
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2018-11-15
Dubbo

精尽 Dubbo 源码分析 —— 过滤器（四）之 ActiveLimitFilter && ExecuteLimitFilter

本文基于 Dubbo 2.6.1 版本，望知悉。

1. 概述

本文分享服务方法的最大可并行调用的限制过滤器，在服务消费者和提供者各有一个 LimitFilter ：

ActiveLimitFilter ，在服务消费者，通过 <dubbo:reference /> 的 "actives" 统一配置项开启：

每服务消费者，每服务的每方法最大并发调用数。

ExecuteLimitFilter ，在服务提供者，通过 <dubbo:service /> 的 "executes" 统一配置项开启：

服务提供者，每服务的每方法最大可并行执行请求数。

另外，在 <dubbo:method /> 的 "actives" 和 "executes" 配置项，可以自定义每个方法的配置。

2. RpcStatus

com.alibaba.dubbo.rpc.RpcStatus ，RPC 状态。可以计入如下维度统计：

1. 基于服务 URL
2. 基于服务 URL + 方法

用于 ActiveLimitFilter 和 ExecuteLimitFilter 中。 当然，Dubbo 中，也有其他类，也会调用到 RpcStatus 。

2.1 构造方法

```
/**
 * 基于服务 URL 为维度的 RpcStatus 集合
```

```
 *
 * key：URL
 */
private static final ConcurrentMap<String, RpcStatus> SERVICE_STATISTICS = new ConcurrentHashMap<String, RpcStatus>()
/**
 * 基于服务 URL + 方法维度的 RpcStatus 集合
 *
 * key1：URL
 * key2：方法名
 */
private static final ConcurrentMap<String, ConcurrentMap<String, RpcStatus>> METHOD_STATISTICS = new ConcurrentHashMa

// 目前没有用到
private final ConcurrentMap<String, Object> values = new ConcurrentHashMap<String, Object>();
/**
 * 调用中的次数
 */
private final AtomicInteger active = new AtomicInteger();
/**
 * 总调用次数
 */
private final AtomicLong total = new AtomicLong();
/**
 * 总调用失败次数
 */
private final AtomicInteger failed = new AtomicInteger();
/**
 * 总调用时长，单位：毫秒
 */
private final AtomicLong totalElapsed = new AtomicLong();
/**
 * 总调用失败时长，单位：毫秒
 */
private final AtomicLong failedElapsed = new AtomicLong();
/**
 * 最大调用时长，单位：毫秒
 */
private final AtomicLong maxElapsed = new AtomicLong();
/**
 * 最大调用失败时长，单位：毫秒
 */
private final AtomicLong failedMaxElapsed = new AtomicLong();
/**
 * 最大调用成功时长，单位：毫秒
 */
private final AtomicLong succeededMaxElapsed = new AtomicLong();

/**
 * Semaphore used to control concurrency limit set by `executes`
 *
 * 服务执行信号量，在 {@link com.alibaba.dubbo.rpc.filter.ExecuteLimitFilter} 中使用
 */
private volatile Semaphore executesLimit;
/**
 * 服务执行信号量大小
 */
private volatile int executesPermits;
```

========== 静态属性 ==========

SERVICE_STATISTICS 属性，基于服务 URL 为维度的 RpcStatus 集合。#getStatus(url) 静态方法，获得 RpcStatus 对象，代码如下：

```
public static RpcStatus getStatus(URL url) {
    String uri = url.toIdentityString();
    // 获得
    RpcStatus status = SERVICE_STATISTICS.get(uri);
    // 不存在，则进行创建
    if (status == null) {
        SERVICE_STATISTICS.putIfAbsent(uri, new RpcStatus());
        status = SERVICE_STATISTICS.get(uri);
    }
    return status;
}
```

METHOD_STATISTICS 属性，基于服务 URL + 方法为维度的 RpcStatus 集合。#getStatus(url, methodName) 静态方法，获得 RpcStatus 对象，代码如下：

```
public static RpcStatus getStatus(URL url, String methodName) {
    String uri = url.toIdentityString();
    // 获得方法集合
    ConcurrentMap<String, RpcStatus> map = METHOD_STATISTICS.get(uri);
    // 不存在，创建方法集合
    if (map == null) {
        METHOD_STATISTICS.putIfAbsent(uri, new ConcurrentHashMap<String, RpcStatus>());
        map = METHOD_STATISTICS.get(uri);
    }

    // 获得 RpcStatus 对象
    RpcStatus status = map.get(methodName);
    // 不存在，创建 RpcStatus 对象
    if (status == null) {
        map.putIfAbsent(methodName, new RpcStatus());
        status = map.get(methodName);
    }
    return status;
}
```

========== 对象属性 ==========

次数相关

- active ，调用中的次数。这个属性在 ActiveLimitFilter 中非常关键。
- total failed

时长相关

- totalElapsed failedElapsed
- failedElapsed failedMaxElapsed succeededMaxElapsed

信号量相关

- executesLimit ，服务执行信号量。这个属性在 ExecuteLimitFilter 中非常关键。
- executesPermits ，服务执行信号量大小。

2.2 beginCount

```
/**
```

```
     * 服务调用开始的计数
     *
     * @param url URL 对象
     * @param methodName 方法名
     */
    public static void beginCount(URL url, String methodName) {
        // `SERVICE_STATISTICS` 的计数
        beginCount(getStatus(url));
        // `METHOD_STATISTICS` 的计数
        beginCount(getStatus(url, methodName));
    }
```

静态方法，在其内部，会调用两次 `#beginCount(RpcStatus)` 方法，分别计数。代码如下：

```
    private static void beginCount(RpcStatus status) {
        // 调用中的次数
        status.active.incrementAndGet();
    }
```

2.3 endCount

```
    /**
     * 服务调用结束的计数
     *
     * @param url URL 对象
     * @param elapsed 时长，毫秒
     * @param succeeded 是否成功
     */
    public static void endCount(URL url, String methodName, long elapsed, boolean succeeded) {
        // `SERVICE_STATISTICS` 的计数
        endCount(getStatus(url), elapsed, succeeded);
        // `METHOD_STATISTICS` 的计数
        endCount(getStatus(url, methodName), elapsed, succeeded);
    }
```

静态方法，在其内部，会调用两次 `#endCount(RpcStatus)` 方法，分别计数。代码如下：

```
    private static void endCount(RpcStatus status, long elapsed, boolean succeeded) {
        // 次数计数
        status.active.decrementAndGet();
        status.total.incrementAndGet();
        status.totalElapsed.addAndGet(elapsed);
        // 时长计数
        if (status.maxElapsed.get() < elapsed) {
            status.maxElapsed.set(elapsed);
        }
        if (succeeded) {
            if (status.succeededMaxElapsed.get() < elapsed) {
                status.succeededMaxElapsed.set(elapsed);
            }
        } else {
            status.failed.incrementAndGet(); // 失败次数
            status.failedElapsed.addAndGet(elapsed);
```

```
                if (status.failedMaxElapsed.get() < elapsed) {
                    status.failedMaxElapsed.set(elapsed);
                }
            }
        }
```

2.4 getSemaphore

```
public Semaphore getSemaphore(int maxThreadNum) {
    if(maxThreadNum <= 0) {
        return null;
    }
    // 若信号量不存在，或者信号量大小改变，创建新的信号量
    if (executesLimit == null || executesPermits != maxThreadNum) {
        synchronized (this) {
            if (executesLimit == null || executesPermits != maxThreadNum) {
                executesLimit = new Semaphore(maxThreadNum);
                executesPermits = maxThreadNum;
            }
        }
    }
    // 返回信号量
    return executesLimit;
}
```

对象方法，获得信号量 `executesPermits` 属性。
创建信号量的条件，信号量 `executesPermits` 不存在，或者信号量大小 `executesLimit` 发生改变。
我们会发生比较“神奇”的是，这个方法是直接返回 Semaphore 对象。考虑到有信号量大小改变的需求，但是信号量不支持批量修改大小，那么剩余的一种合适的方式，创建新的信号量对象。因此，这个方法就选择了直接返回 Semaphore 对象。
在 [《Dubbo源代码分析七：使用executes属性的一个问题》] 中，分享的很不错。

3. ActiveLimitFilter

`com.alibaba.dubbo.rpc.filter.ActiveLimitFilter` ，实现 Filter 接口，每服务消费者每服务、每方法的最大可并行调用数限制的过滤器实现类。

```
 1: @Activate(group = Constants.CONSUMER, value = Constants.ACTIVES_KEY)
 2: public class ActiveLimitFilter implements Filter {
 3:
 4:     @Override
 5:     public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
 6:         URL url = invoker.getUrl();
 7:         String methodName = invocation.getMethodName();
 8:         // 获得服务提供者每服务每方法最大可并行执行请求数
 9:         int max = invoker.getUrl().getMethodParameter(methodName, Constants.ACTIVES_KEY, 0);
10:         // 获得 RpcStatus 对象，基于服务 URL + 方法维度
11:         RpcStatus count = RpcStatus.getStatus(invoker.getUrl(), invocation.getMethodName());
12:         if (max > 0) {
13:             // 获得超时值
14:             long timeout = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.TIMEOUT_KEY,
15:             long start = System.currentTimeMillis();
```

```
16:             long remain = timeout; // 剩余可等待时间
17:             int active = count.getActive();
18:             // 超过最大可并行执行请求数，等待
19:             if (active >= max) {
20:                 synchronized (count) { // 通过锁，有且仅有一个在等待。
21:                     // 循环，等待可并行执行请求数
22:                     while ((active = count.getActive()) >= max) {
23:                         // 等待，直到超时，或者被唤醒
24:                         try {
25:                             count.wait(remain);
26:                         } catch (InterruptedException e) {
27:                         }
28:                         // 判断是否没有剩余时长了，抛出 RpcException 异常
29:                         long elapsed = System.currentTimeMillis() - start; // 本地等待时长
30:                         remain = timeout - elapsed;
31:                         if (remain <= 0) {
32:                             throw new RpcException("Waiting concurrent invoke timeout in client-side for service:
33:                                     + invoker.getInterface().getName() + ", method: "
34:                                     + invocation.getMethodName() + ", elapsed: " + elapsed
35:                                     + ", timeout: " + timeout + ". concurrent invokes: " + active
36:                                     + ". max concurrent invoke limit: " + max);
37:                         }
38:                     }
39:                 }
40:             }
41:         }
42:         try {
43:             long begin = System.currentTimeMillis();
44:             // 调用开始的计数
45:             RpcStatus.beginCount(url, methodName);
46:             try {
47:                 // 服务调用
48:                 Result result = invoker.invoke(invocation);
49:                 // 调用结束的计数（成功）
50:                 RpcStatus.endCount(url, methodName, System.currentTimeMillis() - begin, true);
51:                 return result;
52:             } catch (RuntimeException t) {
53:                 // 调用结束的计数（失败）
54:                 RpcStatus.endCount(url, methodName, System.currentTimeMillis() - begin, false);
55:                 throw t;
56:             }
57:         } finally {
58:             // 唤醒等待的相同服务的相同方法的请求
59:             if (max > 0) {
60:                 synchronized (count) {
61:                     count.notify();
62:                 }
63:             }
64:         }
65:     }
66:
67: }
```

ActiveLimitFilter 基于 RpcStatus.active 属性，判断当前正在调用中的服务的方法的次数来判断。因为，需要有等待超时的特性，所以不使用 RpcStatus.semaphore 信号量的方式来实现。
第 9 行：调用 URL#getMethodParameter(methodName, key, defaultValue) 方法，获得服务提供者每服务每方法最大可并行执行请求数。优先 <dubbo: method /> ，其次 <dubbo:reference /> 。
第 11 行：调用 RpcStatus#getStatus(url, methodName) 方法，获得 RpcStatus 对象，基于服务 URL + 方法为维度。

第 14 行：获得超时值。这里有一点需要注意，此处产生的等待时长，不占用调用服务的超时时长。所以，极端情况下的服务超时，约等于 2 * timeout 。
第 19 行：超过最大可并行执行请求数，需要等待。
第 20 行：通过锁定 synchronized ，有且仅有一个在等待。同时，也保证先调用的可以先执行。
第 22 行：循环，等待可并行执行请求数。为什么需要循环呢？极端情况下，恰好有一个新的调用，在【第 61 行】执行的一瞬间，走到了【第 19 行】，“抢”走了正在锁定等待的请求机会。
第 23 至 27 行：等待，直到超时，或者被唤醒【第 61 行】。
第 28 至 37 行：判断若没有剩余时长了，抛出 RpcException 异常。
第 45 行：调用 RpcStaus#beginCount(url, methodName) 方法，调用开始的计数。
第 48 行：调用 Invoker#invoke(invocation) 方法，服务调用。
第 50 行：调用 RpcStaus#endCount(url, methodName, true) 方法，调用开始的计数（成功）。
第 54 行：调用 RpcStaus#endCount(url, methodName, false) 方法，调用开始的计数（失败）。
第 59 至 63 行：唤醒等待的相同服务的相同方法的请求【第 25 行】。

4. ExecuteLimitFilter

com.alibaba.dubbo.rpc.filter.ExecuteLimitFilter ，实现 Filter 接口，服务提供者每服务、每方法的最大可并行执行请求数的过滤器实现类。

```
 1: @Activate(group = Constants.PROVIDER, value = Constants.EXECUTES_KEY)
 2: public class ExecuteLimitFilter implements Filter {
 3:
 4:     @Override
 5:     public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
 6:         URL url = invoker.getUrl();
 7:         String methodName = invocation.getMethodName();
 8:         Semaphore executesLimit = null; // 信号量
 9:         boolean acquireResult = false; // 是否获得信号量
10:         // 获得服务提供者每服务每方法最大可并行执行请求数
11:         int max = url.getMethodParameter(methodName, Constants.EXECUTES_KEY, 0);
12:         if (max > 0) {
13:             // 获得 RpcStatus 对象，基于服务 URL + 方法维度
14:             RpcStatus count = RpcStatus.getStatus(url, invocation.getMethodName());
15:             // 获得信号量。若获得不到，抛出异常。
16: //            if (count.getActive() >= max) {
17:             /**
18:              * http://manzhizhen.iteye.com/blog/2386408
19:              * use semaphore for concurrency control (to limit thread number)
20:              */
21:             executesLimit = count.getSemaphore(max);
22:             if (executesLimit != null && !(acquireResult = executesLimit.tryAcquire())) {
23:                 throw new RpcException("Failed to invoke method " + invocation.getMethodName() + " in provider "
24:             }
25:         }
26:         long begin = System.currentTimeMillis();
27:         boolean isSuccess = true;
28:         // 调用开始的计数
29:         RpcStatus.beginCount(url, methodName);
30:         try {
31:             // 服务调用
32:             return invoker.invoke(invocation);
33:         } catch (Throwable t) {
34:             isSuccess = false; // 标记失败
```

```
35:             if (t instanceof RuntimeException) {
36:                 throw (RuntimeException) t;
37:             } else {
38:                 throw new RpcException("unexpected exception when ExecuteLimitFilter", t);
39:             }
40:         } finally {
41:             // 调用结束的计数（成功）（失败）
42:             RpcStatus.endCount(url, methodName, System.currentTimeMillis() - begin, isSuccess);
43:             // 释放信号量
44:             if (acquireResult) {
45:                 executesLimit.release();
46:             }
47:         }
48:     }
49:
50: }
```

ActiveLimitFilter 基于 RpcStatus.semaphore 信号量属性，判断若超过最大可并行，抛出 RpcException 异常。
第 11 行：调用 URL#getMethodParameter(methodName, key, defaultValue) 方法，获得服务提供者每服务每方法最大可并行执行请求数。优先 <dubbo: method /> ，其次 <dubbo:service /> 。
第 13 行：调用 RpcStatus#getStatus(url, methodName) 方法，获得 RpcStatus 对象，基于服务 URL + 方法为维度。
第 21 至 21 行：调用 RpcStatus#getSemaphore(max) 方法，获得 Semaphore 对象。
第 22 至 24 行：调用 Semaphore#tryAcquire() 方法，若获得不到信号量，抛出 RpcException 异常。
第 29 行：调用 RpcStaus#beginCount(url, methodName) 方法，调用开始的计数。
第 32 行：调用 Invoker#invoke(invocation) 方法，服务调用。
第 34 行：若发生异常，标记 isSuccess = false ，表示调用失败。
第 42 行：调用 RpcStaus#endCount(url, methodName, success) 方法，调用开始的计数（成功）（失败）。
第 43 至 46 行：调用 Semaphore#release() 方法，释放信号量。

666. 彩蛋

周末写博客，美滋滋。

文章目录

回到首页