

大家好，很高兴我们可以继续学习交流Java高频面试题。本小节是**Java进阶篇章中关于并发编程的最后一节**。在前面两个小节中，我们介绍了一些多线程并发编程的基础高频考察知识点，本小节我们主要来介绍**Java中的线程池相关知识点**，另外还包括**CountDownLatch**，**CyclicBarrier**，**ThreadLocal**以及**Atomic**等关键字的解析。

9

## (1) Java中的线程池有了解吗？

答：java.util.concurrent.ThreadPoolExecutor类就是一个线程池。客户端调用ThreadPoolExecutor.submit(Runnable task)提交任务，线程池内部维护的工作者线程的数量就是该线程池的线程池大小，有3种形态：

- **当前线程池大小**：表示线程池中实际工作者线程的数量
- **最大线程池大小 (maximumPoolSize)**：表示线程池中允许存在的工作者线程的数量上限
- **核心线程大小 (corePoolSize)**：表示一个不大于最大线程池大小的工作者线程数量上限

线程池的优势体现如下：

- 线程池可以重复利用已创建的线程，一次创建可以执行多次任务，有效降低线程创建和销毁所造成的资源消耗；
- 线程池技术使得请求可以快速得到响应，节约了创建线程的时间；
- 线程的创建需要占用系统内存，消耗系统资源，使用线程池可以更好的管理线程，做到统一分配、调优和监控线程，提高系统的稳定性。

解析：

创建线程是有开销的，为了重复利用已创建的线程降低线程创建和销毁的消耗，提高资源的利用效率，所以出现了线程池。线程池的参数字段如下所示：

```
public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue,
                           ThreadFactory threadFactory,
                           RejectedExecutionHandler handler) {
    // 牛客@我是祖国的花朵
```

我们来看下JDK对各个字段的解释：

- **corePoolSize：核心线程数**
- **maximumPoolSize：最大线程数**
- **keepAliveTime**：线程空闲但是保持不被回收的时间
- **unit**：时间单位
- **workQueue**：存储线程的队列
- **threadFactory**：创建线程的工厂
- **handler**：拒绝策略

## 线程池的排队策略:

当我们向线程池提交任务的时候, 需要遵循一定的排队策略, 具体策略如下:

- 如果运行的线程少于corePoolSize, 则Executor始终首选添加新的线程, 而不进行排队
- 如果运行的线程等于或者多于corePoolSize, 则Executor始终首选将请求加入队列, 而不是添加新线程
- 如果无法将请求加入队列, 即队列已经满了, 则创建新的线程, 除非创建此线程超出maximumPoolSize, 在这种情况下, 任务默认将被拒绝。

9

## 常见的线程池类型:

### newCachedThreadPool( )

- 核心线程池大小为0, 最大线程池大小不受限, 来一个创建一个线程
- 适合用来执行大量耗时较短且提交频率较高的任务

### newFixedThreadPool( )

- 固定大小的线程池
- 当线程池大小达到核心线程池大小, 就不会增加也不会减小工作者线程的固定大小的线程池

### newSingleThreadExecutor( )

- 便于实现单 (多) 生产者-消费者模式

## 常见的阻塞队列:

前面我们介绍了线程池内部有一个排队策略, 任务可能需要在队列中进行排队等候。常见的阻塞队列包括如下的三种, 接下来我们一起来看看吧。

### ArrayBlockingQueue:

- 内部使用一个**数组**作为其存储空间, 数组的存储空间是**预先分配**的
- **优点是** put 和 take操作不会增加GC的负担 (因为空间是预先分配的)
- **缺点是** put 和 take操作使用同一个锁, 可能导致锁争用, 导致较多的上下文切换。
- ArrayBlockingQueue适合在生产者线程和消费者线程之间的**并发程序较低**的情况下使用。

### LinkedBlockingQueue:

- 是一个无界队列 (其实队列长度是Integer.MAX\_VALUE)
- 内部存储空间是一个**链表**, 并且链表节点所需的**存储空间是动态分配**的
- **优点是** put 和 take 操作使用两个显式锁 (putLock和takeLock)
- **缺点是**增加了GC的负担, 因为空间是动态分配的。
- LinkedBlockingQueue适合在生产者线程和消费者线程之间的并发程序较高的情况下使用。

### SynchronousQueue:

SynchronousQueue可以被看做一种特殊的有界队列。生产者线程生产一个产品之后, 会等待消费者线程来取走这个产品, 才会接着生产下一个产品, 适合在生产者线程和消费者线程之间的处理能力相差不大的情况下使用。

我们前边介绍newCachedThreadPool时候说，这个线程池来一个线程就创建一个，这是因为其内部队列使用了SynchronousQueue，所以不存在排队。

### 关于线程池，你应该知道的事情：

- 使用JDK提供的快捷方式创建线程池，比如说newCachedThreadPool会出现一些内存溢出的问题，因为队列可以被塞入很多任务。所以，大多数情况下，我们都应该自定义线程池。
- 线程池提供了一些监控API，可以很方便的监控当前以及塞进队列的任务数以及当前线程池已经完成的任务数等。

### 考点分析：

线程池几乎是一个必考的知识点，所以我们必须熟练掌握线程池的基本参数及其含义，并且对排队策略有清晰的理解。常见线程池的类型，包括其使用到的阻塞队列等。

## (2) CountdownLatch和CyclicBarrier有了解吗？

答：两个关键字经常放在一起比较和考察，下边我们分别介绍。

CountDownLatch是一个倒计时协调器，它可以实现一个或者多个线程等待其余线程完成一组特定的操作之后，继续运行。

CountDownLatch的内部实现如下：

- CountDownLatch内部维护一个计数器，CountDownLatch.countDown () 每被执行一次都会使计数器值减少1。
- 当计数器不为0时，CountDownLatch.await () 方法的调用将会导致执行线程被暂停，这些线程就叫做该CountDownLatch上的等待线程。
- CountDownLatch.countDown () 相当于一个通知方法，当计数器值达到0时，唤醒所有等待线程。当然对应还有指定等待时间长度的CountDownLatch.await( long , TimeUnit)方法。

CyclicBarrier是一个栅栏，可以实现多个线程相互等待执行到指定的地点，这时候这些线程会再接着执行，在实际工作中可以用来模拟高并发请求测试。

可以认为是这样的，当我们爬山的时候，到了一个平坦处，前面队伍会稍作休息，等待后边队伍跟上来，当最后一个爬山伙伴也达到该休息地点时，所有人同时开始从该地点出发，继续爬山。

CyclicBarrier的内部实现如下：

- 使用CyclicBarrier实现等待的线程被称为参与方 (Party)，参与方只需要执行CyclicBarrier.await () 就可以实现等待，该栅栏维护了一个显示锁，可以识别出最后一个参与方，当最后一个参与方调用await () 方法时，前面等待的参与方都会被唤醒，并且该最后一个参与方也不会被暂停。
- CyclicBarrier内部维护了一个计数器变量count = 参与方的个数，调用await方法可以使得count -1。当判断到是最后一个参与方时，调用singalAll唤醒所有线程。

## (3) ThreadLocal有了解吗？

答：使用ThreadLocal维护变量时，其为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立的改变自己的副本，而不会影响到其他线程对应的副本。

**ThreadLocal内部实现机制：**

- 每个线程内部都会维护一个类似HashMap的对象，称为**ThreadLocalMap**，里边会包含若干了**Entry（K-V键值对）**，相应的线程被称为这些Entry的属主线程
- **Entry的Key是一个ThreadLocal实例，Value是一个线程特有对象**。Entry的作用是为属主线程建立起一个ThreadLocal实例与一个线程特有对象之间的对应关系
- Entry对Key的引用是弱引用；Entry对Value的引用是强引用。

9

**(4) Atomic有了解吗？**

**答：**介绍Atomic之前先来看一个问题吧，**i++操作是线程安全的吗？**

i++操作并不是线程安全的，它是一个复合操作，包含三个步骤：

- 拷贝i的值到临时变量
- 临时变量++操作
- 拷贝回原始变量i

这是一个**复合操作，不能保证原子性**，所以这不是线程安全的操作。**那么如何实现原子自增等操作呢？**

这里就用到了JDK在java.util.concurrent.atomic包下的AtomicInteger等原子类了。**AtomicInteger类提供了getAndIncrement和incrementAndGet等原子性的自增自减等操作**。Atomic等原子类内部使用了CAS来保证原子性。

接下来，我们来看代码吧，首先是使用变量i的情况：

```

1  class ThreadTest implements Runnable {
2
3      static int i = 0;
4      public void run() {
5          for (int m = 0; m < 1000000; m++) {
6              i++;
7          }
8      }
9  };
10 public class Test {
11     public static void main(String[] args) throws InterruptedException {
12         ThreadTest mt = new ThreadTest();
13
14         Thread t1 = new Thread(mt);
15         Thread t2 = new Thread(mt);
16         t1.start();
17         t2.start();
18         // 休眠一下，让线程执行完毕。
19         Thread.sleep(500);
20         System.out.println(ThreadTest.i);
21     }
22 }

```

该程序的输出是不确定的，比如输出1933446，也就是线程不安全，发生了竞态导致计算结果有误。

当我们使用了**Atomic等原子类**时，会发现每次输出结果都是2000000，符合我们的程序设计要求。

```

1  import java.util.concurrent.atomic.AtomicInteger;
2
3  class ThreadTest implements Runnable {
4
5      static AtomicInteger i = new AtomicInteger(0);
6

```

```
7     public void run() {
8         for (int m = 0; m < 1000000; m++) {
9             i.getAndIncrement();
10        }
11    }
12};
13
14public class Test {
15    public static void main(String[] args) throws InterruptedException {
16        ThreadTest mt = new ThreadTest();
17
18        Thread t1 = new Thread(mt);
19        Thread t2 = new Thread(mt);
20        t1.start();
21        t2.start();
22        // 休眠一下，让线程执行完毕。
23        Thread.sleep(500);
24        System.out.println(ThreadTest.i.get());
25    }
26}
```

9

## (5) 其余常见多线程知识点:

限于文章的篇幅，我这里将其余一些重要的知识点给大家罗列出来。虽然我可以简单给出大家答案，但是我坚定的相信，这些知识点留给大家去主动探索学习会有更好的学习效果。当然，你可以把这些知识点的学习当做是本小节的课后习题。

- 什么是happened-before原则?
- JVM虚拟机对内部锁有哪些优化?
- 如何进行无锁化编程?
- CAS以及如何解决ABA问题?
- AQS (AbstractQueuedSynchronizer) 的原理与实现。

## 总结:

在高效并发编程的三个小节中，我们较为详细的介绍了多线程并发编程的相关知识点。通过使用**面试题+解析**的方式，尽量做到了从入门到深入理解其技术原理实现。当然，我们在最后将其余的一些重要知识点罗列出来，作为大家的课后习题，希望大家可以自主学习这几个知识点，并且将核心答案记录下来，可以评论区互动交流学习心得哦。下一节开始，我们将学习Java进阶篇章中关于**JVM内存机制**相关的知识点。

限于作者水平，文章中难免会有不妥之处。大家在学习过程中遇到我没有表达清楚或者表述有误的地方，欢迎随时在文章下边指出，我会及时关注，随时改正。另外，大家有任何话题都可以在下边留言，我们一起交流探讨。

讨论

评论



刘畅201904211606816

1#

打卡，写的很好，基本重要的点都能覆盖

发表于 2020-01-01 10:10:07

赞(0) 回复(1)

我是祖国的花朵 (作者) : 加油~

2020-01-01 10:54:32

赞(0) 回复(0)

[回复](#)

9



柳杰201905011049420

2#

但是感觉很多都没有看懂

发表于 2020-01-15 18:38:36

[赞\(0\)](#) [回复\(1\)](#)

**我是祖国的花朵** **[作者]**： 你好，看不太懂，说明你对这块知识点掌握较少，基础不牢固。因为面试题毕竟是在对基础知识点的掌握上，建议针对薄弱环节加强学习，可以买对应的经典书籍研究学习。

2020-01-15 20:44:05

[赞\(0\)](#) [回复\(0\)](#)[回复](#)

牧水s

3#

打卡

发表于 2020-01-15 21:18:02

[赞\(0\)](#) [回复\(0\)](#)

牛客248955670号

4#

打开

发表于 2020-02-25 16:27:13

[赞\(0\)](#) [回复\(0\)](#)

禾田誉子

5#

对于SynchronousQueue这个队列的特性，意味着它每次只能接收一个任务吗？

发表于 2020-02-26 17:31:39

[赞\(0\)](#) [回复\(1\)](#)

**我是祖国的花朵** **[作者]**： 这种队列在没有工作者线程从队列中取数据的时候，其实是不可以存储数据的，这也就是为什么newCachedThreadPool线程池来一个任务就创建一个新线程的原因。

2020-02-26 20:32:07

[赞\(0\)](#) [回复\(0\)](#)[回复](#)


求一个Offer

6#

有一个点没有理解，就是线程池中的阻塞队列，如果小于核心线程的话，就直接添加新的线程，而大于核心线程小于最大线程的话，就进入阻塞队列，并且在keepAliveTime时间结束时，执行拒绝策略，而大于最大核心线程数，则直接执行拒绝策略，不知道理解对不对。还有就是newFixedThreadPool 核心线程数等于最大线程数，那岂不是没有阻塞队列，直接执行拒绝策略？

发表于 2020-03-01 11:30:24

赞(0) 回复(1)

我是祖国的花朵  (作者) : 当前线程数大于核心线程数, 那么任务会被塞入队列中;  
keepAliveTime是指空闲线程得存活时间, 也就是空闲之后, 线程会回收; 这块你的理解很混乱, 再  
仔细看下哈

9

2020-03-05 17:53:39

赞(0) 回复(0)

回复




牛客ID: 9047559

7#

楼主分一下aqs吧, 好像是高频面试

发表于 2020-03-05 17:12:59

赞(0) 回复(1)

我是祖国的花朵  (作者) : 简单说下AQS的基本原理哈。AQS又称为队列同步器, 用来构建锁或其他同步组件的基础框架。内部通过一个int成员变量state来控制同步状态, 当state = 0时, 说明没有任何线程占有共享资源的锁; state = 1时, 则说明有线程目前正在使用共享变量, 其他线程必须加入同步队列进行等待, 当然state也可以继续执行+1操作, 比如可重入锁。AQS同步器的实现依赖于内部的同步队列(FIFO的双向链表队列)完成对同步状态(state)的管理, 当前线程获取锁(同步状态)失败时, AQS会将该线程以及相关等待信息包装成一个节点(Node)并将其加入同步队列, 同时会阻塞当前线程, 当同步状态释放时, 会将头结点head中的线程唤醒, 让其尝试获取同步状态。简单来说, 就是同步状态state和同步队列。ReentrantLock锁就是使用了AQS来控制同步状态。

2020-03-05 17:58:24

赞(1) 回复(0)

回复



老宋啊啊啊

8#

打卡

发表于 2020-03-08 11:17:38

赞(0) 回复(0)



Shawn\_Liu

9#

06/01 打卡

发表于 2020-06-02 12:24:27

赞(0) 回复(0)