sentinel 核心概念.pdf

sentinel Rule机制.pdf

滑动窗口统计机制.pdf

sentinel 执行原理.pdf

sentinel 集群流控原理.pdf

sentinel dubbo适配机制.pdf

资源和规则
核心概念
Resource
Context
Entry
DefaultNode
StatisticNode
Slot

资源和规则

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。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。
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。

sentinel中调用SphU或者SphO的entry方法获取限流资源，不同的是前者获取限流资源失败时会抛BlockException异常，后者或捕获该异常并返回false，二者的实现都是基于CtSph类完成的。简单的sentinel示例：

```
Entry entry = null;
try {
 entry = SphU.entry(KEY);
 System.out.println("entry ok...");
} catch (BlockException e1) {
 // 获取限流资源失败
} catch (Exception e2) {
 // biz exception
```

```
} finally {
  if (entry != null) {
  entry.exit();
  }
}

Entry entry = null;
if (SphO.entry(KEY)) {
 System.out.println("entry ok");
} else {
 // 获取限流资源失败
}
```

SphU和SphO二者没有孰优孰略问题，底层实现是一样的，根据不同场景选举合适的一个即可。看了简单示例之后，一起来看下sentinel中的核心概念，便于理解后续内容。

核心概念

Resource

resource是sentinel中最重要的一个概念，sentinel通过资源来保护具体的业务代码或其他后方服务。sentinel把复杂的逻辑给屏蔽掉了，用户只需要为受保护的代码或服务定义一个资源，然后定义规则就可以了，剩下的通通交给sentinel来处理了。并且资源和规则是解耦的，规则甚至可以在运行时动态修改。定义完资源后，就可以通过在程序中埋点来保护你自己的服务了，埋点的方式有两种：

- try-catch 方式（通过 SphU.entry(...)），当 catch 到BlockException时执行异常处理(或fallback)
- if-else 方式（通过 SphO.entry(...)），当返回 false 时执行异常处理(或fallback)

以上这两种方式都是通过硬编码的形式定义资源然后进行资源埋点的，对业务代码的侵入太大，从0.1.1版本开始，sentinel加入了注解的支持，可以通过注解来定义资源，具体的注解为：SentinelResource 。通过注解除了可以定义资源外，还可以指定 blockHandler 和 fallback 方法。

在sentinel中具体表示资源的类是：ResourceWrapper ，他是一个抽象的包装类，包装了资源的 Name 和EntryType。他有两个实现类，分别是：StringResourceWrapper 和

MethodResourceWrapper。顾名思义，StringResourceWrapper 是通过对一串字符串进行包装，是一个通用的资源包装类，MethodResourceWrapper 是对方法调用的包装。

Context

Context是对资源操作时的上下文环境，每个资源操作(针对Resource进行的entry/exit)必须属于一个Context，如果程序中未指定Context，会创建name为"sentinel_default_context"的默认Context。一个Context生命周期内可能有多个资源操作，Context生命周期内的最后一个资源exit时会清理该Context，这也预示这整个Context生命周期的结束。Context主要属性如下：

```
public class Context {
 // context名字，默认名字 "sentinel_default_context"
 private final String name;
 // context入口节点，每个context必须有一个entranceNode
 private DefaultNode entranceNode;
 // context当前entry，Context生命周期中可能有多个Entry，所有curEntry会有变化
 private Entry curEntry;
 // The origin of this context (usually indicate different invokers, e.g. service consumer name or origin IP).
 private String origin = "";
  private final boolean async;
}
```

注意：一个Context生命期内Context只能初始化一次，因为是存到ThreadLocal中，并且只有在非null时才会进行初始化。

如果想在调用 SphU.entry() 或 SphO.entry() 前，自定义一个context，则通过ContextUtil.enter()方法来创建。context是保存在ThreadLocal中的，每次执行的时候会优先到ThreadLocal中获取，为null时会调用 MyContextUtil.myEnter(Constants.CONTEXT_DEFAULT_NAME, "", resourceWrapper.getType()) 创建一个context。当Entry执行exit方法时，如果entry的parent节点为null，表示是当前Context中最外层的Entry了，此时将ThreadLocal中的context清空。

Entry

刚才在Context身影中也看到了Entry的出现，现在就谈谈Entry。每次执行 SphU.entry() 或 SphO.entry() 都会返回一个Entry，Entry表示一次资源操作，内部会保存当前invocation信息。在一个Context生命周期中多次资源操作，也就是对应多个Entry，这些Entry形成parent/child结构保存在Entry实例中，entry类CtEntry结构如下：

```
class CtEntry extends Entry {
 protected Entry parent = null;
 protected Entry child = null;

 protected ProcessorSlot<Object> chain;
 protected Context context;
}
public abstract class Entry implements AutoCloseable {
 private long createTime;
  private Node curNode;
  /**
  * {@link Node} of the specific origin, Usually the origin is the Servic
e Consumer.
  */
  private Node originNode;
  private Throwable error; // 是否出现异常
  protected ResourceWrapper resourceWrapper; // 资源信息
}
```

Entry实例代码中出现了Node，这个又是什么东东呢 :(，咱们接着往下看：

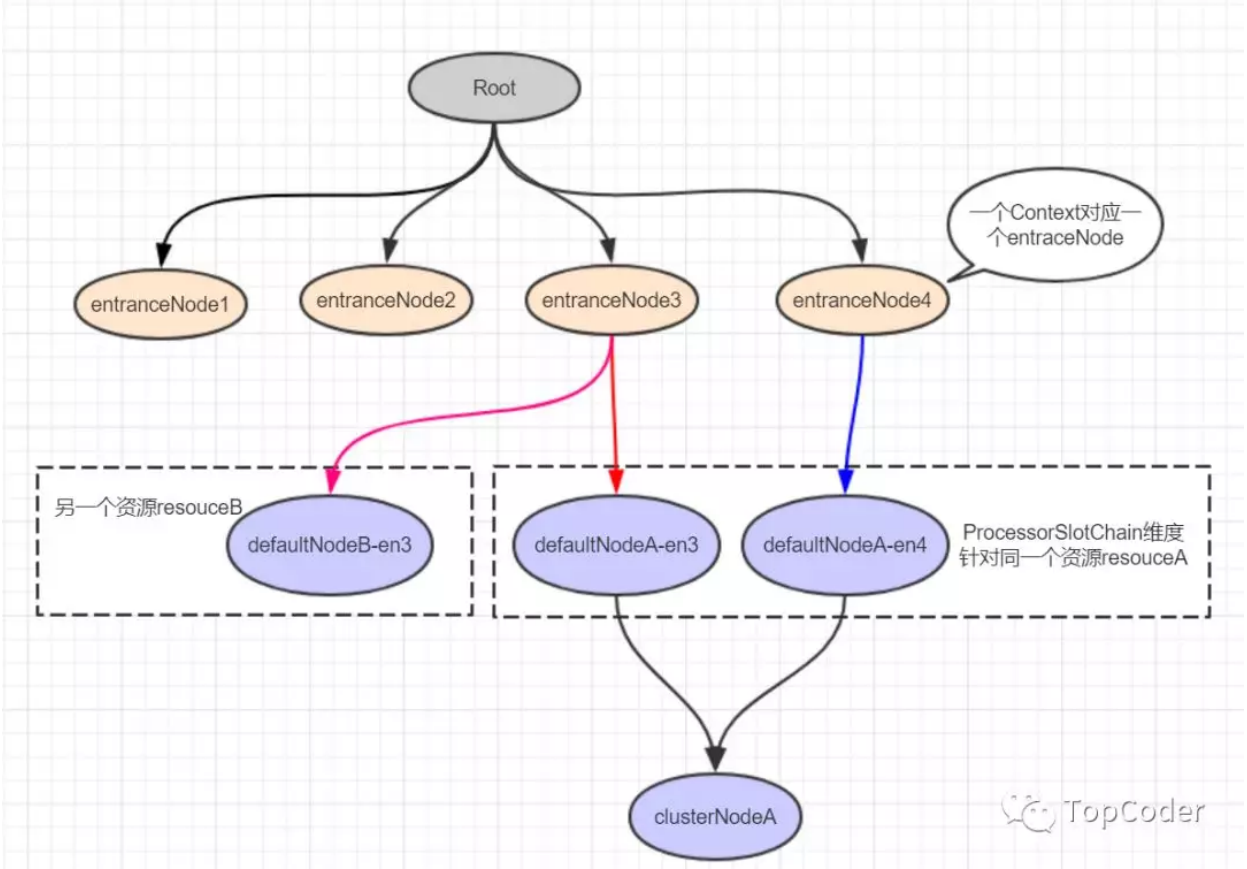
DefaultNode

Node（*关于StatisticNode的讨论放到下一小节*）默认实现类DefaultNode，该类还有一个子类EntranceNode；context有一个entranceNode属性，Entry中有一个curNode属性。

- **EntranceNode**：该类的创建是在初始化Context时完成的（ContextUtil.trueEnter方法），注意该类是针对Context维度的，也就是一个context有且仅有一个EntranceNode。
- **DefaultNode**：该类的创建是在NodeSelectorSlot.entry完成的，当不存在context.name对应的DefaultNode时会新建（new DefaultNode(resourceWrapper, null)，对应resouce）并保存到本地缓存（NodeSelectorSlot中private volatile Map<String, DefaultNode> map）；获

> 取到context.name对应的DefaultNode后会将该DefaultNode设置到当前context的curEntry.curNode属性，也就是说，在NodeSelectorSlot中是一个context有且仅有一个DefaultNode。

看到这里，你是不是有疑问？为什么一个context有且仅有一个DefaultNode，我们的resouece跑哪去了呢，其实，这里的一个context有且仅有一个DefaultNode是在NodeSelectorSlot范围内，NodeSelectorSlot是ProcessorSlotChain中的一环，获取ProcessorSlotChain是根据Resource维度来的。总结为一句话就是：**针对同一个Resource，多个context对应多个DefaultNode；针对不同Resource，(不管是否是同一个context)对应多个不同DefaultNode。**这还没看明白：(，好吧，我不bb了，上图吧：



DefaultNode结构如下：

```
1 public class DefaultNode extends StatisticNode {
2  private ResourceWrapper id;
3  /**
4  * The list of all child nodes.
5  * 子节点集合
6  */
7  private volatile Set<Node> childList = new HashSet<>();
8  /**
9  * Associated cluster node.
```

```
 */
 private ClusterNode clusterNode;
}
```

一个Resouce只有一个clusterNode，多个defaultNode对应一个clusterNode，如果defaultNode.clusterNode为null，则在ClusterBuilderSlot.entry中会进行初始化。

同一个Resource，对应同一个ProcessorSlotChain，这块处理逻辑在lookProcessChain方法中，如下：

```
ProcessorSlot<Object> lookProcessChain(ResourceWrapper resourceWrapper) {
 ProcessorSlotChain chain = chainMap.get(resourceWrapper);
 if (chain == null) {
 synchronized (LOCK) {
 chain = chainMap.get(resourceWrapper);
 if (chain == null) {
 // Entry size limit.
 if (chainMap.size() >= Constants.MAX_SLOT_CHAIN_SIZE) {
 return null;
  }

  chain = SlotChainProvider.newSlotChain();
  Map<ResourceWrapper, ProcessorSlotChain> newMap = new HashMap<ResourceW
rapper, ProcessorSlotChain>(
  chainMap.size() + 1);
  newMap.putAll(chainMap);
  newMap.put(resourceWrapper, chain);
  chainMap = newMap;
  }
  }
  }
  return chain;
}
```

StatisticNode

StatisticNode中保存了资源的实时统计数据（基于滑动时间窗口机制），通过这些统计数据，sentinel才能进行限流、降级等一系列操作。StatisticNode属性如下：

```
public class StatisticNode implements Node {
```

```
    /**
     * 秒级的滑动时间窗口（时间窗口单位500ms）
     */
    private transient volatile Metric rollingCounterInSecond = new ArrayMetr
ic(SampleCountProperty.SAMPLE_COUNT,
    IntervalProperty.INTERVAL);
    /**
     * 分钟级的滑动时间窗口（时间窗口单位1s）
     */
     private transient Metric rollingCounterInMinute = new ArrayMetric(60, 6
0 * 1000, false);
     /**
     * The counter for thread count.
     * 线程个数用户触发线程数流控
     */
     private LongAdder curThreadNum = new LongAdder();
}
public class ArrayMetric implements Metric {
     private final LeapArray<MetricBucket> data;
}
public class MetricBucket {
     // 保存统计值
     private final LongAdder[] counters;
     // 最小rt
     private volatile long minRt;
}
```

其中MetricBucket.counters数组大小为MetricEvent枚举值的个数，每个枚举对应一个统计项，比如PASS表示通过个数，限流可根据通过的个数和设置的限流规则配置count大小比较，得出是否触发限流操作，所有枚举值如下：

```
public enum MetricEvent {
    PASS, // Normal pass.
    BLOCK, // Normal block.
    EXCEPTION,
    SUCCESS,
    RT,
    OCCUPIED_PASS
}
```

Slot

slot是另一个sentinel中非常重要的概念，sentinel的工作流程就是围绕着一个个插槽所组成的插槽链来展开的。需要注意的是每个插槽都有自己的职责，他们各司其职完好的配合，通过一定的编排顺序，来达到最终的限流降级的目的。默认的各个插槽之间的顺序是固定的，因为有的插槽需要依赖其他的插槽计算出来的结果才能进行工作。

但是这并不意味着我们只能按照框架的定义来，sentinel 通过 SlotChainBuilder 作为 SPI 接口，使得 Slot Chain 具备了扩展的能力。我们可以通过实现 SlotsChainBuilder 接口加入自定义的 slot 并自定义编排各个 slot 之间的顺序，从而可以给 sentinel 添加自定义的功能。

那SlotChain是在哪创建的呢？是在 CtSph.lookProcessChain() 方法中创建的，并且该方法会根据当前请求的资源先去一个静态的HashMap中获取，如果获取不到才会创建，创建后会保存到HashMap中。这就意味着，同一个资源会全局共享一个SlotChain。默认生成ProcessorSlotChain为：

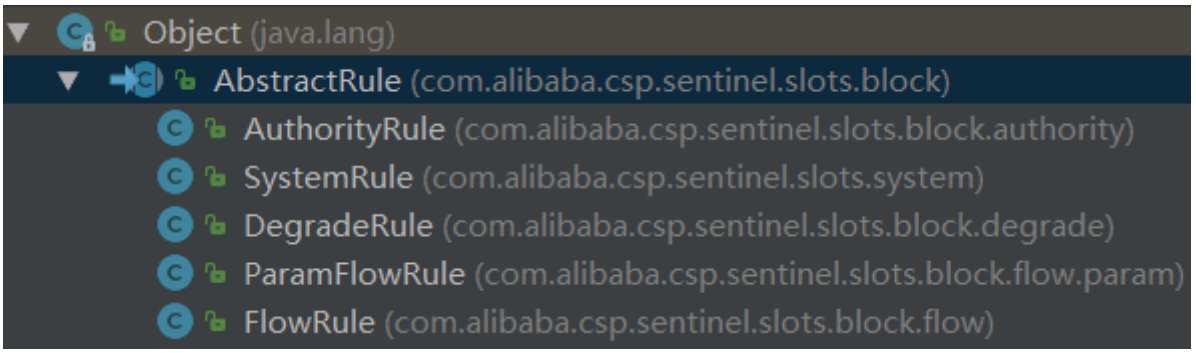
```
// DefaultSlotChainBuilder
public ProcessorSlotChain build() {
 ProcessorSlotChain chain = new DefaultProcessorSlotChain();
 chain.addLast(new NodeSelectorSlot());
 chain.addLast(new ClusterBuilderSlot());
 chain.addLast(new LogSlot());
 chain.addLast(new StatisticSlot());
 chain.addLast(new SystemSlot());
 chain.addLast(new AuthoritySlot());
  chain.addLast(new FlowSlot());
  chain.addLast(new DegradeSlot());

 return chain;
}
```

到这里本文结束了，谢谢小伙伴们的阅读~ 在理解了这些核心概念之后，相信聪明的你回过头再看sentinel源码就不会觉得有难度了：)

sentinel规则接口定义如下：

```
1  public interface Rule {
2   // 是否能通过规则检查
3   boolean passCheck(Context context, DefaultNode node, int count,
Object... args);
4  }
5  public abstract class AbstractRule implements Rule {
6   // 资源名，针对哪个资源的规则
7   private String resource;
8   // 针对的调用来源，若为 default 则不区分调用来源
9   private String limitApp;
10  }
```

规则实现类有如下几种，涉及授权、系统、降级、流控规则等。



下面分别看下对应的规则细节：（父类中的resource和limitApp不再列出）

流控 FlowRule

- count: 限流阈值
- grade: 限流阈值类型（QPS 或并发线程数）
- strategy: 调用关系限流策略
- controlBehavior: 流量控制效果（直接拒绝、Warm Up、匀速排队）

黑白名单 AuthorityRule

- strategy：黑白名单校验策略，0 for whitelist; 1 for blacklist.

熔断降级 DegradeRule

- count：RT threshold or exception ratio threshold count.
- timeWindow：降级持续时间
- grade：降级策略，0: average RT, 1: exception ratio

系统负载 SystemRule

- highestSystemLoad：最高系统负载
- qps：最大QPS

- avgRt：平均RT
- maxThread：最大线程

注意：系统服务规则是针对全局的，不是针对单个resource。每种Rule都有自己的管理器，一般都是xxxRuleManager。

sentinel 处理流程是基于slot链(ProcessorSlotChain)来完成的，比如限流、熔断等，其中重要的一个slot就是StatisticSlot，它是做各种数据统计的，而限流/熔断的数据判断来源就是StatisticSlot，StatisticSlot的各种数据统计都是基于滑动窗口来完成的，因此本文就重点分析StatisticSlot的滑动窗口统计机制。

sentinel 的slot链(ProcessorSlotChain)是责任链模式的体现，那SlotChain是在哪创建的呢？是在CtSph.lookProcessChain() 方法中创建的，并且该方法会根据当前请求的资源先去一个静态的HashMap中获取，如果获取不到才会创建，创建后会保存到HashMap中。这就意味着，同一个资源会全局共享一个SlotChain。默认生成ProcessorSlotChain逻辑为：

```
// DefaultSlotChainBuilder
public ProcessorSlotChain build() {
 ProcessorSlotChain chain = new DefaultProcessorSlotChain();
 chain.addLast(new NodeSelectorSlot());
 chain.addLast(new ClusterBuilderSlot());
 chain.addLast(new LogSlot());
 chain.addLast(new StatisticSlot());
 chain.addLast(new SystemSlot());
 chain.addLast(new AuthoritySlot());
  chain.addLast(new FlowSlot());
  chain.addLast(new DegradeSlot());

  return chain;
 }
```

整个处理过程从第一个slot往后一直传递到最后一个的，当到达StatisticSlot时，开始统计各项指标，统计的结果又会被后续的Slot所采用，作为各种规则校验的依据。各种指标如下：

```
public enum MetricEvent {
 PASS, // Normal pass.
 BLOCK, // Normal block.
 EXCEPTION, // 异常统计
 SUCCESS,
 RT, // rt统计
 OCCUPIED_PASS
}
```

StatisticSlot.entry流程

处理流程走到StatisticSlot时，首先触发后续slot.entry方法，然后统计各项指标，后续slot中数据判断来源就是这里统计的各项指标。StatisticSlot.entry 逻辑如下：

```
@Override
public void entry(Context context, ResourceWrapper resourceWrapper, DefaultNode node, int count, Object... args) throws Throwable {
 try {
 // 触发下一个Slot的entry方法
 fireEntry(context, resourceWrapper, node, count, args);
 // 如果能通过SlotChain中后面的Slot的entry方法，说明没有被限流或降级
 // 统计信息
 node.increaseThreadNum();
 node.addPassRequest();
  // 省略部分代码
  } catch (BlockException e) {
  context.getCurEntry().setError(e);
  // Add block count.
  node.increaseBlockedQps();
  // 省略部分代码
  throw e;
  } catch (Throwable e) {
  context.getCurEntry().setError(e);
  // Should not happen
  node.increaseExceptionQps();
  // 省略部分代码
  throw e;
  }
}
```

由以上代码可知，StatisticSlot主要就做了3件事：

1. 触发后续slot的entry方法，进行规则校验
2. 校验通过则更新node实时指标数据
3. 校验不通过则更新node异常指标数据

注意：由于后续的fireEntry操作和更新本次统计信息是两个操作，不是原子的，会造成限流不准的小问题，比如设置的FlowRule count为20，并发情况下可能稍大于20，不过针对大部分场景来说，这点偏差是可以容忍的，毕竟我们要的是限流效果，而不是必须精确的限流操作。

更新node实时指标数据

我们可以看到 node.addPassRequest() 这段代码是在fireEntry执行之后执行的，这意味着，当前请求通过了sentinel的流控等规则，此时需要将当次请求记录下来，也就是执行node.addPassRequest() 这行代码，具体的代码如下所示：

```
// DefaultNode
public void addPassRequest() {
 super.addPassRequest();
 this.clusterNode.addPassRequest();
}
```

这里的node是一个 DefaultNode 实例，这里特别补充一个 DefaultNode 和 ClusterNode 的区别：

- DefaultNode：保存着某个resource在某个context中的实时指标，每个DefaultNode都指向一个ClusterNode。
- ClusterNode：保存着某个resource在所有的context中实时指标的总和，同样的resource会共享同一个ClusterNode，不管他在哪个context中。

上面代码不管是 DefaultNode 还是 ClusterNode ， 走的都是StatisticNode 对象的 addPassRequest 方法：

```
private transient volatile Metric rollingCounterInSecond = new ArrayMetric(2, 1000);
private transient Metric rollingCounterInMinute = new ArrayMetric(60, 60 * 1000);

public void addPassRequest(int count) {
 rollingCounterInSecond.addPass(count); // 对每秒指标统计
 rollingCounterInMinute.addPass(count); // 每分钟指标统计
}
```

每一个通过的指标（pass）都是调用Metric 的接口进行操作的，并且是通过 ArrayMetric 这种实现类，代码如下：

```
public ArrayMetric(int windowLength, int interval) {
 this.data = new WindowLeapArray(windowLength, interval);
}

public void addPass(int count) {
 // 获取当前时间窗口
 WindowWrap<MetricBucket> wrap = data.currentWindow();
```

```
 wrap.value().addPass(count);
}
```

首先通过 currentWindow() 获取当前时间窗口，然后更新当前时间窗口对应的统计指标，以下代码重点关注几个判断逻辑：

```
// LeapArray
public WindowWrap<T> currentWindow() {
 return currentWindow(TimeUtil.currentTimeMillis());
}
// TimeUtil
public static long currentTimeMillis() {
 // currentTimeMillis是由一个tick线程每个1ms更新一次，具体逻辑在TimeUtil类中
 return currentTimeMillis;
}
// LeapArray
public WindowWrap<T> currentWindow(long timeMillis) {
 // 计算当前时间点落在滑动窗口的下标
 int idx = calculateTimeIdx(timeMillis);
 // Calculate current bucket start time.
 long windowStart = calculateWindowStart(timeMillis);

 // 获取当前时间点对应的windowWrap，array为AtomicReferenceArray
 while (true) {
 WindowWrap<T> old = array.get(idx);
 if (old == null) {
 // 1.为空表示当前时间窗口为初始化过，创建WindowWrap并cas设置到array中
 WindowWrap<T> window = new WindowWrap<T>(windowLengthInMs, windowStart,
newEmptyBucket());
 if (array.compareAndSet(idx, null, window)) {
 return window;
 } else {
 Thread.yield();
 }
 } else if (windowStart == old.windowStart()) {
 // 2.获取的时间窗口正好对应当前时间，直接返回
 return old;
 } else if (windowStart > old.windowStart()) {
 // 3.获取的时间窗口为老的，进行reset操作复用
```

```
33   if (updateLock.tryLock()) {
34   try {
35   return resetWindowTo(old, windowStart);
36   } finally {
37   updateLock.unlock();
38   }
39   } else {
40   Thread.yield();
41   }
42   } else if (windowStart < old.windowStart()) {
43   // 4.时间回拨了，正常情况下不会走到这里
44   return new WindowWrap<T>(windowLengthInMs, windowStart,
newEmptyBucket());
45   }
46   }
47 }
```

获取当前时间窗口对应的WindowWrap之后，就可以进行更新操作了。

```
1 // wrap.value().addPass(count);
2 public void addPass(int n) {
3  add(MetricEvent.PASS, n);
4 }
5 // MetricBucket
6 public MetricBucket add(MetricEvent event, long n) {
7  // 对应MetricEvent枚举中值
8  counters[event.ordinal()].add(n);
9  return this;
10 }
```

到这里为止，整个指标统计流程就完成了，下面重点看下滑动窗口机制。

滑动窗口机制

时间窗口是用WindowWrap对象表示的，其属性如下：

```
1 private final long windowLengthInMs; // 时间窗口的长度
2 private long windowStart; // 时间窗口开始时间
3 private T value; // MetricBucket对象，保存各个指标数据
```

sentinel时间基准由tick线程来做，每1ms更新一次时间基准，逻辑如下：

```
currentTimeMillis = System.currentTimeMillis();
Thread daemon = new Thread(new Runnable() {
 @Override
 public void run() {
 while (true) {
 currentTimeMillis = System.currentTimeMillis();
 try {
 TimeUnit.MILLISECONDS.sleep(1);
 } catch (Throwable e) {
  }
  }
  }
});
daemon.setDaemon(true);
daemon.setName("sentinel-time-tick-thread");
daemon.start();
```

sentinel默认有每秒和每分钟的滑动窗口，对应的LeapArray类型，它们的初始化逻辑是：

```
protected int windowLengthInMs; // 单个滑动窗口时间值
protected int sampleCount; // 滑动窗口个数
protected int intervalInMs; // 周期值（相当于所有滑动窗口时间值之和）

public LeapArray(int sampleCount, int intervalInMs) {
 this.windowLengthInMs = intervalInMs / sampleCount;
 this.intervalInMs = intervalInMs;
 this.sampleCount = sampleCount;

  this.array = new AtomicReferenceArray<WindowWrap<T>>(sampleCount);
}
```

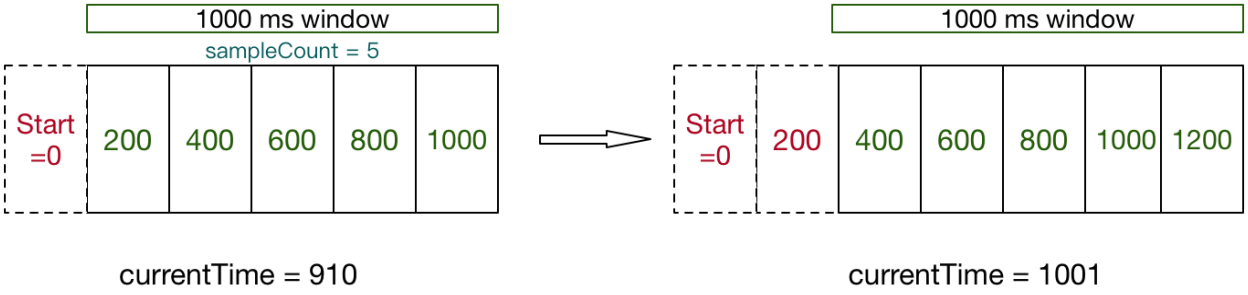
针对每秒滑动窗口，windowLengthInMs=500，sampleCount=2，intervalInMs=1000，针对每分钟滑动窗口，windowLengthInMs=1000，sampleCount=60，intervalInMs=60000，对应代码：

```
private transient volatile Metric rollingCounterInSecond = new ArrayMetric(2, 1000);
private transient Metric rollingCounterInMinute = new ArrayMetric(60, 60 * 1000);
```

> 小结

currentTimeMillis时间基准（tick线程）每1ms更新一次，通过currentWindow(timeMillis)方法获取当前时间点对应的WindowWrap对象，然后更新对应的各种指标，用于做限流、降级时使用。注意，当前时间基准对应的事件窗口初始化时lazy模式，并且会复用的。

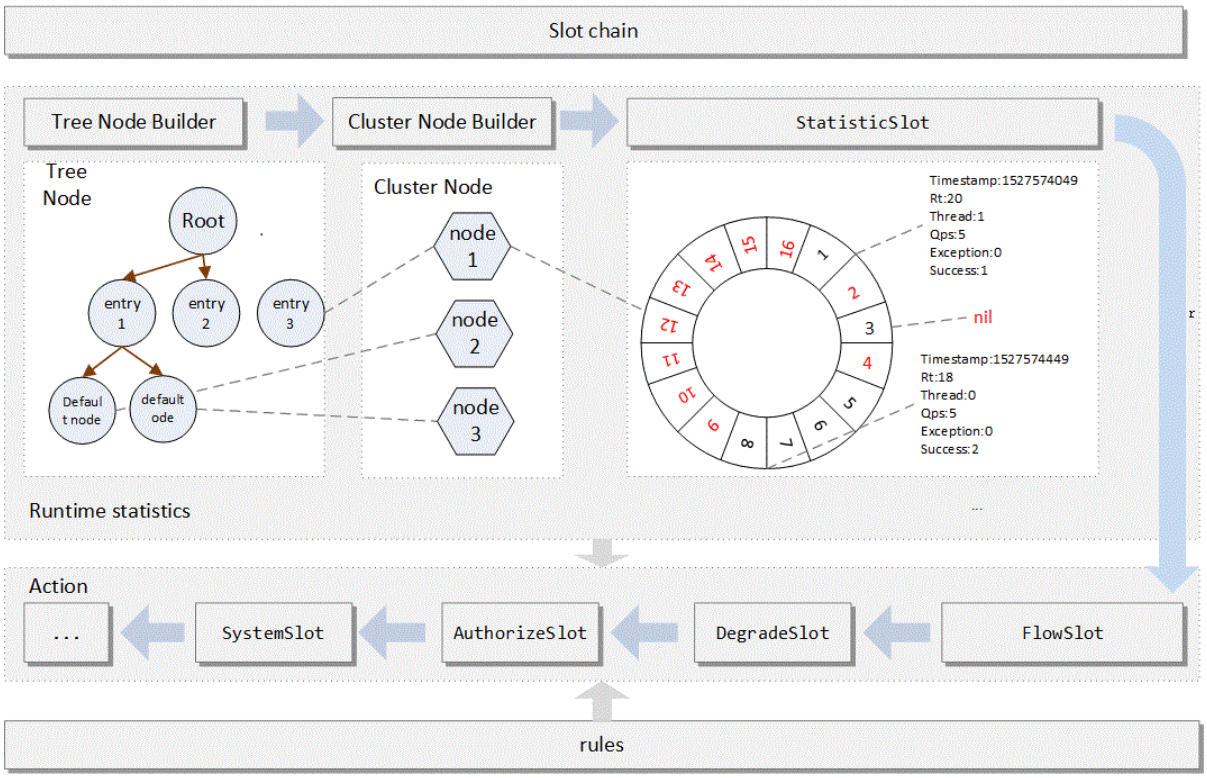
Sentinel 底层采用高性能的滑动窗口数据结构 LeapArray 来统计实时的秒级指标数据，可以很好地支撑写多于读的高并发场景。最后以一张图结束吧：



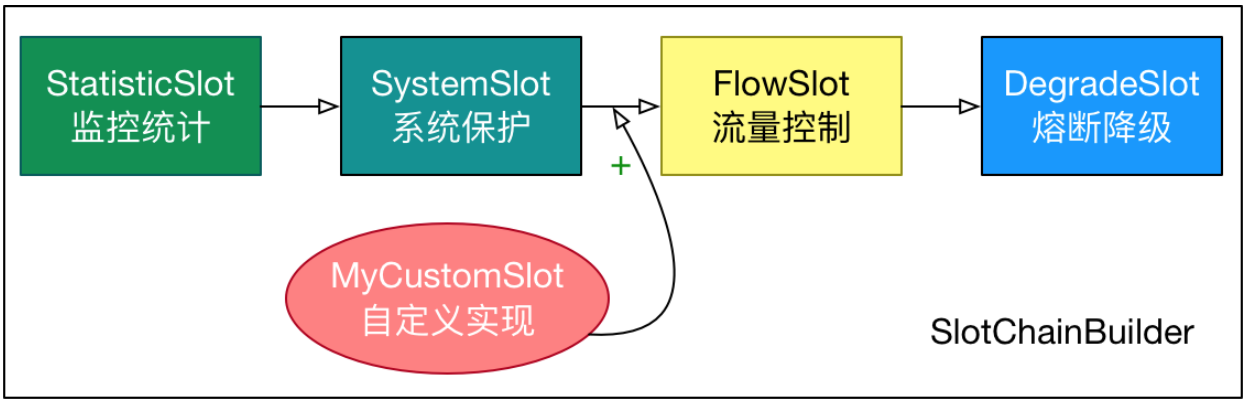
在 Sentinel 里面，所有的资源都对应一个资源名称以及一个 Entry。Entry 可以通过对主流框架的适配自动创建，也可以通过注解的方式或调用 API 显式创建；每一个 Entry 创建的时候，同时也会创建一系列功能插槽（slot chain）。这些插槽有不同的职责，例如:

- 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；

总的流程如下：



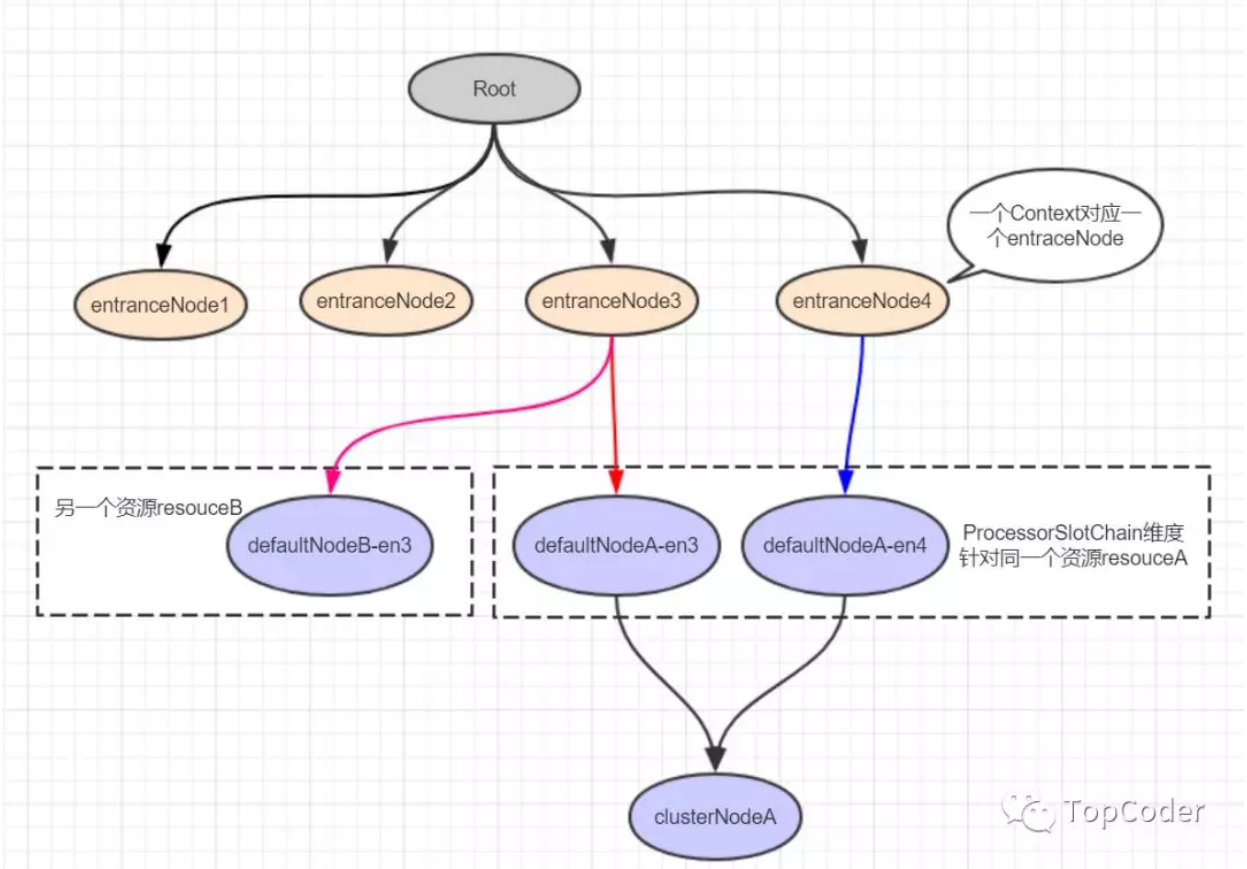
Sentinel 将 SlotChainBuilder（构建slot链）作为 SPI 接口进行扩展，使得 Slot Chain 具备了扩展的能力。您可以自行加入自定义的 slot 并编排 slot 间的顺序，从而可以给 Sentinel 添加自定义的功能。



大致看了sentinel执行流程中各个slot之后，下面就按照上述顺序分析下各自功能及实现。

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，可用于根据调用路径来限流降级。注意，每个资源都会对于一个slot链（NodeSelectorSlot为slot链中一环），在NodeSelectorSlot中针对每个context会创建一个DefaultNode结构，它们之间关系图如下所示：

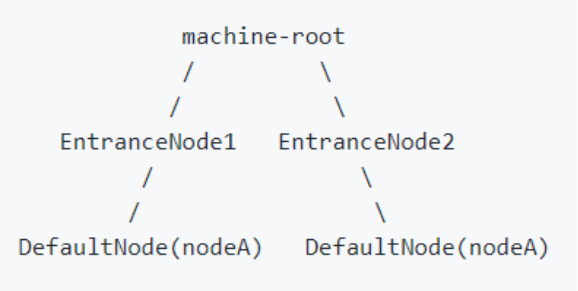


每个 DefaultNode 由资源 ID 和输入名称来标识。一个资源可以有多个不同入口的DefaultNode，换句话说，虽然是同一个resource，但是不同的context对应的是不同的defaultNode。

```
1 ContextUtil.enter("entrance1", "appA");
```

```
Entry nodeA = SphU.entry("nodeA");
if (nodeA != null) {
nodeA.exit();
}
ContextUtil.exit();

ContextUtil.enter("entrance2", "appA");
nodeA = SphU.entry("nodeA");
 if (nodeA != null) {
 nodeA.exit();
 }
 ContextUtil.exit();
```

以上代码将在内存中生成以下结构：



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 Context.enter(contextName，origin) 中的 origin 标记。

```
private ClusterNode clusterNode = null;
// ClusterBuilderSlot
@Override
public void entry(Context context, ResourceWrapper resourceWrapper, Defau
ltNode node, int count,
 boolean prioritized, Object... args)
 throws Throwable {
 if (clusterNode == null) {
 // Create the cluster node.
 clusterNode = Env.nodeBuilder.buildClusterNode();
 HashMap<ResourceWrapper, ClusterNode> newMap = new HashMap<ResourceWrap
per, ClusterNode>(16);
 newMap.putAll(clusterNodeMap);
```

```
12   newMap.put(node.getId(), clusterNode);
13   clusterNodeMap = newMap;
14   }
15   node.setClusterNode(clusterNode);
16
17   // 标记orgin来源
18   if (!"".equals(context.getOrigin())) {
19   Node originNode = node.getClusterNode().getOrCreateOriginNode(context.g
etOrigin());
20   context.getCurEntry().setOriginNode(originNode);
21   }
22   fireEntry(context, resourceWrapper, node, count, prioritized, args);
23  }
```

同一个resoure对应一个chainSlot，其中ClusterBuilderSlot会新建resource所有不同defaultNode的clusterNode，用户存放资源统计信息。

LogSlot

logSlot功能较为简单，就是在出现业务异常时打印错误日志信息，注意这里的业务异常是指sentinel内部发生了异常导致，也无异常这里捕获之后打印日志不在抛出。

StatisticSlot

StatisticSlot 是 Sentinel 的核心功能插槽之一，用于统计实时的调用数据。具体statisticSlot细节参考sentinel滑动窗口统计机制。

```
1  public void entry(Context context, ResourceWrapper resourceWrapper, Defau
ltNode node, int count,
2   boolean prioritized, Object... args) throws Throwable {
3   try {
4   // 触发下一个slot的entry方法
5   fireEntry(context, resourceWrapper, node, count, prioritized, args);
6   // 如果能通过SlotChain中后面的Slot的entry方法，说明没有被限流或降级，则统计数
据
7   node.increaseThreadNum();
8   node.addPassRequest(count);
9   } catch (BlockException e) {
10   // xxx
```

```
    throw e;
    } catch (Throwable e) {
    // xxx
    throw e;
    }
}
```

SystemSlot

Sentinel 系统自适应限流从整体维度对应用入口流量进行控制，结合应用的 Load、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。注意这个功能的两个限制:

- 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() 指定调用类型，如果不指定，默认是EntryType.OUT。
- 只在 Unix-like 的操作系统上生效。

AuthoritySlot

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。

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));
```

FlowSlot

流量控制（flow control），其原理是监控应用流量的 QPS 或并发线程数等指标，当达到指定的阈值时对流量进行控制，以避免被瞬时的流量高峰冲垮，从而保障应用的高可用性。FlowSlot 会根据预设的规则，结合前面 NodeSelectorSlot、ClusterNodeBuilderSlot、StatisticSlot 统计出来的实时信息进行流量控制。更多流量控制资料可参考：流量控制

DegradeSlot

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。由于调用关系的复杂性，如果调用链路中的某个资源不稳定，最终会导致请求发生堆积。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（默认行为是抛出 DegradeException）。
更多熔断降级资料可参考：熔断降级

参考资料：
1、Sentinel工作主流程

为什么需要集群流控呢？假设需要将某个API的总qps限制在100，机器数可能为50，这时很自然的想到使用一个专门的server来统计总的调用量，其他实例与该server通信来判断是否可以调用，这就是基本的集群流控方式，sentinel的实现就是这样的。

如果服务调用使用轮训或者随机路由方式，理论上可以通过在各个单机上设置流控规则即可（单机qps上限=总qps上限 / 机器数）。集群流控可以解决流量分配不均的问题导致总体流控效果不佳的问题，其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，不过由于会与server进行通信，所以性能上会有一定损耗。

集群流控中共有两种身份：

- 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）。

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

大致了解集群流控概念之后，下面一起分析下集群流控规则、client端和server端各自处理机制~

集群流控规则

FlowRule 添加了两个字段用于集群限流相关配置，如下所示。clusterMode在方法FlowRuleChecker.canPassCheck中会用到进行判断是否是集群流控，false表示单机流控；true表示集群流控，会调用方法passClusterCheck与集群流控server端通信判断是否触发了流控，此时异常降级策略为本地流控(*fallbackToLocalOrPass方法，fallbackToLocalWhenFail属性为true时执行本地流控，否则直接返回ture不走流控检查*)。

```
1 private boolean clusterMode; // 标识是否为集群限流配置
2 private ClusterFlowConfig clusterConfig; // 集群限流相关配置项
```

```
3
4  // ClusterFlowConfig属性
5  private Long flowId; // 全局唯一的规则 ID，由集群限流管控端分配.
6  private int thresholdType = ClusterRuleConstant.FLOW_THRESHOLD_AVG_LOCAL;
// 阈值模式，默认（0）为单机均摊，1 为全局阈值.
7  private int strategy = ClusterRuleConstant.FLOW_CLUSTER_STRATEGY_NORMAL;
8  private boolean fallbackToLocalWhenFail = true; // 在 client 连接失败或通信
失败时，是否退化到本地的限流模式
9
10  public boolean canPassCheck(/*@NonNull*/ FlowRule rule, Context context,
DefaultNode node, int acquireCount, boolean prioritized) {
11   String limitApp = rule.getLimitApp();
12   if (limitApp == null) {
13   return true;
14   }
15   if (rule.isClusterMode()) {// 集群模式
16   return passClusterCheck(rule, context, node, acquireCount,
prioritized);
17   }
18   // 单机模式流控
19   return passLocalCheck(rule, context, node, acquireCount, prioritized);
20  }
```

- **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。单机均摊模式表示总qps阈值等于机器数*单机qps阈值；全局阈值等于整个集群配置的阈值。
- **strategy** 集群策略，默认FLOW_CLUSTER_STRATEGY_NORMAL，针对ClusterFlowConfig配置该属性为FLOW_CLUSTER_STRATEGY_NORMAL才合法，除此之外，暂无太多业务意义。

client端处理机制

client端的处理机制和单机是一样的，只不过clusterMode和clusterConfig属性配置上了而已，具体的client使用可以参考官方文档 集群流控，这里不再赘述。如果是集群流控，在FlowRuleChecker.canPassCheck方法中会调用方法passClusterCheck，如下：

```
1  private static boolean passClusterCheck(FlowRule rule, Context context, D
efaultNode node, int acquireCount,
2   boolean prioritized) {
```

```
3   try {
4   TokenService clusterService = pickClusterService();
5   if (clusterService == null) {
6   // 为null降级处理
7   return fallbackToLocalOrPass(rule, context, node, acquireCount, prioriti
zed);
8   }
9   long flowId = rule.getClusterConfig().getFlowId();
10   TokenResult result = clusterService.requestToken(flowId, acquireCount,
prioritized);
11   return applyTokenResult(result, rule, context, node, acquireCount, prio
ritized);
12   } catch (Throwable ex) {
13   RecordLog.warn("[FlowRuleChecker] Request cluster token unexpected fail
ed", ex);
14   }
15   // 降级处理 本地限流
16   return fallbackToLocalOrPass(rule, context, node, acquireCount, priorit
ized);
17  }
```

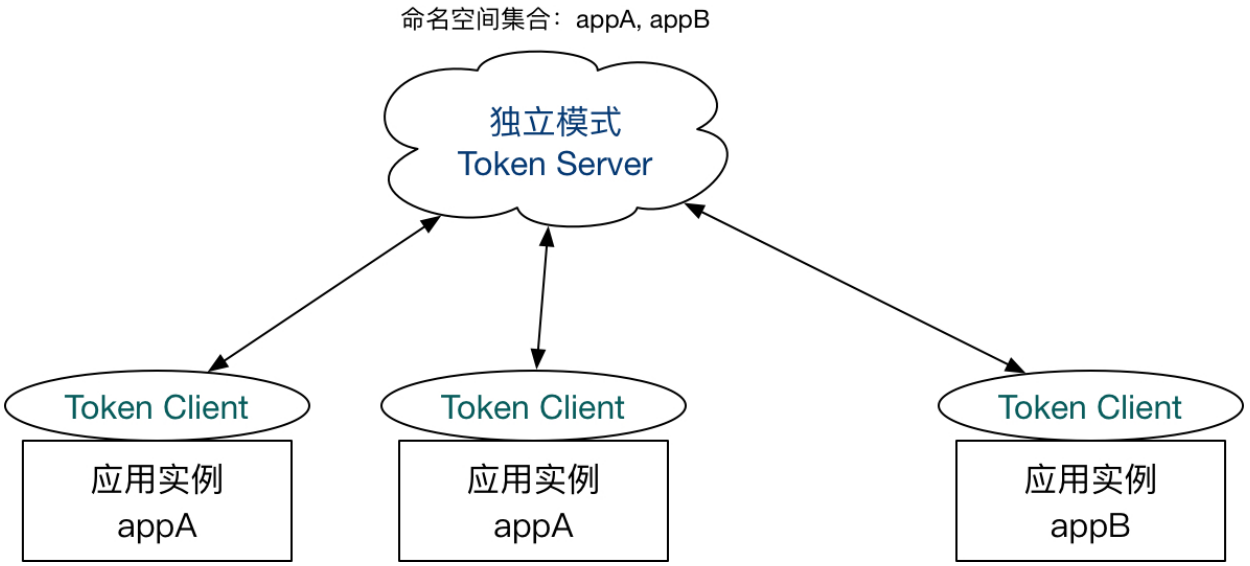
requestToken负责与token server端通信，入参包括 flowId, acquireCount, prioritized，这里是没有Resource信息的，server端通过flowid来获取对应规则进行流控判断。注意，调用writeAndFlush发送请求之后等待响应结果，最大等待时间ClusterClientConfigManager.getRequestTimeout()；请求发送过程中，出现任何异常或者返回错误（这里不包括BLOCKED情况），都会默认走降级本地流控逻辑：fallbackToLocalOrPass。

了解了client端处理流程，接下来看下server端处理流程，client和server端都是用netty作为底层网络通信服务，关于netty的原理不是本文讨论的重点因此会简单带过。如果小伙伴们还不太熟悉netty，请参阅对应资料即可。对于netty，每个Java开发者都需要了解甚至是熟悉的，这样不仅仅帮助我们理解NIO及Reactor模型，还能再阅读基于netty的框架源码（比如dubbo/rocketmq等）时，将重点关注在框架本身实现上，而不是网络通信流程及细节上。

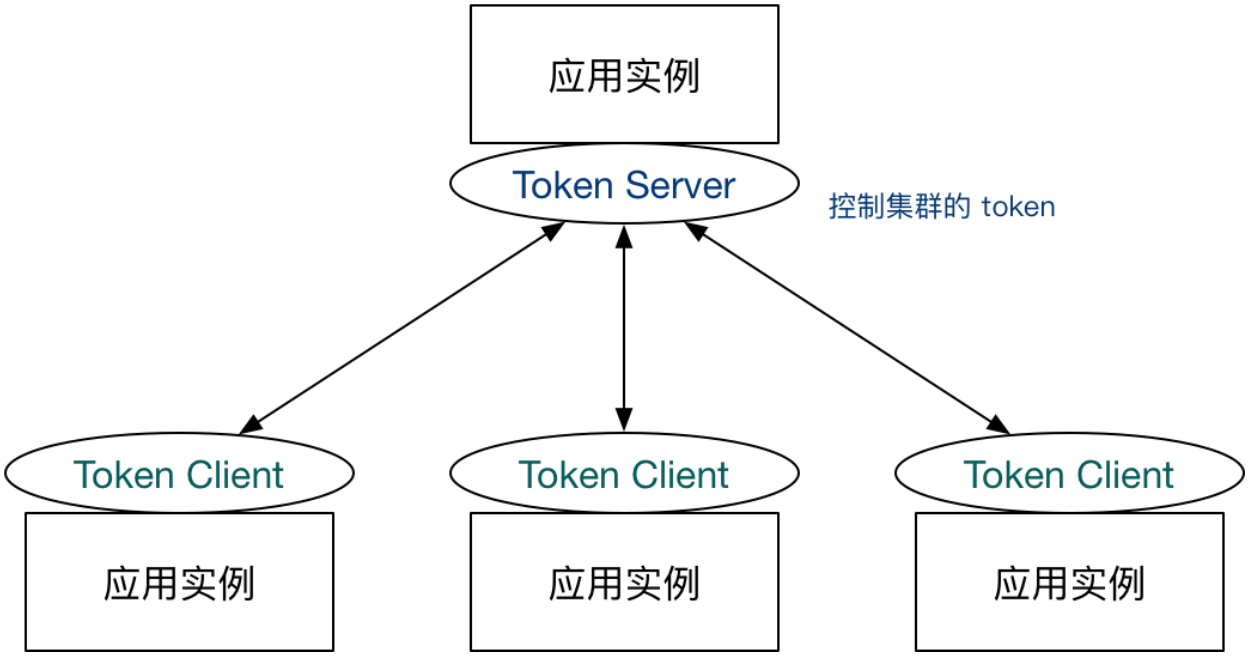
server端处理机制

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。



目前针对token server高可用，sentinel并没有对应的解决方案，不过没有并不意味着没考虑，因为默认可以降级走本地流控。sentinel作为一个限流组件，在大部分应用场景中，如果token server挂了降级为本地流控就可以满足了。

如果必须考虑token server高可用，可考虑token server集群部署，每个token server都能访问（或存储）全量规则数据，多个client通过特定路由规则分配到不同的token server（相同类型服务路由到同一个token server，不同类型服务可路由到不同token server），token server故障时提供failover机制即可。如果此时考虑到相同类型服务出现网络分区，也就是一部分服务可以正常与token server通信，另一个部分服务无法正常与token server通信，如果无法正常通信的这部分服务直接进行failover，会导致集群限流不准的问题，可通过zookeeper来保存在线的token server，如果zookeeper中token server列表有变化，再进行failover；此情况下再出现任何形式的网络分区，再执行降级逻辑，执行本地限流。

server端不管是独立模式还是嵌入模式，都是通过NettyTransportServer来启动的：

```
public void start() {
 ServerBootstrap b = new ServerBootstrap();
 b.group(bossGroup, workerGroup)
 .channel(NioServerSocketChannel.class)
 .childHandler(new ChannelInitializer<SocketChannel>() {
 @Override
 public void initChannel(SocketChannel ch) throws Exception {
 ChannelPipeline p = ch.pipeline();
 p.addLast(new LengthFieldBasedFrameDecoder(1024, 0, 2, 0, 2));
  p.addLast(new NettyRequestDecoder());
  p.addLast(new LengthFieldPrepender(2));
  p.addLast(new NettyResponseEncoder());
  p.addLast(new TokenServerHandler(connectionPool));
  }
  });
  b.bind(port).addListener(new GenericFutureListener<ChannelFuture>() {
  //
  });
 }
```

以上逻辑主要是netty启动逻辑，重点关注initChannel方法，这些是往pipeline添加自定义channelHandler，主要是处理处理粘包、编解码器和业务处理Handler，这里最重要的是TokenServerHandler，因为是请求处理逻辑，所以重点关注其channelRead方法：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exc
eption {
 // 全局保存channel
 globalConnectionPool.refreshLastReadTime(ctx.channel());
 if (msg instanceof ClusterRequest) {
```

```
5   ClusterRequest request = (ClusterRequest)msg;
6   if (request.getType() == ClusterConstants.MSG_TYPE_PING) {
7   // ping请求处理，会记录namespace信息
8   handlePingRequest(ctx, request);
9   return;
10   }
11   // 根据request type获取对应处理器
12   // 针对集群流控，type为MSG_TYPE_FLOW
13   RequestProcessor<?, ?> processor = RequestProcessorProvider.getProcesso
r(request.getType());
14   ClusterResponse<?> response = processor.processRequest(request);
15   writeResponse(ctx, response);
16   }
17  }
```

针对集群流控，type为MSG_TYPE_FLOW，对应处理器为FlowRequestProcessor。首先会提取请求入参 flowId, acquireCount, prioritized，主要步骤如下：

- 根据flowId获取规则，为空返回结果NO_RULE_EXISTS；
- 获取请求namespace对应的RequestLimiter，非空时进行tryPass限流检查，该检查是针对namespace维度；
- 针对flowId对应规则进行限流检查，acquireCount表示该请求需要获取的token数，数据检查基于滑动时间窗口统计来判断的。

根据限流规则检查之后，会统计相关的PASS/BLOCK/PASS_REQUEST/BLOCK_REQUEST等信息，该流程和单机流控流程是类似的，具体代码不再赘述。处理完成之后，会返回client端处理结果，至此整个集群流控流程就分析完了。

sentinel针对目前常见的主流框架都做了适配，比如dubbo、Web Servlet、Spring Cloud、Spring WebFlux等。sentinel的适配做到了开箱即用，那么它是通过什么机制来实现的呢？这里大家可以思考下，如果一个框架本身没有扩展机制（这只是举个极端的例子，一般开源框架都是有自身的扩展机制的），那么sentinel是无法进行适配的，更谈不上开箱即用，除非更改框架源码。所以说，如果明白了框架的扩展机制，那么理解sentinel的适配机制就很easy了，比如dubbo 本身有Filter机制（consumer端和provider端都有），Web servlet也有自己的Filter机制可进行自定义扩展。

关于sentinel dubbo的使用示例，因为官方文档已经有详细说明了，这里不再赘述了。下面主要分析sentinel dubbo的实现原理。

因为dubbo提供有Filter机制，默认需要在 META-INF\dubbo\org.apache.dubbo.rpc.Filter文件中进行配置，sentinel dubbo的配置如下：

```
sentinel.dubbo.provider.filter=com.alibaba.csp.sentinel.adapter.dubbo.SentinelDubboProviderFilter
sentinel.dubbo.consumer.filter=com.alibaba.csp.sentinel.adapter.dubbo.SentinelDubboConsumerFilter
dubbo.application.context.name.filter=com.alibaba.csp.sentinel.adapter.dubbo.DubboAppContextFilter
```

此处可见，consumer端和provider端各对应一个filter类，这里以SentinelDubboProviderFilter为例进行分析：

```
@Activate(group = "provider")
public class SentinelDubboProviderFilter implements Filter {
 @Override
 public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
  // Get origin caller.
  String application = DubboUtils.getApplication(invocation, "");

  Entry interfaceEntry = null;
  Entry methodEntry = null;
  try {
  // resourceName格式为 "接口:方法(入参1,入参2)"
  String resourceName = DubboUtils.getResourceName(invoker, invocation);
  String interfaceName = invoker.getInterface().getName();
  // Only need to create entrance context at provider side, as context will take effect
  // at entrance of invocation chain only (for inbound traffic).
  // 以resourceName作为context name
```

```
17   ContextUtil.enter(resourceName, application);
18   interfaceEntry = SphU.entry(interfaceName, EntryType.IN);
19   methodEntry = SphU.entry(resourceName, EntryType.IN, 1, invocation.getA
rguments());
20
21   Result result = invoker.invoke(invocation);
22   if (result.hasException()) {
23   Throwable e = result.getException();
24   // Record common exception.
25   Tracer.traceEntry(e, interfaceEntry);
26   Tracer.traceEntry(e, methodEntry);
27   }
28   return result;
29   } catch (BlockException e) {
30   return DubboFallbackRegistry.getProviderFallback().handle(invoker, invo
cation, e);
31   } catch (RpcException e) {
32   Tracer.traceEntry(e, interfaceEntry);
33   Tracer.traceEntry(e, methodEntry);
34   throw e;
35   } finally {
36   if (methodEntry != null) {
37   methodEntry.exit(1, invocation.getArguments());
38   }
39   if (interfaceEntry != null) {
40   interfaceEntry.exit();
41   }
42   ContextUtil.exit();
43   }
44   }
45 }
```

SentinelDubboProviderFilter中会对两个维度进行SphU.entry操作：

- 接口维度：resouce name为interfaceName；
- 方法维度：resouce name为方法签名，格式为 "接口:方法(入参1,入参2)"。

处理流程是，先获取接口维度的Resource，再获取方法维度的Resource，二者都获取成功之后，再执行后续的dubbo invoker操作，也就是后续的RPC处理。通过接口和方法两个不同维度，在provider端进行流控更加灵活。

看到这块代码时，笔者有一个疑问：

针对dubbo provider端的流控，SentinelDubboProviderFilter.invoke方法中会先对interfaceName做SphU.entry操作，然后在对method签名做SphU.entry操作，二者通过后再执行invoke后续操作。因为二者不是原子的，有可能针对interfaceName的pass，但是针对method签名的blocked，但是这个时候已经增加了interfaceName对应的pass统计值，这样同一个时间窗口内会影响到针对该接口其他方法的dubbo rpc调用。目前从代码来看，sentinel并未处理这种情况，因为目前没有针对资源的统计值做decrement功能。

试下一下，如果需要解决该问题，应该如何做呢？

- 方案一：增加一个统计值decrement功能，如果针对method签名执行SphU.entry操作被blocked时，调用统计值decrement功能，撤销之前相同时间窗口内针对interfaceName的pass值；
- 方案二：像这种需要针对两个resource做SphU.entry操作的场景，可以在判断是否通过pass时同时判断这两个resource对应的统计值是否满足规则限制，让这两个Resource产生关联，一同判断即可。

笔者倾向于方案二的实现，其实现流程和单个Resource的类似（只不过新增个Resource判断条件），而不像方案一那样需要提供新的方法+增加撤销逻辑来满足，严格来讲，方案一在对资源申请和撤销操作之间，也是会暂用一个pass名额的。