13.1 基本用法

分布式配置中心解决方案：

国内：

- 360：QConf
- 淘宝：diamond
- 百度：disconf

国外：

- Apache Commons Configuration
- owner
- cfg4j

13.1.1 简介

Spring Cloud Config 是一个分布式系统配置管理的解决方案，它包含了 Client 和 Server 。配置文件放在 Server 端，通过 接口的形式提供给 Client。

Spring Cloud Config 主要功能：

- 集中管理各个环境、各个微服务的配置文件
- 提供服务端和客户端支持
- 配置文件修改后，可以快速生效
- 配置文件通过 Git/SVn 进行管理，天然支持版本回退功能。
- 支持高并发查询、也支持多种开发语言。

公众号【江南一点雨】，底部菜单，有一个教程合集 -> Git教程

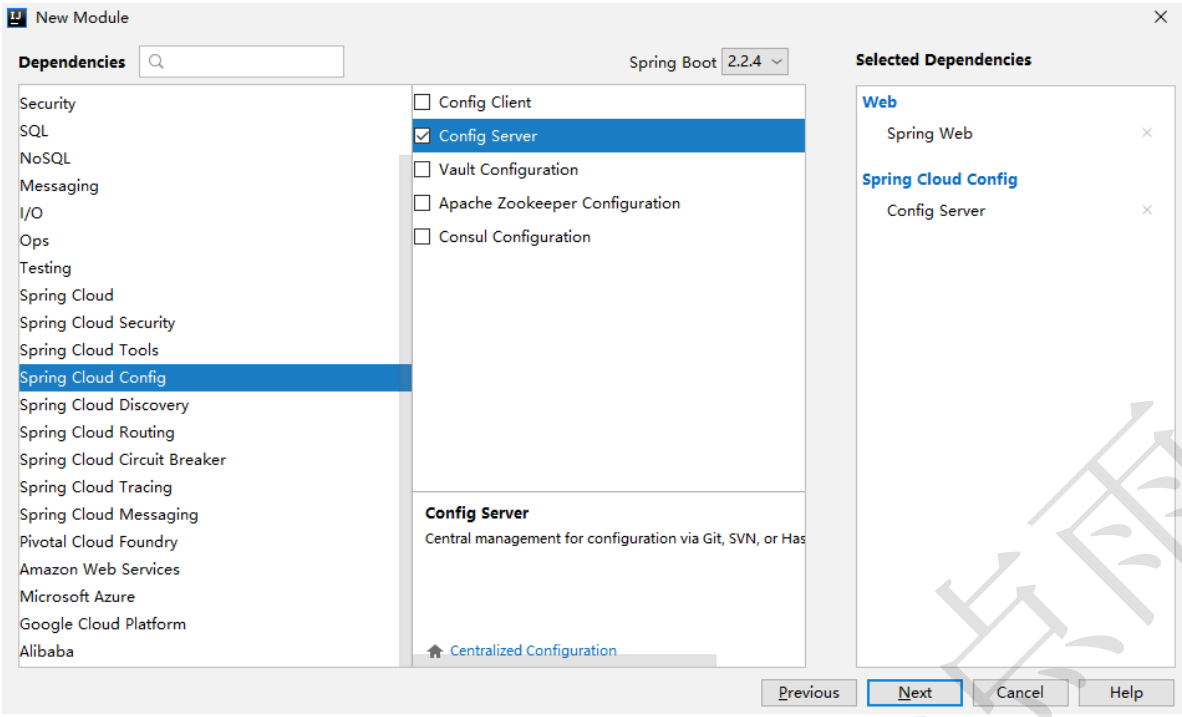
13.1.2 准备工作

准备工作主要是给 GitHub 上提交数据。

本地准备好相应的配置文件，提交到 GitHub：https://github.com/wongsung/configRepo

13.1.2 ConfigServer

首先创建一个 ConfigServer 工程，创建时添加 ConfigServer 依赖：



项目创建成功后，项目启动类上添加注解，开启 config server 功能：

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }

}
```

然后在配置文件中配置仓库的基本信息：

```
spring.application.name=config-server
server.port=8081
# 配置文件仓库地址
spring.cloud.config.server.git.uri=https://github.com/wongsung/configRepo.git
# 仓库中，配置文件的目录
spring.cloud.config.server.git.search-paths=client1
# 仓库的用户名密码
spring.cloud.config.server.git.username=1510161612@qq.com
spring.cloud.config.server.git.password=
```

启动项目后，就可以访问配置文件了。访问地址：http://localhost:8081/client1/prod/master

实际上，访问地址有如下规则：

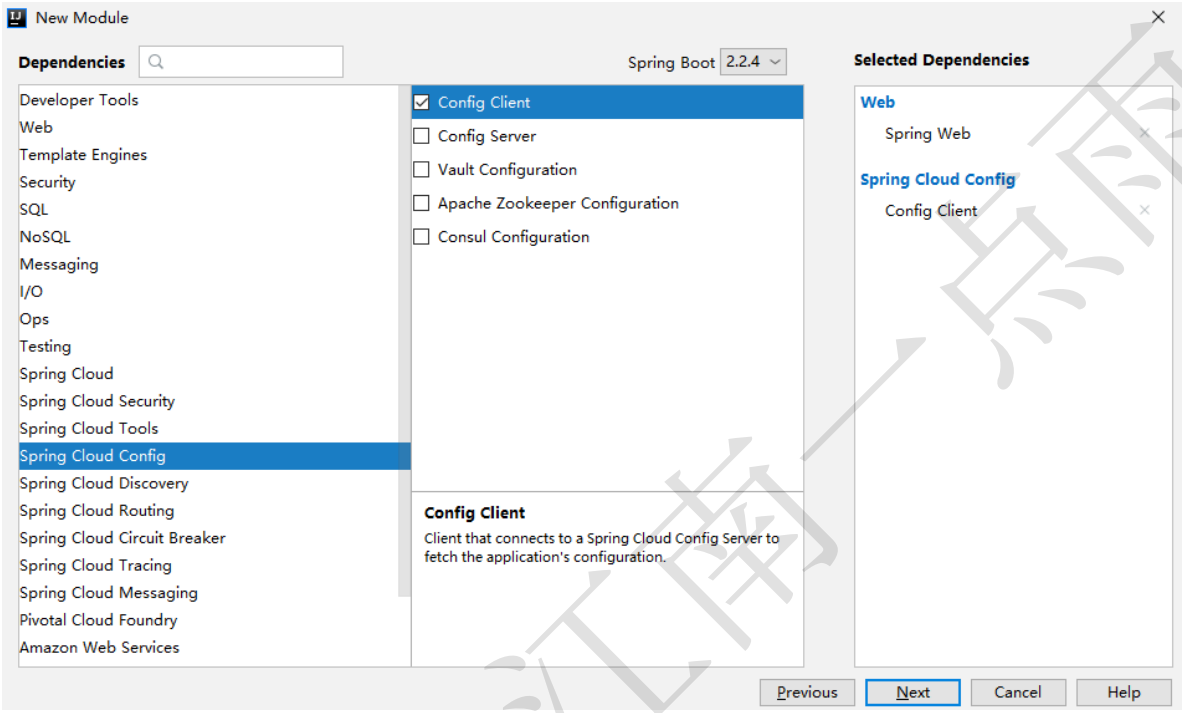
/{application}/{profile}/[{label}]
/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.yml
/{label}/{application}-{profile}.properties

applicaiton 表示配置文件名
profile 表示配置文件 profile，例如 test、dev、prod
label 表示git 分支，参数可选，默认就是master

接下来，可以修改配置文件，并且重新提交到 GitHub，此时，刷新 ConfigServer 接口，就可以及时看到最新的配置内容。

13.1.3 ConfigClient

创建一个 Spring Boot 项目，添加 ConfigClient 依赖：



项目创建成功后，resources 目录下，添加 bootstrap.properties 配置，内容如下：

```
# 下面三行配置，分别对应 config-server 中的 {application}、{profile}以及{label}占位符
spring.application.name=client1
spring.cloud.config.profile=dev
spring.cloud.config.label=master
spring.cloud.config.uri=http://localhost:8080
server.port=8082
```

接下来创建一个 HelloController 进行测试：

```
@RestController
public class HelloController {
    @Value("${javaboy}")
    String javaboy;
    @GetMapping("/hello")
    public String hello() {
        return javaboy;
    }
}
```

13.1.4 配置

使用占位符灵活控制查询目录。

修改 config-server 配置文件：

```
spring.cloud.config.server.git.search-paths={application}
```

这里的 {application} 占位符，表示链接上来的 client1 的 spring.application.name 属性的值。

在 confi-server 中，也可以用 {profile} 表示 client 的 spring.cloud.config.profile，也可以用 {label} 表示 client 的 spring.cloud.config.label

虽然在实际开发中，配置文件一般都是放在 Git 仓库中，但是，config-server 也支持将配置文件放在 classpath 下。

在 config-server 中添加如下配置：

```
# 表示让 config-server 从 classpath 下查找配置，而不是去 Git 仓库中查找
spring.profiles.active=native
```

也可以在 config-server 中，添加如下配置，表示指定配置文件的位置：

```
spring.cloud.config.server.native.search-locations=file:/E:/properties/
```

13.2 配置文件加解密

13.2.1 常见加密方案

- 不可逆加密
- 可逆加密

不可逆加密，就是理论上无法根据加密后的密文推算出明文。一般用在密码加密上，常见的算法如 MD5 消息摘要算法、SHA 安全散列算法。

可逆加密，看名字就知道可以根据加密后的密文推断出明文的加密方式，可逆加密一般又分为两种：

- 对称加密
- 非对称加密

对称加密指加密的密钥和解密的密钥是一样的。常见算法des、3des、aes

非对称加密就是加密的密钥和解密的密钥不一样，加密的叫做公钥，可以告诉任何人，解密的叫做私钥，只有自己知道。常见算法 RSA。

13.2.2 对称加密

首先下载不限长度的 JCE：http://download.oracle.com/otn-pub/java/jce/8/jce_policy-8.zip

将下载的文件解压，解压出来的 jar 拷贝到 Java 安装目录中：C:\Program Files\Java\jdk-13.0.1\lib\security

然后，在 config-server 的 bootstrap.properties 配置文件中，添加如下内容配置密钥：

```
# 密钥
encrypt.key=javaboy
```

然后，启动 config-server ，访问如下地址，查看密钥配置是否OK：http://localhost:8081/encrypt/status

然后，访问：http://localhost:8081/encrypt ，注意这是一个 POST 请求，访问该地址，可以对一段明文进行加密。把加密后的明文存储到 Git 仓库中，存储时，要注意加一个 {cipher} 前缀。

client1-dev.properties

```
javaboy={cipher}2ce7a4d481cba4dc24775b8e4520f1a617d78043f31328ec04b8e4b18288db55
```

13.2.3 非对称加密

非对称加密需要我们首先生成一个密钥对。

在命令行执行如下命令，生成 keystore：

```
keytool -genkeypair -alias config-server -keyalg RSA -keystore
D:\springcloud\config-server.keystore
```

命令执行完成后，拷贝生成的 keystore 文件到 config-server 的 resources 目录下。

然后在 config-server 的 bootstrap.properties 目录中，添加如下配置：

```
encrypt.key-store.location=config-server.keystore
encrypt.key-store.alias=config-server
encrypt.key-store.password=111111
encrypt.key-store.secret=111111
```

重启 config-server ，测试方法与对称加密一致。

注意，在 pom.xml 的 build 节点中，添加如下配置，防止 keystore 文件被过滤掉。

```
<resources>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.keystore</include>
        </includes>
    </resource>
</resources>
```

13.3 安全管理

防止用户直接通过访问 config-server 看到配置文件内容，我们可以用 spring security 来保护 config-server 接口。

首先在 config-server 中添加 spring security 依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

添加完依赖后，config-server 中的接口就自动被保护起来了。

默认自动生成的密码不好记，所以我们可以在 config-server 中，自己配置用户名密码。

在 config-server 的配置文件中，添加如下配置，固定用户名密码：

```
spring.security.user.name=javaboy
spring.security.user.password=123
```

然后，在 config-client 的 bootstrap.properties 配置文件中，添加如下配置：

```
spring.cloud.config.username=javaboy
spring.cloud.config.password=123
```

13.4 服务化

前面的配置都是直接在 config-client 中写死 config-server 的地址。

首先启动 Eureka 。

然后，为了让 config-server 和 config-client 都能注册到 Eureka ，给它俩添加如下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

然后，在 application.properties 配置文件中配置注册信息。

```
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

然后，修改 config-client 的配置文件，不再直接写死 config-server 的地址了。

```
# 下面三行配置，分别对应 config-server 中的 {application}、{profile}以及{label}占位符
spring.application.name=client1
spring.cloud.config.profile=dev
spring.cloud.config.label=master
#spring.cloud.config.uri=http://localhost:8081
# 开启通过 eureka 获取 config-server 的功能
spring.cloud.config.discovery.enabled=true
# 配置 config-server 服务名称
spring.cloud.config.discovery.service-id=config-server
server.port=8082

spring.cloud.config.username=javaboy
spring.cloud.config.password=123

eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

注意，加入 eureka client 之后，启动 config-server 可能会报错，此时，我们重新生成一个 jks 格式的密钥。

```
keytool -genkeypair -alias mytestkey -keyalg RSA -keypass 111111 -keystore
D:\springcloud\config-service.jks -storepass 111111
```

生成之后，拷贝到 configserver 的 resources 目录下，同时修改 bootstrap.properties 配置。

```
# 密钥
#encrypt.key=javaboy

encrypt.key-store.location=classpath:config-service.jks
encrypt.key-store.alias=mytestkey
encrypt.key-store.password=111111
encrypt.key-store.secret=111111

spring.security.user.name=javaboy
spring.security.user.password=123
```

同时也修改一个 pom.xml 中的过滤条件：

```
<resources>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.jks</include>
        </includes>
    </resource>
</resources>
```

13.5 动态刷新

当配置文件发生变化之后，config-server 可以及时感知到变化，但是 config-client 不会及时感知到变化，默认情况下，config-client 只有重启才能加载到最新的配置文件。

首先给 config-client 添加如下依赖：

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

然后，添加配置，使 refresh 端点暴露出来：

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

最后，再给 config-client 使用了配置文件的地方加上 @RefreshScope 注解，这样，当配置改变后，只需要调用 refresh 端点，config-client 中的配置就可以自动刷新。

```
@RestController
@RefreshScope
public class HelloController {
    @Value("${javaboy}")
    String javaboy;
    @GetMapping("/hello")
    public String hello() {
        return javaboy;
    }
}
```

重启 config-client，以后，只要配置文件发生变化，发送 POST 请求，调用 http://localhost:8082/actuator/refresh 接口即可，配置文件就会自动刷新。

13.6 请求失败重试

config-client 在调用 config-server 时，一样也可能发生请求失败的问题，这个时候，我们可以配置一个请求重试的功能。

要给 config-client 添加重试功能，只需要添加如下依赖即可：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
```

然后，修改配置文件，开启失败快速响应。

```
# 开启失败快速响应
spring.cloud.config.fail-fast=true
```

然后，注释掉配置文件的用户名密码，重启 config-client，此时加载配置文件失败，就会自动重试。

也可以通过如下配置保证服务的可用性：

```
# 开启失败快速响应
spring.cloud.config.fail-fast=true
# 请求重试的初识间隔时间
spring.cloud.config.retry.initial-interval=1000
# 最大重试次数
spring.cloud.config.retry.max-attempts=6
# 重试时间间隔乘数
spring.cloud.config.retry.multiplier=1.1
# 最大间隔时间
spring.cloud.config.retry.max-interval=2000
```