> 商汤

1.简历写什么问什么，注意所用技术产品的同类产品生态及对比。

2.kafka数据分区和消费者的关系，kafka的数据offset读取流程，kafka内部如何保证顺序，结合外部组件如何保证消费者的顺序

1、kafka数据分区和消费者的关系：1个partition只能被同组的一个consumer消费，同组的consumer则起到均衡效果

2、kafka的数据offset读取流程

1.连接ZK集群，从ZK中拿到对应topic的partition信息和partition的Leader的相关信息

2.连接到对应Leader对应的broker

3.consumer将自己保存的offset发送给Leader

4.Leader根据offset等信息定位到segment（索引文件和日志文件）

5.根据索引文件中的内容，定位到日志文件中该偏移量对应的开始位置读取相应长度的数据并返回给consumer

3、kafka内部如何保证顺序：

kafka只能保证partition内是有序的，但是partition间的有序是没办法的。爱奇艺的搜索架构，是从业务上把需要有序的打到同一个partition。

4、

3.cms垃圾回收机制

1、概念：CMS全称 Concurrent Mark Sweep，是一款并发的、使用标记-清除算法的垃圾回收器，

2、使用场景：GC过程短暂停，适合对时延要求较高的服务，用户线程不允许长时间的停顿。

3、缺点：

1、服务长时间运行，造成严重的内存碎片化。

2、算法实现比较复杂（如果也算缺点的话）。

4、实现机制：根据GC的触发机制分为：

1、周期性Old GC（被动）：2s执行一次；

2、主动Old GC：触发条件：

i. YGC过程发生Promotion Failed，进而对老年代进行回收

ii. 比如执行了System.gc()，前提是没有参数ExplicitGCInvokesConcurrent

iii. 其它情况...

4.springcloud各个组件功能，内部细节，与dubbo区别，dubbo架构，dubbo负载策略

1、springcloud各个组件功能：

a. Ribbon，客户端负载均衡，特性有区域亲和、重试机制。

b. Hystrix，客户端容错保护，特性有服务降级、服务熔断、请求缓存、请求合并、依赖隔离。

c. Feign，声明式服务调用，本质上就是Ribbon+Hystrix

d. Stream，消息驱动，有Sink、Source、Processor三种通道，特性有订阅发布、消费组、消息分区。

e. Bus，消息总线，配合Config仓库修改的一种Stream实现，

f. Sleuth，分布式服务追踪，需要搞清楚TraceID和SpanID以及抽样，如何与ELK整合。

g. Eureka，服务注册中心，特性有失效剔除、服务保护。

h. Dashboard，Hystrix仪表盘，监控集群模式和单点模式，其中集群模式需要收集器Turbine配合。

i. Zuul，API服务网关，功能有路由分发和过滤。

j. Config，分布式配置中心，支持本地仓库、SVN、Git、Jar包内配置等模式，

2、dubbo负载策略：

Random LoadBalance

- **随机**，按权重设置随机概率。
- 在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

RoundRobin LoadBalance

- **轮询**，按公约后的权重设置轮询比率。
- 存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

LeastActive LoadBalance

- **最少活跃调用数**，相同活跃数的随机，活跃数指调用前后计数差。
- 使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

ConsistentHash LoadBalance

- **一致性 Hash**，相同参数的请求总是发到同一提供者。
- 当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。
- 算法参见：http://en.wikipedia.org/wiki/Consistent_hashing
- 缺省只对第一个参数 Hash，如果要修改，请配置 `<dubbo:parameter key="hash.arguments" value="0,1" />`
- 缺省用 160 份虚拟节点，如果要修改，请配置 `<dubbo:parameter key="hash.nodes" value="320" />`

5.mapreduce原理

1、简介：mapreduce源自google的一篇文章，将海量数据处理的过程拆分为map和reduce。mapreduce 成为了最早的分布式计算框架，这样即使不懂的分布式计算框架的内部运行机制的用户，也可以利用分布式的计算框架实现分布式的计算，并在hadoop上面运行。

2、设计思想：

hadoop 文件系统 ，提供了一个分布式的文件系统，但是hadoop文件系统读写的操作都涉及到大量的网络的操作，并不能很好的完成实时性比较强的任务。

但是hadoop可以给上面的应用提供一个很好的支持。比如hadoop文件系统上面可以运行mapreduce。mapreduce是一个计算的框架，mapreduce是一个分布式的计算框架，这样mapreduce利用分布式的文件系统，将不同的机器上完成不同的计算，然后就计算结果返回。这样很好的利用了分布式的文件系统。

数据分布式的存储，然后计算的时候，分布式的计算，然后将结果返回。这样的好处就是不会涉及到大量的网络传输数据。

3、优点：mapreduce的计算框架的优点是，极强的扩展能力，可以在数千台机器上并发的执行。其次，有很好的容错性，另外，就是向上的接口简洁。用户只需要写map和reduce函数，即可完成大规模数据的并行处理。

4、缺点：mapreduce并不适合对实时性要求比较高的场景，比如交互式查询或者是流式计算。另外，也不适合迭代类的计算（比如机器学习类的应用）。

> 1、mapreduce的启动时间比较长，对于批处理的任务，这个问题并不算大。但是对于实时性比较高的任务，其启动时间长的缺点就很不合适了。
> 2、mapreduce一次执行的过程里面，往往涉及到多出磁盘读写，以及网络的传输。对于迭代的任务，这样很好的开销需要很多次，明显降低了效率。
> 3、而Storm和Spark，一个是流式计算的框架，一个是机器学习的框架。他们更适合解决这类型的任务。

6.nio,bio,sellector/epoll,aio,netty自带编解码器，netty优势，java内存模型

Netty高性能：

1、NIO异步非阻塞通信

2、“零拷贝”

3、内存池ByteBuf

4、Netty提供了多种内存管理策略，通过在启动辅助类中配置相关参数，可以实现差异化的定制。

5、高效的Reactor线程模型：Reactor单线程（多线程、主从）模型，指的是所有的IO操作都在同一个NIO线程上面完成

6、为了尽可能提升性能，Netty采用了串行无锁化设计，在IO线程内部进行串行操作，避免多线程竞争导致的性能下降。表面上看，串行化设计似乎CPU利用率不高，并发程度不够。但是，通过调整NIO线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。

7、高效的并发编程：Netty的高效并发编程主要体现在如下几点：

1）volatile的大量、正确使用；

2）CAS和原子类的广泛使用；

3）线程安全容器的使用；

4）通过读写锁提升并发性能。

8、高效的序列化框架：

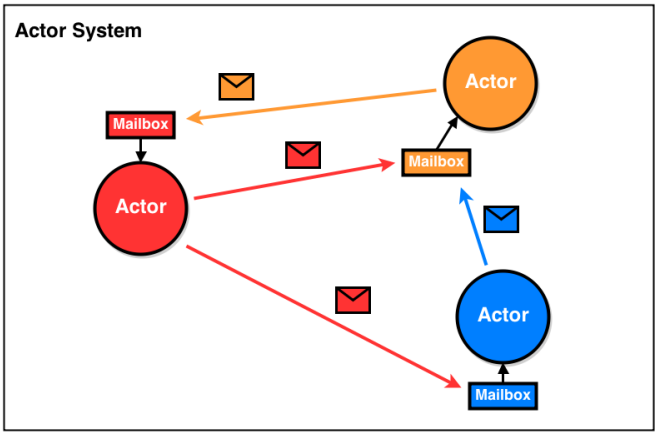
9、灵活的TCP参数配置能力：合理设置TCP参数在某些场景下对于性能的提升可以起到显著的效果，例如SO_RCVBUF和SO_SNDBUF。如果

设置不当，对性能的影响是非常大的。

7.akka模型

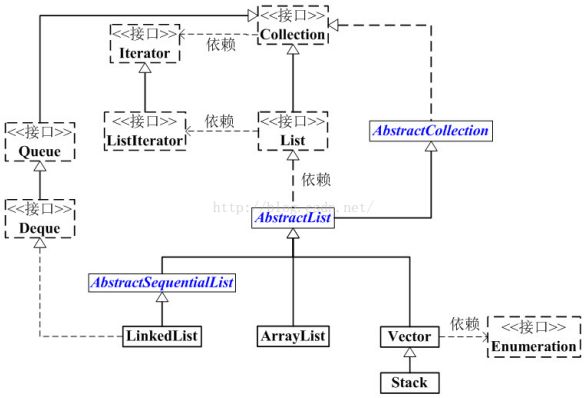
1、概念：Akka是一个构建在JVM上，基于Actor模型的的并发框架，为构建伸缩性强，有弹性的响应式并发应用提高更好的平台。

2、Actor模型：Akka的核心就是Actor，所以不得不说Actor，Actor模型我通俗的举个例子，假定现实中的两个人，他们只知道对方的地址，他们想要交流，给对方传递信息，但是又没有手机，电话，网络之类的其他途径，所以他们之间只能用信件传递消息，很像现实中的的邮政系统，你要寄一封信，只需根据地址把信投寄到相应的信箱中，具体它是如何帮你处理送达的，你就不需要了解了，你也有可能收到收信人的回复，这相当于消息反馈。上述例子中的信件就相当于Actor中的消息，Actor与Actor之间只能通过消息通信。



8.java arraylist,linkedlist区分及实现原理，hashmap和concurrenthashmap区分及实现原理，concurrenthashmap 1.7和1.8区分，实现细节，linkedhashmap排序原理，应用如何保证数据幂等

1、java arraylist,linkedlist区分及实现原理：



1. ArrayList是实现了基于动态数组的数据结构，而LinkedList是基于链表的数据结构；
2. 对于随机访问get和set，ArrayList要优于LinkedList，因为LinkedList要移动指针；
3. 对于添加和删除操作add和remove，一般大家都会说LinkedList要比ArrayList快，因为ArrayList要移动数据。

2、concurrenthashmap 1.7和1.8区分：

- 去除 Segment + HashEntry + Unsafe的实现，
  改为 Synchronized + CAS + Node + Unsafe的实现
  其实 Node 和 HashEntry 的内容一样，但是HashEntry是一个内部类。
  用 Synchronized + CAS 代替 Segment ，这样锁的粒度更小了，并且不是每次都要加锁了，CAS尝试失败了在加锁。
- put()方法中 初始化数组大小时，1.8不用加锁，因为用了个 sizeCtl变量，将这个变量置为-1，就表明table正在初始化。

3、linkedhashmap排序原理：

```
    public LinkedHashMap() {
        // 调用HashMap的构造方法，其实就是初始化Entry[] table
        super();
        // 这里是指是否基于访问排序，默认为false
        accessOrder = false;
    }
```

1、LinkedHashMap存储数据是有序的，而且分为两种：插入顺序和访问顺序。默认为插入顺序。

2、LinkedHashMap有自己的静态内部类Entry，它继承了HashMap.Entry，定义如下：

```
/**
 * LinkedHashMap entry.
 */
private static class Entry<K,V> extends HashMap.Entry<K,V> {
    // These fields comprise the doubly linked list used for iteration.
    Entry<K,V> before, after;

    Entry(int hash, K key, V value, HashMap.Entry<K,V> next) {
        super(hash, key, value, next);
    }
}
```

3、所以LinkedHashMap构造函数，主要就是调用HashMap构造函数初始化了一个Entry[] table，然后调用自身的init初始化了一个只有头结点的双向链表。

9.web.xml listener，filter，servlet加载顺序。如何不再web，xml中配置来加载filter

1、web.xml listener，filter，servlet加载顺序：context-param -> listener -> filter -> servlet

2、

10.无穷数就top K问题，提供多个方案

1、最简单且最容易想到的算法是对数组进行排序（快速排序），然后取最大或最小的K个元素。总的时间复杂度为O(N*logN)+O(K)=O(N*logN)。该算法存在以下问题：

1. 快速排序的平均复杂度为O(N*logN)，但最坏时间复杂度为O(n2)，不能始终保证较好的复杂度
2. 只需要前k大或k小的数,，实际对其余不需要的数也进行了排序，浪费了大量排序时间

总结：通常不会采取该方案。

2、虽然我们不会采用快速排序的算法来实现TOP-K问题，但我们可以利用快速排序的思想，在数组中随机找一个元素key，将数组分成两部分Sa和Sb，其中Sa的元素>=key，Sb的元素<key，然后分析两种情况：

1. 若Sa中元素的个数大于或等于k，则在Sa中查找最大的k个数
2. 若Sa中元素的个数小于k，其个数为len，则在Sb中查找k-len个数字

如此递归下去，不断把问题分解为更小的问题，直到求出结果。

3、寻找N个数中的第K大的数，可以将问题转化寻找N个数中第K大的问题。对于一个给定的数p， 可以在O(N)的时间复杂度内找出所有不小于P的数。

根据分析，可以使用二分查找的算法思想来寻找N个数中第K大的数。假设N个数中最大的数为Vmax，最小的数为Vmin, 那么N个数中第K大的数一定在区间[Vmin,Vmax]之间。然后在这个区间使用二分查找算法。

总结：该算法实际应用效果不佳，尤其是不同的数据类型需要确定max - min > delta，因此时间复杂度跟数据分布有关。 整个算法的时间复杂度为O(N * log（Vmax-Vmin）/delta)，在数据分布平均的情况下，时间复杂度为O(N * logN)。

4、上面几种解法都会对数据访问多次，那么就有一个问题，当数组中元素个数非常大时，如：100亿，这时候数据不能全部加载到内存，就要求我们尽可能少的遍历所有数据。针对这种情况，下面我们介绍一种针对海量数据的解决方案。

在学习堆排序的过程中，我们知道了堆这种数据结构。为了查找Top k大的数，我们可以使用大根堆来存储最大的K个元素。大根堆的堆顶元素就是最大K个数中最小的一个。每次考虑下一个数x时，如果x比堆顶元素小，则不需要改变原来的堆。如果想x比堆顶元素大，那么用x替换堆顶元素， 同时，在替换之后，x可能破坏最小堆的结构，需要调整堆来维持堆的性质。

总结：该算法只需要扫描所有的数据一次，且不会占用太多内存空间（只需要容纳K个元素的空间），尤其适合处理海量数据的场景。算法的时间复杂度为O(N * logk)，这实际上相当于执行了部分堆排序。

5、TOP-K问题是一个经典的问题，这个问题是存在线性算法的，只不过算法的使用范围有一定的限制。如果所有N个数都是正整数，且他们的取值范围并不大，可以考虑申请空间，记录每个整数出现的次数，然后再从大到小取最大的K个。实际就是利用计数排序的思想。 假设所有整数都在（0，maxN）区间，利用一个数组count[maxN]来记录每个整数出现的次数。count[i]表示整数i在N个数中出现的次数。只需要扫描一遍就可以得到count数组，然后寻找第K大的元素。

这是一个典型的以空间换取时间的做法。当数组中取值范围比较大时，是及其浪费空间的。如[3,1...9999]，为了求出最大的K个元素，需要额外申请一个长度为10000的数组。

极端情况下，如果 N 个整数各不相同，我们甚至只需要一个 bit 来存储这个整数是否存在，这样可节省很大的内存空间。

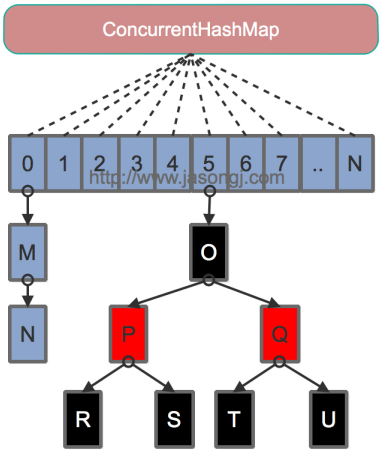
11.a,b,c三张表，做关联查询，如何优化，可做外键，只在c表加a表外键即可。

1. 对于要求全面的结果时，我们需要使用连接操作（LEFT JOIN / RIGHT JOIN / FULL JOIN）；
2. 不要以为使用MySQL的一些连接操作对查询有多么大的改善，核心是索引；
3. 对被驱动表的join字段添加索引；

12.CourrentHashMap JDK1.7和JDK1.8有什么区别?

1. Java 7为实现并行访问，引入了Segment这一结构，实现了分段锁，理论上最大并发度与Segment个数相等。

2. Java 8为进一步提高并发性，摒弃了分段锁的方案，而是直接使用一个大的数组。同时为了提高哈希碰撞下的寻址性能，Java 8在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为O(N)）转换为红黑树（寻址时间复杂度为O(long(N))）。其数据结构如下图所示



13.线程a,b,c,d运行任务，怎么保证当a,b,c线程执行完再执行d线程?

1、CountDownLatch类

一个同步辅助类，常用于某个条件发生后才能执行后续进程。给定计数初始化CountDownLatch，调用countDown()方法，在计数到达零之前，await方法一直受阻塞。

重要方法为countdown()与await()；

2、join方法

将线程B加入到线程A的尾部，当A执行完后B才执行。

```
public static void main(String[] args) throws Exception {
    Th t = new Th("t1");
    Th t2 = new Th("t2");
    t.start();
    t.join();
    t2.start();
}

```

3、notify、wait方法，Java中的唤醒与等待方法，关键为synchronized代码块，参数线程间应相同，也常用Object作为参数。

14.分布式系统中如何保证数据的一致性?

15.拆分微服务应该注意哪些地方，如何拆分?

1、业务方面拆分：所有技术方面的考虑，包括架构设计和解耦拆分都要考虑业务的需要。在服务拆分时，先从业务角度确定拆分的方案。拆分的边界要充分考虑业务的独立性和专业性，比如搜索类服务、支付类服务、购物车类服务，按服务的业务功能合理地划出拆分边界。

2、减少维护成本：拆分前的维护成本 - 拆分后的维护成本 ≥ 0

3、服务独立：确保拆分后的服务由相对独立的团队负责维护，尽量不要出现在不同服务之间的交叉调用。

4、系统扩展：拆分的一个重要理由也是最有价值的结果是提高了系统的扩展性。用户对不同的服务有不同的并发和性能方面的要求，因此服务具有不同的扩展性。把具有不同扩展性要求的服务拆分出来分别进行部署，可以降低成本，提高效率。

16.SpringCloud全家桶包含哪些组件?

1. Ribbon，客户端负载均衡，特性有区域亲和、重试机制。
2. Hystrix，客户端容错保护，特性有服务降级、服务熔断、请求缓存、请求合并、依赖隔离。
3. Feign，声明式服务调用，本质上就是Ribbon+Hystrix
4. Stream，消息驱动，有Sink、Source、Processor三种通道，特性有订阅发布、消费组、消息分区。
5. Bus，消息总线，配合Config仓库修改的一种Stream实现，
6. Sleuth，分布式服务追踪，需要搞清楚TraceID和SpanID以及抽样，如何与ELK整合。
7. Eureka，服务注册中心，特性有失效剔除、服务保护。
8. Dashboard，Hystrix仪表盘，监控集群模式和单点模式，其中集群模式需要收集器Turbine配合。
9. Zuul，API服务网关，功能有路由分发和过滤。

10. Config，分布式配置中心，支持本地仓库、SVN、Git、Jar包内配置等模式，

17.有没了解Docker，Docker和虚拟机有什么区别?

1、虚拟机：我们传统的虚拟机需要模拟整台机器包括硬件，每台虚拟机都需要有自己的操作系统，虚拟机一旦被开启，预分配给他的资源将全部被占用。，每一个虚拟机包括应用，必要的二进制和库，以及一个完整的用户操作系统。

2、Docker：容器技术是和我们的宿主机共享硬件资源及操作系统可以实现资源的动态分配。

容器包含应用和其所有的依赖包，但是与其他容器共享内核。容器在宿主机操作系统中，在用户空间以分离的进程运行。

3、对比：

1. docker启动快速属于秒级别。虚拟机通常需要几分钟去启动。

2. docker需要的资源更少，docker在操作系统级别进行虚拟化，docker容器和内核交互，几乎没有性能损耗，性能优于通过Hypervisor层与内核层的虚拟化。；

3. docker更轻量，docker的架构可以共用一个内核与共享应用程序库，所占内存极小。同样的硬件环境，Docker运行的镜像数远多于虚拟机数量。对系统的利用率非常高

4. 与虚拟机相比，docker隔离性更弱，docker属于进程之间的隔离，虚拟机可实现系统级别隔离；

5. 安全性： docker的安全性也更弱。Docker的租户root和宿主机root等同，一旦容器内的用户从普通用户权限提升为root权限，它就直接具备了宿主机的root权限，进而可进行无限制的操作。虚拟机租户root权限和宿主机的root虚拟机权限是分离的，并且虚拟机利用如Intel的VT-d和VT-x的ring-1硬件隔离技术，这种隔离技术可以防止虚拟机突破和彼此交互，而容器至今还没有任何形式的硬件隔离，这使得容器容易受到攻击。

6. 可管理性：docker的集中化管理工具还不算成熟。各种虚拟化技术都有成熟的管理工具，例如VMware vCenter提供完备的虚拟机管理能力。

7. 高可用和可恢复性：docker对业务的高可用支持是通过快速重新部署实现的。虚拟化具备负载均衡，高可用，容错，迁移和数据保护等经过生产实践检验的成熟保障机制，VMware可承诺虚拟机99.999%高可用，保证业务连续性。

8. 快速创建、删除：虚拟化创建是分钟级别的，Docker容器创建是秒级别的，Docker的快速迭代性，决定了无论是开发、测试、部署都可以节约大量时间。

9. 交付、部署：虚拟机可以通过镜像实现环境交付的一致性，但镜像分发无法体系化；Docker在Dockerfile中记录了容器构建过程，可在集群中实现快速分发和快速部署;

18.同一个宿主机中多个Docker容器之间如何通信？多个宿主机中Docker容器之间如何通信?

1、这里同主机不同容器之间通信主要使用Docker桥接（Bridge）模式。

2、不同主机的容器之间的通信可以借助于 pipework 这个工具。

19.高并发系统如何做性能优化？如何防止库存超卖?

1、高并发系统性能优化：

优化程序，优化服务配置，优化系统配置

1.尽量使用缓存，包括用户缓存，信息缓存等，多花点内存来做缓存，可以大量减少与数据库的交互，提高性能。

2.用jprofiler等工具找出性能瓶颈，减少额外的开销。

3.优化数据库查询语句，减少直接使用hibernate等工具的直接生成语句（仅耗时较长的查询做优化）。

4.优化数据库结构，多做索引，提高查询效率。

5.统计的功能尽量做缓存，或按每天一统计或定时统计相关报表，避免需要时进行统计的功能。

6.能使用静态页面的地方尽量使用，减少容器的解析（尽量将动态内容生成静态html来显示）。

7.解决以上问题后，使用服务器集群来解决单台的瓶颈问题。

2、防止库存超卖：

1、悲观锁：在更新库存期间加锁，不允许其它线程修改；

1、数据库锁：select xxx for update；

2、分布式锁；

2、乐观锁：使用带版本号的更新。每个线程都可以并发修改，但在并发时，只有一个线程会修改成功，其它会返回失败。

1、redis watch：监视键值对，作用时如果事务提交exec时发现监视的监视对发生变化，事务将被取消。

3、消息队列：通过 FIFO 队列，使修改库存的操作串行化。

4、总结：总的来说，不能把压力放在数据库上，所以使用 "select xxx for update" 的方式在高并发的场景下是不可行的。FIFO同步队列的方式，可以结合库存限制队列长，但是在库存较多的场景下，又不太适用。所以相对来说，我会倾向于选择：乐观锁 / 缓存锁 / 分布式锁的方式。

20.如何保证服务幂等性?

1、概念：接口的幂等性实际上就是接口可重复调用，在调用方多次调用的情况下，接口最终得到的结果是一致的。有些接口可以天然的实现幂等性，比如查询接口，对于查询来说，你查询一次和两次，对于系统来说，没有任何影响，查出的结果也是一样。

2、GET幂等：值得注意，幂等性指的是作用于结果而非资源本身。怎么理解呢？例如，这个HTTP GET方法可能会每次得到不同的返回内容，但并不影响资源。

3、POST非幂等：因为它会对资源本身产生影响，每次调用都会有新的资源产生，因此不满足幂等性。

4、如何保证幂等性：

1、全局唯一id：如果使用全局唯一ID，就是根据业务的操作和内容生成一个全局ID，在执行操作前先根据这个全局唯一ID是否存在，来判断这个操作是否已经执行。如果不存在则把全局ID，存储到存储系统中，比如数据库、redis等。如果存在则表示该方法已经执行。

从工程的角度来说，使用全局ID做幂等可以作为一个业务的基础的微服务存在，在很多的微服务中都会用到这样的服务，在每个微服务中都完成这样的功能，会存在工作量重复。另外打造一个高可靠的幂等服务还需要考虑很多问题，比如一台机器虽然把全局ID先写入了存储，但是在写入之后挂了，这就需要引入全局ID的超时机制。

使用全局唯一ID是一个通用方案，可以支持插入、更新、删除业务操作。但是这个方案看起来很美但是实现起来比较麻烦，下面的方案适用于特定的场景，但是实现起来比较简单。

2、去重表：这种方法适用于在业务中有唯一标的插入场景中，比如在以上的支付场景中，如果一个订单只会支付一次，所以订单ID可以作为唯一标识。这时，我们就可以建一张去重表，并且把唯一标识作为唯一索引，在我们实现时，把创建支付单据和写入去去重表，放在一个事务中，如果重复创建，数据库会抛出唯一约束异常，操作就会回滚。

3、插入或更新：这种方法插入并且有唯一索引的情况，比如我们要关联商品品类，其中商品的ID和品类的ID可以构成唯一索引，并且在数据表中也增加了唯一索引。这时就可以使用InsertOrUpdate操作。在mysql数据库中如下：

```
insert into goods_category (goods_id,category_id,create_time,update_time)
       values(#{goodsId},#{categoryId},now(),now())
       on DUPLICATE KEY UPDATE
       update_time=now()
```

4、多版本控制：这种方法适合在更新的场景中，比如我们要更新商品的名字，这时我们就可以在更新的接口中增加一个版本号，来做幂等

```
boolean updateGoodsName(int id,String newName,int version);
```

在实现时可以如下

```
update goods set name=#{newName},version=#{version} where id=#{id} and version<${version}
```

5、状态机控制：这种方法适合在有状态机流转的情况下，比如就会订单的创建和付款，订单的付款肯定是在之前，这时我们可以通过在设计状态字段时，使用int类型，并且通过值类型的大小来做幂等，比如订单的创建为0，付款成功为100。付款失败为99

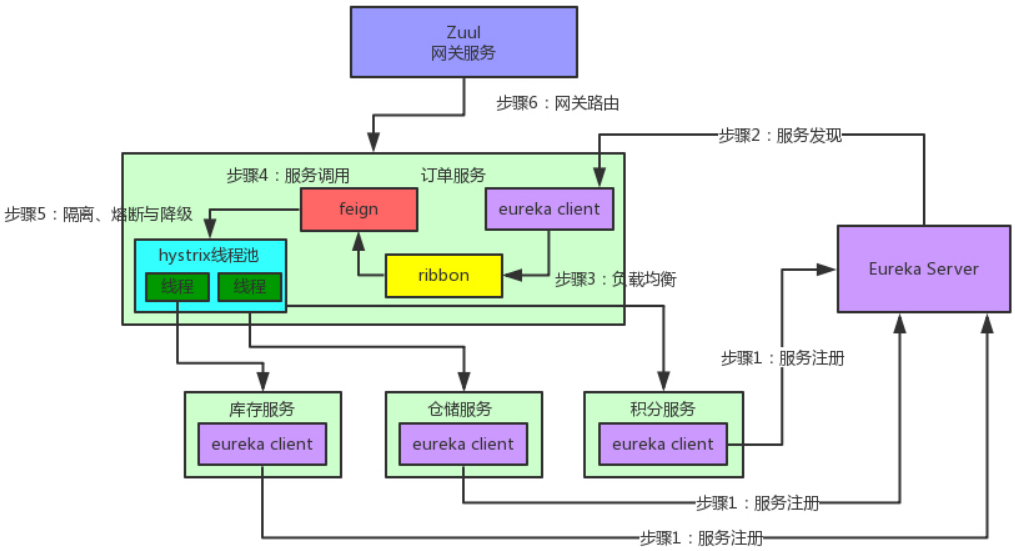
在做状态机更新时，我们就这可以这样控制

```
update `order` set status=#{status} where id=#{id} and status<#{status}
```

> 商汤--0301

1. springcloud有哪些核心组件，以及springcloud服务调用的详细工作流程？

1.



springcloud由以下几个核心组件构成：

Eureka：各个服务启动时，Eureka Client都会将服务注册到Eureka Server，并且Eureka Client还可以反过来从Eureka Server拉取注册表，从而知道其他服务在哪里

Ribbon：服务间发起请求的时候，基于Ribbon做负载均衡，从一个服务的多台机器中选择一台

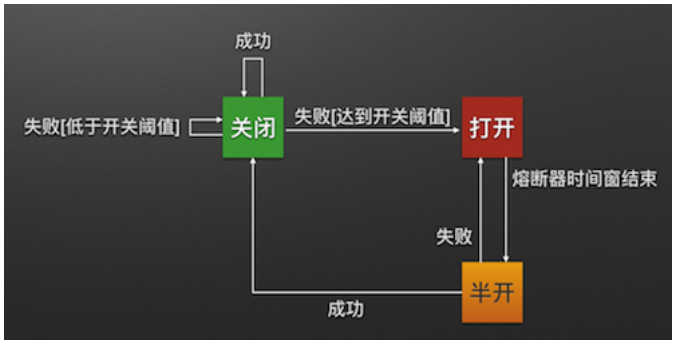
Feign：基于Feign的动态代理机制，根据注解和选择的机器，拼接请求URL地址，发起请求

Hystrix：发起请求是通过Hystrix的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题

Zuul：如果前端、移动端要调用后端系统，统一从Zuul网关进入，由Zuul网关转发请求给对应的服务

2. 熔断的原理，以及如何恢复?

熔断器模式定义了熔断器开关相互转换的逻辑:



服务的健康状况 = 请求失败数 / 请求总数.

熔断器开关由关闭到打开的状态转换是通过当前服务健康状况和设定阈值比较决定的.

1. 当熔断器开关关闭时, 请求被允许通过熔断器. 如果当前健康状况高于设定阈值, 开关继续保持关闭. 如果当前健康状况低于设定阈值, 开关则切换为打开状态.

2. 当熔断器开关打开时, 请求被禁止通过.

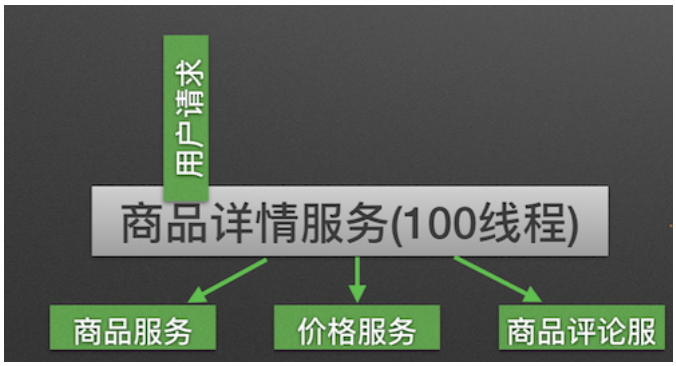
3. 当熔断器开关处于打开状态, 经过一段时间后, 熔断器会自动进入半开状态, 这时熔断器只允许一个请求通过. 当该请求调用成功时, 熔断器恢复到关闭状态. 若该请求失败, 熔断器继续保持打开状态, 接下来的请求被禁止通过.

熔断器的开关能保证服务调用者在调用异常服务时, 快速返回结果, 避免大量的同步等待. 并且熔断器能在一段时间后继续侦测请求执行结果, 提供恢复服务调用的可能.

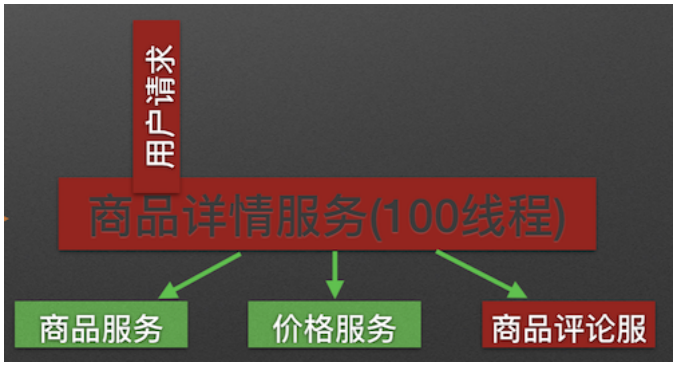
3. 假如A服务可以调用B服务，A服务也可以调用C服务，如果B服务挂了，大量的A-B请求过来，Hystrix如何防止服务雪崩（Hystrix的隔离流程）？

在一个高度服务化的系统中,我们实现的一个业务逻辑通常会依赖多个服务,比如:

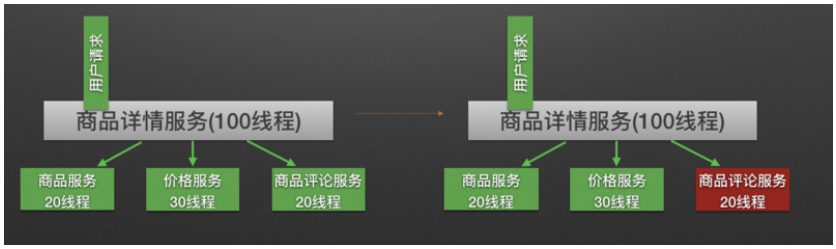
商品详情展示服务会依赖商品服务, 价格服务, 商品评论服务. 如图所示:



调用三个依赖服务会共享商品详情服务的线程池. 如果其中的商品评论服务不可用, 就会出现线程池里所有线程都因等待响应而被阻塞, 从而造成服务雪崩. 如图所示:



Hystrix通过将每个依赖服务分配独立的线程池进行资源隔离, 从而避免服务雪崩.
如下图所示, 当商品评论服务不可用时, 即使商品服务独立分配的20个线程全部处于同步等待状态,也不会影响其他依赖服务的调用.



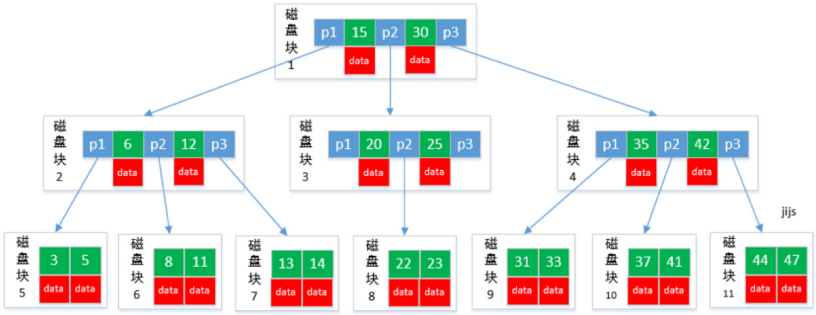
4. mysql有哪些搜索引擎，以及他们之间的区别?
    a. InnoDB：
        1. 支持事务处理
        2. 支持外键
        3. 支持行锁
        4. 不支持FULLTEXT类型的索引（在Mysql5.6已引入）
        5. 不保存表的具体行数，扫描表来计算有多少行
        6. 对于AUTO_INCREMENT类型的字段，必须包含只有该字段的索引
        7. DELETE 表时，是一行一行的删除
        8. InnoDB 把数据和索引存放在表空间里面
        9. 跨平台可直接拷贝使用
        10. 表格很难被压缩
    b. MyISAM：
        1. 不支持事务，回滚将造成不完全回滚，不具有原子性
        2. 不支持外键
        3. 支持全文搜索
        4. 保存表的具体行数,不带where时，直接返回保存的行数
        5. DELETE 表时，先drop表，然后重建表

6. MyISAM 表被存放在三个文件 。frm 文件存放表格定义。 数据文件是MYD (MYData) 。 索引文件是MYI (MYIndex)引伸
7. 跨平台很难直接拷贝
8. AUTO_INCREMENT类型字段可以和其他字段一起建立联合索引
9. 表格可以被压缩

c. 选择：因为MyISAM相对简单所以在效率上要优于InnoDB.如果系统读多，写少。对原子性要求低。那么MyISAM最好的选择。且MyISAM恢复速度快。可直接用备份覆盖恢复。如果系统读少，写多的时候，尤其是并发写入高的时候。InnoDB就是首选了。两种类型都有自己优缺点，选择那个完全要看自己的实际类弄。

5. 请介绍一下mysql索引B+tree。

1. B-tree：

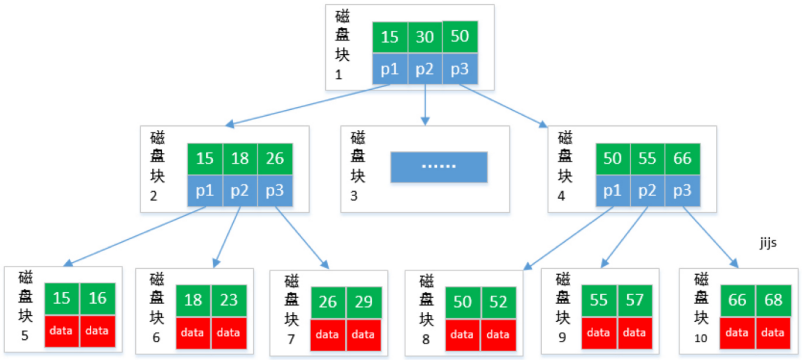


B-tree 利用了磁盘块的特性进行构建的树。每个磁盘块一个节点，每个节点包含了很关键字。把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度。

B-tree巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页（每页为4K），这样每个节点只需要一次I/O就可以完全载入。

B-tree 的数据可以存在任何节点中。

2. B+tree：



B+tree 是 B-tree 的变种，B+tree 数据只存储在叶子节点中。这样在B树的基础上每个节点存储的关键字数更多，树的层级更少所以查询数据更快，所有指关键字指针都存在叶子节点，所以每次查找的次数都相同所以查询速度更稳定;

6. kafka的工作流程?

理解这篇就可以：https://www.w3cschool.cn/apache_kafka/apache_kafka_workflow.html