Kafka专业术语，参考 [Apache kafka 工作原理介绍](https://www.ibm.com/developerworks/cn/opensource/os-cn-kafka/index.html)

**特点：异步、削峰、解耦**

**Broker**：Kafka 集群包含一个或多个服务器，这种服务器被称为 broker。

**Topic**：每条发布到 Kafka 集群的消息都有一个类别，这个类别被称为 Topic。（物理上不同 Topic 的消息分开存储，逻辑上一个 Topic 的消息虽然保存于一个或多个 broker 上，但用户只需指定消息的 Topic 即可生产或消费数据而不必关心数据存于何处）。

**Partition**：Partition 是物理上的概念，每个 Topic 包含一个或多个 Partition。

Replica：副本 包括**leader和follower**

**Producer**：负责发布消息到 Kafka broker。

**Consumer**：消息消费者，向Kafka broker 读取消息的客户端。

**Consumer Group**：每个 Consumer 属于一个特定的 Consumer Group（可为每个 Consumer 指定 group name，若不指定 group name 则属于默认的 group）。

对于**消费者**，kafka中有两个设置的地方：对于老的消费者，由**--zookeeper参数**设置；对于新的消费者，由**--bootstrap-server参数**设置

如果使用了--zookeeper参数,那么consumer的信息将会存放在zk之中

如果使用了--bootstrap-server参数,那么consumer的信息将会存放在kafka之中

对于**console生产者**，**--broker-list参数**指定了所使用的broker

zookeeper 是 kafka 不可分割的一部分，可见其重要程度，所以我们有必要了解一下 zookeeper 在 kafka 中的具体工作内容。

而且，这也是面试时经常问的。

zookeeper 存储了一些关于 consumer 和 broker 的信息，那么就从这两方面说明 zookeeper 的作用。

1. broker

状态

zookeeper 记录了所有 broker 的存活状态，broker 会向 zookeeper 发送心跳请求来上报自己的状态。

zookeeper 维护了一个正在运行并且属于集群的 broker 列表。

控制器选举

kafka 集群中有多个 broker，其中有一个会被选举为控制器。

控制器负责管理整个集群所有分区和副本的状态，例如某个分区的 leader 故障了，控制器会选举新的 leader。

从多个 broker 中选出控制器，这个工作就是 zookeeper 负责的。

限额权限

kafka 允许一些 client 有不同的生产和消费的限额。

这些限额配置信息是保存在 zookeeper 里面的。

所有 topic 的访问控制信息也是由 zookeeper 维护的。

记录 ISR

ISR（in-sync replica） 是 partition 的一组同步集合，就是所有 follower 里面同步最积极的那部分。

一条消息只有被 ISR 中的成员都接收到，才被视为“已同步”状态。

只有处于 ISR 集合中的副本才有资格被选举为 leader。

zookeeper 记录着 ISR 的信息，而且是实时更新的，只要发现其中有成员不正常，马上移除。

node 和 topic 注册

zookeeper 保存了所有 node 和 topic 的注册信息，可以方便的找到每个 broker 持有哪些 topic。

node 和 topic 在 zookeeper 中是以临时节点的形式存在的，只要与 zookeeper 的 session 一关闭，他们的信息就没有了。

topic 配置

zookeeper 保存了 topic 相关配置，例如 topic 列表、每个 topic 的 partition 数量、副本的位置等等。

2. consumer

offset

kafka 老版本中，consumer 的消费偏移量是默认存储在 zookeeper 中的。

新版本中，这个工作由 kafka 自己做了，kafka 专门做了一个 offset manager。

注册

和 broker 一样，consumer 也需要注册。

consumer 会自动注册，注册的方式也是创建一个临时节点，consumer down 了之后就会自动销毁。

分区注册

kafka 的每个 partition 只能被消费组中的一个 consumer 消费，kafka 必须知道所有 partition 与 consumer 的关系。

https://blog.csdn.net/qq\_34668848/article/details/105611546

**Etcd**

etcd是一个分布式的、高可用的、一致的key-value存储数据库，基于Go语言实现，主要用于共享配置和服务发现。

高可用性：一个**leader得到写入后就可以返回给用户，然后异步的复制给副本，无需用户等待**

**一致性：副本会同步leader的数据**

**Raft没有脑裂，因为分区后，一个少数派一个多数派，leader会在多数派中选出来，少数派自动变成follower或者candidator，不会变成leader不会发生脑裂**

**二、为什么需要etcd**  
　　在分布式系统中，各种服务配置信息的管理共享和服务发现是一个很基本也是很重要的问题。etcd可集中管理配置信息，服务端将配置信息存储于etcd，客户端通过etcd得到服务配置信息，etcd监听配置信息的改变，发现改变通知客户端。  
为了防止单点故障，还可启动多个etcd组成集群。etcd集群使用raft一致性算法处理日志复制，保证多节点数据的强一致性。

**三、etcd的raft算法**

1. 主节点选举  
　　etcd集群中有一个主节点（leader，负责写操作），多个从节点（follower，负责读操作）。主节点会发送心跳包给从节点，从节点进行响应。从节点若超过一定时间（一定范围内的随机值）没有收到主节点的心跳包，则认为主节点已不可用，自身可成为候选主节点（candidate），发起投票，若超过一半节点响应，则可成为新的主节点（可能会有几轮争夺）。在每一轮投票中，参与投票的所有节点，只响应收到的第一个投票请求，对后续请求不作响应。  
2. 数据更新  
（1）第一阶段：主节点将修改记录到本地日志，并将日志复制给所有从节点，若超过一半节点响应，则认为操作成功，通知客户端。  
（2）第二阶段：主节点提交本地修改（持久化到磁盘），通知所有从节点也进行数据修改提交

Grpc利用etcd进行服务注册与发现，可以让客户端调用服务端在etcd上注册过的服务

在gRPC里，客户端可以直接调用不同机器上的服务应用的方法，就像是本地对象一样，所以创建分布式应用和服务就很简单了。在很多RPC(Remote Procedure Call Protocol)系统里，gRPC是基于定义一个服务，指定一个可以远程调用的带有参数和返回类型的的方法。在服务端，服务实现这个接口并且运行gRPC服务处理客户端调用。在客户端，有一个stub提供和服务端相同的方法。

**K8s**

**自动部署和回滚**

你可以使用 Kubernetes 描述已部署容器的所需状态，它可以以受控的速率将实际状态 更改为期望状态。例如，你可以自动化 Kubernetes 来为你的部署创建新容器， 删除现有容器并将它们的所有资源用于新容器。

**自动完成装箱计算**

Kubernetes 允许你指定每个容器所需 CPU 和内存（RAM）。 当容器指定了资源请求时，Kubernetes 可以做出更好的决策来管理容器的资源。

**自我修复**

Kubernetes 重新启动失败的容器、替换容器、杀死不响应用户定义的 运行状况检查的容器，并且在准备好服务之前不将其通告给客户端。

Development：灰度上线、

Replica：负载均衡？、注册发现、自动恢复

Pod：dockers

Docker

sudo docker run $(curl -sL http://localhost:3476/v1.0/docker/cli) -it --rm --name analyze\_test\_fix \

-p 8082:8082\

-v /home/xujingshuang/go/src/go.megvii-inc.com/securitycore:/go/src/go.megvii-inc.com/securitycore \

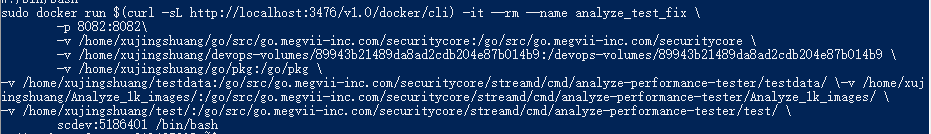
-v /home/xujingshuang/devops-volumes/89943b21489da8ad2cdb204e87b014b9:/devops-volumes/89943b21489da8ad2cdb204e87b014b9 \

-v /home/xujingshuang/go/pkg:/go/pkg \

-v /home/xujingshuang/testdata:/go/src/go.megvii-inc.com/securitycore/streamd/cmd/analyze-performance-tester/testdata/ \-v /home/xujingshuang/Analyze\_1k\_images/:/go/src/go.megvii-inc.com/securitycore/streamd/cmd/analyze-performance-tester/Analyze\_1k\_images/ \

-v /home/xujingshuang/test/:/go/src/go.megvii-inc.com/securitycore/streamd/cmd/analyze-performance-tester/test/ \

scdev:5186401 /bin/bash



docker是一个容器化平台，它以容器的形式将您的应用程序及其所有依赖项打包在一起，以确保您的应用程序在任何环境中无缝运行。  
docker镜像是docker容器的源代码，用于创建容器。使用build命令创建镜像。

# 什么是docker容器？docker容器有几种状态？docker容器内部机制？容器与主机之间的数据拷贝？启动容器并挂在目录？

（1）docker容器：docker容器包括应用程序及其所有依赖项，作为操作系统的独立进程运行。  
（2）docker容器4种状态：运行+已暂停+重新启动+已退出  
（3）docker容器内部机制：每个容器都在自己的命名空间中运行，但使用与所有其他容器完全相同的内核。发生隔离是因为内核知道分配给进程的命名空间，并且在API调用期间确保进程只能访问其自己的命名空间中的资源。  
    【操作系统的一个功能是允许将全局资源（如网络和磁盘）共享到进程。如果将这些全局资源包装在命名空间中，以使它们仅对在同一命名空间中运行的那些进程可见】  
（4）主机copy到容器：docker cp /www 96f7f14e99ab:/www/  容器copy到主机：docker cp 96f7f14e99ab:/www /tmp/  
（5）启动nginx容器(随机端口映射)，并挂载本地文件目录到容器html的命令：  
    docker run -d -P --name nginx2 -v /home/nginx:/usr/share/nginx/html nginx  
    docker run解读相见：https://www.jianshu.com/p/ea4a00c6c21c

多级反馈队列调度算法描述：  
1、该调度算法设置四级就绪队列：前三级就绪队列采用时间片轮转法，时间片大小分别为2、4和8；最后一级就绪队列采用FIFO调度。  
2、任务在进入待调度的队列等待时，首先进入优先级最高的队列等待。  
3、首先调度优先级高的队列中的任务。若高优先级中队列中已没有调度的任务，则调度次优先级队列中的任务，依次类推。  
4、对于同一个队列中的各个任务，按照队列指定调度方法调度。每次任务调度执行后，若没有完成任务，就被降到下一个低优先级队列中。  
5、在低优先级的队列中的任务在运行时，又有新到达的任务，那么在运行完这个时间片后，CPU马上分配给新到达的任务，即算法支持抢占式。  
6、为方便实现，时间以1为单位，用整数数据表示；且每个时间点，最多只有一个任务请求服务（即输入）。

堆、栈、静态存储区

<https://blog.csdn.net/xuehongnizhao/article/details/51074668>

Go 中切片扩容的策略是这样的：

如果切片的容量小于 1024 个元素，于是扩容的时候就翻倍增加容量。上面那个例子也验证了这一情况，总容量从原来的4个翻倍到现在的8个。

一旦元素个数超过 1024 个元素，那么增长因子就变成 1.25 ，即每次增加原来容量的四分之一。

reslice的增长规则：如果新的size是当前size的2倍以上，则大小增长为新size。如果新的size不到当前size的2倍，则按当前size的不同有不同操作。当前size不超过1024，按每次2倍增长，否则按当前大小的1/4增长。

Golang map 哈希<https://www.jianshu.com/p/aa0d4808cbb8、>

Map扩容条件：

扩容条件：  
1.map中数据总个数/桶个数>6.5（装载因子），引发翻倍扩容。mapassign中的overLoadFactor函数。  
2.使用了太多的溢出桶时（溢出桶使用的太多会导致map处理速度降低）。mapassign中的tooManyOverflowBuckets函数。

Go 语言的函数调用都是传值的

这也是为什么后调用的 defer 会优先执行。

Panic会先把defer执行完再执行Panic panic的顺序是先Panic的先进行 defer是后defer的先进行

通道关闭后 读出来的内容是0

G-P-M模型抽象结构：

G: 表示Goroutine，每个Goroutine对应一个G结构体，G存储Goroutine的运行堆栈、状态以及任务函数，可重用。G运行队列是一个栈结构，分全局队列和P绑定的局部队列，每个G不能独立运行，它需要绑定到P才能被调度执行。

P: Processor，表示逻辑处理器， 对G来说，P相当于CPU核，G只有绑定到P(在P的local runq中)才能被调度。对M来说，P提供了相关的执行环境(Context)，如内存分配状态(mcache)，任务队列(G)等，P的数量决定了系统内最大可并行的G的数量（前提：物理CPU核数 >= P的数量），P的数量由用户设置的GOMAXPROCS决定，但是不论GOMAXPROCS设置为多大，P的数量最大为256。

M: Machine，系统物理线程，代表着真正执行计算的资源，在绑定有效的P后，进入schedule循环；而schedule循环的机制大致是从Global队列、P的Local队列以及wait队列中获取G，切换到G的执行栈上并执行G的函数，调用goexit做清理工作并回到M，如此反复。M并不保留G状态，这是G可以跨M调度的基础，M的数量是不定的，由Go Runtime调整，为了防止创建过多OS线程导致系统调度不过来，目前默认最大限制为10000个。

G-P-M模型调度

我们来看看go关键字创建一个协程后其调度器是怎么工作的：

go关键字创建goroutine(G)，优先加入某个P维护的局部队列（当局部队列已满时才加入全局队列）；

P需要持有或者绑定一个M，而M会启动一个系统线程，不断的从P的本地队列取出G并执行；

M执行完P维护的局部队列后，它会尝试从全局队列寻找G，如果全局队列为空，则从其他的P维护的队列里窃取一般的G到自己的队列；

重复以上知道所有的G执行完毕。

当然也有一些情况会造成Goroutine阻塞，如：

系统GC；

系统IO资源的调用，如文件读写；

网络IO的延迟；

管道阻塞；

同步操作。

#### 多级缓存 [#](https://draveness.me/golang/docs/part3-runtime/ch07-memory/golang-memory-allocator/#%e5%a4%9a%e7%ba%a7%e7%bc%93%e5%ad%98)

内存分配器不仅会区别对待大小不同的对象，还会将内存分成不同的级别分别管理，TCMalloc 和 Go 运行时分配器都会引入线程缓存（Thread Cache）、中心缓存（Central Cache）和页堆（Page Heap）三个组件分级管理内存：

内存分配：<https://www.cnblogs.com/shijingxiang/articles/12196677.html>