（在spring.cloud.config.server.git.uri），并使用它来初始化一个迷你SpringApplication。迷你应用程序的Environment用于枚举属性源并通过JSON端点发布。

HTTP服务具有以下格式的资源：

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

其中“应用程序”作为SpringApplication中的spring.config.name注入（即常规Spring Boot应用程序中通常为“应用程序”），“配置文件”是活动配置文件（或逗号分隔列表）的属性），“label”是可选的git标签（默认为“master”）。

Spring Cloud Config服务器从git存储库中提取远程客户端的配置（必须提供）：

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

客户端使用

要在应用程序中使用这些功能，只需将其构建为依赖于spring-cloud-config-client的Spring Boot应用程序（例如，查看配置客户端或示例应用程序的测试用例）。添加依赖关系的最方便的方法是通过Spring Boot启动器org.springframework.cloud:spring-cloud-starter-config。还有一个Maven用户的父pom和BOM（spring-cloud-starter-parent）和用于Gradle和Spring CLI用户的Spring IO版本管理属性文件。示例Maven配置：

*的pom.xml*

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.3.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Brixton.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<!-- repositories also needed for snapshots and milestones -->

那么你可以创建一个标准的Spring Boot应用程序，就像这个简单的HTTP服务器：

@SpringBootApplication

@RestController

public class Application {

@RequestMapping("/")

public String home() {

return "Hello World!";

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

当它运行它将从端口8888上的默认本地配置服务器接收外部配置，如果它正在运行。要修改启动行为，您可以使用bootstrap.properties（如application.properties，但是应用程序上下文的引导阶段）更改配置服务器的位置，例如

spring.cloud.config.uri: http://myconfigserver.com

引导属性将在/env端点中显示为高优先级属性源，例如

$ curl localhost:8080/env

{

"profiles":[],

"configService:https://github.com/spring-cloud-samples/config-repo/bar.properties":{"foo":"bar"},

"servletContextInitParams":{},

"systemProperties":{...},

...

}

（名为“configService：<远程存储库的URL> / <文件名>”的属性源包含值为“bar”的属性“foo”，是最高优先级）。

|  |  |
| --- | --- |
| **注意** | 属性源名称中的URL是git存储库，而不是配置服务器URL。 |

Spring Cloud Config服务器

服务器为外部配置（名称值对或等效的YAML内容）提供了基于资源的HTTP。服务器可以使用@EnableConfigServer注释轻松嵌入到Spring Boot应用程序中。所以这个应用程序是一个配置服务器：

*ConfigServer.java*

@SpringBootApplication

@EnableConfigServer

public class ConfigServer {

public static void main(String[] args) {

SpringApplication.run(ConfigServer.class, args);

}

}

像所有默认情况下在端口8080上运行的所有Spring Boot应用程序一样，但您可以通过各种方式将其切换到常规端口8888。最简单的设置默认配置存储库是通过使用spring.config.name=configserver启动（在Config Server jar中有一个configserver.yml）。另一个是使用你自己的application.properties，例如

*application.properties*

server.port: 8888

spring.cloud.config.server.git.uri: file://${user.home}/config-repo

其中${user.home}/config-repo是包含YAML和属性文件的git仓库。

|  |  |
| --- | --- |
| **注意** | 在Windows中，如果文件URL为绝对带有驱动器前缀，例如file:///${user.home}/config-repo，则需要额外的“/”。 |
| **小费** | 以下是上面示例中创建git仓库的方法：  $ cd $HOME  $ mkdir config-repo  $ cd config-repo  $ git init .  $ echo info.foo: bar > application.properties  $ git add -A .  $ git commit -m "Add application.properties" |

|  |  |
| --- | --- |
| **警告** | 使用本地文件系统进行git存储库仅用于测试。使用服务器在生产环境中托管配置库。 |
| **警告** | 如果您只保留文本文件，则配置库的初始克隆将会快速有效。如果您开始存储二进制文件，尤其是较大的文件，则可能会遇到服务器中第一个配置请求和/或内存不足错误的延迟。 |

环境库

您要在哪里存储配置服务器的配置数据？管理此行为的策略是EnvironmentRepository，服务于Environment对象。这个Environment是Spring Environment（包括propertySources作为主要功能）的域的浅层副本。Environment资源由三个变量参数化：

* {application}映射到客户端的“spring.application.name”;
* {profile}映射到客户端上的“spring.profiles.active”（逗号分隔列表）; 和
* {label}这是一个服务器端功能，标记“版本”的一组配置文件。

存储库实现通常表现得像一个Spring Boot应用程序从“spring.config.name”等于{application}参数加载配置文件，“spring.profiles.active”等于{profiles}参数。配置文件的优先级规则也与常规启动应用程序中的相同：活动配置文件优先于默认配置，如果存在多个配置文件，则最后一个配置文件（例如向Map添加条目））。

示例：客户端应用程序具有此引导配置：

*bootstrap.yml*

spring:

application:

name: foo

profiles:

active: dev,mysql

（像往常一样使用Spring Boot应用程序，这些属性也可以设置为环境变量或命令行参数）。

如果存储库是基于文件的，则服务器将从application.yml创建Environment（在所有客户端之间共享），foo.yml（以foo.yml优先））。如果YAML文件在其中具有指向Spring配置文件的文档，则应用较高优先级（按照列出的配置文件的顺序），并且如果存在特定于配置文件的YAML（或属性）文件，那么这些文件也应用的优先级高于默认值。较高优先级转换为Environment之前列出的PropertySource。（这些规则与独立的Spring Boot应用程序相同。）

Git后端

EnvironmentRepository的默认实现使用Git后端，这对于管理升级和物理环境以及审核更改非常方便。要更改存储库的位置，可以在Config Server中设置“spring.cloud.config.server.git.uri”配置属性（例如application.yml）。如果您使用file:前缀进行设置，则应从本地存储库中运行，以便在没有服务器的情况下快速方便地启动，但在这种情况下，服务器将直接在本地存储库中进行操作，而不会克隆它如果它不是裸机，因为配置服务器永远不会更改“远程”资源库）。要扩展Config Server并使其高度可用，您需要将服务器的所有实例指向同一个存储库，因此只有共享文件系统才能正常工作。即使在这种情况下，最好使用共享文件系统存储库的ssh:协议，以便服务器可以将其克隆并使用本地工作副本作为缓存。

该存储库实现将HTTP资源的{label}参数映射到git标签（提交ID，分支名称或标签）。如果git分支或标签名称包含斜杠（“/”），则应使用特殊字符串“（\_）”指定HTTP URL中的标签，以避免与其他URL路径模糊。例如，如果标签是foo/bar，则替换斜杠将导致类似于foo(\_)bar的标签。如果您使用像curl这样的命令行客户端（例如使用引号将其从shell中转出来），请小心URL中的方括号。

Git URI中的占位符

Spring Cloud Config服务器支持一个Git仓库URL，其中包含{application}和{profile}（以及{label}）的占位符，如果需要，请记住，标签应用为git标签）。因此，您可以使用（例如）轻松支持“每个应用程序的一个repo”策略：

spring:

cloud:

config:

server:

git:

uri: https://github.com/myorg/{application}

或使用类似模式“{一个回报每个配置文件”策略，但使用{profile}。

模式匹配和多个存储库

还可以通过应用程序和配置文件名称的模式匹配来支持更复杂的需求。模式格式是带有通配符的{application}/{profile}名称的逗号分隔列表（其中以通配符开头的模式可能需要引用）。例：

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

repos:

simple: https://github.com/simple/config-repo

special:

pattern: special\*/dev\*,\*special\*/dev\*

uri: https://github.com/special/config-repo

local:

pattern: local\*

uri: file:/home/configsvc/config-repo

如果{application}/{profile}不匹配任何模式，它将使用在“spring.cloud.config.server.git.uri”下定义的默认uri。在上面的例子中，对于“简单”存储库，模式是simple/\*（即所有配置文件中只匹配一个名为“简单”的应用程序）。“本地”存储库与所有配置文件中以“local”开头的所有应用程序名称匹配（将/\*后缀自动添加到任何没有配置文件匹配器的模式）。

|  |  |
| --- | --- |
| **注意** | 在上述“简单”示例中使用的“单行”快捷方式只能在唯一要设置的属性为URI的情况下使用。如果您需要设置其他任何内容（凭据，模式等），则需要使用完整的表单。 |

repo中的pattern属性实际上是一个数组，因此您可以使用属性文件中的YAML数组（或[0]，[1]等后缀）绑定到多个模式。如果要运行具有多个配置文件的应用程序，则可能需要执行此操作。例：

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

repos:

development:

pattern:

- \*/development

- \*/staging

uri: https://github.com/development/config-repo

staging:

pattern:

- \*/qa

- \*/production

uri: https://github.com/staging/config-repo

|  |  |
| --- | --- |
| **注意** | Spring Cloud将猜测包含不在\*中的配置文件的模式意味着您实际上希望匹配从该模式开始的配置文件列表（所以\*/staging是["\*/staging", "\*/staging,\*"]的快捷方式） 。这是常见的，您需要在本地的“开发”配置文件中运行应用程序，但也可以远程运行“云”配置文件。 |

每个存储库还可以选择将配置文件存储在子目录中，搜索这些目录的模式可以指定为searchPaths。例如在顶层：

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

searchPaths: foo,bar\*

在此示例中，服务器搜索顶级和“foo /”子目录以及名称以“bar”开头的任何子目录中的配置文件。

默认情况下，首次请求配置时，服务器克隆远程存储库。服务器可以配置为在启动时克隆存储库。例如在顶层：

spring:

cloud:

config:

server:

git:

uri: https://git/common/config-repo.git

repos:

team-a:

pattern: team-a-\*

cloneOnStart: true

uri: http://git/team-a/config-repo.git

team-b:

pattern: team-b-\*

cloneOnStart: false

uri: http://git/team-b/config-repo.git

team-c:

pattern: team-c-\*

uri: http://git/team-a/config-repo.git

在此示例中，服务器在启动之前克隆了team-a的config-repo，然后它接受任何请求。所有其他存储库将不被克隆，直到请求从存储库配置。

|  |  |
| --- | --- |
| **注意** | 在配置服务器启动时设置要克隆的存储库可以帮助在配置服务器启动时快速识别错误配置的源（例如，无效的存储库URI）。配置源不启用cloneOnStart时，配置服务器可能启动成功配置错误或无效的配置源，并且不会检测到错误，直到应用程序从该配置源请求配置为止。 |

认证

要在远程存储库上使用HTTP基本身份验证，请分别添加“username”和“password”属性（不在URL中），例如

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

username: trolley

password: strongpassword

如果您不使用HTTPS和用户凭据，则在将密钥存储在默认目录（~/.ssh）中，并且uri指向SSH位置时，SSH也应该开箱即用，例如“ [git@github.com](mailto:git@github.com)：配置/云配置”。重要的是，Git服务器的条目存在于~/.ssh/known\_hosts文件中，并且格式为ssh-rsa。不支持其他格式（如ecdsa-sha2-nistp256）。为了避免意外，您应该确保Git服务器的known\_hosts文件中只有一个条目存在，并且与您提供给配置服务器的URL匹配。如果您在URL中使用了一个主机名，那么您希望在known\_hosts文件中具有该主机名，而不是IP。使用JGit访问存储库，因此您发现的任何文档都应适用。HTTPS代理设置可以~/.git/config设置，也可以通过系统属性（-Dhttps.proxyHost和-Dhttps.proxyPort）与任何其他JVM进程相同。

|  |  |
| --- | --- |
| **小费** | 如果您不知道~/.git目录在何处使用git config --global操纵设置（例如git config --global http.sslVerify false）。 |

使用AWS CodeCommit进行认证

[AWS CodeCommit](https://docs.aws.amazon.com/codecommit/latest/userguide/welcome.html)认证也可以完成。当从命令行使用Git时，AWS CodeCommit使用身份验证助手。该帮助器不与JGit库一起使用，因此如果Git URI与AWS CodeCommit模式匹配，则将创建用于AWS CodeCommit的JGit CredentialProvider。AWS CodeCommit URI始终看起来像 https：//git-codecommit.$ {AWS\_REGION} .amazonaws.com / $ {repopath}。

如果您使用AWS CodeCommit URI提供用户名和密码，那么这些URI必须 是用于访问存储库的[AWS accessKeyId和](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSGettingStartedGuide/AWSCredentials.html)secretAccessKey。如果不指定用户名和密码，则将使用[AWS默认凭据提供程序链](https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/credentials.html)检索accessKeyId和secretAccessKey 。

如果您的Git URI与CodeCommit URI模式（上述）匹配，则必须在用户名和密码或默认凭据提供程序链支持的某个位置中提供有效的AWS凭据。AWS EC2实例可以使用EC2实例的 [IAM角色](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html)。

注意：aws-java-sdk-core jar是一个可选的依赖关系。如果aws-java-sdk-core jar不在您的类路径上，则无论git服务器URI如何，都将不会创建AWS代码提交凭据提供程序。

Git搜索路径中的占位符

Spring Cloud Config服务器还支持{application}和{profile}（以及{label}（如果需要））占位符的搜索路径。例：

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

searchPaths: '{application}'

在资源库中搜索与目录（以及顶级）相同名称的文件。通配符在具有占位符的搜索路径中也是有效的（搜索中包含任何匹配的目录）。

力拉入Git存储库

如前所述Spring Cloud Config服务器克隆远程git存储库，如果某种方式本地副本变脏（例如，由操作系统进程更改文件夹内容），则Spring Cloud Config服务器无法从远程存储库更新本地副本。

要解决这个问题，有一个force-pull属性将使Spring Cloud Config Server强制从远程存储库中提取，如果本地副本是脏的。例：

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

force-pull: true

如果您有多个存储库配置，则可以为每个存储库配置force-pull属性。例：

spring:

cloud:

config:

server:

git:

uri: https://git/common/config-repo.git

force-pull: true

repos:

team-a:

pattern: team-a-\*

uri: http://git/team-a/config-repo.git

force-pull: true

team-b:

pattern: team-b-\*

uri: http://git/team-b/config-repo.git

force-pull: true

team-c:

pattern: team-c-\*

uri: http://git/team-a/config-repo.git

|  |  |
| --- | --- |
| **注意** | force-pull属性的默认值为false。 |

版本控制后端文件系统使用

|  |  |
| --- | --- |
| **警告** | 使用基于VCS的后端（git，svn）文件被检出或克隆到本地文件系统。默认情况下，它们放在系统临时目录中，前缀为config-repo-。在linux上，例如可以是/tmp/config-repo-<randomid>。一些操作系统会[定期清除](https://serverfault.com/questions/377348/when-does-tmp-get-cleared/377349#377349)临时目录。这可能会导致意外的行为，例如缺少属性。为避免此问题，请通过将spring.cloud.config.server.git.basedir或spring.cloud.config.server.svn.basedir设置为不在系统临时结构中的目录来更改Config Server所使用的目录。 |

文件系统后端

配置服务器中还有一个不使用Git的“本机”配置文件，只是从本地类路径或文件系统加载配置文件（您想要指向的任何静态URL“spring.cloud.config.server .native.searchLocations“）。要使用本机配置文件，只需使用“spring.profiles.active = native”启动Config Server。

|  |  |
| --- | --- |
| **注意** | 记住使用file:前缀的文件资源（缺省没有前缀通常是classpath）。与任何Spring Boot配置一样，您可以嵌入${}样式的环境占位符，但请记住，Windows中的绝对路径需要额外的“/”，例如file:///${user.home}/config-repo |
| **警告** | searchLocations的默认值与本地Spring Boot应用程序（所以[classpath:/, classpath:/config, file:./, file:./config]）相同。这不会将application.properties从服务器暴露给所有客户端，因为在发送到客户端之前，服务器中存在的任何属性源都将被删除。 |

|  |  |
| --- | --- |
| **小费** | 文件系统后端对于快速入门和测试是非常好的。要在生产中使用它，您需要确保文件系统是可靠的，并在配置服务器的所有实例中共享。 |

搜索位置可以包含{application}，{profile}和{label}的占位符。以这种方式，您可以隔离路径中的目录，并选择一个有用的策略（例如每个应用程序的子目录或每个配置文件的子目录）。

如果您不在搜索位置使用占位符，则该存储库还将HTTP资源的{label}参数附加到搜索路径上的后缀，因此属性文件将从每个搜索位置加载**并**具有相同名称的子目录作为标签（标签属性在Spring环境中优先）。因此，没有占位符的默认行为与添加以/{label}/. For example `file:/tmp/config结尾的搜索位置的添加与file:/tmp/config,file:/tmp/config/{label}相同

Vault后端

Spring Cloud Config服务器还支持[Vault](https://www.vaultproject.io/)作为后端。

Vault是安全访问秘密的工具。一个秘密是你想要严格控制访问的任何东西，如API密钥，密码，证书等等。Vault为任何秘密提供统一的界面，同时提供严格的访问控制和记录详细的审核日志。

有关Vault的更多信息，请参阅[Vault快速入门指南](https://www.vaultproject.io/intro/index.html)。

要使配置服务器使用Vault后端，必须使用vault配置文件运行配置服务器。例如在配置服务器的application.properties中，您可以添加spring.profiles.active=vault。

默认情况下，配置服务器将假定您的Vault服务器正在运行[http://127.0.0.1:8200](http://127.0.0.1:8200/)。它还将假设后端名称为secret，密钥为application。所有这些默认值都可以在配置服务器的application.properties中配置。以下是可配置的Vault属性表。所有属性前缀为spring.cloud.config.server.vault。

| **名称** | **默认值** |
| --- | --- |
| 主办 | 127.0.0.1 |
| 港口 | 8200 |
| 方案 | HTTP |
| 后端 | 秘密 |
| defaultKey | 应用 |
| profileSeparator | , |

所有可配置的属性可以在org.springframework.cloud.config.server.environment.VaultEnvironmentRepository中找到。

运行您的配置服务器，您可以向服务器发出HTTP请求，以从Vault后端检索值。为此，您将需要一个令牌为您的Vault服务器。

首先将一些数据放在你身上Vault。例如

$ vault write secret/application foo=bar baz=bam

$ vault write secret/myapp foo=myappsbar

现在，将HTTP请求发送给您的配置服务器以检索值。

$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"

在提出上述要求后，您应该会看到类似的回复。

{

"name":"myapp",

"profiles":[

"default"

],

"label":null,

"version":null,

"state":null,

"propertySources":[

{

"name":"vault:myapp",

"source":{

"foo":"myappsbar"

}

},

{

"name":"vault:application",

"source":{

"baz":"bam",

"foo":"bar"

}

}

]

}

多个属性来源

使用Vault时，您可以为应用程序提供多个属性源。例如，假设您已将数据写入Vault中的以下路径。

secret/myApp,dev

secret/myApp

secret/application,dev

secret/application

写入secret/application的属性可 [用于使用配置服务器的所有应用程序](https://springcloud.cc/spring-cloud-config.html#_vault_server)。名称为myApp的应用程序将具有写入secret/myApp和secret/application的任何属性。当myApp启用dev配置文件时，写入所有上述路径的属性将可用，列表中第一个路径中的属性优先于其他路径。

与所有应用共享配置

基于文件的存储库

使用基于文件（即git，svn和native）的存储库，所有客户端应用程序（application.properties，application.yml，application-\*.properties等））共享文件名为application\*的资源。您可以使用这些文件名的资源来配置全局默认值，并根据需要将其覆盖应用程序特定的文件。

#\_property\_overrides [属性覆盖]功能也可用于设置全局默认值，并且允许占位符应用程序在本地覆盖它们。

|  |  |
| --- | --- |
| **小费** | 使用“本机”配置文件（本地文件系统后端），建议您使用不属于服务器自身配置的显式搜索位置。否则，默认搜索位置中的application\*资源将被删除，因为它们是服务器的一部分。 |

Vault服务器

当使用Vault作为后端时，您可以通过将配置放在secret/application中与所有应用程序共享配置。例如，如果您运行此Vault命令

$ vault write secret/application foo=bar baz=bam

使用配置服务器的所有应用程序都可以使用属性foo和baz。

复合环境库

在某些情况下，您可能希望从多个环境存储库中提取配置数据。为此，只需在配置服务器的应用程序属性或YAML文件中启用多个配置文件即可。例如，如果您要从Git存储库以及SVN存储库中提取配置数据，那么您将为配置服务器设置以下属性。

spring:

profiles:

active: git, svn

cloud:

config:

server:

svn:

uri: file:///path/to/svn/repo

order: 2

git:

uri: file:///path/to/git/repo

order: 1

除了指定URI的每个repo之外，还可以指定order属性。order属性允许您指定所有存储库的优先级顺序。order属性的数值越低，优先级越高。存储库的优先顺序将有助于解决包含相同属性的值的存储库之间的任何潜在冲突。

|  |  |
| --- | --- |
| **注意** | 从环境仓库检索值时的任何类型的故障将导致整个复合环境的故障。 |
| **注意** | 当使用复合环境时，重要的是所有repos都包含相同的标签。如果您有类似于上述的环境，并且使用标签master请求配置数据，但是SVN repo不包含称为master的分支，则整个请求将失败。 |

自定义复合环境库

除了使用Spring Cloud中的一个环境存储库之外，还可以提供自己的EnvironmentRepository bean作为复合环境的一部分。要做到这一点，你的bean必须实现EnvironmentRepository接口。如果您想在复合环境中控制自定义EnvironmentRepository的优先级，您还应该实现Ordered接口并覆盖getOrdered方法。如果您不实现Ordered接口，则您的EnvironmentRepository将被赋予最低优先级。

属性覆盖

Config Server具有“覆盖”功能，允许操作员为所有应用程序提供配置属性，这些应用程序不会被应用程序使用正常的Spring Boot钩子意外更改。要声明覆盖只是将名称/值对的地图添加到spring.cloud.config.server.overrides。例如

spring:

cloud:

config:

server:

overrides:

foo: bar

将导致配置客户端的所有应用程序独立于自己的配置读取foo=bar。（当然，应用程序可以以任何方式使用Config Server中的数据，因此覆盖不可强制执行，但如果它们是Spring Cloud Config客户端，则它们会提供有用的默认行为。）

|  |  |
| --- | --- |
| **小费** | 通常使用“$ {}”的Spring环境占位符可以使用反斜杠（“\”）来转义（并在客户端上解析）来转义“$”或“{”，例如\${app.foo:bar}解析为“bar “除非该应用程序提供自己的”app.foo“。请注意，在YAML中，您不需要转义反斜杠本身，而是在您执行的属性文件中配置服务器上的覆盖。 |

您可以将客户端中所有覆盖的优先级更改为更为默认值，允许应用程序通过在远程存储库中设置标志spring.cloud.config.overrideNone=true（默认值为false），在环境变量或系统属性中提供自己的值。

健康指标

配置服务器附带运行状况指示器，检查配置的EnvironmentRepository是否正常工作。默认情况下，它要求EnvironmentRepository应用程序名为app，default配置文件和EnvironmentRepository实现提供的默认标签。

您可以配置运行状况指示器以检查更多应用程序以及自定义配置文件和自定义标签，例如

spring:

cloud:

config:

server:

health:

repositories:

myservice:

label: mylabel

myservice-dev:

name: myservice

profiles: development

您可以通过设置spring.cloud.config.server.health.enabled=false来禁用运行状况指示器。

安全

您可以以任何对您有意义的方式（从物理网络安全性到OAuth2承载令牌）来保护您的Config Server，而Spring Security和Spring Boot可以轻松做任何事情。

要使用默认的Spring Boot配置的HTTP Basic安全性，只需在类路径中包含Spring Security（例如，通过spring-boot-starter-security）。默认是“user”的用户名和随机生成的密码，这在实践中不会非常有用，因此我们建议您配置密码（通过security.user.password）进行加密（有关以下内容的说明，请参阅以下内容）怎么做）。

加密和解密

|  |  |
| --- | --- |
| **重要** | **先决条件：**要使用加密和解密功能，您需要在JVM中安装全面的JCE（默认情况下不存在）。您可以从Oracle下载“Java加密扩展（JCE）无限强度管理策略文件”，并按照安装说明（实际上将JRE lib / security目录中的2个策略文件替换为您下载的文件）。 |

如果远程属性源包含加密内容（以{cipher}开头的值）），则在通过HTTP发送给客户端之前，它们将被解密。这种设置的主要优点是，当它们“静止”时，属性值不必是纯文本（例如在git仓库中）。如果值无法解密，则从属性源中删除该值，并添加具有相同键的附加属性，但以“无效”作为前缀。和“不适用”的值（通常为“<n / a>”）。这主要是为了防止密码被用作密码并意外泄漏。

如果要为配置客户端应用程序设置远程配置存储库，可能会包含一个application.yml，例如：

*application.yml*

spring:

datasource:

username: dbuser

password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'

.properties文件中的加密值不能用引号括起来，否则不会解密该值：

*application.properties*

spring.datasource.username: dbuser

spring.datasource.password: {cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ

您可以安全地将此纯文本推送到共享git存储库，并且保密密码。

服务器还暴露了/encrypt和/decrypt端点（假设这些端点将被保护并且只能由授权代理访问）。如果您正在编辑远程配置文件，可以使用Config Server通过POST到/encrypt端点来加密值，例如

$ curl localhost:8888/encrypt -d mysecret

682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

|  |  |
| --- | --- |
| **注意** | 如果要加密的值具有需要进行URL编码的字符，则应使用--data-urlencode选项curl来确保它们已正确编码。 |

逆向操作也可通过/decrypt获得（如果服务器配置了对称密钥或完整密钥对）：

$ curl localhost:8888/decrypt -d 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

mysecret

|  |  |
| --- | --- |
| **小费** | 如果您使用curl进行测试，则使用--data-urlencode（而不是-d）或设置显式Content-Type: text/plain，以确保在有特殊字符时正确地对数据进行编码（'+'特别是棘手）。 |

在加入值之前，先添加{cipher}前缀，然后再将其放在YAML或属性文件中，然后再提交并将其推送到远程，可能不安全的存储区。

/encrypt和/decrypt端点也都接受/\*/{name}/{profiles}形式的路径，当客户端调用到主环境资源时，可以使用每个应用程序（名称）和配置文件控制密码。

|  |  |
| --- | --- |
| **注意** | 为了以这种细微的方式控制密码，您还必须提供TextEncryptorLocator类型的@Bean，每个名称和配置文件都会创建一个不同的加密器。默认提供的不会这样做（所有加密使用相同的密钥）。 |

spring命令行客户端（安装了Spring Cloud CLI扩展名）也可用于加密和解密，例如

$ spring encrypt mysecret --key foo

682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

$ spring decrypt --key foo 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

mysecret

要在文件中使用密钥（例如用于加密的RSA公钥），使用“@”键入键值，并提供文件路径，例如

$ spring encrypt mysecret --key @${HOME}/.ssh/id\_rsa.pub

AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...

关键参数是强制性的（尽管有一个--前缀）。

密钥管理

Config Server可以使用对称（共享）密钥或非对称密钥（RSA密钥对）。非对称选择在安全性方面是优越的，但是使用对称密钥往往更方便，因为它只是配置的一个属性值。

要配置对称密钥，您只需要将encrypt.key设置为一个秘密字符串（或使用环境变量ENCRYPT\_KEY将其保留为纯文本配置文件）。

要配置非对称密钥，您可以将密钥设置为PEM编码的文本值（encrypt.key）或通过密钥库（例如由JDK附带的keytool实用程序创建）。密钥库属性为encrypt.keyStore.\*，\*等于

* location（Resource位置），
* password（解锁密钥库）和
* alias（以识别商店中使用的密钥）。

使用公钥进行加密，需要私钥进行解密。因此，原则上您只能在服务器中配置公钥，如果您只想进行加密（并准备使用私钥本地解密值）。实际上，您可能不想这样做，因为它围绕所有客户端传播密钥管理流程，而不是将其集中在服务器中。另一方面，如果您的配置服务器真的相对不安全，并且只有少数客户端需要加密的属性，这是一个有用的选项。

创建用于测试的密钥库

要创建一个密钥库进行测试，您可以执行以下操作：

$ keytool -genkeypair -alias mytestkey -keyalg RSA \

-dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \

-keypass changeme -keystore server.jks -storepass letmein

将server.jks文件放在类路径（例如）中，然后在配置服务器的application.yml中：

encrypt:

keyStore:

location: classpath:/server.jks

password: letmein

alias: mytestkey

secret: changeme

使用多个键和键旋转

除了加密属性值中的{cipher}前缀之外，配置服务器在（Base64编码）密文开始之前查找{name:value}前缀（零或多个）。密钥被传递给TextEncryptorLocator，它可以执行找到密码的TextEncryptor所需的任何逻辑。如果配置了密钥库（encrypt.keystore.location），默认定位器将使用“key”前缀提供的别名，即使用如下密码来查找存储中的密钥：

foo:

bar: `{cipher}{key:testkey}...`

定位器将寻找一个名为“testkey”的键。也可以通过前缀中的{secret:…​}值提供一个秘密，但是如果不是默认值，则使用密钥库密码（这是您构建密钥存储并且不指定密码时获得的密码）。如果你**这样做** 提供一个秘密建议你也加密使用自定义SecretLocator的秘密。

如果密钥只用于加密几个字节的配置数据（即它们没有在其他地方使用），则密码转换几乎不是必需的，但是如果存在安全漏洞，有时您可能需要更改密钥实例。在这种情况下，所有客户端都需要更改其源配置文件（例如git），并在所有密码中使用新的{key:…​}前缀，当然，先检查配置服务器密钥库中的密钥别名可用。

|  |  |
| --- | --- |
| **小费** | 如果要让Config Server处理所有加密以及解密，也可以将{name:value}前缀添加到发布到/encrypt端点的明文。 |

提供加密属性

有时您希望客户端在本地解密配置，而不是在服务器中进行配置。在这种情况下，您仍然可以拥有/加密和/解密端点（如果提供encrypt.\*配置来定位密钥），但是您需要使用spring.cloud.config.server.encrypt.enabled=false明确地关闭传出属性的解密。如果您不关心端点，那么如果您既不配置密钥也不配置使能的标志，则应该起作用。

服务替代格式

来自环境端点的默认JSON格式对于Spring应用程序的消耗是完美的，因为它直接映射到Environment抽象。如果您喜欢，可以通过向资源路径（“.yml”，“.yaml”或“.properties”）添加后缀来使用与YAML或Java属性相同的数据。这对于不关心JSON端点的结构的应用程序的消费或其提供的额外的元数据可能是有用的，例如，不使用Spring的应用程序可能会受益于此方法的简单性。

YAML和属性表示有一个额外的标志（作为布尔查询参数resolvePlaceholders提供）），以标示Spring ${…​}形式的原始文档中的占位符信号，在渲染之前应尽可能在输出中解析。这对于不了解Spring占位符约定的消费者来说是一个有用的功能。

|  |  |
| --- | --- |
| **注意** | 使用YAML或属性格式存在局限性，主要是与元数据的丢失有关。JSON被构造为属性源的有序列表，例如，名称与源相关联。即使源的起源具有多个源，并且原始源文件的名称丢失，YAML和属性表也合并成一个映射。YAML表示不一定是后台存储库中YAML源的忠实表示：它是由平面属性源的列表构建的，并且必须对键的形式进行假设。 |

服务纯文本

您的应用程序可能不需要使用Environment抽象（或YAML中的其他替代表示形式或属性格式），而是根据自己的环境需要通用的纯文本配置文件。配置服务器通过/{name}/{profile}/{label}/{path}中的附加端点提供这些服务，其中“name”，“profile”和“label”的含义与常规环境端点相同，但“path”是文件名（例如log.xml ）。此端点的源文件位于与环境端点相同的方式：与属性或YAML文件相同的搜索路径，而不是聚合所有匹配的资源，只返回匹配的第一个。

资源位置后，使用正常格式（${…​}）的占位符使用提供的应用程序名称，配置文件和标签的有效Environment来解析。以这种方式，资源端点与环境端点紧密集成。例如，如果您有一个GIT（或SVN）资源库的布局：

application.yml

nginx.conf

其中nginx.conf看起来像这样：

server {

listen 80;

server\_name ${nginx.server.name};

}

和application.yml这样：

nginx:

server:

name: example.com

---

spring:

profiles: development

nginx:

server:

name: develop.com

那么/foo/default/master/nginx.conf资源如下所示：

server {

listen 80;

server\_name example.com;

}

和/foo/development/master/nginx.conf如下所示：

server {

listen 80;

server\_name develop.com;

}

|  |  |
| --- | --- |
| **注意** | 就像环境配置的源文件一样，“配置文件”用于解析文件名，所以如果你想要一个特定于文件的文件，那么/\*/development/\*/logback.xml将被一个名为logback-development.xml的文件解析（优先于logback.xml）。 |

嵌入配置服务器

配置服务器最好作为独立应用程序运行，但如果需要，可以将其嵌入到另一个应用程序中。只需使用@EnableConfigServer注释。在这种情况下可以使用的可选属性是spring.cloud.config.server.bootstrap，它是一个标志，表示服务器应该从自己的远程存储库配置自身。该标志默认关闭，因为它可能会延迟启动，但是当嵌入在另一个应用程序中时，以与其他应用程序相同的方式初始化是有意义的。

|  |  |
| --- | --- |
| **注意** | 应该很明显，但是请记住，如果使用引导标志，配置服务器将需要在bootstrap.yml中配置其名称和存储库URI。 |

要更改服务器端点的位置，您可以（可选）设置spring.cloud.config.server.prefix，例如“/ config”，以提供前缀下的资源。前缀应该开始但不以“/”结尾。它应用于Config Server中的@RequestMappings（即Spring Boot前缀server.servletPath和server.contextPath）之下。

如果您想直接从后端存储库（而不是从配置服务器）读取应用程序的配置，这基本上是一个没有端点的嵌入式配置服务器。如果不使用@EnableConfigServer注释（只设置spring.cloud.config.server.bootstrap=true），则可以完全关闭端点。

推送通知和Spring Cloud Bus

许多源代码存储库提供程序（例如Github，Gitlab或Bitbucket）将通过webhook通知您存储库中的更改。您可以通过提供商的用户界面将webhook配置为URL和一组感兴趣的事件。例如， [Github](https://developer.github.com/v3/activity/events/types/#pushevent) 将使用包含提交列表的JSON主体和“X-Github-Event”等于“push”的头文件发送到webhook。如果您在spring-cloud-config-monitor库中添加依赖关系并激活配置服务器中的Spring Cloud Bus，则启用“/ monitor”端点。

当Webhook被激活时，配置服务器将发送一个RefreshRemoteApplicationEvent针对他认为可能已经改变的应用程序。变更检测可以进行策略化，但默认情况下，它只是查找与应用程序名称匹配的文件的更改（例如，“foo.properties”针对的是“foo”应用程序，“application.properties”针对所有应用程序） 。如果要覆盖该行为的策略是PropertyPathNotificationExtractor，它接受​​请求标头和正文作为参数，并返回更改的文件路径列表。

默认配置与Github，Gitlab或Bitbucket配合使用。除了Github，Gitlab或Bitbucket的JSON通知之外，您还可以通过使用表单编码的身体参数path={name}通过POST为“/ monitor”来触发更改通知。这将广播到匹配“{name}”模式的应用程序（可以包含通配符）。

|  |  |
| --- | --- |
| **注意** | 只有在配置服务器和客户端应用程序中激活spring-cloud-bus时才会传送RefreshRemoteApplicationEvent。 |
| **注意** | 默认配置还检测本地git存储库中的文件系统更改（在这种情况下不使用webhook，但是一旦编辑配置文件，将会播放刷新）。 |

Spring Cloud Config客户端

一个Spring Boot应用程序可以立即利用Spring Config Server（或应用程序开发人员提供的其他外部属性源），并且还将获取与Environment更改事件相关的一些其他有用功能。

配置第一引导

这是在类路径中具有Spring Cloud Config Client的任何应用程序的默认行为。当配置客户端启动时，它将通过配置服务器（通过引导配置属性spring.cloud.config.uri）绑定并使用远程属性源初始化Spring Environment。

这样做的最终结果是所有想要使用配置服务器的客户端应用程序都需要spring.cloud.config.uri（默认为“http：// localhost：8888”）的bootstrap.yml（或环境变量） ）。

发现第一个引导

如果您正在使用“DiscoveryClient”实现，例如Spring Cloud Netflix和Eureka Service Discovery或Spring Cloud Consul（Spring Cloud Zookeeper还不支持此功能），那么您可以让Config Server注册Discovery Discovery，但在默认的“配置优先”模式下，客户端将无法利用注册。

如果您希望使用DiscoveryClient找到Config Server，可以通过设置spring.cloud.config.discovery.enabled=true（默认为“false”）来实现。最终的结果是客户端应用程序都需要具有适当的发现配置的bootstrap.yml（或环境变量）。例如，使用Spring Cloud Netflix，您需要定义Eureka服务器地址，例如eureka.client.serviceUrl.defaultZone。使用此选项的价格是启动时额外的网络往返，以定位服务注册。好处是配置服务器可以更改其坐标，只要发现服务是一个固定点。默认的服务标识是“configserver”，但是您可以使用spring.cloud.config.discovery.serviceId在客户端上进行更改（在服务器上可以通过设置spring.application.name以通常方式更改服务。

发现客户端实现都支持某种元数据映射（例如，对于Eureka，我们有eureka.instance.metadataMap）。可能需要在其服务注册元数据中配置Config Server的一些其他属性，以便客户端可以正确连接。如果使用HTTP Basic安全配置服务器，则可以将凭据配置为“用户名”和“密码”。并且如果配置服务器具有上下文路径，您可以设置“configPath”。例如，对于作为Eureka客户端的配置服务器：

*bootstrap.yml*

eureka:

instance:

...

metadataMap:

user: osufhalskjrtl

password: lviuhlszvaorhvlo5847

configPath: /config

配置客户端快速失败

在某些情况下，如果服务无法连接到配置服务器，则可能希望启动服务失败。如果这是所需的行为，请设置引导配置属性spring.cloud.config.failFast=true，客户端将以异常停止。

配置客户端重试

如果您希望配置服务器在您的应用程序启动时可能偶尔不可用，您可以要求它在发生故障后继续尝试。首先，您需要设置spring.cloud.config.failFast=true，然后您需要将spring-retry和spring-boot-starter-aop添加到您的类路径中。默认行为是重试6次，初始退避间隔为1000ms，指数乘数为1.1，用于后续退避。您可以使用spring.cloud.config.retry.\*配置属性配置这些属性（和其他）。

|  |  |
| --- | --- |
| **小费** | 要完全控制重试，请使用id“configServerRetryInterceptor”添加RetryOperationsInterceptor类型的@Bean。Spring Retry有一个RetryInterceptorBuilder，可以轻松创建一个。 |

查找远程配置资源

配置服务从/{name}/{profile}/{label}提供属性源，客户端应用程序中的默认绑定

* “name”= ${spring.application.name}
* “profile”= ${spring.profiles.active}（实际上是Environment.getActiveProfiles()）
* “label”=“master”

所有这些都可以通过设置spring.cloud.config.\*（其中\*是“name”，“profile”或“label”）来覆盖。“标签”可用于回滚到以前版本的配置; 使用默认的Config Server实现，它可以是git标签，分支名称或提交ID。标签也可以以逗号分隔的列表形式提供，在这种情况下，列表中的项目会逐个尝试，直到成功。例如，当您可能希望将配置标签与您的分支对齐，但使其成为可选（例如spring.cloud.config.label=myfeature,develop）时，这对于在特征分支上工作时可能很有用。

安全

如果您在服务器上使用HTTP基本安全性，那么客户端只需要知道密码（如果不是默认用户名）。您可以通过配置服务器URI，或通过单独的用户名和密码属性，例如

*bootstrap.yml*

spring:

cloud:

config:

uri: https://user:secret@myconfig.mycompany.com

要么

*bootstrap.yml*

spring:

cloud:

config:

uri: https://myconfig.mycompany.com

username: user

password: secret

spring.cloud.config.password和spring.cloud.config.username值覆盖URI中提供的任何内容。

如果您在Cloud Foundry部署应用程序，则提供密码的最佳方式是通过服务凭证（例如URI），因为它甚至不需要在配置文件中。在Cloud Foundry上为本地工作的用户提供的服务的一个例子，名为“configserver”：

*bootstrap.yml*

spring:

cloud:

config:

uri: ${vcap.services.configserver.credentials.uri:http://user:password@localhost:8888}

如果您使用另一种形式的安全性，则可能需要向ConfigServicePropertySourceLocator提供RestTemplate（例如，通过在引导上下文中获取它并注入一个）。ConfigServicePropertySourceLocator [提供{249](https://springcloud.cc/spring-cloud-config.html#custom-rest-template) /}（例如通过在引导上下文中获取它并注入）。

健康指标

配置客户端提供尝试从配置服务器加载配置的Spring Boot运行状况指示器。可以通过设置health.config.enabled=false来禁用运行状况指示器。由于性能原因，响应也被缓存。默认缓存生存时间为5分钟。要更改该值，请设置health.config.time-to-live属性（以毫秒为单位）。

提供自定义RestTemplate

在某些情况下，您可能需要从客户端自定义对配置服务器的请求。通常这涉及传递特殊的Authorization标头来对服务器的请求进行身份验证。要提供自定义RestTemplate，请按照以下步骤操作。

1. 设置spring.cloud.config.enabled=false以禁用现有的配置服务器属性源。
2. 使用PropertySourceLocator的实现创建一个新的配置bean。

*CustomConfigServiceBootstrapConfiguration.java*

@Configuration

public class CustomConfigServiceBootstrapConfiguration {

@Bean

public ConfigClientProperties configClientProperties() {

ConfigClientProperties client = new ConfigClientProperties(this.environment);

client.setEnabled(false);

return client;

}

@Bean

public ConfigServicePropertySourceLocator configServicePropertySourceLocator() {

ConfigClientProperties clientProperties = configClientProperties();

ConfigServicePropertySourceLocator configServicePropertySourceLocator = new ConfigServicePropertySourceLocator(clientProperties);

configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties));

return configServicePropertySourceLocator;

}

}

1. 在resources/META-INF中创建一个名为spring.factories的文件，并指定您的自定义配置。

*spring.factorties*

org.springframework.cloud.bootstrap.BootstrapConfiguration = com.my.config.client.CustomConfigServiceBootstrapConfiguration

Vault

当使用Vault作为配置服务器的后端时，客户端将需要为服务器提供令牌以从Vault中检索值。可以通过在bootstrap.yml中设置spring.cloud.config.token来在客户端中提供此令牌。

*bootstrap.yml*

spring:

cloud:

config:

token: YourVaultToken

Vault

嵌套密钥在Vault

Vault支持将键嵌入存储在Vault中的值。例如

echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' | vault write secret/myapp -

此命令会将JSON对象写入您的Vault。要在Spring中访问这些值，您将使用传统的点（。）注释。例如

@Value("${appA.secret}")

String name = "World";

上述代码将name变量设置为appAsecret。