高并发编程：

CHM

1. cas自旋 + volitate关键字（初始化,第一个节点,）,乐观锁，可见性 （volitate关键字可引申到 jmm模型,乱序问题,内存屏障），synchronize 分段锁
2. countCell,分流统计辅助器
3. 扰动函数，高低位混淆，避免在小容量时冲突严重

```
hashVal = (h ^ (h >>> 16)) & HASH_BITS
```

4. hash算法调优，&替换%， **当n = 2 ^ x时,有公式hash % n = hash & (n-1)**

```
index = (n - 1) & hash)
```

5. sizeCtl 默认0,-1初始化或者扩容,正常时期为 factor * capacity
6. 1.8 红黑树,单链长度大于8,capacity大于64时会进行红黑树转化.未达到容量时采取扩容缓解冲突.

```
            if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
                tryPresize(n << 1);
```

7. 扩容时，多线程参与部分扩容任务,高低链快速迁移，oldcapacity & p.hash = 0 低位链, sizeCtl记录参与扩容线程,前16位记录扩容stamp标示,后15位用与记录参与线程数,默认每个线程处理16个槽位, 否则 oldcapacity / 8/ncpu = 槽位数

```
// 每个线程分多少槽位
(stride = (NCPU > 1) ? (n >>> 3) / NCPU : n)

    int runBit = fh & n;
    Node<K,V> lastRun = f;
    for (Node<K,V> p = f.next; p != null; p = p.next) {
        int b = p.hash & n;
        if (b != runBit) {
            runBit = b;
            lastRun = p;
        }
    }
    if (runBit == 0) {
        ln = lastRun;
        hn = null;
    }
    else {
        hn = lastRun;
        ln = null;
    }
    for (Node<K,V> p = f; p != lastRun; p = p.next) {
        int ph = p.hash; K pk = p.key; V pv = p.val;
        if ((ph & n) == 0)
        ln = new Node<K,V>(ph, pk, pv, ln);
        else
        hn = new Node<K,V>(ph, pk, pv, hn);
    }
    setTabAt(nextTab, i, ln);
    setTabAt(nextTab, i + n, hn);
    setTabAt(tab, i, fwd);
```

8. forward节点用于表示该槽位已被迁移，迁移时仍可进行get请求,理论上如果put请求未命中扩容节点仍可插入成功,不过在增加count时会加入线程扩容，若达到最大线程扩容数或槽位已经被分配完直接结束.

并发安全背景

1. cpu乱序执行,缓存不一致问题.

- cpu中存在高速缓冲（**mesi**）,storebuffer，更新数据时先更新本cpu缓冲数据，并使得其他线程缓存失效，将更新数据提交到storebuffer中，只有其他cpu都返回ack确认后再将storebuffer中数据刷新到主内容中，存在异步生效，导致乱序执行.
- 解决办法是操作系统层面提供内存屏障

2. 内存屏障 -> storebarrier,loadbarrier,storeloadbarrier

- loadbarrier（读屏障） 意味后续的读操作一定发生在storebarrier前面的所有操作.

- storebarrier（写屏障） 意味着前面的写操作一定对后续可见.

3. 由于内存屏障存在于操作系统层面,不应该由开发人员关注处理,所以jvm抽象了一层jmm模型，主内存（共享），工作内存（线程独享），线程操作都是操作线程中工作内存然后再提交到主内存中。
4. happen-before, jvm层面的内存屏障指令（**storeload**）(**storestore**) (**loadstore**) (**loadload**),happen-before解决编译器层面的优化重排序，具有程序相关性，传递性等特性，volitate关键字本质上是在编译层面加入了内存屏障指令,从而使得变量可见效对所有线程生效.

aqs(同步并发工具)

1. 本质

- **state--锁状态**
- mode(share,exclusive)--共享或独占
- 双线链表实现的同步队列

2. 公平锁，非公平锁，默认情况下都是非公平锁实现

- 非公平体现任何一个线程加入之初就尝试获取锁
- 公平体现在如果存在同步队列元素，直接加入排队

3. 抢锁失败加入同步队列，

```
t = tail;
node.pred = t;
cas(tail=node);
t.next=node;
```

4. 默认唤醒头节点下一个节点，如果不存在下一节点，从**尾巴处遍历**.(尾部遍历是因为插入时后设置next,先设置pred,可能next为空,但是pred一定不为空)
5. condition，在同步队列的前提下，再加入一个等待队列，等待队列里的线程不加入锁竞争,只有从等待队列加入到sync队列才重新加入竞争.

syncronized,重量级锁

1. 锁升级，偏向锁->轻量级锁->重量级锁. （对象头中记录锁状态）
2. 锁升级过程

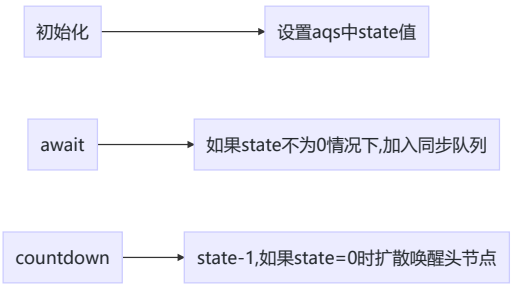
```
线程A
cas设置记录当前线程,偏向锁标示 1

线程B竞争
撤销偏向锁,将锁升级为轻量级锁（01）
自旋尝试获取锁,自适应获取（默认10次
自适应意味着jvm会动态调整这个等待时间）
自旋获取失败会升级为重量级锁（10）
线程阻塞
```

3. 基于jvm层面实现锁，monitor

JUT工具包

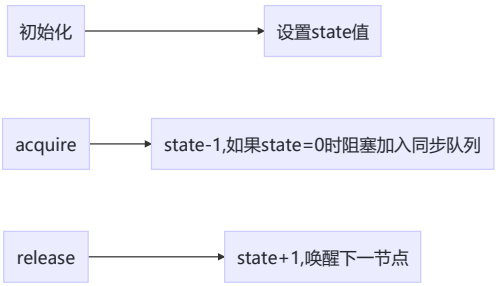
CountDownLatch

- 计数器,计数器为0时才执行后续操作.
- 用法一般有两种
  i. 主线程await,子线程中countdown,用于等待所有线程完成任务执行后续任务.
  ii. 主线程countdown，子线程await,用于等待某个时机同时让线程执行,如构建压测线程.
- 基于AQS实现,share mode



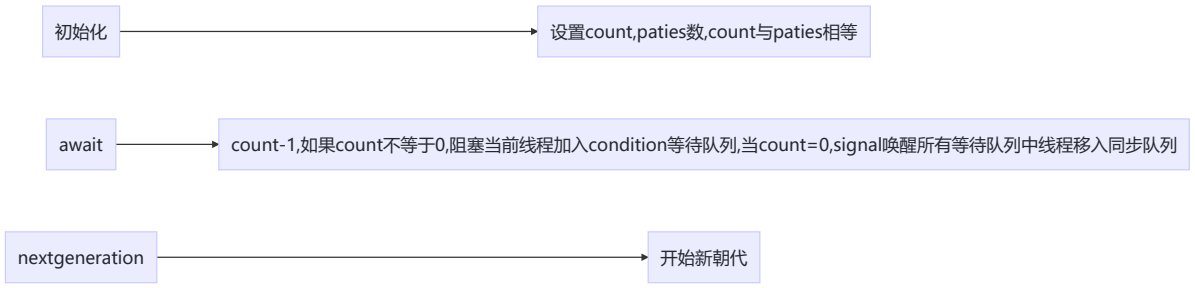
Semaphore

- 信号量,同时只能有n个线程持有共享属性.
- 用法一般可用于简单限流
- 基于AQS share mode 实现



CycliBarrier

- 栅栏，可重复使用，当线程数达到指定条件数量时才往后续执行.
- 用法和CountDownLatch类同
- 和CountDownLatch不同点在于栅栏是可复位重复使用的,且栅栏可指定达到条件后先执行某一操作.
- 基于ReentrantLock、Condition实现
- 复位重用实现是基于generation,复位后开启下一朝代.



阻塞队列

ArrayBlockingQueue

- 添加操作
  - add,队列满了会报错
  - offer，队列满了插入失败
  - put,队列满了阻塞线程
  - offer(e,time,unit)，队列满了阻塞一定时间
- 移除操作
  - remove,移除元素
  - poll，队列为空，返回null
  - take，队列为空，阻塞

    - poll(time,unit),队列为空阻塞一定时间
- 基于两个condition实现 notEmpty,notFull
- 队列其实就是一个数组 array[capacity]
- 记录了putIndex,takeIndex,count(支持循环)

```
private void enqueue(E x) {
// assert lock.getHoldCount() == 1;
// assert items[putIndex] == null;
final Object[] items = this.items;
items[putIndex] = x; //通过putIndex对数据赋
值
if (++putIndex == items.length) // 当putIndex 等于数组长度时，将 putIndex 重置为 0
putIndex = 0;
count++;//记录队列元素的个数
notEmpty.signal();//唤醒处于等待状态下的线程，表 示当前队列中的元素不为空,如果存在消费者线程阻塞，就可以
开始取出元素
}
```

LinkedBlockingQueue

- 基本等于无界队列,队列长度等于Integer.max
- 但是容量是可设置的.

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，否则不能继续添加元素。
- 相当于大小为1到队列.

原子操作

- cas
- volatile
- unsafe类

线程池

线程池核心参数

- coresize
    - 核心线程数
- maxsize
    - 最大线程数
- 空闲时间
    - 超出核心线程会在空闲时间满足时回收
- 拒绝策略
    - 当等待队列满了后当执行策略
    - 四种拒绝策略
        - 拒绝并报错,默认策略
        - 主线程自己执行
        - 抛弃等待队列中最早的,并立即执行当前任务
        - 直接丢弃
- 等待队列
    - 阻塞队列类型

常见四种类型线程池

- singal
    - 单线程
- fix
    - 固定大小

```
new ThreadPoolExecutor(nThreads, nThreads,0L,TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>());
```

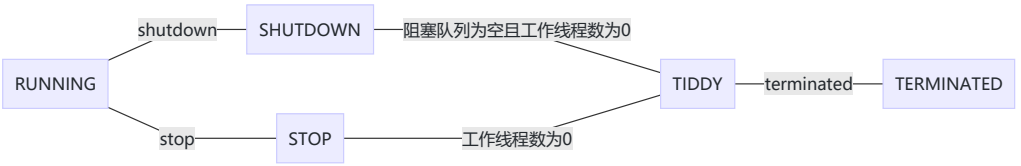
- cached
    - 可自动增加线程当

```
new ThreadPoolExecutor(0, Integer.MAX_VALUE,60L,TimeUnit.SECONDS,new SynchronousQueue<Runnable>());
```

- sheduled
  - 可配置支持定时执行和延迟执行
  - 是实现的ShcheduleExcuteService

线程池实现原理

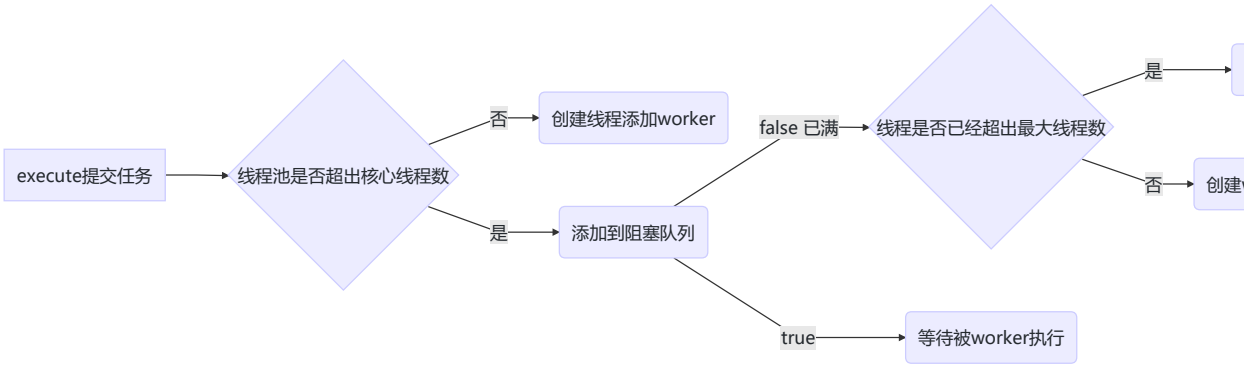
- ctl
  - 前三位存储线程状态
    - RUNNING
    - SHUTDOWN
    - STOP
    - TIDDY
    - TERMINATED
  - 后29位存储工作线程数
  - 状态转化关系



- 线程池中的线程都被封装为worker，且并没有专门标示是否是核心线程.而是通过poll超时且当前工作线程数大于核心线程数则进行回收.

```
//处理销毁线程
 Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();
                if (r != null)
                    return r;
                //记录超时了,下次自旋如果还没有任务则回收线程.
                timedOut = true;
```

- 基本流程链路
  - 创建核心线程若当前工作线程小于核心线程数,否则待执行任务放入阻塞队列中
  - 队列满时尝试添加工作线程
  - 任务闲了回收非核心线程
  - 已经达到最大工作线程数且队列满，执行拒绝策略



线程数设置

- CPU密集型
  - CUP数 + 1
- IO密集型
  - 最佳线程数目 = (线程池设定的线程等待时间+线程 CPU 处理时间)/ 线程 CPU处理 时间)* CPU 数目
    - 线程CPU时间是单线程时运行的平均预估时间（测试出来）