# 简介

1.3 并发带来的风险

1. 安全性问题——这个类能不能正确地工作

2. 活跃性问题——“某件正确的事最终会发生”->死锁、饥饿

3. 性能问题——多线程导致频繁的上下文切换、同步机制抑制编译器优化

1.4 线程无处不在

a) JVM启动时，为JVM的内部任务（GC、终结操作）创建后台线程，并创建一个主线程来执行main方法

b) 一些组件框架。比如Servlet用于部署网页应用程序以及分发来自HTTP客户端的请求。多个客户端同时请求同一个Servlet的服务->Servlet被多个线程同时调用。

第一部分 基础知识

# 第2章 线程安全性

2.1 什么是线程安全性

1. 核心概念是“正确性”——某个类的行为与其规范一致

2. 无状态对象一定是线程安全的。

3. 要编写线程安全的代码，其核心在于要对状态访问操作进行管理，特别是对共享（shared）和可变（mutable）状态的访问。

4. 要使对象是线程安全的，需要采用**同步机制**来协同对对象可变状态的访问。

5. Java中的主要同步机制是关键字synchronised，它提供了一种**独占的**加锁方式。但“加锁”这个术语还包括volatile类型的变量、显式锁（ReentrantLock）、以及原子变量。

2.2 原子性

1. 注意：i++操作不是原子的。三个独立操作（读-加一-写入）。、

2. 尽可能使用现有的线程安全对象（例如AcomicLong）来管理类的状态。

2.3 加锁机制

**内置锁**

1. Java提供了一种内置的锁机制来支持原子性：同步代码块（Synchronised Block）。每个Java对象都可以当做用来同步的锁来使用。
2. Java的内置锁相当于一种互斥体
3. 内置锁是可重入的。



**重入**

1. “重入”==获取锁的操作的粒度是“线程”，而不是“调用”
2. 实现方式：
   1. 为每个锁关联一个计数器和一个所有者线程。
   2. Count=0时，认为锁没有被任何线程持有。此时线程请求它时，JVM会记下锁的持有者，将count置1.
   3. 同一个线程再次获取这个锁，count++；当线程退出同步代码块，count--
   4. 直到count==0，这个锁将被释放。

2.4 用锁来保护状态

- 对于可能被多个线程访问的可变状态变量，在访问它时都需要持有同一个锁——称状态变量是由这个锁保护的。

2.5 活跃性与性能

将不应现共享状态而且执行时间较长的操作从同步代码从syn-block中分离。

Allinall，在保证安全性条件下，缩小锁的粒度。

# 第3章 对象的共享

3.1 可见性

1. 定义：确保当一个线程修改了对象状态后，其他线程能看到发生的状态变化。

a) “不可见”往往是因为重排序

——在没有同步的情况下，编译器、处理器、以及运行时都会发生指令重排序

b) 在没有同步的情况下读取数据，类似于数据库READ\_UNCOMMITTED隔离级别，在这种级别上将牺牲准确性以获取性能的提升。

2. 非原子的64位操作

a) **最低安全性**：当线程在没有同步的情况下读变量时，可能得到失效值，但至少是之前某个线程设置的值

1. 最低安全性适用于大多数变量，但是64位LONG和Double是非volatile的
2. Java内存模型要求，变量的读取操作和写入操作都必须是原子操作。
3. 对于非volatile类型的long和double，JVM允许将64位的读和写分解成两个32位的操作。

3. 加锁与可见性

a) 加锁不仅互斥（原子性）、还能保证可见性

b) that’s why在访问某个共享且可变的变量时要求所有线程在**同一个锁**上同步

4. Volatile变量

a) 出现背景or用途：**Java提供了一种较弱的同步机制**，即Volatile变量。

b) 描述：当把变量声明为Volatile类型后，编译器与运行时就会注意到这个变量是共享的，因此不会将该变量上的操作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什么（直接写入内存），因此读取volatile类型的变量时总会返回最新写入的值。

c) why“稍弱的同步机制”？

-“同步”：类似synchronisedInteger，把读写操作替换成get方法和set方法

-“稍弱”：访问时不执行加锁操作，因此也就不会使线程阻塞。

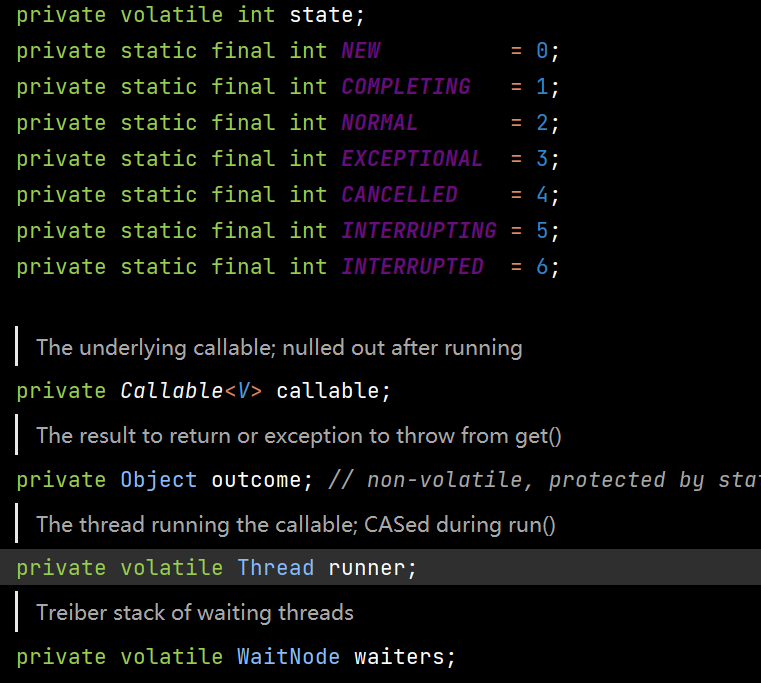
d) **典型用法：检查某个状态标记以判断是否退出这个循环。**

通常用做某个操作完成、发生中断或者状态的标识，比如asleep

【注意】：尽量不用来标识其他的状态信息。因为volatile的语义不足以确保递增操作（i++）的原子性，除非你能确保只有一个线程对变量执行写操作。

当且仅当满足一下条件时，才能使用：

* 对变量的写操作不依赖变量的当前操作（类似：自增），or确保只有单线程更新变量的值
* 该变量不会与其他变量一起纳入不变性条件中。（同一个对象的不同变量之间的约束条件）  *\*不懂*
* 在访问变量时不需要加锁



（举例：FutureTask中用于标记state，state就=xx，而不是state=state+1；）

3.2 发布与逸出

1. “发布”一个对象：使该对象能够在当前作用域之外的代码中使用。

“逸出”：发布了不应该被发布的对象

2. 安全的对象构造过程

不要使this引用逸出 -> 不要在构造函数中启动线程

若要在构造函数中注册一个事件监听器or启动线程 -> 使用私有构造+公共的工厂方法

3. allinall，这一节是为了引出线程封闭的。

3.3 线程封闭

1. 什么是：

要实现线程安全->不使用同步机制-> 不共享数据-> 单线程

-> **仅在单线程内访问数据/对象**

**(哪怕这些对象本身不是线程安全的）**

-> 使用同步机制 -> 通过synchronised加锁以及其他同步机制

-> 使用不可变对象满足同步需求

2. 常见应用：

Swing的可视化组件和数据模型对象都不是线程安全的，Swing把它们封装到Swing的事件分发线程中来实现线程安全性。

JDBC的Connection对象。JDBC规范要求Connection对象是线程安全的。C-S服务中，线程从连接池取得一个Connection对象，该对象用来处理请求。在处理请求时隐含地将Connection对象封闭在线程中。

3. **维持线程封闭性的机制**

a) 局部变量/栈封闭

局部变量在位于执行线程的栈里，而栈不被线程共享

b) **ThreadLocal类**

1. 是什么：

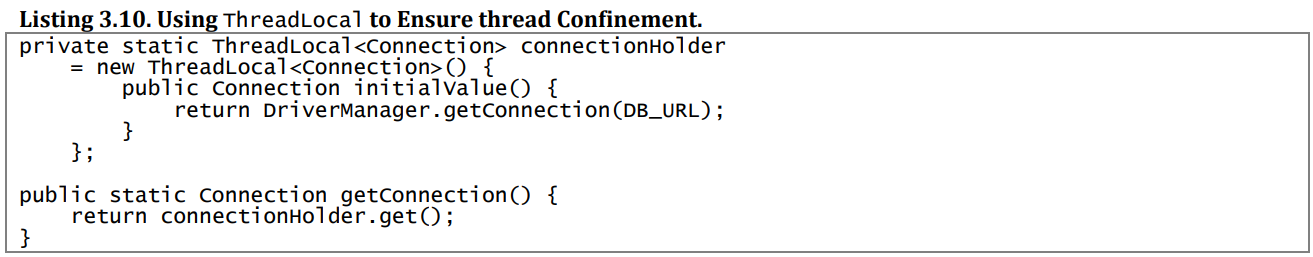
这个类能使线程中的某个值与保存值的对象关联起来。

ThreadLocal提供了get和set等访问接口或方法，这些方法为每个使用该变量的线程都存有一份独立的副本，因此get总是返回由当前执行线程在调用set时设置的最新值。

2. 使用场景：

当某个频繁执行的操作需要一个临时对象，例如一个缓冲区，而同时又想避免在每次执行时都重新分配该临时对象，就可以使用这项技术。

当某个线程初次调用ThreadLocal.get方法是，就会调用initialValue来获取初始值。

****

（这里的使用很类似于java8的supplier类、单例的延迟初始化-占位类模式）

假如你需要将一个单线程应用程序移植到多线程环境中，通过将全局变量转换为ThreadLocal对象，可以维持线程安全性。

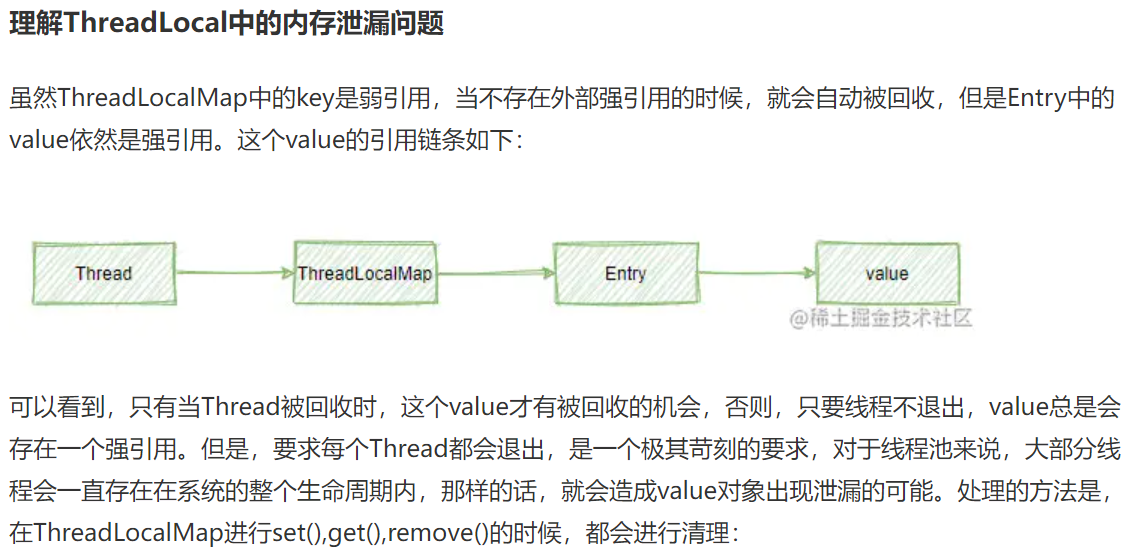
* ThreadLocal 可以用来管理 Session，因为每个人的信息都是不一样的，所以就很适合用 ThreadLocal 来管理；
* 数据库连接，为每一个线程分配一个独立的资源。

3. 实现：

概念上，可以将ThreadLocal<T>视为包含了Map<Thread,T>对象

实际上，这些特定于线程的值保存在Thread对象中

4. 注意



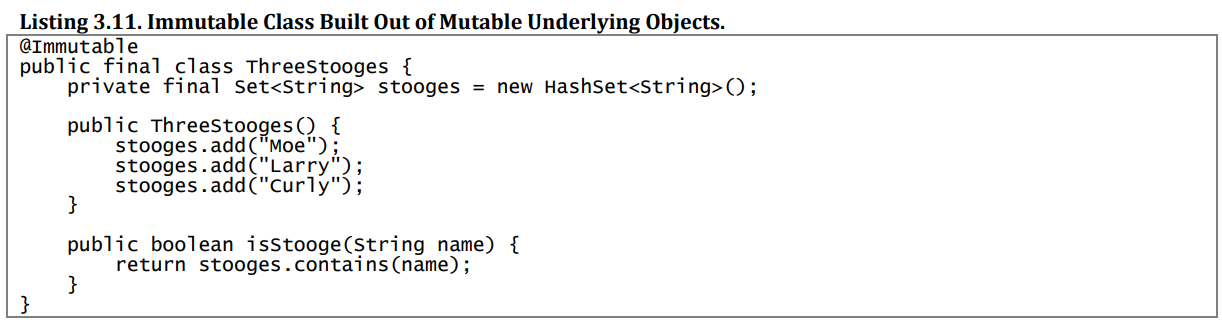
因此，一个良好的习惯依然是：**当你不需要这个ThreadLocal变量时，主动调用remove()，这样对整个系统是有好处的**。

3.4 不变性

a) **不可变对象一定是线程安全的**

对象中所有的域都是final类型的 != 对象是不可变的

（因为在final域中可以保存对可变对象的引用）



（在可变对象基础上构建的不可变类）

b） JMM中，final域还有着特殊的语义。**Final域能确保初始化过程的安全性**，从而可以不受限制地访问不可变对象，并在共享这些对象时无需同步。

3.5 安全发布

# 第4章 对象的组合

4.1 设计线程安全的类

1. 设计三要素：

* 找出构成对象状态的所有变量
* 找出约束**状态变量**的不变性条件
* 建立对象状态的并发访问管理策略

1. 同步策略规定了如何将不可变性、线程封闭与加锁机制等结合起来以维护线程的安全性

4.2 实例封闭

1. 将数据封锁在对象内部，可以将数据的访问限制在对象的方法上，从而更容易**确保**线程在访问数据是总能持有正确的锁。
2. 应用：Java类库有很多线程封闭的实例，其中有些类的唯一用途就是将非线程安全的类转化为线程安全的类。如ArrayList和HashMap是非线程安全的，类库提供包装器工厂方式（例如Collections.sychronisedList），使得它们可以在多线程环境中安全的使用。

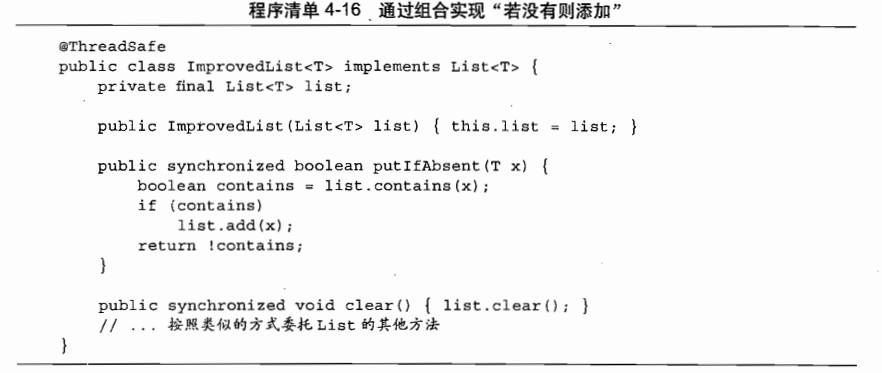
\*这就是“装饰器模式”——

1. 将容器类封装在一个同步的包装类对象中，而包装器能将接口中的每个方法都实现为同步方法，并将调用请求转发到底层的容器对象中。
2. 只要包装类对象拥有对底层容器对象的唯一引用，那么它就是线程安全的。
3. 装饰器模式在线程安全里的应用，也称“监视器模式”。

4.3 线程安全性的委托

4.4 在现有的线程安全类中添加功能

1. 最安全的方法是修改原始的类，但通常无法访问或修改原始类。
2. 扩展-> 继承 重写
3. 当为现有的类添加一个原子操作时，有一个更好的方法：组合



4.5 将同步策略文档化（略）

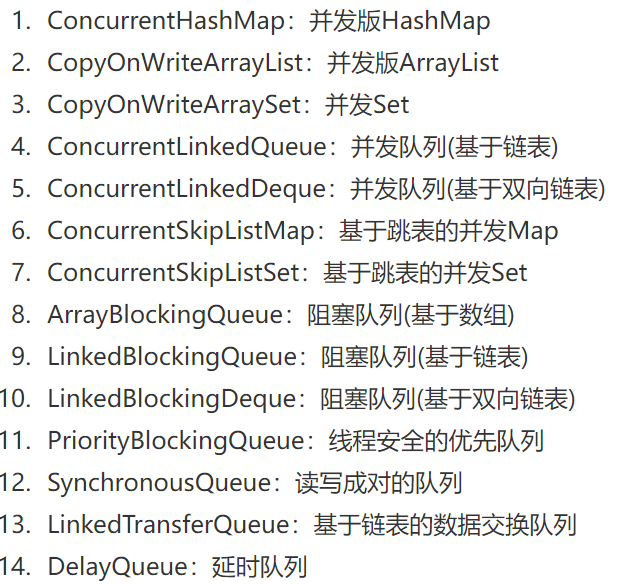
# 第5章 基础构建模块

5.1 同步容器类

1. 同步容器类包括Vector和HashTable，二者是早期JDK的一部分。还有一些别的。
2. 同步容器类都是线程安全的
3. 问题：
   1. 这些类实现线程安全的方式是：将它们的状态封装起来，并对每个公有方法都进行同步，使得每次只有一个线程能访问容器的状态。
   2. 对所有容器状态的方位都串行化。严重降低了并发性，当多个线程竞争容器的锁时，吞吐量将严重降低。

5.2 并发容器

1. 并发容器是针对多个线程并发访问设计的。
2. 有：ConcurrentHashMap以代替同步散列的Map，CopyOnWriteArrayList以代替遍历操作非常多的同步的List，ConcurrentSkipListMap和ConcurrentSkipListSet分别作为同步的SortedMap和SortedSet的并发替代品。BlockingQueue。。。



1. ConcurrentHashMap
   1. 原理：

粒度更细的加锁机制——**分段锁**：任意数量的读取线程可以并发地访问Map，读和写可以并发地访问map，一定数量的（最多16）写入线程可以并发地修改Map。

* 1. 结果：

在并发访问环境下将实现更高的吞吐量，而在单线程中只损失非常小的性能。请多多使用吧！

* 1. 【注意】ConcurrentHashMap**没有**实现对Map加锁以提供**独占访问**。但是对“若没有则添加”、“若相等则移除”等都已实现为原子操作并且在ConcurrentHashMap接口声明了方法。

1. CopyOnWriteArrayList
   1. 原理：

“写入时复制”的安全性在于，只要正确地发布一个事实不可变的对象，那么在访问该对象时就不再需要进一步的同步。。在每次修改时，都会创建并重新发布一个容器副本，从而实现可变性。

* 1. 结果：

每次修改都会复制底层数组，开销大。仅当迭代操作远远多于修改时，才使用。

举例应用场景：事件通知系统。

5.3 阻塞队列&生产者-消费者模式

1. 阻塞队列提供了可阻塞的put和take方法（which简直born for生产者消费者模式），以及支持定时的offer和poll方法。
   1. 类库多种BlockingQueue实现：Array~、Linked~、Priority~、SynchronousQueue（比较特殊：不会为队列中元素维护存储空间，而是维护一组线程。好处是可以直接交付工作，降低了数据从生产者移动到消费者的延迟。但需要有足够多的消费者，并且总有消费者thirsty for任务todo）
2. 生产者-消费者模式简单讲一讲：
   1. 两/多个人关于“洗盘子”劳动分工。。。每个工人只需与盘子架打交道。人们不关心有多少生产者or消费者，以及谁生产了某个指定的盘子架。
   2. “生产者”与“消费者”的角色是相对的
3. Both
   1. BQ简化了消费者程序的编码，因为take会一直阻塞直到有可用资源
   2. BQ简化了生产者程序的编码。如果生产者效率快，工作项在队列中积累会耗尽内存。Put使当队列满时阻塞，供消费者追赶。
      1. Offer()方法，如果数据项不能被添加到队列，将返回失败状态。这样，你可以选用多种策略处理负荷过载的情况，比如将多余生产的工作项序列化写入磁盘、抑制生产者线程等。

5.4 阻塞方法与中断方法

1. 背景之**阻塞**：
   1. 从运行态到阻塞态往往是由于：等待I/O操作结束、等待获得一个锁、等待从Thread.sleep()方法中醒来、等待另一个线程的计算结果。
   2. 阻塞与执行时间很长的普通操作的差别在于：被阻塞的线程必须等待某个不受它控制的事件发生后才能继续执行。
   3. 运行态切换到阻塞态/就绪态的区别：就绪态由于时间片or抢占式。
2. Thread提供了interrupt方法，用于中断线程或者查询线程是否已经被中断。
3. 中断处理的两个选择：
   1. **传递InterruptedException**——避开这个异常，把InterruptedException传递给方法的调用者。具体：根本不捕获该异常，或者捕获，然后再次抛出
   2. **恢复中断**——当必须捕获时：捕获InterruptedException，并调用当前线程上的interrupt方法恢复中断状态，这样在调用栈中更高层的代码看到引发了一个中断。

5.5 同步工具类

同步工具类可以是任意一个对象，只要它根据自身的状态来协调线程的控制流。

包含：阻塞队列、信号量Semaphore、栅栏Barrier、闭锁Latch

同步工具类都包含一些特殊的结构化属性：它们封装了一些状态which将决定执行同步工具类的线程是继续执行还是等待。

【注意】：以下同步工具类在类中，都是作为类的全局private final成员变量来使用的

1. 闭锁Latch
   1. 闭锁可以延迟线程的进度直到其到达终止状态。闭锁的作用相当于一扇门：在闭锁到达结束状态之前，折扇们一直关闭，任何线程都不能通过；当到达结束状态时，这扇门打开允许任何线程通过，并且永远保持打开。
   2. 应用场景：
      1. 确保某个计算在需要的所有的资源都被初始化后才继续执行
      2. 确保某个服务在其依赖的所有其他服务都已经启动之后才启动。
      3. 等待知道某个操作的参与者都就绪再继续执行。（如，多玩家游戏的玩家。）
   3. CountDownLatch是一种灵活的闭锁实现
      1. 一个计数器+countDown()方法递减计数器+await()方法顶戴计数器达到零，意味着所有需要的等待的时间都已经发生。如果计数器非零，await会一直阻塞直到计数器为零、或等待中的线程中断、或等待超时。
2. FutureTask
   1. FutureTask也可以用作闭锁。
   2. 主要的功能是：**获取线程的异步执行结果**

再具体，来提前加载高开销的计算

* 1. 实现：

概念上，实现了*Future（只是一个接口定义）*的语义。实现上，对 Callable 和 Runnable 进行了一层封装。

(参考[Java 线程池中 Future & FutureTask 使用总结](https://dongzl.github.io/2020/05/20/26-JDK-ThreadPool-Future-FutureTask/index.html))

1. 信号量
   1. 计数信号量用来控制同时访问某个特定资源的操作数量。
   2. 简化：二值信号量(0-1)，可以用做互斥体mutex，并具备不可重入的语义.
   3. 应用场景：
      1. 实现某种线程池：数据库连接池
      2. 对容器施加边界：BoundedHashSet
2. 栅栏
   1. Barrier和Latch区别：
      1. 闭锁用于等待事件，栅栏用于等待其他线程
   2. 实现
      1. CyclicBarrier：在并行迭代算法中非常有用；在模拟程序中通常需要用到栅栏
      2. Exchanger：是一种两方栅栏

5.6 实例：构建高效且可伸缩的结果缓存

优化过程：

1. Synchronised整个compute方法 -问题:糟糕的并发性
2. 用concurrenthashmap代替hashmap -问题：如果某个线程策划了一个开销很大的计算，而其他线程不知道这个计算正在进行，那么可能会重复这个计算。
3. Concurrenthashmap<K,V>优化为Concurrenthashmap<K,Future<V>> -小小问题：两个线程仍有可能在同一时间内调用compute来计算相同的值（都没有在缓存中找到），因为compute中的if代码（“先检查再执行”）仍然是非原子的
4. 使用concurrentmap中的原子方法putIfAbsent

第二部分 结构化并发应用程序

# 第6章 任务执行

6.1 在线程中执行任务

当围绕“任务执行”来设计时，第一步就是找出清晰地任务边界。大多数服务器应用系统都提供了一种自然的任务边界选择方式：以独立的客户请求为边界。

6.1.1 串行地执行任务：

资源利用率不行，全耗在网络连接、文件IO、数据库上了，cpu闲死了

6.1.2 显式地为任务创建线程：

为每个请求创建一个新的线程来提供服务，以实现更高的响应性

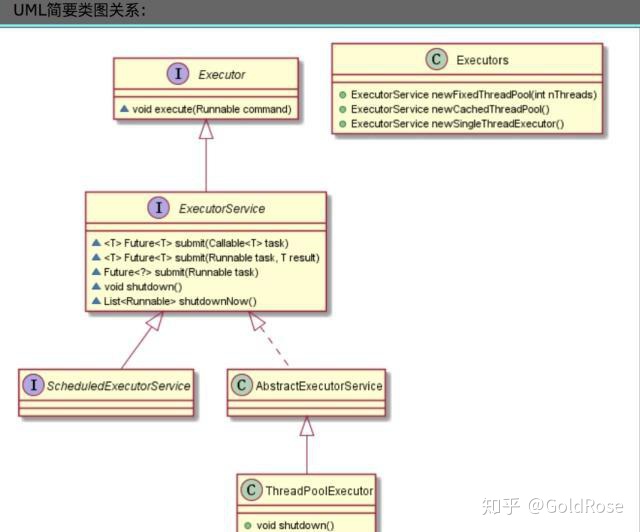
6.1.3 无限制创建线程的不足

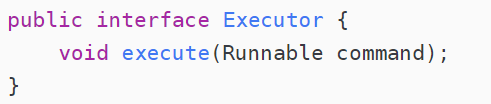
1. 线程的创建和销毁有代价
2. 活跃的线程会消耗系统资源，尤其是内存
3. 可创建线程数有限制，受JVM启动参数、Thread构造函数中请求栈大小限制。

6.2 **Executor框架**

1. **任务**是一组逻辑控制单元，而**线程**是使任务异步执行的机制

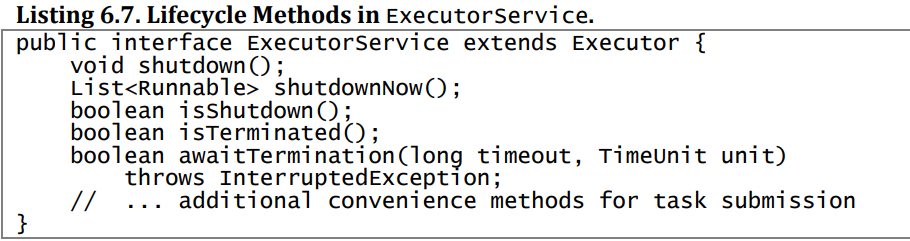
**Executor框架是由任务，任务的执行，异步计算的结果组成**

1. 优点之一，Executor简化了线程的管理工作By将请求处理任务的提交（这一part分给了executor，更准确的说，是ExecutorService）与任务的实际执行解耦开。优点之二，简化任务与线程的生命周期管理（见下，ExecutorService）
2. 线程池
   1. 概念
      1. 是什么？管理一组同构工作线程的资源池。
      2. 主要功能？线程池与工作队列（work queue）密切相关。工作队列保存了所有等待执行的任务。工作者线程（Worker Thread）的任务很简单：从工作队列中获取一个任务，执行任务，然后返回线程并等待下一个任务。
   2. 优点（相比“为每个任务分配一个线程”）：
      1. 通过重用现有的线程而非创建新线程，
         1. 分摊创建和销毁线程的巨大开销
         2. 请求到达，工作线程通常已经存在，提高响应性
      2. 通过调整线程池大小，
         1. 使处理器保持忙碌
         2. 防止多线程相互竞争资源而耗尽内存资源
   3. 创建一个线程池
      1. 类库提供了一个线程池以及一些有用的默认配置。可以通过调用**工具类Executors**中的*静态工厂方法*之一来创建一个线程池。
      2. 区分：Executor、ExecutorSeivice、Executors
         1. 都是Java Executor框架的一部分
         2. Executor是一个抽象层面的核心接口



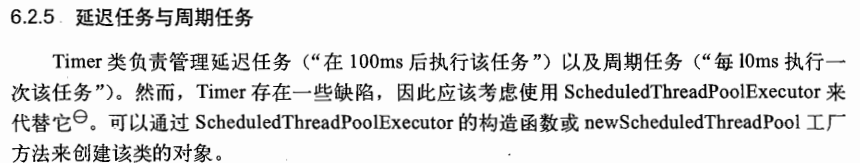
* + - 1. ExecutorService 接口 对 Executor 接口进行了扩展，提供了返回 Future 对象，终止，关闭线程池等方法。
      2. Executors是一个工具类，类似于 Collections。提供工厂方法来创建不同类型的线程池，比如 FixedThreadPool 或 CachedThreadPool。

1. Executor的生命周期
   1. 为什么重要？：Executor的实现通常会创建线程来执行任务。但JVM只有在所有（非守护）线程全部终止后才会退出。->需要正确地关闭Executor ->Executor以异步的方式执行任务，人和我时刻，之前提交的任务的状态不是立刻可见的-> 为了解决执行服务的生命周期问题，Executor扩展了ExecutorService接口，添加了一些用于生命后期管理的方法。



* 1. **ExecutorService**的生命周期有3种状态：运行、关闭、已终止
     1. ExecutorService在初始创建时处于运行状态
     2. Shutdown方法将执行平缓的关闭过程：不再接受新任务，同时等待已经提交的任务完成（包括已提交但还未开始执行的任务）
     3. shutdownNow方法将执行粗暴的关闭过程：它将尝试取消所有运行中的任务，而且不在启动队列中尚未开始执行的任务。

1. 延迟任务与周期任务

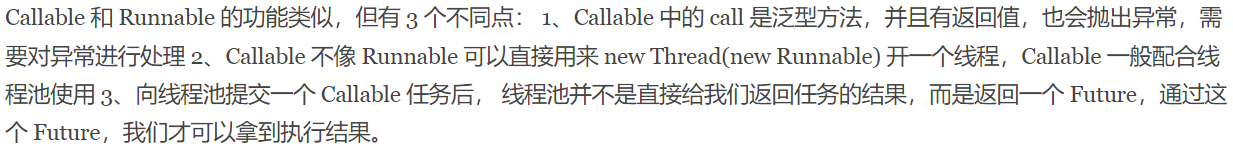


6.3 找出可利用的并行性

6.3.2 携带结果的Callable与Future

1. Executor框架使用Runnable作为其基本的任务表示形式。但是有局限：虽然run能写入到日志文件或者将结果放入某个共享的数据结构，但是它不能**返回一个值或抛出一个受检查的异常。——Callable**

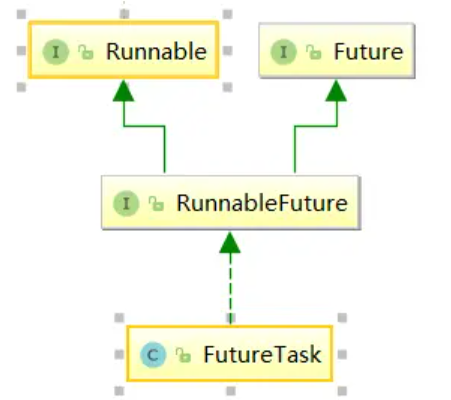
*\*受检异常：除了RuntimeException与其子类，以及错误（Error），其他的都是检查异常（绝对的大家族）。*

**

1. Runnable和Callable描述的都是抽象的计算任务。这些任务有明确的范围、起始点。Executor执行的**任务**有4个生命周期阶段：创建、提交、开始和完成。
2. Future表示一个任务的生命周期，并提供了相应的方法来判断是否已经完成和取消，以及获取任务的结果和取消任务等。
   1. Future规范中隐含的语义：任务的生命周期只能前进，不能后退。
   2. **Future.get()方法**：get方法的行为取决于任务的状态（未开始/正运行/已结束）。如果任务已经完成，get会立即返回或者抛出Exception；如果任务未完成，get将会阻塞直到任务完成
   3. 可以通过多种方法创建一个Future来描述任务：

\*future可看作表示任务的*句柄*

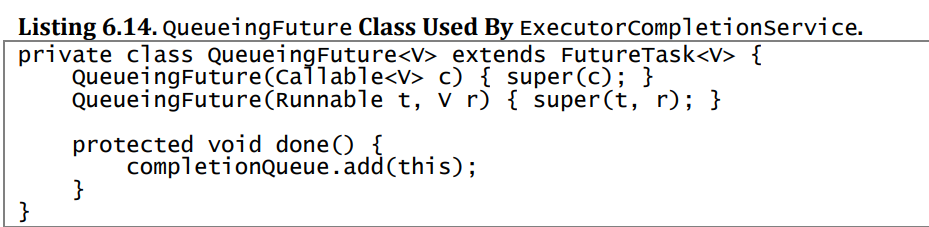
* + 1. ExecutorService中的所有submit方法都将会返回一个future方法都将返回一个Future，从而将一个Runnable或Callable提交给Executor，并得到一个Future*用来获得任务的执行结果或取消任务*。
    2. 还可以显式地为某个指定的Runnable或Callable实例化一个FutureTask。



6.3.4 在异构任务并行化中存在的局限

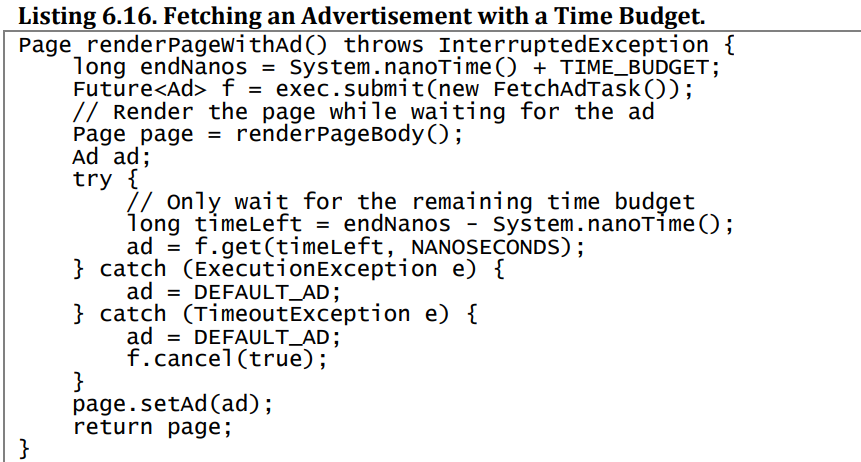
1. 只有当任务都是同类型的并且相互独立时，线程池的性能才能达到最佳
2. 所幸在基于网络的典型服务器应用程序中，它们的请求通常都是同类型并且相互独立的。

6.3.5 CompletionService: Executor与BlockingQueue



6.3.7 为任务设置时限

Future.get方法，catch TimeException异常，处理



6.3.8 invokeAll

支持限时的invokeAll，将多个任务提交到一个ExecutorService并获得结果。

invokeAll方法的参数为一组人物，并返回一组Future

# 第7章 取消与关闭

1. Java中没有一种安全的抢占式方式来停止线程，因此也就没有安全的抢占式方法来停止任务。只有一些协作式的机制，使请求取消的任务和代码都遵循一种协商好的协议。包含：
   1. 中断Interruption，能够使一个线程终止另一个线程的当前工作。
   2. 设置某个“已请求取消（Cancellation Requested）”标识，任务将定期地查看该标志。
      1. 一个可取消的任务必须拥有取消策略，即详细定义how\when\what
2. 中断理解
   1. **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。
      1. 当线程在非阻塞状态下中断，它的中断状态将被设置，然后根据接下来的被取消的操作来检查状态以判断发生了中断。中断“有黏性”——如果不触发InterruptedException，中断状态将一直保持，直到明确地清除中断状态。
      2. \*“然后由线程在下一个合适的时刻中断自己”——取消点
      3. Java语言开发规范中，并没有将中断与取消语义关联起来。然而，**中断**是**实现取消**的最合适方式。
3. 通过使用FutureTask和Executor框架，可以帮助我们构建可取消的任务和服务。
4. 关闭 基于线程的“生产者-消费者”服务 实例
   1. 设置“已请求关闭”标识
   2. 关闭ExecutorService
   3. “毒丸”对象-适用于生产者消费者数量已知
5. **JVM关闭**
   1. **JVM既可以正常关闭，也可以强行关闭**
      1. 正常关闭的触发方式：
         1. 当最后一个“正常线程（非守护线程）”结束时
         2. 当调用了Sysem.exit()时 \*0正常终止，非0异常终止
         3. 通过其他特定于平台的方式关闭，例如：发送SIGINT信号（Linux）或键入Ctrl-C （Windows）
      2. 强行关闭：
         1. 调用Runtime.halt()方法
         2. 在操作系统中“杀死”了JVM进程，例如发送SIGKILL信号
   2. 关闭钩子
      1. 在正常关闭中，JVM首先关闭钩子 --区分正常非正常关闭
      2. 具体做什么？Runtime.addShutdownHook(new Thread(){…})
   3. 守护线程
      1. 线程可分为两种：普通线程和守护线程。

JVM启动时创建的所有线程中，除了main主线程之外，其他都是守护线程（包括垃圾回收器及其他辅助工作的线程

* + 1. 二者唯一差异：线程退出时的操作：当一个线程退出时
       1. 检查正在运行的普通线程
       2. JVM停止时，所有仍然存在的守护线程都被抛弃——既不会执行finally代码块，也不会执行回卷栈。
    2. 我们应尽可能地使用守护线程——很少有操作能在不被清理的情况下被安全地抛弃。

# 第8章 线程池的使用

8.1 在任务和执行策略之间的隐形耦合

1. 线程饥饿死锁：
   1. 原因：因为线程池中，这个任务依赖于其他任务
2. 运行时长较长的任务
   1. 解决（缓解）：限定任务等待资源的时间 ->平台类库的大多数可阻塞方法中，都同时定义了限时版本和无限时版本，例如Thread.join, BlockingQueue.put, CountDownLatch.await, Selector.select

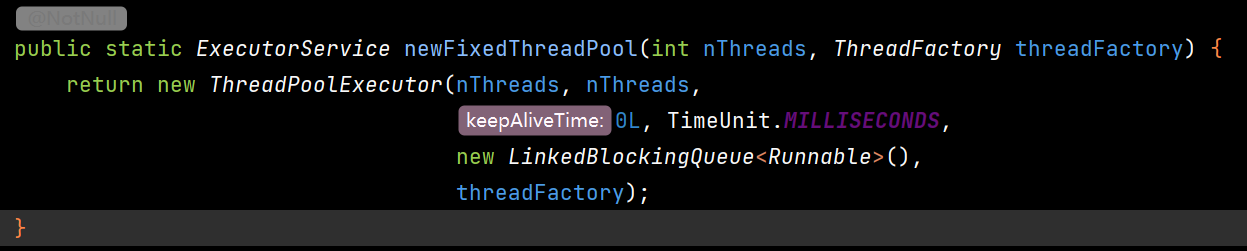
8.2 设置线程池的大小

线程池过大-> 大量线程竞争cpu和内存，反而导致更高的内存占用量，甚至耗尽资源

线程池过小-> 导致许多空闲的处理器无法执行工作，从而降低吞吐量。

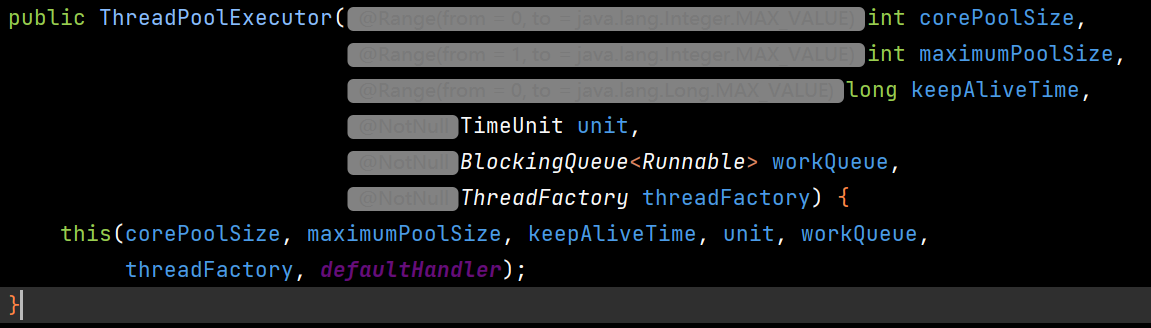
8.3 配置ThreadPoolExecutor

1. ThreadPoolExecutor为一些Executor提供了基本的实现，这些Executor是由Executors中的newFixedThreadPool、newScheduledThreadPool和newScheduledThreadPool等工厂方法返回的。



ThreadPoolExecutor是一个灵活的、稳定的线程池，允许进行各种定制。

* 1. 线程的创建与销毁由线程池负责：



（ThreadPoolExecutor通用构造函数）

* 1. 管理队列任务

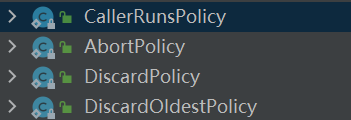
在有限的线程池中会限制可并发执行的任务数量。

基本的任务排列方法有3种：无界队列、有界队列和同步移交（synchronous Handoff）：

* + 1. newFixedThreadPool和newSingleThreadExecutor（\*也是threadpool，因为只有一条所以不叫它pool）在默认情况下将使用一个**无界的LinkedBlockingQueue**——如果任务快速到达，超过了线程池处理它们的速度，那么队列将无限制的增加
    2. 一种更稳妥的资源管理策略是使用**有界队列**，例如ArrayBlockingQueue、有界的LinkedBlockingQueue、PriorityBlockingQueue。——新的问题产生：当队列填满后，新的任务该怎么办？->见：饱和策略
    3. 对于非常大的或者无界的线程池，可以通过使用**SynchronousQueue**来避免任务排队，以及直接将任务从生产者移交给工作者线程。SynchronousQueue不是一个真正的队列，而是一种在**线程之间进行移交的机制**。要将一个元素放入SynchronousQueue中，必须有另一个线程正在等待接受这个元素。
  1. 饱和策略——针对有界队列

ThreadPoolExecutor的饱和策略可以通过setRejectedExecutionHandler来修改

JDK提供了几种不同的RejectedExecutionHandler实现，每种实现都包含有不同的饱和策略：



* + 1. 中止Abort是默认——“Always throws RejectedExecutionException”by文档
    2. Dicard两个顾名思义
    3. “调用者运行”实现了一种调节机制。该策略既不会抛弃任务，也不会抛出异常，将某些任务回退到调用者，从而降低新任务的流量。

8.4 扩展ThreadPoolExecutor

（可以，但最好不要）。

第三部分 活跃性、性能与测试

# 第10章 避免活跃性危险

“在安全性和活跃性之间通常存在着某种制衡。我们需要用加锁机制来确保线程安全。但如果过度地使用加锁，则可能出现锁顺序死锁。同样，我们使用线程池和信号量来限制对资源的使用，但这些被限制的行为可能会导致资源死锁。”

10.1 死锁

1. 是什么？
   1. 每个人都拥有其他人需要的资源，同时又等待其他人已经拥有的资源，并且每个人在获得所有需要的资源之前都不会放弃已经拥有的资源

\*人：线程；资源：锁

* 1. 如果一个类可能发生死锁，不是确定的，而是一种可能性。当死锁出现时，往往是在最糟糕的时候——在高负载的情况下

1. **锁顺序死锁：**
   1. 是什么/发生原因：两个线程试图以不同的顺序来获得相同的锁。
   2. 判断：一个类中有嵌套锁，或者两个协作对象之间。
   3. 解决：尽可能开放调用——尽量不要给整个方法加锁，而是使同步代码块仅被用于保护那些涉及共享状态的操作。
      1. Ps. 开放调用的重构中，可能会使得某个原子操作变为非原子操作，改变了代码的语义。有时是正确的，有时则会影响安全性，需要小心判断。
2. **资源死锁：**
   1. 是什么：正如多个线程相互持有彼此正在等待的锁而又不失方自己已持有的锁时会发生死锁，当它们在相同的资源集合上等待时，也会发生死锁。
   2. 场景：
      1. **两个资源池**（e.g.两个不同数据库的连接池：一个任务要连接两个数据库，并且每次请求连接时顺序不一样，那么线程A已持有数据库D1连接请求连接D2，而线程B则持有D2等待D1连接）（资源池越大，死锁几率越小，因为如果每个资源池都有N个连接，那么发生死锁需要N个循环等待的线程和大量不恰当的时序等待）

\*ps. 资源池通常采用信号量来实现，当资源池为空时阻塞

* + 1. 另一种基于资源的死锁形式是**线程饥饿死锁**。

10.2 死锁的避免和诊断

1. 如果一个程序每次只能获得一个锁，那么就不会出现锁顺序死锁。但是不现实。因此要求：尽量减少潜在的加锁交互数量。
2. 显式使用Lock类中的定时tryLock来代替内置锁机制。——可以检测死锁和从死锁中恢复。
3. 诊断：JVM线程转储（Thread Dump）

JVM中在某一个给定的时刻运行的所有线程的快照

10.3 其他活跃性危险

1. 饥饿
   * 1. 是什么？当线程由于无法访问到它所需要的资源而不能继续执行。
     2. **引发饥饿的最常见资源就是CPU时钟周期**：对线程优先级使用不当、或是持有锁时执行一些无法结束的结构（无限循环、无限制等待）
2. 活锁Livelock
   * 1. 是什么？活锁是另一种形式的活跃性问题，该问题尽管不会阻塞线程，但也不能继续执行。因为线程将不断重复执行之前的操作，并且总会失败。
     2. 发生机制？活锁通常发生在解决事务消息的应用程序中：如果不能成功处理某个消息，那么消息处理机制将回滚整个事务，并把它重新放在队列的开头。
     3. 解决？需要在重试机制中引入随机性。举例：两台机器在同一个载波上发送数据包，不要同时等待1s后重试

# 第11章 性能与可伸缩性

Amdah1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。这个值取决于程序中可并行组件与串行组件所占的比重。

**线程引入的开销：上下文切换+内存同步+阻塞**

\*内存同步：保证可见性用到指令：内存栅栏——刷新缓存，使缓存无效、刷新硬件的写缓冲 -> 大多数操作不能重排序，抑制编译器优化操作

\*阻塞：当在锁上发生竞争时，竞争失败的线程肯定会阻塞。 JVM实现阻塞时，可以采用自旋等待（循环不断尝试获得锁，直到成功）。或者通过操作系统挂起被阻塞的线程。

**减少锁的竞争：缩小锁的范围+减小锁的粒度（锁分解+锁分段）+少用独占锁**

\*缩小锁的范围：still，尽可能开放调用，减少锁的持有时间

\*锁分解：锁在对象上（1个） -> 锁在相互独立的状态变量上（多个）

适用于：竞争**适中**的锁（实际上是把这些锁转变为非竞争的锁）

\*锁分段：锁分解 -> 对一组独立对象上的锁进行分解

\*ConcurrentHashMap：每个锁保护所有散列桶的1/16，其中第N个散列桶由（Nmod16）个锁来保护

劣势：有些情况下需要加锁整个容器：when需要重新计算key的hash以分不到更大的桶集合时

\*替代独占锁：并发容器、读写锁、不可变对象、原子变量

对性能的思考：

1. 什么是“资源”？

CPU时钟周期、内存、网络带宽、I/O带宽、数据库请求、磁盘空间。。。

1. CPU没有得到充分利用的原因一般是？

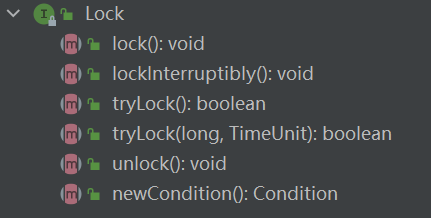
负载不充足、I/O密集、外部限制（数据库、web服务）、锁竞争

第四部分 高级主题

# 第13章 显式锁

与内置锁相比，ReentrantLock提供了一些扩展功能，在处理锁的不可用性方面有着更好地控制。但ReentrantLock不能完全替代synchronised，只有在synchronised无法满足需求时，才应该使用它。

1. Lock接口提供了一种无条件的、可轮询的、定时的以及可中断的锁获取操作，所有加锁和解锁的方式都是显式的。在Lock的实现中必须提供与内部锁相同的内存可见性语义。



1. ReentrantLock实现了Lock接口，并提供了与synchronised相同的互斥性和内存可见性。【注意】必须在finally块中释放锁

**轮询锁与定时锁——tryLock（）**

在内置锁中，死锁很严重，恢复程序的唯一方法是重新启动程序，而防止死锁的唯一方法是在构造程序是避免出现不一致的锁顺序。

可定时的与可轮询的锁提供了另一种选择：避免死锁的发生

**公平性**

在一般情况下（竞争激烈的情况下），非公平性锁的性能高于公平性锁的原因是：在恢复一个被挂起的线程与该线程真正开始运行之间存在着严重的延迟。ABC线程排队，A完成后等待A释放锁与B唤醒之间，如果C请求这个锁，非公平的话就直接给C了。

**读写锁**

ReadWriteLock中暴露了两个Lock对象，分别用于读写操作。

尽管这两个锁看上去是彼此独立的，但读锁和写锁只是读-写锁对象的不同视图。（都是ReentrantReadWriteLock类的成员）。

1. 这两种锁都是可重入的
2. 默认是非公平的。
   1. 在公平的锁中，a.等待时间最长的线程将优先获得锁。b.如果这个锁由读线程持有，而另一个线程请求写入锁，那么其他读线程都不能获得读锁，直到写线程使用完并且释放了写锁。
   2. 在非公平的锁中，a.线程获得访问许可的顺序是不确定的。b.写线程降级为读线程是可以的，但从读线程升级为写线程是不可以的（将导致死锁）
3. 和ReentrantLock类似，ReentrantReadWriteLock中的写入锁只能有唯一的所有者，并且只能由获得该锁的线程释放。

# 第14章 构建自定义的同步工具

类库中包括许多存在**状态依赖性**的类，例如FutureTask、Semaphore、BlockingQueue等。**在这些类中的一些操作有着基于状态的前提条件**，例如，不能够从一个空的队列中删除元素，或者获取一个尚未结束的任务的计算结果。在这些操作可以执行之前，必须等到队列进入“非空”状态，或者任务进入“已完成”状态。

还可以使用Java语言和类库提供的底层机制来构造自己的同步机制，包括内置的条件队列、显式的Condition对象以及AbstractQueuedSynchroniser框架。

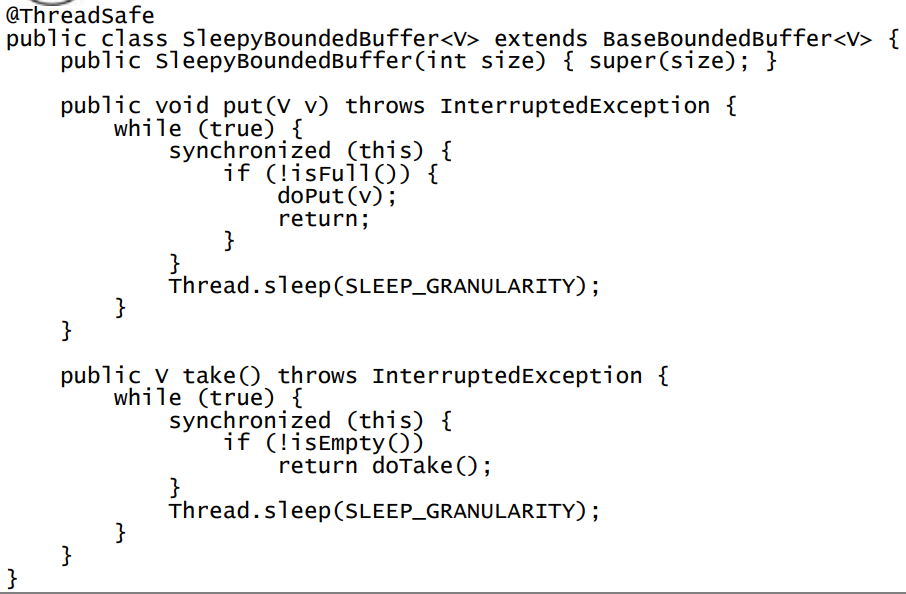
解决状态依赖性问题：轮询和休眠 -> 条件队列

在轮询和休眠之前的粗糙版：

1. 实现原理：“先检查再运行”，take时isempty/put时isfull就抛出异常
2. 问题：“缓存已满”并不是有界缓存的异常条件。就像“红灯”不代表信号灯出了异常。
3. 解决方案：调用者catch异常选择休眠或直接重新调用take -> 客户代码必须在二者之间进行选择：要么容忍自旋导致的CPU时钟周期浪费，要么容忍由于休眠而导致的低响应性。

**轮询和休眠：**

1. 实现原理：将重试机制封装起来，不必调用者操作。By：不满足条件就释放锁并睡一会儿，醒了再来问问满足条件吗？一直循环。
   1. 【备注】阻塞-> 非阻塞：从单线程的视角出发：把自己挂起vs时而回来问一下
2. 改进？：如果存在某种挂起线程的方法，并且这种方法能够去报当某个条件成真时线程立即醒来，那么将极大地简化实现工作，这就是条件队列实现的功能。
   1. 【备注】同步&异步：一种线程通信机制，多线程的全局视角：即来即响应vs回调



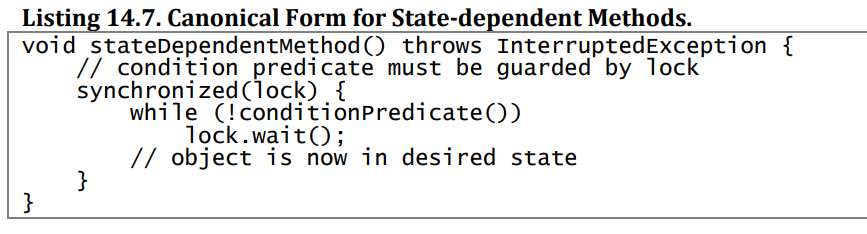
**条件队列**

1. “条件队列”名字来源于：它是的一组线程（等待线程集合）能够通过某种方式来等待特定的条件变成真。条件队列中的元素是一个个正在等待相关条件的队列。
2. 正如每个Java对象都可以作为一个锁，每个对象同样可以作为一个条件队列，并且Object中的wait、notify和notifyAll方法就构成了内部条件队列的API。对象的内置锁与其内部条件队列是相互关联的，要调用对象X中条件队列的任何一个方法，必须持有对象X上的锁。
   1. *Object.wait会自动释放锁，并请求操作系统挂起当前线程，从而使其他线程能够获得这个锁并修改对象的状态。当被挂起的线程醒来时，它将在返回之前重新获取锁。*从直观上看，调用wait意味着“我要去休息了，当发生特定事件时唤醒我”
3. **与使用“休眠”的有界队列相比**，条件队列并没有改变原来的语义。它知识在多个方面进行了优化：CPU效率、上下文切换开销和响应性。

**使用条件队列**

条件等待三要素：加锁+wait方法+一个条件谓词

a.条件谓词中包含着多个状态变量，由一个锁保护，因此在测试条件谓词之前必须持有这个锁。b.锁对象和条件队列对象（即调用wait和notify方法所在的对象）必须是同一个对象。



（状态依赖方法的标准形式）

\***等待**：wait方法的返回并不一定意味着线程正在等待的条件谓词已经变成真了。（从线程被唤醒到wait重新获取锁，有时差，maybe别的线程先wait获得了锁，并修改了对象的状态）-> 因此每当线程从wait中唤醒时，都必须再次测试条件谓词。

\***通知**：尽量使用notifyAll而不是notify。因为多个线程可以基于不同的条件谓词在同一个条件队列上等待。

1. 安全的通知：每当一个对象放入缓存或从缓存中移走一个对象，就执行一次通知。
2. 优化：仅当put或take影响到这些状态转换时，才发出通知。

可以提升性能，但很难实现cause子类的安全很难处理。别用！别优化！

**显式的Condition对象**

**正如Lock是一种广义的内置锁，Condition也是一种广义的内置条件队列。**

1. 一个Condition和一个Lock关联在一起，就像一个条件队列和一个内置锁相关联在一起。
2. 与内置条件队列不同的是，对于每个Lock，可以有任意数量的Condition对象。
3. Condition对象继承了相关的Lock对象的公平性。

**Synchronizer剖析**

在ReentrantLock和Semaphore这两个接口之间存在许多共同点。这两个类都可以：

1. 用作“阀门”，即每次只允许一定数量的线程通过，并当线程到达阀门时，
   1. 可以通过（通过调用lock或acquire时成功返回）
   2. 也可以等待（在调用lock或acquire时阻塞）
   3. 还可以取消（在调用tryLock或tryAcquire是返回false，表示在指定的时间段内锁是不可用的或者无法获得许可）
2. 并且，这两个接口都支持可中断的、不可中断的以及限时的获取操作，
3. 并且也都支持等待线程执行公平或非公平的队列操作

（以上其实就是AQS提供的API，可重写和扩展）

事实上，它们在实现时都使用了一个共同的基类，即AbstractQueuedSynchroniser（AQS），这个类也是其他许多同步类的基类。**AQS是一个用于构建锁和同步器的框架**。

基于AQS来构建同步器的好处：

1. 极大的减少实现工作
2. 不必处理在多个位置上发生的竞争问题，在基于AQS来构建同步器中，只可能在一个时刻发生阻塞，从而降低上下文切换的开销，并提高吞吐量
3. 在设计AQS时充分考虑了可伸缩性，因此java.util.concurrent中所有基于AQS构建的同步器都能取得这个优势。

**AbstractQueuedSynchroniser**

在基于AQS构建的同步器类中，最基本的操作包括各种形式的获取操作和释放操作。

**获取**操作是一种依赖状态的操作，并且通常会阻塞。获取的语义有：

1. With a **lock or semaphore,** the meaning of acquire is straightforward ‐ acquire the lock or a permit ‐ and the caller may have to wait until the synchronizer is in a state where that can happen.
2. With **CountDownLatch**, acquire means "wait until the latch has reached its terminal state",
3. With **FutureTask**, it means "wait until the task has completed".

**“释放”**不是一个阻塞操作。执行“释放”时，所有在请求时被阻塞的线程都会开始执行。

一个类想成为状态依赖的类，它必须有一个**状态**。AQS管理了一个整数状态信息，可以通过getState、setState以及compareAndSetState等protected类型的方法进行操作。状态的语义有：

1. **ReentrantLock** uses it to represent the count of times the owning thread has acquired the lock,
2. **Semaphore** uses it to represent the number of permits remaining,
3. **FutureTask** uses it to represent the state of the task (not yet started, running, completed, cancelled).

\*根据同步器的不同，获取操作可以是独占的，也可以是非独占的

**java.util.concurrent同步器类中的AQS**

**ReenrtantLock**

ReenrtantLock只支持独占方式的获取操作，因此它实现了tryAcquire、tryRelease和isHeldExclusively

**Semaphore和CountDownLatch**

Semaphore将AQS的同步状态用于保存当前可用许可的数量。tryAcquireShared方法首先计算剩余许可的数量，如果没有足够的许可数量，返回失败。如果还有剩余的许可，那么tryAcquireShared会通过compareAndSetState以原子方式来降低许可的计数。

同样，tryRealeaseShared将增加许可计数，这可能会解除等待线程的阻塞状态，并且不断重试直到更新操作（CAS）成功。tryRealeaseShared的返回值表示在这次线程中解除了其他线程的阻塞。

CountDownLatch使用AQS的方式与Semaphore相似。

FutureTask

初看上去，FutureTask甚至不像一个同步器，但FutureTask的语义非常类似于闭锁。

State用于保存任务的状态：new\completing\normal\cancelled…

FutureTask还维护一些额外的状态变量，用来保存计算结果或者抛出异常。

此外，它还维护了一个引用，指向正在执行计算任务的线程。

**ReentrantReadWriteLock**

ReadWriteLock接口表示存在两个锁：一个读取锁和一个写入锁。但在基于AQS实现的ReentrantReadWriteLock中，单个AQS子类将同时管理读取加锁和写入加锁。

ReentrantRead-WriteLock uses 16 bits of the state for the write‐lock count, and the other 16 bits for the read‐lock count.

Operations on the read lock use the shared acquire and release methods; operations on the write lock use the exclusive acquire and release methods.

Internally, AQS maintains a queue of waiting threads, **keeping track of whether a thread has requested exclusive or shared access**.

In ReentrantRead-WriteLock, when the lock becomes available, if the thread at the head of the queue was looking for write access it will get it, and if the thread at the head of the queue was looking for read access, all queued threads up to the first writer will get it.

（锁可用时，FIFO队列第一个写（包含写）之前的都可以执行）

# 第15章 原子变量与非阻塞同步机制

在java.util.concurrent包的许多类中，例如Semaphore和ConcurrentLinkedQueue，都提供了比synchronised机制更高的性能和可伸缩性。本章将介绍这种性能提升的主要来源：原子变量和非阻塞的同步机制。

非阻塞算法用底层的原子机器指令（例如compare‐and‐swap）代替锁来确保数据在并发访问中的一致性。非阻塞算法被广泛地用于在操作系统和JVM中实现线程/进程调度机制、垃圾回收机制以及锁和其他并发数据结构。

与基于锁的方案相比，非阻塞算法在设计和实现上复杂得多，但在性能和可伸缩性上有巨大优势。a.由于非阻塞算法可以使多个线程在竞争相同的数据时不会发生阻塞，因此它能在粒度更细的层次上进行协调，并且极大地减少调度开销。b.而且，在非阻塞算法中不存在死锁和其他活跃性问题。在基于锁的算法中，如果一个线程在休眠或自旋的同时持有一个锁，那么其他线程都无法执行下去，而非阻塞算法不会受到单个线程失败的影响。从Java5.0开始，可以使用原子变量类（例如AtomicInteger和AtomicReference）来构建高效的非阻塞算法。

即使原子变量没有用于非阻塞算法的开发，它们也可以用作一种“更好的volatile类型变量”。原子变量实现了volatile的语义，还多了原子化更新，从而使原子变量适用于实现：计数器、SequenceGenerator（JPA中用到，用于生成一个主键的序列）

15.1 锁的劣势

15.2 硬件对并发的支持

15.3 原子变量类

15.4 非阻塞算法

**锁的劣势**

优点/陈述：通过一致的锁定协议来协调对共享状态的访问，可以确保无论哪个线程持有守护变量的锁，都能采用独占方式来访问这些变量，并且对变量的任何修改对随后获得这个锁的其他线程都是可见的。

如果有多个线程同时请求锁，那么一些线程将被挂起\*并且在稍后恢复运行。当线程恢复运行时，必须等待其他线程执行完它们的时间片以后，才能被调度执行。在挂起和恢复线程等过程中存在巨大开销。

\*当线程在锁上发生竞争时，智能的JVM不一定会挂起线程，而是根据之前获取操作中对锁的持有时间长短，来判断使使此线程挂起还是自旋等待。

与锁相比，volatile是一种更轻量的同步机制，因为在使用这些变量时不存在上下文切换或线程调度等操作。然而，volatile虽然提供了相似的可见性保证，但是不能用于构建原子的复合操作。

锁还存在其他缺点。当一个线程正在等待锁时，它不能做任何其他事情。如果一个线程在持有锁的情况下被延迟执行（例如发生了缺页错误、调度延迟），所有等待这个锁的线程就永远无法执行下去。 “死锁”

\* 如果被阻塞的线程优先级高，持有锁的优先级低——优先级反转

**硬件对并发的支持**

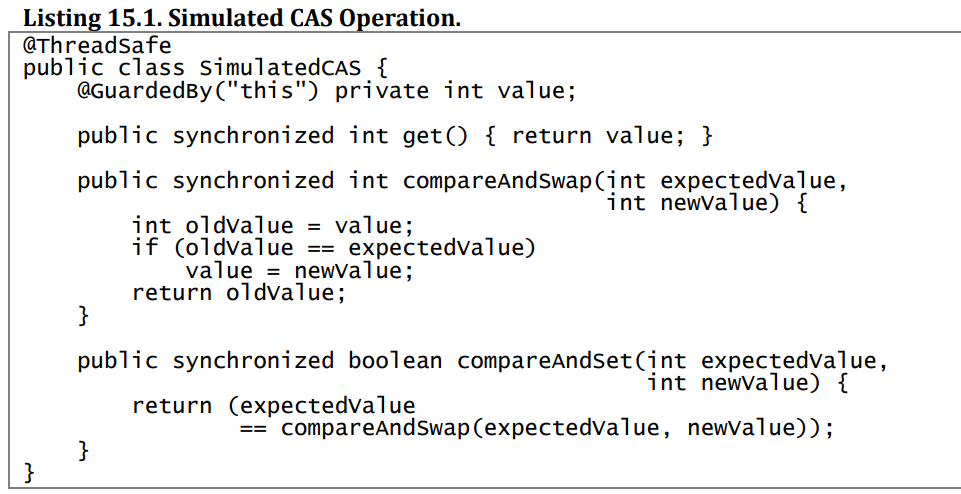
独占锁是一项悲观技术——它假设最坏的情况（只要不锁，就会被破坏），并且只有在确保其他线程不会造成干扰（通过占有锁）的情况下才能执行

对于细粒度的操作，还有一种更高效的、乐观的方法：这种方法需要借助冲突检查机制来判断在更新过程中是否存在来自其他线程的干扰，如果存在，这个操作将失败，并且可以重试。

几乎所有的现代处理器都包含了某种形式的原子读-改-写指令。操作系统和JVM使用这些指令来实现锁和并发的数据结构。

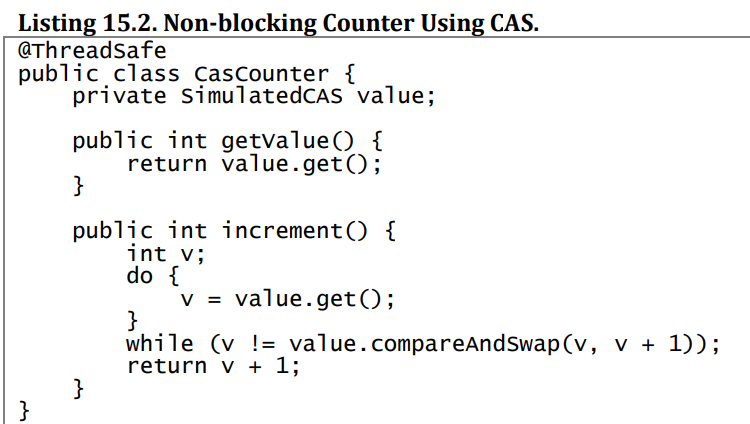
**Compare and Swap**

CAS包含了三个操作数——需要读写的内存位置V、进行比较的值A、拟写入的新值B



（模拟CAS语义，不是实现！）

当多个线程尝试使用CAS同时更新同一个变量时，只有一个线程可以更新成功。其他线程失败，但不会被挂起，而是被告知在这次竞争中失败，并且可以再次尝试。

应用举例：

\*若竞争激烈，也可以在重试之前先等待一段时间或者回退，从而避免活锁问题。

CAS比起锁：

看起来锁的代码简单，但是JVM和OS在管理锁时要完成的工作却并不简单，在实现锁定时需要便利JVM中一条非常复杂的代码路径，并可能导致OS级别的锁定、线程挂起、上下文切换。

这也是CAS的主要缺点，需要调用者自己处理竞争问题（通过重试、回退、放弃），自己设计算法；而在锁中能自动处理竞争问题（线程在获得锁之前将一直阻塞）

**JVM对CAS的支持：**

1. Java5.0中引入了对底层的支持，在int、long和对象的引用上都公开了CAS操作。在支持CAS的平台上，运行时编译成相应的（多条）机器指令，不支持CAS的机器上，JVM将使用自旋锁

2. 原子变量类AtomicXXX

3. java.util.concurrent中大多数类的实现中都使用了这些原子变量类

**原子变量类**

1. 原子变量类相当于一种泛化的volatile，但能够支持原子的和有条件的读-改-写操作

2. Integer和AtomicInteger都扩展了Number类，但AtomicInteger并不能扩展Integer类。a.包装类是不可修改的（final），原子变量类可以修改。b.原子变量类没有重写equals，每个实例都是不同的。

**非阻塞算法**

一个线程的失败或不会导致其他线程也失败或挂起。

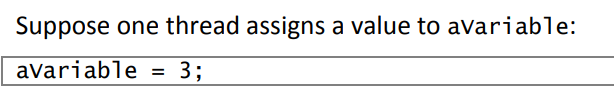
由专家设计。

有：非阻塞的栈、非阻塞的链表。。。

# 第16章 Java内存模型

**16.1 什么是内存模型，为什么需要它**

**16.2 发布**  -- 创建安全的单例



内存模型需要解决这个问题：“在什么条件下，读取aVariable的线程将看到这个值为3？”

看似弱智，但没有同步，确实有很多因素会使这种简单的结果不能成功发生。

比如：

1.编译器中的指令顺序与源代码中的顺序不一致 —— **重排序**为了优化执行

2.缓存可能会改变将写入变量提交到主内存的顺序

3.保存在处理器本地缓存中的值，对其他处理器不可见——不同的处理器提供了不同级别的缓存一致性

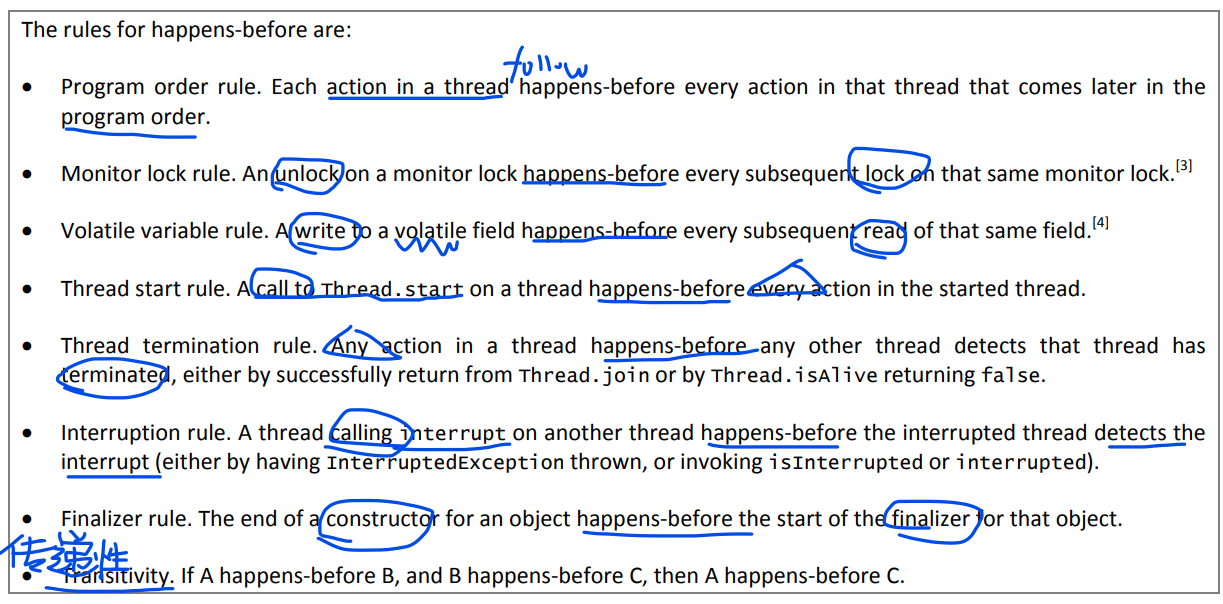
Java语言规范要求JVM在线程中维护一种类似串行的语义：只要程序的最终结果与在严格串行环境中执行的结果相同，那么上述所有操作都是允许的。 --黑盒

JMM规定了JVM必须遵循一组最小保证，which规定了对变量的写入操作在何时将对于其他线程可见。

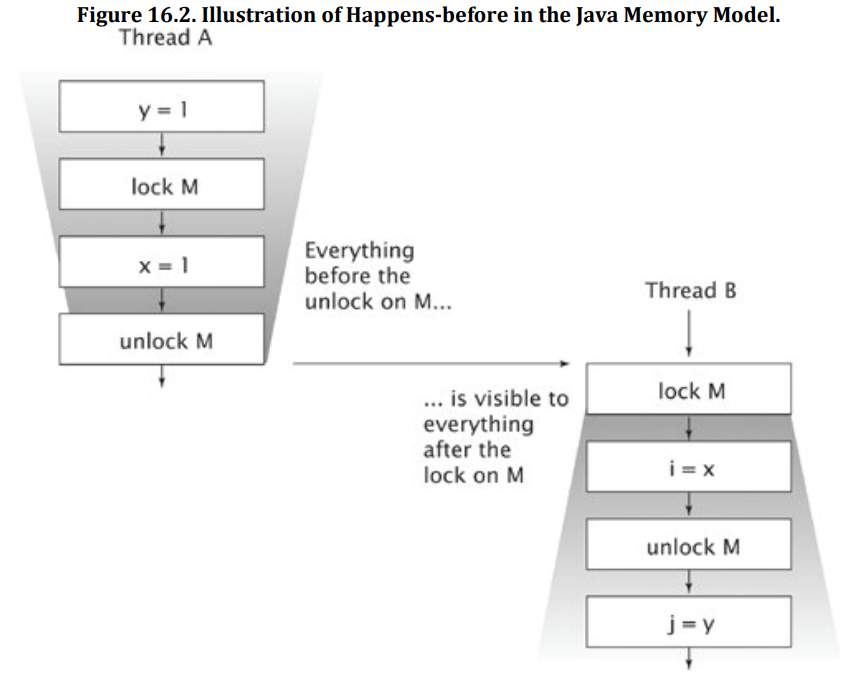
**Java内存模型简介**

Java内存模型是通过各种操作来定义的，包括对变量的读/写操作，监视器的加锁和释放操作，以及线程的启动和合并操作。

JMM为程序中所有操作定义了一个偏序关系，称之为Happens-Before。要想保证执行操作B的线程看到操作A的结果（无论A和B是否在同一个线程中执行），那么A和B之间必须满足Happens-Before关系。



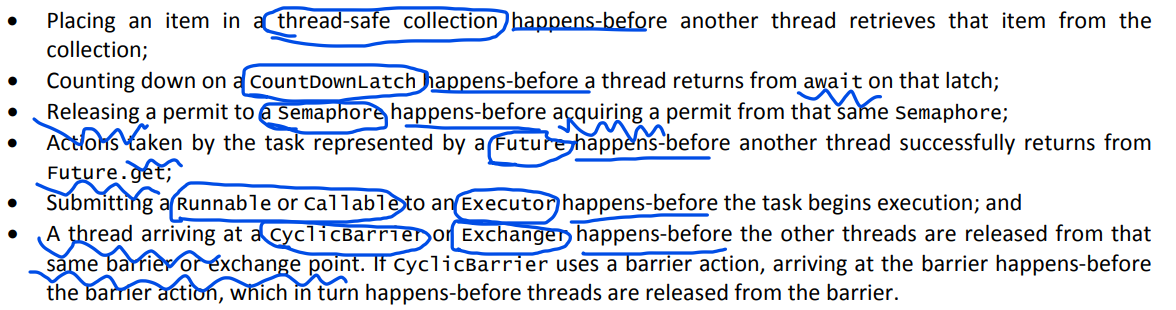
举例：当两个线程使用同一个锁进行同步时，它们之间的Happens-Before关系：



1. 线程A内部的执行顺序按照源程序的先后顺序，B也是

2. A释放了锁之后，B才获得了锁

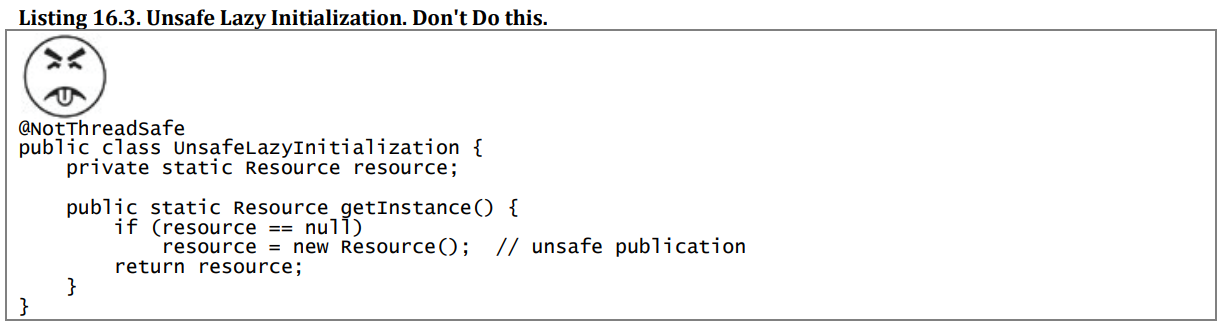
在类库中提供的其他Happens-Before保证包括：



安全的发布——**线程安全的单例模式**

1.延迟初始化

不安全的：



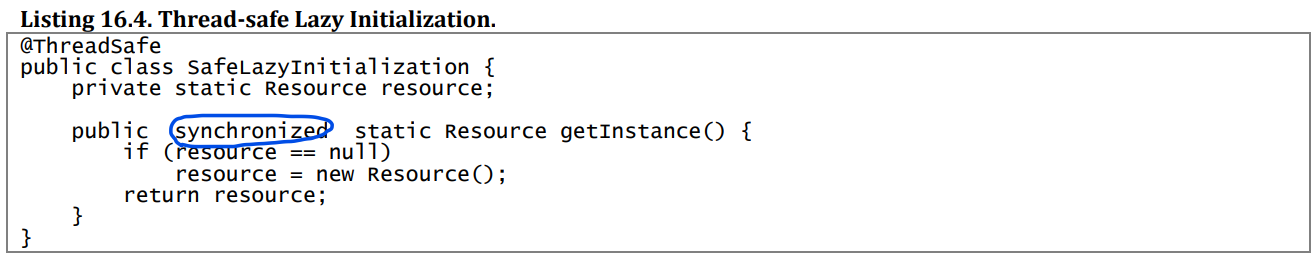
“不安全”在于：

a. 竞态条件 - 加载不止一次

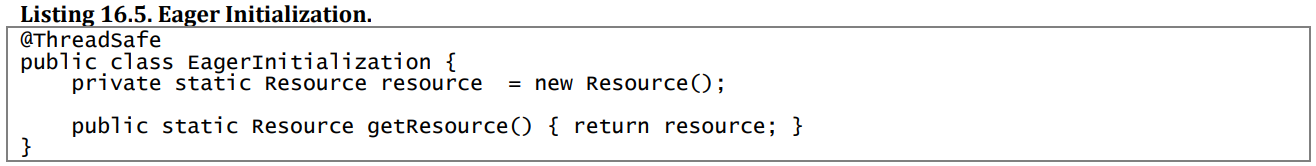
b. 重排序 – 如果无法确保发布共享引用的操作在另一个线程加载该共享引用之前执行，那么新对象引用的写入操作将与对象中各个域的写入操作重排序。

（如果resource是不可变的，那么其实也是安全的）

改进：



2. 线程安全的**提前初始化**Eager



在初始器中采用了特殊的方法来处理静态域，并提供额外的线程安全性保证。

静态初始化器是由JVM在类的初始化阶段执行，即在类被加载后并且被线程使用之前。

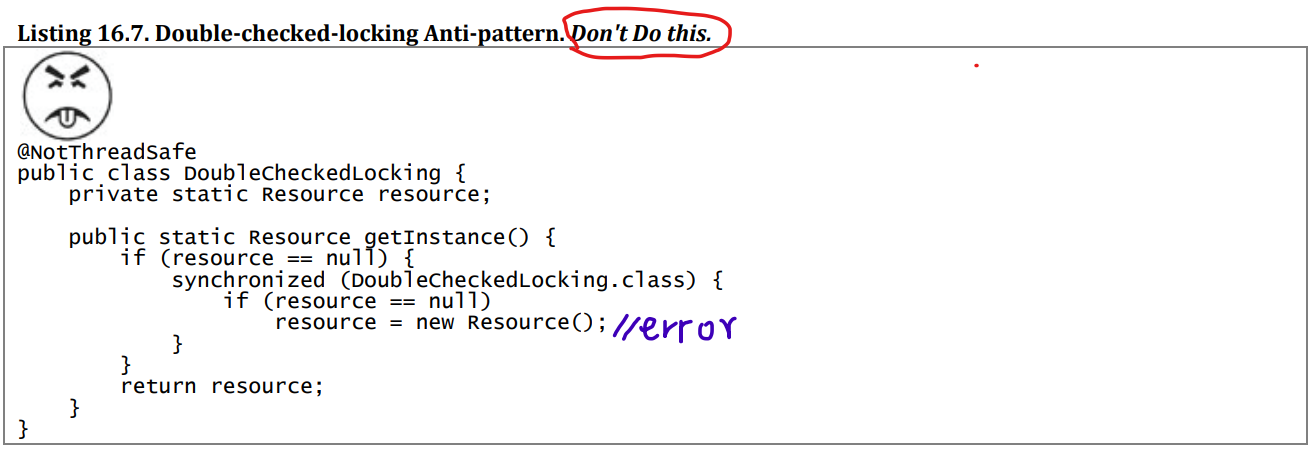
由于JVM将在初始化阶段获得一个锁，并且每个线程都至少获得一次这个锁以确保这个类已经加载，因此在静态初始化期间，内存写入操作将自动对所有线程可见。

\*这个规则仅适用于构造时，因此仅适用于对象是Immutable，如果可变仍要在读写线程中采用同步策略，确保修改的可见性。

优点：避免了每次调用懒汉中的getInstance中产生的同步开销

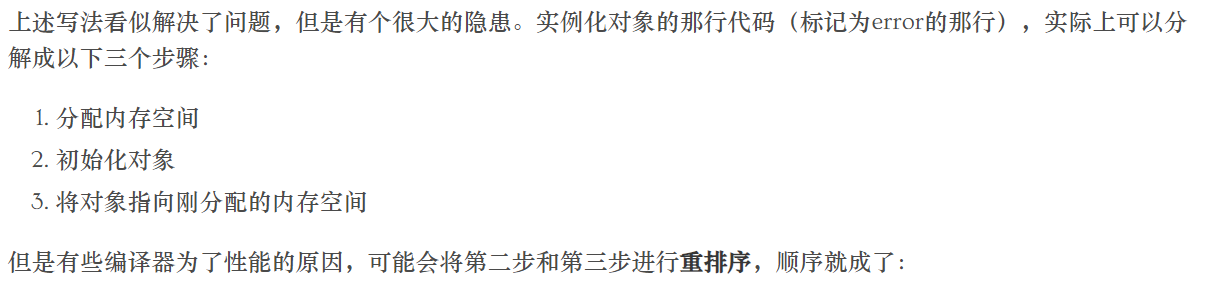
缺点：若不使用，内存浪费

3.**双重检查加锁** **×**



why“自作聪明的”？

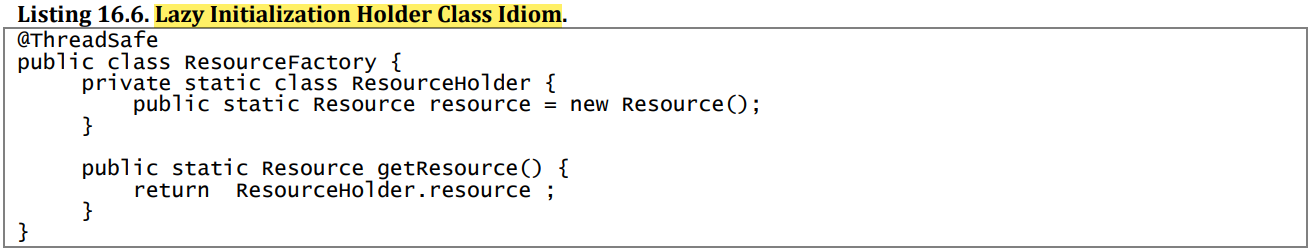
1. 并不安全->可以用volatile在声明时修饰resource



2. DCL的真正问题在于：当在没有同步的情况下读取一个共享对象时，最糟糕的情况是看到一个失效值which is null，但DCL将通过在持有锁的情况下再次尝试，线程有可能看到错误的对象。

3. DCL已被广泛地弃用，因为促使该模式出现的驱动力——无竞争同步的速度很慢、JVM启动很慢——已经不复存在。

4.**延迟初始化 占位类模式** **√**

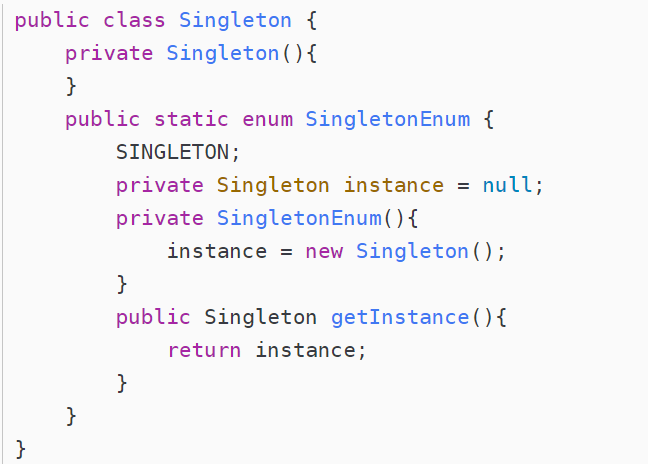


通过静态初始化+JVM的延迟加载机制结合。

另有，克隆、反射、反序列化对单例模式的破坏

解决：1. 单例类不能实现Coleanble接口

2. 使用枚举实现单例模式

point：内部静态单例类

**Java虚拟机会保证枚举类型不能被反射并且构造函数只被执行一次**。