副本 门徒计划–老詹专场01

课程主题

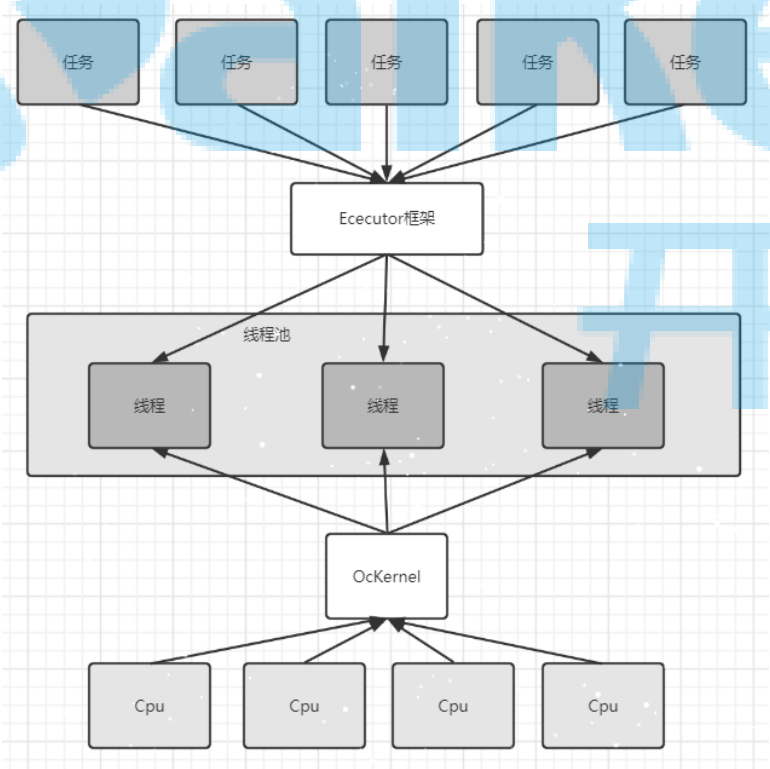
JDK线程池与阻塞队列、面试分享

Executor框架简介

1. Java任务调度模型

Java任务调度的两级调度模型：

1. 在用户层，应用程序被分解为若干个任务，这些任务被用户级的调度执行器（Executor框架）映射为固定数量的线程。
2. 在操作系统底层，操作系统内核会将这些线程映射到CPU上。



2. Executor框架

Executor框架主要成员的介绍：

- **Runnable接口：要执行的任务（Task）,**Runnable接口不会返回结果
- **Callable接口：要执行的任务（Task）,**可以返回结果；

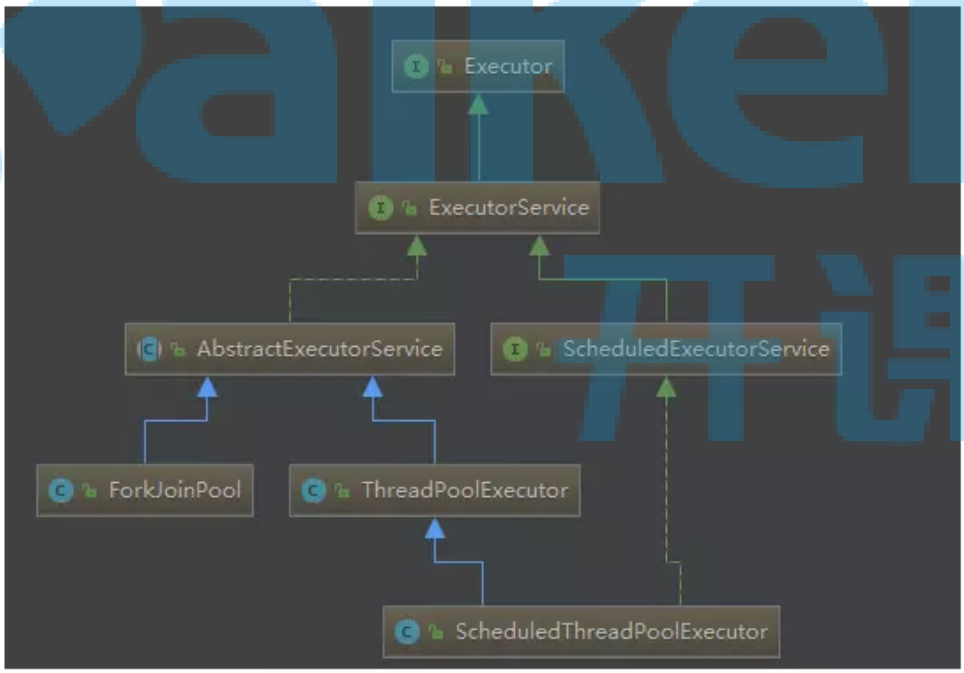
- **Executor接口**：Executor框架的基础，将任务的提交和任务的执行分离。
  - **ThreadPoolExecutor类**：Java线程池的核心实现类，用来执行提交的任务。
  - **ScheduledThreadPoolExecutor类**：Executor框架的另一个关键类，可以在给定的延迟后执行命令，或者定期执行命令。
- **还有Future接口、FutureTask类、Runnable接口、Callable接口**。

Executor框架主要由三部分组成：

- 任务： 被执行任务需要实现**Runnable接口**或者**Callable接口**。
  a. Runnable接口和Callable接口都可以被ThreadPoolExecutor和ScheduledThreadPoolExecutor执行。
  b. Runnable对象可以通过Executors工具类中的Executors.callable(Runnable task)或者Executors.callable(Runnable task, Object result)转化为Callable对象。

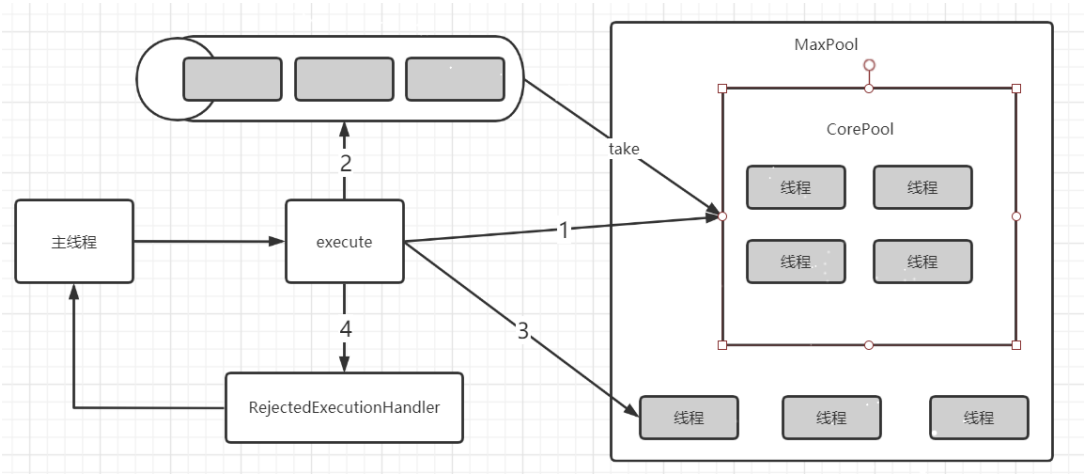
- 执行器： 包括**任务执行机制的核心接口 Executor** ，以及继承自Executor 接口的**ExecutorService接口**。Executor框架有两个关键类ThreadPoolExecutor类和ScheduledThreadPoolExecutor类，实现了ExecutorService接口。

  a. 其中，ScheduledThreadPoolExecutor类继承了ThreadPoolExecutor类，并实现了ScheduledExecutorService接口。
  b. 而ScheduledExecutorService接口又实现了ExecutorService接口。



- 异步执行结果： Future接口和实现了Future接口的FutureTask类。
  a. 将实现Runnable接口或者Callable接口的任务对象（通过ExecutorService.submit()方法）提交给ThreadPoolExecutor或者ScheduledThreadPoolExecutor去执行，会返回一个FutureTask对象。

3. Executor框架的使用流程图



线程池的实现原理分析

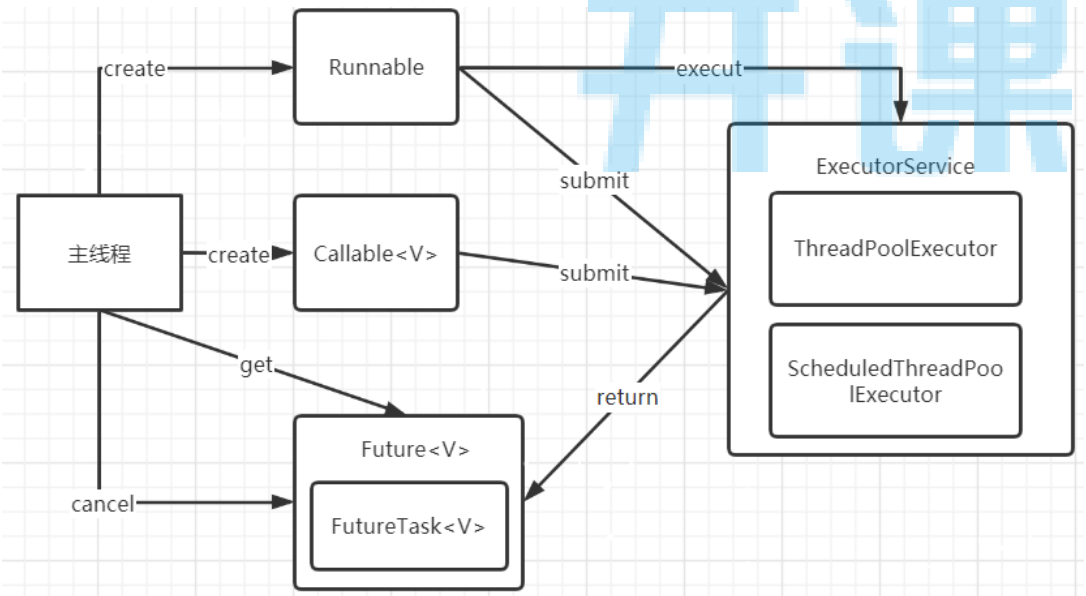
用户通过submit提交一个任务。线程池会执行如下流程:

判断当前运行的worker数量是否超过corePoolSize,如果不超过corePoolSize。就创建一个worker直接执行该任务。—— 线程池最开始是没有worker在运行的

如果正在运行的worker数量超过或者等于corePoolSize,那么就将该任务加入到workQueue队列中去。

如果workQueue队列满了,就检查当前运行的worker数量是否小于maximumPoolSize,如果小于就创建一个worker直接执行该任务。

如果当前运行的worker数量大于等于maximumPoolSize，那么就执行RejectedExecutionHandler来拒绝这个任务的提交。



- 主线程首先创建实现Runnable或者Callable接口的任务对象。
  a. 工具类Executors可以实现将Runnable对象封装成一个Callable对象，方法：Executors.callable(Runnable task)或Executors.callable(Runnable task，Object result)。

- 然后可以把创建完成的Runnable对象直接交给ThreadPoolExecutor或者ScheduledThreadPoolExecutor执行，方法：ThreadPoolExecutor.execute(Runnable command)或者ScheduledThreadPoolExecutor.execute(Runnal command)；或者也可以把Runnable对象或Callable对象提交给ExecutorService执行，方法：ExecutorService.submit(Runnable task)或ExecutorService.submit(Callable task)。

- **执行execute()方法和submit()方法的区别是什么呢?**
  - execute()方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功；
  - submit()方法用于提交需要返回值的任务，线程池会返回一个FutureTask类型的对象，通过这个FutureTask对象可以判断任务是否执行成功。
  - 对于需要要返回值的任务，可以通过FutureTask.get()方法或者FutureTask.get(long timeout, TimeUnit unit)方法来获取返回值。其中，前者会阻塞当前线程直到任务完成；后者，会阻塞当前线程一段时间后立即返回，有可能任务并没有执行完成。

- 最后，主线程可以执行FutureTask.get()方法来等待任务执行完成。主线程也可以执行FutureTask.cancel(boolean mayInterruptIfRunning) 来取消任务的执行。

Java常见的线程池

创建线程池的方式1

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
}

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), handler);
}
```

序号	名称	类型	含义
1	corePoolSize	int	核心线程池大小
2	maximumPoolSize	int	最大线程池大小
3	keepAliveTime	long	线程最大空闲时间
4	unit	TimeUnit	时间单位
5	workQueue	BlockingQueue<Runnable>	线程等待队列
6	threadFactory	ThreadFactory	线程创建工厂
7	handler	RejectedExecutionHandler	拒绝策略

- corePoolSize：指定了线程池中的线程数量，它的数量决定了添加的任务是开辟新的线程去执行，还是放到workQueue任务队列中去；
- maximumPoolSize：指定了线程池中的最大线程数量，这个参数会根据你使用的workQueue任务队列的类型，决定线程池会开辟的最大线程数量；
- keepAliveTime：当线程池中空闲线程数量超过corePoolSize时，多余的线程会在多长时间内被销毁；
- unit：keepAliveTime的单位
- workQueue：任务队列，被添加到线程池中，但尚未被执行的任务；它一般分为直接提交队列、有界任务队列、无界任务队列、优先任务队列几种；
- threadFactory：线程工厂，用于创建线程，一般用默认（DefaultThreadFactory）即可；
- handler：拒绝策略；当任务太多来不及处理时，如何拒绝任务；
    - ThreadPoolExecutor.AbortPolicy()： 抛出RejectedExecutionException异常，默认策略
    - ThreadPoolExecutor.CallerRunsPolicy(): 会用调用execute函数的上层线程去执行被拒绝的任务
    - ThreadPoolExecutor.DiscardOldestPolicy(): 抛弃最旧的任务，再把这个新任务添加到队列
    - ThreadPoolExecutor.DiscardPolicy(): 抛弃当前的任务
    - 自定义策略实现RejectedExecutionHandler接口，来不及处理的任务保存到MQ

创建线程池的方式2

FixedThreadPool

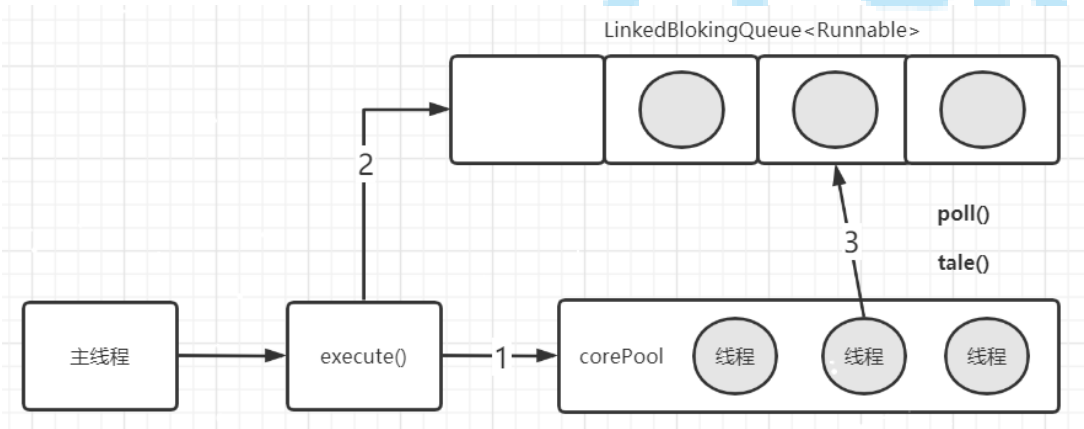
- **使用固定线程数的线程池**，适用于为了满足资源管理的需求，而限制当前线程数的应用场景。或者是适用于负载比较重的服务器。
- 创建方式：Executors.newFixedThreadPool(threadsNum)。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(
                  nThreads,
                  nThreads,
                  0L,
                  TimeUnit.MILLISECONDS,
                  new LinkedBlockingQueue<Runnable>());
}
```

- 具有以下特点：
  a. corePoolSize = maximumPoolSize = newFixedThreadPool(num)方法的参数num，实际只使用了corePool。
  b. keepAliveTime的值为0L，表明工作线程如果空闲会被立即终止。
  c. 使用LinkedBlockingQueue，这是一种基于链表的无界队列，因为队列容量为Integer.MAX_VALUE。队列中的元素采用FIFO序。
  d. 使用无界队列，FixedThreadPool在未调用shutdown()或者shutdownNow()方法的情况下，不会调用饱和策略。

- 关于无界队列：
  a. 当corePoolSize满时，由于使用无界队列，新提交的任务总是可以存放在阻塞队列中。因此，线程池中的线程数永远不会超过corePoolSize。
  b. 由于1，使用无界队列使得线程池中的maximumPoolSize参数无效。
  c. 由于1和2，使用无界队列使得线程池中的keepAliveTime参数无效。
  d. 由于使用无界队列，FixedThreadPool在未调用shutdown()或者shutdownNow()方法的情况下，不会调用饱和策略（即不会RejectExecutionHandler.rejectExecution()方法）。

- **FixedThreadPool的工作流程：**



FixedThreadPool 的 execute() 运行示意图

  a. 如果当前运行的线程数小于corePoolSize，则创建新的线程来执行任务；
  b. 当前运行的线程数等于corePoolSize，将任务加入LinkedBlockingQueue；

c. 工作线程执行完当前的任务后，不会销毁，而是循环地从LinkedBlockingQueue中获取任务来执行；

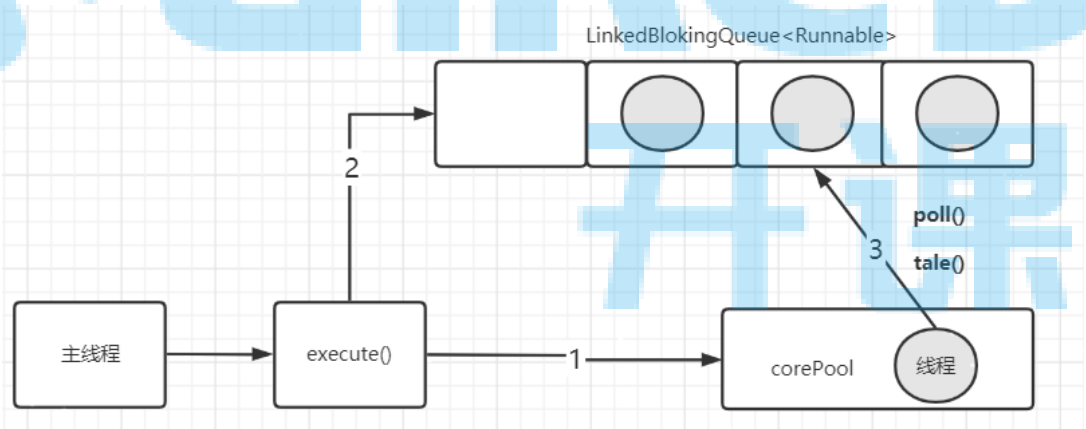
SingleThreadExecutor

- **使用单个线程的线程池**，适用于需要保证各个任务顺序执行，而且在任意时刻，不会有多个活动线程的应用场景。
- 创建方式：Executors.newSingleThreadExecutor()。

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

- 具有以下特点：
  a. corePoolSize = maximumPoolSize = 1，实际只使用了corePool。
  b. keepAliveTime的值为0L，表明工作线程如果空闲会被立即终止。
  c. 使用LinkedBlockingQueue，基于链表的无界队列，队列中的元素采用FIFO顺序。
  d. 使用无界队列，SingleThreadExecutor在未调用shutdown()或者shutdownNow()方法的情况下，不会调用饱和策略。

- SingleThreadExecutor的工作流程：



SingleThreadExecutor 的 execute() 运行示意图

  a. 如果线程池中无运行的线程，则创建一个新的线程执行任务。
  b. 如果线程池中已经有正在运行的线程，则将任务加入到LinkedBlockingQueue中。
  c. 工作线程执行完当前的任务后，不是立即销毁，而是循环地从LinkedBlockingQueue中取出任务来执行。

- 与FixedThreadPool的区别：SingleThreadExecutor线程池中同时最多只有一个线程活跃，同时最多只有一个任务在执行。

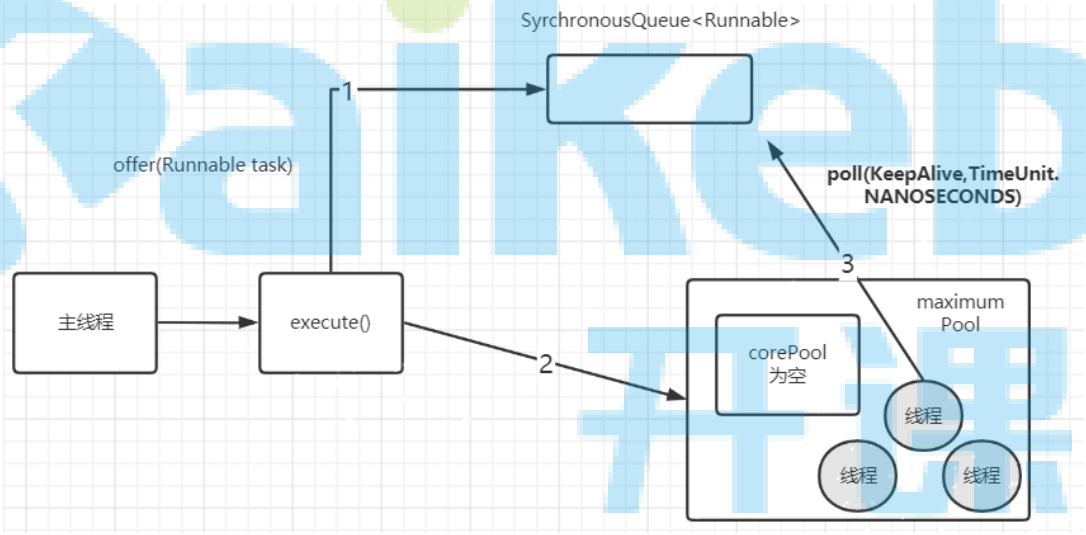
CachedThreadPool

- **根据需要创建新线程的线程池**，适用于执行很多短期异步任务的小程序，或者是负载比较轻的服务器。
- 创建方式：Executors.newCachedThreadPool()。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

- 可以根据需要创建新线程的线程池，具有以下特点：
  a. corePoolSize = 0，maximumPoolSize = Integer.MAX_VALUE，表明线程池无界，实际只使用了maximumPool。
  b. keepAliveTime的值为60s，表明工作线程最多允许空闲60s，超过60s会被终止。
  c. 使用SynchronousQueue，是一种没有容量的阻塞队列，主线程的每个添加操作必须等待另一个线程的移除操作，反之亦然。
  d. CachedThreadPool在线程池中线程数等于maximumPoolSize时，会调用饱和策略。

- CachedThreadPool的工作流程：



CachedThreadPool 的 execute() 运行示意图

  a. 主线程首先执行Synchronous.offer()，即添加操作。如果当前maximumPool中有空闲线程正在执行SynchronousQueue.poll(keepAliveTime,TimeUnit.NANOSECONDS)（即移除操作），那么主线程的offer操作和空闲线程的poll操作配对成功，主线程把任务交给空闲线程执行。
  b. 如果maximumPool为空，或者maximumPool没有空闲线程，则offer操作和poll操作匹配失败。此时，CachedThreadPool会创建新的线程执行该任务。
  c. 工作线程执行完当前线程后，会执行SynchronousQueue.poll(keepAliveTime,TimeUnit.NANOSECONDS)，这个poll操作会让空闲线程等待60秒。如果60秒内主线程发起了offer操作，这个空闲线程便会执行主线程新提交的任务；否则，这个空闲线程将会终止。

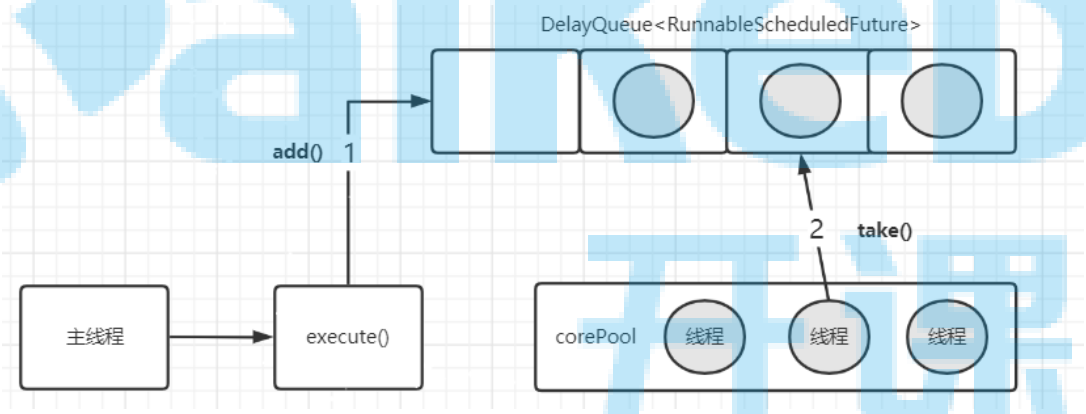
ScheduledThreadPoolExecutor

- **使用单个线程的线程池**，适用于需要使用单个后台线程执行周期任务，同时需要保证各个任务执行顺序的应用场景。
- 创建方式：Executors.newSingleThreadScheduledExecutor()。

```
public static ScheduledExecutorService newSingleThreadScheduledExecutor()
{
    return new DelegatedScheduledExecutorService
        (new ScheduledThreadPoolExecutor(1));
}
```

- 有固定数量的线程池，可以在给定的延迟后执行任务，或者定期执行任务。具有以下特点：
  a. corePoolSize = Executors.newScheduledThreadPool(**num**)中的参数，ScheduledThreadPool的参数大于等于1，而SingleThreadScheduledExecutor的参数为1。maximumPoolSize = Integer.MAX_VALUE，实际只使用了corePool。
  b. keepAliveTime的值为0L，表明线程一旦空闲就会被终止。
  c. 使用DelayQueue，这是一个具有优先级的无界队列，内部封装了一个PriorityQueue，可以实现任务的定期执行和延后执行。
  d. 使用无界队列，ScheduledThreadPool在未调用shutdown()或者shutdownNow()方法的情况下，不会调用饱和策略。

- **ScheduledThreadPool的运行流程**



ScheduledThreadPoolExecutor 的 execute() 运行示意图

  a. 当调用ScheduledThreadPoolExecutor的 scheduleAtFixedRate() 方法或者scheduleWithFixedDelay() 方法时，会向DelayQueue 添加一个实现了RunnableScheduledFuture 接口的 ScheduledFutureTask 。
  b. 线程池中的线程从DelayQueue中获取ScheduledFutureTask，然后执行任务。

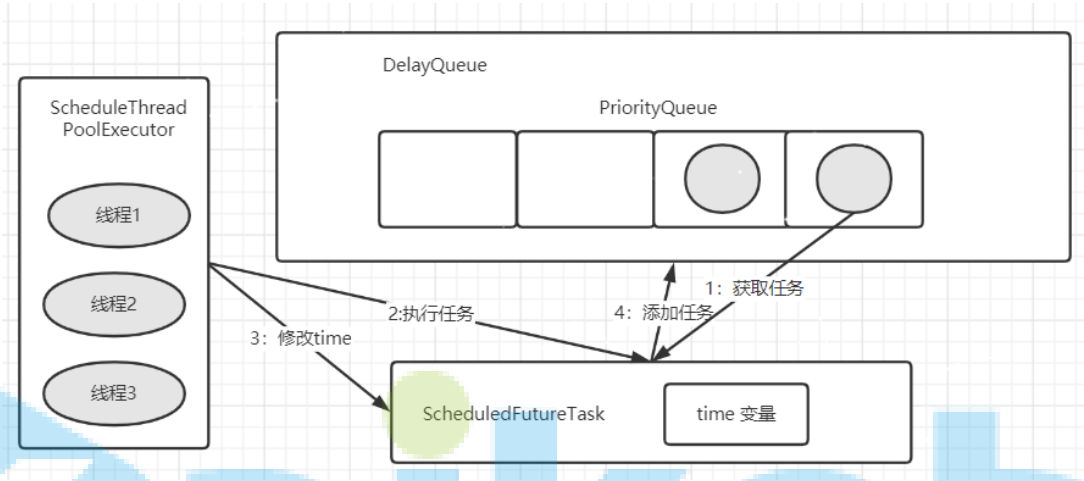
- **与ThreadPoolExecutor的区别**：
  a. 使用DelayQueue作为阻塞队列
  b. 获取任务的方式不同
  c. 执行周期任务后，添加了额外的处理

执行任务周期的步骤，或者如何实现任务周期执行的?

- 预备知识：ScheduledFutureTask中的三个重要的成员变量：

a. **long time**： 表示这个任务将要被执行的具体时间。当time大于等于当前系统的时间，表明该任务已到期，可以执行。
b. **long sequenceNumber**： 表示这个任务被添加到ScheduledThreadPoolExecutor的序号。
c. **long period**： 表示任务执行的时间间隔

- DelayQueue中封装的PriorityQueue，会对这些任务按time进行排序，时间小的先被执行。如果time相同，则按seq进行排序，序号较小的先执行。（即如果任务的time相同，则先提交的先被执行）



ScheduledThreadPoolExecutor执行任务周期的步骤

a. 线程1使用DelayQueue.take()方法从DelayQueue中获取已经到期的ScheduledFutureTask。到期任务是指ScheduledFutureTask的time小于等于当前系统的时间。

```
if (first == null || first.getDelay(NANOSECONDS) > 0)
return null;
```

b. 线程1执行这个ScheduledFutureTask。
c. 线程1修改ScheduledFutureTask中的time为下次将要被执行的时间。
d. 线程1将修改了time后的ScheduledFutureTask使用DelayQueue.add()方法，放回到DelayQueue中。

JDK递归与JVM栈原理分析

LRU缓存算法之JDK源码实现

设计并实现LRU缓存算法

LeetCode题目： 146. LRU 缓存机制

JDK LRU实现分析：LinkedHashMap

1. 基本属性

```
  //双链表的头
  transient LinkedHashMap.Entry<K,V> head;
  //双链表的尾
  transient LinkedHashMap.Entry<K,V> tail;
  //排序模式
  //true:add或get等操作后 节点将移动到双链表尾部
  //false:默认值 不做任何改变
  final boolean accessOrder;
  // 节点类
  static class Entry<K,V> extends HashMap.Node<K,V> {
        Entry<K,V> before, after;//分别指向上一个节点和下一个节点
        Entry(int hash, K key, V value, Node<K,V> next) {
            super(hash, key, value, next);
        }
    }
```

2. 构造方法

```
//传入数组的长度和加载因子
public LinkedHashMap(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor);
        accessOrder = false;
}

//只声明数组长度，
public LinkedHashMap(int initialCapacity) {
        super(initialCapacity);
        accessOrder = false;
}


//均使用默认值
public LinkedHashMap() {
        super();
        accessOrder = false;
}


//直接使用另外的map初始化
public LinkedHashMap(Map<? extends K, ? extends V> m) {
        super();
        accessOrder = false;
        putMapEntries(m, false);
}

//指定数组长度，加载因子，并且会根据accessOrder的值来确定输出顺序
 public LinkedHashMap(int initialCapacity,
                          float loadFactor,
```

```
                          boolean accessOrder) {
        super(initialCapacity, loadFactor);
        this.accessOrder = accessOrder;
    }
```

3.put方法

该方法，并没有被重写，而是复用了 hashMap的方法

```
public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
}


static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

所以新增操作都是调用hashMap的方法,所以如果有不懂的同学，可以自行学习hashMap的源码，在此不再赘述。

3.1插入过程中生成新的节点的方法

```
 Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {

        //生成一个新的节点，然后将它移动到最后
        LinkedHashMapEntry<K,V> p =
            new LinkedHashMapEntry<K,V>(hash, key, value, e);
        linkNodeLast(p);
        return p;
    }


private void linkNodeLast(LinkedHashMapEntry<K,V> p) {

        //临时变量指向双向链表的尾部节点
        LinkedHashMapEntry<K,V> last = tail;
        tail = p;
        //如果尾部节点为空，那么就是第一个节点。这个时候新加入的数据赋给头部节点
        if (last == null)
            head = p;
        else {
         //否则将p添加到末尾
            p.before = last;
            last.after = p;
        }
    }
```

3.2回调方法

HashMap 中的 put 方法，其中涉及到afterNodeAccess和afterNodeInsertion两个方法。

afterNodeAccess(Node<K,V> e) 作用:将e移动到双链表的尾部。

不仅是put、add和get操作后都会调用该方法，默认不启用因为accessOrder为false

我们来详细看一下

afterNodeAccess

```
//将添加的元素移动到链表的尾端，一个简单的操作

//首先要明确这个函数的目的是将访问的这个元素移动到双向链表的尾端，并且要将
//他的后面的元素向前移动

void afterNodeAccess(Node<K,V> e) {
        //last为原链表的尾节点
        LinkedHashMapEntry<K,V> last;

        //如果accessOrder为true，并且尾端元素不是需要访问的元素
        if (accessOrder && (last = tail) != e) {

            //将节点e强制转换成linkedHashMapEntry,b为这个节点的前一个节点，a为
它的后一个节点

            LinkedHashMapEntry<K,V> p =
                (LinkedHashMapEntry<K,V>)e, b = p.before, a = p.after;

            //p为最后一个元素，那么他的后置节点必定是空
            p.after = null;

            //b为e的前置元素，如果b为空，说明此元素必定是链表的第一个元素，更新之
后
            //链表的头结点已经变成尾节点，那么原链表的第二个节点就要变为头结点
            if (b == null)
                head = a;
            else
                //如果b不是空，那么b的后置节点就由p变为p的后置节点
                b.after = a;

            //如果p的后置节点不为空，那么更新后置节点a的前置节点为b
            if (a != null)
                a.before = b;
            else
            //如果p的后置节点为空，那么p就是尾节点，那么更新last的节点为p的前置节
点
                last = b;
            //如果原来的尾节点为空，那么原链表就只有一个元素
            if (last == null)
                head = p;
            else {
```

```
            //更新当前节点p的前置节点为 原尾节点last， last的后置节点是p
                p.before = last;
                last.after = p;
            }

            //p为最新的尾节点
            tail = p;

            //修改modCount
            ++modCount;
        }
    }
```

其实就是换个位置,但是由于需要判断是否存在前后节点的问题,多了一些判断,而且它名字起的很随意,a.b.e.p只要知道是移位到双链表尾部就可以了,默认使用LinkedHashMap是不会开启这个功能的。

afterNodeInsertion

作用：用于在put()等操作后(主要是put)删除双链表的首节点，默认不启用,启用需要继承LinkedHashMap重写removeEldestEntry方法,需要注意removeEldestEntry()一定要写条件比如size超过100后允许删除,直接返回true会导致每次put都删除

```
//根据evict,也就是前文linkedHashMap构造函数中的accessOrder，来判断是否删除双向
//链表中最老的元素，这个是实现lru需要用到的。

void afterNodeInsertion(boolean evict) {
        LinkedHashMapEntry<K,V> first;
        //判断是否需要删除
        if (evict && (first = head) != null && removeEldestEntry(first)) {
            K key = first.key;
            removeNode(hash(key), key, null, false, true);
        }
    }

//根据源码中的注释，这个函数一般返回flase，但是当cache已满的情况下返回true
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
        return false;
    }
```

4.remove方法

remove(Object key)remove也是调用了HashMap的方法,流程都一样,所以这里重点关注afterNodeRemoval这个方法

```
//afterNodeRemoval()方法相对简单，就是在删除后处理其对应链表前后关系（刨掉一截）。
 void afterNodeRemoval(Node<K,V> e) { // unlink
        LinkedHashMapEntry<K,V> p =
            (LinkedHashMapEntry<K,V>)e, b = p.before, a = p.after;
        //待删除节点 p 的前置后置节点都置空
```

```
        p.before = p.after = null;
        //如果b为空，p就是头节点，删除之后a就是头节点，否则将前置节点b的后置节点指向a
        if (b == null)
            head = a;
        else
            b.after = a;
        //如果a为空，那么p就是尾节点，删除之后，b就是新的尾节点，否则a的前置节点就改为b
        if (a == null)
            tail = b;
        else
            a.before = b;
    }
```

5.get方法

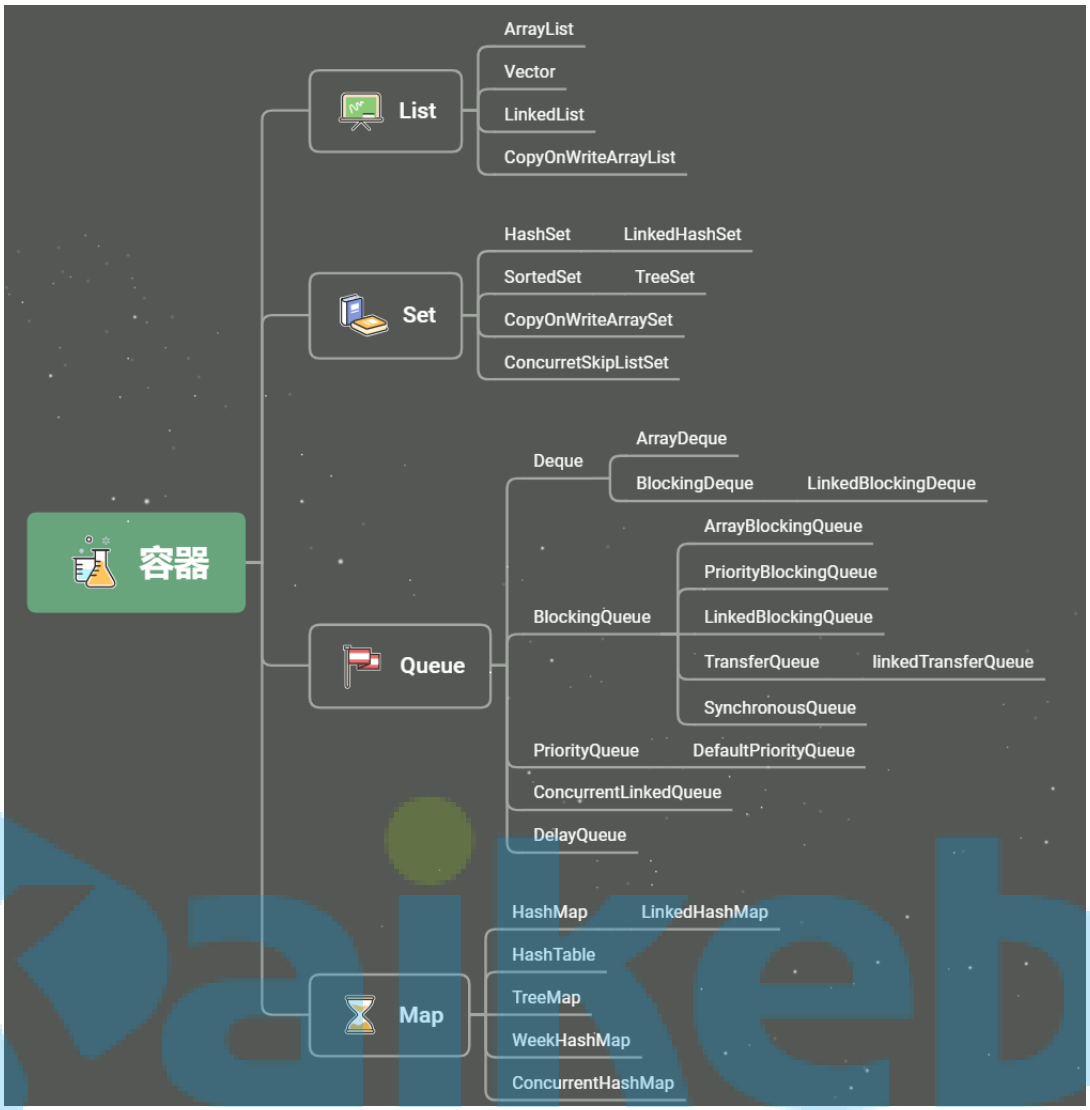
get方法虽然被重写了，但逻辑没变，只是多了afterNodeAccess方法，参见我们上面的分析。

```
public V get(Object key) {
        Node<K,V> e;
        //第一步是直接使用Hashmap中的函数getNode方法，获取value，如果为null，返回null
        if ((e = getNode(hash(key), key)) == null)
            return null;

        //如果构造函数accessOrder为true，那么就把方法的这个函数放置到双向链表的末尾。
        //这就是最新使用的元素。
        if (accessOrder)
            afterNodeAccess(e);
        return e.value;
    }
```

阻塞队列

阻塞队列顾名思义是阻塞性质的，在入队的时候发现队列满了就会一直阻塞在那里，直到队列里面有元素出队空出位置位置，出队的时候也是一个原理，如果发现队列空了，就会一直阻塞直到有元素入队了，就会把这个入队的元素进行出队操作，总而言之就是出队和入队操作如果无法执行，就会一直阻塞在那里直到可以执行为止。



ArrayBlockingQueue

底层使用数组实现的，默认不是一个公平队列(根据阻塞的先后顺序来访问队列)，不过可以在初始化的时候设置为公平队列，是一个有界队列，可以设置队列大小，例如

```
ArrayBlockingQueue Queue = new ArrayBlockingQueue(3,true)。
```

下面这个小例子就是阻塞队列的一个很好的应用

> 初始化一个长度为3的队列，在main方法里面启动一个子线程，子线程里面调用一个入队操作的方法，间隔时间是2秒一次，而又调用了一个arrayBlockingQueue()方法直接填满了队列，然后循环出队，时间间隔3秒。会发现一次打印1、2、3之后就开始打印后面入队的随机数，因为出队的间隔时间是3秒，而入队的间隔时间是2秒，而一开始塞满了队列，但是做入队操作也没有报错，这就是阻塞队列的特性，如果满了就一直阻塞住，整个过程就像排队一样，依次入队，依次出队。

```
public class Demo{
  static ArrayBlockingQueue arrayBlockingQueue = new
ArrayBlockingQueue(3);

  public static void main(String[] args) throws Exception {
      System.out.println(arrayBlockingQueue.size());

```

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    addQueue();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();

        arrayBlockingQueue();
    }

    public static void arrayBlockingQueue() throws Exception {
      arrayBlockingQueue.put(1);
      arrayBlockingQueue.put(2);
      arrayBlockingQueue.put(3);
      while (true) {
        TimeUnit.SECONDS.sleep(3);
        System.out.println(
        arrayBlockingQueue.size() + "-" + arrayBlockingQueue.take());
      }
    }

    public static void addQueue() throws Exception {
      while (true) {
        TimeUnit.SECONDS.sleep(2);
        arrayBlockingQueue.put(new Random().nextInt(100));
      }
    }
}
```

LinkedBlockingQueue

这个阻塞队列是基于链表实现的，也是一个有界队列，不过有默认大小，默认大小和最大长度都是Integer.MAX_VALUE。和ArrayBlockingQueue很相似，只不过底层控制并发阻塞的原理稍稍不同

PriorityBlockingQueue

这个队列也是基于数组实现，值得注意的是这个队列是无界队列，虽然初始化的时候可以设置初始值，但是一旦队列的长度超过的时候就会自动扩容，最重要的是这个队列是一个优先级队列，所以并不是先进先出(FIFO)的顺序，而是按照元素的优先级来进行出队，所以初始化的时候可以设置一个比较器，确保存入的元素是可以比较的，这样就可以根据元素的优先级进行出队了。

```
public static void priorityBlockingQueue() throws Exception{
```

```
  PriorityBlockingQueue queue =
      new PriorityBlockingQueue(5,new Comparator() {

      @Override
      public int compare(Integer o1, Integer o2) {
        return o2-o1;
      }
  });
  queue.add(10);
  queue.add(2);
  queue.add(5);

  while(true){
      System.out.println(queue.take());
  }
}
```

DelayQueue

这个队列也是一个优先级队列，但是它的优先级是以时间为准，放入队列中的元素一定要是Delayed类型，里面有两个方法，分别是compareTo()和getDelay()方法，第一个是设置比较规则，第二个则是判断时间是否已经到达，只有按照getDelay()方法中的代码判断时间已经到达了，这个时候获取队列尾部的元素才可以成功获取到，否则即使有元素也获取不到，也是一个无界队列，底层也是基于数组实现。

下面这个小例子模仿了订单超时的操作，超时时间设置的各有不同，可以发现，根据对比规则先把最早超时的那个订单出队了，依次出队，直到最晚超时的那个订单。

```
public static void delayQueue() throws Exception{
  DelayQueue queue = new DelayQueue();
  long time = System.currentTimeMillis()+10000;

  queue.put(new TestDelayed(1,time + 500));
  queue.put(new TestDelayed(2,time + 300));
  queue.put(new TestDelayed(3,time + 100));
  queue.put(new TestDelayed(4,time + 200));
  queue.put(new TestDelayed(5,time + 400));

  while (true){
      TestDelayed delayed = (TestDelayed)queue.take();
      System.out.println(delayed.orderNumber);
  }
}
// 自定义Delayed类
class TestDelayed implements Delayed{
  // 订单号
  int orderNumber;
  // 超时时间
```

```
    long expire;
    public TestDelayed(int orderNumber, long expire) {
        this.orderNumber = orderNumber;
        this.expire = expire;
    }

    // 超时时间减去当前时间，如果小于0就表示时间到了
    @Override
    public long getDelay(TimeUnit unit) {
        return expire-System.currentTimeMillis();
    }

    // 设置对比规则
    @Override
    public int compareTo(Delayed o) {
        long t1 = this.getDelay(TimeUnit.MILLISECONDS);
        long t2 = o.getDelay(TimeUnit.MILLISECONDS);
        return Long.compare(t1,t2);

    }
}
```

SynchronousQueue

这个队列它自身并没有任何容量，所以每次入队操作之后就必须进行一次出队操作，否则就会阻塞，底层是使用CAS来保证并发问题，适合调度派发任务的场景，生产者源源不断的做入队操作，多个消费者则进行出队操作，它也支持公平队列的设置，在初始化的时候可以进行设置，好处就是在一些调度任务频繁但是又不像使用无界队列就可以使用SynchronousQueue，而且性能更加好。

```
public static void synchronousQueue() throws Exception{
    SynchronousQueue queue = new SynchronousQueue();
    new Thread(new Runnable() {
        @Override
        public void run() {
            while (true) {
                try {
                    TimeUnit.SECONDS.sleep(1);
                    queue.put(new Random().nextInt(100));
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }).start();

    while (true){
```

```
        TimeUnit.SECONDS.sleep(1);
        System.out.println(queue.take());
    }
}
```

面试分享

		工作内容	福利	优点	顾虑	办公地点	薪资
蚂蚁金服	网商银行	做互联网银行服务，用户端和企业端都开发	没啥福利 就是企业背景好	业务复杂度很高 并发也很大 有蚂蚁金服这个牌子， 后期换工作比较方便	太远了，每天通勤2个多小时 幸福感比较差	平日9点，周末双休 地点：呼家楼的环球金融中心 通勤：1个小时	28K*16新=44.8万 基础薪资13个月 年终奖3个月 职级：P6+
快手	海外部门	东南亚方向，开发后台接口， 将快手短视频的业务80%搬迁到海外app上	福利好 三餐面试 住房补贴2000 周末加班双倍工资	业务复杂度比较高 并发：几万qps-几十万qps之间 对自己的技术成长很大	海外业务稳定性	大小周，晚上9点下班 在西二旗 通勤：25分钟	还没谈，offer已过
小米	小米商城	做小米商城做后端服务， 兼职小米之家的一部分订单	没啥福利	电商行业，业务复杂度错	公司偏硬件， 虽然是电商部门，但平日流量不大 也就双十一流量大点，能几万左右	不加班，周末双休 地点：西二旗 通勤：40分钟	30k*14薪=42万 基础薪资12个月 年终奖2个月 职级：16 （换算为阿里的P6）
猿辅导	斑马课程 教研工具	给老师开发教研工具， 主要偏向B端 还有一部分的C端接口	福利好，应该是这几家 福利最好的	轻松，福利好， 技术复杂段比较高，但是并发差	业务比较偏门 后期换工作可能不太好	不加班，周末双休 在望京 通勤：45分钟	还没谈，offer已过
百度	如流 知识图谱管理	一个类似于飞书和企业微信的 主要做类似于知识图	没啥福利 就是企业背景好		没啥技术挑战，并发也比较低， 需给私有化部署，可能去出差	不加班 在西二旗 通勤：30分钟	还没谈
贝壳	装修保洁等	就是做买房之后的，装修保洁等后续服务，类似于订单啥的	还没谈	还没谈		不加班 在西二旗 通勤：25分钟	还没谈

弄完了，你得帮我搞个宣传视频啊

另外你的面试经历以及心得，也总结总结。看看签你一个兼职，让你分享相关的经验

最近的语音暂不能同步

2021年3月15日 15:00

恩 好

2021年3月15日 20:18

东哥 现在有时间不？？

东哥 猿辅导给offer了

可能得去猿辅导了

😭

其他的呢？

薪资2部分
基础薪资：35K*12
每月补助：1.7*12
年终奖:35* (1-3个月）比例 10% 70% 20%

期权：
100股 每股503美金
分4年发放，每年25%

薪资：51W 年终奖按2个月算
期权：8.1W
总包：59W

猿辅导就在我这边

猿辅导给你的offer

猿辅导 不是在望京么?


我擦，确实诱人

相当于我现在翻了 2.2倍


牛逼

😪 感觉 蚂蚁的瞬间就不香了

蚂蚁才给 28*16=44.5w


确实不香了，差的太多了

差了快15万 而且还不加班

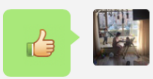
10点 - 7点 一周5天

但就是业务比较恶心 toB的，数据结构很恶心，写教研工具的

"大唐云智慧-李聪" 撤回了一条消息

文档：面试总结.note
链接：http:/[redacted]/noteshare?id=230a1929bf66f[redacted]=FE318140AB564856B5FA0107B8C71772

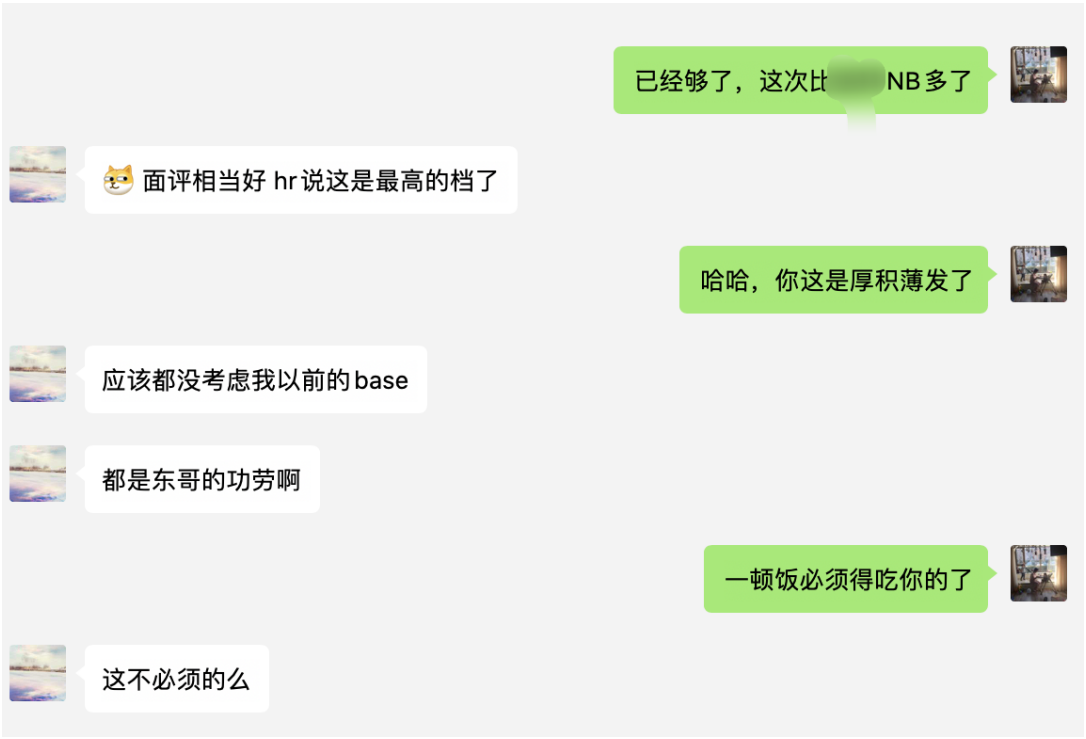
写的面试总结


👍

恩 这差不多是我所有能问道的知识点ab

对标p6+的水准

P7可能就不是这些问题了 😪


已经够了，这次比 NB多了
面评相当好 hr说这是最高的档了
哈哈，你这是厚积薄发了
应该都没考虑我以前的base
都是东哥的功劳啊
一顿饭必须得吃你的了
这不必须的么