**天命流觞**

**后端知识点整理**

**内容来自网络，不保证正确**

**内容来自网络，不保证正确**

**内容来自网络，不保证正确**

**仅限交流学习，勿做他途**

**仅限交流学习，勿做他途**

**仅限交流学习，勿做他途**

**2021-02-21**

目录

[mysql 常见知识点 12](#_Toc1581)

[一、为什么用自增列作为主键 12](#_Toc19941)

[二、为什么使用数据索引能提高效率 13](#_Toc22019)

[MySQL索引类型 13](#_Toc16968)

[三、B+树索引和哈希索引的区别 14](#_Toc18510)

[五、哈希索引不适用的场景： 15](#_Toc2989)

[四、哈希索引的优势： 16](#_Toc24426)

[六、B树和B+树的区别 16](#_Toc28691)

[七、为什么说B+比B树更适合实际应用中操作系统的文件索引和数据库索引？ 17](#_Toc10793)

[八、MySQL联合索引 18](#_Toc18600)

[九、什么情况下应不建或少建索引 18](#_Toc5143)

[十、什么是表分区？ 19](#_Toc21879)

[十一、表分区与分表的区别 19](#_Toc7422)

[十二、表分区有什么好处？ 19](#_Toc32615)

[十三、分区表的限制因素 19](#_Toc22883)

[十四、如何判断当前MySQL是否支持分区？ 20](#_Toc822)

[十五、MySQL支持的分区类型有哪些？ 20](#_Toc27669)

[事务的 四个特征（ACID） 21](#_Toc5470)

[嵌套事务 21](#_Toc26884)

[十六、Mysql的四种隔离级别 22](#_Toc19967)

[1、Read Uncommitted（读取未提交内容） 22](#_Toc21496)

[2、Read Committed（读取提交内容） 22](#_Toc25925)

[3、Repeatable Read（可重读） 23](#_Toc16866)

[4、Serializable（可串行化） 23](#_Toc19436)

[脏读(Drity Read) 23](#_Toc6623)

[不可重复读(Non-repeatable read) 23](#_Toc18623)

[幻读(Phantom Read) 24](#_Toc6950)

[四种隔离级别对照关系 24](#_Toc19279)

[十七、关于MVCC 24](#_Toc14188)

[十八、在MVCC并发控制中，读操作可以分成两类： 25](#_Toc25914)

[十九、行级锁定的优点： 25](#_Toc3475)

[二十、行级锁定的缺点： 25](#_Toc9902)

[二十一、MySQL优化 25](#_Toc28312)

[二十二、key和index的区别 27](#_Toc16406)

[二十三、Mysql 中 MyISAM 和 InnoDB 的区别有哪些？ 27](#_Toc26393)

[二十四、数据库表创建注意事项 28](#_Toc6668)

[1、字段名及字段配制合理性 28](#_Toc23022)

[2、系统特殊字段处理及建成后建议 29](#_Toc24073)

[3、表结构合理性配置 29](#_Toc23719)

[4、其它建议 29](#_Toc32106)

[二十五、数据库范式 30](#_Toc19703)

[范式简介 30](#_Toc29122)

[1、第一范式（1NF）： 30](#_Toc3650)

[2、第二范式（2NF） 30](#_Toc2190)

[3、第三范式（3NF） 31](#_Toc6389)

[4、巴斯-科德范式（BCNF） 31](#_Toc24012)

[目的原则 32](#_Toc4436)

[反范式 32](#_Toc1027)

[BC范式 32](#_Toc2515)

[第四范式 33](#_Toc3510)

[Redis常见知识点 35](#_Toc30022)

[Redis支持的数据类型？ 35](#_Toc2913)

[String字符串： 35](#_Toc12125)

[Hash（哈希） 36](#_Toc27249)

[List（列表） 37](#_Toc30288)

[Set（集合） 38](#_Toc17615)

[zset(sorted set：有序集合) 39](#_Toc18093)

[什么是Redis持久化？Redis有哪几种持久化方式？优缺点是什么？ 40](#_Toc28153)

[RDB： 40](#_Toc21136)

[AOF: 41](#_Toc12741)

[Redis 有哪些架构模式？讲讲各自的特点 42](#_Toc22501)

[单机版 42](#_Toc6711)

[主从复制 42](#_Toc30042)

[哨兵 43](#_Toc32023)

[集群（proxy 型）： 44](#_Toc23588)

[集群（直连型）： 44](#_Toc22311)

[Redis常用命令？ 45](#_Toc27856)

[使用过Redis分布式锁么，它是怎么实现的？ 48](#_Toc27183)

[使用过Redis做异步队列么，你是怎么用的？有什么缺点？ 49](#_Toc32690)

[用法 49](#_Toc31798)

[缺点： 49](#_Toc23986)

[什么是缓存穿透？如何避免？什么是缓存雪崩？何如避免？ 49](#_Toc18841)

[缓存穿透 49](#_Toc21630)

[如何避免？ 49](#_Toc2286)

[缓存雪崩 50](#_Toc26999)

[如何避免？ 50](#_Toc5830)

[布隆过滤器 50](#_Toc2251)

[基本概念 50](#_Toc20692)

[MongoDB知识点 51](#_Toc15004)

[1.什么是MongoDB 51](#_Toc8969)

[2.MongoDB的优势有哪些 52](#_Toc30246)

[3 什么是数据库 53](#_Toc7825)

[4.什么是集合(表) 53](#_Toc11068)

[5 什么是文档(记录) 53](#_Toc29158)

[6 MongoDB和关系型数据库术语对比图 53](#_Toc7323)

[7.什么是非关系型数据库 53](#_Toc4689)

[8 为什么用MongoDB？ 54](#_Toc14817)

[9 在哪些场景使用MongoDB 54](#_Toc10892)

[10 MongoDB中的命名空间是什么意思? 54](#_Toc24104)

[11 monogodb 中的分片什么意思 54](#_Toc16233)

[12 为什么要在MongoDB中使用分析器 54](#_Toc16538)

[13 .MongoDB支持主键外键关系吗 54](#_Toc17056)

[14 MongoDB支持哪些数据类型 55](#_Toc3698)

[15 为什么要在MongoDB中用"Code"数据类型 55](#_Toc19772)

[16 为什么要在MongoDB中用"Regular Expression"数据类型 55](#_Toc8781)

[17 为什么在MongoDB中使用"Object ID"数据类型 55](#_Toc17980)

[18 "ObjectID"有哪些部分组成 55](#_Toc10935)

[19 在MongoDb中什么是索引 55](#_Toc8031)

[20 如何添加索引 55](#_Toc25167)

[21.如何查询集合中的文档 56](#_Toc24888)

[22用什么方法可以格式化输出结果 56](#_Toc19972)

[23 如何使用"AND"或"OR"条件循环查询集合中的文档 56](#_Toc18683)

[24 更新数据 56](#_Toc23090)

[25 如何删除文档 56](#_Toc3496)

[26 在MongoDB中如何排序 56](#_Toc13736)

[27 什么是聚合 56](#_Toc20339)

[28 在MongoDB中什么是副本集（避免单点故障） 56](#_Toc478)

[29 什么是NoSQL数据库？NoSQL和RDBMS有什么区别？在哪些情况下使用和不使用NoSQL数据库？ 56](#_Toc2981)

[30 MongoDB支持存储过程吗？如果支持的话，怎么用？ 57](#_Toc27112)

[31如何理解MongoDB中的GridFS机制，MongoDB为何使用GridFS来存储文件？ 57](#_Toc71)

[32 为什么MongoDB的数据文件很大？ 57](#_Toc32227)

[33 当更新一个正在被迁移的块（Chunk）上的文档时会发生什么？ 57](#_Toc1617)

[34 MongoDB在A:{B,C}上建立索引，查询A:{B,C}和A:{C,B}都会使用索引吗？ 57](#_Toc24308)

[35 mongodb成为最好nosql数据库的原因是什么? 57](#_Toc24121)

[36 如果用户移除对象的属性,该属性是否从存储层中删除? 57](#_Toc26638)

[37 允许空值null吗? 57](#_Toc2581)

[38 更新操作立刻fsync到磁盘? 58](#_Toc8730)

~~[39 如何执行事务/加锁?](#_Toc27054)~~ [58](#_Toc27054)

[40 启用备份故障恢复需要多久? 58](#_Toc1595)

[41 什么是master或primary? 58](#_Toc12333)

[42 我应该启动一个集群分片(sharded)还是一个非集群分片的 mongodb 环境? 58](#_Toc26426)

[43 分片(sharding)和复制(replication)是怎样工作的? 58](#_Toc18518)

[44数据在什么时候才会扩展到多个分片(shard)里? 59](#_Toc25449)

[45 如果在一个分片(shard)停止或者很慢的时候,我发起一个查询会怎样? 59](#_Toc9063)

[46 可以把movechunk目录里的旧文件删除吗? 59](#_Toc2682)

[47 如果块移动操作(movechunk)失败了,我需要手动清除部分转移的文档吗? 59](#_Toc4111)

[48 mongodb是否支持事务 59](#_Toc32532)

[ElasticSearch常见经典知识点 60](#_Toc25127)

[1.为什么要使用Elasticsearch? 60](#_Toc23098)

[2.Elasticsearch是如何实现Master选举的？ 60](#_Toc14599)

[3.Elasticsearch中的节点（比如共20个），其中的10个选了一个master，另外10个选了另一个master，怎么办？ 60](#_Toc1727)

[4.详细描述一下Elasticsearch索引文档的过程。 61](#_Toc25007)

[5.详细描述一下Elasticsearch更新和删除文档的过程 61](#_Toc15691)

[6.详细描述一下Elasticsearch搜索的过程 61](#_Toc31521)

[8.在并发情况下，Elasticsearch如果保证读写一致？ 62](#_Toc19920)

[9.ElasticSearch中的集群、节点、索引、文档、类型是什么？ 62](#_Toc30365)

[10.ElasticSearch中的分片是什么? 63](#_Toc2395)

[11.什么是ElasticSearch？ 63](#_Toc6811)

[12.Elasticsearch中的倒排索引是什么？ 63](#_Toc30624)

[13.ElasticSearch是否有架构？ 64](#_Toc9909)

[14..ElasticSearch中的副本是什么？ 64](#_Toc7604)

[15.ElasticSearch中的分析器是什么？ 64](#_Toc26605)

[16.什么是ElasticSearch中的编译器？ 64](#_Toc2003)

[17.启用属性，索引和存储的用途是什么？ 65](#_Toc18401)

[50道Python知识点集锦（附答案） 65](#_Toc5671)

[Q1、Python中的列表和元组有什么区别？ 65](#_Toc30452)

[list和tuple的一些区别 66](#_Toc19685)

[Q2、Python的主要功能是什么？ 68](#_Toc4214)

[Q3、Python是通用编程语言吗？ 68](#_Toc10818)

[Q4、Python是如何解释语言的？ 68](#_Toc20815)

[Q5、什么是pep？ 68](#_Toc22790)

[Q6、如何在Python中管理内存？ 69](#_Toc31929)

[Q7、Python中的命名空间是什么？ 69](#_Toc3274)

[Q8、什么是PYTHONPATH？ 69](#_Toc10250)

[Q9、什么是python模块？Python中有哪些常用的内置模块？ 69](#_Toc32040)

[Q10、Python中的局部变量和全局变量是什么？ 69](#_Toc11682)

[Q11、python是否区分大小写？ 70](#_Toc24421)

[Q12、什么是Python中的类型转换？ 70](#_Toc18553)

[Q13、如何在Windows上安装Python并设置路径变量？ 70](#_Toc31318)

[Q14、python中是否需要缩进？ 71](#_Toc9440)

[Q15、Python数组和列表有什么区别？ 71](#_Toc3734)

[Q16、Python中的函数是什么？ 71](#_Toc22631)

[Q17、什么是\_\_init\_\_? 71](#_Toc23879)

[Q18、什么是lambda函数？ 71](#_Toc540)

[Q19、Python中的self是什么？ 71](#_Toc10734)

[Q20、区分break，continue和pass？ 72](#_Toc7537)

[Q21、[:: - 1}表示什么？ 72](#_Toc14387)

[Q22、如何在Python中随机化列表中的元素？ 72](#_Toc9602)

[Q23、什么是python迭代器？ 72](#_Toc3201)

[Q24、如何在Python中生成随机数？ 72](#_Toc21420)

[Q25、range＆xrange有什么区别？ 73](#_Toc12005)

[Python3 迭代器与生成器 73](#_Toc23422)

[迭代器 73](#_Toc27032)

[生成器 76](#_Toc22414)

[Q26、如何在python中写注释？ 77](#_Toc2322)

[Q27、什么是pickling和unpickling？ 77](#_Toc30147)

[Q28、python中的生成器是什么？ 77](#_Toc4517)

[Q29、你如何把字符串的第一个字母大写？ 77](#_Toc9995)

[Q30、如何将字符串转换为全小写？ 78](#_Toc16105)

[Q31、如何在python中注释多行？ 78](#_Toc30098)

[Q32、什么是Python中的文档Docstrings？ 78](#_Toc892)

[Q33、operators中的is、not和in各有什么功能？ 78](#_Toc32254)

[Q34、Python中help()和dir()函数的用法是什么？ 78](#_Toc13291)

[Q35、当Python退出时，为什么不清除所有分配的内存？ 78](#_Toc11065)

[Q36、Python中的字典是什么？ 79](#_Toc7615)

[Q37、如何在python中使用三元运算符？ 79](#_Toc4555)

[Q38、为什么使用\* args，\*\* kwargs？ 79](#_Toc5117)

[Q39、len()函数有什么作用？ 79](#_Toc28378)

[Q40、在Python中split()，sub()，subn()功能。 79](#_Toc1178)

[Q41、什么是负指数，功能是什么？ 80](#_Toc12354)

[Q42、什么是Python包？ 80](#_Toc1723)

[Q43、如何在Python中删除文件？ 80](#_Toc22125)

[Q44、什么是python的内置类型？ 80](#_Toc4017)

[Q45、NumPy中有哪些操作Python列表的函数？ 80](#_Toc17863)

[Q46、如何将值添加到python数组？ 81](#_Toc11536)

[Q47、如何删除python数组的值？ 81](#_Toc3336)

[Q48、Python有OOps概念吗？ 81](#_Toc21591)

[Q49、深拷贝和浅拷贝有什么区别？ 81](#_Toc3760)

[Q50、如何在Python中实现多线程？ 82](#_Toc7427)

[golang 知识点(从基础到高级) 82](#_Toc12126)

[1. Golang中除了加Mutex锁以外还有哪些方式安全读写共享变量？ 82](#_Toc8193)

[2. 无缓冲 Chan 的发送和接收是否同步? 82](#_Toc30703)

[3. go语言的并发机制以及它所使用的CSP并发模型． 82](#_Toc20436)

[4. Golang 中常用的并发模型？ 84](#_Toc20871)

[通过channel通知实现并发控制 85](#_Toc30458)

[通过sync包中的WaitGroup实现并发控制 85](#_Toc29425)

[Context上下文，实现并发控制 87](#_Toc13896)

[5. JSON 标准库对 nil slice 和 空 slice 的处理是一致的吗？ 88](#_Toc4607)

[6. 协程，线程，进程的区别。 89](#_Toc27480)

[进程 89](#_Toc16582)

[线程 89](#_Toc26433)

[协程 89](#_Toc24048)

[7. 互斥锁，读写锁，死锁问题是怎么解决。 90](#_Toc24622)

[死锁产生的四个必要条件: 90](#_Toc30849)

[8. Golang的内存模型，为什么小对象多了会造成gc压力。 92](#_Toc7313)

[9. Data Race问题怎么解决？能不能不加锁解决这个问题？ 92](#_Toc19124)

[10. 什么是channel，为什么它可以做到线程安全？ 92](#_Toc3331)

[11. Epoll原理. 93](#_Toc19682)

[12. Golang GC 时会发生什么? 95](#_Toc29200)

[常用的垃圾回收的方法: 96](#_Toc10389)

[引用计数（reference counting） 96](#_Toc21891)

[标记-清除（mark and sweep） 97](#_Toc21810)

[分代搜集（generation） 97](#_Toc6893)

[Golang GC 时会发生什么? 97](#_Toc13016)

[13. Golang 中 Goroutine 如何调度? 98](#_Toc18805)

[14. 并发编程概念是什么？ 100](#_Toc23919)

[15. 负载均衡原理是什么? 101](#_Toc321)

[负载均衡算法 102](#_Toc251)

[16. LVS相关了解. 103](#_Toc24813)

[17. 微服务架构是什么样子的? 104](#_Toc425)

[18. 分布式锁实现原理，用过吗？ 105](#_Toc1002)

[基于数据库实现分布式锁。 105](#_Toc7482)

[基于Redis的实现方式 107](#_Toc18627)

[基于ZooKeeper的实现方式 108](#_Toc6316)

[19. Etcd怎么实现分布式锁? 108](#_Toc18070)

[20. Redis的数据结构有哪些，以及实现场景? 115](#_Toc25688)

[21. Mysql高可用方案有哪些? 117](#_Toc21315)

[22. Go语言的栈空间管理是怎么样的? 118](#_Toc12767)

[23. Goroutine和Channel的作用分别是什么? 121](#_Toc18462)

[24. 怎么查看Goroutine的数量? 123](#_Toc4763)

[25. 说下Go中的锁有哪些?三种锁，读写锁，互斥锁，还有map的安全的锁? 123](#_Toc24870)

[26. 读写锁或者互斥锁读的时候能写吗? 136](#_Toc10860)

[27. 怎么限制Goroutine的数量. 136](#_Toc13078)

[28. Channel是同步的还是异步的. 141](#_Toc8729)

[29. 说一下异步和非阻塞的区别? 141](#_Toc22210)

[30. Log包线程安全吗？ 142](#_Toc7585)

[31. Goroutine和线程的区别? 142](#_Toc7330)

[32. 滑动窗口的概念以及应用? 143](#_Toc24114)

[33. 怎么做弹性扩缩容，原理是什么? 144](#_Toc29053)

[34. 让你设计一个web框架，你要怎么设计，说一下步骤. 144](#_Toc13463)

[35. 说一下中间件原理. 144](#_Toc9610)

[36. 怎么设计orm，让你写,你会怎么写? 145](#_Toc20574)

[37. 用过原生的http包吗？ 145](#_Toc4487)

[38. 一个非常大的数组，让其中两个数想加等于1000怎么算? 145](#_Toc6417)

[39. 各个系统出问题怎么监控报警. 145](#_Toc12689)

[40. 常用测试工具，压测工具，方法? 145](#_Toc27177)

[41. 复杂的单元测试怎么测试，比如有外部接口mysql接口的情况 145](#_Toc19401)

[42. redis集群，哨兵，持久化，事务 145](#_Toc4006)

[43. mysql和redis区别是什么？ 145](#_Toc10499)

[44. 高可用软件是什么？ 145](#_Toc4219)

[45. 怎么搞一个并发服务程序？ 145](#_Toc22552)

[46. 讲解一下你做过的项目，然后找问题问实现细节。 145](#_Toc32035)

[47. mysql事务说下。 145](#_Toc30862)

[48. 怎么做一个自动化配置平台系统？ 145](#_Toc27677)

[49. grpc遵循什么协议？ 145](#_Toc12207)

[50. grpc内部原理是什么？ 145](#_Toc23339)

[51. http2的特点是什么，与http1.1的对比。 145](#_Toc17234)

[52.Go的调度原理. 146](#_Toc9081)

[53.Go语言实战笔记（十二）| Go goroutine 146](#_Toc28636)

[54.Golang调度器源码分析 146](#_Toc24608)

[55.Goroutine调度过程 146](#_Toc31547)

[56.go struct能不能比较 146](#_Toc24993)

[57.Go 关键字 defer 的一些坑 147](#_Toc21753)

[58.select可以用于什么? 147](#_Toc24135)

[59.Go的select主要是处理多个channel的操作. 147](#_Toc21391)

[60.Go语言并发模型：使用 select 147](#_Toc20115)

[61.context包的用途是什么? 147](#_Toc24990)

[62.client如何实现长连接? 147](#_Toc7718)

[63.HTTP Keep-Alive是什么？如何工作？ 147](#_Toc16835)

[64.基本排序，哪些是稳定的 148](#_Toc23740)

[65.Http介绍 149](#_Toc1557)

[67.Http keep-alive 149](#_Toc14385)

[68.Http能不能一次连接多次请求，不等后端返回 149](#_Toc147)

[69.TCP 和 UDP 有什么区别,适用场景 149](#_Toc4026)

[70.UDP 的特点及使用场景: 150](#_Toc23566)

[71.TCP 的特点及使用场景: 150](#_Toc12710)

[72.time-wait的作用 150](#_Toc18770)

[73.数据库如何建索引 150](#_Toc8294)

[74.孤儿进程，僵尸进程 150](#_Toc19092)

[75.死锁条件，如何避免 150](#_Toc24758)

[76.通常一般会用到哪些数据结构? 151](#_Toc14206)

[77.链表和数组相比, 有什么优缺点? 151](#_Toc8722)

[78.如何判断两个无环单链表有没有交叉点? 151](#_Toc9792)

[79.如何判断一个单链表有没有环, 并找出入环点? 151](#_Toc23525)

[80.描述一下 TCP 四次挥手的过程中 151](#_Toc2711)

[81.TCP 有哪些状态? 151](#_Toc3343)

[82.TCP 的 LISTEN 状态是什么? 151](#_Toc8621)

[83.TCP 的 CLOSE\_WAIT 状态是什么? 151](#_Toc30899)

[84.建立一个 socket 连接要经过哪些步骤? 151](#_Toc14897)

[85.常见的 HTTP 状态码有哪些? 151](#_Toc21107)

[86.301和302有什么区别? 151](#_Toc8747)

[87.504和500有什么区别? 151](#_Toc7548)

[88.HTTPS 和 HTTP 有什么区别? 151](#_Toc4169)

[89.算法题: 手写一个快速排序 151](#_Toc23832)

[90.Golang 里的逃逸分析是什么？怎么避免内存逃逸？ 153](#_Toc19640)

[91.配置中心如何保证一致性？ 153](#_Toc27293)

[92.Golang 的GC触发时机是什么? 153](#_Toc15829)

[93.Redis 里数据结构的实现熟悉吗? 153](#_Toc9192)

[94.Etcd的Raft一致性算法原理? 153](#_Toc24364)

[95.微服务概念. 153](#_Toc32472)

[96.SLB原理. 153](#_Toc7582)

[97.分布式一直性原则. 153](#_Toc12632)

[98.如何保证服务宕机造成的分布式服务节点处理问题? 153](#_Toc10675)

[99. 服务发现怎么实现的. 154](#_Toc23290)

[100. 第1题什么是golang? 154](#_Toc6482)

[101. 第2题为什么要用golang? 154](#_Toc17459)

[102. 第3题是谁开发了go语言？ 154](#_Toc15488)

[103. 第4题，go编程中的packages是什么？ 154](#_Toc10426)

[104. 第5题, Go语言是否支持generic编程? 154](#_Toc25813)

[105. 第6题, go语言是大小写敏感的编程语言吗？ 155](#_Toc28057)

[106. 第7题，go语言里面字符串的语法是怎样的？ 155](#_Toc7616)

[Golang 单引号、双引号、反引号 156](#_Toc26853)

[107. 第8题, Go语言的work space是指什么? 156](#_Toc985)

[108. 第9题，go语言中布尔类型的缺省值是什么？ 156](#_Toc20551)

[109. 第10题, Gopath环境变量是什么? 156](#_Toc7117)

[110. 第11题，go语言编程的好处是什么？ 157](#_Toc19886)

[111. 第12题，go语言里比较常用的功能模块。 157](#_Toc25881)

[112. 第13题，goroutine是什么？ 157](#_Toc4005)

[113. 第14题，如何写多行的字符串？ 158](#_Toc16013)

[114. 第15题, go语言里的break是干什么用的？ 158](#_Toc25435)

[115. 第16题，continue的作用是什么？ 158](#_Toc26828)

[116. 第17题，goto是干什么的？ 158](#_Toc9114)

[117. 第18题，for loop如何使用？ 158](#_Toc23995)

[118. 第19题在go语言中如何创建一个函数？ 158](#_Toc19437)

[119. 第20题, 解释一下go语言中的静态类型声明。 159](#_Toc22281)

[120. 第21题, 解释一下go语言中的动态类型声明。 159](#_Toc29259)

[121. 第22题, 如何打印出一个变量的类型? 159](#_Toc29111)

[122. 第23题, 什么是指针? 159](#_Toc11023)

[123. 第24题, 指针是如何表示的? 159](#_Toc5791)

[124. 第25题， := 短声明是不是只能用在一个函数里面? 159](#_Toc4562)

[125. 第26题， 如何不打印来格式化一个字符串? 160](#_Toc6271)

[126. 第27题， go语言的语义是什么类型? 160](#_Toc17883)

[127. 第28题, go语言是否支持继承? 160](#_Toc15121)

[128. 第29题，go语言是否支持操作符的重载? 160](#_Toc24383)

[129. 第30题，go语言是否支持方法重载? 160](#_Toc13368)

[130. 第31题，go语言是否支持指针运算? 160](#_Toc18928)

[131. 第32题，如下代码的输出是什么？ 161](#_Toc32592)

[132. 第33题，go的接口是什么？ 161](#_Toc7750)

[133. 第34题， Go语言里面的类型断言是怎么回事? 162](#_Toc23573)

[134. 第35题， Go语言里面的方法是怎么回事? 162](#_Toc2757)

[135. 第36题, go语言中局部变量的缺省值是什么? 162](#_Toc24009)

[136. 第37题， 全局变量的缺省值是什么? 162](#_Toc16692)

[137. 第38题， 指针的缺省值是什么? 162](#_Toc21628)

[138. 第39题, 如何在运行时检查一个变量的类型? 162](#_Toc10557)

[139. 第40题, 在使用go routine时是否推荐使用全局变量? 163](#_Toc27042)

[140. 第41题, 模块化编程是怎么回事? 163](#_Toc27943)

[算法 163](#_Toc21092)

[字节 163](#_Toc22907)

[网络 163](#_Toc19777)

[熟悉Internet常用协议 163](#_Toc3711)

[HTTP 163](#_Toc3586)

[TCP/IP 163](#_Toc15095)

[RESTful规范 163](#_Toc4623)

[缓存优化、RPC优化 163](#_Toc4496)

[面试注意 163](#_Toc26788)

[Redis 专题 165](#_Toc30417)

[redis八大应用场景 165](#_Toc5499)

[Redis重启时加载AOF与RDB的顺序 167](#_Toc5017)

[Redis慢查询分析 169](#_Toc10717)

[zset跳表的数据结构（必考） 172](#_Toc6246)

[Redis的数据过期策略（必考） 172](#_Toc20289)

[Redis的管道pipeline 179](#_Toc8186)

[MySQL 专题 180](#_Toc6114)

[与 Oracle 相比，Mysql 有什么优势？ 180](#_Toc21538)

[如何区分 FLOAT 和 DOUBLE？ 181](#_Toc19692)

[Mysql 驱动程序是什么？ 181](#_Toc19873)

~~[如何使用 Unix shell 登录 Mysql？](#_Toc28468)~~ [181](#_Toc28468)

[Mysql 如何优化 DISTINCT？ 181](#_Toc12437)

[MySQL如何优化DISTINCT？ 184](#_Toc1689)

[mysql DISTINCT 的实现与优化 184](#_Toc20678)

[mysql中关联查询和子查询那个更快 188](#_Toc32127)

[什么是通用 SQL 函数？ 189](#_Toc17864)

[MYSQL 支持事务吗？ 189](#_Toc28927)

[事务的基本要素 190](#_Toc21720)

[事务隔离级别 190](#_Toc25538)

[如何解决事务的并发问题(脏读，幻读)？ 190](#_Toc1745)

[binlog,redolog,undolog都是什么，起什么作用？ 192](#_Toc28844)

[Mysql结构 195](#_Toc12983)

[为什么选择B+树作为索引结构？ 195](#_Toc17329)

[索引B+树的叶子节点都可以存哪些东西？ 196](#_Toc18853)

[什么是聚簇索引 197](#_Toc4753)

[什么是覆盖索引 197](#_Toc22307)

[为什么不推荐使用select \* 197](#_Toc23640)

[什么是最左匹配原则 197](#_Toc302)

[查询在什么时候不走（预期中的）索引 198](#_Toc11713)

[explain是如何解析sql的？ 198](#_Toc28021)

[多线程 专题 198](#_Toc23040)

[什么是活锁、饥饿、无锁、死锁？ 199](#_Toc23965)

[微服务 专题 201](#_Toc21110)

[Kubernetes 203](#_Toc8844)

[前后端分离是如何做的 204](#_Toc23894)

[oauth2 207](#_Toc18178)

[RPC 的实现原理 207](#_Toc10837)

[Dubbo 的实现原理 208](#_Toc5131)

[算法题 208](#_Toc32202)

[算法题 208](#_Toc26137)

[操作系统 209](#_Toc11763)

[用户态和核心态 209](#_Toc16322)

[用户态和内核态的理解和区别 212](#_Toc19254)

[Python 214](#_Toc12135)

[Python 如何进行内存管理？/Python 中的垃圾回收机制？ 214](#_Toc21399)

[什么是 Python 生成器？ 216](#_Toc1897)

[什么是 Python 迭代器？ 216](#_Toc4800)

[迭代器 217](#_Toc9350)

[生成器 220](#_Toc6978)

[Python3 解释器 221](#_Toc7316)

[list 和 tuple 有什么区别？ 224](#_Toc27393)

[Python 中的 list 和 dict 是怎么实现的？ 224](#_Toc16318)

[python中set、deque、list、dict内部实现原理 224](#_Toc7148)

[Python 中使用多线程可以达到多核CPU一起使用吗？ 228](#_Toc26851)

[什么是装饰器？ 228](#_Toc19303)

[什么是 lambda 表达式？ 228](#_Toc30225)

[什么是深拷贝和浅拷贝？ 229](#_Toc18041)

[Python 直接赋值、浅拷贝和深度拷贝解析 229](#_Toc13015)

[双等于和 is 有什么区别？ 230](#_Toc14887)

[类型转换 230](#_Toc20332)

# mysql 常见知识点

mysql是用C++语言开发的。

## 一、为什么用自增列作为主键

1、如果我们定义了主键(PRIMARY KEY)，那么InnoDB会选择主键作为聚集索引。

如果没有显式定义主键，则InnoDB会选择第一个不包含有NULL值的唯一索引作为主键索引。

如果也没有这样的唯一索引，则InnoDB会选择内置6字节长的ROWID作为隐含的聚集索引(ROWID随着行记录的写入而主键递增，这个ROWID不像ORACLE的ROWID那样可引用，是隐含的)。

2、数据记录本身被存于主索引（一颗B+Tree）的叶子节点上，这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放

因此每当有一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB默认为15/16），则开辟一个新的页（节点）

3、如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页

4、如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置

此时MySQL不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销

同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。

## 二、为什么使用数据索引能提高效率

数据索引的存储是有序的

在有序的情况下，通过索引查询一个数据是无需遍历索引记录的

极端情况下，数据索引的查询效率为二分法查询效率，趋近于 log2(N)

## MySQL索引类型

一、简介

MySQL目前主要有以下几种索引类型：

1.主键索引

2.唯一索引

3.组合索引

4.普通索引

5.全文索引

## B+树 什么意思

B+ 树的创造者Rudolf Bayer没有解释B代表什么。最常见的观点是B代表平衡(balanced)，因为所有的叶子节点在树中都在相同的级别上。B也可能代表Bayer，或者是波音（Boeing），因为他曾经工作于波音科学研究实验室。 [1]

## 三、B+树索引和哈希索引的区别

在MySQL里常用的索引数据结构有B+树索引和哈希索引两种，我们来看下这两种索引数据结构的区别及其不同的应用建议。

二者区别

备注：在MySQL文档里，实际上是把B+树索引写成了BTREE，例如像下面这样的写法：

在MySQL中，只有HEAP/MEMORY引擎表才能显式支持哈希索引（NDB也支持，但这个不常用），InnoDB引擎的自适应哈希索引（adaptive hash index）不在此列，因为这不是创建索引时可指定的。

还需要注意到：HEAP/MEMORY引擎表在mysql实例重启后，数据会丢失。

通常，B+树索引结构适用于绝大多数场景，像下面这种场景用哈希索引才更有优势：

在HEAP表中，如果存储的数据重复度很低（也就是说基数很大），对该列数据以等值查询为主，没有范围查询、没有排序的时候，特别适合采用哈希索引

例如这种SQL：

SELECT … FROM t WHERE C1 = ?; — 仅等值查询

在大多数场景下，都会有范围查询、排序、分组等查询特征，用B+树索引就可以了。

B+树是一个平衡的多叉树，从根节点到每个叶子节点的高度差值不超过1，而且同层级的节点间有指针相互链接，是有序的，如下图：

哈希索引就是采用一定的哈希算法，把键值换算成新的哈希值，检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可,是无序的，如下图所示：

## 五、哈希索引不适用的场景：

1、不支持范围查询

2、不支持索引完成排序

3、不支持联合索引的最左前缀匹配规则

通常，B+树索引结构适用于绝大多数场景，像下面这种场景用哈希索引才更有优势：

在HEAP表中，如果存储的数据重复度很低（也就是说基数很大），对该列数据以等值查询为主，没有范围查询、没有排序的时候，特别适合采用哈希索引，例如这种SQL：

# 仅等值查询

select id, name from table where name='李明';

而常用的 InnoDB 引擎中默认使用的是B+树索引，它会实时监控表上索引的使用情况。

如果认为建立哈希索引可以提高查询效率，则自动在内存中的“自适应哈希索引缓冲区”建立哈希索引（在InnoDB中默认开启自适应哈希索引）。

通过观察搜索模式，MySQL会利用index key的前缀建立哈希索引，如果一个表几乎大部分都在缓冲池中，那么建立一个哈希索引能够加快等值查询。

注意：在某些工作负载下，通过哈希索引查找带来的性能提升远大于额外的监控索引搜索情况和保持这个哈希表结构所带来的开销。

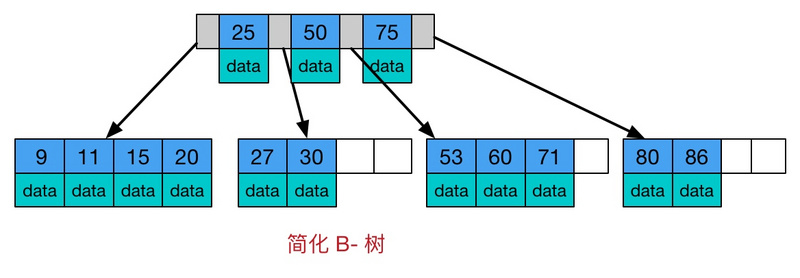
但某些时候，在负载高的情况下，自适应哈希索引中添加的read/write锁也会带来竞争，比如高并发的join操作。like操作和%的通配符操作也不适用于自适应哈希索引，可能要关闭自适应哈希索引。

## 四、哈希索引的优势：

等值查询，哈希索引具有绝对优势（前提是：没有大量重复键值，如果大量重复键值时，哈希索引的效率很低，因为存在所谓的哈希碰撞问题。）

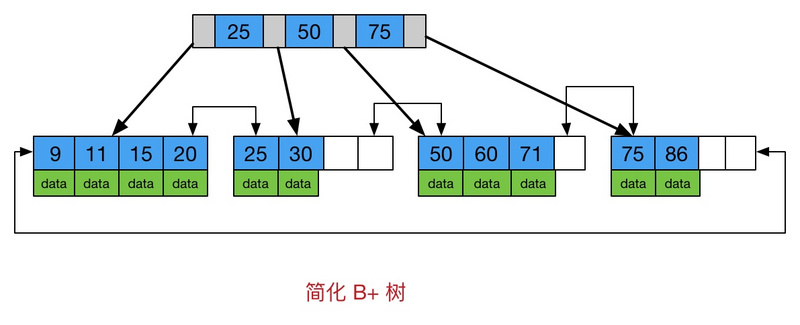
## 六、B树和B+树的区别

1、B树，每个节点都存储key和data，所有节点组成这棵树，并且叶子节点指针为nul，叶子结点不包含任何关键字信息。

M=4 阶的B树:



2、B+树，所有的叶子结点中包含了全部关键字的信息，及指向含有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的顺序链接

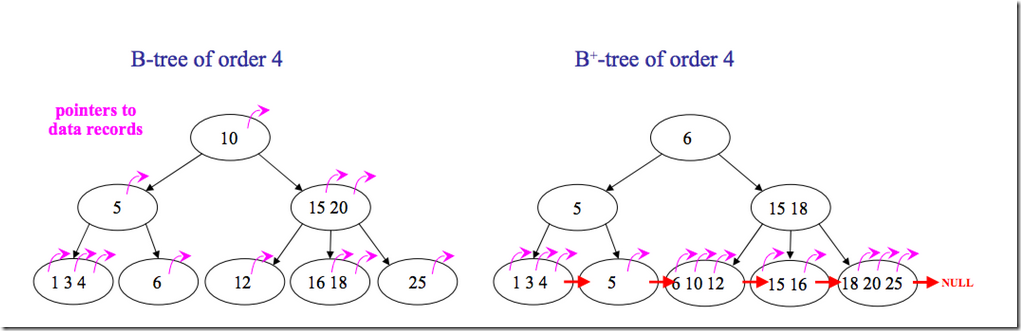




所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。 (而B 树的非终节点也包含需要查找的有效信息)

所有关键字存储在叶子节点出现,内部节点(非叶子节点并不存储真正的 data)

为所有叶子结点增加了一个链指针



## 七、为什么说B+比B树更适合实际应用中操作系统的文件索引和数据库索引？

1、B+的磁盘读写代价更低。

B+的内部结点并没有指向关键字具体信息的指针，因此其内部结点相对B树更小。

如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

2、B+-tree的查询效率更加稳定。

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

## 八、MySQL联合索引

1、联合索引是两个或更多个列上的索引。

对于联合索引:Mysql从左到右的使用索引中的字段，一个查询可以只使用索引中的一部份，但只能是最左侧部分。

例如索引是key index (a,b,c). 可以支持a 、 a,b 、 a,b,c 3种组合进行查找，但不支持 b,c进行查找 .当最左侧字段是常量引用时，索引就十分有效。

2、利用索引中的附加列，您可以缩小搜索的范围，但使用一个具有两列的索引不同于使用两个单独的索引。

复合索引的结构与电话簿类似，人名由姓和名构成，电话簿首先按姓氏对进行排序，然后按名字对有相同姓氏的人进行排序。

如果您知道姓，电话簿将非常有用；如果您知道姓和名，电话簿则更为有用，但如果您只知道名不知道姓，电话簿将没有用处。

## 九、什么情况下应不建或少建索引

1、表记录太少

2、经常增删改的表

3、数据重复且分布平均的表字段，假如一个表有10万行记录，有一个字段A只有T和F两种值，且每个值的分布概率大约为50%，那么对这种表A字段建索引一般不会提高数据库的查询速度。

4、经常和主字段一块查询但主字段索引值比较多的表字段

## 十、什么是表分区？

表分区，是指根据一定规则，将数据库中的一张表分解成多个更小的，容易管理的部分。从逻辑上看，只有一张表，但是底层却是由多个物理分区组成。

## 十一、表分区与分表的区别

分表：指的是通过一定规则，将一张表分解成多张不同的表。比如将用户订单记录根据时间成多个表。

分表与分区的区别在于：分区从逻辑上来讲只有一张表，而分表则是将一张表分解成多张表。

## 十二、表分区有什么好处？

1、存储更多数据。分区表的数据可以分布在不同的物理设备上，从而高效地利用多个硬件设备。和单个磁盘或者文件系统相比，可以存储更多数据

2、优化查询。在where语句中包含分区条件时，可以只扫描一个或多个分区表来提高查询效率；涉及sum和count语句时，也可以在多个分区上并行处理，最后汇总结果。

3、分区表更容易维护。例如：想批量删除大量数据可以清除整个分区。

4、避免某些特殊的瓶颈，例如InnoDB的单个索引的互斥访问，ext3问价你系统的inode锁竞争等。

## 十三、分区表的限制因素

1、一个表最多只能有1024个分区

MySQL5.1中，分区表达式必须是整数，或者返回整数的表达式。在MySQL5.5中提供了非整数表达式分区的支持。

2、全部主键和索引列统一放在一个分区里面。如果分区字段中有主键或者唯一索引的列，那么多有主键列和唯一索引列都必须包含进来。即：分区字段要么不包含主键或者索引列，要么包含。

3、分区表中无法使用外键约束

4、MySQL的分区适用于一个表的所有数据和索引，不能只对表数据分区而不对索引分区，也不能只对索引分区而不对表分区，也不能只对表的一部分数据分区。

## 十四、如何判断当前MySQL是否支持分区？

命令：show variables like '%partition%' 运行结果:

have\_partintioning 的值为YES，表示支持分区。

## 十五、MySQL支持的分区类型有哪些？

1、RANGE分区： 这种模式允许将数据划分不同范围。例如可以将一个表通过年份划分成若干个分区

2、LIST分区： 这种模式允许系统通过预定义的列表的值来对数据进行分割。按照List中的值分区，与RANGE的区别是，range分区的区间范围值是连续的。

3、HASH分区 ：这中模式允许通过对表的一个或多个列的Hash Key进行计算，最后通过这个Hash码不同数值对应的数据区域进行分区。例如可以建立一个对表主键进行分区的表。

4、KEY分区 ：上面Hash模式的一种延伸，这里的Hash Key是MySQL系统产生的。

## 事务的 四个特征（ACID）

事务具有四个特征：原子性（ Atomicity ）、一致性（ Consistency ）、隔离性（ Isolation ）和持续性（ Durability ）。这四个特性简称为 ACID 特性。

1 、原子性。事务是数据库的逻辑工作单位，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。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库系统 运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态，或者说是不一致的状态。

3 、隔离性。一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰。

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。接下来的其它操作或故障不应该对其执行结果有任何影响。

## 嵌套事务

一、问题起源

在MySQL的官方文档中有明确的说明不支持嵌套事务：

Transactions cannot be nested. This is a consequence of the implicit commit performed for any current transaction when you issue a START TRANSACTION statement or one of its synonyms.

但是在我们开发一个复杂的系统时难免会无意中在事务中嵌套了事务，比如A函数调用了B函数，A函数使用了事务，并且是在事务中调用了B函数，B函数也有一个事务，这样就出现了事务嵌套。这时候其实A的事务就意义不大了，为什么呢？上面的文档中就有提到，简单的翻译过来就是：

当执行一个START TRANSACTION指令时，会隐式的执行一个commit操作。

所以我们就要在系统架构层面来支持事务的嵌套。所幸的是在一些成熟的ORM框架中都做了对嵌套的支持，比如doctrine或者laravel。接下来我们就一起来看下这两个框架是怎样来实现的。

友情提示，这两个框架的函数和变量的命名都比较的直观，虽然看起来很长，但是都是通过命名就能直接得知这个函数或者变量的意思，所以不要一看到那么一大坨就被吓到了 :)

## 十六、Mysql的四种隔离级别

SQL标准定义了4类隔离级别，包括了一些具体规则，用来限定事务内外的哪些改变是可见的，哪些是不可见的。低级别的隔离级一般支持更高的并发处理，并拥有更低的系统开销。

### 1、Read Uncommitted（读取未提交内容）

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

### 2、Read Committed（读取提交内容）

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。

### 3、Repeatable Read（可重读）

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。

### 4、Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

出现问题

这四种隔离级别采取不同的锁类型来实现，若读取的是同一个数据的话，就容易发生问题。例如：

### 脏读(Drity Read)

某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。

### 不可重复读(Non-repeatable read)

在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。

### 幻读(Phantom Read)

在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。

## 四种隔离级别对照关系

Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

Read committed (读已提交)：可避免脏读的发生。

Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

## 十七、关于MVCC

MySQL InnoDB存储引擎，实现的是基于多版本的并发控制协议——MVCC (Multi-Version Concurrency Control)

注：与MVCC相对的，是基于锁的并发控制，Lock-Based Concurrency Control

MVCC最大的好处：读不加锁，读写不冲突。在读多写少的OLTP应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，现阶段几乎所有的RDBMS，都支持了MVCC。

LBCC：Lock-Based Concurrency Control，基于锁的并发控制

MVCC：Multi-Version Concurrency Control，多版本的并发控制

基于多版本的并发控制协议。纯粹基于锁的并发机制并发量低，MVCC是在基于锁的并发控制上的改进，主要是在读操作上提高了并发量。

## 十八、在MVCC并发控制中，读操作可以分成两类：

快照读 (snapshot read)：读取的是记录的可见版本 (有可能是历史版本)，不用加锁（共享读锁s锁也不加，所以不会阻塞其他事务的写）

当前读 (current read)：读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录

## 十九、行级锁定的优点：

使用行级锁定的主要有InnoDB存储引擎，以及MySQL的分布式存储引擎NDBCluster

1、当在许多线程中访问不同的行时只存在少量锁定冲突。

2、更改少好回滚。回滚时只有少量的更改。

3、可以长时间锁定单一的行。

## 二十、行级锁定的缺点：

1、比页级或表级锁定占用更多的内存。

2、当在表的大部分中使用时，比页级或表级锁定速度慢，因为你必须获取更多的锁。

3、如果你在大部分数据上经常进行GROUP BY操作或者必须经常扫描整个表，比其它锁定明显慢很多。

用高级别锁定，通过支持不同的类型锁定，你也可以很容易地调节应用程序，因为其锁成本小于行级锁定。

## 二十一、MySQL优化

1、开启查询缓存，优化查询

explain：解释

2、explain你的select查询，这可以帮你分析你的查询语句或是表结构的性能瓶颈。EXPLAIN 的查询结果还会告诉你你的索引主键被如何利用的，你的数据表是如何被搜索和排序的

3、当只要一行数据时使用limit 1，MySQL数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据

避免使用 select \*

4、为搜索字段建索引

5、使用 ENUM 而不是 VARCHAR。如果你有一个字段，比如“性别”，“国家”，“民族”，“状态”或“部门”，你知道这些字段的取值是有限而且固定的，那么，你应该使用 ENUM 而不是VARCHAR

Prepared Statements Prepared Statements很像存储过程，是一种运行在后台的SQL语句集合，我们可以从使用 prepared statements 获得很多好处，无论是性能问题还是安全问题。

6、Prepared Statements 可以检查一些你绑定好的变量，这样可以保护你的程序不会受到“SQL注入式”攻击

预编译语句(Prepared Statements)

6.1. 预编译语句是什么

通常我们的一条sql在db接收到最终执行完毕返回可以分为下面三个过程：

a、词法和语义解析

b、优化sql语句，制定执行计划

c、执行并返回结果

我们把这种普通语句称作Immediate Statements。

但是很多情况，我们的一条sql语句可能会反复执行，或者每次执行的时候只有个别的值不同（比如query的where子句值不同，update的set子句值不同,insert的values值不同）。

如果每次都需要经过上面的词法语义解析、语句优化、制定执行计划等，则效率就明显不行了。

所谓预编译语句就是将这类语句中的值用占位符替代，可以视为将sql语句模板化或者说参数化，一般称这类语句叫Prepared Statements或者Parameterized Statements

预编译语句的优势在于归纳为：一次编译、多次运行，省去了解析优化等过程；此外预编译语句能防止sql注入。

当然就优化来说，很多时候最优的执行计划不是光靠知道sql语句的模板就能决定了，往往就是需要通过具体值来预估出成本代价。

6.2. MySQL的预编译功能

注意MySQL的老版本（4.1之前）是不支持服务端预编译的，但基于目前业界生产环境普遍情况，基本可以认为MySQL支持服务端预编译。

7、垂直分表

8、选择正确的存储引擎

## 二十二、key和index的区别

key 是数据库的物理结构，它包含两层意义和作用，一是约束（偏重于约束和规范数据库的结构完整性），二是索引（辅助查询用的）。包括primary key, unique key, foreign key 等

index是数据库的物理结构，它只是辅助查询的，它创建时会在另外的表空间（mysql中的innodb表空间）以一个类似目录的结构存储。索引要分类的话，分为前缀索引、全文本索引等；

## 二十三、Mysql 中 MyISAM 和 InnoDB 的区别有哪些？

区别：

1、InnoDB支持事务，MyISAM不支持

对于InnoDB每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务；

2、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；

3、InnoDB是聚集索引，数据文件是和索引绑在一起的，必须要有主键，通过主键索引效率很高。

但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此主键不应该过大，因为主键太大，其他索引也都会很大。

而MyISAM是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。

4、InnoDB不保存表的具体行数，执行select count(\*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快；

5、Innodb不支持全文索引，而MyISAM支持全文索引，查询效率上MyISAM要高；

如何选择：

是否要支持事务，如果要请选择innodb，如果不需要可以考虑MyISAM；

如果表中绝大多数都只是读查询，可以考虑MyISAM，如果既有读写也挺频繁，请使用InnoDB

系统奔溃后，MyISAM恢复起来更困难，能否接受；

MySQL5.5版本开始Innodb已经成为Mysql的默认引擎(之前是MyISAM)，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。

## 二十四、数据库表创建注意事项

### 1、字段名及字段配制合理性

剔除关系不密切的字段；

字段命名要有规则及相对应的含义（不要一部分英文，一部分拼音，还有类似a.b.c这样不明含义的字段）；

字段命名尽量不要使用缩写（大多数缩写都不能明确字段含义）；

字段不要大小写混用（想要具有可读性，多个英文单词可使用下划线形式连接）；

字段名不要使用保留字或者关键字；

保持字段名和类型的一致性；

慎重选择数字类型；

给文本字段留足余量；

### 2、系统特殊字段处理及建成后建议

添加删除标记（例如操作人、删除时间）；

建立版本机制；

### 3、表结构合理性配置

多型字段的处理，就是表中是否存在字段能够分解成更小独立的几部分（例如：人可以分为男人和女人）；

多值字段的处理，可以将表分为三张表，这样使得检索和排序更加有调理，且保证数据的完整性！

### 4、其它建议

选用utf8mb4字符集

对于大数据字段，独立表进行存储，以便影响性能（例如：简介字段）；

使用varchar类型代替char，因为varchar会动态分配长度，char指定长度是固定的；

给表创建主键，对于没有主键的表，在查询和索引定义上有一定的影响；

避免表字段运行为null，建议设置默认值（例如：int类型设置默认值为0）在索引查询上，效率立显；

建立索引，最好建立在唯一和非空的字段上，建立太多的索引对后期插入、更新都存在一定的影响（考虑实际情况来创建）；

## 二十五、数据库范式

本词条由“科普中国”科学百科词条编写与应用工作项目 审核 。

设计关系数据库时，遵从不同的规范要求，设计出合理的关系型数据库，这些不同的规范要求被称为不同的范式，各种范式呈递次规范，越高的范式数据库冗余越小。

目前关系数据库有六种范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、第四范式(4NF）和第五范式（5NF，又称完美范式）。

### 范式简介

范式来自英文Normal form，简称NF。要想设计—个好的关系，必须使关系满足一定的约束条件，此约束已经形成了规范，分成几个等级，一级比一级要求得严格。满足这些规范的数据库是简洁的、结构明晰的，同时，不会发生插入(insert)、删除(delete)和更新(update)操作异常。反之则是乱七八糟，不仅给数据库的编程人员制造麻烦，而且面目可憎，可能存储了大量不需要的冗余信息。

目前关系数据库有六种范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、第四范式(4NF）和第五范式（5NF，又称完美范式）。满足最低要求的范式是第一范式（1NF）。在第一范式的基础上进一步满足更多规范要求的称为第二范式（2NF），其余范式以次类推。一般来说，数据库只需满足第三范式(3NF）就行了。

### 1、第一范式（1NF）：

所谓第一范式（1NF）是指在关系模型中，对于添加的一个规范要求，所有的域都应该是原子性的，即数据库表的每一列都是不可分割的原子数据项，而不能是集合，数组，记录等非原子数据项。即实体中的某个属性有多个值时，必须拆分为不同的属性。在符合第一范式（1NF）表中的每个域值只能是实体的一个属性或一个属性的一部分。简而言之，第一范式就是无重复的域。

说明：在任何一个关系数据库中，第一范式（1NF）是对关系模式的设计基本要求，一般设计中都必须满足第一范式（1NF）。不过有些关系模型中突破了1NF的限制，这种称为非1NF的关系模型。换句话说，是否必须满足1NF的最低要求，主要依赖于所使用的关系模型。

### 2、第二范式（2NF）

在1NF的基础上，非码属性必须完全依赖于候选码（在1NF基础上消除非主属性对主码的部分函数依赖）

第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或记录必须可以被唯一地区分。选取一个能区分每个实体的属性或属性组，作为实体的唯一标识。例如在员工表中的身份证号码即可实现每个一员工的区分，该身份证号码即为候选键，任何一个候选键都可以被选作主键。在找不到候选键时，可额外增加属性以实现区分，如果在员工关系中，没有对其身份证号进行存储，而姓名可能会在数据库运行的某个时间重复，无法区分出实体时，设计辟如ID等不重复的编号以实现区分，被添加的编号或ID选作主键。（该主键的添加是在ER设计时添加，不是建库时随意添加）

第二范式（2NF）要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。简而言之，第二范式就是在第一范式的基础上属性完全依赖于主键。

### 3、第三范式（3NF）

在2NF基础上，任何非主属性不依赖于其它非主属性（在2NF基础上消除传递依赖）

数据库范式

数据库范式

第三范式（3NF）是第二范式（2NF）的一个子集，即满足第三范式（3NF）必须满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个关系中不包含已在其它关系已包含的非主关键字信息。例如，存在一个部门信息表，其中每个部门有部门编号（dept\_id）、部门名称、部门简介等信息。那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。简而言之，第三范式就是属性不依赖于其它非主属性，也就是在满足2NF的基础上，任何非主属性不得传递依赖于主属性。

### 4、巴斯-科德范式（BCNF）

Boyce-Codd Normal Form（巴斯-科德范式）

在3NF基础上，任何非主属性不能对主键子集依赖（在3NF基础上消除对主码子集的依赖）

巴斯-科德范式（BCNF）是第三范式（3NF）的一个子集，即满足巴斯-科德范式（BCNF）必须满足第三范式（3NF）。通常情况下，巴斯-科德范式被认为没有新的设计规范加入，只是对第二范式与第三范式中设计规范要求更强，因而被认为是修正第三范式，也就是说，它事实上是对第三范式的修正，使数据库冗余度更小。这也是BCNF不被称为第四范式的原因。某些书上，根据范式要求的递增性将其称之为第四范式是不规范，也是更让人不容易理解的地方。而真正的第四范式，则是在设计规范中添加了对多值及依赖的要求。 [1]

### 目的原则

规范化目的是使结构更合理，消除存储异常，使数据冗余尽量小。便于插入、删除和更新。

遵从概念单一化“一事一地”原则，即一个关系模式描述一个实体或实体间的一种联系。规范的实质就是概念的单一化。

一个关系模式接着分解可以得到不同关系模式集合，也就是说分解方法不是惟一的。最小冗余的要求必须以分解后的数据库能够表达原来数据库所有信息为前提来实现。其根本目标是节省存储空问，避免数据不一致性，提高对关系的操作效率，同时满足应用需求。实际上，并不一定要求全部模式都达到BCNF不可。有时故意保留部分冗余可能更方便数据查询。尤其对于那些更新频度不高，查询频度极高的数据库系统更是如此。

### 反范式

规范化的优点是明显的，它避免了大量的数据冗余，节省了存储空间，保持了数据的一致性。当一个库里的数据经常发生变化时，达到3NF的库可以使用户不必在超过两个以上的地方更改同一个值。那么是不是只要把所有的表都规范为3NF后，数据库的设计就是最优的呢?这可不一定。范式越高意味着表的划分更细，一个数据库中需要的表也就越多，用户不得不将原本相关联的数据分摊到多个表中。当用户同时需要这些数据时只能采用连接表的形式将数据重新合并在一起。同时把多个表联接在一起的花费是巨大的，尤其是当需要连接的两张或者多张表数据非常庞大的时候，表连接操作几乎是一个噩梦，这严重地降低了系统运行性能。 [2]

保持一点冗余，空间换时间

### BC范式

BC范式（BCNF）是Boyce-Codd范式的缩写，其定义是：在关系模式中每一个决定因素都包含候选键，也就是说，只要属性或属性组A能够决定任何一个属性B，则A的子集中必须有候选键。BCNF范式排除了任何属性(不光是非主属性，2NF和3NF所限制的都是非主属性)对候选键的传递依赖与部分依赖。

比如我们有一个学生导师表，其中包含字段：学生ID，专业，导师，专业GPA，这其中学生ID和专业是联合主键。

StudentId Major Advisor MajGPA

1 人工智能 Edward 4.0

2 大数据 William 3.8

1 大数据 William 3.7

3 大数据 Joseph 4.0

这个表的设计满足三范式，有主键，不存在主键的部分依赖，不存在非主键的传递依赖。但是这里存在另一个依赖关系，“专业”函数依赖于“导师”，也就是说每个导师只做一个专业方面的导师，只要知道了是哪个导师，我们自然就知道是哪个专业的了。

所以这个表的部分主键依赖于非主键部分，那么我们可以进行以下的调整，拆分成2个表：

学生导师表：

StudentId Advisor MajGPA

1 Edward 4.0

2 William 3.8

1 William 3.7

3 Joseph 4.0

### 第四范式

如果满足了BC范式，那么就不再会有任何由于函数依赖导致的异常，但是我们还可能会遇到由于多值依赖导致的异常。

比如我们建立课程教师和教材的模型，我们规定，每门课程有对应的一组教师，每门课程也有对应的一组教材，一门课程使用的教程和教师没有关系。这样我们首先肯定有三个实体表，分别表示课程，教师和教材。现在我们要建立这三个对象的关系，于是我们建立的关系表，定义如下：

课程ID，教师ID，教程ID；这三列作为联合主键。

以下是示例，为了表述方便，我们用Name代替ID，这样更容易看懂：

Course Teacher Book

英语 Bill 人教版英语

英语 Bill 美版英语

英语 Jay 美版英语

高数 William 人教版高数

高数 Dave 美版高数

这个表除了主键，就没有其他字段了，所以肯定满足BC范式，但是却存在多值依赖导致的异常。

我们先来看看多值依赖的定义：

一个关系，至少存在三个属性（A、B、C），才能存在这种关系。对于每一个A值，有一组确定的B值和C值，并且这组B的值独立于这组C的值。

假如我们下学期想采用一本新的英版高数教材，但是还没确定具体哪个老师来教，那么我们就无法在这个表中维护Course高数和Book英版高数教材的的关系。

解决办法是我们把这个多值依赖的表拆解成2个表，分别建立关系。这是我们拆分后的表：

Course Teacher

英语 Bill

英语 Jay

高数 William

高数 Dave

Course Book

英语 人教版英语

英语 美版英语

高数 人教版高数

高数 美版高数

第四范式的定义很简单：已经是BC范式，并且不包含多值依赖关系。

除了第四范式外，我们还有更高级的第五范式和域键范式（DKNF），第五范式处理的是无损连接问题，这个范式基本没有实际意义，因为无损连接很少出现，而且难以察觉。而域键范式试图定义一个终极范式，该范式考虑所有的依赖和约束类型，但是实用价值也是最小的，只存在理论研究中。

# Redis常见知识点

http://jasontec.cn/articles/2019/12/20/1576826633007.html

介绍：Redis 是一个开源的使用 ANSI C 语言编写、遵守 BSD 协议、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API的非关系型数据库。

传统数据库遵循 ACID 规则。而 Nosql（Not Only SQL 的缩写，是对不同于传统的关系型数据库的数据库管理系统的统称） 一般为分布式而分布式一般遵循 CAP 定理。

Github 源码：https://github.com/antirez/redis

Redis 官网：https://redis.io/

## Redis支持的数据类型？

### String字符串：

是redis中最基本的数据类型，一个key对应一个value。

String类型是二进制安全的，意思是 redis 的 string 可以包含任何数据。如数字，字符串，jpg图片或者序列化的对象。

格式: set key value

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。

在 C 语言中，字符串里面不能包含空字符，否则这个空字符会被当做是字符串结尾，换句话说，C 语言的字符串默认是以 '\0' 结尾的，这不是二进制安全的，因为图片、音频等二进制数据里面会有 '\0' 这一字符，C 字符串会忽略 '\0' 这一字符后面的数据。在 Redis 中，有这样一个结构：

struct sdshdr {

int len; int free; char buf[]; };

它是用来表示字符串值的，称为 SDS(simple dynamic string)， buf 是一个 char 类型的数组，用来保存二进制数据，len 记录字符串的长度（不一定是 buf 数组的长度），SDS 不使用空字符串，而是使用 len 的值来判断字符串是否结束，因此 Redis 可以保存特殊数据格式，包括二进制数据。

如果要给二进制安全一个定义的话，可以这样描述：一个函数或方法能将任意输入作为原始的，无任何特殊格式意义的数据流，那么它就是二进制安全的。

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

使用：get 、 set 、 del 、 incr、 decr 等

实战场景：

1.缓存： 经典使用场景，把常用信息，字符串，图片或者视频等信息放到redis中，redis作为缓存层，mysql做持久化层，降低mysql的读写压力。

2.计数器：redis是单线程模型，一个命令执行完才会执行下一个，同时数据可以一步落地到其他的数据源。

3.session：常见方案spring session + redis实现session共享，

### Hash（哈希）

格式: hmset name key1 value1 key2 value2

Redis hash 是一个键值(key=>value)对集合。

Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。

是一个Mapmap，指值本身又是一种键值对结构，如 value={{field1,value1},......fieldN,valueN}}

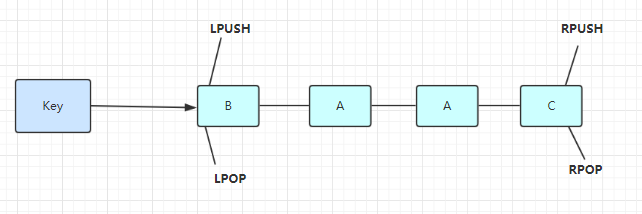
使用：所有hash的命令都是 h 开头的hget 、hset 、 hdel 等

实战场景：

1.缓存： 能直观，相比string更节省空间，的维护缓存信息，如用户信息，视频信息等。

### List（列表）

List 说白了就是链表（redis 使用双端链表实现的 List），是有序的，value可以重复，可以通过下标取出对应的value值，左右两边都能进行插入和删除数据。



使用列表的技巧

lpush+lpop=Stack(栈)

lpush+rpop=Queue（队列）

lpush+ltrim=Capped Collection（有限集合）

lpush+brpop=Message Queue（消息队列）

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）

格式: lpush name value

在 key 对应 list 的头部添加字符串元素

格式: rpush name value

在 key 对应 list 的尾部添加字符串元素

格式: lrem name index

key 对应 list 中删除 count 个和 value 相同的元素

格式: llen name

返回 key 对应 list 的长度

Redis Ltrim 对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

下标 0 表示列表的第一个元素，以 1 表示列表的第二个元素，以此类推。 你也可以使用负数下标，以 -1 表示列表的最后一个元素， -2 表示列表的倒数第二个元素，以此类推。

语法

redis Ltrim 命令基本语法如下：

redis 127.0.0.1:6379> LTRIM KEY\_NAME START STOP

Redis Brpop 命令移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

语法

redis Brpop 命令基本语法如下：

redis 127.0.0.1:6379> BRPOP LIST1 LIST2 .. LISTN TIMEOUT

返回值

假如在指定时间内没有任何元素被弹出，则返回一个 nil 和等待时长。 反之，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 key ，第二个元素是被弹出元素的值。

List实战场景：

1.timeline：例如微博的时间轴，有人发布微博，用lpush加入时间轴，展示新的列表信息。

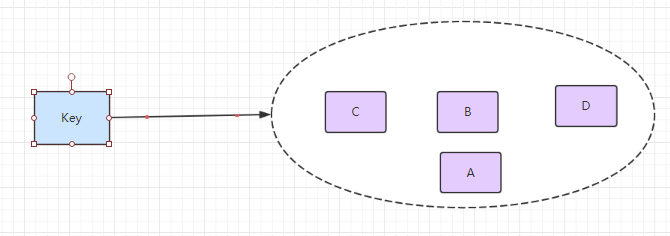
### Set（集合）

格式: sadd name value

Redis的Set是string类型的无序集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

集合类型也是用来保存多个字符串的元素，但和列表不同的是集合中 1. 不允许有重复的元素，2.集合中的元素是无序的，不能通过索引下标获取元素，3.支持集合间的操作，可以取多个集合取交集、并集、差集。



使用：命令都是以s开头的 sset 、srem、scard、smembers、sismember

实战场景;

1.标签（tag）,给用户添加标签，或者用户给消息添加标签，这样有同一标签或者类似标签的可以给推荐关注的事或者关注的人。

2.点赞，或点踩，收藏等，可以放到set中实现

### zset(sorted set：有序集合)

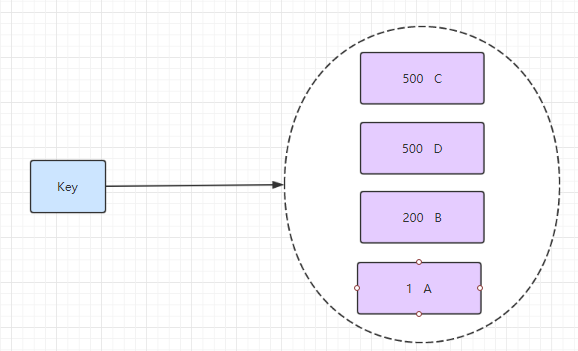
格式: zadd name score value

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

有序集合和集合有着必然的联系，保留了集合不能有重复成员的特性，区别是，有序集合中的元素是可以排序的，它给每个元素设置一个分数，作为排序的依据。

（有序集合中的元素不可以重复，但是score 分数 可以重复，就和一个班里的同学学号不能重复，但考试成绩可以相同）。



zset的成员是唯一的,但分数(score)却可以重复。

使用： 有序集合的命令都是 以 z 开头 zadd 、 zrange、 zscore

实战场景：

1.排行榜：有序集合经典使用场景。例如小说视频等网站需要对用户上传的小说视频做排行榜，榜单可以按照用户关注数，更新时间，字数等打分，做排行。

## 什么是Redis持久化？Redis有哪几种持久化方式？优缺点是什么？

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

Redis 提供了两种持久化方式:RDB（默认） 和AOF

### RDB：

rdb是Redis DataBase缩写

功能核心函数两个函数

rdbSave：生成RDB文件

rdbLoad：从文件加载内存

### AOF:

Aof是Append-only file缩写

每当执行服务器(定时)任务或者函数时flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作

aof写入保存：

WRITE：根据条件，将 aof\_buf 中的缓存写入到 AOF 文件

SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

存储结构:

内容是redis通讯协议(RESP )格式的命令文本存储。

比较：

1、aof文件比rdb更新频率高，优先使用aof还原数据。

2、aof比rdb更安全也更大

3、rdb性能比aof好

4、如果两个都配了优先加载AOF

刚刚上面你有提到redis通讯协议(RESP )，能解释下什么是RESP？有什么特点？（可以看到很多面试其实都是连环炮，面试官其实在等着你回答到这个点，如果你答上了对你的评价就又加了一分）

RESP 是redis客户端和服务端之前使用的一种通讯协议；

RESP 的特点：实现简单、快速解析、可读性好

For Simple Strings the first byte of the reply is "+" 回复

For Errors the first byte of the reply is "-" 错误

For Integers the first byte of the reply is ":" 整数

For Bulk Strings the first byte of the reply is "$" 字符串

For Arrays the first byte of the reply is "\*" 数组

## Redis 有哪些架构模式？讲讲各自的特点

### 单机版

特点：简单

问题：

1、内存容量有限 2、处理能力有限 3、无法高可用。

### 主从复制

Redis 的复制（replication）功能允许用户根据一个 Redis 服务器来创建任意多个该服务器的复制品，其中被复制的服务器为主服务器（master），而通过复制创建出来的服务器复制品则为从服务器（slave）。 只要主从服务器之间的网络连接正常，主从服务器两者会具有相同的数据，主服务器就会一直将发生在自己身上的数据更新同步 给从服务器，从而一直保证主从服务器的数据相同。

特点：

1、master/slave 角色

2、master/slave 数据相同

3、降低 master 读压力在转交从库

问题：

无法保证高可用

没有解决 master 写的压力

### 哨兵

Redis sentinel 是一个分布式系统中监控 redis 主从服务器，并在主服务器下线时自动进行故障转移。其中三个特性：

监控（Monitoring）： Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。

提醒（Notification）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。

自动故障迁移（Automatic failover）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作。

特点：

1、保证高可用

2、监控各个节点

3、自动故障迁移

缺点：

主从模式，切换需要时间

丢数据

没有解决 master 写的压力

### 集群（proxy 型）：

Twemproxy 是一个 Twitter 开源的一个 redis 和 memcache 快速/轻量级代理服务器； Twemproxy 是一个快速的单线程代理程序，支持 Memcached ASCII 协议和 redis 协议。

特点：

1、多种 hash 算法：MD5、CRC16、CRC32、CRC32a、hsieh、murmur、Jenkins

2、支持失败节点自动删除

3、后端 Sharding 分片逻辑对业务透明，业务方的读写方式和操作单个 Redis 一致

缺点：增加了新的 proxy，需要维护其高可用。

failover 逻辑需要自己实现，其本身不能支持故障的自动转移，可扩展性差，进行扩缩容都需要手动干预

### 集群（直连型）：

从redis 3.0之后版本支持redis-cluster集群，Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。

特点：

1、无中心架构（不存在哪个节点影响性能瓶颈），少了 proxy 层。

2、数据按照 slot 存储分布在多个节点，节点间数据共享，可动态调整数据分布。

3、可扩展性，可线性扩展到 1000 个节点，节点可动态添加或删除。

4、高可用性，部分节点不可用时，集群仍可用。通过增加 Slave 做备份数据副本

5、实现故障自动 failover（故障转移），节点之间通过 gossip 协议交换状态信息，用投票机制完成 Slave到 Master 的角色提升。

缺点：

1、资源隔离性较差，容易出现相互影响的情况。

2、数据通过异步复制,不保证数据的强一致性

什么是一致性哈希算法？什么是哈希槽？

这两个问题篇幅过长 网上找了两个解锁的不错的文章

https://www.cnblogs.com/lpfuture/p/5796398.html

http://www.jasontec.cn/articles/2020/04/11/1586586130767.html

## Redis常用命令？

Keys pattern

\*表示区配所有

以bit开头的

查看Exists key是否存在

Set

设置 key 对应的值为 string 类型的 value。

setnx

设置 key 对应的值为 string 类型的 value。如果 key 已经存在，返回 0，nx 是 not exist 的意思。

删除某个key

第一次返回1 删除了 第二次返回0

Expire 设置过期时间（单位秒）

TTL查看剩下多少时间

返回负数则key失效，key不存在了

Setex

设置 key 对应的值为 string 类型的 value，并指定此键值对应的有效期。

Mset

一次设置多个 key 的值，成功返回 ok 表示所有的值都设置了，失败返回 0 表示没有任何值被设置。

Getset

设置 key 的值，并返回 key 的旧值。

Mget

一次获取多个 key 的值，如果对应 key 不存在，则对应返回 nil。

Incr

对 key 的值做加加操作,并返回新的值。注意 incr 一个不是 int 的 value 会返回错误，incr 一个不存在的 key，则设置 key 为 1

incrby

同 incr 类似，加指定值 ，key 不存在时候会设置 key，并认为原来的 value 是 0

Decr

对 key 的值做的是减减操作，decr 一个不存在 key，则设置 key 为-1

Decrby

同 decr，减指定值。

Append

给指定 key 的字符串值追加 value,返回新字符串值的长度。

Strlen

取指定 key 的 value 值的长度。

persist xxx(取消过期时间)

选择数据库（0-15库）

Select 0 //选择数据库

move age 1//把age 移动到1库

Randomkey随机返回一个key

Rename重命名

Type 返回数据类型

08

## 使用过Redis分布式锁么，它是怎么实现的？

先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。

如果在setnx之后执行expire之前进程意外crash或者要重启维护了，那会怎么样？

set指令有非常复杂的参数，这个应该是可以同时把setnx和expire合成一条指令来用的！

单节点Redis实现一个简单的分布式锁。

1、加锁

加锁实际上就是在redis中，给Key键设置一个值，为避免死锁，并给定一个过期时间。

SET lock\_key random\_value NX PX 5000

值得注意的是：

random\_value 是客户端生成的唯一的字符串。

NX 代表只在键不存在时，才对键进行设置操作。

PX 5000 设置键的过期时间为5000毫秒。

这样，如果上面的命令执行成功，则证明客户端获取到了锁。

2、解锁

解锁的过程就是将Key键删除。但也不能乱删，不能说客户端1的请求将客户端2的锁给删除掉。这时候random\_value的作用就体现出来。

为了保证解锁操作的原子性，我们用LUA脚本完成这一操作。先判断当前锁的字符串是否与传入的值相等，是的话就删除Key，解锁成功。

if redis.call('get',KEYS[1]) == ARGV[1] then

return redis.call('del',KEYS[1])

else

return 0

end

## 使用过Redis做异步队列么，你是怎么用的？有什么缺点？

### 用法

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

### 缺点：

在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。

能不能生产一次消费多次呢？

使用pub/sub主题订阅者模式，可以实现1:N的消息队列。

## 什么是缓存穿透？如何避免？什么是缓存雪崩？何如避免？

### 缓存穿透

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。一些恶意的请求会故意查询不存在的key,请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

### 如何避免？

1：对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。

2：对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该bitmap过滤。

### 缓存雪崩

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，会给后端系统带来很大压力。导致系统崩溃。

### 如何避免？

1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

2：做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期

3：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

## 布隆过滤器

布隆过滤器（Bloom Filter）是1970年由布隆提出的。

它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。

优点是空间效率和查询时间都比一般的算法要好的多

缺点是有一定的误识别率和删除困难。

### 基本概念

直观的说，bloom算法类似一个hash set，用来判断某个元素（key）是否在某个集合中。

和一般的hash set不同的是，这个算法无需存储key的值，对于每个key，只需要k个比特位，每个存储一个标志，用来判断key是否在集合中。

算法：

1. 首先需要k个hash函数，每个函数可以把key散列成为1个整数

2. 初始化时，需要一个长度为n比特的数组，每个比特位初始化为0

3. 某个key加入集合时，用k个hash函数计算出k个散列值，并把数组中对应的比特位置为1

4. 判断某个key是否在集合时，用k个hash函数计算出k个散列值，并查询数组中对应的比特位，如果所有的比特位都是1，认为在集合中。

优点：不需要存储key，节省空间

缺点：

1. 算法判断key在集合中时，有一定的概率key其实不在集合中

2. 无法删除

典型的应用场景：

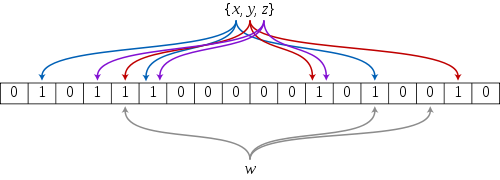
比如在字处理软件中，需要检查一个英语单词是否拼写正确（也就是要判断 它是否在已知的字典中）；在 FBI，一个嫌疑人的名字是否已经在嫌疑名单上；在网络爬虫里，一个网址是否被访问过等等。

某些存储系统的设计中，会存在空查询缺陷：当查询一个不存在的key时，需要访问慢设备，导致效率低下。

比如一个前端页面的缓存系统，可能这样设计：先查询某个页面在本地是否存在，如果存在就直接返回，如果不存在，就从后端获取。但是当频繁从缓存系统查询一个页面时，缓存系统将会频繁请求后端，把压力导入后端。

这是只要增加一个bloom算法的服务，后端插入一个key时，在这个服务中设置一次

需要查询后端时，先判断key在后端是否存在，这样就能避免后端的压力。



# MongoDB知识点

## 1.什么是MongoDB

MongoDB是一个文档数据库，提供好的性能，领先的非关系型数据库。采用BSON存储文档数据。

BSON（）是一种类json的一种二进制形式的存储格式，简称Binary JSON.

相对于json多了date类型和二进制数组。

1. 什么是bson

BSON是一种类json的一种二进制形式的存储格式，简称Binary JSON，它和JSON一样，支持内嵌的文档对象和数组对象，但是BSON有JSON没有的一些数据类型，如Date和BinData类型。

BSON可以做为网络数据交换的一种存储形式，这个有点类似于Google的Protocol Buffer，但是BSON是一种schema-less的存储形式，它的优点是灵活性高，但它的缺点是空间利用率不是很理想，

BSON有三个特点：轻量性、可遍历性、高效性

{“hello":"world"} 这是一个BSON的例子，其中"hello"是key name，它一般是cstring类型，字节表示是cstring::= (byte\*) "/x00" ,其中\*表示零个或多个byte字节，/x00表示结束符;后面的"world"是value值，它的类型一般是string,double,array,binarydata等类型。

2. bson在MongoDB中的使用

MongoDB使用了BSON这种结构来存储数据和网络数据交换。把这种格式转化成一文档这个概念(Document)，因为BSON是schema-free的，所以在MongoDB中所对应的文档也有这个特征，这里的一个Document也可以理解成关系数据库中的一条记录(Record)，只是这里的Document的变化更丰富一些，如Document可以嵌套。

MongoDB以BSON做为其存储结构的一种重要原因是其可遍历性。

## 2.MongoDB的优势有哪些

增：

面向文档的存储：以 JSON 格式的文档保存数据。

复制以及高可扩展性。

自动分片。

删：

改：

快速的即时更新。

查：

任何属性都可以建立索引。

丰富的查询功能。

## 3 什么是数据库

数据库可以看成是一个电子化的文件柜,用户可以对文件中的数据运行新增、检索、更新、删除等操作。数据库是一个

所有集合的容器，在文件系统中每一个数据库都有一个相关的物理文件。

## 4.什么是集合(表)

集合就是一组 MongoDB 文档。它相当于关系型数据库（RDBMS）中的表这种概念。集合位于单独的一个数据库中。

一个集合内的多个文档可以有多个不同的字段。一般来说，集合中的文档都有着相同或相关的目的。

## 5 什么是文档(记录)

文档由一组key value组成。文档是动态模式,这意味着同一集合里的文档不需要有相同的字段和结构。

在关系型数据库中table中的每一条记录相当于MongoDB中的一个文档。

## 6 MongoDB和关系型数据库术语对比图



## 7.什么是非关系型数据库

非关系型数据库的显著特点是不使用SQL作为查询语言，数据存储不需要特定的表格模式。

## 8 为什么用MongoDB？

架构简单

容易调试

容易扩展

没有复杂的连接

不需要转化/映射应用对象到数据库对象

使用内部内存作为存储工作区,以便更快的存取数据。

深度查询能力,MongoDB支持动态查询。

## 9 在哪些场景使用MongoDB

大数据

内容管理系统

移动端Apps

数据管理

## 10 MongoDB中的命名空间是什么意思?

mongodb存储bson对象在丛集(collection)中.数据库名字和丛集名字以句点连结起来叫做名字空间(namespace). 一个集合命名空间又有多个数据域(extent)，集合命名空间里存储着集合的元数据，比如集合名称，集合的

第一个数据域和最后一个数据域的位置等等。而一个数据域由若干条文档(document)组成，每个数据域都有一个

头部，记录着第一条文档和最后一条文档的为知，以及该数据域的一些元数据。extent之间，document之间通过双向链表连接。 索引的存储数据结构是B树，索引命名空间存储着对B树的根节点的指针。

## 11 monogodb 中的分片什么意思

分片是将数据水平切分到不同的物理节点。当应用数据越来越大的时候，数据量也会越来越大。当数据量增长时，单台机器有可能无法存储数据或可接受的读取写入吞吐量。利用分片技术可以添加更多的机器来应对数据量增加以及读写操作的要求。

## 12 为什么要在MongoDB中使用分析器

mongodb中包括了一个可以显示数据库中每个操作性能特点的数据库分析器.通过这个分析器你可以找到比预期慢的查询(或写操作);利用这一信息,比如,可以确定是否需要添加索引.

## 13 .MongoDB支持主键外键关系吗

默认MongoDB不支持主键和外键关系。 用Mongodb本身的API需要硬编码才能实现外键关联，不够直观且难度较大

## 14 MongoDB支持哪些数据类型

String

Integer

Double

Boolean

Datetime

Object

Object ID

Arrays

Min/Max Keys

Code

Regular Expression等

## 15 为什么要在MongoDB中用"Code"数据类型

"Code"类型用于在文档中存储 JavaScript 代码。

## 16 为什么要在MongoDB中用"Regular Expression"数据类型

Regular Expression：正则表达式

"Regular Expression"类型用于在文档中存储正则表达式

## 17 为什么在MongoDB中使用"Object ID"数据类型

"ObjectID"数据类型用于存储文档id

## 18 "ObjectID"有哪些部分组成

一共有四部分组成:时间戳、客户端ID、客户进程ID、三个字节的增量计数器

## 19 在MongoDb中什么是索引

索引用于高效的执行查询,没有索引的MongoDB将扫描整个集合中的所有文档,这种扫描效率很低,需要处理大量的数据. 索引是一种特殊的数据结构,将一小块数据集合保存为容易遍历的形式.索引能够存储某种特殊字段或字段集的值,并按照索引指定的方式将字段值进行排序。

## 20 如何添加索引

使用db.collection.createIndex()在集合中创建一个索引

## 21.如何查询集合中的文档

db.collectionName.find({key:value})

## 22用什么方法可以格式化输出结果

db.collectionName.find().pretty()

## 23 如何使用"AND"或"OR"条件循环查询集合中的文档

db.mycol.find( { $or: [ {key1: value1}, {key2:value2} ] } ).pretty()

## 24 更新数据

db.collectionName.update({key:value},{$set:{newkey:newValue}})

## 25 如何删除文档

db.collectionName.remove({key:value})

## 26 在MongoDB中如何排序

并使用 1 和 -1 来指定排序方式，其中 1 表示升序，而 -1 表示降序。

db.connectionName.find({key:value}).sort({columnName:1})

## 27 什么是聚合

聚合操作能够处理数据记录并返回计算结果。聚合操作能将多个文档中的值组合起来，对成组数据执行各种操作，返回单一的结果。它相当于 SQL 中的 count(\*) 组合 group by。对于 MongoDB 中的聚合操作，应该使用aggregate()方法。

db.COLLECTION\_NAME.aggregate(AGGREGATE\_OPERATION)

## 28 在MongoDB中什么是副本集（避免单点故障）

在MongoDB中副本集由一组MongoDB实例组成，包括一个主节点多个次节点，MongoDB客户端的所有数据都写入主节点(Primary),副节点从主节点同步写入数据，以保持所有复制集内存储相同的数据，提高数据可用性。

## 29 什么是NoSQL数据库？NoSQL和RDBMS有什么区别？在哪些情况下使用和不使用NoSQL数据库？

NoSQL是非关系型数据库，NoSQL = Not Only SQL。

关系型数据库采用的结构化的数据，NoSQL采用的是键值对的方式存储数据。

在处理非结构化/半结构化的大数据时；在水平方向上进行扩展时；随时应对动态增加的数据项时可以优先考虑使用NoSQL数据库。 在考虑数据库的成熟度；支持；分析和商业智能；管理及专业性等问题时，应优先考虑关系型数据库。

## 30 MongoDB支持存储过程吗？如果支持的话，怎么用？

MongoDB支持存储过程，它是javascript写的，保存在db.system.js表中。

## 31如何理解MongoDB中的GridFS机制，MongoDB为何使用GridFS来存储文件？

Grid：网格，fs：file server文件服务

GridFS是一种将大型文件存储在MongoDB中的文件规范。使用GridFS可以将大文件分隔成多个小文档存放，这样我们能够有效的保存大文档，而且解决了BSON对象有限制的问题。

## 32 为什么MongoDB的数据文件很大？

MongoDB采用的预分配空间的方式来防止文件碎片。

## 33 当更新一个正在被迁移的块（Chunk）上的文档时会发生什么？

更新操作会立即发生在旧的块（Chunk）上，然后更改才会在所有权转移前复制到新的分片上。

## 34 MongoDB在A:{B,C}上建立索引，查询A:{B,C}和A:{C,B}都会使用索引吗？

不会，只会在A:{B,C}上使用索引。

## 35 mongodb成为最好nosql数据库的原因是什么?

面向文件的 高性能 高可用性 易扩展性 丰富的查询语言

## 36 如果用户移除对象的属性,该属性是否从存储层中删除?

是的,用户移除属性然后对象会重新保存(re-save()).

## 37 允许空值null吗?

对于对象成员而言,是的.然而用户不能够添加空值(null)到数据库丛集(collection)因为空值不是对象.然而用户能够添加空对象{}.

## 38 更新操作立刻fsync到磁盘?

不会,磁盘写操作默认是延迟执行的，短时间内会批量操作。写操作可能在两三秒(默认在60秒内)后到达磁盘.例如,如果一秒内数据库收到一千个对一个对象递增的操作,仅刷新磁盘一次.

## ~~39 如何执行事务/加锁?~~

~~mongodb没有使用传统的锁或者复杂的带回滚的事务,因为它设计的宗旨是轻量,快速以及可预计的高性能.可以把它类比成mysql mylsam的自动提交模式.通过精简对事务的支持,性能得到了提升,特别是在一个可能会穿过多个服务器的系统里.~~

## 40 启用备份故障恢复需要多久?

从备份数据库声明主数据库宕机到选出一个备份数据库作为新的主数据库将花费10到30秒时间.这期间在主数据库上的操作将会失败–包括写入和强一致性读取(strong consistent read)操作.然而,你还能在第二数据库上执行最终一致性查询(eventually consistent query)(在slaveok模式下),即使在这段时间里.

## 41 什么是master或primary?

它是当前备份集群(replica set)中负责处理所有写入操作的主要节点/成员.在一个备份集群中,当失效备援(failover)事件发生时,一个另外的成员会变成primary.

## 42 我应该启动一个集群分片(sharded)还是一个非集群分片的 mongodb 环境?

(数据量大用集群分片,数据量小用非集群)

为开发便捷起见,我们建议以非集群分片(unsharded)方式开始一个 mongodb 环境,除非一台服务器不足以存放你的初始数据集.从非集群分片升级到集群分片(sharding)是无缝的,所以在你的数据集还不是很大的时候没必要考虑集群分片(sharding).

## 43 分片(sharding)和复制(replication)是怎样工作的?

每一个分片(shard)是一个分区数据的逻辑集合.分片可能由单一服务器或者集群组成,我们推荐为每一个分片(shard)使用集群.

## 44数据在什么时候才会扩展到多个分片(shard)里?

mongodb 分片是基于区域(range)的.所以一个集合(collection)中的所有的对象都被存放到一个块(chunk)中.只有当存在多于一个块的时候,才会有多个分片获取数据的选项.现在,每个默认块的大小是 64mb,所以你需要至少 64 mb 空间才可以实施一个迁移.

## 45 如果在一个分片(shard)停止或者很慢的时候,我发起一个查询会怎样?

如果一个分片(shard)停止了,除非查询设置了“partial”选项,否则查询会返回一个错误.如果一个分片(shard)响应很慢,mongodb则会等待它的响应.

## 46 可以把movechunk目录里的旧文件删除吗?

没问题,这些文件是在分片(shard)进行均衡操作(balancing)的时候产生的临时文件.一旦这些操作已经完成,相关的临时文件也应该被删除掉.但目前清理工作是需要手动的,所以请小心地考虑再释放这些文件的空间.

## 47 如果块移动操作(movechunk)失败了,我需要手动清除部分转移的文档吗?

不需要,移动操作是一致(consistent)并且是确定性的(deterministic);一次失败后,移动操作会不断重试;当完成后,数据只会出现在新的分片里(shard).

## 48 mongodb是否支持事务

旧版本MongoDB 是不支持事务的，因此开发者在需要用到事务的时候，不得不借用其他工具，在业务代码层面去弥补数据库的不足。

MongoDB 4.0 引入的事务功能，支持多文档ACID特性，例如使用 mongo shell 进行事务操作

> s = db.getMongo().startSession()

session { "id" : UUID("3bf55e90-5e88-44aa-a59e-a30f777f1d89") }

> s.startTransaction()

> session.getDatabase("mytest").coll01.insert({x: 1, y: 1})

WriteResult({ "nInserted" : 1 })

> session.getDatabase("mytest").coll02.insert({x: 1, y: 1})

WriteResult({ "nInserted" : 1 })

> s.commitTransaction() （或者 s.abortTransaction()回滚事务）

支持 MongoDB 4.0 的其他语言 Driver 也封装了事务相关接口，用户需要创建一个 Session，然后在 Session 上开启事务，提交事务。

事务和会话(Sessions)关联，一个会话同一时刻只能开启一个事务操作，当一个会话断开，这个会话中的事务也会结束。

# ElasticSearch常见经典知识点

## 1.为什么要使用Elasticsearch?

​因为在我们商城中的数据，将来会非常多，所以采用以往的模糊查询，模糊查询前置配置，会放弃索引，导致商品查询是全表扫面，在百万级别的数据库中，效率非常低下，而我们使用ES做一个全文索引，我们将经常查询的商品的某些字段，比如说商品名，描述、价格还有id这些字段我们放入我们索引库里，可以提高查询速度。

## 2.Elasticsearch是如何实现Master选举的？

Elasticsearch的选主是ZenDiscovery模块负责的，主要包含Ping（节点之间通过这个RPC来发现彼此）和Unicast（单播模块包含一个主机列表以控制哪些节点需要ping通）这两部分；

对所有可以成为master的节点（node.master: true）根据nodeId字典排序，每次选举每个节点都把自己所知道节点排一次序，然后选出第一个（第0位）节点，暂且认为它是master节点。

如果对某个节点的投票数达到一定的值（可以成为master节点数n/2+1）并且该节点自己也选举自己，那这个节点就是master。否则重新选举一直到满足上述条件。

补充：master节点的职责主要包括集群、节点和索引的管理，不负责文档级别的管理；data节点可以关闭http功能。

## 3.Elasticsearch中的节点（比如共20个），其中的10个选了一个master，另外10个选了另一个master，怎么办？

当集群master候选数量不小于3个时，可以通过设置最少投票通过数量（discovery.zen.minimum\_master\_nodes）超过所有候选节点一半以上来解决脑裂问题；

当候选数量为两个时，只能修改为唯一的一个master候选，其他作为data节点，避免脑裂问题。

## 4.详细描述一下Elasticsearch索引文档的过程。

协调节点默认使用文档ID参与计算（也支持通过routing），以便为路由提供合适的分片。

shard = hash(document\_id) % (num\_of\_primary\_shards)

当分片所在的节点接收到来自协调节点的请求后，会将请求写入到Memory Buffer，然后定时（默认是每隔1秒）写入到Filesystem Cache，这个从Momery Buffer到Filesystem Cache的过程就叫做refresh；

当然在某些情况下，存在Momery Buffer和Filesystem Cache的数据可能会丢失，ES是通过translog的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到translog中，当Filesystem cache中的数据写入到磁盘中时，才会清除掉，这个过程叫做flush；

在flush过程中，内存中的缓冲将被清除，内容被写入一个新段，段的fsync将创建一个新的提交点，并将内容刷新到磁盘，旧的translog将被删除并开始一个新的translog。

flush触发的时机是定时触发（默认30分钟）或者translog变得太大（默认为512M）时；

## 5.详细描述一下Elasticsearch更新和删除文档的过程

删除和更新也都是写操作，但是Elasticsearch中的文档是不可变的，因此不能被删除或者改动以展示其变更；

磁盘上的每个段都有一个相应的.del文件。当删除请求发送后，文档并没有真的被删除，而是在.del文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并时，在.del文件中被标记为删除的文档将不会被写入新段。

在新的文档被创建时，Elasticsearch会为该文档指定一个版本号，当执行更新时，旧版本的文档在.del文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是会在结果中被过滤掉。

## 6.详细描述一下Elasticsearch搜索的过程

搜索被执行成一个两阶段过程，我们称之为 Query Then Fetch；

在初始查询阶段时，查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。 每个分片在本地执行搜索并构建一个匹配文档的大小为 from + size 的优先队列。PS：在搜索的时候是会查询Filesystem Cache的，但是有部分数据还在Memory Buffer，所以搜索是近实时的。

每个分片返回各自优先队列中 所有文档的 ID 和排序值 给协调节点，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。

接下来就是 取回阶段，协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分片加载并 丰富 文档，如果有需要的话，接着返回文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。

补充：Query Then Fetch的搜索类型在文档相关性打分的时候参考的是本分片的数据，这样在文档数量较少的时候可能不够准确，DFS Query Then Fetch增加了一个预查询的处理，询问Term和Document frequency，这个评分更准确，但是性能会变差。

7.Elasticsearch对于大数据量（上亿量级）的聚合如何实现？

​Elasticsearch 提供的首个近似聚合是cardinality 度量。它提供一个字段的基数，即该字段的distinct或者unique值的数目。它是基于HLL算法的。HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。其特点是：可配置的精度，用来控制内存的使用（更精确 ＝ 更多内存）；小的数据集精度是非常高的；我们可以通过配置参数，来设置去重需要的固定内存使用量。无论数千还是数十亿的唯一值，内存使用量只与你配置的精确度相关 .

## 8.在并发情况下，Elasticsearch如果保证读写一致？

可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；

另外对于写操作，一致性级别支持quorum/one/all，默认为quorum，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。

对于读操作，可以设置replication为sync(默认)，这使得操作在主分片和副本分片都完成后才会返回；如果设置replication为async时，也可以通过设置搜索请求参数\_preference为primary来查询主分片，确保文档是最新版本。

## 9.ElasticSearch中的集群、节点、索引、文档、类型是什么？

群集是一个或多个节点（服务器）的集合，它们共同保存您的整个数据，并提供跨所有节点的联合索引和搜索功能。群集由唯一名称标识，默认情况下为“elasticsearch”。此名称很重要，因为如果节点设置为按名称加入群集，则该节点只能是群集的一部分。

节点是属于集群一部分的单个服务器。它存储数据并参与群集索引和搜索功能。

索引就像关系数据库中的“数据库”。它有一个定义多种类型的映射。索引是逻辑名称空间，映射到一个或多个主分片，并且可以有零个或多个副本分片。

MySQL =>数据库 ElasticSearch =>索引

文档类似于关系数据库中的一行。不同之处在于索引中的每个文档可以具有不同的结构（字段），但是对于通用字段应该具有相同的数据类型。

MySQL => Databases => Tables => Columns / Rows ElasticSearch => Indices => Types =>具有属性的文档

类型是索引的逻辑类别/分区，其语义完全取决于用户。

## 10.ElasticSearch中的分片是什么?

在大多数环境中，每个节点都在单独的盒子或虚拟机上运行。

索引 - 在Elasticsearch中，索引是文档的集合。

分片 -因为Elasticsearch是一个分布式搜索引擎，所以索引通常被分割成分布在多个节点上的被称为分片的元素。

## 11.什么是ElasticSearch？

Elasticsearch是一个基于Lucene的搜索引擎。它提供了具有HTTP Web界面和无架构JSON文档的分布式，多租户能力的全文搜索引擎。Elasticsearch是用Java开发的，根据Apache许可条款作为开源发布。

## 12.Elasticsearch中的倒排索引是什么？

倒排索引是搜索引擎的核心。搜索引擎的主要目标是在查找发生搜索条件的文档时提供快速搜索。倒排索引是一种像数据结构一样的散列图，可将用户从单词导向文档或网页。它是搜索引擎的核心。其主要目标是快速搜索从数百万文件中查找数据。

## 13.ElasticSearch是否有架构？

ElasticSearch可以有一个架构。架构是描述文档类型以及如何处理文档的不同字段的一个或多个字段的描述。Elasticsearch中的架构是一种映射，它描述了JSON文档中的字段及其数据类型，以及它们应该如何在Lucene索引中进行索引。因此，在Elasticsearch术语中，我们通常将此模式称为“映射”。

Elasticsearch具有架构灵活的能力，这意味着可以在不明确提供架构的情况下索引文档。如果未指定映射，则默认情况下，Elasticsearch会在索引期间检测文档中的新字段时动态生成一个映射。

## 14..ElasticSearch中的副本是什么？

一个索引被分解成碎片以便于分发和扩展。副本是分片的副本。一个节点是一个属于一个集群的ElasticSearch的运行实例。一个集群由一个或多个共享相同集群名称的节点组成。

## 15.ElasticSearch中的分析器是什么？

在ElasticSearch中索引数据时，数据由为索引定义的Analyzer在内部进行转换。 分析器由一个Tokenizer和零个或多个TokenFilter组成。编译器可以在一个或多个CharFilter之前。分析模块允许您在逻辑名称下注册分析器，然后可以在映射定义或某些API中引用它们。

Elasticsearch附带了许多可以随时使用的预建分析器。或者，您可以组合内置的字符过滤器，编译器和过滤器器来创建自定义分析器。

## 16.什么是ElasticSearch中的编译器？

编译器用于将字符串分解为术语或标记流。一个简单的编译器可能会将字符串拆分为任何遇到空格或标点的地方。Elasticsearch有许多内置标记器，可用于构建自定义分析器。

## 17.启用属性，索引和存储的用途是什么？

enabled属性适用于各类ElasticSearch特定/创建领域，如index和size。用户提供的字段没有“已启用”属性。 存储意味着数据由Lucene存储，如果询问，将返回这些数据。

存储字段不一定是可搜索的。默认情况下，字段不存储，但源文件是完整的。因为您希望使用默认值(这是有意义的)，所以不要设置store属性 该指数属性用于搜索。

索引属性只能用于搜索。只有索引域可以进行搜索。差异的原因是在分析期间对索引字段进行了转换，因此如果需要的话，您不能检索原始数据。

# 50道Python知识点集锦（附答案）

Python是目前编程领域最受欢迎的语言。在本文中，我将总结Python面试中最常见的50个问题。每道题都提供参考答案，希望能够帮助你在2019年求职面试中脱颖而出，找到一份高薪工作。这些知识点涉及Python基础知识、Python编程、数据分析以及Python函数库等多个方面。

## Q1、Python中的列表和元组有什么区别？

列表(list)和元组(tuple)的一些基础

#### list和tuple都是一个可以放置任意数据类型的有序集合，都是既可以存放数字、字符串、对象等

#### list和tuple都支持负索引

In [8]: nums[-2]

Out[8]: 'ad'

In [9]: tp[-2]

Out[9]: '33'

#### list和tuple都支持切片操作

In [10]: nums[1:3]

Out[10]: [3, 'ad']

In [11]: tp[1:3]

Out[11]: (3, '33')

#### list和tuple都可以随意嵌套

In [12]: nums = [[1,2,3],['s','ff'],['34',3,5]]

In [13]: tp = ((23,4,'f'),45,'d',('dd',4,'ff'))

### list和tuple的一些区别

#### 列表是动态的，长度大小不固定，可以随意的增加、删除、修改元素

元组是静态的，长度在初始化的时候就已经确定不能更改，更无法增加、删除、修改元素

从图中看出我们对list做出修改是成功的，但是对tuple修改的时候，确抛出了错误，那如果想对tuple做出改变该如何做呢？只能重新开辟一块内存，重新生成新到的tuple了。

从tuple的源码中也可以看出，只有两个自带的方法，一个是统计元素出现的次数一个是查询元素的索引。

list和tuple存储方式的差异

我们先来看个例子

In [19]: nums=['a',1,2]

In [20]: tp=('a',1,2)

In [21]: nums.\_\_sizeof\_\_()

Out[21]: 64

In [22]: tp.\_\_sizeof\_\_()

Out[22]: 48

这里构造了一个list和一个tuple。他们存储的内容是相同的，\_\_sizeof\_\_方法可以打印系统分配空间的大小。可以看到他们所占用的内存空间是不同的，存储的内容相同，但是list比tuple多占用了16自己的内存。

先来看一下一个数组的内存分配过程：

In [23]: l=[]

In [24]: l.\_\_sizeof\_\_() // 空列表分配了40字节的内存

Out[24]: 40

In [25]: l.append('a') // 增加了一个一个元素后，给列表分配了72字节的内存，一个字符8个字节

// 那就是一次性分配了4个字符的内存空间

In [26]: l.\_\_sizeof\_\_()

Out[26]: 72

In [27]: l.append('b') // 再增加字符，占用内存不变

In [28]: l.\_\_sizeof\_\_()

Out[28]: 72

In [29]: l.append('c') // 再增加字符，占用内存不变

In [30]: l.\_\_sizeof\_\_()

Out[30]: 72

In [31]: l.append('d') // 再增加字符，占用内存不变

In [32]: l.\_\_sizeof\_\_()

Out[32]: 72

In [33]: l.append('e') // 再添加元素，内存不够，触发重新的内存分配

In [34]: l.\_\_sizeof\_\_()

Out[34]: 104

可以看出list为了能够实时追踪内存的使用情况，当空间不足时以及分配额外空间，额外的多分配了内存，而且还需要存储指针，指向对应的元素。

我们可以看到，为了减小每次增加 / 删减操作时空间分配的开销，Python 每次分配空间时都会额外多分配一些，这样的机制（over-allocating）保证了其操作的高效性：增加 / 删除的时间复杂度均为 O(1)。但是对于元组，情况就不同了。元组长度大小固定，元素不可变，所以存储空间固定。

所以说在存储大量的数据的时候，这种差异是需要考虑的，如果数据发生变更的可能性不大，就用元组存储，如果数据是需要频繁的进行数据的修改增加，就使用列表

## Q2、Python的主要功能是什么？

Python是一种解释型语言。与C语言等语言不同，Python不需要在运行之前进行编译。

Python是动态语言，当您声明变量或类似变量时，您不需要声明变量的类型。

Python适合面向对象的编程，因为它允许类的定义以及组合和继承。Python没有访问说明（如C ++的public，private）。

在Python中，函数是第一类对象。它们可以分配给变量。类也是第一类对象

编写Python代码很快，但运行比较慢。Python允许基于C的扩展，例如numpy函数库。

Python可用于许多领域。Web应用程序开发，自动化，数学建模，大数据应用程序等等。它也经常被用作“胶水”代码。

## Q3、Python是通用编程语言吗？

Python能够编写脚本，但从一般意义上讲，它被认为是一种通用编程语言。

## Q4、Python是如何解释语言的？

Python在运行之前不需要对程序进行解释。因此，Python是一种解释型语言。

## Q5、什么是pep？

PEP代表Python Enhancement Proposal。它是一组规则，指定如何格式化Python代码以获得最大可读性。

## Q6、如何在Python中管理内存？

python中的内存管理由Python私有堆空间管理。所有Python对象和数据结构都位于私有堆中。程序员无权访问此私有堆。python解释器负责处理这个问题。

Python对象的堆空间分配由Python的内存管理器完成。核心API提供了一些程序员编写代码的工具。

Python还有一个内置的垃圾收集器，它可以回收所有未使用的内存，并使其可用于堆空间。

## Q7、Python中的命名空间是什么？

命名空间是一个命名系统，用于确保名称是唯一性，以避免命名冲突。

## Q8、什么是PYTHONPATH？

它是导入模块时使用的环境变量。每当导入模块时，也会查找PYTHONPATH以检查各个目录中是否存在导入的模块。解释器使用它来确定要加载的模块。

## Q9、什么是python模块？Python中有哪些常用的内置模块？

Python模块是包含Python代码的.py文件。此代码可以是函数类或变量。一些常用的内置模块包括：sys、math、random、data time、JSON。

## Q10、Python中的局部变量和全局变量是什么？

全局变量：在函数外或全局空间中声明的变量称为全局变量。这些变量可以由程序中的任何函数访问。

局部变量：在函数内声明的任何变量都称为局部变量。此变量存在于局部空间中，而不是全局空间中。

## Q11、python是否区分大小写？

是。Python是一种区分大小写的语言。

## Q12、什么是Python中的类型转换？

类型转换是指将一种数据类型转换为另一种数据类型。

int（）  - 将任何数据类型转换为整数类型

float（）  - 将任何数据类型转换为float类型

ord（）  - 将字符转换为整数

hex（） - 将整数转换为十六进制

oct（）  - 将整数转换为八进制

tuple（） - 此函数用于转换为元组。

set（） - 此函数在转换为set后返回类型。

list（） - 此函数用于将任何数据类型转换为列表类型。

dict（） - 此函数用于将顺序元组（键，值）转换为字典。

str（） - 用于将整数转换为字符串。

complex（real，imag）  - 此函数将实数转换为复数（实数，图像）数。

## Q13、如何在Windows上安装Python并设置路径变量？

要在Windows上安装Python，请按照以下步骤操作：

从以下链接安装python：https：//http://www.python.org/downloads/

下载之后，将其安装在您的PC上。在命令提示符下使用以下命令查找PC上安装PYTHON的位置：cmd python。

然后转到高级系统设置并添加新变量并将其命名为PYTHON\_NAME并粘贴复制的路径。

查找路径变量，选择其值并选择“编辑”。

如果值不存在，请在值的末尾添加分号，然后键入％PYTHON\_HOME％

## Q14、python中是否需要缩进？

缩进是Python必需的。它指定了一个代码块。循环，类，函数等中的所有代码都在缩进块中指定。通常使用四个空格字符来完成。如果您的代码没有必要缩进，它将无法准确执行并且也会抛出错误。

## Q15、Python数组和列表有什么区别？

Python中的数组和列表具有相同的存储数据方式。但是，数组只能包含单个数据类型元素，而列表可以包含任何数据类型元素。

## Q16、Python中的函数是什么？

函数是一个代码块，只有在被调用时才会执行。要在Python中定义函数，需要使用def关键字。

## Q17、什么是\_\_init\_\_?

\_\_init\_\_是Python中的方法或者结构。在创建类的新对象/实例时，将自动调用此方法来分配内存。所有类都有\_\_init\_\_方法。

## Q18、什么是lambda函数？

lambda函数也叫匿名函数，该函数可以包含任意数量的参数，但只能有一个执行操作的语句。

## Q19、Python中的self是什么？

self是类的实例或对象。在Python中，self包含在第一个参数中。但是，Java中的情况并非如此，它是可选的。它有助于区分具有局部变量的类的方法和属性。init方法中的self变量引用新创建的对象，而在其他方法中，它引用其方法被调用的对象。

## Q20、区分break，continue和pass？

## Q21、[:: - 1}表示什么？

[:: - 1]用于反转数组或序列的顺序。

## Q22、如何在Python中随机化列表中的元素？

可以使用shuffle函数进行随机列表元素。举例如下：

代码输出为：

## Q23、什么是python迭代器？

迭代器是可以遍历或迭代的对象。

## Q24、如何在Python中生成随机数？

random模块是用于生成随机数的标准模块。该方法定义为：

random.random()方法返回[0,1]范围内的浮点数。该函数生成随机浮点数。随机类使用的方法是隐藏实例的绑定方法。可以使用Random的实例来显示创建不同线程实例的多线程程序。其中使用的其他随机生成器是：

randrange(a,b)：它选择一个整数并定义[a，b]之间的范围。它通过从指定范围中随机选择元素来返回元素。它不构建范围对象。

uniform(a,b)：它选择一个在[a，b)范围内定义的浮点数

normalvariate(mean,sdev)：它用于正态分布，其中mean是平均值，sdev是用于标准偏差的sigma。

使用和实例化的Random类创建一个独立的多个随机数生成器。

## Q25、range＆xrange有什么区别？

在大多数情况下，xrange和range在功能方面完全相同。它们都提供了一种生成整数列表的方法，唯一的区别是range返回一个Python列表对象，xrange返回一个xrange对象。这就表示xrange实际上在运行时并不是生成静态列表。它使用称为yielding的特殊技术根据需要创建值。该技术与一种称为生成器的对象一起使用。因此如果你有一个非常巨大的列表，那么就要考虑xrange。

## Python3 迭代器与生成器

### 迭代器

迭代是Python最强大的功能之一，是访问集合元素的一种方式。

迭代器是一个可以记住遍历位置的对象。

迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：iter() 和 next()。

字符串，列表或元组对象都可用于创建迭代器：

实例(Python 3.0+)

>>> list=[1,2,3,4]

>>> it = iter(list) # 创建迭代器对象

>>> print (next(it)) # 输出迭代器的下一个元素

1

>>> print (next(it))

2

>>>

迭代器对象可以使用常规for语句进行遍历：

实例(Python 3.0+)

#!/usr/bin/python3

list=[1,2,3,4]

it = iter(list) # 创建迭代器对象

for x in it:

print (x, end=" ")

执行以上程序，输出结果如下：

1 2 3 4

创建一个迭代器

把一个类作为一个迭代器使用需要在类中实现两个方法 \_\_iter\_\_() 与 \_\_next\_\_() 。

如果你已经了解的面向对象编程，就知道类都有一个构造函数，Python 的构造函数为 \_\_init\_\_(), 它会在对象初始化的时候执行。

更多内容查阅：Python3 面向对象

\_\_iter\_\_() 方法返回一个特殊的迭代器对象， 这个迭代器对象实现了 \_\_next\_\_() 方法并通过 StopIteration 异常标识迭代的完成。

\_\_next\_\_() 方法（Python 2 里是 next()）会返回下一个迭代器对象。

创建一个返回数字的迭代器，初始值为 1，逐步递增 1：

实例(Python 3.0+)

class MyNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

x = self.a

self.a += 1

return x

myclass = MyNumbers()

myiter = iter(myclass)

print(next(myiter))

print(next(myiter))

print(next(myiter))

print(next(myiter))

print(next(myiter))

执行输出结果为：

1

2

3

4

5

StopIteration

StopIteration 异常用于标识迭代的完成，防止出现无限循环的情况，在 \_\_next\_\_() 方法中我们可以设置在完成指定循环次数后触发 StopIteration 异常来结束迭代。

在 20 次迭代后停止执行：

实例(Python 3.0+)

class MyNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

if self.a <= 20:

x = self.a

self.a += 1

return x

else:

raise StopIteration

myclass = MyNumbers()

myiter = iter(myclass)

for x in myiter:

print(x)

执行输出结果为：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

### 生成器

在 Python 中，使用了 yield 的函数被称为生成器（generator）。

跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。

在调用生成器运行的过程中，每次遇到 yield 时函数会暂停并保存当前所有的运行信息，返回 yield 的值, 并在下一次执行 next() 方法时从当前位置继续运行。

调用一个生成器函数，返回的是一个迭代器对象。

以下实例使用 yield 实现斐波那契数列：

实例(Python 3.0+)

#!/usr/bin/python3

import sys

def fibonacci(n): # 生成器函数 - 斐波那契

a, b, counter = 0, 1, 0

while True:

if (counter > n):

return

yield a

a, b = b, a + b

counter += 1

f = fibonacci(10) # f 是一个迭代器，由生成器返回生成

while True:

try:

print (next(f), end=" ")

except StopIteration:

sys.exit()

执行以上程序，输出结果如下：

0 1 1 2 3 5 8 13 21 34 55

## Q26、如何在python中写注释？

Python中的注释以＃字符开头。也可以使用doc-strings（三重引号中包含的字符串）进行注释。

## Q27、什么是pickling和unpickling？

Pickle模块接受任何Python对象并将其转换为字符串表示形式，并使用dump函数将其转储到文件中，此过程称为pickling。从存储的字符串中检索原始Python对象的过程称为unpickling。

## Q28、python中的生成器是什么？

返回可迭代项集的函数称为生成器。

## Q29、你如何把字符串的第一个字母大写？

在Python中，capitalize()函数可以将字符串的第一个字母大写。如果字符串在开头已经包含大写字母，那么它将返回原始字符串。

## Q30、如何将字符串转换为全小写？

要将字符串转换为小写，可以使用lower()函数。

## Q31、如何在python中注释多行？

注释多行代码时。所有要注释的行都要在开头前加#。还可以使用快捷方式来注释多行，就是按住Ctrl键并在每个想要包含＃字符的地方左键单击并键入一次＃。

## Q32、什么是Python中的文档Docstrings？

Docstrings实际上不是注释，它们是文档字符串。这些文档字符串在三引号内。它们没有分配给任何变量，因此有时也用于注释。

## Q33、operators中的is、not和in各有什么功能？

Operators是特殊函数，它们比较一个或多个值并产生相应的结果。其中is：当2个操作数为true时返回true（例如：“a”是'a'）

not：返回布尔值的倒数

in：检查某个元素是否存在于某个序列中

## Q34、Python中help()和dir()函数的用法是什么？

Help()和dir()这两个函数都可以从Python解释器直接访问，并用于查看内置函数的合并转储。

help()函数：help()函数用于显示文档字符串，还可以查看与模块，关键字，属性等相关的使用信息。

dir()函数：dir()函数用于显示定义的符号。

## Q35、当Python退出时，为什么不清除所有分配的内存？

当Python退出时，尤其是那些对其他对象具有循环引用的Python模块或者从全局名称空间引用的对象并没有被解除分配或释放。

无法解除分配C库保留的那些内存部分。

退出时，由于拥有自己的高效清理机制，Python会尝试取消分配/销毁其他所有对象。

## Q36、Python中的字典是什么？

Python中的内置数据类型称为字典。它定义了键和值之间的一对一关系。字典包含一对键及其对应的值。字典由键索引。

## Q37、如何在python中使用三元运算符？

三元运算符是用于显示条件语句的运算符。这包含true或false值，并且必须为其评估语句。其基本语法为：

三元运算符是用于显示条件语句的运算符。这包含true或false值，并且必须为其评估语句。其基本语法为：

[on\_true] if [expression] else [on\_false] x，y = 25,50big = x if x <y else y

## Q38、为什么使用\* args，\*\* kwargs？

当我们不确定将多少个参数传递给函数，或者我们想要将存储的列表或参数元组传递给函数时，我们使用\* args。\*\*当我们不知道将多少关键字参数传递给函数时使用kwargs，或者它可以用于将字典的值作为关键字参数传递。标识符args和kwargs是一个约定，你也可以使用\* bob和\*\* billy。

## Q39、len()函数有什么作用？

len()函数可用于确定字符串，列表，数组等的长度。

## Q40、在Python中split()，sub()，subn()功能。

如果要修改字符串，Python的“re”模块提供了3种方法。他们是：

split() - 使用正则表达式模式将给定字符串“拆分”到列表中。

sub() - 查找正则表达式模式匹配的所有子字符串，然后用不同的字符串替换它们

subn() - 它类似于sub()，并且还返回新字符串。

## Q41、什么是负指数，功能是什么？

Python中的序列是索引的，它由正数和负数组成。积极的数字使用'0'作为第一个索引，'1'作为第二个索引，进程继续使用。

负数的索引从'-1'开始，表示序列中的最后一个索引，' - 2'作为倒数第二个索引，序列像正数一样前进。

负索引用于从字符串中删除任何换行符，并允许该字符串除了作为S [： - 1]给出的最后一个字符。负索引还用于显示索引以正确的顺序表示字符串。

## Q42、什么是Python包？

Python包是包含多个模块的命名空间。

## Q43、如何在Python中删除文件？

要在Python中删除文件，您需要导入OS模块。之后，您需要使用os.remove()函数。

## Q44、什么是python的内置类型？

Python中的内置类型如下：整型、浮点型、复数、字符串、布尔等。

## Q45、NumPy中有哪些操作Python列表的函数？

Python的列表是高效的通用容器。它们支持（相当）有效的插入，删除，追加和连接，Python的列表推导使它们易于构造和操作。

它们有一定的局限性：它们不支持像素化加法和乘法等“向量化”操作，并且它们可以包含不同类型的对象这一事实意味着Python必须存储每个元素的类型信息，并且必须执行类型调度代码在对每个元素进行操作时。

NumPy不仅效率更高; 它也更方便。你可以免费获得大量的向量和矩阵运算，这有时可以避免不必要的工作。它们也得到有效实施。

NumPy数组更快，你可以使用NumPy，FFT，卷积，快速搜索，基本统计，线性代数，直方图等内置。

## Q46、如何将值添加到python列表？

可以使用append()，extend()和insert(i，x)函数将元素添加到数组中。

## Q47、如何删除python列表的值？

可以使用pop()或remove()方法删除数组元素。这两个函数之间的区别在于前者返回已删除的值，而后者则不返回。

## Q48、Python有OOps概念吗？

Python是一种面向对象的编程语言。这意味着可以通过创建对象模型在python中解决任何程序。同时Python可以被视为程序语言和结构语言。

## Q49、深拷贝和浅拷贝有什么区别？

在创建新实例类型时使用浅拷贝，并保留在新实例中复制的值。浅拷贝用于复制引用指针，就像复制值一样。这些引用指向原始对象，并且在类的任何成员中所做的更改也将影响它的原始副本。浅拷贝允许更快地执行程序，它取决于所使用的数据的大小。

深拷贝用于存储已复制的值。深拷贝不会将引用指针复制到对象。它引用一个对象，并存储一些其他对象指向的新对象。原始副本中所做的更改不会影响使用该对象的任何其他副本。由于为每个被调用的对象创建了某些副本，因此深拷贝会使程序的执行速度变慢。

## Q50、如何在Python中实现多线程？

Python有一个多线程库，但是用多线程来加速代码的效果并不是那么的好，

Python有一个名为Global Interpreter Lock（GIL）的结构。GIL确保每次只能执行一个“线程”。一个线程获取GIL执行相关操作，然后将GIL传递到下一个线程。

虽然看起来程序被多线程并行执行，但它们实际上只是轮流使用相同的CPU核心。

所有这些GIL传递都增加了执行的开销。这意味着多线程并不能让程序运行的更快。

# golang 知识点(从基础到高级)

Golang问题汇总

通常我们去面试肯定会有些不错的Golang的知识点目的，所以总结下，让其他Golang开发者也可以查看到，同时也用来检测自己的能力和提醒自己的不足之处,欢迎大家补充和提交新的知识点目.

Golang问题汇总:

## 1. Golang中除了加Mutex锁以外还有哪些方式安全读写共享变量？

Golang中Goroutine可以通过 Channel 进行安全读写共享变量。

## 2. 无缓冲 Chan 的发送和接收是否同步?

ch := make(chan int) 无缓冲的channel由于没有缓冲发送和接收需要同步.

ch := make(chan int, 2) 有缓冲channel不要求发送和接收操作同步.

channel无缓冲时，发送阻塞直到数据被接收，接收阻塞直到读到发送数据。

channel有缓冲时，当缓冲满时发送阻塞，当缓冲空时接收阻塞。

## 3. go语言的并发机制以及它所使用的CSP并发模型．

CSP模型是上个世纪七十年代提出的,不同于传统的多线程通过共享内存来通信，CSP讲究的是“以通信的方式来共享内存”。用于描述两个独立的并发实体通过共享的通讯 channel(管道)进行通信的并发模型。 CSP中channel是第一类对象，它不关注发送消息的实体，而关注与发送消息时使用的channel。

Golang中channel 是被单独创建并且可以在进程之间传递，它的通信模式类似于 boss-worker 模式的，一个实体通过将消息发送到channel 中，然后由监听这个 channel 的实体处理，两个实体之间是匿名的，这个就实现实体中间的解耦，其中 channel 是同步的一个消息被发送到 channel 中，最终是一定要被另外的实体消费掉的，在实现原理上其实类似一个阻塞的消息队列。

Goroutine 是Golang实际并发执行的实体，它底层是使用协程(coroutine)实现并发，coroutine是一种运行在用户态的用户线程，类似于 greenthread，go底层选择使用coroutine的出发点是因为，它具有以下特点：

用户空间 避免了内核态和用户态的切换导致的成本。

可以由语言和框架层进行调度。

更小的栈空间，允许创建大量的实例。

Golang中的Goroutine的特性:

Golang内部有三个对象： P对象(processor) 代表上下文（或者可以认为是cpu），M(Machine，work thread)代表工作线程，G对象（goroutine）.

正常情况下一个cpu对象启一个工作线程对象，线程去检查并执行goroutine对象。碰到goroutine对象阻塞的时候，会启动一个新的工作线程，以充分利用cpu资源。 所有有时候线程对象会比处理器对象多很多.

我们用如下图分别表示P、M、G:

G（Goroutine） ：我们所说的协程，为用户级的轻量级线程，每个Goroutine对象中的sched保存着其上下文信息.

M（Machine） ：对内核级线程的封装，数量对应真实的CPU数（真正干活的对象）.

P（Processor） ：即为G和M的调度对象，用来调度G和M之间的关联关系，其数量可通过GOMAXPROCS()来设置，默认为核心数.

在单核情况下，所有Goroutine运行在同一个线程（M0）中，每一个线程维护一个上下文（P），任何时刻，一个上下文中只有一个Goroutine，其他Goroutine在runqueue中等待。

一个Goroutine运行完自己的时间片后，让出上下文，自己回到runqueue中（如下图所示）。

当正在运行的G0阻塞的时候（可以需要IO），会再创建一个线程（M1），P转到新的线程中去运行。

当M0返回时，它会尝试从其他线程中“偷”一个上下文过来，如果没有偷到，会把Goroutine放到Global runqueue中去，然后把自己放入线程缓存中。 上下文会定时检查Global runqueue。

Golang是为并发而生的语言，Go语言是为数不多的在语言层面实现并发的语言；也正是Go语言的并发特性，吸引了全球无数的开发者。

Golang的CSP并发模型，是通过Goroutine和Channel来实现的。

Goroutine 是Go语言中并发的执行单位。有点抽象，其实就是和传统概念上的”线程“类似，可以理解为”线程“。 Channel是Go语言中各个并发结构体(Goroutine)之前的通信机制。通常Channel，是各个Goroutine之间通信的”管道“，有点类似于Linux中的管道。

通信机制channel也很方便，传数据用channel <- data，取数据用<-channel。

在通信过程中，传数据channel <- data和取数据<-channel必然会成对出现，因为这边传，那边取，两个goroutine之间才会实现通信。

而且不管传还是取，必阻塞，直到另外的goroutine传或者取为止。

## 4. Golang 中常用的并发模型？

Golang 中常用的并发模型有三种:

### 通过channel通知实现并发控制

无缓冲的通道指的是通道的大小为0，也就是说，这种类型的通道在接收前没有能力保存任何值，它要求发送 goroutine 和接收 goroutine 同时准备好，才可以完成发送和接收操作。

从上面无缓冲的通道定义来看，发送 goroutine 和接收 gouroutine 必须是同步的，同时准备后，如果没有同时准备好的话，先执行的操作就会阻塞等待，直到另一个相对应的操作准备好为止。这种无缓冲的通道我们也称之为同步通道。

func main() {

ch := make(chan struct{})

go func() {

fmt.Println("start working")

time.Sleep(time.Second \* 1)

ch <- struct{}{}

}()

<-ch

fmt.Println("finished")

}

当主 goroutine 运行到 <-ch 接受 channel 的值的时候，如果该 channel 中没有数据，就会一直阻塞等待，直到有值。 这样就可以简单实现并发控制

### 通过sync包中的WaitGroup实现并发控制

Goroutine是异步执行的，有的时候为了防止在结束mian函数的时候结束掉Goroutine，所以需要同步等待，这个时候就需要用 WaitGroup了，在 sync 包中，提供了 WaitGroup ，它会等待它收集的所有 goroutine 任务全部完成。在WaitGroup里主要有三个方法:

Add, 可以添加或减少 goroutine的数量.

Done, 相当于Add(-1).

Wait, 执行后会堵塞主线程，直到WaitGroup 里的值减至0.

在主 goroutine 中 Add(delta int) 索要等待goroutine 的数量。 在每一个 goroutine 完成后 Done() 表示这一个goroutine 已经完成，当所有的 goroutine 都完成后，在主 goroutine 中 WaitGroup 返回返回。

func main(){

var wg sync.WaitGroup

var urls = []string{

"http://www.golang.org/",

"http://www.google.com/",

}

for \_, url := range urls {

wg.Add(1)

go func(url string) {

defer wg.Done()

http.Get(url)

}(url)

}

wg.Wait()

}

在Golang官网中对于WaitGroup介绍是A WaitGroup must not be copied after first use,在 WaitGroup 第一次使用后，不能被拷贝

应用示例:

func main(){

wg := sync.WaitGroup{}

for i := 0; i < 5; i++ {

wg.Add(1)

go func(wg sync.WaitGroup, i int) {

fmt.Printf("i:%d", i)

wg.Done()

}(wg, i)

}

wg.Wait()

fmt.Println("exit")

}

运行:

i:1i:3i:2i:0i:4fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:

sync.runtime\_Semacquire(0xc000094018)

/home/keke/soft/go/src/runtime/sema.go:56 +0x39

sync.(\*WaitGroup).Wait(0xc000094010)

/home/keke/soft/go/src/sync/waitgroup.go:130 +0x64

main.main()

/home/keke/go/Test/wait.go:17 +0xab

exit status 2

它提示所有的 goroutine 都已经睡眠了，出现了死锁。这是因为 wg 给拷贝传递到了 goroutine 中，导致只有 Add 操作，其实 Done操作是在 wg 的副本执行的。

因此 Wait 就死锁了。

这个第一个修改方式:将匿名函数中 wg 的传入类型改为 \*sync.WaitGrou,这样就能引用到正确的WaitGroup了。 这个第二个修改方式:将匿名函数中的 wg 的传入参数去掉，因为Go支持闭包类型，在匿名函数中可以直接使用外面的 wg 变量

### Context上下文，实现并发控制

在Go 1.7 以后引进的，强大的

通常,在一些简单场景下使用 channel 和 WaitGroup 已经足够了，但是当面临一些复杂多变的网络并发场景下 channel 和 WaitGroup 显得有些力不从心了。 比如一个网络请求 Request，每个 Request 都需要开启一个 goroutine 做一些事情，这些 goroutine 又可能会开启其他的 goroutine，比如数据库和RPC服务。 所以我们需要一种可以跟踪 goroutine 的方案，才可以达到控制他们的目的，这就是Go语言为我们提供的 Context，称之为上下文非常贴切，它就是goroutine 的上下文。 它是包括一个程序的运行环境、现场和快照等。每个程序要运行时，都需要知道当前程序的运行状态，通常Go 将这些封装在一个 Context 里，再将它传给要执行的 goroutine 。

context 包主要是用来处理多个 goroutine 之间共享数据，及多个 goroutine 的管理。

context 包的核心是 struct Context，接口声明如下：

// A Context carries a deadline, cancelation signal, and request-scoped values

// across API boundaries. Its methods are safe for simultaneous use by multiple

// goroutines.

type Context interface {

// Done returns a channel that is closed when this `Context` is canceled

// or times out.

Done() <-chan struct{}

// Err indicates why this Context was canceled, after the Done channel

// is closed.

Err() error

// Deadline returns the time when this Context will be canceled, if any.

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

// Value returns the value associated with key or nil if none.

Value(key interface{}) interface{}

}

Done() 返回一个只能接受数据的channel类型，当该context关闭或者超时时间到了的时候，该channel就会有一个取消信号

Err() 在Done() 之后，返回context 取消的原因。

Deadline() 设置该context cancel的时间点

Value() 方法允许 Context 对象携带request作用域的数据，该数据必须是线程安全的。

Context 对象是线程安全的，你可以把一个 Context 对象传递给任意个数的 gorotuine，对它执行 取消 操作时，所有 goroutine 都会接收到取消信号。

一个 Context 不能拥有 Cancel 方法，同时我们也只能 Done channel 接收数据。 其中的原因是一致的：接收取消信号的函数和发送信号的函数通常不是一个。 典型的场景是：父操作为子操作操作启动 goroutine，子操作也就不能取消父操作。

## 5. JSON 标准库对 nil slice 和 空 slice 的处理是一致的吗？

首先JSON 标准库对 nil slice 和 空 slice 的处理是不一致.

通常错误的用法，会报数组越界的错误，因为只是声明了slice，却没有给实例化的对象。

var slice []int

slice[1] = 0

此时slice的值是nil，这种情况可以用于需要返回slice的函数，当函数出现异常的时候，保证函数依然会有nil的返回值。

empty slice 是指slice不为nil，但是slice没有值，slice的底层的空间是空的，此时的定义如下：

slice := make([]int,0）

slice := []int{}

当我们查询或者处理一个空的列表的时候，这非常有用，它会告诉我们返回的是一个列表，但是列表内没有任何值。

总之，nil slice 和 empty slice是不同的东西,需要我们加以区分的.

## 6. 协程，线程，进程的区别。

### 进程

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。每个进程都有自己的独立内存空间，不同进程通过进程间通信来通信。由于进程比较重量，占据独立的内存，所以上下文进程间的切换开销（栈、寄存器、虚拟内存、文件句柄等）比较大，但相对比较稳定安全。

### 线程

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。线程间通信主要通过共享内存，上下文切换很快，资源开销较少，但相比进程不够稳定容易丢失数据。

### 协程

协程是一种用户态的轻量级线程，协程的调度完全由用户控制。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，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

## 7. 互斥锁，读写锁，死锁问题是怎么解决。

互斥锁

互斥锁就是互斥变量mutex，用来锁住临界区的.

条件锁

就是条件变量，当进程的某些资源要求不满足时就进入休眠，也就是锁住了。当资源被分配到了，条件锁打开，进程继续运行；读写锁，也类似，用于缓冲区等临界资源能互斥访问的。

读写锁

通常有些公共数据修改的机会很少，但其读的机会很多。并且在读的过程中会伴随着查找，给这种代码加锁会降低我们的程序效率。读写锁可以解决这个问题。

注意：写独占，读共享，写锁优先级高

死锁

一般情况下，如果同一个线程先后两次调用lock，在第二次调用时，由于锁已经被占用，该线程会挂起等待别的线程释放锁，然而锁正是被自己占用着的，该线程又被挂起而没有机会释放锁，因此就永远处于挂起等待状态了，这叫做死锁（Deadlock）。 另外一种情况是：若线程A获得了锁1，线程B获得了锁2，这时线程A调用lock试图获得锁2，结果是需要挂起等待线程B释放锁2，而这时线程B也调用lock试图获得锁1，结果是需要挂起等待线程A释放锁1，于是线程A和B都永远处于挂起状态了。

### 死锁产生的四个必要条件:

1、互斥条件：一个资源每次只能被一个进程使用

2、请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

3、不剥夺条件:进程已获得的资源，在未使用完之前，不能强行剥夺。

4、循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

a. 预防死锁

可以把资源一次性分配：（破坏请求和保持条件）

然后剥夺资源：即当某进程新的资源未满足时，释放已占有的资源（破坏不可剥夺条件）

资源有序分配法：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）

b. 避免死锁

预防死锁的几种策略，会严重地损害系统性能。因此在避免死锁时，要施加较弱的限制，从而获得 较满意的系统性能。由于在避免死锁的策略中，允许进程动态地申请资源。因而，系统在进行资源分配之前预先计算资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，进程等待。其中最具有代表性的避免死锁算法是银行家算法。

c. 检测死锁

首先为每个进程和每个资源指定一个唯一的号码,然后建立资源分配表和进程等待表.

d. 解除死锁

当发现有进程死锁后，便应立即把它从死锁状态中解脱出来，常采用的方法有.

e. 剥夺资源

从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态.

f. 撤消进程

可以直接撤消死锁进程或撤消代价最小的进程，直至有足够的资源可用，死锁状态.消除为止.所谓代价是指优先级、运行代价、进程的重要性和价值等。

## 8. Golang的内存模型，为什么小对象多了会造成gc压力。

通常小对象过多会导致GC三色法消耗过多的GPU。优化思路是，减少对象分配。

## 9. Data Race问题怎么解决？能不能不加锁解决这个问题？

同步访问共享数据是处理数据竞争的一种有效的方法.golang在1.1之后引入了竞争检测机制，可以使用 go run -race 或者 go build -race来进行静态检测。 其在内部的实现是,开启多个协程执行同一个命令， 并且记录下每个变量的状态.

竞争检测器基于C/C++的ThreadSanitizer 运行时库，该库在Google内部代码基地和Chromium找到许多错误。这个技术在2012年九月集成到Go中，从那时开始，它已经在标准库中检测到42个竞争条件。现在，它已经是我们持续构建过程的一部分，当竞争条件出现时，它会继续捕捉到这些错误。

竞争检测器已经完全集成到Go工具链中，仅仅添加-race标志到命令行就使用了检测器。

$ go test -race mypkg // 测试包

$ go run -race mysrc.go // 编译和运行程序

$ go build -race mycmd // 构建程序

$ go install -race mypkg // 安装程序

要想解决数据竞争的问题可以使用互斥锁sync.Mutex,解决数据竞争(Data race),也可以使用管道解决,使用管道的效率要比互斥锁高.

## 10. 什么是channel，为什么它可以做到线程安全？

Channel是Go中的一个核心类型，可以把它看成一个管道，通过它并发核心单元就可以发送或者接收数据进行通讯(communication),Channel也可以理解是一个先进先出的队列，通过管道进行通信。

Golang的Channel,发送一个数据到Channel 和 从Channel接收一个数据 都是 原子性的。而且Go的设计思想就是:不要通过共享内存来通信，而是通过通信来共享内存，前者就是传统的加锁，后者就是Channel。也就是说，设计Channel的主要目的就是在多任务间传递数据的，这当然是安全的。

## 11. Epoll原理.

开发高性能网络程序时，windows开发者们言必称Iocp，linux开发者们则言必称Epoll。大家都明白Epoll是一种IO多路复用技术，可以非常高效的处理数以百万计的Socket句柄，比起以前的Select和Poll效率提高了很多。

先简单了解下如何使用C库封装的3个epoll系统调用。

int epoll\_create(int size);

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

int epoll\_wait(int epfd, struct epoll\_event \*events,int maxevents, int timeout);

使用起来很清晰，首先要调用epoll\_create建立一个epoll对象。参数size是内核保证能够正确处理的最大句柄数，多于这个最大数时内核可不保证效果。 epoll\_ctl可以操作上面建立的epoll，例如，将刚建立的socket加入到epoll中让其监控，或者把 epoll正在监控的某个socket句柄移出epoll，不再监控它等等。

epoll\_wait在调用时，在给定的timeout时间内，当在监控的所有句柄中有事件发生时，就返回用户态的进程。

从调用方式就可以看到epoll相比select/poll的优越之处是,因为后者每次调用时都要传递你所要监控的所有socket给select/poll系统调用，这意味着需要将用户态的socket列表copy到内核态，如果以万计的句柄会导致每次都要copy几十几百KB的内存到内核态，非常低效。而我们调用epoll\_wait时就相当于以往调用select/poll，但是这时却不用传递socket句柄给内核，因为内核已经在epoll\_ctl中拿到了要监控的句柄列表。

所以，实际上在你调用epoll\_create后，内核就已经在内核态开始准备帮你存储要监控的句柄了，每次调用epoll\_ctl只是在往内核的数据结构里塞入新的socket句柄。

在内核里，一切皆文件。所以，epoll向内核注册了一个文件系统，用于存储上述的被监控socket。当你调用epoll\_create时，就会在这个虚拟的epoll文件系统里创建一个file结点。当然这个file不是普通文件，它只服务于epoll。

epoll在被内核初始化时（操作系统启动），同时会开辟出epoll自己的内核高速cache区，用于安置每一个我们想监控的socket，这些socket会以红黑树的形式保存在内核cache里，以支持快速的查找、插入、删除。这个内核高速cache区，就是建立连续的物理内存页，然后在之上建立slab层，通常来讲，就是物理上分配好你想要的size的内存对象，每次使用时都是使用空闲的已分配好的对象。

static int \_\_init eventpoll\_init(void) {

... ...

/\* Allocates slab cache used to allocate "struct epitem" items \*/

epi\_cache = kmem\_cache\_create("eventpoll\_epi", sizeof(struct epitem),

0, SLAB\_HWCACHE\_ALIGN|EPI\_SLAB\_DEBUG|SLAB\_PANIC,

NULL, NULL);

/\* Allocates slab cache used to allocate "struct eppoll\_entry" \*/

pwq\_cache = kmem\_cache\_create("eventpoll\_pwq",

sizeof(struct eppoll\_entry), 0,

EPI\_SLAB\_DEBUG|SLAB\_PANIC, NULL, NULL);

... ...

}

epoll的高效就在于，当我们调用epoll\_ctl往里塞入百万个句柄时，epoll\_wait仍然可以飞快的返回，并有效的将发生事件的句柄给我们用户。这是由于我们在调用epoll\_create时，内核除了帮我们在epoll文件系统里建了个file结点，在内核cache里建了个红黑树用于存储以后epoll\_ctl传来的socket外，还会再建立一个list链表，用于存储准备就绪的事件，当epoll\_wait调用时，仅仅观察这个list链表里有没有数据即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。所以，epoll\_wait非常高效。

而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll\_wait仅需要从内核态copy少量的句柄到用户态而已，因此就会非常的高效！

然而,这个准备就绪list链表是怎么维护的呢？当我们执行epoll\_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。

如此，一个红黑树，一张准备就绪句柄链表，少量的内核cache，就帮我们解决了大并发下的socket处理问题。执行epoll\_create时，创建了红黑树和就绪链表，执行epoll\_ctl时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据。执行epoll\_wait时立刻返回准备就绪链表里的数据即可。

最后看看epoll独有的两种模式LT和ET。无论是LT和ET模式，都适用于以上所说的流程。区别是，LT模式下，只要一个句柄上的事件一次没有处理完，会在以后调用epoll\_wait时每次返回这个句柄，而ET模式仅在第一次返回。

当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll\_wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表，最后，epoll\_wait需要做的事情，就是检查这些socket，如果不是ET模式（就是LT模式的句柄了），并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表了。所以，非ET的句柄，只要它上面还有事件，epoll\_wait每次都会返回。而ET模式的句柄，除非有新中断到，即使socket上的事件没有处理完，也是不会每次从epoll\_wait返回的。

因此epoll比select的提高实际上是一个用空间换时间思想的具体应用.对比阻塞IO的处理模型, 可以看到采用了多路复用IO之后, 程序可以自由的进行自己除了IO操作之外的工作, 只有到IO状态发生变化的时候由多路复用IO进行通知, 然后再采取相应的操作, 而不用一直阻塞等待IO状态发生变化,提高效率.

## 12. Golang GC 时会发生什么?

首先我们先来了解下垃圾回收.什么是垃圾回收？

内存管理是程序员开发应用的一大难题。传统的系统级编程语言（主要指C/C++）中，程序开发者必须对内存小心的进行管理操作，控制内存的申请及释放。因为稍有不慎，就可能产生内存泄露问题，这种问题不易发现并且难以定位，一直成为困扰程序开发者的噩梦。如何解决这个头疼的问题呢？

过去一般采用两种办法：

内存泄露检测工具。这种工具的原理一般是静态代码扫描，通过扫描程序检测可能出现内存泄露的代码段。然而检测工具难免有疏漏和不足，只能起到辅助作用。

智能指针。这是 c++ 中引入的自动内存管理方法，通过拥有自动内存管理功能的指针对象来引用对象，是程序员不用太关注内存的释放，而达到内存自动释放的目的。这种方法是采用最广泛的做法，但是对程序开发者有一定的学习成本（并非语言层面的原生支持），而且一旦有忘记使用的场景依然无法避免内存泄露。

为了解决这个问题，后来开发出来的几乎所有新语言（java，python，php等等）都引入了语言层面的自动内存管理 – 也就是语言的使用者只用关注内存的申请而不必关心内存的释放，内存释放由虚拟机（virtual machine）或运行时（runtime）来自动进行管理。而这种对不再使用的内存资源进行自动回收的行为就被称为垃圾回收。

### 常用的垃圾回收的方法:

### 引用计数（reference counting）

这是最简单的一种垃圾回收算法，和之前提到的智能指针异曲同工。对每个对象维护一个引用计数，当引用该对象的对象被销毁或更新时被引用对象的引用计数自动减一，当被引用对象被创建或被赋值给其他对象时引用计数自动加一。当引用计数为0时则立即回收对象。

这种方法的优点是实现简单，并且内存的回收很及时。这种算法在内存比较紧张和实时性比较高的系统中使用的比较广泛，如ios cocoa框架，php，python等。

但是简单引用计数算法也有明显的缺点：

频繁更新引用计数降低了性能。

一种简单的解决方法就是编译器将相邻的引用计数更新操作合并到一次更新；还有一种方法是针对频繁发生的临时变量引用不进行计数，而是在引用达到0时通过扫描堆栈确认是否还有临时对象引用而决定是否释放。等等还有很多其他方法，具体可以参考这里。

循环引用。

当对象间发生循环引用时引用链中的对象都无法得到释放。最明显的解决办法是避免产生循环引用，如cocoa引入了strong指针和weak指针两种指针类型。或者系统检测循环引用并主动打破循环链。当然这也增加了垃圾回收的复杂度。

### 标记-清除（mark and sweep）

标记-清除（mark and sweep）分为两步，标记从根变量开始迭代得遍历所有被引用的对象，对能够通过应用遍历访问到的对象都进行标记为“被引用”；标记完成后进行清除操作，对没有标记过的内存进行回收（回收同时可能伴有碎片整理操作）。这种方法解决了引用计数的不足，但是也有比较明显的问题：每次启动垃圾回收都会暂停当前所有的正常代码执行，回收是系统响应能力大大降低！当然后续也出现了很多mark&sweep算法的变种（如三色标记法）优化了这个问题。

### 分代搜集（generation）

java的jvm 就使用的分代回收的思路。在面向对象编程语言中，绝大多数对象的生命周期都非常短。分代收集的基本思想是，将堆划分为两个或多个称为代（generation）的空间。新创建的对象存放在称为新生代（young generation）中（一般来说，新生代的大小会比 老年代小很多），随着垃圾回收的重复执行，生命周期较长的对象会被提升（promotion）到老年代中（这里用到了一个分类的思路，这个是也是科学思考的一个基本思路）。

因此，新生代垃圾回收和老年代垃圾回收两种不同的垃圾回收方式应运而生，分别用于对各自空间中的对象执行垃圾回收。新生代垃圾回收的速度非常快，比老年代快几个数量级，即使新生代垃圾回收的频率更高，执行效率也仍然比老年代垃圾回收强，这是因为大多数对象的生命周期都很短，根本无需提升到老年代。

### Golang GC 时会发生什么?

Golang 1.5后，采取的是“非分代的、非移动的、并发的、三色的”标记清除垃圾回收算法。

golang 中的 gc 基本上是标记清除的过程：

gc的过程一共分为四个阶段：

栈扫描（开始时STW）

第一次标记（并发）

第二次标记（STW）

清除（并发）

整个进程空间里申请每个对象占据的内存可以视为一个图，初始状态下每个内存对象都是白色标记。

先STW，做一些准备工作，比如 enable write barrier。然后取消STW，将扫描任务作为多个并发的goroutine立即入队给调度器，进而被CPU处理

第一轮先扫描root对象，包括全局指针和 goroutine 栈上的指针，标记为灰色放入队列

第二轮将第一步队列中的对象引用的对象置为灰色加入队列，一个对象引用的所有对象都置灰并加入队列后，这个对象才能置为黑色并从队列之中取出。循环往复，最后队列为空时，整个图剩下的白色内存空间即不可到达的对象，即没有被引用的对象；

第三轮再次STW，将第二轮过程中新增对象申请的内存进行标记（灰色），这里使用了write barrier（写屏障）去记录

Golang gc 优化的核心就是尽量使得 STW(Stop The World) 的时间越来越短。

详细的Golang的GC介绍可以参看Golang垃圾回收.

## 13. Golang 中 Goroutine 如何调度?

goroutine是Golang语言中最经典的设计，也是其魅力所在，goroutine的本质是协程，是实现并行计算的核心。 goroutine使用方式非常的简单，只需使用go关键字即可启动一个协程，并且它是处于异步方式运行，你不需要等它运行完成以后在执行以后的代码。

go func()//通过go关键字启动一个协程来运行函数

协程:

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。 因此，协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。 线程和进程的操作是由程序触发系统接口，最后的执行者是系统；协程的操作执行者则是用户自身程序，goroutine也是协程。

groutine能拥有强大的并发实现是通过GPM调度模型实现.

Go的调度器内部有四个重要的结构：M，P，S，Sched，如上图所示（Sched未给出）.

M:M代表内核级线程，一个M就是一个线程，goroutine就是跑在M之上的；M是一个很大的结构，里面维护小对象内存cache（mcache）、当前执行的goroutine、随机数发生器等等非常多的信息

G:代表一个goroutine，它有自己的栈，instruction pointer和其他信息（正在等待的channel等等），用于调度。

P:P全称是Processor，处理器，它的主要用途就是用来执行goroutine的，所以它也维护了一个goroutine队列，里面存储了所有需要它来执行的goroutine

Sched：代表调度器，它维护有存储M和G的队列以及调度器的一些状态信息等。

调度实现:

从上图中可以看到，有2个物理线程M，每一个M都拥有一个处理器P，每一个也都有一个正在运行的goroutine。P的数量可以通过GOMAXPROCS()来设置，它其实也就代表了真正的并发度，即有多少个goroutine可以同时运行。

图中灰色的那些goroutine并没有运行，而是出于ready的就绪态，正在等待被调度。P维护着这个队列（称之为runqueue），Go语言里，启动一个goroutine很容易：go function 就行，所以每有一个go语句被执行，runqueue队列就在其末尾加入一个goroutine，在下一个调度点，就从runqueue中取出（如何决定取哪个goroutine？）一个goroutine执行。

当一个OS线程M0陷入阻塞时，P转而在运行M1，图中的M1可能是正被创建，或者从线程缓存中取出。

当MO返回时，它必须尝试取得一个P来运行goroutine，一般情况下，它会从其他的OS线程那里拿一个P过来， 如果没有拿到的话，它就把goroutine放在一个global runqueue里，然后自己睡眠（放入线程缓存里）。所有的P也会周期性的检查global runqueue并运行其中的goroutine，否则global runqueue上的goroutine永远无法执行。

另一种情况是P所分配的任务G很快就执行完了（分配不均），这就导致了这个处理器P很忙，但是其他的P还有任务，此时如果global runqueue没有任务G了，那么P不得不从其他的P里拿一些G来执行。

通常来说，如果P从其他的P那里要拿任务的话，一般就拿run queue的一半，这就确保了每个OS线程都能充分的使用。

## 14. 并发编程概念是什么？

并发是指两个或多个事件在同一时间间隔发生。

并行是指两个或者多个事件在同一时刻发生；

并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的。而并行是真正意义上的“同时执行”。

并发编程是指在一台处理器上“同时”处理多个任务。并发是在同一实体上的多个事件。多个事件在同一时间间隔发生。并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

如果希望让goroutine 并行，必须使用多于一个逻辑处理器。当有多个逻辑处理器时，调度器会将goroutine 平等分配到每个逻辑处理器上。这会让goroutine 在不同的线程上运行。不过要想真的实现并行的效果，用户需要让自己的程序运行在有多个物理处理器的机器上。否则，哪怕Go 语言运行时使用多个线程，goroutine 依然会在同一个物理处理器上并发运行，达不到并行的效果。

## 15. 负载均衡原理是什么?

负载均衡Load Balance）是高可用网络基础架构的关键组件，通常用于将工作负载分布到多个服务器来提高网站、应用、数据库或其他服务的性能和可靠性。负载均衡，其核心就是网络流量分发，分很多维度。

负载均衡（Load Balance）通常是分摊到多个操作单元上进行执行，例如Web服务器、FTP服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。

负载均衡是建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

通过一个例子详细介绍:

没有负载均衡 web 架构

在这里用户是直连到 web 服务器，如果这个服务器宕机了，那么用户自然也就没办法访问了。 另外，如果同时有很多用户试图访问服务器，超过了其能处理的极限，就会出现加载速度缓慢或根本无法连接的情况。

而通过在后端引入一个负载均衡器和至少一个额外的 web 服务器，可以缓解这个故障。 通常情况下，所有的后端服务器会保证提供相同的内容，以便用户无论哪个服务器响应，都能收到一致的内容。

有负载均衡 web 架构

用户访问负载均衡器，再由负载均衡器将请求转发给后端服务器。在这种情况下，单点故障现在转移到负载均衡器上了。 这里又可以通过引入第二个负载均衡器来缓解。

那么负载均衡器的工作方式是什么样的呢,负载均衡器又可以处理什么样的请求？

负载均衡器的管理员能主要为下面四种主要类型的请求设置转发规则：

HTTP (七层)

HTTPS (七层)

TCP (四层)

UDP (四层)

负载均衡器如何选择要转发的后端服务器？

负载均衡器一般根据两个因素来决定要将请求转发到哪个服务器。首先，确保所选择的服务器能够对请求做出响应，然后根据预先配置的规则从健康服务器池（healthy pool）中进行选择。

因为，负载均衡器应当只选择能正常做出响应的后端服务器，因此就需要有一种判断后端服务器是否健康的方法。为了监视后台服务器的运行状况，运行状态检查服务会定期尝试使用转发规则定义的协议和端口去连接后端服务器。 如果，服务器无法通过健康检查，就会从池中剔除，保证流量不会被转发到该服务器，直到其再次通过健康检查为止。

### 负载均衡算法

负载均衡算法决定了后端的哪些健康服务器会被选中。 其中常用的算法包括：

Round Robin（轮询）：为第一个请求选择列表中的第一个服务器，然后按顺序向下移动列表直到结尾，然后循环。

Least Connections（最小连接）：优先选择连接数最少的服务器，在普遍会话较长的情况下推荐使用。

Source：根据请求源的 IP 的散列（hash）来选择要转发的服务器。这种方式可以一定程度上保证特定用户能连接到相同的服务器。

如果你的应用需要处理状态而要求用户能连接到和之前相同的服务器。可以通过 Source 算法基于客户端的 IP 信息创建关联，或者使用粘性会话（sticky sessions）。

除此之外，想要解决负载均衡器的单点故障问题，可以将第二个负载均衡器连接到第一个上，从而形成一个集群。

## 16. LVS相关了解.

LVS是 Linux Virtual Server 的简称，也就是Linux虚拟服务器。这是一个由章文嵩博士发起的一个开源项目，它的官方网站是LinuxVirtualServer现在 LVS 已经是 Linux 内核标准的一部分。使用 LVS 可以达到的技术目标是：通过 LVS 达到的负载均衡技术和 Linux 操作系统实现一个高性能高可用的 Linux 服务器集群，它具有良好的可靠性、可扩展性和可操作性。 从而以低廉的成本实现最优的性能。LVS 是一个实现负载均衡集群的开源软件项目，LVS架构从逻辑上可分为调度层、Server集群层和共享存储。

LVS的基本工作原理:

当用户向负载均衡调度器（Director Server）发起请求，调度器将请求发往至内核空间

PREROUTING链首先会接收到用户请求，判断目标IP确定是本机IP，将数据包发往INPUT链

IPVS是工作在INPUT链上的，当用户请求到达INPUT时，IPVS会将用户请求和自己已定义好的集群服务进行比对，如果用户请求的就是定义的集群服务，那么此时IPVS会强行修改数据包里的目标IP地址及端口，并将新的数据包发往POSTROUTING链

POSTROUTING链接收数据包后发现目标IP地址刚好是自己的后端服务器，那么此时通过选路，将数据包最终发送给后端的服务器

LVS的组成:

LVS 由2部分程序组成，包括 ipvs 和 ipvsadm。

ipvs(ip virtual server)：一段代码工作在内核空间，叫ipvs，是真正生效实现调度的代码。

ipvsadm：另外一段是工作在用户空间，叫ipvsadm，负责为ipvs内核框架编写规则，定义谁是集群服务，而谁是后端真实的服务器(Real Server)

详细的LVS的介绍可以参考LVS详解.

## 17. 微服务架构是什么样子的?

通常传统的项目体积庞大，需求、设计、开发、测试、部署流程固定。新功能需要在原项目上做修改。

但是微服务可以看做是对大项目的拆分，是在快速迭代更新上线的需求下产生的。新的功能模块会发布成新的服务组件，与其他已发布的服务组件一同协作。 服务内部有多个生产者和消费者，通常以http rest的方式调用，服务总体以一个（或几个）服务的形式呈现给客户使用。

微服务架构是一种思想对微服务架构我们没有一个明确的定义，但简单来说微服务架构是：

采用一组服务的方式来构建一个应用，服务独立部署在不同的进程中，不同服务通过一些轻量级交互机制来通信，例如 RPC、HTTP 等，服务可独立扩展伸缩，每个服务定义了明确的边界，不同的服务甚至可以采用不同的编程语言来实现，由独立的团队来维护。

Golang的微服务框架kit中有详细的微服务的例子,可以参考学习.

微服务架构设计包括：

服务熔断降级限流机制 熔断降级的概念(Rate Limiter 限流器,Circuit breaker 断路器).

框架调用方式解耦方式 Kit 或 Istio 或 Micro 服务发现(consul zookeeper kubeneters etcd ) RPC调用框架.

链路监控,zipkin和prometheus.

多级缓存.

网关 (kong gateway).

Docker部署管理 Kubenetters.

自动集成部署 CI/CD 实践.

自动扩容机制规则.

压测 优化.

Trasport 数据传输(序列化和反序列化).

Logging 日志.

Metrics 指针对每个请求信息的仪表盘化.

微服务架构介绍详细的可以参考:

Microservice Architectures

## 18. 分布式锁实现原理，用过吗？

在分析分布式锁的三种实现方式之前，先了解一下分布式锁应该具备哪些条件：

在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行；

高可用的获取锁与释放锁；

高性能的获取锁与释放锁；

具备可重入特性；

具备锁失效机制，防止死锁；

具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败。

分布式的CAP理论告诉我们“

任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。

”所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

通常分布式锁以单独的服务方式实现，目前比较常用的分布式锁实现有三种：

### 基于数据库实现分布式锁。

基于缓存（redis，memcached，tair）实现分布式锁。

基于Zookeeper实现分布式锁。

尽管有这三种方案，但是不同的业务也要根据自己的情况进行选型，他们之间没有最好只有更适合！

基于数据库的实现方式

基于数据库的实现方式的核心思想是：在数据库中创建一个表，表中包含方法名等字段，并在方法名字段上创建唯一索引，想要执行某个方法，就使用这个方法名向表中插入数据，成功插入则获取锁，执行完成后删除对应的行数据释放锁。

创建一个表：

DROP TABLE IF EXISTS `method\_lock`;

CREATE TABLE `method\_lock` (

`id` int(11) unsigned NOT NULL AUTO\_INCREMENT COMMENT '主键',

`method\_name` varchar(64) NOT NULL COMMENT '锁定的方法名',

`desc` varchar(255) NOT NULL COMMENT '备注信息',

`update\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (`id`),

UNIQUE KEY `uidx\_method\_name` (`method\_name`) USING BTREE

) ENGINE=InnoDB AUTO\_INCREMENT=3 DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';

想要执行某个方法，就使用这个方法名向表中插入数据：

INSERT INTO method\_lock (method\_name, desc) VALUES ('methodName', '测试的methodName');

因为我们对method\_name做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

成功插入则获取锁，执行完成后删除对应的行数据释放锁：

delete from method\_lock where method\_name ='methodName';

注意：这里只是使用基于数据库的一种方法，使用数据库实现分布式锁还有很多其他的用法可以实现！

使用基于数据库的这种实现方式很简单，但是对于分布式锁应该具备的条件来说，它有一些问题需要解决及优化：

1、因为是基于数据库实现的，数据库的可用性和性能将直接影响分布式锁的可用性及性能，所以，数据库需要双机部署、数据同步、主备切换；

2、不具备可重入的特性，因为同一个线程在释放锁之前，行数据一直存在，无法再次成功插入数据，所以，需要在表中新增一列，用于记录当前获取到锁的机器和线程信息，在再次获取锁的时候，先查询表中机器和线程信息是否和当前机器和线程相同，若相同则直接获取锁；

3、没有锁失效机制，因为有可能出现成功插入数据后，服务器宕机了，对应的数据没有被删除，当服务恢复后一直获取不到锁，所以，需要在表中新增一列，用于记录失效时间，并且需要有定时任务清除这些失效的数据；

4、不具备阻塞锁特性，获取不到锁直接返回失败，所以需要优化获取逻辑，循环多次去获取。

5、在实施的过程中会遇到各种不同的问题，为了解决这些问题，实现方式将会越来越复杂；依赖数据库需要一定的资源开销，性能问题需要考虑。

### 基于Redis的实现方式

选用Redis实现分布式锁原因：

Redis有很高的性能；

Redis命令对此支持较好，实现起来比较方便

主要实现方式:

SET lock currentTime+expireTime EX 600 NX，使用set设置lock值，并设置过期时间为600秒，如果成功，则获取锁；

获取锁后，如果该节点掉线，则到过期时间ock值自动失效；

释放锁时，使用del删除lock键值；

使用redis单机来做分布式锁服务，可能会出现单点问题，导致服务可用性差，因此在服务稳定性要求高的场合，官方建议使用redis集群（例如5台，成功请求锁超过3台就认为获取锁），来实现redis分布式锁。详见RedLock。

优点:性能高，redis可持久化，也能保证数据不易丢失,redis集群方式提高稳定性。

缺点:使用redis主从切换时可能丢失部分数据。

### 基于ZooKeeper的实现方式

ZooKeeper是一个为分布式应用提供一致性服务的开源组件，它内部是一个分层的文件系统目录树结构，规定同一个目录下只能有一个唯一文件名。基于ZooKeeper实现分布式锁的步骤如下：

创建一个目录mylock；

线程A想获取锁就在mylock目录下创建临时顺序节点；

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；

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；

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。

这里推荐一个Apache的开源库Curator，它是一个ZooKeeper客户端，Curator提供的InterProcessMutex是分布式锁的实现，acquire方法用于获取锁，release方法用于释放锁。

优点：具备高可用、可重入、阻塞锁特性，可解决失效死锁问题。

缺点：因为需要频繁的创建和删除节点，性能上不如Redis方式。

上面的三种实现方式，没有在所有场合都是完美的，所以，应根据不同的应用场景选择最适合的实现方式。

在分布式环境中，对资源进行上锁有时候是很重要的，比如抢购某一资源，这时候使用分布式锁就可以很好地控制资源。

## 19. Etcd怎么实现分布式锁?

首先思考下Etcd是什么？可能很多人第一反应可能是一个键值存储仓库，却没有重视官方定义的后半句，用于配置共享和服务发现。

A highly-available key value store for shared configuration and service discovery.

实际上，etcd 作为一个受到 ZooKeeper 与 doozer 启发而催生的项目，除了拥有与之类似的功能外，更专注于以下四点。

简单：基于 HTTP+JSON 的 API 让你用 curl 就可以轻松使用。

安全：可选 SSL 客户认证机制。

快速：每个实例每秒支持一千次写操作。

可信：使用 Raft 算法充分实现了分布式。

但是这里我们主要讲述Etcd如何实现分布式锁?

因为 Etcd 使用 Raft 算法保持了数据的强一致性，某次操作存储到集群中的值必然是全局一致的，所以很容易实现分布式锁。锁服务有两种使用方式，一是保持独占，二是控制时序。

保持独占即所有获取锁的用户最终只有一个可以得到。etcd 为此提供了一套实现分布式锁原子操作 CAS（CompareAndSwap）的 API。通过设置prevExist值，可以保证在多个节点同时去创建某个目录时，只有一个成功。而创建成功的用户就可以认为是获得了锁。

控制时序，即所有想要获得锁的用户都会被安排执行，但是获得锁的顺序也是全局唯一的，同时决定了执行顺序。etcd 为此也提供了一套 API（自动创建有序键），对一个目录建值时指定为POST动作，这样 etcd 会自动在目录下生成一个当前最大的值为键，存储这个新的值（客户端编号）。同时还可以使用 API 按顺序列出所有当前目录下的键值。此时这些键的值就是客户端的时序，而这些键中存储的值可以是代表客户端的编号。

在这里Ectd实现分布式锁基本实现原理为：

在ectd系统里创建一个key

如果创建失败，key存在，则监听该key的变化事件，直到该key被删除，回到1

如果创建成功，则认为我获得了锁

应用示例:

package etcdsync

import (

"fmt"

"io"

"os"

"sync"

"time"

"github.com/coreos/etcd/client"

"github.com/coreos/etcd/Godeps/\_workspace/src/golang.org/x/net/context"

)

const (

defaultTTL = 60

defaultTry = 3

deleteAction = "delete"

expireAction = "expire"

)

// A Mutex is a mutual exclusion lock which is distributed across a cluster.

type Mutex struct {

key string

id string // The identity of the caller

client client.Client

kapi client.KeysAPI

ctx context.Context

ttl time.Duration

mutex \*sync.Mutex

logger io.Writer

}

// New creates a Mutex with the given key which must be the same

// across the cluster nodes.

// machines are the ectd cluster addresses

func New(key string, ttl int, machines []string) \*Mutex {

cfg := client.Config{

Endpoints: machines,

Transport: client.DefaultTransport,

HeaderTimeoutPerRequest: time.Second,

}

c, err := client.New(cfg)

if err != nil {

return nil

}

hostname, err := os.Hostname()

if err != nil {

return nil

}

if len(key) == 0 || len(machines) == 0 {

return nil

}

if key[0] != '/' {

key = "/" + key

}

if ttl < 1 {

ttl = defaultTTL

}

return &Mutex{

key: key,

id: fmt.Sprintf("%v-%v-%v", hostname, os.Getpid(), time.Now().Format("20060102-15:04:05.999999999")),

client: c,

kapi: client.NewKeysAPI(c),

ctx: context.TODO(),

ttl: time.Second \* time.Duration(ttl),

mutex: new(sync.Mutex),

}

}

// Lock locks m.

// If the lock is already in use, the calling goroutine

// blocks until the mutex is available.

func (m \*Mutex) Lock() (err error) {

m.mutex.Lock()

for try := 1; try <= defaultTry; try++ {

if m.lock() == nil {

return nil

}

m.debug("Lock node %v ERROR %v", m.key, err)

if try < defaultTry {

m.debug("Try to lock node %v again", m.key, err)

}

}

return err

}

func (m \*Mutex) lock() (err error) {

m.debug("Trying to create a node : key=%v", m.key)

setOptions := &client.SetOptions{

PrevExist:client.PrevNoExist,

TTL: m.ttl,

}

resp, err := m.kapi.Set(m.ctx, m.key, m.id, setOptions)

if err == nil {

m.debug("Create node %v OK [%q]", m.key, resp)

return nil

}

m.debug("Create node %v failed [%v]", m.key, err)

e, ok := err.(client.Error)

if !ok {

return err

}

if e.Code != client.ErrorCodeNodeExist {

return err

}

// Get the already node's value.

resp, err = m.kapi.Get(m.ctx, m.key, nil)

if err != nil {

return err

}

m.debug("Get node %v OK", m.key)

watcherOptions := &client.WatcherOptions{

AfterIndex : resp.Index,

Recursive:false,

}

watcher := m.kapi.Watcher(m.key, watcherOptions)

for {

m.debug("Watching %v ...", m.key)

resp, err = watcher.Next(m.ctx)

if err != nil {

return err

}

m.debug("Received an event : %q", resp)

if resp.Action == deleteAction || resp.Action == expireAction {

return nil

}

}

}

// Unlock unlocks m.

// It is a run-time error if m is not locked on entry to Unlock.

//

// A locked Mutex is not associated with a particular goroutine.

// It is allowed for one goroutine to lock a Mutex and then

// arrange for another goroutine to unlock it.

func (m \*Mutex) Unlock() (err error) {

defer m.mutex.Unlock()

for i := 1; i <= defaultTry; i++ {

var resp \*client.Response

resp, err = m.kapi.Delete(m.ctx, m.key, nil)

if err == nil {

m.debug("Delete %v OK", m.key)

return nil

}

m.debug("Delete %v falied: %q", m.key, resp)

e, ok := err.(client.Error)

if ok && e.Code == client.ErrorCodeKeyNotFound {

return nil

}

}

return err

}

func (m \*Mutex) debug(format string, v ...interface{}) {

if m.logger != nil {

m.logger.Write([]byte(m.id))

m.logger.Write([]byte(" "))

m.logger.Write([]byte(fmt.Sprintf(format, v...)))

m.logger.Write([]byte("\n"))

}

}

func (m \*Mutex) SetDebugLogger(w io.Writer) {

m.logger = w

}

其实类似的实现有很多，但目前都已经过时，使用的都是被官方标记为deprecated的项目。且大部分接口都不如上述代码简单。 使用上，跟Golang官方sync包的Mutex接口非常类似，先New()，然后调用Lock()，使用完后调用Unlock()，就三个接口，就是这么简单。示例代码如下：

package main

import (

"github.com/zieckey/etcdsync"

"log"

)

func main() {

//etcdsync.SetDebug(true)

log.SetFlags(log.Ldate|log.Ltime|log.Lshortfile)

m := etcdsync.New("/etcdsync", "123", []string{"http://127.0.0.1:2379"})

if m == nil {

log.Printf("etcdsync.NewMutex failed")

}

err := m.Lock()

if err != nil {

log.Printf("etcdsync.Lock failed")

} else {

log.Printf("etcdsync.Lock OK")

}

log.Printf("Get the lock. Do something here.")

err = m.Unlock()

if err != nil {

log.Printf("etcdsync.Unlock failed")

} else {

log.Printf("etcdsync.Unlock OK")

}

}

## 20. Redis的数据结构有哪些，以及实现场景?

Redis的数据结构有五种:

string 字符串

String 数据结构是简单的 key-value 类型，value 不仅可以是 String，也可以是数字（当数字类型用 Long 可以表示的时候encoding 就是整型，其他都存储在 sdshdr 当做字符串）。使用 Strings 类型，可以完全实现目前 Memcached 的功能，并且效率更高。还可以享受 Redis 的定时持久化（可以选择 RDB 模式或者 AOF 模式），操作日志及 Replication 等功能。

除了提供与 Memcached 一样的 get、set、incr、decr 等操作外，Redis 还提供了下面一些操作：

LEN niushuai：O(1)获取字符串长度.

APPEND niushuai redis：往字符串 append 内容，而且采用智能分配内存（每次2倍）.

设置和获取字符串的某一段内容.

设置及获取字符串的某一位（bit）.

批量设置一系列字符串的内容.

原子计数器.

GETSET 命令的妙用，请于清空旧值的同时设置一个新值，配合原子计数器使用.

Hash 字典

在 Memcached 中，我们经常将一些结构化的信息打包成 hashmap，在客户端序列化后存储为一个字符串的值（一般是 JSON 格式），比如用户的昵称、年龄、性别、积分等。这时候在需要修改其中某一项时，通常需要将字符串（JSON）取出来，然后进行反序列化，修改某一项的值，再序列化成字符串（JSON）存储回去。简单修改一个属性就干这么多事情，消耗必定是很大的，也不适用于一些可能并发操作的场合（比如两个并发的操作都需要修改积分）。而 Redis 的 Hash 结构可以使你像在数据库中 Update 一个属性一样只修改某一项属性值。

Hash可以用来存储、读取、修改用户属性。

List 列表

List 说白了就是链表（redis 使用双端链表实现的 List），相信学过数据结构知识的人都应该能理解其结构。使用 List 结构，我们可以轻松地实现最新消息排行等功能（比如新浪微博的 TimeLine ）。List 的另一个应用就是消息队列，可以利用 List 的 \*PUSH 操作，将任务存在 List 中，然后工作线程再用 POP 操作将任务取出进行执行。

Redis 还提供了操作 List 中某一段元素的 API，你可以直接查询，删除 List 中某一段的元素。

List 列表应用:

微博 TimeLine.

消息队列.

Set 集合

Set 就是一个集合，集合的概念就是一堆不重复值的组合。利用 Redis 提供的 Set 数据结构，可以存储一些集合性的数据。比如在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。因为 Redis 非常人性化的为集合提供了求交集、并集、差集等操作，那么就可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。

Set 集合应用:

共同好友、二度好友

利用唯一性，可以统计访问网站的所有独立 IP.

好友推荐的时候，根据 tag 求交集，大于某个 threshold 就可以推荐.

Sorted Set有序集合

和Sets相比，Sorted Sets是将 Set 中的元素增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列，比如一个存储全班同学成绩的 Sorted Sets，其集合 value 可以是同学的学号，而 score 就可以是其考试得分，这样在数据插入集合的时候，就已经进行了天然的排序。另外还可以用 Sorted Sets 来做带权重的队列，比如普通消息的 score 为1，重要消息的 score 为2，然后工作线程可以选择按 score 的倒序来获取工作任务。让重要的任务优先执行。

Sorted Set有序集合应用:

1.带有权重的元素，比如一个游戏的用户得分排行榜. 2.比较复杂的数据结构，一般用到的场景不算太多.

Redis 其他功能使用场景:

订阅-发布系统

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在 Redis 中，你可以设定对某一个 key 值进行消息发布及消息订阅，当一个 key 值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

事务——Transactions

谁说 NoSQL 都不支持事务，虽然 Redis 的 Transactions 提供的并不是严格的 ACID 的事务（比如一串用 EXEC 提交执行的命令，在执行中服务器宕机，那么会有一部分命令执行了，剩下的没执行），但是这个 Transactions 还是提供了基本的命令打包执行的功能（在服务器不出问题的情况下，可以保证一连串的命令是顺序在一起执行的，中间有会有其它客户端命令插进来执行）。Redis 还提供了一个 Watch 功能，你可以对一个 key 进行 Watch，然后再执行 Transactions，在这过程中，如果这个 Watched 的值进行了修改，那么这个 Transactions 会发现并拒绝执行。

## 21. Mysql高可用方案有哪些?

Mysql高可用方案包括:

主从复制方案

这是MySQL自身提供的一种高可用解决方案，数据同步方法采用的是MySQL replication技术。MySQL replication就是从服务器到主服务器拉取二进制日志文件，然后再将日志文件解析成相应的SQL在从服务器上重新执行一遍主服务器的操作，通过这种方式保证数据的一致性。为了达到更高的可用性，在实际的应用环境中，一般都是采用MySQL replication技术配合高可用集群软件keepalived来实现自动failover，这种方式可以实现95.000%的SLA。

MMM/MHA高可用方案

MMM提供了MySQL主主复制配置的监控、故障转移和管理的一套可伸缩的脚本套件。在MMM高可用方案中，典型的应用是双主多从架构，通过MySQL replication技术可以实现两个服务器互为主从，且在任何时候只有一个节点可以被写入，避免了多点写入的数据冲突。同时，当可写的主节点故障时，MMM套件可以立刻监控到，然后将服务自动切换到另一个主节点，继续提供服务，从而实现MySQL的高可用。

Heartbeat/SAN高可用方案

在这个方案中，处理failover的方式是高可用集群软件Heartbeat，它监控和管理各个节点间连接的网络，并监控集群服务，当节点出现故障或者服务不可用时，自动在其他节点启动集群服务。在数据共享方面，通过SAN（Storage Area Network）存储来共享数据，这种方案可以实现99.990%的SLA。

Heartbeat/DRBD高可用方案

这个方案处理failover的方式上依旧采用Heartbeat，不同的是，在数据共享方面，采用了基于块级别的数据同步软件DRBD来实现。DRBD是一个用软件实现的、无共享的、服务器之间镜像块设备内容的存储复制解决方案。和SAN网络不同，它并不共享存储，而是通过服务器之间的网络复制数据。

NDB CLUSTER高可用方案

国内用NDB集群的公司非常少，貌似有些银行有用。NDB集群不需要依赖第三方组件，全部都使用官方组件，能保证数据的一致性，某个数据节点挂掉，其他数据节点依然可以提供服务，管理节点需要做冗余以防挂掉。缺点是：管理和配置都很复杂，而且某些SQL语句例如join语句需要避免。

## 22. Go语言的栈空间管理是怎么样的?

Go语言的运行环境（runtime）会在goroutine需要的时候动态地分配栈空间，而不是给每个goroutine分配固定大小的内存空间。这样就避免了需要程序员来决定栈的大小。

分块式的栈是最初Go语言组织栈的方式。当创建一个goroutine的时候，它会分配一个8KB的内存空间来给goroutine的栈使用。我们可能会考虑当这8KB的栈空间被用完的时候该怎么办?

为了处理这种情况，每个Go函数的开头都有一小段检测代码。这段代码会检查我们是否已经用完了分配的栈空间。如果是的话，它会调用morestack函数。morestack函数分配一块新的内存作为栈空间，并且在这块栈空间的底部填入各种信息（包括之前的那块栈地址）。在分配了这块新的栈空间之后，它会重试刚才造成栈空间不足的函数。这个过程叫做栈分裂（stack split）。

在新分配的栈底部，还插入了一个叫做lessstack的函数指针。这个函数还没有被调用。这样设置是为了从刚才造成栈空间不足的那个函数返回时做准备的。当我们从那个函数返回时，它会跳转到lessstack。lessstack函数会查看在栈底部存放的数据结构里的信息，然后调整栈指针（stack pointer）。这样就完成了从新的栈块到老的栈块的跳转。接下来，新分配的这个块栈空间就可以被释放掉了。

分块式的栈让我们能够按照需求来扩展和收缩栈的大小。 Go开发者不需要花精力去估计goroutine会用到多大的栈。创建一个新的goroutine的开销也不大。当 Go开发者不知道栈会扩展到多少大时，它也能很好的处理这种情况。

这一直是之前Go语言管理栈的的方法。但这个方法有一个问题。缩减栈空间是一个开销相对较大的操作。如果在一个循环里有栈分裂，那么它的开销就变得不可忽略了。一个函数会扩展，然后分裂栈。当它返回的时候又会释放之前分配的内存块。如果这些都发生在一个循环里的话，代价是相当大的。 这就是所谓的热分裂问题（hot split problem）。它是Go语言开发者选择新的栈管理方法的主要原因。新的方法叫做栈复制法（stack copying）。

栈复制法一开始和分块式的栈很像。当goroutine运行并用完栈空间的时候，与之前的方法一样，栈溢出检查会被触发。但是，不像之前的方法那样分配一个新的内存块并链接到老的栈内存块，新的方法会分配一个两倍大的内存块并把老的内存块内容复制到新的内存块里。这样做意味着当栈缩减回之前大小时，我们不需要做任何事情。栈的缩减没有任何代价。而且，当栈再次扩展时，运行环境也不需要再做任何事。它可以重用之前分配的空间。

栈的复制听起来很容易，但实际操作并非那么简单。存储在栈上的变量的地址可能已经被使用到。也就是说程序使用到了一些指向栈的指针。当移动栈的时候，所有指向栈里内容的指针都会变得无效。然而，指向栈内容的指针自身也必定是保存在栈上的。这是为了保证内存安全的必要条件。否则一个程序就有可能访问一段已经无效的栈空间了。

因为垃圾回收的需要，我们必须知道栈的哪些部分是被用作指针了。当我们移动栈的时候，我们可以更新栈里的指针让它们指向新的地址。所有相关的指针都会被更新。我们使用了垃圾回收的信息来复制栈，但并不是任何使用栈的函数都有这些信息。因为很大一部分运行环境是用C语言写的，很多被调用的运行环境里的函数并没有指针的信息，所以也就不能够被复制了。当遇到这种情况时，我们只能退回到分块式的栈并支付相应的开销。

这也是为什么现在运行环境的开发者正在用Go语言重写运行环境的大部分代码。无法用Go语言重写的部分（比如调度器的核心代码和垃圾回收器）会在特殊的栈上运行。这个特殊栈的大小由运行环境的开发者设置。

这些改变除了使栈复制成为可能，它也允许我们在将来实现并行垃圾回收。

另外一种不同的栈处理方式就是在虚拟内存中分配大内存段。由于物理内存只是在真正使用时才会被分配，因此看起来好似你可以分配一个大内存段并让操 作系统处理它。下面是这种方法的一些问题

首先，32位系统只能支持4G字节虚拟内存，并且应用只能用到其中的3G空间。由于同时运行百万goroutines的情况并不少见，因此你很可 能用光虚拟内存，即便我们假设每个goroutine的stack只有8K。

第二，然而我们可以在64位系统中分配大内存，它依赖于过量内存使用。所谓过量使用是指当你分配的内存大小超出物理内存大小时，依赖操作系统保证 在需要时能够分配出物理内存。然而，允许过量使用可能会导致一些风险。由于一些进程分配了超出机器物理内存大小的内存，如果这些进程使用更多内存 时，操作系统将不得不为它们补充分配内存。这会导致操作系统将一些内存段放入磁盘缓存，这常常会增加不可预测的处理延迟。正是考虑到这个原因，一 些新系统关闭了对过量使用的支持。

## 23. Goroutine和Channel的作用分别是什么?

进程是内存资源管理和cpu调度的执行单元。为了有效利用多核处理器的优势，将进程进一步细分，允许一个进程里存在多个线程，这多个线程还是共享同一片内存空间，但cpu调度的最小单元变成了线程。

那协程又是什么呢，以及与线程的差异性??

协程，可以看作是轻量级的线程。但与线程不同的是，线程的切换是由操作系统控制的，而协程的切换则是由用户控制的。

最早支持协程的程序语言应该是lisp方言scheme里的continuation（续延），续延允许scheme保存任意函数调用的现场，保存起来并重新执行。Lua,C#,python等语言也有自己的协程实现。

Go中的goroutinue就是协程,可以实现并行，多个协程可以在多个处理器同时跑。而协程同一时刻只能在一个处理器上跑（可以把宿主语言想象成单线程的就好了）。 然而,多个goroutine之间的通信是通过channel，而协程的通信是通过yield和resume()操作。

goroutine非常简单，只需要在函数的调用前面加关键字go即可，例如:

go elegance()

我们也可以启动5个goroutines分别打印索引。

func main() {

for i:=1;i<5;i++ {

go func(i int) {

fmt.Println(i)

}(i)

}

// 停歇5s，保证打印全部结束

time.Sleep(5\*time.Second)

}

在分析goroutine执行的随机性和并发性，启动了5个goroutine，再加上main函数的主goroutine，总共有6个goroutines。由于goroutine类似于”守护线程“，异步执行的,如果主goroutine不等待片刻，可能程序就没有输出打印了。

在Golang中channel则是goroutinues之间进行通信的渠道。

可以把channel形象比喻为工厂里的传送带,一头的生产者goroutine往传输带放东西,另一头的消费者goroutinue则从输送带取东西。channel实际上是一个有类型的消息队列,遵循先进先出的特点。

channel的操作符号

ch <- data 表示data被发送给channel ch；

data <- ch 表示从channel ch取一个值，然后赋给data。

阻塞式channel

channel默认是没有缓冲区的，也就是说，通信是阻塞的。send操作必须等到有消费者accept才算完成。

应用示例:

func main() {

ch1 := make(chan int)

go pump(ch1) // pump hangs

fmt.Println(<-ch1) // prints only 1

}

func pump(ch chan int) {

for i:= 1; ; i++ {

ch <- i

}

}

在函数pump()里的channel在接受到第一个元素后就被阻塞了，直到主goroutinue取走了数据。最终channel阻塞在接受第二个元素，程序只打印 1。

没有缓冲(buffer)的channel只能容纳一个元素，而带有缓冲(buffer)channel则可以非阻塞容纳N个元素。发送数据到缓冲(buffer) channel不会被阻塞，除非channel已满；同样的，从缓冲(buffer) channel取数据也不会被阻塞，除非channel空了。

## 24. 怎么查看Goroutine的数量?

GOMAXPROCS中控制的是未被阻塞的所有Goroutine,可以被Multiplex到多少个线程上运行,通过GOMAXPROCS可以查看Goroutine的数量。

## 25. 说下Go中的锁有哪些?三种锁，读写锁，互斥锁，还有map的安全的锁?

Go中的三种锁包括:互斥锁,读写锁,sync.Map的安全的锁.

互斥锁

Go并发程序对共享资源进行访问控制的主要手段，由标准库代码包中sync中的Mutex结构体表示。

//Mutex 是互斥锁， 零值是解锁的互斥锁， 首次使用后不得复制互斥锁。

type Mutex struct {

state int32

sema uint32

}

sync.Mutex包中的类型只有两个公开的指针方法Lock和Unlock。

//Locker表示可以锁定和解锁的对象。

type Locker interface {

Lock()

Unlock()

}

//锁定当前的互斥量

//如果锁已被使用，则调用goroutine

//阻塞直到互斥锁可用。

func (m \*Mutex) Lock()

//对当前互斥量进行解锁

//如果在进入解锁时未锁定m，则为运行时错误。

//锁定的互斥锁与特定的goroutine无关。

//允许一个goroutine锁定Mutex然后安排另一个goroutine来解锁它。

func (m \*Mutex) Unlock()

声明一个互斥锁：

var mutex sync.Mutex

不像C或Java的锁类工具，我们可能会犯一个错误：忘记及时解开已被锁住的锁，从而导致流程异常。但Go由于存在defer，所以此类问题出现的概率极低。关于defer解锁的方式如下：

var mutex sync.Mutex

func Write() {

mutex.Lock()

defer mutex.Unlock()

}

如果对一个已经上锁的对象再次上锁，那么就会导致该锁定操作被阻塞，直到该互斥锁回到被解锁状态.

fpackage main

import (

"fmt"

"sync"

"time"

)

func main() {

var mutex sync.Mutex

fmt.Println("begin lock")

mutex.Lock()

fmt.Println("get locked")

for i := 1; i <= 3; i++ {

go func(i int) {

fmt.Println("begin lock ", i)

mutex.Lock()

fmt.Println("get locked ", i)

}(i)

}

time.Sleep(time.Second)

fmt.Println("Unlock the lock")

mutex.Unlock()

fmt.Println("get unlocked")

time.Sleep(time.Second)

}

我们在for循环之前开始加锁，然后在每一次循环中创建一个协程，并对其加锁，但是由于之前已经加锁了，所以这个for循环中的加锁会陷入阻塞直到main中的锁被解锁， time.Sleep(time.Second) 是为了能让系统有足够的时间运行for循环，输出结果如下：

> go run mutex.go

begin lock

get locked

begin lock 3

begin lock 1

begin lock 2

Unlock the lock

get unlocked

get locked 3

这里可以看到解锁后，三个协程会重新抢夺互斥锁权，最终协程3获胜。

互斥锁锁定操作的逆操作并不会导致协程阻塞，但是有可能导致引发一个无法恢复的运行时的panic，比如对一个未锁定的互斥锁进行解锁时就会发生panic。避免这种情况的最有效方式就是使用defer。

我们知道如果遇到panic，可以使用recover方法进行恢复，但是如果对重复解锁互斥锁引发的panic却是无用的（Go 1.8及以后）。

package main

import (

"fmt"

"sync"

)

func main() {

defer func() {

fmt.Println("Try to recover the panic")

if p := recover(); p != nil {

fmt.Println("recover the panic : ", p)

}

}()

var mutex sync.Mutex

fmt.Println("begin lock")

mutex.Lock()

fmt.Println("get locked")

fmt.Println("unlock lock")

mutex.Unlock()

fmt.Println("lock is unlocked")

fmt.Println("unlock lock again")

mutex.Unlock()

}

运行:

> go run mutex.go

begin lock

get locked

unlock lock

lock is unlocked

unlock lock again

fatal error: sync: unlock of unlocked mutex

goroutine 1 [running]:

runtime.throw(0x4bc1a8, 0x1e)

/home/keke/soft/go/src/runtime/panic.go:617 +0x72 fp=0xc000084ea8 sp=0xc000084e78 pc=0x427ba2

sync.throw(0x4bc1a8, 0x1e)

/home/keke/soft/go/src/runtime/panic.go:603 +0x35 fp=0xc000084ec8 sp=0xc000084ea8 pc=0x427b25

sync.(\*Mutex).Unlock(0xc00001a0c8)

/home/keke/soft/go/src/sync/mutex.go:184 +0xc1 fp=0xc000084ef0 sp=0xc000084ec8 pc=0x45f821

main.main()

/home/keke/go/Test/mutex.go:25 +0x25f fp=0xc000084f98 sp=0xc000084ef0 pc=0x486c1f

runtime.main()

/home/keke/soft/go/src/runtime/proc.go:200 +0x20c fp=0xc000084fe0 sp=0xc000084f98 pc=0x4294ec

runtime.goexit()

/home/keke/soft/go/src/runtime/asm\_amd64.s:1337 +0x1 fp=0xc000084fe8 sp=0xc000084fe0 pc=0x450ad1

exit status 2

这里试图对重复解锁引发的panic进行recover，但是我们发现操作失败，虽然互斥锁可以被多个协程共享，但还是建议将对同一个互斥锁的加锁解锁操作放在同一个层次的代码中。

读写锁

读写锁是针对读写操作的互斥锁，可以分别针对读操作与写操作进行锁定和解锁操作 。

读写锁的访问控制规则如下：

① 多个写操作之间是互斥的 ② 写操作与读操作之间也是互斥的 ③ 多个读操作之间不是互斥的

在这样的控制规则下，读写锁可以大大降低性能损耗。

在Go的标准库代码包中sync中的RWMutex结构体表示为:

// RWMutex是一个读/写互斥锁，可以由任意数量的读操作或单个写操作持有。

// RWMutex的零值是未锁定的互斥锁。

//首次使用后，不得复制RWMutex。

//如果goroutine持有RWMutex进行读取而另一个goroutine可能会调用Lock，那么在释放初始读锁之前，goroutine不应该期望能够获取读锁定。

//特别是，这种禁止递归读锁定。 这是为了确保锁最终变得可用; 阻止的锁定会阻止新读操作获取锁定。

type RWMutex struct {

w Mutex //如果有待处理的写操作就持有

writerSem uint32 // 写操作等待读操作完成的信号量

readerSem uint32 //读操作等待写操作完成的信号量

readerCount int32 // 待处理的读操作数量

readerWait int32 // number of departing readers

}

sync中的RWMutex有以下几种方法：

//对读操作的锁定

func (rw \*RWMutex) RLock()

//对读操作的解锁

func (rw \*RWMutex) RUnlock()

//对写操作的锁定

func (rw \*RWMutex) Lock()

//对写操作的解锁

func (rw \*RWMutex) Unlock()

//返回一个实现了sync.Locker接口类型的值，实际上是回调rw.RLock and rw.RUnlock.

func (rw \*RWMutex) RLocker() Locker

Unlock方法会试图唤醒所有想进行读锁定而被阻塞的协程，而 RUnlock方法只会在已无任何读锁定的情况下，试图唤醒一个因欲进行写锁定而被阻塞的协程。若对一个未被写锁定的读写锁进行写解锁，就会引发一个不可恢复的panic，同理对一个未被读锁定的读写锁进行读写锁也会如此。

由于读写锁控制下的多个读操作之间不是互斥的，因此对于读解锁更容易被忽视。对于同一个读写锁，添加多少个读锁定，就必要有等量的读解锁，这样才能其他协程有机会进行操作。

package main

import (

"fmt"

"sync"

"time"

)

func main() {

var rwm sync.RWMutex

for i := 0; i < 5; i++ {

go func(i int) {

fmt.Println("try to lock read ", i)

rwm.RLock()

fmt.Println("get locked ", i)

time.Sleep(time.Second \* 2)

fmt.Println("try to unlock for reading ", i)

rwm.RUnlock()

fmt.Println("unlocked for reading ", i)

}(i)

}

time.Sleep(time.Millisecond \* 1000)

fmt.Println("try to lock for writing")

rwm.Lock()

fmt.Println("locked for writing")

}

运行:

> go run rwmutex.go

try to lock read 0

get locked 0

try to lock read 4

get locked 4

try to lock read 3

get locked 3

try to lock read 1

get locked 1

try to lock read 2

get locked 2

try to lock for writing

try to unlock for reading 0

unlocked for reading 0

try to unlock for reading 2

unlocked for reading 2

try to unlock for reading 1

unlocked for reading 1

try to unlock for reading 3

unlocked for reading 3

try to unlock for reading 4

unlocked for reading 4

locked for writing

这里可以看到创建了五个协程用于对读写锁的读锁定与读解锁操作。在 rwm.Lock()种会对main中协程进行写锁定，但是for循环中的读解锁尚未完成，因此会造成mian中的协程阻塞。当for循环中的读解锁操作都完成后就会试图唤醒main中阻塞的协程，main中的写锁定才会完成。

sync.Map安全锁

golang中的sync.Map是并发安全的，其实也就是sync包中golang自定义的一个名叫Map的结构体。

应用示例:

package main

import (

"sync"

"fmt"

)

func main() {

//开箱即用

var sm sync.Map

//store 方法,添加元素

sm.Store(1,"a")

//Load 方法，获得value

if v,ok:=sm.Load(1);ok{

fmt.Println(v)

}

//LoadOrStore方法，获取或者保存

//参数是一对key：value，如果该key存在且没有被标记删除则返回原先的value（不更新）和true；不存在则store，返回该value 和false

if vv,ok:=sm.LoadOrStore(1,"c");ok{

fmt.Println(vv)

}

if vv,ok:=sm.LoadOrStore(2,"c");!ok{

fmt.Println(vv)

}

//遍历该map，参数是个函数，该函数参的两个参数是遍历获得的key和value，返回一个bool值，当返回false时，遍历立刻结束。

sm.Range(func(k,v interface{})bool{

fmt.Print(k)

fmt.Print(":")

fmt.Print(v)

fmt.Println()

return true

})

}

运行 :

a

a

c

1:a

2:c

sync.Map的数据结构:

type Map struct {

// 该锁用来保护dirty

mu Mutex

// 存读的数据，因为是atomic.value类型，只读类型，所以它的读是并发安全的

read atomic.Value // readOnly

//包含最新的写入的数据，并且在写的时候，会把read 中未被删除的数据拷贝到该dirty中，因为是普通的map存在并发安全问题，需要用到上面的mu字段。

dirty map[interface{}]\*entry

// 从read读数据的时候，会将该字段+1，当等于len（dirty）的时候，会将dirty拷贝到read中（从而提升读的性能）。

misses int

}

read的数据结构是：

type readOnly struct {

m map[interface{}]\*entry

// 如果Map.dirty的数据和m 中的数据不一样是为true

amended bool

}

entry的数据结构：

type entry struct {

//可见value是个指针类型，虽然read和dirty存在冗余情况（amended=false），但是由于是指针类型，存储的空间应该不是问题

p unsafe.Pointer // \*interface{}

}

Delete 方法:

func (m \*Map) Delete(key interface{}) {

read, \_ := m.read.Load().(readOnly)

e, ok := read.m[key]

//如果read中没有，并且dirty中有新元素，那么就去dirty中去找

if !ok && read.amended {

m.mu.Lock()

//这是双检查（上面的if判断和锁不是一个原子性操作）

read, \_ = m.read.Load().(readOnly)

e, ok = read.m[key]

if !ok && read.amended {

//直接删除

delete(m.dirty, key)

}

m.mu.Unlock()

}

if ok {

//如果read中存在该key，则将该value 赋值nil（采用标记的方式删除！）

e.delete()

}

}

func (e \*entry) delete() (hadValue bool) {

for {

p := atomic.LoadPointer(&e.p)

if p == nil || p == expunged {

return false

}

if atomic.CompareAndSwapPointer(&e.p, p, nil) {

return true

}

}

}

Store 方法:

func (m \*Map) Store(key, value interface{}) {

// 如果m.read存在这个key，并且没有被标记删除，则尝试更新。

read, \_ := m.read.Load().(readOnly)

if e, ok := read.m[key]; ok && e.tryStore(&value) {

return

}

// 如果read不存在或者已经被标记删除

m.mu.Lock()

read, \_ = m.read.Load().(readOnly)

if e, ok := read.m[key]; ok {

//如果entry被标记expunge，则表明dirty没有key，可添加入dirty，并更新entry

if e.unexpungeLocked() {

//加入dirty中

m.dirty[key] = e

}

//更新value值

e.storeLocked(&value)

//dirty 存在该key，更新

} else if e, ok := m.dirty[key]; ok {

e.storeLocked(&value)

//read 和dirty都没有，新添加一条

} else {

//dirty中没有新的数据，往dirty中增加第一个新键

if !read.amended {

//将read中未删除的数据加入到dirty中

m.dirtyLocked()

m.read.Store(readOnly{m: read.m, amended: true})

}

m.dirty[key] = newEntry(value)

}

m.mu.Unlock()

}

//将read中未删除的数据加入到dirty中

func (m \*Map) dirtyLocked() {

if m.dirty != nil {

return

}

read, \_ := m.read.Load().(readOnly)

m.dirty = make(map[interface{}]\*entry, len(read.m))

//read如果较大的话，可能影响性能

for k, e := range read.m {

//通过此次操作，dirty中的元素都是未被删除的，可见expunge的元素不在dirty中

if !e.tryExpungeLocked() {

m.dirty[k] = e

}

}

}

//判断entry是否被标记删除，并且将标记为nil的entry更新标记为expunge

func (e \*entry) tryExpungeLocked() (isExpunged bool) {

p := atomic.LoadPointer(&e.p)

for p == nil {

// 将已经删除标记为nil的数据标记为expunged

if atomic.CompareAndSwapPointer(&e.p, nil, expunged) {

return true

}

p = atomic.LoadPointer(&e.p)

}

return p == expunged

}

//对entry 尝试更新

func (e \*entry) tryStore(i \*interface{}) bool {

p := atomic.LoadPointer(&e.p)

if p == expunged {

return false

}

for {

if atomic.CompareAndSwapPointer(&e.p, p, unsafe.Pointer(i)) {

return true

}

p = atomic.LoadPointer(&e.p)

if p == expunged {

return false

}

}

}

//read里 将标记为expunge的更新为nil

func (e \*entry) unexpungeLocked() (wasExpunged bool) {

return atomic.CompareAndSwapPointer(&e.p, expunged, nil)

}

//更新entry

func (e \*entry) storeLocked(i \*interface{}) {

atomic.StorePointer(&e.p, unsafe.Pointer(i))

}

因此，每次操作先检查read，因为read 并发安全，性能好些；read不满足，则加锁检查dirty，一旦是新的键值，dirty会被read更新。

Load方法:

Load方法是一个加载方法，查找key。

func (m \*Map) Load(key interface{}) (value interface{}, ok bool) {

//因read只读，线程安全，先查看是否满足条件

read, \_ := m.read.Load().(readOnly)

e, ok := read.m[key]

//如果read没有，并且dirty有新数据，那从dirty中查找，由于dirty是普通map，线程不安全，这个时候用到互斥锁了

if !ok && read.amended {

m.mu.Lock()

// 双重检查

read, \_ = m.read.Load().(readOnly)

e, ok = read.m[key]

// 如果read中还是不存在，并且dirty中有新数据

if !ok && read.amended {

e, ok = m.dirty[key]

// mssLocked（）函数是性能是sync.Map 性能得以保证的重要函数，目的讲有锁的dirty数据，替换到只读线程安全的read里

m.missLocked()

}

m.mu.Unlock()

}

if !ok {

return nil, false

}

return e.load()

}

//dirty 提升至read 关键函数，当misses 经过多次因为load之后，大小等于len（dirty）时候，讲dirty替换到read里，以此达到性能提升。

func (m \*Map) missLocked() {

m.misses++

if m.misses < len(m.dirty) {

return

}

//原子操作，耗时很小

m.read.Store(readOnly{m: m.dirty})

m.dirty = nil

m.misses = 0

}

sync.Map是通过冗余的两个数据结构(read、dirty),实现性能的提升。为了提升性能，load、delete、store等操作尽量使用只读的read；为了提高read的key击中概率，采用动态调整，将dirty数据提升为read；对于数据的删除，采用延迟标记删除法，只有在提升dirty的时候才删除。

## 26. 读写锁或者互斥锁读的时候能写吗?

Go中读写锁包括读锁和写锁，多个读线程可以同时访问共享数据；写线程必须等待所有读线程都释放锁以后，才能取得锁；同样的，读线程必须等待写线程释放锁后，才能取得锁，也就是说读写锁要确保的是如下互斥关系，可以同时读，但是读-写，写-写都是互斥的。

## 27. 怎么限制Goroutine的数量.

在Golang中，Goroutine虽然很好，但是数量太多了，往往会带来很多麻烦，比如耗尽系统资源导致程序崩溃，或者CPU使用率过高导致系统忙不过来。所以我们可以限制下Goroutine的数量,这样就需要在每一次执行go之前判断goroutine的数量，如果数量超了，就要阻塞go的执行。第一时间想到的就是使用通道。每次执行的go之前向通道写入值，直到通道满的时候就阻塞了，

package main

import "fmt"

var ch chan int

func elegance(){

<-ch

fmt.Println("the ch value receive",ch)

}

func main(){

ch = make(chan int,5)

for i:=0;i<10;i++{

ch <-1

fmt.Println("the ch value send",ch)

go elegance()

fmt.Println("the result i",i)

}

}

运行:

> go run goroutine.go

the ch value send 0xc00009c000

the result i 0

the ch value send 0xc00009c000

the result i 1

the ch value send 0xc00009c000

the result i 2

the ch value send 0xc00009c000

the result i 3

the ch value send 0xc00009c000

the result i 4

the ch value send 0xc00009c000

the result i 5

the ch value send 0xc00009c000

the ch value receive 0xc00009c000

the result i 6

the ch value receive 0xc00009c000

the ch value send 0xc00009c000

the result i 7

the ch value send 0xc00009c000

the result i 8

the ch value send 0xc00009c000

the result i 9

the ch value send 0xc00009c000

the ch value receive 0xc00009c000

the ch value receive 0xc00009c000

the ch value receive 0xc00009c000

the result i 10

the ch value send 0xc00009c000

the result i 11

the ch value send 0xc00009c000

the result i 12

the ch value send 0xc00009c000

the result i 13

the ch value send 0xc00009c000

the ch value receive 0xc00009c000

the ch value receive 0xc00009c000

the ch value receive 0xc00009c000

the ch value receive 0xc00009c000

the result i 14

the ch value receive 0xc00009c000

> go run goroutine.go

the ch value send 0xc00007e000

the result i 0

the ch value send 0xc00007e000

the result i 1

the ch value send 0xc00007e000

the result i 2

the ch value send 0xc00007e000

the result i 3

the ch value send 0xc00007e000

the ch value receive 0xc00007e000

the result i 4

the ch value send 0xc00007e000

the ch value receive 0xc00007e000

the result i 5

the ch value send 0xc00007e000

the ch value receive 0xc00007e000

the result i 6

the ch value send 0xc00007e000

the result i 7

the ch value send 0xc00007e000

the ch value receive 0xc00007e000

the ch value receive 0xc00007e000

the ch value receive 0xc00007e000

the result i 8

the ch value send 0xc00007e000

the result i 9

这样每次同时运行的goroutine就被限制为5个了。但是新的问题于是就出现了，因为并不是所有的goroutine都执行完了，在main函数退出之后，还有一些goroutine没有执行完就被强制结束了。这个时候我们就需要用到sync.WaitGroup。使用WaitGroup等待所有的goroutine退出。

package main

import (

"fmt"

"runtime"

"sync"

"time"

)

// Pool Goroutine Pool

type Pool struct {

queue chan int

wg \*sync.WaitGroup

}

// New 新建一个协程池

func NewPool(size int) \*Pool{

if size <=0{

size = 1

}

return &Pool{

queue:make(chan int,size),

wg:&sync.WaitGroup{},

}

}

// Add 新增一个执行

func (p \*Pool)Add(delta int){

// delta为正数就添加

for i :=0;i<delta;i++{

p.queue <-1

}

// delta为负数就减少

for i:=0;i>delta;i--{

<-p.queue

}

p.wg.Add(delta)

}

// Done 执行完成减一

func (p \*Pool) Done(){

<-p.queue

p.wg.Done()

}

// Wait 等待Goroutine执行完毕

func (p \*Pool) Wait(){

p.wg.Wait()

}

func main(){

// 这里限制5个并发

pool := NewPool(5)

fmt.Println("the NumGoroutine begin is:",runtime.NumGoroutine())

for i:=0;i<20;i++{

pool.Add(1)

go func(i int) {

time.Sleep(time.Second)

fmt.Println("the NumGoroutine continue is:",runtime.NumGoroutine())

pool.Done()

}(i)

}

pool.Wait()

fmt.Println("the NumGoroutine done is:",runtime.NumGoroutine())

}

运行:

the NumGoroutine begin is: 1

the NumGoroutine continue is: 6

the NumGoroutine continue is: 7

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 6

the NumGoroutine continue is: 3

the NumGoroutine continue is: 2

the NumGoroutine done is: 1

其中，Go的GOMAXPROCS默认值已经设置为CPU的核数， 这里允许我们的Go程序充分使用机器的每一个CPU,最大程度的提高我们程序的并发性能。runtime.NumGoroutine函数在被调用后，会返回系统中的处于特定状态的Goroutine的数量。这里的特指是指Grunnable\Gruning\Gsyscall\Gwaition。处于这些状态的Groutine即被看做是活跃的或者说正在被调度。

这里需要注意下：垃圾回收所在Groutine的状态也处于这个范围内的话，也会被纳入该计数器。

## 28. Channel是同步的还是异步的.

Channel是异步进行的。

channel存在3种状态：

nil，未初始化的状态，只进行了声明，或者手动赋值为nil

active，正常的channel，可读或者可写

closed，已关闭，千万不要误认为关闭channel后，channel的值是nil

## 29. 说一下异步和非阻塞的区别?

异步和非阻塞的区别:

异步：调用在发出之后，这个调用就直接返回，不管有无结果；异步是过程。

非阻塞：关注的是程序在等待调用结果（消息，返回值）时的状态，指在不能立刻得到结果之前，该调用不会阻塞当前线程。

同步和异步的区别：

步：一个服务的完成需要依赖其他服务时，只有等待被依赖的服务完成后，才算完成，这是一种可靠的服务序列。要么成功都成功，失败都失败，服务的状态可以保持一致。

异步：一个服务的完成需要依赖其他服务时，只通知其他依赖服务开始执行，而不需要等待被依赖的服务完成，此时该服务就算完成了。被依赖的服务是否最终完成无法确定，一次它是一个不可靠的服务序列。

消息通知中的同步和异步：

同步：当一个同步调用发出后，调用者要一直等待返回消息（或者调用结果）通知后，才能进行后续的执行。

异步：当一个异步过程调用发出后，调用者不能立刻得到返回消息（结果）。在调用结束之后，通过消息回调来通知调用者是否调用成功。

阻塞与非阻塞的区别：

阻塞：阻塞调用是指调用结果返回之前，当前线程会被挂起，一直处于等待消息通知，不能够执行其他业务,函数只有在得到结果之后才会返回。

非阻塞：非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

同步与异步是对应的，它们是线程之间的关系，两个线程之间要么是同步的，要么是异步的。

阻塞与非阻塞是对同一个线程来说的，在某个时刻，线程要么处于阻塞，要么处于非阻塞。

阻塞是使用同步机制的结果，非阻塞则是使用异步机制的结果。

## 30. Log包线程安全吗？

Golang的标准库提供了log的机制，但是该模块的功能较为简单（看似简单，其实他有他的设计思路）。在输出的位置做了线程安全的保护。

## 31. Goroutine和线程的区别?

从调度上看，goroutine的调度开销远远小于线程调度开销。

OS的线程由OS内核调度，每隔几毫秒，一个硬件时钟中断发到CPU，CPU调用一个调度器内核函数。这个函数暂停当前正在运行的线程，把他的寄存器信息保存到内存中，查看线程列表并决定接下来运行哪一个线程，再从内存中恢复线程的注册表信息，最后继续执行选中的线程。这种线程切换需要一个完整的上下文切换：即保存一个线程的状态到内存，再恢复另外一个线程的状态，最后更新调度器的数据结构。某种意义上，这种操作还是很慢的。

Go运行的时候包涵一个自己的调度器，这个调度器使用一个称为一个M:N调度技术，m个goroutine到n个os线程（可以用GOMAXPROCS来控制n的数量），Go的调度器不是由硬件时钟来定期触发的，而是由特定的go语言结构来触发的，他不需要切换到内核语境，所以调度一个goroutine比调度一个线程的成本低很多。

从栈空间上，goroutine的栈空间更加动态灵活。

每个OS的线程都有一个固定大小的栈内存，通常是2MB，栈内存用于保存在其他函数调用期间哪些正在执行或者临时暂停的函数的局部变量。这个固定的栈大小，如果对于goroutine来说，可能是一种巨大的浪费。作为对比goroutine在生命周期开始只有一个很小的栈，典型情况是2KB, 在go程序中，一次创建十万左右的goroutine也不罕见（2KB\*100,000=200MB）。而且goroutine的栈不是固定大小，它可以按需增大和缩小，最大限制可以到1GB。

goroutine没有一个特定的标识。

在大部分支持多线程的操作系统和编程语言中，线程有一个独特的标识，通常是一个整数或者指针，这个特性可以让我们构建一个线程的局部存储，本质是一个全局的map，以线程的标识作为键，这样每个线程可以独立使用这个map存储和获取值，不受其他线程干扰。

goroutine中没有可供程序员访问的标识，原因是一种纯函数的理念，不希望滥用线程局部存储导致一个不健康的超距作用，即函数的行为不仅取决于它的参数，还取决于运行它的线程标识。

## 32. 滑动窗口的概念以及应用?

滑动窗口概念不仅存在于数据链路层，也存在于传输层，两者有不同的协议，但基本原理是相近的。其中一个重要区别是，一个是针对于帧的传送，另一个是字节数据的传送。

滑动窗口（Sliding window）是一种流量控制技术。早期的网络通信中，通信双方不会考虑网络的拥挤情况直接发送数据。由于大家不知道网络拥塞状况，同时发送数据，导致中间节点阻塞掉包，谁也发不了数据，所以就有了滑动窗口机制来解决此问题。参见滑动窗口如何根据网络拥塞发送数据仿真视频。

滑动窗口协议是用来改善吞吐量的一种技术，即容许发送方在接收任何应答之前传送附加的包。接收方告诉发送方在某一时刻能送多少包（称窗口尺寸）。

CP中采用滑动窗口来进行传输控制，滑动窗口的大小意味着接收方还有多大的缓冲区可以用于接收数据。发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为0时，发送方一般不能再发送数据报，但有两种情况除外，一种情况是可以发送紧急数据，例如，允许用户终止在远端机上的运行进程。另一种情况是发送方可以发送一个1字节的数据报来通知接收方重新声明它希望接收的下一字节及发送方的滑动窗口大小。

## 33. 怎么做弹性扩缩容，原理是什么?

弹性伸缩（Auto Scaling）根据您的业务需求和伸缩策略，为您自动调整计算资源。您可设置定时、周期或监控策略，恰到好处地增加或减少CVM实例，并完成实例配置，保证业务平稳健康运行。在需求高峰期时，弹性伸缩自动增加CVM实例的数量，以保证性能不受影响；当需求较低时，则会减少CVM实例数量以降低成本。弹性伸缩既适合需求稳定的应用程序，同时也适合每天、每周、每月使用量不停波动的应用程序。

## 34. 让你设计一个web框架，你要怎么设计，说一下步骤.

## 35. 说一下中间件原理.

中间件（middleware）是基础软件的一大类，属于可复用软件的范畴。中间件处于操作系统软件与用户的应用软件的中间。中间件在操作系统、网络和数据库之上，应用软件的下层，总的作用是为处于自己上层的应用软件提供运行与开发的环境，帮助用户灵活、高效地开发和集成复杂的应用软件 IDC的定义是：中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源，中间件位于客户机服务器的操作系统之上，管理计算资源和网络通信。

中间件解决的问题是：

在中间件产生以前，应用软件直接使用操作系统、网络协议和数据库等开发，这些都是计算机最底层的东西，越底层越复杂，开发者不得不面临许多很棘手的问题，如操作系统的多样性，繁杂的网络程序设计、管理，复杂多变的网络环境，数据分散处理带来的不一致性问题、性能和效率、安全，等等。这些与用户的业务没有直接关系，但又必须解决，耗费了大量有限的时间和精力。于是，有人提出能不能将应用软件所要面临的共性问题进行提炼、抽象，在操作系统之上再形成一个可复用的部分，供成千上万的应用软件重复使用。这一技术思想最终构成了中间件这类的软件。中间件屏蔽了底层操作系统的复杂性，使程序开发人员面对一个简单而统一的开发环境，减少程序设计的复杂性，将注意力集中在自己的业务上，不必再为程序在不同系统软件上的移植而重复工作，从而大大减少了技术上的负担。

## 36. 怎么设计orm，让你写,你会怎么写?

## 37. 用过原生的http包吗？

## 38. 一个非常大的数组，让其中两个数想加等于1000怎么算?

## 39. 各个系统出问题怎么监控报警.

## 40. 常用测试工具，压测工具，方法?

goconvey,vegeta

## 41. 复杂的单元测试怎么测试，比如有外部接口mysql接口的情况

## 42. redis集群，哨兵，持久化，事务

## 43. mysql和redis区别是什么？

## 44. 高可用软件是什么？

## 45. 怎么搞一个并发服务程序？

## 46. 讲解一下你做过的项目，然后找问题问实现细节。

## 47. mysql事务说下。

## 48. 怎么做一个自动化配置平台系统？

## 49. grpc遵循什么协议？

## 50. grpc内部原理是什么？

## 51. http2的特点是什么，与http1.1的对比。

| HTTP1.1 | HTTP2 | QUIC |

| -------------------------- | ----------- | --------------------------- |

| 持久连接 | 二进制分帧 | 基于UDP的多路传输（单连接下） |

| 请求管道化 | 多路复用（或连接共享） | 极低的等待时延（相比于TCP的三次握手） |

| 增加缓存处理（新的字段如cache-control） | 头部压缩 | QUIC为 传输层 协议 ，成为更多应用层的高性能选择 |

| 增加Host字段、支持断点传输等（把文件分成几部分） | 服务器推送 | |

## 52.Go的调度原理.

Go调度器: M, P 和 G

## 53.Go语言实战笔记（十二）| Go goroutine

## 54.Golang调度器源码分析

## 55.Goroutine调度过程

## 56.go struct能不能比较

相同struct类型的可以比较

不同struct类型的不可以比较,编译都不过，类型不匹配

package main

import "fmt"

func main() {

type A struct {

a int

}

type B struct {

a int

}

a := A{1}

//b := A{1}

b := B{1}

if a == b {

fmt.Println("a == b")

}else{

fmt.Println("a != b")

}

}

// output

// command-line-arguments [command-line-arguments.test]

// ./.go:14:7: invalid operation: a == b (mismatched types A and B)

go defer（for defer）

## 57.Go 关键字 defer 的一些坑

## 58.select可以用于什么?

## 59.Go的select主要是处理多个channel的操作.

## 60.Go语言并发模型：使用 select

## 61.context包的用途是什么?

godoc: https://golang.org/pkg/context/

Go Context的踩坑经历

Go语言实战笔记（二十）| Go Context

## 62.client如何实现长连接?

TCP协议的KeepAlive机制与HeartBeat心跳包

## 63.HTTP Keep-Alive是什么？如何工作？

主协程如何等其余协程完再操作?

Go并发：利用sync.WaitGroup实现协程同步

Go语言重点笔记-深入了解sync.WaitGroup

slice，len，cap，共享，扩容.

map如何顺序读取?

可以通过sort中的排序包进行对map中的key进行排序

golang:使用 sort 来排序

package main

import (

"fmt"

"sort"

)

func main() {

var m = map[string]int{

"hello": 0,

"morning": 1,

"my": 2,

"girl": 3,

}

var keys []string

for k := range m {

keys = append(keys, k)

}

sort.Strings(keys)

for \_, k := range keys {

fmt.Println("Key:", k, "Value:", m[k])

}

}

实现set

根据go中map的keys的无序性和唯一性，可以将其作为set

golang实现set集合,变相实现切片去重

实现消息队列（多生产者，多消费者）

根据Goroutine和channel的读写可以实现消息队列，

golang channel多生产者和多消费者实例

大文件排序

【算法】对一个20GB大的文件排序

## 64.基本排序，哪些是稳定的

选择排序、快速排序、希尔排序、堆排序不是稳定的排序算法，

冒泡排序、插入排序、归并排序和基数排序是稳定的排序算法

稳定排序和不稳定排序

Http get跟head

get:获取由Request-URI标识的任何信息(以实体的形式)，如果Request-URI引用某个数据处理过程，则应该以它产生的数据作为在响应中的实体，而不是该过程的源代码文本，除非该过程碰巧输出该文本。

head: 除了服务器不能在响应中返回消息体，HEAD方法与GET相同。用来获取暗示实体的元信息，而不需要传输实体本身。常用于测试超文本链接的有效性、可用性和最近的修改。

## 65.Http介绍

Http 401,403

401 Unauthorized： 该HTTP状态码表示认证错误，它是为了认证设计的，而不是为了授权设计的。收到401响应，表示请求没有被认证—压根没有认证或者认证不正确—但是请重新认证和重试。（一般在响应头部包含一个WWW-Authenticate来描述如何认证）。通常由web服务器返回，而不是web应用。从性质上来说是临时的东西。（服务器要求客户端重试）

403 Forbidden：该HTTP状态码是关于授权方面的。从性质上来说是永久的东西，和应用的业务逻辑相关联。它比401更具体，更实际。收到403响应表示服务器完成认证过程，但是客户端请求没有权限去访问要求的资源。

总的来说，401 Unauthorized响应应该用来表示缺失或错误的认证；403 Forbidden响应应该在这之后用，当用户被认证后，但用户没有被授权在特定资源上执行操作。

HTTP响应码403 Forbidden和401 Unauthorized对比

## 67.Http keep-alive

## 68.Http能不能一次连接多次请求，不等后端返回

## 69.TCP 和 UDP 有什么区别,适用场景

TCP 是面向连接的，UDP 是面向无连接的；故 TCP 需要建立连接和断开连接，UDP 不需要。

TCP 是流协议，UDP 是数据包协议；故 TCP 数据没有大小限制，UDP 数据报有大小限制（UDP 协议本身限制、数据链路层的 MTU、缓存区大小）。

TCP 是可靠协议，UDP 是不可靠协议；故 TCP 会处理数据丢包重发以及乱序等情况，UDP 则不会处理。

## 70.UDP 的特点及使用场景:

UDP 不提供复杂的控制机制，利用 IP 提供面向无连接的通信服务，随时都可以发送数据，处理简单且高效，经常用于以下场景：

包总量较小的通信（DNS、SNMP）

视频、音频等多媒体通信（即时通信）

广播通信

## 71.TCP 的特点及使用场景:

相对于 UDP，TCP 实现了数据传输过程中的各种控制，可以进行丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。

在对可靠性要求较高的情况下，可以使用 TCP，即不考虑 UDP 的时候，都可以选择 TCP。

iOS 知识点 TCP UDP 有什么区别？TCP 为什么要三次握手，四次挥手？

## 72.time-wait的作用

TCP/IP状态图的TIME\_WAIT作用

## 73.数据库如何建索引

正确合理的建立MySQL数据库索引

## 74.孤儿进程，僵尸进程

孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

## 75.死锁条件，如何避免

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

git文件版本，使用顺序，merge跟rebase

## 76.通常一般会用到哪些数据结构?

## 77.链表和数组相比, 有什么优缺点?

## 78.如何判断两个无环单链表有没有交叉点?

## 79.如何判断一个单链表有没有环, 并找出入环点?

## 80.描述一下 TCP 四次挥手的过程中

## 81.TCP 有哪些状态?

## 82.TCP 的 LISTEN 状态是什么?

## 83.TCP 的 CLOSE\_WAIT 状态是什么?

## 84.建立一个 socket 连接要经过哪些步骤?

## 85.常见的 HTTP 状态码有哪些?

## 86.301和302有什么区别?

## 87.504和500有什么区别?

## 88.HTTPS 和 HTTP 有什么区别?

## 89.算法题: 手写一个快速排序

快速排序:

func main() {

var arr = []int{19,8,16,15,23,34,6,3,1,0,2,9,7}

quickAscendingSort(arr, 0, len(arr)-1)

fmt.Println("quickAscendingSort:",arr)

quickDescendingSort(arr, 0, len(arr)-1)

fmt.Println("quickDescendingSort:",arr)

}

//升序

func quickAscendingSort(arr []int, start, end int) {

if (start < end) {

i, j := start, end

key := arr[(start + end)/2]

for i <= j {

for arr[i] < key {

i++

}

for arr[j] > key {

j--

}

if i <= j {

arr[i], arr[j] = arr[j], arr[i]

i++

j--

}

}

if start < j {

quickAscendingSort(arr, start, j)

}

if end > i {

quickAscendingSort(arr, i, end)

}

}

}

//降序

func quickDescendingSort(arr []int, start, end int) {

if (start < end) {

i, j := start, end

key := arr[(start + end)/2]

for i <= j {

for arr[i] > key {

i++

}

for arr[j] < key {

j--

}

if i <= j {

arr[i], arr[j] = arr[j], arr[i]

i++

j--

}

}

if start < j {

quickDescendingSort(arr, start, j)

}

if end > i {

quickDescendingSort(arr, i, end)

}

}

}

## 90.Golang 里的逃逸分析是什么？怎么避免内存逃逸？

## 91.配置中心如何保证一致性？

## 92.Golang 的GC触发时机是什么?

## 93.Redis 里数据结构的实现熟悉吗?

## 94.Etcd的Raft一致性算法原理?

## 95.微服务概念.

## 96.SLB原理.

## 97.分布式一直性原则.

## 98.如何保证服务宕机造成的分布式服务节点处理问题?

## 服务发现怎么实现的.

## 第1题什么是golang?

go是一个开源的编程语言，由谷歌开发的。这门语言是设计用来做系统级的编程的。

## 第2题为什么要用golang?

简单点说就是go是一个开源的编程语言，它很容易用来编写简单, 可靠，有效的软件。

## 第3题是谁开发了go语言？

Robert Griesemer, Rob Pike, Ken Thompson.

## 第4题，go编程中的packages是什么？

Go语言的最小集合就是package，它的程序入口是main package。

## 第5题, Go语言是否支持generic编程?

不支持。

泛型编程（Generic Programming）最初提出时的动机很简单直接：发明一种语言机制，能够帮助实现一个通用的标准容器库。所谓通用的标准容器库，就是要能够做到，比如用一个List类存放所有可能类型的对象这样的事；泛型编程让你编写完全一般化并可重复使用的算法，其效率与针对某特定数据类型而设计的算法相同。泛型即是指具有在多种数据类型上皆可操作的含义，与模板有些相似。STL巨大，而且可以扩充，它包含很多计算机基本算法和数据结构，而且将算法与数据结构完全分离，其中算法是泛型的，不与任何特定数据结构或对象类型系在一起。

## 第6题, go语言是大小写敏感的编程语言吗？

是的。

## 第7题，go语言里面字符串的语法是怎样的？

go语言里有两种字符串语法。

一种是原始字符串语法raw string literals。使用back quotes来包含一系列的字符集。支持多行。

另一种是interpreted string literals。用双引号括起来。不支持多行。

Go语言的字符串类型string在本质上就与其他语言的字符串类型不同：

Java的String、C++的std::string以及Python3的str类型都只是定宽字符序列

Go语言的字符串是一个用UTF-8编码的变宽字符序列，它的每一个字符都用一个或多个字节表示

即：一个Go语言字符串是一个任意字节的常量序列。

Golang的双引号和反引号都可用于表示一个常量字符串，不同在于：

双引号用来创建可解析的字符串字面量(支持转义，但不能用来引用多行)

反引号用来创建原生的字符串字面量，这些字符串可能由多行组成(不支持任何转义序列)，原生的字符串字面量多用于书写多行消息、HTML以及正则表达式

而单引号则用于表示Golang的一个特殊类型：rune，类似其他语言的byte但又不完全一样，是指：码点字面量（Unicode code point），不做任何转义的原始内容。

根据我找到的资料以及碰到的情况来看， Go语言的单引号一般用来表示「rune literal」 ，即——码点字面量。

### Golang 单引号、双引号、反引号

名词

单引号 single quotes

双引号 double quotes

反引号 back quotes

说明

Go语言的字符串是一个用UTF-8编码的变宽字符序列，它的每一个字符都用一个或多个字节表示 。

区别

单引号 不能用来表示字符串

双引号 可解析的字符串字面量 (支持转义，但不能用来引用多行)；

反引号 原生的字符串字面量 ，支持多行，不支持转义, 多用于书写多行消息、HTML以及正则表达式。

## 第8题, Go语言的work space是指什么?

Work space用来包含go语言的代码，有三个主要的文件目录, src, pkg, bin。

## 第9题，go语言中布尔类型的缺省值是什么？

false

## 第10题, Gopath环境变量是什么?

Gopath来指定工作空间的位置。在做go语言编程的时候，这个变量设置是必须的。

## 第11题，go语言编程的好处是什么？

编译和运行都很快。

在语言层级支持并行操作。

有垃圾处理器。

内置字符串和maps。

函数是go语言的最基本编程单位。

## 第12题，go语言里比较常用的功能模块。

Container, container list, container heap.

Web server, net/http

Cryptography, Crypto/md5, crypto/sha1.

Compression, compress/gzip.

Database, database/sql.

## 第13题，goroutine是什么？

一个goroutine是可以跟其他函数并行运行的函数。如果想停止一个goroutine，你可以通过一个signal channel传值进去。

## 第14题，如何写多行的字符串？

raw string literal

## 第15题, go语言里的break是干什么用的？

Break是用来终止当前的for loop或者switch的。

## 第16题，continue的作用是什么？

Continue就是跳过当前的运行单元。不会终止当前的loop.

## 第17题，goto是干什么的？

Goto 是为了跳转。

## 第18题，for loop如何使用？

For 有三种情况，一种是condition，一种是range, 最后一种是init, condition, increment。

## 第19题在go语言中如何创建一个函数？

使用func functionname ([Parameter list]) [Return types]

## 第20题, 解释一下go语言中的静态类型声明。

静态类型声明是告诉编译器不需要太多的关注这个变量的细节。静态变量的声明，只是针对于编译的时候, 在连接程序的时候，编译器还要对这个变量进行实际的声明。

## 第21题, 解释一下go语言中的动态类型声明。

动态类型声明是指go语言根据赋值的类型对这个变量的类型进行设定。

## 第22题, 如何打印出一个变量的类型?

%T

## 第23题, 什么是指针?

指针是用来存储变量地址的类型。

## 第24题, 指针是如何表示的?

使用星号加上数据类型。

取地址符&

取值符\*

## 第25题， := 短声明是不是只能用在一个函数里面?

是。

## 第26题， 如何不打印来格式化一个字符串?

fmt.Sprintf

## 第27题， go语言的语义是什么类型?

Go 语言的语意遵循的是extended backus-naur form即延伸的巴克斯.诺尔形式.

## 第28题, go语言是否支持继承?

不支持。

## 第29题，go语言是否支持操作符的重载?

不支持。

## 第30题，go语言是否支持方法重载?

不支持。

## 第31题，go语言是否支持指针运算?

不支持。

## 第32题，如下代码的输出是什么？

package main

import "fmt"

const (

i=7

j

k

)

func main(){

fmt.Println(i,j,k)

}

答案是777.

## 第33题，go的接口是什么？

在go语言中，interface也就是接口，被用来指定一个对象。接口具有下面的要素:

一系列的方法。

具体应用中并用来表示某个数据类型。

在go中使用interface来实现多态。

## 第34题， Go语言里面的类型断言是怎么回事?

类型断言是用来从一个接口里面读取数值给一个具体的类型变量。

类型转换是指转换两个不相同的数据类型。

## 第35题， Go语言里面的方法是怎么回事?

Go语言里面的方法，实际上就是某个函数, 在这个函数中指定一个receiver，在调用这个函数的时候使用这个receiver用dot操作符。

## 第36题, go语言中局部变量的缺省值是什么?

局部变量的缺省值是与这个类型相关的0值。

## 第37题， 全局变量的缺省值是什么?

全局变量的缺省值是与这个类型相关的零值。

## 第38题， 指针的缺省值是什么?

指针的缺省值为nil。

## 第39题, 如何在运行时检查一个变量的类型?

switch type.

## 第40题, 在使用go routine时是否推荐使用全局变量?

不推荐。

## 第41题, 模块化编程是怎么回事?

模块化编程是指把一个大的程序分解成几个小的程序。这么做的目的是为了减少程序的复杂度，易于维护，并且达到最高的效率。

# 算法

# 字节

## 网络

### 熟悉Internet常用协议

### HTTP

### TCP/IP

## RESTful规范

## 缓存优化、RPC优化

## 面试注意

1、我们的技术面试会涉及到算法、数据结构和系统设计，面试前需要你做的准备具体如下：​

提前温习软件工程的核心概念，例如: 如何从一个需求落实到一个系统设计，如何衡量两个不同设计的好坏，如何在各种限制下（人员、时间、资源等）选择其中更合适的设计，以及提升该设计的可拓展性等。​

在白板上练习算法题目，写出清晰、简洁、bug free的代码,，并衡量时间和空间复杂度以及可能存在的副作用。​

我们鼓励即时跟面试官沟通你的想法，一个好的解法往往是思维逻辑的展现，所以与面试官沟通思考的过程是非常重要的，这样在沟通的过程中你也能拿到更多关于问题本身的信息。​

尝试用不同的方法，思路或数据结构去解决同一个问题，并且衡量不同解法之间的优劣。​

我们需要是可执行的代码，而不是伪代码。​

我们需要你能掌握面试岗位对应的技术领域相关知识。​

2、以下是准备面试的资料/网站，包括但不限于：​

《Cracking the Coding Interview》​

https://leetcode.com/​

http://highscalability.com/​

【温馨提示】​

如需修改面试时间，请提前1天与HR电话联系，钟子红18879759823​

请在一个网络畅通且安静的场所进行面试，提前10分钟调试好电脑麦克风和摄像头​

点击视频链接之后，如果超过面试约定时间10分钟面试官仍未上线，请与HR联系​

面试过程中，面试官可能通过手机、电脑查看你的简历，记录你回答的要点​

【关于团队】​

我们致力于打造企业高速发展的引擎工具，帮助更多优秀企业成长，同时也致力于推动全球企业办公模式的变革。团队目前已推出产品飞书，飞书是一站式企业沟通与协作平台，整合聊天、日历、在线文档、云空间、工作台等功能于一体，立志打造高效的办公方式，加速企业成长。​

​

团队成员来自全球互联网与科技行业的顶尖企业，拥有丰富的相关行业经验，我们在北京、深圳、上海、广州、杭州、武汉、成都及硅谷均有团队分支。希望优秀的你也成为这个年轻团队的一员，和我们一起做有挑战的事。

基础知识：不同方向的技术基础知识有些差别，建议人选巩固一下操作系统、网络相关的知识，还 有一些其他学校里学过的，也建议复习一下。基础知识会根据大家工作年限的不同，而有所侧重， 面试问到的占比也会有所调整。

编码能力：人选在拿到编码题的时候，一定要先理解题意，抓住重点，想好整体思路之后再开始。 平时一定要多去github之类的上面看看，练习一下自己的代码书写速度，有良好的代码风格。

算法和数据结构：众所周知，字节跳动的面试是要考算法的，题目不多，但要能根据具体情况给出 多种高效解法。其实对于应届生来说，有些经常打比赛的，可能做起来会更上手些。刷题网站：ht tps://leetcode-cn.com/

项目经验和实操：建议人选把自己简历上写的项目都仔细梳理一遍，特别是自己负责的模块要能够 清晰掌握。既能了解项目全貌，也能说出技术细节点。面试官基本会问到你答不上来才会结束。建 议人选可以提前想一下有没有哪些项目自己是觉得比较亮眼的，面试过程中可以主动向面试官说。

## Redis 专题

什么是Redis及快速理解Redis的使用

Redis解决的问题及Redis的特性

Redis的应用场景及正确安装与启动

### redis八大应用场景

1、缓存

缓存现在几乎是所有中大型网站都在用的必杀技，合理的利用缓存不仅能够提升网站访问速度，还能大大降低数据库的压力。Redis提供了键过期功能，也提供了灵活的键淘汰策略，所以，现在Redis用在缓存的场合非常多。

2、排行榜

很多网站都有排行榜应用的，如京东的月度销量榜单、商品按时间的上新排行榜等。Redis提供的有序集合数据类构能实现各种复杂的排行榜应用。

3、计数器

什么是计数器，如电商网站商品的浏览量、视频网站视频的播放数等。为了保证数据实时效，每次浏览都得给+1，并发量高时如果每次都请求数据库操作无疑是种挑战和压力。Redis提供的incr命令来实现计数器功能，内存操作，性能非常好，非常适用于这些计数场景。

4、分布式会话

集群模式下，在应用不多的情况下一般使用容器自带的session复制功能就能满足，当应用增多相对复杂的系统中，一般都会搭建以Redis等内存数据库为中心的session服务，session不再由容器管理，而是由session服务及内存数据库管理。

5、分布式锁

在很多互联网公司中都使用了分布式技术，分布式技术带来的技术挑战是对同一个资源的并发访问，如全局ID、减库存、秒杀等场景，并发量不大的场景可以使用数据库的悲观锁、乐观锁来实现，但在并发量高的场合中，利用数据库锁来控制资源的并发访问是不太理想的，大大影响了数据库的性能。可以利用Redis的setnx功能来编写分布式的锁，如果设置返回1说明获取锁成功，否则获取锁失败，实际应用中要考虑的细节要更多。

6、社交网络

点赞、踩、关注/被关注、共同好友等是社交网站的基本功能，社交网站的访问量通常来说比较大，而且传统的关系数据库类型不适合存储这种类型的数据，Redis提供的哈希、集合等数据结构能很方便的的实现这些功能。

7、最新列表

Redis列表结构，LPUSH可以在列表头部插入一个内容ID作为关键字，LTRIM可用来限制列表的数量，这样列表永远为N个ID，无需查询最新的列表，直接根据ID去到对应的内容页即可。

8、消息系统

消息队列是大型网站必用中间件，如ActiveMQ、RabbitMQ、Kafka等流行的消息队列中间件，主要用于业务解耦、流量削峰及异步处理实时性低的业务。Redis提供了发布/订阅及阻塞队列功能，能实现一个简单的消息队列系统。另外，这个不能和专业的消息中间件相比。

Redis配置、启动、操作、关闭及版本选择

Redis的数据结构—字符串

Redis全局命令及数据库管理

Redis设计订单应用场景

你应该知道的Redis缓存雪崩

什么是缓存穿透

### Redis重启时加载AOF与RDB的顺序

Redis的持久化两种持久化方式RDB和AOF

有关于redis持久化的好文章比比皆是，在这里我就不仔细阐述了，大家可以出去搜索一下，我在这里做一个大致的诠释：

#### 1、RDB

RDB(redis database)，可以理解为快照/内存快照，RDB持久化过程是将当前进程中的数据生成快照存储到硬盘中，主要分为手动触发和自动触发。

#### 2、AOF

AOF（append only file），以日志的方式记录每次写命令，服务重启的时候重新执行AOF文件中的命令来恢复内存数据。因为解决了数据持久化实时性的问题，所以目前AOF是Redis持久化的主流方式。

#### 3、RDB和AOF的差异

##### RDB

优点：

1.RDB文件小，非常适用于定时备份，用于灾难恢复。

2.Redis加载RDB文件的速度比AOF快很多，因为RDB文件中直接存储的时内存数据，而AOF文件中存储的是一条条命令。

3.如果业务对数据完整性和一致性要求不高，RDB是很好的选择。

缺点：

1.一定时间做备份，redis意外down掉，可能最后一次数据没有到磁盘上

2.Fork的时候，内存中数据被copy了一份，2倍的膨胀性在内存中需要考虑

3.备份时占用内存，因为Redis 在备份时会独立创建一个子进程，将数据写入到一个临时文件（此时内存中的数据是原来的两倍哦），最后再将临时文件替换之前的备份文件。

##### AOF

优点：

1.数据的完整性和一致性更高

缺点：

1.因为AOF记录的内容多，文件会越来越大，数据恢复也会越来越慢。

##### 总结：

1.RDB持久化基于内存快照存储二进制文件，AOF持久化基于写命令存储文本文件。

2.RDB文件采用了压缩算法，比较小；AOF文件随着命令的叠加会越来越大，Redis提供了AOF重写来压缩AOF文件。

3.恢复RDB文件的速度比AOF文件快很多。

4.RDB持久化方式实时性不好，所以AOF持久化更主流。

5.合理的使用AOF的同步策略，理论上不会丢失大量的数据。

#### 4、Redis重启时加载持久化文件的顺序

Redis重启的时候优先加载AOF文件，如果AOF文件不存在再去加载RDB文件。

如果AOF文件和RDB文件都不存在，那么直接启动。

不论加载AOF文件还是RDB文件，只要发生错误都会打印错误信息，并且启动失败。

五种数据结构类型回顾

### Redis慢查询分析

慢查询，顾名思义就是比较慢的查询，但是究竟是哪里慢呢？首先，我们了解一下Redis命令执行的整个过程：

发送命令

命令排队

命令执行

返回结果

在慢查询的定义中，统计比较慢的时间段指的是命令执行这个步骤。没有慢查询，并不表示客户端没有超时问题，有可能网络传输有延迟，也有可能排队的命令比较多。

因为Redis中命令执行的排队机制，慢查询会导致其他命令的级联阻塞，所以当客户端出现请求超时的时候，需要检查该时间点是否有慢查询，从而分析出由于慢查询导致的命令级联阻塞。

什么是慢查询日志？

慢查询日志是Redis服务端在命令执行前后计算每条命令的执行时长，当超过某个阈值是记录下来的日志。日志中记录了慢查询发生的时间，还有执行时长、具体什么命令等信息，它可以用来帮助开发和运维人员定位系统中存在的慢查询。

如何获取慢查询日志？

可以使用slowlog get命令获取慢查询日志，在slowlog get后面还可以加一个数字，用于指定获取慢查询日志的条数，比如，获取3条慢查询日志：

> slowlog get 3

1) 1) (integer) 6107

2) (integer) 1616398930

3) (integer) 3109

4) 1) "config"

2) "rewrite"

2) 1) (integer) 6106

2) (integer) 1613701788

3) (integer) 36004

4) 1) "flushall"

3) 1) (integer) 6105

2) (integer) 1608722338

3) (integer) 20449

4) 1) "scan"

2) "0"

3) "MATCH"

4) "\*comment\*"

5) "COUNT"

6) "10000"

从上面的例子中，可以看出每一条慢查询日志都有4个属性组成：

唯一标识ID

命令执行的时间戳

命令执行时长

执行的命名和参数

如何获取慢查询日志的长度？

可以使用slowlog len命令获取慢查询日志的长度，比如：

> slowlog len

(integer) 121

在上例中，当前Redis中有121条慢查询日志。

如何清理慢查询日志？

可以使用slowlog reset命令清理慢查询日志，比如：

> slowlog len

(integer) 121

> slowlog reset

OK

> slowlog len

(integer) 0

怎么配置慢查询的参数？

正如上面提到的，慢查询需要如下两个配置：

命令执行时长的指定阈值。

存放慢查询日志的条数。

Redis对应提供了两个参数：slowlog-log-slower-than和slowlog-max-len，接下来我们详细介绍一下这两个参数。

slowlog-log-slower-than

slowlog-log-slower-than的作用是指定命令执行时长的阈值，执行命令的时长超过这个阈值时就会被记录下来。它的单位是微秒（1秒 = 1000毫秒 = 1000000微秒），默认是10000微秒。如果把slowlog-log-slower-than设置为0，将会记录所有命令到日志中。如果把slowlog-log-slower-than设置小于0，将会不记录任何命令到日志中。

在实际的生产环境中，需要根据Redis并发量来调整该配置。因为Redis采用单线程响应命令，如果命令执行时间在1000微秒以上，那么Redis最多可支撑OPS不到1000，所以对于高并发场景的Redis建议设置为1000微秒。

slowlog-max-len

slowlog-max-len的作用是指定慢查询日志最多存储的条数。实际上，Redis使用了一个列表存放慢查询日志，slowlog-max-len就是这个列表的最大长度。当一个新的命令满足满足慢查询条件时，被插入这个列表中。当慢查询日志列表已经达到最大长度时，最早插入的那条命令将被从列表中移出。比如，slowlog-max-len被设置为10，当有第11条命令插入时，在列表中的第1条命令先被移出，然后再把第11条命令放入列表。

记录慢查询是Redis会对长命令进行截断，不会大量占用大量内存。在实际的生产环境中，为了减缓慢查询被移出的可能和更方便地定位慢查询，建议将慢查询日志的长度调整的大一些。比如可以设置为1000以上。

慢查询极值设置讲解

慢查询原理及慢查询命令讲解

Redis上线应该做的事什么是织入。什么是织入应用的不同点？

Redis运行原理流程

手写Jedis实战

Redis 部分

Redis的应用场景

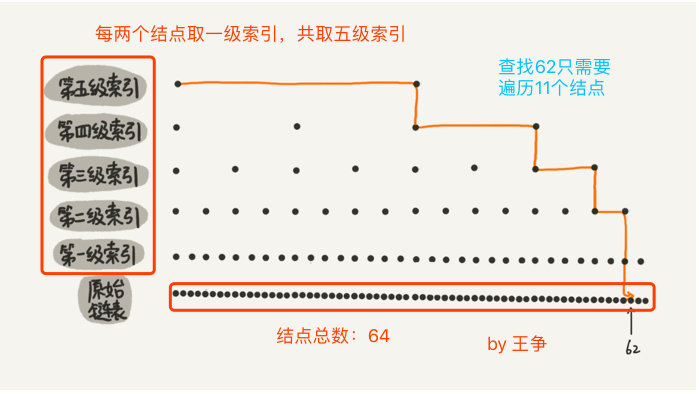
Redis支持的数据类型（必考）

### zset跳表的数据结构（必考）

跳表是一个随机化的数据结构，实质就是一种可以进行二分查找的有序链表。

跳表在原有的有序链表上面增加了多级索引，通过索引来实现快速查找。

跳表不仅能提高搜索性能，同时也可以提高插入和删除操作的性能。



跳表这个动态数据结构，不仅支持查找操作，还支持动态的插入、删除操作，而且插入、删除操作的时间复杂度也是 ○(㏒n)。

对于单纯的单链表，需要遍历每个结点来找到插入的位置。但是对于跳表来说，因为其查找某个结点的时间复杂度是 ○(㏒n)，所以这里查找某个数据应该插入的位置，时间复杂度也是 ○(㏒n)。

### Redis的数据过期策略（必考）

　　本文对Redis的过期机制简单的讲解一下

　　讲解之前我们先抛出一个问题，我们知道很多时候服务器经常会用到redis作为缓存，有很多数据都是临时缓存一下，可能用过之后很久都不会再用到了（比如暂存session，又或者只存放日行情股票数据）那么就会出现一下几个问题了

Redis会自己回收清理不用的数据吗？

如果能，那如何配置？

如果不能，如何防止数据累加后大量占用存储空间的问题？

　　之前一直接触Redis不是很深入，最近项目当中遇到一个需求场景，需要清空一些存放在Redis的数据，主要是通过一些时间进行过滤，删除那些不满足的数据，但是这样的工作每天都需要进行，那工作量就比较大了，而且每天都需要按时去手动清理，这样做也不切实际，后面发现Redis中有个设置时间过期的功能，即对存储在Redis数据库中的值可以设置一个过期时间。作为一个缓存数据库，这是非常实用的。这就是我们本文要讲到的Redis过期机制。其实这个机制运用的场景十分广泛，比如我们一般项目中的token或者一些登录信息，尤其是短信验证码都是有时间限制的，或者是限制请求次数，如果按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性能。

#### 一、设置过期时间

　　Redis对存储值的过期处理实际上是针对该值的键（key）处理的，即时间的设置也是设置key的有效时间。Expires字典保存了所有键的过期时间，Expires也被称为过期字段。

expire key time(以秒为单位)--这是最常用的方式

setex(String key, int seconds, String value)--字符串独有的方式

注：

　　1、除了字符串自己独有设置过期时间的方法外，其他方法都需要依靠expire方法来设置时间

　　2、如果没有设置时间，那缓存就是永不过期

　　3、如果设置了过期时间，之后又想让缓存永不过期，使用persist key

1、常用方式

一般主要包括4种处理过期方，其中expire都是以秒为单位，pexpire都是以毫秒为单位的。

1 EXPIRE key seconds　　//将key的生存时间设置为ttl秒

2 PEXPIRE key milliseconds　　//将key的生成时间设置为ttl毫秒

3 EXPIREAT key timestamp　　//将key的过期时间设置为timestamp所代表的的秒数的时间戳

4 PEXPIREAT key milliseconds-timestamp　　//将key的过期时间设置为timestamp所代表的的毫秒数的时间戳

备注：timestamp为unix时间戳（例如：timestamp=1499788800 表示将在2017.07.12过期）

1、2两种方式是设置一个过期的时间段，就是咱们处理验证码最常用的策略，设置三分钟或五分钟后失效，把分钟数转换成秒或毫秒存储到Redis中。

3、4两种方式是指定一个过期的时间 ，比如优惠券的过期时间是某年某月某日，只是单位不一样。

下面我们就以EXPIREAT为例子简单讲解下用法。

返回值

一个整数值1或0，如下：

如果成功地为该键设置了超时时间，返回 1

如果键不存在或无法设置超时时间，返回 0

语法

以下是以Redis的EXPIREAT命令的基本语法。

1 redis 127.0.0.1:6379> Expireat KEY\_NAME TIME\_IN\_UNIX\_TIMESTAMP

示例

首先，在Redis中创建一个键：akey，并在akey中设置一些值。

1 redis 127.0.0.1:6379> SET akey redis

2 OK

现在，为设置创建的键设置超时时间为60 秒。

复制代码

1 127.0.0.1:6379> SET akey redis

2 OK

3 127.0.0.1:6379> EXPIREAT akey 1393840000

4 (integer) 1

5 127.0.0.1:6379> EXISTS akey

6 (integer) 0

7 127.0.0.1:6379> SET akey redis

8 OK

9 127.0.0.1:6379> EXPIREAT akey 1493840000

10 (integer) 1

11 127.0.0.1:6379> EXISTS akey

12 (integer) 1

复制代码

其他三个用法类似，这里不逐一阐述

2、字符串独有方式

对字符串特殊处理的方式为SETEX命令，SETEX命令为指定的 key 设置值及其过期时间。如果 key 已经存在， SETEX 命令将会替换旧的值。

返回值

设置成功时返回 OK 。

语法

Redis Setex 命令基本语法如下：

redis 127.0.0.1:6379> SETEX KEY\_NAME TIMEOUT VALUE

示例

1 redis 127.0.0.1:6379> SETEX mykey 60 redis

2 OK

3 redis 127.0.0.1:6379> TTL mykey

4 60

5 redis 127.0.0.1:6379> GET mykey

6 "redis

#### 二、常用3种过期策略

定时删除

含义：在设置key的过期时间的同时，为该key创建一个定时器，让定时器在key的过期时间来临时，对key进行删除

优点：保证内存被尽快释放

缺点：

若过期key很多，删除这些key会占用很多的CPU时间，在CPU时间紧张的情况下，CPU不能把所有的时间用来做要紧的事儿，还需要去花时间删除这些key

定时器的创建耗时，若为每一个设置过期时间的key创建一个定时器（将会有大量的定时器产生），性能影响严重

没人用

惰性删除

含义：key过期的时候不删除，每次从数据库获取key的时候去检查是否过期，若过期，则删除，返回null。

优点：删除操作只发生在从数据库取出key的时候发生，而且只删除当前key，所以对CPU时间的占用是比较少的，而且此时的删除是已经到了非做不可的地步（如果此时还不删除的话，我们就会获取到了已经过期的key了）

缺点：若大量的key在超出超时时间后，很久一段时间内，都没有被获取过，那么可能发生内存泄露（无用的垃圾占用了大量的内存）

定期删除

含义：每隔一段时间执行一次删除(在redis.conf配置文件设置hz，1s刷新的频率)过期key操作

优点：

通过限制删除操作的时长和频率，来减少删除操作对CPU时间的占用--处理"定时删除"的缺点

定期删除过期key--处理"惰性删除"的缺点

缺点

在内存友好方面，不如"定时删除"

在CPU时间友好方面，不如"惰性删除"

难点

合理设置删除操作的执行时长（每次删除执行多长时间）和执行频率（每隔多长时间做一次删除）（这个要根据服务器运行情况来定了）

看完上面三种策略后可以得出以下结论：

定时删除和定期删除为主动删除：Redis会定期主动淘汰一批已过去的key

惰性删除为被动删除：用到的时候才会去检验key是不是已过期，过期就删除

惰性删除为redis服务器内置策略

定期删除可以通过：

第一、配置redis.conf 的hz选项，默认为10 （即1秒执行10次，100ms一次，值越大说明刷新频率越快，最Redis性能损耗也越大）

第二、配置redis.conf的maxmemory最大值，当已用内存超过maxmemory限定时，就会触发主动清理策略

注意：

上边所说的数据库指的是内存数据库，默认情况下每一台redis服务器有16个数据库（关于数据库的设置，看下边代码），默认使用0号数据库，所有的操作都是对0号数据库的操作，关于redis数据库的存储结构，查看 第八章 Redis数据库结构与读写原理

# 设置数据库数量。默认为16个库，默认使用DB 0，可以使用"select 1"来选择一号数据库

# 注意：由于默认使用0号数据库，那么我们所做的所有的缓存操作都存在0号数据库上，

# 当你在1号数据库上去查找的时候，就查不到之前set过得缓存

# 若想将0号数据库上的缓存移动到1号数据库，可以使用"move key 1"

databases 16

memcached只是用了惰性删除，而Redis同时使用了惰性删除与定期删除，这也是二者的一个不同点（可以看做是redis优于memcached的一点）

对于惰性删除而言，并不是只有获取key的时候才会检查key是否过期，在某些设置key的方法上也会检查（eg.setnx key2 value2：该方法类似于memcached的add方法，如果设置的key2已经存在，那么该方法返回false，什么都不做；如果设置的key2不存在，那么该方法设置缓存key2-value2。假设调用此方法的时候，发现redis中已经存在了key2，但是该key2已经过期了，如果此时不执行删除操作的话，setnx方法将会直接返回false，也就是说此时并没有重新设置key2-value2成功，所以对于一定要在setnx执行之前，对key2进行过期检查）

#### 三、Redis采用的过期策略

惰性删除 + 定期删除

惰性删除流程

在进行get或setnx等操作时，先检查key是否过期，

若过期，删除key，然后执行相应操作；

若没过期，直接执行相应操作

定期删除流程（简单而言，对指定个数个库的每一个库随机删除小于等于指定个数个过期key）

遍历每个数据库（就是redis.conf中配置的"database"数量，默认为16）

检查当前库中的指定个数个key（默认是每个库检查20个key，注意相当于该循环执行20次，循环体时下边的描述）

如果当前库中没有一个key设置了过期时间，直接执行下一个库的遍历

随机获取一个设置了过期时间的key，检查该key是否过期，如果过期，删除key

判断定期删除操作是否已经达到指定时长，若已经达到，直接退出定期删除。

#### 四、RDB对过期key的处理

过期key对RDB没有任何影响

从内存数据库持久化数据到RDB文件

持久化key之前，会检查是否过期，过期的key不进入RDB文件

从RDB文件恢复数据到内存数据库

数据载入数据库之前，会对key先进行过期检查，如果过期，不导入数据库（主库情况）

#### 五、AOF对过期key的处理

过期key对AOF没有任何影响

从内存数据库持久化数据到AOF文件：

当key过期后，还没有被删除，此时进行执行持久化操作（该key是不会进入aof文件的，因为没有发生修改命令）

当key过期后，在发生删除操作时，程序会向aof文件追加一条del命令（在将来的以aof文件恢复数据的时候该过期的键就会被删掉）

AOF重写

重写时，会先判断key是否过期，已过期的key不会重写到aof文件

Redis的LRU过期策略的具体实现

如何解决Redis缓存雪崩，缓存穿透问题

Redis的持久化机制（必考）

### Redis的管道pipeline

 Redis 的 pipeline(管道)功能在命令行中没有，但 redis 是支持 pipeline 的，而且在各个语言版的 client 中都有相应的实现。 由于网络开销延迟，就算 redis server 端有很强的处理能力，也会由于收到的 client 消息少，而造成吞吐量小。当 client 使用 pipelining 发送命令时，redis server 必须将部分请求放到队列中（使用内存），执行完毕后一次性发送结果；如果发送的命令很多的话，建议对返回的结果加标签，当然这也会增加使用的内存；

  Pipeline 在某些场景下非常有用，比如有多个 command 需要被“及时的”提交，而且他们对相应结果没有互相依赖，对结果响应也无需立即获得，那么 pipeline 就可以充当这种“批处理”的工具；而且在一定程度上，可以较大的提升性能，性能提升的原因主要是 TCP 连接中减少了“交互往返”的时间。

  不过在编码时请注意，pipeline 期间将“独占”链接，此期间将不能进行非“管道”类型的其他操作，直到 pipeline 关闭；如果你的 pipeline 的指令集很庞大，为了不干扰链接中的其他操作，你可以为 pipeline 操作新建 Client 链接，让 pipeline 和其他正常操作分离在2个 client 中。不过 pipeline 事实上所能容忍的操作个数，和 socket-output 缓冲区大小/返回结果的数据尺寸都有很大的关系；同时也意味着每个 redis-server 同时所能支撑的 pipeline 链接的个数，也是有限的，这将受限于 server 的物理内存或网络接口的缓冲能力。

（一）简介

  Redis 使用的是客户端-服务器（CS）模型和请求/响应协议的 TCP 服务器。这意味着通常情况下一个请求会遵循以下步骤：

客户端向服务端发送一个查询请求，并监听 Socket 返回，通常是以阻塞模式，等待服务端响应。

服务端处理命令，并将结果返回给客户端。

  Redis 客户端与 Redis 服务器之间使用 TCP 协议进行连接，一个客户端可以通过一个 socket 连接发起多个请求命令。每个请求命令发出后 client 通常会阻塞并等待 redis 服务器处理，redis 处理完请求命令后会将结果通过响应报文返回给 client，因此当执行多条命令的时候都需要等待上一条命令执行完毕才能执行。比如：

  这里写图片描述

  其执行过程如下图所示：

  这里写图片描述

  由于通信会有网络延迟，假如 client 和 server 之间的包传输时间需要0.125秒。那么上面的三个命令6个报文至少需要0.75秒才能完成。这样即使 redis 每秒能处理100个命令，而我们的 client 也只能一秒钟发出四个命令。这显然没有充分利用 redis 的处理能力。

  而管道（pipeline）可以一次性发送多条命令并在执行完后一次性将结果返回，pipeline 通过减少客户端与 redis 的通信次数来实现降低往返延时时间，而且 Pipeline 实现的原理是队列，而队列的原理是时先进先出，这样就保证数据的顺序性。 Pipeline 的默认的同步的个数为53个，也就是说 arges 中累加到53条数据时会把数据提交。其过程如下图所示：client 可以将三个命令放到一个 tcp 报文一起发送，server 则可以将三条命令的处理结果放到一个 tcp 报文返回。

  这里写图片描述

  需要注意到是用 pipeline 方式打包命令发送，redis 必须在处理完所有命令前先缓存起所有命令的处理结果。打包的命令越多，缓存消耗内存也越多。所以并不是打包的命令越多越好。具体多少合适需要根据具体情况测试。

## MySQL 专题

Mysql 的技术特点是什么

Mysql 服务器默认端口是什么：3306

### 与 Oracle 相比，Mysql 有什么优势？

Mysql 是开源软件、免费

操作简单、部署方便，用户可以根据应用的需求去定制数据库

Mysql 的引擎是插件式

### 如何区分 FLOAT 和 DOUBLE？

MySQL类型float double decimal的区别

float数值类型用于表示单精度浮点数值，而double数值类型用于表示双精度浮点数值，float和double都是浮点型，而decimal是定点型；

MySQL 浮点型和定点型可以用类型名称后加（M，D）来表示，M表示该值的总共长度，D表示小数点后面的长度，M和D又称为精度和标度，如float(7,4)的 可显示为-999.9999，MySQL保存值时进行四舍五入，如果插入999.00009，则结果为999.0001。

FLOAT和DOUBLE在不指 定精度时，默认会按照实际的精度来显示，而DECIMAL在不指定精度时，默认整数为10，小数为0。

请简洁描述 Mysql 中 InnoDB 支持的四种事务隔离级别名 称，以及逐级之间的区别？

### Mysql 驱动程序是什么？

数据库驱动

定义：数据库驱动是不同数据库开发商（比如oracle mysql等）为了某一种开发语言环境（比如java）能够实现数据库调用而开发的一个程序，

他的作用相当于一个翻译人员，将Java中对数据库的调用语言翻译成数据库自己的数据库语言，当然这个翻译（数据库驱动）是由各个开发商针对统一的接口自定义开发的。

### ~~如何使用 Unix shell 登录 Mysql？~~

### Mysql 如何优化 DISTINCT？

分析MySQL中优化distinct的技巧

这篇文章主要介绍了分析MySQL中优化distinct的技巧,主要是通过减少本地扫描的次数来进行优化的方法,需要的朋友可以参考下

java

有这样的一个需求：

select count(distinct nick) from user\_access\_xx\_xx;

这条sql用于统计用户访问的uv，由于单表的数据量在10G以上，即使在user\_access\_xx\_xx上加上nick的索引，

通过查看执行计划，也为全索引扫描，sql在执行的时候，会对整个服务器带来抖动；

root@db 09:00:12>select count(distinct nick) from user\_access;

+———————-+

| count(distinct nick) |

+———————-+

| 806934 |

+———————-+

1 row in set (52.78 sec)

执行一次sql需要花费52.78s，已经非常的慢了

现在需要换一种思路来解决该问题：

我们知道索引的值是按照索引字段升序的，比如我们对（nick，other\_column）两个字段做了索引，那么在索引中的则是按照nick，other\_column的升序排列：

我们现在的sql：select count(distinct nick) from user\_access;则是直接从nick1开始一条条扫描下来，直到扫描到最后一个nick\_n,

那么中间过程会扫描很多重复的nick，如果我们能够跳过中间重复的nick，则性能会优化非常多（在oracle中，这种扫描技术为loose index scan，但在5.1的版本中，mysql中还不能直接支持这种优化技术）：

20155893909060.jpg (532×255)

所以需要通过改写sql来达到伪loose index scan：

root@db 09:41:30>

select count(\*) from ( select distinct(nick) from user\_access)t ;

| count(\*) |

+———-+

| 806934 |

1 row in set (5.81 sec)

Sql中先选出不同的nick，最后在外面套一层，就可以得到nick的distinct值总和；

最重要的是在子查询中：select distinct(nick) 实现了上图中的伪loose index scan，优化器在这个时候的执行计划为Using index for group-by ，

需要注意的是mysql把distinct优化为group by，它首先利用索引来分组，然后扫描索引，对需要的nick只扫描一次；

两个sql的执行计划分别为：

优化写法：

root@db 09:41:10>explain select distinct(nick) from user\_access-> ;

+—-+————-+——————————+——-+—————+————-| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+—-+————-+——————————+——-+—————+————-

| 1 | SIMPLE | user\_access | range | NULL | ind\_user\_access\_nick | 67 | NULL | 2124695 | Using index for group-by |

+—-+————-+——————————+——-+—————+————-

原始写法：

root@db 09:42:55>explain select count(distinct nick) from user\_access;

+—-+————-+——————————+——-+—————+————-

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+—-+————-+——————————+——-+—————+————-

| 1 | SIMPLE | user\_access | index | NULL | ind\_user\_access | 177 | NULL | 19546123 | Using index |

### MySQL如何优化DISTINCT？

在许多情况下结合ORDER BY的DISTINCT需要一个临时表。

请注意因为DISTINCT可能使用GROUP BY，必须清楚MySQL如何使用所选定列的一部分的ORDER BY或HAVING子句中的列。

在大多数情况下，DISTINCT子句可以视为GROUP BY的特殊情况。例如，下面的两个查询是等效的：

SELECT DISTINCT c1, c2, c3 FROM t1 WHERE c1 > const;

SELECT c1, c2, c3 FROM t1 WHERE c1 > const GROUP BY c1, c2, c3;

由于这个等效性，适用于GROUP BY查询的优化也适用于有DISTINCT子句的查询。

### mysql DISTINCT 的实现与优化

DISTINCT实际上和GROUP BY的操作非常相似，只不过是在GROUP BY之后的每组中只取出一条记录而已。所以，DISTINCT的实现和GROUP BY的实现也基本差不多，没有太大的区别。同样可以通过松散索引扫描或者是紧凑索引扫描来实现，当然，在无法仅仅使用索引即能完成DISTINCT的时候，MySQL只能通过临时表来完成。但是，和GROUP BY有一点差别的是，DISTINCT并不需要进行排序。也就是说，在仅仅只是DISTINCT操作的Query如果无法仅仅利用索引完成操作的时候，MySQL会利用临时表来做一次数据的“缓存”，但是不会对临时表中的数据进行filesort操作。当然，如果我们在进行DISTINCT的时候还使用了GROUP BY并进行了分组，并使用了类似于MAX之类的聚合函数操作，就无法避免filesort了。

下面我们就通过几个简单的Query示例来展示一下DISTINCT的实现。

1.首先看看通过松散索引扫描完成DISTINCT的操作：

sky@localhost: example 11:03:41>EXPLAIN SELECT DISTINCT group\_id

-> FROM group\_message\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1SELECT\_type:SIMPLE

table:group\_message

type:range

possible\_keys:NULL

key: idx\_gid\_uid\_gc

key\_len:4ref: NULL

rows:10Extra: Using index for group-by

1 row in set (0.00sec)

我们可以很清晰的看到，执行计划中的Extra信息为“Usingindex for group-by”，这代表什么意思？为什么我没有进行GROUP BY操作的时候，执行计划中会告诉我这里通过索引进行了GROUP BY呢？其实这就是于DISTINCT的实现原理相关的，在实现DISTINCT的过程中，同样也是需要分组的，然后再从每组数据中取出一条返回给客户端。而这里的Extra信息就告诉我们，MySQL利用松散索引扫描就完成了整个操作。当然，如果MySQLQuery Optimizer要是能够做的再人性化一点将这里的信息换成“Using index for distinct”那就更好更容易让人理解了，呵呵。

2. 我们再来看看通过紧凑索引扫描的示例：

sky@localhost: example 11:03:53> EXPLAIN SELECT DISTINCT user\_id

->FROM group\_message

->WHERE group\_id = 2\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id:1SELECT\_type: SIMPLE

table:group\_message

type:ref

possible\_keys:idx\_gid\_uid\_gc

key:idx\_gid\_uid\_gc

key\_len:4ref: const

rows:4Extra: Using WHERE; Using index

1row in set (0.00 sec)

这里的显示和通过紧凑索引扫描实现GROUP BY也完全一样。实际上，这个Query的实现过程中，MySQL会让存储引擎扫描group\_id=2的所有索引键，得出所有的user\_id，然后利用索引的已排序特性，每更换一个user\_id的索引键值的时候保留一条信息，即可在扫描完所有gruop\_id=2的索引键的时候完成整个DISTINCT操作。

3.下面我们在看看无法单独使用索引即可完成DISTINCT的时候会是怎样：

sky@localhost: example 11:04:40> EXPLAIN SELECT DISTINCT user\_id

->FROM group\_message

->WHERE group\_id > 1 AND group\_id < 10\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id:1SELECT\_type: SIMPLE

table:group\_message

type:range

possible\_keys:idx\_gid\_uid\_gc

key:idx\_gid\_uid\_gc

key\_len:4ref: NULL

rows:32Extra: Using WHERE; Using index; Using temporary

1row in set (0.00 sec)

当MySQL无法仅仅依赖索引即可完成DISTINCT操作的时候，就不得不使用临时表来进行相应的操作了。但是我们可以看到，在MySQL利用临时表来完成DISTINCT的时候，和处理GROUP BY有一点区别，就是少了filesort。实际上，在MySQL的分组算法中，并不一定非要排序才能完成分组操作的，这一点在上面的GROUP BY优化小技巧中我已经提到过了。实际上这里MySQL正是在没有排序的情况下实现分组最后完成DISTINCT操作的，所以少了filesort这个排序操作。

4.最后再和GROUP BY结合试试看：

sky@localhost: example 11:05:06> EXPLAIN SELECT DISTINCT max(user\_id)

->FROM group\_message

->WHERE group\_id > 1 AND group\_id < 10

->GROUP BY group\_id\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id:1SELECT\_type: SIMPLE

table:group\_message

type:range

possible\_keys:idx\_gid\_uid\_gc

key:idx\_gid\_uid\_gc

key\_len:4ref: NULL

rows:32Extra: Using WHERE; Using index; Using temporary; Usingfilesort

1row in set (0.00 sec)

最后我们再看一下这个和GROUP BY一起使用带有聚合函数的示例，和上面第三个示例相比，可以看到已经多了filesort排序操作了，因为我们使用了MAX函数的缘故。

对于DISTINCT的优化，和GROUP BY基本上一致的思路，关键在于利用好索引，在无法利用索引的时候，确保尽量不要在大结果集上面进行DISTINCT操作，磁盘上面的IO操作和内存中的IO操作性能完全不是一个数量级的差距。

### mysql中关联查询和子查询那个更快

不一定，要综合索引情况、数据量等情况多方面考虑。

### 什么是通用 SQL 函数？

1、CONCAT(A, B) – 连接两个字符串值以创建单个字符串输出。通常用于将两个或多个字段合并为一个字段。

2、FORMAT(X, D)- 格式化数字 X 到 D 有效数字。

3、CURRDATE(), CURRTIME()- 返回当前日期或时间。

4、NOW（） – 将当前日期和时间作为一个值返回。

5、MONTH（），DAY（），YEAR（），WEEK（），WEEKDAY（） – 从日期值中提取给定数据。

6、HOUR（），MINUTE（），SECOND（） – 从时间值中提取给定数据。

7、DATEDIFF（A，B） – 确定两个日期之间的差异，通常用于计算年龄

8、SUBTIMES（A，B） – 确定两次之间的差异。

9、FROMDAYS（INT） – 将整数天数转换为日期值。

### MYSQL 支持事务吗？

MySQL 支持事务吗？

在缺省模式下，MySQL 是 autocommit 模式的，所有的数据库更新操作都会即时

提交，所以在缺省情况下，MySQL 是不支持事务的。

但是如果你的 MySQL 表类型是使用 InnoDB Tables 或 BDB tables 的话，你的

MySQL 就可以使用事务处理,使用 SET

AUTOCOMMIT=0 就可以使 MySQL 允许在非 autocommit 模式，在非

autocommit 模式下，你必须使用 COMMIT 来提交你的更改，或者用 ROLLBACK来回滚你的更改。

### 事务的基本要素

### 事务隔离级别

### 如何解决事务的并发问题(脏读，幻读)？

#### 事务并发问题：

1. 第一类丢失更新(lost update)： 在完全未隔离事务的情况下，两个事物更新同一条数据资源，某一事物异常终止，回滚造成第一个完成的更新也同时丢失。

2. 脏读(dirty read)：如果第二个事务查询到第一个事务还未提交的更新数据，形成脏读。

3. 虚读(phantom read)：一个事务执行两次查询，第二次查询比第一次多出或少一些数据，造成两次结果不一致。只是另一个事务在这两次查询中间插入或者删除了数据造成的。

4. 不可重复读(unrepeated read)：一个事务两次读取同一行数据，结果得到不同状态结果，如中间正好另一个事务更新了该数据，两次结果相异，不可信任。

5. 第二类丢失更新(second lost updates)：是不可重复读的特殊情况，如果两个事务都读取同一行，然后两个都进行写操作，并提交，第一个事务所做的改变就会丢失。

#### 事务隔离级别：

为了解决数据库事务并发运行时的各种问题数据库系统提供四种事务隔离级别：

1. Serializable 串行化

2. Repeatable Read 可重复读

3. Read Commited 可读已提交

4. Read Uncommited 可读未提交

#### 并发控制：

1.数据库系统采用不同的锁类型来实现以上四种隔离级别，具体的实现过程对用户是透明的。用户应该关心的是如何选择合适的隔离级别。

2.对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为Read Committed，它能够避免脏读，而且具有较好的并发性能。

3.每个数据库连接都有一个全局变量@@tx\_isolation，表示当前的事务隔离级别。JDBC数据库连接使用数据库系统默认的隔离级别。

4.在Hibernate的配置文件中可以显示地设置隔离级别。每一种隔离级别对应着一个正整数。

5.需要注意的是，在受管理环境中，如果Hibernate使用的数据库连接来自于应用服务器提供的数据源，Hibernate不会改变这些连接的事务隔离级别。在这种情况下，应该通过修改应用服务器的数据源配置来修改隔离级别。

6.当数据库系统采用Red Committed隔离级别时，会导致不可重复读和第二类丢失更新的并发问题，在可能出现这种问题的场合。可以在应用程序中采用悲观锁或乐观锁来避免这类问题。

#### 悲观锁

　　正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

　　一个典型的依赖数据库的悲观锁调用：select \* from account where name=”Erica” for update这条 sql 语句锁定了 account 表中所有符合检索条件（ name=”Erica” ）的记录。本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。悲观锁，也是基于数据库的锁机制实现。

　　在Hibernate使用悲观锁十分容易，但实际应用中悲观锁是很少被使用的，因为它每次发送的SQL语句都会加上"for update"用于告诉数据库锁定相关数据，大大限制了并发性：

#### 乐观锁

　　相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（Version）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个"version"字段来实现。

　　乐观锁的工作原理：读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

Hibernate为乐观锁提供了3中实现：

1. 基于version

2. 基于timestamp

3. 为遗留项目添加添加乐观锁 Hibernate为乐观锁提供了3中实现：

MVCC多版本并发控制？

### binlog,redolog,undolog都是什么，起什么作用？

#### redo log日志模块

redo log是InnoDB存储引擎层的日志，又称重做日志文件，用于记录事务操作的变化，记录的是数据修改之后的值，不管事务是否提交都会记录下来。在实例和介质失败（media failure）时，redo log文件就能派上用场，如数据库掉电，InnoDB存储引擎会使用redo log恢复到掉电前的时刻，以此来保证数据的完整性。

在一条更新语句进行执行的时候，InnoDB引擎会把更新记录写到redo log日志中，然后更新内存，此时算是语句执行完了，然后在空闲的时候或者是按照设定的更新策略将redo log中的内容更新到磁盘中，这里涉及到WAL即Write Ahead logging技术，他的关键点是先写日志，再写磁盘。

有了redo log日志，那么在数据库进行异常重启的时候，可以根据redo log日志进行恢复，也就达到了crash-safe。

crash

崩溃

redo log日志的大小是固定的，即记录满了以后就从头循环写。



图片来自极客时间，该图展示了一组4个文件的redo log日志，checkpoint之前表示擦除完了的，即可以进行写的，擦除之前会更新到磁盘中，write pos是指写的位置，当write pos和checkpoint相遇的时候表明redo log已经满了，这个时候数据库停止进行数据库更新语句的执行，转而进行redo log日志同步到磁盘中。

#### binlog日志模块

binlog是属于MySQL Server层面的，又称为归档日志，属于逻辑日志，是以二进制的形式记录的是这个语句的原始逻辑，依靠binlog是没有crash-safe能力的

redo log和binlog区别

redo log是属于innoDB层面，binlog属于MySQL Server层面的，这样在数据库用别的存储引擎时可以达到一致性的要求。

redo log是物理日志，记录该数据页更新的内容；binlog是逻辑日志，记录的是这个更新语句的原始逻辑

redo log是循环写，日志空间大小固定；binlog是追加写，是指一份写到一定大小的时候会更换下一个文件，不会覆盖。

binlog可以作为恢复数据使用，主从复制搭建，redo log作为异常宕机或者介质故障后的数据恢复使用。

一条更新语句执行的顺序

update T set c=c+1 where ID=2;

执行器先找引擎取 ID=2 这一行。ID 是主键，引擎直接用树搜索找到这一行。如果 ID=2 这一行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。

执行器拿到引擎给的行数据，把这个值加上 1，比如原来是 N，现在就是 N+1，得到新的一行数据，再调用引擎接口写入这行新数据。

引擎将这行新数据更新到内存中，同时将这个更新操作记录到 redo log 里面，此时 redo log 处于 prepare 状态。然后告知执行器执行完成了，随时可以提交事务。

执行器生成这个操作的 binlog，并把 binlog 写入磁盘。

执行器调用引擎的提交事务接口，引擎把刚刚写入的 redo log 改成提交（commit）状态，更新完成。

这个update语句的执行流程图，图中浅色框表示是在 InnoDB 内部执行的，深色框表示是在执行器中执行的。



innodb事务日志包括redo log和undo log。redo log是重做日志，提供前滚操作，undo log是回滚日志，提供回滚操作。

undo log不是redo log的逆向过程，其实它们都算是用来恢复的日志：

1.redo log通常是物理日志，记录的是数据页的物理修改，而不是某一行或某几行修改成怎样怎样，它用来恢复提交后的物理数据页(恢复数据页，且只能恢复到最后一次提交的位置)。

2.undo用来回滚行记录到某个版本。undo log一般是逻辑日志，根据每行记录进行记录。

#### 一、重做日志（redo log）

作用：

确保事务的持久性。防止在发生故障的时间点，尚有脏页未写入磁盘，在重启mysql服务的时候，根据redo log进行重做，从而达到事务的持久性这一特性。

#### 二、回滚日志（undo log）

作用：

保存了事务发生之前的数据的一个版本，可以用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读

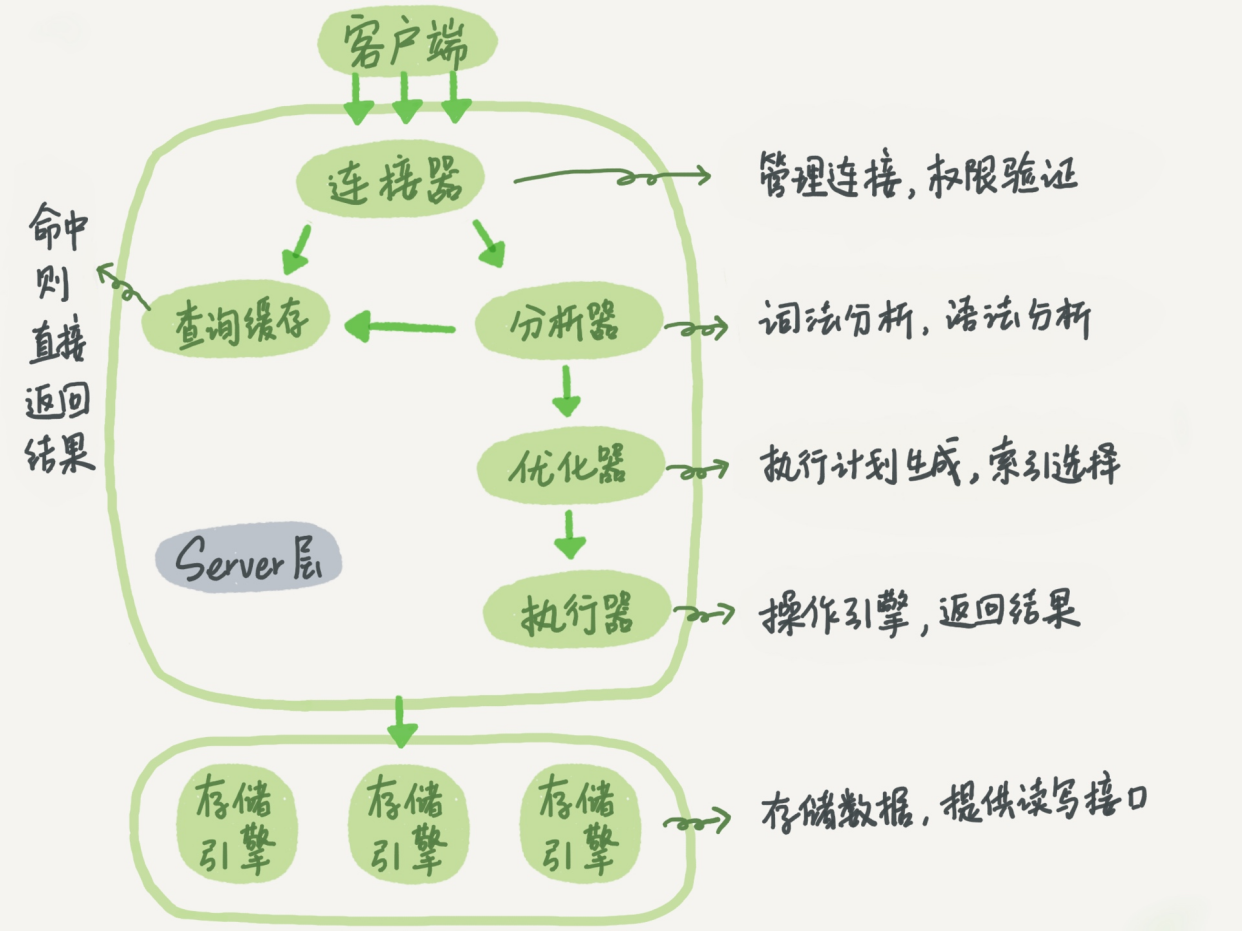
#### 三、二进制日志（binlog）：

作用：

用于复制，在主从复制中，从库利用主库上的binlog进行重播，实现主从同步。

用于数据库的基于时间点的还原。

### Mysql结构



InnoDB的行锁/表锁？

myisam和innodb的区别，什么时候选择myisam？

### 为什么选择B+树作为索引结构？

Hash索引：Hash索引底层是哈希表，哈希表是一种以key-value存储数据的结构，所以多个数据在存储关系上是完全没有任何顺序关系的，所以，对于区间查询是无法直接通过索引查询的，就需要全表扫描。所以，哈希索引只适用于等值查询的场景。而B+ 树是一种多路平衡查询树，所以他的节点是天然有序的（左子节点小于父节点、父节点小于右子节点），所以对于范围查询的时候不需要做全表扫描

二叉查找树：解决了排序的基本问题，但是由于无法保证平衡，可能退化为链表。

平衡二叉树：通过旋转解决了平衡的问题，但是旋转操作效率太低。

红黑树：通过舍弃严格的平衡和引入红黑节点，解决了 AVL旋转效率过低的问题，但是在磁盘等场景下，树仍然太高，IO次数太多。

B+树：在B树的基础上，将非叶节点改造为不存储数据纯索引节点，进一步降低了树的高度；此外将叶节点使用指针连接成链表，范围查询更加高效。

### 索引B+树的叶子节点都可以存哪些东西？

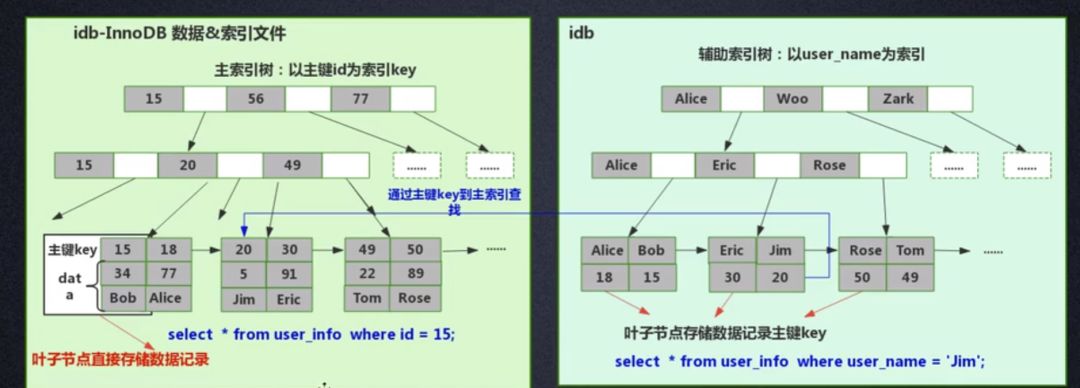
首先MYSQL默认InnoDB引擎，该引擎默认B+树；先说结论：B+树叶子结点存储的是主键KEY或者具体数据。分情况讨论：

#### 主键KEY

比如说user\_name是个索引，当执行该SQL：select \* from user\_info where `user\_name` = 'xiaoming'; InnoDB 就会建立 user\_name 索引 B+树，节点里存的是 user\_name 这个 KEY，叶子节点存储的数据的是主键 KEY。注意，叶子存储的是主键 KEY！拿到主键 KEY 后，InnoDB 才会去主键索引树里根据刚在 user\_name 索引树找到的主键 KEY 查找到对应的数据。

#### 具体数据

在执行 select \* from user\_info where id = 5 这个语句时，InnoDB 就会查询这颗主键 ID 索引 B+树，找到对应的 user\_name='Bob'。



原因

问题来了，为什么 InnoDB 只在主键索引树的叶子节点存储了具体数据，但是其他索引树却不存具体数据呢，而要多此一举先找到主键，再在主键索引树找到对应的数据呢?

其实很简单，因为 InnoDB 需要节省存储空间。一个表里可能有很多个索引，InnoDB 都会给每个加了索引的字段生成索引树，如果每个字段的索引树都存储了具体数据，那么这个表的索引数据文件就变得非常巨大（数据极度冗余了）。从节约磁盘空间的角度来说，真的没有必要每个字段索引树都存具体数据，通过这种看似“多此一举”的步骤，在牺牲较少查询的性能下节省了巨大的磁盘空间，这是非常有值得的。

可能存储的是整行数据，也有可能是主键的值。B+树的叶子节点存储了整行数据的是主键索引，也被称之为聚簇索引。而索引B+ Tree的叶子节点存储了主键的值的是非主键索引，也被称之为非聚簇索引。

注意：

MySQL InnoDB一定会建立聚簇索引，把实际数据行和相关的键值保存在一块，这也决定了一个表只能有一个聚簇索引，即MySQL不会一次把数据行保存在二个地方。

InnoDB通常根据主键值(primary key)进行聚簇

如果没有创建主键，则会用一个唯一且不为空的索引列做为主键，成为此表的聚簇索引

上面二个条件都不满足，InnoDB会自己创建一个虚拟的聚集索引

正因为InnoDB将数据保存在一处，因此其插入速度严重依赖插入顺序。按照主键顺序插入无疑是最快的。如果不是按照主键插入，建议加载完成后最好使用OPTIMIZE TABLE重新组织一下表。

OPTIMIZE：优化

### 什么是聚簇索引

总结下，聚簇索引并不是一种单独的索引类型，而是一种数据存储方式。当表中有聚簇索引时，它的数据行实际上存放在索引的叶子节点中。

### 什么是覆盖索引

一个查询语句的执行只用从索引中就能够取得，不必回表读取。也可以称之为实现了索引覆盖。

### 为什么不推荐使用select \*

增加了不必要的网络开销。在查询巨量时，数据传输是十分耗时的。

我们在上面已经提到了覆盖索引。显然使用select \*导致无法使用覆盖索引。

### 什么是最左匹配原则

最左优先，以最左边的为起点任何连续的索引都能匹配上。同时遇到范围查询(>、<、between、like)就会停止匹配。

注意：

但是这不意味着不按照索引顺序写的查询条件就会100%导致不走索引。MySQL的性能优化器会判断纠正这条sql语句该以什么样的顺序执行效率最高，最后才生成真正的执行计划。

### 查询在什么时候不走（预期中的）索引

1、模糊查询 %like

2、索引列参与计算,使用了函数

3、非最左前缀顺序

4、where对null判断

5、where不等于

6、or操作有至少一个字段没有索引

7、需要回表的查询结果集过大（超过配置的范围）

sql如何优化?

### explain是如何解析sql的？

explain分析查询

使用 EXPLAIN 关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是如何处理你的SQL语句的。这可以帮你分析你的查询语句或是表结构的性能瓶颈。通过explain命令可以得到:

表的读取顺序

数据读取操作的操作类型

哪些索引可以使用

哪些索引被实际使用

表之间的引用

每张表有多少行被优化器查询

order by原理

## 多线程 专题

多线程有什么

线程和进程的区别是什么？

Java 实现线程有哪几种方式？

启动线程方法 start()和 run()有什么区别？

怎么终止一个线程？如何优雅地终止线程

线程中的 wait()和 sleep()方法有什么区别？

为什么要使用线程池？

常用的几种线程池并讲讲其中的工作原理。

线程池启动线程 submit()和 execute()方法有什么不同？

### 什么是活锁、饥饿、无锁、死锁？

#### 死锁：

A线程持有 锁1，接下来要获取锁2；与此同时，B线程持有锁2，要获取锁1。两个线程都在等对方释放自己需要的锁，这时两方会永远等待下去，就形成了死锁。

#### 死锁的四个必要条件：

1.互斥：资源(锁)同时只能被一个线程占用。

2.占有且等待：线程已经占用资源A，同时等待资源B时，不释放资源A。

3.不可抢占：其他线程不能强行获取当前线程占有的资源

4.循环等待：存在一个等待链，即T1等待T2占有的资源，T2等待T3占有的资源，T3等待T1占有的资源。

如果要解决死锁，则需要破坏任意一死锁的必要条件。

一.破坏占有且等待条件

解决方法：只要限定所有资源锁同时获取，同时释放。就可以预防掉死锁。其实就是破坏掉占有且等待条件。

二.破坏循环等待条件

解决方法:对锁进行排序，每次申请锁需要按从小到大顺序申请。这样就不存在循环等待了

三.破坏不可抢占条件

解决方法: 使用 Lock 和UnLock，在finally里执行unlock，主动释放资源。此时别人就可以抢占了。

#### 活锁：

多个线程获取不到资源，就放开已获得的资源，重试。相当于系统空转，一直在做无用功。

例如，行人走路相向而行，互相谦让，一直重复谦让的过程。

解决方法：引入一些随机性，比如暂停随机时间重试。

#### 饥饿：

1：优先级高的线程总是抢占到资源，而优先级低的线程可能会一直等待，从而无法获取资源无法执行；

2：一个线程一直不释放资源，别的线程也会出现饥饿的情况。

3：wait()等待情况下的线程一直都不被notify，而其他的线程总是能被唤醒

解决方法:引入公平锁

#### 无锁：

CAS(campare and swap):内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做并返回false。CAS是原子操作，只有一条cpu指令

无锁即不对资源锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功。一个修改操作在一个循环内进行，线程会不断的尝试修改共享资源，如果没有冲突（CAS判断）就修改成功并退出否则就会继续下一次循环尝试。

如jdk的基于CAS实现的原子操作类，就是对无锁的实现。 还有无锁队列，也是循环线程对变量进行CAS操作的数据结构。

CAS的缺点:

1.ABA问题：V值为A，T1，T2从内存取出V值为A.。然后T2 CAS修改变量V为B , 接着T2 又CAS修改变量V为A。这时T1 CAS 变量V时发现内存中V还是A ，CAS操作成功。

2.循环消耗大

3.只能保证一个共享变量的原子操作

### 微服务 专题

什么是微服务？

维基上对其定义为：一种软件开发技术- 面向服务的体系结构（SOA）架构样式的一种变体，将应用程序构造为一组松散耦合的服务。在微服务体系结构中，服务是细粒度的，协议是轻量级的。

微服务（或微服务架构）是一种云原生架构方法，其中单个应用程序由许多松散耦合且可独立部署的较小组件或服务组成。这些服务通常

● 有自己的堆栈，包括数据库和数据模型；

● 通过REST API，事件流和消息代理的组合相互通信；

● 和它们是按业务能力组织的，分隔服务的线通常称为有界上下文。

尽管有关微服务的许多讨论都围绕体系结构定义和特征展开，但它们的价值可以通过相当简单的业务和组织收益更普遍地理解：

● 可以更轻松地更新代码。

● 团队可以为不同的组件使用不同的堆栈。

● 组件可以彼此独立地进行缩放，从而减少了因必须缩放整个应用程序而产生的浪费和成本，因为单个功能可能面临过多的负载。

微服务是一种架构模型，可以更好地促进所需的运营模型。

可独立部署

微服务的最重要的单一特征可能是，由于服务较小且可独立部署，因此不再需要国会采取行动才能更改应用程序中的一行文字。

微服务向组织承诺了解决方案，以解决因细微变化而引起的内在挫折，这需要花费大量时间。它不需要博士学位。在计算机科学中可以看到或理解更好地促进速度和敏捷性的方法的价值。

但是，速度并不是以这种方式设计服务的唯一价值。一种常见的新兴组织模型是将跨职能的团队聚集在业务问题，服务或产品上。微服务模型非常适合这种趋势，因为它使组织能够围绕一项服务或一组服务创建跨职能的小型团队，并使它们以敏捷的方式运作。

最后，服务的小规模加上清晰的边界和沟通模式，使新团队成员更容易理解代码库并快速做出贡献，这在速度和员工士气方面均具有明显的好处。

正确的工作工具

在传统的n层体系结构模式中，应用程序通常共享一个公共堆栈，而大型关系数据库支持整个应用程序。这种方法有几个明显的缺点-最主要的缺点是，即使对于某些元素有一个清晰，更好的工具，应用程序的每个组件也必须共享一个公共的堆栈，数据模型和数据库。它造成了糟糕的体系结构，并且使开发人员感到沮丧，他们不断意识到可以使用更好，更有效的方式来构建这些组件。

相比之下，在微服务模型中，组件是独立部署的，并通过REST，事件流和消息代理的某种组合进行通信-因此，可以针对该服务优化每个单独服务的堆栈。技术一直在变化，由多个较小的服务组成的应用程序随着更理想的技术的发展而变得更容易且成本更低。

精确缩放

使用微服务，可以单独部署单个服务，但是也可以单独扩展它们。由此带来的好处是显而易见的：如果正确完成，微服务比单片应用程序所需的基础结构要少，因为微服务仅支持对需要它的组件进行精确缩放，而对于单片应用程序则不需要整个应用程序。

#### 关键支持技术和工具编辑

尽管几乎任何现代工具或语言都可以在微服务体系结构中使用，但仍有一些核心工具已成为微服务必不可少的基本定义：

容器，Docker和Kubernetes

微服务的关键要素之一是它通常很小。（没有任意数量的代码来确定某项内容是否为微服务，但名称中恰好有“微”字。）

当码头工人在现代化的集装箱时代迎来了2013年，它也推出了计算模型，将成为微服务最密切相关。由于单个容器没有自己操作系统的开销，因此它们比传统虚拟机更小，更轻，并且可以更快地上下旋转，从而使其与微服务架构中的更小，更轻便的服务完美匹配。 。

随着服务和容器的激增，对大型容器进行编排和管理很快成为关键挑战之一。Kubernetes已经成为世界上最受欢迎的容器编排 技术之一，因为它做得很好。

API网关

微服务通常通过API进行通信，尤其是在首次建立状态时。虽然确实可以实现客户端和服务之间的直接通信，但API网关通常是有用的中介层，尤其是随着应用程序中服务数量的不断增长。API网关通过路由请求，将请求散布到多个服务中并提供额外的安全性和身份验证，充当客户端的反向代理。

有迹象表明，可用于执行API网关，包括API管理平台的多种技术，但如果正在使用集装箱和Kubernetes实现的微服务架构中，网关使用的入口或更近，通常实现Istio。

讯息传递

尽管最佳实践可能是设计无状态服务，但是状态仍然存在，服务需要意识到这一点。尽管API调用通常是初始为给定服务建立状态的有效方法，但它并不是保持最新状态的特别有效方法。不断的民意测验“我们到了吗？” 使服务保持最新状态的方法根本不切实际。

相反，必须将建立状态的API调用与消息传递或事件流耦合在一起，以便服务可以广播状态更改，而其他相关方可以侦听这些更改并相应地进行调整。这项工作可能最适合于通用消息代理，但是在某些情况下，事件流传输平台（例如Apache Kafka）可能是一个很好的选择。

无服务器

无服务器架构将某些核心云和微服务模式纳入其逻辑结论。在无服务器的情况下，执行单元不仅是一个小型服务，而且是一个功能，通常只能是几行代码。无服务器功能与微服务之间的界线模糊，但通常认为功能甚至比微服务小。

无服务器架构和功能即服务（FaaS）平台与微服务共享相似性的地方在于，它们都对创建更小的部署单位以及根据需求进行精确扩展感兴趣。

### Kubernetes

### 前后端分离是如何做的

前后端分离-01：怎么实现前后端分离？

blues\_phone 2019-03-18 09:32:54 34847 收藏 29

分类专栏： 前后端分离 文章标签： 前后端分离

版权

第一步部分：后端（因为笔者是做后端的所以放在第一个），前后端的后端分两个部门。

业务处理：里面包含model，算法，业务逻辑，网络通信，多线程，多进程。

web服务器：使用任何能返回json和二进制的数据类型的框架就行。此时不再去控制前端页面的跳转了。

第二部分：api文档服务器。api文档是作为连接后端和前端的桥梁。他定义了各种api文档的请求方式和返回数据的类型。文档室友后端的人员编写。前端根据文档模拟请求数据。最难的地方在于，随着后台的开发，api的接口会进行改变，增加修改删除或者返回的数据发生变化，这些都是不可避免的。那么api的版本问题就给前端带来了问题，明明昨天请求还是OK的，今天就挂了，没有正确的数据，也会影响前端的开发进度。

有人会说放在word或者excel里面文档，那后端每开发或改动一个新功能，都要给前端一份新的api的文档，我们做的就是软件，追求的是效率，岂能容忍这种低效率的事情。

建议自行搭建一个api文档服务器：每一个API文档能够非常容易的定义请求和响应数据结构，最好能够自动生成请求和响应示例，更重要的是，生成的示例必须看起来是符合业务需求的真实数据；支持api版本发布；自动创建版本历史记录，同时要能够非常方便的查看历史版本；能够通过WEB浏览器访问；树形目录结构，方便阅读和搜索；

第三部分：前端（笔者前端功底还是比较薄弱的）：这里的前端是可以独立的，没有后端也能访问，只是获取不到后端数据。既然可以单独访问也就是说他有自己的web服务器，一般webpack和node.js构建起服务器。那么他如何和后端数据进行通信的？当然是通过api文档文档服务器里面的api描述进行数据请求。前端同时控制页面的跳转。

此时会有一个问题，前端和后端都是独立的，那么如何保证他们之间的通信的安全性？

解决方案：基于oauth2的通信，对返回数据进行加密（用AES加密数据，用RSA加密AES的秘钥）

对目前的web来说，前后端分离已经变得越来越流行了，越来越多的企业/网站都开始往这个方向靠拢。那么，为什么要选择前后端分离呢？前后端分离对实际开发有什么好处呢?

为什么选择前后端分离

在以前传统的网站开发中，前端一般扮演的只是切图的工作，只是简单地将UI设计师提供的原型图实现成静态的HTML页面，而具体的页面交互逻辑，比如与后台的数据交互工作等，可能都是由后台的开发人员来实现的，或者是前端是紧紧的耦合后台。比如，以前淘宝的Web基本上都是基于MVC框架webx，架构决定了前端只能依赖后端。所以他们的开发模式依然是，前端写好静态demo，后端翻译成VM模版，这种模式的问题就不说了，被吐槽了很久。

而且更有可能后台人员直接兼顾前端的工作，一边实现API接口，一边开发页面，两者互相切换着做，而且根据不同的url动态拼接页面，这也导致后台的开发压力大大增加。前后端工作分配不均。不仅仅开发效率慢，而且代码难以维护。而前后端分离的话，则可以很好的解决前后端分工不均的问题，将更多的交互逻辑分配给前端来处理，而后端则可以专注于其本职工作，比如提供API接口，进行权限控制以及进行运算工作。而前端开发人员则可以利用nodejs来搭建自己的本地服务器，直接在本地开发，然后通过一些插件来将api请求转发到后台，这样就可以完全模拟线上的场景，并且与后台解耦。前端可以独立完成与用户交互的整一个过程，两者都可以同时开工，不互相依赖，开发效率更快，而且分工比较均衡。

如何做到前后端分离

(以下的内容都是基于我们的电影购票网站来讨论的)

前端的技术框架是: vue全家桶+nodejs+express(实现的是单页面(SPA)应用)

首先，先分清楚前后端的工作

前端的工作：实现整一个前端页面以及交互逻辑，以及利用ajax与nodejs服务器（中间层)交互

后端的工作：提供API接口，利用redis来管理session,与数据库交互

我们项目的整一个架构如下:

640?wx\_fmt=png

接下来进入正题，如何实现前后端分离

一般来说，要实现前后端分离，前端就需要开启一个本地的服务器来运行自己的前端代码，以此来模拟真实的线上环境，并且，也是为了更好的开发。因为你在实际开发中，你不可能要求每一个前端都去搭建一个java(php)环境，并且在java环境下开发，这对于前端来说，学习成本太高了。但如果本地没有开启服务器的话，不仅无法模拟线上的环境，而且还面临到了跨域的问题，因为你如果写静态的html页面，直接在文件目录下打开的话，你是无法发出ajax请求的(浏览器跨域的限制),因此，你需要在本地运行一个服务器，可是又不想搭建陌生而庞大的java环境，怎么办法呢？nodejs正好解决了这个问题。在我们项目中，我们利用nodejs的express框架来开启一个本地的服务器，然后利用nodejs的一个http-proxy-middleware插件将客户端发往nodejs的请求转发给真正的服务器，让nodejs作为一个中间层。这样，前端就可以无忧无虑的开发了

由于前后端分离后，前端和后台同时开发时，就可能遇到前端已经开发好一个页面了，可是却等待后台API接口的情况。比如说A是负责前端，B是负责后台，A可能用了一周做好了基本的结构，并且需要API接口联调后，才能继续开发，而此时B却还没有实现好所需要的接口，这种情况，怎么办呢？在我们这个项目里，我们是通过mock提供一些模拟数据，我们先规定好了API接口，设计出了一套API文档，然后我们就可以通过API文档，利用mock(http://mockjs.com)来返回一些假数据，这样就可以模拟发送API到接受响应的整一个过程，因此前端也不需要依赖于后端开发了，可以独立开发，等到后台的API全部设计完之后，就可以比较快速的联调。

### oauth2

微服务哪些框架

### RPC 的实现原理

RPC（Remote Procedure Call）远程过程调用，简单的理解是一个节点请求另一个节点提供的服务

SAP系统RPC调用的原理其实很简单，有一些类似于三层构架的C/S系统，第三方的客户程序通过接口调用SAP内部的标准或自定义函数，获得函数返回的数据进行处理后显示或打印。

本地过程调用：如果需要将本地student对象的age+1，可以实现一个addAge()方法，将student对象传入，对年龄进行更新之后返回即可，本地方法调用的函数体通过函数指针来指定。

远程过程调用：上述操作的过程中，如果addAge()这个方法在服务端，执行函数的函数体在远程机器上，如何告诉机器需要调用这个方法呢？

首先客户端需要告诉服务器，需要调用的函数，这里函数和进程ID存在一个映射，客户端远程调用时，需要查一下函数，找到对应的ID，然后执行函数的代码。

客户端需要把本地参数传给远程函数，本地调用的过程中，直接压栈即可，但是在远程调用过程中不再同一个内存里，无法直接传递函数的参数，因此需要客户端把参数转换成字节流，传给服务端，然后服务端将字节流转换成自身能读取的格式，是一个序列化和反序列化的过程。

3.数据准备好了之后，如何进行传输？网络传输层需要把调用的ID和序列化后的参数传给服务端，然后把计算好的结果序列化传给客户端，因此TCP层即可完成上述过程，gRPC中采用的是HTTP2协议。

总结一下上述过程：

// Client端

// Student student = Call(ServerAddr, addAge, student)

1. 将这个调用映射为Call ID。

2. 将Call ID，student（params）序列化，以二进制形式打包

3. 把2中得到的数据包发送给ServerAddr，这需要使用网络传输层

4. 等待服务器返回结果

5. 如果服务器调用成功，那么就将结果反序列化，并赋给student，年龄更新

// Server端

1. 在本地维护一个Call ID到函数指针的映射call\_id\_map，可以用Map<String, Method> callIdMap

2. 等待客户端请求

3. 得到一个请求后，将其数据包反序列化，得到Call ID

4. 通过在callIdMap中查找，得到相应的函数指针

5. 将student（params）反序列化后，在本地调用addAge()函数，得到结果

6. 将student结果序列化后通过网络返回给Client

### Dubbo 的实现原理

Dubbo(读音[ˈdʌbəʊ])是阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和 [1] Spring框架无缝集成。

Dubbo是一款高性能、轻量级的开源Java RPC框架

三大核心能力：

面向接口的远程方法调用

智能容错和负载均衡

服务自动注册和发现。

## 算法题

输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个（剑指offer原题，就是排序改一下比较规则）

对于LR来说，LR如果多了一维冗余特征，其权重和AUC会怎样变化（权重变为1/2, AUC不会变化）

写出LR的损失函数（交叉熵损失函数）

## 算法题

二叉树转双向链表（中序非递归遍历修改指针）

其他部分

高并发系统的限流如何实现？

高并发秒杀系统的设计

负载均衡如何设计？

## 操作系统

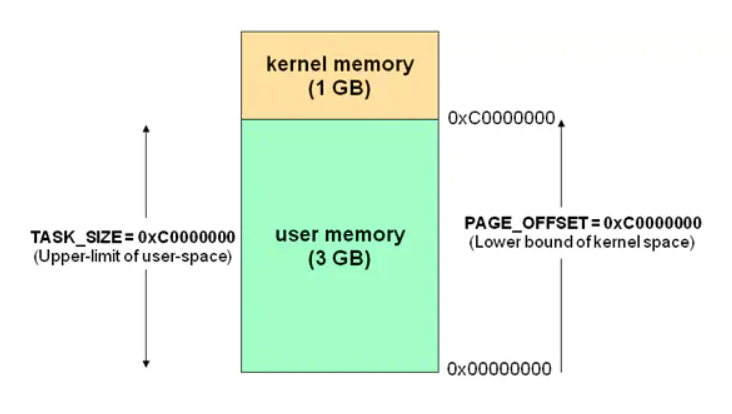
### 用户态和核心态

用户空间与内核空间

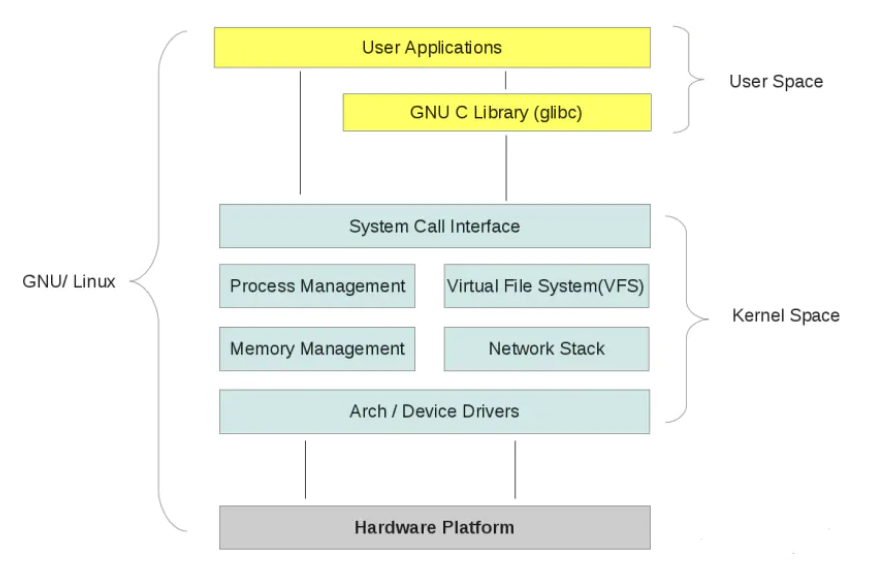
我们知道现在操作系统都是采用虚拟存储器，那么对32位操作系统而言，它的寻址空间（虚拟存储空间）为4G（2的32次方）。操心系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核，保证内核的安全，操心系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。

针对linux操作系统而言，将最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF），供内核使用，称为内核空间，而将较低的3G字节（从虚拟地址0x00000000到0xBFFFFFFF），供各个进程使用，称为用户空间。

每个进程可以通过系统调用进入内核，因此，Linux内核由系统内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有4G字节的虚拟空间。空间分配如下图所示：



有了用户空间和内核空间，整个linux内部结构可以分为三部分，从最底层到最上层依次是：硬件-->内核空间-->用户空间。如下图所示：



需要注意的细节问题：

内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中。

Linux使用两级保护机制：0级供内核使用，3级供用户程序使用。

内核态与用户态

（1）当一个任务（进程）执行系统调用而陷入内核代码中执行时，称进程处于内核运行态（内核态）。此时处理器处于特权级最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。

（2）当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。此时处理器在特权级最低的（3级）用户代码中运行。当正在执行用户程序而突然被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。

进程上下文与中断上下文

程序在执行过程中通常有用户态和内核态两种状态，CPU对处于内核态根据上下文环境进一步细分，因此有了下面三种状态：

（1）内核态，运行于进程上下文，内核代表进程运行于内核空间。

（2）内核态，运行于中断上下文，内核代表硬件运行于内核空间。

（3）用户态，运行于用户空间。

进程上下文

用户空间的应用程序，通过系统调用，进入内核空间。这个时候用户空间的进程要传递 很多变量、参数的值给内核，内核态运行的时候也要保存用户进程的一些寄存 器值、变量等。所谓的“进程上下文”，可以看作是用户进程传递给内核的这些参数以及内核要保存的那一整套的变量和寄存器值和当时的环境等。

相对于进程而言，就是进程执行时的环境。具体来说就是各个变量和数据，包括所有的寄存器变量、进程打开的文件、内存信息等。一个进程的上下文可以分为三个部分:用户级上下文、寄存器上下文以及系统级上下文。

（1）用户级上下文: 正文、数据、用户堆栈以及共享存储区；

（2）寄存器上下文: 通用寄存器、程序寄存器(IP)、处理器状态寄存器(EFLAGS)、栈指针(ESP)；

（3）系统级上下文: 进程控制块task\_struct、内存管理信息(mm\_struct、vm\_area\_struct、pgd、pte)、内核栈。

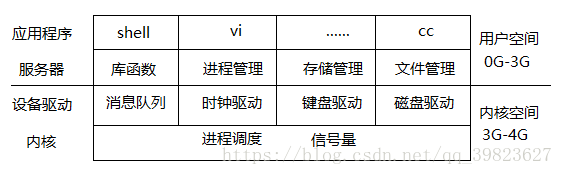
当发生进程调度时，进行进程切换就是上下文切换(context switch).操作系统必须对上面提到的全部信息进行切换，新调度的进程才能运行。而系统调用进行的模式切换(mode switch)。模式切换与进程切换比较起来，容易很多，而且节省时间，因为模式切换最主要的任务只是切换进程寄存器上下文的切换。

中断上下文

硬件通过触发信号，导致内核调用中断处理程序，进入内核空间。这个过程中，硬件的 一些变量和参数也要传递给内核，内核通过这些参数进行中断处理。所谓的“ 中断上下文”，其实也可以看作就是硬件传递过来的这些参数和内核需要保存的一些其他环境（主要是当前被打断执行的进程环境）。中断时，内核不代表任何进程运行，它一般只访问系统空间，而不会访问进程空间，内核在中断上下文中执行时一般不会阻塞。

### 用户态和内核态的理解和区别

1、linux进程有4GB地址空间，如图所示：



3G-4G大部分是共享的，是内核态的地址空间。这里存放整个内核的代码和所有的内核模块以及内核所维护的数据。

2、特权级的概念：

对于任何操作系统来说，创建一个进程是核心功能。创建进程要做很多工作，会消耗很多物理资源。比如分配物理内存，父子进程拷贝信息，拷贝设置页目录页表等等，这些工作得由特定的进程去做，所以就有了特权级别的概念。最关键的工作必须交给特权级最高的进程去执行，这样可以做到集中管理，减少有限资源的访问和使用冲突。inter x86架构的cpu一共有四个级别，0-3级，0级特权级最高，3级特权级最低。

3、用户态和内核态的概念：

当一个进程在执行用户自己的代码时处于用户运行态（用户态），此时特权级最低，为3级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态。Ring3状态不能访问Ring0的地址空间，包括代码和数据；当一个进程因为系统调用陷入内核代码中执行时处于内核运行态（内核态），此时特权级最高，为0级。执行的内核代码会使用当前进程的内核栈，每个进程都有自己的内核栈。

用户运行一个程序，该程序创建的进程开始时运行自己的代码，处于用户态。如果要执行文件操作、网络数据发送等操作必须通过write、send等系统调用，这些系统调用会调用内核的代码。进程会切换到Ring0，然后进入3G-4G中的内核地址空间去执行内核代码来完成相应的操作。内核态的进程执行完后又会切换到Ring3，回到用户态。这样，用户态的程序就不能随意操作内核地址空间，具有一定的安全保护作用。这说的保护模式是指通过内存页表操作等机制，保证进程间的地址空间不会互相冲突，一个进程的操作不会修改另一个进程地址空间中的数据。

4、用户态和内核态的切换

当在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成一些用户态自己没有特权和能力完成的操作时就会切换到内核态。

用户态切换到内核态的3种方式

（1）系统调用

这是用户态进程主动要求切换到内核态的一种方式。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。例如fork（）就是执行了一个创建新进程的系统调用。系统调用的机制和新是使用了操作系统为用户特别开放的一个中断来实现，如Linux的int 80h中断。

（2）异常

当cpu在执行运行在用户态下的程序时，发生了一些没有预知的异常，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进程中，也就是切换到了内核态，如缺页异常。

（3）外围设备的中断

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而转到与中断信号对应的处理程序去执行，如果前面执行的指令时用户态下的程序，那么转换的过程自然就会是 由用户态到内核态的切换。如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后边的操作等。

这三种方式是系统在运行时由用户态切换到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。从触发方式上看，切换方式都不一样，但从最终实际完成由用户态到内核态的切换操作来看，步骤有事一样的，都相当于执行了一个中断响应的过程。系统调用实际上最终是中断机制实现的，而异常和中断的处理机制基本一致。

5、用户态到内核态具体的切换步骤：

（1）从当前进程的描述符中提取其内核栈的ss0及esp0信息。

（2）使用ss0和esp0指向的内核栈将当前进程的cs,eip,eflags,ss,esp信息保存起来，这个过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令。

（3）将先前由中断向量检索得到的中断处理程序的cs,eip信息装入相应的寄存器，开始执行中断处理程序，这时就转到了内核态的程序执行了。

## Python

### Python 如何进行内存管理？/Python 中的垃圾回收机制？

Python引入了一个机制：引用计数

python内部使用引用计数，来保持追踪内存中的对象，Python内部记录了对象有多少个引用，即引用计数，当对象被创建时就创建了一个引用计数，当对象不再需要时，这个对象的引用计数为0时，它被垃圾回收。

python是如何进行内存管理的

引用计数减少情况

1、一个本地引用离开了它的作用域。比如上面的foo(x)函数结束时，x指向的对象引用减1。

2、对象的别名被显式的销毁：del x ;或者del y

3、对象的一个别名被赋值给其他对象：x=789

4、对象从一个窗口对象中移除：myList.remove(x)

5、窗口对象本身被销毁：del myList，或者窗口对象本身离开了作用域。

python是如何进行内存管理的

垃圾回收

1、当内存中有不再使用的部分时，垃圾收集器就会把他们清理掉。它会去检查那些引用计数为0的对象，然后清除其在内存的空间。当然除了引用计数为0的会被清除，还有一种情况也会被垃圾收集器清掉：当两个对象相互引用时，他们本身其他的引用已经为0了。

2、垃圾回收机制还有一个循环垃圾回收器, 确保释放循环引用对象(a引用b，b引用a，导致其引用计数永远不为0)。

在Python中，许多时候申请的内存都是小块的内存，这些小块内存在申请后，很快又会被释放，由于这些内存的申请并不是为了创建对象，所以并没有对象一级的内存池机制。这就意味着Python在运行期间会大量地执行malloc和free的操作，频繁地在用户态和核心态之间进行切换，这将严重影响Python的执行效率。

python是如何进行内存管理的

4

内存池机制

Python提供了对内存的垃圾收集机制，但是它将不用的内存放到内存池而不是返回给操作系统。

Python中所有小于256个字节的对象都使用pymalloc实现的分配器，而大的对象则使用系统的 malloc。另外Python对象，如整数，浮点数和List，都有其独立的私有内存池，对象间不共享他们的内存池。也就是说如果你分配又释放了大量的整数，用于缓存这些整数的内存就不能再分配给浮点数，

### 什么是 Python 生成器？

generator，有两种产生生成器对象的方式：一种是列表生成式加括号：

g1 = (x for x in range(10))

一种是在函数定义中包含yield关键字：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

yield b

a, b = b, a + b

n = n + 1

return 'done'

g2 = fib(8)

对于generator对象g1和g2，可以通过next(g1)不断获得下一个元素的值，如果没有更多的元素，就会报错StopIteration

也可以通过for循环获得元素的值。

生成器的好处是不用占用很多内存，只需要在用的时候计算元素的值就行了。

### 什么是 Python 迭代器？

Python中可以用于for循环的，叫做可迭代Iterable，包括list/set/tuple/str/dict等数据结构以及生成器；可以用以下语句判断一个对象是否是可迭代的：

from collections import Iterable

isinstance(x, Iterable)

迭代器Iterator，是指可以被next()函数调用并不断返回下一个值，直到StopIteration；生成器都是Iterator，而列表等数据结构不是；可以通过以下语句将list变为Iterator：

iter([1,2,3,4,5])

生成器都是Iterator迭代器，但迭代器不一定是生成器。

### 迭代器

迭代是Python最强大的功能之一，是访问集合元素的一种方式。

迭代器是一个可以记住遍历的位置的对象。

迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：iter() 和 next()。

字符串，列表或元组对象都可用于创建迭代器：

实例(Python 3.0+)

>>> list=[1,2,3,4]

>>> it = iter(list) # 创建迭代器对象

>>> print (next(it)) # 输出迭代器的下一个元素

1

>>> print (next(it))

2

>>>

迭代器对象可以使用常规for语句进行遍历：

实例(Python 3.0+)

#!/usr/bin/python3

list=[1,2,3,4]

it = iter(list) # 创建迭代器对象

for x in it:

print (x, end=" ")

执行以上程序，输出结果如下：

1 2 3 4

也可以使用 next() 函数：

实例(Python 3.0+)

#!/usr/bin/python3

import sys # 引入 sys 模块

list=[1,2,3,4]

it = iter(list) # 创建迭代器对象

while True:

try:

print (next(it))

except StopIteration:

sys.exit()

执行以上程序，输出结果如下：

1

2

3

4

创建一个迭代器

把一个类作为一个迭代器使用需要在类中实现两个方法 \_\_iter\_\_() 与 \_\_next\_\_() 。

如果你已经了解的面向对象编程，就知道类都有一个构造函数，Python 的构造函数为 \_\_init\_\_(), 它会在对象初始化的时候执行。

更多内容查阅：Python3 面向对象

\_\_iter\_\_() 方法返回一个特殊的迭代器对象， 这个迭代器对象实现了 \_\_next\_\_() 方法并通过 StopIteration 异常标识迭代的完成。

\_\_next\_\_() 方法（Python 2 里是 next()）会返回下一个迭代器对象。

创建一个返回数字的迭代器，初始值为 1，逐步递增 1：

实例(Python 3.0+)

class MyNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

x = self.a

self.a += 1

return x

myclass = MyNumbers()

myiter = iter(myclass)

print(next(myiter))

print(next(myiter))

print(next(myiter))

print(next(myiter))

print(next(myiter))

执行输出结果为：

1

2

3

4

5

StopIteration

StopIteration 异常用于标识迭代的完成，防止出现无限循环的情况，在 \_\_next\_\_() 方法中我们可以设置在完成指定循环次数后触发 StopIteration 异常来结束迭代。

在 20 次迭代后停止执行：

实例(Python 3.0+)

class MyNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

if self.a <= 20:

x = self.a

self.a += 1

return x

else:

raise StopIteration

myclass = MyNumbers()

myiter = iter(myclass)

for x in myiter:

print(x)

执行输出结果为：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

### 生成器

在 Python 中，使用了 yield 的函数被称为生成器（generator）。

跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。

在调用生成器运行的过程中，每次遇到 yield 时函数会暂停并保存当前所有的运行信息，返回 yield 的值, 并在下一次执行 next() 方法时从当前位置继续运行。

调用一个生成器函数，返回的是一个迭代器对象。

以下实例使用 yield 实现斐波那契数列：

实例(Python 3.0+)

#!/usr/bin/python3

import sys

def fibonacci(n): # 生成器函数 - 斐波那契

a, b, counter = 0, 1, 0

while True:

if (counter > n):

return

yield a

a, b = b, a + b

counter += 1

f = fibonacci(10) # f 是一个迭代器，由生成器返回生成

while True:

try:

print (next(f), end=" ")

except StopIteration:

sys.exit()

执行以上程序，输出结果如下：

0 1 1 2 3 5 8 13 21 34 55

### Python3 解释器

Linux/Unix的系统上，一般默认的 python 版本为 2.x，我们可以将 python3.x 安装在 /usr/local/python3 目录中。

安装完成后，我们可以将路径 /usr/local/python3/bin 添加到您的 Linux/Unix 操作系统的环境变量中，这样您就可以通过 shell 终端输入下面的命令来启动 Python3 。

$ PATH=$PATH:/usr/local/python3/bin/python3 # 设置环境变量

$ python3 --version

Python 3.4.0

在Window系统下你可以通过以下命令来设置Python的环境变量，假设你的Python安装在 C:\Python34 下:

set path=%path%;C:\python34

交互式编程

我们可以在命令提示符中输入"Python"命令来启动Python解释器：

$ python3

执行以上命令后，出现如下窗口信息：

$ python3

Python 3.4.0 (default, Apr 11 2014, 13:05:11)

[GCC 4.8.2] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>>

在 python 提示符中输入以下语句，然后按回车键查看运行效果：

print ("Hello, Python!");

以上命令执行结果如下：

Hello, Python!

当键入一个多行结构时，续行是必须的。我们可以看下如下 if 语句：

>>> flag = True

>>> if flag :

... print("flag 条件为 True!")

...

flag 条件为 True!

脚本式编程

将如下代码拷贝至 hello.py文件中：

print ("Hello, Python!");

通过以下命令执行该脚本：

python3 hello.py

输出结果为：

Hello, Python!

在Linux/Unix系统中，你可以在脚本顶部添加以下命令让Python脚本可以像SHELL脚本一样可直接执行：

#! /usr/bin/env python3

然后修改脚本权限，使其有执行权限，命令如下：

$ chmod +x hello.py

执行以下命令：

./hello.py

输出结果为：

Hello, Python!

Python3 基本数据类型Python3 注释

2 篇笔记 写笔记

chendu

987\*\*\*757@qq.com

### 其他常见Python 解释器

Python 解释器可不止一种哦，有 CPython、IPython、Jython、PyPy 等。

顾名思义，CPython 就是用 C 语言开发的了，是官方标准实现，拥有良好的生态，所以应用也就最为广泛了。

而 IPython 是在 CPython 的基础之上在交互式方面得到增强的解释器（http://ipython.org/）。

Jython 是专为 Java 平台设计的 Python 解释器（http://www.jython.org/），它把 Python 代码编译成 Java 字节码执行。

PyPy 是 Python 语言（2.7.13和3.5.3）的一种快速、兼容的替代实现（http://pypy.org/），以速度快著称。

### list 和 tuple 有什么区别？

list 长度可变，tuple不可变；

list 中元素的值可以改变，tuple 不能改变；

list 支持append; insert; remove; pop等方法，tuple 都不支持

### Python 中的 list 和 dict 是怎么实现的？

### python中set、deque、list、dict内部实现原理

几大重要的数据结构：数组（线性表）、链表、hash表

分别对应了python中的list、deque、dict、set，但其内部实现原理比C的要复杂不少，也有很多好用的接口

#### list

python中的list要满足以下要求：

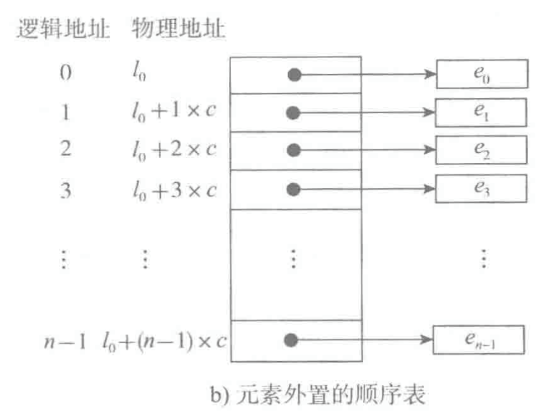
　　是数组类型，以索引就可以直接取到元素，O(1)取元素

　　元素可以是任意类型

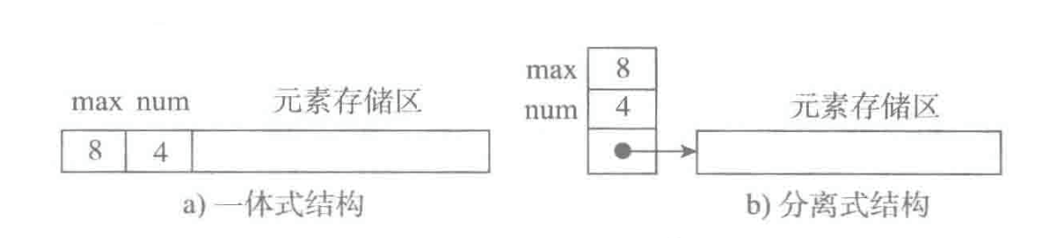
　　可以任意添加新元素，容易扩充

基于以上要求，list的实现设计为：

　　1. 元素外置，每个元素只存储index



　　2. 分离式结构，表内元素和表头信息不在一起



　　3. 动态扩充



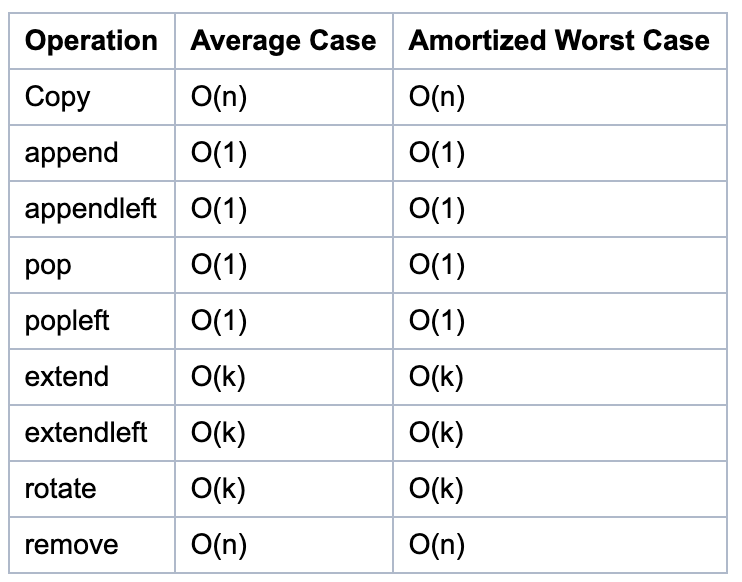
　　在python中创建空list时，会申请一个8个元素大小的内存区域。以后如果满了，就扩容4倍，不断反复，直至当元素总数达到50000时，再扩容就改为2倍

各操作时间复杂度：



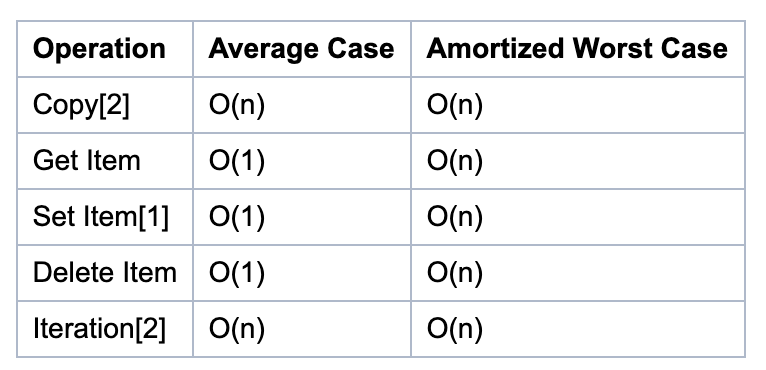
#### deque双向队列

各操作时间复杂度：



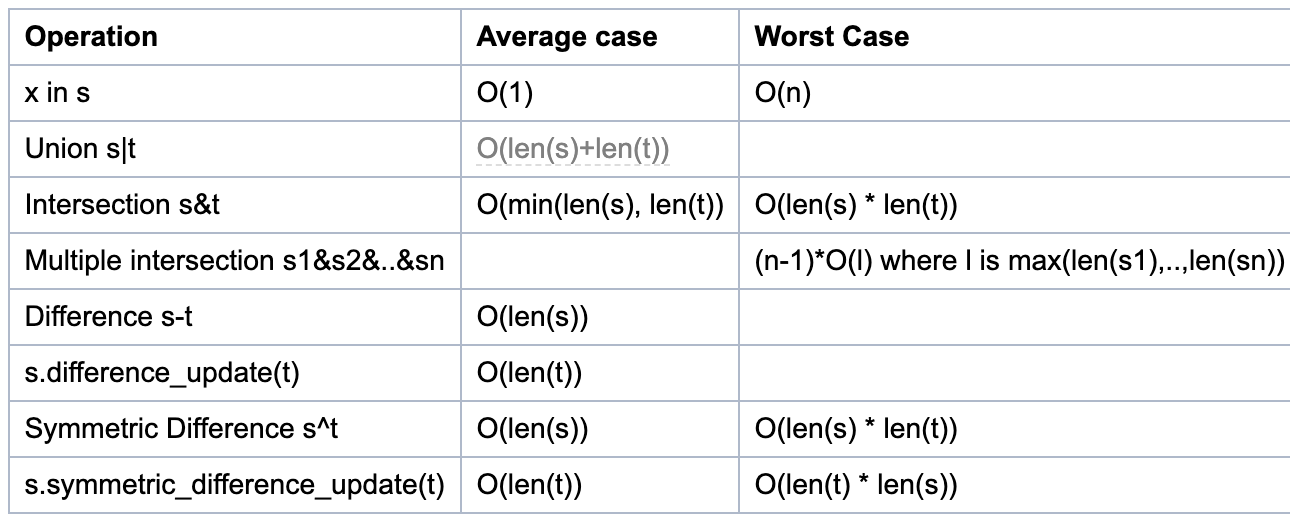
#### dict

各操作时间复杂度：



#### set

各操作时间复杂度：



判断值是否存在，千万不要用list，list查找的复杂度为O(n)，而set和dict由于用hash实现，查找的复杂度为O(1)

### Python 中使用多线程可以达到多核CPU一起使用吗？

Python中有一个被称为Global Interpreter Lock（GIL）的东西，它会确保任何时候你的多个线程中，只有一个被执行。线程的执行速度非常之快，会让你误以为线程是并行执行的，但是实际上都是轮流执行。经过GIL这一道关卡处理，会增加执行的开销。

可以通过多进程实现多核任务。

### 什么是装饰器？

Python 中的垃圾回收机制？

Python垃圾回收机制--完美讲解!

### 什么是 lambda 表达式？

简单来说，lambda表达式通常是当你需要使用一个函数，但是又不想费脑袋去命名一个函数的时候使用，也就是通常所说的匿名函数。

lambda表达式一般的形式是：关键词lambda后面紧接一个或多个参数，紧接一个冒号“：”，紧接一个表达式

### 什么是深拷贝和浅拷贝？

赋值（=），就是创建了对象的一个新的引用，修改其中任意一个变量都会影响到另一个。

浅拷贝 copy.copy：创建一个新的对象，但它包含的是对原始对象中包含项的引用（如果用引用的方式修改其中一个对象，另外一个也会修改改变）

深拷贝：创建一个新的对象，并且递归的复制它所包含的对象（修改其中一个，另外一个不会改变）{copy模块的copy.deepcopy()函数}

### Python 直接赋值、浅拷贝和深度拷贝解析

直接赋值：其实就是对象的引用（别名）。

浅拷贝(copy)：拷贝父对象，不会拷贝对象的内部的子对象。

深拷贝(deepcopy)： copy 模块的 deepcopy 方法，完全拷贝了父对象及其子对象。

字典浅拷贝实例

实例

>>>a = {1: [1,2,3]}

>>> b = a.copy()

>>> a, b

({1: [1, 2, 3]}, {1: [1, 2, 3]})

>>> a[1].append(4)

>>> a, b

({1: [1, 2, 3, 4]}, {1: [1, 2, 3, 4]})

深度拷贝需要引入 copy 模块：

实例

>>>import copy

>>> c = copy.deepcopy(a)

>>> a, c

({1: [1, 2, 3, 4]}, {1: [1, 2, 3, 4]})

>>> a[1].append(5)

>>> a, c

({1: [1, 2, 3, 4, 5]}, {1: [1, 2, 3, 4]})

### 双等于和 is 有什么区别？

==比较的是两个变量的 value，只要值相等就会返回True

is比较的是两个变量的 id，即id(a) == id(b)，只有两个变量指向同一个对象的时候，才会返回True

但是需要注意的是，比如以下代码：

a = 2

b = 2

print(a is b)

按照上面的解释，应该会输出False，但是事实上会输出True，这是因为Python中对小数据有缓存机制，-5~256之间的数据都会被缓存。

其它 Python 知识点

### 类型转换

list(x)

str(x)

set(x)

int(x)

tuple(x)

try...except

list

lst[a:b]：左闭右开

lst.append(value)：在末尾添加元素，复杂度O(1)

lst.pop()：弹出列表末尾元素，复杂度O(1)

lst.pop(index)：弹出任意位置元素，将后面的元素前移，复杂度O(n)

lst.insert(index, value)：插入元素，后面的元素后移，复杂度O(n)

lst.remove(value)：移除等于value的第一个元素，后面的元素前移，复杂度O(n)

lst.count(value)：计数值为value的元素个数

lst.sort(reverse = False)：排序，默认升序

# 六大设计原则(SOLID)

陈琰AC

1

2019.06.28 01:07:23

字数 743

阅读 40,550

一、SOLID

设计模式的六大原则有：

Single Responsibility Principle：单一职责原则

Open Closed Principle：开闭原则

Liskov Substitution Principle：里氏替换原则

Law of Demeter：迪米特法则

Interface Segregation Principle：接口隔离原则

Dependence Inversion Principle：依赖倒置原则

把这六个原则的首字母联合起来（两个 L 算做一个）就是 SOLID （solid，稳定的），其代表的含义就是这六个原则结合使用的好处：建立稳定、灵活、健壮的设计。下面我们来分别看一下这六大设计原则。

二、单一职责原则（Single Responsibility Principle）

一个类应该只有一个发生变化的原因

There should never be more than one reason for a class to change.

六大设计原则之单一职责原则（SRP）

三、开闭原则（Open Closed Principle）

一个软件实体，如类、模块和函数应该对扩展开放，对修改关闭

Software entities like classes, modules and functions should be open for extension but closed for modification

六大设计原则之开闭原则（OCP）

四、里氏替换原则（Liskov Substitution Principle）

所有引用基类的地方必须能透明地使用其子类的对象

Functions that use use pointers or references to base classes must be able to use objects of derived classes without knowing it.

六大设计原则之里氏替换原则（LSP）

五、迪米特法则（Law of Demeter）

只与你的直接朋友交谈，不跟“陌生人”说话

Talk only to your immediate friends and not to strangers

其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

六大设计原则之迪米特法则（LOD）

六、接口隔离原则（Interface Segregation Principle）

1、客户端不应该依赖它不需要的接口。

2、类间的依赖关系应该建立在最小的接口上。

Clients should not be forced to depend upon interfaces that they don`t use.

The dependency of one class to another one should depend on the smallest possible.

注：该原则中的接口，是一个泛泛而言的接口，不仅仅指Java中的接口，还包括其中的抽象类。

六大设计原则之接口隔离原则（ISP）

七、依赖倒置原则（Dependence Inversion Principle）

1、上层模块不应该依赖底层模块，它们都应该依赖于抽象。

2、抽象不应该依赖于细节，细节应该依赖于抽象。