|  |  |
| --- | --- |
|  | 字节 |
| 1 | go channel close后读的问题 |
| 2 | Linux grep命令查找日志文件相关内容 |
| 3 | B+树结构 and 为什么 |
| 4 | io多路复用，epoll和select的区别 |
| 5 | 计网七层协议、线程进程区别 |
| 6 | 线程怎么调度 |
| 7 | 进程通信方法 |
| 8 | tcp 保证可靠性 |
| 9 | go slice 和 array 区别 |
| 10 | GMP 模型 |
| 11 | 分布式缓存框架，singleflight 并发，深挖，一直挖到sync.WaitGroup |
| 12 | 缓存击穿 缓存雪崩 |
| 13 | sync.WaitGroup |
| 14 | cookie session,session如何存储, 多台服务器session存储怎么设计,除了存redis 还能怎么存 |
| 15 | 设计学生成绩数据库，并写出查询语文成绩top3的人。 url输入全过程 。从此引出后端除了响应请求还有什么。 |
| 16 | 什么是分布式系统 |
| 17 | 由上面分布式系统 引出负载均衡 |
| 18 | 负载均衡算法有哪些 |
| 19 | 一致性哈希算法 深挖。一致性哈希与普通哈希的区别。 |
| 20 | GMP模型 |
| 21 | 手撕代码 LRU |
| 22 | 手撕代码 生产者消费者模型 |
| 23 | 手撕代码 反转连表 |
| 24 | 算法题：三数之和 |
| 25 | http 和 https区别 |
| 26 | https详细过程 |
| 27 | get post区别。还有什么其他方法。分别说说是做什么的。 |
| 28 | web安全问题。 |
| 29 | 设计一个短链接服务。如何抗住大qps， 抗大流量 。url 哈希函数怎么设计（怎么存，怎么统计qps） |
|  | 腾讯 |
| 1 | tcp和udp的特性，tcp三次握手、四次挥手 |
| 2 | http2.0了解吗？和http1.1的差距？ |
| 3 | 反爬了解吗？反爬的几项技术在项目里面怎么实现的？ |
| 4 | 项目部署的服务器是单机的，请问如果是大流量高并发请求服务器怎么处理？ |
| 5 | 数据结构：常用排序算法，快排、堆排的原理和实现。 |
| 6 | 算法：topK，时间复杂度 |
| 7 | 如果是亿级数据怎么处理？ |
| 8 | b树 b+树区别 |
| 9 | tcp 可靠性，然后问十六位校验和怎么实现的 |
| 10 | TCP 粘包 |
| 11 | 进程 协程 线程 |
| 12 | 跳表怎么实现 |
| 13 | go的调度 |
| 14 | go struct能不能比较？ |
| 15 | go defer（for defer） |
| 16 | select可以用于什么？ |
| 17 | context包的用途？ |
| 18 | client如何实现长连接？ |
| 19 | 主协程如何等其余协程完再操作 |
| 20 | slice，len，cap，共享，扩容 |
| 21 | map如何顺序读取？ |
| 22 | 实现set |
| 23 | 实现消息队列（多生产者，多消费者） |
| 24 | 大文件排序 |
| 25 | 基本排序，哪些是稳定的 |
| 26 | http get跟head |
| 27 | http 401,403 |
| 28 | http keep-alive |
| 29 | http能不能一次连接多次请求，不等后端返回 |
| 30 | tcp与udp区别，udp优点，适用场景 |
| 31 | time-wait的作用 |
| 32 | 数据库如何建索引 |
| 33 | 孤儿进程，僵尸进程 |
| 34 | 死锁条件，如何避免 |
| 35 | linux命令，查看端口占用，cpu负载，内存占用，如何发送信号给一个进程 |
| 36 | git文件版本，使用顺序，merge跟rebase |
| 37 | 项目实现爬虫的流程 |
| 38 | 爬虫如何做的鉴权吗？ |
| 39 | 怎么实现的分布式爬虫 |
| 40 | 电商系统图片多会造成带宽过高，如何解决？ |
| 41 | micro服务发现 |
| 42 | mysql底层有哪几种实现方式 |
| 43 | channel底层实现 |
| 44 | java nio和go 区别 |
| 45 | 读写锁底层是怎么实现的？ |
| 46 | go-micro 微服务架构怎么实现水平部署的，代码怎么实现？ |
| 47 | micro怎么用 |
| 48 | 怎么做服务发现的 |
| 49 | mysql索引为什么要用B+树？ |
| 50 | mysql语句性能评测？ |
| 51 | 服务发现有哪些机制 |
| 52 | raft算法是那种一致性算法 |
| 53 | raft有什么特点 |
| 54 | 当go服务部署到线上了，发现有内存泄露，该怎么处理 |
| 55 | https 握手，为什么需要 非对称加密 和 对称加密 |
|  | 金山wps |
| 1 | PHP-FPM |
| 2 | CGI 是一个 Web Server 与 CGI 程序之间进行数据传输的协议，保证传递的是标准数据 |
| 3 | PHP-CGI 是 PHP 解析器（CGI 程序） |
| 4 | FastCGI 是用来提高 CGI 程序性能的方案/协议。FastCGI 会先启动一个 master，解析配置文件，初始化执行环境，然后再启动多个 worker。当请求过来时，master 会传递给一个 worker，然后立即可以接收下一个请求，避免重复劳动，提高效率 |
| 5 | PHP-FPM 是实现 FastCGI 的程序 |
| 6 | PHP 和 Go 对比 |
| 7 | MVC 模式 |
| 8 | MySQL 优化（索引、分表分库） |
| 9 | 浏览器地址栏输入网址整个过程 |
| 10 | TCP 三次握手和四次挥手 |
| 11 | Linux 相关（介绍了一下基本操作命令） |
| 12 | Docker 相关 |
| 13 | Git 相关 |
| 14 | 开发环境（Windows、Linux） |
| 15 | CI / CD 发布流程 |
| 16 | HTTP 长连接（HTTP 1.1 版本） |
| 17 | 腾讯音乐 |
| 18 | TCP 拥塞控制（快速恢复、快速重传） |
| 19 | 从面向连接的特性回答 |
| 20 | UDP 实现可靠连接 |
| 21 | 从 TCP 可靠连接特性回答 |
| 22 | 四、MySQL 数据库 |
| 23 | MySQL 索引数据结构 |
| 24 | 索引为什么使用 B+ 树 |
| 26 | 为什么不使用 Hash 结构 |
| 27 | like 模糊查询 |
| 28 | 范围查询 |
| 29 | 结合事务隔离级别 |
| 30 | Go 语言相关 |
| 31 | slice 和 array 区别 |
| 32 | 向为 nil 的 channel 发送数据会怎么样 |
| 33 | map 取一个 key，然后修改这个值，原 map 数据的值会不会变化 |
| 34 | 根据 map 存储的类型回答 |
| 35 | for 循环遍历 slice 有什么问题 |
| 36 | Go 闭包 |
| 37 | 进程、线程、协程区别 |
| 38 | 技术相关 |
| 39 | 输入 URL 发生的整个网络过程 |
| 40 | Redis 怎么保证数据一致性 |
| 41 | TCP 流量控制、拥塞控制 |
| 42 | TCP 半连接队列 |
| 43 | TCP 半关闭状态 |
| 44 | TCP TIME\_WAIT 状态 |
| 45 | 内核态、用户态 |
| 46 | Hash 实现、冲突解决、应用 |
| 47 | 快速排序 |
| 48 | 堆排序 |
| 49 | 大小堆 |
| 51 | 100 枚硬币，其中有一枚硬币重量不一样，用天平秤怎么快速找到这一枚硬币 |
|  | 跟谁学 |
| 1 | 说一下Redis中HashMap的实现(双table,渐进式rehash,扩容条件,缩容条件,bgsave,CopyOnWrite机制) |
| 2 | 扩容过程中有新的请求流程 |
| 3 | Redis其他的数据结构(SDS,RAW,INTSET,ZIPLIST,SKIPLIST,QUICKLIST) |
| 4 | 跳表的实现? |
| 5 | Redis的定时任务怎么实现的? |
| 6 | 订单服务过期是怎么设计的(RabbitMQ死信队列) |
|  | 百度 |
| 1 | 数据库问题，给你10个数据库服务器，每个只能接500的qps，现在要实现4000qps，要怎么做？说用负载均衡，使用binlog保证10个服务器的数据一致性 |
| 2 | 如果有有读有写，如何实现高并发，数据库读写分离 |
| 3 | 对于两个写库，两个请求向分别打到两个写库中，他们互相向对方同步，会不会出现不一致， |
| 4 | 哈希的实现有哪几种，如何取hashcode，冲突检测几种方法 |
| 5 | 用过go，那么进程，协程，线程各自的优缺点 |
| 6 | 算法题 z遍历二叉树，循环有序数组找指定值， |
| 7 | 1.事务是怎么实现的?(undo\_log,MVCC) |
| 8 | mongodb和redis的区别 |
| 9 | 请你说说golang的CSP思想 |
| 10 | go 内存逃逸分析（分析了栈帧，讲五种例子，描述堆栈优缺点，点头） |
| 11 | 是否有逃逸分析过 |
| 12 | defer recover 的问题 |
| 13 | mysql 索引慢分析（线上开启slowlog，提取慢查询，然后仔细分析explain 中 tye字段以及extra字段，发生的具体场景及mysql是怎么做的 |

# 字节跳动

## Go channel close后读问题

给close channel发送数据,引起panic

从close channel 接收数据,如果缓冲区不为空,则接收到缓冲区数据,

如果缓冲区为空,则接收到零值

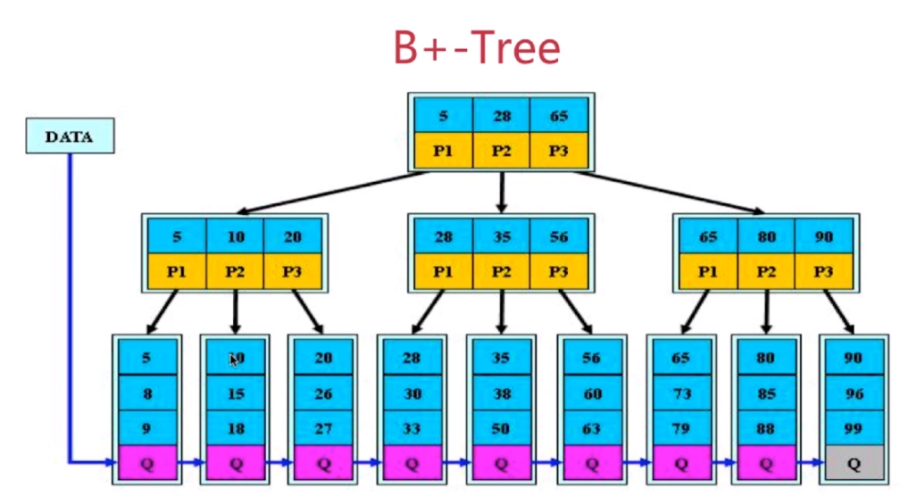
## Linux grep查找日志文件相关内容

grep [查找内容] [查找文件]

-r 递归查找

grep -r [查找内容] [查找目录]

## B+树的结构,为什么是这样的结构



特点:

1. 非叶子节点的子树指针与关键字个数相同
2. 非叶子节点的子树指针p[i],指向关键字的值[k[i],k[i+1])
3. 非叶子节点仅用来索引,数据保存在叶子节点中
4. 所有叶子节点均有一个链指针指向下一个叶子节点

优点:

1. B+树相对树的层级较低,磁盘读写代价更低
2. B+树查询效率更加稳定,所有叶子节点查询效率时间复杂度为O(log n)
3. 有利于数据库扫描

## IO多路复用,select,poll和epoll的区别

|  |  |  |  |
| --- | --- | --- | --- |
|  | select | poll | epoll |
| 底层数据结构 | 数组 | 链表 | 红黑树+双向链表 |
| 如何获取就绪fd | 遍历fd\_set | 遍历链表 | 回调 |
| 时间复杂度 | O(n) | O(n) | O(1) |
| fd数据拷贝 | 每次调用select，需要将fd数据从用户空间拷贝到内核空间 | 每次调用poll，需要将fd数据从用户空间拷贝到内核空间 | 使用内存映射(mmap)，不需要从用户空间频繁拷贝fd数据到内核空间 |
| 最大连接数 | 32位机一般1024  64位机一般位2048 | 无限制 | 无限制 |

IO多路复用

IO多路复用模型是一种同步IO模型,由单线程监控多个IO事件,有IO事件就绪时,就会通知线程执行相应的读写操作,没有就绪事件就阻塞CPU.多路指的是网络连接,复用指的是复用同一线程

select,poll执行流程

1. 用户线程调用select,将fd\_set从用户空间拷贝到内核空间
2. 内核空间遍历fd\_set,检查是否存在就绪的socket描述符,如果没有就进入休眠,直到有就绪的socket描述符
3. 内核返回就绪的文件描述符数量给用户线程
4. 用户拿到就绪的文件描述符数量后,队fd\_set进行再次遍历,找到就绪的文件描述符
5. 用户线程对就绪的文件描述符进行读写操作

epool

1. epoll\_create函数创建eventpoll对象(红黑树+双向链表)
2. 红黑树用来监听所有的文件描述符,并且通过epoll\_ctl将文件描述符添加后者删除到红黑树
3. 双向链表用来存储就绪的文件描述符列表,epoll\_wait调用时,检测链表中是否有数据,有则直接返回
4. 所有添加到eventpoll中的事件都与设备驱动程序简历回调关系

## 计算机网络七层协议,TCP/IP

|  |  |  |  |
| --- | --- | --- | --- |
| OSI七层网络模型 | TCP/IP模型 | 对应协议 | 作用 |
| 应用层 | 应用层 | HTTP,TFTP,FTP,NFS,SMTP | 为应用层序提供网络服务 |
| 表示层 | Telnet | 数据格式化,加解密 |
| 会话层 | DNS | 会话管理 |
| 传输层 | 传输层 | TCP,UDP | 建立端口到端口的连接 |
| 网络层 | 网络层 | IP,ICMP,ARP,RARP | IP寻址和路由选择,建立主机到主机的连接 |
| 数据链路层 | 数据链路层 | FDDI,Ethernet,Arpanet,PDN,SLIP,PPP | 将电信号组成一个数据包,称作帧,帧有两部分构成,标头和数据,标头表明数据发送者,接收者和数据类型 |
| 物理层 | IEEE802.1A,IEEE808.2到IEEE802.11 | 传输比特流0和1 |

## 进程线程和协程的区别

1. 进程是CPU资源分配的基本单位,线程CPU调度的基本单位(独立运行),CPU上真正运行的是线程

2. 进程拥有自己的资源空间(内存空间),一个进程包含若干个线程,多个线程共享同一进程内的资源(内存地址)

3. 协程,协程是一种用户态的轻量级线程,协程拥有自己的寄存器上下文和栈.协程调度切换时,将寄存器上下文和栈保存到其他地方,在切回来的时候,恢复先前保存的寄存器上下文和栈

## 线程调度方法

1. 先进先出(FIFO,First-In-First-Out)

优点:公平,无需任务切换,故而吞吐量大

缺点:平均响应时间长

适合任务队列中耗时差不多的场景

1. 最短耗时任务优先(SJF,Shortest Job First)

优先执行剩余耗时短的任务

优点:平均响应时间按短

缺点:耗时长的任务容易得不到调度,不公平,任务切换额外开销大

几乎没人用

1. 时间片轮转(Round Robin)

任务队列中每个任务分配一个时间片,第一个任务先执行,时间片耗尽后,将此任务放在队列末尾

优点:每个任务都能公平得到调度

缺点:多次切换任务开销大,多次切换导致cpu cache不命中,需要从内存重新加载

1. 最大最小公平

先平均分配,若多的话,再将剩余的给其他人再次平均分配,若不够的话,需要等待.

资源容量为6，现有3个使用方需要使用ABC，要求的资源数分别是1,2.1,4,多轮计算:

第一轮分配,3个使用方ABC参与分配，每个平均分得2个资源，A只需要使用1，还剩1， BC在本轮都不够用,被推迟.

第二轮分配,2个使用方BC参与分配,分别可获得2 + 1/2 = 2.5, B只需要要2.1，则还剩0.4,可给剩余的C分配，C还不够用,被推迟

1. Multi-level Feedback Queue(MFQ)

有多个Level,从上到下,优先级越来越低,分片时长越来越大,高优先级Level的任务可以抢占低优先级Level的任务.

新任务位于高优先级Level,当一个时间片用完之后,若任务结束,则正常退出,若任务没有结束，则下滑到低一等级的Level.若是因等待I/O而主动让出CPU处理，则停留在当前Level或者提高一个Level.

同一Level的任务采用时间片轮转算法

为避免系统中有太多的I/O任务而导致计算型任务迟迟得不到处理，MFQ算法会监控每个任务的处理耗时,确保其拥有公平的资源分配(按照最大最小公平算法).在每个Level的所有任务,若有任务还没有用完分配给他的资源,则相应提高他的优先级,反之则降低其优先级.

## 进程间通信的方法

每个进程各不同的地址空间,任何一个进程的全局变量在另一个进程中都看不到,进程之间要交换数据必须通过内核,在内核中开辟一块缓冲区,进程A把数据从用户空间拷到内核缓冲区,进程B再从内核缓冲区把数据读走,内核提供的这种机制称为进程间通信

1. 匿名管道(pipe):管道是一种半双工的通信方式,数据只能单向流动,而且只能在具有亲缘关系的进程间使用,进程的亲缘关系通常是指父子进程关系.
2. 高级管道(popen):将另一个程序当做一个新的进程在当前程序进程中启动,则它算是当前程序的子进程,这种方式我们成为高级管道方式

3. 有名管道(namedpipe):有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

4. 消息队列(messagequeue):消息队列是由消息的链表,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

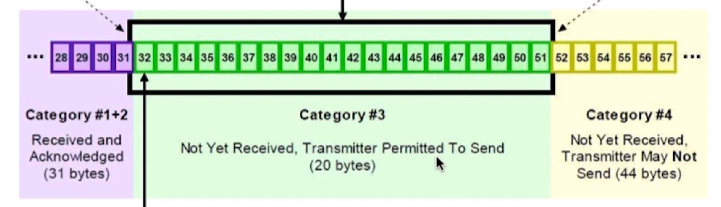
5. 信号量(semophore):信号量是一个计数器,可以用来控制多个进程对共享资源的访问。它常作为一种锁机制,防止某进程正在访问共享资源时,其他进程也访问该资源.因此,主要作为进程间以及同一进程内不同线程之间的同步手段.

6. 信号(sinal):信号是一种比较复杂的通信方式,用于通知接收进程某个事件已经发生

7. 共享内存(sharedmemory)：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IPC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,如信号量,配合使用,来实现进程间的同步和通信

8. 套接字(socket)：套接口也是一种进程间通信机制,与其他通信机制不同的是,它可用于不同机器间的进程通信

## TCP如何保证可靠性



TCP接收方缓存内有三种状态

1 已接收并发送了ack确认

2 未接收但可以接收状态（称为接收窗口）

3 未接收状态

TCP的传输可靠性来自确认重传,TCP滑动窗口可靠性也是来自确认重传,

发送方只有收到接受方对于本段发送窗口内字节的ack确认才会移动发送窗口的左边界,接收窗口只有在前面所有段都确认的情况下才会移动左边界,当前面还有字节未接收,收到后面的字节的情况下,窗口是不会移动的,并不会对后面字节进行确认,保证前面的数据会重传

## Go slice和array的区别,Slice扩容

slice是基于array实现的,slice并不是动态数组或者数组指针,内部是通过指针引用一个底层数组,并且将数据读写限制在一定区域内

slice扩容策略

首先判断,若申请容量大于旧容量的两倍,新容量即为申请容量

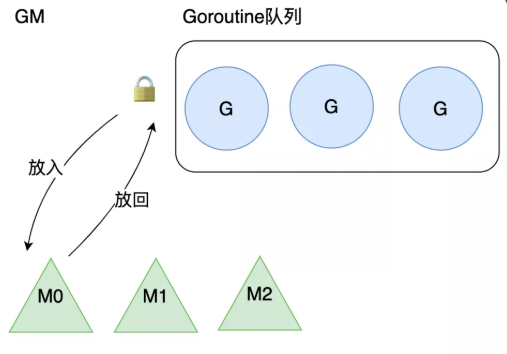
否则判断,若旧切片长度小于1024,则新容量为旧容量的两倍

否则判断,若旧切片长度大于1024,则新容量从旧容量循环增加原来的1/4,直到最终容量大于申请容量

如果最容容量计算值溢出,则最终容量就是新申请容量

## GMP模型

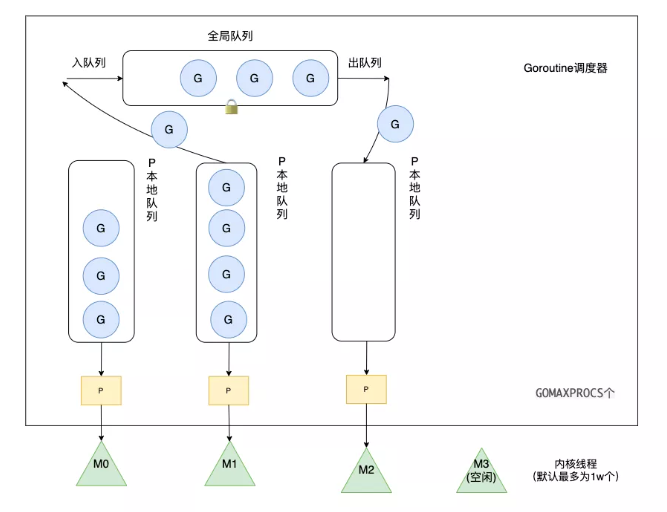
golang 1.1使用GM模型



M从加锁的全局队列获取G,缺点将G放入或者取出都需要获取锁,竞争激烈

M转移G没有把资源最大化利用,比如当M1在执行G1时,M1创建了G2,为了继续执行G1,需要把G2交给M2执行,因为G1和G2是相关的,而寄存器中会保存G1的信息,因此G2最好放在M1上执行,而不是其他的M

GMP



全局队列:当P中的G溢出时会放在全局队列中

P的本地队列:P内置的G队列,容量不超过256

当P1中运行的G1在运行过程中新建G2时,G2优先放在P1的本地队列,如果P1以满,则将P1本地队列中一半的G放入全局队列

P如果为空,则优先区全局队列中获取G,若全局队列也为空,则从其他P中偷取一半的G

P的数量由环境变量GOMAXPROCS或者是runtime.GOMAXPROCS()决定

M默认为10000个,可以通过runtime/debug中SetMaxThreads()改变

G在GMP模型中的流动过程

1. 调用 go func()创建一个goroutine
2. 新创建的G优先保存在P的本地队列中,如果P的本地队列已经满了就会保存在全局的队列中
3. M需要在P的本地队列弹出一个可执行的G,如果P的本地队列为空,则先会去全局队列中获取G,如果全局队列也为空则去其他P中偷取G放到自己的P中
4. G将相关参数传输给M,为M执行G做准备
5. 当M执行某一个G时候如果发生了系统调用产生导致M会阻塞,如果当前P队列中有一些G,runtime会将线程M和P分离,然后再获取空闲的线程或创建一个新的内核级的线程来服务于这个P,阻塞调用完成后G被销毁将值返回
6. 销毁G,将执行结果返回
7. 当M系统调用结束时候,这个M会尝试获取一个空闲的P执行,如果获取不到P,那么这个线程M变成休眠状态,加入到空闲线程中

GMP相较于GM有三个优化点

1. 每个P都有自己的本地队列,获取G优先从自己本地队列获取,不需要加锁,而从全局队列中获取会产生锁竞争
2. GMP实现了Work Stealing算法,如果P的本地队列为空,则会去全局队列或者其他P中窃取G,提高资源利用率
3. hand off机制,如果M在执行G中产生系统调用而阻塞,线程会释放P,把P交给其他M执行,提高资源利用率

## 分布式缓存框架singleflight

singleflight主要用于解决缓存击穿

## 缓存击穿,缓存雪崩,缓存穿透

击穿:热点key失效,造成访问该key的请求直接访问数据库,造成数据库瘫痪

雪崩:缓存层挂掉,所有请求直达数据库,造成数据库瘫痪

穿透:访问缓存中不存在的key,请求直接查询数据库,大量请求导致数据库瘫痪

解决方案:

1. 集群,哨兵,解决雪崩
2. 不同key,设置不同的过期时间,尽量平均分布,防止大量key同时失效
3. 热点key,永不过期(快过期时后台刷新缓存)
4. 缓存失效后,通过加锁控制访问数据库线程数量,如:对某个key只允许一个线程进行访问

## sync.WaitGroup

## cookie和session

1. Cookie

是由服务器发送给客户端的特殊信息,以文本的形式存放在客户端

客户端再次请求时，会把Cookie回发

服务器收到Cookie后，会解析Cookie生成与客户端相对应的内容

1. Session

服务器端机制，在服务器上保存信息

解析客户端请求并操作JESSIONID，按需保存状态信息

1. Session的实现方式

Cookie实现,URL回写实现

1. Session持久化

session的存储方式可以由开发者定义,如磁盘,redis,mysql中

gin session实现可以使用

github.com/gin-contrib/sessions

## 设计学生成绩数据库,查询语文成绩top3的人.url输入全过程,从此引出后端除了响应请求还有什么

select u.name as studentName from score s

left join user u on s.user\_id = u.id

left join course c on s.cource\_id = c.id

where c.name = '语文'

order by source desc

limit 3

浏览器输入url后执行过程

1. DNS解析，由近到远依次是浏览器缓存，系统缓存，路由器缓存，IPS服务器缓存，域名服务器缓存，顶级域名服务器缓存
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求返回HTTP报文
5. 浏览器解析渲染页面，结束连接

## 什么是分布式系统

在<<分布式系统概念与设计>>中的定义:

分布式系统是由一个 硬件或者软件组件分布在不同网络计算机上,彼此之间果果消息传递进行通信和协调的系统

分布式与集群:

分布式:指部署在多台服务器中的不同的服务,通过协同工作,对外提供服务

集群:指部署在多台服务器中相同的应用或者服务,通过负载均衡对外提供服务

## 负载均衡

常见的负载均衡算法:

1. 轮询法
2. 随机法
3. 源地址hash法

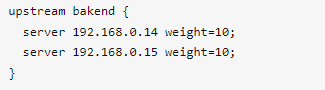
客户端ip通过hash函数得到一个数值,用该值对服务器数量取模运算,当后端服务器数量不变时,每次会映射到同一台服务器

1. 加权轮询法
2. 加权随机法
3. 最小连接数法

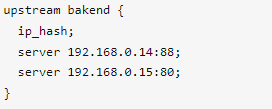
用积压请求最少的一台服务器来处理当前请求,可以尽可能的提高后端服务的利用率

Nginx五种负载均衡算法

1. 轮询(默认)
2. weight

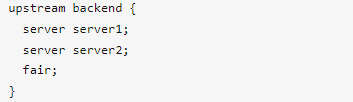


1. ip\_hash



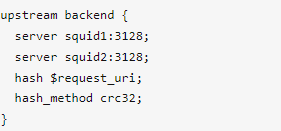
1. fair(第三方)

响应时间短优先分配



1. url\_hash(第三方)

按url的hash分配请求

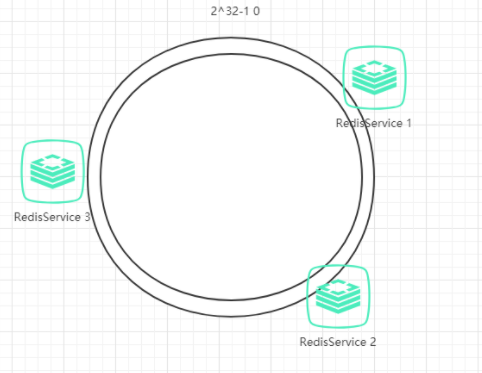


## 一致性Hash

使用一致哈希算法后,哈希表槽位数(大小)的改变平均只需要对 K/n 个关键字重新映射，其中K是关键字的数量,n是槽位数量.然而在传统的哈希表中，添加或删除一个槽位的几乎需要对所有关键字进行重新映射

一致性hash是将整个hash值的空间组织成一个虚拟的圆环,空间为[0,2^32-1]即无符号整形

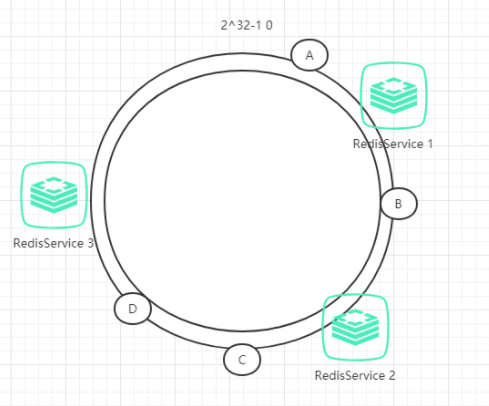
把服务器按照IP或主机名作为关键字进行哈希,这样就能确定其在哈希环的位置



然后,我们就可以使用哈希函数计算值为key的数据在哈希环的具体位置h，根据h确定在环中的具体位置，从此位置沿顺时针滚动，遇到的第一台服务器就是其应该定位到的服务器

例如我们有A,B,C,D四个数据对象,经过哈希计算后,在环空间上的位置如下：

数据A会被定为到Server 1上,数据B被定为到Server 2上,而C,D被定为到Server 3上



容错性

假如RedisService2宕机,那么B对应的数据会从RedisService2转移到RedisService3上,受干扰的只有RedisService2上保存的数据

扩展性

假如RedisService2和RedisService3之间新增了RedisSerivce4,RedisService4正好落在C,D对象之间,那么C对象会由RedisSerivce3对象转移到RedisService4上,受干扰的只有RedisService3上保存的数据

一致性Hash虚拟节点

为了解决多台服务器可能在Hash环中分布不均匀,造成数据存储分布不均匀,可以将每个节点计算多个hash值映射在Hash环中

## http和https区别,https的详细过程

http超文本传输协议,基于tcp,http存在的问题

1. 明文传输,容易被窃取
2. 没有验证身份,容易被冒充
3. 数据完整性未校验,容易被中间人篡改

https即http+ssl/tls

密文传输,需要证书认证默认端口443,加解密消耗资源较多

https流程

1. 客户端发起https请求,连接到服务器443端口
2. 服务器将证书中的公钥发送给客户端
3. 客户端收到公演,验证证书合法性,通过后,就会生成一个随机对称密钥,用证书公钥加密
4. 客户端将随机对称密钥发送给服务器
5. 服务器收到对称密钥用证书私钥解密
6. 这样服务端可客户端都拿拿到了对称密钥,之后就可以采用对称密钥加密传输数据

之所以证书是非对称加密方式,就是用于传输对称密钥,防止对称密钥被窃取

传输采用对称加密,是因为对称加密解密,比非对称效率高得多

## get post区别,还有什么其他方法,分别说说是做什么的

GET在浏览器回退时是无害的,而POST会再次提交请求

GET产生的URL地址可以被收藏,POST不可以

GET会被浏览器主动cache,POST不会,除非手动设置

GET只能进行url编码,POST支持多种编码方式

GET参数会被保留在浏览器历史记录里,POST中的参数不会被保留

GET请求在URL中传送的参数是有长度限制的,POST没有

对参数的数据类型,GET只接受ASCII字符,POST没有限制

GET比POST更不安全,因为参数直接暴露在URL上

GET参数通过URL传递,POST放在Request body中

对于GET方式的请求,浏览器会把http header和data一并发送出去,服务器响应200(返回数据)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)

也并非所有POST都是分开发送,Firefox只会发送一次

其他请求

HEAD:与GET本质一样,HEAD不含有数据,仅含有HTTP头信息,可以用于判断资源是否存在

PUT:更新服务器资源,PUT通常指定了资源存放的位置

DELETE:删除某一个资源

OPTIONS:用于返回当前URL所支持的方法,在HTTP头中返回一个名为 Allow的头,值是所支持的方法,如GET,POST

## web安全问题

Dos denial of service 拒绝服务攻击

DDos distributed denial of service 分布式拒绝服务攻击

DRDos distributed reflection denial of service 分布式反射拒绝服务攻击

Xss cross-site scripting跨站脚本攻击

CSRF cross-site request forgery跨站请求伪造

## 设计一个短链接服务,如何抗住大qps,抗大流量.url哈希函数怎么设计(怎么存,怎么统计qps)

QPS: 每秒查询

TPS: 每秒事务

RT: 响应时间

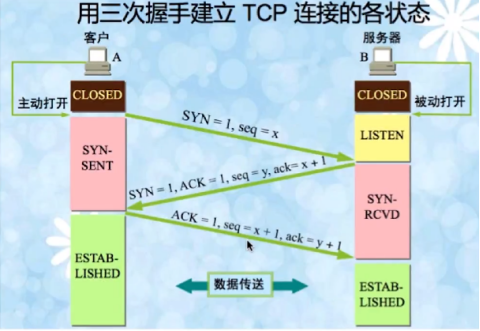
# 腾讯

## TCP和UDP的特性,TCP三次握手,四次挥手

**TCP与UDP的区别**

1. TCP面向连接,UDP面向非连接
2. TCP可靠有序,UDP非可靠
3. TCP传输效率没有UDP高
4. TCP比UDP重量级,TCP头20字节,UDP头8字节

**TCP三次握手**



第一次握手:建立连接时,客户端发送SYN包到服务端,之后客户端进入SYN\_SEND状态,等待服务端确认

第二次握手:服务端收到客户端发送的SYN包,确认客户端的SYN包,同时自己也向客户端发送SYN包,即服务端发送SYN+ACK包,发送之后服务端进入SYN\_RECV状态

第三次握手:客户端收到服务器的SYN+ACK包之后,向服务端发送确认包,此包发送完毕后,客户端服务端都进入ESTAB\_LISHED状态,此时客户端服务端可以进行传输数据,完成三次握手

**三次握手存在的隐患,SYN超时**

Server收到Client的SYN包之后,回复SYN-ACK之后确一直收不到Client的ACK确认,即没有完成三次握手

发生上述情况后,Server会不断重试,Linux默认重试5次等待63秒后断开TCP连接

这种情况可能造成SYN-Flood攻击，SYN-Flood攻击是一种典型的Dos(拒绝服务攻击),恶意程序持续向Server发送SYN包却不响应ACK,直到把SYN队列耗尽

解决方案:SYN队列被耗尽之后,通过tcp-syncookies参数发送SYN Cookie,若为正常连接的Client会发送SYN Cookie,建立连接

**TCP四次挥手**



第一次挥手:Client向Server发送一个FIN包,关闭Client到Server的数据传输,发送后Client进入FIN\_WAIT\_1状态

第二次挥手:Server收到Client的FIN包,发送一个ACK确认给Clinet,Server进入CLOSE\_WAIT状态

第三次挥手:Server向Client发送FIN+ACK包,关闭Server到Client的数据传输,发送后Server进入LAST\_ACK状态

第四次挥手:Client收到FIN包后,Client进入TIME\_WAIT状态,接着发送一个ACK给Server,Server收到ACK之后直接进入CLOSED状态,Client需要等待2MSL之后进入CLOSED状态

**client为什么需要TIME\_WAIT状态**

1. 确保最后一个确认报文能够到达(如果server没收到client发送来的确认报文,那么就会重新发送FIN+ACK请求释放连接,client等待2MSL就可以在这个时间处理该请求,使server正常close)
2. 避免新旧链接混淆(等待2MSL可以让本连接持续时间内所产生的所有报文都从网络中消失,使得下一个新的连接请求不会出现旧的连接请求报文)

## http2.0了解吗?和http1.1的差距

http1.0

1. 默认使用短连接,每次请求都需要建立TCP连接,它可以通过设置Connection:keep-alive强制开启
2. 请求是串行的
3. 不支持断点续传

http1.1

1. 使用长连接,即TCP默认不关闭,一个TCP连接可以被多个HTTP复用
2. 分块传输,即服务端每产生一块数据,就发送一块,用流模式取代缓存模式
3. 管道机制,即在同一个TCP连接,客户端可以同时发送多个请求
4. 新增range字段,用来指定数据字节的位置,支持了断点续传

http2.0

1. 二进制协议,1.1版本头信息是文本(ASCII编码),数据体可是是文本或者二进制,2.0中头信息和数据体都是二进制,如果发送的多个请求的头部是相同的,http2.0会消除相同的部分(通过在客户端和服务端都维护一张索引表实现)
2. 多路复用,在同一个连接中,服务端和浏览器都可以同时发送多个请求或者回应,无需按照顺序,同一个连接里面连续的数据包可能属于不同请求的回应(对数据包做了标记,标记其属于哪一个请求,规定客户端的数据流编号为奇数,服务端数据流编号为偶数)
3. 报头压缩,http协议无状态,每次请求都必须附上所有信息,http2.0引入了头信息压缩机制,使用gzip或者compress压缩后发送
4. 允许服务端未经请求,主动向客户端发送资源

## 反爬了解吗?反爬的几项技术在项目里面怎么实现的?

## go struct能不能比较?

不同结构体无法比较

相同结构体不包含无法比较的类型（slice,map）则可以比较

## go defer (for defer)

defer 传入的函数不是在退出代码块的作用域时执行的，它只会在当前函数和方法返回之前被调用

假设我们想要计算 main 函数运行的时间

|  |
| --- |
| func main() {  startedAt := time.Now()  defer fmt.Println(time.Since(startedAt))  time.Sleep(time.Second)  }  $ go run main.go  0s |

调用 defer关键字会立刻拷贝函数中引用的外部参数,所以 time.Since(startedAt) 的结果不是在 main 函数退出之前计算的,而是在 defer 关键字调用时计算的,最终导致上述代码输出 0s

|  |
| --- |
| func main() {  startedAt := time.Now()  defer func() { fmt.Println(time.Since(startedAt)) }()  time.Sleep(time.Second)  }  $ go run main.go  1s |

调用 defer 关键字时也使用值传递,但是因为拷贝的是函数指针,所以 time.Since(startedAt) 会在 main 函数返回前调用并打印出符合预期的结果

## select可以用于什么

select和switch一样有多个case,但是每一个case都只能用于chan的收发工作

select遇到多个chan同时响应时,会随机执行一种情况

select 若不存在default选项,则

select会阻塞当前Goroutine并等待多个chan中的一个达到可以收发的状态

select 若存在defaulte选项,则为非阻塞收发

存在以下两种情况：

当存在可以收发的chan时,直接处理该chan对应的case

当不存在可以收发的chan时,执行 default 中的语句

## context包的用途

context主要用于设置截止日期,同步信号,传递请求相关值的结构体

context.Context主要包含4个方法

type Context interface {

//返回被取消的时间

Deadline() (deadline time.Time, ok bool)

Done() <-chan struct{}

//如果context.Context被取消返回Canceled错误

//如果context.Context超时返回DeadlineExceeded错误

Err() error

//从context.Context中获取键值,对于同一个上下文

//多次调用同一key,返回的结果总是一样

Value(key interface{}) interface{}

}

## 主协程如何等其余协程完再操作

sync.WaitGroup

|  |
| --- |
| func main() {  wg := sync.WaitGroup{}  wg.Add(10)  for i := 0; i < 10; i++ {  go do(&wg)  }  wg.Wait()  fmt.Println("main exec")  }  func do(wg \*sync.WaitGroup) {  defer wg.Done()  fmt.Println("do exec")  } |

## map如何顺序读取

对key排序

# 算法

## LRU

## 生产者消费者模型

## 反转链表

## 三数之和

## TOPK

## 基本算法-快排

## 基本算法-堆排