一个简单可参考的API网关架构设计

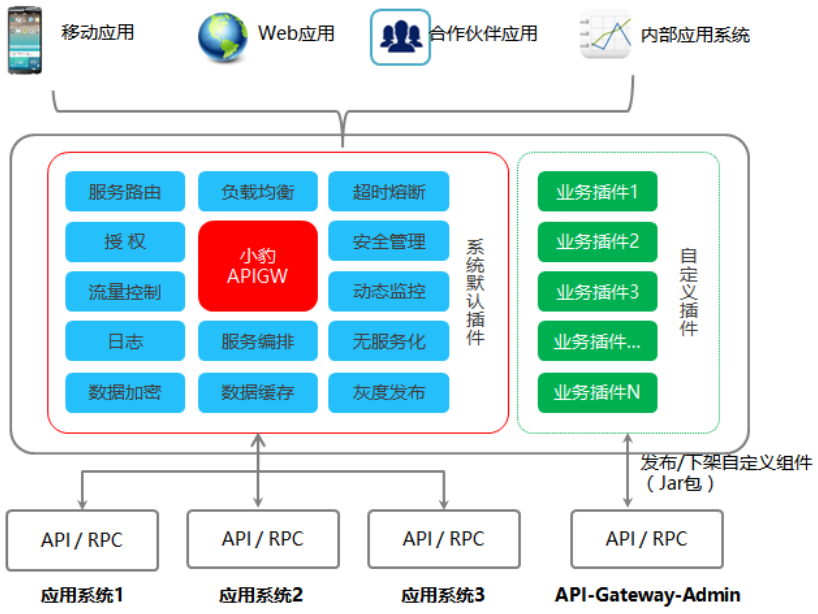
作者 王苏龙 王苏龙 关注 1 他的粉丝 发布于 2018年3月28日. 估计阅读时间: 36 分钟 来 QCon上海2018 关注大数据平台技术选型、搭建、系统迁移和优化的经验。 讨论

网关一词较早出现在网络设备里面，比如两个相互独立的局域网段之间通过路由器或者桥接设备进行通信， 这中间的路由或者桥接设备我们称之为网关。

相应的 API 网关将各系统对外暴露的服务聚合起来，所有要调用这些服务的系统都需要通过 API 网关进行访问，基于这种方式网关可以对 API 进行统一管控，例如：认证、鉴权、流量控制、协议转换、监控等等。

API 网关的流行得益于近几年微服务架构的兴起，原本一个庞大的业务系统被拆分成许多粒度更小的系统进行独立部署和维护，这种模式势必会带来更多的跨系统交互，企业 API 的规模也会成倍增加，API 网关（或者微服务网关）就逐渐成为了微服务架构的标配组件。

如下是我们整理的 API 网关的几种典型应用场景：



1、面向 Web 或者移动 App

这类场景，在物理形态上类似前后端分离，前端应用通过 API 调用后端服务，需要网关具有认证、鉴权、缓存、服务编排、监控告警等功能。

2、面向合作伙伴开放 API

这类场景，主要为了满足业务形态对外开放，与企业外部合作伙伴建立生态圈，此时的 API 网关注重安全认证、权限分级、流量管控、缓存等功能的建设。

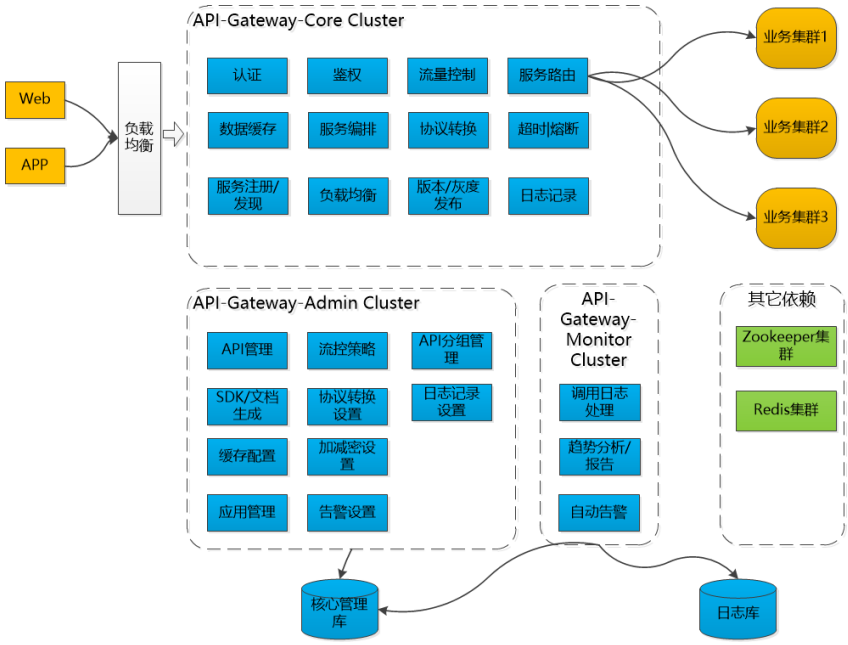
3、企业内部系统互联互通

对于中大型的企业内部往往有几十、甚至上百个系统，尤其是微服务架构的兴起系统数量更是急剧增加。系统之间相互依赖，逐渐形成网状调用关系不便于管理和维护，需要 API 网关进行统一的认证、鉴权、流量管控、超时熔断、监控告警管理，从而提高系统的稳定性、降低重复建设、运维管理等成本。

设计目标

1. 纯 Java 实现；
2. 支持插件化，方便开发人员自定义组件；
3. 支持横向扩展，高性能；
4. 避免单点故障，稳定性要高，不能因为某个 API 故障导致整个网关停止服务；
5. 管理控制台配置更新可自动生效，不需要重启网关；

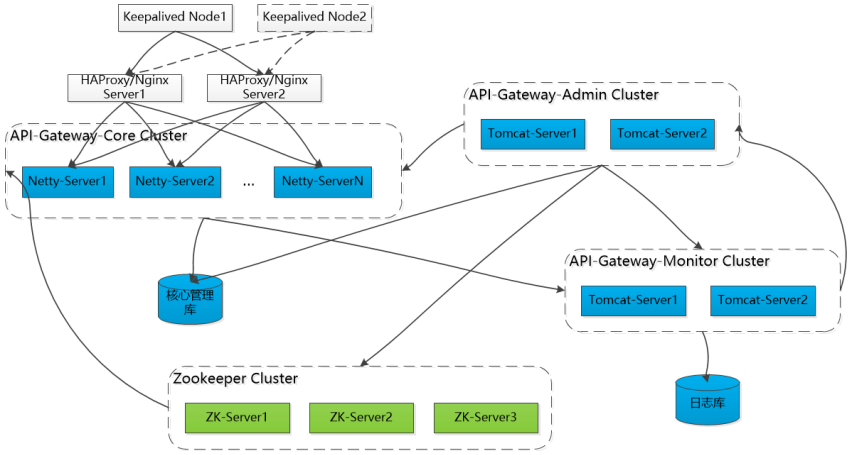
应用架构设计



整个平台拆分成 3 个子系统，Gateway-Core（核心子系统）、Gateway-Admin（管理中心）、Gateway-Monitor（监控中心）。

- Gateway-Core 负责接收客户端请求，调度、加载和执行组件，将请求路由到上游服务端，处理上游服务端返回的结果等；
- Gateway-Admin 提供统一的管理界面，用户可在此进行 API、组件、系统基础信息的设置和维护；
- Gateway-Monitor 负责收集监控日志、生成各种运维管理报表、自动告警等；

系统架构设计



说明：

1. 网关核心子系统通过 HAProxy 或者 Nginx 进行负载均衡，为避免正好路由的 LB 节点服务不可用，可以考虑在此基础上增加 Keepalived 来实现 LB 的失效备援，当 LB Node1 停止服务，Keepalived 会将虚拟 IP 自动飘移到 LB Node2，从而避免因为负载均衡器导致单点故障。DNS 可以直接指向 Keepalived 的虚拟 IP。
2. 网关除了对性能要求很高外，对稳定性也有很高的要求，引入 Zookeeper 及时将 Admin 对 API 的配置更改同步刷新到各网关节点。
3. 管理中心和监控中心可以采用类似网关子系统的高可用策略，如果嫌麻烦管理中心可以省去 Keepalived，相对来说管理中心没有这么高的可用性要求。
4. 理论上监控中心需要承载很大的数据量，比如有 1000 个 API，平均每个 API 一天调用 10 万次，对于很多互联网公司单个 API 的量远远大于 10 万，如果将每次调用的信息都存储起来太浪费，也没有太大的必要。可以考虑将 API 每分钟的调用情况汇总后进行存储，比如 1 分钟的平均响应时间、调用次数、流量、正确率等等。
5. 数据库选型可以灵活考虑，原则上网关在运行时要尽可能减少对 DB 的依赖，否则 IO 延时会严重影响网关性能。可以考虑首次访问后将 API 配置信息缓存，Admin 对 API 配置更改后通过 Zookeeper 通知网关刷新，这样一来 DB 的访问量可以忽略不计，团队可根据自身偏好灵活选型。

非阻塞式 HTTP 服务

管理和监控中心可以根据团队的情况采用自己熟悉的 Servlet 容器部署，网关核心子系统对性能的要求非常高，考虑采用 NIO 的网络模型，实现纯 HTTP 服务即可，不需要实现 Servlet 容器，推荐 Netty 框架（设计优雅，大名鼎鼎的 Spring Webflux 默认都是使用的 Netty，更多的优势就不在此详述了），内部测试在相同的机器上分别通过 Tomcat 和 Netty 生成 UUID，Netty 的性能大约有 20% 的提升，如果后端服务响应耗时较高的话吞吐量还有更大的提升。（补充：Netty4.x 的版本即可，不要采用 5 以上的版本，有严重的缺陷没有解决）

采用 Netty 作为 Http 容器首先需要解决的是 Http 协议的解析和封装，好在 Netty 本身提供了这样的 Handler，具体参考如下代码：

1、构建一个单例的 HttpServer，在 SpringBoot 启动的时候同时加载并启动 Netty 服务

```
int sobacklog = Integer.parseInt(AppConfigUtil.getValue("netty.sobacklog"));

ServerBootstrap b = new ServerBootstrap();

b.group(bossGroup, workerGroup)

.channel(NioServerSocketChannel.class)

.localAddress(new InetSocketAddress(this.portHTTP))

.option(ChannelOption.SO_BACKLOG, sobacklog)

.childHandler(new ChannelHandlerInitializer(null));

// 绑定端口

ChannelFuture f = b.bind(this.portHTTP).sync();

logger.info("HttpServer name is " + HttpServer.class.getName() + " started and
listen on " + f.channel().localAddress());
```

2、初始化 Handler

```
@Override

protected void initChannel(SocketChannel ch) throws Exception {

ChannelPipeline p = ch.pipeline();

p.addLast(new HttpRequestDecoder());

p.addLast(new HttpResponseEncoder());

int maxContentLength = 2000;

try {

maxContentLength =
Integer.parseInt(AppConfigUtil.getValue("netty.maxContentLength"));

} catch (Exception e) {

logger.warn("netty.maxContentLength 配置异常，系统默认为：2000KB");

}

p.addLast(new HttpObjectAggregator(maxContentLength * 1024));// HTTP 消息的合并
处理

p.addLast(new HttpServerInboundHandler());

}
```

HttpRequestDecoder 和 HttpResponseEncoder 分别实现 Http 协议的解析和封装，Http Post 内容超过一个数据包大小会自动分组，通过 HttpObjectAggregator 可以自动将这些数据粘合在一起，对于上层收到是一个完整的 Http 请求。

3、通过 HttpServerInboundHandler 将网络请求转发给网关执行器

```
@Override

public void channelRead0(ChannelHandlerContext ctx, Object msg)

throws Exception {

try {

if (msg instanceof HttpRequest && msg instanceof HttpContent) {

CmptRequest cmptRequest = CmptRequestUtil.convert(ctx, msg);

CmptResult cmptResult = this.gatewayExecutor.execute(cmptRequest);

FullHttpResponse response = encapsulateResponse(cmptResult);

ctx.write(response);

ctx.flush();

}

} catch (Exception e) {

logger.error("网关入口异常，" \+ e.getMessage());

e.printStackTrace();

}

}
```

设计上建议将 Netty 接入层代码跟网关核心逻辑代码分离，不要将 Netty 收到 HttpRequest 和 HttpContent 直接给到网关执行器，可以考虑做一层转换封装成自己的 Request 给到执行器，方便后续可以很容易的将 Netty 替换成其它 Http 容器。（如上代码所示，CmptRequest 即为自定义的 Http 请求封装类，CmptResult 为网关执行结果类）

组件化及自定义组件支持

组件是网关的核心，大部分功能特性都可以基于组件的形式提供，组件化可以有效提高网关的扩展性。

先来看一个简单的微信认证组件的例子：

如下实现的功能是对 API 请求传入的 Token 进行校验，其结果分别是认证通过、Token 过期和无效 Token，认证通过后再将微信 OpenID 携带给上游服务系统。

```
/**

* 微信 token 认证，token 格式:

* {appID:'',openID:'',timestamp:132525144172,sessionKey: ''}

*

public class WeixinAuthTokenCmpt extends AbstractCmpt {

private static Logger logger =
LoggerFactory.getLogger(WeixinAuthTokenCmpt.class);

private final CmptResult SUCCESS_RESULT;

public WeixinAuthTokenCmpt() {

SUCCESS_RESULT = buildSuccessResult();

}

@Override

public CmptResult execute(CmptRequest request, Map<String, FieldDTO> config) {

if (logger.isDebugEnabled()) {

logger.debug("WeixinTokenCmpt ......");

}

CmptResult cmptResult = null;

//Token 认证超时间（传入单位：分）

long authTokenExpireTime = getAuthTokenExpireTime(config);

WeixinTokenDTO authTokenDTO = this.getAuthTokenDTO(request);

logger.debug("Token=" + authTokenDTO);

AuthTokenState authTokenState = validateToken(authTokenDTO,
authTokenExpireTime);

switch (authTokenState) {

case ACCESS: {

cmptResult = SUCCESS_RESULT;

Map<String, String> header = new HashMap<>();
```

```
header.put(HeaderKeyConstants.HEADER\_APP\_ID\_KEY, authTokenDTO.getAppID());

header.put(CmptHeaderKeyConstants.HEADER\_WEIXIN\_OPENID\_KEY,
authTokenDTO.getOpenID());

header.put(CmptHeaderKeyConstants.HEADER\_WEIXIN\_SESSION\_KEY,
authTokenDTO.getSessionKey());

cmptResult.setHeader(header);

break;

}

case EXPIRED: {

cmptResult = buildCmptResult(RespErrCode.AUTH\_TOKEN\_EXPIRED, "token 过期, 请重
新获取 Token! ");

break;

}

case INVALID: {

cmptResult = buildCmptResult(RespErrCode.AUTH\_INVALID\_TOKEN, "Token 无效! ");

break;

}

}

return cmptResult;

}

...

}
```

上面例子看不懂没关系，接下来会详细阐述组件的设计思路。

1、组件接口定义

```
public interface ICmpt {
    /**
     * 组件执行入口
     *
     * @param request
     * @param config，组件实例的参数配置
     * @return
     */
    CmptResult execute(CmptRequest request, Map<String, FieldDTO> config);

    /**
     * 销毁组件持有的特殊资源，比如线程。
     */
    void destroy();
}
```

execute 是组件执行的入口方法，request 前面提到过是 http 请求的封装，config 是组件的特殊配置，比如上面例子提到的微信认证组件就有一个自定义配置 -Token 的有效期，不同的 API 使用该组件可以设置不同的有效期。

FieldDTO 定义如下：

```
public class FieldDTO {

private String title;

private String name;

private FieldType fieldType = FieldType.STRING;

private String defaultValue;

private boolean required;

private String regExp;

private String description;

}
```

CmptResult 为组件执行后的返回结果，其定义如下：

```
public class CmptResult {

RespErrMsg respErrMsg;// 组件返回错误信息

private boolean passed;// 组件过滤是否通过

private byte\[\] data;// 组件返回数据

private Map<String, String> header = new HashMap<String, String>();// 透传后端服务响应头信息

private MediaType mediaType;// 返回响应数据类型

private Integer statusCode = 200;// 默认返回状态码为 200

}
```

2、组件类型定义

执行器需要根据组件类型和组件执行结果判断是要直接返回客户端还是继续往下面执行，比如认证类型的组件，如果认证失败是不能继续往下执行的，但缓存类型的组件没有命中才继续往下执行。当然这样设计存在一些缺陷，比如新增组件类型需要执行器配合调整处理逻辑。（Kong 也提供了大量的功能组件，没有研究过其网关框架是如何跟组件配合的，是否支持用户自定义组件类型，知道的朋友详细交流下。）

初步定义如下组件类型：

认证、鉴权、流量管控、缓存、路由、日志等。

其中路由类型的组件涵盖了协议转换的功能，其负责调用上游系统提供的服务，可以根据上游系统提供 API 的协议定制不同的路由组件，比如：Restful、WebService、Dubbo、EJB 等等。

3、组件执行位置和优先级设定

执行位置：Pre、Routing、After，分别代表后端服务调用前、后端服务调用中和后端服务调用完成后，相同位置的组件根据优先级决定执行的先后顺序。

4、组件发布形式

组件打包成标准的 Jar 包，通过 Admin 管理界面上传发布。

附 - 组件可视化选择 UI 设计

组件热插拔设计和实现

JVM 中 Class 是通过类加载器 + 全限定名来唯一标识的，上面章节谈到组件是以 Jar 包的形式发布的，但相同组件的多个版本的入口类名需要保持不变，因此要实现组件的热插拔和多版本并存就需要自定义类加载器来实现。

大致思路如下：

网关接收到 API 调用请求后根据请求参数从缓存里拿到 API 配置的组件列表，然后再逐一参数从缓存里获取组件对应的类实例，如果找不到则尝试通过自定义类加载器载入 Jar 包，并初始化组件实例及缓存。

附 - 参考示例

```
public static ICmpt newInstance(final CmptDef cmptDef) {
    ICmpt cmpt = null;
    try {
        final String jarPath = getJarPath(cmptDef);
        if (logger.isDebugEnabled()) {
            logger.debug("尝试载入 jar 包,jar 包路径: " + jarPath);
        }
        // 加载依赖 jar
        CmptClassLoader cmptClassLoader =
CmptClassLoaderManager.loadJar(jarPath, true);
        // 创建实例
        if (null ≠ cmptClassLoader) {
            cmpt = LoadClassUtil.newObject(cmptDef.getFullQualifiedName(),
ICmpt.class, cmptClassLoader);
        } else {
            logger.error("加载组件 jar 包失败! jarPath: " + jarPath);
        }
    } catch (Exception e) {
        logger.error("组件类加载失败，请检查类名和版本是否正确。ClassName=" +
cmptDef.getFullQualifiedName() + ", Version=" + cmptDef.getVersion());
        e.printStackTrace();
    }
    return cmpt;
}
```

补充说明：

自定义类加载器可直接需要继承至 URLClassLoader，另外必须指定其父类加载器为执行器的加载器，否则组件没法引用网关的其它类。

API 故障隔离及超时、熔断处理

在详细阐述设计前先讲个实际的案例，大概 12 年的时候某公司自研了一款 ESB 的中间件（企业服务总线跟 API 网关很类似，当年 SOA 理念大行其道的时候都推崇的是 ESB，侧重服务的编排和异构系统的整合。），刚开始用的还行，但随着接入系统的增多，突然某天运维发现大量 API 出现缓慢甚至超时，初步检查发现 ESB 每个节点的线程几乎消耗殆尽，起初判断是资源不够，紧急扩容后还是很快线程占满，最终导致上百个系统瘫痪。

最终找到问题的症结是某个业务系统自身的原因导致服务不可用，下游业务系统请求大量堆积到 ESB 中，从而导致大量线程堵塞。

以上案例说明了一个在企业应用架构设计里面的经典原则 - 故障隔离，由于所有的 API 请求都要经过网关，必须隔离 API 之间的相互影响，尤其是个别 API 故障导致整个网关集群服务中断。

接下来分别介绍故障隔离、超时管控、熔断的实现思路。

1、故障隔离

有两种方式可以实现，一是为每个 API 创建一个线程池，每个线程分配 10~20 个线程，这也是常用的隔离策略，但这种方式有几个明显的缺点：

- 线程数会随着 API 接入数量递增，1000 个 API 就需要 2 万个线程，光线程切换对 CPU 就是不小的开销，而其线程还需要占用一定的内存资源；

- 平均分配线程池大小导致个别访问量较大且响应时间相对较长的 API 吞吐量上不去；
- Netty 本身就有工作线程池了，再增加 API 的线程池，导致某些需要 ThreadLocal 特性的编程变得困难。

二是用信号量隔离，直接复用 Netty 的工作线程，上面线程池隔离提到的 3 个缺点都可以基本避免， 建议设置单个 API 的信号量个数小于等于 Netty 工作线程池数量的 1/3，这样既兼顾了单个 API 的性能又不至于单个 API 的问题导致整个网关堵塞。

具体实现可以考虑直接引用成熟的开源框架，推荐 Hystrix，可以同时解决超时控制和熔断。

参考配置如下：

```
Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey(groupKey))
        .andCommandKey(HystrixCommandKey.Factory.asKey(commandKey ))
        .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                // 舱壁隔离策略 - 信号量

 .withExecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolationStrategy.SEMAPHORE)

                // 设置每组 command 可以申请的信号量最大数

 .withExecutionIsolationSemaphoreMaxConcurrentRequests(CmptInvoker.maxSemaphore)
                /* 开启超时设置 */
                .withExecutionIsolationThreadInterruptOnTimeout(true)
                /* 超时时间设置 */
                .withExecutionIsolationThreadTimeoutInMilliseconds(timeout)
                .withCircuitBreakerEnabled(true)// 开启熔断

 .withCircuitBreakerSleepWindowInMilliseconds(Constants.DEFAULT_CIRCUIT_BREAKER_SLEEP_WINDOW_IN_MI
 5 秒后会尝试闭合回路
```

2、超时管控

API 的超时控制是必须要做的，否则上游服务即便是间歇性响应缓慢也会堵塞大量线程（虽然通过信号量隔离后不会导致整个网关线程堵塞）。

其次，每个 API 最好可以单独配置超时时间，但不建议可以让用户随意设置，还是要有个最大阈值。（API 网关不适合需要长时间传输数据的场景，比如大文件上传或者下载、DB 数据同步等）

实现上可以直接复用开源组件的功能，比如：HttpClient 可以直接设置获取连接和 Socket 响应的超时时间，Hystrix 可以对整个调用进行超时控制等。

3、熔断

熔断类似电路中的保险丝，当超过负荷或者电阻被击穿的时候自动断开对设备起到保护作用。在 API 网关中设置熔断的目的是快速响应请求，避免不必要的等待，比如某个 API 后端服务正常情况下 1s 以内响应，但现在因为各种原因出现堵塞大部分请求 20s 才能响应，虽然设置了 10s 的超时控制，但让请求线程等待 10s 超时不仅没有意义，反而会增加服务提供方的负担。

为此我们可以设置单位时间内超过多少比例的请求超时或者异常，则直接熔断链路，等待一段时间后再次尝试恢复链路。

实现层面可以直接复用 Hystrix。

运行时配置更新机制

前面章节提到过出于性能考虑网关在运行时要尽可能减小对 DB 的访问，设计上可以将 API、组件等关键内容进行缓存，这样一来性能是提升了，但也带来了新的问题，比如 Admin 对 API 或者组件进行配置调整后如何及时更新到集群的各个网关节点。

解决方案很多，比如引入消息中间件，当 Admin 调整配置后就往消息中心发布一条消息，各网关节点订阅消息，收到消息后刷新缓存数据。

我们在具体实现过程中采用的是 Zookeeper 集群数据同步机制，其实现原理跟消息中间件很类似，只不过网关在启动的时候就会向 ZK 节点进行注册，也是被动更新机制。

性能考虑

性能是网关一项非常重要的衡量指标，尤其是响应时间，客户端本来可以直连服务端的，现在增加了一个网关层，对于一个本身耗时几百毫秒的服务接入网关后增加几毫秒，影响倒是可以忽略不计；但如果服务本身只需要几毫秒，因为接入网关再增加一倍的延时，用户感受就会比较明显。

建议在设计上需要遵循如下原则：

1. 核心网关子系统必须是无状态的，便于横向扩展。
2. 运行时不依赖本地存储，尽量在内存里面完成服务的处理和中转。
3. 减小对线程的依赖，采用非阻塞式 IO 和异步事件响应机制。
4. 后端服务如果是 HTTP 协议，尽量采用连接池或者 Http2，测试连接复用和不复用性能有几倍的差距。（TCP 建立连接成本很高）

附 -HttpClient 连接池设置：

```
PoolingHttpClientConnectionManager cmOfHttp = new
PoolingHttpClientConnectionManager();
cmOfHttp.setMaxTotal(maxConn);
cmOfHttp.setDefaultMaxPerRoute(maxPerRoute);
httpClient = HttpClients.custom()
        .setConnectionManager(cmOfHttp)
        .setConnectionManagerShared(true)
        .build();
```

说明：

httpClient 对象可以作为类的成员变量长期驻留内存，这个是连接池复用的前提。

结语

API 网关作为企业 API 服务的汇聚中心，其良好的性能、稳定性和可扩展性是基础，只有这个基础打扎实了，我们才能在上面扩展更多的特性。

这篇文章主要介绍网关的总体架构设计， 后面的篇幅在详细探讨下各种组件的具体设计和实现。

有兴趣的朋友可以加入 QQ 交流群：244054462，备注：API 网关架构设计交流。

作者介绍

王苏龙，成都小豹科技的技术负责人，长期关注企业各类中间件平台和软件架构设计，带队设计和实现了小豹 API 网关产品。加入小豹科技前在平安科技任公共平台资深架构师，主导和参与了多个基础平台的研发和设计，如：业务流程管理平台、集中用户权限管理平台、规则引擎平台等。

感谢郭蕾对本文的审校。