1. Java并发容器
2. ConcurrentHashMap
3. 为什么使用ConcurrentHashMap

HashMap在单线程的情况使用是没有问题的，但是在多线程的环境下使用HashMap会导致HashMap的Entry链表形成环形数据结构，导致死循环；而使用线程安全的HashTable的效率又非常地低下，它使用synchronized来保证线程安全，多线程下效率很低；因此需要使用ConcurrentHashMap，针对HashTable的一个锁的特性，ConcurrentHashMap中使用多把锁，每一把锁用于锁住容器中的一部分数据，那么多线程访问容器中不同数据段的数据时，线程之间就不会存在锁的竞争问题了，这就是锁分段技术，首先将数据分成一段段地存储，然后给每一段数据分配一把锁，一个锁只锁定一段数据。

1. ConcurrentHashMap的结构

ConcurrentHashMap是由Segment数组和HashEntry数组组成的，Segment是ReentrantLock，扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap中包含一个Segment数组，一个Segment数组的结构和HashMap类似，是一种数组和链表结构，一个Segment数组中包含一个HashEntry数组，每个HashEntry就是一个链表结构的元素，要想修改某个HashEntry数组的数据，就需要获得它所对应的Segment锁。

1. ConcurrentHashMap的初始化

ConcurrentHashMap初始化方法是通过initialCapacity，loadFactor和concurrencyLevel几个参数来初始化segment数组，段偏移量segmentShift，段掩码segmentMask和每个segment中的HashEntry数组实现的。

1. 初始化segment数组

segment数组的长度ssize是通过concurrencyLevel计算得出的，必须保证segment数组的长度是2的N次方，因此必须计算出一个大于或等于concurrencyLevel的最小的2的N次方值来作为segment数组的长度。

1. 初始化每个segment

输入参数initialCapacity是ConcurrentHashMap的初始容量，loadfactor是每个segment的负载因子，在构造方法里需要通过这两个参数来初始化数组中的每个segment。起始时，用initialCapacity除以之前计算出的segment数组的长度，如果结果c>1，则表示至少有一个segment中的HashEntry数组的长度不止1，因此需要设置每个segment的HashEntry数组的初始化长度为大于等于c的2的N次方值；如果c==1，则设置每个segment的HashEntry数组的初始化长度cap为1。

segment的HashEntry数组的容量threshold = (int)HashEntry数组的初始化长度 \* loadfactor，默认情况下initialCapacity等于16，loadfactor等于0.75，通过计算得出cap为1，threshold为0。

1. 定位segment

既然concurrentHashMap使用分段锁segment来保护不同段的数据，那么在插入和获取数据时需要通过Hash算法定位到segment，首先使用Hash算法对数据取hashCode，然后将该HashCode再散列，目的是减少散列冲突，使元素能够均匀地分布到不同的segment上。

1. concurrentHashMap的操作
2. get操作

segment的get操作非常简单高效，首先经过一次再散列，然后使用这个散列值通过散列运算定位到segment，再通过散列散发定位到元素。get操作的高效之处在于整个get过程不需要加锁，除非读到的值是空才会加锁重读。它将get方法中要用到的共享变量设置为volatile类型，如用于统计当前segment大小的count字段和用于存储值的HashEntry的value，这些变量在各个线程之间都是可见的，能够被多个线程同时读，并且保证不会读到过期的值，因为Java内存模型的happen before原则，对于volatile字段的写入操作一定先于读操作，即时两个线程同时修改和读取volatile变量，get操作也能够拿到最新的值。

1. put操作

由于put方法需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须加锁。put方法首先定位到segment，然后在segment中进行插入操作。插入操作有两个步骤，第一步判断是否需要对segment中的HashEntry数组扩容，第二步定位添加元素的位置，然后将其放入HashEntry数组中。

判断扩容操作是直接判断当前的segment的HashEntry数组是否超过了容量threshold，如果超过了容量则进行扩容。在扩容的时候，首先创建一个两倍容量的数组，然后将原数组的元素进行再散列后插入到新的数组中。

1. size操作

如果要统计整个concurrentHashMap中元素的大小，就需要统计所有segment里元素的大小后求和，每个segment里的全局变量count是一个volatile变量，那么是否直接将每个segment的count变量相加就可以得到整个concurrentHashMap的大小呢？基本的思路是这样的，但是在多线程的情况下，我们可能在获取每个segment的count时是正确的，但是在相加时可能就会有线程修改了某些count值，所以最安全的做法是在相加时将每个segment的put，remove，clean方法都锁住，这样的效率比较低，因此选择的方案是先尝试2次不锁住segment的方式来统计各个segment的大小，如果在这个过程中有segment的count发生了变化，则再采用加锁的方式。

那么，如何判断segment的count发生了变化呢？其实每个segment的put，remove，clean方法在操作元素时都会将segment中的一个变量modCount进行+1操作，因此我们在统计时判断每个segment的modCount变量是否变化即可。

1. ConcurrentLinkedQueue
2. concurrentLinkedQueue介绍

实现一个线程安全的队列有两种方式，一种是使用阻塞算法，用一个锁或两个锁锁住入队和出队的方式实现；另一种是非阻塞算法， 使用循环CAS实现，线程安全队列concurrentLinkedQueue就是非阻塞实现的。

ConcurrentLinkedQueue是一个基于链接节点的无界线程安全队列，它采用FIFO排序，当我们添加一个元素时它会添加到队列的尾部，当获取一个元素时，它会返回队列头部的元素。

1. concurrentLinkedQueue的结构

ConcurrentLinkedQueue由head节点和tail节点组成，每个节点由节点元素和指向下一个节点的引用组成，从而组成链表结构的队列。

1. 入队列
2. 入队列过程

入队主要做两件事：第一是将入队节点设置为当前队列尾结点的下一个节点；第二是更新tail节点，如果tail节点的next节点不为空，则将入队节点设置为tail节点，如果tail节点的next节点为空，则将入队节点设置为tail的next节点，所以tail节点不总是尾结点。

上述在单线程情况下可以理解，但是在多线程的情况下更复杂，如果一个线程正在入队，那么它必须先获取尾结点，然后设置尾结点的下一个节点为入队节点，但此时可能有另一个线程插队了，那么队列的尾结点发生了变化，这是当前线程要暂停入队操作，然后重新获取尾结点，直到通过CAS将入队节点设置为尾结点的next节点。

1. 定位尾结点

由于tail并不总是尾结点，所以每次入队时都必须先通过tail节点找到尾结点，尾结点可能是tail节点，也可能是tail节点的next节点。

1. 使用CAS设置入队节点为尾结点

p.casNext(null, n)方法用于将入队节点设置为当前队列尾结点的next节点，如果p.next == null ，表示p是当前队列的尾结点，如果不为null，表示已经有其他的线程修改了尾结点，则需要重新获取当前队列的尾结点。

1. HOPS的设计意图

我们可能会奇怪，为什么tail节点并不一定指向尾结点，如果我们每次都让tail节点指向尾结点，不就可以直接使用tail节点的next指针来判断尾结点是否改变了吗？但是，注意一个问题，这样做的话每次都需要使用循环CAS来更新tail节点，那么更新tail节点的次数就会更多了，而如果通过HOPS变量（默认为1）来控制tail节点和尾结点之间的距离，距离越大则更新tail节点的次数越少，同时也需要增加定位尾结点的时间，但是效率依然提升了。因为更新tail节点的操作本质是对volatile变量的写操作，而定位尾结点的操作时对volatile变量的读操作，读操作的开销远远低于写操作。

1. 出队列

出队列就是从队列中返回一个节点元素，并清空该节点对元素的引用。并不是每次出队都会更新head节点，当head节点中有元素时，直接弹出head节点中的元素，而不会更新head节点；只有当head节点中没有元素时，出队操作哦才会更新head节点。这种做法也是通过HOPS变量来减少使用CAS更新head节点的消耗，提高出队效率。

首先获取头结点中的元素，判断该元素是否为空，如果为空则表示另一个线程已经进行了一次出队操作将该节点的元素取走；如果不为空，则使用CAS方式将头结点的引用设置为空，如果CAS成功，则直接返回头结点的元素，如果不成功，则表示另一个线程已经进行了一次出队操作更新了head节点，需要重新获取头结点。

1. 阻塞队列
2. 阻塞队列介绍

阻塞队列是一个支持两个附加操作的队列，这两个附加操作支持阻塞地插入和移除方法。所谓的阻塞地插入就是指当队列满时，队列会阻塞插入元素的线程；阻塞地移除方法是指在队列为空时，获取元素的线程会等待队列变为非空。

阻塞队列常用于生产者和消费者的场景，在阻塞队列不可用时，两个附加操作有4种处理方式：

抛出异常：add(e), remove();

返回true/false：offer(e)，poll();

一直阻塞：put(e)，take();

超时退出：offer(e,time,unit)，poll(time,unit)；

1. 7种阻塞队列
2. ArrayBlokingQueue

用数组实现的有界阻塞队列，默认使用非公平实现，按照FIFO规则排序。

1. LinkedBlockingQueue

用链表实现的有界阻塞队列，默认和最大长度为Integer.MAX\_VALUE，按照FIFO规则排序。

1. PriorityBlockingQueue

支持优先级的无界阻塞队列，默认采用自然升序。

1. DelayQueue

支持延时获取元素的无界阻塞队列，队列使用PriorityQueue实现，队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能够从队列中获取当前的元素，常用于定时系统等。

1. SynchronousQueue

不存储元素的阻塞队列，每一个put操作必须等待一个take操作取走元素，适合传递性场景，并且它的吞吐量大于第一个和第二个阻塞队列。

1. LinkedTransferQueue

由链表结构组成的无界阻塞TransferQueue队列。

1. LinkedBlockingDeque

由链表结构组成的双向阻塞队列。

1. 阻塞队列的实现原理

使用通知模式实现，即当生产者往满的队列里面添加元素时会阻塞住生产者，当消费者消费了一个元素后，会通知生产者当前队列可用，是通过Condition来实现的。阻塞生产者是通过LockSupport.park()来实现的，而它又是通过调用unsafe类中的native方法park()实现的。

1. Fork/Join框架
2. 框架概述

Fork/Join框架是JDK7提供的一个用于并行执行任务的框架，将一个大任务拆分为多个小任务，最终汇总每个小任务的结果得到大任务结果。

1. 工作窃取算法

工作窃取算法是指某个线程从其他队列中窃取任务来执行，当我们将大任务拆分为子任务后，为了减少线程之间的竞争，将子任务分别放到不同的队列中，为每个队列单独创建一个线程来执行任务。如果有的线程执行速度快，则它会从其他的队列中窃取任务来执行，此时两个线程会访问同一个队列，因此为了减少竞争，通常采用双端队列，被窃取任务的线程从头部获取任务，而窃取任务的线程从尾部获取任务。但是这样其实还是有竞争，比如队列中只有一个任务时。

1. 框架的设计
2. 分割任务

一个fork类来将大任务分割为子任务。任务ForkJoinTask的子类有两个，一个是没有返回结果的子类RecursiveAction，另一个是由返回结果的子类RecursiveTask。

1. 执行任务并合并结果

执行子任务，子任务执行完的结果都统一放在一个队列里面，启动一个线程从队列中取数据，合并数据。任务需要通过ForkJoinPool执行，任务添加到当前工作线程所维护的双端队列中，进入队列的头部，如果队列中没有任务，则从其他队列的尾部获取任务。

1. Java的原子操作类
2. 概述

JDK1.5开始提供了juc.atomic包，这个包的原子操作类提供了线程安全地更新一个变量的方式，直接从内存中更新变量的值。atomic包中共有13个类，4种类型的原子更新方式，分别是原子更新基本类型，原子更新数组，原子更新引用和原子更新属性字段，它们都是使用Unsafe实现的。

1. 原子更新基本类型

原本基本类型有六个，使用的都是Unsafe类的CAS 方法，分别是compareAndSwapObject()，compareAndSwapInt()，compareAndSwapLong，对于char,float,double都可以使用unsafe列的CAS方法进行更新。

1. 原子更新数组

将当前的数组赋值一份，然后修改复制的数组即可。

1. 原子更新引用类型

如果需要一次性更新多个变量，就需要更新引用对象，如更新一个自定义对象，首先创建一个原子引用更新类，然后将要修改的对象设置进去，调用compareAndSet方法进行原子更新操作，替换为另一个对象。

1. 原子更新字段

如果只需要原子更新某个类中的某个字段属性，则需要使用原子更新字段类，其中有一个AtomicStampedReference类，可以原子更新带有版本号的引用类型，解决使用CAS出现的ABA问题。