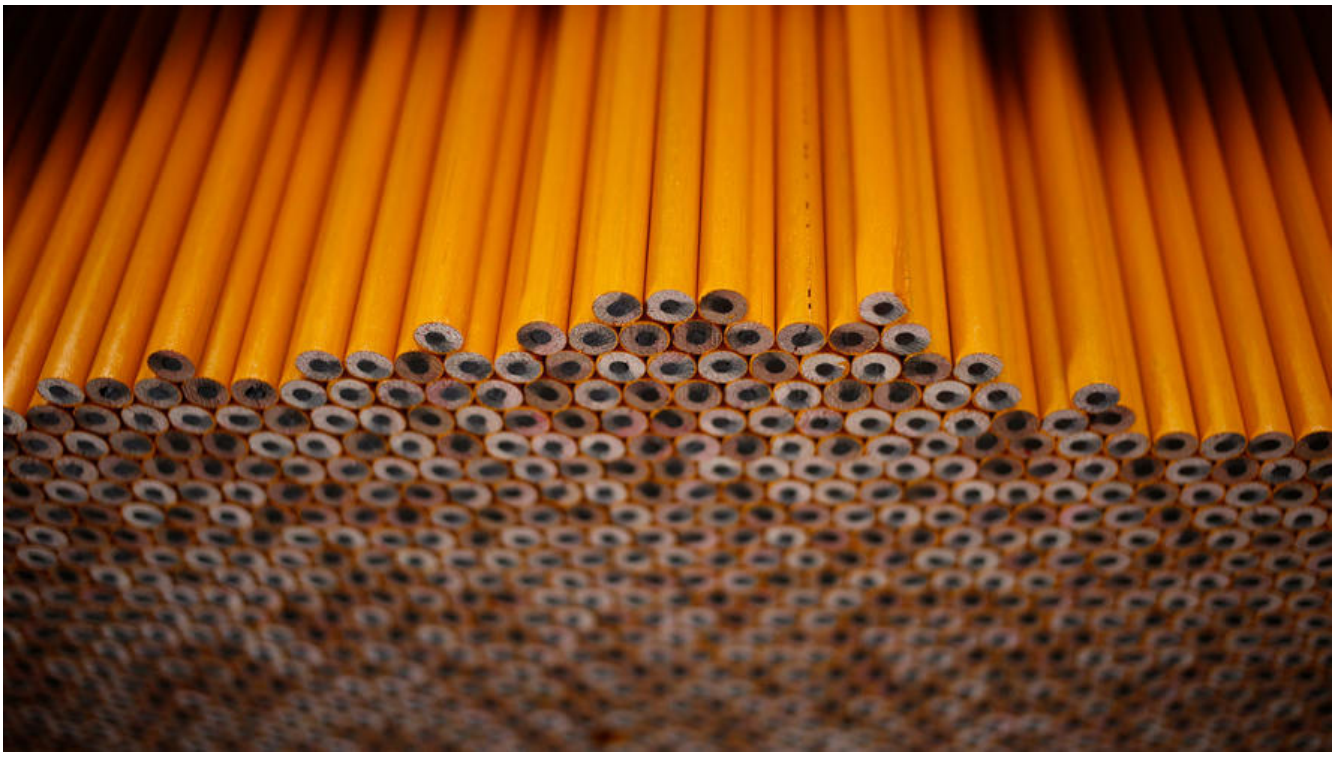
Sentinel 组件 (一)

作者：AlanSune
原文链接：https://ld246.com/article/1591528777554
来源网站：链滴



1. 介绍

当前许多项目承载着大量业务功能，在单一工程中进行开发会造成代码量剧增，为项目的开发、部署运维以及扩展造成障碍。在此背景下，微服务的出现有利于解决上述出现的问题，然而服务的稳定性造成巨大的困扰。

本章介绍的阿里 Sentinel 组件以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度护服务的稳定性。该组件的核心功能不依赖任何外部项目，同时官方称引入 Sentinel 带来的性能损非常小，只有在业务单机量级超过 25W QPS 的时候才会有一些显著的影响（10% 左右），单机 QPS 不太大的时候损耗几乎可以忽略不计。

Sentinel 具有以下特征:

- **丰富的应用场景**：Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀（即发流量控制在系统容量可以承受的范围）、消息削峰填谷、集群流量控制、实时熔断下游不可用应用。
- **完备的实时监控**：Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器级数据，甚至 500 台以下规模的集群的汇总运行情况。
- **广泛的开源生态**：Sentinel 提供开箱即用的与其它开源框架/库的整合模块，例如与 Spring Cloud Dubbo、gRPC 的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入 Sentinel。
- **完善的 SPI 扩展点**：Sentinel 提供简单易用、完善的 SPI 扩展接口。您可以通过实现扩展接口来快地定制逻辑。例如定制规则管理、适配动态数据源等。

更多项目的介绍请移步官方文档。

2. 基础

Sentinel组件以资源为单位，进行数据统计，为控制提供基础数据。每个资源都有一个资源名称，在次调用中会涉及到以下概念：

● **slot**：功能插槽。每个slot负责不同的职责，比如统计slot负责当前系统的数据统计，限流slot负责查限流规则是否成立。

● **slot chain**：功能插槽链。每一个资源都会涉及到不同的检查规则，所以在Sentinel中每个资源都有单独的slot chain，保证资源和slot chain一一对应。在请求资源X时，只需要执行资源X对应的slot hain即可。

● **Node**：统计节点，包含资源的全局数据统计，单次链路调用的统计等多个节点。

● **Entry**：每一次资源的调用都会创建一个Entry，它保存了本次的调用信息，如创建时间、当前的统节点等。一个Entry负责一次slot chain的执行。

● **Context**：调用链路上下文，通过ThreadLocal维持。在一次处理中，可能涉及到对一个资源的多请求（可以理解为一个线程中请求获取两次Entry），例如用户获取文章的信息，会涉及到获取文章阅读量和评论数，这些数据都保存在另外一个服务S中，那么可能就会调用两次资源S，所以创建两个Entry，但是只维护一个Context。

3. 执行流程

在项目中使用Sentinel，最基础的使用方式如下：

```
public static void main(String[] args) {
    // 不断进行资源调用.
    while (true) {
        Entry entry = null;
        try {
        entry = SphU.entry("HelloWorld");
            // 资源中的逻辑.
            System.out.println("hello world");
    } catch (BlockException e1) {
        System.out.println("blocked!");
    } finally {
       if (entry != null) {
          entry.exit();
       }
    }
    }
}
```

我们通过跟踪 **entry = SphU.entry("HelloWorld");** 语句的执行，来看看Sentinel的执行流程。在流程中，关键的代码是CtSph获取Entry的方法。

```
private Entry entryWithPriority(ResourceWrapper resourceWrapper, int count, boolean prioriti
ed, Object... args)
        throws BlockException {
        Context context = ContextUtil.getContext();
        if (context instanceof NullContext) {
            // The {@link NullContext} indicates that the amount of context has exceeded the thre
hold,
            // so here init the entry only. No rule checking will be done.
            return new CtEntry(resourceWrapper, null, context);
        }

        if (context == null) {
            // Using default context.
            context = InternalContextUtil.internalEnter(Constants.CONTEXT_DEFAULT_NAME);
```

```
        }

        // Global switch is close, no rule checking will do.
        if (!Constants.ON) {
            return new CtEntry(resourceWrapper, null, context);
        }

        ProcessorSlot<Object> chain = lookProcessChain(resourceWrapper);

        /*
         * Means amount of resources (slot chain) exceeds {@link Constants.MAX_SLOT_CHAIN_S
ZE},
         * so no rule checking will be done.
         */
        if (chain == null) {
            return new CtEntry(resourceWrapper, null, context);
        }

        Entry e = new CtEntry(resourceWrapper, chain, context);
        try {
            chain.entry(context, resourceWrapper, null, count, prioritized, args);
        } catch (BlockException e1) {
            e.exit(count, args);
            throw e1;
        } catch (Throwable e1) {
            // This should not happen, unless there are errors existing in Sentinel internal.
            RecordLog.info("Sentinel unexpected exception", e1);
        }
        return e;
    }
```

3.1. 规则开关

通过设置 **Constants.ON** 的值，可以控制是否进行规则检查。

```
if (!Constants.ON) {
    return new  CtEntry(resourceWrapper, null, context);
}
```

3.2. Context

```
Context context = ContextUtil.getContext();
if (context instanceof NullContext) {
    // The {@link NullContext} indicates that the amount of context has exceeded the threshold

    // so here init the entry only. No rule checking will be done.
    return new CtEntry(resourceWrapper, null, context);
}

if (context == null) {
    // Using default context.
    context = InternalContextUtil.internalEnter(Constants.CONTEXT_DEFAULT_NAME);
}
```

该部分获取上面提到的Conext对象，首先关注ContextUtil类：

```
private static ThreadLocal<Context> contextHolder = new ThreadLocal<>();

public static Context getContext() {
        return contextHolder.get();
    }
```

可以看见Sentinel为每一个线程实例化了一个Context对象，通过ThreadLocal进行保存，通常一次户请求在一个线程中完成，所以保证了一个线程拥有一份唯一的上下文。

一般在线程第一次获取上下文时，此方法返回为null，此时需要实例化，实例化方法如下：

```
protected static Context trueEnter(String name, String origin) {
        Context context = contextHolder.get();
        if (context == null) {
            Map<String, DefaultNode> localCacheNameMap = contextNameNodeMap;
            DefaultNode node = localCacheNameMap.get(name);
            if (node == null) {
                if (localCacheNameMap.size() > Constants.MAX_CONTEXT_NAME_SIZE) {
                    setNullContext();
                    return NULL_CONTEXT;
                } else {
                    try {
                        LOCK.lock();
                        node = contextNameNodeMap.get(name);
                        if (node == null) {
                            if (contextNameNodeMap.size() > Constants.MAX_CONTEXT_NAME_SIZE) {
                                setNullContext();
                                return NULL_CONTEXT;
                            } else {
                                node = new EntranceNode(new StringResourceWrapper(name, EntryType.
N), null);
                                // Add entrance node.
                                Constants.ROOT.addChild(node);

                                Map<String, DefaultNode> newMap = new HashMap<>(contextNameN
deMap.size() + 1);
                                newMap.putAll(contextNameNodeMap);
                                newMap.put(name, node);
                                contextNameNodeMap = newMap;
                            }
                        }
                    } finally {
                        LOCK.unlock();
                    }
                }
            }
            context = new Context(node, name);
            context.setOrigin(origin);
            contextHolder.set(context);
        }

        return context;
    }
```

由代码可知，Sentinel会创建一个 Context 放入到 **contextHolder** 中，下面关注下 **contextNameNodeMap**。该变量的申明如下：

```
private static volatile Map<String, DefaultNode> contextNameNodeMap = new HashMap<>();
```

此变量以Context的Name为key，保存了 **EntranceNode** 对象。在组件中，资源的调用路径以树状构存储起来，用于根据调用路径进行流量控制。在默认情况下，一次调用会形成以下结构：

```
          machine-root
            /
           /
      EntranceNode(sentinel_default_context)
          /
         /
    DefaultNode(nodeA)
```

上面 **EntranceNode** 是由上述代码生成的。注意在生成 **EntranceNode** 时，如果超过 **MAX_CONTEXT_NAME_SIZE**，就会返回 **NullContext** 类型的context。如果在一次调用中使用以下方法生成不的ContextName，那么就会形成下述结构。

```
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();
```

```
            machine-root
              /       \
             /         \
      EntranceNode1   EntranceNode2
           /             \
          /               \
   DefaultNode(nodeA)   DefaultNode(nodeA)
```

详细介绍见官方文档

3.3. slot chain

```
ProcessorSlot<Object> chain = lookProcessChain(resourceWrapper);
```

上述代码用于获取此次执行的slot chain，即一系列数据统计和规则检查的slot。

通过 **lookProcessChain** 方法，我们可以获取以下信息：

1. 上面提到每一个资源都有一份slot chain，该信息以资源的名称为key，保存于 **Ctsph::chainMap**。

2. 一个项目最多只能存在 **Constants.MAX_SLOT_CHAIN_SIZE**（默认6000）个资源，超过后不生效。

3. 通过 **SlotChainProvider::newSlotChain**方法获取slot chain，默认情况下会使用 **DefaultSlo ChainBuilder ** 构造slot chain，可以通过SPI进行单独的配置（SPI，即Service Provider Interface 具体详情可以自行搜索，后续文章会进行简单介绍）。

在获取slot chain后，构造出Entry对象，然后通过以下方法调用slot chain，依次执行每个slot，基本程结束。

```
chain.entry(context, resourceWrapper, null, count, prioritized, args);
```

在 **DefaultSlotChainBuilder** 中，通过以下方法返回slot chain，其中的每个slot将会在以后的文章详细介绍。

4. 后记

本文简单介绍了Sentinel，并讲述了基本的执行流程。在此基础上，后面会跟随源码学习Sentinel的同slot。