Java并发编程实战

目录

[Java并发编程实战 1](#_Toc484607433)

[一、 线程安全性 4](#_Toc484607434)

[1.1 活跃性问题 4](#_Toc484607435)

[1.2 会另起线程的代码 4](#_Toc484607436)

[1.3 主要同步机制 4](#_Toc484607437)

[1.4 可变的状态变量出现线程安全问题 4](#_Toc484607438)

[1.5 竞态条件 4](#_Toc484607439)

[1.6 内置锁重入 4](#_Toc484607440)

[1.7 Synchronized 实现可见性 4](#_Toc484607441)

[1.8 在读写共享变量时都需要使用同步 5](#_Toc484607442)

[1.9 原子变量与同步块 5](#_Toc484607443)

[1.10 对象锁个数 5](#_Toc484607444)

[二、 对象的内存 5](#_Toc484607445)

[2.1 最低安全性 5](#_Toc484607447)

[2.2 Volatile可见性 6](#_Toc484607448)

[2.3 Jvm优化与多线程调试问题 –server 6](#_Toc484607449)

[2.4 发布与逸出 6](#_Toc484607450)

[2.5 Ad-hoc线程封闭 7](#_Toc484607451)

[2.6 栈封闭 7](#_Toc484607452)

[2.7 ThreadLoad类 8](#_Toc484607453)

[2.8 不可变对象 8](#_Toc484607454)

[三、 对象的共享 8](#_Toc484607455)

[3.1 正确构造对象的安全发布方式 8](#_Toc484607457)

[3.2 静态初始化对象引用 8](#_Toc484607458)

[3.3 对象可变性与发布 8](#_Toc484607459)

[3.4 使用和共享对象 9](#_Toc484607460)

[四、 对象的组合 9](#_Toc484607461)

[4.1 设计线程安全类 9](#_Toc484607464)

[4.2 Java基础容器类同步 9](#_Toc484607465)

[4.3 Java监视器模式 9](#_Toc484607466)

[4.4 线程安全性的委托 9](#_Toc484607467)

[4.5 客户端加锁 10](#_Toc484607468)

[五、 基础构建模块 10](#_Toc484607469)

[5.1 同步容器vs 并发容器 10](#_Toc484607471)

[5.2 同步工具类 10](#_Toc484607472)

[5.3 栅栏 11](#_Toc484607473)

[P84线程CPU数与吞吐量 11](#_Toc484607474)

[六、 结构化并发应用程序 12](#_Toc484607475)

[6.1 Executor 12](#_Toc484607477)

[6.2 Executor生命周期 12](#_Toc484607478)

[6.3 延迟任务与周期任务 12](#_Toc484607479)

[6.4 CompletionService 12](#_Toc484607480)

[6.5 为任务设计时限 13](#_Toc484607481)

[七、 取消与关闭 13](#_Toc484607482)

[7.1 Callable 13](#_Toc484607484)

[7.2 Interrupt 13](#_Toc484607485)

[7.3 Interrupted 13](#_Toc484607486)

[7.4 中断策略 14](#_Toc484607487)

[7.5 不可中断阻塞 14](#_Toc484607488)

[7.6 newTaskFor自定义中断 14](#_Toc484607489)

[7.7 异常处理 15](#_Toc484607490)

[7.8 jvm关闭钩子 15](#_Toc484607491)

[7.9 守护线程 15](#_Toc484607492)

[八、 线程池的使用 15](#_Toc484607493)

[8.1 ThreadLocal不适用于线程池 15](#_Toc484607495)

[8.2 饥饿死锁 15](#_Toc484607496)

[8.3 限制运行时间 16](#_Toc484607497)

[8.4 线程池大小 16](#_Toc484607498)

[8.5 无界队列 16](#_Toc484607499)

[8.6 有界队列 16](#_Toc484607500)

[8.7 同步移交 17](#_Toc484607501)

[8.8 饱和策略 17](#_Toc484607502)

[8.9 PrivilegedThreadFactory设置安全策略 17](#_Toc484607503)

[8.10 扩展ThreadPoolExextor 17](#_Toc484607504)

[九、 避免活跃性危险 18](#_Toc484607505)

[9.1 锁顺序死锁 18](#_Toc484607507)

[9.2 开放调用 18](#_Toc484607508)

[9.3 tryLoack定时取锁 19](#_Toc484607509)

[9.4 线程转储Thread Dump识别死锁 19](#_Toc484607510)

[9.5 饥饿Starvation 19](#_Toc484607511)

[9.6 活锁livelock 19](#_Toc484607512)

[十、 性能与可伸缩性 19](#_Toc484607513)

[10.1 Amdahl定律 19](#_Toc484607515)

[10.2 上下文切换 19](#_Toc484607516)

[10.3 内存同步 20](#_Toc484607517)

[10.4 阻塞 20](#_Toc484607518)

[10.5 降低锁的竞争程度 20](#_Toc484607519)

[10.6 减小锁的粒度 20](#_Toc484607520)

[10.7 避免热点域 21](#_Toc484607521)

[10.8 替代独占锁 21](#_Toc484607522)

[10.9 检测CPU利用率 21](#_Toc484607523)

[10.10 并发不要使用对象池 21](#_Toc484607524)

[十一、 并发程序的测试 21](#_Toc484607525)

[11.1 阻塞测试 21](#_Toc484607527)

[11.2 安全性测试 22](#_Toc484607528)

[11.3 资源管理测试 22](#_Toc484607529)

[11.4 更多交替操作 22](#_Toc484607530)

[11.5 性能测试陷阱 22](#_Toc484607531)

[11.5.1 垃圾回收 22](#_Toc484607532)

[11.5.2 动态编译 22](#_Toc484607533)

[11.5.3 代码路径不真实采样 22](#_Toc484607534)

[11.5.4 无用代码消除 23](#_Toc484607535)

[11.6 常见并发错误 23](#_Toc484607536)

[11.6.1 在构造函数中启动一个线程 23](#_Toc484607537)

[11.6.2 条件等待中的错误 23](#_Toc484607538)

[11.6.3 在休眠或等待的同时持有一个锁 23](#_Toc484607539)

[十二、 显式锁 23](#_Toc484607540)

[12.1 ReentrantLock 23](#_Toc484607542)

[12.2 轮询锁与定时锁 23](#_Toc484607543)

[12.3 公平性 24](#_Toc484607544)

[12.4 Synchronized VS ReentrantLock 24](#_Toc484607545)

[12.5 读写锁 24](#_Toc484607546)

[十三、 构建自定义的同步工具 24](#_Toc484607547)

[13.1 状态依赖性问题 24](#_Toc484607549)

[13.2 条件谓词 24](#_Toc484607550)

[13.3 通知 25](#_Toc484607551)

[13.4 子类安全问题 25](#_Toc484607552)

[13.5 入口协议与出口协议 25](#_Toc484607553)

[13.6 显示Condition 25](#_Toc484607554)

[13.7 AQS 26](#_Toc484607555)

[十四、 原子变量与非阻塞同步机制 26](#_Toc484607556)

[14.1 并发CAS 26](#_Toc484607558)

[14.2 原子变量类 26](#_Toc484607559)

[14.3 非阻塞算法 26](#_Toc484607560)

[14.4 ABA问题 26](#_Toc484607561)

[十五、 Java内存模型 27](#_Toc484607562)

[15.1 Happens-before 27](#_Toc484607564)

[15.2 不安全发布 27](#_Toc484607565)

# 线程安全性

## 活跃性问题

当某个操作无法执行下去时,就会发生活跃性问题,如死锁,饥饿,活锁等

## 会另起线程的代码

Timer, servlet/JSP,RMI,swing/AWT

## 主要同步机制

1. Synchronized
2. Volatile
3. 显式锁(Explicit Lock)
4. 原子变量

最好将一个有关联的同步操作放在同一个线程安全类中,由一个类提供入口,在类中做好同步措施,客户端调用类时不需要再考虑同步问题,比如concurrent包下的类

## 可变的状态变量出现线程安全问题

1. 不在线程之间共享该变量
2. 将状态变量改为不可变的变量
3. 在访问状态变量时使用同步

## 竞态条件

当计算的正确性取决于多个线程的交替执行顺序时，就会发生竞态条件

常见的竞态条件是“先检查后执行”(check then act)，检查和执行的间隙被其他线程隔断，发生错误

应尽量使用线程安全的类来管理类的状态,如原子类

当状态变量有多个且相互关联时,单纯的原子类已经不够用了,应使用同步代码管理,此时可以不用原子变量了

## 内置锁重入

每个java对象可以用作同步的锁,称为内置锁

内置锁是可重入的,因此,如果某线程试图获取一个已经由它自己持有的锁,那这个请求会成功.重入意味着获取锁操作的粒度是线程,而不是调用

例如子类的synchronized方法内部是可以调度父类的synchronized方法的

## Synchronized 实现可见性

线程执行互斥代码的过程

1. 获得互斥锁
2. 清空工作内存
3. 从主内存拷贝变量的最新副本到工作内存
4. 执行代码
5. 将更新后的共享变量的值刷新到主内存
6. 释放互斥锁

## 在读写共享变量时都需要使用同步

对于可能被多个线程同时访问的可变状态变量，在访问时需要持有同一个锁，状态变量由这个锁保护

## 原子变量与同步块

若原子变量的操作已经在同步代码块内，则可放弃使用原子变量，普通变量更好——不同的同步机制容易造成混乱

## 对象锁个数

某个线程在获得对象的锁之后，只能阻止其他线程获得同一个锁。之所以每个对象都有一个内置锁，只是为了免去显示创建锁对象。

一个实例就只有一个锁？一个类有一个锁？类实例化的所有对象共享同一个锁？

锁分为对象锁和类锁

* 对象锁

是用于对象实例方法，或者一个对象实例上的，不同对象实例的对象锁是互不干扰的

* 类锁

是用于类的静态方法或者一个类的class对象上的，但是每个类只有一个类锁，其实类锁只是一个概念上的东西，并不是真实存在的，它只是用来帮助我们理解锁定实例方法和静态方法的区别的

* 对象锁和类锁互不干扰

一个线程获得对象锁，另一个线程同时也可以获得类锁，可以交替执行

* 多个对象锁不关联

一个类的对象锁和另一个类的对象锁是没有关联的，当一个线程获得A类的对象锁时，它同时也可以获得B类的对象锁。

Collection.Synchronized()方法

# 对象的内存



## 最低安全性

Java内存模型要求，变量读写必须是原子操作，当无线程同步时，读取到的变量值必为某个线程设置的值，而不是一个随机值

但对于非volatile类型的long和double变量，jvm允许将64位的读/写操作分解为两个32位的操作，这就有可能造成高32位和低32位不是原组合的问题

解决方法：用volatile修饰或者用锁保护起来

## Volatile可见性

Volatile修饰的变量操作不会与其他内存操作一起重排序，volatile变量不会被缓存在寄存器或者其他处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值

* 使用场景

1. 对变量的写入操作不依赖变量的当前值，或者只有单个线程更新变量值
2. 该变量不会与其他状态变量一起纳入不变性条件中
3. 在变量访问时不需要加锁

Example

Volatile boolean asleep;

…

While(asleep)

CountSomeSleep()

若不使用volatile，可能当asleep被一个线程修改时，执行判断的线程修改不了，此时用volatile笔锁机制简单方便

但是，volatile只保证可见性，而不保证原子性，如volatile不保证count++的原子性，但锁机制可以

可以使用volatile类型来发布不可变对象P40

## Jvm优化与多线程调试问题 –server

对于服务器应用程序，在开发和测试阶段都，启动jvm时都要指定-server命令行选项，server模式的jvm将比client模式的jvm进行更多的优化，例如将循环中未被修改的变量提升到循环外部，可能导致开发环境(client模式的jvm)正常的代码到部署环境(server模式的jvm)发生异常

在Volatile示例代码中，若asleep未声明为volatile类型，那么server模式的jvm会将asleep的判断条件提升到循环体外部，而client模式的jvm不会这么做

## 发布与逸出

* 发布

对象能够在当前作用域之外的代码中使用(其他对象可以引用该对象)

* 逸出

当某个不应该被发布的对象被发布时，就是逸出

当在对象构造完成之前发布该对象到其他线程，就会破坏线程安全性，特别的，当从对象的构造函数中发布对象时，只是发布了一个尚未构造完成的对象(即this未初始化完成，但你this却可以被构造函数中新发布(实例化)的对象引用)

不要在构造过程中使用this引用逸出

Example

Wrong

public class ThisEscape {

　　public ThisEscape(EventSource source) {

　　　　source.registerListener(new EventListener() {

public void onEvent(Event e) {

　　　　　　　　doSomething(e);

　　　　　　}

　　　　});

}

Correct

public class SafeListener {

　　private final EventListener listener;

　　private SafeListener() {

　　　　listener = new EventListener() {

　　　　　　public void onEvent(Event e) {

　　　　　　　　doSomething(e);

　　　　　　}

　　　　};

　　}

　　public static SafeListener newInstance(EventSource source) {

　　　　SafeListener safe = new SafeListener(); //先构造完成

　　　　source.registerListener(safe.listener); //再发布

　　　　return safe;

　　}

## Ad-hoc线程封闭

指线程封闭性的职责完全由程序实现来承担,不共享数据,仅在单线程内访问数据,将对象封闭到目标线程上;因其脆弱性，应该尽量少用它，应使用更强的线程封闭技术，如栈封闭或threadlocal类

## 栈封闭

变量只存在于执行线程栈中，只在线程内部使用

如果在线程内部上下文中使用非线程安全的对象,那么该对象仍然是线程安全的

## ThreadLoad类

类可以将ThreadLoad<T>视为Map<Thread ,T>

Threadlocad提供了set get方法，这些方法为使用该变量的线程都保存一份独立的副本，因此get总是返回当前执行线程在调用set时设置的最新值

## 不可变对象

* 对象不可变性

1. 状态不可修改
2. 所有域都是final类型
3. 正确的构造过程

任何线程都可以在不需要同步的情况下安全地访问不可变对象，及时发布这些对象时没有使用同步

如果final类型的域指向的对象时可变对象,那么就是引用不可变,但可变对象的状态是可变的,此时访问对象状态时仍需同步

# 对象的共享



## 正确构造对象的安全发布方式

1. 在静态初始化函数中初始化一个对象引用
2. 将对象的引用保存到volatile类型的域或者AtomicReferance对象中
3. 将对象引用保存到某个正确构造对象的final域中
4. 将对象引用保存到一个由锁保护的域中

可以将对象放入到线程安全的容器中:

Hashtable synchronizedMap concurrentMap vector copyOnWriterArrayList copyOnWriterSet synchronizedList synchronizedSet blockingQueue concurrentLinkedQueue

## 静态初始化对象引用

静态初始化器由jvm在类的初始化阶段执行，由于在jvm内部存在这同步机制，故对象可以被安全地发布

Public static Hodlder holder = new Hodeler(42);

## 对象可变性与发布

不可变对象可以通过任意机制发布

事实不可变对象必须通过安全方式发布

可变对象必须通过安全方式发布，并且必须是线程安全的或由锁保护起来

## 使用和共享对象

* 线程封闭

对象只能由一个线程拥有，被封闭在线程中

* 只读共享
* 线程安全共享

线程安全对象在其内部实现同步，多个线程可以通过对象的共有接口访问而无需进一步同步

* 保护对象

被保护对象只能通过持有特定的锁来访问，保护对象包括封装在其他线程安全对象中的对象，以及已发布的由某个特定锁保护的对象

# 对象的组合



## 设计线程安全类

1. 找出构成对象状态的所有变量
2. 找出约束状态变量的不变性条件
3. 建立对象状态的并发访问策略

## Java基础容器类同步

一些java基础同步类并不是线程安全的,但可以通过包装器工厂方法(collections.synchronizedList()),将容器类封装在一个同步的包装器对象中

## Java监视器模式

把对象的所有可变状态都封装起来，并用对象自己的内置锁来保护

如vector和hashtable

使用对象私有的锁(private)可能更有优点

同时，在获取被保护的对象时，可以返回复制对象，修改对象时通过保护对象共有方法修改即可(不是直接修改返回的复制对象)

## 线程安全性的委托

如果一个类是由多个独立且线程安全的状态变量组成，并且在所有的操作中都不包括无效状态转换，则可将线程安全性委托给底层的状态变量

线程安全可以将状态变量赋予线程安全的类来管理，比如线程安全容器，不可变容器，原子类等

涉及线程安全的变量，尽量设为final类型

返回引用时，特别需要注意是否会造成逸出，可以返回复制对象，或者不可变对象(对象本身不可变（注意是否可以修改引用），不可变容器，同步容器)

## 客户端加锁

需要同步的对象可以放到客户端中同步，需要注意同步时加锁同一个锁

如vector为同步类，其内部方法操作是同步的，但涉及几个操作按序同步执行时，可以在客户端加锁实现，此时，所加的锁应与vector对象原本的锁一致，即vector对象自身

Synchronized(vector){ … }

# 基础构建模块



## 同步容器vs 并发容器

* 同步容器

Vector、Hashtable、同步封装类，可以由Collections.synchronizedXxxx等方法创建

同步容器类虽然都是线程安全的，但是在某些情况下（复合操作），仍然需要加锁来保护；

同步容器对所有容器状态的访问都串行化，严重降低了并发性；当多个线程竞争锁时，吞吐量严重下降；

* 并发容器

java5.0之后提供了多种并发容器来改善同步容器的性能，如ConcurrentHashMap、CopyOnWriteArrayList、CopyOnWriteArraySet、ConcurrentSkipListMap等

**以ConcurrentHashMap为例**

采用分离锁技术，同步容器中，是一个容器一个锁，但在ConcurrentHashMap中，会将hash表的数组部分分成若干段，每段维护一个锁，以达到高效的并发访问；

迭代器弱一致性，迭代期间不会抛出ConcurrentModificationException异常；

size()、isEmpty()等方法返回的是一个近似值；

增加了若干原子操作方法，如putIfAbsent（没有该key，则添加）

注意，此时不能再通过客户端加锁新建新的原子操作了，客户端只能对并发容器自身加锁，但并发容器内部使用的并不是自身锁

写入时复制容器：Copyonwrite，在每次修改时都会创建并重新发布一个新的容器副本，从而实现可见性

* 总结

只有在应用程序需要对容器加锁进行独占式访问时，才用同步容器，否则使用非并发容器以保证更优性能

## 同步工具类

同步工具类可以根据自身状态来协调线程的控制流

* 阻塞队列blockingQueue

不仅能作为保存对象的容器，还能协调生产者和消费者等线程之间的控制流，因为take和put方法将阻塞

阻塞方式：(以put()方法为例)

1. 在构建blockingQueue对象时,创建lock及对应的Condition

lock = new ReentrantLock(fair);

Condition notEmpty = lock.newCondition();

Condition notFull = lock.newCondition();

1. put时,先lock,再在循环中判断是否为空,为空则在Condition上等待,直至被take等操作唤醒

lock.lockInterruptibly();

try {

while (count == items.length)

notFull.await();

insert(e);

} finally {

lock.unlock();

}

注意,必定要有循环,当被唤醒时,需要回到循环中再次做判断是否符合条件

* 闭锁

可以确保某些活动直到其他活动都完成后继续执行，一旦达到结束状态，将不会再更改状态

CountDownLatch：可以使一个或多个线程等待事件的发生，闭锁状态包括一个计数器countDown方法，当前线程调用此方法，则计数减一; await方法，调用此方法会一直阻塞当前线程，直到计时器的值为0

FutureTask:可生成结果的runnable，包括3种状态：等待运行、正在运行和运行完成。若任务已经完成，则future.get()会立即返回结果，否则阻塞直至完成状态.

FutureTask使用场景:用ConcurrentMap <A, Future<V>>缓存计算,vaule值是Future,P89

* 信号量

用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量

Semaphore管理着一组虚拟许可，执行操作时需先acquire获得许可，没有则阻塞直到有许可，使用后release释放许可

## 栅栏

用于等待其他线程，所有线程必须同时到达栅栏位置，才能继续执行

CyclicBarier可以使一定数量的参与方反复地在栅栏位置汇集，在并行迭代算法中非常有用

Exchanger 是一种两方栅栏，各方在栅栏位置上交换数据，用于双方执行不对称操作

### P84线程CPU数与吞吐量

在不涉及I/O操作或共享数据访问时，当线程数量为cpu数或CPU数+1 时，将获得最优的吞吐量。一个进程下的线程是如此，如果有多个进程呢？进程间的时间分配？

# 结构化并发应用程序



## Executor

提供了一种标准方法，将任务的提交过程和执行过程解耦，还提供了对生命周期的支持，以及统计信息收集，应用程序管理机制和性能监视等机制

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 线程数 | 异常 | 特性 |
| newfixedThreadPool | 固定长度 | 出现异常而结束则补充一个线程 | 逐步增加，直到最大 |
| newCachedThreadPool | 无限 |  | 可缓存线程池，线程池规模多于当前需求，则回收空闲线程,线程池可无限扩大 |
| newSingleThreadExecutor | 1 | 出现异常而结束则另起一个线程 | 单线程按优先级等顺序执行 |
| newScheduledThreadPool | 固定长度 |  | 以延迟或定时的方式执行 |

## Executor生命周期

三种状态：运行、关闭和已终止

四个生命周期阶段：创建，提交，开始和完成

已提交但尚未开始的任务可以取消,而已开始执行的任务,只有当它们能响应中断时才能取消

JVM只有在所有非守护线程全部终止后才会退出，如果无法正确关闭executor，那么jvm则无法结束

而关闭时有两种方式

1. 平缓关闭shutdown：停止接受新任务，执行完所有正在执行和在等待队列中的任务
2. 强制关闭shutdownNow：取消所有运行中的任务，不再启动等待队列中的任务

## 延迟任务与周期任务

Java.util.Timer类在执行所有定时任务时只会创建一个线程，若某个任务执行时间过长，那么将破坏其他TimerTask的定时精确性

Timer若抛出异常，则会取消所有timer类下的定时线程，不会恢复执行

## CompletionService

用于提交一组计算任务，并获得计算结果

使用future，需要保留每个任务关联的future，反复使用get方法，同时将参数timeout设为0，通过轮询判断任务是否完成

CompletionService：将executor和blockingqueue的功能融合在一起，将callable任务交给提交给他来执行，然后使用类似于队列操作的take和poll等方法获得future，再future.get()返回结果

声明：

CompletionService<T> completionService = new ExecutorCompletionService<T>(executor);

执行一个计算

CompletionService.submit(new Callable<T>(){

public T call(){ return …; }

})

获取一个返回结果

Future <T> f = completionService.take();

T t = f.get();

## 为任务设计时限

Future为一个任务设计时限：时限内有结果，get立即返回，超过时限抛出TimeOutException

Future.get(long,timeType)

提交一组任务

InvokeAll：将多个任务提交到一个ExecutorService并获得结果，InvokeAll按照任务集合中迭代器的顺序添加到返回集合，由此可关联各个future与callable

# 取消与关闭

Java没有提供任何机制来安全地终止线程，而是提供了中断(interrupion)，能使一个线程终止另一个线程的当前工作



## Callable

Callable认为主入口点将返回一个值，并可能抛出一个异常

无返回值，可使用Callable<Void>

## Interrupt

调用interrupt并不意味着立即停止目标线程正在进行的工作，而只是传递了请求中断的消息，然后由线程在下一个合适的时刻中断自己

通常，中断是实现取消最合理的方式，而不是设置标志位

## Interrupted

获得中断状态,并清除当前线程的中断状态.

在调用interrupted时返回了true,则会清除线程中断状态,下次再调用interrupted时就已经不是中断状态了,故需要对中断做处理—抛出interruptException或者再次调用interrupt恢复中断状态:Thread.currentThread().interrupt();

## 中断策略

最合理的取消操作是某种形式的线程级取消操作或服务级取消操作：尽快退出，在必要时进行清理，通知某个所有者线程已经退出

线程应该只能由其所有者中断，所有者可以将线程的中断策略信息封装到某个合适的取消机制中，例如shutdown方法

* 基本中断策略

传递interruptedException:将interruptedException传递给方法的调用者

恢复中断：当前代码是runnable一部分时，不能抛出异常，必须捕获异常，并通过调用当前线程上的interrupt方法恢复中断状态，这样在调用栈中更高层的代码将看到引发了一个异常

public void run() {

try {

processTask(queue.take());

} catch (InterruptedException e) {

// restore interrupted status

Thread.currentThread().interrupt();

}

}

当尝试取消某个任务时，不宜直接中断线程池，只能通过任务的future来实现取消

## 不可中断阻塞

在如socket I/O或者等待获得内置锁而阻塞时，那么中断请求只能设置线程的中断状态，除此之外并无多大作用。此时应该中断底层的阻塞操作，抛出异常，以此响应中断

Example：

Socket读取阻塞

public void interrupt() {

try {

socket.close();

} catch (IOException ignored) {

} finally {

super.interrupt();

}

}

## newTaskFor自定义中断

P123 重新封装cancel方法

可以中断线程，也可以取消底层阻塞方法

注意，在取消生产者-消费者操作时，需要同时取消生产者和消费者

* 方法重写

可以设置一个Boolean flag标识是否取消。同时设置一个计数器统计当前任务队列中任务数量，关闭时设置flag，中断线程，而底层的生产者方法就判断flag是否已关闭，抛出异常，消费者则只在flag和计数器值为0时取消，否则一直处理任务队列，直到完成所有任务。

* 毒丸对象

往任务队列中添加约定的对象，消费者每次都查看对象，判断是否退出

也可以再做一个统计，达到数量才退出，这样就可以确保取消多个线程

注意：只有在无界队列中，毒丸对象才能可靠地工作

## 异常处理

导致线程提前死亡的最主要原因就是runtimeException，在线程代码中可以使用try-catch代码块捕获异常并进行处理

未捕获异常

UncaughtExceptionHandler,Thread API中提供的处理异常类，能检测出某个线程由于未捕获的异常而终结的情况，至少将异常信息打印到日志表中。需要为ThreadPoolExecutor的构造函数提供一个ThreadFactory

## jvm关闭钩子

Shutdown hook:正常关闭jvm时，jvm首先调用所有已经注册好的关闭钩子，指通过Runtime.addShutdownHook()注册但未开始的线程

Jvm并不会停止或者中断任何在关闭时仍然运行的应用程序线程，当jvm最终结束时，守护线程将被强行结束

## 守护线程

普通线程：主线程创建的所有线程都是普通线程，普通线程继承了创建它的线程的守护状态

守护线程：非普通线程，当一个线程退出时，jvm会检查正在运行的线程，若都是守护线程，则jvm退出，当jvm停止时，所有的守护线程将被抛弃，直接退出

守护线程最好执行“内部任务”

# 线程池的使用



## ThreadLocal不适用于线程池

只有当线程本地值得生命周期受限于任务的生命周期时,在线程池的线程中使用threadlocal才有意义,而在线程池的线程中不应该使用threadlocal在任务之间传递值

## 饥饿死锁

只要线程池中的任务需要无限期地等待一些必须由池中其他任务才能提供的资源或条件，除非线程池足够大，否则将发生线程饥饿死锁

每当提交一个有依赖性的executor任务时，需要知道可能会出现线程饥饿死锁，故而需要在代码或配置executor的配置文件中记录线程池的大小限制或配置限制

只有当任务相互独立时,为线程池工作队列设置界限才是合理的,如果任务之间存在依赖性,那么有界的线程池或队列就可能导致线程”饥饿死锁”问题,此时应该使用无界的线程池,例如newCachePool

## 限制运行时间

可阻塞方法大都定义了限时版本和不限时版本，如Thread.jion, blockingQueue.put, countDownLatch.await, selector.select等。若等待超时，可以把任务标识为失败，然后终止任务或者将任务重新放回队列以便随后执行

## 线程池大小

线程池大小不应该固定，应该通过配置机制提供,或者根据Runtime.getRuntime().availableProcessors()来动态计算

如果需要执行不同类别的任务，并且它们之间的行为相差很大，那么应该考虑使用多个线程池，从而使每个线程池可以根据各自的工作负载来调整

计算密集型：线程池大小为CPU数+1

I/O操作或其他阻塞操作：线程并不会一直执行，规模应该更大，需要估算任务的等待时间与计算时间的比值

要使处理器达到期望的使用率，线程池的最优大小等于：

可以通过runtime来获得CPU数目

int cpu = Runtime.getRuntime().availableProcessors();

threadPoolExecutor允许提供一个BlockingQueue来保存等待执行的任务,基本的任务排队方法有3种:无界队列,有界队列和同步移交

## 无界队列

工厂方法newFixedPoolExecutor , newSingleThreadExecutor在默认情况下使用一个无界的linkedBlockedQueue

## 有界队列

ArrayBlockingQueue,有界的linkedBlockingQueue、PriorityBlockingQueue，有界队列有助于避免资源耗尽的情况发生

## 同步移交

对于非常大或者无界的线程池，可以通过synchronousQueue来避免任务排队，以及直接将任务从生产者交给工作者线程

若没有线程正在等待接受任务，并且线程池未满，则新创建线程接受任务，否则根据饱和策略，这个任务将被拒绝

NewCachedThreadPool工厂方法使用了synchronousQueue

只有当线程池是无界的或者可以拒绝任务时, synchronousQueue才有价值

## 饱和策略

当有界队列被填满后，或者Executor已关闭，饱和策略开始发挥作用，ThreadPoolExextor的饱和策略可以通过调用setRejectedExcutionHandler修改

不同的RejectedExcutionHandler实现

* 中止Abort

默认的饱和策略，抛出未检查的RejectedExcutionHandlerException

* 抛弃discard

当新提交的任务无法保存到队列中等待执行时，则悄悄抛弃该任务

* 抛弃最旧的discard-oldest

抛弃下一个被执行的任务，然后尝试重新提交任务，若工作队列市是优先队列，则将抛弃优先级最高的任务，不妥

* 调用者运行Caller-Runs

将任务回退到调用者，由调用者运行，从而占用调用者线程，降低新任务流量

Webserver中,在主线程由于回退执行任务时，新到达的请求将保存在TCP层的队列中而不是在应用程序的队列中，若持续过载，TCP队列满，会向调用者抛弃请求，逐级层层上抛，最终到达客户端，实现平缓的性能降低

threadPoolExecutor.setRejectedExecutionHandler(new ThreadPoolExecutor.DiscardPolicy());

## PrivilegedThreadFactory设置安全策略

可以控制对某些特殊代码库的访问权限，所创建的线程将与创建PrivilegedThreadFactory的线程拥有相同的访问权限、AccessControlContext和contextClassLoader

若不使用PrivilegedThreadFactory，新建的线程将从调用exectute或submit的客户程序中继承访问权限

## 扩展ThreadPoolExextor

提供子类改写方法beforeExecute, afterExecute, terminated

Run方法返回或抛出异常，afterExecute都会执行，若有error，则不会执行afterExecute

若beforeExecute抛出RuntimeException异常,则任务不被执行，且afterExecute也不会被调用

# 避免活跃性危险



## 锁顺序死锁

线程需要以固定一致的顺序获取锁

需要注意的是，虽然对象引用顺序是固定的，但在两次加锁时其实际对象是交换的，这实际上就不是固定顺序加锁，容易导致死锁

加锁时可以以唯一,不可变的值作为加锁的排序依据,比如账号,id等

在制定锁的顺序时，可以使用system.identityHashCode()获取对象hashcode值，以hashcode值为顺序加锁，又对象可能有相同hashcode值，那么可以使用加时赛锁，即当判断hashcode值一样时，就对加时赛锁上锁，然后再以一个固定顺序上锁

如果在持有锁的情况下调用某个外部方法，要检查被调用外部方法是否也有同步操作，避免出现死锁问题

## 开放调用

开放调用：在调用某个方法时不需要持有锁

在程序中应该尽量使用开放调用，更加容易进行死锁分析

死锁

**Class A**

public synchronized void setLocation(Point location) {

...

if (location.equals(destination))

dispatcher.notifyAvailable(this); //方法调用也加锁

}

**class B**

public synchronized void notifyAvailable(Taxi taxi) {

availableTaxis.add(taxi);

}

开放调用

public void setLocation(Point location) {

boolean reachedDestination; //新增中间变量

synchronized (this) {

…

reachedDestination = location.equals(destination);

}

if (reachedDestination)

dispatcher.notifyAvailable(this);

}

## tryLoack定时取锁

可以指定一个超时时限，在超时后会返回一个失败信息

## 线程转储Thread Dump识别死锁

Jvm通过线程转储帮助识别死锁的发生，线程转储包括各个运行中的线程的栈追踪信息，加锁信息(每个线程有哪些锁，那些帧栈获得这些锁，被阻塞的线程正在等待获取哪一个锁)

线程转储前，jvm将在等待关系图中通过搜索循环来找出死锁，若发现死锁，则获取相应死锁信息

## 饥饿Starvation

线程由于无法访问它所需要的资源而不能继续执行时，就发生了饥饿，引发饥饿最常见的资源就是CPU时钟周期

更改线程优先级且使用不当，或者在持有锁时执行一些无法结束的结构（无限循环，无线等待etc）

要避免使用线程优先级，因为这会增加平台依赖性，并可能导致活跃性危险

## 活锁livelock

线程不断重复执行相同操作，但总是失败。如在事务消息处理中，若不能成功地处理某个消息，那么消息处理机制将回滚整个事务，并将它重新放到队列的开头

当多个相互协作的线程都对彼此进行响应从而修改各自的状态，并让线程无法继续执行，如行人同时互相让路，此时引入随机数，可以避免问题

# 性能与可伸缩性

避免不成熟的优化，首先使程序正确，然后再提高运行速度——如果运行不够快



## Amdahl定律

在增加计算资源的情况下，程序在理论上能够实现的最高加速比，取决于程序中可并行组件与串行组件的比重

F： 串行执行比率

N：处理器个数

## 上下文切换

线程调度过程中需要访问由操作系统和jvm共享的数据结构，其开销包括jvm和操作系统代码执行开销，同时，由于新线程切换进来时，它所需要的数据结构可能不在当前处理器本地缓存中，故还有缓存切换开销

Jvm可以将阻塞的线程挂起并允许它被交换出去，当线程频发发生阻塞，则CPU密集型的程序就会发生越多的上下文切换，从而增加调度开销，降低吞吐量

在大多数通用的处理器中，上下文切换的开销相当于5000~10000个时钟周期，几微秒

Unix系统的vmstat命令和windows perform工具能够报告上下文切换次数及在内核中执行时间所占比例等信息。若内核占用率超过10%，那么通常表示调度活动频繁，可能是由I/O或者竞争锁导致的阻塞造成的。

## 内存同步

* 内存栅栏Memory Barrier

Synchronized和volatile 提供的可见性保证中可能使用内存栅栏，可以刷新缓存，刷新硬件的写冲突，以及停止执行管道。内存栅栏可能会抑制一些编译器优化操作，比如大多数操作不能被重排序

* jvm优化锁

现代jvm能通过优化去掉一些不会发生竞争的锁，从而减少不必要的同步开销

若锁只能由当前线程访问，如锁住的是线程内新建的对象，则jvm可以去掉锁获取操作

当jvm可以通过逸出分析找出不会发布到堆的本地对象引用，可以去掉锁获取操作，也可以锁粒度粗化

## 阻塞

Jvm在实现阻塞行为时，可以

1. 自旋等待(spin –waiting循环不断地尝试获取锁直到成功)
2. 通过操作系统挂起被阻塞的线程

其效率取决于上下文切换的开销以及在成功获得锁之前需要等待的时间。

等待时间短，则选择自旋等待，等待时间长，则选择线程挂起

阻塞时，有两次上下文切换，包括两次必要的系统操作和缓存操作：

1. 被阻塞的线程在其执行时间片还未用完时就被交换出去
2. 接替的线程被切换回来

## 降低锁的竞争程度

1. 减少锁的持有时间
2. 降低锁的请求频率
3. 使用带有协调机制的独占锁，这些机制允许更高的并发性

## 减小锁的粒度

* 锁分解

若一个锁要保护多个相互独立的状态变量，那么可以讲锁分解为多个锁，每个锁只保护一个变量，降低每个锁被请求的频率

* 锁分段

在锁上的竞争频率高于被锁保护的数据的竞争频率时，可以将锁分解技术进一步扩展成对一组独立对象上的锁进行分解，如ConcurrentHashMap包含16个锁的数组，每个锁保护所以散列桶的1/16，第N个散列桶由N mod 16来保护

劣势：获取锁的复杂度和开销更高，某些情况下需要加锁整个容器。如当ConcurrentHashMap需要扩展映射范围，以及重新键值的散列值要分布到更大的桶集合时，就需要获取分段锁集合中的所有锁

## 避免热点域

在ConcurrentHashMap中的size函数，并不是返回直接返回一个储存在map中的全局计数值，因为这会导致这个值成为热点值(每次增删操作都会修改，即使不是同一线程，会导致锁竞争)，而是每个分段独立维护各自分段的计数值，计算map size值是直接枚举分段计数值相加即可

## 替代独占锁

使用并发容器，读写锁，不可变对象、原子变量

## 检测CPU利用率

工具命令UNIX vmstat/mpstat, windows perfmom

CPU没有被充分利用原因

* 负载不均衡

增加测试负载

* I/O密集

通过iostat/ perfmom判断某个应用程序是否是I/O密集型的，或者通过监测网络的通信流量级别来判断是否需要提高带宽

* 外部限制

使用分析工具或数据库管理工具判断外部限制

* 锁竞争

被阻塞的线程将在线程转储信息中存在相应的帧栈，含有”waiting to lock monitor ….”，锁竞争越激烈，越频繁出现在线程转储中

## 并发不要使用对象池

线程从线程池中请求对象时若被阻塞，其阻塞开销将是内存分配操作(新建对象)的数百倍

另外，需要确保重新使用对象时要将对象重置到正确状态

# 并发程序的测试



## 阻塞测试

可以使用中断来解除阻塞，在主线程中启动含有阻塞操作的测试线程，此时测试线程阻塞中，在主线程中中断测试线程，测试线程抛出InterruptException，测试线程执行join操作，确保测试线程走完，然后当测试线程.isAlive()==false则表示阻塞测试成功

使用Thread.getState来验证线程能否在一个条件等待上阻塞，这并不可靠

## 安全性测试

测试由于数据竞争而引发的错误，需要多个线程分别执行put和take操作

关键问题是：找出容易检查的属性，且这些属性在发生错误的情况下极有可能失败，同时又不能使得错误检查代码人为地限制并发性

可以使用校验和计算函数来计算入列和出列的元素校验和，如果二者相等，代码正确。需要考虑是否顺序敏感

测试时，可以使用CyclicBarrier或者CountDownLatch来统一运行多线程测试程序，同时执行到同一位置，避免创建线程时导致的不同步问题

当抛出异常，或者无限循环时，测试可能永远不会结束，此时测试程序可以设置最大等待时间，过时不执行，后期再排查问题

测试线程数量应该多于CPU数量,则任意时刻都有线程在运行和被交换出去,增加交替行为

## 资源管理测试

不需要对象时，销毁对象引用

## 更多交替操作

1. 使用Thread.yield() 或 Thread.sleep(). (sleep 会好一些) 使用AOP提高方便性
2. 使用产生更多数量的活动线程,至少高于处理器数量

## 性能测试陷阱

### 垃圾回收

* 不执行垃圾回收

确保垃圾回收操作在整个测试运行期间不运行 –verbose:gc

* 多次执行垃圾回收

充分反映出运行期间内存分配与垃圾回收等开销

### 动态编译

当某个类第一次被加载时，JVM通过解释字节码的方式执行，而热点代码在运行中可能会被动态编器编译成机器代码，则代码将热点代码变为直接执行；代码也可能被退回解释执行，重新编译

* 降低解释/编译时间占比

让测试程序运行足够长的时间

* 避开解释/编译时间

使用-XX:+PrintCompilation，当动态编译时会输出信息，验证动态编译是在测试运行前

### 代码路径不真实采样

动态编译器可能针对一个单线程测试程序进行一些专门优化，但只要在真实的应用程序中包含一些并行，都会使这些优化不复存在——将单线程性能测试与多线程性能测试结合在一起

### 无用代码消除

HotSpot中，-server模式比-client模式更好，-server模式编译器能产生更有效的代码，而且这种模式更易于通过优化消除无用代码

* 避免运算被优化且不引起过高开销方法

计算某个对象中域的散列值，并将它与任意值进行比较，如System.nanoTime的当前值，若相等，则输出无用且可悲忽略的消息

if ( f.x.hashCode() == System.nanoTime() ) {

System.out.println(" ");

}

## 常见并发错误

### 在构造函数中启动一个线程

如果在构造函数中启动一个线程,那么将可能带来子类化问题,同时还会导致this引用从构造函数中逸出

### 条件等待中的错误

当在一个条件队列上等待时,object.wait和condition.await方法应该在检查了状态谓词之后,在某个循环之中调用,同时需要持有正确的锁,如果在调用object.wait和 condition.await方法时没有持有锁,或者不在某个循环中,或者没有检查某些状态谓词,那么通常都是一个错误

### 在休眠或等待的同时持有一个锁

如果在调用thread.sleep时持有一个锁,那么将导致其他线程在很长一段时间内无法执行,因此可能导致严重的活跃性问题.如果在调用object.wait或condition.await时持有两个锁,那么也可能导致同样的问题

# 显式锁

必须在finally块中释放锁

在synchronized内置锁中，出现死锁时，恢复程序的唯一方式是重启程序，而防止死锁的唯一方式是在构造程序时避免出现不一致的锁顺序



## ReentrantLock

特性：可定时，可轮询，可中断的锁获取操作，公平队列, 非块结构

reentrantLock.lockInterruptibly(); 可中断的锁获取操作

## 轮询锁与定时锁

提供另一种选择来避免死锁的发生

如果不能获取锁，会释放已经获得的锁，然后重新尝试获取所有锁

定时锁能根据剩余时间来提供一个时限，如果操作不能在指定时限内完成，则程序提前结束

## 公平性

竞争激烈时,非公平锁的性能高于公平锁的性能的一个原因是:在恢复一个被挂起的线程与该线程真正开始运行之前存在着严重的延迟

当持有锁的时间较长,或者请求锁的平均时间间隔较长,应该使用公平锁

## Synchronized VS ReentrantLock

仅当内置锁无法满足需求的情况下，才使用ReentrantLock

使用ReentrantLock场景：可定时的，可轮询的与可中断的锁获取操作，公平队列，以及非块结构的锁

## 读写锁

读写锁能提高读取多处理器系统上的数据结构的速度，而在其他情况下，读写锁的性能比较差

当锁由读线程持有,而有写线程请求锁时,其他读线程只能等到写线程使用完并释放了写入锁后才能持有读取锁

写线程拥有更高的优先级，写线程可以降级为读线程，而读线程不能升级为写线程，否则容易导致死锁:如果两个读线程试图同时升级为写入锁,那么二者都不会释放读取锁,

# 构建自定义的同步工具



## 状态依赖性问题

某些操作是基于状态的，如不能从空队列删除元素，要获取尚未结束的任务的计算结果，必须等到队列进入“非空”状态或者任务进入已完成状态

依赖状态的操作可以一直阻塞直到可以继续执行，可以通过轮询(循环)与休眠来实现简单的阻塞，其思路是使用循环方式，重试直到成功，这并不是一种好的实现

使用基于 LinkedBlockingQueue latch Semaphore FutureTask的条件队列

## 条件谓词

public synchronized void put(V v) throws InterruptedException {

while (isFull())

wait();

doPut(v);

notifyAll();

}

在循环体中调用wait(),若在唤醒时wait前提的条件已经更改，则需要重新判断，所以需要循环

## 通知

每当在等待一个条件时，一定要确保在条件谓词变成真时通过某种方式发出通知

如*条件谓词* 中，每当put一个元素后，都执行notifyAll(放后，尽快退出, notify和notifyAll方法都不释放锁,只是通知wait状态的线程准备获取锁)，通知唤醒在take上等待的线程,

注意是唤醒哪个锁上的对象

* notifyAll

使用notifyAll而不是notify，可以避免多种类型的条件在等待一个锁时，唤醒的不是自己想要唤醒的锁上的线程，避免了信号丢失问题

* notify

使用notify应同时满足以下两个条件

1. 所有等待线程的类型都相同

只有一个条件谓词与条件队列相关，并且每个线程在从wait返回后将执行相同的操作

1. 单进单出

在条件变量上的每次通知，最多只能唤醒一个线程来执行

在队列中，非空才能take，非满才能put，则只满足2，而在条件队列上等待的线程有两个关联条件谓词

## 子类安全问题

对于状态依赖的类，要么将其等待和通知等协议完全向子类公开，并且写入正式文档，要么完全阻止子类参与到等待和通知等过程中

## 入口协议与出口协议

用于描述wait和notify方法的正确使用

对于每个依赖状态依赖的操作，与每个修改其他操作依赖状态的操作，都应该定义一个入口协议和出口协议

入口协议即操作的条件谓词，出口协议则包括检查被该操作修改的所有状态变量，并确认他们是否使某个其他的条件谓词为真，若是，则通知相关的条件队列

## 显示Condition

* 创建

Condition可以由相关联的lock来创建，每个lock可以有任意多个Condition对象

private final Condition notEmpty = lock.newCondition();

* 公平性

Condition对象继承了lock对象的公平性，对于公平的锁，线程会按照FIFO顺序从Condition await中释放

* 方法改写

在Condition对象中，wait，notify和notifyAll方法对应的分别是await，signal和signalAll——务必确保使用正确的版本

* 使用场景

使用公平的队列操作或者在每个锁上对应多个

等待线程集

## AQS

AbstractQueuedSynchronizer(AQS)是一个用于构建锁和同步器的框架，许多同步器可以通过AQS很容易并高效地构造出来，如ReentrantLock，Semaphore，FutureTask，

CountDownLatch

# 原子变量与非阻塞同步机制



## 并发CAS

CAS包含3个操作数——需要读写的内存位置V，进行比较的值A和拟写入的新值B

当且仅当V的值等于A时，CAS才会通过原子方式用新值B来更新V的值，否则不执行任何操作

CAS是一项乐观的技术，它希望能成功地执行更新操作，并且如果有另一个线程在最近一次检查后更新了该变量，那么CAS能检测到这个错误，在本次更新时不执行更新操作

由于CAS能检测到来自其他线程的干扰，因此即使不用锁也能实现原子的读-改-写操作序列

经验法则：在大多数处理器上，在无竞争的锁获取和释放的“快捷代码路径”上的开销，大约是CAS开销的两倍

## 原子变量类

12个原子变量类，分成4组

1. 标量类

支持CAS，AtomicInteger，AtomicBoolean，AtomicLong，AtomicReference

1. 数组类

只支持Integer，Long，Reference的原子数组类中的元素，可以实现原子更新

1. 更新器类
2. 符合变量类

## 非阻塞算法

非阻塞算法:一个线程的失败或挂起不会导致其他线程也失败或挂起

无锁算法:在算法的每个步骤中都存在某个线程能够执行下去

构建非阻塞算法的技巧在于:将执行原子修改的范围缩小到单个变量上

## ABA问题

在算法执行中,值改变后又改变回原来的值,在CAS判断时就有误判

解决方案:不是更新某个引用的值,而是更新两个值,包括一个引用和一个版本号

# Java内存模型



## Happens-before

如果两个操作之间缺乏Happens-before关系,那么JVM可以任意地重排序

Volatile变量规则:对Volatile变量的写入操作必须在对该变量的读操作之前执行

线程启动规则:在线程上对Thread.start()的调用必须在该线程中执行任何操作之前执行

传递性:A在B前完成,B在C前完成,则A在C前完成

## 不安全发布

当缺少Happens-before关系时,就可能出现重排序问题,所以才会出现在没有充分同步的情况下发布一个对象会导致另一个线程看到一个只被部分构造的对象

# 问答

## Java同步集合与并发集合的区别

同步集合类，Hashtable 和 Vector 还有同步集合包装类，Collections.synchronizedMap()和Collections.synchronizedList() 提供了一个基本的有条件的线程安全的Map和List的实现。

并发集合像ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性

## 如何避免死锁

死锁的发生必须满足以下四个条件：

1. 互斥条件：一个资源每次只能被一个进程使用。
2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。
4. 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁

## 如何在Java中创建Immutable对象

通过构造方法初始化所有成员

对变量不要提供setter方法

将所有的成员声明为私有的,这样就不允许直接访问这些成员

在getter方法中，返回克隆对象

域为final

## volatile 变量和 atomic 变量有什么不同

Volatile变量可以确保先行关系，即写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用volatile修饰count变量那么 count++ 操作就不是原子性的

AtomicInteger类提供的atomic方法可以让这种操作具有原子性