透明多级分流系统

- **前言**
  - **不同部件价值**
    - 一些位于客户端或网络边缘的部件，能够迅速响应用户的请求，避免给后方I/O与CPU带来压力，如本地缓存、内容分发网络CDN、反向代理等。
    - 一些部件的处理能力能够线性拓展、易于伸缩，可以使用较小的代价堆叠机器来获得与用户数量相匹配的并发能力，应尽量作为业务逻辑的主要载体，如集群中能够自动扩缩的服务节点。
    - 一些部件稳定服务对系统运行有全局性的影响，要时刻保持容错备份，维护高可用性，如服务注册中心、配置中心。
    - 一些部件是天生的单点部件，只能依靠升级机器本身的网络、存储和运算性能来提升处理能力，如位于系统入口的路由、网关或负载均衡器（它们可以做集群，但一次请求中无可避免有一个是单点的部件）、位于请求调用链末端的传统关系数据库。
  - **设计原则**
    - **尽可能减少单点部件。若单点无可避免，则尽量减少到达单点部件的流量**。适当引导请求分流到最合适的组件中，避免绝大多数流量汇聚到单点部件如数据库中，同时依然能够在绝大多数时候保证处理结果的准确性，使单点系统在出现故障时可以自动且迅速实施补救措施，即**系统多级分流的意义所在**。
    - **奥卡姆剃刀原则，在能够满足需求的前提下，最简单的系统就是最好的系统。**
- **客户端缓存**
  - **状态缓存：**不经过服务器，客户端直接根据缓存信息对目标网站的状态判断，对应响应码301Moved Permanently永久重定向。如资源已经永久移到新地址，缓存信息在第二次访问时会根据缓存自动向新地址发出请求。
  - **强制缓存**
    - **定义：**假设在某个时间点来临前，资源的内容和状态一定不会改变，因此客户端可以无须经过任何请求，在时间点到来前一直持有并使用该资源的本地副本。
    - **生效场景：**在浏览器的地址输入、页面链接跳转、新开窗口、前进后退中均可生效，但用户主动刷新页面时应该自动失效。
    - **Expires**
      - **定义：**当服务器返回某个资源带有此Header时，则意味着服务器保证资源在截止时间之前不会发生改变。浏览器可以直接缓存该数据，不再重新发出请求。从HTTP 1.0开始提供。
      - HTTP/1.1 200 OK
        Expires: Wed, 8 Apr 2020 07:28:00 GMT
      - **缺陷：**受限于客户端本地时间；无法处理设计用户身份的私有资源；无法描述不缓存的语义。
    - **Cache-Control**
      - **定义：**与Expires类似，从HTTP 1.1开始提供。与Expires相比，可同时存在于请求和响应Header中，与Expires同时存在时以Cache-Control为准。
      - HTTP/1.1 200 OK
        Cache-Control: max-age=600
      - **参数max-age和s-maxage：**表示资源缓存有效期的相对时间，单位为秒。s-maxage为共享缓存的有效时间。
      - **参数public和private：**public表示可以被CDN、代理等缓存的资源，private表示只允许被用户的客户端缓存的私有资源。
      - **参数no-cache和no-store：**no-cache表示不使用缓存过期的数据，即每次请求都判断资源是否过期，从服务器获取资源还是从客户端缓存获取。no-store表示彻底不缓存资源，每次请求都从服务器获取资源。
      - **参数no-transform：**禁止以任何形式修改资源，例如禁止CDN、透明代理压缩资源。
      - **参数min-fresh和only-if-cached：**仅用于客户端请求Header，min-fresh用于建议服务器返回一个不少于该时间的缓存资源，即包含在max-age内，但不少于min-fresh。only-if-cached表示客户端仅使用事先缓存的资源进行响应，若缓存不命中则返回503/Service Unavaliable。
      - **参数must-revalidate和proyx-revalidate：**must-revalidate表明资源过期后，一定要从服务器进行获取，即超过max-age后等同于no-cache。proxy-revalidate与前者类似，用于提示代理和CDN等设备。
  - **协商缓存**
    - **定义：**不同于强制缓存基于时效性，协商缓存基于变化检测，在一致性上有更好表现，但需要多一次变化检测的交互开销，性能稍差一些。
    - **生效场景：**在浏览器的地址输入、页面链接跳转、新开窗口、前进后退中均可生效，**并且用户主动刷新页面时同样生效**，只有用户强制刷新时才会失效。
    - **与强制缓存并存：**当强制缓存存在时，直接从强制缓存返回资源，无需进行变化检测；当强制缓存超时失效时，或者使用no-cache和must-revalidate时，协商缓存仍可以正常工作。
    - **Last-Modified和If-Modified-Since**
      - **定义：**Last-Modified用于响应Header，表明资源的最后修改时间。客户端对该资源的再次请求时，会使用If-Modified-Since加上这个最后修改时间发送回服务器。若服务器发现资源在该时间后没有修改，则直接返回304/Not Modified；若修改则返回完整资源信息和新的最后修改时间。
      - HTTP/1.1 304 Not Modified
        Cache-Control：public，max-age=600
        Last-Modified：Wed, 8 Apr 2020 15:31:30 GMT
      - HTTP/1.1 200 OK
        Cache-Control：public，max-age=600

Last-Modified：Wed, 8 Apr 2020 15:31:30 GMT
Content

- **ETag和 If-None-Match**
  - **定义：**ETag是服务端的响应Header，用于告诉客户端资源的唯一标识，由服务端来实现标识的生成。对于该资源，客户端进行再次请求时会一并发送此标识，检测资源是否发生了变化。
  - HTTP/1.1 304 Not-Modified
    Cache-Control：public，max-age=600
    Etag："asdiw-jias-asjidwn"
  - HTTP/1.1 200 OK
    Cache-Control：public，max-age=600
    Etag："asdiw-jias-asjidwn"
    Content
  - **优势：**Etag是HTTP中一致性最强的缓存机制，Last-Modified只能精确到秒级，若某个文件在一秒内被修改多次或者某些文件定期生成但内容并未发生变化，在这些情况下Last-Modified没有太有效的作用。
  - **缺陷：**Etag是HTTP中性能最差的缓存机制，每次请求时，服务端都要进行资源的Etag计算。Etag与Last-Modified可以共存，优先验证ETag。

域名解析

- **DNS解析步骤**
  - 1、客户端先检查本地的 DNS 缓存，查看是否存在可用的该域名地址记录。DNS 以存活时间TTL来衡量缓存有效情况，并依靠 TTL 超期后重新获取来保证一致性。
  - 2、客户端将地址发送给本机操作系统中配置的本地 DNS
  - 3、本地 DNS 收到查询请求后，按照
    “是否有www.icyfenix.com.cn的权威服务器”→
    “是否有icyfenix.com.cn的权威服务器”→
    “是否有com.cn的权威服务器”→
    “是否有cn的权威服务器”的顺序，依次查询自己的地址记录，如果都没有查询到，就会一直找到最后点号代表的根域名服务器为止。
  - 4、假设本地 DNS 是全新的，不存在任何域名的权威服务器记录，当 DNS 查询请求按顺序一直查到根域名服务器之后，将会得到“cn的权威服务器”的地址记录，逐级获取下一级的地址记录，最后找到能够解释www.icyfenix.com.cn的权威服务器地址。
  - 5、通过“www.icyfenix.com.cn的权威服务器”，查询www.icyfenix.com.cn的地址记录，地址记录并不一定就是指 IP 地址。
- **DNS服务器概念**
  - **权威域名服务器（Authoritative DNS）：**是指负责翻译特定域名的 DNS 服务器，“权威”意味着这个域名应该翻译出怎样的结果是由它来决定的。
  - **根域名服务器（Root DNS）**是指固定的、无需查询的顶级域名服务器，可以默认已内置在操作系统代码之中。全世界一共有 13 组根域名服务器。 DNS 主要采用 UDP 传输协议来进行数据交换，未分片的 UDP 数据包在 IPv4 下最大有效值为 512 字节，最多可以存放 13 组地址记录。
- **优势：**每种记录类型中可以包括多条记录，一个域名下可以配置多条不同的 A 记录为例，权威服务器可以根据策略来进行选择，如**智能线路**，根据访问者所处的不同地区、不同服务商等因素来确定返回最合适的 A 记录，达到智能加速的目的。
- **缺陷：**DNS系统多级分流的设计使得DNS能够经受住全球网络流量不间断的冲击，而典型的问题是**响应速度**，当极端情况（各级服务器均无缓存）下的域名解析可能导致每个域名都必须递归多次才能查询到结果，影响传输的响应速度。常用**DNS预取**的前端优化方式来避免此问题。
- **安全风险：**DNS 的分级查询意味着每一级都有可能受到中间人攻击的威胁，产生被劫持的风险。要攻陷位于顶层的服务器和链路非常困难，但很多位于底层或者来自本地运营商的 Local DNS 服务器的安全防护则相对松懈，甚至不少地区的运营商自己就会主动进行劫持。
- **DNS新方式HTTPDNS：**将原本的 DNS 解析服务开放为一个基于 HTTPS 协议的查询服务，替代基于 UDP 传输协议的 DNS 域名解析，通过程序代替操作系统直接从权威 DNS 或者可靠的 Local DNS 获取解析数据，从而绕过传统 Local DNS。

传输链路

- **背景：**经过客户端缓存的节流、经过 DNS 服务的解析指引，程序发出的请求流量便正式离开客户端，经过传输链路到达服务端。而**程序发出的请求能否与应用层、传输层协议提倡的方式相匹配，对传输的效率也会有极大影响**。
- **连接数优化**
  - **HTTP和TCP设计的”不协调“：** HTTP/3 以前是以 TCP 为传输层的应用层协议，而**HTTP 传输对象的主要特征是数量多、时间短、资源小、切换快，而TCP 协议本身是面向于长时间、大数据传输来设计的**（慢启动和三次握手），在长时间尺度下，它连接建立的高昂成本才不至于成为瓶颈。
  - **解决方案：**减少发出的请求数量，同时增加客户端到服务端的连接数量。除了使用前端的一些优化技巧，更重要的是从协议层面去解决连接成本过高的问题。
  - **Keep-Alive**
    - **定义：**HTTP1.1时默认打开的连接复用技术，原理是让客户端对同一个域名长期持有一个或多个不会用完即断的 TCP 连接。典型做法是在客户端维护一个 FIFO 队列，每次取完数据之后一段时间内不自动断开连接，以便获取下一个资源时直接复用，避免创建 TCP 连接的成本。
    - **缺陷：**常见队首阻塞问题，若队列中有10个资源请求和1个TCP连接，即使它们可以并发请求，但是缺乏对返回的资源包归属的判定，只能按照顺序逐渐返回资源的数据包，若第一个资源请求操作未收到所有资源数据包，其他资源请求只能等待。
  - **HTTP管道：**HTTP 服务器中也建立类似客户端的 FIFO 队列，让客户端一次将所有要请求的资源名单全部发给服务端，由服务端来安排返回顺序，管理传输队列。虽然无法完全避免队首阻塞的问题，但服务端能够较为准确地评估资源消耗情况，进而能够更紧凑地安排资源传输。由于 HTTP 管道需要多方共同支持，协调起来相当复杂，推广得并不算成功。
  - **HTTP/2多路复用**
    - **定义：**在 HTTP/1.x 中，HTTP 请求就是传输过程中最小粒度的信息单位，在 HTTP/2 中，帧才是最小粒度的信息单位。每个帧都附带一个流 ID 以标识这个帧属于哪个流。在同一个 TCP 连接中传输的多个数据帧就可以根据流 ID 轻易区分出开来，在客户端毫不费力地将不同流中的数据重组出不同 HTTP 请求和响应报文来
    - **优势：**有了多路复用的支持，HTTP/2 就可以对每个域名只维持一个 TCP 连接来以任意顺序传输任意数量的资源，既减轻了服务器的连

接压力，开发者也不用去考虑域名分片等突破浏览器对每个域名的连接数限制了；没有了 TCP 连接数的压力，就无须刻意压缩 HTTP 请求了，在有些情况下压缩请求、节省Header并不奏效。

- **传输压缩**
  - **背景：**HTTP 支持GZip压缩，由于 HTTP 传输的主要内容是文本数据，压缩的收益较高，传输数据量一般会降至20%左右。而对于不适合压缩的资源，Web 服务器则能根据 MIME 类型来判断是否对响应进行压缩，已经采用过压缩算法存储的资源，便不会被二次压缩，空耗性能。早期，采用**“静态预压缩”**，静态资源先预先压缩存放起来，而现在由服务器对符合条件的请求将在输出时进行**“即时压缩”**。
  - **压缩与持久连接的冲突**
    - **冲突：**使用即时压缩后，服务器再也没有办法给出Content-Length 这个响应 Header 了；使用持久连接后，不再依靠 TCP 连接是否关闭来判断资源请求是否结束。因此，HTTP/1.0的话，由于缺乏判断资源请求是否结束的标志，持久链接和即时压缩只能二选其一。
    - **解决方式：**HTTP/1.1增加了“**分块传输编码**”的资源结束判断机制，在响应 Header 中加入“Transfer-Encoding: chunked”之后，报文中的 Body 改为用一系列分块来传输。每个分块包含十六进制的长度值和对应长度的数据内容，长度值独占一行，数据从下一行开始。最后以一个长度值为 0 的分块来表示资源结束。
- **快速UDP网络连接**
  - **背景：**从职责上讲，持久连接、多路复用、分块编码这些能力，已经或多或少超过了应用层的范畴。要从根本上改进 HTTP，必须直接替换掉 HTTP over TCP 的根基，即 TCP 传输协议，这便最新一代 HTTP/3 协议的设计重点。
  - **优势**
    - QUIC 会以 UDP 协议为基础，没有丢包自动重传的特性，可靠传输能力完全由自己来实现，**可以对每个流能做单独控制，即使某个流发生错误，仍然可以独立地继续为其他流提供服务。**TCP 协议接到数据包丢失或损坏通知之前，可能已经收到了大量数据，在纠正错误之前，其他的正常请求都会等待甚至被重发，这也是HTTP/2 未能解决传输大文件慢的根本原因。
    - QUIC 在移动设备上的优势体现在网络切换时的响应速度上，譬如当移动设备在不同IP之间切换，若使用 TCP 协议，现存所有连接都必定会超时、中断，然后重新创建。**QUIC 提出了连接标识符的概念，唯一标识客户端与服务器之间的连接，无须依靠 IP 地址，只需向服务端发送包含标识符的数据包即可重用既有连接**

内容分发网络CDN

- **定义：**客户端缓存、域名解析、传输链路的综合应用，内容分发网络的工作过程，主要涉及路由解析、内容分发、负载均衡和所能支持的 CDN 应用内容四个方面。
- **互联网系统的速度决定**
  因素（网络传输角度）
  - **网站服务器**接入网络运营商的链路所能提供的**出口带宽**。
  - **用户客户端**接入网络运营商的链路所能提供的**入口带宽**。
  - 从网站到用户之间经过的**不同运营商之间互联节点的带宽**，一般来说两个运营商之间只有固定的若干个点是互通的，所有跨运营商之间的交互都要经过这些点。
  - 从网站到用户之间的**物理链路传输时延ping**
  - 除了用户客户端入口带宽只能通过换更好的宽带才能解决之外，其余三个都能通过内容分发网络来显著改善。一个运作良好的内容分发网络，**能为互联网系统解决跨运营商、跨地域物理距离所导致的时延问题，能为网站流量带宽起到分流、减负的作用。**
- **路由解析**
  - **方式：**CDN将用户请求路由到特定资源服务器上是依靠DNS服务器来实现的。
  - **工作过程**
    - 1、架设好“icyfenix.cn”的服务器后，将IP地址在CDN服务商上注册为源站，注册后会得到一个 CNAME，即“icyfenix.cn.cdn.dnsv1.com.”。
    - 2、将得到的CNAME在购买域名的DNS服务商上注册为一条CNAME 记录。
    - 3、当第一位用户来访站点时，首先发生一次未命中缓存的 DNS 查询，域名服务商解析出 CNAME 后，返回给本地 DNS，至此之后链路解析的主导权就开始由内容分发网络的调度服务接管了。
    - 4、本地 DNS 查询 CNAME 时，由于能解析该 CNAME 的权威服务器只有 CDN 服务商所架设的权威DNS，因此DNS服务将根据特定均衡策略和参数，在全国各地能提供服务的 CDN 缓存节点中挑选一个最适合的，将它的 IP 代替源站的 IP 地址，返回给本地 DNS。
    - 5、浏览器从本地 DNS 拿到 IP 地址，将该 IP 当作源站服务器来进行访问，此时该 IP 的 CDN 节点上可能没有缓存过源站的资源，经过内容分发后的 CDN 节点，就有能力代替源站向用户提供所请求的资源。
- **内容分发**
  - **背景：**CDN的缓存节点接管了用户向服务器发出的资源请求，而缓存节点中必须有用户请求的资源副本，才可能代替源站来响应用户请求。其中，包括了两个子问题，即“如何获取源站资源（内容分发）”和“如何管理/更新资源”。
  - **主流内容分发方式**
    - **主动分发Push：**分发由源站主动发起，将内容从源站或者其他资源库推送到用户边缘的各个 CDN 缓存节点上。可以采用任何传输方式、推送策略、推送时间，只要与更新策略相匹配即可。由于主动分发通常需要源站、CDN 服务双方提供程序 API 接口层面的配合，所以它对源站并不是透明的，只对用户一侧单向透明。主动分发一般用于网站要预载大量资源的场景。
    - **被动回源Pull：**被动回源是由用户访问所触发全自动、双向透明的资源缓存过程。当资源首次被请求时CDN缓存节点发现没有该资源，将实时从源站中获取，响应时间可认为是资源从源站到 CDN 缓存节点+资源从 CDN 发送到用户的时间之和，首次访问通常比较慢的，但并不一定比用户直接访问源站更慢。被动回源不适合应用于数据量较大的资源。优点是不需要源站在程序上做任何的配合，使用方便，是小型站点使用 CDN 服务的主流选择。
  - **管理/更新资源**
    - **背景：**HTTP 协议中关于缓存的 Header 定义中有对 CDN 这类共享缓存的一些指引性参数，如Cache-Control的 s-maxage，但是否要遵循，完全取决于 CDN 本身的实现策略。如果CDN完全照着 HTTP Headers 来控制缓存失效和更新，效果反而会相当的差，因此，CDN 缓存的管理就不存在通用的准则
    - **做法：**最常见的做法是超时被动失效与手工主动失效相结合。超时失效是指给予缓存资源一定的生存期，超过了生存期就在下次请求时重新被动回源一次。而手工失效是指 CDN 服务商一般会提供给程序调用来失效缓存的接口，在网站更新时，由持续集成的流水线自动调用该接口来实现缓存更新。
- **CDN应用**

- **加速静态资源**
- **安全防御：**CDN 在广义上可以视作网站的堡垒机，源站只对 CDN 提供服务，由 CDN 来对外界其他用户服务，这样恶意攻击者就不容易直接威胁源站。
- **协议升级：**CDN可以实现源站是 HTTP 协议的，对外开放的网站是基于 HTTPS。
  可以实现源站到 CDN 是 HTTP/1.x 协议，CDN 提供的外部服务是 HTTP/2 3；
  可以实现源站是基于 IPv4 网络的，CDN 提供的外部服务支持 IPv6 网络。
- **修改资源：**CDN 可以在返回资源给用户的时候修改任何内容，实现不同的目的。
- **访问控制：**CDN 可以实现 IP 黑/白名单功能，根据不同的来访 IP 提供不同的响应结果、IP的访问流量来实现 QoS 控制、根据 HTTP 的 Referer 来实现防盗链等。

负载均衡

- **背景：**除了DNS层面的负载均衡，还有网络请求进入数据中心入口之后的其他级次的负载均衡。
- **基础概念**
  - **四层负载均衡：**四层的意思是说这些工作模式的共同特点是维持着同一个 TCP 连接，不是说只工作在第四层。这些模式主要都是工作在二层（数据链路层，改写 MAC 地址）和三层（网络层，改写 IP 地址）上。
  - **七层负载均衡：**流量已经到达目标主机上，谈不上流量转发，只能做代理。
  - 四层负载均衡的优势是性能高，七层负载均衡的优势是功能强；做多级混合负载均衡，通常应是低层的负载均衡在前，高层的负载均衡在后。
- **数据链路层负载均衡**
  - **工作原理：**修改请求的数据帧中的 MAC 目标地址，让原本发送给负载均衡器的请求的数据帧，被二层交换机根据新的 MAC 目标地址转发到服务器集群中对应的服务器的网卡上，即真实服务器获得了一个原本目标并不是发送给它的数据帧。
  - **限制：**由于第三层的IP数据包中包含了源和目标的 IP 地址，只有真实服务器IP 地址与数据包中的目标 IP 地址一致，数据包才能被正确处理。因此，需要把真实物理服务器集群所有机器的虚拟 IP 地址配置成与负载均衡器的虚拟 IP 一样，这样经均衡器转发后的数据包就能在真实服务器中顺利地使用。
  - **优势：**响应结果不再需要通过负载均衡服务器进行地址交换，可将响应结果的数据包直接从真实服务器返回给用户的客户端，避免负载均衡器网卡带宽成为瓶颈，因此数据链路层的负载均衡效率是相当高的。
  - **缺陷：**二层负载均衡器直接改写目标 MAC 地址的工作原理决定了它与真实的服务器的通信必须是二层可达的，通俗地说就是必须位于同一个子网当中，无法跨 VLAN。
  - **适用场景：**优势效率高和劣势不能跨子网共同决定了数据链路层负载均衡最适合用来做数据中心的第一级均衡设备，用来连接其他的下级负载均衡器。
- **网络层负载均衡**
  - **工作原理：**与数据链路层负载均衡原理类似，通过改变IP数据包的 IP 地址来实现数据包的转发。
  - **修改方式**
    - **IP隧道：**保持原来的数据包不变，新创建一个数据包，把原来数据包的 Headers 和 Payload 整体作为另一个新的数据包的 Payload，在这个新数据包的 Headers 中写入真实服务器的 IP 作为目标地址，然后把它发送出去。
    - **IP隧道优势：**由于没有修改原有数据包中的任何信息，I 隧道的转发模式仍然具备三角传输的特性，即负载均衡器转发来的请求，可以由真实服务器去直接应答。且由于 IP 隧道工作在网络层，所以可以跨越 VLAN，因此摆脱了直接路由模式中网络侧的约束。
    - **IP隧道缺陷：**要求真实服务器必须支持IP 隧道协议，所幸几乎所有的 Linux 系统都支持 IP 隧道协议；这种模式仍必须通过专门的配置，必须保证所有的真实服务器与均衡器有着相同的虚拟 IP 地址，因为回复该数据包时，需要使用这个虚拟 IP 作为响应数据包的源地址，这样客户端收到这个数据包时才能正确解析。
    - **改变目标数据包：**把数据包 Headers 中的目标地址改掉，通过三层交换机转发给真实服务器。但是如果真实服务器直接将应答包返回客户端的话，应答数据包的源 IP 是客户端不认识的真实服务器的 IP，无法正常处理。因此，只能让应答流量继续回到负载均衡，由负载均衡把应答包的源 IP 改回自己的 IP，再发给客户端。这种通过网络地址转换的负载均衡器成为NAT负载均衡器。
    - **NAT模式优势：**真实服务器不必支持IP隧道，真实服务器与负载均衡器不必有相同虚拟IP。
    - **NAT模式缺陷：**流量压力比较大的时候，NAT 模式的负载均衡会带来较大的性能损失，比起直接路由和 IP 隧道模式，甚至会出现数量级上的下降。
    - **SNAT：**均衡器在转发时，不仅修改目标 IP 地址，连源 IP 地址也一起改了，源地址就改成均衡器自己的 IP。好处是真实服务器无须配置网关就能够让应答流量经过正常的三层路由回到负载均衡器上。缺点是真实服务器处理请求时无法拿到客户端IP 地址，有一些需要根据目标 IP 进行控制的业务逻辑就无法进行。
- **应用层负载均衡**
  - **与四层负载均衡区别：**四层负载均衡时，客户端到响应请求的真实服务器维持着同一条 TCP 通道。工作在四层之后的负载均衡模式无法进行转发，只能进行代理，此时真实服务器、负载均衡器、客户端三者之间由两条独立的 TCP 通道来维持通信。
  - **七层负载均衡优势：**七层负载均衡器属于反向代理中的一种，只论网络性能比不过四层均衡器，它比四层均衡器至少多一轮 TCP 握手，有着跟 NAT 转发模式一样的带宽问题，而且通常要耗费更多的 CPU，因为可用的解析规则远比四层丰富。它工作在应用层，可以感知应用层通信的具体内容，往往能够做出更明智的决策。
  - **可实现功能**
    - 所有 CDN 可以做的缓存方面的工作七层均衡器全都可以实现，譬如静态资源缓存、协议升级、安全防护、访问控制等。
    - 七层均衡器可以实现更智能化的路由。譬如根据 Session 路由，以实现亲和性的集群；根据 URL 路由，实现专职化服务，相当于网关；根据用户身份路由，实现对部分用户的特殊服务，如某些站点的贵宾服务器等。
    - 某些安全攻击可以由七层均衡器来抵御
    - 链路治理措施都需要在七层中进行，譬如服务降级、熔断、异常注入等
- **均衡策略与实现**
  - 轮循均衡（Round Robin）
  - 权重轮循均衡（Weighted Round Robin）
  - 随机均衡（Random）

- 权重随机均衡（Weighted Random）
- 一致性哈希均衡（Consistency Hash）
- 响应速度均衡（Response Time）
- 最少连接数均衡（Least Connection）

- **服务端缓存**
  - **缺陷**
    - **从开发角度来说：**引入缓存会提高系统复杂度，因为你要考虑缓存的失效、更新、一致性等问题。
    - **从运维角度来说：**缓存会掩盖掉一些缺陷，让问题在更久的时间以后，出现在距离发生现场更远的位置上。
    - **从安全角度来说：**缓存可能泄漏某些保密数据，也是容易受到攻击的薄弱点。
  - **引入服务端缓存理由**
    - **为缓解 CPU 压力而做缓存：**譬如把方法运行结果存储起来、把原本要实时计算的内容提前算好、把公用数据进行复用，节省 CPU 算力，顺带提升响应性能。
    - **为缓解 I/O 压力而做缓存：**譬如把原本对网络、磁盘等较慢介质的读写访问变为对内存等较快介质的访问，将原本对单点部件如数据库的读写访问变为到可扩缩部件如缓存中间件的访问，顺带提升响应性能。
    - 缓存是典型以空间换时间来提升性能的手段，但**出发点是缓解 CPU 和 I/O 资源在峰值流量下的压力，“顺带”地提升响应性能** 。如果可以通过增强 CPU、I/O 本身的性能来满足需要的话，那升级硬件往往是更好的解决方案，即使需要一些额外的投入成本，也通常要优于引入缓存后可能带来的风险。
  - **缓存属性**
    - **吞吐量：**缓存的吞吐量使用每秒操作数OPS 值来衡量，反映了对缓存进行并发读、写操作的效率，即缓存本身的工作效率高低。
      - **缓存中最主要数据竞争：**读取数据的同时，也会伴随着对数据状态的写入操作，写入数据的同时，也会伴随着数据状态的读取操作。例如，读取时要同时更新数据的最近访问时间和访问计数器的状态，以实现缓存的淘汰策略；又或者读取时要同时判断数据的超期时间等信息，以实现失效重加载等其他扩展功能。
      - **状态维护方式一：**以 Guava Cache 为代表的同步处理机制，即在访问数据时一并完成缓存淘汰、统计、失效等状态变更操作，通过分段加锁等来尽量减少竞争。
      - **状态维护方式二：**以 Caffeine 为代表的异步日志提交机制，将对数据的读、写过程看作是日志即对数据的操作指令的提交过程。异步提交的日志已经将原本在 Map 内的锁转移到日志的追加写操作上，日志里优化的余地就比在 Map 中要大得多。
    - **命中率：**缓存的命中率即成功从缓存中返回结果次数与总请求次数的比值，命中率越低，引入缓存的收益越小，价值越低。与命中率相关的是缓存的淘汰策略。
      - **FIFO：**优先淘汰最早进入被缓存的数据。越是频繁被用到的数据，可能会越早被存入缓存之中。如果采用这种淘汰策略，很可能会大幅降低缓存的命中率。
      - **LRU：**优先淘汰最久未被使用访问过的数据。LRU 通常会采用如 LinkedHashMap来实现。LRU尤其适合处理短时间内频繁访问的热点对象。但如果热点数据在系统中经常被频繁访问，但最近一段时间因为某种原因未被访问过，此时这些热点数据依然要面临淘汰的命运，LRU 依然可能错误淘汰价值更高的数据。
      - **LFU：**优先淘汰最不经常使用的数据。LFU 会给每个数据添加一个访问计数器，需要淘汰时就清理计数器数值最小的那批数据。LFU可以解决热点数据间隔一段时间不访问就被淘汰的问题，但需要对每个缓存数据维护一个计数器，每次访问都要更新，这样做会带来高昂的维护开销；另一个问题是不便于处理随时间变化的热度变化，譬如某个曾经频繁访问的数据现在不需要了，它也很难自动被清理出缓存。
      - **TinyLFU：**TinyLFU 是 LFU 的改进版本。为了避免访问修改计数器带来的性能负担，采用 Sketch 对访问数据进行分析；采用了基于“滑动时间窗”的热度衰减算法，以此解决“旧热点”数据难以清除的问题。
      - **W-TinyLFU：**TinyLFU 的改进版本。应对短时间的突发访问是 LRU 的强项，它结合了 LRU 和 LFU 两者的优点，将新记录暂时放入一个前端 LRU 缓存里面，让这些对象累积热度，如果能通过 TinyLFU 的过滤器，再进入主缓存中存储。
    - **扩展功能：**缓存除了基本读写功能外，还提供哪些额外的管理功能，譬如最大容量、失效时间、失效事件、命中率统计，等等。
    - **分布式支持：**缓存可分为“进程内缓存”和“分布式缓存”两大类，前者只为节点本身提供服务，无网络访问操作，速度快但缓存的数据不能在各个服务节点中共享，后者则相反。
      - **复制式缓存：**复制式缓存可以看作是“能够支持分布式的进程内缓存”，缓存中所有数据在分布式集群的每个节点里面都存在有一份副本，读取数据时直接从当前节点的进程内存中返回；当数据发生变化时，将变更同步到集群的每个节点中，复制性能随着节点的增加呈现平方级下降，变更数据的代价十分高昂。
      - **集中式缓存：**分布式缓存的主流形式，读写都需要网络访问，但不会随着集群节点数量增加产生额外负担，坏处是不可能达到进程内缓存的高性能。与使用缓存的应用分处在独立的进程空间中，能够为异构语言提供服务。如果要缓存对象等复杂类型的话，基本上就只能靠序列化来支撑具体语言的类型系统，不仅有序列化的成本，还很容易导致传输成本也显著增加。主流的是Redis。
      - 根据分布式缓存集群是否能保证数据一致性，可以将它分为 AP 和 CP 两种类型，Redis是典型的AP缓存集群。
      - **多级缓存：**使用进程内缓存做一级缓存，分布式缓存做二级缓存，如果能在一级缓存中查询到结果就直接返回，否则便到二级缓存中去查询，再将二级缓存中的结果回填到一级缓存，以后再访问该数据就没有网络请求了。如果二级缓存也查询不到，就发起对最终数据源的查询，将结果回填到一、二级缓存中去。
  - **缓存风险**
    - **缓存穿透：**如果查询的数据在数据库中根本不存在的话，缓存里自然也不会有，这类请求的流量每次都不会命中，每次都会触及到末端的数据库，缓存就起不到缓解压力的作用了。
      - 对于业务逻辑本身就不能避免的缓存穿透，可以约定在一定时间内对返回为空的 Key 值依然进行缓存，使得在一段时间内缓存最多被穿透一次。后续业务在数据库中对该 Key 值插入了新记录，那应当在插入之后主动清理掉缓存的 Key 值。
      - 对于恶意攻击导致的缓存穿透，通常会在缓存之前设置一个布隆过滤器来解决。如果布隆过滤器给出的判定结果是请求的数据不存在，那就直接返回即可，连缓存都不必去查。
    - **缓存击穿：**如果缓存中某些热点数据忽然因某种原因失效了，譬如由于超期而失效，此时又有多个针对该数据的请求同时发送过来，这些请求将全部未能命中缓存，都到达真实数据源中去，导致其压力剧增。
      - **加锁同步：**以请求该数据的 Key 值为锁，使得只有第一个请求可以流入到真实的数据源中，其他线程采取阻塞或重试策略。进程内缓存施加普通互斥锁，分布式缓存施加分布式锁，数据源就不会同时收到大量针对同一个数据的请求了。
      - **手动管理：**缓存击穿是仅针对热点数据被自动失效才引发的问题，对于这类数据，可以由开发者通过代码来有计划地完成更新、失效，避

免由缓存的策略自动管理。

- **缓存雪崩：**由于大批不同的数据在短时间内一起失效，导致了这些数据的请求都击穿了缓存到达数据源，同样令数据源在短时间内压力剧增。
  - 提升缓存系统可用性，建设分布式缓存的集群。
  - 启用透明多级缓存，各个服务节点一级缓存中的数据通常会具有不一样的加载时间，也就分散了它们的过期时间。
  - 将缓存的生存期从固定时间改为一个时间段内的随机时间，譬如原本是一个小时过期，缓存不同数据时，设置生存期为 55 分钟到 65 分钟之间的某个随机时间。
- **缓存污染：**指缓存中的数据与真实数据源中的数据不一致的现象。为了尽可能的提高使用缓存时的一致性，已经总结不少更新缓存可以遵循设计模式，譬如 Cache Aside等。
  - 读数据时，先读缓存，没有的话再读数据源，然后将数据放入缓存，再响应请求。
  - 写数据时，先写数据源，然后失效（而不是更新）掉缓存。