宇宙最全系列 | 分布式系统面试题

写在前面

分布式系统理论

01 ❤分布式系统CAP理论知道吗。

02 ❤什么是强一致性、弱一致性、最终一致性。

03 ❤什么是Quorum机制。

04 拜占庭将军问题的含义。

05 ❤2PC和3PC算法知道吗？有什么不同。

06 ❤什么是幂等性？如何实现？

一致性协议和算法

01 介绍一下Paxos协议。

02 ❤介绍一下Raft协议。

03 ❤一致性哈希算法了解吗？

分布式系统技术

01 ❤集群“脑裂”怎么产生，如何解决。

02 什么是IO Fence技术？

03 关于分布式系统中间件了解多少？

04 ❤什么是分布式锁？如何实现之？

05 关于分布式消息队列了解多少？

06 关于etcd了解多少？

07 关于Docker了解多少？

08 关于Kubernetes了解多少？

09 ❤如何设计一套整点秒杀系统？讲一讲思路。

总结

国外分布式系统课程

作者：迹寒

写在前面

欢迎关注迹寒编程

随着云和大数据技术的高速发展，企业对高性能，高并发，高可用的要求越来越高。许多岗位都和云计算、云存储、分布式密切相关。对于希望从事相关岗位的同学有必要好好了解一下。

分布式系统主要涉及：

- 分布式系统理论：CAP、强弱一致性、Quorum机制；
- 一致性协议和算法：Paxos协议、Raft协议，一致性算法；
- 分布式系统技术：“脑裂”的产生和解决、IO Fence、秒杀系统的设计；

最好的教科书就是企业真实的生产环境。

赛道已铺好，只待尔努力！加油你就是offer收割机！

分布式系统理论

01 ❤分布式系统CAP理论知道吗。

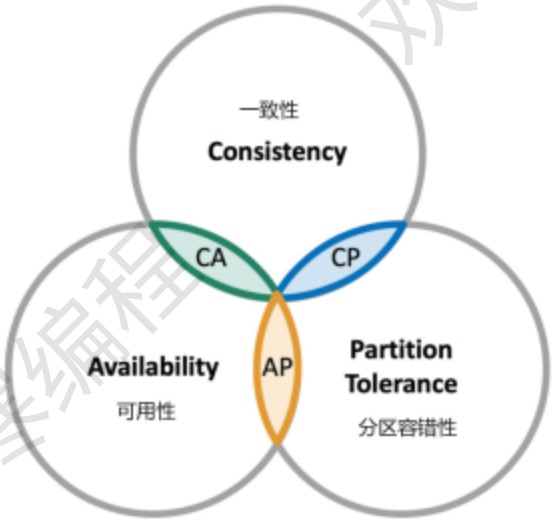
CAP分别指**一致性**(Consistency)、**可用性**(Availability)、**分区容忍度**(Partition tolerance)。

一致性是强调数据的**正确**，可用性是强调数据**不会丢失**，分区容忍则是强调**持续服务**(即便节点之间网络不通也能持续服务)。

CAP理论表示：一个分布式系统不可能同时满足CAP三个条件。一个分布式系统一定满足P，而C，A之间选哪个需要进行权衡。具体而言：

- 如果选择了C，放弃了A，那么系统需要暂停业务，等待更新完成；
- 如果选择了A，放弃了C，那么系统一份数据可能存在不一致。

欢迎关注迹寒编程



02 ❤什么是强一致性、弱一致性、最终一致性。

强一致性：

- 任何一次读都能读到某个数据的最近一次写的数据。
- 系统中的所有进程，看到的操作顺序，都和**全局时钟下的顺序一致**。

不满足强一致性的都是**弱一致性**。

最终一致性：不保证在任意时刻，所有节点的同一份数据是相同的，但在一定的时间后，所有节点的数据都趋于相同。

最终一致性又分为

- **因果一致性**（Casual Consistency）。如果进程A通知进程B它已更新了一个数据项，那么进程B的后续访问将返回更新后的值，且一次写入将保证取代前一次写入。与进程A无因果关系的进程C的访问，遵守一般的最终一致性规则。
- **读写一致性**（Read Your Writes）。或者文艺一点，读己之所写。举个例子，你在知乎上回复一个问题，然后你觉得写的不够装x，再修改了一下，这个时候你再提交。虽然你提交的内容没有到官方的服务器，但你在本地刷新之后看到永远是最新的内容。
- **单调读一致性**。如果进程已经看到过数据对象的某个值，那么任何后续访问都不会返回在那个值之前的值。
- **单调写一致性**。系统保证来自同一个进程的写操作顺序执行。要是系统不能保证这种程度的一

欢迎关注迹寒编程

致性，就非常难以编程了。

参考资料：强一致性、顺序一致性、弱一致性和共识

03 ❤什么是Quorum机制。

Quorum意思是仲裁人数，也就是最小合法人数。比如你是一个选民，选择下一任总统，要求投票的人数必须大于500，这个500就是quorum。quorum机制是一种C和A之间的权衡机制。

考虑N个副本，我们设**最少**要写W个副本，**最多**要读R个副本才能读到更新的数据。那么W+R>N，即读写副本有重叠情况，一般W+R=N+1。例如N=5，W=3，R=3，我们任意更新三个副本，再最多读3个副本就一定能读到更新的数据。

Quorum机制无法保证强一致性，所以是一种弱一致性算法。通常基于Quorum来选择primary（主副本，用于读写），中心节点（服务器）读取R个副本，选择版本号最高的副本作为新的primary，但新选出primary不能马上服务，至少要与W个副本同步完成后才能进行服务。

Quorum机制

04 拜占庭将军问题的含义。

拜占庭位于如今的土耳其的伊斯坦布尔，是东罗马帝国的首都。由于当时拜占庭罗马帝国国土辽阔，为了达到防御目的，每个军队都分隔很远，将军与将军之间只能靠信差传消息。在战争的时候，拜占庭军队内所有将军和副官必须达成一致的共识，决定是否有赢的机会才去攻打敌人的阵营。但是，在军队内有可能存有叛徒和敌军的间谍，左右将军们的决定又扰乱整体军队的秩序。在

进行共识时，结果并不代表大多数人的意见。这时候，在已知有成员谋反的情况下，其余忠诚的将军在不受叛徒的影响下如何达成一致的协议，拜占庭问题就此形成。

拜占庭问题的含义在于：**存在消息丢失的信道上通过通信来达到一致性是不可能的。**

05 ❤2PC和3PC算法知道吗？有什么不同。

在两阶段提交中，主要涉及到两个角色，分别是协调者和参与者。

- 第一阶段：

事务发起者首先向协调者发起请求，然后协调者会给所有参与者发 prepare 请求，进行**预提交**，并返回是否可 以提交。

- 第二阶段： 如果这个时候所有参数者都返回可以提交的消息，这个时候协调者会给参与者发送 commit 请求，当参与者收到后提交完毕会给协调者发送提交成功的响应。

两阶段提交可以保证数据的**强一致性**，但也存在诸多问题：

- **单点故障**，协调者一挂整个服务都会不可用
- **同步阻塞协议**，参与者对事物处理后不提交，如果协调者不可用，那么资源就无法被释放
- **效率低**

3PC的引入是为了解决 2PC 的同步阻塞和减少数据不一致的情况，3PC比2PC多了一个**询问**阶段，也就是**询问准备、预提交、提交**这三个阶段。

虽然解决了同步阻塞问题，但当协调者挂了，参与者还是无法得知整体情况。

且多引入一次通信，还会多增加一次通讯的开销。在实际应用中基本只会出现2PC而几乎没有3PC。

06 ❤什么是幂等性？如何实现？

幂等性：任意多次执行所产生的影响均与一次执行的影响相同。

举个最简单的例子，那就是支付，用户购买商品后支付，支付扣款成功，但是返回结果的时候网络异常，此时钱已经扣了，用户再次点击按钮，此时会进行第二次扣款，返回结果成功，用户查询余额发现多扣钱了，流水记录也变成了两条，造成用户的损失，如何避免这种情况呢?

一种方法是**为每个交易设置唯一的ID**，一旦交易完成，更新订单状态，然后保存流水；如果系统收到相同的请求，查询订单状态，发现已经支付，直接返回，否则进行支付，再返回结果。

一致性协议和算法

主要是paxos算法和raft算法。

01 介绍一下Paxos协议。

Paxos协议是基于**消息传递**且具有**高度容错性**的分布式一致性算法。Paxos算法运行在允许宕机故障的异步系统中，不要求可靠的消息传递。它利用大多数机制保证了2F+1的容错能力，即2F+1个节点的系统允许F个节点同时出现故障。

有三类角色：**提案者**（proposer）、**接收者**（acceptor）和**学习者**（learner）。

第一阶段：

proposer选择一个提案编号n，向半数以上的acceptor广播prepare(n)请求；

被广播的acceptor需做出回应：

- 若 n 大于该 Acceptor 已经响应过提案的**最大编号**，那么它作为响应返回给 Proposer ，并且将不再被允许接受任何小于 n 的提案。
- 如果该 Acceptor **已经批准**过提案，那么就向 Proposer 反馈当前已经批准过提案编号中最大但小于 n 的那个值

第二阶段：

- 若 n 大于该 Acceptor 已经响应过提案的**最大编号**，那么它作为响应返回给 Proposer ，并且将不再被允许接受任何小于 n 的提案。
- 如果该 Acceptor **已经批准**过提案，那么就向 Proposer 反馈当前已经批准过提案编号中最大但小于 n 的那个值

learner如何获取提案：

- **方案一** 当 Acceptor 批准了一个提案时，将将该提案发送给其所有的 Learner，该方案非常简单直接，但通信的次数最大会达到 M * N。
- **方案二** 选定一个主 Learner，在 Acceptor 批准了一个提案时，就将该提案发送给主 Learner，主 Learner 再转发给其他Learner。 虽然该方案较方案一通信次数大大减少，但很可能会出现单点故障问题。

- **方案三** 将主 Learner 的范围扩大，即 Acceptor 可以将批准的提案发送给一个特定的 Learner 集合，然后集合中的 Learner 再转发通知其他 Learner。 该方案可以提高可靠性，但也会增加其网络通信的复杂度。

参考资料：理解 Paxos 协议——浅谈分布式一致性协议，祥光：Paxos算法详解

分布式配置，命名注册Zookeeper采用的是Paxos协议。

02 ❤介绍一下Raft协议。

Raft协议是一种分布式一致性协议，相对Paxos协议，他更好理解。

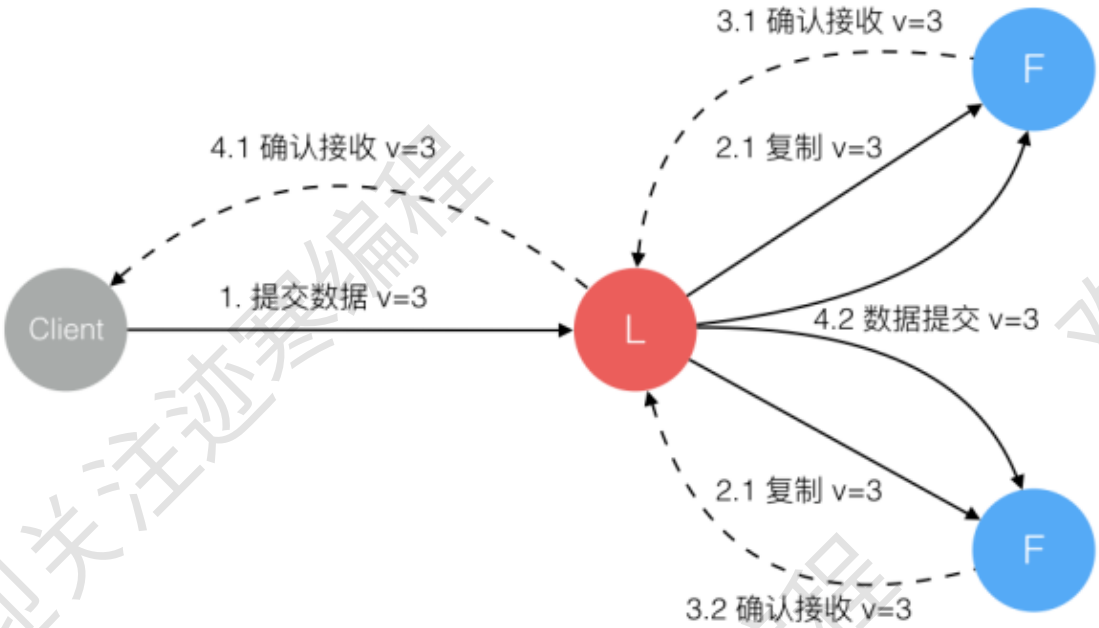
一个节点有三种状态，**leader**、**candidate**和**follower**。

一开始，所有节点都是follower状态，当集群不存在leader时候，follower便化身为candidate，向其它节点发起投票，如果某个candidate获取的票数超过总票数的一半，则当选为leader。

日志复制：

leader负责传达任何改变，每一次变更都会作为一个entry加入到日志中，此时entry状态为uncommitted，只有把entry复制到所有follower节点，然后leader等待大部分节点都复制完成后，leader才会提交entry，节点值发生变更，通知所有follower "entry is committed"。

欢迎关注迹寒编程



raft协议更新流程

选举流程：

时间被分为很多连续的随机长度的term（任期），term有唯一的id。每个term一开始就进行选主：

1. Follower将自己维护的当前任期号加1。
2. 然后将自己的状态转成Candidate
3. 发送RequestVoteRPC消息(带上当前任期号) 给 其它所有节点

这个过程有三种结果：

- 自己被选成了主。当收到了大多数的投票后，状态切成Leader，并且定期给其它的所有server发心跳消息以告诉对方自己是当前任期leader。当一个server收到的任期号比本地的任期号时，就更新本地任期号为最新任期号，并且如果当前state为leader或者candidate时，将自己的状态切成follower。如果任期号比本地任期号更小，则拒绝这个RPC消息。
- 别人成为了主。如果收到了大于或者等于本地任期号的声明，则将自己的状态变为follower，并且更新本地任期号。
- 没有选出leader。这种情况下每个candidate等待的投票的过程就超时了，接着candidates都

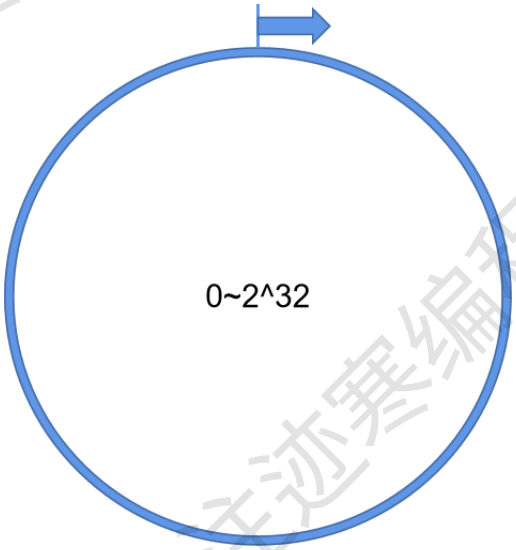
欢迎关注迹寒编程

会将本地任期号再加1，发起新一轮的选举。

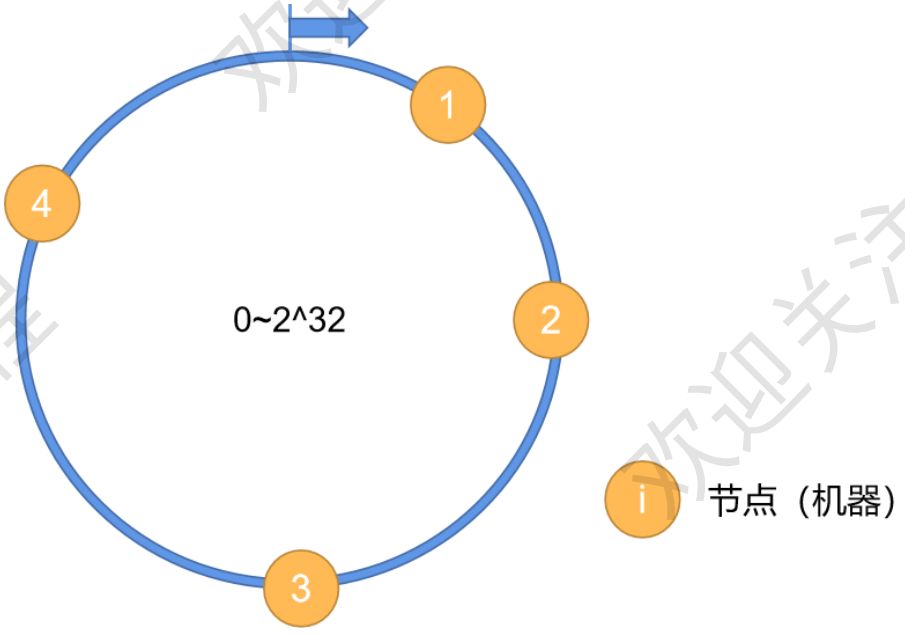
参考资料：Young Wang：全面理解Raft协议，丁凯：Raft协议详解

03 ❤一致性哈希算法了解吗?

首先，我们选择一个足够大的哈希空间，通常(0~2^32)，构成一个哈希环。

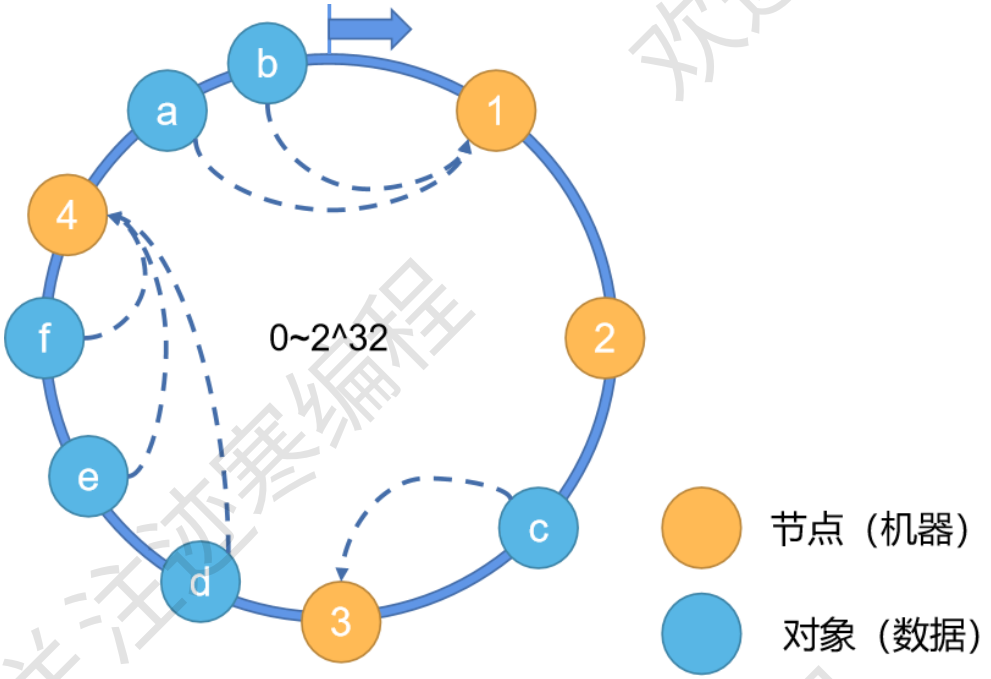


对于缓存集群内的每个存储服务器节点计算 Hash 值，可以用服务器的 IP 或 主机名计算得到哈希值，计算得到的哈希值就是服务节点在 Hash 环上的位置。

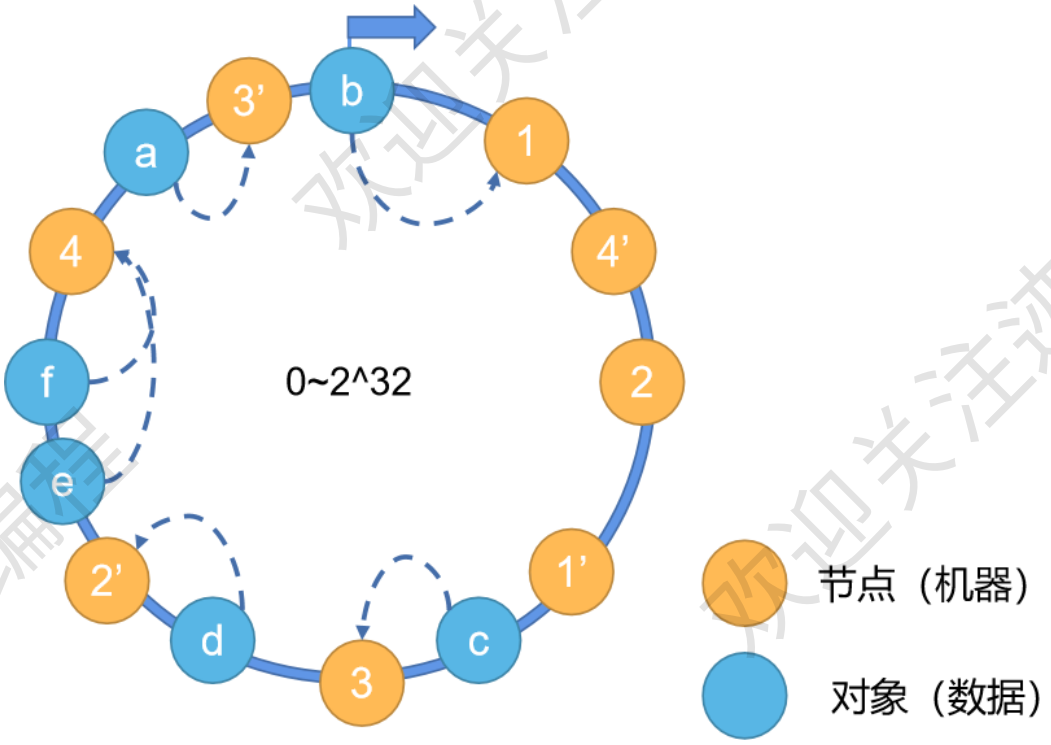


之后把对象也计算哈希值映射到环上，按照**顺时针找到的第一个节点，**存储该对象。

欢迎关注迹寒编程



这样设计的好处是，当向集群增加或删除节点时，只需要迁移节点本身的数据到下个顺时针节点即可，原本的哈希映射没有被破坏。但是这样又有一个新的问题“**数据倾斜**”，以上图为例，节点4存储了三个对象，而节点2没有对象，这样就会导致负载不均衡。解决方法是引入“**虚拟节点**”，将现有的物理节点通过虚拟的方法复制多个，需要注意的是类似于多副本，一旦节点映射到环上，就被统一视作虚拟节点，地位都相同。



欢迎关注迹寒编程

如何判断查找的对象在哪个机器上?

如果有全局的节点哈希值，则可先把节点哈希值排序，再计算对象哈希值，利用二分法找到大于对象哈希值的最小的那个节点。

如果没有全局节点哈希，可用采用chord算法，每个节点只保存最多m项的路由表，通常是其后继节点的信息。

当在某个节点上查找资源时，首先判断**其后继节点**有没有持有该资源，若没有则直接从该节点的路由表**从最远处开始查找**。直到找到第一个hash(node)小于hash(data)的节点，然后跳转到此节点上，进行新一轮的查找。当hash(data)落在此节点和其后继节点之间时，则说明资源存储在当前节点的后继节点上。

参考资料：知一致性哈希算法（consistent hashing）

分布式系统技术

分布式系统的相关技术。

01 ❤集群“脑裂”怎么产生，如何解决。

两个机房之间的网络通信出现故障时，选举机制就有可能在不同的网络分区中选出两个Leader。当网络恢复时，这两个Leader都想会存储数据，导致发生数据不一致。这也就出现了“脑裂”现象。

解决脑裂，主要有两种方式：

- **仲裁quorum机制**

设置一个参考IP，当心跳完全断开的时候，节点ping一下参考IP，如果ping不通则表明当前节点网络发生阻塞或者故障，自行退出对资源的争夺。

- **增加心跳线**

比如使用多根以太网线来进行通信，增加可用性。

- **启用磁盘锁/IO Fence**

正在服务一方锁住共享磁盘，脑裂发生的时候，让对方完全抢不走共享的磁盘资源。

以上方式可以同时使用，可以减少集群中脑裂情况的发生，但不能完全保证，比如仲裁机制中2台机器同时宕机，那么此时集群中没有Leader 可以使用。此时就需要人工干预了。

欢迎关注迹寒编程

参考资料：集群基本概念–脑裂的产生和解决方案

02 什么是IO Fence技术?

I/O Fencing使用SCSI–3 persistent reservations (PR)技术来实现共享存储保护。它需要一个或多个**协作磁盘**（Coordinator Disks），每个节点都要注册自己节点ID到协作磁盘，这个注册Key并没有写到磁盘上，而是在磁盘的芯片或是RAID控制器中，每个注册节点都能看到所有节点注册的Key，共同构成一个集群环境。

接下来群集开始启动服务，加载资源组，加载磁盘后，节点会向数据磁盘写入注册信息，节点A向它所控制的磁盘写入Key A，节点B向它所控制的磁盘写入Key B。此后只有注册的系统可以向数据磁盘写入数据。各节点排它使用数据磁盘。

如果节点B失效（心跳线断开），则节点A清除节点B在协作磁盘上注册的ID，集群导入原来由节点B控制的数据磁盘，清除节点B注册信息，写入节点A的注册信息。实现资源转移。

参考资料：I/O Fencing技术简介

03 关于分布式系统中间件了解多少?

中间件是介于操作系统软件和用户应用软件之间的一种独立的基本系统软件或服务程序。它采用分布式架构，以满足系统高并发，高吞吐，低延迟等需求。例如：

- redis：实现缓存；
- RocketMQ：实现分布式消息队列；
- Zookeeper：分布式锁；
- Elasticsearch：全文搜索；
- Nginx：实现分布式负载均衡；

04 ❤什么是分布式锁？如何实现之?

分布式锁的目的是**并发控制**。

分布式锁的特性是：

1. 互斥性：在任意时刻，只有一个客户端能持有锁；
2. 不会发生死锁：即使有一个客户端在持有锁期间崩溃而没有主动解锁，也能保证其它客户端能加锁；
3. 加锁和解锁必须是同一个客户端；
4. 具有容错性，只要大多数redis节点正常运行，客户端就能获取和释放锁。

主要实现有三种途径：

- 采用Redis的setnx；
- 采用数据库乐观锁（唯一性索引）；
- 采用Zookeeper分布式锁；

分类	方案	实现原理	优点	缺点
基于数据库	基于mysql 表唯一索引	1.表增加唯一索引 2.加锁：执行insert语句，若报错，则表明加锁失败 3.解锁：执行delete语句	完全利用DB现有能力，实现简单	1.锁无超时自动失效机制，有死锁风险 2.不支持锁重入，不支持阻塞等待 3.操作数据库开销大，性能不高
基于分布式协调系统	基于ZooKeeper	1.加锁：在/lock目录下创建临时有序节点，判断创建的节点序号是否最小。若是，则表示获取到锁；否，则watch /lock目录下序号比自身小的前一个节点 2.解锁：删除节点	1.由zk保障系统高可用 2.Curator框架已原生支持系列分布式锁命令，使用简单	需单独维护一套zk集群，维保成本高
基于缓存	基于redis命令	1. 加锁：执行setnx，若成功再执行expire添加过期时间 2. 解锁：执行delete命令	实现简单，相比数据库和分布式系统的实现，该方案最轻，性能最好	1.setnx和expire分2步执行，非原子操作；若setnx执行成功，但expire执行失败，就可能出现死锁 2.delete命令存在

欢迎关注迹寒编程

				2.delete命令存在误删除非当前线程持有的锁的可能 3.不支持阻塞等待、不可重入
	基于redis Lua脚本能力	1. 加锁：执行SET lock_name random_value EX seconds NX 命令 2. 解锁：执行Lua脚本	同上；实现逻辑上也更严谨，除了单点问题，生产环境采用用这种方案，问题也不大。	不支持锁重入，不支持阻塞等待

> 锁重入：指任意线程在获取到锁之后，再次获取该锁而不会被该锁所阻塞。关联一个线程持有者+计数器，重入意味着锁操作的颗粒度为“线程”。

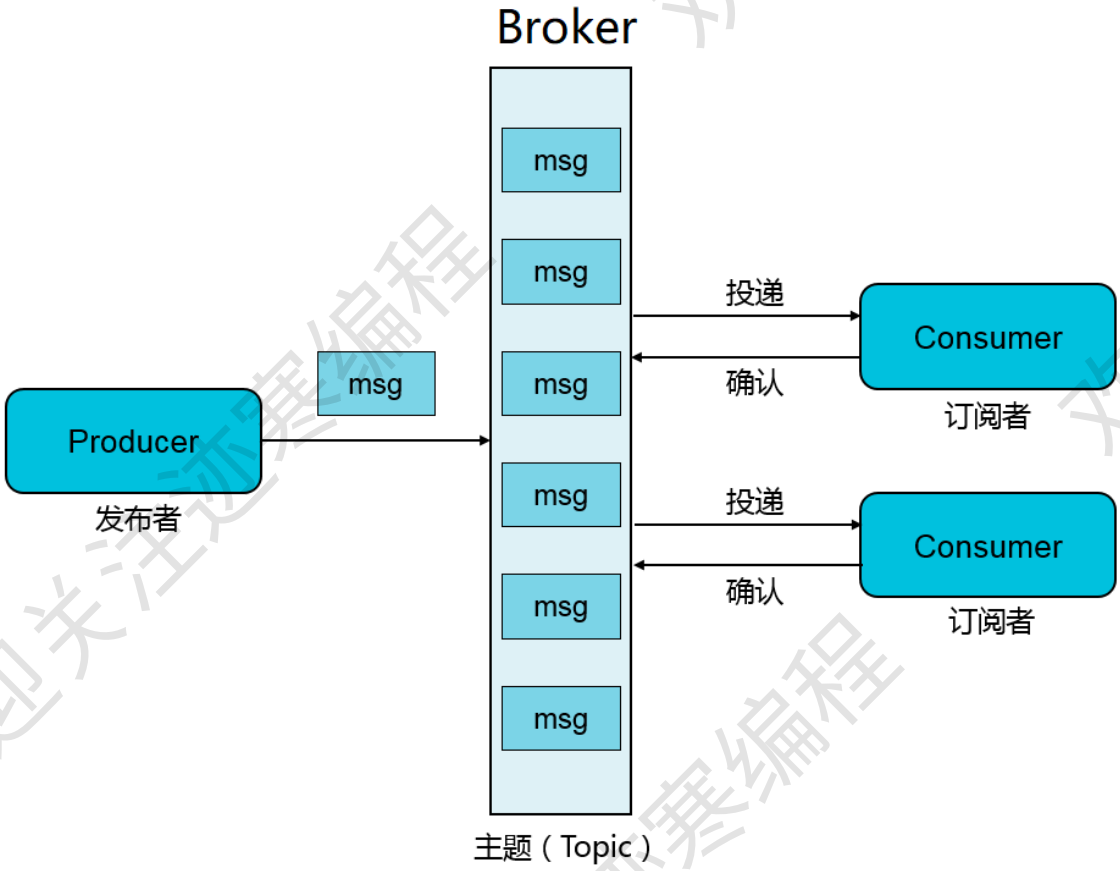
参考资料：分布式锁-这一篇全了解(Redis实现分布式锁完美方案)，
Distributed Locks with Redis

05 关于分布式消息队列了解多少?

消息队列是在消息的传输过程中保存消息的容器，用于接收消息并以文件的方式存储，一个消息队列可以被一个也可以被多个消费者消费，包含以下 3 元素：

- Producer：消息生产者，负责产生和发送消息到 Broker；
- Broker：消息处理中心，负责消息存储、确认、重试等，一般其中会包含多个 Queue；
- Consumer：消息消费者，负责从 Broker 中获取消息，并进行相应处理。

欢迎关注迹寒编程



消息队列的模式：

- 点对点模式：多个生产者可以向同一个消息队列发送消息，一个具体的消息只能由一个消费者消费。
- 发布/订阅模式：单个消息可以被多个订阅者并发的获取和处理。

应用场景：

- **应用解耦**：消息队列减少了服务之间的耦合性，不同的服务可以通过消息队列进行通信，而不用关心彼此的实现细节。
- **异步处理**：消息队列本身是异步的，它允许接收者在消息发送很长时间后再取回消息。
- **流量削锋**：当上下游系统处理能力存在差距的时候，利用消息队列做一个通用的”载体”，在下游有能力处理的时候，再进行分发与处理。
- **日志处理**：日志处理是指将消息队列用在日志处理中，比如 Kafka 的应用，解决大量日志传输的问题。
- **消息通讯**：消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯，比如实现点对点消息队列，或者聊天室等。
- **消息广播**：如果没有消息队列，每当一个新的业务方接入，我们都要接入一次新接口。有了消息队

欢迎关注迹寒编程

列，我们只需要关心消息是否送达了队列，至于谁希望订阅，是下游的事情，无疑极大地减少了开发和联调的工作量。

常用的消息队列有：RabbitMQ、Kafka、RocketMQ等。

06 关于etcd了解多少?

tcd 是云原生架构中重要的基础组件，由 CNCF 孵化托管。etcd 在微服务和 Kubernetes 集群中不仅可以作为**服务注册与发现**，还可以作为 key-value 存储的中间件。采用raft协议确保一致性。

07 关于Docker了解多少?

Docker 是一个开源的应用容器引擎，基于 Go 语言 并遵从 Apache2.0 协议开源。

Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）,更重要的是容器性能开销极低。

Docker 包括三个基本概念:

- 镜像（Image）：Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:16.04 就包含了完整的一套 Ubuntu16.04 最小系统的 root 文件系统。
- 容器（Container）：镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。
- 仓库（Repository）：仓库可看成一个代码控制中心，用来保存镜像。

08 关于Kubernetes了解多少?

Kubernetes 是一个生产级别的开源平台，可协调在计算机集群内和跨计算机集群的应用容器的部署（调度）和执行.

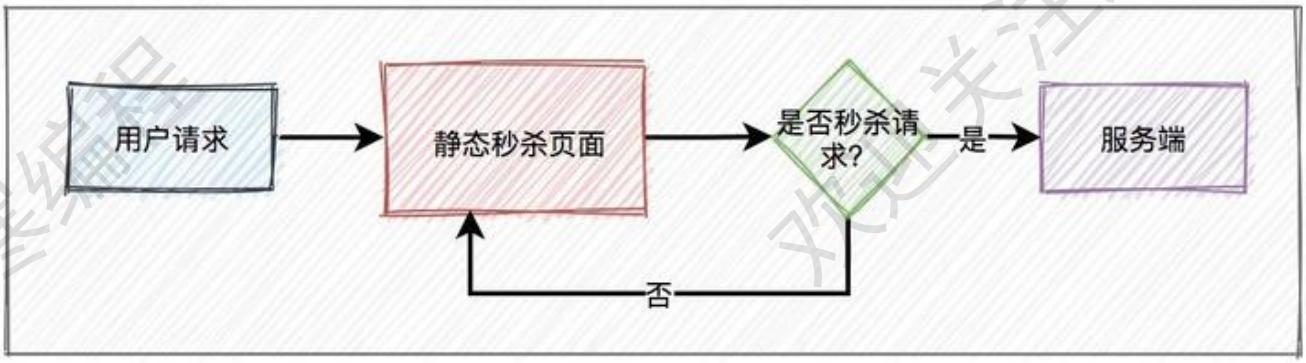
- **Master（主节点）：** 控制 Kubernetes 节点的机器，也是创建作业任务的地方。
- **Node（节点）：** 这些机器在 Kubernetes 主节点的控制下执行被分配的任务。
- **Pod：** 由一个或多个容器构成的集合，作为一个整体被部署到一个单一节点。同一个 pod 中的容器共享 IP 地址、进程间通讯（IPC）、主机名以及其它资源。Pod 将底层容器的网络和存储抽象出来，使得集群内的容器迁移更为便捷。
- **Replication controller（复制控制器）：** 控制一个 pod 在集群上运行的实例数量。
- **Service（服务）：** 将服务内容与具体的 pod 分离。Kubernetes 服务代理负责自动将服务请求分发到正确的 pod 处，不管 pod 移动到集群中的什么位置，甚至可以被替换掉。
- **Kubelet：** 这个守护进程运行在各个工作节点上，负责获取容器列表，保证被声明的容器已经启动并且正常运行。
- **kubectl：** 这是 Kubernetes 的命令行配置工具。

09 ❤如何设计一套整点秒杀系统？讲一讲思路。

> 秒杀一般出现在商城的促销活动中，指定了一定数量（比如：10个）的商品，以极低的价格（比如：9.9元），让大量用户参与活动，但只有极少数用户能够购买成功。这类活动商家绝大部分是不赚钱的，说白了是找个噱头宣传自己。

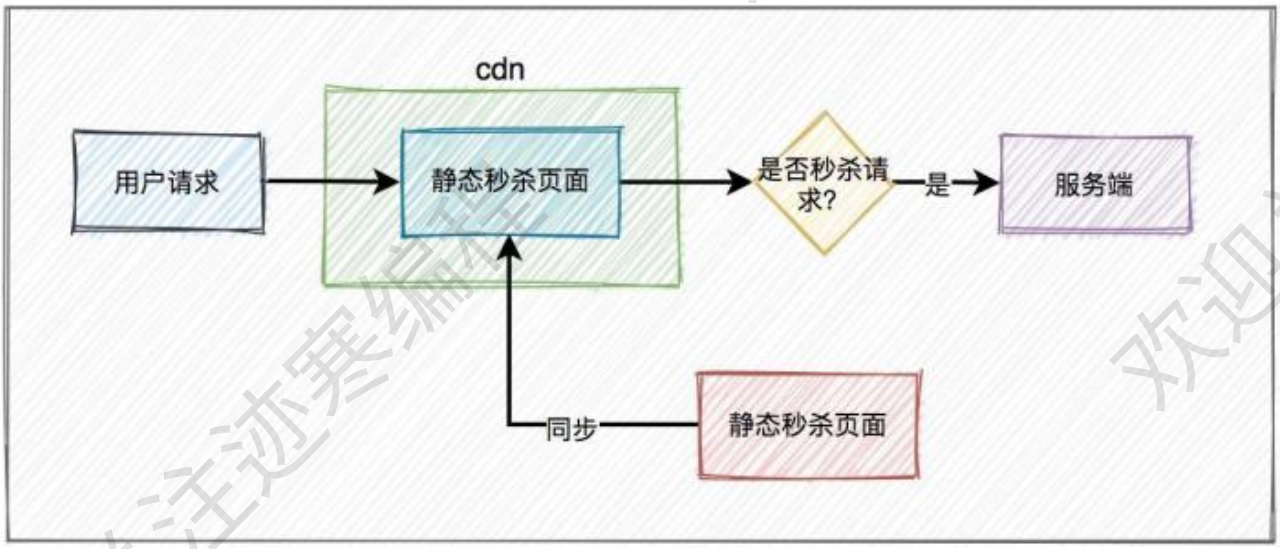
一般在秒杀点（比如12点）前几分钟，用户量激增，到达时间点时，并发量到达顶峰。正常情况下，大部分用户收到商品已抢完的提醒，就会退出当前页面，导致并发量急剧下降，我们称这种现象为**瞬时高并发**。传统的系统是应付不了这么高并发的，所以要进行一系列优化：

1. **界面静态化**：也就是对**活动页面**作静态化处理，用户浏览界面并不会发送到服务器，只有到达秒杀点，且用户点击了秒杀按钮，才允许访问服务端；

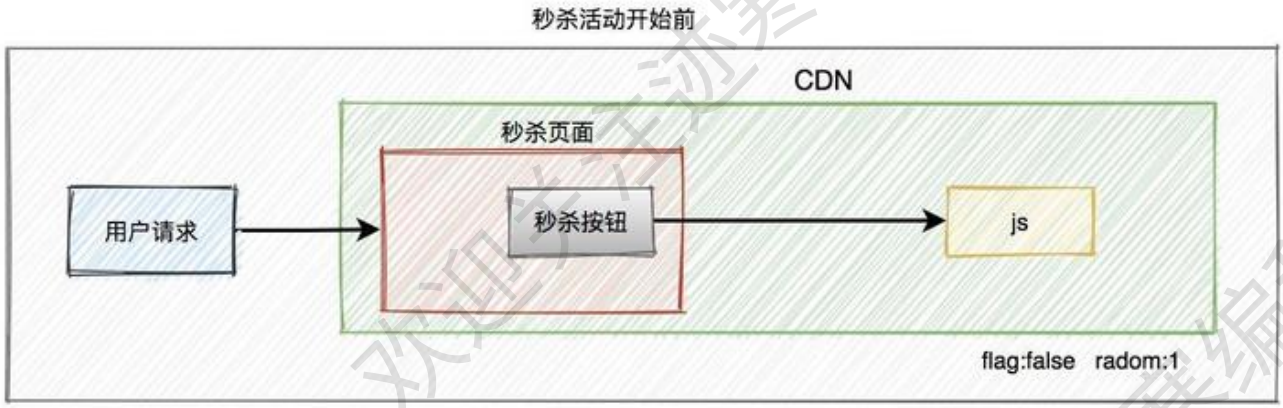


2. **CDN加速**：为了让用户以最快速度访问活动页面，需要内容分发网络(CDN)的帮助，用户从最

近的CDN获取内容，降低网络阻塞。

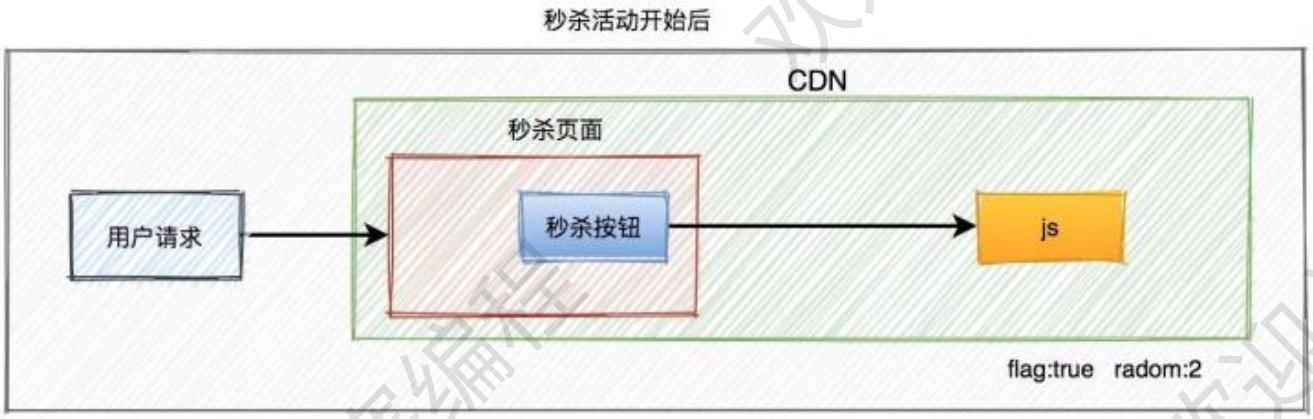


3. **秒杀按钮**：提前进入活动页面，一般按钮是灰的，不可点击。为了在静态页面控制秒杀按钮，就必须采用js文件控制，秒杀开始之前，js标志为false，还有另外一个随机参数。



当秒杀开始的时候系统会生成一个新的js文件，此时标志为true，并且随机参数生成一个新值，然后同步给CDN。由于有了这个随机参数，CDN不会缓存数据，每次都能从CDN中获取最新的js代码。

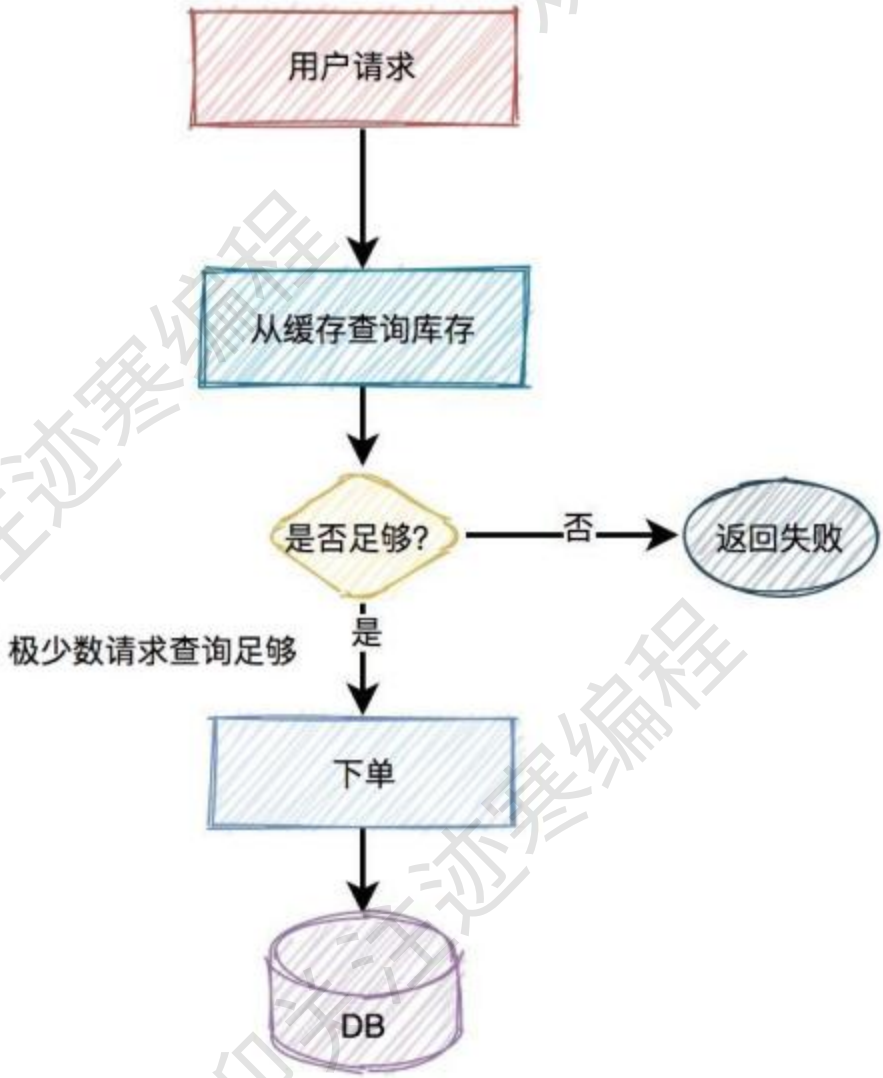
欢迎关注迹寒编程



此外，前端还可以加一个定时器，控制比如：5秒之内，只允许发起一次请求。如果用户点击了一次秒杀按钮，则在5秒之内置灰，不允许再次点击，等到过了时间限制，又允许重新点击该按钮。

4. **读多写少**。由于只有少部分人能够抢到商品，大部分人仅仅是查询，这是一个典型的“读多写少”场景。应该使用缓存，比如redis。

欢迎关注迹寒编程



在瞬时高并发的场景下，可能缓存也够呛，出现**缓存穿透**和**缓存击穿**两种情况。

- 缓存穿透就是指大量请求访问**缓存和数据库都不存在的key**，导致要绕过缓存访问数据库，造成数据库压力过大。

通常可以采用**布隆过滤器**，如果返回false，表明key一定不存在可以直接返回。

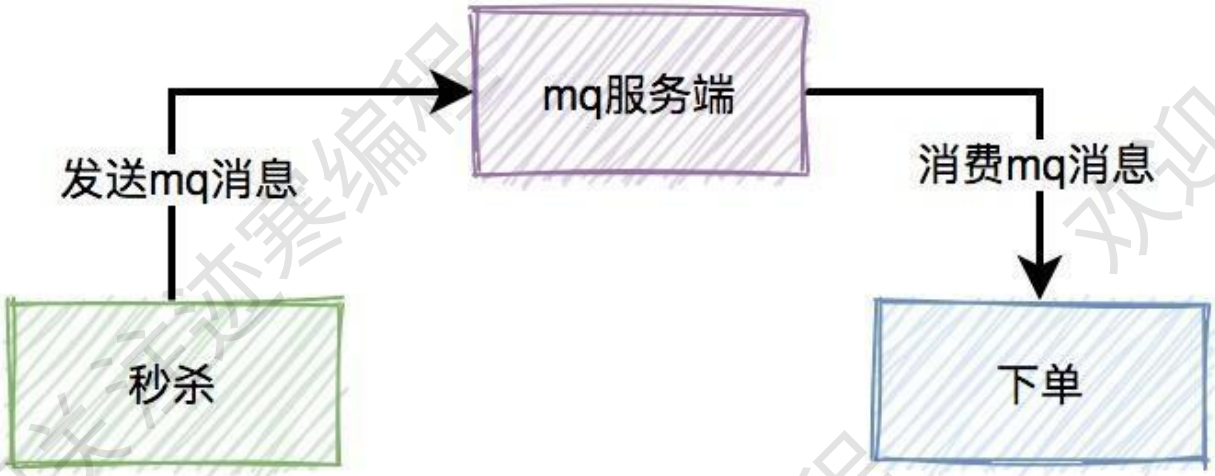
- 缓存击穿就是key存在于数据库，但是不存在于缓存的情况，导致大量请求绕过缓存访问数据库，压力暴增。

这种情况就要采用**分布式锁，**请求访问数据库前先必须获得锁。当然，针对这种情况，最好在项目启动之前，先把缓存进行预热。即事先把所有的商品，同步到缓存中，这样商品基本都能直接从缓存中获取到，就不会出现缓存击穿的问题了。

5. MQ异步处理

欢迎关注迹寒编程

真实业务场景：秒杀→下单→支付，只有秒杀的并发量最高，下单和支付的并发量其实很少。所以，我们在设计秒杀系统时，有必要把下单和支付功能从秒杀的主流程中拆分出来，特别是下单功能要做成mq异步处理的。而支付功能，比如支付宝支付，是业务场景本身保证的异步。



6. 限流

为了防止某个用户，请求接口的次数过于频繁，可以只针对该用户进行限制。

- 限制同一用户id，比如5分钟只能请求5次接口；
- 对同一ip进行限流；
- 对接口进行限流；
- 采用验证码，普通验证码，由于生成的数字或者图案比较简单，可能会被破解。优点是生成速度比较快，缺点是有安全隐患。还有一个验证码叫做：移动滑块，它生成速度比较慢，但比较安全，是目前各大互联网公司的首选。

参考资料：面试必考：秒杀系统如何设计？ - 腾讯云开发者社区-腾讯云

总结

分布式系统涵盖非常多的知识，是属于计算机体系里面的“上层建筑”。许多著名的项目都是基于分布式的。在生产环境中，如果你负责开发一个模块，就必然要进行产品选型，用什么技术，是否做二次开发，只有大量的实践才能确定最优的选择。然而这注定是非常艰苦耗时的历程。

最后分享一些课程，希望对你有帮助！

欢迎关注迹寒编程

国外分布式系统课程

6.824: Distributed Systems

–

CS244b: Distributed systems

https://www.scs.stanford.edu/14au-cs244b/

Distributed Systems

An introductory course on the techniques for creating functional, usable, and scalable distributed s...

https://www.cs.cmu.edu/~15-440/

欢迎关注迹寒编程