

## 05 | HTTP调用：你考虑到超时、重试、并发了吗？

2020-03-19 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)



讲述：王少泽

时长 21:09 大小 19.37M



你好，我是朱晔。今天，我们一起聊聊进行 HTTP 调用需要注意的超时、重试、并发等问题。

与执行本地方法不同，进行 HTTP 调用本质上是通过 HTTP 协议进行一次网络请求。网络请求必然有超时的可能性，因此我们必须考虑到这三点：

首先，框架设置的默认超时是否合理；

其次，考虑到网络的不稳定，超时后的请求重试是一个不错的选择，但需要考虑服务接口的幂等性设计是否允许我们重试；



最后，需要考虑框架是否会像浏览器那样限制并发连接数，以免在服务并发很大的情况下，HTTP 调用的并发数限制成为瓶颈。

Spring Cloud 是 Java 微服务架构的代表性框架。如果使用 Spring Cloud 进行微服务开发，就会使用 Feign 进行声明式的服务调用。如果不使用 Spring Cloud，而直接使用 Spring Boot 进行微服务开发的话，可能会直接使用 Java 中最常用的 HTTP 客户端 Apache HttpClient 进行服务调用。

接下来，我们就看看使用 Feign 和 Apache HttpClient 进行 HTTP 接口调用时，可能会遇到的超时、重试和并发方面的坑。

## 配置连接超时和读取超时参数的学问

对于 HTTP 调用，虽然应用层走的是 HTTP 协议，但网络层面始终是 TCP/IP 协议。TCP/IP 是面向连接的协议，在传输数据之前需要建立连接。几乎所有的网络框架都会提供这么两个超时参数：

连接超时参数 ConnectTimeout，让用户配置建连阶段的最长等待时间；

读取超时参数 ReadTimeout，用来控制从 Socket 上读取数据的最长等待时间。

这两个参数看似是网络层偏底层的配置参数，不足以引起开发同学的重视。但，正确理解和配置这两个参数，对业务应用特别重要，毕竟超时不是单方面的事情，需要客户端和服务端对超时有一致的估计，协同配合方能平衡吞吐量和错误率。

### 连接超时参数和连接超时的误区有这么两个：

**连接超时配置得特别长，比如 60 秒。**一般来说，TCP 三次握手建立连接需要的时间非常短，通常在毫秒级最多到秒级，不可能需要十几秒甚至几十秒。如果很久都无法建连，很可能是网络或防火墙配置的问题。这种情况下，如果几秒连接不上，那么可能永远也连接不上。因此，设置特别长的连接超时意义不大，将其配置得短一些（比如 1~5 秒）即可。如果是纯内网调用的话，这个参数可以设置得更短，在下游服务离线无法连接的时候，可以快速失败。


**排查连接超时问题，却没理清连的是哪里。**通常情况下，我们的服务会有多个节点，如果别的客户端通过客户端负载均衡技术来连接服务端，那么客户端和服务端会直接建立

连接，此时出现连接超时大概率是服务端的问题；而如果服务端通过类似 Nginx 的反向代理来负载均衡，客户端连接的其实是 Nginx，而不是服务端，此时出现连接超时应该排查 Nginx。

**读取超时参数和读取超时则会有更多的误区，我将其归纳为如下三个。**

**第一个误区：**认为出现了读取超时，服务端的执行就会中断。

我们来简单测试下。定义一个 client 接口，内部通过 HttpClient 调用服务端接口 server，客户端读取超时 2 秒，服务端接口执行耗时 5 秒。

 复制代码

```
1  @RestController
2  @RequestMapping("clientreadtimeout")
3  @Slf4j
4  public class ClientReadTimeoutController {
5      private String getResponse(String url, int connectTimeout, int readTimeout) {
6          return Request.Get("http://localhost:45678/clientreadtimeout" + url)
7              .connectTimeout(connectTimeout)
8              .socketTimeout(readTimeout)
9              .execute()
10             .returnContent()
11             .asString();
12     }
13
14     @GetMapping("client")
15     public String client() throws IOException {
16         log.info("client1 called");
17         //服务端5s超时，客户端读取超时2秒
18         return getResponse("/server?timeout=5000", 1000, 2000);
19     }
20
21     @GetMapping("server")
22     public void server(@RequestParam("timeout") int timeout) throws InterruptedException {
23         log.info("server called");
24         TimeUnit.MILLISECONDS.sleep(timeout);
25         log.info("Done");
26     }
27 }
```

调用 client 接口后，从日志中可以看到，客户端 2 秒后出现了 SocketTimeoutException，原因是读取超时，服务端却丝毫没受影响在 3 秒后执行完成。

```
1 [11:35:11.943] [http-nio-45678-exec-1] [INFO ] [.t.c.c.d.ClientReadTimeoutCont
2 [11:35:12.032] [http-nio-45678-exec-2] [INFO ] [.t.c.c.d.ClientReadTimeoutCont
3 [11:35:14.042] [http-nio-45678-exec-1] [ERROR] [.a.c.c.C.[.[./].[dispatcherSe
4 java.net.SocketTimeoutException: Read timed out
5     at java.net.SocketInputStream.socketRead0(Native Method)
6     ...
7 [11:35:17.036] [http-nio-45678-exec-2] [INFO ] [.t.c.c.d.ClientReadTimeoutCont
```

我们知道，类似 Tomcat 的 Web 服务器都是把服务端请求提交到线程池处理的，只要服务端收到了请求，网络层面的超时和断开便不会影响服务端的执行。因此，出现读取超时不能随意假设服务端的处理情况，需要根据业务状态考虑如何进行后续处理。

**第二个误区：**认为读取超时只是 Socket 网络层面的概念，是数据传输的最长耗时，故将其配置得非常短，比如 100 毫秒。

其实，发生了读取超时，网络层面无法区分是服务端没有把数据返回给客户端，还是数据在网络上耗时较久或丢包。

但，因为 TCP 是先建立连接后传输数据，对于网络情况不是特别糟糕的服务调用，通常可以认为出现连接超时是网络问题或服务不在线，而出现读取超时是服务处理超时。确切地说，读取超时指的是，向 Socket 写入数据后，我们等到 Socket 返回数据的超时时间，其中包含的时间或者说绝大部分的时间，是服务端处理业务逻辑的时间。

**第三个误区：**认为超时时间越长任务接口成功率就越高，将读取超时参数配置得太长。

进行 HTTP 请求一般是需要获得结果的，属于同步调用。如果超时时间很长，在等待服务端返回数据的同时，客户端线程（通常是 Tomcat 线程）也在等待，当下游服务出现大量超时的时候，程序可能也会受到拖累创建大量线程，最终崩溃。

对定时任务或异步任务来说，读取超时配置得长些问题不大。但面向用户响应的请求或是微服务短平快的同步接口调用，并发量一般较大，我们应该设置一个较短的读取超时时间，以防止被下游服务拖慢，通常不会设置超过 30 秒的读取超时。

你可能会说，如果把读取超时设置为 2 秒，服务端接口需要 3 秒，岂不是永远都拿不到执行结果了？的确是这样，因此设置读取超时一定要根据实际情况，过长可能会让下游抖动影


响到自己，过短又可能影响成功率。甚至，有些时候我们还要根据下游服务的 SLA，为不同的服务端接口设置不同的客户端读取超时。

## Feign 和 Ribbon 配合使用，你知道怎么配置超时吗？

刚才我强调了根据自己的需求配置连接超时和读取超时的重要性，你是否尝试过为 Spring Cloud 的 Feign 配置超时参数呢，有没有被网上的各种资料绕晕呢？

在我看来，为 Feign 配置超时参数的复杂之处在于，Feign 自己有两个超时参数，它使用的负载均衡组件 Ribbon 本身还有相关配置。那么，这些配置的优先级是怎样的，又有哪些坑呢？接下来，我们做一些实验吧。

为测试服务端的超时，假设有这么一个服务端接口，什么都不干只休眠 10 分钟：

 复制代码


```
1 @PostMapping("/server")
2 public void server() throws InterruptedException {
3     TimeUnit.MINUTES.sleep(10);
4 }
```

首先，定义一个 Feign 来调用这个接口：

 复制代码

```
1 @FeignClient(name = "clientsdk")
2 public interface Client {
3     @PostMapping("/feignandribbon/server")
4     void server();
5 }
```

然后，通过 Feign Client 进行接口调用：

 复制代码

```
1 @GetMapping("client")
2 public void timeout() {
3     long begin=System.currentTimeMillis();
4     try{
5         client.server();
6     }catch (Exception ex){
```

```
7         log.warn("执行耗时: {}ms 错误: {}", System.currentTimeMillis() - begin, e
8     }
9 }
```

在配置文件仅指定服务端地址的情况下：

```
1 clientsdk.ribbon.listOfServers=localhost:45678
```

 复制代码

得到如下输出：

```
1 [15:40:16.094] [http-nio-45678-exec-3] [WARN ] [o.g.t.c.h.f.FeignAndRibbonCont
```

 复制代码

从这个输出中，我们可以得到**结论一，默认情况下 Feign 的读取超时是 1 秒，如此短的读取超时算是坑点一。**

我们来分析一下源码。打开 RibbonClientConfiguration 类后，会看到 DefaultClientConfigImpl 被创建出来之后，ReadTimeout 和 ConnectTimeout 被设置为 1s：

```
1 /**
2  * Ribbon client default connect timeout.
3  */
4 public static final int DEFAULT_CONNECT_TIMEOUT = 1000;
5
6 /**
7  * Ribbon client default read timeout.
8  */
9 public static final int DEFAULT_READ_TIMEOUT = 1000;
10
11 @Bean
12 @ConditionalOnMissingBean
13 public IClientConfig ribbonClientConfig() {
14     DefaultClientConfigImpl config = new DefaultClientConfigImpl();
15     config.loadProperties(this.name);
16     config.set(CommonClientConfigKey.ConnectTimeout, DEFAULT_CONNECT_TIMEOUT);
17     config.set(CommonClientConfigKey.ReadTimeout, DEFAULT_READ_TIMEOUT);
18 }
```

 复制代码

```
18     config.set(CommonClientConfigKey.GZipPayload, DEFAULT_GZIP_PAYLOAD);
19     return config;
20 }
```

如果要修改 Feign 客户端默认的两个全局超时时间，你可以设置 `feign.client.config.default.readTimeout` 和 `feign.client.config.default.connectTimeout` 参数：

```
1 feign.client.config.default.readTimeout=3000
2 feign.client.config.default.connectTimeout=3000
```

 复制代码


修改配置后重试，得到如下日志：

```
1 [15:43:39.955] [http-nio-45678-exec-3] [WARN ] [o.g.t.c.h.f.FeignAndRibbonCont
```

 复制代码

可见，3 秒读取超时生效了。注意：这里有一个大坑，如果你希望只修改读取超时，可能会只配置这么一行：

```
1 feign.client.config.default.readTimeout=3000
```

 复制代码

测试一下你就会发现，这样的配置是无法生效的！

**结论二，也是坑点二，如果要配置 Feign 的读取超时，就必须同时配置连接超时，才能生效。**

打开 `FeignClientFactoryBean` 可以看到，只有同时设置 `ConnectTimeout` 和 `ReadTimeout`，`Request.Options` 才会被覆盖：

```
1 if (config.getConnectTimeout() != null && config.getReadTimeout() != null) {
2     builder.options(new Request.Options(config.getConnectTimeout(),
```

 复制代码



```
3         config.getReadTimeout()));  
4     }
```


更进一步，如果你希望针对单独的 Feign Client 设置超时时间，可以把 default 替换为 Client 的 name：

```
1 feign.client.config.default.readTimeout=3000  
2 feign.client.config.default.connectTimeout=3000  
3 feign.client.config.clientsdk.readTimeout=2000  
4 feign.client.config.clientsdk.connectTimeout=2000
```

 复制代码


可以得出**结论三**，单独的超时可以覆盖全局超时，这符合预期，不算坑：

```
1 [15:45:51.708] [http-nio-45678-exec-3] [WARN ] [o.g.t.c.h.f.FeignAndRibbonCont
```

 复制代码


**结论四**，除了可以配置 Feign，也可以配置 Ribbon 组件的参数来修改两个超时时间。这里的坑点三是，参数首字母要大写，和 Feign 的配置不同。

```
1 ribbon.ReadTimeout=4000  
2 ribbon.ConnectTimeout=4000
```

 复制代码

可以通过日志证明参数生效：

```
1 [15:55:18.019] [http-nio-45678-exec-3] [WARN ] [o.g.t.c.h.f.FeignAndRibbonCont
```

 复制代码

最后，我们来看看同时配置 Feign 和 Ribbon 的参数，最终谁会生效？如下代码的参数配置：

```
1 clientsdk.ribbon.listOfServers=localhost:45678
```

 复制代码



```
2 feign.client.config.default.readTimeout=3000
3 feign.client.config.default.connectTimeout=3000
4 ribbon.ReadTimeout=4000
5 ribbon.ConnectTimeout=4000
```


日志输出证明，最终生效的是 Feign 的超时：

 复制代码

```
1 [16:01:19.972] [http-nio-45678-exec-3] [WARN ] [o.g.t.c.h.f.FeignAndRibbonCont
```


**结论五，同时配置 Feign 和 Ribbon 的超时，以 Feign 为准。**这有点反直觉，因为 Ribbon 更底层所以你会觉得后者的配置会生效，但其实不是这样的。

在 LoadBalancerFeignClient 源码中可以看到，如果 Request.Options 不是默认值，就会创建一个 FeignOptionsClientConfig 代替原来 Ribbon 的 DefaultClientConfigImpl，导致 Ribbon 的配置被 Feign 覆盖：

 复制代码

```
1 IClientConfig getClientConfig(Request.Options options, String clientName) {
2     IClientConfig requestConfig;
3     if (options == DEFAULT_OPTIONS) {
4         requestConfig = this.clientFactory.getClientConfig(clientName);
5     }
6     else {
7         requestConfig = new FeignOptionsClientConfig(options);
8     }
9     return requestConfig;
10 }
```

但如果这么配置最终生效的还是 Ribbon 的超时（4 秒），这容易让人产生 Ribbon 覆盖了 Feign 的错觉，其实这还是因为坑二所致，单独配置 Feign 的读取超时并不能生效：

 复制代码

```
1 clientsdk.ribbon.listOfServers=localhost:45678
2 feign.client.config.default.readTimeout=3000
3 feign.client.config.clientsdk.readTimeout=2000
4 ribbon.ReadTimeout=4000
```

## 你是否知道 Ribbon 会自动重试请求呢？

一些 HTTP 客户端往往会内置一些重试策略，其初衷是好的，毕竟因为网络问题导致丢包虽然频繁但持续时间短，往往重试下第二次就能成功，但一定要小心这种自作主张是否符合我们的预期。


之前遇到过一个短信重复发送的问题，但短信服务的调用方用户服务，反复确认代码里没有重试逻辑。那问题究竟出在哪里了？我们来重现一下这个案例。

首先，定义一个 Get 请求的发送短信接口，里面没有任何逻辑，休眠 2 秒模拟耗时：

 复制代码

```
1 @RestController
2 @RequestMapping("ribbonretryissueserver")
3 @Slf4j
4 public class RibbonRetryIssueServerController {
5     @GetMapping("sms")
6     public void sendSmsWrong(@RequestParam("mobile") String mobile, @RequestPa
7         //输出调用参数后休眠2秒
8         log.info("{} is called, {}=>{}", request.getRequestURL().toString(), m
9         TimeUnit.SECONDS.sleep(2);
10    }
11 }
```

配置一个 Feign 供客户端调用：

 复制代码


```
1 @FeignClient(name = "SmsClient")
2 public interface SmsClient {
3     @GetMapping("/ribbonretryissueserver/sms")
4     void sendSmsWrong(@RequestParam("mobile") String mobile, @RequestParam("me:
5 }
```

Feign 内部有一个 Ribbon 组件负责客户端负载均衡，通过配置文件设置其调用的服务端为两个节点：

 复制代码

```
1 SmsClient.ribbon.listOfServers=localhost:45679,localhost:45678
```


写一个客户端接口，通过 Feign 调用服务端：

 复制代码

```
1 @RestController
2 @RequestMapping("ribbonretryissueclient")
3 @Slf4j
4 public class RibbonRetryIssueClientController {
5     @Autowired
6     private SmsClient smsClient;
7
8     @GetMapping("wrong")
9     public String wrong() {
10         log.info("client is called");
11         try{
12             //通过Feign调用发送短信接口
13             smsClient.sendSmsWrong("13600000000", UUID.randomUUID().toString());
14         } catch (Exception ex) {
15             //捕获可能出现的网络错误
16             log.error("send sms failed : {}", ex.getMessage());
17         }
18         return "done";
19     }
20 }
```

在 45678 和 45679 两个端口上分别启动服务端，然后访问 45678 的客户端接口进行测试。因为客户端和服务端控制器在一个应用中，所以 45678 同时扮演了客户端和服务端的角色。

在 45678 日志中可以看到，29 秒时客户端收到请求开始调用服务端接口发短信，同时服务端收到了请求，2 秒后（注意对比第一条日志和第三条日志）客户端输出了读取超时的错误信息：

 复制代码

```
1 [12:49:29.020] [http-nio-45678-exec-4] [INFO ] [c.d.RibbonRetryIssueClientCont
2 [12:49:29.026] [http-nio-45678-exec-5] [INFO ] [c.d.RibbonRetryIssueServerCont
3 [12:49:31.029] [http-nio-45678-exec-4] [ERROR] [c.d.RibbonRetryIssueClientCont
```

而在另一个服务端 45679 的日志中还可以看到一条请求，30 秒时收到请求，也就是客户端接口调用后的 1 秒：

```
1 [12:49:30.029] [http-nio-45679-exec-2] [INFO ] [c.d.RibbonRetryIssueService] 复制代码
```

客户端接口被调用的日志只输出了一次，而服务端的日志输出了两次。虽然 Feign 的默认读取超时时间是 1 秒，但客户端 2 秒后才出现超时错误。**显然，这说明客户端自作主张进行了一次重试，导致短信重复发送。**

翻看 Ribbon 的源码可以发现，MaxAutoRetriesNextServer 参数默认为 1，也就是 Get 请求在某个服务端节点出现问题（比如读取超时）时，Ribbon 会自动重试一次：

复制代码

```
1 // DefaultClientConfigImpl
2 public static final int DEFAULT_MAX_AUTO_RETRIES_NEXT_SERVER = 1;
3 public static final int DEFAULT_MAX_AUTO_RETRIES = 0;
4
5 // RibbonLoadBalancedRetryPolicy
6 public boolean canRetry(LoadBalancedRetryContext context) {
7     HttpMethod method = context.getRequest().getMethod();
8     return HttpMethod.GET == method || lbContext.isOkToRetryOnAllOperations();
9 }
10
11 @Override
12 public boolean canRetrySameServer(LoadBalancedRetryContext context) {
13     return sameServerCount < lbContext.getRetryHandler().getMaxRetriesOnSameServer()
14         && canRetry(context);
15 }
16
17 @Override
18 public boolean canRetryNextServer(LoadBalancedRetryContext context) {
19     // this will be called after a failure occurs and we increment the counter
20     // so we check that the count is less than or equals to too make sure
21     // we try the next server the right number of times
22     return nextServerCount <= lbContext.getRetryHandler().getMaxRetriesOnNextServer()
23         && canRetry(context);
24 }
```


解决办法有两个：

一是，把发短信接口从 Get 改为 Post。其实，这里还有一个 API 设计问题，有状态的 API 接口不应该定义为 Get。根据 HTTP 协议的规范，Get 请求用于数据查询，而 Post 才是把数据提交到服务端用于修改或新增。选择 Get 还是 Post 的依据，应该是 API 的行为，而不是参数大小。**这里的一个误区是，Get 请求的参数包含在 Url QueryString**

中，会受浏览器长度限制，所以一些同学会选择使用 JSON 以 Post 提交大参数，使用 Get 提交小参数。

二是，将 MaxAutoRetriesNextServer 参数配置为 0，禁用服务调用失败后在下一个服务端节点的自动重试。在配置文件中添加一行即可：

```
1 ribbon.MaxAutoRetriesNextServer=0
```

 复制代码

看到这里，你觉得问题出在用户服务还是短信服务呢？

在我看来，双方都有问题。就像之前说的，Get 请求应该是无状态或者幂等的，短信接口可以设计为支持幂等调用的；而用户服务的开发同学，如果对 Ribbon 的重试机制有所了解的话，或许就能在排查问题上少走些弯路。


## 并发限制了爬虫的抓取能力

除了超时和重试的坑，进行 HTTP 请求调用还有一个常见的问题是，并发数的限制导致程序的处理能力上不去。

我之前遇到过一个爬虫项目，整体爬取数据的效率很低，增加线程池数量也无济于事，只能堆更多的机器做分布式的爬虫。现在，我们就来模拟下这个场景，看看问题出在了哪里。


假设要爬取的服务端是这样的一个简单实现，休眠 1 秒返回数字 1：

```
1 @GetMapping("server")
2 public int server() throws InterruptedException {
3     TimeUnit.SECONDS.sleep(1);
4     return 1;
5 }
```

 复制代码

爬虫需要多次调用这个接口进行数据抓取，为了确保线程池不是并发的瓶颈，我们使用一个没有线程上限的 newCachedThreadPool 作为爬取任务的线程池（再次强调，除非你非常清楚自己的需求，否则一般不要使用没有线程数量上限的线程池），然后使用 HttpClient

实现 HTTP 请求，把请求任务循环提交到线程池处理，最后等待所有任务执行完成后输出执行耗时：

 复制代码

```
1 private int sendRequest(int count, Supplier<CloseableHttpClient> client) throw:
2     //用于计数发送的请求个数
3     AtomicInteger atomicInteger = new AtomicInteger();
4     //使用HttpClient从server接口查询数据的任务提交到线程池并行处理
5     ExecutorService threadPool = Executors.newCachedThreadPool();
6     long begin = System.currentTimeMillis();
7     IntStream.rangeClosed(1, count).forEach(i -> {
8         threadPool.execute(() -> {
9             try (CloseableHttpResponse response = client.get().execute(new Http
10                 atomicInteger.addAndGet(Integer.parseInt(EntityUtils.toString(
11                 } catch (Exception ex) {
12                 ex.printStackTrace();
13             }
14         }));
15     });
16     //等到count个任务全部执行完毕
17     threadPool.shutdown();
18     threadPool.awaitTermination(1, TimeUnit.HOURS);
19     log.info("发送 {} 次请求, 耗时 {} ms", atomicInteger.get(), System.currentTim
20     return atomicInteger.get();
21 }
```


首先，使用默认的 PoolingHttpClientConnectionManager 构造的 CloseableHttpClient，测试一下爬取 10 次的耗时：

 复制代码

```
1 static CloseableHttpClient httpClient1;
2
3 static {
4     httpClient1 = HttpClients.custom().setConnectionManager(new PoolingHttpCli
5 }
6
7 @GetMapping("wrong")
8 public int wrong(@RequestParam(value = "count", defaultValue = "10") int count
9     return sendRequest(count, () -> httpClient1);
10 }
```

虽然一个请求需要 1 秒执行完成，但我们的线程池是可以扩张使用任意数量线程的。按道理说，10 个请求并发处理的时间基本相当于 1 个请求的处理时间，也就是 1 秒，但日志中

显示实际耗时 5 秒：


 复制代码

```
1 [12:48:48.122] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.h.r.RouteLimitControll
```

查看 `PoolingHttpClientConnectionManager` 源码，可以注意到有两个重要参数：

**`defaultMaxPerRoute=2`，也就是同一个主机 / 域名的最大并发请求数为 2。我们的爬虫需要 10 个并发，显然是默认值太小限制了爬虫的效率。**

`maxTotal=20`，也就是所有主机整体最大并发为 20，这也是 `HttpClient` 整体的并发度。目前，我们请求数是 10 最大并发是 10，20 不会成为瓶颈。举一个例子，使用同一个 `HttpClient` 访问 10 个域名，`defaultMaxPerRoute` 设置为 10，为确保每一个域名都能达到 10 并发，需要把 `maxTotal` 设置为 100。

 复制代码

```
1 public PoolingHttpClientConnectionManager(  
2     final HttpClientConnectionOperator httpClientConnectionOperator,  
3     final HttpConnectionFactory<HttpRoute, ManagedHttpClientConnection> connFa  
4     final long timeToLive, final TimeUnit timeUnit) {  
5     ...  
6     this.pool = new CPool(new InternalConnectionFactory(  
7         this.configData, connFactory), 2, 20, timeToLive, timeUnit);  
8     ...  
9 }  
10  
11 public CPool(  
12     final ConnFactory<HttpRoute, ManagedHttpClientConnection> connFactory,  
13     final int defaultMaxPerRoute, final int maxTotal,  
14     final long timeToLive, final TimeUnit timeUnit) {  
15     ...  
16 }}
```

`HttpClient` 是 Java 非常常用的 HTTP 客户端，这个问题经常出现。你可能会问，为什么默认值限制得这么小。

其实，这不能完全怪 `HttpClient`，很多早期的浏览器也限制了同一个域名两个并发请求。对于同一个域名并发连接的限制，其实是 HTTP 1.1 协议要求的，[🔗 这里有这么一段话](#)：



```
1 Clients that use persistent connections SHOULD limit the number of simultaneou:
```

HTTP 1.1 协议是 20 年前制定的，现在 HTTP 服务器的能力强很多了，所以有些新的浏览器没有完全遵从 2 并发这个限制，放开并发数到了 8 甚至更大。如果需要通过 HTTP 客户端发起大量并发请求，不管使用什么客户端，请务必确认客户端的实现默认的并发度是否满足需求。

既然知道了问题所在，我们就尝试声明一个新的 HttpClient 放开相关限制，设置 maxPerRoute 为 50、maxTotal 为 100，然后修改一下刚才的 wrong 方法，使用新的客户端进行测试：

```
1 httpClient2 = HttpClient.custom().setMaxConnPerRoute(10).setMaxConnTotal(20).l
```

输出如下，10 次请求在 1 秒左右执行完成。可以看到，因为放开了一个 Host 2 个并发的默认限制，爬虫效率得到了大幅提升：

```
1 [12:58:11.333] [http-nio-45678-exec-3] [INFO ] [o.g.t.c.h.r.RouteLimitControll
```

## 重点回顾

今天，我和你分享了 HTTP 调用最常遇到的超时、重试和并发问题。

连接超时代表建立 TCP 连接的时间，读取超时代表了等待远端返回数据的时间，也包括远端程序处理的时间。在解决连接超时问题时，我们要搞清楚连的是谁；在遇到读取超时问题的时候，我们要综合考虑下游服务的服务标准和自己的服务标准，设置合适的读取超时时间。此外，在使用诸如 Spring Cloud Feign 等框架时务必确认，连接和读取超时参数的配置是否正确生效。

对于重试，因为 HTTP 协议认为 Get 请求是数据查询操作，是无状态的，又考虑到网络出现丢包是比较常见的事情，有些 HTTP 客户端或代理服务器会自动重试 Get/Head 请求。

如果你的接口设计不支持幂等，需要关闭自动重试。但，更好的解决方案是，[🔗 遵从 HTTP 协议](#)的建议来使用合适的 HTTP 方法。

最后我们看到，包括 HttpClient 在内的 HTTP 客户端以及浏览器，都会限制客户端调用的最大并发数。如果你的客户端有比较大的请求调用并发，比如做爬虫，或是扮演类似代理的角色，又或者是程序本身并发较高，如此小的默认值很容易成为吞吐量的瓶颈，需要及时调整。

今天用到的代码，我都放在了 GitHub 上，你可以点击[🔗 这个链接](#)查看。

## 思考与讨论

1. 第一节中我们强调了要注意连接超时和读取超时参数的配置，大多数的 HTTP 客户端也都有这两个参数。有读就有写，但为什么我们很少看到“写入超时”的概念呢？
2. 除了 Ribbon 的 AutoRetriesNextServer 重试机制，Nginx 也有类似的重试功能。你了解 Nginx 相关的配置吗？

针对 HTTP 调用，你还遇到过什么坑吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你  
攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有[现金](#)奖励。

上一篇 04 | 连接池：别让连接池帮了倒忙

下一篇 06 | 20%的业务代码的Spring声明式事务，可能都没处理正确

## 精选留言 (25)

写留言



Monday 置顶

2020-03-20

我们来分析一下源码。打开 RibbonClientConfiguration 类后，会看到 DefaultClientConfigurationImpl 被创建出来之后，ReadTimeout 和 ConnectTimeout 被设置为 1s：

```
/**  
 * Ribbon client default connect timeout....
```

展开 ▾

作者回复: 1、启动时不进断点不代表不是，执行后会进断点，原因是LoadBalancerFeignClient.execute()，运行时注入依赖的，这个方法一路追下去：

```
IClientConfig getClientConfig(Request.Options options, String clientName) {  
    IClientConfig requestConfig;  
    if (options == DEFAULT_OPTIONS) {  
        requestConfig = this.clientFactory.getClientConfig(clientName);  
    }  
    else {  
        requestConfig = new FeignOptionsClientConfig(options);  
    }  
    return requestConfig;  
}
```

2、ribbon是netflix的三方库，不是spring boot @ConfigurationProperties玩法，Key定义在：

com.netflix.client.config.CommonClientConfigKey



4



Darren 置顶

2020-03-19

试着回答下问题：

1、为什么很少见到写入超时，客户端发送数据到服务端，首先接力连接（TCP），然后写入TCP缓冲区，TCP缓冲区根据时间窗口，发送数据到服务端，因此写入操作可以任务是自己本地的操作，本地操作是不需要什么超时时间的，如果真的有什么异常，那也是连接（TCP）不上，或者超时的问题，连接超时和读取超时就能覆盖这种场景。...

展开 ∨

作者回复: 📬



👍 4



小美 置顶

2020-03-19

老师，我这边工作过程中遇到服务端 499 这块要怎么从链接超时和读取超时设置去分析呢？

作者回复: 499情况比较特殊，虽然表现为服务端（一般为代理，比如nginx）记录和返回499状态码，但是其实是因为处理时间太长，客户端超时主动关闭连接，排查两点：

- 1、客户端读取超时时间多久
- 2、服务端为什么处理这么慢超过了客户端的读取超时

如果希望不要499的话，对于nginx可以开启  
proxy\_ignore\_client\_abort，这样可以让请求在服务端执行完成



👍 1



2020-03-19

这已经不仅仅是一个坑了，而是N一个场景下，多种多样的坑。

Spring Boot 带来了【约定大于配置】的说法，但是，本文告诉我们，越是约定大于配置，越是要对那些“默认配置”心里有数才行。

HTTP请求，说到底，还是网络调用。某个老师曾说过，网络，就是不靠谱的。就存在拥塞，丢包等各种情况。从而使得排查的难度更大。要考虑的角度，宽度，都更广。不单...

展开 ∨

作者回复: 总结的不错



👍 6



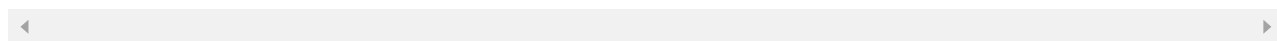
**Monday**  
2020-03-20

花了两个晚上终于还是把这节啃了下来，准备运行环境，重现所有问题，翻看相关源码。终于等到你，还好我没放弃。

个人感悟，这些坑对以后快速排查问题，肯定有帮助。就算以后淡忘了这节的内容，但至少还会有些许记忆的，哪个专栏，哪个老师，哪篇文章。感谢老师！

展开 ▾

作者回复: 如果觉得有用可以多转发分享



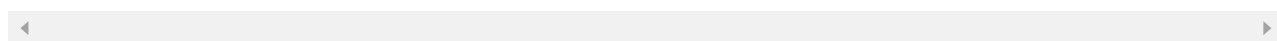
👍 2



**一个想偷懒的程序坑**  
2020-03-20

虽然没处理过这块儿的东西，但看完了解了许多知识，赞！

作者回复: 如果觉得有用可以多转发分享



👍 2

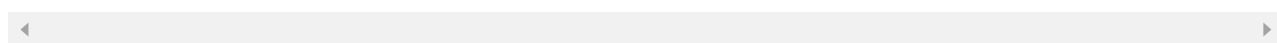


**Monday**  
2020-03-19

好文章，好“坑”。

展开 ▾

作者回复: 如果觉得好，可以多分享转发



👍 2

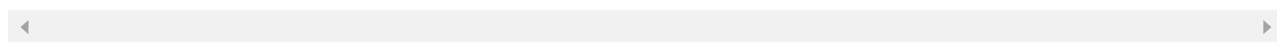


**justinzhong**  
2020-03-21

老师，针对发送短信的那个例子，解决重试问题的方法一：就是get请求换成post请求，我试了几次都是不行的，还是会重试一次，但是方法二是完全可以的。可以针对方法一的解决重试问题的思路再描述的清楚一些吗？

展开 ▾

作者回复: 1. 确定没看错，也就是RibbonRetryIssueServerController只看到一次输出  
2. 确定参数ribbon.OkToRetryOnAllOperations没有设置为true





1

**斐波那契**

2020-03-20

里面说的坑也许过了一段时间就忘了 当时有四个字是我学到的 那就是“查看源码”



1

**陈天柱**

2020-03-20

之前用Spring Cloud就遇到过feign调用超时的坑，始终配置readTimeout值都不生效。虽然后面网上查阅了资料暂时性解决了，但是看了老师的解决问题思路才发现，这个时候就需要带着问题去阅读源码找寻答案，提高自己阅读源码的能力。

展开 ▾



1

**汝林外史**

2020-03-20

sendRequest(int count, Supplier<CloseableHttpClient> client) 这个方法中第二个参数为什么要用一个函数接口而不是直接用CloseableHttpClient类型呢？我看也没用到什么特性，只是调用了execute方法而已？

课后问题：1. 感觉写入超时已经包含在读取超时这个里面，没必要单独定义这么细的超时。...

展开 ▾

作者回复: 这里是可以直接传CloseableHttpClient的



1

**Alpha**

2020-03-20

非常同意选择Get还是Post应该依据API的行为。

但是有时数据查询的API参数确实不得已很长，会导致浏览器的长度限制，老师有好的办法吗？

展开 ▾

作者回复: 这么长的参数看看是否合理，对于有一些数据它可能并不是查询参数可以放头里传



1



**Monday**  
2020-03-19

```
public class ClientReadTimeoutController {  
    private String getResponse(String url, int connectTimeout, int readTimeout) throws IOException {  
        return Request.Get("http://localhost:45678/clientreadtimeout" + url)  
            .connectTimeout(connectTimeout)...
```

展开 ∨

作者回复: 源码里面有, 在clientreadtimeout里

Request是在这里:

```
<dependency>  
    <groupId>org.apache.httpcomponents</groupId>  
    <artifactId>fluent-hc</artifactId>  
    <version>4.5.9</version>  
</dependency>
```

1

1



**终结者999号**

2020-03-19

老师, 对于Http Client和Ok Http相比, 是不是OkHttp支持得更好, 而且HTTP2相比于HTTP1.1的新特性是不是也使得我们不用过去的一些配置了啊

展开 ∨

作者回复: 我个人觉得okhttp易用性更高一点, 不过okhttp应该在安卓领域更火一点, 后端使用okhttp的应该不多。万变不离其宗, 使用任何httpclient都要考虑连接池、超时配置、自动重试和并发问题

1

1



**大尾巴老猫**

2020-03-19

```
void server();  
这一句什么意思?
```

作者回复: 就是模拟一个服务端接口

1

1





**Monday**

2020-03-19

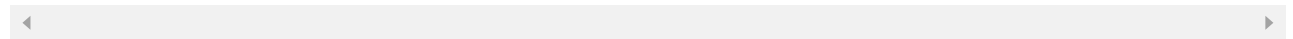
`ribbon.ReadTimeout=4000`

`ribbon.ConnectTimeout=4000`

这个参数的key命名不规范，是有故事，还是开发人员不够专业？

展开 ∨

作者回复: 这就不清楚了



1

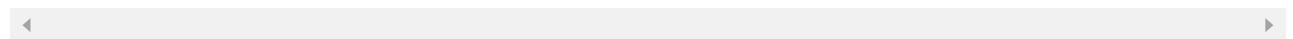


**小美**

2020-03-19

老师我是做客户端的，我们这边还有个写超时概念这块老师方便分享下不

作者回复: 其实也就是到socket sendbuffer的超时（满的话等待空间释放的超时）



1



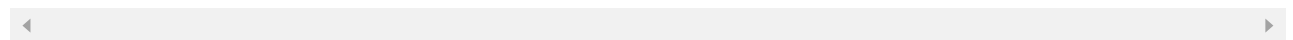
**公号-云原生程序员**

2020-03-19

老师总结的很有深度、很全面、很有业务实战

展开 ∨

作者回复: 如果觉得好，可以多分享



1



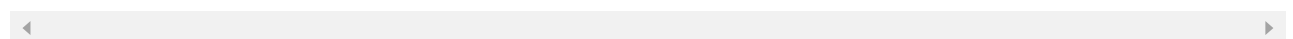
**pedro**

2020-03-19

对于数据写入，开发者都可以直接控制，要么先write然后再一次性flush，要么边write边flush，至于最后socket缓冲区中的数据如何发送，都交给了tcp。

展开 ∨

作者回复: 嗯大概意思对 可以再搜一下相关资料继续研究一下



1



梦倚栏杆

2020-03-19

按照老师解释的读取超时的概念：字节流放入socket--->服务端处理----->服务端返回-->取出字节流。

那写入超时估计就是字节流放入socket的时间，这个属于自己主动控制的可能没有必要吧，具体可能还需要了解一下网络编程才能知道。

展开 ∨

作者回复: 嗯大概意思对 可以再搜一下相关资料继续研究一下

