41、分布式任务调度？

定时任务随着技术发展，从单线程调度到多线程调度，从单机部署到集群部署，从独立执行到多任务协同执行。

第一阶段
单线程调度，在Java1.5之前，基于线程的等待(sleep或wait)机制定时执行，需要开发者实现调度逻辑，单个线程(Thread)处理单个任务有些浪费，
　　但是一个线程(Timer)处理多个任务容易因为某个任务繁忙导致其他任务阻塞。

第二阶段
线程池调度，在Java1.5开始提供ScheduledExecutorService调度线程池，调度线程池支持固定的延时和固定间隔模式，
　　对于需要在某天或者某月的时间点执行就不大方便，需要计算时间间隔，转换成启动延时和固定间隔，处理起来比较麻烦。

第三阶段
Spring任务调度，Spring简化了任务调度，通过@Scheduled注解支持将某个Bean的方法定时执行，除了支持固定延时和固定间隔模式外，
　　还支持cron表达式，使得定时任务的开发变得极其简单。 加注解@Scheduled(cron = "0/20 * * * * *")

第四阶段
Quartz任务调度，在任务服务集群部署下，Quartz通过数据库锁，实现任务的调度并发控制，避免同一个任务同时执行的情况。
　　Quartz通过Scheduler提供了任务调度API，开发可以基于此开发自己的任务调度管理平台。

第五阶段
分布式任务平台，提供一个统一的平台，无需再去做和调度相关的开发，业务系统只需要实现具体的任务逻辑，自动注册到任务调度平台，
　　在上面进行相关的配置就完成了定时任务的开发。

42、java和C++有什么相同点和不同点吗
　Java语言不需要程序对内存进行分配和回收。Java丢弃了C++ 中很少使用的、很难理解的、令人迷惑的那些特性，如操作符重载、多继承、自动的强制类型转换。
　特别地，Java语言不使用指针，内存的分配和回收都是自动进行的，程序员无须考虑内存碎片的问题。

43、JDK11和JDK15的新特性？
　JDK11：
　1Local Var
　　在Lambda表达式中，可以使用var关键字来标识变量，变量类型由编译器自行推断。
　2HttpClient
　　长期以来，如果要访问Http资源，JDK的标准库中只有一个HttpURLConnection，这个古老的API使用非常麻烦，而且已经不适用于最新的HTTP协议。
　　JDK11的新的HttpClient支持HTTP/2和WebSocket，并且可以使用异步接口：
　3List API

对于List接口，新增了一个of(T...)接口，用于快速创建List对象：List list = List.of("Java", "Python", "Ruby");

```
JDK15：
1封闭类sealed
      可以是封闭类和或者封闭接口，用来增强 Java 编程语言，防止其他类或接口扩展或实现它们。
      有了这个特性，意味着以后不是你想继承就继承，想实现就实现了，你得经过允许permits才行。
      例子：public abstract sealed class Student
             permits ZhangSan, LiSi, ZhaoLiu {
             ...
         }

2准备禁用和废除偏向锁
     在 JDK 15 中，默认情况下禁用偏向锁（Biased Locking），并弃用所有相关的命令行选项。
     后面再确定是否需要继续支持偏向锁，国为维护这种锁同步优化的成本太高了。

3ZGC垃圾回收器 功能转正
     ZGC是一个可伸缩、低延迟的垃圾回收器。默认仍然是 G1。
```

44.CMS会产生什么问题？

内存碎片（原因是采用了标记-清除算法）

对 CPU 资源敏感（原因是并发时和用户线程一起抢占 CPU）

浮动垃圾：在并发标记阶段产生了新垃圾不会被及时回收，而是只能等到下一次GC

45.CMS中产生的碎片有什么方法解决吗?

CMS是一款基于“标记-清除”算法实现的收集器，如果读者对前面这部分介绍还有印象的话，就可能想到这意味着收集结束时会有大量空间碎片产生。

空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很多剩余空间，但就是无法找到足够大的连续空间来分配当前对象，

而不得不提前触发一次Full GC的情况。

```
虚拟机设计者们还提供了另外一个参数-XX：CMSFullGCsBefore-Compaction（此参数从JDK 9开始废弃），
     这个参数的作用是要求CMS收集器在执行过若干次（数量由参数值决定）不整理空间的Full GC之后，
     下一次进入Full GC前会先进行碎片整理（默认值为0，表示每次进入Full GC时都进行碎片整理）。
```

46、Redis集群的全量同步和增量同步是通过什么实现的？

全量同步： master执行bgsave生成rdb数据快照发送给Slave

增量同步：

从服务器 每秒钟 向从服务器发送REPLCONF ACK <replication_offset>命令来做心跳检测，以及告诉主节点自己的复制偏移量。

主服务器若发现从服务器的复制偏移量小于自己的，根据从服务器发过来的offset，在复制积压缓冲区中找到offset之后的数据，并将其发给从节点执行就可以了。

47、CurrentHashMap的size函数？

在实际的项目过程中，我们通常需要获取集合类的长度， 那么计算 ConcurrentHashMap 的元素大小就是一个有趣的问题，因为他是并发操作的，

就是在你计算 size 的时候，它还在并发的插入数据，可能会导致你计算出来的 size 和你实际的 size 有差距。

```
众所周知，concurrenthashmap有很多个segments，首先遍历segments将每个segment的count加起来作为整个concurrenthashMap的size。
     如果没有并发的情况下这自然就可以了，但这是多线程的，如果前脚统计完后脚有变化了，这就不准确了，源码中引入了modCount和两次比较来实现size的确认。
```

1.进行第一遍遍历segments数组，将每个segemnt的count加起来作为总数，期间把每个segment的modCount加起来sum作为结果是否被修改的判断依据。

这里需要提一下modCount，这个是当segment有任何操作都会进行一次增量操作，代表的是对Segment中元素的数量造成影响的操作的次数，

这个值只增不减！！只增不减很重要，这样就不会出现一个segment+1，导致modcount+1,而另一个segment-1，即modcount-1 ,从而在统计所有的时候modcount没有变化。

2.size操作就是遍历了两次所有的Segments，每次记录Segment的modCount值，然后将两次的modCount进行比较，如果相同，则表示期间没有发生过写入操作，

就将原先遍历的结果返回，如果不相同，则把这个过程再重复做一次，如果再不相同，则就需要将所有的Segment都锁住，然后一个一个遍历了。

3.而之所以之所以要先不加锁进行判断，道理很明显，就是不希望因为size操作获取这么多锁，因为获取锁不光占用资源，

也会影响其他线程对ConcurrentHash的使用，影响并发情况下程序执行的效率。使用锁要谨慎！

总结：在 JDK1.7 中，

第一种方案他会使用不加锁的模式去尝试多次计算 ConcurrentHashMap 的 size，最多三次，

比较前后两次计算的结果，结果一致就认为当前没有元素加入，计算的结果是准确的。

第二种方案是如果第一种方案不符合，他就会给每个 Segment 加上锁，然后计算 ConcurrentHashMap 的 size 返回。

JDK1.8 size

是通过对 baseCount 和 counterCell 进行 CAS 计算，最终通过 baseCount 和 遍历 CounterCell 数组得出 size。

48.newCachedThreadPool的实现原理？

newCachedThreadPool创建一个可缓存线程池，用于处理大量短时间工作任务的线程池。

1）核心线程数为0；

2）最大线程数为Interger.MAX_VALUE，即0x7fffffff（2147483647）

3）线程空闲时长为60秒，如果空闲超过60秒，则线程将被终止，并移出缓存。

该线程池，使用J.U.C的SynchronousQueue阻塞队列，该队列具有以下几个特性：

1）SynchronousQueue没有容量。与其他BlockingQueue不同，SynchronousQueue是一个不存储元素的BlockingQueue。

每一个put操作必须要等待一个take操作，否则不能继续添加元素，反之亦然。

2）因为没有容量，所以对应 peek, contains, clear, isEmpty ... 等方法其实是无效的。

例如clear是不执行任何操作的，contains始终返回false,peek始终返回null。

49.红黑树和平衡二叉树的区别？跳表的查找时间复杂度？

红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。

平衡二叉树追求绝对平衡，条件比较苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知。

最高级索引 h 满足 2 = n/2^h，即 h = log2n - 1，最高级索引 h 为索引层的高度加上原始数据一层，跳表的总高度 h = log2n。所以查找的时间复杂度是常数*log2n

50.新生代转换为老年代有几种情况？

1.Eden区满时，进行Minor GC

当Eden和一个Survivor区中依然存活的对象无法放入到Survivor中，则通过分配担保机制提前转移到老年代中。

2.对象体积太大，新生代无法容纳

-XX:PretenureSizeThreshold即对象的大小大于此值，就会绕过新生代，直接在老年代分配，此参数只对Serial及ParNew两款收集器有效。

3.长期存活的对象将进入老年代

虚拟机对每个对象定义了一个对象年龄（Age）计数器。当年龄增加到一定的临界值时，就会晋升到老年代中，该临界值由参数：-XX:MaxTenuringThreshold来设置。

如果对象在Eden出生并在第一次发生MinorGC时仍然存活，并且能够被Survivor中所容纳的话，则该对象会被移动到Survivor中，并且设Age=1；

以后每经历一次Minor GC，该对象还存活的话Age=Age+1。

4.动态对象年龄判定

虚拟机并不总是要求对象的年龄必须达到MaxTenuringThreshold才能晋升到老年代，如果在Survivor区中相同年龄（设年龄为age）

的对象的所有大小之和超过Survivor空间的一半，年龄大于或等于该年龄（age）的对象就可以直接进入老年代，

无需等到MaxTenuringThreshold中要求的年龄。