

JUC基础

创建线程的三种方法

1. Thread

- 继承Thread类，重写 `run()` 方法，调用 `start()` 方法启动线程

2. Runnable

- 实现Runnable接口，重写 `run()` 方法，将Runnable对象作为参数传递给Thread对象，调用 `start()` 方法启动线程

3. FutureTask

- 实现Callable接口，重写 `call()` 方法，然后创建FutureTask对象，参数为Callable对象；然后创建Thread对象，参数为FutureTask对象（因为FutureTask继承了Runnable接口），调用 `start()` 方法启动线程

对线程安全的理解

1. 原子性

- 确保当某个线程修改共享变量时，没有其他线程可以同时修改这个变量，即这个操作是不可分割的。

2. 可见性

- 确保一个线程对共享变量的修改可以立即被其他线程看到。

3. 活跃性

- 要确保线程不会因为死锁、饥饿、活锁等问题导致无法继续执行。

进程和线程的区别

进程：是操作系统分配资源的最小单位

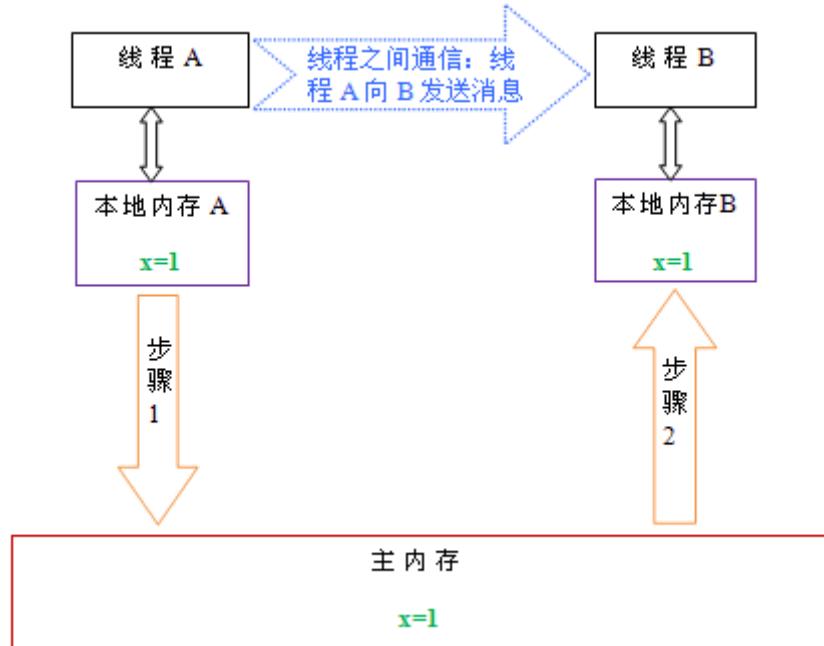
线程：是操作系统进行调度的最小单位。多个线程可以共享同一个进程的资源，如内存；每个线程都有自己独立的栈和寄存器。

线程的共享内存

Java中采取共享内存的方式实现线程之间的通信。

线程 A 与线程 B 之间如要通信的话，必须要经历下面 2 个步骤：

- 线程 A 把本地内存 A 中的共享变量副本刷新到主内存中。
- 线程 B 到主内存中读取线程 A 刷新过的共享变量，再同步到自己的共享变量副本中。

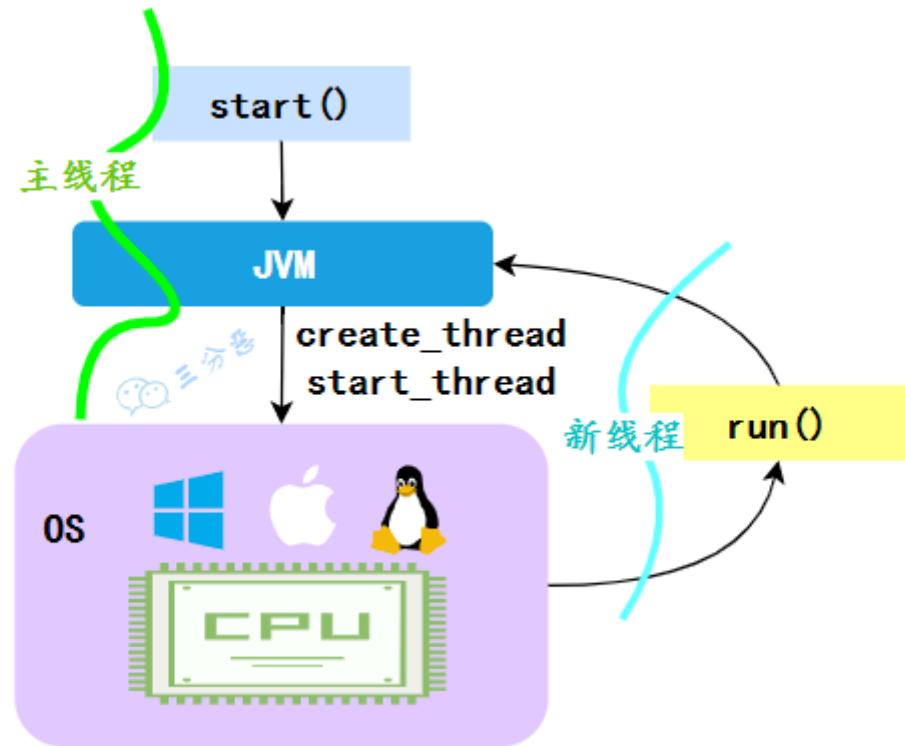


启动一个Java程序里面有哪些线程

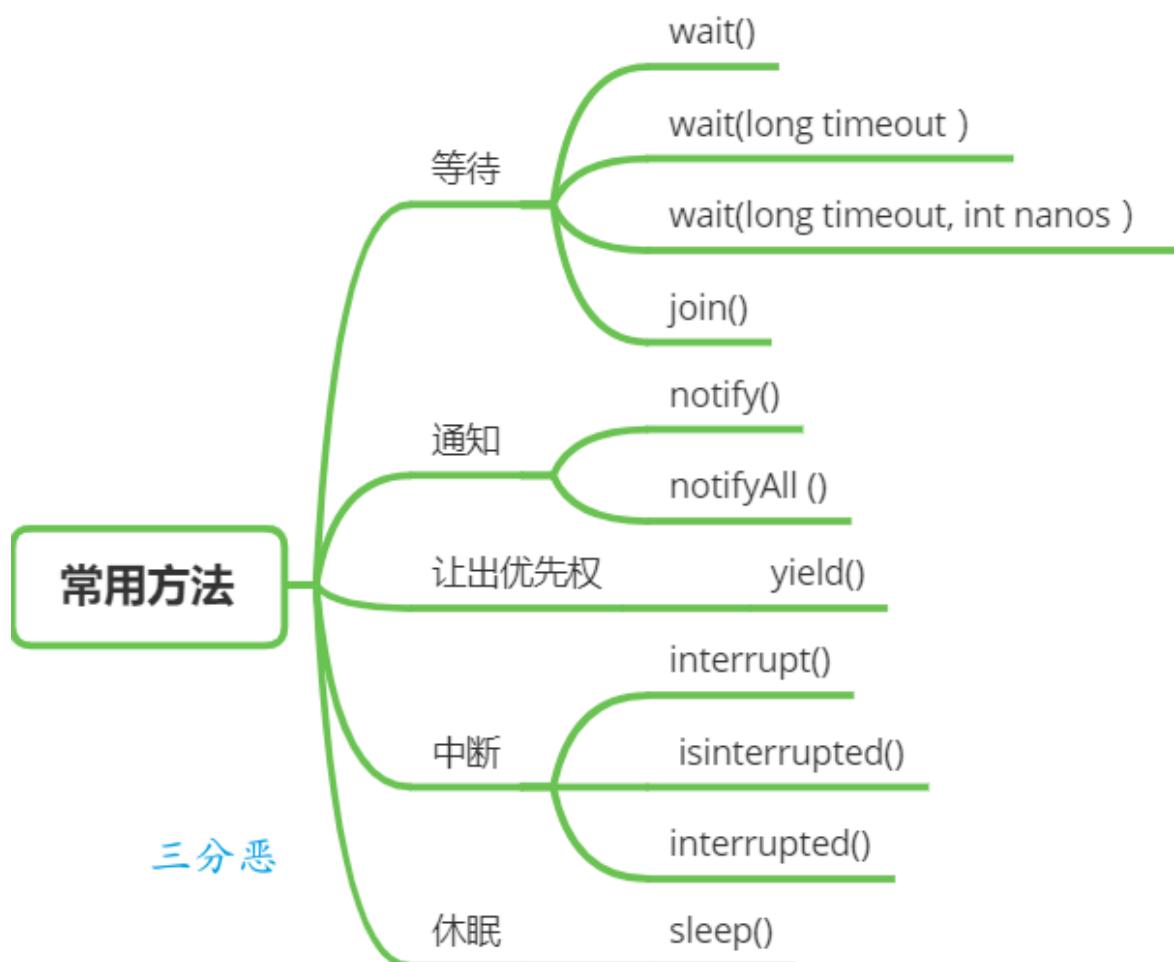
1. **main主线程**: 程序开始执行的入口
2. **垃圾回收线程**: 后台线程，负责回收不再使用的对象
3. **编译器线程**: 在及时编译中 (JIT)，负责把一部分热点代码编译后放到 codeCache 中，以提升程序的执行效率。

start()和run()的区别

- 当调用 `start()` 方法时，会启动一个新的线程，并让这个新线程调用 `run()` 方法。
- 如果直接调用 `run()` 方法，那么 `run()` 方法就在当前线程中运行，没有新的线程被创建，也就没有实现多线程的效果。



线程有哪些常用的调度方法



`yield()`: Thread 类中的静态方法，当一个线程调用 `yield` 方法时，实际是在暗示线程调度器，当前线程请求让出自己的 CPU (不会释放锁)，但是线程调度器可能会“装看不见”忽略这个暗示。

interrupt()和stop()的区别

`interrupt()`

- 用于请求一个线程中断（设置中断标志为 `true`），并不强制停止线程的运行。

`stop()`

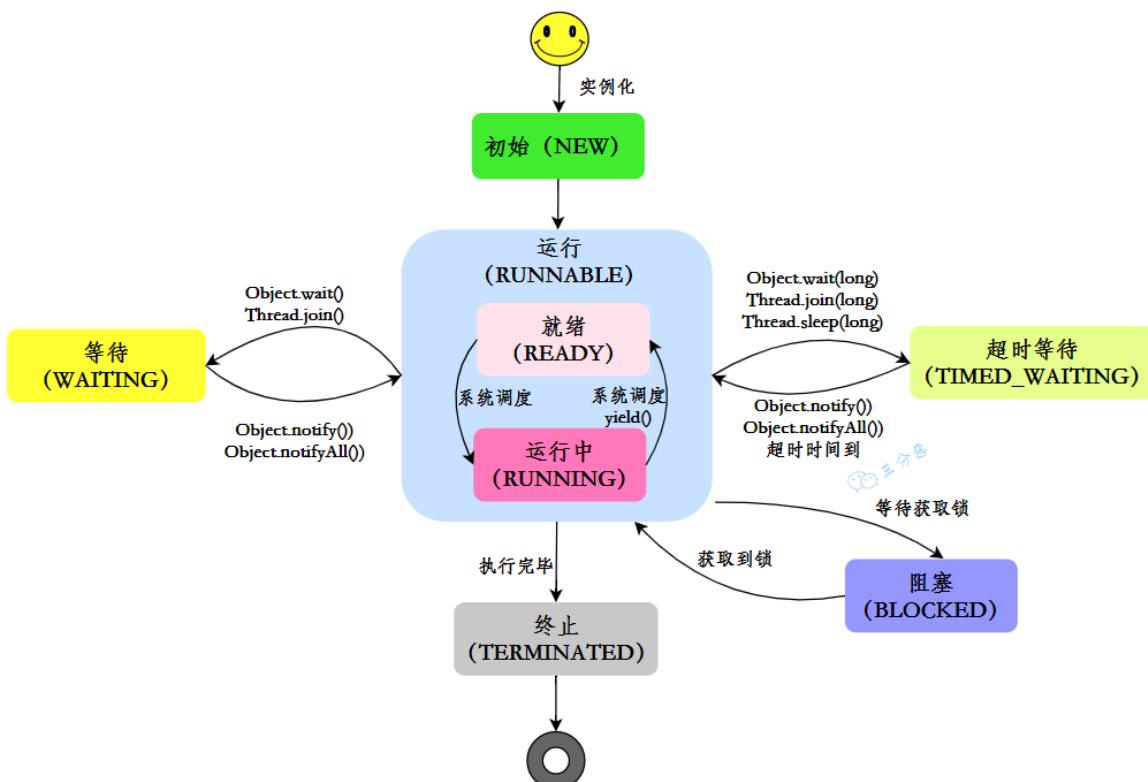
- 强制停止一个线程的运行，无论线程当前处于什么状态。
- 已废弃，不安全

线程状态

操作系统层面（5种状态）

- 初始状态
- 可运行状态
- 运行状态
- 阻塞状态
- 终止状态

Java层面（6种状态）



- NEW
- RUNNABLE：包含操作系统的【可运行状态】、【运行状态】和【阻塞状态】
- BLOCKED：如没抢到`synchronized`的锁
- WAITING：`join()`、`wait()`，需要其他线程显式唤醒

- TIMED_WAITING: wait(1000)、sleep(1000)
- TERMINATED

什么是线程上下文切换

上下文切换：CPU 资源的分配采用了时间片轮转也就是给每个线程分配一个时间片，线程在时间片内占用 CPU 执行任务。当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

什么是守护线程

Java 中线程分为两类：**守护线程、用户线程**

- 守护线程的生命周期间接依赖用户线程，如果所有用户线程都结束了，守护线程会自动终止，JVM 退出。守护线程的生命周期完全依赖于 JVM 的运行状态。
- 守护线程一般用于后台服务，比如垃圾回收线程、线程池管理线程等。

线程之间有哪些通信方式

1. 使用共享对象

- volatile
- synchronized

2. 使用 `wait()` 和 `notify()`

3. 使用 `Exchanger`

- 一个线程调用 `exchange()` 方法，将数据传递给另一个线程，同时接收另一个线程的数据。

4. 使用 `CompletableFuture`

sleep() 和 wait() 的区别

锁行为不同

- 如果一个线程在持有某个对象的锁时调用了 sleep，它在睡眠期间仍然会持有这个锁。
- 当线程执行 wait 方法时，必须拥有锁，它会释放它持有的那个对象的锁，这使得其他线程可以有机会获取该对象的锁

使用条件不同

- `sleep()` 方法可以在任何地方被调用。
- `wait()` 方法必须在同步代码块或同步方法中被调用

唤醒方式不同

- 调用 sleep 方法后，线程会进入 TIMED_WAITING 状态（定时等待状态），即在指定的时间内暂停执行。当指定的时间结束后，线程会自动恢复到 RUNNABLE 状态（就绪状态），等待 CPU 调度再次执行。

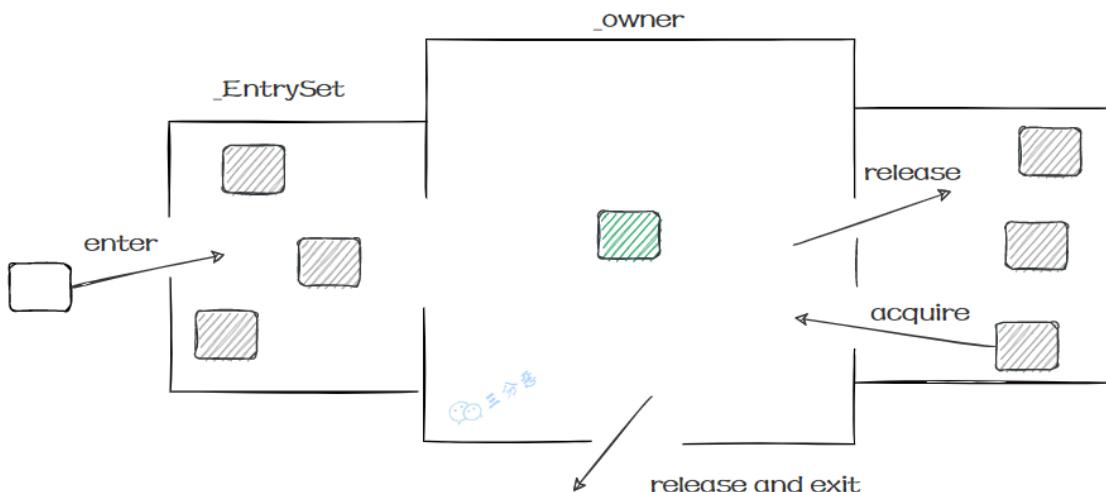
- 调用 `wait` 方法后，线程会进入 WAITING 状态（无限期等待状态），直到有其他线程在同一对象上调用 `notify` 或 `notifyAll`，线程才会从 WAITING 状态转变为 RUNNABLE 状态，准备再次获得 CPU 的执行权。

锁

Synchronized锁住的是什么

`synchronized`是基于**Monitor**实现的。

实例对象结构里有对象头，对象头里面有一块结构叫 Mark Word，Mark Word 指针指向了**Monitor**。



EntrySet指的是 等待获取锁的线程队列。当线程尝试获取锁但失败时，它们会进入EntrySet进行排队，等待锁的释放。

WaitSet指的是 线程调用 `Object.wait()` 之后进入的等待队列。这些线程正在等待某个线程调用 `notify()` 或 `notifyAll()` 唤醒它们。

Synchronized会牵扯到操作系统层面吗

`Synchronized` 升级为重量级锁时，依赖于操作系统的互斥量（mutex）来实现，mutex 用于保证任何给定时间内，只有一个线程可以执行某一段特定的代码段

Synchronized怎么保证可见性、有序性、可重入

可见性

- 线程加锁前，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值
- 线程加锁后，其它线程无法获取主内存中的共享变量
- 线程解锁前，必须把共享变量的最新值刷新到主内存中

有序性

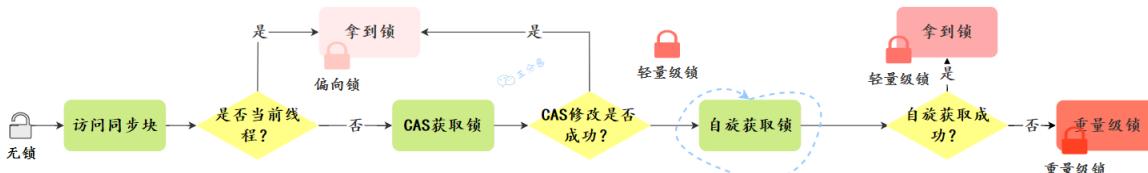
- synchronized 保证顺序性是指的将并发执行变成了串行，同一时刻代码是单线程执行的，但并不能保证内部指令重排序问题
- 因为 as-if-serial 语义的存在，单线程的程序能保证最终结果是有序的，但是不保证不会指令重排

可重入

- synchronized 之所以支持可重入，是因为 Java 的对象头包含了一个 Mark Word，用于存储对象的状态，包括锁信息。
 - 当一个线程获取对象锁时，JVM 会将该线程的 ID 写入 Mark Word，并将锁计数器设为 1。如果一个线程尝试再次获取已经持有的锁，JVM 会检查 Mark Word 中的线程 ID。如果 ID 匹配，表示的是同一个线程，锁计数器递增。
- 当线程退出同步块时，锁计数器递减。如果计数器值为零，JVM 将锁标记为未持有状态，并清除线程 ID 信息。

Synchronized优化原理

Mark Word 变化示意图						
锁状态	61 bit				偏向锁标记	锁类型
	1 bit	2 bit				
无锁	未使用 25 bit	HashCode 32 bit	未使用 1bit	分代年龄 4bit	0	01
偏向锁	当前线程指针 54 bit	Epoch 2 bit	未使用 1bit	分代年龄 4bit	1	01
轻量锁	指向栈中锁记录的指针 62 bit				三分离	00
重量级锁	指向重量级锁的指针 62 bit					10



1. 轻量级锁

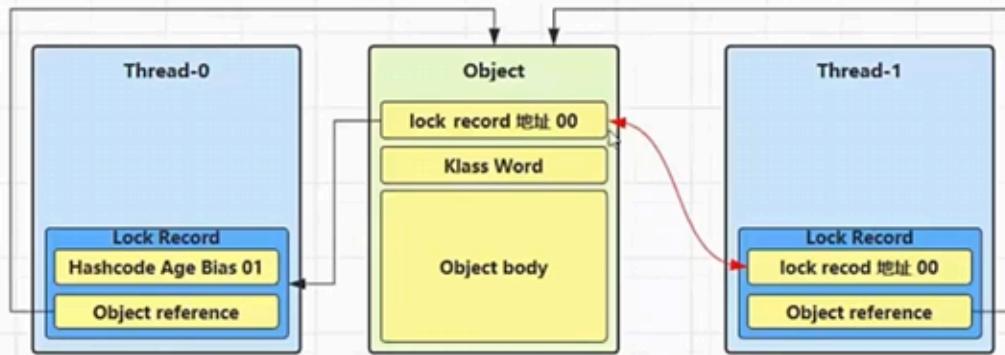
2. 锁膨胀

如果在尝试加轻量级锁的过程中，CAS 操作无法成功，这时一种情况就是有其它线程为此对象加上了轻量级锁（有竞争），这时需要进行锁膨胀，将轻量级锁变为重量级锁。

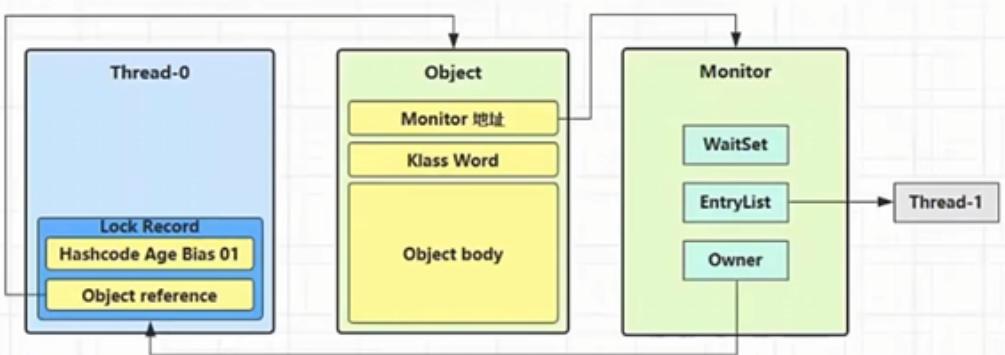
```
static Object obj = new Object();
public static void method1() {
    synchronized( obj ) {
        // 同步块
    }
}
```

java

- 当 Thread-1 进行轻量级加锁时，Thread-0 已经对该对象加了轻量级锁



- 这时 Thread-1 加轻量级锁失败，进入锁膨胀流程
 - 即为 Object 对象申请 Monitor 锁，让 Object 指向重量级锁地址
 - 然后自己进入 Monitor 的 EntryList BLOCKED



- 当 Thread-0 退出同步块解锁时，使用 cas 将 Mark Word 的值恢复给对象头，失败。这时会进入重量级解锁流程，即按照 Monitor 地址找到 Monitor 对象，设置 Owner 为 null，唤醒 EntryList 中 BLOCKED 线程

3. 自旋优化

当线程尝试获取轻量级锁失败时，它会进行自旋，即循环检查锁是否可用，以避免立即进入阻塞状态。如果当前线程自旋成功（即持锁线程已经退出同步块，释放了锁），这时当前线程就可以避免阻塞。

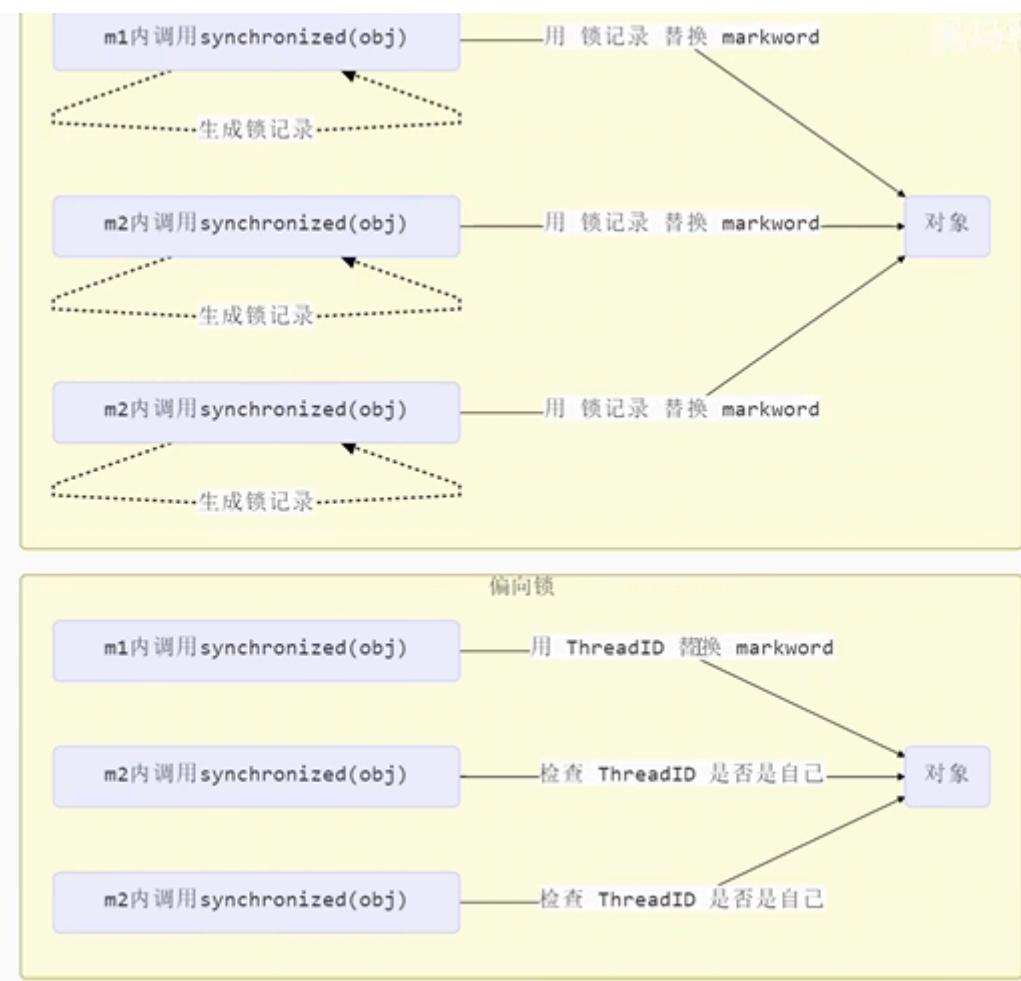
自旋重试成功的情况

线程 1 (cpu 1 上)	对象 Mark	线程 2 (cpu 2 上)
-	10 (重量锁)	I
访问同步块, 获取 monitor	10 (重量锁) 重量锁指针	-
成功 (加锁)	10 (重量锁) 重量锁指针	-
执行同步块	10 (重量锁) 重量锁指针	-
执行同步块	10 (重量锁) 重量锁指针	访问同步块, 获取 monitor
执行同步块	10 (重量锁) 重量锁指针	自旋重试
执行完毕	10 (重量锁) 重量锁指针	自旋重试
成功 (解锁)	01 (无锁)	自旋重试
-	10 (重量锁) 重量锁指针	成功 (加锁)
-	10 (重量锁) 重量锁指针	执行同步块
-

自旋重试失败的情况

线程 1 (cpu 1 上)	对象 Mark	线程 2 (cpu 2 上)
-	10 I (重量锁)	-
访问同步块, 获取 monitor	10 (重量锁) 重量锁指针	-
成功 (加锁)	10 (重量锁) 重量锁指针	-
执行同步块	10 (重量锁) 重量锁指针	-
执行同步块	10 (重量锁) 重量锁指针	访问同步块, 获取 monitor
执行同步块	10 (重量锁) 重量锁指针	自旋重试
执行同步块	10 (重量锁) 重量锁指针	自旋重试
执行同步块	10 (重量锁) 重量锁指针	自旋重试
执行同步块	10 (重量锁) 重量锁指针	阻塞
-

4. 偏向锁



偏向锁撤销

- 在偏向锁状态下，调用 hashCode() 方法会撤销偏向锁，改为轻量级锁
- 当有其它线程使用偏向锁对象时，会将偏向锁升级为轻量级锁
- 调用 wait / notify

(如果另一个线程尝试获取这个已被偏向的锁，JVM 会检查当前持有偏向锁的线程是否活跃。如果持有偏向锁的线程不活跃，则可以将锁重偏向至新的线程；如果持有偏向锁的线程还活跃，则需要撤销偏向锁，升级为轻量级锁。)

批量重偏向

当撤销偏向锁阈值超过20次后，JVM会认为自己偏向错了，于是会在给这些对象加锁时重新偏向至加锁线程

批量撤销

当撤销偏向锁阈值超过40次时，JVM会认为不该偏向，于是整个类的所有对象都会变成不可偏向的。

5. 锁消除

如果只有一个线程可能用到锁，JIT即时编译会对代码进行优化，去掉锁。

6. 锁粗化

如果 JVM 检测到一系列连续的锁操作实际上是在单一线程中完成的，则会将多个锁操作合并为一个更大范围的锁操作，这可以减少锁请求的次数。

ReentrantLock是什么

可中断 (Synchronized不行)

```
1 | `lock.lockInterruptibly()`
2 |
3 | 如果没有竞争此方法就获取lock对象锁
4 |
5 | 如果有竞争就进入阻塞队列，可以被其它线程用interrupt方法打断
```

可以设置超时时间 (Synchronized不行)

```
1 | `lock.tryLock(1, TimeUnit.SECONDS)`
2 |
3 | 返回值是boolean类型，获取锁成功为true
```

可以设置公平锁 (Synchronized不行)

支持多个条件变量 (Synchronized不行)

```
1 Condition A = ReentrantLock.newCondition();
2
3 A.await();
4
5 A.signal();
```

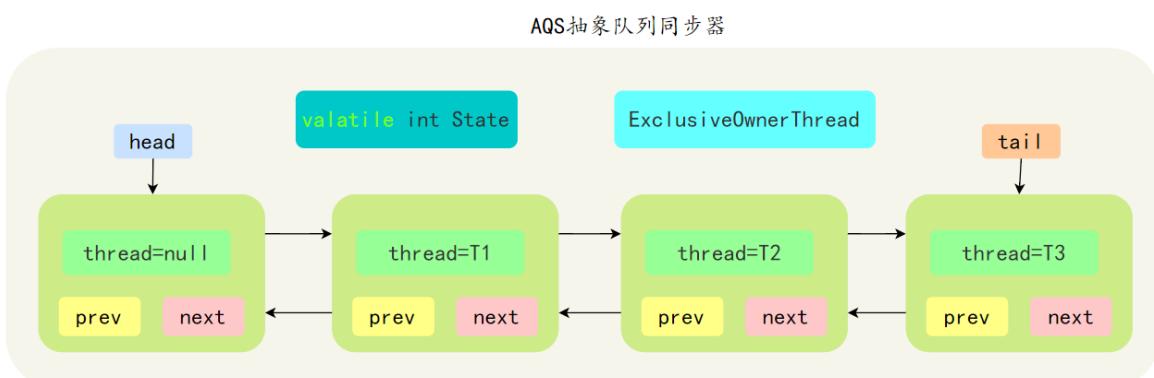
与synchronized一样，都支持可重入

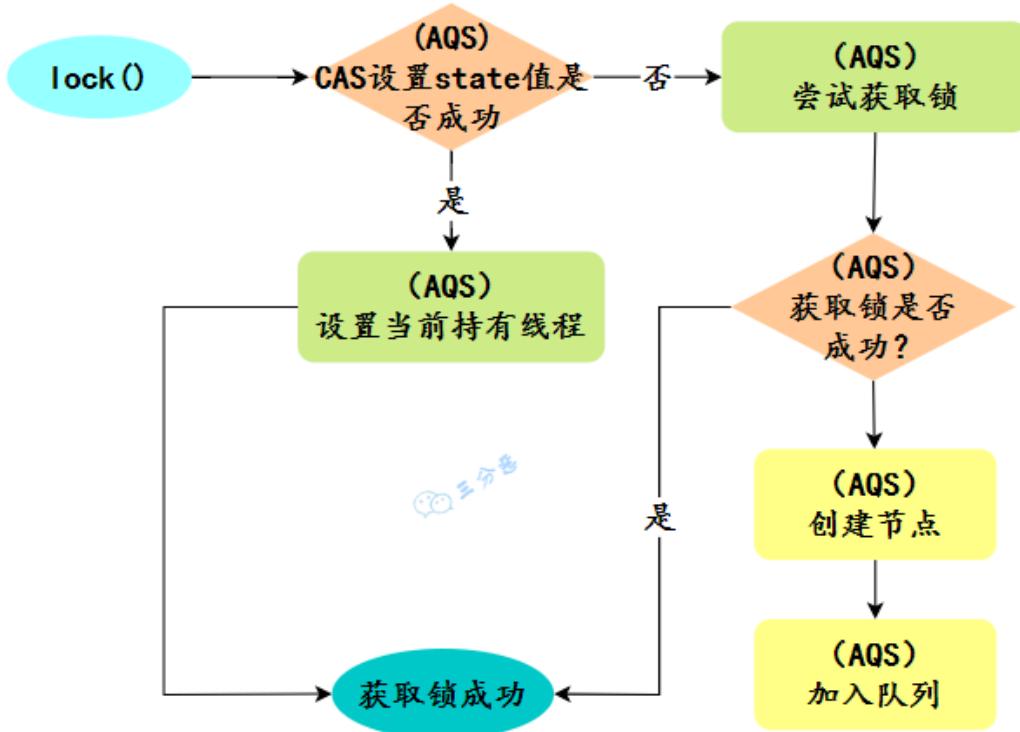
ReentrantLock和Synchronized的区别

区别	synchronized	ReentrantLock
锁实现机制	对象头监视器模式	依赖AQS
灵活性	不灵活	支持响应中断、超时、尝试获取锁
释放锁形式	自动释放锁	显式调用unlock()
支持锁类型	非公平锁	公平锁&非公平锁
条件队列	单条件队列	多个条件队列
可重入支持	支持	支持

AQS了解多少

AQS(Abstract Queue Synchronizer)的思想是，如果被请求的共享资源空闲，则当前线程能够成功获取资源；否则，它将进入一个等待队列，当有其他线程释放资源时，系统会挑选等待队列中的一个线程，赋予其资源。





CAS了解多少

CAS是一种乐观锁的实现方式，全称为Compare and Swap，是一种无锁的原子操作。

CAS是乐观锁，线程执行的时候不会加锁，它会假设此时没有冲突，然后完成某项操作；如果因为冲突失败了就重试，直到成功为止。

CAS怎么保证数据原子性

为了保证CAS的原子性，CPU 提供了两种实现方式

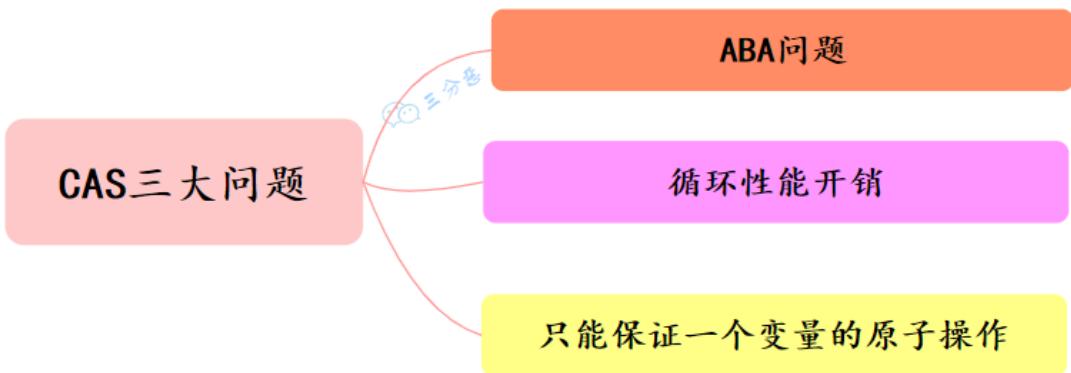
1. 总线锁定

- 通过锁定 CPU 的总线，禁止其他 CPU 或设备访问内存。

2. 缓存锁定（优先）

- 当多个 CPU 操作同一块内存地址时，如果该内存地址已经被缓存到某个 CPU 的缓存中，缓存锁定机制会锁定该缓存行，防止其他 CPU 对这块内存进行修改。
- MESI协议

CAS有什么问题？如何解决



ABA问题

如果一个位置的值原来是 A，后来被改为 B，再后来又被改回 A，那么进行 CAS 操作的线程将无法知晓该位置的值在此期间已经被修改过。

可以使用版本号/时间戳的方式来解决 ABA 问题。

比如说，每次变量更新时，不仅更新变量的值，还更新一个版本号。CAS 操作时不仅要求值匹配，还要求版本号匹配。

循环性能开销

自旋 CAS，如果一直循环执行，一直不成功，会给 CPU 带来非常大的执行开销。

怎么解决循环性能开销问题？

在 Java 中，很多使用自旋 CAS 的地方，会有一个自旋次数的限制，超过一定次数，就停止自旋。

只能保证一个变量的原子操作

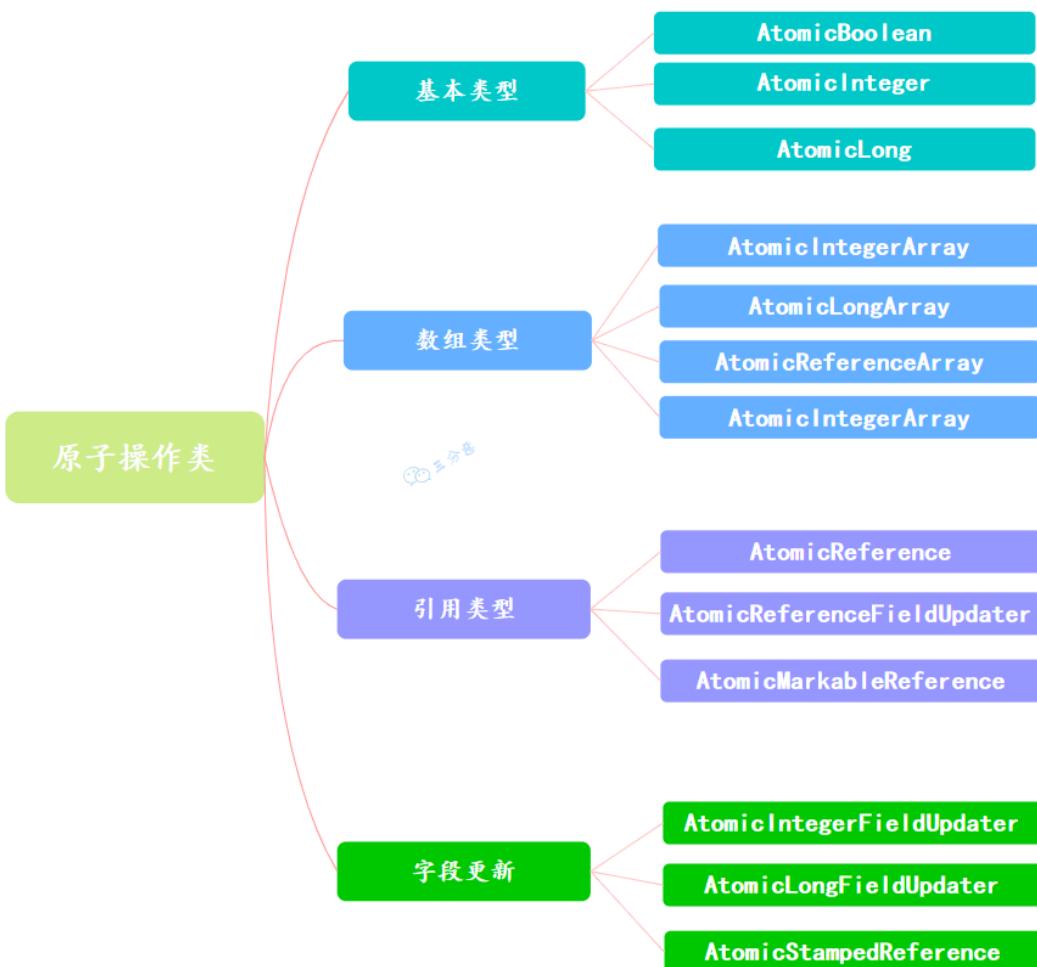
CAS 保证的是对一个变量执行操作的原子性，如果对多个变量操作时，CAS 目前无法直接保证操作的原子性的。

```
1 class Account {  
2     AtomicInteger balance1 = new AtomicInteger(100);  
3     AtomicInteger balance2 = new AtomicInteger(200);  
4  
5     void transfer(int amount) {  
6         // 🚨 不是原子操作！两个 CAS 之间可能被其他线程打断  
7         balance1.getAndAdd(-amount); // 从账户 1 扣钱  
8  
9         // 在两个变量的 CAS 之间另一个线程读，此时账户 1 扣的钱还没加到账户 2 中  
10  
11         balance2.getAndAdd(amount); // 向账户 2 加钱  
12     }  
13 }  
14 }
```

怎么解决只能保证一个变量的原子操作问题?

- 可以考虑改用锁来保证操作的原子性
- 可以考虑合并多个变量，将多个变量封装成一个对象，通过 AtomicReference 来保证原子性。

原子操作类有哪些



原子操作类的原理

使用CAS实现

以 AtomicInteger 的添加方法为例:

```
1 public final int getAndIncrement() {  
2     return unsafe.getAndAddInt(this, valueOffset, 1);  
3 }
```

```
1 public final int getAndAddInt(Object var1, long var2, int var4) {  
2     int var5;  
3     do {  
4         var5 = this.getIntVolatile(var1, var2);  
5     } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));  
6  
7     return var5;  
8 }
```

如果 var1 的 var2 偏移地址处的值 **仍然等于 var5**:

- **更新为** `var5 + var4` (加 `var4`) 。

否则：

- **更新失败** (说明其他线程修改了该值) , 重新读取并重试。

死锁产生的条件

死锁产生四条件

- 1. 互斥条件**: 线程不能被多的线程共享
- 2. 持有并等待条件**: 一个线程至少已经持有一个资源，且正在等待额外的资源
- 3. 不可剥夺条件**: 资源不能被强制从一个线程中抢占过来
- 4. 循环等待条件**: 等待关系中产生环路

如何避免死锁

至少破坏死锁产生的一个条件

死锁问题怎么排查

- 首先从系统级别上排查，比如说在 Linux 生产环境中，可以先使用 `top` `ps` 等命令查看进程状态，看看是否有进程占用了过多的资源。
- 接着使用 JDK 自带的一些性能监控工具进行排查，比如说，使用 `jps -l` 查看当前 Java 进程，然后使用 `jstack 进程号` 查看当前 Java 进程的线程堆栈信息，看看是否有线程在等待锁资源。

什么是线程同步

对比项	线程同步 (Synchronization)	互斥 (Mutex)
作用	保证线程之间按正确顺序执行	确保多个线程不能同时访问共享资源
多个线程能否同时访问共享资源？	可能可以 (如读操作)	不能 (同一时刻只能有一个线程访问)
关注点	线程执行的时序问题	线程执行的独占性问题
常见实现	<code>wait()</code> / <code>notify()</code> 、 <code>CountDownLatch</code>	<code>synchronized</code> 、 <code>Lock</code>
应用场景	生产者-消费者模式	多线程计数器、银行账户操作

互斥是线程同步的一种特殊情况，所有互斥都是同步，但并不是所有同步都是互斥。

线程同步的实现方式有哪些

- 互斥量
- 读写锁
- 条件变量
- 自旋锁
- 屏障
- 信号量

悲观锁和乐观锁

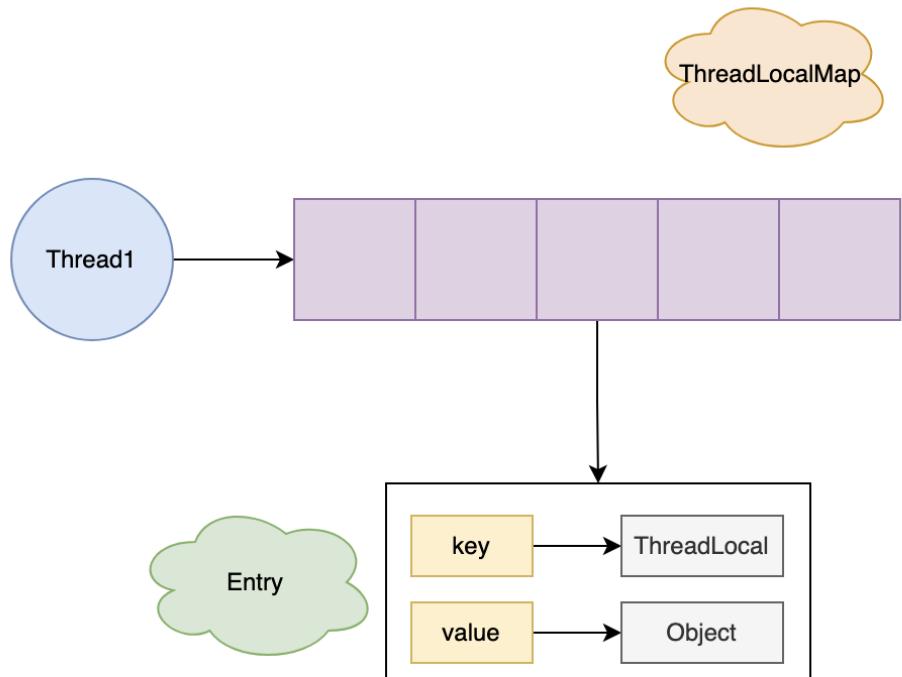
- 乐观锁多用于“读多写少”的环境，避免频繁加锁影响性能；
 - 在读多的场景下，不同的线程可以并行访问共享的资源，提高性能
 - 在写多的场景下，大量线程失败，进入自旋，大量占用CPU资源
- 悲观锁多用于“写多读少”的环境，避免频繁失败和重试影响性能。

ThreadLocal

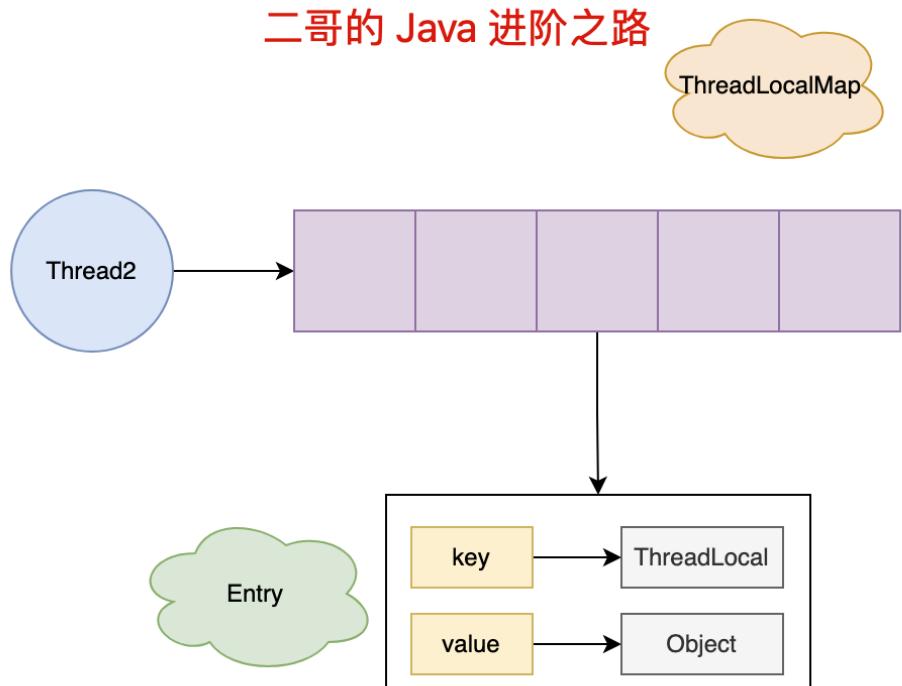
ThreadLocal怎么实现的

ThreadLocal 的实现原理就是，每个线程维护一个 ThreadLocalMap，key 为 ThreadLocal 对象，value 为想要实现线程隔离的对象。

- 1、当需要存线程隔离的对象时，通过 ThreadLocal 的 set 方法将对象存入 Map 中。
- 2、当需要取线程隔离的对象时，通过 ThreadLocal 的 get 方法从 Map 中取出对象。
- 3、Map 的大小由 ThreadLocal 对象的多少决定。

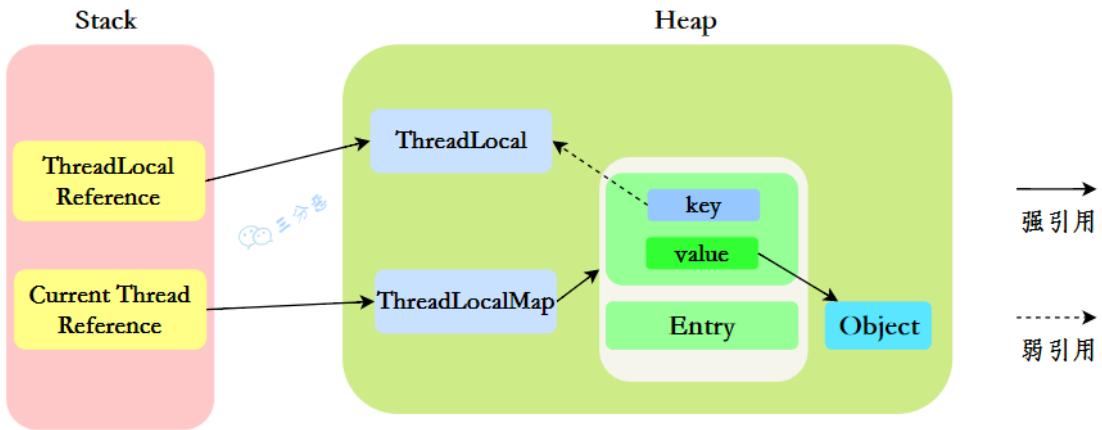


二哥的 Java 进阶之路



ThreadLocal内存泄露是怎么回事

什么是弱引用，什么是强引用



如果某个 ThreadLocal 变量在代码中不再有强引用（例如 `threadLocal = null;`），并且它的 key 是 弱引用（在 ThreadLocalMap 中），那么在下一次垃圾回收时，ThreadLocal 实例就会被回收。

在 ThreadLocal 被垃圾收集后，下一次访问 ThreadLocalMap 时，Java 会自动清理那些键为 null 的条目（参照源码中的 `replaceStaleEntry` 方法），这个过程会在执行 ThreadLocalMap 相关操作（如 `get()`, `set()`, `remove()`）时触发。

内存泄露

通常情况下，随着线程 Thread 的结束，其内部的 ThreadLocalMap 也会被回收，从而避免了内存泄漏。

但如果一个线程一直在运行，并且其 ThreadLocalMap 中的 `Entry.value` 一直指向某个强引用对象，那么这个对象就不会被回收，从而导致内存泄漏。当 `Entry` 非常多时，可能就会引发更严重的内存溢出问题。

ThreadLocalMap的源码看过吗

1. 元素数组

一个 `table` 数组，存储 `Entry` 类型的元素，`Entry` 是 ThreadLocal 弱引用作为 `key`，`Object` 作为 `value` 的结构。

```
1 | private Entry[] table;
```

2. 散列方法

散列方法就是怎么把对应的 `key` 映射到 `table` 数组的相应下标，ThreadLocalMap 用的是哈希取余法，取出 `key` 的 `threadLocalHashCode`，然后和 `table` 数组长度减一&运算（相当于取余）。

```
1 | int i = key.threadLocalHashCode & (table.length - 1);
```

每创建一个 ThreadLocal 对象，它就会新增 `0x61c88647`，这个值很特殊，它是斐波那契数也叫 黄金分割数。`hash` 增量为 这个数字，带来的好处就是 `hash` 分布非常均匀。

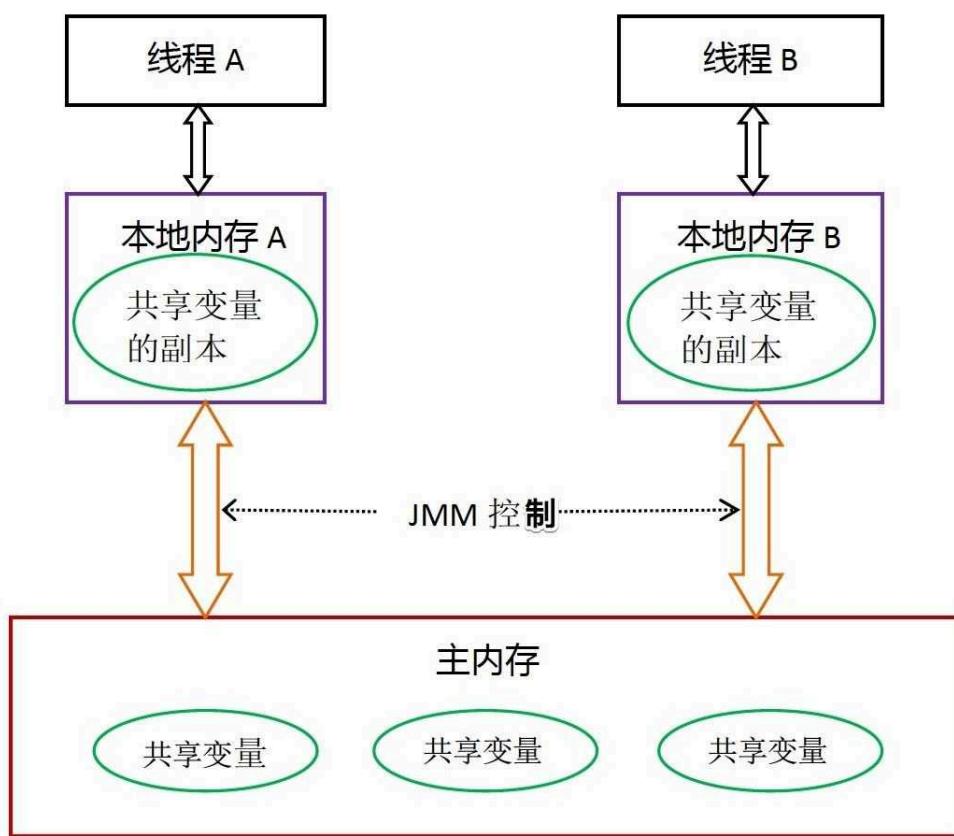
```
1 private static final int HASH_INCREMENT = 0x61c88647;  
2  
3 private static int nextHashCode() {  
4     return nextHashCode.getAndAdd(HASH_INCREMENT);  
5 }
```

父子线程怎么共享数据

父线程能用 ThreadLocal 来给子线程传值吗？不能。

内存模型

为什么线程要用自己的内存



- 线程需要自己的栈来存储局部变量（比如一个方法中的局部变量，不同线程调用同一方法时，局部变量不能互相干扰）
- 如果所有线程都直接操作主内存中的共享变量，会引发更多的内存访问竞争，这不仅影响性能，还增加了线程安全问题的复杂度。
- 现代 CPU 为了优化执行效率，可能会对指令进行乱序执行（指令重排序）。使用本地内存（CPU 缓存和寄存器）可以在不影响最终执行结果的前提下，使得 CPU 有更大的自由度来乱序执行指令，从而提高执行效率。

final变量如何保证可见性

Java 内存模型 (JMM) 规定：

- 普通变量 在构造对象时，可能先发布（对象可见），后赋值，导致其他线程看到的是未初始化的值（即重排序问题）。
- final 变量 由于 JMM 规则，构造方法执行完成后，final 变量的值必须对其他线程可见，避免了重排序导致的未初始化问题。

什么是指令重排

1. 编译器优化

- 编译器会调整指令顺序，以提高指令流水线效率。

2. CPU 指令乱序执行

- CPU 可能会调整指令的执行顺序，以提高指令并行度。

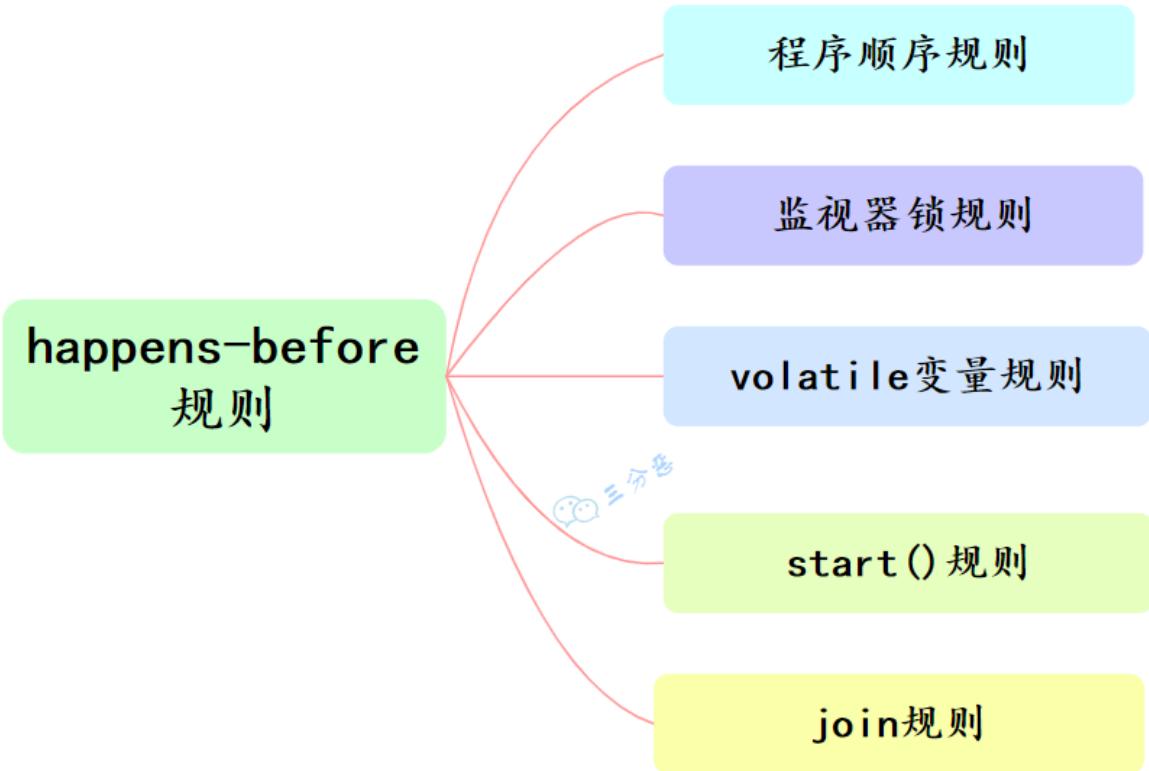
3. 内存系统的重排序

- 处理器会使用缓存和写缓冲区，导致内存访问顺序可能与代码执行顺序不同。

指令重排有限制吗? happens-before了解吗?

指令重排也是有一些限制的，有两个规则happens-before和as-if-serial来约束。

- 如果一个操作 happens-before 另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。
- 两个操作之间存在 happens-before 关系，并不意味着 Java 平台的具体实现必须要按照 happens-before 关系指定的顺序来执行。如果重排序之后的执行结果，与按 happens-before 关系来执行的结果一致，那么这种重排序并不非法



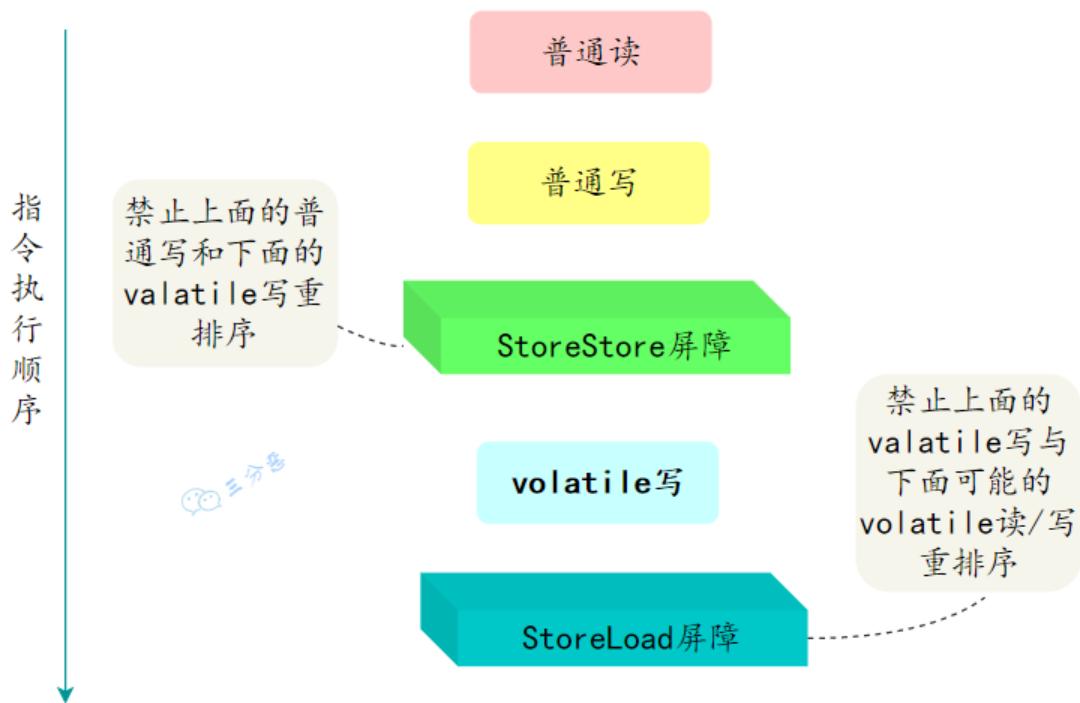
- **程序顺序规则**: 一个线程中的每个操作, happens-before 于该线程中的任意后续操作。
- **监视器锁规则**: 对一个锁的解锁, happens-before 于随后对这个锁的加锁。
- **volatile 变量规则**: 对一个 volatile 域的写, happens-before 于任意后续对这个 volatile 域的读。
- **传递性**: 如果 A happens-before B, 且 B happens-before C, 那么 A happens-before C。
- **start() 规则**: 如果线程 A 执行操作 ThreadB.start() (启动线程 B) , 那么 A 线程的 ThreadB.start() 操作 happens-before 于线程 B 中的任意操作。
- **join() 规则**: 如果线程 A 执行操作 ThreadB.join() 并成功返回, 那么线程 B 中的任意操作 happens-before 于线程 A 从 ThreadB.join() 操作成功返回。

as-if-serial 又是什么? 单线程的程序一定是顺序的吗

as-if-serial 语义的意思是: 不管怎么重排序 (编译器和处理器为了提高并行度), **单线程程序的执行结果不能被改变**。

volatile原理

volatile写插入内存屏障后生成的指令序列示意图



1. 如何保证可见性

- 对volatile变量的写指令后会加入**写屏障**
 - 写屏障保证在该屏障之前，对所有变量的改动都同步到主存当中
- 对volatile变量的读指令前会加入**读屏障**
 - 读屏障保证在该屏障之后，对共享变量的读取加载的是主存中最新数据

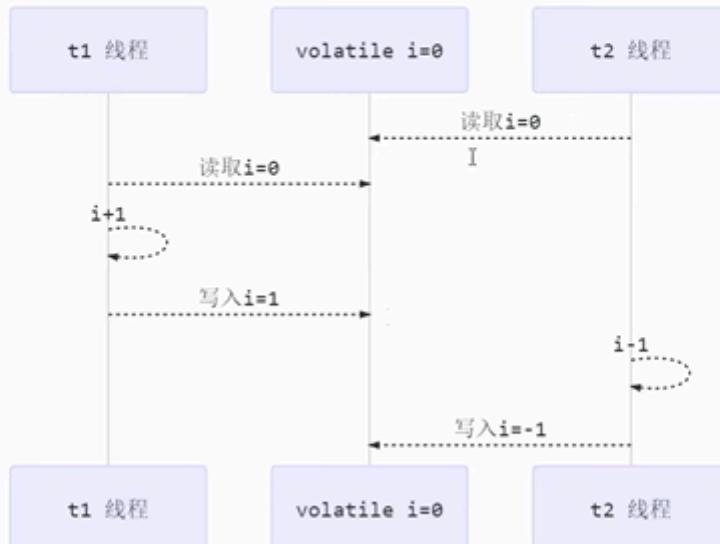
2. 如何保证有序性

- 写屏障会确保指令重排序时，不会将写屏障之前的代码排在写屏障之后
- 读屏障会确保指令重排序时，不会将读屏障之后的代码排在读屏障之前

3. 不能解决指令交错

还是那句话，不能解决指令交错：

- 写屏障仅仅是保证之后的读能够读到最新的结果，但不能保证读跑到它前面去
- 而有序性的保证也只是保证了本线程内相关代码不被重排序



```
1 | private volatile SomeObject obj = new SomeObject();
```

虽然 volatile 确保了 obj 引用的可见性，但对 obj 引用的具体对象的操作并不受 volatile 保护。如果需要保证引用对象内部状态的线程安全，需要使用其他同步机制（如 synchronized 或 ReentrantLock）。

懒汉式单例模式

```
1 | public final class Singleton {  
2 |     private Singleton(){}  
3 |     private static Singleton INSTANCE = null;  
4 |  
5 |     public static Singleton getInstance() {  
6 |         if(INSTANCE == null){  
7 |             synchronized (Singleton.class){  
8 |                 if(INSTANCE == null){  
9 |                     INSTANCE = new Singleton();  
10 |                 }  
11 |             }  
12 |         }  
13 |         return INSTANCE;  
14 |     }  
15 | }
```

这段代码是不正确的，因为 INSTANCE 变量并没有完全被 synchronized 代码块包围住。且 synchronized 内部是有可能发生指令重排序的，即先给 INSTANCE 赋值，再调用 new Singleton() 方法初始化，这时有第二个线程在 INSTANCE 初始化之前就返回了 INSTANCE。

```

1  public final class Singleton {
2      private Singleton(){}
3      private static Singleton INSTANCE = null;
4
5      public static Singleton getInstance() {
6          if(INSTANCE == null){
7              synchronized (Singleton.class){
8                  if(INSTANCE == null){
9                      INSTANCE = new Singleton();
10                 }
11             }
12         }
13         return INSTANCE;
14     }
15 }
16

```

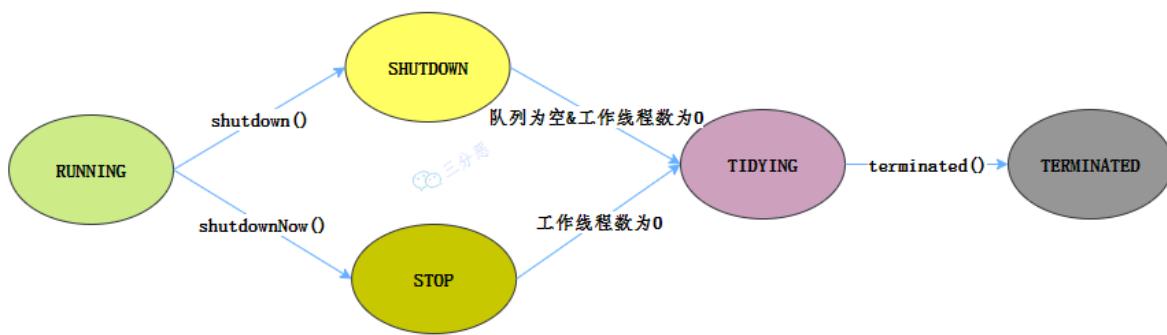
在INSTANCE变量上加上volatile防止重排序，通过写屏障保证初始化在赋值之前执行。

线程池

线程池状态有哪些

ThreadPoolExecutor使用int的高3位来表示线程池状态，低29位表示线程数量

状态名	高 3 位	接收新任务	处理阻塞队列任务	说明
RUNNING	111	Y	Y	
SHUTDOWN	000	N	Y	不会接收新任务，但会处理阻塞队列剩余任务
STOP	001	N	N	会中断正在执行的任务，并抛弃阻塞队列任务
TIDYING	010	-	-	任务全执行完毕，活动线程为 0 即将进入终结
TERMINATED	011	-	-	终结状态



线程池参数有哪些

corePoolSize: 核心线程数

maximumPoolSize: 最大线程数 (救急线程 = maximumPoolSize - corePoolSize, 当核心线程和阻塞队列都满了的时候，下一个来的任务交给救急线程运行，救急线程执行完了就会销毁，不会像核心线程一样保留)

keepAliveTime: 生存时间 — 针对救急线程

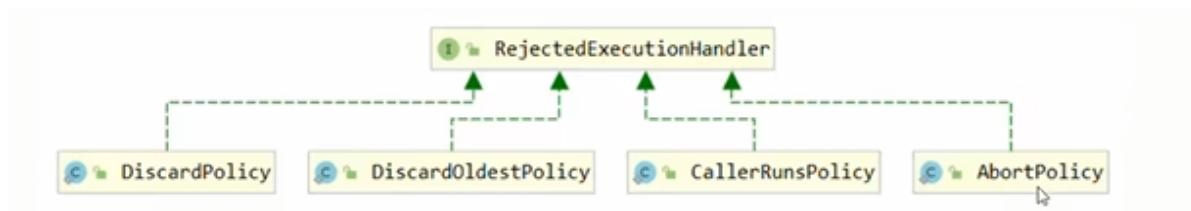
unit: 时间单位

workQueue: 阻塞队列

threadFactory: 线程工厂 — 为线程创建时起名字

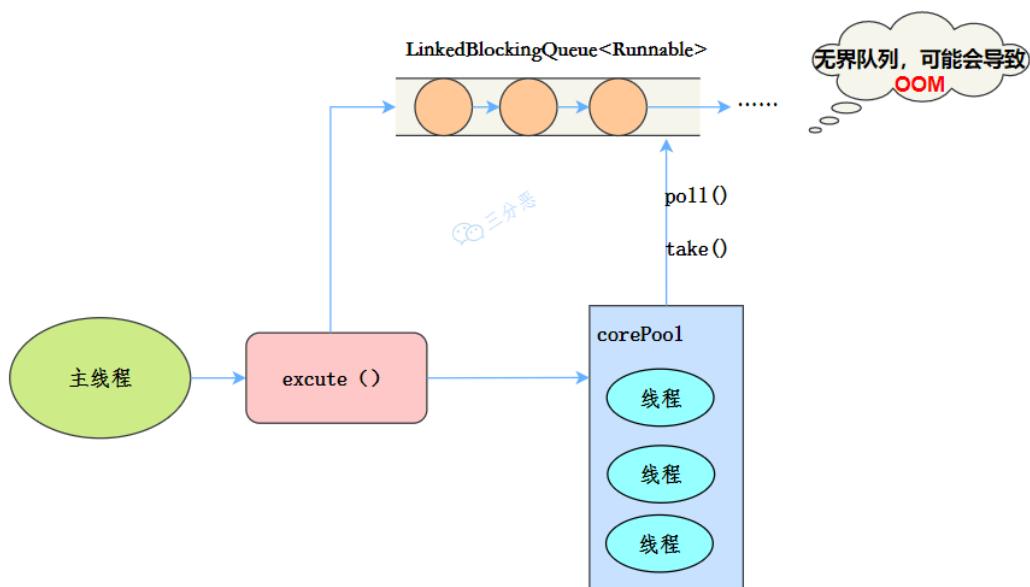
handler: 拒绝策略

线程池的拒绝策略有哪些



线程池类型

`newFixedThreadPool`

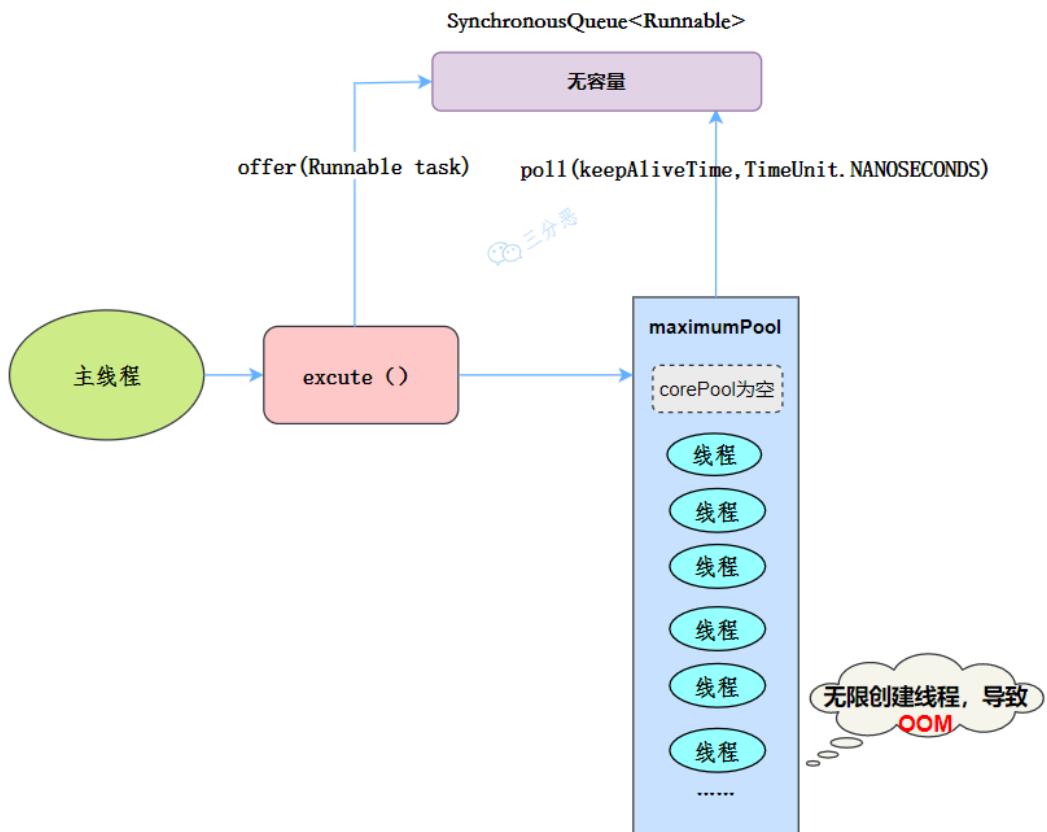


```
1 public static ExecutorService newFixedThreadPool(int nThreads) {  
2     return new ThreadPoolExecutor(  
3         nThreads,  
4         nThreads,  
5         0L,  
6         TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());  
7 }
```

- 核心线程 = 最大线程数，没有救急线程，因此也无需超时时间
- 阻塞队列是无界的

适合于任务量已知，相对耗时的任务

newCachedThreadPool

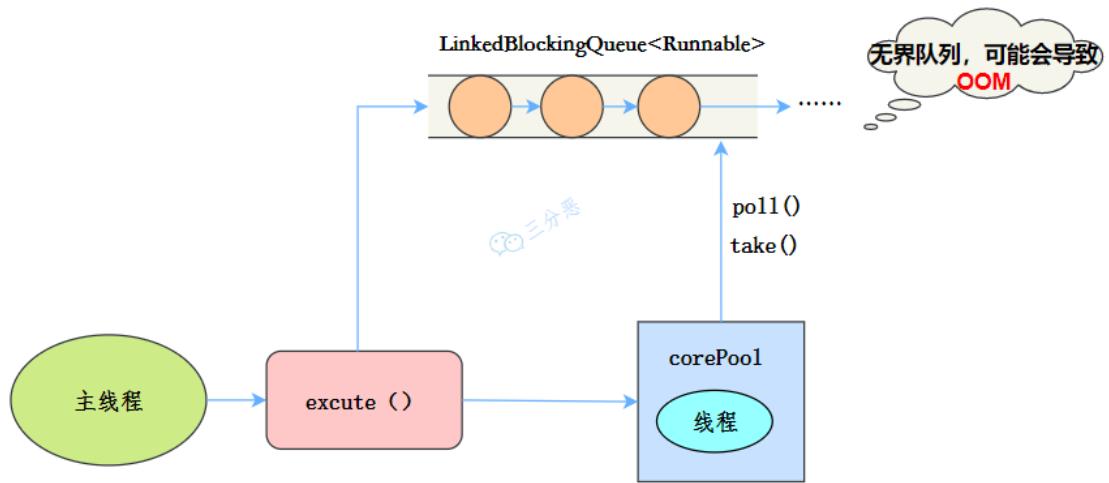


```
1 public static ExecutorService newCachedThreadPool() {
2     return new ThreadPoolExecutor(
3         0,
4         Integer.MAX_VALUE,
5         60L,
6         TimeUnit.SECONDS,
7         new SynchronousQueue<Runnable>());
8 }
```

- 核心线程数是0，最大线程数是`Integer.MAX_VALUE`，生存时间是60s
 - 全部都是救急线程
 - 救急线程可以无限创建
- 队列采用了`SynchronousQueue`实现特点是，没有容量，没有线程来取是放不进去的

适合任务数密集，但每个任务执行时间较短的情况

newSingleThreadExecutor



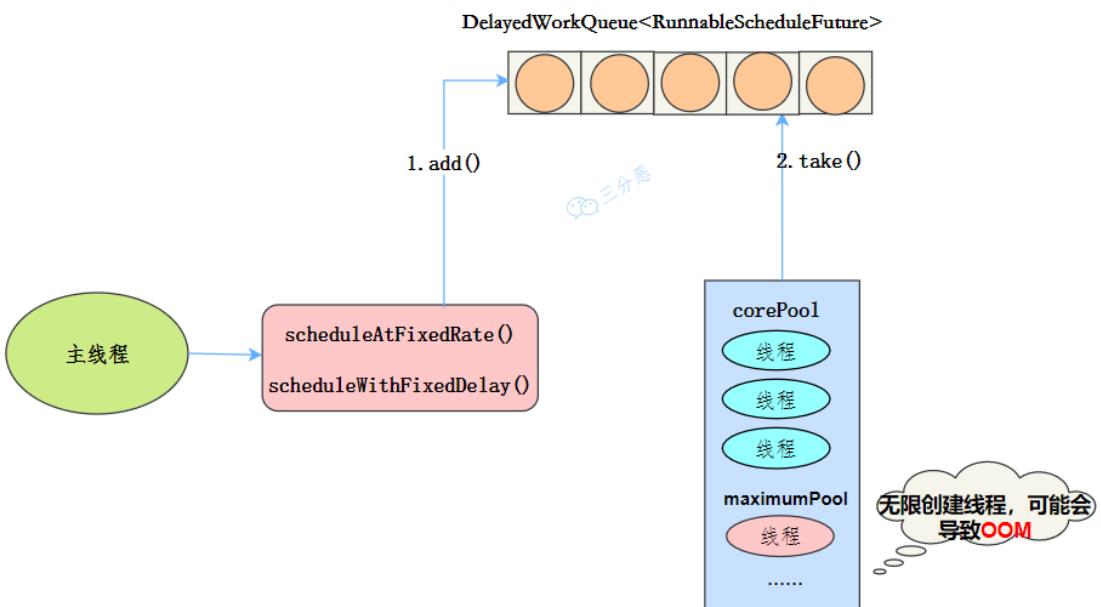
```

1 public static ExecutorService newSingleThreadExecutor() {
2     return new FinalizableDelegatedExecutorService(
3         new ThreadPoolExecutor(
4             1,
5             1,
6             0L,
7             TimeUnit.MILLISECONDS,
8             new LinkedBlockingQueue<Runnable>()
9         );
10 }

```

- 线程数固定为1且始终为1，不能修改
- 任务数多于1时，会放入无界队列排队
- 还会新建一个线程，保证池的正常工作（始终保证有一个可用的线程）

`newScheduledThreadPool`



```
1 public static ExecutorService newScheduledThreadExecutor() {  
2     return new ThreadPoolExecutor(  
3         corePoolSize,  
4         Integer.MAX_VALUE,  
5         0,  
6         TimeUnit.NANOSECONDS,  
7         new DelayedWorkQueue<Runnable>()  
8     );  
9 }
```

- 最大线程数为 Integer.MAX_VALUE，也有 OOM 的风险
- 阻塞队列是 DelayedWorkQueue
- keepAliveTime 为 0
- scheduleAtFixedRate()：按某种速率周期执行
- scheduleWithFixedDelay(): 在某个延迟后执行

线程池的阻塞队列有哪些

- ArrayBlockingQueue
 - 有界的先进先出的阻塞队列，底层是一个数组，适合固定大小的线程池
- LinkedBlockingQueue
 - 底层数据结构是链表，如果不指定大小，默认大小是 Integer.MAX_VALUE，相当于一个无界队列。
- DelayQueue
 -
- PriorityBlockingQueue
 - 支持优先级排序的无界阻塞队列。任务按照其自然顺序或通过构造器给定的 Comparator 来排序。
- SynchronousQueue
 - 实际上它不是一个真正的队列，因为没有容量。每个插入操作(put())必须等待另一个线程的移除(get())操作，同样任何一个移除操作都必须等待另一个线程的插入操作。

线程池的shutdown()和shutdownNow()区别

shutdown()

- 线程池变为SHUTDOWN状态
- 线程池不能在接收新的任务
- 已经提交的任务（包括当前队列中的任务）仍然会执行，都执行完后线程池才真正关闭
- 正在执行的任务继续运行，不会被中断

shutdownNow()

- 线程池变为STOP状态
- 线程池不能在接受新任务
- 当前队列中的会被直接返回，不执行
- 正在执行的任务会被 `interrupt()` 中断

最终线程池都会变成TERMINATED状态

线程池提交`execute()`和`submit()`有什么区别

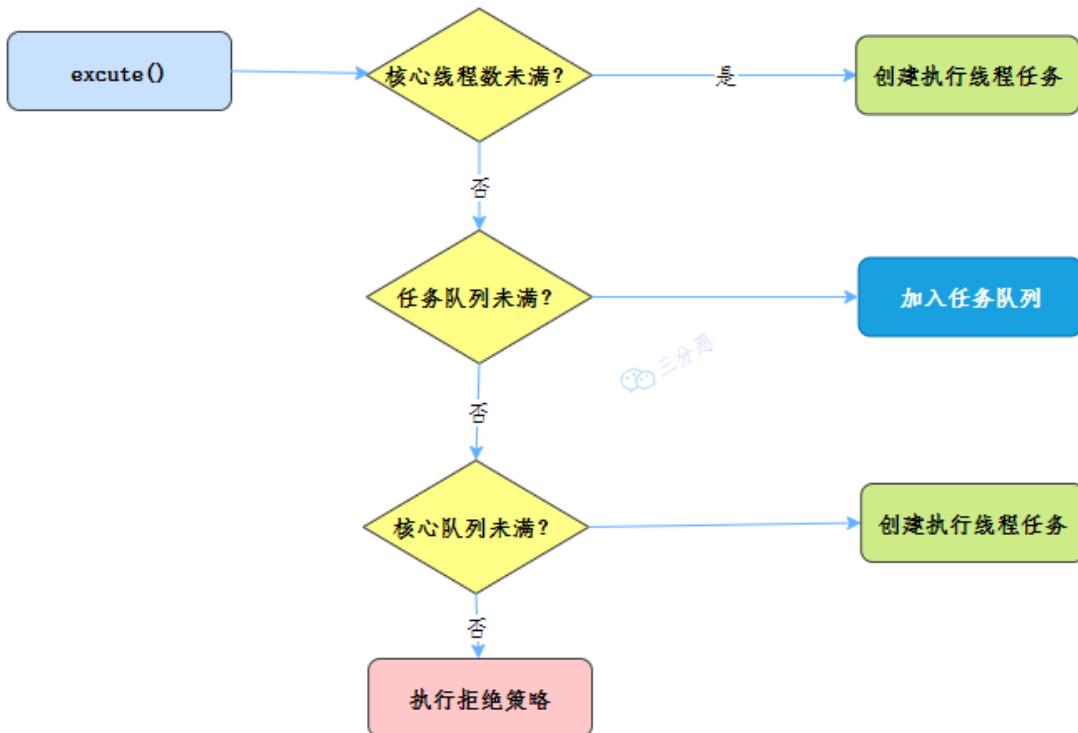
1. `execute()` 用于提交不需要返回值的任务

```
1 | ThreadPool.execute(new Runnable() {  
2 |     @Override  
3 |     public void run() {  
4 |         // TODO  
5 |     }  
6 | });
```

2. `submit()` 用于提交需要返回值的任务。线程池会返回一个future类型的对象，可以通过future的get方法获取返回值。

```
1 | Future<Object> future = executor.submit(ReturnValueTask);  
2 | try{  
3 |     Object s = future.get();  
4 | } catch(Exception e){  
5 |     // 捕获异常  
6 | }
```

线程池的工作流程



当任务提交时，ThreadPoolExecutor 的执行逻辑如下：

1. 如果当前线程数 < corePoolSize
 - 直接创建新线程 来执行任务，而不是进入队列。
2. 如果当前线程数 ≥ corePoolSize
 - 任务被添加到**任务队列 (workQueue)**，等待线程执行。
3. 如果任务队列满了
 - 如果线程数 < maxPoolSize，就**创建新线程** 处理任务。
4. 如果线程数达到 maxPoolSize，并且任务队列也满了
 - 任务会被拒绝，执行**拒绝策略**。

ThreadPoolExecutor 只会在有任务提交时才创建线程，即不会在创建线程池时立即创建 corePoolSize 个线程。

线程池的线程数应该怎么配置

首先分析线程池中执行的任务类型是 CPU 密集型还是 IO 密集型

- 对于 CPU 密集型任务，我的目标是尽量减少线程上下文切换，以优化 CPU 使用率。一般来说，核心线程数设置为处理器的核心数或核心数加一（以备不时之需，如某些线程因等待系统资源而阻塞时）是较理想的选择。
 - 对于 CPU 密集型任务，线程数接近 CPU 核心数即可
- 对于 IO 密集型任务，由于线程经常处于等待状态（等待 IO 操作完成），可以设置更多的线程来提高并发性（比如说 2 倍），从而增加 CPU 利用率。
 - 对于 IO 密集型任务，线程数可以简单设置为 CPU 核心数 × 2。

如何知道设置的线程数多了还是少了

可以先通过 top 命令观察 CPU 的使用率，如果 CPU 使用率较低，可能是线程数过少；如果 CPU 使用率接近 100%，但吞吐量未提升，可能是线程数过多。

线程池怎么处理异常

问题：线程池中的线程抛出异常不会被主线程捕获

```
1 ExecutorService executor = Executors.newFixedThreadPool(2);
2 executor.execute(() -> {
3     throw new RuntimeException("任务执行异常");
4 });
5 System.out.println("主线程不会受到影响");
6 executor.shutdown();
```

1. 使用try-catch在任务内部捕获异常

2. 使用 Future 获取异常（适用于 submit()）

```
1 ExecutorService executor = Executors.newFixedThreadPool(2);
2 Future<?> future = executor.submit(() -> {
3     throw new RuntimeException("任务异常");
4 });
5
6 try {
7     future.get(); // 获取任务执行结果，如果有异常，会在这里抛出 ExecutionException
8 } catch (InterruptedException | ExecutionException e) {
9     System.out.println("捕获异常: " + e.getCause().getMessage());
10 }
11 executor.shutdown();
12
```

3. 自定义 ThreadFactory 捕获异常

```
1 ExecutorService executor = Executors.newFixedThreadPool(2, runnable -> {
2     Thread thread = new Thread(runnable);
3     thread.setUncaughtExceptionHandler((t, e) ->
4         System.out.println("线程 " + t.getName() + " 捕获异常: " +
e.getMessage());
5     return thread;
6 });
7
8 executor.execute(() -> {
9     throw new RuntimeException("线程池任务异常");
10 });
11
12 executor.shutdown();
13
```

4. 重写 ThreadPoolExecutor.afterExecute() 方法

```
1 public class AfterExecuteExample extends ThreadPoolExecutor {  
2     public AfterExecuteExample(int corePoolSize, int maxPoolSize, long  
3         keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {  
4         super(corePoolSize, maxPoolSize, keepAliveTime, unit, workQueue);  
5     }  
6  
7     @Override  
8     protected void afterExecute(Runnable r, Throwable t) {  
9         super.afterExecute(r, t);  
10        if (t == null && r instanceof Future<?>) {  
11            try {  
12                ((Future<?>) r).get(); // 捕获 submit() 任务的异常  
13            } catch (InterruptedException | ExecutionException e) {  
14                t = e.getCause();  
15            }  
16            if (t != null) {  
17                System.out.println("捕获线程池异常: " + t.getMessage());  
18            }  
19        }  
20    }  
}
```

线程池在使用的时候需要注意什么

1. 选择合适的线程池大小

- 过小的线程池可能会导致任务一直在排队
- 过大的线程池可能会导致大家都在竞争 CPU 资源，增加上下文切换的开销

2. 任务队列的选择

- 使用有界队列可以避免资源耗尽的风险，但是可能会导致任务被拒绝
- 使用无界队列虽然可以避免任务被拒绝，但是可能导致内存耗尽

3. 尽量使用自定义的线程池

- newFixedThreadPool 线程池由于使用了 LinkedBlockingQueue，队列的容量默认无限大，实际使用中出现任务过多时会导致内存溢出
- newCachedThreadPool 线程池由于核心线程数无限大，当任务过多的时候会导致创建大量的线程，可能机器负载过高导致服务宕机

如何自己设计一个线程池

单机线程池执行时断电了怎么处理

对阻塞队列持久化；正在处理任务事务控制；断电之后正在处理任务的回滚，通过日志恢复该次操作；服务器重启后阻塞队列中的数据再加载。

并发工具类

HashMap有哪些线程安全问题

JDK7 HashMap并发死链

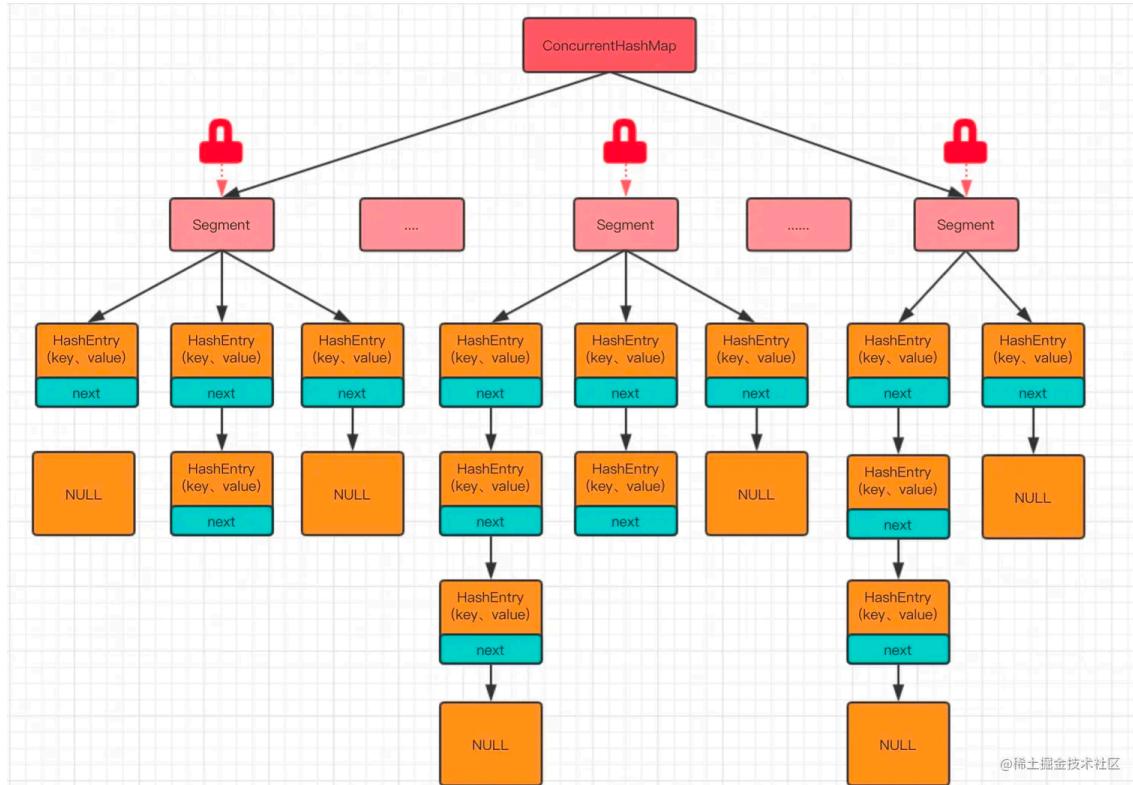
因为JDK7中的HashMap扩容时，采用的是头插法，多线程的情况下，一个线程扩容完了后，由于节点链表已经改变，另一个线程再进行操作就可能会产生循环链表，死循环。

HashMap和ConcurrentHashMap的区别

- HashMap 是非线程安全的，多线程环境下应该使用 ConcurrentHashMap。
- 由于 HashMap 仅在单线程环境下使用，所以不需要考虑同步问题，因此效率高于 ConcurrentHashMap。

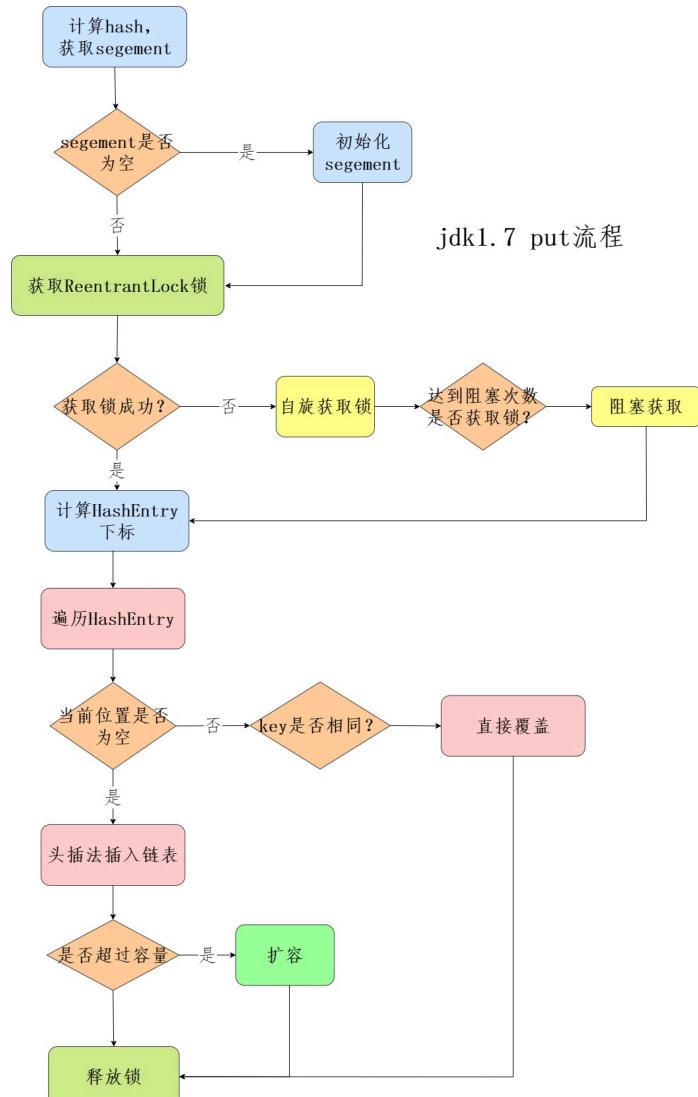
ConcurrentHashMap的原理

- JDK 7



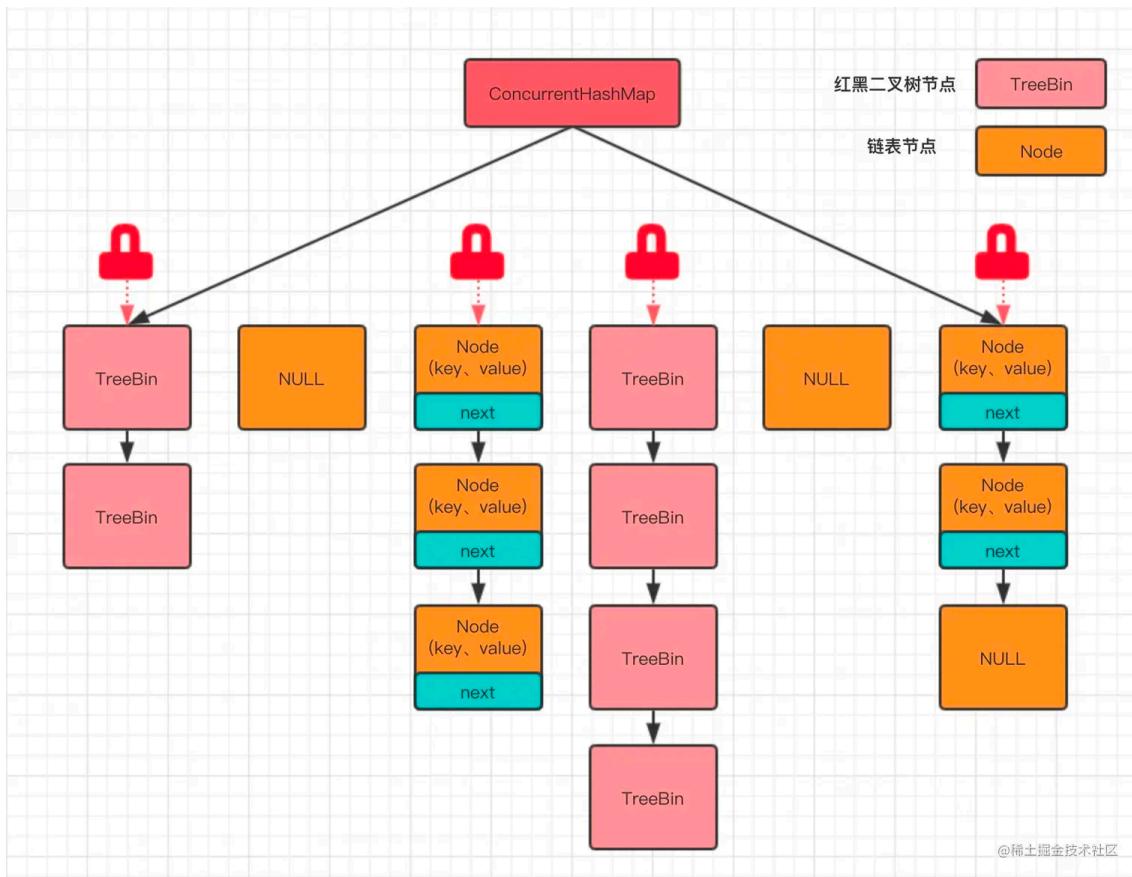
- 采用的是分段锁机制（Segment Locking），整个Map被分为若干段，每个段都可以独立地加锁。因此，不同线程可以同时操作不同的段，从而实现并发访问。
- **put方法流程：** ConcurrentHashMap 的 put 流程和 HashMap 非常类似，只不过是先定位到具体的 Segment，然后通过 ReentrantLock 去操作而已。
 1. 计算 hash，定位到 segment，segment 如果是空就先初始化；

2. 使用 ReentrantLock 加锁，如果获取锁失败则尝试自旋，自旋超过次数就阻塞获取，保证一定能获取到锁；
3. 遍历 HashEntry，key 相同就直接替换，不存在就插入。
4. 释放锁。



- **get方法流程**: get 也很简单，通过 `hash(key)` 定位到 segment，再遍历链表定位到具体的元素上，需要注意的是 value 是 volatile 的，所以 get 是不需要加锁的。

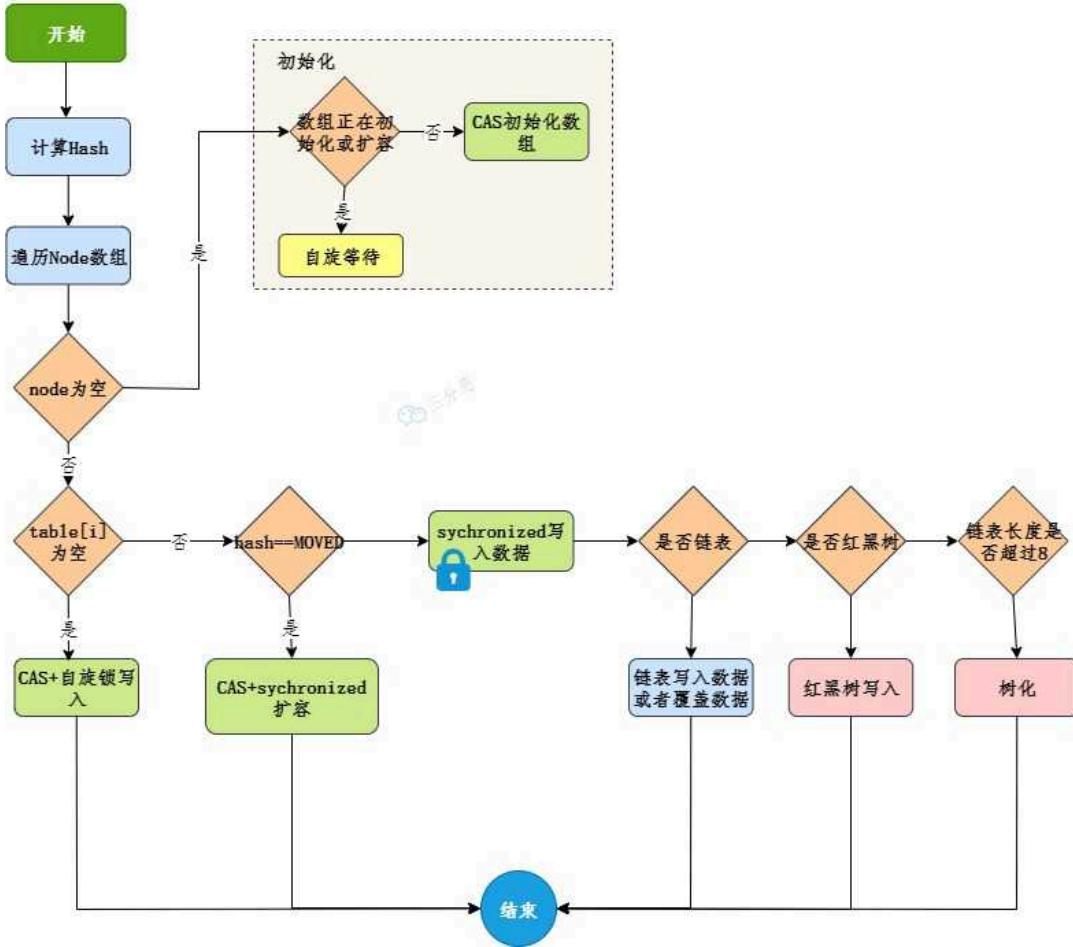
- JDK 8以后



@稀土掘金技术社区

- 在JDK8及以上的版本中，进行了优化，不再使用分段锁，而是使用了一种更加精细化的锁—桶锁，以及CAS无锁算法。每个桶（Node 数组的每个元素）都可以独立地加锁，从而实现更高级别的并发访问。
- 对于读操作，通常不需要加锁，可以直接读取，ConcurrentHashMap 内部使用了 volatile 变量来保证内存可见性。
- 对于写操作，ConcurrentHashMap 使用 CAS 操作来实现无锁的更新，这是一种乐观锁的实现，因为它假设没有冲突发生，在实际更新数据时才检查是否有其他线程在尝试修改数据，如果有，采用悲观的锁策略，如 synchronized 代码块来保证数据的一致性。
- put方法流程：**通过计算键的哈希值确定存储位置，如果桶为空，使用 CAS 插入节点；如果存在冲突，通过链表或红黑树插入。在冲突时，如果 CAS 操作失败，会退化为 synchronized 操作。写操作可能触发扩容或链表转为红黑树（当链表长度超过3/4时进行扩容，当链表长度超过 8 就转换成红黑树）。

o 1.8 ConcurrentHashMap Put 流程



- o **get 方法流程**: 通过计算哈希值快速定位桶，在桶中查找目标节点，多个 key 值时链表遍历和红黑树查找。读操作是无锁的，依赖 volatile 保证线程可见性。

CountDownLatch 了解吗

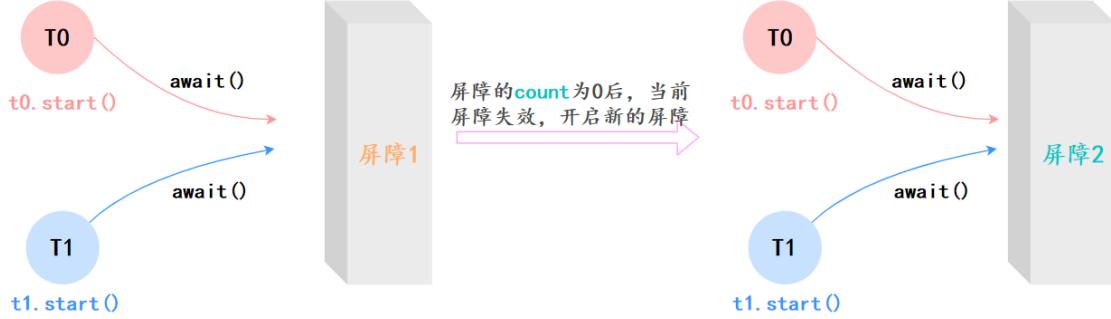
用于协调多个线程之间的同步，允许一个或多个线程等待，直到其他线程中执行的一组操作完成。

- 初始化: 创建 CountDownLatch 对象时，指定计数器的初始值。
- 等待 (await) : 一个或多个线程调用 await 方法，进入等待状态，直到计数器的值变为零。
- 倒计数 (countDown) : 其他线程在完成各自任务后调用 countDown 方法，将计数器的值减一。当计数器的值减到零时，所有在 await 上等待的线程会被唤醒，继续执行。

CyclicBarrier (同步屏障) 了解吗

让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

`await`会触发`count`值减1，如果此时`count`仍不为0，当前线程将等待其它线程到达

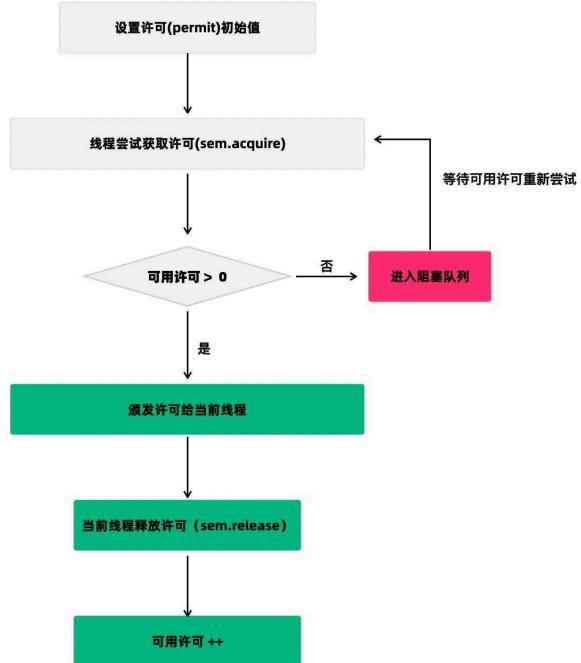


CyclicBarrier和CountDownLatch有什么区别

- CountDownLatch 是一次性的，而 CyclicBarrier 则可以多次设置屏障，实现重复利用；
- CountDownLatch 中的各个子线程不可以等待其他线程，只能完成自己的任务；而 CyclicBarrier 中的各个线程可以等待其他线程
- 在 CyclicBarrier 中，如果某个线程遇到了中断、超时等问题时，则处于 await 的线程都会出现问题；在 CountDownLatch 中，如果某个线程出现问题，其他线程不受影响

Semaphore (信号量) 了解吗

Semaphore (信号量) 是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源。



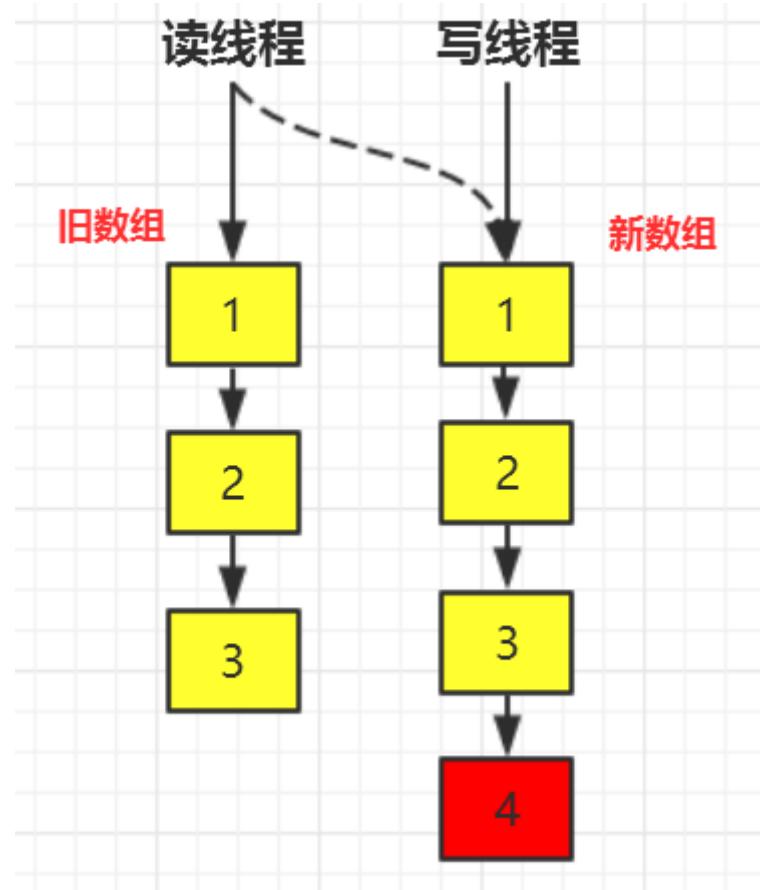
©稀土掘金技术社区

Exchanger了解吗

Exchanger (交换者) 是一个用于线程间协作的工具类。Exchanger 用于进行线程间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。

CopyOnWriteArrayList的实现原理

CopyOnWriteArrayList 是一个线程安全的 ArrayList，它遵循写时复制（Copy-On-Write）的原则，即在写操作时，会先复制一个新的数组，然后在新的数组上进行写操作，写完之后再将原数组引用指向新数组。



- 读写分离，在写操作上使用Synchronized加锁，读操作上不加锁
- 读操作弱一致性
- 适用于“读多写少”的场景，如果需要实时一致性，应该用 synchronized或 ConcurrentHashMap。

BlockingQueue是什么

是线程安全的队列

- 当队列为空，消费者线程（`take()`）会 **自动阻塞**，直到队列中有元素可取。
- 当队列满了，生产者线程（`put()`）会 **自动阻塞**，直到队列有空位可用。

BlockingQueue的实现原理

put()

- 先获取锁 `lock.lockInterruptibly()`。
- 如果队列 **已满**，调用 `notFull.await()` 阻塞，等待 `take()` 唤醒。
- 当 `take()` 消费一个元素，调用 `notFull.signal()` 唤醒 `put()` 线程。

take()

- 先获取锁 `lock.lockInterruptibly()`。
- 如果队列 **为空**, 调用 `notEmpty.await()` **阻塞**, 等待 `put()` 唤醒。
- 当 `put()` 放入一个元素, 调用 `notEmpty.signal()` 唤醒 `take()` 线程。

Fork / Join框架了解吗

参考资料

[黑马程序员JUC并发编程教程](#):

- P11 - 15 (创建线程)
- p44 - 46 (线程状态)
- p78 - 87 (Synchronized优化原理)
- p120 - 127 (ReentrantLock)
- p146 - 151 (volatile原理)
- P200 - 219 (从自定义线程池到ThreadPoolExecutor)
- P274 - 296 (ConcurrentHashMap)

[Java面试](#)