

3. 忽略所有微服务，只路由指定微服务。

很多场景下，可能只想要让 Zuul 代理指定的微服务，此时可以将 zuul.ignored-services 设为 '*'。

```
zuul:  
  ignored-services: '*'  # 使用 '*' 可忽略所有微服务  
  routes:  
    microservice-provider-user: /user/**
```

这样就可以让 Zuul 只路由 microservice-provider-user 微服务。

4. 同时指定微服务的 serviceId 和对应路径。例如：

```
zuul:  
  routes:  
    user-route:                      # 该配置方式中，user-route只是给路由一个名  
      service-id: provider-microservice-user  
      path: /user/**                  # service-id对应的路径
```

本例配置的效果同示例 1。

5. 同时指定 path 和 URL，例如：

```
zuul:  
  routes:  
    user-route:                      # 该配置方式中，user-route只是给路由一个名  
      url: http://localhost:8000/ # 指定的url  
      path: /user/**            # url对应的路径。
```

这样就可以将 /user/** 映射到 http://localhost:8000/**。

需要注意的是，使用这种方式配置的路由不会作为 HystrixCommand 执行，同时也不能使用 Ribbon 来负载均衡多个 URL。例 6 可解决该问题。

6. 同时指定 path 和 URL，并且不破坏 Zuul 的 Hystrix、Ribbon 特性。

```
zuul:  
  routes:  
    user-route:  
      path: /user/**  
      service-id: microservice-provider-user  
  ribbon:  
    eureka:
```

```

    enabled: false      # 为 Ribbon 禁用 Eureka
microservice-provider-user:
  ribbon:
    listOfServers: localhost:8000,localhost:8001

```

这样就可以既指定 path 与 URL，又不破坏 Zuul 的 Hystrix 与 Ribbon 特性了。

7. 使用正则表达式指定 Zuul 的路由匹配规则

借助 PatternServiceRouteMapper，实现从微服务到映射路由的正则配置。例如：

```

@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
    // 调用构造函数PatternServiceRouteMapper(String servicePattern, String
    routePattern)
    // servicePattern指定微服务的正则
    // routePattern指定路由正则
    return new PatternServiceRouteMapper("(?<name>^.+)-(?<version>v.+$)", "${
        version}/${name}");
}

```

通过这段代码即可实现将诸如 microservice-provider-user-v1 这个微服务，映射到/v1/microservice-provider-user/** 这个路径。

8. 路由前缀

示例 1：

```

zuul:
  prefix: /api
  strip-prefix: false
  routes:
    microservice-provider-user: /user/**

```

这样，访问 Zuul 的/api/microservice-provider-user/1 路径，请求将会被转发到 microservice-provider-user 的/api/1

示例 2：

```

zuul:
  routes:
    microservice-provider-user:
      path: /user/**
      strip-prefix: false

```

这样访问 Zuul 的/user/1 路径，请求将会被转发到 microservice-provider-user 的/user/1。



- 可参考该 Issue 辅助理解: <https://github.com/spring-cloud/spring-cloud-netflix/issues/1365>。
- 该特性可能有 Bug, 相关 Issue: <https://github.com/spring-cloud/spring-cloud-netflix/issues/1514>。

9. 忽略某些路径

上文讲解了如何忽略微服务，但有时还需要更细粒度的路由控制。例如，想让 Zuul 代理某个微服务，同时又想保护该微服务的某些敏感路径。此时，可使用 ignoredPatterns，指定忽略的正则。例如：

```
zuul:  
  ignoredPatterns: /**/admin/**  # 忽略所有包含/admin/的路径  
  routes:  
    microservice-provider-user: /user/**
```

这样就可将 microservice-provider-user 微服务映射到 /user/** 路径，但会忽略该微服务中所有包含 /admin/ 的路径。



读者如无法掌握 Zuul 路由的规律，可将 com.netflix 包的日志级别设为 DEBUG。这样，Zuul 就会打印转发的具体细节，从而有助于更好地理解 Zuul 的路由配置，例如：

```
logging:  
  level:  
    com.netflix: DEBUG
```

8.6 Zuul 的安全与 Header

本节来讨论 Zuul 的安全与 Header。

8.6.1 敏感 Header 的设置

一般来说，可在同一个系统中的服务之间共享 Header。不过应尽量防止让一些敏感的 Header 外泄。因此，在很多场景下，需要通过为路由指定一系列敏感 Header 列表。例如：

```
zuul:  
  routes:
```

```
microservice-provider-user:
  path: /users/**
  sensitive-headers: Cookie,Set-Cookie,Authorization
  url: https://downstream
```

这样就可为 microservice-provider-user 微服务指定敏感 Header 了。

也可用 zuul.sensitive-headers 全局指定敏感 Header，例如：

```
zuul:
  sensitive-headers: Cookie,Set-Cookie,Authorization # 默认是Cookie,Set-Cookie,
  Authorization
```

需要注意的是，如果使用 zuul.routes.*.sensitive-headers 的配置方式，会覆盖掉全局的配置。

8.6.2 忽略 Header

可使用 zuul.ignoredHeaders 属性丢弃一些 Header，例如：

```
zuul:
  ignored-headers: Header1, Header2
```

这样设置后，Header1 和 Header2 将不会传播到其他的微服务中。

默认情况下，zuul.ignored-headers 是空值，但如果 Spring Security 在项目的 classpath 中，那么 zuul.ignored-headers 的默认值就是 Pragma, Cache-Control, X-Frame-Options, X-Content-Type-Options, X-XSS-Protection, Expires。所以，当 Spring Security 在项目的 classpath 中，同时又需要使用下游微服务的 Spring Security 的 Header 时，可以将 zuul.ignoreSecurity-Headers 设置为 false。



- 笔者在 Github 上提的 Issue: <https://github.com/spring-cloud/spring-cloud-netflix/issues/1487>，希望可以帮助大家理解。
- sensitive-headers 与 ignored-headers 的单元测试：<https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/test/java/org/springframework/cloud/netflix/zuul/filters/pre/PreDecorationFilterTests.java>。
- 事实上，sensitive-headers 会被添加到 ignored-headers 中，详见代码 org.springframework.cloud.netflix.zuul.filters.ProxyRequestHelper.addIgnored-Headers(String...) 和 org.springframework.cloud.netflix.zuul.filters.pre.PreDecorationFilter.run()。

8.7 使用 Zuul 上传文件

本节来讨论 Zuul 的文件上传。

对于小文件（1M 以内）上传，无须任何处理，即可正常上传。对于大文件（10M 以上）上传，需要为上传路径添加/zuul 前缀。也可使用zuul.servlet-path 自定义前缀。

也就是说，假设zuul.routes.microservice-file-upload = /microservice-file-upload/**，如果http://{HOST}:{PORT}/upload 是微服务 microservice-file-upload 的上传路径，则可使用 Zuul 的/zuul/microservice-file-upload/upload 路径上传大文件。

如果 Zuul 使用了 Ribbon 做负载均衡，那么对于超大的文件（例如 500M），需要提升超时设置，例如：

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000
```

下面编写一个文件上传的微服务，详细讲解 Zuul 的文件上传。

编写文件上传微服务

1. 创建一个 Maven 工程，ArtifactId 是microservice-file-upload，并为项目添加以下依赖。

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

2. 在启动类上添加@SpringBootApplication、@EnableEurekaClient注解。
3. 编写 Controller。

```

@Controller
public class FileUploadController {
    /**
     * 上传文件
     * 测试方法：
     * 有界面的测试：http://localhost:8050/index.html
     * 使用命令：curl -F "file=@文件全名" localhost:8050/upload
     * ps.该示例比较简单，没有做IO异常、文件大小、文件非空等处理
     * @param file 待上传的文件
     * @return 文件在服务器上的绝对路径
     * @throws IOException IO异常
     */
    @RequestMapping(value = "/upload", method = RequestMethod.POST)
    public @ResponseBody String handleFileUpload(@RequestParam(value = "file",
        required = true) MultipartFile file) throws IOException {
        byte[] bytes = file.getBytes();
        File fileToSave = new File(file.getOriginalFilename());
        FileCopyUtils.copy(bytes, fileToSave);
        return fileToSave.getAbsolutePath();
    }
}

```

- 配置文件 application.yml，在其中添加如下内容。

```

server:
  port: 8050
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
instance:
  prefer-ip-address: true
spring:
  application:
    name: microservice-file-upload
http:
  multipart:
    max-file-size: 2000Mb      # Max file size, 默认1M
    max-request-size: 2500Mb   # Max request size, 默认10M

```

由配置可知，已经将该服务注册到 Eureka Server 上，并配置了文件上传大小的限制。

这样一个文件上传的微服务就编写完成了。可使用以下命令测试文件上传。

```
curl -F "file=@文件全名" localhost:8050/upload
```



测试

笔者使用的测试工具是 cURL，大家也可使用其他工具进行测试。

1. 准备 1 个小文件（1M 以下），记为 small.file；1 个超大文件（1G 以上，2G 以上），记为 large.file。
2. 启动 microservice-discovery-eureka。
3. 启动 microservice-file-upload。
4. 启动 microservice-gateway-zuul。
5. 测试直接上传到 microservice-file-upload 上：使用命令 curl -F "file=@large.file" localhost:8050/upload，上传大文件，发现可以正常上传。同理，小文件也可以上传。
6. 测试通过 Zuul 上传小文件。

```
curl -v -H "Transfer-Encoding: chunked" -F "file=@small.file" localhost:8040/microservice-file-upload/upload
```

发现可以正常上传。

7. 测试通过 Zuul 上传大文件，不添加/zuul 前缀。

```
curl -v -H "Transfer-Encoding: chunked" -F "file=@large.file" localhost:8040/microservice-file-upload/upload
```

发现 Zuul 会报类似以下异常：

```
org.apache.tomcat.util.http.fileupload.FileUploadBase$FileSizeLimitExceededException: The field file exceeds its maximum permitted size of 1048576 bytes.
```

8. 测试通过 Zuul 上传大文件，添加/zuul 前缀。

```
curl -v -H "Transfer-Encoding: chunked" -F "file=@large.file" localhost:8040/zuul/microservice-file-upload/upload
```

此时，Zuul 会报以下异常：

```
Caused by: com.netflix.hystrix.exception.HystrixRuntimeException: microservice-file-upload timed-out and no fallback available.
```

可以看到，此时已经不是文件大小限制的异常了，而只是 Hystrix 的超时异常。

9. 因此，不妨在 application.yml 中添加如下内容。

```
# 上传大文件得将超时时间设置长一些，否则会报超时异常。以下几行超时设
    置来自http://cloud.spring.io/spring-cloud-static/Camden.SR4/#_
    _uploading_files_through_zuul
    hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:
        60000
    ribbon:
        ConnectTimeout: 3000
        ReadTimeout: 60000
```

详见本书配套代码中的 microservice-gateway-zuul-file-upload 项目。

10. 关闭microservice-gateway-zuul，启动microservice-gateway-zuul-file-upload再次使用。

```
curl -v -H "Transfer-Encoding: chunked" -F "file=@large.file" localhost:
    8040/zuul/microservice-file-upload/upload
```

此时已可正常上传文件。

8.8 Zuul 的过滤器

过滤器是 Zuul 的核心组件，本节来详细讨论 Zuul 的过滤器。

8.8.1 过滤器类型与请求生命周期

Zuul 大部分功能都是通过过滤器来实现的。Zuul 中定义了 4 种标准过滤器类型，这些过滤器类型对应于请求的典型生命周期。

- PRE：这种过滤器在请求被路由之前调用。可利用这种过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等。
- ROUTING：这种过滤器将请求路由到微服务。这种过滤器用于构建发送给微服务的请求，并使用 Apache HttpClient 或 Netflix Ribbon 请求微服务。
- POST：这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的 HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。
- ERROR：在其他阶段发生错误时执行该过滤器。

除了默认的过滤器类型，Zuul 还允许创建自定义的过滤器类型。例如，可以定制一种 STATIC 类型的过滤器，直接在 Zuul 中生成响应，而不将请求转发到后端的微服务。

Zuul 请求的生命周期如图 8-5 所示，该图详细描述了各种类型的过滤器的执行顺序。

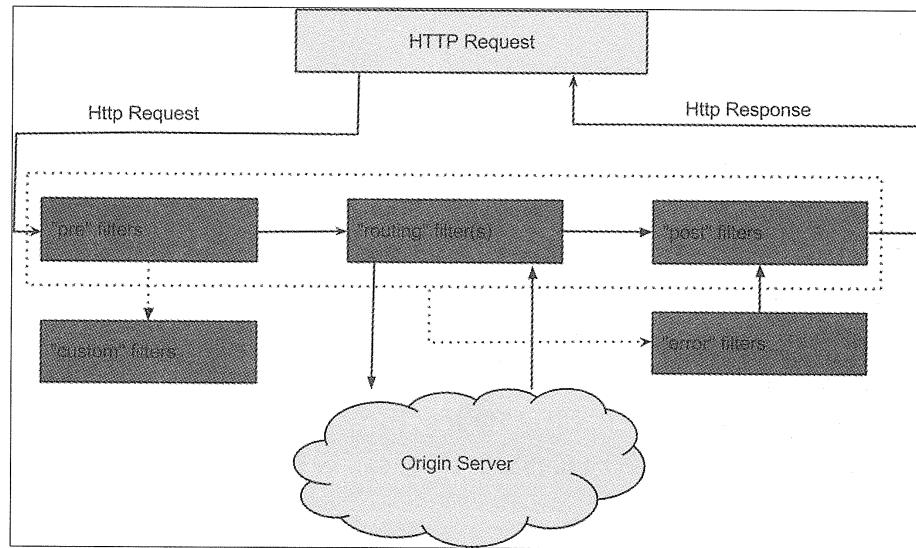


图 8-5 Zuul 请求的生命周期

8.8.2 编写 Zuul 过滤器

理解过滤器类型和请求生命周期后，来编写一个 Zuul 过滤器。编写 Zuul 的过滤器非常简单，只须继承抽象类 ZuulFilter，然后实现几个抽象方法就可以了。

那么现在来编写一个简单的 Zuul 过滤器，让该过滤器打印请求日志。

1. 复制项目 microservice-gateway-zuul，将 ArtifactId 修改为 microservice-gateway-zuul-filter。
2. 编写自定义 Zuul 过滤器。

```
public class PreRequestLogFilter extends ZuulFilter {  
    private static final Logger LOGGER = LoggerFactory.getLogger(  
        PreRequestLogFilter.class);  
  
    @Override  
    public String filterType() {  
        return "pre";  
    }
```

```
@Override  
public int filterOrder() {  
    return 1;  
}  
  
@Override  
public boolean shouldFilter() {  
    return true;  
}  
  
@Override  
public Object run() {  
    RequestContext ctx = RequestContext.getCurrentContext();  
    HttpServletRequest request = ctx.getRequest();  
    PreRequestLogFilter.LOGGER.info(String.format("send %s request to %s",  
        request.getMethod(), request.getRequestURL().toString()));  
    return null;  
}  
}
```

由代码可知，自定义的 Zuul Filter 需实现以下几个方法。

- `filterType`: 返回过滤器的类型。有 `pre`、`route`、`post`、`error` 等几种取值，分别对应上文的几种过滤器。详细可以参考 `com.netflix.zuul.ZuulFilter.filterType()` 中的注释。
- `filterOrder`: 返回一个 `int` 值来指定过滤器的执行顺序，不同的过滤器允许返回相同的数字。
- `shouldFilter`: 返回一个 `boolean` 值来判断该过滤器是否要执行，`true` 表示执行，`false` 表示不执行。
- `run`: 过滤器的具体逻辑。本例中让它打印了请求的 HTTP 方法以及请求的地址。

3. 修改启动类，为启动类添加以下内容：

```
@Bean  
public PreRequestLogFilter preRequestLogFilter() {  
    return new PreRequestLogFilter();  
}
```



测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-gateway-zuul-filter。
4. 访问 `http://localhost:8040/microservice-provider-user/1`，可获得类似如下的日志。

```
[nio-8040-exec-6] c.i.c.s.filters.pre.PreRequestLogFilter : send GET  
request to http://localhost:8040//microservice-provider-user/1
```

说明编写的自定义 Zuul 过滤器被执行了。

8.8.3 禁用 Zuul 过滤器

Spring Cloud 默认为 Zuul 编写并启用了一些过滤器，例如 DebugFilter、FormBodyWrapperFilter、PreDecorationFilter 等。这些过滤器都存放在 spring-cloud-netflix-core 这个 Jar 包的 org.springframework.cloud.netflix.zuul.filters 包中。

一些场景下，想要禁用掉部分过滤器，该怎么办呢？

答案非常简单，只须设置 `zuul.<SimpleClassName>.<filterType>.disable=true`，即可禁用 SimpleClassName 所对应的过滤器。以过滤器 `org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter` 为例，只须设置 `zuul.SendResponseFilter.post.disable=true`，即可禁用该过滤器。

同理，如果想要禁用 8.8.2 节编写的过滤器，只需设置 `zuul.PreRequestLogFilter.pre.disable=true` 即可。



相关代码：`com.netflix.zuul.ZuulFilter.disablePropertyName()`、`com.netflix.zuul.ZuulFilter.isFilterDisabled()`、`com.netflix.zuul.ZuulFilter.runFilter()`。

8.9 Zuul 的容错与回退

在 Spring Cloud 中，Zuul 默认已经整合了 Hystrix。首先来做一个简单的实验：

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。

3. 启动项目 microservice-gateway-zuul。
4. 启动项目 microservice-hystrix-dashboard。
5. 访问 <http://localhost:8040/microservice-provider-user/1>，可正常获得结果。
6. 访问 <http://localhost:8040/hystrix.stream>，可获得 Hystrix 的监控数据。
7. 访问 <http://localhost:8030/hystrix>，并在监控地址一栏填入 <http://localhost:8040/hystrix.stream>，即可看到类似图 8-6 的图表。

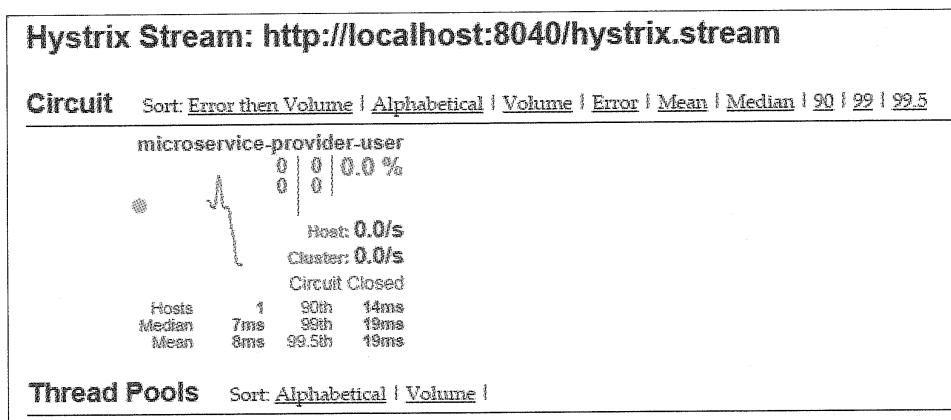


图 8-6 Hystrix Dashboard 监控页面

由图可知，Zuul 的 Hystrix 监控的粒度是微服务，而不是某个 API；同时也说明，所有经过 Zuul 的请求，都会被 Hystrix 保护起来。

8. 关闭项目 microservice-provider-user，再次访问 <http://localhost:8040/microservice-provider-user/1>，将会看到页面输出类似于如下的异常：

Whitelabel Error Page

```
This application has no explicit mapping for /error, so you are seeing this as
a fallback.
```

Thu Nov 24 15:28:20 CST 2016

There was an unexpected error (type=Internal Server Error, status=500).

TIMEOUT

下面来谈谈如何为 Zuul 实现回退。

为 Zuul 添加回退

想要为 Zuul 添加回退，需要实现 ZuulFallbackProvider 接口。在实现类中，指定为哪个微服务提供回退，并提供一个 ClientHttpResponse 作为回退响应。

下面来编写代码。

1. 复制项目microservice-gateway-zuul，将ArtifactId修改为microservice-gateway-zuul-fallback。
2. 编写Zuul的回退类：

```
@Component
public class UserFallbackProvider implements ZuulFallbackProvider {
    @Override
    public String getRoute() {
        // 表明是为哪个微服务提供回退
        return "microservice-provider-user";
    }

    @Override
    public ClientHttpResponse fallbackResponse() {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                // fallback时的状态码
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                // 数字类型的状态码，本例返回的其实就是200，详见HttpStatus
                return this.getStatusCode().value();
            }

            @Override
            public String getStatusText() throws IOException {
                // 状态文本，本例返回的其实就是OK，详见HttpStatus
                return this.getStatusCode().getReasonPhrase();
            }

            @Override
            public void close() {
            }
        };
    }
}
```

```
public InputStream getBody() throws IOException {
    // 响应体
    return new ByteArrayInputStream("用户微服务不可用，请稍后再试。",
        getBytes());
}

@Override
public HttpHeaders getHeaders() {
    // headers设定
    HttpHeaders headers = new HttpHeaders();
    MediaType mt = new MediaType("application", "json",
        Charset.forName("UTF-8"));
    headers.setContentType(mt);
    return headers;
}
};

}
```

添加回退之后，重复之前的实验，当 microservice-provider-user 微服务无法正常响应时，将会返回以下内容。

用户微服务不可用，请稍后再试。

8.10 Zuul 的高可用

Zuul 的高可用非常关键，因为外部请求到后端微服务的流量都会经过 Zuul。故而在生产环境中一般都需要部署高可用的 Zuul 以避免单点故障。

笔者分两种场景讨论 Zuul 的高可用。

8.10.1 Zuul 客户端也注册到了 Eureka Server 上

这种情况下，Zuul 的高可用非常简单，只须将多个 Zuul 节点注册到 Eureka Server 上，就可实现 Zuul 的高可用。此时，Zuul 的高可用与其他微服务的高可用没什么区别。

如图 8-7，当 Zuul 客户端也注册到 Eureka Server 上时，只须部署多个 Zuul 节点即可实现其高可用。Zuul 客户端会自动从 Eureka Server 中查询 Zuul Server 的列表，并使用 Ribbon 负载均衡地请求 Zuul 集群。

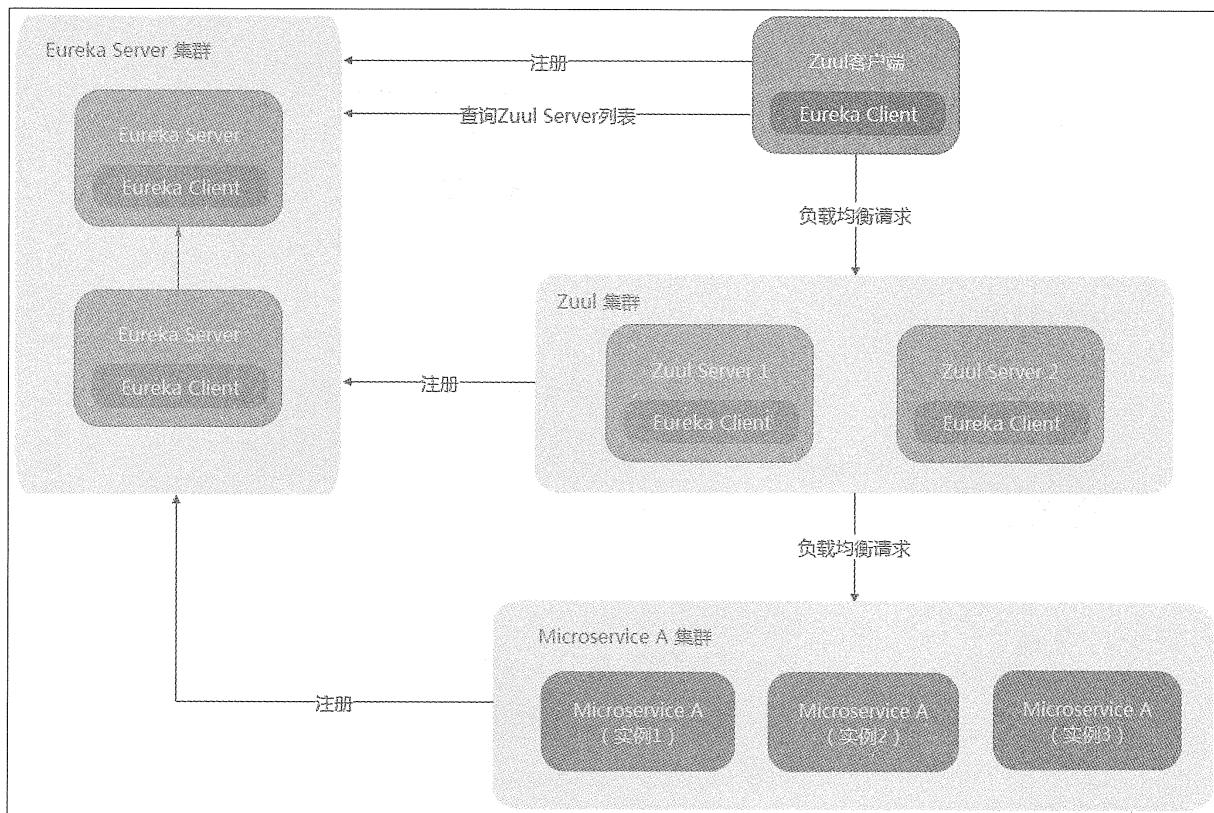


图 8-7 Zuul 高可用架构图

8.10.2 Zuul 客户端未注册到 Eureka Server 上

现实中，这种场景往往更常见，例如，Zuul 客户端是一个手机 APP——不可能让所有的手机终端都注册到 Eureka Server 上。这种情况下，可借助一个额外的负载均衡器来实现 Zuul 的高可用，例如 Nginx、HAProxy、F5 等。

如图 8-8，Zuul 客户端将请求发送到负载均衡器，负载均衡器将请求转发到其代理的其中一个 Zuul 节点。这样，就可以实现 Zuul 的高可用。

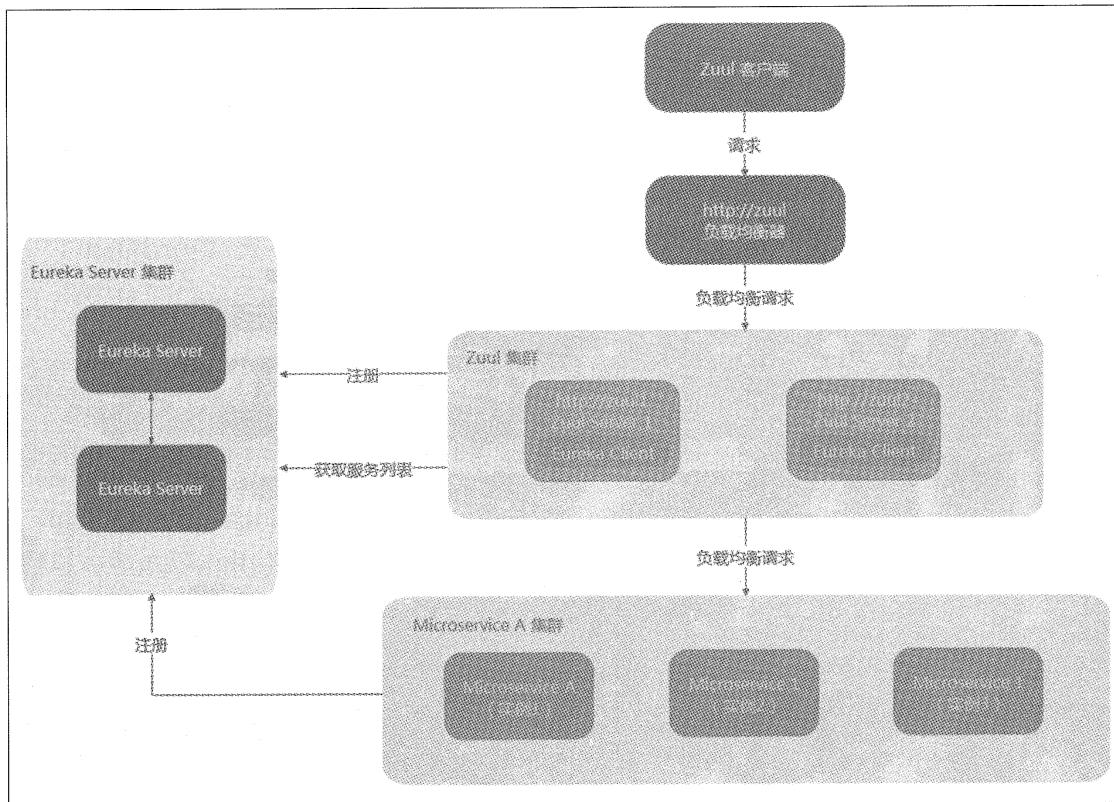


图 8-8 Zuul 高可用架构图

8.11 使用 Sidecar 整合非 JVM 微服务

在 4.9 节，笔者曾经提到，非 JVM 微服务可操作 Eureka 的 REST 端点，从而实现注册与发现。事实上，也可使用 Sidecar（挎斗）更加方便地整合非 JVM 微服务。

Spring Cloud Netflix Sidecar 的灵感来自 Netflix Prana，它包括了一个简单的 HTTP API 来获取指定服务所有实例的信息（例如主机和端口）。不仅如此，还可通过内嵌的 Zuul 来代理服务调用，该代理从 Eureka Server 中获取信息。非 JVM 微服务需要实现健康检查，以便 Sidecar 将它的状态上报给 Eureka Server，健康检查的形式如下：

```
{
  "status": "UP"
}
```

其中，`status` 用于描述微服务的状态，常见取值有 `UP`、`DOWN`、`OUT_OF_SERVICE` 以及 `UNKNOWN` 等。

下面来详细讨论如何使用 Sidecar 整合非 JVM 微服务。

8.11.1 编写 Node.js 微服务

先来编写一个非 JVM 微服务，以 Node.js 为例进行演示。

代码如下，记为 node-service。

```
var http = require('http');
var url = require("url");
var path = require('path');

// 创建server
var server = http.createServer(function(req, res) {
    // 获得请求的路径
    var pathname = url.parse(req.url).pathname;
    res.writeHead(200, { 'Content-Type' : 'application/json; charset=utf-8' });
    // 访问http://localhost:8060/，将会返回{"index":"欢迎来到首页"}
    if (pathname === '/') {
        res.end(JSON.stringify({ "index" : "欢迎来到首页" }));
    }
    // 访问http://localhost:8060/health，将会返回{"status":"UP"}
    else if (pathname === '/health.json') {
        res.end(JSON.stringify({ "status" : "UP" }));
    }
    // 其他情况返回404
    else {
        res.end("404");
    }
});
// 创建监听，并打印日志
server.listen(8060, function() {
    console.log('listening on localhost:8060');
});
```

这是一个非常简单的 Node.js 服务。笔者已在代码注释中详细描述，不再赘述。



测试

1. 使用node node-service.js 命令即可启动该服务。
2. 访问 http://localhost:8060/health.json，将会返回如下内容。

```
{
    "status": "UP"
}
```

3. 访问 `http://localhost:8060/`, 将会返回如下内容。

```
{
    "index": "欢迎来到首页"
}
```

4. 访问其他端点将会返回 404。

8.11.2 编写 Sidecar

本节来编写 Sidecar 微服务，并使用 Sidecar 整合非 JVM 微服务。

1. 创建一个 Maven 工程，ArtifactId 是 `microservice-sidecar`，并为项目添加以下依赖。

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

2. 在启动类上添加 `@EnableSidecar` 注解，声明这是一个 Sidecar。

```
@SpringBootApplication
@EnableSidecar
public class SidecarApplication {
    public static void main(String[] args) {
        SpringApplication.run(SidecarApplication.class, args);
    }
}
```

`@EnableSidecar` 是一个组合注解，它整合了三个注解，分别是：`@EnableCircuitBreaker`、`@EnableDiscoveryClient` 和 `@EnableZuulProxy`。

3. 在项目的配置文件 application.yml 中添加如下内容：

```
server:
  port: 8070
spring:
  application:
    name: microservice-sidecar-node-service
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
sidecar:
  port: 8060          # Node.js微服务的端口
  health-uri: http://localhost:8060/health.json # Node.js微服务的健康检查URL
```

由配置可知，现在已经把 Sidecar 注册到了 Eureka Server 上，并用 sidecar.port 属性指定了非 JVM 微服务所监听的端口，用 sidecar.health-uri 属性指定了非 JVM 微服务的健康检查 URL。

这样一个 Sidecar 就编写完成了。



测试 1：非 JVM 微服务访问 JVM 微服务

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 node-service。
4. 启动 microservice-sidecar。
5. 访问 <http://localhost:8070/microservice-provider-user/1>，可获得如下结果。

```
{
  "id": 1,
  "username": "account1",
  "name": "张三",
  "age": 20,
  "balance": 100
}
```

因此，非 JVM 微服务可通过这种方式调用注册在 Eureka Server 上的 JVM 微服务。



测试 2：JVM 微服务调用非 JVM 微服务的接口

使用 Sidecar 微服务的 serviceId，即可调用非 JVM 微服务的接口。以下是一个使用 Ribbon 请求 node-service 的示例：

```
@GetMapping("/test")
public String findById() {
    // 将会请求到: http://localhost:8060/，返回结果: {"index":"欢迎来到首页"}
    return this.restTemplate.getForObject("http://microservice-sidecar-node-
        service/", String.class);
}
```

详见本书配套代码中的 microservice-sidecar-client-ribbon 项目。

8.11.3 Sidecar 的端点

Sidecar 提供了一些端点，这些端点有助于管理 Sidecar。

1. /

该端点返回一个测试页面，该页面展示 Sidecar 的常用端点。

2. /hosts/{serviceId}

该端点返回 DiscoveryClient.getInstances(serviceId)，即指定微服务在 Eureka 上的实例列表。

3. /ping

该端点返回“OK”字符串。

4. /{serviceId}

由于 Sidecar 整合了 Zuul，因此可使用该端点，将请求转发到 serviceId 所对应的微服务。



相关代码可详见：org.springframework.cloud.netflix.sidecar.SidecarController。

8.11.4 Sidecar 与 Node.js 微服务分离部署

前文是将 Sidecar 与非 JVM 微服务部署在同一台主机上。现实中，常常会将 Sidecar 与 JVM 微服务分离部署，例如部署在不同的主机或者容器中。此时应该如何配置呢？

对这个问题，笔者做了一点总结。

方法一：

```
eureka:  
  instance:  
    hostname: 非JVM微服务的hostname
```

方法二：

对于 Spring Cloud Netflix 1.3.0 及以后的版本，除方法一外，还可通过以下属性实现分离部署。

```
sidecar:  
  hostname: 非JVM微服务的hostname  
  ip-address: 非JVM微服务的IP地址
```



读者可参考该 Issue 辅助理解：<https://github.com/spring-cloud/spring-cloud-netflix/issues/981>。

8.11.5 Sidecar 原理分析

本节来探讨 Sidecar 的原理。

1. 访问 Eureka Server 的路径：<http://localhost:8761/eureka/apps/microservice-sidecar-node-service>，可获得类似于如下的结果。

```
<application>  
  <name>MICROSERVICE-SIDECAR-NODE-SERVICE</name>  
  <instance>  
    <instanceId>itmuch:microservice-sidecar-node-service:8070</instanceId>  
    <hostName>192.168.0.59</hostName>  
    <app>MICROSERVICE-SIDECAR-NODE-SERVICE</app>  
    <ipAddr>192.168.0.59</ipAddr>  
    <status>UP</status>  
    <overriddenstatus>UNKNOWN</overriddenstatus>  
    <port enabled="true">8060</port>  
    <securePort enabled="false">443</securePort>  
    <countryId>1</countryId>  
    <dataCenterInfo class="com.netflix.appinfo.InstanceInfo$  
      DefaultDataCenterInfo">
```

```

<name>MyOwn</name>
</dataCenterInfo>
<leaseInfo>
    <renewalIntervalInSecs>30</renewalIntervalInSecs>
    <durationInSecs>90</durationInSecs>
    <registrationTimestamp>1481963830415</registrationTimestamp>
    <lastRenewalTimestamp>1481963830415</lastRenewalTimestamp>
    <evictionTimestamp>0</evictionTimestamp>
    <serviceUpTimestamp>1481963829086</serviceUpTimestamp>
</leaseInfo>
<metadata class="java.util.Collections$EmptyMap"/>
<homePageUrl>http://itmuch:8060/</homePageUrl>
<statusPageUrl>http://itmuch:8070/info</statusPageUrl>
<healthCheckUrl>http://itmuch:8070/health</healthCheckUrl>
<vipAddress>microservice-sidecar-node-service</vipAddress>
<secureVipAddress>microservice-sidecar-node-service</secureVipAddress>
<isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
<lastUpdatedTimestamp>1481963830415</lastUpdatedTimestamp>
<lastDirtyTimestamp>1481963825748</lastDirtyTimestamp>
<actionType>ADDED</actionType>
</instance>
</application>

```

由结果可知，注册到 Eureka Server 上的端口是 8060，`homePageUrl` 是 `http://itmuch:8060/`，恰恰是 node-service 的端口和首页。因此，注册到 Eureka Server 上的微服务可使用 `microservice-sidecar-node-service` 这个名称请求 node-service 的接口。

- 由于`@EnableSidecar`整合了注解`@EnableZuulProxy`，可尝试访问 Sidecar 的`/routes` 端点：`http://localhost:8070/routes`，获得类似如下的结果。

```
{
  /microservice-provider-user/**: "microservice-provider-user"
}
```

因此，非 JVM 微服务可通过 Sidecar 请求其他注册在 Eureka Server 的微服务。

- 可尝试将 node-service 多次启停，并观察 Sidecar 的`/health` 端点。Sidecar 会获取 node-service 的健康状态，并将该状态传播到 Eureka Server。使用这种方式，Eureka Server 就能感知到非 JVM 微服务的健康状态。



相关代码如下：

- org.springframework.cloud.netflix.sidecar.
LocalApplicationHealthCheckHandler.getStatus(InstanceStatus)
- org.springframework.cloud.netflix.sidecar.
LocalApplicationHealthIndicator.doHealthCheck(Builder)

8.12 使用 Zuul 聚合微服务

许多场景下，外部请求需要查询 Zuul 后端的多个微服务。举个例子，一个电影售票手机 APP，在购票订单页上，既需要查询“电影微服务”获得电影相关信息，又需要查询“用户微服务”获得当前用户的信息。如果让手机端直接请求各个微服务（即使使用 Zuul 进行转发），那么网络开销、流量耗费、耗费时长可能都无法令我们满意。那么对于这种场景，可使用 Zuul 聚合微服务请求——手机 APP 只需发送一个请求给 Zuul，由 Zuul 请求用户微服务以及电影微服务，并组织好数据给手机 APP。

使用这种方式，手机端只须发送一次请求即可，简化了客户端侧的开发；不仅如此，由于 Zuul、用户微服务、电影微服务一般都在同一个局域网中，因此速度会非常快，效率会非常高。

下面围绕以上场景，来编写代码示例。在本例中，使用 RxJava 结合 Zuul 来实现微服务请求的聚合。

1. 复制项目 microservice-gateway-zuul，将 ArtifactId 修改为 microservice-gateway-zuul-aggregation。
2. 修改启动类 ZuulApplication：

```
@SpringBootApplication  
@EnableZuulProxy  
public class ZuulApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(ZuulApplication.class, args);  
    }  
  
    @Bean  
    @LoadBalanced  
    public RestTemplate restTemplate() {
```

```
    return new RestTemplate();
}
```

3. 创建实体类：

```
public class User {  
    private Long id;  
    private String username;  
    private String name;  
    private Integer age;  
    private BigDecimal balance;  
    // getters and setters ...  
}
```

4. 创建 Java 类，名为 AggregationService：

```
@Service
public class AggregationService {
    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "fallback")
    public Observable<User> getUserById(Long id) {
        // 创建一个被观察者
        return Observable.create(observer -> {
            // 请求用户微服务的/{id}端点
            User user = restTemplate.getForObject("http://microservice-provider-user/{id}", User.class, id);
            observer.onNext(user);
            observer.onCompleted();
        });
    }

    @HystrixCommand(fallbackMethod = "fallback")
    public Observable<User> getMovieUserByUserId(Long id) {
        return Observable.create(observer -> {
            // 请求电影微服务的/user/{id}端点
            User movieUser = restTemplate.getForObject("http://microservice-consumer-movie/user/{id}", User.class, id);
            observer.onNext(movieUser);
            observer.onCompleted();
        });
    }
}
```

```
    });
}

public User fallback(Long id) {
    User user = new User();
    user.setId(-1L);
    return user;
}
}
```

5. 创建 Controller，在 Controller 中聚合多个请求。

```
@RestController
public class AggregationController {
    public static final Logger LOGGER = LoggerFactory.getLogger(ZuulApplication.class);

    @Autowired
    private AggregationService aggregationService;

    @GetMapping("/aggregate/{id}")
    public DeferredResult<HashMap<String, User>> aggregate(@PathVariable Long id)
    {
        Observable<HashMap<String, User>> result = this.aggregateObservable(id);
        return this.toDeferredResult(result);
    }

    public Observable<HashMap<String, User>> aggregateObservable(Long id) {
        // 合并两个或者多个Observables发射出的数据项，根据指定的函数变换它们
        return Observable.zip(
            this.aggregationService.getUserById(id),
            this.aggregationService.getMovieUserByUserId(id),
            (user, movieUser) -> {
                HashMap<String, User> map = Maps.newHashMap();
                map.put("user", user);
                map.put("movieUser", movieUser);
                return map;
            }
        );
    }
}
```

```

public DeferredResult<HashMap<String, User>> toDeferredResult(Observable<
    HashMap<String, User>> details) {
    DeferredResult<HashMap<String, User>> result = new DeferredResult<>();
    // 订阅
    details.subscribe(new Observer<HashMap<String, User>>() {
        @Override
        public void onCompleted() {
            LOGGER.info("完成...");
        }

        @Override
        public void onError(Throwable throwable) {
            LOGGER.error("发生错误...", throwable);
        }

        @Override
        public void onNext(HashMap<String, User> movieDetails) {
            result.setResult(movieDetails);
        }
    });
    return result;
}
}

```

这样，代码就编写完成了。相信熟悉 RxJava 的读者朋友们能非常轻松地阅读本段代码的含义，对于不了解的 RxJava 的读者朋友们，建议花一点时间入门 RxJava。



测试一：微服务聚合测试

1. 启动项目 microservice-discovery-eureka；
2. 启动项目 microservice-provider-user；
3. 启动项目 microservice-consumer-movie；
4. 启动项目 microservice-gateway-zuul-aggregation；
5. 访问<http://localhost:8040/aggregate/1>，可获得如下结果：

```
{
  "movieUser": {
    "id": 1,
}
```

```
"username": "account1",
"name": "张三",
"age": 20,
"balance": 100.00
},
"user": {
"id": 1,
"username": "account1",
"name": "张三",
"age": 20,
"balance": 100.00
}
}
```

说明已成功用 Zuul 聚合了用户微服务以及电影微服务的 RESTful API。



测试二：Hystrix 容错测试

1. 在测试一的基础上，停止项目 microservice-provider-user 以及 microservice-consumer-movie；
2. 访问 <http://localhost:8040/aggregate/1>，可获得如下结果：

```
{
  "movieUser": {
    "id": -1,
    "username": null,
    "name": null,
    "age": null,
    "balance": null
  },
  "user": {
    "id": -1,
    "username": null,
    "name": null,
    "age": null,
    "balance": null
  }
}
```

说明 fallback 方法正常被触发，能够正常回退。



使用 Spring Cloud Config 统一管理微服务配置

9.1 为什么要统一管理微服务配置

对于传统的单体应用，常使用配置文件管理所有配置。例如一个 Spring Boot 开发的单体应用，可将配置内容放在 application.yml 文件中。如果需要切换环境，可设置多个 Profile，并在启动应用时指定 spring.profiles.active={profile}。在本书 4.6 节中使用的也是这种方式。当然也可借助 Maven 的 Profile 实现环境切换。

然而，在微服务架构中，微服务的配置管理一般有以下需求：

- 集中管理配置。一个使用微服务架构的应用系统可能会包含成百上千个微服务，因此集中管理配置是非常有必要的。
- 不同环境不同配置。例如，数据资源配置在不同的环境（开发、测试、预发布、生产等）中是不同的。
- 运行期间可动态调整。例如，可根据各个微服务的负载情况，动态调整数据源连接池大小或熔断阈值，并且在调整配置时不停止微服务。

- 配置修改后可自动更新。如配置内容发生变化，微服务能够自动更新配置。

综上所述，对于微服务架构而言，一个通用的配置管理机制是必不可少的，常见做法是使用配置服务器管理配置。

9.2 Spring Cloud Config 简介

Spring Cloud Config 为分布式系统外部化配置提供了服务器端和客户端的支持，它包括 Config Server 和 Config Client 两部分。由于 Config Server 和 Config Client 都实现了对 Spring Environment 和 PropertySource 抽象的映射，因此，Spring Cloud Config 非常适合 Spring 应用程序，当然也可与任何其他语言编写的应用程序配合使用。

Config Server 是一个可横向扩展、集中式的配置服务器，它用于集中管理应用程序各个环境下的配置，默认使用 Git 存储配置内容（也可使用 Subversion、本地文件系统或 Vault 存储配置，限于篇幅，本书不做讨论），因此可以很方便地实现对配置的版本控制与内容审计。

Config Client 是 Config Server 的客户端，用于操作存储在 Config Server 中的配置属性。如图 9-1 所示，所有的微服务都指向 Config Server。各个微服务在启动时，会请求 Config Server 以获取所需要的配置属性，然后缓存这些属性以提高性能。

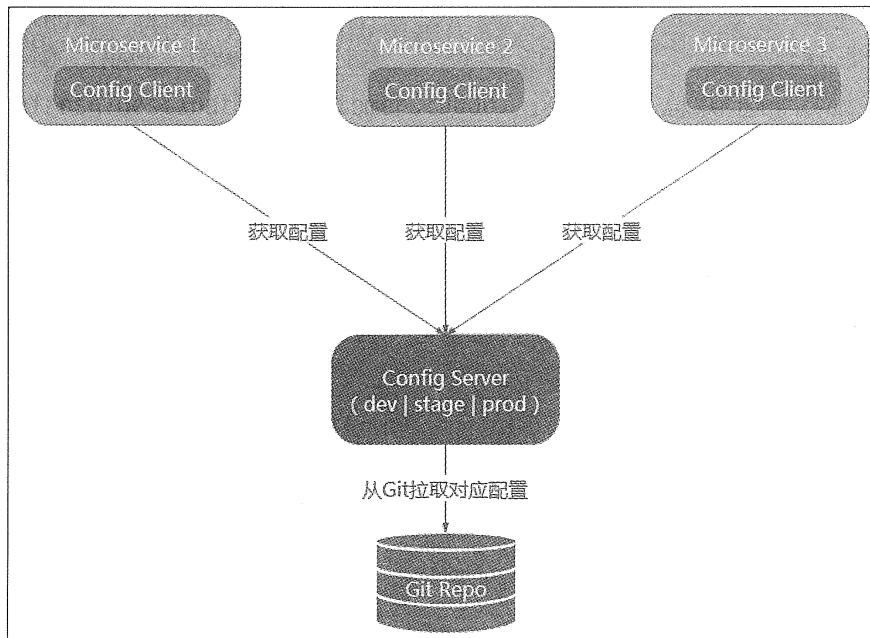


图 9-1 Spring Cloud Config 架构图



Spring Cloud Config 的 GitHub 地址: <https://github.com/spring-cloud/spring-cloud-config>。

9.3 编写 Config Server

本节来编写一个 Config Server。在本例中，使用 Git 作为 Config Server 的后端存储。

1. 在 Git 仓库<https://git.oschina.net/itmuch/spring-cloud-config-repo>中新建几个配置文件，例如：

```
microservice-foo.properties
microservice-foo-dev.properties
microservice-foo-test.properties
microservice-foo-production.properties
```

内容分别是：

```
profile=default-1.0
profile=dev-1.0
profile=test-1.0
profile=production-1.0
```

为了测试版本控制，为该 Git 仓库创建 config-label-v2.0 分支，并将各个配置文件中的 1.0 改为 2.0。

2. 创建一个 Maven 工程，ArtifactId 是 microservice-config-server，并为项目添加以下依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

3. 编写启动类，在启动类上添加注解 @EnableConfigServer，声明这是一个 Config Server。

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

4. 编写配置文件 application.yml，并在其中添加以下内容。

```
server:
  port: 8080
spring:
  application:
    name: microservice-config-server
  cloud:
    config:
      server:
        git:
          # 配置Git仓库的地址
          uri: https://git.oschina.net/itmuch/spring-cloud-config-repo
          # Git仓库的账号
          username:
          # Git仓库的密码
          password:
```

这样，一个 Config Server 就完成了。

Config Server 的端点

可以使用 Config Server 的端点获取配置文件的内容。端点与配置文件的映射规则如下：

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

以上端点都可以映射到{application}-{profile}.properties这个配置文件，{application}表示微服务的名称，{label}对应Git仓库的分支，默认是master。

按照以上规则，对于本例，可使用以下URL访问到Git仓库master分支的microservice-foo-dev.properties，例如：

- *http://localhost:8080/microservice-foo/dev*
- *http://localhost:8080/microservice-foo-dev.properties*
- *http://localhost:8080/microservice-foo-dev.yml*



测试

1. 访问 <http://localhost:8080/microservice-foo/dev>，可获得如下结果。

```
{  
    "name": "microservice-foo",  
    "profiles": [  
        "dev"  
    ],  
    "label": "master",  
    "version": "6791f5acf796efd14be92ec2b4fc31779ad97d46",  
    "state": null,  
    "propertySources": [  
        {  
            "name": "https://git.oschina.net/itmuch/spring-cloud-config-repo  
                  /microservice-foo-dev.properties",  
            "source": {  
                "profile": "dev-1.0"  
            }  
        },  
        {  
            "name": "https://git.oschina.net/itmuch/spring-cloud-config-repo  
                  /microservice-foo.properties",  
            "source": {  
                "profile": "default-1.0"  
            }  
        }  
    ]  
}
```

从结果可以直观地看到应用名称、项目 profile、Git label、Git version、配置文件 URL、配置详情等信息。

2. 访问 <http://localhost:8080/microservice-foo-dev.properties>，返回配置文件中的属性：

```
profile: dev-1.0
```

3. 访问 <http://localhost:8080/config-label-v2.0/microservice-foo-dev.properties>，可获得如下结果：

```
profile: dev-2.0
```

说明获得了 Git 仓库 config-label-v2.0 分支中的配置信息。

至此，已成功构建了 Config Server，并通过构造 URL 的方式，获取了 Git 仓库中的配置信息。



需要注意的是，访问 `http://localhost:8080/microservice-foo/dev`，结果中类似 `https://git.oschina.net/itmuch/spring-cloud-config-repo/microservice-foo-dev.properties` 的 URL 并不能访问。这是正常的，因为它并不代表配置文件的实际 URL 路径，而只是一个标识。有兴趣的读者可前往下列页面进行拓展阅读：<https://github.com/spring-cloud/spring-cloud-config/issues/571>。

9.4 编写 Config Client

前文已经构建了一个 Config Server，并使用 Config Server 端点获取配置内容。本节来讨论 Spring Cloud 微服务如何获取配置信息。

下面来编写一个微服务，该微服务整合了 Config Client。

1. 创建一个 Maven 工程，ArtifactId 是 `microservice-config-client`，并为项目添加以下依赖。

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

2. 创建一个基本的 Spring Boot 启动类。

```
@SpringBootApplication
public class ConfigClientApplication {
```

```

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

```

3. 编写配置文件 application.yml，并在其中添加如下内容。

```

server:
  port: 8081

```

4. 创建配置文件 bootstrap.yml，并在其中添加如下内容。

```

spring:
  application:
    # 对应config server所获取的配置文件的{application}
    name: microservice-foo
  cloud:
    config:
      uri: http://localhost:8080/
      # profile对应config server所获取的配置文件中的{profile}
      profile: dev
      # 指定Git仓库的分支，对应config server所获取的配置文件的{label}
      label: master

```

其中：

spring.application.name：对应 Config Server 所获取的配置文件中的 {application}。

spring.cloud.config.uri：指定 Config Server 的地址，默认是 `http://localhost:8888`。

spring.cloud.config.profile：profile 对应 Config Server 所获取的配置文件中的 {profile}。

spring.cloud.config.label：指定 Git 仓库的分支，对应 Config Server 所获取配置文件的 {label}。

值得注意的是，以上属性应配置在 bootstrap.yml，而不是 application.yml 中。如果配置在 application.yml 中，该部分配置就不能正常工作。例如，Config Client 会连接 `spring.cloud.config.uri` 的默认值 `http://localhost:8888`，而并非配置的 `http://localhost:8080/`。

Spring Cloud 有一个“引导上下文”的概念，这是主应用程序的父上下文。引导上下文负责从配置服务器加载配置属性，以及解密外部配置文件中的属性。和主应用程序加载 `application.*` (yml 或 properties) 中的属性不同，引导上下文加载 `bootstrap.*` 中的属性。配置在 `bootstrap.*` 中的属性有更高的优先级，因此默认情况下它们不能被本地配置覆盖。

如需禁用引导过程，可设置`spring.cloud.bootstrap.enabled=false`。

5. 编写 Controller。

```
@RestController  
public class ConfigClientController {  
    @Value("${profile}")  
    private String profile;  
  
    @GetMapping("/profile")  
    public String hello() {  
        return this.profile;  
    }  
}
```

在 Controller 中，通过注解`@Value("${profile}")`，绑定 Git 仓库配置文件中的 profile 属性。

测试

1. 启动 microservice-config-server。
2. 启动 microservice-config-client。
3. 访问`http://localhost:8081/profile`，可获得如下的结果。

dev-1.0

说明 Config Client 能够正常通过 Config Server 获得 Git 仓库中对应环境的配置。



Spring Cloud 引导上下文详解：http://cloud.spring.io/spring-cloud-static/Camden.SR4/#_the_bootstrap_application_context。

9.5 Config Server 的 Git 仓库配置详解

前文中，使用`spring.cloud.config.server.git.uri`指定了一个 Git 仓库，事实上，该属性非常灵活。本节来详细讨论 Config Server 的 Git 仓库配置。

1. 占位符支持

Config Server 的占位符支持`{application}`、`{profile}`和`{label}`。

示例：

```
server:  
  port: 8080  
spring:  
  application:  
    name: microservice-config-server  
  cloud:  
    config:  
      server:  
        git:  
          uri: https://git.oschina.net/itmuch/{application}  
          username:  
          password:
```

使用这种方式，即可轻松支持一个应用对应一个 Git 仓库。同理，也可支持一个 profile 对应一个 Git 仓库。

2. 模式匹配

模式匹配指的是带有通配符的 {application}/{profile} 名称的列表。如果 {application}/{profile} 不匹配任何模式，它将会使用 spring.cloud.config.server.git.uri 定义的 URI。

```
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
```

该例中，对于 simple 仓库，它只匹配所有配置文件中名为 simple 的应用程序。local 仓库则匹配所有配置文件中以 local 开头的所有应用程序的名称。

3. 搜索目录

很多场景下，可能把配置文件放在了 Git 仓库子目录中，此时可以使用 search-path 指定，search-path 同样支持占位符。

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://git.oschina.net/itmuch/spring-cloud-config-repo
          search-paths: foo,bar*
```

这样，Config Server 就会在 Git 仓库根目录、foo 子目录，以及所有以 bar 开始的子目录中查找配置文件。

4. 启动时加载配置文件

默认情况下，在配置被首次请求时，Config Server 才会 clone Git 仓库。也可让 Config Server 在启动时就 clone Git 仓库，例如：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            team-a:
              pattern: microservice-*
              clone-on-start: true
              uri: http://git.oschina.net/itmuch/spring-cloud-config-repo
```

将属性 `spring.cloud.config.server.git.repos.*.clone-on-start` 设为 `true`，即可让 Config Server 启动时 clone 指定 Git 仓库。

当然，也可使用 `spring.cloud.config.server.git.clone-on-start = true` 进行全局配置。

配置 `clone-on-start = true`，可帮助 Config Server 启动时快速识别错误的配置源（例如无效的 Git 仓库）。



将以下包的日志级别设为 DEBUG，就可打印 Config Server 请求 Git 仓库的细节。可以通过这种方式，更好地理解 Config Server 的 Git 仓库配置，同时也便于快速定位问题。

```
logging:  
  level:  
    org.springframework.cloud: DEBUG  
    org.springframework.boot: DEBUG
```

9.6 Config Server 的健康状况指示器

Config Server 自带了一个健康状况指示器，用于检查所配置的 EnvironmentRepository 是否正常工作。可使用 Config Server 的/health 端点查询当前健康状态。默认情况下，健康指示器向 EnvironmentRepository 请求的 {application} 是 app，{profile} 和 {label} 是对应 EnvironmentRepository 实现的默认值。对于 Git，{profile} 是 default，{label} 是 master。

同样也可以自定义健康状况指示器的配置，从而检查更多的 {application}、自定义的 {profile} 以及自定义的 {label}，例如：

```
server:  
  port: 8080  
spring:  
  application:  
    name: microservice-config-server  
  cloud:  
    config:  
      server:  
        git:  
          # 配置Git仓库的地址  
          uri: https://git.oschina.net/itmuch/spring-cloud-config-repo/  
          # Git仓库的账号  
          username:  
            # Git仓库的密码  
            password:  
  health:  
    repositories:  
      a-foo:  
        label: config-label-v2.0
```

```
name: microservice-foo
profiles: dev
```

详见本书配套代码中的 microservice-config-server-health 项目。

如需禁用健康状况指示器，可设置 spring.cloud.config.server.health.enabled=false。

9.7 配置内容的加解密

前文是在 Git 仓库中明文存储配置属性的。很多场景下，对于某些敏感的配置内容（例如数据库账号、密码等），应当加密存储。

Config Server 为配置内容的加密与解密提供了支持。

9.7.1 安装 JCE

Config Server 的加解密功能依赖 Java Cryptography Extension (JCE)。

Java 8 JCE 的地址是：<http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html>。

下载 JCE 并解压，按照其中的 README.txt 的说明安装。JCE 的安装非常简单，其实就是将 JDK/jre/lib/security 目录中的两个 jar 文件替换为压缩包中的 jar 文件。

其他 Java 版本的 JCE 地址，在 Spring Cloud 文档中都有提及，详见：http://cloud.spring.io/spring-cloud-static/Camden.SR4/#_cloud_native_applications。

9.7.2 Config Server 的加解密端点

Config Server 提供了加密与解密的端点，分别是 /encrypt 与 /decrypt。可使用以下代码来加密明文：

```
curl $CONFIG_SERVER_URL/encrypt -d 想要加密的明文
```

使用以下代码来解密密文：

```
curl $CONFIG_SERVER_URL/decrypt -d 想要解密的密文
```

9.7.3 对称加密

1. 复制项目 microservice-config-server，将 ArtifactId 修改为 microservice-config-server-encryption。
2. 修改 application.yml，添加以下内容。

```
encrypt:
  key: foo # 设置对称密钥
```

这样，代码就编写完成了。

测试

- 输入命令：

```
curl http://localhost:8080/encrypt -d mysecret
```

可返回 851a6effab6619f43157a714061f4602be0131b73b56b0451a7e268c880daea3，说明 mysecret 已被加密。

- 输入命令：

```
curl http://localhost:8080/decrypt -d
851a6effab6619f43157a714061f4602be0131b73b56b0451a7e268c880daea3
```

可返回 mysecret，说明能够正常解密。

9.7.4 存储加密的内容

加密后的内容，可使用 {cipher} 密文的形式存储。

- 准备一个配置文件，命名为 encryption.yml。

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}851a6effab6619f43157a714061f4602be0131b73b56b0451a7e268c8
80daea3'
```

并将其 push 到 Git 仓库 <https://git.oschina.net/itmuch/spring-cloud-config-repo>。此处需注意 spring.datasource.password 上的单引号不能少。如读者使用 properties 格式管理配置，则不能使用单引号，否则该值不会被解密，正确写法如下：

```
spring.datasource.username=dbuser
spring.datasource.password={cipher}851a6effab6619f43157a714061f4602be0131b73b56b
0451a7e268c880daea3
```

- 使用 <http://localhost:8080/encryption-default.yml> 可获得如下结果。

```
profile: default
spring:
  datasource:
```

```
password: mysecret
username: dbuser
```

说明 Config Server 能自动解密配置内容。

一些场景下，想要让 Config Server 直接返回密文本身，而并非解密后的内容，可设置`spring.cloud.config.server.encrypt.enabled=false`，这时可由 Config Client 自行解密。

9.7.5 非对称加密

前文讨论了对称加密的方式，Spring Cloud 同样支持非对称加密。

1. 复制项目`microservice-config-server`，将`ArtifactId`修改为`microservice-config-server-encryption-rsa`。
2. 执行以下命令，并按照提示操作，即可创建一个 Key Store。

```
keytool -genkeypair -alias mytestkey -keyalg RSA -dname "CN=Web Server,OU=Unit,0=Organization,L=City,S=State,C=US" -keystore server.jks -storepass letmein
```

3. 将生成的`server.jks`文件复制到项目的`classpath`下。
4. 在`application.yml`中添加以下内容。

```
encrypt:
  keyStore:
    location: classpath:/server.jks # jks文件的路径
    password: letmein             # storepass
    alias: mytestkey              # alias
    secret: changeme               # keypass
```

这样，使用以下命令：

```
curl http://localhost:8080/encrypt -d mysecret
```

尝试加密时，就会得到类似以下的结果：

```
AQB38UyNckYzW64rvsaIhy00V4MUmS7krdHrw+VLUdqXJ4ZVdZL8/ouwSOAYM+6MSjKvzmkaU8Iv2cQ5
MWhlZhCrm0f0d2ubc1MH96KBHTix9AroajeTiofPwPoBnWfBo9cC4PU1vD+rcvAvvvdR5q7rYbFc4yut
4uJZRzpAXGgf680kAtb6tEtLx7c4/35PEaGXFwd2m8gn21vzWdvhbP6cdC9YlburL0Rq/0H1G+uEX99Z
VIWJ0hVn4rpLLWPMLUGA2ZVEyVRorIRX/2z5MU7cVPtJ6X1JZDpU4GVz8/3rD5BnbVFTGo6DfBrEzJn5
8Bzjl6aqo9ca/3j42RH0oQDOHXGqRX/843RbPdvMqTzd0rTOBHTUrVG9E15sCajiLkw=
```

相对于对称加密，非对称加密的安全性更高，但对称加密相对方便。读者可按照需求，自行选择加密方案。

9.8 使用/refresh 端点手动刷新配置

很多场景下，需要在运行期间动态调整配置。如果配置发生了修改，微服务要如何实现配置的刷新呢？

要想实现配置刷新，须对之前的代码进行一点改造。

1. 复制项目 microservice-config-client，将 ArtifactId 修改为 microservice-config-client-refresh。
2. 为项目添加 spring-boot-starter-actuator 的依赖，该依赖包含了 /refresh 端点，用于配置的刷新。

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

3. 在 Controller 上添加注解 @RefreshScope。添加 @RefreshScope 的类会在配置更改时得到特殊的处理。

```
@RestController
@RefreshScope
public class ConfigClientController {
    @Value("${profile}")
    private String profile;

    @GetMapping("/profile")
    public String hello() {
        return this.profile;
    }
}
```

这样就完成了代码改造。



测试

1. 启动 microservice-config-server。
2. 启动 microservice-config-client-refresh。
3. 访问 <http://localhost:8081/profile>，获得结果：dev-1.0。
4. 修改 Git 仓库中 microservice-foo-dev.properties 文件内容为 profile=dev-1.0-change。

5. 重新访问`http://localhost:8081/profile`, 发现结果依然是 dev-1.0, 说明配置尚未刷新。

6. 发送 POST 请求到`http://localhost:8081/refresh`, 例如:

```
curl -X POST http://localhost:8081/refresh
```

返回结果: "profile", 表示 profile 这个配置属性已被刷新。

7. 再次访问`http://localhost:8081/profile`, 返回 dev-1.0-change, 说明配置已经刷新。

9.9 使用 Spring Cloud Bus 自动刷新配置

前文讨论了使用`/refresh`端点手动刷新配置, 但如果所有微服务节点的配置都需要手动去刷新, 工作量可想而知。不仅如此, 随着系统的不断扩张, 会越来越难以维护。因此, 实现配置的自动刷新是很必要的, 本节将使用 Spring Cloud Bus 实现配置的自动刷新。

9.9.1 Spring Cloud Bus 简介

Spring Cloud Bus 使用轻量级的消息代理(例如 RabbitMQ、Kafka 等)连接分布式系统的节点, 这样就可以广播传播状态的更改(例如配置的更新)或者其他管理指令。可将 Spring Cloud Bus 想象成一个分布式的 Spring Boot Actuator。使用 Spring Cloud Bus 后的架构如图 9-2 所示。

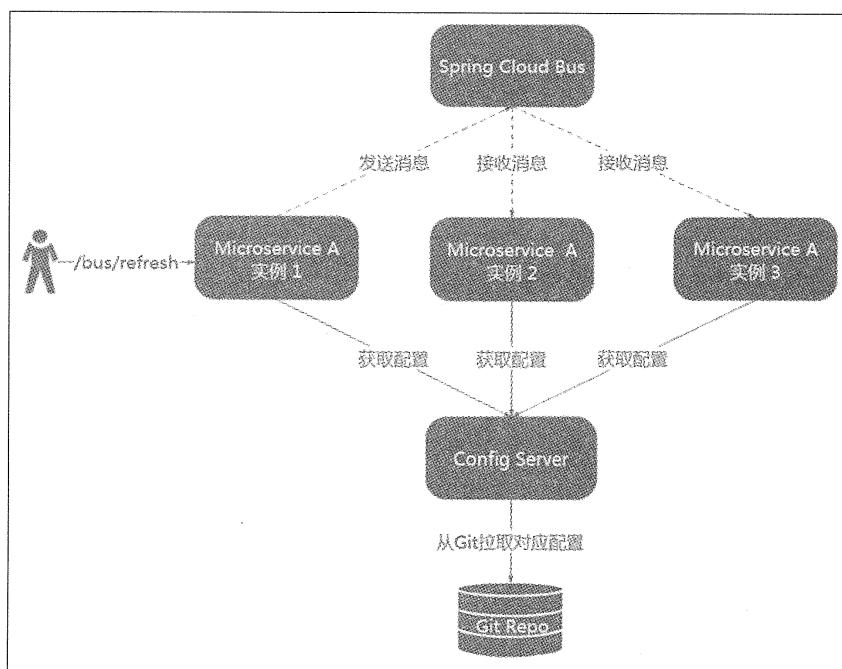


图 9-2 使用 Spring Cloud Bus 的架构图

由图可知，微服务 A 的所有实例都通过消息总线连接到了一起，每个实例都会订阅配置更新事件。当其中一个微服务节点的 /bus/refresh 端点被请求时，该实例就会向消息总线发送一个配置更新事件，其他实例获得该事件后也会更新配置。

9.9.2 实现自动刷新

安装 RabbitMQ（详见 7.5.3.1 节）后，接下来为项目整合 Spring Cloud Bus 并实现自动刷新。

1. 复制项目 microservice-config-client-refresh，将 ArtifactId 修改为 microservice-config-client-refresh-cloud-bus。
2. 为项目添加 spring-cloud-starter-bus-amqp 的依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

3. 在 bootstrap.yml 中添加以下内容：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

这样代码就改造完成了。

测试

1. 启动 microservice-config-server。
2. 启动 microservice-config-client-refresh-cloud-bus，可发现此时控制台会输出类似于如下的内容。

```
[main] o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "{{[/bus/refresh],methods=[POST]}}" onto public void org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String)
```

由结果可知，此时项目有一个 /bus/refresh 端点。

3. 将 microservice-config-client-refresh-cloud-bus 的端口改为 8082，再启动一个节点。

4. 访问`http://localhost:8081/profile`，可获得结果：dev-1.0。
5. 将 Git 仓库中的 `microservice-foo-dev.properties` 文件内容修改为如下内容。

```
profile=dev-1.0-bus
```

6. 发送 POST 请求到其中一个 Config Client 实例的`/bus/refresh` 端点，例如：

```
curl -X POST http://localhost:8081/bus/refresh
```

7. 访问两个 Config Client 节点的`/profile` 端点，会发现两个节点都会返回`dev-1.0-bus`，说明配置内容已被刷新。

借助 Git 仓库的 WebHooks，就可轻松实现配置的自动刷新。如图 9-3 所示。

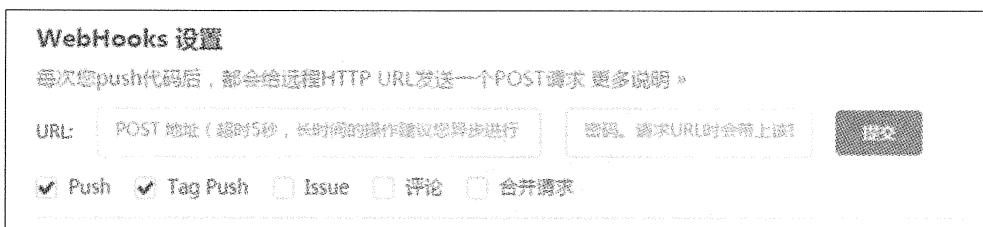


图 9-3 Git WebHooks 设置

9.9.3 局部刷新

某些场景下（例如灰度发布等），若只想刷新部分微服务的配置，可通过`/bus/refresh` 端点的 `destination` 参数来定位要刷新的应用程序。

例如：`/bus/refresh?destination=customers:9000`，这样消息总线上的微服务实例就会根据 `destination` 参数的值来判断是否需要刷新。其中，`customers:9000` 指的是各个微服务的 `ApplicationContext ID`。

`destination` 参数也可以用来定位特定的微服务。例如：`/bus/refresh?destination=customers:**`，这样就可以触发 `customers` 微服务所有实例的配置刷新。



配置的局部刷新与 `ApplicationContext ID` 有关。默认情况下，`ApplicationContext ID` 是 `spring.application.name:server.port`。笔者在博客中对此有较为详细的分析，详见：<http://www.itmuch.com/spring-cloud-code-read/spring-cloud-code-read-spring-cloud-bus/>。

9.9.4 架构改进

在前面的示例中，通过请求某个微服务/bus/refresh 端点的方式来实现配置刷新，但这种方式并不优雅。原因如下：

1. 破坏了微服务的职责单一原则。业务微服务只应关注自身业务，不应承担配置刷新的职责。
2. 破坏了微服务各节点的对等性。
3. 有一定的局限性。例如，微服务在迁移时，网络地址常常会发生变化。此时如果想自动刷新配置，就不得不修改 WebHook 的配置。

不妨改进一下架构，如图 9-4 所示。

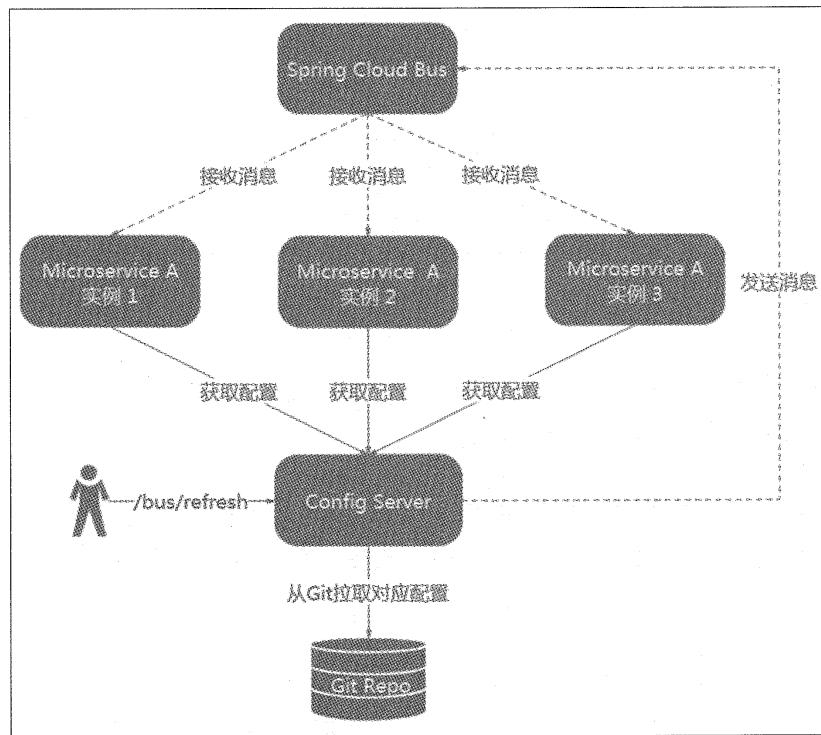


图 9-4 使用 Spring Cloud Bus 的架构图

如图 9-4 所示，将 Config Server 也加入到消息总线中，并使用 Config Server 的/bus/refresh 端点来实现配置的刷新。这样，各个微服务只需要关注自身的业务，而不再承担配置刷新的职责。

详见配套代码中的 microservice-config-server-refresh-cloud-bus 项目。

9.9.5 跟踪总线事件

一些场景下如果希望知道 Spring Cloud Bus 事件传播的细节，可以跟踪总线事件（`RemoteApplicationEvent` 的子类都是总线事件）。

想要跟踪总线事件非常简单，只须设置`spring.cloud.bus.trace.enabled=true`，这样在`/bus/refresh` 端点被请求后，访问`/trace` 端点就可获得类似如下的结果：

```
{  
    "timestamp": 1481098786017,  
    "info": {  
        "signal": "spring.cloud.bus.ack",  
        "event": "RefreshRemoteApplicationEvent",  
        "id": "66d172e0-e770-4349-baf7-0210af62ea8d",  
        "origin": "microservice-foo:8081",  
        "destination": "***"  
    }  
, {  
    "timestamp": 1481098779073,  
    "info": {  
        "signal": "spring.cloud.bus.sent",  
        "type": "RefreshRemoteApplicationEvent",  
        "id": "66d172e0-e770-4349-baf7-0210af62ea8d",  
        "origin": "microservice-config-server:8080",  
        "destination": "***:***"  
    }  
}...  
}
```

这样就可清晰地知道事件的传播细节。

9.10 Spring Cloud Config 与 Eureka 配合使用

前文在微服务中指定了 Config Server 地址，这种方式无法利用服务发现组件的优势。本节将讨论 Config Server 注册到 Eureka Server 上时，如何使用 Spring Cloud Config。

1. 将 Config Server 和 Config Client 都注册到 Eureka Server 上。
2. Config Client 的 `bootstrap.yml` 可配置如下。

```
spring:  
  application:  
    # 对应config server所获取的配置文件的{application}
```

```
name: microservice-foo
cloud:
  config:
    profile: dev
    label: master
    discovery:
      # 表示使用服务发现组件中的Config Server, 而不自己指定Config Server
      # 的uri, 默认false
      enabled: true
      # 指定Config Server在服务发现中的serviceId, 默认是configserver
      service-id: microservice-config-server-eureka
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

简要说明一下 bootstrap.yml 中的配置属性：

- `spring.cloud.config.discovery.enabled = true`: 表示开启通过服务发现组件访问 Config Server 的功能；
- `spring.cloud.config.discovery.service-id`: 指定 Config Server 在服务发现组件中的 serviceId。

详见本书配套代码中的 `microservice-config-server-eureka` 项目和 `microservice-config-client-eureka` 项目。

9.11 Spring Cloud Config 的用户认证

在前文的示例中，Config Server 是允许匿名访问的。为了防止配置内容的外泄，应该保护 Config Server 的安全。有多种方式做到这一点，例如通过物理网络安全，或者为 Config Server 添加用户认证等。

本节来为 Config Server 添加基于 HTTP Basic 的用户认证。

先来构建一个需要用户认证的 Config Server。

1. 复制项目 `microservice-config-server`，将 `ArtifactId` 修改为 `microservice-config-server-authenticating`。
2. 为项目添加以下依赖。

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

3. 在 application.yml 中添加如下内容。

```
security:
  basic:
    enabled: true          # 开启基于HTTP basic的认证
  user:
    name: user            # 配置登录的账号是user
    password: password123 # 配置登录的密码是password123
```

这样就为 Config Server 添加了基于 HTTP basic 的认证。如果不设置这段内容，账号默认是 user，密码是一个随机值，该值会在启动时打印出来。

此时启动该 Config Server，将会看到如下的登录对话框，如图 9-5 所示。

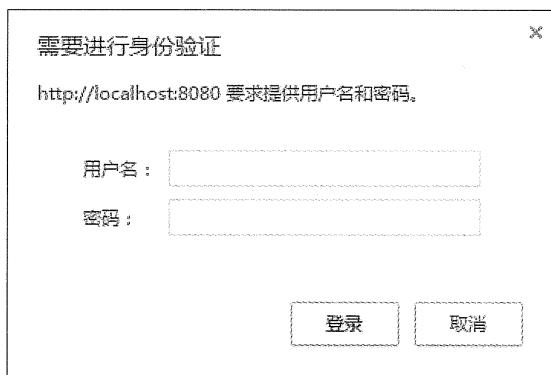


图 9-5 Config Server 登录界面

Config Client 连接需用户认证的 Config Server

Config Client 有两种方式使用需用户认证的 Config Server。

方式一：使用 curl 风格的 URL。示例：

```
spring:
  cloud:
    config:
      uri: http://user:password123@localhost:8080/
```

方法二：指定 Config Server 的账号与密码。示例：

```
spring:  
  cloud:  
    config:  
      uri: http://localhost:8080/  
      username: user  
      password: password123
```

需要注意的是`spring.cloud.config.password` 和 `spring.cloud.config.username` 的优先级较高，它们会覆盖 URL 中包含的账号与密码。



对于 Config Server 注册到 Eureka Server 的情况，可参考本书配套代码中的 `microservice-config-server-eureka-authenticating` 项目和 `microservice-config-client-eureka-authenticating` 项目。

9.12 Config Server 的高可用

前文构建的都是单节点的 Config Server，本节来讨论如何构建高可用的 Config Server 集群，包括 Config Server 的高可用依赖 Git 仓库的高可用以及 RabbitMQ 的高可用。

下面先来讨论 Git 仓库的高可用。

9.12.1 Git 仓库的高可用

由于配置内容都存储在 Git 仓库中，所以要想实现 Config Server 的高可用，必须有一个高可用的 Git 仓库。有两种方式可以实现 Git 仓库的高可用。

- 使用第三方 Git 仓库：这种方式非常简单，可使用例如 GitHub、BitBucket、git@osc、Coding 等提供的仓库托管服务，这些服务本身就已实现了高可用。
- 自建 Git 仓库管理系统：使用第三方服务的方式虽然省去了很多烦恼，但是很多场景下，倾向于自建 Git 仓库管理系统。此时就需要保证自建 Git 的高可用。

以 GitLab 为例，读者可参照官方文档搭建高可用的 GitLab：<https://about.gitlab.com/high-availability/>。

9.12.2 RabbitMQ 的高可用

还记得前文使用 Spring Cloud Bus 实现了配置的自动刷新吗？由于 Spring Cloud Bus 依赖 RabbitMQ（当然也可使用其他 MQ），所以 RabbitMQ 的高可用也是必不可少的。

搭建高可用 RabbitMQ 的资料详见：<https://www.rabbitmq.com/ha.html>。由于比较简单，笔者不做赘述。当然，也可使用云平台的提供的 RabbitMQ 服务。

9.12.3 Config Server 自身的高可用

本节来讨论如何实现 Config Server 自身的高可用。笔者分两种场景进行讨论。

Config Server 未注册到 Eureka Server 上

对于这种情况，Config Server 的高可用可借助一个负载均衡器来实现，如图 9-6 所示。

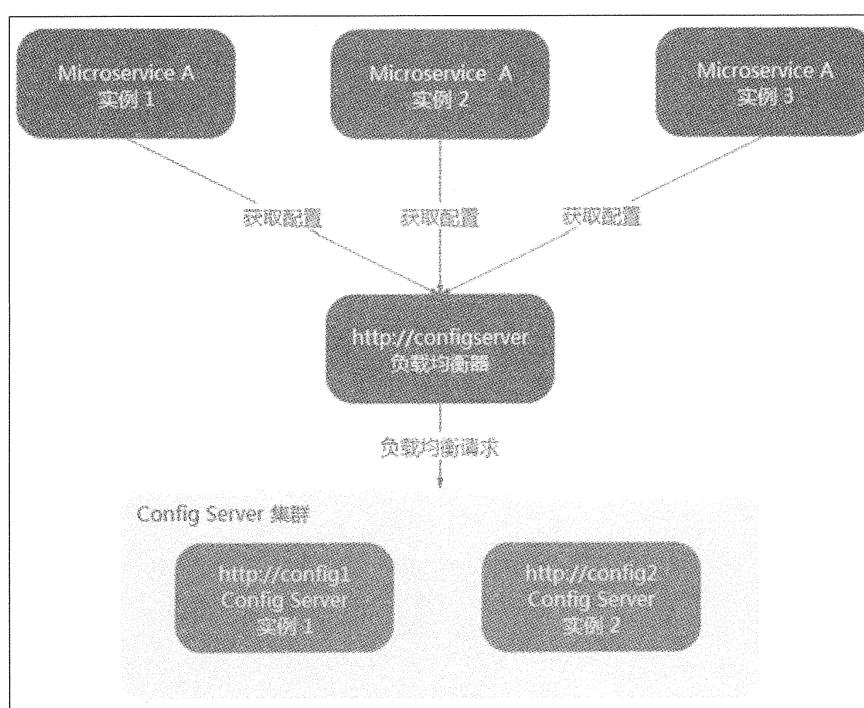


图 9-6 Config Server 高可用架构图

如图所示，各个微服务将请求发送到负载均衡器，负载均衡器将请求转发到其代理的其中一个 Config Server 节点。这样，就可以实现 Config Server 的高可用。

Config Server 注册到 Eureka Server 上

这种情况下，Config Server 的高可用相对简单，只须将多个 Config Server 节点注册到 Eureka Server 上，即可实现 Config Server 的高可用。架构如图 9-7 所示。

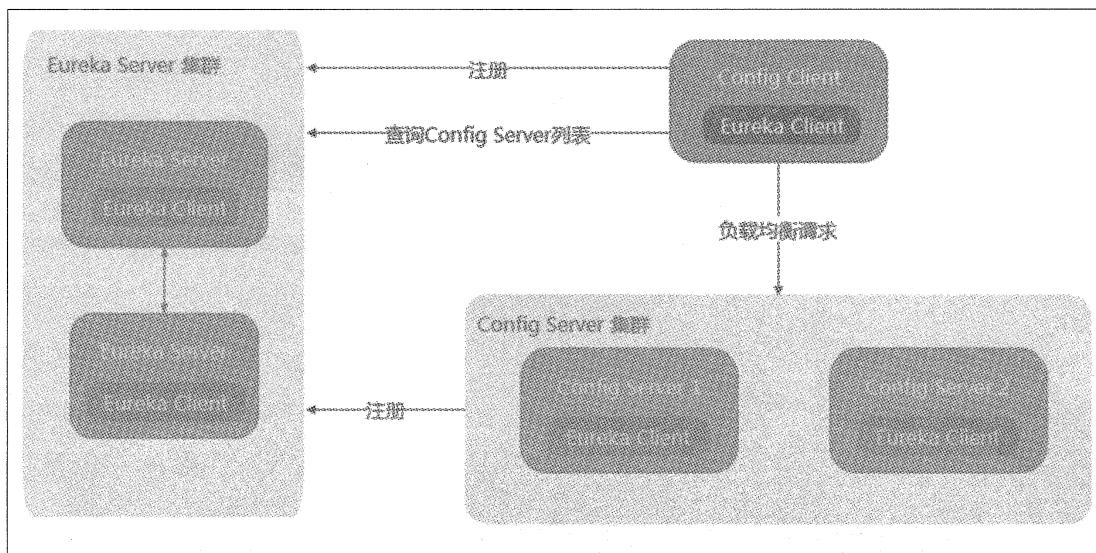


图 9-7 Config Server 高可用架构图

使用 Spring Cloud Sleuth 实现微服务 跟踪

前文已经讲过几种监控微服务的方式，例如使用 Spring Boot Actuator 监控微服务实例，使用 Hystrix 监控 Hystrix Command 等。

本章来讨论微服务“跟踪”。

10.1 为什么要实现微服务跟踪

谈到微服务跟踪，就不得不提一下 Peter Deutsch 的文章 *The Eight Fallacies of Distributed Computing*（分布式计算的八大误区），内容大致如下：

- 网络可靠
- 延迟为零
- 带宽无限
- 网络绝对安全
- 网络拓扑不会改变

- 必须有一名管理员
- 传输成本为零
- 网络同质化（由 Java 之父 Golsing 补充）

从中可以看到，该文章很多点都在描述一个问题——网络问题。网络常常很脆弱，同时，网络资源也是有限的。

我们知道，微服务之间通过网络进行通信。如果能够跟踪每个请求，了解请求经过哪些微服务（从而了解信息是如何在服务之间流动）、请求耗费时间、网络延迟、业务逻辑耗费时间等指标，那么就能更好地分析系统瓶颈、解决系统问题。因此，微服务跟踪很有必要。



The Eight Fallacies of Distributed Computing (分布式计算的八大误区) 原文：
<https://blogs.oracle.com/jag/resource/Fallacies.html>。

10.2 Spring Cloud Sleuth 简介

Spring Cloud Sleuth 为 Spring Cloud 提供了分布式跟踪的解决方案，它大量借用了 Google Dapper、Twitter Zipkin 和 Apache HTrace 的设计。

先来了解一下 Sleuth 的术语，Sleuth 借用了 Dapper 的术语。

- **span (跨度)**：基本工作单元。span 用一个 64 位的 id 唯一标识。除 ID 外，span 还包含其他数据，例如描述、时间戳事件、键值对的注解（标签），span ID、span 父ID 等。
span 被启动和停止时，记录了时间信息。初始化 span 被称为“root span”，该 span 的 id 和 trace 的 ID 相等。
- **trace (跟踪)**：一组共享“root span”的 span 组成的树状结构称为 trace。trace 也用一个 64 位的 ID 唯一标识，trace 中的所有 span 都共享该 trace 的 ID。
- **annotation (标注)**：annotation 用来记录事件的存在，其中，核心 annotation 用来定义请求的开始和结束。
 - CS (Client Sent 客户端发送)：客户端发起一个请求，该 annotation 描述了 span 的开始。
 - SR (Server Received 服务器端接收)：服务器端获得请求并准备处理它。如果用 SR 减去 CS 时间戳，就能得到网络延迟。

- SS (Server Sent 服务器端发送): 该 annotation 表明完成请求处理 (当响应发回客户端时)。如果用 SS 减去 SR 时间戳，就能得到服务器端处理请求所需的时间。
- CR (Client Received 客户端接收): span 结束的标识。客户端成功接收到服务器端的响应。如果 CR 减去 CS 时间戳，就能得到从客户端发送请求到服务器响应的所需的时间。

图 10-1 详细描述了请求依次经过 SERVICE1—SERVICE2—SERVICE3—SERVICE4 时，span、trace、annotation 的变化。

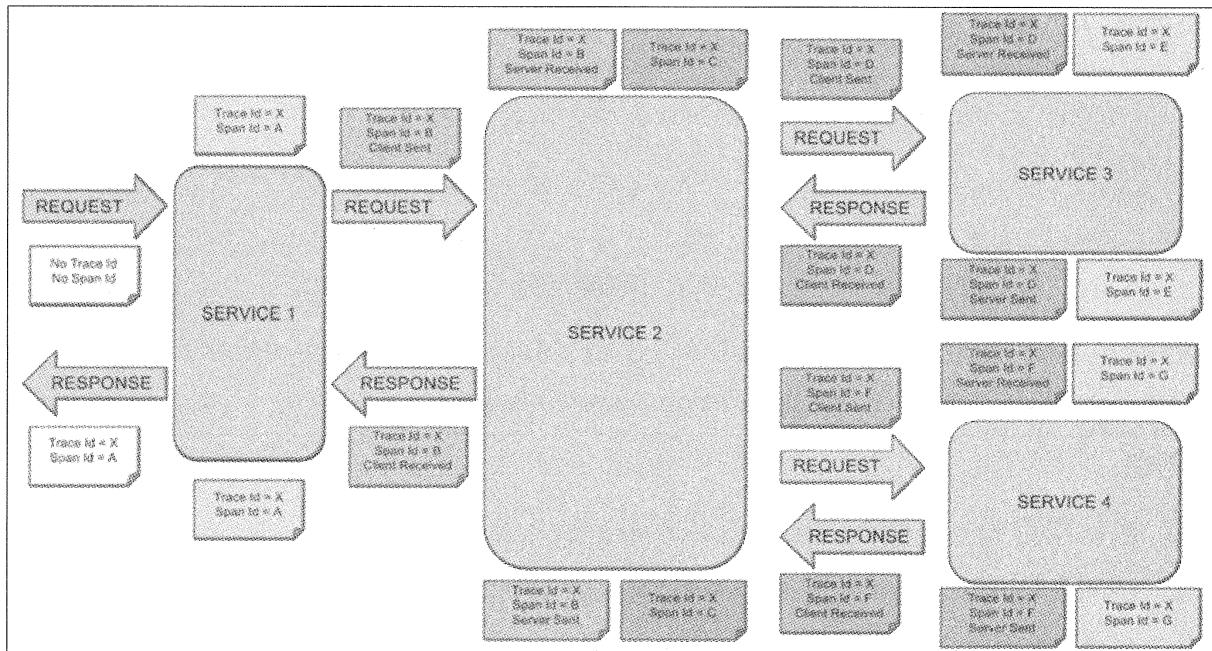


图 10-1 微服务追踪



读者如无法理解这些术语，也不必担心，在实战时会有直观的体验。有一定基础后再来看本节，就能深刻理解这些术语的含义。



Dapper 论文: <https://research.google.com/pubs/pub36356.html>。

10.3 整合 Spring Cloud Sleuth

本节来为之前编写的项目整合 Sleuth。简单起见，使用 microservice-simple-provider-user 这个最原始的项目进行改造。

1. 复制项目 microservice-simple-provider-user，将 ArtifactId 修改为 microservice-simple-provider-user-trace。
2. 为项目添加以下依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

3. 修改 application.yml，添加以下内容：

```
spring:
  application:
    name: microservice-provider-user
logging:
  level:
    root: INFO
    org.springframework.web.servlet.DispatcherServlet: DEBUG
```

这样就整合好 Sleuth 了。同理，也可为项目 microservice-simple-consumer-movie 整合 Sleuth，记为 microservice-simple-consumer-movie-trace。



测试

1. 启动项目 microservice-simple-provider-user-trace。
2. 访问 <http://localhost:8000/1>，控制台会输出类似如下的日志。

```
2017-01-17 15:07:03.874 DEBUG [microservice-provider-user,22f5f12f22222a36
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.web.servlet.Dispa
tcherServlet : DispatcherServlet with name 'dispatcherServlet' proc
essing GET request for [/1]
2017-01-17 15:07:03.877 DEBUG [microservice-provider-user,22f5f12f22222a36
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.web.servlet.Dispa
tcherServlet : Last-Modified value for [/1] is: -1
2017-01-17 15:07:03.882 DEBUG [microservice-provider-user,22f5f12f22222a36
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.web.servlet.Dispa
tcherServlet : Null ModelAndView returned to DispatcherServlet with
name 'dispatcherServlet': assuming HandlerAdapter completed request handl
ing
2017-01-17 15:07:03.883 DEBUG [microservice-provider-user,22f5f12f22222a36
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.web.servlet.Dispa
tcherServlet : Successfully completed request
```

其中，22f5f12f22222a36 是 trace ID，ba88c199dbfa022a 等是 span ID。仔细分析日志，不难看出请求的具体过程。也可将日志如下设置：

```
logging:  
  level:  
    root: INFO  
    org.springframework.cloud.sleuth: DEBUG
```

这样，就能了解 span 从创建到关闭的详细过程，例如：

```
2017-01-17 15:07:03.873 DEBUG [microservice-provider-user,22f5f12f22222a36  
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.c.sleuth.instrume  
nt.web.TraceFilter : No parent span present - creating a new span  
2017-01-17 15:07:03.877 DEBUG [microservice-provider-user,22f5f12f22222a36  
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.c.s.i.web.TraceHa  
ndlerInterceptor : Created new span [Trace: 22f5f12f22222a36, Span: ba8  
8c199dbfa022a, Parent: 22f5f12f22222a36, exportable:false] with name [find  
-by-id]  
2017-01-17 15:07:03.877 DEBUG [microservice-provider-user,22f5f12f22222a36  
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.c.s.i.web.TraceHa  
ndlerInterceptor : Adding a method tag with value [findById] to a span  
[Trace: 22f5f12f22222a36, Span: ba88c199dbfa022a, Parent: 22f5f12f22222a36  
, exportable:false]  
2017-01-17 15:07:03.877 DEBUG [microservice-provider-user,22f5f12f22222a36  
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.c.s.i.web.TraceHa  
ndlerInterceptor : Adding a class tag with value [UserController] to a span  
[Trace: 22f5f12f22222a36, Span: ba88c199dbfa022a, Parent: 22f5f12f222  
22a36, exportable:false]  
2017-01-17 15:07:03.882 DEBUG [microservice-provider-user,22f5f12f22222a36  
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.c.s.i.web.TraceHa  
ndlerInterceptor : Closing span [Trace: 22f5f12f22222a36, Span: ba88c19  
9dbfa022a, Parent: 22f5f12f22222a36, exportable:false]  
2017-01-17 15:07:03.883 DEBUG [microservice-provider-user,22f5f12f22222a36  
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.c.sleuth.instrume  
nt.web.TraceFilter : Closing the span [Trace: 22f5f12f22222a36, Span: 22f  
5f12f22222a36, Parent: null, exportable:false] since the response was succ  
essful
```

3. 启动项目 microservice-simple-consumer-movie-trace。
4. 访问<http://localhost:8010/user/1>会发现两个项目都会打印类似以上的日志。

10.4 Spring Cloud Sleuth 与 ELK 配合使用

ELK 是一款非常流行的日志分析系统，相信大家都不陌生。本节来为 Sleuth 整合 ELK。

1. 搭建 ELK。

本书使用 ELK 5.1.2 进行测试。ELK 的搭建比较简单，读者可参考官方文档（<https://www.elastic.co/guide/index.html>）自行安装。

2. 复制项目 microservice-simple-provider-user-trace，将 ArtifactId 修改为 microservice-simple-provider-user-trace-elk。

3. 为项目添加以下依赖。

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.6</version>
</dependency>
```

4. 在项目的 src/main/resources 目录下新建文件 logback-spring.xml，在其中添加如下内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml" />

    <springProperty scope="context" name="springAppName" source="spring.
        application.name" />
    <!-- Example for logging into the build folder of your project -->
    <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}" />

    <property name="CONSOLE_LOG_PATTERN"
        value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5
        p}) %clr([${springAppName:-},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-
        B3-ParentSpanId:-},%X{X-Span-Export:-}])%{yellow} %clr(${PID:- })%{magenta
        } %clr(---){faint} %clr[%15.15t]{faint} %clr(%-40.40logger{39})%{cyan}
        %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}" />

    <!-- Appender to log to console -->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <!-- Minimum logging level to be presented in the console logs -->
            <level>DEBUG</level>
```

```
</filter>
<encoder>
    <pattern>${CONSOLE_LOG_PATTERN}</pattern>
    <charset>utf8</charset>
</encoder>
</appender>

<!-- Appender to log to file -->
<appender name="flatfile" class="ch.qos.logback.core.rolling.
    RollingFileAppender">
    <file>${LOG_FILE}</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${LOG_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>
        <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder>
        <pattern>${CONSOLE_LOG_PATTERN}</pattern>
        <charset>utf8</charset>
    </encoder>
</appender>

<!-- Appender to log to file in a JSON format -->
<appender name="logstash" class="ch.qos.logback.core.rolling.
    RollingFileAppender">
    <file>${LOG_FILE}.json</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
        <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder class="net.logstash.logback.encoder.
        LoggingEventCompositeJsonEncoder">
        <providers>
            <timestampl>
                <timeZone>UTC</timeZone>
            </timestampl>
            <pattern>
                <pattern>
                    {
                        "severity": "%level",

```

```

        "service": "${springAppName:-}",
        "trace": "%X{X-B3-TraceId:-}",
        "span": "%X{X-B3-SpanId:-}",
        "parent": "%X{X-B3-ParentSpanId:-}",
        "exportable": "%X{X-Span-Export:-}",
        "pid": "${PID:-}",
        "thread": "%thread",
        "class": "%logger{40}",
        "rest": "%message"
    }
</pattern>
</pattern>
</providers>
</encoder>
</appender>

<root level="INFO">
    <appender-ref ref="console" />
    <appender-ref ref="logstash" />
    <!--<appender-ref ref="flatfile"/> -->
</root>
</configuration>

```

5. 编写 bootstrap.yml，并将 application.yml 中的以下属性移动到 bootstrap.yml 中。

```

spring:
  application:
    name: microservice-provider-user

```

由于使用了自定义的 logback-spring.xml，并且该文件中含有变量（例如 springAppName），spring.application.name 属性必须设置在 bootstrap.yml 文件中，否则，logback-spring.xml 将无法正确读取属性。

6. 编写 Logstash 配置文件，命名为 logstash.conf，内容如下。

```

input {
  file {
    codec => json
    path => "/opt/build/*.json" # 改成你项目打印的json日志文件。
  }
}
filter {

```

```

grok {
  match => { "message" => "%{TIMESTAMP_ISO8601:timestamp}\s+%\{LOGLEVEL:
    severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:
    exportable}\]\s+%\{DATA:pid\}---\s+\[%{DATA:thread}\]\s+%\{DATA:class}\s+:\s+
    +%\{GREEDYDATA:rest\}" }
}

output {
  elasticsearch {
    hosts => "elasticsearch:9200" # 改成你的Elasticsearch地址
  }
}

```

测试

- 启动 ELK (其中, 使用本节编写的 logstash.conf 启动 Logstash)。
- 将项目 microservice-simple-provider-user-trace-elk 打包成 jar 包, 并将其放在 /opt 目录下。
- 多次访问 <http://localhost:8000/1>, 产生一些日志。
- 访问 <http://localhost:5601>, 可以看到 Kibana 的首页, 如图 10-2 所示。

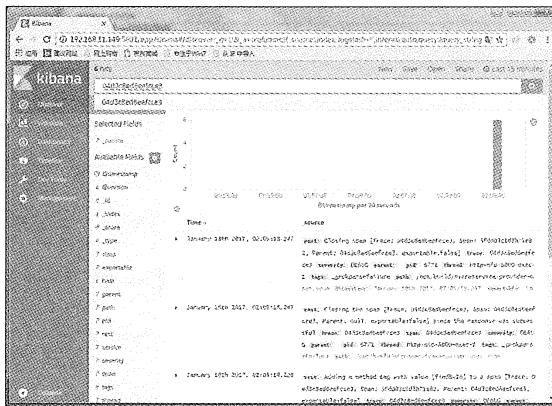


图 10-2 Kibana 首页

由图 10-2 可知, 已经可以正常使用 ELK 查询、分析跟踪日志。



再次强调, 必须将 spring.application.name 属性设置在 bootstrap.yml 中。

10.5 Spring Cloud Sleuth 与 Zipkin 配合使用

本节会将 Sleuth 与 Zipkin 配合使用。先来了解一下什么是 Zipkin。

10.5.1 Zipkin 简介

Zipkin 是 Twitter 开源的分布式跟踪系统，基于 Dapper 的论文设计而来。它的主要功能是收集系统的时序数据，从而追踪微服务架构的系统延时等问题。Zipkin 还提供了一个非常友好的界面，来帮助分析追踪数据。



Zipkin 官方网站：[http://zipkin.io/。](http://zipkin.io/)

10.5.2 编写 Zipkin Server

本节将编写一个 Zipkin Server。

1. 创建一个 ArtifactId 是 microservice-trace-zipkin-server 的 Maven 工程，并为项目添加以下依赖。

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>
```

2. 编写启动类，使用 @EnableZipkinServer 注解，声明一个 Zipkin Server。

```
@SpringBootApplication
@EnableZipkinServer
public class ZipkinServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ZipkinServerApplication.class, args);
    }
}
```

3. 编写配置文件，在 application.yml 中添加如下内容。

```
server:
  port: 9411
```



测试

1. 启动项目 microservice-trace-zipkin-server。
2. 访问 <http://localhost:9411/>，可以看到如图 10-3 的界面。

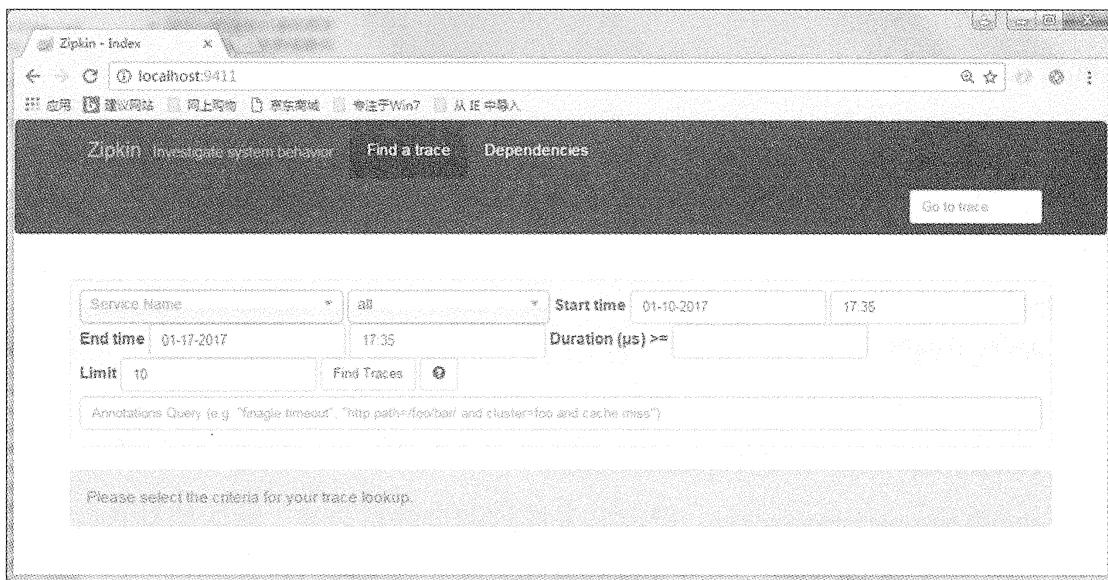


图 10-3 Zipkin Server 首页

简单讲解图中各个查询条件的含义：

- Service Name 表示服务名称，也就是各个微服务 `spring.application.name` 的值。
- 第二列表示 span 的名称，all 表示所有 span，也可选择指定 span。
- Start time、End time，分别用于指定起始时间和截止时间。
- Duration 表示持续时间，即 span 从创建到关闭所经历的时间。
- Limit 表示查询几条数据。类似于 MySQL 数据库中的 limit 关键词。
- Annotations Query，用于自定义查询条件。

10.5.3 微服务整合 Zipkin

本节来为微服务整合 Zipkin，以项目 `microservice-simple-provider-user-trace` 为例。

1. 复制项目 `microservice-simple-provider-user-trace`，将 `ArtifactId` 修改为 `microservice-simple-provider-user-trace-zipkin`。

2. 为项目添加以下依赖:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

3. 在配置文件 application.yml 中添加如下内容:

```
spring:
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      percentage: 1.0
```

其中:

spring.zipkin.base-url: 指定 Zipkin 的地址。

spring.sleuth.sampler.percentage: 指定需采样的请求的百分比, 默认值是 0.1, 即 10%。这是因为在分布式系统中, 数据量可能会非常大, 因此采样非常重要。

这样就为项目整合了 Zipkin。同理, 为项目 microservice-simple-consumer-movie-trace 整合 Zipkin, 记为 microservice-simple-consumer-movie-trace-zipkin。



测试

1. 启动项目 microservice-trace-zipkin-server。
2. 启动项目 microservice-simple-provider-user-trace-zipkin。
3. 启动项目 microservice-simple-consumer-movie-trace-zipkin。
4. 访问 <http://localhost:8010/user/1>, 可正常获得结果。
5. 访问 Zipkin Server 首页 <http://localhost:9411/>, 填入起始时间、结束时间等筛选条件后, 单击 Find a trace 按钮, 可看到 trace 列表, 如图 10-4 所示。