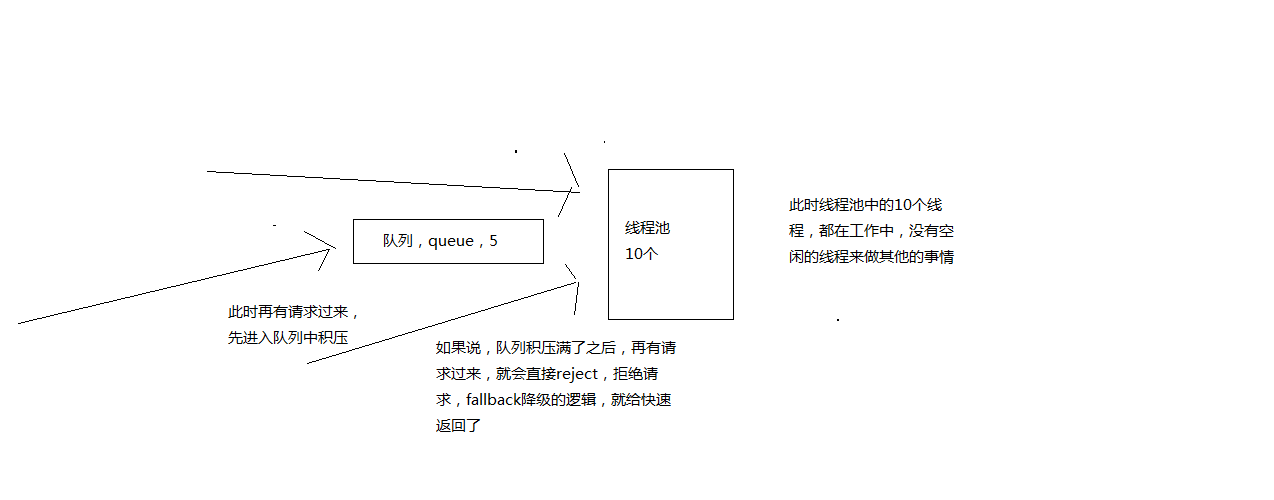
资源隔离，两种策略，线程池隔离，信号量隔离

对资源隔离这一块东西，做稍微更加深入一些的讲解，告诉你，除了可以选择隔离策略以外，对你选择的隔离策略，可以做一定的细粒度的一些控制



线程池和queue的工作原理

1、execution.isolation.strategy

指定了HystrixCommand.run()的资源隔离策略，THREAD或者SEMAPHORE，一种是基于线程池，一种是信号量

线程池机制，每个command运行在一个线程中，限流是通过线程池的大小来控制的

信号量机制，command是运行在调用线程中，但是通过信号量的容量来进行限流

如何在线程池和信号量之间做选择？

默认的策略就是线程池

线程池其实最大的好处就是对于网络访问请求，如果有超时的话，可以避免调用线程阻塞住

而使用信号量的场景，通常是针对超大并发量的场景下，每个服务实例每秒都几百的QPS，那么此时你用线程池的话，线程一般不会太多，可能撑不住那么高的并发，如果要撑住，可能要耗费大量的线程资源，那么就是用信号量，来进行限流保护

一般用信号量常见于那种基于纯内存的一些业务逻辑服务，而不涉及到任何网络访问请求

netflix有100+的command运行在40+的线程池中，只有少数command是不运行在线程池中的，就是从纯内存中获取一些元数据，或者是对多个command包装起来的facacde command，是用信号量限流的

// to use thread isolation

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(ExecutionIsolationStrategy.THREAD)

// to use semaphore isolation

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)

2、command名称和command组

线程池隔离，依赖服务->接口->线程池，如何来划分

你的每个command，都可以设置一个自己的名称，同时可以设置一个自己的组

private static final Setter cachedSetter =

Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"));

public CommandHelloWorld(String name) {

super(cachedSetter);

this.name = name;

}

command group，是一个非常重要的概念，默认情况下，因为就是通过command group来定义一个线程池的，而且还会通过command group来聚合一些监控和报警信息

同一个command group中的请求，都会进入同一个线程池中

3、command线程池

threadpool key代表了一个HystrixThreadPool，用来进行统一监控，统计，缓存

默认的threadpool key就是command group名称

每个command都会跟它的threadpool key对应的thread pool绑定在一起

如果不想直接用command group，也可以手动设置thread pool name

public CommandHelloWorld(String name) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"))

.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("HelloWorldPool")));

this.name = name;

}

command threadpool -> command group -> command key

command key，代表了一类command，一般来说，代表了底层的依赖服务的一个接口

command group，代表了某一个底层的依赖服务，合理，一个依赖服务可能会暴露出来多个接口，每个接口就是一个command key

command group，在逻辑上去组织起来一堆command key的调用，统计信息，成功次数，timeout超时次数，失败次数，可以看到某一个服务整体的一些访问情况

command group，一般来说，推荐是根据一个服务去划分出一个线程池，command key默认都是属于同一个线程池的

比如说你以一个服务为粒度，估算出来这个服务每秒的所有接口加起来的整体QPS在100左右

你调用那个服务的当前服务，部署了10个服务实例，每个服务实例上，其实用这个command group对应这个服务，给一个线程池，量大概在10个左右，就可以了，你对整个服务的整体的访问QPS大概在每秒100左右

一般来说，command group是用来在逻辑上组合一堆command的

举个例子，对于一个服务中的某个功能模块来说，希望将这个功能模块内的所有command放在一个group中，那么在监控和报警的时候可以放一起看

command group，对应了一个服务，但是这个服务暴露出来的几个接口，访问量很不一样，差异非常之大

你可能就希望在这个服务command group内部，包含的对应多个接口的command key，做一些细粒度的资源隔离

对同一个服务的不同接口，都使用不同的线程池

command key -> command group

command key -> 自己的threadpool key

逻辑上来说，多个command key属于一个command group，在做统计的时候，会放在一起统计

每个command key有自己的线程池，每个接口有自己的线程池，去做资源隔离和限流

但是对于thread pool资源隔离来说，可能是希望能够拆分的更加一致一些，比如在一个功能模块内，对不同的请求可以使用不同的thread pool

command group一般来说，可以是对应一个服务，多个command key对应这个服务的多个接口，多个接口的调用共享同一个线程池

如果说你的command key，要用自己的线程池，可以定义自己的threadpool key，就ok了

4、coreSize

设置线程池的大小，默认是10

HystrixThreadPoolProperties.Setter()

.withCoreSize(int value)

一般来说，用这个默认的10个线程大小就够了

5、queueSizeRejectionThreshold

控制queue满后reject的threshold，因为maxQueueSize不允许热修改，因此提供这个参数可以热修改，控制队列的最大大小

HystrixCommand在提交到线程池之前，其实会先进入一个队列中，这个队列满了之后，才会reject

默认值是5

HystrixThreadPoolProperties.Setter()

.withQueueSizeRejectionThreshold(int value)

6、execution.isolation.semaphore.maxConcurrentRequests

设置使用SEMAPHORE隔离策略的时候，允许访问的最大并发量，超过这个最大并发量，请求直接被reject

这个并发量的设置，跟线程池大小的设置，应该是类似的，但是基于信号量的话，性能会好很多，而且hystrix框架本身的开销会小很多

默认值是10，设置的小一些，否则因为信号量是基于调用线程去执行command的，而且不能从timeout中抽离，因此一旦设置的太大，而且有延时发生，可能瞬间导致tomcat本身的线程资源本占满

HystrixCommandProperties.Setter()

.withExecutionIsolationSemaphoreMaxConcurrentRequests(int value)

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案