

多线程并发在电商系统下的应用

- 线程相关的基本理论与工具
- 多线程程序下的性能调优
- 电商场景下多线程的使用

1. 多线程 J.U.C

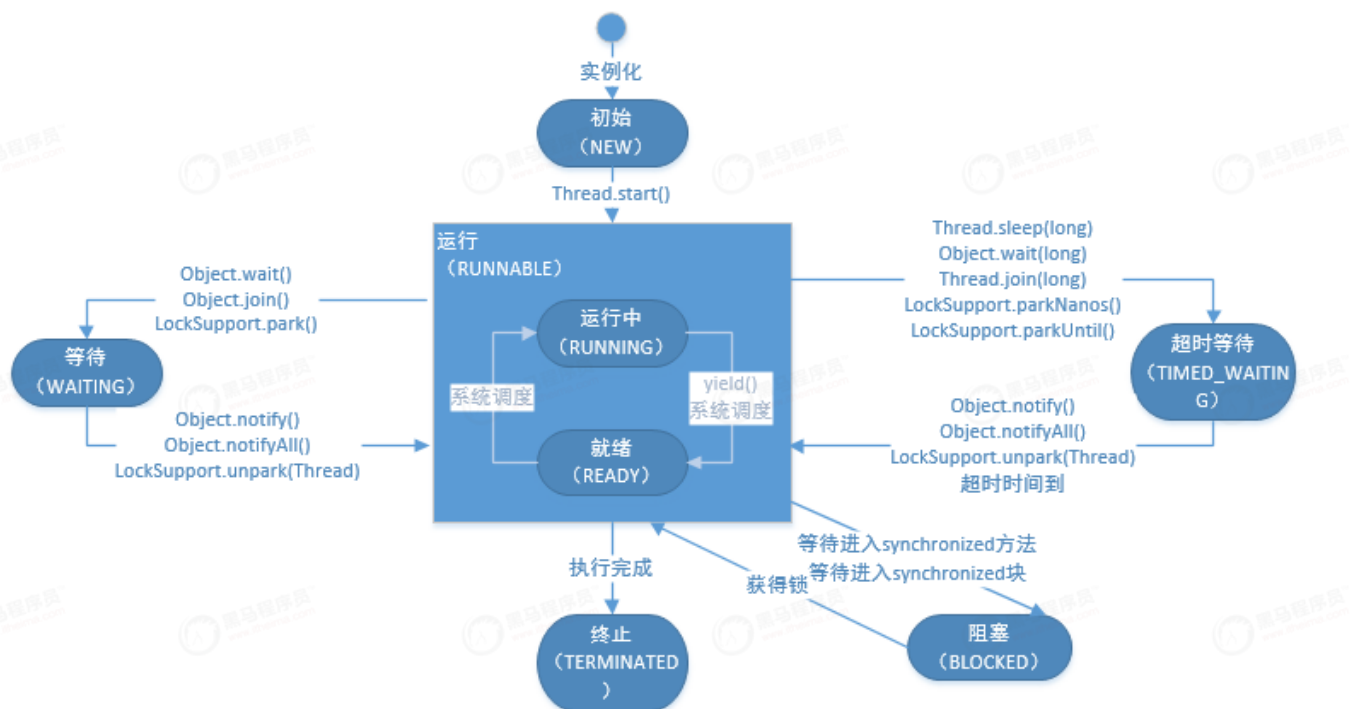
1.1 线程池

1.1.1 概念

1) 回顾线程创建方式

- 继承Thread
- 实现Runnable

2) 线程的状态



- NEW: 刚刚创建，没做任何操作

```
Thread thread = new Thread();
System.out.println(thread.getState());
```

- RUNNABLE: 调用run，可以执行，但不代表一定在执行 (RUNNING, READY)

```
thread.start();
System.out.println(thread.getState());
```

- BLOCKED: 抢不到锁

```
final byte[] lock = new byte[0];
new Thread(new Runnable() {
    public void run() {
        synchronized (lock){
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}).start();
Thread thread2 = new Thread(new Runnable() {
    public void run() {
        synchronized (lock){
        }
    }
});
thread2.start();
Thread.sleep(1000);
System.out.println(thread2.getState());
```

- WAITING

```
Thread thread2 = new Thread(new Runnable() {
    public void run() {
        LockSupport.park();
    }
});

thread2.start();
Thread.sleep(500);
System.out.println(thread2.getState());
LockSupport.unpark(thread2);
Thread.sleep(500);
System.out.println(thread2.getState());
```

- TIMED_WAITING

```
Thread thread3 = new Thread(new Runnable() {
    public void run() {
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
thread3.start();
Thread.sleep(500);
System.out.println(thread3.getState());
```

- TERMINATED

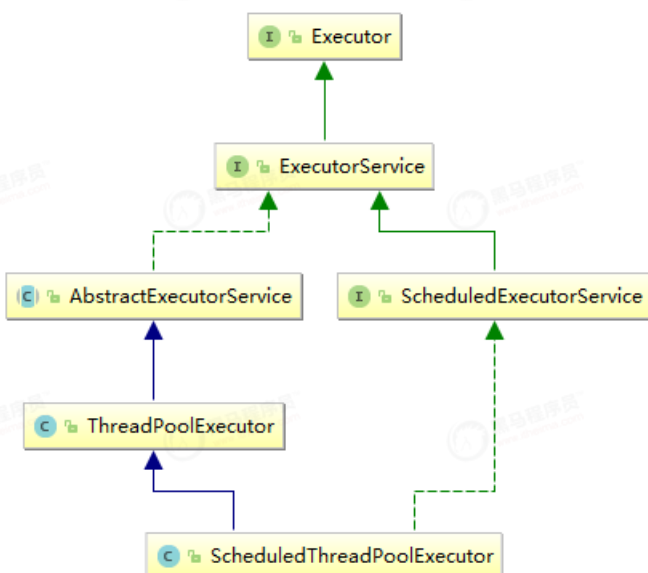
```
//等待1s后再来看
Thread.sleep(1000);
System.out.println(thread.getState());
```

3) 线程池基本概念

根据上面的状态，普通线程执行完，就会进入TERMINATED销毁掉，而线程池就是创建一个缓冲池存放线程，执行结束以后，该线程并不会死亡，而是再次返回线程池中成为空闲状态，等候下次任务来临，这使得线程池比手动创建线程有着更多的优势：

- 降低系统资源消耗，通过重用已存在的线程，降低线程创建和销毁造成的消耗；
- 提高系统响应速度，当有任务到达时，通过复用已存在的线程，无需等待新线程的创建便能立即执行；
- 方便线程并发数的管控。因为线程若是无限制的创建，可能会导致内存占用过多而产生OOM
- 节省cpu切换线程的时间成本（需要保持当前执行线程的现场，并恢复要执行线程的现场）。
- 提供更强大的功能，延时定时线程池。（*Timer vs ScheduledThreadPoolExecutor*）

4) 常用线程池类结构，可以通过idea查看到（查看：*ScheduledThreadPoolExecutor*，*ForkJoinPool*类图）



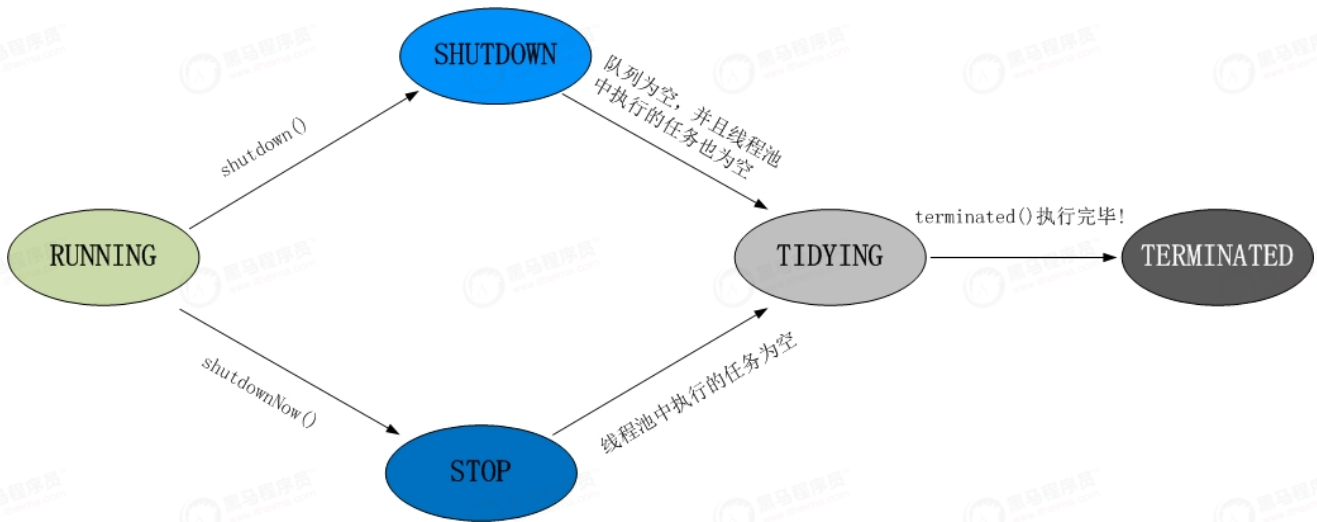
说明：

- 最常用的是ThreadPoolExecutor
- 调度用ScheduledThreadPoolExecutor
- 任务拆分合并用ForkJoinPool
- Executors是工具类，协助你创建线程池的

1.1.2 工作机制

在线程池的编程模式下，任务是提交给整个线程池，而不是直接提交给某个线程，线程池在拿到任务后，就在内部协调空闲的线程，如果有，则将任务交给某个空闲的线程。一个线程同时只能执行一个任务，但可以同时向一个线程池提交多个任务。

1) 线程池状态



- **RUNNING:** 初始化状态是RUNNING。线程池被一旦被创建，就处于RUNNING状态，并且线程池中的任务数为0。RUNNING状态下，能够接收新任务，以及对已添加的任务进行处理。
- **SHUTDOWN:** SHUTDOWN状态时，不接收新任务，但能处理已添加的任务。调用线程池的shutdown()接口时，线程池由RUNNING -> SHUTDOWN。

//shutdown后不接受新任务，但是task1，仍然可以执行完成

```
ExecutorService poolExecutor = Executors.newFixedThreadPool(5);
poolExecutor.execute(new Runnable() {
    public void run() {
        try {
            Thread.sleep(1000);
            System.out.println("finish task 1");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
poolExecutor.shutdown();
poolExecutor.execute(new Runnable() {
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
System.out.println("ok");
```

- **STOP:** 不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。调用线程池的shutdownNow()接口时，线程池由(RUNNING 或 SHUTDOWN) -> STOP

注意：运行中的任务还会打印，直到结束，因为调的是Thread.interrupt

//改为shutdownNow后，任务立马终止，sleep被打断，新任务无法提交，task1停止

```
poolExecutor.shutdownNow();
```

- **TIDYING:** 所有的任务已终止，队列中的“任务数量”为0，线程池会变为TIDYING。线程池变为TIDYING状态时，会执行钩子函数terminated()，可以通过重载terminated()函数来实现自定义行为

```
//自定义类, 重写terminated方法
public class MyExecutorService extends ThreadPoolExecutor {

    public MyExecutorService(int corePoolSize, int maximumPoolSize, long keepAliveTime,
        TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
    }

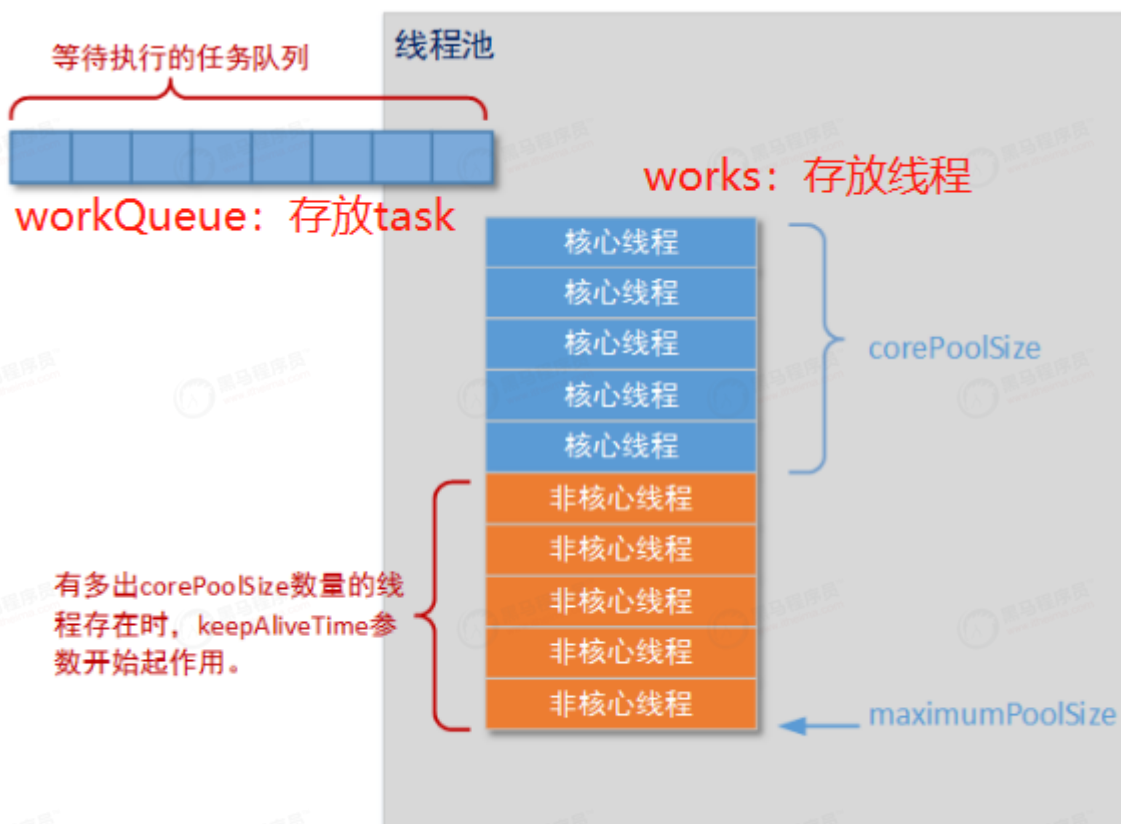
    @Override
    protected void terminated() {
        super.terminated();
        System.out.println("terminated");
    }

    //调用 shutdownNow, terminated方法被调用打印
    public static void main(String[] args) throws InterruptedException {
        MyExecutorService service = new MyExecutorService(1,2,10000,TimeUnit.SECONDS,new
        LinkedBlockingQueue<Runnable>(5));
        service.shutdownNow();
    }
}
```

- **TERMINATED:** 线程池处在TIDYING状态时, 执行完terminated()-后, 就会由 TIDYING -> TERMINATED

2) 结构说明

(源码查看: 两个集合, 一个queue, 一个hashset)



3) 任务的提交

- 添加任务，如果线程池中线程数没达到coreSize，直接创建新线程执行
- 达到core，放入queue
- queue已满，未达到maxSize继续创建线程
- 达到maxSize，根据reject策略处理
- 超时后，线程被释放，下降到coreSize

1.1.3 源码剖析

//任务提交阶段：（4个if条件路线）

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    int c = ctl.get();
    //判断工作数，如果小于coreSize，addWork，注意第二个参数core=true
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    //否则，如果线程池还在运行，offer到队列
    if (isRunning(c) && workQueue.offer(command)) {
        //再检查一下状态
        int recheck = ctl.get();
        //如果线程池已经终止，直接移除任务，不再响应
        if (!isRunning(recheck) && remove(command))
            reject(command);
        //否则，如果没有线程干活的话，创建一个空work，该work会从队列获取任务去执行
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    //队列也满，继续调addWork，但是注意，core=false，开启到maxSize的大门
    //超出max的话，addWork会返回false，进入reject
    else if (!addWorker(command, false))
        reject(command);
}
```

//线程创建

```
private boolean addWorker(Runnable firstTask, boolean core) {  
    //第一步，计数判断，不符合条件打回false  
    retry:  
    for (;;) {  
        int c = ctl.get();  
        int rs = runStateOf(c);  
  
        // Check if queue empty only if necessary.  
        if (rs >= SHUTDOWN &&  
            ! (rs == SHUTDOWN &&  
                firstTask == null &&  
                ! workQueue.isEmpty()))  
            return false;  
  
        for (;;) {  
            int wc = workerCountOf(c);  
            //判断线程数，注意这里！  
            //也就说明线程池的线程数是不可能设置任意大的。  
            //最大29位（CAPACITY=29位二进制）  
            if (wc >= CAPACITY ||  
                wc >= (core ? corePoolSize : maximumPoolSize))  
                return false;  
            if (compareAndIncrementWorkerCount(c))  
                break retry;  
            c = ctl.get(); // Re-read ctl  
            if (runStateOf(c) != rs)  
                continue retry;  
            // else CAS failed due to workerCount change; retry inner loop  
        }  
    }  
    //第二步，创建新work放入线程集合works（一个HashSet）  
    boolean workerStarted = false;  
    boolean workerAdded = false;  
    Worker w = null;  
    try {  
        //符合条件，创建新的work并包装task  
        w = new Worker(firstTask);  
        final Thread t = w.thread;  
        if (t != null) {  
            final ReentrantLock mainLock = this.mainLock;  
            mainLock.lock();  
            try {  
                // Recheck while holding lock.  
                // Back out on ThreadFactory failure or if  
                // shut down before lock acquired.  
                int rs = runStateOf(ctl.get());  
  
                if (rs < SHUTDOWN ||  
                    (rs == SHUTDOWN && firstTask == null)) {  
                    if (t.isAlive()) // precheck that t is startable  
                        throw new IllegalThreadStateException();  
                    workerStarted = true;  
                    workerAdded = true;  
                    works.add(w);  
                    w.start();  
                }  
            } finally {  
                mainLock.unlock();  
            }  
        }  
    } catch (Throwable t) {  
        // workerStartFailed() called by worker thread  
        return false;  
    }  
    return workerStarted;  
}
```



```
        //在这里!!!
        workers.add(w);
        int s = workers.size();
        if (s > largestPoolSize)
            largestPoolSize = s;
        workerAdded = true;
    }
} finally {
    mainLock.unlock();
}
if (workerAdded) {
    //注意，只要是成功add了新的work，那么将该新work立即启动，任务得到执行
    t.start();
    workerStarted = true;
}
}
} finally {
    if (! workerStarted)
        addWorkerFailed(w);
}
return workerStarted;
}
```

//任务获取与执行

//在worker执行runWorker()的时候，不停循环，先查看自己有没有携带Task，如果有，执行
while (task != null || (task = getTask()) != null)

//如果没用，会调用getTask，从队列获取任务

```
private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

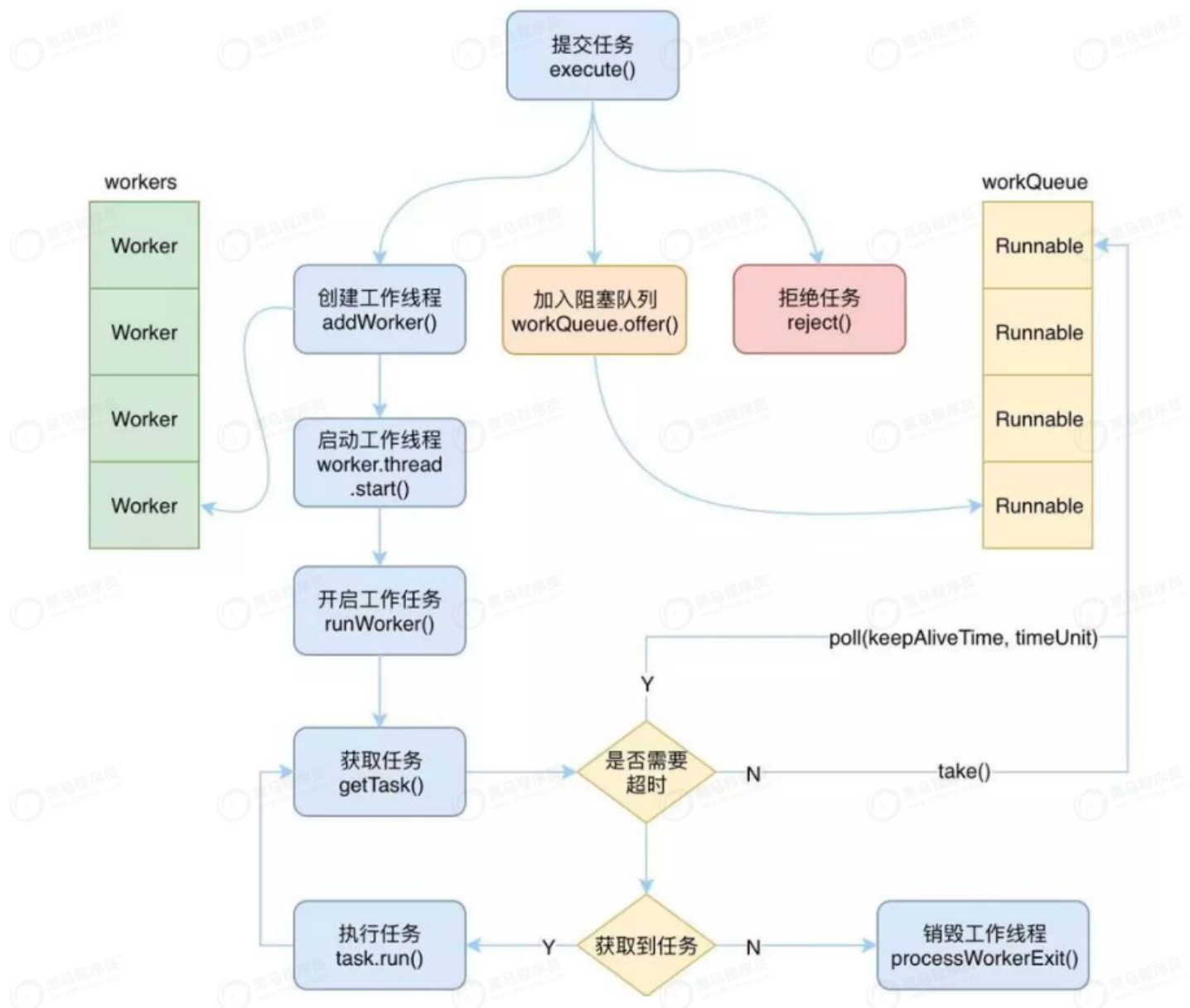
        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);

        // Are workers subject to culling?
        //判断是不是要超时处理，重点!!! 决定了当前线程要不要被释放
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
        //线程数超出max，并且上次循环中poll等待超时了，那么说明该线程已终止
        //将线程队列数量原子性减
        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }

        try {
            //重点!!!
            //如果线程可被释放，那就poll，释放的时间为：keepAliveTime
            //否则，线程是不会被释放的，take一直被阻塞在这里，知道来了新任务继续工作
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
            if (r != null)
                return r;
            //到这里说明可被释放的线程等待超时，已经销毁，设置该标记，下次循环将线程数减少
            timedOut = true;
        } catch (InterruptedException retry) {
            timedOut = false;
        }
    }
}
```

完整流程回顾：



1.1.4 注意点

1) 线程池是如何保证线程不被销毁的呢？

答案：如果队列中没有任务时，核心线程会一直阻塞在获取任务的方法，直到返回任务。而任务执行完后，又会进入下一轮 `work.runWork()` 中循环

验证：秘密就藏在核心源码里 `ThreadPoolExecutor.getTask()`

```

//work.runWork():
while (task != null || (task = getTask()) != null)

//work.getTask():
boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
Runnable r = timed ?
    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
    workQueue.take();
  
```

2) 那么线程池中的线程会处于什么状态?

答案: TIMED_WAITING, RUNNABLE, WAITING

验证: 起一个线程池, 放置一个任务sleep, debug查看结束前后的状态

```
//debug add watcher:  
((ThreadPoolExecutor) poolExecutor).workers.iterator().next().thread.getState()
```

```
ThreadPoolExecutor poolExecutor = Executors.newFixedThreadPool(5);  
poolExecutor.execute(new Runnable() {  
    public void run() {  
        try {  
            Thread.sleep(5000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
});  
System.out.println("ok");
```

3) 核心线程与非核心线程有区别吗?

答案: 没有。被销毁的线程和创建的先后无关。即便是第一个被创建的核心线程, 仍然有可能被销毁

验证: 看源码, 每个works在runWork的时候去getTask, 在getTask内部, 并没有针对性的区分当前work是否是核心线程或者类似的标记。只要判断works数量超出core, 就会调用poll(), 否则take()

1.1.5 Executors工具

以上构造函数比较多, 为了方便使用, 提供了一个Executors工具类

- 1) newCachedThreadPool(): 弹性线程数
- 2) newFixedThreadPool(int nThreads): 固定线程数
- 3) newSingleThreadExecutor(): 单一线程数
- 4) newScheduledThreadPool(int corePoolSize): 可调度, 常用于定时

1.2 锁

1.2.1 概述

锁是一种互斥的机制, 在多线程环境中实现对资源的协调与控制, 凡是有资源被多线程共享, 涉及到你改我改的情况就要考虑锁的加持。

从一个案例看起, 在写代码的时候, 不注意往往会遇到以下代码...

- 1) 糟糕的实现

```

package com.itheima;

public class BadCounter {
    private static int i=0;
    public int get(){
        return i;
    }
    public void inc(){
        int j=get();
        try {
            Thread.sleep(100);
            j++;
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        i=j;
    }

    public static void main(String[] args) throws InterruptedException {
        final BadCounter counter = new BadCounter();
        //不使用线程10次，对比使用线程10次，看结果
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                public void run() {
                    counter.inc();
                }
            }).start();
        }
        Thread.sleep(3000);
        //理论上10才对。可是....
        System.out.println(counter.i);
    }
}

```

出问题了....

在遍地spring bean的年代，单例模式下的类变量尤其要注意！

1.2.2 实现方式

1) synchronized

```

//加synchronized，再测试
public synchronized void inc()

```

2) Lock

```
//换lock方式测试
Lock lock = new ReentrantLock();
public void inc() {
    lock.lock();

    //...

    lock.unlock();
}
```

无论哪种方式加锁均能实现正确计数，但是这个性能实在是感人，后面调优还会提到。

1.2.3 锁的分类及详解

1) 乐观锁/悲观锁

乐观锁顾名思义，很乐观的认为每次读取数据的时候总是认为没人动过，所以不去加锁。但是在更新的时候回去对比一下原来的值，看有没有被别人更改过。适用于读多写少的场景。

mysql中类比version号更新 `update xxx set a=aaa where id=xx and version=1`

java中的atomic包属于乐观锁实现，即CAS（下节会详细介绍）

悲观锁在每次读取数据的时候都认为其他人会修改数据，所以读取数据的时候也加锁，这样别人想拿的时候就会阻塞，直到这个线程释放锁，这就影响了并发性能。适合写操作比较多的场景。

mysql中类比for select xxx for update; update update xx set a = aaa

案例中synchronized实现就是悲观锁（1.6之后优化为锁升级机制），悲观锁书写不当很容易影响性能（性能部分会讲到）

2) 独享锁/共享锁

很好理解，独享锁是指该锁一次只能被一个线程所持有，而共享锁是指该锁可被多个线程所持有。

案例一：ReentrantLock，独享锁

```

package com.itheima;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PrivateLock {
    Lock lock = new ReentrantLock();
    long start = System.currentTimeMillis();
    void read() {
        lock.lock();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
        System.out.println("read time = " + (System.currentTimeMillis() - start));
    }

    public static void main(String[] args) {
        final PrivateLock lock = new PrivateLock();
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                public void run() {
                    lock.read();
                }
            }).start();
        }
    }
}

```

结果分析：每个线程结束的时间点逐个上升，锁被独享，一个用完下一个，依次获取锁

```

end time = 104
end time = 204
end time = 306
end time = 407
end time = 508
end time = 609
end time = 710
end time = 811
end time = 912
end time = 1013

```

案例二：ReadWriteLock，read共享，write独享

```

package com.itheima;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class SharedLock {
    ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    Lock lock = readWriteLock.readLock();
    long start = System.currentTimeMillis();
    void read() {
        lock.lock();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
        System.out.println("end time = " + (System.currentTimeMillis() - start));
    }

    public static void main(String[] args) {
        final SharedLock lock = new SharedLock();
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                public void run() {
                    lock.read();
                }
            }).start();
        }
    }
}

```

结果分析：每个线程独自跑，各在100ms左右，证明是共享的


```
end time = 105
end time = 105
end time = 105
end time = 105
end time = 105
end time = 106
end time = 107
end time = 107
end time = 107
end time = 107
```

案例三：同样是上例，换成writeLock

```
lock lock = readWriteLock.writeLock();
```

结果分析：恢复到了1s时长，变为独享

```
end time = 104
end time = 205
end time = 306
end time = 407
end time = 519
end time = 620
end time = 721
end time = 821
end time = 922
end time = 1023
```

小节：

- 读锁的共享锁可保证并发读是非常高效的，读写，写读，写写的过程是互斥的。
- 独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

3) 分段锁

从Map一家子说起....

HashMap是线程不安全的，在多线程环境下，使用HashMap进行put操作时，可能会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。

于是有了HashTable，HashTable是线程安全的。但是HashTable线程安全的策略实在不怎么高明，将get/put等所有相关操作都整成了synchronized的。

```

/**...*/
public synchronized V put(K key, V value) {...}

/**...*/
public synchronized V remove(Object key) {...}

/**...*/
public synchronized void putAll(Map<? extends K, ? extends V> t) {...}

/**...*/
public synchronized void clear() {...}

/**...*/
public synchronized Object clone() {...}

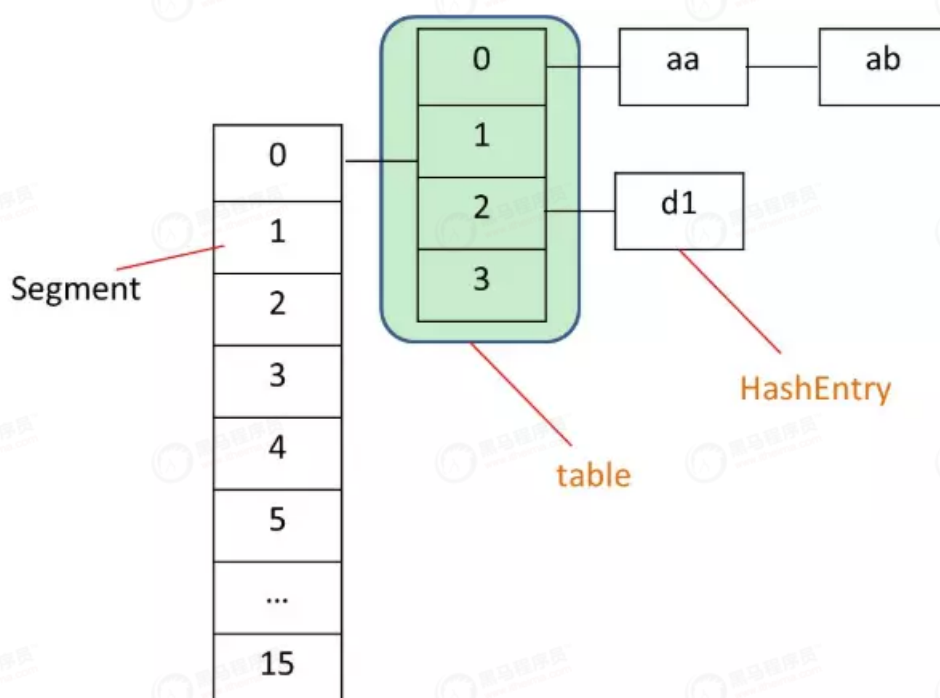
/**...*/
public synchronized String toString() {...}

```

那有没有办法做到线程安全，又不这么粗暴呢？基于分段锁的ConcurrentHashMap诞生...

ConcurrentHashMap使用Segment（分段锁）技术，将数据分成一段一段的存储，Segment数组的意义就是将一个大的table分割成多个小的table来进行加锁，Segment数组中每一个元素一把锁，每一个Segment元素存储的是HashEntry数组+链表，这个和HashMap的数据存储结构一样。当访问其中一个段数据被某个线程加锁的时候，其他段的数据也能被其他线程访问，这就使得ConcurrentHashMap不仅保证了线程安全，而且提高了性能。

但是这也引来一个负面影响：ConcurrentHashMap 定位一个元素的过程需要进行两次Hash操作，第一次 Hash 定位到 Segment，第二次 Hash 定位到元素所在的链表。所以 Hash 的过程比普通的 HashMap 要长。



备注: `JDK1.8ConcurrentHashMap` 中抛弃了原有的 `Segment` 分段锁, 而采用了 `CAS + synchronized` 来保证并发安全性。

4) 可重入锁

可重入锁指的获取到锁后, 如果同步块内需要再次获取同一把锁的时候, 直接放行, 而不是等待。其意义在于防止死锁。前面使用的 `synchronized` 和 `ReentrantLock` 都是可重入锁。

实现原理实现是通过为每个锁关联一个请求计数器和一个占有它的线程。如果同一个线程再次请求这个锁, 计数器将递增, 线程退出同步块, 计数器值将递减。直到计数器为0锁被释放。

场景见于父类和子类的锁的重入 (调 `super` 方法), 以及多个加锁方法的嵌套调用。

案例一: 父子可重入

```
package com.itheima;

public class ParentLock {
    byte[] lock = new byte[0];
    public void f1(){
        synchronized (lock){
            System.out.println("f1 from parent");
        }
    }
}
```

```
package com.itheima;

public class SonLock extends ParentLock {
    public void f1() {
        synchronized (super.lock){
            super.f1();
            System.out.println("f1 from son");
        }
    }
    public static void main(String[] args) {
        SonLock lock = new SonLock();
        lock.f1();
    }
}
```

案例二: 内嵌方法可重入

```
package com.itheima;

public class NestedLock {
    public synchronized void f1(){
        System.out.println("f1");
    }
    public synchronized void f2(){
        f1();
        System.out.println("f2");
    }

    public static void main(String[] args) {
        NestedLock lock = new NestedLock();
        //可以正常打印 f1,f2
        lock.f2();
    }
}
```

案例三：不可重入锁的典型错误，不要这么做！！

先看代码，猜一猜结果？

```

public class BadLock{
    Lock lock = new Lock();
    public void f1(){
        System.out.println("f1");
        lock.lock();
        f2();
        lock.unlock();
    }
    public void f2(){
        lock.lock();
        System.out.println("f2");
        lock.unlock();
    }

    public static void main(String[] args) {
        BadLock badLock = new BadLock();
        //理论上，会打印 f1 和 f2
        //实际上，这个错误的设计会导致卡死在f1
        badLock.f1();
    }

    //自定义的锁，现实中不要这么做!!!
    class Lock{
        private boolean isLocked = false;
        public synchronized void lock(){
            try {
                //想要拿锁，一直判断标记，如果被占就wait等待
                while(isLocked) {
                    wait();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //一旦被唤醒，退出while了，自己拿到锁，将标记改为true（已占用）
            isLocked = true;
        }
        public synchronized void unlock(){
            //占用标记改成false
            isLocked = false;
            //同时唤醒等待锁的线程
            notify();
        }
    }
}

```

5) 公平锁/非公平锁

基本概念:

常见于AQS，公平锁就是在并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空，或者当前线程是等待队列的第一个，就占有锁，否则就会加入到等待队列中，直到按照FIFO的规则从队列中取到自己。

非公平锁与公平锁基本类似，只是在放入队列前先判断当前锁是否被线程持有。如果锁空闲，那么他可以直接抢占，而不需要判断当前队列中是否有等待线程。只有锁被占用的话，才会进入排队。

在现实中想象一下游乐场旋转木马排队现象.....

优缺点：

公平锁的优点是等待锁的线程不会饿死，进入队列规规矩矩的排队，迟早会轮到。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU唤醒阻塞线程的开销比非公平锁大。

非公平锁的性能要高于公平锁，因为线程有几率不阻塞直接获得锁。ReentrantLock默认使用非公平锁就是基于性能考量。但是非公平锁的缺点是可能引发队列中的线程始终拿不到锁，一直排队被饿死。

编码方式：

很简单，ReentrantLock支持创建公平锁和非公平锁（默认），想要实现公平锁，使用new ReentrantLock(true)。

背后原理：

AQS，后面还会详细讲到。AQS中有一个state标识锁的占用情况，一个队列存储等待线程。

state=0表示锁空闲。如果是公平锁，那就看看队列有没有线程在等，有的话不参与竞争乖乖追加到尾部。如果是非公平锁，那就直接参与竞争，不管队列有没有等待者。

state>0表示有线程占着锁，这时候无论公平与非公平，都直接去排队（想抢也没有）

备注：

因为ReentrantLock是可以定义公平非公平锁，次数。所以是>0而不是简单的0和1

而synchronized只能是非公平锁

6) 锁升级

java中每个对象都可作为锁，锁有四种级别，按照量级从轻到重分为：无锁、偏向锁、轻量级锁、重量级锁。

如何理解呢？A占了锁，B就要阻塞等。但是，在操作系统中，阻塞就要存储当前线程状态，唤醒就要再恢复，这个过程是要消耗时间的...

如果A使用锁的时间远远小于B被阻塞和挂起的执行时间，那么我们将B挂起阻塞就相当的不合算。

于是出现自旋：自旋指的是锁已经被其他线程占用时，当前线程不会被挂起，而是在不停的试图获取锁（可以理解为不停的循环），每循环一次表示一次自旋过程。显然这种操作会消耗CPU时间，但是相比线程上下文切换时间要少的时候，自旋划算。

而偏向锁、轻量锁、重量锁就是围绕如何使得cpu的占用更划算而展开的。

举个生活的例子，假设公司只有一个会议室（共享资源）

偏向锁：

前期公司只有1个团队，那么什么时候开会都能满足，就不需要询问和查看会议室的占用情况，直接进入使用状态。会议室门口挂了个牌子写着A使用，A默认不需要预约（ThreadID=A）

轻量级锁：

随着业务发展，扩充为2个团队，B团队肯定不会同意A无法无天，于是当AB同时需要开会时，两者竞争，谁抢到谁算谁的。偏向锁升级为轻量级锁，但是未抢到者在门口会不停敲门询问（自旋，循环），开完没有？开完没有？

重量级锁：

后来发现，这种不停敲门的方式很烦，A可能不理不睬，但是B要不停的闹腾。于是锁再次升级。

如果会议室被A占用，那么B团队直接闭嘴，在门口安静的等待（wait进入阻塞），直到A用完后会通知B（notify）。

注意点：

- 上面几种锁都是JVM自己内部实现，我们不需要干预，但是可以配置jvm参数开启/关闭自旋锁、偏向锁。
- 锁可以升级，但是不能反向降级：偏向锁→轻量级锁→重量级锁
- 无锁争用的时候使用偏向锁，第二个线程到了升级为轻量级锁进行竞争，更多线程时，进入重量级锁阻塞

偏重场景：

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要CAS操作，没有额外的性能消耗，和执行非同步方法相比仅存在纳秒级的差距	若线程间存在锁竞争，会带来额外的锁撤销的消耗	只有一个线程访问同步块或者同步方法
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	若线程长时间竞争不到锁，自旋会消耗 CPU 性能	线程交替执行同步块或者同步方法，追求响应时间，锁占用时间很短，阻塞还不如自旋的场景
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢，在多线程下，频繁的获取释放锁，会带来巨大的性能消耗	追求吞吐量，锁占用时间较长

8）互斥锁/读写锁

- 典型的互斥锁：synchronized，ReentrantLock，读写锁：ReadWriteLock 前面都用过了
- 互斥锁属于独享锁，读写锁里的写锁属于独享锁，而读锁属于共享锁

案例：互斥锁用不好可能会失效，看一个典型的锁不住现象！


```

package com.itheima;

public class ObjectLock {
    public static Integer i=0;
    public void inc(){
        synchronized (this){
            int j=i;
            try {
                Thread.sleep(100);
                j++;
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            i=j;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                public void run() {
                    //重点!
                    new ObjectLock().inc();
                }
            }).start();
        }
        Thread.sleep(3000);
        //理论上10才对。可是....
        System.out.println(ObjectLock.i);
    }
}

```

结果分析：每个线程内都是new对象，所以this不是同一把锁，结果锁不住，输出1

- this，换成static的 i 变量试试？
- 换成ObjectLock.class 试试？
- 换成String.class
- 去掉synchronized块，外部方法上加 static synchronized

1.2.4 AQS

1) 概念

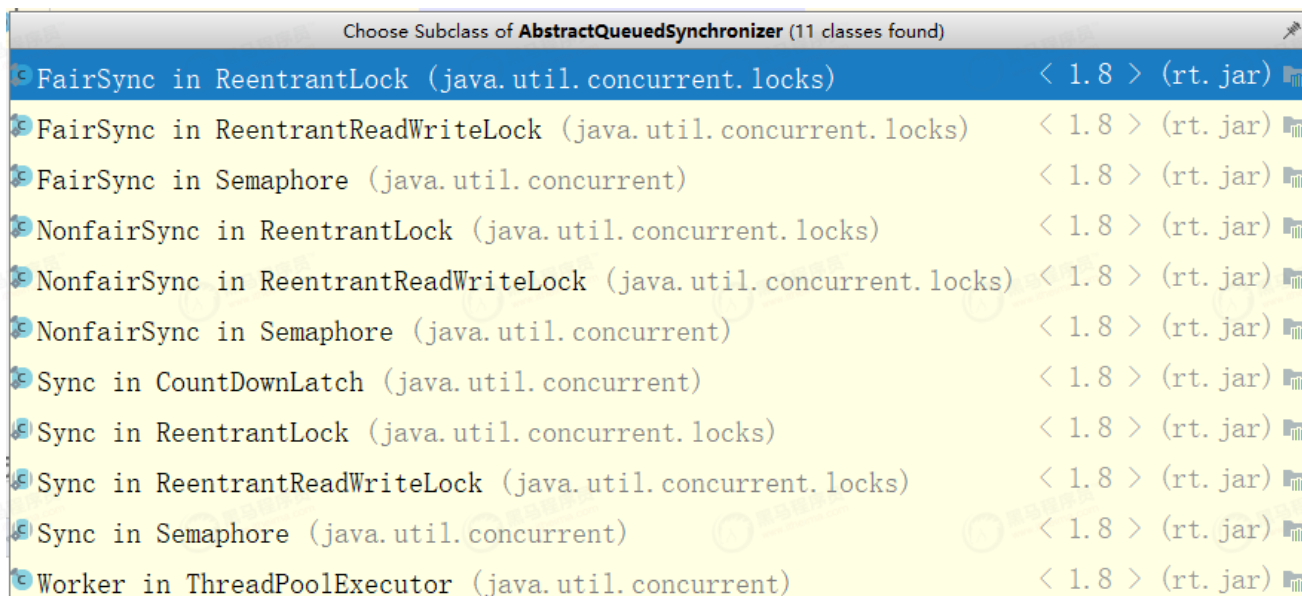
首先搞清楚，AbstractQueuedSynchronizer抽象的队列式同步器，是一个抽象类，这个类在java.util.concurrent.locks包。除了java自带的synchronized关键字之外，jdk提供的另外一种锁机制。如果需要自己实现锁的逻辑，可以考虑使用AQS，非常的便捷。

```

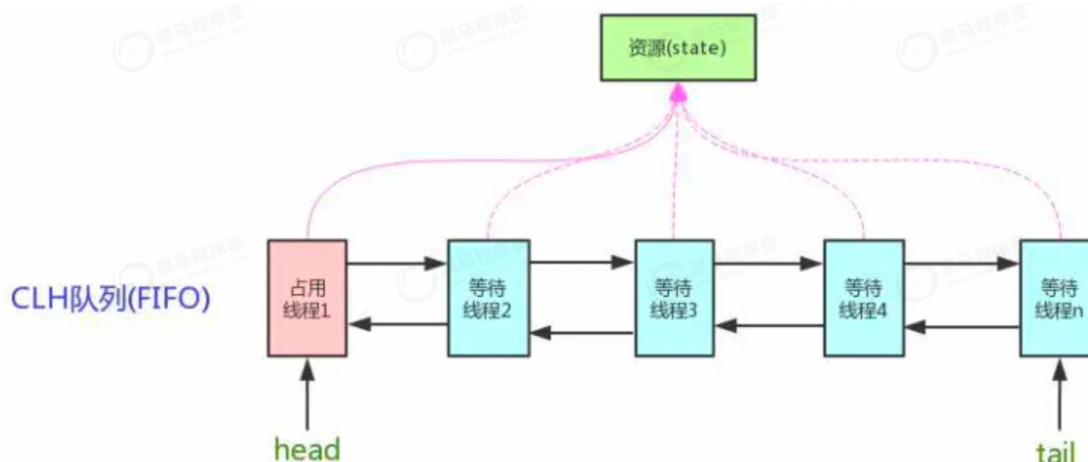
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer
    implements java.io.Serializable

```

jdk中使用AQS的线程工具类很多，自旋锁、互斥锁、读锁写锁、信号量、通过类继承关系可以轻松查看：

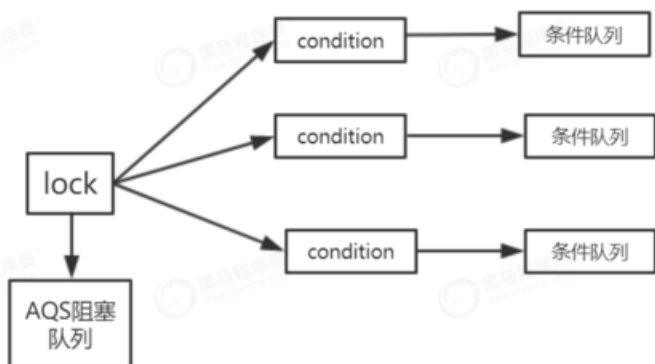


2) 原理



- AQS使用一个int类型的成员变量state来表示同步状态，当state>0时表示已经获取了锁，当state = 0时表示释放了锁。它提供了三个方法（getState()、setState(int newState)、compareAndSetState(int expect,int update)）来对同步状态state进行操作
- AQS通过内置的FIFO同步队列来完成资源获取线程的排队工作，如果当前线程获取同步状态失败（锁）时，AQS则会将当前线程以及等待状态等信息构造成一个节点（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，则会把节点中的线程唤醒，使其再次尝试获取同步状态。
- CLH（Craig, Landin, and Hagersten）队列是一个虚拟的双向队列，即不存在队列实例，使用前后节点和指针来实现关联。

拓展：AQS中有两个队列，这里的阻塞队列，还有多个ConditionObject维护的条件队列。和condition的await/signal有关



3) 实现方式

AQS使用了模板设计模式。只需要实现指定的锁获取方法即可，内部的机制AQS已帮你封装好。

(AQS源码idea中查看)

需要子类继承AQS，并实现的方法（protected）：

- tryAcquire(int arg): 独占式获取同步状态，其他线程需要等待该线程释放同步状态
- tryRelease(int arg): 独占式释放同步状态
- tryAcquireShared(int arg): 共享式获取同步状态，返回值大于等于0则表示获取成功，否则获取失败
- tryReleaseShared(int arg): 共享式释放同步状态

使用时，调用的是父类的方法（public）

- acquire(int arg): 独占式获取
- release(int arg): 独占式释放
- acquireShared(int arg): 共享式获取
- releaseShared(int arg): 共享式释放

4) 源码分析

```

public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer
    implements java.io.Serializable {
    //可共享式获取锁，外部调用，模板模式
    public final void acquireShared(int arg) {
        if (tryAcquireShared(arg) < 0)
            doAcquireShared(arg);
    }
    //需要实现的部分，空protected方法，被上面的对外方法所调用
    protected int tryAcquireShared(int arg) {
        throw new UnsupportedOperationException();
    }

    //同理，锁的释放，模板模式
    public final boolean releaseShared(int arg) {
        if (tryReleaseShared(arg)) {
            doReleaseShared();
            return true;
        }
        return false;
    }
    protected boolean tryReleaseShared(int arg) {
        throw new UnsupportedOperationException();
    }

    //独占式获取
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
    protected boolean tryAcquire(int arg) {
        throw new UnsupportedOperationException();
    }

    //独占式释放
    public final boolean release(int arg) {
        if (tryRelease(arg)) {
            Node h = head;
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);
            return true;
        }
        return false;
    }
    protected boolean tryRelease(int arg) {
        throw new UnsupportedOperationException();
    }
}

```

4) 场景案例

用AQS自己实现一个锁，最大允许指定数量的线程并行运作。其他排队等候

```
package com.itheima;

import java.util.concurrent.locks.AbstractQueuedSynchronizer;

public class MyLock extends AbstractQueuedSynchronizer {
    public MyLock(int count){
        setState(count);
    }

    @Override
    protected int tryAcquireShared(int arg) {
        for (; ; ) {
            int current = getState();
            int newCount = current - arg;
            if (newCount < 0 || compareAndSetState(current, newCount)) {
                return newCount;
            }
        }
    }

    @Override
    protected boolean tryReleaseShared(int arg) {
        for (; ; ) {
            int current = getState();
            int newState = current + arg;
            if (compareAndSetState(current, newState)) {
                return true;
            }
        }
    }

    public static void main(String[] args) {
        final MyLock lock = new MyLock(3);

        for (int i = 0; i < 30; i++) {
            new Thread(new Runnable() {
                public void run() {
                    lock.acquireShared(1);
                    try {
                        Thread.currentThread().sleep(1000);
                        System.out.println("ok");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        lock.releaseShared(1);
                    }
                }
            }).start();
        }
    }
}
```

验证结果：虽然30个一次性start，但是会每1s输出3个ok，达到了并发控制

1.3 原子操作(atomic)

1.3.1 概念

原子（atom）本意是“不能被进一步分割的最小粒子”，而原子操作（atomic operation）意为“不可被中断的一个或一系列操作”。类比于数据库事务，redis的multi。

1.3.2 CAS

Compare And Set（或Compare And Swap），翻译过来就是比较并替换，CAS操作包含三个操作数——内存位置（V）、预期原值（A）、新值(B)。从第一视角来看，理解为：我认为位置 V 应该是 A，如果是A，则将 B 放到这个位置；否则，不要更改，只告诉我这个位置现在的值即可。

计数器问题发生归根结底是取值和运算后的赋值中间，发生了插队现象，他们不是原子的操作。前面的计数器使用加锁方式实现了正确计数，下面，基于CAS的原子类上场....

```
package com.itheima;

import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter {
    private static AtomicInteger i = new AtomicInteger(0);
    public int get(){
        return i.get();
    }
    public void inc(){
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        i.incrementAndGet();
    }

    public static void main(String[] args) throws InterruptedException {
        final AtomicCounter counter = new AtomicCounter();
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                public void run() {
                    counter.inc();
                }
            }).start();
        }
        Thread.sleep(3000);
        //同样可以正确输出10
        System.out.println(counter.i.get());
    }
}
```

1.3.3 atomic

上面展示了AtomicInteger，关于atomic包，还有很多其他类型：

- 基本类型
 - AtomicBoolean：以原子更新的方式更新boolean；
 - AtomicInteger：以原子更新的方式更新Integer；
 - AtomicLong：以原子更新的方式更新Long；
- 引用类型
 - AtomicReference：原子更新引用类型
 - AtomicReferenceFieldUpdater：原子更新引用类型的字段
 - AtomicMarkableReference：原子更新带有标志位的引用类型
- 数组
 - AtomicIntegerArray：原子更新整型数组里的元素。
 - AtomicLongArray：原子更新长整型数组里的元素。
 - AtomicReferenceArray：原子更新引用类型数组里的元素。
- 字段
 - AtomicIntegerFieldUpdater：原子更新整型的字段的更新器。
 - AtomicLongFieldUpdater：原子更新长整型字段的更新器。
 - AtomicStampedReference：原子更新带有版本号的引用类型。

1.3.4 注意！

使用atomic要注意原子性的边界，把握不好会起不到应有的效果，原子性被破坏。

案例：原子性被破坏现象


```

package com.itheima;

import java.util.concurrent.atomic.AtomicInteger;

public class BadAtomic {
    AtomicInteger i = new AtomicInteger(0);
    static int j=0;

    public void badInc(){
        int k = i.incrementAndGet();
        try {
            Thread.sleep(new Random().nextInt(100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        j=k;
    }

    public static void main(String[] args) throws InterruptedException {
        BadAtomic atomic = new BadAtomic();
        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                atomic.badInc();
            }).start();
        }
        Thread.sleep(3000);
        System.out.println(atomic.j);
    }
}

```

结果分析:

- 每次都不同，总之不是10
- 在badInc上加synchronized，问题解决

1.4 ThreadLocal

1.4.1 概念

ThreadLocal类并不是用来解决多线程环境下的共享变量问题，而是用来提供线程内部的共享变量。在多线程环境下，可以保证各个线程之间的变量互相隔离、相互独立。

1.4.2 使用

ThreadLocal实例一般定义为private static类型的，在一个线程内，该变量共享一份，类似上下文作用，可以用来上下传递信息。

```

package com.itheima.local;

public class ThreadLocalDemo implements Runnable{
    private static ThreadLocal<Integer> threadLocal = new ThreadLocal<>();
    public void run(){
        for (int i = 0; i < 3; i++) {
            threadLocal.set(i);
            System.out.println(Thread.currentThread().getName()+" value="+threadLocal.get());
        }
    }

    public static void main(String[] args) {
        ThreadLocalDemo demo = new ThreadLocalDemo();
        new Thread(demo).start();
        new Thread(demo).start();
    }
}

```

结果分析:

- 同一个demo实例，不同的thread嵌套
- 结果打印了各自的变量值，线程内上下文被传递，不同线程间被隔离

Thread-0, value=0

Thread-0, value=1

Thread-0, value=2

Thread-1, value=0

Thread-1, value=1

Thread-1, value=2

1.4.3 应用场景

- 数据库连接，session管理
- 下面的基于日志平台的访问链路追踪中，会用到

一个失败的案例:

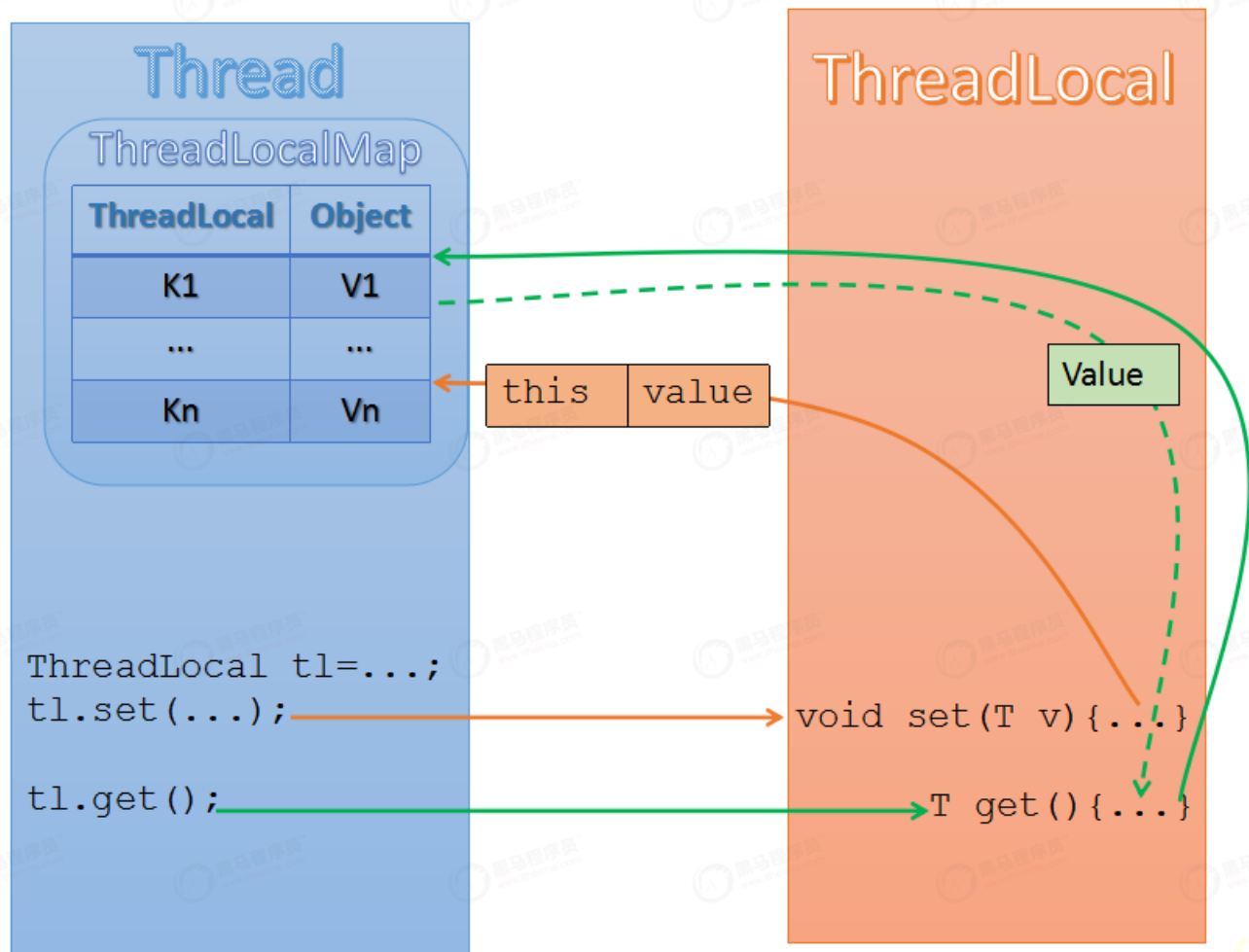
遇到过一个项目，电商商铺详情页凌晨调度生成。需要上下传递shopid，为每个商铺重新生成一下。在商铺详情页里因为是按面包屑分片生成，比如商铺信息、热卖商品、最多好评、店主推荐、最新上架等。

其他信息全部生成ok，唯独商品列表多个列表出现问题。经查，在商品部分的查询中用到了ThreadLocal，造成当前商铺id丢失。

1.4.4 实现原理

ThreadLocalMap是ThreadLocal内部类，由ThreadLocal创建，每个Thread里维护一个ThreadLocal.ThreadLocalMap类型的属性threadLocals。所有的value值其实是存储在ThreadLocalMap中。

这个存储结构的思路是反转的....



1) set方法源码

```
public void set(T value) {  
    //取到当前线程  
    Thread t = Thread.currentThread();  
    //从当前线程中拿出Map  
    ThreadLocalMap map = getMap(t);  
    if (map != null)  
        //如果非空, 说明之前创建过了  
        //以当前创建的ThreadLocal对象为key, 需要存储的值为value, 写入Map  
        //因为每个线程Thread里有自己独自的Map, 所以起到了隔离作用  
        map.set(this, value);  
    else  
        //如果没有, 那就创建  
        createMap(t, value);  
}
```

2) get方法源码

```

public T get() {
    Thread t = Thread.currentThread();
    //获取到当前线程下的Map
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        //如果非空，根据当前ThreadLocal为key，取出对应的value即可
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    //如果map是空的，往往返回一个初始值，这是一个protect方法
    //这就是为什么创建ThreadLocal的时候往往要求实现这个方法
    return setInitialValue();
}

```

3) remove方法

```

public void remove() {
    ThreadLocalMap m = getMap(Thread.currentThread());
    //很简单，获取到map后，调用remove移除掉
    if (m != null)
        m.remove(this);
}

```

4) 内存泄露问题如何解决

在上述的get方法中，Entry类继承了WeakReference，即每个Entry对象都有一个ThreadLocal的弱引用，GC对于弱引用的对象采取积极的内存回收策略，避免无人搭理时发生内存泄露。

```
ThreadLocalMap.Entry e = map.getEntry(this);
```

```

/**...*/
static class ThreadLocalMap {
    /**...*/
    static class Entry extends WeakReference<ThreadLocal<?>> {...}
}

```

验证代码：

```
ThreadLocal local = new ThreadLocal();
local.set(100);
System.out.println(local.get());
System.gc();
//不会回收，因为local被强引用
System.out.println(local.get());
local = null;
//debug，查看currentThread里面的localMaps
//注意table里的reference
Thread currentThread = Thread.currentThread();
//断点1：虽然local被赋值null，但是ThreadLocal内部依然存在引用（内存泄露风险！）
System.out.println(1);
System.gc();
//断点2：gc后，引用消失
System.out.println(2);
```

ThreadLocal对象只是作为ThreadLocalMap的一个key而存在的，现在它被回收了，那么value呢？针对这一问题，ThreadLocalMap类在每次get(), set(), remove() ThreadLocalMap中的值的时候，会自动清理key为null的value。如此一来，value也能被回收了。

用完ThreadLocal后，手动remove是一个好习惯！

1.4.5 注意！

ThreadLocal如果指向了同一个引用，会打破隔离而失效。

案例：隔离失败了！

```

package com.itheima.thread;

import java.util.HashMap;
import java.util.Map;

public class BadLocal{

    public static void main(String[] args) {
        ThreadLocal<Map> local = new ThreadLocal();
        Map map = new HashMap();
        new Thread()->{
            //在线程设置后，过段时间取name
            //猜一猜结果?
            map.put("name", "i am "+Thread.currentThread().getName());
            local.set(map);
            System.out.println(Thread.currentThread().getName()+":"+
                +local.get().get("name"));
            //do something...
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()+":"+
                +local.get().get("name"));
        }).start();

        new Thread()->{
            //在线程中赋值name
            map.put("name", "i am "+Thread.currentThread().getName());
            local.set(map);
        }).start();
    }
}

```

1.5 Fork/Join

1.5.1 概念

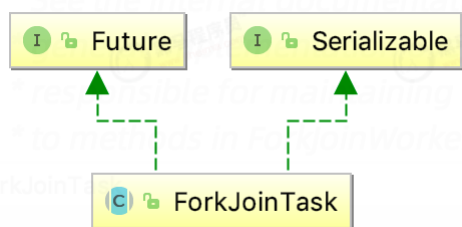
ForkJoin是由JDK1.7后提供多线程并发处理框架。ForkJoinPool由Java大师Doug Lea主持编写，处理逻辑大概分为两步。

1.任务分割：Fork（分岔），先把大的任务分割成足够小的子任务，如果子任务比较大的话还要对子任务进行继续分割。

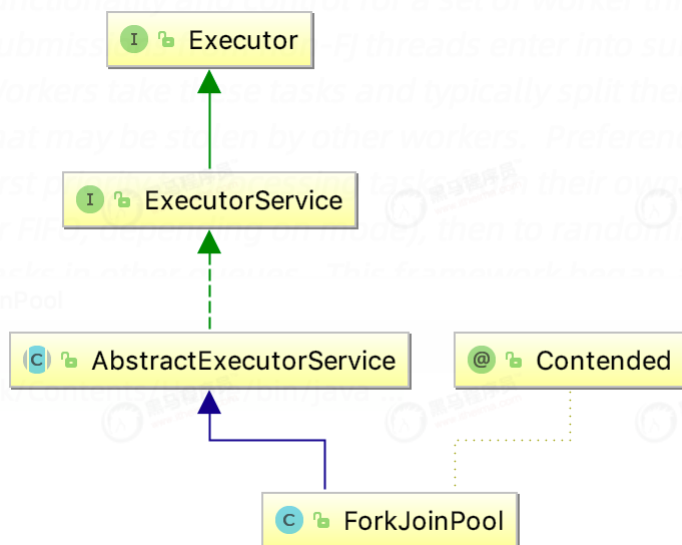
2.合并结果：join，分割后的子任务被多个线程执行后，再合并结果，得到最终的完整输出。

1.5.2 组成

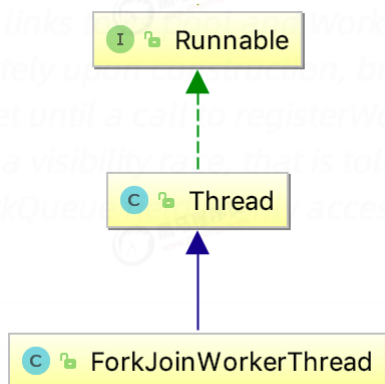
- **ForkJoinTask**: 主要提供fork和join两个方法用于任务拆分与合并；多数使用RecursiveAction（无返回值的任务）和RecursiveTask（需要返回值）来实现compute方法。



- **ForkJoinPool**: 调度ForkJoinTask的线程池;



- **ForkJoinWorkerThread**: Thread的子类，存放于线程池中的工作线程（Worker）;



- **WorkQueue**: 任务队列，用于保存任务;

1.5.3 基本使用

一个典型的例子：计算1-1000的和


```

package com.itheima.thread;

import java.util.concurrent.*;

public class SumTask {

    private static final Integer MAX = 100;

    static class SubTask extends RecursiveTask<Integer> {
        // 子任务开始计算的值
        private Integer start;

        // 子任务结束计算的值
        private Integer end;

        public SubTask(Integer start , Integer end) {
            this.start = start;
            this.end = end;
        }

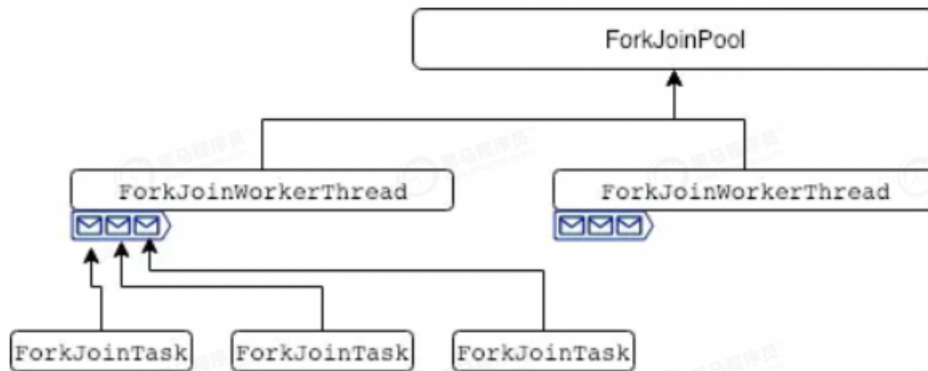
        @Override
        protected Integer compute() {
            if(end - start < MAX) {
                //小于边界，开始计算
                System.out.println("start = " + start + ";end = " + end);
                Integer totalValue = 0;
                for(int index = this.start ; index <= this.end ; index++) {
                    totalValue += index;
                }
                return totalValue;
            }else {
                //否则，中间劈开继续拆分
                SubTask subTask1 = new SubTask(start, (start + end) / 2);
                subTask1.fork();
                SubTask subTask2 = new SubTask((start + end) / 2 + 1 , end);
                subTask2.fork();
                return subTask1.join() + subTask2.join();
            }
        }
    }

    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        Future<Integer> taskFuture = pool.submit(new SubTask(1,1000));
        try {
            Integer result = taskFuture.get();
            System.out.println("result = " + result);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace(System.out);
        }
    }
}

```

1.5.4 设计思想

- 普通线程池内部有两个重要集合：工作线程集合，和任务队列。
- ForkJoinPool也类似，工作集合里放的是特殊线程ForkJoinWorkerThread，任务队列里放的是特殊任务ForkJoinTask
- 不同之处在于，普通线程池只有一个队列。而ForkJoinPool的工作线程ForkJoinWorkerThread每个线程内都绑定一个双端队列。



- 在fork的时候，也就是任务拆分，将拆分的task会被当前线程放到自己的队列中。
- 队列中的任务被线程执行时，有两种模式，默认是同步模式（`asyncMode==false`）从队尾取任务（LIFO）
- 窃取：当自己队列中执行完后，工作线程会到其他队列的队首获取任务（FIFO），取到后如果任务再次fork，拆分会被放入当前线程的队列，依次扩张

1.5.5 注意点

使用ForkJoin将相同的计算任务通过多线程执行。但是在使用中需要注意：

- 注意任务切分的粒度，也就是fork的界限。并非越小越好
- 判断要不要使用ForkJoin。任务量不是太大的话，串行可能优于并行。因为多线程会涉及到上下文的切换

1.6 volatile

1.6.1 基本概念

回顾Java 内存模型中的可见性、原子性和有序性：

- 可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的
- 原子性，指的是这个操作是原子不可拆分的，不允许别的线程中间插队操作
- 有序性指的是你写的代码的顺序要和最终执行的指令保持一致。因为在Java内存模型中，允许编译器和处理器对指令进行重排序，重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

volatile要解决的就是可见性和有序性问题。

1.6.2 使用方式

先看一个经典案例：

```

public class VolatileTest extends Thread {
    private static boolean flag = true;

    public void run() {
        while (flag);
        System.out.println("finish");
    }

    public static void main(String[] args) throws Exception {
        new VolatileTest().start();
        Thread.sleep(2000);
        flag = false;
    }
}

```

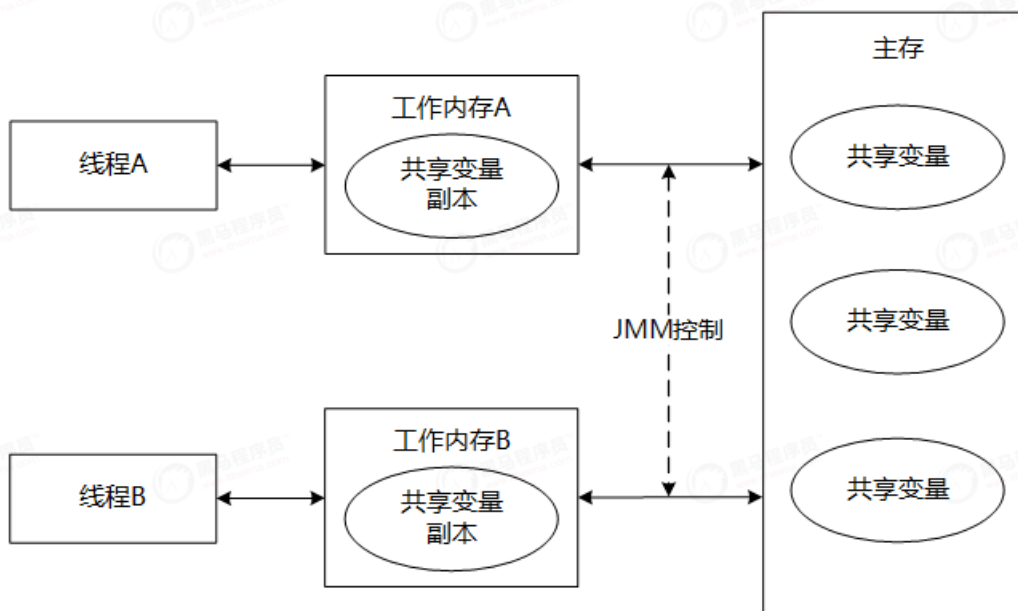
猜一猜结果？

给flag加上 volatile再试试.....

1.6.3 原理

Java内存模型分为主内存和线程工作内存两大类。

- 主内存：多个线程共享的内存。方法区和堆属于主内存区域。
- 线程工作内存：每个线程独享的内存。虚拟机栈、本地方法栈、程序计数器属于线程独享的工作内存。



Java内存模型规定，所有变量都需要存储在主内存中，线程需要时，在自己的工作内存保存变量的副本，线程对变量的所有操作都在工作内存中进行，执行结束后再同步到主内存中去。这里必然会存在时间差，在这个时间差内，该线程对副本的操作，对于其他线程是不见的，从而造成了可见性问题。

但是，当对volatile变量进行写操作的时候，JVM会向处理器发送一条lock前缀的指令，将这个缓存中的变量回写到系统主存中。

同时，在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议。每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期，一旦发现过期就会将当前处理器的缓存行设置成无效状态，强制从主内存读取，这就保障了可见性。

而volatile变量，通过内存屏障（JMM课程）可以禁止指令重排。从而实现指令的有序性。

1.6.4 注意！

volatile不能保证锁的原子性。

案例：给前面的计数器案例里加上volatile试试

```
package com.itheima;

public class BadVolatile {
    private static volatile int i=0;
    public int get(){
        return i;
    }
    public void inc(){
        int j=get();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        i=j+1;
    }

    public static void main(String[] args) throws InterruptedException {
        final BadVolatile counter = new BadVolatile();
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                public void run() {
                    counter.inc();
                }
            }).start();
        }
        Thread.sleep(3000);
        //理论上10才对。可是....
        System.out.println(counter.i);
    }
}
```

达不到目的。说明原子性无法保障。

1.7 ConcurrentHashMap

1.7.1 基本使用

很简单，new创建即可：

```
public static void main(String[] args) throws InterruptedException {  
    //定义ConcurrentHashMap  
    Map map = new ConcurrentHashMap();  
    for (int i = 0; i < 10; i++) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                //多线程下的put可以放心使用  
                map.put(UUID.randomUUID().toString(), "1");  
            }  
        }).start();  
    }  
    Thread.sleep(3000);  
    System.out.println(map);  
}
```

1.7.2 实现原理

1.7是分段锁，上面阐述过，1.8采用的是cas + synchronized 操作，具体看代码：

```

final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode()); // 计算hash
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) { // 自旋，确保插入成功
        Node<K,V> f; int n, i, fh; K fk; V fv;
        if (tab == null || (n = tab.length) == 0)
            tab = initTable(); // 表为空的话，初始化表
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 否则，插入元素，看下面的 casTabAt 方法
            // cas 在这里！ 比较是否为null，如果null才会设置并break，否则到else
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value)))
                break;
        }

        // ...

        else {
            V oldVal = null;
            // 其他情况下，加锁保持
            // synchronized 在这里！
            // 加锁，锁的是当前插槽上的头节点f（类似分段锁）
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        // 当前插槽上的节点数量
                        binCount = 1;
                        // 沿着Node链往后找
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            // 如果找到相同key，说明之前put过，覆盖
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                // put方法上，onlyIfAbsent=false
                                // 即要不要覆盖？
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            // 否则，新key，新Node插入到最后
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key, value);
                                break;
                            }
                        }
                    }
                }
            }
            // 如果是红黑树，说明已经转化过，按树的规则放入Node
            else if (f instanceof TreeBin) {
                Node<K,V> p;
                binCount = 2;
            }
        }
    }
}

```

```

        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                value)) != null) {
            oldVal = p.val;
            if (!onlyIfAbsent)
                p.val = value;
        }
    }
    else if (f instanceof ReservationNode)
        throw new IllegalStateException("Recursive update");
    }
}
if (binCount != 0) {
    //如果节点数达到临界值，链表转成树
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
}
}
//计数
addCount(1L, binCount);
return null;
}

//compareAndSetObject, 比较并插入，典型CAS操作
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
                                     Node<K,V> c, Node<K,V> v) {
    return U.compareAndSetObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}

//get取值
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    int h = spread(key.hashCode());
    //判断table是不是空的，当前桶上是不是空的
    //如果为空，返回null
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        //找到对应hash槽的第一个node，如果key相等，返回value
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
    }
    //如果正在扩容，不影响，继续顺着node找即可
    else if (eh < 0)
        return (p = e.find(h, key)) != null ? p.val : null;
    //其他情况，逐个遍历，比对key，找到后返回value
    while ((e = e.next) != null) {
        if (e.hash == h &&
            ((ek = e.key) == key || (ek != null && key.equals(ek))))

```



```
        return e.val;
    }
}
return null;
}
```

总结：

put过程：

- 1.根据key的hash值定位到桶位置
- 2.如果table为空if，先初始化table。
- 3.如果table当前桶里没有node，cas添加元素。成功则跳出循环，失败则进入下一轮for循环。
- 4.判断是否有其他线程在扩容，有则帮忙扩容，扩容完成再添加元素。
- 5.如果桶的位置不为空，遍历该桶的链表或者红黑树，若key已存在，则覆盖，不存在则将key插入到链表或红黑树的尾部。

get过程：

- 1.根据key的hash值定位到桶位置。
- 2.map是否初始化，没有初始化则返回null
- 3.定位的桶是否有头结点，没有返回null
- 4.是否有其他线程在扩容，有的话调用find方法沿node指针往后查找。扩容与find可以并行，因为node的next指针不会变
- 5.若没有其他线程在扩容，则遍历桶对应的链表或者红黑树，使用equals方法进行比较。key相同则返回value,不存在则返回null

1.7.3 注意！

注意正确理解ConcurrentHashMap线程安全这个问题。看一个典型案例：


```

package com.itheima;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class BadConcurrent {

    public static void main(String[] args) throws InterruptedException {
        Map<String,Integer> map = new ConcurrentHashMap();
        map.put("val",0);

        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                int v = map.get("val");
                v++;
                try {
                    Thread.currentThread().sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                map.put("val",v);
            }).start();
        }

        Thread.sleep(3000);
        System.out.println(map);
    }
}

```

猜一猜结果？

1.8 并发容器

除了上面提到的ConcurrentHashMap，还有很多其他的并发容器，本节统一汇总。

1.8.1 背景

java中的集合类非常丰富（ArrayList，HashMap之类），在单线程下用的顺风顺水，但这些集合类都是非线程安全的，即在多线程的环境下，都需要其他额外的手段来保证数据的正确性。常见手段有两种：

- 自己通过synchronized关键字将所有使用到非线程安全的容器代码全部同步执行
 - Vector、Stack、HashTable、Collections.synchronized等同步容器法，在早期的jdk中用的比较多，实现方式和上面几乎一样，而且多步操作时如果外面不额外加一层synchronized，依然锁不住。实际效果还不如上面
- 于是，并发容器诞生.....

1.8.2 清单

1.ConcurrentHashMap

对应：HashMap

目标：代替Hashtable、synchronizedMap，使用最多，前面详细介绍过

原理：JDK7中采用Segment分段锁，JDK8中采用CAS+synchronized

2.CopyOnWriteArrayList

对应：ArrayList

目标：代替Vector、synchronizedList

原理：高并发往往是读多写少的特性，读操作不加锁，而对写操作加Lock独享锁，先复制一份新的集合，在新的集合上面修改，然后将新集合赋值给旧的引用，并通过volatile 保证其可见性。

查看源码：volatile array，lock加锁，数组复制

3.CopyOnWriteArraySet

对应：HashSet

目标：代替synchronizedSet

原理：与CopyOnWriteArrayList实现原理类似。

4.ConcurrentSkipListMap

对应：TreeMap

目标：代替synchronizedSortedMap(TreeMap)

原理：基于Skip list（跳表）来代替平衡树，按照分层key上下链接指针来实现。

附加：跳表

5.ConcurrentSkipListSet

对应：TreeSet

目标：代替synchronizedSortedSet(TreeSet)

原理：内部基于ConcurrentSkipListMap实现，原理一致

6.ConcurrentLinkedQueue

对应：LinkedList

对应：无界线程安全队列

原理：通过队首队尾指针，以及Node类元素的next实现FIFO队列

7.BlockingQueue

对应：Queue

特点：拓展了Queue，增加了可阻塞的插入和获取等操作

原理：通过ReentrantLock实现线程安全，通过Condition实现阻塞和唤醒

实现类：

- LinkedBlockingQueue：基于链表实现的可阻塞的FIFO队列
- ArrayBlockingQueue：基于数组实现的可阻塞的FIFO队列
- PriorityBlockingQueue：按优先级排序的队列

2. 性能调优

2.1 锁优化

2.1.1 Synchronized优化

synchronized使用起来非常简单，但是需要注意的是synchronized加锁的是什么维度

对象级别：

```
public synchronized void test(){
    // TODO
}

public void test(){
    synchronized (this) {
        // TODO
    }
}
```

类级别：

```
public static synchronized void test(){
    // TODO
}

public void test(){
    synchronized (TestSynchronized.class) {
        // TODO
    }
}
```

案例：看一个加锁粒度的案例

```

package com.itheima.thread.opt;

import java.util.concurrent.atomic.AtomicLong;

public class BadSync implements Runnable{

    long start = System.currentTimeMillis();
    AtomicLong atomicLong = new AtomicLong(0);
    volatile int i=0;
    public void inc(){
        i++;
    }

    @Override
    public synchronized void run() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        inc();
        atomicLong.getAndAdd(System.currentTimeMillis() - start);
    }

    public static void main(String[] args) throws InterruptedException {
        BadSync sync = new BadSync();

        for (int i = 0; i < 5; i++) {
            new Thread(sync).start();
        }

        Thread.sleep(3000);

        System.out.println("最终计数: i="+ sync.i);
        System.out.println("最终耗时: time="+sync.atomicLong.get());
    }
}

```

- 看一下最后的结果和耗时
- 将synchronized换到inc方法上，再试试最后的结果和耗时

结论是什么？

2.1.2 Lock锁优化

看一个小需求：电商系统中记录首页被用户浏览的次数，以及最后一次操作的时间（含读或写）。

```

package com.itheima;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.locks.ReentrantLock;

public class TotalLock {
    //类创建的时间
    final long start = System.currentTimeMillis();
    //总耗时
    AtomicLong totalTime = new AtomicLong(0);
    //缓存变量
    private Map<String,Long> map = new HashMap(){put("count",0L);};
    ReentrantLock lock = new ReentrantLock();

    //查看map被写入了多少次
    public Map read(){
        lock.lock();
        try {
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        long end = System.currentTimeMillis();
        //最后操作完成的时间
        map.put("time",end);
        lock.unlock();
        System.out.println(Thread.currentThread().getName()+" ,read="+end-start);
        totalTime.addAndGet(end - start);
        return map;
    }
    //写入
    public Map write(){
        lock.lock();
        try {
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //写入计数
        map.put("count",map.get("count")+1);
        long end = System.currentTimeMillis();
        map.put("time",end);
        lock.unlock();
        System.out.println(Thread.currentThread().getName()+" ,write="+end-start);
        totalTime.addAndGet(end - start);
        return map;
    }
}

public static void main(String[] args) throws InterruptedException {

```

```

TotalLock count = new TotalLock();

//读
for (int i = 0; i < 4; i++) {
    new Thread(()->{
        count.read();
    }).start();
}
//写
for (int i = 0; i < 1; i++) {
    new Thread(()->{
        count.write();
    }).start();
}

Thread.sleep(3000);
System.out.println(count.map);
System.out.println("读写总共耗时: "+count.totalTime.get());
}
}

```

仔细看读的时间变化和执行的总时间，思考一下，从业务和技术角度有没有可优化空间？

仔细分析业务：查看次数这里其实是可以并行读取的，我们关注的业务是写入次数，也就是count，至于读取发生的时间time的写入操作，只是一个单步put，每次覆盖，不需要原子性保障，对这个加互斥锁没有必要。

改成读写锁试试.....

```

package com.itheima;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadAndWrite {
    //类创建的时间
    final long start = System.currentTimeMillis();
    //总耗时
    AtomicLong totalTime = new AtomicLong(0);
    //缓存变量，注意！因为read并发，这里换成ConcurrentHashMap
    private Map<String, Long> map = new ConcurrentHashMap(){put("count", 0L);};
    ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

    //查看map被写入了多少次
    public Map read(){
        lock.readLock().lock();
        try {
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        long end = System.currentTimeMillis();
        //最后操作完成的时间
        map.put("time", end);
        lock.readLock().unlock();
        System.out.println(Thread.currentThread().getName()+" ,read="+ (end-start));
        totalTime.addAndGet(end - start);
        return map;
    }
    //写入
    public Map write(){
        lock.writeLock().lock();
        try {
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //写入计数
        map.put("count", map.get("count")+1);
        long end = System.currentTimeMillis();
        map.put("time", end);
        lock.writeLock().unlock();
        System.out.println(Thread.currentThread().getName()+" ,write="+ (end-start));
        totalTime.addAndGet(end - start);
        return map;
    }

    public static void main(String[] args) throws InterruptedException {

```

```

ReadAndWrite rw = new ReadAndWrite();

//读
for (int i = 0; i < 4; i++) {
    new Thread()->{
        rw.read();
    }.start();
}
//写
for (int i = 0; i < 1; i++) {
    new Thread()->{
        rw.write();
    }.start();
}

Thread.sleep(3000);
System.out.println(rw.map);
System.out.println("读写总共耗时: "+rw.totalTime.get());
}
}

```

再来看读的时间变化和总执行时间。

当read远大于write时，这个差距会更明显（改成9:1试试.....）

2.1.3 CAS乐观锁优化

回顾上面的计数器，我们用synchronized实现了准确计数，本节我们看执行时间，追究性能问题。

案例一：直接加synchronized锁


```

package com.itheima;

public class NormalSync implements Runnable{
    Long start = System.currentTimeMillis();
    int i=0;

    public synchronized void run() {
        int j = i;
        //实际业务中可能会有一堆的耗时操作，这里等待100ms模拟
        try {
            //做一系列操作
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //业务结束后，增加计数
        i = j+1;
        System.out.println(Thread.currentThread().getId()+
            " ok,time="+ (System.currentTimeMillis() - start));
    }

    public static void main(String[] args) throws InterruptedException {
        NormalSync test = new NormalSync();
        new Thread(test).start();
        new Thread(test).start();
        Thread.currentThread().sleep(1000);
        System.out.println("last value="+test.i);
    }
}

```

线程二最终耗时会在200ms+，总耗时300ms，原因是悲观锁卡在了read后的耗时操作上，但是保证了最终结果是2

案例二：基于CAS思想，compare再set

```

package com.itheima;

import sun.misc.Unsafe;

import java.lang.reflect.Field;

public class CasSync implements Runnable{
    long start = System.currentTimeMillis();
    int i=0;

    public void run() {
        int j = i;
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //CAS处理, 在这里理解思想, 实际中不推荐大家使用!
        try {
            Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);
            Unsafe unsafe = (Unsafe) f.get(null);
            long offset = unsafe.objectFieldOffset(CasSync.class.getDeclaredField("i"));
            while (!unsafe.compareAndSwapInt(this, offset, j, j+1)){
                j = i;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        //实际开发中, 要用atomic包, 或者while+synchronized自旋
        // synchronized (this){
        //     //注意这里!
        //     while (j != i){
        //         j = i;
        //     }
        //     i = j+1;
        // }

        System.out.println(Thread.currentThread().getName()+
            " ok,time="+<System.currentTimeMillis() - start>);
    }

    public static void main(String[] args) throws InterruptedException {
        CasSync test = new CasSync();
        new Thread(test).start();
        new Thread(test).start();
        Thread.currentThread().sleep(1000);
        System.out.println("last value="+test.i);
    }
}

```

线程一、二均在100ms+，总耗时200ms，最终结果还是2

2.1.4 一些经验

- 减少锁的时间
不需要同步执行的代码，能不放在同步快里面执行就不要放在同步快内，可以让锁尽快释放
- 减少锁的粒度
将物理上的一个锁，拆成逻辑上的多个锁，增加并行度，从而降低锁竞争，典型如分段锁
- 锁的粒度
拆锁的粒度不能无限拆，最多可以将一个锁拆为当前cup数量相等
- 减少加减锁的次数
假如有一个循环，循环内的操作需要加锁，我们应该把锁放到循环外面，否则每次进出循环，都要加锁
- 使用读写锁
业务细分，读操作加读锁，可以并发读，写操作使用写锁，只能单线程写，参考计数器案例
- 善用volatile
volatile的控制比synchronized更轻量化，在某些变量上可以加以运用，如单例模式中

2.2 线程池参数调优

2.2.1 代码调试

- 创建线程池，无限循环添加task，debug看works和queue数量增长规律
- 等待一段时间后，查看works数量是否回落到core

```

public class ExecutorTest {
    public static void main(String[] args) {
        BlockingQueue queue = new ArrayBlockingQueue(2);
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            3,
            5,
            20,
            TimeUnit.SECONDS,
            queue,
            new RejectedExecutionHandler() {
                public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
                    System.out.println("reject");
                }
            }
        );

        for (;;) {
            executor.execute(new Runnable() {
                public void run() {
                    try {
                        Thread.currentThread().sleep(10000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }
    }
}

```

2.2.2 Executors剖析

Executors只是一个工具类，协助你创建线程池。Executors对特定场景下做了参数调优。

1) newCachedThreadPool

```

//core=0
//max=Integer
//timeout=60s
//queue=1
//也就是只要线程不够用，就一直开，不用就全部释放。线程数0-max之间弹性伸缩
//注意：任务并发太高且耗时较长时，造成cpu高消耗，同时要警惕OOM

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

```

2) newFixedThreadPool

```
//core=max=指定数量
//timeout=0
//queue=无界链表
//也就是说，线程数一直保持制定数量，不增不减，永不超时
//如果不够用，就沿着队列一直追加上去，排队等候
//注意：并发太高时，容易造成长时间等待无响应，如果任务临时变量数据过多，容易OOM

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

3) newSingleThreadExecutor

```
//core=max=1
//timeout=0
//queue=无界链表
//只有一个线程在慢吞吞的干活，可以认为是fix的特例
//适用于任务零散提交，不紧急的情况

new ThreadPoolExecutor(1, 1,
                        0L, TimeUnit.MILLISECONDS,
                        new LinkedBlockingQueue<Runnable>());
```

4) newScheduledThreadPool

```
//core=制定数
//max=Integer
//timeout=0
//queue=DelayedWorkQueue（重点！）
//用于任务调度，DelayedWorkQueue限制住了任务可被获取的时机(getTask方法)，也就实现了时间控制

super(corePoolSize, Integer.MAX_VALUE, 0, NANSECONDS,
      new DelayedWorkQueue(), threadFactory);
```

2.2.3 一些经验

1) corePoolSize

基本线程数，一旦有任务进来，在core范围内会立刻创建线程进入工作。所以这个值应该参考业务并发量在绝大多数时间内的并发情况。同时分析任务的特性。

高并发，执行时间短的，要尽可能小的线程数，如配置CPU个数+1，减少线程上下文的切换。因为它不怎么占时间，让少量线程快跑干活。

并发不高、任务执行时间长的要分开看：如果时间都花在了IO上，那就调大CPU，如配置两倍CPU个数+1。不能让CPU闲下来，线程多了并行处理更快。如果时间都花在了运算上，运算的任务还很重，本身就很占cpu，那尽量减少cpu，减少切换时间。参考第一条

如果高并发，执行时间还很长.....

2) workQueue

任务队列，用于传输和保存等待执行任务的阻塞队列。这个需要根据你的业务可接受的等待时间。是一个需要权衡时间还是空间的地方，如果你的机器cpu资源紧张，jvm内存够大，同时任务又不是那么紧迫，减少coresize，加大这里。如果你的cpu不是问题，对内存比较敏感比较害怕内存溢出，同时任务又要求快点响应。那么减少这里。

3) maximumPoolSize

线程池最大数量，这个值和队列要搭配使用，如果你采用了无界队列，那很大程度上，这个参数没有意义。同时要注意，队列盛满，同时达到max的时候，再来的任务可能会丢失（下面的handler会讲）。

如果你的任务波动较大，同时对任务波峰来的时候，实时性要求比较高。也就是来的很突然并且都是着急的。那么调小队列，加大这里。如果你的任务不那么着急，可以慢慢做，那就扔队列吧。

队列与max是一个权衡。队列空间换时间，多花内存少占cpu，轻视任务紧迫度。max舍得cpu线程开销，少占内存，给任务最快的响应。

4) keepaliveTime

线程存活保持时间，超出该时间后，线程会从max下降到core，很明显，这个决定了你养闲人所花的代价。如果你不缺cpu，同时任务来的时间没法琢磨，波峰波谷的间隔比较短。经常性的来一波。那么实当的延长销毁时间，避免频繁创建和销毁线程带来的开销。如果你的任务波峰出现后，很长一段时间不再出现，间隔比较久，那么要适当调小该值，让闲着不干活的线程尽快销毁，不要占据资源。

5) threadFactory（自定义展示实例）

线程工厂，用于创建新线程。threadFactory创建的线程也是采用new Thread()方式，threadFactory创建的线程名都具有统一的风格：pool-m-thread-n（m为线程池的编号，n为线程池内的线程编号）。如果需要自己定义线程的某些属性，如个性化的线程名，可以在这里动手。一般不需要折腾它。

6) handler

线程饱和策略，当线程池和队列都满了，再加入线程会执行此策略。默认不处理的话会抛出异常，打进日志。这个与任务处理的数据重要程度有关。如果数据是可丢弃的，那不需要额外处理。如果数据极其重要，那需要在这里采取措施防止数据丢失，如扔消息队列或者至少详细打入日志文件可追踪。

2.3 协程

2.3.1 概念

面试官：你知道协程吗？

你：知道，出差或旅游经常用。

面试官：.....

很大一部分的程序员不知道协程是啥，项目中也未用到协程。

先看概念：计算机有进程，线程和协程。前两者大家都很熟。而协程，则是基于线程之上自主开辟的异步任务。

- 线程的切换由操作系统负责调度，协程由用户自己进行调度
- 线程的默认Stack大小是1M，而协程更轻量，接近1K。因此可以在相同的内存中开启更多的协程。
- 多个协程在同一个线程上，因此不必使用锁，也减少了上下文切换。

再说结论：

- 一般需要使用第三方框架来实现
- java里相对非主流，go和python相对用的多
- 实际web开发中用的较少，还是要把主要精力放到线程上

2.3.2 使用方式

使用Kilim框架看协程与线程的编码

解压kilim.zip包，导入工程，按说明执行

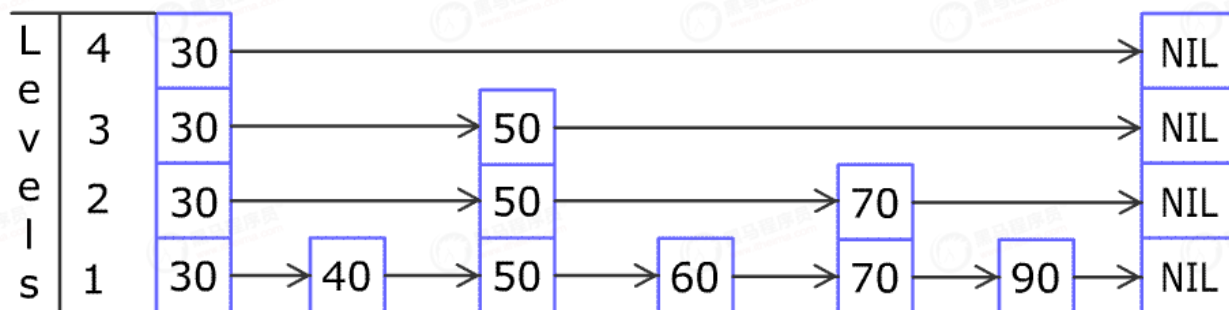
2.4 并发容器选择

1) 案例一：电商网站中记录一次活动下各个商品售卖的数量。

- 场景分析：需要频繁按商品id做get和set，但是商品id（key）的数量相对稳定不会频繁增删
- 初级方案：选用HashMap，key为商品id，value为商品购买的次数。每次下单取出次数，增加后再写入
- 问题：HashMap线程不安全！在多次商品id写入后，如果发生扩容，在JDK1.7之前，在并发场景下HashMap会出现死循环，从而导致CPU使用率居高不下。JDK1.8中修复了HashMap扩容导致的死循环问题，但在高并发场景下，依然会有数据丢失以及不准确的情况出现。
- 选型：Hashtable不推荐，锁太重，选ConcurrentHashMap确保高并发下多线程的安全性

2) 案例二：在一次活动下，为每个用户记录浏览商品的历史和次数。

- 场景分析：每个用户各自浏览的商品量级非常大，并且每次访问都要更新次数，频繁读写
- 初级方案：为确保线程安全，采用上面的思路，ConcurrentHashMap
- 问题：ConcurrentHashMap内部机制在数据量大时，会把链表转换为红黑树。而红黑树在高并发情况下，删除和插入过程中有个平衡的过程，会牵涉到大量节点，因此竞争锁资源的代价相对比较高
- 选型：用跳表，ConcurrentSkipListMap将key值分层，逐个切段，增删效率高于ConcurrentHashMap



结论：如果对数据有强一致要求，则需使用 Hashtable；在大部分场景通常都是弱一致性的情况下，使用 ConcurrentHashMap 即可；如果数据量级很高，且存在大量增删改操作，则可以考虑使用 ConcurrentSkipListMap。

3) 案例三：在活动中，创建一个用户列表，记录冻结的用户。一旦冻结，不允许再下单抢购，但是可以浏览。

- 场景分析：违规被冻结的用户不会太多，但是绝大多数非冻结用户每次抢单都要去查一下这个列表。低频写，高频读。
- 初级方案：ArrayList记录要冻结的用户id
- 问题：ArrayList对冻结用户id的插入和读取操作在高并发时，线程不安全。Vector可以做到线程安全，但并发性能差，锁太重。
- 选型：综合业务场景，选CopyOnWriteArrayList，会占空间，但是也仅仅发生在添加新冻结用户的时候。绝大多数的访问在非冻结用户的读取和比对上，不会阻塞。

2.5 上下文切换优化

2.5.1 基本操作

CPU通过时间片分配算法来循环执行任务，时间片一般是几十毫秒（ms）。切换就要保存旧状态，完成恢复时就要读取存储的内容。这个操作过程就是上下文的切换。

（现实例子：参考日常需求开发与应急bug处理开发任务被挂起的场景-_-!）

2.5.2 竞争锁

- 1) 锁的持有时间越长，就意味着有越多的线程在等待该竞争资源释放。上下文的切换代价就越多。
- 2) 将锁贴近需要加锁的地方，越近越好！在synchronized性能优化中有案例

```
public void f(){
    synchronized (this){
        f1();
        f2();
        f3();
    }
}
```

```
public void f(){
    f1();
    synchronized (this){
        f2();
    }
    f3();
}
```

- 3) 结论：类锁 < 静态锁 < 方法锁 < 代码块锁，能共享锁的地方尽量不用独享锁

2.5.3 wait/notify

- 1) 过时通知

看一个典型错误，猜一猜结果.....


```

package com.itheima;

public class WaitInvalid {
    volatile int total = 0;
    byte[] lock = new byte[0];

    //计算1-100的和，算完后通知print
    public void count(){
        synchronized (lock){
            for (int i = 1; i < 101; i++) {
                total += i;
            }
            lock.notify();
        }
        System.out.println("count finish");
    }
    //打印，等候count的通知
    public void print(){
        synchronized (lock){
            try {
                lock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(total);
    }

    public static void main(String[] args) {
        WaitInvalid waitInvalid = new WaitInvalid();

        new Thread()->{
            waitInvalid.count();
        }.start();

        new Thread()->{
            waitInvalid.print();
        }.start();
    }
}

```

将count和wait的顺序交换，再看一下结果，思考一下为什么？？

分析：

count先执行时，提前释放了notify通知，这时候，print还没进入wait，收不到这个信号。

等print去wait的时候，再等通知等不到了，典型的通知过时现象。

仅仅因为一行代码的顺序问题，如果不注意，造成整个程序卡死

2) 额外唤醒

跑一下看看结果.....

```
package com.itheima;

import java.util.ArrayList;
import java.util.List;

public class NotifyInvalid {
    List list = new ArrayList();
    byte[] lock = new byte[0];

    public void del() {
        synchronized (lock){
            //没值就等，有值就删
            if (list.isEmpty()){
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            list.remove(0);
        }
    }

    public void add(){
        synchronized (lock){
            //加个值后唤醒
            list.add(0,0);
            lock.notifyAll();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        NotifyInvalid notifyInvalid = new NotifyInvalid();

        //启动两个线程等候删除
        for (int i = 0; i < 2; i++) {
            new Thread(()->{
                notifyInvalid.del();
            }).start();
        }

        //新线程添加一个
        new Thread(()->{
            notifyInvalid.add();
        }).start();

        Thread.sleep(1000);

        System.out.println(notifyInvalid.list.size());
    }
}
```

分析：

出异常了！因为等候的两个线程第一个删除后，第二个唤醒时，等待前的状态已失效。

方案：

线程唤醒后，要警惕睡眠前后状态不一致，要二次判断

2.5.4 线程池

1) 线程池的线程数量设置不宜过大，因为一旦线程池的工作线程总数超过系统所拥有的处理器数量，就会导致过多的上下文切换。

2) 慎用Executors，尤其如newCachedThreadPool。这个方法前面分析过。如果任务过多会无休止创建过多线程，增加了上下文的切换。最好根据业务情况，自己创建线程池参数。

2.5.5 虚拟机

1) 很多JVM垃圾回收器（serial收集器、ParNew收集器）在回收旧对象时，会产生内存碎片

2) 碎片内存整理中就需要移动存活的对象。而移动内存对象就意味着这些对象所在的内存地址会发生变化

3) 内存地址变化就要去移动对象前暂停线程，在移动完成后需要再次唤醒。无形中增加了上下文的切换

4) 结论：合理搭配JVM内存调优，减少JVM垃圾回收的频率可以有效地减少上下文切换

2.5.6 协程

1) 协程不需要切换上下文，更轻量化。

2) 平时用的相对较少。用不好会出问题。

3. 电商实际应用

3.1 常见问题

3.1.1 线程协作

先搞懂线程协作的一些基本操作，面试经常要用到！

1) Object中

- wait：让出锁，阻塞等待
- notify/notifyAll：唤醒wait的进程，注意，具体唤醒哪一个要看优先级，同优先级的看运气
notifyAll优先级测试，猜一下输出？

```
package com.itheima.busi;

public class NotifyTest {
    public static void main(String[] args) {

        byte[] lock = new byte[0];

        Thread t1 = new Thread()->{
            synchronized (lock){
                try {
                    lock.wait();
                    System.out.println("t1 started");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        Thread t2 = new Thread()->{
            synchronized (lock){
                try {
                    lock.wait();
                    System.out.println("t2 started");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        Thread t3 = new Thread()->{
            synchronized (lock){
                try {
                    Thread.sleep(1000);
                    System.out.println("t3 notify");
                    lock.notifyAll();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        t1.setPriority(1);
        t2.setPriority(3);
        t3.setPriority(2);

        t1.start();
        t2.start();
        t3.start();
    }
}
```

```
}
```

结果分析: wait让出锁, t3得到执行, t3唤醒后, 虽然t1先start, 但是优先级低, 所以t2优先执行

2) Thread中

- sleep: 暂停一下, 只是让出CPU的执行权, 并不释放锁。

猜一下结果.....

```
package com.itheima;

public class SleepTest{

    public static void main(String[] args) {
        final byte[] lock = new byte[0];
        new Thread()->{
            synchronized (lock){
                System.out.println("start");
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("end");
            }
        }).start();

        new Thread()->{
            synchronized (lock){
                System.out.println("need lock");
            }
        }).start();
    }
}
```

分析:

新的thread无法异步执行, 被迫等待锁, 跟着sleep

- yield: 不释放锁, 运行中转为就绪, 让出cpu给大家去竞争。当然有可能自己又抢了回来
想一下, 以下代码有可能是什么结果.....

```

package com.itheima;

public class YieldTest{

    public static void main(String[] args) throws InterruptedException {
        final byte[] lock = new byte[0];

        //让出执行权，但是锁不释放
        Thread t1 = new Thread(()->{
            synchronized (lock){
                System.out.println("start");
                Thread.yield();
                System.out.println("end");
            }
        });

        //可以抢t1，但是拿不到锁，白费
        Thread t2 = new Thread(()->{
            synchronized (lock){
                System.out.println("need lock");
            }
        });

        //不需要锁，可以抢t1的执行权，但是能不能抢得到，不一定
        //所以多执行几次，会看到不同的结果.....
        Thread t3 = new Thread(()->{
            System.out.println("t3 started");
        });

        t1.start();
        t2.start();
        t3.start();
    }
}

```

分析：

t3会插队抢到执行权，但是t2不会，因为t2和t1共用一把锁而yield不会释放
t3不见得每次都能抢到。可能t1让出又抢了回去

- join：父线程等待子线程执行完成后再执行，将异步转为同步。注意调的是子线程，阻断的是父线程

一个典型的join案例，打开和关闭join看下结果：

```

package com.itheima;

public class JoinTest implements Runnable{

    @Override
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("I am sub");
    }

    public static void main(String[] args) throws InterruptedException {
        Thread sub = new Thread(new JoinTest());
        sub.start();
        //      sub.join();
        System.out.println("I am main");
    }
}

```

分析：

如果不join，main先跑完

如果join，main必须等待sub之后才输出

扩展：concurrent.lock中，Condition.await()，signal/signalAll 与 wait/notify效果一样

3.1.1 死锁

1) 现象

很简单，先看一个案例。双锁互相等待。


```

package com.itheima;

public class DeadLock {
    byte[] lock1 = new byte[0];
    byte[] lock2 = new byte[0];

    void f1() throws InterruptedException {
        synchronized (lock1){
            Thread.sleep(1000);
            synchronized (lock2){
                System.out.println("f1");
            }
        }
    }

    void f2() throws InterruptedException {
        synchronized (lock2){
            Thread.sleep(1000);
            synchronized (lock1){
                System.out.println("f2");
            }
        }
    }

    public static void main(String[] args) {
        DeadLock deadLock = new DeadLock();
        new Thread(()->{
            try {
                deadLock.f1();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        new Thread(()->{
            try {
                deadLock.f2();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}

```

2) 死锁的条件

- 互斥使用，即资源只能独享，一个占了其他都必须等。
- 不可抢占，资源一旦被占，就只能等待占有者主动释放，其他线程抢不走。
- 贪婪占有，占着一把锁不释放，同时又需要申请另一把。
- 循环等待，即存在等待环路， $A \rightarrow B \rightarrow C \rightarrow A$ 。

3) 排查

jdk自带工具

- jps + jstack pid

通过jps找到线程号，再执行jstack pid，找到 Found one Java-level deadlock:xxx

- jconsole

执行jconsole，打开窗口，找到 线程 → 检测死锁

- jvisualvm

执行jvisualvm，打开窗口，双击线程pid，打开线程，会提示死锁，dump查看线程信息

4) 如何避免

- 合理搭配锁顺序，如果必须获取多个锁，我们就要考虑不同线程获取锁的次序搭配
- 少用synchronized，多用Lock.tryLock方法并配置超时时间
- 对多线程保持谨慎。拿不准的场景宁可不用。线上一旦死锁往往正是高访问时间段。代价巨大

3.1.2 饥饿线程

1) 概念

如果一个线程因为 CPU 时间全部被其他线程抢走而始终得不到 CPU 运行时间，看一个案例：

读代码，猜一猜结果？

```

package com.itheima;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class HungryThread{
    ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    void write(){
        readWriteLock.writeLock().lock();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        readWriteLock.writeLock().unlock();
    }
    void read(){
        readWriteLock.readLock().lock();
        System.out.println("read");
        readWriteLock.readLock().unlock();
    }

    public static void main(String[] args) {
        HungryThread hungryThread = new HungryThread();

        Thread t1 = new Thread(()->{
            //不停去拿写锁，拿到后sleep一段时间，释放
            while (true) {
                hungryThread.write();
            }
        });

        Thread t2 = new Thread(()->{
            //不停去拿读锁，虽然是读锁，但是...看下面！
            while (true){
                hungryThread.read();
            }
        });

        t1.setPriority(9);
        //优先级低！
        t2.setPriority(1);

        t1.start();
        t2.start();
    }
}

```

结果分析：

read几乎不会出现，甚至一直都拿不到锁。处于饥饿状态

StampedLock

```

package com.itheima;

import java.util.concurrent.locks.StampedLock;

public class StampedThread {
    StampedLock lock = new StampedLock();
    void write(){
        long stamp = lock.writeLock();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        lock.unlock(stamp);
    }
    void read(){
        //乐观读
        long stamp = lock.tryOptimisticRead();
        //判断是否有写在进行，没占用的话，得到执行，打印read
        if (lock.validate(stamp)){
            System.out.println("read");
        }
    }
}

public static void main(String[] args) {
    StampedThread stampedThread = new StampedThread();

    Thread t1 = new Thread(()->{
        while (true) {
            stampedThread.write();
        }
    });

    Thread t2 = new Thread(()->{
        while (true){
            stampedThread.read();
        }
    });

    t1.setPriority(9);
    t2.setPriority(1);

    t1.start();
    t2.start();
}
}

```

结果分析：

read间隔性打出，提升了读操作的并发性

注意，StampedLock的使用有局限性！

- 对于读多写少的场景 StampedLock 性能很好，简单的应用场景基本上可以替代 ReadWriteLock

- StampedLock 在命名上并没有 Reentrant，StampedLock 是不可重入的！
- StampedLock 的悲观读锁、写锁都不支持条件变量（无法使用Condition）

案例：StampedLock是不可重入锁！

```
package com.itheima.thread;

import java.util.concurrent.locks.StampedLock;

public class StampedReentrant {
    public static void main(String[] args) {
        StampedLock lock = new StampedLock();

        long stamp1 = lock.writeLock();
        System.out.println(1);

        long stamp2 = lock.writeLock();
        System.out.println(2);
        lock.unlock(stamp2);

        lock.unlock(stamp1);
    }
}
```

2) 饥饿线程产生原因

- 高优先级线程吞噬所有的低优先级线程的 CPU 时间。
- 锁始终被别的线程抢占。

3) 解决饥饿问题的方案

- 保证资源充足
- 避免持有锁的线程长时间执行，设置一定的退出机制
- 在高风险地方使用公平锁

3.2 解决方案

3.2.1 demo准备

1) boot项目

搭建springboot web项目，集成mybatis，druid连接池，rabbitmq（抢单用），swagger

mysql，rabbitmq使用docker启动，操作参考如下

docker安装说明：

<https://www.runoob.com/docker/windows-docker-install.html>

启动：

#mysql

#注意, D:/data/mysql是机器上要挂载的数据库存放目录

#需要在自己机器上创建这个路径

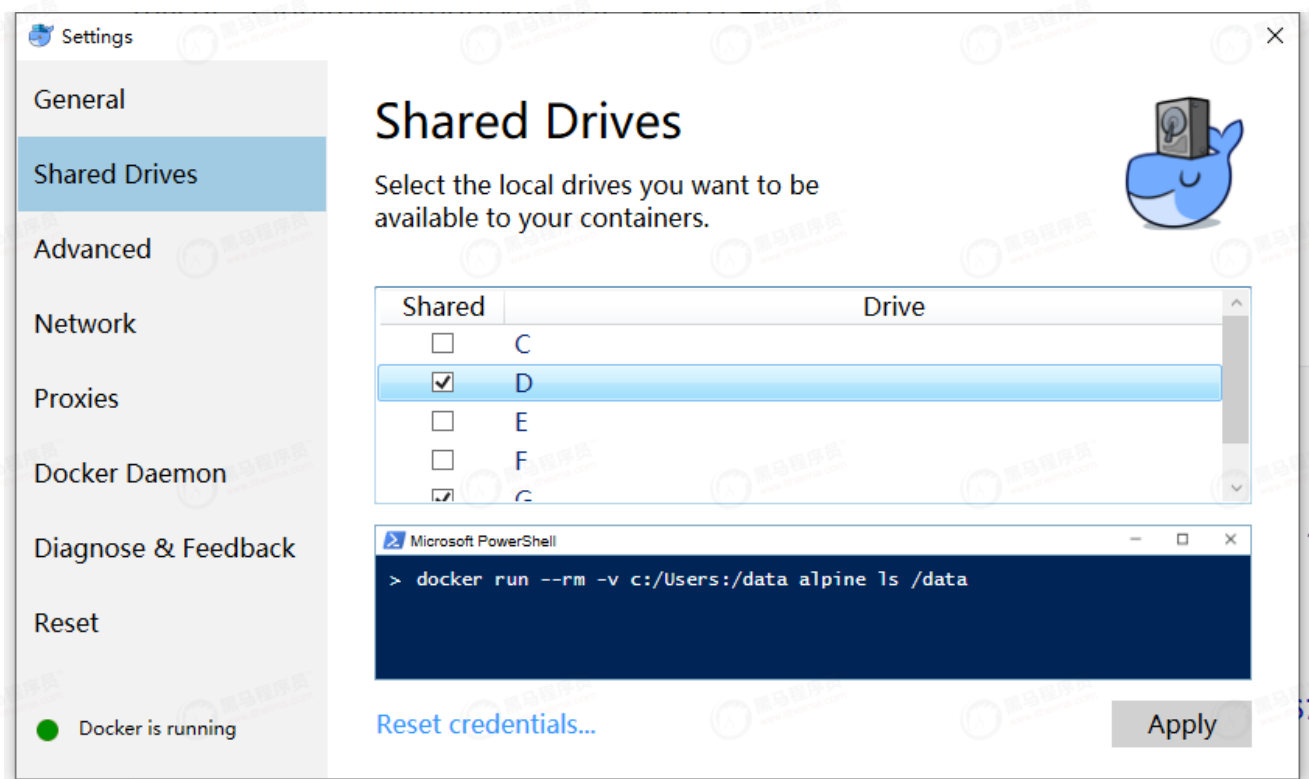
#同时要注意, windows下安装docker后, 必须选中磁盘share, 参考下面的截图

```
docker run --name mysql -v D:/data/mysql:/var/lib/mysql -p3306:3306 -e MYSQL_ROOT_PASSWORD=root  
-d daocloud.io/mysql:5.7.4
```

#rabbitmq

#这个不需要挂盘

```
docker run -d --hostname my-rabbit --name rabbit -p 15672:15672 -p5672:5672  
daocloud.io/library/rabbitmq:3.6.10-management
```



2) 建表

- orders表, 超时订单案例会用到

```
CREATE TABLE `orders` (  
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,  
  `name` varchar(255) DEFAULT NULL COMMENT '商品名称 (demo使用, 现实中为外键id)',  
  `createtime` datetime DEFAULT NULL COMMENT '创建时间',  
  `updatetime` datetime DEFAULT NULL COMMENT '更新时间',  
  `invalid` int(11) DEFAULT NULL COMMENT '失效时间 (单位秒)',  
  `status` tinyint(4) DEFAULT NULL COMMENT '状态 (0=新增, -1=失效)',  
  PRIMARY KEY (`id`)  
);
```

- product表, 库存和排序会用到

```
CREATE TABLE `product` (  
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,  
  `name` varchar(255) DEFAULT NULL COMMENT '商品名称',  
  `num` int(11) DEFAULT '0' COMMENT '库存数',  
  `price` float DEFAULT '0' COMMENT '价格',  
  PRIMARY KEY (`id`)  
);
```

- flashorder表，记录抢购后的单子

```
CREATE TABLE `flashorder` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `productid` int(11) NOT NULL COMMENT '抢到的商品id',  
  `userid` int(11) NOT NULL COMMENT '用户id，这里用抢购线程的id模拟',  
  PRIMARY KEY (`id`)  
);
```

3) 调试

启动springboot项目后，访问 <http://localhost:8080/doc.html> 进入swagger可以调试所有demo接口

3.2.2 超时订单

3.2.2.1 设计方案

1) 定时扫表：写定时任务轮询扫订单表，挨个比对时间，超时的更新掉

- 数据量小时，一般万级以内可以。几万到上亿的数据，显然不可取。
- 当前项目多处于分库分表模式，扫描需要扫多个表甚至跨库

2) 延迟消费：在下订单时，同时投放一个队列用于延迟操作，常见队列有三种

- DelayQueue，简单，不借助任何外部中间件，可借助db事务，down机丢失，同时注意内存占用
- 消息队列，设置延迟并监听死信队列，注意消息堆积产生的监控报警
- redis过期回调，redis对内存空间的占用

具体采取哪种延迟手段，根据企业实际情况，临时性的场合（比如某个抢购活动），可以采用方案一，系统化的订单取消，比如电商系统默认30分钟不支付取消规则，2号方案居多。

为加深线程相关内容，本章节采用方案一

3.2.2.2 实现

- 定义delay的对象，实现Delay接口

```

package com.itheima.thread.order;

import com.itheima.thread.mapper.Orders;

import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;

public class OrderDto implements Delayed {
    private int id;
    private long invalid;

    public OrderDto(Orders o){
        this.id = o.getId();
        this.invalid = o.getInvalid()*1000 + System.currentTimeMillis();
    }

    //倒计时，降到0时队列会吐出该任务
    @Override
    public long getDelay(TimeUnit unit) {
        return invalid - System.currentTimeMillis();
    }

    @Override
    public int compareTo(Delayed o) {
        OrderDto o1 = (OrderDto) o;
        return this.invalid - o1.invalid <= 0 ? -1 : 1;
    }

    public int getId() {
        return id;
    }

    public long getInvalid() {
        return invalid;
    }
}

```

- 定义监控类，启动守护进程，如果有超时任务，提交进线程池


```

package com.itheima.thread.order;

import com.itheima.thread.mapper.Orders;
import com.itheima.thread.mapper.OrdersMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

@Component
public class OrderMonitor {
    @Autowired
    OrdersMapper mapper ;
    //延时队列
    final DelayQueue<OrderDto> queue = new DelayQueue<OrderDto>();
    //任务池
    ExecutorService service = Executors.newFixedThreadPool(2);

    //投放延迟订单
    public void put(OrderDto dto){
        this.queue.put(dto);
        System.out.println("put task:"+dto.getId());
    }

    //在构造函数中启动守护线程
    public OrderMonitor(){
        this.execute();
        System.out.println("monitor started");
    }

    //守护线程
    public void execute(){
        new Thread()->{
            while (true){
                try {
                    OrderDto dto = queue.take();
                    System.out.println("take task:"+dto.getId());
                    service.execute(new Task(dto));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }

    //任务类
    class Task implements Runnable{
        OrderDto dto;
        Task(OrderDto dto){
            this.dto = dto;
        }
    }
}

```

```

    }
    @Override
    public void run() {
        Orders orders = new Orders();
        orders.setId(dto.getId());
        orders.setUptime(new Date());
        orders.setStatus(-1);
        System.out.println("cancel order:"+orders.getId());
        mapper.updateByPrimaryKeySelective(orders);
    }
}
}

```

- 在add订单业务中，同时扔一份到queue，注意事务性

```

@PostMapping("/add")
@Transactional
public int add(@RequestParam String name){
    Orders order = new Orders();
    order.setName(name);
    order.setCreatetime(new Date());
    order.setUptime(new Date());
    //超时时间，取10-20之间的随机数（秒）
    order.setInvalid(new Random().nextInt(10)+10);
    order.setStatus(0);
    mapper.insert(order);
    //事务性验证
    //    int i = 1/0;
    monitor.put(new OrderDto(order));
    return order.getId();
}

```

3.2.3 加/减库存

3.2.3.1 设计方案

- 1) rabbitmq异步排队：使用rabbitmq先排队，请求到来时之间入队，界面显示排队中，消费端逐个消费，同时扣减库存，界面轮询查询结果。可能会出现排队半天抢完的情况。
- 2) 库存预热：使用缓存或内存变量，活动开始前从db中提取库存值初始化，请求到来时直接扣减，及时提醒。可能出现一种感觉，活动刚开始就抢没了.....

实际企业秒杀场景下，方案1居多，为讲解多线程，本课程采用2

3.2.3.2 实现

初始化库存缓存

```

public Map load(){
    products.clear();
    List<Product> list = productMapper.selectByExample(null);
    list.forEach(p -> {
        products.put(p.getId(),new AtomicInteger(p.getNum()));
    });
    return products;
}

```

抢购代码，开启10个线程，不停去抢，减库存，如果抢到，异步刷库。

```

public void go(int productId){
    for (int i = 0; i < 10; i++) {
        new Thread()->{
            int count = 0;
            long userId = Thread.currentThread().getId();
            while (products.get(productId).getAndDecrement() > 0){
                count++;
                //扔消息队列，异步处理
                template.convertAndSend("promotion.order",productId+", "+userId);
            }
            System.out.println(Thread.currentThread().getName()+"抢到:"+count);
        }).start();
    }
}

```

注意分析控制台结果：

- 前端线程立刻抢购得到结果，给出每个线程抢到的商品数
- 后面异步处理缓慢得到结果，操作db

3.2.4 价格排序

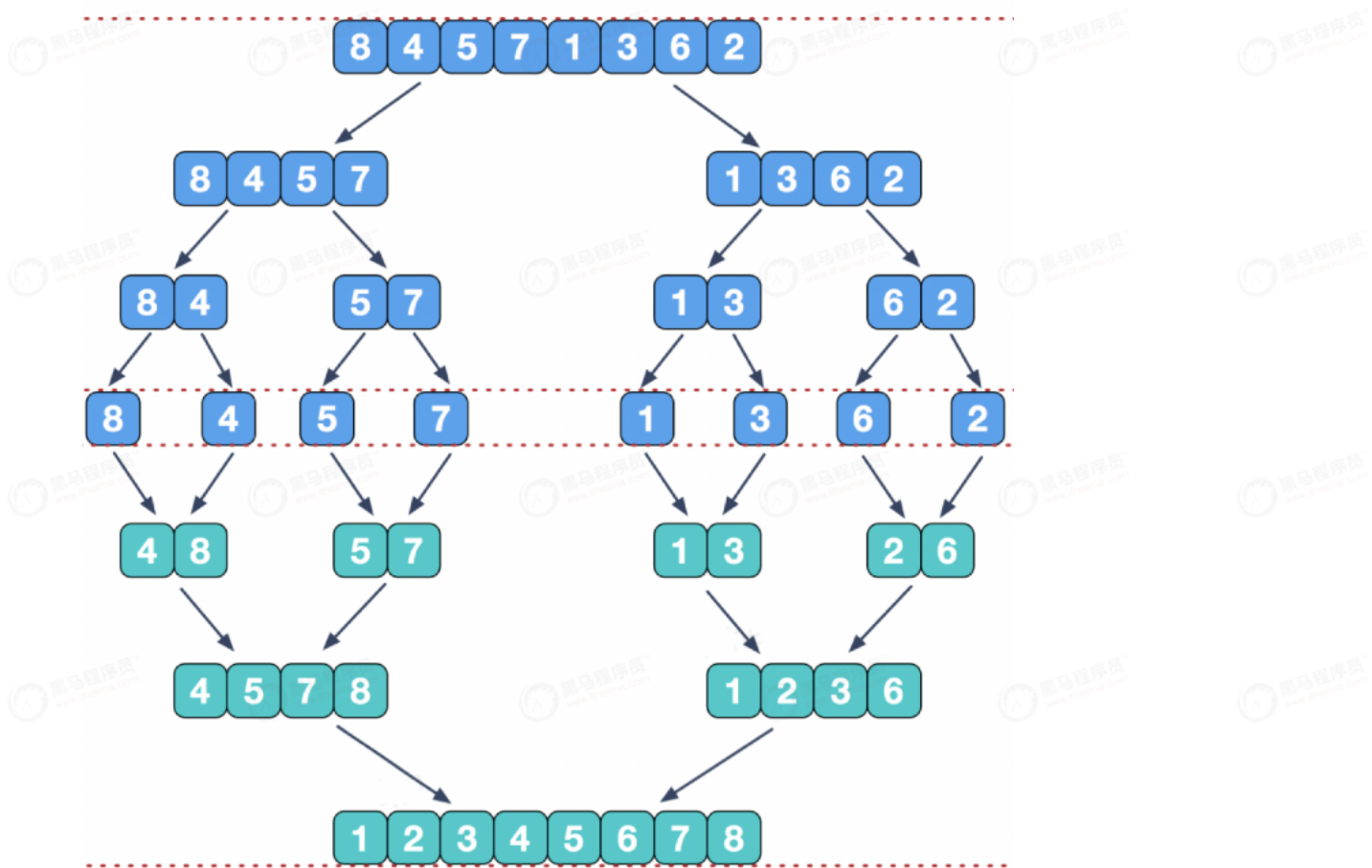
3.2.4.1 设计方案

- 1) 直接数据库sort，这种最典型
- 2) redis缓存zset获取，在商品列表缓存，web网站排序场景中常见
- 3) 内存排序，有时候，需要复杂的运算和比较逻辑，sql sort操作表达不出来时，必须进入内存运算

本课程使用方案3，规则模拟按价格排序

3.2.4.2 实现

- 1) 针对内存排序，首先想到的是实现Comparable接口，在多线程知识背景下，可以运用所学的ForkJoin实现归并排序。
- 2) 算法回顾



3) ForkJoinTask, 任务实现算法

```

package com.itheima.thread.order;

import com.itheima.thread.mapper.Product;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class SortTask extends RecursiveTask<List<Product>> {
    private List<Product> list;
    public SortTask(List<Product> list){
        this.list = list;
    }

    @Override
    //分拆与合并
    protected List<Product> compute() {
        if (list.size() > 2){
            //如果拆分的长度大于2，继续拆
            int middle = list.size() / 2 ;
            //拆成两个
            List<Product> left = list.subList(0,middle);
            List<Product> right = list.subList(middle+1,list.size());
            //子任务fork出来
            SortTask task1 = new SortTask(left);
            task1.fork();
            SortTask task2 = new SortTask(right);
            task2.fork();
            //join并返回
            return mergeList(task1.join(),task2.join());
        }else if (list.size() == 2 && list.get(0).getPrice() > list.get(1).getPrice()){
            //如果长度达到2个了，但是顺序不对，交换一下
            //其他如果2个且有序，或者1个元素的情况，不需要管他
            Product p = list.get(0);
            list.set(0,list.get(1));
            list.set(1,p);
        }
        //交换后的返回，这个list已经是每个拆分任务里的有序值了
        return list;
    }
}

```

//归并排序的合并操作，目的是将两个有序的子list合并成一个整体有序的集合
 //遍历两个子list，依次取值，两边比较，从小到大放入新list
 //注意，left和right是两个有序的list，已经从小到大排好序了

```

private List<Product> mergeList(List<Product> left,List<Product> right){
    if (left == null || right == null) return null;

    //合并后的list
    List<Product> total = new ArrayList<>(left.size()+right.size());
    //list1的下标
    int index1 = 0;
    //list2的下标
    int index2 = 0;

```

```

//逐个放入total，所以需要遍历两个size的次数之和
for (int i = 0; i < left.size()+right.size(); i++) {
    //如果list1的下标达到最大，说明list1已经都全部放入total
    if (index1 == left.size()){
        //那就从list2挨个取值，不需要比较直接放入total
        total.add(i,right.get(index2++));
        continue;
    }else if (index2 == right.size()){
        //如果list2已经全部放入，那规律一样，取list1
        total.add(i,left.get(index1++));
        continue;
    }

    //到这里说明，1和2中还都有元素，那就需要比较，把小的放入total
    //list1当前商品的价格
    Float p1 = left.get(index1).getPrice();
    //list2当前商品的价格
    Float p2 = right.get(index2).getPrice();

    Product min = null;
    //取里面价格小的，取完后，将它的下标增加
    if (p1 <= p2){
        min = left.get(index1++);
    }else{
        min = right.get(index2++);
    }
    //放入total
    total.add(min);
}
//这样处理后，total就变为两个子list的所有元素，并且从小到大排好序
System.out.println(total);
System.out.println("-----");
return total;
}
}

```

4) 调用过程

```
package com.itheima.thread.order;

import com.itheima.thread.mapper.Product;
import com.itheima.thread.mapper.ProductMapper;
import io.swagger.annotations.Api;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;

@RestController
@RequestMapping("/sort")
@Api(value = "多线程排序测试demo")
public class SortController {
    @Autowired
    ProductMapper mapper;

    @GetMapping("/list")
    List<Product> sort() throws ExecutionException, InterruptedException {
        //查商品列表
        List<Product> list = mapper.selectByExample(null);
        //线程池
        ForkJoinPool pool = new ForkJoinPool(2);
        //开始运算，拆分与合并
        Future<List<Product>> future = pool.submit(new SortTask(list));
        return future.get();
    }
}
```