自己的

LAZADA

数据结构

1. 堆排序的时间复杂度，怎么实现的，和其他排序的区别？

   两个特征

   - 堆中某个节点的值总是不大于或者不小于其父节点的值
   - 堆总是一棵完全二叉树（每个节点的编号和满二叉树的编号一一对应，那就是完全二叉树）

   建堆的步骤：

   - 先建立一棵二叉树，再进行堆化（从最后一个非叶子节点开始进行调整，将最大值或者最小值上浮）
   - 排序，交换堆顶元素和最后一个元素的位置，缩小堆的范围
   - 重复执行上一步

   时间复杂度$O(nlogn)$，空间复杂度$O(1)$， n是待排序列的大小，不需要额外的空间。

2. AVL树讲一下，怎么进行平衡，平衡的方法？

   AVL是两个数学家的名字。AVL树是一棵自平衡的二叉搜索树。**平衡二叉树是一个特殊的二叉搜索树。**在每个节点上通过维护一个平衡因子来保持树的平衡。平衡因子是指该节点的左子树高度减去右子树高度的值。如果绝对值大于1，就说明不平衡。

   当AVL树进行插入或删除操作时，会在插入或删除节点后，从被修改的节点开始向上遍历，检查每个节点的平衡因子（每个节点有个自己的高度）。如果发现某个节点的平衡因子超出允许范围，就意味着该节点及其祖先节点可能不平衡。

   调整方式有四种

   - 右单旋转：在某一个节点的左子树的左子树上增删
   - 左单旋转 ： 在右子树的右子树上增删
   - 先左后右双旋转：在左子树的右子树上增删
   - 先右后左双旋转：在右子树的左子树上增删

3. 256MB的内存对10G的数据进行排序？

   使用外部排序方法，用于处理大型数据集的排序算法，先划分成适应内存容量的块，并在内存和磁盘之间进行多次读取和写入。

   1. 将10GB的数据划分为若干个能够适应内存的块（比如每个块的大小为256MB）。
   2. 依次将这些块加载到内存中，并使用合适的排序算法（如快速排序或归并排序）对每个块进行排序。
   3. 将排序后的块写回磁盘，并在磁盘上创建一个索引来记录每个块的位置和顺序。
   4. 依次读取每个块的第一个元素到内存中，并进行归并操作，选择最小的元素，并将其写入输出文件。
   5. 重复上述步骤，直到所有的块都被处理完毕，并且输出文件中包含了所有的元素。

4. OSI七层模型及其对应的协议？

OSI七层模型

OSI定义了网络互连的七层模型（物理层、数据链路层、网络层、传输层、会话层、表示层、应用层），如下图所示：

OSI 七层模型			
层级	层	英文全称	常用协议
7	应用层	Application Layer	HTTP、FTP、SMTP、POP3、TELNET、NNTP、IMAP4、FINGER
6	表示层	Presentation Layer	LPP、NBSSP
5	会话层	Session Layer	SSL、TLS、DAP、LDAP
4	传输层	Transport Layer	TCP、UDP
3	网络层	Network Layer	IP、ICMP、RIP、IGMP、OSPF
2	数据链路层	Data Link Layer	以太网、网卡、交换机、PPTP、L2TP、ARP、ATMP
1	物理层	Physical Layer	物理线路、光纤、中继器、集线器、双绞线

应用层：为应用程序或用户请求提供各种请求服务

表示层：数据编码，格式转换，数据加密

会话层：创建管理和维护会话

传输层：数据通信

网络层：IP选址和路由选择（RIP route information protocol和OSPF open shortest path first，前者是一种距离矢量路由协议，用距离作为度量标准，通过交换路由信息计算达到目的地的最短路径，每30s向相邻路由器发送路由更新信息，最大是15跳，用于小型网络；OSPF是一种链路状态路由协议，使用Dijkstra算法计算，使用链路状态作为度量标准，通过触发更新，交换链路信息构建邻居关系，支持更大规模的网络）

数据链路层：提供介质访问和链路管理

物理层：管理通信设备和网络媒体之间的互联互通

5. HTTP和HTTPS的区别？摘要算法怎么生成的？SSL的四次握手过程？

   窃听，篡改，冒充，端口号不同，对称加密（AES），非对称加密（RSA），哈希函数（SHA）。

   摘要算法：MD5，SHA-1等

   1. 初始化：选择适当的摘要算法并进行初始化。
   2. 数据输入：将待摘要的数据按照规定的方式输入到摘要算法中。
   3. 摘要计算：摘要算法根据输入的数据进行计算，并生成固定长度的摘要。
   4. 输出摘要：将生成的摘要作为结果输出。

   SSL（Secure Sockets Layer）：包括握手和记录

   1. 第一次握手：客户端向服务器发送一个SSL版本号和加密套件列表，以及一个随机数作为客户端随机数。
   2. 第二次握手：服务器收到客户端的请求后，选择一个SSL版本号和加密套件，发送给客户端，并生成一个随机数作为服务器随机数。
   3. 第三次握手：客户端收到服务器的响应后，验证服务器的数字证书，并生成一个随机数作为主密钥预备密钥（Pre-Master Secret），通过服务器的公钥加密后发送给服务器。

4. 第四次握手：服务器使用自己的私钥解密客户端发送的预备密钥，然后双方使用客户端随机数、服务器随机数和预备密钥生成主密钥。此后，客户端和服务器都拥有相同的主密钥，用于加密和解密通信数据

计算机网络

1. 什么是缺页中断?

   缺页中断（Page Fault）是指当程序访问一个尚未分配物理内存的虚拟内存页面时，操作系统发生的一种中断。在虚拟内存管理中，将物理内存划分成固定大小的页面（Page），而程序使用的地址空间被分割成相同大小的虚拟页面。当程序访问某个虚拟页面时，操作系统需要将该页面加载到物理内存中，如果该页面尚未加载到物理内存，则会触发缺页中断。

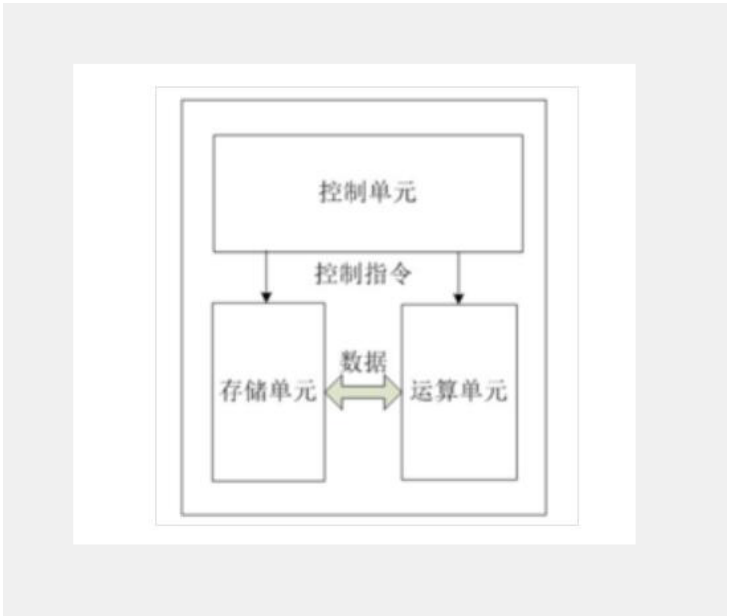
2. 进程的调度算法?

   1. 先来先服务调度（FCFS）：按照进程到达的顺序进行调度，先到达的进程先执行。该算法简单直观，但可能导致长作业等待时间较长，无法适应实时性要求高的场景。
   2. 最短作业优先调度（SJF）：选择执行时间最短的进程先执行。该算法能够最小化平均等待时间，但需要预先知道每个进程的执行时间。
   3. 优先级调度算法：为每个进程分配一个优先级，优先级高的进程先执行。可以根据进程的特性、重要性等进行优先级赋值。但可能导致低优先级进程长时间等待。
   4. 时间片轮转调度算法（Round Robin）：将CPU时间划分为固定大小的时间片，每个进程按照时间片顺序执行，当时间片用完后，切换到下一个进程。能够平均分配CPU时间，但可能导致长时间的等待和响应延迟。
   5. 最短剩余时间优先调度（SRTF）：根据进程剩余执行时间选择最短的进程执行。在SJF的基础上进行动态调度，能够适应实时性要求高的场景。
   6. 多级反馈队列调度算法：将进程划分为多个优先级队列，每个队列具有不同的时间片大小。新到达的进程进入最高优先级队列，如果时间片用完仍未执行完，则降低优先级，进入下一级队列。适用于多种类型的进程，能够平衡响应时间和吞吐量

操作系统

1. CPU的模块划分？怎么执行的1+1=2?

   主要就是控制存储和运算



- 控制单元
- 算术逻辑单元
- 总线接口单元：BIU负责与主存储器和其他外部设备之间的数据传输和通信，包括指令读取、数据读写等操作。
- 寄存器

- 数据缓存和指令缓存

关于执行1+1=2的过程，以下是一个简化的说明：

1. 程序中的1+1表达式被编译成机器指令，并存储在内存中。
2. 控制单元从内存中读取指令，并将其送入指令寄存器（IR）中。
3. 控制单元解码指令，确定所需的操作和操作数。在这个例子中，识别到是加法操作，并将操作数1和操作数2（均为1）发送给ALU。
4. ALU接收操作数1和操作数2，并执行加法运算。ALU将运算结果（2）存储在累加器（ACC）中。
5. 最终的结果（2）可以进一步存储在寄存器或内存中，以供后续的计算和处理使用。

毫末智行

1. 渲染管线

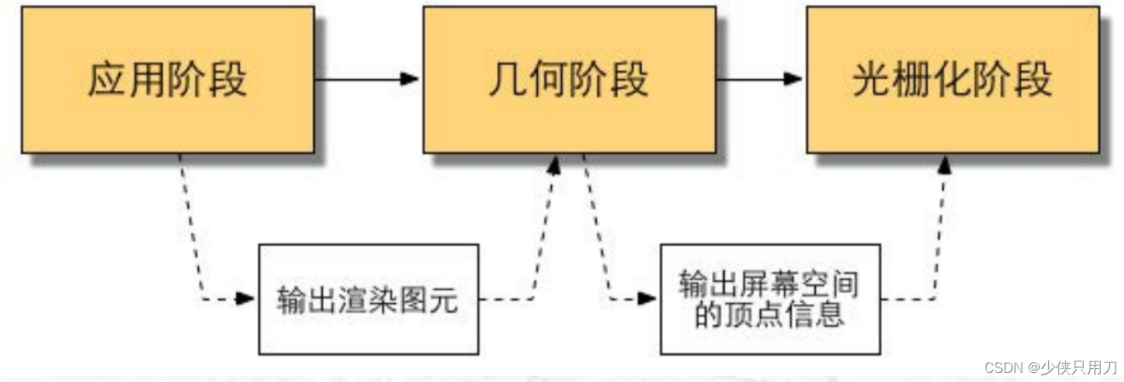
   渲染管线（Rendering Pipeline）是指计算机图形学中用于将三维场景转化为最终图像的一系列处理步骤。它是图形渲染的核心流程，包含多个阶段，每个阶段负责不同的任务，以最终生成可视化的图像。

   传统的渲染管线通常分为图元处理阶段和图像处理阶段两部分，下面是它们的简要介绍：

   1. 图元处理阶段（Geometry Processing Stage）：
      - 输入阶段（Input Stage）：从CPU传递三维模型数据、纹理坐标等输入信息到GPU。
      - 顶点着色器（Vertex Shader）：对输入的顶点进行变换、光照计算、纹理坐标处理等。
      - 几何着色器（Geometry Shader）：可选阶段，用于对图元（如点、线、三角形）进行进一步的处理和变换。
      - 图元装配（Primitive Assembly）：将顶点组装成图元（如三角形）。
      - 光栅化（Rasterization）：将图元转化为像素，并计算像素的位置、深度等信息。
   2. 图像处理阶段（Pixel Processing Stage）：
      - 像素着色器（Pixel Shader）：对每个像素进行着色，包括纹理采样、光照计算、阴影计算等。
      - 像素操作（Pixel Operations）：执行像素级别的操作，如深度测试、模板测试、混合操作等。
      - 输出合并（Output Merging）：合并最终的像素颜色、深度等信息。
      - 帧缓冲（Frame Buffer）：将最终的像素数据写入帧缓冲区，生成最终的图像。

   渲染管线的每个阶段都有各自的功能和任务，通过流水线的方式将三维场景的处理流程分解为多个可并行执行的阶段，从而提高图形渲染的效率。现代图形API（如OpenGL和DirectX）通常提供对渲染管线的灵活配置和扩展，使开发者可以根据需求进行优化和定制，实现各种图形效果和渲染技术。

   渲染管线（渲染流水线）是将三维场景模型转换到屏幕像素空间输出的过程。图形渲染管线接受一组3D坐标，然后把它们转变为屏幕上的有色2D像素输出。



   流水线可抽象为三个阶段：应用阶段、几何阶段、光栅化阶段。

- **应用阶段**：这是一个由开发者完全控制的阶段，在这一阶段将进行数据准备，并通过CPU向GPU输送数据，例如顶点数据、摄像机位置、视锥体数据、场景模型数据、光源等等；此外，为了提高渲染性能，还会对这些数据进行处理，比如剔除不可见物体；最后还要设置每个模型的渲染状态，这些渲染状态包括但不限于所使用的材质、纹理、shader等。**这一阶段最重要的输出是渲染所需的几何信息，即渲染图元，通俗来讲渲染图元可以是点、线、面等。**
- **几何阶段**：几何阶段运行在GPU中，几何阶段用于处理我们要绘制的几何相关事情，它和每个渲染图元打交道。几何阶段最重要的任务是将顶点坐标变换到屏幕空间中。后面会对几何阶段进行更详细的表述。
- **光栅化阶段**：光栅化阶段运行在GPU中，其主要任务是决定每个渲染图元中哪些像素应该被绘制在屏幕上，它需要对上一阶段得到的逐顶点数据进行插值，然后进行逐像素处理。

应用阶段：开发者控制，起点是CPU，CPU和GPU的通信即是应用阶段。

- 加载

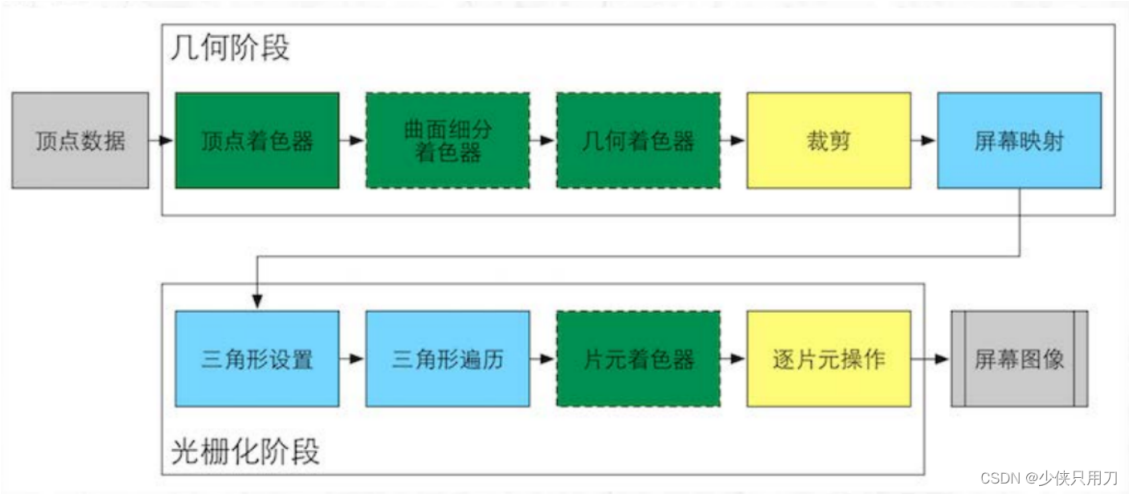
  大多数显卡没有直接访问RAM的能力，将数据加载到显存中使GPU能更快的访问这些数据。当把数据加载到显存后，内存中的数据便可以释放了，但对于一些还需要使用的数据则需要继续保留在内存中，如CPU需要网格数据进行碰撞检测。

- 设置渲染状态

  渲染状态的一个通俗解释就是，定义了场景中的网格是怎样被渲染的。例如，使用哪个顶点着色器/片段着色器、光源属性、材质等。如果不设置渲染状态，那所有的网格将使用同一种渲染，显然这是不希望得到的结果

- 调用Draw Call进行渲染

  当所有的数据准备好后，CPU就需要调用一个渲染指令告诉GPU，按照上述设置进行渲染，这个渲染命令就是Draw Call。Draw Call命令仅仅会指向一个需要被渲染的图元列表，而不包含任何材质信息，因为这些信息已经在上一个阶段中完成。执行DrawCall后GPU就会按照渲染流水线进行渲染计算，并输出到显示设备中，所执行的操作便是下述GPU渲染管线的内容。



几何阶段：

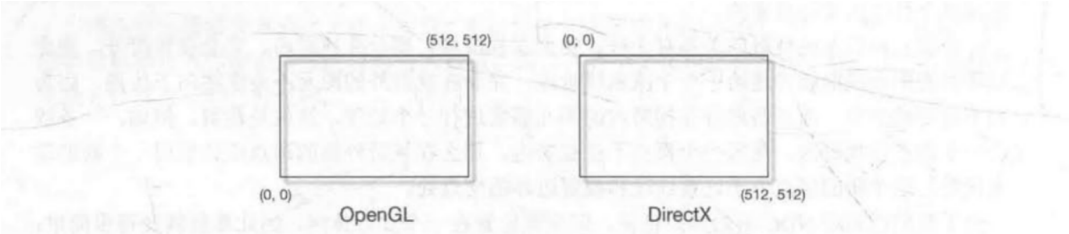
绿色是可编程部分。GPU的很多小核心上的程序就是shader。

- **顶点数据**：顶点数据是渲染流水线的输入，顶点数据包括顶点坐标、法线、切线、顶点颜色、纹理坐标等
- **顶点着色器**：顶点着色器的处理单位是顶点，即对于输入的每个顶点都会调用一次顶点着色器。**顶点着色器主要功能是进行坐标系变换操**作，所输入的顶点坐标等位于模型局部坐标空间，在这一阶段需要将顶点坐标数据变换为到齐次裁剪空间。当顶点坐标被变换到齐次裁剪空间后，通常再由硬件做透视除法，最终得到归一化的设备坐标(NDC)。

- **曲面细分着色器(可选)**：曲面细分着色器是一个可选的阶段。曲面细分是利用镶嵌化处理技术**对三角形进行细分**，以此来增加物体表面的三角面数量。
- **几何着色器(可选)**：几何着色器也是一个可选的阶段。顶点着色器以顶点数据作为输入，**而几何着色器则以完整的图元(Primitive)作为输入数据。**例如，以三角形的三个顶点作为输入，然后输出对应的图元。与顶点着色器不能销毁或创建顶点不同，**几何着色器的主要亮点就是可以创建或销毁几何图元**，此功能让GPU可以实现一些有趣的效果。例如，根据输入图元类型扩展为一个或更多其他类型的图元，或者不输出任何图元。需要注意的是，几何着色器的输出图元不一定和输入图元相同。几何着色器的一个拿手好戏就是将一个点扩展为一个四边形(即两个三角形)。
- **裁剪（硬件完成）**：裁剪操作就是将相机看不到的物体、顶点剔除，使其不被下一阶段处理。只有当图元完全位于视锥体内时，才会将它送到下一阶段，对于部分位于视锥体内的图元，外部的顶点将被剔除掉。由于已经知道在NDC下的顶点位置（即顶点位置在一个立方体内），因此裁剪就变得简单：只需要将图元裁剪到单位立方体内。裁剪这一步骤是硬件的固定操作，因此是不可编程的。
- **屏幕映射（硬件完成）**：这一步输入的坐标仍是三维坐标（范围在单位立方体内），屏幕映射的任务就是将每个图元的x、y值变换到屏幕坐标系（屏幕坐标系是一个2D空间）。由于输入坐标范围在[-1,1]，因此这是一个拉伸到屏幕分辨率大小的过程。对于输入的坐标z值不做任何处理（实际上屏幕坐标系和z坐标一起构成窗口坐标系），这些值会被一起传递到光栅化阶段。

  屏幕坐标系在OpenGL和DirectX中的定义方式不同。

  OpenGL的原点在笛卡尔坐标系的原点，DirextX与之关于x轴对称。



▲图 2.11　OpenGL 和 DirectX 的屏幕坐标系差异。对于一张 512*512 大小的图像，在 OpenGL 中其（0, 0）点在左下角，而在 DirectX 中其(0, 0)点在左上角

- **三角形设置**：光栅化第的第一个流水线阶段是三角形设置，这个阶段会计算光栅化一个三角形所需的信息。说白了就是离散化，生成三角形信息，为将来找到覆盖的元素做准备
- **三角形遍历**：检查是否覆盖，如果覆盖，覆盖部分就有一个片元，这就是找像素的过程。并且对于覆盖区域的深度进行插值。
- **片元着色器**：前面的光栅化阶段实际上并不会影响每个像素的颜色值。

  片元着色器的输入是上一个阶段对顶点信息进行插值的结果（是根据从顶点着色器输出的数据插值得到的），而它的输出是像素颜色值。这一阶段可以完成很多重要的渲染技术，**其中最重要的技术有纹理采样、逐片光照计算等，覆盖片元的纹理坐标是通过前述的阶段的顶点数据插值得到的。**
- **逐片元操作（输出合并）**：
  - 决定每个片元的可见性，这涉及到很多测试功能，例如**深度测试、模板测试。**
    - **模板测试**与之相关的是**模板缓冲**，模板测试通常用来限制渲染的区域，渲染阴影，轮廓渲染等。无论一个片元有没有经过模板测试，都可以根据模板测试和下面的深度测试结果来修改模板缓冲区。

      stencil buffer

        - 如果开启了深度测试，GPU会把该片元的深度值和已经存在于**深度缓冲区**中的深度值进行比较。通常这个值是小于等于的关系，因为我们总想显示出离相机最近的物体(不包括透明/半透明)，而那些被遮挡的就不需要出现在屏幕。如果一个片元通过了测试，那么开发者可以指定是否要用这个片元的深度值覆盖所有的深度值
    - 如果一个片元通过了所有测试，就需要把这些片元的颜色值和颜色缓冲中已有的颜色值进行混合。

2. 阴影的算法

方法

1. 阴影贴图（Shadow Mapping）：通过生成光源视角下的深度贴图，并将其应用于场景的渲染，来确定片元是否位于阴影中。
2. 阴影卷积贴图（Shadow Volume/Stencil Shadow）：使用几何体的阴影体来计算阴影。该方法通过创建阴影体，并使用模板缓冲进行投影来确定片元是否在阴影中。
3. 阴影映射（Shadow Mapping）的改进算法：
    - PCF（Percentage Closer Filtering）：对阴影贴图进行多个采样，然后根据采样结果进行平均，以减少阴影锯齿的出现。
    - VSM（Variance Shadow Mapping）：使用阴影贴图的深度信息的方差来表示阴影的柔和度，从而减少锯齿和阴影失真。
    - ESM（Exponential Shadow Mapping）：使用指数函数来处理阴影深度，以获得更柔和的阴影效果。
4. 光线追踪（Ray Tracing）：利用光线的路径追踪来计算真实的阴影效果。通过追踪光线并检测光线是否与阴影物体相交，可以生成准确的阴影效果。
5. 光线传播（Light Propagation）：使用光线传播技术来模拟光线在场景中传播的方式，从而计算出准确的阴影效果。
6. SSAO（Screen Space Ambient Occlusion）：通过在屏幕空间中模拟环境光遮蔽的效果，来增强场景的阴影效果和真实感。
7. 光子映射（Photon Mapping）：通过发射光子并收集它们在场景中的交互信息，以计算阴影和全局光照效果

步骤

1. 生成阴影贴图：
    - 选择一个光源作为阴影生成的参考，如平行光源或点光源。
    - 从光源的视角渲染场景，并将深度信息（阴影贴图）存储在一个纹理或缓冲区中。这可以通过帧缓冲或纹理附件实现。
    - 渲染时，只考虑光源的可见性，而不关心物体的颜色和纹理。光源视角的渲染通常采用正交投影或透视投影来捕捉场景的深度信息。
2. 应用阴影贴图：
    - 使用常规的渲染管线将场景渲染到屏幕上。
    - 对于每个片元（像素），进行阴影测试来确定其是否处于阴影中。
    - 首先，将片元从摄像机视角转换到光源视角。
    - 然后，使用转换后的坐标与阴影贴图中的深度信息进行比较。
    - 如果片元的深度大于阴影贴图中对应位置的深度值，表示该片元位于阴影中；否则，表示该片元受到光源照射。
3. 阴影衰减（可选）：
    - 可以根据光源的位置和类型，以及阴影贴图的分辨率等因素，对阴影进行衰减。
    - 这可以通过计算片元与光源之间的距离、设置阴影贴图的采样半径等方式来实现。
    - 阴影衰减可以让阴影在远处逐渐减弱，提高场景的真实感和视觉效果。

需要注意的是，阴影渲染的具体实现会因使用的渲染引擎、图形API和算法而有所不同。较复杂的阴影渲染技术，如软阴影、PCF（Percentage Closer Filtering）和CSM（Cascaded Shadow Maps），可能涉及更多的步骤和计算。此外，还有其他高级的阴影算法可用于解决阴影失真、阴影锯齿等问题，以获得更高质量的阴影效果。

3. 容器的底层实现

   set,map系列都是红黑树，有序

   unordered系列都是hash表无序

   红黑树：红黑树（Red-Black Tree）是一种自平衡的二叉搜索树，它在插入和删除节点时通过一系列的旋转和重新着色操作来保持树的平衡性。红黑树的名称来源于节点上的额外属性，即每个节点要么是红色，要么是黑色。

   1. 每个节点要么是红色，要么是黑色。
   2. 根节点是黑色的。
   3. 所有叶节点（空节点）都是黑色的。
   4. 如果一个节点是红色的，则它的两个子节点都是黑色的。
   5. 从任意节点到其每个叶子节点的所有路径都包含相同数量的黑色节点（即具有相同的黑色高度）

   最高的高度是$2log(n+1)$, 增删查的时间复杂度是$O(logn)$

云鲸智能

1. C++11新特性

新的类型	功能	示例
auto	自动类型推断	auto x = 10
基于范围的for循环	基于范围的for循环	for (auto& item : container)
列表初始化	使用一致的语法初始化变量，无论是基本类型、自定义类型还是容器类型	int x{10};std::vector vec{1, 2, 3};
右值引用	新的引用类型&&，用于支持移动语义和完美转发。	
移动语义	所有权从一个对象转移到另一个对象	通过移动构造函数和移动赋值运算符来实现。
Lambda表达式	定义匿名函数的语法	
Null指针常量(nullptr)	表示空指针常量	
并发支持库	多线程编程的支持库，包括线程、互斥量、条件变量等	
静态断言	在编译时对某些条件进行静态断言，条件不满足时报错	

2. 虚函数是怎么实现的，每一个类都有一个虚函数表么？

通过虚函数表vtable实现的，每一个类在编译的时候就会生成一个虚函数表，该表是一个存储函数指针的数组。

对于类创建的对象，编译器会给每个对象中插入一个指向虚函数表的指针，通常叫做vptr即虚函数表指针。用于动态绑定。

VBTable（Virtual Base Table）是针对虚继承的情况而存在的。当一个类以虚继承方式继承自一个或多个基类时，编译器会在派生类对象的内存布局中插入一个 VBTable。VBTable 中存储了用于解决虚继承中的二义性问题的偏移量信息

VBPtr是指向VBTable的指针，用于在虚继承中定位虚基类的子对象。

3. static变量，局部static变量什么时候创建？

   全局static是程序开始时被创建，局部static是函数首次被调用的时候被创建。都具有静态生存期，不会随着函数的退出而销毁。

4. 智能指针有哪些？线程安全是什么？shared_ptr本身是线程安全的么？指向的对象是么？

   智能指针

   1. unique_ptr： unique_ptr是一种独占所有权的智能指针，它管理一个对象，并确保在不再需要时自动释放内存。它禁止多个unique_ptr共享同一个对象，确保只有一个unique_ptr拥有对对象的所有权。当unique_ptr超出作用域或被显式释放时，它会自动删除所管理的对象。
   2. shared_ptr： shared_ptr是一种共享所有权的智能指针，多个shared_ptr可以同时管理同一个对象。它使用引用计数的方式来追踪对象的引用次数，当引用计数为0时，自动释放内存。shared_ptr允许通过拷贝构造函数和赋值运算符在多个shared_ptr之间共享对象。
   3. weak_ptr是一种弱引用的智能指针，它用于解决shared_ptr循环引用导致的内存泄漏问题。weak_ptr可以观测shared_ptr所管理的对象，但并不会增加引用计数。当所有shared_ptr都释放后，weak_ptr不再有效。可以通过lock()方法获取一个有效的shared_ptr来访问对象。

   线程安全是指在多线程环境下，对共享资源的访问和操作能够正确地执行，而不会产生不确定的结果或导致不一致的状态。具体而言，线程安全的代码能够确保多个线程同时访问共享资源时，不会发生数据竞争、死锁、活锁等并发问题，保证数据的一致性和正确性。

   方法：互斥锁，条件变量，原子操作，线程局部存储等同步机制。

   条件变量有两个基本操作：

   - 等待（Wait）：一个线程调用等待操作时，它会释放对互斥锁的占有并进入等待状态，直到其他线程发出信号（Signal）或广播（Broadcast）时才被唤醒。
   - 通知（Signal/Broadcast）：一个线程发送信号或广播时，它会通知等待在条件变量上的一个或多个线程，告诉它们条件已满足。

     线程局部存储：

   使用线程局部存储，可以将全局变量或静态变量转换为每个线程独立的副 本，避免了多线程之间的数据竞争和同步问题。每个线程可以在自己的线 程局部存储中读取和修改数据，而不会干扰其他线程的数据。

   shared_ptr是线程安全的么？

   不是。它的引用计数是通过原子操作来管理的，确保在多线程环境下对引 用计数的增减操作是原子的，避免竞争条件。然而，`std::shared_ptr`的 拷贝构造函数和赋值运算符并不是原子操作，它们会涉及到引用计数的增 减操作。因此，在多线程环境下，对同一个`std::shared_ptr`对象进行并 发的拷贝构造或赋值操作可能会导致竞争条件，从而引发线程安全问题。

   对于读操作时没有问题的，但是对于写操作，需要进行同步

shared_ptr指向的对象是线程安全的么？

指向的对象是否线程安全取决于对象本身的设计和实现。

1. STL的相关容器？vector的结构，vector的数据是在哪里的？list的结构？deque的结构，deque中控器存放的是什么？

   deque是中控器+缓冲区，中控器存储指向缓冲区的指针，缓冲区存储的是数据。

2. 原子变量？

   和PV（信号量）操作不同，粒度更细。原子变量是一种并发编程中用于实现线程安全操作的特殊类型的变量。它们确保在多个线程同时访问时能够保持数据的一致性，并避免竞态条件和数据访问冲突

   1. 原子性：原子变量的操作是原子的，即在执行期间不会被其他线程中断。它们能够以不可分割的方式执行，要么完全成功，要么完全失败，没有中间状态。
   2. 线程安全：原子变量提供了一种线程安全的机制，多个线程可以同时对其进行读写操作，而不会产生数据竞争或冲突。
   3. 无锁操作：原子变量通常使用硬件指令或操作系统提供的原子操作来实现。这些操作不需要显式的互斥锁机制，从而减少了线程间的同步开销和竞争。
   4. 支持多种操作：原子变量提供了一系列原子操作，例如读取、写入、交换、比较交换等。这些操作可以在不同线程之间进行，确保数据的正确性和一致性。

3. Hashmap实现原理和扩容机制？

   是一个桶元素组成的数组，同一个桶元素之间通过链表进行连接。就是数组+链表。扩容是为了让负载因子能够保持在一定范围（0.7-0.8）之内，这样就能尽可能避免hash冲突。处理冲突的两种方式：链表法和开放地址法（线性探测，二次探测），创建新的桶数组，迁移元素，更新至臻和释放桶数组。桶元素可以是链表，红黑树等数据结构。

4. set和multi_set？

   `std::set`和`std::unordered_set`是C++标准库中提供的两种常用集合容器，用于存储一组唯一的元素。它们之间的主要区别在于底层实现和性能特点。

   set是基于红黑树实现的，可以快速查找插入和删除，内部有序，搜索效率O(log(N))，需要额外的指针和存储空间来维护树结构

   multiset是基于哈希表实现的无序集合，搜索插入删除都为O(1),最差是O(N)，存储要求低，桶元素即可。

   set有序，插入和删除效率高

   无序，搜索的性能要求高，选multiset

   2面

5. ROS的service和topic？各自的实现场景？

   服务和话题。

   - 话题topic：是发布订阅机制，允许节点之间异步传递消息，多对多，不需要知道彼此的存在
   - 服务service：请求响应机制，节点之间同步传递消息，一对一

6. C++11新特性中的lambda表达式？

   是一个匿名函数，闭包形式能够实现对参数的值捕获，引用捕获和隐式捕获。隐式捕获可以捕获当前的所有变量。

```
[capture-list](parameters) -> return-type {
    // 函数体
}
[捕获列表](参数) 返回值 {函数体}
```

7. unique_ptr中怎么将其复制出来?

   是独占性质的，所以不能使用普通的复制操作。两种方法

   - 使用移动语义

     ```
     std::unique_ptr<int> sourcePtr(new int(42));
     std::unique_ptr<int> destPtr;

     // 使用 std::move 将所有权从 sourcePtr 移动到 destPtr
     destPtr = std::move(sourcePtr);
     ```

   - 使用显式创建，下面两种方法都能够创建

     ```
     std::unique_ptr<int> sourcePtr(new int(42));
     std::unique_ptr<int> destPtr(new int(*sourcePtr));  // 通过显式创建新的
     unique_ptr
     // 或者使用 reset() 来重置 destPtr
     destPtr.reset(new int(*sourcePtr));
     ```

8. vector的优势？reserve一个长度?

   - 动态大小
   - 随机访问
   - 内存连续
   - 高效的插入和删除
   - 尾部插入弹出方便
   - 支持动态内存管理

9. 项目的这套能不能使用ROS来做?

吉比特

一面：

1. 倒转乾坤怎么实现的?
2. 一个场景中有多个物体模型，怎么优化显示?
3. 多态的概念？虚函数和纯虚函数之间的区别和联系?

   纯虚函数=0，不实现，包含纯虚函数的类叫做抽象基类。不能实例化对象，只提供了接口。

4. 线程间通信，协程是什么?

   管道，套接字，消息队列。

   共享内存，条件变量和互斥量。

   协程（Coroutine）是一种特殊的函数或程序组件，它可以在执行过程中暂停并在需要时继续执行，从而实现多个执行流程之间的切换。协程提供了一种非抢占式的并发模型，允许程序员通过显式的控制流来编写异步、可序列化的代码。

   协程的应用场景包括但不限于：

1. 异步编程：协程可以用于编写异步任务，使得代码更加简洁、易读。通过协程，可以避免回调地狱和线程切换的开销，提升代码的可维护性和性能。
2. 事件驱动编程：协程可以响应事件并在事件到来时执行相应的操作，而不需要阻塞整个程序。这种模型可以用于处理高并发的网络编程、GUI编程等场景。
3. 状态机：协程可以用于实现复杂的状态机，使得状态之间的转换和操作更加直观和简单。通过协程，可以将状态的切换和操作封装为独立的函数，提高代码的可读性和可维护性。

5. 快速幂（笔试题）？堆的实现，堆排序?

二面：

1. 智能指针

   (1) shared_ptr

   实现原理：采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

   智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对象共享同一指针每次创建类的新对象时，初始化指针并将引用计数置为1,当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数
   对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）
   (2) unique_ptr

   unique_ptr采用的是独享所有权语义，一个非空的unique_ptr总是拥有它所指向的资源。转移一个unique_ptr将会把所有权全部从源指针转移给目标指针，源指针被置空；所以unique_ptr不支持普通的拷贝和赋值操作，不能用在STL标准容器中；局部变量的返回值除外（因为编译器知道要返回的对象将要被销毁）；如果你拷贝一个unique_ptr，那么拷贝结束后，这两个unique_ptr都会指向相同的资源，造成在结束时对同一内存指针多次释放而导致程序崩溃。

   (3) weak_ptr

   weak_ptr：弱引用。 引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存都无法释放。需要使用weak_ptr打破环形引用。weak_ptr是一个弱引用，它是为了配合shared_ptr而引入的一种智能指针，它指向一个由shared_ptr管理的对象而不影响所指对象的生命周期，也就是说，它只引用，不计数。如果一块内存被shared_ptr和weak_ptr同时引用，当所有shared_ptr析构了之后，不管还有没有weak_ptr引用该内存，内存也会被释放。所以weak_ptr不保证它指向的内存一定是有效的，在使用之前使用函数lock()检查weak_ptr是否为空指针。

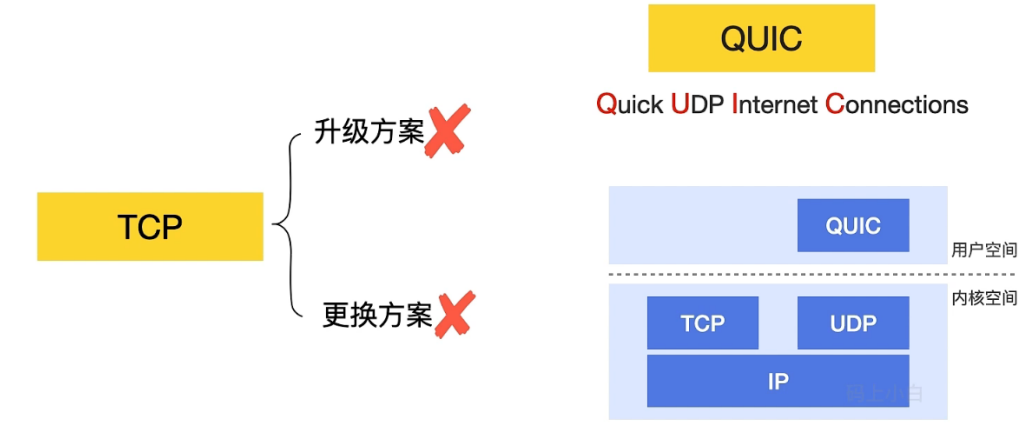
   (4) auto_ptr

   主要是为了解决“有异常抛出时发生内存泄漏”的问题 。因为发生异常而无法正常释放内存。

   auto_ptr有拷贝语义，拷贝后源对象变得无效，这可能引发很严重的问题；而unique_ptr则无拷贝语义，但提供了移动语义，这样的错误不再可能发生，因为很明显必须使用std::move()进行转移。

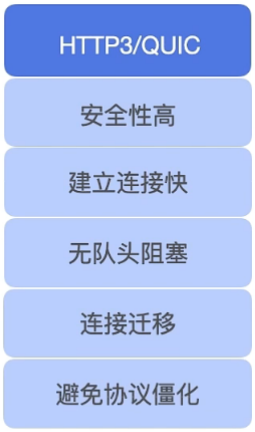
   auto_ptr不支持拷贝和赋值操作，不能用在STL标准容器中。STL容器中的元素经常要支持拷贝、赋值操作，在这过程中auto_ptr会传递所有权，所以不能在STL中使用。

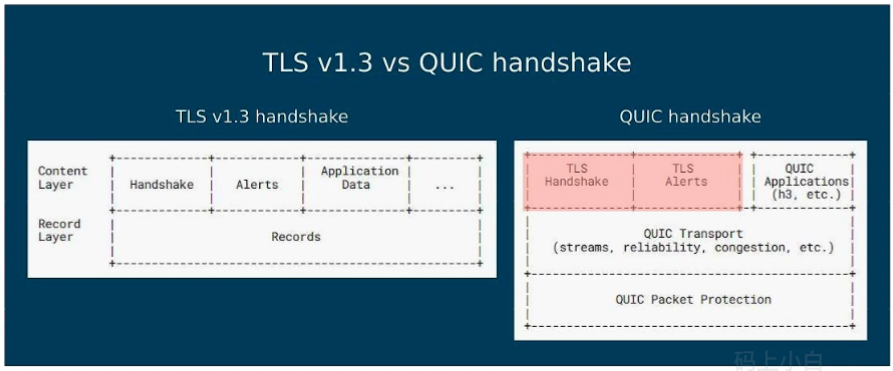
2. LOL是TCP还是UDP传输的，UDP怎么实现可靠传输?

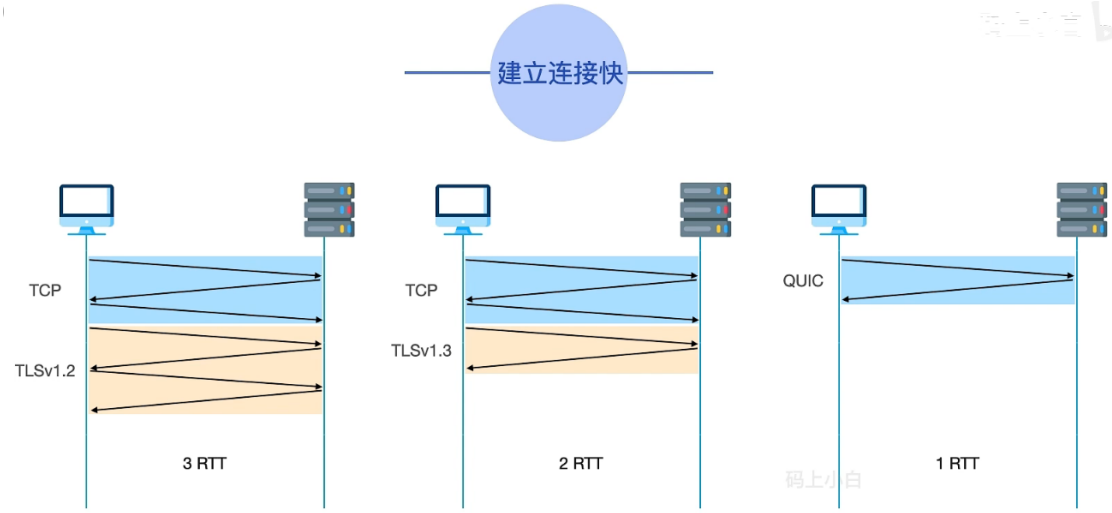


它在用户空间实现了类似TCP的流量控制、拥塞控制和重传机制等功能

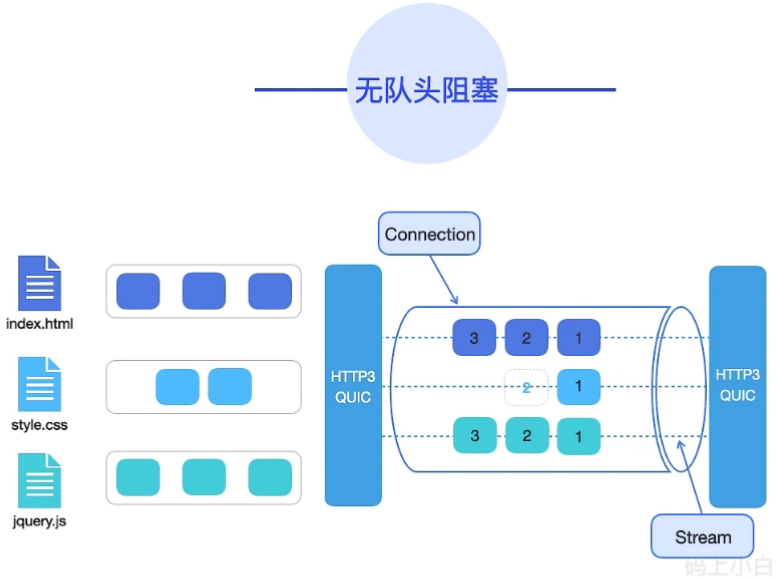
使用UDP避免重新部署，协议速度快。





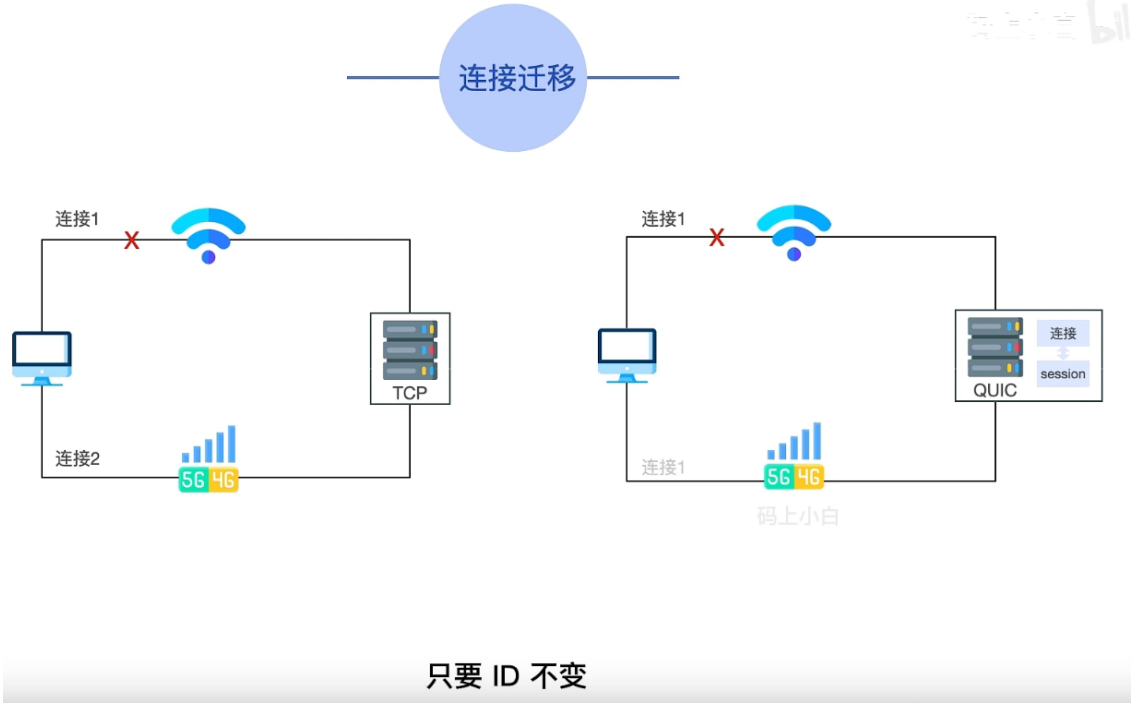


只需要1个RTT就可以完成安全连接的建立

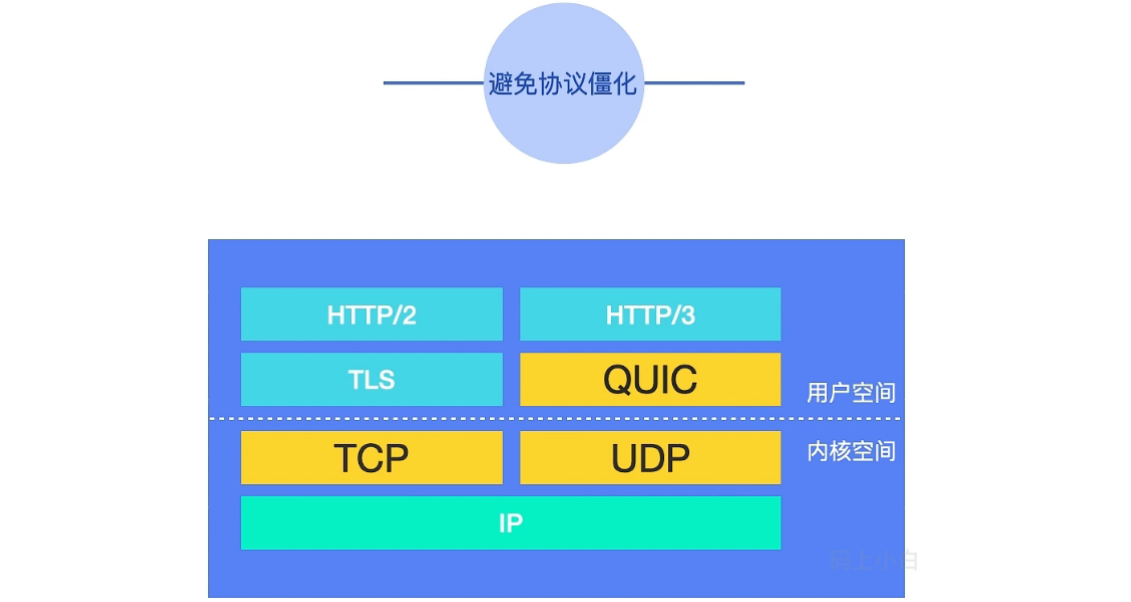


这也就在很大程度上缓解甚至消除了队头阻塞的影响

connection类似于TCP连接，stream是一条http请求。多个stream独立，不会相互依赖。

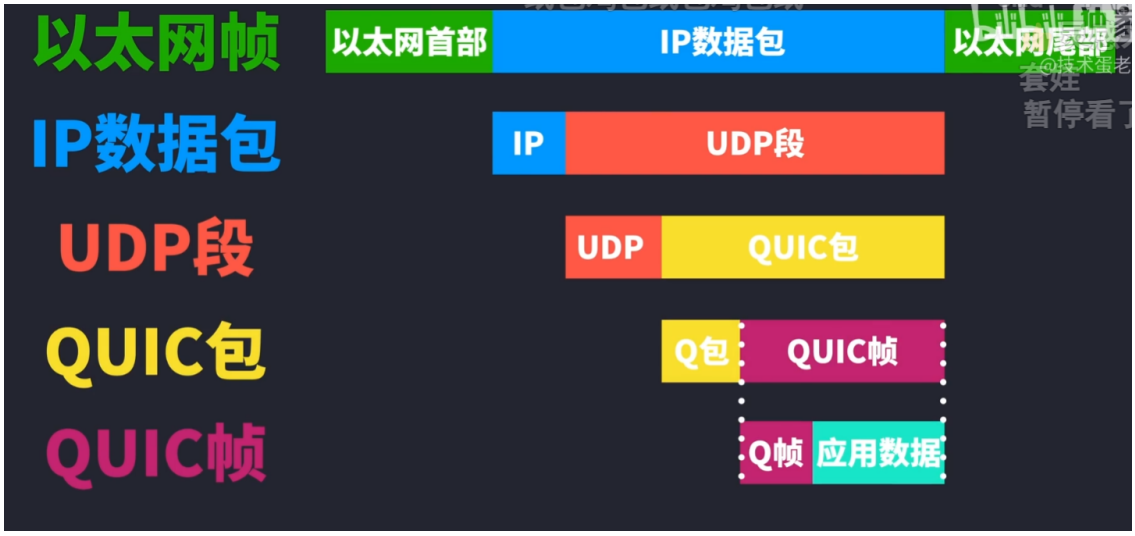


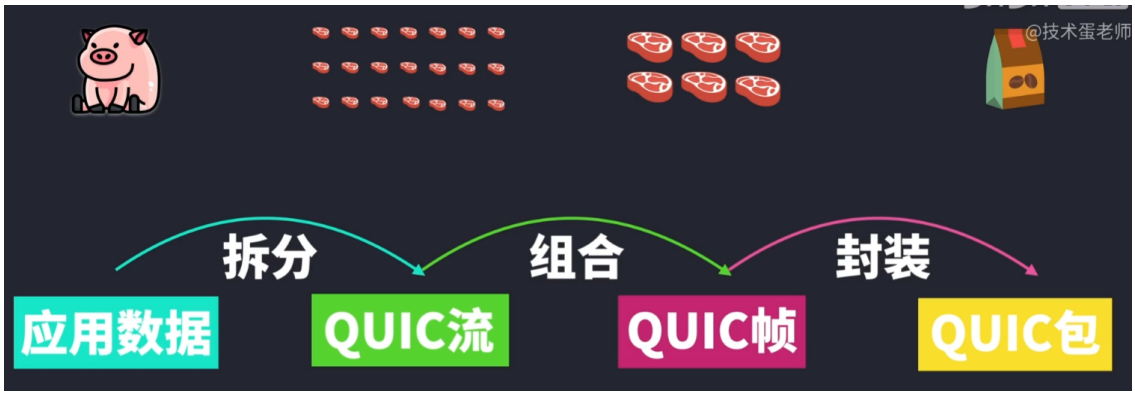
只要 ID 不变



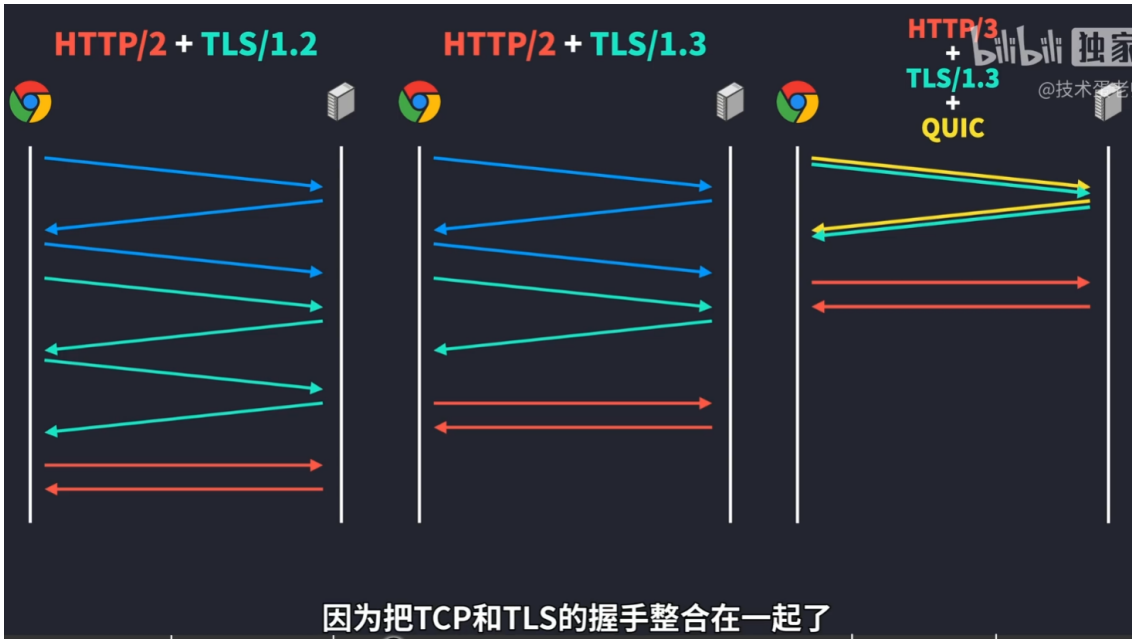
避免协议僵化问题，能够在应用层进行修改。

QUIC催生了HTTP3





丢失了只需要重传就行，不需要等，解决队头阻塞问题。



3. Hash表实现方式，各自适用于什么场景？
4. 红黑树的在STL中的应用？查询时间是多少？为什么用红黑树不用map和hash表？
5.

飞步

1. 虚函数占用空间，普通成员函数不占用空间，只有一份。
2. 右值引用，移动语义和完美转发？

   右值引用（Rvalue Reference）是 C++11 引入的一种引用类型，用于表示对临时对象（右值）的引用。右值引用通过 `&&` 语法声明，例如 `T&&`，其中 `T` 是类型名。

移动语义（Move Semantics）是一种特性，允许将资源（如动态分配的内存）从一个对象转移到另一个对象，而不是进行昂贵的拷贝操作。移动语义通常与右值引用一起使用。使用的时候用`std::move`来完成

完美转发（Perfect Forwarding）是一种技术，允许在函数模板中保留参数的值类别（左值或右值）并将其传递给其他函数，同时保持参数的原始值类别。这样可以实现通用的转发行为，将参数按照原始的值类别进行传递，避免了不必要的拷贝和移动操作。`std::forward`

右值引用和移动语义的主要目标是提高程序的性能和资源利用效率，通过避免不必要的拷贝操作和昂贵的资源管理来提高效率。

完美转发则是一种实现通用函数转发的技术，允许参数在函数调用中保持原始的值类别，以实现灵活和高效的代码。

`noexcept`说明不会引起异常，意味着函数执行的时候不会引发异常

3. 为什么不把编译器的栈设置得大一些?
    - 内存消耗：栈过大会占用更多的内存资源，可能导致栈溢出，
    - 资源分配：增加系统资源开销，
    - 代码可移植性：栈的大小设置得非常大可能会导致代码在不同平台上的不一致性和不可移植性，
    - 栈溢出风险：递归深度过大会隐藏代码中存在的栈溢出错误。
4. 引用折叠?
    - 左值引用折叠

      左值绑定左值，还是一个左值

      ```
      int x = 10;
      int& ref1 = x;  // ref1 是 int& 类型
      int& ref2 = ref1;  // ref2 仍然是 int& 类型
      ```

    - 右值引用折叠

      右值绑定右值，还是一个右值

      ```
      int&& rvref1 = 10;  // rvref1 是 int&& 类型
      int&& rvref2 = std::move(rvref1);  // rvref2 仍然是 int&& 类型
      ```

    - 左值引用和右值引用的折叠

      左值绑定右值，是左值

      ```
      int&& rvref = 10;  // rvref 是 int&& 类型
      int& lref = rvref;  // lref 是 int& 类型
      ```

    - 右值引用与左值引用的折叠

      右值绑定左值，是右值

      ```
      int x = 10;
      int& lref = x;  // lref 是 int& 类型
      int&& rvref = static_cast<int&&>(lref);  // rvref 是 int&& 类型
      ```

      使用引用折叠进行完美转发

```
template<typename T>
void forwardFunction(T&& arg) {
    otherFunction(std::forward<T>(arg));
}
```

原理

- 当 `forwardFunction` 接收到一个左值时，`T` 被推导为左值引用类型，因此 `arg` 也是一个左值引用。当 `forwardFunction` 接收到一个右值时，`T` 被推导为右值引用类型，因此 `arg` 也是一个右值引用。
- 在调用 `otherFunction` 时，我们使用了 `std::forward` 来将 `arg` 按照原始的值类别和引用类型进行转发。如果 `arg` 是一个左值引用，那么 `std::forward<T>(arg)` 会将它转发为左值引用，如果 `arg` 是一个右值引用，那么 `std::forward<T>(arg)` 会将它转发为右值引用。

笔试中的题目

得物

chmod指令：`chmod` 是一个用于更改文件或目录权限的命令，它在类Unix操作系统（比如Linux、macOS等）中常常被使用。它可以用来设置文件或目录的读、写和执行权限，以及确定这些权限是针对文件的所有者、所属组还是其他用户。

$$rwx\text{分别代表读写和执行，权限按照位数转换成数字}111\text{代表全部有那就是}7$$

文件所有者 所属组 其他用户 如果都是那么就是777.`ls -l` 就能查看权限，一共3*3=9位。

两个线程分别给同一个变量自增500，怎么保证执行完程序以后变量值是1000?

```
1.互斥量
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;  // 创建一个互斥量对象
int counter = 0;

// 自增函数
void increment() {
    for (int i = 0; i < 500; ++i) {
        std::lock_guard<std::mutex> lock(mtx);  // 使用 lock_guard 来管理互斥锁
        ++counter;
    }
}

int main() {
    std::thread thread1(increment);
    std::thread thread2(increment);

    thread1.join();
    thread2.join();

    std::cout << "Final counter value: " << counter << std::endl;
```

```
    return 0;
}
```

2.原子变量

```
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> counter(0);

// 自增函数
void increment() {
    for (int i = 0; i < 500; ++i) {
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::thread thread1(increment);
    std::thread thread2(increment);

    thread1.join();
    thread2.join();

    std::cout << "Final counter value: " << counter.load() << std::endl;

    return 0;
}
```

3.条件变量

```
    #include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
int counter = 0;
bool done = false;

// 自增函数
void increment() {
    for (int i = 0; i < 500; ++i) {
        {
            std::unique_lock<std::mutex> lock(mtx);
            ++counter;
        }
        cv.notify_one();  // 通知主线程
    }
}

int main() {
    std::thread thread1(increment);
    std::thread thread2(increment);

    // 等待两个线程完成
    {
```

```
        // 使用unique_lock是为了能够在wait等待和唤醒 lock_gurad要手动实现释放及加锁
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, []{ return counter == 1000; });
        done = true;
    }

    thread1.join();
    thread2.join();

    std::cout << "Final counter value: " << counter << std::endl;

    return 0;
}

4.信号量
#include <iostream>
#include <thread>
#include <semaphore>

std::counting_semaphore<1> sem; // 初始化信号量为1
int counter = 0;

// 自增函数
void increment() {
    for (int i = 0; i < 500; ++i) {
        sem.acquire(); // 信号量减一
        ++counter;
        sem.release(); // 信号量加一
    }
}

int main() {
    std::thread thread1(increment);
    std::thread thread2(increment);

    thread1.join();
    thread2.join();

    std::cout << "Final counter value: " << counter << std::endl;

    return 0;
}
```

网上的

百度

1.`emplace_back`和`push_back`的区别?

前者是原地构造，避免拷贝，基本数据类型+自定义类

后者是构造副本再拷贝。自定义类

2.`vector`是线程安全的么，怎么保证线程安全?

不是的，加锁，读写锁和原子变量。

```
std::atomic<std::vector<int>> atomicVec;

void writeToVector(int value) {
    std::vector<int> localVec = atomicVec.load(); // Read the atomic vector
    localVec.push_back(value); // Modify the local copy
    atomicVec.store(localVec); // Store the modified vector back to the atomic
vector
}
```

3.模板的工作原理

模板定义和模板实例化。

函数模板，类模板和成员函数模板。

模板定义的代码不会生成具体的函数或类，而是为之后的实例化做准备。编译器在这个阶段并不生成实际的代码，而是将模板参数视为占位符。

模板实例化：在编译的时候替换掉，实现实例化。

4.虚函数表在哪里?

编译阶段生成的，用来支持函数的动态绑定。一个类就有一个虚函数表，一般是作为对象的第一个成员。

5.如何实现某一个线程可以独享内存?

在多线程编程中，为了实现某个线程独享内存，可以使用线程局部存储（Thread-Local Storage，TLS）的机制。线程局部存储允许在每个线程中创建一个独立的内存区域，每个线程都可以独立地访问和修改自己的局部存储，而不会影响其他线程的局部存储。

C++11 提供了 `thread_local` 关键字，用于定义线程局部变量。使用 `thread_local` 关键字修饰的变量将会在每个线程中有一份独立的拷贝。

```
#include <iostream>
#include <thread>

thread_local int threadLocalValue = 0;

void incrementThreadLocalValue() {
    threadLocalValue++;
}

int main() {
    std::thread t1(incrementThreadLocalValue);
    std::thread t2(incrementThreadLocalValue);

    t1.join();
    t2.join();

    // 输出结果：1，因为每个线程都有独立的 threadLocalValue
    std::cout << "Thread 1: " << threadLocalValue << std::endl;
```

```
    std::cout << "Thread 2: " << threadLocalValue << std::endl;

    return 0;
}
```

6.说一下原子变量atomic的实现原理?

原子变量（`std::atomic`）是 C++11 引入的一种线程安全的数据类型，用于在多线程环境中对共享数据进行原子操作。原子变量保证了操作的原子性，即对它们的操作要么全部执行成功，要么全部不执行，没有中间状态。`std::atomic` 的实现原理通常使用硬件支持和操作系统提供的原子操作指令来实现。这些原子操作指令可以在硬件层面保证原子性，从而避免了在用户空间使用锁机制进行同步的开销。

硬件原子操作指令+内存模型和屏障+自旋锁+编译器优化。

7. 在多线程编程中，使用互斥锁和条件变量是为了确保线程之间的同步、协调和通信。选择使用哪种机制取决于您需要实现的特定需求和问题。下面我会简要介绍在什么情况下使用互斥锁，什么情况下使用互斥锁加条件变量：

使用互斥锁：

1. **共享资源的互斥访问**：当多个线程需要同时访问和修改共享资源（例如全局变量、数据结构等），您需要使用互斥锁来确保同时只有一个线程可以访问该资源，从而避免竞态条件和数据损坏。
2. **临界区的保护**：当您有一个临界区，即一段代码在任何时间点只能由一个线程执行，使用互斥锁可以防止多个线程同时进入临界区。
3. **避免数据竞争**：当多个线程并发地修改共享数据时，使用互斥锁可以防止数据竞争问题，从而确保正确性。

使用互斥锁 + 条件变量：

1. **等待特定条件的满足**：当一个或多个线程需要等待某个条件满足时，可以使用条件变量。通过在条件不满足时等待条件变量，并在满足条件时通知其他线程，您可以实现线程之间的有效通信和同步。
2. **线程间的通信**：条件变量提供了一种线程间通信的方式，允许一个线程等待另一个线程发出的信号，以便在特定情况下继续执行。
3. **避免忙等待**：使用条件变量可以避免忙等待（busy-waiting）的问题，即线程不断轮询等待某个条件的满足，而是在等待时释放锁，允许其他线程执行。

总之，互斥锁用于保护共享资源的互斥访问，而互斥锁加条件变量用于线程之间的通信、等待特定条件的满足以及避免忙等待。在选择合适的机制时，需要考虑您的问题的特定性质和需求，以及避免死锁、竞态条件等多线程编程常见问题。

topN最大的数

1.排序或者部分排序

2.分治求得最前面N个数

3.内存不够大的情况下，多次读取，因此采用分布式再汇总

4.单个机器的话就要考虑使用堆来存储，小顶堆，小于顶端元素直接舍弃；如果大于顶端元素，就去掉顶端元素，调整堆。

10亿个4字节大概是是4个G

40亿个数字中找是否有和给定数字相同的?

1.set/map进行映射

2.分布式读取，减少IO读取的时间

3.用bitmap，位图法，每个数只用一个位表示即可。

原来是40亿个4字节的数，现在是40亿个1位的bool数，降低了4*8=32倍，这样就能将16G的内存使用降低到500M，2G的内存条也够用。

这里要申请的是32亿个位，要把整数都覆盖了（这里要和40亿区别，40亿是个数，2^32=42亿个是int的取值范围，第32位代表int的最大值，这样就可以对应40亿甚至50亿60亿个int值了）。这样才能表示int类型的全部值。每一位用01表示就行。

4.由于申请的空间可以表示42亿个数，但是其实题目中给的是40亿，那么先对其排序，

如果数据是1 2 3 4 6 7.....这种的，那么可以用(1,4)和(6,2)来表示，这样一来，连续的数都变成了2个数表示。 来了一个新数之后，就用二分法进行查找了。

这样最多有2亿个断点，一个断点4*2=8字节，一共16亿字节，内存是1.6G，可以一次性加载处理。