IO

Concurrent类

hashMap：多线程下使用put会引起死循环 使得CPU打满(<https://blog.csdn.net/qq_36520235/article/details/82417949>)

数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的。

如果定位到的数组位置不含链表（当前entry的next指向null）：那么对于查找，添加等操作很快，仅需一次寻址即可；

如果定位到的数组包含链表：对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好

1.7：使用头插法：单链表进行纵向延伸 使用头插法容易出现逆序且环形列表死循环问题

扩容后位置计算 （hash值 & length-1） 重新计算

1.8：使用尾插法：加入红黑树使用尾插法 避免死循环问题

扩容后计算： 原位置or 原位置+扩容值

Put:

1. 判断table是否存在
2. 初始化table
3. 判断key是否为null
4. 计算hashcode
5. 是否冲突
6. 新增一个entry放入值

哈希冲突其他解决方案：

1. 开放定址法：即发生冲突时，去寻找下一个空的哈希地址。只要哈希表足够大，总能找到空的哈希地址。
2. 再哈希法：即发生冲突时，由其他的函数再计算一次哈希值
3. 建立公共溢出区：将哈希表分为基本表和溢出表，发生冲突时，将冲突的元素放入溢出表

hashTable：简单粗暴的使用synchronized 相当于给整个哈希表加锁 粒度太大 性能低

ConcurrentHashMap 避免了全局锁优化成了局部锁

1.7：

采用segment+hashEntry。Segment继承了ReentrantLock,自带了锁的功能

分段锁技术 将数据分成一段一段的存储，然后给每一段数据配一把锁

1.8：利用了volatile、final、CAS 等lock-free 技术减少锁竞争

数组+红黑树+CAS+ Synchronize

放弃了segment 采用了node node中保存了key value hashcode value和next都使用了volatile 修饰

1、如果没有初始化就先调用initTable（）方法来进行初始化过程

2、如果没有hash冲突就直接CAS插入

3、如果还在进行扩容操作就先进行扩容

4、如果存在hash冲突，就加锁来保证线程安全，这里有两种情况，一种是链表形式就直

5、遍历到尾端插入，一种是红黑树就按照红黑树结构插入，

Synchronize：

Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

1. **monitorenter**：每个对象都是一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：
   1. 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者；
   2. 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1；
   3. 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权；
2. monitorexit：执行monitorexit的线程必须是objectref所对应的monitor的所有者。指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

monitorexit指令出现了两次，第1次为同步正常退出释放锁；第2次为发生异步退出释放锁；

AQS是一个用于构建锁和同步容器的框架。事实上concurrent包内许多类都是基于AQS构建，例如ReentrantLock，Semaphore，CountDownLatch，ReentrantReadWriteLock，FutureTask等。AQS解决了在实现同步容器时设计的大量细节问题

await内部实现流程:

判断state计数是否为0，不是，则直接放过执行后面的代码

大于0，则表示需要阻塞等待计数为0

当前线程封装Node对象，进入阻塞队列

然后就是循环尝试获取锁，直到成功（即state为0）后出队，继续执行线程后续代码

countDown内部实现流程:

尝试释放锁tryReleaseShared，实现计数-1

若计数已经小于0，则直接返回false

否则执行计数(AQS的state)减一

若减完之后，state==0，表示没有线程占用锁，即释放成功，然后就需要唤醒被阻塞的线程了

释放并唤醒阻塞线程 doReleaseShared

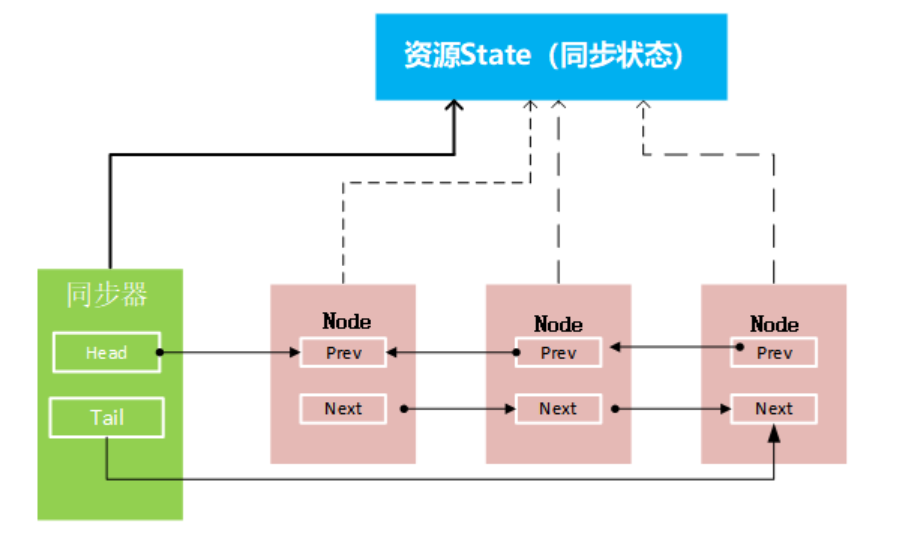
如果队列为空，即表示没有线程被阻塞（也就是说没有线程调用了 CountDownLatch#wait()方法），直接退出

头结点如果为SIGNAL, 则依次唤醒头结点下个节点上关联的线程，并出队

AQS：抽象的队列式的同步器—双端双向队列

CLH同步队列是一个FIFO双向队列，AQS依赖它来完成同步状态的管理，当前线程如果获取同步状态失败时，AQS则会将当前线程已经等待状态等信息构造成一个节点（Node）并将其加入到CLH同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点唤醒（公平锁），使其再次尝试获取同步状态。

在CLH同步队列中，一个节点表示一个线程，它保存着线程的引用（thread）、状态（waitStatus）、前驱节点（prev）、后继节点（next），其数据结构如下



负载：(需要仔细补充)

一段时间内平均线程数 通常时间：1 5 15

单个CPU 0.7系数

1分钟系统负荷表示最近的暂时现象。15分钟系统负荷表示是持续现象

··