go 面试资料整理

go 语言基础

熟悉语法，撸个百十道基础面试题就差不多了。

go 语言进阶

go 中有哪些锁？

sync.Mutex 互斥锁
sync.RWMutex 读写锁

CSP 并发模型？

CSP 并发模型它并不关注发送消息的实体，而关注的是发送消息时使用的 channel，go 语言借用了 process 和 channel 这两个概念，process 表现为 go 里面的 goroutine，是实际并发执行的实体，每个实体之间是通过 channel 来进行匿名传递消息使之解藕，从而达到通讯来实现数据共享。

不要通过共享内存来通信，而要通过通信来实现内存共享。

1、sync.mutex 互斥锁（获取锁和解锁可以不在同一个协程，当获取到锁之后，未解锁，此时再次获取锁将会阻塞）
2、通过 channel 通信
3、sync.WaitGroup

GPM 模型指的是什么？goroutine 的调度时机有哪些？如果 syscall 阻塞会发生什么？

在 go 中是通过 channel 通信来共享内存的。

G：指的是 Goroutine，也就是协程，go 中的协程做了优化处理，内存占用仅几 kb 且调度灵活，切换成本低。
P：指的是 processor,也就是处理器，感觉也可理解为协程调度器。
M：指的是 thread，内核线程。

调度器的设计策略：
1、线程复用：当本线程无可运行的 G 时，M-P-G0 会处于自旋状态，尝试从全局队列获取 G，再从其他线程绑定的 P 队列中偷取 G，而不是销毁线程；当本线程因为 G 进行系统调用阻塞时，线程会释放绑定的 P 队列，如果有空闲的线程可用就复用空闲的线程，不然就创建一个新的线程来接管释放出来的 P 队列。
2、利用并行：GOMAXPROCS 设置 P 的数量，最多有这么多个线程分布在多个 cpu 上同时运行。
3、抢占：在 coroutine 中要等待一个协程主动让出 CPU 才执行下一个协程，在 Go

中，一个 goroutine 最多占用 CPU 10ms，防止其他 goroutine 被饿死。

go func 的流程：
1、创建一个 G，新建的 G 优先保存在 P 的本地队列中，如果满了则会保存到全局队列中。
2、G 只能运行在 M 中，一个 M 必须持有一个 P，M 与 P 时 1:1 关系，M 会从 P 的本地队列
弹出一个可执行状态的 G 来执行。
3、一个 M 调度 G 执行的过程是一个循环机制。
4、如果 G 阻塞，则 M 也会被阻塞，runtime 会把这个线程 M 从 P 摘除，再创建或者复用其他线程来接管 P 队列。
5、当 G、M 不在被阻塞，即系统调用结束，会先尝试找会之前的 P 队列，如果之前的 P 队列已经被其他线程接管，那么这个 G 会尝试获取一个空闲的 P 队列执行，并放入到这个 P 的本地队列。否则这个线程 M 会变成休眠状态，加入空闲线程队列，而 G 则会被放入全局队列中。

M0：
M0 是启动程序后的编号为 0 的主线程，这个 M 对应的实例会在全局变量 runtime.m0 中，不需要在 heap 上分配，M0 负责执行初始化操作和启动第一个 G，之后 M0 与其他的 M 一样。
G0：
G0 是每次启动一个 M 都会第一个创建的 goroutine，G0 仅负责调度，不指向任何可执行函数，每个 M 都会有一个自己的 G0，在调度或者系统调用时会使用 G0 的栈空间，全局变量
的 G0 是 M0 的。

N:1-----出现阻塞的瓶颈，无法利用多个 cpu
1:1-----跟多线程/多进程模型无异，切换协程代价昂贵
M:N-----能够利用多核，过于依赖协程调度器的优化和算法

同步协作式调度
异步抢占式调度

channel 底层的数据结构是什么？发送和接收元素的本质是什么？

```
type hchan struct {
    qcount   uint           // *chan 里元素数量
    dataqsiz uint           // *底层循环数组的长度，就是 chan 的容量
    buf      unsafe.Pointer // *指向大小为 dataqsiz 的数组，有缓冲的 channe
l
    elemsize uint16         // chan 中的元素大小
    closed   uint32         // chan 是否被关闭的标志
    elemtype *_type         // chan 中元素类型
    recvx    uint           // *当前可以接收的元素在底层数组索引(<-chan)
    sendx    uint           // *当前可以发送的元素在底层数组索引(chan<-)
    recvq    waitq          // 等待接收的协程队列(<-chan)
```

```
    sendq    waitq          // 等待发送的协程队列(chan<-)
    lock     mutex          // 互斥锁,保证每个读 chan 或者写 chan 的操作都是
原子的
}

// waitq 是 sudog 的一个双向链表，sudog 实际上是对 goroutine 的一个封装。
type waitq struct {
    first *sudog
    last  *sudog
}

// channel 的发送和接收操作本质上都是"值的拷贝"(只是拷贝它的值而已),
```

channel 使用应该注意哪些情况，在哪些情况下会死锁/阻塞？

1、一个无缓冲 channel 在一个主 go 程里同时进行读和写；
2、无缓冲 channel 在 go 程开启之前使用通道；
3、通道 1 中调用了通道 2，通道 2 中调用了通道 1；
4、读取空的 channel；
5、超过 channel 缓存继续写入数据；
6、向已经关闭的 channel 中写入数据不会导致死锁，但会 Panic 异常。
7、close 一个已经关闭的 channel 会 Panic 异常。

那些类型不能作 map 的为 key？map 的 key 为什么是无序的？

map 的 key 必须可以比较，func、map、slice 这三种类型不可比较，
只有在都是 nil 的情况下，才可与 nil (== or !=)。因此这三种类型
不能作为 map 的 key。

数组或者结构体能够作为 key？？？？有些能，有些不能，要看字段或者元素是否可比较

1、map 在扩容后，会发生 key 的搬迁，原来落在同一个 bucket 中的 key 可能分散，ke
y 的位置发生了变化。
2、go 中遍历 map 时，并不是固定从 0 号 bucket 开始遍历，每次都是从一个随机值序号
的 bucket 开始遍历，
并且是从这个 bucket 的一个随机序号的 cell 开始遍历。
3、哈希查找表用一个哈希函数将 key 分配到不同的 bucket(数组的下标 index)。不同
的哈希函数实现也
会导致 map 无序。

"迭代 map 的结果是无序的"这个特性是从 go1.0 开始加入的。

如何解决哈希查找表存在的“碰撞”问题（hash 冲突）？

hash 碰撞指的是：两个不同的原始值被哈希之后的结果相同，也就是不同的 key 被哈希
分配到了同一个 bucket。

链表法：将一个 bucket 实现成一个链表，落在同一个 bucket 中的 key 都会插入这个链表。

开放地址法：碰撞发生后，从冲突的下标处开始往后探测，到达数组末尾时，从数组开始处探测，直到找到一个
空位置存储这个 key，当找不到位置的情况下会触发扩容。

map 是线程安全的么？

map 不是线程安全的，sync.map 是线程安全的。

在查找、赋值、遍历、删除的过程中都会检测写标志，一旦发现写标志"置位"等于 1，则直接 panic，
因为这表示有其他协程同时在进行写操作。赋值和删除函数在检测完写标志是"复位"之后，先将
写标志位"置位"，才会进行之后的操作。

思考：为什么 sync.map 为啥是线程安全？？

map 的底层实现原理是什么？

```
type hmap struct {
    count      int   // len(map)元素个数
    flags      uint8 //写标志位
    B          uint8 // buckets 数组的长度的对数，buckets 数组的长度是 2^B
    noverflow  uint16
    hash0      uint32
    buckets    unsafe.Pointer // 指向 buckets 数组
    oldbuckets unsafe.Pointer // 扩容的时候，buckets 长度会是 oldbuckets
的两倍
    nevacuate  uintptr
    extra      *mapextra
}

// 编译期间动态创建的 bmap
type bmap struct {
    topbits  [8]uint8
    keys     [8]keytype
    values   [8]valuetype
    pad      uintptr
    overflow uintptr
}
```

在 go 中 map 是数组存储的，采用的是哈希查找表，通过哈希函数将 key 分配到不同的 bucket，
每个数组下标处存储的是一个 bucket，每个 bucket 中可以存储 8 个 kv 键值对，当每个 bucket

存储的 kv 对到达 8 个之后，会通过 overflow 指针指向一个新的 bucket，从而形成一个链表。

map 的扩容过程是怎样的？

相同容量扩容
2 倍容量扩容

扩容时机：
1、当装载因子超过 6.5 时，表明很多桶都快满了，查找和插入效率都变低了，触发扩容。

扩容策略：元素太多，bucket 数量少，则将 B 加 1，buctet 最大数量(2^B)直接变为原来 bucket 数量的 2 倍，再渐进式的把 key/value 迁移到新的内存地址。

2、无法触发条件 1，overflow bucket 数量太多，查找、插入效率低，触发扩容。
(可以理解为：一座空城，房子很多，但是住户很少，都分散了，找起人来很困难)

扩容策略：开辟一个新的 bucket 空间，将老 bucket 中的元素移动到新 bucket，使得同一个 bucket 中的 key 排列更紧密，节省空间，提高 bucket 利用率。

map 的 key 的定位过程是怎样的？

对 key 计算 hash 值，计算它落到那个桶时，只会用到最后 B 个 bit 位，再用哈希值的高 8 位
找到 key 在 bucket 中的位置。桶内没有 key 会找第一个空位放入，冲突则从前往后找到第一个空位。

iface 和 eface 的区别是什么？值接收者和指针接收者的区别？

iface 和 eface 都是 Go 中描述接口的底层结构体，区别在于 iface 包含方法。
而 eface 则是不包含任何方法的空接口：interface{}

注意：编译器会为所有接收者为 T 的方法生成接收者为*T 的包装方法，但是链接器会把程序中确定不会用到的方法都裁剪掉。因此*T 和 T 不能定义同名方法。
生成包装方法是为了接口，因为接口不能直接使用接收者为值类型的方法。

如果方法的接收者是值类型，无论调用者是对象还是对象指针，修改的都是对象的副本，不影响调用
者；如果方法的接收者是指针类型，则调用者修改的是指针指向的对象本身。

如果类型具备"原始的本质"，如 go 中内置的原始类型，就定义值接收者就好。
如果类型具备"非原始的本质"，不能被安全的复制，这种类型总是应该被共享，则可定义为指针接收者。

context 是什么？如何被取消？有什么作用？

```
type Context interface {
    // 当 context 被取消或者到了 deadline，返回一个被关闭的 channel
    Done() <-chan struct{}
    // 在 channel Done 关闭后，返回 context 取消原因
    Err() error
    // 返回 context 是否会被取消以及自动取消时间(即 deadline)
    Deadline() (deadline time.Time,ok boll)
    // 获取 key 对应的 value
    Value(key interface{}) interface{}
}

type canceler interface {
    cancel(removeFromParent bool, err error)
    Done() <-chan struct{}
}
```

context：goroutine 的上下文，包含 goroutine 的运行状态、环境、现场等信息。

实现了 canceler 接口的 Context，就表明是可取消的。

context 用来解决 goroutine 之间退出通知、元数据传递的功能。比如并发控制和超时控制。

注意事项：
1、不要将 Context 塞到结构体里，直接将 Context 类型作为函数的第一参数，而且一般都
命名为 ctx。
2、不要向函数传入一个 nil 的 Context，如果你实在不知道传什么，标准库给你准备好了
一个 Context：todo
3、不要把本应该作为函数参数的类型塞到 Context 中，Context 存储的应该是一些共同的数据。例如：登陆的 session、cookie 等
4、同一个 Context 可能会被传递到多个 goroutine，Context 是并发安全的。

slice 的底层数据结构是怎样的？

```
type slice struct {
    array unsafe.Pointer // 底层数组的起始位置
    len int
    cap int
}
```

slice 的元素要存在一段连续的内存中，底层数据是数组，slice 是对数组的封装，它描述一个数组的片段。
slice 可以向后扩展，不可以向前扩展。
s[i]不可以超越 len(s),向后扩展不可以超越底层数组 cap(s)。
make 会为 slice 分配底层数组，而 new 不会为 slice 分配底层数组，所以 array 其实

位置会是 nil，可以通过 append 来分配底层数组。

slice 扩容方案计算：
1、预估扩容后的容量：即假设扩容后的 cap 等于扩容后元素的个数
if
    oldCap * 2 < cap，则 newCap = cap
else
    oldLen < 1024,则 newCap = oldCap * 2
    oldLen >= 1024,则 newCap = oldCap * 1.25

2、预估内存大小（int 一个元素占 8 子节，string 一个元素占 16 子节）
假设元素类型是 int，预估容量 newCap = 5，那么预估内存 = 5 * 8 = 40 byte

3、匹配到合适的内存规格（内存分配规格为 8、16、32、48、64、80....）
实际申请的内存为 48 byte， cap = 48 / 8 = 6

你了解 GC 么？常见的 GC 实现方式有哪些？

GC 即垃圾回收机制：引用计数、三色标记法+混合写屏障机制

go 的 GC 有那三个阶段？流程是什么？如果内存分配速度超过了标记清除速度怎么办？

goV1.3 之前采用的是普通标记清除，流程如下：
1、开始 STW，暂停程序业务逻辑，找出不可达的对象和可达对象；
2、给所有可达对象做上标记；
3、标记完成之后，开始清除未标记的对象；
4、停止 STW，让程序继续运行，然后循环重复这个过程，直到程序生命周期结束。

goV1.5 三色标记法，流程如下：
1、只要是新创建的对象，默认的颜色都标记为白色；
2、每次 GC 回收开始，从根节点开始遍历所有对象，把遍历到的对象从白色集合放入灰色集合；
3、遍历灰色集合，将灰色对象引用的对象从白色集合放入灰色集合，之后将此灰色对象放入黑色集合；
4、重复 3 中内容，直到灰色集合中无任何对象；
5、回收白色集合中的所有对象。

犹如剥洋葱一样，一层一层的遍历着色，但同时满足以下条件会导致对象丢失：
条件 1：一个白色对象被黑色对象引用；
条件 2：灰色对象与白色对象之间的可达关系同时被解除。

强三色：强制性的不允许黑色对象引用白色对象。
弱三色：黑色对象可以引用白色对象，但白色对象存在其他灰色对象对它的引用，或者可达它的链路上游存在灰色对象。

goV1.8 三色+混合写屏障机制，栈不启动屏障，流程如下：
1、GC 开始将栈上的对象全部扫描并标记为黑色(之后不再进行重复扫描，无需 STW)；
2、GC 期间，任何在栈上创建的新对象均标记为黑色；
3、被删除的对象和被添加的对象均标记为灰色；
4、回收白色集合中的所有对象。

总结：
v1.3 普通标记清除法，整体过程需要 STW，效率极低；
v1.5 三色标记法+屏障，堆空间启动写屏障，栈空间不启动，全部扫描之后，需要重新扫描
一次栈(需要 STW)，效率普通；
v1.8 三色标记法+混合写屏障，堆空间启动，栈空间不启动屏障，整体过程几乎不需要 STW，
效率较高。

如果申请内存的速度超过预期，运行时就会让申请内存的应用程序辅助完成垃圾收集的扫描阶段，
在标记和标记终止阶段结束之后就会进入异步的清理阶段，将不用的内存增量回收。并发标记会
设置一个标志，并在 mallocgc 调用时进行检查，当存在新的内存分配时，会暂停分配内存过快
的哪些 goroutine，并将其转去执行一些辅助标记的工作，从而达到放缓内存分配和加速 GC 工作
的目的。

内存泄漏如何解决？

1、通过 pprof 工具获取内存相差较大的两个时间点 heap 数据。htop 可以查看内存增长情况。
2、通过 go tool pprof 比较内存情况，分析多出来的内存。
3、分析代码、修复代码。

内存逃逸分析？

在函数中申请一个新的对象，如果分配在栈中，则函数执行结束可自动将内存回收；
如果分配在堆中，则函数执行结束可交给 GC 处理。

案例：
函数返回局部变量指针；
申请内存过大超过栈的存储能力。

你是如何实现单元测试的？有哪些框架？

testing、GoMock、testify

mysql

sql 语句中 group by 和 order by 谁先执行？

select (all | distinct) 字段或者表达式 (from 子句) (where 子句) (group by 子句) (having 子句) (order by 子句) (limit 子句);

1、from 子句:构成提供给 select 的数据源,所以一般后面写表名
2、where 子句:where 子句是对数据源中的数据进行过滤的条件设置
3、group by 子句:对通过 where 子句过滤后的数据进行分组
4、having 子句:对分组后的的数据进行过滤,作用和 where 类似
5、order by 子句:对取得的数据结果以某个标准(字段)进行排序,默认是 asc(正序)
6、limit 子句:对取出来的数据进行数量的限制,从第几行开始取出来几行数据
7、all | distinct 用于设置 select 出来的数据是否消除重复行，默认是 all 既不消除所有的数据都出来；distinct 表示会消除重复行

mysql 的存储引擎有哪些？都有什么区别？

MyISAM，InnoDB
区别：
1、MySAM 不支持事务、不支持外键，而 innodb 支持
2、都是 b+树作为索引结构，但是实现方式不同,innoDb 是聚集索引，myisam 是非聚集索引。
3、innoDb 不保存表的具体行数，count(*)需要全表扫描，而 myisam 用一个变量保存了整个
表的行数，速度更快。
4、innodb 组小的锁粒度是行锁，myisam 最小的锁粒度是表锁。myisam 一个更新语句会锁住
整张表，导致其他查询、更新都会被阻塞，因此并发当问受限。

为什么需要 B-树/B+树？

因为传统的树是在内存中进行的数据搜索，而当数据量非常大时，内存不够用，大部分数据只能存放在
磁盘上，只有需要的数据才加载到内存中。一般情况下内存访问的时间约为 50ns,而磁盘在 10ms 左右，
大部分时间会阻塞在磁盘 IO 上，因此要提高性能，得减少磁盘 IO 次数。

mysql 索引底层实现？

MyISAM 引擎使用 B+树作为索引结构，叶子节点的 data 域存放的是数据记录的地址，需要再寻址一次才能
得到数据，是"非聚集索引"。

InnoDB 引擎也使用 B+树作为索引结构，区别如下：
1、InnoDB 的叶子节点 data 域保存了完整的数据记录，因此主键索引很高效，是"聚集索引"。

2、InnoDB 非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询做到覆盖索引
会很高效。

hash 索引底层的数据结构是哈希表，在绝大多数需求为单条记录查询的时候，可以选择哈希索引，
查询性能最快；其余大部分场景，建议都选择 BTree 索引。

为什么 MongoDB 使用 B-树，而 mysql 使用 B+树？

Mg 是类似 json 格式的数据模型，对性能要求高，属于聚合型数据库，而 B-树恰好 key 和 data 域聚合
在一起，且最佳查询时间复杂度为 O(1);

mysql 是一种关系型数据库，B+树的特性可以增加区间访问性，而 B-树并不支持；B+树的查询时间复杂
度始终为 O(log n)，查询效率更加稳定。

B-树和 B+树的区别？

1、B+树非叶子节点只存储 key 的副本，相当于叶子节点的索引，真实的 key 和 data 域都在叶子节点存储，查询时间复杂度固定为 O(log n)，而 B-树节点内部每个 key 都带着 data 域，
查询时间复杂度不固定，与 key 在树中的位置有关，最好为 O(1)。

2、B+树叶子节点带有顺序指针，两两相连大大增加区间访问性，利用磁盘预读提前将访问节点附近的数据读入内存，减
少了磁盘 IO 的次数，也可使用在范围查询，而 B-树每个节点 key 和 data 在一起，则无法区间查找。

3、磁盘是分 block 的，一次磁盘 IO 会读取若干个 block，具体和操作系统有关，磁盘 IO 数据大小是固定的，在一
次 IO 中，单个元素越小，量就越大。由于 B+树的节点只存储 key 的副本，这就意味着 B+树单次磁盘 IO 的数据大于 B-树，
自然磁盘 IO 次数也更少。

覆盖索引是什么？

从索引中就能查到记录，而不需要索引之后再回表中查记录，避免了回表的产生，减少了树的搜索次数。

索引设计的原则？需要注意事项？

1、出现在 where 子句中的列，或者连接子句中指定的列。
2、基数较小的列，索引效果较差，没有必要在此列建立索引。
3、取值离散大的字段放在联合索引的前面，count()值越大说明字段唯一值越多，离散程度高。

4、尽量使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，节省索引空间。
5、应符合最左前缀匹配原则，mysql 会一直向右匹配直到遇到范围查询(>、<、between、like)
就停止匹配，其中(=和 in)可以乱序，mysql 查询优化器会优化成索引可以识别的形式。

注意：
1、不要过度索引，并非索引越多也好，索引需要空间来存储，也需要定期维护，并且索引会降低
写操作的性能。
2、非必要情况，所有列都应该指定列为 NOT NULL，使用零值作为默认值。

什么是数据库事务？

事务是一个不可分割的数据操作序列，也是数据库并发控制的基本单位，其执行结果必须使数据库
从一种一致性状态变到另一种一致性状态。事务是逻辑上的一组操作，要么都执行，要么都不执行。

事务的四大特性(ACID)?

1、原子性：事务是最小的执行单位，包含的所有数据库操作要么全部成功，要么全部失败回滚；
2、一致性：执行事务前后数据都必须处于一致性状态；
3、隔离性：一个事务未提交的业务结果是否对于其它事务可见，对应有四种事务隔离级别。
4、持久性：一个事务一旦被提交了，那么对数据库中数据的改变就是永久性的，即使是在数据库
系统遇到故障的情况下也不会丢失提交事务的操作。

事务隔离级别有哪些？如何解决脏读和幻读？

数据库定义了四种不同的事务隔离级别，由低到高依次为：
Read uncommitted 读取未提交-----三者均可能
Read committed 读取已提交------解决了脏读，可能出现不可重复读和幻读
Repeatable read 可重复读------可能出现幻读，解决了脏读和不可重复读
Serializable 可串行化（对所涉及到的表加锁，并非行级锁）

脏读:读取未提交数据(A 事务读到未提交的 B 事务数据)
解决：设置事务级别为 Read committed

不可重复读:前后多次读取，数据内容不一致(原因：读取了其它事务更改的数据且提交了事务，针对 update 操作)
解决：使用行级锁，锁定该行，事务 A 多次读取操作完成后才释放该锁，才允许其它事务操作。

幻读：前后多次读取，数据总量不一致(原因：读取了其它事务新增的数据且提交了事务，

针对 insert 和 delete 操作)
解决：使用表级锁，锁定整张表，事务 A 多次读取数据总量之后才释放该锁，才允许其它事务操作。

mysql 的事务隔离级别默认为 Repeatable read(可重复读，可能会出现幻读)

乐观锁和悲观锁的实现方式？

乐观锁：基于数据版本号实现(基于 mvcc 理论)
悲观锁：数据库的锁机制

innoDB 存储引擎的锁的算法有那三种？

Record lock：单个行记录上的锁
Gap lock：间隙锁，锁定一个范围，不包括记录本身
Next-key lock：record+gap 锁定一个范围，包含记录本身

注意：(间隙：范围查询中，存在于范围内，但不存在的记录，这称为间隙)

mysql 的 mvcc 实现原理是什么？

MVCC 只适用 mysql 事务隔离级别：Read committed 和 Repeatable Read，MVCC 的版本是在事务提交之后才会产生。

多版本并发控制：Undo log 实现 MVCC，并发控制通过锁来实现。
简单来说就是在每一行记录的后面增加两个隐藏列，记录创建版本号和删除版本号，而每一个事务在启动的时候，都有一个唯一的递增的版本号，通过版本号来减少锁的争用。

1、在插入操作时，记录的创建版本号就是事务版本号；
2、在更新操作时，采用的是先标记旧的那行记录为已删除，并且删除版本号是事务版本号，然后插入一行新的记录；
3、删除操作的时候，就把事务版本号作为删除版本号；
4、查询时，符合删除版本号大于当前事务版本号并且创建版本号小于或者等于当前事务版本号
这两个条件的记录才能被事务查询出来。

bin log、redo log、undo log 作用是什么？有什么区别？

bin log:
记录 mysql 服务层产生的日志，常用来进行数据恢复、数据库复制。

redo log:
记录了数据操作在物理层面的修改。mysql 中大量使用缓存，缓存存在与内存中，
修改操作时会直接修改内存，而不是立刻修改磁盘，当内存和磁盘数据不一致时，称内存中
的数据为脏页。

为了保证数据的安全性，事务进行时会不断产生 redo log，在事务提交时进行一次 flus

h 操作，
保存到磁盘中，redo log 是按照顺序写入的，磁盘的顺序读写的速度远大于随机读写。
当数据
库或主机失效重启时，会根据 redo log 进行数据的恢复，如果 redo log 中有事务提交，则进行
事务提交修改数据。这样实现了事务的原子性、一致性、持久性。

undo log:
当进行数据修改时除了记录 redo log，还会记录 undo log，它记录了修改的反向操作，用于
数据的撤回操作，可以实现事务回滚，mvcc 就是根据 undo log 实现回溯到某个特定的版本的
数据的。

大表数据查询，如何优化？

1、优化 sql 语句+索引
2、增加缓存，memcached、redis
3、做主从复制，读写分离
4、拆表---垂直拆分、水平拆分

Mysql 数据库 cpu 飙升如何排查？

1、首先 top 查看是否真是由于 mysqld 占用导致的。
2、show processlist，分析 session 情况，有没有激增，是不是有消耗资源的 sql 在运行。
3、找出消耗高的 sql，explain 查看执行计划。

主从复制的实现步骤？

1、主库 db 的操作事件被写入到 binlog
2、从库发起连接，连接到主库
3、此时主库创建一个 binlog dump thread 线程，把 binlog 的内容发送到从库
4、从库启动之后，创建一个 I/O 线程，读取主库传过来的 binlog 内容并写入到 relay log
5、还会创建一个 SQL 线程，从 relay log 里面读取内容，从 Exec_Master_Log_Pos 位置
开始执行读取到的操作事件，将内容写入到 slave 的 db

kafka

kafka 为什么性能高？

1、kafka 本身是分布式集群，同时采用了分区技术，具有较高的并发度；
2、顺序写入磁盘，Kafka 的 producer 生产数据，要写入到 log 文件中，写的过程是一直追加到文件末端，为顺序写。
3、零拷贝技术

kafka 重复消费可能的原因以及处理方式？

原因 1：消费者宕机、重启等，导致消息已经消费但是没有提交 offset；
原因 2：消费者使用自动提交 offset，但当还没有提交的时候，有新的消费者加入或者移除，发生了 rebalance。
再次消费的时候，消费者会根据提交的偏移量来，于是重复消费了数据。
原因 3：消息处理耗时，或者消费者拉取的消息量太多，处理耗时，超过了 max.poll.interval.ms 的配置时间，
导致认为当前消费者已经死掉，触发再均衡。

解决方案：消费者实现消费幂
1、消费表
2、数据库唯一索引
3、缓存消费过的消息 ID

触发重平衡(rebalanced)的情况？

1、有新的消费者加入消费组、或已有消费者主动离开组
2、消费者超过 session 时间未发送心跳（已有 consumer 崩溃了）
3、一次 poll()之后的消息处理时间超过了 max.poll.interval.ms 的配置时间，因为一次 poll()
处理完才会触发下次 poll()（已有 consumer 崩溃了）
4、订阅主题数发生变更
5、订阅主题的分区数发生变更

kafka 消息丢失的原因以及解决方式？

生产者丢失消息情况：可能因为网络问题并没有发送出去。
解决：可以给 send 方法添加回调函数,按一定次数、间隔重试。

消费者丢失消息情况：消费者自动提交 offset，拿到消息还未真正消费，就挂掉了，但是 offset 却被自动提交了。
解决：关闭自动提交 offset，每次在真正消费完消息之后之后再自己手动提交 offset，这样解决了消息丢失，但会带来重复消费问题。

kafka 丢失消息情况：
leader 副本所在的 broker 突然挂掉，那么就要从 follower 副本重新选出一个 leader ，但是 leader 的数据还有一些
没有被 follower 副本的同步的话，就会造成消息丢失。

解决：设置 ack = all。ack 是 Kafka 生产者很重要的一个参数。代表则所有副本都要接收到该消息之后该消息才算真正成功被发送。

Kafka 中的消息有序吗？

kafka 无法保证整个 topic 多个分区有序，但是由于每个分区（partition）内，
每条消息都有一个 offset，故可以保证分区内有序

topic 的分区数可以增加或减少吗？为什么？

topic 的分区数只能增加不能减少，因为减少掉的分区也就是被删除的分区的数据难以处理.

注意：消费者组中的消费者个数如果超过 topic 的分区，那么就会有消费者消费不到数据.

kafka 是怎么维护 offset 的？

维护 offset 的原因：
由于 consumer 在消费过程中可能会出现断电宕机等故障，consumer 恢复后，需要从故障前的位置
的继续消费，所以 consumer 需要实时记录自己消费到了哪个 offset，以便故障恢复后继续消费。

维护 offset 的方式：
Kafka 0.9 版本之前，consumer 默认将 offset 保存在 Zookeeper 中，从 0.9 版本开始，
consumer 默认将 offset 保存在 Kafka 一个内置的 topic 中，该 topic 为**__consumer_offsets**。

关于 offset 的常识：
消费者提交消费位移时提交的是当前消费到的最新消息的 offset+1 而不是 offset。

kafka 集群消息积压问题如何处理？

从两个角度去分析：
1、如果是 Kafka 消费能力不足，则可以考虑增加 Topic 的分区数，并且同时提升消费组的消费者数量，消费者数=分区数。（两者缺一不可）
2、如果是下游的数据处理不及时，提高每批次拉取的数量。如果是因为批次拉取数据过少
（拉取 数据/处理时间<生产速度），也会使处理的数据小于生产的数据，造成数据积压。

redis

redis 单线程为什么效率也这么高？

1. redis 是基于内存的，内存的读写速度非常快.
2. redis 是单线程的，省去了很多上下文切换线程的时间.
3. IO 多路复用

redis 有那五种常用的数据结构？应用场景以及实现原理是什么？

ziplist：压缩列表，此数据结构是为了节约内存而开发
intset：整数集合，是集合键的底层实现方式之一
quicklist：ziplist 的一个双向链表
skiplist：跳表

1. string 计数 sds(raw，embstr，int)
2. hash 缓存结构数据 quicklist(hashtable，ziplist)
3. list 异步消息队列 (ziplist，linkedlist)
4. set(无序、成员唯一) 计算共同喜好(交集)、统计访问 ip (intset，hashtable)
5. zset(有序、成员唯一) 排行榜、延迟队列 (ziplist，skiplist)

redis 的过期策略？

定期删除+惰性删除策略

- 定期删除策略：Redis 启用一个定时器定时监视所有的 key，判断 key 是否过期，过期的话就删除。
这种策略可以保证过期的 key 最终都会被删除，但是也存在严重的缺点：每次都遍历内存中所有的数据，
非常消耗 CPU 资源，并且当 key 已过期，但是定时器还处于未唤起状态，这段时间内 key 仍然可以用。

- 惰性删除策略：在获取 key 时，先判断 key 是否过期，如果过期则删除。这种方式存在一个缺点：
如果这个 key 一直未被使用，那么它一直在内存中，其实它已经过期了，会浪费大量的空间。

- 这两种策略天然的互补，结合起来之后，定时删除策略就发生了一些改变，不在是每次扫描全部的
key 了，而是随机抽取一部分 key 进行检查，这样就降低了对 CPU 资源的损耗，惰性删除
策略互补了为检查到的 key，基本上满足了所有要求。但是有时候就是那么的巧，既没有被定时器抽取到，
又没有被使用，这些数据又如何从内存中消失？没关系，还有内存淘汰机制，当内存不够用时，内存淘汰机制就会上场。

Redis 中的批量操作 Pipeline？

非 pipeline：client 一个请求，redis server 一个响应，期间 client 阻塞。
Pipeline：redis 的管道命令，允许 client 将多个请求依次发给服务器，过程中不需要等待请求的回复，而是在最后读取所有结果。

redis 与 mysql 数据一致性解决方案？

延迟双删策略：
1、先删除缓存，然后更新数据库，但可能在更新未完成之前，有请求穿透到 db 取了旧数据并写入了缓存，因此需要更新完数据库之后，延迟几十毫秒，再删一次缓存。
2、先更新数据库，再删除缓存，再延迟删一次。
如果删除失败则重试，比如放入队列循环删除。

发生缓存穿透、击穿、雪崩的原因以及解决方案？

- 缓存穿透
是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据
则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，
可能 DB 就挂掉了，要是有人利用不存在的 key 频繁攻击我们的应用，这就是漏洞。(简单来说就是缓存和数据库
都不存在这个数据，这种情况称为穿透)
解决方案：
1、接口层增加校验，比如 id<=0 这种一定不存在的情况直接拦截掉。
2、如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，
但它的过期时间会很短，最长不超过五分钟。
3、采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这
个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。

- 缓存雪崩
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。重启导致缓存失效，也可能出现并发到 DB。
解决方案：
1、考虑用加锁(互斥锁)，锁定 key 之后完成 db 的查询以及缓存的更新之后再释放锁定 key，从而避免失效时大量的并发请求落到底层存储系统上。
2、缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。
3、重启导致缓存失效，我们可以采用缓存预热，提前使用一个接口更新好缓存，再启动服务。

- 缓存击穿(场景：热点数据)
缓存中不存在，但数据库中有的数据(一般是缓存时间到期)。缓存在某个时间点过期的时候，恰好在这个时间点对这个 Key 有大量的并发
请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端 DB 压垮。与雪崩区别是击穿指并发查同一条数据。
解决方案：
1、使用互斥锁(SETNX)
2、设置热点数据永远不过期，数据是需要维护的。

redis 的持久化方案 RDB 和 AOF 详解？

RDB:在指定的时间间隔内将内存中的数据集快照写入磁盘，它恢复时就是将快照文件直接读到内存里。
AOF:持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命

令来还原数据集。AOF 文件中的命令全部以 Redis 协议的格式来保存，
新命令会被追加到文件的末尾,Redis 还可以在后台对 AOF 文件进行重写(rewrite)，
使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小

网络协议

浏览器访问一个网站，都经历了怎样一个流程？

1、DNS 解析:将域名解析成 IP 地址
2、TCP 连接：TCP 三次握手
3、发送 HTTP 请求
4、服务器处理请求并返回 HTTP 报文
5、浏览器解析渲染页面
6、断开连接：TCP 四次挥手

什么是 HTTP 与 HTTPS 有什么区别？

1、HTTP 的 URL 以 http:// 开头，而 HTTPS 的 URL 以 https:// 开头
2、HTTP 是不安全的，而 HTTPS 是安全的
3、HTTP 标准端口是 80 ，而 HTTPS 的标准端口是 443
4、在 OSI 网络模型中，HTTP 工作于应用层，而 HTTPS 的安全传输机制工作在传输层
5、HTTP 无法加密，而 HTTPS 对传输的数据进行加密
6、HTTP 无需证书，而 HTTPS 需要 CA 机构颁发的 SSL 证书

什么是 Http 协议无状态协议?怎么解决 Http 协议无状态协议?

无状态协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息。
也就是说，当客户端一次 HTTP 请求完成以后，客户端再发送一次 HTTP 请求，HTTP 并不知道当前客户端是一个”老用户“。
可以使用 Cookie 来解决无状态的问题，Cookie 就相当于一个通行证，第一次访问的时候给客户端发送一个 Cookie，
当客户端再次来的时候，拿着 Cookie(通行证)，那么服务器就知道这个是”老用户“

HTTP 请求报文与响应报文格式？

- 请求报文包含：
  1、请求行：包含请求方法、URI、HTTP 版本信息
  2、请求头部字段
  3、空行
  4、请求内容实体

- 响应报文包含：
  1、状态行：包含 HTTP 版本、状态码、状态码的原因短语
  2、响应头部字段
  3、空行
  4、响应内容实体

HTTP 常见的状态码有哪些？

1XX 系列：指定客户端应相应的某些动作，代表请求已被接受，需要继续处理。由于 HTTP/1.0 协议中没有定义任何
1xx 状态码，所以除非在某些试验条件下，服务器禁止向此类客户端发送 1xx 响应。

2XX 系列：代表请求已成功被服务器接收、理解、并接受。这系列中最常见的有 200、201 状态码。
200 状态码：表示请求已成功，请求所希望的响应头或数据体将随此响应返回
201 状态码：表示请求成功并且服务器创建了新的资源，且其 URI 已经随 Location 头信息返回。假如需要的资源
无法及时建立的话，应当返回 ‘202 Accepted’

202 状态码：服务器已接受请求，但尚未处理

3XX 系列：代表需要客户端采取进一步的操作才能完成请求，这些状态码用来重定向，后续的请求地址（重定向目标
）在本次响应的 Location 域中指明。这系列中最常见的有 301、302 状态码。
301 状态码：被请求的资源已永久移动到新位置。服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动
将请求者转到新位置。

302 状态码：请求的资源临时从不同的 URI 响应请求，但请求者应继续使用原有位置来进行以后的请求

304 自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容。 如果网页自请求者上次请
求后再也没有更改过，您应将服务器配置为返回此响应(称为 If-Modified-Since HTTP 标头)。

4XX 系列：表示请求错误。代表了客户端看起来可能发生了错误，妨碍了服务器的处理。常见有：401、404 状态码。
401 状态码：请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。
403 状态码：服务器已经理解请求，但是拒绝执行它。与 401 响应不同的是，身份验证并不能提供任何帮助，而且
这个请求也不应该被重复提交。

404 状态码：请求失败，请求所希望得到的资源未被在服务器上发现。没有信息能够告诉用户这个状况到底是暂
时的还是永久的。假如服务器知道情况的话，应当使用 410 状态码来告知旧资源因为某些内部的配置机制问题，已
经永久的不可用，而且没有任何可以跳转的地址。404 这个状态码被广泛应用于当服务器不想揭示到底为何请求被
拒绝或者没有其他适合的响应可用的情况下。

5xx 系列：代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬

件资源无法完成对请求的处理。常见有 500、503 状态码。
500 状态码：服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在
服务器的程序码出错时出现。
503 状态码：由于临时的服务器维护或者过载，服务器当前无法处理请求。通常，这个是暂时状态，一段时间会恢复

网络的七层结构及其作用？

- 应用层（数据）：确定进程之间通信的性质以满足用户需要以及提供网络与用户应用
- 表示层（数据）：主要解决用户信息的语法表示问题，如加密解密
- 会话层（数据）：提供包括访问验证和会话管理在内的建立和维护应用之间通信的机制，如服务器验证用户登录便是由会话层完成的
- 传输层（段）：实现网络不同主机上用户进程之间的数据通信，可靠与不可靠的传输，传输层的错误检测，流量控制等
- 网络层（包）：提供逻辑地址（IP）、选路，数据从源端到目的端的传输
- 数据链路层（帧）：将上层数据封装成帧，用 MAC 地址访问媒介，错误检测与修正
- 物理层（比特流）：设备之间比特流的传输，物理接口，电气特性等

TCP/IP 协议四层?

应用层、传输层、网络层、数据链路层

TCP 协议和 UDP 协议有什么区别？

CP 和 UDP 协议属于传输层协议，主要区别：
1、TCP 是面向连接的，UDP 是无连接的;
2、TCP 是可靠的，UDP 是不可靠的;
3、TCP 只支持点对点通信，UDP 支持一对一、一对多、多对一、多对多的通信模式;
4、TCP 是面向字节流的，UDP 是面向报文的;
5、TCP 有拥塞控制机制;UDP 没有拥塞控制，适合媒体通信;
6、TCP 首部开销(20 个字节)比 UDP 的首部开销(8 个字节)要大;

TCP 协议的三次握手和四次挥手？为什么是三次和四次？

- 三次握手(我要和你建立链接，你真的要和我建立链接么，我真的要和你建立链接，成功)

第一次握手：Client 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 Server，Client 进入 SYN_SENT 状态，
等待 Server 确认。

第二次握手：Server 收到数据包后由标志位 SYN=1 知道 Client 请求建立连接，Server 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 Client 以确认连接请求，Server 进入 SYN_RCVD 状态。

第三次握手：Client 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 Server，Server 检查 ack 是否为 K+1，ACK 是否为 1，如果正确则连接建立成功，Client 和 Server 进入 ESTABLISHED 状
态，完成三次握手，随后 Client 与 Server 之间可以开始传输数据了。

- 四次挥手(我要和你断开链接;好的，断吧。我也要和你断开链接;好的，断吧)

第一次挥手：Client 发送一个 FIN，用来关闭 Client 到 Server 的数据传送，Client 进入 FIN_WAIT_1 状态。

第二次挥手：Server 收到 FIN 后，发送一个 ACK 给 Client，确认序号为收到序号+1 (与 SYN 相同，一个 FIN 占用一个序号)，Server 进
入 CLOSE_WAIT 状态。此时 TCP 链接处于半关闭状态，即客户端已经没有要发送的数据了，但服务端若发送数据，则客户端仍要接收。

第三次挥手：Server 发送一个 FIN，用来关闭 Server 到 Client 的数据传送，Server 进入 LAST_ACK 状态。

第四次挥手：Client 收到 FIN 后，Client 进入 TIME_WAIT 状态，接着发送一个 ACK 给 Server，确认序号为收到序号+1，Server 进
入 CLOSED 状态，完成四次挥手。

三次握手：目的是为了防止已失效的链接请求报文突然又传送到了服务端，因而产生错误。

四次挥手：TCP 协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP 是全双工模式，这就意味着，当 A
向 B 发出 FIN 报文段时，只是表示 A 已经没有数据要发送了，而此时 A 还是能够接受到来自 B 发出的数据;B 向 A 发
出 ACK 报文段也只是告诉 A ，它自己知道 A 没有数据要发了，但 B 还是能够向 A 发送数据。

http1.0、http1.1、http2.0 有什么区别？

http1.0 和 http1.1 区别：
1、http1.1 默认开启长连接 keep-alive，在一个 TCP 连接(一次完整的 tcp 握手和挥手)上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟。
2、http1.0 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 支持只发送 header 信息（不带任何 body 信息），
如果服务器认为客户端有权限请求服务器，则返回 100，客户端接收到 100 才开始把请求 body 发送到服务器；如果返回 401，客户端就可以不用发送请求 body 了节约了带宽。
3、http1.1 的请求消息和响应消息都支持 host 域，且请求消息中如果没有 host 域会报告一个错误（400 Bad Request）。

http1.1 和 http2.0 区别：

1、http2.0 在 1.1 的基础上增加了多路复用，做到同一个连接并发处理多个请求，而且并发请求的数量比 HTTP1.1 大了好几个数量级。
http1.1 也可以多建立几个 TCP 连接，来支持处理更多并发的请求，但是创建 TCP 连接本身也是有开销的。
2、http1.1 不支持 header 数据的压缩，http2.0 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。
3、http2.0 引入了 server push，它允许服务端推送资源给浏览器，免得客户端再次创建连接发送请求到服务器端获取。(例如：客户端向服务器发送一个获取 html 的请求，不需要再次请求去获取 html 所依赖的 css、js，而是在第一次 html 请求时，服务器端主动的都推送给客户端)

ProtoBuff 协议相比其它有什么好处？

pb 是一种轻便高效的"结构化数据"存储格式，可以用于"结构化数据"的序列化和反序列化。
1、跨语言，支持大多数语言开发，代码开源，运行稳定可靠。
2、性能好、效率高，占据空间和运行时间相比 json 和 xml 小，二进制序列化格式，数据压缩紧凑，占据字节数小。
3、支持向后向前兼容，比如向后兼容：A、B 两模块，B 升级有"statue"属性，可设置为非必填或者缺省，这样 A 就被兼容了。
4、适合数据大、传输速率敏感的场合使用。
5、支持数据类型多。
6、数据结构化定义灵活，可嵌套定义。

分布式系统、微服务架构

什么是分布式事务？

二阶段提交、三阶段提交

分布式锁有哪些实现方式？分别会存在什么问题，哪种实现更好？

zk(ZooKeeper)锁实现原理：
（1）创建一个目录 mylock；
（2）线程 A 想获取锁就在 mylock 目录下创建临时顺序节点；
（3）获取 mylock 目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则说明当前线程顺序号最小，获得锁；
（4）线程 B 获取所有节点，判断自己不是最小节点，设置监听比自己次小的节点；
（5）线程 A 处理完，删除自己的节点，线程 B 监听到变更事件，判断自己是不是最小的节点，如果是则获得锁。

redis 事务：
1、事务提供了一种将多个命令打包，然后一次性有序（FIFO）执行的机制
2、事务执行过程不会被中断
3、带 WATCH 命令的事务会将客户端和被监视的键在数据库 watched_keys 字典中进行关联，当键被修改程序会将所有监视键的客
户端 REDIS_DIRTY_CAS 标识打开

4、在客户端提交 EXEC 命令时，会检查 REDIS_DIRTY_CAS 标识，标识打开事务将不会被执行
5、Redis 事务具有 ACID 的特性（当服务器运行在 AOF 模式下，并且 appendfsync 选项值为 always 时才具有持久性

redis 实现分布式锁：
2.6.12 版本之后命令： SET key value EX 10 NX （合并了 1、2 两个步骤）
核心思路：
1、使用 setnx 设置互斥锁
2、为了避免异常情况导致死锁，因此需要为锁设置过期时间
3、为了避免删锁混乱，导致锁永久失效，需要为每个请求分配唯一的 value 值，再删锁时，验证是否属于自身的锁。
4、为了避免在删锁的操作过程中的异常情况，如锁过期，新的请求获得锁，此时删除的是新的锁。可以再执行任务中新启一个协程每隔 10s 去检查主程是否还持有锁，
如果还持有锁，则为锁进行续期。

基于 lua 脚本实现 redis 的乐观锁

说下微服务架构有哪些组件，这些组件是如何实现自身功能的？

微服务架构如何设计？

如何防止超卖？

1、使用 redis 的队列来实现。将要促销的商品数量以队列的方式存入 redis 中，每当用户抢到一件促销商品则从队列中删除一个数据，确保商品不会超卖

2、乐观锁，增加版本号，查询库存和更新库存时比较版本号

数据结构与算法

题目：将 6，2，10，32，9，5，18，14，30，29 从小到大进行排列,使用冒泡排序

```go
package main

import "fmt"

func main() {
    // 定义数组
    arr := [10]int{6, 2, 10, 32, 9, 5, 18, 14, 30, 29}
    for i := 0; i < len(arr); i++ {
        for j := 0; j < len(arr)-i-1; j++ {
            if arr[j] > arr[j+1] {
```

```
                arr[j], arr[j+1] = arr[j+1], arr[j]
            }
        }
    }
    fmt.Println(arr)
}
```

快排

选择排序

堆排

动态规划

其它

关系型数据库和非关系型数据库有什么区别？