C++方向

计算机网络

TCP/IP网络模型/ OSI模型

OSI模型， 是国际标准化组织（ISO）制定的一个用于计算机或通信系统间互联的标准体系，将计算机网络通信划分为七个不同的层级，每个层级都负责特定的功能。每个层级都构建在其下方的层级之上，并为上方的层级提供服务。七层从下到上分别是物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。虽然OSI模型在理论上更全面，但在实际网络通信中，TCP/IP模型更为实用。TCP/IP模型分为四个层级，每个层级负责特定的网络功能。

1. 应用层：该层与OSI模型的应用层和表示层以及会话层类似，提供直接与用户应用程序交互的接口。它为网络上的各种应用程序提供服务，如电子邮件（SMTP）、网页浏览（HTTP）、文件传输（FTP）等。
2. 传输层：该层对应OSI模型的传输层。它负责端到端的数据传输，提供可靠的、无连接的数据传输服务。主要的传输层协议有TCP和UDP。TCP提供可靠的数据传输，确保数据的正确性和完整性；而UDP则是无连接的，适用于不要求可靠性的传输，如实时音频和视频流。
3. 网际层：该层对应OSI模型的网络层。主要协议是IP，它负责数据包的路由和转发，选择最佳路径将数据从源主机传输到目标主机。IP协议使用IP地址来标识主机和网络，并进行逻辑地址寻址。
4. 网络接口层：该层对应OSI模型的数据链路层和物理层。它负责物理传输媒介的传输，例如以太网、Wi-Fi等，并提供错误检测和纠正的功能。此外，网络接口层还包含硬件地址（MAC地址）的管理。

HTTP有哪些请求方式？GET请求和POST请求的区别

HTTP请求方式：

1. **GET**：请求指定的资源。
2. **POST**：向指定资源提交数据进行处理请求（例如表单提交）。
3. **PUT**：更新指定资源。
4. **DELETE**：删除指定资源。
5. **HEAD**：获取报文首部，不返回报文主体。
6. **OPTIONS**：查询服务器支持的请求方法。
7. **PATCH**：对资源进行部分更新。

GET请求和POST请求的区别：

1. **用途**：GET请求通常用于获取数据，POST请求用于提交数据。
2. **数据传输**：GET请求将参数附加在URL之后，POST请求将数据放在请求体中。
3. **安全性**：GET请求由于参数暴露在URL中，安全性较低；POST请求参数不会暴露在URL中，相对更安全。
4. **数据大小**：GET请求受到URL长度限制，数据量有限；POST请求理论上没有大小限制。
5. **幂等性**：GET请求是幂等的，即多次执行相同的GET请求，资源的状态不会改变；POST请求不是幂等的，因为每次提交都可能改变资源状态。
6. **缓存**：GET请求可以被缓存，POST请求默认不会被缓存。

什么是强缓存和协商缓存

强缓存和协商缓存是HTTP缓存机制的两种类型，它们用于减少服务器的负担和提高网页加载速度。

1. **强缓存**：客户端在没有向服务器发送请求的情况下，直接从本地缓存中获取资源。强缓存通过HTTP响应头中的 `Cache-Control` 字段实现，如 `max-age` ，告诉浏览器在指定时间内可以直接使用缓存数据，无需再次请求。

2. **协商缓存**：当强缓存失效时，浏览器会发送请求到服务器，通过 `ETag` 或 `Last-Modified` 等HTTP响应头与服务器进行验证，以确定资源是否被修改。如果资源未修改，服务器返回 `304 Not Modified` 状态码，告知浏览器使用本地缓存；如果资源已修改，则返回新的资源，浏览器更新本地缓存。这种方式需要与服务器通信，但可以确保用户总是获取最新的内容。

强缓存和协商缓存是HTTP缓存机制的两种类型，它们用于减少服务器的负担和提高网页加载速度。

1. **强缓存**：客户端在没有向服务器发送请求的情况下，直接从本地缓存中获取资源。

- `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。但是因为 `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。这里有一个巨大的漏洞：“如果我本地时间不准咋办？”所以目前已经被废弃了。
- `Cache-Control强缓存`：目前使用的强缓存是通过HTTP响应头中的 `Cache-Control` 字段实现，通过 `max-age` 来告诉浏览器在指定时间内可以直接使用缓存数据，无需再次请求。

1. **协商缓存**：当强缓存失效时，浏览器会发送请求到服务器，通过 `ETag` 或 `Last-Modified` 等HTTP响应头与服务器进行验证，以确定资源是否被修改。如果资源未修改，服务器返回 `304 Not Modified` 状态码，告知浏览器使用本地缓存；如果资源已修改，则返回新的资源，浏览器更新本地缓存。这种方式需要与服务器通信，但可以确保用户总是获取最新的内容。

- 基于 `Last-Modified` 的协商缓存
  - `Last-Modified` 是资源的最后修改时间，服务器在响应头部中返回。
  - 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。
  - 服务器比较请求中的 `If-Modified-Since` 值与当前资源的 `Last-Modified` 值，如果比对的结果是没有变化，表示资源未发生变化，返回状态码 `304 Not Modified`。如果比对的结果说资源已经更新了，就会给浏览器正常返回资源，返回200状态。

但是这样的协商缓存有两个缺点：

  - 因为是更改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）。
    - `ETag` 是服务器为资源生成的唯一标识符（文件指纹），可以是根据文件内容计算出的哈希值，服务端将其和资源一起放回给客户端。
    - 客户端在请求头部的 `If-None-Match` 字段中携带上次响应的 `ETag` 值。
    - 服务器比较请求中的 `If-None-Match` 值与当前资源的 `ETag` 值，如果匹配，表示资源未发生变化，返回状态码 `304 Not Modified`。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

HTTP1.0和HTTP1.1的区别

1. **持久连接**：`HTTP/1.1` 默认支持持久连接，允许在一个TCP连接上发送多个HTTP请求和响应，减少了连接建立和关闭的开销。而 `HTTP/1.0` 默认为短连接，每次请求都需要建立一个TCP连接，并通过 `Connection: keep-alive` 头来实现持久连接。
2. **管道化**：`HTTP/1.1` 支持管道化，允许客户端在第一个请求的响应到达之前发送多个请求，这可以减少等待时间，提高效率。HTTP/1.0不支持管道化。
3. **缓存控制**：`HTTP1.0` 主要使用 `If-Modified-Since/Expires` 来做为缓存判断的标准，而 `HTTP1.1` 则引入了更多的缓存控制策略例如 `Etag / If-None-Match` 等更多可供选择的缓存头来控制缓存策略。
4. 错误处理：`HTTP/1.1` 增加了一些新的HTTP状态码，如 `100 Continue`，用于增强错误处理和请求的中间响应。
5. `Host` 头：`HTTP/1.1` 引入了 `Host` 头，允许客户端指定请求的主机名，这使得在同一台服务器上托管多个域名成为可能。HTTP/1.0没有这个头字段。
6. 带宽优化：`HTTP1.0` 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能， 而 `HTTP1.1` 则在请求头引入了 `range` 头域，它允许只请求资源的某个部分，即返回码是 `206（Partial Content）`

HTTP2.0与HTTP1.1的区别?

1. **二进制协议**：`HTTP/2.0` 采用二进制格式传输数据，而非 `HTTP/1.1` 的文本格式，使得解析更高效，减少了解析时间。
2. **多路复用**：`HTTP/2.0` 支持多路复用，允许在单个TCP连接上并行交错发送多个请求和响应，解决了 `HTTP/1.1` 中的**队头阻塞**问题。
3. **头部压缩**：`HTTP/2.0` 引入了 `HPACK` 压缩算法，对请求和响应的头部信息进行压缩，减少了冗余头部信息的传输，提高了传输效率。
4. **服务器推送**：`HTTP/2.0` 允许服务器主动推送资源给客户端，而不需要客户端明确请求，这可以减少页面加载时间。
5. **优先级和依赖**：`HTTP/2.0` 允许客户端为请求设置优先级，并表达请求之间的依赖关系，资源加载更加有序。

HTTPS和HTTP有哪些区别

两者的主要区别在于安全性和数据加密：

1. **加密层**：`HTTPS` 在 `HTTP` 的基础上增加了 `SSL/TLS` 协议作为加密层，确保数据传输的安全性。
2. **数据安全**：`HTTPS` 通过加密，保护数据在传输过程中不被窃听或篡改，而 `HTTP` 数据传输是明文的，容易受到攻击。
3. **端口**：`HTTPS` 通常使用端口 `443` ，而 `HTTP` 使用端口80。

HTTPS工作原理

`HTTPS` 主要基于 `SSL/TLS` 协议，确保了数据传输的安全性和完整性, 其建立连接并传输数据的过程如下：

1. **密钥交换**：客户端发起HTTPS请求后，服务器会发送其公钥证书给客户端。
2. **证书验证**：客户端会验证服务器的证书是否由受信任的证书颁发机构（ `CA` ）签发，并检查证书的有效性。
3. **加密通信**：一旦证书验证通过，客户端会生成一个随机的对称加密密钥，并使用服务器的公钥加密这个密钥，然后发送给服务器。

4. **建立安全连接**：服务器使用自己的私钥解密得到对称加密密钥，此时客户端和服务器都有了相同的密钥，可以进行加密和解密操作。
5. **数据传输**：使用对称加密密钥对所有传输的数据进行加密，确保数据在传输过程中的安全性。
6. **完整性校验**：SSL/TLS协议还包括消息完整性校验机制，如消息认证码，确保数据在传输过程中未被篡改。
7. **结束连接**：数据传输完成后，通信双方会进行会话密钥的销毁，以确保不会留下安全隐患。

TCP和UDP的区别

1. TCP是**面向连接**的协议，需要在数据传输前建立连接；UDP是无连接的，不需要建立连接。
2. TCP提供**可靠**的数据传输，保证数据包的顺序和完整性；UDP不保证数据包的顺序或完整性。
3. TCP具有**拥塞控制机制**，可以根据网络状况调整数据传输速率；UDP没有拥塞控制，发送速率通常固定。
4. TCP通过**滑动窗口机制**进行流量控制，避免接收方处理不过来；UDP没有流量控制。
5. TCP能够**检测并重传**丢失或损坏的数据包；UDP不提供错误恢复机制。
6. TCP有复杂的报文头部，包含序列号、确认号等信息；UDP的报文头部相对简单。
7. 由于TCP的连接建立、数据校验和重传机制，其性能开销通常比UDP大；UDP由于简单，性能开销小。
8. **适用场景：**TCP适用于需要可靠传输的应用，如网页浏览、文件传输等；UDP适用于对实时性要求高的应用，如语音通话、视频会议等。

三次握手的过程，为什么是三次

(1) 三次握手的过程

1. **第一次握手**：客户端向服务器发送一个 `SYN` （同步序列编号）报文，请求建立连接，客户端进入 `SYN_SENT` 状态。
2. **第二次握手**：服务器收到 `SYN` 报文后，如果同意建立连接，则会发送一个 `SYN-ACK` （同步确认）报文作为响应，同时进入 `SYN_RCVD` 状态。

3. **第三次握手**：客户端收到服务器的 `SYN-ACK` 报文后，会发送一个 `ACK` （确认）报文作为最终响应，之后客户端和服务器都进入 `ESTABLISHED` 状态，连接建立成功。

(2)为什么需要三次握手

通过三次握手，客户端和服务器都能够确认对方的接收和发送能力。第一次握手确认了客户端到服务器的通道是开放的；第二次握手确认了服务器到客户端的通道是开放的；第三次握手则确认了客户端接收到服务器的确认，从而确保了双方的通道都是可用的。

而如果仅使用两次握手，服务器可能无法确定客户端的接收能力是否正常，比如客户端可能已经关闭了连接，但之前发送的连接请求报文在网络上延迟到达了服务器，服务器就会主动去建立一个连接，但是客户端接收不到，导致资源的浪费。而四次握手可以优化为三次。

四次挥手的过程，为什么是四次

（1）四次挥手的过程

1. 第一次挥手：客户端发送一个 `FIN` 报文给服务端，表示自己要断开数据传送，报文中会指定一个序列号 `(seq=x)` 。然后,客户端进入 `FIN-WAIT-1` 状态。
2. 第二次挥手：服务端收到 `FIN` 报文后，回复 `ACK` 报文给客户端，且把客户端的序列号值 `+1` ，作为ACK `+1` 报文的序列号 `(seq=x+1)` 。然后，服务端进入 `CLOSE-WAIT``(seq=x+1)` 状态，客户端进入 `FIN-WAIT-2` 状态。
3. 第三次挥手：服务端也要断开连接时，发送 `FIN` 报文给客户端，且指定一个序列号 `(seq=y+1)` ，随后服务端进入 `LAST-ACK` 状态。
4. 第四次挥手：客户端收到 `FIN` 报文后，发出 `ACK` 报文进行应答，并把服务端的序列号值 `+1` 作为 `ACK` 报文序列号 `(seq=y+2)` 。此时客户端进入 `TIME-WAIT` 状态。服务端在收到客户端的 `ACK` 报文后进入 `CLOSE` 状态。如果客户端等待 `2MSL` 没有收到回复，才关闭连接。

（2）为什么需要四次挥手

`TCP` 是全双工通信，可以双向传输数据。任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。 当另一方也没有数据再发送的时候，则发出连接释放通知，对方确认后才会完全关闭 `TCP` 连接。因此两次握手可以释放一

端到另一端的 `TCP` 连接，完全释放连接一共需要四次握手。

只有通过四次挥手，才可以确保双方都能接收到对方的最后一个数据段的确认，主动关闭方在发送完最后一个 `ACK` 后进入 `TIME-WAIT` 状态，这是为了确保被动关闭方接收到最终的 `ACK` ，如果被动关闭方没有接收到，它可以重发 `FIN` 报文，主动关闭方可以再次发送 `ACK` 。

而如果使用三次挥手，被动关闭方可能在发送最后一个数据段后立即关闭连接，而主动关闭方可能还没有接收到这个数据段的确认。

TCP连接如何确保可靠性

TCP通过序列号、确认应答、超时重传、数据校验、流量控制、拥塞控制等机制，确保了数据传输的可靠性和效率。

1. **序列号**：每个TCP段都有一个序列号，确保数据包的顺序正确。
2. **确认应答**：接收方发送ACK确认收到的数据，如果发送方在一定时间内没有收到确认，会重新发送数据。
3. **超时重传**：发送方设置一个定时器，如果在定时器超时之前没有收到确认，发送方会重传数据。
4. **数据校验**：TCP使用校验和来检测数据在传输过程中是否出现错误，如果检测到错误，接收方会丢弃该数据包，并等待重传。
5. **流量控制**：TCP通过滑动窗口机制进行流量控制，确保接收方能够处理发送方的数据量。
6. **拥塞控制**：TCP通过算法如慢启动、拥塞避免、快重传和快恢复等，来控制数据的发送速率，防止网络拥塞。

拥塞控制的实现机制

1. **慢启动：** 开始时以较低的速率发送数据。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。
2. **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。
3. **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。

4. **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。

HTTP的Keep-Alive是什么？TCP 的 Keepalive 和 HTTP 的 Keep-Alive 是一个东西吗?

1. `HTTP` 的 `Keep-Alive`，是由应用层实现的，称为 HTTP 长连接

每次请求都要经历这样的过程：建立 `TCP` 连接 -> `HTTP` 请求资源 -> 响应资源 -> 释放连接，这就是HTTP短连接，但是这样每次建立连接都只能请求一次资源，所以 `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，就就是 **HTTP 长连接**。通过设置HTTP头 `Connection: keep-alive` 来实现。

1. `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

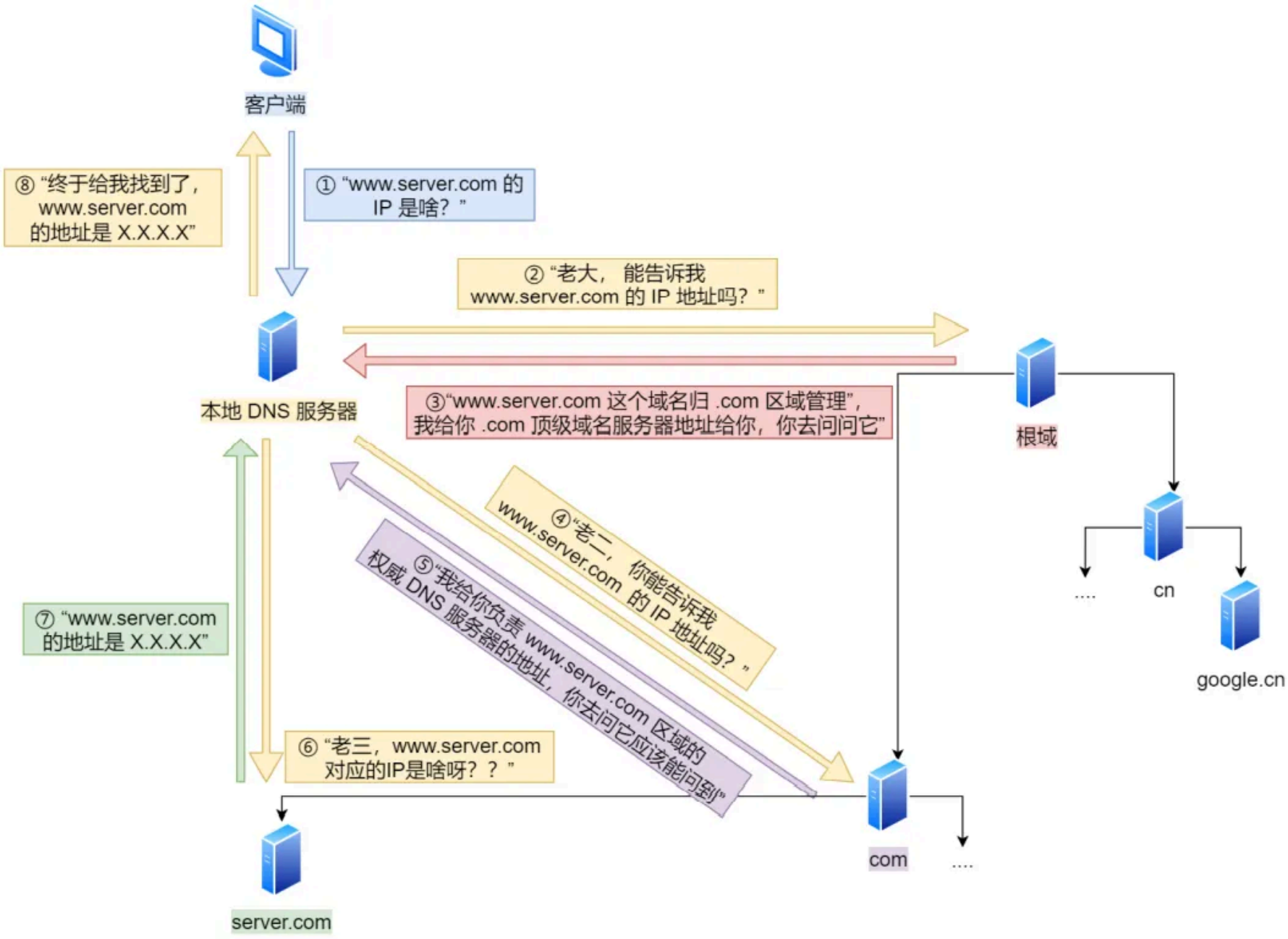
当 `TCP` 连接建立后，如果一段时间内没有任何数据传输， `TCP Keepalive` 会发送探测包来检查连接是否仍然有效。

DNS查询过程

`DNS` 用来将主机名和域名转换为IP地址, 其查询过程一般通过以下步骤：

1. **本地DNS缓存检查**：首先查询本地DNS缓存，如果缓存中有对应的IP地址，则直接返回结果。
2. 如果本地缓存中没有，则会向**本地的DNS服务器**（通常由你的互联网服务提供商（ISP）提供， 比如中国移动）发送一个DNS查询请求。
3. 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。
4. 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。

5. 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。
6. 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。
7. **浏览器发起连接：** 本地DNS解析器已经将IP地址返回给您的计算机，您的浏览器可以使用该IP地址与目标服务器建立连接，开始获取网页内容。



CDN是什么

30477字

CDN是一种分布式网络服务，通过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，从而加速互联网上的内容传输。

- **就近访问**：CDN 在全球范围内部署了多个服务器节点，用户的请求会被路由到距离最近的 CDN 节点，提供快速的内容访问。
- **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 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。通过缓存内容，减少了对原始服务器的请求，减轻了源站的负载。
- 可用性：即使某些节点出现问题，用户请求可以被重定向到其他健康的节点。

Cookie和Session是什么？有什么区别？

(1) Cookie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。

1. Cookie

- 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。

1. 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。服务器为每个用户分配一个唯一的 `Session ID` ，通常存储在 `Cookie` 中。

（2） Cookie和Session的区别？

- 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。
- 生命周期：`Cookie` 可以设置过期时间，`Session` 依赖于会话的持续时间或用户活动。
- 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。

操作系统

进程和线程的区别

进程是资源分配和调度的基本单位。

线程是程序执行的最小单位，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。

资源开销：

- 进程：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：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：

- 进程：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。
- 线程：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：

- 进程：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。
- 线程：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。

并行和并发有什么区别

- 并行是在同一时刻执行多个任务。
- 并发是在相同的时间段内执行多个任务，任务可能交替执行，通过调度实现。

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，每个任务都在不同的处理单元（如多个CPU核心）上执行。在并行系统中，多个处理单元可以同时处理独立的子任务，从而加速整体任务的完成。

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，通过时间片轮转或者事件驱动的方式。并发通常与任务之间的交替执行和任务调度有关。

解释一下用户态和核心态

1．用户态和内核态的区别

用户态和内核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**控制进程或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。

1．在什么场景下，会发生内核态和用户态的切换

- 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。

进程调度算法你了解多少

- 先来先服务：按照请求的顺序进行调度。 这种调度方式简单，但是能导致较长作业阻塞较短作业。
- 最短作业优先：非抢占式的调度算法，按估计运行时间最短的顺序进行调度。 但是如果一直有短作业到来，那么长作业永远得不到调度，造成长作业“饥饿”现象。
- 最短剩余时间优先：基于最短作业优先改进，按剩余运行时间的顺序进行调度。当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。
- 优先级调度：为每个进程分配一个优先级，按优先级进行调度。为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。
- 时间片轮转：为每个进程分配一个时间片，进程轮流执行，时间片用完后切换到下一个进程。
- 多级队列：时间片轮转调度算法和优先级调度算法的结合。 将进程分为不同的优先级队列，每个队列有自己的调度算法。

进程间有哪些通信方式

1. **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。
2. 命名管道： 类似管道，也是半双工的通信方式，但是它允许在不相关的进程间通信。
3. **消息队列**：允许进程发送和接收消息，而消息队列是消息的链表，可以设置消息优先级。
4. **信号**：用于发送通知到进程，告知其发生了某种事件或条件。
5. **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。

6. **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，
7. `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。
8. 互斥锁：一种信号量，用于保护共享数据结构，防止多个进程同时访问。
9. 条件变量：与互斥锁配合使用，用于进程间的同步，等待某些条件成立。

解释一下进程同步和互斥，以及如何实现进程同步和互斥

进程同步是指**多个并发执行的进程之间协调和管理它们的执行顺序，以确保它们按照一定的顺序或时间间隔执行。**

互斥指的是在某一时刻只允许一个进程访问某个共享资源。当一个进程正在使用共享资源时，其他进程不能同时访问该资源。

解决进程同步和互斥的问题有很多种方法，其中一种常见的方法是使用**信号量和 PV 操作**。信号量是一种特殊的变量，它表示系统中某种资源的数量或者状态。PV 操作是一种对信号量进行增加或者减少的操作，它们可以用来控制进程之间的同步或者互斥。

- **P操作**：相当于“检查”信号量，如果资源可用，就减少计数，然后使用资源。
- **V操作**：相当于“归还”资源，增加信号量的计数，并可能唤醒等待的进程。

除此之外，下面的方法也可以解决进程同步和互斥问题：

- **临界区**：将可能引发互斥问题的代码段称为临界区，里面包含了需要互斥访问的资源。进入这个区域前需要先获取锁，退出临界区后释放该锁。这确保同一时间只有一个进程可以进入临界区。
- **互斥锁（Mutex）**：互斥锁是一种同步机制，用于实现互斥。每个共享资源都关联一个互斥锁，进程在访问该资源前需要先获取互斥锁，使用完后释放锁。只有获得锁的进程才能访问共享资源。
- **条件变量**：条件变量用于在进程之间传递信息，以便它们在特定条件下等待或唤醒。通常与互斥锁一起使用，以确保等待和唤醒的操作在正确的时机执行。

什么是死锁，如何预防死锁?

死锁是系统中两个或多个进程在执行过程中，因争夺资源而造成的一种僵局。当每个进程都持有一定的资源并等待其他进程释放它们所需的资源时，如果这些资源都被其他进程占有且不释放，就导致了死锁。

死锁只有同时满足以下四个条件才会发生：

- 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。
- 循环等待条件：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。

避免死锁：**通过破坏死锁的四个必要条件之一来预防死锁。比如破坏循环等待条件，让所有进程按照相同的顺序请求资源。**检测死锁：通过检测系统中的资源分配情况来判断是否存在死锁。例如，可以使用资源分配图或银行家算法进行检测。解除死锁：一旦检测到死锁存在，可以采取一些措施来解除死锁。例如，可以通过抢占资源、终止某些进程或进行资源回收等方式来解除死锁。

讲一讲你理解的虚拟内存

虚拟内存是指在每一个进程创建加载的过程中，会分配一个连续虚拟地址空间，**它不是真实存在的，**而是**通过映射与实际物理地址空间**对应，这样就可以使每个进程看起来都有自己独立的连续地址空间，并允许程序访问比物理内存 `RAM` 更大的地址空间，每个程序都可以认为它拥有足够的内存来运行。

需要虚拟内存的原因：

- 内存扩展： 虚拟内存使得每个程序都可以使用比实际可用内存更多的内存，从而允许运行更大的程序或处理更多的数据。
- 内存隔离：虚拟内存还提供了进程之间的内存隔离。每个进程都有自己的虚拟地址空间，因此一个进程无法直接访问另一个进程的内存。
- 物理内存管理：虚拟内存允许操作系统动态地将数据和程序的部分加载到物理内存中，以满足当前正在运行的进程的需求。当物理内存不足时，操作系统可以将不常用的数据或程序暂时移到硬盘上，从而释放内存，以便其他进程使用。
- 页面交换：当物理内存不足时，操作系统可以将一部分数据从物理内存写入到硬盘的虚拟内存中，这个过程被称为页面交换。当需要时，数据可以再次从虚拟内存中加载到物理内存中。这样可以保证系统可以继续运行，尽管物理内存有限。

- 内存映射文件：虚拟内存还可以用于将文件映射到内存中，这使得文件的读取和写入可以像访问内存一样高效。

你知道的线程同步的方式有哪些?

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。常见的线程同步机制有以下几种：

1. 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）
2. 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。
3. 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。
4. 信号量：用于控制多个线程对共享资源进行访问的工具。

介绍一下几种典型的锁

- **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。

有哪些页面置换算法

常见页面置换算法有最佳置换算法（OPT）、先进先出（FIFO）、最近最久未使用算法（LRU）、时钟算法（Clock） 等。

1. 最近最久未使用算法 `LRU` ：LRU算法**基于页面的使用历史，通过选择最长时间未被使用的页面进行置换。**

2. 先进先出 `FIFO` 算法：也就是**最先进入内存的页面最先被置换出去。**
3. 最不经常使用 `LFU` ：淘汰访问次数最少的页面，考虑页面的访问频率。
4. 时钟算法 `CLOCK` ：Clock算法的核心思想是通过使用一个指针(称为时钟指针)在环形链表上遍历，检查页面是否被访问过，当需要进行页面置换时，Clock算法从时钟指针的位置开始遍历环形链表。 如果当前页面的访问位为0，表示该页面最久未被访问，可以选择进行置换。将访问位设置为1，继续遍历下一个页面。 如果当前页面的访问位为1，表示该页面最近被访问过，它仍然处于活跃状态。将访问位设置为0，并继续遍历下一个页面如果遍历过程中找到一个访问位为0的页面，那么选择该页面进行置换。
5. 最佳置换算法: 该**算法根据未来的页面访问情况，选择最长时间内不会被访问到的页面进行置换。**那么就有一个问题了，未来要访问什么页面，操作系统怎么知道的呢?操作系统当然不会知道，所以这种算法只是一种理想情况下的置换算法，通常是无法实现的。

select、poll、epoll的区别

I/O多路复用通常通过select、poll、epoll等系统调用来实现。

- **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。

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。

数据库

一条SQL查询语句是如何执行的?

1. 连接器:连接器负责跟客户端建立连接、获取权限、维持和管理连接。
2. 查询缓存: `MySQL` 拿到一个查询请求后，会先到查询缓存看看，之前是不是执行过这条语句。之前执行过的语句及其结果可能会以 `key-value` 对的形式，被直接缓存在内存中。
3. 分析器:你输入的是由多个字符串和空格组成的一条 `SQL` 语句， `MySQL` 需要识别出里面的字符串分别是什么，代表什么。
4. 优化器:优化器是在表里面有多个索引的时候，决定使用哪个索引; 或者在一个语句有多表关联(`join`)的时候，决定各个表的连接顺序。
5. 执行器: `MySQL` 通过分析器知道了你要做什么，通过优化器知道了该怎么做，于是就进入了执行器阶段，开始执行语句。

数据库的事务隔离级别有哪些?

1. **读未提交（Read Uncommitted）：**
   - 允许一个事务读取另一个事务尚未提交的数据修改。
   - 最低的隔离级别，存在脏读、不可重复读和幻读的问题。
2. **读已提交（Read Committed）：**
   - 一个事务只能读取已经提交的数据。其他事务的修改在该事务提交之后才可见。
   - 解决了脏读问题，但仍可能出现不可重复读和幻读。
3. **可重复读（Repeatable Read）：**
   - 事务执行期间，多次读取同一数据会得到相同的结果，即在事务开始和结束之间，其他事务对数据的修改不可见。
   - 解决了不可重复读问题，但仍可能出现幻读。
4. **序列化（Serializable）：**
   - 最高的隔离级别，确保事务之间的并发执行效果与串行执行的效果相同，即不会出现脏读、不可重复读和幻读。

事务的四大特性有哪些?

事务的四大特性通常被称为 `ACID` 特性

1. 原子性：确保事务的所有操作要么全部执行成功，要么全部失败回滚，不存在部分成功的情况。
2. 一致性：事务在执行前后，数据库从一个一致性状态转变到另一个一致性状态。
3. 隔离性：多个事务并发执行时，每个事务都应该被隔离开来，一个事务的执行不应该影响其他事务的执行。
4. 持久性：一旦事务被提交，它对数据库的改变就是永久性的，即使在系统故障或崩溃后也能够保持。

MySQL的执行引擎有哪些?

MySQL的执行引擎主要负责查询的执行和数据的存储, 其执行引擎主要有 `MyISAM` 、 `InnoDB` 、 `Memery` 等。

- `InnoDB` 引擎提供了对事务ACID的支持，还提供了行级锁和外键的约束，是目前MySQL的默认存储引擎，适用于需要事务和高并发的应用。
- `MyISAM` 引擎是早期的默认存储引擎，支持全文索引，但是不支持事务，也不支持行级锁和外键约束，适用于快速读取且数据量不大的场景。
- `Memery` 就是将数据放在内存中，访问速度快，但数据在数据库服务器重启后会丢失。

MySQL为什么使用B+树来作索引

B+树是一个B树的变种，提供了高效的数据检索、插入、删除和范围查询性能。

- 单点查询：B 树进行单个索引查询时，最快可以在 O(1) 的时间代价内就查到。从平均时间代价来看，会比 B+ 树稍快一些。但是 B 树的查询波动会比较大，因为每个节点既存索引又存记录，所以有时候访问到了非叶子节点就可以找到索引，而有时需要访问到叶子节点才能找到索引。**B+树的非叶子节点不存放实际的记录数据**，仅存放索引，所以数据量相同的情况下，相比存储即存索引又存记录的 B 树，B+树的非叶子节点可以存放更多的索引，因此 B+ 树可以比 B 树更「矮胖」，查询底层节点的磁盘 I/O次数会更少。

- 插入和删除效率：B+ 树有大量的冗余节点，删除一个节点的时候，可以直接从叶子节点中删除，甚至可以不动非叶子节点，删除非常快。B+ 树的插入也是一样，有冗余节点，插入可能存在节点的分裂（如果节点饱和），但是最多只涉及树的一条路径。B 树没有冗余节点，删除节点的时候非常复杂，可能涉及复杂的树的变形。
- 范围查询：B+ 树所有叶子节点间有一个链表进行连接，而 B 树没有将所有叶子节点用链表串联起来的结构，因此只能通过树的遍历来完成范围查询，这会涉及多个节点的磁盘 I/O 操作，范围查询效率不如 B+ 树。存在大量范围检索的场景，适合使用 B+树，比如数据库。而对于大量的单个索引查询的场景，可以考虑 B 树，比如nosql的MongoDB。

说一下索引失效的场景?

索引失效意味着查询操作不能有效利用索引进行数据检索，从而导致性能下降，下面一些场景会发生索引失效。

1. **使用OR条件**：当使用OR连接多个条件，并且每个条件用到不同的索引列时，索引可能不会被使用。
2. **使用非等值查询**：当使用 `!=` 或 `<>` 操作符时，索引可能不会被使用，特别是当非等值条件在WHERE子句的开始部分时。
3. 对列进行类型转换： 如果在查询中对列进行类型转换，例如将字符列转换为数字或日期，索引可能会失效。
4. **使用LIKE语句**：以通配符 `%` 开头的LIKE查询会导致索引失效。
5. **函数或表达式**：在列上使用函数或表达式作为查询条件，通常会导致索引失效。
6. 表连接中的列类型不匹配： 如果在连接操作中涉及的两个表的列类型不匹配，索引可能会失效。例如，一个表的列是整数，另一个表的列是字符，连接时可能会导致索引失效。

undo log、redo log、binlog 有什么用?

- `undo log` 是 `Innodb` 存储引擎层生成的日志，实现了事务中的原子性，主要用于事务回滚和 `MVCC` 。
- `redo log` 是物理日志，记录了某个数据页做了什么修改，每当执行一个事务就会产生一条或者多条物理日志。
- `binlog` (归档日志）是 `Server` 层生成的日志，主要用于数据备份和主从复制。

什么是慢查询？原因是什么？可以怎么优化?

数据库查询的执行时间超过指定的超时时间时，就被称为慢查询。

原因：

- 查询语句比较复杂：查询涉及多个表，包含复杂的连接和子查询，可能导致执行时间较长。
- 查询数据量大：当查询的数据量庞大时，即使查询本身并不复杂，也可能导致较长的执行时间。
- 缺少索引：如果查询的表没有合适的索引，需要遍历整张表才能找到结果，查询速度较慢。
- 数据库设计不合理：数据库表设计庞大，查询时可能需要较多时间。
- 并发冲突：当多个查询同时访问相同的资源时，可能发生并发冲突，导致查询变慢。
- 硬件资源不足：如果MySQL服务器上同时运行了太多的查询，会导致服务器负载过高，从而导致查询变慢

优化：

1. 运行语句，找到慢查询的sql
2. 查询区分度最高的字段
3. explain：显示mysql如何使用索引来处理select语句以及连接表，可以帮助选择更好的索引、写出更优化的查询语句
4. `order by limit` 形式的sql语句，让排序的表优先查
5. 考虑建立索引原则

Redis有什么优缺点？为什么用Redis查询会比较快

(1) Redis有什么优缺点?

`Redis` 是一个基于内存的数据库，**读写速度非常快，通常被用作缓存**、消息队列、分布式锁和键值存储数据库。它支持多种数据结构，如字符串、哈希表、列表、集合、有序集合等， Redis 还提供了**分布式**特性，可以将数据分布在多个节点上，以提高可扩展性和可用性。但是 `Redis` 受限于物理内存的大小，不适合存储超大量数据，并且需要大量内存，相比磁盘存储成本更高。

(2) **为什么Redis查询快**

- **基于内存操作**： 传统的磁盘文件操作相比减少了IO，提高了操作的速度。

- 高效的数据结构：Redis专门设计了STRING、LIST、HASH等高效的数据结构，依赖各种数据结构提升了读写的效率。
- 单线程：单线程操作省去了上下文切换带来的开销和CPU的消耗，同时不存在资源竞争，避免了死锁现象的发生。
- I/O多路复用：采用I/O多路复用机制同时监听多个Socket，根据Socket上的事件来选择对应的事件处理器进行处理。

Redis的数据类型有那些?

Redis 常见的五种数据类型：**String（字符串），Hash（哈希），List（列表），Set（集合）及 Zset(sorted set：有序集合)**。

1. 字符串 `STRING`：存储字符串数据，最基本的数据类型。
2. 哈希表 `HASH`：存储字段和值的映射，用于存储对象。
3. 列表 `LIST`：存储有序的字符串元素列表。
4. 集合 `SET`：存储唯一的字符串元素，无序。
5. 有序集合 `ZSET`：类似于集合，但每个元素都关联一个分数，可以按分数进行排序。

Redis版本更新，又增加了几种数据类型，

- `BitMap`：存储位的数据结构，可以用于处理一些位运算操作。
- `HyperLogLog`：用于基数估算的数据结构，用于统计元素的唯一数量。
- `GEO`： 存储地理位置信息的数据结构。
- `Stream`：专门为消息队列设计的数据类型。

Redis是单线程的还是多线程的，为什么?

`Redis` 在其传统的实现中是单线程的(网络请求模块使用单线程进行处理，其他模块仍用多个线程)，这意味着它使用单个线程来处理所有的客户端请求。这样的设计选择有几个关键原因：

1. **简化模型**：单线程模型简化了并发控制，避免了复杂的多线程同步问题。
2. **性能优化**：由于大多数操作是内存中的，单线程避免了线程间切换和锁竞争的开销。

3. **原子性保证**：单线程执行确保了操作的原子性，简化了事务和持久化的实现。
4. **顺序执行**：单线程保证了请求的顺序执行。

但是Redis的单线程模型并不意味着它在处理客户端请求时不高效。实际上，由于其操作主要在内存中进行，Redis能够提供极高的吞吐量和低延迟的响应。

此外，`Redis 6.0` 引入了多线程的功能，用来处理网络I/O这部分，充分利用CPU资源，减少网络I/O阻塞带来的性能损耗。

Redis持久化机制有哪些

- **AOF 日志**：每执行一条写操作命令，就把该命令以追加的方式写入到一个文件里；
- **RDB 快照**：将某一时刻的内存数据，以二进制的方式写入磁盘；
- **混合持久化方式**：Redis 4.0 新增的方式，集成了 AOF 和 RBD 的优点；

缓存雪崩、击穿、穿透和解决办法

1. 缓存雪崩是指在某个时间点，大量缓存同时失效，导致请求直接访问数据库或其他后端系统，增加了系统负载。

对于缓存雪崩，可以通过合理设置缓存的过期时间，分散缓存失效时间点，或者采用永不过期的策略，再结合定期更新缓存。

1. 缓存击穿是指一个缓存中不存在但是数据库中存在的数据，当有大量并发请求查询这个缓存不存在的数据时，导致请求直接访问数据库，增加数据库的负载。典型的场景是当一个缓存中的数据过期或被清理，而此时有大量请求访问这个缓存中不存在的数据，导致大量请求直接访问底层存储系统。

对于缓存击穿，可以采用**互斥锁（例如分布式锁）**或者在查询数据库前先检查缓存是否存在，如果不存在再允许查询数据库，并将查询结果写入缓存。

1. 缓存穿透是指查询一个在缓存和数据库都不存在的数据，这个数据始终无法被缓存，导致每次请求都直接访问数据库，增加数据库的负载。典型的情况是攻击者可能通过构造不存在的 key 大量访问缓存，导致对数据库的频繁查询。

对于缓存穿透，可以采用布隆过滤器等手段来过滤掉恶意请求，或者在查询数据库前先进行参数的合法性校验。

如何保证数据库和缓存的一致性

Cache Aside

- **原理**：先从缓存中读取数据，如果没有就再去数据库里面读数据，然后把数据放回缓存中，如果缓存中可以找到数据就直接返回数据；更新数据的时候先把数据持久化到数据库，然后再让缓存失效。
- **问题**：假如有两个操作一个更新一个查询，第一个操作先更新数据库，还没来及删除缓存，查询操作可能拿到的就是旧的数据；更新操作马上让缓存失效了，所以后续的查询可以保证数据的一致性；还有的问题就是有一个是读操作没有命中缓存，然后就到数据库中取数据，此时来了一个写操作，写完数据库后，让缓存失效，然后，之前的那个读操作再把老的数据放进去，也会造成脏数据。
- **可行性**：出现上述问题的概率其实非常低，需要同时达成读缓存时缓存失效并且有并发写的操作。数据库读写要比缓存慢得多，所以读操作在写操作之前进入数据库，并且在写操作之后更新，概率比较低。

Read/Write Through

- **原理**：Read/Write Through原理是把更新数据库（Repository）的操作由缓存代理，应用认为后端是一个单一的存储，而存储自己维护自己的缓存。
- **Read Through**：就是在查询操作中更新缓存，也就是说，当缓存失效的时候，Cache Aside策略是由调用方负责把数据加载入缓存，而Read Through则用缓存服务自己来加载，从而对调用方是透明的。
- **Write Through**：当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回。如果命中了缓存，则更新缓存，然后再由缓存自己更新数据库（这是一个同步操作）。

Write Behind

- **原理**：在更新数据的时候，只更新缓存，不更新数据库，而缓存会异步地批量更新数据库。这个设计的好处就是让数据的I/O操作非常快，带来的问题是，数据不是强一致性的，而且可能会丢。
- **第二步失效问题**：这种可能性极小，缓存删除只是标记一下无效的软删除，可以看作不耗时间。如果会出问题，一般程序在写数据库那里就没有完成：故意在写完数据库后，休眠很长时间再来删除缓存。

C++

静态变量和全局变量、局部变量的区别、在内存上是怎么分布的

1. **静态局部变量**
   - 特点：
     - 作用域：仅限于声明它们的函数或代码块内部。
     - 生命周期：静态局部变量在程序的整个运行期间都存在，只初始化一次（在第一次使用前）。
     - 初始化：在首次进入函数时初始化，并保持值直到程序结束。
   - 使用场景：
     - 当你需要一个仅在函数内部使用，但希望其值在函数调用之间保持不变的变量时。
     - 适用于需要缓存数据以提高性能的情况。
   - 内存分布：静态局部变量存储在**全局/静态存储区。**
1. **局部变量**
   - 特点：
     - 作用域：局部变量仅在声明它们的函数或代码块内部可见。
     - 生命周期：局部变量在函数调用时创建，函数调用结束后销毁。
     - 初始化：必须在使用前显式初始化。
   - 使用场景：
     - 需要临时存储数据，且这些数据只在当前作用域内使用时。
     - 作为循环计数器或中间计算结果。
   - 内存分布：**局部变量存储在栈上**，与它们所在的作用域（如函数）相关联。
1. 全局变量

- 特点：
  - 作用域：全局变量在整个程序中都是可见的，可以在任何函数或代码块中访问。
  - 生命周期：全局变量同样具有静态存储期，它们在程序的整个运行期间都存在。
  - 初始化：通常在程序启动时初始化。
- 使用场景：
  - 当你需要在程序的多个部分共享数据时。
  - 适用于存储配置信息或程序的状态信息。
  - 需要注意全局变量可能导致代码难以测试和维护。
- 内存分布：**全局变量也存储在全局/静态存储区。**

指针和引用的区别

1．从概念上来说：
  - 指针是一个存储另一个【变量地址】的变量，它指向内存中的一个位置。
  - 引用就是变量的别名，从一而终，不可变，必须初始化

2．空状态：
  - 指针可以被初始化为NULL或nullptr，表示它不指向任何地址。
  - 引用在定义时必须被初始化，不能引用NULL或不存在的内存地址。

3．可变性：
  - 指针： 可以改变指针的指向，使其指向不同的内存地址。
  - 引用： 一旦引用被初始化，它将一直引用同一个对象，不能改变绑定。

4．操作
  - 指针： 可以通过解引用操作符 `*` 来访问指针指向的变量的值，还可以通过地址运算符 `&` 获取变量的地址。

- 引用： 引用在声明时被初始化，并在整个生命周期中一直引用同一个变量。不需要使用解引用操作符，因为引用本身就是变量的别名。

5. 用途：
   - 指针： 通常用于动态内存分配、数组操作以及函数参数传递。
   - 引用： 通常用于函数参数传递、操作符重载以及创建别名。

static关键字和const关键字的作用

`static` 和 `const` 是 `C++` 中两个常用的关键字, 有以下作用：

1. `static` 关键字: 用于控制变量和函数的生命周期、作用域和访问权限。
   - 声明静态变量：静态变量的生命周期直到程序结束。当在函数内部声明静态变量时，即使函数执行完了也不会释放它，下次调用该函数时会保留上次的状态。
   - 在类中，被static声明的成员被称为静态成员。
     - 静态成员变量：在类中使用static关键字修饰的成员变量，表示该变量属于类而不是类的实例，所有实例共享同一份数据
     - 静态成员函数：在类内使用static关键字修饰的成员函数，所有对象共享同一个函数；静态成员函数只能访问静态成员变量；静态成员函数调用可以不需要通过创建类的实例，而是直接通过类名调用。
   - `static` 变量如果被多个线程访问，需要特别注意线程安全问题。
2. `const` : 关键字用于定义常量，即一旦初始化后其值不能被修改：
   - 常量变量：声明变量，使变量的值不能修改（只读）
   - 常量成员函数，表示该函数不会修改对象的成员变量
   - 常量指针：可以指向一个 `const` 类型的值，或者是一个指向 `const` 值的指针，表明指针指向的值不能通过这个指针被修改。
   - `const` 变量由于其不可变性，天然具有线程安全性。

3. 有时候 `static` 和 `const` 可以组合使用，如 `static const` 变量，表示一个静态的常量。

总结来说， `static` 关键字用于创建类的静态成员，而 `const` 关键字用于定义常量。

常量指针和指针常量之间有什么区别

常量指针"和 "指针常量"是两种不同的概念，它们的区别主要在于它们所指向的数据是否可以被修改，以及它们自己的值是否可以改变。

1. 常量指针是指**指针所指向的数据是常量**，不能通过这个指针来修改它指向的数据。但是，指针本身的值（即它所指向的地址）是可以改变的。
2. 指针常量是指**指针本身的值是常量**，一旦被初始化后就不能指向其他地址。但是，它所指向的数据是可以修改的（除非那个数据本身是常量）
3. 也可以同时使用 const 关键字来定义一个指针，既是常量指针又是指针常量，即它所指向的数据不能被修改，同时指针本身的值也不能改变。
4. 总结
   - **常量指针**：可以改变指针本身的值，但不能修改它所指向的数据。
   - **指针常量**：指针本身的值不能改变，但可以修改它所指向的数据（除非数据本身是常量）。

结构体和类的区别

1. `struct` 只能包含成员变量，不能包含成员函数。而在 C++ 中， `struct` 类似于 `class` ，既可以包含成员变量，又可以包含成员函数。
2. 不同点：
   - `class` 中的成员默认都是 `private` 的，而 `struct` 中的成员默认都是 `public` 的；
   - `class` 继承默认是 `private` ， `struct` 继承默认是 `public` ；
   - `class` 可以用于定义模板函数，而 `struct` 不行。

- 实际使用中，`struct` 我们通常用来定义一些 `POD(plain old data)` 类型，它是用来描述一种数据类型的特性，主要用于在内存中表示简单的数据结构。

什么是智能指针，C++有哪几种智能指针

智能指针是C++中用来自动管理动态分配内存的一种模板类，维护着一个指向动态分配对象的指针，并在智能指针对象被销毁时，自动释放该内存，从而避免内存泄漏。

C++有以下几种智能指针：

- **`std::unique_ptr`**：独占式拥有指针，保证同一时间只有一个 `unique_ptr` 指向特定内存，适用于不需要共享所有权的场景，如栈上对象的管理。
- **`std::shared_ptr`**：多个 `shared_ptr` 可以共享同一内存，使用引用计数机制来管理内存的生命周期，适用于多个对象需要共享同一个资源的场景。
- **`std::weak_ptr`**：弱引用，用于解决 `shared_ptr` 可能导致的循环引用问题，它不拥有所指向的对象。

智能指针的实现原理是什么

1. `std::unique_ptr`
    - `unique_ptr` 代表独占所有权的智能指针，同一时间只能有一个 `unique_ptr` 实例指向特定资源。
    - 它通过析构函数来管理资源的释放。当 `unique_ptr` 超出作用域时，会自动调用删除操作符来释放其指向的内存。
    - `std::unique_ptr` 通过**删除复制构造函数和复制赋值运算符来确保所有权的唯一性**，但提供移动构造函数和移动赋值运算符，允许所有权的转移。
2. `std::shared_ptr`
    - `shared_ptr` 允许多个指针实例共享对同一资源的所有权，使用**引用计数机制**来跟踪有多少个 `shared_ptr` 指向同一资源。

- 内部维护一个控制块，通常包含引用计数和资源的原始指针。每当创建一个新的 `shared_ptr` 或将一个 `shared_ptr` 赋值给另一个时，引用计数增加。
- 当 `shared_ptr` 被销毁或通过标准库提供的 `reset` 成员函数重置时，引用计数减少。当引用计数降到零时，控制块会释放资源并自我销毁。

3. `std::weak_ptr`
   - `std::weak_ptr` 是一种不拥有对象的智能指针，它观察 `std::shared_ptr` 管理的对象，但不增加引用计数。
   - 它用于解决 `std::shared_ptr` 之间可能产生的循环引用问题，因为循环引用会导致引用计数永远不会达到零，从而造成内存泄漏。

new和molloc有什么区别

`new` 和 `malloc` 在C++中都用于动态内存分配，但它们之间有几个关键的区别：

1. **语法层面**：
   - `new` 是C++的操作符，可以直接用来分配对象或数组。
   - `malloc` 是一个函数，通常需要包含头文件 `<cstdlib>`，并且只分配原始内存。
2. **类型安全**：
   - `new` 是类型安全的，它会根据分配的对象类型进行正确的内存分配和构造函数调用。
   - `malloc` 不是类型安全的，它只分配原始内存，不调用构造函数。返回类型是 `void*`，需要强制类型转换为具体的指针类型。
3. **构造与析构**：
   - 使用 `new` 分配的对象在对象生命周期结束时需要使用 `delete` 来释放，`delete` 会自动调用对象的析构函数。
   - 使用 `malloc` 分配的内存需要使用 `free` 来释放，`free` 不会自动调用析构函数，因此如果分配的是对象数组，需要手动调用析构函数。
4. **异常安全性**：

- `new` 在分配失败时会抛出 `std::bad_alloc` 异常。
- `malloc` 在分配失败时返回 `NULL` 指针。

5. 管理机制：
   - C++中的 `new` 和 `delete` 通常由编译器实现，可能包含一些额外的内存管理机制。
   - C语言的 `malloc` 和 `free` 由C标准库提供，与编译器无关。

总结来说， `new` 和 `malloc` 都是动态内存分配的手段，但 `new` 提供了类型安全和构造/析构的自动化，而 `malloc` 则提供了更底层的内存分配方式，需要手动管理构造和析构。在C++中，推荐使用 `new` 来分配对象，以保持类型安全和自动化的资源管理。

delete 和 free 有什么区别?

`delete` 和 `free` 都是用来释放动态分配的内存，但它们有不同的使用方式：

1. **语法**：
   - `delete` 是C++中的关键字，用于释放由 `new` 分配的对象。
   - `free` 是C语言中的函数，通常包含在 `<stdlib.h>` 头文件中，用于释放由 `malloc` 分配的内存。
2. **对象销毁**：
   - 当使用 `delete` 释放对象内存时， `C++` 编译器会自动调用对象的析构函数，释放与对象相关的资源，并执行对象的清理工作。
   - `free` 仅释放内存，不调用析构函数。因此，如果使用 `malloc` 分配了 `C++` 对象的内存，需要手动调用析构函数后再调用 `free` 。
3. **数组处理**：
   - 如果是数组，C++提供了 `delete[]` 来释放整个数组的内存，而C语言中仍然使用 `free` ，没有区分单个对象和数组。
4. 返回值:
   - `free` 没有返回值，即使内存释放失败，也不会反馈任何信息。
   - `delete` 之后，指针会自动置为 `nullptr`

1. 类型检查:
    - `delete` 进行类型检查，确保删除的对象类型与 `new` 分配时的类型一致。
    - `free` 不进行类型检查，因为它只处理 `void*` 类型的指针。

总结来说， `delete` 和 `free` 都是用来释放动态内存的，但它们分别用于C++和C语言中的内存管理。 `delete` 适用于C++对象，会自动调用析构函数；而 `free` 适用于C语言分配的内存，不涉及对象的析构。

什么是内存泄漏, 如何检测和防止?

1. **如果程序的某一段代码在内存池中动态申请了一块内存而没有及时将其释放，就会导致那块内存一直处于被占用的状态而无法使用，造成了资源的浪费。**内存泄漏并不是说物理上的消失掉了，是因为无法使用该区域，在外界看来这块内存就好像被泄漏了一样。
2. 什么操作会导致内存泄漏
    - 忘记释放内存：使用 `new` 或 `malloc` 等分配内存后，没有使用 `delete` 或 `free` 释放内存。
    - 子类继承父类时，没有将基类的析构函数定义为虚函数。
    - 野指针：指针被赋值为 `nullptr` 或重新赋值后，丢失了对先前分配内存的引用，导致无法释放。
    - 循环引用：在使用引用计数的智能指针（如 `std::shared_ptr` ）时，循环引用会导致引用计数永远不会归零，从而无法释放内存。
    - 使用不匹配的内存释放函数: 使用 `delete` 释放由 `new[]` 分配的内存，或使用 `delete[]` 释放由 `new` 分配的内存，这可能导致未定义行为。
    - 资源未关闭：对于文件、网络连接等资源，如果没有正确关闭，虽然不直接导致内存泄漏，但会占用系统资源，可能导致资源耗尽。
3. 如何检测：使用工具如Valgrind、AddressSanitizer或Visual Studio的诊断工具来检测内存泄漏。
4. 如何避免
    - 使用**智能指针**:优先使用 `std::unique_ptr` 、 `std::shared_ptr` 等智能指针来自动管理内存。

- 确保资源释放: 对于手动分配的内存，确保在不再需要时使用 `delete` 或 `free` 释放。
- 内存泄漏检测工具: 在开发和测试阶段，定期使用内存泄漏检测工具检查程序。

什么是野指针？如何避免？

1. 什么是野指针

野指针是指“**指向已经被释放的或无效的内存地址的指针**”。在 C 和 C++ 这类允许直接操作内存地址的语言中，如果指针没有被正确初始化，或者指针所指向的内存已经被释放，那么这个指针就成为了野指针。使用野指针可能会导致程序崩溃、数据损坏或者其他一些不可预测的行为。

2. 在什么情况下会产生野指针?
   - 在释放后没有置空指针： 使用 `delete` 或 `free` 释放了内存后，没有将指针设置为 `nullptr`，指针仍然指向已释放的内存地址。
   - 返回局部变量的指针 : 如果函数返回了指向其局部变量的指针，一旦函数返回，这些局部变量的生命周期结束，返回的指针成为野指针。
   - 越界访问：指针访问的内存超出了其合法的内存块边界。
   - 函数参数指针被释放。
3. 如何避免野指针
   - 在释放内存后将指针置为 `nullptr` 。
   - 避免返回局部变量的指针。
   - 使用智能指针（如 `std::unique_ptr` 和 `std::shared_ptr` ）。
   - 注意函数参数的生命周期，避免在函数内释放调用方传递的指针，或者通过引用传递指针。

C++内存分区

30477字很多情况下，提到 C++ 程序的内存分区时，会简化为下面五个主要区域

- 栈（`Stack`）：用于存储局部变量和函数调用的上下文。栈的内存分配是自动的，由编译器管理。
- 堆（`Heap`）：用于动态内存分配。程序员可以使用`new`、`malloc`等操作符或函数从堆上分配内存，并使用`delete`、`free`释放内存。
- 全局/静态存储区（`Global/Static Storage`）：**存储全局变量和静态变量**，包括：
  - 数据段：存储初始化的全局变量和静态变量。
  - `BSS 段`：存储未初始化的全局变量和静态变量。
- 常量存储区（`Constant Data`）：存储程序中的常量数据，如字符串字面量。
- 代码段（`Code Segment 或 Text Segment`）：存储程序的可执行代码和函数的二进制指令。

堆和栈的区别

堆`（Heap）`和栈`（Stack）`是程序运行时两种不同的内存分配区域

- 内存分配：
  - 栈
    - 是由编译器自动管理的，用于存储局部变量和函数调用的上下文信息。
    - 栈上的对象在定义它们的块或函数调用结束后自动销毁。
    - 栈的内存分配和释放速度很快，因为栈的内存管理是连续的，不需要搜索空闲内存。
  - 堆
    - 由程序员手动管理的，用于存储动态分配的对象。
    - 堆上的对象需要程序员手动释放，否则可能导致内存泄漏。
    - 堆的内存分配和释放速度通常比栈慢，因为可能需要搜索合适的内存块，并且涉及内存碎片整理。
- 大小限制：
  - 栈的大小通常有限制，远小于堆的大小，且在不同系统和编译器中可能不同。
  - 堆的大小通常很大，受限于系统可用内存。

- 使用场景：
  - 栈主要用于存储函数参数、局部变量等。
  - 堆用于存储生存期不受限于单个函数调用的对象，如使用 `new` 或 `malloc` 分配的对象。

C++面向对象三大特性

面向对象编程是C++的核心特性之一。`OOP` 是一种编程范式，通过它，程序可以以对象的形式组织数据和功能，并通过对象之间的交互来实现任务。面向对象编程具有**封装、继承和多态**三个主要特性：

1. 封装：将客观事物封装成为抽象的类，类把自己数据和方法进行隐藏，仅对外公开接口来和对象进行交互，防止外界干扰或不确定性访问。
2. 继承：指一个类（称为子类或派生类）可以从另一个类（称为父类或基类）中继承属性和行为的能力。通过继承，子类可以重用父类的代码，并且可以在不修改父类的情况下添加新的功能或修改已有的功能。继承使得代码具有层次性和可扩展性，能够建立起类之间的层次关系。
3. 多态：多态是指同一个操作作用于不同的对象时，可以有不同的解释和行为。多态性允许以统一的方式处理不同类型的对象，从而提高了代码的可扩展性和可维护性。在C++中，**多态性通常通过虚函数（virtual functions）来实现。**通过基类中定义虚函数，并在派生类中重新定义该函数，可以实现运行时的动态绑定。

简述一下 C++的重载和重写，以及它们的区别和实现方式

1. 重载和重写是两种不同的概念，它们都用于实现多态性，但方式和使用场景有所不同。
2. 重载：在同一个类或命名空间中，声明多个同名函数， 但是参数列表不同。编译器根据参数的类型、数量或顺序来区分不同的函数。
3. 重写：重写发生在继承体系中，在子类中，声明一个与父类中虚函数具有相同名称、相同参数列表和相同返回类型的函数，并在子类函数前加上 `override` 关键字。
4. 区别：

- 作用域：重载发生在同一个作用域内，而重写发生在继承体系中。
- 参数列表：对于重载的函数，参数列表必须不同；对于重写的函数，参数列表必须与被重写的函数完全相同。
- 返回类型：重载函数的返回类型可以不同，但重写函数的返回类型必须与被重写的函数相同（或与之兼容，C++中称为协变返回类型）。
- 虚函数：重写通常与虚函数一起使用，以实现**运行时多态性**；而重载是**编译时多态性**，由编译器在编译期间确定调用哪个函数。
- 关键字：重写函数可以使用 `override` 关键字，明确指出该函数是对父类虚函数的重写。

C++怎么实现多态

1. C++ 的多态分为编译时多态（也被称为静态多态）和运行时多态（也被称为动态动态）
2. **编译时多态**
   - 函数**重载**（`Function Overloading`）:允许在同一作用域内声明多个同名函数，只要它们的参数列表不同。
   - 运算符重载（`Operator Overloading`）：允许为自定义类型定义或修改运算符的行为。
   - 模板（`Templates`）：允许创建泛型类和函数，它们可以在多种数据类型上使用。

编译时多态在编译期间就确定了具体的函数或类型，由编译器根据函数的签名或模板实例化来选择正确的函数或实例。

1. **运行时多态:** 运行时多态主要通过虚函数和抽象类实现，父类中定义声明虚函数，子类实现对虚函数的重写。由虚函数表和虚函数指针在运行时确定调用哪个函数。
   - 虚函数（`Virtual Functions`）：虚函数在基类定义，派生类可以**重写**这些虚函数。
   - 抽象类（`Abstract Classes`）：包含至少一个纯虚函数的类，不能被实例化，但可以作为其他类的基类。
1. 运行时多态实现原理：
   - 运行时多态涉及到**虚函数表（`**vtable**`）**和虚函数指针（`vptr`）。当类包含虚函数时，编译器会自动为该类创建一个虚函数表，表中包含类中所有虚函数的地址。
   - 子类的虚函数表继承了父类的虚函数表，但会使用自己重写的虚函数将虚函数表中对应的虚函数进行覆盖。

- 当通过基类指针或引用调用虚函数时，程序会根据对象的实际类型在运行时查找正确的函数地址并调用相应的函数，实现多态。

虚函数和纯虚函数的区别

虚函数和纯虚函数都用于实现多态。

1. 虚函数
   - 虚函数是在普通函数之前加一个 `virtual` 关键字
   - 虚函数是在基类中声明的，并且可以在派生类中被重写。
   - 虚函数可以有实现，也就是说，基类中的虚函数可以有一个定义，派生类可以选择提供自己的实现，也可以使用基类的实现。
   - 通过虚函数，可以在基类指针或引用中实现动态绑定，即在运行时确定调用哪个类中的函数实现。
1. 纯虚函数
   - 纯虚函数是在虚函数后面加一个 `=0`
   - 纯虚函数也是在基类中声明的，但它没有实现，只有声明。
   - 当一个类包含至少一个纯虚函数时，它就成为了一个抽象类，这意味着你不能实例化这样的类，但可以声明这种类型的指针或引用。
1. 区别
   - 是否实现:
     - 虚函数提供函数声明和实现，即提供虚函数的默认实现。
     - 纯虚函数没有函数具体实现，只提供函数声明。
   - 派生类是否实现
     - 派生类可以选择是否覆盖虚函数的默认实现。
     - 当一个类包含至少一个纯虚函数时，派生类必须提供具体实现，否则他们也变成抽象类。
   - 实例化:

    - 包含纯虚函数的类是抽象类，不能被实例化；
    - 而包含虚函数的类不一定是抽象类，可以被实例化，除非它也包含纯虚函数。
- 目的:
    - 虚函数用于提供一个可以在派生类中被重写的方法实现；
    - 通过纯虚函数，抽象类提供一种接口规范，要求派生类必须提供具体实现。
- 动态绑定:
    - 虚函数支持动态绑定，
    - 纯虚函数由于没有实现，它们本身不参与动态绑定，但可以作为接口的一部分，影响整个类的多态性。

虚函数怎么实现的

虚函数的实现依赖于一种称为**虚函数表**的机制。

1. **虚函数表的创建**: 当一个类包含虚函数时，编译器会自动为这个类创建一个虚函数表。这个表是一个函数指针数组，每个指针指向一个虚函数的实现。
2. **虚函数表指针**: 编译器会在对象的内存布局中添加一个隐式的虚函数表指针（通常是一个指向 `vtable` 的指针），这样每个对象都可以通过这个指针访问到类的虚函数表。
3. **虚函数的声明**: 在类中声明虚函数时，可以使用 `virtual` 关键字。如果一个函数被声明为虚函数，编译器会在类的 `vtable` 中为这个函数分配一个入口。
4. **重写虚函数**: 当从基类继承并创建派生类时，可以在派生类中重写基类的虚函数。重写的函数会替换掉 `vtable` 中对应的基类实现。
5. **动态绑定**: 当通过基类指针或引用调用虚函数时，程序会使用对象的虚函数表指针来查找正确的函数实现。这个过程称为动态绑定或晚期绑定。
6. **调用虚函数**: 程序运行时，当调用一个虚函数时，会先通过对象的虚函数表指针找到 `vtable`，然后在 `vtable` 中查找对应的函数指针，并调用该函数。

简短来说，每个类都有一个虚表，里面有这个类的虚函数地址；每个对象都有指向它的类的虚表的指针，这个指针称为虚指针。当调用虚函数时，编译器会调用对象的虚指针查找虚表，通过虚函数的地址来执行相应的虚函数。

虚函数表是什么

1. 虚函数表是 C++ 中实现运行时多态（动态绑定）的关键机制之一。
2. 虚函数表是一个或多个函数指针的集合，它存储了类中所有虚函数的地址。**当类包含虚函数时，编译器会自动为这个类创建一个虚函数表。**
3. **虚函数表的主要目的是在运行时能够确定通过基类指针或引用调用的是哪个派生类中的虚函数实现，从而实现动态绑定。**
4. 原理
   - 创建虚函数表：当类声明至少一个虚函数时，编译器会为这个类生成一个虚函数表。
   - 虚函数表指针：编译器会为包含虚函数的类的对象添加一个隐藏的虚函数表指针（通常是一个指针或引用），指向类的虚函数表。
   - 调用虚函数：当通过基类指针或引用调用虚函数时，程序会使用对象的虚函数表指针来查找并调用正确的函数实现。

什么是构造函数和析构函数？构造函数、析构函数可以是虚函数嘛？

1. 构造函数
   - 构造函数是创建对象时自动调用的成员函数，它的作用是初始化成员变量，为对象分配资源，执行必要的初始化操作。
   - 特点
     - 函数名必须与类名相同，且没有返回类型；
     - 可以有多个构造函数；
     - 如果没有为类定义一个构造函数，编译器会自动生成一个默认构造函数，它没有参数，也可能执行一些默认的初始化操作。
   - 构造函数不能是虚函数。

- 构造函数在对象创建时被调用，此时不涉及多态性。
- 虚函数对应一个虚表，这个表存在对象的内存空间，如果此时构造函数是虚函数，对象还没实例化没有分配内存空间，也就无法调用；
- 虚函数是用在信息不全的情况下，能使重载的函数使用。但构造函数本身就是要初始化对象，因此没必要是虚函数。

1. 析构函数

- 析构函数是对象生命周期结束时自动调用的函数，它的作用是释放对象占用的资源，执行一些必要的清理操作。
- 析构函数特点：
  - 函数名为 `~类名`；
  - 没有参数；
  - 如果没有为类定义一个析构函数，编译器会自动生成一个默认析构函数，执行简单的清理操作。
- 析构函数可以是虚函数。
  - 虚析构函数可以在运行时实现多态性；
  - 如果基类的析构函数不是虚函数，当通过基类指针去删除派生类对象时，不会调用派生类的析构函数。可能会导致派生类的资源未被正确释放，从而造成资源泄漏

C++构造函数有几种，分别什么作用

在C++中，构造函数有几种不同的类型，每种都有其特定的作用：

1. **默认构造函数**：没有参数的构造函数，用于创建对象的默认实例。
2. **参数化构造函数**：带参数的构造函数，允许在创建对象时初始化成员变量。
3. **拷贝构造函数**：以同一类的实例为参数的构造函数，用于复制已有对象。
4. **移动构造函数**：以同一类的实例的右值引用为参数，用于利用即将销毁的对象的资源。
5. **转换构造函数**：允许将其他类型或值隐式转换为当前类类型的实例。

委托构造函数：一个构造函数调用另一个构造函数来完成初始化，可以是同一个类的其他构造函数。

6.

7. **初始化列表构造函数**：使用成员初始化列表来初始化成员变量，这是最高效的初始化方式。
8. **常量构造函数**：声明为 `const` 的构造函数，可以用于创建常量对象。
9. **constexpr构造函数**：允许在编译时初始化对象，用于定义和初始化字面量类型的对象。

每种构造函数的使用场景不同，例如：

- **默认构造函数**用于快速创建对象，而不需要显式提供任何初始化参数。
- **参数化构造函数**提供了灵活性，允许在创建对象时定制其状态。
- **拷贝构造函数**和**移动构造函数**分别用于对象的复制和移动，是实现资源管理的关键。
- **转换构造函数**和**委托构造函数**提供了更灵活的对象初始化方式。
- **初始化列表构造函数**是C++中推荐的成员初始化方式，因为它可以提高效率。

深拷贝与浅拷贝的区别

1. 浅拷贝
   - 定义：浅拷贝**仅复制对象本身，不复制对象所指向的动态分配的内存**。换句话说，它只复制内存中的对象副本，而不复制对象内部指向的任何动态分配的资源。
   - 实现：通常通过复制构造函数或赋值运算符实现。
   - 特点：
     - 速度快，因为只涉及基本数据类型的复制。
     - 如果原始对象包含指针，浅拷贝会导致两个对象尝试管理相同的动态内存，这可能导致多重释放和悬空指针问题。
1. 深拷贝
   - 定义：深拷贝不仅复制对象本身，还**递归地复制对象所指向的所有动态分配的内存**。这意味着每个对象都有自己的独立资源副本。

- 实现：通常需要自定义复制构造函数或赋值运算符来确保所有动态分配的资源都被正确复制。
- 特点：
  - 速度慢，因为需要递归地复制所有资源。
  - 可以安全地使用复制出的对象，而不担心资源管理问题。

STL 容器了解哪些

1. 序列容器
   - `std::vector`：**动态数组，提供快速随机访问。**
   - `std::deque`：**双端队列，提供从两端快速插入和删除的能力。**
   - `std::list`：**双向链表，提供高效的元素插入和删除。**
   - `std::forward_list`：单向链表，每个元素只存储下一个元素的引用。
   - `std::array`：固定大小的数组，具有静态分配的内存。
1. 关联容器：
   - `std::set`：基于红黑树，存储唯一元素的集合，会默认按照升序进行排序。
   - `std::multiset`：允许容器中有多个相同的元素。
   - `std::map`：基于红黑树，存储键值对的有序映射。
   - `std::multimap`：允许映射中有多个相同的键。
   - `std::unordered_set`：基于哈希表，提供平均时间复杂度为 `O(1)` 的查找。
   - `std::unordered_map`：基于哈希表，存储键值对的无序映射。
1. 容器适配器（Container Adapters）：
   - `std::stack`：后进先出（LIFO）的栈。
   - `std::queue`：先进先出（FIFO）的队列。

- `std::priority_queue`：优先队列，元素按优先级排序。

vector和list的区别

1. `vector`
   - 基于动态数组：`std::vector` 基于可以动态扩展的数组实现，这意味着它在内存中**连续存储**元素。
   - 随机访问：提供快速的随机访问能力，可以**通过索引快速访问**任何元素。
   - 内存分配：通常在内存分配上更紧凑，因为元素紧密排列，没有额外的空间用于链接或指针。
   - 时间复杂度：
     - 元素访问：`O(1)`，即常数时间复杂度。
     - 插入和删除：在 `vector` 的末尾是 `O(1)`，但如果需要在中间插入或删除元素，则可能需要 `O(n)`，因为可能需要移动后续所有元素。
   - 内存管理：使用连续内存分配，可以利用缓存的优势，提高访问速度。
1. `list`
   - 基于双向链表：`std::list` 是基于**双向链表**的容器，每个元素通过节点链接到前一个和后一个元素。
   - 非连续存储：**元素在内存中不是连续存储的**，每个元素包含指向前一个和后一个元素的指针。
   - 时间复杂度：
     - 元素访问：`O(n)`，需要从头开始遍历到所需位置。
     - 插入和删除：非常快速，特别是当需要在列表中间插入或删除元素时，操作是 `O(1)`，前提是已经拥有指向待插入或删除元素的迭代器。
   - 内存管理：由于元素间通过指针链接，内存分配可能更分散，但插入和删除操作不需要移动其他元素。
1. 使用场景
   - `std::vector`：
     - 当你需要快速随机访问元素时。

    - 当你需要在末尾快速添加或删除元素时。
    - 当你关心内存使用效率时。
- `std::list`：
    - 当你需要在列表中间高效地插入或删除元素时。
    - 当你不需要随机访问元素时。
    - 当你需要一个灵活的容器，可以动态地添加和删除元素而不会引起大量的内存复制或移动。

vector 底层原理和扩容过程

1. 底层原理

- `vector` 是 `C++` 标准库中的一个容器，可以看作是一个**动态数组**，它的大小可以根据元素的增加而增长。它通过在堆上分配一段**连续的内存空间存放元素**，支持时间复杂度为 `O（1 ）` 的随机访问。
- `vector` 底层维护了三个**迭代器**和两个变量，这三个迭代器分别指向对象的起始字节位置，最后一个元素的末尾字节和整个 `vector` 分配空间的末尾字节。两个变量分别是 `size` 和 `capacity` ， `Size` 表示当前存储元素的数量， `capacity` 表示当前分配空间的大小。当创建一个 `vector` 对象时，会分配一个初始化大小的空间存放元素，初始化空间可以通过构造函数的参数指定，缺省情况下为 `0` 。当对 `vector` 容器进行增加和删除元素时，只需要调整末尾元素指针，而不需要移动整个内存块。

1. 扩容机制

- 当添加元素的数量达到当前分配空间的大小时， `vector` 会申请一个更大的内存块，然后将元素从旧的内存块拷贝到新的内存块中，并释放旧的内存块。 扩容可能导致原有迭代器和指针失效，扩容完成后，容器返回指向新内存区域的迭代器或指针。
- `vector` 扩容的机制分为固定扩容和加倍扩容。
    - 固定扩容就是在每次扩容时在原容量的基础上增加固定容量。但是固定扩容可能会面临多次扩容(扩容的不够大)的情况，时间复杂度较高。

- 加倍扩容就是在每次扩容时原容量翻倍，优点是使得正常情况下扩容的次数大大减少，时间复杂度低，缺点是空间利用率低。

push_back()和emplace_back()的区别

`push_back()` 和 `emplace_back()` 都是C++标准库容器（如 `std::vector` ）中用来添加元素的方法，但它们在添加元素的方式上有所不同：

1. `push_back()` ：
   - 语法是 `container.push_back(value);` ，传入的是一个值或对象的副本/移动版本。
   - 它接受一个元素的副本或移动该元素，然后将其添加到容器的末尾。
   - 这个方法需要先构造一个元素的副本或移动构造一个临时对象，然后再将其添加到容器中。
2. `emplace_back()` ：
   - 语法是 `container.emplace_back(args...);` ，传入的是构造新元素所需的参数列表。
   - 它使用就地构造（emplace）的方式，直接在容器的内存空间中构造新元素。
   - 这个方法避免了元素的复制或移动操作，因为它直接在容器的末尾空间构造新元素。
3. 性能：
   - `emplace_back()` 通常比 `push_back()` 更高效，因为它避免了额外的复制或移动操作。
   - 当构造函数需要大量资源时， `emplace_back()` 的优势更加明显。

map dequeu list的实现原理

1. `std:: map`

- 基于红黑树： `std::map` 基于一种自平衡的二叉搜索树——红黑树实现。
- 有序容器：元素按照键的顺序自动排序，通常是按照小于（<）运算符定义的顺序。

- 唯一键：每个键都是唯一的，不允许有重复的键。
- 时间复杂度：提供对数时间复杂度 `（O(log n)）` 的查找、插入和删除操作。
- 迭代器：由于 `std::map` 是基于树的，迭代器在遍历时是有序的。

1. `std::list`

- 基于双向链表：`std::list` 是一个双向链表，每个元素都持有指向前一个和后一个元素的指针。
- 无序容器：元素在容器中没有特定的顺序。
- 插入和删除：提供高效的插入和删除操作，特别是当需要在容器中间插入或删除元素时。
- 时间复杂度：提供线性时间复杂度 `（O(n)）` 的查找操作，但插入和删除可以在 `O(1)` 时间内完成，前提是已经拥有指向待插入或删除元素的迭代器。
- 迭代器：由于 `std::list` 是线性结构，迭代器在遍历时是顺序的，但不支持随机访问。

1. `std::deque`

- 基于动态数组：`std::deque` 是一个基于动态数组的序列容器，可以高效地从两端添加或删除元素。
- 允许序列操作：可以快速地在队列的前端和后端添加或删除元素。
- 时间复杂度：提供常数时间复杂度 `（O(1)）` 的前端和后端插入和删除操作。中间插入或删除操作可能需要 `O(n)` 时间。

map && unordered_map的区别和实现机制

1. `map`

- 基于**红黑树**：`std::map` 基于一种自平衡的二叉搜索树（通常是红黑树）实现，可以保持元素有序。
- **有序**容器：元素按照键的顺序自动排序，可以通过键值进行有序遍历。
- 元素访问：提供对元素的快速查找、插入和删除操作，时间复杂度为 `O(log n)`。
- **唯一**键：每个键都是唯一的，不允许有重复的键。
- 迭代器稳定性：由于基于树结构，迭代器在遍历时是稳定的，即使容器发生插入或删除操作，迭代器指向的元素也不会改变，除非该元素被删除。

1. `unordered_map`

- 基于**哈希表**：`std::unordered_map` 基于哈希表实现，通过哈希函数将键分布到数组的槽位中。
- **无序**容器：元素在容器中是无序的，不能按键的顺序进行遍历。
- 元素访问：理想情况下，提供平均时间复杂度为 `O(1)` 的快速查找、插入和删除操作。最坏情况下，性能可能下降到 `O(n)`。
- 允许重复键：实际上，`std::unordered_map` 不允许有重复的键，因为哈希表的设计不允许两个元素具有相同的哈希值。如果发生哈希冲突，会通过某种方式（如链表或开放寻址）解决。
- 迭代器稳定性：由于基于哈希表，迭代器的稳定性不如 `std::map`。在发生哈希表的重新哈希 `（rehashing）` 时，迭代器可能会失效。
- 遍历顺序与创建该容器时输入元素的顺序是不一定一致的，遍历是按照哈希表从前往后依次遍历的。

1. 使用场景

- **当需要元素有序且对性能有较高要求时，应选择 `**std::map**`。**
- **当元素的顺序不重要，且需要快速访问元素时，应选择 `**std::unordered_map**`。**

1. 实现机制

- `std::map` 的实现依赖于红黑树的旋转和颜色变换来保持树的平衡，确保操作的时间复杂度。
- `std::unordered_map` 的实现依赖于一个良好的哈希函数来最小化冲突，并通过**解决冲突的机制（如链表或开放寻址）**来存储具有相同哈希值的元素。

C++11新特性有哪些

1. 类型推导：
    - `auto` 关键字，允许编译器根据初始化表达式推导变量类型。
    - `decltype` 分析表达式并得到它的类型，却不实际计算表达式的值。

1. 基于范围的 `for` 循环：提供了一种更简洁的遍历容器的方法。

2. `lambda` 表达式：允许在需要的地方定义匿名函数。
3. 智能指针（如 `std::unique_ptr` 和 `std::shared_ptr`）：提供了自动内存管理。
4. 右值引用和移动语义：优化资源的移动操作，高效的将资源从一个对象转移到另一个对象，减少拷贝的开销。
5. `nullptr`：空指针，明确表示空指针的关键字。

移动语义有什么作用，原理是什么

1. 移动语义是 C++11 引入的一项特性，对于大型对象或包含资源的对象（如动态分配的内存、文件句柄等），复制构造函数可能会非常昂贵。移动语义允许对象的资源被“移动”到新对象，而不是复制，从而节省资源和时间。其主要作用是优化资源的利用，特别是在对象的复制操作中。
2. 移动语义通过移动构造函数和移动赋值运算符实现。在移动构造或移动赋值过程中，源对象的资源被“拿走”，并转移到目标对象，源对象变为无效状态。

左值引用和右值引用的区别

在C++中，左值和右值是两种不同类型的表达式，它们分别对应着不同的引用方式：

1. 左值引用：
    - 左值引用使用 `&` 符号
    - 左值引用绑定到左值表达式上，即那些具有持久存储期的表达式，如变量或者对象。
    - 它们在内存中有一个持久的地址，可以被赋值和修改。
2. 右值引用：
    - 右值引用使用两个 `&` 符号
    - 右值引用绑定到右值表达式上，通常是临时对象或即将销毁的对象，它们没有持久的存储期。
    - 右值引用的主要目的是通过移动语义来利用这些临时资源，避免不必要的复制。

3．区别总结

- 绑定对象：左值引用绑定到具有持久状态的对象，而右值引用绑定到临时或即将销毁的对象。
- 生命周期：左值引用延长了它所引用对象的生命周期，右值引用则表示对一个临时值的引用。
- 可修改性：左值引用可以被用来修改其所引用的对象，而右值引用通常用于移动语义，不涉及修改。
- 标准库支持：C++11 标准库中的某些函数和算法（如 `std::move`）特别设计来与右值引用配合使用。

说一下lambda函数

`lambda` 表达式（也称为匿名函数）是在 `C++11` 标准中引入的一种方便的函数编写方式。 `Lambda` 允许你在需要一个函数对象的地方快速定义函数的行为，而不需要按照传统方式定义一个完整的函数。

我们主要出于这些情况用 `lambda` 函数：

- **API**： `Lambda` 函数可以简化回调函数的编写，特别是在使用 `STL` 时。
- **简洁**：在需要临时使用一个函数但不想定义一个完整函数的情况下。
- **闭包**： `Lambda` 可以捕获外部变量，形成闭包，使得函数可以访问和操作外部作用域的变量。

说一下 select、poll、epoll

I/O多路复用通常通过select、poll、epoll等系统调用来实现。

- **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。

总结：`select`是最早的`I/O`多路复用技术，但受到文件描述符数量和效率方面的限制。`poll`克服了文件描述符数量的限制，但仍然存在一定的效率问题。`epoll`是一种高效的I/O多路复用技术，尤其适用于高并发场景，但它仅在`Linux`平台上可用。一般来说，`epoll`的效率是要比`select`和`poll`高的，但是对于活动连接较多的时候，由于回调函数触发的很频繁，其效率不一定比`select`和`poll`高。所以`epoll`在连接数量很多，但活动连接较小的情况性能体现的比较明显。

C++如何实现一个单例模式

在C++中实现单例模式，主要是确保一个类只有一个实例，并提供一个全局访问点, C++实现单例模式需要满足以下几点要求：

- **私有化构造函数**：将类的构造函数定义为私有，防止外部通过`new`关键字创建多个实例。
- **静态实例**：在类内部提供一个静态私有实例，这个实例将作为整个程序的唯一实例。
- **公有访问方法**：提供一个公有的静态方法，通常称为`getInstance`，用于获取类的唯一实例。
- **删除拷贝构造函数和赋值操作符**：为了防止通过拷贝或赋值来创建新的实例，需要将拷贝构造函数和赋值操作符定义为私有或删除。

单例模式有懒汉式和饿汉式两种实现。

- 懒汉式 类实例只有在第一次被使用时才会创建，这个时候需要注意多线程下的访问，需要利用互斥锁来加以控制。
- 饿汉式 类实例在类被加载时就进行创建。