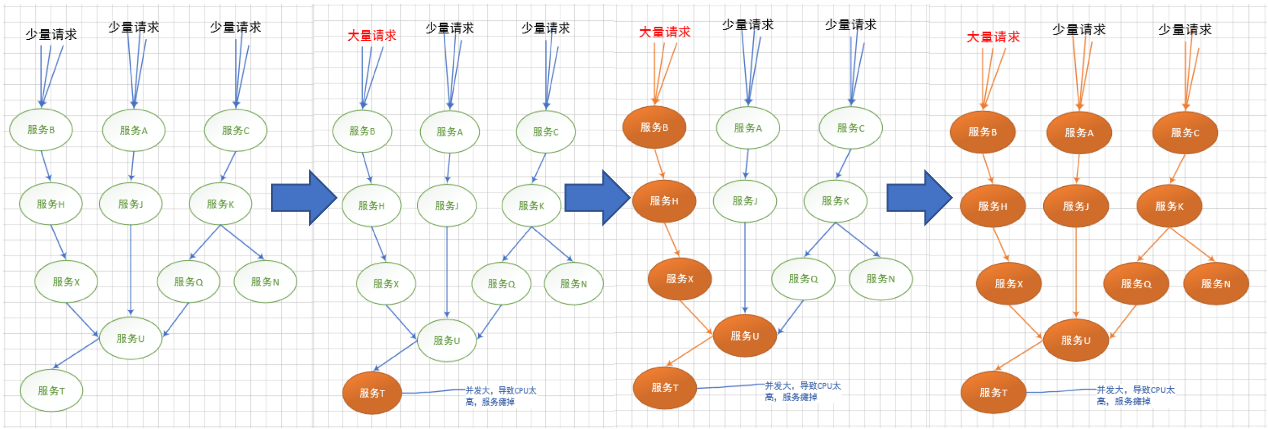
**1 什么是灾难性雪崩效应？**

　　如下图的过程所示，灾难性雪崩形成原因就大致如此：



　　造成灾难性雪崩效应的原因，可以简单归结为下述三种：

* 服务提供者不可用。如：硬件故障、程序BUG、缓存击穿、并发请求量过大等。
* 重试加大流量。如：用户重试、代码重试逻辑等。
* 服务调用者不可用。如：同步请求阻塞造成的资源耗尽等。

　　雪崩效应最终的结果就是：服务链条中的某一个服务不可用，导致一系列的服务不可用，最终造成服务逻辑崩溃。这种问题造成的后果，往往是无法预料的。

**2 如何解决灾难性雪崩效应？**

　　解决灾难性雪崩效应的方式通常有：降级、隔离、熔断、请求缓存、请求合并。

　　在Spring cloud中处理服务雪崩效应，都需要依赖hystrix组件。在pom文件中都需要引入下述依赖：

<dependency>

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

　　<artifactId>**spring-cloud-starter-hystrix**</artifactId>

</dependency>

　　通常来说，开发的时候，使用ribbon处理服务灾难雪崩效应（因此章节2示例均采用Ribbon，章节3是Feign实现方式详解），开发的成本低。维护成本高。使用feign技术处理服务灾难雪崩效应，开发的成本较高，维护成本低。

**2.1 降级**

　　降级是指，当请求超时、资源不足等情况发生时进行服务降级处理，不调用真实服务逻辑，而是使用**快速失败（fallback）**方式直接返回一个托底数据，保证服务链条的完整，避免服务雪崩。

　　解决服务雪崩效应，都是避免application client请求application service时，出现服务调用错误或网络问题。处理手法都是在application client中实现。我们需要在application client相关工程中导入hystrix依赖信息。并在对应的启动类上增加新的注解@EnableCircuitBreaker，这个注解是用于开启hystrix熔断器的，简言之，就是让代码中的hystrix相关注解生效。

　　引入依赖：

<!-- hystrix依赖， 处理服务灾难雪崩效应的。 -->

<dependency>

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

<artifactId>**spring-cloud-starter-hystrix**</artifactId>

</dependency>

　　启动器代码：

[复制代码](javascript:void(0);)

/\*\*

\* @EnableCircuitBreaker - 开启断路器。就是开启hystrix服务容错能力。

\* 当应用启用Hystrix服务容错的时候，必须增加的一个注解。

\*/

**@EnableCircuitBreaker**

**@EnableEurekaClient**

@SpringBootApplication

public class HystrixApplicationClientApplication {

public static void main(String[] args) {

　　SpringApplication.run(HystrixApplicationClientApplication.class, args);

}

}

[复制代码](javascript:void(0);)

　　在调用application service相关代码中，增加新的方法注解@HystrixCommand，代表当前方法启用Hystrix处理服务雪崩效应。

**@HystrixCommand**注解中的属性：**fallbackMethod** - 代表当调用的application service出现问题时，调用哪个fallback快速失败处理方法返回托底数据。

　　实现类：

[复制代码](javascript:void(0);)

@Service

public class HystrixService {

@Autowired

private LoadBalancerClient loadBalancerClient;

/\*\*

\* 服务降级处理。

\* 当前方法远程调用application service服务的时候，如果service服务出现了任何错误（超时，异常等）

\* 不会将异常抛到客户端，而是使用本地的一个fallback（错误返回）方法来返回一个托底数据。

\* 避免客户端看到错误页面。

\* 使用注解来描述当前方法的服务降级逻辑。

\* **@HystrixCommand - 开启Hystrix命令的注解。代表当前方法如果出现服务调用问题，使用Hystrix逻辑来处理。**

\* 重要属性 - **fallbackMethod**

\* 错误返回方法名。如果当前方法调用服务，远程服务出现问题的时候，调用本地的哪个方法得到托底数据。

\* Hystrix会调用fallbackMethod指定的方法，获取结果，并返回给客户端。

\* @return

\*/

**@HystrixCommand**(**fallbackMethod**="downgradeFallback")

public List<Map<String, Object>> testDowngrade() {

System.out.println("testDowngrade method : " + Thread.currentThread().getName());

ServiceInstance si =

this.loadBalancerClient.choose("eureka-application-service");

StringBuilder sb = new StringBuilder();

sb.append("http://").append(si.getHost())

.append(":").append(si.getPort()).append("/test");

System.out.println("request application service URL : " + sb.toString());

RestTemplate rt = new RestTemplate();

ParameterizedTypeReference<List<Map<String, Object>>> type =

new ParameterizedTypeReference<List<Map<String, Object>>>() {

};

ResponseEntity<List<Map<String, Object>>> response =

rt.exchange(sb.toString(), HttpMethod.GET, null, type);

List<Map<String, Object>> result = response.getBody();

return result;

}

/\*\*

\* **fallback方法**。本地定义的。用来处理远程服务调用错误时，返回的基础数据。

\*/

private List<Map<String, Object>> downgradeFallback(){

List<Map<String, Object>> result = new ArrayList<>();

Map<String, Object> data = new HashMap<>();

data.put("id", -1);

data.put("name", "downgrade fallback datas");

data.put("age", 0);

result.add(data);

return result;

}

}

[复制代码](javascript:void(0);)

**2.2 缓存**

　　缓存是指请求缓存。通常意义上说，就是将同样的GET请求结果缓存起来，使用缓存机制（如redis、mongodb）提升请求响应效率。

　　使用请求缓存时，需要注意**非幂等性操作对缓存数据的影响**。

　　请求缓存是依托某一缓存服务来实现的。在案例中使用redis作为缓存服务器，那么可以使用spring-data-redis来实现redis的访问操作。需要在application client相关工程中导入下述依赖：

[复制代码](javascript:void(0);)

<!-- hystrix依赖， 处理服务灾难雪崩效应的。 -->

<dependency>

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

<artifactId>**spring-cloud-starter-hystrix**</artifactId>

</dependency>

<!-- spring-data-redis spring cloud中集成的spring-data相关启动器。 -->

<dependency>

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

<artifactId>**spring-boot-starter-data-redis**</artifactId>

</dependency>

[复制代码](javascript:void(0);)

　　在Spring Cloud应用中，启用spring对cache的支持，需要在启动类中增加注解@EnableCaching，此注解代表当前应用开启spring对cache的支持。简言之就是使spring-data-redis相关的注解生效，如：@CacheConfig、@Cacheable、@CacheEvict等。

　　启动器：

[复制代码](javascript:void(0);)

/\*\*

\* @EnableCircuitBreaker - 开启断路器。就是开启hystrix服务容错能力。

\* 当应用启用Hystrix服务容错的时候，必须增加的一个注解。

\*/

**@EnableCircuitBreaker**

/\*\*

\* @EnableCaching - 开启spring cloud对cache的支持。

\* 可以自动的使用请求缓存，访问redis等cache服务。

\*/

**@EnableCaching**

**@EnableEurekaClient**

@SpringBootApplication

public class HystrixApplicationClientApplication {

public static void main(String[] args) {

SpringApplication.run(HystrixApplicationClientApplication.class, args);

}

}

[复制代码](javascript:void(0);)

　　spring cloud会检查每个幂等性请求，如果请求完全相同（路径、参数等完全一致），则首先访问缓存redis，查看缓存数据，如果缓存中有数据，则不调用远程服务application service。如果缓存中没有数据，则调用远程服务，并将结果缓存到redis中，供后续请求使用。

　　如果请求是一个非幂等性操作，则会根据方法的注解来动态管理redis中的缓存数据，避免数据不一致。

　　注意：使用请求缓存会导致很多的隐患，如：缓存管理不当导致的数据不同步、问题排查困难等。在商业项目中，解决服务雪崩效应不推荐使用请求缓存。

　　实现类：

[复制代码](javascript:void(0);)

/\*\*

\* 在类上，增加**@CacheConfig注解**，用来描述当前类型可能使用cache缓存。

\* 如果使用缓存，则缓存数据的key的前缀是cacheNames。

\* **cacheNames**是用来定义一个**缓存集的前缀命名**的，相当于分组。

\*/

**@CacheConfig(cacheNames={"test.hystrix.cache"})**

@Service

public class HystrixService {

@Autowired

private LoadBalancerClient loadBalancerClient;

/\*\*

\* 请求缓存处理方法。

\* 使用**注解@Cacheable描述方法**。配合启动器中的相关注解，**实现一个请求缓存逻辑**。

\* 将当期方法的返回值缓存到cache中。

\* 属性 value | cacheNames - 代表缓存到cache的数据的key的一部分。

\* **可以使用springEL来获取方法参数数据，定制特性化的缓存key**。

**\* 只要方法增加了@Cacheable注解，每次调用当前方法的时候，spring cloud都会先访问cache获取数据，**

**\* 如果cache中没有数据，则访问远程服务获取数据。远程服务返回数据，先保存在cache中，再返回给客户端。**

**\* 如果cache中有数据，则直接返回cache中的数据，不会访问远程服务。**

\*

\* 请求缓存会有缓存数据不一致的可能。

\* 缓存数据过期、失效、脏数据等情况。

\* 一旦使用了请求缓存来处理幂等性请求操作。**则在非幂等性请求操作中必须管理缓存。避免缓存数据的错误。**

\* @return

\*/

**@Cacheable("testCache4Get")**

public List<Map<String, Object>> testCache4Get() {

System.out.println("testCache4Get method thread name : " + Thread.currentThread().getName());

ServiceInstance si =

this.loadBalancerClient.choose("eureka-application-service");

StringBuilder sb = new StringBuilder();

sb.append("http://").append(si.getHost())

.append(":").append(si.getPort()).append("/test");

System.out.println("request application service URL : " + sb.toString());

RestTemplate rt = new RestTemplate();

ParameterizedTypeReference<List<Map<String, Object>>> type =

new ParameterizedTypeReference<List<Map<String, Object>>>() {

};

ResponseEntity<List<Map<String, Object>>> response =

rt.exchange(sb.toString(), HttpMethod.GET, null, type);

List<Map<String, Object>> result = response.getBody();

return result;

}

/\*\*

\* **非幂等性操作。**用于模拟删除逻辑。

\* 一旦非幂等性操作执行，则必须管理缓存。就是释放缓存中的数据。删除缓存数据。

\* 使用注解@CacheEvict管理缓存。

\* 通过数据cacheNames | value来删除对应key的缓存。

\* **删除缓存的逻辑，是在当前方法执行结束后。**

\* @return

\*/

**@CacheEvict("testCache4Get")**

public List<Map<String, Object>> testCache4Del() {

ServiceInstance si =

this.loadBalancerClient.choose("eureka-application-service");

StringBuilder sb = new StringBuilder();

sb.append("http://").append(si.getHost())

.append(":").append(si.getPort()).append("/test");

System.out.println("request application service URL : " + sb.toString());

RestTemplate rt = new RestTemplate();

ParameterizedTypeReference<List<Map<String, Object>>> type =

new ParameterizedTypeReference<List<Map<String, Object>>>() {

};

ResponseEntity<List<Map<String, Object>>> response =

rt.exchange(sb.toString(), HttpMethod.GET, null, type);

List<Map<String, Object>> result = response.getBody();

return result;

}

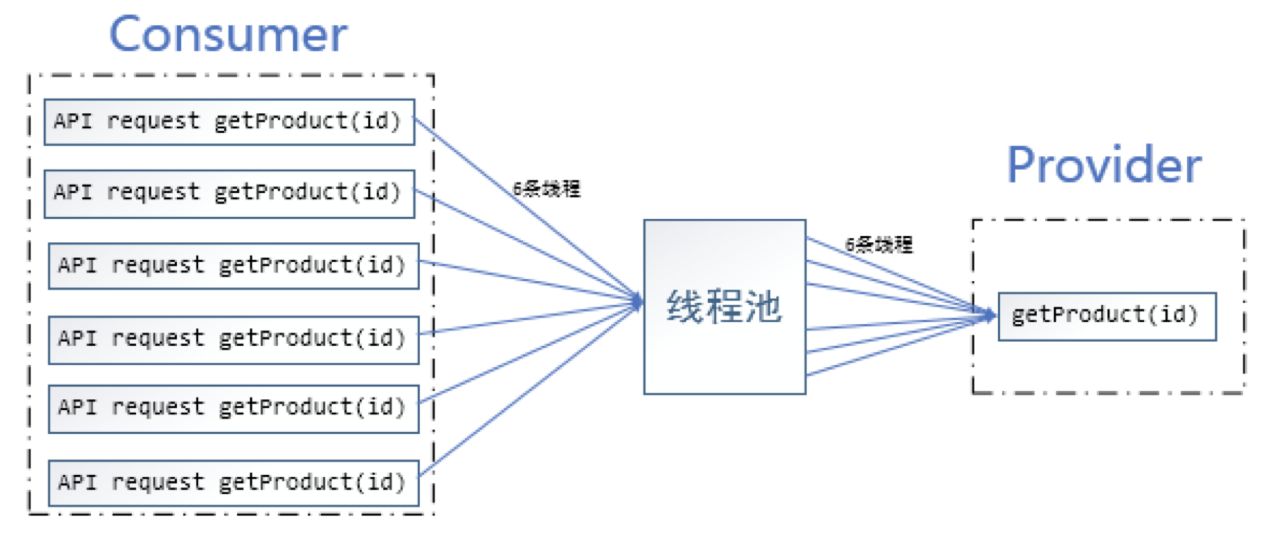
}

[复制代码](javascript:void(0);)

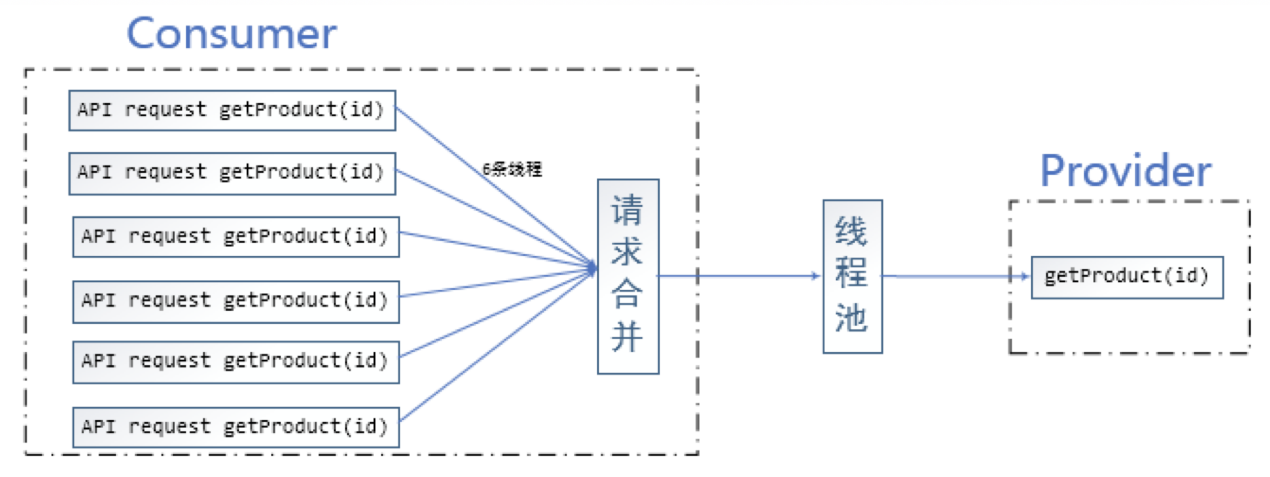
**2.3 请求合并**

　　请求合并是指，在一定时间内，收集一定量的同类型请求，合并请求需求后，一次性访问服务提供者，得到批量结果。这种方式可以减少服务消费者和服务提供者之间的通讯次数，提升应用执行效率。

　　未使用请求合并：



　　使用请求合并：



　　什么情况下使用请求合并：

　　在微服务架构中，我们将一个项目拆分成很多个独立的模块，这些独立的模块通过远程调用来互相配合工作，但是，在高并发情况下，通信次数的增加会导致总的通信时间增加，同时，线程池的资源也是有限的，高并发环境会导致有大量的线程处于等待状态，进而导致响应延迟，为了解决这些问题，我们需要来了解Hystrix的请求合并。

**通常来说，服务链条超出4个，不推荐使用请求合并。因为请求合并有等待时间。**

　　请求合并的缺点：

　　设置请求合并之后，本来一个请求可能5ms就搞定了，但是现在必须再等10ms看看还有没有其他的请求一起的，这样一个请求的耗时就从5ms增加到15ms了，不过，如果我们要**发起的命令本身就是一个高延迟的命令**，那么这个时候就可以使用请求合并了，因为这个时候时间窗的时间消耗就显得微不足道了，另外**高并发也是请求合并的一个非常重要的场景**。

　　引入依赖：

<!-- hystrix依赖， 处理服务灾难雪崩效应的。 -->

<dependency>

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

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

</dependency>

　　启动器：

[复制代码](javascript:void(0);)

/\*\*

\* @EnableCircuitBreaker - 开启断路器。就是开启hystrix服务容错能力。

\* 当应用启用Hystrix服务容错的时候，必须增加的一个注解。

\*/

**@EnableCircuitBreaker**

**@EnableEurekaClient**

@SpringBootApplication

public class HystrixApplicationClientApplication {

public static void main(String[] args) {

SpringApplication.run(HystrixApplicationClientApplication.class, args);

}

}

[复制代码](javascript:void(0);)

　　使用注解@HystrixCollapser来描述需要合并请求的方法，并提供合并方法使用注解@HystrixCommand来描述。当合并条件（@HystrixCollapser）满足时，会触发合并方法（@HystrixCommand）来调用远程服务并得到结果。

**@HystrixCollapser**注解介绍：此注解描述的方法，**返回值类型必须是java.util.concurrent.Future类型的**。代表方法为**异步方法**。

　　@HystrixCollapser注解的属性：

**batchMethod** - 请求合并方法名。

**scope** - 请求合并方式。可选值有REQUEST和GLOBAL。REQUEST代表在一个request请求生命周期内的多次远程服务调用请求需要合并处理，此为默认值。GLOBAL代表所有request线程内的多次远程服务调用请求需要合并处理。

**timerDelayInMilliseconds** - 多少时间间隔内的请求进行合并处理，默认值为10ms。建议设置时间间隔短一些，如果单位时间并发量不大，并没有请求合并的必要。

**maxRequestsInBatch** - 设置合并请求的最大极值，也就是timerDelayInMilliseconds时间内，最多合并多少个请求。默认值是Integer.MAX\_VALUE。

　　实现类：

[复制代码](javascript:void(0);)

@Service

public class HystrixService {

@Autowired

private **LoadBalancerClient** loadBalancerClient;

/\*\*

\* 需要合并请求的方法。

\* 这种方法的**返回结果一定是Future类型的**。

\* 这种方法的**处理逻辑都是异步的**。

\* 是application client在一定时间内收集客户端请求，或收集一定量的客户端请求，一次性发给application service。

\* **application service返回的结果，application client会进行二次处理，封装为future对象并返回**

\* **future对象需要通过get方法获取最终的结果**。 **get方法是由控制器调用的。所以控制器调用service的过程是一个异步处理的过程**。

\* **合并请求的方法需要使用@HystrixCollapser注解描述**。

\* **batchMethod** - 合并请求后，使用的方法是什么。**如果当前方法有参数，合并请求后的方法参数是当前方法参数的集合,如 int id >> int[] ids**。

\* **scope** - 合并请求的请求作用域。可选值有global和request。

\* **global**代表**所有的请求线程**都可以等待可合并。 **常用，所有浏览器或者请求源（Postman、curl等）调用的请求**

\* **request**代表**一个请求线程**中的多次远程服务调用可合并

\* **collapserProperties** - 细致配置。就是配置合并请求的特性。如等待多久，如可合并请求的数量。

\* 属性的类型是**@HystrixProperty类型数组**，可配置的属性值**可以直接通过字符串或常量类定义**。

\*  **timerDelayInMilliseconds - 等待时长**

\* **maxRequestsInBatch - 可合并的请求最大数量**。

\*

**\* 方法处理逻辑不需要实现，直接返回null即可。**

**\* 合并请求一定是可合并的。也就是同类型请求。同URL的请求。**

\* @param id

\* @return

\*/

**@HystrixCollapser**(**batchMethod** = "mergeRequest",

**scope** = com.netflix.hystrix.HystrixCollapser.Scope.GLOBAL,

**collapserProperties** = {

// 请求时间间隔在20ms之内的请求会被合并为一个请求,默认为10ms

**@HystrixProperty**(**name** = "timerDelayInMilliseconds", **value** = "20"),

// 设置触发批处理执行之前，在批处理中允许的最大请求数。

@HystrixProperty(name = "maxRequestsInBatch", value = "200"),

})

public Future<Map<String, Object>> testMergeRequest(**Long id**){

return null;

}

/\*\*

\* 批量处理方法。就是合并请求后真实调用远程服务的方法。

\* 必须使用@HystrixCommand注解描述，代表当前方法是一个Hystrix管理的服务容错方法。

\* 是用于处理请求合并的方法。

\* @param ids

\* @return

\*/

**@HystrixCommand**

public List<Map<String, Object>> mergeRequest(**List<Long> ids**){

ServiceInstance si =

this.loadBalancerClient.choose("eureka-application-service");

StringBuilder sb = new StringBuilder();

sb.append("http://").append(si.getHost())

.append(":").append(si.getPort()).append("/testMerge?");

for(int i = 0; i < ids.size(); i++){

Long id = ids.get(i);

if(i != 0){

sb.append("&");

}

sb.append("ids=").append(id);

}

System.out.println("request application service URL : " + sb.toString());

RestTemplate rt = new RestTemplate();

ParameterizedTypeReference<List<Map<String, Object>>> type =

new ParameterizedTypeReference<List<Map<String, Object>>>() {

};

ResponseEntity<List<Map<String, Object>>> response =

rt.exchange(sb.toString(), HttpMethod.GET, null, type);

List<Map<String, Object>> result = response.getBody();

return result;

}

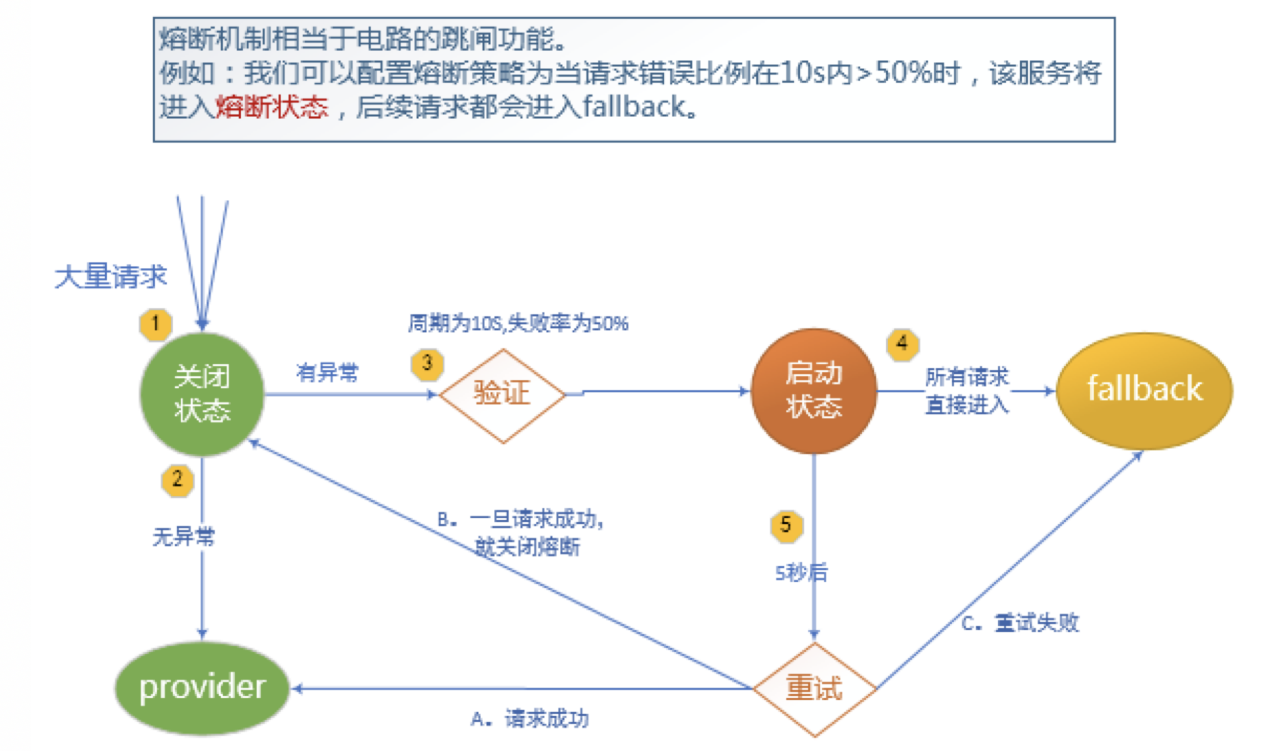
}

[复制代码](javascript:void(0);)

**2.4 熔断**

　　当一定时间内，异常请求比例（请求超时、网络故障、服务异常等）达到阀值时，启动熔断器，熔断器一旦启动，则会停止调用具体服务逻辑，通过fallback快速返回托底数据，保证服务链的完整。

　　熔断有自动恢复机制，如：当熔断器启动后，每隔5秒，尝试将新的请求发送给服务提供者，如果服务可正常执行并返回结果，则关闭熔断器，服务恢复。如果仍旧调用失败，则继续返回托底数据，熔断器持续开启状态。



　　引入依赖：

<!-- hystrix依赖， 处理服务灾难雪崩效应的。 -->

<dependency>

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

<artifactId>**spring-cloud-starter-hystrix**</artifactId>

</dependency>

　　启动器：

[复制代码](javascript:void(0);)

/\*\*

\* @EnableCircuitBreaker - 开启断路器。就是开启hystrix服务容错能力。

\* 当应用启用Hystrix服务容错的时候，必须增加的一个注解。

\*/

**@EnableCircuitBreaker**

**@EnableEurekaClient**

@SpringBootApplication

public class HystrixApplicationClientApplication {

public static void main(String[] args) {

SpringApplication.run(HystrixApplicationClientApplication.class, args);

}

}

[复制代码](javascript:void(0);)

　　熔断的实现是在调用远程服务的方法上增加@HystrixCommand注解。当注解配置满足则开启或关闭熔断器。

[复制代码](javascript:void(0);)

@Service

public class HystrixService {

@Autowired

private LoadBalancerClient loadBalancerClient;

/\*\*

\* 熔断机制

\* **相当于一个强化的服务降级。 服务降级是只要远程服务出错，立刻返回fallback结果**。

**\*** 熔断是收集一定时间内的错误比例，如果达到一定的错误率。则启动熔断，返回fallback结果。

\* **间隔一定时间会将请求再次发送给application service进行重试。如果重试成功，熔断关闭。**

\* **如果重试失败，熔断持续开启，并返回fallback数据。**

\* @HystrixCommand 描述方法。

\* fallbackMethod - fallback方法名

\* **commandProperties - 具体的熔断标准。类型是HystrixProperty数组。**

\* 可以通过字符串或常亮类配置。

\* CIRCUIT\_BREAKER\_REQUEST\_VOLUME\_THRESHOLD - 错误数量。在10毫秒内，出现多少次远程服务调用错误，则开启熔断。

\* 默认20个。 10毫秒内有20个错误请求则开启熔断。

\* CIRCUIT\_BREAKER\_ERROR\_THRESHOLD\_PERCENTAGE - 错误比例。在10毫秒内，远程服务调用错误比例达标则开启熔断。

\* CIRCUIT\_BREAKER\_SLEEP\_WINDOW\_IN\_MILLISECONDS - 熔断开启后，间隔多少毫秒重试远程服务调用。默认5000毫秒。

\* @return

\*/

**@HystrixCommand**(**fallbackMethod** = "**breakerFallback**",

**commandProperties** = {

// 默认20个;10ms内请求数大于20个时就启动熔断器，当请求符合熔断条件时将触发getFallback()。

**@HystrixProperty**(name=**HystrixPropertiesManager.CIRCUIT\_BREAKER\_REQUEST\_VOLUME\_THRESHOLD**,

value="10"),

// 请求错误率大于50%时就熔断，然后for循环发起请求，当请求符合熔断条件时将触发getFallback()。

@HystrixProperty(name=**HystrixPropertiesManager.CIRCUIT\_BREAKER\_ERROR\_THRESHOLD\_PERCENTAGE**,

value="50"),

// **默认5秒**;熔断多少秒后去尝试请求

@HystrixProperty(name=**HystrixPropertiesManager.CIRCUIT\_BREAKER\_SLEEP\_WINDOW\_IN\_MILLISECONDS**,

value="5000")}

)

public List<Map<String, Object>> testBreaker() {

System.out.println("testBreaker method thread name : " + Thread.currentThread().getName());

ServiceInstance si =

this.loadBalancerClient.choose("eureka-application-service");

StringBuilder sb = new StringBuilder();

sb.append("http://").append(si.getHost())

.append(":").append(si.getPort()).append("/test");

System.out.println("request application service URL : " + sb.toString());

RestTemplate rt = new RestTemplate();

ParameterizedTypeReference<List<Map<String, Object>>> type =

new ParameterizedTypeReference<List<Map<String, Object>>>() {

};

ResponseEntity<List<Map<String, Object>>> response =

rt.exchange(sb.toString(), HttpMethod.GET, null, type);

List<Map<String, Object>> result = response.getBody();

return result;

}

private List<Map<String, Object>> **breakerFallback**(){

System.out.println("breakerFallback method thread name : " + Thread.currentThread().getName());

List<Map<String, Object>> result = new ArrayList<>();

Map<String, Object> data = new HashMap<>();

data.put("id", -1);

data.put("name", "breaker fallback datas");

data.put("age", 0);

result.add(data);

return result;

}

}

[复制代码](javascript:void(0);)

　　注解属性描述：

[复制代码](javascript:void(0);)

CIRCUIT\_BREAKER\_ENABLED

"circuitBreaker.enabled";

# 是否开启熔断策略。默认值为true。

CIRCUIT\_BREAKER\_REQUEST\_VOLUME\_THRESHOLD

"circuitBreaker.requestVolumeThreshold";

# 10ms内，请求并发数超出则触发熔断策略。默认值为20。

CIRCUIT\_BREAKER\_SLEEP\_WINDOW\_IN\_MILLISECONDS

"circuitBreaker.sleepWindowInMilliseconds";

# 当熔断策略开启后，延迟多久尝试再次请求远程服务。默认为5秒。

CIRCUIT\_BREAKER\_ERROR\_THRESHOLD\_PERCENTAGE

"circuitBreaker.errorThresholdPercentage";

# 10ms内，出现错误的请求百分比达到限制，则触发熔断策略。默认为50%。

CIRCUIT\_BREAKER\_FORCE\_OPEN

"circuitBreaker.forceOpen";

# 是否强制开启熔断策略。即所有请求都返回fallback托底数据。默认为false。

CIRCUIT\_BREAKER\_FORCE\_CLOSED

"circuitBreaker.forceClosed";

# 是否强制关闭熔断策略。即所有请求一定调用远程服务。默认为false。

[复制代码](javascript:void(0);)

**2.5 隔离**

　　所谓隔离，就是当服务发生问题时，使用技术手段隔离请求，保证服务调用链的完整。隔离分为线程池隔离和信号量隔离两种实现方式。

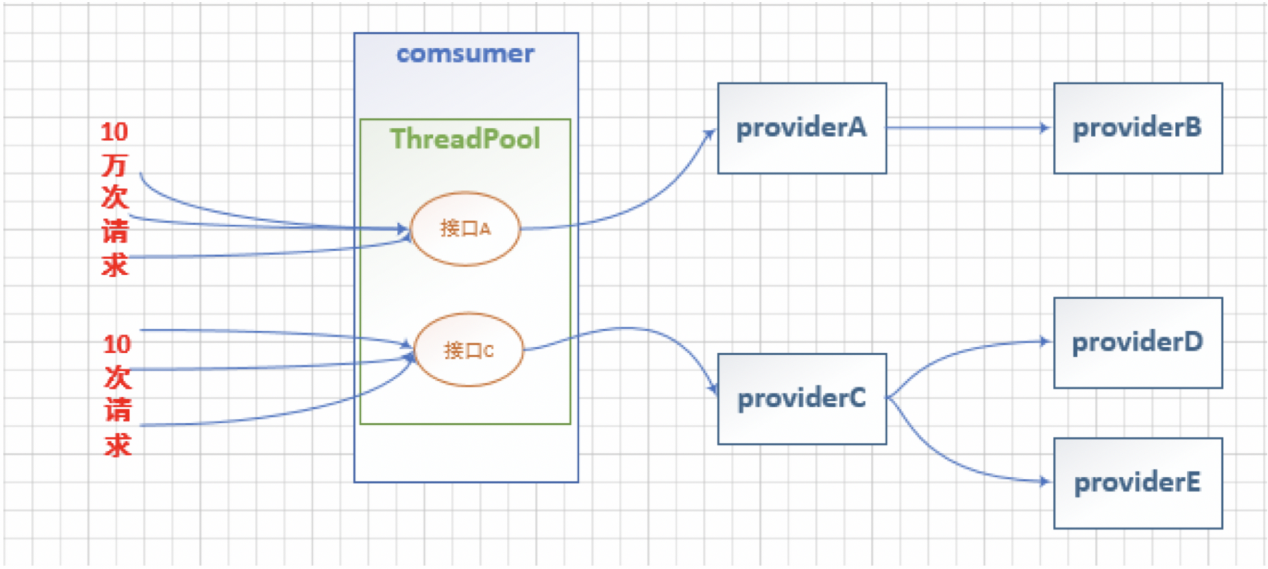
**2.5.1 线程池隔离**

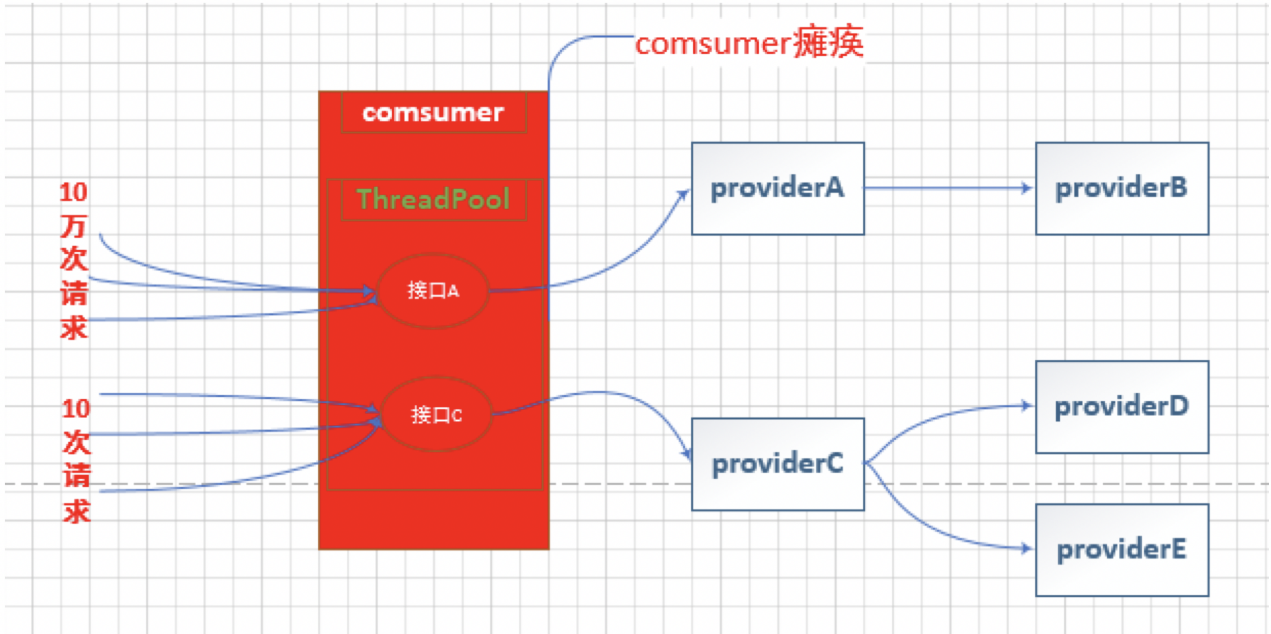
　　所谓线程池隔离，就是将并发请求量大的部分服务使用独立的线程池处理，避免因个别服务并发过高导致整体应用宕机。

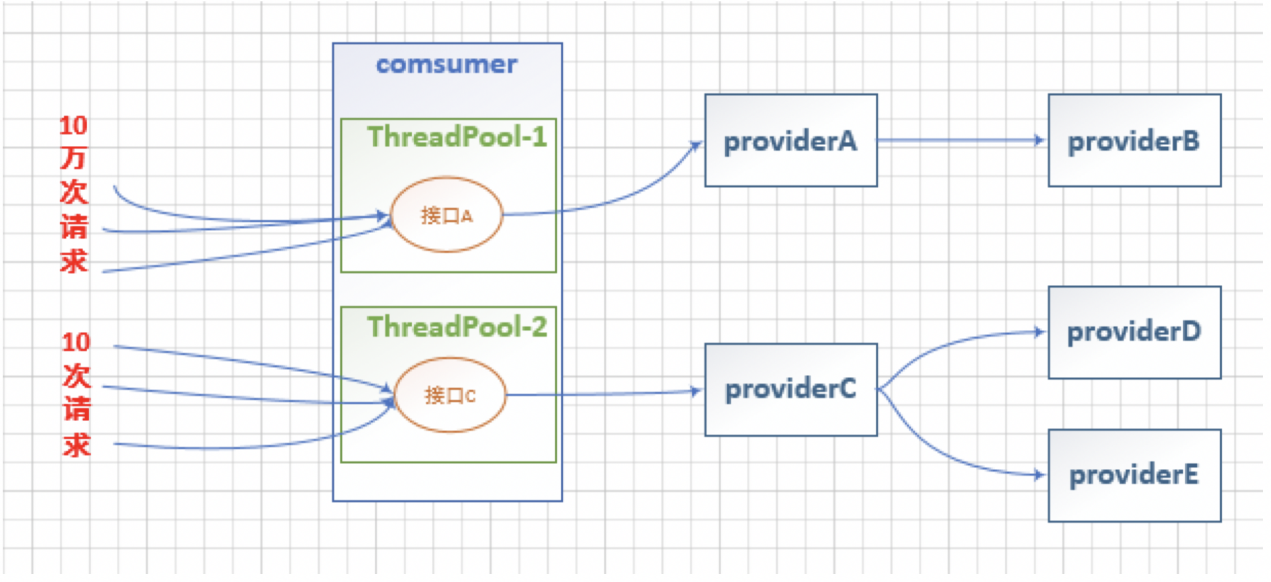
　　线程池隔离优点：

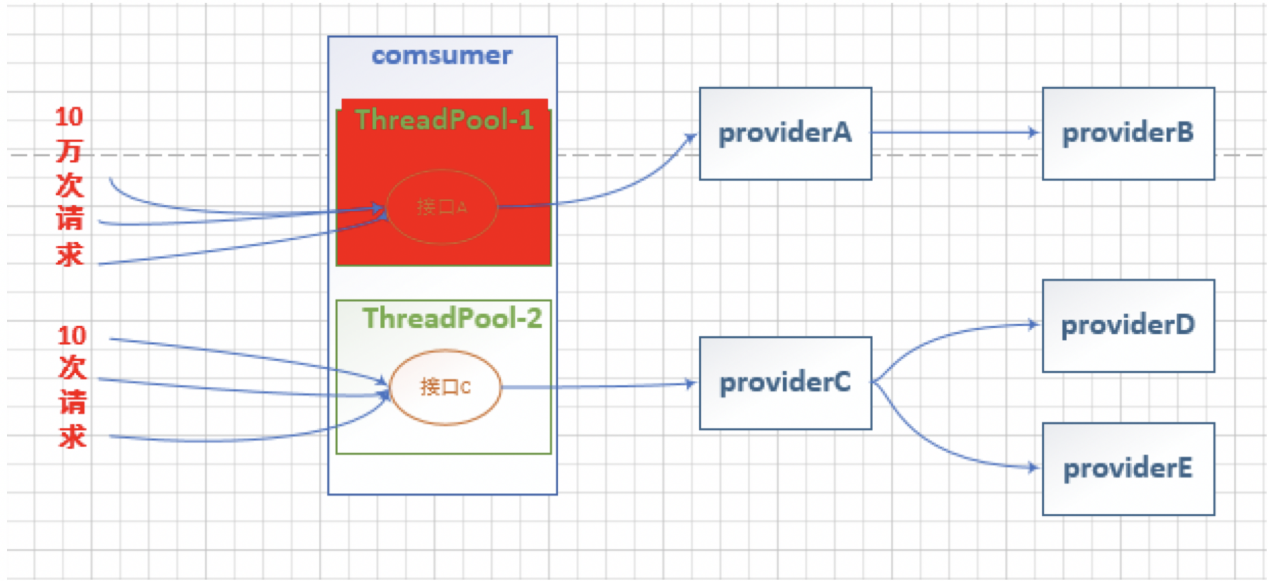
* 使用线程池隔离可以完全隔离依赖的服务，请求线程可以快速放回。
* 当线程池出现问题时，线程池是完全隔离状态的，是独立的，不会影响到其他服务的正常执行。
* 当崩溃的服务恢复时，线程池可以快速清理并恢复，不需要相对漫长的恢复等待。
* 独立的线程池也提供了并发处理能力。

　　线程池隔离缺点：  
　　线程池隔离机制，会导致服务硬件计算开销加大（CPU计算、调度等），每个命令的执行都涉及到排队、调度、上下文切换等，这些命令都是在一个单独的线程上运行的。









　　线程池隔离的实现方式同样是使用@HystrixCommand注解。相关注解配置属性如下：

* **groupKey** - 分组命名，在application client中会为每个application service服务设置一个分组，同一个分组下的服务调用使用同一个线程池。默认值为this.getClass().getSimpleName();
* **commandKey** - Hystrix中的命令命名，默认为当前方法的方法名。可省略。用于标记当前要触发的远程服务是什么。
* **threadPoolKey** - 线程池命名。要求一个应用中全局唯一。多个方法使用同一个线程池命名，代表使用同一个线程池。默认值是groupKey数据。
* **threadPoolProperties** - 用于为线程池设置的参数。其类型为HystrixProperty数组。常用线程池设置参数有：
* **coreSize** - 线程池最大并发数，建议设置标准为：requests per second at peak when healthy \* 99th percentile latency in second + some breathing room。 即每秒最大支持请求数\*（99%平均响应时间 + 一定量的缓冲时间(99%平均响应时间的10%-20%)）。如：每秒可以处理请求数为1000，99%的响应时间为60ms，自定义提供缓冲时间为60\*0.2=12ms，那么结果是 1000\*(0.060+0.012) = 72。
* **maxQueueSize** - BlockingQueue的最大长度，默认值为-1，即不限制。如果设置为正数，等待队列将从同步队列SynchronousQueue转换为阻塞队列LinkedBlockingQueue。
* **queueSizeRejectionThreshold** - 设置拒绝请求的临界值。默认值为5。此属性是配合阻塞队列使用的，也就是不适用maxQueueSize=-1（为-1的时候此值无效）的情况。是用于设置阻塞队列限制的，如果超出限制，则拒绝请求。此参数的意义就是在服务启动后，可以通过Hystrix的API调用config API动态修改，而不用用重启服务，不常用。
* **keepAliveTimeMinutes** - 线程存活时间，单位是分钟。默认值为1。
* **execution.isolation.thread.timeoutInMilliseconds** - 超时时间，默认为1000ms。当请求超时自动中断，返回fallback，避免服务长期阻塞。
* **execution.isolation.thread.interruptOnTimeout** - 是否开启超时中断。默认为TRUE。和上一个属性配合使用。

　　引入依赖：

<!-- hystrix依赖， 处理服务灾难雪崩效应的。 -->

<dependency>

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

<artifactId>**spring-cloud-starter-hystrix**</artifactId>

</dependency>

　　启动器：

[复制代码](javascript:void(0);)

/\*\*

\* @EnableCircuitBreaker - 开启断路器。就是开启hystrix服务容错能力。

\* 当应用启用Hystrix服务容错的时候，必须增加的一个注解。

\*/

**@EnableCircuitBreaker**

**@EnableEurekaClient**

@SpringBootApplication

public class HystrixApplicationClientApplication {

public static void main(String[] args) {

SpringApplication.run(HystrixApplicationClientApplication.class, args);

}

}

[复制代码](javascript:void(0);)

　　实现类：

[复制代码](javascript:void(0);)

@Service

public class HystrixService {

@Autowired

private LoadBalancerClient loadBalancerClient;

/\*\*

\* 如果使用了@HystrixCommand注解，则Hystrix自动创建独立的线程池。

\* groupKey和threadPoolKey默认值是当前服务方法所在类型的simpleName

\*

\* 所有的fallback方法，都执行在一个HystrixTimer线程池上。

\* 这个线程池是Hystrix提供的一个，专门处理fallback逻辑的线程池。

\*

\* 线程池隔离实现

**\* 线程池隔离，就是为某一些服务，独立划分线程池。让这些服务逻辑在独立的线程池中运行。**

\* **不使用tomcat提供的默认线程池。**

\* **线程池隔离也有熔断能力。如果线程池不能处理更多的请求的时候，会触发熔断，返回fallback数据。**

\* **groupKey** - 分组名称，就是为服务划分分组。如果不配置，默认使用threadPoolKey作为组名。

\* **commandKey** - 命令名称，默认值就是当前业务方法的方法名。

\* **threadPoolKey** - 线程池命名，真实线程池命名的一部分。Hystrix在创建线程池并命名的时候，会提供完整命名。默认使用gourpKey命名

\* 如果多个方法使用的threadPoolKey是同名的，则使用同一个线程池。

\* **threadPoolProperties** - 为Hystrix创建的线程池做配置。**可以使用字符串或HystrixPropertiesManager中的常量指定**。

\* 常用线程池配置：

\* **coreSize** - 核心线程数。最大并发数。1000\*（99%平均响应时间 + 适当的延迟时间）

\* **maxQueueSize** - 阻塞队列长度。如果是-1这是同步队列。如果是正数这是LinkedBlockingQueue。如果线程池最大并发数不足，

\* 提供多少的阻塞等待。

\* **keepAliveTimeMinutes** - 心跳时间，超时时长。单位是分钟。

\* **queueSizeRejectionThreshold** - 拒绝临界值，当最大并发不足的时候，超过多少个阻塞请求，后续请求拒绝。

\*/

@**HystrixCommand**(**groupKey**="test-thread-quarantine",

**commandKey** = "testThreadQuarantine",

**threadPoolKey**="test-thread-quarantine",

**threadPoolProperties** = {

@**HystrixProperty**(name="coreSize", value="30"),

@HystrixProperty(name="maxQueueSize", value="100"),

@HystrixProperty(name="keepAliveTimeMinutes", value="2"),

@HystrixProperty(name="queueSizeRejectionThreshold", value="15")

},

**fallbackMethod** = "**threadQuarantineFallback**")

public List<Map<String, Object>> testThreadQuarantine() {

System.out.println("testQuarantine method thread name : " + Thread.currentThread().getName());

ServiceInstance si =

this.loadBalancerClient.choose("eureka-application-service");

StringBuilder sb = new StringBuilder();

sb.append("http://").append(si.getHost())

.append(":").append(si.getPort()).append("/test");

System.out.println("request application service URL : " + sb.toString());

RestTemplate rt = new RestTemplate();

ParameterizedTypeReference<List<Map<String, Object>>> type =

new ParameterizedTypeReference<List<Map<String, Object>>>() {

};

ResponseEntity<List<Map<String, Object>>> response =

rt.exchange(sb.toString(), HttpMethod.GET, null, type);

List<Map<String, Object>> result = response.getBody();

return result;

}

private List<Map<String, Object>> **threadQuarantineFallback**(){

System.out.println("threadQuarantineFallback method thread name : " + Thread.currentThread().getName());

List<Map<String, Object>> result = new ArrayList<>();

Map<String, Object> data = new HashMap<>();

data.put("id", -1);

data.put("name", "thread quarantine fallback datas");

data.put("age", 0);

result.add(data);

return result;

}

}

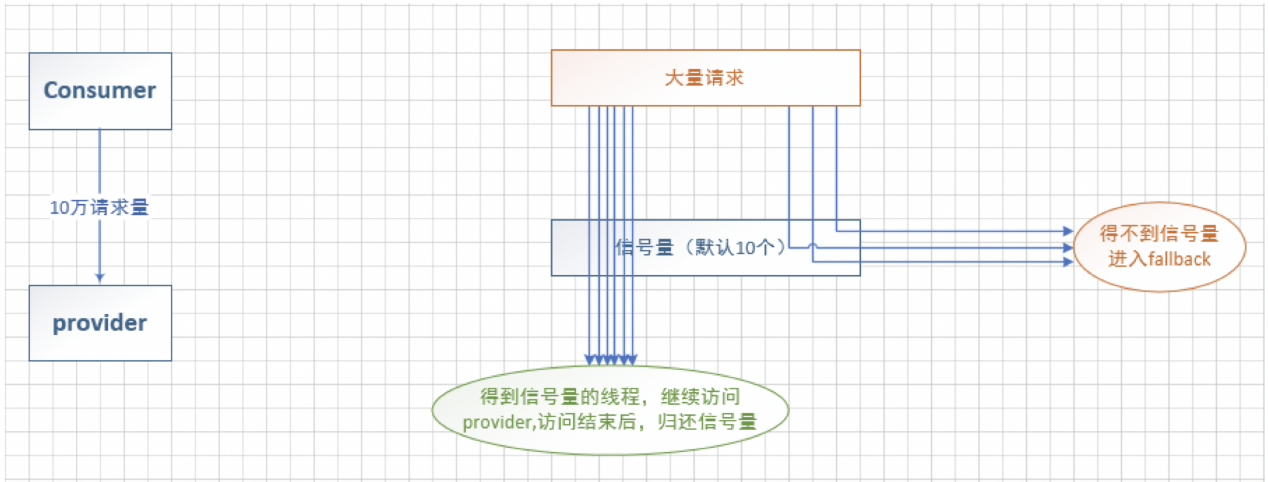
[复制代码](javascript:void(0);)

　　关于线程池：

* 对于所有请求，都交由tomcat容器的线程池处理，是一个以http-nio开头的的线程池；
* 开启了线程池隔离后，tomcat容器默认的线程池会将请求转交给threadPoolKey定义名称的线程池，处理结束后，由定义的线程池进行返回，无需还回tomcat容器默认的线程池。线程池默认为当前方法名；
* 所有的fallback都单独由Hystrix创建的一个线程池处理。

**2.5.2 信号量隔离**

　　所谓信号量隔离，就是设置一个并发处理的最大极值。当并发请求数超过极值时，通过fallback返回托底数据，保证服务完整性。



　　信号量隔离同样通过@HystrixCommand注解配置，常用注解属性有：

* commandProperty - 配置信号量隔离具体数据。属性类型为HystrixProperty数组，常用配置内容如下：
* execution.isolation.strategy - 设置隔离方式，默认为线程池隔离。可选值只有THREAD和SEMAPHORE。
* execution.isolation.semaphore.maxConcurrentRequests - 最大信号量并发数，默认为10。

　　依赖注入和启动器同线程池隔离，实现类如下：

[复制代码](javascript:void(0);)

@Service

public class HystrixService {

@Autowired

private LoadBalancerClient loadBalancerClient;

/\*\*

\* 信号量隔离实现

\* **不会使用Hystrix管理的线程池处理请求。使用容器（Tomcat）的线程处理请求逻辑。**

\* **不涉及线程切换，资源调度，上下文的转换等，相对效率高。**

\* **信号量隔离也会启动熔断机制。如果请求并发数超标，则触发熔断，返回fallback数据。**

\* **commandProperties** - 命令配置，**HystrixPropertiesManager中的常量或字符串来配置**。

\* **execution.isolation.strategy** - 隔离的种类，可选值只有**THREAD**（线程池隔离）和**SEMAPHORE**（信号量隔离）。

**\* 默认是THREAD线程池隔离。**

**\* 设置信号量隔离后，线程池相关配置失效。**

\* execution.isolation.semaphore.**maxConcurrentRequests** - 信号量最大并发数。**默认值是10。常见配置500~1000**。

\* 如果并发请求超过配置，其他请求进入fallback逻辑。

\*/

@**HystrixCommand**(**fallbackMethod**="semaphoreQuarantineFallback",

**commandProperties**={

@**HystrixProperty**(

name=HystrixPropertiesManager.**EXECUTION\_ISOLATION\_STRATEGY**,

**value="SEMAPHORE"**), // 信号量隔离

@**HystrixProperty**(

name=HystrixPropertiesManager.EXECUTION\_ISOLATION\_SEMAPHORE\_MAX\_CONCURRENT\_REQUESTS,

value="100") // 信号量最大并发数

})

public List<Map<String, Object>> testSemaphoreQuarantine() {

System.out.println("testSemaphoreQuarantine method thread name : " + Thread.currentThread().getName());

ServiceInstance si =

this.loadBalancerClient.choose("eureka-application-service");

StringBuilder sb = new StringBuilder();

sb.append("http://").append(si.getHost())

.append(":").append(si.getPort()).append("/test");

System.out.println("request application service URL : " + sb.toString());

RestTemplate rt = new RestTemplate();

ParameterizedTypeReference<List<Map<String, Object>>> type =

new ParameterizedTypeReference<List<Map<String, Object>>>() {

};

ResponseEntity<List<Map<String, Object>>> response =

rt.exchange(sb.toString(), HttpMethod.GET, null, type);

List<Map<String, Object>> result = response.getBody();

return result;

}

private List<Map<String, Object>> **semaphoreQuarantineFallback**(){

System.out.println("threadQuarantineFallback method thread name : " + Thread.currentThread().getName());

List<Map<String, Object>> result = new ArrayList<>();

Map<String, Object> data = new HashMap<>();

data.put("id", -1);

data.put("name", "thread quarantine fallback datas");

data.put("age", 0);

result.add(data);

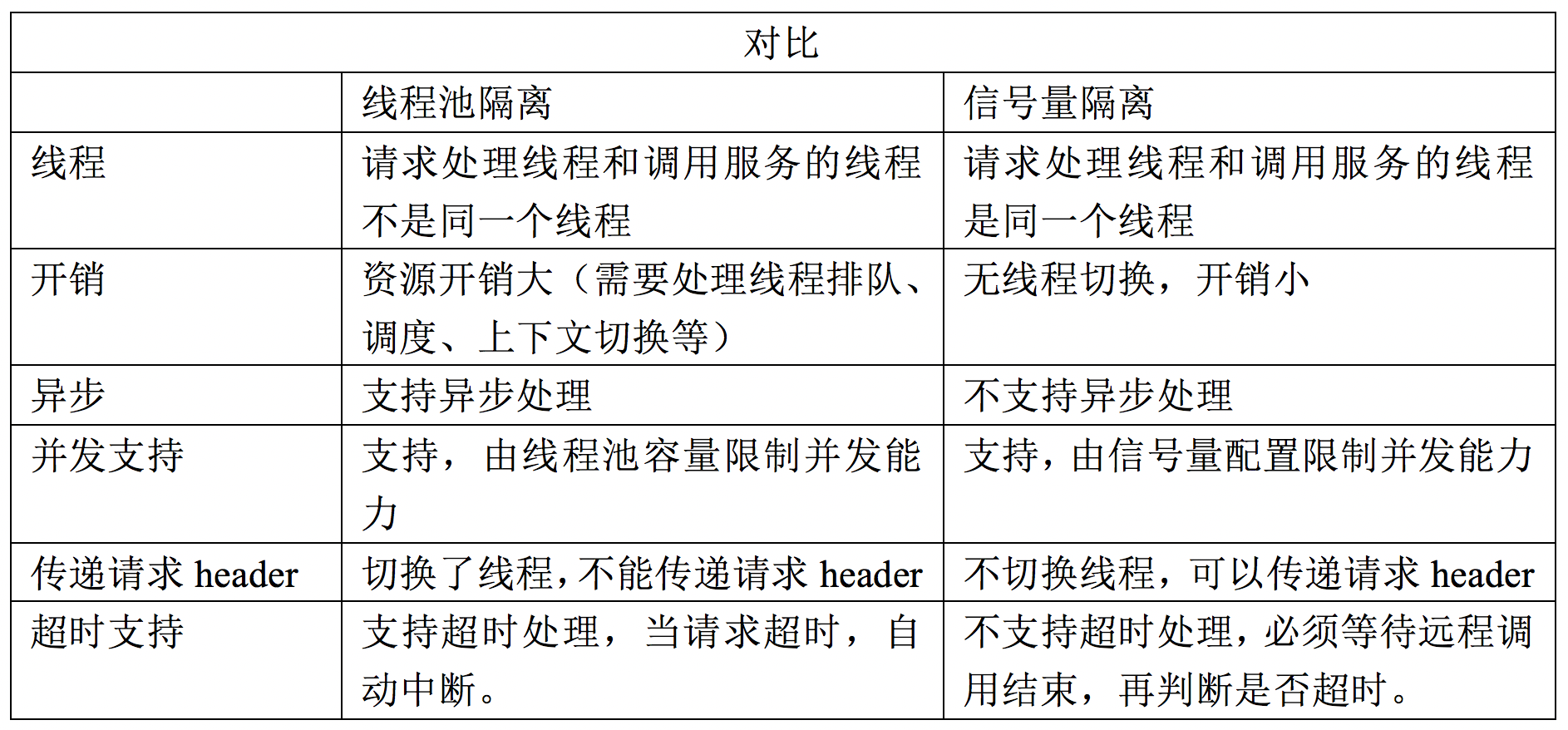
return result;

}

}

[复制代码](javascript:void(0);)

**2.5.3线程池隔离和信号量隔离的对比**



**2.5.4线程池隔离和信号量隔离的选择**

* **线程池隔离：**请求并发大，**耗时较长**（一般都是计算大，**服务链长或访问数据库**）时使用线程池隔离。可以尽可能保证外部容器（如Tomcat）线程池可用，不会因为服务调用的原因导致请求阻塞等待。
* **信号量隔离：**请求并发大，**耗时短**（**计算小，服务链段或访问缓存**）时使用信号量隔离。因为这类服务的响应快，不会占用外部容器（如Tomcat）线程池太长时间，减少线程的切换，可以避免不必要的开销，提高服务处理效率。

**3 Feign的雪崩处理**

　　在声明式远程服务调用Feign中，实现服务灾难性雪崩效应处理也是通过Hystrix实现的。而feign启动器spring-cloud-starter-feign中是包含Hystrix相关依赖的。如果只使用服务降级、熔断功能不需要做独立依赖。如果需要使用Hystrix其他服务容错能力，需要依赖spring-cloud-starter-hystrix资源。

<!-- hystrix依赖。 -->

<dependency>

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

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

</dependency>

**从Dalston版本后，feign默认关闭Hystrix支持。**所以必须在全局配置文件中开启feign技术中的Hystrix支持。配置如下：

feign.hystrix.enabled=true

　　如果不使用Hystrix服务容错功能，在application client端，服务接口只需要继承服务标准api接口即可实现远程服务调用。如果使用了Hystrix，则有不同的编写方式。具体如下。

**3.1 代码实现 - 接口实现类方式**

　　定义和服务标准api相同的application client服务接口。并通过@FeignClient注解来描述fallback方法所在类是什么。这个fallback方法所在类就是接口的实现类，实现的方法就是接口中定义方法的fallback方法。

[复制代码](javascript:void(0);)

/\*\*

\* **如果在Feign中使用Hystrix，则不能直接继承服务标准接口。**

\* 因为继承接口，一般都不会给予实现。会缺少fallback方法。熔断机制链条不完整。

**\* 在当前接口中，重复定义服务标准接口中定义的方法。**

**\* 远程服务调用的时候，是通过@FeignClient实现的。**

**\* 如果远程服务调用失败，则触发fallback注解属性定义的接口实现类中的对应方法，作为fallback方法。**

\*

\* **在默认的Hystrix配置环境中，使用的是服务降级保护机制。**

\*

\* **服务降级，默认的情况下，包含了请求超时**。

\* feign声明式远程服务调用，在启动的时候，初始化过程比较慢（通过注释@FeignClient描述接口，接口生成动态代理对象，实现服务调用）。比ribbon要慢很多。

\* 很容易在第一次访问的时候，产生超时。导致返回fallback数据。

\*/

**@FeignClient**(**name**="test-feign-application-service",

**fallback**=FirstClientFeignServiceImpl.class

)

**public interface FirstClientFeignService**{

@RequestMapping(value="/testFeign", method=RequestMethod.GET)

public List<String> testFeign();

@RequestMapping(value="/get", method=RequestMethod.GET)

public FeignTestPOJO getById(@RequestParam(value="id") Long id);

@RequestMapping(value="/get", method=RequestMethod.POST)

public FeignTestPOJO getByIdWithPOST(@RequestBody Long id);

@RequestMapping(value="/add", method=RequestMethod.GET)

public FeignTestPOJO add(@RequestParam("id") Long id, @RequestParam("name") String name);

@RequestMapping(value="/addWithGET", method=RequestMethod.GET)

public FeignTestPOJO add(@RequestBody FeignTestPOJO pojo);

@RequestMapping(value="/addWithPOST", method=RequestMethod.POST)

public FeignTestPOJO addWithPOST(@RequestBody FeignTestPOJO pojo);

}

[复制代码](javascript:void(0);)

　　为接口提供实现类，类中的方法实现就是fallback逻辑。实现类需要spring容器管理，使用@Component注解来描述类型。

[复制代码](javascript:void(0);)

/\*\*

\* **实现类中的每个方法，都是对应的接口方法的fallback。**

\* 一定要提供spring相关注解（@Component/@Service/@Repository等）。

\* 注解是为了让当前类型的对象被spring容器管理。

**\* fallback是本地方法。**

**\* 是接口的实现方法。**

\*/

@Component

**public class FirstClientFeignServiceImpl implements FirstClientFeignService** {

@Override

public List<String> testFeign() {

List<String> result = new ArrayList<>();

result.add("this is testFeign method fallback datas");

return result;

}

@Override

public FeignTestPOJO getById(Long id) {

return new FeignTestPOJO(-1L, "this is getById method fallback datas");

}

@Override

public FeignTestPOJO getByIdWithPOST(Long id) {

return new FeignTestPOJO(-1L, "this is getByIdWithPOST method fallback datas");

}

@Override

public FeignTestPOJO add(Long id, String name) {

return new FeignTestPOJO(-1L, "this is add(id, name) method fallback datas");

}

@Override

public FeignTestPOJO add(FeignTestPOJO pojo) {

return new FeignTestPOJO(-1L, "this is add(pojo) method fallback datas");

}

@Override

public FeignTestPOJO addWithPOST(FeignTestPOJO pojo) {

return new FeignTestPOJO(-1L, "this is addWithPOST method fallback datas");

}

}

[复制代码](javascript:void(0);)

**3.2 相关配置**

　　在Feign技术中，一般不使用请求合并，请求缓存等容错机制。常用的机制是隔离，降级和熔断。

**3.2.1 Properties全局配置**

[复制代码](javascript:void(0);)

**# hystrix.command.default和hystrix.threadpool.default中的default为默认CommandKey，CommandKey默认值为服务方法名。**  
**# 在properties配置中配置格式混乱，如果需要为每个方法设置不同的容错规则，建议使用yml文件配置。**

# Command Properties

**# Execution相关的属性的配置：**

# 隔离策略，默认是Thread, 可选Thread｜Semaphore

hystrix.command.default.execution.isolation.strategy=THREAD

#命令执行超时时间，默认1000ms，只在线程池隔离中有效。

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=1000

# 执行是否启用超时，默认启用true，只在线程池隔离中有效。

hystrix.command.default.execution.timeout.enabled=true

# 发生超时是是否中断，默认true，只在线程池隔离中有效。

hystrix.command.default.execution.isolation.thread.interruptOnTimeout=true

# 最大并发请求数，默认10，该参数当使用ExecutionIsolationStrategy.SEMAPHORE策略时才有效。如果达到最大并发请求数，请求会被拒绝。  
# 理论上选择semaphore的原则和选择thread一致，但选用semaphore时每次执行的单元要比较小且执行速度快（ms级别），否则的话应该用thread。  
# semaphore应该占整个容器（tomcat）的线程池的一小部分。

hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests=10

# 如果并发数达到该设置值，请求会被拒绝和抛出异常并且fallback不会被调用。默认10。  
# 只在信号量隔离策略中有效，建议设置大一些，这样并发数达到execution最大请求数时，会直接调用fallback，而并发数达到fallback最大请求数时会被拒绝和抛出异常。

hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests=10

**# ThreadPool 相关参数**

# 并发执行的最大线程数，默认10

hystrix.threadpool.default.coreSize=10

# BlockingQueue的最大队列数，当设为-1，会使用SynchronousQueue，值为正时使用LinkedBlcokingQueue。  
# 该设置只会在初始化时有效，之后不能修改threadpool的queue size，除非reinitialising thread executor。默认-1。

hystrix.threadpool.default.maxQueueSize=-1

# 即使maxQueueSize没有达到，达到queueSizeRejectionThreshold该值后，请求也会被拒绝。

hystrix.threadpool.default.queueSizeRejectionThreshold=20

# 线程存活时间，单位是分钟。默认值为1。

hystrix.threadpool.default.keepAliveTimeMinutes=1

**# Fallback相关的属性**   
# 当执行失败或者请求被拒绝，是否会尝试调用fallback方法 。默认true   
hystrix.command.default.fallback.enabled=true   
  
**# Circuit Breaker相关的属性**   
# 是否开启熔断器。默认true   
hystrix.command.default.circuitBreaker.enabled=true   
# 一个rolling window内最小的请求数。如果设为20，那么当一个rolling window的时间内（比如说1个rolling window是10毫秒）收到19个请求  
# 即使19个请求都失败，也不会触发circuit break。默认20

hystrix.command.default.circuitBreaker.requestVolumeThreshold=20

# 触发短路的时间值，当该值设为5000时，则当触发circuit break后的5000毫秒内都会拒绝远程服务调用，也就是5000毫秒后才会重试远程服务调用。默认5000

hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=5000

# 错误比率阀值，如果错误率>=该值，circuit会被打开，并短路所有请求触发fallback。默认50

hystrix.command.default.circuitBreaker.errorThresholdPercentage=50

# 强制打开熔断器

hystrix.command.default.circuitBreaker.forceOpen=false

# 强制关闭熔断器

hystrix.command.default.circuitBreaker.forceClosed=false

[复制代码](javascript:void(0);)

**3.2.2 YML全局配置**

　　YML配置文件，对SpringEL的支持更加优秀。可以通过SpringEL定制化的为每个服务调用配置Hystrix的容错处理方案。对Hystrix的配置粒度相比较Properties的配置方案更加细致。

　　在YML中可配置的Hystrix信息，和Properties中配置的内容是一致。

　　如果需要对每个服务做定制化配置，建议使用yml配置文件。在语法和格式上更容易管理和维护。

[复制代码](javascript:void(0);)

spring:

application:

name: test-feign-application-client

server:

port: 9008

feign:

hystrix:

enabled: true

hystrix:

command:

**# default代表全部服务配置，如果为某个具体服务定制配置，使用：'服务接口名#方法名(参数类型列表)'的方式来定义。**

# 如：'FirstClientFeignService#test(int)'。如果接口名称在应用中唯一，可以只写simpleName。

**# 如果接口名称在应用中不唯一，需要写fullName（包名.类名）**

"FirstClientFeignService#testFeign()":

fallback:

enabled: true

[复制代码](javascript:void(0);)

**3.3 代码实现 - Factory实现方式**

　　在服务接口的@FeignClient注解中，不再使用fallback属性，而是定义**fallbackFactory**属性。这个属性的类型是Class类型的，用于配置fallback代码所处的Factory。

　　再定义一个Java类，实现接口FallbackFactory，实现其中的**create方法。**使用**匿名内部类的方式**，为服务接口定义一个实现类，定义fallback方法实现。

　　本地接口定义：

[复制代码](javascript:void(0);)

**@FeignClient**(**name**="test-feign-application-service",

**fallbackFactory**=FirstClientFeignServiceFallbackFactory.class

)

public interface **FirstClientFeignService**{

@RequestMapping(value="/testFeign", method=RequestMethod.GET)

public List<String> testFeign();

@RequestMapping(value="/get", method=RequestMethod.GET)

public FeignTestPOJO getById(@RequestParam(value="id") Long id);

@RequestMapping(value="/get", method=RequestMethod.POST)

public FeignTestPOJO getByIdWithPOST(@RequestBody Long id);

@RequestMapping(value="/add", method=RequestMethod.GET)

public FeignTestPOJO add(@RequestParam("id") Long id, @RequestParam("name") String name);

@RequestMapping(value="/addWithGET", method=RequestMethod.GET)

public FeignTestPOJO add(@RequestBody FeignTestPOJO pojo);

@RequestMapping(value="/addWithPOST", method=RequestMethod.POST)

public FeignTestPOJO addWithPOST(@RequestBody FeignTestPOJO pojo);

}

[复制代码](javascript:void(0);)

　　FallbackFactory实现类：

[复制代码](javascript:void(0);)

/\*\*

**\* 使用Factory方式实现Feign的Hystrix容错处理。**

**\* 编写的自定义Factory必须实现接口FallbackFactory。**

**\* FallbackFactory中的方法是**

**\* 服务接口的类型 create(Throwable 远程服务调用的错误)**

\*

\* 工厂实现方案和服务接口实现类实现方案的区别：

\* **工厂可以提供自定义的异常信息处理逻辑。因为create方法负责传递远程服务调用的异常对象。**

\* **实现类可以快速的开发，但是会丢失远程服务调用的异常信息。**

\*/

@Component

public class **FirstClientFeignServiceFallbackFactory** implements **FallbackFactory**<FirstClientFeignService> {

Logger logger = LoggerFactory.getLogger(FirstClientFeignServiceFallbackFactory.class);

/\*\*

\* **create方法 - 就是工厂的生产产品的方法。**

\* 当前工厂生产的产品就是服务接口的Fallback处理对象。 就是服务接口的实现类的对象。

\*/

@Override

public FirstClientFeignService **create**(**final Throwable cause**) {

**return new FirstClientFeignService()** {

@Override

public List<String> testFeign() {

logger.warn("testFeign() - ", cause);

List<String> result = new ArrayList<>();

result.add("this is testFeign method fallback datas");

return result;

}

@Override

public FeignTestPOJO getById(Long id) {

return new FeignTestPOJO(-1L, "this is getById method fallback datas");

}

@Override

public FeignTestPOJO getByIdWithPOST(Long id) {

return new FeignTestPOJO(-1L, "this is getByIdWithPOST method fallback datas");

}

@Override

public FeignTestPOJO add(Long id, String name) {

return new FeignTestPOJO(-1L, "this is add(id, name) method fallback datas");

}

@Override

public FeignTestPOJO add(FeignTestPOJO pojo) {

return new FeignTestPOJO(-1L, "this is add(pojo) method fallback datas");

}

@Override

public FeignTestPOJO addWithPOST(FeignTestPOJO pojo) {

return new FeignTestPOJO(-1L, "this is addWithPOST method fallback datas");

}

};

}

}

[复制代码](javascript:void(0);)

　　这种实现逻辑的优势是，可以获取远程调用服务的异常信息。为后期异常处理提供参考。

　　工厂实现方案和实现类的实现方案，没有效率和逻辑上的优缺点对比。只是在远程服务调用异常的处理上有区别。

**4 Hystrix Dashboard - 数据监控**

　　Hystrix dashboard是一款针对Hystrix进行实时监控的工具，通过Hystrix Dashboard我们可以在直观地看到各Hystrix Command的请求响应时间, 请求成功率等数据。

**4.1 实现单服务单节点数据监控**

　　在使用了Hystrix技术的application client工程中增加下述依赖：

[复制代码](javascript:void(0);)

<dependency>

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

<artifactId>**spring-boot-starter-actuator**</artifactId>

</dependency>

<dependency>

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

<artifactId>**spring-cloud-starter-hystrix-dashboard**</artifactId>

</dependency>

[复制代码](javascript:void(0);)

　　在**启动器上增加注解@EnableHystrixDashboard、@EnableHystrix**。

[复制代码](javascript:void(0);)

@EnableCircuitBreaker

@EnableCaching

@EnableEurekaClient

@SpringBootApplication

**@EnableHystrixDashboard**

**@EnableHystrix**

public class HystrixDashboardApplication {

public static void main(String[] args) {

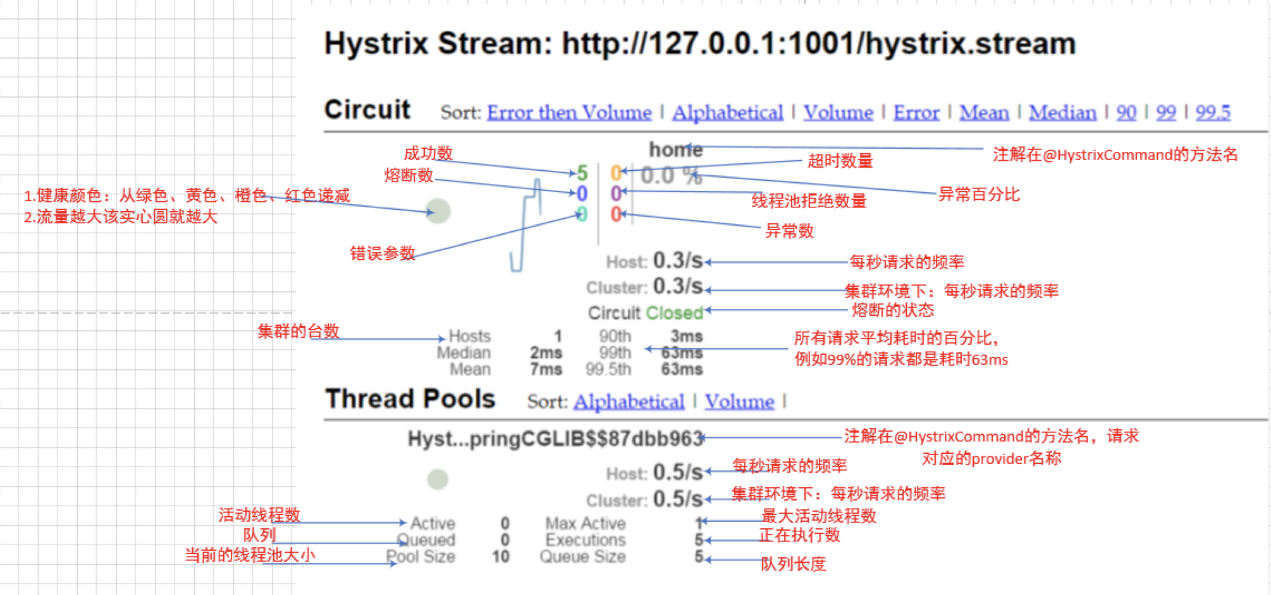
SpringApplication.run(HystrixDashboardApplication.class, args);

}

}

[复制代码](javascript:void(0);)

　　启动工程后，如果触发了Hystrix，则可以通过http://ip:port/hystrix.stream得到监控数据。这种监控数据的获取都是JSON数据。且数据量级较大。不易于查看。可以使用Hystrix Dashboard提供的视图界面来观察监控结果。视图界面访问路径为http://ip:port/hystrix。视图界面中各数据的含义如下：

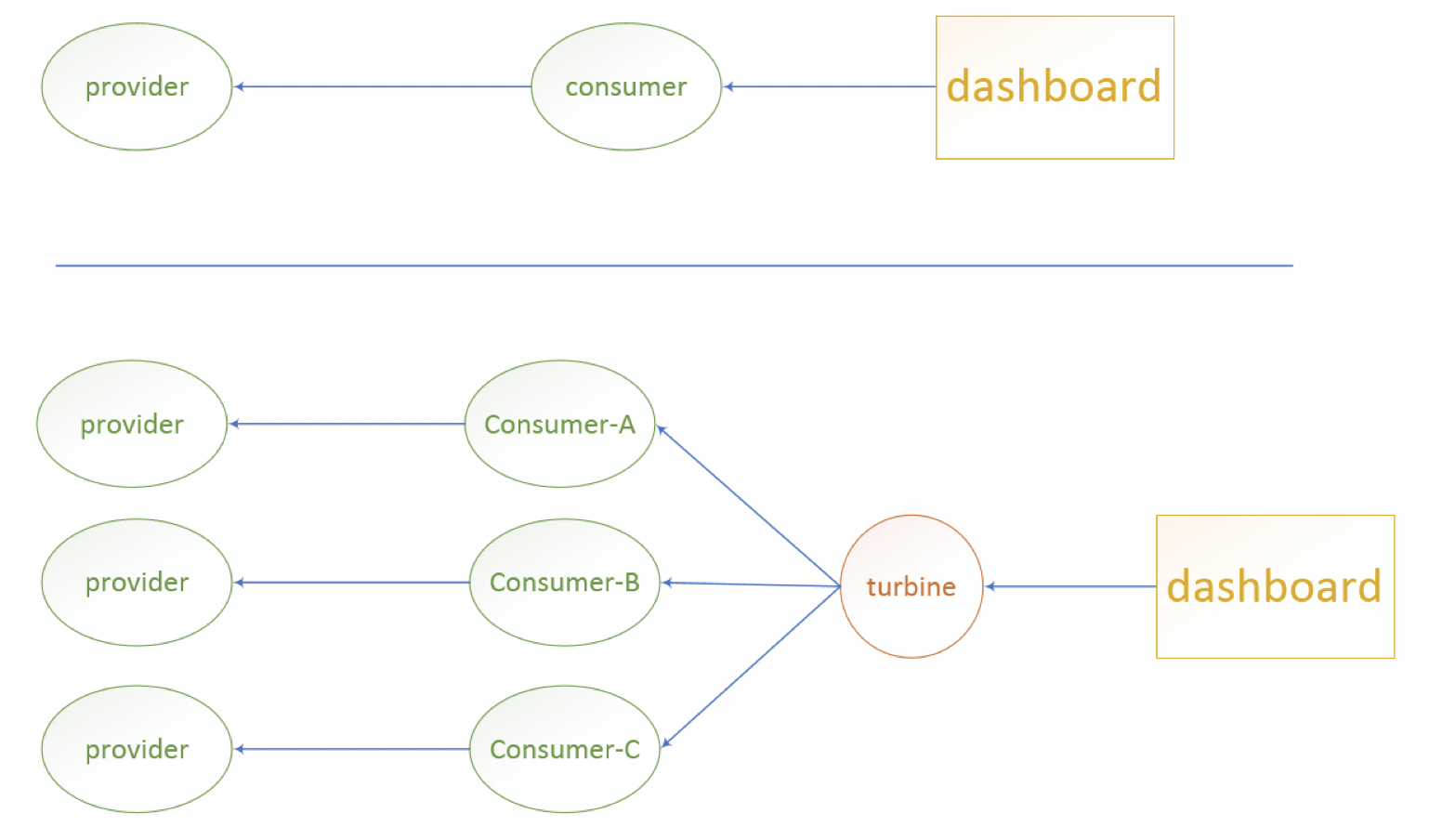


**建议：监控中心建议使用独立工程来实现。这样更便于维护。**

**4.2 使用Turbine实现多服务或集群的数据监控**

　　Turbine是聚合服务器发送事件流数据的一个工具，hystrix的监控中，只能监控单个服务或单个节点，实际生产中都为多服务集群，因此可以通过turbine来监控多集群服务。

　　Turbine在Hystrix Dashboard中的作用如下：



**4.2.1多服务监控**

　　当使用Turbine来监控多服务状态时，需提供一个独立工程来搭建Turbine服务逻辑。并在工程中增加下述依赖：

[复制代码](javascript:void(0);)

<!-- Dashboard需要的依赖信息。 -->

<dependency>

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

<artifactId>**spring-boot-starter-actuator**</artifactId>

</dependency>

<dependency>

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

<artifactId>**spring-cloud-starter-hystrix-dashboard**</artifactId>

</dependency>

<!-- Turbine需要的依赖信息。 -->

<dependency>

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

<artifactId>**spring-cloud-starter-turbine**</artifactId>

</dependency>

<dependency>

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

<artifactId>**spring-cloud-netflix-turbine**</artifactId>

</dependency>

[复制代码](javascript:void(0);)

　　并在全局配置文件中增加下述配置：

[复制代码](javascript:void(0);)

#配置Eureka中的serviceId列表，标记监控哪些服务，多个服务名用逗号分隔，可以配置监控的服务，必须开启了Hystrix Dashboard。

turbine.appConfig=hystrix-application-client,test-feign-application-client

#指定聚合哪些集群，多个使用","分割，default代表默认集群。集群就是服务名称。需要配置clusterNameExpression使用。

turbine.aggregator.clusterConfig=default

# 1. clusterNameExpression指定集群名称，默认表达式appName；此时：turbine.aggregator.clusterConfig需要配置想要监控的应用名称；

# 2. 当clusterNameExpression: default时，turbine.aggregator.clusterConfig可以不写，因为默认就是default；代表所有集群都需要监控

turbine.clusterNameExpression="default"

[复制代码](javascript:void(0);)

　　在应用启动类中，增加注解@EnableTurbine，代表开启Turbine服务，提供多服务集群监控数据收集。

[复制代码](javascript:void(0);)

/\*\*

\* @EnableTurbine - 开启Turbine功能。

\* 可以实现收集多个App client的Dashboard监控数据。

\*/

@SpringBootApplication

@EnableTurbine

public class HystrixTurbineApplication {

public static void main(String[] args) {

SpringApplication.run(HystrixTurbineApplication.class, args);

}

}

[复制代码](javascript:void(0);)

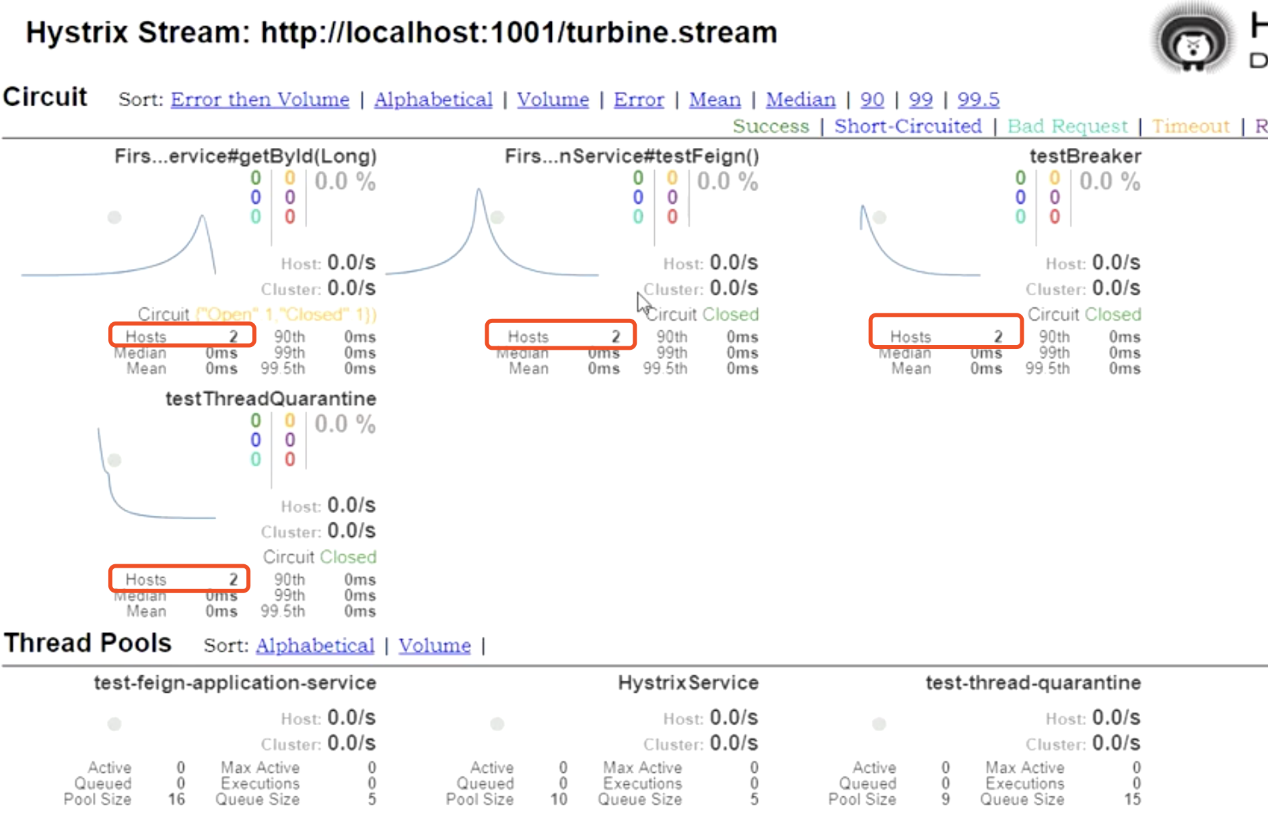
　　最后再Hystrix Dashboard视图监控服务中，使用**http://ip:port/turbine.stream**作为监控数据来源，提供可视化监控界面。

　　注意：使用Turbine做多服务监控的时候，要求全局配置文件中配置的服务列表命名在Eureka注册中心中可见。就是先启动Application client再启动Turbine。

**4.2.2服务集群监控**

　　在spring cloud中，服务名相同的多服务结点会自动形成集群，并提供服务。在Turbine中，监控服务集群不需要提供任何的特殊配置，因为turbine.appConfig已经配置了要监控的服务名称。集群监控数据会自动收集。

　　在Hystrix Dashboard的可视化监控界面中，hosts信息会显示出服务集群中的节点数量。如图所示：

**

 　　注意：使用Turbine做服务集群监控的时候，必须先启动application client集群，再启动Turbine。保证Turbine启动的时候，可以在eureka注册中心中发现要监控的服务集群。