

深度了解Sentinel的流量控制

在Sentinel中，限流的直接表现形式是，在执行 `Entry nodeA = SphU.entry(resourceName)` 的时候抛出 `FlowException` 异常。`FlowException` 是 `BlockException` 的子类，您可以捕捉 `BlockException` 来自定义被限流之后的处理逻辑。

并且，对于同一个资源或者不同资源可以分别创建多条限流规则，`FlowSlot`会对该资源的所有限流规则依次遍历，直到有规则触发限流或者所有规则遍历完毕。

一条限流规则主要由下面几个因素组成，我们可以组合这些元素来实现不同的限流效果：

- `resource`：资源名，即限流规则的作用对象
- `count`：限流阈值
- `grade`：限流阈值类型（QPS 或并发线程数）
- `limitApp`：流控针对的调用来源，若为 `default` 则不区分调用来源
- `strategy`：限流策略
- `controlBehavior`：流量控制效果（直接拒绝、Warm Up、匀速排队）

除了资源与规则以外，还有一个很重要的角色，就是根据什么纬度来实现规则判断？

Sentinel中提供了两个纬度：

1. 并发线程数
2. QPS

也就是说，我们可以选择根据不同的纬度，根据这些纬度的指标去匹配限流规则，一旦达到阈值，则直接触发流量控制。

默认情况下是根据QPS来限流的，这个属性是通过 `grade` 进行设置。

并发线程数控制

并发数控制用于保护业务线程池不被慢调用耗尽。

例如，当应用所依赖的下游应用由于某种原因导致服务不稳定、响应延迟增加，对于调用者来说，意味着吞吐量下降和更多的线程数占用，极端情况下甚至导致线程池耗尽。

为了应对太多线程占用的情况，业内有使用隔离的方案，比如通过不同业务逻辑使用不同线程池来隔离业务自身之间的资源争抢（线程池隔离）。

这种隔离方案虽然隔离性比较好，但是代价就是线程数目太多，线程上下文切换的 `overhead(开销)` 比较大，特别是对低延时的调用有比较大的影响。

Sentinel 并发控制不负责创建和管理线程池，而是简单统计当前请求上下文的线程数目（正在执行的调用数目），如果超出阈值，新的请求会被立即拒绝，效果类似于信号量隔离。

并发线程数控制通常在调用端进行配置。

并发线程数控制参数配置：

1. `grade: RuleConstant.FLOW_GRADE_THREAD`
2. `count`：此时它的含义是并发线程数量

QPS流量控制

当 QPS 超过某个阈值的时候，则采取措施进行流量控制行为（类似于我们前面说过的限流算法上的差异），Sentinel提供了四种流量控制行为

1. **直接拒绝** (CONTROL_BEHAVIOR_DEFAULT)
2. **Warm Up** (CONTROL_BEHAVIOR_WARM_UP)
3. **匀速排队** (CONTROL_BEHAVIOR_RATE_LIMITER, 漏桶算法)
4. **冷启动+匀速器** (CONTROL_BEHAVIOR_WARM_UP_RATE_LIMITER) , 除了让流量缓慢增加，还控制的了请求的间隔时间，让请求已均匀速度通过。这种策略是1.4.0版本新增的。

这四个行为，是通过 `FlowRule` 中的 `controlBehavior` 属性来控制，默认是直接拒绝。

直接拒绝行为

直接拒绝 (`RuleConstant.CONTROL_BEHAVIOR_DEFAULT`) 方式是默认的流量控制方式，当 QPS 超过任意规则的阈值后，新的请求就会被立即拒绝，拒绝方式为抛出 `FlowException`。

这种方式适用于对系统处理能力确切已知的情况下，比如通过压测确定了系统的准确水位时。

Warm Up

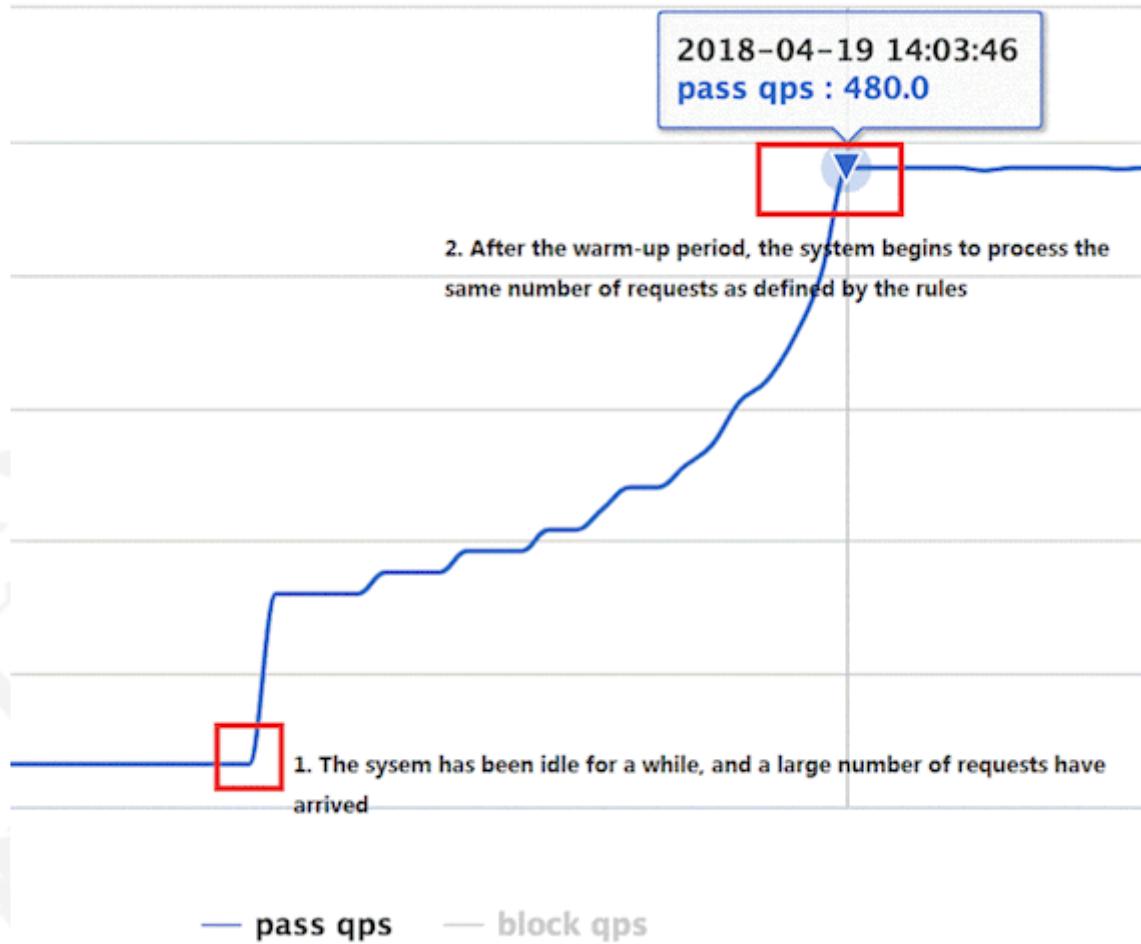
Warm Up (`RuleConstant.CONTROL_BEHAVIOR_WARM_UP`) 方式，即预热/冷启动方式。当系统长期处于低水位的情况下，当流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮。

通过“冷启动”，让通过的流量缓慢增加，在一定时间内逐渐增加到阈值上限，给冷系统一个预热的时间，避免冷系统被压垮。

以下都会随着系统访问量增加逐步预热来提升性能的因素。

- 缓存预热
- 数据库连接池初始化

如下图所示，当前系统所能够处理的最大并发数是480，首先在最下面的标记位置，系统一直处于空闲状态，接着请求量突然直线升高，这个时候系统并不是直接将QPS拉到最大值，而是在一定时间内逐步增加阈值，而中间这段时间就是一个系统逐步预热的过程。



属性设置：

- controlBehavior: RuleConstant.CONTROL_BEHAVIOR_WARM_UP
- warmUpPeriodSec: 预热时间, 默认60s。

案例演示：

基于 `gpmall-alibaba-portal` 模块中现有的限流接口: `http://localhost:8080/hi`, 修改流控规则

```
public class FlowRuleInitFunc implements InitFunc {
    @Override
    public void init() throws Exception {
        List<FlowRule> rules=new ArrayList<>();
        FlowRule rule=new FlowRule();
        rule.setResource("doTest");
        rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
        rule.setControlBehavior(RuleConstant.CONTROL_BEHAVIOR_WARM_UP); //行为: warmup
        rule.setWarmUpPeriodSec(60); //warmup时间 60s
        rule.setCount(1000); //并发数量1000
        rules.add(rule);
        FlowRuleManager.loadRules(rules);
    }
}
```

Sentinel-Dashboard中的见证结果如下，看图绿色部分，当达到60s时，qps达到1000。

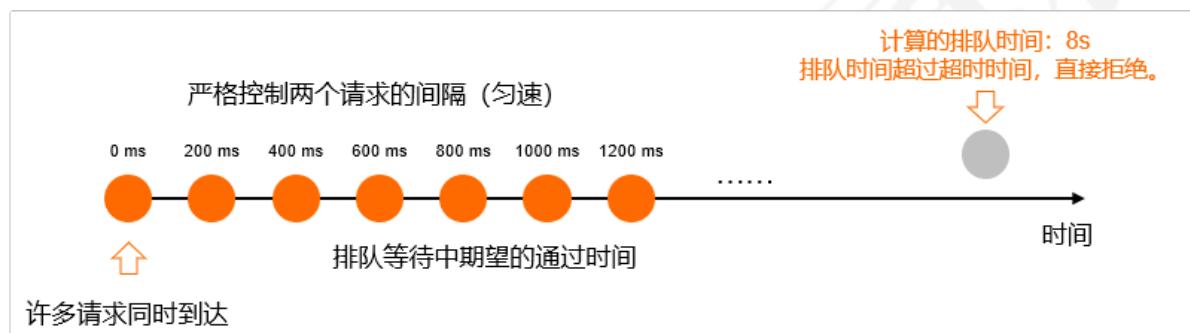


匀速排队

匀速排队 (`RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER`) 方式会严格控制请求通过的间隔时间，也即是让请求以均匀的速度通过，其实对应的是漏桶算法。

当请求数量远远大于阈值时，这些请求会排队等待，这个等待时间可以设置，如果超过等待时间，那这个请求会被拒绝。

如下图所示，假设qps=5，表示请求每200ms才能通过1个，多处的请求排队等待，超时时间达标最大排队时间，超过最大排队时间则直接拒绝。



属性设置：

- `controlBehavior: RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER`
- `maxQueueingTimeMs: 排队等待时间，表示每一次请求最长等待时间，默认是500ms`

这种方式主要用于处理间隔性突发的流量，例如消息队列。想象一下这样的场景，在某一秒有大量的请求到来，而接下来的几秒则处于空闲状态，我们希望系统能够在接下来的空闲期间逐渐处理这些请求，而不是在第一秒直接拒绝多余的请求。

所以，这里等待时间大一点，可以保证让所有请求都能正常通过；假设这里设置的排队等待时间过小的话，导致排队等待的请求超时而抛出异常`BlockException`，最终结果可能是这100个并发请求中只有一个请求或几个才能正常通过，所以使用这种模式得根据访问资源的耗时时间决定排队等待时间。

基于调用关系的流量控制

在分布式架构中，一个请求会包含调用方和被调用方，Sentinel还提供了服务调用关系的流量控制策略，所谓的调用关系，就是根据不同的调用纬度来触发流量控制。

1. 根据调用方限流 (`STRATEGY_DIRECT`)
2. 根据调用链路入口限流 (`STRATEGY_CHAIN`)
3. 具有关系的资源流量控制 (`STRATEGY_RELATE`)

根据调用方限流

顾名思义，假设有两个服务分别是A和B，都向某一个服务C发起请求调用，这个时候我们希望对来自服务B的请求进行限流，那就可以采用调用方限流策略，具体配置如下：

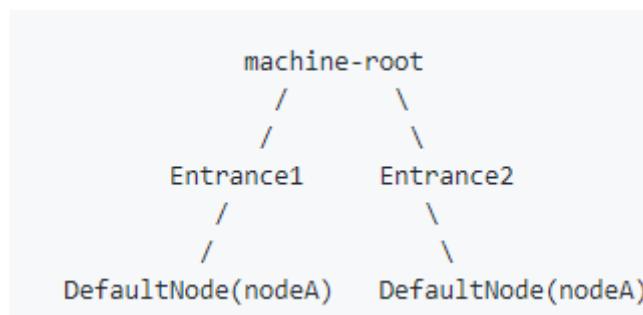
1. 设置FlowRule的strategy为STRATEGY_DIRECT
2. 设置FlowRule的LimitApp，表示指定调用方，这个字段有三种选项
 - `default`，表示不区分调用者，任何调用者的请求都会进行流量统计。
 - `${some_origin_name}`，针对某个特定的调用者，只有这个调用者的请求才会进行流量控制
 - `other`，表示针对除了 `${some_origin_name}`以外的其他调用方的流量进行流量控制。
假设资源NodeA配置了一条针对调用者`caller1`的限流规则，接着又配置了一条调用者为`other`的规则，那么任意来自非`caller1`的对NodeA的调用，请求并发数都不能超过`other`这条规则定义的阈值。

演示方法如下

```
public class OtherRuleExample {  
  
    private static void init(){  
        List<FlowRule> rules = new ArrayList<>();  
        FlowRule rule = new FlowRule();  
        rule.setResource("hello");  
        rule.setCount(4);  
        rule.setLimitApp("caller"); //配置caller为调用者  
        rules.add(rule);  
        FlowRule rule1 = new FlowRule();  
        rule1.setResource("hello");  
        rule1.setLimitApp("other"); //配置other  
        rule1.setCount(3);  
        rules.add(rule1);  
        FlowRuleManager.loadRules(rules);  
    }  
  
    public static void main(String[] args) {  
        init();  
        for (int i = 0; i < 6; i++) {  
            ContextUtil.enter("context", "caller11");  
            //声明当前的调用方的应用名称， 通过origin参数  
            try(Entry entry=SphU.entry("hello")){  
                System.out.println("访问成功");  
            }catch (BlockException e){  
                System.out.println("被限流了");  
            }  
        }  
    }  
}  
// 如果ContextUtil.enter中的origin设置为`caller11`， 表示是非`caller`调用，则按照  
limitApp=other的匹配规则进行流量控制，也就是QPS=3  
// 如果ContextUtil.enter中的origin设置为`caller`， 表示当前是`caller`的调用，则按照  
limitApp=caller的匹配规则进行流量控制，也就是QPS=4
```

根据调用链路入口限流

一个被限流的保护方法，可能来自于不同的调用链路，比如针对资源NodeA，入口 Entrance1 和 Entrance2 的请求都调用到了资源 NodeA，Sentinel 允许只根据某个入口的统计信息对资源限流。



设置方式：

1. 设置FlowRule中的strategy=STRATEGY_CHAIN
2. 设置FlowRule中的refResource为 Entrance1 来表示只有从入口 Entrance1 的调用才会进行流量控制

具有关系的资源流量控制

当两个资源之间具有资源争抢或者依赖关系的时候，这两个资源便具有了关联

比如对数据库同一个字段的读操作和写操作存在争抢，读的速度过高会影响写的速度，写的速度过高会影响读的速度。

如果放任读写操作争抢资源，则争抢本身带来的开销会降低整体的吞吐量。

可使用关联限流来避免具有关联关系的资源之间过度的争抢，举例来说，`read_db` 和 `write_db` 这两个资源分别代表数据库读写，我们可以给 `read_db` 设置限流规则来达到写优先的目的：

设置方式：

1. 设置FlowRule中的strategy=STRATEGY_RELATE
2. 设置FlowRule中的refResource为 `write_db` 表示设置关联资源

通过这样的设置后，如果 `write_db` 资源超过阈值时，就会对 `read_db` 资源进行限流。

阶段性总结

在这个阶段，我们跟进一步了解了Sentinel限流规则中的各种限流纬度，也让我们对 Sentinel有了更深刻的理解。

不管是基于并发线程数、还是QPS、还是根据调用链路等纬度，本质上都是当前系统所关注的资源保护指标，最终意义是让系统平稳运行！

下面，我们来了解Sentinel中的另外一个功能，熔断降级。

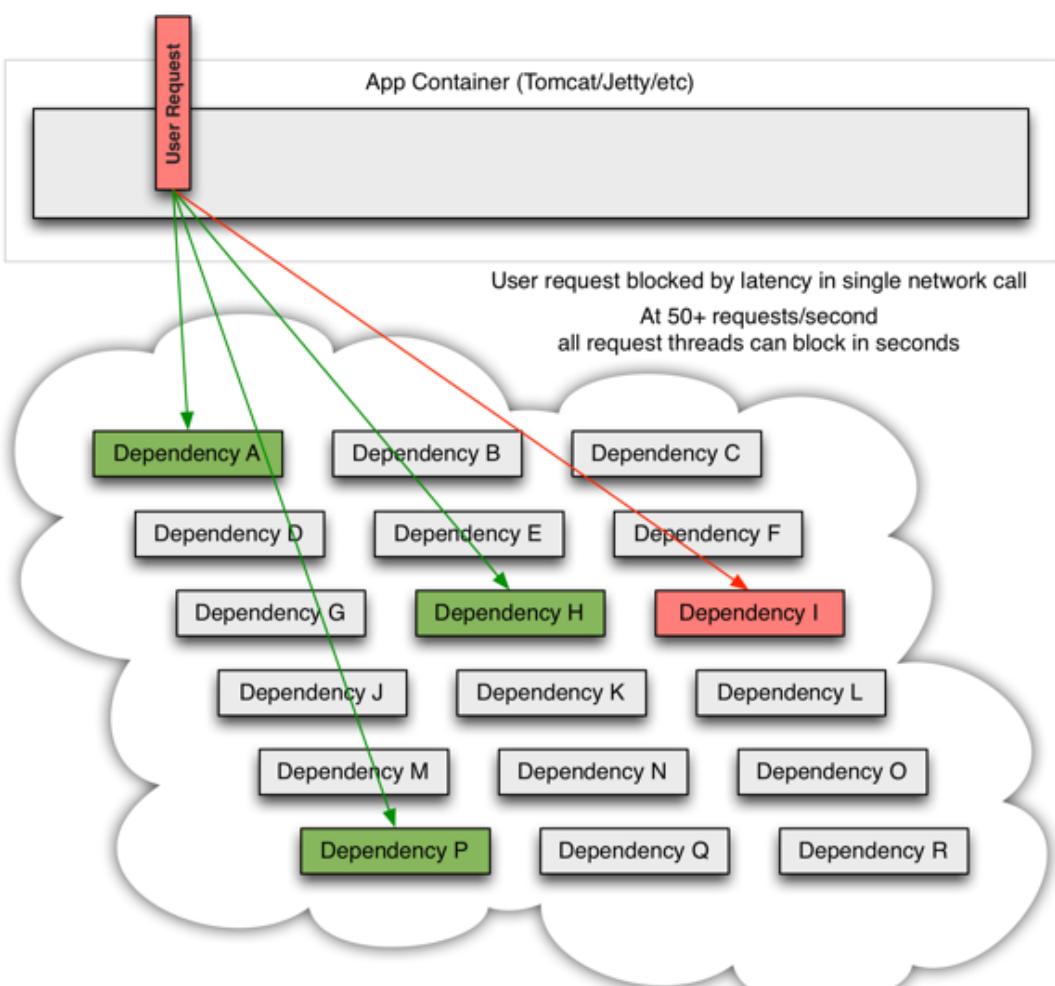
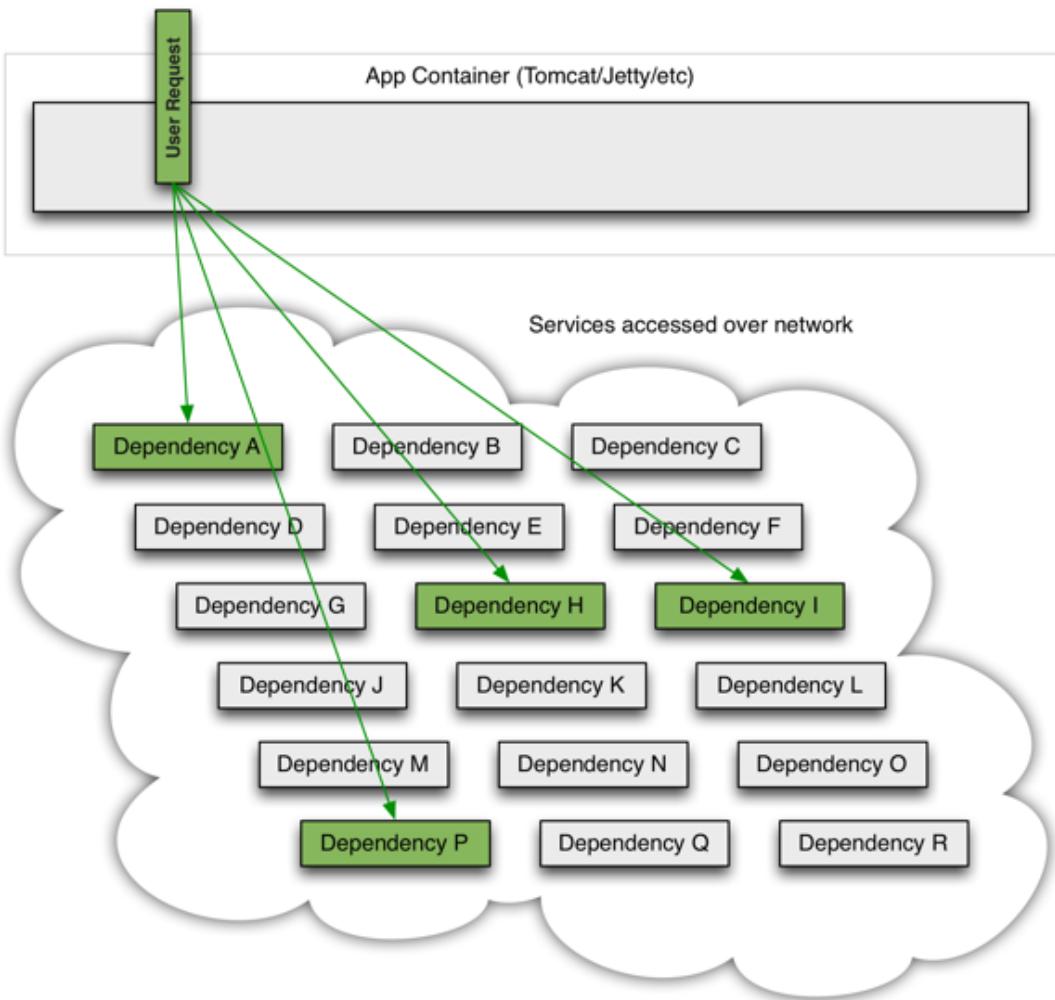
Sentinel中熔断降级

熔断是一种系统保护措施，就是当系统的某些阈值触发到设定的临界点时，所触发的行为方式，熔断的概念在很多地方都有听到。

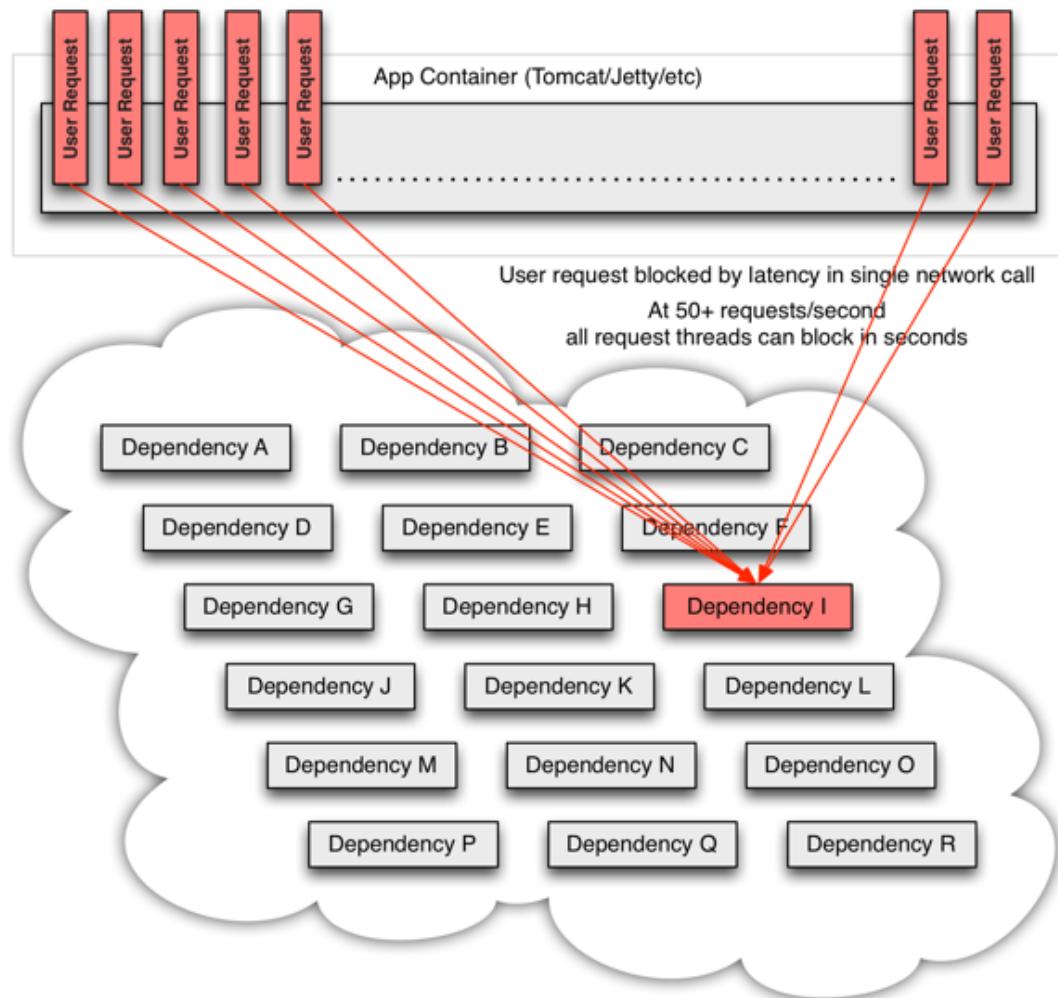
1. 股市熔断，比如美股在7%、13%、20%的时候会分别触发熔断15分钟，在这个期间，一切交易将会终止。故事情断的意义是稳定市场走势，给投资者冷静思考的时间。
2. 电路熔断，当电流超出导线所能承受的最大电流时，则会触发熔断，此时所有电路全部断电，避免异常电流引发火灾。这种熔断场景是为了保护财产和生命安全。

在软件架构中，这类的问题也同样存在。

如下图所示，在微服务架构中，一个请求过来，可能会经过多个服务进行处理，导致整个处理链路会比较长。而在整条调用链路中，可能会因为某个节点因为网络故障导致响应时间比较长，而这个节点的阻塞将会影响这条链路的结果返回（右图）。



当访问量比较高的请求下，一个后端依赖节点的延迟响应可能导致所有服务器上的所有资源在数秒内饱和。一旦出现这个问题，会导致系统资源被快速消耗，从而导致服务宕机等问题，最坏的情况会导致服务雪崩。



所以在软件系统中，为了防止这种问题的产生，也引入了熔断的概念。所以，熔断的意义是：如果某个目标服务调用比较慢或者大量的超时，这个时候如果触发熔断机制，则可以保证后续的请求不会继续发送到目标服务上，而是直接返回降级的逻辑并且快速释放资源。如果目标服务的情况恢复了，那么熔断机制又会动态进行关闭。

Sentinel 熔断的基本介绍

Sentinel 熔断降级会在调用链路中某个资源出现不稳定状态时（例如调用超时或异常比例升高），对这个资源的调用进行限制，让请求快速失败，避免影响到其它的资源而导致级联错误。当资源被降级后，在接下来的降级时间窗口之内，对该资源的调用都自动熔断。

如何判定资源不稳定呢？

大家思考一下，怎么去判断资源是否处于稳定状态呢？

1. 慢调用比例 (SLOW_REQUEST_RATIO)。
2. 异常比例 (ERROR_RATIO)。
3. 异常数 (ERROR_COUNT)。

这些纬度，在Sentinel中提供了DegradeRule对象来实现规则设置，核心属性如下。

- resource, 资源名称
- count, 阈值, [异常比例/异常数模式下为对应的阈值, 慢调用比例模式下为慢调用临界 RT]
- grade, 熔断模式, 根据RT降级、根据异常比例、根据异常数量
- timeWindow, 熔断时间, 单位为秒

下面分别对熔断的三个纬度做一个详细说明。

慢调用比例 (SLOW_REQUEST_RATIO)

在一定请求次数中，一段时间内，如果有一定比例的请求响应时间大于某一个阈值，则认为目标服务异常，则在接下来的指定时间内，请求都会被自动熔断。当经过熔断时长后，熔断器会进入到探测恢复状态，若接下来的一个请求响应时间小于设置的慢调用 RT 则结束熔断，若大于设置的慢调用 RT 则会再次被熔断。

假设有一个场景，如果1s内连续发送10个请求，在1分钟以内，其中20%的请求平均响应时间都超过3s，则触发熔断，熔断时间为5s，针对这种情况的设置方式如下。

1. grade=CircuitBreakerStrategy.SLOW_REQUEST_RATIO, (熔断模式)
2. count=3000, 最大的响应时间, 单位为(毫秒)
3. TimeWindow=5 (单位为s)
4. minRequestAmount=5, 最小请求数量, 请求数量小于这个值, 即使异常比例超出阈值也不会熔断, 默认是5次。
5. slowRatioThreshold=0.2, 慢调用比例阈值, 仅仅在慢调用比例模式下有效。
6. statIntervalMs=1000*60, 统计时长为60秒, 默认为1秒

```
DegradeRule rule = new DegradeRule(RESOURCE_KEY)
    .setGrade(CircuitBreakerStrategy.SLOW_REQUEST_RATIO.getType())
    // Max allowed response time
    .setCount(3000)
    // Retry timeout (in second)
    .setTimeWindow(5)
    // Circuit breaker opens when slow request ratio > 20%
    .setSlowRatioThreshold(0.2)
    .setMinRequestAmount(10)
    .setStatIntervalMs(60000);
```

异常比例 (ERROR_RATIO)

当资源的每秒请求量 ≥ 5 ，并且每秒异常总数占通过量的比值超过阈值时，则触发熔断，配置方式如下。

1. grade=CircuitBreakerStrategy.ERROR_RATIO
2. count (异常比例) , 范围[0.0 , 1.0], 代表0%~100%
3. TimeWindow=5 (单位为s)
4. minRequestAmount, 最小请求数量, 请求数量小于这个值, 即使异常比例超出阈值也不会熔断, 默认是5次。

异常数量(ERROR_COUNT)

当单位统计时长内的异常数目超过阈值之后会自动进行熔断。经过熔断时长后熔断器会进入探测恢复状态（HALF-OPEN 状态），若接下来的一个请求成功完成（没有错误）则结束熔断，否则会再次被熔断。

当资源近 1 分钟的异常数目超过阈值之后会进行熔断。注意由于统计时间窗口是分钟级别的，若 timeWindow 小于 60s，则结束熔断状态后仍可能再进入熔断状态。

1. grade=CircuitBreakerStrategy.ERROR_COUNT
2. count (异常数量)
3. timeWindow=5 (熔断时间窗口, 单位为s)
4. statIntervalMs=60*1000 (统计时长, 单位为ms)

Spring Cloud Alibaba集成Sentinel实现熔断

详见课程源代码。

Sentinel集成Nacos实现动态流控规则

在上述案例分析中，Sentinel的限流或者熔断规则，是配置在项目本地通过SPI扩展点来实现的。

既然Sentinel在Spring Cloud Alibaba生态下，我们是不是能够把Sentinel中的限流规则存储在配置中心呢？答案显然是可以的

Sentinel 的理念是开发者只需要关注资源的定义，当资源定义成功后可以动态增加各种流控降级规则。Sentinel 提供两种方式修改规则：

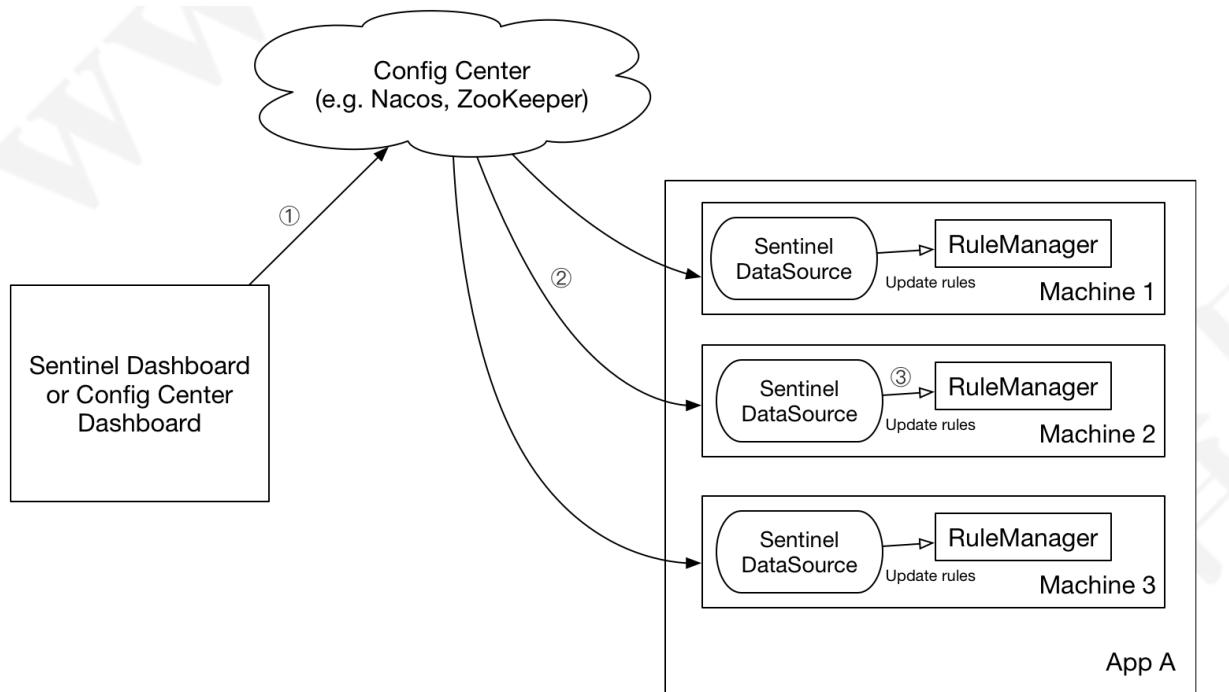
- 通过 API 直接修改 (`loadRules`)
- 通过 `DataSource` 适配不同数据源修改

手动通过 API 修改比较直观，可以通过以下几个 API 修改不同的规则：

```
FlowRuleManager.loadRules(List<FlowRule> rules); // 修改流控规则  
DegradeRuleManager.loadRules(List<DegradeRule> rules); // 修改降级规则
```

手动修改规则（硬编码方式）一般仅用于测试和演示，生产上一般通过动态规则源的方式来动态管理规则

动态规则管理的架构图如下。



从图中可以看到，我们可以通过 Sentinel Dashboard 或者 Config Center Dashboard 把流控规则推送到统一的配置中心（Nacos、Zookeeper 等），客户端通过实现 `ReadableDataSource` 接口来监听配置中心，从而实时获取变更的规则实现动态控制。

关于DataSource说明

`DataSource` 扩展常见的实现方式有：

- **拉模式**：客户端主动向某个规则管理中心定期轮询拉取规则，这个规则中心可以是 RDBMS、文件，甚至是 VCS 等。这样做的方式是简单，缺点是无法及时获取变更；
- **推模式**：规则中心统一推送，客户端通过注册监听器的方式时刻监听变化，比如使用 Nacos、Zookeeper 等配置中心。这种方式有更好的实时性和一致性保证。

Sentinel 目前支持以下数据源扩展：

- Pull-based: 动态文件数据源、Consul, Eureka
- Push-based: ZooKeeper, Redis, Nacos, Apollo, etcd

拉模式拓展 实现拉模式的数据源最简单的方式是继承 `AutoRefreshDataSource` 抽象类，然后实现 `readSource()` 方法，在该方法里从指定数据源读取字符串格式的配置数据。比如基于文件的数据源。

推模式拓展 实现推模式的数据源最简单的方式是继承 `AbstractDataSource` 抽象类，在其构造方法中添加监听器，并实现 `readSource()` 从指定数据源读取字符串格式的配置数据。比如 基于 Nacos 的数据源。

控制台通常需要做一些改造来直接推送应用维度的规则到配置中心。功能示例可以参考 AHAS Sentinel 控制台的规则推送功能。改造指南可以参考 在生产环境中使用 Sentinel 控制台。

基于配置形式实现动态限流

上面的动态限流规则配置，是基于 SPI 扩展的 `InitFunc` 来实现的

在 Spring Cloud Alibaba 中，我们可以直接在 `application.properties` 文件中增加如下配置，达到同样的效果

```
spring.cloud.sentinel.datasource.nacos.nacos.serverAddr=192.168.8.133:8848
spring.cloud.sentinel.datasource.nacos.nacos.dataId=com.gupaoedu.gpmall.user.HelloServ
ice
spring.cloud.sentinel.datasource.nacos.nacos.groupId=SENTINEL_GROUP
spring.cloud.sentinel.datasource.nacos.nacos.dataType=json
spring.cloud.sentinel.datasource.nacos.nacos.ruleType=flow
spring.cloud.sentinel.datasource.nacos.nacos.username=nacos
spring.cloud.sentinel.datasource.nacos.nacos.password=nacos
```

添加配置项之后，可以直接把InitFunc的扩展点删除，再次对接口进行压测，得到结果是相同的。

注意，Sentinel-Dashboard中配置的流控规则，默认情况下是没有做持久化的，如果需要实现配置的存储，需要改造Sentinel-Dashboard。