# 基本概念

## 资源

资源是 Sentinel 的关键概念。它可以是 Java 应用程序中的任何内容，例如，由应用程序提供的服务，或由应用程序调用的其它应用提供的服务，甚至可以是一段代码。在接下来的文档中，我们都会用资源来描述代码块。

只要通过 Sentinel API 定义的代码，就是资源，能够被 Sentinel 保护起来。大部分情况下，可以使用方法签名，URL，甚至服务名称作为资源名来标示资源。

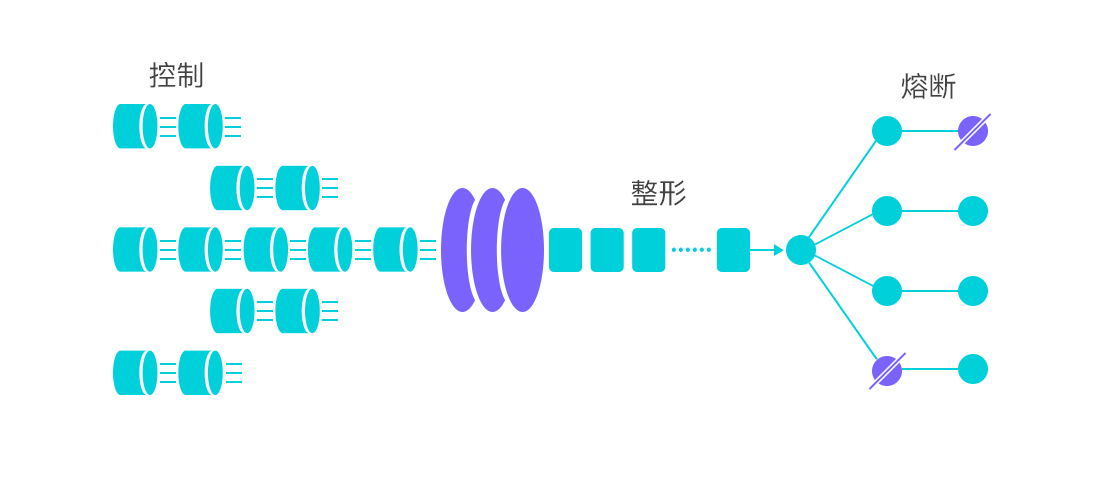
## 规则

围绕资源的实时状态设定的规则，可以包括流量控制规则、熔断降级规则以及系统保护规则。所有规则可以动态实时调整。

## 功能和设计理念

### 流量控制

流量控制在网络传输中是一个常用的概念，它用于调整网络包的发送数据。然而，从系统稳定性角度考虑，在处理请求的速度上，也有非常多的讲究。任意时间到来的请求往往是随机不可控的，而系统的处理能力是有限的。我们需要根据系统的处理能力对流量进行控制。Sentinel 作为一个调配器，可以根据需要把随机的请求调整成合适的形状，如下图所示：



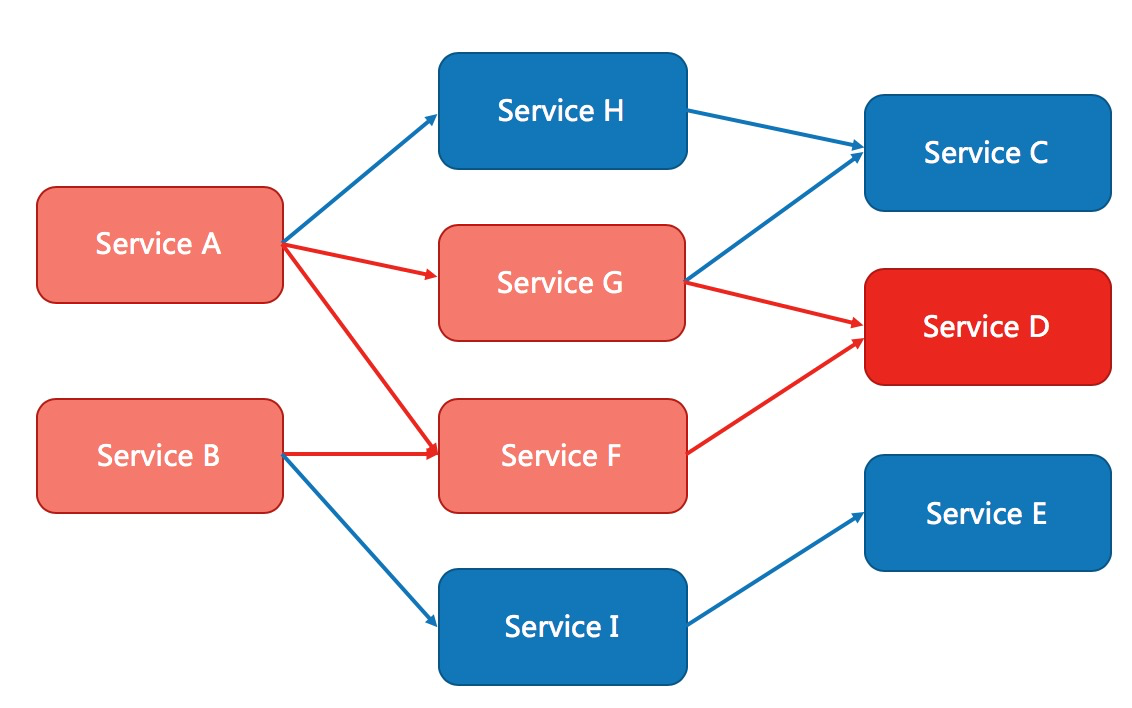
流量控制有以下几个角度:

* 资源的调用关系，例如资源的调用链路，资源和资源之间的关系；
* 运行指标，例如 QPS、线程池、系统负载等；
* 控制的效果，例如直接限流、冷启动、排队等。

Sentinel 的设计理念是让您自由选择控制的角度，并进行灵活组合，从而达到想要的效果。

### 熔断降级

除了流量控制以外，降低调用链路中的不稳定资源也是 Sentinel 的使命之一。由于调用关系的复杂性，如果调用链路中的某个资源出现了不稳定，最终会导致请求发生堆积。这个问题和 Hystrix 里面描述的问题是一样的。



Sentinel 和 Hystrix 的原则是一致的: 当调用链路中某个资源出现不稳定，例如，表现为 timeout，异常比例升高的时候，则对这个资源的调用进行限制，并让请求快速失败，避免影响到其它的资源，最终产生雪崩的效果。

在限制的手段上，Sentinel 和 Hystrix 采取了完全不一样的方法。

* Hystrix 通过线程池的方式，来对依赖(在我们的概念中对应资源)进行了隔离。这样做的好处是资源和资源之间做到了最彻底的隔离。缺点是除了增加了线程切换的成本，还需要预先给各个资源做线程池大小的分配。

Sentinel 对这个问题采取了两种手段:

* 通过并发线程数进行限制：和资源池隔离的方法不同，Sentinel 通过限制资源并发线程的数量，来减少不稳定资源对其它资源的影响。这样不但没有线程切换的损耗，也不需要您预先分配线程池的大小。当某个资源出现不稳定的情况下，例如响应时间变长，对资源的直接影响就是会造成线程数的逐步堆积。当线程数在特定资源上堆积到一定的数量之后，对该资源的新请求就会被拒绝。堆积的线程完成任务后才开始继续接收请求。
* 通过响应时间对资源进行降级：除了对并发线程数进行控制以外，Sentinel 还可以通过响应时间来快速降级不稳定的资源。当依赖的资源出现响应时间过长后，所有对该资源的访问都会被直接拒绝，直到过了指定的时间窗口之后才重新恢复。

### 系统负载保护

Sentinel 同时提供系统维度的自适应保护能力。防止雪崩，是系统防护中重要的一环。当系统负载较高的时候，如果还持续让请求进入，可能会导致系统崩溃，无法响应。在集群环境下，网络负载均衡会把本应这台机器承载的流量转发到其它的机器上去。如果这个时候其它的机器也处在一个边缘状态的时候，这个增加的流量就会导致这台机器也崩溃，最后导致整个集群不可用。

针对这个情况，Sentinel 提供了对应的保护机制，让系统的入口流量和系统的负载达到一个平衡，保证系统在能力范围之内处理最多的请求。

### 工作机制

Sentinel 的主要工作机制如下：

* 对主流框架提供适配或者显示的 API，来定义需要保护的资源，并提供设施对资源进行实时统计和调用链路分析。
* 根据预设的规则，结合对资源的实时统计信息，对流量进行控制。同时，Sentinel 提供开放的接口，方便您定义及改变规则。
* Sentinel 提供实时的监控系统，方便您快速了解目前系统的状态。

# 工作主流程

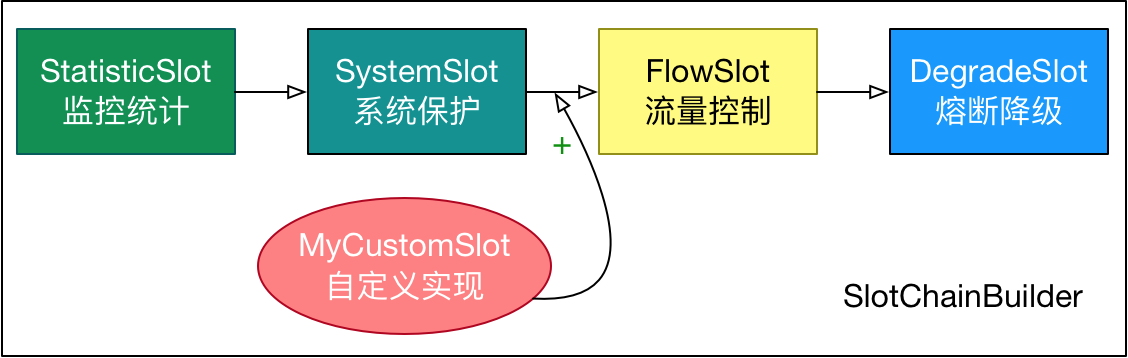
在 Sentinel 里面，所有的资源都对应一个资源名称以及一个 Entry。Entry 可以通过对主流框架的适配自动创建，也可以通过注解的方式或调用 API 显式创建；每一个 Entry 创建的时候，同时也会创建一系列功能插槽（slot chain）。这些插槽有不同的职责，例如:

* NodeSelectorSlot 负责收集资源的路径，并将这些资源的调用路径，以树状结构存储起来，用于根据调用路径来限流降级；
* ClusterBuilderSlot 则用于存储资源的统计信息以及调用者信息，例如该资源的 RT, QPS, thread count 等等，这些信息将用作为多维度限流，降级的依据；
* StatisticSlot 则用于记录、统计不同纬度的 runtime 指标监控信息；
* FlowSlot 则用于根据预设的限流规则以及前面 slot 统计的状态，来进行流量控制；
* AuthoritySlot 则根据配置的黑白名单和调用来源信息，来做黑白名单控制；
* DegradeSlot 则通过统计信息以及预设的规则，来做熔断降级；
* SystemSlot 则通过系统的状态，例如 load1 等，来控制总的入口流量；

总体的框架如下:

[总体框架](sentinel总体框架.png)

Sentinel 将 ProcessorSlot 作为 SPI 接口进行扩展（1.7.2 版本以前 SlotChainBuilder 作为 SPI），使得 Slot Chain 具备了扩展的能力。您可以自行加入自定义的 slot 并编排 slot 间的顺序，从而可以给 Sentinel 添加自定义的功能。



## NodeSelectorSlot

这个 slot 主要负责收集资源的路径，并将这些资源的调用路径，以树状结构存储起来，用于根据调用路径来限流降级。

## ClusterBuilderSlot

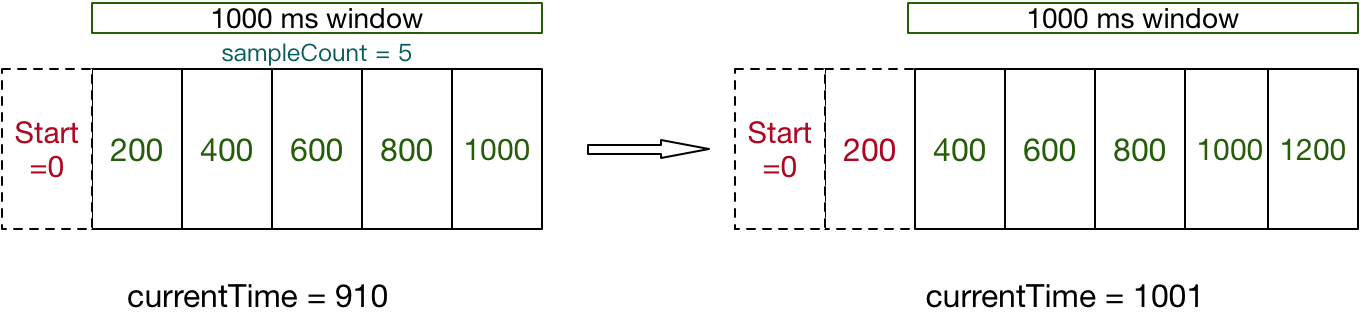
此插槽用于构建资源的 ClusterNode 以及调用来源节点。ClusterNode 保持资源运行统计信息（响应时间、QPS、block 数目、线程数、异常数等）以及原始调用者统计信息列表。来源调用者的名字由 ContextUtil.enter(contextName，origin) 中的 origin 标记。可通过如下命令查看某个资源不同调用者的访问情况：curl http://localhost:8719/origin?id=caller：

## StatisticSlot

StatisticSlot 是 Sentinel 的核心功能插槽之一，用于统计实时的调用数据。

* clusterNode：资源唯一标识的 ClusterNode 的 runtime 统计
* origin：根据来自不同调用者的统计信息
* defaultnode: 根据上下文条目名称和资源 ID 的 runtime 统计
* 入口的统计

Sentinel 底层采用高性能的滑动窗口数据结构 LeapArray 来统计实时的秒级指标数据，可以很好地支撑写多于读的高并发场景。



FlowSlot

这个 slot 主要根据预设的资源的统计信息，按照固定的次序，依次生效。如果一个资源对应两条或者多条流控规则，则会根据如下次序依次检验，直到全部通过或者有一个规则生效为止:

* 指定应用生效的规则，即针对调用方限流的；
* 调用方为 other 的规则；
* 调用方为 default 的规则。

## DegradeSlot

这个 slot 主要针对资源的平均响应时间（RT）以及异常比率，来决定资源是否在接下来的时间被自动熔断掉。

## SystemSlot

这个 slot 会根据对于当前系统的整体情况，对入口资源的调用进行动态调配。其原理是让入口的流量和当前系统的预计容量达到一个动态平衡。

注意系统规则只对入口流量起作用（调用类型为 EntryType.IN），对出口流量无效。可通过 SphU.entry(res, entryType) 指定调用类型，如果不指定，默认是EntryType.OUT。

# 基本使用

## 流量控制

FlowSlot 会根据预设的规则，结合前面 NodeSelectorSlot、ClusterNodeBuilderSlot、StatistcSlot 统计出来的实时信息进行流量控制。

限流的直接表现是在执行 Entry nodeA = SphU.entry(资源名字) 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。

同一个资源可以对应多条限流规则。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 或线程数
* strategy: 根据调用关系选择策略

### 基于QPS/并发数的流量控制

流量控制主要有两种统计类型，一种是统计线程数，另外一种则是统计 QPS。类型由 FlowRule.grade 字段来定义。其中，0 代表根据并发数量来限流，1 代表根据 QPS 来进行流量控制。其中线程数、QPS 值，都是由 StatisticSlot 实时统计获取的。可以通过下面的命令查看实时统计信息：

curl http://localhost:8719/cnode?id=resourceName

输出内容格式如下：

* thread： 代表当前处理该资源的线程数；
* pass： 代表一秒内到来到的请求；
* blocked： 代表一秒内被流量控制的请求数量；
* success： 代表一秒内成功处理完的请求；
* total： 代表到一秒内到来的请求以及被阻止的请求总和；
* RT： 代表一秒内该资源的平均响应时间；
* 1m-pass： 则是一分钟内到来的请求；
* 1m-block： 则是一分钟内被阻止的请求；
* 1m-all： 则是一分钟内到来的请求和被阻止的请求的总和；
* exception： 则是一秒内业务本身异常的总和。

#### 并发线程数流量控制

线程数限流用于保护业务线程数不被耗尽。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。为应对高线程占用的情况，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），或者使用信号量来控制同时请求的个数（信号量隔离）。这种隔离方案虽然能够控制线程数量，但无法控制请求排队时间。当请求过多时排队也是无益的，直接拒绝能够迅速降低系统压力。Sentinel线程数限流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个数，如果超出阈值，新的请求会被立即拒绝。

#### 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。流量控制的手段包括下面 3 种，对应 FlowRule 中的 controlBehavior 字段：

1. 直接拒绝（RuleConstant.CONTROL\_BEHAVIOR\_DEFAULT）方式。该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。
2. 冷启动（RuleConstant.CONTROL\_BEHAVIOR\_WARM\_UP）方式。该方式主要用于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，避免冷系统被压垮的情况。
3. 匀速器（RuleConstant.CONTROL\_BEHAVIOR\_RATE\_LIMITER）方式。这种方式严格控制了请求通过的间隔时间，也即是让请求以均匀的速度通过，对应的是漏桶算法。

### 基于调用关系的流量控制

调用关系包括调用方、被调用方；方法又可能会调用其它方法，形成一个调用链路的层次关系。Sentinel 通过 NodeSelectorSlot 建立不同资源间的调用的关系，并且通过 ClusterNodeBuilderSlot 记录每个资源的实时统计信息。有了调用链路的统计信息，我们可以衍生出多种流量控制手段。

#### 根据调用方限流

ContextUtil.enter(resourceName, origin) 方法中的 origin 参数标明了调用方身份。这些信息会在 ClusterBuilderSlot 中被统计。可通过以下命令来展示不同的调用方对同一个资源的调用数据：

curl http://localhost:8719/origin?id=nodeA

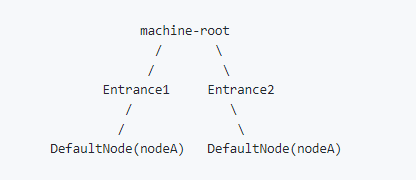
调用数据示例：

限流规则中的 limitApp 字段用于根据调用方进行流量控制。该字段的值有以下三种选项，分别对应不同的场景：

* default：表示不区分调用者，来自任何调用者的请求都将进行限流统计。如果这个资源名的调用总和超过了这条规则定义的阈值，则触发限流。
* {some\_origin\_name}：表示针对特定的调用者，只有来自这个调用者的请求才会进行流量控制。例如 NodeA 配置了一条针对调用者caller1的规则，那么当且仅当来自 caller1 对 NodeA 的请求才会触发流量控制。
* other：表示针对除 {some\_origin\_name} 以外的其余调用方的流量进行流量控制。例如，资源NodeA配置了一条针对调用者 caller1 的限流规则，同时又配置了一条调用者为 other 的规则，那么任意来自非 caller1 对 NodeA 的调用，都不能超过 other 这条规则定义的阈值。
* 同一个资源名可以配置多条规则，规则的生效顺序为：{some\_origin\_name} > other > default

#### 根据调用链路入口限流：链路限流

NodeSelectorSlot 中记录了资源之间的调用链路，这些资源通过调用关系，相互之间构成一棵调用树。这棵树的根节点是一个名字为 machine-root 的虚拟节点，调用链的入口都是这个虚节点的子节点。一棵典型的调用树如下图所示：



上图中来自入口 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。比如我们可以设置 FlowRule.strategy 为 RuleConstant.CHAIN，同时设置 FlowRule.ref\_identity 为 Entrance1 来表示只有从入口 Entrance1 的调用才会记录到 NodeA 的限流统计当中，而对来自 Entrance2 的调用漠不关心。调用链的入口是通过 API 方法 ContextUtil.enter(name) 定义的。

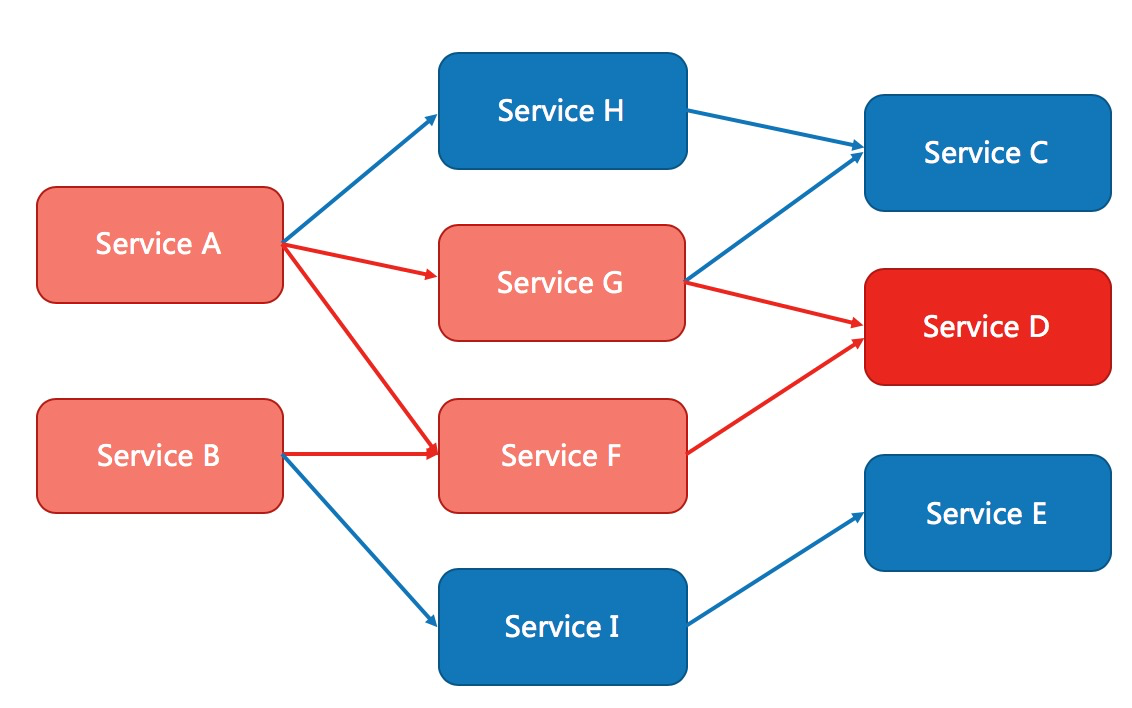
#### 具有关系的资源流量控制：关联流量控制

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得速度，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 FlowRule.strategy 为 RuleConstant.RELATE 同时设置 FlowRule.ref\_identity 为 write\_db。这样当写库操作过于频繁时，读数据的请求会被限流。

## 熔断降级

除了流量控制以外，对调用链路中不稳定的资源进行熔断降级也是保障高可用的重要措施之一。一个服务常常会调用别的模块，可能是另外的一个远程服务、数据库，或者第三方 API 等。例如，支付的时候，可能需要远程调用银联提供的 API；查询某个商品的价格，可能需要进行数据库查询。然而，这个被依赖服务的稳定性是不能保证的。如果依赖的服务出现了不稳定的情况，请求的响应时间变长，那么调用服务的方法的响应时间也会变长，线程会产生堆积，最终可能耗尽业务自身的线程池，服务本身也变得不可用。



现代微服务架构都是分布式的，由非常多的服务组成。不同服务之间相互调用，组成复杂的调用链路。以上的问题在链路调用中会产生放大的效果。复杂链路上的某一环不稳定，就可能会层层级联，最终导致整个链路都不可用。因此我们需要对不稳定的弱依赖服务调用进行熔断降级，暂时切断不稳定调用，避免局部不稳定因素导致整体的雪崩。熔断降级作为保护自身的手段，通常在客户端（调用端）进行配置。

### 熔断策略

Sentinel 提供以下几种熔断策略：

* 慢调用比例 (SLOW\_REQUEST\_RATIO)：选择以慢调用比例作为阈值，需要设置允许的慢调用 RT（即最大的响应时间），请求的响应时间大于该值则统计为慢调用。当单位统计时长（statIntervalMs）内请求数目大于设置的最小请求数目，并且慢调用的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。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。
* 异常比例 (ERROR\_RATIO)：当单位统计时长（statIntervalMs）内请求数目大于设置的最小请求数目，并且异常的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。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。异常比率的阈值范围是 [0.0, 1.0]，代表 0% - 100%。
* 异常数 (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。

注意异常降级仅针对业务异常，对 Sentinel 限流降级本身的异常（BlockException）不生效。为了统计异常比例或异常数，需要通过 Tracer.trace(ex) 记录业务异常。示例：

Entry entry = null;

try {

entry = SphU.entry(resource);

// Write your biz code here.

// <<BIZ CODE>>

} catch (Throwable t) {

if (!BlockException.isBlockException(t)) {

Tracer.trace(t);

}

} finally {

if (entry != null) {

entry.exit();

}

}

开源整合模块，如 Sentinel Dubbo Adapter, Sentinel Web Servlet Filter 或 @SentinelResource 注解会自动统计业务异常，无需手动调用。

### 熔断降级规则说明

熔断降级规则（DegradeRule）包含下面几个重要的属性：

|  |
| --- |
|  |
| **Field** | **说明** | **默认值** |
| resource | 资源名，即规则的作用对象 |  |
| grade | 熔断策略，支持慢调用比例/异常比例/异常数策略 | 慢调用比例 |
| count | 慢调用比例模式下为慢调用临界 RT（超出该值计为慢调用）；异常比例/异常数模式下为对应的阈值 |  |
| timeWindow | 熔断时长，单位为 s |  |
| minRequestAmount | 熔断触发的最小请求数，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 |
| statIntervalMs | 统计时长（单位为 ms），如 60\*1000 代表分钟级（1.8.0 引入） | 1000 ms |
| slowRatioThreshold | 慢调用比例阈值，仅慢调用比例模式有效（1.8.0 引入） |  |

### 熔断器事件监听

Sentinel 支持注册自定义的事件监听器监听熔断器状态变换事件（state change event）。示例：

EventObserverRegistry.getInstance().addStateChangeObserver("logging",

(prevState, newState, rule, snapshotValue) -> {

if (newState == State.OPEN) {

// 变换至 OPEN state 时会携带触发时的值

System.err.println(String.format("%s -> OPEN at %d, snapshotValue=%.2f", prevState.name(),

TimeUtil.currentTimeMillis(), snapshotValue));

} else {

System.err.println(String.format("%s -> %s at %d", prevState.name(), newState.name(),

TimeUtil.currentTimeMillis()));

}

});

## 系统自适应保护

Sentinel 系统自适应保护从整体维度对应用入口流量进行控制，结合应用的 Load、总体平均 RT、入口 QPS 和线程数等几个维度的监控指标，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。

长期以来，系统自适应保护的思路是根据硬指标，即系统的负载 (load1) 来做系统过载保护。当系统负载高于某个阈值，就禁止或者减少流量的进入；当 load 开始好转，则恢复流量的进入。这个思路给我们带来了不可避免的两个问题：

* load 是一个“果”，如果根据 load 的情况来调节流量的通过率，那么就始终有延迟性。也就意味着通过率的任何调整，都会过一段时间才能看到效果。当前通过率是使 load 恶化的一个动作，那么也至少要过 1 秒之后才能观测到；同理，如果当前通过率调整是让 load 好转的一个动作，也需要 1 秒之后才能继续调整，这样就浪费了系统的处理能力。所以我们看到的曲线，总是会有抖动。
* 恢复慢。想象一下这样的一个场景（真实），出现了这样一个问题，下游应用不可靠，导致应用 RT 很高，从而 load 到了一个很高的点。过了一段时间之后下游应用恢复了，应用 RT 也相应减少。这个时候，其实应该大幅度增大流量的通过率；但是由于这个时候 load 仍然很高，通过率的恢复仍然不高。

TCP BBR 的思想给了我们一个很大的启发。我们应该根据系统能够处理的请求，和允许进来的请求，来做平衡，而不是根据一个间接的指标（系统 load）来做限流。最终我们追求的目标是 在系统不被拖垮的情况下，提高系统的吞吐率，而不是 load 一定要到低于某个阈值。如果我们还是按照固有的思维，超过特定的 load 就禁止流量进入，系统 load 恢复就放开流量，这样做的结果是无论我们怎么调参数，调比例，都是按照果来调节因，都无法取得良好的效果。

### 系统规则

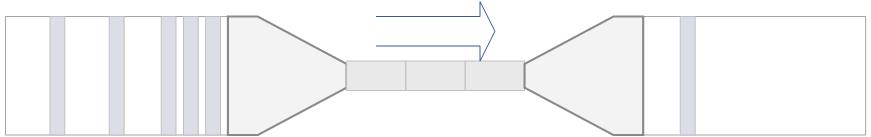
系统保护规则是从应用级别的入口流量进行控制，从单台机器的总体 Load、RT、入口 QPS 和线程数四个维度监控应用数据，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

系统保护规则是应用整体维度的，而不是资源维度的，并且仅对入口流量生效。入口流量指的是进入应用的流量（EntryType.IN），比如 Web 服务或 Dubbo 服务端接收的请求，都属于入口流量。

系统规则支持以下的阈值类型：

* Load（仅对 Linux/Unix-like 机器生效）：当系统 load1 超过阈值，且系统当前的并发线程数超过系统容量时才会触发系统保护。系统容量由系统的 maxQps \* minRt 计算得出。设定参考值一般是 CPU cores \* 2.5。
* CPU usage（1.5.0+ 版本）：当系统 CPU 使用率超过阈值即触发系统保护（取值范围 0.0-1.0）。
* RT：当单台机器上所有入口流量的平均 RT 达到阈值即触发系统保护，单位是毫秒。
* 线程数：当单台机器上所有入口流量的并发线程数达到阈值即触发系统保护。
* 入口 QPS：当单台机器上所有入口流量的 QPS 达到阈值即触发系统保护。

### 原理



我们把系统处理请求的过程想象为一个水管，到来的请求是往这个水管灌水，当系统处理顺畅的时候，请求不需要排队，直接从水管中穿过，这个请求的RT是最短的；反之，当请求堆积的时候，那么处理请求的时间则会变为：排队时间 + 最短处理时间。

* 推论一: 如果我们能够保证水管里的水量，能够让水顺畅的流动，则不会增加排队的请求；也就是说，这个时候的系统负载不会进一步恶化。

我们用 T 来表示(水管内部的水量)，用RT来表示请求的处理时间，用P来表示进来的请求数，那么一个请求从进入水管道到从水管出来，这个水管会存在 P \* RT　个请求。换一句话来说，当 T ≈ QPS \* Avg(RT) 的时候，我们可以认为系统的处理能力和允许进入的请求个数达到了平衡，系统的负载不会进一步恶化。

* 推论二:　当保持入口的流量是水管出来的流量的最大的值的时候，可以最大利用水管的处理能力。

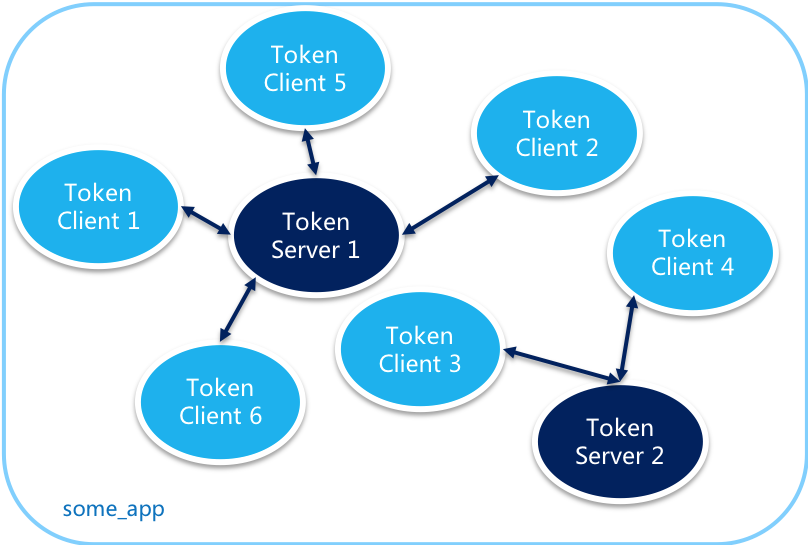
然而，和 TCP BBR 的不一样的地方在于，还需要用一个系统负载的值（load1）来激发这套机制启动。注：这种系统自适应算法对于低 load 的请求，它的效果是一个“兜底”的角色。**对于不是应用本身造成的 load 高的情况（如其它进程导致的不稳定的情况），效果不明显**。

## 集群流量控制

为什么要使用集群流控呢？假设我们希望给某个用户限制调用某个 API 的总 QPS 为 50，但机器数可能很多（比如有 100 台）。这时候我们很自然地就想到，找一个 server 来专门来统计总的调用量，其它的实例都与这台 server 通信来判断是否可以调用。这就是最基础的集群流控的方式。

另外集群流控还可以解决流量不均匀导致总体限流效果不佳的问题。假设集群中有 10 台机器，我们给每台机器设置单机限流阈值为 10 QPS，理想情况下整个集群的限流阈值就为 100 QPS。不过实际情况下流量到每台机器可能会不均匀，会导致总量没有到的情况下某些机器就开始限流。因此仅靠单机维度去限制的话会无法精确地限制总体流量。而集群流控可以精确地控制整个集群的调用总量，结合单机限流兜底，可以更好地发挥流量控制的效果。

* Token Client：集群流控客户端，用于向所属 Token Server 通信请求 token。集群限流服务端会返回给客户端结果，决定是否限流。
* Token Server：即集群流控服务端，处理来自 Token Client 的请求，根据配置的集群规则判断是否应该发放 token（是否允许通过）。



### 模块结构

Sentinel 1.4.0 开始引入了集群流控模块，主要包含以下几部分：

* sentinel-cluster-common-default: 公共模块，包含公共接口和实体
* sentinel-cluster-client-default: 默认集群流控 client 模块，使用 Netty 进行通信，提供接口方便序列化协议扩展
* sentinel-cluster-server-default: 默认集群流控 server 模块，使用 Netty 进行通信，提供接口方便序列化协议扩展；同时提供扩展接口对接规则判断的具体实现（TokenService），默认实现是复用 sentinel-core 的相关逻辑

### 集群流控规则

FlowRule 添加了两个字段用于集群限流相关配置：

private boolean clusterMode; // 标识是否为集群限流配置

private ClusterFlowConfig clusterConfig; // 集群限流相关配置项

其中 用一个专门的 ClusterFlowConfig 代表集群限流相关配置项，以与现有规则配置项分开：

// 全局唯一的规则 ID，由集群限流管控端分配.

private Long flowId;

// 阈值模式，默认（0）为单机均摊，1 为全局阈值.

private int thresholdType = ClusterRuleConstant.FLOW\_THRESHOLD\_AVG\_LOCAL;

private int strategy = ClusterRuleConstant.FLOW\_CLUSTER\_STRATEGY\_NORMAL;

// 在 client 连接失败或通信失败时，是否退化到本地的限流模式

private boolean fallbackToLocalWhenFail = true;

* flowId 代表全局唯一的规则 ID，Sentinel 集群限流服务端通过此 ID 来区分各个规则，因此务必保持全局唯一。一般 flowId 由统一的管控端进行分配，或写入至 DB 时生成。
* thresholdType 代表集群限流阈值模式。其中单机均摊模式下配置的阈值等同于单机能够承受的限额，token server 会根据客户端对应的 namespace（默认为 project.name 定义的应用名）下的连接数来计算总的阈值（比如独立模式下有 3 个 client 连接到了 token server，然后配的单机均摊阈值为 10，则计算出的集群总量就为 30）；而全局模式下配置的阈值等同于整个集群的总阈值。

### 集群限流客户端

要想使用集群限流功能，必须引入集群限流 client 相关依赖：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentinel-cluster-client-default</artifactId>

<version>1.8.5</version>

</dependency>

用户可以通过 API 将当前模式置为客户端模式：

http://<ip>:<port>/setClusterMode?mode=<xxx>

其中 mode 为 0 代表 client，1 代表 server。设置成功后，若已有客户端的配置，集群限流客户端将会开启并连接远程的 token server。我们可以在 sentinel-record.log 日志中查看连接的相关日志。

若集群限流客户端未进行配置，则用户需要对客户端进行基本的配置，比如指定集群限流 token server。我们提供了 API 进行配置：

http://<ip>:<port>/cluster/client/modifyConfig?data=<config>

其中 data 是 JSON 格式的 ClusterClientConfig，对应的配置项：

* serverHost: token server host
* serverPort: token server 端口
* requestTimeout: 请求的超时时间（默认为 20 ms）

当然也可以通过动态配置源进行配置。我们可以通过 ClusterClientConfigManager 的 registerServerAssignProperty 方法注册动态配置源。配置源注册的相关逻辑可以置于 InitFunc 实现类中，并通过 SPI 注册，在 Sentinel 初始化时即可自动进行配置源加载监听。

若用户未引入集群限流 client 相关依赖，或者 client 未开启/连接失败/通信失败，则对于开启了集群模式的规则：

* 集群热点限流默认直接通过
* 普通集群限流会退化到 local 模式的限流，即在本地按照单机阈值执行限流检查

### 集群限流服务端

要想使用集群限流服务端，必须引入集群限流 server 相关依赖：

<dependency>

<groupId>com.alibaba.csp</groupId>

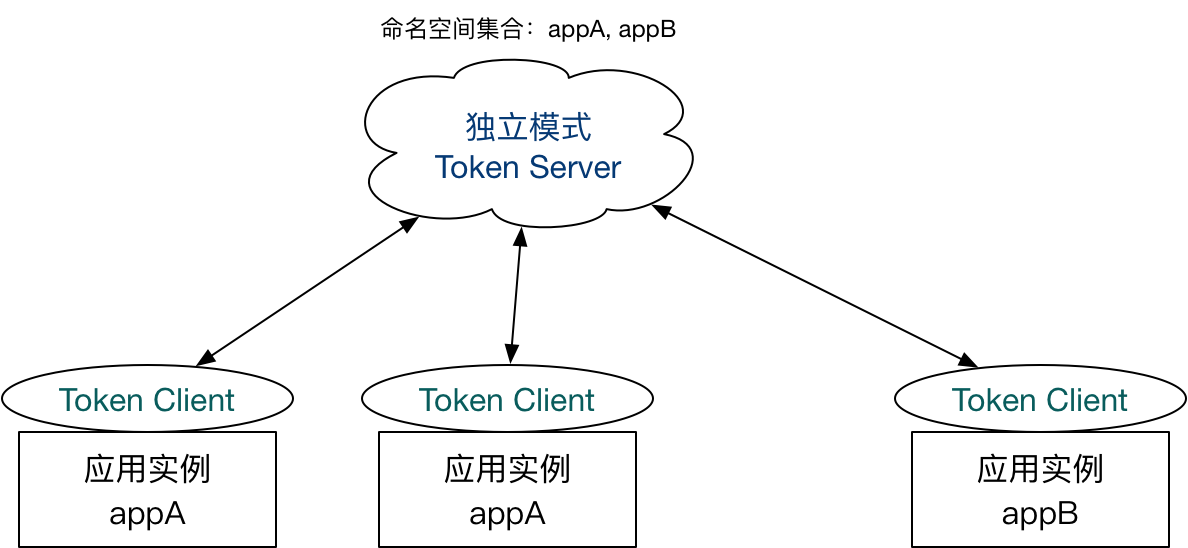
<artifactId>sentinel-cluster-server-default</artifactId>

<version>1.8.5</version>

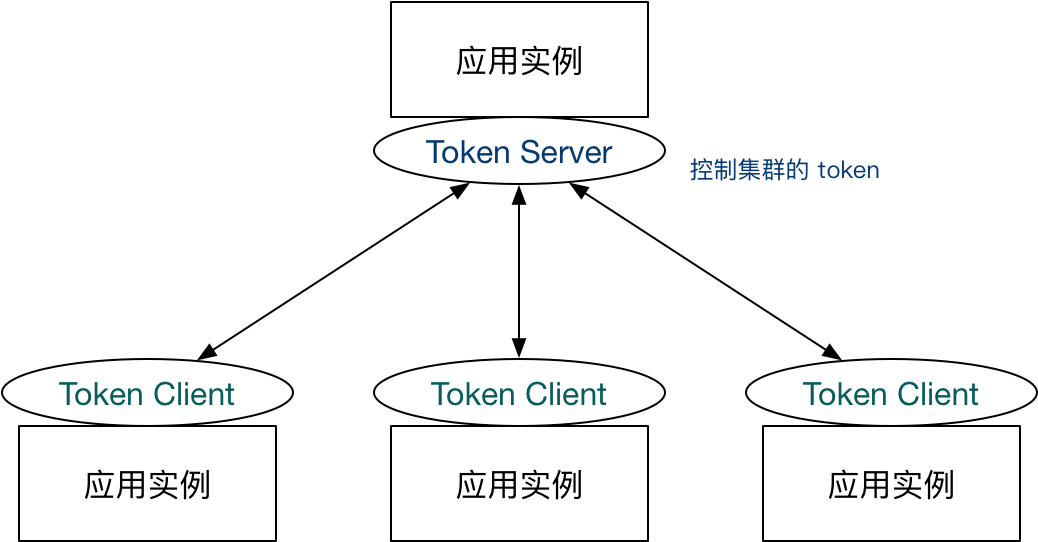
</dependency>

Sentinel 集群限流服务端有两种启动方式：

* 独立模式（Alone），即作为独立的 token server 进程启动，独立部署，隔离性好，但是需要额外的部署操作。独立模式适合作为 Global Rate Limiter 给集群提供流控服务。



* 嵌入模式（Embedded），即作为内置的 token server 与服务在同一进程中启动。在此模式下，集群中各个实例都是对等的，token server 和 client 可以随时进行转变，因此无需单独部署，灵活性比较好。但是隔离性不佳，需要限制 token server 的总 QPS，防止影响应用本身。嵌入模式适合某个应用集群内部的流控。



我们提供了 API 用于在 embedded 模式下转换集群流控身份：

http://<ip>:<port>/setClusterMode?mode=<xxx>

其中 mode 为 0 代表 client，1 代表 server，-1 代表关闭。注意应用端需要引入集群限流客户端或服务端的相应依赖。在独立模式下，我们可以直接创建对应的 ClusterTokenServer 实例并在 main 函数中通过 start 方法启动 Token Server。

配置类型有以下几种：

* namespace set: 集群限流服务端服务的作用域（命名空间），可以设置为自己服务的应用名。集群限流 client 在连接到 token server 后会上报自己的命名空间（默认为 project.name 配置的应用名），token server 会根据上报的命名空间名称统计连接数。
* transport config: 集群限流服务端通信相关配置，如 server port
* flow config: 集群限流服务端限流相关配置，如滑动窗口统计时长、格子数目、最大允许总 QPS等

我们可以通过 ClusterServerConfigManager 的各个 registerXxxProperty 方法来注册相关的配置源。从 1.4.1 版本开始，Sentinel 支持给 token server 配置最大允许的总 QPS（maxAllowedQps），来对 token server 的资源使用进行限制，防止在嵌入模式下影响应用本身。

### [Token Server 分配配置](Token%20Server%20分配配置.png)

### 集群限流控制台

[整体架构](整体架构.png)

sentinel-core 中的通用扩展接口：

* TokenService: 集群限流功能接口，server / client 均可复用
* ClusterTokenClient: 集群限流功能客户端
* ClusterTokenServer: 集群限流服务端接口
* EmbeddedClusterTokenServer: 集群限流服务端接口（embedded 模式）

以下通用接口位于 sentinel-cluster-common-default:

* EntityWriter
* EntityDecoder

Client 扩展接口，集群流控 Client 端通信相关扩展接口：

* ClusterTransportClient：集群限流通信客户端
* RequestEntityWriter
* ResponseEntityDecoder

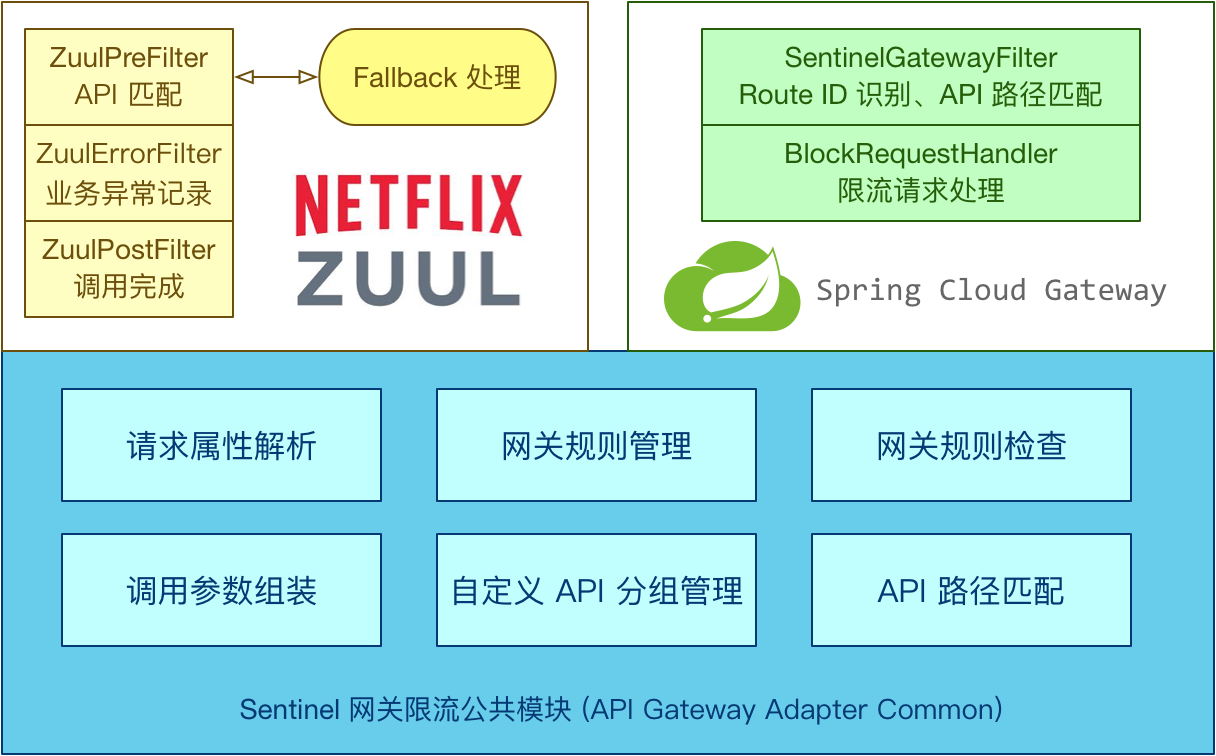
Server 扩展接口，集群流控 Server 端通信相关扩展接口：

* ResponseEntityWriter
* RequestEntityDecoder

集群流控 Server 端请求处理扩展接口：

* RequestProcessor: 请求处理接口 (request -> response)

## 网关流量控制



Sentinel 1.6.0 引入了 Sentinel API Gateway Adapter Common 模块，此模块中包含网关限流的规则和自定义 API 的实体和管理逻辑：

* GatewayFlowRule：网关限流规则，针对 API Gateway 的场景定制的限流规则，可以针对不同 route 或自定义的 API 分组进行限流，支持针对请求中的参数、Header、来源 IP 等进行定制化的限流。
* ApiDefinition：用户自定义的 API 定义分组，可以看做是一些 URL 匹配的组合。比如我们可以定义一个 API 叫 my\_api，请求 path 模式为 /foo/\*\* 和 /baz/\*\* 的都归到 my\_api 这个 API 分组下面。限流的时候可以针对这个自定义的 API 分组维度进行限流。

其中网关限流规则 GatewayFlowRule 的字段解释如下：

* resource：资源名称，可以是网关中的 route 名称或者用户自定义的 API 分组名称。
* resourceMode：规则是针对 API Gateway 的 route（RESOURCE\_MODE\_ROUTE\_ID）还是用户在 Sentinel 中定义的 API 分组（RESOURCE\_MODE\_CUSTOM\_API\_NAME），默认是 route。
* grade：限流指标维度，同限流规则的 grade 字段。
* count：限流阈值
* intervalSec：统计时间窗口，单位是秒，默认是 1 秒。
* controlBehavior：流量整形的控制效果，同限流规则的 controlBehavior 字段，目前支持快速失败和匀速排队两种模式，默认是快速失败。
* burst：应对突发请求时额外允许的请求数目。
* maxQueueingTimeoutMs：匀速排队模式下的最长排队时间，单位是毫秒，仅在匀速排队模式下生效。
* paramItem：参数限流配置。若不提供，则代表不针对参数进行限流，该网关规则将会被转换成普通流控规则；否则会转换成热点规则。其中的字段：
  + parseStrategy：从请求中提取参数的策略，目前支持提取来源 IP（PARAM\_PARSE\_STRATEGY\_CLIENT\_IP）、Host（PARAM\_PARSE\_STRATEGY\_HOST）、任意 Header（PARAM\_PARSE\_STRATEGY\_HEADER）和任意 URL 参数（PARAM\_PARSE\_STRATEGY\_URL\_PARAM）四种模式。
  + fieldName：若提取策略选择 Header 模式或 URL 参数模式，则需要指定对应的 header 名称或 URL 参数名称。
  + pattern：参数值的匹配模式，只有匹配该模式的请求属性值会纳入统计和流控；若为空则统计该请求属性的所有值。（1.6.2 版本开始支持）
  + matchStrategy：参数值的匹配策略，目前支持精确匹配（PARAM\_MATCH\_STRATEGY\_EXACT）、子串匹配（PARAM\_MATCH\_STRATEGY\_CONTAINS）和正则匹配（PARAM\_MATCH\_STRATEGY\_REGEX）。（1.6.2 版本开始支持）

用户可以通过 GatewayRuleManager.loadRules(rules) 手动加载网关规则，或通过 GatewayRuleManager.register2Property(property) 注册动态规则源动态推送（推荐方式）。

### Spring Cloud Gateway

从 1.6.0 版本开始，Sentinel 提供了 Spring Cloud Gateway 的适配模块，可以提供两种资源维度的限流：

* route 维度：即在 Spring 配置文件中配置的路由条目，资源名为对应的 routeId
* 自定义 API 维度：用户可以利用 Sentinel 提供的 API 来自定义一些 API 分组

使用时需引入以下模块（以 Maven 为例）：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentinel-spring-cloud-gateway-adapter</artifactId>

<version>x.y.z</version>

</dependency>

使用时只需注入对应的 SentinelGatewayFilter 实例以及 SentinelGatewayBlockExceptionHandler 实例即可。比如：

@Configuration

public class GatewayConfiguration {

private final List<ViewResolver> viewResolvers;

private final ServerCodecConfigurer serverCodecConfigurer;

public GatewayConfiguration(ObjectProvider<List<ViewResolver>> viewResolversProvider,

ServerCodecConfigurer serverCodecConfigurer) {

this.viewResolvers = viewResolversProvider.getIfAvailable(Collections::emptyList);

this.serverCodecConfigurer = serverCodecConfigurer;

}

@Bean

@Order(Ordered.HIGHEST\_PRECEDENCE)

public SentinelGatewayBlockExceptionHandler sentinelGatewayBlockExceptionHandler() {

// Register the block exception handler for Spring Cloud Gateway.

return new SentinelGatewayBlockExceptionHandler(viewResolvers, serverCodecConfigurer);

}

@Bean

@Order(-1)

public GlobalFilter sentinelGatewayFilter() {

return new SentinelGatewayFilter();

}

}

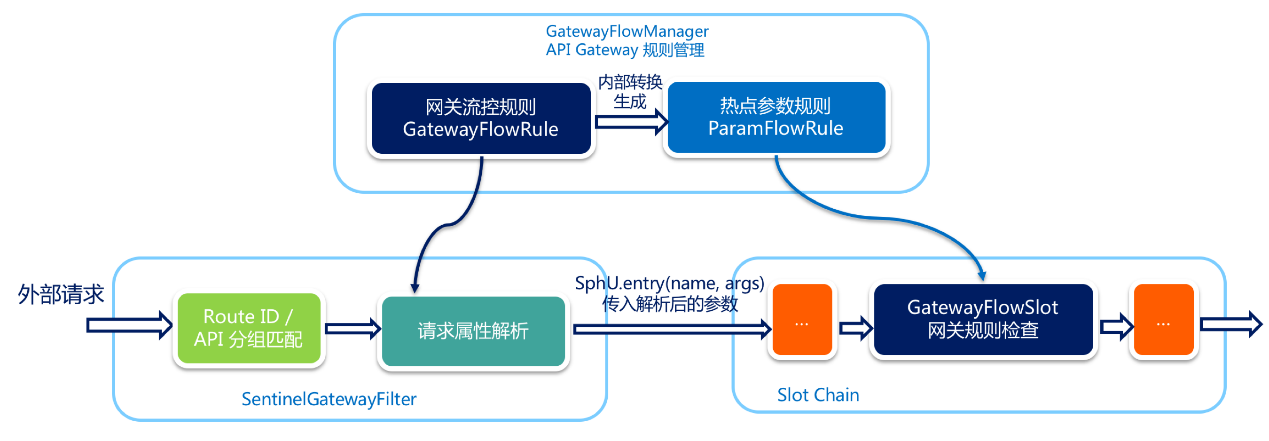
注意：

* Sentinel 网关流控默认的粒度是 route 维度以及自定义 API 分组维度，默认不支持 URL 粒度。若通过 Spring Cloud Alibaba 接入，请将 spring.cloud.sentinel.filter.enabled 配置项置为 false（若在网关流控控制台上看到了 URL 资源，就是此配置项没有置为 false）。
* 若使用 Spring Cloud Alibaba Sentinel 数据源模块，需要注意网关流控规则数据源类型是 gw-flow，若将网关流控规则数据源指定为 flow 则不生效。

### 网关流控实现原理

当通过 GatewayRuleManager 加载网关流控规则（GatewayFlowRule）时，无论是否针对请求属性进行限流，Sentinel 底层都会将网关流控规则转化为热点参数规则（ParamFlowRule），存储在 GatewayRuleManager 中，与正常的热点参数规则相隔离。转换时 Sentinel 会根据请求属性配置，为网关流控规则设置参数索引（idx），并同步到生成的热点参数规则中。

外部请求进入 API Gateway 时会经过 Sentinel 实现的 filter，其中会依次进行 路由/API 分组匹配、请求属性解析和参数组装。Sentinel 会根据配置的网关流控规则来解析请求属性，并依照参数索引顺序组装参数数组，最终传入 SphU.entry(res, args) 中。Sentinel API Gateway Adapter Common 模块向 Slot Chain 中添加了一个 GatewayFlowSlot，专门用来做网关规则的检查。GatewayFlowSlot 会从 GatewayRuleManager 中提取生成的热点参数规则，根据传入的参数依次进行规则检查。若某条规则不针对请求属性，则会在参数最后一个位置置入预设的常量，达到普通流控的效果。

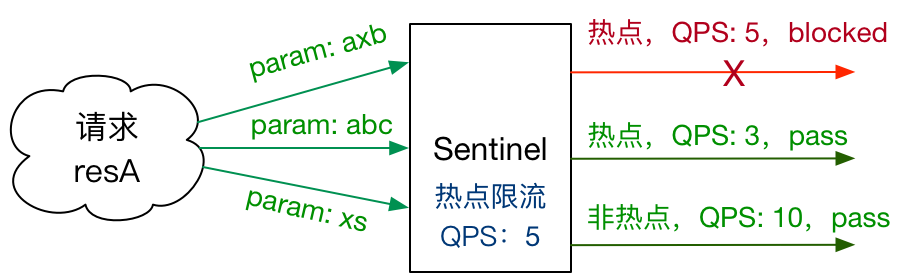


## 热点参数限流

热点即经常访问的数据。很多时候我们希望统计某个热点数据中访问频次最高的 Top K 数据，并对其访问进行限制。比如：

* 商品 ID 为参数，统计一段时间内最常购买的商品 ID 并进行限制
* 用户 ID 为参数，针对一段时间内频繁访问的用户 ID 进行限制

热点参数限流会统计传入参数中的热点参数，并根据配置的限流阈值与模式，对包含热点参数的资源调用进行限流。热点参数限流可以看做是一种特殊的流量控制，仅对包含热点参数的资源调用生效。



Sentinel 利用 LRU 策略统计最近最常访问的热点参数，结合令牌桶算法来进行参数级别的流控。

### 基本使用

要使用热点参数限流功能，需要引入以下依赖：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentinel-parameter-flow-control</artifactId>

<version>x.y.z</version>

</dependency>

然后为对应的资源配置热点参数限流规则，并在 entry 的时候传入相应的参数，即可使热点参数限流生效。我们可以通过 SphU 类里面几个 entry 重载方法来传入：

public static Entry entry(String name, EntryType type, int count, Object... args) throws BlockException

public static Entry entry(Method method, EntryType type, int count, Object... args) throws BlockException

其中最后的一串 args 就是要传入的参数，有多个就按照次序依次传入。比如要传入两个参数 paramA 和 paramB，则可以：

// paramA in index 0, paramB in index 1.

// 若需要配置例外项或者使用集群维度流控，则传入的参数只支持基本类型。

SphU.entry(resourceName, EntryType.IN, 1, paramA, paramB);

注意：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。正确的示例：

Entry entry = null;

try {

entry = SphU.entry(resourceName, EntryType.IN, 1, paramA, paramB);

// Your logic here.

} catch (BlockException ex) {

// Handle request rejection.

} finally {

if (entry != null) {

entry.exit(1, paramA, paramB);

}

}

对于 @SentinelResource 注解方式定义的资源，若注解作用的方法上有参数，Sentinel 会将它们作为参数传入 SphU.entry(res, args)。比如以下的方法里面 uid 和 type 会分别作为第一个和第二个参数传入 Sentinel API，从而可以用于热点规则判断：

@SentinelResource("myMethod")

public Result doSomething(String uid, int type) {

// some logic here...

}

### 热点参数规则

热点参数规则（ParamFlowRule）类似于流量控制规则（FlowRule）：

|  |  |  |
| --- | --- | --- |
| **属性** | **说明** | **默认值** |
| resource | 资源名，必填 |  |
| count | 限流阈值，必填 |  |
| grade | 限流模式 | QPS 模式 |
| durationInSec | 统计窗口时间长度（单位为秒），1.6.0 版本开始支持 | 1s |
| controlBehavior | 流控效果（支持快速失败和匀速排队模式），1.6.0 版本开始支持 | 快速失败 |
| maxQueueingTimeMs | 最大排队等待时长（仅在匀速排队模式生效），1.6.0 版本开始支持 | 0ms |
| paramIdx | 热点参数的索引，必填，对应 SphU.entry(xxx, args) 中的参数索引位置 |  |
| paramFlowItemList | 参数例外项，可以针对指定的参数值单独设置限流阈值，不受前面 count 阈值的限制。**仅支持基本类型和字符串类型** |  |
| clusterMode | 是否是集群参数流控规则 | false |
| clusterConfig | 集群流控相关配置 |  |

我们可以通过 ParamFlowRuleManager 的 loadRules 方法更新热点参数规则，下面是一个示例：

ParamFlowRule rule = new ParamFlowRule(resourceName)

.setParamIdx(0)

.setCount(5);

// 针对 int 类型的参数 PARAM\_B，单独设置限流 QPS 阈值为 10，而不是全局的阈值 5.

ParamFlowItem item = new ParamFlowItem().setObject(String.valueOf(PARAM\_B))

.setClassType(int.class.getName())

.setCount(10);

rule.setParamFlowItemList(Collections.singletonList(item));

ParamFlowRuleManager.loadRules(Collections.singletonList(rule));

## 来源访问控制（黑白名单）

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 的黑白名单控制的功能。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。

调用方信息通过 ContextUtil.enter(resourceName, origin) 方法中的 origin 参数传入。

### 规则配置

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

### 示例

比如我们希望控制对资源 test 的访问设置白名单，只有来源为 appA 和 appB 的请求才可通过，则可以配置如下白名单规则：

AuthorityRule rule = new AuthorityRule();

rule.setResource("test");

rule.setStrategy(RuleConstant.AUTHORITY\_WHITE);

rule.setLimitApp("appA,appB");

AuthorityRuleManager.loadRules(Collections.singletonList(rule));

## 注解埋点支持

Sentinel 提供了 @SentinelResource 注解用于定义资源，并提供了 AspectJ 的扩展用于自动定义资源、处理 BlockException 等。使用 Sentinel Annotation AspectJ Extension 的时候需要引入以下依赖：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentinel-annotation-aspectj</artifactId>

<version>x.y.z</version>

</dependency>

### @SentinelResource 注解

@SentinelResource 用于定义资源，并提供可选的异常处理和 fallback 配置项。 @SentinelResource 注解包含以下属性（注意：注解方式埋点不支持 private 方法）：

* value：资源名称，必需项（不能为空）
* entryType：entry 类型，可选项（默认为 EntryType.OUT）
* blockHandler / blockHandlerClass: blockHandler 对应处理 BlockException 的函数名称，可选项。blockHandler 函数访问范围需要是 public，返回类型需要与原方法相匹配，参数类型需要和原方法相匹配并且最后加一个额外的参数，类型为 BlockException。blockHandler 函数默认需要和原方法在同一个类中。若希望使用其他类的函数，则可以指定 blockHandlerClass 为对应的类的 Class 对象，注意对应的函数必需为 static 函数，否则无法解析。
* fallback：fallback 函数名称，可选项，用于在抛出异常的时候提供 fallback 处理逻辑。fallback 函数可以针对所有类型的异常（除了 exceptionsToIgnore 里面排除掉的异常类型）进行处理。fallback 函数签名和位置要求：
  + 返回值类型必须与原函数返回值类型一致；
  + 方法参数列表需要和原函数一致，或者可以额外多一个 Throwable 类型的参数用于接收对应的异常。
  + fallback 函数默认需要和原方法在同一个类中。若希望使用其他类的函数，则可以指定 fallbackClass 为对应的类的 Class 对象，注意对应的函数必需为 static 函数，否则无法解析。
* defaultFallback（since 1.6.0）：默认的 fallback 函数名称，可选项，通常用于通用的 fallback 逻辑（即可以用于很多服务或方法）。默认 fallback 函数可以针对所以类型的异常（除了 exceptionsToIgnore 里面排除掉的异常类型）进行处理。若同时配置了 fallback 和 defaultFallback，则只有 fallback 会生效。defaultFallback 函数签名要求：
  + 返回值类型必须与原函数返回值类型一致；
  + 方法参数列表需要为空，或者可以额外多一个 Throwable 类型的参数用于接收对应的异常。
  + defaultFallback 函数默认需要和原方法在同一个类中。若希望使用其他类的函数，则可以指定 fallbackClass 为对应的类的 Class 对象，注意对应的函数必需为 static 函数，否则无法解析。
* exceptionsToIgnore（since 1.6.0）：用于指定哪些异常被排除掉，不会计入异常统计中，也不会进入 fallback 逻辑中，而是会原样抛出。

特别地，若 blockHandler 和 fallback 都进行了配置，则被限流降级而抛出 BlockException 时只会进入 blockHandler 处理逻辑。若未配置 blockHandler、fallback 和 defaultFallback，则被限流降级时会将 BlockException 直接抛出。（注：1.6.0 之前的版本 fallback 函数只针对降级异常（DegradeException）进行处理，不能针对业务异常进行处理。）

示例：

public class TestService {

// 对应的 `handleException` 函数需要位于 `ExceptionUtil` 类中，并且必须为 static 函数.

@SentinelResource(value = "test", blockHandler = "handleException", blockHandlerClass = {ExceptionUtil.class})

public void test() {

System.out.println("Test");

}

// 原函数

@SentinelResource(value = "hello", blockHandler = "exceptionHandler", fallback = "helloFallback")

public String hello(long s) {

return String.format("Hello at %d", s);

}

// Fallback 函数，函数签名与原函数一致或加一个 Throwable 类型的参数.

public String helloFallback(long s) {

return String.format("Halooooo %d", s);

}

// Block 异常处理函数，参数最后多一个 BlockException，其余与原函数一致.

public String exceptionHandler(long s, BlockException ex) {

// Do some log here.

ex.printStackTrace();

return "Oops, error occurred at " + s;

}

}

从1.4.0 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 1.4.0 以前的版本需要自行调用 Tracer.trace(ex) 来记录业务异常。

## 动态规则扩展

### 规则

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

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。

### DataSource 扩展

上述 loadRules() 方法只接受内存态的规则对象，但更多时候规则存储在文件、数据库或者配置中心当中。DataSource 接口给我们提供了对接任意配置源的能力。相比直接通过 API 修改规则，实现 DataSource 接口是更加可靠的做法。

我们推荐通过控制台设置规则后将规则推送到统一的规则中心，客户端实现 ReadableDataSource 接口端监听规则中心实时获取变更，流程如下：

[DataSource 扩展流程](DataSource%20扩展流程.png)

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

### 拉模式扩展

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。

### 推模式扩展

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。

### 注册数据源

通常需要调用以下方法将数据源注册至指定的规则管理器中：

ReadableDataSource<String, List<FlowRule>> flowRuleDataSource = new NacosDataSource<>(remoteAddress, groupId, dataId, parser);

FlowRuleManager.register2Property(flowRuleDataSource.getProperty());

若不希望手动注册数据源，可以借助 Sentinel 的 InitFunc SPI 扩展接口。只需要实现自己的 InitFunc 接口，在 init 方法中编写注册数据源的逻辑。比如：

public class DataSourceInitFunc implements InitFunc {

@Override

public void init() throws Exception {

final String remoteAddress = "localhost";

final String groupId = "Sentinel:Demo";

final String dataId = "com.alibaba.csp.sentinel.demo.flow.rule";

ReadableDataSource<String, List<FlowRule>> flowRuleDataSource = new NacosDataSource<>(remoteAddress, groupId, dataId,

source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {}));

FlowRuleManager.register2Property(flowRuleDataSource.getProperty());

}

}

接着将对应的类名添加到位于资源目录（通常是 resource 目录）下的 META-INF/services 目录下的 com.alibaba.csp.sentinel.init.InitFunc 文件中，比如：

com.test.init.DataSourceInitFunc

### 示例

#### API 模式

使用客户端规则 API 配置规则。Sentinel Dashboard 通过 Sentinel 客户端自带的规则 API 来实时查询和更改内存中的规则。注意: 要使客户端具备规则 API，需在客户端引入以下依赖：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentienl-http-simple-transport</artifactId>

<version>x.y.z</version>

</dependency>

#### 拉模式

* 使用文件配置规则：FileRefreshableDataSource 会周期性的读取文件以获取规则，当文件有更新时会及时发现，并将规则更新到内存中。使用时只需添加以下依赖：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentinel-datasource-extension</artifactId>

<version>x.y.z</version>

</dependency>

* 使用 Nacos 配置规则：Nacos 是阿里中间件团队开源的服务发现和动态配置中心。Sentinel 针对 Nacos 作了适配，底层可以采用 Nacos 作为规则配置数据源。使用时只需添加以下依赖：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentinel-datasource-nacos</artifactId>

<version>x.y.z</version>

</dependency>

然后创建 NacosDataSource 并将其注册至对应的 RuleManager 上即可。

// remoteAddress 代表 Nacos 服务端的地址

// groupId 和 dataId 对应 Nacos 中相应配置

ReadableDataSource<String, List<FlowRule>> flowRuleDataSource = new NacosDataSource<>(remoteAddress, groupId, dataId,

source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {}));

FlowRuleManager.register2Property(flowRuleDataSource.getProperty());

#### 推模式

* 使用 ZooKeeper 配置规则：Sentinel 针对 ZooKeeper 作了相应适配，底层可以采用 ZooKeeper 作为规则配置数据源。使用时只需添加以下依赖：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentinel-datasource-zookeeper</artifactId>

<version>x.y.z</version>

</dependency>

然后创建 ZookeeperDataSource 并将其注册至对应的 RuleManager 上即可。

// remoteAddress 代表 ZooKeeper 服务端的地址

// path 对应 ZK 中的数据路径

ReadableDataSource<String, List<FlowRule>> flowRuleDataSource = new ZookeeperDataSource<>(remoteAddress, path, source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {}));

FlowRuleManager.register2Property(flowRuleDataSource.getProperty());

* 使用 Apollo 配置规则：Sentinel 针对 Apollo 作了相应适配，底层可以采用 Apollo 作为规则配置数据源。使用时只需添加以下依赖：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentinel-datasource-apollo</artifactId>

<version>x.y.z</version>

</dependency>

然后创建 ApolloDataSource 并将其注册至对应的 RuleManager 上即可。

// namespaceName 对应 Apollo 的命名空间名称

// ruleKey 对应规则存储的 key

// defaultRules 对应连接不上 Apollo 时的默认规则

ReadableDataSource<String, List<FlowRule>> flowRuleDataSource = new ApolloDataSource<>(namespaceName, ruleKey, defaultRules, source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {}));

FlowRuleManager.register2Property(flowRuleDataSource.getProperty());

## 日志

Sentinel 日志目录可通过 csp.sentinel.log.dir 启动参数进行配置。

### 拦截详情日志（block 日志）

无论触发了限流、熔断降级还是系统保护，它们的秒级拦截详情日志都在 ${user\_home}/logs/csp/sentinel-block.log里。如果没有发生拦截，则该日志不会出现。日志格式如下:

2014-06-20 16:35:10|1|sayHello(java.lang.String,long),FlowException,default,origin|61,0

2014-06-20 16:35:11|1|sayHello(java.lang.String,long),FlowException,default,origin|1,0

日志含义：

| **index** | **例子** | **说明** |
| --- | --- | --- |
| 1 | 2014-06-20 16:35:10 | 时间戳 |
| 2 | 1 | 该秒发生的第一个资源 |
| 3 | sayHello(java.lang.String,long) | 资源名称 |
| 4 | XXXException | 拦截的原因, 通常 FlowException 代表是被限流规则拦截，DegradeException 则表示被降级，SystemBlockException 则表示被系统保护拦截 |
| 5 | default | 生效规则的调用来源（参数限流中代表生效的参数） |
| 6 | origin | 被拦截资源的调用者，可以为空 |
| 7 | 61,0 | 61 被拦截的数量，０无意义可忽略 |

### 秒级监控日志

所有的资源访问都会产生秒级监控日志，日志文件默认为 ${user\_home}/logs/csp/${app\_name}-${pid}-metrics.log（会随时间滚动）。格式如下:

1532415661000|2018-07-24 15:01:01|sayHello(java.lang.String)|12|3|4|2|295|0|0|1

1. 1532415661000：时间戳
2. 2018-07-24 15:01:01：格式化之后的时间戳
3. sayHello(java.lang.String)：资源名
4. 12：表示到来的数量，即此刻通过 Sentinel 规则 check 的数量（passed QPS）
5. 3：实际该资源被拦截的数量（blocked QPS）
6. 4：每秒结束的资源个数（完成调用），包括正常结束和异常结束的情况（exit QPS）
7. 2：异常的数量
8. 295：资源的平均响应时间（RT）
9. 0: 该秒占用未来请求的数目（since 1.5.0）
10. 0: 最大并发数（预留用）
11. 1: 资源分类（since 1.7.0）

### 业务日志

其它的日志在 ${user\_home}/logs/csp/sentinel-record.log.xxx 里。该日志包含规则的推送、接收、处理等记录，排查问题的时候会非常有帮助。

### 集群限流日志

${log\_dir}/sentinel-cluster-client.log：Token Client 日志，会记录请求失败的信息

### SPI 扩展机制

1.7.2 版本开始，Sentinel 支持 Logger 扩展机制，可以实现自定义的 Logger SPI 来将 record log 等日志自行处理。metric/block log 暂不支持定制。

## 实时监控

Sentinel 提供对所有资源的实时监控。如果需要实时监控，客户端需引入以下依赖（以 Maven 为例）：

<dependency>

<groupId>com.alibaba.csp</groupId>

<artifactId>sentinel-transport-simple-http</artifactId>

<version>x.y.z</version>

</dependency>

引入上述依赖后，客户端便会主动连接 Sentinel 控制台。通过 Sentinel 控制台 即可查看客户端的实时监控。以下为监控 API 的文档。

## 启动配置项

### 配置方式

Sentinel 提供如下的配置方式：

* JVM -D 参数方式
* properties 文件方式（1.7.0 版本开始支持）

其中，project.name 参数只能通过 JVM -D 参数方式配置（since 1.8.0 取消该限制），其它参数支持所有的配置方式。

**优先级顺序**：JVM -D 参数的优先级最高。若 properties 和 JVM 参数中有相同项的配置，以 JVM 参数配置的为准。

用户可以通过 -Dcsp.sentinel.config.file 参数配置 properties 文件的路径，支持 classpath 路径配置（如 classpath:sentinel.properties）。默认 Sentinel 会尝试从 classpath:sentinel.properties 文件读取配置，读取编码默认为 UTF-8。

### 配置项列表

sentinel-core 的配置项，基础配置项：

| **名称** | **含义** | **类型** | **默认值** | **是否必需** | **备注** |
| --- | --- | --- | --- | --- | --- |
| project.name | 指定应用的名称 | String | null | 否 |  |
| csp.sentinel.app.type | 指定应用的类型 | int | 0 (APP\_TYPE\_COMMON) | 否 | 1.6.0 引入 |
| csp.sentinel.metric.file.single.size | 单个监控日志文件的大小 | long | 52428800 (50MB) | 否 |  |
| csp.sentinel.metric.file.total.count | 监控日志文件的总数上限 | int | 6 | 否 |  |
| csp.sentinel.statistic.max.rt | 最大的有效响应时长（ms），超出此值则按照此值记录 | int | 4900 | 否 | 1.4.1 引入 |
| csp.sentinel.spi.classloader | SPI 加载时使用的 ClassLoader，默认为给定类的 ClassLoader | String | default | 否 | 若配置 context 则使用 thread context ClassLoader。1.7.0 引入 |

其中 project.name 项用于指定应用名（appName）。若未指定，则默认解析 main 函数的类名作为应用名。实际项目使用中建议手动指定应用名。

日志相关配置项：

| **名称** | **含义** | **类型** | **默认值** | **是否必需** | **备注** |
| --- | --- | --- | --- | --- | --- |
| csp.sentinel.log.dir | Sentinel 日志文件目录 | String | ${user.home}/logs/csp/ | 否 | 1.3.0 引入 |
| csp.sentinel.log.use.pid | 日志文件名中是否加入进程号，用于单机部署多个应用的情况 | boolean | false | 否 | 1.3.0 引入 |
| csp.sentinel.log.output.type | Record 日志输出的类型，file 代表输出至文件，console 代表输出至终端 | String | file | 否 | 1.6.2 引入 |

注意：若需要在单台机器上运行相同服务的多个实例，则需要加入 -Dcsp.sentinel.log.use.pid=true 来保证不同实例日志的独立性。

sentinel-transport-common 的配置项：

| **名称** | **含义** | **类型** | **默认值** | **是否必需** |
| --- | --- | --- | --- | --- |
| csp.sentinel.dashboard.server | 控制台的地址，指定控制台后客户端会自动向该地址发送心跳包。地址格式为：hostIp:port | String | null | 是 |
| csp.sentinel.heartbeat.interval.ms | 心跳包发送周期，单位毫秒 | long | null | 非必需，若不进行配置，则会从相应的 HeartbeatSender 中提取默认值 |
| csp.sentinel.api.port | 本地启动 HTTP API Server 的端口号 | int | 8719 | 否 |
| csp.sentinel.heartbeat.client.ip | 指定心跳包中本机的 IP | String | - | 若不指定则通过 HostNameUtil 解析；该配置项多用于多网卡环境 |

注：csp.sentinel.api.port 可不提供，默认为 8719，若端口冲突会自动向下探测可用的端口。