你了解 CDN 么？如何使用内容发布网络（CDN)优化网络性能 ？

一、CDN 的基本原理和基础架构

CDN 是将源站内容分发至最接近用户的节点，使用户可就近取得所需内容，提高用户访问的响应速度和成功率。解决因分布、带宽、服务器性能带来的访问延迟问题，适用于站点加速、点播、直播等场景。

最简单的 CDN 网络由一个 DNS 服务器和几台缓存服务器组成：

1. 当用户点击网站页面上的内容 URL，经过本地 DNS 系统解析，DNS 系统会最终将域名的解析权交给 CNAME 指向的 CDN 专用 DNS 服务器。
2. CDN 的 DNS 服务器将 CDN 的全局负载均衡设备 IP 地址返回用户。
3. 用户向 CDN 的全局负载均衡设备发起内容 URL 访问请求。
4. CDN 全局负载均衡设备根据用户 IP 地址，以及用户请求的内容 URL，选择一台用户所属区域的区域负载均衡设备，告诉用户向这台设备发起请求。
5. 区域负载均衡设备会为用户选择一台合适的缓存服务器提供服务，选择的依据包括：根据用户 IP 地址，判断哪一台服务器距用户最近；根据用户所请求的 URL 中携带的内容名称，判断哪一台服务器上有用户所需内容；查询各个服务器当前的负载情况，判断哪一台服务器尚有服务能力。基于以上这些条件的综合分析之后，区域负载均衡设备会向全局负载均衡设备返回一台缓存服务器的 IP 地址。
6. 全局负载均衡设备把服务器的 IP 地址返回给用户。
7. 用户向缓存服务器发起请求，缓存服务器响应用户请求，将用户所需内容传送到用户终端。如果这台缓存服务器上并没有用户想要的内容，而区域均衡设备依然将它分配给了用户，那么这台服务器就要向它的上一级缓存服务器请求内容，直至追溯到网站的源服务器将内容拉到本地。

CDN 关键组件

- **LVS 做四层均衡负载**

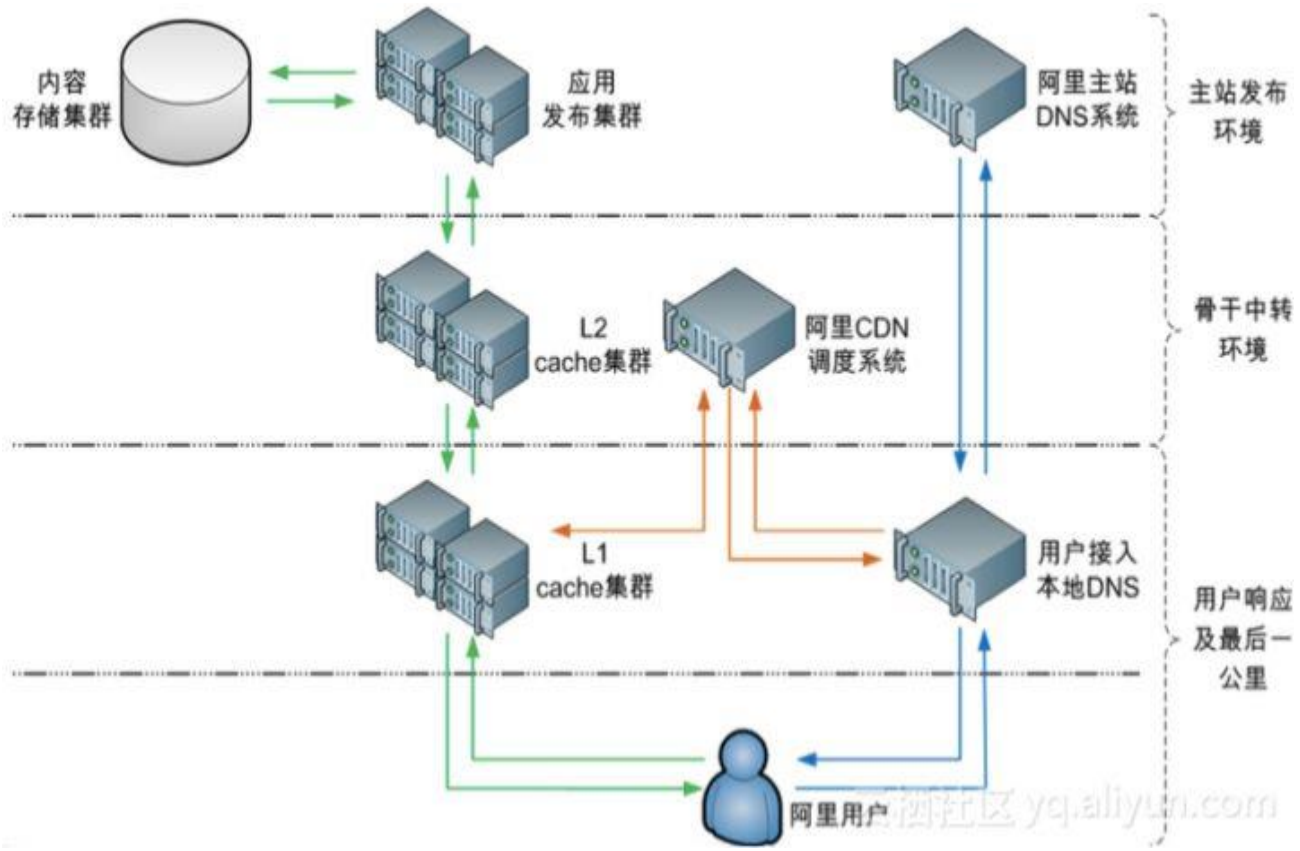
DR 模式
双 LVS 做 Active-Active 互备
负载均衡算法采用 wrr

- **Tengine 做七层负载均衡**

阿里基于 Nginx 开发的高性能 HTTP 服务器，已经开源。
主动健康检查
SPDY v3 支持

- Swift 做 HTTP 缓存

高性能 Cache
磁盘（SSD/SATA）



二、为什么要使用 CDN？或者说 CDN 能解决什么问题？

如果你在经营一家网站，那你应该知道几点因素是你制胜的关键：

- 内容有吸引力
- 访问速度快
- 支持频繁的用户互动
- 可以在各处浏览无障碍

另外，你的网站必须能在复杂的网络环境下运行，考虑到全球的用户访问体验。你的网站也会随着使用越来越多的对象（如图片、帧、CSS 及 APIs）和形形色色的动作（分享、跟踪）而系统逐渐庞大。所以，系统变慢带来用户的流失。

Google 及其它网站的研究表明，一个网站每慢一秒钟，就会丢失许多访客，甚至这些访客永远不会再次光顾这些网站。可以想像，如果网站是你的盈利渠道或是品牌窗口，那么网站速度慢将是一个致命的打击。

这就是你使用 CDN 的第一个也是最重要的原因：**为了加速网站的访问**

除此之外，CDN 还有一些作用：

1. 为了实现跨运营商、跨地域的全网覆盖

互联不互通、区域 ISP 地域局限、出口带宽受限制等种种因素都造成了网站的区域性无法访问。CDN 加速可以覆盖全球的线路，通过和运营商合作，部署 IDC 资源，在全国骨干节点商，合理部署 CDN 边缘分发存储节点，充分利用带宽资源，平衡源站流量。阿里云在国内有 500+节点，海外 300+节点，覆盖主流国家和地区不是问题，可以确保 CDN 服务的稳定和快速。

2. 为了保障你的网站安全

CDN 的负载均衡和分布式存储技术，可以加强网站的可靠性，相当无无形中给你的网站添加了一把保护伞，应对绝大部分的互联网攻击事件。防攻击系统也能避免网站遭到恶意攻击。

3. 为了异地备援

当某个服务器发生意外故障时，系统将会调用其他临近的健康服务器节点进行服务，进而提供接近 100%的可靠性，这就让你的网站可以做到永不宕机。

4. 为了节约成本投入

使用 CDN 加速可以实现网站的全国铺设，你根据不用考虑购买服务器与后续的托管运维，服务器之间镜像同步，也不用为了管理维护技术人员而烦恼，节省了人力、精力和财力。

5. 为了让你更专注业务本身

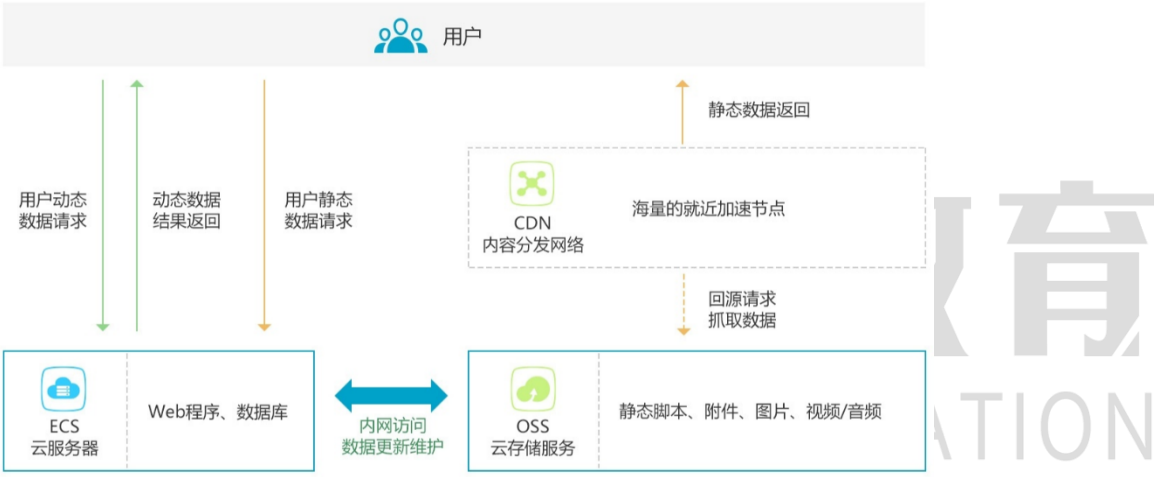
CDN 加速厂商一般都会提供一站式服务，业务不仅限于 CDN，还有配套的云存储、大数据服务、视频云服务等，而且一般会提供 7x24 运维监控支持，保证网络随时畅通，你可以放心使用。并且将更多的精力投入到发展自身的核心业务之上。

三、CDN 适用哪些场景？

1、网站站点/应用加速

站点或者应用中大量静态资源的加速分发，建议将站点内容进行动静分离，动态文件可以结合云服务器 ECS，静态资源如各类型图片、html、css、js 文件等，建议结合 对象存储 OSS 存储海量静态资源，可以有效加速内容加载速度，轻松搞定网站图片、短视频等内容分发。

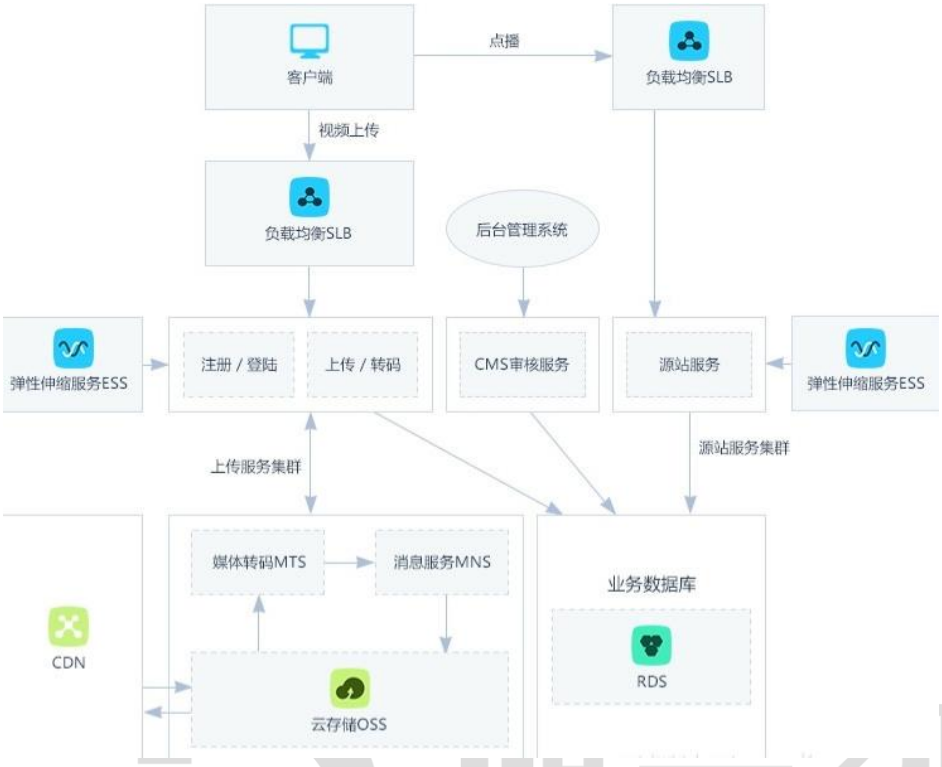
- 架构示意图



2、视音频点播/大文件下载分发加速

支持各类文件的下载、分发，支持在线点播加速业务，如 mp4、flv 视频文件或者平均单个文件大小在 20M 以上，主要的业务场景是视音频点播、大文件下载（如安装包下载）等，建议搭配对象存储 OSS 使用，可提升回源速度，节约近 2/3 回源带宽成本。

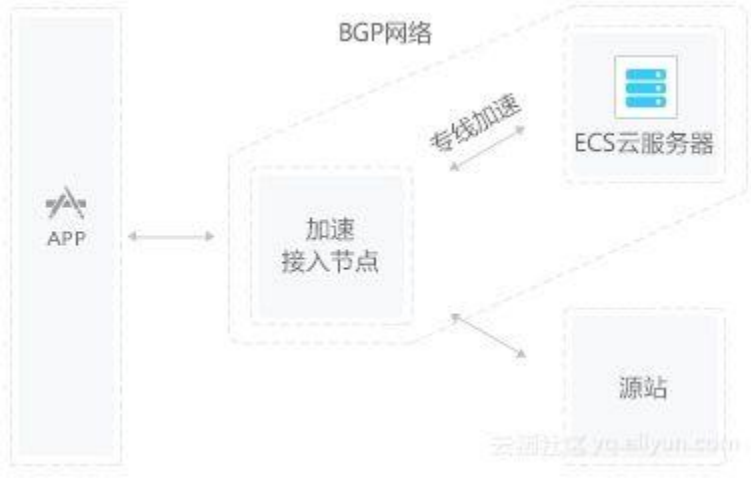
- 架构示意图



4、移动应用加速

移动 APP 更新文件（apk 文件）分发，移动 APP 内图片、页面、短视频、UGC 等内容的优化加速分发。提供 httpDNS 服务，避免 DNS 劫持并获得实时精确的 DNS 解析结果，有效缩短用户访问时间，提升用户体验。

- 架构示意图



四、如何加速

众所周知，当前是互联网时代，无论大大小小的公司，无论是互联网企业还是传统企业，统统离不开一个“网”字，而相比之下，硬件服务器、操作系统、应用组件等，在没有特殊必要需求的话（如果硬件不坏、软件或系统无急需修复的 BUG、无急需实现的客户需求），这些都是不用去主动改变。

相比之下网络则不同，最频繁变化的，最不想变化而又最无奈被动跟随变化的，就是网络质量，而 CDN 的缩写也是 Content Delivery Network，但网络问题存在一定的随机性和不可预测性。如果我们可以驾驭网络，相信在这个互联互通的互联网时代，我们将变得如虎添翼、叱咤风云！

为此，这里提出我对 CDN 的另一个解读：CDN - Can Do sth. on Network（我们可以在网络层面搞些事情）

提到优化，其实从 OSI 七层模型来讲（准确说是 TCP/IP 模型，二者是不同的，具体可以 google 一下），从物理层到应用层其实都是可以优化的，优化手段各异。

L1 物理层：硬件优化（升级硬件设备，承载更多业务）
L2 链路层：资源好坏（寻找更快的网络节点、确保 Lastmile 尽量短）
L3 路由层：路径优化（寻找 A 到 B 的最短路径）
L4 传输层：协议栈优化（TCP Optimization）
L7 应用层：能做的事情太多了（主要面向 HTTP 面向业务）

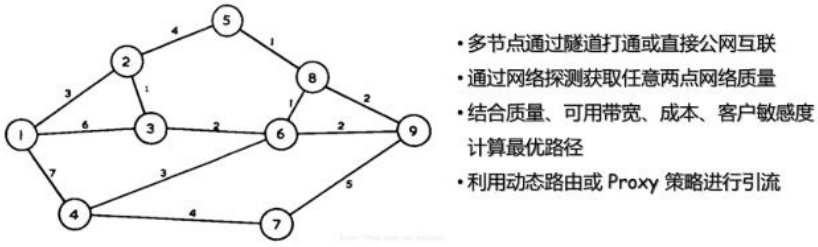
针对 CDN 而言，常用的是 L4 和 L7 的优化，例如上图所述。

- 分布式就近部署：确保网民访问到的 Cache Server 离他是最近的。
- 策略性缓存：针对明确已知的图片、CSS、JS 等，若源站更新并不快，但却没有明确高值缓存多长时间，CDN Cache Server 可以自定义一个缓存时间（例如 60s），这样可以确保在 60s 之内的同样请求会快速给出数据而不用穿越整个 Internet 从源站获取。
- 传输路径优化：寻找从边缘 CDN Cache Server 经 upper CDN Cache Server 到源站的最优传输路径，确保动态传输的数据可以走端到端的最优路径。
- 连接加速：通过修改协议栈的 Handshake Timer 来实现快速重试，以弥补由于丢包导致的重试超时。
- 传输层优化：这里主要是指 TCP 协议，针对 TCP 协议可以做很多优化策略，由于篇幅问题后面再讲。
- 内容预取：解析 WEB 内容，对于里面的 Object，在网民请求之前，优先由 CDN Cache Server 主动获取并缓存下来，以缩短数据交互时间，提升网民体验感。
- 合并回源：当有多个人先后下载同一个还未缓存住的内容时（例如一个 mp4 视频文件），CDN Cache Server 做到合并连接从 upstream 拿数据，而不是几个请

求，几个 to uptream connection，既做到了带宽成本优化，又做到了给 upstream 降载，后请求的人还能享受之前 CDN Cache Server 已经下载过的部分文件，这部分内容直接 HIT，只有当追上第一个 downloader 的时候才一起等待 MISS 数据。

- 持久连接池：在 Middlemile 之间预先建立好 TCP Connection，并一直保持不断开，当网民有新请求过来时，边缘 CDN Cache Server 直接借助与 upper 建立好连接，直接发送 HTTP 的 GET/POST 报文到 upper CDN Cache Server，进行 TCP “去握手化”，减少由于 TCP 连接建立而造成的时间损耗（多适用于高并发小文件请求）。
- 主动压缩：很多源站由于规划设计问题或担心负载过高问题，页面中的 HTML、CSS、JS 文件（这种文件具有高度可压缩性）并未压缩传输，此时 CDN Cache Server 可以主动对其进行压缩后传输并缓存，以减少传输量、降低交互时间、提升用户体验感。
- Offline：当源站挂了怎么办？网民访问时，会拿不到数据。那么 CDN 此时可以策略性发送最新缓存的一份旧数据给网民，而不是生硬的告知用户不可访问，以提升用户体验感。

1、路径优化



如前文所述，所谓路径优化就是找到两点间的最优路径。

对于网络而言，A 到 B 最快 ≠ A 距离 B 最近，从广东联通访问福建联通，可能不如广东联通经北京联通再到福建联通更快，因此要对节点做实时探测，计算最优路径。

计算最优路径时，还要考虑带宽饱和度、成本、客户敏感度问题综合计算，因此不是看上去那么简单的。

带宽饱和度：作为中转节点（例如上例所说的北京），如果带宽本身已经没有多少剩余，那么穿越北京的路径优化可能会作为压死大象的最后一根稻草，使原本还 OK 的北京节点变得不堪重负。

成本：还以北京为例，北京资源的带宽成本肯定远高于其它省市，例如比河北联通、天津联通可能要贵很多，但可能只比河北天津慢几个毫秒（运动员起跑时最快的反应时间是 150 毫秒），那么为了这几毫秒要多支付很多带宽费用显然是不值当的，那么利用北京进行中转显然就是不值得的（当然，有的时候就是为了和对手 PK，那也没办法）。

客户敏感度：有了中转路径，提速效果当然是好的，但如前文所述也是有代价的，那么是所有业务流量都走最优路径呢？还是只让个别业务走最优路径呢？这个就要看客户敏感度了。例如重点大客户，例如对质量要求较高的高价优质客户，这些客户可能就是首选。

2、传输层优化

如前文所述，所谓传输层优化主要是指 TCP 优化，这是所有互联网行业的通用技术，是重中之重的领域，TCP 优化如果做的好，可弥补节点质量低下而造成的响应时间过大的损失。

赛马比赛时，有好马当然跑的快。如果马一般（不是太差），骑手的骑术精湛，或许同样也可以得第一，就是这个道理！

另外一点 TCP 优化重要的原因在于，TCP 是互联网尤其是 CDN 的基础协议，基本上所有业务都是 over TCP 来进行传输的，如果将 TCP 优化做好，受益面非常广，可以说全局收益（不仅是提升客户体验感，也能提升内部支撑系统的使用体验感，例如日志传输、配置下发等）。

谈到 TCP 优化，需要先将 TCP 协议基础知识。需要首先明确一些名词属于的概念。

- CWND：Congestion Window，拥塞窗口，负责控制单位时间内，数据发送端的报文发送量。TCP 协议规定，一个 RTT（Round-Trip Time，往返时延，大家常说的 ping 值）时间内，数据发送端只能发送 CWND 个数据包（注意不是字节数）。TCP 协议利用 CWND/RTT 来控制速度。
- SS：Slow Start，慢启动阶段。TCP 刚开始传输的时候，速度是慢慢涨起来的，除非遇到丢包，否则速度会一直指数性增长（标准 TCP 协议的拥塞控制算法，例如 cubic 就是如此。很多其它拥塞控制算法或其它厂商可能修改过慢启动增长特性，未必符合指数特性）。
- CA：Congestion Avoid，拥塞避免阶段。当 TCP 数据发送方感知到有丢包后，会降低 CWND，此时速度会下降，CWND 再次增长时，不再像 SS 那样指数增，而是线性增（同理，标准 TCP 协议的拥塞控制算法，例如 cubic 是这样，很多其它拥塞控制算法或其它厂商可能修改过慢启动增长特性，未必符合这个特性）。
- ssthresh：Slow Start Threshold，慢启动阈值。当数据发送方感知到丢包时，会记录此时的 CWND，并计算合理的 ssthresh 值（ssthresh <= 丢包时的 CWND），当 CWND 重新由小至大增长，直到 sshtresh 时，不再 SS 而是 CA。但因为数据确认超时（数据发送端始终收不到对端的接收确认报文），发送端会骤降 CWND 到最初始的状态。

TCP 优化实际上是在用带宽换用户体验感，低成本低质量网络虽然可以通过 TCP 优化提升体验感，但综合成本甚至可能比直接采购优质高价节点更高，效果还未必比优质节点直接服务好。

TCP 之所以叫优化不叫加速，是因为它可以让那些原本应当传输很快，由于算法不合理导致的传输变慢的传输变得快起来，但却不能让那些链路质量原本没有问题的变得更快。

还有一些其他优化

- 建连优化：TCP 在建立连接时，如果丢包，会进入重试，重试时间是 1s、2s、4s、8s 的指数递增间隔，缩短定时器可以让 TCP 在丢包环境建连时间更快，非常适用于高并发短连接的业务场景。
- 首包优化：此优化其实没什么实质意义，若要说一定会有意义的话，可能就是满足一些评测标准的需要吧，例如有些客户以首包时间作为性能评判的一个依据。所谓首包时间，简单解释就是从 HTTP Client 发出 GET 请求开始计时，到收到 HTTP 响应的时间。为此，Server 端可以通过 TCP_NODELAY 让服务器先吐出 HTTP 头，再吐出实际内容（分包发送，原本是粘到一起的），来进行提速和优化。据说更有甚者先让服务器无条件返回 "HTTP/" 这几个字符，然后再去 upstream 拿数据。这种做法在真实场景中没有任何帮助，只能欺骗一下探测者罢了，因此还没见过有直接发 "HTTP/" 的，其实是一种作弊行为。

- 平滑发包：如前文所述，在 RTT 内均匀发包，规避微分时间内的流量突发，尽量避免瞬间拥塞，此处不再赘述。
- 丢包预判：有些网络的丢包是有规律性的，例如每隔一段时间出现一次丢包，例如每次丢包都连续丢几个等，如果程序能自动发现这个规律（有些不明显），就可以针对性提前多发数据，减少重传时间、提高有效发包率。
- RTO 探测：如前文讲 TCP 基础时说过的，若始终收不到 ACK 报文，则需要触发 RTO 定时器。RTO 定时器一般都时间非常长，会浪费很多等待时间，而且一旦 RTO，CWND 就会骤降（标准 TCP），因此利用 Probe 提前与 RTO 去试探，可以规避由于 ACK 报文丢失而导致的速度下降问题。
- 带宽评估：通过单位时间内收到的 ACK 或 SACK 信息可以得知客户端有效接收速率，通过这个速率可以更合理的控制发包速度。
- 带宽争抢：有些场景（例如合租）是大家互相挤占带宽的，假如你和室友各 1Mbps 的速度看电影，会把 2Mbps 出口占满，而如果一共有 3 个人看，则没人只能分到 1/3。若此时你的流量流量达到 2Mbps，而他俩还都是 1Mbps，则你至少仍可以分到 2/(2+1+1) * 2Mbps = 1Mbps 的 50% 的带宽，甚至更多，代价就是服务器侧的出口流量加大，增加成本。（TCP 优化的本质就是用带宽换用户体验感）
- 链路质量记忆：如果一个 Client IP 或一个 C 段 Network，若已经得知了网络质量规律（例如 CWND 多大合适，丢包规律是怎样的等），就可以在下次连接时，优先使用历史经验值，取消慢启动环节直接进入告诉发包状态，以提升客户端接收数据速率。