美团点评技术团队

服务容错模式

绿麟 ·2016-11-11 16:27

背景

随着美团点评服务框架和服务治理体系的逐步成熟，服务化已成为公司内部系统设计的趋势。本着大系统小做、职责单一的原则，我们度假技术团队对业务系统进行了不少服务化拆分工作。随着业务复杂度的增加，依赖的服务也逐步增加，出现了不少由于服务调用出现异常问题而导致的重大事故，如：

1）系统依赖的某个服务发生延迟或者故障，数秒内导致所有应用资源（线程，队列等）被耗尽，造成所谓的雪崩效应 (Cascading Failure (https://en.wikipedia.org/wiki/Cascading_failure))，导致整个系统拒绝对外提供服务。

2）系统遭受恶意爬虫袭击，在放大效应下没有对下游依赖服务做好限速处理，最终导致下游服务崩溃。

容错是一个很大的话题，受篇幅所限，本文将介绍仅限定在服务调用间常用的一些容错模式。

设计原则

服务容错的设计有个基本原则，就是“Design for Failure”。为了避免出现“千里之堤溃于蚁穴”这种情况，在设计上需要考虑到各种边界场景和对于服务间调用出现的异常或延迟情况，同时在设计和编程时也要考虑周到。这一切都是为了达到以下目标：

1）一个依赖服务的故障不会严重破坏用户的体验。

2）系统能自动或半自动处理故障，具备自我恢复能力。

基于这个原则和目标，衍生出下文将要介绍的一些模式，能够解决分布式服务调用中的一些问题，提高系统在故障发生时的存活能力。

一些经典的容错模式

所谓模式，其实就是某种场景下一类问题及其解决方案的总结归纳，往往可以重用。模式可以指导我们完成任务，作出合理的系统设计方案，达到事半功倍的效果。而在服务容错这个方向，行业内已经有了不少实践总结出来的解决方案。

超时与重试（Timeout and Retry）

超时模式，是一种最常见的容错模式，在美团点评的工程实践中大量存在。常见的有设置网络连接超时时间，一次RPC的响应超时时间等。在分布式服务调用的场景中，它主要解决了当依赖服务出现建立网络连接或响应延迟，不用无限等待的问题，调用方可以根据事先设计的超时时间中断调用，及时释放关键资源，如

或响应延迟，不用无限等待的问题，调用方可以根据事先设计的超时时间中断调用，及时释放关键资源，如Web容器的连接数，数据库连接数等，避免整个系统资源耗尽出现拒绝对外提供服务这种情况。

重试模式，一般和超时模式结合使用，适用于对于下游服务的数据强依赖的场景（不强依赖的场景不建议使用！），通过重试来保证数据的可靠性或一致性，常用于因网络抖动等导致服务调用出现超时的场景。与超时时间设置结合使用后，需要考虑接口的响应时间分布情况，超时时间可以设置为依赖服务接口99.5%响应时间的值，重试次数一般1-2次为宜，否则会导致请求响应时间延长，拖累到整个系统。

一些实现说明：

```
public class RetryCommand<T> {
    private int maxRetries = 2;// 重试次数 默认2次
    private long retryInterval = 5;//重试间隔时间ms 默认5ms
    private Map<String, Object> params;

       public RetryCommand() {

    }

    public RetryCommand(long retryInterval, int maxRetries) {
           this.retryInterval = retryInterval;
        this.maxRetries = maxRetries;
    }

    public T command(Map<String, Object> params){
          //Some remote service call with timeout
           serviceA.doSomethingWithTimeOut(timeout);
    }

    private final T retry() throws RuntimeException {
        int retryCounter = 0;
        while (retryCounter < maxRetries) {
            try {
                return command(params);
            } catch (Exception e) {
                retryCounter++;
                if (retryCounter >= maxRetries) {
                    break;
               }
            }
    }
    throw new RuntimeException("Command failed on all of " + maxRetries + " retries");
}

       //省略
}
```

限流(Rate Limiting/Load Shedder)

限流模式，常用于下游服务容量有限，但又怕出现突发流量猛增（如恶意爬虫，节假日大促等）而导致下游服务因压力过大而拒绝服务的场景。常见的限流模式有控制并发和控制速率，一个是限制并发的数量，一个是限制并发访问的速率。

控制并发

属于一种较常见的限流手段，在工程实践中可以通过信号量机制（如Java中的Semaphore）来控制，举个例子：

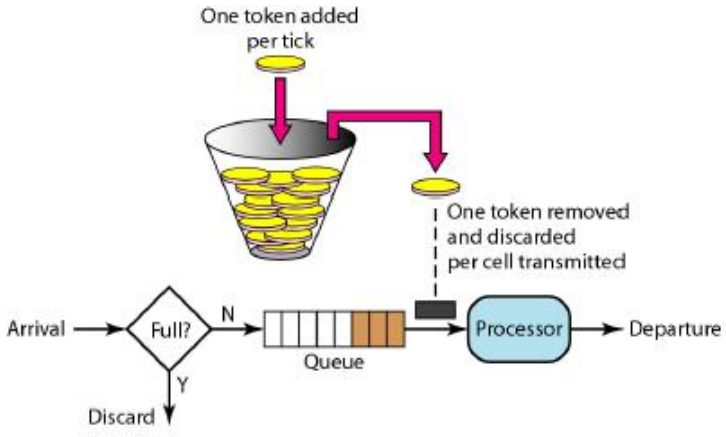
假如有一个需求，要读取几万个文件的数据，因为都是IO密集型任务，我们可以启动几十个线程并发的读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有十个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接。这个时候，我们就可以使用Semaphore来控制并发数，如：

```
public class SemaphoreTest {

    private static final int THREAD_COUNT = 30;

    private static ExecutorService threadPool = Executors
        .newFixedThreadPool(THREAD_COUNT);

    private static Semaphore s = new Semaphore(10);

    public static void main(String[] args) {
        for (int i = 0; i < THREAD_COUNT; i++) {
            threadPool.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        s.acquire();
                        System.out.println("save data");
                        s.release();
                    } catch (InterruptedException e) {
                        e.printStack();
                    }
                }
            });
        }

        threadPool.shutdown();
    }
}
```

在代码中，虽然有30个线程在执行，但是只允许10个并发的执行。Semaphore的构造方法Semaphore(int permits) 接受一个整型的数字，表示可用的许可证数量。Semaphore(10)表示允许10个线程获取许可证，也就是最大并发数是10。Semaphore的用法也很简单，首先线程使用Semaphore的acquire()获取一个许可证，使用完之后调用release()归还许可证，还可以用tryAcquire()方法尝试获取许可证。

控制速率

在我们的工程实践中，常见的是使用令牌桶算法来实现这种模式，其他如漏桶算法也可以实现控制速率，但在我们的工程实践中使用不多，这里不做介绍，读者请自行了解。



在Wikipedia上，令牌桶算法是这么描述的：

1. 每秒会有r个令牌放入桶中，或者说，每过1/r秒桶中增加一个令牌。
2. 桶中最多存放b个令牌，如果桶满了，新放入的令牌会被丢弃。
3. 当一个n字节的数据包到达时，消耗n个令牌，然后发送该数据包。
4. 如果桶中可用令牌小于n，则该数据包将被缓存或丢弃。

令牌桶控制的是一个时间窗口内通过的数据量，在API层面我们常说的QPS、TPS，正好是一个时间窗口内的请求量或者事务量，只不过时间窗口限定在1s罢了。以一个恒定的速度往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。令牌桶的另外一个好处是可以方便的改变速度，一旦需要提高速率，则按需提高放入桶中的令牌的速率。

在我们的工程实践中，通常使用Guava中的Ratelimiter来实现控制速率，如我们不希望每秒的任务提交超过两个：

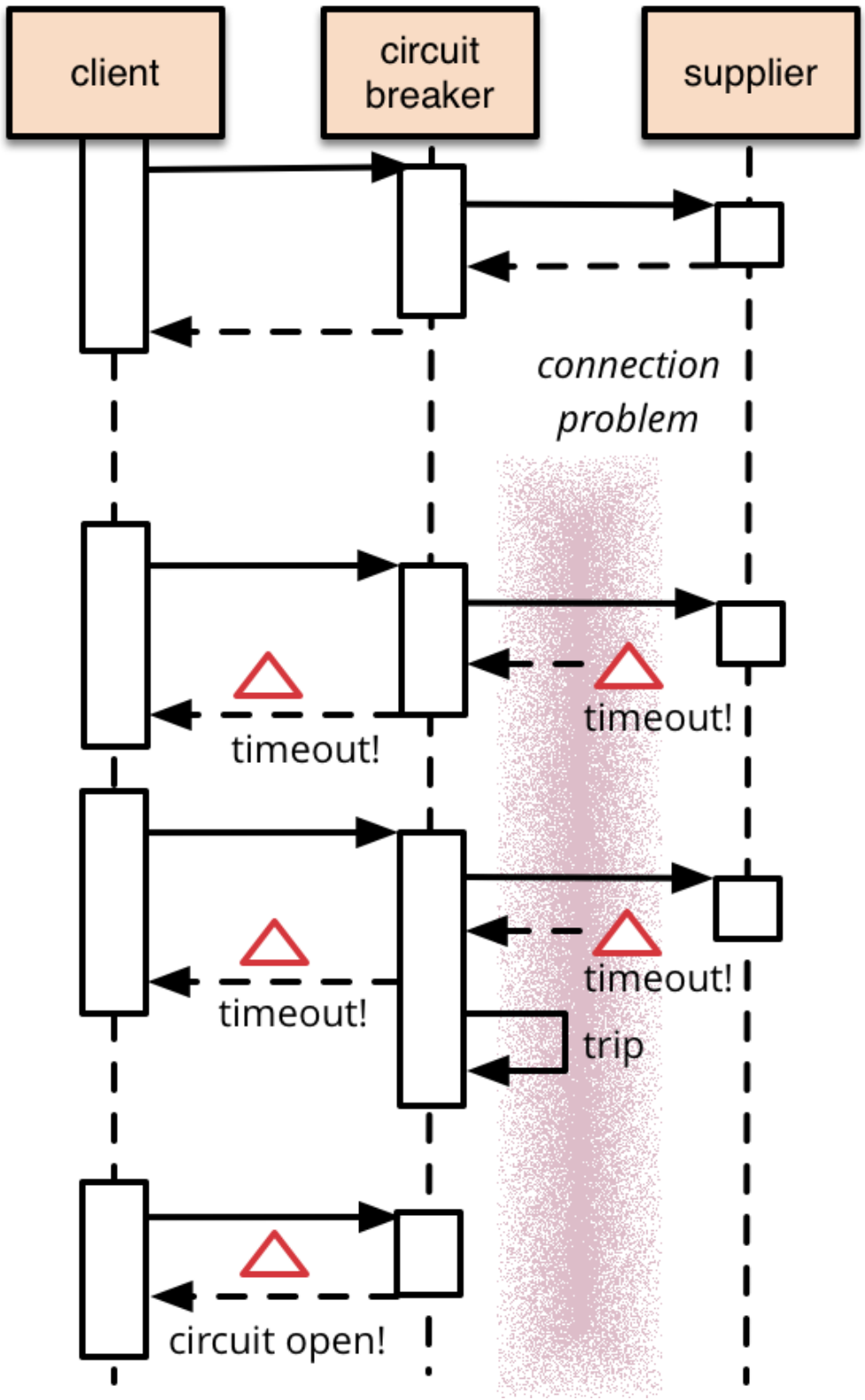
```
//速率是每秒两个许可
final RateLimiter rateLimiter = RateLimiter.create(2.0);

void submitTasks(List tasks, Executor executor) {
    for (Runnable task : tasks) {
        rateLimiter.acquire(); // 也许需要等待
        executor.execute(task);
    }
}
```

电路熔断器(Circuit Breaker)

在我们的工程实践中，偶尔会遇到一些服务由于网络连接超时，系统有异常或load过高出现暂时不可用等情况，导致对这些服务的调用失败，可能需要一段时间才能修复，这种对请求的阻塞可能会占用宝贵的系统资源，如：内存，线程，数据库连接等等，最坏的情况下会导致这些资源被消耗殆尽，使得系统里不相关的部分所使用的资源也耗尽从而拖累整个系统。在这种情况下，调用操作能够立即返回错误而不是等待超时的发生或者重试可能是一种更好的选择，只有当被调用的服务有可能成功时我们再去尝试。

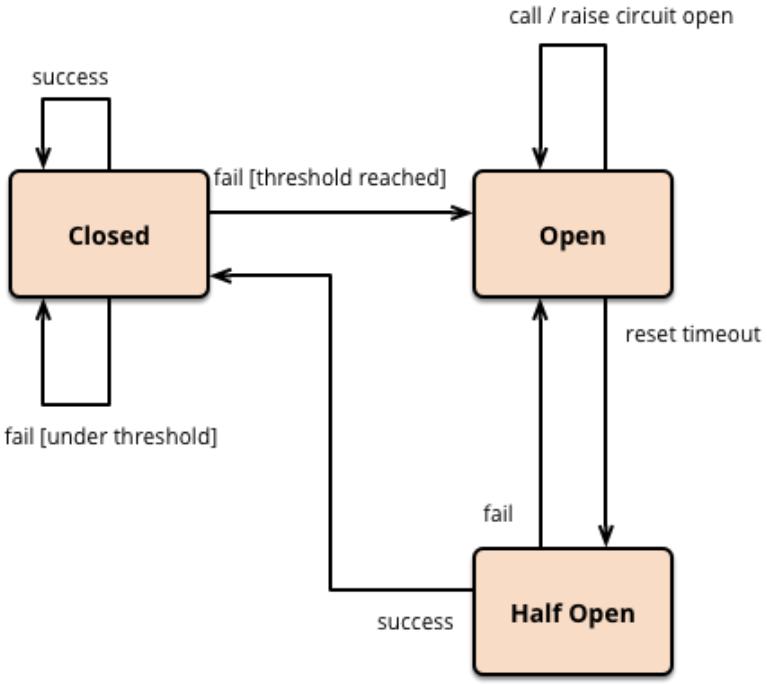
熔断器模式可以防止我们的系统不断地尝试执行可能会失败的调用，使得我们的系统继续执行而不用等待修正错误，或者浪费CPU时间去等到长时间的超时产生。熔断器模式也可以使我们系统能够检测错误是否已经修正。如果已经修正，系统会再次尝试调用操作。下图是介绍熔断器模式的调用流程：

修正，如果已经修正，系统会再次尝试调用操作。下图是下使用熔断器模式的调用流程：



可以从图中看出，当超时出现的次数达到一定条件后，熔断器会触发打开状态，客户端的下次调用将直接返回，不用等待超时产生。

在熔断器内部，往往有以下几种状态：

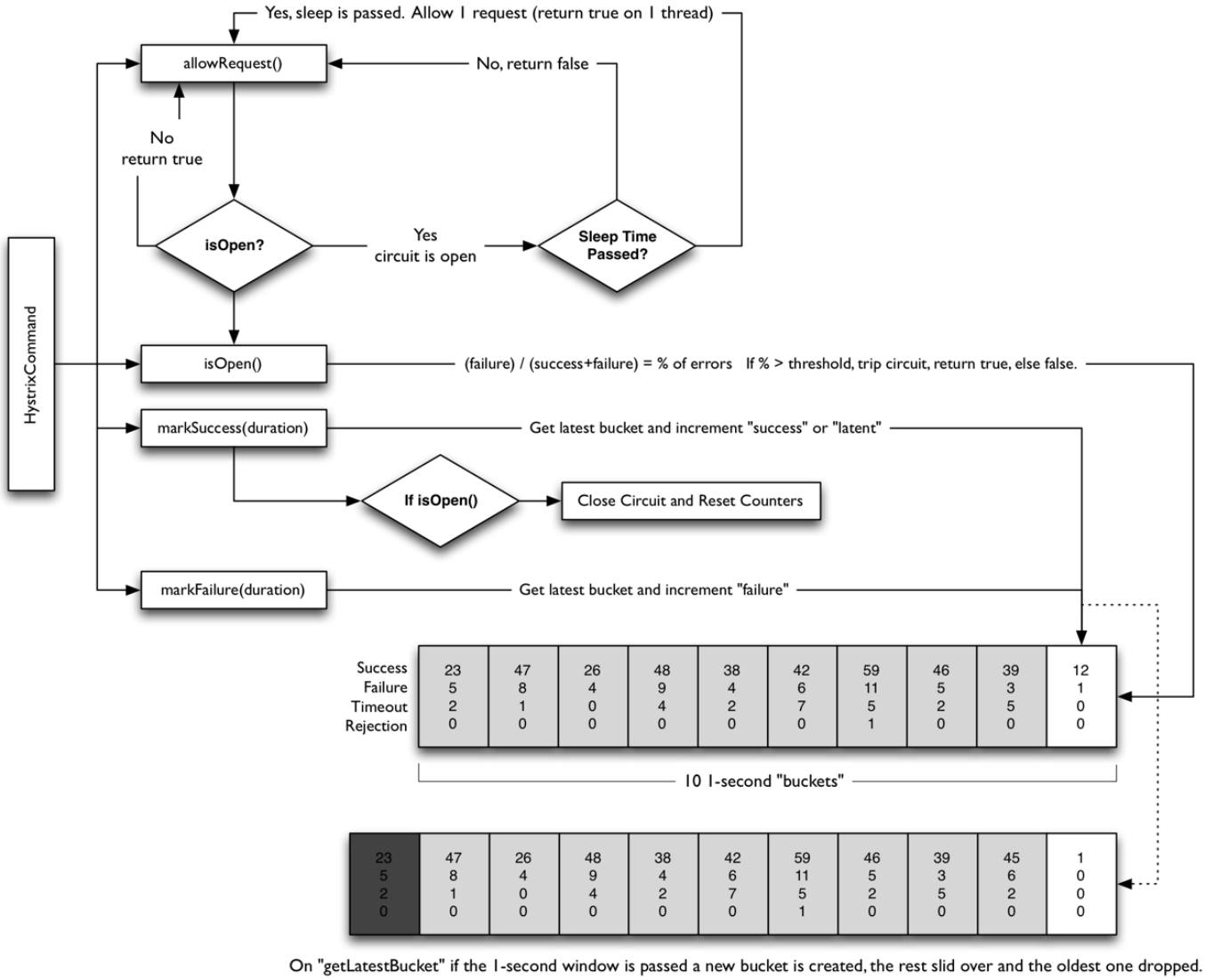


1）闭合（closed）状态：该状态下能够对目标服务或方法进行正常的调用。熔断器类维护了一个时间窗口内调用失败的次数，如果某次调用失败，则失败次数加1。如果最近失败次数超过了在给定的时间窗口内允许失败的阈值(可以是数量也可以是比例)，则熔断器类切换到断开(Open)状态。此时熔断器设置了一个计时器，当时钟超过了该时间，则切换到半断开（Half-Open）状态，该睡眠时间的设定是给了系统一次机会来修正导致调用失败的错误。

2）断开(Open)状态：在该状态下，对目标服务或方法的请求会立即返回错误响应，如果设置了fallback方法，则会进入fallback的流程。

3）半断开（Half-Open）状态：允许对目标服务或方法的一定数量的请求可以去调用服务。如果这些请求对服务的调用成功，那么可以认为之前导致调用失败的错误已经修正，此时熔断器切换到闭合状态（并且将错误计数器重置）；如果这一定数量的请求有调用失败的情况，则认为导致之前调用失败的问题仍然存在，熔断器切回到断开方式，然后开始重置计时器来给系统一定的时间来修正错误。半断开状态能够有效防止正在恢复中的服务被突然而来的大量请求再次拖垮。

在我们的工程实践中，熔断器模式往往应用于服务的自动降级，在实现上主要基于Netflix开源的组件Hystrix来实现，下图和代码分别是Hystrix中熔断器的原理和定义，更多了解可以查看Hystrix的源码：



On "getLatestBucket" if the 1-second window is passed a new bucket is created, the rest slid over and the oldest one dropped.

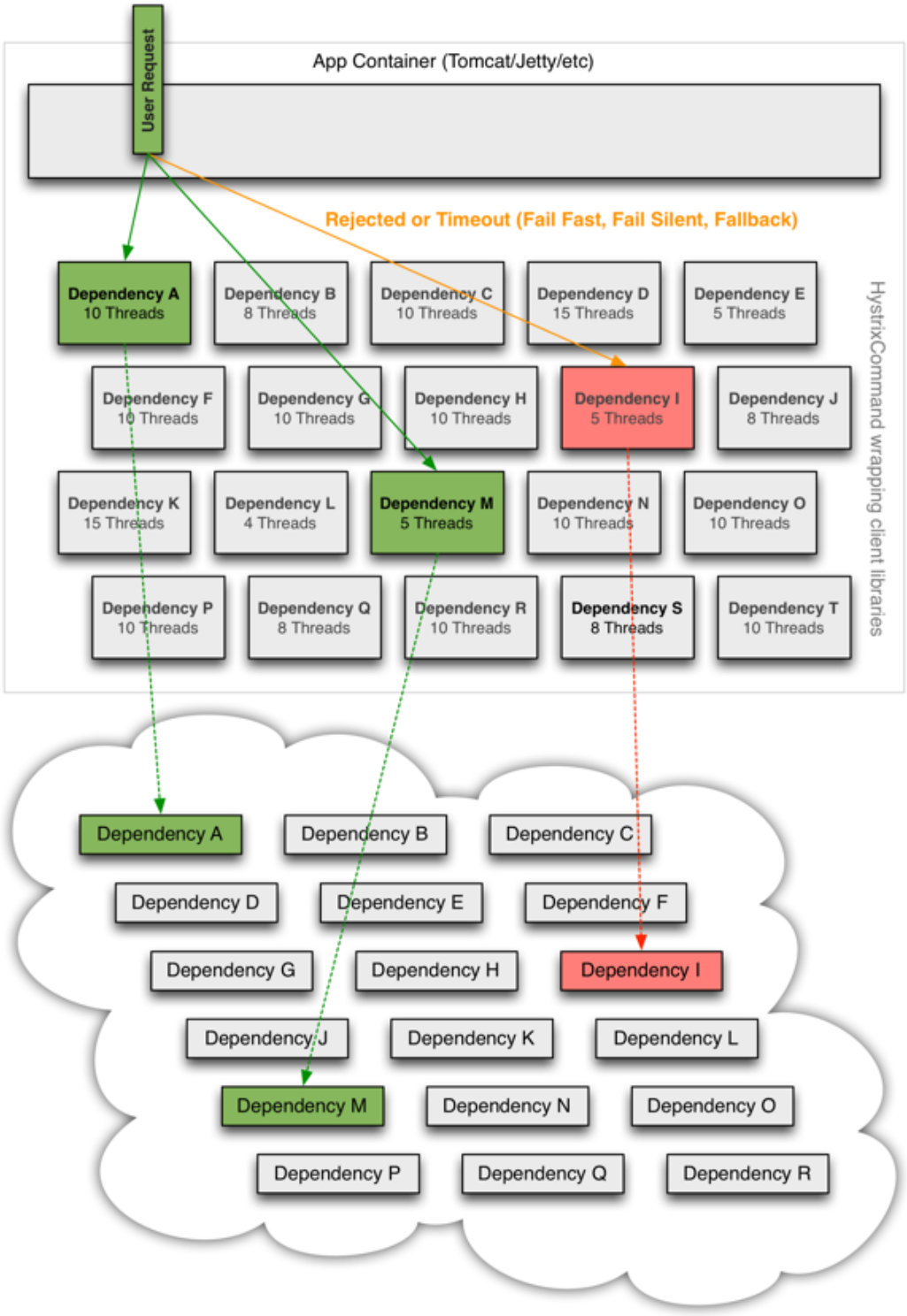
```
public interface HystrixCircuitBreaker {

    /**
     * Every {@link HystrixCommand} requests asks this if it is allowed to proceed or not.
      * <p>
      * This takes into account the half-open logic which allows some requests through when determining if it
      *
      * @return boolean whether a request should be permitted
      */
     public boolean allowRequest();

     /**
      * Whether the circuit is currently open (tripped).
      *
      * @return boolean state of circuit breaker
     */
     public boolean isOpen();

    /**
     * Invoked on successful executions from {@link HystrixCommand} as part of feedback mechanism when in a H
     */
    public void markSuccess();
}
```

舱壁隔离(Bulkhead Isolation)

在造船行业，往往使用此类模式对船舱进行隔离，利用舱壁将不同的船舱隔离起来，这样如果一个船舱破了进水，只损失一个船舱，其它船舱可以不受影响，而借鉴造船行业的经验，这种模式也在软件行业得到使用。

线程隔离(Thread Isolation)就是这种模式的常见的一个场景。例如，系统A调用了ServiceB/ServiceC/ServiceD三个远程服务，且部署A的容器一共有120个工作线程，采用线程隔离机制，可以给对ServiceB/ServiceC/ServiceD的调用各分配40个线程。当ServiceB慢了，给ServiceB分配的40个线程因慢而阻塞并最终耗尽，线程隔离可以保证给ServiceC/ServiceD分配的80个线程可以不受影响。如果没有这种隔离机制，当ServiceB慢的时候，120个工作线程会很快全部被对ServiceB的调用吃光，整个系统会全部慢下来，甚至出现系统停止响应的情况。

这种Case在我们实践中经常遇到，如某接口由于数据库慢查询，外部RPC调用超时导致整个系统的线程数过高，连接数耗尽等。我们可以使用舱壁隔离模式，为这种依赖服务调用维护一个小的线程池，当一个依赖服务由于响应慢导致线程池任务满的时候，不会影响到其他依赖服务的调用，它的缺点就是会增加线程数。



无论是超时/重试，熔断器，还是舱壁隔离模式，它们在使用过程中都会出现异常情况，异常情况的处理方式间接影响到用户的体验，针对异常情况的处理也有一种模式支撑，这就是回退(fallback)模式。

回退(Fallback)

在超时，重试失败，熔断或者限流发生的时候，为了及时恢复服务或者不影响到用户体验，需要提供回退的机制，常见的回退策略有：

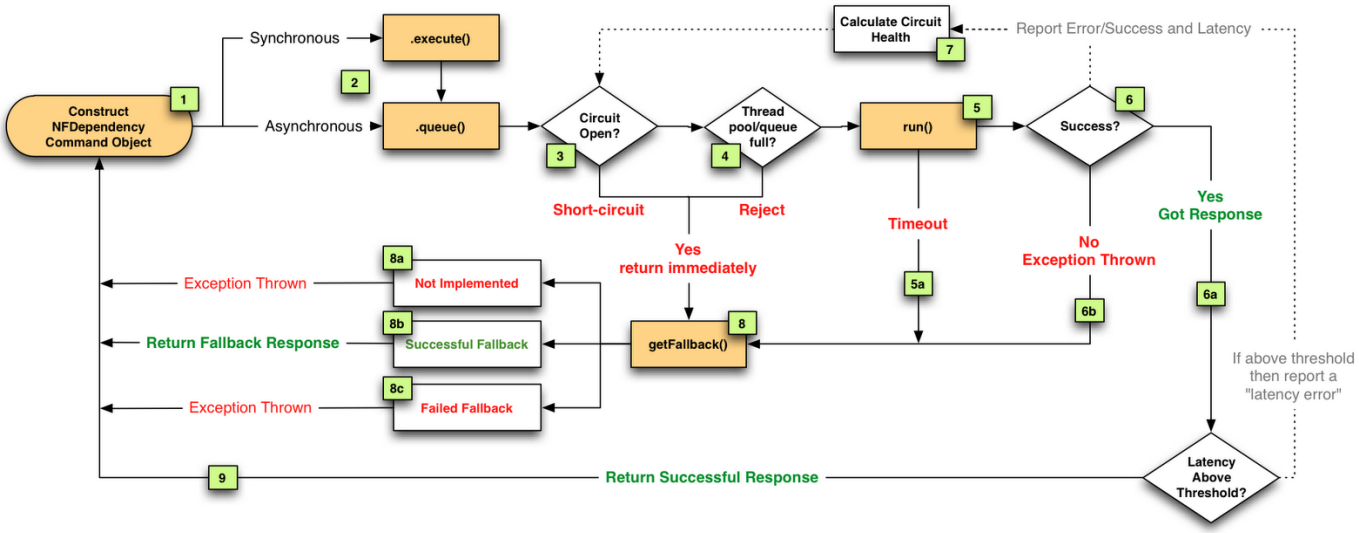
1. 自定义处理：在这种场景下，可以使用默认数据，本地数据，缓存数据来临时支撑，也可以将请求放入队列，或者使用备

用服务获取数据等，适用于业务的关键流程与严重影响用户体验的场景，如商家/产品信息等核心服务。

2. 故障沉默（fail-silent）：直接返回空值或缺省值，适用于可降级功能的场景，如产品推荐之类的功能，数据为空也不太影响用户体验。
3. 快速失败（fail-fast）：直接抛出异常，适用于数据非强依赖的场景，如非核心服务超时的处理。

应用实例

在实际的工程实践中，这四种模式既可以单独使用，也可以组合使用，为了让读者更好的理解这些模式的应用，下面以Netflix的开源组件Hystrix的流程为例说明。



图中流程的说明:

1. 将远程服务调用逻辑封装进一个HystrixCommand。
2. 对于每次服务调用可以使用同步或异步机制，对应执行execute()或queue()。
3. 判断熔断器(circuit-breaker)是否打开或者半打开状态，如果打开跳到步骤8，进行回退策略，如果关闭进入步骤4。
4. 判断线程池/队列/信号量（使用了舱壁隔离模式）是否跑满，如果跑满进入回退步骤8，否则继续后续步骤5。
5. run方法中执行了实际的服务调用。

   a. 服务调用发生超时时，进入步骤8。
6. 判断run方法中的代码是否执行成功。

   a. 执行成功返回结果。

   b. 执行中出现错误则进入步骤8。
7. 所有的运行状态(成功，失败，拒绝，超时)上报给熔断器，用于统计从而影响熔断器状态。
8. 进入getFallback()回退逻辑。

   a. 没有实现getFallback()回退逻辑的调用将直接抛出异常。

   b. 回退逻辑调用成功直接返回。

   c. 回退逻辑调用失败抛出异常。