Hystrix

1.信号量模式

2.线程池模式

**1.初始化**

org.springframework.cloud.openfeign

FeignClientFactoryBean

getTarget()

loadBalance

HystrixTargeter

**使用hystrix通信初始化流程：**

1.为indfect的client创建动态代理

//FeignClientFactoryBean.java

<T> T getTarget() {

**if** (!StringUtils.*hasText*(**this**.**url**)) {  
 **if** (!**this**.**name**.startsWith(**"http"**)) {  
 **this**.**url** = **"http://"** + **this**.**name**;  
 } **else** {  
 **this**.**url** = **this**.**name**;  
 }  
 **this**.**url** += cleanPath();  
 **return** (T) loadBalance(builder, context,  
 **new** HardCodedTarget<>(**this**.**type**, **this**.**name**, **this**.**url**));  
}

Targeter targeter = get(context, Targeter.**class**);

**return** (T) targeter.target(**this**, builder, context,  
 **new** HardCodedTarget<>(**this**.**type**, **this**.**name**, url));

}

通过HystrixTargeter对象创建动态代理对象。

//HystrixTargeter.java

@Override  
**public** <T> T target(FeignClientFactoryBean factory, Feign.Builder feign,  
 FeignContext context, Target.HardCodedTarget<T> target) {  
 **return** feign.target(target);  
}

//Feign.java

**public** <T> T target(Target<T> target) {  
 **return** build().newInstance(target);  
}

创建ReflectiveFeign对象来实例化动态代理对象.

**public** <T> T newInstance(Target<T> target) {  
 **for** (Method method : target.type().getMethods()) {  
 **if** (method.getDeclaringClass() == Object.**class**) {  
 **continue**;  
 } **else if** (Util.*isDefault*(method)) {  
 DefaultMethodHandler handler = **new** DefaultMethodHandler(method);  
 defaultMethodHandlers.add(handler);  
 methodToHandler.put(method, handler);  
 } **else** {  
 methodToHandler.put(method,

nameToHandler.get(Feign.*configKey*(target.type(), method)));  
 }  
 }   
 InvocationHandler handler = **factory**.create(target, methodToHandler);  
 T proxy = (T) Proxy.*newProxyInstance*(target.type().getClassLoader(),  
 **new** Class<?>[] {target.type()}, handler);  
  
 **for** (DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {  
 defaultMethodHandler.bindTo(proxy);  
 }  
 **return** proxy;  
}

target是要代理的feign client接口类。

遍历其所有方法，保存在容器中，创建动态代理对象对应的InvocationHandler

最终通过Proxy.*newProxyInstance方法创建对应的动态代理对象。*

Feign build(**final** FallbackFactory<?> nullableFallbackFactory) {  
 **super**.invocationHandlerFactory(**new** InvocationHandlerFactory() {  
 @Override  
 **public** InvocationHandler create(Target target,  
 Map<Method, MethodHandler> dispatch) {  
 **return new** HystrixInvocationHandler(target, dispatch, **setterFactory**,  
 nullableFallbackFactory);  
 }  
 });  
 **super**.contract(**new** HystrixDelegatingContract(**contract**));  
 **return super**.build();  
}

HystrixInvocationHandler就是动态代理对象对应的InvocationHandler。

**2.通过jdk的动态代理调用:**

调用feign client注入对象就是调用其创建的动态代理对象，最终调用

HystrixInvocationHandler.invoke方法：

@Override  
**public** Object invoke(**final** Object proxy, **final** Method method, **final** Object[] args)**throws** Throwable {

HystrixCommand<Object> hystrixCommand =  
 **new** HystrixCommand<Object>(**setterMethodMap**.get(method)) {  
 @Override  
 **protected** Object run() **throws** Exception {  
 **try** {  
 **return** HystrixInvocationHandler.**this**.**dispatch**.get(method).invoke(args);  
 } **catch** (Exception e) {  
 **throw** e;  
 } **catch** (Throwable t) {  
 **throw** (Error) t;  
 }  
 }

**return** hystrixCommand.execute();

}

为每一个feign请求创建一个HystrixCommand对象，调用HystrixCommand的execute方法。

在这个地方区分是线程池模式还是信号量模式。

最终调用HystrixInvocationHandler.**this**.**dispatch**.get(method).invoke(args)方法。

通过方法名称找到对应的SynchronousMethodHandler进行处理。

//SynchronousMethodHandler.java

**public** Object invoke(Object[] argv) **throws** Throwable {

//将请求参数封装成http请求头结构  
 RequestTemplate template = **buildTemplateFromArgs**.create(argv);  
 Retryer retryer = **this**.**retryer**.clone();  
 **while** (**true**) {  
 **try** {  
 **return** executeAndDecode(template);  
 } **catch** (RetryableException e) {  
}

Object executeAndDecode(RequestTemplate template) **throws** Throwable {  
 Request request = targetRequest(template);

Response response;  
**long** start = System.*nanoTime*();  
**try** {  
 response = **client**.execute(request, **options**);  
} **catch** (IOException e) {

}

}

调用Default类的execute使用okhttp发送http请求并且接收响应。

//Client.java

@Override  
**public** Response execute(Request request, Options options) **throws** IOException {  
 HttpURLConnection connection = convertAndSend(request, options);  
 **return** convertResponse(connection, request);  
}

将响应数据解析成对象返回。

**3.负载均衡代理对象的创建：**

**protected** <T> T loadBalance(Feign.Builder builder, FeignContext context,  
 HardCodedTarget<T> target) {  
 Client client = getOptional(context, Client.**class**);  
 **if** (client != **null**) {  
 builder.client(client);  
 Targeter targeter = get(context, Targeter.**class**);  
 **return** targeter.target(**this**, builder, context, target);  
 }  
  
 **throw new** IllegalStateException(  
 **"No Feign Client for loadBalancing defined. Did you forget to include spring-cloud-starter-netflix-ribbon?"**);  
}

创建LoadBalancerFeignClient，调用HystrixTargeter的target方法和走okhttp一样的动态代理对象创建的流程。

也就是直接用okhttp和用ribbon的区别是创建的client对象不一样，一个是Default，一个是

LoadBalancerFeignClient。

4.负载均衡方法的调用：

//LoadBalancerFeignClient.java

@Override  
**public** Response execute(Request request, Request.Options options) **throws** IOException {  
 **try** {  
 URI asUri = URI.*create*(request.url());  
 String clientName = asUri.getHost();  
 URI uriWithoutHost = *cleanUrl*(request.url(), clientName);  
 FeignLoadBalancer.RibbonRequest ribbonRequest = **new** FeignLoadBalancer.RibbonRequest(  
 **this**.**delegate**, request, uriWithoutHost);  
  
 IClientConfig requestConfig = getClientConfig(options, clientName);  
 **return** lbClient(clientName)  
 .executeWithLoadBalancer(ribbonRequest, requestConfig).toResponse();  
 }  
 **catch** (ClientException e) {  
 **throw new** RuntimeException(e);  
 }  
}

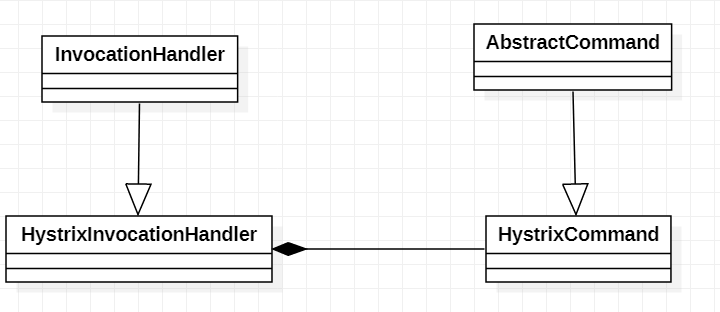
总结：

注解和动态代理结合使用。

使用动态代理完成http协议rpc调用，对接口创建动态代理对象使用InvocationHandler进行统一处理，使用注解中的参数作为http调用路径，参数作为http请求体。接口的方法的返回类型进行解析响应体。

**Hystrix细化学习：**

**HystrixInvocationHandler创建流程：**



**HystrixInvocationHandler成员：**

**成员1：Map<Method, MethodHandler> dispatch;**

MethodHandler创建流程：

通过ParseHandlersByName类的apply方法将client的方法名称创建对应的SynchronousMethodHandler。

**//**ReflectiveFeign.java

ParseHandlersByName

**public** Map<String, MethodHandler> apply(Target key) {  
 List<MethodMetadata> metadata = **contract**.parseAndValidatateMetadata(key.type());  
 Map<String, MethodHandler> result = **new** LinkedHashMap<String, MethodHandler>();  
 **for** (MethodMetadata md : metadata) {  
 result.put(md.configKey(),  
 **factory**.create(key, md, buildTemplate, **options**, **decoder**, **errorDecoder**));  
 }  
 **return** result;  
 }  
}

SynchronousMethodHandler.Factory

**public** MethodHandler create(Target<?> target,  
 MethodMetadata md,  
 RequestTemplate.Factory buildTemplateFromArgs,  
 Options options,  
 Decoder decoder,  
 ErrorDecoder errorDecoder) {  
 **return new** SynchronousMethodHandler(target, **client**, **retryer**, **requestInterceptors**, **logger**,  
 **logLevel**, md, buildTemplateFromArgs, options, decoder,  
 errorDecoder, **decode404**, **closeAfterDecode**, **propagationPolicy**);  
}

SynchronousMethodHandler:

Client **client**: 主要使用client来处理http请求。

Options **options: h**ttp请求的时候connect/read超时时间。

默认connectTimeoutMillis 10 \* 1000

默认readTimeoutMillis 60 \* 1000

**//**Feign.java

**private** Options **options** = **new** Options();

**public** Options() {  
 **this**(10 \* 1000, 60 \* 1000);  
}

之后在ReflectiveFeign遍历反射的fegin clien的Method，创建对应SynchronousMethodHandlerMap映射。

创建HystrixInvocationHandler对象的时候传递给dispatch成员。

**成员2：Map<Method, Setter> setterMethodMap;**

包含fegin接口方法对应的setter。

**HystrixCommand**

通过**setterMethodMap 中的**Setter来创建HystrixCommand

AbstractCommand：

**commandGroup：temperature-client**

**commandKey：TemperatureClient#findLatestAbnormal(Integer,Integer)**

HystrixCommandProperties **properties**

定义在工程配置文件中hystrix开头的配置属性。

//AbstractCommand.java

**private static** HystrixCommandProperties initCommandProperties(HystrixCommandKey commandKey, HystrixPropertiesStrategy propertiesStrategy, HystrixCommandProperties.Setter commandPropertiesDefaults) {  
 **if** (propertiesStrategy == **null**) {  
 **return** HystrixPropertiesFactory.*getCommandProperties*(commandKey, commandPropertiesDefaults);  
 } **else** {  
 *// used for unit testing* **return** propertiesStrategy.getCommandProperties(commandKey, commandPropertiesDefaults);  
 }  
}

使用HystrixPropertiesFactory来创建feign client接口的属性。

最终调用到HystrixPropertiesStrategy的getCommandProperties。

|  |  |  |
| --- | --- | --- |
| 属性名称 | 属性作用 | 默认值 |
| **circuitBreaker.enabled** | 是否打开熔断 | True |
| **circuitBreaker.requestVolumeThreshold** |  |  |
| **circuitBreaker.requestVolumeThreshold** |  |  |
| **circuitBreaker.sleepWindowInMilliseconds** |  |  |
| **circuitBreaker.errorThresholdPercentage** |  |  |
| **circuitBreaker.forceOpen** |  |  |
| **circuitBreaker.forceClosed** |  |  |
| **execution.isolation.strategy** |  |  |
| **execution.isolation.thread.timeoutInMilliseconds** |  |  |
| **execution.timeout.enabled** |  |  |
| **execution.isolation.thread.interruptOnTimeout** |  |  |
| **execution.isolation.thread.interruptOnFutureCancel** |  |  |
| **execution.isolation.semaphore.maxConcurrentRequests** |  |  |
| **fallback.isolation.semaphore.maxConcurrentRequests** |  |  |
| **fallback.enabled** |  |  |
| **metrics.rollingStats.timeInMilliseconds** |  |  |
| **metrics.rollingStats.numBuckets** |  |  |
| **metrics.rollingPercentile.enabled** |  |  |
| **metrics.rollingPercentile.timeInMilliseconds** |  |  |
| **metrics.rollingPercentile.numBuckets** |  |  |
| **metrics.rollingPercentile.bucketSize** |  |  |
| **metrics.healthSnapshot.intervalInMilliseconds** |  |  |
| **requestCache.enabled** |  |  |
| **threadPoolKeyOverride** |  |  |

HystrixCommandMetrics **metrics**

HystrixThreadPoolKey **threadPoolKey**

以feginClient的name作为一个key。

HystrixThreadPool **threadPool**

**//**AbstractCommand.java

**Command构造函数来创建或者获取线程池：**

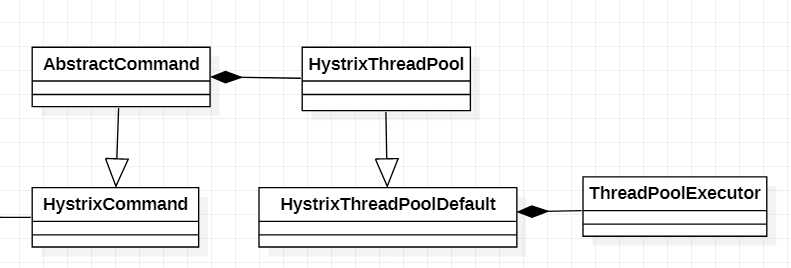
**this**.**threadPool** = *initThreadPool*(threadPool, **this**.**threadPoolKey**, threadPoolPropertiesDefaults);

//HystrixThreadPool.java

**static** HystrixThreadPool getInstance(HystrixThreadPoolKey threadPoolKey, HystrixThreadPoolProperties.Setter propertiesBuilder) {String key = threadPoolKey.name();

*//以***threadPoolKey来保存*threadPools map容器中，所以线程池隔离是以一个client***

***//类型来进行隔离的***HystrixThreadPool previouslyCached = ***threadPools***.get(key);  
 **if** (previouslyCached != **null**) {  
 **return** previouslyCached;  
 }**synchronized** (HystrixThreadPool.**class**) {  
 **if** (!***threadPools***.containsKey(key)) {  
 ***threadPools***.put(key, **new** HystrixThreadPoolDefault(threadPoolKey, propertiesBuilder));  
 }  
 }  
 **return *threadPools***.get(key);  
}

****

**//**HystrixThreadPoolDefault 构造函数。

**public** HystrixThreadPoolDefault(HystrixThreadPoolKey threadPoolKey, HystrixThreadPoolProperties.Setter propertiesDefaults) {  
 **this**.**properties** = HystrixPropertiesFactory.*getThreadPoolProperties*(threadPoolKey, propertiesDefaults);  
 HystrixConcurrencyStrategy concurrencyStrategy = HystrixPlugins.*getInstance*().getConcurrencyStrategy();  
 **this**.**queueSize** = **properties**.maxQueueSize().get();  
  
 **this**.**metrics** = HystrixThreadPoolMetrics.*getInstance*(threadPoolKey,  
 concurrencyStrategy.getThreadPool(threadPoolKey, **properties**),  
 **properties**);  
 **this**.**threadPool** = **this**.**metrics**.getThreadPool();  
 **this**.**queue** = **this**.**threadPool**.getQueue();  
HystrixMetricsPublisherFactory.*createOrRetrievePublisherForThreadPool*(threadPoolKey, **this**.**metrics**, **this**.**properties**);  
}

线程池属性获取：

HystrixThreadPoolProperties **properties**;

|  |  |  |
| --- | --- | --- |
| 属性名称 | 属性作用 | 属性默认值 |
| **allowMaximumSizeToDivergeFromCoreSize** | **True：Coresize大于maximum的时候使用coresize** | **false** |
| **coreSize** | **线程池线程基本大小** | 10 |
| **maximumSize** | **线程池最大线程数** | 10 |
| **keepAliveTimeMinutes** | **如果一个线程处在空闲状态的时间超过了该属性值，就会因为超时而退出。** | 1 |
| **maxQueueSize** |  | -1 |
| **queueSizeRejectionThreshold** |  | 5 |
| **metrics.rollingStats.timeInMilliseconds** |  | 10000 |
| **metrics.rollingStats.numBuckets** |  | 10 |

线程池的创建：

**//HystrixConcurrencyStrategy.java**

**public** ThreadPoolExecutor getThreadPool(**final** HystrixThreadPoolKey threadPoolKey, HystrixThreadPoolProperties threadPoolProperties) {  
 **final** ThreadFactory threadFactory = *getThreadFactory*(threadPoolKey);  
  
 **final boolean** allowMaximumSizeToDivergeFromCoreSize = threadPoolProperties.getAllowMaximumSizeToDivergeFromCoreSize().get();  
 **final int** dynamicCoreSize = threadPoolProperties.coreSize().get();  
 **final int** keepAliveTime = threadPoolProperties.keepAliveTimeMinutes().get();  
 **final int** maxQueueSize = threadPoolProperties.maxQueueSize().get();  
 **final** BlockingQueue<Runnable> workQueue = getBlockingQueue(maxQueueSize);  
  
 **if** (allowMaximumSizeToDivergeFromCoreSize) {  
 **final int** dynamicMaximumSize = threadPoolProperties.maximumSize().get();  
 **if** (dynamicCoreSize > dynamicMaximumSize) {  
 **return new** ThreadPoolExecutor(dynamicCoreSize, dynamicCoreSize, keepAliveTime, TimeUnit.***MINUTES***, workQueue, threadFactory);  
 } **else** {  
 **return new** ThreadPoolExecutor(dynamicCoreSize, dynamicMaximumSize, keepAliveTime, TimeUnit.***MINUTES***, workQueue, threadFactory);  
 }  
 } **else** {  
 **return new** ThreadPoolExecutor(dynamicCoreSize, dynamicCoreSize, keepAliveTime, TimeUnit.***MINUTES***, workQueue, threadFactory);  
 }  
}

线程池创建线程的方法：

**private static** ThreadFactory getThreadFactory(**final** HystrixThreadPoolKey threadPoolKey) {  
 **if** (!PlatformSpecific.*isAppEngineStandardEnvironment*()) {  
 **return new** ThreadFactory() {  
 **private final** AtomicInteger **threadNumber** = **new** AtomicInteger(0);  
  
 @Override  
 **public Thread newThread(Runnable r) {  
 Thread thread = new Thread(r, "hystrix-" + threadPoolKey.name() + "-" + threadNumber.incrementAndGet());  
 thread.setDaemon(true);  
 return thread;  
 }**  
 };  
 } **else** {  
 **return** PlatformSpecific.*getAppEngineThreadFactory*();  
 }  
}

HystrixCommand一个command对象创建完成。

**HystrixCommand的执行：**

SynchronousMethodHandler.invoke

在线程池中调用Client的execute

LoadBalancerFeignClient.execute

总结：

一个注解的fegin client相同name类型对应一个HystrixThreadPool。

一个注解的fegin client对象对应一个HystrixInvocationHandler。

一个client对象的Method对应一个MethodHandler。

一次服务之间的rpc调用对应一个HystrixCommand。

做一个http发送的通用写法，可以使用接口类和方法加上注解（请求方法/请求的路径/请求url）

加动态代理实现统一的发送动作。