

# Sentinel源码分析

## 1.Sentinel的基本概念

Sentinel实现限流、隔离、降级、熔断等功能，本质要做的就是两件事情：

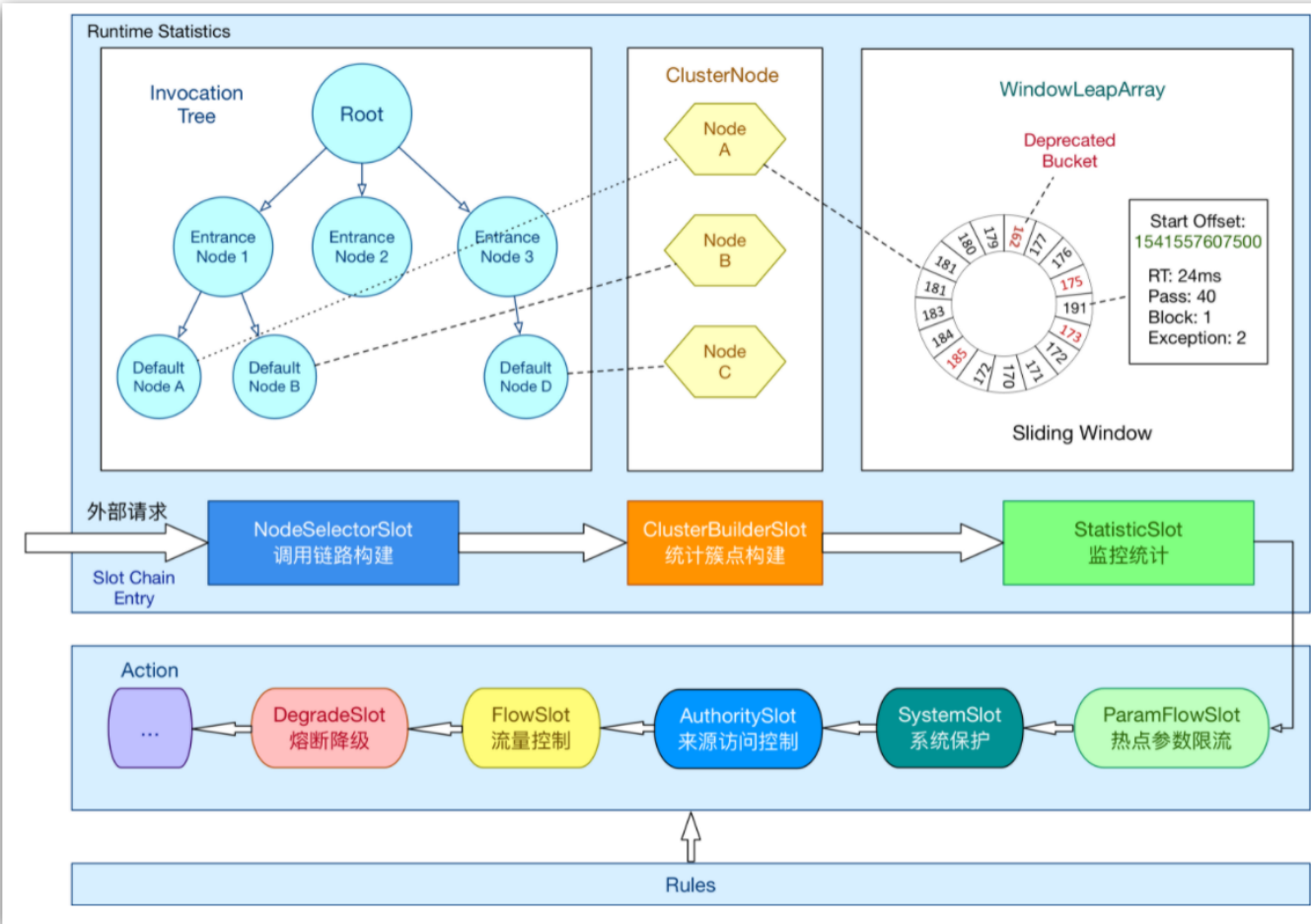
- 统计数据：统计某个资源的访问数据（QPS、RT等信息）
- 规则判断：判断限流规则、隔离规则、降级规则、熔断规则是否满足

这里的资源就是希望被Sentinel保护的业务，例如项目中定义的controller方法就是默认被Sentinel保护的资源。

### 1.1.ProcessorSlotChain

实现上述功能的核心骨架是一个叫做ProcessorSlotChain的类。这个类基于责任链模式来设计，将不同的功能（限流、降级、系统保护）封装为一个个的Slot，请求进入后逐个执行即可。

其工作流如图：



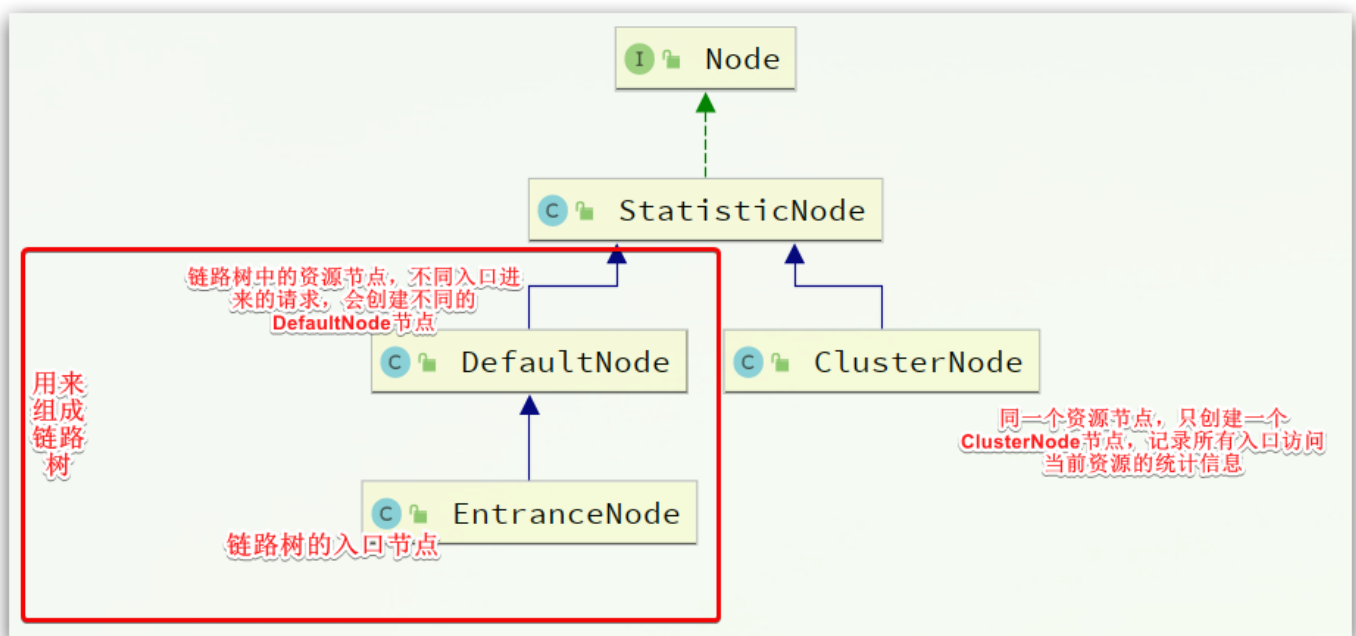
责任链中的Slot也分为两大类：

- 统计数据构建部分（statistic）
  - **NodeSelectorSlot**：负责构建簇点链路中的节点（DefaultNode），将这些节点形成链路树

- ClusterBuilderSlot: 负责构建某个资源的ClusterNode, ClusterNode可以保存资源的运行信息（响应时间、QPS、block 数目、线程数、异常数等）以及来源信息（origin名称）
- StatisticSlot: 负责统计实时调用数据，包括运行信息、来源信息等
- 规则判断部分（rule checking）
  - AuthoritySlot: 负责授权规则（来源控制）
  - SystemSlot: 负责系统保护规则
  - ParamFlowSlot: 负责热点参数限流规则
  - FlowSlot: 负责限流规则
  - DegradeSlot: 负责降级规则

## 1.2.Node

Sentinel中的簇点链路是由一个个的Node组成的，Node是一个接口，包括下面的实现：



所有的节点都可以记录对资源的访问统计数据，所以都是StatisticNode的子类。

按照作用分为两类Node：

- DefaultNode: 代表链路树中的每一个资源，一个资源出现在不同链路中时，会创建不同的DefaultNode节点。而树的入口节点叫EntranceNode，是一种特殊的DefaultNode
- ClusterNode: 代表资源，一个资源不管出现在多少链路中，只会有一个ClusterNode。记录的是当前资源被访问的所有统计数据之和。

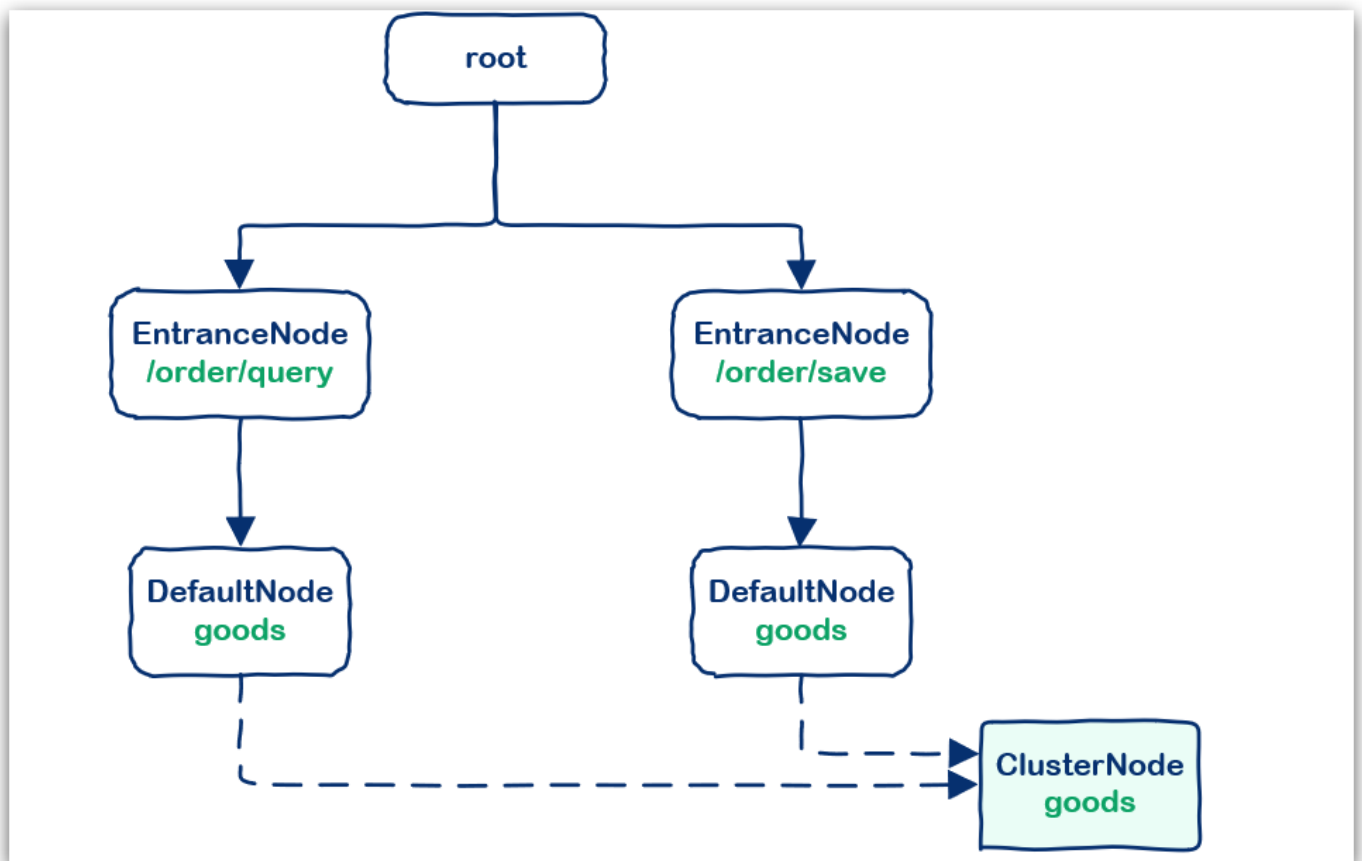
DefaultNode记录的是资源在当前链路中的访问数据，用来实现基于链路模式的限流规则。ClusterNode记录的是资源在所有链路中的访问数据，实现默认模式、关联模式的限流规则。

例如：我们在一个SpringMVC项目中，有两个业务：

- 业务1: controller中的资源 `/order/query` 访问了service中的资源 `/goods`

- 业务2: controller中的资源 `/order/save` 访问了service中的资源 `/goods`

创建的链路图如下:



## 1.3.Entry

默认情况下, Sentinel会将controller中的方法作为被保护资源, 那么问题来了, 我们该如何将自己的一段代码标记为一个Sentinel的资源呢?

Sentinel中的资源用Entry来表示。声明Entry的API示例:

```
// 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
}
```

### 1.3.1.自定义资源

例如, 我们在order-service服务中, 将 `OrderService` 的 `queryOrderById()` 方法标记为一个资源。

1) 首先在order-service中引入sentinel依赖

```

<!--sentinel-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>

```

## 2) 然后配置Sentinel地址

```

spring:
  cloud:
    sentinel:
      transport:
        dashboard: localhost:8089 # 这里我的sentinel用了8089的端口

```

## 3) 修改OrderService类的queryOrderById方法

代码这样来实现：

```

public Order queryOrderById(Long orderId) {
    // 创建Entry，标记资源，资源名为resource1
    try (Entry entry = SphU.entry("resource1")) {
        // 1.查询订单，这里是假数据
        Order order = Order.build(101L, 4999L, "小米 MIX4", 1, 1L, null);
        // 2.查询用户，基于Feign的远程调用
        User user = userClient.findById(order.getUserId());
        // 3.设置
        order.setUser(user);
        // 4.返回
        return order;
    } catch (BlockException e) {
        log.error("被限流或降级", e);
        return null;
    }
}

```

## 4) 访问

打开浏览器，访问order服务：`http://localhost:8080/order/101`

然后打开sentinel控制台，查看簇点链路

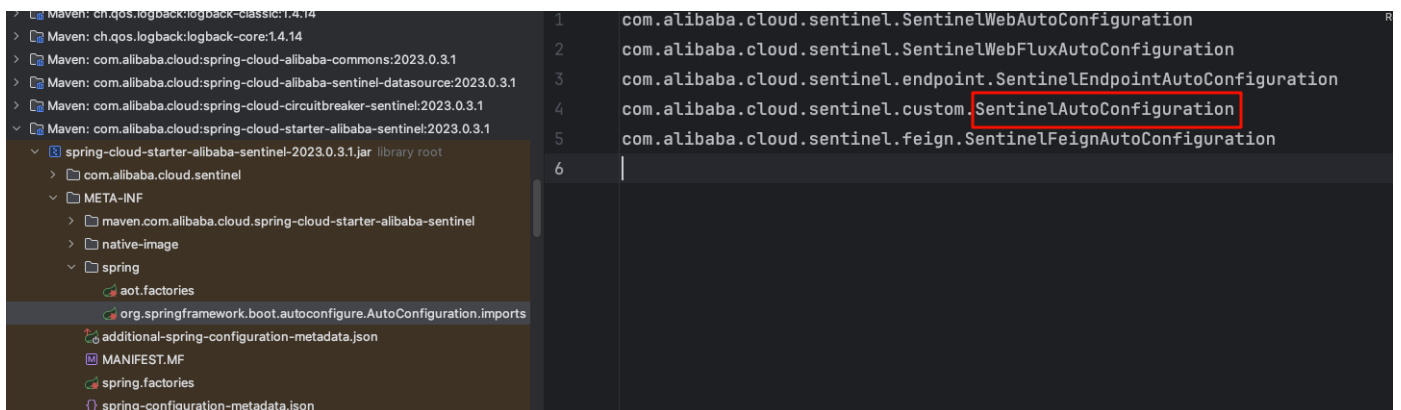
order-service								树状视图	列表视图
簇点链路		192.168.150.1:8720		关键字		刷新			
资源名	通过QPS	拒绝QPS	线程数	平均RT	分钟通过	分钟拒绝	操作		
▼ sentinel_spring_web_context <span>默认入口</span>	0	0	0	0	1	0	+ 流控	+ 降级	+ 热点
▼ /order/{orderId} <span>controller中的资源</span>	0	0	0	0	1	0	+ 流控	+ 降级	+ 热点
resource1 <span>OrderService中的自定义资源</span>	0	0	0	0	1	0	+ 流控	+ 降级	+ 热点

### 1.3.2.基于注解标记资源

在之前学习Sentinel的时候，我们知道可以通过给方法添加@SentinelResource注解的形式来标记资源。这个是怎么实现的呢？

```
@SentinelResource("resource1")
public Order queryOrderByid(Long orderId) {
    // 1. 查询订单，这里是假数据
    Order order = Order.build(id: 101L, price: 4999L, name: "小米 MIX4", num: 1, userId: 1L, user: null);
    // 2. 查询用户，基于Feign的远程调用
    User user = userClient.findById(order.getUserId());
    // 3. 设置
    order.setUser(user);
    // 4. 返回
    return order;
}
```

来看下我们引入的Sentinel依赖包，其中的spring.factories声明需要就是自动装配的配置类，扩展部分在 `spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`，内容如下：



我们来看下 `SentinelAutoConfiguration` 这个类：

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnProperty(name = "spring.cloud.sentinel.enabled", matchIfMissing = true)
@EnableConfigurationProperties(SentinelProperties.class)
public class SentinelAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public SentinelResourceAspect sentinelResourceAspect() {
        return new SentinelResourceAspect();
    }
}
```

可以看到，在这里声明了一个Bean，`SentinelResourceAspect`：

```

@Aspect
public class SentinelResourceAspect extends AbstractSentinelAspectSupport {
    //定义切入点，扫描具有SentinelResource注解的方法
    @Pointcut("@annotation(com.alibaba.csp.sentinel.annotation.SentinelResource)")
    public void sentinelResourceAnnotationPointcut() {
    }
    //环绕通知
    @Around("sentinelResourceAnnotationPointcut()")
    public Object invokeResourceWithSentinel(ProceedingJoinPoint pjp) throws Throwable {
        Method originMethod = resolveMethod(pjp); //获取带有注解的方法
        SentinelResource annotation =
originMethod.getAnnotation(SentinelResource.class); //获取注解信息
        if (annotation == null) {
            // Should not go through here.
            throw new IllegalStateException("Wrong state for SentinelResource
annotation");
        }
        String resourceName = getResourceName(annotation.value(), originMethod);
        EntryType entryType = annotation.entryType();
        int resourceType = annotation.resourceType();
        Entry entry = null;
        try {
            entry = SphU.entry(resourceName, resourceType, entryType, pjp.getArgs()); //实
际起作用的地方，先创建资源
            return pjp.proceed(); //执行方法
        } catch (BlockException ex) {
            //.....
        }
    }
}

```

简单来说，@SentinelResource注解就是一个标记，而Sentinel基于AOP思想，对被标记的方法做环绕增强，完成资源（Entry）的创建。

## 1.4.Context

上一节，我们发现簇点链路中除了controller方法、service方法两个资源外，还多了一个默认的入口节点：sentinel\_spring\_web\_context，是一个EntranceNode类型的节点，这个节点是在初始化Context的时候由Sentinel帮我们创建的。

### 1.4.1.什么是Context

那么，什么是Context呢？

- Context 代表调用链路上下文，贯穿一次调用链路中的所有资源（Entry），基于ThreadLocal。
- Context 维持着入口节点（entranceNode）、本次调用链路的 curNode（当前资源节点）、调用来源（origin）等信息。
- 后续的Slot都可以通过Context拿到DefaultNode或者ClusterNode，从而获取统计数据，完成规则判断
- Context初始化的过程中，会创建EntranceNode，contextName就是EntranceNode的名称

对应的API如下：

```
// 创建context，包含两个参数：context名称、来源名称
ContextUtil.enter("contextName", "originName");
```

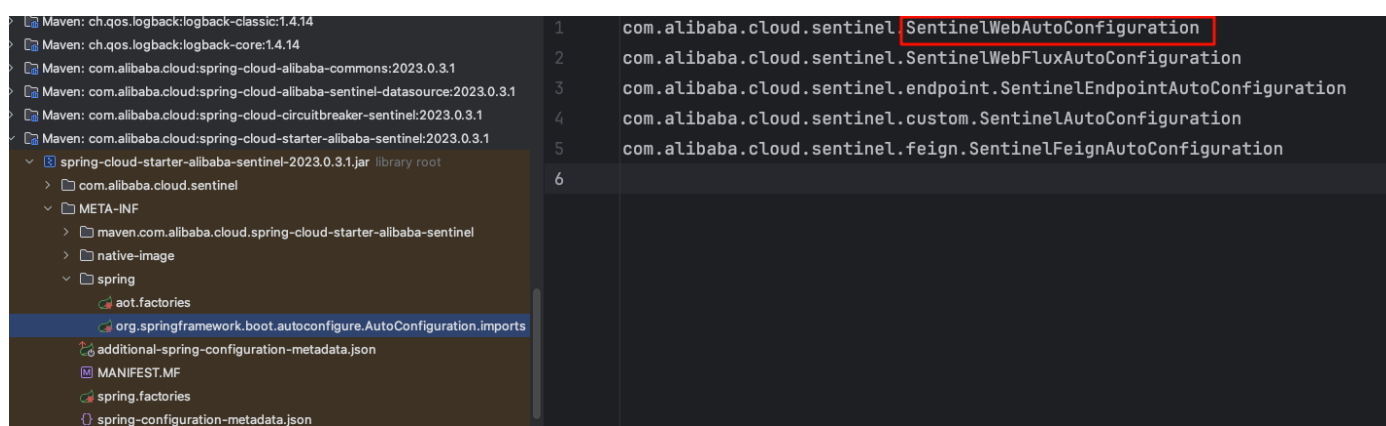
## 1.4.2.Context的初始化

那么这个Context又是在何时完成初始化的呢？

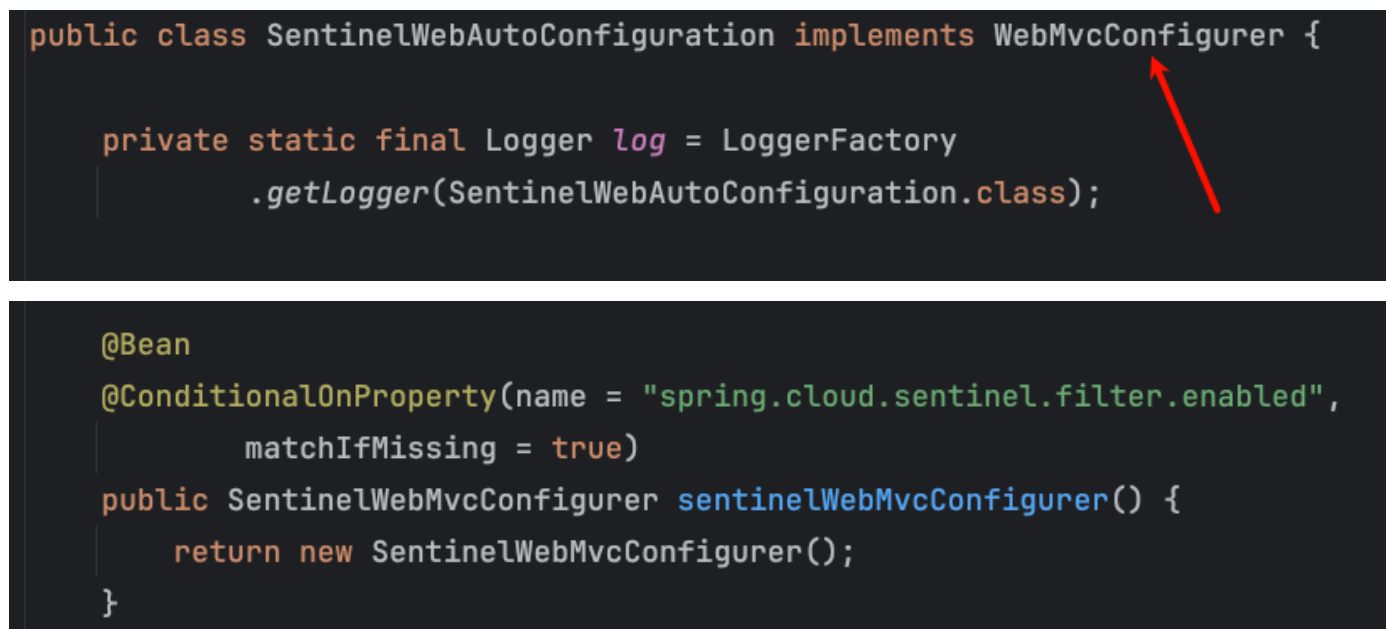
### 1.4.2.1.自动装配

来看下我们引入的Sentinel依赖包,其中的

spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports 声明需要就是自动装配的配置类，内容如下



我们先看SentinelWebAutoConfiguration这个类：



这个类中创建了Bean:SentinelWebMvcConfigurer实现了WebMvcConfigurer，我们知道这个是SpringMVC自定义配置用到的类，可以配HandlerInterceptor：

```
public class SentinelWebMvcConfigurer implements WebMvcConfigurer {
    private static final Logger log =
        LoggerFactory.getLogger(SentinelWebMvcConfigurer.class);
```



```

@Autowired
private SentinelProperties sentinelProperties;
@Autowired
private Optional<SentinelWebInterceptor> sentinelWebInterceptorOptional;
@Override
public void addInterceptors(InterceptorRegistry registry) {
    if (!sentinelWebInterceptorOptional.isPresent()) {
        return;
    }
    SentinelProperties.Filter filterConfig = sentinelProperties.getFilter();
    registry.addInterceptor(sentinelWebInterceptorOptional.get())
        .order(filterConfig.getOrder())
        .addPathPatterns(filterConfig.getUrlPatterns());
}
}

```

可以看到这里配置了一个 `SentinelWebInterceptor` 的拦截器。`SentinelWebInterceptor` 的声明如下：

```

public class SentinelWebInterceptor extends AbstractSentinelInterceptor {}
public abstract class AbstractSentinelInterceptor implements HandlerInterceptor {}
//HandlerInterceptor的方法
public interface HandlerInterceptor {
    default boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        return true;
    }
    default void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, @Nullable ModelAndView modelAndView) throws Exception {
    }
    default void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler, @Nullable Exception ex) throws Exception {
    }
}
}

```

`SentinelWebInterceptor` 继承了，`AbstractSentinelInterceptor`，实现了 `HandlerInterceptor`。

`HandlerInterceptor` 拦截器会拦截一切进入controller的方法，执行 `preHandle` 前置拦截方法，而Context的初始化就是在这里完成的。

### 1.4.2.2.AbstractSentinelInterceptor

`HandlerInterceptor` 拦截器会拦截一切进入controller的方法，执行 `preHandle` 前置拦截方法，而Context的初始化就是在这里完成的。

我们来看看这个类的 `preHandle` 实现：

```

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler) throws Exception {
    String resourceName = "";
    try {
        // 获取资源名称，一般是controller方法的@RequestMapping路径，例如/order/{orderId}
    }
}

```



```

        resourceName = getResourceName(request);
        if (StringUtil.isEmpty(resourceName)) {
            return true;
        }
        if (increaseReference(request, this.baseWebMvcConfig.getRequestRefName(), 1) !=
1) {
            return true;
        }
        // Parse the request origin using registered origin parser.
        // 从request中获取请求来源，将来做 授权规则 判断时会用
        String origin = parseOrigin(request);
        // 获取 contextName，默认是sentinel_spring_web_context
        String contextName = getContextName(request);
        // 创建 Context
        ContextUtil.enter(contextName, origin);
        // 创建资源，名称就是当前请求的controller方法的映射路径
        Entry entry = SphU.entry(resourceName, ResourceTypeConstants.COMMON_WEB,
EntryType.IN);
        request.setAttribute(baseWebMvcConfig.getRequestAttributeName(), entry);
        return true;
    } catch (BlockException e) {
        //xxx
    }
}

```

### 1.4.2.3.ContextUtil

创建Context的方法就是 `ContextUtil.enter(contextName, origin)`；我们进入该方法：

```

public static Context enter(String name, String origin) {
    if (Constants.CONTEXT_DEFAULT_NAME.equals(name)) {
        throw new ContextNameDefineException(
            "The " + Constants.CONTEXT_DEFAULT_NAME + " can't be permit to defined!");
    }
    return trueEnter(name, origin);
}

```

进入 `trueEnter` 方法：

```

protected static Context trueEnter(String name, String origin) {
    // 尝试获取context
    Context context = contextHolder.get();
    // 判空
    if (context == null) {
        // 如果为空，开始初始化
        Map<String, DefaultNode> localCacheNameMap = contextNameNodeMap;
        // 尝试获取入口节点
        DefaultNode node = localCacheNameMap.get(name);
        if (node == null) {
            LOCK.lock();
            try {

```

```

        node = contextNameNodeMap.get(name);
        if (node == null) {
            // 入口节点为空, 初始化入口节点 EntranceNode
            node = new EntranceNode(new StringResourceWrapper(name, EntryType.IN),
null);

            // 添加入口节点到 ROOT
            Constants.ROOT.addChild(node);
            // 将入口节点放入缓存
            Map<String, DefaultNode> newMap = new HashMap<>
(contextNameNodeMap.size() + 1);
            newMap.putAll(contextNameNodeMap);
            newMap.put(name, node);
            contextNameNodeMap = newMap;
        }
    } finally {
        LOCK.unlock();
    }
}
// 创建Context, 参数为: 入口节点 和 contextName
context = new Context(node, name);
// 设置请求来源 origin
context.setOrigin(origin);
// 放入ThreadLocal
contextHolder.set(context);
}
// 返回
return context;
}

```

## 2.ProcessorSlotChain执行流程

接下来我们跟踪源码, 验证下ProcessorSlotChain的执行流程。

### 2.1.入口

首先, 回到一切的入口, AbstractSentinelInterceptor 类的 preHandle 方法:

```

public abstract class AbstractSentinelInterceptor implements HandlerInterceptor {
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
        // Parse the request origin using registered origin parser.
        String origin = parseOrigin(request);
        String contextName = getContextName(request);
        ContextUtil.enter(contextName, origin);
        Entry entry = SphU.entry(resourceName, ResourceTypeConstants.COMMON_WEB, EntryType.IN);
        request.setAttribute(baseWebMvcConfig.getRequestAttributeName(), entry);
        return true;
}

```

还有, SentinelResourceAspect 的环绕增强方法:

```

public class SentinelResourceAspect extends AbstractSentinelAspectSupport {
    public Object invokeResourceWithSentinel(ProceedingJoinPoint pjp) throws Throwable {

        String resourceName = getResourceName(annotation.value(), originMethod);
        EntryType entryType = annotation.entryType();
        int resourceType = annotation.resourceType();
        Entry entry = null;
        try {
            entry = SphU.entry(resourceName, resourceType, entryType, pjp.getArgs());
            return pjp.proceed();
        } catch (BlockException ex) {
            return handleBlockException(pjp, annotation, ex);
        }
    }
}

```

可以看到，任何一个资源必定要执行 `SphU.entry()` 这个方法，逐步跟下去：

```

//SphU.entry
public static Entry entry(String name, int resourceType, EntryType trafficType, Object[]
args) throws BlockException {
    return Env.sph.entryWithType(name, resourceType, trafficType, 1, args);
}

//CtSph.entryWithType
public Entry entryWithType(String name, int resourceType, EntryType entryType, int count,
Object[] args) throws BlockException {
    return this.entryWithType(name, resourceType, entryType, count, false, args);
}

//CtSph.entryWithType
public Entry entryWithType(String name, int resourceType, EntryType entryType, int count,
boolean prioritized, Object[] args) {
    // 将 资源名称等基本信息 封装为一个 StringResourceWrapper对象
    StringResourceWrapper resource = new StringResourceWrapper(name, entryType,
resourceType);
    return this.entryWithPriority(resource, count, prioritized, args);
}

//CtSph.entryWithPriority
private Entry entryWithPriority(ResourceWrapper resourceWrapper, int count, boolean
prioritized, Object... args) {
    //获取上下文
    Context context = ContextUtil.getContext();
    if (context instanceof NullContext) {
        return new CtEntry(resourceWrapper, (ProcessorSlot)null, context);
    } else {
        if (context == null) {
            context = CtSph.InternalContextUtil.internalEnter("sentinel_default_context");
        }
        if (!Constants.ON) {
            return new CtEntry(resourceWrapper, (ProcessorSlot)null, context);
        } else {
            // 获取 slot执行链，同一个资源，会创建一个执行链，放入缓存
            ProcessorSlot<Object> chain = this.lookProcessChain(resourceWrapper);
            if (chain == null) {
                return new CtEntry(resourceWrapper, (ProcessorSlot)null, context);
            } else {
                // 创建 Entry，并将 resource、chain、context 记录在 Entry中
                Entry e = new CtEntry(resourceWrapper, chain, context, count, args);
            }
        }
    }
}

```

```

        try {
            // 执行 slotChain
            chain.entry(context, resourceWrapper, (Object)null, count,
prioritized, args);
        } catch (BlockException e1) {
            e.exit(count, args);
            throw e1;
        } catch (Throwable e1) {
            RecordLog.info("Sentinel unexpected exception", e1);
        }
        return e;
    }
}
}
}
}

```

在这段代码中，会获取 `ProcessorSlotChain` 对象，然后基于 `chain.entry()` 开始执行 `slotChain` 中的每一个 `Slot`。而这里创建的是其实现类：`DefaultProcessorSlotChain`。具体创建时候，会 `build` 一个 `chain`，就是

`DefaultProcessorSlotChain`

```

ProcessorSlot<Object> chain = lookProcessChain(resourceWrapper); //跟进去 会直行道下面
public class DefaultSlotChainBuilder implements SlotChainBuilder {
    @Override
    public ProcessorSlotChain build() {
        ProcessorSlotChain chain = new DefaultProcessorSlotChain();
        //这里比较关键的地方是：list是排序好的，会按照顺序加进去。loadInstanceListSorted->load排序
        跟进去会发现是按照AbstractLinkedProcessorSlot实现类上的Spi注解进行排序的，这个决定了执行链的顺序。其中DefaultProcessorSlotChain 不带顺序，获取的first就是selectNode槽
        List<ProcessorSlot> sortedSlotList =
SpiLoader.of(ProcessorSlot.class).loadInstanceListSorted();
        for (ProcessorSlot slot : sortedSlotList) {
            if (!(slot instanceof AbstractLinkedProcessorSlot)) {
                RecordLog.warn("The ProcessorSlot(" + slot.getClass().getCanonicalName() +
") is not an instance of AbstractLinkedProcessorSlot, can't be added into
ProcessorSlotChain");
                continue;
            }
            chain.addLast((AbstractLinkedProcessorSlot<?>) slot);
        }
        return chain;
    }
}
}

```

然后将获取的 `ProcessorSlotChain` 以后保存到一个 `Map` 中，`key` 是 `ResourceWrapper`，`value` 是 `ProcessorSlotChain`。其中：`ResourceWrapper` 的 `hashCode` 方法进行了重写，是 `name.hashCode()`。所以，一个资源只会会有一个 `ProcessorSlotChain`。

## 2.2.DefaultProcessorSlotChain

然后我们进入 `DefaultProcessorSlotChain` 的 `entry` 方法开始执行：

```

@Override
public void entry(Context context, ResourceWrapper resourceWrapper, Object t, int count,
boolean prioritized, Object... args)
    throws Throwable {
    // first, 就是责任链中的第一个 slot
    first.transformEntry(context, resourceWrapper, t, count, prioritized, args);
}

```

这里的first，类型是AbstractLinkedProcessorSlot：

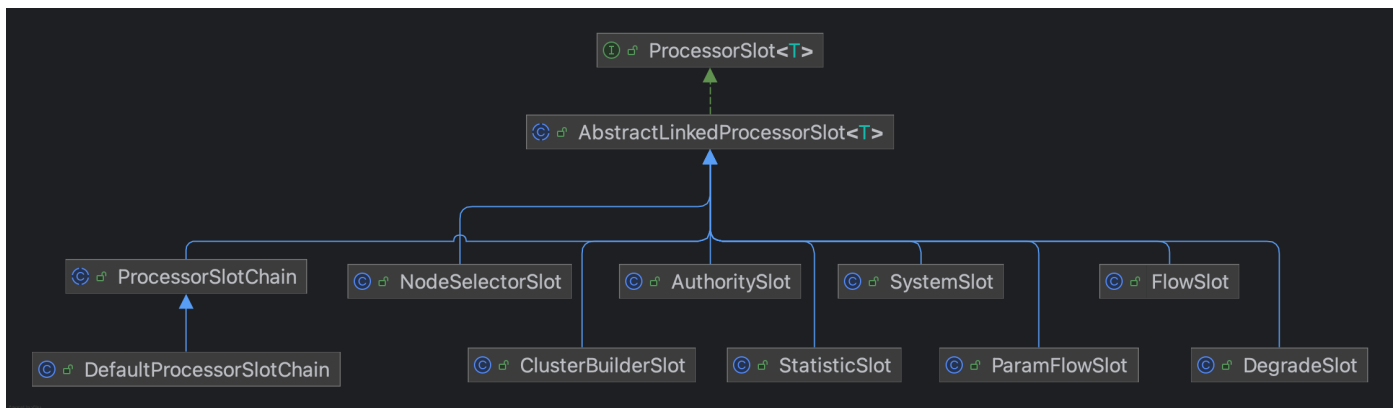
```

public class DefaultProcessorSlotChain extends ProcessorSlotChain {

    AbstractLinkedProcessorSlot<?> first = new AbstractLinkedProcessorSlot<Object>() {

```

看下继承关系：



```

public abstract class AbstractLinkedProcessorSlot<T> implements ProcessorSlot<T> {

    Choose Implementation of AbstractLinkedProcessorSlot (13 found)

    private AbstractLinkedProcessorSlot<T> next;

    @Override
    public void fireEntry(Context context, ResourceWrapper resourceWrapper, Object t, int count,
        throws Throwable {
        if (next != null) {
            next.transformEntry(context, resourceWrapper, t, count, prioritized, args);
        }
    }

    /unchecked/
    void transformEntry(Context context, ResourceWrapper resourceWrapper, Object t, int count,

```

因此，first一定是这些实现类中的一个，按照最早讲的责任链顺序，first应该就是 NodeSelectorSlot。

不过，既然是基于责任链模式，所以这里只要记住下一个slot就可以了，也就是next：

```

public abstract class AbstractLinkedProcessorSlot<T> implements ProcessorSlot<T> {

    private AbstractLinkedProcessorSlot<?> next = null;

    @Override
    public void fireEntry(Context context, ResourceWrapper resourceWrapper, Object obj, int count, boolean prioritized, Object... args)
        throws Throwable {
        if (next != null) {
            next.transformEntry(context, resourceWrapper, obj, count, prioritized, args);
        }
    }
}

```

next确实是NodeSelectSlot类型。

而NodeSelectSlot的next一定是ClusterBuilderSlot，依次类推：

```

this = {DefaultProcessorSlotChain$1@8343}
  > f this$0 = {DefaultProcessorSlotChain@8276}
  > f next = {NodeSelectorSlot@8304}
    > f map = {HashMap@8387} size = 1
    > f next = {ClusterBuilderSlot@8306}
      > f clusterNode = {ClusterNode@8459}
      > f next = {LogSlot@8307}
        > f next = {StatisticSlot@8482}
          > f next = {AuthoritySlot@8483}
            > f next = {SystemSlot@8484}
              > f next = {ParamFlowSlot@8485}
                > f next = {FlowSlot@8486}
                  > f checker = {FlowRuleChecker@8487}
                  > f ruleProvider = {FlowSlot$1@8488}
                  > f next = {DegradeSlot@8315}
                    f next = null

```

责任链就建立起来了。

## 2.3.NodeSelectorSlot

NodeSelectorSlot负责构建簇点链路中的节点（DefaultNode），将这些节点形成链路树。核心代码：

```

@Override
public void entry(Context context, ResourceWrapper resourceWrapper, Object obj, int count,
    boolean prioritized, Object... args)
    throws Throwable {
    // 尝试获取 当前资源的 DefaultNode
    DefaultNode node = map.get(context.getName());
    if (node == null) {
        synchronized (this) {
            node = map.get(context.getName());
            if (node == null) {
                // 如果为空，为当前资源创建一个新的 DefaultNode
                node = new DefaultNode(resourceWrapper, null);
            }
        }
    }
}

```

```

        HashMap<String, DefaultNode> cacheMap = new HashMap<String, DefaultNode>
(map.size());
        cacheMap.putAll(map);
        // 放入缓存中, 注意这里的 key是contextName,
        // 这样不同链路进入相同资源, 就会创建多个 DefaultNode
        cacheMap.put(context.getName(), node);
        map = cacheMap;
        // 当前节点加入上一节点的 child中, 这样就构成了调用链路树
        ((DefaultNode) context.getLastNode()).addChild(node);
    }

    }
}
// context中的curNode (当前节点) 设置为新的 node
context.setCurNode(node);
// 执行下一个 slot
fireEntry(context, resourceWrapper, node, count, prioritized, args);
}

```

这个Slot完成了这么几件事情:

- 为当前资源创建 DefaultNode
- 将DefaultNode放入缓存中, key是contextName, 这样不同链路入口的请求, 将会创建多个DefaultNode, 相同链路则只有一个DefaultNode
- 将当前资源的DefaultNode设置为上一个资源的childNode
- 将当前资源的DefaultNode设置为Context中的curNode (当前节点)

下一个slot, 就是ClusterBuilderSlot

## 2.4.ClusterBuilderSlot

ClusterBuilderSlot负责构建某个资源的ClusterNode, 核心代码:

```

@Override
public void entry(Context context, ResourceWrapper resourceWrapper, DefaultNode node,
    int count, boolean prioritized, Object... args)
    throws Throwable {
    // 判空, 注意ClusterNode是共享的成员变量, 也就是说一个资源只有一个ClusterNode, 与链路无关
    if (clusterNode == null) {
        synchronized (lock) {
            if (clusterNode == null) {
                // 创建 cluster node.
                clusterNode = new ClusterNode(resourceWrapper.getName(),
resourceWrapper.getResourceType());
                HashMap<ResourceWrapper, ClusterNode> newMap = new HashMap<>
(Math.max(clusterNodeMap.size(), 16));
                newMap.putAll(clusterNodeMap);
                // 放入缓存, 可以是nodeId, 也就是resource名称
                newMap.put(node.getId(), clusterNode);
                clusterNodeMap = newMap;
            }
        }
    }
}

```



```

        }
    }
}
// 将资源的 DefaultNode与 ClusterNode关联
node.setClusterNode(clusterNode);
// 记录请求来源 origin 将 origin放入 entry
if (!"".equals(context.getOrigin())) {
    Node originNode =
node.getClusterNode().getOrCreateOriginNode(context.getOrigin());
    context.getCurEntry().setOriginNode(originNode);
}
// 继续下一个slot
fireEntry(context, resourceWrapper, node, count, prioritized, args);
}

```

## 2.5.StatisticSlot

StatisticSlot负责统计实时调用数据，包括运行信息（访问次数、线程数）、来源信息等。

StatisticSlot是实现限流的关键，其中基于滑动时间窗口算法维护了计数器，统计进入某个资源的请求次数。

核心代码：

```

@Override
public void entry(Context context, ResourceWrapper resourceWrapper, DefaultNode node,
    int count, boolean prioritized, Object... args) throws Throwable {
    try {
        // 放行到下一个 slot，做限流、降级等判断
        fireEntry(context, resourceWrapper, node, count, prioritized, args);

        // 请求通过了，线程计数器 +1 ，用作线程隔离
        node.increaseThreadNum();
        // 请求计数器 +1 用作限流
        node.addPassRequest(count);

        if (context.getCurEntry().getOriginNode() != null) {
            // 如果有 origin，来源计数器也都要 +1
            context.getCurEntry().getOriginNode().increaseThreadNum();
            context.getCurEntry().getOriginNode().addPassRequest(count);
        }

        if (resourceWrapper.getEntryType() == EntryType.IN) {
            // 如果是入口资源，还要给全局计数器 +1.
            Constants.ENTRY_NODE.increaseThreadNum();
            Constants.ENTRY_NODE.addPassRequest(count);
        }

        // 请求通过后的回调.
        for (ProcessorSlotEntryCallback<DefaultNode> handler :
StatisticSlotCallbackRegistry.getEntryCallbacks()) {
            handler.onPass(context, resourceWrapper, node, count, args);
        }
    }
}

```

```
    } catch (Throwable e) {  
        // 各种异常处理就省略了。。。  
        context.getCurEntry().setError(e);  
  
        throw e;  
    }  
}
```

另外，需要注意的是，所有的计数+1动作都包括两部分，以 `node.addPassRequest(count);` 为例：

```
@Override  
public void addPassRequest(int count) {  
    // DefaultNode的计数器，代表当前链路的 计数器  
    super.addPassRequest(count);  
    // ClusterNode计数器，代表当前资源的 总计数器  
    this.clusterNode.addPassRequest(count);  
}
```

具体计数方式，我们后续再看。

接下来，进入规则校验的相关slot了，依次是：

- AuthoritySlot：负责授权规则（来源控制）
- SystemSlot：负责系统保护规则
- ParamFlowSlot：负责热点参数限流规则
- FlowSlot：负责限流规则
- DegradeSlot：负责降级规则

## 2.6.AuthoritySlot

负责请求来源origin的授权规则判断，如图：

新增授权规则

资源名

resource1

流控应用

gateway

授权类型

☒ 白名单 ☐ 黑名单

新增并继续添加

新增

取消

核心API:

```
@Override
public void entry(Context context, ResourceWrapper resourceWrapper, DefaultNode node, int
count, boolean prioritized, Object... args)
    throws Throwable {
    // 校验黑白名单
    checkBlackWhiteAuthority(resourceWrapper, context);
    // 进入下一个 slot
    fireEntry(context, resourceWrapper, node, count, prioritized, args);
}
```

黑白名单校验的逻辑:

```
void checkBlackWhiteAuthority(ResourceWrapper resource, Context context) throws
AuthorityException {
    // 获取授权规则
    Map<String, Set<AuthorityRule>> authorityRules =
    AuthorityRuleManager.getAuthorityRules();

    if (authorityRules == null) {
        return;
    }

    Set<AuthorityRule> rules = authorityRules.get(resource.getName());
    if (rules == null) {
        return;
    }
    // 遍历规则并判断
    for (AuthorityRule rule : rules) {
        if (!AuthorityRuleChecker.passCheck(rule, context)) {
            // 规则不通过, 直接抛出异常
        }
    }
}
```

```

        throw new AuthorityException(context.getOrigin(), rule);
    }
}
}

```

再看下 `AuthorityRuleChecker.passCheck(rule, context)` 方法:

```

static boolean passCheck(AuthorityRule rule, Context context) {
    // 得到请求来源 origin
    String requester = context.getOrigin();

    // 来源为空, 或者规则为空, 都直接放行
    if (StringUtil.isEmpty(requester) || StringUtil.isEmpty(rule.getLimitApp())) {
        return true;
    }

    // rule.getLimitApp()得到的就是 白名单 或 黑名单 的字符串, 这里先用 indexOf方法判断
    int pos = rule.getLimitApp().indexOf(requester);
    boolean contain = pos > -1;

    if (contain) {
        // 如果包含 origin, 还要进一步做精确判断, 把名单列表以","分割, 逐个判断
        boolean exactlyMatch = false;
        String[] appArray = rule.getLimitApp().split(",");
        for (String app : appArray) {
            if (requester.equals(app)) {
                exactlyMatch = true;
                break;
            }
        }
        contain = exactlyMatch;
    }
    // 如果是黑名单, 并且包含origin, 则返回false
    int strategy = rule.getStrategy();
    if (strategy == RuleConstant.AUTHORITY_BLACK && contain) {
        return false;
    }
    // 如果是白名单, 并且不包含origin, 则返回false
    if (strategy == RuleConstant.AUTHORITY_WHITE && !contain) {
        return false;
    }
    // 其它情况返回true
    return true;
}

```

## 2.7.SystemSlot

SystemSlot是对系统保护的规则校验：

新增系统保护规则

阈值类型

☒ LOAD ☐ RT ☐ 线程数 ☐ 入口 QPS ☐ CPU 使用率

阈值

[0, ~)的正整数

新增

取消

核心API：

```
@Override
public void entry(Context context, ResourceWrapper resourceWrapper, DefaultNode node,
    int count,boolean prioritized, Object... args) throws Throwable {
    // 系统规则校验
    SystemRuleManager.checkSystem(resourceWrapper);
    // 进入下一个 slot
    fireEntry(context, resourceWrapper, node, count, prioritized, args);
}
```

来看下 `SystemRuleManager.checkSystem(resourceWrapper);` 的代码：

```
public static void checkSystem(ResourceWrapper resourceWrapper) throws BlockException {
    if (resourceWrapper == null) {
        return;
    }
    // Ensure the checking switch is on.
    if (!checkSystemStatus.get()) {
        return;
    }

    // 只针对入口资源做校验，其它直接返回
    if (resourceWrapper.getEntryType() != EntryType.IN) {
        return;
    }

    // 全局 QPS校验
    double currentQps = Constants.ENTRY_NODE == null ? 0.0 :
    Constants.ENTRY_NODE.successQps();
    if (currentQps > qps) {
        throw new SystemBlockException(resourceWrapper.getName(), "qps");
    }
}
```

```

// 全局 线程数 校验
int currentThread = Constants.ENTRY_NODE == null ? 0 :
Constants.ENTRY_NODE.curThreadNum();
if (currentThread > maxThread) {
    throw new SystemBlockException(resourceWrapper.getName(), "thread");
}
// 全局平均 RT校验
double rt = Constants.ENTRY_NODE == null ? 0 : Constants.ENTRY_NODE.avgRt();
if (rt > maxRt) {
    throw new SystemBlockException(resourceWrapper.getName(), "rt");
}

// 全局 系统负载 校验
if (highestSystemLoadIsSet && getCurrentSystemAvgLoad() > highestSystemLoad) {
    if (!checkBbr(currentThread)) {
        throw new SystemBlockException(resourceWrapper.getName(), "load");
    }
}

// 全局 CPU使用率 校验
if (highestCpuUsageIsSet && getCurrentCpuUsage() > highestCpuUsage) {
    throw new SystemBlockException(resourceWrapper.getName(), "cpu");
}
}

```

## 2.8.ParamFlowSlot

ParamFlowSlot就是热点参数限流，如图：

新增热点规则

资源名

/order/{orderId}

限流模式

QPS 模式

参数索引

0

单机阈值

2

统计窗口时长

1

秒

是否集群

☐

参数例外项

参数类型

long

参数值

例外项参数值

限流阈值

限流阈值

+ 添加

参数值	参数类型	限流阈值	操作
<div>101</div>	long	<div>4</div>	<div><div></div>删除</div>
<div>102</div>	long	<div>6</div>	<div><div></div>删除</div>

是针对进入资源的请求，针对不同的请求参数值分别统计QPS的限流方式。

- 这里的单机阈值，就是最大令牌数量：maxCount
- 这里的统计窗口时长，就是统计时长：duration

含义是每隔duration时间长度内，最多生产maxCount个令牌，上图配置的含义是每1秒钟生产2个令牌。

核心API：



```

@Override
public void entry(Context context, ResourceWrapper resourceWrapper, DefaultNode node,
                  int count, boolean prioritized, Object... args) throws Throwable {
    // 如果没有设置热点规则，直接放行
    if (!ParamFlowRuleManager.hasRules(resourceWrapper.getName())) {
        fireEntry(context, resourceWrapper, node, count, prioritized, args);
        return;
    }
    // 热点规则判断
    checkFlow(resourceWrapper, count, args);
    // 进入下一个 slot
    fireEntry(context, resourceWrapper, node, count, prioritized, args);
}

```

### 2.8.1.令牌桶

热点规则判断采用了令牌桶算法来实现参数限流，为每一个不同参数值设置令牌桶，Sentinel的令牌桶有两部分组成：

```

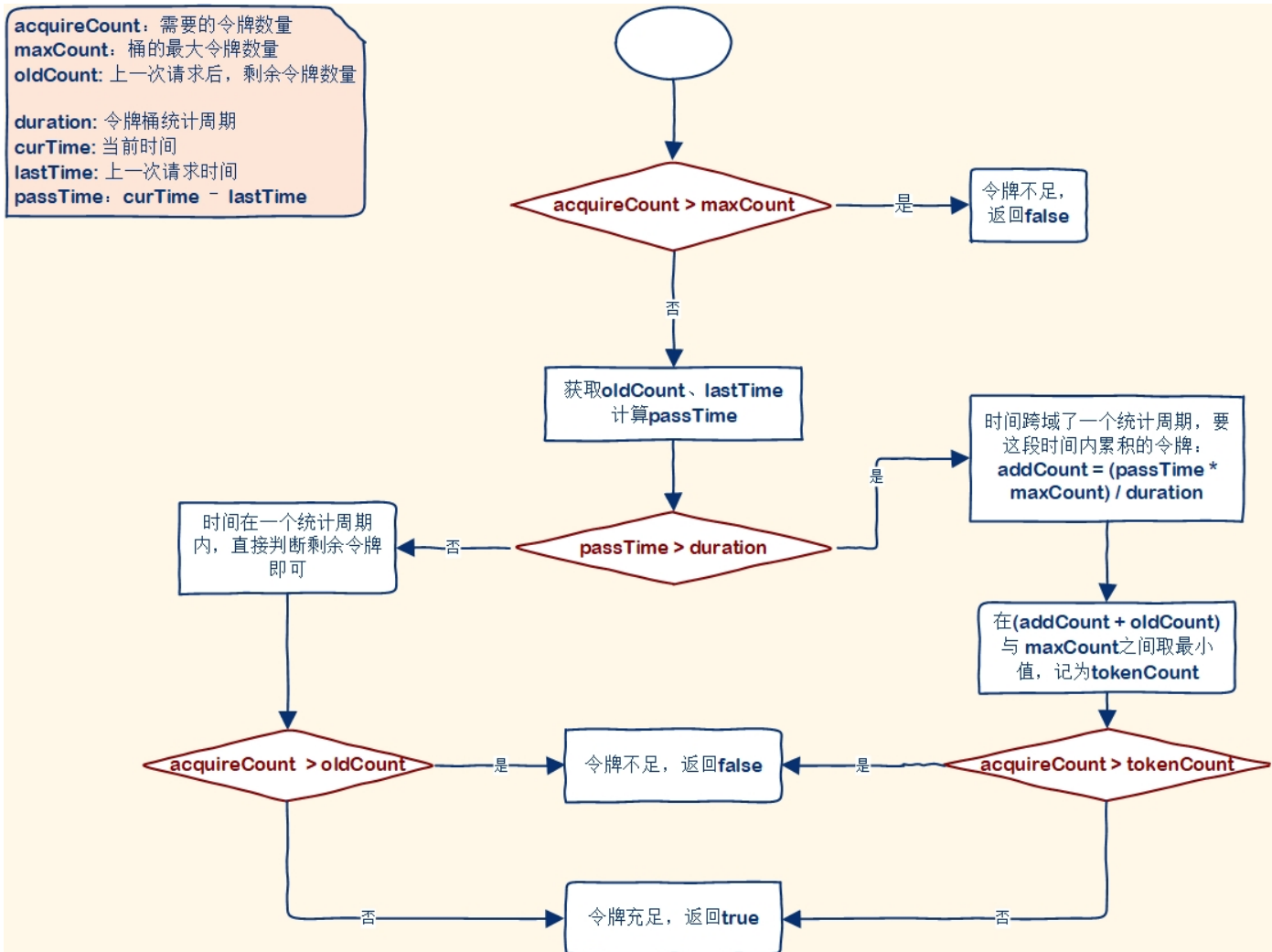
static boolean passDefaultLocalCheck(ResourceWrapper resourceWrapper, ParamFlowRule rule, int acquireCount,
                                     Object value) {
    ParameterMetric metric = getParameterMetric(resourceWrapper);
    CacheMap<Object, AtomicLong> tokenCounters = metric == null ? null : metric.getRuleTokenCounter(rule);
    CacheMap<Object, AtomicLong> timeCounters = metric == null ? null : metric.getRuleTimeCounter(rule);
}

```

这两个Map的key都是请求的参数值，value却不同，其中：

- tokenCounters：用来记录剩余令牌数量
- timeCounters：用来记录上一个请求的时间

当一个携带参数的请求到来后，基本判断流程是这样的：



## 2.9.FlowSlot

FlowSlot是负责限流规则的判断, 如图:

新增流控规则

资源名

/order/{orderId}

针对来源

default

阈值类型

☒ QPS ☐ 线程数

单机阈值

5

是否集群

☐

流控模式

☒ 直接 ☐ 关联 ☐ 链路

流控效果

☒ 快速失败 ☐ Warm Up ☐ 排队等待

关闭高级选项

新增并继续添加

新增

取消

包括：

- 三种流控模式：直接模式、关联模式、链路模式
- 三种流控效果：快速失败、warm up、排队等待

三种流控模式，从底层数据统计角度，分为两类：

- 对进入资源的所有请求（ClusterNode）做限流统计：直接模式、关联模式
- 对进入资源的部分链路（DefaultNode）做限流统计：链路模式

三种流控效果，从限流算法来看，分为两类：

- 滑动时间窗口算法：快速失败、warm up
- 漏桶算法：排队等待效果

### 2.9.1.核心流程

核心API如下：

```

@Override
public void entry(Context context, ResourceWrapper resourceWrapper, DefaultNode node, int
count,
                boolean prioritized, Object... args) throws Throwable {
    // 限流规则检测
    checkFlow(resourceWrapper, context, node, count, prioritized);
    // 放行
    fireEntry(context, resourceWrapper, node, count, prioritized, args);
}

```

checkFlow方法:

```

void checkFlow(ResourceWrapper resource, Context context, DefaultNode node, int count,
boolean prioritized)
    throws BlockException {
    // checker是 FlowRuleChecker 类的一个对象
    checker.checkFlow(ruleProvider, resource, context, node, count, prioritized);
}

```

跟入FlowRuleChecker:

```

public void checkFlow(Function<String, Collection<FlowRule>> ruleProvider,
                ResourceWrapper resource, Context context, DefaultNode node,
                int count, boolean prioritized) throws BlockException {
    if (ruleProvider == null || resource == null) {
        return;
    }
    // 获取当前资源的所有限流规则
    Collection<FlowRule> rules = ruleProvider.apply(resource.getName());
    if (rules != null) {
        for (FlowRule rule : rules) {
            // 遍历, 逐个规则做校验
            if (!canPassCheck(rule, context, node, count, prioritized)) {
                throw new FlowException(rule.getLimitApp(), rule);
            }
        }
    }
}

```

这里的FlowRule就是限流规则接口, 其中的几个成员变量, 刚好对应表单参数:

```

public class FlowRule extends AbstractRule {
    /**
     * 阈值类型 (0: 线程, 1: QPS).
     */
    private int grade = RuleConstant.FLOW_GRADE_QPS;
    /**
     * 阈值.

```

```

    */
    private double count;
    /**
     * 三种限流模式.
     *
     * {@link RuleConstant#STRATEGY_DIRECT} 直连模式;
     * {@link RuleConstant#STRATEGY_RELATE} 关联模式;
     * {@link RuleConstant#STRATEGY_CHAIN} 链路模式.
     */
    private int strategy = RuleConstant.STRATEGY_DIRECT;
    /**
     * 关联模式关联的资源名称.
     */
    private String refResource;
    /**
     * 3种流控效果.
     * 0. 快速失败, 1. warm up, 2. 排队等待, 3. warm up + 排队等待
     */
    private int controlBehavior = RuleConstant.CONTROL_BEHAVIOR_DEFAULT;
    // 预热时长
    private int warmUpPeriodSec = 10;
    /**
     * 队列最大等待时间.
     */
    private int maxQueueingTimeMs = 500;
    // ... 略
}

```

校验的逻辑定义在 `FlowRuleChecker` 的 `canPassCheck` 方法中:

```

public boolean canPassCheck(/*@NonNull*/ FlowRule rule, Context context, DefaultNode node,
int acquireCount,
                                boolean prioritized) {
    // 获取限流资源名称
    String limitApp = rule.getLimitApp();
    if (limitApp == null) {
        return true;
    }
    // 校验规则
    return passLocalCheck(rule, context, node, acquireCount, prioritized);
}

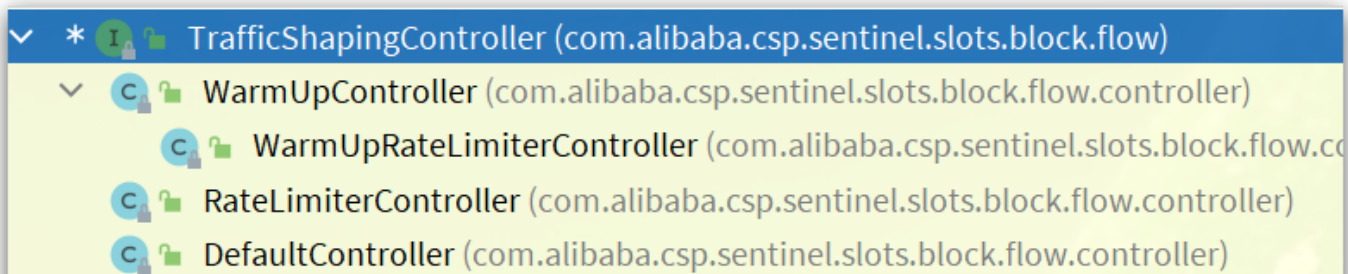
```

进入 `passLocalCheck()`:

```
private static boolean passLocalCheck(FlowRule rule, Context context, DefaultNode node,
                                     int acquireCount, boolean prioritized) {
    // 基于限流模式判断要统计的节点,
    // 如果是直连模式, 关联模式, 对ClusterNode统计, 如果是链路模式, 则对DefaultNode统计
    Node selectedNode = selectNodeByRequesterAndStrategy(rule, context, node);
    if (selectedNode == null) {
        return true;
    }
    // 判断规则
    return rule.getRater().canPass(selectedNode, acquireCount, prioritized);
}
```

这里对规则的判断先要通过 `FlowRule#getRater()` 获取流量控制器 `TrafficShapingController`, 然后再做限流。

而 `TrafficShapingController` 有3种实现:



- DefaultController: 快速失败, 默认的方式, 基于滑动时间窗口算法
- WarmUpController: 预热模式, 基于滑动时间窗口算法, 只不过阈值是动态的
- RateLimiterController: 排队等待模式, 基于漏桶算法

最终的限流判断都在 `TrafficShapingController` 的 `canPass` 方法中。

## 2.9.2.滑动时间窗口

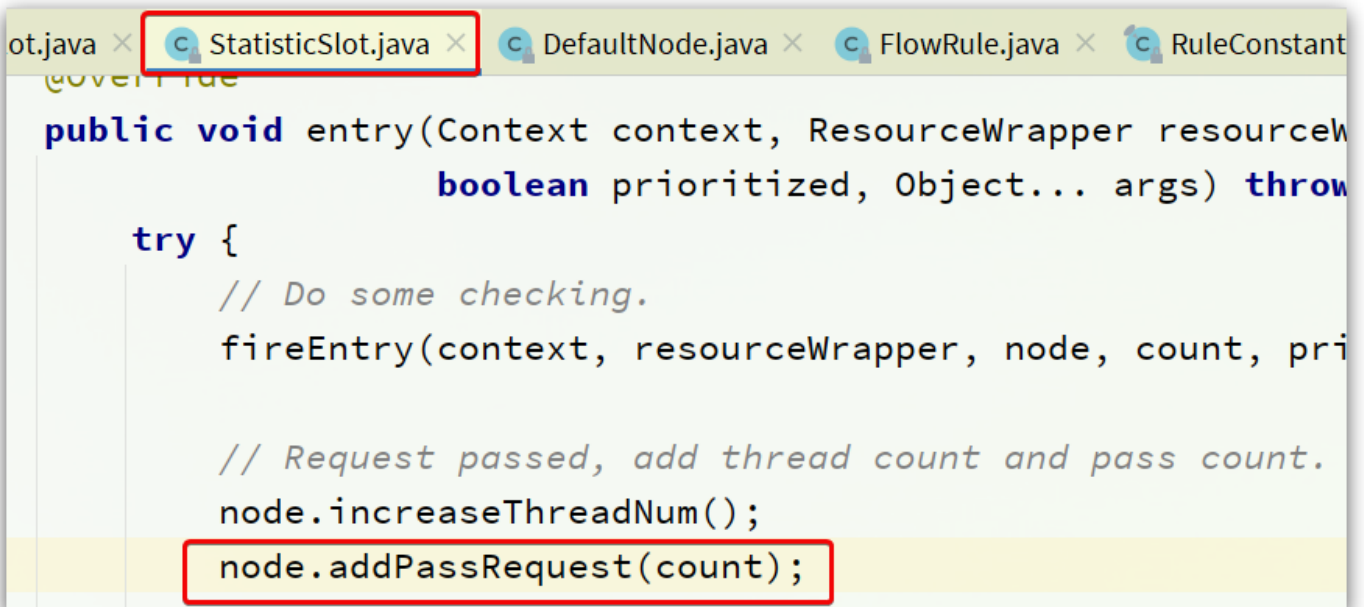
滑动时间窗口的功能分两部分来看:

- 一是时间区间窗口的QPS计数功能, 这个是在 `StatisticSlot` 中调用的
- 二是对滑动窗口内的时间区间窗口QPS累加, 这个是在 `FlowRule` 中调用的

先来看时间区间窗口的QPS计数功能。

### 2.9.2.1.时间窗口请求量统计

回顾2.5章节中的 `StatisticSlot` 部分, 有这样一段代码:

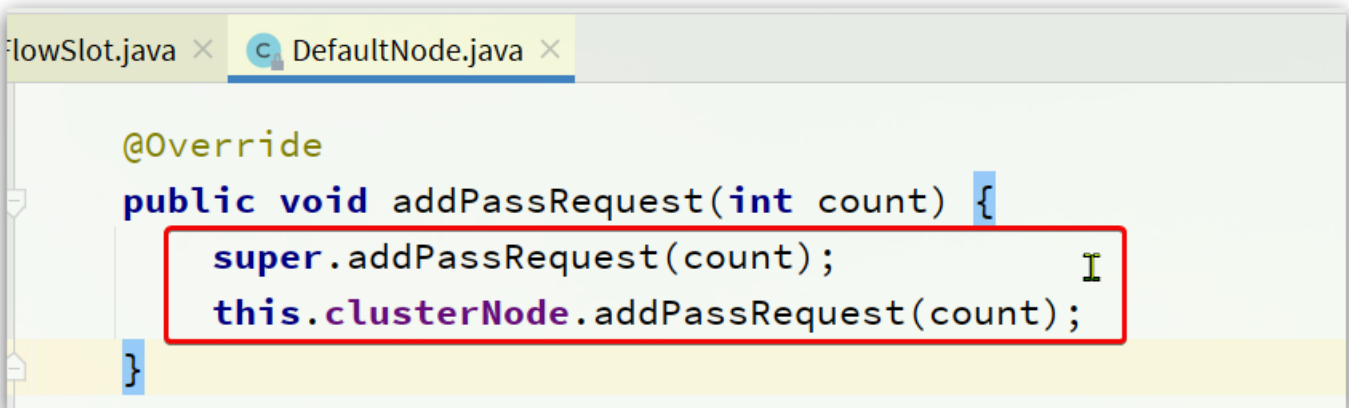


```
ot.java x StatisticSlot.java x DefaultNode.java x FlowRule.java x RuleConstant
@Override
public void entry(Context context, ResourceWrapper resourceWrapper,
    boolean prioritized, Object... args) throws
    try {
        // Do some checking.
        fireEntry(context, resourceWrapper, node, count, pri

        // Request passed, add thread count and pass count.
        node.increaseThreadNum();
        node.addPassRequest(count);
```

就是在统计通过该节点的QPS，我们跟入看看，这里进入了DefaultNode内部：

[



```
FlowSlot.java x DefaultNode.java x
@Override
public void addPassRequest(int count) {
    super.addPassRequest(count);
    this.clusterNode.addPassRequest(count);
}
```

发现同时对 DefaultNode 和 ClusterNode 在做QPS统计，我们知道 DefaultNode 和 ClusterNode 都是 StatisticNode 的子类，这里调用 addPassRequest() 方法，最终都会进入 StatisticNode 中。

随便跟入一个：



```

java x DefaultNode.java x StatisticNode.java x

@Override
public int curThreadNum() { return (int)curThreadNum.sum(); }

@Override
public void addPassRequest(int count) {
    rollingCounterInSecond.addPass(count);
    rollingCounterInMinute.addPass(count);
}

```

这里有秒、分两种纬度的统计，对应两个计数器。找到对应的成员变量，可以看到：

```

Author: qinan.qn, jialiang.linjl
public class StatisticNode implements Node {

    Holds statistics of the recent INTERVAL seconds. The
    INTERVAL is divided into time spans by given sampleCount.

    private transient volatile Metric rollingCounterInSecond = new ArrayMetric(SampleCountProperty.SAMPLE_COUNT,
        IntervalProperty.INTERVAL);

    Holds statistics of the recent 60 seconds. The
    windowLengthInMs is deliberately set to 1000 milliseconds,
    meaning each bucket per second, in this way we can get
    accurate statistics of each second.

    private transient Metric rollingCounterInMinute = new ArrayMetric(sampleCount: 60, intervalInMs: 60 * 1000, enableOccup

```

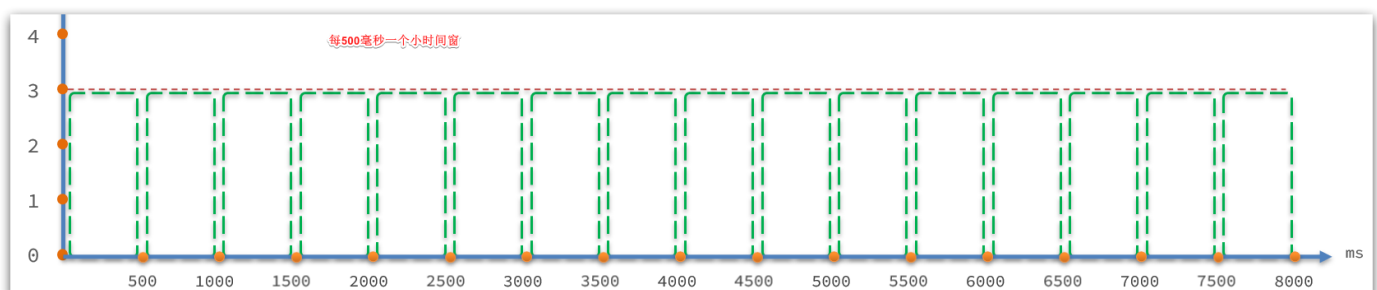
两个计数器都是ArrayMetric类型，并且传入了两个参数：

```

// intervalInMs: 是滑动窗口的时间间隔，默认为 1 秒
// sampleCount: 时间窗口的分隔数量，默认为 2，就是把 1秒分为 2个小时时间窗
public ArrayMetric(int sampleCount, int intervalInMs) {
    this.data = new OccupiableBucketLeapArray(sampleCount, intervalInMs);
}

```

如图：



接下来，我们进入 ArrayMetric 类的 addPass 方法：

```

@Override
public void addPass(int count) {
    // 获取当前时间所在的时间窗
    WindowWrap<MetricBucket> wrap = data.currentWindow();
    // 计数器 +1
    wrap.value().addPass(count);
}

```

那么，计数器如何知道当前所在的窗口是哪一个呢？

这里的数据是一个LeapArray：

```

public class ArrayMetric implements Metric {

    private final LeapArray<MetricBucket> data;
}

```

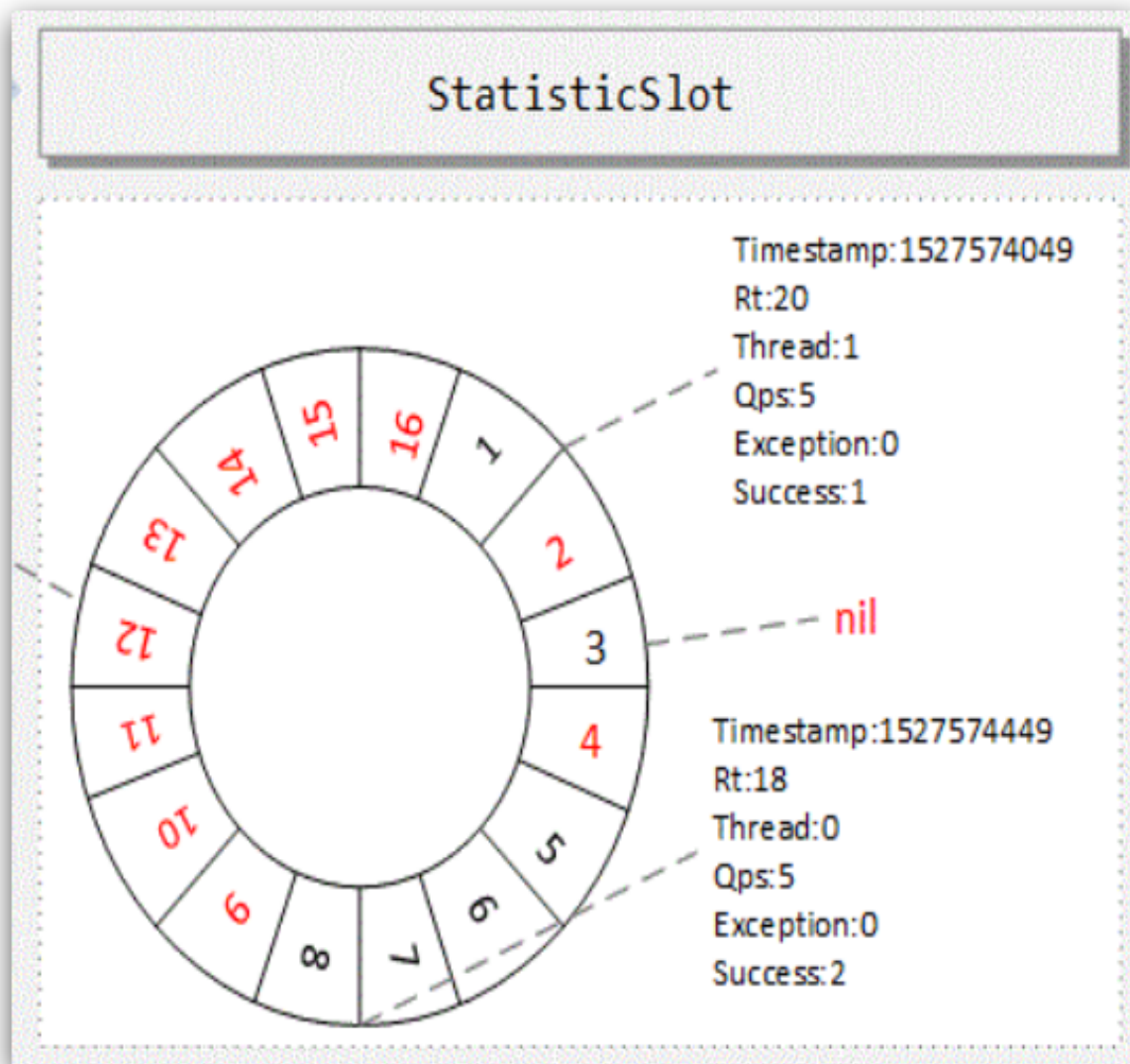
LeapArray的四个属性：

```

public abstract class LeapArray<T> {
    // 小窗口的时间长度，默认是500ms，值 = intervalInMs / sampleCount
    protected int windowLengthInMs;
    // 滑动窗口内的小窗口数量，默认为 2
    protected int sampleCount;
    // 滑动窗口的时间间隔，默认为 1000ms
    protected int intervalInMs;
    // 滑动窗口的时间间隔，单位为秒，默认为 1
    private double intervalInSeconds;
}

```

LeapArray是一个环形数组，因为时间是无限的，数组长度不可能无限，因此数组中每一个格子放入一个时间窗（window），当数组放满后，角标归0，覆盖最初的window。



因为滑动窗口最多分成sampleCount数量的小窗口，因此数组长度只要大于sampleCount，那么最近的一个滑动窗口内的2个小窗口就永远不会被覆盖，就不用担心旧数据被覆盖的问题了。

我们跟入 `data.currentWindow()` 方法：

```
public WindowWrap<T> currentWindow(long timeMillis) {
    if (timeMillis < 0) {
        return null;
    }
    // 计算当前时间对应的数组角标
    int idx = calculateTimeIdx(timeMillis);
    // 计算当前时间所在窗口的开始时间。
    long windowStart = calculateWindowStart(timeMillis);

    /*
     * 先根据角标获取数组中保存的 oldWindow 对象，可能是旧数据，需要判断。
     *
     * (1) oldWindow 不存在，说明是第一次，创建新 window并存入，然后返回即可
     * (2) oldWindow的 starTime = 本次请求的 windowStar，说明正是要找的窗口，直接返回。
     * (3) oldWindow的 starTime < 本次请求的 windowStar，说明是旧数据，需要被覆盖，创建
     *     新窗口，覆盖旧窗口
     */
}
```

```

        */
        while (true) {
            WindowWrap<T> old = array.get(idx);
            if (old == null) {
                // 创建新 window
                WindowWrap<T> window = new WindowWrap<T>(windowLengthInMs, windowStart,
newEmptyBucket(timeMillis));
                // 基于CAS写入数组，避免线程安全问题
                if (array.compareAndSet(idx, null, window)) {
                    // 写入成功，返回新的 window
                    return window;
                } else {
                    // 写入失败，说明有并发更新，等待其它人更新完成即可
                    Thread.yield();
                }
            } else if (windowStart == old.windowStart()) {
                return old;
            } else if (windowStart > old.windowStart()) {
                if (updateLock.tryLock()) {
                    try {
                        // 获取并发锁，覆盖旧窗口并返回
                        return resetWindowTo(old, windowStart);
                    } finally {
                        updateLock.unlock();
                    }
                } else {
                    // 获取锁失败，等待其它线程处理就可以了
                    Thread.yield();
                }
            } else if (windowStart < old.windowStart()) {
                // 这种情况不应该存在，写这里只是以防万一。
                return new WindowWrap<T>(windowLengthInMs, windowStart,
newEmptyBucket(timeMillis));
            }
        }
    }
}

```

找到当前时间所在窗口（WindowWrap）后，只要调用WindowWrap对象中的add方法，计数器+1即可。

这里只负责统计每个窗口的请求量，不负责拦截。限流拦截要看FlowSlot中的逻辑。

### 2.9.2.2.滑动窗口QPS计算

在2.9.1小节我们讲过，FlowSlot的限流判断最终都由 `TrafficShapingController` 接口中的 `canPass` 方法来实现。该接口有三个实现类：

- DefaultController：快速失败，默认的方式，基于滑动时间窗口算法
- WarmUpController：预热模式，基于滑动时间窗口算法，只不过阈值是动态的
- RateLimiterController：排队等待模式，基于漏桶算法

因此，我们跟入默认的DefaultController中的canPass方法来分析：

```

@Override
public boolean canPass(Node node, int acquireCount, boolean prioritized) {
    // 计算目前为止滑动窗口内已经存在的请求量
    int curCount = avgUsedTokens(node);
    // 判断：已使用请求量 + 需要的请求量 (1) 是否大于 窗口的请求阈值
    if (curCount + acquireCount > count) {
        // 大于，说明超出阈值，返回false
        if (prioritized && grade == RuleConstant.FLOW_GRADE_QPS) {
            long currentTime;
            long waitInMs;
            currentTime = TimeUtil.currentTimeMillis();
            waitInMs = node.tryOccupyNext(currentTime, acquireCount, count);
            if (waitInMs < OccupyTimeoutProperty.getOccupyTimeout()) {
                node.addWaitingRequest(currentTime + waitInMs, acquireCount);
                node.addOccupiedPass(acquireCount);
                sleep(waitInMs);

                // PriorityWaitException indicates that the request will pass after
                // waiting for {@link @waitInMs}.
                throw new PriorityWaitException(waitInMs);
            }
        }
        return false;
    }
    // 小于等于，说明在阈值范围内，返回true
    return true;
}

```

因此，判断的关键就是 `int curCount = avgUsedTokens(node);`

```

private int avgUsedTokens(Node node) {
    if (node == null) {
        return DEFAULT_AVG_USED_TOKENS;
    }
    return grade == RuleConstant.FLOW_GRADE_THREAD ? node.curThreadNum() : (int)
(node.passQps());
}

```

因为我们采用的是限流，走 `node.passQps()` 逻辑：

```

// 这里又进入了 StatisticNode类
@Override
public double passQps() {
    // 请求量 ÷ 滑动窗口时间间隔，得到的就是QPS
    return rollingCounterInSecond.pass() /
    rollingCounterInSecond.getWindowIntervalInSec();
}

```

那么 `rollingCounterInSecond.pass()` 是如何得到请求量的呢？

```
// rollingCounterInSecond 本质是ArrayMetric, 之前说过
@Override
public long pass() {
    // 获取当前窗口
    data.currentWindow();
    long pass = 0;
    // 获取 当前时间的 滑动窗口范围内 的所有小窗口
    List<MetricBucket> list = data.values();
    // 遍历
    for (MetricBucket window : list) {
        // 累加求和
        pass += window.pass();
    }
    // 返回
    return pass;
}
```

来看看 `data.values()` 如何获取 滑动窗口范围内 的所有小窗口：

```
// 此处进入LeapArray类中:

public List<T> values(long timeMillis) {
    if (timeMillis < 0) {
        return new ArrayList<T>();
    }
    // 创建空集合, 大小等于 LeapArray长度
    int size = array.length();
    List<T> result = new ArrayList<T>(size);
    // 遍历 LeapArray
    for (int i = 0; i < size; i++) {
        // 获取每一个小窗口
        WindowWrap<T> windowWrap = array.get(i);
        // 判断这个小窗口是否在 滑动窗口时间范围内 (1秒内)
        if (windowWrap == null || isWindowDeprecated(timeMillis, windowWrap)) {
            // 不在范围内, 则跳过
            continue;
        }
        // 在范围内, 则添加到集合中
        result.add(windowWrap.value());
    }
    // 返回集合
    return result;
}
```

那么, `isWindowDeprecated(timeMillis, windowWrap)` 又是如何判断窗口是否符合要求呢?

```

public boolean isWindowDeprecated(long time, WindowWrap<T> windowWrap) {
    // 当前时间 - 窗口开始时间 是否大于 滑动窗口的最大间隔 (1秒)
    // 也就是说, 我们要统计的时 距离当前时间1秒内的 小窗口的 count之和
    return time - windowWrap.windowStart() > intervalInMs;
}

```

### 2.9.3.漏桶

上一节我们讲过, FlowSlot的限流判断最终都由 TrafficShapingController 接口中的 canPass 方法来实现。该接口有三个实现类:

- DefaultController: 快速失败, 默认的方式, 基于滑动时间窗口算法
- WarmUpController: 预热模式, 基于滑动时间窗口算法, 只不过阈值是动态的
- RateLimiterController: 排队等待模式, 基于漏桶算法

因此, 我们跟入默认的RateLimiterController中的canPass方法来分析:

```

@Override
public boolean canPass(Node node, int acquireCount, boolean prioritized) {
    // Pass when acquire count is less or equal than 0.
    if (acquireCount <= 0) {
        return true;
    }
    // 阈值小于等于 0 , 阻止请求
    if (count <= 0) {
        return false;
    }
    // 获取当前时间
    long currentTime = TimeUtil.currentTimeMillis();
    // 计算两次请求之间允许的最小时间间隔
    long costTime = Math.round(1.0 * (acquireCount) / count * 1000);

    // 计算本次请求 允许执行的时间点 = 最近一次请求的可执行时间 + 两次请求的最小间隔
    long expectedTime = costTime + latestPassedTime.get();
    // 如果允许执行的时间点小于当前时间, 说明可以立即执行
    if (expectedTime <= currentTime) {
        // 更新上一次的请求的执行时间
        latestPassedTime.set(currentTime);
        return true;
    } else {
        // 不能立即执行, 需要计算 预期等待时长
        // 预期等待时长 = 两次请求的最小间隔 + 最近一次请求的可执行时间 - 当前时间
        long waitTime = costTime + latestPassedTime.get() - TimeUtil.currentTimeMillis();
        // 如果预期等待时间超出阈值, 则拒绝请求
        if (waitTime > maxQueueingTimeMs) {
            return false;
        } else {
            // 预期等待时间小于阈值, 更新最近一次请求的可执行时间, 加上costTime
            long oldTime = latestPassedTime.addAndGet(costTime);
            try {

```

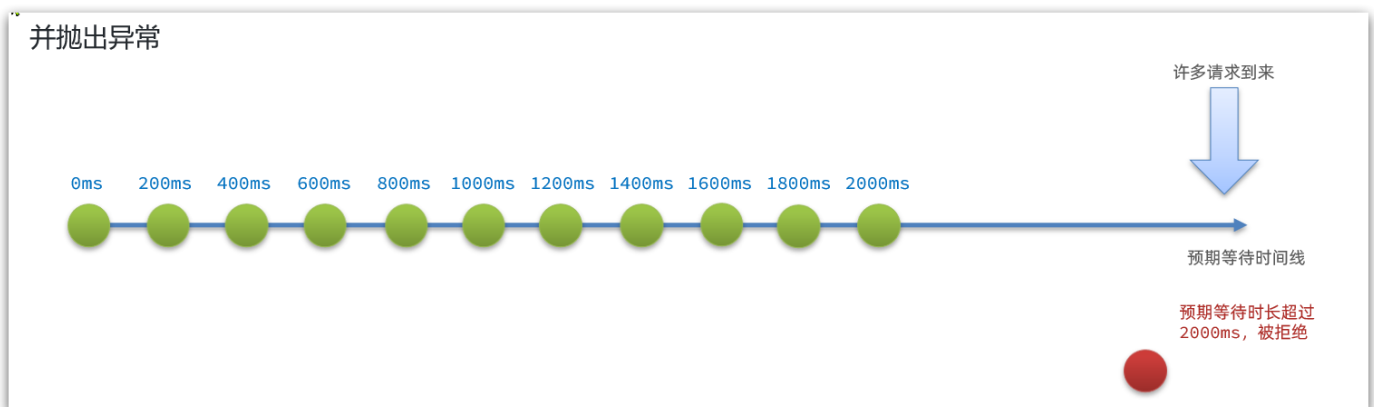


```

        // 保险起见，再判断一次预期等待时间，是否超过阈值
        waitTime = oldTime - TimeUtil.currentTimeMillis();
        if (waitTime > maxQueueingTimeMs) {
            // 如果超过，则把刚才 加 的时间再 减回来
            latestPassedTime.addAndGet(-costTime);
            // 拒绝
            return false;
        }
        // in race condition waitTime may <= 0
        if (waitTime > 0) {
            // 预期等待时间在阈值范围内，休眠要等待的时间，醒来后继续执行
            Thread.sleep(waitTime);
        }
        return true;
    } catch (InterruptedException e) {
    }
}
return false;
}

```

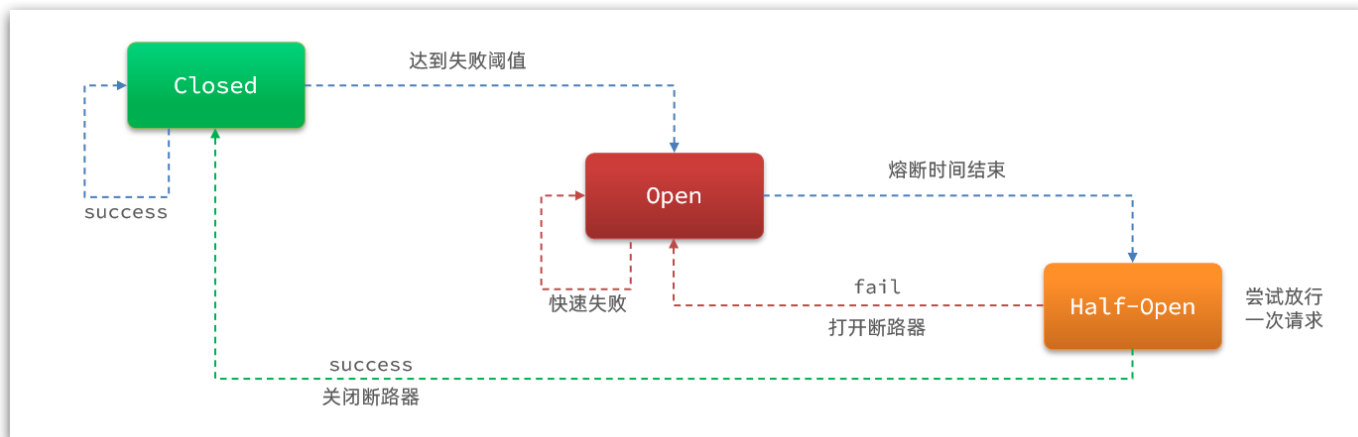
与我们之前分析的漏桶算法基本一致：



## 2.10.DegradeSlot

最后一关，就是降级规则判断了。

Sentinel的降级是基于状态机来实现的：



对应的实现在DegradeSlot类中，核心API:

```
@Override
public void entry(Context context, ResourceWrapper resourceWrapper, DefaultNode node,
    int count, boolean prioritized, Object... args) throws Throwable {
    // 熔断降级规则判断
    performChecking(context, resourceWrapper);
    // 继续下一个slot
    fireEntry(context, resourceWrapper, node, count, prioritized, args);
}
```

继续进入performChecking方法:

```
void performChecking(Context context, ResourceWrapper r) throws BlockException {
    // 获取当前资源上的所有的断路器 CircuitBreaker
    List<CircuitBreaker> circuitBreakers =
    DegradeRuleManager.getCircuitBreakers(r.getName());
    if (circuitBreakers == null || circuitBreakers.isEmpty()) {
        return;
    }
    for (CircuitBreaker cb : circuitBreakers) {
        // 遍历断路器，逐个判断
        if (!cb.tryPass(context)) {
            throw new DegradeException(cb.getRule().getLimitApp(), cb.getRule());
        }
    }
}
```

## 2.10.1.CircuitBreaker

我们进入CircuitBreaker的tryPass方法中:

```
@Override
public boolean tryPass(Context context) {
    // 判断状态机状态
    if (currentState.get() == State.CLOSED) {
        // 如果是closed状态，直接放行
    }
}
```

```

        return true;
    }
    if (currentState.get() == State.OPEN) {
        // 如果是OPEN状态，断路器打开
        // 继续判断OPEN时间窗是否结束，如果是则把状态从OPEN切换到 HALF_OPEN，返回true
        return retryTimeoutArrived() && fromOpenToHalfOpen(context);
    }
    // OPEN状态，并且时间窗未到，返回false
    return false;
}

```

有关时间窗的判断在 `retryTimeoutArrived()` 方法：

```

protected boolean retryTimeoutArrived() {
    // 当前时间 大于 下一次 HalfOpen的重试时间
    return TimeUtil.currentTimeMillis() >= nextRetryTimestamp;
}

```

OPEN到HALF\_OPEN切换在 `fromOpenToHalfOpen(context)` 方法：

```

protected boolean fromOpenToHalfOpen(Context context) {
    // 基于CAS修改状态，从 OPEN到 HALF_OPEN
    if (currentState.compareAndSet(State.OPEN, State.HALF_OPEN)) {
        // 状态变更的事件通知
        notifyObservers(State.OPEN, State.HALF_OPEN, null);
        // 得到当前资源
        Entry entry = context.getCurEntry();
        // 给资源设置监听器，在资源Entry销毁时（资源业务执行完毕时）触发
        entry.whenTerminate(new BiConsumer<Context, Entry>() {
            @Override
            public void accept(Context context, Entry entry) {
                // 判断 资源业务是否异常
                if (entry.getBlockError() != null) {
                    // 如果异常，则再次进入OPEN状态
                    currentState.compareAndSet(State.HALF_OPEN, State.OPEN);
                    notifyObservers(State.HALF_OPEN, State.OPEN, 1.0d);
                }
            }
        });
        return true;
    }
    return false;
}

```

这里出现了从OPEN到HALF\_OPEN、从HALF\_OPEN到OPEN的变化，但是还有几个没有：

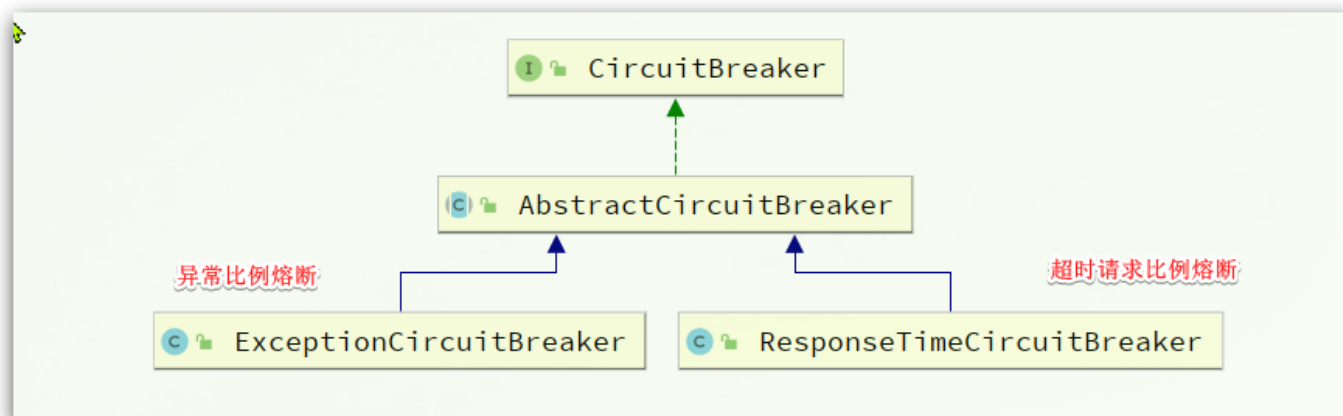
- 从CLOSED到OPEN
- 从HALF\_OPEN到CLOSED

## 2.10.2.触发断路器

请求经过所有插槽 后，一定会执行exit方法，而在DegradeSlot的exit方法中：

```
DegradeSlot.java x AbstractCircuitBreaker.java x ExceptionCircuitBreaker.java x Entry.java x CircuitBreaker.java
69 fireExit(context, r, count, args);
70 return;
71 }
72
73 if (curEntry.getBlockError() == null) {
74     // passed request
75     for (CircuitBreaker circuitBreaker : circuitBreakers) {
76         circuitBreaker.onRequestComplete(context);
77     }
78 }
79
```

会调用CircuitBreaker的onRequestComplete方法。而CircuitBreaker有两个实现：



我们这里以异常比例熔断为例来看，进入ExceptionCircuitBreaker的onRequestComplete方法：

```
@Override
public void onRequestComplete(Context context) {
    // 获取资源 Entry
    Entry entry = context.getCurEntry();
    if (entry == null) {
        return;
    }
    // 尝试获取 资源中的 异常
    Throwable error = entry.getError();
    // 获取计数器，同样采用了滑动窗口来计数
    SimpleErrorCounter counter = stat.currentWindow().value();
    if (error != null) {
        // 如果出现异常，则 error计数器 +1
        counter.getErrorCount().add(1);
    }
    // 不管是否出现异常，total计数器 +1
    counter.getTotalCount().add(1);
    // 判断异常比例是否超出阈值
}
```

```
    handleStateChangeWhenThresholdExceeded(error);  
}
```

来看阈值判断的方法：

```
private void handleStateChangeWhenThresholdExceeded(Throwable error) {  
    // 如果当前已经是OPEN状态，不做处理  
    if (currentState.get() == State.OPEN) {  
        return;  
    }  
    // 如果已经是 HALF_OPEN 状态，判断是否需求切换状态  
    if (currentState.get() == State.HALF_OPEN) {  
        if (error == null) {  
            // 没有异常，则从 HALF_OPEN 到 CLOSED  
            fromHalfOpenToClose();  
        } else {  
            // 有一次，再次进入OPEN  
            fromHalfOpenToOpen(1.0d);  
        }  
        return;  
    }  
    // 说明当前是CLOSE状态，需要判断是否触发阈值  
    List<SimpleErrorCounter> counters = stat.values();  
    long errCount = 0;  
    long totalCount = 0;  
    // 累加计算 异常请求数量、总请求数量  
    for (SimpleErrorCounter counter : counters) {  
        errCount += counter.errorCount.sum();  
        totalCount += counter.totalCount.sum();  
    }  
    // 如果总请求数量未达到阈值，什么都不做  
    if (totalCount < minRequestAmount) {  
        return;  
    }  
    double curCount = errCount;  
    if (strategy == DEGRADE_GRADE_EXCEPTION_RATIO) {  
        // 计算请求的异常比例  
        curCount = errCount * 1.0d / totalCount;  
    }  
    // 如果比例超过阈值，切换到 OPEN  
    if (curCount > threshold) {  
        transformToOpen(curCount);  
    }  
}
```