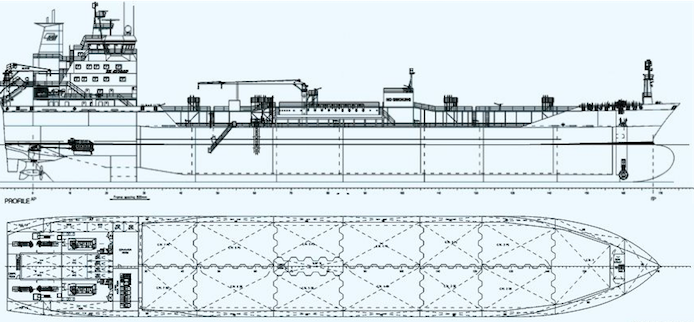
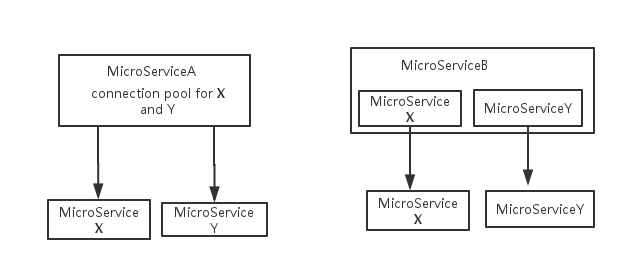
# Bulkhead Isolation

工业中使用舱壁将船舶划分为几个部分，以便在船体破坏的情况下，可以将船舶各个部件密封起来，如下图：



舱壁的概念在软件开发中被应用到隔离工作负载或者服务的关键资源，如连接池、内存和CPU，使用舱壁模式可以保护有限资源不被耗尽，例如数据库的连接数。这种模式避免单个工作负载或者服务消耗所有资源，从而导致其他服务出现故障，依次来增加系统的弹性。

在应用中需要使用REST通过HTTP连接两个不同的微服务，使用普通的线程池去维持这些连接，如下图左侧：

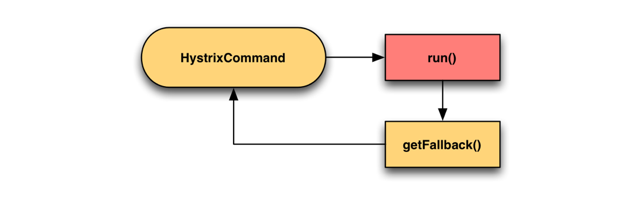


当其中一个服务由于某个原因出现异常，会出现线程池中所有线程因等待响应而被阻塞，从而造成服务雪崩。可以使用舱壁模式，将每个依赖服务分配独立的线程池进行资源隔离，如果服务X或者服务Y其中任何一个行为异常，不影响连接池的整体行为。舱壁模式降低服务对整个系统的影响，保护有限的资源不被耗尽，增加系统的弹性。

Docker通过舱壁模式实现进程的隔离，使得容器之间不会互相影响。而Hystrix则使用该模式实现线程池的隔离，其会为每个Hystrix命令创建独立的线程池，就算在某个Hystrix命令包装下的依赖服务出现延迟过高的请求，也只是对依赖服务的调用产生影响，而不会拖慢其他的服务。在Hystrix中，使用@HystrixCommand将某个函数包装成Hystrix命令，这里除了定义服务降级之外，Hystrix框架就会自动的为这个函数实现调用的隔离，依赖隔离和服务降级在使用的时候都是一体化实现的，其定义如下：

# HystrixCommand使用示例

下面是HystrixCommand的执行流程如下：

****

在HystrixCommand中除了HystrixBadRequestException异常之外，所有从run方法抛出的异常都算作失败，并触发降级getFalback和断路器逻辑。

HystrixBadReqeustException用在非法参数或非系统故障异常等不应触发回退逻辑的场景。执行示例如下：

*public class HelloWorldCommand extends HystrixCommand<String> {*

*private final String name;*

*public HelloWorldCommand(String name) {*

*super(*

*Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("HelloWorldGroup"))*

*.andCommandPropertiesDefaults( //设置执行时间timeout*

*HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(500)*

*)*

*);*

*this.name = name;*

*}*

*@Override*

*protected String run() throws Exception {*

*TimeUnit.MILLISECONDS.sleep(1000); //超时*

*return "Hello " + name + " Thread: " + Thread.currentThread().getName();*

*}*

*@Override //断路器调用逻辑*

*protected String getFallback() {*

*return "Execute Falled";*

*}*

*}*

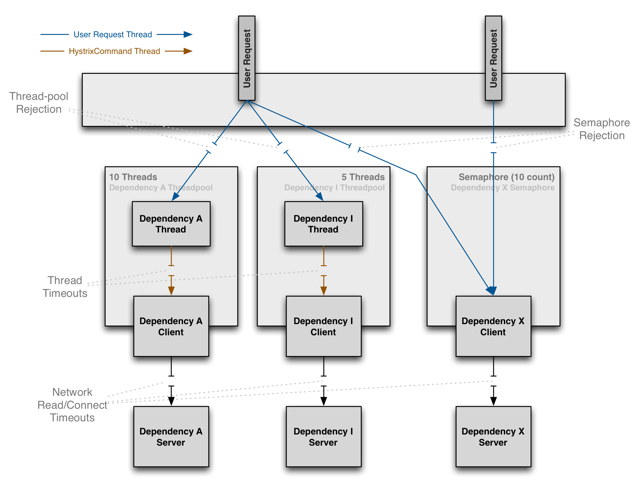
执行结果如下：

*Result: Execute Fallback*

超时触发了Fallback

# Hystrix线程隔离

在Hystrix隔离方式中可以采用线程/信号的方式，通过隔离限制依赖的并发量和阻塞扩散，如下图：



在继承HystrixCommand的构造函数中实现添加线程池参数来标记资源隔离，如下代码所示：

*.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey(name))*

*.andCommandPropertiesDefaults(HystrixCommandProperties.Setter()*

*.withExecutionTimeoutInMilliseconds(5000))*

*.andThreadPoolPropertiesDefaults (*

*HystrixThreadPoolProperties.Setter().withMaxQueueSize(10) //配置队列大小*

*.withCoreSize(2) // 配置线程池里的线程数)*

HystrixCommand执行时，将每个类型的Command都会生成一个线程池，该线程池被放入到ConcurrentHashMap中，其执行如下：

*HystrixThreadPool#getIstance(threadPoolKey)*

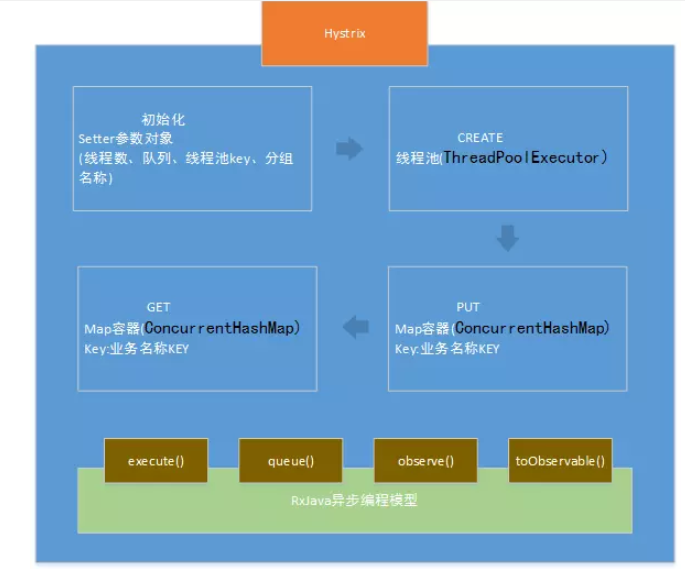
*synchronized (HystrixThreadPool.class) {*

*if (!threadPools.containsKey(key)) {*

*threadPools.put(key,*

*new HystrixThreadPoolDefault(threadPoolKey, propertiesBuilder)); } }*

对于HystrixCommand的执行如下：



1. 创建线程池

*public ThreadPoolExecutor getThreadPool(final HystrixThreadPoolKey threadPoolKey, HystrixThreadPoolProperties threadPoolProperties) {*

*......*

*return new ThreadPoolExecutor(dynamicCoreSize,*

*dynamicCoreSize,*

*keepAliveTime,*

*TimeUnit.MINUTES,*

*workQueue,*

*threadFactory);*

*}*

*}*

1. 执行代码，在示例中以queue的方式执行，其创建一个新线程运行run

*public Future<R> queue() {*

*final Future<R> delegate = this.toObservable().toBlocking().toFuture();*

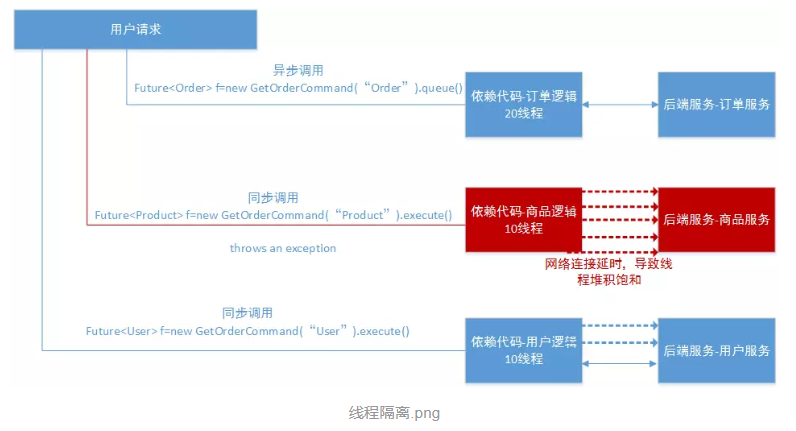
*......*

*Thread t = (Thread)HystrixCommand.this.executionThread.get();*

*if (t != null && !t.equals(Thread.currentThread())) {*

*t.interrupt();}}*

1. 执行依赖diam与请求线程分离，其结合使用RxJava来实现异步编程，通过线程池大小来控制并发访问量



其中依赖代码通过代理模式执行，其使用ThreadPool在第一部分已经描述。

https://github.com/Netflix/Hystrix

https://www.cnblogs.com/yepei/p/7169127.html

https://blog.csdn.net/a\_fengzi\_code\_110/article/details/51610655

https://github.com/Netflix/Hystrix/wiki/How-To-Use#command-thread-pool

https://www.jianshu.com/p/df1525d58c20

http://blog.sina.com.cn/s/blog\_4adc4b090102wdyr.html

https://www.jianshu.com/p/67e5eea73e86