目　录

介绍

深入理解BFE

本书围绕BFE开源项目，向读者介绍网络接入的相关技术原理，说明BFE开源软件的设计思想，及如何基于BFE开源软件搭建网络接入平台。具有开发能力的读者也可根据本书的说明，按照自己的需要开发BFE的扩展模块，或者向BFE开源项目贡献代码。

BFE开源项目

BFE是百度统一的七层负载均衡接入转发平台。BFE平台从2012年开始建设，截至2020年底，BFE平台每日转发的请求超过万亿，日峰值请求超过1000万QPS，在业界有巨大影响力。2019年7月，BFE的转发引擎对外开源，并于2020年6月被CNCF（云原生计算基金会）接受为“沙盒项目”（Sandbox Project），这是网络方向中国首个被CNCF接受的开源项目。

BFE开源项目地址: https://github.com/bfenetworks/bfe

本书作者

姓名	Github ID
章淼	mileszhang2016
杨思杰	iyangsj
戴明	daimg
陶春华	ohscartao

版权许可

背景篇

BFE简介

BFE简介

什么是BFE？

BFE最初是Baidu Front End的缩写（中文名为“百度统一前端”），是百度统一的七层负载均衡接入转发平台。BFE平台从2012年开始建设。截至2020年底，BFE平台每日转发的请求超过1万亿，日峰值请求超过1000万QPS。

2014年，BFE平台的核心转发引擎基于Go语言重构，并于2015年1月在百度全量上线。在中国范围内，BFE是较早将Go语言用于负载均衡场景、及大规模使用的项目。

2019年初，BFE平台成功的支持了百度春晚红包项目。在本次项目中，BFE平台提供了亿级别的转发能力，在海量的流量下支持了HTTPS卸载、精确限流等关键能力，保证了活动的顺利进行。

2019年7月，BFE的转发引擎对外开源。因为BFE项目在业界的巨大影响力，开源项目名称仍保持为BFE，但改名为Beyond Front End。希望通过BFE的开源，能够推动负载均衡技术的发展。

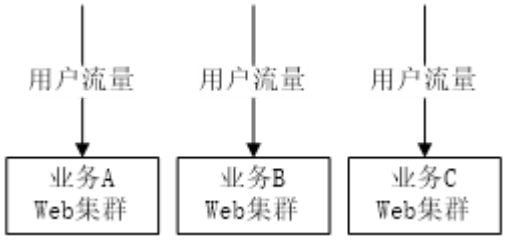
2020年6月，BFE被CNCF（云原生计算基金会）接受为“沙盒项目”（Sandbox Project）。

BFE开源项目的地址为 https://github.com/bfenetworks/bfe

BFE平台

为什么需要BFE平台？

在传统的方案中，并不存在统一的七层负载均衡接入层。在存在多个服务的场景下，各业务流量在经过四层负载均衡的转发后，直接到达业务的Web服务集群。



这种方案存在以下问题：

(1) 功能重复开发

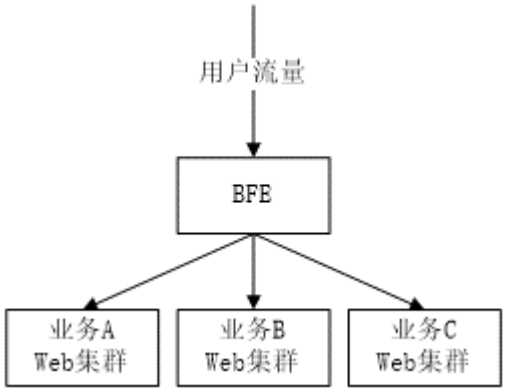
在协议处理、安全等方面，有很多重要的功能是都需要支持的。考虑到各种业务集群在语言、技术栈上的差异，在多个不同的业务集群上支持相同的功能会带来很高的研发成本。

(2) 运维成本高

在某些场景下，需要对于多个业务同时上线相同的安全策略。在缺乏统一七层负载均衡接入层的情况下，需要在多个业务集群反复上线。这不仅仅带来大量的上线工作成本，而且也使得策略的上线时间很长。

(3) 流量统一控制能力低

由于各业务集群的分散，从公司的层面，缺乏对于流量情况的统一观察和控制能力。这也阻碍了对于网络资源的统一控制，及对网络服务质量的统一管理。



在引入BFE平台后，所有流量都经过BFE的接入转发后才到达业务集群，从而带来了以下优点：

(1) 功能统一开发

无论业务集群的技术差异，各种相关功能只需要在BFE做一次开发即可。

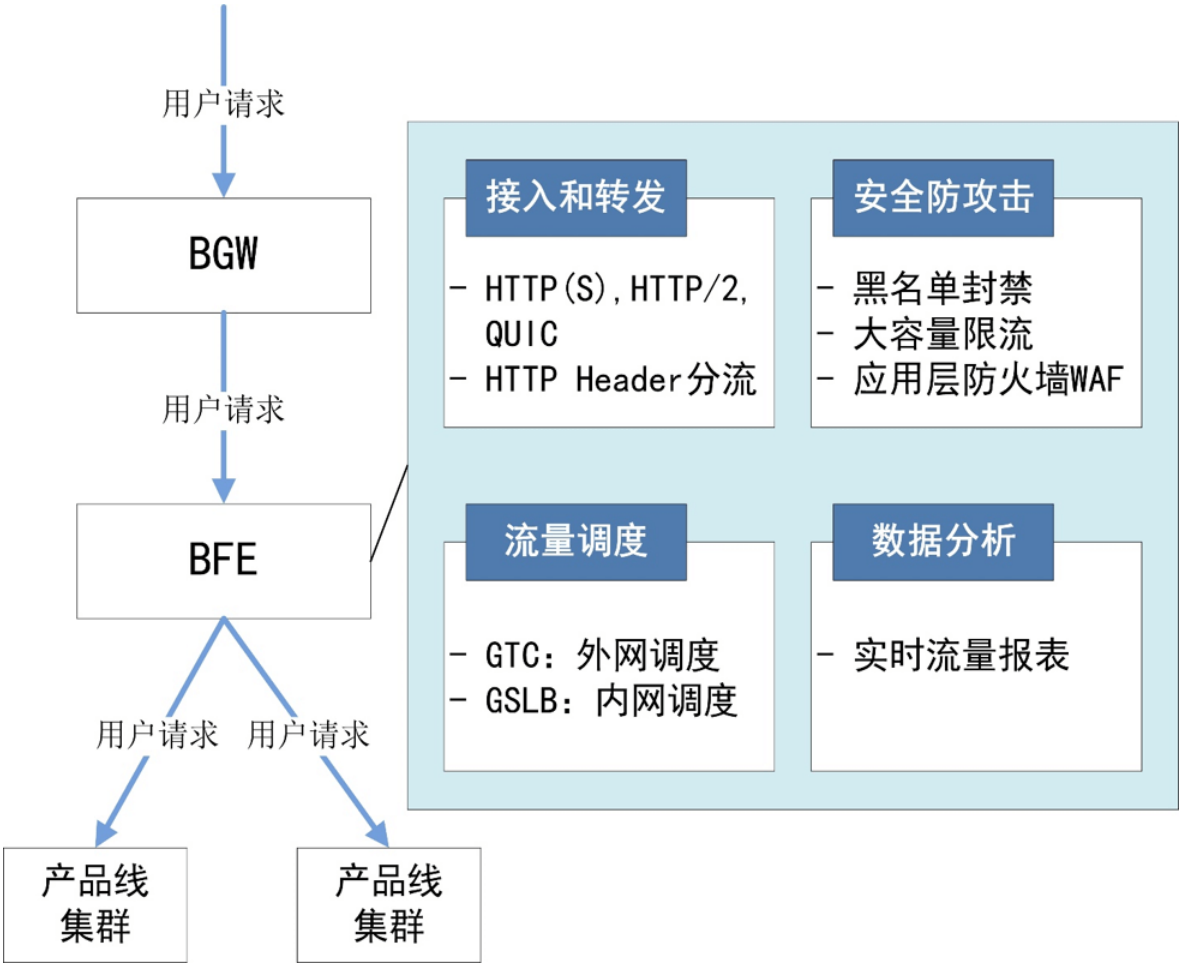
(2) 运维统一管理

对于需要普遍使用的安全策略，只需要在平台上统一上线即可。

(3) 增强流量控制能力

和四层负载均衡相比，七层负载均衡可以“看到”流量中更多的内容，可以在流量转发、安全、数据统计等方面提供更强的能力。

BFE平台的主要功能



作为一个综合的大型七层负载均衡流量转发平台，BFE平台包括以下4个主要功能：

(1) 接入和转发

BFE平台可以接收处理HTTP、HTTPS、HTTP/2、QUIC等多种协议的流量，并基于HTTP Header的信息做分流转发。

(2) 流量调度

BFE平台包括内网流量调度和外网流量调度所组成的两层流量调度体系。

(3) 安全防攻击

BFE平台支持多种安全能力，包括黑名单封禁、大容量限流、WAF（应用层防火墙）等。

(4) 数据分析

BFE平台可以基于转发日志生成实时报表，反映业务的流量变化情况及下游业务集群的健康状态（错误数、延迟等）

BFE开源项目

BFE平台的转发相关模块

BFE平台是由很多模块构成的一个比较复杂的分布式系统，可以分为数据平台和控制平台，及一些相关的依赖模块。

数据平面的主要模块包括：

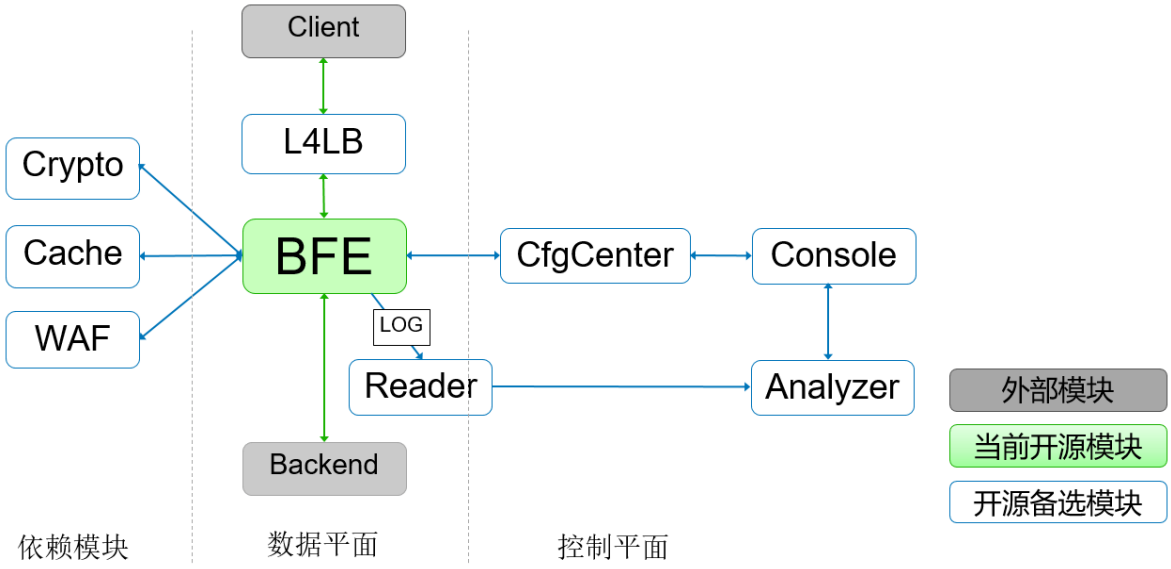
- 转发引擎（Forwarding Engine）：提供流量的基础转发功能
- 日志单机模块（Log Reader）：负责在单个节点上对转发日志进行处理

控制平面的主要模块包括：

- 日志分析模块（Log Analyzer）：在日志单机模块的基础上，对日志数据进行集群化集中处理
- 配置中心（Config Center）：集中管理BFE的转发配置
- 管理控制台（Console）：供人员管理配置，查看平台状态

和转发的相关依赖模块包括：

- 加解密模块（Crypto）：可以对HTTPS处理中的部分高消耗计算提供远程服务，从而降低对CPU资源的消耗
- 缓存模块（Cache）：提供转发所需要的缓存功能，如对HTTPS中所使用的session cache
- 应用层防火墙（WAF）：提供对恶意请求进行检查的功能



BFE开源项目的内容

目前BFE开源项目中的内容为BFE平台中的转发引擎，包括核心框架和大部分的功能，具体功能包括：

- 主流接入协议的支持

  包括：HTTP，HTTPS，SPDY，HTTP/2，Web Socket，TLS

- 灵活的模块框架

  在BFE开源项目中已经包含了很多扩展模块，也支持开发第三方扩展模块

- 基于请求内容的路由

  提供BFE专有的“条件表达式”（Condition Expression）能力，可以灵活的定制转发规则

- 多种负载均衡策略

支持“集群-子集群-实例”的多层级负载均衡能力，可以支持多机房、集群粒度的负载均衡，支持过载保护

- 丰富的监控探针

在BFE中内置了丰富详尽的监控指标，配合第三方监控系统（如：Zabbix，Prometheus）可以充分掌握系统的状态

以上这些内容，将在后面的章节中做详细的介绍。

原理篇

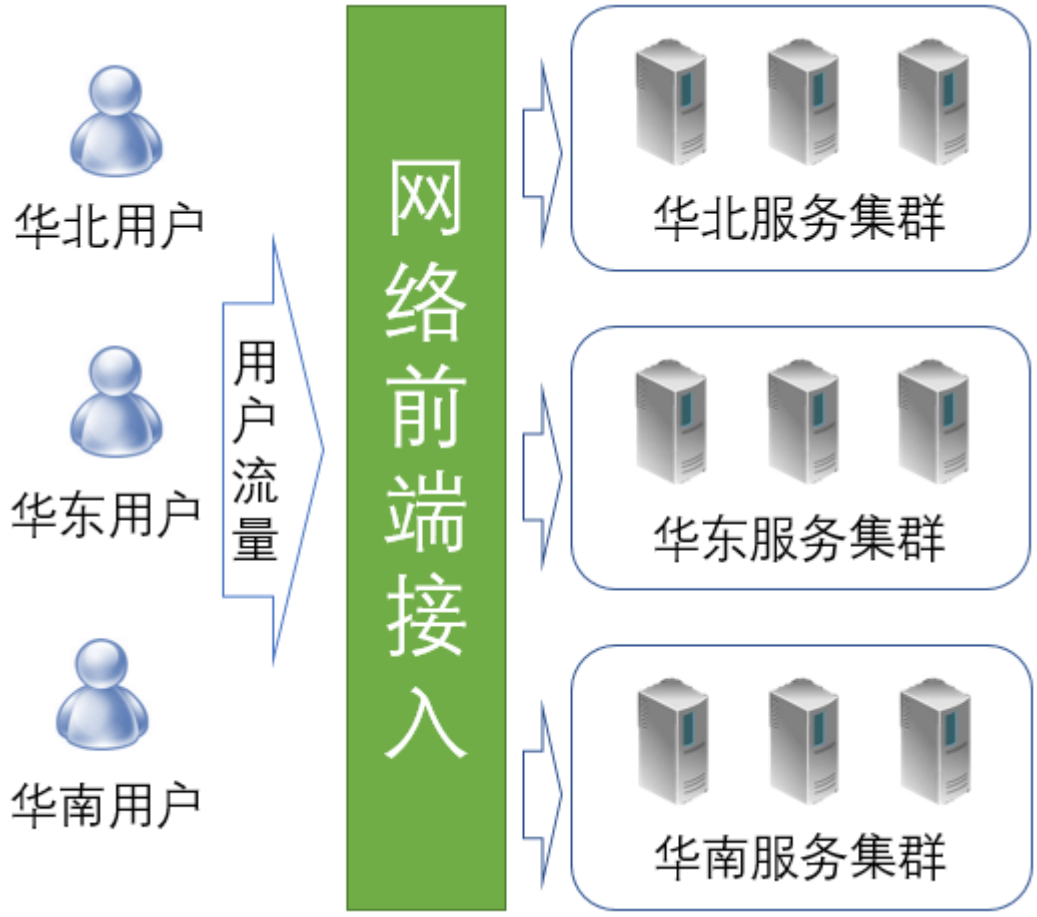
网络前端接入技术简介

什么是网络前端接入

“前端”（FE，Front End）这个词经常会用在区分软件工程师的角色：在浏览器上基于javacript、html等技术开发前端程序的工程师，常被称为“前端工程师”，或“FE工程师”；而在服务器上基于C++、Java、Go等编程语言开发后台程序的工程师，被称为“后端工程师”。

网络前端接入的英文也是Front End，缩写为FE。从事网络前端接入方向的工程师不是“前端工程师”，而是“网络研发工程师”。在这里，“前端”（Frond End）是从网络和用户访问的角度出发而产生的概念。“后端”（Back End）的服务位于数据中心中，是用户无法直接访问的；用户的流量必须要经过网络前端接入（也就是“前端”）的转发才能到达后端。我们也可以把网络前端接入定义为“流量从用户到达服务的过程”。

流量从用户到达数据中心内的服务并不是一个很简单的过程。流量从用户到达服务，要经过很多系统或设备的处理，其中包括家庭或公司的网络，也包括运营商的网络，还有服务提供商自己的网络。网络前端接入非常关键，如果这个环节出了问题，即使数据中心内的服务是正常的，用户仍然无法很好的访问服务。



网络前端接入面临的挑战

在网络前端接入中，面临以下几方面的挑战：

(1) 可用性

这方面可能出现的问题包括：

- 网络的故障。用户网络、运营商网络、服务提供商的网络都可能出现故障。
- 服务的故障。数据中心内部署的服务可能由于数据中心、服务器、后台程序的问题而无法正常提供服务。
- 网络攻击。黑客可能发动DDoS攻击或应用层攻击，从而导致网络带宽拥塞、相关服务系统过载或崩溃。

可用性是网络前端接入中最严重的问题，会导致用户根本无法正常访问服务，从而出现流量损失。

(2) 性能

这方面可能出现的问题包括：

- 低效的网络协议。网络协议对于传输性能有很大的影响。尤其是在移动无线互联网场景下，在延迟抖动和高丢包率的影响下，之前为有线网络所设计的网络协议很多都出现了性能方面的问题。这种情况也推动了近年来网络协议的快速升级。
- 不优化的调度。在大型企业存在多地域、多运营商接入点的情况下，如何将不同地域、运营商的用户调度到合理的网络接入点是非常重要的技术。如果没有实现就近的接入、或调度到了不同运营商的接入点，都可能导致网络访问性能的下降。

性能方面的问题，会导致用户访问服务的速度变慢，而访问速度会直接影响到互联网用户的访问体验。

(3) 安全

这方面可能出现的问题包括：

- 流量劫持。黑客可能通过DNS劫持等手段将用户流量引导到伪造的网站去。
- 内容劫持。黑客可能对未使用HTTPS技术加密的网页进行内容插入或修改。例如，可能对访问的内容中插入非网站主提供的广告内容，从而获取非法的经济利益。
- 隐私泄露。黑客可以对未使用HTTPS技术加密的访问进行嗅探。例如，可以对用户的访问情况进行监听收集，从而了解用户的兴趣，并将这些信息用于广告等目的。

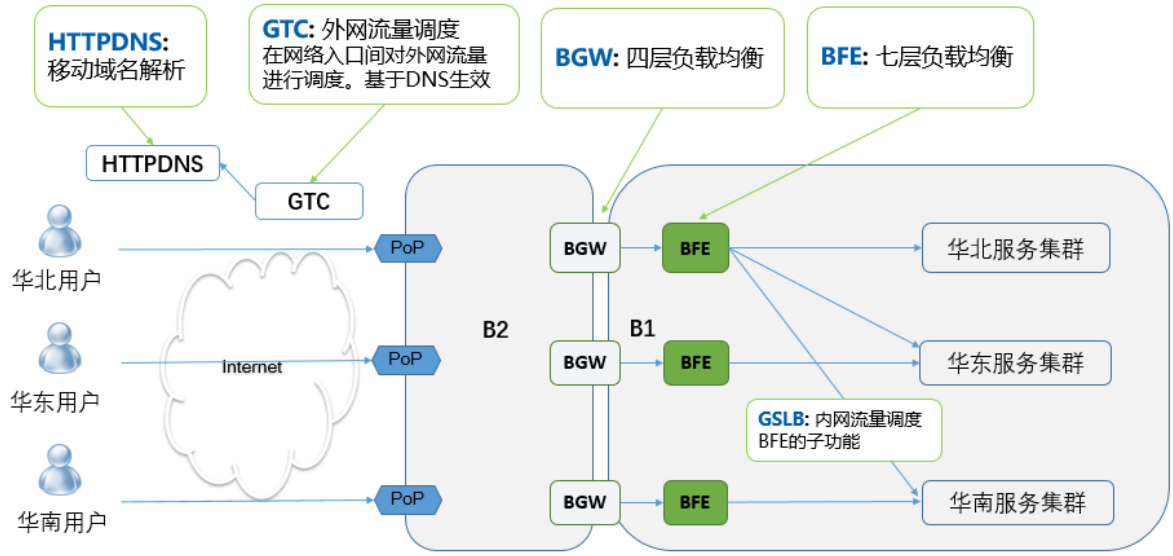
安全方面的问题对于互联网服务企业和互联网用户都造成了极大的威胁，可能造成巨大的经济损失。

(4) 效率

在多地域/数据中心/实例的场景下，不优化的调度可能导致地域/数据中心/实例级别的负载不均衡。

这会导致服务资源间忙闲不均，从而导致服务资源无法充分利用。

百度的网络前端接入方案



百度作为一个大型的互联网企业，具有多地域、多数据中心的复杂场景。其网络前端接入方案如上图所示，包括以下几个关键系统：

- GTC（Global Traffic Control）：外网流量调度系统。用于在网络入口间对外网流量进行调度。

  在网络流量调度方面，有两种可能的技术方案：DNS，或BGP路由。由于带宽资费等方面的原因，在国内普遍使用由运营商提供IP地址的“静态带宽”，而不是由网站服务商自己提供地址的“BGP”带宽，所以GTC也主要基于DNS来生效。

- HTTPDNS：移动域名解析。用于为移动客户端提供域名解析服务。

  DNS作为互联网的重要基础设置，一直存在容易被劫持、生效速度慢、解析准确性低等固有问题。随着移动互联网的发展，尤其是移动APP的广泛使用，为解决DNS的问题提供了新的机遇。HTTPDNS基于加密的Web服务，可以解决DNS所存在的一系列问题，已经在百度所有重要移动客户端上普遍使用。

- BGW（Baidu GateWay）：四层负载均衡系统。为流量提供网络负载均衡服务。

  BGW的功能类似于著名的开源软件LVS，但它是由百度基于DPDK技术自研的系统。

- BFE：七层负载均衡系统。为流量提供应用层负载均衡服务。

  内网流量调度GSLB作为BFE的子功能，提供跨数据中心的集群粒度的流量调度。

网络前端接入技术的发展趋势

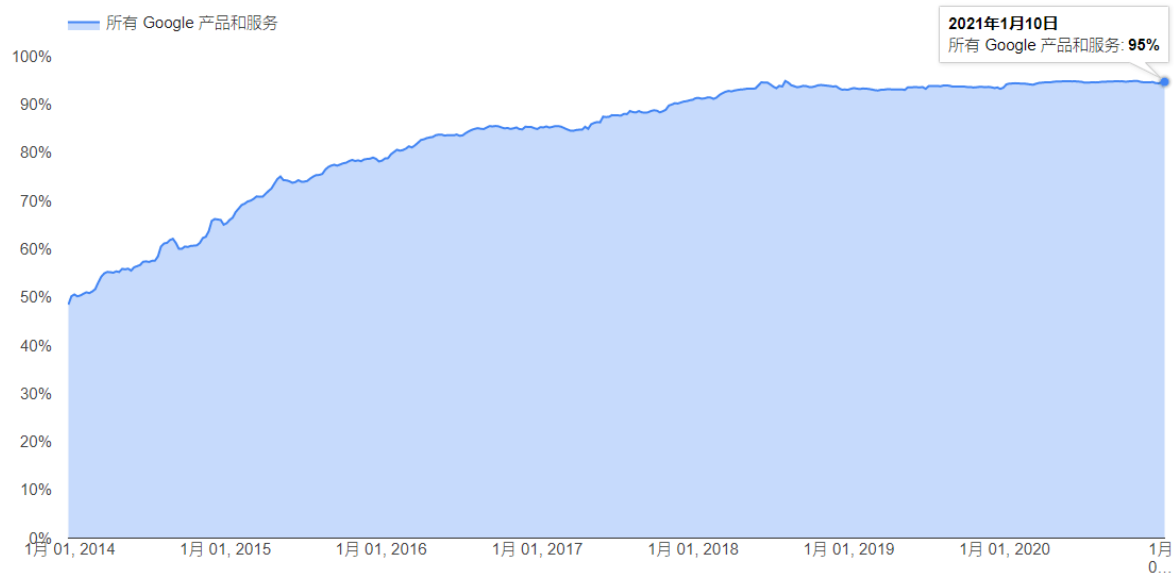
网络传输的加密化

作为一种对网络访问进行加密的技术，HTTPS早在2000年就已经在RFC2818中被定义。但是在之后的十多年时间中，HTTPS在业界并没有被普遍的使用。

随着近年来互联网在人类生活领域的快速渗透，互联网所承载的经济利益越来越高，HTTP明文传输所导致的安全风险越来越大。对于未使用HTTPS技术的网站，可能会存在以下威胁：

- 配合DNS劫持技术，网站可能被伪造。对于包含具有经济利益的账户的网站（如银行、电商），黑客可以通过这样的方式骗取到用户的账号名和密码。
- 用户获得的访问内容可能被篡改。例如，对于一些高质量内容的网站，黑客可能在网站主没有感知的情况下通过广告内容来获取非法利益。
- 用户的访问信息可能被嗅探。例如，通过嗅探用户对于搜索、电商等网站的访问情况，黑客可以将这些信息转卖给第三方公司用于广告投放的优化。

从保护搜索流量不被劫持和窃听的角度，谷歌从2014年起大力推动自身的HTTPS化改造，2014年至2020年谷歌的产品和服务HTTPS流量的覆盖率从50%提升至95%（见 https://transparencyreport.google.com/https/overview ）。同时，谷歌也通过Chrome浏览器的巨大影响力，推动第三方网站升级到HTTPS，对于未使用HTTPS的网站，在Chrome浏览器中会显示为“不安全”。几乎在同一时期，国内的各大互联网公司也都开始了全站HTTPS化改造的工作。



必须要说明，HTTPS化绝对不是“将HTTP访问加密”这么简单，HTTPS化给使用这个技术的公司带来了一系列的挑战：

- 对网络资源管理的要求提高

  在HTTPS化中涉及到HTTPS证书的问题，而证书和域名紧密相关。在HTTPS改造过程中，部分企业可能会发现之前无序申请的大量域名可能会在证书的使用和成本方面带来很大的问题。

- 对于外部的依赖增强

在使用HTTPS后，由于技术更加复杂，服务器一侧对于客户端的兼容性难度增加。在服务端程序升级时，有可能由于部分客户端程序/代码库不兼容导致服务访问的异常。另外，在使用HTTPS后，对于CA（Certificate Authority，证书颁发机构）产生了很大的依赖，CA成为网站稳定性、安全性的一个新的隐患。

- 对性能方面的影响

  在HTTPS中由于使用到加解密技术，肯定会对性能产生一定的影响。尤其是其中使用的非对称加解密计算，会消耗大量的CPU计算资源，从而导致网站成本的提升，并增加了DDoS攻击的风险。

- 对延迟方面的影响

  和HTTP相比，HTTPS会增加1到2轮的协议报文交互，从而增加用户的访问延迟。对于用户和网站之间本来网络延迟就比较大的场景，HTTPS会加倍的放大这种延迟。

对于以上这些挑战，都需要在技术和机制上采用一些措施来解决。

网络协议的技术门槛提高

在大约20年前，搭建一个网站可能是一个相对简单的事情。HTTP就是主流的网络协议。一个对网络可能没有太多了解的技术人员，根据一些简单的说明，下载一个Apache软件（后来是Nginx），安装并运行就可以了。

最近几年来，随着安全技术的升级、及针对移动互联网的网络协议的升级，HTTPS、SPDY、HTTP/2、QUIC等协议不断的出现、或被广泛的使用。相比HTTP协议，这些协议的复杂性要提高很多，已经不是任何人都可以快速掌握。如果对于相关协议技术没有深入的了解，可能会在安全、稳定性、性能等方面出现很多问题。这种情况推动了网络前端接入人员的专业化，在很多公司、尤其是规模较大的互联网公司，都建立专门的网络前端接入团队。

移动化对网络前端接入的影响

网络访问的移动化体现在两个方面。首先，从网络传输的介质来说，已经有相当大比例的流量是通过无线互联网（Wifi，3G/4G/5G）来访问的。另外一个方面，相当大比例的用户是通过移动终端/移动客户端来访问，而不是通过PC。早在2014年10月，百度搜索在移动客户端上的流量就超过了PC。

网络访问的移动化，对网络前端接入产生了重要的影响，表现在：

- 传输协议的快速升级

  互联网最经常使用的TCP协议，其设计中包含的很多重要假设都是基于有线网络的。例如：报文的丢失，主要是由于网络中发生了拥塞，所以报文丢失可以作为拥塞发生的信号；端到端的延迟和带宽在一定时间内是相对稳定的，通过超时时钟的机制来发现丢失。在无线互联网场景下，以上的假设很多被打破了。由于无线链路的特点，很多报表的丢失并不是由于拥塞；端到端的延迟和带宽也是不稳定的。这种变化引发了对互联网协议的升级。

- 传输协议的私有化

  众所周知，TCP位于操作系统的网络协议栈中。全世界只有3家公司可以对移动端的操作系统进行修改：谷歌（Android），苹果（iOS，Mac OS），微软（Windows）。对其它公司，在客户端一侧对TCP协议进行持续优化的可能性很低，之前的大量优化都只能从服务端想办法。移动端、尤其是Native App的广泛使用改变了这种情况。QUIC协议就是一个例子，它基于UDP协议，运行在APP中，任何公司都可以根据自己的需要对其进行修改和优化

- HTTPDNS的兴起

DNS是互联网的3大基础机制之一（另外2个分别是IP路由和TCP拥塞控制）。DNS容易被劫持、生效速度慢、解析准确性低等固有问题，在移动互联网时代迎来了解决的机会。

网络安全防护的重要性提高

由于互联网服务所具有的经济利益，也引来了各种攻击，如通过大压力使得网站无法服务的DDoS攻击，或利用网站的各种漏洞所实施的应用层攻击。网络攻击是被业务的价值所吸引的，公司的营收越大，受到攻击的可能性就越大。如果没有足够的安全防护能力，想平平安安的用互联网来从事经济活动或客户服务是不可能的。我们经常可以听到类似这样的新闻：某个游戏公司的游戏做的很好，玩家非常喜欢，用户量不断上升，但突然被恶意DDoS攻击，使正常的用户无法使用，导致用户大量流失，甚至公司倒闭；某个公司由于存在漏洞，被黑客从数据库中窃取隐私的客户信息，从而影响到公司的市场声誉。

对于一般的公司来说，DDoS防护和WAF防护等安全机制已经成为普遍需要的基础能力。

在DDoS攻击的防御方面，由两种可能的思路：

- 对攻击流量进行过滤

  这适用于攻击流量小于带宽容量，或攻击流量小于负载均衡系统的场景

- 通过调度进行应对

  当攻击流量大于带宽容量或负载均衡系统时，需要将流量调度到其它空闲的网络接入点

在应用层攻击防御方面，重点在于以下两点：

- 检查规则的完备程度

  WAF的有效性，对检查规则的依赖很高。0 Day场景仍然是对WAF的巨大挑战。

- 执行检查规则所需要的计算资源

  检查所需要的资源，随着规则数的增加而增长。在资源消耗和安全性之间，存在着权衡。

对攻击的防御，本质上是一种基于资源（包括带宽、计算能力等）的对抗。防攻击的能力，部分决定于资源动员的能力。在被攻击的时候，是否可以灵活调用所有的带宽资源、服务器资源参与对抗，这决定了攻击防御的结果。建设强大的资源调度能力，对于提升攻击的防御能力非常重要。

数据驱动的运营

在互联网最初出现的时候，有一种说法：互联网是尽力而为（Best Effort），和电信网络相比，无法保证非常高的稳定性。但是，目前很多互联网服务已经成为生活所必须的基础服务，重要性类似于水和电。这就对互联网服务的稳定性提出了极高的要求。

对应于网络前端接入来说，以前可能对于服务、流量、故障的情况并没有持续和精确的数据报表及监控。在很多公司，对于公司整体的每秒请求数、并发连接数、HTTPS握手成功率等是缺乏准确数据的，更不要说按照服务、域名、地域等维度对这些数据进行深入分析。

实现提升服务可靠性的目标（如，从99.9%提升到99.999%），肯定无法仅依靠人工运营的方式，而必须依靠技术手段。转向基于数据的运营是必由之路。前端接入网络数据方面的建设工作包括：

- 建立相关的报表体系
- 建立关键的监控
- 为自动化/智能化控制提供所需要的数据

从数据内容的角度，可以考虑在以下3方面建立报表和监控体系：

- 使用网络前端接入的业务的用户流量数据
- 各服务的后端服务的状况

  网络前端接入作为调用方，可以发现后端服务在延迟、错误率等方面的异常
- 外部网络的状况

  对于网络前端接入来说，外部网络是最难以控制的。需要尽可能的对Local DNS的解析情况、各地区到接入点的连通性等情况进行监控。

自动化/智能化的控制

前端网络接入中，有很多需要决策的地方，如：流量的调度，止损的处理等等。在很多公司里，这些问题仍然是依靠人工来处理的。

对于从人工升级到自动控制的目的，很多人的认识仍然停留在“节省人力成本”。其实，这还不是最重要的。在提升服务的可靠性方面，很多自动控制的能力是人工根本无法实现的。比如：在外网流量调度方面，百度的自动调度程序可以持续根据带宽资源、服务容量、网络连通性等信息，持续进行计算；在出现故障的情况下，在2分钟内实现自动的切换。这样的效果是人工完全无法完成的（即使一个人可以7*24小时的工作）。

在自动化控制角度，可以优化的方向包括：

- 将以前靠人来执行的策略，固化为系统中的策略，从而减少对个人的依赖
- 在建立模型的基础上，不断的优化模型和策略
- 从定性的控制，升级为定量的控制。在这方面，人工是很难做到的

这里必须要说明，实现自动化的控制并不是那么容易的，对于系统设计提出了很高的要求。和之前基于人工控制的系统相比，自动控制的系统有两个基本的前提：

- 清晰的模型

  自动化系统的难点不是系统和编码，而是模型的设计和优化。模型设计的质量，决定了系统最终的质量。而对很多软件工程师来说，模型的设计能力是一个亟待提高的短板。
- 完备、可靠、可量化的数据

  控制算法的执行效果，严重依赖于所输入的数据。用于控制的数据，可靠性要求远高于报表数据。如果数据系统出现了故障，也会导致最终控制的失败。在自动控制的系统设计中，需要在数据采集和控制方面做出很多容错的设计。

网络前端接入的软件化/服务化/开源化

之前大部分的网络前端接入功能都是由硬件设备提供的。使用者需要购置相关的设备，部署在自己的数据中心中。

最近这些年来，这方面已经发生了很大的变化，表现为：

- 软件化

基于标准服务器部署软件来实现网络前端接入的功能（如，负载均衡，DNS）。这不仅可降低设备采购成本，而且增强了这些功能的弹性扩缩容能力。

- 服务化

  对于某些功能，甚至不需要部署软件，而可以直接使用服务。包括公有云中的各种负载均衡服务、DNS，及CDN、网络代理接入等，都已经有服务可以购买。

- 开源化

  在云计算领域，开源是重要的驱动力，在网络前端接入领域也是如此。开源增强了使用者对于软件的控制力，也增强了软件的进化能力。

网络前端接入的云原生化

云原生（Cloud Native）是目前云计算的重要方向。

网络前端接入系统一方面要支持业务的云原生化，支持业务的微服务化、多租户、弹性扩缩容等能力；另外也要实现自身的云原生化，自身具备微服务化、弹性扩缩容的特性。

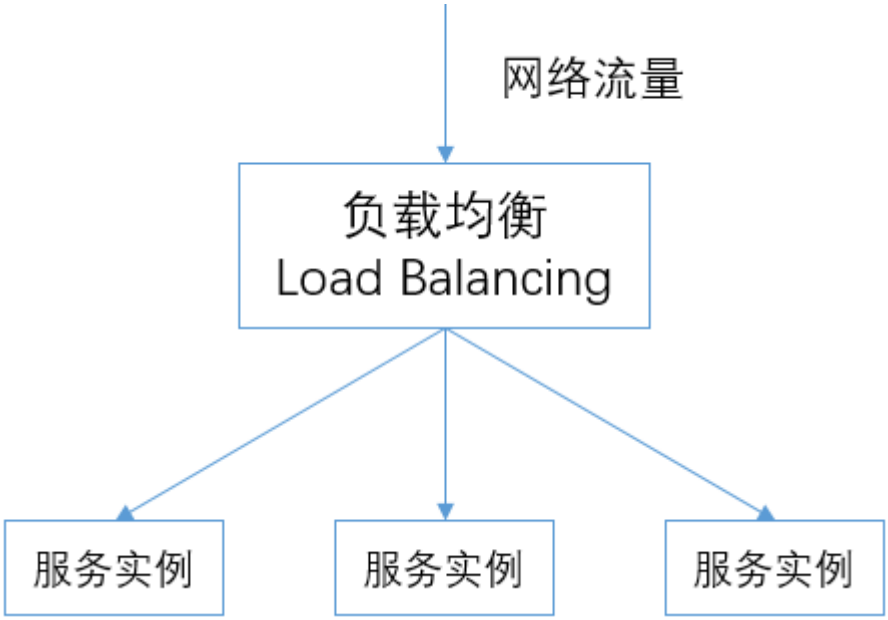
网络负载均衡技术简介

前面从“网络前端接入”的角度，介绍了BFE的背景。从系统分类的角度，BFE也属于“网络负载均衡”的范畴。为了便于大家理解，下面对网络负载均衡技术做一个简要的介绍。

概念

负载均衡，英文为Load Balancing，在计算机领域是指将一组任务分布到一组计算单元上来处理。负载均衡技术在分布式计算的场景有广泛的使用，只要计算单元大于一个，都需要考虑使用负载均衡技术。

在网络领域，负载均衡器负责将网络流量分发给下游的多个实例。



网络均衡器 vs 名字服务

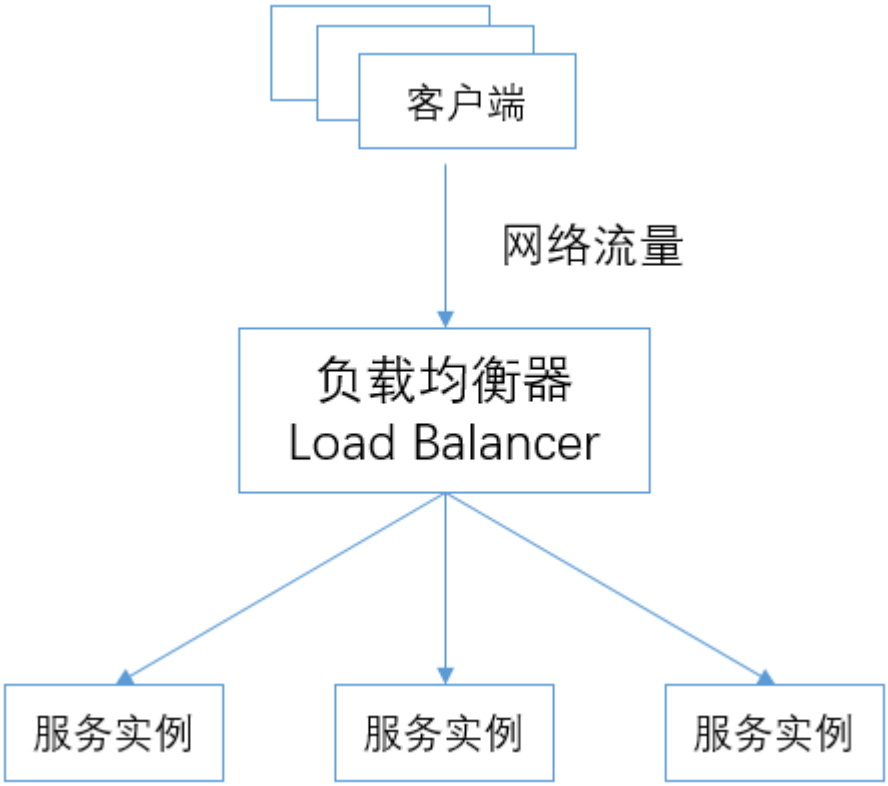
机制说明

网络负载均衡功能的实现，可以基于两种方式：

- 负载均衡器的方式

  所有的网络流量都先发送给负载均衡器（Load Balancer），再由负载均衡器转发给下游的服务实例。

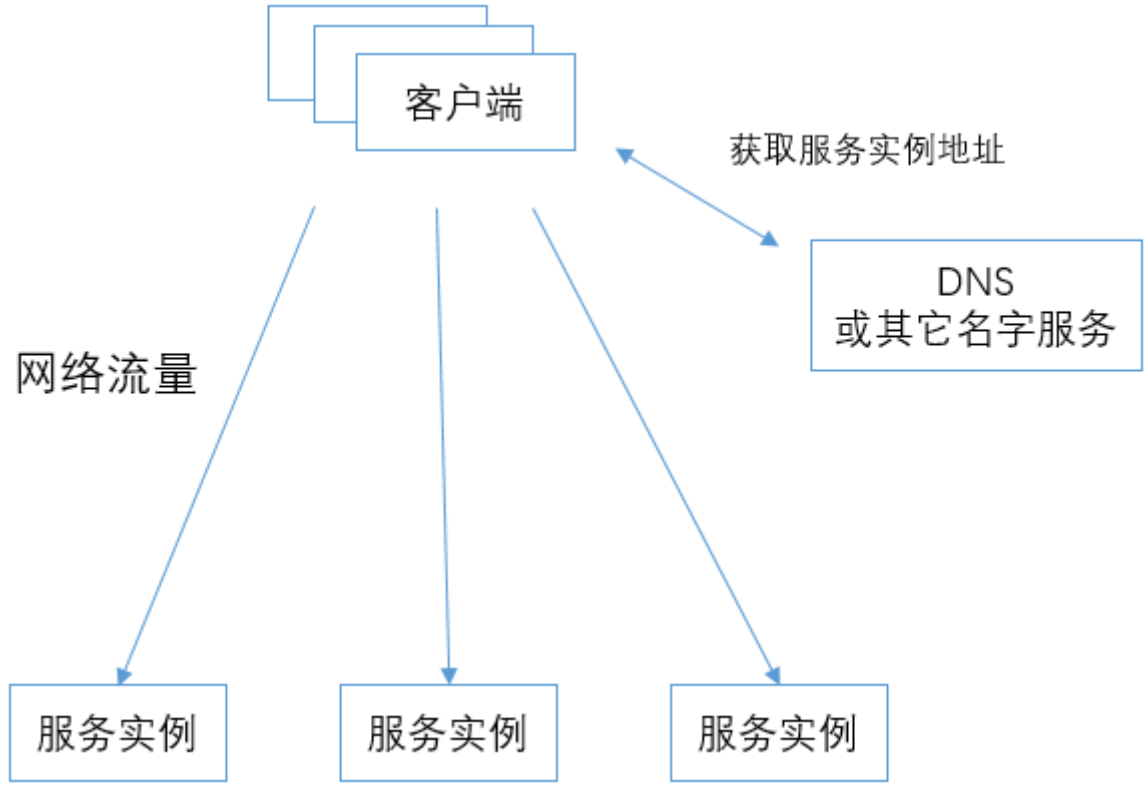
  在这种方式下，负载均衡的功能都由负载均衡器来完成。



- 名字服务 + 客户端策略的方式

  客户端通过DNS（或其它名字服务）获得服务实例的地址列表，然后客户端把网络流量直接发送给服务实例。

  在这种方式下，负载均衡的功能都由客户端和名字服务配合完成。名字服务在返回服务实例地址列表时，有一定的策略；客户端在获得服务实例的地址列表后，在发送网络流量时也可以有一定的策略。



方案对比

下表中对以上两种方案进行了对比。

基于负载均衡器的方式适用于对流量控制要求比较高的场景，可以实现单个连接/请求粒度的精细转发控制。而且这种方式对客户端的要求很低，客户端不需要实现任何策略，也不涉及客户端SDK的引入。这种方式的弊端是需要在负载均衡器方面投入额外的资源，在使用时需要计算一下所需要的资源成本。

基于名字服务+客户端策略的方式的好处是不需要额外的资源消耗，客户端直接访问服务。但是这种方式的执行效果强依赖于客户端的配合，在客户端上要实现较复杂的策略，通常要引入客户端SDK，从而引入了客户端升级的成本。对很多公司来说，客户端软件的种类很多，分属不同的团队，要做到及时的升级很困难。另外，这种方式的执行粒度比较粗，客户端和名字服务之间的交互不可能过于频繁，秒级的交互频度已经是很高的了，但即使这样也不可能做到单个连接/请求粒度的控制。

在某些场景下，无法使用负载均衡器，而只能使用名字服务+客户端策略的方式。如，在外网调度场景下，就只能使用DNS的方式。

方案	对流量的控制力	资源消耗	对客户端的要求	适用场景
基于负载均衡器	**强**。 可以达到单个连接 / 请求的粒度。	**高**。 负载均衡器引入了额外的资源消耗。	**低**。 客户端基本不需要实现策略。	总体流量规模不大（从负载均衡器资消耗的角度）； 应用场景对流量控要求高。
基于名字服务 + 客户端策略	**弱**。 客户端直接访问服务， 没有可靠的卡控点。	**低**。 不需要额外的资源	**高**。 客户端需要支持比较复杂的策略，	总体流量规模较大； 应用场景对流量控要求低； 无法使用负载均衡

方案	无法实现精细的流量控制测量。	投入。资源消耗	且涉及升级的问对客户端的要求题。	适场景

四层负载均衡 vs 七层负载均衡

传统的硬件网络负载均衡器中，包含了比较综合的功能。从协议的角度，包含了对TCP、UDP流量的支持，也包含了对HTTP、HTTPS等协议的处理。

而在大多数互联网公司中，普遍使用软件形态的负载均衡器，并且基于所处理的协议区分为两种不同的系统：

- 四层负载均衡：仅用于对TCP、UDP流量进行处理。也被称为网络负载均衡。

  四层负载均衡在转发中主要基于IP地址、端口等信息。

  四层负载均衡的开源软件包括LVS，DPVS等。

- 七层负载均衡：支持HTTP、HTTPS、SSL、TLS等协议的处理。也被称为应用负载均衡。

  七层负载均衡在转发中可以利用应用层的信息（如：HTTP的请求头部），而这些信息对四层负载均衡来说是不可见的。

  七层负载均衡的开源软件包括Nginx, BFE, Traefik, Envoy等。

有一些软件同时支持了四层负载均衡和七层负载均衡，如Nginx和Haproxy。但是在大规模部署场景，一般会使用专用的四层负载均衡软件，主要原因是四层负载均衡和七层负载均衡两个场景存在较大的差异，适合使用不同的技术栈来实现。

- 四层负载均衡软件：需要很高的处理能力，以实现较高的性价比，并用于抵御来自外网的DDoS攻击。

  在“网络前端接入技术简介”中所介绍的BGW软件，它使用C语言，基于DPDK技术研发，在单台x86服务器上可以处理50Gbps的网络流量，每秒可以处理的新建连接超过100万。要获得更高的性能，就需要软件整体的逻辑比较简单，没有高资源消耗的功能。同时，因为性能和稳定性方面的高要求，这样的软件新功能研发的成本也比较高，开发新功能及变更上线的周期比较长。

- 七层负载均衡软件：功能比较复杂，且需要不断增加新的功能。

  七层负载均衡软件可以“看到”应用层的信息，功能的空间比四层负载均衡软件要大很多。由于它更贴近业务，也会不断收到来自业务的需求，需要不断的开发出新的功能特性；由于互联网业务的特性，需要更快的开发和上线速度。七层负载均衡软件对于性能方面的要求比四层负载均衡软件要低，从带宽吞吐方面几乎差距一个量级。

下图以百度的BGW和BFE为例，说明了四层负载均衡软件和七层负载均衡软件混合使用的场景。

- BGW和交换机的互连

  BGW通过BGP或OSPF路由协议和上游的交换机进行路由交互。交换机使用ECMP（等价路由）的机制，将流量哈希分发到多个BGW实例。

  对于某个VS（Virtual Server，由（IP地址，协议，端口）来标示）来说，所有的BGW实例都可以接收和处理其流量。通过这种方式，实现了BGW的分布式容错，在单个BGW实例故障的情况下，BGW集群仍能够继续处理流量转发。

- BGW和下游服务的互连

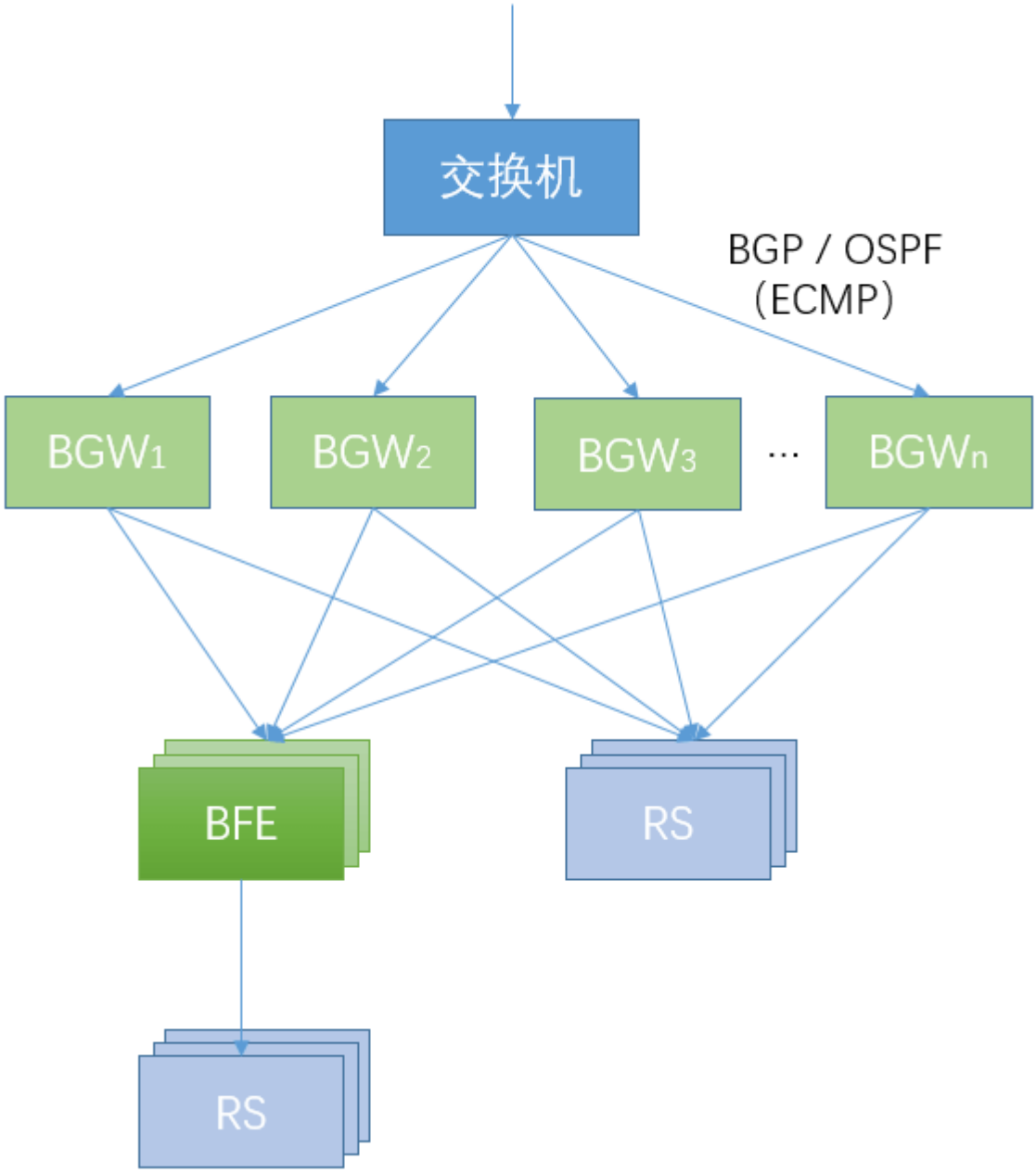
  每个BGW实例按照Round Robin(或设定的其它策略)，将流量转发给下游的RS（Real Server，和VS相对应）。

- BGW和BFE的互连

  在使用七层负载均衡的场景，BGW把BFE当作RS，将发往某个VS的流量转发给下游的BFE实例。在BFE实例出现故障的情况下，BGW可以将有故障的BFE实例自动摘除。

- BFE和下游服务的互连

  每个BFE实例按照Round Robin(或设定的其它策略)，将流量转发给下游的RS。



设计篇

BFE的设计思想

BFE引擎重构的缘起

百度的BFE转发平台最早于2012初上线使用。那时所基于的转发引擎是一个名为Transmit的内部系统。Transmit基于C语言实现，是多进程+libevent的模型。

到2013年底的时候，有了要重构转发引擎的想法。主要的驱动力如下：

- 平台化的需要

  BFE平台在上线初期，只有十多个业务使用。到2013年底的时候，已经有几十个业务了。原有的系统中，没有多租户机制，不易做多业务的配置管理。另外，配置的格式是非结构化的，不易使用程序来生成和处理；原来的系统在配置热加载方面的机制也比较复杂。

- 网络协议栈的维护成本

  Transmit中的HTTP协议栈是百度自研的，在使用过程中发现了一些协议一致性方面的细节问题，维护成本较高。另外，2013年底，百度已经启动了对于HTTPS的调研，需要在转发引擎上增加对HTTPS的支持。网络协议栈是反向代理系统的重要模块。从长期来看，维护完全自研的网络协议栈成本很高。

- 状态监控能力欠缺

  对于一个工业级水平的转发系统来说，需要有很强的状态监控能力。Transmit原有的监控信息较少，增加新的监控状态也比较困难。

- 转发配置的维护难度较高

  Transmit的转发配置主要使用正则表达式来描述。在实践中，发现正则表达式存在可维护性方面的问题。

2014年初，确定基于Go语言来重构BFE转发引擎。2014年4月份，开始编写代码，2014年底完成开发，2015年初Go版本的转发引擎在百度完成全量上线。

BFE为什么要基于Go语言

在2014年初对BFE重构做技术选型时，曾经考虑过两种技术路线：

- 基于Nginx。这是业界普遍使用的方案，绝大多数企业的七层负载均衡是基于Nginx搭建的。
- 基于Go语言。

回到2014年，Go语言在国内的使用案例还比较少。这么多年来，不断的有人在问，你们为什么要选用Go语言。下面是当时的一些考虑：

- 研发效率

  对C和Python都使用过的人应该有这样的体会，Python的研发效率要远高于C。我们的一个基本判断是，在未来很多年内，七层负载均衡仍然有很多功能需要开发。Go语言的研发效率接近Python，这对于快速交付功能提供了非常大的优势。

- 稳定性

  负载均衡的稳定性要求很高。如果负载均衡转发引擎崩溃了，无论数据中心内其它服务的稳定性由多高，用户根本无法访问服务。对于使用C语言研发的系统来说，内存访问错误占据非常高的比例，部分错误可以直接导致系统的崩溃；而且C语言对于错误缺乏保护机制。而对于Go语言来说，内存的回收是系统负责，无需开发者关心，这大大降低了问题发生的概率；另外，在Go语言中可以使用Recover机制来捕捉可能发现的Panic。

- 安全性

  从理论上讲，C语言编写的程序都具有缓冲区溢出的隐患，而这是很多恶意攻击可以成功的基础。Go的内存管理机制使得缓冲区溢出方面的安全风险大大降低。

- 代码可维护性

  Go语言相对于C语言（及Nginx中常用的Lua语言），代码的可读性和可维护性都更好。另外，在编写高并发程序方面，Go Routine使得可以类似多线程的模型来编写程序，而不需要设计复杂的状态机。这使得编写程序的难度降低。

- 网络协议栈

  对于一个负载均衡软件来说，网络协议栈是重要的考虑因素。BFE利用了Go系统库中成熟稳定的网络协议栈，这来源于谷歌在网络协议栈方面的强大实力。近年来的很多网络协议栈升级都由谷歌发起，如HTTP/2、QUIC。Go系统库中也会很快提供新协议的支持。

从实践来看，2014年所做的选择是非常正确的。基于Go语言重构的BFE引擎及时响应了百度内部业务对七层负载均衡的各种需求，并且长期保持稳定。自2015年初全量上线以来，BFE引擎从来没有在线上环境中发生过crash。

当然，Go语言也有它的短板。和Nginx相比，基于Go实现的BFE引擎性能要差一些。这种性能方面的差距主要来自于两方面：

- BFE没有在内存拷贝方面做极致的优化

  Nginx在内存拷贝方面做了端到端的极致优化，而内存拷贝是性能消耗的主要来源之一。出于对网络协议栈一致性方面的考虑，BFE尽量保持Go系统库网络协议栈实现的原貌，所以在内存拷贝方面多了一些消耗。

- BFE无法利用CPU Affinity

  Nginx可以通过“绑定CPU”的方式来减少进程切换代理的性能损耗。对于BFE来说，开发者只能控制Go Routine，底层的线程是被系统所控制的，无法利用CPU Affinity来优化性能。

这里还有一点要重点说明的是Go语言的GC（Garbage Collection，垃圾回收）延迟对于BFE研发的影响。在2014年时Go版本为1.3，GC延迟的问题非常严重，BFE的实测效果，GC延迟达到了400ms，完全无法接受。为此，当时在BFE中引入了“多进程轮转”的机制，以降低GC延迟对于转发流量的影响（这个机制的详情见附1）。GC延迟的问题在2017年初发布的Go 1.8中有了较好的解决，大部分的GC延迟都降低在1ms内，可以满足业务的要求，于是在2017年从BFE中去掉了多进程轮转机制。

BFE引擎的主要设计思想

基于Go来重写BFE转发引擎，绝对不仅仅是换了一种编程语言。在新版BFE中，有以下设计考虑：

- 对转发模型做了较大的修改

在引擎中明确引入了租户概念，可以基于hostname来区分租户（在各功能模块的配置中，也引入了对租户的区分）。另外，基于之前所发现的正则表达式的问题，尽量减少正则表达式的使用，为此设计了“条件表达式”（Condition Expression）机制。

- 降低动态配置加载的难度

  配置的动态加载，这是负载均衡软件的一个重要需求。除了升级可执行程序的场景，软件应可以持续运行，以保证流量转发的持续性。新版的BFE区将配置分为“常规配置”和“动态配置”：常规配置仅在程序启动时生效；动态配置可在程序执行过程中动态加载。动态配置统一使用JSON格式，兼顾了程序读取和人工阅读的需求。另外，系统提供了统一的动态加载机制，在实现新的模块时可以直接使用。

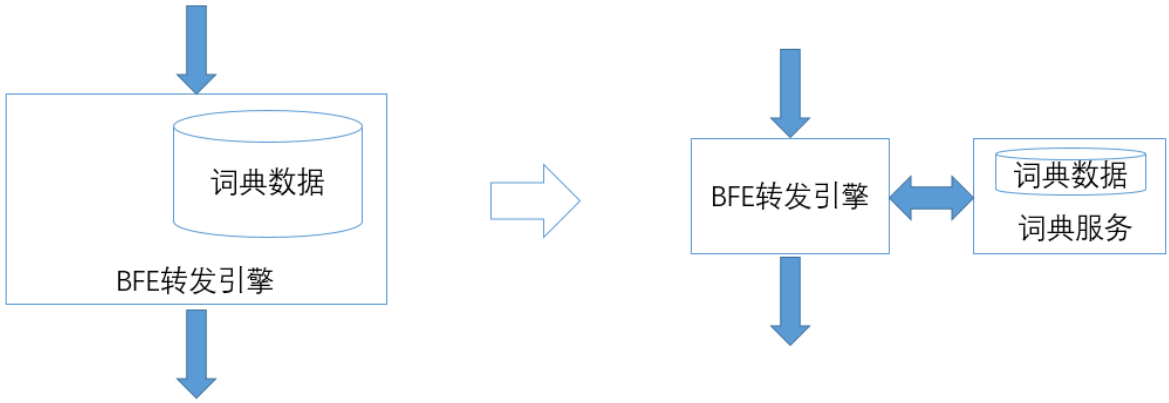
- 增强服务状态监控能力

  在重构BFE时，同时编写了web-monitor框架：每个BFE运行实例可以通过独立的HTTP服务向外展现内部的执行状态；增加新的内部状态非常简单，只需要一行代码。

- 将大存储功能转移到外部

  在原有的实现中，类似“词典查找”这样的功能也包含在BFE内部。这样的模块在启动时，需要使用较长的时间来加载词典数据，不利于BFE程序的快速启动。而BFE程序的快速启动能力，对于系统的稳定性至关重要。在发生故障的时候，一个需要几分钟才能启动的程序，其故障的恢复时间要长的多。为此，在重构BFE时将词典查找功能改写为独立的“词典服务”，由BFE远程调用。这保证了BFE可以在数秒内完成重启。

以上这些内容将在后续中的章节中给出详细的说明。



BFE和相关开源项目的对比

下面将BFE和一些相关的开源项目（Nginx/Envoy/Traefik）进行对比。需要说明的是，由于这些项目在活跃开发中，信息可能过期或有误。

项目的定位

在各开源项目的官网上对其定位描述如下：

- BFE: BFE是一个开源的七层负载均衡系统。
- Nginx: Nginx是HTTP服务、反向代理服务、邮件代理服务、通用TCP/UDP代理服务。
- Envoy: Envoy是开源的边缘和服务代理，为云原生应用而设计。
- Traefik: Traefik是先进的HTTP反向代理和负载均衡。

功能对比

协议支持

- 这四个系统都支持HTTPS和HTTP/2, 并计划或正在开发支持HTTP/3

健康检查

- BFE和Nginx只支持“被动”模式的健康检查。
- Envoy支持主动、被动和混合模式的健康检查。
- Traefik只支持“主动”模式的健康检查。

注：Nginx商业版支持“主动”模式的健康检查。

实例级别负载均衡

- 这四个系统都支持实例级别负载均衡

集群级别负载均衡

- BFE、Enovy、Traefik都支持集群级别负载均衡
- Nginx不支持集群级别负载均衡

注：Envoy基于全局及分布式负载均衡策略

对于转发规则的描述方式

- BFE基于条件表达式
- Nginx基于正则表达式
- Envoy支持基于域名、Path及Header的转发规则
- Traefik支持基于请求内容的分流，但无法支持灵活的与或非逻辑

扩展开发能力

编程语言

- BFE和Traefik都基于Go语言
- Nginx使用C和Lua开发
- Envoy使用C++开发

可插拔架构

- 这4个系统都使用了可插拔架构

新功能开发成本

由于编程语言方面的差异，BFE和Traefik的开发成本较低，Nginx和Envoy的开发成本较高。

异常处理能力

由于编程语言方面的差异，BFE和Traefik可以对异常（在Go语言中称为Panic）进行捕获处理，从而避免程序的异常结束；而Nginx和Envoy无法对内存等方面的错误进行捕获，这些错误很容易导致程序崩溃。

可运维性

内部状态展示

- BFE对程序内部状态，提供了丰富的展示
- Nginx和Traefik提供的内部状态信息较少
- Envoy也提供了丰富的内部状态展示

配置热加载

- 4个系统都提供配置热加载功能
- Nginx配置生效需重启进程，中断活跃长连接

注：Nginx商业版支持动态配置，在不重启进程的情况下热加载配置生效

BFE的转发模型

对于一个七层负载均衡软件来说，转发模型是它的核心。本章首先对BFE的转发模型做一个概要的介绍。在后面的章节中，会详细介绍BFE的路由转发机制和内网流量调度机制。

基本概念

在BFE中，有以下基本概念：

- 租户（Tenant）

  使用BFE转发的业务，可以基于“租户”的单位来区分。BFE引擎中的配置，比如转发策略、各扩展模块的配置等，都是以租户为单位来区分的。

  由于历史原因，在BFE中，租户也被称为“产品线”（Product）。

- 集群（Cluster）

  具有同类功能的后端定义为一个集群(Cluster)。对于一个租户，可以定义多个集群。在某些场景，集群也被称为服务（Service）。

  在一个租户内，可以使用租户的路由转发表将流量转发给合适的集群。详细的机制可以参考后面章节中关于BFE路由转发机制的说明。

- 子集群（Sub Cluster）

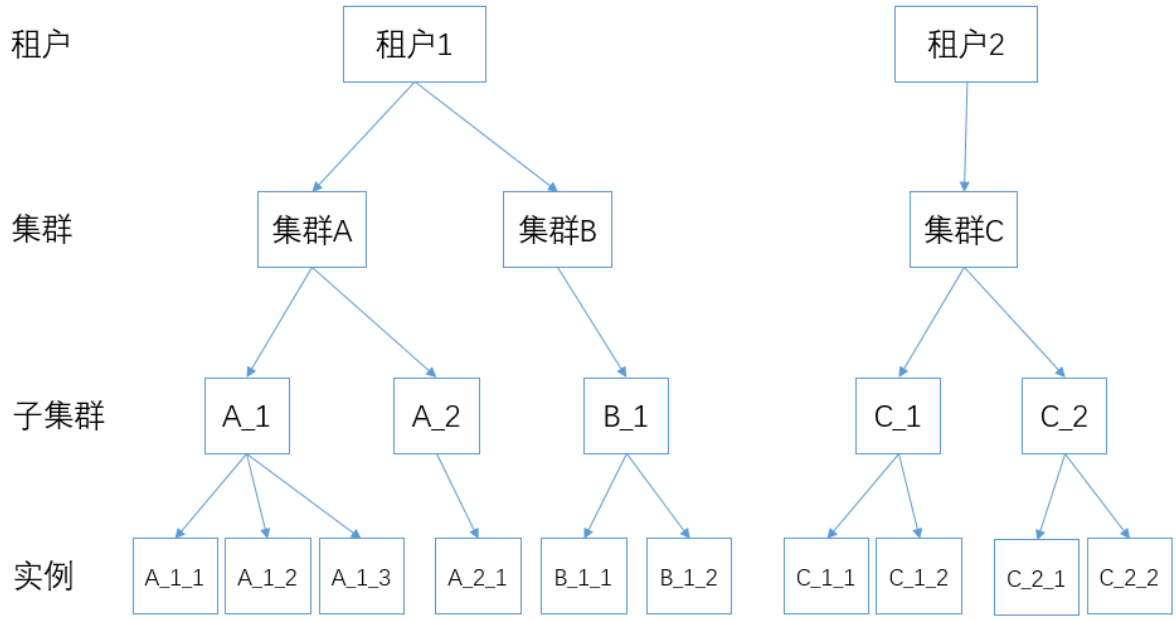
  在多数据中心场景下，集群可以划分为多个子集群(Sub Cluster)。通常，可以将集群中处于同一IDC（Internet Data Center）中的后端定义为一个子集群。在某些场景，子集群也被称为实例组（Instance Group）。

  子集群概念的引入，主要是为了处理多数据中心场景下的流量调度。详细的机制可以参考后面章节中关于BFE内网流量调度机制的说明。

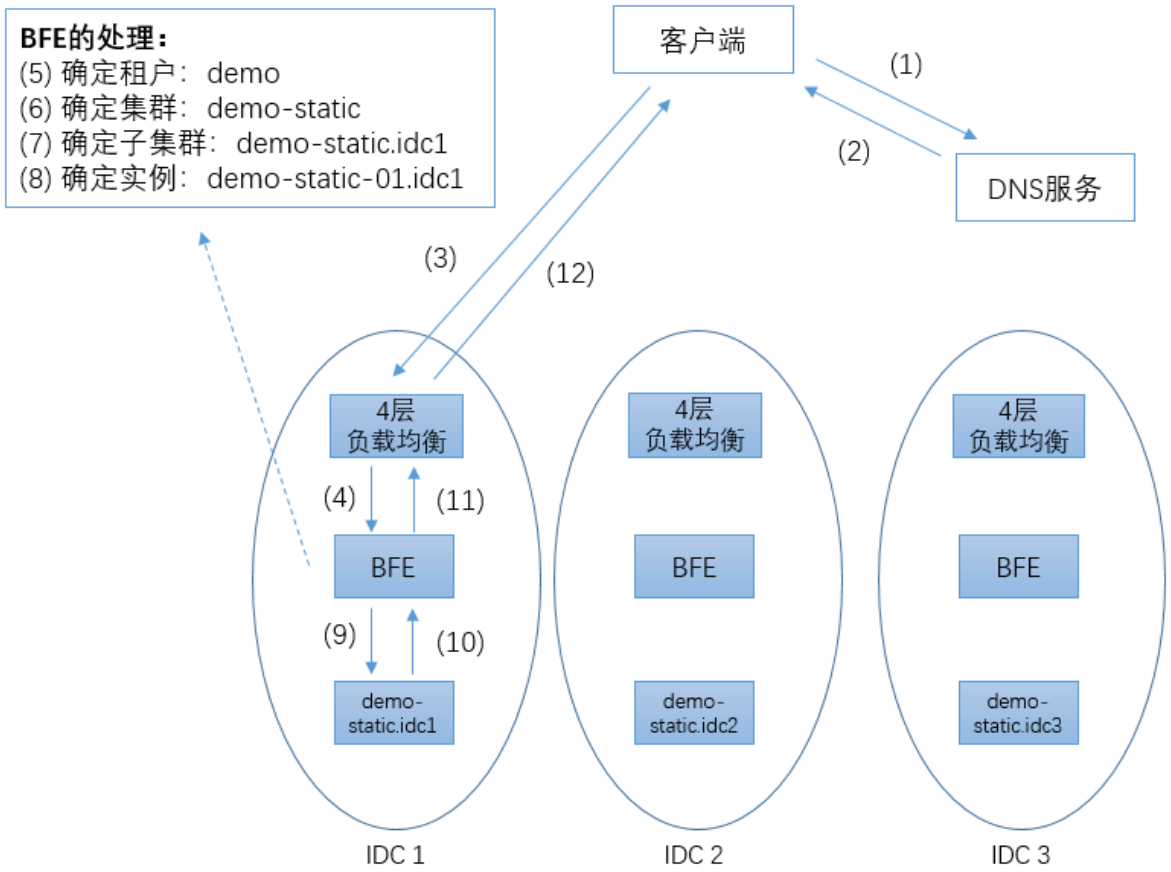
- 实例（Instance）

  每个子集群可包含多个后端服务实例（Instance）。每个后端实例通过“IP地址 + 端口号”标识。

下图用一个例子对以上这些概念之间的关系给出了说明。其中包含2个租户。租户1配置了2个集群（集群A和集群B），这2个集群分别有2个子集群和1个子集群。各子集群有1-3个实例。租户2只配置了一个集群（集群C），集群C有2个子集群，各自有2个实例。



转发过程



下面以上图为例，说明BFE的转发过程。

这里是一个多数据中心的场景，包含3个数据中心（IDC1至 IDC3），在3个数据中心各有一个外网出口。这3个数据中心可能在临近区域内（一般被称为Region），也可能不在同一地域内。

在每个数据中心内，都部署了4层负载均衡集群，也部署了基于BFE的7层负载均衡集群。在一般的部署场景下，BFE都是作为四层负载均衡的RS（Real Server）存在。在一个BFE集群中，一般都会包含2个以上BFE运行实例，位于不同的服务器上。在某个BFE实例出现故障（由于服务器硬件、操作系统、或BFE自身的问题）的情况下，四层负载均衡系统可以自动摘除有问题的实例，从而实现高可用（HA：High Availability）。

这里我们假设有一个提供静态页面访问的服务，命名为demo-static。这个服务在3个数据中心中都有部署，组织为3个独立的子集群（demo-static.idc1 至 demo-static.idc3）。

demo-static使用demo.example.com域名来对外提供服务。在3个数据中心各分配了一个外网IP，假设分别为6.6.6.6、7.7.7.7和8.8.8.8（注：这些域名和IP地址都是虚构的，仅用于说明BFE的运行机制）。

客户端要访问demo-static服务，首先要进行域名解析，将域名解析为合适的IP地址。在上图的步骤1和2中，根据客户端的来源地，智能DNS返回IP地址为6.6.6.6。

之后客户端向6.6.6.6地址的80端口发起建立TCP连接。为了使案例简单，这里假设客户端和服务端使用普通HTTP协议交互，而没有使用HTTPS协议。通过四层负载均衡系统的代理，最终客户端和BFE集群中的某个BFE实例建立起TCP连接。在一个TCP连接内，可以发送1个或多个HTTP请求。一个连接内发送HTTP请求的数量，被称为连接复用率。在HTTP/2出现之前，一般场景下的连接复用率为2-3，也就是说一个TCP连接发送2-3个HTTP请求后就关闭了。在HTTP/2出现后，连接的复用率有了较大的提高。

在HTTP请求到达BFE后，步骤5至步骤8是BFE处理的关键步骤。

- 步骤 5：确定HTTP请求所属的租户

  多租户支持是BFE根据云场景所设计提供的能力。目前BFE可以根据HTTP请求头中的Host字段或HTTP请求的目标IP地址来确定租户。

  在本案例中，针对HTTP请求头中demo.example.com域名，BFE找到对应的租户为demo。

- 步骤 6：根据租户的分流规则，决定HTTP请求的目的集群

  对于每个租户，可以配置一张独立的路由转发表。通过查找路由转发表，确定请求所属的目的集群。路由转发机制的详情将在后面的章节中介绍。

  在本案例中，经查表确定对应的目的集群为demo-static。

- 步骤 7：根据集群的内网流量调度策略，选择合适的子集群

  对于每个BFE集群，可以针对每个集群的各子集群设置转发权重。BFE根据设置的转发权重来执行转发操作。内网流量调度机制的详情将在后面的章节中介绍。

  在本案例中，假设在IDC1的BFE集群上，demo-static的3个子集群对应的转发权重为（100, 0, 0）。所以，确定转发的目标子集群为demo-static.idc1。

- 步骤 8：根据集群的子集群负载均衡策略，选择合适的实例

  对于每个集群，可以设置子集群的负载均衡策略，如WRR（加权轮询）、WLC（加权最小连接数）等。BFE根据子集群的负载均衡策略，在子集群中选择合适服务实例来处理请求。

  在本案例中，最终选择demo-static-01.idc1来处理请求。

随后，请求被发往后端实例demo-static-01.idc1（步骤9）。BFE收到后端实例回复的响应（步骤 10），通过四层负载均衡系统将响应返回给用户（步骤11、12）。

多租户机制的讨论

“多租户支持”是云计算系统的重要需求。在七层负载均衡转发服务的多租户实现机制上，有两种可能：

（1） 使用隔离的转发资源

这是在很多公司中常用的方式。对于不同的业务，搭建独立的七层负载均衡转发集群。这种方式的主要好处是避免了业务间的互相干扰，但是也会导致资源的忙闲不均。另外，每个独立的转发集群规模都不大，在抵御突发流量或攻击流量的能力都不足。

（2）使用公用的转发资源

这是在百度内部采用的方式。BFE平台是一个支持多租户的平台，上千个租户混用同一组转发资源。公有的BFE转发集群具有足够的容量来抵御突发流量或攻击流量。使用这种方式，要求转发引擎本身具有多租户的支持，从而对多个租户间的配置进行隔离；另外，也要求配合提供平台化的能力，支持多个租户同时发起配置的变更。

一个常见的问题是，在公用转发资源的情况下，如何解决转发资源的冲突问题。在某些情况下，可能会由于一个业务的流量突发，对共享资源的其它业务产生干扰。在共享资源的模式下，这种情况肯定是无法完全避免的。对比网络中传统QoS问题的解决，有两种思路：增加对业务使用资源的复杂控制机制；不做复杂的控制机制，而靠供应足够的资源来解决。从以往的历史经验看，在互联网的技术发展过程中下第二种思路取得了胜利。在七层负载均衡转发场景下，使用复杂的控制机制必然会导致额外的资源消耗，我们选用的机制是不使用复杂的控制机制、而提供足够的共享资源。如果真的有某个业务有特别大的流量，则可以通过在四层负载均衡上对对应业务的Virtual Server限速来解决。

BFE的路由转发机制

概述

在BFE的转发过程中，在确定请求所属的租户后，要根据HTTP报头的内容，进一步确定处理该请求的目标集群。在BFE内对每个租户维护一张独立的“转发表”。对于每个属于该租户的请求，通过查询转发表，获得目标集群。

转发表

转发表由多条“转发规则”组成。在查询时，对多条转发规则顺序查找；只要命中任何一条转发规则，就会结束退出。其中最后一条规则为“默认规则（Default）”。在所有转发规则都没有命中的时候，执行默认规则。

每条转发规则包括两部分：匹配条件，目标集群。其中匹配条件使用BFE自研的“条件表达式“（Condition Expression）来表述。

下图展示了一个转发表的例子。

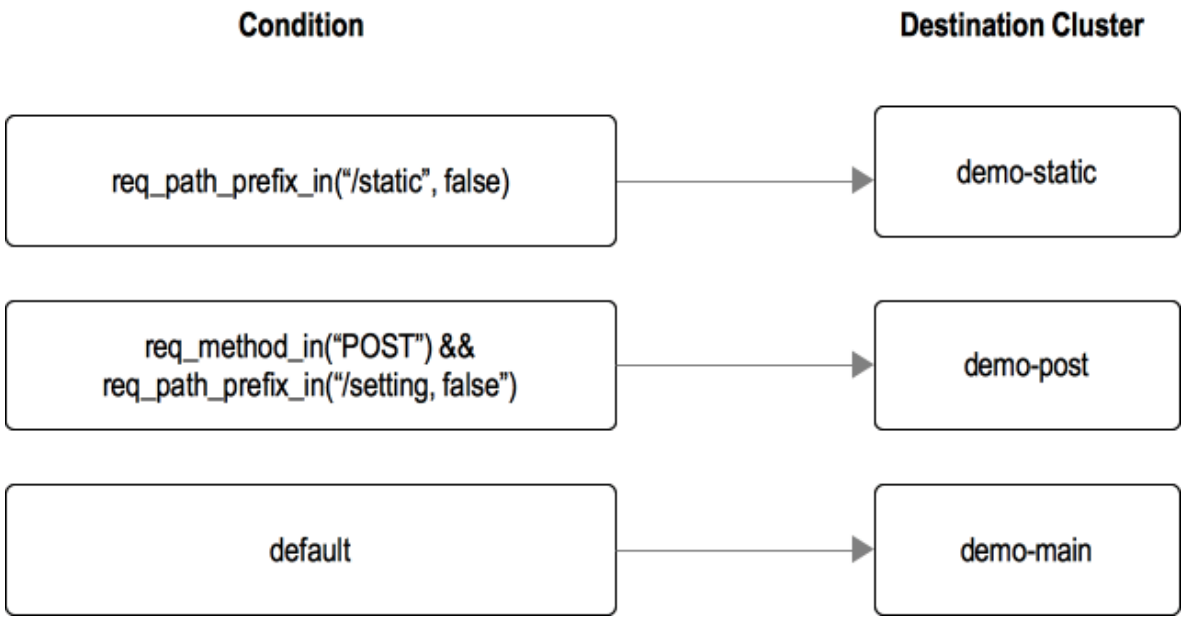
在这个例子中，包含以下3种服务集群：

- 静态集群(demo-static)：服务静态流量
- post集群(demo-post)：服务post流量
- main集群(demo-main)：服务其他流量

期望的转发逻辑如下：

- 对于path以”/static”为前缀的，都发往demo-static集群
- 请求方法为”POST”、且path以”/setting”为前缀的，都发往demo-post
- 其它请求，都发往demo-main

在转发表中，使用条件表达式来描述以上转发条件。



条件表达式

条件表达式是BFE路由转发的核心机制，下面对条件表达式的设计思想和机制进行介绍。

更多关于条件表达式的细节，可以查看BFE开源官网中关于条件表达式的说明

设计思想

在使用Go语言重构BFE转发引擎之前，BFE使用正则表达式来描述转发的条件。在实践中，发现正则表达式存在以下2个严重问题：

- 配置难以维护

  正则表达式存在严重的可读性问题。用正则表达式编写的转发条件很难看懂，且易存在二义性。也经常会发现一个人编写的分流条件，其他人很难接手继续维护。

- 性能存在隐患

  对于编写不当的正则表达式，可能在特定的流量特征下出现严重的性能退化。在线上曾经发生过这样的情况：原本每秒可以处理几千请求的服务，由于增加了一个正则表达式描述，性能下降到每秒只能处理几十个请求。

针对正则表达式存在的问题，在重构BFE转发引擎时设计了条件表达式。条件表达式的设计中有以下思想：

- 在表述中明确指定所使用的HTTP请求字段，提升可读性

  例如，从req_path_prefix_in()这样的名字，可以立刻看出是针对请求中的path部分进行前缀匹配；从req_method_in()可以看出是针对请求的method字段进行匹配。

- 控制计算的复杂度，降低性能退化的风险

  条件表达式主要使用精确匹配、前缀匹配、后缀匹配等计算方式。这些计算方式的计算复杂度都较低。

基本概念

条件原语（Condition Primitive）

- 条件原语是基本的内置条件判断单元，执行某种比较来判断是否满足条件

```
// 如果请求host是"bfe-networks.com"或"bfe-networks.org", 返回true
req_host_in("bfe-networks.com|bfe-networks.org")
```

- 条件原语是判断的最小单元。按照请求、响应、会话、系统等几个分类，建立了几十个条件原语。也可以根据需求，增加新的条件原语。

条件表达式（Condition Expression）

- 条件表达式是多个条件原语与操作符(例如与、或、非)的组合

```
// 如果请求域名是"bfe-networks.com"且请求方法是"GET", 返回true
req_host_in("bfe-networks.com") && req_method_in("GET")
```

条件变量（Condition Variable）

- 可以将条件表达式赋值给一个变量，这个变量被定义为条件变量

```
// 将条件表达式赋值给变量bfe_host
bfe_host = req_host_in("bfe-networks.com")
```

高级条件表达式（Advanced Condition Expression）

- 高级条件表达式是多个条件原语和条件变量与操作符(例如与、或、非)的组合
- 在高级条件表达式中，条件变量以$前缀作为标示

```
// 如果变量bfe_host为true且请求方法是"GET"，返回true
$bfe_host && req_method_in("GET")
```

- 条件变量和高级条件表达式的引入，是为了便于条件表达式逻辑的复用。

语法

条件原语的语法

条件原语的形式如下：

```
func_name(params)
```

- **func_name**是条件原语名称
- **params**是条件原语的参数，可能是0个或多个
- 返回值类型是**bool**

条件表达式的语法

条件表达式(CE: Condition Expression)的语法定义如下：

```
CE = CE && CE
   | CE || CE
   | ( CE )
   | ! CE
   | ConditionPrimitive
```

高级条件表达式的语法

高级条件表达式(ACE: Advanced Condition Expression)的语法定义如下：

```
ACE = ACE && ACE
    | ACE || ACE
    | ( ACE )
    | ! ACE
    | ConditionPrimitive
    | ConditionVariable
```

操作符优先级

操作符的优先级和结合律与C语言中类似。下表列出了所有操作符的优先级及结合律。操作符从上至下按操作符优先级降序排列。

优先级	操作符	含义	结合律
1	()	括号	从左至右
2	!	逻辑非	从右至左
3	&&	逻辑与	从左至右
4	||	逻辑或	从左至右

条件原语所匹配的内容

条件原语可以对请求、响应、会话及请求上下文中的内容进行匹配。每个条件原语都会对某种内容进行有针对性的精确匹配。这样的方式提升了转发配置的描述准确性，也提升了可读性和可维护性。

条件原语匹配的内容包括：

- **cip**：Client IP，客户端地址
  - 如：req_cip_range(start_ip, end_ip)
- **vip**：Virtual IP，服务端虚拟IP地址
  - 如：req_vip_in(vip_list)
- **cookie**：HTTP头部所携带的cookie
  - 对一个cookie，包含key和value两部分
  - 如：req_cookie_key_in(key_list)，req_cookie_value_in(key, value_list, case_insensitive)
- **header**：准确的说，应该是HTTP头部字段（HTTP header field）
  - 对一个HTTP头部字段，包含key和value两部分
  - 如：req_header_key_in(key_list)，req_header_value_in(header_name, value_list, case_insensitive)
- **method**：HTTP方法
  - HTTP方法包括GET、POST、PUT、DELETE等
  - 如：req_method_in(method_list)
- **URL**：统一资源定位符
  - 如：req_url_regmatch(reg_exp)

对于URL，其详细格式为：

```
scheme:[//authority]path[?query][#fragment]
```

其中，authority的格式为：

```
[userinfo@]host[:port]
```

针对URL，可以进一步匹配其中的内容：

- **host**：主机名
  - 如：req_host_in(host_list)
- **port**：端口
  - 如：req_port_in(port_list)
- **path**：路径
  - 如：req_path_in(path_list, case_insensitive)
- **query**：查询字符串
  - 对一个查询字符串，包含key和value两部分
  - 如：req_query_key_in(key_list)，req_query_value_in(key, value_list, case_insensitive)

条件原语名称的规范

目前在BFE开源项目中，已经包括40多种条件原语。条件原语的名称会遵循一定的规范，以便于分类和阅读。

BFE开源项目所支持条件原语的列表，可以查看BFE开源官网

条件原语名称前缀

- 针对Request的原语，会以”**req_**“开头
  - 如：req_host_in()
- 针对Response的原语，会以”**res_**“开头
  - 如：res_code_in()
- 针对Session的原语，会以”**ses_**“开头
  - 如：ses_vip_in()
- 针对系统原语，会以”**bfe_**“ 开头
  - 如：bfe_time_range()

条件原语中比较的动作名称

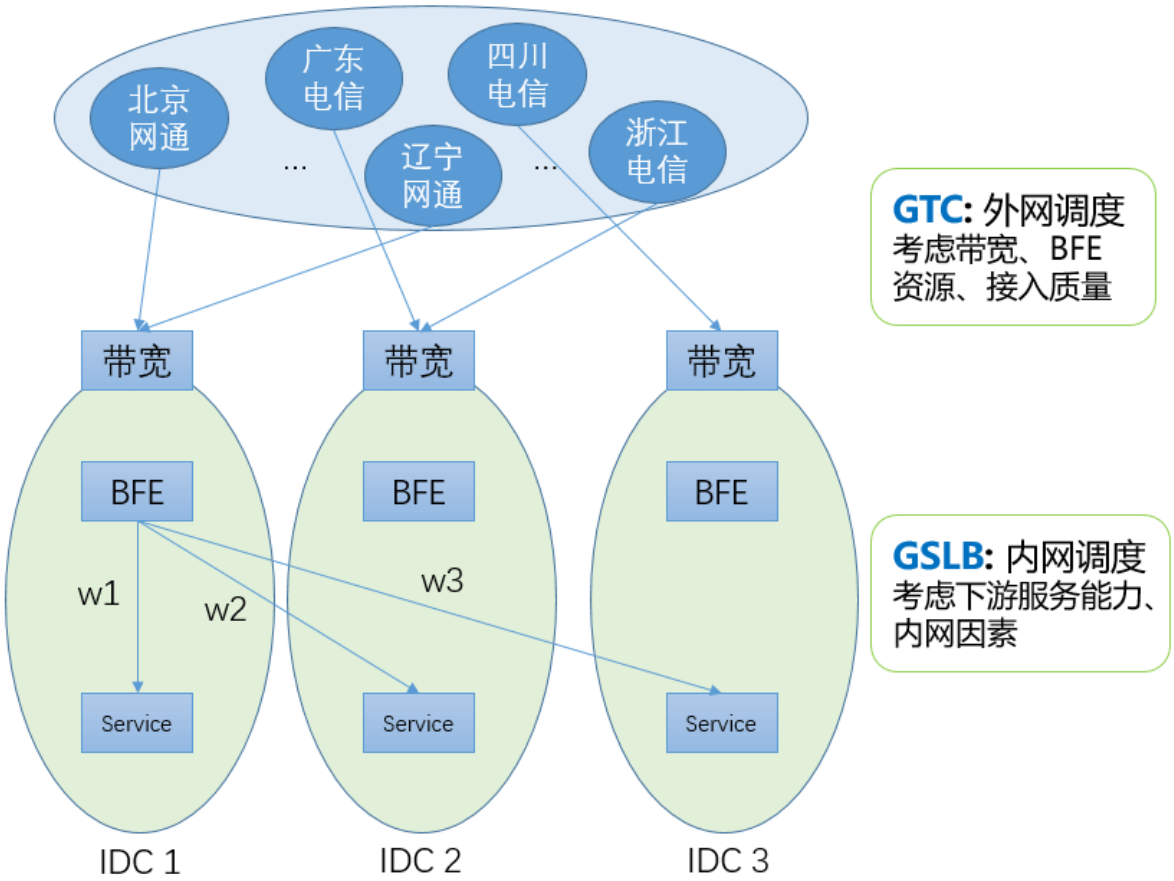
- **match**：精确匹配
  - 如：req_tag_match()
- **in**：值是否在某个集合中
  - 如：req_host_in()
- **prefix_in**：值的前缀是否在某个集合中
  - 如：req_path_prefix_in()
- **suffix_in**：值的后缀是否在某个集合中
  - 如：req_path_suffix_in()
- **key_exist**：是否存在指定的key
  - 如：req_query_key_exist()
- **value_in**：对给定的key，其value是否落在某个集合中
  - 如：req_header_value_in()

- **value_prefix_in**：对给定的key，其value的前缀是否在某个集合中
  - 如：req_header_value_prefix_in()
- **value_suffix_in**：对给定的key，其value的后缀是否在某个集合中
  - 如：req_header_value_suffix_in()
- **range**：范围匹配
  - 如：req_cip_range()
- **regmatch**：正则匹配
  - 如：req_url_regmatch()
  - 注：这类条件原语不合理使用将明显影响性能，谨慎使用
- **contain**: 字符串包含匹配
  - 如：req_cookie_value_contain()

BFE的内网流量调度机制

背景

全局流量调度解决方案



经过多年的建设，对于由IDC服务的业务流量，百度形成了两层的全局流量调度系统。包括：

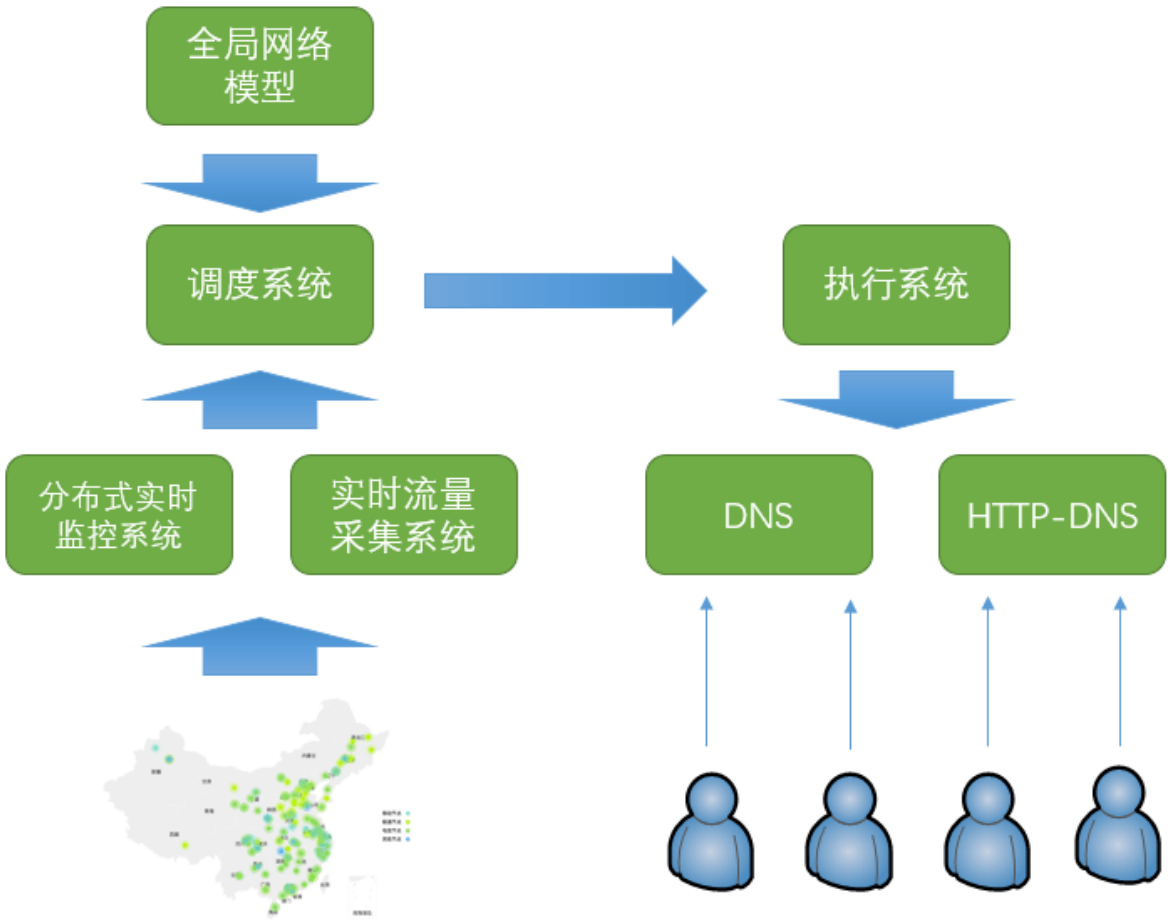
- GTC：外网流量调度

  基于DNS生效，将各省、运营商的用户流量引导到合适的网络入口。在调度计算中，GTC要考虑外网带宽（容量和使用情况）、BFE平台的转发资源（容量和使用情况）、用户到各带宽出口的接入质量（连通性、访问延迟）等因素。

- GSLB：内网流量调度

  基于BFE生效，将到达各BFE集群的流量，按照权重转发到位于各数据中心的子集群。

外网流量调度



GTC负责在各网络入口间进行流量调度。GTC包括以下3个主要步骤：

- 实时监控

  由位于各地的监控节点，持续向各外网接入点发送探测信号，对各地和接入点之间的连通性和质量进行监控。如果发现异常，分布式的实时监控系统会在1分钟内将故障信号上报给调度系统。

- 调度计算

  实时流量采集系统从路由器获取实时的带宽使用情况，从七层负载系统获取实时的每秒请求情况。调度系统根据实时流量数据和实时监控情况，配合全局网络模型，在1分钟内计算出新的调度方案。

- 下发执行

  调度系统将调度方案下发给DNS和HTTPDNS执行。由于DNS缓存的因素，客户端的生效需要一定的时间。百度大部分域名的DNS TTL设置为300秒（即5分钟）。一般在下发后，要经过8-10十分钟才能完成90%以上用户的生效。

和前一代外网调度系统相比，GTC有以下两方面的提升：

- 加快了外网故障处理的速度

  通过“实时监控+自动调度计算”，从故障发生到启动DNS下发的时间压缩至2分钟以内。

- 降低了配置维护的成本

不需要针对域名维护复杂的预案。

业内很多类似的系统采用“预案”机制。例如，存在A和B两个备选的外网IP。预案会这么写：如果A出问题了，就把流量切换到B；如果B出问题了，就把流量切换到A。对于每个直接分配了IP地址的域名（也就是写为A记录的域名），都需要写这么一个预案。

预案机制的最大问题就是维护成本很高。首先，维护成本和外网出口的数量成指数关系。2个出口的情况是非常简单的；如果有5个甚至10个出口，预案是非常不好写的，需要考虑各种可能性。另外，维护成本和域名的数量成线性关系。如果有几千个域名，这时如果要对带宽出口做一个调整（增加、或删除一个出口），所要付出的工作量是惊人的。

外网流量调度主要适用于以下场景：

- 网络入口故障

  由于网络入口本地、或运营商网络的故障，导致用户无法访问网络入口

- 网络入口由于攻击导致拥塞

  大规模的DDoS攻击可以达到数百G，甚至达到T级别，可以直接将网络入口的入向带宽打满

- 网络接入系统故障

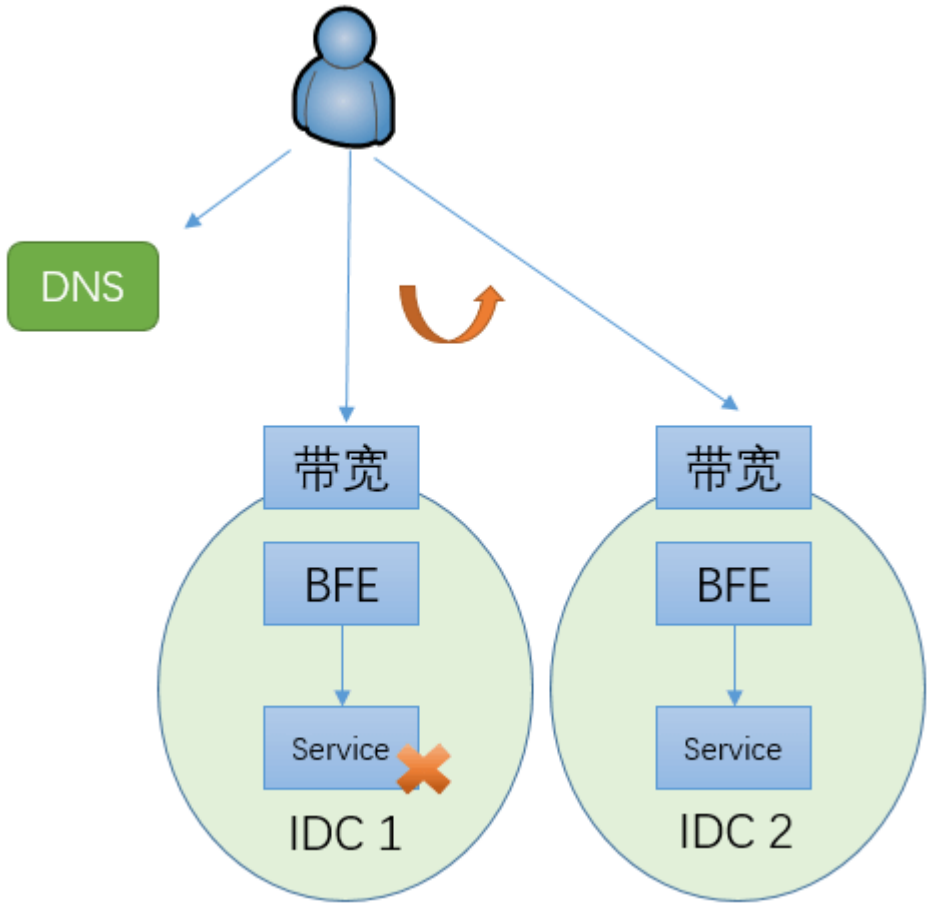
  如四层负载均衡系统或七层负载均衡系统的故障

- 分省连通性故障

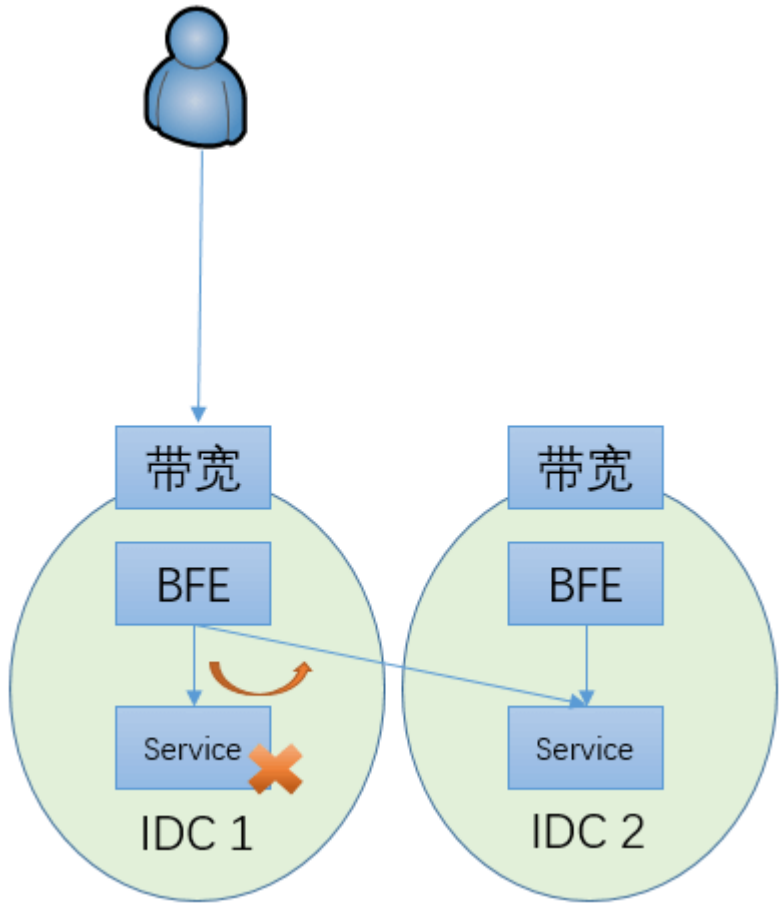
  虽然从总体看网络入口可以访问，但是从某个运营商的某个地区无法访问。这部分是由于用户所在的地区出现网络局部异常，也可能是由于服务所使用的IP在局部地区被误封禁。

为什么需要内网流量调度

在多数据中心的场景下，如果没有内网流量调度机制，当一个数据中心内的服务发生故障时，只能通过改变域名对应的IP地址将流量调度到另外一个数据中心。如上文所说，在改变权威DNS的配置后，需要8-10分钟才能完成90%以上用户的生效。在完成切换之前，原来由故障IDC所服务的用户都无法使用服务。而且，运营商的Local DNS数量很大，可能会存在有故障或不遵循DNS TTL的Local DNS，从而导致对应的用户使用更长的时间完成切换，甚至一直都不切换。

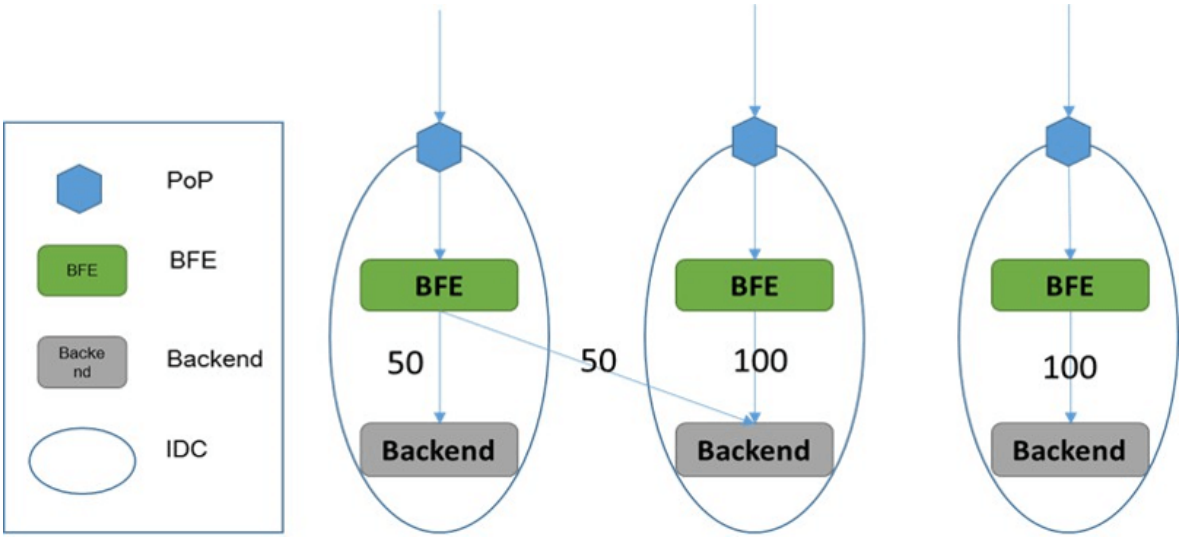


在引入内网流量调度机制后，可以通过修改BFE的配置将流量从故障的服务集群切走。在百度内部，配合自动的内网流量调度计算模块，在感知故障后，可以在30秒内完成流量的调度。和完全依赖外网流量调度的机制相比，故障止损时间有很大的降低，从8-10分钟降低至30秒内。而且，由于执行调度的BFE集群都在内部，内网流量调度的可控性也比基于DNS的外网流量调度要好的多。



内网流量调度

基本工作机制



内网流量调度的工作机制如上图所示，其基本原理非常简单。在每个BFE集群，针对一个服务集群的每个后端子集群分配一组权重。在流量转发时，BFE按照这个权重来决定请求的目标子集群。

另外，对每个服务集群，还包含一个虚拟的子集群，称为BLACKHOLE（黑洞）。在黑洞集群对应的权重不为0的情况下，分给黑洞子集群的流量会被BFE主动丢弃。在到达BFE流量超过服务集群总体容量的情况下，可以通过启用黑洞集群用来防止服务集群的整体过载。

内网流量调度适用于和内部服务相关的场景，包括：

- 内部服务故障

  在某些场景下（如：服务的灰度发布），可能单个服务子集群出现故障，从而导致服务容量下降、甚至完全无法提供服务。这时候可以通过内网流量调度快速完成止损处理。

- 内部服务压力不均

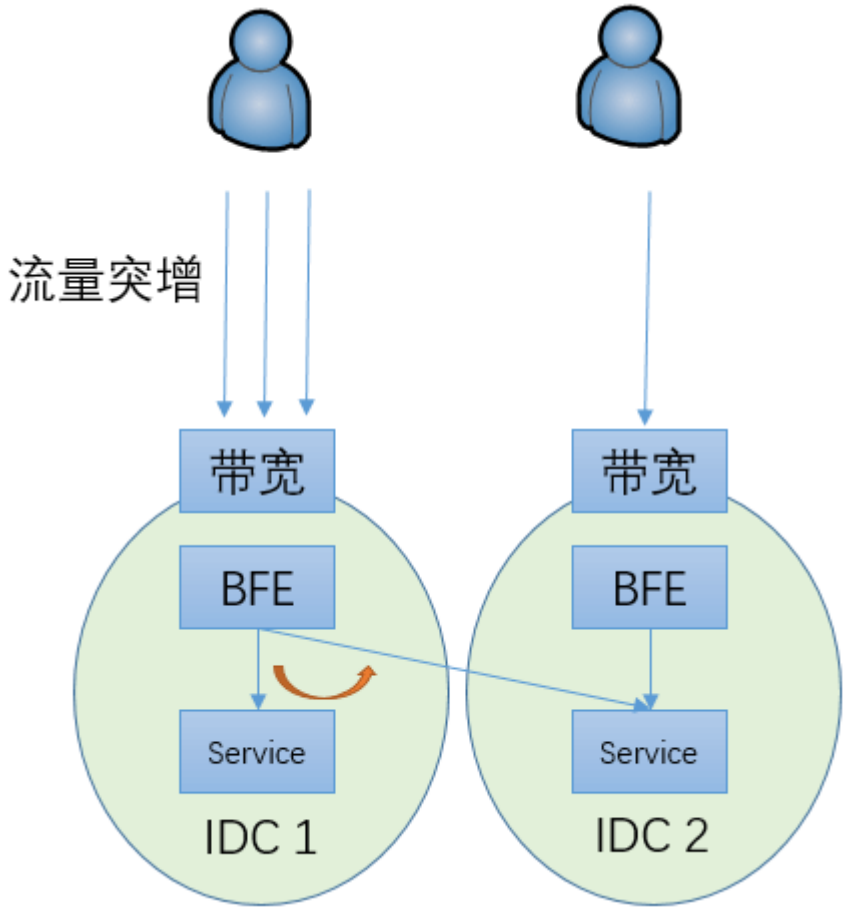
  包括以下两种可能的场景：

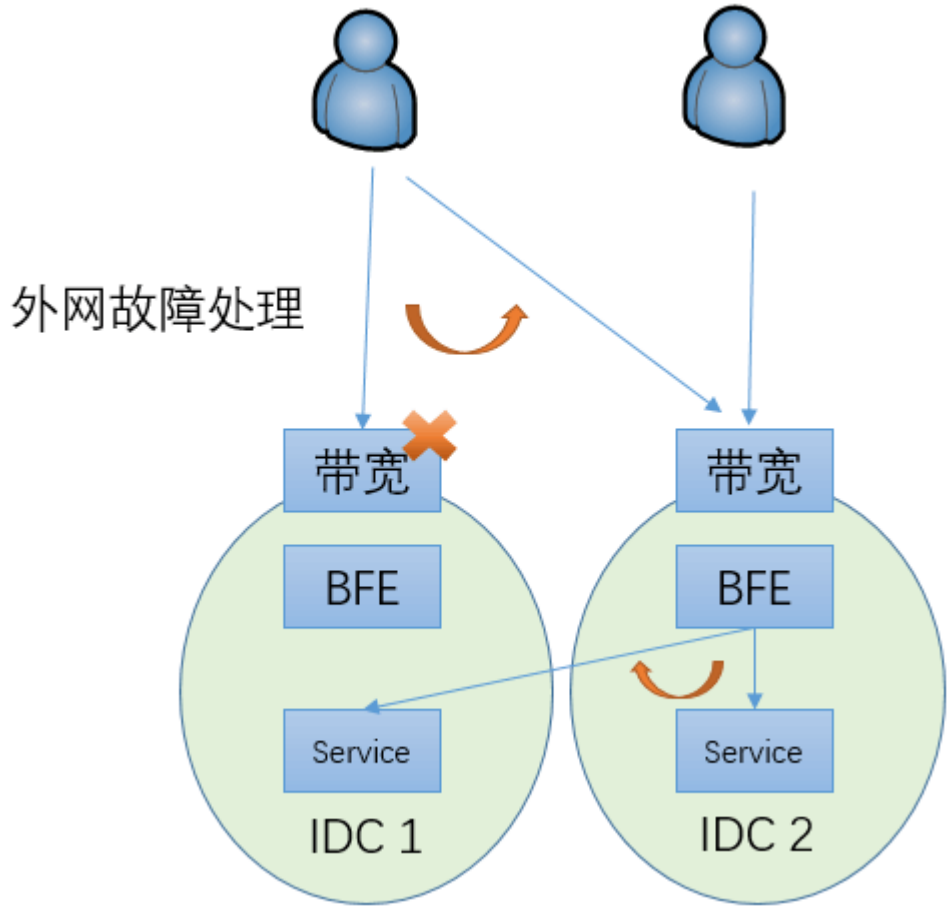
  - 某个地区的用户流量突增，导致单个数据中心内的子集群服务压力超过容量

    这时可以将部分流量调度到其它子集群来服务

  - 由于外网故障处理，在外网将部分流量从一个网络入口调度到另外一个网络入口，从而导致相关的子集群压力超过容量

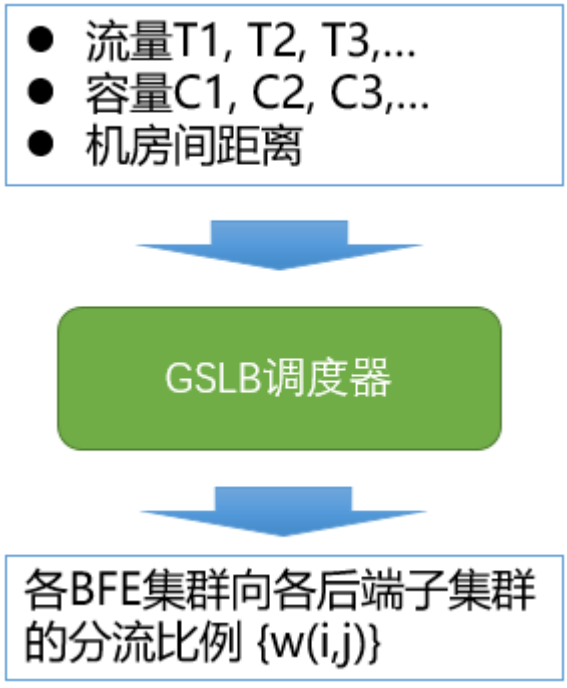
    这时也可以将部分流量调度到其它子集群来服务





内网自动流量调度

内网流量调度的权重可以手工设置。但实际上这个权重不应该是固定的，而应随着用户流量、服务集群的容量及机房间的连通性情况等因素的变化而调整。为此，在百度内实现了一个内网流量的调度器，用于对分流的权重比例进行计算。



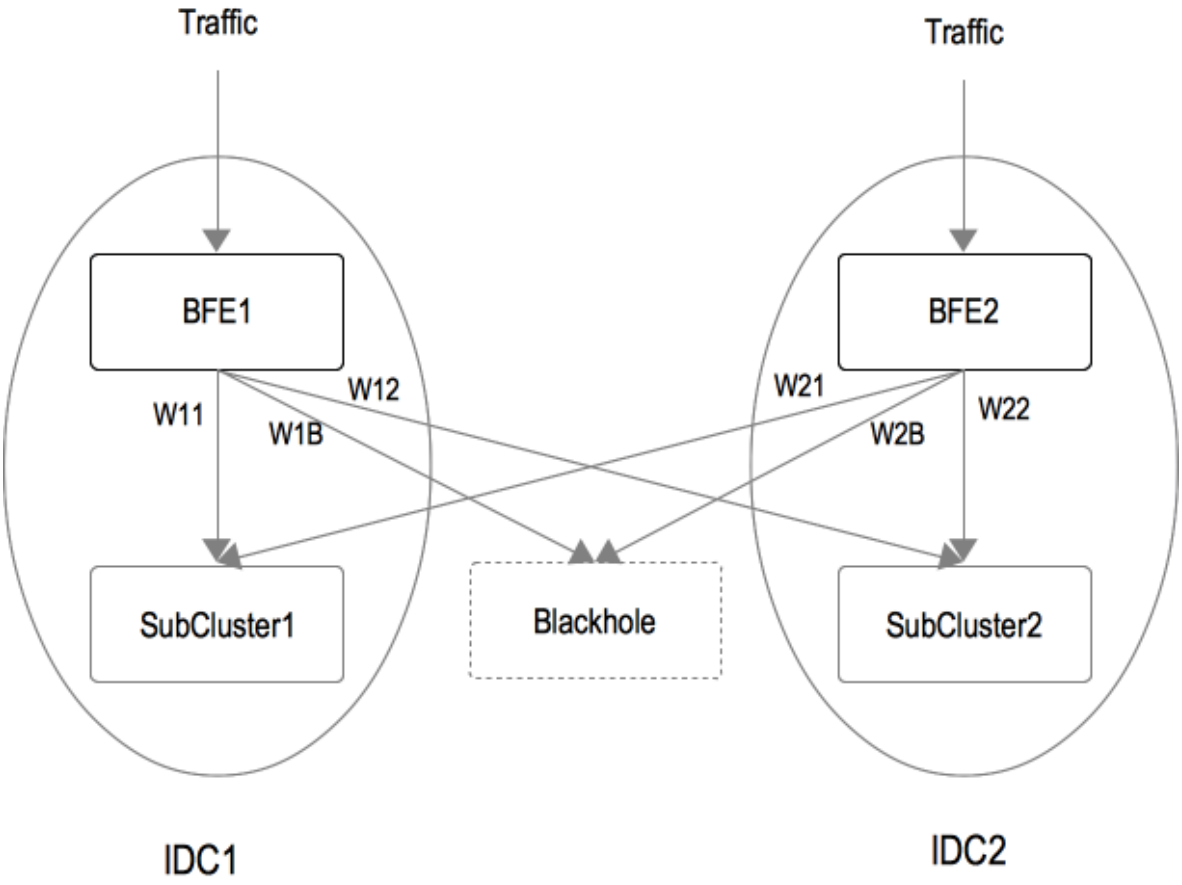
内网流量调度的总体机制为：

- 流量采集：基于BFE的访问日志，实时获取到达各BFE集群的各服务的流量
- 权重计算：根据流量、各服务子集群的容量、各数据中心网络连通性/距离等因素，计算各BFE集群向各服务子集群的分流权重
- 下发执行：由各BFE集群按照分流权重执行转发

目前在BFE开源项目中，支持内网流量调度权重的手工设置，未包含内网自动流量调度的相关模块。

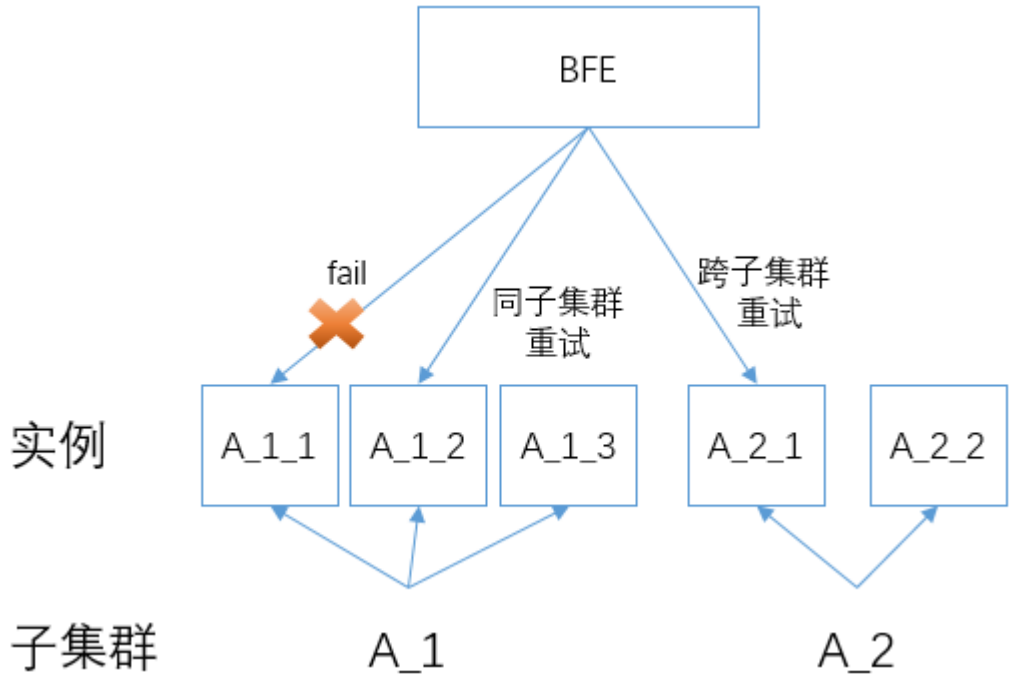
示例场景

- 示例场景如下图所示，包含：
    - 两个IDC：IDC_1和IDC_2
    - 两个BFE集群：BFE_1和BFE_2
    - 后端集群有两个子集群：SubCluster_1和SubCluster_2
- 可以针对BFE集群，设置子集群的分流比例，如：
    - BFE_1集群的分流配置为：{SubCluster_1: W11, SubCluster_2: W12, Blackhole: W1B}
    - BFE_2集群的分流配置为：{SubCluster_1: W21, SubCluster_2: W22, Blackhole: W2B}
- BFE实例根据上述配置做WRR调度（加权轮询），向子集群转发请求
    - 例如，当BFE_1的分流配置{W11, W12, W1B}为{45, 45, 10}时，BFE_1转发给SubCluster_1、SubCluster_2、Blackhole的流量比例依次为：45%、45%、10%。
- 通过修改上述配置，可以将流量在不同的子集群之间切换，实现负载均衡、快速止损、过载保护等目的。



内网转发的其它机制

失败重试机制



BFE在转发时，支持以下两种失败重试机制：

- 同子集群重试
  - 在一次转发失败后，选择原目标子集群内的其它服务实例进行重试。
  - 在同子集群内重试的最大次数，可以通过配置集群的参数**“同子集群重试次数”**来控制。
- 跨子集群重试
  - 在转发失败后，在原目标子集群之外，使用另外一个子集群进行重试。
  - 跨子集群重试的最大次数，可以通过配置集群的参数**“跨子集群重试次数”**来控制。

在转发失败后，BFE会首先尝试同子集群重试（如果同子集群重试次数大于1），然后再尝试跨子集群重试（如果跨子集群重试次数大于0）。

启用跨子集群重试功能要小心，在某些场景下这个功能可能将过量的流量转移到其它健康的集群，从而导致这些集群的压力过大、甚至被压垮。和上面“内网流量调度”中按照权重将流量转发到各子集群的机制不同，跨子集群重试所引发的流量压力是有一定不可控性的。

BFE并不是在所有请求失败的情况下都会进行重试。如果BFE感知到下游实例已经读取了请求（即使没有完整的读取），也不会再重试了。在这种情况下，BFE无法确认下游实例是否已经处理了请求，如果再次发送可能会导致状态的错误，所以采取了比较保守的策略。

连接池

BFE和下游实例的连接支持两种方式：

- 短连接方式：每次BFE向下游实例转发请求，均需要建立新的TCP连接
- 连接池方式：
  - BFE为每个下游实例维护一个连接池。当BFE需要向某个下游实例转发请求时：
    - 如果连接池中有idle连接，则复用这个连接
    - 如果连接池中没有idle连接，则会建立一个新的TCP连接
  - 当BFE处理完一个请求时
    - 如果连接池中的idle连接数量小于连接池的大小，则将当前使用的连接放入连接池
    - 如果连接池中的idle连接数量大于等于连接池的大小，则关闭当前使用的连接

使用连接池的方式，可以避免新建TCP连接所导致的延迟，从而降低总的转发延迟。由于BFE需要对于每个下游实例都保持长连接，在某些情况下（如，BFE的实例数较大）可能导致下游实例的并发连接数较大。在使用连接池和设置连接池的参数时，需要结合以上因素综合考虑。

会话保持

BFE向下游转发请求时，支持将相同来源请求，转发至固定的业务后端（某个子集群或某个实例），这个功能被称为“会话保持”。

在执行会话保持时，BFE可以基于以下请求来源标识：

- 请求来源IP
- 请求特定头部，例如请求Cookie等

BFE支持以下两种会话保持级别：

- 子集群级别：相同来源的请求，被转发至固定的业务子集群（注：子集群中的任意实例）
- 实例级别：相同来源的请求，被转发至固定的业务实例

BFE的模块插件机制

概述

为了便于增加新的功能，BFE定义了一套完整的模块插件机制，支持快速开发新的模块。

BFE模块插件机制的要点如下：

- 在BFE的转发过程中，提供多个回调点
- 对于一个模块，可以针对这些回调点、对应编写回调函数
- 在模块初始化时，把这些回调函数注册到对应的回调点
- 在处理一个连接或请求时，当执行到某个回调点，会顺序执行所有注册的回调函数

下面将简要介绍BFE中的回调点设置，并列出BFE内置的扩展模块。回调框架的详细设计，请参考实现篇中“回调框架”中的说明。在“如何开发BFE扩展模块”一章中，结合一个具体例子对开发BFE扩展模块的方法进行了详细的说明。

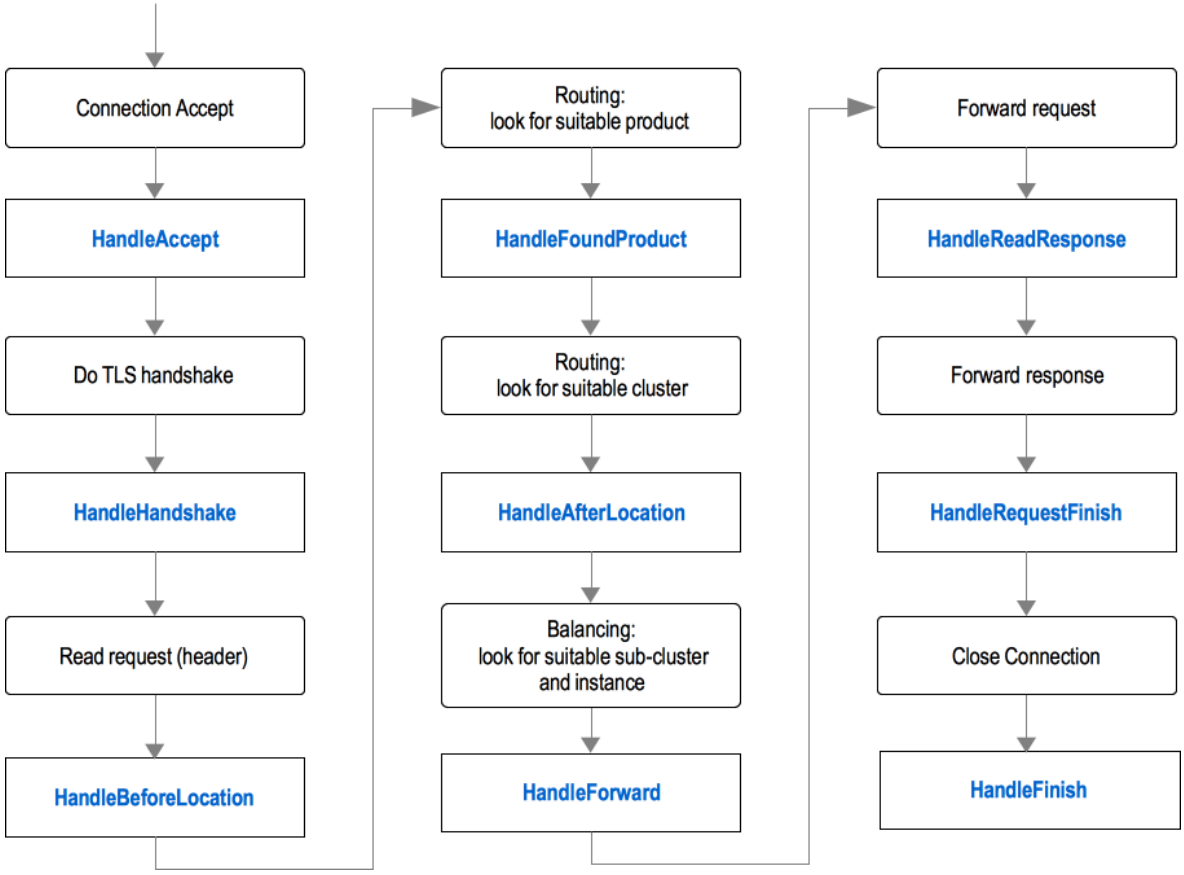
BFE的回调点设置

在BFE中，设置了以下9个回调点：

- HandleAccept: 位于和客户端的TCP连接建立后
- HandleHandshake：位于和客户端的SSL或TLS握手完成后
- HandleBeforeLocation：位于查找产品线之前
- HandleFoundProduct：位于完成查找产品线之后
- HandleAfterLocation：位于完成查找集群之后
- HandleForward：位于完成查找子集群和后端实例之后，位于转发请求之前
- HandleReadResponse：位于读取到后端响应之后
- HandleRequestFinish：位于后端响应处理完毕后
- HandleFinish：位于和客户端的TCP连接关闭后

回调点的定义，可以查看/bfe_module/bfe_callback.go

BFE转发过程中的回调点如下图所示。



BFE内置模块

在BFE源码的 bfe_modules/ 目录中内置了大量的扩展模块。简要说明如下：

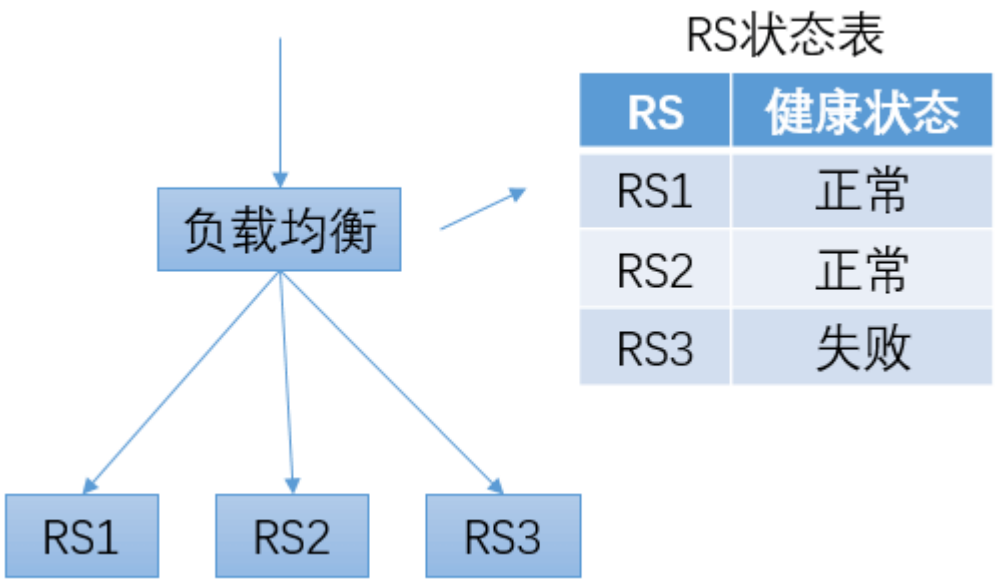
模块类别	模块名称	模块说明
流量管理	mod_rewrite	根据自定义条件，修改请求的URI
	mod_header	根据自定义条件，修改请求或响应的头部
	mod_redirect	根据自定义条件，对请求进行重定向
	mod_geo	基于地理信息字典， 通过用户IP获取相关的地理信息
	mod_tag	根据自定义的条件，为请求设置Tag标识
	mod_logid	用来生成请求标识及会话标识
	mod_trust_clientip	基于配置信任IP列表， 检查并标识访问用户真实IP是否属于信任IP
	mod_doh	支持DNS over HTTPS

模块类别	模块名称	模块说明
	mod_compress	根据自定义条件，支持对响应主体压缩
	mod_errors	根据自定义条件，将响应内容替换为/ 重定向至指定错误页
	mod_static	根据自定义条件，返回静态文件作为响应
	mod_userid	根据自定义的条件， 为新用户自动在Cookie中添加用户标识
	mod_markdown	根据自定义条件， 将响应中markdown格式内容转换为html格式
安全防攻击	mod_auth_basic	根据自定义的条件，支持HTTP基本认证
	mod_auth_jwt	根据自定义的条件，支持JWT (JSON Web Token)认证
	mod_auth_request	根据自定义的条件， 支持将请求转发认证服务进行认证。
	mod_block	根据自定义的条件，对连接或请求进行封禁
	mod_prison	根据自定义的条件， 限定单位时间用户的访问频次
	mod_waf	根据自定义的条件， 对请求执行应用防火墙规则检测或封禁
	mod_cors	根据自定义的条件，设置跨源资源共享策略
	mod_secure_link	根据自定义的条件， 对请求签名或有效期进行验证
流量可见性	mod_access	以指定格式记录请求日志和会话日志
	mod_key_log	以NSS key log格式记录TLS会话密钥， 便于基于解密分析TLS密文流量诊断分析
	mod_trace	根据自定义的条件，为请求开启分布式跟踪
	mod_http_code	统计HTTP响应状态码

可以通过访问BFE实例的 http://localhost:8299/monitor/module_handlers 监控地址，查看到当前运行的BFE实例中所有的回调点、在各回调点注册的模块回调函数列表以及顺序。

健康检查机制

健康检查的原理



在负载均衡系统内，会维护一个RS的状态表，用于保存每个RS的健康状态。在转发流量时，只会将流量转发给健康状态为“正常”的RS。

为了获得RS的健康状态，负载均衡系统会定期向RS发送探测请求，并根据收到响应的情况来修改对应RS的健康状态。

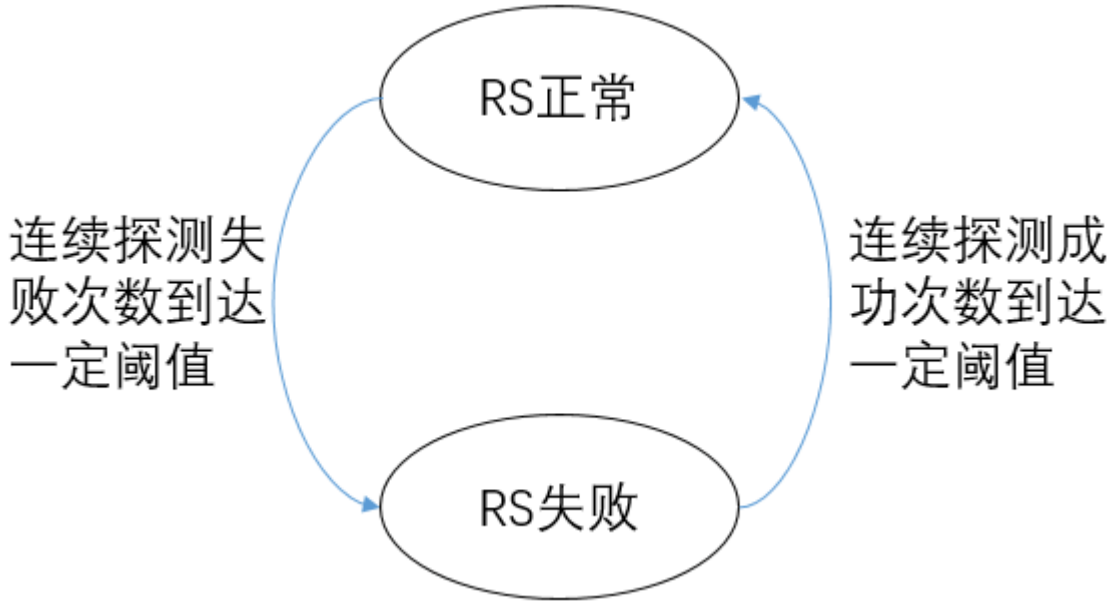
主动健康检查 vs 被动健康检查

主动健康检查

“**主动健康检查**”的机制可以简要描述如下：

- 负载均衡系统**持续**向RS发送探测请求
- 在RS为“正常”的状态下，如果连续探测失败的次数到达一定的阈值，则改变RS的健康状态为“失败”
- 在RS为“失败”的状态下，如果连续探测成功的次数到达一定的阈值，则改变RS的健康状态为“正常”

主动健康检查中RS的状态变化可以用一个简单的有限状态机来描述。

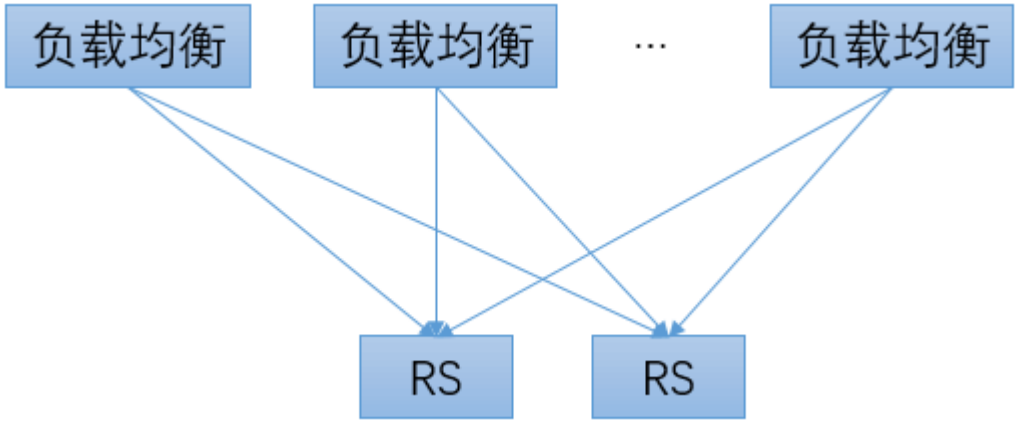


对于健康检查来说，一个重要的指标是“响应时间”，即：当RS的状态发生改变后，经过多长时间负载均衡系统才能感知到变化。如果不能及时感知到RS的失败，可能会导致请求发送到失败的RS节点上，从而导致服务的失败；如果不能及时感知到RS的恢复，可能会使其它正常的RS节点压力过大。

对于主动健康检查来说，降低响应时间的方法是提高发送探测请求的频率。但这个方案也是有代价的。一方面，它会增大负载均衡系统发送探测请求的压力；另外一方面，也会增大RS的压力。第二个问题在软件负载均衡场景中变得更为突出。

被动健康检查

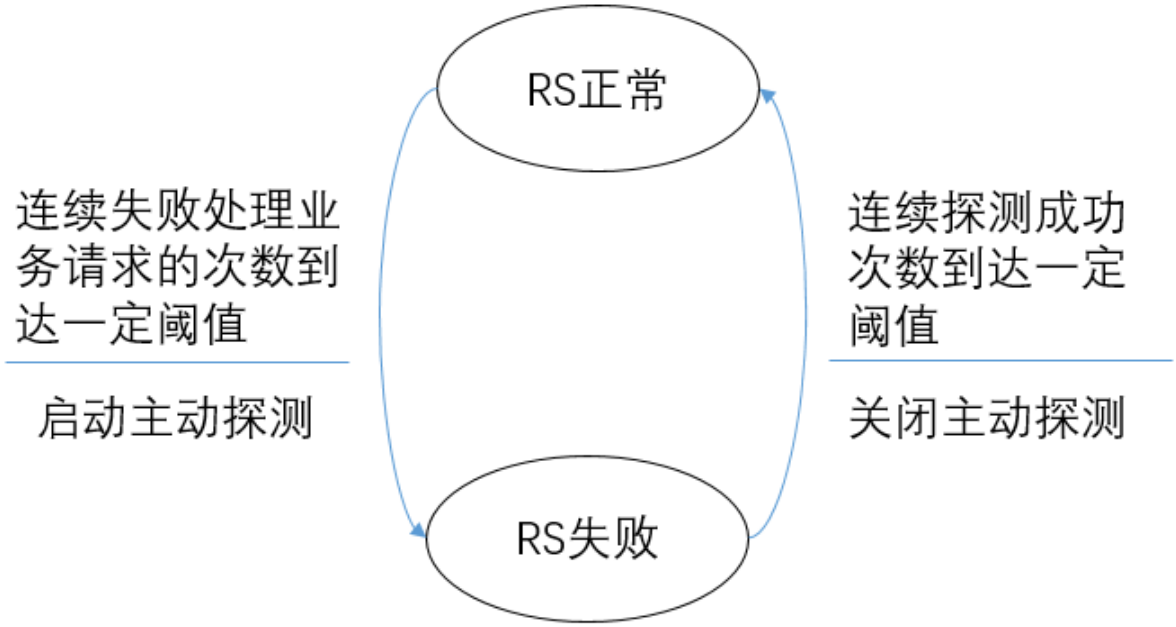
在前面章节“网络前端技术的发展趋势”中提到，“软件化”是网络前端接入系统的重要趋势。软件负载均衡的一个重要特征是支持负载均衡节点的大规模横向扩展部署。在传统硬件负载均衡场景，最常见是“主+备”的部署模式，RS的上游一般是2个负载均衡节点。而在软件负载均衡场景下，一个负载均衡集群可能由几十个节点构成。尤其是对于七层负载均衡场景，单个负载均衡节点的容量较小，单个集群可达到数百个节点。在这种情况下，如果继续使用“主动健康检查”的机制，持续的健康检查探测请求会对下游的RS产生很大的压力。



为了解决上面的问题，可以使用”**被动健康检查**“的机制，简要描述如下：

- 在RS为“正常”的状态下，负载均衡系统不会主动向RS发送探测请求
- 在RS为“正常”的状态下，如果RS**处理业务请求**连续失败的次数超过一定阈值，则改变RS的健康状态为“失败”
- 在RS为“失败”的状态下，负载均衡系统向RS发送探测请求
- 在RS为“失败”的状态下，如果连续探测成功的次数到达一定的阈值，则改变RS的健康状态为“正常”

被动健康检查中RS的状态变化及主动探测的启停可以用一个下面的有限状态机来描述。



主动和被动的结合

和主动健康检查相比，被动健康检查的一个显著的差异是：使用正常的业务请求来“捎带”的进行探测。这样做的好处是：

- 在RS正常的情况下，不需要额外的发送探测请求，从而降低了负载均衡系统和RS的处理成本
- 在RS处理正常请求的频度较高（即，正常请求的频度远高于健康检查探测请求的频度）的情况下，被动健康检查可以更快的发现RS的异常

被动健康检查也有它的缺点：

- 如果无法对业务请求进行失败重试，在RS失败的情况下，可能导致正常请求的失败
- 在业务请求频度较低（如：对某个RS，几分钟才有一个请求）的情况下，可能导致无法及时发现RS的失败

综合主动方式和被动方式的优缺点，建议在实践中采用主动和被动相结合的方式：

- 启用被动健康检查，可以帮助请求频度较高的RS快速发现失败的情况，而不需要承担高频探测请求的成本
- 同时启动“低频”（如：30秒或60秒一次）的主动健康检查，用于及时发现请求频度较低RS的失败

在使用主动和被动结合机制时，RS的健康状态由两种检查结果汇总得到。如果两者之一的检查结果为失败，则RS的状态为失败；仅当两种检查的结果都为成功时，RS状态才是成功。

RS	被动检查	主动检查	汇总结果
RS1	正常	失败	失败
RS2	正常	正常	正常
RS3	失败	正常	失败

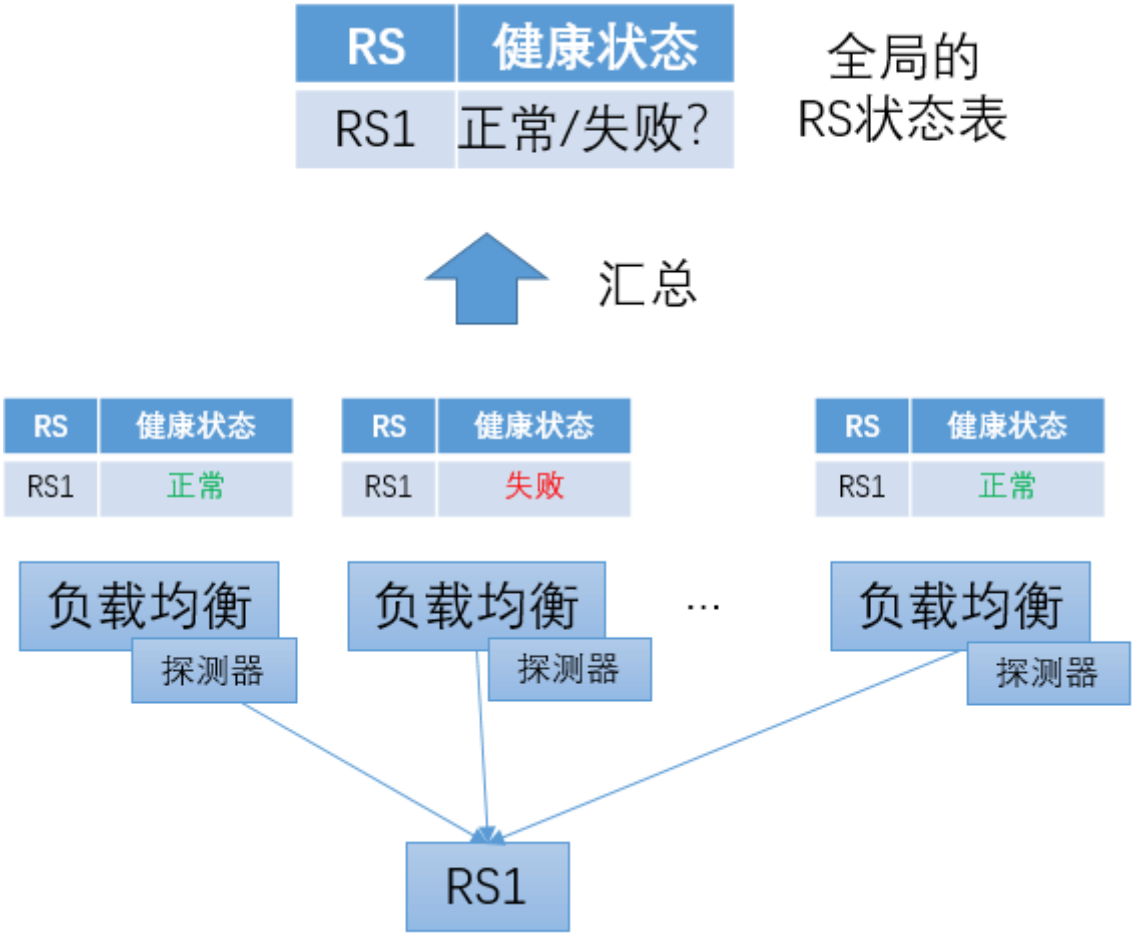
集中式健康检查 vs 分布式健康检查

在使用主动健康检查机制时，按照探测器部署的位置，可以分为“分布式健康检查”和“集中式健康检查”。

在**分布式健康检查**方式中，探测器和负载均衡转发程序同机部署。每个负载均衡转发实例都根据自己的探测结果维护独立的RS状态表。对于同一个RS，每个负载均衡转发实例由于服务器状态、网络状态（如TOR、路由、机房连通性）等方面的差异，可能关于健康状况得到不同的判断结果。这也恰恰是分布式健康检查的优势：它可以最准确的反映出每个负载均衡转发实例所“看到”的RS状况。

分布式健康检查方式的一个问题是：如何得到全局的RS状态表。运维人员通常都需要在管理控制台上集中查看RS的状态信息。在分布式健康检查方式下，仅从一个负载均衡转发实例获取RS状态信息显然是不合理的，很可能出现“以偏概全”的情况；而如果从所有的负载均衡转发实例读取状态信息来汇总，在转发实例数量比较多的情况下，是一种很不经济的方案；如果只选取部分转发实例的信息，选择的策略也不是很简单的，需要考虑多种因素。

分布式健康检查方式的另外一个问题是：即使发送探测请求的频度比较低，在转发实例数量比较大（如，上百个）、而下游RS的数量比较少（如，仅有几个）的时候，仍然会对RS产生很大的压力。



为了解决以上问题，可以使用”**集中式健康检查**“，要点描述如下：

- 探测器程序独立部署

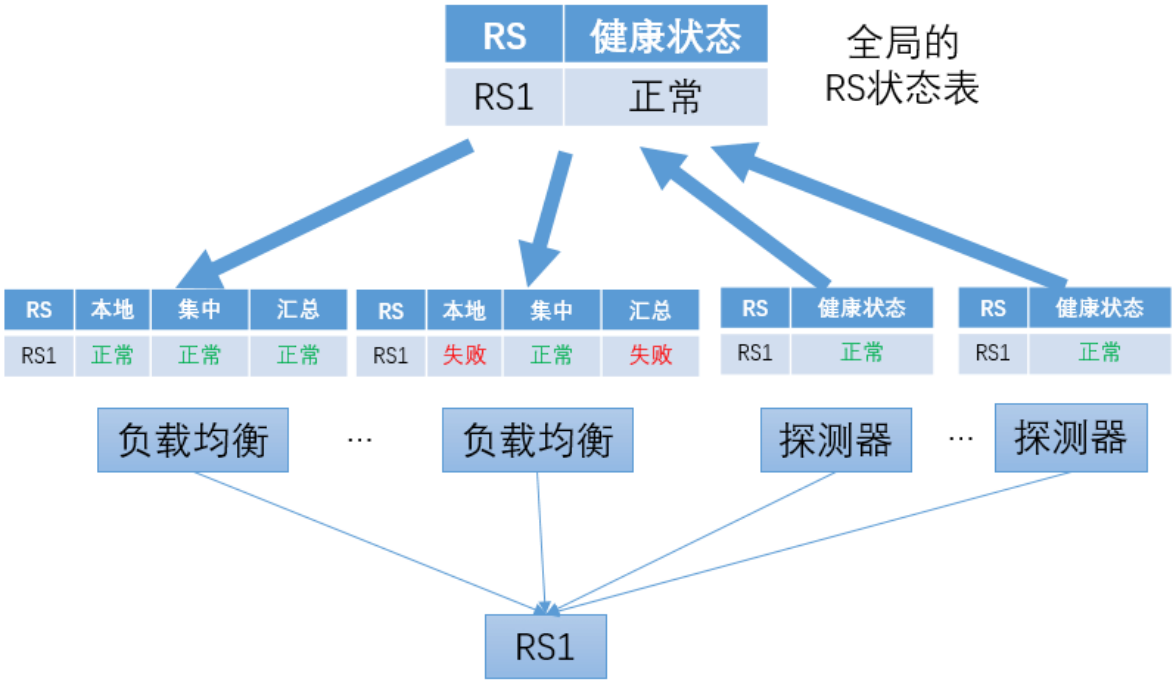
  和数量较大的负载均衡转发实例相比，探测器的实例数相对较少。比如：在每个数据中心内部署几个探测器实例。

  在探测器的部署方面，可以有一定的考虑，比如：将探测器分布在数据中心不同的网络汇聚节点下及不同的TOR下。

- 通过对探测器的状态汇总得到全局的RS状态表

- 主动探测的结果会下发至各负载均衡转发实例

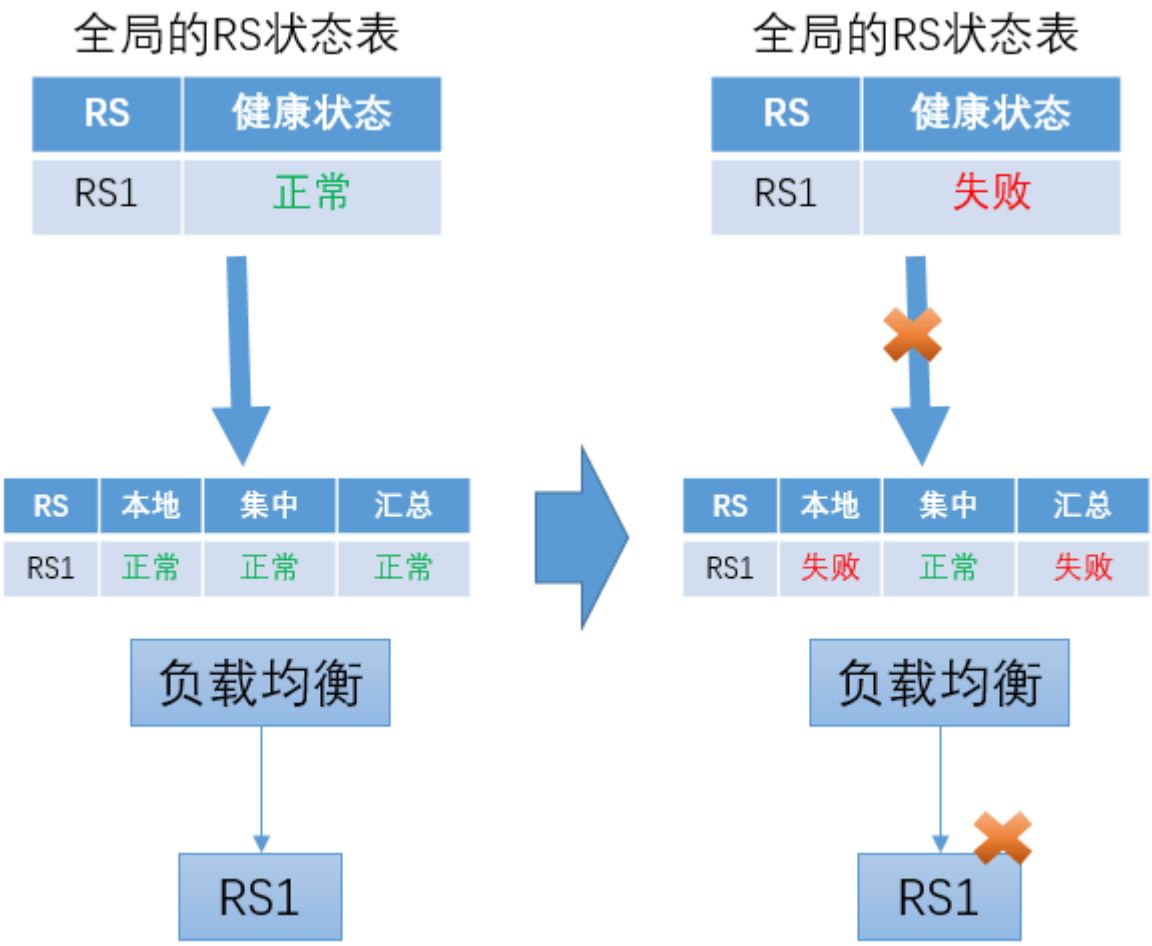
- 负载均衡转发实例综合本地被动健康检查的结果和从中心获得的主动健康检查结果，汇总得到RS的状态



负载均衡转发实例在使用从中心获得的主动健康检查结果时，要采取一定的容错机制，以防止无法及时获得主动健康检查结果的更新。可以考虑以下两种情况：

- 上次主动健康检查RS状态为“正常”，之后来自中心的更新中断

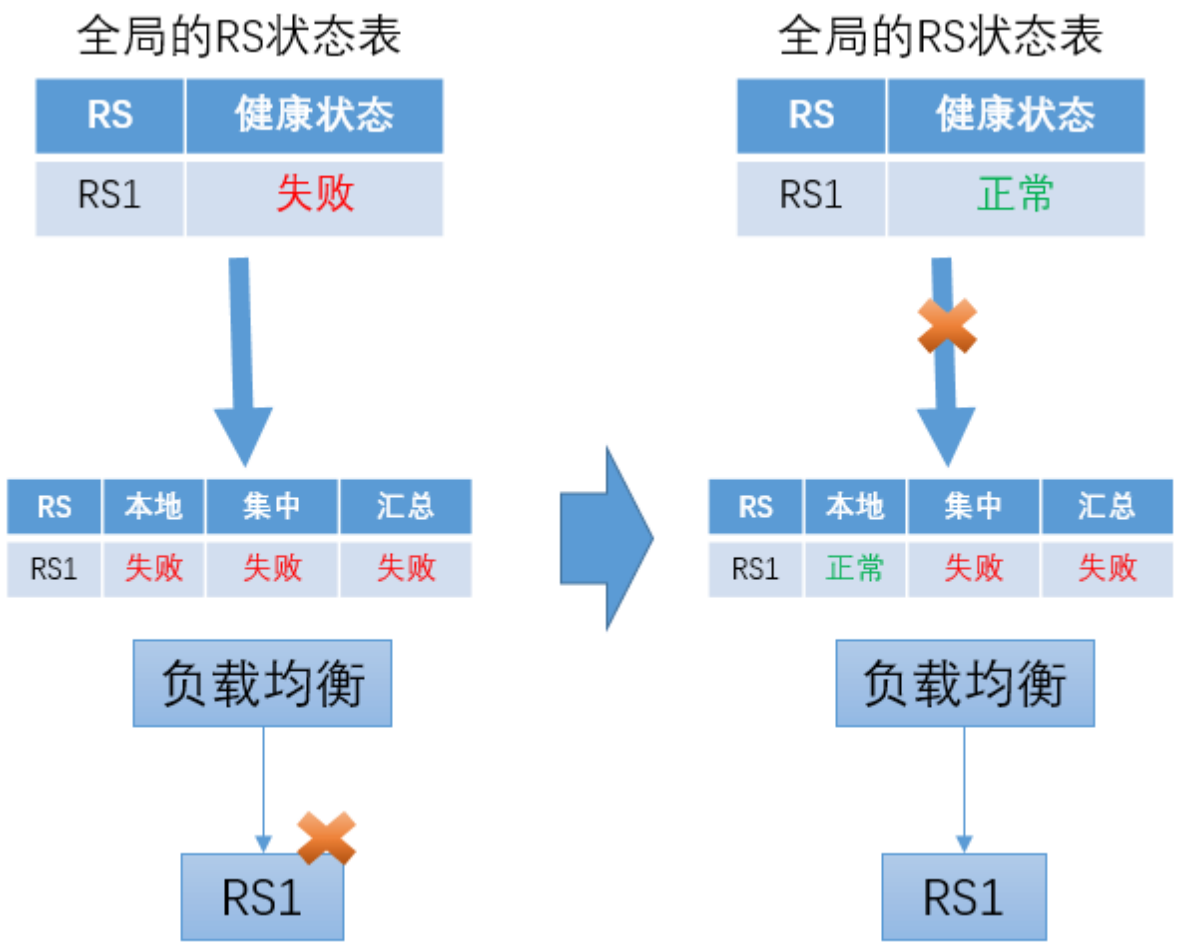
  在这种情况下，如果后续RS转变为“失败”，负载均衡转发实例可以根据本地的被动健康检查来发现，并不会导致严重的问题。



- 上次主动健康检查RS状态为“失败”，之后来自中心的更新中断

  在这种情况下，如果后续RS转变为“成功”，负载均衡转发实例继续使用之前获得的“失败”信息，从而继续判断RS为失败。这会导致始终无法将RS恢复为正常状态。

  对这个问题，可以在负载均衡转发节点增加超时机制来解决。对于从中心点获得的状态信息，在一定时间后失效。如果没有新的更新信息，则退化为只依靠本地的被动健康检查结果。



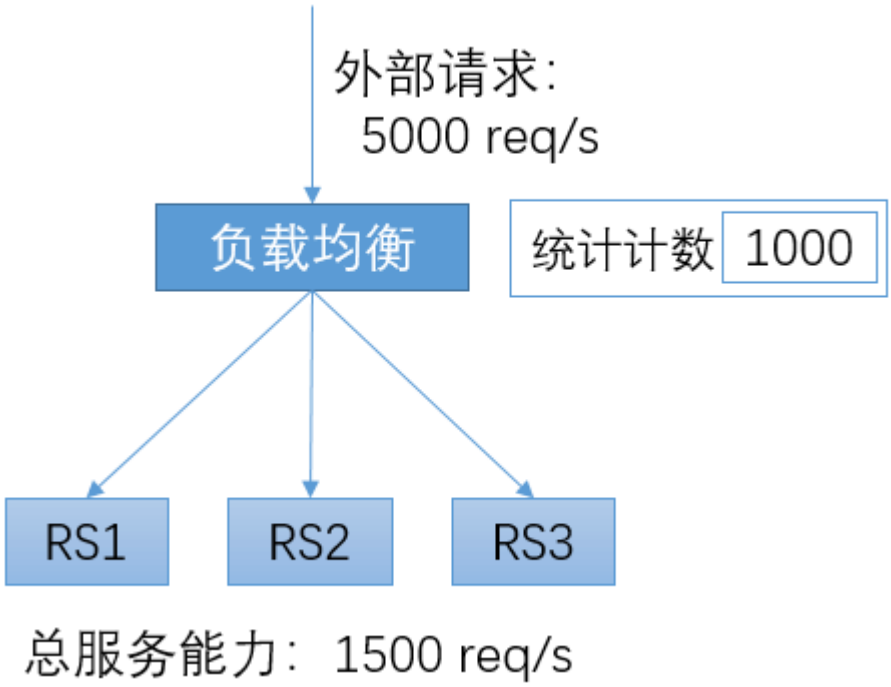
BFE的健康检查

BFE开源项目目前只支持被动健康检查。基于BFE开源项目所实现的商业版产品中包含了集中式的主动健康检查。

未来将在BFE开源项目中增加分布式的主动健康检查。

限流机制

概念



限流的场景如上图所示。由3个RS构成一个服务，总的服务能力为每秒1500个请求。而在某个时刻，外部请求的速率为每秒5000个请求。如果负载均衡直接将所有外部请求都转发给RS，RS将发生过载，从而导致对所有请求的处理都延迟增大，并导致部分请求被拒绝；严重的情况下，过载的流量可以将服务程序压垮，或发生其它不可预期的结果。在客户端程序中往往包含超时机制，延迟过大的响应从客户端看来常常被判定为失败，从而导致所有用户都感知到服务的失败。

互联网是一个非常开放的架构，无法对客户端的行为进行限制。在正常的场景下，外部请求超过后端服务能力也是经常发生的。如：某些新闻事件导致新闻网站或搜索网站的访问量突增。另外，还会出现由于恶意攻击而导致的服务过载。

负载均衡系统往往会提供限流机制，以对下游的服务进行保护。在负载均衡系统中，可以针对下游服务设置服务阈值；在外部请求超过服务阈值后，负载均衡将停止向RS的转发，而直接向客户端返回响应、或者直接拒绝请求。这样，对于在超过服务阈值前转发的请求仍然是可以正常处理的。

限流有时也被称为“熔断”，“熔断”这个词对于描述这个机制是非常形象的。

限流的配置

统计特征	统计周期	阈值	动作
www.demo.com	1秒	1200	关闭连接
api.baidu.com/api	1秒	200	返回静态结果

对于限流来说，典型的配置如上图所示，包括：

- 统计特征

可能会将域名、URL或其它更多HTTP请求中的信息作为统计的特征。

- 统计周期

  虽然可以设置为其它值，但是一般都将统计周期设置为1秒。如果设置为更长的周期，可能会出现所有阈值范围内的请求都在最开始的1秒内到达，从而将服务压垮；如果设置的更短，考虑到系统中的各种延迟，会给实现带来很高的难度。

- 阈值

  可以根据服务的容量设置限流的阈值。服务的容量可能是静态的（如：通过离线的压测得到），也可能是动态的（如：实时通过CPU等资源的情况推测）。

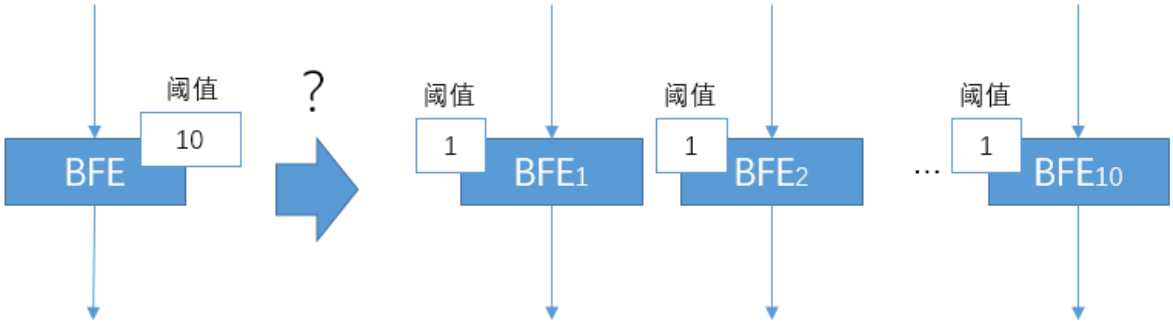
- 动作

  在统计周期内，如果请求的数量超过阈值，则会触发预先设置的动作。可能的动作包括关闭连接、展示指定网页等。

分布式限流

在限流的实现中，需要在负载均衡转发实例上对请求计数。当计数超过预定的阈值时，则执行预定的动作。

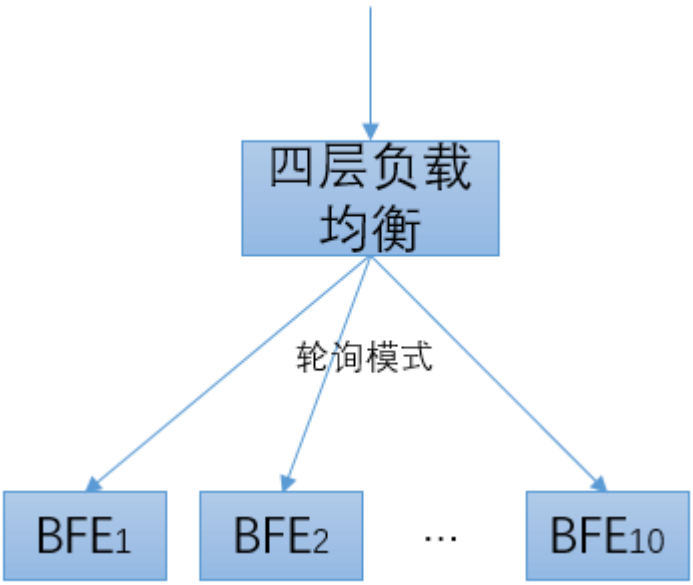
在使用软件负载均衡的场景，转发往往是由多个转发实例构成的负载均衡集群来完成的。针对限流这个功能，是否可以简单的将阈值在多个负载均衡转发实例间平均分配呢？例如，在下图中，在单机上限流的阈值为10。如果转发由10个BFE实例来完成，在每个转发实例上都配置限流阈值为1，是否可以达到总体限流阈值为10的效果呢？



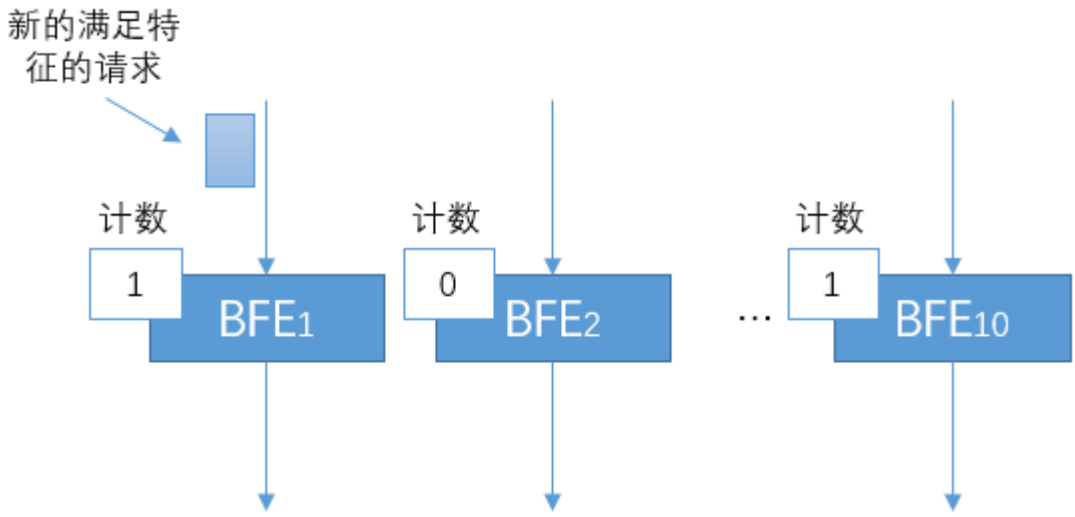
在实践中，发现最终通过的流量经常比设定的总体阈值要更小。发生这种情况的原因是：

- BFE上游使用四层负载均衡来分配流量

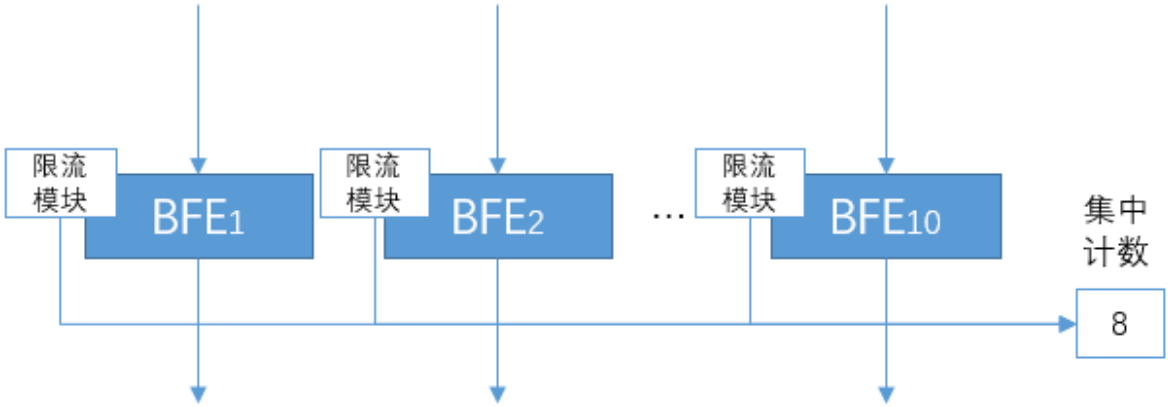
- 四层负载均衡可以实现：对于一个Virtual Server，将新建连接按照Round Robin模式均分到多个BFE实例上



- 需要注意的是：以上的均分是**针对单个Virtual Server的所有请求**，而不是**针对限流统计特征的所有请求**。
- 对于满足限流特征的请求，在多个BFE实例间的分配是不均衡的。会出现下图所示的场景：某些BFE实例的计数仍然为0，而某些BFE实例的计数已经达到阈值从而开始拒绝流量。



为了解决上面提到的问题，需要在一个BFE集群内采用集中计数的方式。当有新的请求到达某个BFE实例时，BFE的限流模块会上报给集中计数的系统（可以使用redis来实现）。如果集中的计数超过预定的阈值，则执行预定的动作。



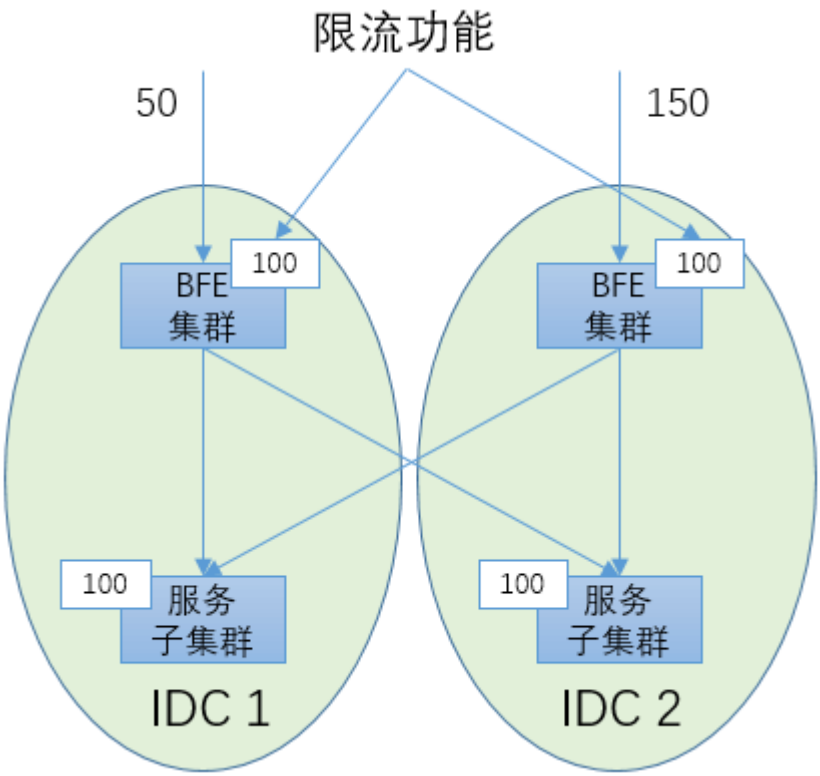
这两类限流方案也可以组合形成两级限流方案。首先通过第一级的分布式计数限流，实现粗粒度限流，并减少下一级别需处理的流量规模。然后再通过第二级的集中式计数限流，实现高精度限流。

入口限流 vs 目的限流

将限流机制用于多数据中心场景，会遇到一些新的问题，下面做一个简要的分析。

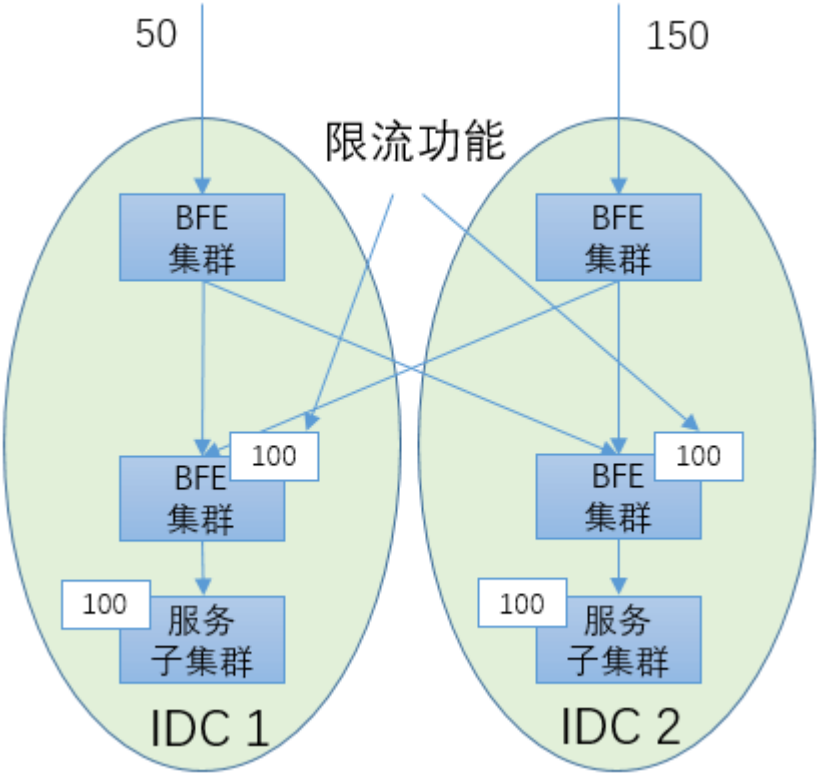
如下图所示，有2个数据中心，其中各部署了一个BFE集群用于七层负载均衡。有2个服务子集群，分别位于2个数据中心。在2个BFE集群上都开启限流功能。假设每个服务子集群的服务能力为100 req/s。那么为了保护服务子集群不过载，将每个BFE集群的限流阈值设为100是否可以呢？

其实这样是不行的。不能简单的假设到达2个BFE集群的流量是相等的。如“内网流量调度机制”中所述，外网流量会随着多种原因而发生变化，多个网络入口之间的流量关系是很不确定的。下面描述了这样一种可能：到达IDC1入口的流量为50 req/s，到达IDC2入口的流量为150 req/s。虽然从总体上看总的流量并没有超出总的服务能力，但是由于限流的配置，在IDC2的BFE集群会将超过100的流量丢弃。



发生这种问题的根本原因是：在限流的**目的**（保护服务子集群）和限流的**手段**（在网络入口处限流）之间出现了不一致。要从根本上解决这个问题，需要将“入口限流”改变为“目的限流”。具体方案如下图所示，需要使用2层的七层负载均衡集群：上面那

层BFE集群用于跨机房调度，下面那层BFE集群用于执行限流。这个方案的代价较大，需要增加一层负载均衡的资源消耗，可以在对限流准确性有较高要求的场景使用。



限流 vs 内网流量调度

从表面上看，限流和“内网流量调度”似乎有一些相似之处。内网流量调度提供了“黑洞”能力，可以主动丢弃一定比例的流量，也能起到过载保护的效果。

在过载保护的能力方面，限流和内网流量调度存在以下差异：

- 生效速度

  在流量超过阈值的情况下，限流可以在秒级内产生效果；而内网流量调度需要几十秒才能做出调整，对于秒级的突发流量无能为力

- 生效粒度

  基于计数器，限流可以精确到单个请求的粒度；内网流量调度的生效基于BFE处设置的权重，控制粒度较粗

监控机制

BFE作为一个七层负载均衡软件，需要7*24小时的持续稳定运转。为了保证系统的稳定性和正确性，对于系统的监控非常重要。下面对BFE的监控机制做一个简要的介绍。

日志监控及其问题

很多系统依赖于对外输出的错误日志来发现系统的问题，具体方法是：

- 在错误发生时，打印日志，其中包含错误的信息
- 配置监控系统，对于日志的内容进行监控，如果发现有错误的信息，则输出报警

在很多时候，不仅希望看到系统错误的情况，也希望能够看到系统的一些状态。比如：目前并发的连接数有多少，每秒处理请求的数量有多少，等等。类似这样的需求，在很多系统中也是依靠分析系统输出的日志来实现的。

基于系统日志来监控的机制，存在以下问题：

- 被监控系统的资源消耗较高

  打印日志会使用磁盘IO，是一个消耗资源很高的操作。如果是输出文本日志，日志格式化时所进行的字符串操作也是消耗CPU资源较多的。

  大家可以做这样的试验：对于BFE或者Nginx，打开或关闭访问日志的输出，会看到明显的性能变化。

- 监控系统的资源消耗较高

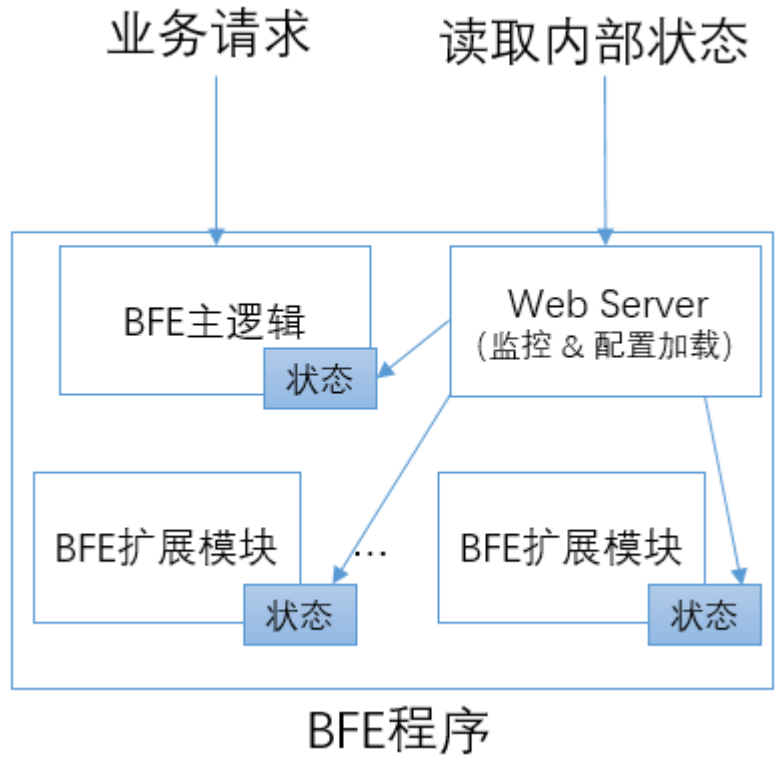
  对日志进行监控，监控系统要进行读取、解析、匹配等操作，都是资源消耗较多的。

  曾经听说过这样的案例：业务系统运行时使用了4核CPU；为了分析这个系统输出的日志，监控系统也使用了4核CPU。监控使用了和业务系统几乎一样多的资源，成本有些太高了。

- 很多状态信息并不适合都打印输出

  如果希望了解BFE内部系统间的调用处理情况，不可能将这些内部调用的情况都通过打印日志的方式输出。

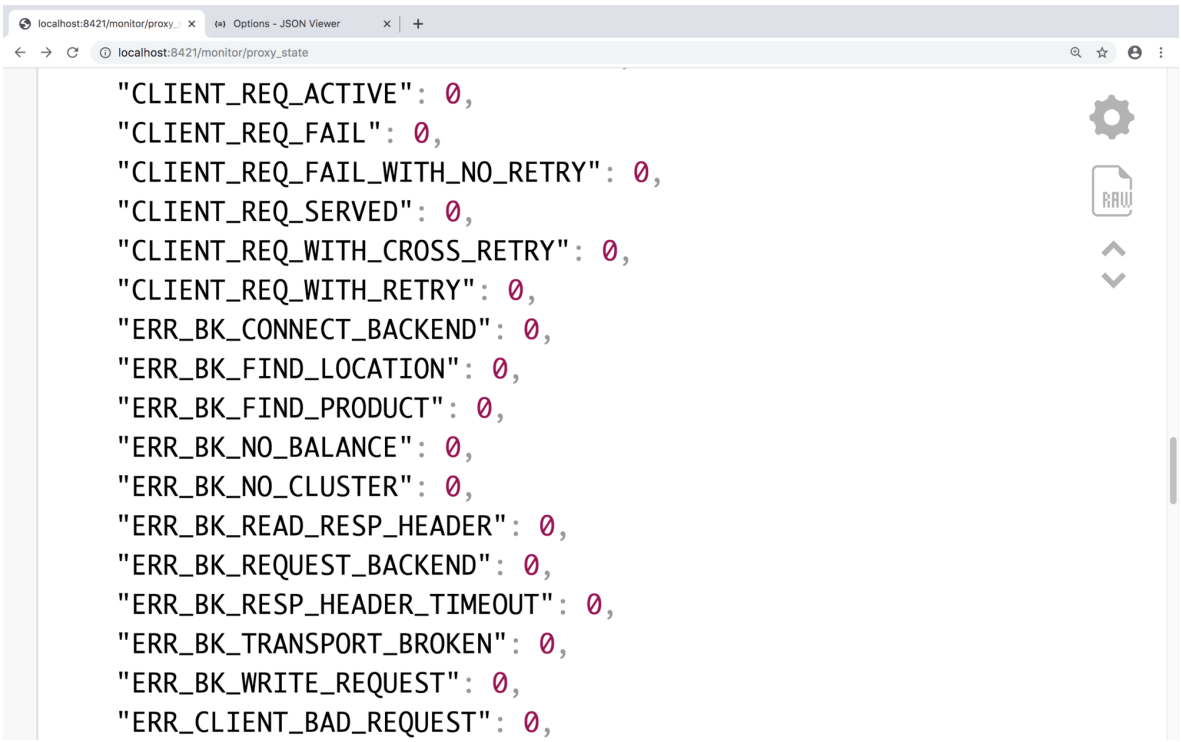
BFE的内部状态输出



为了更方便的向外展示内部的状态信息，BFE做了一些专门的设计：

- 在BFE的主逻辑和BFE的各扩展模块中，使用专门的存储来维护状态信息
- 在BFE程序中嵌入一个Web Server，用于外部读取BFE内部的状态信息及触发配置加载

下面是一个在浏览器中查看从BFE监控端口读取结果的例子。状态信息默认以JSON格式输出，每项包括状态变量名及变量的值。比如：CLIENT_REQ_ACTIVE是指当前活跃的请求数，CLIENT_REQ_SERVED是指从BFE程序启动以来，一共服务的请求数。



这个设计的好处是：

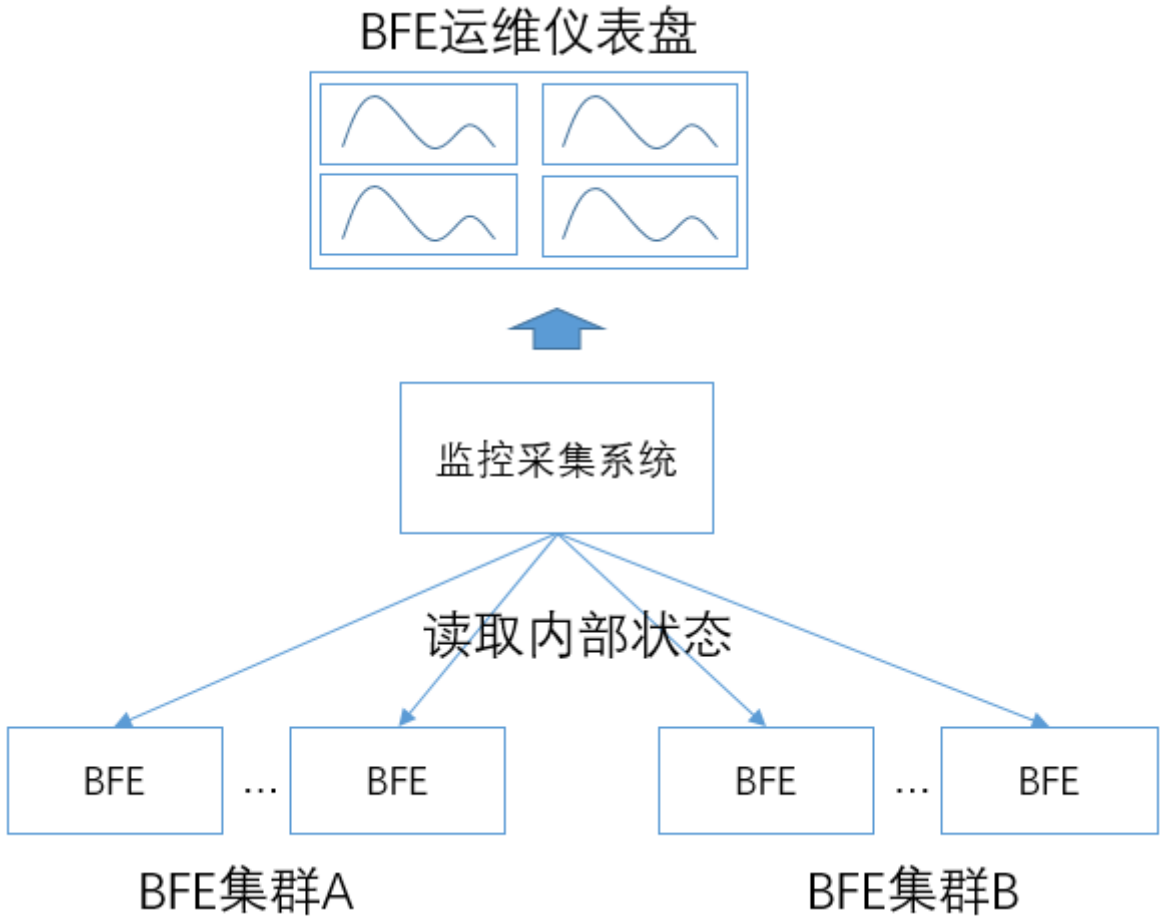
- 状态信息可以低成本的收集和汇聚

  一个状态的累加计算，成本仅仅是对BFE内存中一个变量的“加1”操作。

- 状态信息可以低成本的读取

  监控系统通过BFE对外的监控端口，一次可以读取几十甚至几百个变量。由于BFE所输出的状态信息为格式化数据，也便于监控系统对内容进行解析。

通过以上方式，BFE向外暴露数千个内部状态信息，可以反映出系统内部各方面的实时状态。使用监控系统（如Prometheus）对各BFE程序实例的状态信息做采集和汇聚，可以形成BFE集群的运维仪表盘。



统计状态和日志的配合使用

虽然监控不需要依赖日志，但是日志仍然是有用的。

- 对于一些错误，从状态信息只能看到”错误的发生”，但是无法看到“错误的细节”
- 在这种场景，在基于状态信息监控到错误的发生后，可以进一步查询对应的日志，以便进一步深入了解错误的情况

关于BFE日志的机制，可以参考“日志机制”。

Web Monitor基础库

以上所介绍的BFE相关状态输出机制已经封装为独立的基础库，命名为Web Monitor。下面介绍一下Web Monitor的设计机制和使用方法。

Web Monitor的代码位于https://github.com/baidu/go-lib 中的web-monitor目录下。

概述

Web Monitor提供Web接口，帮助持续运行的后台程序提供内部状态展示和配置热加载功能。

Web Monitor中主要提供以下3类支持：

- 专用的定制Web Server

  这个Web Server可以嵌入到后台程序中运行

- 回调接口注册

  需要外部访问状态或配置热加载的后台程序模块，可以向Web Monitor注册对应的状态展示函数或配置加载函数

- 内部状态维护

  Web Monitor为后台程序维护内部状态提供了多种形式的支持

状态变量维护

变量类型

在内部状态维护场景，考虑了以下几种场景：

- 计数器变量（Counter）：只能单向增长
- 计量变量（Gauge）：可以增加、减少，也可以直接改变取值
- 状态变量（State）：可以设置一个字符串作为状态，如“on”、“off”、“red”、“green”

以上几种变量对应的类型定义可以查看https://github.com/baidu/go-lib 中 /web-monitor/metrics目录下的counter.go、gauge.go和state.go。

差值计算

对于计数器变量，Web Monitor还提供了“获取指定时间周期内的差值”的能力。比如，对于”CLIENT_REQ_SERVED”，获取BFE程序启动后一共处理了多少请求并没有太大意义，我们更希望得到“最近20秒内服务了多少请求”。获取差值的能力实现在/web-monitor/metrics的metrics.go中。Metrics的用法如下：

(1) 定义包含统计变量的数据类型

```
import "github.com/baidu/go-lib/web-monitor/metrics"

// define counter struct type
type ServerState {
    ReqServed *metrics.Counter
    ConServed *metrics.Counter
    ConActive *metrics.Gauge
}

var s ServerState
```

(2) 定义和初始化Metrics变量

在Metrics的Init()函数中

- 第二个参数为“前缀字符串”

  Web Monitor输出有一种格式为key-value方式，在这种方式下输出时，会将“前缀字符串“放在原始的变量名称前面，以便在复杂场景时在全局区分各统计变量。在这个例子中，增加了”PROXY“前缀，以key-value方式输出时会显示为：

  ```
  PROXY_REQ_SERVED: 0
  PROXY_CON_SERVED: 0
  PROXY_CON_ACTIVE: 0
  ```

- 第三个参数为“差值计算的间隔时间”

  这个例子中设为20秒

```
// create metrics
var m metrics.Metrics
m.Init(&s, "PROXY", 20)
```

(3) 统计变量的相关操作。如：

```
// counter operations
s.ConActive.Inc(2)
s.ConServed.Inc(1)
s.ReqServed.Inc(1)
s.ConActive.Dec(1)
```

(4) 获得结果

通过调用Metrics的GetAll()接口，可以获得其中所有变量的“绝对值”；调用GetDiff()接口，可以获得其中Counter类型在20秒内的“变化值”。

```
// get absoulute data for all metrics
stateData := m.GetAll()
// get diff data for all counters
stateDiff := m.GetDiff()
```

使用案例

BFE主逻辑的统计变量定义在/bfe_server目录下的proxy_state.go中。

在扩展模块开发中也会使用状态变量的机制，可以参考”如何开发BFE扩展模块“中的说明。

延迟统计变量维护

在BFE中，也需要对一些处理的延迟进行统计，如：转发处理的延迟，HTTPS握手的延迟等。

在Web Monitor中，提供了Delay Counter的机制，以支持对于延迟的统计。

机制说明

Delay Counter支持以下能力：

- 平均延迟

  通过记录样本的数量和延迟总和，可以计算得到平均延迟

- 延迟的分布

  可以用户可以指定延迟统计分档的数量及每个分档的时间大小，可以获得落入各延迟分档的请求数量

以上这些统计数据，都是针对一定的时间周期的。Delay Counter的使用者需要指定统计的周期（如：60秒）。Delay Counter会显示“当前周期”的统计数据，如果刷新Web Monitor的接口，会发现这些统计数据在持续发生变化。从观测长期变化的角度，需要在每个周期结束后，获取其稳定不变的统计值，为此Delay Counter也会同时提供“上一周期”的统计结果。

延迟分档	延迟分布(ms)	请求个数
1	0-1	2001
2	1-2	11
3	2-3	60
4	3-4	3
5	>4	22

当前周期的统计结果(更新中)

延迟分档	延迟分布(ms)	请求个数
1	0-1	12019
2	1-2	921
3	2-3	60
4	3-4	193
5	>4	218

上一周期的统计结果

使用方法

Delay Counter的用法如下：

(1) 定义和初始化Delay Counter

```
import "github.com/baidu/go-lib/web-monitor/delay_counter"

ProxyDelay = new(delay_counter.DelayRecent)
```

```
// Init的3个参数为:  统计周期，延迟分档（毫秒），分档个数
ProxyDelay.Init(60, 1, 10)
```

(2) 增加样本值

```
ProxyDelay.AddBySub(startTime, endTime)
```

(3) 输出文本形式的结果

```
// params是由Web Monitor的Web Server传入的参数
ProxyDelay.FormatOutput(params)
```

建立专用的Web Server

BFE内嵌的监控专用Web Server，在/bfe_server的web_server.go中定义：

```
func newBfeMonitor(srv *BfeServer, monitorPort int) (*BfeMonitor, error) {
    m := &BfeMonitor{nil, nil, srv}

    // initialize web handlers
    m.WebHandlers = web_monitor.NewWebHandlers()
    if err := m.WebHandlersInit(m.srv); err != nil {
        log.Logger.Error("newBfeMonitor(): in WebHandlersInit(): ", err.Error())
        return nil, err
    }

    // initialize web server
    m.WebServer = web_monitor.NewMonitorServer("bfe", srv.Version, monitorPort)
    m.WebServer.HandlersSet(m.WebHandlers)

    return m, nil
}
```

上面的代码中，建立了维护回调函数的变量m.WebHandlers，也建立了Web Server的变量m.WebServer。

最后，启动Web Server

```
func (m *BfeMonitor) Start() {
    go m.WebServer.Start()
}
```

注册回调函数

在上面一段所调用的m.WebHandlersInit()中，既注册了用于显示内部状态的回调函数，也注册了用于动态加载配置的回调函数：

```
func (m *BfeMonitor) WebHandlersInit(srv *BfeServer) error {
    // register handlers for monitor
    err := web_monitor.RegisterHandlers(m.WebHandlers, web_monitor.WebHandleMonitor,
        m.monitorHandlers())
    if err != nil {
        return err
    }
```

```
    // register handlers for for reload
    err = web_monitor.RegisterHandlers(m.WebHandlers, web_monitor.WebHandleReload,
        m.reloadHandlers())
    if err != nil {
        return err
    }

    return nil
}
```

以上是BFE主逻辑注册回调函数的逻辑。在各扩展模块，也有独立的注册逻辑，可以参考"如何开发BFE扩展模块"中的说明。

日志机制

“打印日志”是一般程序经常使用到的机制，看起来很简单。但深入研究，会发现日志也有很多需要注意的细节。下面对BFE开源项目中的日志机制进行说明。

日志类型

在BFE中，对不同用途的日志做出明确的区分。

- 访问日志（Access Log）

  由程序处理外部请求所触发的日志。也包括由于外部建立连接所触发的日志。

- 服务日志（Server Log）

  由和处理外部请求无关的操作所触发的日志。如：配置加载，程序执行异常（非外部请求处理）等。

- 密钥日志（Key Log）
  由程序处理TLS连接所记录的TLS主密钥日志。

不同日志的使用场景有很大的差异。服务日志主要用于反映程序的运行状态，一般数据量较小，常用于系统监控、故障诊断场景。访问日志的数据量较大，可以用于业务请求的数据分析。密钥日志一般按需抽样启用，并与第三方工具配合解密及诊断密文抓包流量。由于这样的差异，不同日志应该分开输出，并使用不同的机制来做后续的处理。例如，服务日志可以和监控系统进行联动查询，用于故障精确定位；访问日志可以使用大数据分析平台来分析和存储。

在实践中，常见的错误是将访问日志混杂在服务日志中打印。尤其是对很多在处理请求过程中发生的错误，有些程序会把相关的信息打印在服务日志中。这会增大服务日志的数据量，使严重的系统错误信息被淹没在访问信息中。这样的问题应在程序编写中尽量避免。

日志打印的注意事项

对一般的程序来说，日志的输出应满足以下要求：

- 日志的输出不能阻塞程序正常的处理流程

  日志的输出涉及到磁盘IO操作，在部分场景可能出现阻塞的情况。有部分程序由于实现不当，业务主逻辑对日志输出模块为“同步调用”，在日志打印阻塞时，主逻辑也被阻塞。

  正确的方法是，将业务主逻辑对日志模块的调用修改为“异步方式”。在最坏情况下，磁盘IO操作阻塞只会导致日志打印失败，而不会阻塞业务主逻辑的执行。

- 应支持日志文件的切割和滚动覆盖

  日志文件的大小会随着程序的运行而持续增长，如果不做任何处理，会将磁盘空间用尽，从而导致系统问题。很多打印日志的基础库都支持对日志文件做定期的切割，并可以指定日志文件滚动覆盖的参数。在切割的日志文件数量超过指定的数量后，会删除历史的日志文件。

  一些程序未使用“内置”的日志基础库来实现切割和滚动覆盖的功能，而采用在crontab中定期运行脚本的方式来实现。这样的方式虽然也可以达到类似的效果，但是提高了运维的复杂性。时常出现由于遗忘增加crontab配置、或错误修改crontab

配置导致的日志超限问题。建议尽量使用“内置”的日志基础库来实现以上功能。

BFE的服务日志使用https://github.com/baidu/go-lib下的log库来输出。log库在开源的log4go基础上进行了封装和修改，增加了“按照时间切割日志”的功能。在log4go中，实现了“异步写入”的机制，日志信息首先被提交到一个队列，然后由独立的go routine读出并输出。如果队列超限，则日志信息会被丢弃。

BFE的访问日志

BFE的访问日志由BFE的扩展模块mod_access来输出。BFE的访问日志中包括“请求”（Request）和“会话”（Session）两类日志。会话对应于客户端和BFE间建立的TCP连接，一个会话中可能包含多个请求。

mod_access提供了模板配置能力，用户可以对请求日志和会话日志中输出的数据字段做定制。

超时设置

BFE中有一些涉及超时的配置，包括：

- BFE和客户端间通信的超时
- BFE和后端实例间通信的超时

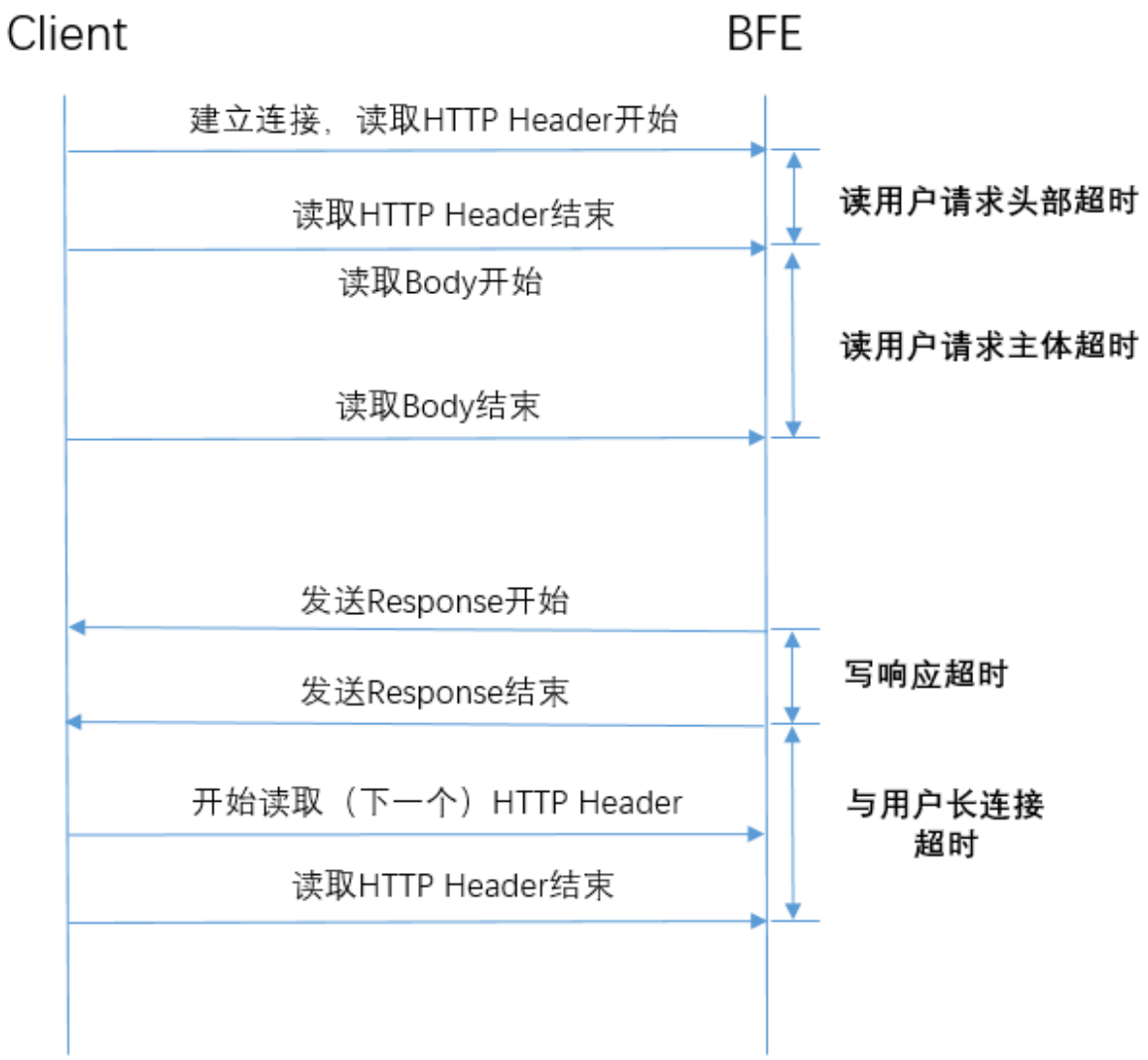
超时的设置，对于业务流量的处理非常重要。

客户端和BFE间通信的超时

定义

BFE和客户端间通信的超时包括：

- **读用户请求头部超时**：从连接建立开始，到完整读取到来自客户端请求头部
- **读用户请求主体超时**: 从完成读取请求头部，到完成读取请求主体
- **写响应超时**: 从发送响应头部开始，到将响应完全发送给客户端
- **与用户长连接超时**：从上一个请求结束，到完成读取下一个请求头部



配置方法

- **读用户请求头部超时**

  在/conf/bfe.conf中统一配置，单位为秒。这个配置只能在程序重新启动时生效，无法热加载。

```
[Server]
...
# read timeout, in seconds
ClientReadTimeout = 60
...
```

- **读用户请求主体超时**

  在/conf/server_data_conf/cluster_conf.data中，针对各集群（Cluster）独立配置，单位为毫秒。

```
{
    "Version": "init version",
    "Config": {
            "cluster_example": {
```

```
            ...
            "ClusterBasic": {
                "TimeoutReadClient": 30000,
                ...
            }
        }
    }
}
```

- **写响应超时**

  在/conf/server_data_conf/cluster_conf.data中，针对各集群（Cluster）独立配置，单位为毫秒。

```
{
    "Version": "init version",
    "Config": {
          "cluster_example": {
            ...
            "ClusterBasic": {
                "TimeoutWriteClient": 60000,
                ...
            }
        }
    }
}
```

在/conf/bfe.conf中，包含写响应超时的缺省配置。这个配置适用于不知道请求对应的集群、又需要控制与客户端超时的场景。例如：还没有执行到通过路由转发规则确定后端集群、而需要从BFE直接返回自定义响应的时候，使用缺省的超时配置。

```
[Server]
...
# write timeout, in seconds
ClientWriteTimeout = 60
...
```

- **与用户长连接超时**

  在/conf/server_data_conf/cluster_conf.data中，针对各集群（Cluster）独立配置，单位为毫秒。

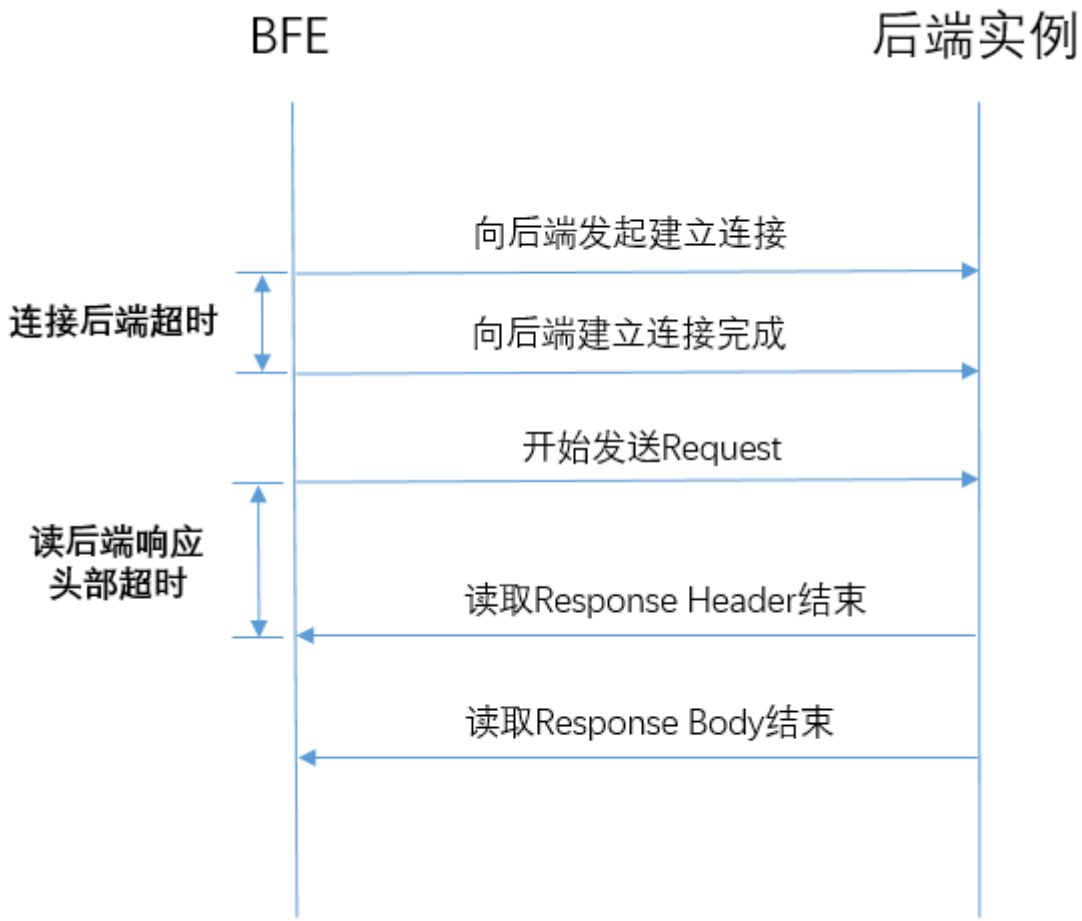
```
{
    "Version": "init version",
    "Config": {
          "cluster_example": {
            ...
            "ClusterBasic": {
                "TimeoutReadClientAgain": 30000,
                ...
            }
        }
    }
}
```

BFE和后端实例间通信的超时

定义

BFE和后端实例间通信的超时包括：

- **连接后端超时**：从向后端发起建立连接，到建立连接完成
- **读后端响应头部超时**：从BFE开始发送请求，到完成接收响应头部



配置方法

- **连接后端超时**

  在/conf/server_data_conf/cluster_conf.data中，针对各集群（Cluster）独立配置，单位为毫秒。

```
{
    "Version": "init version",
    "Config": {
            "cluster_example": {
                ...
                "BackendConf": {
                    "TimeoutConnSrv": 2000,
                    ...
                }
            }
    }
}
```

- **读后端响应头部超时**

  在/conf/server_data_conf/cluster_conf.data中，针对各集群（Cluster）独立配置，单位为毫秒。

```
{
    "Version": "init version",
    "Config": {
            "cluster_example": {
             ...
             "BackendConf": {
                 "TimeoutResponseHeader": 50000,
                 ...
             }
        }
    }
}
```

配置管理

BFE配置文件的分布

配置文件都位于/conf目录下。为便于维护，配置文件按功能分类存放在相应目录。

- BFE主逻辑的主要配置文件如下：

功能类别	配置文件目录位置	配置文件	说明
服务基础配置	/conf/	bfe.conf	包括BFE的服 缺省超时配置 还包括TLS的 HTTPS的加密 Cache的配置
接入协议配置	/conf/tls_conf/	server_cert_conf.data	服务端的证书
接入协议配置	/conf/tls_conf/	session_ticket_key.data	TLS Session
接入协议配置	/conf/tls_conf/	tls_rule_conf.data	按照租户粒度
流量路由配置	/conf/server_data_conf/	vip_rule.data	各租户的VIP
流量路由配置	/conf/server_data_conf/	host_rule.data	各租户的域名
流量路由配置	/conf/server_data_conf/	route_rule.data	各租户的分流
流量路由配置	/conf/server_data_conf/	cluster_conf.data	各集群的转发 GSLB基础配 后端基础配置
流量路由配置	/conf/server_data_conf/	name_conf.data	服务名字和服
负载均衡配置	/conf/cluster_conf/	cluster_table.data	各后端集群包 及各子集群中
负载均衡配置	/conf/cluster_conf/	gslb.data	用于配置各集

- BFE扩展模块的配置文件

  出于便于管理的目的，BFE各扩展模块的配置文件和BFE主逻辑的配置文件分开存放。对于每个扩展模块，有一个独立的配置文件目录，位于”/conf/mod_/“目录下。如：/conf/mod_block/目录下是mod_block的配置文件。

常规配置 vs 动态配置

在BFE中，将配置分为“常规配置”和“动态配置”：

- 常规配置

  仅在程序启动时生效。在BFE中，常规配置一般基于INI格式，常规配置文件名一般使用”.conf”的后缀。

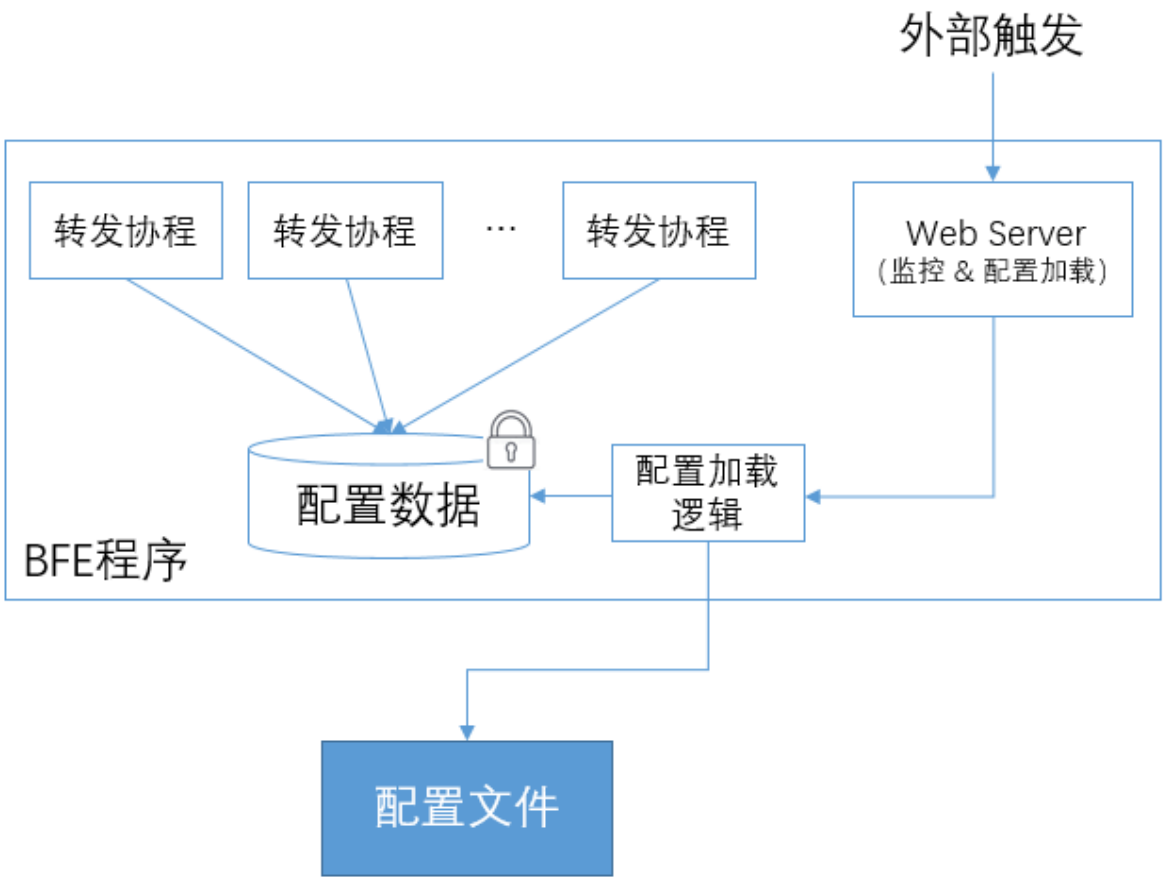
- 动态配置

  可在程序执行过程中动态加载。在BFE中，动态配置一般基于JSON格式，兼顾程序读取和人工阅读的需求。

  动态配置文件名一般使用“.data”的后缀。

动态配置的实现机制

BFE中动态配置的实现机制如下图所示，主要包括“配置加载”和“配置生效”两方面。



配置加载

在“监控机制”中介绍过，在BFE程序中嵌入一个Web Server，用于外部读取BFE内部的状态信息。这个Web Server也用于触发配置的动态加载。

例如：通过访问 http://127.0.0.1:8421/reload/gslb_data_conf，可以触发gslb.data的重新加载。

出于安全的考虑，仅当从部署BFE程序的同服务器发起对这个接口的访问时，才会通过。这个限制位于Web Monitor的实现代码中：

```
// source ip address allowed to do reload
var RELOAD_SRC_ALLOWED = map[string]bool{
    "127.0.0.1": true,
    "::1":       true,
}
```

如果从这个范围外的地址发起访问，则会返回类似下面这样的错误

```
{
    "error": "reload is not allowed from [xxx.xxx.xxx.xxx:xxx]"
}
```

通过在嵌入Web Server中注册的回调函数，配置加载逻辑会执行以下逻辑：

- 配置文件的读取
- 配置文件的解析和正确性检查
- 运行态配置信息的更新

配置生效

在BFE中，程序并行处理基于Go语言提供的"Go协程"（Go routine）。BFE使用的是"单进程内多协程"的机制，不需要考虑考虑多进程通信，只需要考虑协程间的共享数据。在多协程间的共享数据方面，和多线程机制类似，可以访问位于同一进程内的共享数据，并可以使用"锁"来做互斥访问。由于协程和线程实现机制的不同，"协程锁"的开销要远远小于"线程锁"。

在BFE中，从文件加载的配置信息会被保存在一个受到协程锁保护的临界区中。负责转发的协程在使用配置数据时，需要通过特定的接口来从临近区中读取；配置加载逻辑在更新配置信息时，也通过特定的接口来操作。

以mod_block为例，其中的ProductRuleTable用于保存各租户的配置信息。为了读取配置信息，提供了如下接口：

```
func (t *ProductRuleTable) Search(product string) (*blockRuleList, bool) {
    t.lock.RLock()
    productRules := t.productRules
    t.lock.RUnlock()

    rules, ok := productRules[product]
    return rules, ok
}
```

为了更新配置信息，提供了如下接口：

```
func (t *ProductRuleTable) Update(conf productRuleConf) {
    t.lock.Lock()
    t.version = conf.Version
    t.productRules = conf.Config
    t.lock.Unlock()
}
```

以上这两个接口，使用读写锁来保护。并且，在临界区中的操作都尽量简单，以降低对多个处理协程间并行度的影响。

HTTPS优化机制

HTTPS化背景及必要性

伴随着互联网的飞速发展和用户规模的迅猛增长，互联网长期面临着严峻的安全威胁和数据隐私风险。围绕HTTP流量的黑产在非法牟利的同时，对互联网用户体验及安全隐私带来严重影响，也给互联网服务提供商的声誉和利益带来巨大损失。使用HTTP协议面临的典型问题包括：

- **内容篡改**：用户访问的页面内容被恶意篡改，例如搜索结果页链接被恶意修改、下载页面安装包被恶意替换、浏览页面植入大量广告等
- **隐私泄露**：用户网络活动行为被嗅探，个人数据被泄漏及利用，受到垃圾广告或诈骗电话干扰等
- **流量劫持**：用户访问被劫持到钓鱼网站，用户账号信息在诱导下被窃取等

HTTPS相比HTTP协议在底层使用了TLS协议传输，提供完整性、私密性、身份认证机制，可以保障互联网用户流量的接入安全。国内外诸多大型互联网公司已经全面支持HTTPS。浏览器厂商、移动应用商店等生态厂商也在加速推进HTTPS。例如：

- Google Chrome浏览器在HTTP域名输入框前增加“不安全”的提示。Google正推动 Chrome 浏览器将 HTTPS作为默认设置（而非 HTTP）。用户直接输入域名后，Chrome 将首先尝试使用 HTTPS协议访问。
- Apple iOS10 ATS (App Transport Security) 策略强制要求所以在iOS 应用商店上架的应用都必须支持HTTPS。

HTTPS化的挑战

对于一个中大型的网站而言，全面HTTPS化除了需完成网站页面HTTPS改造，还面临以下重要问题：

- **访问速度问题**：HTTPS相比HTTP一般会额外引入1～2个RTT。当然，这并不包括一些情况下用户首先访问HTTP再跳转到HTTPS的延迟，以及HTTPS证书状态检查所引入的延迟。在移动网络环境下往返延迟往往更大，还会带来更明显的影响。
- **性能及成本问题**：协议握手及数据加密传输引入的密码学计算，带来不可忽视的性能开销。尤其是TLS握手过程中的非对称计算，是性能开销的主要来源。以Nginx为例，在短连接及完全握手情况下，HTTP相比HTTPS吞吐高达10倍。
- **安全性问题**：正确的部署及保障HTTPS安全性，需掌握一定的安全领域知识及最佳实践。运维人员往往容易在HTTPS部署时留下安全隐患，无法满足业务运营所要求的安全合规标准。
- **可用性问题**：HTTPS生态更为复杂，第三方CA成为网站稳定性的一个新依赖。同时，由于客户端的多样性，端协议实现的兼容性及缺陷，也可能导致服务访问异常。

HTTPS中的优化机制

常见的HTTPS优化手段简要说明如下。

访问延迟优化

HTTPS引入总延迟大小取决于往返延迟大小和往返交互次数。从用户访问流程角度看，HTTPS交互延迟既包括建立安全会话的延迟，还包括从HTTP向HTTPS跳转延迟及HTTPS证书状态检查延迟。相应的，我们可以通过如下多种方法来优化访问延迟。

优化方法	具体机制

优化方法	具体机制
降低往返延迟	通过在边缘节点完成TLS握手
减少交互次数	通过TLS会话复用握手来减少交互次数，其中TLS 1.2需要2个RTT(含TCP握手)，而TLS 1.3只需1个RTT(含TCP握手)，使用QUIC只需0个RTT。此外，可以通过HSTS机制或重定向缓存机制，优化从HTTP再跳转到HTTPS的延迟；通过OCSP Stapling机制优化证书状态检查延迟。
隐藏交互延迟	通过启发式预建立连接，来屏蔽建连延迟的影响

非对称计算性能优化

HTTPS服务端在通信过程中，非对称密码学计算是性能开销的主要来源。非对称密码学算法安全长度的进一步提升还会加剧这个问题。例如自2010年起主流CA已停止签发不安全的 RSA 1024长度证书，并使用RSA 2048长度证书。针对非对称计算性能的优化有如下多种方法。

优化方法	具体机制
减少非对称计算次数	提升连接复用率、 会话复用率来减少TLS完全握手及引入的非对称计算次数
优化非对称计算性能	通过硬件加速卡或支持密码学计算指令的CPU， 提升非对称算法计算性能
优先使用高性能算法	自适应优先使用ECC证书而不是RSA证书 （不具备硬件加速的场景）

安全性评估及巡检

HTTPS安全性风险来源于CA基础设施、协议及算法漏洞、HTTPS部署配置、HTTP应用等。有一些公开服务支持自动评估HTTPS站点等安全性，并提示潜在的安全漏洞。

此外，HTTPS服务提供商需要制定并应用HTTPS部署最佳实践及规范。例如SSLLabs编写的《SSL/TLS部署最佳实践》，NIST发布的《保护Web事务的安全：TLS服务端证书管理》等。

同时，需建立关于HTTPS的监控机制。包括HTTPS证书监控、HTTPS混合内容监控、安全Cookie监控、HTTPS安全漏洞扫描等。

稳定性风险应对

通过灰度机制及冗余机制，来支撑业务线构建变更风险小、止损速度快、更稳定可靠的HTTPS服务。相关机制（控制HTTP向HTTPS的灰度跳转、HTTPS证书灰度更新等）具体说明详见下文。

BFE相关增强机制

为更好适应大型站点的需求，BFE相比常见开源方案进行了一些针对性改进，下文针对一些差异化的机制简要介绍如下。

分布式TLS会话缓存

BFE支持将TLS会话状态，存储在分布式Cache集群中。客户端与BFE集群中任一实例连接后，可基于Session ID完成会话复用握手，从而提升在集群部署模式下的TLS会话复用率。

TLS会话状态格式

在上文分布式TLS会话缓存中，存储的会话状态可以采用两种格式：原始格式及OpenSSL格式。如果是BFE新用户，可优先使用原始格式，这种格式更紧凑可降低分布式缓存的存储开销；如果是需要替换基于OpenSSL反向代理，可以配置OpenSSL格式，这样两种程序可兼容读取对方保存的会话状态，实现平滑迁移而不影响会话复用握手。

Session Ticket密钥更新

如果Session Ticket key泄漏将无法保障前向安全性。基于Session Ticket会话复用的连接，如果恶意攻击者预先录制流量，后续使用泄漏的Session Ticket key，可以成功计算出会话密钥并解密还原出明文信息。

为了避免以上问题，应定期更新Session Ticket key文件。BFE支持热加载并更新Session Ticket key，同时无需进程热重启导致长连接中断。

OSCP Staple的自适应容错

OCSP Staple文件具有一定的有效期。如果握手过程中服务端意外发送了一个过期的OCSP Staple文件，可能导致握手失败。在一个大规模的分布式集群环境中，由于流程或机制原因导致OCSP Staple未及时成功更新或遗漏更新，这样的情况并不罕见。

BFE支持自适应处理即将过期的OCSP Staple文件。在OCSP Staple即将过期时，BFE将自动降级并停止发送OCSP Staple文件，避免潜在的握手失败影响。

非对称密码学计算优化

非对称密钥算法在TLS握手过程中，一般用于身份认证及密钥交换。目前主流的非对称算法包含RSA/ECC。由于CA根证书兼容性原因，RSA类型证书依然是目前最广泛使用的证书。目前RSA算法建议的安全长度是2048位，相比等价强度的256位ECC算法而言，RSA算法的性能开销要远大于ECC。

如果用户流量全部来自自有移动端，或握手报文满足特定特征，可通过选择ECC类型证书及私钥，来减少非对称密码学计算引入性能开销。

如果用户流量大量来自低端客户端，则需要使用RSA类型证书及私钥以保证兼容性。通过使用支持RSA算法的硬件加速卡，可以大幅降低RSA计算引入性能开销。

基于硬件加速的方案一般有两种类型：

- **同机模式**：同机部署支持RSA加速的CPU或部署RSA硬件加速扩展卡
- **远程模式**：远程访问具备RSA硬件加速条件的非对称计算服务

远程模式相比同机模式的优势是：

- 不必全面升级现有的转发机器，降级升级周期及成本
- 可以弹性适配BFE转发机器的计算需求，避免硬件加速卡资源浪费（注：基于硬件加速卡）
- 可以为密钥提供集中式、基于硬件的安全保护机制（注：基于CPU特殊指令）

由于远程硬件加速服务可用性难以达到100%，为避免引入依赖远程硬件加速服务降低BFE整体可用性，在访问远程硬件加速卡出现偶发异常时，通过自动降级使用本地计算来消除影响。

多证书选择

在多租户的环境中，需要为不同的租户使用不同的证书。同一个租户由于业务原因（例如品牌考虑），也可能使用了多个证书。

在实际部署中，BFE为不同租户提供了不同的VIP，可以为不同的VIP配置不同的证书。

内存安全问题

Google工程师统计发现现在Chrome代码库中所有严重的安全漏洞，70%都是内存管理的安全漏洞。微软工程师也宣称，微软过去12年的安全更新大概70%都是在解决内存安全漏洞。OpenSSL著名的心脏出血漏洞(Heatbleed Bug)被称为互联网史上最严重的安全漏洞之一，由于内存信息越界访问导致内存中敏感信息泄漏，波及了大量常用网站和服务。

受益于Go语言内置内存安全特性，BFE可以避免常见C语言缓冲区溢出内存问题所引发的安全问题。

TLS安全等级

正确配置服务的各项TLS/SSL参数（协议版本、加密套件）要求运维人员对TLS/SSL安全有较深入的理解。为降低管理员误配置引入部署安全风险，BFE提供了四种TLS安全等级，具体如下。不同安全等级下BFE所支持的TLS协议版本及加密套件是不同的。

安全等级	说明
等级A+	安全性最高、兼容性最低
等级A	安全性较高、兼容性一般
等级B	安全性一般、兼容性较高
等级C	安全性最低、兼容性最高

相应的，不同安全等级具有不同的安全性及兼容性，适用不同的业务场景。例如，安全等级A+仅支持TLS 1.2及以上版本和安全强度更高的加密套件。等级A+适用于要求PCI DSS级别安全合规的业务场景，例如金融支付业务。

各安全等级的协议和加密套件的详细定义，见“配置HTTPS服务”中的说明。

密钥安全存储

使用硬件方案（HSM: Hardware Security Module）在硬件内部创建及保存私钥是最安全的方案。这种方案中涉及私钥相关的计算操作，由HSM硬件直接完成。私钥永远无法脱离HSM，也无法通过物理方式来提取。

如果不具备使用硬件的场景下，可以对密钥进行加密保护并定期更换密钥，在一定程度控制密钥泄漏的风险及影响。目前，CA厂商开始缩短签发证书的有效期，不再签发2年及以上有效期证书。

TLS/SSL JA3指纹

BFE支持根据ClientHello消息中特征，计算TLS/SSL客户端的指纹。BFE采用了JA3算法，可简单高效的识别客户端程序，并与业务层联动进行反爬取或反作弊。

在BFE中可以通过mod_header模块，在请求头部中携带JA3指纹传递给下游。具体用法参见开源BFE用户手册mod_header模块相关说明。

TLS/SSL可见性

HTTPS化对旁路式流量攻击检测以及复杂网络问题诊断带来了挑战。旁路式攻击检测系统由于无法处理密文流量将无法有效工作。另外，研发运维人员有时依赖于对密文形式的网络抓包文件进行解密及分析。BFE可与旁路式攻击检测系统配合，通过安全共享会话密钥支撑其实现传输层、安全层、应用层等攻击特征分析。同时，在BFE还可选择性将TLS会话主密钥写入到key.log日志文件中，并供wireshark软件分析加密网络报文。

HTTP灰度跳转HTTPS

由于HTTPS改造的复杂性，从HTTP迁移到HTTPS需要一套灵活的灰度机制来控制跳转。BFE支持实现细粒度的策略，例如区分域名、客户端、用户地域等，设置灰度跳转策略。

证书灰度更新

证书变更是一个高风险的操作。证书链配置错误、证书链中级CA证书发送变化、证书中缺失必要扩展、证书格式不规范等，都可能触发兼容性问题，并导致证书更新后部分用户访问异常。可靠控制变更风险、快速发现潜在问题，对普通运维人员而言不可或缺。

BFE支持按用户抽样灰度更新证书，并支持实时统计新旧证书握手成功率变化，可以快速拦截在小流量阶段，难以通过总流量波动发现的异常。

多CA互备证书

CA成为HTTPS网站稳定性风险的一个新的来源。CA厂商的选型是一个关键的问题。除了考量CA厂商的证书成本，还需考量CA厂商的根证书兼容性、OCSP服务稳定性、安全合规历史记录、是否是核心主营业务等。

随着HTTPS的大规模普及推广，近年来主流CA厂商引发的HTTPS重大故障并不罕见。例如，2016年GlobalSign由于OCSP服务故障影响了多个知名的大型网站。2018年全球最大的CA厂商Verisign由于安全违规，Chrome/Firefox/Safari等浏览器开始停止信任Symantec签发证书。

HTTPS站点在有条件情况下可以通过签发多个CA厂商的证书实现冗余互备。当某个CA厂商证书访问连通率异常时，可以快速切换迁移至其它CA证书止损。

信息的透传

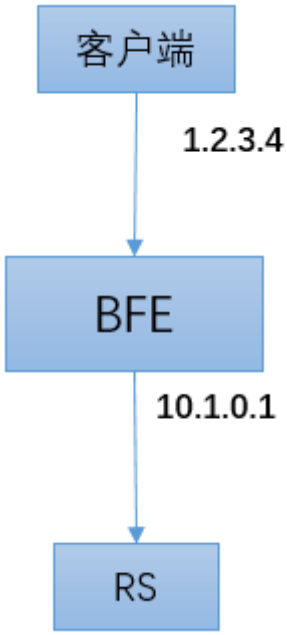
作为一个HTTP反向代理，BFE除了转发原始的HTTP请求之外，还会通过在HTTP Header中增加字段的方式向后端或用户传递一些额外的信息。

客户端IP地址的透传

需求来源

在经过BFE转发后，RS无法获得原始的客户端IP地址，而只能获得BFE实例的IP地址。

如下图所示，客户端的IP地址为1.2.3.4，在经过BFE转发后，BFE和RS建立了新的TCP连接，RS只能看到BFE的地址10.1.0.1。



很多应用都需要获取请求原始的IP地址。需要提供机制将原始的IP地址传递到RS。

透传方案

BFE在扩展模块mod_header中默认提供了捎带客户端IP地址和端口的功能。只要在BFE启动时配置加载mod_header，在转发后请求中就会包含这两个信息。

在经过BFE转发后，在请求头部会增加2个字段：

- X-Real-Ip：用于传递原始的客户端IP地址
- X-Real-Port：用于传递原始的客户端端口

有些人可能会考虑从“X-Forwarded-For”来获取客户端的IP地址。BFE使用独立定义的“X-Real-Ip”是为了避免“X-Forwarded-For”被伪造。如果请求在到达BFE时已经包含了“X-Real-Ip”字段，BFE会将这个字段的值重写为BFE所见的客户端IP地址，从而避免这个字段被伪造。

其它信息的透传

除了客户端的IP地址和端口外，mod_header也提供对请求中加入其它信息的能力。

租户配置表

在mod_header中，可以对每个租户提供一张配置表，每条配置包括：

- 匹配的条件：使用“条件表达式”（见“BFE的路由转发机制”）来描述
- 执行的动作列表：在命中匹配条件后，可以执行1至多个动作

  可执行的动作见下面的介绍。
- last：如果为true，则直接返回，不再匹配下面的规则

  和“转发表”（见“BFE的路由转发机制”）中“只会命中一次”的机制不同，在mod_header中一个请求可能同时命中多条规则，并执行多条规则对应的动作。在希望继续匹配其它规则的时候，可以将last设置为false。

condition	actions	last
req_path_prefix_in("/header", false)	[action1, action2]	true
req_path_prefix_in("/other", false)	[action3, action4]	false

可能的动作

在mod_header中，对于请求和响应都可以执行设置、添加或删除操作。

在设置或添加的时候，需要提供头部字段的名称和取值。

动作名称	含义	参数列表说明
REQ_HEADER_SET	设置请求头	HeaderName, HeaderValue
REQ_HEADER_ADD	添加请求头	HeaderName, HeaderValue
REQ_HEADER_DEL	删除请求头	HeaderName
REQ_HEADER_RENAME	重命名请求头	HeaderName, HeaderName2
RSP_HEADER_SET	设置响应头	HeaderName, HeaderValue
RSP_HEADER_ADD	添加响应头	HeaderName, HeaderValue
RSP_HEADER_DEL	删除响应头	HeaderName
REQ_HEADER_RENAME	重命名响应头	HeaderName, HeaderName2

例子

下面是mod_header配置的一个例子。

针对example_product这个租户，只配置了一条规则。如果命中规则，则顺序执行3个动作，对请求头做2次设置，对响应头做1次设置。

在请求头中透传信息时，用户可以根据自己的需要来设置头部字段的名称。

```
{
    "Version": "20190101000000",
    "Config": {
        "example_product": [
            {
                "cond": "req_path_prefix_in(\"/header\", false)",
                "actions": [
                    {
                        "cmd": "REQ_HEADER_SET",
                        "params": [
                            "X-Bfe-Log-Id",
                            "%bfe_log_id"
                        ]
                    },
                    {
                        "cmd": "REQ_HEADER_SET",
                        "params": [
                            "X-Bfe-Vip",
                            "%bfe_vip"
                        ]
                    },
                    {
                        "cmd": "RSP_HEADER_SET",
                        "params": [
                            "X-Proxied-By",
                            "bfe"
                        ]
                    }
                ],
                "last": true
            }
        ]
    }
}
```

内置变量

在上面的例子中，在设置"X-Bfe-Log-Id"和"X-Bfe-Vip"时，使用了内置的变量。在mod_header中还提供了其它内置变量，可以在设置Header时使用。

变量名	含义	依
%bfe_client_ip	客户端IP	
%bfe_client_port	客户端端口	
%bfe_request_host	请求Host	
%bfe_session_id	会话ID	

变量名	含义	需
%bfe_log_id	请求ID	根
%bfe_cip	客户端IP (CIP)	
%bfe_vip	服务端IP (VIP)	
%bfe_server_name	BFE实例地址	
%bfe_cluster	目的后端集群	
%bfe_backend_info	后端信息	

%bfe_ssl_resume	是否TLS/SSL会话复用	
%bfe_ssl_cipher	TLS/SSL加密套件	
%bfe_ssl_version	TLS/SSL协议版本	
%bfe_ssl_ja3_raw	TLS/SSL客户端JA3算法指纹数据	
%bfe_ssl_ja3_hash	TLS/SSL客户端JA3算法指纹哈希值	
%bfe_protocol	访问协议	
%client_cert_serial_number	客户端证书序列号	
%client_cert_subject_title	客户端证书Subject title	
%client_cert_subject_common_name	客户端证书Subject Common Name	
%client_cert_subject_organization	客户端证书Subject Organization	
%client_cert_subject_organizational_unit	客户端证书Subject Organizational Unit	
%client_cert_subject_province	客户端证书Subject Province	
%client_cert_subject_country	客户端证书Subject Country	
%client_cert_subject_locality	客户端证书Subject Locality	

操作篇

BFE服务的安装部署

本章将对BFE的下载安装方式进行具体介绍，帮助读者了解如何运行BFE服务。

BFE安装方式

BFE支持多种方式进行安装，本章将介绍以下三种方式：

- 软件包下载安装
- 源代码编译安装
- docker方式安装

软件包方式下载安装

获取BFE软件安装包

BFE的安装软件包可以直接从BFE项目在github.com的页面中下载。
对于不同的操作系统(Linux/MacOS/windows)，页面上都提供了相应的软件包。

下载地址如下：

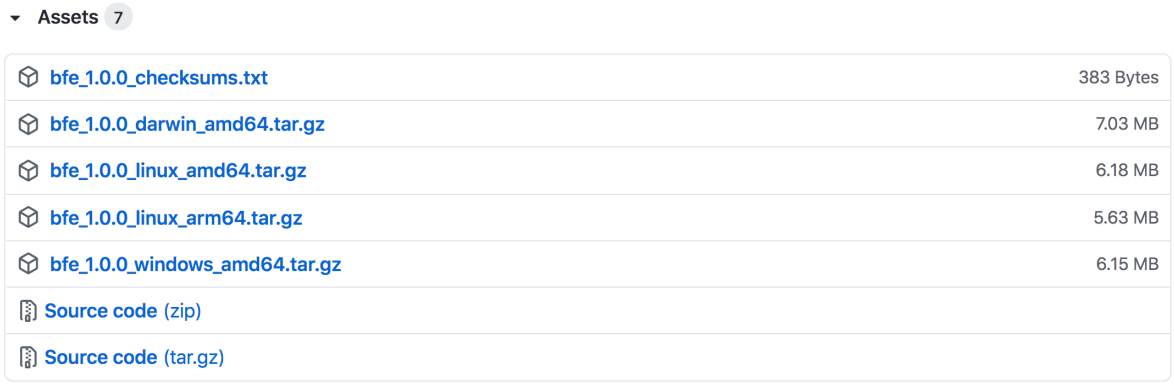
```
https://github.com/bfenetworks/bfe/releases
```

读者可根据操作系统类型，选择所需的软件版本。

下载BFE软件包

以Linux环境为例，下载BFE的1.0.0版本。

在上述下载页面上，找到”BFE v1.0.0”，点击展开“Asserts”，如下图：



点击下载名为”bfe_1.0.0_linux_amd64.tar.gz”的压缩包。

BFE软件包中的内容

解压下载的文件bfe_1.0.0_linux_amd64.tar.gz：

```
sh-4.2# tar zxf bfe_1.0.0_linux_amd64.tar.gz
sh-4.2#
sh-4.2# cd bfe_1.0.0_linux_amd64
sh-4.2#
sh-4.2# ls
CHANGELOG.md    LICENSE        README.md    bin        conf
```

其中包括了两个主要目录：

- bin ： 该目录包含可执行程序bfe。

```
sh-4.2$ ls bin
bfe
```

- conf： 该目录中包含了bfe的配置文件。其中，bfe.conf为BFE的主配置文件。

```
sh-4.2$ ls conf
bfe.conf         mod_block         mod_header         mod_static         server_data_conf
cluster_conf         mod_compress         mod_key_log         mod_tag             tls_conf
mod_access         mod_cors         mod_markdown         mod_trace
mod_auth_basic         mod_doh             mod_prison         mod_trust_clientip
mod_auth_jwt         mod_errors         mod_redirect         mod_userid
mod_auth_request     mod_geo             mod_rewrite         mod_waf
```

运行BFE服务

执行如下命令，在系统后台启动一个BFE实例。该实例使用了缺省配置启动。

```
sh-4.2# cd bin
sh-4.2# ./bfe -c ../conf -l ../log &
[1] 31024
```

检查端口8080，可以发现已经处于listen状态

```
sh-4.2$ ss -nltp | grep 8080
0      2048                                *:8080                             *:*      users:(("bfe",31024,10))
```

停止BFE服务

如需要停止BFE服务，直接执行kill命令:

```
sh-4.2$ kill 31024
sh-4.2$
[1]+  Done                    ./bfe -c ../conf -l ../log
```

在MacOs或windows中下载安装BFE

与Linux系统下安装过程相似：在下载页面 *https://github.com/bfenetworks/bfe/releases* 中下载对应操作系统版本的BFE安装包后，就可启动BFE服务。

源代码编译方式安装

BFE源代码完全在github.com上开源，用户也可以通过自行编译源代码的方式进行适配安装。

环境准备

- 环境要求
  - golang 1.13+
  - git 2.0+
- Go语言环境准备

  下载地址为 *https://golang.org/dl/* 或者 *https://golang.google.cn/dl/*。

  在下载页面，用户可以根据使用的操作系统环境，下载相应的版本。
  下载后，按照 *https://golang.org/doc/install* 或者 *https://golang.google.cn/doc/install* 的说明，进行安装。
- git安装

  用户需要安装git命令，具体安装不再赘述。

BFE源码下载

通过 *git clone* 命令，下载BFE源代码

```
$ git clone https://github.com/bfenetworks/bfe
```

BFE源代码编译

进入目录*bfe*，执行*make*命令，编译BFE源代码:

```
$ cd bfe
$ make
```

> tips: 如果遇到超时错误”https fetch: Get ... connect: connection timed out”, 请设置GO_PROXY代理后重试.

运行BFE

编译后完成后，可执行目标文件在目录*output/bin/*下:

```
$ file output/bin/bfe
output/bin/bfe: ELF 64-bit LSB executable, ...
```

执行如下命令，运行BFE服务:

```
$ cd output/bin/
$ ./bfe -c ../conf -l ../log
```

docker方式安装运行

BFE也提供了基于docker的容器镜像，可以方便的通过docker进行安装部署。
BFE的镜像可以在docker hub中找到 *https://hub.docker.com/r/bfenetworks/bfe*

docker环境设置

参考 *docker.com*，设置好系统的docker环境。

运行BFE容器

执行以下命令可以启动一个BFE容器

```
# docker run -d -p 8080:8080 -p 8443:8443 -p 8421:8421 bfenetworks/bfe
```

```
sh-3.2$ docker run -d -p 8080:8080 -p 8443:8443 -p 8421:8421 bfenetworks/bfe
Unable to find image 'bfenetworks/bfe:latest' locally
latest: Pulling from bfenetworks/bfe
21c83c524219: Pull complete
dd797bab7ecd: Pull complete
fe31021e23c9: Pull complete
a3df6933411b: Pull complete
Digest: sha256:d1887721db70fa2aff5088dc8068c60f762e334aca7d5d335bcd36fb6434baec
Status: Downloaded newer image for bfenetworks/bfe:latest
7e15304bb972820c2f2703221a8fb4c9e019ac4ed39cedf9c405a2ae3f5da078
```

上述命令将运行一个BFE容器，同时把BFE容器中的三个缺省端口映射到本地。

查看容器的运行状态，可以看到容器id为7e15304bb972

```
sh-3.2$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                    NAMES
7e15304bb972        bfenetworks/bfe     "./bfe -c ../conf/ -…"   3 minutes ago       Up 3 minutes        0.0.0.0:8080->8080/tc
p, 0.0.0.0:8421->8421/tcp, 0.0.0.0:8443->8443/tcp   funny_swirles
```

如果要停止容器的运行，可以执行以下命令：

```
sh-3.2$ docker stop 7e15304bb972
7e15304bb972
sh-3.2$
sh-3.2$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS
      NAMES
7e15304bb972        bfenetworks/bfe     "./bfe -c ../conf/ -…"   6 minutes ago       Exited (0) 5 seconds ago
      funny_swirles
```

BFE命令行参数

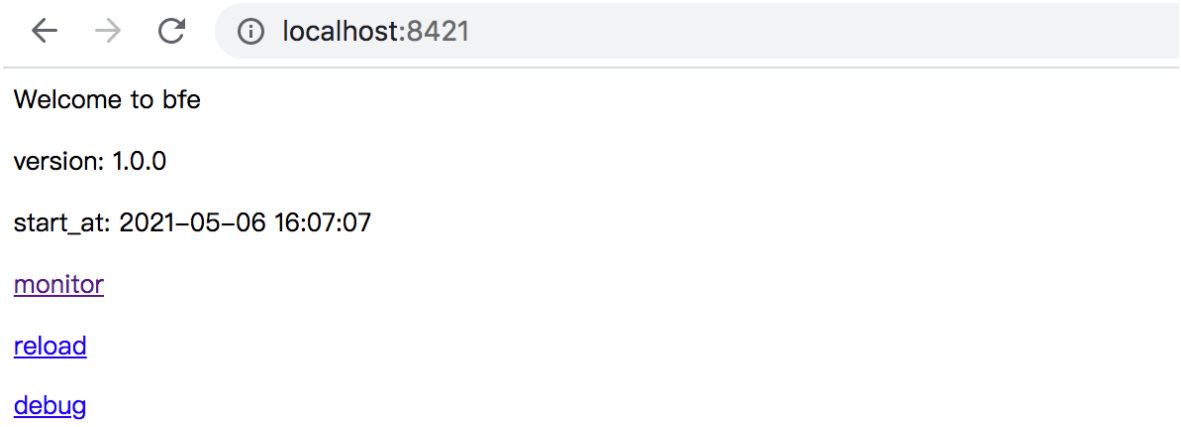
在上面的章节中，我们使用了最常用的命令行参数 *-c* 和 *-l* 。BFE支持的命令行参数如下。

选项	说明
-c	配置文件的根目录，默认路径./conf
-l	日志文件的根目录，默认路径./log
-s	打印log到stdout
-d	打印debug日志
-v	显示bfe的版本号

选项	说明
-V	显示bfe的版相关的详细信息
-h	显示帮助

查看BFE服务的运行状态

BFE提供了接口，可以通过该接口查看服务运行的各种状态数据。端口缺省为8421。用户可以直接使用浏览器访问该端口，如下图：



BFE服务的基础配置

上一章对如何下载运行BFE进行了介绍，用安装包中的缺省配置运行了BFE程序。

本章将对BFE的配置进行进一步介绍，通过一个简单的负载均衡的例子，说明如何基于BFE配置一个基础的负载均衡系统。

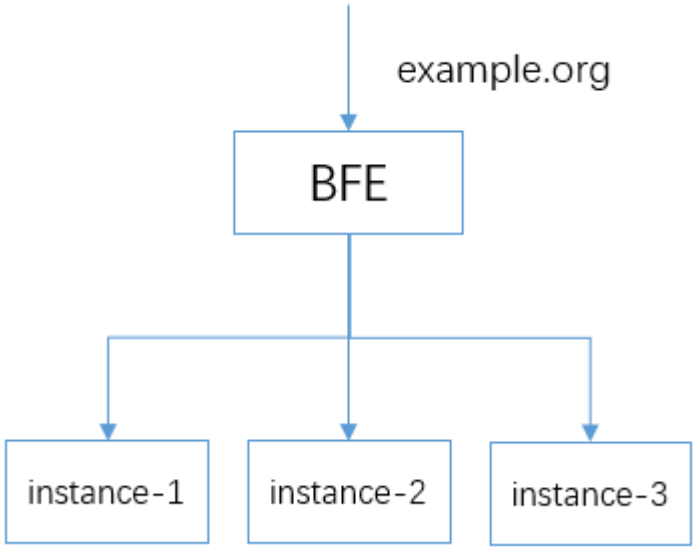
场景说明

对于负载均衡，基本能力就是将从客户端的请求，转发到一组后端服务示例上去。BFE的概念中，相同功能的后端服务定义为一个集群。

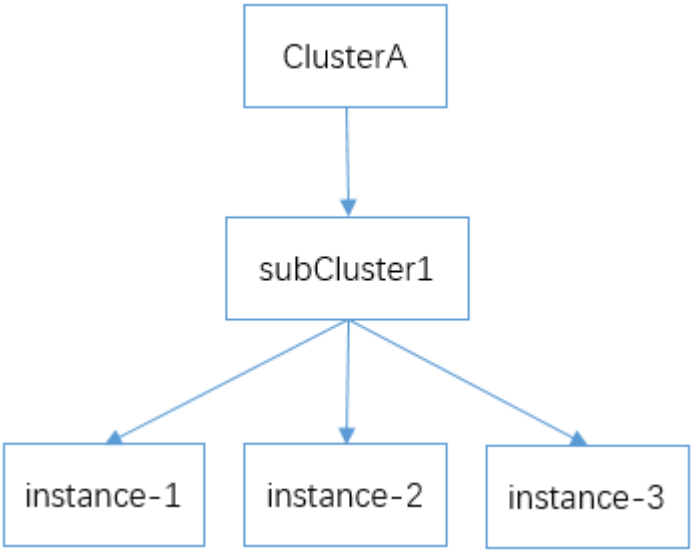
下面我们通过一个简单的例子，展示如何配置BFE转发到后端集群上。

现在有一个域名为“example.org”网站，后端有三台服务器，instance-1,instance-2,instance-3。需要通过BFE将HTTP请求转发到后端三台服务器上。

如下示意图：



在后续配置中，三个实例将定义为在同一个逻辑集群clusterA和逻辑子集群subCluster1中：



修改基础配置文件*bfe.conf*

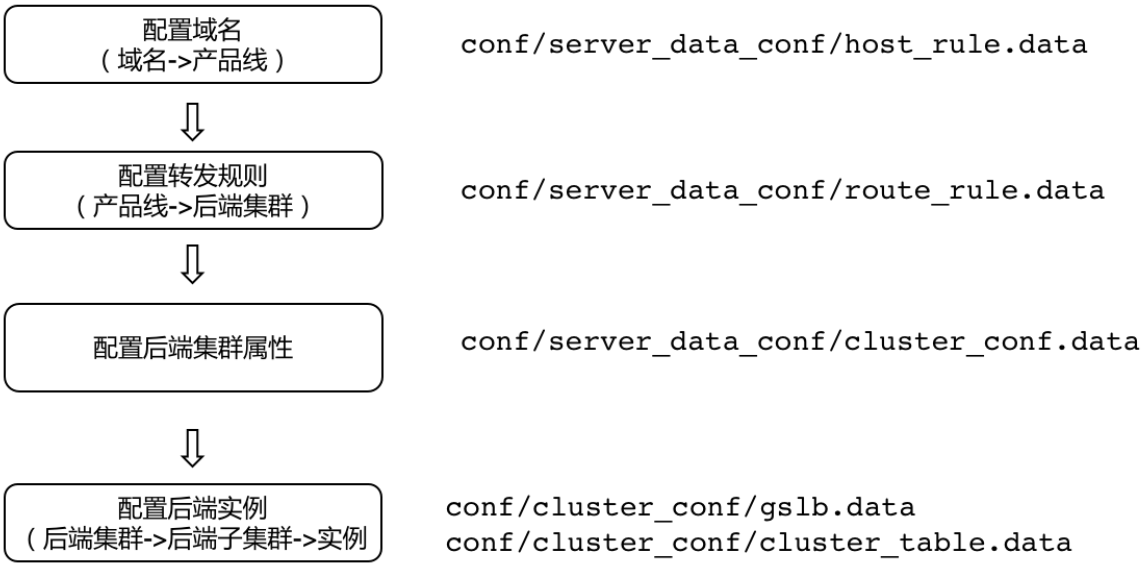
*bfe.conf*包含了BFE的基础配置。读者了解配置文件，可以从这个文件入手。该配置文件包含大量配置选项，具体含义将在后续进行描述。

在bfe.conf中可以查看缺省端口配置，用户可以根据需要，修改监听的端口。。

```
[Server]
# listen port for http request
HttpPort = 8080
# listen port for https request
HttpsPort = 8443
# listen port for monitor request
MonitorPort = 8421
```

转发配置流程

确定基础配置文件bfe.conf后，用户可以设置转发相关的配置。



转发配置的推荐流程如上图所示，主要步骤为：

1. 配置域名：配置需要处理的域名，这也用于区分租户/产品线。
2. 配置转发规则：定义消息到后端集群的转发/映射规则，即按什么规则转发消息到某个后端集群。
3. 配置后端集群的属性：设置后端集群的一些参数，比如后端建连的属性，健康检查方式等。
4. 配置后端集群实例：配置后端集群实例和权重信息，包括子集群的权重，子集群中的实例的权重等等。

案例说明

下面将以添加一个租户example_product为例，详细描述如何为该租户创建相关配置。

产品线域名配置 host_rule.data

conf/server_data_conf/host_rule.data是BFE的产品线域名表配置文件。

host_rule.data中的"HostTags"字段定义了租户信息，我们添加一个名为"example_product"的租户。租户下可以定义多个域名的tag。字段"Hosts"定义了该tag包含的域名。例子中的tag：exampleTag，包含了需要支持的域名 example.org。文件中的tag由用户自行指定。

该配置文件的示例如下：

```
{
    "Version": "1",
    "DefaultProduct": null,
    "Hosts": {
        "exampleTag":[
            "example.org"
        ]
    },
    "HostTags": {
        "example_product":[
            "exampleTag"
        ]
    }
}
```

转发规则配置 route_rule.data

conf/server_data_conf/route_rule.data 是BFE的分流配置文件。

在上述例子中，将定义一个后端集群cluster_A，它将代表后端服务的集群。

以下示例中，“Cond”使用了条件原语req_host_in()，将请求header的Host域为"example.org"的消息，发送到后端集群cluster_A。

```
{
    "Version": "1",
    "ProductRule": {
        "example_product": [
            {
                "Cond": "req_host_in(\"example.org\")",
                "ClusterName": "cluster_A"
            },
            {
                "Cond": "default_t()",
                "ClusterName": "cluster_default"
            }
        ]
    }
}
```

后端集群配置 cluster_conf.data

conf/server_data_conf/cluster_conf.data 为后端集群的配置文件。包含了后端集群“cluster_A”的相关配置。包括：后端基础配置、健康检查配置、GSLB基础配置和集群基础配置。

上述的例子，可以使用以下配置：

```
{
    "Version": "1",
    "Config": {
        "cluster_A": {
```

```
            "BackendConf": {
                "TimeoutConnSrv": 2000,
                "TimeoutResponseHeader": 50000,
                "MaxIdleConnsPerHost": 0,
                "RetryLevel": 0
            },
            "CheckConf": {
                "Schem": "http",
                "Uri": "/",
                "Host": "example.org",
                "StatusCode": 200,
                "FailNum": 10,
                "CheckInterval": 1000
            },
            "GslbBasic": {
                "CrossRetry": 0,
                "RetryMax": 2
            },
            "ClusterBasic": {
                "TimeoutReadClient": 30000,
                "TimeoutWriteClient": 60000,
                "TimeoutReadClientAgain": 30000,
            }
        }
    }
}
```

cluster_conf.data中的配置项较多，各个项的具体描述如下：

- 后端基础配置BackendConf

配置项	描述
BackendConf.TimeoutConnSrv	Integer 连接后端的超时时间，单位是毫秒 默认值2
BackendConf.TimeoutResponseHeader	Integer 从后端读响应头的超时时间，单位是毫秒 默认值60
BackendConf.MaxIdleConnsPerHost	Integer BFE实例与每个后端的最大空闲长连接数 默认值2
BackendConf.MaxConnsPerHost	Integer BFE实例与每个后端的最大长连接数， 0代表无限制 默认值0

配置项	描述
BackendConf.RetryLevel	Integer 请求重试级别。0：连接后端失败时， 进行重试；1：连接后端失败、 转发GET请求失败时均进行重试 默认值0

* 健康检查配置CheckConf

配置项	描述
CheckConf.Schem	String 健康检查协议，支持HTTP和TCP 默认值 HTTP
CheckConf.Uri	String 健康检查请求URI (仅HTTP) 默认值 /health_check
CheckConf.Host	String 健康检查请求HOST (仅HTTP) 默认值 “”
CheckConf.StatusCode	Integer 期待返回的响应状态码 (仅HTTP) 默认值 0，代表任意状态码
CheckConf.FailNum	Integer 健康检查启动阈值（转发请求连续失败FailNum次后， 将后端实例置为不可用状态，并启动健康检查） 默认值5
CheckConf.SuccNum	Integer 健康检查成功阈值（健康检查连续成功SuccNum次后， 将后端实例置为可用状态） 默认值1
CheckConf.CheckTimeout	Integer 健康检查的超时时间，单位是毫秒 默认值0（无超时）

配置项	描述
CheckConf.CheckInterval	Integer 健康检查的间隔时间，单位是毫秒 默认值1

- GSLB基础配置GslbBasic

配置项	描述
GslbBasic.CrossRetry	Integer 跨子集群最大重试次数 默认值0
GslbBasic.RetryMax	Integer 子集群内最大重试次数 默认值2

配置后端集群实例

该部分配置包括：

- 子集群负载均衡配置 conf/cluster_conf/gslb.data
- 实例负载均衡配置 conf/cluster_conf/cluster_table.data

####

glsb.data
该文件描述了一个集群所包含的子集群的权重。在上述例子中，我们只需要为集群“cluster_A”定义一个子集群”subCluster1”，权重为100。

```
{
    "Clusters": {
        "cluster_A": {
            "GSLB_BLACKHOLE": 0,
            "subCluster1": 100
        }
    },
    "Hostname": "",
    "Ts": "0"
}
```

cluster_table.data

该文件描述了子集群中的实例的信息。示例的子集群”subCluster1”将包含后端的三个实例instance-1、instance-2、instance-3。同时指定了每个实例的IP地址。

```
{
    "Config": {
        "cluster_A": {
            "subCluster1": [
                {
                    "Addr": "192.168.2.1",
                    "Name": "instance-1",
                    "Port": 8080,
                    "Weight": 10
                },
                {
                    "Addr": "192.168.2.2",
                    "Name": "instance-2",
                    "Port": 8080,
                    "Weight": 10
                },
                {
                    "Addr": "192.168.2.3",
                    "Name": "instance-3",
                    "Port": 8080,
                    "Weight": 10
                }
            ]
        }
    },
    "Version": "1"
}
```

完成上述配置后，重新启动bfe实例，让配置生效。

服务访问验证

在客户端使用curl命令，访问bfe的地址和端口，可以看到看到请求被转发到后端服务器实例。

> 注意：curl命令需要通过参数 -H “Host: example.org”，指定请求的header中的Host字段，否则会收到500错误。

配置的重新加载

上述配置修改后，也可以通过动态方式重新加载配置。

BFE的配置可大致分为两部分：常规配置和动态配置。常规配置，比如bfe.conf，重新加载需要重启BFE进程，新配置才能生效。

动态配置，可以通过访问监控端口，缺省为8421，来触发配置文件的重新读取，无需重启BFE进程，对服务无影响。具体可参考配置管理。

配置负载均衡算法及会话保持

配置负载均衡系统时，其中重要的参数就是如何设置负载均衡的算法。这决定了负载均衡系统如何将消息转发到后端的实例。

对BFE来说，一个后端集群可能包含多个子集群，子集群又可能包含多个实例。所以，负载均衡的算法在两层生效：子集群级别和实例级别。

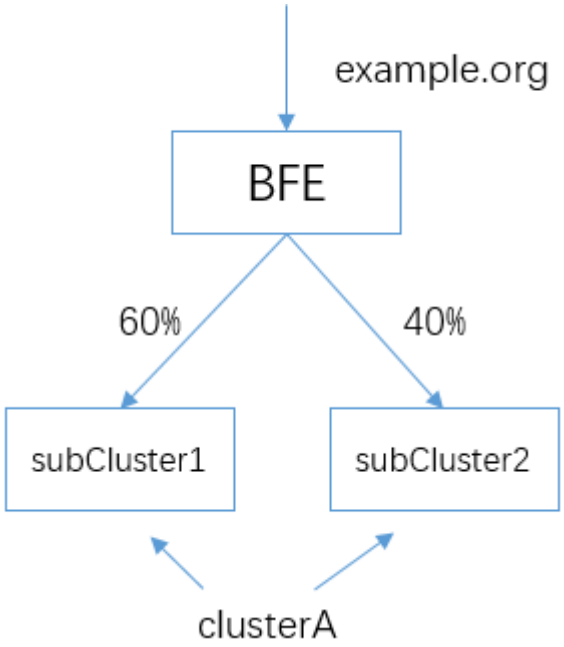
子集群间的负载均衡

对于后端服务的一个集群，我们可以设置其所包含的子集群的权重。BFE会按子集群所设置权重，将请求的消息分配到子集群。

子集群间的负载均衡使用hash算法，再根据子集群的权重进行消息的按比例分配。缺省使用源IP进行hash计算。

配置示例

基于前面一章的例子，我们为后端服务的集群A进行扩容，增加一个子集群subCluster2，现在clusterA包含两个子集群subCluster1和subCluster2。由于两个子集群的处理能力不同，分别为其设置不同的流量比例。



子集群的流量比例，可以在conf/cluster_conf/gslb.data中指定。

下面为该配置文件的内容。我们可以分别为两个子集群指定权重值，以支持上面的流量分配：

```
{
    "Clusters": {
        "clusterA": {
            "GSLB_BLACKHOLE": 0,
            "subCluster1": 60,
            "subCluster2": 40,
        }
    }
}
```

子集群会话保持

对于子集群间的流量分配，BFE是根据请求消息中的字段（如IP地址、消息头等）进行hash计算，再按照配置的权重选择转发的后端。hash计算能保证携带相同信息的消息能被分配到相同的子集群，也就实现了子集群级别的会话保持。

当然，由于实现的方式是基于hash计算。所以，当子集群的数量有变化时，会对会话保持会受到一定影响，这个需要注意。

会话保持可以基于请求的源IP地址，或者请求的header中的特定域。具体使用哪种方式，可以通过配置文件来设置。

配置示例

在*conf/server_data_conf/cluster_conf.data*中，修改配置文件中的"HashConf"字段，设置子集群会话保持的属性。

如下实例中，BFE将会使用请求中名为UID的cookie进行hash计算。

```
Cookie: UID=12345
```

```
{
    "config": {
        ...
        "cluster_example": {
            ...
            "GslbBasic": {
                ...
                "HashConf": {
                    "HashStrategy": 0,
                    "HashHeader": "Cookie:UID",
                    "SessionSticky":false
                }
            }
    }
}
```

参数具体含义

"HashConf"中字段的含义：

- HashStrategy： 设置会话保持中使用的策略：
    - 0：使用请求header中的域做会话保持，域的名字由"HashHeader"指定。
    - 1: 使用请求的源IP地址做会话保持
    - 2：优先使用header，如该header不存在，使用源IP。
- HashHeader: 当使用header中的域进行会话保持时，该参数指定了header中域的名字。 比如在上面示例中，”Cookie:UID”指定使用名为UID的cookie做会话保持。
- SessionSticky: 是否开启实例级别的会话保持。

实例间的负载均衡

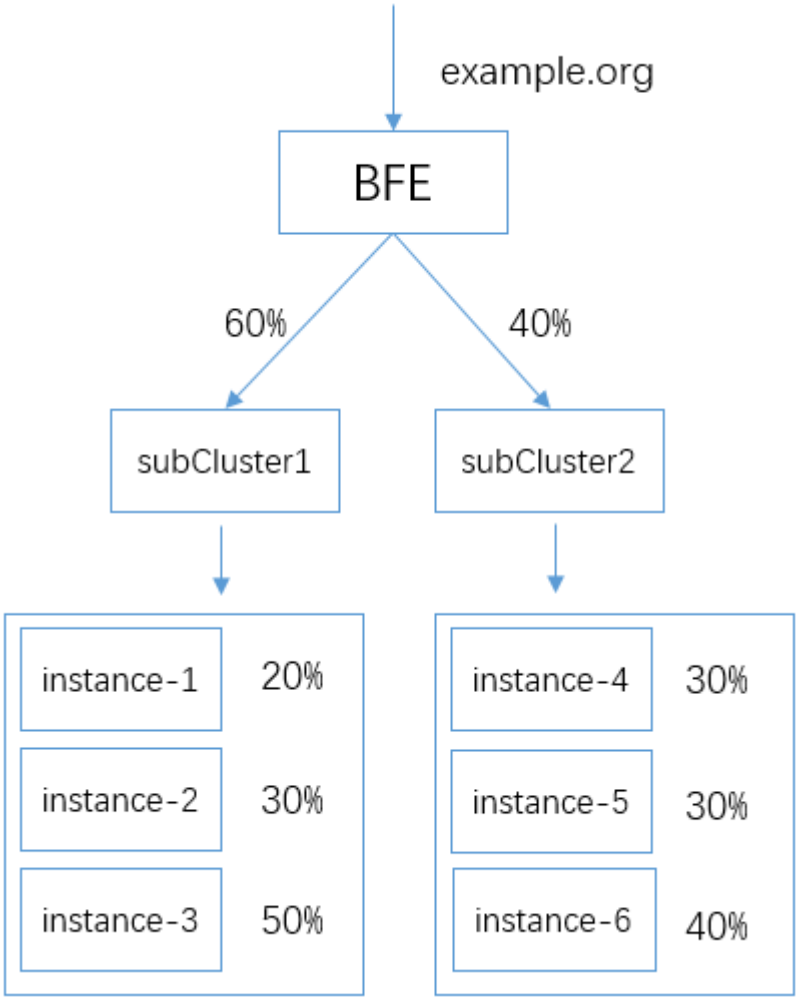
一个子集群内部，我们可以定义多个实例。如何将这个子集群上的流量，分配到这些实例当中，这就涉及到实例间的负载均衡问题。

子集群的实例间的负载均衡算法支持：

- 平滑加权轮询
- 最少连接数

加权轮询的配置示例

加权轮询时是实例间负载均衡的缺省配置。使用这种方式，用户只需要设置实例的权重。
基于前面的子集群负载均衡的例子，我们为两个子集群中的实例设置不同的权重。支持如下的流量转发：



如前面所提到，实例的权重配置文件为 *conf/cluster_conf/cluster_table.data*

上述场景的配置如下：

```
{
    "Config": {
        "cluster_A": {
            "subCluster1": [
                {
                    "Addr": "192.168.2.1",
                    "Name": "instance-1",
                    "Port": 8080,
                    "Weight": 2
                },
                {
                    "Addr": "192.168.2.2",
                    "Name": "instance-2",
                    "Port": 8080,
```

```
                "Weight": 3
            },
            {
                "Addr": "192.168.2.3",
                "Name": "instance-3",
                "Port": 8080,
                "Weight": 5
            }
        ],
        "subCluster2": [
            {
                "Addr": "192.168.3.1",
                "Name": "instance-4",
                "Port": 8080,
                "Weight": 3
            },
            {
                "Addr": "192.168.3.2",
                "Name": "instance-5",
                "Port": 8080,
                "Weight": 3
            },
            {
                "Addr": "192.168.3.3",
                "Name": "instance-5",
                "Port": 8080,
                "Weight": 4
            }
        ]
    }
  },
  "Version": "1"
}
```

最小连接数配置的示例

如果需要在子集群的实例间使用最小连接数的负载均衡算法，修改*conf/server_data_conf/cluster_conf.data*中的"BalanceMode"字段，如下：

```
{
  "config": {
    ...
    "cluster_example": {
      ...
      "GslbBasic": {
        ...
        "BalanceMode": "WLC",
        ...
      }
  }
}
```

修改上述配置项，后端请求会被转发到子集群中连接数最少的实例。

实例级别的会话保持示例

我们可以设置会话保持到后端实例。
修改配置，将*conf/server_data_conf/cluster_conf.data*中的"SessionSticky"变为true：

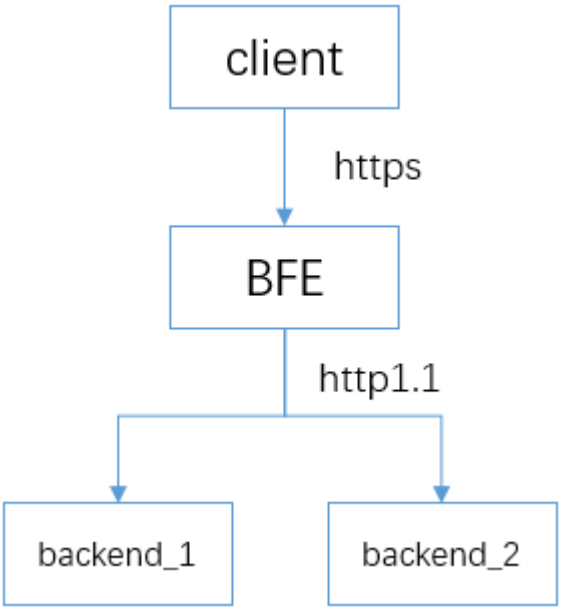
```
        "GslbBasic": {
            ...
            "HashConf": {
                "HashStrategy": 0,
                "HashHeader": "Cookie:UID",
                "SessionSticky":true
            }
```

当开启该功能，BFE会使用hash方式，计算得到子集群中的后端实例。这样，相同UID的消息，总能hash得到相同的后端，从而实现了会话保持功能。

这里也会出现同样的问题：如果后端实例列表发生变化，会话会转移到其他实例上。

配置HTTPS服务

BFE可以支持TLS卸载（TLS offloading），将收到的加密的HTTPS流量进行解密后，再转发给后端服务。



HTTPS基础配置

HTTPS配置包括加密套件、证书文件的配置、TLS协议相关参数的配置。使用HTTPS需要配置相应的证书文件，请提前准备好证书文件。

配置HTTPS端口

在bfe.conf中可以配置HTTPS的端口

```
[Server]
...

# listen port for https request
HttpsPort = 8443
```

配置加密套件

bfe.conf中包含了如下的加密套件，用户可以根据需要进行修改。这部分定义了TLS握手中支持的加密套件。

```
CipherSuites=TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256|TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
CipherSuites=TLS_ECDHE_RSA_WITH_RC4_128_SHA
CipherSuites=TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
CipherSuites=TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
CipherSuites=TLS_RSA_WITH_RC4_128_SHA
CipherSuites=TLS_RSA_WITH_AES_128_CBC_SHA
CipherSuites=TLS_RSA_WITH_AES_256_CBC_SHA
```

配置服务端证书

bfe.conf中也包含了服务端证书的配置文件的的路径，可以直接使用缺省配置。

```
ServerCertConf = tls_conf/server_cert_conf.data
```

打开服务端证书配置文件tls_conf/server_cert_conf.data，可以看到如下证书信息：

```
{
    "Version": "1",
    "Config": {
        "Default": "example.org.cert",
        "CertConf": {
            "example.org.cert": {
                "ServerCertFile": "tls_conf/certs/server.crt",
                "ServerKeyFile" : "tls_conf/certs/server.key"
            }
        }
    }
}
```

该文件包含了服务端证书的信息，其中"Default"字段指示了服务端的缺省证书。"CertConf"中包含了证书文件的具体内容。上面的例子中，配置了一个名为"example.org.cert"的证书。
"CertConf"中可以包含多个证书，分别用不同的名字进行标识。每个证书的定义都包含"ServerCertFile"和"ServerKeyFile"两个字段，分别指示服务端证书文件和对应的私钥文件。

证书名字，比如上例中的"example.org.cert"，将在后续的tls_rule_conf.data配置中被使用。

配置TLS规则

准备好证书之后，需要设置TLS相关的规则，比如如何选择证书。

bfe.conf包含了TLS规则文件的路径：

```
TlsRuleConf = tls_conf/tls_rule_conf.data
```

下面示例是我们为租户"example_product"配置的规则。

```
{
    "Version": "1",
    "DefaultNextProtos": ["http/1.1"],
    "Config": {
        "example_product": {
            "SniConf": "example.org",
            "CertName": "example.org.cert",
            "NextProtos": [
                "http/1.1"
            ],
            "Grade": "C"
        }
    }
}
```

其中"SniConf"指示了使用SNI的域名。"NextProtos"标识了ALPN协商中使用的协议。安全等级"Grade"定义了TLS协商中服务端可以使用的加密套件的等级，具体描述见后续章节。

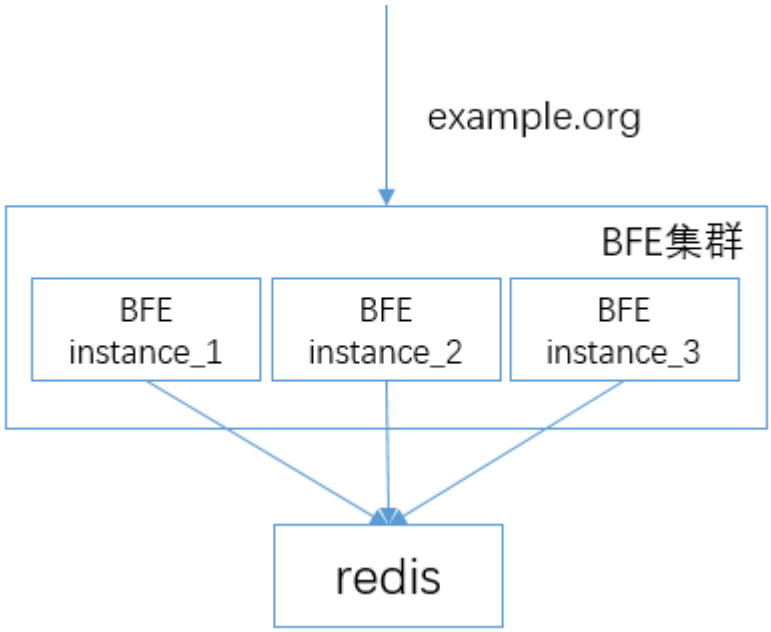
上述配置后，重启BFE让配置生效。就可以通过HTTPS访问配置的HTTPS端口。

配置TLS会话重用

BFE支持使用两种TLS会话重用的方式：Session Cache和Session Ticket。

配置Session Cache

BFE使用Redis进行集中的TLS Session信息存储。多个BFE实例可以连接共享的Redis服务。连接如下图：



为开启session cache，在bfe.conf中设置SessionCacheDisabled为false。

```
[SessionCache]
# disable tls session cache or not
SessionCacheDisabled = false

# tcp address of redis server
Servers = "example.redis.cluster"

# prefix for cache key
KeyPrefix = "bfe"

# connection params (ms)
ConnectTimeout = 50
ReadTimeout = 50
WriteTimeout = 50

# max idle connections in connection pool
MaxIdle = 20

# expire time for tls session state (second)
SessionExpire = 3600
```

上述配置中的"Servers"字段，指明了连接的redis服务器名字。该名字的具体地址，在conf/server_data_conf/name_conf.data中指定：

```
{
    "Version": "1",
```

```
    "Config": {
        "example.redis.cluster": [
            {
                "Host": "192.168.3.1",
                "Port": 6379,
                "Weight": 10
            }
        ]
    }
}
```

配置Session Ticket

如需支持TLS session ticket，在bfe.conf文件中，设置SessionTicketsDisabled 为 false：

```
[SessionTicket]
# disable tls session ticket or not
SessionTicketsDisabled = false
# session ticket key
SessionTicketKeyFile = tls_conf/session_ticket_key.data
```

在SessionTicketKeyFile指向的文件中，用户可以配置加密ticket的密钥，格式为包含字符a-z/0-9，长度为48的字符串。

开启上述配置后，如client支持session ticket，TLS握手中就可实现基于session ticket的会话重用。

配置双向认证

在一些场景，我们需要配置双向TLS，对客户端进行认证。BFE上可以支持配置客户端证书。

在bfe.conf中可以配置clientCA证书的目录：

```

# client ca certificates base directory
# Note: filename suffix for ca certificate file should be ".crt", eg. example_ca_bundle.crt
ClientCABaseDir = tls_conf/client_ca

# client certificate crl base directory
# Note: filename suffix for crl file should be ".crl", eg. example_ca_bundle.crl
ClientCRLBaseDir = tls_conf/client_crl
```

在conf/tls_conf/client_ca中放置客户端的CA证书，证书名需以.crt结尾，如下：

```
# ls conf/tls_conf/client_ca
example_ca.crt
```

修改conf/tls_conf/tls_rule_conf.data中租户”example_product”的配置，将
“ClientAuth”设置为true，”ClientCAName”为上述证书文件的名字，本例中为”example_ca”。

```
{
    ...
    "Config": {
        "example_product": {
            ...
            "ClientAuth": true,
            "ClientCAName": "example_ca"
```

```
        }
    }
}
```

注意，当前实现中，BFE验证客户端证书必须在”Extended Key Usage”中开启”clientAuth”。可以查看证书信息进行确认。如下，查看client.crt的具体信息：

```
# openssl x509 -in client.crt -text -noout

Certificate:
    ...
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Key Usage:
                Digital Signature, Non Repudiation, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication, Code Signing, E-mail Protection
    ...
```

配置完成后，客户端就可使用上述证书client.crt和相应的私钥访问HTTPS的服务端口了。

安全等级说明

TLS协议配置文件中包含配置项“Grade”，该值指定了使用的加密套件的安全级别。

BFE支持多种安全等级（A+/A/B/C）。各安全等级差异在于支持的协议版本及加密套件。A+等级安全性最高、连通性最低；C等级安全性最低、连通性最高。

- 安全等级A+

支持协议	支持加密套件
TLS1.2	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_OLD_SHA256 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_256_CBC_SHA

- 安全等级A

支持协议	支持加密套件

支持协议	支持加密套件
TLS1.2 TLS1.1 TLS1.0	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_OLD_SHA256 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_256_CBC_SHA

- 安全等级B

支持协议	支持加密套件
TLS1.2 TLS1.1 TLS1.0	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_OLD_SHA256 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_256_CBC_SHA
SSLv3	TLS_ECDHE_RSA_WITH_RC4_128_SHA TLS_RSA_WITH_RC4_128_SHA

- 安全等级C

支持协议	支持加密套件
TLS1.2 TLS1.1 TLS1.0	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_OLD_SHA256 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_256_CBC_SHA TLS_ECDHE_RSA_WITH_RC4_128_SHA TLS_RSA_WITH_RC4_128_SHA
SSLv3	TLS_ECDHE_RSA_WITH_RC4_128_SHA TLS_RSA_WITH_RC4_128_SHA

配置rewrite

本章介绍如何配置HTTP rewrite。该功能对收到的HTTP请求消息进行修改，再转发到后端服务。

开启rewrite

在conf/bfe.conf中，打开该模块

```
Modules = mod_write
```

模块配置

模块配置在目录conf/mod_rewrite/中，包含两个文件:

```
$ ls
mod_rewrite.conf    rewrite.data
```

mod_rewrite.conf为模块基础配置文件，指向rewrite规则文件，通常无需修改。

```
$ cat mod_rewrite.conf
[basic]
DataPath = mod_rewrite/rewrite.data
```

rewrite.data 包含rewrite规则，可动态加载。安装包中的示例配置文件如下：

```
$ cat rewrite.data
{
    "Version": "1",
    "Config": {
        "example_product": [
            {
                "Cond": "req_path_prefix_in(\"/rewrite\", false)",
                "Actions": [
                    {
                        "Cmd": "PATH_PREFIX_ADD",
                        "Params": [
                            "/bfe/"
                        ]
                    }
                ],
                "Last": true
            }
        ]
    }
}
```

上述配置为产品线*example_product*中增加了一个规则：对满足条件"Cond"的请求，执行"Actions"动作（包含动作名"Cmd"和对应的参数），如果"Last"为true，停止执行后续动作，否则继续匹配下一条规则。

最终，该规则将修改Path为/rewrite开头的请求，为其增加路径前缀/bfe/，也就是将Path从/rewrite变为/bfe/rewrite。

重写动作

“Actions”中的”Cmd”指示了具体的重写动作。

HOST_SET

设置请求header中的Host值，参数为需设置的值。

示例：

```
{
    "Cmd": "HOST_SET",
    "Params": ["www.example.com"]
}
```

结果：
原始值	http://abc.example.com
修改后	http://www.example.com

HOST_SET_FROM_PATH_PREFIX

将Path中的第一段设置为Host的值，并在Path中去掉该段。例如：如果Path为/x.example.com/xxxx，设置Host为x.example.com，PATH为/xxxx。

示例：

```
{
    "cmd": "HOST_SET_FROM_PATH_PREFIX",
    "params": []
}
```

结果：
原始值	http://www.example.com/test.example.com/xxxx
修改后	http://test.example.com/xxxx

HOST_SUFFIX_REPLACE

替换域名中的特定后缀。两个参数分别为被替换的后缀字符串和替换后的字符串。

示例：

```
{
    "cmd": "HOST_SUFFIX_REPLACE",
    "params": ["net", "com"]
}
```

结果：

|原始值 | http://www.example.net |
|修改后 | http://www.example.com |

PATH_SET

设置Path为指定值，参数为的新Path值。

示例：

```
{
    "cmd": "PATH_SET",
    "params": ["/index"]
}
```

结果：
原始值	http://www.example.com/current
修改后	http://www.example.com/index

PATH_PREFIX_ADD

为Path增加前缀，参数为需增加的前缀。

示例：

```
{
    "cmd": "PATH_PREFIX_ADD",
    "params": ["/index"]
}
```

结果：
原始值	http://www.example.com/current
修改后	http://www.example.com/index/current

PATH_PREFIX_TRIM

删除Path前缀，参数为需删除的前缀。

示例：

```
{
    "cmd": "PATH_PREFIX_TRIM",
    "params": ["/service"]
}
```

结果：
| | |

|-|-|
|原始值 | http://www.example.com/service/index.html |
|修改后 | http://www.example.com/index.html |

QUERY_ADD

增加Query，参数指定需增加的query的key和value。

示例：

```
{
    "cmd": "QUERY_ADD",
    "params": ["name", "alice"]
}
```

结果：
原始值	http://www.example.com/
修改后	http://www.example.com/?name=alice

QUERY_RENAME

对Query重命名，参数指定key的原名字和新名字。

示例：

```
{
    "cmd": "QUERY_ADD",
    "params": ["name", "user"]
}
```

结果：
原始值	http://www.example.com/?name=alice
修改后	http://www.example.com/?user=alice

QUERY_DEL

删除指定Query，参数指定key的名字。

示例：

```
{
    "cmd": "QUERY_ADD",
    "params": ["name"]
}
```

结果：
原始值	http://www.example.com/?name=alice
修改后	http://www.example.com/

QUERY_DEL_ALL_EXCEPT

删除Query中指定key外所有其他key。

示例：

```
{
    "cmd": "QUERY_DEL_ALL_EXCEPT",
    "params": ["name"]
}
```

结果：
原始值	http://www.example.com/?name=alice?key1=value2&key2=value2
修改后	http://www.example.com/?name=alice

配置redirect

本章介绍如何配置HTTP重定向（redirect）。该功能将对收到的请求返回重定向响应码，指示客户端跳转到新的URL。

开启redirect

在conf/bfe.conf中，打开该模块

```
Modules = mod_redirect
```

模块配置

模块配置在目录conf/mod_redirect/中，包含两个文件：

```
$ ls
mod_redirect.conf    redirect.data
```

mod_redirect.conf为模块基础配置文件，指向重定向规则文件，通常无需修改。

```
$ cat mod_redirect.conf
[basic]
DataPath = mod_redirect/redirect.data
```

rewrite.data 包含重定向规则，可动态加载。安装包中的示例中的配置文件如下：

```
{
    "Version": "1",
    "Config": {
        "example_product": [
            {
                "Cond": "req_path_prefix_in(\"/redirect\", false)",
                "Actions": [
                    {
                        "Cmd": "URL_SET",
                        "Params": ["https://example.org"]
                    }
                ],
                "Status": 301
            }
        ]
    }
}
```

上述配置为产品线*example_product*中增加了一个规则：对满足条件"Cond"（请求路径的前缀为"/redirect"）的请求，执行"Actions"动作（重定向到https://example.org），返回HTTP响应码为301。

重定向动作

"Actions"中的"Cmd"指示了如何设置重定向中的URL。

URL_SET

重定向请求到指定URL，参数为重定向的URL。

示例：

```
{
    "Cmd": "URL_SET",
    "Params": ["http://www.example.com/more"]
}
```

结果：
请求	http://www.example.com/unknown
重定向到	http://www.example.com/more

URL_FROM_QUERY

将query中的某个key的值设置为重定向地址。参数为query中该key的名字。

示例：

```
{
    "Cmd": "URL_FROM_QUERY",
    "Params": ["url"]
}
```

结果：
请求	http://www.example.com/redirect?url=http://news.example.com
重定向到	http://news.example.com

URL_PREFIX_ADD

设置重定向URL为当前URL加上特定前缀。参数为需要增加的前缀。

示例：

```
{
    "Cmd": "URL_PREFIX_ADD",
    "Params": ["/v1"]
}
```

结果：
请求	http://www.example.com/test.html
重定向到	http://www.example.com/v1/test.html

SCHEME_SET

设置重定向URL的scheme为HTTP或者HTTPS。参数指定为http或者https。

示例：

```
{
    "Cmd": "SCHEME_SET",
    "Params": ["https"]
}
```

结果：
请求	http://www.example.com/index.html
重定向到	https://www.example.com/index.html

配置限流

本章介绍如何配置限流功能。该功能将限制特定请求的速率，以对后端服务进行保护。

开启限流模块

在conf/bfe.conf中，打开该模块

```
Modules = mod_prison
```

模块配置

模块配置在目录conf/mod_redirect/中，包含两个文件：

```
# ls
mod_prison.conf    prison.data
```

与其他模块配置相似，mod_prison.conf为模块基础配置文件，指向redirect规则文件。

```
[basic]
ProductRulePath = mod_prison/prison.data
```

示例的prison.data如下:

```
{
    "Version": "1",
    "Config": {
        "example_product": [{
            "Name": "example_prison",
            "Cond": "req_path_prefix_in(\"/prison\", false)",
            "accessSignConf": {
                "url": false,
                "path": false,
                "query": [],
                "header": [],
                "Cookie": [
                    "UID"
                ]
            },
            "action": {
                "cmd": "CLOSE",
                "params": []
            },
            "checkPeriod": 10,
            "stayPeriod": 10,
            "threshold": 5,
            "accessDictSize": 1000,
            "prisonDictSize": 1000
        }]
    }
}
```

上述示例对路径为“/prison”的请求进行限流。其中，”accessSignConf”指示了限制的流量的维度，具体见后续描述。本例子中，将统计cookies中的”UID”，限制不同”UID”的访问流量。

限制特定维度的流量

通过accessSignConf字段，我们可以指定请求统计的维度，判断统计值是否达到限流阈值。
可配置的维度包括：

- UseClientIP

对请求按clientIp进行统计做限流。可以限制每个clientIP的请求速度。

- UseUrl：

对请求按URL进行统计。可以限制对每一个URL的请求速度。

- UseHost

对请求按Host进行统计。可以限制对每一个Host的请求速度。

- UsePath

对请求按Path进行统计。可以限制对每一种Path的请求速度。

- UrlRegexp

对请求的URL做正则匹配，以匹配结果为维度进行统计。可以实现按URL中的部分内容来进行限流。

- header

以请求的特定Header为维度来做统计流量。可以限制该Header所标识的每一种消息的请求速度。

- Cookie

以请求中的特定Cookie为维度进行统计。可以限制该Cookie标识的每一种请求的请求速度。比如，如果我们用Cookie中的UID来标识不同用户，可以通过指定UID，来限制每个用户的访问速度。

- query

以指定的query为维度，统计请求量来做限流。

- UseHeaders

以每个请求中的所有Header为维度来进行统计(合并一个消息的所有header)。限制不同Header的每一种消息的请求速度。

设置限流门限

通过下述参数配置限流：

- checkPeriod

设置统计周期，单位秒。

- stayPeriod

被限流后的惩罚时长。在该时间段内，该维度访问请求都将被限制。

- threshold

限制的阈值。一个维度的统计数量达到该阈值将触发限流。

- accessDictSize

访问统计表的大小。统计表保存了当前配置的维度的所有统计值。比如，如果以ClientIp为维度进行的统计，该表包含每个ClientIp的访问量。

- prisonDictSize

访问封禁表的大小。按维度统计后，每类命中限流的请求，都会被保存在封禁表中。所以，封禁表保存了当前所有处于封禁状态的某类请求。比如某频繁访问，被限流的IP地址等。

设置限流动作

某类请求命中限流规则后，可以在”action”中配置对该类请求的后续动作。

- CLOSE

直接关闭请求的连接。

- PASS

正常转发，不做任何处理。

- FINISH

响应后关闭连接。

- REQ_HEADER_SET

正常转发，在请求header中插入指定key和value。key和value在params中指定。

支持更多协议

本章将介绍如何配置BFE以支持更多的协议，如：HTTP/2、SPDY、TLS、WebSocket、FastCGI等。

HTTP/2配置

为支持HTTP/2 over TLS，用户需修改TLS规则配置文件 conf/tls_conf/tls_rule_conf.data。

在该文件中，对需要配置的产品线，修改"NextProtos"域，增加"h2"选项，这将在TLS握手的ALPN协商中增加对HTTP2的支持。如果客户端也支持HTTP/2，建立的连接将使用HTTP/2。
如下述实例，我们对产品线"example_product"进行修改，增加了"h2"。

```
"example_product": {
    ...
    "NextProtos": ["h2", "http/1.1"],
    ...
}
```

使用curl连接BFE的HTTPS端口进行测试，可以看到连接已经使用HTTP/2。

```
# curl -v -k -H "Host: example.org"  https://localhost:8443
* Rebuilt URL to: https://127.0.0.1:8443/
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
...
* ALPN, server accepted to use h2
...
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
* Using Stream ID: 1 (easy handle 0x7fabe6000000)
> GET / HTTP/2
> Host: example.org
> User-Agent: curl/7.54.0
> Accept: */*
...
```

上述"NextProtos"配置同时包含了["h2", "http/1.1"]，这表示会同时支持HTTP/2和HTTP/1.1，其中HTTP/2的优先级较高。

如果希望只使用HTTP/2，可以将"NextProtos"设置为"h2;level=2"。level为协商级别，2表示必选。如下示例：

```
"example_product": {
    ...
    "NextProtos": ["h2;level=2"],
    ...
}
```

SPDY配置

与HTTP/2相似，在"NextProtos"中设置"spdy/3.1"，可开启对SPDY的支持。

```
"example_product": {
    ...
    "NextProtos": ["spdy/3.1", "http/1.1"],
    ...
}
```

WebSocket配置

- 支持WS（WebSocket over TCP）

支持ws无需特殊配置，只需配置好支持ws的后端服务和正确的转发路由即可。

使用curl连接BFE的HTTP端口进行测试，带上header："Connection: Upgrade"和"Upgrade: websocket"。我们可以看到连接升级为使用websocket，如下：

```
# curl -v \
       -H "Host: example.org" \
       -H "Connection: Upgrade"  \
       -H "Upgrade: websocket" \
       -H "sec-websocket-key:1234567890"  \
       http://127.0.0.1:8080
* Rebuilt URL to: http://127.0.0.1:8080/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: example.org
> User-Agent: curl/7.54.0
> Accept: */*
> Connection: Upgrade
> Upgrade: websocket
> sec-websocket-key:1234567890
>
< HTTP/1.1 101 Switching Protocols
< Connection: Upgrade
< Sec-Websocket-Accept: qrAsbG+EeIo8ooFLgckbiuFtlYE=
< Upgrade: websocket
```

- 支持WSS（WebSocket over TLS）

为支持WSS，在"NextProtos"中设置如下：

```
"example_product": {
    ...
    "NextProtos": ["stream;level=2"],
    ...
}
```

使用curl连接BFE的HTTPS端口进行测试，可以看到连接也升级为使用WebSocket：

```
# curl -v -k \
       -H "Host: example.org" \
       -H "Connection: Upgrade"  \
       -H "Upgrade: websocket" \
       -H "sec-websocket-key:1234567890"  \
```

```
      https://127.0.0.1:8443

* Rebuilt URL to: https://127.0.0.1:8443/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 8443 (#0)
...
> GET / HTTP/1.1
> Host: example.org
> User-Agent: curl/7.54.0
> Accept: */*
> Connection: Upgrade
> Upgrade: websocket
> sec-websocket-key:1234567890
>
< HTTP/1.1 101 Switching Protocols
< Connection: Upgrade
< Sec-Websocket-Accept: qrAsbG+EeIo8ooFLgckbiuFt1YE=
< Upgrade: websocket
...
```

连接后端服务的协议

BFE支持使用不同的协议连接后端服务器。我们可以在后端集群配置文件conf/server_data_conf/cluster_conf.data中指定该配置。支持的协议包括：

- HTTP
- h2c
- FastCGI

HTTP

HTTP为连接后端使用的缺省协议，无需特殊配置。

h2c

为支持h2c，将"BackendConf"中的"Protocol"设置为"h2c"。

```
"cluster_example": {
    ...
    "BackendConf": {
        "Protocol": "h2c",
        ...
    },
    ...
}
```

FastCGI

为支持FastCGI，将"BackendConf"中的"Protocol"设置为"fcgi"。其他可设置的参数包括请求文件的根路径"Root"和环境变量"EnvVars"。

```
"cluster_example": {
    "BackendConf": {
        "Protocol": "fcgi",
```

```
        ...
        "FCGIConf": {
            "Root": "/home/work",
            "EnvVars": {
                "VarKey": "VarVal"
            }
        }
    }
}
```

实现篇

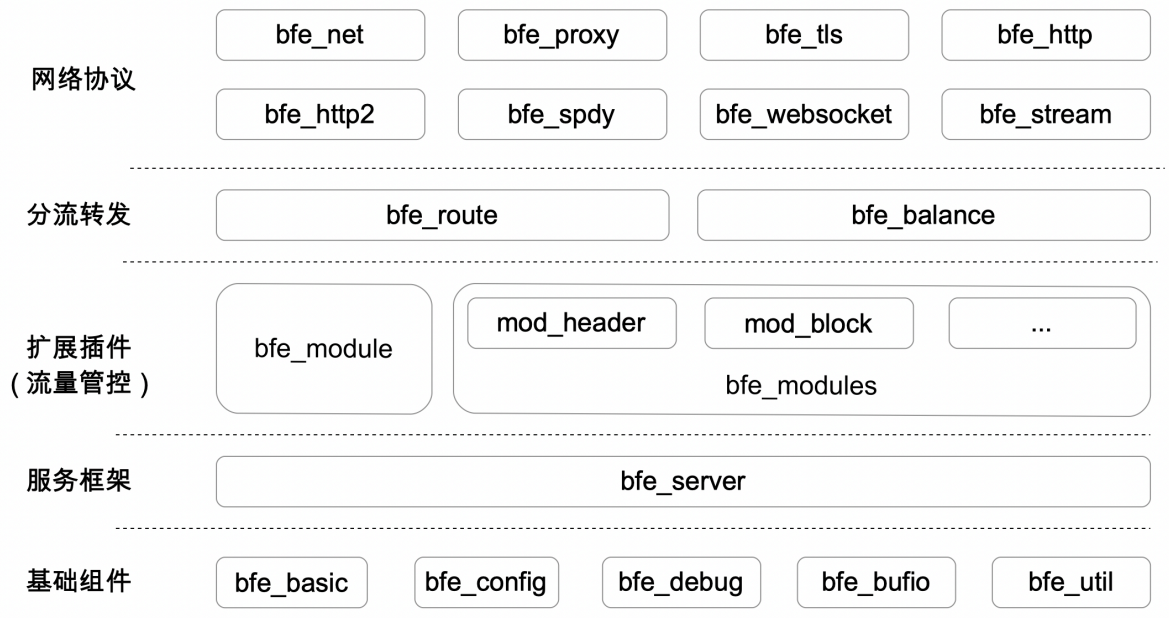
BFE的代码组织

BFE最新的代码可以从BFE开源项目发布页面https://github.com/bfenetworks/bfe/releases/下载。本章将以BFE v1.0.0版本为例介绍BFE。

在代码目录的顶层可以看到BFE包含如下目录或文件：

```
$ ls bfe/
ADOPTERS.md         MAINTAINERS.md        bfe_basic        bfe_modules        bfe_util
CHANGELOG.md        Makefile              bfe_bufio        bfe_net            bfe_websocket
CODE_OF_CONDUCT.md  NOTICE                bfe_config       bfe_proxy          conf
CONTRIBUTING.md     README.md             bfe_debug        bfe_route          docs
CONTRIBUTORS.md     SECURITY.md           bfe_fcgi         bfe_server         go.mod
Dockerfile          VERSION               bfe_http         bfe_spdy           go.sum
GOVERNANCE.md       bfe.go                bfe_http2        bfe_stream         snap
LICENSE             bfe_balance           bfe_module       bfe_tls            staticcheck.conf
```

按逻辑关系各目录的层次结构如下：



按自顶向下顺序各目录对应的功能模块说明如下。

网络协议

- `bfe_net`: BFE网络相关基础库代码
- `bfe_http`: BFE HTTP协议基础代码
- `bfe_tls`: BFE TLS协议基础代码
- `bfe_http2`: BFE HTTP2协议基础代码
- `bfe_spdy`: BFE SPDY协议基础代码
- `bfe_stream`: BFE TLS代理基础代码
- `bfe_websocket`: BFE WebSocket代理基础代码
- `bfe_proxy`: BFE Proxy协议基础代码

分流转发

- `bfe_route`: BFE分流转发相关代码
- `bfe_balance`: BFE负载均衡相关代码

扩展模块

- `bfe_module`: BFE模块框架相关代码
- `bfe_modules`: BFE扩展模块相关代码

服务框架

- `bfe_server`: BFE服务端主体部分

基础组件

- `bfe_basic`: BFE基础数据类型定义
- `bfe_config`: BFE配置加载相关代码
- `bfe_debug`: BFE模块调试开关相关代码
- `bfe_util`: BFE基础库相关代码

进程模型

BFE系统的转发实例由一组协同工作的进程组成（详见BFE简介章节）。这里以其中最核心的转发进程为重点介绍。

协程的分类

BFE的转发进程是由Go语言编写、基于Go协程实现的高并发网络服务器。BFE的转发进程中包含了几类重要的协程:

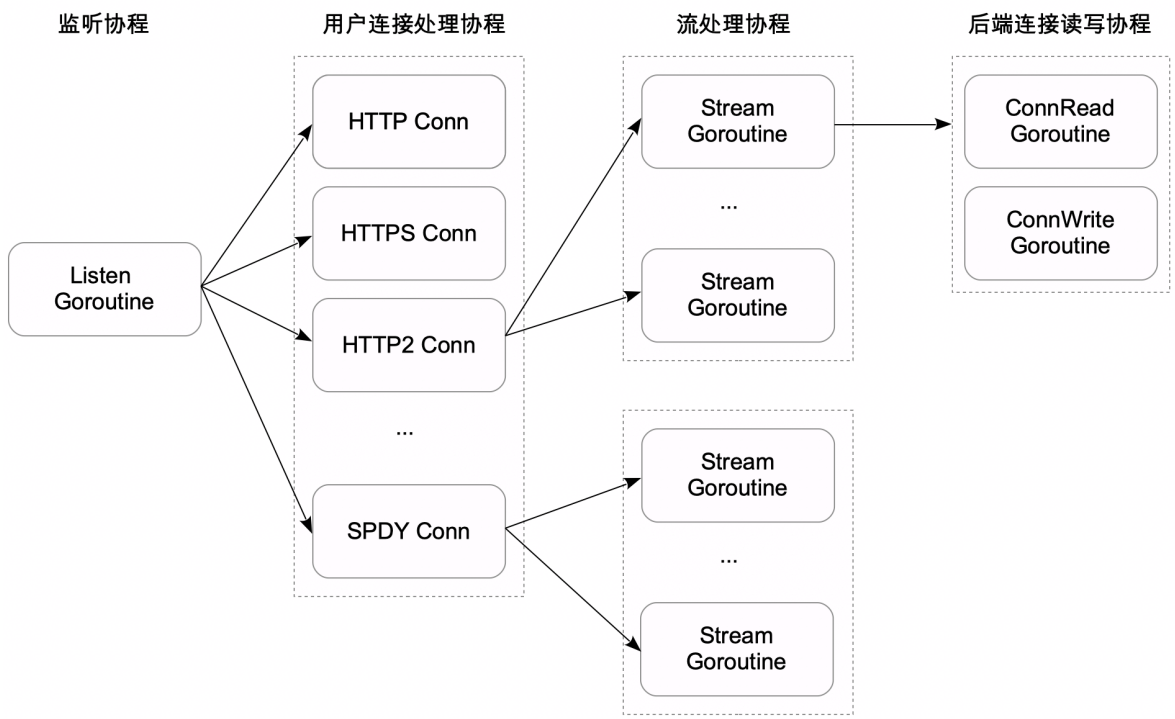
- **网络相关协程**
  - 用户连接的监听协程、用户连接的处理协程，以及协议相关的请求处理协程。
  - 后端连接的建立协程、后端连接的读写协程
- **管理相关协程**
  - 针对后端健康状态的检查协程
  - 监控及热加载请求的处理协程
- **辅助相关协程**
  - 扩展模块也可以创建协程，用于后台定期操作或异步执行处理
  - 例如：定期进行日志切割、异步更新缓存等

并发模型

BFE转发实例可以启动一个或多个监听协程。如果用户访问大量是短连接，单个监听协程仅使用的单个CPU核并可能成为瓶颈。这时可以通过适当调节监听协程数量来提升吞吐。

每个新到的用户连接将在独立的用户连接处理协程中并发处理。对于HTTP/HTTPS协议，用户连接处理协程串行读取请求并处理；对于HTTP2/SPDY协议，由于协议支持多路复用，在多个独立的流处理协程中并发处理请求。

在向后端实例转发请求并读取响应时，涉及到一组后端连接读写协程，分别负责将请求数据写往后端连接、从后端连接读取响应数据。



并发能力

BFE协程可以充分利用单机的多核CPU提升并发量及吞吐。但基于Go语言协程的并发模型也存在局限：

- 并发能力与CPU核数并非持续呈线性增长

  当单机核数非常大时，由于锁竞争继续增加核数可能并不能提升极限性能。在实践中，一般通过针对流量接入场景制定最佳机型套餐或容器规格，通过多实例来提供服务，从而达到线性提升整体性能的效果。

- 难以绑定到固定的CPU核以利用CPU亲缘性提升极限性能

异常恢复

由于转发程序的复杂性及快速迭代特点，潜在PANIC问题难以通过线下测试完全被发现。但在一些情况下触发PANIC后，会对转发集群稳定性带来非常严重的影响，例如由特定类型请求（Query of Death）触发。

因此，BFE所有网络相关协程都利用了Go语言内置的PANIC恢复机制，避免在连接/请求处理过程中由于未知Bug导致BFE转发实例大规模PANIC退出。

```
// bfe_server/http_conn.go:serve()

defer func() {
    if err := recover(); err != nil {
        log.Logger.Warn("panic: conn.serve(): %v, readTotal=%d,writeTotal=%d,reqNum=%d,%v\n%s",
        c.remoteAddr, c.session.ReadTotal, c.session.WriteTotal, c.session.ReqNum,
        err, gotrack.CurrentStackTrace(0))
        ...
    }
    ...
}()
```

在出现PANIC后，往往仅影响单个连接或请求。同时PANIC恢复阶段输出的上下文日志，也便于可高效分析定位出问题根因。对于一些难以在线下复现的问题，基于PANIC修复的经验，往往通过代码分析就能找到答案。

请求处理流程及响应

连接的建立

BFE的监听器处理协程（BfeServer.Serve函数)循环接受新到的客户端连接，并创建新协程处理该连接。

```
// bfe_server/http_server.go

func (srv *BfeServer) Serve(l net.Listener, raw net.Listener, proto string) error {
    ...
      for {
            // accept new connection
            rw, e := l.Accept()
            ...
            // start goroutine for new connection
            go func(rwc net.Conn, srv *BfeServer) {
                  c, err := newConn(rw, srv)
                  ...
                  c.serve()
            }(rw, srv)
      }
}
```

连接的处理

BFE的连接bfe_server.conn 执行serve函数处理该连接，主要包含：

- 步骤1. **回调点处理**：执行HandleAccept回调点的回调链函数

```
// bfe_server/http_conn.go

// Callback for HANDLE_ACCEPT
hl = c.server.CallBacks.GetHandlerList(bfe_module.HandleAccept)
if hl != nil {
    retVal = hl.FilterAccept(c.session)
    ...
}
```

- 步骤2. **握手及协商**：执行TLS握手（如果用户发起TLS连接）

在TLS握手成功后，执行HandleHandshake回调点的回调链函数

```
// bfe_server/http_conn.go

// Callback for HANDLE_HANDSHAKE
hl = c.server.CallBacks.GetHandlerList(bfe_module.HandleHandshake)
if hl != nil {
    retVal = hl.FilterAccept(c.session)
    ...
}
```

然后基于协商协议，选择并执行应用层协议Handler（HTTP2/SPDY/STREAM）

```
// bfe_server/bfe_server.go

tlsNextProto[tls_rule_conf.SPDY31] = bfe_spdy.NewProtoHandler(nil)
tlsNextProto[tls_rule_conf.HTTP2] = bfe_http2.NewProtoHandler(nil)
tlsNextProto[tls_rule_conf.STREAM] = bfe_stream.NewProtoHandler(
        &bfe_stream.Server{BalanceHandler: srv.Balance})
```

- 步骤3. **连接协议处理**：区分连接的协议，执行：

  - 如果是HTTP(S)连接，在当前协程中顺序读取请求并处理
  - 如果是HTTP2/SPDY连接，在新建协程中并发读取请求并处理
  - 如果是STREAM连接，在新建协程处理数据的双向转发（下文略去）

  关于协议的实现说明，详见核心协议实现一章。

请求的处理

BFE的连接对象bfe_server.conn 执行serveRequest函数处理请求。虽然HTTP/HTTPS/HTTP2/SPDY/使用不同方式传输数据，但BFE从协议层接收到HTTP请求后，在上层都转化为相同的内部请求类型(bfe_http.Request)，并执行统一的逻辑处理。

请求处理的具体流程如下：

- 步骤1. **回调点处理**

  执行HandleBeforeLoation回调点的回调链函数

```
// bfe_server/reverseproxy.go

// Callback for HandleBeforeLocation
hl = srv.CallBacks.GetHandlerList(bfe_module.HandleBeforeLocation)
if hl != nil {
    retVal, res = hl.FilterRequest(basicReq)
    ...
}
```

- 步骤2. **租户路由**：

  查找请求归属的租户。详见请求路由中的说明。

```
// bfe_server/reverseproxy.go

// find product
if err := srv.findProduct(basicReq); err != nil {
    ...
}
```

- 步骤3.**回调点处理**：

  执行HandleFoundProduct回调点的回调链函数
  ` go
  // bfe_server/reverseproxy.go

```
// Callback for HandleFoundProduct
hl = srv.CallBacks.GetHandlerList(bfe_module.HandleFoundProduct)
if hl != nil {
retVal, res = hl.FilterRequest(basicReq)
...
}
```

- 步骤4. **集群路由**：

  查找请求归属的目的集群。详见[请求路由](../routing/routing.md)中的说明。

```go
// bfe_server/reverseproxy.go

if err = srv.findCluster(basicReq); err != nil {
    ...
}
```

- 步骤5.**回调点处理**：执行HandleAfterLocation回调点的回调链函数

  ```go
  // bfe_server/reverseproxy.go

// Callback for HandleAfterLocation
hl = srv.CallBacks.GetHandlerList(bfe_module.HandleAfterLocation)
if hl != nil {
...
}
```

- 步骤6. **请求预处理**：

  对请求最终转发前，对请求进行预处理并获取转发参数（例如超时时间）

- 步骤7. **负载均衡及转发**：

  向下游集群转发HTTP请求。详见[负载均衡](../balancing/balancing.md)中的说明

```go
// bfe_server/reverseproxy.go

res, action, err = p.clusterInvoke(srv, cluster, basicReq, rw)
basicReq.HttpResponse = res
```

- 步骤8.**回调点处理**：执行HandleReadResponse回调点的回调链函数

  ```go
  // bfe_server/reverseproxy.go

// Callback for HandleReadResponse
hl = srv.CallBacks.GetHandlerList(bfe_module.HandleReadResponse)
if hl != nil {
...
}
```

- 步骤9. **响应发送**：向用户端发送响应

```go
// bfe_server/reverseproxy.go

err = p.sendResponse(rw, res, resFlushInterval, cancelOnClientClose)
if err != nil {
    ...
}
```

请求的结束

- 执行HandleRequestFinish回调点的回调链函数

```go
// bfe_server/reverseproxy.go

// Callback for HandleRequestFinish
hl := srv.CallBacks.GetHandlerList(bfe_module.HandleRequestFinish)
if hl != nil {
    ...
}
```

- 检查连接是否需关闭（例如请求被封禁或HTTP KeepAlive未启用）
- 如需关闭，连接将停止读取后续请求并执行关闭操作

连接的结束

连接在结束前，还需要执行以下操作：

- 执行HandleFinish回调点的回调链函数

```go
// bfe_server/http_conn.go

// Callback for HandleFinish
hl := srv.CallBacks.GetHandlerList(bfe_module.HandleFinish)
if hl != nil {
    hl.FilterFinish(c.session)
}
```

- 写出连接缓存区数据并关闭连接

模块框架

BFE定义了一套模块机制，通过编写模块插件来快速开发新功能。与模块相关的代码包含以下部分：

代码目录	说明
bfe_module	包含了模块基础类型的定义，例如模块的接口、模块回调类型、回调链类型、回调框架类型
bfe_moudles	包含了内置模块的实现，例如流量管理、安全防攻击、流量可见性等类别的各种模块
bfe_server	实现了连接/请求处理的生命周期的管理，并在关键处理阶段设置了回调点。注册在这些回调点的模块回调函数链，将被顺序执行，从而支持对连接/请求的定制化处理

模块基础类型

模块的定义

BFE模块代表一个高内聚、低耦合、实现特定流量处理功能（例如检测、过滤、改写等）的代码单元。

每个模块需满足如下模块接口(BfeModule)，其中：

- Name() 方法返回模块的名称
- Init() 方法用于执行模块的初始化

```
// bfe_module/bfe_module.go

type BfeModule interface {
    // Name returns the name of module.
    Name() string

    // Init initializes the module. The cbs are callback handlers for
    // processing connection or request/response. The whs are web monitor
    // handlers for exposing internal status or reloading specified
    // configuration. The cr is the root path of module configuration
    // files.
    Init(cbs *BfeCallbacks, whs *web_monitor.WebHandlers, cr string) error
}
```

模块除了需要实现以上基础接口外，往往还需实现特定回调接口，针对连接或请求实现自定义逻辑。关于回调接口的类型在下文详述。

模块的回调类型

在BFE中按需求场景定义了如下5种回调类型，用于处理连接、请求或响应。

```
// bfe_module/bfe_filter.go

// RequestFilter filters incomming requests and return a response or nil.
// Filters are chained together into a HandlerList.
type RequestFilter interface {
    FilterRequest(request *bfe_basic.Request) (int, *bfe_http.Response)
}

// ResponseFilter filters outgoing responses. This can be used to modify
// the response before it is sent.
type ResponseFilter interface {
    FilterResponse(req *bfe_basic.Request, res *bfe_http.Response) int
}

// AcceptFilter filters incoming connections.
type AcceptFilter interface {
    FilterAccept(*bfe_basic.Session) int
}

// ForwardFilter filters to forward request
type ForwardFilter interface {
    FilterForward(*bfe_basic.Request) int
}

// FinishFilter filters finished session(connection)
type FinishFilter interface {
    FilterFinish(*bfe_basic.Session) int
}
```

各回调类型的详细说明如下：

回调类型	说明	示例自定义逻辑
RequestFilter	回调函数用于处理到达的请求， 并可直接返回响应或继续向下执行	检测请求并限流；重定向请求； 修改请求内容并继续执行
ResponseFilter	回调函数用于处理到达的响应， 并可修改响应并继续向下执行	修改响应的内容；对响应进行压缩 统计响应状态码；记录请求日志
AcceptFilter	回调函数用于处理到达的连接， 并可关闭连接或并继续向下执行	检测连接并限流； 记录连接的TLS握手信息并继续执
ForwardFilter	回调函数用于处理待转发的请求， 并可修改请求的目的后端实例	修改请求的目标后端实例并继续执
FinishFilter	回调函数用于处理完成的连接	记录会话日志并继续执行

回调函数的返回值决定了回调函数执行完成时后续的操作。回调函数的返回值有以下几种可能的取值：

```
// bfe_module/bfe_handler_list.go
```

```
// Return value of handler.
const (
    BfeHandlerFinish   = 0 // to close the connection after response
    BfeHandlerGoOn     = 1 // to go on next handler
    BfeHandlerRedirect = 2 // to redirect
    BfeHandlerResponse = 3 // to send response
    BfeHandlerClose    = 4 // to close the connection directly, with no data sent.
)
```

各返回值代表含义如下：

回调函数返回值	含义说明
BfeHandlerFinish	直接返回响应，在响应发送完成后关闭连接
BfeHandlerGoOn	继续向下执行
BfeHandlerRedirect	直接返回重定向响应
BfeHandlerResponse	直接返回指定的响应
BfeHandlerClose	直接关闭连接

回调链类型

回调链(HandlerList)代表了多个回调函数构成的有序列表。一个回调链中的回调函数类型是相同的。

```
// bfe_module/bfe_handler_list.go

// HandlerList type.
const (
    HandlersAccept   = 0 // for AcceptFilter
    HandlersRequest  = 1 // for RequestFilter
    HandlersForward  = 2 // for ForwardFilter
    HandlersResponse = 3 // for ResponseFilter
    HandlersFinish   = 4 // for FinishFilter
)

type HandlerList struct {
    handlerType int        // type of handlers (filters)
    handlers    *list.List // list of handlers (filters)
}
```

回调框架

回调框架管理了BFE中所有回调点对应的回调链。

```
// bfe_module/bfe_callback.go

// Callback point.
const (
    HandleAccept         = 0
    HandleHandshake      = 1
```

```
    HandleBeforeLocation = 2
    HandleFoundProduct   = 3
    HandleAfterLocation  = 4
    HandleForward        = 5
    HandleReadResponse   = 6
    HandleRequestFinish  = 7
    HandleFinish         = 8
)

type BfeCallbacks struct {
    callbacks map[int]*HandlerList
}
```

不同回调点可注册的回调函数类型不同，具体情况如下：

回调点	回调点含义	回调函数类型
HandleAccept	与用户端连接建立成功后	AcceptFilter
HandleHandshake	与用户端TLS握手成功后	AcceptFilter
HandleBeforeLocation	请求执行租户路由前	RequestFilter
HandleFoundProduct	请求执行租户路由后	RequestFilter
HandleAfterLocation	请求执行集群路由后	RequestFilter
HandleForward	请求向后端转发前	ForwardFilter
HandleReadResponse	读取到后端响应头部时	ResponseFilter
HandleRequestFinish	向用户端发送响应完成时	ResponseFilter
HandleFinish	与用户端连接处理完成时	FinishFilter

连接/请求处理及回调函数的调用

BFE在转发过程对连接/请求对处理及回调点如BFE的模块插件机制图示。在各回调点，BFE将查询指定回调点注册的回调链，并按顺序执行回调链中的各回调函数。

例如，BFE在接收到用户请求头时，将查询在HandleBeforeLocation回调点注册的回调链，然后按序执行各回调函数，并基于返回值决定后续操作。

```
// bfe_server/reverseproxy.go

// Get callbacks for HandleBeforeLocation
hl = srv.CallBacks.GetHandlerList(bfe_module.HandleBeforeLocation)
if hl != nil {
    // process FilterRequest handlers
    retVal, res = hl.FilterRequest(basicReq)
    basicReq.HttpResponse = res
```

```
    switch retVal {
    case bfe_module.BfeHandlerClose:
        // Close the connection directly (without reply)
        action = closeDirectly
        return

    case bfe_module.BfeHandlerFinish:
        // Close the connection after response
        action = closeAfterReply
        basicReq.BfeStatusCode = bfe_http.StatusInternalServerError
        return

    case bfe_module.BfeHandlerRedirect:
        // Make an redirect
        Redirect(rw, req, basicReq.Redirect.Url, basicReq.Redirect.Code, basicReq.Redirect.Header)
        isRedirect = true
        basicReq.BfeStatusCode = basicReq.Redirect.Code
        goto send_response

    case bfe_module.BfeHandlerResponse:
        // Send the generated response
        goto response_got
    }
}
```

请求路由

BFE可同时接入多个租户的流量，每个租户包含多个集群，各集群分别处理不同类型业务的请求。**请求路由**指在BFE转发请求过程中，确定请求所属的租户及目标集群的过程。

BFE路由转发的过程可以参考BFE的转发模型中的说明。

关键数据结构

在路由模块 bfe_route/host_table.go 中定义了用于管理路由规则的数据结构。其中主要包含以下类型：

域名表(hostTable)

hostTable用于管理域名与租户标识的映射关系。

hostTable 是一个Trie树类型的数据结构，以便于支持泛域名查找。简单来说，该Trie树中每个叶子节点到根节点的路径代表了一个域名，叶子节点存有租户名称。

Trie树节点的数据类型如下：

- Children指向该节点的所有子节点
- Entry存放该节点到根节点路径所代表的域名(例x.example.com)，对应的租户名称
- Splat存放该节点到根节点路径所代表的泛域名(例*.x.example.com)，对应的租户名称

```
// bfe_route/trie/trie.go

type trieChildren map[string]*Trie

type Trie struct {
    Children   trieChildren
    Entry      interface{}
    Splat      interface{}
}
```

VIP表(vipTable)

vipTable用于管理VIP与租户标识的映射关系。
vipTalbe 是一个哈希表类型的数据结构。其中键代表VIP，值代表租户名称。

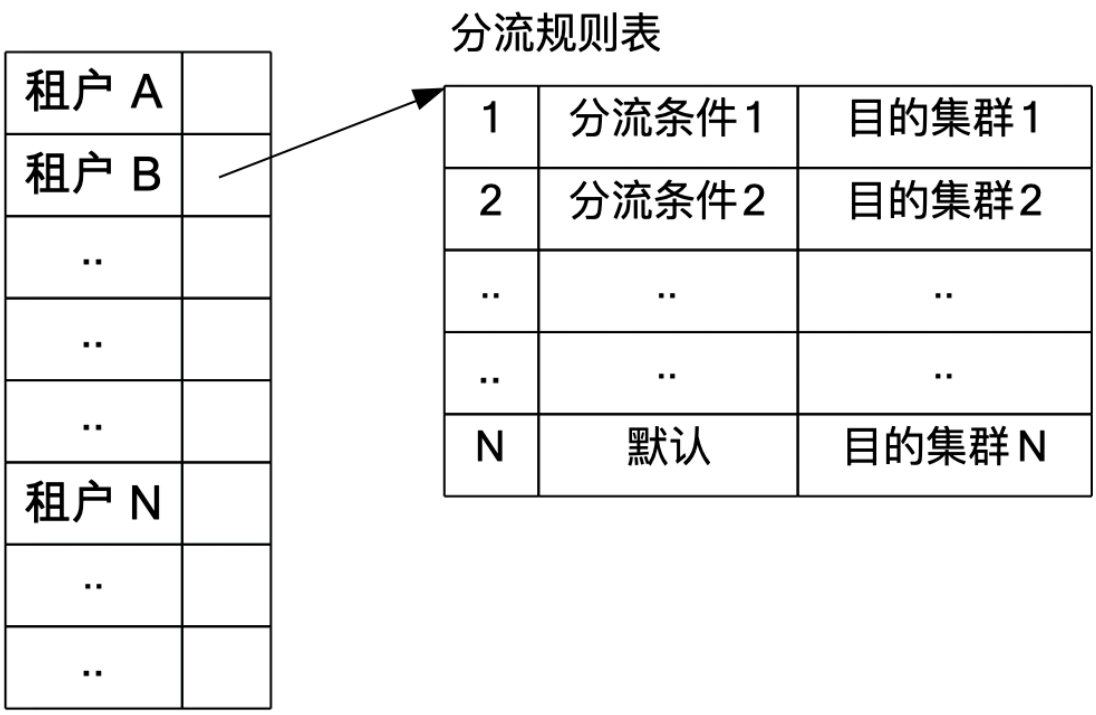
```
// bfe_config/bfe_route_conf/vip_rule_conf/vip_table_load.go

type Vip2Product map[string]string
```

分流规则表(productRouteTable)

productRouteTable用于管理各租户的分流规则表。
productRouteTable 是一个哈希表类型的数据结构。其中的键代表租户名称，值为该租户的分流规则表。
各租户的分流规则表包含了一组有序的分流规则。每条规则由规则条件及目的集群组成。

```
// bfe_config/bfe_route_conf/route_rule_conf/route_table_load.go

type ProductRouteRule map[string]RouteRules
```

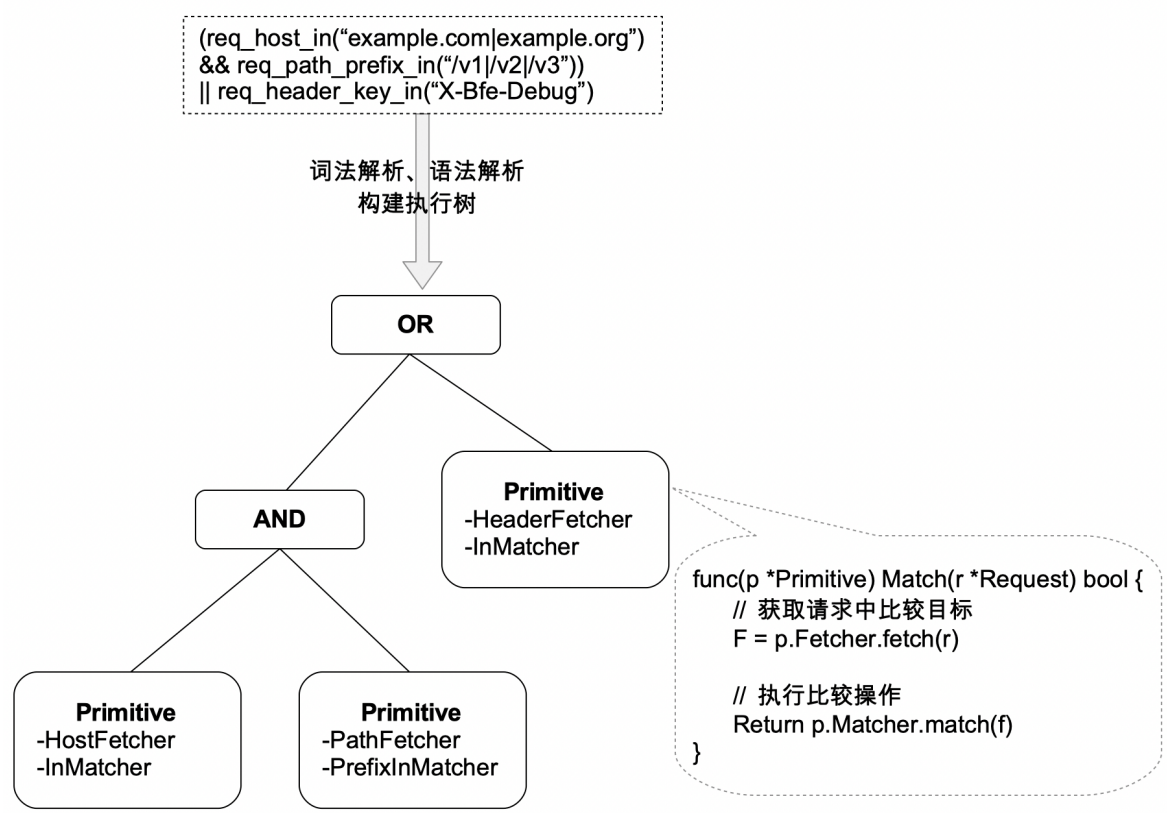
```
// bfe_route/host_table.go

type HostTable struct {
    //...

    productRouteTable route_rule_conf.ProductRouteRule
}
```



分流条件(Condition)

分流条件是一个由条件原语及操作符组成的条件表达式。条件表达式的语法可详见BFE的路由转发机制中的说明。

条件表达式在BFE的内部数据结构，是一个中缀表达式形式的二叉树。二叉树的非叶子节点代表了操作符。叶子节点代表条件原语。对请求执行与分流条件匹配时，相当于对该中缀表达式进行求值。其返回值是布尔类型，代表请求是否匹配规则。



目的租户路由

HostTable的LookupHostTagAndProduct()实现了目的租户的查找。

查找的基本流程如下：

- 步骤一: 根据请求的Host字段值，尝试查找hostTable并返回命中的租户名称。
- 步骤二: 如果查找失败，根据请求的访问VIP值，尝试查找vipTable并返回命中的租户名称。
- 步骤三: 如果查找失败，返回缺省的租户名称。

```
// bfe_route/host_table.go

// LookupHostTagAndProduct find hosttag and product with given hostname.
func (t *HostTable) LookupHostTagAndProduct(req *bfe_basic.Request) error {
    hostName := req.HttpRequest.Host

    // lookup product by hostname
    hostRoute, err := t.findHostRoute(hostName)

    // if failed, try to lookup product by visited vip
    if err != nil {
        if vip := req.Session.Vip; vip != nil {
            hostRoute, err = t.findVipRoute(vip.String())
        }
    }

    // if failed, use default proudct
    if err != nil && t.defaultProduct != "" {
        hostRoute, err = route{product: t.defaultProduct}, nil
    }
```

```
    // set hostTag and product
    req.Route.HostTag = hostRoute.tag
    req.Route.Product = hostRoute.product
    req.Route.Error = err

    return err
}
```

目的集群路由

HostTable的LookupCluster()实现了目的集群的查找。

查找的基本流程如下：

- 步骤一：根据请求的归属租户名称查找分流规则表。
- 步骤二：在租户的分流规则表中，按顺序将请求与表中各规则的条件进行匹配。
- 步骤三：对最先匹配到的规则，返回所包含的目的集群。

注意分流规则表的最后一条规则是缺省规则，如果执行到最后一条规则，最终将返回缺省目的集群。

```
// bfe_route/host_table.go

// LookupCluster find clusterName with given request.
func (t *HostTable) LookupCluster(req *bfe_basic.Request) error {
    var clusterName string
    // get route rules
    rules, ok := t.productRouteTable[req.Route.Product]
    if !ok {
        ...
    }

    // matching route rules
    for _, rule := range rules {
        if rule.Cond.Match(req) {
            clusterName = rule.ClusterName
            break
        }
    }
    ...

    // set clusterName
    req.Route.ClusterName = clusterName
    return nil
}
```

负载均衡

负载均衡

租户的**后端集群**一般包含了多个子集群，每个**后端子集群**分别部署在不同地域不同机房中。每个子集群包含了一组处理能力差异化的**后端实例**。

业务通常采用多个后端子集群方式来管理后端服务，可以带来如下好处：

1. 多个子集群属于不同的故障隔离域，某个子集群出现故障（例如分级变更上线异常），可以快速切换流量止损并提升整体可用性
2. 多个子集群分布在离用户更近位置，可支持就近处理用户请求并优化访问体验
3. 多个子集群同时服务提升整体容量，以满足高并发的互联网用户请求

相对应的，BFE的流量**负载均衡**包含了两个层级：

1. **全局负载均衡**(GSLB)：BFE集群利用全量的用户流量、后端容量、网络情况，在多个后端子集群之间实现负载均衡。实现全局近实时决策优化，满足就近分发、调度止损、过载保护等目标。
2. **分布式负载均衡**(SLB)：BFE实例分别独立的，将某个子集群的流量，在其多个后端实例之间实现负载均衡。实现细粒度实时负载均衡，满足实例均衡、异常实例屏蔽、重试容错等目标。

全局负载均衡

BFE在后端集群的多个子集群之间，采用基于WRR算法的负载均衡策略。 算法实现详见均衡模块 bfe_balance/bal_gslb/bal_gslb.go:subClusterBalance()。

全局负载均衡算法包括如下两个执行步骤：

步骤一、请求亲缘性及分桶处理

用户请求可能具有亲缘性，即需要常态将特定请求固定转发给某个子集群。例如:

1. 来自某个用户的请求，常态固定转发给某个子集群处理，以便于用户分组管理
2. 包含某个查询的请求，常态固定转发给某个子集群处理，以满足缓存友好性

为实现感知请求内容的负载均衡，BFE支持以下三种方式标识请求：

1. 基于请求指定Header或Cookie

2. 基于请求来源IP

3. 优先基于请求指定Header或Cookie，缺失情况下基于请求IP

```
// bfe_balance/bal_gslb/bal_gslb.go

switch *bal.hashConf.HashStrategy {
      case cluster_conf.ClientIdOnly:
            hashKey = getHashKeyByHeader(req, *bal.hashConf.HashHeader)

      case cluster_conf.ClientIpOnly:
            hashKey = clientIP

      case cluster_conf.ClientIdPreferred:
            hashKey = getHashKeyByHeader(req, *bal.hashConf.HashHeader)
```

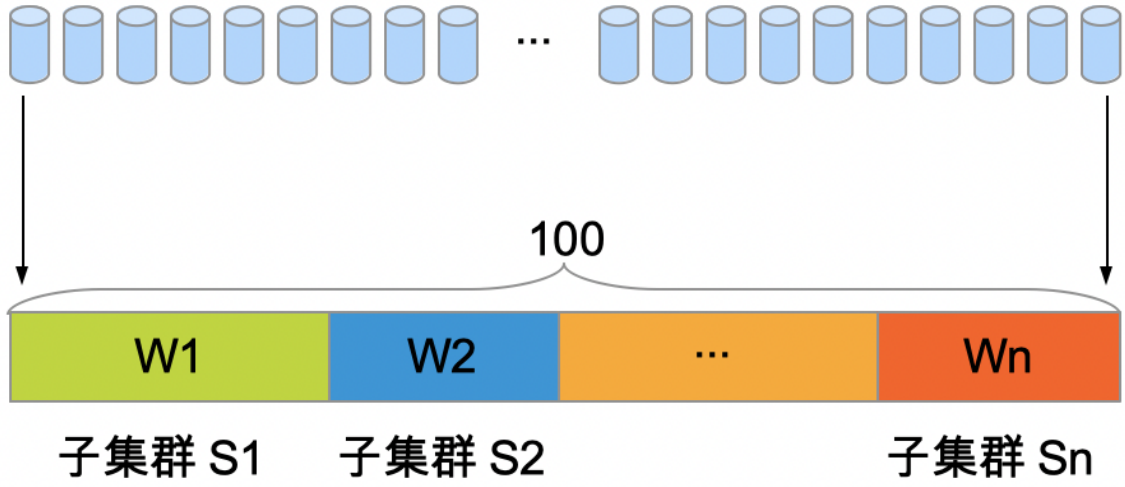
```
            if hashKey == nil {
                hashKey = clientIP
            }
}

// if hashKey is empty, use random value
if len(hashKey) == 0 {
        hashKey = make([]byte, 8)
        binary.BigEndian.PutUint64(hashKey, rand.Uint64())
}

return hashKey
```

算法将用户的请求切分为100个桶，并基于指定策略(例如基于请求Cookie中的用户ID)，将特定请求固定哈希到其中某个桶

```
// bfe_balance/bal_slb/bal_rr.go

func GetHash(value []byte, base uint) int {
    var hash uint64

    if value == nil {
        hash = uint64(rand.Uint32())
    } else {
        hash = murmur3.Sum64(value)
    }
    return int(hash % uint64(base))
}
```

步骤二、请求桶分配及均衡

算法将100个桶，分配给权重和为100的所有子集群。



例如后端集群包含3个子集群S1/S2/S3，其权重分别为W1/W2/W3 且 W1+W2+W3=100, 则:

- 子集群S1分配到桶号范围为 [0, W1)
- 子集群S2分配到桶号范围为 [W1, W1+W2)
- 子集群S3分配到桶号范围为 [W1+W2,100)

```
// bfe_balance/bal_gslb/bal_gslb.go

// Calculate bucket number for incoming request
w = bal_slb.GetHash(value, uint(bal.totalWeight))

for i := 0; i < len(bal.subClusters); i++ {
        subCluster = bal.subClusters[i]

        // Find target subcluster for specified bucket
        w -= subCluster.weight
        if w < 0 {
              return subCluster, nil
        }
}

// Never come here
return nil, err.News("subcluster balancing failure")
```

分布式负载均衡

BFE在后端子集群的多个实例之间，支持多种负载均衡策略，包括:

- WRR: 加权轮训策略
- WLC: 加权最小连接数策略

算法实现详见 bfe_balance/bal_slb/bal_rr.go:Balance()。下文仅以WRR算法为例结合示例场景重点介绍。

步骤一、初始随机排序后端实例列表

BFE各转发实例在初始加载（或更新）后端子集群实例时，对实例列表预处理并随机排序。

```
// bfe_config/bfe_cluster_conf/cluster_table_conf/cluster_table_load.go

func (allClusterBackend AllClusterBackend) Shuffle() {
     for _, clusterBackend := range allClusterBackend {
           for _, backends := range clusterBackend {
                 backends.Shuffle()
           }
     }
}
```

这是为了避免在BFE转发实例较多情况下，由于各BFE转发实例产生相同的均衡结果，导致负载不均的情况。举例说明：假如上游的BFE集群规模是1000个实例，实际到达到用户请求是1000QPS，下游的后端子集群包含了10个后端实例，则可能周期性出现：

- 第一秒各BFE转发实例仅向第一个后端实例转发1000个请求
- 第二秒各BFE转发实例仅向第二个后端实例转发1000个请求
- 依次类推

BFE通过预先随机打乱后端子集群实例顺序，来避免以上负载不均的问题。

步骤二、平滑均衡选择后端实例

在后端实例权重差异较大情况下，也可能会出现负载不均的情况，表现为：虽然一个周期内各实例选中次数满足相应权重比例，但可能出现权重较大的实例连续多次被选择，而使得其它低权重的实例较长时间未分配流量。

为避免负载不均的情况，BFE使用了如下的WRR算法，简化的算法伪代码如下：

```
// bfe_balance/bal_slb/bal_rr.go

func smoothBalance(backs BackendList) (*backend.BfeBackend, error) {
      var best *BackendRR
      total, max := 0, 0

      for _, backendRR := range backs {
                backend := backendRR.backend

                // select backend with greatest current weight
                if best == nil || backendRR.current > max {
                      best = backendRR
                      max = backendRR.current
                }
                total += backendRR.current

                // update current weight
                backendRR.current += backendRR.weight
      }

      // update current weight for chosen backend
      best.current -= total

      return best.backend, nil
}
```

算法针对每个实例维护了两个参数：实例权重（Weight）、实例偏好指数（Current）。每次算法从所有可用后端列表中选出最佳后端的过程如下：

- 选择实例偏好指数最大的实例
- 更新各实例偏好指数：
  - 对各实例的偏好指数值，分别加上该实例权重
- 对于选中实例，对其偏好指数值减去所有实例的偏好指数（在加上实例权重之前的值）总和

例如: 假设后端子集群包含三个后端实例a/b/c，权重分别为 5/1/1。如果基于以上算法，选择的过程如下：

轮数	选择前偏好指数	选中节点	选择后偏好指数
1	5 1 1	a	3 2 2
2	3 2 2	a	1 3 3
3	1 3 3	b	6 -3 4
4	6 -3 4	a	4 -2 5
5	4 -2 5	c	9 -1 -1
6	9 -1 -1	a	7 0 0
7	7 0 0	a	5 1 1

核心协议实现

BFE的HTTP/HTTP2/SPDY/WebSocket/TLS等网络协议基于Go语言官方开源协议库。为更好满足反向代理的需求场景，在BFE中进行了二次定制开发，包括性能优化、防攻击机制完善、兼容性改进、增加探针等。

本章重点介绍HTTP/HTTP2协议的实现。SPDY的实现与HTTP2的实现非常相似，这里不再赘述。其它协议的实现可参考BFE的代码组织中的说明查阅对应源码，也可在BFE开源社区提问交流。

HTTP协议

HTTP代码的组织

在bfe_http目录下包含如下代码：

```
$ ls bfe/bfe_http
chunked.go        cookie.go        header_test.go   readrequest_test.go   response.go          sniff.go      tr
ansfer_test.go
chunked_test.go   cookie_test.go   httputil         request.go            response_test.go     state.go      tr
ansport.go
client.go         eof_reader.go    lex.go           request_test.go       response_writer.go   status.go
common.go         header.go        lex_test.go      requestwrite_test.go  responsewrite_test.go transfer.go
```

各文件的功能说明如下：

类别	文件名或目录	说明
基础类型	common.go	HTTP基础数据类型定义
	state.go	HTTP协议内部状态指标
	eof_reader.go	EofReader类型定义， 实现了io.ReadCloser接口，并永远返回EOF
协议消息	request.go	HTTP请求类型的定义、读取及发送
	response.go	HTTP响应类型的定义、读取及发送
	header.go	HTTP头部类型定义及相关操作
	cookie.go	HTTP Cookie字段的处理
	status.go	HTTP响应状态码定义
	lex.go	HTTP合法字符表
消息收发	client.go	RoundTrpper接口定义， 支持并发的发送请求并获取响应

类别	文件名或目录	说明
	transport.go	HTTP连接池管理，实现了RoundTrpper接口，在反向代理场景用于管理与后端的HTTP通信
	transfer.go	transferWriter/transfterReader类型定义，在反向代理场景用于向后端流式发送请求及读取响应
	response_writer.go	ResponseWriter类型定义，在反向代理场景用于构造响应并发送
辅助工具	httputil	HTTP相关辅助函数
	chunked.go	HTTP Chunked编码处理
	sniff.go	HTTP MIME检测算法实现 (https://mimesniff.spec.whatwg.org)

从用户读取HTTP请求

在 bfe_http/request.go文件中实现了从HTTP连接上读取一个HTTP请求，包括以下步骤：

- 读取HTTP请求行并解析请求方法、URI及协议版本号
- 读取HTTP请求头部并解析
- 读取HTTP请求主体

```
// bfe_http/request.go

// ReadRequest reads and parses a request from b.
func ReadRequest(b *bfe_bufio.Reader, maxUriBytes int) (req *Request, err error) {
    tp := newTextprotoReader(b)
    req = new(Request)
    req.State = new(RequestState)

    // Read first line (eg. GET /index.html HTTP/1.0)
    var s string
    if s, err = tp.ReadLine(); err != nil {
        return nil, err
    }
  ...

    // Parse request method, uri, proto
    var ok bool
    req.Method, req.RequestURI, req.Proto, ok = parseRequestLine(s)
    if !ok {
        return nil, &badStringError{"malformed HTTP request", s}
     }
    rawurl := req.RequestURI
    if req.ProtoMajor, req.ProtoMinor, ok = ParseHTTPVersion(req.Proto); !ok {
        return nil, &badStringError{"malformed HTTP version", req.Proto}
    }
```

```
    if req.URL, err = url.ParseRequestURI(rawurl); err != nil {
        return nil, err
    }
  ...

    // Read and parser request header
    mimeHeader, headerKeys, err := tp.ReadMIMEHeaderAndKeys()
    if err != nil {
        return nil, err
    }
    req.Header = Header(mimeHeader)
    req.HeaderKeys = headerKeys
  ...

  // Read request body
    err = readTransfer(req, b)
    if err != nil {
        return nil, err
    }

    return req, nil
}
```

注意在最后一个步骤中，readTransfer(req, b)并未直接将请求内容立即读取到内存中。如果这样做，会大大增加反向代理的内存开销，同时也会增加请求转发延迟。

在readTransfer函数中，根据请求方法、传输编码、请求主体长度，返回满足io.ReadCloser接口类型的不同实现，用于按需读取请求内容。

```
// bfe_http/transfer.go

// Prepare body reader. ContentLength < 0 means chunked encoding
// or close connection when finished, since multipart is not supported yet
switch {
case chunked(t.TransferEncoding):
        if noBodyExpected(t.RequestMethod) {
            t.Body = EofReader
        } else {
            t.Body = &body{src: newChunkedReader(r), hdr: msg, r: r, closing: t.Close}
        }

case realLength == 0:
        t.Body = EofReader

case realLength > 0:
        // set r for peek data from body
        t.Body = &body{src: io.LimitReader(r, realLength), r: r, closing: t.Close}

default:
        // realLength < 0, i.e. "Content-Length" not mentioned in header
        if t.Close {
            // Close semantics (i.e. HTTP/1.0)
            t.Body = &body{src: r, closing: t.Close}
        } else {
            // Persistent connection (i.e. HTTP/1.1)
            t.Body = EofReader
        }
}
```

向后端转发请求并获取响应

在 bfe_http/transport.go中Transport类型实现了RoundTripper接口，支持发送请求并获取响应。主要包括以下步骤：

- 检查请求的合法性
- 从连接池获取到目地后端的闲置连接，或新建连接（如无闲置连接）
- 使用该连接发送请求，并读取响应

连接的数据类型是persistConn，包含的核心成员如下：

```
// bfe_http/transport.go

// persistConn wraps a connection, usually a persistent one
// (but may be used for non-keep-alive requests as well)
type persistConn struct {
    t         *Transport
    cacheKey string // its connectMethod.String()
    conn      net.Conn
    closed    bool                // whether conn has been closed

    reqch     chan requestAndChan // written by roundTrip; read by readLoop
    writech   chan writeRequest   // written by roundTrip; read by writeLoop
    closech   chan struct{}       // broadcast close when readLoop (TCP connection) closes
    ...
}
```

同时，persistConn包含两个相关协程 writeLoop()和readLoop()，分别用于向后端连接发送请求及读取响应。

```
// bfe_http/transport.go

func (pc *persistConn) writeLoop() {
    defer close(pc.closech)
    ...
    for {
        select {
        case wr := <-pc.writech:
            ...
            // Write the HTTP request and flush buffer
            err := wr.req.Request.write(pc.bw, pc.isProxy, wr.req.extra)
            if err == nil {
                err = pc.bw.Flush()
            }
            if err != nil {
                err = WriteRequestError{Err: err}
                pc.markBroken()
            }
            // Return the write result
            wr.ch <- err
        case <-pc.closech:
            return
        }
    }
}

func (pc *persistConn) readLoop() {
    defer close(pc.closech)
    ...
    alive := true
```

```
    for alive {
        ...
        rc := <-pc.reqch
        var resp *Response
        if err == nil {
            // Read the HTTP response
            resp, err = ReadResponse(pc.br, rc.req)
            ...
        }
        ...
        if err != nil {
            pc.close()
        } else {
            ...
            // Wrapper the HTTP Body
            resp.Body = &bodyEOFSignal{body: resp.Body}
        }
        ...

        // Return the read result
        if err != nil {
            err = ReadRespHeaderError{Err: err}
        }
        rc.ch <- responseAndError{resp, err}
        ...
    }
}
```

向用户回复HTTP响应

反向代理通过ResponseWriter接口来构造及发送响应，包括以下接口：

- Header(): 通过该方法设置响应头部
- WriteHeader(): 通过该方法设置响应状态码并发送响应头部
- Write(): 通过该方法发送响应主体数据

```
// bfe_http/response_writer.go

// A ResponseWriter interface is used by an HTTP handler to
// construct an HTTP response.
type ResponseWriter interface {
    // Header returns the header map that will be sent by WriteHeader.
    // Changing the header after a call to WriteHeader (or Write) has
    // no effect.
    Header() Header

    // Write writes the data to the connection as part of an HTTP reply.
    // If WriteHeader has not yet been called, Write calls
    // WriteHeader(http.StatusOK) before writing the data.
    // If the Header does not contain a Content-Type line, Write adds a
    // Content-Type set to the result of passing the initial 512 bytes of
    // written data to DetectContentType.
    Write([]byte) (int, error)

    // WriteHeader sends an HTTP response header with status code.
    // If WriteHeader is not called explicitly, the first call to Write
    // will trigger an implicit WriteHeader(http.StatusOK).
    // Thus explicit calls to WriteHeader are mainly used to
    // send error codes.
```

```
    WriteHeader(int)
}
```

在bfe_server/response.go文件中实现了ResponseWriter接口，用于发送HTTP/HTTPS响应。

HTTP2协议

HTTP2代码的组织

在bfe_http2目录下包含如下代码：

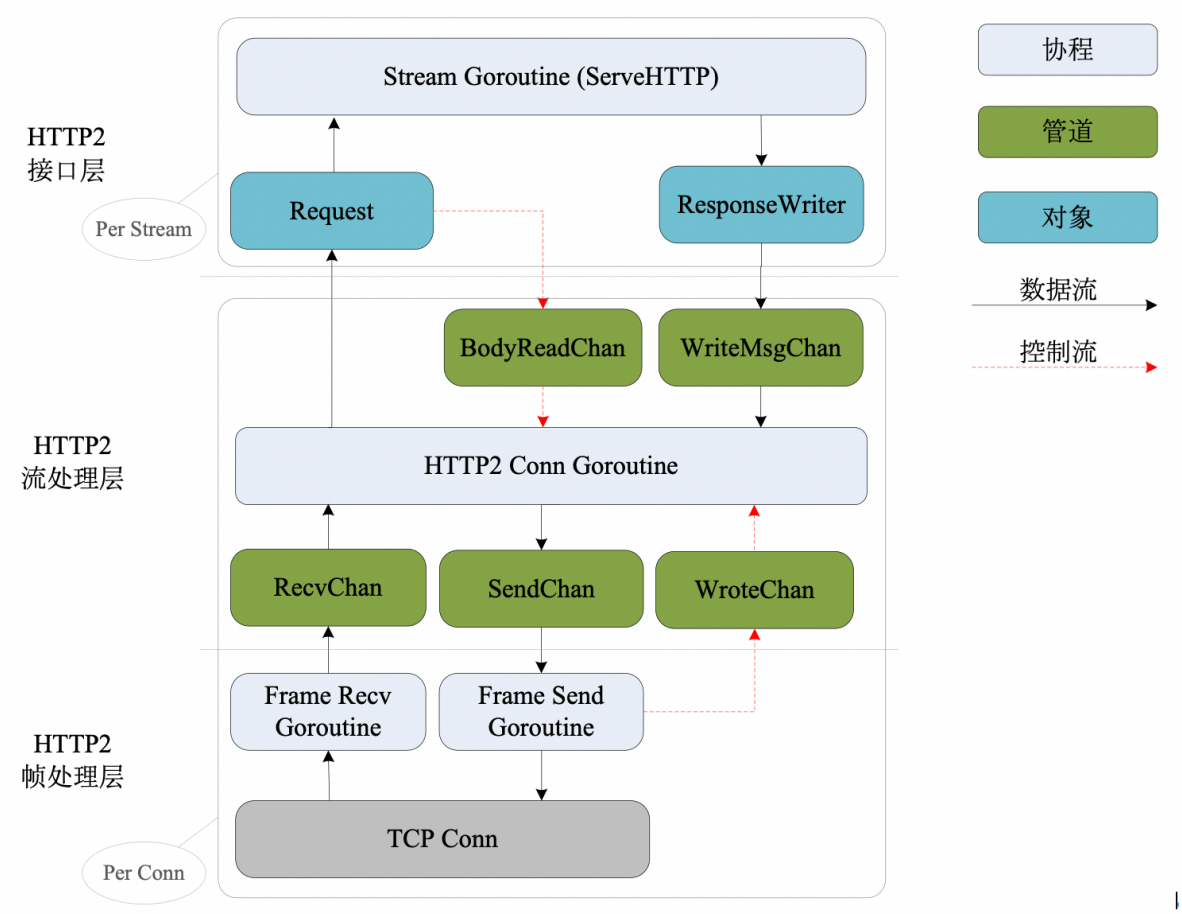
```
$ls bfe/bfe_http2
errors.go       flow_test.go   headermap.go  http2_test.go     server_test.go  transport.go    z_spec_test.go
errors_test.go  frame.go       hpack         priority_test.go  state.go        write.go
flow.go         frame_test.go  http2.go      server.go         testdata        writesched.go
```

各文件的功能说明如下：

类别	文件名或子目录	说明
流处理层	server.go	HTTP2协议连接核心处理逻辑
	flow.go	HTTP2流量控制窗口
	writesched.go	HTTP2协议帧发送优先级队列
帧处理层	frame.go	HTTP2协议帧定义及解析
	write.go	HTTP2协议帧发送方法
	hpack/	HTTP2协议头部压缩算法HPACK
基础数据类型	headermap.go	HTTP2常见请求头部定义
	errors.go	HTTP2协议错误定义
	state.go	HTTP2协议内部状态指标
辅助工具	transport.go	封装了HTTP2客户端；仅用于与后端实例通信

HTTP2连接处理模块

BFE在接收到一个HTTP2连接后，除了创建连接处理主协程，还会创建多个子协程配合完成协议逻辑的处理。单个HTTP2协议连接处理模块结构如图所示。



模块内部结构自底向上划分为三个层级：

帧处理层

- 帧处理层实现HTTP2协议帧序列化、压缩及传输
- 帧处理层包含两个独立收发协程，分别负责协议帧的接收与发送
- 帧处理层与流处理层通过管道通信 (RecvChan/SendChan/WroteChan)

流处理层

- 流处理层实现协议核心逻辑，例如：流创建、流数据传输、流关闭、多路复用、流优先级、流量控制等
- 流处理层为每流创建Request/ResponseWriter实例，并在独立协程中运行应用逻辑

接口层

- 为HTTP应用Handler提供标准Request/ResponseWriter实现，屏蔽HTTP2协议数据传输细节
- HTTP应用Handler运行在Stream Goroutine协程中
- HTTP应用Handler通过Request实例获取HTTP 请求（读取自特定HTTP2流）
- HTTP应用Handler通过ResponseWriter实例发送HTTP响应（发往特定HTTP2流）

HTTP2连接相关协程及关系

每个HTTP2连接的各协程，基于CSP(Communicating Sequential Processes)模型协作，具体如下：

帧处理层的协程

每个HTTP2连接包含两个读写协程，分别负责读取或发送HTTP2协议帧，其中：

- 帧接收协程(Frame Recv Goroutine) 从连接上读取HTTP2协议帧并放入帧接收队列

```
// bfe_http2/server.go

// readFrames is the loop that reads incoming frames.
// It's run on its own goroutine.
func (sc *serverConn) readFrames() {
    gate := make(gate)
    gateDone := gate.Done
    for {
        f, err := sc.framer.ReadFrame()
        ...

        // Send the frame to readFrameCh
        select {
        case sc.readFrameCh <- readFrameResult{f, err, gateDone}:
        case <-sc.doneServing:
            return
        }

        // Waiting for the frame to be processed
        select {
            case <-gate:
            case <-sc.doneServing:
                return
        }
        ...
    }
}
```

- 帧发送协程(Frame Send Goroutine) 从帧发送队列获取帧，并写入连接，同时将写结果放入写结果队列WroteChan

```
// bfe_http2/server.go

// writeFrames runs in its own goroutine and writes frame
// and then reports when it's done.
func (sc *serverConn) writeFrames() {
    var wm frameWriteMsg
    var err error

    for {
        // get frame from sendChan
        select {
        case wm = <-sc.writeFrameCh:
        case <-sc.doneServing:
            return
        }

        // write frame
        err = wm.write.writeFrame(sc)
        log.Logger.Debug("http2: write Frame: %v, %v", wm, err)

        // report write result
        select {
        case sc.wroteFrameCh <- frameWriteResult{wm, err}:
        case <-sc.doneServing:
            return
```

```
        }
    }
}
```

流处理层的协程

主协程与其它协程通过管道(golang Chan)进行通信，例如:

- BodyReadChan：请求处理协程读取请求Body后，通过BodyReadChan向主协程发送读结果消息，主协议接收到消息执行流量控制操作并更新流量控制窗口
- WriteMsgChan: 请求处理协程发送响应后，通过WriteMsgChan向主协程发送写申请消息，主协议接收到消息后，转换为HTTP2数据帧并放入流发送队列。在合适到时机
- ReadChan/SendChan/WroteChan：从连接上获取或发送HTTP2协议帧

```
// bfe_http2/server.go

func (sc *serverConn) serve() {
    ...

      // Write HTTP2 Settings frame and read preface.
    sc.writeFrame(frameWriteMsg{write: writeSettings{...}})
    err := sc.readPreface()
    ...

    // Start readFrames/writeFrames goroutines.
    go sc.readFrames()
    go sc.writeFrames()

    for {
        select {
        case wm := <-sc.wantWriteFrameCh:
            sc.writeFrame(wm)
        case res := <-sc.wroteFrameCh:
            sc.wroteFrame(res)
        case res := <-sc.readFrameCh:
            if !sc.processFrameFromReader(res) {
            return
            }
            ...
        case m := <-sc.bodyReadCh:
            sc.noteBodyRead(m.st, m.n)
        case <-sc.closeNotifyCh: // graceful shutdown
            sc.goAway(ErrCodeNo)
            sc.closeNotifyCh = nil
        ...
        }
    }
}
```

接口层的协程

每个HTTP2连接为应用层封装了Request对象及ResponseWriter对象，并创建独立的请求处理协程（Stream Goroutine）处理请求并返回响应

- Stream Goroutine 从Request对象中获取请求
- Stream Goroutine 向ResponseWriter对象发送响应

```
// bfe_http2/server.go

func (sc *serverConn) processHeaders(f *MetaHeadersFrame) error {
    sc.serveG.Check()
    id := f.Header().StreamID
    ...

    // Create a new stream
    st = &stream{
        sc:    sc,
        id:    id,
        state: stateOpen,
        isw:   sc.srv.initialStreamRecvWindowSize(sc.rule),
    }
    ...

    // Create the Reqeust and ResponseWriter
    rw, req, err := sc.newWriterAndRequest(st, f)
    if err != nil {
        return err
    }
    st.body = req.Body.(*RequestBody).pipe // may be nil
    st.declBodyBytes = req.ContentLength
    ...

    // Process the request in a new goroutine
    handler := sc.handler.ServeHTTP
    go sc.runHandler(rw, req, handler)
    return nil
}
```

开发篇

如何开发BFE扩展模块

如何开发BFE扩展模块

概述

在为BFE编写一个新的模块时，需要考虑以下方面：

- 配置的加载
- 回调函数的编写
- 模块状态的展示

在下面的讲述中，将结合mod_block的实现作为例子。

mod_block的代码位于/bfe_modules/mod_block

配置的加载

配置的分类

对于一个模块，包括两种配置：

- 静态加载的配置：在BFE程序启动的时候加载
    - 对每个模块有一个这样的配置文件
    - 配置文件的名字和模块名字一致，并以.conf结尾
    - 如：mod_block.conf
- 可动态加载的配置：在BFE程序启动的时候加载，在BFE运行过程中也可以动态加载
    - 对每个模块可以有一个或多个这样的配置文件
    - 配置文件的名字一般以.data结尾
    - 如：在mod_block下有 block_rules.data 和 ip_blocklist.data

对于每一个配置文件，应编写独立的加载逻辑。

配置文件的放置

- 模块的配置文件，应该统一放置于/conf目录下为每个模块独立建立的目录中
- 如：mod_block的配置文件，都放置在/conf/mod_block中

配置加载的检查

无论对于静态加载的配置，还是对于可动态加载的配置，为了保证程序正常的运行，在配置加载的时候，都需要对于配置文件的正确性进行检查。

- 在BFE程序启动阶段，如果模块的配置文件加载失败，则BFE无法启动
- 在BFE程序启动后，如果模块的可动态加载配置文件加载失败，BFE仍然会继续运行

配置的动态加载

对于可动态加载的配置，需要在BFE用于监控和加载的专用web server上做回调注册。这样，通过访问BFE对外监控端口的特定URL，就可以触发某个配置的动态加载。

例如，在mod_block的初始化函数中，可以看到类似下面的逻辑，就是在注册配置加载的回调(详见mod_block.go):

```
    // register web handler for reload
    err = whs.RegisterHandler(web_monitor.WebHandleReload, m.name, m.loadConfData)
    if err != nil {
        ...
    }
```

回调函数的编写和注册

回调函数的编写

根据模块的功能需求，选择合适的回调点，对应编写回调函数。

注意，对于不同的回调点，回调函数的形式可能不同。BFE所提供的回调点和回调函数的形式，可参考BFE的回调机制

例如，在mod_block中，编写了以下两个回调函数(详见mod_block.go)：

```
func (m *ModuleBlock) globalBlockHandler(session *bfe_basic.Session) int {
    ...
}

func (m *ModuleBlock) productBlockHandler(request *bfe_basic.Request) (int, *bfe_http.Response) {
    ...
}
```

回调函数的注册

为了让回调函数生效，需要在模块初始化的时候对回调函数进行注册。

例如，在mod_block中，回调函数的注册逻辑如下(详见mod_block.go)：

```
func (m *ModuleBlock) Init(cbs *bfe_module.BfeCallbacks, whs *web_monitor.WebHandlers, cr string) error {
    ...
    // register handler
    err = cbs.AddFilter(bfe_module.HandleAccept, m.globalBlockHandler)
    if err != nil {
        ...
    }

    err = cbs.AddFilter(bfe_module.HandleFoundProduct, m.productBlockHandler)
    if err != nil {
        ...
    }
    ...
}
```

模块状态的展示

对于每个BFE的模块，强烈推荐通过BFE规定的机制，向外暴露足够的内部状态信息。

在模块对外暴露内部状态，需要做以下3步：

- 定义状态变量
- 注册显示内部状态的回调函数
- 在代码中插入状态设置逻辑

定义状态变量

需要首先设计在模块中需要统计哪些值，并通过结构体成员变量的方式定义出来。

如，在mod_block中，有如下定义(详见mod_block.go)：

```
type ModuleBlockState struct {
    ConnTotal    *metrics.Counter // all connnetion checked
    ConnAccept   *metrics.Counter // connection passed
    ConnRefuse   *metrics.Counter // connection refused
    ReqTotal     *metrics.Counter // all request in
    ReqToCheck   *metrics.Counter // request to check
    ReqAccept    *metrics.Counter // request accepted
    ReqRefuse    *metrics.Counter // request refused
    WrongCommand *metrics.Counter // request with condition satisfied, but wrong command
}
```

然后，要在ModuleBlock中定义一个类型为ModuleBlockState的成员变量，还需要定义一个Metrics类型的成员变量，用于相关的计算。

```
type ModuleBlock struct {
    ...
    state   ModuleBlockState // module state
    metrics metrics.Metrics
    ...
```

然后，需要在构造函数中做初始化的操作

```
func NewModuleBlock() *ModuleBlock {
    m := new(ModuleBlock)
    m.name = ModBlock
    m.metrics.Init(&m.state, ModBlock, 0)
    ...
}
```

注册显示内部状态的回调函数

为了可以通过BFE的监控端口查看模块的内部状态，需要首先实现回调函数。

如，在mod_block中，有如下逻辑(详见mod_block.go)，其中monitorHandlers()是回调函数：

```
func (m *ModuleBlock) getState(params map[string][]string) ([]byte, error) {
    s := m.metrics.GetAll()
    return s.Format(params)
}

func (m *ModuleBlock) getStateDiff(params map[string][]string) ([]byte, error) {
    s := m.metrics.GetDiff()
    return s.Format(params)
}
```

```
func (m *ModuleBlock) monitorHandlers() map[string]interface{} {
    handlers := map[string]interface{}{
        m.name:            m.getState,
        m.name + ".diff": m.getStateDiff,
    }
    return handlers
}
```

然后，在模块的初始化时，需要注册这个回调函数

```
    // register web handler for monitor
    err = web_monitor.RegisterHandlers(whs, web_monitor.WebHandleMonitor, m.monitorHandlers())
    if err != nil {
        ...
    }
```

在代码中插入统计逻辑

在模块的执行逻辑中，可以插入一些统计的代码。

如，在mod_block中，可以看到如下代码(详见mod_block.go)：

```
func (m *ModuleBlock) globalBlockHandler(session *bfe_basic.Session) int {
    ...
    m.state.ConnTotal.Inc(1)
    ...
}
```

附录篇

BFE的多进程GC机制

BFE的多进程GC机制

1. 背景

在2014年初启动了基于Go语言重构BFE转发引擎的工作。当时Go版本为1.3，GC（Garbage Collection, 垃圾回收）延迟的问题非常严重，BFE的实测效果，在100万并发连接的情况下，GC延迟达到了400ms，完全无法满足转发服务的延迟要求。为此，当时在BFE中引入了“多进程轮转”的机制，以降低GC延迟对于转发流量的影响。GC延迟的问题在2017年初发布的Go1.8中有了较好的解决，大部分的GC延迟都降低在1ms内，可以满足业务的要求，于是在2017年从BFE中去掉了多进程轮转机制。

虽然目前这个方案已经废弃，但是其中的一些设计具有一定的通用性，可能未来在类似的场景下可以借鉴使用。

2. 模型

2.1 多进程轮转

让我们先回顾一个经典的高性能服务器设计模式：Prefork模式。该模式下，程序启动后先创建socket，然后fork出多个子进程。根据linux的进程模型，fork后子进程直接继承了父进程创建的socket对象。如伪代码1所示，父进程执行listen操作，子进程执行accept操作。父进程一般完成一些管理功能，比如重启子进程等；子进程完成外部请求的处理，多个子进程能力是等价的。

从内核3.9版本开始，Linux支持一种叫reuseport的机制，可以达到相同的效果。

```
fd = socket()                   // 创建socket
bind(fd, IP_ADDRESS)            // 绑定地址
...
listen(fd, back)                // 开始监听
...
pid = fork()                    // 创建子进程
if (pid == 0) {
    // 子进程代码
    newFd = accept(fd)
    process(newFd)              // 处理请求逻辑
    ...
} else {
    // 父进程代码
    ...
}
```

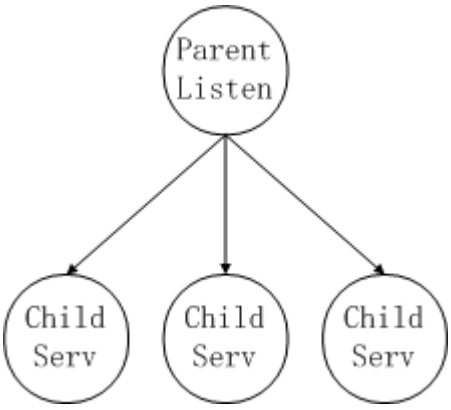


图2-1 Prefork模式

针对Go语言版本BFE所存在的GC延迟问题，初期我们曾经尝试使用以上的多进程机制来降低GC延迟的影响。在使用多个子进程同时服务的情况下，每个子进程承担的请求数量就会变小，进而可以缓解GC时带来的延迟。

以上的机制虽然可以降低GC延迟的绝对值，但是实测下来GC延迟仍然有几十甚至上百毫秒，对于转发业务还是无法接受。

为了解决上述问题，在多进程模式的基础上，增加了“多进程轮转”的机制。”多进程轮转”的基本想法是：控制Prefork模式中多个子进程，让它们轮流处于”服务”或“GC”状态。当进程处于“服务”状态时，可以接受新到来的连接，并处理请求，为了避免GC延迟的影响会关闭GC；当进程处于“GC”的状态时，停止接受新的连接，开启GC以释放不使用的内存。

“多进程轮转”机制的本质是通过增加内存的消耗来换取延迟的降低。由于子进程GC处理的延后，子进程会消耗更多的内存，具体的消耗量取决于停止GC持续的时长、及在这段时间内服务流量的吞吐量。在使用“多进程轮转”机制时，需要对内存的使用量有充分的预估。

“多进程轮转”机制所基于的一个重要假设是：当时BFE所服务的流量主要为HTTP请求，多为短连接，在一个TCP连接内发送的请求大多为2-3个。对于大多数连接来说，通过选择合适的状态切换时间参数，大部分请求的处理会在“服务”状态内完成，只有少量的请求会落在“GC”状态。在“服务”状态和“GC”状态之间还增加了“等待”状态，在这个状态中不接受新的连接，也会关闭GC，以降低GC延迟对于已建立连接中HTTP请求的影响。

2.2 子进程状态定义

在“多进程轮转”模式下，BFE的每个子进程都有4个状态（如图2-2）：

- Init（初始）
    - 子进程初始化状态
    - 完成初始化后，进入Serve状态
- Serve（服务）
    - 子进程接受新的连接（执行accept操作），处理请求
    - 子进程主动关闭GC
- Wait（等待）
    - 子进程不再接受新的连接（不执行accept操作），仅处理已存长连接中新到达的请求
    - 在这个状态，仍然关闭GC
- Gc（回收）
    - 子进程不接受新的连接，但仍会处理已存长连接中新到达的请求
- 在这个状态，会让系统执行GC

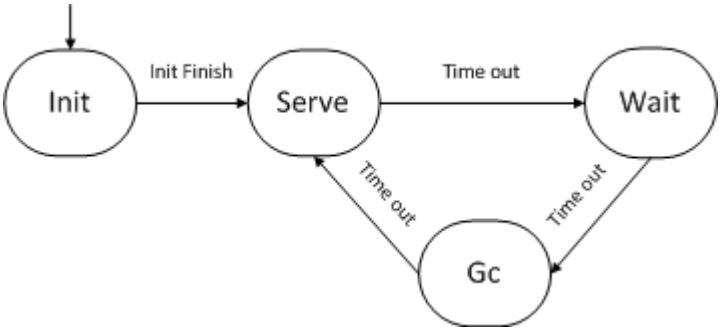


图2-2 子进程的状态

在Server、Wait、GC这3个状态之间，使用超时作为状态切换的触发机制。

当子进程处在Serve和Wait状态时，子进程所服务的连接中的请求不会受到GC延迟的影响；当子进程处于GC状态时，如果这时所服务的连接中有请求在处理，则请求有可能受到GC延迟的影响（取决于请求到达时间和内存回收执行时间的重合关系）。

3. 相关参数的确定

将“多进程轮转”机制使用到实际程序中，还有细节问题需要解决，包括：

- 3个状态间切换时间参数的选择
- 子进程数量的确定
- 内存的需求量

下面对这些问题进行讨论。

3.1 切换时间参数的选择

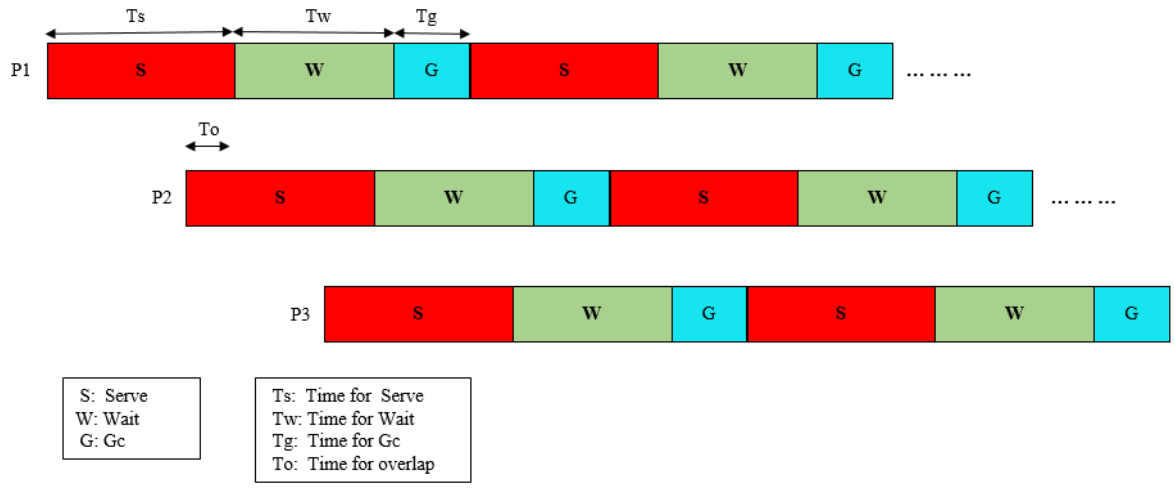


图3-1 BFE多进程切换的时间参数定义

图3-1展示了多个子进程并存场景下状态切换的时序关系。其中展示了子进程状态切换的3个时间参数：

- Ts：Serve状态的持续时间
- Tw：Wait状态的持续时间
- Tg：GC状态的持续时间

对于多个子进程来说，在任何时刻都至少要有一个子进程处于Server状态。考虑到控制进程状态切换的时间误差问题，在相邻的2个子进程间，要求它们在Serve状态有一定的重叠（overlap），以保证不会出现Server状态出现空缺。引入一个参数To用于表示两个子进程在Serve状态重叠（overlap）的时间。

下面我们来讨论这几个时间参数如何选择。

- Ts的选择：

  Ts的取值越大，子进程在GC前所消耗的内存就越多；Ts的取值越小，所需要的子进程数就越多。在实践中，一般将Ts设为5秒。

- Tw的选择：

  Tw的取值应尽量大，以保证在这个时间区间内对于大部分连接都可以完成其中HTTP请求的处理；但同时要考虑到，Tw的取值越大，子进程在GC前所消耗的内存就越多。在实践中，一般将Tw设为30秒。

- Tg的选择：

  GC的处理需要消耗一定的时间。Tg应该足够大，以保证GC的处理可以在下一个Serve状态到来前完成；但是如果Tg太大，又会导致系统所需要进程数的增加。在实践中，一般将Tg设置为2-3秒。

- To的选择：

  To只要能够保证多个子进程间不会出现“服务的空档”即可。在实践中，一般将To设置为1秒。

3.2 子进程数的计算

根据给定的时间参数，可以计算出所需要的子进程数量。

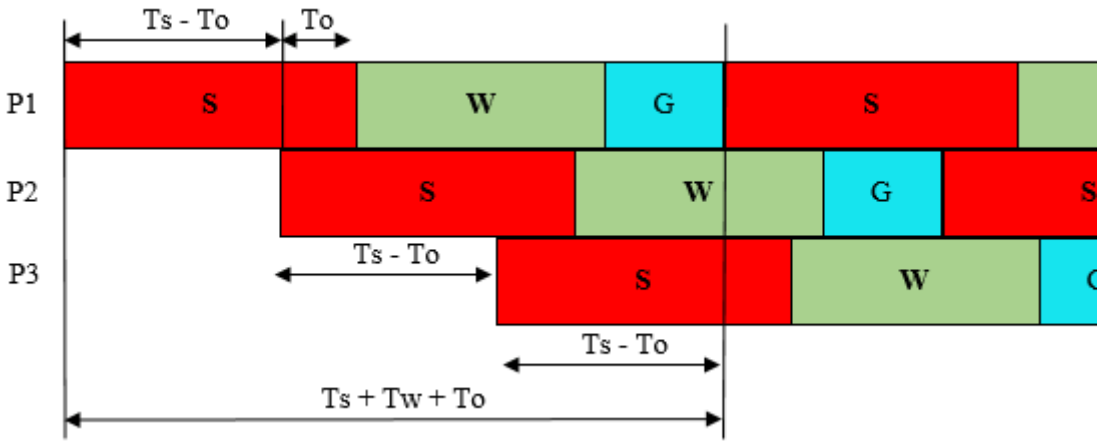


图3-2 BFE子进程数的计算

定义所需要的BFE子进程数为N，N的计算公式为：

$$N = 1 + \left\lceil \frac{Tw + Tg + To}{Ts - To} \right\rceil$$

- 其中，对于 (Tw+Tg+To)/(Ts-To) 要向上取整

- 解释：
  - 考虑一个BFE进程的服务周期（如附图3-2所示，包括Serve、Wait和GC这3个状态）
  - 需要其它子进程在To+Tw+Tg的时间内提供服务
  - 由于有To的存在，每个子进程可以覆盖的时间为（Ts - To）
- 举例：
  - Ts = 5秒，Tw = 20秒，Tg = 3秒，To = 1秒
  - N = 1 + (20 + 3 + 1) / (5 - 1) = 7

3.3 内存消耗的计算

在“多进程轮转”机制下，由于在Serve和Wait状态主动关闭GC，需要消耗大量的内存。这里对“多进程轮转”机制下BFE的内存消耗量做一个估算。

- 定义内存的最大消耗量为M
  - M = (Ts + Tw + Tg) * 内存消耗速度
  - 其中内存消耗速度只能依靠线下压力测试或线上实测数据来获得
- 举例：
  - Ts = 5秒，Tw = 20秒，Tg = 3秒
  - 假设经观测获得内存消耗速度为每分钟20GB
  - M = (5 + 20 + 3) * 20 / 60 = 9.4GB