JUC 并发编程

Pionpill [1]

本文档为作者学习 Java 并发时的笔记。

2022 年 12 月 6 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

笔者为软件工程系在校本科生，有计算机学科理论基础 (操作系统，数据结构，计算机网络，编译原理等)，本人在撰写此笔记时已有 Java 开发经验，基础知识不再赘述。

本文需要的前置知识: 扎实的 Java 基础，对 JVM 具有一定了解。

本文各部分内容如下:

- Java 并发编程: 并发编程基础，Java 实现并发的原理。参考 «Java 并发编程之美 »[1] 。

本人的编写及开发环境如下：

- Java: Java17
- OS: Windows11

2022 年 12 月 6 日

[1]«Java 并发编程之美 »: 翟陆续 2018 年出版

目录

第一部分

Java 并发编程

I 线程与锁基础

1 Thread 线程

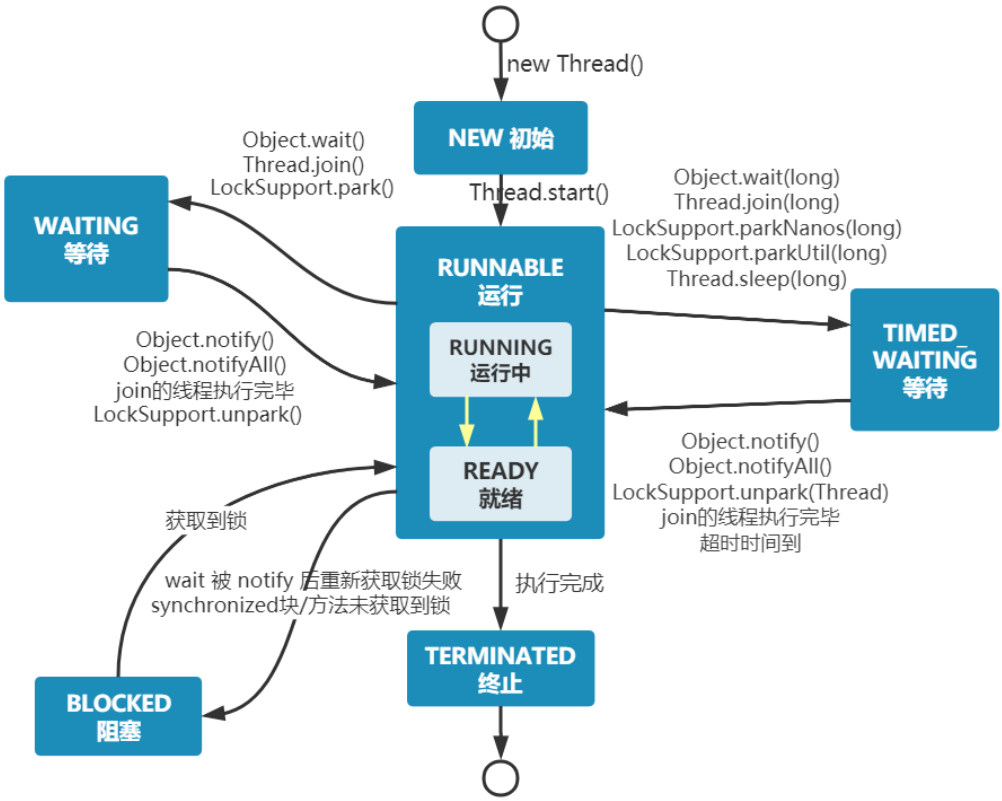
1.1 线程概念

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。

线程是进程中的一个实体，线程本身是不会独立存在的。同类的多个线程共享进程的堆和方法区资源，但每个线程有自己的程序计数器、虚拟机栈和本地方法栈，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

线程是 CPU 的基本执行单位，CPU 执行线程一般使用的是时间片轮转法，在单位时间片中，线程可能不会被执行完，程序计数器记录了线程执行的位置，用于下一次执行。每个线程都有自己的局部资源，这些资源被保存在线程私有的栈中。

一个线程的生命周期如下[1]:



理解线程周期之前，有一些必要的线程属性 (均为私有的) 需要知道:

[1]图片作者:https://mp.weixin.qq.com/s/UOrXql_LhOD8dhTq_EPI0w

- volatile String name: 线程名。
- int priority: 线程优先级。
  - public static final int MAX_PRIORITY = 10;
  - public static final int NORM_PRIORITY = 5;
  - public static final int MIN_PRIORITY = 1;
- boolean daemon = false: 是否为后台进程。
- volatile boolean interrupted: 是否处于终端状态。
- Runnable target: 运行目标。
- ThreadGroup group: 该线程所属的线程组。
- final long stackSize: 该线程所需要的栈大小。
- final long tid: 线程 id。
- volatile int threadStatus: 线程状态。
- volatile Interruptible blocker: 导致线程阻塞的对象。

此外，还有几个常用的静态方法 (都是公有的):

- native Thread currentThread(): 获取当前正在执行的线程。
- native void yield(): 使当前线程从 running 变为 ready。可以理解为让步，但也可能下一个执行的还是自己。
- native void sleep(long millis): 让当前线程休眠。
  - 有时候，我们会使用 sleep(0), 看似没用，但这可以让线程放弃 CPU，释放时间片，让其他线程执行，相当于一个让位动作。
- void onSpinWait() : 线程让步。
  - 这是一个空方法，但是被 @IntrinsicCandidate 注解，意味着在 JVM 中有高效实现，它的主要作用是代替 sleep(0), 因为性能更好。
- boolean interrupted(): 判断并重置当前线程的中断状态。
- int activeCount(): 获取当前线程组的激活状态的线程数。
- int enumerate(Thread tarray[]): 将该线程组及其子组复制到指定数组。
- void dumpStack(): 打印当前线程的执行栈。
- native boolean holdsLock(Object obj): 判断当前线程持有某个对象的锁。
- public static Map<Thread, StackTraceElement[]> getAllStackTraces(): 加强版 dumpStack()。

1.2 线程的构造方法，创建与运行

这一节说明如何创建 Thread，介绍 Thread 的 start() 方法，这章只涉及 NEW 状态与 RUNNING 状态:

```
1 public class Thread implements Runnable
```

在创建线程之前，先看一下 Thread 的 start() 方法，只有了解了 start() 方法需要什么，才

能更好的理解线程的创建与构造函数:

```
public synchronized void start() {
    // 线程状态不为0(not yet started)
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    // 加入线程组
    group.add(this);
    boolean started = false;
    try {
        // JVM 调用 run 方法
        start0();
        started = true;
    } finally {
        try {
            // 未成功 start，状态记入 group
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
            // 不处理 start0 的错误
        }
    }
}
```

可以看到，start() 方法只干了两件事:

- 调用 native 方法 start0()，核心是调用 run() 方法。
- 在线程组 group 中记录该线程的状态。

线程组 group 可以自动分配，那么 start() 的核心就是调用 run() 方法:

```
@Override
public void run() {
    if (target != null) {
        target.run();
    }
}
```

从 run() 方法源码中可以发现，它是调用的 target 的 run() 方法。那么我们想要启用一个线程就有了两总思路:

- 派生出一个 Thread 子类，重写 run() 方法。这样 start() 调用的就是 Thread 子类本身的 run() 方法。
- 传入一个 target 对象，调用 target 的 run() 方法。

第一种方法不做解释，继承后直接调用 start() 方法即可。第二种方法则需要在 Thread 的构造方法中传入一个 Runnable 对象，由于 Runnable 是函数式接口，因此也可以使用 lambda 表达式快速传入:

```
Thread thread = new Thread(() -> {
    System.out.println(Thread.currentThread().getName());
```

```
});
thread.run();
```

这两种方法各有好处，具体用哪种视情况而定。

下面看一下 Thread 的构造函数，本质上每个构造函数都调用了 private Thread 方法:

```
private Thread(ThreadGroup g, Runnable target, String name, long stackSize,
    AccessControlContext acc, boolean inheritThreadLocals)
```

在我们实际调用构造函数时，每一个参数都是可选的 (因为存在空的构造函数):

- g: 默认情况下，由父线程组管理，最顶层的线程组为 main 线程组。
- target: 指定的目标对象，调用其 run() 方法，如果没有且没有重写 start() 方法，啥都不干也不抛错。
- name: 线程名，不起的话，钩爪函数会自动帮我们起”Thread-”+nextThreadNum()。
- stackSize: 默认构造时为 0。
- acc: 如果为 null，会通过 AccessController.getContext() 获取。
- inheritThreadLocals: 默认为 true。用于获取一些初始化变量。

一般的，我们都是通过构造函数 Thread(Runnable target) 初始化一个线程，再通过 start() 方法让线程进入运行状态。

- 初始状态 (NEW): 调用构造函数，构造函数会给线程加载各种属性值。
- 运行状态 (RUNNABLE): 除了 CPU 资源，其余所有资源都具备，线程可以运行。
  - 运行中 (RUNNING): 线程获取 CPU 时间片，进入运行状态。
  - 就绪 (READY): 线程为获取时间篇，随时准备获取时间片并运行。

1.3 线程通知与等待

这章介绍 wait() 与 notify() 方法，涉及到 RUNNABLE, BLOCKED, WAITING 状态的转换。

1.3.1 wait 系列方法

先看一下 Object.wait() 方法:

```
public final void wait() throws InterruptedException
```

无参的 wait() 方法会调用被 native 修饰的其他同名方法，被 interrupt 方法调用时抛出 InterruptedException 错误并清除中断状态。

当一个线程调用一个共享变量的 wait() 方法时，该线程会进入等待 (WAITING)[2]状态，直到发生如下事件:

- 其他线程调用了该共享对象的 notify() 或 notifyAll() 方法;

[2]也也常被称为挂起状态，但挂起是进程的状态，一般认为挂机是进入 WAITING 或 TIMED_WAITING 状态。

- 其他线程调用了该线程的 interrupt() 方法，该线程抛出 InterruptedException 异常返回。

值得注意的是，即使没有使用如上操作，也没有进行其他手动操作，线程也可能被唤醒(虚假唤醒)，虽然虚假唤醒在应用实践中很少发生，但要防患于未然，做法就是不停地去测试该线程被唤醒的条件是否满足。

```
synchronized (obj) {
    while(条件) {
        obj.wait();
    }
}
```

另外需要注意的是，如果调用 wait() 方法的线程没有事先获取该对象的监视器锁，则调用 wait() 方法时调用线程会抛出 IllegalMonitorStateException 异常。获取共享变量监视器锁的方式如下[3]:

- 执行 synchronized 同步代码块时，使用该共享变量作为参数。
- 调用该共享变量的方法，并且该方法使用了 synchronized 修饰。

可以认为 synchronized 关键字代表了监视器锁。只有通过 synchronized 获取了监视器锁，才能调用 wait() 方法。

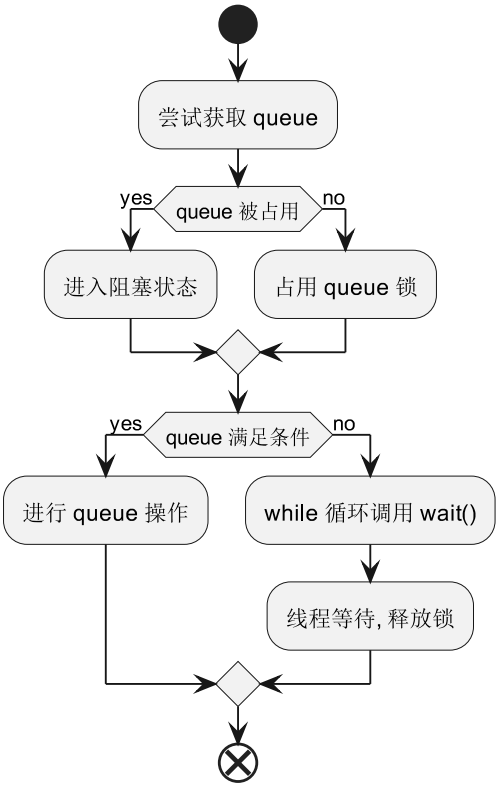
看这样一段代码:

```
// 生产者线程
synchronized (queue) {
    while(queue.size() == MAX_SIZE) {
        try {
            queue.wait();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    queue.add(1);
    queue.notifyAll();
}

// 消费者线程
synchronized (queue) {
    while(queue.size() == 0) {
        try {
            queue.wait();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    queue.poll();
    queue.notifyAll();
}
```

[3]synchronized 会在下问讲解，这里只知道就行了。

当某个线程尝试执行任意一端代码时，会执行如下流程:



当线程调用共享变量的 wait() 方法后，只会释放当前共享变量的锁。此时程序计数器指向获取锁这一步骤，之前的锁并不会被释放，否则程序计数器需要指向其他地方。

wait() 还有两个同名方法，增加了两个参数:

- **wait(long timeoutMillis)**: 这是个 native 方法，wait() 本质上时调用 wait(0)，表示不限制时间。如果一个线程调用共享对象的该方法挂起后，没有在指定的 timeoutMillis 时间内被其他线程调用该共享变量的 notify() 或者 notifyAll() 方法唤醒，那么该函数还是会因为超时而返回。
- **wait(long timeoutMillis, int nanos)**: 本质上还是调用 wait(long) 方法，当 nanos 在 (0, Long.MAX_VALUE) 区间内时，timeoutMillis 自增 1。

至此我们知道线程不进入 RUNNABLE 态有两种方式:

- 获取 synchronized 锁失败，进入阻塞态。
- 线程调用资源的 wait 系方法，进入等待态。

当线程因为 wait 系方法进入等待态后，对应的资源会维护一个等待队列，将这些线程放入到等待队列中，这个过程在虚拟机中进行。

1.3.2 notifyAll 系列方法

首先看一下 notify() 方法，当某个线程调用共享资源的 notify() 方法后，会唤醒一个在该共享资源上调用 wait 系方法后被挂起的资源。一个共享资源上可能有多个线程在等待，具体唤醒哪个线程时随机的。

被唤醒的资源并不能马上从 wait 方法返回并执行，还需要和其他线程一起竞争共享资源锁，然后才能继续执行。notify() 方法同样需要获取共享变量的监视器锁，否则会抛异常。

不同于 notify() 方法只会唤醒一个线程，notifyAll() 方法会唤醒所有在该共享变量上由于调用 wait 系方法而被挂起的线程。

需要注意的是，notify 系方法只会唤醒调用他之前因为 wait 系方法进入等待状态的线程。这在逻辑上很好理解，但在实际运行过程中，我们不知道线程的执行顺序，也就无法判断 notify 方法之前，究竟哪些线程调用了 wait 方法，而在这之后，又会有哪些线程调用 wait 方法。比较常用的方法是通过 Thread.sleep(1000) 让主线程 (也可以是其他线程) 休眠一定时间，保证其他线程执行完。

我们知道，notify 方法只会唤醒 wait 等待线程，且该线程不会直接获取锁，那么这个过程发生了什么呢:

- 同步队列: 尝试获取 Monitor(监视器锁) 失败的线程进入。
- 等待队列: 因为 wait 方法等待的线程进入的队列。

不同的 notify 方法会唤醒不同的队列元素:

- notify: 唤醒等待队列队首的一个线程。
- notifyAll: 唤醒等待队列里所有的线程。

线程被唤醒后不会直接获取锁并运行，而是进入同步队列队尾，等待获取锁资源。同时我们也知道 Java 常用的队列就有两种: FIFO 的常规队列与优先队列。队列如何实现，后文细说。

1.3.3 join 线程间等待

前文所讲的都是线程与共享资源之间的关系，线程之间也存在依赖关系，某个线程的执行需要其他线程执行完，否则需要等待所依赖的线程先执行。

实现这一逻辑的是 Thread 的 join 系方法，该方法和 Object 的 wait 系方法类似，都有三个同名方法，一个无参方法，一个实际起作用的方法。看下核心方法:

```
1  public final synchronized void join(final long millis) throws InterruptedException {
2      if (millis > 0) {
3          if (isAlive()) {
4              final long startTime = System.nanoTime();
5              long delay = millis;
6              do {
7                  wait(delay);
8              } while (isAlive() && (delay = millis -
                   TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - startTime)) > 0);
9          }
10     } else if (millis == 0) {
11         while (isAlive()) {
12             wait(0);
13         }
14     } else {
```

```
        throw new IllegalArgumentException("timeout value is negative");
    }
}
```

所以实际上还是调用了 Thread 本身的 wait 方法，即共享资源是线程本身，由于方法带了 synchronized 关键字，锁了线程对象本身，该方法无需再放入到 synchronized 语句中。

```
Thread threadA = new Thread(() -> {
    ...
    threadB.join(); // 等待 B 线程运行完才能继续运行
    ...
});
```

既然 join 系方法本质上是调用 wait 系方法，那么如何唤醒等待线程呢，很遗憾，整个 Thread 类源码里都没有 notify 系方法，那就只能是通过 jvm 唤醒了，查资料可知: 某个线程执行完被关闭之前，jvm 会唤醒所有阻塞在该线程上的其他线程。

1.4 线程睡眠与让权

Thread 有一个静态的 sleep 系方法，核心方法被 native 修饰。当某个线程调用该方法时，线程会暂时让出执行权，不参与 CPU 调度。但此时线程不会释放所拥有的监视器资源。指定时间 (必须有一个时间参数) 后，线程再次进入 READY 状态参与 CPU 调度。在睡眠时同样能被 interrupt 中断。调用该方法，线程将进入 TIMED_WAITING 状态。

Thread 有一个静态的被 native 修饰的 yield 方法，其作用是暗示线程调度器当前线程请求让出自己的 CPU 使用，但是线程调度器可以无条件忽略这个暗示。

当一个线程调用了 Thread 类的静态方法 yield 时，是在告诉线程调度器自己占有的时间片中还没有使用完的部分自己不想使用了，这暗示线程调度器现在就可以进行下一轮的线程调度。

当一个线程调用 yield 方法时，当前线程会让出 CPU 使用权，然后处于就绪状态，线程调度器会从线程就绪队列里面获取一个线程优先级最高的线程，当然也有可能会调度到刚让出的线程。

yield 与 sleep 方法的却别在于，yield 只会释放 CPU 资源，不会进入等待状态。yield 方法很少用，造轮子的时候可能会用。

1.5 线程中断

Java 中的线程中断是一种线程间的协作模式，通过设置线程的中断标志并不能直接终止该线程的执行，而是被中断的线程根据中断状态自行处理。interrupt 系共有三个方法:

- **boolean isInterrupted()**: 判断并清除线程的中断状态。

```
private volatile boolean interrupted;
```

```

public boolean isInterrupted() {
    return interrupted;
}
```

- **static boolean interrupted()**: 判断当前线程中断状态。

  如果未被中断，则直接返回中断状态 false，否则清除中断标志，仍然返回 false。

```
public static boolean interrupted() {
    Thread t = currentThread();
    boolean interrupted = t.interrupted;
    if (interrupted) {
        t.interrupted = false;
        clearInterruptEvent(); // native 方法
    }
    return interrupted;
}
```

- **void interrupt()**: 中断线程。

  如果某个线程因为 wait, join, sleep 方法而处于阻塞/等待状态，此时调用其 interrupt() 方法会抛出 InterruptedException 异常。

```
public void interrupt() {
    if (this != Thread.currentThread()) {
        checkAccess(); // 线程可能不让访问
        synchronized (blockerLock) {
            Interruptible b = blocker;
            if (b != null) {
                interrupted = true;
                interrupt0(); // native 方法，通知 jvm 中断线程
                b.interrupt(this);
                return;
            }
        }
    }
    interrupted = true;
    interrupt0();
}
```

这些方法的具体实现都依靠虚拟机。

所以 interrupt 方法有什么用呢。如果某个线程等待很长一段时间，为了让其他资源准备完成，但是其他资源可能并不需要这么长的时间，这个时候就可以使用 interrupt 打断正在 sleep 的线程，在 catch 语句中执行接下来要做的事。

```
Thread threadA = new Thread(()->{
    try {
        System.out.println("begin sleep for 200s");
        Thread.sleep(200*1000);
        System.out.println("threadA awake");
    } catch (InterruptedException e) {
        System.out.println("threadA is interrupted while sleeping");
```

```
        return;
    }
    System.out.println("threadA leaving normally");
});

threadA.start();
Thread.sleep(1000); // 确保线程A进入睡眠状态
threadA.interrupt();
threadA.join();
System.out.println("main thread over");
```

运行结果如下:

```
begin sleep for 200s
threadA is interrupted while sleeping
main thread over
```

1.6 线程知识点补充

线程死锁

线程上下文切换: 当前线程使用完时间片后，就会处于就绪状态并让出 CPU 让其他线程占用。线程上下文切换时机有:

- 当前线程的 CPU 时间片使用完转变为就绪状态
- 当前线程被其他线程中断

线程死锁: 两个或两个以上的线程在执行过程中，因争夺资源而造成的互相等待的现象，在无外力作用的情况下，这些线程会一直相互等待而无法继续运行下去。

如果已经造成了死锁，只有两种方案可以解开: 请求并持，环路等待。在实际运用中，最好的方案是所有使用共享资源的线程都按顺序竞争共享资源，这样就一定不会造成死锁。但是项目越复杂，这就越难做到。

守护线程与用户线程

Java 中的线程分为两类: daemon 线程 (守护线程) 和 user 线程 (用户线程)。默认情况下，我们创建的线程都是用户线程，这两者的区别如下:

- 守护线程: JVM 内部启动的线程，比如 GC 线程。
- 用户线程: 我们创建的线程，只要存在用户线程未执行结束，JVM 就不会退出。

我们可以通过 setDaemon(boolean) 方法设置线程类型。

另外，主线程结束子线程并不一定会结束，有时候即使 main 线程结束，其他用户线程仍在运行，jvm 就不会退出。

1.7 ThreadLocal

1.7.1 ThreadLocal 使用

多线程访问同一个共享变量时特别容易出现并发问题，特别是在多个线程需要对一个共享变量进行写入时 (读变量一般不会出问题)。为了保证线程安全，一般使用者在访问共享变量时需要进行适当的同步，同步的措施一般是加锁，但这需要使用者对锁有一定的了解。

ThreadLocal 可以做到这样的事: 当创建一个变量后，每个线程对其进行访问的时候访问的是自己线程的变量。虽然 ThreadLocal 并不是为了干这样的事而被设计出来的。

创建一个 ThreadLocal 变量后，每个线程都会复制一个变量到自己的本地内存。

```
static ThreadLocal<String> threadLocal = new ThreadLocal<>();
public static void main(String[] args) throws InterruptedException {
    Thread threadA = new Thread(() -> {
        threadLocal.set("My name is thread A");
        System.out.println(threadLocal.get());
    });
}
```

查看 ThreadLocal 源码可知，它一共有一个构造函数，三个公有实例方法，一个静态函数:

- void ThreadLocal(): 无参构造函数，什么都不做。
- void set(): 设置当前线程的变量。
- T get(): 获取当前线程的变量。
- void remove(): 移除当前线程的变量。
- static <S> ThreadLocal<S> withInitial(Supplier<? extends S> supplier): 构建 ThreadLocal 对象的静态方法。

第一次使用 ThreadLocal 会非常令人费解，它的作用是为线程维护 “私有” 变量，但是为什么这些变量全放在一个 ThreadLocal 对象里呢? 下面看一下 set() 源码:

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        map.set(this, value);
    } else {
        createMap(t, value);
    }
}
```

这里只能看出 ThreadLocal 是通过 ThreadLocalMap(一个定制化 HashMap) 维护线程 “私有” 变量，这个 map 在哪，还需要继续看 getMap 源码:

```
// ThreadLocal
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

```
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
public void remove() {
    ThreadLocalMap m = getMap(Thread.currentThread());
    if (m != null) {
        m.remove(this);
    }
}
// Thread
ThreadLocal.ThreadLocalMap threadLocals = null;
```

这下明了了，Thread 对象本身会持有一个 ThreadLocal.ThreadLocalMap 对象，用于保存线程 “私有” 变量，而这个线程私有变量具体的存取操作则由 ThreadLocal 对象完成，Thread 本身没有对 ThreadLocalMap 的功能性操作，也即在设计本地变量时将对本地变量的操作抽离到了 ThreadLocal 对象上，ThreadLocal 本质是一个工具类。

另外，值得注意的是，一个 threadLocal 只能存储一个本地变量。threadLocal 会提供一个哈希值和本地变量一起作为键值对被存在 Thread 的 ThreadLocalMap 中。

ThreadLocal 中还有一个静态方法，用于快速构建本地变量 (不需要再 set):

```
public static <S> ThreadLocal<S> withInitial(Supplier<? extends S> supplier) {
    return new SuppliedThreadLocal<>(supplier);
}

static final class SuppliedThreadLocal<T> extends ThreadLocal<T> {
    private final Supplier<? extends T> supplier;
    SuppliedThreadLocal(Supplier<? extends T> supplier) {
        this.supplier = Objects.requireNonNull(supplier);
    }
    @Override
    protected T initialValue() {
        return supplier.get();
    }
}

// Supplier
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

SuppliedThreadLocal 继承自 ThreadLocal，本质上还是操作 Thread.threadLocal 中的数据。由于 Supplier 是一个函数式接口，因此用 withInitial 方法给线程装入本地变量十分方便:

```
ThreadLocal<Integer> balance = ThreadLocal.withInitial(() -> 1000);
```

1.7.2 InheritableThreadLocal

ThreadLocal 本身只会将本地变量加入到对应线程中，如果要让子线程访问父线程的本地变量，则需要 InheritableThreadLocal 类:

```
public class InheritableThreadLocal<T> extends ThreadLocal<T>
```

和 ThreadLocal 一样，InheritableThreadLocal 也负责操作一个 Thread 成员:

```
// Thread
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

InheritableThreadLocal 没有提供任何新的公有方法，重写了几个方法用于将操作对象改为 inheritableThreadLocals。此外重写了一个 childValue 方法:

```
protected T childValue(T parentValue) {
    return parentValue;
}
```

既然 InheritableThreadLocal 没有提供什么新操作，那么可以继承的原理就应该在 Thread 中:

```
private Thread(ThreadGroup g, Runnable target, String name, long stackSize,
     AccessControlContext acc, boolean inheritThreadLocals) {
    ......
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
           this.inheritableThreadLocals =
               ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
    ......
}

static ThreadLocalMap createInheritedMap(ThreadLocalMap parentMap) {
    return new ThreadLocalMap(parentMap);
}
```

这就很明了了，线程在创建的时候通过 inheritableThreadLocals 判断 (多数情况为 true) 是否要将父线程的本地变量复制到子线程中。

2 锁，可见性与原子操作

2.1 并发概念

并发, 并行与多线程安全

首先区分一下并发与并行的关系:

- **并发**: 同一时间段内多个任务同时执行，并不是同一时间点同时进行。
- **并行**: 单位时间内多个任务同时执行。

并行与并发的概念源于单 CPU 时代，再单个 CPU(不考虑超线程技术) 上处理多个线程，多个线程之间轮流获取 CPU 资源，此时可以说并发执行。而在多 CPU 时代，多个 CPU 同一时间点处理多个线程，这可以说是并行处理。再实际编程中，线程数是多余 CPU 个数，所以一般称为多线程并发编程。

并发编程会带来线程安全问题，其本质是对共享线程的非原子性操作。例如我们有这样一个操作 (例子，实际情况并不一定如此): 让某个对象自增，这个操作在 CPU 中会被分成三个步骤:

- 1. 从内存获取对象，
- 2. 在 CPU 中让对象自增，
- 3. 将对象写回内存。

多线程下会发生什么事情呢，假设有两个线程 A,B 都需要执行以上三个步骤，执行顺序如下: A1 → A2 → B1 → A3 → B2 → B3。由于 B 线程取对象时，A 线程尚未将对象写回内存，因此最后该对象在内存中只被自增了一次 (虽然 CPU 执行了两次自增)。

大部分未经处理的的 Java 操作都是非源自性的，count++ 就不是源自操作。为了解决这个问题，Java 提供了各种数据同步方案，会在下文一一介绍。

Java 内存模型

接下来了解一下 Java 内存模型，Java 内存模型规定，将所有的变量都存放在主内存中，当线程使用变量时，会把主内存里面的变量复制到自己的工作空间或者叫作工作内存，线程读写变量时操作的是自己工作内存中的变量。

更具体的，可以认为主内存在物理机的内存条，而线程的工作内存是 L1 缓存，L2 缓存是所有 CPU 共享的。L1 缓存，L2 缓存，寄存器都可以是工作内存。当一个线程操作共享变量时，它首先从主内存复制共享变量到自己的工作内存，然后对工作内存里的变量进行处理，处理完后将变量值更新到主内存。线程安全问题就发生在工作内存与主内存交互数据的过程中。

另外，CPU 进行上下文切换时也会消耗资源，因为需要将工作内存与主内存中数据交换。如果是单 CPU，那么多线程并不能带来效率提升，反而会降低效率。

内存可见性是指当一个线程修改了某个变量的值，其它线程总是能知道这个变量变化。

2.2 synchronized 与 volatile 关键字

synchronized 关键字

synchronized 块是 Java 提供的一种原子性内置锁, Java 中的每个对象都可以把它当作一个同步锁来使用。这些 Java 内置的使用者看不到的锁被称为内部锁，也叫作监视器锁。

线程的执行代码在进入 synchronized 代码块前会自动获取内部锁，这时候其他线程访问该同步代码块时会被阻塞。拿到内部锁的线程会在正常退出同步代码块或者抛出异常后或者在同步块内调用了该内置锁资源的 wait 系列方法时释放该内置锁。

内置锁是排它锁，也就是当一个线程获取这个锁后，其他线程必须等待该线程释放锁后才能获取该锁。

Java 中的线程是与操作系统的原生线程一一对应的，所以当阻塞一个线程时，需要从用户态切换到内核态[4]执行阻塞操作，这是很耗时的操作，而 synchronized 的使用就会导致上下文切换。

synchronized 的内存语义可以将 synchronized 块内使用到的变量从线程的工作内存中清除. 这样在 synchronized 块内使用到该变量时就不会从线程的工作内存中获取，而是直接从主内存中获取。退出 synchronized 块的内存语义是把在 synchronized 块内对共享变量的修改刷新到主内存。

说人话一点, 使用到了 synchronized 的资源, 线程操作该资源有如下变动:

- 线程获取锁: 从主存中获取资源.
- 线程释放锁: 将资源刷新到主存, **清空工作内存中对应资源数据**.

这也是加锁和释放锁的语义，当获取锁后会清空锁块内本地内存中将会被用到的共享变量，在使用这些共享变量时从主内存进行加载，在释放锁时将本地内存中修改的共享变量刷新到主内存。

除可以解决共享变量内存可见性问题外，synchronized 经常被用来实现原子性操作。另外请注意，synchronized 关键字会引起线程上下文切换并带来线程调度开销。

synchronized 可以修饰四种不同类型的代码块:

- 实例方法: 锁对象是 this，即对象本身。
- 静态方法：锁对象是类，JVM 中只有一个类，一个类只能有一个线程在运行。
- 代码块: 锁对象是 () 中的对象。
- 静态代码块: 锁对象是类。

volatile 关键字

使用 synchronized 锁方式可以解决共享内存可见性问题, 但是锁太笨重, 切换上下文会带来很大开销. 对于解决内存可见性问题, Java 还提供了一种弱形式的同步, 也就是使用 volatile 关键字。该关键字可以确保对一个变量的更新对其他线程马上可见, 同时不会加锁。

[4]操作系统针对硬件与用户程序的两种操作状态.

当一个变量被声明为 volatile 时，线程在写入变量时不会把值缓存在寄存器或者其他地方，而是会把值刷新回主内存。当其他线程读取该共享变量时，会从主内存重新获取最新值，而不是使用当前线程的工作内存中的值。可以理解为在写入变量时**不再使用 CPU 缓存**, 注意计算时还是要用。

```
public class ThreadSafeInteger {
    private volatile int value;
    public int get() {
        return value;
    }
    public void set(int value) {
        this.value = value;
    }
}
```

相比 synchronized 关键字, volatile 有如下优缺点:

- 不需要加锁, 不会阻塞其他线程。
- 不能保证操作的原子性。

那么什么情况下可以不用 synchronized 而用 volatile 关键字呢: 写入变量不依赖变量当前值, 即不存在计算过程。

此外 volatile 只能修饰成员变量。

指令重排序

Java 内存模型允许编译器和处理器对指令重排序以提高运行性能，并且只会对**不存在数据依赖性**的指令重排序。例如如下代码:

```
int a = 1;     // (1)
int b = 2;     // (2)
int c = b - a; // (3)
```

上诉代码 (1) 和 (2) 之间不存在依赖关系，执行顺序可以互换，不一定是显示的执行顺序。(3) 依赖 (1) 和 (2)，必须在这两句之后执行。

重排序在多线程下会导致非预期的结果出现，使用 volatile 关键字修饰对应的的成员变量可以避免重排序。

写 volatile 变量时，可以确保 volatile 写之前的操作不会被编译器重排序到 volatile 写之后。读 volatile 变量时，可以确保 volatile 读之后的操作不会被编译器重排序到 volatile 读之前。

2.3 原子性，CAS 操作与 Unsafe 类

原子性操作是指, 一系列操作, 要么全部执行, 要么全部不执行。比如前面说过的 count++ 实质上有三个步骤，如果我们使用 java -c 查看汇编代码，会发现有四个步骤，所以 count++ 不是一个原子操作。

保证原子性最简单 (目前只知道) 的方法是使用 synchronized 关键字，但是 synchronized 加锁会有较高的性能开销，而且会造成线程阻塞。如果使用 volatile 关键字可以保证部分内存可见性，但是不能解决 “读-改-写” 原子性问题。

CAS 操作

CAS(Compare and Swap) 是 JDK 提供的非阻塞原子性操作，通过**硬件**保证了比较-更新操作的原子性。JDK 里面的 Unsafe 类提供了一系列的 compareAndSwap* 方法，下面看下 compareAndSwapLong 方法 (了解，许多原子操作通过 CAS 实现，一般不会手动调用 CAS 操作):

- boolean compareAndSwapLong(Object obj,long valueOffset,long expect, long update)
  - obj: 对象内存位置。
  - valueOffset: 对象中的变量的偏移量。
  - expect: 变量预期值。
  - update: 新的值。

其操作含义是，如果对象 obj 中内存偏移量为 valueOffset 的变量值为 expect，则使用新的值 update 替换旧的值 expect。这是处理器提供的一个原子性指令。

CAS 存在 ABA 问题，简单阐述如下: 线程 I 将数据 A 变成了 B，然后又重新变成了 A，此时另一个线程读取该数据时，发现 A 没有变化，就误认为是原来的 A，到那时 A 的一些属性或者状态已经发生了变化。

ABA 问题的产生是因为变量的状态值产生了环形转换，就是变量的值可以从 A 到 B，然后再从 B 到 A。如果变量的值只能朝着一个方向转换，比如 A 到 B，B 到 C，不构成环形，就不会存在问题。JDK 中的 AtomicStampedReference 类给每个变量的状态值都配备了一个时间戳，从而避免了 ABA 问题的产生。

Unsafe 类

Unsafe 提供了很多硬件级别的原子性操作，类中的方法本质上都是 native 方法，通过 JNI 访问本地 C++ 实现库，几个需要了解的方法有:

- long objectFieldOffset(Field f)

  访问指定的变量在所属类中的内存偏移地址，该地址仅仅在该 Unsafe 函数中使用。
- int arrayBaseOffset(Class<?> arrayClass)

  获取第一个数组元素的地址。
- int arrayIndexScale(Class<?> arrayClass)

  获取数组中一个元素占用的字节。
- long getLongVolatile(Object o, long offset)

  获取对象 obj 中偏移量为 offset 的变量对应 volatile 语义的值。
- void putLongVolatile(Object o, long offset, long x)

  设置 obj 对象中 offset 偏移的类型为 long 的 field 的值为 value，支持 volatile 语义。

- void putOrderedLong(Object o, long offset, long x)

  有延迟的 putLongvolatile 方法，并且不保证值修改对其他线程立刻可见。只有在变量使用 volatile 修饰并且预计会被意外修改时才使用该方法。

- void park(boolean isAbsolute, long time)

  阻塞当前线程，time 表示经过多长时间被唤醒。isAbsolute 用于和 time 配合产生不同效果。

- void unpark(Object thread)

  唤醒被 park 阻塞的线程。

- long getAndSetLong(Object o, long offset, long newValue)

  获取对象 obj 中偏移量为 offset 的变量 volatile 语义的当前值，并设置变量 volatile 语义的值为 update。

- long getAndAddLong(Object o, long offset, long delta)

  获取对象 obj 中偏移量为 offset 的变量 volatile 语义的当前值，并设置变量值为原始值 +addValue。

2.4 伪共享

为了解决计算机主存与 CPU 运行速度差问题，会在 CPU 与内存之间添加一级或多级高速缓冲处理器 (cache)。cache 内部是按行存储的，其中每一行称为一个 cache 行，是 cache 与主存进行数据交换的单位。

当 CPU 访问某个变量时，会有如下操作顺序:

- 查 CPU cache，如果有直接从中获取; 如果没有，进行下一步。
- 查主存获取变量,然后把该变量所在的内存区域的一个 cache 行大小复制到 CPU cache 中。

在第二步中，由于存放 cache 行的是内存块而不是单个变量，所以可能会把多个变量存放到一个 cache 行中。当多个线程同时修改一个缓存里面的多个变量时，由于同时只有一个线程操作缓存行，所以相比每个变量放到一个缓存行，性能会有所下降，这被称为伪共享。

伪共享的产生是因为多个变量被放入了一个缓存行中，并且多个线程同时去写入缓存行中不同的变量。例如:

```
long a = 1;
long b = 2;
long c = 3;
long d = 4;
```

当需要读取变量 a 时，往往会把邻近的 b,c,d 变量一起放入缓存行中。多个线程的私有缓存都可能会读写同一缓存行，既有可能造成冲突，也会降低性能。

在单个线程下顺序修改一个缓存行中的多个变量，会充分利用程序运行的局部性原则，从而加速了程序的运行。而在多线程下并发修改一个缓存行中的多个变量时就会竞争缓存行，

从而降低程序运行性能。

在 JDK 8 之前一般都是通过字节填充的方式来避免该问题，也就是创建一个变量时使用填充字段填充该变量所在的缓存行，这样就避免了将多个变量存放在同一个缓存行中。

```
public final static class FilledLong {
    public volatile long value = 0L;
    public long p1,p2,p3,p4,p5,p6; // 填充数据
}
```

假如缓存行为 64 字节，那么 FilledLong 里有效变量 value 占 8 字节，6 个天从字段栈 48 字节，再加上对象头占 8 字节，这样正好 64 字节放入一个缓存行中。显然这样处理有一个明显缺陷: 写无用的代码。

JDK8 提供了一个 sun.misc.Contended 注解，用来解决伪共享问题。在默认情况下，@Contended 注解只用于 Java 核心类，比如 rt 包下的类。如果用户类路径下的类需要使用这个注解，则需要添加 JVM 参数：-XX:-RestrictContended。填充的宽度默认为 128，要自定义宽度则可以设置-XX:ContendedPaddingWidth 参数。

2.5 锁概念

乐观锁与悲观锁

乐观锁与悲观锁的概念源于数据库。

悲观锁指对数据被外界修改持保守态度，认为数据很容易就会被其他线程修改，所以在数据被处理前先对数据进行加锁，并在整个数据处理过程中，使数据处于锁定状态。悲观锁的实现往往依靠数据库提供的锁机制，即在数据库中，在对数据记录操作前给记录加排它锁。如果获取锁失败，则说明数据正在被其他线程修改，当前线程则等待或者抛出异常。如果获取锁成功，则对记录进行操作，然后提交事务后释放排它锁。

乐观锁是相对悲观锁来说的，它认为数据在一般情况下不会造成冲突，所以在访问记录前不会加排它锁，而是在进行数据提交更新时，才会正式对数据冲突与否进行检测。乐观锁并不会使用数据库提供的锁机制，一般在表中添加 version 字段或者使用业务状态来实现。乐观锁直到提交时才锁定，所以不会产生任何死锁。

公平锁与非公平锁

根据线程获取锁的抢占机制，锁可以分为公平锁和非公平锁。

公平锁表示线程获取锁的顺序是按照线程请求锁的时间早晚来决定的，也就是最早请求锁的线程将最早获取到锁。而非公平锁则在运行时闯入，也就是先来不一定先得。

ReentrantLock 提供了公平和非公平锁的实现。

- 公平锁: ReentrantLock pairLock = new ReentrantLock(true)。
- 非公平锁: ReentrantLock pairLock = new ReentrantLock(false), 不传参默认为 false。

在没有公平性需求的前提下尽量使用非公平锁，因为公平锁会带来性能开销。

独享锁与共享锁

根据锁只能被单个线程持有还是能被多个线程共同持有，锁可以分为独占锁和共享锁。

独占锁保证任何时候都只有一个线程能得到锁，ReentrantLock 就是以独占方式实现的。共享锁则可以同时由多个线程持有，例如 ReadWriteLock 读写锁，它允许一个资源可以被多线程同时进行读操作。

独占锁是一种悲观锁，由于每次访问资源都先加上互斥锁，这限制了并发性，因为读操作并不会影响数据的一致性，而独占锁只允许在同一时间由一个线程读取数据，其他线程必须等待当前线程释放锁才能进行读取。

共享锁则是一种乐观锁，它放宽了加锁的条件，允许多个线程同时进行读操作。

可重入锁

当一个线程要获取一个被其他线程持有的独占锁时，该线程会被阻塞，那么当一个线程再次获取它自己已经获取的锁时是否会被阻塞呢? 如果不被阻塞，那么我们说该锁是可重入的，也就是只要该线程获取了该锁，那么可以无限次数（严格来说是有限次数）地进入被该锁锁住的代码。

```
public class Hello {
    public synchronized void helloA() {
        System.out.println("Hello A!");
    }

    public synchronized void helloB() {
        System.out.println("Hello B!");
        helloA;
    }
}
```

上述代码调用 helloB 之前会先获取内置锁，打印语句，然后尝试调用 helloA 方法，在调用 helloA 之前会先再次尝试获取内置锁，如果锁时不可重入的 (不可再次获取已获取的锁)，那么调用线程会一直被阻塞。

synchronized 内部锁是可重入锁。可重入锁的原理是在锁内部维护一个线程标示，用来标示该锁目前被哪个线程占用，然后关联一个计数器。一开始计数器值为 0，说明该锁没有被任何线程占用。

- 当一个线程获取了该锁时，计数器的值会变成 1，这时其他线程再来获取该锁时会发现锁的所有者不是自己而被阻塞。
- 但是当获取了该锁的线程再次获取锁时发现锁拥有者是自己，就会把计数器值加 +1，当释放锁后计数器值-1。当计数器值为 0 时，锁里面的线程标示被重置为 null，这时候被阻塞的线程会被唤醒来竞争获取该锁。

自旋锁

由于 Java 中的线程是与操作系统中的线程一一对应的，所以当一个线程在获取锁（比如独占锁）失败后，会被切换到内核状态而被挂起。当该线程获取到锁时又需要将其切换到内核状态而唤醒该线程。而从用户状态切换到内核状态的开销是比较大的，在一定程度上会影响并发性能。

自旋锁则是，当前线程在获取锁时，如果发现锁已经被其他线程占有，它不马上阻塞自己，在不放弃 CPU 使用权的情况下，多次尝试获取（默认次数是 10，可以使用-XX:PreBlockSpinsh 参数设置该值），很有可能在后面几次尝试中其他线程已经释放了锁。如果尝试指定的次数后仍没有获取到锁则当前线程才会被阻塞挂起。由此看来自旋锁是使用 CPU 时间换取线程阻塞与调度的开销，但是很有可能这些 CPU 时间白白浪费了。

II 并发工具

3 常用并发工具

3.1 ThreadLocalRandom

3.1.1 Random 的原理与局限性

首先看一下 Random 类的声明:

```
public class Random implements RandomGenerator, java.io.Serializable
```

Random 有两个构造函数:

```
public Random() {
    this(seedUniquifier() ^ System.nanoTime());
}
public Random(long seed) {
    ...
}
```

可以看出，无论使用哪个构造函数，都会初始化一个 seed 成员，这个种子决定了随机数是怎样生成的，所有的随机算法都依赖于这个种子 (伪随机)。来看一下是怎样利用种子生成随机数的。

```
public int nextInt(int bound) {
    if (bound <= 0)
        throw new IllegalArgumentException(BAD_BOUND);
    // (1) 根据老种子生成新种子
    int r = next(31);
    // (2) 生成随机数
    int m = bound - 1;
    if ((bound & m) == 0)
        r = (int)((bound * (long)r) >> 31);
    else {
        for (int u = r;
             u - (r = u % bound) + m < 0;
             u = next(31))
            ;
    }
    return r;
}
```

由此可见，新的随机数生成需要两个步骤:

- 根据老的种子生成新的种子。

- 根据新的种子生成随机数。

在单线程情况下每次调用 nextInt 都是根据老的种子计算出新的种子，这是可以保证随机数产生的随机性的。但是在多线程下多个线程可能都拿同一个老的种子去执行计算新的种子，这会导致多个线程产生的新种子是一样的，由于生成随机数的算法是相同的，最终多个线程产生的随机值也会相同。

所以步骤 (1) 要保证原子性，也就是说当多个线程根据同一个老种子计算新种子时，第一个线程的新种子被计算出来后，第二个线程要丢弃自己老的种子，而使用第一个线程的新种子来计算自己的新种子，依此类推，只有保证了这个，才能保证在多线程下产生的随机数是随机的。Random 函数使用一个原子变量达到了这个效果，在创建 Random 对象时初始化的种子就被保存到了种子原子变量里面，下面看 next() 的代码。

```
protected int next(int bits) {
    long oldseed, nextseed;
    AtomicLong seed = this.seed;
    do {
        oldseed = seed.get();
        nextseed = (oldseed * multiplier + addend) & mask;
    // (3) CAS 操作，使用新的种子替换老的种子
    } while (!seed.compareAndSet(oldseed, nextseed));
    // 根据种子计算随机数
    return (int)(nextseed >>> (48 - bits));
}
```

在多线程情形下，可能多个线程都执行到了 (3) 之前，拿到且计算的新种子是一样的，但是 (3) 的 CAS 操作会保证只有一个线程可以更新老的种子为新的，失败的线程会循环获取新的种子，这就解决了上面提到的问题，保证了随机数的随机性。

3.1.2 ThreadLocalRandom

Random 的缺陷在于在多线程操作过程中，他会采用 while 循环不断尝试获取最新的种子值，这会影响性能。为了弥补这一缺陷，Java 提供了 ThreadLocalRandom 工具。

```
public class ThreadLocalRandom extends Random
```

ThreadLocalRandom 会让我们联想到 ThreadLocal(事实上原理相同)，既然 Random 的缺陷在于需要不断尝试竞争获取最新的“公有”种子，那么让每个线程持有一个不同的种子不就行了吗。这两者的区别在于:

- Random: 多个线程竞争获取 Random 中的种子。
- ThreadLocalRandom: 每个线程持有一个不同的私有的种子，ThreadLocalRandom 只负责计算随机数。

在 Thread 类中，有以下相关的成员变量 (关键成员只有第一个，其他两个都是辅助生成第一个变量的):

```
long threadLocalRandomSeed;  // 随机种子
```

```
int threadLocalRandomProbe;  // 辅助生成随机种子
int threadLocalSecondarySeed; // 辅助生成随机种子
```

和 ThreadLocal 类似，ThreadLocalRandom 也持有一个 Thread 对象引用。具体的操作原理不解释了，和 ThreadLocal 一致。

ThreadLocalRandom 没有公有的构造方法，获取 ThreadLocalRandom 需要使用它的静态方法 current():

```
public static ThreadLocalRandom current() {
    // Unsafe 检测
    if (U.getInt(Thread.currentThread(), PROBE) == 0)
        localInit();  // 生成当前线程的随机种子
    return instance;  // ThreadLocalRandom 实例
}
```

ThreadLocalRandom 的很多方法都使用到了 Unsafe 的方法，用于和线程之间交互。

3.2 Atomic 原子操作

Java 的原子类都直接或间接用到了 Unsafe 方法，这些方法是 native 的，我们无法知道具体的实现方案。因此，这节多是概念，具体实现逻辑也没什么好讲的，问就是 CAS。

3.2.1 AtomicLong 操作

原子变量包括 AtomicBoolean，AtomicInteger，AtomicLong 等原子操作，原理类似，以 AtomicLong 为例。

```
public class AtomicLong extends Number implements java.io.Serializable
```

它的几个重要成员如下:

```
// 判断 JVM 是否支持 Long 类型无锁 CAS
static final boolean VM_SUPPORTS_LONG_CAS = VMSupportsCS8();
// 获取 Unsafe 实例
private static final Unsafe unsafe = Unsafe.getUnsafe();
// 存放变量 value 的偏移量
private static final long valueOffset;
// 实际变量值
private volatile long value;
```

value 被声明为 volatile，是因为其操作已经是原子操作，那么就可以使用 volatile 保证其内存可见性，这样是线程安全的。

AtomicLong 的大部分功能性方法最终都是调用了 CAS 操作，我们只需要知道，CAS 可以保证原子性，是非阻塞的，相比 synchronized 关键字，CAS 操作性能更高。具体 CAS 是如何操作的涉及到硬件层面。

3.2.2 LongAdder 操作

AtomicLong 通过 CAS 提供了非阻塞的原子性操作，虽然相比阻塞算法性能已经很好了，但是 Java 通过了性能更佳的方案。

在高并发下大量线程会同时去竞争更新同一个原子变量，但是由于同时只有一个线程的 CAS 操作会成功，大量线程竞争失败后会进入无限循环不断通过自旋尝试 CAS 操作，这会白白浪费 CPU 资源 (类似 Random 的局限)。

LongAdder 的思想是这样的: 既然 AtomicLong 的性能瓶颈是由于过多线程同时去竞争一个变量的更新而产生的，那么把一个变量分解为多个变量，让同样多的线程去竞争多个资源，就解决了性能问题。

使用 LongAdder 时，是在内部维护多个 Cell 变量，每个 Cell 里面有一个初始值为 0 的 long 型变量，这样，在同等并发量的情况下，争夺单个变量更新操作的线程量会减少，这变相地减少了争夺共享资源的并发量。另外，多个线程在争夺同一个 Cell 原子变量时如果失败了，它并不是在当前 Cell 变量上一直自旋 CAS 重试，而是尝试在其他 Cell 的变量上进行 CAS 尝试，这个改变增加了当前线程重试 CAS 成功的可能性。最后，在获取 LongAdder 当前值时，是把所有 Cell 变量的 value 值累加后再加上 base 返回的。

LongAdder 维护了一个延迟初始化的原子性更新数组（默认情况下 Cell 数组是 null）和一个基值变量 base。由于 Cells 占用的内存是相对比较大的，所以一开始并不创建它，而是在需要时创建，也就是惰性加载。

当一开始判断 Cell 数组是 null 并且并发线程较少时，所有的累加操作都是对 base 变量进行的。保持 Cell 数组的大小为 2 的 N 次方，在初始化时 Cell 数组中的 Cell 元素个数为 2，数组里面的变量实体是 Cell 类型。Cell 类型是 AtomicLong 的一个改进，用来减少缓存的争用，也就是解决伪共享问题。

对于大多数孤立的多个原子操作进行字节填充是浪费的，因为原子性操作都是无规律地分散在内存中的（也就是说多个原子性变量的内存地址是不连续的），多个原子变量被放入同一个缓存行的可能性很小。但是原子性数组元素的内存地址是连续的，所以数组内的多个元素能经常共享缓存行，因此使用 @sun.misc.Contended 注解对 Cell 类进行字节填充，这防止了数组中多个元素共享一个缓存行，在性能上是一个提升。

```
1 public class LongAdder extends Striped64 implements Serializable
```

Striped64 维护着 3 个主要成员变量:

```
1 transient volatile Cell[] cells;
2 transient volatile long base;
3 transient volatile int cellsBusy;
```

cellsBusy 状态只有 0 或 1，用来实现自旋锁，当创建 Cell 元素，扩容 Cell 数组或者初始化 Cell 数组时，使用 CAS 操作该变量来保证同时只有一个线程可以进行其中之一的操作。

Cell 的构造很简单，其内部维护一个被声明为 volatile 的 value 变量，这里声明为 volatile 是因为线程操作 value 变量时没有使用锁，为了保证变量的内存可见性这里将其声明为 volatile

的。另外有个 cas 函数通过 CAS 操作，保证了当前线程更新时被分配的 Cell 元素中 value 值的原子性。另外，Cell 类使用 @sun.misc.Contended 修饰是为了避免伪共享。

3.2.3 LongAccumlator

LongAdder 类是 LongAccumulator 的一个特例，LongAccumulator 比 LongAdder 的功能更强大。

```
public class LongAccumulator extends Striped64 implements Serializable {
    private final LongBinaryOperator function;
    private final long identity;
    public LongAccumulator(LongBinaryOperator accumulatorFunction, long identity) {
        this.function = accumulatorFunction;
        base = this.identity = identity;
    }
}
```

调用 LongAdder 就相当于通过如下方式构建 LongAccumulator:

```
LongAccumulator longAccumulator = new LongAccumulator((long left, long right) -> {return
    (left + right);},0);
```

LongAccumulator 相比于 LongAdder，可以为累加器提供非 0 的初始值，后者只能提供默认的 0 值。另外，前者还可以指定累加规则，比如不进行累加而进行相乘，只需要在构造 LongAccumulator 时传入自定义的双目运算器即可，后者则内置累加的规则。

4 锁的原理

4.1 LockSupport

LockSupport 是个工具类，没有公有的构造函数。它的主要作用是挂起和唤醒线程，是创建锁和其他同步类的基础:

```
public class LockSupport {
    private static final Unsafe U = Unsafe.getUnsafe();
}
```

LockSupport 类与每个使用它的线程都会关联一个许可证，在默认情况下调用 LockSupport 类的方法的线程是不持有许可证的。

park() 方法

park 系方法是 LockSupport 的核心方法，本质上是调用 Unsafe 的 park() 方法，具体方法实现是不可知的:

```
public static void park() {
    U.park(false, 0L);
}
```

如果调用 park 方法的线程已经拿到了与 LockSupport 关联的许可证，则调用 LockSupport.park() 时会马上返回，否则调用线程会被禁止参与线程的调度，也就是会被阻塞挂起。

在其他线程调用 unpark(Thread thread) 方法并且将当前线程作为参数时，调用 park 方法而被阻塞的线程会返回。另外，如果其他线程调用了阻塞线程的 interrupt() 方法，设置了中断标志或者线程被虚假唤醒，则阻塞线程也会返回。所以在调用 park 方法时最好也使用循环条件判断方式。

需要注意的是，因调用 park() 方法而被阻塞的线程被其他线程中断而返回时并不会抛出 InterruptedException 异常。

unpark(Thread thread) 方法

```
public static void unpark(Thread thread) {
    if (thread != null)
        U.unpark(thread);
}
```

当一个线程调用 unpark 时，如果参数 thread 线程没有持有 thread 与 LockSupport 类关联的许可证，则让 thread 线程持有。如果 thread 之前因调用 park() 而被挂起，则调用 unpark 后，该线程会被唤醒。如果 thread 之前没有调用 park，则调用 unpark 方法后，再调用 park 方法，其会立刻返回。

park 方法返回时不会告诉你因何种原因返回，所以调用者需要根据之前调用 park 方法的

原因，再次检查条件是否满足，如果不满足则还需要再次调用 park 方法。例如:

```
// 调用 park 方法，挂起自己，只有被中断才退出
while(!Thread.currentThread().isInterrupted) {
    LockSupport.park();
}
```

其他 park 系方法

park 系方法还有几个参数:

- long nanos: 如果没有拿到许可证，则调用线程会被挂起 nanos 时间后修改为自动返回。
- long deadline: 和 nAQS 类似，nanos 是从当前时间开始等待多久，deadline 是等待到什么时间。
- Object blocker: 当线程在没有持有许可证的情况下调用 park 方法而被阻塞挂起时，这个 blocker 对象会被记录到该线程内部，该对象通常为 this，即调用线程的类本身。

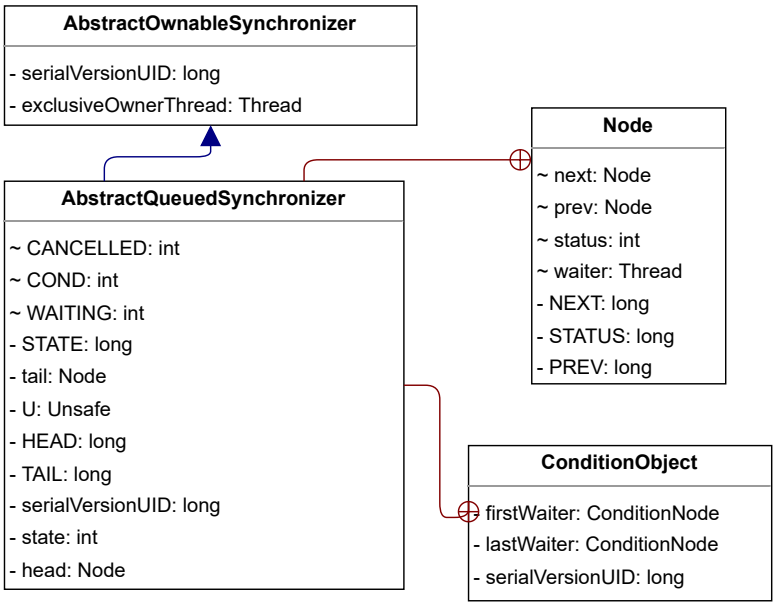
。

4.2 AQS 抽象同步队列

4.2.1 AQS 基础介绍

AbstractQueuedSynchronizer 抽象同步队列简称 AQS，它是实现同步器的基础组件，并发包中锁的底层就是使用 AQS 实现的。开过过程中几乎不会直接调用 AQS，理解即可。

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



AQS 核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的

工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是用 CLH(虚拟的双向队列) 队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

AQS 是一个 FIFO 双向队列，他有两个关键的内部类，他们的关键属性如下:

- Node: 队列元素 (没有公有方法)
  - Thread waiter: 存放进入 AQS 队列的线程。Node 有两个子类用于标记线程阻塞原因:
    - SharedNode: 标记线程是获取共享资源时被挂起进入 AQS 队列。
    - ExclusiveNode: 标记线程时获取独占资源时被挂起后进入 AQS 队列。
  - int status: 用于记录当前线程等待状态, 提供了三个常量:
    - WAITING: 普通的等待状态。
    - CANCELLED: 线程被取消。
    - COND: 条件等待。
- ConditionObject: 条件变量，加强版 notify 和 wait 操作，下文细嗦。
  - ConditionNode: 继承自 Node 的内部类，记录等待节点顺序。
  - firstWaiter/lastWaiter: 用于记录第一和最后一个等待线程节点。

AQS 最关键的属性是 state，不同的锁对其有不同的用法:

- ReentrantLock: 表示当前线程获取锁的可重入次数。
- ReentrantReadWriteLock: 高于 16 表示读状态，低于 16 表示可重入次数。
- semaphore: 当前可用信号数。
- CountDownlatch: 计数器当前的值。

对于 AQS 来说，线程同步的关键是对状态值 state 进行操作。根据 state 是否属于一个线程，操作 state 的方式分为独占方式和共享方式。AQS 通过 acquire 系方法获取资源，release 系方法释放资源。共享和独占资源对应的方法不同:

- 使用独占方式获取的资源是与具体线程绑定的，就是说如果一个线程获取到了资源，就会标记是这个线程获取到了，其他线程再尝试操作 state 获取资源时会发现当前该资源不是自己持有的，就会在获取失败后被阻塞。
- 对应共享方式的资源与具体线程是不相关的，当多个线程去请求资源时通过 CAS 方式竞争获取资源，当一个线程获取到了资源后，另外一个线程再次去获取时如果当前资源还能满足它的需要，则当前线程只需要使用 CAS 方式进行获取即可。

获取与释放资源

在前面我们已经了解到，state 属性值的具体含义由子类赋予，因此与之相关的操作也全部由子类负责，父类仅提供接口。

在独占模式下，获取与释放资源有如下过程:

- 获取资源: acquire(int arg) 方法。

```java
public final void acquire(int arg) {
    if (!tryAcquire(arg)) // 具体实现由子类提供
        acquire(null, arg, false, false, false, 0L);
}
```

  - 首先会使用 tryAcquire(arg) 方法尝试获取资源，具体是设置状态变量 state 的值。
  - 如果成功，直接返回。
  - 如果失败，将当前线程封装为 ExclusiveNode 插入到 AQS 队列的尾部，调用 LockSupport.park 方法挂起自己。

- 释放资源: release(int age) 方法。

```java
public final boolean release(int arg) {
    if (tryRelease(arg)) { // 同样由子类实现
        signalNext(head); // 激活第一个被阻塞的线程并尝试获取资源
        return true;
    }
    return false;
}
```

  - 首先尝试使用 tryRelease 操作释放资源，具体也是操作 state 的值。
  - 调用 LockSupport.unpark 激活 AQS 队列里被阻塞的第一个线程。
  - 被激活的线程调用 acquire 方法，尝试获取资源。

在共享模式下，获取与释放资源的操作类似，不过在获取资源失败后会线程会被封装为 SharedNode 插入队尾。

除了上述介绍的方法外，还需要重写 isHeldExclusively() 方法，来判断锁是否已经被持有。

AQS 还提供了带 Interruptibly 后缀的 acquire 方法，不带 Interruptibly 关键字的方法的意思是不对中断进行响应，那么该线程不会因为被中断而抛出异常，而带 Interruptibly 关键字的方法要对中断进行响应。

虚拟的双向队列不存在队列实例，仅存在结点之间的关联关系，因此 AQS 每次入队操作实质上都是将线程封装成 Node 节点后添加进双向链表的操作。

4.2.2 条件变量的支持

正如 notify 和 wait 是配合 synchronized 内置锁实现线程间同步，ConditionObject(条件变量) 的 signal 和 await 方法也是用来配合锁 (AQS 实现的锁) 实现线程同步的。不同点在于 synchronized 是共享线程独占锁，而 AQS 的锁可以对应多个条件变量。

```java
public class ConditionObject implements Condition, java.io.Serializable
```

在调用共享变量的 notify 和 wait 方法前必须先获取该共享变量的内置锁，同理，在调用条件变量的 signal 和 await 方法前也必须先获取条件变量对应的锁。

下面看一个 ReentrantLock(AQS 实现) 例子:

```java
ReentrantLock lock = new ReentrantLock(); // 创建一个 ReentrantLock 锁
```

```
Condition condition = lock.newCondition(); // 创建一个条件变量

lock.lock();  // 获取锁
try {
    System.out.println("begin wait");
    condition.await();    // 阻塞挂起当前线程
    System.out.println("end wait");
} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    lock.unlock();    // 释放锁
}
```

这里的一些操作解释如下:

- Lock 对象: 等价于 synchronized 加上共享变量。
- lock.lock(): 等价于进入 synchronized 块尝试获取锁。
- lock.unlock(): 等价于退出 synchronized 块释放锁。
- await() 方法: 等价于 Object 的 wait() 方法。
- signal() 方法: 等价于 Object 的 notify() 方法。

当线程调用条件变量的 await() 方法时 (先调用 lock() 方法), 在内部会构造一个 ConditionNode 型的 node 节点, 然后插入节点, 之后当前线程会释放获取的锁并被阻塞挂起。await() 方法的部分源码如下:

```
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    // 创建新的节点
    ConditionNode node = new ConditionNode();
    // 释放当前线程获取的锁, 插入节点
    int savedState = enableWait(node);
    // 阻塞挂起当前线程
    LockSupport.setCurrentBlocker(this);
    ......
}
```

继续深入看一下如何获取锁并插入节点:

```
private int enableWait(ConditionNode node) {
    if (isHeldExclusively()) { // 锁被持有
        node.waiter = Thread.currentThread(); // 将线程装入节点
        node.setStatusRelaxed(COND | WAITING);
        ConditionNode last = lastWaiter;
        if (last == null)
            firstWaiter = node;
        else
            last.nextWaiter = node;
        lastWaiter = node;           // 插入节点完成, 注意这是一个单向队列
        int savedState = getState();
        if (release(savedState))
```

```
            return savedState;
        }
        node.status = CANCELLED; // 获取锁失败
        throw new IllegalMonitorStateException();
    }
```

当多个线程同时调用 lock.lock() 方法获取锁时，只有一个线程获取到了锁，其他线程会被转换为 Node 节点插入到 lock 锁对应的 AQS 阻塞队列里面，并做自旋 CAS 尝试获取锁。

如果获取到锁的线程又调用了对应的条件变量的 await() 方法，则该线程会释放获取到的锁，并被转换为 Node 节点插入到条件变量对应的条件队列[1]里面。

当另外一个线程调用条件变量的 signal() 或者 signalAll() 方法时，会把条件队列里面的一个或者全部 Node 节点移动到 AQS 的阻塞队列里面，等待时机获取锁。

一个锁对应一个 AQS 阻塞对象，对应多个条件变量。下图解释了条件队列与 AQS 阻塞队列的关系:

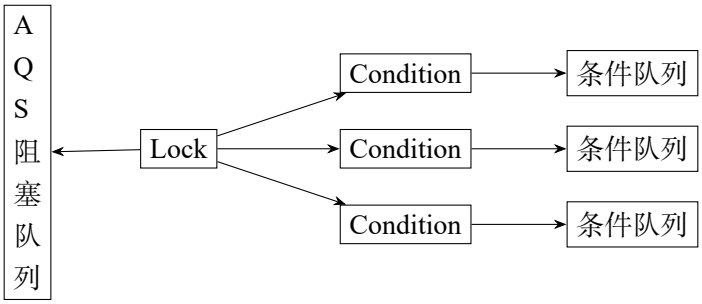


图 4.1 条件队列

4.3 ReentrantLock 独占锁

Lock 接口与 ReentrantLock 基础

所有锁都需要实现 Lock 接口，它提供了最基础的方法:

- void lock(): 获取锁。
- void lockInterruptibly(): 获取锁，被中断则抛异常。
- boolean tryLock(): 尝试获取锁，不会引起阻塞。
- boolean tryLock(long time, TimeUnit unit): 在一段时间内尝试获取锁。
- void unlock(): 释放锁。
- Condition newCondition(): 获取条件变量。

ReentrantLock 是可重入的独占锁，同时只能有一个线程可以获取该锁，其他获取该锁的线程会被阻塞而被放入该锁的 AQS 阻塞队列里面。

```
public class ReentrantLock implements Lock, java.io.Serializable
```

[1]条件队列: https://blog.csdn.net/weixin_33188789/article/details/114869897

ReentrantLock 只有一个功能性成员变量: Sync sync，Sync 继承自 AQS 重写了一些方法。ReentrantLock 本质上还是使用 AQS 来实现的，可以根据参数决定它是一个公平锁还是非公平锁。

```
public ReentrantLock() {
    sync = new NonfairSync();
}

public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

其中 NonfairSyn 和 FairSync 继承自 Sync，分别实现了锁的非公平与公平策略。

在这里，state 表示线程获取锁的可重入次数，默认情况下，state = 0 表示当前锁没有被任何线程持有。当一个线程第一次获取该锁时会尝试使用 CAS 设置 state 值为 1，如果 CAS 成功则当前线程获取了该锁，然后记录该锁的持有者为当前线程。在该线程没有释放锁的情况下第二次获取该锁后，状态值被设置为 2，这就是可重入次数。在该线程释放该锁时，会尝试使用 CAS 让状态值减 1，如果减 1 后状态值为 0，则当前线程释放该锁。

公平锁与非公平

我们说过 AQS 没有提供可用的 tryAcquire 方法，由子类提供具体实现。让我们看一下 ReentrantLock 的策略:

```
// FairSync
protected final boolean tryAcquire(int acquires) {
    if (getState() == 0 && !hasQueuedPredecessors() &&
        compareAndSetState(0, acquires)) {
        setExclusiveOwnerThread(Thread.currentThread());
        return true;
    }
    return false;
}
```

非公平锁只在 if 语句中少了 compareAndSetState(0, acquires) 判断。也即非公平锁不会判断共享资源的状态，直接进行抢占，那么我们可以梳理以下两种情况下线程抢占资源的过程:

- 非公平锁: 线程尝试获取资源 -> 尝试抢占 -> 失败则进入同步队列队尾。
- 公平锁: 线程尝试获取资源 -> 判断资源是否已被占用 -> 被占用进入队尾。

当线程进入同步队列时，涉及到用户态向内核态转换过程，这个过程比较消耗资源。而公平更有可能经历这个过程，因此，公平锁的效率往往更低。

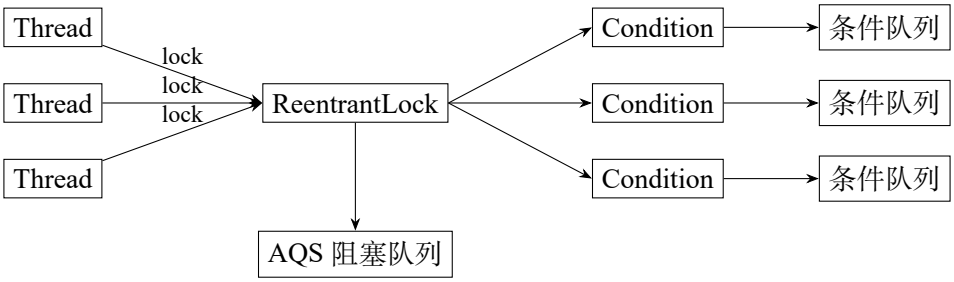


图 4.2 ReentrantLock

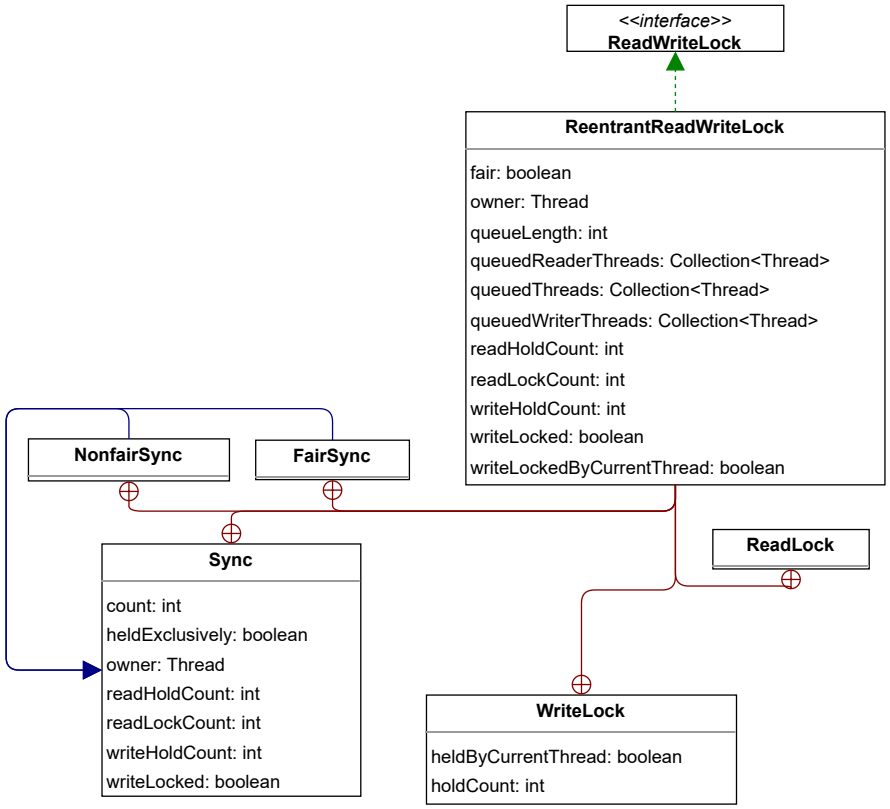
4.4 ReentrantReadWriteLock 读写锁

线程安全问题使用 ReentrantLock 即可，但是 ReentrantLock 是独占锁，某一时刻只能被一个线程获取，实际中存在读多写少的场景。ReentrantReadWriteLock 采用读写分离的策略，允许多个线程可以同时获取读锁。

读写锁的结构

读写锁实现了 ReadWriteLock:

```
1 public class ReentrantReadWriteLock implements ReadWriteLock, java.io.Serializable
```



ReadWriteLock 提供了两个方法，分别用于获取读锁和写锁:

```
1 public interface ReadWriteLock {
2     Lock readLock();
```

```
    Lock writeLock();
}
```

可以看出,不同于 ReentrantLock 直接实现 Lock 接口,其本身就是一个锁,而 ReentrantReadWriteLock 更像一个管理类，用于控制读锁和写锁。

ReentrantReadWriteLock 内部维护了一个 ReadLock 和一个 WriteLock，他们又依赖 Sync 实现功能。和 ReentrantLock 类似，Sync 也提供了公平锁和非公平锁。

我们知道，AQS 只维护一个 state，而 ReentrantReadWriteLock 需要维护两个锁，怎么办呢? 作为 int 型的 state 拥有 32 位，ReentrantReadWriteLock 将其一分为二，高位表示读状态，低位表示写状态:

```
// Sync
static final int SHARED_SHIFT = 16;
static final int SHARED_UNIT = (1 << SHARED_SHIFT);    // 读锁状态值
static final int MAX_COUNT   = (1 << SHARED_SHIFT) - 1; // 读锁最大值
static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1; // 写锁掩码
static int sharedCount(int c) { return c >>> SHARED_SHIFT; } // 读锁线程数
static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; } // 写锁可重入数
```

此外，Sync 还记录了首个获取读锁的线程，其重入次数，其他获线程获取读锁的可重入次数等等信息。

写锁的获取与释放

ReentrantReadWriteLock 中的写锁是独占锁，某时只有一个线程可以获取该锁。

- 当前没有线程获取读锁和写锁: 可以获取写锁并返回。
- 已有线程获取读锁和写锁: 当前请求写锁的线程被阻塞挂起。

来看看 tryAcquire 方法:

```
protected final boolean tryAcquire(int acquires) {
    Thread current = Thread.currentThread();
    int c = getState();
    int w = exclusiveCount(c); // 写锁可重入数
    // c != 0 说明已经有读锁或写锁被获取过
    if (c != 0) {
        // 写锁已被占有 且 不是被当前线程占用
        if (w == 0 || current != getExclusiveOwnerThread())
            return false;
        // 当前线程获取了写锁，重入次数达到上限
        if (w + exclusiveCount(acquires) > MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        // 重入
        setState(c + acquires);
        return true;
    }
    // 第一个获取写锁的线程
    if (writerShouldBlock() || !compareAndSetState(c, c + acquires))
```

```
        return false;
    setExclusiveOwnerThread(current);
    return true;
}
```

针对第一个获取写锁的线程，writeShouldBlock 有不同的做法:

```
\\ 非公平锁：总是 false，抢占式获取锁
final boolean writerShouldBlock() {
    return false;
}
\\ 公平锁：判断是否有前去节点，有则放弃抢占，进入阻塞队列
final boolean writerShouldBlock() {
    return hasQueuedPredecessors();
}
```

可以看出，ReentrantReadWriteLock 殊途同归地使用了和 ReentrantLock 类似的方法。

读锁的获取与释放

获取读锁的过程如下:

- 如果没有其他线程持有写锁，则当前线程可以获得读锁，AQS 状态值高 16 位值加一。
- 如果其他一个线程持有写锁，当前线程阻塞。

来看看 tryAcquireShared 方法:

```
protected final int tryAcquireShared(int unused) {
    Thread current = Thread.currentThread();
    int c = getState();
    // 写锁被占且不是当前线程占用
    if (exclusiveCount(c) != 0 && getExclusiveOwnerThread() != current)
        return -1;
    // 获取读锁计数
    int r = sharedCount(c);
    // 多个线程只有一个会成功获取读锁
    if (!readerShouldBlock() && r < MAX_COUNT && compareAndSetState(c, c + SHARED_UNIT)) {
        // 读锁未被占用过
        if (r == 0) {
            firstReader = current;
            firstReaderHoldCount = 1;
        // 当前线程是第一个获取读锁的线程
        } else if (firstReader == current) {
            firstReaderHoldCount++;
        // 记录最后一个获取读锁的线程或记录其他线程读锁的可重入数
        } else {
            HoldCounter rh = cachedHoldCounter;
            if (rh == null || rh.tid != LockSupport.getThreadId(current))
                cachedHoldCounter = rh = readHolds.get();
            else if (rh.count == 0)
                readHolds.set(rh);
            rh.count++;
```

```
        }
        return 1;
    }
    // 获取失败，重试
    return fullTryAcquireShared(current);
}
```

如果当前要获取读锁的线程已经持有了写锁，则也可以获取读锁。但是需要注意，当一个线程先获取了写锁，然后获取了读锁处理事情完毕后，要记得把读锁和写锁都释放掉，不能只释放写锁。

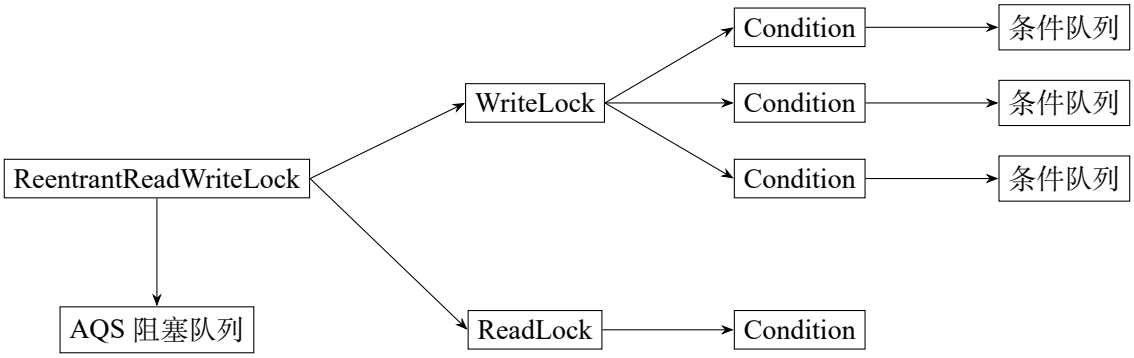


图 4.3 ReentrantReadWriteLock

4.5 StampedLock 戳记锁

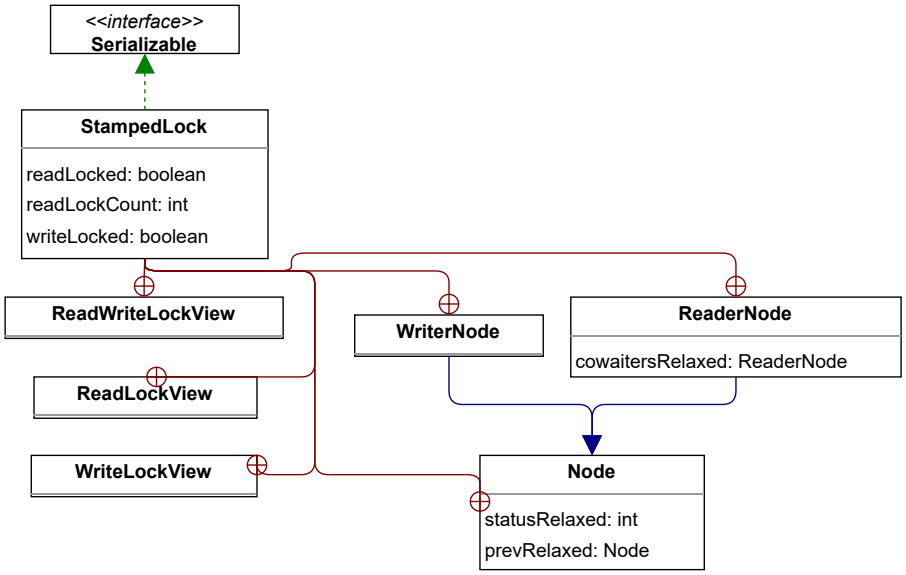
StampedLock 基础

StampedLock 是加强版的 ReentrantReadWriteLock, 提供了可操作性更强的功能:

```
public class StampedLock implements java.io.Serializable
```

注意，StampedLock 不是基于 AQS 实现的。

StampedLock 提供了三种模式的读写控制，当调用获取锁的系列函数时，会返回一个 long 型变量，称之为 stamp，stamp 代表了锁的状态，stamp 为 0 表示获取锁失败。



StampedLock 提供的几种读写模式锁分别如下:

- **写锁 writeLock**: 排他/独占锁，和 ReentrantReadWriteLock 读锁最大的不同是不可重入。
- **悲观读锁 readLock**: 共享锁，在没有线程获取独占写锁的情况下，多个线程可以同时获取该锁。如果已经有线程持有写锁，则请求该锁的线程被阻塞。同样不可重入。悲观指的是具体操作数据前会悲观地认为其他线程可能要对自己操作的数据进行修改，所以需要先对数据加锁，这是读少写多情况下地一种考虑。
- **乐观读锁 tryOptimisticRead**: 相对悲观锁来说，在操作数据前并没有通过 CAS 设置锁地状态，仅仅通过位运算测试。如果当前没有线程持有写锁，则简单地返回一个非 0 的 stamp 版本信息。

由于 tryOptimisticRead 并没有使用 CAS 设置锁状态，所以不需要显式地释放该锁。该锁的一个特点是适用于读多写少的场景，因为获取读锁只是使用位操作进行检验，不涉及 CAS 操作，所以效率会高很多。

但是同时由于没有使用真正的锁，在保证数据一致性上需要复制一份要操作的变量到方法栈，并且在操作数据时可能其他写线程已经修改了数据，而我们操作的是方法栈里面的数据，也就是一个快照，所以最多返回的不是最新的数据，但是一致性还是得到保障的。

StampedLock 还支持这三种锁在一定条件下进行相互转换。例如 long tryConvertToWriteLock(long stamp) 期望把 stamp 标记的锁升级为写锁，这个函数会在下面几种情况下返回一个有效的 stamp(也就是晋升写锁成功):

- 当前锁已经是写锁模式。
- 当前锁处于读锁模式，并且没有其他线程是读锁模式。
- 当前处于乐观读模式，并且当前写锁可用。

StampedLock 策略

StampedLock 通过一个 int 型和 state 和一个队列来实现，对 state 的分割如下:

- 高位的 24 位标识版本号，低位的 8 位表示锁。
- 低 8 位第 1 位表示是否为写锁，剩下 7 位表示悲观读锁的个数。

队列同样使用节点封装线程，采用双向链表实现。

StampedLock 的源码比较复杂，这里只讲三种锁的执行策略:

- 写锁的获取与释放
  - 获取写锁
    - 只有当 state 的低八位为 0000 0000 的才能申请写锁，即加锁前既没有写锁也没有读锁。
    - 成功获取锁，state 低八位变成 1000 0000，返回 state 值。
    - 申请失败，开始自旋，队列不为空直接阻塞，否则有限次数内自旋申请不到就阻塞。
    - 进入阻塞态，线程被封装为节点进入队尾，如果是第二个节点 (第一个为哨兵节点)，则开始新一轮自旋。
  - 释放写锁
    - 根据枪锁时返回的 stamp 检验异常。
    - 没有异常则将 state 版本号 +1，唤醒后继节点抢锁。
- 悲观读锁的获取与释放
  - 获取读锁
    - 如果该锁被另一个线程保持，则阻塞线程，否则自旋申请锁。
    - 如果申请锁失败，将线程封装为节点 node，此时分两种情况。
    - 1. 当前尾节点是写锁，则将 node 添加到队尾。等 node 成为第二个节点再开始自旋申请锁。
    - 2. 当前尾节点是读锁，则将 node 通过名为 cowait 指针与尾结点联系起来，然后阻塞。等尾节点抢锁时，唤醒的线程会通过 cowait 唤醒相关节点来抢锁。
  - 释放读锁
    - 根据枪锁时返回的 stamp 检验异常。
    - 没有异常则将 state 版本号 +1，唤醒后继节点抢锁。
- 乐观读锁: 没有锁，本地线程保存快照进行修改，返回时比较，失败则重试。

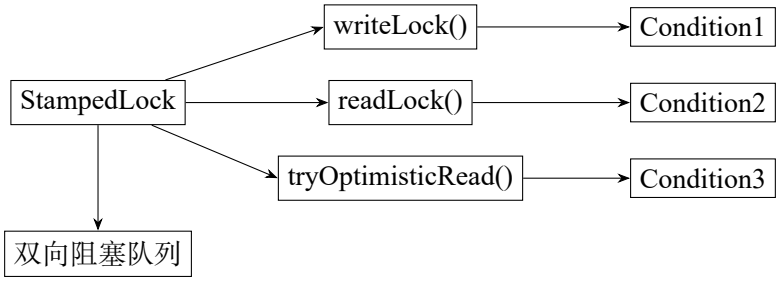


图 4.4 StampedLock

5 并发集合

本节默认读者看过 utils 包下非并发集合，掌握 Java 的基本数据结构。

5.1 List

JUC 只提供了一个并发列表，采用了写时复制策略。

5.1.1 CopyOnWriteArrayList

并发列表只有一个 CopyOnWriteList，顾名思义，它是一个线程安全的 ArrayList，对其进行的操作都是在底层的一个复制的数组 (快照) 上进行的，使用了写时复制策略。

```
// Java17
public class CopyOnWriteArrayList<E> implements List<E>, RandomAccess, Cloneable,
    java.io.Serializable {
    final transient Object lock = new Object();
    private transient volatile Object[] array;
}
```

在 Java8 中使用的时 ReentrantLock(一种可重入独占锁) 实现锁机制，而 Java17 中使用的是 synchronized 锁住普通的 Object 对象 lock。这两种机制实现的效果类似，没有很大差别，正如注释所述:

We have a mild preference for builtin monitors over ReentrantLock when either will do.

CopyOnWriteArrayList 在涉及到修改数组的操作时会使用 synchronized(lock) 的方式上锁，这里的修改是指任何非只读操作，包括增加，替换，排序，删除...

我们看一下 add 操作:

```
public boolean add(E e) {
    synchronized (lock) {
        Object[] es = getArray();
        int len = es.length;
        es = Arrays.copyOf(es, len + 1);
        es[len] = e;
        setArray(es);
        return true;
    }
}
```

最关键的是第一步操作，每次调用该操作都是尝试获取 lock 锁，如果有其他线程占用了 lock 锁 (不仅是 add 操作)，线程会被阻塞。

此外，我们会发现，CopyOnWriteArrayList 没有扩容机制，他在 add 时会直接申请一个新的空间，这非常费时间,set 也类似。

```
public E set(int index, E element) {
```

```
    synchronized (lock) {
        Object[] es = getArray();
        E oldValue = elementAt(es, index);
        if (oldValue != element) {
            es = es.clone();
            es[index] = element;
        }
        setArray(es);
        return oldValue;
    }
}
```

多线程操作可迭代对象存在弱一致性问题，这个问题是指返回迭代器后，其他线程对 list 的增删改对迭代器是不可见的。下面看看 CopyOnWriteArrayList 是如何解决这个问题的:

```
public Iterator<E> iterator() {
    return new COWIterator<E>(getArray(), 0);
}

static final class COWIterator<E> implements ListIterator<E> {
    // 数组快照
    private final Object[] snapshot;
    // 数组下标游标，用于迭代获取元素
    private int cursor;

    COWIterator(Object[] es, int initialCursor) {
        cursor = initialCursor;
        snapshot = es;
    }

    public boolean hasNext() {
        return cursor < snapshot.length;
    }

    @SuppressWarnings("unchecked")
    public E next() {
        if (! hasNext())
            throw new NoSuchElementException();
        return (E) snapshot[cursor++];
    }

    public int nextIndex() {
        return cursor;
    }

    // Not supported. Always throws UnsupportedOperationException.
    public void remove() {
        throw new UnsupportedOperationException();
    }
    public void set(E e) {
        throw new UnsupportedOperationException();
    }
```

```
38      public void add(E e) {
39          throw new UnsupportedOperationException();
40      }
41  }
```

获取 CopyOnWriteArrayList 的迭代器对象时，实际上会返回一个 COWIterator 对象，该对象通过 snapshot 保存了当前 list 的内容。注意 snapshot 本质上只是个指针。

如果在遍历期间，其他线程对该 list 进行了修改，那么 snapshot 就是快照了，因为增删改后 list 里面的数组被新数组替换了，这时候老数组被 snapshot 引用。这也说明获取迭代器后，使用该迭代器元素时，其他线程对该 list 进行的增删改不可见，因为它们操作的是两个不同的数组，这就是弱一致性。

这也间接说明，在调用 System 的 copyOf 后，原数组不会被立刻清除，如果没有引用才会被 GC。此外，COWIterator 对象不允许使用修改操作。

总而言之，CopyOnWriteArrayList 使用写时复制的策略来保证 list 的一致性，而获取—修改—写入三步操作并不是原子性的，所以在增删改的过程中都使用了独占锁，来保证在某个时间只有一个线程能对 list 数组进行修改。

另外 CopyOnWriteArrayList 提供了弱一致性的迭代器，从而保证在获取迭代器后，其他线程对 list 的修改是不可见的，迭代器遍历的数组是一个快照。

> **注意 5.1.** 一般情况下，不要使用 *CopyOnWriteArrayList*，它只有一个优点：保证线程安全。但是写时复制策略会照成很大的时间和空间开销，他没有 *ArrayList* 的扩容机制，迭代器会保存快照，导致旧的数组空间无法被 *GC*。只有极少情况，需要保存快照且需要保证线程安全时才会用到它。

5.2 Set

5.2.1 CopyOnWriteSet

并发集合 (Set)，JUC 也只提供了一个 CopyOnWriteSet，从名字就知道他是用的写时复制技术，本质上它是用 CopyOnWriteArrayList 实现的[2]。

```
1   public class CopyOnWriteArraySet<E> extends AbstractSet<E> implements java.io.Serializable {
2
3       private final CopyOnWriteArrayList<E> al;
4       public CopyOnWriteArraySet() {
5           al = new CopyOnWriteArrayList<E>();
6       }
7
8       public boolean add(E e) {
9           return al.addIfAbsent(e);
10      }
11
12      public boolean contains(Object o) {
```

[2]最常用的非并发集合是 HashSet，采用 HashMap 实现，这何尝不是一种牛头人行为。

```
        return al.contains(o);
    }

    public boolean remove(Object o) {
        return al.remove(o);
    }
}
```

5.3 Queue

JUC 提供了很多队列，多到数量是其他容器数量之和。大致可分为两种:

- **ConcurrentLinkedQueue**: 纯 CAS 实现的非阻塞高并发队列。
- **BlockingQueue**: 实现了 BlockingQueue 接口的阻塞队列，使用锁机制实现，并发性相对低。

5.3.1 ConcurrentLinkedQueue

顾名思义，ConcurrentLinkedQueue 是使用” 单向” 链表实现的线程安全队列，出队和入队操作使用 CAS 实现线程安全。

```
public class ConcurrentLinkedQueue<E> extends AbstractQueue<E> implements Queue<E>,
    java.io.Serializable
```

ConcurrentLinkedQueue 内部的队列使用单向链表方式实现，其中有两个 volatile 类型的 Node 节点分别用来存放队列的首、尾节点。

```
transient volatile Node<E> head;
private transient volatile Node<E> tail;
```

默认情况下，head 和 tail 时指向空节点即哨兵节点:

```
public ConcurrentLinkedQueue() {
    head = tail = new Node<E>();
}
```

Node 节点很简单，只有两个元素用于存储值和下一个 Node 的指针:

```
volatile E item;
volatile Node<E> next;
```

ConcurrentLinkedQueue 有一个特殊之处，tail 节点并不一定是尾节点，也有可能是尾节点的前一个节点。为什么要这样做呢?

- 假设我们不这样做，让 tail 节点始终是尾部节点，那么我们每次添加元素都需要对 tail 进行 CAS 操作 (注意只是 tail 节点)。这样的好处是代码简单，缺陷是频繁的 CAS 操作效率低。

- 如果我们每两个或者多个节点对 tail 节点进行 CAS 操作，就可以减少 CAS 更新 tail 节点的次数，提高入队的效率。

在 Java1.7 中，可以使用 hops 变量控制 tail 节点每添加多少新节点才更新，这是一种优化策略。hops 值小则需要频繁的对 tail 进行 CAS 操作，过大则需要通过链表遍历查找尾节点，violate 修饰的 tail 节点对读有更好的优化。因此在 Java8 之后，默认添加两个节点才更新 tail 节点。

当然，这样做提高性能的同时也是有代价的: 极大地增加了代码复杂度，不借助资料看懂基本看不懂，借助资料看源码也很迷。

此外，由于采用 CAS 操作没有加锁，在计算队列长度时有可能出错，因为计算过程中，有可能增加或者删除元素节点。

到此，从应用层面我们只需要记住以下几点:

- ConcurrentLinkedQueue 使用单向链表存储数据。
- ConcurrentLinkedQueue 采用 CAS 保证原子性，因此可能会导致 size 计算失误。
- ConcurrentLinkedQueue 对 tail 节点进行了优化，每添加两次元素，tail 节点移动一次，减少 CAS 操作。

5.3.2 LinkedBlockingQueue

从名字上可以知道，LinkedBlockingQueue 是链表实现的阻塞队列。既然是阻塞，那肯定是通过上锁保证原子性。

```
public class LinkedBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>,
    java.io.Serializable
```

BlockingQueue 定义了基础的 queue 方法，功能上它有点像标记接口，定义了一套阻塞方法。

LinkedBlockingQueue 有以下几个关键属性:

```
private final AtomicInteger count = new AtomicInteger();
transient Node<E> head;
private transient Node<E> last;
private final ReentrantLock takeLock = new ReentrantLock();
private final Condition notEmpty = takeLock.newCondition();
private final ReentrantLock putLock = new ReentrantLock();
private final Condition notFull = takeLock.newCondition();

public LinkedBlockingQueue(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
    last = head = new Node<E>(null);
}
```

和 ConcurrentLinkedQueue 一样保存了头节点和尾节点，同时维持一个 count 属性用于计算节点数量。此外为了线程安全创造了两个 ReentrantLock 实例，分别用来控制元素入队和出

队的原子性：

- **takeLock**: 控制同时只有一个线程可以从队列头获取元素，其他线程必须等待。
- **putLock**: 控制同时只能有一个线程可以获取锁，在队列尾部添加元素，其他线程必须等待。

notEmpty 和 notFull 是条件变量，它们内部都有一个条件队列用来存放进队和出队时被阻塞的线程。

由于条件变量 notEmpty 内部的条件队列的维护使用的是 takeLock 的锁状态管理机制，所以在调用 notEmpty 的 await 和 signal 方法前调用线程必须先获取到 takeLock 锁，否则会抛出 IllegalMonitorStateException 异常。notEmpty 内部则维护着一个条件队列，当线程获取到 takeLock 锁后调用 notEmpty 的 await 方法时，调用线程会被阻塞，然后该线程会被放到 notEmpty 内部的条件队列进行等待，直到有线程调用了 notEmpty 的 signal 方法。noFull 类似。

LinkedBlockingQueue 的原理非常简单，就是采用 ReentrantLock，这里只贴出 offer() 的部分代码：

```
public boolean offer(E e) {
    // 省略一些检查，初始化操作
    putLock.lock();
    try {
        if (count.get() == capacity)
            return false;
        enqueue(node);
        c = count.getAndIncrement();
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        putLock.unlock();
    }
    // ...
    return true;
}
```

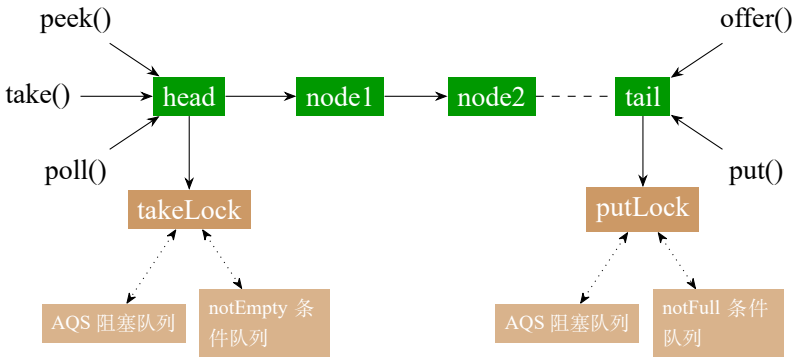


图 5.1 BlockingLinkedQueue

5.3.3 ArrayBlockingQueue

ArrayBlockingQueue 和 LinkedBlockingQueue 类似，只不过底层采用数组存储元素，这节只讲不同之处。

```
public class ArrayBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>,
    java.io.Serializable {
    final Object[] items;
    int takeIndex;
    int putIndex;
    int count;
    final ReentrantLock lock;
    @SuppressWarnings("serial")
    private final Condition notEmpty;
    @SuppressWarnings("serial")
    private final Condition notFull;
}
```

由于采用数组存储元素，ArrayBlockingQueue 是有界的，因此需要在构造函数传入队列大小参数:

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity]; // items 是存储元素的数组
    lock = new ReentrantLock(fair);  // 默认情况下，采用非公平策略
    notEmpty = lock.newCondition();
    notFull = lock.newCondition();
}
```

和写时复制以及扩容机制不同，ArrayBlockingQueue 数组的大小是不会变的，如果超出数组大小会出现错误:

```
public boolean offer(E e) {
    Objects.requireNonNull(e);
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        if (count == items.length)
            return false;
        else {
            enqueue(e);
            return true;
        }
    } finally {
        lock.unlock();
    }
}
```

保证线程安全的策略和 LinkedBlockingQueue 类似，采用锁策略，不过 ArrayBlockingQueue 只有一个锁:

```
final ReentrantLock lock;
```

所以操作，包括读写，转换都会用到 lock 锁，因此效率是比较低的，由于采用了锁机制，tail，head 都没有被 violate 修饰，因为锁可以保证内存可见性。

总而言之，在保证线程安全方面，ArrayBlockingQueue 采用了简单的直接对所有方法加锁 (ReentrantReadWriteLock) 的方式。不同于 LinkedBlockingQueue 对头部和尾部采用了不同的锁，ArrayBlockingQueue 只采用了一个锁 (两个条件变量)，效率明显不及 LinkedBlockingQueue。但由于 ArrayBlockingQueue 采用定长数组存储数据，可以实现随机查找，这方面的优势较大。

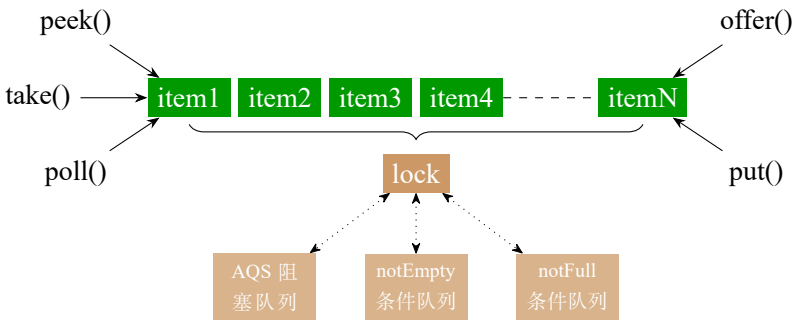


图 5.2 BlockingArrayQueue

5.3.4 PriorityBlockingQueue

PriorityBlockingQueue 是带优先级的无界阻塞队列，其内部使用平衡二叉树实现，直接遍历元素不保证有序。默认使用对象的 compareTo 方法提供比较规则。扩容机制参考 ArrayList，优先机制参考 TreeMap，阻塞机制参考 ArrayBlockingQueue。这节只讲多线程下的不同点。

```
public class PriorityBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>,
    java.io.Serializable {
    private transient Object[] queue;
    private transient int size;
    private transient Comparator<? super E> comparator;
    private final ReentrantLock lock = new ReentrantLock();
    @SuppressWarnings("serial")
    private final Condition notEmpty = lock.newCondition();
    private transient volatile int allocationSpinLock;
}
```

可以发现，PriorityBlockingQueue 种有一个 queue 数组用于存放元素，由于是无界队列，queue 是可以自动扩容的。allocationSpinLock 是一个自旋锁，通过 CAS 操作保证同时只有一个线程可以扩容队列，状态为 0 或 1，0 表示没有进行扩容，1 表示正在进行扩容。

```
public PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator) {
    if (initialCapacity < 1)
        throw new IllegalArgumentException();
    this.comparator = comparator;
    this.queue = new Object[Math.max(1, initialCapacity)]; // 默认初始大小为 11
}
```

在扩容过程中，PriorityBlockingQueue 会检查数组大小，如果小于 64，容量 +2, 大于 64 容量翻倍，扩容代码如下:

```
private void tryGrow(Object[] array, int oldCap) {
    lock.unlock(); // must release and then re-acquire main lock
    Object[] newArray = null;
    if (allocationSpinLock == 0 && ALLOCATIONSPINLOCK.compareAndSet(this, 0, 1)) {
        try {
            int growth = (oldCap < 64) ? (oldCap + 2) : (oldCap >> 1);
            int newCap = ArraysSupport.newLength(oldCap, 1, growth);
            if (queue == array)
                newArray = new Object[newCap];
        } finally {
            allocationSpinLock = 0;
        }
    }
    if (newArray == null) // back off if another thread is allocating
        Thread.yield();
    lock.lock();
    if (newArray != null && queue == array) {
        queue = newArray;
        System.arraycopy(array, 0, newArray, 0, oldCap);
    }
}
```

我们发现在进入扩容是首先会释放锁，然后使用 CAS 控制只有一个扩容可以成功。为什么要释放锁呢? 这是一个优化方案，不释放锁对功能没有影响，但是扩容是一个耗时的过程，如果不释放锁，在扩容期间就不能进行出队和入队操作，这会大大降低并发性。

此外，虽然 PriorityBlockingQueue 和 ArrayList 一样没有动态缩容机制，但是 ArrayList 提供了公有的 trimToSize() 方法用于手动缩容，而 PriorityBlockingQueue 没有提供任何缩容方法。

具体的二叉树算法这里不介绍。

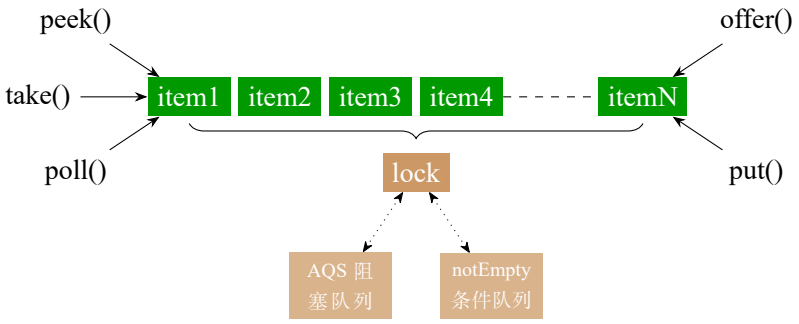


图 5.3 PriorityBlockingQueue

5.3.5 DelayQueue

DelayQueue 并发队列是一个无界阻塞延迟队列，队列中的每个元素都有个过期时间，当从队列获取元素时，只有过期元素才会出队列。队列头元素是最快要过期的元素。

```
public class DelayQueue<E extends Delayed> extends AbstractQueue<E> implements
    BlockingQueue<E> {
    private final transient ReentrantLock lock = new ReentrantLock();
    private final PriorityQueue<E> q = new PriorityQueue<E>();
    private Thread leader;
    private final Condition available = lock.newCondition();
    public DelayQueue() {}
}
```

DelayQueue 内部使用 PriorityQueue 存放数据，使用 ReentrantLock 实现线程同步。加入 DelayQueue 的元素都需要实现 Delayed 接口，由于每个元素都有一个过期时间，所以要实现获知当前元素还剩下多少时间就过期了的接口，由于内部使用优先级队列来实现，所以要实现元素之间相互比较的接口。：

```
public interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
```

具体方法上，没什么好说的，就是用 ReentrantLock 加锁然后调用 PriorityQueue 的方法。poll 和 take 方法比较特殊，会查看队列元素是否过期，过期了才取。

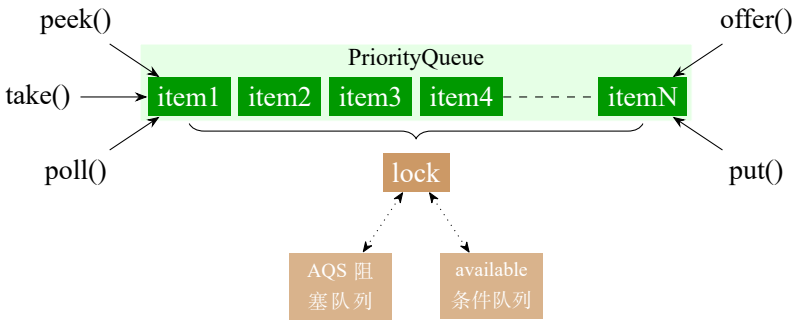


图 5.4 DelayQueue

Queue 到这里就介绍完了，对入门来说非常详细，并发容器中还有几个 Deque 其原理和对应的 Queue 类似，不再赘述。

5.4 Map

5.4.1 ConcurrentHashMap

本文默认讲 Java8 的 ConcurrentHashMap 逻辑，具体代码为 Java17。ConcurrentHashMap 的实现非常复杂，源码有 6k+ 行，因此源码解析和 Java7 的实现方案不再赘述 (写不下)。

ConcurrentHashMap 基于 HashMap 逻辑，实现了针对数组元素的细粒度锁，相比 HashTable 有更好的并发性。

```
public class ConcurrentHashMap<K,V> extends AbstractMap<K,V> implements ConcurrentMap<K,V>,
    Serializable
```

锁粒度 我们知道，HashTable 使用 synchronized 锁对象的方式解决了多线程问题，但如果以整个容器为一个资源进行锁定，那么就变为了串行操作。而根据 hash 表的特性，具有冲突的操作只会出现在同一槽位，而与其它槽位的操作互不影响。

因此可以将资源锁粒度缩小到槽位 (数组节点) 上，这样热点一分散，冲突的概率就大大降低，并发性能就能得到很好的增强。

ConcurrentHashMap 使用 JUC 包中通过直接操作内存中的对象，将比较与替换合并为一个原子操作的乐观锁形式（CAS）来进行简单的值替换操作，对于一些含有复杂逻辑的流程对 Node 节点对象使用 synchronize 进行同步。

并发扩容 ConcurrentHashMap 扩容采用多线程扩容方式提高了扩容的效率，扩容时不允许进行其他操作。在扩容过程中，ConcurrentHashMap 会被分成多端由不同的空闲线程将各段存储到新的容器中。

主要方法 put get 方法的逻辑与主要代码

在执行 put 方法时，主要使用如下逻辑:

- 由 spread 函数根据 key 获取 hash 值。
- 判断数组是否需要初始化。
- 定位到数组的 Node 节点 f。
  - 槽位为空 (f 为 null)，采用 CAS 方式添加。
  - 如果处于扩容阶段，线程协助扩容。
  - 其他情况，通过 synchronized 锁住节点，根据结构不同 (链表或红黑树) 插入新界节点。
- 链表长度达到 8 ，进行扩容。

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    // 计算 hash 值
    int hash = spread(key.hashCode());
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh; K fk; V fv;
        // 判断是否要初始化
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // 空槽位，添加节点
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value)))
                break;
        }
        // 处于扩容阶段
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        // 非空槽位，锁住对应槽位并添加新节点
        else {
            synchronized (f) {
```

```
                // 省略链表与红黑树添加节点具体操作
            }
            if (binCount != 0) {
                // 链表长度大于8，转换为红黑树
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    addCount(1L, binCount);
    return null;
}
```

get 和 hashMap 类似，没有加锁，因为 Node 元素是 volatile 修饰的:

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    // 计算 hash 值
    int h = spread(key.hashCode());
    // 数组是否为空，槽位是否为空
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        // 在数组上，直接放回
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        // 红黑树结构
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        // 链表结构
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

看样子已经完成被 CopyOnWriteArrayList 填满了。

并发集合总结

表 2.1 并发集合总结

数据结构	类/获取方式	锁类型	技术	适用情形
List	Collections.synchronizedList(List)	synchronized	锁对象	-
	CopyOnWriteriteArrayList	synchronized	写时复制	读多写少
Set	Collections.synchronizedMap(Map)	synchronized	锁对象	-
	CopyOnWriteSet	synchronized	写时复制	读多写少
Queue	ConcurrentLinkedQueue	CAS	-	默认选项
	BlockingQueue	ReentrantLock	数组或元素锁	读多写少
Map	ConcurrentHashMap	CAS+syn	锁槽位	默认选项
	HashTable	synchronized	锁对象	-

针对 Queue 和 Map 很好选择：ConcurrentLinkedQueue 和 ConcurrentHashMap，极少情况才会考虑其他选项。比较特殊的 Stack，非并发情况下使用 LinkedList，否则使用 Stack。而 List 和 Set 则需要慎重考虑:

针对 List，由于 CopyOnWriteArrayList 采用的是写时复制策略，在数据量大或者需要频繁修改数据的情况，性能非常低。因此我们首先要考虑的是也不应该采用高并发 List，能否采用其他数据结构或者将 List 放入非并发情况下。如果一定要用，则需要知道写时复制的缺陷，如果不能接受，建议派生 ArrayList 或用 Collections.synchronizedList 构建并发列表，最后才考虑 CopyOnWriteArrayList。

针对 Set，同样需要考虑写时复制的缺陷。Java 没有给出高效率的并发 List 和 Set 是有原因的，这两个数据结构在多线程中并没有其他两个常用，因此在设计之初就需要考虑是否真的有必要使用这两种数据结构的并发版本。

6 线程池化技术

6.1 ThreadPoolExecutor

线程池主要解决两个问题:

- **线程复用，提高异步任务执行性能：**如果不使用线程池，每次执行异步任务需要创建和销毁线程。
- **资源限制与线程管理：**可以对线程进行管理，限制线程个数，动态新增线程，保留线程相关的一些数据。

6.1.1 继承结构

ThreadPoolExecutor 有如下派生结构: Executor → ExecutorService → AbstractExecutorService → ThreadPoolExecutor。

```
public class ThreadPoolExecutor extends AbstractExecutorService
```

Executor 函数式接口，有一个 execute 方法，表示会执行的任务。

```
public interface Executor {
    void execute(Runnable command);
}
```

Future 同样是接口，用于调用其他线程完成好后的结果，再返回到当前线程。Future 接口中的方法如下:

- boolean cancel(): 如果等太久，是否要直接取消任务。
- boolean isCanceled(): 任务是否被取消。
- boolean isDone(): 任务是否被执行。
- V get(): 有两个，不带参数表示无穷等待执行任务，带参数则可以指定等待时间。

ExecutorService 对 Executor 接口进行了扩展，同样是一个接口。提供返回 Future 对象，终止，关闭线程等方法。

- void shutdown(): 关闭线程池，不再接受新线程，已有的等待线程继续执行。
- List<Runnable> shutdownNow(): 立即关闭线程池 (包括正在执行的)，停止所有等待的线程并返回。
- boolean isShutDown(): 判断线程池是否已关闭，只有调用了前两个方法才返回 True。
- boolean isTerminated(): 调用 shutdown 且等待线程执行完返回 True。
- boolean awaitTermination(): shutdown 之后所有线程完成或者调用了中断，返回 True，否则一直阻塞。

- Future submit(): 提交一个任务，返回的对象中包含该任务的返回值。
- List invokeAll(): 将每个线程的执行结果封装为 Future 对象并返回。
- T invokeAny(): 当集合中任意一个线程完成任务时返回，同时取消其他线程。

AbstractExecutorService 是 ExecutorService 的默认实现，主要看一下 submit 的实现:

```
public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}
public <T> Future<T> submit(Runnable task, T result) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task, result);
    execute(ftask);
    return ftask;
}
```

有两个注意点，一是 submit 本质是调用 execute，二是实现了 Runnable 和 Callable 的对象都可以被执行。

submit 与 execute 的区别就像 Runnable 和 Callable 的区别，execute 只执行，不做任何处理。submit 会有返回值，可能会抛出错误。

ThreadPoolExecutor 最常用的线程池。

在 ThreadPoolExecutor 中，有一个 ctl 用于存储信息:

```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
```

其中，ctl 的高 3 位用于表示线程状态，后 29 位用于记录线程池线程个数，默认为运行状态。线程池有如下状态:

- RUNNING: 接受新任务，处理阻塞队列里的任务。
- SHUTDOWN: 拒绝新任务，处理阻塞队列里的任务。
- STOP: 拒绝新任务，抛弃阻塞队列里的任务，中断正在处理的任务。
- TIDYING: 所有任务执行完后当前线程池活动线程数为 0，将要调用 terminated 方法。
- TERMINATED: 终止状态，terminated 方法调用之后的状态。

线程池状态转换如下:

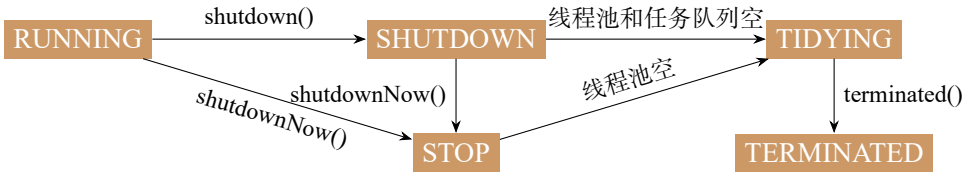


图 6.1 线程池状态转换关系

此外，还有几个重要属性:

```
private final ReentrantLock mainLock = new ReentrantLock(); // 锁 Worker 对象
private final Condition termination = mainLock.newCondition();
```

Worker 对象是对线程的封装，继承了 AQS 和 Runnable 接口，是具体承载任务的对象。它的状态如下:

- state = 0: 锁未被获取。
- state = 1: 锁已被获取。
- state = -1: 创建 Worker 时默认的状态。

new 一个线程池需要很多参数，含义如下:

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit
    unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

- corePoolSize: 核心线程数。
- maximumPoolSize: 最大线程数。
- keepAliveTime: 存活时间; 线程数量多，处于闲置状态，闲置线程能存活的最大时间。
- TimeUnit: 时间单位。
- workQueue: 用于保存等待执行任务的阻塞队列，一般为 LinkedBlockingQueue。
- threadFactory: 创建线程的工厂。
- handler: 队列满时拒绝线程，这个过程的处理方法。

Executors 工具类，类似于 Collections。提供工厂方法来创建不同类型的线程池。

- **newFixedThreadPool**

  创建一个核心线程个数和最大线程个数都为 nThreads 的线程池，并且阻塞队列长度为 Integer.MAX_VALUE。keeyAliveTime=0 说明只要线程个数比核心线程个数多并且当前空闲则回收。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS, new
        LinkedBlockingQueue<Runnable>());
}
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
    threadFactory) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS, new
```

```
        LinkedBlockingQueue<Runnable>(), threadFactory);
}
```

- **newSingleThreadExecutor**

创建一个核心线程个数和最大线程个数都为 1 的线程池，并且阻塞队列长度为 Integer.MAX_VALUE。keeyAliveTime=0 说明只要线程个数比核心线程个数多并且当前空闲则回收。

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new
            LinkedBlockingQueue<Runnable>()));
}
public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory) {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new
            LinkedBlockingQueue<Runnable>(), threadFactory));
}
```

- **newCachedThreadPool**

创建一个按需创建线程的线程池，初始线程个数为 0，最多线程个数为 Integer.MAX_VALUE，并且阻塞队列为同步队列。keeyAliveTime=60 说明只要当前线程在 60s 内空闲则回收。这个类型的特殊之处在于，加入同步队列的任务会被马上执行，同步队列里面最多只有一个任务。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS, new
        SynchronousQueue<Runnable>());
}
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS, new
        SynchronousQueue<Runnable>(), threadFactory);
}
```

此外，Executors 还提供了几个线程工厂，比较简单，有兴趣自己看源码。

6.1.2 源码分析

execute ThreadPoolExecutor 实现机制是一个生产者-消费者模型，execute 方法的作用是提交任务 command 到现称池进行执行。

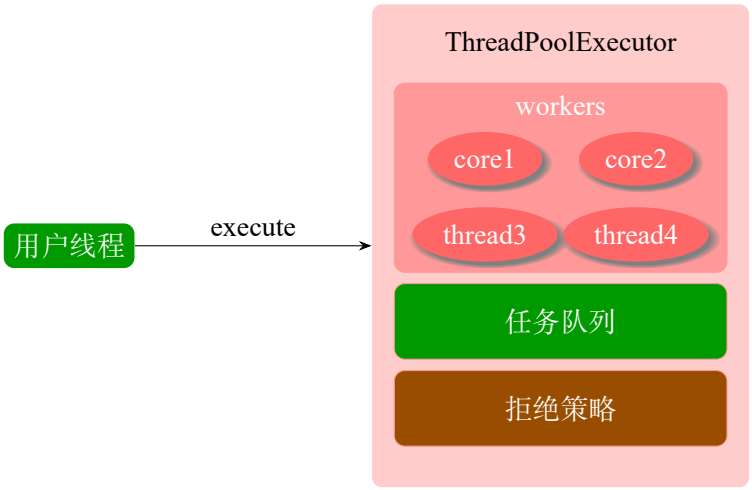


图 6.2 execute 方法

具体的 execute 代码如下:

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    int c = ctl.get();
    // 判断当前线程个数，小于核心线程数则开启新线程运行
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // 线程处于 RUNNING 态，将任务添加到阻塞队列
    if (isRunning(c) && workQueue.offer(command)) {
        // 二次检查
        int recheck = ctl.get();
        // 如果线程池状态不是 RUNNING 则从队列中删除任务，执行拒绝策略
        if (!isRunning(recheck) && remove(command))
            reject(command);
        // 如果线程池为空，添加线程
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    // 增加线程失败，则拒绝
    else if (!addWorker(command, false))
        reject(command);
}
```

简言之，做了如下判断:

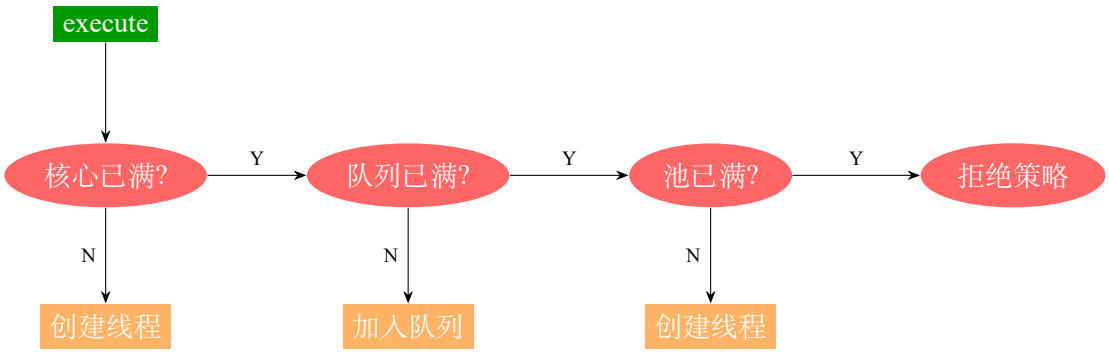


图 6.3 execute 执行顺序

addWorker 方法比较复杂，总的来说做了这几件事:

- 检查队列状态 (RUNNING)，通过循环 CAS 增加线程个数。
- 将任务封装为 worker，获取 mainLock 锁并执行。

来看看内部类 Worker 的构造方法:

```
Worker(Runnable firstTask) {
    setState(-1);
    this.firstTask = firstTask;
    this.thread = getThreadFactory().newThread(this);
}
```

这里为什么要将状态设置为 -1 呢，因为如果状态 >= 0，当其他线程调用了线程池的 shutdownNow 时，Worker 会被中断。在 runWorker 中会在 unlock 后将 Worker 状态设置为 0。在初始化阶段，Worker 并没有执行所以没有中断必要。

showdown 调用 shutdown 方法后，线程池就不会再接受新的任务了，但是工作队列里面的任务还是要执行的。该方法会立刻返回，并不等待队列任务完成再返回。

```
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 权限检查
        checkShutdownAccess();
        // 设置当前线程池状态为 SHUTDOWN
        advanceRunState(SHUTDOWN);
        // 设置中断标志
        interruptIdleWorkers();
        onShutdown();
    } finally {
        mainLock.unlock();
    }
    // 尝试将状态变为TERMINATED
    tryTerminate();
}
```

shutdownNow 调用 shutdownNow 方法后，线程池就不会再接受新的任务了，并且会丢弃工作队列里面的任务，正在执行的任务会被中断，该方法会立刻返回，并不等待激活的任务执行完成。返回值为这时候队列里面被丢弃的任务列表。

public List<Runnable> shutdownNow() List<Runnable> tasks; final ReentrantLock mainLock = this.mainLock; mainLock.lock(); try // 权限检查 checkShutdownAccess(); // 设置当前线程池状态为 SHUTDOWN advanceRunState(STOP); // 设置中断标志 interruptIdleWorkers(); // 将队列任务移动到 task 中 tasks = drainQueue(); finally mainLock.unlock(); tryTerminate(); return tasks;

6.2 ScheduledThreadPoolExecutor

ScheduledThreadPoolExecutor 可以在指定一定延迟时间后或者定时进行任务调度执行的线程池。

ScheduledThreadPoolExecutor 继承自 ThreadPoolExecutor 并实现了 ScheduledExecutorService 接口。线程池队列是 DelayedWorkQueue，其和 DelayedQueue 类似，是一个延迟队列。

```
public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor implements
    ScheduledExecutorService
```

ScheduledExecutorService 定义了一系列 schedule 方法，表示一段时候后执行。

看一下 ScheduledThreadPoolExecutor 参数最多的构造方法:

```
public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory,
    RejectedExecutionHandler handler) {
    super(corePoolSize, Integer.MAX_VALUE, DEFAULT_KEEPALIVE_MILLIS, MILLISECONDS, new
        DelayedWorkQueue(), threadFactory, handler);
}
```

可以发现，它本质上是调用了 ThreadPoolExecutor 的构造方法，默认使用 DelayedWorkQueue 队列。

此外，它给出了更复杂的线程池状态:

- NEW: 初始化。
- COMPLETING: 执行中。
- NORMAL: 正常运行结束。
- EXCEPTIONAL: 运行中异常。
- CANCELED: 任务取消。
- INTERRUPTING: 任务正在被中断。
- INTERRUPTED: 任务已经被中断。

ScheduledThreadPoolExecutor 的 execute 方法被重写，本质上是调用 schedule 方法:

```
public void execute(Runnable command) {
    schedule(command, 0, NANOSECONDS);
}
```

```
public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit) {
    if (command == null || unit == null)
        throw new NullPointerException();
    RunnableScheduledFuture<Void> t = decorateTask(command,
        new ScheduledFutureTask<Void>(command, null, triggerTime(delay, unit),
            sequencer.getAndIncrement()));
    delayedExecute(t);
    return t;
}
```

7 线程同步器

7.1 CountDownLatch

在日常开发中经常会遇到需要在主线程中开启多个线程去并行执行任务，并且主线程需要等待所有子线程执行完毕后再进行汇总的场景。在 CountDownLatch 出现之前一般都使用线程的 join() 方法来实现这一点，但是 join 方法不够灵活，不能够满足不同场景的需要，所以 JDK 开发组提供了 CountDownLatch 这个类。

CountDownLatch 的实现很简单，啥都没继承，有一个内部类 Syn 继承自 AQS:

```
private static final class Sync extends AbstractQueuedSynchronizer {
    private static final long serialVersionUID = 4982264981922014374L;
    Sync(int count) {
        setState(count);
    }
    int getCount() {
        return getState();
    }
    protected int tryAcquireShared(int acquires) {
        return (getState() == 0) ? 1 : -1;
    }
    protected boolean tryReleaseShared(int releases) {
        for (;;) {
            int c = getState();
            if (c == 0)
                return false;
            int nextc = c - 1;
            if (compareAndSetState(c, nextc))
                return nextc == 0;
        }
    }
}
```

可以发现非常简单，state 为 0 可以获取，再看一下 CountDownLatch 的构造方法:

```
public CountDownLatch(int count) {
    if (count < 0) throw new IllegalArgumentException("count < 0");
    this.sync = new Sync(count);
}
```

所以本质上就是传一个数字给 CountDownLatch，计数到 0 允许允许，其他的方法如下:

```
public void await() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}
public boolean await(long timeout, TimeUnit unit)
    throws InterruptedException {
    return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
}
public void countDown() {
```

```
    sync.releaseShared(1);
}
public long getCount() {
    return sync.getCount();
}
```

CountDownLatch 一般的使用方式: 初始化，设置 state 值 → 调用 await() 阻塞线程 → 线程执行完，调用 countDown() → state 值降到 0，await() 线程被唤醒。

7.2 CyclicBarrier

CountDownLatch 简化了 join 方法，但 CountDownLatch 的计数器是一次性的，也就是等到计数器值变为 0 后，再调用 CountDownLatch 的 await 和 countdown 方法都会立刻返回，这就起不到线程同步的效果了。为了重用计数器，JDK 提供了 CyclicBarrier。

从字面意思理解，CyclicBarrier 是回环屏障的意思，它可以让一组线程全部达到一个状态后再全部同时执行。

- 回环: 当所有等待线程执行完毕，并重置 CyclicBarrier 的状态后它可以被重用。
- 屏障: 线程调用 await 方法后就会被阻塞，这个阻塞点就称为屏障点，等所有线程都调用了 await 方法后，线程们就会冲破屏障，继续向下运行。

CyclicBarrier 的几个重要个属性如下:

```
public class CyclicBarrier {
    private final ReentrantLock lock = new ReentrantLock();
    private final Condition trip = lock.newCondition();
    private final int parties;
    private final Runnable barrierCommand;
    private Generation generation = new Generation();
    private int count;

    public CyclicBarrier(int parties, Runnable barrierAction) {
        if (parties <= 0) throw new IllegalArgumentException();
        this.parties = parties;
        this.count = parties;
        this.barrierCommand = barrierAction;
    }
    public CyclicBarrier(int parties) {
        this(parties, null);
    }
}
```

可以看到，CyclicBarrier 基于独占锁实现，本质上还是基于 AQS 的。在构造函数中，parties 用于记录线程个数，标识表示多少线程调用 await 后，所有线程才会冲破屏障继续往下运行。

count 一开始等于 parties，每当有线程调用 await 方法就递减 1，当 count 为 0 时就表示所有线程都到了屏障点。维护两个变量的原因是为了复用，parties 用于记录线程总数，count 用于计数。

此外还有一个 barrierAction 表示屏障被打破时需要运行的方法。

```
public int await() throws InterruptedException, BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
public int await(long timeout, TimeUnit unit) throws InterruptedException,
    BrokenBarrierException, TimeoutException {
    return dowait(true, unit.toNanos(timeout));
}
```

当线程调用 CyclicBarrier 的 await 系方法时会被阻塞，直到满足下面条件之一才会返回:

- parties 个线程都调用了 await() 方法，即线程都到了屏障点。
- 其他线程调用了当前线程的 interrupt() 方法中断了当前线程，抛出 InterruptedException 异常。
- 屏障点关联的 Generation 对象的 broken 标志被设置为 true 时，抛出 BrokenBarrierException 异常。

```
private int dowait(boolean timed, long nanos) throws InterruptedException,
    BrokenBarrierException, TimeoutException {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        ...... // 省略一些检查
        int index = --count;
        // 到达屏障点
        if (index == 0) { // tripped
            // 执行方法
            Runnable command = barrierCommand;
            if (command != null) {
                try {
                    command.run();
                } catch (Throwable ex) {
                    breakBarrier();
                    throw ex;
                }
            }
            // 开始下一次回环
            nextGeneration();
            return 0;
        }
        // 没到屏障点
        for (;;) {
            try {
                // 没有设置超时时间
                if (!timed)
                    trip.await();
                // 设置了超时时间
```

```
                else if (nanos > 0L)
                    nanos = trip.awaitNanos(nanos);
            } catch (InterruptedException ie) {
                if (g == generation && ! g.broken) {
                    breakBarrier(); // 破坏屏障，开启下一次循环
                    throw ie;
                } else {
                    Thread.currentThread().interrupt();
                }
            }
            .....  // 省略一些检查
        }
    } finally {
        lock.unlock();
    }
}
private void nextGeneration() {
    trip.signalAll();
    count = parties;
    generation = new Generation();
}
private void breakBarrier() {
    generation.broken = true;
    count = parties;
    trip.signalAll();
}
```

CyclicBarrier 一般的使用方式: 初始化，设置 parties 和 barrierAction → 调用 await() 阻塞线程 → 阻塞线程数量到达 parties，自动突破屏障执行 barrierAction。

7.3 Semaphore

Semaphore 信号量内部的计数器是递增的，并且在一开始初始化 Semaphore 时可以指定一个初始值，但是并不需要知道需要同步的线程个数，而是在需要同步的地方调用 acquire 方法时指定需要同步的线程个数。

```
public class Semaphore implements java.io.Serializable {
    private final Sync sync;
    public Semaphore(int permits) {
        sync = new NonfairSync(permits);
    }
    public Semaphore(int permits, boolean fair) {
        sync = fair ? new FairSync(permits) : new NonfairSync(permits);
    }
}
```

Semaphore 还是使用 AQS 实现的。Sync 只是对 AQS 的一个修饰。permits 用于设置初始 state 值。

```java
public void acquire() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}
public void acquire(int permits) throws InterruptedException {
    if (permits < 0) throw new IllegalArgumentException();
    sync.acquireSharedInterruptibly(permits);
}
```

acquire 系方法用于获取信号量资源，当信号量资源达到需求时能正确返回，否则会阻塞。Semaphore 默认采用非公平策略，即多个线程获取信号量非公平，也可以指定为公平策略。

```java
public void release() {
    sync.releaseShared(1);
}
public void release(int permits) {
    if (permits < 0) throw new IllegalArgumentException();
    sync.releaseShared(permits);
}
```

release 系方法的作用是把当前 Semaphore 对象的信号量值增加，如果当前有线程因为调用 aquire 方法被阻塞而被放入了 AQS 的阻塞队列，则会根据策略选择一个信号量个数能被满足的线程进行激活，激活的线程会尝试获取刚增加的信号量。

Semaphore 的一般使用方法: 初始化 Semaphore 对象 → 线程调用 release 系方法增加信号量 → 线程调用 acquire 系方法阻塞等待信号量满足需求 → 信号量满足需求，acquire 线程继续执行。