Threadlocal类，注意哪些问题（显式关闭）
Atomic原子类；
怎样读取文件，注意哪些问题；
CAP原理（一致性，可用性，扩展性）
java调试经验，工具

数据库
建立联合索引（a,b,c），访问a，ab，abc可以命中索引；
是否了解分库分表；

cpu密集型和io密集型所对应的线程池，哪一种的线程应该更多？

cpu密集型尽量使用较小的线程池，一般为CPU的核心数+1，因为cpu密集型任务cpu使用率很高，如开过多的线程，只能增加线程上下文的切换次数，带来额外开销。
IO密集型：可以使用交大的线程池，一般cpu核心数*2，IO密集型cpu使用率不高，可以让cpu等待io的时候处理别的任务，充分利用cpu时间。

jvm的参数设置，包括哪些参数？

默认的java虚拟机的大小比较小，在对大数据进行处理时java就会报错：java.lang.OutOfMemoryError。

a、参数解释：

-Xmx

Java Heap（堆）最大值，默认值为物理内存的1/4，最佳设值应该视物理内存大小及计算机内其他内存开销而定；

-Xms

Java Heap初始值，Server端JVM最好将-Xms和-Xmx设为相同值，开发测试机JVM可以保留默认值；

-Xmn

Java Heap Young区（年轻代）大小；

-Xss

每个线程的Stack大小；

-XX:PermSize

JVM初始分配的非堆内存大小，默认是物理内存的1/64。

-XX:MaxPermSize

JVM最大分配的非堆内存大小，默认是物理内存的1/4。

-XX:NewSize

JVM初始分配的新生代堆区域内存大小。

-XX:MaxNewSize

JVM最大分配的新生代堆区域内存大小。

-XX:ReservedCodeCacheSize

编译代码时的缓存空间大小。

java的线程池，涉及哪几种参数，用什么队列？

corePoolSize:线程池大小，线程池中的线程数量。线程池创建之后不会立即去创建线程，而是等待线程的到来。当当前执行的线程大于改值时，线程会加入到缓冲队列中。

maxmumPoolSize：线程池中创建的最大线程数

keepAliveTime：空闲的线程多久时间后被销毁。当线程池线程数量超过corePoolSize时，多余的空闲线程的存活时间。即超过corePoolSize的空闲线程，在多长时间内，会被销毁。

unit:TimeUnit枚举类型的值，代表keepAliveTime时间单位，可以取以下几个值

TimeUnit.DAYS//天

TimeUnit.HOURS TimeUnit.MINUTES TimeUnit.SECONDS TimeUnit.MILLISECONDS

TimeUnit.MICOSECONDS TimeUnit.NANOSECONDS

workQueue:阻塞队列，用来存储等待执行的任务，决定了线程池的排队策略，有以下值。

ArrayBlockingQueue; LinkedBlockingQueue; SynchronousQueue

threadFactory:线程工程，用来创建线程。

handler:线程拒绝策略。当创建的线程超出maximumPoolSize,且缓冲队列已满时，新任务会被拒绝，有以下取值：

ThreadPoolExecutor.AbortPolicy：丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy: 也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy: 丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy: 由调用线程处理该任务。

2.LRU缓存的实现

LRU的翻译就是“最近最少使用”，LRU缓存就是使用这种原理实现，简单的说就是缓存一定量的数据，当超过设定的阈值时就把一些过期的数据删除掉，比如我们缓存10000条数据，当数据小于10000时可以随意添加，当超过10000时就需要把新的数据添加进来，同时要把过期数据删除，以确保我们最大缓存10000条，怎么确定删除哪条过期数据了，采用LRU算法实现的话就是将最老的数据删除。java中实现LRU缓存通常有两种方法，一种是LinkedHashMap，另一种是自己设计数据结构，使用链表+hashMap

1

LinkedHashMap的使用：

LHM有可以按两种顺序读取：

1. 按照存储顺序读取；构造方法内，参数accessOrder=false

在存储顺序读取模式下，可以实现FIFO缓存机制；

2. 按照访问顺序存取；构造方法内，参数accessOrder=true

按照访问顺序存取即最后一次访问的元素会自动置于表头；

若元素不存在，调用addEntry()方法；

若元素已经存在，调用recordEntry()方法，实现原理是先将最先访问的元素移除，再添加到表头来实现；按照访问顺序读取，可以实现LRU缓存（least recently used)。

LinkedHashMap有一个方法：

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest){
    return false;
}
```

该方法默认返回false，若实现LRU缓存，可以重写此方法，改变其逻辑，当map的size大于capacity时，返回true，则在容量满时，会自动移除访距访问时间最久的元素。

```java
/*
 * 这个问题是利用LinkedHashMap实现LRU缓存机制。
 */
public class LRUCache<K, V> {
     //这个是加载因子，加载因子是hash表中元素的填满程度。若加载因子越大，填满的元素就越多，好多是空间利用率提高
    //但冲突的机会大了，反之，加载因子越小，填满的元素越少，好处是冲突的机会减少，但空间浪费多了
    private static final float hashTableLoadFactor = 0.75f;

    private LinkedHashMap<K, V> map;
    private int cacheSize;

    public LRUCache(int cacheSize) {
        this.cacheSize = cacheSize;
        //ceil是向上取整
        int hashTableCapacity = (int) (Math.ceil(cacheSize / hashTableLoadFactor) + 1);
        map = new LinkedHashMap<K, V>(hashTableCapacity, hashTableLoadFactor, true){

            private static final long serialVersionUID = 1L;
            //因为在源码中的removeEldestEntry没有写return这个，所以需要加入。意思是如果map的尺寸大于设定的最大长度，返回tru
            //再新加入对象时删除最老的对象
            @Override
            protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
                return size() > LRUCache.this.cacheSize;
            }
        };
    }

    public synchronized V get(K key) {
        return map.get(key);
    }

    public synchronized void put(K key, V value) {
        map.put(key, value);
    }

    public synchronized void clear() {
        map.clear();
    }

    public synchronized int usedEntries() {
        return map.size();
    }
```

```
    public synchronized int usedEntries() {
        return map.size();
    }

    public synchronized Collection<Map.Entry<K, V>> getAll() {
        return new ArrayList<Map.Entry<K,V>>(map.entrySet());
    }

    public static void main(String [] args) {
        LRUCache<String, String> c = new LRUCache<>(3);
        c.put("1", "one");   //1
        c.put("2", "two");   //2 1
        c.put("3", "three"); //3 2 1
        c.put("4", "four");  //4 3 2
        if(c.get("2") == null) //2 4 3
            throw new Error();
        c.put("5", "five");  //5 2 4
        c.put("4", "second four"); //4 5 2
        if(c.usedEntries() != 3)    throw new Error();
        if(!c.get("4").equals("second four")) throw new Error();
        if(!c.get("5").equals("five")) throw new Error();
        if(!c.get("2").equals("two")) throw new Error();

        for(Map.Entry<String, String> e : c.getAll())
            System.out.println(e.getKey() + " : " + e.getValue());

    }
}
```

2.将cache的所有位置都用双链表连接起来，当一个位置被命中后，就将通过调整链表的指向，将该位置调整到链表头的位置，新加入的cache直接加到链表头好重，这样，在多次进行cache操作后，最近被命中的，就会向链表头方向移动，而没有命中的，向链表后面移动，链表尾则表示最近最少使用的cache。当需要替换内容时，链表的最后位置就是最少被命中的位置，我们只需要淘汰链表最后的部分即可。

注：此实现为非线程安全，若在多线程环境下使用需要在相关方法上添加synchronized以实现线程安全操作

```
public class LRUCache {


        private int cacheSize;
        private Hashtable<Object, Entry> nodes;//缓存容器
        private int currentSize;
        private Entry first;//链表头
        private Entry last;//链表尾

        public LRUCache(int i) {
                currentSize = 0;
                cacheSize = i;
                nodes = new Hashtable<Object, Entry>(i);//缓存容器
        }
```

```
/**
 * 获取缓存中对象,并把它放在最前面
 */
public Entry get(Object key) {
        Entry node = nodes.get(key);
        if (node != null) {
                moveToHead(node);
                return node;
        } else {
                return null;
        }
}

/**
 * 添加 entry到hashtable, 并把entry
 */
public void put(Object key, Object value) {
        //先查看hashtable是否存在该entry, 如果存在，则只更新其value
        Entry node = nodes.get(key);

        if (node == null) {
                //缓存容器是否已经超过大小.
                if (currentSize >= cacheSize) {
                        nodes.remove(last.key);
                        removeLast();
                } else {
                        currentSize++;
                }
                node = new Entry();
        }
        node.value = value;
        //将最新使用的节点放到链表头，表示最新使用的.
        moveToHead(node);
        nodes.put(key, node);
```

```
    }

    /**
     * 将entry删除, 注意：删除操作只有在cache满了才会被执行
     */
    public void remove(Object key) {
        Entry node = nodes.get(key);
        //在链表中删除
        if (node != null) {
            if (node.prev != null) {
                node.prev.next = node.next;
            }
            if (node.next != null) {
                node.next.prev = node.prev;
            }
            if (last == node)
                last = node.prev;
            if (first == node)
                first = node.next;
        }
        //在hashtable中删除
        nodes.remove(key);
    }

    /**
     * 删除链表尾部节点，即使用最后 使用的entry
     */
    private void removeLast() {
        //链表尾不为空,则将链表尾指向null. 删除连表尾（删除最少使用的缓存对象）
        if (last != null) {
            if (last.prev != null)
                last.prev.next = null;
            else
                first = null;
```

```
                last = last.prev;
            }
        }

        /**
         * 移动到链表头，表示这个节点是最新使用过的
         */
        private void moveToHead(Entry node) {
            if (node == first)
                return;
            if (node.prev != null)
                node.prev.next = node.next;
            if (node.next != null)
                node.next.prev = node.prev;
            if (last == node)
                last = node.prev;
            if (first != null) {
                node.next = first;
                first.prev = node;
            }
            first = node;
            node.prev = null;
            if (last == null)
                last = first;
        }
        /*
         * 清空缓存
         */
        public void clear() {
            first = null;
            last = null;
            currentSize = 0;
        }

}
```

```
class Entry {
        Entry prev;//前一节点
        Entry next;//后一节点
        Object value;//值
        Object key;//键
}
```

grep，awk是干什么的

grep主要用于搜索某些字符串、而awk用于处理文本。？？？？

从浏览器访问url到打开页面经历的过程

https://blog.csdn.net/lzghxjt/article/details/51458540

有了客户端和服务器，就可以开始通信了，整体分为3个步骤：

1.因为http是建立在tcp之上，那么要经过3次握手创建连接。

2.创建连接后，服务器会根据url请求中的信息作出处理，作出响应。

3.客户端得到响应后，进行渲染。

创建连接：对于http的客户端，它的输入是一个url，而对于创建连接，它需要的只是url的host（主机）部分，而主机地址一般是网站的域名，所以第一步是域名解析，也就是要通过DNS服务器进行域名解析得到网站的ip地址，然后向这个ip地址发送一个连接建立的请求，如果服务器接收到请求会返回一个确认，客户端得到确认再次发送确认，连接建立成功，涉及到很多东西。

服务器处理：

建立好连接后，客户端就会发送http请求，请求信息包含一个头部和一个请求体。

一般的web技术都会把请求进行封装然后交给我们的服务器进行处理，比如servlet会把请求封装成httpserverletrequest对象，把响应封装成httpserveletResponse对象。

在请求对象中我们可以得到path，queryString，body（提交的数据）等。

我们可以根据path找到客户端想要的文件，读取这个文件，然后通过响应对象把内容返回给客户端。

客户端渲染：

从浏览器的角度讲，它包含几大组件，网络功能（比如http的实现）算是其中之一，渲染引擎也是其中之一，还有其它的一些比如自己UI界面，javascript解释器，客户端数据存储等等。在这里我们主要关注渲染引擎和javascript解释器，对于web开发者来说，这才是浏览器的核心。