以我们之前对 hello-service 服务的调用为例，可以在配置文件中增加如下内容：

```
spring.cloud.loadbalancer.retry.enabled=true

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=10000

hello-service.ribbon.ConnectTimeout=250
hello-service.ribbon.ReadTimeout=1000
hello-service.ribbon.OkToRetryOnAllOperations=true
hello-service.ribbon.MaxAutoRetriesNextServer=2
hello-service.ribbon.MaxAutoRetries=1
```

其中各参数的配置说明如下所示。

- spring.cloud.loadbalancer.retry.enabled：该参数用来开启重试机制，它默认是关闭的。这里需要注意，官方文档中的配置参数少了 enabled。该参数的源码定义如下：

```
@ConfigurationProperties("spring.cloud.loadbalancer.retry")
public class LoadBalancerRetryProperties {
    private boolean enabled = false;
    ...
}
```

- hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds：断路器的超时时间需要大于 Ribbon 的超时时间，不然不会触发重试。
- hello-service.ribbon.ConnectTimeout：请求连接的超时时间。
- hello-service.ribbon.ReadTimeout：请求处理的超时时间。
- hello-service.ribbon.OkToRetryOnAllOperations：对所有操作请求都进行重试。
- hello-service.ribbon.MaxAutoRetriesNextServer：切换实例的重试次数。
- hello-service.ribbon.MaxAutoRetries：对当前实例的重试次数。

根据如上配置，当访问到故障请求的时候，它会再尝试访问一次当前实例（次数由 MaxAutoRetries 配置），如果不行，就换一个实例进行访问，如果还是不行，再换一次实例访问（更换次数由 MaxAutoRetriesNextServer 配置），如果依然不行，返回失败信息。

第 5 章

服务容错保护：Spring Cloud Hystrix

在微服务架构中，我们将系统拆分成了很多服务单元，各单元的应用间通过服务注册与订阅的方式互相依赖。由于每个单元都在不同的进程中运行，依赖通过远程调用的方式执行，这样就有可能因为网络原因或是依赖服务自身问题出现调用故障或延迟，而这些问题会直接导致调用方的对外服务也出现延迟，若此时调用方的请求不断增加，最后就会因等待出现故障的依赖方响应形成任务积压，最终导致自身服务的瘫痪。

举个例子，在一个电商网站中，我们可能会将系统拆分成用户、订单、库存、积分、评论等一系列服务单元。用户创建一个订单的时候，客户端将调用订单服务的创建订单接口，此时创建订单接口又会向库存服务来请求出货（判断是否有足够库存来出货）。此时若库存服务因自身处理逻辑等原因造成响应缓慢，会直接导致创建订单服务的线程被挂起，以等待库存申请服务的响应，在漫长的等待之后用户会因为请求库存失败而得到创建订单失败的结果。如果在高并发情况之下，因这些挂起的线程在等待库存服务的响应而未能释放，使得后续到来的创建订单请求被阻塞，最终导致订单服务也不可用。

在微服务架构中，存在着那么多的服务单元，若一个单元出现故障，就很容易因依赖关系而引发故障的蔓延，最终导致整个系统的瘫痪，这样的架构相较传统架构更加不稳定。为了解决这样的问题，产生了断路器等一系列的服务保护机制。

断路器模式源于 Martin Fowler 的 *Circuit Breaker* 一文。“断路器”本身是一种开关装置，用于在电路上保护线路过载，当线路中有电器发生短路时，“断路器”能够及时切断故障电路，防止发生过载、发热甚至起火等严重后果。

在分布式架构中，断路器模式的作用也是类似的，当某个服务单元发生故障（类似用

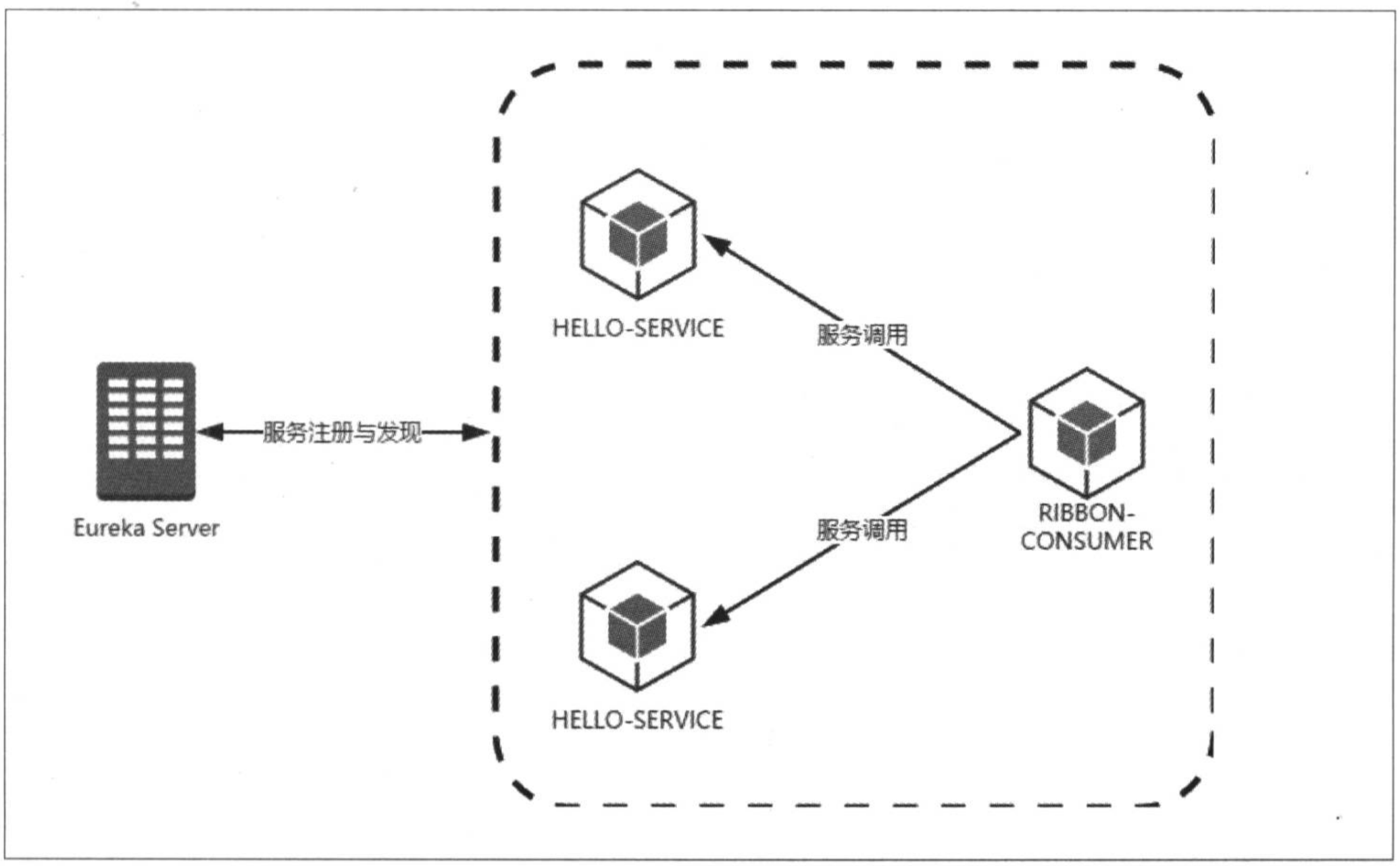
电器发生短路）之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个错误响应，而不是长时间的等待。这样就不会使得线程因调用故障服务被长时间占用不释放，避免了故障在分布式系统中的蔓延。

针对上述问题，Spring Cloud Hystrix 实现了断路器、线程隔离等一系列服务保护功能。它也是基于 Netflix 的开源框架 Hystrix 实现的，该框架的目标在于通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。Hystrix 具备服务降级、服务熔断、线程和信号隔离、请求缓存、请求合并以及服务监控等强大功能。

接下来，我们就从一个简单示例开始对 Spring Cloud Hystrix 的学习与使用。

快速入门

在开始使用 Spring Cloud Hystrix 实现断路器之前，我们先用之前实现的一些内容作为基础，构建一个如下图架构所示的服务调用关系。



我们在这里需要启动的工程有如下一些。

- `eureka-server` 工程：服务注册中心，端口为 1111。
- `hello-service` 工程：`HELLO-SERVICE` 的服务单元，两个实例启动端口分别为 8081 和 8082。

- ribbon-consume 工程：使用 Ribbon 实现的服务消费者，端口为 9000。

在未加入断路器之前，关闭 8081 的实例，发送 GET 请求到 http://localhost:9000/ribbon-consumer，可以获得下面的输出：

```
{
  "timestamp": 1473992080343,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "org.springframework.web.client.ResourceAccessException",
  "message": "I/O error on GET request for \"http://HELLO-SERVICE/hello\": 
Connection refused: connect; nested exception is java.net.ConnectException: Connection 
refused: connect",
  "path": "/ribbon-consumer"
}
```

下面我们开始引入 Spring Cloud Hystrix。

- 在 ribbon-consumer 工程的 pom.xml 的 dependency 节点中引入 spring-cloud-starter-hystrix 依赖：

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

- 在 ribbon-consumer 工程的主类 ConsumerApplication 中使用@EnableCircuitBreaker注解开启断路器功能：

```
@EnableCircuitBreaker
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {

    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

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

}
```

注意：这里还可以使用 Spring Cloud 应用中的`@SpringCloudApplication`注解来修饰应用主类，该注解的具体定义如下所示。可以看到，该注解中包含了上述我们所引用的三个注解，这也意味着一个 Spring Cloud 标准应用应包含服务发现以及断路器。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public @interface SpringCloudApplication {
}
```

- 改造服务消费方式，新增 `HelloService` 类，注入 `RestTemplate` 实例。然后，将在 `ConsumerController` 中对 `RestTemplate` 的使用迁移到 `helloService` 函数中，最后，在 `helloService` 函数上增加`@HystrixCommand`注解来指定回调方法：

```
@Service
public class HelloService {

    @Autowired
    RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "helloFallback")
    public String helloService() {
        return restTemplate.getForEntity("http://HELLO-SERVICE/hello",
String.class).getBody();
    }

    public String helloFallback() {
        return "error";
    }

}
```

- 修改 `ConsumerController` 类，注入上面实现的 `HelloService` 实例，并在 `helloConsumer` 中进行调用：

```
@RestController
public class ConsumerController {
```

```
    @Autowired
    HelloService helloService;

    @RequestMapping(value = "/ribbon-consumer", method = RequestMethod.GET)
    public String helloConsumer() {
        return helloService.helloService();
    }

}
```

下面，我们来验证一下通过断路器实现的服务回调逻辑，重新启动之前关闭的 8081 端口的 Hello-Service，确保此时服务注册中心、两个 Hello-Service 以及 RIBBON-CONSUMER 均已启动，访问 http://localhost:9000/ribbon-consumer 可以轮询两个 HELLO-SERVICE 并返回一些文字信息。此时我们继续断开 8081 的 HELLO-SERVICE，然后访问 http://localhost:9000/ribbon-consumer，当轮询到 8081 服务端时，输出内容为 error，不再是之前的错误内容，Hystrix 的服务回调生效。除了通过断开具体的服务实例来模拟某个节点无法访问的情况之外，我们还可以模拟一下服务阻塞（长时间未响应）的情况。我们对 HELLO-SERVICE 的/hello 接口做一些修改，具体如下：

```
    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String hello() throws Exception {
        ServiceInstance instance = client.getLocalServiceInstance();
        // 让处理线程等待几秒钟
        int sleepTime = new Random().nextInt(3000);
        logger.info("sleepTime:" + sleepTime);
        Thread.sleep(sleepTime);

        logger.info("/hello, host:" + instance.getHost() + ", service_id:" +
instance.getServiceId());
        return "Hello World";
    }
```

通过 Thread.sleep()函数可让/hello 接口的处理线程不是马上返回内容，而是在阻塞几秒之后才返回内容。由于 Hystrix 默认超时时间为 2000 毫秒，所以这里采用了 0 至 3000 的随机数以让处理过程有一定概率发生超时来触发断路器。为了更精准地观察断路器的触发，在消费者调用函数中做一些时间记录，具体如下：

```
    @HystrixCommand(fallbackMethod = "helloFallback", commandKey = "helloKey")
    public String hello() {
        long start = System.currentTimeMillis();

        // 消费服务的逻辑
        ...
```

```
        long end = System.currentTimeMillis();

        logger.info("Spend time : " + (end - start) );
        return result.toString();
    }
```

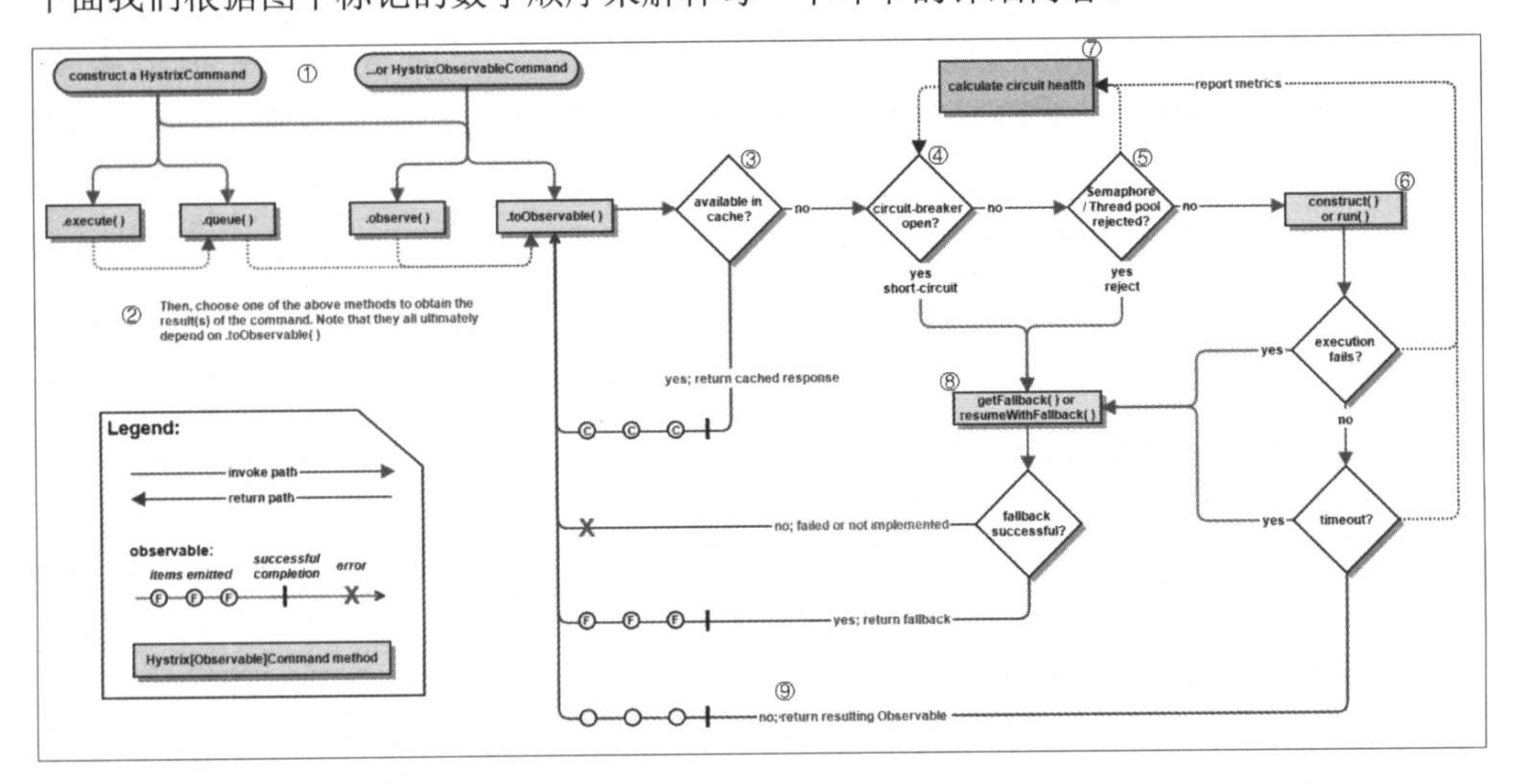
重新启动 HELLO-SERVICE 和 RIBBON-CONSUMER 的实例，连续访问 http://localhost:9000/ribbon-consumer 几次，我们可以观察到，当 RIBBON-CONSUMER 的控制台中输出的 Spend time 大于 2000 的时候，就会返回 error，即服务消费者因调用的服务超时从而触发熔断请求，并调用回调逻辑返回结果。

原理分析

通过上面的快速入门示例，我们对 Hystrix 的使用场景和使用方法已经有了一个基础的认识。接下来我们通过解读 Netflix Hystrix 官方的流程图来详细了解一下：当一个请求调用了相关服务依赖之后 Hystrix 是如何工作的（即如上例中所示，当访问了 http://localhost:9000/ribbon-consumer 请求之后，在 RIBBON-CONSUMER 中是如何处理的）。

工作流程

下面我们根据图中标记的数字顺序来解释每一个环节的详细内容。



1. 创建 HystrixCommand 或 HystrixObservableCommand 对象

首先，构建一个 HystrixCommand 或是 HystrixObservableCommand 对象，用

来表示对依赖服务的操作请求，同时传递所有需要的参数。从其命名中我们就能知道它采用了“命令模式”来实现对服务调用操作的封装。而这两个 Command 对象分别针对不同的应用场景。

- HystrixCommand：用在依赖的服务返回单个操作结果的时候。
- HystrixObservableCommand：用在依赖的服务返回多个操作结果的时候。

命令模式，将来自客户端的请求封装成一个对象，从而让你可以使用不同的请求对客户端进行参数化。它可以被用于实现“行为请求者”与“行为实现者”的解耦，以便使两者可以适应变化。下面的示例是对命令模式的简单实现：

```
// 接收者
public class Receiver {
    public void action(){
        // 真正的业务逻辑
    }
}
// 抽象命令
interface Command {
    void execute();
}
// 具体命令实现
public class ConcreteCommand implements Command {
    private Receiver receiver;
    public ConcreteCommand(Receiver receiver){
        this.receiver = receiver;
    }
    public void execute() {
        this.receiver.action();
    }
}
// 客户端调用者
public class Invoker {
    private Command command;
  public void setCommand(Command command) {
    this.command = command;
    }
  public void action(){
        this.command.execute();
  }
}
public class Client {
    public static void main(String[] args){
```

```
        Receiver receiver = new Receiver();
        Command command = new ConcreteCommand(receiver);
        Invoker invoker = new Invoker();
        invoker.setCommand(command);
        invoker.action(); //客户端通过调用者来执行命令
    }
}
```

从代码中，我们可以看到这样几个对象。

- Receiver：接收者，它知道如何处理具体的业务逻辑。
- Command：抽象命令，它定义了一个命令对象应具备的一系列命令操作，比如 execute()、undo()、redo()等。当命令操作被调用的时候就会触发接收者去做具体命令对应的业务逻辑。
- ConcreteCommand：具体的命令实现，在这里它绑定了命令操作与接收者之间的关系，execute()命令的实现委托给了 Receiver 的 action()函数。
- Invoker：调用者，它持有一个命令对象，并且可以在需要的时候通过命令对象完成具体的业务逻辑。

从上面的示例中，我们可以看到，调用者 Invoker 与操作者 Receiver 通过 Command 命令接口实现了解耦。对于调用者来说，我们可以为其注入多个命令操作，比如新建文件、复制文件、删除文件这样三个操作，调用者只需在需要的时候直接调用即可，而不需要知道这些操作命令实际是如何实现的。而在这里所提到的 HystrixCommand 和 HystrixObservableCommand 则是在 Hystrix 中对 Command 的进一步抽象定义。在后续的内容中，会逐步展开介绍它的部分内容来帮助理解其运作原理。

从上面的示例中我们也可以发现，Invoker 和 Receiver 的关系非常类似于“请求-响应”模式，所以它比较适用于实现记录日志、撤销操作、队列请求等。

在下面这些情况下应考虑使用命令模式。

- 使用命令模式作为“回调（CallBack）”在面向对象系统中的替代。“CallBack”讲的便是先将一个函数登记上，然后在以后调用此函数。
- 需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命期。换言之，原先的请求发出者可能已经不在了，而命令对象本身仍然是活动的。这时命令的接收者可以是在本地，也可以在网络的另外一个地址。命令对象可以在序列化之后传送到另外一台机器上去。
- 系统需要支持命令的撤销。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用 undo()方法，把命令所产生的效果撤销掉。命令对象还可以提供 redo()方法，以供客户端在需要时再重新实施命令效果。

- 如果要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志读回所有的数据更新命令，重新调用 Execute()方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。

2. 命令执行

从图中我们可以看到一共存在 4 种命令的执行方式，而 Hystrix 在执行时会根据创建的 Command 对象以及具体的情况来选择一个执行。其中 HystrixCommand 实现了下面两个执行方式。

- execute()：同步执行，从依赖的服务返回一个单一的结果对象，或是在发生错误的时候抛出异常。
- queue()：异步执行，直接返回一个 Future 对象，其中包含了服务执行结束时要返回的单一结果对象。

```
R value = command.execute();
Future<R> fValue = command.queue();
```

而 HystrixObservableCommand 实现了另外两种执行方式。

- observe()：返回 Observable 对象，它代表了操作的多个结果，它是一个 Hot Observable。
- toObservable()：同样会返回 Observable 对象，也代表了操作的多个结果，但它返回的是一个 Cold Observable。

```
Observable<R> ohValue = command.observe();
Observable<R> ocValue = command.toObservable();
```

在 Hystrix 的底层实现中大量地使用了 RxJava，为了更容易地理解后续内容，在这里对 RxJava 的观察者-订阅者模式做一个简单的入门介绍。

上面我们所提到的 Observable 对象就是 RxJava 中的核心内容之一，可以把它理解为“事件源”或是“被观察者”，与其对应的 Subscriber 对象，可以理解为“订阅者”或是“观察者”。这两个对象是 RxJava 响应式编程的重要组成部分。

- Observable 用来向订阅者 Subscriber 对象发布事件，Subscriber 对象则在接收到事件后对其进行处理，而在这里所指的事件通常就是对依赖服务的调用。
- 一个 Observable 可以发出多个事件，直到结束或是发生异常。
- Observable 对象每发出一个事件，就会调用对应观察者 Subscriber 对象的 onNext()方法。
- 每一个 Observable 的执行，最后一定会通过调用 Subscriber.onCompleted()或者 Subscriber.onError()来结束该事件的操作流。

下面我们通过一个简单的例子来直观理解一下 Observable 与 Subscribers：

```
    // 创建事件源 observable
    Observable<String> observable = Observable.create(new Observable.OnSubscribe
<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            subscriber.onNext("Hello RxJava");
            subscriber.onNext("I am 程序猿 DD");
            subscriber.onCompleted();
        }
    });

    // 创建订阅者 subscriber
    Subscriber<String> subscriber = new Subscriber<String>() {
        @Override
        public void onCompleted() {
        }

        @Override
        public void onError(Throwable e) {
        }
        @Override
        public void onNext(String s) {
            System.out.println("Subscriber : " + s);
        }
    };

    // 订阅
    observable.subscribe(subscriber);
```

在该示例中，创建了一个简单的事件源 observable，一个对事件传递内容输出的订阅者 subscriber，通过 observable.subscribe(subscriber) 来触发事件的发布。

在这里我们对于事件源 observable 提到了两个不同的概念：Hot Observable 和 Cold Observable，分别对应了上面 command.observe() 和 command.toObservable() 的返回对象。其中 Hot Observable，它不论“事件源”是否有“订阅者”，都会在创建后对事件进行发布，所以对于 Hot Observable 的每一个“订阅者”都有可能是从“事件源”的中途开始的，并可能只是看到了整个操作的局部过程。而 Cold Observable 在没有“订阅者”的时候并不会发布事件，而是进行等待，直到有“订阅者”之后才发布事件，所以对于 Cold Observable 的订阅者，它可以保证从一开始看到整个操作的全部过程。

大家从表面上可能会认为只是在 HystrixObservableCommand 中使用了 RxJava，

然而实际上 execute()、queue() 也都使用了 RxJava 来实现。从下面的源码中我们可以看到：

- execute() 是通过 queue() 返回的异步对象 Future<R>的 get() 方法来实现同步执行的。该方法会等待任务执行结束，然后获得 R 类型的结果进行返回。
- queue() 则是通过 toObservable() 来获得一个 Cold Observable，并且通过 toBlocking() 将该 Observable 转换成 BlockingObservable，它可以把数据以阻塞的方式发射出来。而 toFuture 方法则是把 BlockingObservable 转换为一个 Future，该方法只是创建一个 Future 返回，并不会阻塞，这使得消费者可以自己决定如何处理异步操作。而 execute() 就是直接使用了 queue() 返回的 Future 中的阻塞方法 get() 来实现同步操作的。同时通过这种方式转换的 Future 要求 Observable 只发射一个数据，所以这两个实现都只能返回单一结果。

```
public R execute() {
    try {
        return queue().get();
    } catch (Exception e) {
        throw decomposeException(e);
    }
}

public Future<R> queue() {
    final Observable<R> o = toObservable();
    final Future<R> f = o.toBlocking().toFuture();

    if (f.isDone()) {
        // 处理立即抛出的错误
        ...
    }
    return f;
}
```

3. 结果是否被缓存

若当前命令的请求缓存功能是被启用的，并且该命令缓存命中，那么缓存的结果会立即以 Observable 对象的形式返回。

4. 断路器是否打开

在命令结果没有缓存命中的时候，Hystrix 在执行命令前需要检查断路器是否为打开状态：

- 如果断路器是打开的，那么 Hystrix 不会执行命令，而是转接到 fallback 处理逻辑（对应下面第 8 步）。

- 如果断路器是关闭的，那么 Hystrix 跳到第 5 步，检查是否有可用资源来执行命令。

关于断路器的具体实现细节，后续会做更加详细的分析。

5. 线程池/请求队列/信号量是否占满

如果与命令相关的线程池和请求队列，或者信号量（不使用线程池的时候）已经被占满，那么 Hystrix 也不会执行命令，而是转接到 fallback 处理逻辑（对应下面第 8 步）。

需要注意的是，这里 Hystrix 所判断的线程池并非容器的线程池，而是每个依赖服务的专有线程池。Hystrix 为了保证不会因为某个依赖服务的问题影响到其他依赖服务而采用了“舱壁模式”（Bulkhead Pattern）来隔离每个依赖的服务。关于依赖服务的隔离与线程池相关的内容见后续详细介绍。

6. `HystrixObservableCommand.construct()` 或 `HystrixCommand.run()`

Hystrix 会根据我们编写的方法来决定采取什么样的方式去请求依赖服务。

- `HystrixCommand.run()`：返回一个单一的结果，或者抛出异常。
- `HystrixObservableCommand.construct()`：返回一个 `Observable` 对象来发射多个结果，或通过 `onError` 发送错误通知。

如果 `run()` 或 `construct()` 方法的执行时间超过了命令设置的超时阈值，当前处理线程将会抛出一个 `TimeoutException`（如果该命令不在其自身的线程中执行，则会通过单独的计时线程来抛出）。在这种情况下，Hystrix 会转接到 fallback 处理逻辑（第 8 步）。同时，如果当前命令没有被取消或中断，那么它最终会忽略 `run()` 或者 `construct()` 方法的返回。

如果命令没有抛出异常并返回了结果，那么 Hystrix 在记录一些日志并采集监控报告之后将该结果返回。在使用 `run()` 的情况下，Hystrix 会返回一个 `Observable`，它发射单个结果并产生 `onCompleted` 的结束通知；而在使用 `construct()` 的情况下，Hystrix 会直接返回该方法产生的 `Observable` 对象。

7. 计算断路器的健康度

Hystrix 会将“成功”、“失败”、“拒绝”、“超时”等信息报告给断路器，而断路器会维护一组计数器来统计这些数据。

断路器会使用这些统计数据来决定是否要将断路器打开，来对某个依赖服务的请求进行“熔断/短路”，直到恢复期结束。若在恢复期结束后，根据统计数据判断如果还是未达到健康指标，就再次“熔断/短路”。

8. fallback 处理

当命令执行失败的时候，Hystrix 会进入 fallback 尝试回退处理，我们通常也称该操作为“服务降级”。而能够引起服务降级处理的情况有下面几种：

- 第 4 步，当前命令处于“熔断/短路”状态，断路器是打开的时候。
- 第 5 步，当前命令的线程池、请求队列或者信号量被占满的时候。
- 第 6 步，`HystrixObservableCommand.construct()` 或 `HystrixCommand.run()` 抛出异常的时候。

在服务降级逻辑中，我们需要实现一个通用的响应结果，并且该结果的处理逻辑应当是从缓存或是根据一些静态逻辑来获取，而不是依赖网络请求获取。如果一定要在降级逻辑中包含网络请求，那么该请求也必须被包装在 `HystrixCommand` 或是 `HystrixObservableCommand` 中，从而形成级联的降级策略，而最终的降级逻辑一定不是一个依赖网络请求的处理，而是一个能够稳定地返回结果的处理逻辑。

在 `HystrixCommand` 和 `HystrixObservableCommand` 中实现降级逻辑时还略有不同：

- 当使用 `HystrixCommand` 的时候，通过实现 `HystrixCommand.getFallback()` 来实现服务降级逻辑。
- 当使用 `HystrixObservableCommand` 的时候，通过 `HystrixObservable-Command.resumeWithFallback()` 实现服务降级逻辑，该方法会返回一个 `Observable` 对象来发射一个或多个降级结果。

当命令的降级逻辑返回结果之后，Hystrix 就将该结果返回给调用者。当使用 `HystrixCommand.getFallback()` 的时候，它会返回一个 `Observable` 对象，该对象会发射 `getFallback()` 的处理结果。而使用 `HystrixObservableCommand.resumeWithFallback()` 实现的时候，它会将 `Observable` 对象直接返回。

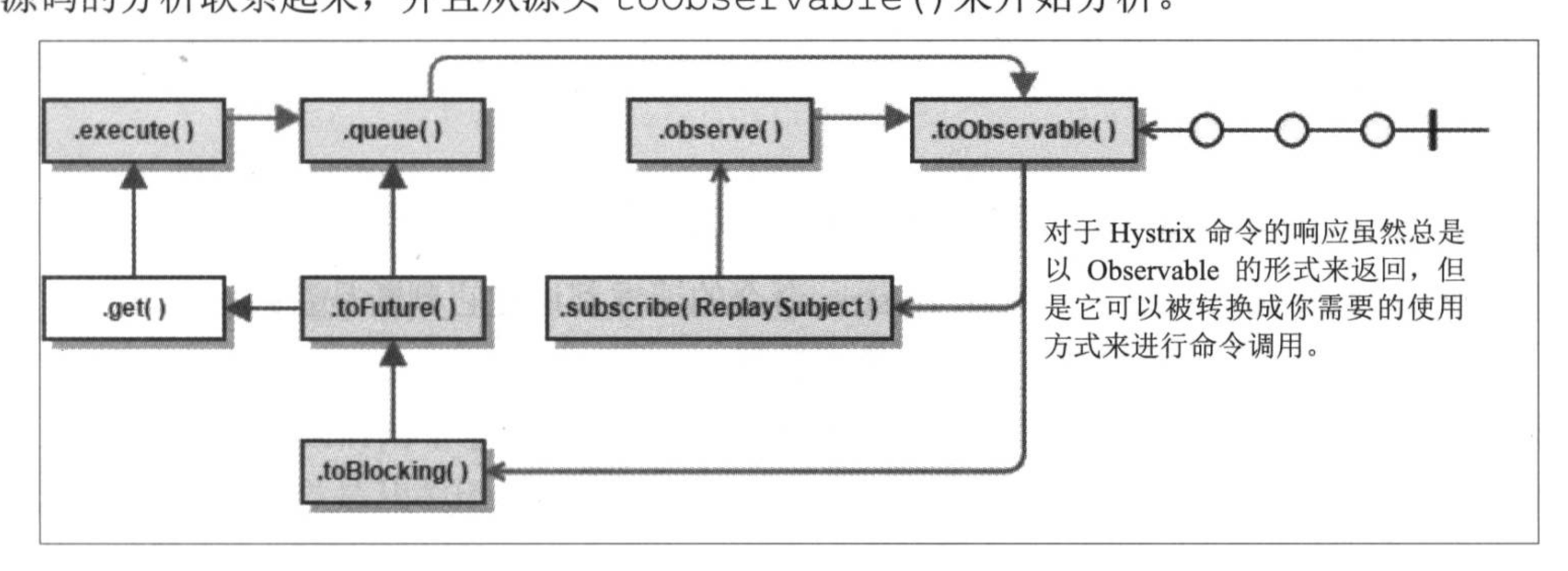
如果我们没有为命令实现降级逻辑或者降级处理逻辑中抛出了异常，Hystrix 依然会返回一个 `Observable` 对象，但是它不会发射任何结果数据，而是通过 `onError` 方法通知命令立即中断请求，并通过 `onError()` 方法将引起命令失败的异常发送给调用者。实现一个有可能失败的降级逻辑是一种非常糟糕的做法，我们应该在实现降级策略时尽可能避免失败的情况。

当然完全不可能出现失败的完美策略是不存在的，如果降级执行发现失败的时候，Hystrix 会根据不同的执行方法做出不同的处理。

- execute()：抛出异常。
- queue()：正常返回 Future 对象，但是当调用 get()来获取结果的时候会抛出异常。
- observe()：正常返回 Observable 对象，当订阅它的时候，将立即通过调用订阅者的 onError 方法来通知中止请求。
- toObservable()：正常返回 Observable 对象，当订阅它的时候，将通过调用订阅者的 onError 方法来通知中止请求。

9. 返回成功的响应

当 Hystrix 命令执行成功之后，它会将处理结果直接返回或是以 Observable 的形式返回。而具体以哪种方式返回取决于之前第 2 步中我们所提到的对命令的 4 种不同执行方式，下图中总结了这 4 种调用方式之间的依赖关系。我们可以将此图与在第 2 步中对前两者源码的分析联系起来，并且从源头 toObservable()来开始分析。



- toObservable()：返回最原始的 Observable，必须通过订阅它才会真正触发命令的执行流程。
- observe()：在 toObservable()产生原始 Observable 之后立即订阅它，让命令能够马上开始异步执行，并返回一个 Observable 对象，当调用它的 subscribe 时，将重新产生结果和通知给订阅者。
- queue()：将 toObservable()产生的原始 Observable 通过 toBlocking()方法转换成 BlockingObservable 对象，并调用它的 toFuture()方法返回异步的 Future 对象。
- execute()：在 queue()产生异步结果 Future 对象之后，通过调用 get()方法阻塞并等待结果的返回。

断路器原理

断路器在 HystrixCommand 和 HystrixObservableCommand 执行过程中起到了举足轻重的作用，它是 Hystrix 的核心部件。那么断路器是如何决策熔断和记录信息的呢？

我们先来看看断路器 HystrixCircuitBreaker 的定义：

```
public interface HystrixCircuitBreaker {

  public static class Factory {...}

  static class HystrixCircuitBreakerImpl implements HystrixCircuitBreaker {...}

  static class NoOpCircuitBreaker implements HystrixCircuitBreaker {...}

  public boolean allowRequest();

  public boolean isOpen();

  void markSuccess();
}
```

可以看到它的接口定义并不复杂，主要定义了三个断路器的抽象方法。

- allowRequest()：每个 Hystrix 命令的请求都通过它判断是否被执行。
- isOpen()：返回当前断路器是否打开。
- markSuccess()：用来闭合断路器。

另外还有三个静态类。

- 静态类 Factory 中维护了一个 Hystrix 命令与 HystrixCircuitBreaker 的关系集合：ConcurrentHashMap<String, HystrixCircuitBreaker> circuitBreakersByCommand，其中 String 类型的 key 通过 HystrixCommandKey 定义，每一个 Hystrix 命令需要有一个 key 来标识，同时一个 Hystrix 命令也会在该集合中找到它对应的断路器 HystrixCircuitBreaker 实例。
- 静态类 NoOpCircuitBreaker 定义了一个什么都不做的断路器实现，它允许所有请求，并且断路器状态始终闭合。
- 静态类 HystrixCircuitBreakerImpl 是断路器接口 HystrixCircuitBreaker 的实现类，在该类中定义了断路器的 4 个核心对象。
  - HystrixCommandProperties properties：断路器对应 HystrixCommand 实例的属性对象，它的详细内容我们将在后续章节做具体的介绍。

- HystrixCommandMetrics metrics：用来让 HystrixCommand 记录各类度量指标的对象。
- AtomicBoolean circuitOpen：断路器是否打开的标志，默认为 false。
- AtomicLong circuitOpenedOrLastTestedTime：断路器打开或是上一次测试的时间戳。

HystrixCircuitBreakerImpl 对 HystrixCircuitBreaker 接口的各个方法实现如下所示。

- isOpen()：判断断路器的打开/关闭状态。详细逻辑如下所示。
  - 如果断路器打开标识为 true，则直接返回 true，表示断路器处于打开状态。否则，就从度量指标对象 metrics 中获取 HealthCounts 统计对象做进一步判断（该对象记录了一个滚动时间窗内的请求信息快照，默认时间窗为 10 秒）。
    - 如果它的请求总数（QPS）在预设的阈值范围内就返回 false，表示断路器处于未打开状态。该阈值的配置参数为 circuitBreakerRequestVolumeThreshold，默认值为 20。
    - 如果错误百分比在阈值范围内就返回 false，表示断路器处于未打开状态。该阈值的配置参数为 circuitBreakerErrorThresholdPercentage，默认值为 50。
    - 如果上面的两个条件都不满足，则将断路器设置为打开状态（熔断/短路）。同时，如果是从关闭状态切换到打开状态的话，就将当前时间记录到上面提到的 circuitOpenedOrLastTestedTime 对象中。

```
public boolean isOpen() {
    if (circuitOpen.get()) {
        return true;
    }
    HealthCounts health = metrics.getHealthCounts();
    if (health.getTotalRequests() <
properties.circuitBreakerRequestVolumeThreshold().get()) {
        return false;
    }
    if (health.getErrorPercentage() <
properties.circuitBreakerErrorThresholdPercentage().get()) {
        return false;
    } else {
```

```
        if (circuitOpen.compareAndSet(false, true)) {
            circuitOpenedOrLastTestedTime.set(System.currentTimeMillis());
            return true;
        } else {
            return true;
        }
    }
}
```

- allowRequest()：判断请求是否被允许，这个实现非常简单。先根据配置对象 properties 中的断路器判断强制打开或关闭属性是否被设置。如果强制打开，就直接返回 false，拒绝请求。如果强制关闭，它会允许所有请求，但是同时也会调用 isOpen()来执行断路器的计算逻辑，用来模拟断路器打开/关闭的行为。在默认情况下，断路器并不会进入这两个强制打开或关闭的分支中去，而是通过!isOpen() || allowSingleTest()来判断是否允许请求访问。!isOpen()之前已经介绍过，用来判断和计算当前断路器是否打开，如果是断开状态就允许请求。那么 allowSingleTest()是用来做什么的呢？

```
@Override
public boolean allowRequest() {
    if (properties.circuitBreakerForceOpen().get()) {
        return false;
    }
    if (properties.circuitBreakerForceClosed().get()) {
        isOpen();
        return true;
    }
    return !isOpen() || allowSingleTest();
}
```

从 allowSingleTest()的实现中我们可以看到，这里使用了在 isOpen()函数中当断路器从闭合到打开时候所记录的时间戳。当断路器在打开状态的时候，这里会判断断开时的时间戳 + 配置中的 circuitBreakerSleepWindowInMilliseconds 时间是否小于当前时间，是的话，就将当前时间更新到记录断路器打开的时间对象 circuitOpenedOrLastTestedTime 中，并且允许此次请求。简单地说，通过 circuitBreakerSleepWindowInMilliseconds 属性设置了一个断路器打开之后的休眠时间（默认为 5 秒），在该休眠时间到达之后，将再次允许请求尝试访问，

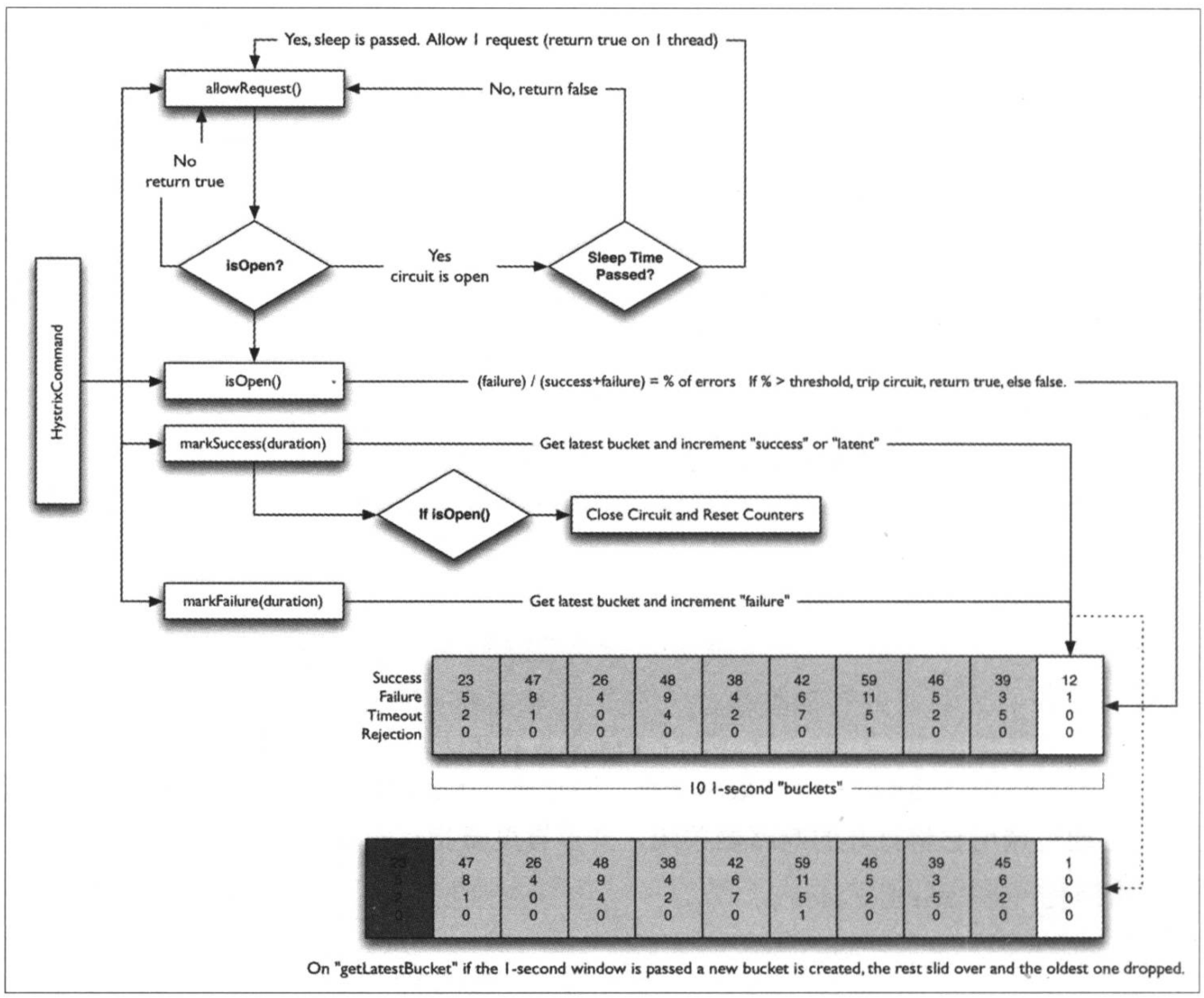
此时断路器处于“半开”状态，若此时请求继续失败，断路器又进入打开状态，并继续等待下一个休眠窗口过去之后再次尝试；若请求成功，则将断路器重新置于关闭状态。所以通过 allowSingleTest() 与 isOpen() 方法的配合，实现了断路器打开和关闭状态的切换。

```
public boolean allowSingleTest() {
    long timeCircuitOpenedOrWasLastTested = circuitOpenedOrLastTestedTime.get();
    if (circuitOpen.get() && System.currentTimeMillis() >
      timeCircuitOpenedOrWasLastTested +
properties.circuitBreakerSleepWindowInMilliseconds().get()) {
        if (circuitOpenedOrLastTestedTime.compareAndSet(
          timeCircuitOpenedOrWasLastTested, System.currentTimeMillis())) {
            return true;
        }
    }
    return false;
}
```

- markSuccess()：该函数用来在“半开路”状态时使用。若 Hystrix 命令调用成功，通过调用它将打开的断路器关闭，并重置度量指标对象。

```
public void markSuccess() {
    if (circuitOpen.get()) {
        if (circuitOpen.compareAndSet(true, false)) {
            metrics.resetStream();
        }
    }
}
```

下图是 Netflix Hystrix 官方文档中关于断路器的详细执行逻辑，可以帮助我们理解上面的分析内容。



依赖隔离

“舱壁模式”对于熟悉 Docker 的读者一定不陌生，Docker 通过“舱壁模式”实现进程的隔离，使得容器与容器之间不会互相影响。而 Hystrix 则使用该模式实现线程池的隔离，它会为每一个依赖服务创建一个独立的线程池，这样就算某个依赖服务出现延迟过高的情况，也只是对该依赖服务的调用产生影响，而不会拖慢其他的依赖服务。

通过实现对依赖服务的线程池隔离，可以带来如下优势：

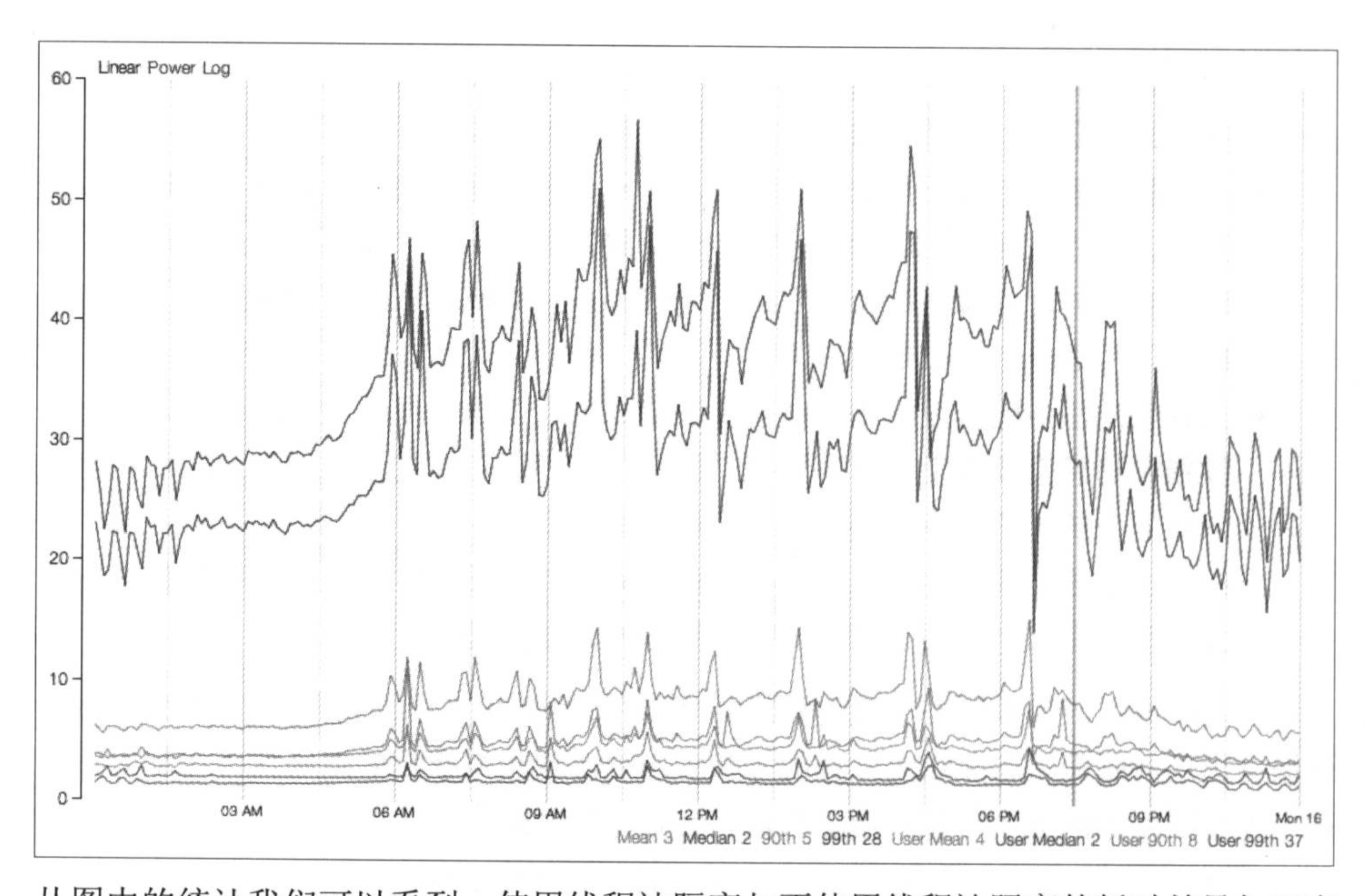
- 应用自身得到完全保护，不会受不可控的依赖服务影响。即便给依赖服务分配的线程池被填满，也不会影响应用自身的其余部分。
- 可以有效降低接入新服务的风险。如果新服务接入后运行不稳定或存在问题，完全不会影响应用其他的请求。

- 当依赖的服务从失效恢复正常后，它的线程池会被清理并且能够马上恢复健康的服务，相比之下，容器级别的清理恢复速度要慢得多。
- 当依赖的服务出现配置错误的时候，线程池会快速反映出此问题（通过失败次数、延迟、超时、拒绝等指标的增加情况）。同时，我们可以在不影响应用功能的情况下通过实时的动态属性刷新（后续会通过 Spring Cloud Config 与 Spring Cloud Bus 的联合使用来介绍）来处理它。
- 当依赖的服务因实现机制调整等原因造成其性能出现很大变化的时候，线程池的监控指标信息会反映出这样的变化。同时，我们也可以通过实时动态刷新自身应用对依赖服务的阈值进行调整以适应依赖方的改变。
- 除了上面通过线程池隔离服务发挥的优点之外，每个专有线程池都提供了内置的并发实现，可以利用它为同步的依赖服务构建异步访问。

总之，通过对依赖服务实现线程池隔离，可让我们的应用更加健壮，不会因为个别依赖服务出现问题而引起非相关服务的异常。同时，也使得我们的应用变得更加灵活，可以在不停止服务的情况下，配合动态配置刷新实现性能配置上的调整。

虽然线程池隔离的方案带来如此多的好处，但是很多使用者可能会担心为每一个依赖服务都分配一个线程池是否会过多地增加系统的负载和开销。对于这一点，使用者不用过于担心，因为这些顾虑也是大部分工程师们会考虑到的，Netflix 在设计 Hystrix 的时候，认为线程池上的开销相对于隔离所带来的好处是无法比拟的。同时，Netflix 也针对线程池的开销做了相关的测试，以用结果打消 Hystrix 实现对性能影响的顾虑。

下图是 Netflix Hystrix 官方提供的一个 Hystrix 命令的性能监控图，该命令以每秒 60 个请求的速度（QPS）对一个单服务实例进行访问，该服务实例每秒运行的线程数峰值为 350 个。



从图中的统计我们可以看到，使用线程池隔离与不使用线程池隔离的耗时差异如下表所示：

比较情况	未使用线程池隔离	使用了线程池隔离	耗时差距
中位数	2ms	2ms	2ms
90 百分位	5ms	8ms	3ms
99 百分位	28ms	37ms	9ms

在 99%的情况下，使用线程池隔离的延迟有 9ms，对于大多数需求来说这样的消耗是微乎其微的，更何况可为系统在稳定性和灵活性上带来巨大的提升。虽然对于大部分的请求我们可以忽略线程池的额外开销，而对于小部分延迟本身就非常小的请求（可能只需要 1ms），那么 9ms 的延迟开销还是非常昂贵的。实际上 Hystrix 也为此设计了另外的解决方案：信号量。

在 Hystrix 中除了可使用线程池之外，还可以使用信号量来控制单个依赖服务的并发度，信号量的开销远比线程池的开销小，但是它不能设置超时和实现异步访问。所以，只有在依赖服务是足够可靠的情况下才使用信号量。在 `HystrixCommand` 和 `HystrixObservableCommand` 中有两处支持信号量的使用。

- **命令执行：**如果将隔离策略参数 `execution.isolation.strategy` 设置为 `SEMAPHORE`，Hystrix 会使用信号量替代线程池来控制依赖服务的并发。

- **降级逻辑：**当 Hystrix 尝试降级逻辑时，它会在调用线程中使用信号量。

信号量的默认值为 10，我们也可以通过动态刷新配置的方式来控制并发线程的数量。对于信号量大小的估算方法与线程池并发度的估算类似。仅访问内存数据的请求一般耗时在 1ms 以内，性能可以达到 5000rps（rps 指每秒的请求数），这样级别的请求可以将信号量设置为 1 或者 2，我们可以按此标准并根据实际请求耗时来设置信号量。

使用详解

在“快速入门”一节中我们已经使用过 Hystrix 中的核心注解`@HystrixCommand`，通过它创建了`HystrixCommand`的实现，同时利用`fallback`属性指定了服务降级的实现方法。然而这些还只是 Hystrix 使用的一小部分，在实现一个大型分布式系统时，往往还需要更多高级的配置功能。接下来我们将详细介绍 Hystrix 各接口和注解的使用方法。

创建请求命令

Hystrix 命令就是我们之前所说的`HystrixCommand`，它用来封装具体的依赖服务调用逻辑。

我们可以通过继承的方式来实现，比如：

```
public class UserCommand extends HystrixCommand<User> {

    private RestTemplate restTemplate;
    private Long id;

    public UserCommand(Setter setter, RestTemplate restTemplate, Long id) {
        super(setter);
        this.restTemplate = restTemplate;
        this.id = id;
    }

    @Override
    protected User run() {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

}
```

通过上面实现的`UserCommand`，我们既可以实现请求的同步执行也可以实现异步执行。

- **同步执行：** User u = new UserCommand(restTemplate, 1L).execute();。
- **异步执行：** Future<User> futureUser = new UserCommand(restTemplate, 1L).queue();。异步执行的时候，可以通过对返回的 futureUser 调用 get 方法来获取结果。

另外，也可以通过@HystrixCommand 注解来更为优雅地实现 Hystrix 命令的定义，比如：

```
public class UserService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

}
```

虽然@HystrixCommand 注解可以非常优雅地定义 Hystrix 命令的实现，但是如上定义的 getUserById 方式只是同步执行的实现，若要实现异步执行则还需另外定义，比如：

```
@HystrixCommand
public Future<User> getUserByIdAsync(final String id) {
    return new AsyncResult<User>() {
        @Override
        public User invoke() {
            return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
        }
    };
}
```

除了传统的同步执行与异步执行之外，我们还可以将 HystrixCommand 通过 Observable 来实现响应式执行方式。通过调用 observe()和 toObservable()方法可以返回 Observable 对象，比如：

```
Observable<String> ho = new UserCommand(restTemplate, 1L).observe();
Observable<String> co = new UserCommand(restTemplate, 1L).toObservable();
```

observe()和 toObservable()虽然都返回了 Observable，但是它们略有不同，前者返回的是一个 Hot Observable，该命令会在 observe()调用的时候立即执行，当 Observable 每次被订阅的时候会重放它的行为；而后者返回的是一个 Cold Observable，

toObservable()执行之后，命令不会被立即执行，只有当所有订阅者都订阅它之后才会执行。更多关于这两个方法的区别可见“原理分析”小节的内容，这里不做具体说明。

虽然 HystrixCommand 具备了 observe()和 toObservable()的功能，但是它的实现有一定的局限性，它返回的 Observable 只能发射一次数据，所以 Hystrix 还提供了另外一个特殊命令封装 HystrixObservableCommand，通过它实现的命令可以获取能发射多次的 Observable。

如果使用 HystrixObservableCommand 来实现命令封装，需要将命令的执行逻辑在 construct 方法中重载，这样 Hystrix 才能将具体逻辑包装到 Observable 内，如下所示：

```
public class UserObservableCommand extends HystrixObservableCommand<User> {

    private RestTemplate restTemplate;
    private Long id;

    public UserObservableCommand(Setter setter, RestTemplate restTemplate, Long id) {
        super(setter);
        this.restTemplate = restTemplate;
        this.id = id;
    }

    @Override
    protected Observable<User> construct() {
        return Observable.create(new Observable.OnSubscribe<User>() {
            @Override
            public void call(Subscriber<? super User> observer) {
                try {
                    if (!observer.isUnsubscribed()) {
                        User user = restTemplate.getForObject("http://USER-SERVICE/
users/{1}", User.class, id);
                        observer.onNext(user);
                        observer.onCompleted();
                    }
                } catch (Exception e) {
                    observer.onError(e);
                }
            }
        } );
    }
}
```

而对此的注解实现依然是使用@HystrixCommand，只是方法定义需要做一些变化，具体内容与 construct()的实现类似，如下所示：

```
    @HystrixCommand
    public Observable<User> getUserById(final String id) {
        return Observable.create(new Observable.OnSubscribe<User>() {
            @Override
            public void call(Subscriber<? super User> observer) {
                try {
                    if (!observer.isUnsubscribed()) {
                        User user = restTemplate.getForObject("http://HELLO-SERVICE/
users/{1}", User.class, id);
                        observer.onNext(user);
                        observer.onCompleted();
                    }
                } catch (Exception e) {
                    observer.onError(e);
                }
            }
        });
    }
```

在使用@HystrixCommand 注解实现响应式命令时，可以通过 observable-ExecutionMode 参数来控制是使用 observe()还是 toObservable()的执行方式。该参数有下面两种设置方式。

- @HystrixCommand(observableExecutionMode = ObservableExecution-Mode.EAGER)：EAGER 是该参数的模式值，表示使用 observe()执行方式。
- @HystrixCommand(observableExecutionMode = ObservableExecution-Mode.LAZY)：表示使用 toObservable()执行方式。

定义服务降级

fallback 是 Hystrix 命令执行失败时使用的后备方法，用来实现服务的降级处理逻辑。在HystrixCommand中可以通过重载getFallback()方法来实现服务降级逻辑，Hystrix 会在 run()执行过程中出现错误、超时、线程池拒绝、断路器熔断等情况时，执行 getFallback()方法内的逻辑，比如我们可以用如下方式实现服务降级逻辑：

```
public class UserCommand extends HystrixCommand<User> {

    private RestTemplate restTemplate;
    private Long id;

    public UserCommand(Setter setter, RestTemplate restTemplate, Long id) {
        super(setter);
        this.restTemplate = restTemplate;
        this.id = id;
```

```
    }

    @Override
    protected User run() {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class, id);
    }

    @Override
    protected User getFallback() {
        return new User();
    }

}
```

在 HystrixObservableCommand 实现的 Hystrix 命令中，我们可以通过重载 resumeWithFallback 方法来实现服务降级逻辑。该方法会返回一个 Observable 对象，当命令执行失败的时候，Hystrix 会将 Observable 中的结果通知给所有的订阅者。

若要通过注解实现服务降级只需要使用@HystrixCommand 中的 fallbackMethod 参数来指定具体的服务降级实现方法，如下所示：

```
public class UserService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "defaultUser")
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

    public User defaultUser() {
        return new User();
    }
}
```

在使用注解来定义服务降级逻辑时，我们需要将具体的 Hystrix 命令与 fallback 实现函数定义在同一个类中，并且 fallbackMethod 的值必须与实现 fallback 方法的名字相同。由于必须定义在一个类中，所以对于 fallback 的访问修饰符没有特定的要求，定义为 private、protected、public 均可。

在上面的例子中，defaultUser 方法将在 getUserById 执行时发生错误的情况下被执行。若 defaultUser 方法实现的并不是一个稳定逻辑，它依然可能会发生异常，那么我们也可以为它添加@HystrixCommand 注解以生成 Hystrix 命令，同时使用 fallbackMethod

来指定服务降级逻辑，比如：

```
public class UserService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "defaultUser")
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

    @HystrixCommand(fallbackMethod = "defaultUserSec")
    public User defaultUser() {
        // 此处可能是另外一个网络请求来获取，所以也有可能失败
        return new User("First Fallback");
    }

    public User defaultUserSec() {
        return new User("Second Fallback");
    }
}
```

在实际使用时，我们需要为大多数执行过程中可能会失败的 Hystrix 命令实现服务降级逻辑，但是也有一些情况可以不去实现降级逻辑，如下所示。

- **执行写操作的命令：**当 Hystrix 命令是用来执行写操作而不是返回一些信息的时候，通常情况下这类操作的返回类型是 `void` 或是为空的 `Observable`，实现服务降级的意义不是很大。当写入操作失败的时候，我们通常只需要通知调用者即可。
- **执行批处理或离线计算的命令：**当 Hystrix 命令是用来执行批处理程序生成一份报告或是进行任何类型的离线计算时，那么通常这些操作只需要将错误传播给调用者，然后让调用者稍后重试而不是发送给调用者一个静默的降级处理响应。

不论 Hystrix 命令是否实现了服务降级，命令状态和断路器状态都会更新，并且我们可以由此了解到命令执行的失败情况。

异常处理

异常传播

在 HystrixCommand 实现的 run()方法中抛出异常时，除了 HystrixBadRequestException 之外，其他异常均会被 Hystrix 认为命令执行失败并触发服务降级的处理逻辑，所以当需要在命令执行中抛出不触发服务降级的异常时来使用它。

而在使用注册配置实现 Hystrix 命令时，它还支持忽略指定异常类型功能，只需要通过设置@HystrixCommand 注解的 ignoreExceptions 参数，比如：

```
    @HystrixCommand(ignoreExceptions = {BadRequestException.class})
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
id);
    }
```

如上面代码的定义，当 getUserById 方法抛出了类型为 BadRequestException 的异常时，Hystrix 会将它包装在 HystrixBadRequestException 中抛出，这样就不会触发后续的 fallback 逻辑。

异常获取

当 Hystrix 命令因为异常（除了 HystrixBadRequestException 的异常）进入服务降级逻辑之后，往往需要对不同异常做针对性的处理，那么我们如何来获取当前抛出的异常呢？

在以传统继承方式实现的 Hystrix 命令中，我们可以用 getFallback()方法通过 Throwable getExecutionException()方法来获取具体的异常，通过判断来进入不同的处理逻辑。

除了传统的实现方式之外，注解配置方式也同样可以实现异常的获取。它的实现也非常简单，只需要在 fallback 实现方法的参数中增加 Throwable e 对象的定义，这样在方法内部就可以获取触发服务降级的具体异常内容了，比如：

```
    @HystrixCommand(fallbackMethod = "fallback1")
    User getUserById(String id) {
        throw new RuntimeException("getUserById command failed");
    }

    User fallback1(String id, Throwable e) {
        assert "getUserById command failed".equals(e.getMessage());
    }
```

命令名称、分组以及线程池划分

以继承方式实现的 Hystrix 命令使用类名作为默认的命令名称，我们也可以在构造函数中通过 Setter 静态类来设置，比如：

```
public UserCommand() {
    super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("GroupName"))
      .andCommandKey(HystrixCommandKey.Factory.asKey("CommandName")););
}
```

从上面 Setter 的使用中可以看到，我们并没有直接设置命令名称，而是先调用了 withGroupKey 来设置命令组名，然后才通过调用 andCommandKey 来设置命令名。这是因为在 Setter 的定义中，只有 withGroupKey 静态函数可以创建 Setter 的实例，所以 GroupKey 是每个 Setter 必需的参数，而 CommandKey 则是一个可选参数。

通过设置命令组，Hystrix 会根据组来组织和统计命令的告警、仪表盘等信息。那么为什么一定要设置命令组呢？因为除了根据组能实现统计之外，Hystrix 命令默认的线程划分也是根据命令分组来实现的。默认情况下，Hystrix 会让相同组名的命令使用同一个线程池，所以我们需要在创建 Hystrix 命令时为其指定命令组名来实现默认的线程池划分。

如果 Hystrix 的线程池分配仅仅依靠命令组来划分，那么它就显得不够灵活了，所以 Hystrix 还提供了 HystrixThreadPoolKey 来对线程池进行设置，通过它我们可以实现更细粒度的线程池划分，比如：

```
    public UserCommand() {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("CommandGroupKey"))
                .andCommandKey(HystrixCommandKey.Factory.asKey("CommandKey"))
                .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("ThreadPoolKey")));
    }
```

如果在没有特别指定 HystrixThreadPoolKey 的情况下，依然会使用命令组的方式来划分线程池。通常情况下，尽量通过 HystrixThreadPoolKey 的方式来指定线程池的划分，而不是通过组名的默认方式实现划分，因为多个不同的命令可能从业务逻辑上来看属于同一个组，但是往往从实现本身上需要跟其他命令进行隔离。

上面已经介绍了如何为通过继承实现的 HystrixCommand 设置命令名称、分组以及线程池划分，那么当我们使用@HystrixCommand 注解的时候，又该如何设置呢？只需设置@HystrixCommand 注解的 commandKey、groupKey 以及 threadPoolKey 属性即可，它们分别表示了命令名称、分组以及线程池划分，比如我们可以像下面这样进行设置：

```
    @HystrixCommand(commandKey = "getUserById", groupKey = "UserGroup", threadPoolKey
= "getUserByIdThread")
    public User getUserById(Long id) {
```

```
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
id);
    }
```

请求缓存

当系统用户不断增长时，每个微服务需要承受的并发压力也越来越大。在分布式环境下，通常压力来自于对依赖服务的调用，因为请求依赖服务的资源需要通过通信来实现，这样的依赖方式比起进程内的调用方式会引起一部分的性能损失，同时 HTTP 相比于其他高性能的通信协议在速度上没有任何优势，所以它有些类似于对数据库这样的外部资源进行读写操作，在高并发的情况下可能会成为系统的瓶颈。既然如此，我们很容易地可以联想到，类似数据访问的缓存保护是否也可以应用到依赖服务的调用上呢？

答案显而易见，在高并发的场景之下，Hystrix 中提供了请求缓存的功能，我们可以方便地开启和使用请求缓存来优化系统，达到减轻高并发时的请求线程消耗、降低请求响应时间的效果。

开启请求缓存功能

Hystrix 请求缓存的使用非常简单，我们只需要在实现 `HystrixCommand` 或 `HystrixObservableCommand` 时，通过重载 `getCacheKey()`方法来开启请求缓存，比如：

```
    public class UserCommand extends HystrixCommand<User> {

        private RestTemplate restTemplate;
        private Long id;

        public UserCommand(RestTemplate restTemplate, Long id) {
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("UserGroup")));
            this.restTemplate = restTemplate;
            this.id = id;
        }

        @Override
        protected User run() {
            return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
        }

        @Override
        protected String getCacheKey() {
            return String.valueOf(id);
```

```
    }

}
```

在上面的例子中，我们通过在 getCacheKey 方法中返回的请求缓存 key 值（使用了传入的获取 User 对象的 id 值），就能让该请求命令具备缓存功能。此时，当不同的外部请求处理逻辑调用了同一个依赖服务时，Hystrix 会根据 getCacheKey 方法返回的值来区分是否是重复的请求，如果它们的 cacheKey 相同，那么该依赖服务只会在第一个请求到达时被真实地调用一次，另外一个请求则是直接从请求缓存中返回结果，所以通过开启请求缓存可以让我们实现的 Hystrix 命令具备下面几项好处：

- 减少重复的请求数，降低依赖服务的并发度。
- 在同一用户请求的上下文中，相同依赖服务的返回数据始终保持一致。
- 请求缓存在 run() 和 construct() 执行之前生效，所以可以有效减少不必要的线程开销。

清理失效缓存功能

使用请求缓存时，如果只是读操作，那么不需要考虑缓存内容是否正确的问题，但是如果请求命令中还有更新数据的写操作，那么缓存中的数据就需要我们在进行写操作时进行及时处理，以防止读操作的请求命令获取到了失效的数据。

在 Hystrix 中，我们可以通过 HystrixRequestCache.clear() 方法来进行缓存的清理，具体示例如下：

```
    public class UserGetCommand extends HystrixCommand<User> {

        private static final HystrixCommandKey GETTER_KEY = HystrixCommandKey.Factory.
asKey("CommandKey");
        private RestTemplate restTemplate;
        private Long id;

        public UserGetCommand(RestTemplate restTemplate, Long id) {
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("GetSetGet"))
                    .andCommandKey(GETTER_KEY));
            this.restTemplate = restTemplate;
            this.id = id;
        }

        @Override
        protected User run() {
            return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
```

```
    }

    @Override
    protected String getCacheKey() {
        // 根据id置入缓存
        return String.valueOf(id);
    }

    public static void flushCache(Long id) {
        // 刷新缓存，根据id进行清理
        HystrixRequestCache.getInstance(GETTER_KEY,
                HystrixConcurrencyStrategyDefault.getInstance()).clear(String.
valueOf(id));
    }

  }

  public class UserPostCommand extends HystrixCommand<User> {

    private RestTemplate restTemplate;
    private User user;

    public UserPostCommand(RestTemplate restTemplate, User user) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("GetSetGet")));
        this.restTemplate = restTemplate;
        this.user = user;
    }

    @Override
    protected User run() {
        // 写操作
        User r = restTemplate.postForObject("http://USER-SERVICE/users", user,
User.class);
        // 刷新缓存，清理缓存中失效的User
        UserGetCommand.flushCache(user.getId());
        return r;
    }

  }
```

该示例中主要有两个请求命令：UserGetCommand 用于根据 id 获取 User 对象、而 UserPostCommand 用于更新 User 对象。当我们对 UserGetCommand 命令实现了请求缓存之后，那么势必需要为 UserPostCommand 命令实现缓存的清理，以保证 User 被更新

之后，Hystrix 请求缓存中相同缓存 Key 的结果被移除，这样在下一次获取 User 的时候不会从缓存中获取到未更新的结果。

我们可以看到，在上面 UserGetCommand 的实现中，增加了一个静态方法 flushCache，该方法通过HystrixRequestCache.getInstance(GETTER_KEY, HystrixConcurrency-StrategyDefault.getInstance()) 方法从默认的 Hystrix 并发策略中根据 GETTER_KEY 获取到该命令的请求缓存对象 HystrixRequestCache 的实例，然后再调用该请求缓存对象实例的 clear 方法，对 Key 为更新 User 的 id 值的缓存内容进行清理。而在 UserPostCommand 的实现中，在 run 方法调用依赖服务之后，增加了对 UserGetCommand 中静态方法 flushCache 的调用，以实现对失效缓存的清理。

工作原理

通过上面的入门例子，我们已经能够体会到在 Hystrix 中实现请求缓存是非常方便的，那么它是如何做到的呢？我们不妨通过分析其源码来了解一下它的实现原理，对其有一个深入的理解，有助于指导我们正确使用和配置请求缓存。由于 getCacheKey 方法在 AbstractCommand抽象命令类中实现，所以我们可以先从这个抽象命令类的实现中看起。

从下面 AbstractCommand 的源码片段中，我们可以看到，getCacheKey 方法默认返回的是 null，并且从 isRequestCachingEnabled 方法的实现逻辑中我们还可以知道，如果不重写 getCacheKey 方法，让它返回一个非 null 值，那么缓存功能是不会开启的；同时请求命令的缓存开启属性也需要设置为 true 才能开启（该属性默认为 true，所以通常用该属性来控制请求缓存功能的强制关闭）。

```
    abstract class AbstractCommand<R> implements HystrixInvokableInfo<R>,
HystrixObservable<R> {
        ...
        protected final HystrixRequestCache requestCache;
        ...
        protected String getCacheKey() {
            return null;
        }
        protected boolean isRequestCachingEnabled() {
            return properties.requestCacheEnabled().get() && getCacheKey() != null;
        }
        ...
        public Observable<R> toObservable() {
            ...
            // 尝试从缓存中获取结果
```

```
        final boolean requestCacheEnabled = isRequestCachingEnabled();
        final String cacheKey = getCacheKey();
        final AbstractCommand<R> _cmd = this;
        if (requestCacheEnabled) {
            HystrixCommandResponseFromCache<R> fromCache =
(HystrixCommandResponseFromCache<R>) requestCache.get(cacheKey);
            if (fromCache != null) {
                isResponseFromCache = true;
                return handleRequestCacheHitAndEmitValues(fromCache, _cmd);
            }
        }

        ...

        Observable<R> hystrixObservable =
Observable.defer(applyHystrixSemantics).map(wrapWithAllOnNextHooks);
        Observable<R> afterCache;

        // 加入缓存
        if (requestCacheEnabled && cacheKey != null) {
            HystrixCachedObservable<R> toCache =
HystrixCachedObservable.from(hystrixObservable, this);
            HystrixCommandResponseFromCache<R> fromCache =
(HystrixCommandResponseFromCache<R>) requestCache.putIfAbsent(cacheKey, toCache);
            if (fromCache != null) {
                toCache.unsubscribe();
                isResponseFromCache = true;
                return handleRequestCacheHitAndEmitValues(fromCache, _cmd);
            } else {
                afterCache = toCache.toObservable();
            }
        } else {
            afterCache = hystrixObservable;
        }
    }
    ...
}
```

另外，从命令异步执行的核心方法 `toObservable()` 中，我们可以看到与缓存相关的主要执行步骤，它分为两部分内容：尝试获取请求缓存以及将请求结果加入缓存。

- **尝试获取请求缓存：**Hystrix 命令在执行前会根据之前提到的 isRequestCachingEnabled 方法来判断当前命令是否启用了请求缓存。如果开启了请求缓存并且重写了 getCacheKey 方法，并返回了一个非 null 的缓存 Key 值，那么就使用 getCacheKey 返回的 Key 值去调用 HystrixRequestCache 中的 get(String cacheKey) 来获取缓存的 HystrixCachedObservable 对象。
- **将请求结果加入缓存：**在执行命令缓存操作之前，我们可以看到已经获得了一个延迟执行的命令结果对象 hystrixObservable。接下来与尝试获取请求缓存操作一样，需要先判断当前命令是否开启了请求缓存功能，如果开启了请求缓存并且 getCacheKey 返回了具体的 Key 值，就将 hystrixObservable 对象包装成请求缓存结果 HystrixCachedObservable 的实例对象 toCache，然后将其放入当前命令的缓存对象中。从调用的方法 putIfAbsent 中，我们大致可以猜到在请求缓存对象 HystrixRequestCache 中维护了一个线程安全的 Map 来保存请求缓存的响应，所以在调用 putIfAbsent 将包装的请求缓存放入缓存对象后，对其返回结果 fromCache 进行了判断，如果其不为 null，说明当前缓存 Key 的请求命令缓存命中，直接对 toCache 执行取消订阅操作（即，不再发起真实请求），同时调用缓存命令的处理方法 handleRequestCacheHitAndEmitValues 来执行缓存命中的结果获取。如果返回的 fromCache 为 null 说明缓存没有命中，则将当前结果 toCache 缓存起来，并将其转换成 Observable 返回给调用者使用。

使用注解实现请求缓存

Hystrix 的请求缓存除了可以通过上面传统的方式实现之外，还可以通过注解的方式进行配置实现。注解配置的定义实现同 JSR 107 的定义非常相似，但由于 Hystrix 不需要独立外置的缓存系统来支持，所以没有 JSR 107 的定义那么复杂，它只提供了三个专用于请求缓存的注解。

注解	描述	属性
@CacheResult	该注解用来标记请求命令返回的结果应该被缓存，它必须与 @HystrixCommand 注解结合使用	cacheKeyMethod
@CacheRemove	该注解用来让请求命令的缓存失效，失效的缓存根据定义的 Key 决定	commandKey, cacheKeyMethod
@CacheKey	该注解用来在请求命令的参数上标记，使其作为缓存的 Key 值，如果没有标注则会使用所有参数。如果同时还使用了 @CacheResult 和@CacheRemove 注解的 cacheKeyMethod 方法指定缓存 Key 的生成，那么该注解将不会起作用	value

JSR 107 是 Java 缓存 API 的定义，也被称为 JCache。它定义了一系列开发人员使用的标准化 Java 缓存 API 和服务提供商使用的标准 SPI。

下面我们从几个方面的实例来看看这几个注解的具体使用方法。

- **设置请求缓存：**通过注解为请求命令开启缓存功能非常简单，如下例所示，我们只需添加@CacheResult 注解即可。当该依赖服务被调用并返回 User 对象时，由于该方法被@CacheResult 注解修改，所以 Hystrix 会将该结果置入请求缓存中，而它的缓存 Key 值会使用所有的参数，也就是这里 Long 类型的 id 值。

```
    @CacheResult
    @HystrixCommand
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }
```

- **定义缓存 Key：**当使用注解来定义请求缓存时，若要为请求命令指定具体的缓存 Key 生成规则，我们可以使用 @CacheResult 和 @CacheRemove 注解的 cacheKeyMethod 方法来指定具体的生成函数；也可以通过使用@CacheKey 注解在方法参数中指定用于组装缓存 Key 的元素。

使用 cacheKeyMethod 方法的示例如下，它通过在请求命令的同一个类中定义一个专门生成 Key 的方法，并用@CacheResult 注解的 cacheKeyMethod 方法来指定它即可。它的配置方式类似于@HystrixCommand 服务降级 fallbackMethod 的使用。

```
    @CacheResult(cacheKeyMethod = "getUserByIdCacheKey")
    @HystrixCommand
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

    private Long getUserByIdCacheKey(Long id) {
        return id;
    }
```

通过@CacheKey 注解实现的方式更加简单，具体示例如下所示。但是在使用@CacheKey 注解的时候需要注意，它的优先级比 cacheKeyMethod 的优先级低，如果已经使用了 cacheKeyMethod 指定缓存 Key 的生成函数，那么@CacheKey 注解不会生效。

```
    @CacheResult
    @HystrixCommand
    public User getUserById(@CacheKey("id") Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
```

```
User.class, id);
    }
```

@CacheKey 注解除了可以指定方法参数作为缓存 Key 之外，它还允许访问参数对象的内部属性作为缓存 Key。比如下面的例子，它指定了 User 对象的 id 属性作为缓存 Key。

```
    @CacheResult
    @HystrixCommand
    public User getUserById(@CacheKey("id") User user) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
user.getId());
    }
```

- **缓存清理：**在之前的例子中，我们已经通过@CacheResult 注解将请求结果置入 Hystrix 的请求缓存之中。若该内容调用了 update 操作进行了更新，那么此时请求缓存中的结果与实际结果就会产生不一致（缓存中的结果实际上已经失效了），所以我们需要在 update 类型的操作上对失效的缓存进行清理。在 Hystrix 的注解配置中，可以通过@CacheRemove 注解来实现失效缓存的清理，比如下面的例子所示：

```
    @CacheResult
    @HystrixCommand
    public User getUserById(@CacheKey("id") Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
id);
    }

    @CacheRemove(commandKey = "getUserById")
    @HystrixCommand
    public void update(@CacheKey("id") User user) {
        return restTemplate.postForObject("http://USER-SERVICE/users", user,
User.class);
    }
```

需要注意的是，@CacheRemove 注解的 commandKey 属性是必须要指定的，它用来指明需要使用请求缓存的请求命令，因为只有通过该属性的配置，Hystrix 才能找到正确的请求命令缓存位置。

请求合并

微服务架构中的依赖通常通过远程调用实现，而远程调用中最常见的问题就是通信消耗与连接数占用。在高并发的情况之下，因通信次数的增加，总的通信时间消耗将会变得不那么理想。同时，因为依赖服务的线程池资源有限，将出现排队等待与响应延迟的情况。为了优化这两个问题，Hystrix 提供了 HystrixCollapser 来实现请求的合并，以减少通信消耗和线程数的占用。

HystrixCollapser 实现了在 HystrixCommand 之前放置一个合并处理器，将处于一个很短的时间窗（默认 10 毫秒）内对同一依赖服务的多个请求进行整合并以批量方式发起请求的功能（服务提供方也需要提供相应的批量实现接口）。通过 HystrixCollapser 的封装，开发者不需要关注线程合并的细节过程，只需关注批量化服务和处理。下面我们从 HystrixCollapser 的使用实例中对其合并请求的过程一探究竟。

```
    public abstract class HystrixCollapser<BatchReturnType, ResponseType,
RequestArgumentType> implements
            HystrixExecutable<ResponseType>, HystrixObservable<ResponseType> {
        ...
        public abstract RequestArgumentType getRequestArgument();

        protected abstract HystrixCommand<BatchReturnType>
createCommand(Collection<CollapsedRequest<ResponseType, RequestArgumentType>>
requests);

        protected abstract void mapResponseToRequests(BatchReturnType batchResponse,
Collection<CollapsedRequest<ResponseType, RequestArgumentType>> requests);
        ...
    }
```

从 HystrixCollapser 抽象类的定义中可以看到，它指定了三个不同的类型。

- BatchReturnType：合并后批量请求的返回类型。
- ResponseType：单个请求返回的类型。
- RequestArgumentType：请求参数类型。

而对于这三个类型的使用可以在它的三个抽象方法中看到。

- RequestArgumentType getRequestArgument()：该函数用来定义获取请求参数的方法。
- HystrixCommand<BatchReturnType>createCommand(Collection<CollapsedRequest<ResponseType, RequestArgumentType>> requests)：合并请求产生批量命令的具体实现方法。
- mapResponseToRequests(BatchReturnType batchResponse, Collection<CollapsedRequest<ResponseType, RequestArgumentType>> requests)：批量命令结果返回后的处理，这里需要实现将批量结果拆分并传递给合并前的各个原子请求命令的逻辑。

接下来，我们通过一个简单的示例来直观理解实现请求合并的过程。

假设当前微服务 USER-SERVICE 提供了两个获取 User 的接口。

- /users/{id}：根据 id 返回 User 对象的 GET 请求接口。
- /users?ids={ids}：根据 ids 返回 User 对象列表的 GET 请求接口，其中 ids 为以逗号分隔的 id 集合。

而在服务消费端，已经为这两个远程接口通过 RestTemplate 实现了简单的调用，具体如下所示：

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private RestTemplate restTemplate;

    @Override
    public User find(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

    @Override
    public List<User> findAll(List<Long> ids) {
        return restTemplate.getForObject("http://USER-SERVICE/users?ids={1}",
List.class, StringUtils.join(ids, ","));
    }

}
```

接着，我们实现将短时间内多个获取单一 User 对象的请求命令进行合并。

- 第一步，为请求合并的实现准备一个批量请求命令的实现，具体如下：

```
public class UserBatchCommand extends HystrixCommand<List<User>> {

    UserService userService;
    List<Long> userIds;

    public UserBatchCommand(UserService userService, List<Long> userIds) {
        super(Setter.withGroupKey(asKey("userServiceCommand")));
        this.userIds = userIds;
        this.userService = userService;
    }

    @Override
    protected List<User> run() throws Exception {
```

```
        return userService.findAll(userIds);
    }

}
```

批量请求命令实际上就是一个简单的 HystrixCommand 实现，从上面的实现中可以看到它通过调用 userService.findAll 方法来访问/users?ids={ids}接口以返回 User 的列表结果。

- 第二步，通过继承 HystrixCollapser 实现请求合并器：

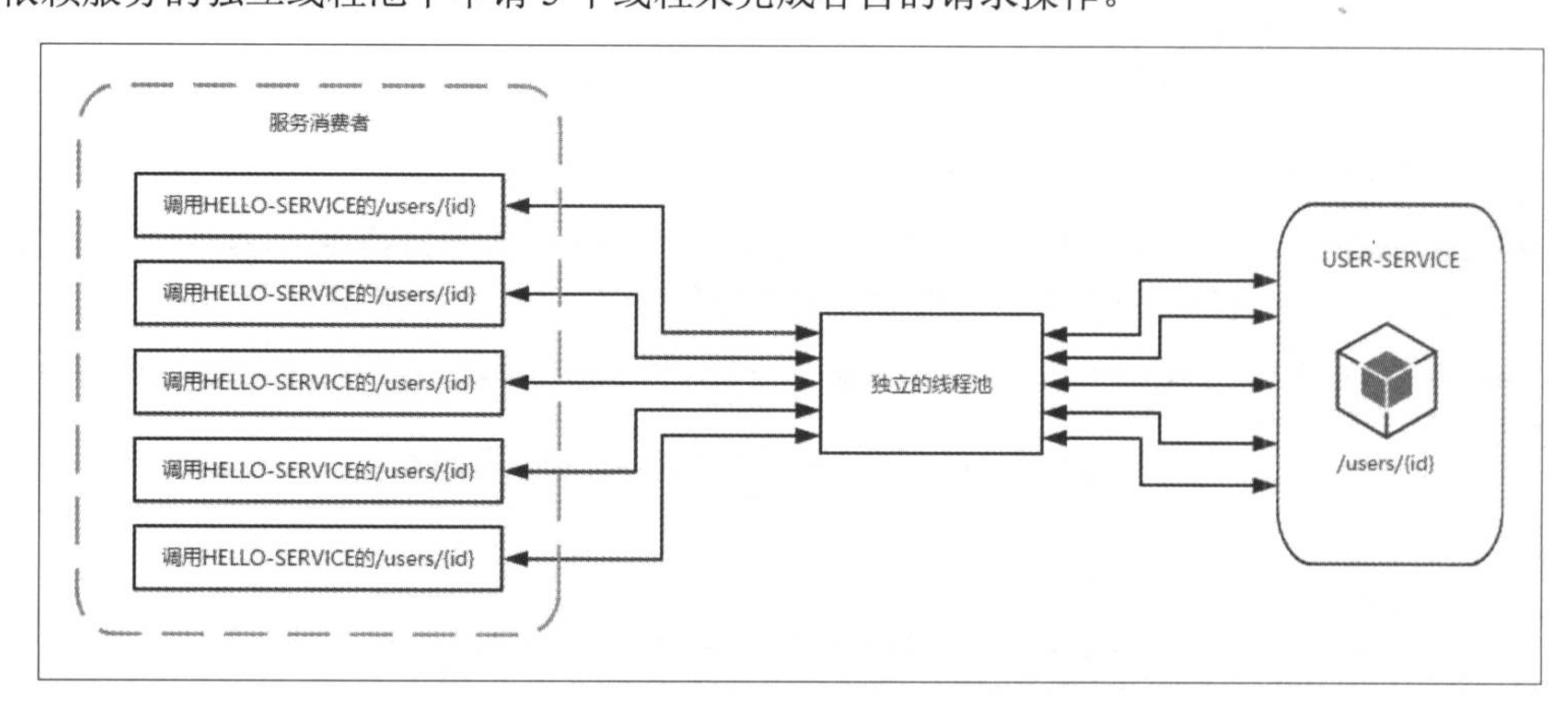
```
    public class UserCollapseCommand extends HystrixCollapser<List<User>, User,
Long> {

        private UserService userService;
        private Long userId;

        public UserCollapseCommand(UserService userService, Long userId) {
            super(Setter.withCollapserKey(HystrixCollapserKey.Factory.asKey
("userCollapseCommand")).andCollapserPropertiesDefaults(

HystrixCollapserProperties.Setter().withTimerDelayInMilliseconds(100)));
            this.userService = userService;
            this.userId = userId;
        }

        @Override
        public Long getRequestArgument() {
            return userId;
        }

        @Override
        protected HystrixCommand<List<User>>
createCommand(Collection<CollapsedRequest<User, Long>> collapsedRequests) {
            List<Long> userIds = new ArrayList<>(collapsedRequests.size());

userIds.addAll(collapsedRequests.stream().map(CollapsedRequest::getArgument).collec
t(Collectors.toList()));
            return new UserBatchCommand(userService, userIds);
        }

        @Override
        protected void mapResponseToRequests(List<User> batchResponse,
Collection<CollapsedRequest<User, Long>> collapsedRequests) {
            int count = 0;
            for (CollapsedRequest<User, Long> collapsedRequest : collapsedRequests) {
                User user = batchResponse.get(count++);
```

```
            collapsedRequest.setResponse(user);
        }
    }

}
```

在上面的构造函数中，我们为请求合并器设置了时间延迟属性，合并器会在该时间窗内收集获取单个 User 的请求并在时间窗结束时进行合并组装成单个批量请求。getRequestArgument 方法返回给定的单个请求参数 userId，而 createCommand 和 mapResponseToRequests 是请求合并器的两个核心。

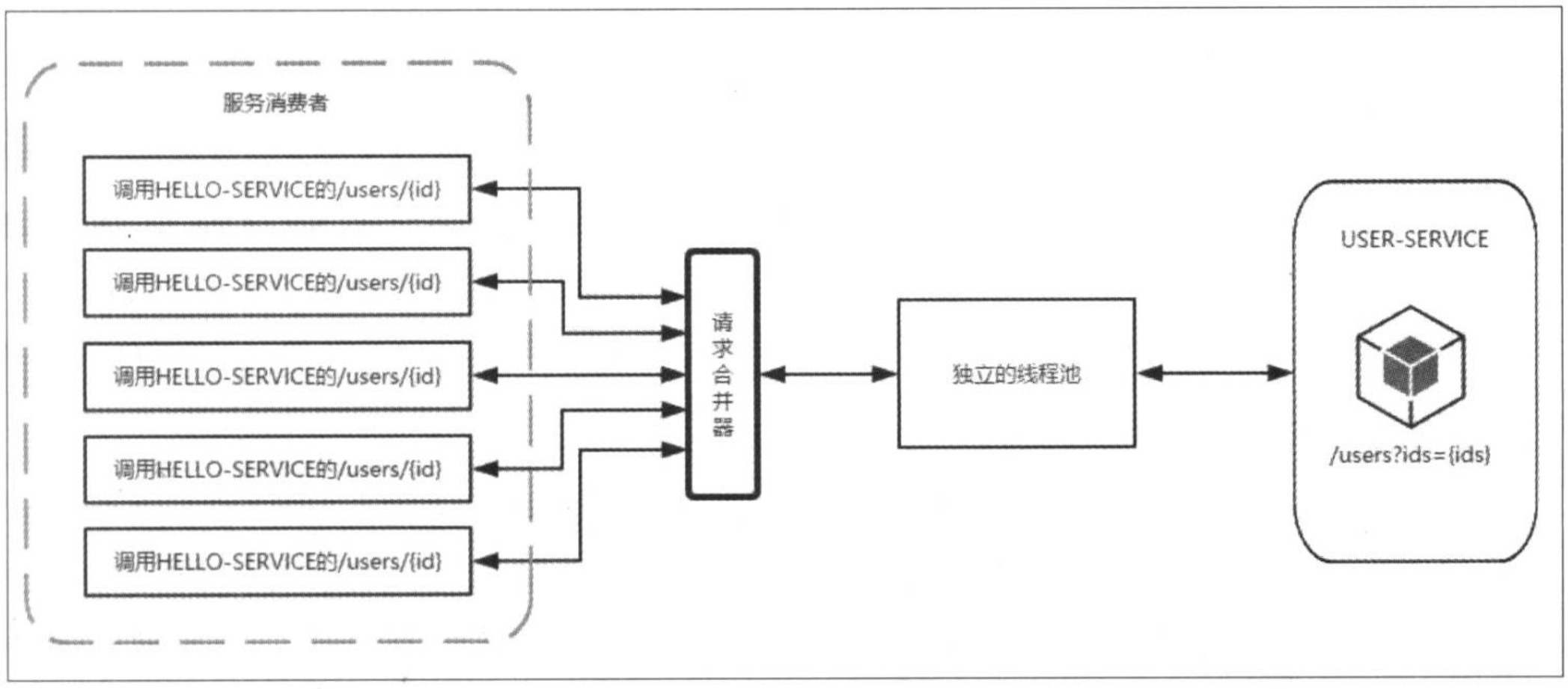
- createCommand：该方法的 collapsedRequests 参数中保存了延迟时间窗中收集到的所有获取单个 User 的请求。通过获取这些请求的参数来组织上面我们准备的批量请求命令 UserBatchCommand 实例。
- mapResponseToRequests：在批量请求命令 UserBatchCommand 实例被触发执行完成之后，该方法开始执行，其中 batchResponse 参数保存了 createCommand 中组织的批量请求命令的返回结果，而 collapsedRequests 参数则代表了每个被合并的请求。在这里我们通过遍历批量结果 batchResponse 对象，为 collapsedRequests 中每个合并前的单个请求设置返回结果，以此完成批量结果到单个请求结果的转换。

下图展示了在未使用 HystrixCollapser 请求合并器之前的线程使用情况。可以看到，当服务消费者同时对 USER-SERVICE 的/users/{id}接口发起了 5 个请求时，会向该依赖服务的独立线程池中申请 5 个线程来完成各自的请求操作。



而在使用了 HystrixCollapser 请求合并器之后，相同情况下的线程占用如下图所示。由于同一时间发生的 5 个请求处于请求合并器的一个时间窗内，这些发向/users/{id}

接口的请求被请求合并器拦截下来，并在合并器中进行组合，然后将这些请求合并成一个请求发向 USER-SERVICE 的批量接口/users?ids={ids}。在获取到批量请求结果之后，通过请求合并器再将批量结果拆分并分配给每个被合并的请求。从图中我们可以看到，通过使用请求合并器有效减少了对线程池中资源的占用。所以在资源有效并且短时间内会产生高并发请求的时候，为避免连接不够用而引起的延迟可以考虑使用请求合并器的方式来处理和优化。



使用注解实现请求合并器

在快速入门的例子中，我们使用@HystrixCommand 注解优雅地实现了 HystrixCommand 的定义，那么对于请求合并器是否也可以通过注解来定义呢？答案是肯定的！

以上面实现的请求合并器为例，还可以通过如下方式实现：

```
@Service
public class UserService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCollapser(batchMethod = "findAll", collapserProperties = {
            @HystrixProperty(name="timerDelayInMilliseconds", value = "100")
    })
    public User find(Long id) {
        return null;
    }

    @HystrixCommand
```

```
        public List<User> findAll(List<Long> ids) {
            return restTemplate.getForObject("http://USER-SERVICE/users?ids={1}",
List.class, StringUtils.join(ids, ","));
        }
    }
```

我们之前已经介绍过@HystrixCommand 了，可以看到，这里通过它定义了两个 Hystrix 命令，一个用于请求/users/{id}接口，一个用于请求/users?ids={ids}接口。而在请求/users/{id}接口的方法上通过@HystrixCollapser 注解为其创建了合并请求器，通过 batchMethod 属性指定了批量请求的实现方法为 findAll 方法（即请求/users?ids={ids}接口的命令），同时通过 collapserProperties 属性为合并请求器设置了相关属性，这里使用@HystrixProperty(name="timerDelayInMilliseconds", value = "100")将合并时间窗设置为 100 毫秒。这样通过@HystrixCollapser 注解简单而又优雅地实现了在/users/{id}依赖服务之前设置了一个批量请求合并器。

请求合并的额外开销

虽然通过请求合并可以减少请求的数量以缓解依赖服务线程池的资源，但是在使用的时候也需要注意它所带来的额外开销：用于请求合并的延迟时间窗会使得依赖服务的请求延迟增高。比如，某个请求不通过请求合并器访问的平均耗时为 5ms，请求合并的延迟时间窗为 10ms（默认值），那么当该请求设置了请求合并器之后，最坏情况下（在延迟时间窗结束时才发起请求）该请求需要 15ms 才能完成。

由于请求合并器的延迟时间窗会带来额外开销，所以我们是否使用请求合并器需要根据依赖服务调用的实际情况来选择，主要考虑下面两个方面。

- **请求命令本身的延迟**。如果依赖服务的请求命令本身是一个高延迟的命令，那么可以使用请求合并器，因为延迟时间窗的时间消耗显得微不足道了。
- **延迟时间窗内的并发量**。如果一个时间窗内只有 1~2 个请求，那么这样的依赖服务不适合使用请求合并器。这种情况不但不能提升系统性能，反而会成为系统瓶颈，因为每个请求都需要多消耗一个时间窗才响应。相反，如果一个时间窗内具有很高的并发量，并且服务提供方也实现了批量处理接口，那么使用请求合并器可以有效减少网络连接数量并极大提升系统吞吐量，此时延迟时间窗所增加的消耗就可以忽略不计了。

属性详解

在之前介绍 Hystrix 的使用方法时，已经涉及过一些 Hystrix 属性的配置，我们可以根

据实现 HystrixCommand 的不同方式将配置方法分为如下两类。

- 当通过继承的方式实现时，可使用 Setter 对象来对请求命令的属性进行设置，比如下面的例子：

```
    public HystrixCommandInstance(int id) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("ExampleGroup"))
            .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                .withExecutionTimeoutInMilliseconds(500)));
        this.id = id;
    }
```

- 当通过注解的方法实现时，只需使用@HystrixCommand 中的 command-Properties 属性来设置，比如：

```
    @HystrixCommand(commandKey = "helloKey",
        commandProperties = {
            @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",
value = "5000")
        }
    )
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
id);
    }
```

实际上，Hystrix 为我们提供的配置内容和配置方式远不止上面这些，它提供了非常丰富和灵活的配置方法，下面我们将详解介绍 HystrixPropertiesStrategy 实现的各项配置属性。在具体说明这些属性之前，我们需要了解一下这些属性都存在下面 4 个不同优先级别的配置（优先级由低到高）。

- **全局默认值：**如果没有设置下面三个级别的属性，那么这个属性就是默认值。由于该属性通过代码定义，所以对于这个级别，我们主要关注它在代码中定义的默认值即可。
- **全局配置属性：**通过在配置文件中定义全局属性值，在应用启动时或在与 Spring Cloud Config 和 Spring Cloud Bus 实现的动态刷新配置功能配合下，可以实现对“全局默认值”的覆盖以及在运行期对“全局默认值”的动态调整。
- **实例默认值：**通过代码为实例定义的默认值。通过代码的方式为实例设置属性值来覆盖默认的全局配置。

- **实例配置属性**：通过配置文件来为指定的实例进行属性配置，以覆盖前面的三个默认值。它也可用 Spring Cloud Config 和 Spring Cloud Bus 实现的动态刷新配置功能实现对具体实例配置的动态调整。

通过理解 Hystrix 4 个级别的属性配置，对设置 Hystrix 的默认值以及在线上如何根据实际情况去调整配置非常有帮助，下面我们来具体看看它有哪些具体的属性配置。

Command 属性

Command 属性主要用来控制 HystrixCommand 命令的行为。

它主要有下面 5 种不同类型的属性配置。

execution 配置

execution 配置控制的是 HystrixCommand.run() 的执行。

- execution.isolation.strategy：该属性用来设置 HystrixCommand.run() 执行的隔离策略，它有如下两个选项。
  - THREAD：通过线程池隔离的策略。它在独立的线程上执行，并且它的并发限制受线程池中线程数量的限制。
  - SEMAPHORE：通过信号量隔离的策略。它在调用线程上执行，并且它的并发限制受信号量计数的限制。

属性级别	默认值、配置方式、配置属性
全局默认值	THREAD
全局配置属性	hystrix.command.default.execution.isolation.strategy
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionIsolationStrategy (ExecutionIsolationStrategy.THREAD) 设置，也可通过@HystrixProperty (name="execution.isolation.strategy", value= "THREAD") 注解设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.strategy

- execution.isolation.thread.timeoutInMilliseconds：该属性用来配置 HystrixCommand 执行的超时时间，单位为毫秒。当 HystrixCommand 执行时间超过该配置值之后，Hystrix 会将该执行命令标记为 TIMEOUT 并进入服务降级处理逻辑。

属性级别	默认值、配置方式、配置属性
全局默认值	1000 毫秒

续表

属性级别	默认值、配置方式、配置属性
全局配置属性	hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(int value) 设置，也可通过 @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",value="2000") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.thread.timeoutInMilliseconds

- execution.timeout.enabled：该属性用来配置 HystrixCommand.run() 的执行是否启用超时时间。默认为 true，如果设置为 false，那么 execution.isolation.thread.timeoutInMilliseconds 属性的配置将不再起作用。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.execution.timeout.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionTimeoutEnabled(boolean value) 设置，也可通过 @HystrixProperty(name="execution.timeout.enabled", value="false") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.timeout.enabled

- execution.isolation.thread.interruptOnTimeout：该属性用来配置当 HystrixCommand.run() 执行超时的时候是否要将它中断。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.execution.isolation.thread.interruptOnTimeout
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionIsolationThreadInterruptOnTimeout(boolean value) 设置，也可通过 @HystrixProperty(name="execution.isolation.thread.interruptOnTimeout", value="false") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.thread.interruptOnTimeout

- execution.isolation.thread.interruptOnCancel：该属性用来配置当HystrixCommand.run()执行被取消的时候是否要将它中断。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.execution.isolation.thread.interruptOnCancel
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionIsolationThreadInterruptOnCancel(boolean value)设置，也可通过@HystrixProperty(name="execution.isolation.thread.interruptOnCancel", value="false")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.thread.interruptOnCancel

- execution.isolation.semaphore.maxConcurrentRequests:当HystrixCommand的隔离策略使用信号量的时候，该属性用来配置信号量的大小（并发请求数）。当最大并发请求数达到该设置值时，后续的请求将会被拒绝。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionIsolationSemaphoreMaxConcurrentRequests(int value)设置，也可通过@HystrixProperty(name="execution.isolation.semaphore.maxConcurrentRequests", value="2")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.semaphore.maxConcurrentRequests

fallback 配置

下面这些属性用来控制 HystrixCommand.getFallback()的执行。这些属性同时适用于线程池的信号量的隔离策略。

- fallback.isolation.semaphore.maxConcurrentRequests：该属性用来设置从调用线程中允许 HystrixCommand.getFallback()方法执行的最大并发请求数。当达到最大并发请求数时，后续的请求将会被拒绝并抛出异常（因为它已经没有后续的 fallback 可以被调用了）。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests
实例默认值	通过 HystrixCommandProperties.Setter().withFallbackIsolationSemaphoreMaxConcurrentRequests(int value)设置，也可通过@HystrixProperty(name="fallback.isolation.semaphore.maxConcurrentRequests", value="20")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.fallback.isolation.semaphore.maxConcurrentRequests

- fallback.enabled：该属性用来设置服务降级策略是否启用，如果设置为 false，那么当请求失败或者拒绝发生时，将不会调用 HystrixCommand.getFallback()来执行服务降级逻辑。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.fallback.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withFallbackEnabled(boolean value)设置，也可通过@HystrixProperty(name= "fallback.enabled", value="false")注解来设置
实例配置属性	配置属性：hystrix.command.HystrixCommandKey.fallback.enabled

circuitBreaker 配置

下面这些是断路器的属性配置，用来控制 HystrixCircuitBreaker 的行为。

- circuitBreaker.enabled：该属性用来确定当服务请求命令失败时，是否使用断路器来跟踪其健康指标和熔断请求。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.circuitBreaker.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerEnabled(boolean value)设置，也可通过@HystrixProperty(name="circuitBreaker.enabled",value="false")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker.enabled

- circuitBreaker.requestVolumeThreshold：该属性用来设置在滚动时间窗中，断路器熔断的最小请求数。例如，默认该值为 20 的时候，如果滚动时间窗（默认 10 秒）内仅收到了 19 个请求，即使这 19 个请求都失败了，断路器也不会打开。

属性级别	默认值、配置方式、配置属性
全局默认值	20
全局配置属性	hystrix.command.default.circuitBreaker.requestVolumeThreshold
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerRequestVolumeThreshold(int value)设置，也可通过@HystrixProperty(name="circuitBreaker.requestVolumeThreshold", value="30")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker.requestVolumeThreshold

- circuitBreaker.sleepWindowInMilliseconds：该属性用来设置当断路器打开之后的休眠时间窗。休眠时间窗结束之后，会将断路器置为“半开”状态，尝试熔断的请求命令，如果依然失败就将断路器继续设置为“打开”状态，如果成功就设置为“关闭”状态。

属性级别	默认值、配置方式、配置属性
全局默认值	5000
全局配置属性	hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerSleepWindowInMilliseconds(int value)设置，也可通过@HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds",value="3000")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker.sleepWindowInMilliseconds

- circuitBreaker.errorThresholdPercentage：该属性用来设置断路器打开的错误百分比条件。例如，默认值为 5000 的情况下，表示在滚动时间窗中，在请求数量超过 circuitBreaker.requestVolumeThreshold 阈值的前提下，如果错误请求数的百分比超过 50，就把断路器设置为“打开”状态，否则就设置为“关闭”状态。

属性级别	默认值、配置方式、配置属性
全局默认值	50

续表

属性级别	默认值、配置方式、配置属性
全局配置属性	hystrix.command.default.circuitBreaker.errorThresholdPercentage
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerErrorThresholdPercentage(int value) 设置，也可通过@HystrixProperty (name="circuitBreaker.errorThresholdPercentage", value="40") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker. errorThresholdPercentage

- circuitBreaker.forceOpen：如果将该属性设置为 true，断路器将强制进入“打开”状态，它会拒绝所有请求。该属性优先于 circuitBreaker.forceClosed 属性。

属性级别	默认值、配置方式、配置属性
全局默认值	false
全局配置属性	hystrix.command.default.circuitBreaker.forceOpen
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerForceOpen (boolean value) 设置，也可通过@HystrixProperty (name="circuitBreaker. forceOpen", value="true") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker. forceOpen

- circuitBreaker.forceClosed：如果将该属性设置为 true，断路器将强制进入“关闭”状态，它会接收所有请求。如果 circuitBreaker.forceOpen 属性为 true，该属性不会生效。

属性级别	默认值、配置方式、配置属性
全局默认值	false
全局配置属性	hystrix.command.default.circuitBreaker.forceClosed
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerForceClosed(boolean value) 设置，也可通过 @HystrixProperty(name="circuitBreaker.forceClosed", value="true") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker. forceClosed

metrics 配置

下面的属性均与 HystrixCommand 和 HystrixObservableCommand 执行中捕获

的指标信息有关。

- metrics.rollingStats.timeInMilliseconds：该属性用来设置滚动时间窗的长度，单位为毫秒。该时间用于断路器判断健康度时需要收集信息的持续时间。断路器在收集指标信息的时候会根据设置的时间窗长度拆分成多个“桶”来累计各度量值，每个“桶”记录了一段时间内的采集指标。例如，当采用默认值 10000 毫秒时，断路器默认将其拆分成 10 个桶（桶的数量也可通过 metrics.rollingStats.numBuckets 参数设置），每个桶记录 1000 毫秒内的指标信息。

属性级别	默认值、配置方式、配置属性
全局默认值	10000
全局配置属性	hystrix.command.default.metrics.rollingStats.timeInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingStatisticalWindowInMilliseconds(int value) 设置，也可通过 @HystrixProperty(name="metrics.rollingStats.timeInMilliseconds", value="20000")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingStats.timeInMilliseconds

---

注意：该属性从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- metrics.rollingStats.numBuckets：该属性用来设置滚动时间窗统计指标信息时划分“桶”的数量。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.command.default.metrics.rollingStats.numBuckets
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingStatisticalWindowBuckets(int value) 设置，也可通过 @HystrixProperty(name="metrics.rollingStats.numBuckets",value="20")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingStats.numBuckets

注意： metrics.rollingStats.timeInMilliseconds 参数的设置必须能够被 metrics.rollingStats.numBuckets 参数整除，不然将抛出异常。该参数与 metrics.rollingStats.timeInMilliseconds 一样，从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- metrics.rollingPercentile.enabled：该属性用来设置对命令执行的延迟是否使用百分位数来跟踪和计算。如果设置为 false，那么所有的概要统计都将返回 -1。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.metrics.rollingPercentile.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingPercentileEnabled(boolean value) 设置，也可通过 @HystrixProperty(name="metrics.rollingPercentile.enabled", value="false") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingPercentile.enabled

- metrics.rollingPercentile.timeInMilliseconds：该属性用来设置百分位统计的滚动窗口的持续时间，单位为毫秒。

属性级别	默认值、配置方式、配置属性
全局默认值	60000
全局配置属性	hystrix.command.default.metrics.rollingPercentile.timeInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingPercentileWindowInMilliseconds(int value) 设置，也可通过 @HystrixProperty(name="metrics.rollingPercentile.timeInMilliseconds", value="50000") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingPercentile.timeInMilliseconds

注意：该属性从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- `metrics.rollingPercentile.numBuckets`：该属性用来设置百分位统计滚动窗口中使用“桶”的数量。

属性级别	默认值、配置方式、配置属性
全局默认值	6
全局配置属性	hystrix.command.default.metrics.rollingPercentile.numBuckets
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingPercentileWindowBuckets(int value)设置，也可通过@HystrixProperty(name="metrics.rollingPercentile.numBuckets",value="5")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingPercentile.numBuckets

注意：metrics.rollingPercentile.timeInMilliseconds 参数的设置必须能够被 metrics.rollingPercentile.numBuckets 参数整除，不然将会抛出异常。该属性从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- `metrics.rollingPercentile.bucketSize`：该属性用来设置在执行过程中每个“桶”中保留的最大执行次数。如果在滚动时间窗内发生超过该设定值的执行次数，就从最初的位置开始重写。例如，将该值设置为 100，滚动窗口为 10 秒，若在 10 秒内一个“桶”中发生了 500 次执行，那么该“桶”中只保留最后的 100 次执行的统计。另外，增加该值的大小将会增加内存量的消耗，并增加排序百分位数所需的计算时间。

属性级别	默认值、配置方式、配置属性
全局默认值	100
全局配置属性	hystrix.command.default.metrics.rollingPercentile.bucketSize
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingPercentileBucketSize(int value)设置，也可通过@HystrixProperty(name="metrics.rollingPercentile.bucketSize",value= "120")注解来设置

续表

属性级别	默认值、配置方式、配置属性
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingPercentile.bucketSize

注意：该属性从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- metrics.healthSnapshot.intervalInMilliseconds：该属性用来设置采集影响断路器状态的健康快照（请求的成功、错误百分比）的间隔等待时间。

属性级别	默认值、配置方式、配置属性
全局默认值	500
全局配置属性	hystrix.command.default.metrics.healthSnapshot.intervalInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsHealthSnapshotIntervalInMilliseconds(int value)设置，也可通过@HystrixProperty(name="metrics.healthSnapshot.intervalInMilliseconds",value="600")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.healthSnapshot.intervalInMilliseconds

requestContext 配置

下面这些属性涉及 HystrixCommand 使用的 HystrixRequestContext 的设置。

- requestCache.enabled：此属性用来配置是否开启请求缓存。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.requestCache.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withRequestCacheEnabled(boolean value)设置，也可通过@HystrixProperty(name="requestCache.enabled", value="false")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.requestCache.enabled

- requestLog.enabled：该属性用来设置 HystrixCommand 的执行和事件是否打印日志到 HystrixRequestLog 中。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.requestLog.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withRequestLogEnabled(boolean value)设置，也可通过@HystrixProperty(name="requestLog.enabled", value="false")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.requestLog.enabled

collapser 属性

该属性除了在代码中用 set 和配置文件配置之外，也可使用注解进行配置。可使用 @HystrixCollapser 中的 collapserProperties 属性来设置，比如：

```
@HystrixCollapser(batchMethod = "batch", collapserProperties = {
    @HystrixProperty(name="timerDelayInMilliseconds", value = "20")
})
```

下面这些属性用来控制命令合并相关的行为。

- maxRequestsInBatch：该参数用来设置一次请求合并批处理中允许的最大请求数。

属性级别	默认值、配置方式、配置属性
全局默认值	Integer.MAX_VALUE
全局配置属性	hystrix.collapser.default.maxRequestsInBatch
实例默认值	通过 HystrixCollapserProperties.Setter().withMaxRequestsInBatch(int value)设置，也可通过@HystrixProperty(name="maxRequestsInBatch", value="false")注解来设置
实例配置属性	hystrix.collapser.HystrixCollapserKey.maxRequestsInBatch

- timerDelayInMilliseconds：该参数用来设置批处理过程中每个命令延迟的时间，单位为毫秒。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.collapser.default.timerDelayInMilliseconds
实例默认值	通过 HystrixCollapserProperties.Setter().withTimerDelayInMilliseconds(int value)设置，也可通过@HystrixProperty(name="timerDelayInMilliseconds",value="20")注解来设置
实例配置属性	hystrix.collapser.HystrixCollapserKey.timerDelayInMilliseconds

- requestCache.enabled：该参数用来设置批处理过程中是否开启请求缓存。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.collapser.default.requestCache.enabled
实例默认值	通过 HystrixCollapserProperties.Setter().withRequestCacheEnabled (boolean value)设置，也可通过@HystrixProperty(name="requestCache. enabled", value="false")注解来设置
实例配置属性	hystrix.collapser.HystrixCollapserKey.requestCache. enabled

threadPool 属性

该属性除了在代码中用 set 和配置文件配置之外，还可使用注解进行配置。可使用 @HystrixCommand 中的 threadPoolProperties 属性来设置，比如：

```
@HystrixCommand(fallbackMethod = "helloFallback", commandKey = "helloKey",
    threadPoolProperties = {
        @HystrixProperty(name="coreSize", value = "20")
    }
)
```

下面这些属性用来控制 Hystrix 命令所属线程池的配置。

- coreSize：该参数用来设置执行命令线程池的核心线程数，该值也就是命令执行的最大并发量。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.threadpool.default.coreSize
实例默认值	通过 HystrixThreadPoolProperties.Setter().withCoreSize(int value) 设置，也可通过@HystrixProperty(name="coreSize", value="false")注解来设置
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.coreSize

- maxQueueSize：该参数用来设置线程池的最大队列大小。当设置为-1 时，线程池将使用 SynchronousQueue 实现的队列，否则将使用 LinkedBlockingQueue 实现的队列。

属性级别	默认值、配置方式、配置属性
全局默认值	-1
全局配置属性	hystrix.threadpool.default.maxQueueSize

续表

属性级别	默认值、配置方式、配置属性
实例默认值	通过 HystrixThreadPoolProperties.Setter().withMaxQueueSize (int value) 设置，也可通过 @HystrixProperty(name="maxQueueSize", value="10")注解来设置
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.maxQueueSize

注意：该属性只有在初始化的时候才有用，无法通过动态刷新的方式来调整。

- queueSizeRejectionThreshold：该参数用来为队列设置拒绝阈值。通过该参数，即使队列没有达到最大值也能拒绝请求。该参数主要是对 LinkedBlockingQueue 队列的补充，因为 LinkedBlockingQueue 队列不能动态修改它的对象大小，而通过该属性就可以调整拒绝请求的队列大小了。

属性级别	默认值、配置方式、配置属性
全局默认值	5
全局配置属性	hystrix.threadpool.default.queueSizeRejectionThreshold
实例默认值	通过 HystrixThreadPoolProperties.Setter().withQueueSizeRejectionThreshold(int value) 设置，也可通过 @HystrixProperty(name="queueSizeRejectionThreshold", value="10")注解来设置
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.queueSizeRejectionThreshold

注意：当 maxQueueSize 属性为-1 的时候，该属性不会生效。

- metrics.rollingStats.timeInMilliseconds：该参数用来设置滚动时间窗的长度，单位为毫秒。该滚动时间窗的长度用于线程池的指标度量，它会被分成多个“桶”来统计指标。

属性级别	默认值、配置方式、配置属性
全局默认值	10000
全局配置属性	hystrix.threadpool.default.metrics.rollingStats.timeInMilliseconds
实例默认值	通过 HystrixThreadPoolProperties.Setter().withMetricsRollingStatisticalWindowInMilliseconds(int value)设置，也可通过@HystrixProperty(name="metrics.rollingStats.timeInMilliseconds", value="10")注解来设置

续表

属性级别	默认值、配置方式、配置属性
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.metrics.rollingStats.timeInMilliseconds

- metrics.rollingStats.numBuckets：该参数用来设置滚动时间窗被划分成“桶”的数量。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.threadpool.default.metrics.rollingStats. numBuckets
实例默认值	通过 HystrixThreadPoolProperties.Setter().withMetricsRolling-StatisticalWindowBuckets(int value) 设置，也可通过 @HystrixProperty (name="metrics.rollingStats.numBuckets", value="10")注解来设置
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.metrics.rollingStats.numBuckets

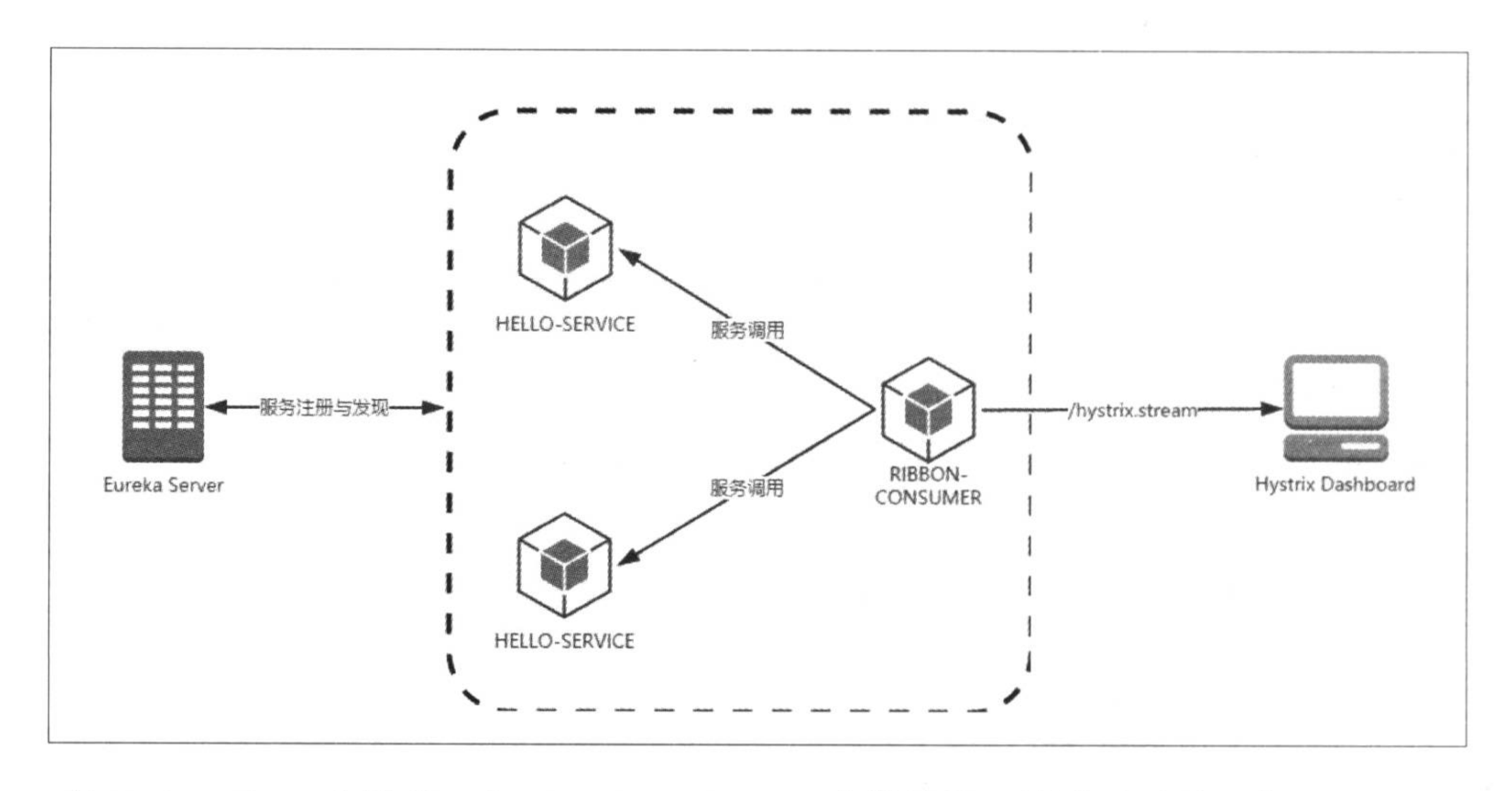
注意： metrics.rollingStats.timeInMilliseconds 参数的设置必须能够被 metrics.rolling-Stats.numBuckets 参数整除，不然将会抛出异常。

Hystrix 仪表盘

在断路器原理的介绍中，我们多次提到关于请求命令的度量指标的判断。这些度量指标都是 HystrixCommand 和 HystrixObservableCommand 实例在执行过程中记录的重要信息，它们除了在 Hystrix 断路器实现中使用之外，对于系统运维也有非常大的帮助。这些指标信息会以“滚动时间窗”与“桶”结合的方式进行汇总，并在内存中驻留一段时间，以供内部或外部进行查询使用，Hystrix 仪表盘就是这些指标内容的消费者之一。

通过之前的内容，我们已经体验到了 Spring Cloud 对 Hystrix 的优雅整合。除此之外，Spring Cloud 还完美地整合了它的仪表盘组件 Hystrix Dashboard，它主要用来实时监控 Hystrix 的各项指标信息。通过 Hystrix Dashboard 反馈的实时信息，可以帮助我们快速发现系统中存在的问题，从而及时地采取应对措施。

本节中我们将在 Hystrix 入门例子的基础上，构建一个 Hystrix Dashboard 来对 RIBBON-CONSUMER 实现监控，完成后的架构如下图所示。



在 Spring Cloud 中构建一个 Hystrix Dashboard 非常简单，只需要下面 4 步：

- 创建一个标准的 Spring Boot 工程，命名为 `hystrix-dashboard`。
- 编辑 `pom.xml`，具体依赖内容如下所示：

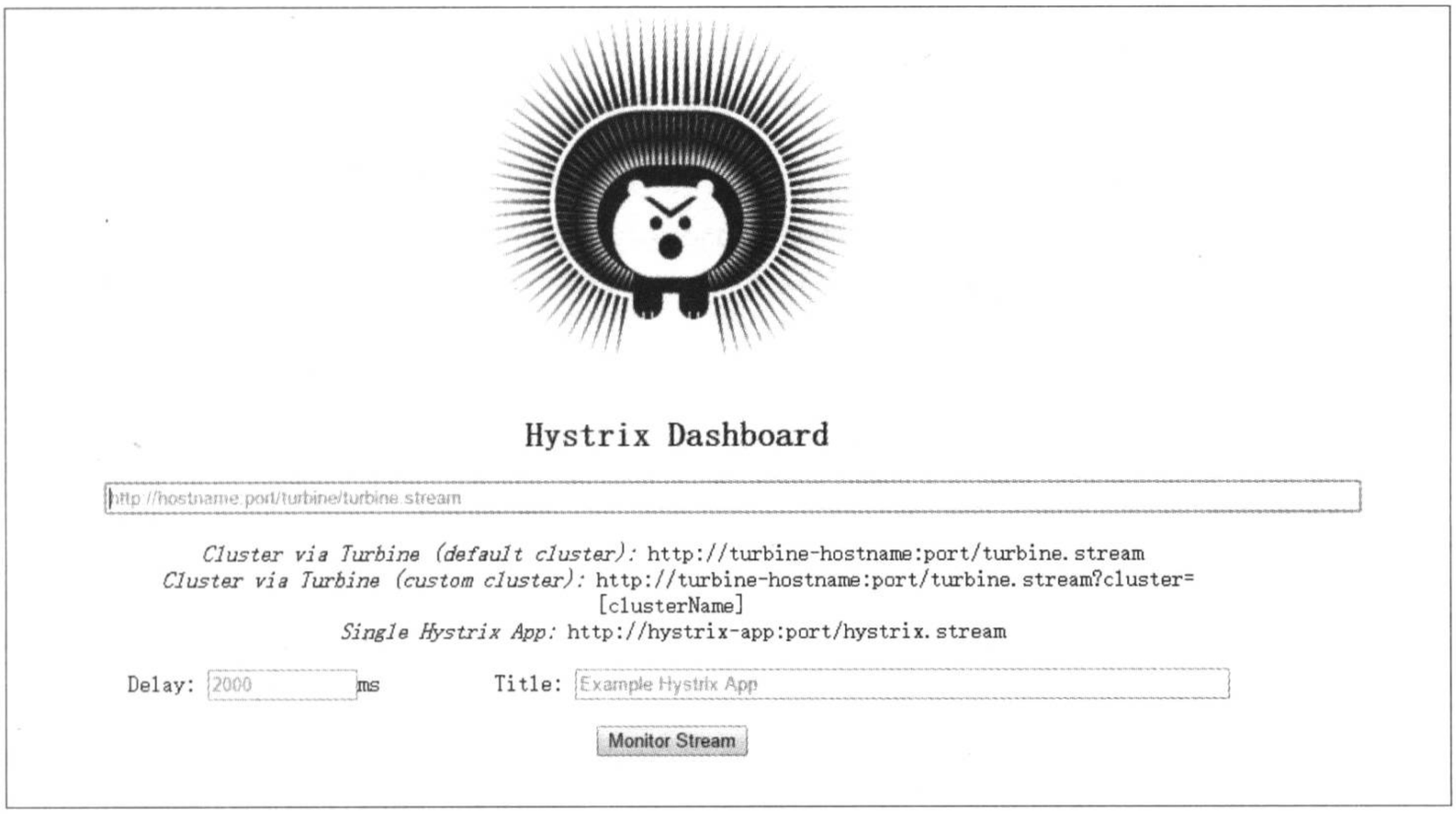
```
<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Brixton.SR5</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

- 为应用主类加上`@EnableHystrixDashboard`，启用 Hystrix Dashboard 功能。

- 根据实际情况修改 application.properties 配置文件，比如选择一个未被占用的端口等，此步不是必需的。

```
spring.application.name=hystrix-dashboard
server.port=2001
```

到这里我们已经完成了基本配置，接下来可以启动该应用，并访问 http://localhost:2001/hystrix。可以看到如下页面：



这是 Hystrix Dashboard 的监控首页，该页面中并没有具体的监控信息。从页面的文字内容中我们可以知道，Hystrix Dashboard 共支持三种不同的监控方式，如下所示。

- **默认的集群监控**：通过 URL http://turbine-hostname:port/turbine.stream 开启，实现对默认集群的监控。
- **指定的集群监控**：通过 URL http://turbine-hostname:port/turbine.stream?cluster=[clusterName]开启，实现对 clusterName 集群的监控。
- **单体应用的监控**：通过 URL http://hystrix-app:port/hystrix.stream 开启，实现对具体某个服务实例的监控。

前两者都是对集群的监控，需要整合 Turbine 才能实现，这部分内容我们将在下一节中做详细介绍。在本节中，我们主要实现对单个服务实例的监控，这里我们先来实现单个服务实例的监控。

既然 Hystrix Dashboard 监控单实例节点需要通过访问实例的/hystrix.stream 接口来实现，我们自然需要为服务实例添加这个端点，而添加该功能的步骤也同样简单，只需

要下面两步。

- 在服务实例pom.xml中的dependencies节点中新增spring-boot-starter-actuator 监控模块以开启监控相关的端点，并确保已经引入断路器的依赖spring-cloud-starter-hystrix:

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

- 确保在服务实例的主类中已经使用@EnableCircuitBreaker 注解，开启了断路器功能。

在为 RIBBON-CONSUMER 加入上面的配置之后，重启它的实例，此时我们可以在控制台中看到打印了大量的监控端点，其中/hystrix.stream 就是用于 Hystrix Dashboard 来展现监控信息的接口。

```
    o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/hystrix.stream/**]}" onto
public org.springframework.web.servlet.ModelAndView
org.springframework.cloud.netflix.endpoint.ServletWrappingEndpoint.handle(javax.ser
vlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse) throws
java.lang.Exception
```

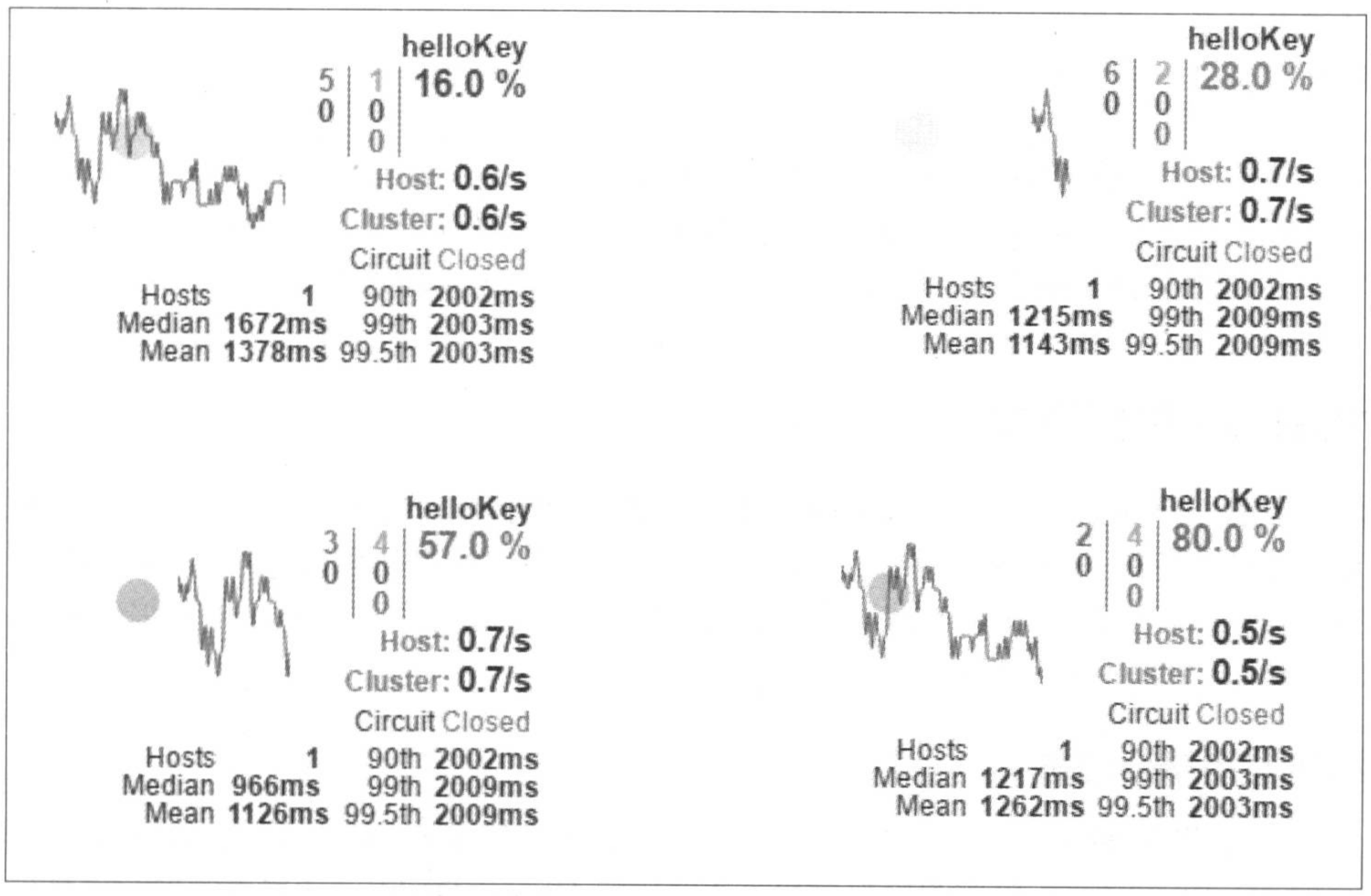
到这里已经完成了所有的配置，在 Hystrix Dashboard 的首页输入 http://localhost:9000/hystrix.stream，可以看到已启动对 RIBBON-CONSUMER 的监控，单击 Monitor Stream 按钮，可以看到如下页面。

在对该页面进行介绍之前，先看看在首页中我们还没有介绍的另外两个参数。

- `Delay`：该参数用来控制服务器上轮询监控信息的延迟时间，默认为 2000 毫秒，可以通过配置该属性来降低客户端的网络和 CPU 消耗。
- `Title`：该参数对应了上图头部标题 Hystrix Stream 之后的内容，默认会使用具体监控实例的 URL，可以通过配置该信息来展示更合适的标题。

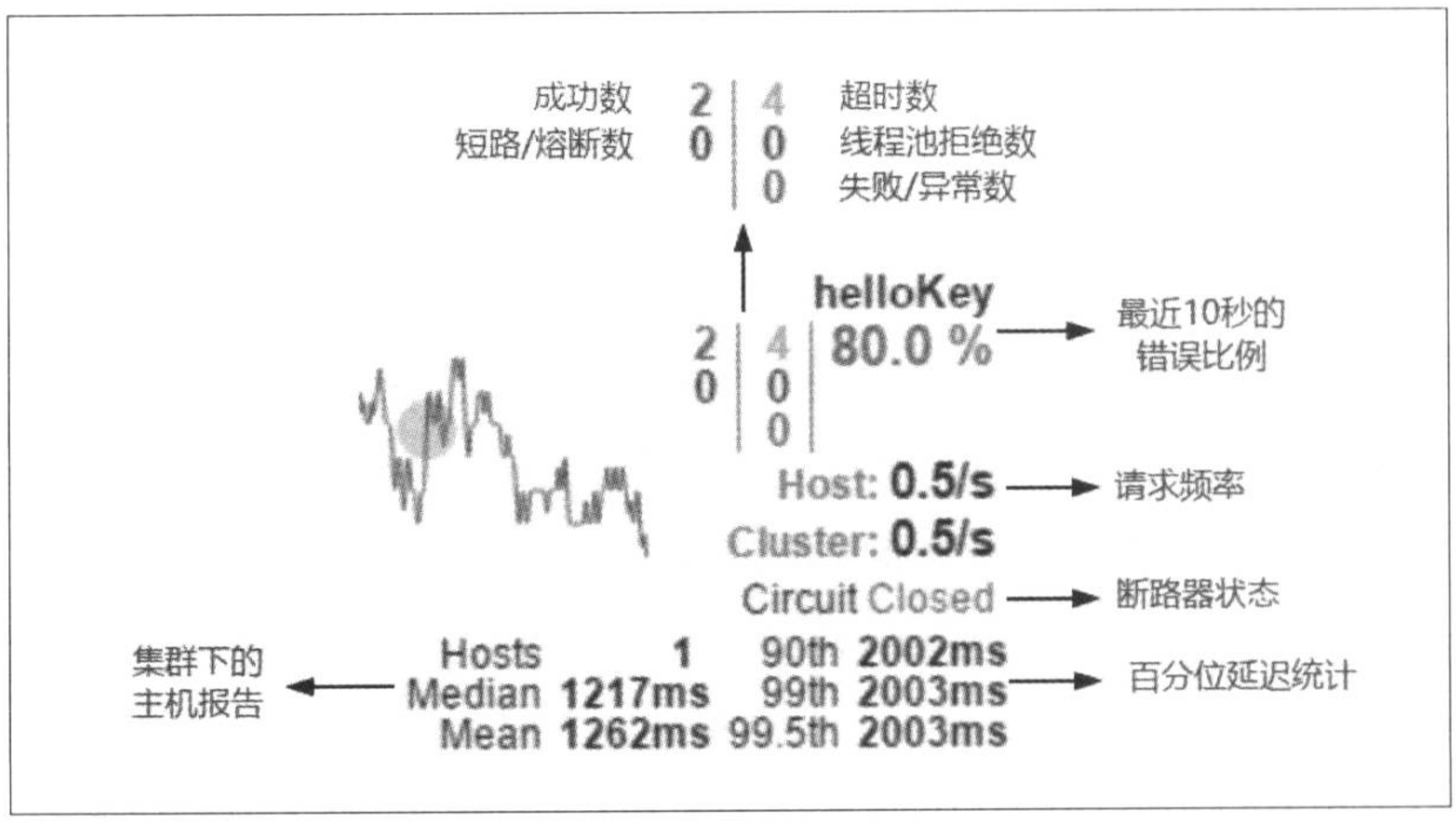
回到监控页面，我们来详细说说其中各元素的具体含义。

- 可以在监控信息的左上部找到两个重要的图形信息：一个实心圆和一条曲线。
  - 实心圆：其有两种含义。通过颜色的变化代表了实例的健康程度，如下图所示，它的健康度从绿色、黄色、橙色、红色递减。该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大。所以通过该实心圆的展示，我们可以在大量的实例中快速发现故障实例和高压力实例。



  - 曲线：用来记录 2 分钟内流量的相对变化，可以通过它来观察流量的上升和下降趋势。

- 其他一些数量指标如下图所示。



通过本节内容我们已经能够使用 Hystrix Dashboard 来对单个实例做信息监控了，但是在分布式系统中，往往有非常多的实例需要去维护和监控。到目前为止，我们能做的就是通过开启多个窗口来监控多个实例，很显然这样的做法并不合理。在下一节中，我们将介绍利用 Turbine 和 Hystrix Dashboard 配合实现对集群的监控。

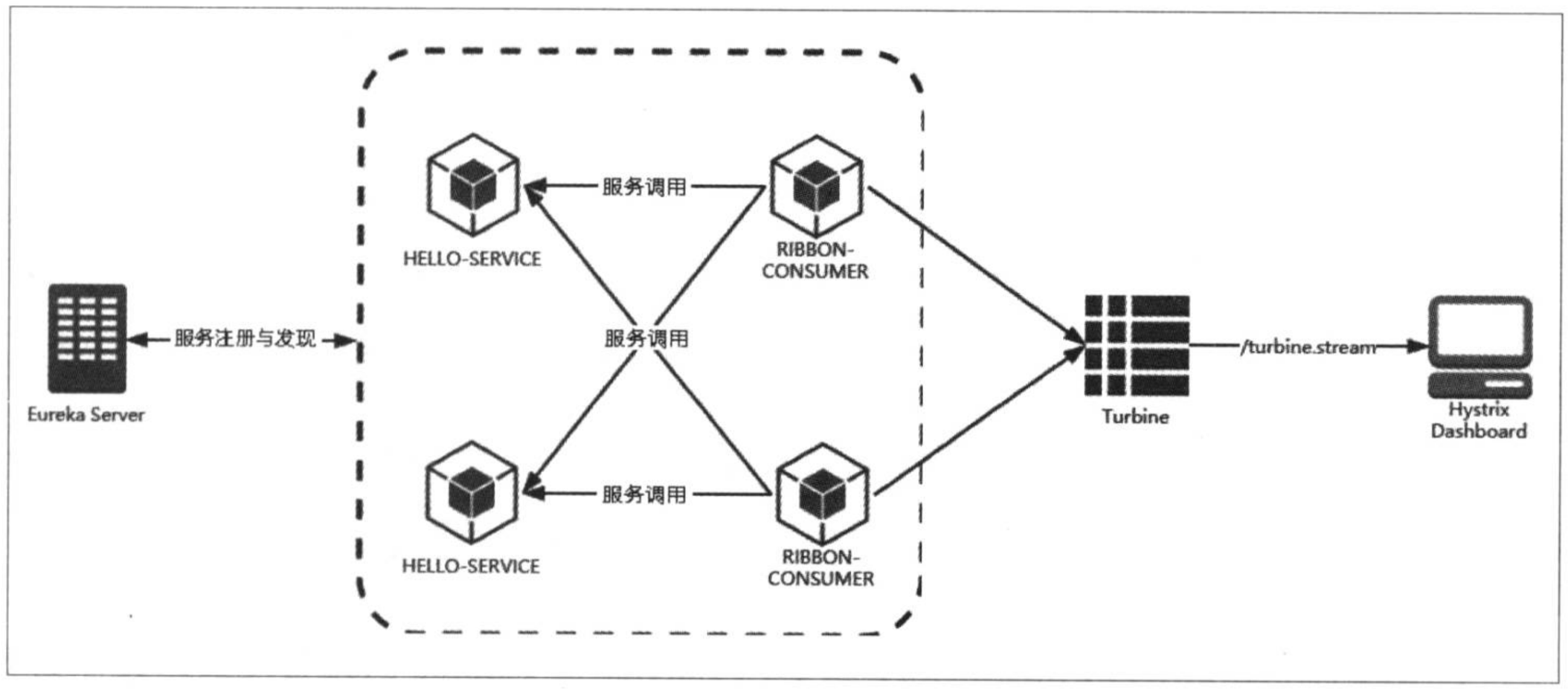
注意：当使用 Hystrix Board 来监控 Spring Cloud Zuul 构建的 API 网关时，Thread Pool 信息会一直处于 Loading 状态。这是由于 Zuul 默认会使用信号量来实现隔离，只有通过 Hystrix 配置把隔离机制改为线程池的方式才能够得以展示。

Turbine 集群监控

在上一节介绍 Hystrix Dashboard 的首页时，我们提到过除了可以开启单个实例的监控页面之外，还有一个监控端点`/turbine.stream`是对集群使用的。从端点的命名中，可猜测到这里我们将引入 Turbine，通过它来汇集监控信息，并将聚合后的信息提供给 Hystrix Dashboard 来集中展示和监控。

构建监控聚合服务

下面我们将在上一节内容的基础上做一些扩展，通过引入 Turbine 来聚合`RIBBON-CONSUMER`服务的监控信息，并输出给 Hystrix Dashboard 来进行展示，最后完成如下图所示的结构。



具体实现步骤如下：

- 创建一个标准的 Spring Boot 工程，命名为 turbine。
- 编辑 pom.xml，具体依赖内容如下所示。

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Brixton.SR5</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

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

- 创建应用主类 TurbineApplication，并使用@EnableTurbine 注解开启 Turbine。

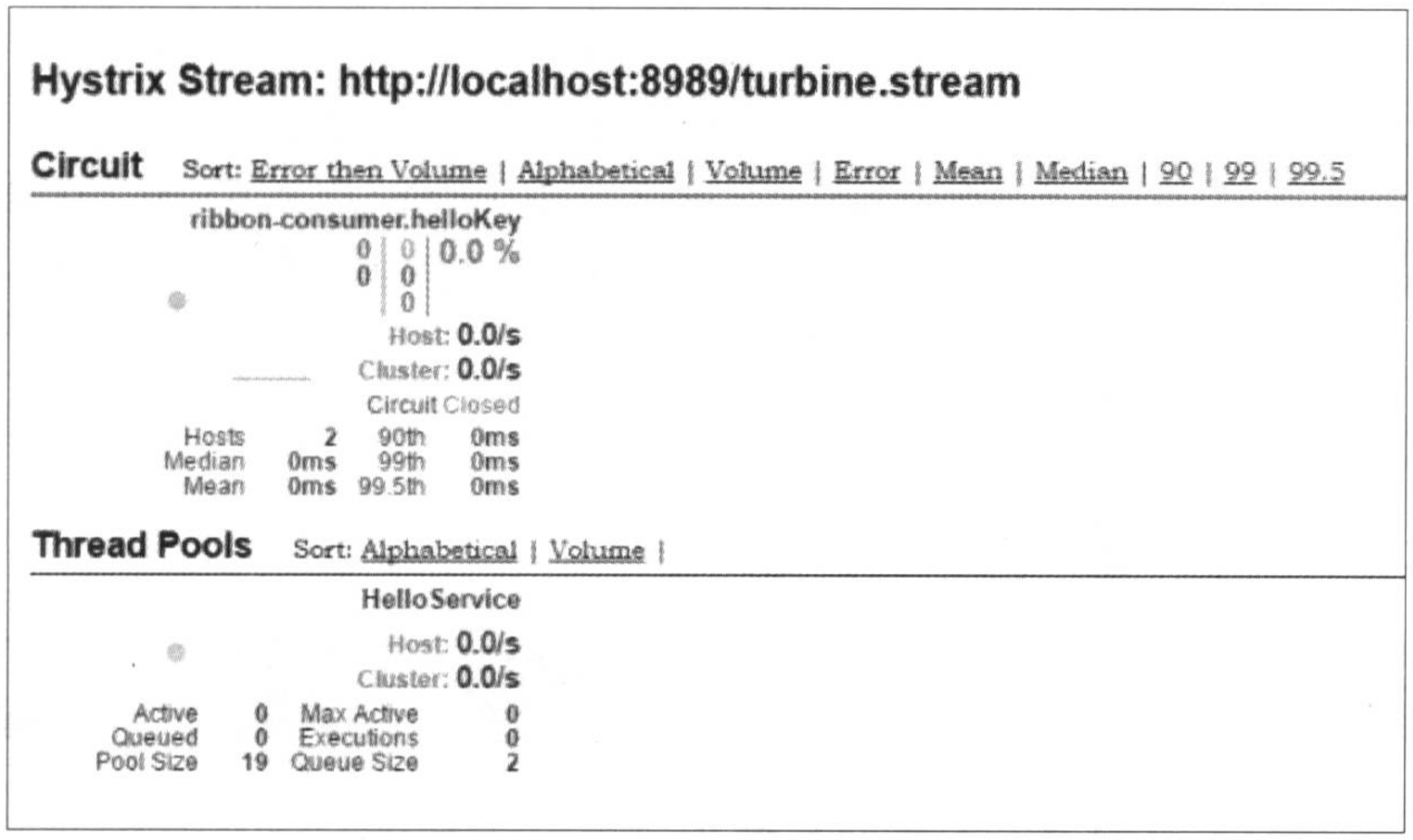
```
@EnableTurbine
@EnableDiscoveryClient
@SpringBootApplication
public class TurbineApplication {

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

}
```

在 application.properties 中加入 Eureka 和 Turbine 的相关配置，具体如下：

```
spring.application.name=turbine

server.port=8989
management.port=8990

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

turbine.app-config=RIBBON-CONSUMER
turbine.cluster-name-expression="default"
turbine.combine-host-port=true
```

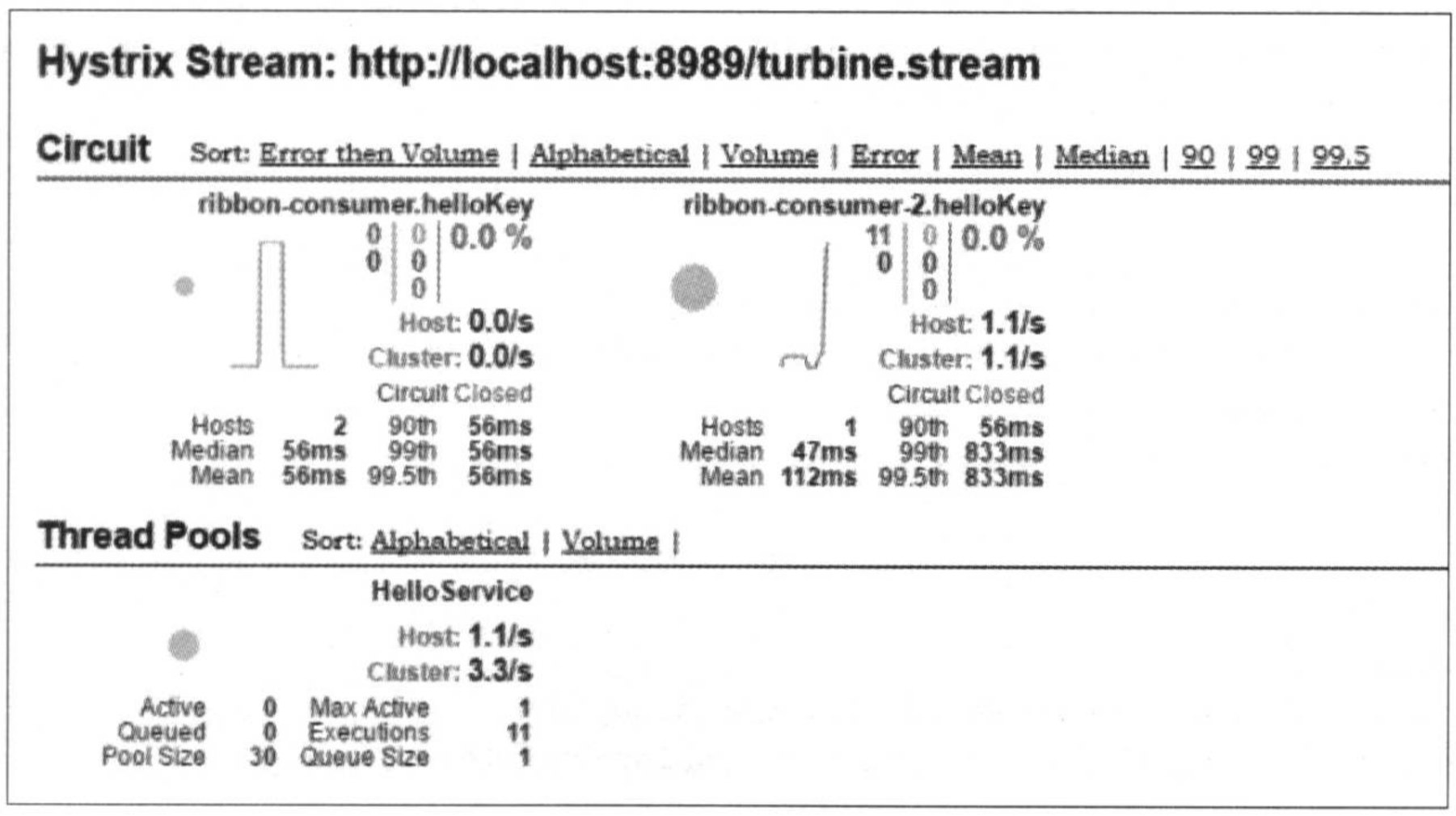
其中，turbine.app-config 参数指定了需要收集监控信息的服务名；turbine.cluster-name-expression 参数指定了集群名称为 default，当服务数量非常多的时候，可以启动多个 Turbine 服务来构建不同的聚合集群，而该参数可以用来区分这些不同的聚合集群，同时该参数值可以在 Hystrix 仪表盘中用来定位不同的聚合集群，只需在 Hystrix Stream 的 URL 中通过 cluster 参数来指定；turbine.combine-host-port 参数设置为 true，可以让同一主机上的服务通过主机名与端口号的组合来进行区分，默认情况下会以 host 来区分不同的服务，这会使得在本地调试的时候，本机上的不同服务聚合成一个服务来统计。

在完成了上面的内容构建之后，我们来体验一下 Turbine 对集群的监控能力。分别启动 eureka-server、HELLO-SERVICE、RIBBON-CONSUMER、Turbine 以及 Hystrix Dashboard。访问 Hystrix Dashboard，并开启对 http://localhost:8989/turbine.stream 的监控，我们可以看到如下页面：



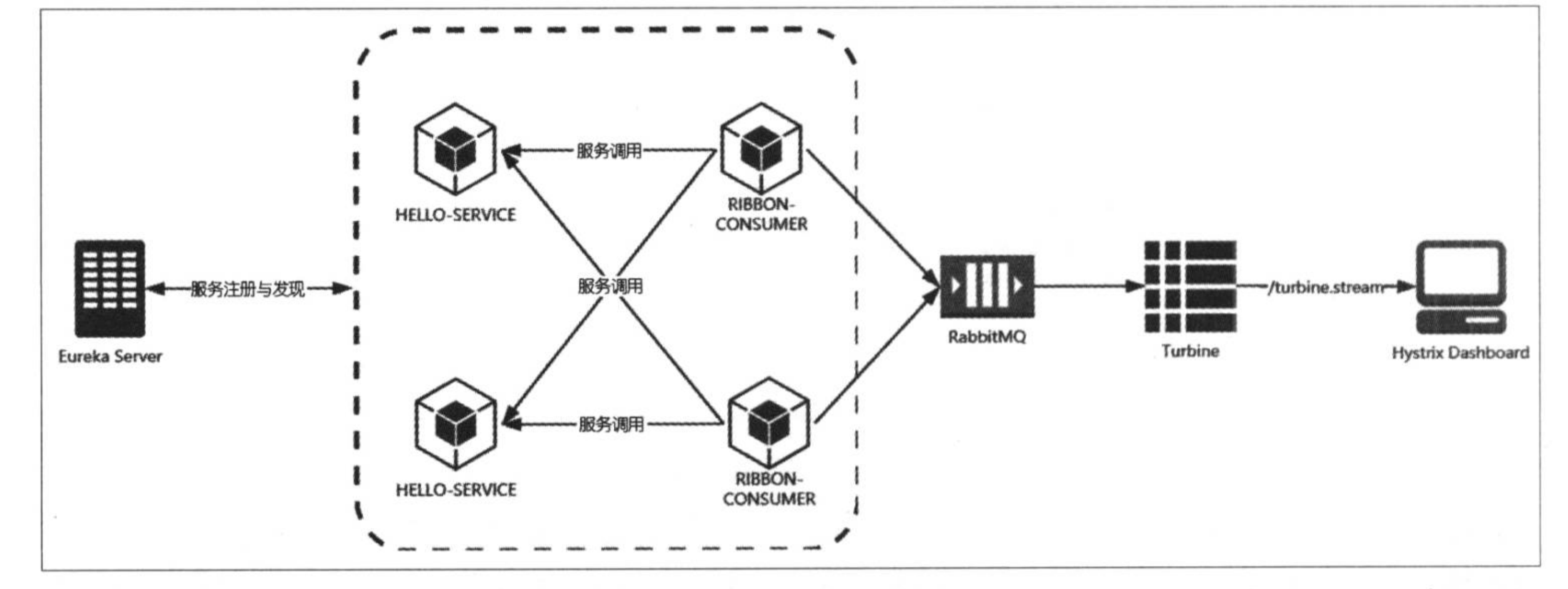
从图中可以看到，虽然我们如之前的架构那样启动了两个 RIBBON-CONSUMER，但是在监控页面中依然只是展示了一个监控图。不过仔细的读者可能已经发现，图中集群报告区域中的 Hosts 属性与之前尝试单机监控时已经有所不同。由此我们可以知道 RIBBON-CONSUMER 启动了两个实例，这里只展现了一个监控图，是由于这两个实例是同一个服务，而对于集群来说我们关注的是服务集群的高可用性，所以 Turbine 会将相同服务作为整体来看待，并汇总成一个监控图。

也可以为 RIBBON-CONSUMER 设置一个新的 spring.application.name，比如 ribbon-consumer-2，并启动它，此时我们就有两个服务会将监控信息输出给 Turbine 汇总了。刷新之前的监控页面，可以看到如下有两个监控图的页面：



与消息代理结合

Spring Cloud 在封装 Turbine 的时候，还封装了基于消息代理的收集实现。所以，我们可以将所有需要收集的监控信息都输出到消息代理中，然后 Turbine 服务再从消息代理中异步获取这些监控信息，最后将这些监控信息聚合并输出到 Hystrix Dashboard 中。通过引入消息代理，我们的 Turbine 和 Hystrix Dashboard 实现的监控架构可以改成如下图所示的结构。



从图中可以看到，这里多了一个重要元素 RabbitMQ。对于 RabbitMQ 的安装我们可以查看第 9 章的相关内容，这里不做过多说明。下面，我们来构建一个新的应用以实现基于消息代理的 Turbine 聚合服务，具体步骤如下所示。

- 创建一个标准的 Spring Boot 工程，命名为 `turbine-amqp`。
- 编辑 `pom.xml`，具体依赖内容如下所示：

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Brixton.SR5</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

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

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

可以看到这里主要引入了 spring-cloud-starter-turbine-amqp 依赖，它实际上包装了 spring-cloud-starter-turbine-stream 和 pring-cloud-starter-stream-rabbit。

注意：这里我们需要使用 Java 8 来运行。

- 在应用主类中使用@EnableTurbineStream 注解来启用 Turbine Stream 的配置。

```
@EnableTurbineStream
@EnableDiscoveryClient
@SpringBootApplication
public class TurbineApplication {

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

}
```

- 配置 application.properties 文件。

```
spring.application.name=turbine

server.port=8989
management.port=8990

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

对于 Turbine 的配置已经完成了，下面需要对服务消费者 RIBBON-CONSUMER 做一些修改，使其监控信息能够输出到 RabbitMQ 上。这个修改也非常简单，只需在 pom.xml 中增加对 spring-cloud-netflix-hystrix-amqp 的依赖，具体如下：

```
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-netflix-hystrix-amqp</artifactId>
    </dependency>
</dependencies>
```

在完成了上面的配置之后，继续启动 eureka-server、HELLO-SERVICE、RIBBON-CONSUMER、Turbine 以及 Hystrix Dashboard，同时确保 RabbitMQ 已在正常运行。访问 Hystrix Dashboard，并开启对 http://localhost:8989/turbine.stream 的监控，我们可以获得如之前实现的同样结果，只是这里的监控信息收集是通过消息代理异步实现的。

第 6 章

■■■

声明式服务调用：Spring Cloud Feign

通过前两章对 Spring Cloud Ribbon 和 Spring Cloud Hystrix 的介绍，我们已经掌握了开发微服务应用时的两个重磅武器，学会了如何在微服务架构中实现客户端负载均衡的服务调用以及如何通过断路器来保护我们的微服务应用。这两者将被作为基础工具类框架广泛地应用在各个微服务的实现中，不仅包括我们自身的业务类微服务，也包括一些基础设施类微服务（比如网关）。此外，在实践过程中，我们会发现对这两个框架的使用几乎是同时出现的。既然如此，那么是否有更高层次的封装来整合这两个基础工具以简化开发呢？本章我们即将介绍的 Spring Cloud Feign 就是这样一个工具。它基于 Netflix Feign 实现，整合了 Spring Cloud Ribbon 与 Spring Cloud Hystrix，除了提供这两者的强大功能之外，它还提供了一种声明式的 Web 服务客户端定义方式。

我们在使用 Spring Cloud Ribbon 时，通常都会利用它对 RestTemplate 的请求拦截来实现对依赖服务的接口调用，而 RestTemplate 已经实现了对 HTTP 请求的封装处理，形成了一套模板化的调用方法。在之前的例子中，我们只是简单介绍了 RestTemplate 调用的实现，但是在实际开发中，由于对服务依赖的调用可能不止于一处，往往一个接口会被多处调用，所以我们通常都会针对各个微服务自行封装一些客户端类来包装这些依赖服务的调用。这个时候我们会发现，由于 RestTemplate 的封装，几乎每一个调用都是简单的模板化内容。综合上述这些情况，Spring Cloud Feign 在此基础上做了进一步封装，由它来帮助我们定义和实现依赖服务接口的定义。在 Spring Cloud Feign 的实现下，我们只需创建一个接口并用注解的方式来配置它，即可完成对服务提供方的接口绑定，简化了在使用 Spring Cloud Ribbon 时自行封装服务调用客户端的开发量。Spring Cloud Feign 具备可插拔的注解支持，包括 Feign 注解和 JAX-RS 注解。同时，为了适应 Spring 的广大用户，它在 Netflix Feign

的基础上扩展了对 Spring MVC 的注解支持。这对于习惯于 Spring MVC 的开发者来说，无疑是一个好消息，因为这样可以大大减少学习使用它的成本。另外，对于 Feign 自身的一些主要组件，比如编码器和解码器等，它也以可插拔的方式提供，在有需求的时候我们可以方便地扩展和替换它们。

快速入门

在本节中，我们将通过一个简单的示例来展现 Spring Cloud Feign 在服务客户端定义上所带来的便利。下面的示例将继续使用之前我们实现的 `hello-service` 服务，这里我们会通过 Spring Cloud Feign 提供的声明式服务绑定功能来实现对该服务接口的调用。

- 首先，创建一个 Spring Boot 基础工程，取名为 `feign-consumer`，并在 `pom.xml` 中引入 `spring-cloud-starter-eureka` 和 `spring-cloud-starter-feign` 依赖。具体内容如下所示：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <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.cloud</groupId>
        <artifactId>spring-cloud-starter-feign</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
```

```
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Brixton.SR5</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- 创建应用主类 ConsumerApplication，并通过@EnableFeignClients 注解开启 Spring Cloud Feign 的支持功能。

```
@EnableFeignClients
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {

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

}
```

- 定义 HelloService 接口，通过@FeignClient 注解指定服务名来绑定服务，然后再使用 Spring MVC 的注解来绑定具体该服务提供的 REST 接口。

```
@FeignClient("hello-service")
public interface HelloService {

    @RequestMapping("/hello")
    String hello();

}
```

注意：这里服务名不区分大小写，所以使用 hello-service 和 HELLO-SERVICE 都是可以的。另外，在 Brixton.SR5 版本中，原有的 serviceId 属性已经被废弃，若要写属性名，可以使用 name 或 value。

- 接着，创建一个 ConsumerController 来实现对 Feign 客户端的调用。使用 @Autowired 直接注入上面定义的 HelloService 实例，并在 helloConsumer 函数中调用这个绑定了 hello-service 服务接口的客户端来向该服务发起 /hello 接口的调用。

```
@RestController
public class ConsumerController {

    @Autowired
    HelloService helloService;

    @RequestMapping(value = "/feign-consumer", method = RequestMethod.GET)
    public String helloConsumer() {
        return helloService.hello();
    }

}
```

- 最后，同 Ribbon 实现的服务消费者一样，需要在 application.properties 中指定服务注册中心，并定义自身的服务名为 feign-consumer，为了方便本地调试与之前的 Ribbon 消费者区分，端口使用 9001。

```
spring.application.name=feign-consumer
server.port=9001

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

测试验证

如之前验证 Ribbon 客户端负载均衡一样，我们先启动服务注册中心以及两个 HELLO-SERVICE，然后启动 FEIGN-CONSUMER，此时我们在 Eureka 信息面板中可看到如下内容：

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
FEIGN-CONSUMER	n/a (1)	(1)	UP (1) - PC-201602152056:feign-consumer:9001
HELLO-SERVICE	n/a (2)	(2)	UP (2) - PC-201602152056:hello-service:8081 , PC-201602152056:hello-service:8082

发送几次 GET 请求到 http://localhost:9001/feign-consumer，可以得到如之前 Ribbon 实现时一样的效果，正确返回了“Hello World”。并且根据控制台的输出，我们可以看到 Feign 实现的消费者，依然是利用 Ribbon 维护了针对 HELLO-SERVICE 的服务列表信息，并且通过轮询实现了客户端负载均衡。而与 Ribbon 不同的是，通过 Feign 我们只需定义服务绑定接口，以声明式的方法，优雅而简单地实现了服务调用。

参数绑定

在上一节的示例中，我们使用 Spring Cloud Feign 实现的是一个不带参数的 REST 服务绑定。然而现实系统中的各种业务接口要比它复杂得多，我们会在 HTTP 的各个位置传入各种不同类型的参数，并且在返回请求响应的时候也可能是一个复杂的对象结构。在本节

中，我们将详细介绍 Feign 中对几种不同形式参数的绑定方法。

在开始介绍 Spring Cloud Feign 的参数绑定之前，我们先扩展一下服务提供方 hello-service。增加下面这些接口定义，其中包含带有 Request 参数的请求、带有 Header 信息的请求、带有 RequestBody 的请求以及请求响应体中是一个对象的请求。

```
@RequestMapping(value = "/hello1", method = RequestMethod.GET)
public String hello(@RequestParam String name) {
    return "Hello " + name;
}

@RequestMapping(value = "/hello2", method = RequestMethod.GET)
public User hello(@RequestHeader String name, @RequestHeader Integer age) {
    return new User(name, age);
}

@RequestMapping(value = "/hello3", method = RequestMethod.POST)
public String hello(@RequestBody User user) {
    return "Hello "+ user.getName() + ", " + user.getAge();
}
```

User 对象的定义如下，这里省略了 getter 和 setter 函数，需要注意的是，这里必须要有 User 的默认构造函数。不然，Spring Cloud Feign 根据 JSON 字符串转换 User 对象时会抛出异常。

```
public class User {

    private String name;
    private Integer age;

    public User() {
    }

    public User(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    // 省略getter和setter

    @Override
    public String toString() {
        return "name=" + name + ", age=" +age;
    }

}
```

在完成了对 hello-service 的改造之后，下面我们开始在快速入门示例的 feign-consumer 应用中实现这些新增的请求的绑定。

- 首先，在 feign-consumer 中创建与上面一样的 User 类。
- 然后，在 HelloService 接口中增加对上述三个新增接口的绑定声明，修改后，完成的 HelloService 接口如下所示：

```
@FeignClient("HELLO-SERVICE")
public interface HelloService {

    @RequestMapping("/hello")
    String hello();

    @RequestMapping(value = "/hello1", method = RequestMethod.GET)
    String hello(@RequestParam("name") String name) ;

    @RequestMapping(value = "/hello2", method = RequestMethod.GET)
    User hello(@RequestHeader("name") String name, @RequestHeader("age") Integer
age);

    @RequestMapping(value = "/hello3", method = RequestMethod.POST)
    String hello(@RequestBody User user);

}
```

这里一定要注意，在定义各参数绑定时，@RequestParam、@RequestHeader 等可以指定参数名称的注解，它们的 value 千万不能少。在 Spring MVC 程序中，这些注解会根据参数名来作为默认值，但是在 Feign 中绑定参数必须通过 value 属性来指明具体的参数名，不然会抛出 IllegalStateException 异常，value 属性不能为空。

```
    Caused by: java.lang.IllegalStateException: RequestParam.value() was empty on
parameter 0
        at feign.Util.checkState(Util.java:128)
```

- 最后，在 ConsumerController 中新增一个/feign-consumer2 接口，来对本节新增的声明接口进行调用，修改后的完整代码如下所示：

```
@RestController
public class ConsumerController {

    @Autowired
    HelloService helloService;

    @RequestMapping(value = "/feign-consumer", method = RequestMethod.GET)
    public String helloConsumer() {
        return helloService.hello();
```

```
    }

    @RequestMapping(value = "/feign-consumer2", method = RequestMethod.GET)
    public String helloConsumer2() {
        StringBuilder sb = new StringBuilder();
        sb.append(helloService.hello()).append("\n");
        sb.append(helloService.hello("DIDI")).append("\n");
        sb.append(helloService.hello("DIDI", 30)).append("\n");
        sb.append(helloService.hello(new User("DIDI", 30))).append("\n");
        return sb.toString();
    }
}
```

测试验证

在完成上述改造之后，启动服务注册中心、两个 hello-service 服务以及我们改造过的 feign-consumer。通过发送 GET 请求到 http://localhost:9001/feign-consumer2，触发 HelloService 对新增接口的调用。最终，我们会获得如下输出，代表接口绑定和调用成功。

```
Hello World
Hello DIDI
name=DIDI, age=30
Hello DIDI, 30
```

继承特性

通过“快速入门”以及“参数绑定”小节中的示例实践，相信很多读者已经观察到，当使用 Spring MVC 的注解来绑定服务接口时，我们几乎完全可以从服务提供方的 Controller 中依靠复制操作，构建出相应的服务客户端绑定接口。既然存在这么多复制操作，我们自然需要考虑这部分内容是否可以得到进一步的抽象呢？在 Spring Cloud Feign 中，针对该问题提供了继承特性来帮助我们解决这些复制操作，以进一步减少编码量。下面，我们详细看看如何通过 Spring Cloud Feign 的继承特性来实现 REST 接口定义的复用。

- 为了能够复用 DTO 与接口定义，我们先创建一个基础的 Maven 工程，命名为 hello-service-api。
- 由于在 hello-service-api 中需要定义可同时复用于服务端与客户端的接口，我们要使用到 Spring MVC 的注解，所以在 pom.xml 中引入 spring-boot-starter-web 依赖，具体内容如下所示：

```
<groupId>com.didispace</groupId>
<artifactId>hello-service-api</artifactId>
```

```
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>hello-service-api</name>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.7.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

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

- 将上一节中实现的 User 对象复制到 hello-service-api 工程中，比如保存到 com.didispace.dto.User。
- 在 hello-service-api 工程中创建 com.didispace.service.HelloService 接口，内容如下，该接口中的 User 对象为本项目中的 com.didispace.dto.User。
- 将上一节中实现的 User 对象复制到 com.didispace.dto.User。创建 com.didispace.service.HelloService 接口，内容如下，该接口中的 User 对象为本项目中的 com.didispace.dto.User。

```
@RequestMapping("/refactor")
public interface HelloService {

    @RequestMapping(value = "/hello4", method = RequestMethod.GET)
    String hello(@RequestParam("name") String name) ;

    @RequestMapping(value = "/hello5", method = RequestMethod.GET)
    User hello(@RequestHeader("name") String name, @RequestHeader("age") Integer
age);
```

```
    @RequestMapping(value = "/hello6", method = RequestMethod.POST)
    String hello(@RequestBody User user);

}
```

因为后续还会通过之前的 hello-service 和 feign-consumer 来重构，所以为了避免接口混淆，在这里定义 HelloService 时，除了头部定义了/rafactor 前缀之外，同时将提供服务的三个接口更名为/hello4、/hello5、/hello6。

- 下面对 hello-service 进行重构，在 pom.xml 的 dependency 节点中，新增对 hello-service-api 的依赖。

```
<dependency>
    <groupId>com.didispace</groupId>
    <artifactId>hello-service-api</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

- 创建 RefactorHelloController 类继承 hello-service-api 中定义的 HelloService 接口，并参考之前的 HelloController 来实现这三个接口，具体内容如下所示：

```
@RestController
public class RefactorHelloController implements HelloService {

    @Override
    public String hello(@RequestParam("name") String name) {
        return "Hello " + name;
    }

    @Override
    public User hello(@RequestHeader("name") String name, @RequestHeader("age")
Integer age) {
        return new User(name, age);
    }

    @Override
    public String hello(@RequestBody User user) {
        return "Hello "+ user.getName() + ", " + user.getAge();
    }

}
```

我们可以看到通过继承的方式，在 Controller 中不再包含以往会定义的请求映射注解 @RequestMapping，而参数的注解定义在重写的时候会自动带过来。在这个类中，除了

要实现接口逻辑之外，只需再增加@RestController 注解使该类成为一个 REST 接口类就大功告成了。

- 完成了服务提供者的重构，接下来在服务消费者 feign-consumer 的 pom.xml 文件中，如在服务提供者中一样，新增对 hello-service-api 的依赖。
- 创建 RefactorHelloService 接口，并继承 hello-service-api 包中的 HelloService 接口，然后添加@FeignClient 注解来绑定服务。

```
@FeignClient(value = "HELLO-SERVICE")
public interface RefactorHelloService extends com.didispace.service.HelloService {

}
```

- 最后，在 ConsumerController 中，注入 RefactorHelloService 的实例，并新增一个请求/feign-consumer3 来触发对 RefactorHelloService 的实例的调用。

```
@Autowired
RefactorHelloService refactorHelloService;

@RequestMapping(value = "/feign-consumer3", method = RequestMethod.GET)
public String helloConsumer3() {
    StringBuilder sb = new StringBuilder();
    sb.append(refactorHelloService.hello("MIMI")).append("\n");
    sb.append(refactorHelloService.hello("MIMI", 20)).append("\n");
    sb.append(refactorHelloService.hello(new com.didispace.dto.User("MIMI",
20))).append("\n");
    return sb.toString();
}
```

测试验证

这次的验证过程需要注意几个工程的构建顺序，由于 hello-service 和 feign-consumer 都依赖 hello-service-api 工程中的接口和 DTO 定义，所以必须先构建hello-service-api 工程，然后再构建hello-service 和 feign-consumer。接着我们分别启动服务注册中心，hello-service 和 feign-consumer，并访问 http://localhost:9001/feign-consumer3，调用成功后可以获得如下输出：

```
Hello MIMI
name=MIMI, age=20
Hello MIMI, 20
```

优点与缺点

使用 Spring Cloud Feign 继承特性的优点很明显，可以将接口的定义从 Controller 中剥

离，同时配合 Maven 私有仓库就可以轻易地实现接口定义的共享，实现在构建期的接口绑定，从而有效减少服务客户端的绑定配置。这么做虽然可以很方便地实现接口定义和依赖的共享，不用再复制粘贴接口进行绑定，但是这样的做法使用不当的话会带来副作用。由于接口在构建期间就建立起了依赖，那么接口变动就会对项目构建造成影响，可能服务提供方修改了一个接口定义，那么会直接导致客户端工程的构建失败。所以，如果开发团队通过此方法来实现接口共享的话，建议在开发评审期间严格遵守面向对象的开闭原则，尽可能地做好前后版本的兼容，防止牵一发而动全身的后果，增加团队不必要的维护工作量。

Ribbon 配置

由于 Spring Cloud Feign 的客户端负载均衡是通过 Spring Cloud Ribbon 实现的，所以我们可以直接通过配置 Ribbon 客户端的方式来自定义各个服务客户端调用的参数。那么我们如何在使用 Spring Cloud Feign 的工程中使用 Ribbon 的配置呢？

全局配置

全局配置的方法非常简单，我们可以直接使用 `ribbon.<key>=<value>`的方式来设置 ribbon 的各项默认参数。比如，修改默认的客户端调用超时时间：

```
ribbon.ConnectTimeout=500
ribbon.ReadTimeout=5000
```

指定服务配置

大多数情况下，我们对于服务调用的超时时间可能会根据实际服务的特性做一些调整，所以仅仅依靠默认的全局配置是不行的。在使用 Spring Cloud Feign 的时候，针对各个服务客户端进行个性化配置的方式与使用 Spring Cloud Ribbon 时的配置方式是一样的，都采用 `<client>.ribbon.key=value` 的格式进行设置。但是，这里就有一个疑问了，`<client>`所指代的 Ribbon 客户端在哪里呢？

回想一下，在定义 Feign 客户端的时候，我们使用了`@FeignClient` 注解。在初始化过程中，Spring Cloud Feign 会根据该注解的 `name` 属性或 `value` 属性指定的服务名，自动创建一个同名的 Ribbon 客户端。也就是说，在之前的示例中，使用`@FeignClient(value = "HELLO-SERVICE")`来创建 Feign 客户端的时候，同时也创建了一个名为 `HELLO-SERVICE` 的 Ribbon 客户端。既然如此，我们就可以使用`@FeignClient` 注解中的 `name` 或 `value` 属性值来设置对应的 Ribbon 参数，比如：

```
HELLO-SERVICE.ribbon.ConnectTimeout=500
HELLO-SERVICE.ribbon.ReadTimeout=2000
HELLO-SERVICE.ribbon.OkToRetryOnAllOperations=true
```

```
HELLO-SERVICE.ribbon.MaxAutoRetriesNextServer=2
HELLO-SERVICE.ribbon.MaxAutoRetries=1
```

重试机制

在 Spring Cloud Feign 中默认实现了请求的重试机制，而上面我们对于 HELLO-SERVICE 客户端的配置内容就是对于请求超时以及重试机制配置的详情，具体内容可参考第 4 章最后一节关于 Spring Cloud Ribbon 重试机制的介绍。我们可以通过修改之前的示例做一些验证。

- 在 hello-service 应用的/hello 接口实现中，增加一些随机延迟，比如：

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello() throws Exception {
    ServiceInstance instance = client.getLocalServiceInstance()    ;
    // 测试超时
    int sleepTime = new Random().nextInt(3000);
    logger.info("sleepTime:" + sleepTime);
    Thread.sleep(sleepTime);

    logger.info("/hello, host:" + instance.getHost() + ", service_id:" +
instance.getServiceId());
    return "Hello World";
}
```

- 在 feign-consumer 应用中增加上文中提到的重试配置参数。其中，由于 HELLO-SERVICE.ribbon.MaxAutoRetries 设置为 1，所以重试策略先尝试访问首选实例一次，失败后才更换实例访问，而更换实例访问的次数通过 HELLO-SERVICE.ribbon.MaxAutoRetriesNextServer 参数设置为 2，所以会尝试更换两次实例进行重试。
- 最后，启动这些应用，并尝试访问几次 http://localhost:9001/feign-consumer 接口。当请求发生超时的时候，我们在 hello-service 的控制台中可能会获得如下输出内容（由于 sleepTime 的随机性，并不一定每次相同）：

```
INFO 19264 --- [nio-8001-exec-6] com.didispace.web.HelloController      : sleepTime:2929
INFO 19264 --- [nio-8001-exec-7] com.didispace.web.HelloController      : sleepTime:253
INFO 19264 --- [nio-8001-exec-7] com.didispace.web.HelloController      : /hello, host:
192.168.0.105, service_id:hello-service
INFO 19264 --- [nio-8001-exec-6] com.didispace.web.HelloController      : /hello, host:
192.168.0.105, service_id:hello-service
```

从控制台输出中，我们可以看到这次访问的第一次请求延迟时间为 2929 毫秒，由于超时时间设置为 2000 毫秒，Feign 客户端发起了重试，第二次请求的延迟为 253 秒，

没有超时。Feign 客户端在进行服务调用时，虽然经历了一次失败，但是通过重试机制，最终还是获得了请求结果。所以，对于重试机制的实现，对于构建高可用的服务集群来说非常重要，而 Spring Cloud Feign 也为其提供了足够的支持。

这里需要注意一点，Ribbon 的超时与 Hystrix 的超时是两个概念。为了让上述实现有效，我们需要让 Hystrix 的超时时间大于 Ribbon 的超时时间，否则 Hystrix 命令超时后，该命令直接熔断，重试机制就没有任何意义了。

Hystrix 配置

在 Spring Cloud Feign 中，除了引入了用于客户端负载均衡的 Spring Cloud Ribbon 之外，还引入了服务保护与容错的工具 Hystrix。默认情况下，Spring Cloud Feign 会为将所有 Feign 客户端的方法都封装到 Hystrix 命令中进行服务保护。在上一节末尾，我们介绍重试机制的配置时，也提到了关于 Hystrix 的超时时间配置。那么在本节中，我们就来详细介绍一下，如何在使用 Spring Cloud Feign 时配置 Hystrix 属性以及如何实现服务降级。

全局配置

对于 Hystrix 的全局配置同 Spring Cloud Ribbon 的全局配置一样，直接使用它的默认配置前缀 `hystrix.command.default` 就可以进行设置，比如设置全局的超时时间：

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=5000
```

另外，在对 Hystrix 进行配置之前，我们需要确认 `feign.hystrix.enabled` 参数没有被设置为 `false`，否则该参数设置会关闭 Feign 客户端的 Hystrix 支持。而对于我们之前测试重试机制时，对于 Hystrix 的超时时间控制除了可以使用上面的配置来增加熔断超时时间，也可以通过 `feign.hystrix.enabled=false` 来关闭 Hystrix 功能，或者使用 `hystrix.command.default.execution.timeout.enabled=false` 来关闭熔断功能。

禁用 Hystrix

上文我们提到了，在 Spring Cloud Feign 中，可以通过 `feign.hystrix.enabled=false` 来关闭 Hystrix 功能。另外，如果不想全局地关闭 Hystrix 支持，而只想针对某个服务客户端关闭 Hystrix 支持时，需要通过使用`@Scope("prototype")`注解为指定的客户端配置 `Feign.Builder` 实例，详细实现步骤如下所示。

- 构建一个关闭 Hystrix 的配置类。

```
@Configuration
```

```
public class DisableHystrixConfiguration {

    @Bean
    @Scope("prototype")
    public Feign.Builder feignBuilder() {
        return Feign.builder();
    }

}
```

- 在 HelloService 的@FeignClient 注解中，通过 configuration 参数引入上面实现的配置。

```
@FeignClient(name="HELLO-SERVICE", configuration = DisableHystrixConfiguration.class)
public interface HelloService {
    ...
}
```

指定命令配置

对于 Hystrix 命令的配置，在实际应用时往往也会根据实际业务情况制定出不同的配置方案。配置方法也跟传统的 Hystrix 命令的参数配置相似，采用 hystrix.command.<commandKey>作为前缀。而<commandKey>默认情况下会采用 Feign 客户端中的方法名作为标识，所以，针对上一节介绍的尝试机制中对/hello 接口的熔断超时时间的配置可以通过其方法名作为<commandKey>来进行配置，具体如下：

```
hystrix.command.hello.execution.isolation.thread.timeoutInMilliseconds=5000
```

在使用指定命令配置的时候，需要注意，由于方法名很有可能重复，这个时候相同方法名的 Hystrix 配置会共用，所以在进行方法定义与配置的时候需要做好一定的规划。当然，也可以重写 Feign.Builder 的实现，并在应用主类中创建它的实例来覆盖自动化配置的 HystrixFeign.Builder 实现。

服务降级配置

Hystrix 提供的服务降级是服务容错的重要功能，由于 Spring Cloud Feign 在定义服务客户端的时候与 Spring Cloud Ribbon 有很大差别，HystrixCommand 定义被封装了起来，我们无法像之前介绍 Spring Cloud Hystrix 时，通过@HystrixCommand 注解的 fallback 参数那样来指定具体的服务降级处理方法。但是，Spring Cloud Feign 提供了另外一种简单的定义方式，下面我们在之前创建的 feign-consumer 工程中进行改造。

- 服务降级逻辑的实现只需要为Feign客户端的定义接口编写一个具体的接口实现类。比如为 HelloService 接口实现一个服务降级类 HelloServiceFallback，其中每个重写方法的实现逻辑都可以用来定义相应的服务降级逻辑，具体如下：

```
@Component
public class HelloServiceFallback implements HelloService {

    @Override
    public String hello() {
        return "error";
    }

    @Override
    public String hello(@RequestParam("name") String name) {
        return "error";
    }

    @Override
    public User hello(@RequestHeader("name") String name, @RequestHeader("age")
Integer age) {
        return new User("未知", 0);
    }

    @Override
    public String hello(@RequestBody User user) {
        return "error";
    }

}
```

- 在服务绑定接口 HelloService 中，通过@FeignClient 注解的 fallback 属性来指定对应的服务降级实现类。

```
@FeignClient(name="HELLO-SERVICE", fallback = HelloServiceFallback.class)
public interface HelloService {

    @RequestMapping("/hello")
    String hello();

    @RequestMapping(value = "/hello1", method = RequestMethod.GET)
    String hello(@RequestParam("name") String name) ;

    @RequestMapping(value = "/hello2", method = RequestMethod.GET)
    User hello(@RequestHeader("name") String name, @RequestHeader("age") Integer
age);
```

```
    @RequestMapping(value = "/hello3", method = RequestMethod.POST)
    String hello(@RequestBody User user);

}
```

测试验证

下面我们来验证一下服务降级逻辑的实现。启动服务注册中心和 feign- consumer，但是不启动 hello-service 服务。发送 GET 请求到 http://localhost:9001/feign-consumer2，该接口会分别调用 HelloService 中的 4 个绑定接口，但因为 hello-service 服务没有启动，会直接触发服务降级，并获得下面的输出内容：

```
error
error
name=未知, age=0
error
```

正如我们在 HelloServiceFallback 类中实现的内容，每一个服务接口的断路器实际就是实现类中的重写函数的实现。

注意：在 Brixton.SR5 版本中，fallback 的实现函数中不再支持返回 com.netflix.hystrix.HystrixCommand 和 rx.Observable 类型的异步执行方式和响应式执行方式。

其他配置

请求压缩

Spring Cloud Feign 支持对请求与响应进行 GZIP 压缩，以减少通信过程中的性能损耗。我们只需通过下面两个参数设置，就能开启请求与响应的压缩功能：

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
```

同时，我们还能对请求压缩做一些更细致的设置，比如下面的配置内容指定了压缩的请求数据类型，并设置了请求压缩的大小下限，只有超过这个大小的请求才会对其进行压缩。

```
feign.compression.request.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

上述配置的 feign.compression.request.mime-types 和 feign.compression.request.min-request-size 均为默认值。

日志配置

Spring Cloud Feign 在构建被@FeignClient 注解修饰的服务客户端时，会为每一个客户端都创建一个 feign.Logger 实例，我们可以利用该日志对象的 DEBUG 模式来帮助分析 Feign 的请求细节。可以在 application.properties 文件中使用 logging.level.<FeignClient>的参数配置格式来开启指定 Feign 客户端的 DEBUG 日志，其中<FeignClient>为 Feign 客户端定义接口的完整路径，比如针对本章中我们实现的 HelloService 可以按如下配置开启：

```
logging.level.com.didispace.web.HelloService=DEBUG
```

但是，只是添加了如上配置，还无法实现对 DEBUG 日志的输出。这时由于 Feign 客户端默认的 Logger.Level 对象定义为 NONE 级别，该级别不会记录任何 Feign 调用过程中的信息，所以我们需要调整它的级别，针对全局的日志级别，可以在应用主类中直接加入 Logger.Level 的 Bean 创建，具体如下：

```
@EnableFeignClients
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {

    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }

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

}
```

当然也可以通过实现配置类，然后在具体的 Feign 客户端来指定配置类以实现是否要调整不同的日志级别，比如下面的实现：

```
@Configuration
public class FullLogConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
```

```
}

@FeignClient(name="HELLO-SERVICE", configuration = FullLogConfiguration.class)
public interface HelloService {
    ...
}
```

在调整日志级别为 FULL 之后，我们可以再访问一下之前的 http://localhost:9001/feign-consumer 接口，这时我们在 feign-consumer 的控制台中就可以看到类似下面的请求详细日志：

```
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] ---> GET http://HELLO-SERVICE/hello HTTP/1.1
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] ---> END HTTP (0-byte body)
    ...
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] <--- HTTP/1.1 200 OK (3342ms)
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Server: Apache-Coyote/1.1
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Content-Length: 11
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] X-Application-Context: hello-service:8001
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Date: Sat, 07 Jan 2017 11:49:11 GMT
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Content-Type: text/plain;charset=UTF-8
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello]
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Hello World
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] <--- END HTTP (11-byte body)
```

对于 Feign 的 Logger 级别主要有下面 4 类，可根据实际需要进行调整使用。

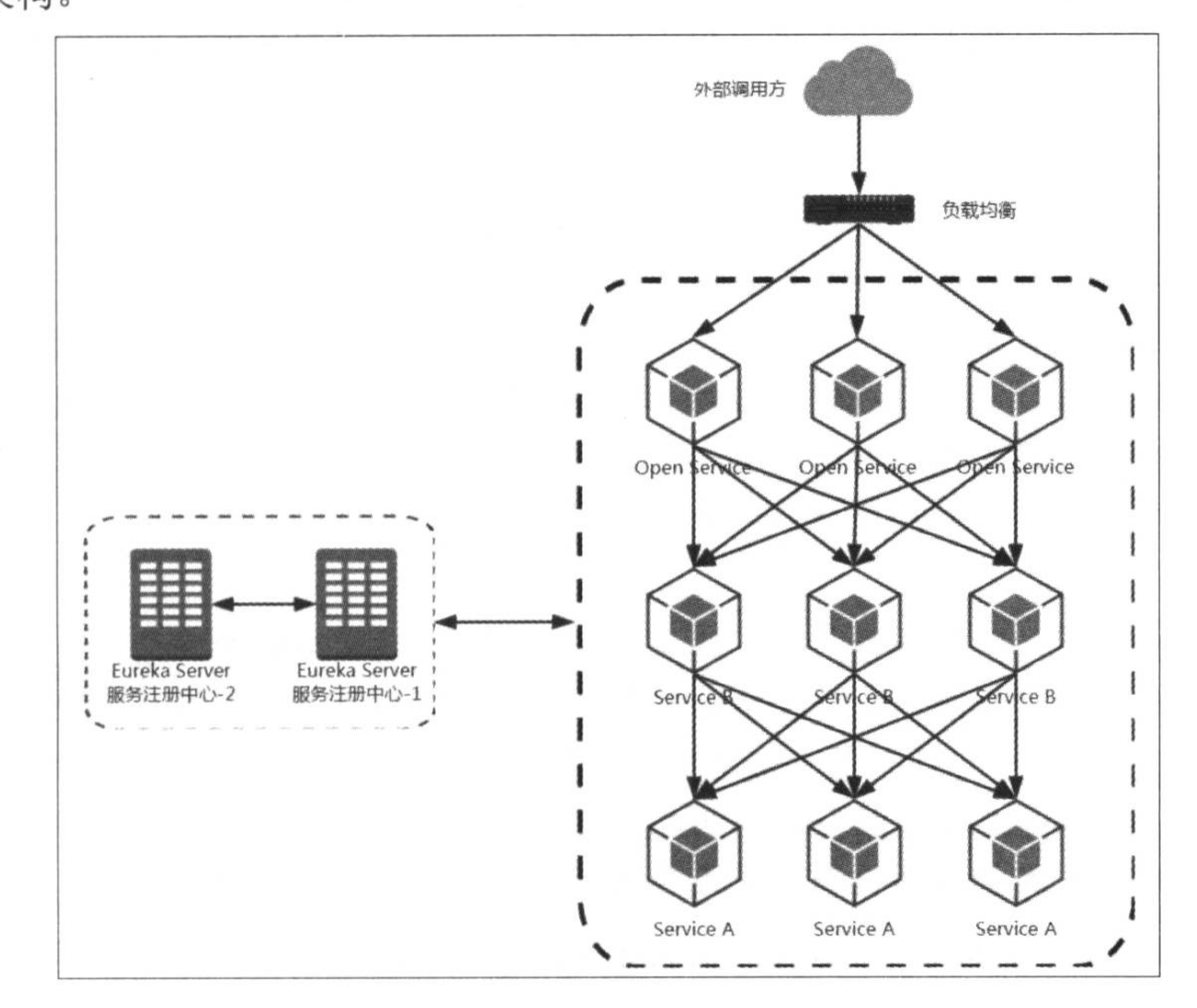
- NONE：不记录任何信息。
- BASIC：仅记录请求方法、URL 以及响应状态码和执行时间。
- HEADERS：除了记录 BASIC 级别的信息之外，还会记录请求和响应的头信息。
- FULL：记录所有请求与响应的明细，包括头信息、请求体、元数据等。

第 7 章

■■■

API 网关服务：Spring Cloud Zuul

通过前几章的介绍，我们对于 Spring Cloud Netflix 下的核心组件已经了解了一大半。这些组件基本涵盖了微服务架构中最为基础的几个核心设施，利用这些组件我们已经可以构建起一个简单的微服务架构系统，比如，通过使用 Spring Cloud Eureka 实现高可用的服务注册中心以及实现微服务的注册与发现；通过 Spring Cloud Ribbon 或 Feign 实现服务间负载均衡的接口调用；同时，为了使分布式系统更为健壮，对于依赖的服务调用使用 Spring Cloud Hystrix 来进行包装，实现线程隔离并加入熔断机制，以避免在微服务架构中因个别服务出现异常而引起级联故障蔓延。通过上述思路，我们可以设计出类似下图的基础系统架构。



在该架构中，我们的服务集群包含内部服务 Service A 和 Service B，它们都会向 Eureka Server 集群进行注册与订阅服务，而 Open Service 是一个对外的 RESTful API 服务，它通过 F5、Nginx 等网络设备或工具软件实现对各个微服务的路由与负载均衡，并公开给外部的客户端调用。

在本章中，我们将把视线聚焦在对外服务这块内容，通常也称为边缘服务。首先需要肯定的是，上面的架构实现系统功能是完全没有问题的，但是我们还是可以进一步思考一下，这样的架构是否还有不足的地方会使运维人员或开发人员感到痛苦。

首先，我们从运维人员的角度来看看，他们平时都需要做一些什么工作来支持这样的架构。当客户端应用单击某个功能的时候往往会发出一些对微服务获取资源的请求到后端，这些请求通过 F5、Nginx 等设施的路由和负载均衡分配后，被转发到各个不同的服务实例上。而为了让这些设施能够正确路由与分发请求，运维人员需要手工维护这些路由规则与服务实例列表，当有实例增减或是 IP 地址变动等情况发生的时候，也需要手工地去同步修改这些信息以保持实例信息与中间件配置内容的一致性。在系统规模不大的时候，维护这些信息的工作还不会太过复杂，但是如果当系统规模不断增大，那么这些看似简单的维护任务会变得越来越难，并且出现配置错误的概率也会逐渐增加。很显然，这样的做法并不可取，所以我们需要一套机制来有效降低维护路由规则与服务实例列表的难度。

其次，我们再从开发人员的角度来看看，在这样的架构下，会产生一些怎样的问题呢？大多数情况下，为了保证对外服务的安全性，我们在服务端实现的微服务接口，往往都会有一定的权限校验机制，比如对用户登录状态的校验等；同时为了防止客户端在发起请求时被篡改等安全方面的考虑，还会有一些签名校验的机制存在。这时候，由于使用了微服务架构的理念，我们将原本处于一个应用中的多个模块拆成了多个应用，但是这些应用提供的接口都需要这些校验逻辑，我们不得不在这些应用中都实现这样一套校验逻辑。随着微服务规模的扩大，这些校验逻辑的冗余变得越来越多，突然有一天我们发现这套校验逻辑有个 BUG 需要修复，或者需要对其做一些扩展和优化，此时我们就不得不去每个应用里修改这些逻辑，而这样的修改不仅会引起开发人员的抱怨，更会加重测试人员的负担。所以，我们也需要一套机制能够很好地解决微服务架构中，对于微服务接口访问时各前置校验的冗余问题。

为了解决上面这些常见的架构问题，API 网关的概念应运而生。API 网关是一个更为智能的应用服务器，它的定义类似于面向对象设计模式中的 Facade 模式，它的存在就像是整个微服务架构系统的门面一样，所有的外部客户端访问都需要经过它来进行调度和过滤。它除了要实现请求路由、负载均衡、校验过滤等功能之外，还需要更多能力，比如与服务治理框架的结合、请求转发时的熔断机制、服务的聚合等一系列高级功能。

既然 API 网关对于微服务架构这么重要，那么在 Spring Cloud 中是否有相应的解决方

案呢？答案是很肯定的，在 Spring Cloud 中了提供了基于 Netflix Zuul 实现的 API 网关组件——Spring Cloud Zuul。那么，它是如何解决上面这两个普遍问题的呢？

首先，对于路由规则与服务实例的维护问题。Spring Cloud Zuul 通过与 Spring Cloud Eureka 进行整合，将自身注册为 Eureka 服务治理下的应用，同时从 Eureka 中获得了所有其他微服务的实例信息。这样的设计非常巧妙地将服务治理体系中维护的实例信息利用起来，使得将维护服务实例的工作交给了服务治理框架自动完成，不再需要人工介入。而对于路由规则的维护，Zuul 默认会将通过以服务名作为 ContextPath 的方式来创建路由映射，大部分情况下，这样的默认设置已经可以实现我们大部分的路由需求，除了一些特殊情况（比如兼容一些老的 URL）还需要做一些特别的配置。但是相比于之前架构下的运维工作量，通过引入 Spring Cloud Zuul 实现 API 网关后，已经能够大大减少了。

其次，对于类似签名校验、登录校验在微服务架构中的冗余问题。理论上来说，这些校验逻辑在本质上与微服务应用自身的业务并没有多大的关系，所以它们完全可以独立成一个单独的服务存在，只是它们被剥离和独立出来之后，并不是给各个微服务调用，而是在 API 网关服务上进行统一调用来对微服务接口做前置过滤，以实现对微服务接口的拦截和校验。Spring Cloud Zuul 提供了一套过滤器机制，它可以很好地支持这样的任务。开发者可以通过使用Zuul来创建各种校验过滤器，然后指定哪些规则的请求需要执行校验逻辑，只有通过校验的才会被路由到具体的微服务接口，不然就返回错误提示。通过这样的改造，各个业务层的微服务应用就不再需要非业务性质的校验逻辑了，这使得我们的微服务应用可以更专注于业务逻辑的开发，同时微服务的自动化测试也变得更容易实现。

微服务架构虽然可以将我们的开发单元拆分得更为细致，有效降低了开发难度，但是它所引出的各种问题如果处理不当会成为实施过程中的不稳定因素，甚至掩盖掉原本实施微服务带来的优势。所以，在微服务架构的实施方案中，API 网关服务的使用几乎成为了必然的选择。

下面我们将详细介绍 Spring Cloud Zuul 的使用方法、配置属性以及一些不足之处和需要进行的思考。

快速入门

介绍了这么多关于 API 网关服务的概念和作用，在这一节中，我们不妨用实际的示例来直观地体验一下 Spring Cloud Zuul 中封装的 API 网关是如何使用和运作，并应用到微服务架构中去的。

构建网关

首先，在实现各种 API 网关服务的高级功能之前，我们需要做一些准备工作，比如，构建起最基本的 API 网关服务，并且搭建几个用于路由和过滤使用的微服务应用等。对于微服务应用，我们可以直接使用之前章节实现的 hello-service 和 feign-consumer。虽然之前我们一直将 feign-consumer 视为消费者，但是在 Eureka 的服务注册与发现体系中，每个服务既是提供者也是消费者，所以 feign-consumer 实质上也是一个服务提供者。之前我们访问的 http://localhost:9001/feign-consumer 等一系列接口就是它提供的服务。读者也可以使用自己实现的微服务应用，因为这部分不是本章的重点，任何微服务应用都可以被用来进行后续的试验。这里，我们详细介绍一下 API 网关服务的构建过程。

- 创建一个基础的 Spring Boot 工程，命名为 api-gateway，并在 pom.xml 中引入 spring-cloud-starter-zuul 依赖，具体如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

对于 spring-cloud-starter-zuul 依赖，可以通过查看它的依赖内容了解到：该模块中不仅包含了 Netflix Zuul 的核心依赖 zuul-core，它还包含了下面这些网关服务需要的重要依赖。

- spring-cloud-starter-hystrix：该依赖用来在网关服务中实现对微服务转发时候的保护机制，通过线程隔离和断路器，防止微服务的故障引发 API 网关资源无法释放，从而影响其他应用的对外服务。
- spring-cloud-starter-ribbon：该依赖用来实现在网关服务进行路由转发时候的客户端负载均衡以及请求重试。
- spring-boot-starter-actuator：该依赖用来提供常规的微服务管理端点。另外，在 Spring Cloud Zuul 中还特别提供了/routes 端点来返回当前的所有路由规则。

- 创建应用主类，使用@EnableZuulProxy 注解开启 Zuul 的 API 网关服务功能。

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }
}
```

- 在 application.properties 中配置 Zuul 应用的基础信息，如应用名、服务端口号等，具体内容如下：

```
spring.application.name=api-gateway
server.port=5555
```

完成上面的工作后，通过 Zuul 实现的 API 网关服务就构建完毕了。

请求路由

下面，我们将通过一个简单的示例来为上面构建的网关服务增加请求路由的功能。为了演示请求路由的功能，我们先将之前准备的 Eureka 服务注册中心和微服务应用都启动起来。此时，我们在 Eureka 信息面板中可以看到如下图所示的两个微服务应用已经被注册成功了。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
FEIGN-CONSUMER	n/a (1)	(1)	UP (1) - PC-201602152056:feign-consumer:9001
HELLO-SERVICE	n/a (1)	(1)	UP (1) - PC-201602152056:hello-service

传统路由方式

使用 Spring Cloud Zuul 实现路由功能非常简单，只需要对 api-gateway 服务增加一

些关于路由规则的配置，就能实现传统的路由转发功能，比如：

```
zuul.routes.api-a-url.path=/api-a-url/**
zuul.routes.api-a-url.url=http://localhost:8080/
```

该配置定义了发往 API 网关服务的请求中，所有符合/api-a-url/**规则的访问都将被路由转发到 http://localhost:8080/地址上，也就是说，当我们访问 http://localhost:5555/api-a-url/hello 的时候，API 网关服务会将该请求路由到 http://localhost:8080/hello 提供的微服务接口上。其中，配置属性 zuul.routes.api-a-url.path 中的 api-a-url 部分为路由的名字，可以任意定义，但是一组 path 和 url 映射关系的路由名要相同，下面将要介绍的面向服务的映射方式也是如此。

面向服务的路由

很显然，传统路由的配置方式对于我们来说并不友好，它同样需要运维人员花费大量的时间来维护各个路由 path 与 url 的关系。为了解决这个问题，Spring Cloud Zuul 实现了与 Spring Cloud Eureka 的无缝整合，我们可以让路由的 path 不是映射具体的 url，而是让它映射到某个具体的服务，而具体的 url 则交给 Eureka 的服务发现机制去自动维护，我们称这类路由为面向服务的路由。在 Zuul 中使用服务路由也同样简单，只需做下面这些配置。

- 为了与 Eureka 整合，我们需要在 api-gateway 的 pom.xml 中引入 spring-cloud-starter-eureka 依赖，具体如下：

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

- 在 api-gateway 的 application.properties 配置文件中指定 Eureka 注册中心的位置，并且配置服务路由。具体如下：

```
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.serviceId=hello-service

zuul.routes.api-b.path=/api-b/**
zuul.routes.api-b.serviceId=feign-consumer

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

针对我们之前准备的两个微服务应用 hello-service 和 feign-consumer，在上面的配置中分别定义了两个名为 api-a 和 api-b 的路由来映射它们。另外，通过指定 Eureka Server 服务注册中心的位置，除了将自己注册成服务之外，同时也让 Zuul 能够获取

hello-service 和 feign-consumer 服务的实例清单，以实现 path 映射服务，再从服务中挑选实例来进行请求转发的完整路由机制。

在完成了上面的服务路由配置之后，我们可以将 eureka-server、hello-service、feign-consumer 以及这里用 Spring Cloud Zuul 构建的 api-gateway 都启动起来。启动完毕，在 eureka-server 的信息面板中，我们可以看到，除了 hello-service 和 feign-consumer 之外，多了一个网关服务 API-GATEWAY。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - PC-201602152056:api-gateway:5555
FEIGN-CONSUMER	n/a (1)	(1)	UP (1) - PC-201602152056:feign-consumer:9001
HELLO-SERVICE	n/a (1)	(1)	UP (1) - PC-201602152056:hello-service

通过上面的搭建工作，我们已经可以通过服务网关来访问 hello-service 和 feign-consumer 这两个服务了。根据配置的映射关系，分别向网关发起下面这些请求。

- http://localhost:5555/api-a/hello：该 url 符合/api-a/**规则，由 api-a 路由负责转发，该路由映射的 serviceId 为 hello-service，所以最终/hello 请求会被发送到 hello-service 服务的某个实例上去。
- http://localhost:5555/api-b/feign-consumer：该 url 符合/api-b/**规则，由 api-b 路由负责转发，该路由映射的 serviceId 为 feign-consumer，所以最终/feign-consumer 请求会被发送到 feign-consumer 服务的某个实例上去。

通过面向服务的路由配置方式，我们不需要再为各个路由维护微服务应用的具体实例的位置，而是通过简单的 path 与 serviceId 的映射组合，使得维护工作变得非常简单。这完全归功于 Spring Cloud Eureka 的服务发现机制，它使得 API 网关服务可以自动化完成服务实例清单的维护，完美地解决了对路由映射实例的维护问题。

请求过滤

在实现了请求路由功能之后，我们的微服务应用提供的接口就可以通过统一的 API 网关入口被客户端访问到了。但是，每个客户端用户请求微服务应用提供的接口时，它们的访问权限往往都有一定的限制，系统并不会将所有的微服务接口都对它们开放。然而，目前的服务路由并没有限制权限这样的功能，所有请求都会被毫无保留地转发到具体的应用并返回结果，为了实现对客户端请求的安全校验和权限控制，最简单和粗暴的方法就是为每个微服务应用都实现一套用于校验签名和鉴别权限的过滤器或拦截器。不过，这样的做法并不可取，它会增加日后系统的维护难度，因为同一个系统中的各种校验逻辑很多情况

下都是大致相同或类似的，这样的实现方式会使得相似的校验逻辑代码被分散到了各个微服务中去，冗余代码的出现是我们不希望看到的。所以，比较好的做法是将这些校验逻辑剥离出去，构建出一个独立的鉴权服务。在完成了剥离之后，有不少开发者会直接在微服务应用中通过调用鉴权服务来实现校验，但是这样的做法仅仅只是解决了鉴权逻辑的分离，并没有在本质上将这部分不属于冗余的逻辑从原有的微服务应用中拆分出，冗余的拦截器或过滤器依然会存在。

对于这样的问题，更好的做法是通过前置的网关服务来完成这些非业务性质的校验。由于网关服务的加入，外部客户端访问我们的系统已经有了统一入口，既然这些校验与具体业务无关，那何不在请求到达的时候就完成校验和过滤，而不是转发后再过滤而导致更长的请求延迟。同时，通过在网关中完成校验和过滤，微服务应用端就可以去除各种复杂的过滤器和拦截器了，这使得微服务应用接口的开发和测试复杂度也得到了相应降低。

为了在 API 网关中实现对客户端请求的校验，我们将继续介绍 Spring Cloud Zuul 的另外一个核心功能：请求过滤。Zuul 允许开发者在 API 网关上通过定义过滤器来实现对请求的拦截与过滤，实现的方法非常简单，我们只需要继承 `ZuulFilter` 抽象类并实现它定义的 4 个抽象函数就可以完成对请求的拦截和过滤了。

下面的代码定义了一个简单的 Zuul 过滤器，它实现了在请求被路由之前检查 `HttpServletRequest` 中是否有 `accessToken` 参数，若有就进行路由，若没有就拒绝访问，返回 401 Unauthorized 错误。

```
public class AccessFilter extends ZuulFilter {

    private static Logger log = LoggerFactory.getLogger(AccessFilter.class);

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 0;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
```

```
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();

        log.info("send {} request to {}", request.getMethod(),
request.getRequestURL().toString());

        Object accessToken = request.getParameter("accessToken");
        if(accessToken == null) {
            log.warn("access token is empty");
            ctx.setSendZuulResponse(false);
            ctx.setResponseStatusCode(401);
            return null;
        }
        log.info("access token ok");
        return null;
    }

}
```

在上面实现的过滤器代码中，我们通过继承 ZuulFilter 抽象类并重写下面 4 个方法来实现自定义的过滤器。这 4 个方法分别定义了如下内容。

- filterType：过滤器的类型，它决定过滤器在请求的哪个生命周期中执行。这里定义为 pre，代表会在请求被路由之前执行。
- filterOrder：过滤器的执行顺序。当请求在一个阶段中存在多个过滤器时，需要根据该方法返回的值来依次执行。
- shouldFilter：判断该过滤器是否需要被执行。这里我们直接返回了 true，因此该过滤器对所有请求都会生效。实际运用中我们可以利用该函数来指定过滤器的有效范围。
- run：过滤器的具体逻辑。这里我们通过 ctx.setSendZuulResponse(false) 令 zuul 过滤该请求，不对其进行路由，然后通过 ctx.setResponseStatus-Code(401) 设置了其返回的错误码，当然也可以进一步优化我们的返回，比如，通过 ctx.setResponseBody(body) 对返回的 body 内容进行编辑等。

在实现了自定义过滤器之后，它并不会直接生效，我们还需要为其创建具体的 Bean 才能启动该过滤器，比如，在应用主类中增加如下内容：

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {

    public static void main(String[] args) {
```

```
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

    @Bean
    public AccessFilter accessFilter() {
        return new AccessFilter();
    }
}
```

在对 api-gateway 服务完成了上面的改造之后，我们可以重新启动它，并发起下面的请求，对上面定义的过滤器做一个验证。

- http://localhost:5555/api-a/hello：返回 401 错误。
- http://localhost:5555/api-a/hello&accessToken=token：正确路由到 hello-service 的/hello 接口，并返回 Hello World。

到这里，对于 API 网关服务的快速入门示例就完成了。通过对 Spring Cloud Zuul 两个核心功能的介绍，相信读者已经能够体会到 API 网关服务对微服务架构的重要性了，就目前掌握的 API 网关知识，我们可以将具体原因总结如下：

- 它作为系统的统一入口，屏蔽了系统内部各个微服务的细节。
- 它可以与服务治理框架结合，实现自动化的服务实例维护以及负载均衡的路由转发。
- 它可以实现接口权限校验与微服务业务逻辑的解耦。
- 通过服务网关中的过滤器，在各生命周期中去校验请求的内容，将原本在对外服务层做的校验前移，保证了微服务的无状态性，同时降低了微服务的测试难度，让服务本身更集中关注业务逻辑的处理。

实际上，基于 Spring Cloud Zuul 实现的 API 网关服务除了上面所示的优点之外，它还有一些更加强大的功能，我们将在后续的章节对其进行更深入的介绍。通过本节的内容，我们只是希望以一个简单的例子带领读者先来简单认识一下 API 网关服务提供的基础功能以及它在微服务架构中的重要地位。

路由详解

在“快速入门”一节的请求路由示例中，我们对 Spring Cloud Zuul 中的两类路由功能已经做了简单的使用介绍。在本节中，我们将进一步详细介绍关于 Spring Cloud Zuul 的路由功能，以帮助读者更好地理解和使用它。

传统路由配置

所谓的传统路由配置方式就是在不依赖于服务发现机制的情况下，通过在配置文件中

具体指定每个路由表达式与服务实例的映射关系来实现 API 网关对外部请求的路由。

没有 Eureka 等服务治理框架的帮助，我们需要根据服务实例的数量采用不同方式的配置来实现路由规则。

- **单实例配置：**通过 zuul.routes.<route>.path 与 zuul.routes.<route>.url 参数对的方式进行配置，比如：

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.url=http://localhost:8080/
```

  该配置实现了对符合 /user-service/** 规则的请求路径转发到 http://localhost:8080/地址的路由规则。比如，当有一个请求 http://localhost:5555/user-service/hello 被发送到 API 网关上，由于/user-service/hello 能够被上述配置的 path 规则匹配，所以 API 网关会转发请求到 http://localhost:8080/hello 地址。

- **多实例配置：**通过 zuul.routes.<route>.path 与 zuul.routes.<route>.serviceId 参数对的方式进行配置，比如：

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
ribbon.eureka.enabled=false
user-service.ribbon.listOfServers=http://localhost:8080/,http://localhost:8081/
```

  该配置实现了对符合 /user-service/** 规则的请求路径转发到 http://localhost:8080/和 http://localhost:8081/两个实例地址的路由规则。它的配置方式与服务路由的配置方式一样，都采用了 zuul.routes.<route>.path 与 zuul.routes.<route>.serviceId 参数对的映射方式，只是这里的 serviceId 是由用户手工命名的服务名称，配合 ribbon.listOfServers 参数实现服务与实例的维护。由于存在多个实例，API 网关在进行路由转发时需要实现负载均衡策略，于是这里还需要 Spring Cloud Ribbon 的配合。由于在 Spring Cloud Zuul 中自带了对 Ribbon 的依赖，所以我们只需做一些配置即可，比如上面示例中关于 Ribbon 的各个配置，它们的具体作用如下所示。

  - ribbon.eureka.enabled：由于 zuul.routes.<route>.serviceId 指定的是服务名称，默认情况下 Ribbon 会根据服务发现机制来获取配置服务名对应的实例清单。但是，该示例并没有整合类似 Eureka 之类的服务治理框架，所以需要将该参数设置为 false，否则配置的 serviceId 获取不到对应实例的清单。

- user-service.ribbon.listOfServers：该参数内容与 zuul.routes.<route>.serviceId 的配置相对应，开头的 user-service 对应了 serviceId 的值，这两个参数的配置相当于在该应用内部手工维护了服务与实例的对应关系。

不论是单实例还是多实例的配置方式，我们都需要为每一对映射关系指定一个名称，也就是上面配置中的<route>，每一个<route>对应了一条路由规则。每条路由规则都需要通过 path 属性来定义一个用来匹配客户端请求的路径表达式，并通过 url 或 serviceId 属性来指定请求表达式映射具体实例地址或服务名。

服务路由配置

对于服务路由，我们在快速入门示例中已经有过基础的介绍和体验，Spring Cloud Zuul 通过与 Spring Cloud Eureka 的整合，实现了对服务实例的自动化维护，所以在使用服务路由配置的时候，我们不需要向传统路由配置方式那样为 serviceId 指定具体的服务实例地址，只需要通过 zuul.routes.<route>.path 与 zuul.routes.<route>.serviceId 参数对的方式进行配置即可。

比如下面的示例，它实现了对符合/user-service/**规则的请求路径转发到名为 user-service 的服务实例上去的路由规则。其中<route>可以指定为任意的路由名称。

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
```

对于面向服务的路由配置，除了使用 path 与 serviceId 映射的配置方式之外，还有一种更简洁的配置方式：zuul.routes.<serviceId>=<path>，其中<serviceId>用来指定路由的具体服务名，<path>用来配置匹配的请求表达式。比如下面的例子，它的路由规则等价于上面通过 path 与 serviceId 组合使用的配置方式。

```
zuul.routes.user-service=/user-service/**
```

传统路由的映射方式比较直观且容易理解，API 网关直接根据请求的 URL 路径找到最匹配的 path 表达式，直接转发给该表达式对应的 url 或对应 serviceId 下配置的实例地址，以实现外部请求的路由。那么当采用 path 与 serviceId 以服务路由的方式实现时，在没有配置任何实例地址的情况下，外部请求经过 API 网关的时候，它是如何被解析并转发到服务具体实例的呢？

在本章一开始，我们就提到了 Zuul 巧妙地整合了 Eureka 来实现面向服务的路由。实际上，我们可以直接将 API 网关也看作 Eureka 服务治理下的一个普通微服务应用。它除了会将自己注册到 Eureka 服务注册中心上之外，也会从注册中心获取所有服务以及它们的实例清单。所以，在 Eureka 的帮助下，API 网关服务本身就已经维护了系统中所有 serviceId

与实例地址的映射关系。当有外部请求到达 API 网关的时候，根据请求的 URL 路径找到最佳匹配的 path 规则，API 网关就可以知道要将该请求路由到哪个具体的 serviceId 上去。由于在 API 网关中已经知道 serviceId 对应服务实例的地址清单，那么只需要通过 Ribbon 的负载均衡策略，直接在这些清单中选择一个具体的实例进行转发就能完成路由工作了。

服务路由的默认规则

虽然通过 Eureka 与 Zuul 的整合已经为我们省去了维护服务实例清单的大量配置工作，剩下只需要再维护请求路径的匹配表达式与服务名的映射关系即可。但是在实际的运用过程中会发现，大部分的路由配置规则几乎都会采用服务名作为外部请求的前缀，比如下面的例子，其中 path 路径的前缀使用了 user-service，而对应的服务名称也是 user-service。

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
```

对于这样具有规则性的配置内容，我们总是希望可以自动化地完成。非常庆幸，Zuul 默认实现了这样的贴心功能，当我们为 Spring Cloud Zuul 构建的 API 网关服务引入 Spring Cloud Eureka 之后，它为 Eureka 中的每个服务都自动创建一个默认路由规则，这些默认规则的 path 会使用 serviceId 配置的服务名作为请求前缀，就如上面的例子那样。

由于默认情况下所有 Eureka 上的服务都会被 Zuul 自动地创建映射关系来进行路由，这会使得一些我们不希望对外开放的服务也可能被外部访问到。这个时候，我们可以使用 zuul.ignored-services 参数来设置一个服务名匹配表达式来定义不自动创建路由的规则。Zuul 在自动创建服务路由的时候会根据该表达式来进行判断，如果服务名匹配表达式，那么 Zuul 将跳过该服务，不为其创建路由规则。比如，设置为 zuul.ignored-services=*的时候，Zuul 将对所有的服务都不自动创建路由规则。在这种情况下，我们就要在配置文件中逐个为需要路由的服务添加映射规则（可以使用 path 与 serviceId 组合的配置方式，也可使用更简洁的 zuul.routes.<serviceId>=<path>配置方式），只有在配置文件中出现的映射规则会被创建路由，而从 Eureka 中获取的其他服务，Zuul 将不会再为它们创建路由规则。

自定义路由映射规则

我们在构建微服务系统进行业务逻辑开发的时候，为了兼容外部不同版本的客户端程序（尽量不强迫用户升级客户端），一般都会采用开闭原则来进行设计与开发。这使得系统在迭代过程中，有时候会需要我们为一组互相配合的微服务定义一个版本标识来方便管理

它们的版本关系，根据这个标识我们可以很容易地知道这些服务需要一起启动并配合使用。比如可以采用类似这样的命名：userservice-v1、userservice-v2、orderservice-v1、orderservice-v2。默认情况下，Zuul 自动为服务创建的路由表达式会采用服务名作为前缀，比如针对上面的 userservice-v1 和 userservice-v2，它会产生/userservice-v1 和/userservice-v2 两个路径表达式来映射，但是这样生成出来的表达式规则较为单一，不利于通过路径规则来进行管理。通常的做法是为这些不同版本的微服务应用生成以版本代号作为路由前缀定义的路由规则，比如/v1/userservice/。这时候，通过这样具有版本号前缀的 URL 路径，我们就可以很容易地通过路径表达式来归类和管理这些具有版本信息的微服务了。

针对上面所述的需求，如果我们的各个微服务应用都遵循了类似 userservice-v1 这样的命名规则，通过-分隔的规范来定义服务名和服务版本标识的话，那么，我们可以使用 Zuul 中自定义服务与路由映射关系的功能，来实现为符合上述规则的微服务自动化地创建类似/v1/userservice/**的路由匹配规则。实现步骤非常简单，只需在 API 网关程序中，增加如下 Bean 的创建即可：

```
@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
    return new PatternServiceRouteMapper(
            "(?<name>^.+)-(?<version>v.+$)",
            "${version}/${name}");
}
```

PatternServiceRouteMapper 对象可以让开发者通过正则表达式来自定义服务与路由映射的生成关系。其中构造函数的第一个参数是用来匹配服务名称是否符合该自定义规则的正则表达式，而第二个参数则是定义根据服务名中定义的内容转换出的路径表达式规则。当开发者在 API 网关中定义了 PatternServiceRouteMapper 实现之后，只要符合第一个参数定义规则的服务名，都会优先使用该实现构建出的路径表达式，如果没有匹配上的服务则还是会使用默认的路由映射规则，即采用完整服务名作为前缀的路径表达式。

路径匹配

不论是使用传统路由的配置方式还是服务路由的配置方式，我们都需要为每个路由规则定义匹配表达式，也就是上面所说的 path 参数。在 Zuul 中，路由匹配的路径表达式采用了 Ant 风格定义。

Ant 风格的路径表达式使用起来非常简单，它一共有下面这三种通配符。

通配符	说明
?	匹配任意单个字符
*	匹配任意数量的字符
**	匹配任意数量的字符，支持多级目录

我们可以通过下表中的示例来进一步理解这三个通配符的含义并进行参考使用。

URL 路径	说明
/user-service/?	它可以匹配/user-service/之后拼接一个任务字符的路径，比如/user-service/a、/user-service/b、/user-service/c
/user-service/*	它可以匹配/user-service/之后拼接任意字符的路径，比如/user-service/a、/user-service/aaa、/user-service/bbb。但是它无法匹配/user-service/a/b
/user-service/**	它可以匹配/user-service/*包含的内容之外，还可以匹配形如/user-service/a/b的多级目录路径

另外，当我们使用通配符的时候，经常会碰到这样的问题：一个 URL 路径可能会被多个不同路由的表达式匹配上。比如，有这样一个场景，我们在系统建设的一开始实现了 user-service 服务，并且配置了如下路由规则：

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
```

但是随着版本的迭代，我们对 user-service 服务做了一些功能拆分，将原属于 user-service 服务的某些功能拆分到了另外一个全新的服务 user-service-ext 中去，而这些拆分的外部调用 URL 路径希望能够符合规则/user-service/ext/**，这个时候我们需要就在配置文件中增加一个路由规则，完整配置如下：

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service

zuul.routes.user-service-ext.path=/user-service/ext/**
zuul.routes.user-service-ext.serviceId=user-service-ext
```

此时，调用 user-service-ext 服务的 URL 路径实际上会同时被/user-service/**和/user-service/ext/**两个表达式所匹配。在逻辑上，API 网关服务需要优先选择/user-service/ext/**路由，然后再匹配/user-service/**路由才能实现上述需求。但是如果使用上面的配置方式，实际上是无法保证这样的路由优先顺序的。

从下面的路由匹配算法中，我们可以看到它在使用路由规则匹配请求路径的时候是通过线性遍历的方式，在请求路径获取到第一个匹配的路由规则之后就返回并结束匹配过程。所以当存在多个匹配的路由规则时，匹配结果完全取决于路由规则的保存顺序。

```
@Override
public Route getMatchingRoute(final String path) {
    ...
    ZuulRoute route = null;
    if (!matchesIgnoredPatterns(adjustedPath)) {
        for (Entry<String, ZuulRoute> entry : this.routes.get().entrySet()) {
            String pattern = entry.getKey();
            log.debug("Matching pattern:" + pattern);
            if (this.pathMatcher.match(pattern, adjustedPath)) {
                route = entry.getValue();
                break;
            }
        }
    }
    log.debug("route matched=" + route);
    return getRoute(route, adjustedPath);
}
```

下面所示的代码是基础的路由规则加载算法，我们可以看到这些路由规则是通过 LinkedHashMap 保存的，也就是说，路由规则的保存是有序的，而内容的加载是通过遍历配置文件中路由规则依次加入的，所以导致问题的根本原因是对配置文件中内容的读取。

```
protected Map<String, ZuulRoute> locateRoutes() {
    LinkedHashMap<String, ZuulRoute> routesMap = new LinkedHashMap<String,
ZuulRoute>();
    for (ZuulRoute route : this.properties.getRoutes().values()) {
        routesMap.put(route.getPath(), route);
    }
    return routesMap;
}
```

由于 properties 的配置内容无法保证有序，所以当出现这种情况的时候，为了保证路由的优先顺序，我们需要使用 YAML 文件来配置，以实现有序的路由规则，比如使用下面的定义：

```
zuul:
  routes:
    user-service-ext:
      path: /user-service/ext/**
      serviceId: user-service-ext
    user-service:
      path: /user-service/**
      serviceId: user-service
```

忽略表达式

通过 path 参数定义的 Ant 表达式已经能够完成 API 网关上的路由规则配置功能，但是为了更细粒度和更为灵活地配置路由规则，Zuul 还提供了一个忽略表达式参数 zuul.ignored-patterns。该参数可以用来设置不希望被 API 网关进行路由的 URL 表达式。

比如，以快速入门中的示例为基础，如果不希望/hello 接口被路由，那么我们可以这样设置：

```
zuul.ignored-patterns=/**/hello/**
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.serviceId=hello-service
```

然后，可以尝试通过网关来访问 hello-service 的/hello 接口 http://localhost:5555/api-a/hello。虽然该访问路径完全符合 path 参数定义的/api-a/**规则，但是由于该路径符合 zuul.ignored-patterns 参数定义的规则，所以不会被正确路由。同时，我们在控制台或日志中还能看到没有匹配路由的输出信息：

```
o.s.c.n.z.f.pre.PreDecorationFilter      : No route found for uri: /api-a/hello
```

另外，该参数在使用时还需要注意它的范围并不是对某个路由，而是对所有路由。所以在设置的时候需要全面考虑 URL 规则，防止忽略了不该被忽略的 URL 路径。

路由前缀

为了方便全局地为路由规则增加前缀信息，Zuul 提供了 zuul.prefix 参数来进行设置。比如，希望为网关上的路由规则都增加/api 前缀，那么我们可以在配置文件中增加配置：zuul.prefix=/api。另外，对于代理前缀会默认从路径中移除，我们可以通过设置 zuul.stripPrefix=false 来关闭该移除代理前缀的动作，也可以通过 zuul.routes.<route>.strip-prefix=true 来对指定路由关闭移除代理前缀的动作。

注意，在使用 zuul.prefix 参数的时候，目前版本的实现还存在一些 Bug，所以请谨慎使用，或是避开会引发 Bug 的配置规则。具体会引发 Bug 的规则如下：

假设我们设置 zuul.prefix=/api，当路由规则的 path 表达式以/api 开头的时候，将会产生错误的映射关系。可以进行下面的配置实验来验证这个问题：

```
zuul.routes.api-a.path=/api/a/**
zuul.routes.api-a.serviceId=hello-service

zuul.routes.api-b.path=/api-b/**
zuul.routes.api-b.serviceId=hello-service
```

```
zuul.routes.api-c.path=/ccc/**
zuul.routes.api-c.serviceId=hello-service
```

这里配置了三个路由关系：/api/a/**、/api-b/**、/ccc/**，这三个路径规则都将被路由到hello-service服务上去。当我们没有设置zuul.prefix=/api的时候，一切运作正常。但是在增加了zuul.prefix=/api配置之后，会得到下面这样的路由关系：

```
o.s.c.n.zuul.web.ZuulHandlerMapping     : Mapped URL path [/api/api/a/a/**] onto
handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]
    o.s.c.n.zuul.web.ZuulHandlerMapping     : Mapped URL path [/api/api-b-b/**] onto
handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]
    o.s.c.n.zuul.web.ZuulHandlerMapping     : Mapped URL path [/api/ccc/**] onto
handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]
```

从日志信息中我们可以看到，以/api开头的路由规则解析除了两个看似就有问题的映射 URL，我们可以通过该输出的 URL 来访问，实际是路由不到正确的服务接口的，只有非/api开头的路由规则/ccc/**能够正常路由。

上述实验基于 Brixton.SR7 和 Camden.SR3 测试均存在问题，所以在使用该版本或以下版本时候，务必避免让路由表达式的起始字符串与zuul.prefix参数相同。

本地跳转

在 Zuul 实现的 API 网关路由功能中，还支持forward形式的服务端跳转配置。实现方式非常简单，只需通过使用path与url的配置方式就能完成，通过url中使用forward来指定需要跳转的服务器资源路径。

下面的配置实现了两个路由规则，api-a路由实现了将符合/api-a/**规则的请求转发到http://localhost:8001/；而api-b路由则使用了本地跳转，它实现了将符合/api-b/**规则的请求转发到 API 网关中以/local为前缀的请求上，由 API 网关进行本地处理。比如，当 API 网关接收到请求/api-b/hello，它符合api-b的路由规则，所以该请求会被 API 网关转发到网关的/local/hello请求上进行本地处理。

```
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.url=http://localhost:8001/

zuul.routes.api-b.path=/api-b/**
zuul.routes.api-b.url=forward:/local
```

这里要注意，由于需要在 API 网关上实现本地跳转，所以相应的我们也需要为本地跳转实现对应的请求接口。按照上面的例子，在 API 网关上还需要增加一个/local/hello的接口实现才能让api-b路由规则生效，比如下面的实现。否则 Zuul 在进行 forward 转发的时候会因为找不到该请求而返回 404 错误。

```
@RestController
public class HelloController {

    @RequestMapping("/local/hello")
    public String hello() {
        return "Hello World Local";
    }

}
```

Cookie 与头信息

默认情况下，Spring Cloud Zuul 在请求路由时，会过滤掉 HTTP 请求头信息中的一些敏感信息，防止它们被传递到下游的外部服务器。默认的敏感头信息通过 `zuul.sensitiveHeaders` 参数定义，包括 `Cookie`、`Set-Cookie`、`Authorization` 三个属性。所以，我们在开发 Web 项目时常用的 `Cookie` 在 Spring Cloud Zuul 网关中默认是不会传递的，这就会引发一个常见的问题：如果我们要将使用了 Spring Security、Shiro 等安全框架构建的 Web 应用通过 Spring Cloud Zuul 构建的网关来进行路由时，由于 `Cookie` 信息无法传递，我们的 Web 应用将无法实现登录和鉴权。为了解决这个问题，配置的方法有很多。

- 通过设置全局参数为空来覆盖默认值，具体如下：

```
zuul.sensitiveHeaders=
```

  这种方法并不推荐，虽然可以实现 Cookie 的传递，但是破坏了默认设置的用意。在微服务架构的 API 网关之内，对于无状态的 RESTful API 请求肯定是要远多于这些 Web 类应用请求的，甚至还有一些架构设计会将 Web 类应用和 App 客户端一样都归为 API 网关之外的客户端应用。

- 通过指定路由的参数来配置，方法有下面两种。

```
# 方法一：对指定路由开启自定义敏感头
zuul.routes.<router>.customSensitiveHeaders=true
# 方法二：将指定路由的敏感头设置为空
zuul.routes.<router>.sensitiveHeaders=
```

  比较推荐使用这两种方法，仅对指定的 Web 应用开启对敏感信息的传递，影响范围小，不至于引起其他服务的信息泄露问题。

重定向问题

在解决了 Cookie 问题之后，我们已经能够通过网关来访问并登录到我们的 Web 应用

了。但是这个时候又会发现另外一个问题：虽然可以通过网关访问登录页面并发起登录请求，但是登录成功之后，我们跳转到的页面 URL 却是具体 Web 应用实例的地址，而不是通过网关的路由地址。这个问题非常严重，因为使用 API 网关的一个重要原因就是要将网关作为统一入口，从而不暴露所有的内部服务细节。那么是什么原因导致了这个问题呢？

通过浏览器开发工具查看登录以及登录之后的请求详情，可以发现，引起问题的大致原因是由于 Spring Security 或 Shiro 在登录完成之后，通过重定向的方式跳转到登录后的页面，此时登录后的请求结果状态码为 302，请求响应头信息中的 Location 指向了具体的服务实例地址，而请求头信息中的 Host 也指向了具体的服务实例 IP 地址和端口。所以，该问题的根本原因在于 Spring Cloud Zuul 在路由请求时，并没有将最初的 Host 信息设置正确。那么如何解决这个问题呢？

针对这个问题，目前在 `spring-cloud-netflix-core-1.2.x` 版本的 Zuul 中增加了一个参数配置，能够使得网关在进行路由转发前为请求设置 Host 头信息，以标识最初的服务端请求地址。具体配置方式如下：

```
zuul.addHostHeader=true
```

由于 Spring Cloud 的版本依赖原因，目前的 Brixton 版本采用了 `spring-cloud-netflix-core-1.1.x`，只有 Camden 版本采用了 `spring-cloud-netflix-core-1.2.x`。所以重定向的问题如果要在 Zuul 中解决，最简单的方法就使用 Camden 版本。如果要在 Brixton 版本中解决，可以参考 Camden 的 `PreDecorationFilter` 的实现扩展过滤器链来增加 Host 信息。

Hystrix 和 Ribbon 支持

在“快速入门”一节中介绍 `spring-cloud-starter-zuul` 依赖时，我们提到了它自身就包含了对 `spring-cloud-starter-hystrix` 和 `spring-cloud-starter-ribbon` 模块的依赖，所以 Zuul 天生就拥有线程隔离和断路器的自我保护功能，以及对服务调用的客户端负载均衡功能。但是需要注意，当使用 `path` 与 `url` 的映射关系来配置路由规则的时候，对于路由转发的请求不会采用 `HystrixCommand` 来包装，所以这类路由请求没有线程隔离和断路器的保护，并且也不会有负载均衡的能力。因此，我们在使用 Zuul 的时候尽量使用 `path` 和 `serviceId` 的组合来进行配置，这样不仅可以保证 API 网关的健壮和稳定，也能用到 Ribbon 的客户端负载均衡功能。

我们在使用 Zuul 搭建 API 网关的时候，可以通过 Hystrix 和 Ribbon 的参数来调整路由请求的各种超时时间等配置，比如下面这些参数的设置。

- hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds：该参数可以用来设置 API 网关中路由转发请求的 HystrixCommand 执行超时时间，单位为毫秒。当路由转发请求的命令执行时间超过该配置值之后，Hystrix 会将该执行命令标记为 TIMEOUT 并抛出异常，Zuul 会对该异常进行处理并返回如下 JSON 信息给外部调用方。

```
{
  "timestamp": 1481350975323,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "com.netflix.zuul.exception.ZuulException",
  "message": "TIMEOUT"
}
```

- ribbon.ConnectTimeout：该参数用来设置路由转发请求的时候，创建请求连接的超时时间。当 ribbon.ConnectTimeout 的配置值小于 hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds 配置值的时候，若出现路由请求出现连接超时，会自动进行重试路由请求，如果重试依然失败，Zuul 会返回如下 JSON 信息给外部调用方。

```
{
  "timestamp": 1481352582852,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "com.netflix.zuul.exception.ZuulException",
  "message": "NUMBEROF_RETRIES_NEXTSERVER_EXCEEDED"
}
```

  如果 ribbon.ConnectTimeout 的配置值大于 hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds 配置值的时候，当出现路由请求连接超时时，由于此时对于路由转发的请求命令已经超时，所以不会进行重试路由请求，而是直接按请求命令超时处理，返回 TIMEOUT 的错误信息。
- ribbon.ReadTimeout：该参数用来设置路由转发请求的超时时间。它的处理与 ribbon.ConnectTimeout 类似，只是它的超时是对请求连接建立之后的处理时间。当 ribbon.ReadTimeout 的配置值小于 hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds 配置值的时候，若路由请求的处理时间超过该配置值且依赖服务的请求还未响应的时候，会自动进

行重试路由请求。如果重试后依然没有获得请求响应，Zuul 会返回 NUMBEROF_RETRIES_NEXTSERVER_EXCEEDED 错误。如果 ribbon.ReadTimeout 的配置值大于 hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds 配置值，若路由请求的处理时间超过该配置值且依赖服务的请求还未响应时，不会进行重试路由请求，而是直接按请求命令超时处理，返回 TIMEOUT 的错误信息。

根据上面的介绍我们可以知道，在使用 Zuul 的服务路由时，如果路由转发请求发生超时（连接超时或处理超时），只要超时时间的设置小于 Hystrix 的命令超时时间，那么它就会自动发起重试。但是在有些情况下，我们可能需要关闭该重试机制，那么可以通过下面的两个参数来进行设置：

```
zuul.retryable=false
zuul.routes.<route>.retryable=false
```

其中，zuul.retryable 用来全局关闭重试机制，而 zuul.routes.<route>.retryable=false 则是指定路由关闭重试机制。

过滤器详解

在本章一开始的快速入门示例中，我们已经介绍了一部分关于请求过滤的功能。在本节中，我们将对 Zuul 的请求过滤器功能做进一步的介绍和总结。

过滤器

通过快速入门的示例，我们对于 Zuul 的第一印象通常是这样的：它包含了对请求的路由和过滤两个功能，其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础；而过滤器功能则负责对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的基础。然而实际上，路由功能在真正运行时，它的路由映射和请求转发都是由几个不同的过滤器完成的。其中，路由映射主要通过 pre 类型的过滤器完成，它将请求路径与配置的路由规则进行匹配，以找到需要转发的目标地址；而请求转发的部分则是由 route 类型的过滤器来完成，对 pre 类型过滤器获得的路由地址进行转发。所以，过滤器可以说是 Zuul 实现 API 网关功能最为核心的部件，每一个进入 Zuul 的 HTTP 请求都会经过一系列的过滤器处理链得到请求响应并返回给客户端。

在 Spring Cloud Zuul 中实现的过滤器必须包含 4 个基本特征：过滤类型、执行顺序、执行条件、具体操作。这些元素看起来非常熟悉，实际上它就是 ZuulFilter 接口中定义

的 4 个抽象方法：

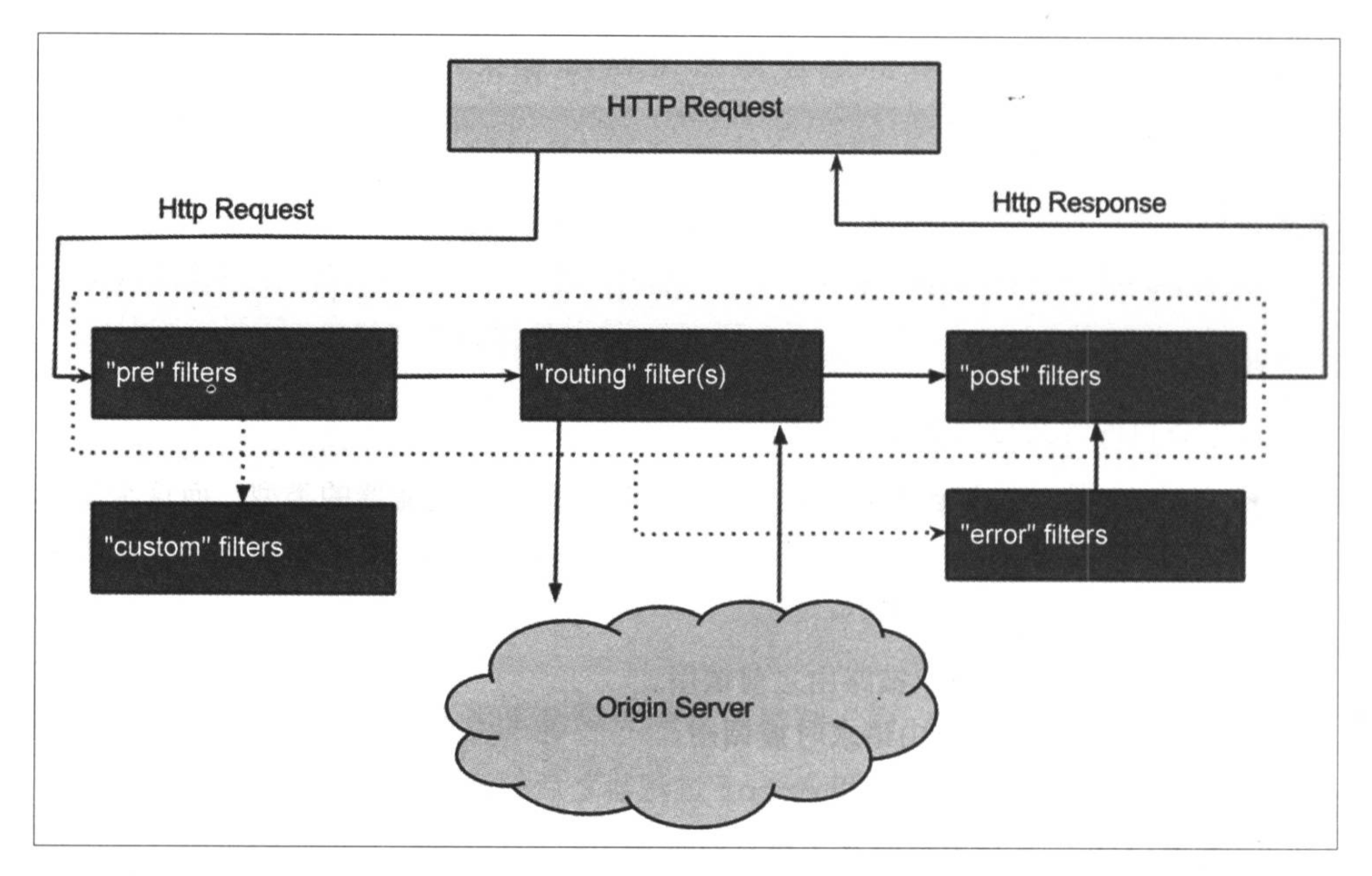
```
String filterType();

int filterOrder();

boolean shouldFilter();

Object run();
```

它们各自的含义与功能总结如下。

- filterType：该函数需要返回一个字符串来代表过滤器的类型，而这个类型就是在 HTTP 请求过程中定义的各个阶段。在 Zuul 中默认定义了 4 种不同生命周期的过滤器类型，具体如下所示。
  - pre：可以在请求被路由之前调用。
  - routing：在路由请求时被调用。
  - post：在 routing 和 error 过滤器之后被调用。
  - error：处理请求时发生错误时被调用。
- filterOrder：通过 int 值来定义过滤器的执行顺序，数值越小优先级越高。
- shouldFilter：返回一个 boolean 值来判断该过滤器是否要执行。我们可以通过此方法来指定过滤器的有效范围。
- run：过滤器的具体逻辑。在该函数中，我们可以实现自定义的过滤逻辑，来确定是否要拦截当前的请求，不对其进行后续的路由，或是在请求路由返回结果之后，对处理结果做一些加工等。

请求生命周期

对于 Zuul 中的过滤器类型 filterType，我们已经做过一些简单的介绍。Zuul 默认定义了 4 种不同的过滤器类型，它们覆盖了一个外部 HTTP 请求到达 API 网关，直到返回请求结果的全部生命周期。下图源自 Zuul 的官方 WiKi 中关于请求生命周期的图解，它描述了一个 HTTP 请求到达 API 网关之后，如何在各种不同类型的过滤器之间流转的详细过程。



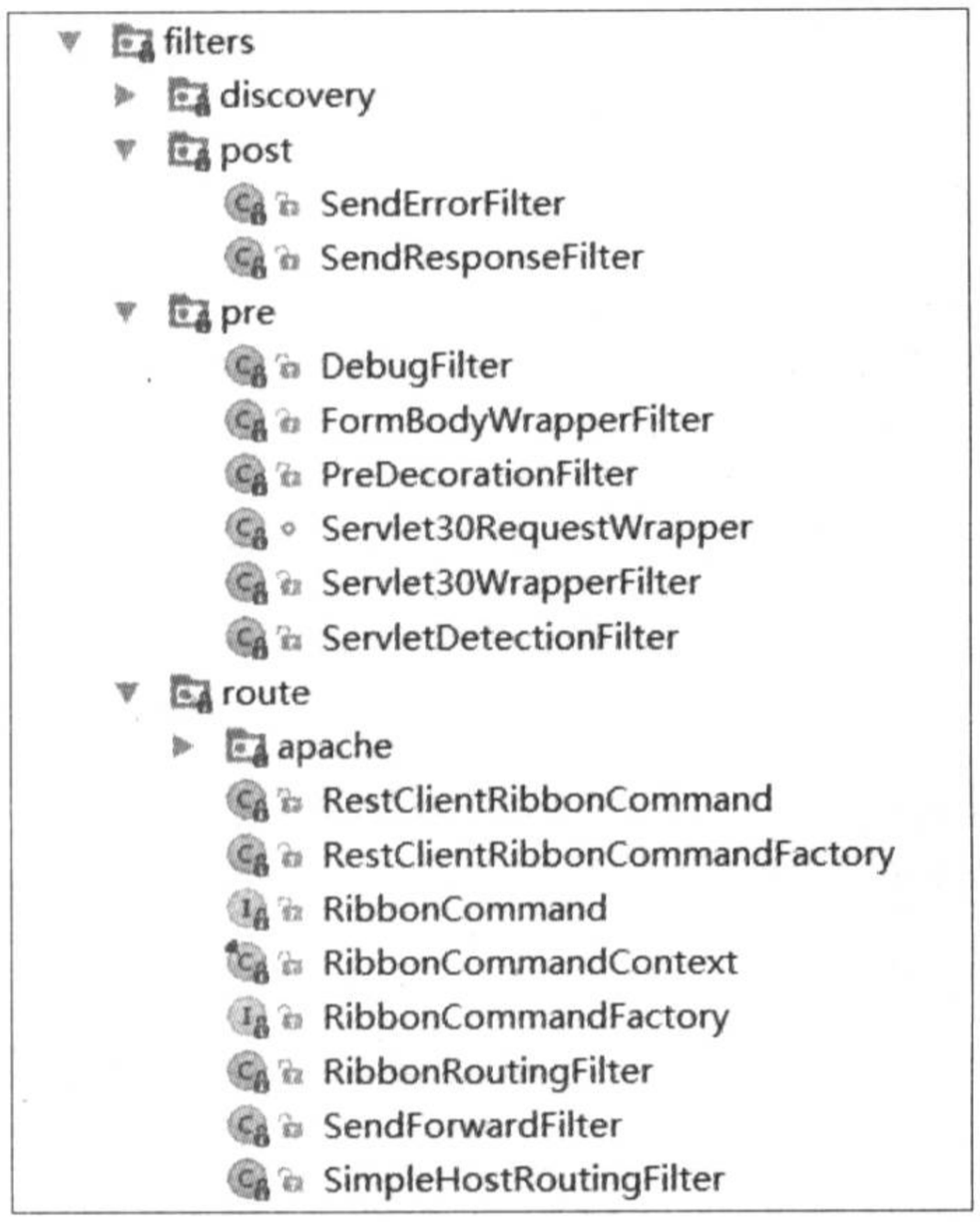
从上图中我们可以看到，当外部 HTTP 请求到达 API 网关服务的时候，首先它会进入第一个阶段 pre，在这里它会被 pre 类型的过滤器进行处理，该类型过滤器的主要目的是在进行请求路由之前做一些前置加工，比如请求的校验等。在完成了 pre 类型的过滤器处理之后，请求进入第二个阶段 routing，也就是之前说的路由请求转发阶段，请求将会被 routing 类型过滤器处理。这里的具体处理内容就是将外部请求转发到具体服务实例上去的过程，当服务实例将请求结果都返回之后，routing 阶段完成，请求进入第三个阶段 post。此时请求将会被 post 类型的过滤器处理，这些过滤器在处理的时候不仅可以获取到请求信息，还能获取到服务实例的返回信息，所以在 post 类型的过滤器中，我们可以对处理结果进行一些加工或转换等内容。另外，还有一个特殊的阶段 error，该阶段只有在上述三个阶段中发生异常的时候才会触发，但是它的最后流向还是 post 类型的过滤器，因为它需要通过 post 过滤器将最终结果返回给请求客户端（对于 error 过滤器的处理，在 Spring Cloud Zuul 的过滤链中实际上有一些不同，后续我们在介绍核心过滤器时会做详细分析）。

核心过滤器

在 Spring Cloud Zuul 中，为了让 API 网关组件可以被更方便地使用，它在 HTTP 请求生命周期的各个阶段默认实现了一批核心过滤器，它们会在 API 网关服务启动的时候被自动加载和启用。我们可以在源码中查看和了解它们，它们定义于 spring-cloud-

netflix-core 模块的 org.springframework.cloud.netflix.zuul.filters 包下。

如下图所示，在默认启用的过滤器中包含三种不同生命周期的过滤器，这些过滤器都非常重要，可以帮助我们理解 Zuul 对外部请求处理的过程，以及帮助我们在此基础上扩展过滤器去完成自身系统需要的功能。下面，我们将逐个对这些过滤器做详细的介绍。



pre 过滤器

- ServletDetectionFilter：它的执行顺序为-3，是最先被执行的过滤器。该过滤器总是会被执行，主要用来检测当前请求是通过 Spring 的 DispatcherServlet 处理运行的，还是通过 ZuulServlet 来处理运行的。它的检测结果会以布尔类型保存在当前请求上下文的 isDispatcherServletRequest 参数中，这样在后续的过滤器中，我们就可以通过 RequestUtils.isDispatcherServletRequest() 和 RequestUtils.isZuulServletRequest() 方法来判断请求处理的源头，以实现后续不同的处理机制。一般情况下，发送到 API 网关的外部请求都会被 Spring 的 DispatcherServlet 处理，除了通过/zuul/*路径访问的请求会绕过 DispatcherServlet，被 ZuulServlet 处理，主要用来应对处理大文件上传的情况。另外，对于 ZuulServlet 的访问路径/zuul/*，我们可以通过 zuul.servletPath 参数来进行修改。

- Servlet30WrapperFilter：它的执行顺序为-2，是第二个执行的过滤器。目前的实现会对所有请求生效，主要为了将原始的 HttpServletRequest 包装成 Servlet30RequestWrapper 对象。
- FormBodyWrapperFilter：它的执行顺序为-1，是第三个执行的过滤器。该过滤器仅对两类请求生效，第一类是 Content-Type 为 application/x-www-form-urlencoded 的请求，第二类是 Content-Type 为 multipart/form-data 并且是由 Spring 的 DispatcherServlet 处理的请求（用到了 ServletDetectionFilter 的处理结果）。而该过滤器的主要目的是将符合要求的请求体包装成 FormBodyRequestWrapper 对象。
- DebugFilter：它的执行顺序为 1，是第四个执行的过滤器。该过滤器会根据配置参数 zuul.debug.request 和请求中的 debug 参数来决定是否执行过滤器中的操作。而它的具体操作内容则是将当前请求上下文中的 debugRouting 和 debugRequest 参数设置为 true。由于在同一个请求的不同生命周期中都可以访问到这两个值，所以我们在后续的各个过滤器中可以利用这两个值来定义一些 debug 信息，这样当线上环境出现问题的时候，可以通过请求参数的方式来激活这些 debug 信息以帮助分析问题。另外，对于请求参数中的 debug 参数，我们也可以通过 zuul.debug.parameter 来进行自定义。
- PreDecorationFilter：它的执行顺序为 5，是 pre 阶段最后被执行的过滤器。该过滤器会判断当前请求上下文中是否存在 forward.to 和 serviceId 参数，如果都不存在，那么它就会执行具体过滤器的操作（如果有一个存在的话，说明当前请求已经被处理过了，因为这两个信息就是根据当前请求的路由信息加载进来的）。而它的具体操作内容就是为当前请求做一些预处理，比如，进行路由规则的匹配、在请求上下文中设置该请求的基本信息以及将路由匹配结果等一些设置信息等，这些信息将是后续过滤器进行处理的重要依据，我们可以通过 RequestContext.getCurrentContext()来访问这些信息。另外，我们还可以在该实现中找到一些对 HTTP 头请求进行处理的逻辑，其中包含了一些耳熟能详的头域，比如 X-Forwarded-Host、X-Forwarded-Port。另外，对于这些头域的记录是通过 zuul.addProxyHeaders 参数进行控制的，而这个参数的默认值为 true，所以 Zuul 在请求跳转时默认会为请求增加 X-Forwarded-*头域，包括 X-Forwarded-Host、X-Forwarded-Port、X-Forwarded-For、X-Forwarded-Prefix、X-Forwarded-Proto。也可以通过设置 zuul.addProxyHeaders=false 关闭对这些头域的添加动作。

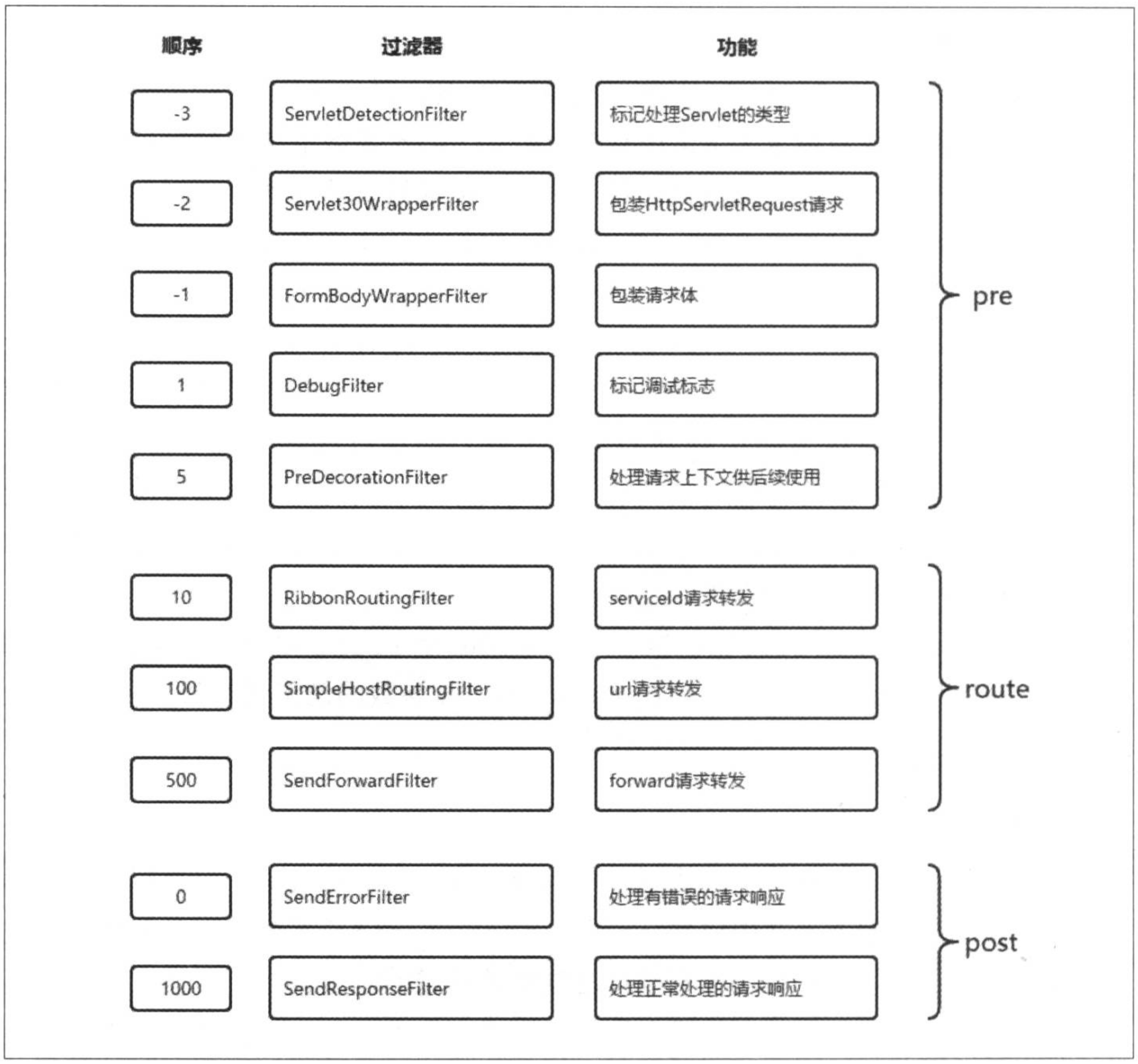
route 过滤器

- RibbonRoutingFilter：它的执行顺序为 10，是 route 阶段第一个执行的过滤器。该过滤器只对请求上下文中存在 serviceId 参数的请求进行处理，即只对通过 serviceId 配置路由规则的请求生效。而该过滤器的执行逻辑就是面向服务路由的核心，它通过使用 Ribbon 和 Hystrix 来向服务实例发起请求，并将服务实例的请求结果返回。
- SimpleHostRoutingFilter：它的执行顺序为 100，是 route 阶段第二个执行的过滤器。该过滤器只对请求上下文中存在 routeHost 参数的请求进行处理，即只对通过 url 配置路由规则的请求生效。而该过滤器的执行逻辑就是直接向 routeHost 参数的物理地址发起请求，从源码中我们可以知道该请求是直接通过 httpclient 包实现的，而没有使用 Hystrix 命令进行包装，所以这类请求并没有线程隔离和断路器的保护。
- SendForwardFilter：它的执行顺序为 500，是 route 阶段第三个执行的过滤器。该过滤器只对请求上下文中存在 forward.to 参数的请求进行处理，即用来处理路由规则中的 forward 本地跳转配置。

post 过滤器

- SendErrorFilter：它的执行顺序为 0，是 post 阶段第一个执行的过滤器。该过滤器仅在请求上下文中包含 error.status_code 参数（由之前执行的过滤器设置的错误编码）并且还没有被该过滤器处理过的时候执行。而该过滤器的具体逻辑就是利用请求上下文中的错误信息来组成一个 forward 到 API 网关/error 错误端点的请求来产生错误响应。
- SendResponseFilter：它的执行顺序为 1000，是 post 阶段最后执行的过滤器。该过滤器会检查请求上下文中是否包含请求响应相关的头信息、响应数据流或是响应体，只有在包含它们其中一个的时候执行处理逻辑。而该过滤器的处理逻辑就是利用请求上下文的响应信息来组织需要发送回客户端的响应内容。

下图对上述过滤器根据顺序、名称、功能、类型做了综合整理，可以帮助我们在自定义过滤器或是扩展过滤器的时候用来参考并全面地考虑整个请求生命周期的处理过程。



异常处理

通过上面请求生命周期和核心过滤器的介绍，我们会发现在核心过滤器中并没有实现 error 阶段的过滤器。那么当过滤器出现异常的时候需要如何处理呢？我们不妨在快速入门示例中做一个简单的试验，来看看过滤器中的异常需要被如何处理。

首先，我们尝试创建一个 pre 类型的过滤器，并在该过滤器的 run 方法实现中抛出一个异常。比如下面的实现，在 run 方法中调用的 doSomething 方法将抛出 RuntimeException 异常。

```
@Component
public class ThrowExceptionFilter extends ZuulFilter {

    private static Logger log =
```

```
LoggerFactory.getLogger(ThrowExceptionFilter.class);

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 0;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        log.info("This is a pre filter, it will throw a RuntimeException");
        doSomething();
        return null;
    }

    private void doSomething() {
        throw new RuntimeException("Exist some errors...");
    }

}
```

注意在该类的定义上添加@Component 注解，让 Spring 能够创建该过滤器实例，或者也可以像快速入门的例子中那样使用@Bean 注解在应用主类中为其创建实例。

在添加了上面的过滤器之后，我们可以将该应用以及之前的相关应用运行起来，并根据网关配置的路由规则访问服务接口，比如 http://localhost:5555/api-a/hello。此时我们会发现，在 API 网关服务的控制台中输出了 ThrowExceptionFilter 的过滤逻辑中的日志信息，但是并没有输出任何异常信息，同时发起的请求也没有获得任何响应结果。为什么会出现这样的情况呢？我们又该如何在过滤器中处理异常呢？翻遍 Spring Cloud Zuul 的文档以及 Netflix Zuul 的 Wiki 都没有找到相关的内容，但是这个问题却又是开发过程中普遍存在的。所以在本节中，我们将详细分析核心过滤器的异常处理机制以及如何在自定义过滤器中处理异常等内容。

try-catch 处理

回想一下，我们在上一节中介绍的所有核心过滤器，是否记得有一个 post 过滤器 SendErrorFilter 是用来处理异常信息的？根据正常的处理流程，该过滤器会处理异常信息，那么这里没有出现任何异常信息说明很有可能就是这个过滤器没有被执行。所以，不妨来详细看看 SendErrorFilter 的 shouldFilter 函数：

```
public boolean shouldFilter() {
    RequestContext ctx = RequestContext.getCurrentContext();
    return ctx.containsKey("error.status_code") && !ctx.getBoolean(SEND_ERROR_
FILTER_RAN, false);
}
```

可以看到，该方法的返回值中有一个重要的判断依据 ctx.containsKey("error.status_code")，也就是说请求上下文中必须有 error.status_code 参数，我们实现的 ThrowExceptionFilter 中并没有设置这个参数，所以自然不会进入 SendErrorFilter 过滤器的处理逻辑。那么如何使用这个参数呢？可以看一下 route 类型的几个过滤器，由于这些过滤器会对外发起请求，所以肯定会有异常需要处理，比如 RibbonRoutingFilter 的 run 方法实现如下：

```
public Object run() {
    RequestContext context = RequestContext.getCurrentContext();
    this.helper.addIgnoredHeaders();
    try {
        RibbonCommandContext commandContext = buildCommandContext(context);
        ClientHttpResponse response = forward(commandContext);
        setResponse(response);
        return response;
    }
    catch (ZuulException ex) {
        context.set(ERROR_STATUS_CODE, ex.nStatusCode);
        context.set("error.message", ex.errorCause);
        context.set("error.exception", ex);
    }
    catch (Exception ex) {
        context.set("error.status_code", HttpServletResponse.SC_INTERNAL_SERVER_
ERROR);
        context.set("error.exception", ex);
    }
    return null;
}
```

可以看到，整个发起请求的逻辑都采用了 try-catch 块处理。在 catch 异常的处理逻辑中并没有做任何输出操作，而是向请求上下文中添加了一些 error 相关的参数，主要有下面三个参数。

- error.status_code：错误编码。
- error.exception：Exception 异常对象。
- error.message：错误信息。

其中，error.status_code 参数就是 SendErrorFilter 过滤器用来判断是否需要执行的重要参数。分析到这里，实现异常处理的大致思路就开始明朗了，我们可以参考 RibbonRoutingFilter 的实现对 ThrowExceptionFilter 的 run 方法做一些异常处理的改造，具体如下：

```
public Object run() {
    log.info("This is a pre filter, it will throw a RuntimeException");
    RequestContext ctx = RequestContext.getCurrentContext();
    try {
        doSomething();
    } catch (Exception e) {
        ctx.set("error.status_code", HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        ctx.set("error.exception", e);
    }
    return null;
}
```

通过上面的改造之后，我们再尝试访问之前的接口，这个时候我们可以得到如下响应内容：

```
{
  "timestamp": 1481674980376,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "java.lang.RuntimeException",
  "message": "Exist some errors..."
}
```

此时，异常信息已经被 SendErrorFilter 过滤器正常处理并返回给客户端了，同时在网关的控制台中也输出了异常信息。从返回的响应信息中，可以看到几个之前我们在请求上下文中设置的内容，它们的对应关系如下所示。

- status：对应 error.status_code 参数的值。
- exception：对应 error.exception 参数中 Exception 的类型。
- message：对应 error.exception 参数中 Exception 的 message 信息。对于 message 的信息，我们在过滤器中还可以通过 ctx.set("error.message", "自定义异常消息")；来定义更友好的错误信息。SendErrorFilter 会优先取 error.message 作为返回的 message 内容，如果没有的话才会使用 Exception 中的 message 信息。

ErrorFilter 处理

通过上面的分析与实验，我们已经知道如何在过滤器中正确处理异常，让错误信息能够顺利地流转到后续的 SendErrorFilter 过滤器来组织和输出。但是，即使我们不断强调要在过滤器中使用 try-catch 来处理业务逻辑并向请求上下文中添加异常信息，但是不可控的人为因素、意料之外的程序因素等，依然会使得一些异常从过滤器中抛出，对于意外抛出的异常又会导致没有控制台输出也没有任何响应信息的情况出现，那么是否有什么好的方法来为这些异常做一个统一的处理呢？

这个时候，我们就可以用到 error 类型的过滤器了。由于在请求生命周期的 pre、route、post 三个阶段中有异常抛出的时候都会进入 error 阶段的处理，所以可以通过创建一个 error 类型的过滤器来捕获这些异常信息，并根据这些异常信息在请求上下文中注入需要返回给客户端的错误描述。这里我们可以直接沿用在 try-catch 处理异常信息时用的那些 error 参数，这样就可以让这些信息被 SendErrorFilter 捕获并组织成响应消息返回给客户端。比如，下面的代码就实现了这里所描述的一个过滤器：

```
public class ErrorFilter extends ZuulFilter {

    Logger log = LoggerFactory.getLogger(ErrorFilter.class);

    @Override
    public String filterType() {
        return "error";
    }

    @Override
    public int filterOrder() {
        return 10;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        Throwable throwable = ctx.getThrowable();
        log.error("this is a ErrorFilter : {}", throwable.getCause().getMessage());
        ctx.set("error.status_code", HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        ctx.set("error.exception", throwable.getCause());
```

```
        return null;
    }

}
```

在将该过滤器加入 API 网关服务之后，我们可以尝试使用之前介绍 try-catch 处理时实现的 ThrowExceptionFilter（不包含异常处理机制的代码），让该过滤器能够抛出异常。这个时候我们再通过 API 网关来访问服务接口。此时，我们就可以在控制台中看到 ThrowExceptionFilter 过滤器抛出的异常信息，并且请求响应中也能获得如下的错误信息内容，而不是什么信息都没有的情况了。

```
{
  "timestamp": 1481674993561,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "java.lang.RuntimeException",
  "message": "Exist some errors..."
}
```

不足与优化

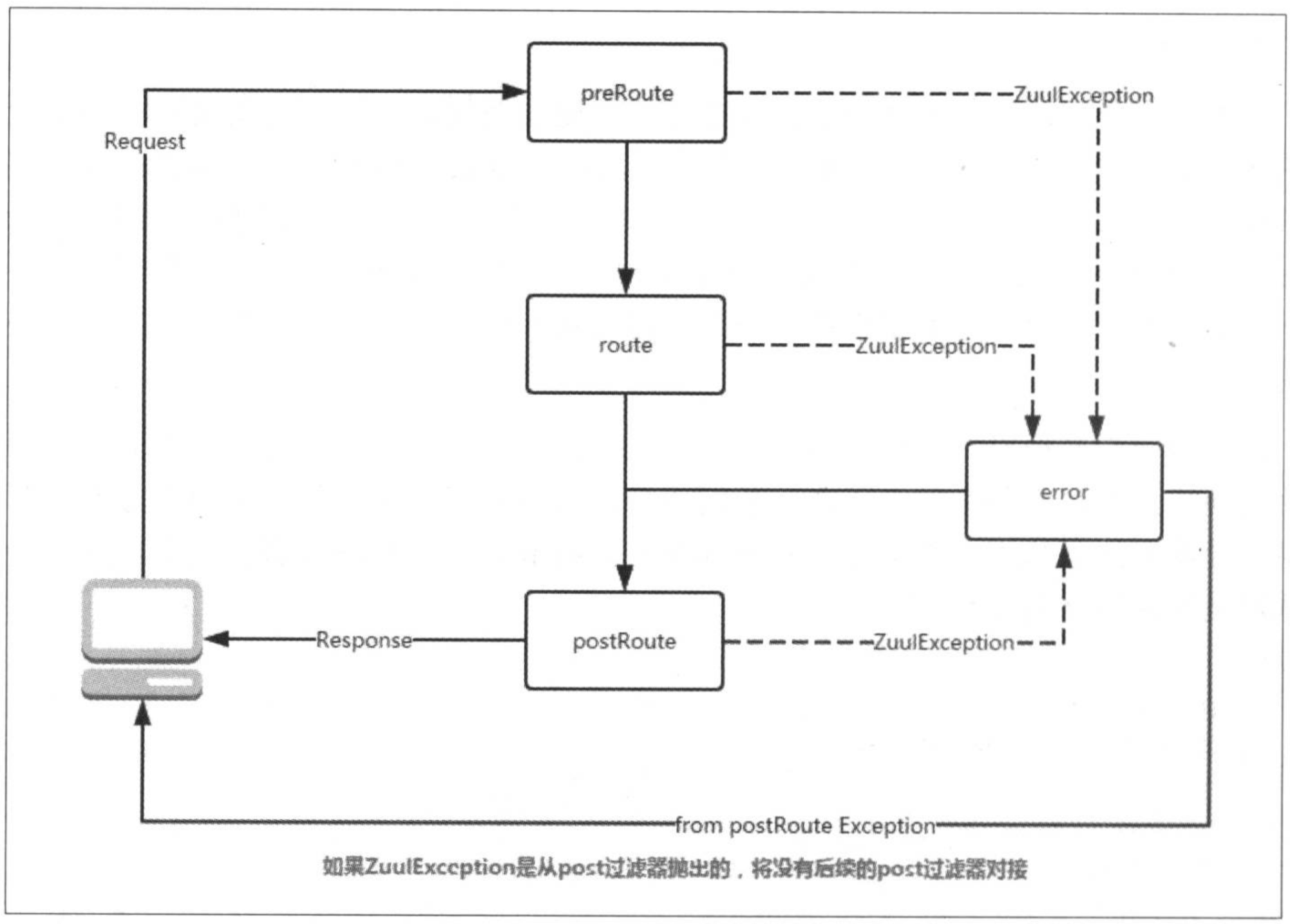
到这里，我们已经基本掌握了在核心过滤器处理逻辑之下，对自定义过滤器中处理异常的两种基本解决方法：一种是通过在各个阶段的过滤器中增加 try-catch 块，实现过滤器内部的异常处理；另一种是利用 error 类型过滤器的生命周期特性，集中处理 pre、route、post 阶段抛出的异常信息。通常情况下，我们可以将这两种手段同时使用，其中第一种是对开发人员的基本要求；而第二种是对第一种处理方式的补充，以防止意外情况的发生。

这样的异常处理机制看似已经完美，但是如果在多一些应用实践或源码分析之后，我们会发现依然存在一些不足。下面，我们不妨跟着源码来看看，到底上面的方案还有哪些不足之处需要注意和进一步优化。先来看看外部请求到达 API 网关服务之后，各个阶段的过滤器是如何进行调度的：

```
try {
    preRoute();
} catch (ZuulException e) {
    error(e);
    postRoute();
    return;
}
try {
    route();
} catch (ZuulException e) {
    error(e);
```

```
        postRoute();
        return;
    }
    try {
        postRoute();
    } catch (ZuulException e) {
        error(e);
        return;
    }
```

上面的代码源自 com.netflix.zuul.http.ZuulServlet 的 service 方法实现，它定义了 Zuul 处理外部请求过程时，各个类型过滤器的执行逻辑。从代码中我们可以看到三个 try-catch 块，它们依次分别代表了 pre、route、post 三个阶段的过滤器调用。在 catch 的异常处理中我们可以看到它们都会被 error 类型的过滤器进行处理（之前使用 error 过滤器来定义统一的异常处理也正是利用了这个特性）；error 类型的过滤器处理完毕之后，除了来自 post 阶段的异常之外，都会再被 post 过滤器进行处理。所以，上面代码中各个处理阶段的逻辑如下图所示：



通过图中的分析，我们可以看到，对于从 post 过滤器中抛出异常的情况，在经过 error 过滤器处理之后，就没有其他类型的过滤器来接手了，这就是使用之前所述方案存

在不足之处的根源。回想一下之前实现的两种异常处理方法，其中非常核心的一点是，这两种处理方法都在异常处理时向请求上下文中添加了一系列的 error.*参数，而这些参数真正起作用的地方是在 post 阶段的 SendErrorFilter，在该过滤器中会使用这些参数来组织内容返回给客户端。而对于 post 阶段抛出异常的情况下，由 error 过滤器处理之后并不会再调用 post 阶段的请求，自然这些 error.*参数也就不会被 SendErrorFilter 消费输出。所以，如果我们在自定义 post 过滤器的时候，没有正确处理异常，就依然有可能出现日志中没有异常但请求响应内容为空的问题。我们可以通过将之前 ThrowExceptionFilter 的 filterType 修改为 post 来验证这个问题的存在，注意去掉 try-catch 块的处理，让它能够抛出异常。

解决上述问题的方法有很多种，最直接的是我们可以在实现 error 过滤器的时候，直接组织结果返回就能实现效果。但是这样做的缺点也很明显，对于错误信息组织和返回的代码实现会存在多份，这样非常不利于日后的代码维护工作。所以为了保持对异常返回处理逻辑的一致性，我们还是希望将 post 过滤器抛出的异常交给 SendErrorFilter 来处理。

在前文中，我们已经实现了一个 ErrorFilter 来捕获 pre、route、post 过滤器抛出的异常，并组织 error.*参数保存到请求的上下文中。由于我们的目标是沿用 SendErrorFilter，这些 error.*参数依然对我们有用，所以可以继续沿用该过滤器，让它在 post 过滤器抛出异常的时候，继续组织 error.*参数，只是这里我们已经无法将这些 error.*参数再传递给 SendErrorFilter 过滤器来处理了。所以，我们需要在 ErrorFilter 过滤器之后再定义一个 error 类型的过滤器，让它来实现 SendErrorFilter 的功能，但是这个 error 过滤器并不需要处理所有出现异常的情况，它仅对 post 过滤器抛出的异常有效。根据上面的思路，我们完全可以创建一个继承自 SendErrorFilter 的过滤器，复用它的 run 方法，然后重写它的类型、顺序以及执行条件，实现对原有逻辑的复用，具体实现如下：

```
@Component
public class ErrorExtFilter extends SendErrorFilter {

    @Override
    public String filterType() {
        return "error";
    }

    @Override
    public int filterOrder() {
        return 30;  // 大于ErrorFilter的值
    }
```

```
    @Override
    public boolean shouldFilter() {
        // TODO 判断：仅处理来自post过滤器引起的异常
        return true;
    }

}
```

到这里，我们在过滤器调度上的实现思路已经很清晰了，但是又有一个问题出现在我们面前：怎么判断引起异常的过滤器来自什么阶段呢？shouldFilter 方法该如何实现，对于这个问题，我们第一反应会寄希望于请求上下文 RequestContext 对象，可是在查阅文档和源码后发现其中并没有存储异常来源的内容，所以我们不得不扩展原来的过滤器处理逻辑。当有异常抛出的时候，记录下抛出异常的过滤器，这样我们就可以在 ErrorExtFilter 过滤器的 shouldFilter 方法中获取并以此判断异常是否来自 post 阶段的过滤器了。

为了扩展过滤器的处理逻辑，为请求上下文增加一些自定义属性，我们需要深入了解 Zuul 过滤器的核心处理器：com.netflix.zuul.FilterProcessor。该类中定义了下面列出的过滤器调用和处理相关的核心方法。

- getInstance()：该方法用来获取当前处理器的实例。
- setProcessor(FilterProcessor processor)：该方法用来设置处理器实例，可以使用此方法来设置自定义的处理器。
- processZuulFilter(ZuulFilter filter)：该方法定义了用来执行 filter 的具体逻辑，包括对请求上下文的设置，判断是否应该执行，执行时一些异常的处理等。
- getFiltersByType(String filterType)：该方法用来根据传入的 filterType 获取 API 网关中对应类型的过滤器，并根据这些过滤器的 filterOrder 从小到大排序，组织成一个列表返回。
- runFilters(String sType)：该方法会根据传入的 filterType 来调用 getFiltersByType(String filterType)获取排序后的过滤器列表，然后轮询这些过滤器，并调用 processZuulFilter(ZuulFilter filter)来依次执行它们。
- preRoute()：调用 runFilters("pre")来执行所有 pre 类型的过滤器。
- route()：调用 runFilters("route")来执行所有 route 类型的过滤器。
- postRoute()：调用 runFilters("post")来执行所有 post 类型的过滤器。
- error()：调用 runFilters("error")来执行所有 error 类型的过滤器。

根据之前的设计，可以直接扩展 processZuulFilter(ZuulFilter filter)，当过滤器执行抛出异常的时候，我们捕获它，并向请求上下中记录一些信息。比如下面的具体实现：

```
public class DidiFilterProcessor extends FilterProcessor {

    @Override
    public Object processZuulFilter(ZuulFilter filter) throws ZuulException {
        try {
            return super.processZuulFilter(filter);
        } catch (ZuulException e) {
            RequestContext ctx = RequestContext.getCurrentContext();
            ctx.set("failed.filter", filter);
            throw e;
        }
    }

}
```

在上面代码的实现中，我们创建了一个 FilterProcessor 的子类，并重写了 processZuulFilter(ZuulFilter filter)，虽然主逻辑依然使用了父类的实现，但是在最外层，我们为其增加了异常捕获，并在异常处理中为请求上下文添加了 failed.filter 属性，以存储抛出异常的过滤器实例。在实现了这个扩展之后，我们也就可以完善之前 ErrorExtFilter 中的 shouldFilter()方法了，通过从请求上下文中获取该信息做出正确的判断，具体实现如下：

```
@Component
public class ErrorExtFilter extends SendErrorFilter {

    @Override
    public String filterType() {
        return "error";
    }

    @Override
    public int filterOrder() {
        return 30;  // 大于ErrorFilter的值
    }

    @Override
    public boolean shouldFilter() {
        // 判断：仅处理来自post过滤器引起的异常
        RequestContext ctx = RequestContext.getCurrentContext();
        ZuulFilter failedFilter = (ZuulFilter) ctx.get("failed.filter");
```

```
        if(failedFilter != null && failedFilter.filterType().equals("post")) {
            return true;
        }
        return false;
    }

}
```

到这里，我们的优化任务还没有完成，因为扩展的过滤器处理类并还没有生效。最后，需要在应用主类中，通过调用 FilterProcessor.setProcessor(new DidiFilterProcessor());方法来启用自定义的核心处理器以完成我们的优化目标。

自定义异常信息

在实现了对自定义过滤器中的异常处理之后，实际应用到业务系统中时，往往默认的错误信息并不符合系统设计的响应格式，那么我们就需要对返回的异常信息进行定制。对于如何定制这个错误信息有很多种方法可以实现。最直接的是，可以编写一个自定义的 post 过滤器来组织错误结果，该方法实现起来简单粗暴，完全可以参考 SendErrorFilter 的实现，然后直接组织请求响应结果而不是 forward 到/error 端点，只是使用该方法时需要注意：为了替代 SendErrorFilter，还需要禁用 SendErrorFilter 过滤器，禁用的配置方法在后文中会详细介绍。

那么如果不采用重写过滤器的方式，依然想要使用 SendErrorFilter 来处理异常返回的话，我们要如何定制返回的响应结果呢？这个时候，我们的关注点就不能放在 Zuul 的过滤器上了，因为错误信息的生成实际上并不是由 Spring Cloud Zuul 完成的。我们在介绍 SendErrorFilter 的时候提到过，它会根据请求上下中保存的错误信息来组织一个 forward 到/error 端点的请求来获取错误响应，所以我们的扩展目标转移到了对/error 端点的实现。

/error 端点的实现来源于 Spring Boot 的 org.springframework.boot.autoconfigure.web.BasicErrorController，它的具体定义如下：

```
    @RequestMapping
    @ResponseBody
    public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
        Map<String, Object> body = getErrorAttributes(request,
                isIncludeStackTrace(request, MediaType.ALL));
        HttpStatus status = getStatus(request);
        return new ResponseEntity<Map<String, Object>>(body, status);
    }
```

从源码中可以看到，它的实现非常简单，通过调用 getErrorAttributes 方法来根据请求参数组织错误信息的返回结果，而这里的 getErrorAttributes 方法会将具体组织逻

辑委托给 org.springframework.boot.autoconfigure.web.ErrorAttributes 接口提供的 getErrorAttributes 来实现。在 Spring Boot 的自动化配置机制中，默认会采用 org.springframework.boot.autoconfigure.web.DefaultErrorAttributes 作为该接口的实现。如下代码所示，在定义 Error 处理的自动化配置中，该接口的默认实现采用了@ConditionalOnMissingBean 修饰，说明 DefaultErrorAttributes 对象实例仅在没有 ErrorAttributes 接口的实例时才会被创建来使用，所以我们只需要自己编写一个自定义的 ErrorAttributes 接口实现类，并创建它的实例就能替代这个默认的实现，从而达到自定义错误信息的效果了。

```
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class })
@ConditionalOnWebApplication
@AutoConfigureBefore(WebMvcAutoConfiguration.class)
@Configuration
public class ErrorMvcAutoConfiguration {

  @Autowired
  private ServerProperties properties;

  @Bean
  @ConditionalOnMissingBean(value = ErrorAttributes.class, search =
SearchStrategy.CURRENT)
  public DefaultErrorAttributes errorAttributes() {
    return new DefaultErrorAttributes();
  }
  ...
}
```

举个简单的例子，比如我们不希望将 exception 属性返回给客户端，那么就可以编写一个自定义的实现，它可以基于 DefaultErrorAttributes，然后重写 getErrorAttributes 方法，从原来的结果中将 exception 移除即可，具体实现如下：

```
public class DidiErrorAttributes extends DefaultErrorAttributes {

  @Override
  public Map<String, Object> getErrorAttributes (
        RequestAttributes requestAttributes, boolean includeStackTrace){
      Map<String, Object> result = super.getErrorAttributes(requestAttributes,
includeStackTrace);
      result.remove("exception");
      return result;
  }

}
```

最后，为了让自定义的错误信息生成逻辑生效，需要在应用主类中加入如下代码，为其创建实例来替代默认的实现：

```
@Bean
public DefaultErrorAttributes errorAttributes() {
    return new DidiErrorAttributes();
}
```

通过上面介绍的方法，我们就能基于Zuul的核心过滤器来灵活地自定义错误返回信息，以满足实际应用系统的响应格式了。

禁用过滤器

不论是核心过滤器还是自定义过滤器，只要在 API 网关应用中为它们创建了实例，那么默认情况下，它们都是启用状态的。那么如果有些过滤器我们不想使用了，如何禁用它们呢？大多情况下初识 Zuul 的使用者第一反应就是通过重写 shouldFilter 逻辑，让它返回 false，这样该过滤器对于任何请求都不会被执行，基本实现了对过滤器的禁用。但是，对于自定义过滤器来说似乎是实现了过滤器不生效的功能，但是这样的做法缺乏灵活性。由于直接要修改过滤器逻辑，我们不得不重新编译程序，并且如果该过滤器在未来一段时间还有可能被启用的时候，那么就又得修改代码并编译程序。同时，对于核心过滤器来说，就更为麻烦，我们不得不获取源码来进行修改和编译。

实际上，在 Zuul 中特别提供了一个参数来禁用指定的过滤器，该参数的配置格式如下：

```
zuul.<SimpleClassName>.<filterType>.disable=true
```

其中，<SimpleClassName>代表过滤器的类名，比如快速入门示例中的AccessFilter；<filterType>代表过滤器类型，比如快速入门示例中 AccessFilter 的过滤器类型 pre。所以，如果我们想要禁用快速入门示例中的 AccessFilter 过滤器，只需要在 application.properties 配置文件中增加如下配置即可：

```
zuul.AccessFilter.pre.disable=true
```

该参数配置除了可以对自定义的过滤器进行禁用配置之外，很多时候可以用它来禁用 Spring Cloud Zuul 中默认定义的核心过滤器。这样我们就可以抛开 Spring Cloud Zuul 自带的那套核心过滤器，实现一套更符合我们实际需求的处理机制。

通过本节对异常处理的介绍，也许这些方法并一定完全适用读者所接触的实际系统，但是通过这些分析可以帮助我们进一步理解 Zuul 过滤器的运行机制，帮助我们基于 Spring Cloud Zuul 去实现更适合自身系统的 API 网关服务。

动态加载

在微服务架构中，由于 API 网关服务担负着外部访问统一入口的重任，它同其他应用不同，任何关闭应用和重启应用的操作都会使系统对外服务停止，对于很多 7 × 24 小时服务的系统来说，这样的情况是绝对不被允许的。所以，作为最外部的网关，它必须具备动态更新内部逻辑的能力，比如动态修改路由规则、动态添加/删除过滤器等。

通过 Zuul 实现的 API 网关服务当然也具备了动态路由和动态过滤器的能力。我们可以在不重启 API 网关服务的前提下，为其动态修改路由规则和添加或删除过滤器。下面我们分别来看看如何通过 Zuul 来实现动态 API 网关服务。

动态路由

通过之前对请求路由的详细介绍，我们可以发现对于路由规则的控制几乎都可以在配置文件 `application.properties` 或 `application.yaml` 中完成。既然这样，对于如何实现 Zuul 的动态路由，我们很自然地会将它与 Spring Cloud Config 的动态刷新机制联系到一起。只需将 API 网关服务的配置文件通过 Spring Cloud Config 连接的 Git 仓库存储和管理，我们就能轻松实现动态刷新路由规则的功能。

在介绍如何具体实现 API 网关服务的动态路由之前，我们首先需要一个连接到 Git 仓库的分布式配置中心 `config-server` 应用。如果还没有搭建过分布式配置中心的话，建议先阅读第 8 章的内容，对分布式配置中心的运作机制有一个基础的了解，并构建一个 `config-server` 应用，以配合完成下面的内容。

在具备了分布式配置中心之后，为了方便理解，我们重新构建一个 API 网关服务，该服务的配置中心不再配置于本地工程中，而是从 `config-server` 中获取，构建过程如下所示。

- 创建一个基础的 Spring Boot 工程，命名为 `api-gateway-dynamic-route`。
- 在 `pom.xml` 中引入对 `zuul`、`eureka` 和 `config` 的依赖，具体内容如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <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-starter-config</artifactId>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Brixton.SR5</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

- 在/resource 目录下创建配置文件 bootstrap.properties，并在该文件中指定 config-server 和 eureka-server 的具体地址，以获取应用的配置文件和实现服务注册与发现。

```
spring.application.name=api-gateway
server.port=5556

spring.cloud.config.uri=http://localhost:7001/

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

- 创建用来启动 API 网关的应用主类。这里我们需要使用@RefreshScope 注解来将 Zuul 的配置内容动态化，具体实现如下：

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

    @Bean
```

```
    @RefreshScope
    @ConfigurationProperties("zuul")
    public ZuulProperties zuulProperties() {
        return new ZuulProperties();
    }
}
```

- 在完成了所有程序相关的编写之后，我们还需要在 Git 仓库中增加网关的配置文件，取名为 api-gateway.properties。在配置文件中，我们为 API 网关服务预定义以下路由规则，比如：

```
zuul.routes.service-a.path=/service-a/**
zuul.routes.service-a.serviceId=hello-service
zuul.routes.service-b.path=/service-b/**
zuul.routes.service-b.url=http://localhost:8001/
```

对于 API 网关服务在 Git 仓库中的配置文件名称完全取决于网关应用配置文件 bootstrap.properties 中 spring.application.name 属性的配置值。由于本章我们主要介绍 API 网关的使用，所以这里省略了关于 label 和 profile 的配置，默认会使用 master 分支和 default 配置文件，更细致的配置方式读者可查阅第 8 章介绍的内容。

测试与验证

在完成了上述内容之后，我们可以将 config-server、eureka-server、api-gateway-dynamic-route 以及配置文件中路由规则指向的具体服务，比如 hello-service 启动起来。此时，在 API 网关应用 api-gateway-dynamic-route 的控制台中，我们可以看到它输出了从 config-server 中获取配置文件过程的日志信息，根据这些信息，可以判断获取配置信息的路径等内容是否正确。具体输出内容如下所示：

```
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://localhost:7001/
    c.c.c.ConfigServicePropertySourceLocator : Located environment: name=api-gateway,
profiles=[default], label=master, version=1dab6126ca2972c5409fcb089934b057cf2bf77d
    b.c.PropertySourceBootstrapConfiguration : Located property source:
CompositePropertySource [name='configService', propertySources=[MapPropertySource
[name='overrides'], MapPropertySource
[name='http://git.oschina.net/didispace/SpringCloud-Learning/spring_cloud_in_action
/config-repo/api-gateway.properties']]]
    com.didispace.Application                : No active profile set, falling back to
default profiles: default
```

在 api-gateway-dynamic-route 启动完成之后，可以通过对 API 网关服务调用 /routes 接口来获取当前网关上的路由规则，根据上述配置我们可以得到如下返回信息：

```
{
  "/service-a/**": "hello-service",
  "/service-b/**": "http://localhost:8001/"
}
```

我们可以先尝试着通过上述路由规则访问一下对应路由服务提供的接口，验证当前的路由规则是否生效。接着，我们开始尝试动态修改这些路由规则，只需要以下两步。

- 修改 Git 仓库中的 api-gateway.properties 配置文件，比如，将 service-a 路由规则的 path 修改为/aaa/**，并把 service-b 的路由规则从 url 修改为 serviceId 方式。具体配置如下：

```
zuul.routes.service-a.path=/aaa/**
zuul.routes.service-a.serviceId=hello-service

zuul.routes.service-b.path=/service-b/**
zuul.routes.service-b.url=hello-service
```

- 在修改完配置文件之后，将修改内容推送到远程仓库。然后，通过向 api-gateway-dynamic-route 的/refresh 接口发送 POST 请求来刷新配置信息。当配置文件有修改的时候，该接口会返回被修改的属性名称，根据上面的修改，我们会得到如下返回信息：

```
[
  "zuul.routes.service-b.serviceId",
  "zuul.routes.service-b.url",
  "zuul.routes.service-a.path"
]
```

由于修改了 service-b 路由的 url 方式为 serviceId 方式，相当于删除了 url 参数配置，增加了 serviceId 参数配置，所以这里会出现两条关于 service-b 路由的变更信息。

到这里，我们就已经完成了路由规则的动态刷新，可以继续通过 API 网关服务的/routes 接口来查看当前的所有路由规则，该接口将返回如下信息：

```
{
  "/aaa/**": "hello-service",
  "/service-b/**": "hello-service"
}
```

从/routes 接口的信息中我们可以看到，路由 service-a 的匹配表达式被修改成了/aaa/**，而路由 service-b 的映射目标从原来的物理地址 http://localhost:8001/ 修改成了 hello-service 服务。

通过本节对动态路由加载内容的介绍，我们可以看到，通过 Zuul 构建的 API 网关服务对于动态路由的实现总体上来说还是非常简单的。美中不足的一点是，Spring Cloud Config 并没有 UI 管理界面，我们不得不通过 Git 客户端来进行修改和配置，所以在使用的时候并不是特别方便，当然有条件的团队可以自己开发一套 UI 界面来帮助管理这些路由规则。

动态过滤器

既然通过 Zuul 构建的 API 网关服务能够轻松地实现动态路由的加载，那么对于 API 网关服务的另外一大重要功能——请求过滤器的动态加载自然也不能放过，只是对于请求过滤器的动态加载与请求路由的动态加载在实现机制上会有所不同。这个不难理解，通过之前介绍的请求路由和请求过滤的示例，我们可以看到请求路由通过配置文件就能实现，而请求过滤则都是通过编码实现。所以，对于实现请求过滤器的动态加载，我们需要借助基于 JVM 实现的动态语言的帮助，比如 Groovy。

下面，我们将通过一个简单的示例来演示如何构建一个具备动态加载 Groovy 过滤器能力的 API 网关服务的详细步骤。

- 创建一个基础的 Spring Boot 工程，命名为 `api-gateway-dynamic-filter`。
- 在 `pom.xml` 中引入对 `zuul`、`eureka` 和 `groovy` 的依赖，具体内容如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <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.codehaus.groovy</groupId>
        <artifactId>groovy-all</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
```

```
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Brixton.SR5</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

- 在/resource 目录下创建配置文件 application.properties，并在该文件中设置 API 网关服务的应用名和服务端口号，以及指定 eureka-server 的具体地址。同时，再配置一个用于测试的路由规则，我们可以用之前章节实现的 hello-service 为例。具体配置内容如下：

```
spring.application.name=api-gateway
server.port=5555

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

zuul.routes.hello.path=/hello-service/**
zuul.routes.hello.serviceId=hello-service
```

到这里一个基础的 API 网关服务已经构建完成，下面我们来为它增加动态加载过滤器的功能。

- 为了方便使用，我们先自定义一些用来配置动态加载过滤器的参数，并将它们的配置值加入到 application.properties 中，比如：

```
zuul.filter.root=filter
zuul.filter.interval=5
```

  其中，zuul.filter.root 用来指定动态加载的过滤器存储路径；zuul.filter.interval 用来配置动态加载的间隔时间，以秒为单位。
- 创建用来加载自定义属性的配置类，命名为 FilterConfiguration，具体内容如下：

```
@ConfigurationProperties("zuul.filter")
public class FilterConfiguration {

  private String root;
  private Integer interval;

  public String getRoot() {
     return root;
```

```
    }

    public void setRoot(String root) {
        this.root = root;
    }

    public Integer getInterval() {
        return interval;
    }

    public void setInterval(Integer interval) {
        this.interval = interval;
    }
}
```

- 创建应用启动主类，并在该类中引入上面定义的 FilterConfiguration 配置，并创建动态加载过滤器的实例。具体内容如下：

```
@EnableZuulProxy
@EnableConfigurationProperties({FilterConfiguration.class})
@SpringCloudApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

    @Bean
    public FilterLoader filterLoader(FilterConfiguration filterConfiguration) {
        FilterLoader filterLoader = FilterLoader.getInstance();
        filterLoader.setCompiler(new GroovyCompiler());
        try {
            FilterFileManager.setFilenameFilter(new GroovyFileFilter());
            FilterFileManager.init(
                    filterConfiguration.getInterval(),
                    filterConfiguration.getRoot() + "/pre",
                    filterConfiguration.getRoot() + "/post");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return filterLoader;
    }

}
```

至此，我们就已经完成了为基础的 API 网关服务增加动态加载过滤器的能力。根据上

面的定义，API 网关应用会每隔 5 秒，从 API 网关服务所在位置的 filter/pre 和 filter/post 目录下获取 Groovy 定义的过滤器，并对其进行编译和动态加载使用。对于动态加载的时间间隔，可通过 zuul.filter.interval 参数来修改。而加载过滤器实现类的根目录可通过 zuul.filter.root 调整根目录的位置来修改，但是对于根目录的子目录，这里写死了读取/pre 和/post 目录，实际使用的时候读者可以做进一步扩展。

在完成上述构建之后，我们可以将涉及的服务，比如 eureka-server、hello-service 以及上述实现的 API 网关服务都启动起来。在没有加入任何自定义过滤器的时候，根据路由规则定义，我们可以尝试向 API 网关服务发起请求：http://localhost:5555/hello-service/hello，如果配置正确，该请求会被 API 网关服务路由到 hello-service 上，并返回输出 Hello World。

接下来，我们可以在 filter/pre 和 filter/post 目录下，用 Groovy 来创建一些过滤器来看看实际效果，举例如下。

- 在 filter/pre 目录下创建一个 pre 类型的过滤器，命名为 PreFilter.groovy。由于 pre 类型的过滤器在请求路由前执行，通常用来做一些签名校验的功能，所以我们可以在过滤器中输出一些请求相关的信息，比如下面的实现：

```
class PreFilter extends ZuulFilter {
  Logger log = LoggerFactory.getLogger(PreFilter.class)

  @Override
  String filterType() {
    return "pre"
  }

  @Override
  int filterOrder() {
    return 1000
  }

  @Override
  boolean shouldFilter() {
    return true
  }

  @Override
  Object run() {
    HttpServletRequest request = RequestContext.getCurrentContext().getRequest()
    log.info("this is a pre filter: Send {} request to {}",
             request.getMethod(),
```

```
                request.getRequestURL().toString())
        return null
    }
}
```

在加入了该过滤器之后，不需要重启 API 网关服务，只需要稍等几秒就会生效。我们可以继续尝试向 API 网关服务发起请求：http://localhost:5555/hello-service/hello，此时在控制台中可以看到 PreFilter.groovy 过滤器中定义的日志信息，具体如下：

```
com.didispace.filter.pre.PreFilter      : this is a pre filter: Send GET request to
http://localhost:5555/ddd/hello
```

- 在 filter/post 目录下创建一个 post 类型的过滤器，命名为 PostFilter.groovy。由于 post 类型的过滤器在请求路由返回后执行，我们可以进一步对这个结果做一些处理，对微服务返回的信息做一些加工。比如下面的实现：

```
class PostFilter extends ZuulFilter{
  Logger log = LoggerFactory.getLogger(PostFilter.class)

  @Override
  String filterType() {
    return "post"
  }

  @Override
  int filterOrder() {
    return 2000
  }

  @Override
  boolean shouldFilter() {
    return true
  }

  @Override
  Object run() {
    log.info("this is a post filter: Receive response")
    HttpServletResponse response = RequestContext.getCurrentContext().getResponse()
    response.getOutputStream().print(", I am zhaiyongchao")
    response.flushBuffer()
  }
}
```

在加入了该过滤器之后，我们也不需要重启 API 网关服务，稍等几秒后就可以尝试向 API 网关服务发起请求：http://localhost:5555/hello-service/hello，

此时不仅可以在控制台中看到 PostFilter.groovy 过滤器中定义的日志输出信息，也可以从请求返回的内容发现过滤器的效果，该接口返回的内容不再是 Hello World，而是经过加工处理后的 Hello World, I am zhaiyongchao。

通过本节对动态过滤器加载的内容的介绍，可以看到，API 网关服务的动态过滤器功能可以帮助我们增强 API 网关的持续服务能力，对于网关中的处理逻辑维护也变得更为灵活，不仅可以动态地实现请求校验，还可以动态地实现对请求内容的干预。但是，目前版本下的动态过滤器还是一个半成品，从 org.springframework.cloud.netflix.zuul.ZuulFilterInitializer 的源码中我们也可以看到，对于动态过滤器的加载是被注释掉的，并且被标注了 TODO 状态。不过，目前在实际使用过程中，对于处理一些简单的常用过滤功能还是没有什么问题的，只是需要注意一些已知的问题并避开这些情况来使用即可。比如，在使用 Groovy 定义动态过滤器的时候，删除 Groovy 文件并不能从当前运行的 API 网关中移除这个过滤器，所以如果要移除的话可以通过修改 Groovy 过滤器的 shouldFilter 返回 false。另外还需要注意一点，目前的动态过滤器是无法直接注入 API 网关服务的 Spring 容器中加载的实例来使用的，比如，我们是无法直接通过注入 RestTemplate 等实例，在动态过滤器中对各个微服务发起请求的。

第 8 章

分布式配置中心：Spring Cloud Config

Spring Cloud Config 是 Spring Cloud 团队创建的一个全新项目，用来为分布式系统中的基础设施和微服务应用提供集中化的外部配置支持，它分为服务端与客户端两个部分。其中服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置仓库并为客户端提供获取配置信息、加密/解密信息等访问接口；而客户端则是微服务架构中的各个微服务应用或基础设施，它们通过指定的配置中心来管理应用资源与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。Spring Cloud Config 实现了对服务端和客户端中环境变量和属性配置的抽象映射，所以它除了适用于 Spring 构建的应用程序之外，也可以在任何其他语言运行的应用程序中使用。由于 Spring Cloud Config 实现的配置中心默认采用 Git 来存储配置信息，所以使用 Spring Cloud Config 构建的配置服务器，天然就支持对微服务应用配置信息的版本管理，并且可以通过 Git 客户端工具来方便地管理和访问配置内容。当然它也提供了对其他存储方式的支持，比如 SVN 仓库、本地化文件系统。接下来，我们从一个简单的入门示例开始学习 Spring Cloud Config 服务端以及客户端的详细构建与使用方法。

快速入门

在本节中，我们将演示如何构建一个基于 Git 存储的分布式配置中心，同时对配置的详细规则进行讲解，并在客户端中演示如何通过配置指定微服务应用的所属配置中心，并让其能够从配置中心获取配置信息并绑定到代码中的整个过程。

构建配置中心

通过 Spring Cloud Config 构建一个分布式配置中心非常简单，只需要以下三步。

- 创建一个基础的 Spring Boot 工程，命名为 `config-server`，并在 `pom.xml` 中引入下面的依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

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

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- 创建 Spring Boot 的程序主类，并添加`@EnableConfigServer` 注解，开启 Spring Cloud Config 的服务端功能。

```
@EnableConfigServer
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

- 在 application.properties 中添加配置服务的基本信息以及 Git 仓库的相关信息，如下所示：

```
spring.application.name=config-server
server.port=7001

spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/SpringCloud
-Learning/
spring.cloud.config.server.git.searchPaths=spring_cloud_in_action/config-repo
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

其中 Git 的配置信息分别表示如下内容。

- spring.cloud.config.server.git.uri：配置 Git 仓库位置。
- spring.cloud.config.server.git.searchPaths：配置仓库路径下的相对搜索位置，可以配置多个。
- spring.cloud.config.server.git.username：访问 Git 仓库的用户名。
- spring.cloud.config.server.git.password：访问 Git 仓库的用户密码。

到这里，使用一个通过 Spring Cloud Config 实现，并使用 Git 管理配置内容的分布式配置中心就完成了。我们可以将该应用先启动起来，确保没有错误产生，然后进入下面的学习内容。

配置规则详解

为了验证上面完成的分布式配置中心 config-server，根据 Git 配置信息中指定的仓库位置，在 http://git.oschina.net/didispace/SpringCloud-Learning/spring_cloud_in_action/下创建了一个 config-repo 目录作为配置仓库，并根据不同环境新建下面 4 个配置文件：

- didispace.properties
- didispace-dev.properties
- didispace-test.properties
- didispace-prod.properties

在这 4 个配置文件中均设置了一个 from 属性，并为每个配置文件分别设置了不同的值，如下所示：

- from=git-default-1.0
- from=git-dev-1.0
- from=git-test-1.0

- from=git-prod-1.0

为了测试版本控制，在该 Git 仓库的 master 分支中，我们为 from 属性加入 1.0 的后缀，同时创建一个 config-label-test 分支，并将各配置文件中的值用 2.0 作为后缀。

完成了这些准备工作之后，我们就可以通过浏览器、POSTMAN 或 CURL 等工具直接来访问我们的配置内容了。访问配置信息的 URL 与配置文件的映射关系如下所示：

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

上面的 url 会映射{application}-{profile}.properties 对应的配置文件，其中{label}对应 Git 上不同的分支，默认为 master。我们可以尝试构造不同的 url 来访问不同的配置内容，比如，要访问 config-label-test 分支，didispace 应用的 prod 环境，就可以访问这个 url：http://localhost:7001/didispace/prod/config-label-test，并获得如下返回信息：

```
{
    "name": "didispace",
    "profiles": [
        "prod"
    ],
    "label": "config-label-test",
    "version": "4c4f3909b0499b8518aba1f76e8a90b0dbad535d",
    "propertySources": [
        {
            "name": "http://git.oschina.net/didispace/SpringCloud-Learning/spring_
cloud_in_action/config-repo/didispace-prod.properties",
            "source": {
                "from": "git-prod-2.0"
            }
        },
        {
            "name": "http://git.oschina.net/didispace/SpringCloud-Learning/spring_
cloud_in_action/config-repo/didispace.properties",
            "source": {
                "from": "git-default-2.0"
            }
        }
    ]
}
```

我们可以看到该 JSON 中返回了应用名 didispace，环境名 prod，分支名 config-label-test，以及 default 环境和 prod 环境的配置内容。另外，之前没有提到过的 version，从下图我们可以观察到，它对应的是在 Git 上的 commit 号。

```
commit 4c4f3909b0499b8518aba1f76e8a90b0dbad535d
1 parent 8f4e9938bc
```

同时，我们可以看到 config-server 的控制台中还输出了下面的内容，配置服务器在从 Git 中获取配置信息后，会存储一份在 config-server 的文件系统中，实质上 config-server 是通过 git clone 命令将配置内容复制了一份在本地存储，然后读取这些内容并返回给微服务应用进行加载。

```
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@7d09c9:
startup date [Fri Sep 16 21:56:43 CST 2016]; root of context hierarchy
o.s.c.c.s.e.NativeEnvironmentRepository : Adding property source:
file:/C:/Users/ADMINI~1/AppData/Local/Temp/config-repo-9143470827238107 66/spring_cl
oud_in_action/config-repo/didispace-prod.properties
o.s.c.c.s.e.NativeEnvironmentRepository : Adding property source:
file:/C:/Users/ADMINI~1/AppData/Local/Temp/config-repo-9143470827238107 66/spring_cl
oud_in_action/config-repo/didispace.properties
s.c.a.AnnotationConfigApplicationContext : Closing
org.springframework.context.annotation.AnnotationConfigApplicationContext@7d09c9:
startup date [Fri Sep 16 21:56:43 CST 2016]; root of context hierarchy
```

config-server 通过 Git 在本地仓库暂存，可以有效防止当 Git 仓库出现故障而引起无法加载配置信息的情况。我们可以通过断开网络，再次发起 http://localhost:7001/didispace/prod/config-label-test 请求，在控制台中可输出如下内容。可以看到，config-server 提示无法从远程获取该分支内容的报错信息：Could not pull remote for config-label-test，但是它依然会为该请求返回配置内容，这些内容源于之前访问时存于 config-server 本地文件系统中的配置内容。

```
.c.s.e.MultipleJGitEnvironmentRepository : Could not pull remote for config-label-test
(current ref=Ref[refs/heads/config-label-test=
4c4f3909b0499b8518aba1f76e8a90b0dbad535d]), remote:
http://git.oschina.net/didispace/SpringCloud-Learning
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@55c747ad:
startup date [Mon Sep 19 10:51:04 CST 2016]; root of context hierarchy
o.s.c.c.s.e.NativeEnvironmentRepository : Adding property source:
file:/C:/Users/ADMINI~1/AppData/Local/Temp/config-repo-7514536618307405051/spring_c
loud_in_action/config-repo/didispace-prod.properties
o.s.c.c.s.e.NativeEnvironmentRepository : Adding property source:
file:/C:/Users/ADMINI~1/AppData/Local/Temp/config-repo-7514536618307405051/spring_c
loud_in_action/config-repo/didispace.properties
s.c.a.AnnotationConfigApplicationContext : Closing
```

```
org.springframework.context.annotation.AnnotationConfigApplicationContext@55c747ad:
startup date [Mon Sep 19 10:51:04 CST 2016]; root of context hierarchy
```

客户端配置映射

在完成了上述验证之后，确定配置服务中心已经正常运作，下面我们尝试如何在微服务应用中获取上述配置信息。

- 创建一个 Spring Boot 应用，命名为 `config-client`，并在 `pom.xml` 中引入下述依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <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>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- 创建 Spring Boot 的应用主类，具体如下：

```
@SpringBootApplication
public class Application {
```

```
    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

- 创建 bootstrap.properties 配置，来指定获取配置文件的 config-server 位置，例如：

```
spring.application.name=didispace
spring.cloud.config.profile=dev
spring.cloud.config.label=master
spring.cloud.config.uri=http://localhost:7001/

server.port=7002
```

上述配置参数与 Git 中存储的配置文件中各个部分的对应关系如下所示。

- spring.application.name：对应配置文件规则中的{application}部分。
- spring.cloud.config.profile：对应配置文件规则中的{profile}部分。
- spring.cloud.config.label：对应配置文件规则中的{label}部分。
- spring.cloud.config.uri：配置中心 config-server 的地址。

这里需要格外注意，上面这些属性必须配置在 bootstrap.properties 中，这样 config-server 中的配置信息才能被正确加载。在第 2 章中，我们详细说明了 Spring Boot 对配置文件的加载顺序，对于本应用 jar 包之外的配置文件加载会优先于应用 jar 包内的配置内容，而通过 bootstrap.properties 对 config-server 的配置，使得该应用会从 config-server 中获取一些外部配置信息，这些信息的优先级比本地的内容要高，从而实现了外部化配置。

- 创建一个 RESTful 接口来返回配置中心的 from 属性，通过@Value("${from}")绑定配置服务中配置的 from 属性，具体实现如下：

```
@RefreshScope
@RestController
public class TestController {

    @Value("${from}")
    private String from;

    @RequestMapping("/from")
    public String from() {
        return this.from;
    }
}
```

- 除了通过@Value 注解绑定注入之外，也可以通过 Environment 对象来获取配置属性，比如：

```
@RefreshScope
@RestController
public class TestController {

    @Autowired
    private Environment env;

    @RequestMapping("/from")
    public String from() {
        return env.getProperty("from", "undefined");
    }
}
```

启动 config-client 应用，并访问 http://localhost:7002/from，我们就可以根据配置内容输出对应环境的 from 内容了。根据当前配置，我们可以获得如下返回内容：git-dev-1.0。可以继续通过修改 bootstrap.properties 中的配置内容获取不同的配置信息来熟悉配置服务中的配置规则。

服务端详解

在上一节中，我们实现了一个具备基本结构的配置管理服务端和客户端，同时讲解了其中一些配置的基本原理和规则。在本节中，我们将进一步介绍 Spring Cloud Config 服务端的一些相关知识和用法。

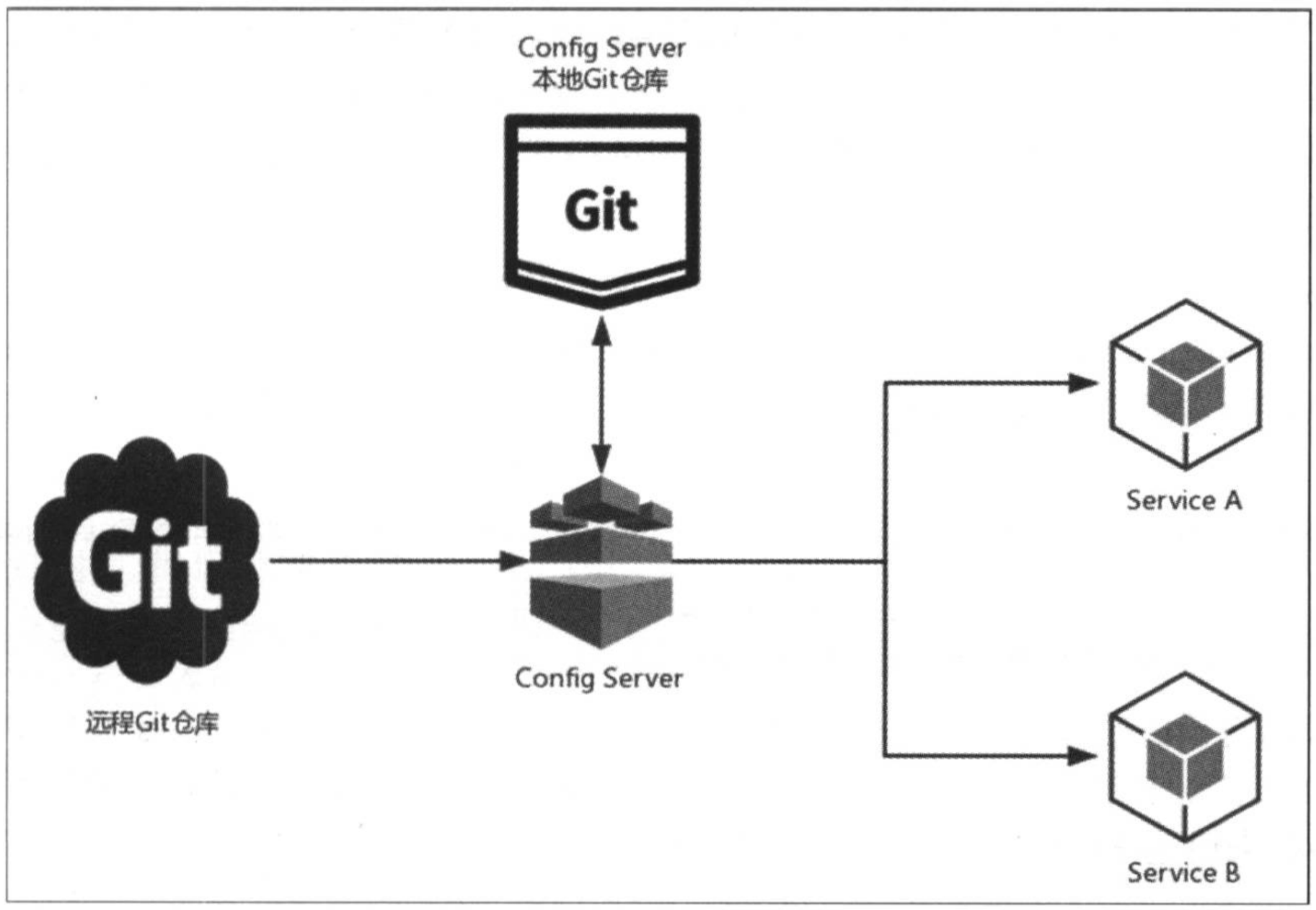
基础架构

在动手实践了上面关于 Spring Cloud Config 的基础入门内容之后，在这里我们深入理解一下它是如何运作起来的。下图所示的是上一节我们所构建案例的基本结构。

其中，主要包含下面几个要素。

- 远程 Git 仓库：用来存储配置文件的地方，上例中我们用来存储针对应用名为 didispace 的多环境配置文件：didispace-{profile}.properties。
- Config Server：这是我们上面构建的分布式配置中心，config-server 工程，在该工程中指定了所要连接的 Git 仓库位置以及账户、密码等连接信息。
- 本地 Git 仓库：在 Config Server 的文件系统中，每次客户端请求获取配置信息时，Config Server 从 Git 仓库中获取最新配置到本地，然后在本地 Git 仓库中读取并返回。当远程仓库无法获取时，直接将本地内容返回。

- Service A、Service B：具体的微服务应用，它们指定了 Config Server 的地址，从而实现从外部化获取应用自己要用的配置信息。这些应用在启动的时候，会向 Config Server 请求获取配置信息来进行加载。



客户端应用从配置管理中获取配置信息遵从下面的执行流程：

1. 应用启动时，根据 `bootstrap.properties` 中配置的应用名`{application}`、环境名`{profile}`、分支名`{label}`，向 Config Server 请求获取配置信息。
2. Config Server 根据自己维护的 Git 仓库信息和客户端传递过来的配置定位信息去查找配置信息。
3. 通过 `git clone` 命令将找到的配置信息下载到 Config Server 的文件系统中。
4. Config Server 创建 Spring 的 `ApplicationContext` 实例，并从 Git 本地仓库中加载配置文件，最后将这些配置内容读取出来返回给客户端应用。
5. 客户端应用在获得外部配置文件后加载到客户端的`ApplicationContext`实例，该配置内容的优先级高于客户端 Jar 包内部的配置内容，所以在 Jar 包中重复的内容将不再被加载。

Config Server 巧妙地通过 `git clone` 将配置信息存于本地，起到了缓存的作用，即使当 Git 服务端无法访问的时候，依然可以取 Config Server 中的缓存内容进行使用。