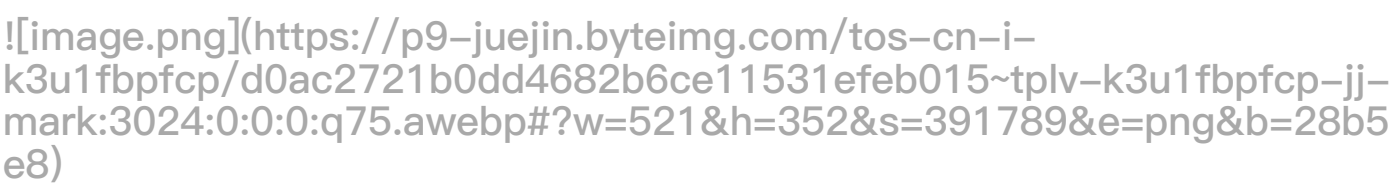
从‘海量请求’到‘丝滑响应’：巧解海量httpclient请求优化难题~

大家好，我是石头~

今天接到一个需求，其关键功能需要依托一个外部HTTP接口来实现，而且这项功能还是整个业务中请求频率最高的。

开完需求会后，我就开始头疼了。众所周知，在高并发场景下进行HTTP请求，会降低整个服务的性能，怎样进行性能优化，就成为了实现本次需求的核心了。



1、大量HTTP请求的弊端

在进行性能优化之前，我们先来了解下为什么大量HTTP请求，会造成服务性能下降。

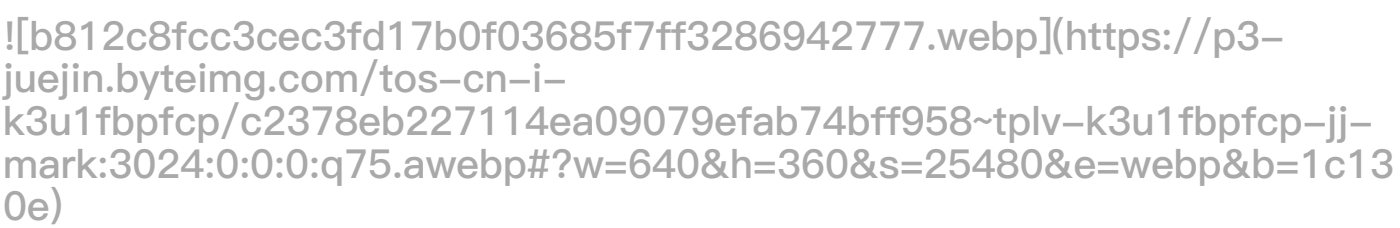
这个我们可以从以下几方面来看：

1. `网络资源竞争：`服务端在向其他服务发起HTTP请求时，会产生网络带宽的竞争。特别是当请求量很大时，大量的数据包在网络中穿梭，容易导致网络带宽饱和，增加延迟，甚至产生网络拥塞，使得请求响应时间延长。
2. `系统资源消耗：`服务端在处理每个HTTP请求时，都需要占用CPU、内存、文件句柄等系统资源。特别是在并发请求较高时，服务端必须创建和维护多个连接，处理请求和解析响应，这些都会消耗大量系统资源。一旦资源耗尽，新进的请求将无法得到及时处理，严重影响服务性能。
3. `高并发下的连接管理：`对于每次HTTP请求，服务端通常需要创建一个新的TCP连接。如果连接创建和销毁过于频繁，会大大增加系统开销。如果不采取连接池等优化手段，服务端可能会因连接管理负担过重而降低性能。
4. `请求排队与响应时间：`服务端发起的HTTP请求也需要排队等待对方服务器的响应，尤其是在目标服务本身负载较大或者网络条件不佳的情况下，响应时间的增长将进一步拖慢服务端的处理速度，最终可能形成连锁反应，导致整个

系统性能下降。

综上所述，大量HTTP请求就像高峰期的交通堵塞，不仅挤占了网络通道，也给服务器处理能力带来巨大挑战。

那么，针对这些问题，我们要怎样进行优化？



2、大量HTTP请求的优化策略

由于团队内部采用的是httpclient，那接下来，我们就以httpclient为例进行优化。

策略一：连接池管理

如同高效有序的物流仓库，高效的httpclient请求离不开合理的连接池管理。通过设置合适的最大连接数、超时时间以及重试策略，我们可以避免频繁创建和关闭连接带来的性能损耗，同时也能应对突发的大流量请求。

```
    PoolingHttpClientConnectionManager connMgr = new PoolingHttpClientConnectionManager();
    connMgr.setMaxTotal(200); // 设置最大连接数
    connMgr.setDefaultMaxPerRoute(20); // 设置每个路由基础的默认最大连接数

    RequestConfig requestConfig = RequestConfig.custom()
            .setSocketTimeout(5000) // 设置SO_TIMEOUT，即从连接成功建立到读取到数据之间的等待时间
            .setConnectTimeout(3000) // 设置连接超时时间
            .setConnectionRequestTimeout(1000) // 设置从连接池获取连接的等待时间
```

```
            .build();

    CloseableHttpClient httpClient = HttpClients.custom()
            .setConnectionManager(connMgr)
            .setDefaultRequestConfig(requestConfig)
            .build();
```

策略二：异步化处理与线程池

面对大量的网络请求，同步处理方式可能会导致线程阻塞，影响整体性能。采用异步处理机制结合线程池技术，能够将请求放入队列并分配给空闲线程执行，从而大大提高系统的并发处理能力，降低响应时间。

```
    // 创建线程池
    ThreadPoolExecutor executor = new ThreadPoolExecutor(
        10, // 核心线程数
        20, // 最大线程数
        60, // 空闲线程存活时间（单位秒）
        TimeUnit.SECONDS,
        new LinkedBlockingQueue<>(100)); // 工作队列

    CloseableHttpClient httpClient = ...; // 初始化 HttpClient

    List<String> urls = ...; // 要请求的URL列表

    for (String url : urls) {
        final String finalUrl = url;
        Runnable task = () -> {
            try (CloseableHttpResponse response = httpClient.execute(new
HttpGet(finalUrl))) {
                // 处理响应逻辑
            } catch (IOException e) {
                // 处理异常
            }
        };

        executor.execute(task);
    }

    // 在所有任务完成后关闭线程池
    executor.shutdown();
```

```
    // 可以选择等待所有任务完成
    executor.awaitTermination(Long.MAX_VALUE,
TimeUnit.NANOSECONDS);

    // 或者在特定条件下停止并强制关闭线程池
    if (!executor.isTerminated()) {
        executor.shutdownNow();
    }

    // 别忘了在最后关闭HttpClient资源
    httpClient.close();

```

策略三：请求合并与批量处理

对于类似的或依赖关系不强的请求，可以考虑合并为一个请求或者批量处理，减少网络交互次数，显著提升效率。例如，利用HTTP/2的多路复用特性，或者对数据进行归类整合后一次性获取。

```
    List<String> userIds = ...; // 用户ID列表

    HttpUriRequest[] requests = userIds.stream()
            .map(userId -> new HttpGet("http://example.com/api/user/" +
userId))
            .toArray(HttpUriRequest[]::new);

    HttpRequestRetryHandler retryHandler = ...; // 自定义重试处理器
    HttpClient httpClient =
HttpClients.custom().setRetryHandler(retryHandler).build();

    List<Future<HttpResponse>> futures = new ArrayList<>();
    for (HttpUriRequest request : requests) {
        futures.add(httpClient.execute(request, new
FutureCallback<HttpResponse>() {...}));
    }

    // 等待所有请求完成并处理结果
    for (Future<HttpResponse> future : futures) {
        HttpResponse response = future.get();
        // 处理每个用户的响应信息
    }
```

```

**策略四：缓存优化**
---

对于部分不变或短期内变化不大的数据，可以通过本地缓存或分布式缓存（如Redis）来避免重复请求，既节省了带宽，也减轻了服务器压力。

```
    LoadingCache<String, String> cache = CacheBuilder.newBuilder()
            .maximumSize(1000) // 设置缓存的最大容量
            .expireAfterWrite(1, TimeUnit.HOURS) // 数据写入一小时后过期
            .build(new CacheLoader<String, String>() {
                @Override
                public String load(String key) throws Exception {
                    // 如果缓存中没有该key，则通过httpclient请求获取数据并返回
                    CloseableHttpResponse response = httpClient.execute(new HttpGet("http://example.com/api/data/" + key));
                    return EntityUtils.toString(response.getEntity());
                }
            });

    // 获取数据，如果缓存中有则直接返回，否则发起网络请求并将结果存入缓存
    String data = cache.get("someKey");
```

3、结语
====

优化之路永无止境，每一个环节都可能存在更深层次的改进空间，希望上面的内容在当你遇到类似问题时，对你有所帮助~

\*\*MORE | 更多精彩文章\*\*

* 面试官：说说单点登录都是怎么实现的?
* 面试不慌张：一文读懂FactoryBean的实现原理
* JWT vs Session：到底哪个才是你的菜?
* JWT重放漏洞如何攻防？你的系统安全吗?
* JWT：你真的了解它吗?
```