go后端面试目录

Go入门

1、**与其他语言相比，go的好处**

- Go的垃圾回收是自动的
- Go是静态编译语言，能够预先编译，类似c、Java，python动态解释语言
- 兼具python等动态语言的开发速度和C语言等编译型语言的性能与安全性
- 访问底层操作系统以及提供了强大的网络编程和并发编程支持
- Go 语言开发的开源项目，其中包括docker、Go-Ethereum、Thrraform 和 Kubernetes

2、golang使用什么数据类型

- 内置类型3
- 引用类型4
- 结构体

3、go程序中的包是什么

- Go 工作区中包含 Go 源文件或其他包的目录，每 个 Go 源文件都属于一个包，package/import

4、nil切片和空切片?

- nil切片指向的数组引用地址是0，就是不指向任何实际地址，var声明；空切片的指向的数组引用地址是有值的，make([]int,0)

5、字符串转成byte切片，会发生内存拷贝吗？会，只要类型强转都会发生内存拷贝

6、翻转含有中文数字英文字母的字符串?

golang面试题：翻转含有中文、数字、英文字母的字符串 (qq.com)

7、拷贝大切片一定比小切片代价大吗?

golang面试题：拷贝大切片一定比小切片代价大吗？ (qq.com)

8、json包变量不加tag会怎样

- 如果变量 `首字母小写` ，则为 `private` 。无论如何 `不能转` ，因为取不到 `反射信息` 。
- 如果变量 `首字母大写` ，则为 `public` 。
  - `不加tag` ，可以正常转为 `json` 里的字段， `json` 内字段名跟结构体内字段 `原名一致` 。
  - `加了tag` ，从 `struct` 转 `json` 的时候， `json` 的字段名就是 `tag` 里的字段名，原字段名已经没用。

9、reflect如何获取字段tag？为什么json包不能导出私有变量的tag

golang面试题：reflect（反射包）如何获取字段tag？为什么json包不能导出私有变

量的tag？ (qq.com)

- `printTag` 方法传入的是j的指针。
- `reflect.TypeOf(stru).Elem()` 获取指针指向的值对应的结构体内容。
- `NumField()` 可以获得该结构体的含有几个字段。
- 遍历结构体内的字段，通过 `t.Field(i).Tag.Get("json")` 可以获取到 `tag` 为 `json` 的字段。
- 如果结构体的字段有 `多个tag` ，比如叫 `otherTag` ,同样可以通过 `t.Field(i).Tag.Get("otherTag")` 获得。

10、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

11、go支持什么形式的类型转换？显式的，即需要明确指定要转换的类型，不支持隐式即自动转（存在将一个大的整数转换为小的整数类型，导致溢出）

12、Log包线程安全吗？安全

13、**goroutine和线程的区别**?

从调度上看，goroutine远远小于线程。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。Goroutine采用M:N调度技术，M个Goroutine可以映射到N个系统线程上

从栈空间上，goroutine的栈空间更加动态灵活。线程有固定大小的栈内存差不多2Mb，goroutine生命周期开始只有一个很小的栈2Kb，不是固定大小按需增大缩小轻量级-Goroutine初始堆栈空间2KB，且按需增大缩小；线程占用一定的内存和CPU资源，堆栈大小通常是固定的，如2MB

调度方式-Goroutine由Go语言的运行时（runtime）进行调度，可以在用户空间中进行，效率更高，调度器使用了GMP；线程的调度是操作系统的内核调度器，在用户模式和内核模式之间切换会有一定的开销

创建与销毁开销-Goroutine的创建和销毁开销较小；线程的创建和销毁则需要更多的资源，因为每个线程都有自己的堆栈和上下文信息

并发模型-Goroutine采用了一种称为M:N的模型，即M个Goroutine可以映射到N个系统线程上；线程则采用了一种1:1的模型，即每个线程都映射到一个操作系统线程上

~~内存管理-Goroutine的栈大小是动态调整；线程的栈大小通常是固定的~~

通信与同步-Goroutine之间的通信通过channel；线程之间的通信可能会涉及到共享内存的同步和互斥机制

数量限制-可以创建大量的Goroutine；过多的线程可能会导致资源耗尽或性能下降

14、什么是goruntine？如何停止?

- Goroutine（或称为goroutine）是Go语言中的并发执行单元。
- 通过channel、context包的Done()和select语句结合使用

15、Golang中除了加Mutex锁以外还有哪些方式安全读写共享变量？channel

16、无缓冲 Chan 的发送和接收是否同步?同步

17、**go语言的并发机制以及它所使用的CSP并发模型**

- goroutine和channel，go底层选择使用coroutine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。
- CSP模型是“以通信共享内存”
- GMP：在GMP模型中，M代表工作线程，是运行Goroutine的实体，P在GMP模型中代表处理器，包含了运行Goroutine的资源，每个P都有一个本地队列，用于存储等待运行的Goroutine。
  当需要执行一个Goroutine时，系统首先会尝试找到一个空闲的P和一个与之关联的M。然后，M从P的本地队列中获取Goroutine并执行。如果P的本地队列为空，M会尝试从全局队列获取Goroutine，或者从其他P的本地队列中“偷取”Goroutine。
  这种设计使得Goroutine的调度更加轻量级和高效，减少了线程切换的开销，从而提高了并发性能

18、为什么P的逻辑不直接加在M上?

- 主要还是因为`M`其实是**内核**线程，内核只知道自己在跑线程，而`golang`的运行时（包括调度，垃圾回收等）其实都是**用户空间**里的逻辑。

19、**golang常用的并发模型?

- 通过channel通知实现并发控制
- 通过sync包中的WaitGroup实现并发控制
- Context上下文，实现并发控制

20、JSON 标准库对 nil slice 和 空 slice 的处理是一致的吗?

- 不一致，使用JSON标准库（如`encoding/json`）进行编码时，nil slice会被编码为JSON中的null值，而空slice则会被编码为一个空的JSON数组（例如`[]`）

21、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。

22、互斥锁，读写锁，死锁问题是怎么解决。

23、Golang的内存模型，为什么小对象多了会造成gc压力。

- 小对象的频繁分配和回收意味着GC需要更频繁地运行，以释放不再使用的内存空间。这增加了GC的负担，可能导致程序在执行过程中出现延迟或卡顿，尤其是在业务繁忙时
- 通常小对象过多会导致GC三色法消耗过多的GPU，基于标记清除算法的GC需要遍历内存中的所有对象，以标记出仍然活跃的对象，并清除那些不再被引用的对象

24、Go中的锁有哪些?

- 互斥锁，读写锁，sync.Map的安全的锁

25、**什么是channel，为什么它可以做到线程安全**？看底层代码

- 可以把它看成一个管道，通过它goroutine可以通信，也可以理解是一个先进先出的队列，通过管道进行通信
- 发送一个数据到Channel 和从Channel接收一个数据都是原子性的，channel在底层实现时使用了锁

26、读写锁或者互斥锁读的时候能写吗?

- 可以同时读，但读写、写写都互斥

27、**怎么限制Goroutine的数量**?

- channel、 `sync` 包中`Semaphore`限制goroutine数量、创建一个固定大小的goroutine池

28、Channel是同步的还是异步的?

- 同步还是异步，取决于channel的类型（无缓冲还是带缓冲）以及当前的使用情况（例如，缓冲区是否已满或为空）

29、**Data Race问题怎么解决？能不能不加锁解决这个问题?**

数据竞态-多个goroutine同时访问同一数据时，至少有一个goroutine在修改（写）这块内存，导致结果不可预测或错误。最终导致程序崩溃、数据损坏、难以复现

- 使用同步机制：`sync.Mutex`或`sync.RWMutex`等加锁机制，确保同一时间只有一个goroutine可以访问共享数据
- 避免共享数据：通过Channel来传递类型化的数据，实现goroutine之间的同步，避免直接共享内存
- 使用原子操作：`sync/atomic`包提供了原子内存操作支持。原子操作可以保证任何时刻只有一个goroutine可以操作数据
- 使用并发安全的数据结构：如sync.Map
- 使用数据竞态检测感觉：可以在运行时检测并报告数据竞态

30、如何在运行时检查变量类型?

- 反射

31、Go 两个接口之间可以存在什么关系?

- 等价：如果两个接口有相同的方法列表，那么他们就是等价的，可以相互赋值。
- 子集：如果接口 A的方法列表是接口 B的方法列 表的自己，那么接口 B可以赋值给接口 A。
- 接口查询是否成功（即一个接口能否赋值给另一个接口）通常需要在运行时确定，因为具体的类型信息可能只有在运行时才可用

32、Go 语言当中值传递和地址传递（引用传递）如何运用？

- 值传递只会把参数的值复制一份放进对应的函数，两个变量的地址不同，不可相互修改。 数组
- 地址传递(引用传递)会将变量本身传入对应的函数，在函数中可以对该变量进行值内容的修改。slice

33、**Go 语言是如何实现切片扩容的？扩容前后的 Slice 是否相同？**

- 判断当前容量是否足够：如果当前容量足够，新元素将被直接添加到切片的末尾，并更新切片的长度，但容量保持不变
  如果当前容量不足，将触发扩容机制
- 扩容策略: 如果当前容量小于 256，则将容量翻倍。
  如果当前容量大于等于 256 且小于 4096，则将容量增加 1.5 倍。
  如果当前容量大于等于 4096，则将容量增加 1.25 倍。
- 根据扩容策略计算新的容量后，分配新的内存空间
- 拷贝旧数据至新分配的内存
- 更新切片的长度和容量，并调整切片的指针以指向新的内存空间

情况一：原数组还有容量可以扩容（实际容量没有填充完），这种情况下，扩容以后的数组还是指向原来的数组，对一个切片的操作可能影响多个指针指向相同地址的 Slice。

情况二：原来数组的容量已经达到了最大值，再想扩容， Go 默认会先开一片内存区域，把原来的值拷贝过来，然后再执行 append()操作。这种情况丝毫不影响原数组。

要复制一个 Slice，最好使用 Copy函数。

34、defer 的执行顺序是什么

- 延迟执行/多个 defer 执行顺序是逆序执行

35、**Golang Map 底层实现？如何扩容？**

- Golang 的 Map 底层是一个哈希表，这个哈希表由一个数组组成，数组的每个元素被称为“桶”（Bucket），用于存储键值对
- 每个桶内部可以包含一个数据结构（如链表或动态数组），用于存储具有相同哈希值的多个键值对，以解决哈希冲突
- 散列冲突处理：当多个键具有相同的哈希值时，会发生散列冲突。Golang 使用链地址法（Separate Chaining）来处理冲突，即每个桶内部使用链表或其他数据结构来存储所有具有相同哈希值的键值对
- 扩容：

36、**Channel 的 ring buffer 实现**

- ring buffer环形缓冲区

Go进阶

37、for select时，如果通道已经关闭会怎么样？如果只有一 个case呢？

- for循环`select`时，如果其中一个case通道已经关闭，则每次都会执行到这个case。

- 如果select里边只有一个case，而这个case被关闭了，则会出现死循环。
golang面试官：for select时，如果通道已经关闭会怎么样？如果select中只有一个case呢？_(qq.com)

38、对已经关闭的的chan进行读写，会怎么样？为什么？

- 读**已经关闭**的 `chan` 能一直读到东西，但是读到的内容根据通道内`关闭前`是否有元素而不同。
  - 如果 `chan` 关闭前，`buffer` 内有元素**还未读**，会正确读到 `chan` 内的值，且返回的第二个 bool 值（是否读成功）为 `true`。
  - 如果 `chan` 关闭前，`buffer` 内有元素**已经被读完**，`chan` 内无值，接下来所有接收的值都会非阻塞直接成功，返回 `channel` 元素的**零值**，但是第二个 `bool` 值一直为 `false`。
- 写**已经关闭**的 `chan` 会 `panic`
- channel底层读取的时候，会传val的地址
golang面试题：对已经关闭的的chan进行读写，会怎么样？为什么？_(qq.com)

39、对未初始化的的chan进行读写，会怎么样？为什么？

- panic，nil的channel没有被分配内存（堆上）

40、能说说uintptr和unsafe.Pointer的区别吗？
golang面试题：能说说uintptr和unsafe.Pointer的区别吗？_(qq.com)

- `uintptr` 是一个无符号整数类型，用于存储指针地址并进行指针运算，
- `unsafe.Pointer` 是一种特殊的指针类型，它可以包含任意类型变量的地址

41、**简单聊聊内存逃逸？怎么避免内存逃逸？**

- 在函数内部分配的变量在函数执行完后仍然被其他部分引用，导致变量逃逸到堆上分配内存，而不是在栈上分配内存。这种情况会增加垃圾回收的负担，降低程序执行效率。
- **内存逃逸的原因**：
  1. 变量的生命周期超出作用域：函数内部声明的变量，在函数返回后仍被引用，就会导致内存逃逸。这些变量将被分配到堆上，以确保它们在函数返回后仍然可用。
  2. 引用外部变量：如果函数内部引用了外部作用域的变量，这也可能导致内存逃逸。编译器无法确定这些外部变量的生命周期，因此它们可能会被分配到堆上。
  3. 使用闭包：在Go中，闭包（函数值）可以捕获外部变量，这些变量的生命周期可能超出了闭包本身的生命周期。这导致了内存逃逸。
- 能引起变量逃逸到堆上的**典型情况**：
  1. 在方法内把局部变量指针返回： 局部变量原本应该在栈中分配，在栈中回收。但是由于返回时被外部引用，因此其生命周期大于栈，则溢出。
  2. 发送指针或带有指针的值到 channel 中：在编译时，是没有办法知道哪个 goroutine 会在 channel 上接收数据。所以编译器没法知道变量什么时候才会被释放。

3. 在一个切片上存储指针或带指针的值：一个典型的例子就是 []string 。这会导致切片的内容逃逸。尽管其后面的数组可能是在栈上分配的，但其引用的值一定是在堆上。
4. slice 的背后数组被重新分配了，因为 append 时可能会超出其容量(cap)： slice 初始化的地方在编译时是可以知道的，它最开始会在栈上分配。如果切片背后的存储要基于运行时的数据进行扩充，就会在堆上分配。
5. 在 interface 类型上调用方法：在 interface 类型上调用方法都是动态调度的 —— 方法的真正实现只能在运行时知道。想像一个 io.Reader 类型的变量 r , 调用 r.Read(b) 会使得 r 的值和切片b 的背后存储都逃逸掉，所以会在堆上分配。
   高频golang面试题：简单聊聊内存逃逸？ (qq.com)

- **如何避免内存逃逸**：
  1. 尽量使用局部变量：局部变量更有可能在栈上分配，从而避免内存逃逸，尽量避免在函数间传递大的结构体或数组。
  2. 使用指针传递而非值传递：对于大的结构体或数组，使用指针传递可以减少因复制而导致的性能开销和内存逃逸风险。
  3. 避免不必要的闭包：因为闭包可能会导致外部变量逃逸，尤其是当这些闭包被传递到其他函数中去时。
  4. 使用编译器逃逸分析工具： `go build -gcflags=-m`
     golang面试题：怎么避免内存逃逸？ (qq.com)

     42、并发编程概念是什么?
- 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。

  43、Mutex 几种状态?
- locked、woken、starving、waitersCount

  44、Mutex 正常模式和饥饿模式
- 在正常模式下，`sync.Mutex`会公平地处理多个goroutine对锁的请求。当多个goroutine同时请求锁时，它们会按照先进先出（FIFO）的顺序被放入一个等待队列中。当锁被释放时，等待队列中的第一个goroutine会获得锁。在正常模式下，如果等待锁的goroutine在一段时间内（如1毫秒）无法获得锁，那么`sync.Mutex`可能会切换到饥饿模式
- 当等待锁的goroutine在一段时间内（如1毫秒）无法获得锁时，`sync.Mutex`会自动切换到饥饿模式。在饥饿模式下，锁的所有权会直接从释放锁的goroutine传递给等待队列中的第一个goroutine，而不会让新到达的goroutine尝试获取锁。
- 饥饿模式会一直持续，直到等待锁的goroutine满足某些条件（如成为等待队列中的最后一个goroutine，或者等待时间不超过1毫秒）时，`sync.Mutex`才会切换回正常模式

  45、**WaitGroup 用法?**

- 实现原理：WaitGroup是Go语言中sync包中的一个结构体，用来等待所有的goroutine完成任务
  WaitGroup内部维护了一个计数器，可以通过Add()方法增加计数器的值，Done()方法减少计数器的值，Wait()方法用于阻塞当前goroutine，直到计数器归零
  WaitGroup 主要维护了2 个计数器，一个是请求计数器 v，一个是等待计数器 w，二者组成一个64bit 的值， 请求计数器占高32bit，等待计数器占低32bit。 每次Add执行，请求计数器 v 加1，Done方法执行，请求计数器减1，v 为0 时通过信号量唤醒 Wait()
  46、sync.Once
- 确保只执行一次
- 主要应用场景：
  单例模式：确保类的实例只被创建一次，比如初始化配置、保持数据库连接等。
  延迟初始化：可以在代码的任意位置初始化和调用，延迟到实际使用时再执行，避免不必要的资源消耗和程序加载时间
- `sync.Once` 的实现原理如下：
  - 它包含一个互斥锁（`sync.Mutex`）和一个标志位（通常是一个 `uint32` 类型的 `done` 字段），用于记录函数是否已经被执行过
  - `Do` 方法会首先检查标志位，如果函数已经被执行过（标志位不为0），则直接返回。否则，它会获取互斥锁，再次检查标志位，如果仍未被执行，则执行函数 `f`，并将标志位设置为1，以确保下次调用时不会再次执行 `f`
    47、什么操作叫做原子操作？原子操作和锁的区别？
- 要么都执行要么都不执行，中间状态对外不可见，不被中断
- 原子操作由底层硬件支持，而锁则由操作系统的调度器实现。
  48、什么是 CAS

Go高阶

49、内存泄漏？

- 获取长字符串中的一段导致长字符串未释放
- 获取长slice中的一段导致长slice未释放
- 在长slice新建slice导致泄漏
- goroutine泄漏
- time.Ticker未关闭导致泄漏
- Finalizer导致泄漏
- Deferring Function Call导致泄漏
  50、**GMP 调度流程?**
- 在GMP模型中，M代表工作线程，是运行Goroutine的实体，P在GMP模型中代表处理器，包含了运行Goroutine的资源，每个P都有一个本地队列，用于存储等待运行的Goroutine。

- 当需要执行一个Goroutine时，系统首先会尝试找到一个空闲的P和一个与之关联的M。然后，M从P的本地队列中获取Goroutine并执行。如果P的本地队列为空，M会尝试从全局队列获取Goroutine，或者从其他P的本地队列中“偷取”Goroutine。
  这种设计使得Goroutine的调度更加轻量级和高效，减少了线程切换的开销，从而提高了并发性能
  51、GMP 调度过程中存在哪些阻塞
- I/O，select 、block on syscall 、channel 、等待锁、 runtime.Gosched()
  52、**GC 触发时机？GC 如何调优**
- 主动触发：调用 runtime.GC
- 被动触发：
  使用系统监控，该触发条件由 runtime.forcegcperiod 变量控制，默认为2 分钟。当超过两分钟没有产生任何 GC 时，强制触发 GC。
  使用步调（Pacing）算法，其核心思想是控制内存增长的比例。
- 调优： 通过 go tool pprof 和 go tool trace 等工具
  控制内存分配的速度，限制 goroutine 的数量，从而提高赋值器对 CPU 的利用率。
  减少并复用内存，例如使用 sync.Pool 来复用需要频繁创建临时对象，例如提前分配足够的内存来降低多余的 拷贝。
  需要时，增大 GOGC 的值，降低 GC 的运行频率。
  53、**Go 语言中 GC 的流程是什么?**
- 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
- 是自动的，并且是分代的。这意味着对象的生命周期不同，因此它们被划分为不同的代（代）。新创建的对象被放置在第0代，当该代的对象经过一定次数垃圾收集周期后，还存活的对象会被晋升到第1代，第1代的对象如果更长时间仍然存活，则会被晋升到第2代。每个代都有不同的垃圾收集频率
  54、**Go语言的栈空间管理是怎么样的?**
  55、怎么查看Goroutine的数量?
- 使用runtime包中的NumGoroutine()函数
- 使用go tool pprof工具
- `GOMAXPROCS` 控制的是 Go 运行时能够同时使用的操作系统线程的最大数量，这会影响到 Goroutine 的并发执行能力，但它并不直接反映当前 Goroutine 的数量

微服务

分布式

1、分布式锁实现原理，用过吗?
2、说说 ZooKeeper 一般都有哪些使用场景?
3、分布式事务了解吗?
4、ZK 和 Redis 的区别，各自有什么优缺点?

Redis

1、redis数据类型?

- string（字符串），hash（哈希），list（列表），set（集合）及 zset
  2、Redis 是单进程单线程的?
- Redis 是单进程单线程的，redis 利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销。
  3、Redis 的持久化机制是什么？各自的优缺点?
- Redis 提供两种持久化机制 RDB 和 AOF 机制
- RDB：是指用数据集快照的方式半持久化模式)记录 redis 数据库的所有键值对,在某个时间点将数据写入一个临时文件， 持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复。
- AOF：是指所有的命令行记录以 redis 命令请求协议的格式完全持久化存储)保存为 aof 文件。
  4、怎么理解 Redis 事务?
- 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会 被其他客户端发送来的命令请求所打断。
- 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。
  5、MySQL 里有2000w 数据，Redis 中只存20w 的数据，如何保证 redis 中 的数据都是热点数据?
- Redis 内存数据集大小上升到一定大小的时候，就 会施行数据淘汰策略。
  6、使用过 Redis 分布式锁么，它是什么回事?
- 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。
  7、**如果避免缓存”穿透”的问题?**
- 高并发地访问，热点数据失效后，大量请求同时涌入后端存储，导致后端存储负载增大
- 永不过期/缓存预热/使用互斥锁或者分布式锁来对并发请求进行控制，避免对同一资源的并发读写竞争
  8、**如何避免缓存“雪崩”的问题?**
- 缓存中大量的数据同时失效或过期，导致后续请求都落到后端存储上，从而引起系统负载暴增
- 设置不同的过期时间
  9、**如果避免缓存“击穿”的问题?**
- 攻击者通过请求缓存和后端存储中不存在的数据，使得所有请求落到后端存储上，导致系统瘫痪
- 布隆过滤器(数据结构，用于快速检索一个元素是否存在,足够大)/黑白名单/缓存预热/如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），把这个空

结果进行缓存，但它的过期时间会很短，最长不超过五分钟

10、**缓存数据的淘汰策略有哪些?**

- 1、定时去清理过期的缓存。
- 2、当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓 存。

Memcached

Memcached 是什么，有什么作用?
Memcached 与 Redis 的区别?

RabbitMQ

RabbitMQ routing 路由模式
RabbitMQ 有那些基本概念?
消息如何分发?
什么是消费者 Consumer?

RocketMQ

RocketMQ 由哪些角色组成，每个角色作用和特点是什么?

角色	作用
Nameserver	无状态，动态列表；这是和 ZooKeeper 的重要区别之一。ZooKeeper 是有状态的。
Producer	消息生产者，负责发消息到 Broker。
Broker	就是 MQ 本身，负责收发消息、持久化消息等。
Consumer	消息消费者，负责从 Broker 上拉取消息进行消费，消费完进行 ack。

RocketMQ 消费模式有几种?
消费模型由 Consumer 决定，消费维度为 Topic。
集群消费
广播消费
Broker 如何处理拉取请求的?

Consumer 首次请求 Broker。

- Broker 中是否有符合条件的消息
- 有
  - 响应 Consumer。
  - 等待下次 Consumer 的请求。
- 没有
  - DefaultMessageStore#ReputMessageService#run 方法。 - PullRequestHoldService 来 Hold 连接，每个 5s 执行一次检查pullRequestTable 有没有消息，有的话立即推送。
  - 每隔 1ms 检查 commitLog 中是否有新消息，有的话写入到pullRequestTable。
  - 当有新消息的时候返回请求。
  - 挂起 consumer 的请求，即不断开连接，也不返回数据。
  - 使用 consumer 的 offset。

RocketMQ 如何保证消息不丢失
Producer 端如何保证消息不丢失
Broker 端如何保证消息不丢失
Consumer 端如何保证消息不丢失

Kafka

Kafka 是什么？主要应用场景有哪些?
什么是 Producer、Consumer、Broker、Topic、 Partition?
Kafka 如何保证消息不丢失?
和其他消息队列相比，Kafka 的优势在哪里?
Zookeeper 在 Kafka 中的作用?

容器docker

Docker 与虚拟机有何不同?
容器
镜像
Dockerfile

MySQL

1、隔离级别与锁的关系？MySQL 中有哪几种锁?
2、MySQL 中都有哪些触发器?

- Before Insert-After Insert -Before Update - After Update - Before Delete - After Delete
  3、大表怎么优化？分库分表是怎么做的？分表分库有什么问题？ 有用到 中间件么？他们的原理知道么?
  4、B+ Tree 索引和 Hash 索引区别?
  5、MySQL 存储引擎 MyISAM 与 InnoDB 区别?
  6、什么是索引？MyISAM 索引与 InnoDB 索引的区别?

7、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?
8、MyISAM 表类型将在哪里存储，并且还提供其存储格式?
9、MySQL 的 Binlog 有有几种录入格式？分别有什么区别?
10、MySQL 有关权限的表都有哪几个?
11、什么是临时表，何时删除临时表?
12、最左前缀原则？最左匹配原则
13、聚簇索引？何时使用聚簇索引与非聚簇索引
14、MySQL 查询缓存
15、分析器？优化器？执行器?
16、外链接？内链接?

MongDB

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?

ClickHouse

什么是 ClickHouse?

- ClickHouse是一个高性能的列式数据库管理系统（DBMS）
  ClickHouse 列式存储的优缺点有哪些?

Elasticsearch

什么是es？运用?
es的索引、node

Linux

Linux 中进程有哪几种状态？在 ps 显示出来的信息中，分别用什么符号表 示的?

网络