1 问题背景

之前有和大家介绍，微服务的重要基础之一就是 RPC 框架。之前学习 Visual Basic 时也感觉这种架构类似一种微服务，客户端与 COM+组件的中间层服务调用，就是一种 RPC 调用模式。那么，我们采用的 SpringCloud 全家桶中的 Feign 这种 rpc 框架，也会在各个微服务之间采用 http 请求进行服务调用。

这次我遇到的问题是，采用 Feign 进行远程调用时，需要集成 Hystrix 这个容错组件对服务雪崩进行预防，出现了线程变量传递的问题。

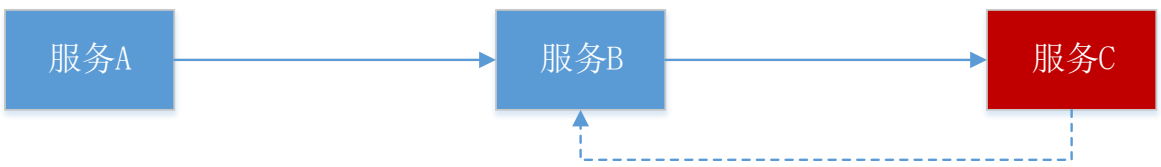
ps：本文是基于 springboot 框架设计且出现的问题，需要一定的 springboot 基础

2 概念介绍

2.1 服务雪崩

分布式背景下，很多服务都会有依赖现象，A 服务的 A1 接口需要调用 B 服务的 B1 接口，B 服务的 B1 接口依赖于 C 服务的 C1 接口。就出现如下图的情况：

如果此时，C 服务异常，大量的 B 服务请求阻塞在调用 C 这一块，B 的资源被 C 阻塞耗尽。此时 A 调用 B 也就随之阻塞，那么 A 的资源也极有可能被消耗殆尽。那么由于 C 服务的异常，引发了 B 和 A 的一系列服务的异常，我们称之为服务雪崩



2.2 hystrix 组件

既然出现了这样的问题，作为服务的间相互调用的 rpc 框架自然要针对问题进行解决。其实我们自己写代码是很好解决的，设置一个超时时间然后抛出异常好了。hystrix 组件的思路大概也是这样，当出现需要容错的情况时，它会停止对服务的访问，返回降级结果（这里与我解决的问题无关，暂不深究，有空再谈）。接下来，我们就简单介绍一下，hystrix 在我们应用场景下如何实现服务容错的。

PS：hystrix 是 netflix 旗下的一款开源框架，详细了解可以前往官网：https://github.com/Netflix/Hystrix/wiki

2.3 资源隔离

hystrix 组件会**通过资源隔离的方式防止对服务大量的并发访问**。举个例子，A 有大量的请求同时调用 C，此时会造成 A 调用 C 的线程相互阻塞甚至死锁。此时，我们需要**对 C 这个资源进行隔离访问**，就像加上一把锁，每次只允许**一定量的并发访问**。

2.3.1 hystrix 隔离策略

- 线程与线程池（Thread & Thread Pools）

当一个请求到服务 A，tomcat 服务器会为服务 A 当前的请求分配一个线程 1，当线程 1 需要调用服务 B，hystrix 为 B 资源创建了一个大小为 10 的线程池，由其中的一个线程来执行调用请求。当 B 资源的线程池被耗尽时，获取 B 资源的请求将会被降级处理，无法访问 B 资源。以下是原文，我大概重述了一下：

Clients (libraries, network calls, etc) execute on separate threads. This isolates them from the calling thread (Tomcat thread pool) so that the caller may “walk away” from a dependency call that is taking too long.

- 信号量（Semaphores）

信号量就像一个**计数器**，为每个资源分配了一个最大额，**默认是 10**。每当一个请求开始调用依赖时，就会计数信号量减一，资源释放后就加一，也就是说请求必须要拿

到信号量才能。这样也能起到资源隔离的作用。但是 hystrix 官方也说到了，这样的话，超时和退出等能力就无法由 hystrix 提供了。

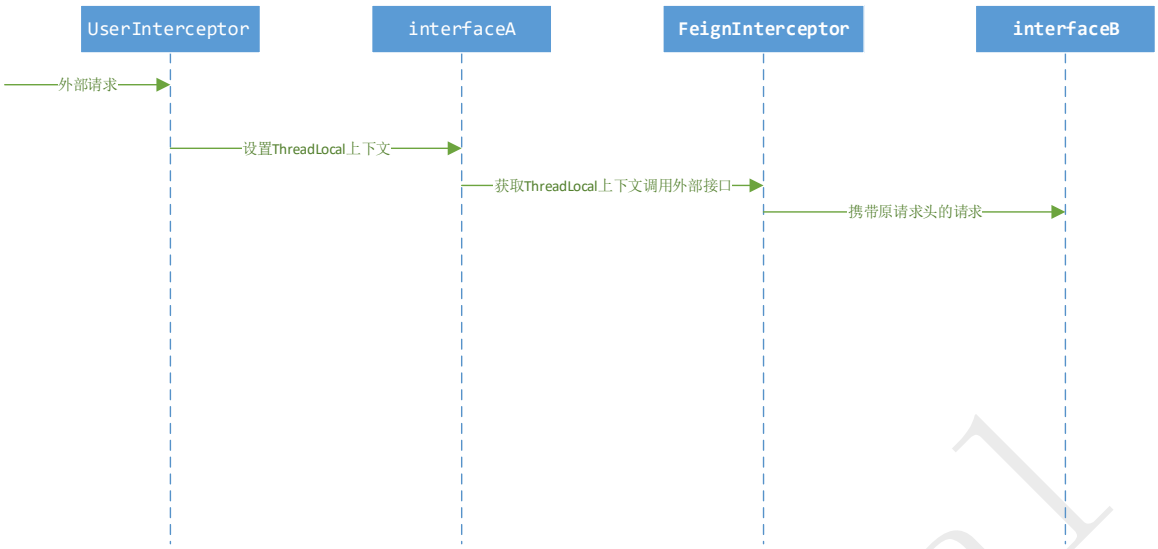
- 隔离策略总结（使用线程池隔离的原因）

hystrix 的线程池和信号量隔离策略。相比线程池，信号量不需要线程切换，因此避免了不必要的开销。**但是信号量不支持异步，也不支持超时**，所以在请求的服务不可用时，信号量超过限制会立刻返回，所以已经持有信号量的线程只能等待服务响应超时返回，即可能出现长时间等待。线程池模式下，当超过指定时间未响应的服务，Hystrix 会通过响应中断的方式通知线程立即结束并返回。其实线程开销相对于网络传输等消耗，都是在内存中执行会相对可以忽略，但是超时和异步是我们实在需要的能力。

	线程切换	异步	超时	熔断	限流	开销
信号量	否	否	否	是	是	小
线程池	是	是	是	是	是	大

3 问题介绍

我们使用 feign 集成的 hystrix 进行服务容错，并且 feign 是默认支持基于线程池的访问。请求在到服务上时，请求头中会携带用户 id 作为上下文基础信息，java 提供了 ThreadLocal 这样一个线程变量，用于在线程中存储一个全局的上下文。我绘制一张图让大家理解一下整个调用链路：



3.1 基础代码

- 请求头上下文类

```
public class CommonRequestHeader {
    /**
     * 令牌
     */
    private String authorization;

    /**
     * 用户id
     */
    private Long userId;

    public CommonRequestHeader(String authorization, Long userId) {
        this.authorization = authorization;
        this.userId = userId;
    }
}
```

- 上下文操作类

```
public class CommonRequestHeaderHandler {
    /**
     * 利用ThreadLocal存储线程级别的全局上下文内容
     */
    private static final ThreadLocal<CommonRequestHeader> context = new ThreadLocal<>();
```

```
    public static void clear() {
        context.set(null);
    }

    public static void setContext(CommonRequestHeader header) {
        context.set(header);
    }

    public static CommonRequestHeader getContext() {
        return context.get();
    }
}
```

- feign 拦截器

当 A 服务调用 B 服务时，需要将 A 中的上下文传递给 B 服务，很简单，Feign 框架提供了拦截器抽线接口，实现该接口并交给 spring 容器管理就能拦截所有使用 feign 发送出去的请求。我们只需要在拦截器将上下文内容取出放到 A 发送给 B 的请求头中即可

```
/**
 * feign 请求拦截器
 * @author Daniel
 * @date 2021/8/12 9:51
 */
@Slf4j
@Configuration
public class FeignInterceptor implements RequestInterceptor {

    @Override
    public void apply(RequestTemplate requestTemplate) {
        //将线程 ThreadLocal 中的头文件变量取出来，传递到 feign 的请求头中：这样 hystrix 也可以使用了
        log.info("当前线程名称为：{}，线程上下文为：{}",Thread.currentThread().getName(),CommonRequestHeaderHandler.getContext());
        if (!Objects.isNull(CommonRequestHeaderHandler.getContext())) {
            log.info("userId：{}", CommonRequestHeaderHandler.getContext().getUserId());

requestTemplate.header(ContextHeaderConstants.K_TOKEN,CommonRequestHeaderHandler.getContext().getAuthorization());

requestTemplate.header(ContextHeaderConstants.K_TOKEN_USER,String.valueOf(CommonRequestHeaderHandler.getContext().getUserId()));
        }
```

```
    }
}
```

3.2 ThreadLocal 数据丢失

根据上面的介绍，我们知道了，feign 调用的资源隔离模式有线程隔离和信号量隔离，默认采用线程隔离。其实大部分也会使用线程隔离。那么在线程隔离的模式下，我们调用一个 feign 请求看看拿到的上下文是什么？

测试：

```
    //测试启用hystrix，采用线程隔离模式，是否会导致线程变量无法传递的现象
    @Test
    public void testFeignHystrix() throws Exception {
        //创建异步线程，线程任务未调用feign 的线程：此时会调用hystrix 线程池中的线程
        Thread thread = new Thread(() -> {
            try {
                //模拟实际请求的上下文的userId
                Long userId = 324684371049185280L;
                //初始化
                CommonRequestHeader commonRequestHeader = new CommonRequestHeader("",
userId);
                CommonRequestHeaderHandler.setContext(commonRequestHeader);
                log.info("当前线程名称为：{}，线程上下文为：{}",
Thread.currentThread().getName(), CommonRequestHeaderHandler.getContext());
                //feign 调用，采用hystrix 线程隔离，会调用线程池中的空闲线程发送请求
                userFeign.findContext();
            } catch (Exception e) {
                log.error("调用失败");
            }
        });
        //执行异步线程
        thread.start();
        Thread.sleep(10000L);
    }
```

测试结果：

```
当前线程名称为：Thread-14，线程上下文为：CommonRequestHeader(authorization=,
userId=324684371049185280)

当前线程名称为：hystrix-service-user-1，线程上下文为：null
```

调用 feign 时，hystrix 为我们要调用的 user 服务创建了一个大小为 10 的线程池，请求是由 hystrix-service-user-1 发送出去的，显然 ThreadLocal 的线程变量并没有传递从调用方 Thread-14 传递到 hystrix 的资源线程 hystrix-service-user-1。这就是 hystrix 线程隔离导致的线程变量丢失问题。

3.2.1 InheritableThreadLocal 解决

后来呢，我发现线程变量本身其实也可以做做文章，既然存在切换线程，那必然有线程间数据传递，那么我只要找找有没有那种被传递过去的线程变量就好了，果然有这种变量 InheritableThreadLocal。 InheritableThreadLocal 本身也是继承自 ThreadLocal，重写了 3 个方法，在当前线程上创建了一个新的线程实例 Thread 的时候，会把相关线程变量从当前线程 copy 到新的线程实例。但是为什么创建新的线程会拿到父线程的变量呢，直接看看 Thread 的源码

- 新增了 inheritableThreadLocals 属性

```
/* ThreadLocal values pertaining to this thread. This map is maintained
 * by the ThreadLocal class. */
ThreadLocal.ThreadLocalMap threadLocals = null;

/*
 * InheritableThreadLocal values pertaining to this thread. This map is
 * maintained by the InheritableThreadLocal class.
 */
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

- init 源码中，获取父线程的 inheritableThreadLocals 属性

```
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =
            ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
```

也就是说，Thread 本身除了拥有 ThreadLocal 变量用来存储本地变量，还新增了一个属性用于线程之间数据传递，那我们就知道为啥能把父线程的变量传递到子线程了。既然该变量会在线程父线程创建子线程执行任务时被继承过去。

测试：

（1） 先把上下文中线程变量由 ThreadLocal 替换为 InheritableThreadLocal

```
//InheritableThreadLocal 是继承自 Thread，根据里氏替换原则，父类都可以用子类代替
private static final ThreadLocal<CommonRequestHeader> context = new
InheritableThreadLocal<>();
```

（2）方便起见，测试方法中，我将创建线程的方法抽到一个单独方法中

```
    private Thread createUserThread(Long userId, Long sleepTime) {
        return new Thread(() -> {
            try {
                //模拟实际请求的上下文的 userId
                //初始化
                CommonRequestHeader commonRequestHeader = new CommonRequestHeader("",
userId);
                CommonRequestHeaderHandler.setContext(commonRequestHeader);
                log.info("当前线程名称为：{}，线程上下文为：{}",
Thread.currentThread().getName(), CommonRequestHeaderHandler.getContext());
                //feign 调用，采用 hystrix 线程隔离，会调用线程池中的空闲线程发送请求
                userFeign.findContext();
                Thread.sleep(sleepTime);
            } catch (Exception e) {
                log.error("调用失败");
            }
        });
    }

    @Test
    public void testInheritableThreadLocal() throws Exception {
        //创建线程
        Thread thread = createUserThread(324684371049185280L,0L);
        thread.start();
        Thread.sleep(30000L);
    }
```

测试结果：

```
当前线程名称为：Thread-14，线程上下文为：
CommonRequestHeader(authorization=,userId=324684371049185280)

当前线程名称为：hystrix-service-user-1，线程上下文为：CommonRequestHeader(authorization=,
userId=324684371049185280)
```

3.2.2 InheritableThreadLocal 的问题

其实到这里我也以为问题已经解决了，但是直到后来测试提了 bug。认证时获取的用户信息有误，我通过查询日志发现，发送出的日志和用户上下文中携带的不一致，所以 userId 也就不一致。

明明我们已经采用了 InheritableThreadLocal 来解决线程间变量传递的问题，为什么还有这种情况呢？？这就是一个非常严重的问题，比如线程池中会存在线程复用的情况，此时使用 InheritableThreadLocal 进行传值，因为 InheritableThreadLocal 只有新创建线程的时候才进行传值，线程复用并不会做这个操作。

问题测试：

构建测试方法，先消耗调用 hystrix 提供的 10 个线程资源，此时这 10 个线程资源的 InheritableThreadLocal 都已经有值，那么下一个请求从线程池中获取的资源必然就是之前设置的值

```
//测试启用hystrix，采用线程隔离模式，是否会导致线程变量无法传递的现象
@Test
public void testFeignHystrix() throws Exception {
    //创建异步线程，线程任务未调用feign 的线程：此时会调用hystrix 线程池中的线程
    Thread[] threads = new Thread[10];
    //阻塞其余线程
    for (int i = 0; i < 10; i++) {
        //将第一个线程使用调用后立刻释放
        long sleep = 20000L;
        if (i == 0) {
            sleep = 1000L;
        }
        threads[i] = createUserThread((long) i,sleep);
        threads[i].start();
    }
    //创建一个线程去消耗第一个线程资源
    Thread.sleep(20000L);
    Thread thread = createUserThread(324684371049185280L,5000L);
    thread.start();
    Thread.sleep(30000L);
}
```

测试结果：

消耗线程资源中变量的日志就不展示了，直接看最后的线程资源

```
当前线程名称为：Thread-29，线程上下文为：CommonRequestHeader(authorization=,
userId=324684371049185280)

当前线程名称为：hystrix-service-user-1，线程上下文为：CommonRequestHeader(authorization=,
userId=7)
```

果不其然，**问题出现了**，我明明设置的 userId 是 324684371049185280，但是最终调用时获取的确实之前耗用资源给的 userId 为 7。

3.2.3 Hystrix 自定义上下文策略

其实，我在项目中使用的是 TransmittableThreadLocal，是阿里对 InheritableThreadLocal 的继承扩展，是用对线程池之间变量传递方式的包装，目的就是做到线程复用时能将线程变量传递到另一个线程，这样听起来好像可以解决这个问题。

但是这个开源框架也有局限，就是必须要对创建的线程池进行修饰，而 hystrix 自定义的线程显然是在自身框架源码中，除非我们跑进去将源码的线程用阿里的 transmittable-thread-local 进行修饰，否则无法实现该功能。

基于已有能力进行修改才是最优雅的实现，通过在 SpringCloud 和 hystrix 的官网上检索相关应用场景，的确是有一些关于 hystrix 的扩展的。在 Spring Cloud 上会一些组件对 hystrix 进行自定义并发策略。最后通过一些博客的转述和翻译，大概能知道 hystrix 提供了自定义并发策略进行线程变量传递，并且 Spring Cloud 中的相关组件也进行了实践。看一下官网的描述：

The wrapCallable() method allows you to decorate every Callable executed by Hystrix. This can be essential to systems that rely upon ThreadLocal state for application functionality. The wrapping Callable can capture and copy state from parent to child thread as needed.

翻译过来就是，wrapCallable 方法允许我们修饰所有被 hystrix 执行的 callale 对象。这对于那些依赖 ThreadLocal 为基础的系统是非常必要的。修饰好的 Callable 对象可以根据需要从父线程捕获以及复制到子线程。这不正是需要达到的效果。

然后我在源码里找到了对应的抽象类，一般国外的项目基本会写点 demo 告诉别人这东西怎么用的，于是查了一下引用，有了发现，在 HystrixCommandTestWithCustomConcurrencyStrategy 这个类中就有 demo。不过 demo 比较长，大家有兴趣自己去看看，链接如下：
https://github.com/Netflix/Hystrix/blob/3cb21589895e9f8f87cfcdbc9d96d9f63d48b848/hystrix-core/src/test/java/com/netflix/hystrix/HystrixCommandTestWithCustomConcurrencyStrategy.java#L265

主要就是在新增了自定义的 HystrixConcurrencyStrategy，然后用 HystrixPlugins.getInstance().registerConcurrencyStrategy(strategy)注册一下就可以用了。比较方便，至于传递变量，则是在这个描述中讲了原理

```
    /**
     * Provides an opportunity to wrap/decorate a {@code Callable<T>} before execution.
     * <p>
     * This can be used to inject additional behavior such as copying of thread state (such as {@link ThreadLocal}).
     * <p>
     * <b>Default Implementation</b>
     * <p>
     * Pass-thru that does no wrapping.
     *
     * @param callable
     *          {@code Callable<T>} to be executed via a {@link ThreadPoolExecutor}
     * @return {@code Callable<T>} either as a pass-thru or wrapping the one given
     */
    public <T> Callable<T> wrapCallable(Callable<T> callable) {
        return callable;
    }
```

大概意思是我们有个机会在执行前先修饰一下这个 callable 参数，它最终会被 hystrix 的线程池所执行的。而 callable 本身是个接口，我们只要实现它就可以了。我们直接看实现：

```
// 重写装饰类，将上下文内容直接取出，放入自定义包装类
@Override
public <T> Callable<T> wrapCallable(Callable<T> callable) {
    CommonRequestHeader commonRequestHeader = CommonRequestHeaderHandler.getContext();
    return new WrappedCallable<>(callable,commonRequestHeader);
}

// 包装类，增加上下文属性，用于 hystrix 线程获取调用线程的 ThreadLocal 变量
static class WrappedCallable<T> implements Callable<T> {

    private final Callable<T> target;
    private final CommonRequestHeader commonRequestHeader;

    WrappedCallable(Callable<T> target, CommonRequestHeader commonRequestHeader) {
        this.target = target;
        this.commonRequestHeader = commonRequestHeader;
    }

    @Override
    public T call() throws Exception {
        try {
            CommonRequestHeaderHandler.setContext(commonRequestHeader);
            return target.call();
        }
        finally {
            CommonRequestHeaderHandler.clear();
        }
    }
}
```

实现思路：

- 定义一个实际类实现 Callable 接口，该接口只有一个方法 call 用于在线程池中执行

- 重点新增一个我们所需要的请求头对象，这就等着 wrap 调用的时候来初始化就好了
- 最重要的 call 方法就，我们直接用当前对象的属性 commonRequestHeader 取出设置到线程池当前线程的变量中
- 把这个自定义策略注册一下，我直接写在构造方法里了，然后把这个类注入到容器中就会自动调用构造方法进行策略注册了

```
        // Registers existing plugins excepts the Concurrent Strategy plugin.
        HystrixPlugins.getInstance().registerConcurrencyStrategy(this);
```

测试结果：

```
当前线程名称为：Thread-29，线程上下文为：CommonRequestHeader(authorization=,
userId=324684371049185280)

当前线程名称为：hystrix-service-user-10，线程上下文为：CommonRequestHeader(authorization=,
userId=324684371049185280)
userId：324684371049185280
```

完结撒花！

4 总结

这种问题的出现，是我们在业务应用时功能与应用框架冲突导致。框架要支持 hystrix 线程池配置资源隔离，我们需要线程间传递上下文以起到业务连贯性。在从线程本身改造的思考再到框架插件的思考，其实问题的本质是：

- 知道问题的根源：线程与线程池中的线程变量无法传递
- 如何获取解决问题的方案

那么解决问题的方案，从方法论上来说，一般是**主客两个方向**：

主观上先去找了线程作为调用方如何进行 ThreadLocal 的传递，发现对方是线程池，无法作为同一对象接收相关参数。

客观上，去寻找框架本身应对这种方案是否有解决方案，显然我们在框架本身找到了插件式变量传递。

希望从知识和思路上都能对大家有所帮助。