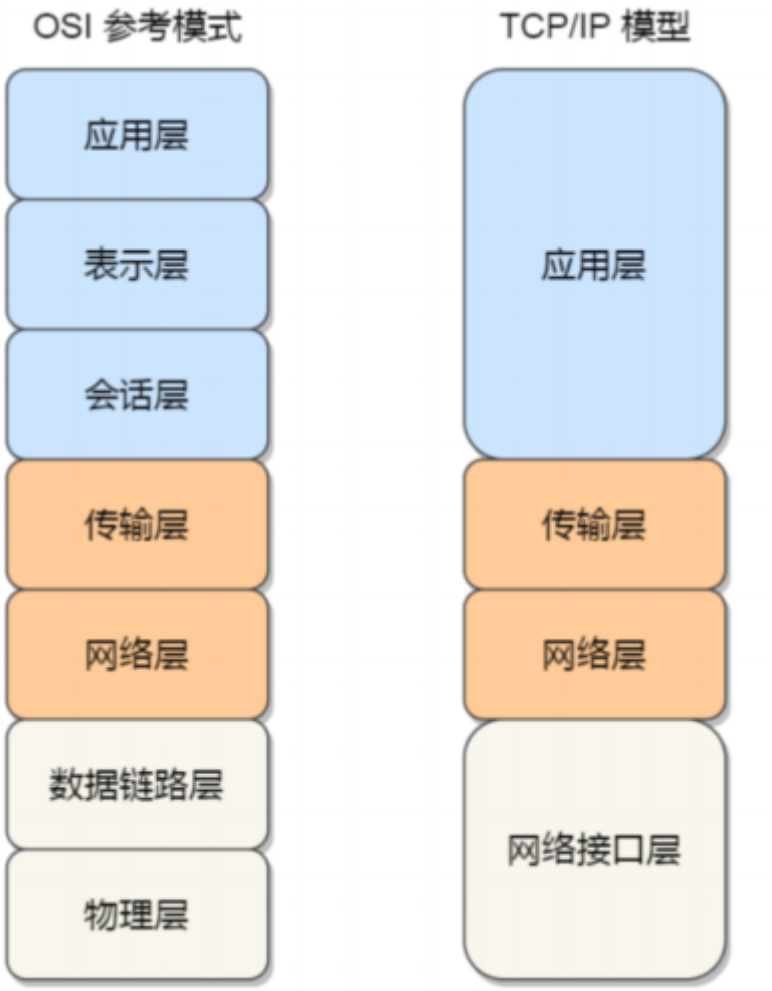
计算机网络



OSI七层参考模型

- 物理层：负物理传输媒介的传输。主要作用是传输比特流，这一层的数据叫做比特。
- 数据链路层：建立逻辑连接，进行硬件地址寻址、差错校验等功能。定义了如何让格式化数据以帧为单位进行传输，以及如何控制对物理介质的访问。将比特组合成字节进而组合成帧，用MAC地址访问介质，传输单位是帧。
- 网络层：负责数据的路由和转发，选择最佳路径将数据从源主机传输到目标主机。它使用IP地址来标识不同主机和网络，并进行逻辑地址寻址。传输单位是数据报。
- 传输层：提供端到端的数据传输服务，它使用TCP（传输控制协议）和UDP（用户数据报协议）来管理数据传输。
- 会话层：建立、管理和终止应用程序之间的会话连接。它处理会话建立、维护和终止，以及处理会话过程中的异常情况。
- 表示层：负责数据的格式转换、加密和解密，确保数据在不同系统之间的正确解释和呈现，也就是把计算机能够识别的东西转换成人能够识别的东西。
- 应用层：网络服务与最终用户的一个接口。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。常见的协议有：FTP、SMTP、HTTP、DNS。

TCP/IP网络模型有哪几层？每一层有什么作用？

- 应用层：负责向用户提供一组应用程序，比如 HTTP、DNS、FTP 等;
- 传输层：负责端到端的通信，比如 TCP、UDP 等；
- 网络层：负责网络包的封装、分片、路由、转发，比如 IP、ICMP 等；
- 网络接口层：负责网络包在物理网络中的传输，比如网络包的封帧、 MAC 寻址、差错检测，以及通过网卡传输网络帧等；

在浏览器中输入URL并按下回车之后会发生什么

- 输入URL并解析
- DNS域名解析，将域名解析成对应的IP地址
- 建立起TCP连接之三次握手
- 浏览器发送HTTP/HTTPS请求到web服务器
- 服务器处理HTTP请求并返回HTTP报文
- 浏览器渲染页面
- 断开连接之TCP四次挥手

DNS是什么

DNS是一种用于将域名转换为IP地址的分布式系统。在互联网上，计算机和其他网络设备使用IP地址来相互识别和通信。然而，

IP地址是一串数字，不太方便人们使用和记忆，所以就使用了域名来代替复杂的IP地址。

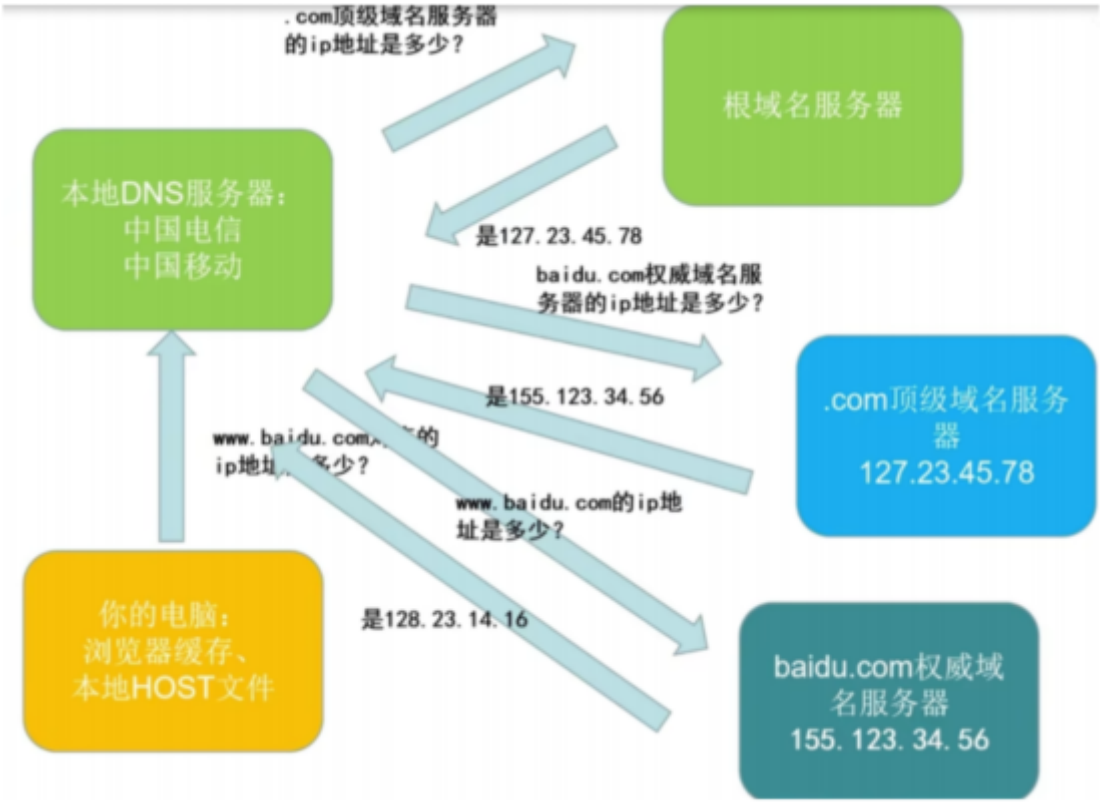
DNS服务器为什么不采用集中式的设计

- **单点故障**：如果 DNS 服务器崩溃，那么整个网络随之瘫痪。
- **远距离集中式数据库**：单个 DNS 服务器不可能 邻近 所有的用户，假设在美国的 DNS 服务器不可能临近让澳大利亚的查询使用，其中查询请求势必会经过低速和拥堵的链路，造成严重的时延;
- **维护**：维护成本巨大，而且还需要频繁更新。

域名层级关系

DNS 中的域名都是用句点来分隔的，比如 www.server.com ，这里的句点代表了不同层次之间的界限。在域名中，越靠右的位置表示其层级越高。

DNS解析过程是怎么样的



- 先查询浏览器缓存中是否有该域名对应的IP地址
- 如果浏览器缓存中没有，会去计算机的Host文件中查询是否有对应的缓存
- 如果Host文件中没有则会向本地的DNS服务器发送一个DNS查询请求
- 如果本地DNS解析器有该域名的ip地址，就会直接返回，如果没有缓存该域名的解析记录，它会向**根DNS服务器**发出查询请求。根DNS服务器并不负责解析域名，但它能告诉本地DNS解析器应该向哪个顶级域（.com/.net/.org）的DNS服务器继续查询。
- 本地DNS解析器接着向指定的**顶级域名DNS服务器**发出查询请求。顶级域DNS服务器也不负责具体的域名解析，但它能告诉本地DNS解析器应该前往哪个权威DNS服务器查询下一步的信息。
- 本地DNS解析器最后向**权威DNS服务器**发送查询请求。 权威DNS服务器是负责存储特定域名和IP地址映射的服务器。当权威DNS服务器收到查询请求时，它会查找"example.com"域名对应的IP地址，并将结果返回给本地DNS解析器。
- 本地DNS解析器将收到的IP地址返回给浏览器，并且还会将域名解析结果缓存在本地，以便下次访问时更快地响应。

DNS查询过程中用于获取域名解析信息的方法是什么

- **递归查询**：DNS客户端（通常是本地DNS解析器）向上层DNS服务器（如根域名服务器、顶级域名服务器）发起查询请求，并要求这些服务器直接提供完整的解析结果。递归查询的特点是，DNS客户端只需要发送一个查询请求，然后等待完整的解析结果。上层DNS服务器会自行查询下一级的服务器，并将最终结果返回给DNS客户端。
- **迭代查询**：DNS客户端向上层DNS服务器发起查询请求，但不要求直接提供完整的解析结果。相反，DNS客户端只是询问上层服务器一个更高级的域名服务器的地址，然后再自行向那个更高级的服务器发起查询请求，以此类推，直到获取完整的解析结果为止。

HTTP有哪些特性

- 简单：基本报文格式为header+body，头部信息也是key-value简单文本的形式，易于理解。
- 灵活和易于扩展
    - HTTP协议里的各种请求方法、URI/URL、状态码、头字段等每个组成要求都没有被固定死，允许开发人员自定义和扩充;

    - HTTP工作在应用层（OSI第七层），下层可以随意变化;
    - HTTPS就是在HTTP与TCP之间增加了SSL/TSL安全传输层，HTTP/3把TCP换成了基于UDP的QUIC。
- 无状态、明文传输、不安全
    - 服务器不会去记忆HTTP的状态，所以不需要额外的资源来记录状态信息，这能减轻服务器的负担。但它在完成有关联性的操作时会非常麻烦。

      > 可用**Cookie** 技术， Cookie 通过在请求和响应报文中写入 Cookie 信息来控制客户端的状态。

    - 明文传输为我们调试工作带来了极大的便利性，但信息透明，容易被窃取。
    - 通信使用明文（不加密），内容可能被窃听，不验证通信方的身份，因此有可能遭遇伪装，无法证明报文的完整性，所以有可能已遭篡改。

HTTP版本演变过程

- HTTP/0.9：最早的HTTP版本，在1991年就已经发布，只支持 GET 方法，也没有请求头，服务器只能返回 HTML格式的内容。
- HTTP/1.0：HTTTP 协议的第一个正式版本
    - 引入了请求头和响应头，支持多种请求方法和状态码
    - 不支持持久连接，每次请求都需要建立新的连接
- HTTP/1.1

  解决问题：

    - 长连接：为了解决 HTTP/1.0 每次请求都需要建立新的连接的问题， HTTP/1.1 提出了长**连接（持久连接）**，只要客户端和服务器任意一端没有明确提出断开连接，则保持TCP连接状态。
    - 管道网络传输：在同一个 TCP 连接里面，客户端可以发起多个请求，只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以**减少整体的响应时间。**

  存在问题：

    - 队头阻塞：当顺序发送的请求序列中的一个请求因为某种原因被阻塞时，在后面排队的所有请求也一同被阻塞了，会招致客户端一直请求不到数据，这也就是「**队头阻塞**」
    - 头部冗余：每个请求和响应都需要带有一定的头部信息，每次互相发送相同的首部造成的浪费较多;
    - 请求只能从客户端开始，服务器只能被动响应。
    - 没有请求优先级控制。
- HTTP/2：协议是基于 HTTPS 的, 所以HTTP/2的安全性也是有保障的

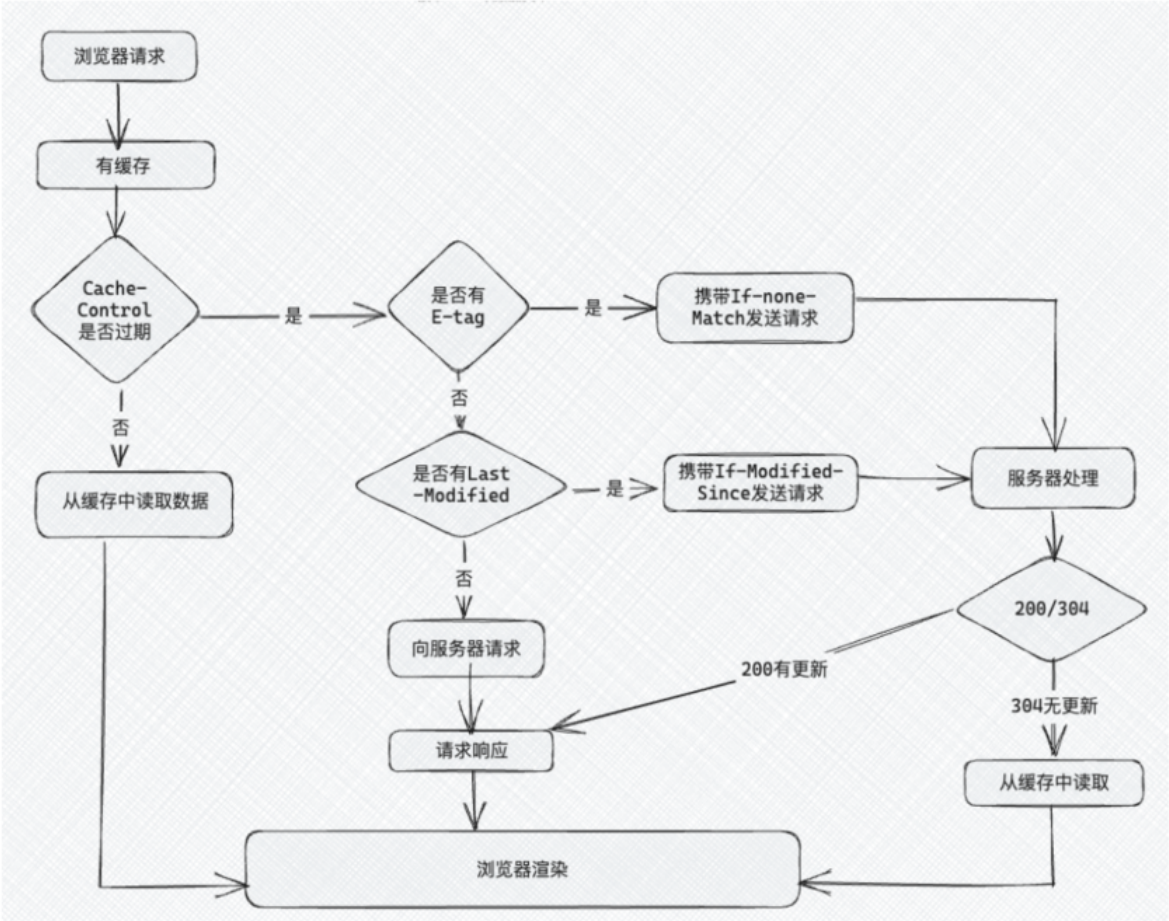
  解决问题：

    - 头部压缩：HTTP/2 使用 HPACK 压缩算法对请求和响应头部进行压缩，减少了传输的头部数据量，降低了延迟
    - 二进制帧：HTTP/2 将数据分割成二进制帧进行传输，分为头信息帧和数据帧，增加了数据传输的效率。
    - 并发传输：引出了 Stream 概念，多个 Stream 复用在一条 TCP 连接，针对不同的 HTTP 请求用独一无二的Stream ID 来区分，接收端可以通过 Stream ID 有序组装成 HTTP 消息，不同 Stream 的帧是可以乱序发送的，因此可以并发不同的 Stream ，也就是 HTTP/2 可以并行交错地发送请求和响应。
    - 服务器推送：在 HTTP/2 中，服务器可以对客户端的一个请求发送多个响应，即服务器可以额外的向客户端推送资源，而无需客户端明确的请求。

  存在问题：

    - HTTP/2 是基于 TCP 协议来传输数据的，TCP 是字节流协议，TCP 层必须保证收到的字节数据是完整且连续的，这样内核才会将缓冲区里的数据返回给 HTTP 应用，那么当「前 1 个字节数据」没有到达时，后收到的字节数据只能存放在内核缓冲区里，只有等到这 1 个字节数据到达时，HTTP/2 应用层才能从内核中拿到数据，这就是 HTTP/2队头阻塞问题。
- HTTP/3：HTTP/3基于 QUIC 协议，把HTTP下层的TCP协议改成了UDP！
    - 零RTT连接建立：QUIC 允许在首次连接时进行零往返时间（Zero Round Trip Time）连接建立，从而减少了连接延迟，加快了页面加载速度。
    - 无队头阻塞: QUIC 使用 UDP 协议来传输数据。一个连接上的多个stream之间没有依赖, 如果一个stream丢了一个UDP包，不会影响后面的stream，不存在 TCP 队头阻塞
    - 连接迁移：QUIC 允许在网络切换（如从 Wi-Fi 到移动网络）时，将连接迁移到新的 IP 地址，从而减少连接的中断时间。
    - 向前纠错机制：每个数据包除了它本身的内容之外，还包括了部分其他数据包的数据，因此少量的丢包可以通过其他包的冗余数据直接组装而无需重传。向前纠错牺牲了每个数据包可以发送数据的上限，但是减少了因为丢包导致的数据重传。

什么是HTTP缓存，并详细谈一谈有哪些策略以及原理是怎样的



对于已经请求过的资源，客户端或代理服务器会将其副本保存在本地存储中，并且在下次请求同一资源时，首先检查缓存中是否存在有效的副本。如果存在有效的缓存，就直接读取本地的数据，不必在通过网络获取服务器的响应了，这就是 **HTTP缓存**

- 强制缓存：浏览器判断请求的目标资源是否有效命中强缓存，如果命中，则可以直接从内存中读取目标资源，无需与服务器做任何通讯。
    - Expires强缓存 ： 设置一个强缓存时间，此时间范围内，从内存中读取缓存并返回，因为 Expires 判断强缓存过期的机制是**获取本地时间戳**，与之前拿到的资源文件中的 Expires 字段的时间做比较。来判断是否需要对服务器发起请求, 所以这里有一个巨大的漏洞：“如果我本地时间不准咋办？”正是因为这个原因，该字段目前已经基本上被废弃了。

- Cache-Control强缓存： http1.1 中增加该字段，只要在资源的响应头上写上需要缓存多久就好了，单位是秒。
    - max-age 决定客户端资源被缓存多久。
    - s-maxage 决定代理服务器缓存的时长。
    - no-cache 表示是强制进行协商缓存。
    - no-store 是表示禁止任何缓存策略。
    - public 表示资源既可以被浏览器缓存也可以被代理服务器缓存。
    - private 表示资源只能被浏览器缓存，默认为private

    流程：
    - 当浏览器第一次请求访问服务器资源时，服务器会在返回这个资源的同时，在 Response 头部加上 Cache-Control，Cache-Control 中设置了过期时间大小；
    - 浏览器再次请求访问服务器中的该资源时，会先**通过请求资源的时间与 Cache-Control 中设置的过期时间大小，来计算出该资源是否过期**，如果没有，则使用该缓存，否则重新请求服务器；
    - 服务器再次收到请求后，会再次更新 Response 头部的 Cache-Control。
- 协商缓存：通过服务端告知客户端是否可以使用缓存的方式被称为协商缓存。
    - 基于 Last-Modified 和 If-Modified-Since 的协商缓存

      流程：
      - 首先需要在服务器端读出文件修改时间
      - 将读出来的修改时间赋给响应头的 Last-Modified 字段
      - 最后设置 Cache-control:no-cache
      - 当客户端读取到 Last-Modified 的时候，会在下次的请求标头中携带一个字段: If-Modified-Since，而这个请求头中的 If-Modified-Since 就是服务器第一次修改时候给他的时间
      - 之后每次对该资源的请求，都会带上 If-Modified-Since 这个字段，而服务端就需要拿到这个时间并再次读取该资源的修改时间，让他们两个做一个比对来决定是读取缓存还是返回新的资源
      - 如果最后修改时间较新（大），说明资源又被改过，则返回最新资源， HTTP 200 OK 如果最后修改时间较旧（小），说明资源无新修改，响应 HTTP 304 走缓存。

      缺点：
      - 因为是根据文件修改时间来判断的，所以，在文件内容本身不修改的情况下，依然有可能更新文件修改时间（比如修改文件名再改回来），这样，就有可能文件内容明明没有修改，但是缓存依然失效了。
      - 当文件在极短时间内完成修改的时候（比如几百毫秒）。因为文件修改时间记录的最小单位是秒，所以，如果文件在几百毫秒内完成修改的话，文件修改时间不会改变，这样，即使文件内容修改了，依然不会 返回新的文件。
    - 基于 ETag 的协商缓存：将原先协商缓存的比较**时间戳**的形式修改成了比较**文件指纹（根据文件内容计算出的唯一哈希值）**。

      流程：
      - 第一次请求某资源的时候，服务端读取文件并计算出文件指纹，将文件指纹放在响应头的 Etag 字段中跟资源一起返回给客户端。
      - 第二次请求某资源的时候，客户端自动从缓存中读取出上一次服务端返回的 ETag 也就是文件指纹。并赋给请求头的 If-None-Match 字段，让上一次的文件指纹跟随请求一起回到服务端。
      - 服务端拿到请求头中的 If-None-Match 字段值（也就是上一次的文件指纹），并再次读取目标资源并生成文件指纹，两个指纹做对比。如果两个文件指纹完全吻合，说明文件没有被改变，则直接返回 304 状态码和一个空的响应体并return。如果两个文件指纹不

吻合，则说明文件被更改，那么将新的文件指纹重新存储到响应头的ETag中并返回给客户端

缺点：

- ETag需要计算文件指纹这样意味着，服务端需要更多的计算开销。如果文件尺寸大，数量多，并且计算频繁，那么ETag的计算就会影响服务器的性能。显然，ETag在这样的场景下就不是很适合。
- ETag有强验证和弱验证，所谓将强验证，ETag生成的哈希码深入到每个字节。哪怕文件中只有一个字节改变了，也会生成不同的哈希值，它可以保证文件内容绝对的不变。但是，强验证非常消耗计算量。ETag还有一个弱验证，弱验证是提取文件的部分属性来生成哈希值。因为不必精确到每个字节，所以他的整体速度会比强验证快，但是准确率不高。会降低协商缓存的有效性。

HTTPS有什么特点

- 信息加密：采用对称加密+非对称加密的混合加密的方式，对传输的数据加密，实现信息的机密性，解决了窃听的风险。
- 校验机制：用摘要算法为数据生成独一无二的「指纹」校验码，指纹用来校验数据的完整性，解决了被篡改的风险。
- 身份证书：将服务端的公钥放入到CA数字证书中，解决了服务端被冒充的风险。

HTTPS有哪些优点和缺点

优点

- 在数据传输过程中，使用秘钥加密，安全性更高
- 可认证用户和服务器，确保数据发送到正确的用户和服务器

缺点

- 握手阶段延时较高：在会话前还需进行SSL握手
-
- 部署成本高：需要购买CA证书；需要加解密计算，占用CPU资源，需要服务器配置或数目高

HTTP和HTTPS有什么区别

- HTTP： 以明文的方式在网络中传输数据，HTTPS 解决了HTTP 不安全的缺陷，在 TCP 和 HTTP 网络层之间加入了 SSL/TLS 安全协议，使得报文能够**加密传输**。
- HTTPS 在 TCP 三次握手之后，还需进行 SSL/TLS 的握手过程，才可进入加密报文传输。
- HTTP 的端口号是 80，HTTPS 的端口号是 443。
- HTTPS 协议需要向 CA（证书权威机构） 申请数字证书，来保证服务器的身份是可信的。

HTTPS的建立流程是怎样的

- 客户端向服务器端发送加密通信请求，请求与服务端建立连接。
- 服务端产生一对公私钥（**公钥是密钥对中公开的部分，私钥则是非公开的部分。** 公钥通常用于加密会话密钥、验证数字签名，或加密可以用相应的私钥解密的数据。），然后将自己的公钥发送给CA机构，CA机构也有一对公私钥，然后CA机构使用自己的私钥将服务器发送来的公钥进行加密，产生一个CA数字证书。
- 服务端响应客户端的请求，将CA机构生成的数字证书发送给客户端。
- 客户端将服务端发来的数字证书进行解析，验证这个CA数字证书是否合法，如果不合法，会发送一个警告。如果合法，从数字证书中取出服务端生成的公钥。
- 客户端取出公钥并生成一个随机码key。

- 客户端将加密后的随机码key发送给服务端，作为接下来的对称加密的密钥。
- 服务端接收到随机码key后，使用自己的私钥对它进行解密，然后获得随机码key。
- 服务端使用随机码key对传输的数据进行加密，在传输加密后的内容给客户端。
- 客户端使用自己生成的随机码key解密服务端发送过来的数据，之后客户端和服务端通过对称对称加密传输数据，随机码key作为传输的密钥。

SSL/TLS是什么

SSL是安全套接字，TLS是安全传输层协议，HTTPS是基于TLS/SSL的安全版本的HTTP协议

SSL和TLS协议通过以下方式确保安全通信：

- **加密：** 使用加密算法对传输的数据进行加密，防止第三方截取和读取敏感信息。
- **身份验证：** 使用数字证书验证通信双方的身份，确保数据传输的可信性。
- **数据完整性：** 通过使用消息摘要算法，确保传输的数据在传输过程中没有被篡改或损坏。

什么是对称加密和非对称加密

- 对称加密：对称加密也称为私钥加密，使用相同的密钥来进行加密和解密。
    - 在加密过程中，明文数据通过应用特定的算法和密钥进行加密，生成密文数据。解密过程则是将密文数据应用同样的算法和密钥进行解密，恢复为明文数据。
    - 由于加密和解密都使用相同的密钥，因此对称加密算法的速度通常较快，但密钥的安全性很重要。如果密钥泄漏，攻击者可以轻易地解密数据。
- 非对称加密：非对称加密也称为公钥加密，使用一对不同但相关的密钥：公钥和私钥。
    - 公钥用于加密数据，私钥用于解密数据。如果使用公钥加密数据，只有拥有相应私钥的人才能解密数据；如果使用私钥加密数据，可以使用相应公钥解密。
    - 除了加密和解密，非对称加密还用于【数字签名】，可以验证消息的来源和完整性。

TCP和UDP的概念和特点

TCP是面向连接的、可靠的、基于字节流的传输层通信协议。

- 面向连接的协议：在通信之前，TCP需要在发送方和接收方之间建立连接，然后在通信完成后关闭连接。这种连接和建立过程称为“三次握手”和“四次握手”，确保可靠的数据传输。
- TCP提供可靠的数据传输：它使用序列号和确认机制来确保数据包的有序性和完整性。如果数据包丢失或损坏，TCP会重新发送丢失的数据包，直到接收方正确接收为止。
- 流量控制和拥塞控制：TCP使用流量控制和拥塞控制算法，确保数据发送的速率不会超过接收方的处理能力，并防止网络拥塞。
- 有序传输：TCP确保数据包按照发送的顺序到达接收方，并在接收方重新组装成正确的顺序。
- 适用于可靠传输的场景：TCP适用于那些对数据传输可靠性要求较高的应用。

UDP是一种无连接的协议

- 无连接：UDP在通信之前不需要建立连接，直接发送数据包。
- 不可靠：UDP不提供可靠的数据传输。它发送数据包后不会关心数据包是否成功到达接收方。
- 速度极快：由于没有连接建立和确认过程，UDP传输速度较快，适用于实时传输，如实时音频和视频流。
- 无序传输：UDP不保证数据包的有序性，因此接收方接收到的数据包可能是无序的。
- 适用于实时传输的场景：UDP适用于对数据传输可靠性要求不高的场景

如何用UDP实现可靠传输

为什么需要可靠的UDP?

在弱网（2G、3G、信号不好）环境下，使用 TCP 连接的延迟很高，影响体验。使用 UDP 是很好的解决方案，既然把 UDP 作为弱网里面的 TCP 来使用，就必须保证数据传输能像 TCP 一样可靠。

实现的关键在于两点

- 提供超时重传，能避免数据报丢失。
- 提供确认序列号，可以对数据报进行确认和排序。

本端：首先在UDP数据报定义一个首部，首部包含确认序列号和时间戳，时间戳是用来计算RTT(数据报传输的往返时间)，计算出合适的RTO(重传的超时时间)。然后以等-停的方式发送数据报，即收到对端的确认之后才发送下一个的数据报。当时间超时，本端重传数据报，同时RTO扩大为原来的两倍，重新开始计时。

对端：接受到一个数据报之后取下该数据报首部的时间戳和确认序列号，并添加本端的确认数据报首部之后发送给对端。根据此序列号对已收到的数据报进行排序并丢弃重复的数据报。

Keep-Alive是什么

Keep-Alive是一种协议的机制，也被称为HTTP长连接。

在启用 Keep-alive 的情况下，客户端和服务器在完成一个 HTTP 请求和响应后，并不立即关闭连接，而是继续保持连接处于打开状态。在连接保持打开的情况下，客户端可以继续发送其他请求，服务器可以继续发送响应，而无需重新建立连接，减少了连接的建立和关闭的开销，从而提高性能和效率。

Keep-Alive的优缺点

优点：

- TCP 连接的建立和关闭需要时间和资源，通过保持连接打开，可以减少这些开销，从而提高性能和效率。
- 客户端可以在同一个连接上同时发送多个请求，服务器可以并行地处理这些请求，提高并发性能。
- Keep-alive 连接中的多个请求共享同一个连接的头部信息（如用户代理、Cookie 等），减少了头部信息的重复传输。

缺点：

- 长时间的持久连接可能会占用服务器资源，特别是在高并发的情况下。为了平衡资源利用和性能，服务器和客户端通常会**设置 Keep-alive 的超时时间**，以便在一段时间内保持连接打开，超过该时间则关闭连接。

TCP KeepAlive是什么

TCP Keep-Alive 是在操作系统和网络协议栈级别实现的，它通过发送特定的探测数据包来维护连接的活跃性。

- 在启用 TCP Keep-Alive 的情况下，操作系统会定期发送一些特定的探测数据包到连接的另一端。这些数据包通常是空的，没有实际的数据内容。
- 如果一端收到了探测数据包，它会回复一个确认（ACK）数据包。如果一段时间内没有收到确认数据包，发送端将认为连接可能已经断开，从而触发连接关闭。
- TCP Keep-Alive 的主要目的是检测连接是否处于空闲状态，即没有实际数据传输。它不仅可以检测到连接断开，还可以在空闲连接超过一定时间时释放连接，从而释放资源。

TCP的KeepAlive和HTTP的Keep-Alive是一个东西吗?

- HTTP 的 Keep-Alive，是由应用层实现的，称为 HTTP 长连接。HTTP 的 Keep-Alive 实现了使用同一个 TCP 连接来发送和接收多个 HTTP请求/应答，避免了连接建立和释放的开销，这就是 **HTTP 长连接**。
- TCP 的 Keepalive，是由TCP 层（内核态）实现的，称为 TCP 保活机制，是一种用于在 TCP 连接上检测空闲连接状态的机制

CDN是什么?

CDN，全称为内容分发网络, 将内容存储在分布式的服务器上，使用户可以从距离较近的服务器获取所需的内容，从而减少数据传输的时间和距离，提高内容的传输速度、减少延迟和提升用户体验。

说一说CDN的工作流程

- 当用户输入一个域名或点击一个链接时，首先会进行域名解析。如果网站启用了 CDN，DNS 解析会返回距离用户最近的 CDN 节点的 IP 地址，而不是原始源服务器的 IP 地址。
- 用户的请求会被路由到距离最近的 CDN 节点，并且CDN 节点可以根据服务器的负载和可用性，动态地将请求分发到最适合的服务器节点上。
- CDN 会首先检查是否已经缓存了该资源。如果有缓存，CDN 节点会直接返回缓存的资源，如果没有缓存所需资源，它会从源服务器（原始服务器）回源获取资源，并将资源缓存到节点中，以便以后的请求。

CDN是如何加速的?

- **就近访问**：CDN 在全球范围内部署了多个服务器节点，当用户请求访问一个网站时，CDN 会选择距离用户最近的节点来提供内容。这减少了数据传输的距离和时间，从而降低了延迟。
- **内容缓存**：CDN 节点会缓存静态资源，如图片、样式表、脚本等。当用户请求访问这些资源时，CDN 可以直接从缓存中返回，避免了从源服务器获取资源的延迟。
- **前置缓存**：CDN 可以根据网站的配置，提前将热门的内容缓存在节点中，以备用户请求时快速响应。
- **智能负载均衡**：CDN 会根据服务器的负载和可用性，动态地将请求分发到合适的服务器节点上，确保资源的快速获取。
- **压缩技术**：CDN 使用压缩技术对内容进行压缩，减少传输数据的大小，从而加快内容的传输速度。
- **并行下载**：由于 CDN 支持多路复用，用户可以在同一个连接上同时下载多个资源，从而提高并行下载的效率。

Cookie是什和Session是什么?

Cookie 和 Session 都用于管理用户的状态和身份, Cookie 通过在客户端记录信息确定用户身份，Session 通过在服务器端记录信息确定用户身份。

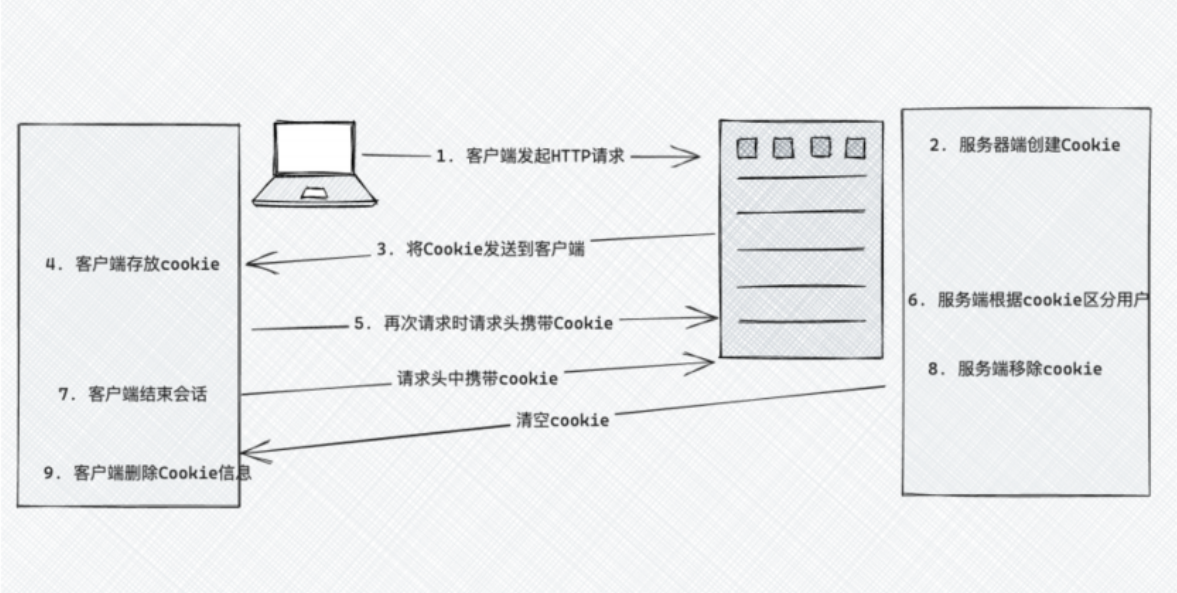
Cookie

- Cookie 是存储在用户浏览器中的小型文本文件，用于在用户和服务器之间传递数据。通常，服务器会将一个或多个 Cookie 发送到用户浏览器，然后浏览器将这些 Cookie 存储在本地。
- 服务器在接收到来自客户端浏览器的请求之后，就能够通过分析存放于请求头的Cookie得到客户端特有的信息，从而动态生成与该客户端相对应的内容。

Session

客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是 Session 。Session 主要用于维护用户登录状态、存储用户的临时数据和上下文信息等。

Cookie的工作原理是什么?



- 客户端（浏览器）第一次发送请求到服务器
- 服务器可以通过 HTTP 响应的头部信息中的 "Set-Cookie" 标头来创建一个 Cookie。这个标头指定了Cookie 的名称、值和其他参数，如过期时间、域名、路径等。
- 浏览器接收到 "Set-Cookie" 标头后，会将 Cookie 存储在用户的本地计算机上。
- 当用户再次访问同一个网站时，浏览器会将与该域名相关的 Cookie 信息包括在 HTTP 请求的头部中。
- 服务器收到包含 Cookie 的 HTTP 请求后，可以读取 Cookie 的值，根据其中的数据来判断用户的状态、偏好等，并根据需要做出相应的响应，如根据用户登录状态提供个性化内容。
- 服务器可以通过发送新的 "Set-Cookie" 标头来更新 Cookie 的值或设置新的参数。
- Cookie 可以设置过期时间，可以是会话级的（浏览器关闭时失效）或持久性的（在一段时间后失效）。当过期时间到达后，浏览器不再发送该 Cookie。

Session的工作原理

Session 是一种在服务器端存储和管理用户状态和数据的机制，通常基于会话标识符（Session ID）进行操作。

- 用户首次访问网站时，服务器会为该用户创建一个唯一的会话标识符（Session ID）。这个标识符可以是一个加密的字符串，通常以 Cookie 的形式存储在用户的浏览器中。
- 每个会话标识符对应着服务器上的一个会话存储空间。这个存储空间用于存储该用户在会话期间的状态和数据。
- 当用户与服务器交互时，服务器可以通过会话标识符来访问对应的会话存储空间。服务器可以将数据存储在会话中，如用户的登录状态、购物车内容、用户偏好等。
- 服务器为每个会话设置一个超时时间，如果用户在一段时间内没有活动，会话会自动过期。一旦会话过期，会话数据将被清除。
- 用户可以手动终止会话，例如通过退出登录操作。这会导致服务器删除与该用户相关的会话数据。

Cookie和Session有什么区别?

- 存储位置：Cookie 数据存储在用户的浏览器中，而 Session 数据存储在服务器上。
- 数据容量：Cookie 存储容量较小，一般为几 KB。Session 存储容量较大，通常没有固定限制，取决于服务器的配置和资源。
- 安全性：由于 Cookie 存储在用户浏览器中，因此可以被用户读取和篡改。相比之下，Session 数据存储在服务器上，更难被用户访问和修改。
- 传输方式：Cookie 在每次 HTTP 请求中都会被自动发送到服务器，而 Session ID 通常通过 Cookie 或 URL 参数传递。

TCP重传机制是怎么实现的?

当发送方的数据在传输过程中丢失、损坏或延迟，接收方可以请求发送方重新传输这些数据。

- **序号与确认号**：在 TCP 通信中，每个发送的字节都有一个唯一的序号，而每个接收的字节都有一个确认号。发送方维护了一个发送窗口，接收方维护了一个接收窗口。发送方会持续发送数据，并等待接收方的确认。
- **超时检测**：发送方为每个发送的数据段设置一个定时器，这个定时器的时长称为超时时间。发送方假设在这个超时时间内，数据能够到达接收方并得到确认。如果在超时时间内没有收到确认，发送方会认为数据丢失或损坏，触发重传。
- **重传策略**：当发送方在超时时间内没有收到确认，它会认为数据丢失，然后重新发送相应的数据段。如果只有一个数据段丢失，发送方只会重传丢失的数据段。如果有多个数据段丢失，发送方可能会使用更复杂的算法来决定哪些数据需要重传。
- **快速重传和快速恢复**：为了更快地发现丢失的数据，接收方可以使用快速重传策略。当接收方连续接收到相同的确认号时，它会立即向发送方发送冗余的确认，以触发发送方进行重传。此外，发送方在接收到快速重传的确认后，不需要等到超时再次发送，而是可以使用快速恢复算法继续发送未丢失的数据。

TCP流量控制是怎么实现的?

流量控制就是让发送方发送速率不要过快，让接收方来得及接收。利用**滑动窗口机制**就可以实施流量控制，主要方法就是动态调整发送方和接收方之间数据传输速率。

- **滑动窗口大小：** 在TCP通信中，每个TCP报文段都包含一个窗口字段，该字段指示发送方可以发送多少字节的数据而不等待确认。这个窗口大小是动态调整的。
- **接收方窗口大小：** 接收方通过TCP报文中的窗口字段告诉发送方自己当前的可接收窗口大小。这是接收方缓冲区中还有多少可用空间。
- **流量控制的目标：** 流量控制的目标是确保发送方不要发送超过接收方缓冲区容量的数据。如果接收方的缓冲区快满了，它会减小窗口大小，通知发送方暂停发送，以防止溢出。
- **动态调整：** 发送方会根据接收方的窗口大小动态调整发送数据的速率。如果接收方的窗口大小增加，发送方可以加速发送数据。如果窗口大小减小，发送方将减缓发送数据的速率。
- **确认机制：** 接收方会定期发送确认（ACK）报文，告知发送方已成功接收数据。这也与流量控制密切相关，因为接收方可以通过ACK报文中的窗口字段来通知发送方它的当前窗口大小。

TCP拥塞控制是怎么实现的?

TCP拥塞控制可以在网络出现拥塞时动态地调整数据传输的速率，以防止网络过载。

- **慢启动**：初始阶段，TCP发送方会以较小的发送窗口开始传输数据。随着每次成功收到确认的数据，发送方逐渐增加发送窗口的大小，实现指数级的增长，这称为慢启动。这有助于在网络刚开始传输时谨慎地逐步增加速率，以避免引发拥塞。
- **拥塞避免**：一旦达到一定的阈值（通常是慢启动阈值），TCP发送方就会进入拥塞避免阶段。在拥塞避免阶段，发送方以线性增加的方式增加发送窗口的大小，而不再是指数级的增长。这有助于控

制发送速率，以避免引起网络拥塞。

- **快速重传**：如果发送方连续收到相同的确认，它会认为发生了数据包的丢失，并会快速重传未确认的数据包，而不必等待超时。这有助于更快地恢复由于拥塞引起的数据包丢失。
- **快速恢复**：在发生快速重传后，TCP进入快速恢复阶段。在这个阶段，发送方不会回到慢启动阶段，而是将慢启动阈值设置为当前窗口的一半，并将拥塞窗口大小设置为慢启动阈值加上已确认但未被快速重传的数据块的数量。这有助于更快地从拥塞中恢复。

什么是SSO

普通登录认证机制在登录认证成功后，服务器把用户的登录信息写入 session，并为该用户生成一个 cookie，返回并写入浏览器；当用户再次访问这个系统的时候，请求中会带上这个 cookie，服务端会根据这个 cookie 找到对应的 session，通过session来判断这个用户是否已经登录。

普通的登录认证机制在多系统的环境下，在操作不同的系统时，需要多次登录，会变得很不方便。

单点登录（SSO）允许用户在一次登录后访问多个关联的应用程序或服务，而无需再次输入其凭据。简而言之，用户只需一次登录，就能够无缝地访问多个应用。

优点

- 降低密码重用风险： 因为用户只需一个凭据，不再需要在多个应用程序间重复使用相同的密码，从而减少了因密码泄露而引发的安全风险。
- 方便的用户体验： 用户只需要在一次登录过程中提供凭据，然后就可以无缝地访问多个应用程序，无需为每个应用程序输入用户名和密码。这大大简化了用户的登录过程，提供了更流畅的体验。
- 简化应用开发： 对于应用程序开发人员来说，他们可以将认证和授权的责任交给SSO系统，从而减少了在每个应用中实现这些功能的工作量。

HTTP的请求过程是怎么样的

- 首先，我们在浏览器地址栏中，输入要查找页面的URL，按下Enter
- 浏览器依次在 浏览器缓存 -->>系统缓存 -->>路由器缓存中去寻找匹配的URL，若有，就会直接在屏幕中显示出页面内容。若没有，则跳到第三步操作
- 发送HTTP请求前，浏览器需要先进行域名解析(即DNS解析)，以获取相应的IP地址；(浏览器DNS缓存、路由器缓存、DNS缓存)
- 获取到IP地址之后，浏览器向服务器发起TCP连接，与浏览器建立TCP三次握手
- 握手成功之后，浏览器就会向服务器发送HTTP请求，来请求服务器端的数据包
- 服务器处理从浏览器端收到的请求，接着将数据返回给浏览器
- 浏览器收到HTTP响应
- 查询状态，状态成功则进行下一步，不成功则弹出相应指示
- 再读取页面内容、进行浏览器渲染、解析HTML源码；(生成DOM树、解析CCS样式、处理JS交互，客户端和服务器交互）进行展示
- 关闭TCP连接（四次挥手）

说一说SYN攻击

原理

攻击者伪造不同IP地址的SYN报文请求连接，服务端收到连接请求后分配资源，回复ACK+SYN包，但是由于IP地址是伪造的，无法收到回应，久而久之造成服务端半连接队列被占满，无法正常工作。

避免方式

- 修改半连接队列大小：使服务端能够容纳更多半连接。此外还可以修改服务端超时重传次数，使服务端尽早丢弃无用连接

- 正常服务端行为是收到客户端SYN报文后，将其加入到内核半连接队列，接着发送ACK+SYN报文给客户端，当收到客户端ACK报文后把连接从半连接队列移动到accept队列。当半连接队列满时，启动syn cookie,后续连接不进入半连接队列，而是计算一个cookie值，作为请求报文序列号发送给客户端，如果服务端收到客户端确认报文，会检查ack包合法性，如果合法直接加入到accept队列

TCP保活机制

在一个定义的时间段内TCP连接无任何活动时，会启动TCP保活机制，每隔一定时间间隔发送一个探测报文，等待响应。

机制

- 对端正常响应，重置保活时间;
- 对端程序崩溃，响应一个RTS报文，将TCP连接重置;
- 保活报文不可达，等待达到保活探测次数后关闭连接。

TCP为啥需要流量控制

- 由于通讯双方网速不同，通讯方任意一方发送过快都会导致对方详细处理不过来，所以就需要把数据放到缓冲区中
- 如果缓冲区满了，发送方还在疯狂发送，那接收方只能把数据包丢弃。因此我们需要控制发送速率
- 我们缓冲区剩余大小称之为接收窗口，用变量win表示。如果win=0，则发送方停止发送

操作系统

什么是中断和异常

中断和异常都会导致处理器暂停当前正在执行的任务，并转向执行一个特定的处理程序。然后在处理完这些特殊情况后，处理器会返回到被打断的任务继续执行。

- 中断是由计算机系统外部事件触发的，通常与硬件设备相关。中断的目的是为了及时响应重要事件而暂时中断正常的程序执行。典型的中断包括时钟中断、I/O设备中断（如键盘输入、鼠标事件）和硬件错误中断等。
- 异常是由计算机系统内部事件触发的，通常与正在执行的程序或指令有关，比如程序的非法操作码、地址越界、运算溢出等错误引起的事件，异常不能被屏蔽，当出现异常时，计算机系统会暂停正常的执行流程，并转到异常处理程序来处理该异常。

说一说用户态和核心态

用户态和核心态是操作系统为了保护系统资源和实现权限控制而设计的两种不同的CPU运行级别，可以控制进程和程序对计算机硬件资源的访问权限和操作范围。

- 用户态：在用户态下，进程或程序只能访问受限的资源和执行受限的指令集，不能直接访问操作系统的核心部分，也不能直接访问硬件资源。
- 核心态：核心态是操作系统的特权级别，允许进程或程序执行特权指令和访问操作系统的核心部分。在核心态下，进程可以直接访问硬件资源，执行系统调用，管理内存、文件系统等操作。

什么场景下内核态和用户态切换

- 系统调用：当用户程序需要请求操作系统提供的服务时，会通过系统调用进入内核态。
- 异常：当程序执行过程中出现错误或异常情况时，CPU会自动切换到内核态，以便操作系统能够处理这些异常。
- 中断：外部设备（如键盘、鼠标、磁盘等）产生的中断信号会使CPU从用户态切换到内核态。操作系统会处理这些中断，执行相应的中断处理程序，然后再将CPU切换回用户态。

什么是并行和并发

- 并行是指在同一时刻执行多个任务，这些任务可以同时进行。
- 并发是指在相同的时间段内执行多个任务，这些任务可能不是同时发生的，而是交替执行。

什么是内部碎片和外部碎片

- 内部碎片是指已经被分配出去的空间中，由于分配的大小比需要的大小大了一些，因此分配出去后剩下的部分不能被使用，直被浪费。这种碎片是在已被分配的内存块之中发生的，因此称为内部碎片
- 外部碎片则是由已分配内存块与空闲内存块交替形成的未分配内存空间中的碎片。当需要分配一块内存时，系统会在未分配的空闲内存块中寻找符合要求的内存块，但可能由于这些空闲内存块被分割成多个小块，而分配不出连续空间大小符合要求的大块空间。这样的空间碎片称为外部碎片。

说一说僵尸进程和孤儿进程

- 孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。
- 僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。

信号和信号量有什么区别

- 信号：一种处理异步事件的方式。信号是比较复杂的通信方式，用于通知接收进程有某种事件发生，除了用于进程外，还可以发送信号给进程本身。
- 信号量：进程间通信处理同步互斥的机制。是在多线程环境下使用的一种设施，它负责协调各个线程，以保证它们能够正确，合理的使用公共资源。

说一说局部性原理

在一段时间内，程序倾向于多次访问相同的数据或接近的数据，而不是随机地访问内存中的各个位置。

- 时间局部性
    - 如果一个数据被访问，那么在不久的将来它很可能会再次被访问。这意味着程序在短时间内倾向于反复使用相同的数据项，例如在循环中反复访问数组的元素。
    - 通过利用时间局部性，程序可以将频繁使用的数据存储在缓存中，从而减少访问主内存的次数，提高程序的执行速度。
- 空间局部性
    - 如果一个数据被访问，那么它附近的数据也很可能会被访问。这意味着程序在访问一个数据时，通常会在接近该数据的附近访问其他数据，例如遍历数组时，往往会访问相邻的元素。
    - 文件系统在磁盘上存储数据时，通常会将相关的数据块放在相邻的磁盘扇区上，以便在访问一个数据块时能够快速地访问相邻的数据块。

说一说进程和线程，有什么区别呢

进程是系统进行资源分配和调度的基本单位

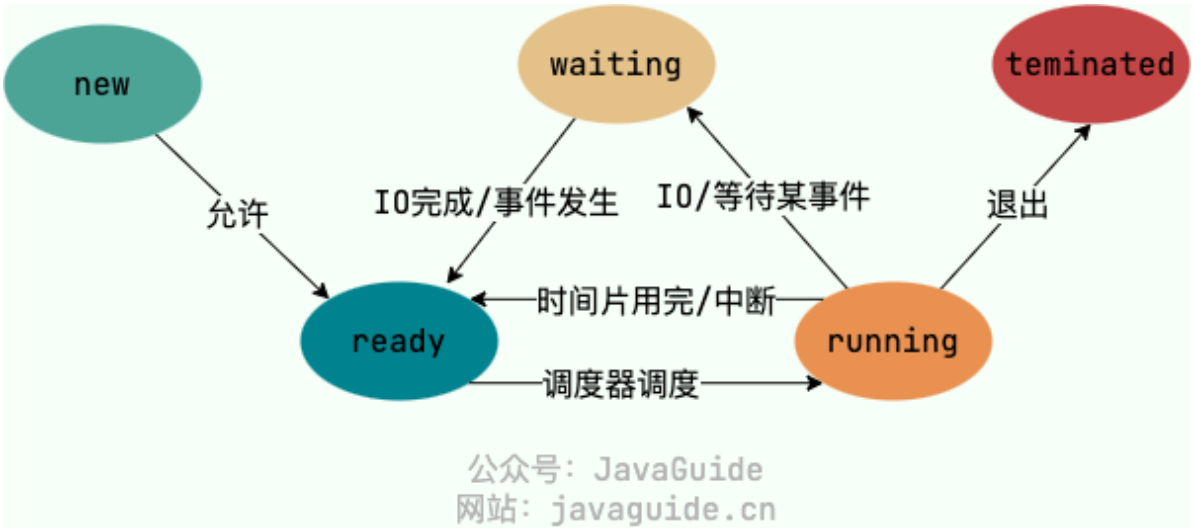
线程是操作系统能够进行运算的最小单位，线程是进程的子任务，是进程内的执行单元。一个进程至少有一个线程，一个进程可以运行多个线程，这些线程共享同一块内存。

区别

- **资源开销**

    - 由于每个进程都有独立的内存空间，创建和销毁进程的开销较大。进程间切换需要保存和恢复整个进程的状态，因此上下文切换的开销较高。
    - 线程共享相同的内存空间，创建和销毁线程的开销较小。线程间切换只需要保存和恢复少量的线程上下文，因此上下文切换的开销较小。
- **通信与同步**
    - 由于进程间相互隔离，进程之间的通信需要使用一些特殊机制，如管道、消息队列、共享内存等。
    - 由于线程共享相同的内存空间，它们之间可以直接访问共享数据，线程间通信更加方便。
- **安全性**
    - 由于进程间相互隔离，一个进程的崩溃不会直接影响其他进程的稳定性。
    - 由于线程共享相同的内存空间，一个线程的错误可能会影响整个进程的稳定性。

进程的状态有哪些



- **创建状态**：进程在创建时需要申请一个空白PCB，向其中填写控制和管理进程的信息，完成资源分配。如果创建工作无法完成，比如资源无法满足，就无法被调度运行，把此时进程所处状态称为创建状态。
- **就绪状态**：进程已经准备好，已分配到所需资源，只要分配到CPU就能够立即运行。
- **执行状态**：进程处于就绪状态被调度后，进程进入执行状态。
- **阻塞状态**：正在执行的进程由于某些事件（I/O请求，申请缓存区失败）而暂时无法运行，进程受到阻塞。在满足请求时进入就绪状态等待系统调用。
- **终止状态**：进程结束，或出现错误，或被系统终止，进入终止状态。无法再执行

进程之间的通信方式有哪些

- **管道**：是一种**半双工**的通信方式，数据只能单向流动而且只能在具有父子进程关系的进程间使用。
- 命名管道： 也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
- **信号量**：是一个计数器，可以用来控制多个进程对共享资源的访问，常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此主要作为进程间以及同一进程内不同线程之间的同步手段。
- **消息队列**：消息队列是消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- **信号**：用于通知接收进程某个事件已经发生，从而迫使进程执行信号处理程序。
- **共享内存**：就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的进程通信方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，比如信号量配合使用，来实现进程间的同步和通信。

- **Socket 套接字**：是支持TCP/IP 的网络通信的基本操作单元，主要用于在客户端和服务器之间通过网络进行通信。

线程之间的同步方式

线程同步机制是指在多线程编程中，为了保证线程之间的互不干扰，而采用的一种机制。

- 互斥锁：互斥锁是最常见的线程同步机制。它允许只有一个线程同时访问被保护的临界区（共享资源）
- 条件变量：条件变量用于线程间通信，允许一个线程等待某个条件满足，而其他线程可以发出信号通知等待线程。通常与互斥锁一起使用。
- 读写锁： 读写锁允许多个线程同时读取共享资源，但只允许一个线程写入资源。
- 信号量：用于控制多个线程对共享资源进行访问的工具。

介绍一下你知道的锁

两个基础的锁

- 互斥锁：互斥锁是一种最常见的锁类型，用于实现互斥访问共享资源。在任何时刻，只有一个线程可以持有互斥锁，其他线程必须等待直到锁被释放。这确保了同一时间只有一个线程能够访问被保护的资源。
- 自旋锁：自旋锁是一种基于忙等待的锁，即线程在尝试获取锁时会不断轮询，直到锁被释放。

其他的锁都是基于这两个锁的

- 读写锁：允许多个线程同时读共享资源，只允许一个线程进行写操作。分为读（共享）和写（排他）两种状态。
- 悲观锁：认为多线程同时修改共享资源的概率比较高，所以访问共享资源时候要上锁。
- 乐观锁：先不管，修改了共享资源再说，如果出现同时修改的情况，再放弃本次操作。

什么情况下会产生死锁

死锁是指**两个或多个进程在争夺系统资源时，由于互相等待对方释放资源而无法继续执行的状态。**

满足四个条件

- 互斥条件：一个进程占用了某个资源时，其他进程无法同时占用该资源。
- 请求保持条件：一个线程因为请求资源而阻塞的时候，不会释放自己的资源。
- 不可剥夺条件：资源不能被强制性地从一个进程中剥夺，只能由持有者自愿释放。
- 环路等待条件：多个进程之间形成一个循环等待资源的链，每个进程都在等待下一个进程所占有的资源。

如何解除死锁

- 破坏请求与保持条件：一次性申请所有的资源。
- 破坏不可剥夺条件：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
- 破坏循环等待条件：靠**按序申请资源**来预防。让所有进程按照相同的顺序请求资源，释放资源则反序释放。

说一说I/O多路复用

I/O多路复用通常通过select、poll、epoll等系统调用来实现。

I/O多路复用允许在一个线程中处理多个I/O操作，避免了创建多个线程或进程的开销，允许在一个线程中处理多个I/O操作，避免了创建多个线程或进程的开销。

- **select**：select是一个最古老的I/O多路复用机制，它可以监视多个文件描述符的可读、可写和错误状态。然而，但是它的效率可能随着监视的文件描述符数量的增加而降低。
- **poll**：poll是select的一种改进，它使用**轮询方式**来检查多个文件描述符的状态，避免了select中文件描述符数量有限的问题。但对于大量的文件描述符，poll的性能也可能变得不足够高效。
- **epoll**： epoll是Linux特有的I/O多路复用机制，相较于select和poll，它在处理大量文件描述符时更加高效。epoll使用事件通知的方式，只有在文件描述符就绪时才会通知应用程序，而不需要应用程序轮询。