高并发系统保护

在开发高并发系统时有三把利器用来保护系统：缓存、降级和限流。

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。

系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。本文将介绍一些笔者在实际工作中遇到的或见到过的一些降级方案供大家参考。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

雪崩效应常见场景

- 硬件故障：如服务器宕机，机房断电，光纤被挖断等。
- 流量激增：如异常流量，重试加大流量等。
- 缓存穿透：一般发生在应用重启，所有缓存失效时，以及短时间内大量缓存失效时。大量的缓存不命中，使请求直击后端服务，造成服务提供者超负荷运行，引起服务不可用。
- 程序BUG：如程序逻辑导致内存泄漏，JVM长时间FullGC等。
- 同步等待：服务间采用同步调用模式，同步等待造成的资源耗尽。

降级预案

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

降级的类别

- 降级按照是否自动化可分为：自动开关降级和人工开关降级。
- 降级按照功能可分为：读服务降级、写服务降级。
- 降级按照处于的系统层次可分为：多级降级。

降级的功能点

降级的功能点主要从服务端链路考虑，即根据用户访问的服务调用链路来梳理哪里需要降级：

页面降级：在大促或者某些特殊情况下，某些页面占用了一些稀缺服务资源，在紧急情况下可以对其整个降级，以达到丢卒保帅；

页面片段降级：比如商品详情页中的商家部分因为数据错误了，此时需要对其进行降级；

页面异步请求降级：比如商品详情页上有推荐信息/配送至等异步加载的请求，如果这些信息响应慢或者后端服务有问题，可以进行降级；

服务功能降级：比如渲染商品详情页时需要调用一些不太重要的服务：相关分类、热销榜等，而这些服务在异常情况下直接不获取，即降级即可；

读降级：比如多级缓存模式，如果后端服务有问题，可以降级为只读缓存，这种方式适用于对读一致性要求不高的场景；

写降级：比如秒杀抢购，我们可以只进行Cache的更新，然后异步同步扣减库存到DB，保证最终一致性即可，此时可以将DB降级为Cache。

爬虫降级：在大促活动时，可以将爬虫流量导向静态页或者返回空数据从而降级保护后端稀缺资源。

降级策略

1、自动开关降级

自动降级是根据系统负载、资源使用情况、SLA等指标进行降级。

超时降级

当访问的数据库/http服务/远程调用响应慢或者长时间响应慢，且该服务不是核心服务的话可以在超时后自动降级；

对于这种服务是可以超时降级的。如果是调用别人的远程服务，和对方定义一个服务响应最大时间，如果超时了则自动降级。

在实际场景用一定主要配置好超时时间和超时重试次数和机制。

统计失败次数降级

有时候依赖一些不稳定的API，比如调用外部机票服务，当失败调用次数达到一定阀值自动降级；然后通过异步线程去探测服务是否恢复了，则取消降级。

故障降级

比如要调用的远程服务挂掉了（网络故障、DNS故障、http服务返回错误的状态码、rpc服务抛出异常），则可以直接降级。

降级后的处理方案有：

限流降级

当我们去秒杀或者抢购一些限购商品时，此时可能会因为访问量太大而导致系统崩溃，此时开发者会使用限流来进行限制访问量，当达到限流阀值，后续请求会被降级；

降级后的处理方案可以是：

排队页面（将用户导流到排队页面等一会重试）
无货（直接告知用户没货了）
错误页（如活动太火爆了，稍后重试）

2、人工开关降级

- 在大促期间通过监控发现线上的一些服务存在问题，这个时候需要暂时将这些服务摘掉；
- 还有有时候通过任务系统调用一些服务，但是服务依赖的数据库可能存在：网卡被打满了、挂掉了或者很多慢查询，此时需要暂停下任务系统让服务方进行处理；
- 还有发现突然调用量太大，可能需要改变处理方式（比如同步转换为异步）；

此时就可以**使用开关来完成降级。**

开关可以存放到配置文档、存放到数据库、存放到Redis/ZooKeeper；如果不是存放在本地，可以定期同步开关数据（比如1秒同步一次）。然后通过判断某个KEY的值来决定是否降级。

另外对于新开发的服务想上线进行灰度测试；但是不太确定该服务的逻辑是否正确，此时就需要设置开关，当新服务有问题可以通过开关切换回老服务。

还有多机房服务，如果某个机房挂掉了，此时需要将一个机房的服务切到另一个机房，此时也可以通过开关完成切换。

还有一些是因为功能问题需要暂时屏蔽掉某些功能，比如商品规格参数数据有问题，数据问题不能用回滚解决，此时需要开关控制降级。

3、读服务降级

对于读服务降级一般采用的策略有：

暂时切换读（降级到读缓存、降级到走静态化）
暂时屏蔽读（屏蔽读入口、屏蔽某个读服务）

接入层缓存→应用层本地缓存→分布式缓存→RPC服务/DB

我们会在接入层、应用层设置开关，当分布式缓存、RPC服务/DB有问题自动降级为不调用。当然这种情况适用于对读一致性要求不高的场景。

页面降级、页面片段降级、页面异步请求降级都是读服务降级，目的是丢卒保帅（比如因为这些服务也要使用核心资源、或者占了带宽影响到核心服务）或者因数据问题暂时屏蔽。

还有一种是页面静态化场景：

动态化降级为静态化：比如平时网站可以走动态化渲染商品详情页，但是到了大促来临之际可以将其切换为静态化来减少对核心资源的占用，而且可以提升性能；其他还有如列表页、首页、频道页都可以这么玩；可以通过一个进程定期的推送静态页到缓存或者生成到磁盘，出问题时直接切过去；

静态化降级为动态化：比如当使用静态化来实现商品详情页架构时，平时使用静态化来提供服务，但是因为特殊原因静态化页面有问题了，需要暂时切换回动态化来保证服务正确性。

以上都保证出问题了有预案，用户还是可以使用网站，不影响用户购物。

4、写服务降级

写服务在大多数场景下是不可降级的，不过可以通过一些迂回战术来解决问题。比如将同步操作转换为异步操作，或者限制写的量/比例。

比如扣减库存一般这样操作：

方案1：

a、扣减DB库存；

b、扣减成功后更新Redis中的库存；

方案2：

a、扣减Redis库存；

b、同步扣减DB库存，如果扣减失败则回滚Redis库存；

前两种方案非常依赖DB，假设此时DB性能跟不上则扣减库存就会遇到问题；因此我们可以想到

方案3：

a、扣减Redis库存：

b、正常同步扣减DB库存，性能扛不住时降级为发送一条扣减DB库存的消息，然后异步进行DB库存扣减实现最终一致即可；

这种方式发送扣减DB库存消息也可能成为瓶颈；这种情况我们可以考虑

方案4：

a、扣减Redis库存；

b、正常同步扣减DB库存，性能扛不住时降级为写扣减DB库存消息到本机，然后本机通过异步进行DB库存扣减来实现最终一致性。

也就是说正常情况可以同步扣减库存，在性能扛不住时降级为异步；另外如果是秒杀场景可以直接降级为异步，从而保护系统。

还有如下单操作可以在大促时暂时降级将下单数据写入Redis，然后等峰值过去了再同步回DB，当然也有更好的解决方案，但是更复杂，不是本文的重点。

还有如用户评价，如果评价量太大，也可以把评价从同步写降级为异步写。当然也可以对评价按钮进行按比例开放（比如一些人的看不到评价操作按钮）。比如评价成功后会发一些奖励，在必要的时候降级同步到异步。

5、多级降级

缓存是离用户最近越高效；而降级是离用户越近越能对系统保护的好。因为业务的复杂性导致越到后端QPS/TPS越低。

页面JS降级开关：主要控制页面功能的降级，在页面中通过JS脚本部署功能降级开关，在适当时机开启/关闭开关；

接入层降级开关：主要控制请求入口的降级，请求进入后会首先进入接入层，在接入层可以配置功能降级开关，可以根据实际情况进行自动/人工降级；

尤其在后端应用服务出问题时，通过接入层降级从而给应用服务有足够的时间恢复服务；

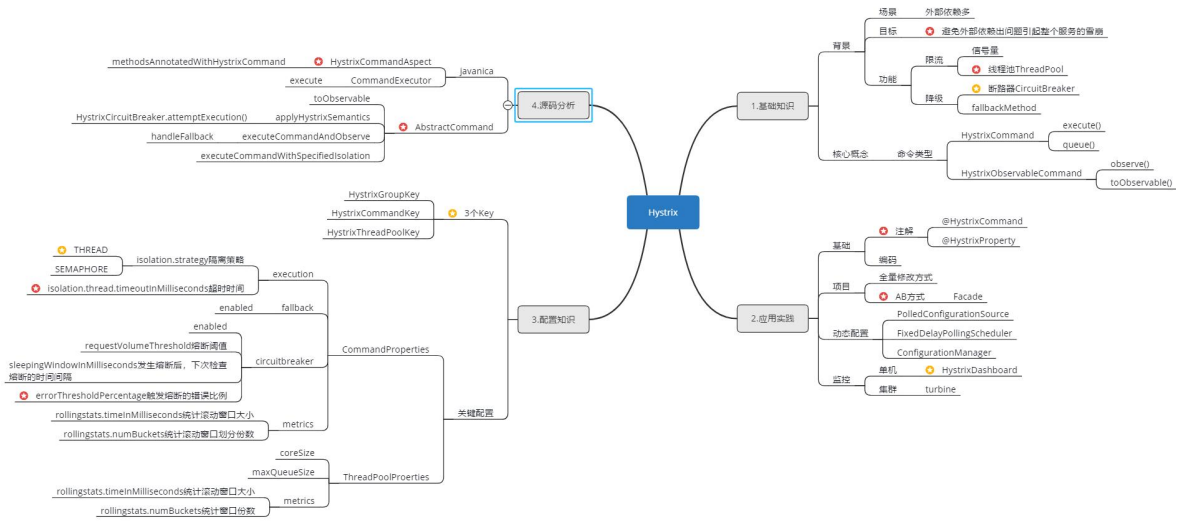
应用层降级开关：主要控制业务的降级，在应用中配置相应的功能开关，根据实际业务情况进行自动/人工降级。

总结：

降级能保障系统在大促中活下来，而不是死去，达到丢卒保帅的作用。对用户提供有损服务，总比不服务要好。根据自己的场景设计相应的降级策略，保障系统在危机时刻能通过降级手段平稳度过。

分布式降级框架

Hystrix



初探Hystrix

Hystrix [hɪst'rɪks]，中文含义是豪猪，因其背上长满棘刺，从而拥有了自我保护的能力。本文所说的Hystrix是Netflix开源的一款容错框架，同样具有自我保护能力。为了实现容错和自我保护，下面我们看看Hystrix如何设计和实现的。

Hystrix设计目标：

- 对来自依赖的延迟和故障进行防护和控制——这些依赖通常都是通过网络访问的
- 阻止故障的连锁反应
- 快速失败并迅速恢复
- 回退并优雅降级
- 提供近实时的监控与告警

Hystrix遵循的设计原则：

- 防止任何单独的依赖耗尽资源（线程）
- 过载立即切断并快速失败，防止排队
- 尽可能提供回退以保护用户免受故障
- 使用隔离技术（例如隔板，泳道和断路器模式）来限制任何一个依赖的影响
- 通过近实时的指标，监控和告警，确保故障被及时发现
- 通过动态修改配置属性，确保故障及时恢复
- 防止整个依赖客户端执行失败，而不仅仅是网络通信

Hystrix如何实现这些设计目标?

- 使用命令模式将所有对外部服务（或依赖关系）的调用包装在HystrixCommand或HystrixObservableCommand对象中，并将该对象放在单独的线程中执行；
- 每个依赖都维护着一个线程池（或信号量），线程池被耗尽则拒绝请求（而不是让请求排队）。
- 记录请求成功，失败，超时和线程拒绝。
- 服务错误百分比超过了阈值，熔断器开关自动打开，一段时间内停止对该服务的所有请求。
- 请求失败，被拒绝，超时或熔断时执行降级逻辑。
- 近实时地监控指标和配置的修改。

Hystrix入门

Hystrix简单示例

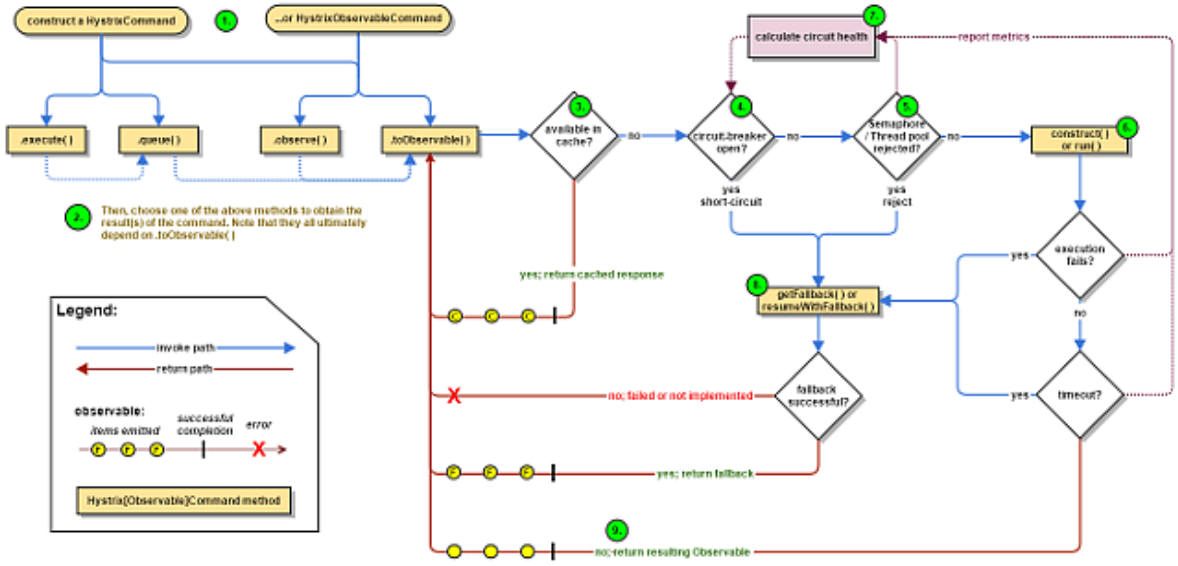
开始深入Hystrix原理之前，我们先简单看一个示例。

第一步，继承HystrixCommand实现自己的command，在command的构造方法中需要配置请求被执行需要的参数，并组合实际发送请求的对象，代码如下：

```
public class QueryOrderIdCommand extends HystrixCommand<Integer> {
    private final static Logger logger =
LoggerFactory.getLogger(QueryOrderIdCommand.class);
    private OrderServiceProvider orderServiceProvider;

    public QueryOrderIdCommand(OrderServiceProvider orderServiceProvider) {

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

.andCommandKey(HystrixCommandKey.Factory.asKey("queryByOrderId"))
                .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                        .withCircuitBreakerRequestVolumeThreshold(10)//至少有10个
请求，熔断器才进行错误率的计算
                        .withCircuitBreakerSleepWindowInMilliseconds(5000)//熔断
器中断请求5秒后会进入半打开状态,放部分流量过去重试
                        .withCircuitBreakerErrorThresholdPercentage(50)//错误率达
到50开启熔断保护
                        .withExecutionTimeoutEnabled(true))
                .andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties
                        .Setter().withCoreSize(10)));
        this.orderServiceProvider = orderServiceProvider;
    }

    @Override
    protected Integer run() {
        return orderServiceProvider.queryByOrderId();
    }

    @Override
    protected Integer getFallback() {
        return -1;
    }
}
```

第二步，调用HystrixCommand的执行方法发起实际请求。

```
```
@Test
public void testQueryByOrderIdCommand() {
    Integer r = new QueryOrderIdCommand(orderServiceProvider).execute();
    logger.info("result:{}", r);
}
```

# Hystrix处理流程

Hystrix流程图如下：
```



Hystrix整个工作流如下：

1. 构造一个 HystrixCommand或HystrixObservableCommand对象，用于封装请求，并在构造方法配置请求被执行需要的参数；
2. 执行命令，Hystrix提供了4种执行命令的方法，后面详述；
3. 判断是否使用缓存响应请求，若启用了缓存，且缓存可用，直接使用缓存响应请求。Hystrix支持请求缓存，但需要用户自定义启动；
4. 判断熔断器是否打开，如果打开，跳到第8步；
5. 判断线程池/队列/信号量是否已满，已满则跳到第8步；
6. 执行HystrixObservableCommand.construct()或HystrixCommand.run()，如果执行失败或者超时，跳到第8步；否则，跳到第9步；
7. 统计熔断器监控指标；
8. 走Fallback备用逻辑
9. 返回请求响应

从流程图上可知道，第5步线程池/队列/信号量已满时，还会执行第7步逻辑，更新熔断器统计信息，而第6步无论成功与否，都会更新熔断器统计信息。

执行命令的几种方法

Hystrix提供了4种执行命令的方法，execute()和queue() 适用于HystrixCommand对象，而observe()和toObservable()适用于HystrixObservableCommand对象。

execute()

以同步堵塞方式执行run()，只支持接收一个值对象。hystrix会从线程池中取一个线程来执行run()，并等待返回值。

queue()

以异步非阻塞方式执行run()，只支持接收一个值对象。调用queue()就直接返回一个Future对象。可通过 Future.get()拿到run()的返回结果，但Future.get()是阻塞执行的。若执行成功，Future.get()返回单个返回值。当执行失败时，如果没有重写fallback，Future.get()抛出异常。

observe()

事件注册前执行run()/construct()，支持接收多个值对象，取决于发射源。调用observe()会返回一个hot Observable，也就是说，调用observe()自动触发执行run()/construct()，无论是否存在订阅者。

如果继承的是HystrixCommand，hystrix会从线程池中取一个线程以非阻塞方式执行run()；如果继承的是HystrixObservableCommand，将以调用线程阻塞执行construct()。

observe()使用方法：

1. 调用observe()会返回一个Observable对象
2. 调用这个Observable对象的subscribe()方法完成事件注册，从而获取结果

toObservable()

事件注册后执行run()/construct()，支持接收多个值对象，取决于发射源。调用toObservable()会返回一个cold Observable，也就是说，调用toObservable()不会立即触发执行run()/construct()，必须有订阅者订阅Observable时才会执行。

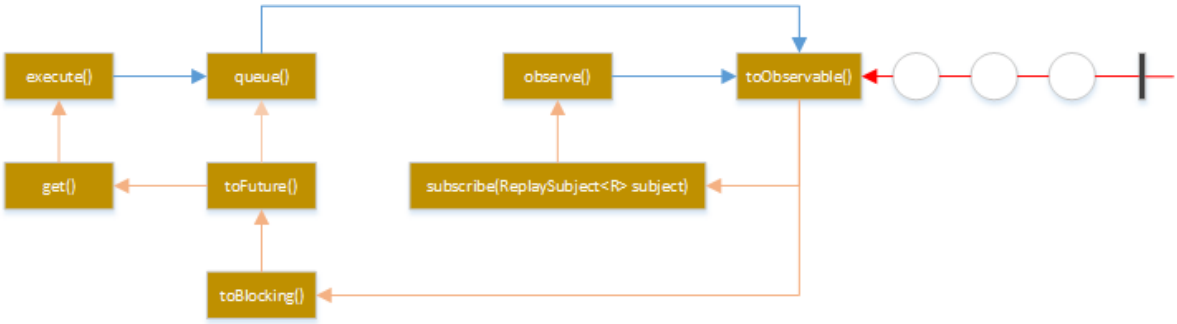
如果继承的是HystrixCommand，hystrix会从线程池中取一个线程以非阻塞方式执行run()，调用线程不必等待run()；如果继承的是HystrixObservableCommand，将以调用线程堵塞执行construct()，调用线程需等待construct()执行完才能继续往下走。

toObservable()使用方法：

1. 调用observe()会返回一个Observable对象
2. 调用这个Observable对象的subscribe()方法完成事件注册，从而获取结果

需注意的是，HystrixCommand也支持toObservable()和observe()，但是即使将HystrixCommand转换成Observable，它也只能发射一个值对象。只有HystrixObservableCommand才支持发射多个值对象。

几种方法的关系



- execute()实际是调用了queue().get()
- queue()实际调用了toObservable().toBlocking().toFuture()
- observe()实际调用toObservable()获得一个cold Observable，再创建一个ReplaySubject对象订阅Observable，将源Observable转化为hot Observable。因此调用observe()会自动触发执行run()/construct()。

Hystrix总是以Observable的形式作为响应返回，不同执行命令的方法只是进行了相应的转换。

Hystrix容错

Hystrix的容错主要是通过添加容许延迟和容错方法，帮助控制这些分布式服务之间的交互。 还通过隔离服务之间的访问点，阻止它们之间的级联故障以及提供回退选项来实现这一点，从而提高系统的整体弹性。Hystrix主要提供了以下几种容错方法：

- 资源隔离
- 熔断
- 降级

下面我们详细谈谈这几种容错机制。

资源隔离

资源隔离主要指对线程的隔离。Hystrix提供了两种线程隔离方式：线程池和信号量。

线程隔离-线程池

通过将发送请求线程与执行请求的线程分离，可有效防止发生级联故障。当线程池或请求队列饱和时，Hystrix将拒绝服务，使得请求线程可以快速失败，从而避免依赖问题扩散。

线程池隔离优缺点

优点：

- 保护应用程序以免受来自依赖故障的影响，指定依赖线程池饱和不会影响应用程序的其余部分。
- 当引入新客户端lib时，即使发生问题，也是在本lib中，并不会影响到其他内容。
- 当依赖从故障恢复正常时，应用程序会立即恢复正常的性能。
- 当应用程序一些配置参数错误时，线程池的运行状况会很快检测到这一点（通过增加错误，延迟，超时，拒绝等），同时可以通过动态属性进行实时纠正错误的参数配置。
- 如果服务的性能有变化，需要实时调整，比如增加或者减少超时时间，更改重试次数，可以通过线程池指标动态属性修改，而且不会影响到其他调用请求。
- 除了隔离优势外，hystrix拥有专门的线程池可提供内置的并发功能，使得可以在同步调用之上构建异步门面（外观模式），为异步编程提供了支持（Hystrix引入了Rxjava异步框架）。

注意：尽管线程池提供了线程隔离，我们的客户端底层代码也必须要有超时设置或响应线程中断，不能无限制的阻塞以致线程池一直饱和。

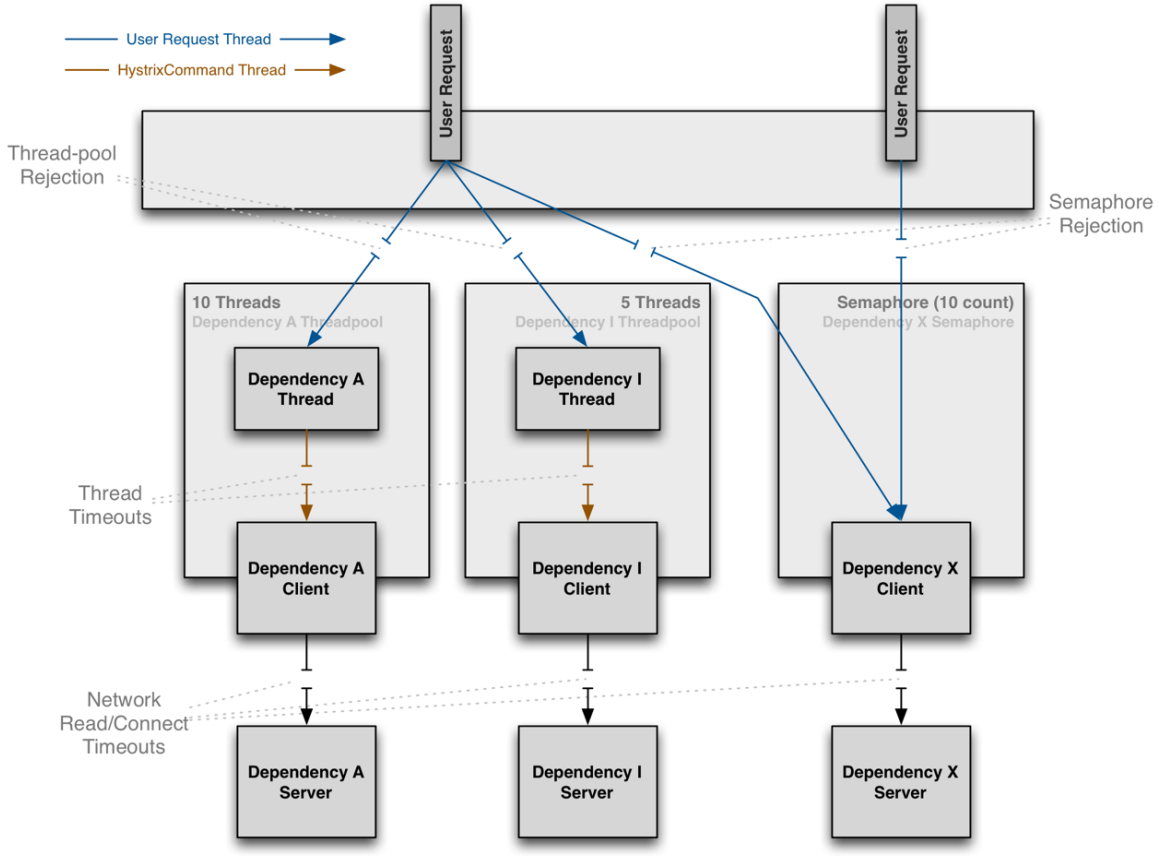
缺点：

线程池的主要缺点是增加了计算开销。每个命令的执行都在单独的线程完成，增加了排队、调度和上下文切换的开销。因此，要使用Hystrix，就必须接受它带来的开销，以换取它所提供的好处。

通常情况下，线程池引入的开销足够小，不会有重大的成本或性能影响。但对于一些访问延迟极低的服务，如只依赖内存缓存，线程池引入的开销就比较明显了，这时候使用线程池隔离技术就不适合了，我们需要考虑更轻量级的方式，如信号量隔离。

线程隔离-信号量

上面提到了线程池隔离的缺点，当依赖延迟极低的服务时，线程池隔离技术引入的开销超过了它所带来的好处。这时候可以使用信号量隔离技术来代替，通过设置信号量来限制对任何给定依赖的并发调用量。下图说明了线程池隔离和信号量隔离的主要区别：



使用线程池时，发送请求的线程和执行依赖服务的线程不是同一个，而使用信号量时，发送请求的线程和执行依赖服务的线程是同一个，都是发起请求的线程。先看一个使用信号量隔离线程的示例：

```
public class QueryByOrderIdCommandSemaphore extends HystrixCommand<Integer> {
    private final static Logger logger =
LoggerFactory.getLogger(QueryByOrderIdCommandSemaphore.class);
    private OrderServiceProvider orderServiceProvider;

    public QueryByOrderIdCommandSemaphore(OrderServiceProvider
orderServiceProvider) {

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

.andCommandKey(HystrixCommandKey.Factory.asKey("queryByOrderId"))
                .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                        .withCircuitBreakerRequestVolumeThreshold(10)////至少有10
个请求，熔断器才进行错误率的计算
                        .withCircuitBreakerSleepWindowInMilliseconds(5000)//熔断
器中断请求5秒后会进入半打开状态,放部分流量过去重试
                        .withCircuitBreakerErrorThresholdPercentage(50)//错误率达
到50开启熔断保护

.withExecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolationStrat
egy.SEMAPHORE)

.withExecutionIsolationSemaphoreMaxConcurrentRequests(10)));//最大并发请求量
        this.orderServiceProvider = orderServiceProvider;
    }

    @Override
```

```
    protected Integer run() {
        return orderServiceProvider.queryByOrderId();
    }

    @Override
    protected Integer getFallback() {
        return -1;
    }
}
```

由于Hystrix默认使用线程池做线程隔离，使用信号量隔离需要显示地将属性execution.isolation.strategy设置为ExecutionIsolationStrategy.SEMAPHORE，同时配置信号量个数，默认为10。客户端需向依赖服务发起请求时，首先要获取一个信号量才能真正发起调用，由于信号量的数量有限，当并发请求量超过信号量个数时，后续的请求都会直接拒绝，进入fallback流程。

信号量隔离主要是通过控制并发请求量，防止请求线程大面积阻塞，从而达到限流和防止雪崩的目的。

线程隔离总结

线程池和信号量都可以做线程隔离，但各有各的优缺点和支持的场景，对比如下：

	线程切换	支持异步	支持超时	支持熔断	限流	开销
信号量	否	否	否	是	是	小
线程池	是	是	是	是	是	大

线程池和信号量都支持熔断和限流。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，也就是说当所请求的服务不可用时，信号量会控制超过限制的请求立即返回，但是已经持有信号量的线程只能等待服务响应或从超时中返回，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。

熔断

熔断器简介

现实生活中，可能大家都有注意到家庭电路中通常会安装一个保险盒，当负载过载时，保险盒中的保险丝会自动熔断，以保护电路及家里的各种电器，这就是熔断器的一个常见例子。Hystrix中的熔断器(Circuit Breaker)也是起类似作用，Hystrix在运行过程中会向每个commandKey对应的熔断器报告成功、失败、超时和拒绝的状态，熔断器维护并统计这些数据，并根据这些统计信息来决策熔断开关是否打开。如果打开，熔断后续请求，快速返回。隔一段时间（默认是5s）之后熔断器尝试半开，放入一部分流量请求进来，相当于对依赖服务进行一次健康检查，如果请求成功，熔断器关闭。

熔断器配置

Circuit Breaker主要包括如下6个参数：

1、circuitBreaker.enabled

是否启用熔断器，默认是TRUE。
2 、circuitBreaker.forceOpen

熔断器强制打开，始终保持打开状态，不关注熔断开关的实际状态。默认值FLASE。
3、circuitBreaker.forceClosed
熔断器强制关闭，始终保持关闭状态，不关注熔断开关的实际状态。默认值FLASE。

4、circuitBreaker.errorThresholdPercentage
错误率，默认值50%，例如一段时间（10s）内有100个请求，其中有54个超时或者异常，那么这段时间内的错误率是54%，大于了默认值50%，这种情况下会触发熔断器打开。

5、circuitBreaker.requestVolumeThreshold

默认值20。含义是一段时间内至少有20个请求才进行errorThresholdPercentage计算。比如一段时间了有19个请求，且这些请求全部失败了，错误率是100%，但熔断器不会打开，总请求数不满足20。

6、circuitBreaker.sleepWindowInMilliseconds

半开状态试探睡眠时间，默认值5000ms。如：当熔断器开启5000ms之后，会尝试放过去一部分流量进行试探，确定依赖服务是否恢复。

回退降级

降级，通常指务高峰期，为了保证核心服务正常运行，需要停掉一些不太重要的业务，或者某些服务不可用时，执行备用逻辑从故障服务中快速失败或快速返回，以保障主体业务不受影响。Hystrix提供的降级主要是为了容错，保证当前服务不受依赖服务故障的影响，从而提高服务的健壮性。要支持回退或降级处理，可以重写HystrixCommand的getFallBack方法或HystrixObservableCommand的resumeWithFallback方法。

Hystrix在以下几种情况下会走降级逻辑：

- 执行construct()或run()抛出异常
- 熔断器打开导致命令短路
- 命令的线程池和队列或信号量的容量超额，命令被拒绝
- 命令执行超时

总结

本文介绍了Hystrix及其工作原理，还介绍了Hystrix线程池隔离、信号量隔离和熔断器的工作原理，以及如何使用Hystrix的资源隔离，熔断和降级等技术实现服务容错，从而提高系统的整体健壮性。

Sentinel 简介

Sentinel 可以简单的分为 Sentinel 核心库和 Dashboard。核心库不依赖 Dashboard，但是结合 Dashboard 可以取得最好的效果。

这篇文章主要介绍 Sentinel 核心库的使用。如果希望有一个最快最直接的了解，可以参考 新手指南 来获取一个最直观的感受。

我们说的资源，可以是任何东西，服务，服务里的方法，甚至是一段代码。使用 Sentinel 来进行资源保护，主要分为几个步骤:

1. 定义资源
2. 定义规则
3. 检验规则是否生效

先把可能需要保护的资源定义好，之后再配置规则。也可以理解为，只要有了资源，我们就可以在任何时候灵活地定义各种流量控制规则。在编码的时候，只需要考虑这个代码是否需要保护，如果需要保护，就将之定义为一个资源。

对于主流的框架，我们提供适配，只需要按照适配中的说明配置，Sentinel 就会默认定义提供的服务，方法等为资源。

定义资源

方式一：主流框架的默认适配

为了减少开发的复杂程度，我们对大部分的主流框架，例如 Web Servlet、Dubbo、Spring Cloud、gRPC、Spring WebFlux、Reactor 等都做了适配。您只需要引入对应的依赖即可方便地整合 Sentinel。可以参见: 主流框架的适配。

方式二：抛出异常的方式定义资源

`SphU` 包含了 try-catch 风格的 API。用这种方式，当资源发生了限流之后会抛出 `BlockException`。这个时候可以捕捉异常，进行限流之后的逻辑处理。示例代码如下:

```
// 1.5.0 版本开始可以利用 try-with-resources 特性（使用有限制）
// 资源名可使用任意有业务语义的字符串，比如方法名、接口名或其它可唯一标识的字符串。
try (Entry entry = SphU.entry("resourceName")) {
  // 被保护的业务逻辑
  // do something here...
} catch (BlockException ex) {
  // 资源访问阻止，被限流或被降级
  // 在此处进行相应的处理操作
}
```

特别地，若 entry 的时候传入了热点参数，那么 exit 的时候也一定要带上对应的参数（`exit(count, args)`），否则可能会有统计错误。这个时候不能使用 try-with-resources 的方式。另外通过 `Tracer.trace(ex)` 来统计异常信息时，由于 try-with-resources 语法中 catch 调用顺序的问题，会导致无法正确统计异常数，因此统计异常信息时也不能在 try-with-resources 的 catch 块中调用 `Tracer.trace(ex)`。

手动 exit 示例:

```
Entry entry = null;
// 务必保证 finally 会被执行
try {
  // 资源名可使用任意有业务语义的字符串，注意数目不能太多（超过 1K），超出几千请作为参数传入而不要直接作为资源名
  // EntryType 代表流量类型（inbound/outbound），其中系统规则只对 IN 类型的埋点生效
  entry = SphU.entry("自定义资源名");
  // 被保护的业务逻辑
  // do something...
} catch (BlockException ex) {
  // 资源访问阻止，被限流或被降级
  // 进行相应的处理操作
} catch (Exception ex) {
  // 若需要配置降级规则，需要通过这种方式记录业务异常
  Tracer.traceEntry(ex, entry);
} finally {
  // 务必保证 exit，务必保证每个 entry 与 exit 配对
  if (entry != null) {
    entry.exit();
  }
}
```

热点参数埋点示例：

```
Entry entry = null;
try {
    // 若需要配置例外项，则传入的参数只支持基本类型。
    // EntryType 代表流量类型，其中系统规则只对 IN 类型的埋点生效
    // count 大多数情况都填 1，代表统计为一次调用。
    entry = SphU.entry(resourceName, EntryType.IN, 1, paramA, paramB);
    // Your logic here.
} catch (BlockException ex) {
    // Handle request rejection.
} finally {
    // 注意：exit 的时候也一定要带上对应的参数，否则可能会有统计错误。
    if (entry != null) {
        entry.exit(1, paramA, paramB);
    }
}
```

`SphU.entry()` 的参数描述：

参数名	类型	解释	默认值
entryType	`EntryType`	资源调用的流量类型，是入口流量（`EntryType.IN`）还是出口流量（`EntryType.OUT`），注意系统规则只对 IN 生效	`EntryType.OUT`
count	`int`	本次资源调用请求的 token 数目	1
args	`Object[]`	传入的参数，用于热点参数限流	无

注意：`SphU.entry(xxx)` 需要与 `entry.exit()` 方法成对出现，匹配调用，否则会导致调用链记录异常，抛出 `ErrorEntryFreeException` 异常。常见的错误：

- 自定义埋点只调用 `SphU.entry()`，没有调用 `entry.exit()`
- 顺序错误，比如：`entry1 -> entry2 -> exit1 -> exit2`，应该为 `entry1 -> entry2 -> exit2 -> exit1`

方式三：返回布尔值方式定义资源

`SphO` 提供 if-else 风格的 API。用这种方式，当资源发生了限流之后会返回 `false`，这个时候可以根据返回值，进行限流之后的逻辑处理。示例代码如下：

```
  // 资源名可使用任意有业务语义的字符串
  if (SphO.entry("自定义资源名")) {
    // 务必保证finally会被执行
    try {
      /**
      * 被保护的业务逻辑
      */
    } finally {
      SphO.exit();
    }
  } else {
```

```
    // 资源访问阻止，被限流或被降级
    // 进行相应的处理操作
  }
```

注意：`SphO.entry(xxx)` 需要与 SphO.exit() 方法成对出现，匹配调用，位置正确，否则会导致调用链记录异常，抛出 ErrorEntryFreeException` 异常。

方式四：注解方式定义资源

Sentinel 支持通过 `@SentinelResource` 注解定义资源并配置 `blockHandler` 和 `fallback` 函数来进行限流之后的处理。示例：

```
// 原本的业务方法.
@SentinelResource(blockHandler = "blockHandlerForGetUser")
public User getUserById(String id) {
    throw new RuntimeException("getUserById command failed");
}

// blockHandler 函数，原方法调用被限流/降级/系统保护的时候调用
public User blockHandlerForGetUser(String id, BlockException ex) {
    return new User("admin");
}
```

注意 `blockHandler` 函数会在原方法被限流/降级/系统保护的时候调用，而 `fallback` 函数会针对所有类型的异常。请注意 `blockHandler` 和 `fallback` 函数的形式要求，更多指引可以参见 Sentinel 注解支持文档。

方式五：异步调用支持

Sentinel 支持异步调用链路的统计。在异步调用中，需要通过 `SphU.asyncEntry(xxx)` 方法定义资源，并通常需要在异步的回调函数中调用 `exit` 方法。以下是一个简单的示例：

```
try {
    AsyncEntry entry = SphU.asyncEntry(resourceName);

    // 异步调用.
    doAsync(userId, result -> {
        try {
            // 在此处处理异步调用的结果.
        } finally {
            // 在回调结束后 exit.
            entry.exit();
        }
    });
} catch (BlockException ex) {
    // Request blocked.
    // Handle the exception (e.g. retry or fallback).
}
```

`SphU.asyncEntry(xxx)` 不会影响当前（调用线程）的 Context，因此以下两个 entry 在调用链上是平级关系（处于同一层），而不是嵌套关系：

```
// 调用链类似于：
// -parent
// ---asyncResource
// ---syncResource
asyncEntry = SphU.asyncEntry(asyncResource);
entry = SphU.entry(normalResource);
```

若在异步回调中需要嵌套其它的资源调用（无论是 `entry` 还是 `asyncEntry`），只需要借助 Sentinel 提供的上下文切换功能，在对应的地方通过 `ContextUtil.runOnContext(context, f)` 进行 Context 变换，将对应资源调用处的 Context 切换为生成的异步 Context，即可维持正确的调用链路关系。示例如下：

```
public void handleResult(String result) {
    Entry entry = null;
    try {
        entry = SphU.entry("handleResultForAsync");
        // Handle your result here.
    } catch (BlockException ex) {
        // Blocked for the result handler.
    } finally {
        if (entry != null) {
            entry.exit();
        }
    }
}

public void someAsync() {
    try {
        AsyncEntry entry = SphU.asyncEntry(resourceName);

        // Asynchronous invocation.
        doAsync(userId, result -> {
            // 在异步回调中进行上下文变换，通过 AsyncEntry 的 getAsyncContext 方法获取
异步 Context
            ContextUtil.runOnContext(entry.getAsyncContext(), () -> {
                try {
                    // 此处嵌套正常的资源调用.
                    handleResult(result);
                } finally {
                    entry.exit();
                }
            });
        });
    } catch (BlockException ex) {
        // Request blocked.
        // Handle the exception (e.g. retry or fallback).
    }
}
```

此时的调用链就类似于：

```
-parent
---asyncInvocation
-----handleResultForAsync
```

更详细的示例可以参考 Demo 中的 AsyncEntryDemo，里面包含了普通资源与异步资源之间的各种嵌套示例。

规则的种类

Sentinel 的所有规则都可以在内存态中动态地查询及修改，修改之后立即生效。同时 Sentinel 也提供相关 API，供您来定制自己的规则策略。

Sentinel 支持以下几种规则：**流量控制规则**、**熔断降级规则**、**系统保护规则**、**来源访问控制规则** 和 **热点参数规则**。

流量控制规则 (FlowRule)

流量规则的定义

重要属性：

Field	说明	默认值
resource	资源名，资源名是限流规则的作用对象	
count	限流阈值	
grade	限流阈值类型，QPS 模式（1）或并发线程数模式（0）	QPS 模式
limitApp	流控针对的调用来源	`default`，代表不区分调用来源
strategy	调用关系限流策略：直接、链路、关联	根据资源本身（直接）
controlBehavior	流控效果（直接拒绝 / 排队等待 / 慢启动模式），不支持按调用关系限流	直接拒绝
clusterMode	是否集群限流	否

同一个资源可以同时有多个限流规则，检查规则时会依次检查。

通过代码定义流量控制规则

理解上面规则的定义之后，我们可以通过调用 `FlowRuleManager.loadRules()` 方法来用硬编码的方式定义流量控制规则，比如：

```
private void initFlowQpsRule() {
    List<FlowRule> rules = new ArrayList<>();
    FlowRule rule = new FlowRule(resourceName);
    // set limit qps to 20
    rule.setCount(20);
    rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
    rule.setLimitApp("default");
    rules.add(rule);
    FlowRuleManager.loadRules(rules);
}
```

更多详细内容可以参考 流量控制。

熔断降级规则 (DegradeRule)

熔断降级规则包含下面几个重要的属性：

Field	说明	默认值
resource	资源名，即限流规则的作用对象	
count	阈值	
grade	熔断策略，支持秒级 RT/秒级异常比例/分钟级异常数	秒级平均 RT
timeWindow	降级的时间，单位为 s	
rtSlowRequestAmount	RT 模式下 1 秒内连续多少个请求的平均 RT 超出阈值方可触发熔断（1.7.0 引入）	5
minRequestAmount	异常熔断的触发最小请求数，请求数小于该值时即使异常比率超出阈值也不会熔断（1.7.0 引入）	5

同一个资源可以同时有多个降级规则。

理解上面规则的定义之后，我们可以通过调用 `DegradeRuleManager.loadRules()` 方法来用硬编码的方式定义流量控制规则。

```
private void initDegradeRule() {
    List<DegradeRule> rules = new ArrayList<>();
    DegradeRule rule = new DegradeRule();
    rule.setResource(KEY);
    // set threshold RT, 10 ms
    rule.setCount(10);
    rule.setGrade(RuleConstant.DEGRADE_GRADE_RT);
    rule.setTimeWindow(10);
    rules.add(rule);
    DegradeRuleManager.loadRules(rules);
}
```

更多详情可以参考 熔断降级。

系统保护规则 (SystemRule)

Sentinel 系统自适应限流从整体维度对应用入口流量进行控制，结合应用的 Load、CPU 使用率、总体平均 RT、入口 QPS 和并发线程数等几个维度的监控指标，通过自适应的流控策略，让系统的入口流量和系统的负载达到一个平衡，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

系统规则包含下面几个重要的属性：

Field	说明	默认值
highestSystemLoad	`load1` 触发值，用于触发自适应控制阶段	-1 (不生效)
avgRt	所有入口流量的平均响应时间	-1 (不生效)
maxThread	入口流量的最大并发数	-1 (不生效)
qps	所有入口资源的 QPS	-1 (不生效)
highestCpuUsage	当前系统的 CPU 使用率（0.0-1.0）	-1 (不生效)

理解上面规则的定义之后，我们可以通过调用 `SystemRuleManager.loadRules()` 方法来用硬编码的方式定义流量控制规则。

```
private void initSystemRule() {
    List<SystemRule> rules = new ArrayList<>();
    SystemRule rule = new SystemRule();
    rule.setHighestSystemLoad(10);
    rules.add(rule);
    SystemRuleManager.loadRules(rules);
}
```

注意系统规则只针对入口资源（EntryType=IN）生效。更多详情可以参考 系统自适应保护文档。

访问控制规则 (AuthorityRule)

很多时候，我们需要根据调用方来限制资源是否通过，这时候可以使用 Sentinel 的访问控制（黑白名单）的功能。黑白名单根据资源的请求来源（`origin`）限制资源是否通过，若配置白名单则只有请求来源位于白名单内时才可通过；若配置黑名单则请求来源位于黑名单时不通过，其余的请求通过。

授权规则，即黑白名单规则（`AuthorityRule`）非常简单，主要有以下配置项：

- `resource`：资源名，即限流规则的作用对象
- `limitApp`：对应的黑名单/白名单，不同 origin 用 `,` 分隔，如 `appA,appB`
- `strategy`：限制模式，`AUTHORITY_WHITE` 为白名单模式，`AUTHORITY_BLACK` 为黑名单模式，默认为白名单模式

更多详情可以参考 来源访问控制。

热点规则 (ParamFlowRule)

详情可以参考 热点参数限流。

查询更改规则

引入了 transport 模块后，可以通过以下的 HTTP API 来获取所有已加载的规则：

```
http://localhost:8719/getRules?type=<XXXX>
```

其中，`type=flow` 以 JSON 格式返回现有的限流规则，degrade 返回现有生效的降级规则列表，system 则返回系统保护规则。

获取所有热点规则：

```
http://localhost:8719/getParamRules
```

定制自己的持久化规则

上面的规则配置，都是存在内存中的。即如果应用重启，这个规则就会失效。因此我们提供了开放的接口，您可以通过实现 `DataSource` 接口的方式，来自定义规则的存储数据源。通常我们的建议有：

- 整合动态配置系统，如 ZooKeeper、Nacos、Apollo 等，动态地实时刷新配置规则
- 结合 RDBMS、NoSQL、VCS 等来实现该规则
- 配合 Sentinel Dashboard 使用

更多详情请参考 动态规则配置。

规则生效的效果

判断限流降级异常

在 Sentinel 中所有流控降级相关的异常都是异常类 `BlockException` 的子类：

- 流控异常：`FlowException`
- 熔断降级异常：`DegradeException`
- 系统保护异常：`SystemBlockException`
- 热点参数限流异常：`ParamFlowException`

我们可以通过以下函数判断是否为 Sentinel 的流控降级异常：

```
BlockException.isBlockException(Throwable t);
```

除了在业务代码逻辑上看到规则生效，我们也可以通过下面简单的方法，来校验规则生效的效果：

- **暴露的 HTTP 接口**：通过运行下面命令 `curl http://localhost:8719/cnode?id=<资源名称>`，观察返回的数据。如果规则生效，在返回的数据栏中的 `block` 以及 `block(m)` 中会有显示
- **日志**：Sentinel 提供秒级的资源运行日志以及限流日志，详情可以参考: 日志

block 事件

Sentinel 提供以下扩展接口，可以通过 `StatisticSlotCallbackRegistry` 向 `StatisticSlot` 注册回调函数：

- `ProcessorSlotEntryCallback`: callback when resource entry passed (`onPass`) or blocked (`onBlocked`)
- `ProcessorSlotExitCallback`: callback when resource entry successfully completed (`onExit`)

可以利用这些回调接口来实现报警等功能，实时的监控信息可以从 `ClusterNode` 中实时获取。

其它 API

业务异常统计 Tracer

业务异常记录类 `Tracer` 用于记录业务异常。相关方法：

- `trace(Throwable e)`：记录业务异常（非 `BlockException` 异常），对应的资源为当前线程 context 下 entry 对应的资源。该方法必须在 `SphU.entry(xxx)` 成功之后且 exit 之前调用，否则

当前 context 为空则会抛出异常。

- `trace(Throwable e, int count)`：记录业务异常（非 `BlockException` 异常），异常数目为传入的 `count`。该方法必须在 `SphU.entry(xxx)` 成功之后且 exit 之前调用，否则当前 context 为空则会抛出异常。
- `traceEntry(Throwable, int, Entry)`：向传入 entry 对应的资源记录业务异常（非 `BlockException` 异常），异常数目为传入的 `count`。

如果用户通过 `SphU` 或 `SphO` 手动定义资源，则 Sentinel 不能感知上层业务的异常，需要手动调用 `Tracer.trace(ex)` 来记录业务异常，否则对应的异常不会统计到 Sentinel 异常计数中。注意不要在 try-with-resources 形式的 `SphU.entry(xxx)` 中使用，否则会统计不上。

从 1.3.1 版本开始，注解方式定义资源支持自动统计业务异常，无需手动调用 `Tracer.trace(ex)` 来记录业务异常。Sentinel 1.3.1 以前的版本需要手动记录。

上下文工具类 ContextUtil

相关静态方法：

标识进入调用链入口（上下文）：

以下静态方法用于标识调用链路入口，用于区分不同的调用链路：

- `public static Context enter(String contextName)`
- `public static Context enter(String contextName, String origin)`

其中 `contextName` 代表调用链路入口名称（上下文名称），`origin` 代表调用来源名称。默认调用来源为空。返回值类型为 `Context`，即生成的调用链路上下文对象。

注意：`ContextUtil.enter(xxx)` 方法仅在调用链路入口处生效，即仅在当前线程的初次调用生效，后面再调用不会覆盖当前线程的调用链路，直到 exit。`Context` 存于 ThreadLocal 中，因此切换线程时可能会丢掉，如果需要跨线程使用可以结合 `runOnContext` 方法使用。

流控规则中若选择“流控方式”为“链路”方式，则入口资源名即为上面的 `contextName`。

退出调用链（清空上下文）：

- `public static void exit()`：该方法用于退出调用链，清理当前线程的上下文。

获取当前线程的调用链上下文：

- `public static Context getContext()`：获取当前线程的调用链路上下文对象。

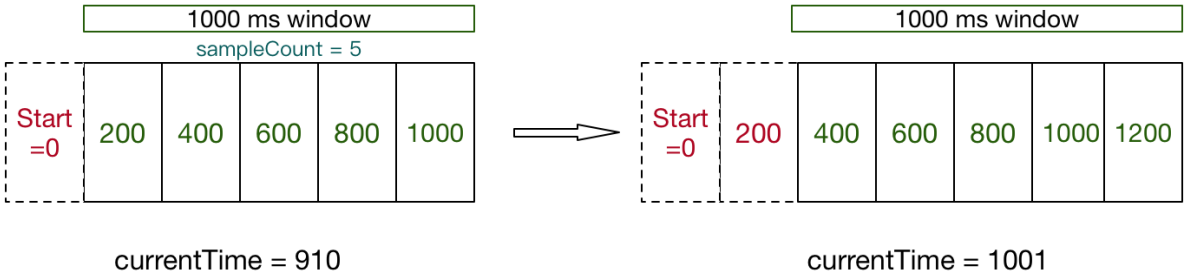
在某个调用链上下文中执行代码：

- `public static void runOnContext(Context context, Runnable f)`：常用于异步调用链路中 context 的变换。

指标统计配置

Sentinel 底层采用高性能的滑动窗口数据结构来统计实时的秒级指标数据，并支持对滑动窗口进行配置。主要有以下两个配置：

- `windowIntervalMs`：滑动窗口的总的时间长度，默认为 1000 ms
- `sampleCount`：滑动窗口划分的格子数目，默认为 2；格子越多则精度越高，但是内存占用也会越多



我们可以通过 `SampleCountProperty` 来动态地变更滑动窗口的格子数目，通过 `IntervalProperty` 来动态地变更滑动窗口的总时间长度。注意这两个配置都是**全局生效**的，会影响所有资源的所有指标统计。

功能对比:

功能	Sentinel	Hystrix	resilience4j
隔离策略	信号量隔离（并发线程数限流）	线程池隔离/信号量隔离	信号量隔离
熔断降级策略	基于响应时间、异常比率、异常数	基于异常比率	基于异常比率、响应时间
实时统计实现	滑动窗口（LeapArray）	滑动窗口（基于 RxJava）	Ring Bit Buffer
动态规则配置	支持多种数据源	支持多种数据源	有限支持
扩展性	多个扩展点	插件的形式	接口的形式
基于注解的支持	支持	支持	支持
限流	基于 QPS，支持基于调用关系的限流	有限的支持	Rate Limiter
流量整形	支持预热模式、匀速器模式、预热排队模式(流量规则处可配置)	不支持	简单的 Rate Limiter 模式
系统自适应保护	支持	不支持	不支持
控制台	提供开箱即用的控制台，可配置规则、查看秒级监控、机器发现等	简单的监控查看	不提供控制台，可对接其它监控系统