

极客大学 Java 进阶训练营

第 7 课

Java 并发编程 (2)



KimmKing

Apache Dubbo/ShardingSphere PMC

Apache Dubbo/ShardingSphere PMC

前某集团高级技术总监/阿里架构师/某银行北京研发中心负责人

阿里云 MVP、腾讯 TVP、TGO 会员

10 多年研发管理和架构经验

熟悉海量并发低延迟交易系统的设计实现

目录

1. Java并发包（JUC）
2. 到底什么是锁*
3. 并发原子类*
4. 并发工具类详解*
5. 第 7 课总结回顾与作业实践

1. Java 并发包

JDK 核心库的包

java.lang.*

java.io.*

java.util.*

java.math.*

java.net.*

java.rmi.*

java.sql.*

最基础, Integer/String

IO 读写, 文件操作

工具类, 集合/日期

数学计算, BigInteger

网络编程, Socket

Java 内置的远程调用

JDBC 操作数据库

javax.*

sun.*

java 扩展 API
sun 的 JDK 实现包

JDK 公开 API
所有 JDK 都需要实现

java.util.concurrent



java.util.concurrent

锁机制类 **Locks** : Lock, Condition, ReentrantLock, ReadWriteLock, LockSupport

原子操作类 **Atomic** : AtomicInteger, AtomicLong, LongAdder

线程池相关类 **Executor** : Future, Callable, Executor, ExecutorService

信号量三组工具类 **Tools** : CountdownLatch, CyclicBarrier, Semaphore

并发集合类 **Collections** : CopyOnWriteArrayList, ConcurrentMap

2.到底什么是锁

为什么需要显式的 Lock

回忆一下，上节课讲过的，
synchronized 可以加锁，
wait/notify 可以看做加锁和解锁。
那为什么还需要一个显式的锁呢？

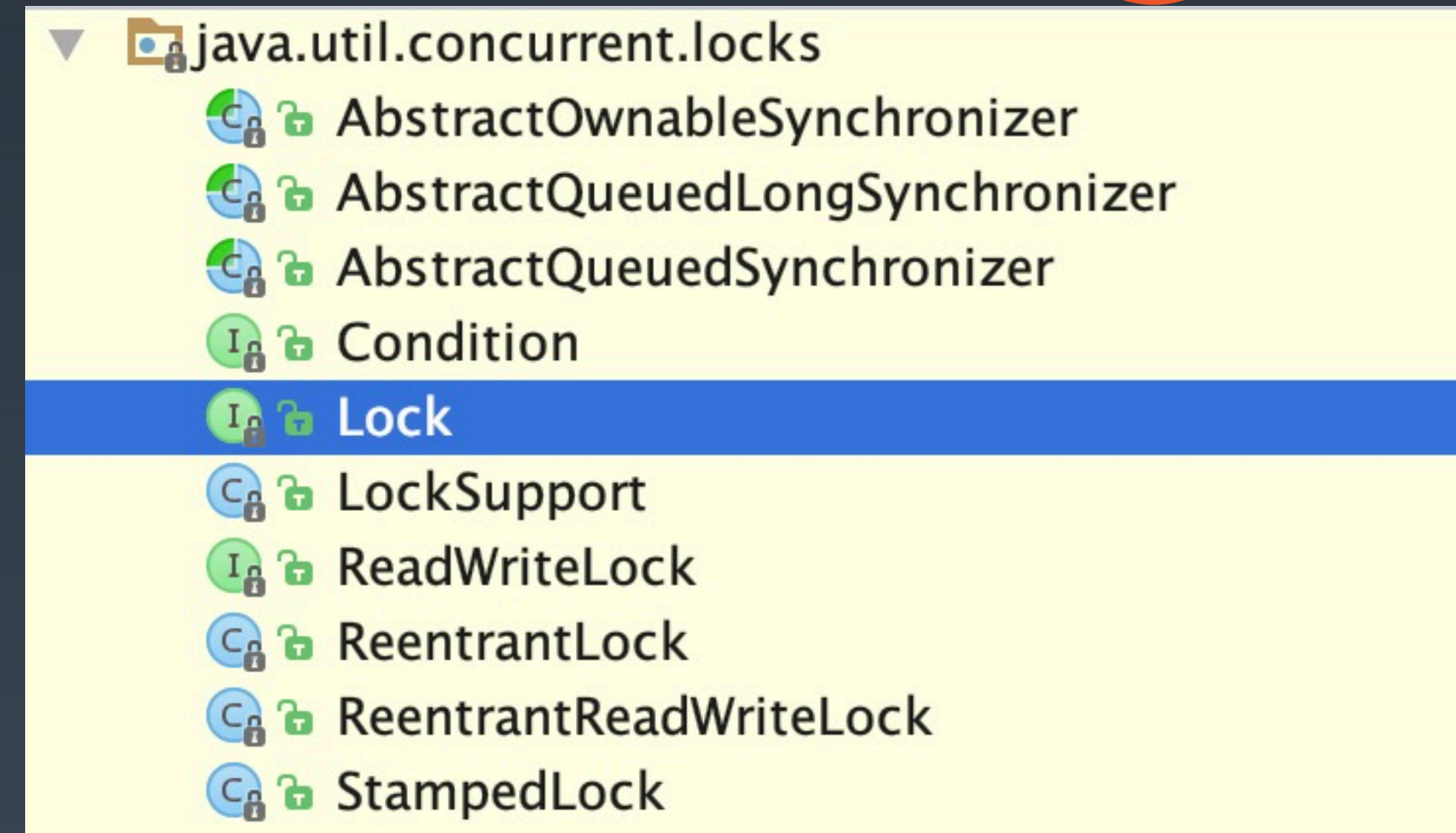
synchronized 方式的问题：

- 1、同步块的阻塞无法中断（不能 Interruptibly）
- 2、同步块的阻塞无法控制超时（无法自动解锁）
- 3、同步块无法异步处理锁（即不能立即知道是否可以拿到锁）
- 4、同步块无法根据条件灵活的加锁解锁（即只能跟同步块范围一致）

更自由的锁: Lock

1. 使用方式灵活可控
2. 性能开销小
3. 锁工具包: `java.util.concurrent.locks`

思考: Lock 的性能比 `synchronized` 高吗?



Lock 接口设计:

// 1.支持中断的 API

`void lockInterruptibly() throws InterruptedException;`

// 2.支持超时的 API

`boolean tryLock(long time, TimeUnit unit) throws InterruptedException;`

// 3.支持非阻塞获取锁的 API

`boolean tryLock();`

基础接口 - Lock

重要方法	说明
void lock();	获取锁; 类比 synchronized (lock)
void lockInterruptibly() throws InterruptedException;	获取锁; 允许打断;
boolean tryLock(long time, TimeUnit unit) throws InterruptedException;	尝试获取锁; 成功则返回 true ; 超时则退出
boolean tryLock();	尝试【无等待】获取锁; 成功则返回 true
void unlock();	解锁; 要求当前线程已获得锁; 类比同步块结束
Condition newCondition();	新增一个绑定到当前Lock的条件; 示例: (类比: Object monitor) final Lock lock = new ReentrantLock(); final Condition notFull = lock.newCondition(); final Condition notEmpty = lock.newCondition();

Lock 示例

```
public class LockCounter {  
    private int sum = 0;  
    // 可重入锁+公平锁  
    private Lock lock = new ReentrantLock(true);  
    public int addAndGet() {  
        try {  
            lock.lock();  
            return ++sum;  
        } finally {  
            lock.unlock();  
        }  
    }  
    public int getSum() {  
        return sum;  
    }  
}
```

```
// 测试代码  
public static void testLockCounter() {  
    int loopNum = 100_000;  
    LockCounter counter = new LockCounter();  
    IntStream.range(0, loopNum).parallel()  
        .forEach(i -> counter.incrAndGet());  
}
```

思考：

什么是可重入锁？

-- 第二次进入时是否阻塞。

什么是公平锁？

-- 公平锁意味着排队靠前的优先。

-- 非公平锁则是都是同样机会。

读写锁 - 接口与实现

重要方法	说明
Lock readLock();	获取读锁; 共享锁
Lock writeLock();	获取写锁; 独占锁(也排斥读锁)

```
// 构造方法
public ReentrantReadWriteLock(boolean fair) {
    sync = fair ? new FairSync() : new
    NonfairSync();
    readerLock = new ReadLock(this);
    writerLock = new WriteLock(this);
}
```

注意: ReadWriteLock 管理一组锁, 一个读锁, 一个写锁。

读锁可以在没有写锁的时候被多个线程同时持有, 写锁是独占的。

所有读写锁的实现必须确保写操作对读操作的内存影响。每次只能有一个写线程, 但是同时可以有多个线程并发地读数据。ReadWriteLock 适用于读多写少的并发情况。

```
public class ReadWriteLockCounter {
    private int sum = 0;
    // 可重入-读写锁-公平锁
    private ReadWriteLock lock = new
    ReentrantReadWriteLock(true);
    public int incrAndGet() {
        try {
            lock.writeLock().lock(); // 写锁; 独占锁; 被读锁排斥
            return ++sum;
        } finally {
            lock.writeLock().unlock();
        }
    }
    public int getSum() {
        try {
            lock.readLock().lock(); // 读锁; // 共享锁; 保证可见性
            return ++sum;
        } finally {
            lock.readLock().unlock();
        }
    }
}
```


基础接口 - Condition

重要方法	说明
void await() throws InterruptedException;	等待信号; 类比 Object#wait()
void awaitUninterruptibly();	等待信号;
boolean await(long time, TimeUnit unit) throws InterruptedException;	等待信号; 超时则返回 false
boolean awaitUntil(Date deadline) throws InterruptedException;	等待信号; 超时则返回 false
void signal();	给一个等待线程发送唤醒信号; 类比 Object#notify ()
void signalAll();	给 所有 等待线程发送唤醒信号; 类比 Object#notifyAll()

通过 Lock.newCondition() 创建。

可以看做是 Lock 对象上的信号。类似于 wait/notify。

LockSupport--锁当前线程

```
1 public static void park(Object blocker); // 暂停当前线程
2 public static void parkNanos(Object blocker, long nanos); // 暂停当前线程，不过有超时时间的限制
3 public static void parkUntil(Object blocker, long deadline); // 暂停当前线程，直到某个时间
4 public static void park(); // 无期限暂停当前线程
5 public static void parkNanos(long nanos); // 暂停当前线程，不过有超时时间的限制
6 public static void parkUntil(long deadline); // 暂停当前线程，直到某个时间
7 public static void unpark(Thread thread); // 恢复当前线程
8 public static Object getBlocker(Thread t);
```

LockSupport 类似于 Thread 类的静态方法，专门处理（执行这个代码的）本线程的。

思考：为什么 unpark 需要加一个线程作为参数？

因为一个 park 的线程，无法自己唤醒自己，所以需要其他线程来唤醒。

用锁的最佳实践

Doug Lea 《Java 并发编程：设计原则与模式》一书中，推荐的三个用锁的最佳实践，它们分别是：

1. 永远只在更新对象的成员变量时加锁
2. 永远只在访问可变的成员变量时加锁
3. 永远不在调用其他对象的方法时加锁

KK 总结-最小使用锁：

- 1、降低锁范围：锁定代码的范围/作用域
- 2、细分锁粒度：讲一个大锁，拆分成多个小锁

3.并发原子类

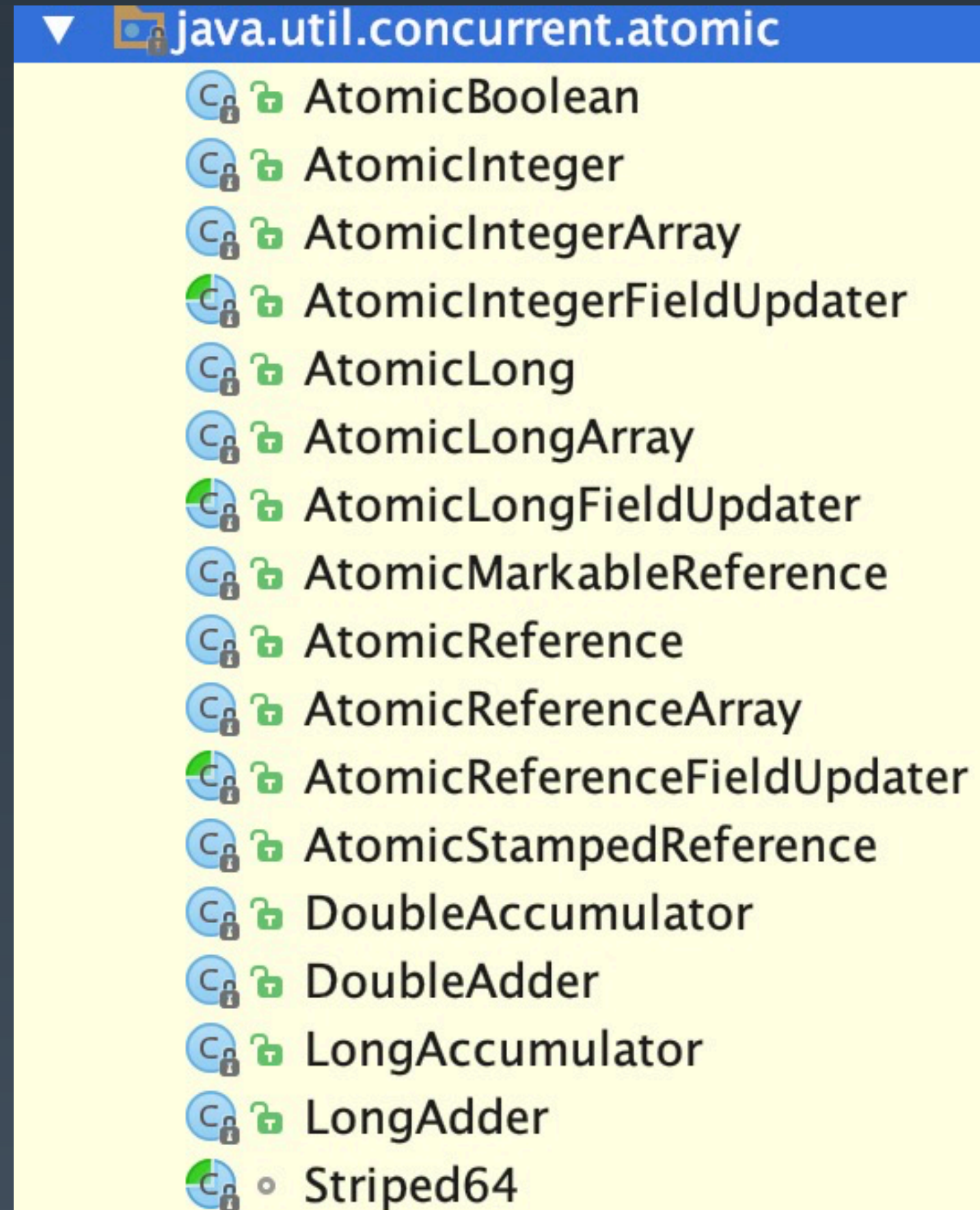
Atomic 工具类

1. 原子类工具包:

java.util.concurrent.atomic

// 使用示例

```
public class AtomicCounter {  
    private AtomicInteger sum = new AtomicInteger(0);  
    public int incrAndGet() {  
        return sum.incrementAndGet();  
    }  
    public int getSum() {  
        return sum.get();  
    }  
}
```



▼ java.util.concurrent.atomic

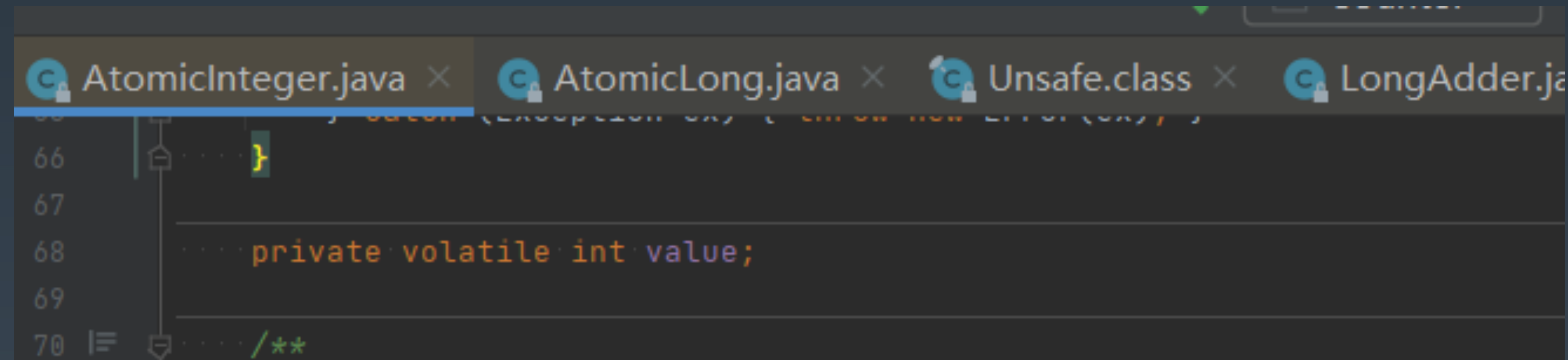
- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicIntegerFieldUpdater
- AtomicLong
- AtomicLongArray
- AtomicLongFieldUpdater
- AtomicMarkableReference
- AtomicReference
- AtomicReferenceArray
- AtomicReferenceFieldUpdater
- AtomicStampedReference
- DoubleAccumulator
- DoubleAdder
- LongAccumulator
- LongAdder
- Striped64

对比前面讲的，int sum, sum++ 线程不安全的例子。

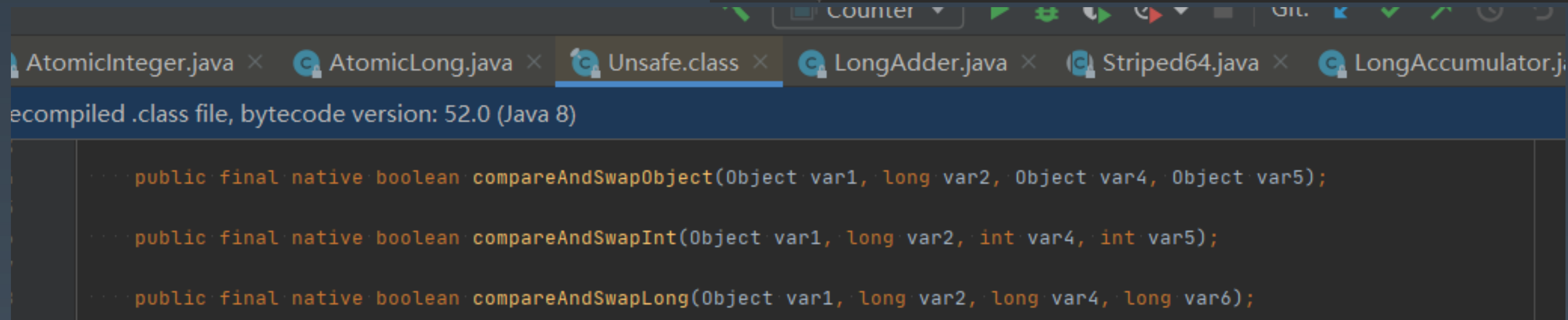
无锁技术 - Atomic 工具类

2. 无锁技术的底层实现原理

- Unsafe API - **CompareAndSwap**
- CPU 硬件指令支持 - **CAS 指令**
- Value 的可见性 - **volatile 关键字**



```
AtomicInteger.java x AtomicLong.java x Unsafe.class x LongAdder.ja
66
67
68 private volatile int value;
69
70 /**
```



```
AtomicInteger.java x AtomicLong.java x Unsafe.class x LongAdder.java x Striped64.java x LongAccumulator.ji
ecompiled .class file, bytecode version: 52.0 (Java 8)

public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object var5);

public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);

public final native boolean compareAndSwapLong(Object var1, long var2, long var4, long var6);
```

核心实现原理：

- 1、volatile 保证读写操作都可见（注意不保证原子）；
- 2、使用 CAS 指令，作为乐观锁实现，通过自旋重试保证写入。

锁与无锁之争

3. 思考一下，到底是有锁好，还是无锁好？

什么情况下有锁好

什么情况下无锁好

乐观锁、悲观锁

数据库事务锁

CAS 本质上没有使用锁。

并发压力跟锁性能的关系：

- 1、压力非常小，性能本身要求就不高；
- 2、压力一般的情况下，无锁更快，大部分都一次写入；
- 3、压力非常大时，自旋导致重试过多，资源消耗很大。

LongAdder 对 AtomicLong 的改进

通过分段思想改进原子类，
大家想想，还有哪些是用这个思想？

多路归并的思想：

- 快排
- G1 GC
- ConcurrentHashMap

LongAdder 的改进思路：

- 1、AtomicInteger 和 AtomicLong 里的 value 是所有线程竞争读写的热点数据；
- 2、将单个 value 拆分成跟线程一样多的数组 Cell[]；
- 3、每个线程写自己的 Cell[i]++，最后对数组求和。

还记得我们讲的爬山，做一个大项目，都需要加里程碑，也是分段

4.并发工具类

什么是并发工具类

思考一下：

多个线程之间怎么相互协作？

前面讲到的：

1、wait/notify

2、Lock/Condition

可以作为简单的协作机制。

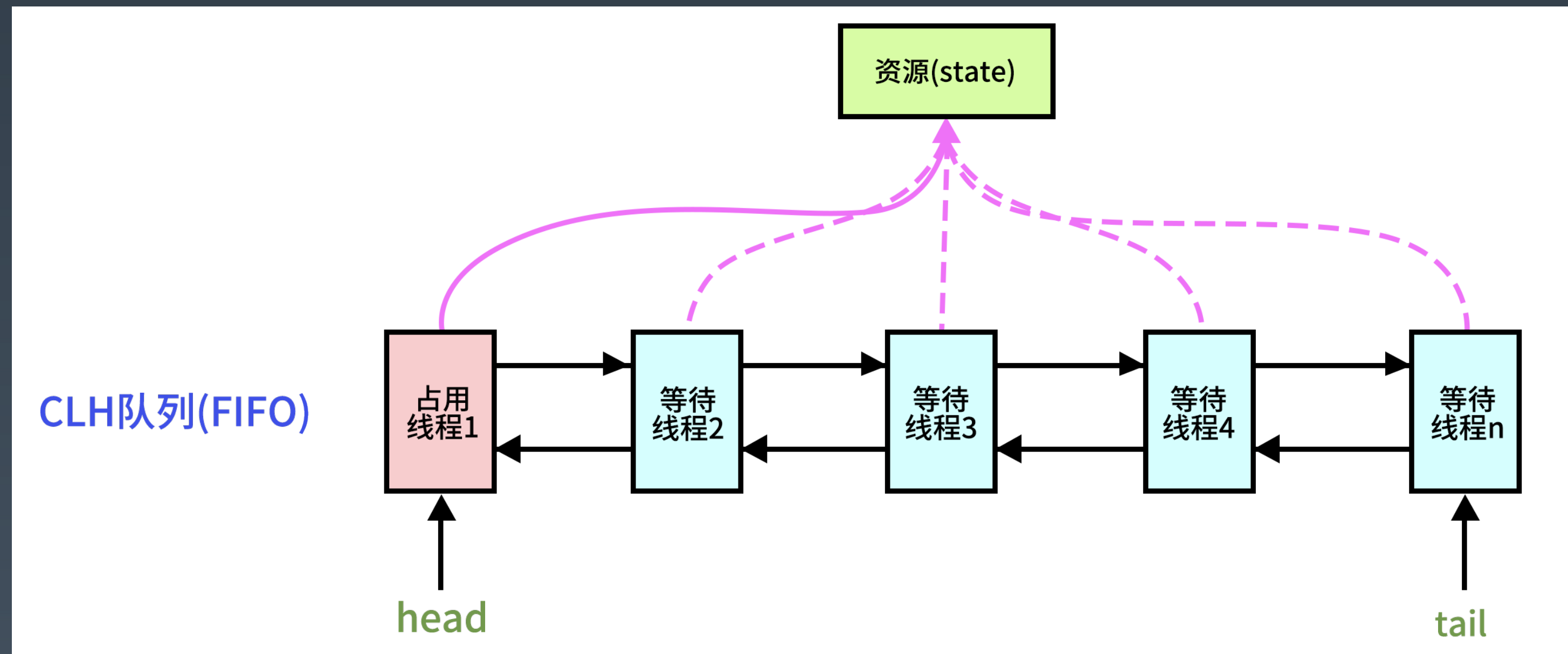
但是更复杂的，需要这些线程
满足某些条件（数量，时间）。

更复杂的应用场景，比如

- 我们需要控制实际并发访问资源的并发数量
- 我们需要多个线程在某个时间同时开始运行
- 我们需要指定数量线程到达某个状态再继续处理

AQS

- AbstractQueuedSynchronizer，即队列同步器。它是构建锁或者其他同步组件的基础（如 Semaphore、CountDownLatch、ReentrantLock、ReentrantReadWriteLock），是 JUC 并发包中的核心基础组件，抽象了竞争的资源 and 线程队列。



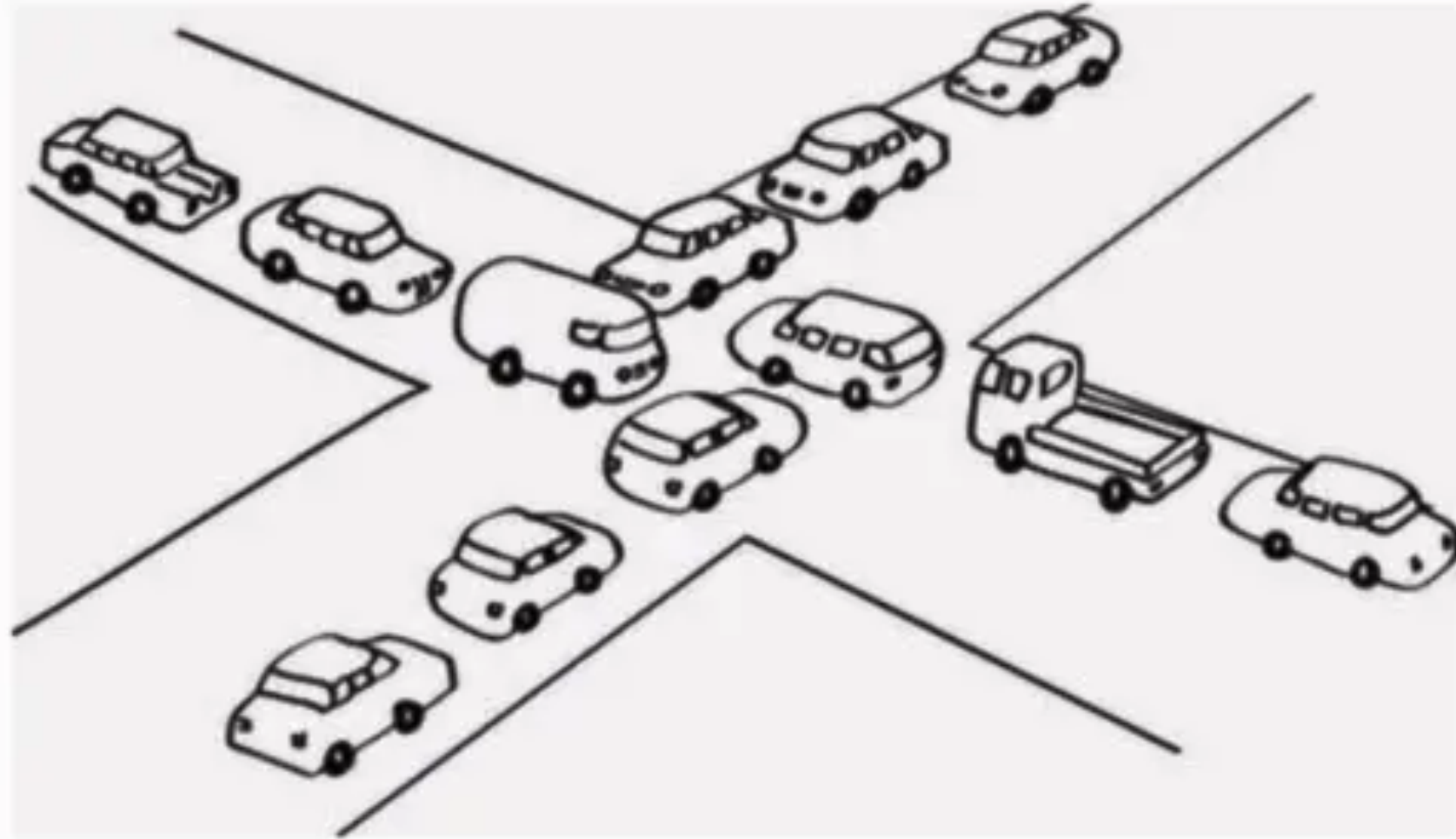
- AbstractQueuedSynchronizer: 抽象队列式的同步器
- 两种资源共享方式: 独占 | 共享，子类负责实现公平 OR 非公平

Semaphore - 信号量

1. 准入数量 N , $N=1$ 则等价于独占锁
2. 相当于 `synchronized` 的进化版

使用场景：同一时间控制并发线程数

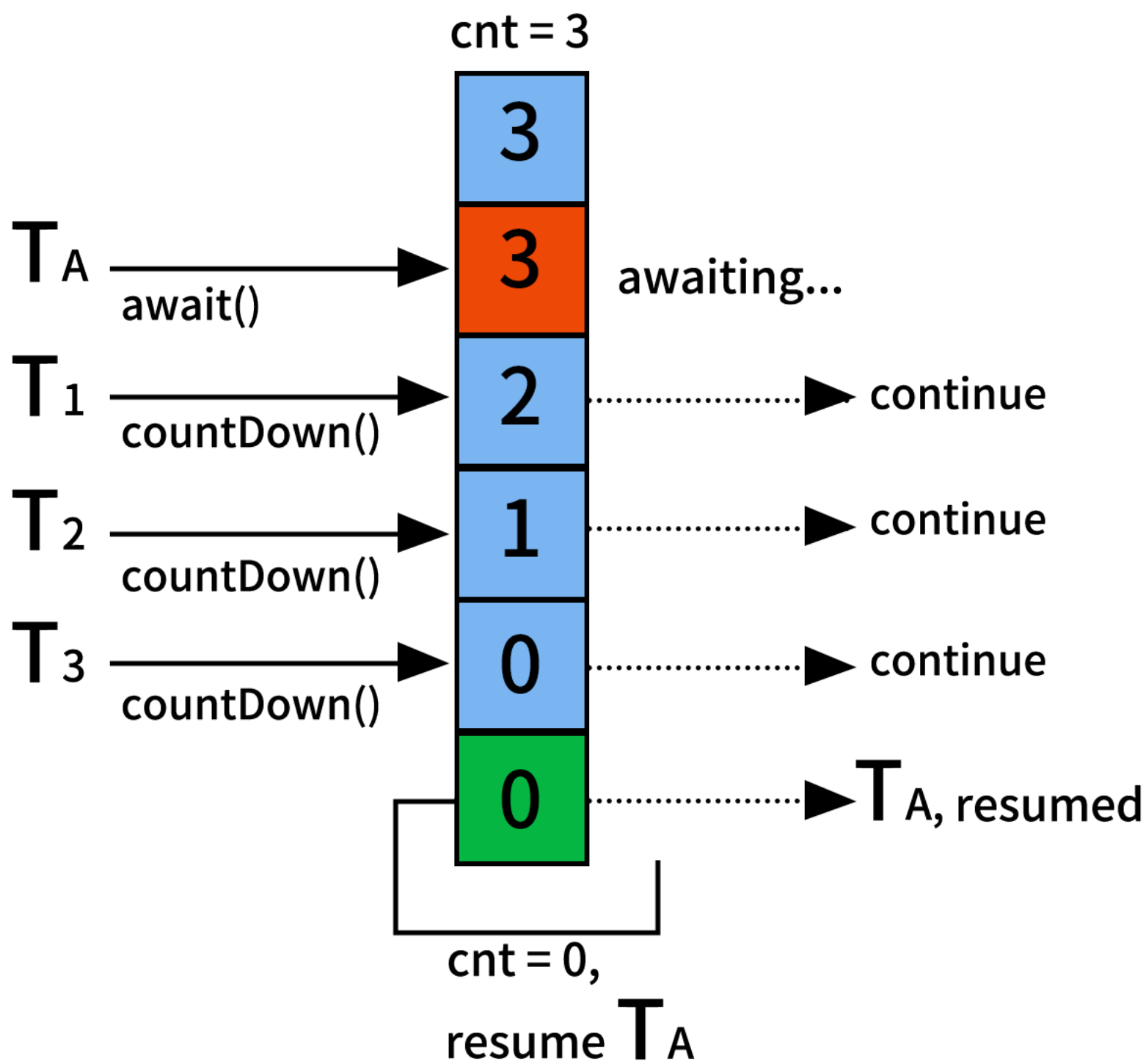
Semaphore



```
public class SemaphoreCounter {  
    private int sum = 0;  
    private Semaphore readSemaphore = new Semaphore(100, true);  
    private Semaphore writeSemaphore = new Semaphore(1);  
    public int incrAndGet() {  
        try {  
            writeSemaphore.acquireUninterruptibly();  
            return ++sum;  
        } finally {  
            writeSemaphore.release();  
        }  
    }  
    public int getSum() {  
        try {  
            readSemaphore.acquireUninterruptibly();  
            return sum;  
        } finally {  
            readSemaphore.release();  
        }  
    }  
}
```

CountDownLatch

CountDownLatch



重要方法	说明
public CountDownLatch(int count)	构造方法（总数）
void await() throws InterruptedException	等待数量归0
boolean await(long timeout, TimeUnit unit)	限时等待
void countDown()	等待数减1
long getCount()	返回剩余数量

阻塞主线程，N个子线程满足条件时主线程继续。

场景：Master 线程等待 Worker 线程把任务执行完

示例：等所有人干完手上的活，一起去吃饭。

CountDownLatch 示例

```
public static class CountdownLatchTask
implements Runnable {
    private CountdownLatch latch;
    public CountdownLatchTask(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public void run() {
        Integer millis = new Random().nextInt(10000);
        try {
            TimeUnit.MILLISECONDS.sleep(millis);
            this.latch.countDown();
            System.out.println("我的任务OK了:"+Thread.currentThread().getName());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

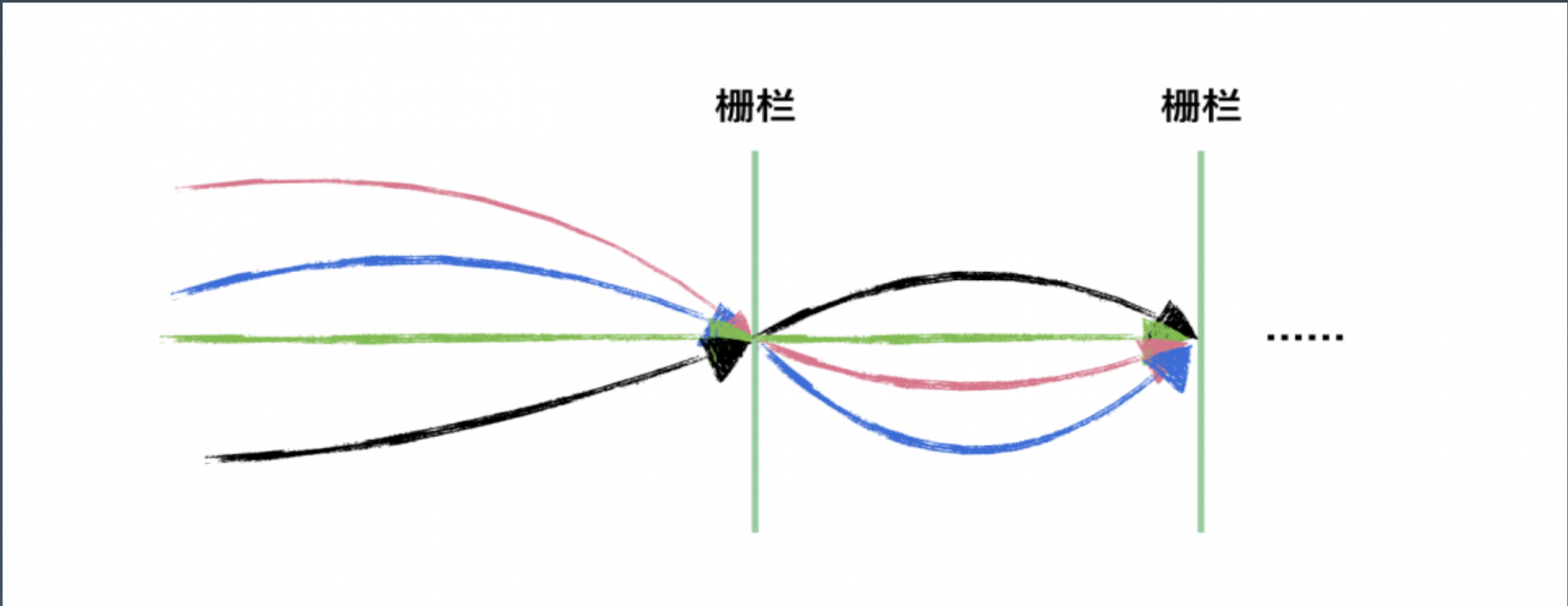
```
// 使用示例
public static void main(String[] args)
    throws Exception {
    int num = 100;
    CountdownLatch latch = new
CountDownLatch(num);
    List<CompletableFuture> list = new
ArrayList<>(num);
    for (int i = 0; i < num; i++) {
        CompletableFuture<Void> future =
            CompletableFuture.runAsync(
                new CountdownLatchTask(latch));
        list.add(future);
    }
    latch.await();
    for (CompletableFuture future : list) {
        future.get();
    }
}
```

CyclicBarrier

重要方法	说明
<code>public CyclicBarrier(int parties)</code>	构造方法（需要等待的数量）
<code>public CyclicBarrier(int parties, Runnable barrierAction)</code>	构造方法（需要等待的数量, 需要执行的任务）
<code>int await()</code>	任务内部使用; 等待大家都到齐
<code>int await(long timeout, TimeUnit unit)</code>	任务内部使用; 限时等待到齐
<code>void reset()</code>	重新一轮

场景：任务执行到一定阶段, 等待其他任务对齐, 阻塞 N 个线程时所有线程被唤醒继续。

示例：等待所有人都到达, 再一起开吃。



CyclicBarrier 示例

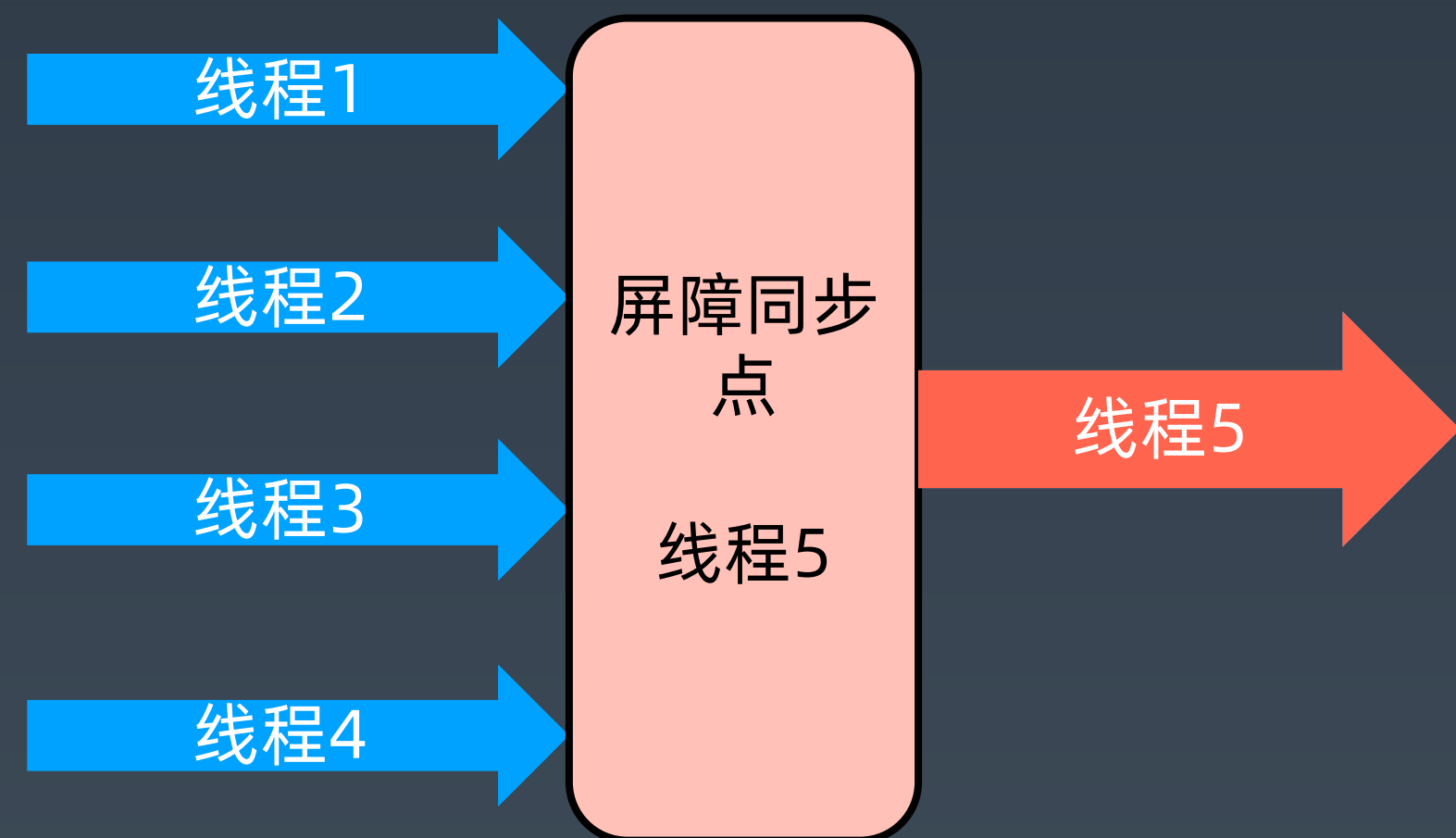
```
public static class CyclicBarrierTask
    implements Runnable {
    private CyclicBarrier barrier;
    public CyclicBarrierTask(CyclicBarrier barrier) {
        this.barrier = barrier;
    }
    @Override
    public void run() {
        Integer millis = new Random().nextInt(10000);
        try {
            TimeUnit.MILLISECONDS.sleep(millis);
            this.barrier.await(); // 线程阻塞
            System.out.println("开吃:" +
Thread.currentThread().getName());
            TimeUnit.MILLISECONDS.sleep(millis);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
// 使用
public static void main(String[] args) throws Exception {
    int num = 8;
    CyclicBarrier barrier = new CyclicBarrier(num);
    List<CompletableFuture> list = new ArrayList<>(num);
    for (int i = 0; i < num; i++) {
        CompletableFuture<Void> future =
            CompletableFuture.runAsync(
                new CyclicBarrierTask(barrier));
        list.add(future);
    }
    for (CompletableFuture future : list) {
        future.get();
    }
    barrier.reset();
}
```

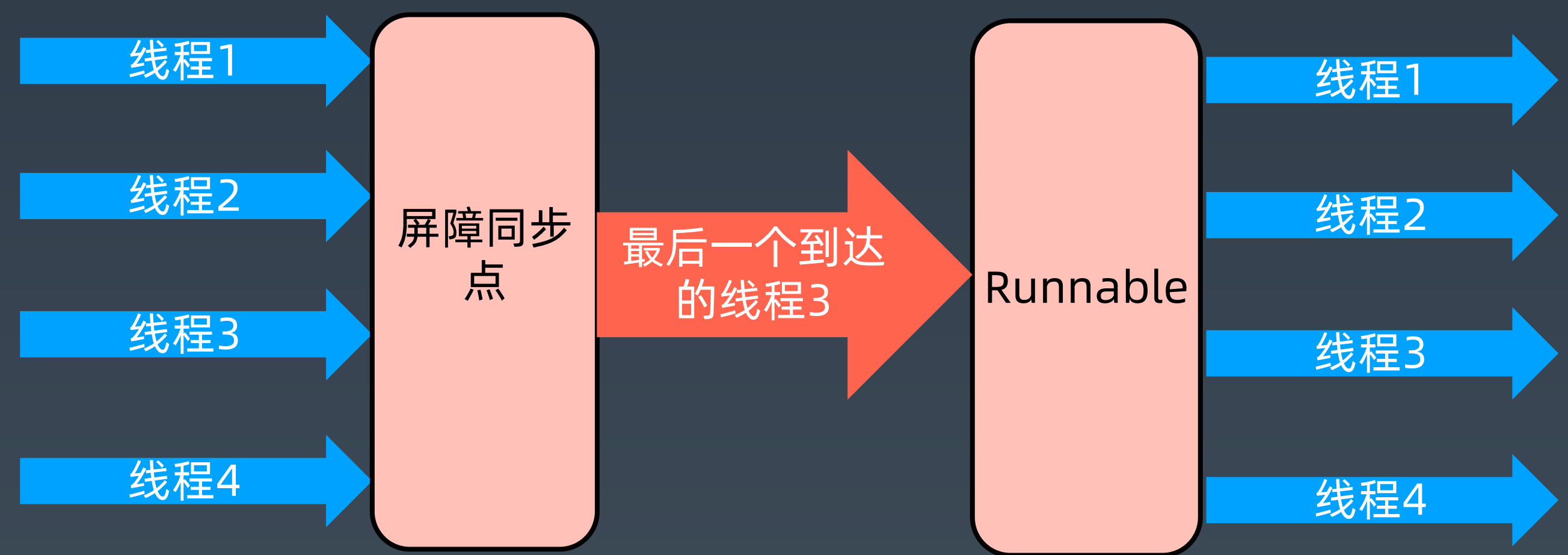
CountDownLatch与CyclicBarrier比较

CountDownLatch	CyclicBarrier
在主线程里 await 阻塞并做聚合	直接在各个子线程里 await 阻塞，回调聚合
N 个线程执行了countdown，主线程继续	N 个线程执行了 await 时，N 个线程继续
主线程里拿到同步点	回调被最后到达同步点的线程执行
基于 AQS 实现，state 为 count，递减到0	基于可重入锁 condition.await/signalAll实现
不可以复用	计数为0时重置为 N，可以复用

CountDownLatch 与 CyclicBarrier 比较

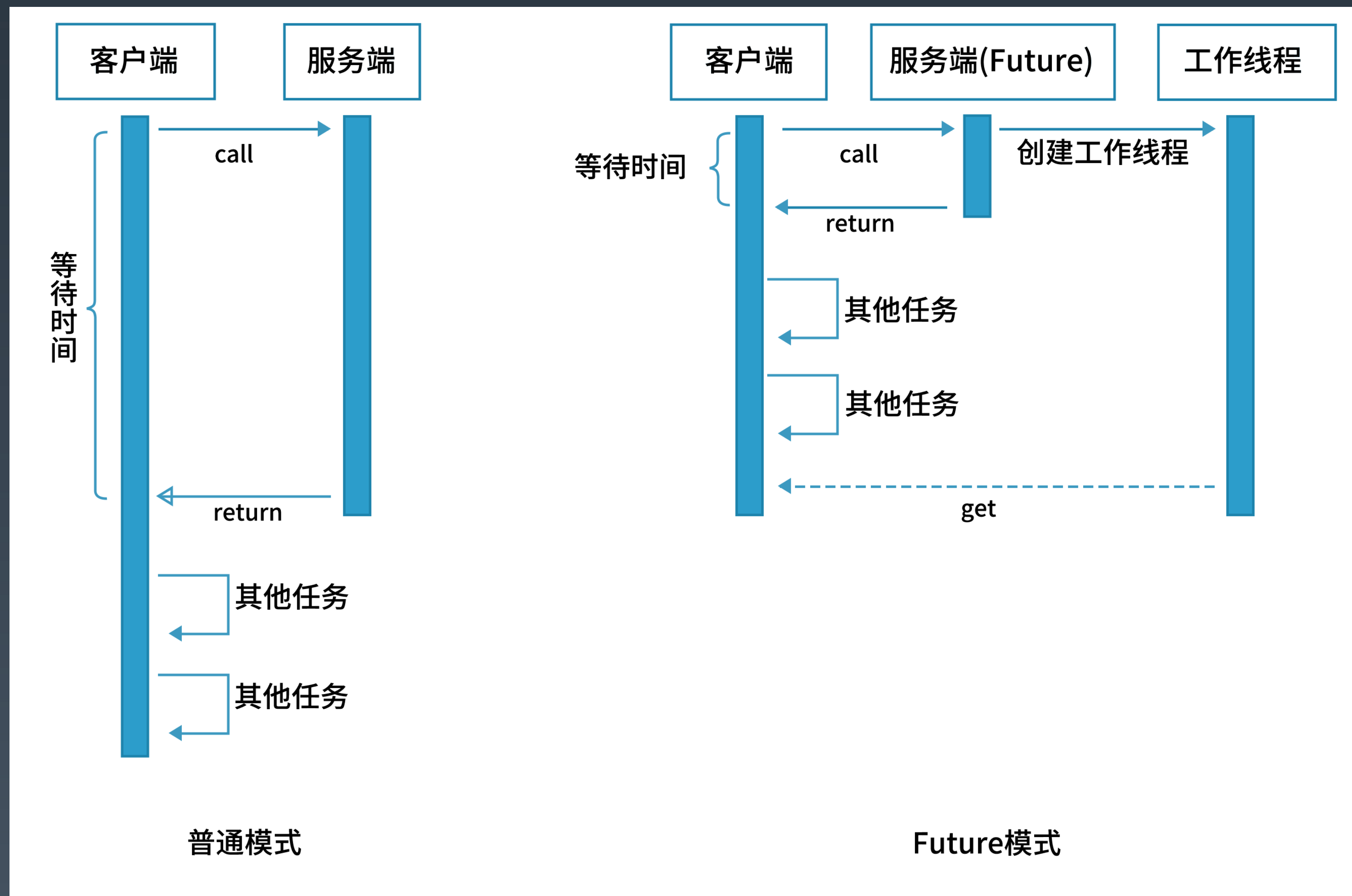


CountDownLatch



CyclicBarrier

Future/FutureTask/CompletableFuture



Future/FutureTask/CompletableFuture

Future

单个线程/任务的执行结果

FutureTask

CompletableFuture

异步，回调，组合

CompletableFuture

重要方法	说明
static final boolean <i>useCommonPool</i> = (ForkJoinPool.getCommonPoolParallelism() > 1);	是否使用内置线程池
static final Executor <i>asyncPool</i> = <i>useCommonPool</i> ? ForkJoinPool.commonPool() : new ThreadPerTaskExecutor();	线程池
CompletableFuture<Void> runAsync(Runnable runnable);	异步执行
CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)	异步执行, 使用自定义线程池
T get()	等待执行结果
T get(long timeout, TimeUnit unit)	限时等待执行结果
T getNow(T valueIfAbsent)	立即获取结果(默认值)
.....	



```
CompletableFuture<String> cf
    = CompletableFuture.supplyAsync(() -> "hello,楼下小黑哥");
//
cf.thenApply(String::toLowerCase) ;
// 需要重新创建一个 CompletionStage
cf.thenCompose(s -> CompletableFuture.supplyAsync(s::toLowerCase));
```

5.总结回顾与作业实践

第 7 节课总结回顾

Java 并发包

什么是锁

并发原子类

并发工具类

第7节课作业实践

- 1、（选做）把示例代码，运行一遍，思考课上相关的问题。也可以做一些比较。
- 2、**（必做）**思考有多少种方式，在main函数启动一个新线程，运行一个方法，拿到这个方法的返回值后，退出主线程？

写出你的方法，越多越好，提交到github。

一个简单的代码参考：

<https://github.com/kimmking/JavaCourseCodes/tree/main/03concurrency/0301/src/main/java/java0/conc0303/Homework03.java>

THANKS! |  极客大学