面试题

面试

阿里巴巴

大文娱

- RDB和AOF区别

面经

猿辅导

付新河

- 使用容量为N的数组实现N/L个容量为L的队列的入队出队操作
- 给边长输出一个顺时针的矩阵：
  给边长4
  1 2 3 4
  5 6 7 8
  9 10 11 12
  13 14 15 16
- 给定字符串，把其中的字母倒序，其他字符不变，例如：$ab&cd > $dc&ba
- https://leetcode.com/problems/house-robber/description/ 就是链接里的题
- 一个无序 可重复的整数数组，找到第一个缺失的正整数arr[1,2,3,3] return 4
- 实现一个LRU算法;查找的时间复杂度O(1);插入的时间复杂度也是O(1);空间不做要求。;Class LRU{ query(int value) insert(int value)}
- 二叉树节点间最大距离
- N个有序数组merge一个有序数组
- 复杂链表复制
- Synchronized，Lock，Kafka，Zookeeper
- https://blog.csdn.net/Fan0628/article/details/99715419?ops_request_misc=&request_id=&biz_id=102&utm_term=猿辅导面经&utm_medium=dis

理论

GoLang

内存管理

内存分配

1. 内存管理基于 tcmalloc
2. 一个 Goroutine 的运行需要 G + P + M 三部分结合起来
3. 如果一个变量被取地址，那么它就有可能被分配到堆上。然而，还要对这些变量做逃逸分析，如果函数 return 之后，变量不再被引用，则将其分配到栈上。
4. mcache: per-P cache，可以认为是 local cache。
5. mcentral: 全局 cache，mcache 不够用的时候向 mcentral 申请。
6. mheap: 当 mcentral 也不够用的时候，通过 mheap 向操作系统申请。

垃圾回收方法

1. 经典GC方法
    i. 引用计数：指针指向加1，删除减1，为0则回收
    ii. 标注-清扫：标记阶段表明所有的存活单元，清扫阶段将垃圾单元回收
    iii. 节点复制：整个堆等分为两个半区，一个包含现有数据，另一个包含已被废弃的数据
    iv. 分代收集：将对象按生命周期长短存放到堆上的两个（或者更多）区域，生命周期长则新生代->老生代
2. GoLang GC
    i. 在堆上分配大于 32K byte 对象的时候进行检测此时是否满足垃圾回收条件
    ii. 当前堆上的活跃对象大于我们初始化时候设置的 GC 触发阈值
    iii. 三色标记法，所有对象最开始都是白色，从 root 开始找到所有可达对象，标记为灰色，放入待处理队列。遍历灰色对象队列，将其引用对象标记为灰色放入待处理队列，自身标记为黑色。处理完灰色对象队列，执行清扫工作。

算法

排序

介绍一下快排

- 根据哨兵元素，用两个指针指向待排序数组的首尾，首指针从前往后移动找到比哨兵元素大的，尾指针从后往前移动找到比哨兵元素小的，交换两个元素
- 直到两个指针相遇，这是一趟排序，经常这趟排序后，比哨兵元素大的在右边，小的在左边。经过多趟排序后，整个数组有序。

链表

如何判断链表有环

判断一个链表是否有环有两种办法：一种最经典是定义两个指针，一个指针每次向前走一步，一个指针每次向前走两步，如果两个指针最终重合。 则证明有环。

实现一个栈，o(1)时间找最大值

删除单链表的倒数第k个节点

双指针

树

二叉树的最近公共祖先

其他

手撕二分查找

矩阵从左上到右下找最小路径

数据库日志文件记录了登录登出操作，怎么求最大在线人数

数据结构

树

红黑树和其他平衡树有什么不同

- AVL树：AVL树是带有平衡条件的二叉查找树，一般是用平衡因子差值判断是否平衡并通过旋转来实现平衡，左右子树树高不超过1，和红黑树相比，AVL树是严格的平衡二叉树，平衡条件必须满足（所有节点的左右子树高度差不超过1）。不管我们是执行插入还是删除操作，只要不满足上面的条件，就要通过旋转来保持平衡，而的英文旋转非常耗时的，由此我们可以知道AVL树适合用于插入与删除次数比较少，但查找多的情况
- 红黑树：一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树（由于是弱平衡，可以看到，在相同的节点情况下，AVL树的高度低于红黑树），相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，我们就用红黑树。
  i. 每个节点非红即黑
  ii. 根节点是黑的;
  iii. 每个叶节点（叶节点即树尾端NULL指针或NULL节点）都是黑的;

iv. 如图所示，如果一个节点是红的，那么它的两儿子都是黑的;
v. 对于任意节点而言，其到叶子点树NULL指针的每条路径都包含相同数目的黑节点;
vi. 每条路径都包含相同的黑节点;

操作系统

进程与线程

进程与线程区别

- 进程是运行中的程序，线程是进程的内部的一个执行序列
- 进程是资源分配的单元，线程是执行行单元
- 进程间切换代价大，线程间切换代价小
- 进程拥有资源多，线程拥有资源少
- 多个线程共享进程的资源

进程的状态，以及转换

1. 运行态(running):占有处理器正在运行
2. 就绪态(ready):具备运行条件，等待系统分配处理器以便运行
3. 等待态(blocked):不具备运行条件，正在等待某个事件的完成
4. 运行态—→等待态：等待使用资源；如等待外设传输；等待人工干预。
5. 等待态—→就绪态：资源得到满足；如外设传输结束；人工干预完成。
6. 运行态—→就绪态：运行时间片到；出现有更高优先权进程。
7. 就绪态—→运行态：CPU 空闲时选择一个就绪进程。

进程有哪些数据

- 程序段：描述进程本身要完成的功能；
- 数据段：程序加工的对象和场所；
- 进程控制块：内存中存储，记录进程生存周期内状态变化的存储区域。不同操作系统有不同的进程控制块格式和信息，但基本包括进程标识符，当前状态，现场保护区，存储指针，占用资源表以及进程优先级等信息。它是进程存在的唯一标志。

进程的通信方式

- 共享存储就是操作系统另外分配一个内存区域用于进程间的通信，这就是共享空间，两个进程都可以访问这个共享空间，但是两个进程对共享空间的访问必须是互斥的，即同一时间段只能有一个进程访问这个空间，类似于线程那样。
- 管道：是指用于连接读写进程的一个共享文件，又名pipe文件。其实就是在内存中开辟一个大小固定的缓冲区。半双工通信,各进程要互斥地访问管道。
- 消息传递又包括两种通信方式：直接通信方式和间接通信方式。直接：每个进程都有一个消息缓存队列。当进程A要给进程B通信，就先调用发送原语，把消息发送到进程B的消息缓存

队列当中，然后进程B会调用接收原语，从自己的消息缓存队列当中取出消息。间接：消息要先发送到中间实体（信箱）中，因此也称信箱通信方式，比如计算机网络中的电子邮箱系统。

- 信号是Linux系统中用于进程间互相通信或者操作的一种机制，信号可以在任何时候发给某一进程，而无需知道该进程的状态。如果该进程当前并未处于执行状态，则该信号就有内核保存起来，知道该进程回复执行并传递给它为止。
  如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消是才被传递给进程。

进程调度算法，CFS，RR

- 先来先服务调度算法：该算法既可用于作业调度，也可用于进程调度
- 短作业(进程)优先调度算法：分别用于作业调度和进程调度
- 时间片轮转法：把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾
- 多级反馈队列调度算法：应设置多个就绪队列，并为各个队列赋予不同的优先级。当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。仅当第一队列空闲时，调度程序才调度第二队列中的进程运行
- 优先权调度算法：系统将从后备队列中选择若干个优先权最高的作业装入内存。

如何调用操作系统方法

- 系统调用：在用户空间操纵用户数据，调用用户空间函数。但在很多情况下，应用程序需要获得系统服务，这时就必须利用系统提供给用户的特殊接口--系统调用。
- 内核：是一组程序模块，作为可信软件来提供支持进程并发执行的基本功能和基本操作，常驻留在内核空间，运行于核心态，具有访问硬件设备和所有主存空间权限，是仅有的能执行特权指令的程序。

如何陷入操作系统内核态

- 当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）。其他的属于用户态。用户程序运行在用户态,操作系统运行在内核态.
- 用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现
- 当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。
- 当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。

内存管理

页面置换算法

- 最佳置换算法：标记最大的页应该被置换。
- 先进先出置换算法：总是选择在主存中停留时间最长（即最老）的一页置换，即先进入内存的页，先退出内存
- 最近最久未使用：当需要置换一页时，选择在之前一段时间里最久没有使用过的页面予以置换
- Clock置换算法：LRU算法的近似实现
- 最少使用（LFU）置换算法：在采用最少使用置换算法时，应为在内存中的每个页面设置一个移位寄存器，用来记录该页面被访问的频率。该置换算法选择在之前时期使用最少的页面作为淘汰页

中断的过程

- 中断请求、中断判优、中断响应、中断处理和中断返回。

MySQL

MySQL架构与历史

InnoDB和MyISAM对比

- InnoDB 支持事务，MyISAM 不支持事务。这是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一
- InnoDB 支持外键，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败
- InnoDB 是聚集索引，MyISAM 是非聚集索引
- InnoDB 不保存表的具体行数，而MyISAM 用一个变量保存了整个表的行数
- InnoDB 最小的锁粒度是行锁，MyISAM 最小的锁粒度是表锁
- MyISAM支持全文类型索引，而InnoDB不支持全文索引

事务的四大特性

1. 原子性：事务是数据库的逻辑工作单位，不可分割，事务中包含的各操作要么都做，要么都不做
2. 一致性：事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。
3. 隔离性：一个事务的执行不能其它事务干扰
4. 持续性：也称永久性，指一个事务一旦提交，它对数据库中的数据的改变就应该是永久性的，不能回滚

事务的隔离级别

1. 读未提交：该隔离级别的事务会读到其它未提交事务的数据，此现象也称之为 脏读
2. 读提交：一个事务可以读取另一个已提交的事务，多次读取会造成不一样的结果，此现象称

为不可重复读问题
3. 可重复读：该隔离级别是 MySQL 默认的隔离级别，在同一个事务里， select 的结果是事务开始时时间点的状态，因此，同样的 select 操作读到的结果会是一致的，但是，会有 幻读现象
4. 幻读：在该隔离级别下事务都是串行顺序执行的，MySQL 数据库的 InnoDB 引擎会给读操作隐式加一把读共享锁，从而避免了脏读、不可重读复读和幻读问题。

有哪些索引，为什么用B+树，为什么快， B+树结构

- B+索引
  - 每个中间节点不保存数据,只用来索引,也就意味着所有非叶子节点的值都被保存了一份在叶子节点中.
  - 叶子节点之间根据自身的顺序进行了链接.
- 聚簇索引
  - 聚簇索引不是一种索引类型,而是一种存储数据的方式.Innodb的聚簇索引是在同一个数据结构中保存了索引和数据.
- 覆盖索引
  - 当一个索引包含(或者说是覆盖)需要查询的所有字段的值时,我们称之为覆盖索引.

b树b+树区别

- M阶的b树
  - 定义任意非叶子结点最多只有M个儿子，且M>2
  - 根结点的儿子数为[2, M]
  - 除根结点以外的非叶子结点的儿子数为[M/2, M]，向上取整
  - 非叶子结点的关键字个数=儿子数-1；
  - 所有叶子结点位于同一层
  - k个关键字把节点拆成k+1段，分别指向k+1个儿子，同时满足查找树的大小关系。
- B树特性
  - 关键字集合分布在整颗树中
  - 任何一个关键字出现且只出现在一个结点中
  - 搜索有可能在非叶子结点结束
  - 其搜索性能等价于在关键字全集内做一次二分查找
- m阶的b+树的特征
  - 有n棵子树的非叶子结点中含有n个关键字（b树是n-1个），这些关键字不保存数据，只用来索引，所有数据都保存在叶子节点（b树是每个关键字都保存数据）。
  - 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
  - 所有的非叶子结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字。
  - 通常在b+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。
  - 同一个数字会在不同节点中重复出现，根节点的最大元素就是b+树的最大元素。
- b+树相比于b树的查询优势

    - b+树的中间节点不保存数据，所以磁盘页能容纳更多节点元素，更“矮胖”；
    - b+树查询必须查找到叶子节点，b树只要匹配到即可不用管元素位置，因此b+树查找更稳定（并不慢）
    - 对于范围查找来说，b+树只需遍历叶子节点链表即可，b树却需要重复地中序遍历

mvcc

- 每个连接到数据库的读者，在某个瞬间看到的是数据库的一个快照，写者写操作造成的变化在写操作完成之前（或者数据库事务提交之前）对于其他的读者来说是不可见的
- 读写并存的时候，写操作会根据目前数据库的状态，创建一个新版本，并发的读则依旧访问旧版本的数据。
- MVCC(Multiversion concurrency control) 就是 同一份数据临时保留多版本的一种方式，进而实现并发控制
- 查找数据行版本号早于当前事务版本号的数据行记录
- 查找删除版本号要么为NULL，要么大于当前事务版本号的记录

锁

数据库的锁。什么时候用表锁，什么时候用行锁

- 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高 ，并发度最低。
- 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。
- 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表级锁。
- 间隙锁：当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-Key锁）
- 乐观锁：用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。
- 悲观锁：悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作
- 共享锁 （也称为 S 锁，读锁）： 允许事务读取一行数据
- 独占锁 （也称为 X 锁，写锁）： 允许事务删除或更新一行数据。
- S 锁和 S 锁是 兼容 的，X 锁和其它锁都 不兼容
- 意向共享锁 （IS）： 事务即将给表中的各个行设置共享锁，事务给数据行加 S 锁前必须获得该表的 IS 锁。
- 意向排他锁 （IX）： 事务即将给表中的各个行设置排他锁，事务给数据行加 X 锁前必须获得该表 IX 锁。

Redis

数据结构与对象

有哪些基础数据结构

1. string：动态字符串，是可以修改的字符串
2. list：表的存储结构用的是双向链表而不是数组
3. hash：第一维是数组，第二维是链表
4. set：它的内部也使用hash结构，所有的value都指向同一个内部值
5. zset：底层实现使用了两个数据结构，第一个是hash，第二个是跳跃列表

项目哪里用到了redis，说一下常用数据类型的使用场景

- 高性能适合当做缓存
- 丰富的数据格式性能更高，应用场景丰富；set——可以简单的理解为ID-List的模式，如微博中一个人有哪些好友，set最牛的地方在于，可以对两个set提供交集、并集、差集操作
- 单线程可以作为分布式锁
- 自动过期能有效提升开发效率
- 分布式和持久化有效应对海量数据和高并发
- 秒杀和Redis的结合
    - 提前预热数据，放入Redis
    - 商品列表放入Redis List
    - 商品的详情数据 Redis hash保存，设置过期时间
    - 商品的库存数据Redis sorted set保存
    - 用户的地址信息Redis set保存
    - 订单产生扣库存通过Redis制造分布式锁，库存同步扣除
    - 订单产生后发货的数据，产生Redis list，通过消息队列处理
    - 秒杀结束后，再把Redis数据和数据库进行同步

分布式锁怎么实现的

- set key value nx px time
- nx: SET IF NOT EXIST

并发

缓存穿透，缓存雪崩及相应的解决

- 缓存雪崩：是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。解决方案：缓存失效时间分散开
- 缓存穿透：对于系统A，假设一秒 5000 个请求，结果其中 4000 个请求是黑客发出的恶意攻击。 黑客发出的那 4000 个攻击，缓存中查不到，每次你去数据库里查，也查不到。解决方

案：每次系统 A 从数据库中只要没查到，就写一个空值到缓存里去
- 缓存击穿：就是说某个 key 非常热点，访问非常频繁，处于集中式高并发访问的情况，当这个 key 在失效的瞬间，大量的请求就击穿了缓存，直接请求数据库，就像是在一道屏障上凿开了一个洞。解决方案：永远不过期；或者基于 redis or zookeeper 实现互斥锁，等待第一个请求构建完缓存之后，再释放锁，进而其它请求才能通过该 key 访问数据

rsdis为什么快

- redis是基于内存的，内存的读写速度非常快
- redis是单线程的，省去了很多上下文切换线程的时间
- redis使用多路复用技术，可以处理并发的连接

为什么单线程快

- 因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽
- 不需要各种锁的性能消耗
- 单线程多进程集群方案
- 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU。

IO多路复用技术

- 多路-指的是多个socket连接，复用-指的是复用一个线程。多路复用主要有三种技术：select，poll，epoll。epoll是最新的也是目前最好的多路复用技术。
- 使用了单线程来轮询描述符，将数据库的开、关、读、写都转换成了事件，减少了线程切换时上下文的切换和竞争。

单机数据库的实现

rdb和aof的区别

- rdb：RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。
- aof：AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

Redis和MySQL保持数据一致

- 采用延时双删策略：写库前后都进行redis.del(key)操作，并且设定合理的超时时间。先删除缓存；再写数据库；休眠500毫秒；再次删除缓存
- 异步更新缓存：MySQL binlog增量订阅消费+消息队列+增量数据更新到redis；读Redis：热

数据基本都在Redis，写MySQL:增删改都是操作MySQL，更新Redis数据：MySQ的数据操作binlog，来更新到Redis

计算机网络

协议

计算机网络TCP/IP协议

- TCP 是面向连接的、可靠的流协议。流就是指不间断的数据结构，当应用程序采用 TCP 发送消息时，虽然可以保证发送的顺序，但还是犹如没有任何间隔的数据流发送给接收端。

tcp udp区别 滑动窗口

- UDP 是不具有可靠性的数据报协议。细微的处理它会交给上层的应用去完成。
- TCP 用于在传输层有必要实现可靠传输的情况；而在一方面，UDP 主要用于那些对高速传输和实时性有较高要求的通信或广播通信。TCP 和 UDP 应该根据应用的目的按需使用。

tcp三次握手

- 建立一个 TCP 连接时需要客户端和服务器端总共发送三个包以确认连接的建立。在socket编程中，这一过程由客户端执行connect来触发。
- 第一次握手：客户端将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给服务器端，客户端进入SYN_SENT状态，等待服务器端确认。
- 第二次握手：服务器端收到数据包后由标志位SYN=1知道客户端请求建立连接，服务器端将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给客户端以确认连接请求，服务器端进入SYN_RCVD状态。
- 第三次握手：客户端收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给服务器端，服务器端检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，客户端和服务器端进入ESTABLISHED状态，完成三次握手，随后客户端与服务器端之间可以开始传输数据了。

tcp四次挥手

- 四次挥手即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close来触发。
- 第一次挥手：客户端发送一个FIN=M，用来关闭客户端到服务器端的数据传送，客户端进入FIN_WAIT_1状态。意思是说"我客户端没有数据要发给你了"，但是如果你服务器端还有数据没有发送完成，则不必急着关闭连接，可以继续发送数据。
- 第二次挥手：服务器端收到FIN后，先发送ack=M+1，告诉客户端，你的请求我收到了，但是我还没准备好，请继续你等我的消息。这个时候客户端就进入FIN_WAIT_2 状态，继续等待服务器端的FIN报文。

- 第三次挥手：当服务器端确定数据已发送完成，则向客户端发送FIN=N报文，告诉客户端，好了，我这边数据发完了，准备好关闭连接了。服务器端进入LAST_ACK状态。
- 第四次挥手：客户端收到FIN=N报文后，就知道可以关闭连接了，但是他还是不相信网络，怕服务器端不知道要关闭，所以发送ack=N+1后进入TIME_WAIT状态，如果Server端没有收到ACK则可以重传。服务器端收到ACK后，就知道可以断开连接了。客户端等待了2MSL后依然没有收到回复，则证明服务器端已正常关闭，那好，我客户端也可以关闭连接了。最终完成了四次握手。

tcp拥塞控制，流量控制

- 由滑动窗口协议（连续ARQ协议）实现。滑动窗口协议既保证了分组无差错、有序接收，也实现了流量控制。主要的方式就是接收方返回的 ACK 中会包含自己的接收窗口的大小，并且利用大小来控制发送方的数据发送。
- 拥塞控制：拥塞控制是作用于网络的，它是防止过多的数据注入到网络中，避免出现网络负载过大的情况；常用的方法就是：（1）慢开始、拥塞避免（2）快重传、快恢复
- 流量控制：流量控制是作用于接收者的，它是控制发送者的发送速度从而使接收者来得及接收，防止分组丢失的。

Web

浏览器

客户输入一个https的url，发生了什么

- URL输入
- DNS解析
- TCP连接
- 发送HTTP请求
- 服务器处理请求
- 服务器响应请求
- 浏览器解析渲染页面
- 连接结束

session和cookie区别，如何实现登录

- 由于HTTP协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是Session
- 思考一下服务端如何识别特定的客户？这个时候Cookie就登场了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。
- Session是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中；
- Cookie是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现Session的一种方式。

HTTP和HTTPS的区别

- https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
- http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
- http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443

HTTPS怎么加密的

- HTTPS = HTTP + SSL
- 客户端发起HTTPS请求:这个没什么好说的，就是用户在浏览器里输入一个HTTPS网址，然后连接到服务端的443端口。
- 服务端的配置:采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面。这套证书其实就是一对公钥和私钥。如果对公钥不太理解，可以想象成一把钥匙和一个锁头，只是世界上只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。
- 传送证书: 这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。
- 客户端解析证书: 这部分工作是由客户端的SSL/TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警示框，提示证书存在的问题。如果证书没有问题，那么就生成一个*随机值*。然后用证书（也就是公钥）对这个随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。
- 传送加密信息: 这部分传送的是用证书加密后的随机值，目的是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。
- 服务端解密信息: 服务端用私钥解密后，得到了客户端传过来的随机值，然后把内容通过该随机值进行对称加密，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。
- 传输加密后的信息: 这部分信息就是服务端用私钥加密后的信息，可以在客户端用随机值解密还原。
- 客户端解密信息: 客户端用之前生产的私钥解密服务端传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

架构

秒杀

对秒杀活动如何控制数量的想法。

服务的熔断与降级

分布式

事务

两阶段提交

1. 投票
2. 事务提交

Kubernetes

概念

operator

网络

网络问题

- 每一个 Pod 都有它自己的IP地址，这就意味着你不需要显式地在每个 Pod 之间创建链接，你几乎不需要处理容器端口到主机端口之间的映射。这将创建一个干净的、向后兼容的模型，在这个模型里，从端口分配、命名、服务发现、负载均衡、应用配置和迁移的角度来看，Pod 可以被视作虚拟机或者物理主机。

其他

乐观锁悲观锁

悲观锁的应用场景