代码随想录知识星球-八股文速记版

前言

在代码随想录知识星球里每天会给录友们安排每日一题，都是根据星球里面经高频题目 整理而来，如图：

在知识星球的每日一题中，每个问题下有「回答作业排行榜」，基本高赞的都是回答的比较好的内容，点进去就是 「录友回答内容」：

回答作业排行榜

录友回答内容

这些每日一题以及高赞回答，就整理成了这份【代码随想录知识星球–八股文速记版】，这份文档，是结合了代码随想录知识星球里两年多来录友们的面经里高频问题以及对应的高频回答，**最后又精细整理，去冗精简而成**。

本文档只分享在 代码随想录知识星球 里，**作为星球录友准备求职，突击八股文的资料。**

星球里每日一题在不断更新中，在知识星球里的录友如果不知道每日打卡做些什么，学写什么，那么就可以按照星球里的每日一题进行打卡，这是每日一题汇总链接，或者也可以在星球专栏里找到每日一题汇总，开始大家的打卡之旅。

本PDF是快速背诵版（问题+回答），如果想检查自己的学习成果，可以按照 每日一题汇总链接 去回答问题，链接里面只有问题，可以尝试锻炼自己独立回答，看看答的如何。

计算机网络

1. 从输入 URL 到页面展示到底发生了什么?

本题星球问答链接

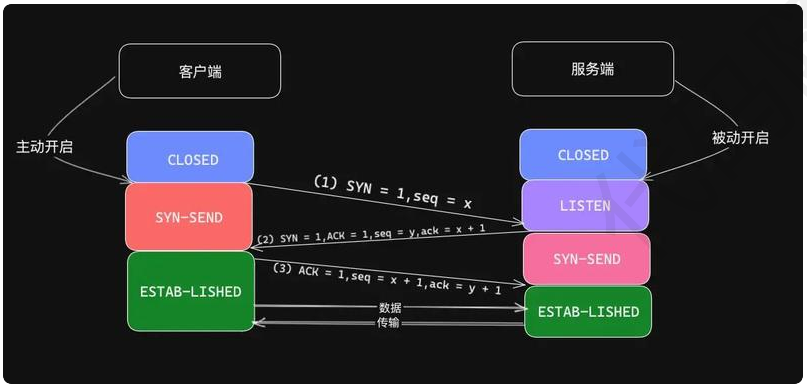
1. 浏览器接收到用户请求，先**检查浏览器缓存里是否有缓存该资源**，如果有直接返回；如果没有进入下一步网络请求。
2. 网络请求前，进行 **DNS解析** ，以获取请求域名的 IP地址 。如果请求协议是 HTTPS ，那么还需要 建立TLS连接 。DNS解析时会按本地浏览器缓存->本地 Host 文件->路由器缓存-> DNS 服务器->根 DNS 服务器的顺序查询域名对应 IP ，直到找到为止。
3. 浏览器与服务器IP建立TCP连接。连接建立后，浏览器端会构建请求行、请求头等信息，并把和该域名相关的 Cookie 等数据附加到请求头中，向服务器构建请求信息。
4. 服务器接收到请求信息，根据请求生成响应数据。
5. 浏览器解析响应头。若响应头状态码为 301、302 ，会重定向到新地址；若响应数据类型是字节流类型，一般会将请求提交给下载管理器；若是HTML类型，会进入下一部渲染流程。
6. 浏览器解析 HTML 文件，创建 DOM 树，解析 CSS 进行样式计算，然后将CSS和DOM合并，构建渲染树；最后布局和绘制渲染树，完成页面展示。

2. 三次握手的过程，以及为什么是三次，而不是四次，两次?

本题星球问答链接

三次握手的过程如下：

1. 客户端向服务器发送 SYN 报文、初始化序列号 ISN（seq=x） ，然后客户端进入 SYN_SEND 状态，等待服务器确认。
2. 服务端发送 ACK 确认服务端的 SYN 报文（ack=x+1） ，同时发出一个 SYN 报文，带上自己的初始化序列号 （seq=y ），然后服务端进入 SYN_RECV 状态。
3. 客户端接收到服务端的 SYN、ACK 报文，ACK确认服务端的 SYNC 报文 （ACK=y+1） ，然后客户端和服务器端都进入 ESTABLISHED 状态，完成 TCP 三次握手。



为什么不是四次握手？ 为什么不能两次握手?

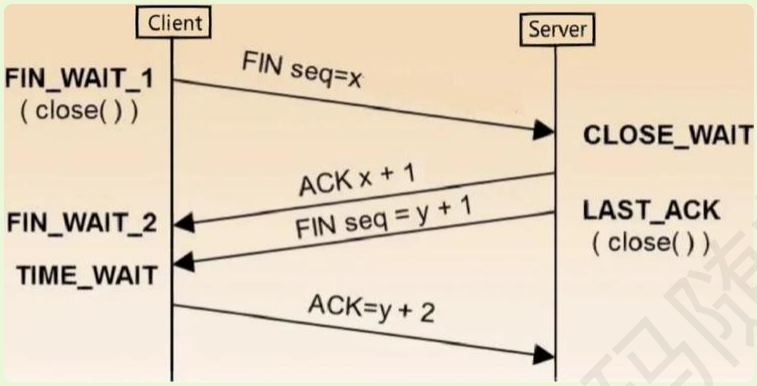
因为三次握手才能保证双方具有接收和发送的能力。 两次握手可能导致资源的浪费，由于没有第三次握手，服务端就无法确认客户端是否收到了自己的回复，所以每收到一个 `SYN` ，服务器都会主动去建立一个连接, 而四次握手可以优化为三次。

3. 四次挥手的过程，以及为什么是四次?

本题星球问答链接

四次挥手的过程:

1. 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. 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. 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. 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。



为什么是四次挥手?

`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了 `TCP` 连接。 总结：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

4. TCP与UDP的概念，特点，区别和对应的使用场景?

本题星球问答链接

1. TCP与UDP的概念

- `TCP`（传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议。
- `UDP`（用户数据报协议）为应用程序提供了一种无需建立连接就可以发送封装的IP数据包的方法。

2. 特点

- `TCP`：面向连接，传输可靠，传输形式为字节流，传输效率慢，所需资源多。
- `UDP`：无连接、传输不可靠、传输形式为数据报文段，传输效率快，所需资源少。

3. 区别

- 是否面向连接: TCP 是面向连接的传输，UDP 是无连接的传输。
- 是否是可靠传输：TCP是可靠的传输服务，在传递数据之前，会有三次握手来建立连接；在数据传递时，有确认、窗口、重传、拥塞控制机制。 UDP是不可靠传输，数据传递不需要给出任何确认，且不保证数据不丢失及到达顺序。
- 是否有状态：TCP 传输是有状态的，它会去记录自己发送消息的状态比如消息是否发送了、是否被接收了等等，而 UDP 是无状态的。
- 传输形式: TCP 是面向字节流的，UDP 是面向报文的。
- 传输效率:由于TCP 传输的时候多了连接、确认重传等机制，所以TCP 的传输效率要比UDP 低。
- 首部开销 :TCP 首部开销 (20 ~ 60字节)比UDP 首部开销 (8字节)要大。
- 是否提供广播或多播服务: TCP 只支持点对点通信UDP 支持一对一、一对多、多对一、多对多。

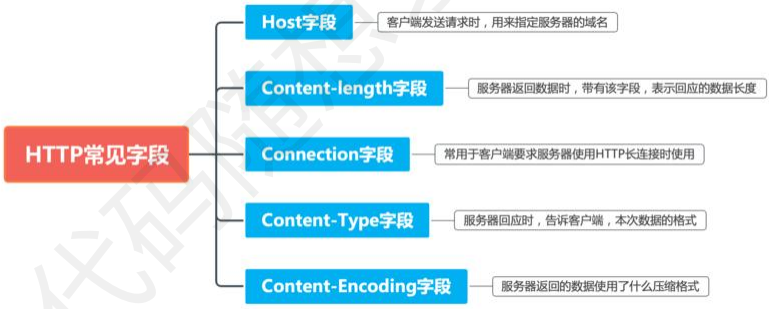
4. 对应的使用场景

- TCP常用于要求通信数据可靠场景（如网页浏览、文件传输、邮件传输、远程登录、数据库操作等）。
- UDP常用于要求通信速度高场景（如域名转换、视频直播、实时游戏等）。

	TCP	UDP
可靠性	可靠	不可靠
连接性	面向连接	无连接
报文	面向字节流	面向报文
效率	传输效率低	传输效率高
双工性	全双工	一对一、一对多、多对一、多对多
流量控制	滑动窗口	无
拥塞控制	慢开始、拥塞避免、快重传、快恢复	无
传输速度	慢	快
应用场景	对效率要求低，对准确性要求高或者要求有连接的场景	对效率要求高，对准确性要求低

5. HTTP请求常见的状态码和字段

本题星球问答链接



6. 常见的请求方式？GET和POST请求的区别？

本题星球问答链接

1. 作用不同

- GET用于从服务端获取资源
- POST一般用来向服务器端提交数据

2. 参数传递方式不同

- GET请求的参数一般写在URL中，且只接受ASCII字符
- POST请求参数一般放在请求体中，对于数据类型也没有限制

3. 安全性不同

- 因为参数传递方式的不同，所以两者安全性不同，GET请求的参数直接暴露在URL中，所以更不安全，不能用来传递敏感信息。

4. 参数长度限制不同

- GET传送的数据量较小，不能大于2KB。
- POST传送的数据量较大，一般被默认为不受限制。
- **HTTP 协议没有 Body 和 URL 的长度限制，对 URL 限制的大多是浏览器和服务器的原因。**

5. 编码方式不同

GET 请求只能进行 URL 编码（application/x-www-form-urlencoded）

POST 支持多种编码方式（application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多种编码。）

6. 缓存机制不同

- GET 请求会被浏览器主动cache，而 POST 不会，除非手动设置。
- GET 请求参数会被完整保留在浏览器历史记录里，而 POST 中的参数不会被保留。
- GET 产生的 URL 地址可以被 保存为书签，而 POST 不可以。
- GET 在浏览器回退时是无害的，而 POST 会再次提交请求。

7. 时间消耗不同

- GET 产生一个 TCP 数据包；
- POST 产生两个 TCP 数据包。

对于 GET 方式的请求，浏览器会把 header 和 data 一并发送出去，服务器响应 200（返回数据）；而对于 POST，浏览器先发送 Header，服务器响应 100 continue，浏览器再发送 data，服务器响应 200 ok（返回数据）

8. 幂等

意思是多次执行相同的操作，结果都是「相同」的。

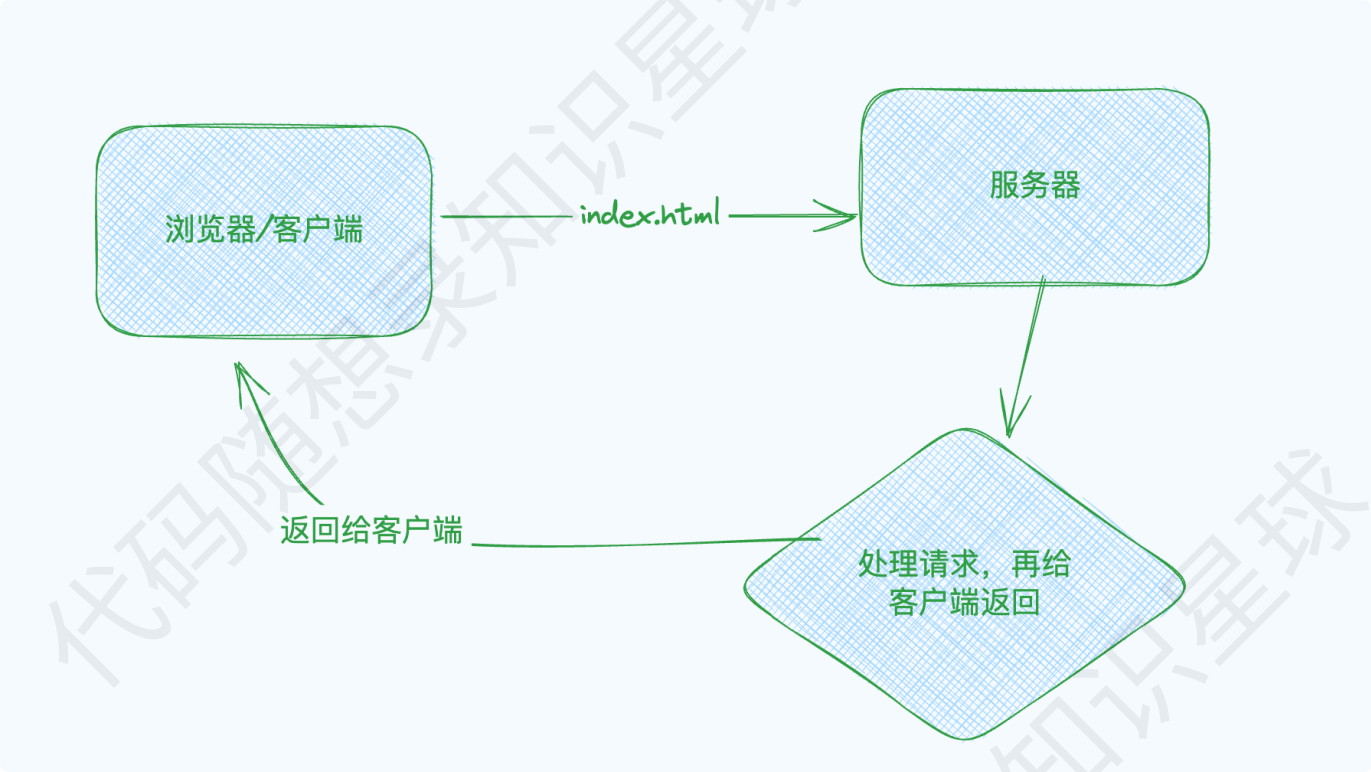
- **GET 方法就是安全且幂等的**，因为它是「只读」操作，无论操作多少次，服务器上的数据都是安全的，

且每次的结果都是相同的。

- POST 因为是「新增或提交数据」的操作，会修改服务器上的资源，所以是**不安全**的，且多次提交数据就会创建多个资源，所以**不是幂等**的。

7. 什么是强缓存和协商缓存

本题星球问答链接



缓存可以解决什么问题：

- 减少不必要的网络传输，节约带宽
- 更快的加载页面
- 减少服务器负载，避免服务过载的情况出现

强缓存：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。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`：`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。`Cache-Control:max-age=N`，有**max-age、s-maxage、no-cache、no-store、private、public**这六个属性。
  - 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

1. 基于 `last-modified` 的协商缓存

- 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。**

缺点：

- **因为是更具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。**

2. **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）**。

- 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。

有哈希值的文件设置强缓存即可。没有哈希值的文件（比如index.html）设置协商缓存

8. HTTP1.0和HTTP1.1的区别?

本题星球问答链接

1. 长连接

- `HTTP1.1` 支持长连接，每一个TCP连接上可以传送多个HTTP请求和响应，默认开启 `Connection:Keep-Alive`
- `HTTP1.0` 默认为短连接，每次请求都需要建立一个TCP连接。

2. 缓存

- `HTTP1.0` 主要使用 `If-Modified-Since/Expires` 来做为缓存判断的标准
- `HTTP1.1` 则引入了更多的缓存控制策略例如 `Entity tag / If-None-Match` 等更多可供选择的缓存头来控制缓存策略。

3. 管道化

- 基于 `HTTP1.1` 的长连接，使得请求管线化成为可能。管线化使得请求能够“并行”传输，但是响应必须按照请求发出的顺序依次返回，性能在一定程度上得到了改善。

4. 增加Host字段

使得一个服务器能够用来创建多个 Web 站点。

5. 状态码

新增了24个错误状态响应码

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
- `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)`

9. HTTP2.0与HTTP1.1的区别?

本题星球问答链接

1. 二进制分帧

在应用层 `（HTTP/2.0）` 和传输层 `（TCP or UDP）` 之间增加一个二进制分帧层，从而突破 `HTTP1.1` 的性能限制，改进传输性能，实现低延迟和高吞吐量。

2. 多路复用（MultiPlexing）

允许同时通过单一的 `HTTP/2` 连接发起多重的请求–响应消息，这个强大的功能则是基于“二进制分帧”的特性。

3. 首部压缩

`HTTP1.1` 不支持 `header` 数据的压缩，`HTTP/2.0` 使用 `HPACK` 算法对 `header` 的数据进行压缩，这样数据体积小了，在网络上传输就会更快。高效的压缩算法可以很大的压缩 `header` ，减少发送包的数量从而降低延迟。

4. 服务端推送 `（server push）`

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。

10. HTTPS的工作原理？（https是怎么建立连接的）

本题星球问答链接

1. 首先，客户端向服务器端发送请求报文，请求与服务端建立连接。
2. 服务端产生一**对公私钥**，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端发送过来的公钥进行加密，产生一个CA数字证书。
3. 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。
4. 客户端将服务端发送过来的数字证书进行解析(因为浏览器产商跟CA机构有合作，所以浏览器中已经保存了大部分CA机构的密钥，用于对服务端发送过来的数字证书进行解密)，验证这个数字证书是否合法，如果不合法，会发送一个警告。如果合法，取出服务端生成的公钥。
5. 客户端取出公钥并生成一个随机码key（其实就是对称加密中的密钥）
6. 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
7. 服务端接收到随机码key后，使用自己的私钥对它进行解密，然后获得到随机码key。
8. 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
9. 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加密传输数据，随机码Key作为传输的密钥。

11. HTTPS与HTTP的区别

本题星球问答链接

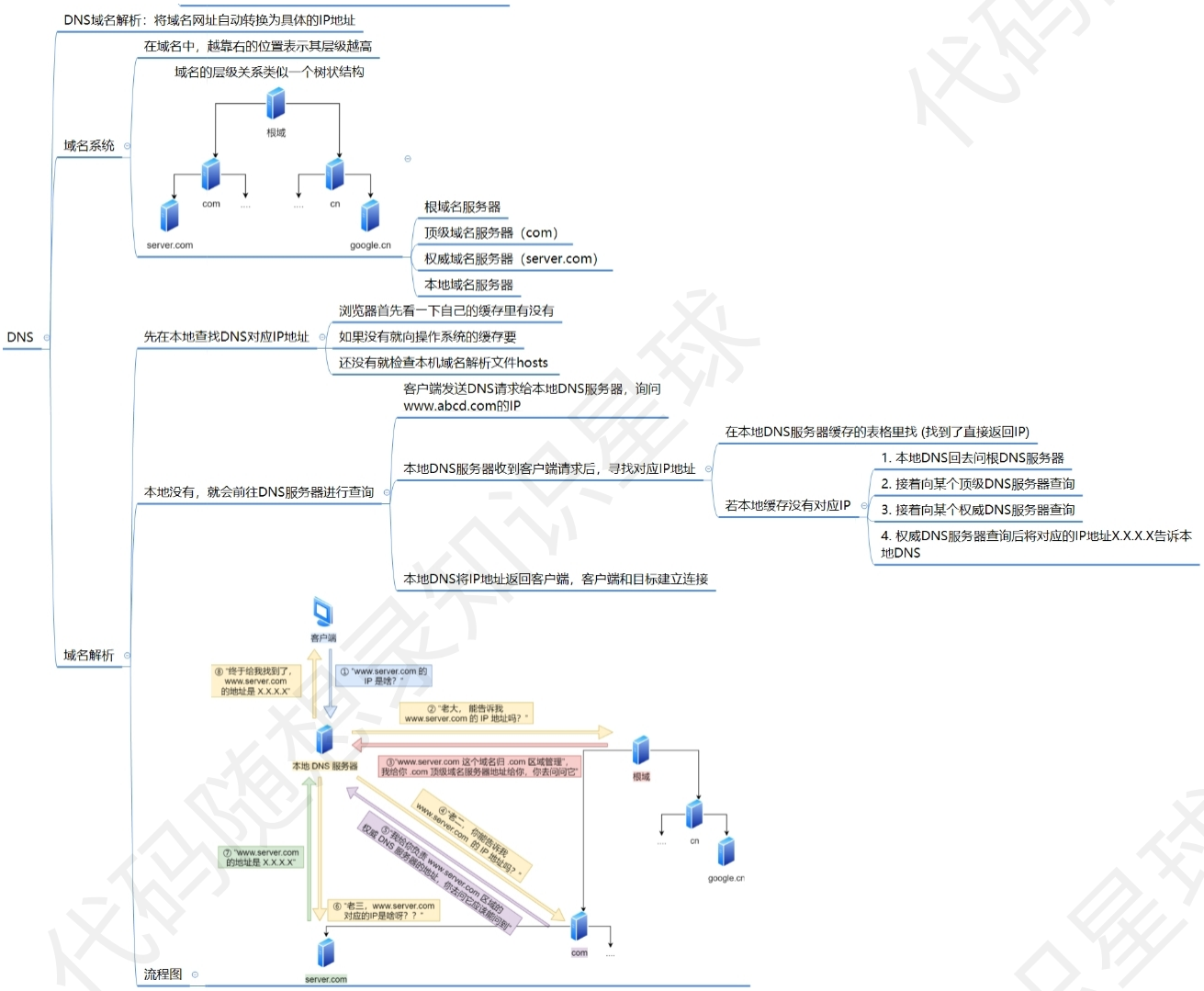
- HTTP 是明文传输，而HTTPS 通过 `SSL\TLS` 进行了加密
- HTTP 的端口号是 80，HTTPS 是 443
- HTTPS 需要到 `CA` 申请证书
- HTTP 的连接简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 HTTP 协议**安全**。

12. DNS是什么，及其查询过程

本题星球问答链接

DNS（Domain Name System）域名管理系统，是当用户使用浏览器访问网址之后，使用的第一个重要协议。DNS 要解决的是域名和 IP 地址的映射问题。

1. 首先用户在浏览器输入URL地址后，会先查询浏览器缓存是否有该域名对应的IP地址。
2. 如果浏览器缓存中没有，会去计算机本地的Host文件中查询是否有对应的缓存。
3. 如果Host文件中也没有则会向本地的DNS解析器（通常由你的互联网服务提供商（ISP）提供）发送一个DNS查询请求。
4. 如果本地DNS解析器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。
5. 本地DNS解析器接着向指定的顶级域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。
6. 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。
7. 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。



13. HTTP多个TCP连接怎么实现

本题星球问答链接

多个tcp连接是靠某些服务器对 `Connection: keep-alive` 的 `Header` 进行了支持。简而言之，完成这个 HTTP 请求之后，不要断开 HTTP 请求使用的 TCP 连接。这样的好处是连接可以被重新使用，之后发送 HTTP 请求的时候不需要重新建立 TCP 连接，以及如果维持连接，那么 SSL 的开销也可以避免。

14. TCP 的 Keepalive 和 HTTP 的 Keep-Alive 是一个东西吗?

本题星球问答链接

HTTP 的 Keep-Alive，是由应用层（用户态） 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 -> 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。

TCP 的 Keepalive，是由 TCP 层（内核态） 实现的，称为 TCP 保活机制；

通俗地说，就是TCP有一个定时任务做倒计时，超时后会触发任务，内容是发送一个探测报文给对端，用来判断对端是否存活。

15. TCP连接如何确保可靠性

本题星球问答链接

TCP连接确保可靠性方法如下：

1. 数据块大小控制： 应用数据被分割成TCP认为最合适发送的数据块，再传输给网络层，数据块被称为报文段或段。
2. 序列号： TCP给每个数据包指定序列号，接收方根据序列号对数据包进行排序，并根据序列号对数据包去重。
3. 校验和： TCP将保持它首部和数据的校验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到报文的检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段。
4. 流量控制： TCP连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP利用滑动窗口实现流量控制。
5. 拥塞控制： 当网络拥塞时，减少数据的发送。
6. 确认应答： 通过 `ARQ` 协议实现。基本原理是每发完一个分组就停止发送，等待对方确认。如果没收到确认，会重发数据包，直到确认后再发下一个分组。
7. 7、超时重传： 当TCP发出一个数据段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

16. 既然提到了拥塞控制，那你能说说说拥塞控制是怎么实现的嘛

本题星球问答链接

拥塞控制算法主要有以下几种：

1. 慢启动

在连接刚开始时，发送方会逐渐增加发送窗口大小，从而以指数增长的速度增加发送的数据量。

2. 拥塞避免

一旦慢启动阶段过去，发送方进入拥塞避免阶段。在这个阶段，发送方逐渐增加发送窗口的大小，但增加速率较慢，避免过快增加导致网络拥塞。

3. 超时重传

如果发送方在超时时间内未收到确认，它会认为数据包丢失，并重传这些数据包。这是拥塞控制的最后手段，用于检测和处理网络中的丢包或拥塞情况。当网络出现拥塞，也就是会发生数据包重传

4. 快速重传（Fast Retransmit）和快速恢复（Fast Recovery）

当发送方发送的数据包丢失或网络出现拥塞时，接收方会发送重复确认（duplicate ACK）通知发送方有数据包丢失。当发送方收到一定数量的重复确认时，它会立即重传丢失的数据包，而不是等待超时。这样可以减少网络的拥塞程度。

5. 拥塞窗口调整

发送方根据网络的拥塞程度动态调整发送窗口的大小，通过监测网络延迟和丢包情况来确定合适的发送速率，以避免网络拥塞。

17. Cookie和Session是什么？有什么区别?

`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

- 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。

- 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。

操作系统

1. 进程和线程的区别

本题星球问答链接

进程是系统进行资源分配和调度的基本单位。

线程 `Thread` 是操作系统能够进行运算调度的最小单位，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。

2. 进程调度算法你了解多少?

本题星球问答链接

进程调度算法是操作系统中用来**管理和调度进程（也称为任务或作业）执行的方法**。这些算法决定了在多任务环境下，如何为各个进程分配 `CPU` 时间，以实现公平性、高吞吐量、低延迟等不同的调度目标。

1. 先来先服务调度算法
   按照进程到达的先后顺序进行调度，即最早到达的进程先执行，直到完成或阻塞。
2. 最短作业优先调度算法
   优先选择运行时间最短的进程来运行
3. 高响应比优先调度算法
   综合考虑等待时间和服务时间的比率，选择具有最高响应比的进程来执行
4. 时间片轮转调度算法
   将 `CPU` 时间划分为时间片（时间量），每个进程在一个时间片内运行，然后切换到下一个进程。
5. 最高优先级调度算法
   为每个进程分配一个优先级，优先级较高的进程先执行。这可能导致低优先级进程长时间等待，可能引发饥饿问题。
6. 多级反馈队列调度算法
   将进程划分为多个队列，每个队列具有不同的优先级，进程在队列之间移动。具有更高优先级的队列的进程会更早执行，而长时间等待的进程会被提升到更高优先级队列。
7. 最短剩余时间优先

   每次选择剩余执行时间最短的进程来执行。
8. 最大吞吐量调度

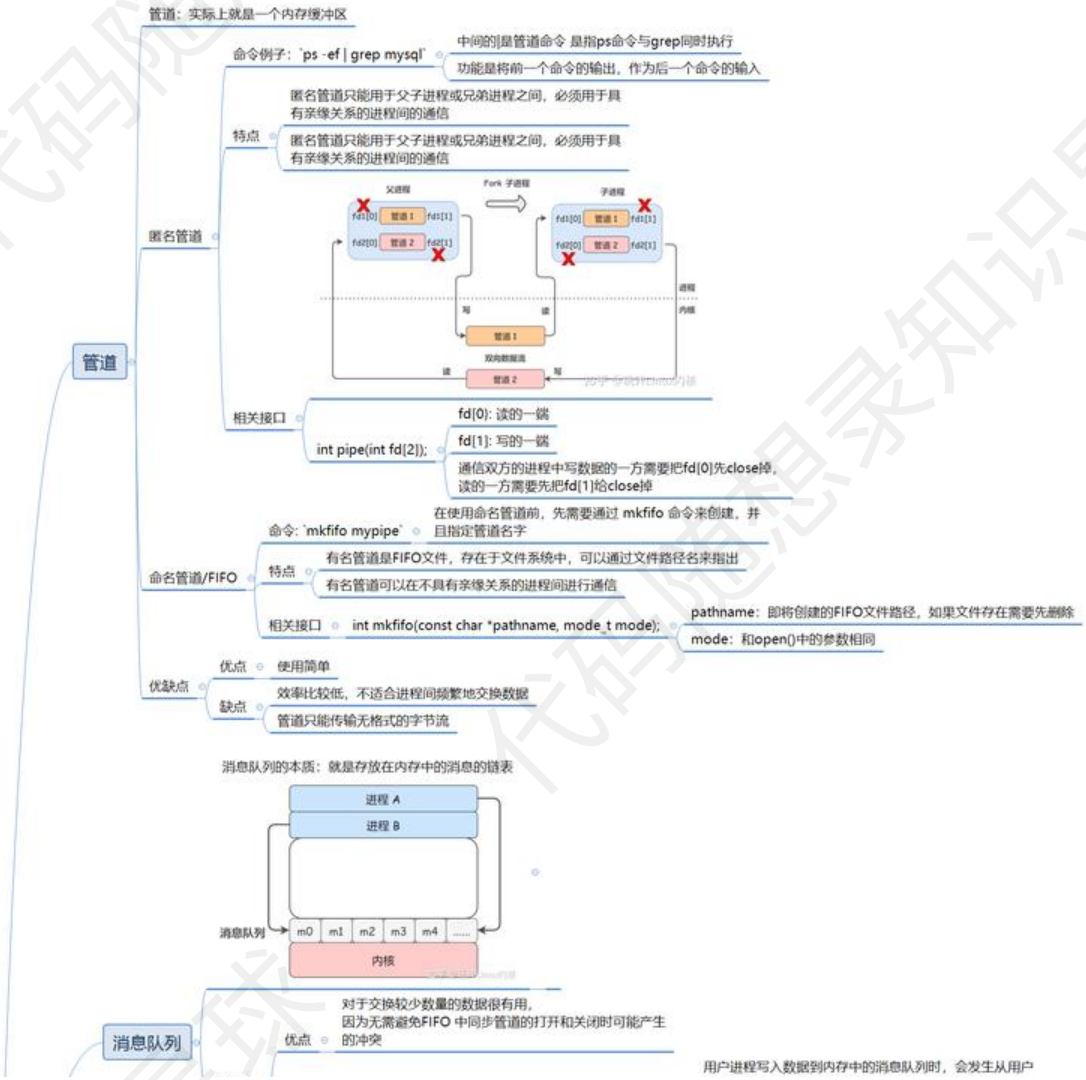
   旨在最大化单位时间内完成的进程数量
9. 最大吞吐量调度

   旨在最大化单位时间内完成的进程数量

3. 进程间有哪些通信方式

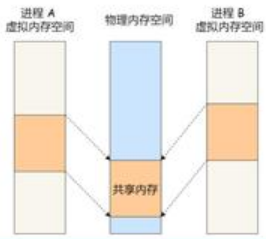
本题星球问答链接

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： 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。
3. **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。
4. **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。
5. 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。
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，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。
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。



- 进程间通信
  - 共享内存
    - 优缺点
      - 缺点
        - 消息队列的读取和写入的过程，都会有发生用户态与内核态之间的消息拷贝过程
          - 态拷贝数据到内核态的过程
          - 用户进程读取内存中的消息数据时，会发生从内核态拷贝数据到用户态的过程
          - 如果数据量较大，使用消息队列就会造成频繁的系统调用，即需要消耗更多的时间以便内核介入

共享内存的机制：就是拿出一块虚拟地址空间来，映射到相同的物理内存中，所有进程都可以访问共享内存中的地址



- 共享内存
  - 优点
    - 不需要像消息队列那样频繁的拷贝消息、进行系统调用，共享速度快
  - 优缺点
    - 缺点
      - 共享内存机制可能会发生冲突：如果多个进程同时修改同一个共享内存，先来的那个进程写的内容就会被后来的覆盖
- 信号量
  - 信号量定义
    - 操作系统提供的一种协调共享资源访问的方法，主要用于实现进程间的互斥与同步
    - 实际上就是一个整型计数器(sem)，用于表示资源的数量
  - 控制信号量的原子操作
    - P 操作：将 sem 减 1，相减后，如果 sem < 0，则进程进入阻塞等待；如果sem>=0，表明还有资源可用，进程可正常继续执行
    - V 操作：将 sem 加 1，相加后，如果 sem <= 0，唤醒一个等待中的进程；如果sem>0，表明当前没有阻塞中的进程
    - PV操作用于进入共享资源之前，V操作用于离开共享资源之后；两个操作必须成对出现
    - 信号量和 PV 操作具体的定义(伪代码)

```
// 信号量数据结构
type struct sem_t{
    int sem;   // 资源个数
    queue_t *q;// 等待队列
} sem_t;

// 初始化信号量
void init(sem_t *s, int sem) {
    s->sem = sem;
    queue_init(s->q);
}

// P操作
void P(sem_t *s) {
    s->sem--;
    if(s->sem < 0) { //表明该类资源已经分配完毕
        1. 保留调用线程 CPU 现场，
        2. 将该线程的 TCB 插入到 s 的等待队列，
        3. 设置该线程为等待状态，
        4. 执行调度程序，
    }
}

// V操作
void V(sem_t *s) {
    s->sem++;
    if(s->sem <= 0) { //表明还有进程在等待该类资源
        1. 移出 s 等待队列首元素，
        2. 将该线程的 TCB 插入就绪队列，
        3. 设置该线程为「就绪」状态，
    }
}
```

- 信号
  - 信号定义：信号是进程通信机制中唯一的异步通信机制，它可以在任何时候发送信号给某个进程，以迫使进程执行信号处理程序
  - 信号事件的来源
    - 硬件来源
      - Ctrl+C 产生 SIGINT 信号，表示终止该进程
      - Ctrl+Z 产生 SIGTSTP 信号，表示停止该进程，但还未结束
    - 软件来源
      - 输入kill命令如`kill -9 1111`，表示给 PID 为 1111 的进程发送 SIGKILL 信号，让其立即结束
- Socket
  - Socket概念
    - Socket可以完成跨网络与不同主机上的进程进行通信
    - Socket 的本质其实是一个API，它把复杂的 TCP/IP 协议族隐藏在 Socket 接口后面
  - 相关接口
    - int socket(int domain, int type, int protocal)
      - domain 参数：用来指定协议族
        - AF_INET用于IPV4
        - AF_INET6用于IPV6
        - AF_LOCAL / AF_UNIX用于本机
      - type 参数用来指定通信特性
        - SOCK_STREAM表示的是字节流，对应TCP
        - SOCK_DGRAM 表示的是数据报，对应UDP
        - SOCK_RAW表示的是原始套接字
      - protocal 参数原本是用来指定通信协议的，但现在基本废弃。因为协议已经通过前面两个参数指定完成，protocol目前一般写成0即可
    - 实现TCP字节流通信
      - socket类型是AF_INET和SOCK_STREAM

4. 什么是死锁？如何避免死锁？

本题星球问答链接

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。**

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。
- 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。

只需要破坏上面一个条件就可以破坏死锁。

- 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。

5. 什么是虚拟内存？为什么需要虚拟内存？

本题星球问答链接

虚拟内存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。

6. 什么是内存分段和分页？作用是什么？

本题星球问答链接

内存分段是将一个程序的内存空间分为不同的逻辑段 `segments` ，每个段代表程序的一个功能模块或数据类型，如代码段、数据段、堆栈段等。每个段都有其自己的大小和权限。

内存分页是把整个虚拟和物理内存空间分成固定大小的页(如4KB)。这样一个连续并且尺寸固定的内存空间，我们叫页 `Page`

作用：

1. 逻辑隔离： 内存分段和分页都实现了程序的逻辑隔离，使不同的功能模块或数据类型能够被单独管理和保护，提高了程序的可靠性和安全性。
2. 内存保护： 通过将不同的段或页面设置为只读、可读写、不可执行等权限，操作系统可以确保程序不会越界访问或修改其他段的内容，从而提高了系统的稳定性。
3. 虚拟内存： 分段和分页都有助于实现虚拟内存的概念，允许应用程序认为它们在使用的是一个比实际物理内存更大的内存空间。
4. 内存共享： 通过分页，操作系统可以实现内存页面的共享，从而节省内存空间，多个进程可以共享相同的代码或数据页面。
5. 内存管理： 分页更加灵活，允许操作系统将不同进程的页面分散存放在物理内存中，从而提高内存利用率。分段则更适用于管理不同的逻辑模块。

分段与分页的区别

- 分页对用户不可见，分段对用户可见
- 分页的地址空间是一维的，分段的地址空间是二维的
- 分页（单级页表）、分段访问一个逻辑地址都需要两次访存，分段存储中可以引入快表机制
- 分段更容易实现信息的共享和保护（纯代码或可重入代码可以共享）

分段与分页优缺点：

- 分页管理： 内存空间利用率高，不会产生外部碎片，只会有少量的页内碎片。但是不方便按照逻辑模块实现信息的共享和保护 。
- 分段管理： 很方便按照逻辑模块实现信息的共享和保护。但是**如果段长过大，为其分配很大的连续空间会很不方便，段式管理会产生外部碎片。**

7. 解释一下用户态和核心态

本题星球问答链接

用户态 `User Mode` 和核心态 `Kernel Mode` ，是操作系统中两种不同的执行模式，用于**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，用户态下的 CPU 不允许独占，也就是说 CPU 能够被其他程序获取。
- 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。处于内核态的 CPU 可以从一个程序切换到另外一个程序，并且占用 CPU 不会发生抢占情况，一般处于特权级 0 的状态我们称之为内核态。

8. 解释一下页面置换算法，例如LRU（最近最少使用）、FIFO（先进先出）等

本题星球问答链接

假设你的手机内存有限，只能同时运行四个原神的角色。当你想切换到一个新的角色时，你需要从内存中换出一个旧的角色，以便为新的角色腾出空间。不同的页面置换算法就相当于不同的切换策略，例如：

- **LRU（最近最少使用）算法**：每次选择最长时间没有被使用的角色进行切换。这种策略基于你对角色的喜好，认为最近被使用过的角色很可能还会被使用，而最久未被使用的角色很可能不会再被使用。LRU算法可以有效地减少切换次数，但是实现起来比较复杂，需要记录每个角色的使用时间或者维护一个使用顺序的列表。
- **FIFO（先进先出）算法**：每次选择最早进入内存的角色进行切换。这种策略很简单，只需要维护一个角色队列，每次淘汰队首的角色，然后把新的角色加入队尾。但是FIFO算法可能会淘汰一些经常被使用的角色，导致切换次数增加。而且FIFO算法有可能出现贝拉迪异常（Belady anomaly），即当分配给内存的空间增加时，切换次数反而增加。

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）

1. 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。
2. 先进先出算法：也就是**最先进入内存的页面最先被置换出去**。这个算法比较简单明了，就不过多解释了。但是先进先出算法会存在一个问题，就是Belady问题，即随着分配给进程的空闲页面数增加，缺页的情次反而也会增加。 这和我们常识是相悖的，因为我们通常认为如果一个进程经常发生缺页，那么就应该应该为他多分配一点内存。然而使用FIFO算法时，反而可能导致更多缺页情况出现。这就是Belady问题，Belady问题只会在使用FIFO算法时出现。
3. 最近最久未使用算法：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。**LRU算法的核心思想是，最近被访问的页面可能在未来被再次访问，而最长时间未被访问的页面可能是最不常用的，因此将其置换出去可以腾出空间给新的页面。LRU算法通常是使用一个数据结构去维护页面的使用历史，维护使用历史就是通过访问字段实现的。访问字段的位数和操作系统分配给该进程的页面数有关，比如分配4个页面，访问字段就是2位，16个页面，访问字段就是4位，依次类推。如此，每一个页面的访问字段都可以不同，通过访问字段的不同，我们就可以判断页面的使用历史。
4. 时钟算法：Clock算法基于一个环形链表或者循环队列数据结构来管理页面的访问情况，用于选择被置换的页面。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。这个访问过同样需要我们上面说到的访问字段来表示，此时访问字段只有一位。每个页面都与一个访问位相关联，标记该页面是否被访问过。

   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。

9. 解释一下进程同步和互斥，以及解决这些问题的方法

本题星球问答链接

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。**比如说，你想要和你的队友一起完成一个副本，你们需要相互配合，有时候等待对方的信号或者消息，有时候按照对方的要求执行某些动作，这就是进程同步。

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。比如说，你想要使用一个祭坛来祈愿，但是这个祭坛一次只能被一个人使用，如果有其他人也想要使用，他们就必须等待你使用完毕后再去使用，这就是进程互斥。

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。

举个例子，假设有一个信号量 s 表示一个祭坛是否可用，初始值为 1。如果 s 的值为 1，表示祭坛空闲；如果 s 的值为 0，表示祭坛被占用；如果 s 的值为 -1，表示有一个人在等待使用祭坛。那么我们可以用 PV 操作来实现对祭坛的互斥访问：

- 如果你想要使用祭坛，你就执行 P(s) 操作，将 s 的值减 1。如果结果为 0 或者正数，表示你可以使用祭坛；如果结果为负数，表示有人在使用祭坛，你就必须等待。
- 如果你使用完了祭坛，你就执行 V(s) 操作，将 s 的值加 1。如果结果为正数或者 0 ，表示没有人在等待使用祭坛；如果结果为负数，表示有人在等待使用祭坛，你就需要唤醒他们中的一个。

这样就可以保证每次只有一个人能够使用祭坛，实现了进程互斥。

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（Critical Section）： 将可能引发互斥问题的代码段称为临界区。为了实现互斥，每个进程在进入临界区前必须获取一个锁，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（Condition Variable）： 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。

10. 什么是中段和异常？它们有什么区别？

本题星球问答链接

中断和异常是两种不同的事件，它们都会导致CPU暂停当前的程序执行，转而去执行一个特定的处理程序。

中断和异常的区别主要有以下几点：

- **中断是由外部设备或其他处理器产生的**，它们通常是异步的，也就是说，它们可以在任何时候发生，与当前执行的指令无关。例如，键盘输入、鼠标移动、网络数据到达等都会产生中断信号，通知CPU去处理这些事件。
- **异常是由CPU内部产生的**，它们通常是同步的，也就是说，它们只会在执行某些指令时发生，与当前执行的指令有关。例如，除法运算时除数为零、访问非法内存地址、执行非法指令等都会产生异常信号，通知CPU去处理这些错误或故障。
- **中断可以被屏蔽或禁止**，这意味着CPU可以通过设置某些标志位或寄存器来忽略或延迟响应某些中断信号。这样可以避免中断过于频繁或干扰重要的任务。
- **异常不能被屏蔽或禁止**，这意味着CPU必须立即响应异常信号，并进行相应的处理。这样可以保证程序的正确性和系统的稳定性。

用原神举个例子：

中断就像是游戏中的一些突发事件，它们会打断你当前的任务或活动，让你去处理它们。例如，当你在探索地图时，可能会遇到一些敌人的袭击、一些隐藏的宝箱、一些突然出现的任务等，这些都会产生中断信号，让你去应对这些事件。你可以选择是否响应这些中断信号，也可以选择什么时候响应它们。

异常就像是游戏中的一些错误或故障，它们会导致你当前的任务或活动无法继续进行，让你去修复它们。例如，当你在进行战斗时，可能会遇到一些游戏卡顿、画面闪烁、角色卡死等，这些都会产生异常信号，让你去解决这些问题。你不能选择是否响应这些异常信号，也不能选择什么时候响应它们。

中断的定义：

CPU在执行指令时，收到某个中断信号转而去执行预先设定好的代码，然后再返回到原指令流中继续执行，这就是中断机制。

中断的作用：

设计中断机制的目的在于中断机制有以下4个作用，这些作用可以帮助操作系统实现自己的功能。这四个作用分别是：

1. 外设异步通知CPU： 外设发生了什么事情或者完成了什么任务或者有什么消息要告诉CPU，都可以异步给CPU发通知。
2. CPU之间发送消息： 在SMP系统中，一个CPU想要给另一个CPU发送消息，可以给其发送IPI(处理器间中断)。
3. 处理CPU异常： CPU在执行指令的过程中遇到了异常会给自己发送中断信号来处理异常。例如，做整数除法运算的时候发现被除数是0，访问虚拟内存的时候发现虚拟内存没有映射到物理内存上。
4. 实现系统调用： 早期的系统调用就是靠中断指令来实现的，后期虽然开发了专用的系统调用指令，但是其基本原理还是相似的。

中断的产生

那么中断信号又是如何产生的呢？中断信号的产生有以下4个来源：

- 外设：外设产生的中断信号是异步的，一般也叫做硬件中断(注意硬中断是另外一个概念)。硬件中断按照是否可以屏蔽分为可屏蔽中断和不可屏蔽中断。例如，网卡、磁盘、定时器都可以产生硬件中断。
- CPU: 这里指的是一个CPU向另一个CPU发送中断，这种中断叫做IPI(处理器间中断)。IPI也可以看出是一种特殊的硬件中断，因为它和硬件中断的模式差不多，都是异步的。
- CPU异常，CPU在执行指令的过程中发现异常会向自己发送中断信号，这种中断是同步的，一般也叫做软件中断(注意软中断是另外一个概念)。 CPU异常按照是否需要修复以及是否能修复分为3类
    - 陷阱(trap)，不需要修复，中断处理完成后继续执行下一条指令，
    - 故障(fault)，需要修复也有可能修复，中断处理完成后重新执行之前的指令
    - 中止(abort)，需要修复但是无法修复，中断处理完成后，进程或者内核将会崩溃。
- 中断指令，直接用CPU指令来产生中断信号，这种中断和CPU异常一样是同步的，也可以叫做软件中断。

11. 介绍一下几种典型的锁

本题星球问答链接

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。

12. 你知道的线程同步的方式有哪些?

本题星球问答链接

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：

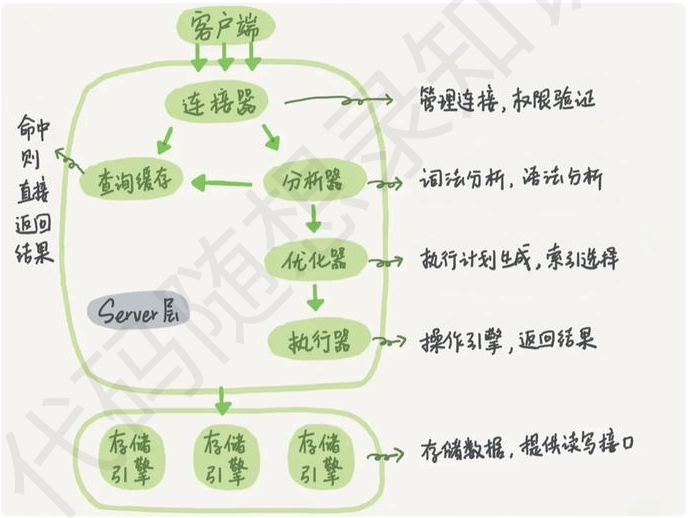
1. 互斥锁：互斥锁是最常见的线程同步机制。它允许只有一个线程同时访问被保护的临界区（共享资源）
2. 条件变量：条件变量用于线程间通信，允许一个线程等待某个条件满足，而其他线程可以发出信号通知等待线程。通常与互斥锁一起使用。
3. 读写锁： 读写锁允许多个线程同时读取共享资源，但只允许一个线程写入资源。
4. 信号量：用于控制多个线程对共享资源进行访问的工具。

数据库

1. 一条SQL查询语句是如何执行的?

本题星球问答链接

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。



MySQL 的逻辑架构图

2. 说一说事物隔离级别

本题星球问答链接

当数据库上有多个事务同时执行的时候，就可能出现**脏读（dirty read）、不可重复读（non-repeatable read）、幻读（phantom read）**的问题，为了解决这些问题，就有了“隔离级别”的概念。

SQL标准的事务隔离级别包括：**读未提交（read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（serializable ）**

- 读未提交是指，一个事务还没提交时，它做的变更就能被别的事务看到。
- 读提交是指，一个事务提交之后，它做的变更才会被其他事务看到。
- 可重复读是指，一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的。当然在可重复读隔离级别下，未提交变更对其他事务也是不可见的。
- 串行化，顾名思义是对于同一行记录，“写”会加“写锁”，“读”会加“读锁”。当出现读写锁冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。

3. 事务的四大特性有哪些?

本题星球问答链接

ACID事务 四大特性

1. **原子性**：事务是不可分割的最小工作单元，一个事务对应一个完整的业务。**一个事务的所有操作要么全部完成要么全部没完成**，不能停留在中间状态，如果事务执行过程中出现错误就会被回滚到原来的状态。
2. **一致性:** 一个事务执行之前和执行之后都必须处于一致性状态。比如a与b账户共有1000块，两人之间转账之后无论成功还是失败，它们的账户总和还是1000。
3. **隔离性:** 允许多个并发的事务同时对数据修改和读取，执行互不干扰，防止多个事务并发执行由于交叉执行造成数据不一致的情况。和隔离的级别有关, 例如read和commitied, 一个事务只能读到已经提交的修改。
4. **持久性:** 一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

4. 索引有哪些种类

本题星球问答链接

从数据结构维度进行分类:

- B+树索引:所有数据存储在叶子节点，复杂度为O(logn)，适合范围查询。
- 哈希索引:适合等值查询，检索效率高，一次到位
- 全文索引: `MyISAM` 和 `InnoDB` 中都支持使用全文索引，一般在文本类型char，text，varchar类型上创建
- `R-Tree` 索引:用来对 `GIS` 数据类型创建 `SPATIAL` 索引

从物理存储维度进行分类:

- 聚集索引:数据存储与索引一起存放，叶子节点会存储一整行记录，找到索引也就找到了数据。
- 非聚集索引:数据存储与索引分开存放，叶子节点不存储数据，存储的是数据行地址。

从逻辑维度进行分类:

- 主键索引:一种特殊的唯一索引，不允许有空值。
- 普通索引:MySQL中基本索引类型，允许空值和重复值
- 联合索引:多个字段创建的索引，使用时遵循最左前缀原则
- 唯一索引:索引列中的值必须是唯一的，但是允许为空值
- 空间索引:MySQL5.7之后支持空间索引，在空间索引这方面遵循OpenGIS几何数据模型规则。

5. MySQL什么使用B+树来作索引，它的优势什么?

本题星球问答链接

1. 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，因为每个节点即存索引又存记录，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，数据量相同的情况下，B+树的非叶子节点可以存放更多的索引，查询底层节点的磁盘 I/O次数会更少。
2. 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。
3. 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，范围查询效率不如 B+ 树。B+ 树的插入和删除效率更高。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。

6. 什么时候需要创建索引?

本题星球问答链接

- 表的主关键字：自动建立唯一索引
- 直接条件查询的字段：经常用于WHERE查询条件的字段，这样能够提高整个表的查询速度
- 查询中与其它表关联的字段：例如字段建立了外键关系
- 查询中排序的字段：排序的字段如果通过索引去访问将大大提高排序速度
- 唯一性约束列： 如果某列具有唯一性约束，那么为了确保数据的唯一性，可以在这些列上创建唯一索引。
- 大表中的关键列： 在大表中，如果查询的效率变得很低，可以考虑在关键列上创建索引。

7. 什么时候不需要创建索引?

本题星球问答链接

- 小表： 对小表创建索引可能会带来额外的开销，因为在小数据集中扫描整个表可能比使用索引更快。
- 频繁的插入、更新和删除操作： 索引的维护成本会随着数据的插入、更新和删除操作而增加。如果表经常被修改，过多的索引可能会影响性能。
- 数据重复且分布平均的表字段：假如一个表有10万行记录，性别只有男和女两种值，且每个值的分布概率大约为50%，那么对这种字段建索引一般不会提高数据库的查询速度。
- 很少被查询的列： 如果某列很少被用于查询条件，那么为它创建索引可能没有明显的性能提升。
- 查询结果总行数较少的表： 如果查询的结果集总行数很少，使用索引可能不会有太大的性能提升。

8. 说一说你了解的MVCC机制

本题星球问答链接

MVCC（Multi-Version Concurrency Control）多版本并发控制，用于管理多个事务同时访问和修改数据库的数据，而不会导致数据不一致或冲突。MVCC的核心思想是每个事务在数据库中看到的数据版本是事务开始时的一个快照，而不是实际的最新版本。这使得多个事务可以并发执行，而不会互相干扰。

MySQL的事务有ACID四大特性，其中的隔离性可以通过锁和MVCC来实现，MVCC适合在一些锁性能较为差的情况下使用，提高效率。

如何实现：

每一个 `UndoLog` 日志中都有一个 `roll_pointer` （回滚指针）用于指向上一个版本的 `Undo Log` 。这样对于每一条记录就会构成一个版本链，用于记录所有的修改，每一次进行新的修改后，新的 `Undo Log` 会放在版本链的头部。

在我们进行查询的时候应该查询哪个版本呢？这时候就可以通过 `ReadView` 来实现。

在事务SELECT查询数据时，就会构造一个 `ReadView` ，它包含了版本链的统计信息

- `m_ids` 当前活跃的所有事务id（所有未提交的事务）
- `min_trx_id` 版本链尾的id
- `max_trx_id` 下一个将要分配的事务id（版本链头事务id+1）
- `creator_trx_id` 创建这个ReadView的事务的id 查询规则：

该版本是否为当前事务创建（读取自己修改的数据），如果是就返回，否则进入下一个判断

该版本的事务id是否小于min_trx_id（在ReadView创建之前，数据已经提交），可以直接访问

该版本的事务id是否大于max_trx_id（在ReadView创建后，该版本才开启），不能被访问

该版本事务id在[min_trx_id, max_trx_id]之间，则判断当前版本事务id是否在m_ids中，如果不在，说明事务已经提交可以访问，否则不能访问。

9. 索引失效的场景有哪些

本题星球问答链接

- OR 条件： 当查询中使用多个 OR 条件时，如果这些条件不涉及同一列，索引可能无法有效使用。数据库可能会选择全表扫描而不是使用多个索引。
- 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。
- 使用通配符前缀搜索： 在使用通配符前缀（如 `LIKE 'prefix%'` ）进行搜索时，大多数索引无法使用，因为索引通常是按照列的完整值进行排序的。
- 不等号条件： 当查询中包含不等号条件（如>,<,>=,<=）时，索引可能会失效。通常情况下，索引只能用于等值比较。
- 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。

10. MySQL的执行引擎有哪些?

本题星球问答链接

主要有MyISAM、InnoDB、Memery等引擎：

- `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。
- `MyISAM` 引擎不支持事务，也不支持行级锁和外键约束。
- `Memery` 就是将数据放在内存中，数据处理速度很快，但是安全性不高。

11. MySQL日志文件有哪几种?

本题星球问答链接

- `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。
- `relay log` 中继日志，用于主从复制场景下， `slave` 通过io线程拷贝master的 `bin log` 后本地生成的日志

12. MySQL有哪些锁? 作用是什么?

本题星球问答链接

1. 全局锁

全局锁主要应用于做**全库逻辑备份**，这样在备份数据库期间，不会因为数据或表结构的更新，而出现备份文件的数据与预期的不一样，加上全局锁，意味着整个数据库都是只读状态。

2. 表级锁

- 元数据锁（MDL）：对数据库表进行操作时，会自动给这个表加上元数据锁，为了保证当用户对表执行 CRUD 操作时，其他线程对这个表结构做了变更。元数据锁在事务提交后才会释放。
- 意向锁：对某些记录加上共享锁之前，需要先在表级别加上一个意向共享锁，对某些纪录加上独占锁之前，需要先在表级别加上一个意向独占锁。普通的 select 是不会加行级锁的，普通的 select 语句是利用 MVCC 实现一致性读，是无锁的。
- AUTO-INC 锁：表里的主键通常都会设置成自增的，之后可以在插入数据时，可以不指定主键的值，数据库会自动给主键赋值递增的值通过 **AUTO-INC 锁**实现的。**在插入数据时，会加一个表级别的 AUTO-INC 锁**，然后为被 AUTO_INCREMENT 修饰的字段赋值递增的值，等插入语句执行完成后，才会把 AUTO-INC 锁释放掉。其他事务的如果要向该表插入语句都会被阻塞，从而保证插入数据时字段的值是连续递增的。

3. 行锁

- 记录锁：锁住的是一条记录，记录锁分为排他锁和共享锁。
- 间隙锁：只存在于可重复读隔离级别，目的是为了解决可重复读隔离级别下幻读的现象。间隙锁之间是兼容的，两个事务可以同时持有包含共同间隙范围的间隙锁，并不存在互斥关系。
- Next-Key Lock：Next-Key Lock临键锁，是 Record Lock + Gap Lock 的组合，锁定一个范围，并且锁定记录本身。next-key lock 即能保护该记录，又能阻止其他事务将新纪录插入到被保护记录前面的间隙中。
- 插入意向锁：一个事务在插入一条记录的时候，需要判断插入位置是否已被其他事务加了间隙锁（next-key lock 也包含间隙锁）。如果有的话，插入操作就会发生阻塞，直到拥有间隙锁的那个事务提交为止，在此期间会生成一个插入意向锁，表明有事务想在某个区间插入新记录，但是现在处于等待状态。