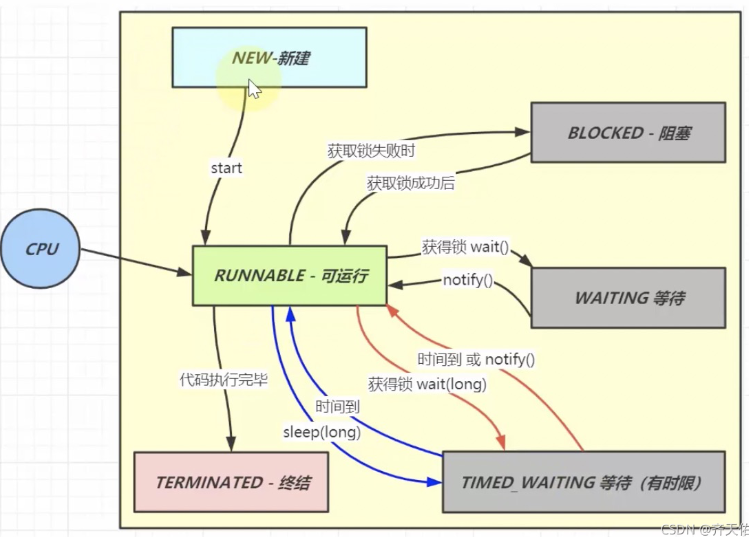
## 基础题目

##### Java线程的状态

有6种和5种两种划分方式

**6种**



* **新建/初始：**用new创建了一个新的线程对象。在这个状态的线程没有与操作系真正的线程产生关联，仅仅是一个java对象。
* **可运行：**位于可运行线程池中，等待被线程调度选中，获得CPU的使用权。只有处于可运行状态的线程才会得到cpu资源。

包括**就绪、运行中**两种状态，

##### 进程和线程的区别，进程间如何通讯，线程间如何通讯

##### HashMap的数据结构是什么？如何实现的。和HashTable，ConcurrentHashMap的区别

##### Cookie和Session的区别

##### 索引有什么用？如何建索引？

##### ArrayList是如何实现的，ArrayList和LinkedList的区别？ArrayList如何实现扩容。

##### equals方法实现

##### 面向对象

##### 线程状态，BLOCKED和WAITING有什么区别

##### JVM如何加载字节码文件

##### JVM GC，GC算法。

##### 什么情况会出现Full GC，什么情况会出现young GC。

JDK8以前的堆内存空间分为三个部分：新生代（包括Eden和Survivor）、老年代（old）和永久代（Permanent），JDK8及之后的永久代被元空间（Metaspace）所取代。

* 新对象优先在Eden区分配，如果Eden区满了就会触发young GC（也属于Minor GC，从新生代回收内存都叫做Minor GC）。
* 大对象直接进入老年代
* 如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间（s0 或者 s1）中，并将对象年龄设为 1(Eden 区->Survivor 区后对象的初始年龄变为 1)。对象在 Survivor 中每熬过一次 MinorGC,年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置。
* 如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且不能被Survivor 容纳但能被老年代容纳的话，就会把新生代的对象提前进入老年代。
* 当手动调用system.gc()方法，或者当统计得到的Minor GC晋升到老年代的平均大小大于老年代的剩余空间时，就会触发full GC。
* full GC对**整个堆**都进行了回收，包括新生代、老年代、永久代/元空间，发生full GC时所有线程都不可用，所以要尽量避免。

##### JVM内存模型

##### Java运行时数据区

##### 事务的实现原理

##### CopyOnWriteArrayList实现原理

* 线程安全的List
* 读取的时候完全不加锁，写入的时候也不会阻塞读取操作，只有写入和写入之间需要进行同步等待。适合读多写少的场景。
* CopyOnWriteArrayList 类的所有可变操作（add，set 等等）都是通过创建底层数组的新副本来实现的。当 List 需要被修改的时候，我并不修改原有内容，而是对原有数据进行一次复制，将修改的内容写入副本。写完之后，再将修改完的副本替换原来的数据，这样就可以保证写操作不会影响读操作了。

##### 深克隆和浅克隆

**浅克隆（深拷贝）：**在堆上创建一个新对象，如果这个对象的属性是引用类型的，那么这个属性仍指向原对象该属性的内存地址。

浅拷贝方式：直接调用Object.clone()，注意要implements Cloneable



如上图，这样调用person.clone()的时候只是把person拷贝了，它的引用类型属性addresss没有拷贝

**深克隆（浅拷贝）：**不仅创建了一个新对象，连同对象的引用类型属性也被克隆了。

深拷贝方式：重写该类的clone()方法，在该重写方法里把每个引用类型属性也给clone()了





如上图，把address也implements Cloneable，然后把person的clone方法改一下，把address也clone了

##### 事务隔离级别

并发数据库操作中可能会出现的几个问题：

|  |  |
| --- | --- |
| 名称 | 描述 |
| 更新丢失 | 更新被覆盖 |
| 脏读 | 读到了其他事务未提交的数据 |
| 不可重复读 | 同一事务对同一行读取两次，两次读取到的结果不同，  因为另一事务在这期间对这行做了修改。 |
| 幻读 | （本质上也属于不可重复读，但是指一定范围的多行）  同一事务两次读取某一范围的数据时读到不同的结果，  因为另一事务在这期间insert了新的行 |

不可重复读侧重的是数据的修改，幻读侧重的是数据的新增或删除。解决不可重复读，只有锁住行，而解决幻读，需要锁住整张表。

为解决这些问题提出的四个事务隔离级别：

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 描述 | 解决的问题 | 没解决的问题 |
| 读取未提交 | 一个事务读的时候没有限制，写的时候其他事务不能写但可以读，所以这个事务进行的更新操作不会丢失但它未提交的数据会被其他事务读到， | 更新丢失 | 脏读、不可重复读、幻读 |
| 读取已提交  （大多数数据库默认的） | 一个事务读的时候没有限制，写的时候在提交之前别的事务不能读也不能写，从而避免读到未提交的数据，但读操作没有限制，所以两次读取还是可能不同 | 更新丢失、脏读 | 不可重复读、幻读 |
| 可重复读  （mysql默认的） | 读时禁止写（只是禁止对现在读取的行进行写，还是可以insert新的行），写时禁止任何，保证同一事务中多次读取的结果一样 | 更新丢失、脏读、不可重复读 | 幻读（因为加的锁是行级锁，不是表级锁） |
| 可串行化 | 所有事务串行执行 | 所有 |  |

##### 事务的四个关键属性(ACID)

|  |  |
| --- | --- |
| 属性 | 描述 |
| 原子性A | 事务包含的一组更新操作是原子不可分的，要么全部成功，要么全部失败回滚（如i++包含从主内存读取i、在工作内存把i+1、把新的值写回到主内存这三个步骤） |
| 一致性C | 事务执行前后数据库都要处于一致性状态（如银行转账系统A转钱给B前后总钱数一样多） |
| 隔离性I | 当多个用户并发访问数据库时，数据库为每个用户开启的事务不能被其他事务干扰，多个并发事务要相互隔离 |
| 持久性D | 一旦事务提交，所作的修改要永远保存到数据库中，即使系统崩溃也不能丢失 |

##### 排序算法

**内部排序：**数据记录在内存中进行排序。

**外部排序：**因排序的数据很大，一次不能容纳全部的排序记录，在排序过程中需要访问外存（磁盘）。这里只讨论内部排序。



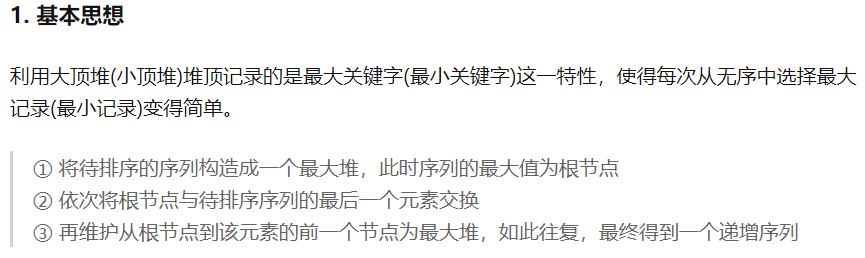
**n：**数据规模

**k：**“桶” 的个数

**In-place：**占用常数内存，不占用额外内存

**Out-place：**占用额外内存

* **冒泡排序：**重复地遍历要排序的序列，依次比较两个元素，如果它们的顺序错误就把它们交换过来。遍历序列的工作是重复地进行直到没有再需要交换为止，此时说明该序列已经排序完成。每一轮都会把此轮最大/最小的那个移到最后面。
* **选择排序：**首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。【**不稳定**：在把后面的数据放到前面时，实际上两个数据进行了交换（注意不是整体后移），所以当a1=a2时且原本a1在前面时，a1可能因为和a2后面的某个数字发生交换而跑到了a2后面，所以最后的结果a1可能会在a2后面】
* **插入排序：**从第一个元素开始，认为它已经排好序，对于未排序的数据，从已排序部分从后向前进行扫描，找到合适的位置将未排序的数据插进去。【**稳定：**因为是整体后移而非交换】
* **希尔排序：**先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，设置若干个gap，每相隔一个gap的元素在同一组，组内自己插入排序，一开始gap=length/2，后面每一轮gap都缩为前一轮的一半。等gap为1时，其实就是传统的全组插入排序，但这时已经排过很多轮了，微调一下就可以了。【**不稳定：**因为gap的原因相同元素可能被分到不同组】
* **归并排序：**分治法的典型应用，一个递归的过程——递归函数的输入为一个长度为n的数组，如果这个n为1则直接返回，否则把这个数组切成两半，递归调用两次，分别对两个子数组进行排序，并将这两个排序好的子数组进行合并，合并过程如下：若两个子数组分别长n1、n2，则设置一个长为n1+n2的额外空间，然后双指针法两个指针分别指向两个子数组的第一个元素，然后不断把最小的那个元素移到额外空间里，最后返回这个额外空间就是这两个子数组合并后的结果，然后返回到上一层递归函数，再进行合并……【稳定的：合并时按顺序】
* **快速排序：**也是分治法、递归——递归函数的参数是左边界l、右边界r，然后i=l，j=r，基准值pivot=nums[i]，while(i<j)时j、i交替走，交替更改nums[i]、nums[j]，最后ij相遇，另nums[i]=pivot，然后把[l,i-1]和[i+1,r]两半部分也进行递归。【不稳定:因为j--的条件是>=而不是>，i++的条件是<=而不是<】
* **堆排序：**



【空间复杂度O(1)：因为所有元素是就地构建为大顶堆/小顶堆的，不稳定：因为调整堆的过程中可能会改变相同元素的相对顺序】

## 技术深度

##### 有没有看过JDK源码，看过的类实现原理是什么。

##### HTTP协议

##### TCP协议

##### 一致性Hash算法

##### JVM如何加载字节码文件

##### 类加载器如何卸载字节码

##### IO和NIO的区别，NIO优点

|  |  |
| --- | --- |
| IO | NIO |
| 面向流。每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方，也不能前后移动流中的数据。 | 面向缓冲。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。 |
| 阻塞IO。当线程发出read请求时，如果当前数据还没来，该线程会一直阻塞等在那里，直到数据可读；当线程发出写请求时，会一直在那里写，直到数据全部写入。 | 非阻塞IO。线程发出read请求时，若现在数据还没来，该线程不会阻塞，而是去做别的事情，直到数据可读取再读；当线程发出write请求时，把数据写入通道即可，不需要等待数据完全写入。 |
| 无选择器 | 有选择器。Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。 |

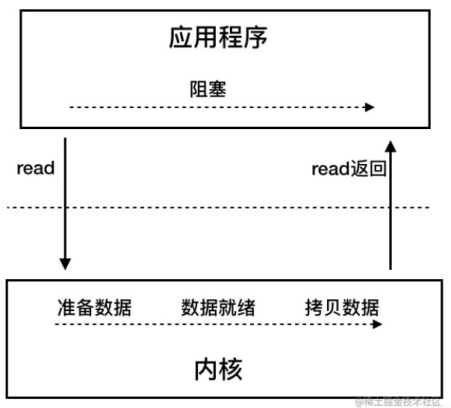
补充：JAVA中的IO

用户空间的程序不能直接访问内核空间。因此，用户进程想要执行 IO 操作的话，必须通过 系统调用 来间接访问内核空间。也就是说，我们的应用程序实际上只是发起了 IO 操作的调用而已，具体 IO 的执行是由操作系统的内核来完成的。

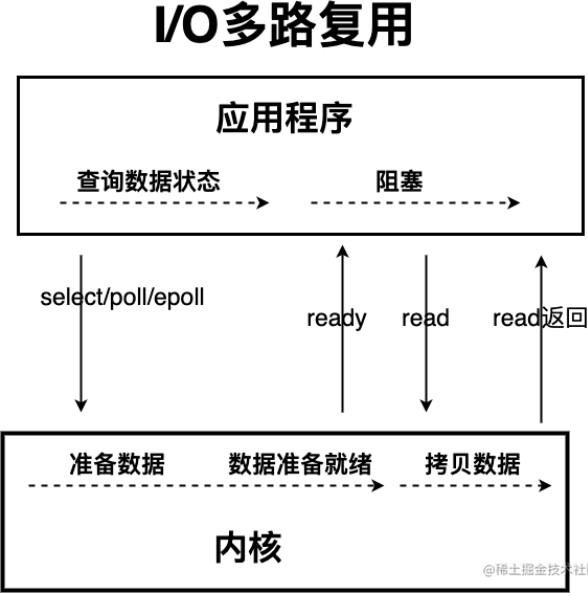
当应用程序发起 I/O 调用后，会经历两个步骤：1.内核等待 I/O 设备准备好数据，2.内核将数据从内核空间拷贝到用户空间。

JAVA中3种常见的IO模型：

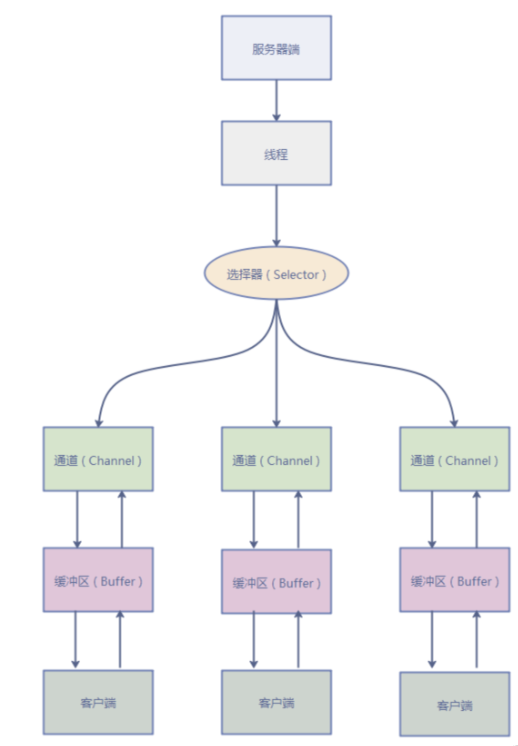
* 同步阻塞IO模型(BIO)：应用程序发起 read 调用后，会一直阻塞，直到内核把数据拷贝到用户空间。缺点：难以应对高并发。



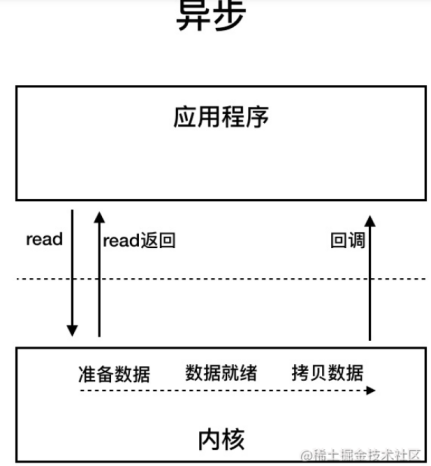
* 多路复用IO模型(NIO)：支持面向缓冲的，基于通道的 I/O 操作方法。 对于高负载、高并发的应用，应使用 NIO 。线程首先通过发起 select 调用，询问内核数据是否准备就绪，等内核把数据准备好了，用户线程再发起 read 调用。read 调用的过程（数据从内核空间 -> 用户空间）还是阻塞的。



NIO中的选择器（多路复用器）概念：使得一个线程就可以管理多个客户端的通道，根据通道现在的情况，选择向哪一个通道写数据或读数据。



* 异步IO模型(AIO)：基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。



##### Java线程池的实现原理，keepAliveTime等参数的作用。

##### HTTP连接池实现原理

##### 数据库连接池实现原理

##### 数据库的实现原理

## 技术框架

1. 看过哪些开源框架的源码
2. 为什么要用Redis，Redis有哪些优缺点？Redis如何实现扩容？
3. Netty是如何使用线程池的，为什么这么使用
4. 为什么要使用Spring，Spring的优缺点有哪些
5. Spring的IOC容器初始化流程
6. Spring的IOC容器实现原理，为什么可以通过byName和ByType找到Bean
7. Spring AOP实现原理
8. 消息中间件是如何实现的，技术难点有哪些
9. Zookeeper实现原理，以及选主算法
10. 为什么需要配置中心，配置中心如何实现的

## 系统架构

1. 如何搭建一个高可用系统
2. 哪些设计模式可以增加系统的可扩展性
3. 介绍设计模式，如模板模式，命令模式，策略模式，适配器模式、桥接模式、装饰模式，观察者模式，状态模式，访问者模式。
4. 抽象能力，怎么提高研发效率。
5. 什么是高内聚低耦合，请举例子如何实现
6. 什么情况用接口，什么情况用消息
7. 如果AB两个系统互相依赖，如何解除依赖
8. 如何写一篇设计文档，目录是什么
9. 什么场景应该拆分系统，什么场景应该合并系统
10. 系统和模块的区别，分别在什么场景下使用

## 分布式系统

1. 分布式事务，两阶段提交。
2. 如何实现分布式锁
3. 如何实现分布式Session
4. 如何保证消息的一致性
5. 负载均衡
6. 正向代理（客户端代理）和反向代理（服务器端代理）
7. CDN实现原理
8. 怎么提升系统的QPS和吞吐量
9. DNS的实现原理
10. 介绍下PAXOS协议
11. 介绍下Zookeeper的ZAB协议，如何选举LEADER？如何

## 实战能力

1. 有没有处理过线上问题？出现内存泄露，CPU利用率标高，应用无响应时如何处理的。
2. 开发中有没有遇到什么技术问题？如何解决的
3. 如果有几十亿的白名单，每天白天需要高并发查询，晚上需要更新一次，如何设计这个功能。
4. 新浪微博是如何实现把微博推给订阅者
5. Google是如何在一秒内把搜索结果返回给用户的。
6. 12306网站的订票系统如何实现，如何保证不会票不被超卖。
7. 如何实现一个秒杀系统，保证只有几位用户能买到某件商品。
8. 缓存失效如何解决？
9. 从数据库查询10G的数据并加载到内存中？
10. 如何设计一个流控功能？

## 软能力

1. 如何学习一项新技术，比如如何学习Java的，重点学习什么
2. 有关注哪些新的技术
3. 工作任务非常多非常杂时如何处理
4. 项目出现延迟如何处理
5. 和同事的设计思路不一样怎么处理
6. 如何保证开发质量
7. 职业规划是什么？短期，长期目标是什么
8. 团队的规划是什么
9. 能介绍下从工作到现在自己的成长在那里