zuul 网关超时优化 - 1. 解决问题

作者：boolean-dev
原文链接：https://ld246.com/article/1600963322508
来源网站：链滴

1. 概述

前段时间，线上的服务不知道为啥，突然全部的服务都超时，所有的请求经过网关都超时，后来进行路追踪排查，发现有一个服务链接 RDS 数据库，一个查询花费了 20S 的查询时间，导致后续调用该务的应用都超时。然后超时的连接占满了 zuul 的转发池，最终导致了所有经过 gateway 的服务都在待，导致全体服务全部超时。

原因找出来之后，我就在纳闷，为何一个服务超时会导致所有服务都延时，作为一个高可用的网关，uul 的设计应该不会这么差吧，并且，当时系统的 QPS 也不是很高，所以，必须找出这次超时的问所在，于是，我就开始了此次网关超时的排查和优化！

下图是出现问题是的相关监控

因为公司的服务是 java 和 node 应用都存在，所以使用的 zuul 是简单路由转发，未使用 服务注册心之类的，所以，熔断器和

2. 问题分析

首先，因为网关的相关配置都是之前填的，有些参数不是特别的清楚，下面的配置文件是我当时系统相关配置。

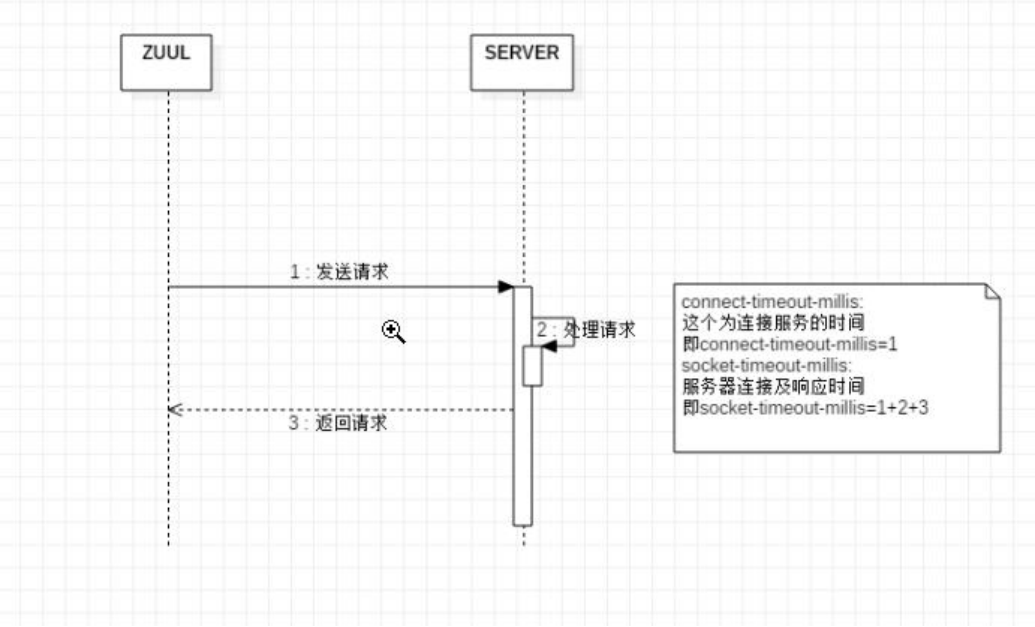
```
zuul:
  host:
    connect-timeout-millis: 30000
    socket-timeout-millis: 60000

ribbon:
  ConnectTimeout: 300000
  ReadTimeout: 60000
  eureka:
    enabled: false

hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 70000
```

2.1 zuul 参数解释

connect-timeout-millis

此参数为 zuul 网关连接服务的时间，单位为毫秒，我这里配置的是 30S,如果 30S 之内未连接到待转的下一服务，则转发将报错，此请求也就将结束。这段时间为下图的第 1 步的时间

如下图所示：



socket-timeout-millis

zuul 网关连接到服务并且服务返回结果这一部分时间，单位为毫秒，我这里配置的是 60S，如果 60S

之内下一服务还没有返回， gateway 将报转发超时。

此时间为上图的 1+2+3 三部分时间

ribbon.ConnectTimeout

此为 ribbon 转发的连接时间，如果 zuul 使用的服务调用，则将采用此时间

ribbon.ReadTimeout

此为 ribbon 转发到返回的时间，如果 zuul 使用的是服务调用，则将采用此时间

超时时间采用哪个?

以上四个都是 zuul 的超时时间，但是问题来了，四个时间，到底采用哪两个呢?

我通过阅读 zuul 的相关文档，了解到，如果 zuul 使用的服务发现，则将会使用 rebbon进行负载均，即 ribbon.ConnectTimeout 和 ribbon.ReadTimeout。 如果 zuul 使用的是简单路由（通过配置 u l 进行路由转发），则将采用 socket-timeout-millis 和 connect-timeout-millis。

If you want to configure the socket timeouts and read timeouts for requests proxied through uul, you have two options, based on your configuration:

- If Zuul uses service discovery, you need to configure these timeouts with the ribbon.Rea Timeout and ribbon.SocketTimeout Ribbon properties.
- If you have configured Zuul routes by specifying URLs, you need to use zuul.host.connec -timeout-millis and zuul.host.socket-timeout-millis.

2.2 问题分析

因为我的服务未使用服务注册中心，所以，很明显， 配置的 ribbon 并未生效，zuul 超时时间使用的 ocket-timeout-millis 和 connect-timeout-millis。

但是问题又来了，我 zuul 的超时时间为 60S，为何我的服务相应时间平均达到了 3 分钟，远远超过我设置的 60S，所以肯定是 zuul 还出现了相应的问题。

后续通过了解到 zuul 源码和架构，明白 zuul 是 NIO 架构，即一个请求进来，经过 zuul 拦截器 Filte ，最终交给 HttpClient 进行请求，zuul 为了高性能，使用 HttpClient 连接池。

获取连接源码

下面是 zuul 获取 HttpClient 连接池的代码

AbstractConnPool.getPoolEntryBlocking,看这个名字就知道。这是一个阻塞获取池资源的方法

```
    private E getPoolEntryBlocking(
            final T route, final Object state,
            final long timeout, final TimeUnit timeUnit,
            final Future<E> future) throws IOException, InterruptedException, ExecutionException,
TimeoutException {
```

```
Date deadline = null;
if (timeout > 0) {
    deadline = new Date (System.currentTimeMillis() + timeUnit.toMillis(timeout));
}
this.lock.lock();
try {
    final RouteSpecificPool<T, C, E> pool = getPool(route);
    E entry;
    for (;;) {
        Asserts.check(!this.isShutDown, "Connection pool shut down");
        if (future.isCancelled()) {
            throw new ExecutionException(operationAborted());
        }
        for (;;) {
            entry = pool.getFree(state);
            if (entry == null) {
                break;
            }
            if (entry.isExpired(System.currentTimeMillis())) {
                entry.close();
            }
            if (entry.isClosed()) {
                this.available.remove(entry);
                pool.free(entry, false);
            } else {
                break;
            }
        }
        if (entry != null) {
            this.available.remove(entry);
            this.leased.add(entry);
            onReuse(entry);
            return entry;
        }

        // New connection is needed
        final int maxPerRoute = getMax(route);
        // Shrink the pool prior to allocating a new connection
        final int excess = Math.max(0, pool.getAllocatedCount() + 1 - maxPerRoute);
        if (excess > 0) {
            for (int i = 0; i < excess; i++) {
                final E lastUsed = pool.getLastUsed();
                if (lastUsed == null) {
                    break;
                }
                lastUsed.close();
                this.available.remove(lastUsed);
                pool.remove(lastUsed);
            }
        }

        if (pool.getAllocatedCount() < maxPerRoute) {
            final int totalUsed = this.leased.size();
            final int freeCapacity = Math.max(this.maxTotal - totalUsed, 0);
```

```
                if (freeCapacity > 0) {
                    final int totalAvailable = this.available.size();
                    if (totalAvailable > freeCapacity - 1) {
                        if (!this.available.isEmpty()) {
                            final E lastUsed = this.available.removeLast();
                            lastUsed.close();
                            final RouteSpecificPool<T, C, E> otherpool = getPool(lastUsed.getRoute())

                            otherpool.remove(lastUsed);
                        }
                    }
                    final C conn = this.connFactory.create(route);
                    entry = pool.add(conn);
                    this.leased.add(entry);
                    return entry;
                }
            }

            boolean success = false;
            try {
                pool.queue(future);
                this.pending.add(future);
                if (deadline != null) {
                    success = this.condition.awaitUntil(deadline);
                } else {
                    this.condition.await();
                    success = true;
                }
                if (future.isCancelled()) {
                    throw new ExecutionException(operationAborted());
                }
            } finally {
                // In case of 'success', we were woken up by the
                // connection pool and should now have a connection
                // waiting for us, or else we're shutting down.
                // Just continue in the loop, both cases are checked.
                pool.unqueue(future);
                this.pending.remove(future);
            }
            // check for spurious wakeup vs. timeout
            if (!success && (deadline != null && deadline.getTime() <= System.currentTimeMilli
())) {
                break;
            }
        }
        throw new TimeoutException("Timeout waiting for connection");
    } finally {
        this.lock.unlock();
    }
}
```

1. 代码已建立有一个deadline， 然后判断timeout， 这个timeout要注意。如果大于零才会赋值deadl
ne， 如果为0 则不会赋值deadline 也就是说deadline始终为null

```
Date deadline = null;
if (timeout > 0) {
    //如果超时时间有效，则设定deadline
    deadline = new Date (System.currentTimeMillis() + tunit.toMillis(timeout));
}
```

2. 进入锁代码。pool.getFree 获取池资源。如果获取到了，并且Connect的检验并没有被关闭，则接return entry

```
Asserts.check(!this.isShutDown, "Connection pool shut down");
for (;;) {
    //获取池资源
    entry = pool.getFree(state);
    if (entry == null) {
        break;
    }
    //校验超时
    if (entry.isExpired(System.currentTimeMillis())) {
        entry.close();
    }
    if (entry.isClosed()) {
        this.available.remove(entry);
        pool.free(entry, false);
    } else {
        break;
    }
}
if (entry != null) {
    this.available.remove(entry);
    this.leased.add(entry);
    onReuse(entry);
    return entry;
}
```

3. 如果没有获取到 进行接下来的代码。

4. 判断是否达到了host配置的最大池数量，是否需要增加, 如果需要增加，则会在增加新连接之前缩池，然后再分配返回entry

```
// New connection is needed  获取是否需要创建新的连接
final int maxPerRoute = getMax(route);
// Shrink the pool prior to allocating a new connection
final int excess = Math.max(0, pool.getAllocatedCount() + 1 - maxPerRoute);
if (excess > 0) {
    for (int i = 0; i < excess; i++) {
        final E lastUsed = pool.getLastUsed();
        if (lastUsed == null) {
            break;
        }
        lastUsed.close();
        this.available.remove(lastUsed);
        pool.remove(lastUsed);
    }
}
```

```
if (pool.getAllocatedCount() < maxPerRoute) {
    final int totalUsed = this.leased.size();
    final int freeCapacity = Math.max(this.maxTotal - totalUsed, 0);
    if (freeCapacity > 0) {
        final int totalAvailable = this.available.size();
        if (totalAvailable > freeCapacity - 1) {
            if (!this.available.isEmpty()) {
                final E lastUsed = this.available.removeLast();
                lastUsed.close();
                final RouteSpecificPool<T, C, E> otherpool = getPool(lastUsed.getRoute())

                otherpool.remove(lastUsed);
            }
        }
        final C conn = this.connFactory.create(route);
        entry = pool.add(conn);
        this.leased.add(entry);
        return entry;
    }
}
```

6. 如果并不是上面的情况，实际情况就是池子被用光了，而且还达到了最大。就不能从池子中获取资了。只能等了……

7. 等待的时候会判断deadline， 如果deadline不为null 就会await一个时间。如果为null，那么等待会无限等待，直到有资源。

```
boolean success = false;
try {
    if (future.isCancelled()) {
        throw new InterruptedException("Operation interrupted");
    }
    pool.queue(future);
    this.pending.add(future);
    //判断deadline是否有效
    if (deadline != null) {
        //如果有效就等待至deadline
        success = this.condition.awaitUntil(deadline);
    } else {
       //如果无效就一直等待，没有超时时间
        this.condition.await();
        success = true;
    }
    if (future.isCancelled()) {
        throw new InterruptedException("Operation interrupted");
    }
} finally {
    // In case of 'success', we were woken up by the
    // connection pool and should now have a connection
    // waiting for us, or else we're shutting down.
    // Just continue in the loop, both cases are checked.
    pool.unqueue(future);
    this.pending.remove(future);
}
```

创建连接池源码

问题通过以上的源码就发现了，关键问题是线程池的等待时间，设置一个连接的等待时间即可解决，得线程不会一直等待 HttpCllient 连接，我找到相应的创建 CloseableHttpClient 卫士，位于 SimpleostRoutingFilter#newClient#newClient()，源码如下

```
protected CloseableHttpClient newClient() {
    final RequestConfig requestConfig = RequestConfig.custom()
            .setConnectionRequestTimeout(
            // 设置超时链接时间
                    this.hostProperties.getConnectionRequestTimeoutMillis())
            .setSocketTimeout(this.hostProperties.getSocketTimeoutMillis())
            .setConnectTimeout(this.hostProperties.getConnectTimeoutMillis())
            .setCookieSpec(CookieSpecs.IGNORE_COOKIES).build();
    return httpClientFactory.createBuilder().setDefaultRequestConfig(requestConfig)
            .setConnectionManager(this.connectionManager).disableRedirectHandling()
            .build();
}
```

此处可看到，主要是从 this.hostProperties.getConnectionRequestTimeoutMillis()，拿到超时时，最终我找到了 Springboot 配置 connectionRequestTimeoutMillis 位置，即 ZuulProperties.Hos#connectionRequestTimeoutMillis，代码如下

```
/**
 * Represents a host.
 */
public static class Host {

    /**
     * The maximum number of total connections the proxy can hold open to backends.
     */
    private int maxTotalConnections = 200;

    /**
     * The maximum number of connections that can be used by a single route.
     */
    private int maxPerRouteConnections = 20;

    /**
     * The socket timeout in millis. Defaults to 10000.
     */
    private int socketTimeoutMillis = 10000;

    /**
     * The connection timeout in millis. Defaults to 2000.
     */
    private int connectTimeoutMillis = 2000;

    /**
     * The timeout in milliseconds used when requesting a connection from the
     * connection manager. Defaults to -1, undefined use the system default.
     * 此处时间为 -1，即永久等待
     */
    private int connectionRequestTimeoutMillis = -1;
```

```
    /**
     * The lifetime for the connection pool.
     */
    private long timeToLive = -1;

    /**
     * The time unit for timeToLive.
     */
    private TimeUnit timeUnit = TimeUnit.MILLISECONDS;

    public Host() {
    }

    // get set and toString

}
```

通过源码可看到，zuul 有如下的相关配置：

- maxTotalConnections：HttpClient 总连接数，默认值为200
- maxPerRouteConnections：HttpClient 单个服务（即服务发现中的每个服务）连接数，默认为 20
- socketTimeoutMillis：连接服务时间，单位为毫秒，默认为10秒
- connectTimeoutMillis：服务返回时间，单位为毫秒，默认时间为20秒
- connectionRequestTimeoutMillis：连接 HttpClient 等待时间，默认为-1，即永久！

下面来对为何超时进行一个复现：

假设 100 个请求进来，HttpClient 连接池大小为 20，其中 20 个请求一直在等待远程服务返回，其余 80 个请求一直在等待连接池的空闲连接，所以连接一直在等待，最终导致其它服务无法进入，最终致所有服务都瘫痪了。

因为我的 gateway 未使用路由发现，所以，微服务中的熔断器和负载均衡，均使用不上。所以只能用如下方法：

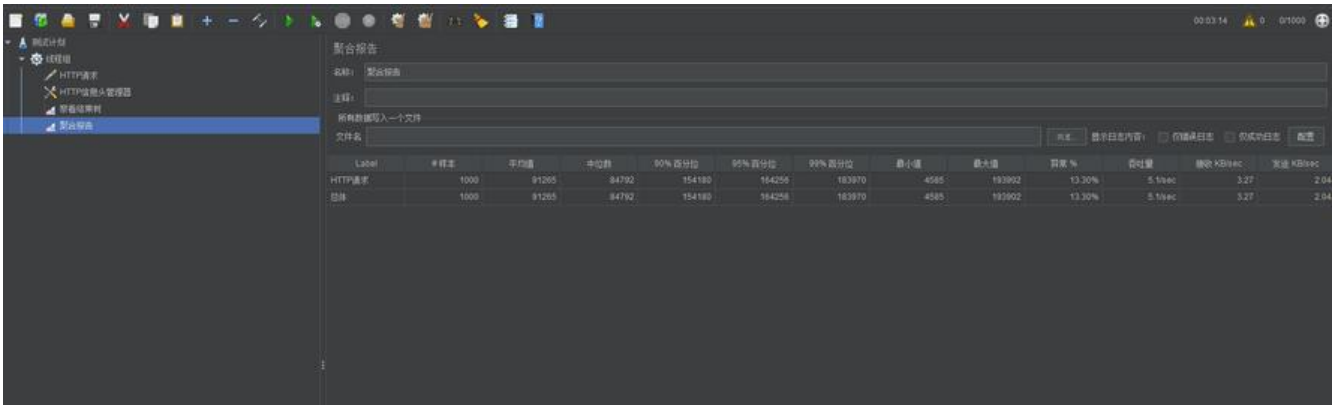
- 设置 HttpClient 连接时间，即 connectionRequestTimeoutMillis 设置为 10S
- 增大 HttpClient 连接池大小，使得有足够多的连接数，来增大并发量，即设置 maxTotalConnections 和 maxPerRouteConnections，这里我设置成了 500 和 250

下面是我调优后的参数：

```
zuul:
  host:
    connect-timeout-millis: 30000
    socket-timeout-millis: 60000
    max-total-connections: 500
    max-per-route-connections: 250
    connection-request-timeout-millis: 10000
```

3. 后续

虽然增大了 HttpClient 连接池大小，修改了连接 HttpClient 的时间，但是进行压测时，依旧会出现些请求，时间超过了 60S，我们以上的设置为 60S 再加上抢占连接池，总时间也不过 70S，如果超过0S，则会报抢占 HttpClient 连接池异常。但是我压测后的结果，并不如此，以下是我压测的结果



以上压测显示，95% 请求时间为 164 S，远远超过了我们预测的 70S，并且压测的 QPS 很低，才 5.，所以 zuul 的效率还不是很高，还待优化！

预知后事如何，请期待接下来的博客