> 丰巢科技

1.bio与nio的区别

1、bio同步阻塞io：在此种方式下，用户进程在发起一个IO操作以后，必须等待IO操作的完成，只有当真正完成了IO操作以后，用户进程才能运行。JAVA传统的IO模型属于此种方式！

2、nio同步非阻塞式I/O；java NIO采用了双向通道进行数据传输，在通道上我们可以注册我们感兴趣的事件：连接事件、读写事件；NIO主要有三大核心部分：Channel(通道)，Buffer(缓冲区), Selector。传统IO基于字节流和字符流进行操作，而NIO基于Channel和Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。

1. BIO （Blocking I/O）：同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。这里使用那个经典的烧开水例子，这里假设一个烧开水的场景，有一排水壶在烧开水，BIO的工作模式就是， 叫一个线程停留在一个水壶那，直到这个水壶烧开，才去处理下一个水壶。但是实际上线程在等待水壶烧开的时间段什么都没有做。

2. NIO （New I/O）：同时支持阻塞与非阻塞模式，但这里我们以其同步非阻塞I/O模式来说明，那么什么叫做同步非阻塞？如果还拿烧开水来说，NIO的做法是叫一个线程不断的轮询每个水壶的状态，看看是否有水壶的状态发生了改变，从而进行下一步的操作。

3. AIO （ Asynchronous I/O）：异步非阻塞I/O模型。异步非阻塞与同步非阻塞的区别在哪里？异步非阻塞无需一个线程去轮询所有IO操作的状态改变，在相应的状态改变后，系统会通知对应的线程来处理。对应到烧开水中就是，为每个水壶上面装了一个开关，水烧开之后，水壶会自动通知我水烧开了。

2.select与poll的区别

1、io多路复用：

1、概念：IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。

2、优势：与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

3、系统：目前支持I/O多路复用的系统调用有 select，pselect，poll，epoll。

2、select：select目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。select的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在Linux上一般为1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。

3、poll：它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：

a. 大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。

b. poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。

> epoll跟select都能提供多路I/O复用的解决方案。在现在的Linux内核里有都能够支持，其中epoll是Linux所特有，而select则应该是POSIX所规定，一般操作系统均有实现。

3.zookeeper的工作原理

1、定义：zookeeper是一种为分布式应用所设计的高可用、高性能且一致的开源协调服务，它提供了一项基本服务：分布式锁服务。后来摸索出了其他使用方法：配置维护、组服务、分布式消息队列、分布式通知/协调等。

2、特点：

1、能够用在大型分布式系统中；

2、具有一致性、可用性、容错性，不会因为一个节点的错误而崩溃；

3、用途：用户大型分布式系统，作协调服务角色；

1、分布式锁应用：通过对集群进行master选举，来解决分布式系统中的单点故障（一主n从，主挂全挂）。

2、协调服务；

3、注册中心；

4、原理：

术语：

数据结构Znode：zookeeper数据采用树形层次结构，和标准文件系统非常相似，树中每个节点被称为Znode；

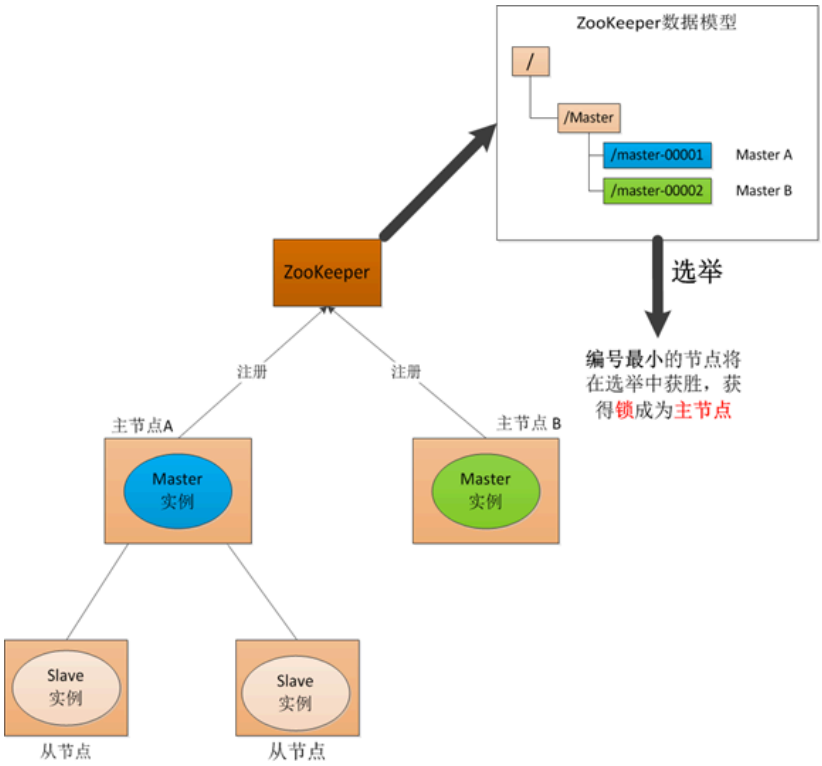
通知机制Watcher：zookeeper可以为所有的读操作（exists()、getChilden()及getData()）设置watch，watch事件是一次性出发器，当watch的对象状态发生改变时，将会触发次对象上watch所对应的事件。watch事件将被异步的发送给客户端，并且zookeeper为watch机制提供了有序的一致性保证。

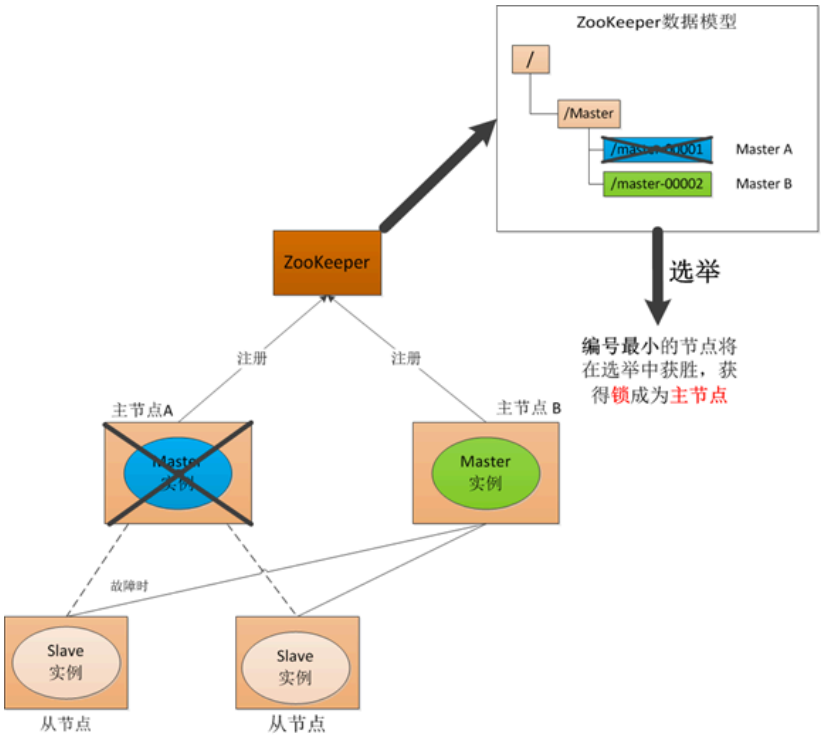
基本流程：分布式锁应用场景

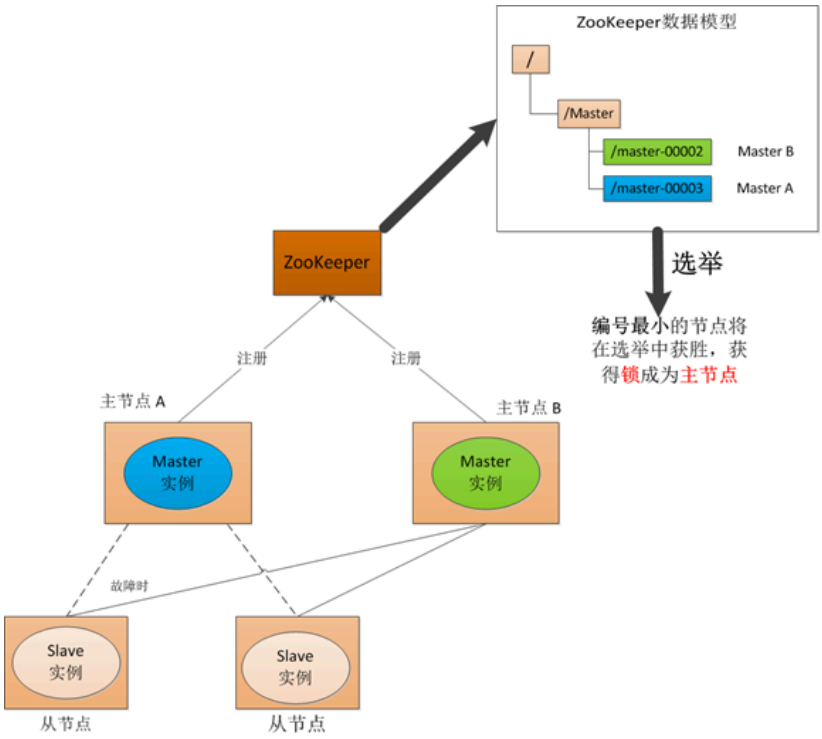
1、传统的一主n从分布式系统，容易发生单点故障，传统解决方式是增加一个备用节点，定期给主节点发送Ping包，主节点回复ack，但是如果网络原因ack丢失，那么会出现两个主节点，造成数据混乱。

2、zookeeper的引入可以管理两个主节点，其中挂了一个，会将另外一个作为新的主节点，挂的节点回来时担任备用节

点；


ZooKeeper数据模型
/
/Master
/master-00001
Master A
/master-00002
Master B
选举
编号最小的节点将在选举中获胜，获得锁成为主节点
ZooKeeper
注册
注册
主节点A
主节点 B
Master 实例
Master 实例
Slave 实例
Slave 实例
从节点
从节点


ZooKeeper数据模型
/
/Master
/master-00001
Master A
/master-00002
Master B
选举
编号最小的节点将在选举中获胜，获得锁成为主节点
ZooKeeper
注册
注册
主节点A
主节点 B
Master 实例
Master 实例
故障时
Slave 实例
Slave 实例
从节点
从节点



4.cap理论

1、概念：一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。

2、一致性：更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致，所以，一致性，说的就是数据一致性。

3、可用性：服务一直可用，而且是正常响应时间。

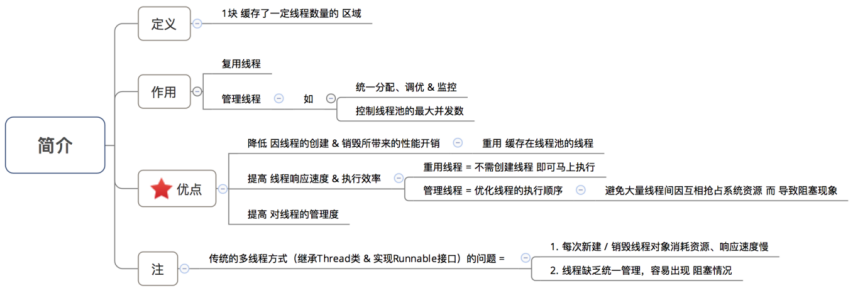
4、分区容错性：分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。

5.二段式满足cap理论的哪两个理论

两阶段提交协议在正常情况下能保证系统的强一致性，但是在出现异常情况下，当前处理的操作处于错误状态，需要管理员人工干预解决，因此可用性不够好，这也符合CAP协议的一致性和可用性不能兼得的原理。

6.线程池的参数配置，为什么java官方提供工厂方法给线程池

1、线程池简介：



2、核心参数：

参数	意义	说明
corePoolSize	核心线程数	默认情况下，核心线程会一直存活 （包括 空闲状态）
maximumPoolSize	线程池所能容纳的最大线程数	当活动线程数到达该数值后，后续的新任务将会阻塞
keepAliveTime	非核心线程 闲置超时时长	超过该时长，非核心线程就会被回收 （当将 allowCoreThreadTimeout 设置为true时，keepAliveTime同样作用于核心线程）
unit	指定keepAliveTime参数的时间单位	常用：（毫秒）TimeUnit.MILLISECONDS、（秒）TimeUnit.SECONDS、（分）TimeUnit.MINUTES
workQueue	任务队列	通过线程池的execute（）方法提交的Runnable对象 将存储在该参数中
threadFactory	线程工厂 （是1个接口）	• 作用 = 为线程池创建新线程 • 具体使用 = 只有1个方法：Thread newThread(Runnable r)

3、工厂方法作用：ThreadPoolExecutor类就是Executor的实现类，但ThreadPoolExecutor在使用上并不是那么方便，在实例化时需要传入很多歌参数，还要考虑线程的并发数等与线程池运行效率有关的参数，所以官方建议使用Executors工程类来创建线程池对象。

7.分布式框架dubbo的好处，不用dubbo可不可以。为什么要使用分布式

1、dubbo好处：

1、远程通讯: 提供对多种基于长连接的NIO框架抽象封装， 包括多种线程模型，序列化，以及“请求-响应”模式的信息交换方式。

2、软负载均衡及容错机制: 提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。

可在内网替代F5等硬件负载均衡器，降低成本，减少单点。

3、服务自动注册与发现: 基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器 。

4、提供完善的管理控制台dubbo-admin与简单的控制中心dubbo-monitor

5、Dubbo提供了伸缩性很好的插件模型，很方便进行扩展（ExtensionLoader）

2、不用dubbo可不可以：可以，使用springcloud。

3、分布式作用：

a. 系统之间的耦合度大大降低，可以独立开发、独立部署、独立测试，系统与系统之间的边界非常明确，排错也变得相当容易，开发效率大大提升。

b. 系统之间的耦合度降低，从而系统更易于扩展。我们可以针对性地扩展某些服务。假设这个商城要搞一次大促，下单量可能会大大提升，因此我们可以针对性地提升订单系统、产品系统的节点数量，而对于后台管理系统、数据分析系统而言，节点数量维持原有水平即可。

c. 服务的复用性更高。比如，当我们将用户系统作为单独的服务后，该公司所有的产品都可以使用该系统作为用户系统，无需重复开发。

8.七个垃圾回收器之间如何搭配使用

1. Serial New收集器是针对新生代的收集器，采用的是复制算法；
2. Parallel New（并行）收集器，新生代采用复制算法，老年代采用标记整理；
3. Parallel Scavenge（并行）收集器，针对新生代，采用复制收集算法；
4. Serial Old（串行）收集器，新生代采用复制，老年代采用标记清理；
5. Parallel Old（并行）收集器，针对老年代，标记整理；
6. CMS收集器，基于标记清理；

7. G1收集器(JDK)：整体上是基于标记清理，局部采用复制；

综上：新生代基本采用复制算法，老年代采用标记整理算法。cms采用标记清理；

9.接口限流方案

1. 限制 总并发数（比如 数据库连接池、线程池）
2. 限制 瞬时并发数（如 `nginx` 的 `limit_conn` 模块，用来限制 瞬时并发连接数）
3. 限制 时间窗口内的平均速率（如 `Guava` 的 `RateLimiter`、`nginx` 的 `limit_req` 模块，限制每秒的平均速率）
4. 限制 远程接口 调用速率
5. 限制 `MQ` 的消费速率
6. 可以根据 网络连接数、网络流量、`CPU` 或 内存负载 等来限流

10.ConcurrentHashMap使用原理

1、工作机制（分片思想）：它引入了一个“分段锁”的概念，具体可以理解为把一个大的Map拆分成N个小的segment，根据key.hashCode()来决定把key放到哪个HashTable中。可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。

2、应用：当读>写时使用，适合做缓存，在程序启动时初始化，之后可以被多个线程访问；

3、hash冲突：

1、简介：HashMap中调用hashCode()方法来计算hashCode。由于在Java中两个不同的对象可能有一样的hashCode,所以不同的键可能有一样hashCode，从而导致冲突的产生。

2、hash冲突解决：使用平衡树来代替链表，当同一hash中的元素数量超过特定的值便会由链表切换到平衡树

4、无锁读：ConcurrentHashMap之所以有较好的并发性是因为ConcurrentHashMap是无锁读和加锁写，并且利用了分段锁（不是在所有的entry上加锁，而是在一部分entry上加锁）；

```
/* Specialized implementations of map methods */

V get(Object key, int hash) {
    if (count != 0) { // read-volatile
        HashEntry<K,V> e = getFirst(hash);
        while (e != null) {
            if (e.hash == hash && key.equals(e.key)) {
                V v = e.value;
                if (v != null)
                    return v;
                return readValueUnderLock(e); // recheck
            }
            e = e.next;
        }
    }
    return null;
}
```

读之前会先判断count(jdk1.6)，其中的count是被volatile修饰的(当变量被volatile修饰后，每次更改该变量的时候会将更改结果写到系统主内存中，利用多处理器的缓存一致性，其他处理器会发现自己的缓存行对应的内存地址被修改，就会将自己处理器的缓存行设置为失效，并强制从系统主内存获取最新的数据。)，故可以实现无锁读。

11.解决map的并发问题方案

HashMap不是线程安全的；Hashtable线程安全，但效率低，因为是Hashtable是使用synchronized的，所有线程竞争同一把锁；而ConcurrentHashMap不仅线程安全而且效率高，因为它包含一个segment数组，将数据分段存储，给每一段数据配一把锁，也就是所谓的锁分段技术。

12.什么是协程，以及实现要点

1、生产者/消费者模式不是高性能的实现：

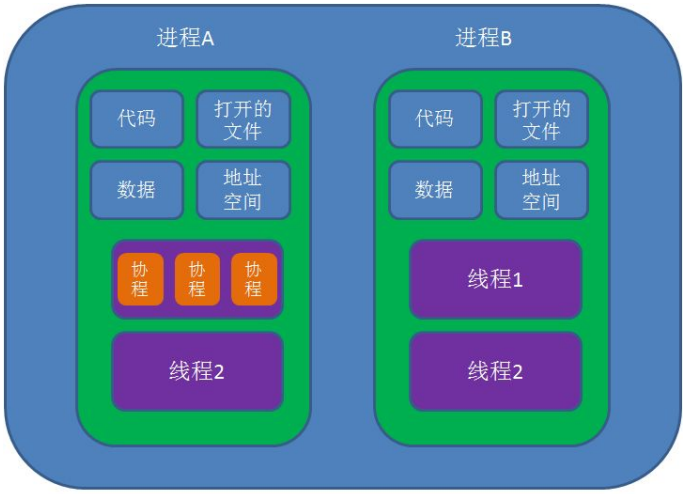
1.涉及到同步锁。

2.涉及到线程阻塞状态和可运行状态之间的切换。

3.涉及到线程上下文的切换。

2、协成定义：协程，英文Coroutines，是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。

这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。



3、协成优点：协程的暂停完全由程序控制，线程的阻塞状态是由操作系统内核来进行切换。因此，协程的开销远远小于线程的开销。

4、实现：

1、Lua语言

Lua从5.0版本开始使用协程，通过扩展库coroutine来实现。

2、Python语言

正如刚才所写的代码示例，python可以通过 yield/send 的方式实现协程。在python 3.5以后， async/await 成为了更好的替代方案。

3、Go语言

Go语言对协程的实现非常强大而简洁，可以轻松创建成百上千个协程并发执行。

4、Java语言

Java语言并没有对协程的原生支持，但是某些开源框架模拟出了协程的功能

13.lru cache 使用hash map 的实现（算法）

1、概念：其实解释起来很简单，LRU就是Least Recently Used的缩写，翻译过来就是“最近最少使用”。也就是说LRU算法会将最近最少用的缓存移除，让给最新使用的缓存。而往往最常读取的，也就是读取次数最多的，所以利用好LRU算法，我们能够提供对热点数据的缓存效率，能够提高缓存服务的内存使用率。

2、实现：

1、思路：

i. 限制缓存大小

ii. 查询出最近最晚用的缓存

iii. 给最近最少用的缓存做一个标识

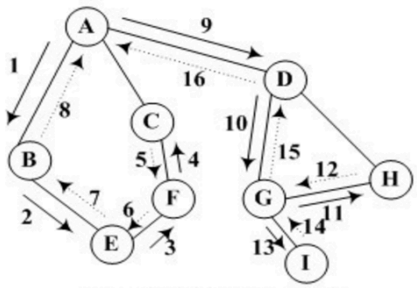
2、代码：

```
import java.util.LinkedHashMap;
import java.util.Map;
/**
 * 简单用LinkedHashMap来实现的LRU算法的缓存
 */
public class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private int cacheSize;
    public LRUCache(int cacheSize) {
        super(16, (float) 0.75, true);
         this.cacheSize = cacheSize;
     }
     protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
```

```
13          return size() > cacheSize;
14      }
15  }
```

14.图的深度遍历和广度遍历（算法）

1、深度优先遍历：

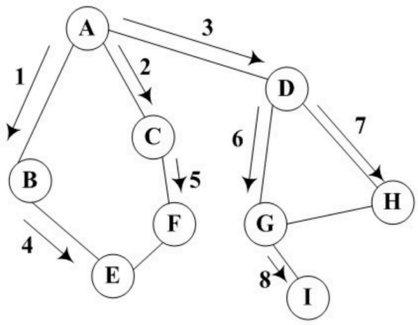


图G_6的深度优先遍历过程

深度优先遍历结果是： A B E F C D G H I

> 深度优先遍历尽可能优先往深层次进行搜索

2、广度优先遍历：



广度优先遍历结果是： A B C D E F G H I

> 广度优先遍历按层次优先搜索最近的结点，一层一层往外搜索。

15.基本排序（算法）

1. 快速排序：
   a. 原理：快速排序采用的是一种分治的思想,它先找一个基准数,然后将比这个基准数小的数字都放到它的左边,然后再递归调用,分别对左右两边快速排序,直到每一边只有一个数字.整个排序就完成了.
   b. 复杂度：O(n)
   c. 特点：快速排序是我们平常最常使用的一种排序算法,因为它速度快,效率高,是最优秀的一种排序算法.
2. 冒泡排序：
   a. 原理：冒泡排序其实就是逐一比较交换,进行里外两次循环,外层循环为遍历所有数字,逐个确定每个位置,里层循环为确定了位置后,遍历所有后面没有确定位置的数字,与该位置的数字进行比较,只要比该位置的数字小,就和该位置的数字进行交换.
   b. 复杂度：O(n^2)，最佳时间复杂度为O(n)
   c. 特点：冒泡排序在我们实际开发中,使用的还是比较少的.它更加适合数据规模比较少的时候,因为它的效率是比较低的,但是优点是逻辑简单,容易让我们记得.
3. 直接插入排序：
   a. 原理：直接插入排序是将从第二个数字开始,逐个拿出来,插入到之前排好序的数列里.
   b. 复杂度：O(n^2)，最佳时间复杂度为O(n)
   c. 特点：
4. 直接选择排序：
   a. 原理：直接选择排序是从第一个位置开始遍历位置,找到剩余未排序的数据里最小的,找到最小的后,再做交换
   b. 复杂度：O(n^2)

c. 特点：和冒泡排序一样,逻辑简单,但是效率不高,适合少量的数据排序

16.设计模式的使用

17.java 8 流式使用

```
List<Integer> evens = nums.stream().filter(num -> num % 2 == 0).collect(Collectors.toList(
//1、stream()操作将集合转换成一个流，
//2、filter()执行我们自定义的筛选处理，这里是通过lambda表达式筛选出所有偶数，
//3、最后我们通过collect()对结果进行封装处理，并通过Collectors.toList()指定其封装成为一个List集合返回
```

18.java的对象一致性

19.操作系统的读写屏障

20.java 域的概念

field，域是一种属性，可以是一个类变量，一个对象变量，一个对象方法变量或者是一个函数的参数。

21.分布式设计领域的概念

1、分布式系统设计的两大思路：中心化和去中心化

- 中心化：中心化的设计思想在自然界和人类生活中是如此的普遍和自然，它的设计思想也很简单，分布式集群中的节点按照角色分工，可以分为两种角色--“领导”和“干活的”，中心化的一个思路就是“领导”通常分发任务并监督“干活的”，谁空闲了就给它安排任务，谁病倒了就一脚踢出去，然后把它的任务分给其他人；中心化的另一个思路是领导只负责生成任务而不再指派任务，由每个“干活的”自发去领任务。
- 去中心化：全球IP互联网就是一个典型的去中心化的分布式控制架构，联网的任意设备宕机都只会影响很小范围的功能。去中心化设计通常没有“领导”和“干活的”，角色一样，地位平等，因此不存在单点故障。实际上，完全意义的去中心化分布式系统并不多见，很多看起来是去中心化但工作机制采用了中心化设计思想的分布式系统正在不断涌现，在这种架构下，集群中的领导是动态选择出来的，而不是人为预先指定的，而且在集群发生故障的情况下，集群的成员会自发举行会议选举新的领导。典型案例如：zookeeper、以及Go语言实现的Etcd。

2、分布式系统的一致性原理

- 在说明一致性原理之前，可以先了解一下cap理论和base理论，具体见《事务与柔性事务》中的说明。
- 对于多副本的一致性处理，通常有几种方法：同步更新--即写操作需要等待两个节点都更新成功才返回，这样的话如果一旦发生网络分区故障，写操作便不可用，牺牲了A。异步更新--即写操作直接返回，不需要等待节点更新成功，节点异步地去更新数据，这种方式，牺牲了C来保证A。折衷--只要保证集群中超过半数的节点正常并达到一致性即可满足要求，此时读操作只要比较副本集数据的修改时间或者版本号即可选出最新的，所以系统是强一致性的。如果允许“数据一致性存在延迟时间”，则是最终一致性。
- 如Cassandra中的折衷型方案QUORUM，只要超过半数的节点更新成功便返回，读取时返回多数副本的一致的值。然后，对于不一致的副本，可以通过read repair的方式解决。read repair：读取某条数据时，查询所有副本中的这条数据，比较数据与大多数副本的最新数据是否一致，若否，则进行一致性修复。此种情况是强一致性的。
- 又如Redis的master-slave模式，更新成功一个节点即返回，其他节点异步地去备份数据。这种方式只保证了最终一致性。最终一致性：相比于数据时刻保持一致的强一致性，最终一致性允许某段时间内数据不一致。但是随着时间的增长，数据最终会到达一致的状态。此种情况只能保证最终一致性。著名的DNS也是最终一致性的成功例子。
- 强一致性算法：1989年就诞生了著名的Paxos经典算法（zookeeper就采用了Paxos算法的近亲兄弟Zab算法），但由于Paxos算法难以理解、实现和排错，所以不断有人尝试优化算法，2013年终于有了重大突破：Raft算法的出现，其中Go语言实现的Raft算法就是Etcd，功能类似于zookeeper。
- Base的思想：基本可用、柔性状态、最终一致性，主要针对数据库领域的数据拆分，通过数据分片（如Mycat、Amodeba等）来提升系统的可用性。由于分片拆分后会涉及分布式事务，所以接下来看一下如何用最终一致性的思路来实现分布式事务，也就是柔性事务。

3、柔性事务：具体见《事务与柔性事务》。

4、分布式系统的关键Zookeeper

- 目标是解决分布式系统的几个问题：集群集中化配置，集群节点动态发现机制，简单可靠的节点Leader选举机制，分布式锁。
- ZNode有一个ACL访问权限控制列表，提供对节点增删改查的API，提供监听ZNode变化的实时通知接口--Watch接口。
- ZNode类型：持久节点（可以实现配置中心）、临时节点（和创建这个节点的客户端会话绑定，可实现集群节点动态发现，可以实现服务注册中心）、时序节点（创建节点时会加上数字后缀，通过选择编号最小的ZNode可以实现Leader选举机制）、临时性时序节点（同时具备临时节点和时序节点的特性，主要用于分布式锁的实现）。

22.如何实现双11的购物限流（redis实现方案）

1、限流策略：

- Nginx接入层限流

按照一定的规则如帐号、IP、系统调用逻辑等在Nginx层面做限流

- 业务应用系统限流

通过业务代码控制流量这个流量可以被称为信号量，可以理解成是一种锁，它可以限制一项资源最多能同时被多少进程访问。

2、lua脚本：

```
local key = KEYS[1] --限流KEY（一秒一个）
local limit = tonumber(ARGV[1]) --限流大小
local current = tonumber(redis.call('get', key) or "0")
if current + 1 > limit then --如果超出限流大小
    return 0
else --请求数+1，并设置2秒过期
    redis.call("INCRBY", key,"1")
    redis.call("expire", key,"2")
end
 return 1
```

减少网络开销: 不使用 Lua 的代码需要向 Redis 发送多次请求, 而脚本只需一次即可, 减少网络传输;
原子操作: Redis 将整个脚本作为一个原子执行, 无需担心并发, 也就无需事务;
复用: 脚本会永久保存 Redis 中, 其他客户端可继续使用.

2、ip限流lua脚本：

```
local key = "rate.limit:" .. KEYS[1]
local limit = tonumber(ARGV[1])
local expire_time = ARGV[2]

local is_exists = redis.call("EXISTS", key)
if is_exists == 1 then
    if redis.call("INCR", key) > limit then
        return 0
    else
        return 1
     end
 else
     redis.call("SET", key, 1)
     redis.call("EXPIRE", key, expire_time)
     return 1
 end
```

3、java执行代码：

```
import org.apache.commons.io.FileUtils;

import redis.clients.jedis.Jedis;

import java.io.File;
import java.io.IOException;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.List;
 import java.util.concurrent.CountDownLatch;

public class RedisLimitRateWithLUA {
```

```
13
14      public static void main(String[] args) {
15          final CountDownLatch latch = new CountDownLatch(1);
16
17          for (int i = 0; i < 7; i++) {
18              new Thread(new Runnable() {
19                  public void run() {
20                      try {
21                          latch.await();
22                          System.out.println("请求是否被执行: "+accquire());
23                      } catch (Exception e) {
24                          e.printStackTrace();
25                      }
26                  }
27              }).start();
28
29          }
30
31          latch.countDown();
32      }
33
34      public static boolean accquire() throws IOException, URISyntaxException {
35          Jedis jedis = new Jedis("127.0.0.1");
36          File luaFile = new File(RedisLimitRateWithLUA.class.getResource("/").toURI().getP
37          String luaScript = FileUtils.readFileToString(luaFile);
38
39          String key = "ip:" + System.currentTimeMillis()/1000; // 当前秒
40          String limit = "5"; // 最大限制
41          List<String> keys = new ArrayList<String>();
42          keys.add(key);
43          List<String> args = new ArrayList<String>();
44          args.add(limit);
45          Long result = (Long)(jedis.eval(luaScript, keys, args)); // 执行lua脚本，传入参数
46          return result == 1;
47      }
48  }
49
```

23.mysql调优

1、选择最合适的字段属性：类型、长度、是否允许NULL等；尽量把字段设为not null，一面查询时对比是否为null；

2.要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

3.应尽量避免在 where 子句中对字段进行 null 值判断、使用!= 或 <> 操作符，否则将导致引擎放弃使用索引而进行全表扫描

4.应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描

5.in 和 not in 也要慎用，否则会导致全表扫描

6.模糊查询也将导致全表扫描，若要提高效率，可以考虑字段建立前置索引或用全文检索；

7.如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。

9.应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。

10.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

12.不要写一些没有意义的查询，如需要生成一个空表结构：

13.Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志。

14.对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

15.select count(*) from table；这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。

16.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。

17.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

18.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连 接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

19.尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

20.任何地方都不要使用 select * from t ，用具体的字段列表代替“*”，不要返回用不到的任何字段。

21.尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

22. 避免频繁创建和删除临时表，以减少系统表资源的消耗。临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件， 最好使用导出表。

23.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

24.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

25.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

26.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

27.与临时表一样，游标并不是不可使用。对小型数据集使用 FAST_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE_IN_PROC 消息。

29.尽量避免大事务操作，提高系统并发能力。

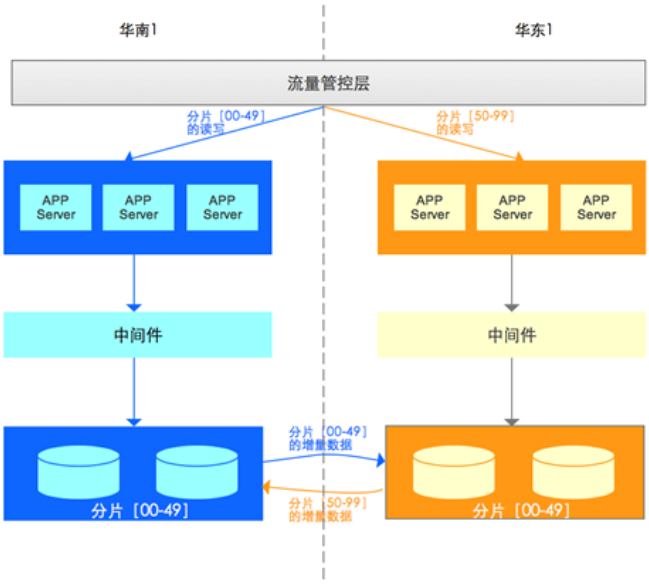
30.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

24.cdn（异地多活）

1、异地多活：异地多活指分布在异地的多个站点同时对外提供服务的业务场景。异地多活是高可用架构设计的一种，与传统的灾备设计的最主要区别在于“多活”，即所有站点都是同时在对外提供服务的。

2、两地容灾切换方案：

容灾是异地多活中最核心的一环， 以两个城市异地多活部署架构图为例：



- 在两个城市（城市1位于华南1地域、城市2位于华东1地域）均部署一套完整的业务系统。
- 下单业务按照“user_id”% 100 进行分片，在正常情况下：
  - [00~49]分片所有的读写都在城市1的数据库实例主库。
  - [50～99]分片所有的读写都在城市2的数据库实例主库。
- “城市1的数据库实例主库”和 “城市2的数据库实例主库”建立DTS双向复制。

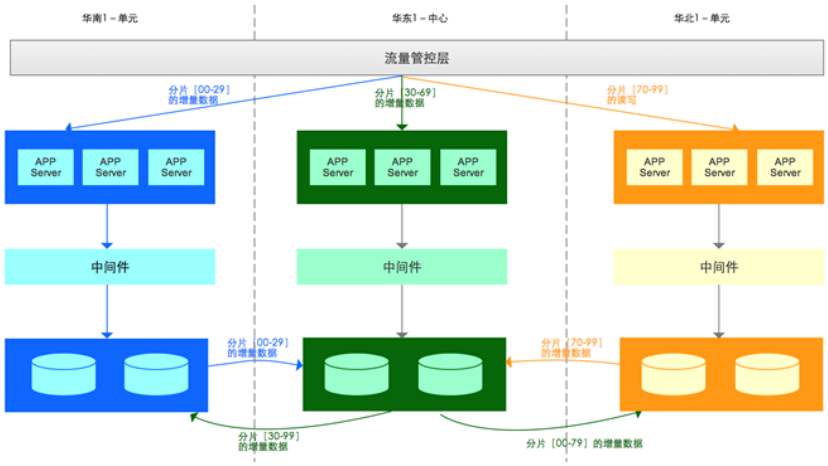
当出现异常时，需要进行容灾切换。可能出现的场景有以下4种：

序号	异常情况	操作
1	城市1数据库主库故障	1. 数据库引擎完成主备切换 2. DTS自动切换到城市1新主库读取新的增量更新，然后同步到城市2的数据库实例
2	城市1所有APP Server故障	有两种处理方案： • 方案1：数据库层无任何操作，APP Server切换到城市2，并跨城市读写城市1的数据库 • 方案2：APP Server和数据库都切换到城市2
3	城市1所有数据库故障	有两种处理方案： • 方案1：数据库层切换到城市2，APP Server跨城市读写城市2的数据库 • 方案2：APP Server和数据库都切换到城市2
4	城市1整体故障（包括所有APP Server ＋数据库等）	1. 城市1的全部数库流量切换到城市2 2. 城市1数据库到城市2数据库的DTS数据同步链路停止 3. 在城市2中，DTS启动，保存［00-49］分片的变更 4. 城市1故障恢复后，［00-49］的增量数据同步到城市1的数据库实例 5. 同步结束后，将［00-49］的数据库流量从城市2切回到城市1启动［00-49］分片从城市1到城市2的DTS同步

将第2种、第3种异常情况，全部采用第2种方案进行处理，那么不管是所有的APP Server异常、所有的数据库异常、整个城市异常，就直接按照城市级容灾方案处理，直接将APP Server、数据库切换到到另一个城市。

3、多城异地多活：

多城市异地多活模式指的是3个或者3个以上城市间部署异地多活。该模式下存在中心节点和单元节点：



- 中心节点：指单元节点的增量数据都需要实时的同步到中心节点，同时中心节点将所有分片的增量数据同步到其他单元节点。

- 单元节点：即对应分片读写的节点，该节点需要将该分片的增量同步到中心节点，并且接收来自于中心节点的其他分片的增量数据。

下图是3城市异地多活架构图，其中华东1就是中心节点，华南1和华北1是单元节点。

25.进程之间的通信方式

1、匿名管道通信：

a. 父进程创建管道，得到两个文件描述符指向管道的两端

b. 父进程fork出子进程，子进程也有两个文件描述符指向同一管道。

c. 父进程关闭fd[0],子进程关闭fd[1]，即父进程关闭管道读端,子进程关闭管道写端（因为管道只支持单向通信）。父进程可以往管道里写,子进程可以从管道里读,管道是用环形队列实现的,数据从写端流入从读端流出,这样就实现了进程间通信。

2、高级管道通信：将另一个程序当做一个新的进程在当前程序进程中启动，则它算是当前程序的子进程，这种方式我们成为高级管道方式。

3、有名管道通信：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

4、消息队列通信：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

5、信号量通信：

6、信号通信：

7、共享内存通信：

8、套接字通信：

26.tcp/ip协议、http协议

27.二级交换器传输协议

28.spring 事务的传播

29.分布式下down机的处理方案（心跳检测）

1、dubbo：服务器宕机，zk临时被删除；

2、springcloud：每30s发送心跳检测重新进行租约，如果客户端不能多次更新租约，它将在90s内从服务器注册中心移除。

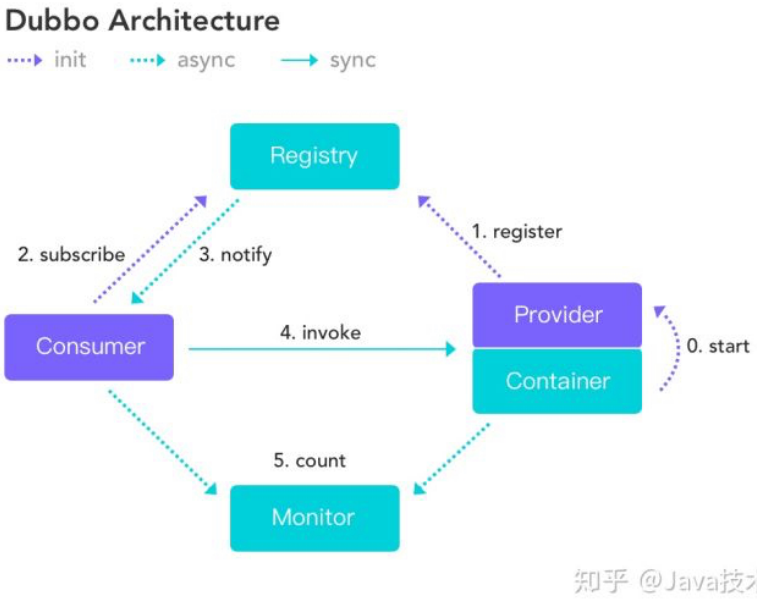
3、apm监控：

30、分布式弱一致性下down机的处理方案

31、dubbo与zookeeper 两者作为注册中心的区别，假如注册中心挂了，消费者还能调用服务吗，用什么调用的

1. 注册中心对等集群，任意一台宕掉后，会自动切换到另一台
2. 注册中心全部宕掉，服务提供者和消费者仍可以通过本地缓存通讯
3. 服务提供者无状态，任一台 宕机后，不影响使用
4. 服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复

32、dubbo的原理图（画出注册中心，消费者，生产者的关系图，并说出每个角色的作用）



Consumer服务消费者，Provider服务提供者。Container服务容器。消费当然是invoke提供者了，invoke这条实线按照图上的说明当然同步的意思了。但是在实际调用过程中，Provider的位置对于Consumer来说是透明的，上一次调用服务的位置（IP地址）和下一次调用服务的位置，是不确定的。这个地方就需要使用注册中心来实现软负载。

33、项目中有没有用到多线程，用到的话用了哪些

34、HashMap的底层原理（包括底层数据结构，怎么扩容的）

1、数据结构中有数组和链表来实现对数据的存储，但这两者基本上是两个极端。那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？答案是肯定的，这就是我们要提起的哈希表。哈希表（(Hash table）既满足了数据的查找方便，同时不占用太多的内容空间，使用也十分方便。

2、HashMap底层是采用数组来维护的.Entry静态内部类的数组

```
/**
* The table, resized as necessary. Length MUST Always be a power of two.
*/
transient Entry[] table;

static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
     final int hash;
     ……
}
```

3、HashMap添加元素：将准备增加到map中的对象与该位置上的对象进行比较(equals方法)，如果相同，那么就将该位置上的那个对象(Entry类型)的value值替换掉，否则沿着该Entry的链继续重复上述过程，如果到链的最后任然没有找到与此对象相同的对象，那么这个时候就会被增加到数组中，将数组中该位置上的那个Entry对象链到该对象的后面(先hashcode计算位置，如果有值便替换，无值则重复hashcode计算，直到最后在添加到hashmap最后面；)

35、ConcurrentHashMap的原理

1、工作机制（分片思想）：它引入了一个“分段锁”的概念，具体可以理解为把一个大的Map拆分成N个小的segment，根据key.hashCode()来决定把key放到哪个HashTable中。可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。

2、应用：当读>写时使用，适合做缓存，在程序启动时初始化，之后可以被多个线程访问；

3、hash冲突：

1、简介：HashMap中调用hashCode()方法来计算hashCode。由于在Java中两个不同的对象可能有一样的hashCode，所以

不同的键可能有一样hashCode，从而导致冲突的产生。

2、hash冲突解决：使用平衡树来代替链表，当同一hash中的元素数量超过特定的值便会由链表切换到平衡树

4、无锁读：ConcurrentHashMap之所以有较好的并发性是因为ConcurrentHashMap是无锁读和加锁写，并且利用了分段锁（不是在所有的entry上加锁，而是在一部分entry上加锁）；

```
/* Specialized implementations of map methods */

V get(Object key, int hash) {
    if (count != 0) { // read-volatile
        HashEntry<K,V> e = getFirst(hash);
        while (e != null) {
            if (e.hash == hash && key.equals(e.key)) {
                V v = e.value;
                if (v != null)
                    return v;
                return readValueUnderLock(e); // recheck
            }
            e = e.next;
        }
    }
    return null;
}
```

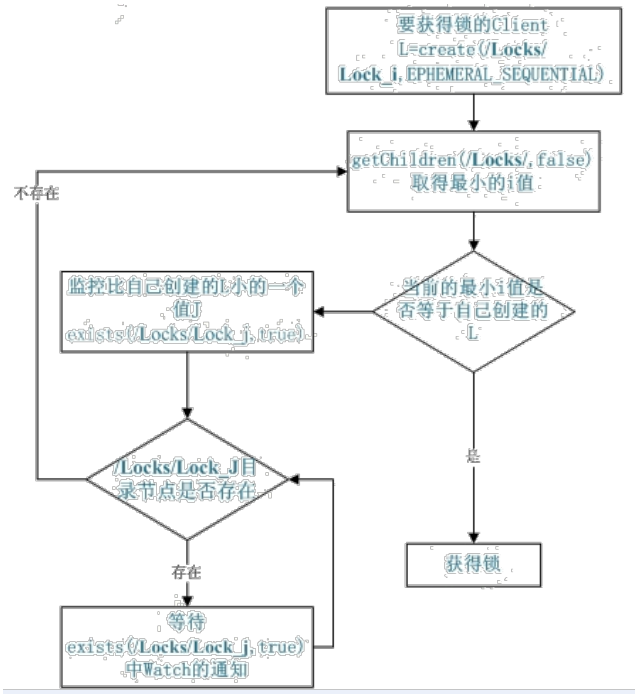
读之前会先判断count(jdk1.6)，其中的count是被volatile修饰的(当变量被volatile修饰后，每次更改该变量的时候会将更改结果写到系统主内存中，利用多处理器的缓存一致性，其他处理器会发现自己的缓存行对应的内存地址被修改，就会将自己处理器的缓存行设置为失效，并强制从系统主内存获取最新的数据。)，故可以实现无锁读。

36、 分布式锁的实现

基本原理：用一个状态值表示锁，对锁的占用和释放通过状态值来标识。

1、三种分布式锁：

1、Zookeeper：基于zookeeper瞬时有序节点实现的分布式锁，其主要逻辑如下（该图来自于IBM网站）。大致思想即为：每个客户端对某个功能加锁时，在zookeeper上的与该功能对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。



2、优点

锁安全性高，zk可持久化，且能实时监听获取锁的客户端状态。一旦客户端宕机，则瞬时节点随之消失，zk因而能第一时间释放锁。这也省去了用分布式缓存实现锁的过程中需要加入超时时间判断的这一逻辑。

3、缺点

性能开销比较高。因为其需要动态产生、销毁瞬时节点来实现锁功能。所以不太适合直接提供给高并发的场景使用。

4、实现

可以直接采用zookeeper第三方库curator即可方便地实现分布式锁。

5、适用场景

对可靠性要求非常高，且并发程度不高的场景下使用。如核心数据的定时全量/增量同步等。

2、memcached：memcached带有add函数，利用add函数的特性即可实现分布式锁。add和set的区别在于：如果多线程并发set，则每个set都会成功，但最后存储的值以最后的set的线程为准。而add的话则相反，add会添加第一个到达的值，并返回true，后续的添加则都会返回false。利用该点即可很轻松地实现分布式锁。

2、优点

并发高效

3、缺点

memcached采用列入LRU置换策略，所以如果内存不够，可能导致缓存中的锁信息丢失。

memcached无法持久化，一旦重启，将导致信息丢失。

4、使用场景

高并发场景。需要 1)加上超时时间避免死锁； 2)提供足够支撑锁服务的内存空间； 3)稳定的集群化管理。

3、redis：redis分布式锁即可以结合zk分布式锁锁高度安全和memcached并发场景下效率很好的优点，其实现方式和memcached类似，采用setnx即可实现。需要注意的是，这里的redis也需要设置超时时间。以避免死锁。可以利用jedis客户端实现。

```
ICacheKey cacheKey = new ConcurrentCacheKey(key, type);
return RedisDao.setnx(cacheKey, "1");
```

2、数据库死锁机制和解决方案：

1、死锁：死锁是指两个或者两个以上的事务在执行过程中，因争夺锁资源而造成的一种互相等待的现象。

2、处理机制：解决死锁最有用最简单的方法是不要有等待，将任何等待都转化为回滚，并且事务重新开始。但是有可能影响并发性能。

1、超时回滚，innodb_lock_wait_time设置超时时间；

2、wait-for-graph方法：跟超时回滚比起来，这是一种更加主动的死锁检测方式。InnoDB引擎也采用这种方式。

37、分布式session，如何保持一致

1、Session粘滞

1、将用户的每次请求都通过某种方法强制分发到某一个Web服务器上，只要这个Web服务器上存储了对应Session数据，就可以实现会话跟踪。

2、优点：使用简单，没有额外开销。

3、缺点：一旦某个Web服务器重启或宕机，相对应的Session数据将会丢失，而且需要依赖负载均衡机制。

4、适用场景：对稳定性要求不是很高的业务情景。

2、Session集中管理

1、在单独的服务器或服务器集群上使用缓存技术，如Redis存储Session数据，集中管理所有的Session，所有的Web服务器都从这个存储介质中存取对应的Session，实现Session共享。

2、优点：可靠性高，减少Web服务器的资源开销。

3、缺点：实现上有些复杂，配置较多。

4、适用场景：Web服务器较多、要求高可用性的情况。

5、可用方案：开源方案Spring Session，也可以自己实现，主要是重写HttpServletRequestWrapper中的getSession方法，博主也动手写了一个，github搜索joincat用户，然后自取。

3、基于Cookie管理

1、这种方式每次发起请求的时候都需要将Session数据放到Cookie中传递给服务端。

2、优点：不需要依赖额外外部存储，不需要额外配置。

3、缺点：不安全，易被盗取或篡改；Cookie数量和长度有限制，需要消耗更多网络带宽。

4、适用场景：数据不重要、不敏感且数据量小的情况。

总结

这四种方式，相对来说，Session集中管理更加可靠，使用也是最多的。

38、消息中间件都用到哪些，他们的区别

特性	ActiveMQ	RabbitMQ	RocketMQ	kafka
开发语言	java	erlang	java	scala
单机吞吐量	万级	万级	10万级	10万级
时效性	ms级	us级	ms级	ms级以内
可用性	高(主从架构)	高(主从架构)	非常高(分布式架构)	非常高(分布式架构)
功能特性	成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好	基于erlang开发，所以并发能力很强，性能极其好，延时很低;管理界面较丰富	MQ功能比较完备，扩展性佳	只支持主要的MQ功能，像一些消息查询，消息溯等功能没有提供，是为大数据准备的，在数据领域应用广。

1. 中小型公司首选RabbitMQ：管理界面简单，高并发。
2. 大型公司可以选择RocketMQ：更高并发，可对rocketmq进行定制化开发。
3. 日志采集功能，首选kafka，专为大数据准备。

技术要点：

1. java基础知识：如：jdk源码，concurrent包常用类的原理，NIO原理，垃圾回收机制等
2. 中间件及数据库相关知识：消息中间件、缓存、elasticsearch、ealsticjob、数据库优化等
3. 微服务相关知识：dubbo、springcloud、限流、隔离、熔断机制、调用链、分布式环境数据一致性等
4. 生产环境问题定位和处理能力：cpu异常、内存溢出、线程池爆、redis挂、kafka挂等
5. 有参与过知名的开源项目的面试人优先
6. 有分享高质量博客、熟读主流开源框架原理的优先
7. 在过往工作中有突出成绩的优先，包括但不限于：推动不合理的架构向合理的架构转变、能主动用技术的手段去解决工作中不合理的情况等
8，HashMap的底层原理（包括底层数据结构，怎么扩容的）

> 丰巢--0313

1. 我们知道hashmap线程不安全，那用什么类可以代替它保证线程安全呢？他们又是如何实现线程安全的呢?
2. 说说几种GC机制，年轻代什么时候触发gc?
3. 说说一致性hash?
4. mybatis基础知识；
5. mysql基础知识；
6. mysql单表达到多少数据量需要分库分表?
7. hibernate基础知识。
8. 说说kafka的原理，为什么能保证这么高的吞吐量?
9. 对webservice有什么了解?
10. 说说你们公司git分支管理方案?
11. 用过mysql吗？如何进行分表分库？什么时候需要分表分库?
12. 你们如何和前端进行接口联调?
13. 说说你平时遇到的重大难题或者挑战，以及你解决问题的思路和流程。