链滴

SpringCloud 系列 --5.Hystrix

作者：289306290
原文链接：https://ld246.com/article/1584608156812
来源网站：链滴

Hystrix功能:

- 当所依赖的网络服务发生延迟或者失败时，对访问的客户端程序进行保护。
- 在分布式系统中，停止级联故障
- 网络服务恢复正常后，可以快速恢复客户端的访问能力
- 调用失败时，执行服务回退
- 支持实时监控、报警和其他操作。

先简单用示例看下Hystrix的作用

搭建一个简单的服务端接口,端口8080

```
package org.crazyit.cloud;

import javax.servlet.http.HttpServletRequest;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MyController {

    @GetMapping("/normalHello")
    @ResponseBody
    public String normalHello(HttpServletRequest request) {
        return "Hello World";
    }

    @GetMapping("/errorHello")
    @ResponseBody
    public String errorHello(HttpServletRequest request) throws Exception {
        // 模拟需要处理10秒
        Thread.sleep(10000);
        return "Error Hello World";
    }
}
```

搭建客户端来调用

客户端先继承HystrixComman类

```
package org.crazyit.cloud;

import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
```

```
import org.apache.http.util.EntityUtils;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;

/**
 * 调用一个正常的服务
 * @author 杨恩雄
 *
 */
public class HelloCommand extends HystrixCommand<String> {

    private String url;

    CloseableHttpClient httpclient;

    public HelloCommand(String url) {
        // 调用父类的构造器，设置命令组的key，默认用来作为线程池的key
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        // 创建HttpClient客户端
        this.httpclient = HttpClients.createDefault();
        this.url = url;
    }

    protected String run() throws Exception {
        try {
            // 调用 GET 方法请求服务
            HttpGet httpget = new HttpGet(url);
            // 得到服务响应
            HttpResponse response = httpclient.execute(httpget);
            // 解析并返回命令执行结果
            return EntityUtils.toString(response.getEntity());
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "";
    }

    protected String getFallback() {
        System.out.println("执行 HelloCommand 的回退方法");
        return "error";
    }

}
```

正常调用服务端

```
package org.crazyit.cloud;

public class HelloMain {

    public static void main(String[] args) {
        // 请求正常的服务
```

```
        String normalUrl = "http://localhost:8080/normalHello";
        HelloCommand command = new HelloCommand(normalUrl);
        String result = command.execute();
        System.out.println("请求正常的服务，结果：" + result);
    }
}
```

输出: 请求正常的服务，结果：Hello World

调用超时的服务

```
package org.crazyit.cloud;

public class HelloErrorMain {

    public static void main(String[] args) {
        // 请求异常的服务
        String normalUrl = "http://localhost:8080/errorHello";
        HelloCommand command = new HelloCommand(normalUrl);
        String result = command.execute();
        System.out.println("请求异常的服务，结果：" + result);
    }
}
```

输出:

执行 HelloCommand 的回退方法

请求异常的服务，结果：error

可以看到超时之后会触发HelloCommand 中 getFallback方法执行回退操作(默认是超过1秒则回退)。

Hystrix运作流程图:

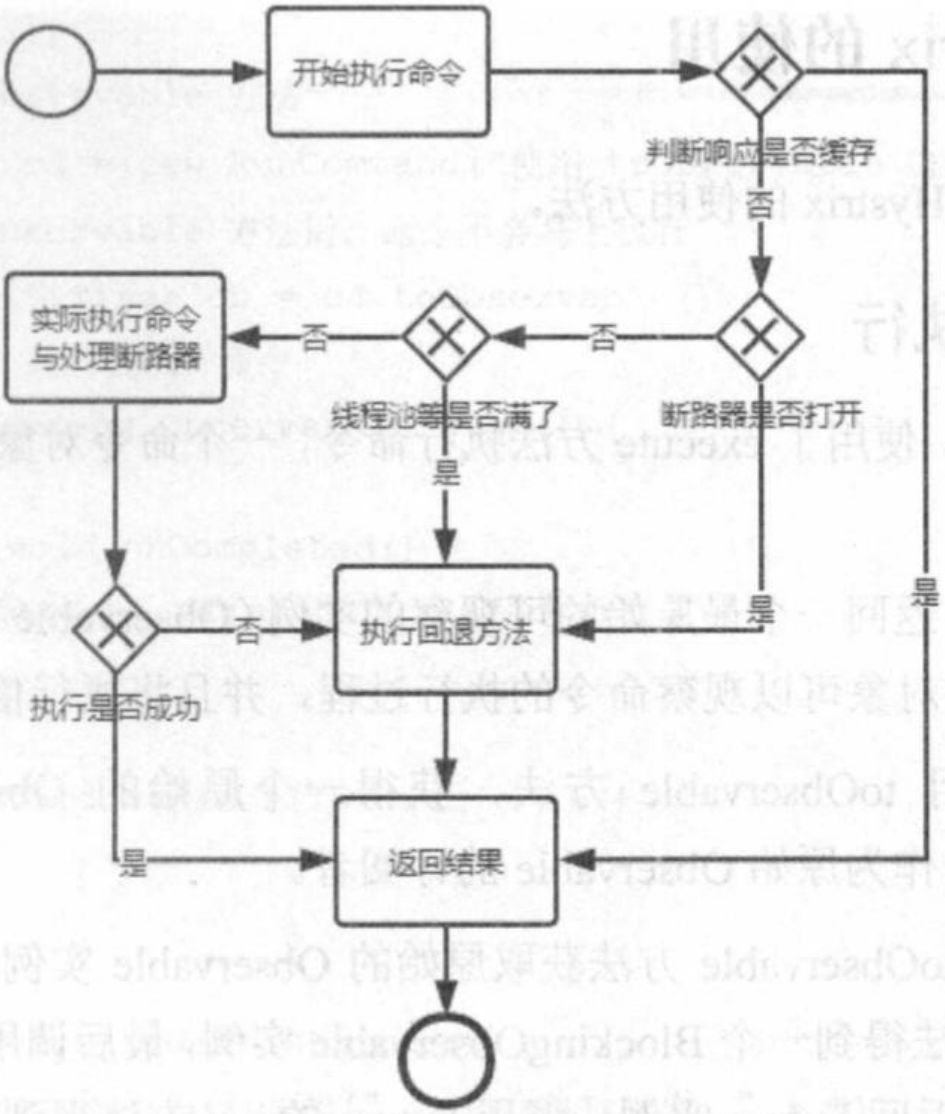

图 6-3　Hystrix 的运作流程图

上面例子中我们使用了HystrixCommand的execute方法命令，除此外还有以下方法来执行命令:

● toObservable: 返回一个最原始的课观察的实例(Observable),Observable是RxJava的类,使用该对可以观察命令的执行过程，并且将执行的信息传递给订阅者。

● observe: 调用toObservable方法,获得一个原始的Observable实例，使用ReplaySubject作为原始bservable的订阅者。

● queue： 通过toObservable方法获取原始的Observable实例,再调用Observable的toBlocking方得到一个BlockingObservable实例,最后调用BlockingObservable的toFuture方法返回Future实例调用Future的get方法得到执行结果

● execute: 调用queue的get方法返回命令的执行结果，该方法同步执行。

以上4个方法,除execute方法外,其他方法均为异步执行.observe与toOBservable方法的区别在于,toObervalbe被调用后,命令不会立即执行，只有当返回的Observable实例被订阅后,才会真正执行命令。而bserve方法的实现中,会调用toObservable得到的Observable实例,在对其进行订阅,因此调用observ法后会立即执行命令(异步).

代码参考:

```
package org.crazyit.cloud.run;

import rx.Observable;
import rx.Observer;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;

public class RunTest {

    public static void main(String[] args) throws Exception {
        // 使用execute方法
        RunCommand c1 = new RunCommand("使用execute方法执行命令");
        c1.execute();
        // 使用queue方法
        RunCommand c2 = new RunCommand("使用queue方法执行命令");
        c2.queue();
        // 使用 observe 方法
        RunCommand c3 = new RunCommand("使用 observe 方法执行命令");
        c3.observe();
        // 使用 toObservable 方法
        RunCommand c4 = new RunCommand("使用 toObservable 方法执行命令");
        // 调用 toObservable 方法后，命令不会马上执行
        Observable<String> ob = c4.toObservable();
        // 进行订阅，此时会执行命令
        ob.subscribe(new Observer<String>() {

            public void onCompleted() {
                System.out.println("    命令执行完成");
            }

            public void onError(Throwable e) {

            }

            public void onNext(String t) {
                System.out.println("    命令执行结果：" + t);
            }

        });
        Thread.sleep(100);
    }

    // 测试命令
    static class RunCommand extends HystrixCommand<String> {

        String msg;

        public RunCommand(String msg) {
            super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
            this.msg = msg;
        }
```

```
        protected String run() throws Exception {
            System.out.println(msg);
            return "success";
        }
    }
}
```

输出:

使用execute方法执行命令

使用queue方法执行命令

使用 observe 方法执行命令

使用 toObservable 方法执行命令

命令执行结果：success

命令执行完成

Hystrix可以配置一些属性可以参考 https://github.com/Netflix/ystrix/wiki/Configuration

关于回退需要注意的是,回退方法A中也可以触发另一个命令B,如果B执行失败也会触发B回退

特殊的情况:

还有其他较为复杂的例子，例如银行转账。假设一个转账命令包含调用 A 银行扣款、B 银行加款两个命令，其中一个命令失败后，执行转账命令的回退，如图 6-4 所示。

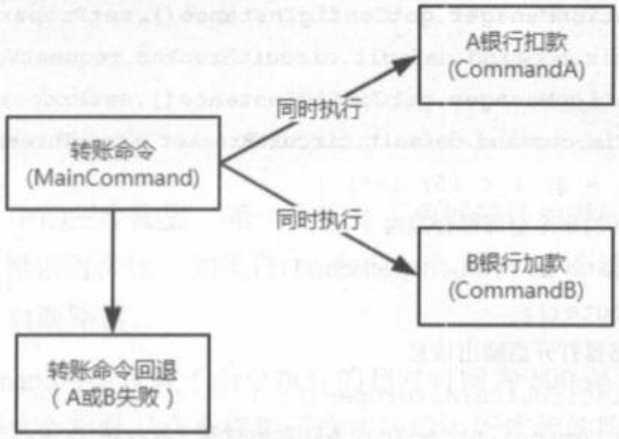

图 6-4　多命令回退

要做到图 6-4 所示的多命令只执行一次回退的效果，CommandA 与 CommandB 不能有回退方法。如果 CommandA 命令执行失败，并且该命令有回退方法，此时将不会执行 MainCommand 的回退方法。除了上面所提到的链式的回退以及多命令回退，读者还可以根据实际情况来设计回退。

断路器开启需要满足两个条件:

- 整个链路达到一定阈值,默认情况下,10s内产生超过20次请求，则符合第一个条件。
- 满足第一个条件的情况下,如果请求的错误百分比大于阈值，则会打开断路器,默认为50%

```
package org.crazyit.cloud.breaker;

import com.netflix.config.ConfigurationManager;
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.netflix.hystrix.HystrixCommandMetrics.HealthCounts;
import com.netflix.hystrix.HystrixCommandProperties;
import com.netflix.hystrix.strategy.concurrency.HystrixRequestContext;

/**
 * 断路器开启测试
 * @author 杨恩雄
 *
 */
public class OpenTest {

    public static void main(String[] args) throws Exception {
        /**
         * 配置了三项
         * 1.配置了数据的统计时间10000毫秒=10s
         * 2.配置了阈值10个请求
         * 3.配置了错误百分比50%,
         * 注: MyCommand中配置了超时时间为500毫秒,命令执行睡眠800毫秒,也就是该命令总会超时
因为500+800 = 1.3 > 1s)
         */
        // 10秒内有10个请求，则符合第一个条件
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.metrics.rollingStats.timeInMilliseconds", 10000);
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.circuitBreaker.requestVolumeThreshold", 10);
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.circuitBreaker.errorThresholdPercentage", 50);
        for(int i = 0; i < 15; i++) {
            // 执行的命令全部都会超时
            MyCommand c = new MyCommand();
            c.execute();
            // 断路器打开后输出信息
            if(c.isCircuitBreakerOpen()) {
                System.out.println("断路器被打开，执行第 " + (i + 1) + " 个命令");
            }
        }
    }

    /**
     * 模拟超时的命令
     * @author 杨恩雄
     *
     */
    static class MyCommand extends HystrixCommand<String> {
        // 设置超时的时间为500毫秒
```

```
        public MyCommand() {
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"
)
                    .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                            .withExecutionTimeoutInMilliseconds(500))
                    );
        }

        protected String run() throws Exception {
            // 模拟处理超时
            Thread.sleep(800);
            return "";
        }

        @Override
        protected String getFallback() {
            return "";
        }
    }
}
```

运行结果:

断路器被打开，执行第 11 个命令

断路器被打开，执行第 12 个命令

断路器被打开，执行第 13 个命令

断路器被打开，执行第 14 个命令

断路器被打开，执行第 15 个命令

可见前10个命令没有开启断路器,到了第11个命令，断路器被打开。

断路器关闭

断路器打开后,在一段时间内,命令不会再执行(一直触发回退),这段时间我们称作"休眠期",休眠期默认为s,休眠期结束后,Hystrix会尝试性地执行一次命令,此时断路器的状态不是开启,也不是关闭，而是一个开的状态，如果这一次命令执行成功,则会关闭断路器并清空链路的健康信息; 如果执行失败，断路器继续保持打开的状态。

```
package org.crazyit.cloud.breaker;

import com.netflix.config.ConfigurationManager;
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.netflix.hystrix.HystrixCommandMetrics.HealthCounts;
import com.netflix.hystrix.HystrixCommandProperties;

public class CloseTest {

    public static void main(String[] args) throws Exception {
        // 10秒内有3个请求就满足第一个开启断路器的条件
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.metrics.rollingStats.timeInMilliseconds", 10000);
```

```
    ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.circuitBreaker.requestVolumeThreshold", 3);
    // 请求的失败率，默认值为50%
    ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.circuitBreaker.errorThresholdPercentage", 50);
    // 设置休眠期，断路器打开后，这段时间不会再执行命令，默认值为5秒，此处设置为3秒
    ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds", 3000);
    // 该值决定是否执行超时
    boolean isTimeout = true;
    for(int i = 0; i < 10; i++) {
        // 执行的命令全部都会超时
        MyCommand c = new MyCommand(isTimeout);
        c.execute();
        // 输出健康状态等信息
        HealthCounts hc = c.getMetrics().getHealthCounts();
        System.out.println("断路器状态：" + c.isCircuitBreakerOpen() +
                ", 请求总数：" + hc.getTotalRequests());
        if(c.isCircuitBreakerOpen()) {
            // 断路器打开，让下一次循环成功执行命令
            isTimeout = false;
            System.out.println("===== 断路器打开了，等待休眠期结束 =====");
            // 休眠期会在3秒后结束，此处等待4秒，确保休眠期结束
            Thread.sleep(4000);
        }
    }
}

/**
 * 模拟超时的命令
 * @author 杨恩雄
 *
 */
static class MyCommand extends HystrixCommand<String> {

    private boolean isTimeout;

    // 设置超时的时间为500毫秒
    public MyCommand(boolean isTimeout) {
        super(
                Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))
                .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                        .withExecutionTimeoutInMilliseconds(500))
            );
        this.isTimeout = isTimeout;
    }

    protected String run() throws Exception {
        // 让外部决定是否超时
        if(isTimeout) {
            // 模拟处理超时
            Thread.sleep(800);
        } else {
            Thread.sleep(200);
```

```
        }
        return "";
    }

    @Override
    protected String getFallback() {
        return "";
    }
  }
}
```

运行结果:

断路器状态：false, 请求总数：0

断路器状态：false, 请求总数：1

断路器状态：false, 请求总数：2

断路器状态：true, 请求总数：3

===== 断路器打开了，等待休眠期结束 =====

断路器状态：false, 请求总数：0

断路器状态：false, 请求总数：0

断路器状态：false, 请求总数：0

断路器状态：false, 请求总数：3

断路器状态：false, 请求总数：3

断路器状态：false, 请求总数：5

我们再代码中配置休眠期为3s,循环10次，创建10个命令并执行，在执行完第4个命令后，断路器被打，此时我们等待休眠期结束，让下一次循环命令执行成功。第5次会执行成功，此时断路器会被关闭剩下的命令全部都可以正常执行，在循环体内使用了HealthCounts对象,用于记录链路的健康信息，果断路器关闭(链路恢复健康),HealthCounts里面的健康信息会被重置。

隔离机制

命令真正执行除了要保证断路器关闭外，还需要判断执行命令的线程池或者信号量是否满载的情况，果满载则不会执行，直接回退，这样的机制在控制命令的执行上，实现了错误的隔离，Hystrix提供两隔离策略:

- THREAD（线程）默认值,由线程池来决定命令的执行，如果线程池满载则不会执行命令.Hystrix使了ThreadPoolExecutor来控制线程池行为,线程池的默认大小为10.
- SEMAPHORE(信号量):由信号量来决定命令的执行,当请求的并发数高于阈值时，就不在执行命令，当于线程池策略，信号量策略开销更小，但是该策略不支持超时以及异步，除非对调用的服务有足够信任，否则不建议使用该策略进行隔离。

```
package org.crazyit.cloud.isolation;

import java.util.concurrent.ThreadPoolExecutor;

import com.netflix.hystrix.HystrixCircuitBreaker;
```

```
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.netflix.hystrix.HystrixCommandProperties;

public class MyCommand extends HystrixCommand<String> {

    int index;

    public MyCommand(int index) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory
                .asKey("ExampleGroup")));
        this.index = index;
    }

    protected String run() throws Exception {
        Thread.sleep(500);
        System.out.println("执行方法，当前索引：" + index);
        return "";
    }

    @Override
    protected String getFallback() {
        System.out.println("执行 fallback，当前索引：" + index);
        return "";
    }
}
```

测试线程池

```
package org.crazyit.cloud.isolation;

import java.util.concurrent.ThreadPoolExecutor;

import com.netflix.config.ConfigurationManager;
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.netflix.hystrix.HystrixCommandProperties;

/**
 * 线程隔离策略测试类
 * @author 杨恩雄
 *
 */
public class ThreadIso {

    public static void main(String[] args) throws Exception {
        // 配置线程池大小为3
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.threadpool.default.coreSize", 3);

        for(int i = 0; i < 6; i++) {
            MyCommand c = new MyCommand(i);
            c.queue();
        }
```

```
        Thread.sleep(5000);
    }
}
```

配置了线程池大小为3，进行6次循环，意味着有3次会回退。

输出:

执行 fallback，当前索引：3

执行 fallback，当前索引：4

执行 fallback，当前索引：5

执行方法，当前索引：1

执行方法，当前索引：0

执行方法，当前索引：2

测试信号量

```
package org.crazyit.cloud.isolation;

import java.util.concurrent.ThreadPoolExecutor;

import com.netflix.config.ConfigurationManager;
import com.netflix.hystrix.HystrixCircuitBreaker;
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.netflix.hystrix.HystrixCommandProperties;
import com.netflix.hystrix.HystrixCommandProperties.ExecutionIsolationStrategy;

/**
 * 信号量隔离策略测试类
 *
 * @author 杨恩雄
 *
 */
public class SemaphoreIso {

    public static void main(String[] args) throws Exception {
        // 配置使用信号量的策略进行隔离
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.execution.isolation.strategy",
                ExecutionIsolationStrategy.SEMAPHORE);
        // 设置最大并发数，默认值为10
        ConfigurationManager
                .getConfigInstance()
                .setProperty(
                        "hystrix.command.default.execution.isolation.semaphore.maxConcurrentReque
ts",
                        2);
        // 设置执行回退方法的最大并发，默认值为10
        ConfigurationManager
                .getConfigInstance()
```

```
            .setProperty(
                    "hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequest
",
                    20);
        for (int i = 0; i < 6; i++) {
            final int index = i;
            Thread t = new Thread() {
                public void run() {
                    MyCommand c = new MyCommand(index);
                    c.execute();
                }
            };
            t.start();
        }
        Thread.sleep(5000);
    }
}
```

配置了线程池最大并发数为2，循环6次，意味着有4次会回退

输出:

执行 fallback，当前索引：4

执行 fallback，当前索引：0

执行 fallback，当前索引：1

执行 fallback，当前索引：3

执行方法，当前索引：5

执行方法，当前索引：2

请求合并

对于URL相同但是参数不同的请求，Hystrix提供了合并请求的功能，减少线程开销和网络连接，提性能，有点像批处理功能。 实现合并请求功能，至少包含以下3个条件:

- 需要有一个执行请求的命令，将全部参数进行整理，然后调用外部服务。
- 需要有一个合并处理器，用于手机请求，以及处理结果。
- 外部接口提供支持，例如一个接口是根据姓名查询人员信息/person/{personName} ,另外服务端提供批量查询/persons用于查找多个Person

这种情况，如果有多个请求单个Person的请求，就可以合并为一个批量查询进行处理。

```
package org.crazyit.cloud.collapse;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
```

```
import java.util.UUID;
import java.util.concurrent.Future;

import com.netflix.config.ConfigurationManager;
import com.netflix.hystrix.HystrixCollapser;
import com.netflix.hystrix.HystrixCollapser.CollapsedRequest;
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.netflix.hystrix.strategy.concurrency.HystrixRequestContext;

public class CollapseTest {

    public static void main(String[] args) throws Exception {
        // 收集 1 秒内发生的请求，合并为一个命令执行
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.collapser.default.timerDelayInMilliseconds", 1000);
        // 请求上下文
        HystrixRequestContext context = HystrixRequestContext
                .initializeContext();
        // 创建请求合并处理器
        MyHystrixCollapser c1 = new MyHystrixCollapser("Angus");
        MyHystrixCollapser c2 = new MyHystrixCollapser("Crazyit");
        MyHystrixCollapser c3 = new MyHystrixCollapser("Sune");
        MyHystrixCollapser c4 = new MyHystrixCollapser("Paris");
        // 异步执行
        Future<Person> f1 = c1.queue();
        Future<Person> f2 = c2.queue();
        Future<Person> f3 = c3.queue();
        Future<Person> f4 = c4.queue();
        System.out.println(f1.get());
        System.out.println(f2.get());
        System.out.println(f3.get());
        System.out.println(f4.get());
        context.shutdown();
    }

    /**
     * 合并执行的命令类
     *
     * @author 杨恩雄
     *
     */
    static class CollapserCommand extends HystrixCommand<Map<String, Person>> {
        // 请求集合，第一个类型是单个请求返回的数据类型，第二是请求参数的类型
        Collection<CollapsedRequest<Person, String>> requests;

        private CollapserCommand(
                Collection<CollapsedRequest<Person, String>> requests) {
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory
                    .asKey("ExampleGroup")));
            this.requests = requests;
        }

        @Override
```

```
    protected Map<String, Person> run() throws Exception {
        System.out.println("收集参数后执行命令，参数数量：" + requests.size());
        // 处理参数
        List<String> personNames = new ArrayList<String>();
        for(CollapsedRequest<Person, String> request : requests) {
            personNames.add(request.getArgument());
        }
        // 调用服务（此处模拟调用），根据名称获取Person的Map
        Map<String, Person> result = callService(personNames);
        return result;
    }

    // 模拟服务返回
    private Map<String, Person> callService(List<String> personNames) {
        Map<String, Person> result = new HashMap<String, Person>();
        for(String personName : personNames) {
            Person p = new Person();
            p.id = UUID.randomUUID().toString();
            p.name = personName;
            p.age = new Random().nextInt(30);
            result.put(personName, p);
        }
        return result;
    }
}

static class Person {
    String id;
    String name;
    Integer age;

    public String toString() {
        // TODO Auto-generated method stub
        return "id: " + id + ", name: " + name + ", age: " + age;
    }
}

/**
 * 合并处理器
 * 第一个类型为批处理返回的结果类型
 * 第二个为单请求返回的结果类型
 * 第三个是请求参数类型
 * @author 杨恩雄
 */
static class MyHystrixCollapser extends
        HystrixCollapser<Map<String, Person>, Person, String> {

    String personName;

    public MyHystrixCollapser(String personName) {
        this.personName = personName;
    }

    @Override
```

```
        public String getRequestArgument() {
            return personName;
        }

        @Override
        protected HystrixCommand<Map<String, Person>> createCommand(
                Collection<CollapsedRequest<Person, String>> requests) {
            return new CollapserCommand(requests);
        }

        @Override
        protected void mapResponseToRequests(Map<String, Person> batchResponse,
                Collection<CollapsedRequest<Person, String>> requests) {
            // 让结果与请求进行关联
            for (CollapsedRequest<Person, String> request : requests) {
                // 获取单个响应返回的结果
                Person singleResult = batchResponse.get(request.getArgument());
                // 关联到请求中
                request.setResponse(singleResult);
            }
        }
    }

}
```

设置了"时间段",在1秒内执行的请求将会被合并到一起执行，该"时间段"的默认值为10毫秒。

执行结果:

收集参数后执行命令，参数数量：4

id: 0a41cb54-fc48-470a-89bd-964f3d4dbb03, name: Angus, age: 28

id: 9a7cfdf5-eab8-47f3-ae0b-e17948cb54e6, name: Crazyit, age: 4

id: 943de7e9-5c0e-4629-9b95-6ff5c9a469f9, name: Sune, age: 29

id: 59282c89-d0d5-4dde-9296-0478b72df8c9, name: Paris, age: 8

一般来书合并请求进行批处理，比发送多个请求快，对于URL相同、参数不同的请求，推荐使用合并求功能。

请求缓存

Hystrix支持缓存功能，一次请求过程中,多个地方调用同一个接口考虑使用缓存，缓存打开后下一次令不会执行直接从缓存中获取响应；开启缓存较为简单在命令中重写父类的getCacheKey即可。

```
package org.crazyit.cloud.cache;

import org.crazyit.cloud.HelloCommand;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.netflix.hystrix.HystrixCommandKey;
import com.netflix.hystrix.HystrixCommandProperties;
import com.netflix.hystrix.HystrixRequestCache;
```

```java
import com.netflix.hystrix.HystrixCommand.Setter;
import com.netflix.hystrix.strategy.concurrency.HystrixConcurrencyStrategy;
import com.netflix.hystrix.strategy.concurrency.HystrixConcurrencyStrategyDefault;
import com.netflix.hystrix.strategy.concurrency.HystrixRequestContext;

public class CacheMain {

    public static void main(String[] args) {
        // 初始化请求上下文
        HystrixRequestContext context = HystrixRequestContext.initializeContext();
        // 请求正常的服务
        String key = "cache-key";
        MyCommand c1 = new MyCommand(key);
        MyCommand c2 = new MyCommand(key);
        MyCommand c3 = new MyCommand(key);
        // 输出结果
        System.out.println(c1.execute() + "c1 是否读取缓存: " + c1.isResponseFromCache());
        System.out.println(c2.execute() + "c2 是否读取缓存: " + c2.isResponseFromCache());
        System.out.println(c3.execute() + "c3 是否读取缓存: " + c3.isResponseFromCache());
        // 获取缓存实例
        HystrixRequestCache cache = HystrixRequestCache.getInstance(
                HystrixCommandKey.Factory.asKey("MyCommandKey"),
                HystrixConcurrencyStrategyDefault.getInstance());
        // 清空缓存
        cache.clear(key);
        // 重新执行命令
        MyCommand c4 = new MyCommand(key);
        System.out.println(c4.execute() + "c4 是否读取缓存: " + c4.isResponseFromCache());

        context.shutdown();
    }

    static class MyCommand extends HystrixCommand<String> {

        private String key;

        public MyCommand(String key) {
            super(
                    Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))
                    .andCommandKey(HystrixCommandKey.Factory.asKey("MyCommandKey"))
                    );
            this.key = key;
        }

        protected String run() throws Exception {
            System.out.println("执行命令");
            return "";
        }

        @Override
        protected String getCacheKey() {
            return this.key;
        }
    }
```

```
}
```

输出:

执行命令

c1 是否读取缓存: false

c2 是否读取缓存: true

c3 是否读取缓存: true

执行命令

c4 是否读取缓存: false

可见c2,c3都读取了缓存。c4因为执行前清空了缓存，所有没有读取缓存。

注意：合并请求，请求缓存，必须在一次请求过程中才能实现，因此需要先初始化请求上下文:

```
// 初始化请求上下文
    HystrixRequestContext context = HystrixRequestContext.initializeContext();
xxxxxxxxxx

context.shutdown();
```