- 谈谈你对UnSafe的理解
- AQS（AbstractQueuedSynchronizer）
  - 工作原理
- ReentrantLock如何获得锁

微服务框架

- Spring
  - 创建Spring IOC流程
  - 创建Spring Bean的生命周期
  - Spring 的 AOP 是怎么实现的?
    - Spring 的 AOP 有哪几种创建代理的方式?
    - JDK 动态代理和 Cglib 代理的区别?
    - @PostConstruct修饰的方法/init-method里用到了其他 bean 实例，会有问题吗?
  - 要在 Spring IOC 容器构建完毕==之后==执行一些逻辑，怎么实现?
  - Spring怎么解决循环依赖的问题?
  - 什么是事务
  - Spring 事务的实现原理?
  - Spring事务传播机制/行为
  - Spring事务隔离级别
  - Spring 的事务隔离级别是如何做到和数据库不一致的?
  - bean的自动装配
  - spring事务什么时候会失效?
  - Spring AOP 所有通知执行顺序
- Spring Boot
  - SpringBoot 自动配置原理
- MyBatis
  - MyBatis工作原理
- Spring Cloud
  - 注解
  - Eureka
    - Eureka自我保护机制
  - Ribbon
    - LB负载均衡(Load Balance)
    - 负载均衡(轮询)算法原理
      - Ribbon本地负载均衡客户端 VS Nginx服务端负载均衡区别
  - Feign与OpenFeign
    - OpenFeign超时控制
    - OpenFeign日志打印功能
  - Hystrix断路器
  - zuul路由网关
  - Gateway新一代==网关==
    - 为什么会出现Gateway
    - Gateway工作流程
  - 配置中心
  - Stream 消息驱动
    - 工作原理图
  - Sleuth 请求链路跟踪
    - zipkin 链路跟踪原理
  - SkyWalking 请求链路跟踪
  - Nacos服务注册和配置中心
    - 各个注册中心对比
  - Seata处理分布式事务
    - 分布式事务
    - Seata处理流程
- Spring Security
  - Spring Security工作原理
    - 常用注解

- 分布式id生成方案
  - uuid
  - 数据库自增序列
    - 单机模式：
  - 基于redis、mongodb、zk等中间件生成
  - ==Leaf-segment（美团的Leaf算法）==
  - 雪花算法
- 分布式事务解决方案

ElasticSearch

- 倒排索引
  - 原理
  - 底层数据结构

Zookeeper

- 为什么Zookeeper可以用来作为注册中心
- Zookeeper可以用来作为配置中心、集群管理
- ZAB协议
- 简述Paxos算法
- 领导者选举的流程是怎样的
- Zookeeper集群中节点之间数据是如何同步的
- 讲下Zookeeper watch机制
- zk实现分布式锁
- zk的数据模型和节点类型（了解）

消息队列

- RabbitMQ
  - RabbitMQ架构设计/工作原理
    - 生产者发送消息的流程
    - 消费者获取消息的过程
- Kafka
  - Kafka架构设计/工作原理
    - 发送消息
    - 消费数据
  - Kafka的Pull和Push分别有什么优缺点
- RocketMQ架构设计/工作原理
  - RocketMQ工作流程
  - RocketMQ的事务消息是如何实现的
- 消息队列如何保证消息可靠传输
  - 如何保证消息幂等性

场景题

- redis分布式锁的应用场景
- 1. 线上几百万条数据积压如何处理?
- 2. 超时未支付解决方案
- 3. 1000个数据按顺序执行复杂计算，要求过程中重启后数据还能安全继续执行
- 4. MySQL分表、分库、分区
- 如何实现亿级的数据统计
- 618的排行榜是如何实现的
- 如何从1000w条数据中找出最热门的10个记录

0. 前言

这套面试题整理什么?

1. 自己见到的，且目前答案不全面的
2. 自己觉得难的
3. 自己刷题过程中出过错的

1. 基础篇

1. String、StringBuffer和StringBuilder

String

- String:字符串，使用一对""引起来表示

1. String 声明为==final==的，不可被继承
2. String实现了Serializable接口：表示字符串是==支持序列化的==。
   实现了Comparable接口：表示String==可以比较大小==
3. String内部定义了==final char[] value==用于存储字符串数据
4. String:代表不可变的字符序列。简称：不可变性
   - 体现：
     1. 当对字符串重新赋值时，需要重写指定内存区域，不能使用原有的value进行赋值
     2. 当对现有的字符串进行连接操作时，也需要重新指定内存赋值，不能再原有的value进行赋值
     3. 当调用String的replace()方法修改字符串时，需要重新指定内存赋值，不能再原有的value进行赋值
5. 通过字面量的方式(区别new)给一个字符串赋值，此时的字符串值声明在字符串常量池中。
6. ==字符串常量池中是不会存储相同的内容的字符串的==

String、StringBuffer、StringBuilder三者的异同?

1. 异同

- String:不可变的字符序列；底层使用char[]存储
- StringBuffer:可变的字符序列；线程安全的，效率低；底层使用char[]存储
- StringBuilder:可变的字符序列；jdk5.0新增的，线程不安全的，效率高；底层使用char[]存储
- StringBuffer与StringBuilder有一个共同的父类AbstractStringBuilder

1. 源码分析：

```
String str = new String();//char[] value = new char[0];
String str1 = new String("abc");//char[] value = new char[]{'a','b','c'};

StringBuffer sb1 = new StringBuffer();//char[] value = new char[16];底层创建了一个长度是16的数组。
System.out.println(sb1.length());//
sb1.append('a');//value[0] = 'a';
sb1.append('b');//value[1] = 'b';

StringBuffer sb2 = new StringBuffer("abc");//char[] value = new char["abc".length() + 16];
```

1. StringBuffer、StringBuilder扩容问题

- 如果要添加的数据底层数组盛不下了，那就需要扩容底层的数组。
  - 默认情况下，扩容为原来容量的2倍 + 2，同时将原有数组中的元素复制到新的数组中。

2. 集合

数组和集合的异同?

1. 数组一旦初始化以后，其长度就不可修改
2. 数组中提供的方法非常有限，对于添加、删除、插入数据等操作，非常不便。同时效率不高
3. 获取数组中实际的个数的需求，数组并没有现成的属性和方法可用
4. 数组存储数据的特点：有序、可重复。对于无序、不可重复的需求，不能满足。

Collection 和 Collections 的区别?

1. Collection是存储单列数据集合的接口，里面常见的接口有List、Set。
2. Collections类是操作Collection和Map的工具类(注意：Collection是接口，Collections是类)

ArrayList、LinkedList、Vector的源码分析

Collection重要接口介绍

```
|----Collection接口： 单列集合，用来存储一个一个对象
          |----List接口:存储有序的、可重复的数据     -->"动态数组"
               |---ArrayList：作为List接口的主要实现类；线程不安全，效率高；底层用Object[] elementData存储
               |---LinkedList：对于频繁的插入、删除操作，使用此类效率比ArrayList高，底层使用双向链表存储
               |---Vector：作为List接口的古老实现类；线程安全，效率低；底层使用Object[] elementData存储
```

ArrayList

- jdk7中的ArrayList的创建类似于单例的饿汉式，而jdk8中的ArrayList的对象的创建类似于单例的懒汉式，延迟了数组的创建，节省内存

jdk7

```
 ArrayList list=new ArrayList(); //底层创建了长度为10的Object[] elementData数组
 list.add(124); //elementData[0]=new Integer(124);
 ...
 list.add(11); //如果此次的添加导致底层elementData数组容量不够，则扩容。
                默认情况下，扩容为原来的容量的1.5倍，同时需要将原有数组中的数据复制到新的数组
中。
 结论：建议开发中使用带参的构造器：ArrayList list =new ArrayList(int initialCapacity)
```

jdk8

```
 ArrayList list=new ArrayList(); //底层Object[] elementData 初始化为{}，并没有创建长度
 list.add(123); //第一次调用add()时,底层才创建了长度10的数组，并将数据123添加到
elementData[0]
 ...
 后续的添加和扩容操作与jdk7无异
```

LinkedList

```
 LinkedList list= new LinkedList();  //内部声明了Node类型的first和last属性，默认值为
null
 list.add(123); //将123封装到Node中，创建了Node对象。

 其中，Node定义为：  证明(体现)LinkedList为双向链表
    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
```

Vector

```
jdk7和jdk8中通过Vector()构造器创建对象时，底层都创建了长度为10的数组
在扩容方面，默认扩容为原来的数组长度的2倍。
```

HashMap的源码分析

Map重要实现类介绍

```
|-----Map：双列集合，用来存储一对一对的数据(key-value)
        |-----HashMap: 作为Map的主要实现类;线程不安全，效率高；存储null的key和value
             |-----LinkedHashMap: 保证在遍历map元素时，可以按照添加的顺序实现遍历
                                  原因：在原有的HashMap底层结构基础上，添加了一对指针，指
向前一个和后一个元素
                                  对于频繁的遍历操作，此类执行效率高于HashMap
        |-----SortedMap
```

```
            |-----TreeMap: 保证按照添加的key-value对进行排序，(按照key)实现排序遍历。此时考虑key的自然排序和定制排序
                                底层使用红黑树;不能存储null的key值
        |-----Hashtable: 作为古老的实现类;线程安全的，效率低;不能存储null的key和values
            |-----Properties: 常用来处理配置文件。key和value都是String类型

Map结构的理解：
    Map中的key:无序的、不可重复的，使用Set存储所有的key    -->key所在的类要重写equals()和hashCode()方法(以HashMap为例)
    Map中的value:无序的、可重复的，使用Collection存储所有的value   --->value所在的类要重写equals()
    一个键值对：key-value构成了一个Entry对象
    Map中的Entry:无序的、不可重复的，使用Set存储所有的Entry
```

HashMap的底层实现原理

```
HashMap的底层：  数组+链表  (jdk7及之前)
                 数组+链表+红黑树 (jdk8)

HashMap的底层实现原理？以jdk7为例说明
    HashMap map=new HashMap():
    在实例化以后，底层创建了长度为16的一维数组Entry[] table.
     ...可能已经执行了很多put...
    map.put(key_n,value_n):
    首先，调用key1所在类的hashCode()计算key1哈希值以后，此hash值经过某种算法计算以后，得到在Entry[]数组中的存放位置
    如果此位置的数据为空，此时的key1-value1(意味着Entry)添加成功   --->情况1
    如果此位置上的数据不为空，（意味着此位置上存在一个或多个数据(以链表形式存在)),
     比较key1和已经存在的一个或多个数据的哈希值：
          如果key1的哈希值与已经存在的数据的哈希值都不相同，此时key1-value1添加成功。   -->情况2
          如果key1的哈希值和已经存在的某一个数据(key2-value2)的哈希值相同，继续比较:调用key1所在类的equals()方法，比较：
              如果equals()返回false:此时key1-value1添加成功   -->情况3
              如果equals()返回true:使用value1替换相同key的value2值.

  补充：关于情况2和情况3：
    此时key1-value1和原来的数据以链表的方式存储
    在不断的添加过程中，会涉及到扩容问题，当超出临界值(threshold=12)且此位置不为空(需要形成链表)时，
    默认的扩容方式：扩容为原来容量的2倍，并将原有的数据复制过来

    jdk8 相较于jdk7在底层实现方面的不同：
    1.new HashMap():底层没有创建一个长度为16的数组
    2.jdk8底层的数组是：Node[],而非Entry[]
        jdk8:
            static class Node<K,V> implements Map.Entry<K,V>{...}
        jdk7:
            static class Entry<K,V> implements Map.Entry<K,V>{...}
    3.首次调用put方法时，底层创建长度为16的数组
    4.jdk7底层结构只有：数组+链表。jdk8中底层结构：数组+链表+红黑树。
         4.1 形成链表时，七上八下（jdk7:新的元素指向旧的元素。jdk8：旧的元素指向新的元素）
         4.2 当数组的某一个索引位置上的元素以链表形式存在的数据个数 > 8 且当前数组的长度 > 64时，此时此索引位置上的所有数据改为使用红黑树存储。
```

```
DEFAULT_INITIAL_CAPACITY : HashMap的默认容量，16
DEFAULT_LOAD_FACTOR：HashMap的默认加载因子：0.75
threshold：扩容的临界值=容量*填充因子：16 * 0.75 => 12
TREEIFY_THRESHOLD：Bucket中链表长度大于该默认值，转化为红黑树:8
MIN_TREEIFY_CAPACITY：桶中的Node被树化时最小的hash表容量:64
```

HashMap使用需要注意什么?

- 首先了解过其底层的都知道，HashMap有个扩容机制，是比较耗时的，所以为了减少扩容次数，在知道要存放多少元素的前提下最好指定HashMap的链表初始大小。
- 为了减少链表的碰撞次数，尽可能的选择不可变的类型作为Key，以为其不可变形，其HashCode的值也会不可变。如String类型
- 使用HashMap做缓存时，因为其线程不安全特性，最好使用ConcurrentHashMap代替。
- key相同会覆盖之前的数据，因为其key的hashcode相同会向其index所对应的entry进行equals相同则覆盖。

Hashtable\HashMap\ConcurrentHashMap

Hashtable\HashMap\ConcurrentHashMap的区别?

```
1.Hashtable、ConcurrentHashMap是线程安全的，且键\值不能为null
2.效率：HashMap>ConcurrentHashMap>Hashtable
3.Hashtable的初始化容量是11，扩容时是当前容量*2+1
  HashMap\ConcurrentHashMap的初始化容量是16，扩容时是当前容量*2
```

Hashtable

```
1.Hashtable的每个操作都使用了synchronized上了锁，甚至读的操作也上锁。
2.Hashtable的键\值不能为null
3.Hashtable的初始化容量是11，扩容时是当前容量*2+1
```

ConcurrentHashMap

> jdk7

```
构造方法-->put方法
一、构造方法
new ConcurrentHashMap<String,String>();
----
public ConcurrentHashMap(
int initialCapacity,
float loadFactor,
int concurrencyLevel) {
     ...
 }
initialCapacity(16)：容量(（Segment对象中）HashEntry数组长度总和)
loadFactor(0.75f)：扩容因子
concurrencyLevel(16):并发等级，最多支持多少个线程操作Segment数组，也就是Segment数组的长度
----
1.无参构造：
默认Segment数组长度为16，（Segment对象中）HashEntry数组长度为2
```

为什么是2，不是1？
理论上是1，但是因为最后控制了，他最小容量必须是2

```
//理论是1
int c = initialCapacity / ssize;
if (c * ssize < initialCapacity)
    ++c;
//最小容量必须是2
int cap = MIN_SEGMENT_TABLE_CAPACITY;
while (cap < c)
    cap <<= 1;
//Segment数组的索引为0的位置存储Segment对象s0,其中规定了扩容因子、扩容阈值、HashEntry数组长度
Segment<K,V> s0 =
    new Segment<K,V>(loadFactor, (int)(cap * loadFactor),(HashEntry<K,V>[])new HashEntry[cap]);
```

2.有参构造：
2.1.如果规定了concurrencyLevel，则会计算出Segment数组的长度ssize，且ssize是2的倍数
2.2.通过ssize算出HashEntry数组长度cap,且cap为2的倍数

二、put方法
1.通过hashcode & segment数组长度-1（segmentMask）获得segment数组索引
2.判断此索引上是否存在segment对象
    2.1 如果存在，则调用segment对象的put方法向HashEntry中添加元素
    2.2 如果不存在，则利用s0，获取或设置一个segment对象
        2.2.1 会通过getObjectVolatile()方法，拿此位置的segment对象
        2.2.2 如果一直拿不到此位置上的Segment对象则，通过compareAndSwapObject(CAS，比较并交换)，设置此位置的Segment对象
3.调用segment对象的put方法
    3.1 先上锁
        先进行tryLock尝试加锁，如果不能尝试加锁则自旋加锁。
        如果自旋加锁到达规定的重试次数，利用lock()完成阻塞加锁
    3.2 计算出当前hashcode
    3.3 确定hashEntry的位置，找到此hashEntry[i]的头节点first；
        a. 此节点是否是null，若不是则需要遍历
        b. 遍历完成或HashEntry对象key都不相同，创建HashEntry，插入
        c. 此处要注意判断当前segement的count数量,是否需要进行rehash(扩容)
    3.4 释放锁
    （3.1.先进行tryLock尝试加锁，如果不能尝试加锁然后scanAndLockForPut阻塞加锁（自旋锁+ReentrantLock加锁）
    <加锁中...>
    <与HashMap的put方法一致>
    3.2. scanAndLockForPut():有多个线程
      a. tryLock():尝试加锁
      b. 先拿出这个位置上的HashEntry，如果有则遍历链表,找出key相同的，或者链表尾部，到了尾部创建HashEntry
      c. 如果自旋加锁到达规定的重试次数，利用lock()完成阻塞加锁
      d. 返回创建的HashEntry）

三、扩容问题：
1.扩容扩建的是（Segment对象中）HashEntry数组，都是单独的Segment对象，单独扩容。Segment数组传入的是多少就是多少
2.HashEntry数组扩容与HashMap一致，都是扩容为原来的2倍，并将内容复制进新数组

总结：
1. 1.7版本使用的是数组+链表结构

存入的数据用Segment类型来封装。
2.

> jdk1.8

构造方法-->put方法
一、构造方法
```
new ConcurrentHashMap<String,String>();
----
public ConcurrentHashMap() {}
----
```

二、put方法
1. 根据hash值，确定此Node<K,V>[] table中的真实索引位置
2.如果没有初始化Node数组，则初始化
3.如果初始化了Node数组，判断此位置上是否存在Node对象
    3.1 位置没有元素，则用cas自旋获锁，存入节点
4.位置有元素，且正在扩容，则协助其转移
5.位置有元素，则对根节点上锁（synchronized ）
    5.1 如果是链表，尾插法
    5.2 判断链表上元素的个数是否大于等于8，如果大于等于8就升级为红黑树。

```
 final V putVal(K key, V value, boolean onlyIfAbsent) {
        if (key == null || value == null) throw new NullPointerException();
        int hash = spread(key.hashCode());
        int binCount = 0;
        for (Node<K,V>[] tab = table;;) {
            Node<K,V> f; int n, i, fh;
            if (tab == null || (n = tab.length) == 0)
                tab = initTable(); //初始化数组
            else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {//该位置没有元
素，则用cas自旋获锁，存入节点
                if (casTabAt(tab, i, null,
                             new Node<K,V>(hash, key, value, null)))
                    break;                   // no lock when adding to empty bin
            }
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);//如果ConcurrentHashMap正在扩容，则协助其
转移
            else {
                V oldVal = null;
                synchronized (f) {//对根节点上锁
                    if (tabAt(tab, i) == f) {
                        if (fh >= 0) {//fh>=0说明是链表，否则是红黑树
                            binCount = 1;
                            for (Node<K,V> e = f;; ++binCount) {
                                K ek;
                                if (e.hash == hash &&
                                    ((ek = e.key) == key ||
                                     (ek != null && key.equals(ek)))) {
                                    oldVal = e.val;
                                    if (!onlyIfAbsent)
                                        e.val = value;
                                    break;
                                }
```

```
                                Node<K,V> pred = e;
                                if ((e = e.next) == null) {
                                    pred.next = new Node<K,V>(hash, key,
                                                              value, null); //尾插
法插入
                                    break;
                                }
                            }
                        }
                        else if (f instanceof TreeBin) {//红黑树
                            Node<K,V> p;
                            binCount = 2;
                            if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                           value)) != null) {
                                oldVal = p.val;
                                if (!onlyIfAbsent)
                                    p.val = value;
                            }
                        }
                    }
                }
                if (binCount != 0) {{//判断链表的值是否大于等于8，如果大于等于8就升级为红
黑树。
                    if (binCount >= TREEIFY_THRESHOLD)
                        treeifyBin(tab, i);
                    if (oldVal != null)
                        return oldVal;
                    break;
                }
            }
        }
        addCount(1L, binCount);
        return null;
    }
```

总结：

1.1.8版本放弃了Segment，跟HashMap一样，用Node描述插入集合中的元素。但是Node中的val和next使用了volatile来修饰，保存了内存可见性。与HashMap相同的是，ConcurrentHashMap1.8版本使用了数组+链表+红黑树的结构。

2.同时，ConcurrentHashMap使用了CAS+Synchronized保证了并发的安全性

LinkedHashMap的底层实现原理

```
static class Entry<K,V> extends HashMap.Node<K,V> {
         Entry<K,V> before, after; //能够记录添加的元素的先后顺序,对于频繁的遍历操作,考虑
LinkedHashMap
         Entry(int hash, K key, V value, Node<K,V> next) {
             super(hash, key, value, next);
         }
     }
```

BIO、NIO、AIO的区别?

1. BIO（Blocking I/O）：
   BIO是最传统的I/O模型，==在BIO中，I/O操作是阻塞的，也就是说当一个线程执行I/O操作时，它将被阻塞，直到操作完成或超时==。这意味着在进行I/O操作时，线程无法执行其他任务，因此需要使用多线程来处理多个并发连接。BIO适用于连接数较少且连接时间较长的场景，例如传统的客户端/服务器模型。
2. NIO（Non-blocking I/O）：
   NIO是Java 1.4引入的改进版I/O模型，它引入了通道（Channel）和缓冲区（Buffer）的概念。在NIO中，==I/O操作是非阻塞的，线程在执行I/O操作时不会被阻塞，可以继续执行其他任务==。NIO使用选择器（Selector）来监听多个通道上的事件，当一个通道有数据可读或可写时，才会进行相应的读写操作。NIO适用于连接数较多且连接时间较短的场景，例如实现高性能的网络服务器。
3. AIO（Asynchronous I/O）：
   AIO是Java 1.7引入的最新的I/O模型，也称为NIO.2。==AIO采用了异步操作的方式，也就是说当一个I/O操作启动后，线程可以继续执行其他任务，当操作完成时，会通过回调方式通知线程进行处理==。AIO适用于连接数较多且连接时间不确定的场景，例如实现高并发的网络服务器，或者需要大量并发读写文件的情况。

ThreadLocal的原理和使用场景

强引用、软引用、弱引用、虚引用分别是什么?

1. 强引用(默认支持模式)
   - 当内存不足，JVM开始垃圾回收，对于强引用的对象，就算是出现了OOM也不会对该对象进行回收，死都不收。
2. 软引用(SoftReference)
   - 当系统内存充足时它==不会==被回收，当系统内存不足时它==会==被回收。
3. 弱引用(WeakReference)
   - 只要垃圾回收机制一运行，不管JVM的内存空间是否足够，都会回收该对象占用的内存。
4. 虚引用(PhantomReference)
   - 如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收

浏览器输入URL后发生了什么

1. 浏览器先分析超链接中域名是否规范
2. 浏览器向DNS请求解析请求解析超链接中的IP地址
3. DNS将解析出的ip地址返回浏览器
4. 浏览器与服务器建立TCP连接
5. 浏览器请求资源
6. 服务器给出响应，将资源发送给浏览器，浏览器进行解封装。
7. 浏览器渲染页面
8. 释放TCP连接

MySQL

binlog与redolog对比、binlog、undolog对比

MVCC整体操作流程

ReadView的规则

MySQL 的可重复读怎么实现的?

> MVCC:Undo log多版本链 + Read view

请说一下数据库锁的种类?

并发事务会产生哪些问题

1. 更新丢失
2. 脏读
3. 不可重复读
4. 幻读

什么是分库分表?什么时候进行分库分表?

> 马士兵-2023年\2023新版-B站面试答案讲义\2023新版-MySQL面试题-答案讲义..pdf

说一下 MySQL 执行一条查询语句的内部执行过程?

> 马士兵-2023年\2023新版-B站面试答案讲义\2023新版-MySQL面试题-答案讲义..pdf

多线程

线程

线程的状态

线程的结束方式

> interrupt()方式

- 通过打断WAITING或者TIMED_WAITING状态的线程，从而抛出异常自行处理

###

线程池

企业中通常使用自定义线程

> ThreadPoolExecutor的使用
> https://blog.csdn.net/Day_and_Night_2017/article/details/124404629

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler);
```

`int corePoolSize`：核心线程数。代表线程池主要用来执行任务的线程的数量。
`int maximumPoolSize`：最大线程数。当核心线程处理不过来的时候，会创建一些非核心线程一起处理。核心线程+非核心线程数的最大值就是最大线程数。
`long keepAliveTime`：非核心线程的最大空闲时间 。当任务数量逐渐稀疏以后，非核心线程逐渐得以空闲，达到最大空闲时间后，会被线程池销毁。当`allowCoreThreadTimeout()`设置为`true`时，核心线程也不例外，超过最大空闲时间也会被销毁。
`TimeUnit unit`：keepAliveTime的时间单位
`BlockingQueue workQueue`：阻塞队列。 当前核心线程达到最大值时，优先向阻塞队列进行填充。
`ThreadFactory threadFactory`：线程工厂，用于为线程池提供线程。如果不指定，会采用默认的工厂。
`RejectedExecutionHandler handler`：拒绝策略。当线程达到最大线程数时，将采用拒绝策略处理后续的任务。

JVM

1.JVM调优案例（https://blog.csdn.net/v123411739/article/details/123778478）

1.1 基本思路

1. 先肯定，JVM一般不需要调优

```
我们的 JVM 参数配置大多还是会遵循 JVM 官方的建议
1.-XX:NewRatio=2，年轻代:老年代=1:2

2.-XX:SurvivorRatio=8，eden:survivor=8:1

3.堆内存设置为物理内存的3/4左右
```

2. 少量场景需要调优

2.1 其次说明可能还是存在少量场景需要调优，我们可以对一些 JVM核心指标配置监控告警，当出现波动时人为介入分析评估

```
1.AVG/TP999/TP9999 耗时
    1.1 jvm.gc.time：每分钟的GC耗时在1s以内，500ms以内尤佳
    1.2 jvm.gc.meantime：每次YGC耗时在100ms以内，50ms以内尤佳
    1.3 jvm.fullgc.count：FGC最多几小时1次，1天不到1次尤佳
    1.4 jvm.fullgc.time：每次FGC耗时在1s以内，500ms以内尤佳


2.
```

3. 分析排查

3.1 CPU指标

```
1*. 查看占用CPU最多的进程\线程
2*.查看线程堆栈快照信息
    常见的工具：JProfiler、JVM、Profiler、Arthas等。
分析代码执行热点
查看哪个代码占用CPU执行时间最长
查看每个方法占用CPU时间比例
**例子**：
    // 显示系统各个进程的资源使用情况
    top
    // 查看某个进程中的线程占用情况
    top -Hp pid
    // 查看当前 Java 进程的线程堆栈信息
    jstack pid
```

3.2 JVM 内存指标

```
1.查看当前 JVM 堆内存参数配置是否合理
2.查看是哪个区域导致的GC
**例子**：
    // 输出 Java 堆详细信息
    jmap -heap pid
    // 显示堆中对象的统计信息
    jmap -histo:live pid
此时需要使用到：      -Xms     指定初始堆大小
                -Xmx     指定堆的上限大小
                -Xmn     指定年轻代的大小
                -Xss     指定虚拟机栈的大小
                -XX:MaxMetaspaceSize 设置元空间的上限大小
```

3.3 JVM GC指标

```
1.查看每分钟YGC\FullGC\GC时间是否正常
```

```
// 打印GC的详细信息
-XX:+PrintGCDetails

// 在GC前后打印堆信息
-XX:+PrintHeapAtGC

// 打印GC时应用程序的停止时间
-XX:+PrintGCApplicationStoppedTime
```

4. 确定优化目标

5. 列举真实案例

5.1 背景

晚上8点是我们的业务高峰，一到高峰的时候，发现TP99耗时会变高，有明显的毛刺,通过排查发现内存使用率也会增大，然后再释放,其他各项指标正常，于是怀疑是GC导致的,观察服务器的GC情况,发现YongGC情况如下，大概每5分钟，GC55次，峰值最高可以达到220次。FullGC比较频繁,每5分钟大概0.5次，峰值50次。 那么问题在于Fullgc频繁,而且youngGC峰值也很高

5.2 原因

==FullGC频繁，那么会触发stop-the- world==。此时会导致我们的系统进行停顿,这个可能是导致我们的系统tp99耗时上升的主要原因。

由于并发很高，我们的Y==oungGC频繁==，那么可能会造成，我们有些本应该在==YoungGC就回收的对象，没有回收成功==，直接进入了老年代，由于对象的晋升，导致了我们的老年代继续触发FullGC。于是峰值变高。

5.3 目标

1、YoungGC次数减少
2、YoungGC耗时减少
3、Ful1GC不超过6次一天
4、FucllGC耗时减少

5.4 优化

```
1、先看垃圾收集器
我们的jdk版本为8，并且这个服务未指定特定的收集器,所以走的是我们默认的收集器组台，年轻代为Parallel Scavenge，老年代为Parallel Old。
这两种并行收集器的组台提高了系统的吞吐量,而不是低延迟配比，我们首先应该换一种低延迟的收集器。
低延迟的组台，我们选择ParNew与CMS的组台，如果jdk的版本高，其实也可以选择Gl或者ZGC。

2 、年轻代参数配置
-Xms4096MB -Xmx4096MB -Xmn1024MB
以上的配置只配了堆的大小，像我们年轻代的占比都是走的默认的，-XX:SurvivorRatio= 8，也就是4G*0.2。总共0.8G。这个就比较小了，观察一下老年代的对象占用空间，大概是1.5G，也就是说有一些堆空间其实是空闲的。
那么当我们年轻代的空间小，而且并发大的时候，年轻代的对象会激增，并且晋升到老年代。
然而收集器ParralellOld又会导致stw，无法与用户线程并行，那么就会造成我们的服务停顿，TP99升高。

3、元数据区
jdk1.8后，原来的永久代变为元数据区，如果我们没有指定元数据区的大小。其默认的初始值只有21MB，那么我们如果是动态代理的对象比较多，就会导致我们的元数据区进行GC回收，
元数据区的回收也会触发FullGC，再次导致我们的stw。所以我们观看一下元数据的常驻对象的大小，大概是100M左右，
```

所以我们直接用参数指定元数据区的大小为256MB。我们的元数据区的最大容量也同时指定为256M，防止其进行动态调整。

4、并发预清理
在FullGC发生时，会产生我们的GC root追踪。
老年代与年轻代间又会存在跨年龄引用，如果我们在CMS收集器进行收集前，进行一次重新标记，其实会减少我们的对象扫描，减少我们的FullGC时间。所以我们就让进行FulIGC前，强制做一次MinorGC。
我们配置如下参数Xx:+CMSScavengeBeforeRemark，这样就减少了我们要扫描的对象,减少了 Remart时间。

1. 最后解决方案

   1、指定收集器为ParNew+CMS、扩充年轻代的占比为前的1.5倍、指定元数据区的大小
   2、使用并发预清理
   跟据上四种方式配置后，我们重新进行了一次压测，发现我们的TP99延时较前降低了6o%。FulIGC耗时降低80%。YoungGC次数减少30%。耗时基本持平，因为我们的年轻代容量变大。完全符台预期。

并发编程（JUC）

CAS

compare and swap的缩写，中文翻译成比较并交换,实现并发算法时常用到的一种技术。它包含三个操作数——内存位置、预期原值及更新值。
执行CAS操作的时候，将内存位置的值与预期原值比较：
如果相匹配，那么处理器会自动将该位置值更新为新值，
如果不匹配，处理器不做任何操作，多个线程同时执行CAS操作只有一个会成功。

CAS是一条CPU的原子指令（cmpxchg指令），不会造成所谓的数据不一致问题，Unsafe提供的CAS方法（如compareAndSwapXXX）底层实现即为CPU指令cmpxchg。

谈谈你对UnSafe的理解

++是CAS的核心类，由于Java方法无法直接访问底层系统，需要通过本地（native）方法来访问，Unsafe相当于一个后门，基于该类可以直接操作特定内存的数据++。
Unsafe类存在于sun.misc包中，其内部方法操作可以像C的指针一样直接操作内存，因为Java中CAS操作的执行依赖于Unsafe类的方法。

AQS（AbstractQueuedSynchronizer）

> 抽象的队列同步器

state：同步器状态变量，值为0时表示当前可以被加锁。值为1时表示有线程占用，其他线程需要进入到同步队列等待，同步队列是一个双向链表。
exclusiveOwnerThread：当前获取锁的线程
head：指向基于Node类构造的队列的队头，同步队列是一个双向链表。
tail：指向基于Node类构造的队列的队尾，同步队列是一个双向链表。
Thread：表示当前线程的引用，比如需要唤醒的线程。

工作原理

AQS 的工作原理基于一个内部的 FIFO 队列（CLH 队列），该队列维护等待获取锁的线程。当一个线程尝试获取锁时，如果锁已被占用，则线程将被加入到队列中并进入等待状态。一旦锁释放，AQS 会从队列中选择一个线程唤醒，使其获取锁。

ReentrantLock如何获得锁

> ReentrantLock加锁过程

1. 通过lock()方法加锁
2. 调用lock()之后，内部会判断当前锁的状态，只有在state=0的情况下锁才是空闲的。
3. 在空闲状态下，并不能立刻加锁成功
    - 如果是公平锁：空闲则用hasQueuePreProcessor()判断等待队列中有没有等待的线程。如果没有则直接加锁成功，如果有则优先从队列中获取线程加锁。
    - 非公平锁：只要判断锁是空闲的，则直接尝试CAS获取锁，而不会判断自己是否需要排队
4. 如果state >0则说明当前锁被线程持有
5. 判断是否当前线程持有该锁，如果是则对state累加1（通过这种方式实现锁重入）；
   [如果该锁不是当前线程持有，则把该线程加入到队列中等待。]
6. 如果该锁不是当前线程持有，判断等待队列是否为空，如果是空，则初始化之后再将自己插入，否则直接将自己插入队尾
7. 判读自己是否是head的下一个（.next）节点，如果是则tryAcquire(1)再次尝试获取锁，如果不是调用park()让自己睡眠。

> ReentrantLock解锁过程

0、调用unlock()方法（释放锁）
1、将status设置为0
2、设置持有当前锁的线程为null
3、唤醒队列里waitstatus不等于0的节点，调用unpark(解除阻塞线程)方法。

微服务框架

Spring

创建Spring IOC流程

> 框架

创建Spring Bean的生命周期

> 框架

Spring 的 AOP 是怎么实现的?

> 千峰

Spring 的 AOP 有哪几种创建代理的方式?

> 千峰

JDK 动态代理和 Cglib 代理的区别?

> 千峰

@PostConstruct修饰的方法/init-method里用到了其他 bean 实例，会有问题吗?

> 没有问题

1. PostConstruct注解在创建阶段触发
2. 属性的依赖注入是在 populateBean 方法里，属于属性填充阶段。
3. 属性填充阶段位于初始化之前，所以本题答案为==没有问题==。

要在 Spring IOC 容器构建完毕==之后==执行一些逻辑，怎么实现?

> 千峰

Spring怎么解决循环依赖的问题?

> 千峰

Spring 是通过提前暴露 bean 的引用来解决的，具体如下。

1. Spring首先使用构造函数创建一个“不完整”的bean实例（之所以说不完整，是因为此时该bean实例还未初始化），并且提前曝光该 bean 实例的ObjectFactory（提前曝光就是将ObjectFactory放到singletonFactories （三级）缓存）.
2. 通过 ObjectFactory 我们可以拿到该bean实例的引用，如果出现循环引用，我们可以通过缓存中的ObjectFactory来拿到bean实例，从而避免出现循环引用导致的死循环。

什么是事务

事务是数据库操作最基本单元，逻辑上一组操作，要么都成功，如果有一个失败所有操作都失败

事务四个特性（ACID）

1. 原子性
2. 隔离性
3. 持久性
4. 一致性

Spring 事务的实现原理?

> 千峰

Spring事务的底层实现主要使用的技术:AOP(动态代理)+ ThreadLocal + try/catch。

1、先做准备工作，解析各个方法上事务相关的属性，根据具体的属性来判断是否开始新事务
2、当需要开启的时候，获取数据库连接，关闭自动提交功能，开启事务。

3、 执行具体的sql逻辑操作
4、在操作过程中，如果执行失败了，那么会通过completeTransactionAfterThrowing来完成事务的回滚操作，回滚的具体逻辑是通过doRollBack方法来实现的，实现的时候也是要先获取连接对象，通过连接对象来回滚。
5、 如果执行过程中没有任何意外情况的发生，那么通过commitTransactionAfterReturing来完成事务的提交操作，提交的具体逻辑是通过doCommit方法来实现的，实现的时候也是要先获取连接对象，通过连接对象来提交。
6、当事务执行完毕之后需要清除相关的事务信息cleanupTransationInfo

Spring事务传播机制/行为

> 千峰

Spring事务隔离级别

> 千峰

Spring 的事务隔离级别是如何做到和数据库不一致的?

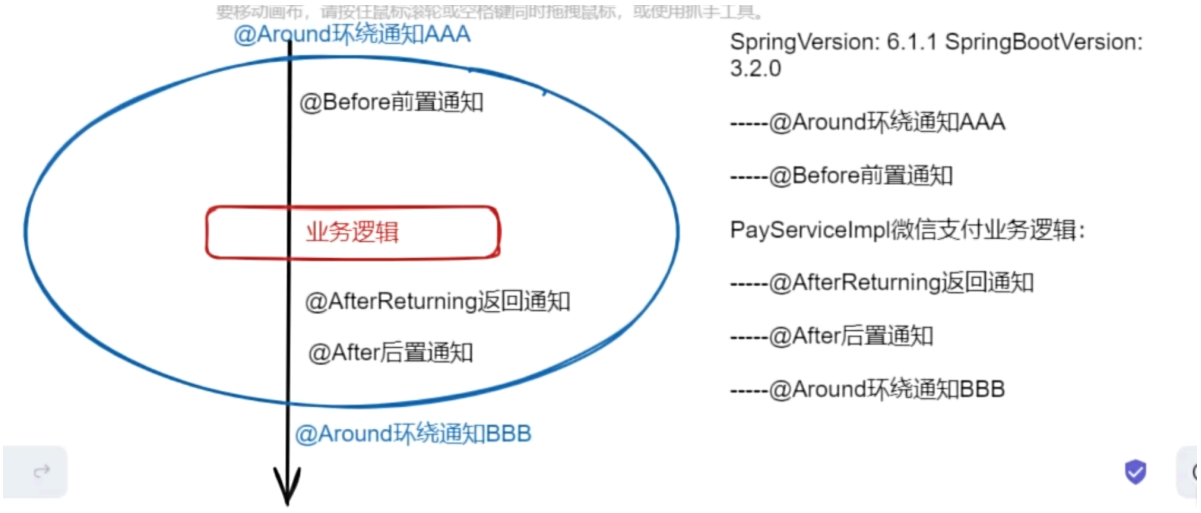
> 千峰

bean的自动装配

spring事务什么时候会失效?

Spring AOP 所有通知执行顺序

语雀



Spring Boot

SpringBoot 自动配置原理

@SpringBootApplication是一个组合注解，由 @ComponentScan、@EnableAutoConfiguration 和 @SpringBootConfiguration 组成

1. @SpringBootConfiguration 与普通 @Configuration 相比，唯一区别是前者要求整个 app 中只出现一次
2. @ComponentScan
    - excludeFilters - 用来在组件扫描时进行排除，也会排除自动配置类
3. @EnableAutoConfiguration 也是一个组合注解，由下面注解组成
    - @AutoConfigurationPackage – 用来记住扫描的起始包
    - @Import(AutoConfigurationImportSelector.class) 用来加载 `META-INF/spring.factories` 中的自动配置类

MyBatis

MyBatis工作原理

1. Mybatis 读取XML配置文件后会将内容放在一个Configuration类中，Configuration类会存在整个Mybatis生命周期，以便重复读取。
2. 通过Configuration中的数据生成并生成Executor和SqlSessionFactoryBuilder
3. SqlSessionFactoryBuilder会读取Configuration类中信息创建SqlSessionFactory（DefaultSqlSessionFactory）。
4. SqlSessionFactory（会话工厂）创建SqlSession（DefaultSqlSession）对象，该对象中包含了执行SQL语句的所有方法。
5. 调用getMapper方法（DefaultSqlSession.getMapper()）拿到MapperProxyFactory
6. 通过MapperProxyFactory拿到MapperProxy（MapperProxy：将Mapper接口的方法调用转换为对SqlSession的调用。）
7. SQL接口的执行逻辑交给Executor执行(Executor之前就已经通过配置生成好了)
8. Executor的执行逻辑交给StatementHandler执行
9. StatementHandler会根据SQL类型选择SimpleStatementHandler、PreparedStatementHandler、CallableStatementHandler三者之一去执行SQL，比如先执行PreparedStatementHandler
10. 通过StatementHandler处理数据前，通过ParameterHandler给SQL语句动态赋值
11. 继续通过StatementHandler处理SQL逻辑
12. SQL逻辑全部处理完后，通过ResultSetHandler生成结果集，返回数据给客户端

Spring Cloud

注解

- @EnableFeignClients+\@FeignClient(fallback 属性) 开启OpenFeign
- @EnableEurekaServer,@EnableEurekaClient Eureka注册中心
- @EnableDiscoveryClient （除Eureka注册中心）注册服务
- @EnableCircuitBreaker+\@HystrixCommand+\@DefaultProperties 开启Hystrix服务异常处理
- @EnableZuulProxy zuul网关
- @SentinelResource （Sentinel）服务熔断、降级、限流
- @GlobalTransactional Seata(注意：只需要在订单层加，库存、优惠卷不加)
- @EnableConfigServer+\@RefreshScope Bus(消息总线)

- @EnableBinding+\@StreamListener Stream(消息总线)

Eureka

> Eureka包含两个组件：Eureka Server和Eureka Client

1. Eureka Server提供服务注册服务
   各个微服务节点通过配置启动后，会在EurekaServer中进行注册，这样EurekaServer中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观看到。
2. EurekaClient通过注册中心进行访问
   是一个Java客户端，用于简化Eureka Server的交互，客户端同时也具备一个内置的、使用轮询(round-robin)负载算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳(默认周期为30秒)。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，EurekaServer将会从服务注册表中把这个服务节点移除（默认90秒）

Eureka自我保护机制

- 默认情况下，如果EurekaServer在一定时间内没有接收到某个微服务实例的心跳，EurekaServer将会注销该实例（默认90秒）。
- Eureka通过“自我保护模式”来解决这个问题——当EurekaServer节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。
- 在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再注销任何服务实例。

Ribbon

- Spring Cloud Ribbon是基于Netflix Ribbon实现的一套客户端　　负载均衡的工具。
- 客户端的软件负载均衡算法和服务调用。Ribbon客户端组件提供一系列完善的配置项如连接超时，重试等。

LB负载均衡(Load Balance)

- 简单的说就是将用户的请求平摊的分配到多个服务上，从而达到系统的HA（高可用）。
- 常见的负载均衡有软件Nginx，LVS，硬件 F5等。

负载均衡(轮询)算法原理

负载均衡算法：rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标 ，每次服务重启动后rest接口计数从1开始。

Ribbon本地负载均衡客户端 VS Nginx服务端负载均衡区别

- Nginx是服务器负载均衡，客户端所有请求都会交给nginx，然后由nginx实现转发请求。即负载均衡是由服务端实现的。
- Ribbon本地负载均衡，在调用微服务接口时候，会在注册中心上获取注册信息服务列表之后缓存到JVM本地，从而在本地实现RPC远程服务调用技术。

Feign与OpenFeign

Feign是一个声明式的Web服务客户端，让编写Web服务客户端变得非常容易，只需创建一个接口并在接口上添加注解即可

OpenFeign超时控制

- 默认Feign客户端只等待一秒钟，但是服务端处理需要超过1秒钟，导致Feign客户端不想等待了，直接返回报错。
- 为了避免这样的情况，有时候我们需要设置Feign客户端的超时控制。
- yml文件中开启配置

OpenFeign日志打印功能

对Feign接口的调用情况进行监控和输出

Hystrix断路器

Hystrix是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，==Hystrix能够保证在一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性==。

- 服务降级
  - 程序运行异常
  - 超时
  - 服务熔断触发服务降级
  - 线程池/信号量打满也会导致服务降级
- 服务熔断
- 服务限流

zuul路由网关

Zuul是Netflix出品的一个基于JVM路由和服务端的负载均衡器。

- 路由
- 过滤
- 负载均衡
- 灰度发布

Gateway新一代==网关==

- 反向代理
- 鉴权
- 流量控制
- 熔断
- 日志监控

为什么会出现Gateway

Gateway是基于异步非阻塞模型上进行开发的
基于Spring Framework 5, Project Reactor 和 Spring Boot 2.0 进行构建
Zuul 1.x，是一个基于阻塞 I/ O （基于Servlet 2. 5使用阻塞架构）的 API Gateway

Gateway工作流程

> 路由转发+执行过滤器链

- web请求，通过一些匹配条件，定位到真正的服务节点。并在这个转发过程的前后，进行一些精细化控制。
- predicate就是我们的匹配条件；
- 而filter，就可以理解为一个无所不能的拦截器。有了这两个元素，再加上目标uri，就可以实现一个具体的路由了

配置中心

分布式配置中心不用config+bus了，有nacos\zookeeper\Apollo

Stream 消息驱动

> 屏蔽底层消息中间件的差异,降低切换成本，统一消息的编程模型

- 应用程序通过 inputs 或者 outputs 来与 Spring Cloud Stream中==binder对象==交互。
- 通过我们配置来binding(绑定)，而 Spring Cloud Stream 的 binder对象负责与消息中间件交互。所以，我们只需要搞清楚如何与 Spring Cloud Stream 交互就可以方便使用消息驱动的方式。
- 发布-订阅、消费组、分区的三个核心概念。

工作原理图

Sleuth 请求链路跟踪

- Spring Cloud Sleuth提供了一套完整的服务跟踪的解决方案
- 在分布式系统中提供追踪解决方案并且兼容支持了zipkin

zipkin 链路跟踪原理

SkyWalking 请求链路跟踪

> https://blog.csdn.net/RenshenLi/article/details/123617432

Nacos服务注册和配置中心

> 一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。

各个注册中心对比

Seata处理分布式事务

Seata是一款开源的分布式事务解决方案，致力于在微服务架构下提供高性能和简单易用的分布式事务服务。

分布式事务

一次业务操作需要跨多个数据源或需要跨多个系统进行远程调用，就会产生分布式事务问题

Seata处理流程

- 简易版：

```
1. TM 向 TC 申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的 XID,XID在微服务调用链路的上下文中传播；
2. RM 向 TC 注册分支事务，将其纳入 XID 对应全局事务的管辖；
3. RM向TC汇报资源准备状态
4. TM通知TC提交或回滚事务（事务一阶段）
5. TC汇总事务信息，决定分布式事务是提交或回滚
6. TC通知所有RM提交或回滚资源（事务二阶段）
```

- 原理版：
  - 一阶段加载
  - 二阶段提交
  - 二阶段回滚

Spring Security

Spring Security工作原理

1. 用户点击登录链接,进入登录页登录。
    1. 登录请求被 Spring Security 的 Filters 链拦截。
    2. 调用 AuthenticationManager 进行认证,成功后将用户信息存储到 SecurityContext。
    3. 重定向到首页。
2. 用户访问需要权限的页面,请求被拦截。
    1. 检查当前请求是否已认证(从 SecurityContext 获取用户信息)。
    2. 如果已认证,检查当前用户是否有访问权限(==调用 AccessDecisionManager==)。
    3. 如果有权限,放行请求,继续调用链。
    4. 如果无权限,返回 403 响应。
3. 用户退出,清空 SecurityContext。
    1. 点击退出链接,Spring Security 清空 SecurityContext。
    2. 重定向到首页。

常用注解

- @EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)
- @Secured：具有某种角色才能访问
- @PreAuthorize：具有某种权限才可以访问这个路径
- @PostFilter：允许方法调用，但必须按照表达式来过滤方法的结果
- @PreFilter：允许方法调用，但必须在进入方法之前过滤输入值

Redis

Redis 在项目中的使用场景

- 缓存（核心）
- 分布式锁（set +lua脚本）
- 排行榜（zset）
- 计数（incrby）
- 消息队列（stream）
- 地理位置（geo）
- 访客统计（hyperloglog）等。

redis底层原理

> 单线程+基于非阻塞的IO多路复用模型

IO多路复用程序会==同时监听多个socket==，当被监听的socket准备好执行accept、read、write、close等操作时，与这些操作相对应的文件事件就会产生。
IO多路复用程序会把所有产生事件的socket压入一个队列中，然后有序地每次仅一个socket的方式传送给文件事件分派器，文件事件分派器接收到socket之后会根据socket产生的事件类型调用对应的事件处理器进行处理。
得到就绪状态后进行真正的操作可以在==同一个线==程里执行，也可以==启动其他线程==执行

Redis 常见的数据结构

string

> 字符串，最基础的数据类型

底层数据结构

String的数据结构为简单动态字符串。是可以修改的字符串，内部结构实现上类似于Java的ArrayList，采用==预分配冗余空间的方式来减少内存的频繁分配==。

List

> 列表

底层数据结构

Redis将链表和ziplist结合起来组成了quicklist。也就是将多个ziplist使用双向指针串起来形成一个==双向链表==使用。这样既满足了快速的插入删除性能，又不会出现太大的空间冗余。

Hash

> 哈希对象：键值对集合

底层数据结构

当field-value长度较短且个数较少时，使用==ziplist==，否则使用==hashtable==。

Set

> 集合：无序、无重复数据的列表

底层数据结构

==Set数据结构是用哈希表实现字典的。==
Java中HashSet的内部实现使用的是HashMap，只不过所有的value都指向同一个对象。Redis的set结构也是一样，它的内部也使用hash结构，所有的value都指向同一个内部值。

Sorted Set

> 集合：有序集合，Set 的基础上加了个分值（score）。

底层数据结构

zset底层使用了两个数据结构:

- hash，hash的作用就是关联==元素value和权重score==，保障元素value的唯一性，可以通过元素value找到相应的score值。
- 跳跃表，跳跃表的目的在于给==元素value排序==，根据score的范围获取元素列表。

Sorted Set 为什么同时使用字典和跳跃表?

Sorted Set 为什么使用跳跃表，而不是红黑树?

HyperLogLog

> 通常用于基数统计

Geo

> 可以将用户给定的地理位置信息储存起来， 并对这些信息进行操作：获取2个位置的距离、根据给定地理位置坐标获取指定范围内的地理位置集合。

Bitmap

> 位图

Stream

> 主要用于消息队列，类似于 kafka，可以认为是 pub/sub 的改进版。提供了消息的持久化和主备复制功能，可以让任何客户端访问任何时刻的数据，并且能记住每一个客户端的访问位置，还能保证消息不丢失。

RDB 和 AOF 机制（持久化机制）

RDB

在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

优点

1. 整个Redis数据库将只包含一个rdb文件，方便持久化。
2. 容灾性好，方便备份。
3. 性能最大化，fork子进程来完成写操作，让主进程继续处理命令，所以是 IO最大化。使用单独子进程来进行持久化，主进程不会进行任何IO 操作，保证了 redis 的高性能
4. 相对于数据集大时，比 AOF 的启动效率更高。4

缺点

1. 数据安全性低。RDB是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候
2. 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

AOF

以日志的形式记录服务器所处理的每一个写、删除操作（查询操作不会记录），以文本的方式且是增量记录，可以打开文件看到详细的操作记录

优点

1. 数据安全，Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。
    - 事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。
    - 而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中
2. 通过 append 模式写文件，即使中途服务器宕机也不会破坏已经存在的内容，可以通过redis-check-aof工具解决数据一致性问题。
3. AOF 机制的rewrite模式。定期对AOF文件进行重写，以达到压缩的目的

缺点

1. AOF 文件比 RDB 文件大，且恢复速度慢。
2. 数据集大的时候，比 rdb 启动效率低。
3. 运行效率没有RDB高

总结

- AOF文件比RDB更新频率高，优先使用AOF还原数据。
- AOF比RDB更安全也更大
- RDB性能比AOF好
- 如果两个都配了优先加载AOF

Redis的过期键的删除策略(redis过期策略)

> Redis中同时使用了惰性过期和定期过期两种过期策略。

- **惰性过期**：==只有当访问一个key时，才会判断该key是否已过期，过期则清除==。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
- **定期过期**:==每隔一定的时间，会扫描一定数量的数据库的字典中一定数量的key，并清除其中已过期的key==。该策略是一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

Redis淘汰策略

- **noeviction**：不淘汰任何数据，当内存满时，新的写入操作会报错。
- **volatile-lru**：淘汰设置了过期时间的数据中，最近最少使用的数据。这种策略适用于缓存数据，可以保证缓存中的数据都是最近使用过的。
- **volatile-ttl**：淘汰设置了过期时间的数据中，距离过期时间最近的数据。这种策略适用于一些临时性的数据，可以保证数据不会过期。
- **volatile-random**：淘汰设置了过期时间的数据中，随机选择一个数据进行淘汰。
- **allkeys-lru**：淘汰所有数据中，最近最少使用的数据。这种策略适用于缓存数据和持久化数据混合使用的情况。
- **allkeys-random**：淘汰所有数据中，随机选择一个数据进行淘汰。

```
对于一些重要的数据，可以采用 noeviction 策略，以保证数据的完整性；
对于一些缓存数据，可以采用 volatile-lru 策略，以保证缓存的有效性。
```

Redis实现分布式锁

> 单机set命令

千峰

> 多机(集群)实现的分布式锁RedLock+Redisson

如果++线程一在Redis的master节点上拿到了锁++，但是加锁的key还没同步到slave节点。恰好这时，master节点发生故障，一个slave节点就会升级为master节点。++线程二就可以获取同个key的锁++啦，但线程一也已经拿到锁了，==锁的安全性就没了==。
为了解决这个问题，Redis作者 antirez提出一种高级的分布式锁算法：Redlock。

> 多个Redis master部署，以保证它们不会同时宕掉。
> 并且这些master节点是完全相互独立的，相互之间不存在数据同步。
> 同时，需要确保在这多个master实例上，是与在Redis单实例，使用相同方法来获取和释放锁。

我们假设当前有5个Redis master节点，在5台服务器上面运行这些Redis实例。

RedLock的实现步骤:
1.获取当前时间，以毫秒为单位。
2.按顺序向5个master节点请求加锁。客户端设置锁失效时间和响应超时时间，并且超时时间要小于锁的失效时间。（假设锁自动失效时间为10秒，则超时时间一般在5-50毫秒之间,我们就假设超时时间是50ms吧）。如果超时，跳过该master节点，尽快去尝试下一个master节点。
3.客户端使用当前时间减去开始获取锁时间（即步骤1记录的时间），得到获取锁使用的时间。当且仅当超过一半（N/2+1）的Redis master节点都获得锁，并且使用的时间小于锁失效时间时，锁才算获取成功。（如上图，10s> 30ms+40ms+50ms+4m0s+50ms）
4.如果取到了锁，key的真正有效时间就变啦，需要减去获取锁所使用的时间。
5.如果获取锁失败（没有在至少N/2+1个master实例取到锁，有或者获取锁时间已经超过了有效时间），客户端要在所有的master节点上解锁（即便有些master节点根本就没有加锁成功，也需要解锁，以防止有些漏网之鱼）。

> 简化下步骤就是：
>
> 按顺序向5个master节点请求加锁
> 根据设置的超时时间来判断，是不是要跳过该master节点。
> 如果大于等于3个节点加锁成功，并且使用的时间小于锁的有效期，即可认定加锁成功啦。
> 如果获取锁失败，解锁！
>
> 存在的问题：
> 1.严重依赖系统时钟。如果线程1从3个实例获取到了锁，但是这3个实例中的某个实例的系统时间走的稍微快一点，则它持有的锁会提前过期被释放，当他释放后，此时又有3个实例是空闲的，
> 则线程2也可以获取到锁，则可能出现两个线程同时持有锁了
> 2.如果线程1从3个实例获取到了锁，但是万一其中有1台重启了，则此时又有3个实例是空闲的，
> 则线程2也可以获取到锁，此时又出现两个线程同时持有锁了

单机set命令实现分布式的问题

Redis 分布式锁过期了，还没处理完怎么办?

千峰

- 守护线程“续命”：额外起一个线程，定期检查线程是否还持有锁，如果有则延长过期时间。

Redisson 里面就实现了这个方案，使用“看门狗”定期检查（每1/3的锁时间检查1次），如果线程还持有锁，则刷新过期时间。

- 超时回滚：当我们解锁时发现锁已经被其他线程获取了，说明此时我们执行的操作已经是“不安全”的了，此时需要进行回滚，并返回失败。

守护线程续命的方案有什么问题吗?

千峰

主从复制、哨兵机制、集群模式

主从复制

通过执行slaveof命令或设置slaveof选项，让一个服务器去复制另一个服务器的数据。主数据库可以进行读写操作，当写操作导致数据变化时会自动将数据同步给从数据库。
而从数据库一般是只读的，并接受主数据库同步过来的数据。一个主数据库可以拥有多个从数据库，而一个从数据库只能拥有一个主数据库。

全量复制：

1. ++主节点通过（bgsave命令）fork子进程进行RDB持久化++，该过程是非常消耗CPU、内存(页表复制)、硬盘IO的
    - 主节点需要进行RDB持久化
2. ++主节点通过网络将RDB文件发送给从节点++，对主从节点的带宽都会带来很大的消耗
3. ++从节点清空老数据、载入新RDB文件++，此过程是阻塞的，无法响应客户端的命令

部分复制：

1. 复制偏移量：执行复制的双方，++主从节点，分别会维护一个复制偏移量offset++
2. 复制积压缓冲区：主节点内部维护了一个固定长度的、先进先出(FIFO)队列 作为复制积压缓冲区，当主从节点offset的差距过大超过缓冲区长度时，将无法执行部分复制，只能执行全量复制。
3. 服务器运行ID(runid)：每个Redis节点，都有其运行ID，运行ID由节点在启动时自动生成，++主节点会将自己的运行ID发送给从节点，从节点会将主节点的运行ID存起来。从节点Redis断开重连的时候，就是根据运行ID来判断同步的进度++：
    - 如果从节点保存的运行ID与主节点现在的运行ID相同，说明主从节点之前同步过，主节点会继续尝试使用部分复制(到底能不能部分复制还要看offset和复制积压缓冲区的情况)
    - 如果从节点保存的运行ID与主节点现在的运行ID不同，说明从节点在断线前同步的Redis节点并不是当前的主节点，只能进行全量复制。

哨兵机制

哨兵（Sentinel） 是Redis的高可用性解决方案：由一个或多个 Sentinel实例组成的Sentinel系统可以监视任意多个主服务器，以及这些主服务器属下的所有从服务器。
Sentinel可以在被监视的主服务器进入下线状态时，自动将下线主服务器的某个从服务器升级为新的主服务器，然后由新的主服务器代替已下线的主服务器继续处理命令请求。

当哨兵监测到某个主节点客观下线之后，就会开始故障转移流程。核心流程如下：

1. 发起一次选举，选举出领头Sentinel
2. 领头Sentinel在已下线主服务器的所有从服务器里面，挑选出一个从服务器，并将其升级为新的主服务器。
3. 领头 Sentinel将剩余的所有从服务器改为复制新的主服务器。
4. 领头 Sentinel更新相关配置信息，当这个旧的主服务器重新上线时，将其设置为新的主服务器的从服务器。

Redis脑裂问题

- 配置连接到master(主服务器)的最少slave数量，以及他的延迟时间
- 这样一来发生脑裂旧的主服务器就会拒绝写响应，由新的主服务器处理，这样==减少数据的丢失==

集群模式

> 为了解决哨兵机制将所有数据放在一台服务器上

集群模式具备的特点如下：

- 采取去中心化的集群模式，将数据按槽存储分布在多个 Redis 节点上，每个节点负责处理部分槽
- 使用 CRC16 算法来计算 key 所属的槽
- 所有的 Redis 节点彼此互联，通过PING-PONG机制来进行节点间的心跳检测。
- 分片内采用一主多从保证高可用，并提供复制和故障恢复功能。
- ==客户端与Redis节点直连==，不需要中间代理层（proxy）。==客户端不需要连接集群所有节点==，==连接集群中任何一个可用节点即可==。

缓存穿透、缓存击穿、缓存雪崩

1. 缓存雪崩：如果缓存中某一时刻大批热点数据同时过期，那么就可能导致大量请求直接访问Mysql，解决办法就是在过期时间上增加一点随机值，另外如果搭建一个高可用的Redis集群也是防止缓存雪崩的有效手段
2. 缓存击穿：和缓存雪崩类似，缓存雪崩是大批热点数据失效，而缓存击穿是指某一个热点key突然失效，也导致了大量请求直接访问Mysql数据库，这就是缓存击穿，解决方案就是考虑这个热点key不设过期时间或者设置较长的过期时间
3. 缓存穿透：假如某一时刻访问redis的大量key都在redis中不存在（比如黑客故意伪造一些乱七八糟的key），那么也会给数据库造成压力，这就是缓存穿透，解决方案是使用布隆过滤器，它的作用就是如果它认为一个key不存在，那么这个key就肯定不存在，所以可以在缓存之前加一层布隆过滤器来拦截不存在的key。

如何保证数据库和缓存的数据一致性?

> Redis和MySQL双写一致

采用分布式事务(不推荐)

> 2PC、TCC、MQ事务消息
> 引入分布式事务必然会带来性能上的影响，这与我们当初引入缓存来提升性能的目的是相违背的

先更新数据库，再删除缓存

> https://www.jianshu.com/p/1a6abe6eed74

1. 更新数据库数据
2. 数据库会将操作信息写入binlog日志当中
3. 订阅程序提取出所需要的数据以及key（canal）
4. 另起一段非业务代码，获得该信息
5. 尝试删除缓存操作，发现删除失败
6. 将这些信息发送至消息队列

7. 重新从消息队列中获得该数据，重试操作。
   上述的订阅binlog程序在mysql中有现成的中间件叫canal，可以完成订阅binlog日志的功能。

Redis6.0 为什么要引入多线程?

- Redis 将所有数据放在内存中，内存的响应时长大约为 100 纳秒
- 但随着越来越复杂的业务场景，有些公司动不动就上亿的交易量，因此需要更大的 QPS。(处理QPS是多线程，但是处理命令还是单线程)
- 为什么不对数据进行分区并采用多个服务器
  - 缺点
    - 管理的 Redis 服务器太多，维护代价大；
    - 某些适用于单个Redis服务器的命令不适用于数据分区；
    - 数据分区无法解决热点读/写问题；
    - 数据偏斜，重新分配和放大/缩小变得更加复杂等等

怎么理解 Redis 中事务?

- 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==。不过 Redis 的是弱事务。
- 事务是 Redis 实现在服务器端的行为，**用户执行 MULTI 命令时，服务器会将对应这个用户的客户端对象设置为一个特殊的状态，在这个状态下后续用户执行的查询命令不会被真的执行，而是被服务器缓存起来，直到用户执行 EXEC 命令为止，服务器会将这个用户对应的客户端对象中缓存的命令按照提交的顺序依次执行。**
- Redis 提供了简单的事务，之所以说它简单，==主要是因为它不支持事务中的回滚特性==, 同时无法实现命令之间的逻辑关系计算，当然也体现了 Redis 的“keep it simple”的特性

Redis 如何解决 key 冲突?

1. 业务隔离：不同的业务场景的key尽量不同/业务不同的key放到不同的机器上
2. key的设计：biz-pay-id
3. 多并发情况下，使用分布式锁
4. 遇到 hash 冲突采用链表进行处理

怎么提高缓存命中率?

1. 数据通过缓存预加载（预热）
2. 增加缓存的存储空间，提高缓存的数据、提高命中率
3. 调正缓存的存储类型
4. 提升缓存更新频次

Redis 集群会有写操作丢失吗？为什么?

> redis集群不能确保数据的强一致性

1. 过期 key 被清理
2. 最大内存不足，导致 Redis 自动清理部分 key 以节省空间

3. 主库故障后自动重启，从库自动同步（主库重启过程中所有写请求丢失）
4. 单独的主备方案，网络不稳定触发哨兵的自动切换主从节点，切换期间会有数据丢失

Redis 常见性能问题和解决方案有哪些?

1. 持久化（RDB\AOF）
    - 主从架构中，主库不要做持久化，从节点做持久化
2. 保证主从复制流畅，最好在同一个局域网内
3. 主从复制不要采用网状结构
4. 主从复制尽量避免主库压力很大，再次增加从库

分布式

服务治理、服务注册与发现

- 服务治理：在传统的rpc远程调用框架中，管理每个服务与服务之间依赖关系比较复杂，管理比较复杂，所以需要使用服务治理，管理服务于服务之间依赖关系，可以实现服务调用、负载均衡、容错等，实现服务发现与注册。
- 服务注册与发现：当服务器启动的时候，会把当前自己服务器的信息 比如服务地址通讯地址等以别名方式注册到注册中心上。另一方（消费者|服务提供者），以该别名的方式去注册中心上获取到实际的服务通讯地址，然后再实现本地RPC调用

CAP理论

C（Consistency）表示强一致性，
A（Availability）表示可用性
P（Partition Tolerance）表示分区容错性
分布式系统要么保证CP，要么保证AP，无法同时保证CAP。

BASE理论

BASE是Basically Available（==基本可用==）、Soft state(==软状态==)和 Eventually consistent（==最终一致性==） 三个短语的缩写。BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结， 是==基于CAP定理逐步演化而来的==。BASE理论的核心思想是：==即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。==

基本可用

- 基本可用是指分布式系统在出现不可预知故障的时候，==允许损失部分可用性==（注意，这绝不等价于系统不可用）。
    - 响应时间上的损失
    - 系统功能上的损失

软状态

软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即==允许系统在不同节点的数据副本之间进行数据同步的过程存在延时==

最终一致性

最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。==因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。==

分布式id生成方案

> 分布式ID

uuid

优点：代码简单，性能好;保证唯一
缺点：

- 每次生成的ID都是无序的，而且不是全数字，且无法保证趋势递增。
- UUID生成的是字符串，字符串存储性能差，查询效率慢
- UUID长度过长，不适用于存储，耗费数据库性能。
- ID无一定业务含义，可读性差。
- 有信息安全问题，有可能泄露mac地址

数据库自增序列

单机模式：

优点：

- 实现简单，依靠数据库即可，成本小。
- ID数字化，单调自增，满足数据库存储和查询性能。
- 具有一定的业务可读性。（结合业务code）
  缺点：
- 强依赖DB，存在单点问题，如果数据库宕机，则业务不可用。
- DB生成ID性能有限，单点数据库压力大，无法扛高并发场景。
- 信息安全问题，比如暴露订单量，url查询改一下id查到别人的订单

基于redis、mongodb、zk等中间件生成

> redis:redis+lua脚本，控制自增范围

优点：

- 不依赖于数据库，灵活方便，且性能优于数据库。
- 数字ID天然排序，对分页或者需要排序的结果很有帮助。
  缺点：
- 如果系统中没有Redis，还需要引入新的组件，增加系统复杂度。
- 需要编码和配置的工作量比较大。

==Leaf-segment（美团的Leaf算法）==

采用每次获取一个ID区间段的方式来解决，区间段用完之后再去数据库获取新的号段，这样一来可以大大减轻数据库的压力

雪花算法

- 缺点：
  - 依赖服务器时间，服务器时钟回拨时可能会生成重复 id。

分布式事务解决方案

- [x] 图灵

> 2PC:两阶段提交
> 3PC:三阶段提交
> TCC（补偿事务）:Try、Confirm、Cancel

ElasticSearch

> Elasticsearch 是基于 Lucene 的 Restful 的分布式实时全文搜索引擎，每个字段都被索引并可被搜索，可以快速存储、搜索、分析海量的数据。

倒排索引

> 文档内容中的单词作为索引，将包含该词的文档 ID 作为记录的结构。

原理

1. 首先要对搜索的数据进行分词(将一段连续的文本按照语义拆分为多个单词)
2. .然后按照分解的单词来作为索引，对应的文档 id 建立一个链表，就能构成倒排索引结构
3. 整个搜索过程中我们不需要做任何文本的模糊匹配。
4.

底层数据结构

Lucene 的倒排索，增加了最左边的一层「字典树」term index，它不存储所有的单词，只存储单词前缀，通过字典树找到单词所在的块，也就是单词的大概位置，再在块里二分查找，找到对应的单词，再找到单词对应的文档列表。

Lucene 还用了 FST（Finite State Transducers 有穷状态转换器）对它进一步压缩。

Zookeeper

为什么Zookeeper可以用来作为注册中心

可以利用Zookeeper的临时节点和watch机制来实现注册中心的自动注册和发现，另外Zookeeper中的数据都是存在内存中的，并且Zookeeper底层采用了nio，多线程模型，所以Zookeeper的性能也是比较高的，所以可以用来作为注册中心，但是如果考虑到注册中心应该是注册可用性的话，那么Zookeeper则不太合适，因为Zookeeper是CP的，它注重的是一致性，所以集群数据不一致时，集群将不可用，所以用Redis、Eureka、Nacos来作为注册中心将更合适。

Zookeeper可以用来作为配置中心、集群管理

> 图灵210道：简述zk的命名服务、配置管理、集群管理

ZAB协议

图灵

简述Paxos算法

Paxos算法解决的是一个分布式系统如何就某个值（决议）达成一致,在Paxos算法中，有三种角色：
Proposer （提议者),Acceptor（接受者），Learners（记录员）
Proposer提议者:只要Proposer发的提案Propose被半数以上的Acceptor接受，
Proposer就认为该提案例的value被选定了。
Acceptor接受者：只要Acceptor接受了某个提案，Acceptor就认为该提案例的value被选定了Learner记录员： Acceptor告诉Learner哪个value被选定，Learner就认为哪个value被选定。

1.
a.Proposer(提议者)收到client请求或者发现本地有未提交的值，选择一个提案编号N，
然后向半数以上的Acceptor（接受者）发送编号为N的Prepare请求。
b.Acceptor（接受者）收到一个编号为 N的 Prepare请求
b1.本节点已经有已提交的value记录，对比记录的编号和N，大于N则拒绝回应，否则返回该记录value及编号
b2.没有已提交记录，判断本地是否有编号N1，N1>N、则拒绝响应，否则将N1改为N（如果没有N1，则记录N），
并响应prepare
2.
如果Proposer(提议者)收到半数以上Acceptor（接受者）对其发出的编号为N的 Prepare请求的响应，那么它就会发送一个针对[N,V]提案的
Accept请求给半数以上的Acceptor（接受者）。V就是收到的响应中编号最大的value
，如果响应中不包含任何value，那么V就由Proposer自己决定。

活锁*: accept（接受）时被拒绝，加大N，重新accept（接受），此时另外一个proposer也进行相同操作，导致accep（接受）失败，无法完成算法
multi-paxos：
区别于paxos只是确定一个值，multi-paxos可以确定多个值，
收到accept请求后，则一定时间内不再accept其他节点的请求，以此保证后续的编号不需要在经过preprae确认，直接进行accept操作。此时该节点成为了leader，直到accept被拒绝，重新发起prepare请求竞争leader资格

领导者选举的流程是怎样的

什么时候发生选举?
（1）. 服务器初始化启动

（2）.服务器运行期间无法与Leader保持连接

Zookeeper集群中节点之间数据是如何同步的

Zookeeper集群重的角色
1.Leader：
1.1 发起与提交写请求。
1.2 崩溃恢复时负责恢复数据以及同步数据到Learnr
2.Follower
2.1 与老大Leader保持心跳连接
2.2 当Leader挂了的时候，经过投票后成为新的leader。leader的重新选举是Follower们内部投票决定的。
2.3 向leader发送消息与请求
2.4 处理leader发来的消息与请求
3.observer：
3.1 Observer的主要作用是提高zookeeper集群的读性能
3.2 与leader同步数据
3.3 不参与leader选举，没有投票权。也不参与写操作的提议过程。

讲下Zookeeper watch机制

图灵

zk实现分布式锁

> **Zookeeper 是基于==临时顺序节点==以及 ==Watcher 监听器机制==实现分布式锁的**

1. 一把分布式锁通常使用一个 Znode 节点表示；如果==锁对应的 Znode 节点不存在，首先创建 Znode 节点==。这里假设为 /test/lock，代表了一把需要创建的分布式锁。
2. ==抢占锁的所有客户端==，使用锁的 Znode 节点的子节点列表来表示；如果某个客户端需要占用锁，==则在 /test/lock(Znode节点) 下创建一个临时顺序的子节点==。比如，如果子节点的前缀为 /test/lock/seq-，则第一次抢锁对应的子节点为 /test/lock/seq-000000001，第二次抢锁对应的子节点为 /test/lock/seq-000000002，以此类推。
3. 当客户端创建子节点后，需要进行判断：自己创建的子节点，是否为当前==子节点列表中序号最小的子节点==。如果是，则加锁成功；如果不是，则监听前一个 Znode 子节点变更消息，等待前一个节点释放锁。
4. 一旦队列中的后面的节点，获得前一个子节点变更通知，则开始进行判断，判断自己是否为当前子节点列表中序号最小的子节点，如果是，则认为加锁成功；如果不是，则持续监听，一直到获得锁。
5. 获取锁后，开始处理业务流程。完成业务流程后，删除自己的对应的子节点，完成释放锁的工作，以方面后继节点能捕获到节点变更通知，获得分布式锁。

zk的数据模型和节点类型（了解）

图灵

消息队列

RabbitMQ

RabbitMQ架构设计/工作原理

Broker：rabbitmq的服务节点
Queue：队列
Exchange：交换器
Channel：信道。信道是建立在Connection 之上的虚拟连接

生产者发送消息的流程

1. 生产者连接 RabbitMQ ， 建立 TCP 连接 (Connection) ， 开启信道/通道 （ Channel ）
2. 生产者声明一个Exchange（交换器），并设置相关属性，比如交换器类型、是否持久化等
3. 生产者声明一个队列并设置相关属性，比如是否排他、是否持久化、是否自动删除等
4. 生产者通过 bindingKey（绑定Key）将交换器和队列绑定 （ binding ）起来
5. 生产者发送消息至Broker，其中包含routingKey（路由键）、交换器等信息
6. 相应的交换器根据接收到的routingKey查找相匹配的队列。
7. 如果找到，则将从生产者发送过来的消息存入相应的队列中。 如果没有找到，则根据生产者配置的属性选择丢弃还是回退给生产者
8. 关闭信道。
9. 关闭连接

消费者获取消息的过程

1. 消费者连接到RabbitMQ Broker，建立一个连接(Connection)，开启一个信道(Channel) 。
2. 消费者向Broker请求消费相应队列中的消息，可能会设置相应的回调函数，以及 做一些准备工作
3. 等待Broker回应并投递相应队列中的消息，消费者接收消息。
4. 消费者确认 (ack) 接收到的消息。
5. RabbitMQ 从队列中删除相应已经被确认的消息。
6. 关闭信道。
7. 关闭连接。

Kafka

Kafka架构设计/工作原理

Topic(主题)：可以理解为一个队列，Topic将消息分类，生产者和消费者面向的是同一个 Topic。
Partition（分区）：一个Topic以多个Partition的方式分布到多个Broker上，每个 Partition 是一个 有序的队列。

消费组:考虑到多个消费者的场景，kafka在设计的时候，可以由多个消费者组成一个消费组，同一个消费组者的消费者可以消费同一个 topic下不同分区的数据，同一个分区只会被一个消费组内的某个消费者所消费，防止出现==重复消费==的问题。==但是不同的组，可以消费同一个分区的数据==！

发送消息

消费数据

与生产者一样，消费者主动的去kafka集群拉取消息时，也是从Leader分区去拉取数据。

Kafka的Pull和Push分别有什么优缺点

RocketMQ架构设计/工作原理

- NameServer：NameServer是一个非常简单的Topic路由注册中心，支持Broker的动态注册与发现。主要包括两个功能：==Broker管理和路由信息管理==
- BrokerServer：Broker 主要负责消息的存储、投递和查询以及服务高可用保证。消息存储包含 CommitLog、ConsumeQueue、Index 这三种文件的存储，还需要负责==管理客户端（Producer/Consumer）==和==维护 Consumer 的 Topic 订阅信息==。还提供高可用服务，==对 Master Broker 和 Slave Broker 进行消息的同步==，所以 Broker Server 是 RocketMQ 的核心组件。
- Producer：消息生产者，往 Broker ==发送指定 Topic 的消息==，可以==同步、异步、oneway 的方式==进行发送。
- Consumer：消息消费者，==支持 Push、Pull 两种消息消费的模式==，并且==支持集群方式和广播方式进行消费==，提供==实时消息订阅机制==。

RocketMQ工作流程

RocketMQ的事务消息是如何实现的

消息队列如何保证消息可靠传输

消息可靠传输代表了两层意思，既不能多也不能少。

1. 为了保证消息不多，也就是消息不能重复，也就是生产者不能重复生产消息，或者消费者不能重复消费消息
    1. 首先要确保消息不多发，这个不常出现，也比较难控制，因为如果出现了多发，很大的原因是生产者自己的原因，如果要避免出现问题，就需要在消费端做控制
    2. 要避免不重复消费，最保险的机制就是消费者实现幂等性，保证就算重复消费，也不会有问题，通过幂等性，也能解决生产者重复发送消息的问题
2. 消息不能少，意思就是消息不能丢失，生产者发送的消息，消费者一定要能消费到，对于这个问题，就要考虑两个方面

1. 生产者发送消息时，要确认broker确实收到==并持久化==了这条消息，比如==RabbitMQ的confirm机制==，
   ==Kafka、RocketMQ的ack机制==都可以保证生产者能正确的将消息发送给broker
   1.1 如果消息发送失败进行重试，超过重试次数后，则要持久化磁盘，由补偿服务来进行扫描
2. broker要等待消费者真正确认消费到了消息时才删除掉消息，这里通常就是消费端ack机制，消费
   者接收到一条消息后，如果确认没问题了，就可以给broker发送一个ack，broker接收到ack后才会
   删除消息

如何保证消息幂等性

> 解决消费者重复消费消息的问题，所有的MQ产品都没有这个机制，需要消费端做控制

1. 使用Redis缓存

   让每个消息携带一个全局的唯一ID，即可保证消息的幂等性，具体消费过程为：

   > 全局唯一的ID(用==分布式ID:美团的Leaf算法==)放到set集合

   1. 消费者获取到消息后先根据id去查询redis/db（set集合）是否存在该消息
   2. 如果不存在，则正常消费，消费完毕后写入redis/db
   3. 如果存在，则证明消息被消费过，直接丢弃
2. 使用乐观锁
   1. 先查询数据库，判断数据是否已经被更新过。如果是，则直接返回消费完成，否则执行消费。
   2. 如果不是，则更新数据。更新数据库时，带上数据的状态。如果更新失败，则直接返回消费完成，否则执行消费。

场景题

redis分布式锁的应用场景

1. 系统是一个分布式系统（集群），java锁已经锁不住了
2. ==操做共享资源，比如说多个服务（进程）操作数据库同一条记录或者用户资源==
3. 同步访问，即多个进程同时操作共享资源

1. 线上几百万条数据积压如何处理?

2. 超时未支付解决方案

> RabbitMQ方案

```
1. 在生成订单时(表中记录订单的创建时间),发送一个消息到延时消息队列（默认30分钟），消息为订单号

2.
    2.1 支付成功消费延时消息队列中的消息，此订单改为已支付。
    2.2 如果超时未消费则会进入死信队列，代表未支付。（监听并）消费死信队列消息（配置固定的死信队列：x-dead-letter-exchange和x-dead-letter-routing-key）
    ，删除数据库中未付款的订单，修改商品的库存
```

> RocketMQ、kafaka方案

1.请求订单服务下单，下单成功后，发送`orderId`等信息到延迟队列
2.延迟时间到，消费端接到消息
3.调用订单服务查询订单状态，如果订单为未支付，则变更为取消状态（分布式事务保持一致性：订单状态、库存等一致性）

3. 1000个数据按顺序执行复杂计算，要求过程中重启后数据还能安全继续执行

1. 将计算完的数据的状态设置为已完成
2. 如果未完成且停止计算就回滚该条数据
3. 重启后从这条数据开始计算

4. MySQL分表、分库、分区

- 分片
- 分表
- 分区
- 分表 VS 分区
- 分库

如何实现亿级的数据统计

- 使用redis

618的排行榜是如何实现的

- mysql
- 借助大数据
- 使用redis

如何从1000w条数据中找出最热门的10个记录

- 背景
- 解决方案