

# 多线程 结合PDF

---

## 进程与线程

进程包括线程

线程共享内存空间

线程更轻量

## 并行和并发有什么区别

单核CPU：实际还是串行执行的（微观串行）

宏观并行：不停切换

这就是并发

多核CPU：

真正的并行

## 创建线程的方式

### 继承`Thread`类

重写`run`方法

调用`start`启动线程

### 实现`Runnable`接口

也是重写run方法

传入Thread类

调用start启动线程

## ■ 实现callable接口

重写call方法 有返回值（还可以throw异常） 记住 runnable没有返回值 不能throw

用futuretask包装实现类

再用Thread类 包装futuretask

start方法启动线程 futuretask的get方法拿到返回值

## ■ run和start区别？

start才能真正开启新线程；run只能在当前线程跑

## ■ 线程池

用Executor库

自动分配

！！！！！！ 阿里不建议用Executors创建线程池 因为允许创建数量为Integer 最大值 会OOM

而是通过ThreadPoolExecutor

## 线程状态

六个状态 加一个running(内部没有定义)

new runnable blocked waiting timed waiting terminated

看PDF

start()方法 转变为runnable

没锁 就blocked

wait()后是waiting 被通知了才能继续

sleep() 则timed waiting

## 线程的顺序

■ *join()*

t1.join()

等t1线程执行完

■ *notify*和*notifyall*

wait会释放锁的

wait 让当前线程进入等待状态，直到被 `notify()` 或 `notifyAll()` 唤醒。

后者唤醒所有

前者唤醒随机

**wait**在**object**类

唤醒后还要抢

等待应该出现在循环中，防止虚假唤醒 if改为while判断

## wait和sleep区别？

后者是Thread类 **wait**在**object**类

wait会释放锁的 sleep如果在synchronized内执行 则不会释放锁

wait(固定时间) 也会被提前唤醒

### 共同点

wait(), wait(long) 和 sleep(long) 的效果都是让当前线程暂时放弃 CPU 的使用权，进入阻塞状态

### 不同点

#### 1.方法归属不同

- sleep(long) 是 Thread 的静态方法
- 而 wait(), wait(long) 都是 Object 的成员方法，每个对象都有

#### 2.醒来时机不同

- 执行 sleep(long) 和 wait(long) 的线程都会在等待相应毫秒后醒来
- wait(long) 和 wait() 还可以被 notify 唤醒，wait() 如果不唤醒就一直等下去
- 它们都可以被打断唤醒

#### 3. 锁特性不同（重点）

- wait 方法的调用必须先获取 wait 对象的锁，而 sleep 则无此限制
  - wait 方法执行后会释放对象锁，允许其它线程获得该对象锁（我放弃 cpu，但你们还可以用）
  - 而 sleep 如果在 synchronized 代码块中执行，并不会释放对象锁（我放弃 cpu，你们也用不了）
- 

## 如何停止线程执行

用标记法

stop法（已作废）

使用**interrupt**方法中断线程 其实也是标记法

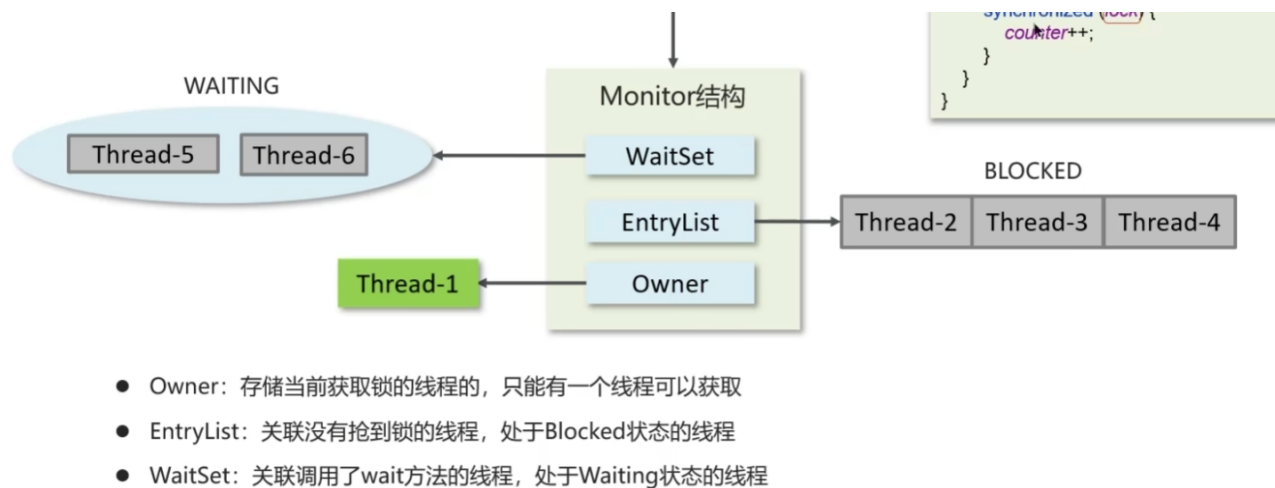
# 并发安全

## Synchronized和其底层原理

互斥的方式 对象锁

底层是Monitor监视器

结构: WaitSet entrylist Owner 三个队列存线程



Monitor是重量级锁 JVM管理 涉及到内核切换 上下文切换 性能低

1.6后 偏向锁和轻量级锁 没有竞争的时候使用

## 偏向锁

偏向锁是指一段同步代码一直被一个线程所访问, 那么该线程会自动获取锁。降低获取锁的代价。

当一个线程第一次访问同步代码块时, JVM 会将对象的 **Mark Word** 设置为该线程的 ID, 表示这个锁已“偏向”于该线程

如果有其他线程尝试获取这个锁, 那么会触发 锁撤销 (锁会从偏向锁变成轻量级锁, 之后的操作与轻量级锁相同)。JVM 会使用 **CAS (Compare and Swap)** 操作将原本偏向锁的对象头修改为轻量级锁或重量级锁标识, 从而实现锁的升级。

## 轻量级锁

用类栈来存锁！！！！

CAS交换 对象的mark word 和lock record 地址 交换

重入了 还要在lock record栈里面再加一个null记录占位（数量+1）并更新对象指针为栈顶

轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。

解锁的话 record最后一个锁 再交换mark word和Lock record

轻量级锁类似于**redis**的分布式锁 存记录 不过轻量级锁没有用到HASHMAP那么简便

## ■ 重量级锁

重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让其他申请的线程进入阻塞，性能降低。

因为是要记录阻塞 如**monitor**所示

## JMM

每个线程分配一个工作内存 线程自己私有

主内存 共享变量

通过主内存进行同步

## CAS和乐观锁

compare and swap

一种乐观锁的思想 在无锁情况下保证线程操作共享数据的原子性

AQS框架 Atomic类

旧值与主内存的值对比 一样 才会修改 若失败则自旋 循环 把主内存的值赋给旧值 重新运算再比对

竞争激烈，效率会受影响

JDK 内部的自旋锁是有限自旋

可重入锁类是自适应自旋 上次自旋成功率高，则增加自旋次数

### 底层

unsafe类 的c++代码

### 和悲观锁的区别

synchronized是悲观锁 互斥等待

## volatile

JIT会优化代码

方案一：使用 `-Xint` 关闭优化器 （得不偿失）

方案二：变量加上`volatile` 告诉jit不要优化

### 禁止指令重排

在读写共享变量时加入不同的屏障，组织其他读写操作越过屏障 达到阻止重排序的效果

加屏障有要求的 不能随便加

### volatile使用技巧：

- 写变量让volatile修饰的变量的在代码最后位置
- 读变量让volatile修饰的变量的在代码最开始位置

两个都volatile 也行 就是性能问题

## AQS pdf183页

抽象队列同步器 基础框架

就是并发安全的集合

### 基本工作机制

state状态 是否有锁

fifo队列（双向链表） 存等待的线程



如何保证资源原子性

用CAS设置 **state**状态 保证原子性

公平锁与否？

可公平 也不公平

又来一个线程 且没排队 那就和队列中那个线程抢 -----非公平

新的只能等-----公平锁

Lock CountDownLatch Semaphore ReentrantLock都是基于AQS

## ReentrantLock

可重入锁

可中断(**syn**不能中断)

设置超时时间 没有获取锁可以放弃

可以设置公平锁

支持多个条件变量

可重入（**syn**也可重入）

**lock()**方法和**unlock()**方法

底层

AQS+CAS

无参构造默认非公平

底层结构:**state** 双向链表的头尾指针 指向抢到锁的线程的指针

## Synchronized和Lock的区别

### 语法

Synchronized 源码是C++实现

后者是JDK实现

前者自动释放 后者要手动释放

### 功能

都是悲观锁

Lock可以实现 公平锁 可打断 可超时 多条件变量 功能更多 重入锁 读写锁

lockInterruptibly() 获取可打断的锁 然后interrupt();

可超时： trylock() 尝试获取锁 可以引入时间参数 超过时间没拿到锁就失败 次数取决于负载

### 性能

Lock粒度高 性能可以更高

### 多条件变量

Condition类 声明条件变量 await()和signal()

## 死锁

互斥条件（Mutual Exclusion）： 至少有一个资源是不可共享的，即一次只能被一个线程使用。

占有且等待条件（Hold and Wait）： 一个线程持有至少一个资源，并等待获取其他线程持有的资源。

非抢占条件（No Preemption）：资源只能在被线程显式释放之后才能被其他线程获取，不能被强制抢占。

循环等待条件（Circular Wait）：一组线程形成一个循环，每个线程都在等待下一个线程所持有的资源。

## 死锁诊断

jps 看运行的进程状态信息

jstack 看线程的堆栈信息

jps查死锁线程ID

jstack 查线程看看死锁情况

可视化工具:Jconsole 查死锁

VisualVM 跟jstack差不多

## ConcurrentHashMap

线程安全的hashmap

1.7以前 加一层Segment分片过滤（默认16个分片 不能扩容） 每一个分片 再存HashMap 有锁 CAS操作和自旋获取锁 再修改

`ConcurrentHashMap` 将整个映射分为多个“段”，每个“段”是一个包含一定数量条目的哈希表，实际上是一个内部类 `Segment`，它继承自 `HashMap`。每个 `Segment` 都有自己的锁

就是把hashmap分段 每段一个锁 CAS操作加锁和自旋锁

1.8以后 CAS+synchronized

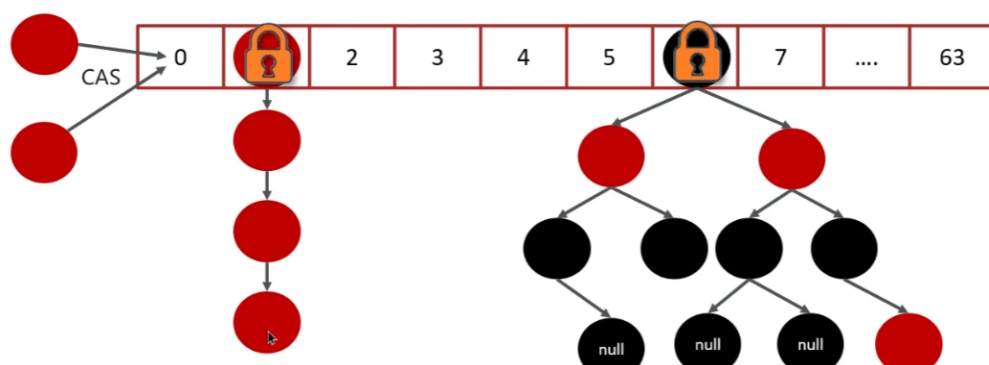
在 `ConcurrentHashMap` 中，每个桶（**bucket**）是独立加锁的，如果多个线程访问的是不同的桶，它们可以并行执行，而不需要等待其他线程释放锁。具体来说，

`ConcurrentHashMap` 中的每个桶都使用了一个 独立的锁

Java8 中的 `ConcurrentHashMap` 使用的 `Synchronized` 锁加 CAS 的机制。结构也由 Java7 中的 `Segment` 数组 + `HashEntry` 数组 + 链表 进化成了 **Node** 数组 + 链表 / 红黑树，**Node** 是类似于一个 `HashEntry` 的结构。它的冲突再达到一定大小时会转化成红黑树，在冲突小于一定数量时又退回链表。

## 细化的1.7

- CAS控制数组节点的添加
- synchronized只锁定当前链表或红黑二叉树的首节点，只要hash不冲突，就不会产生并发的问题，效率得到提升



高何软件 / 人才培训

## 并发程序为啥会出问题（三大特性）

原子性

## 可见性

## 有序性

## 原子性

一口气完成 加锁就完事了

## 可见性

一个线程对共享变量的修改，能否及时地被其他线程看到。简单来说，就是多个线程之间是否能够“看到”彼此对共享数据的更新。

Key:防止JIT优化代码 用volatile

或者共享变量加锁

有序性

防止指令重排

## 线程池

再次重提：用ThreadPoolExecutor 不会OOM

七大参数

```
public ThreadPoolExecutor(int corePoolSize, //核心线程池大小
                           int maximumPoolSize, //最大核心线程池大小
                           long keepAliveTime, //超时了没有人调用就会释放
                           TimeUnit unit, //超时单位
                           BlockingQueue<Runnable> workQueue, //阻塞队
                           ThreadFactory threadFactory, //线程工厂，创
                           RejectedExecutionHandler handler //拒绝策略)
//七个参数
```

当一个新的任务提交时，首先会尝试将任务提交到线程池中的工作队列（`workQueue`）（如果核心线程空闲肯定直接拿过去了）

如果队列未满，任务会被放入队列，等待空闲的线程来取执行。

核心线程开了不会销毁了

如果 `workQueue` 已满且线程池的线程数小于最大线程数 (`maximumPoolSize`)，则会创建新的线程来处理任务。

要动态地去理解

人不多只有一个师傅 人多得不行就多一个师傅 师傅多了也不行 那就拒绝

第一次满了加线程 加了线程也满就丢弃了

线程池也会抢线程的

#### 四种拒绝策略

```
private static final RejectedExecutionHandler defaultHandler =  
    new AbortPolicy(); 默认拒绝策略    满了 还有人进来，不处理这个人 抛出异常
```

最大承载: Deque+max

`CallerRunsPolicy()` 哪里来的去哪里  
我的实验代码中 去main处理了

`DiscardPolicy()` //队列满了 不抛异常 丢掉任务

`DiscardOldestPolicy()` //队列满了 尝试和第一个竞争 不行 还是丢掉 不抛异常

```
public static class DiscardOldestPolicy implements  
RejectedExecutionHandler {  
    /**  
     * Creates a {@code DiscardOldestPolicy} for the given  
    executor.  
     */  
    public DiscardOldestPolicy() { }  
  
    public void rejectedExecution(Runnable r,  
ThreadPoolExecutor e) {  
        if (!e.isShutdown()) {  
            e.getQueue().poll();  
            e.execute(r);  
        }  
    }  
}
```

```
}
```

## 常见的阻塞队列

### blockingQueue的实现

**ArrayBlockingQueue** 数组的队列 FIFO（有界）（一把锁 锁数组）

**LinkedBlockingQueue** 链表的队列 FIFO（默认无解，可以有界（最好有界防止OOM）） 有两把锁（头尾锁 效率高）

**DelayedWorkQueue** 在任务的处理上有延迟机制，只有当任务的延迟时间达到后，任务才能从队列中被取出并执行 最先到期的任务最先被执行。

**SynchronousQueue** 理解成容量为1的队列（实际上不存元素），但是没有空闲线程，提交任务的线程会被阻塞，直到有线程空闲来消费该任务。

## 如何确定核心线程数

根据项目最大流量来定吧。。。。

IO密集： $2N+1$  (N为核数) （基本这个）

CPU密集型： $N+1$

高并发任务执行时间短： $N+1$  （减少线程上下文切换）

并发高、业务执行时间长要不要缓存和加服务器

## 线程池的种类？

常见四种 Executors库

**FixedThreadPool** 固定数量 **SingleThreadPool** 单一线程

以上底层里面没有给阻塞队列设置值（默认MAX VALUE） 所以会OOM 再次重提

**CachedThreadPool**: 只用临时线程（可能会导致反复新开和销毁） 也会无限新开 队列是 **SynchronousQueue**

允许创建线程数量为`integer.Max_value`，可能会创建大量的线程 导致OOM

**ScheduledThreadPool**:可延迟执行的（底层用延迟队列） 问题不大

## 使用场景（建议结合**WST**）

▮ *CountDownLatch* 倒计时锁

阻塞 线程完成调用`countdown()` 直到`countdown`值为0

结合**WST**来看

▮ *Future*

**Future** 是一个接口，表示异步计算的结果，它用于获取执行的结果、检查任务状态、取消任务等。

**Future** 的常用方法包括：**get()**（获取结果）、**cancel()**（取消任务）、**isDone()**（检查任务是否完成）等。

## 信号量

**Semaphore**

资源数量 用于限流

**acquire**请求（成功则减一）和**release**释放

满了就等待 等到被释放为止



release放在 finally 跟Lock解锁是一样的 放finally

## ThreadLocal

处理 线程局部变量 的类。它为每个线程提供了一个独立的变量副本，使得每个线程都能独立地操作自己的一份数据，而不与其他线程的数据发生冲突。

set方法 存线程自己变量

get方法获取对应线程的值

remove清除

底层？

每个线程持有一个Map对象

`Thread` 类内部有一个 `threadLocals` 字段，它指向该线程的 `ThreadLocalMap`。这个 `threadLocals` 字段是 `Thread` 的一个私有成员。

**ThreadLocal 键：**每个 `ThreadLocal` 对象在 `ThreadLocalMap` 中充当键。它唯一标识了线程局部变量。通过这个 `ThreadLocal` 对象，我们可以访问和修改线程局部的变量。

**ThreadLocal** 的设计是每个线程在自己的 `ThreadLocalMap` 中存储一个局部的变量副本。当线程第一次访问某个 `ThreadLocal` 变量时，它会在 `ThreadLocalMap` 中查找这个变量，如果没有找到就调用 `initialValue()` 方法创建并初始化它。

每个`Thread`对象中都存在一个`ThreadLocalMap`，Map的key为`ThreadLocal`对象，Map的value为需要缓存的值

key为`ThreadLocal`对象可以确保线程局部变量是根据每个 `ThreadLocal` 实例来存储和检索的。

key为`ThreadLocal` 本身跟类型有关 所以有不同的`ThreadLocal` 实例

## 强软弱虚引用

强应用：普通的new对象 **SoftReference**类

软引用：JVM 认为内存不足时，才会回收这些对象。

弱引用：弱引用的对象（**WeakReference**类）弱引用所指向的对象在垃圾回收时会立即被回收。

虚引用：**Phantom Reference** 虚引用不能单独使用，它的存在仅用于跟踪对象是否已经被垃圾回收。在对象即将被垃圾回收时，虚引用会收到通知。

**ThreadLocalMap**的entry对象继承了软引用 key为使用弱引用的**ThreadLocal**实例 会清楚但是value为强引用 不会清除 导致内存泄漏