## Java 并发编程

### 1 并发编程Bug的源头

Cpu、内存、I/O设备在不断的迭代，但是在快速发展的过程中，有一个核心矛盾存在，就是这三者的速度差异。

为了合理利用CPU的高性能，平衡这三者的速度差异，计算机体系机构、操作系统、编译程序都作出了贡献，主要体现为：

1. CPU增加了缓存，以均衡与内存的速度差异；
2. 操作系统增加了进程、线程，以分时复用CPU，进而均衡CPU与I/O设备的速度差异；
3. 编译程序优化指令执行次序，使得缓存能够得到更加合理地利用。

但是这些也是并发程序很多诡异问题的根源。

#### 缓存导致的可见性问题

在单核时代，所有的线程都是在一颗CPU上执行，CPU缓存与内存的数据一致性容易解决。因为所有线程都是操作同一个CPU的缓存，一个线程对缓存的写，对另一个线程来说一定是可见的。例如在下面图中，线程A和线程B都是操作同一个CPU里面的缓存，所以线程A更新了变量V的值，那么线程B之后再访问变量V，得到的一定是V的最新值（线程A写过的值）。

线程A

线程B

变量：V

CPU

变量：V

内存

CPU缓存和内存的关系图

一个线程对共享变量的修改，另外一个线程能够立刻看到，我们称之为可见性。

多核时代，每颗CPU都有自己的缓存，这是CPU缓存与内存的数据一致性就没那么容易解决了，当多个线程在不同的CPU上执行时，这些线程操作的是不同的CPU缓存。比如下图中，线程A操作的是CPU-1上的缓存，而线程B操作的是CPU-2上的缓存，很明显，这个时候线程A对变量V的操作对于线程B而言就不具备可见性。这个就属于硬件程序员给软件程序员挖的“坑”。

下面的一段代码验证了多核场景下的可见性问题。

线程A

线程B

变量：V

CPU1

变量：V

内存

CPU2

变量：V

多核CPU的缓存与内存关系图

|  |
| --- |
|  |

直觉告诉我们应该是20000，因为在单线程里调用两次add10K()方法，count的值就是20000，但是实际上calc()的执行结果是个10000到20000之间的随机数。为什么呢？

我们假设线程A和线程B同时开始执行，那么第一次都会将count=0读到各自的CPU缓存里，执行完count+1之后，各自缓存里的值都是1，同时写入内存后，我们会发现内存中是1，而不是我们期望的2。之后由于各自的CPU缓存里都有了count的值，两个线程都是基于CPU缓存里的count值来计算，所以导致最终count的值都是小于20000的。这就是缓存的可见性问题。

#### 线程切换带来的原子性问题

Java并发程序都是基于多线程的，因此会涉及到任务切换，但是任务切换是并发编程里Bug的源头之一。任务切换的时机大多数是在时间片结束的时候，JAVA是高级编程语言，高级编程语言里一条语句往往需要多条的CPU指令完成，例如上面代码中的count += 1，至少需要三条CPU指令。

### 3 互斥锁

#### 3.1简易锁模型

一般使用互斥解决锁。

加锁操作：lock()

临界区：一段代码

解锁操作：unlock()

简易锁模型

我们把一段需要互斥执行的代码称为临界区。线程在进入临界区之前，首先尝试加锁lock()，如果成功，则进入临界区，此时我们称这个线程持有锁；否则呢就等待，直到持有锁的线程解锁；持有锁的线程执行完临界区的代码后，执行解锁unlock()。

#### 3.2改进后的锁模型

在并发编程的世界里，锁和资源有对应关系。

首先，我们要把临界区要保护的资源标注出来，如图中临界区里增加一个元素：受保护的资源R；其次，我们要保护资源R就得为它创建一把锁LR；最后，针对这把锁LR，我们还需要在进出临界区时添加上加锁操作和解锁操作。另外，在锁LR和受保护资源之间是有关联的。

#### 3.2Java语言提供的锁技术：Synchronized

锁是一种通用的解决方案，Java语言提供的Synchronized关键字，就是锁的一种实现。Synchronized关键字可以用来修饰方法，也可以用来修饰代码块。

其中加锁和解锁的操作都是被java默默加上的，Java编译器会在Synchronized修饰的方法或者代码块前后自动加上加锁lock()和解锁unlock()，这样做的好处就是加锁lock()和解锁unlock()一定是成对出现的。

那么Synchronized里的加锁lock()和解锁unlock()锁定的对象在那里呢？上面的代码看到只有修饰代码块的时候，锁定了一个obj对象，那么修饰方法的时候锁定的是什么呢？这个也是Java的一条隐式规则：

当修饰静态方法的时候，锁定的是当前类的Class对象，在上面的例子中就是Class X;

当修饰非静态方法的时候，锁定的是当前实例对象的this。

### 5 死锁

在上一节中，我们使用Account.class作为互斥锁，来解决银行业务里面的转账问题，虽然方案不存在方案问题，但是所有的账户操作都是串行的，例如账户A转账户B，账户C转账户D这两个操作现实中是可以并行的，但是在这个方案中却被串行化了，这样的话，性能太差。

提高性能

在现实世界中，账户的转账操作是支持并发的，而且绝对是真正的并行，银行所有的窗口都可以做转账操作。只要我们能仿照现实世界做转账操作，串行的问题就解决了。

试想在古代，没有信息化，账户的存在的形式真的就是一个账本，而且每个账户都有一个账本，这些账本都统一放在文件架上。银行柜员在给我们做转账的时，要去文件架上把转出的账本和转入的账本都拿到手，然后做转账。这个柜员在拿账本的时候可能遇到以下三种情况：

1. 文件架上恰好有转出账本和转入账本，那就同时拿走；
2. 如果文件架上只有转出账本和转入账本之一，那这个柜员就先把文件架上有的账本拿到手，同时等着其他柜员把另外一个账本送回来；
3. 转出账本和转入账本都没有，那这个柜员就等着两个账本被送回。

上面这个过程在编程世界里怎么实现呢？其实用两把锁就可以了，转出账本一把，转入账本一把。在transfer()方法内部，我们尝试锁定转出账户this（先把转出账本拿到手），然后尝试锁定转入账户target（再把转入账本拿到手），只有两者都成功时，才执行转账操作。这个逻辑可以图形化为下图。

详细的代码优化如下：

没有免费的午餐

上面的实现看上去很完美，并且也算是将锁用的出神入化了。相对于用Account.class作为互斥锁，锁定的范围太大，而我们锁定两个账户范围就小很多了，这样的锁，叫做细粒度锁。使用细粒度锁可以提高并行度，是性能优化的一个重要手段。

但是使用细粒度锁是有代价的，这个代价就是可能会导致死锁。

如何预防死锁

并发程序一旦死锁，一般没有特别好的方法，很多时候只能重启应用。因此，解决死锁问题最好的方法还是规避死锁。

首先得知道什么时候出现死锁，下面四个条件发生时才会出现死锁：

1. 互斥，共享资源X和Y只能被一个线程占用；
2. 占有且等待，线程T1已经取得共享资源X，在等待共享资源Y的时候，不释放共享资源Y；
3. 不可抢占，其他资源不能抢占线程T1占有的资源；
4. 循环等待，线程T1等待线程T2占有的资源，线程T2等待T1占有的资源，就是循环等待。

反过来分析，也就是说，我们只要破坏其中一个，就可以成功避免死锁的发生。

其中，互斥这个条件我们没法破坏，因为我们用锁就是为了互斥。不过其他的三个条件都是有办法破坏掉的。

1. 对于“占用且等待”这个条件，我们一次性申请所有的资源，这样就不用等待了。
2. 对于“不可抢占“这个条件，占用部分资源的线程进一步申请其他资源的时候，如果申请不到，可以主动释放它占有的资源，这样不可抢占的这个条件就破坏掉了。
3. 对于“循环等待”这个条件，可以按序申请资源来预防。所谓按序申请，是指资源是有线性顺序的，申请的时候可以先申请资源小的，再申请资源大的，这样线性化后自然就不存在循环了。

1. 破坏占用且等待条件

从理论上讲，要破坏这个条件，可以一次性申请所有的资源。在现实世界里，就拿前面我们提到的转账操作来说，它需要两个资源，一个是转出账户，一个是转入账户，党费这两个账户同时被申请时，我们该怎么解决这个问题呢？

可以增加一个账本管理员，然后只允许账本管理员从文件架上拿账本，也就是说官员不能直接从文件架上拿账本，必须通过账本管理员才能拿到想要的账本。例如，张三同时申请账本A和B，账本管理员如果发现文件架上只有账本A，这个时候账本管理员是不会把账本A拿下来给张三的，只有账本A和B都在的时候才会给张三。这样就保证了“一次性申请所有的资源”。

对应到编程领域，“同时申请”这个操作是一个临界区，我们也需要一个角色（java里面的类）来管理这个临界区，我们把这个角色定义为Allocator。他有两个重要的功能，分别是：同时申请资源apply()和同时释放资源free()。账户Account类里面持有一个Allocator的单利（必须单例，只能由一个人来分配资源）。当账户Account在执行转账操作的时候，首先向Allocator同时申请转出账户和转入账户这两个资源，成功后再锁定这两个资源；当转账操作执行完，释放锁之后，我们需通知Allocator同时释放转出账户和转入账户这两个资源。

1. 破坏不可抢占条件

破坏不可抢占条件，这一条synchronized是做不到的。原因是synchronized申请资源的时候，如果申请不到，线程直接进入阻塞状态了，而线程进入阻塞状态，啥都干不了，也释放不了已经占用的资源。

在java.util.concurrent这个包下面提供的Lock是可以轻松解决这个问题的。

1. 破坏循环等待条件，需要对资源进行排序，然后按序申请资源。这个实现实现很简单，我们假设每个账户都有不同的属性ID，这ID可以作为排序字段，申请的时候，我们可以按照从小到大的顺序来申请。比如下面代码中，1~6处的代码对转账（this）和转入账户(target)排序，然后按照序号从小到大的顺序锁定账户，这样就不存在“”循环等待了。

### 6 用“等待-通知”机制优化循环等待

在破坏占用且等待条件的时候，如果转出账本和转入账本不满足同时在文本架上这个条件，就用死循环的方式来循环等待。

如果apply()操作耗时非常短，而且并发冲突量也不大时，这个方案还挺不错的，因为这种场景下，循环上几十次或者几十次就能一次性获取转出账户和转入账户了。但是如果apply()操作耗时长，或者并发冲突量大的时候，循环等待这种方案就不适用了，因为在这种场景下，可能要循环上万次才能获取到锁，太消耗CPU了。

其实在这种场景下，最好的方案应该是：如果线程要求的条件（转出账本和转入账本同在文件架上）不满足，则线程阻塞自己，进入等待状态；当线程要求的条件（转出账本和转入账本同在文本架上）满足后，通知等待的线程重新执行的。其中，使用线程阻塞的方式就能避免循环等待消耗CPU的问题。

那Java语言是否支持这种等待-通知机制呢？答案是：一定支持。

完美的就医流程

在介绍Java语言如何支持等待-通知机制之前，我们先看一个现实世界里面的就医流程，因为它有着完善的等待-通知，所以对比就医流程，我们能够更好地理解和应用并发编程中的等待-通知机制。

完美的就医流程

现实世界的就医流程，有着完善的等待-通知机制。

就医流程基本上是这样：

1. 患者先去挂号，然后到旧镇门口分诊，等待叫号；
2. 当叫到自己的号时，患者就可以找大夫就诊了；
3. 就诊过程中，大夫可能会让患者取做检查，同时叫下一位患者；
4. 当患者做完检查后，拿检测报告重新分诊，等待叫号；
5. 当大夫再次叫到自己的号时，患者再去找大夫就诊。

下面我们来对比看一下前面忽视了哪些细节。

1. 患者到就诊门口分诊，类似于线程去获取互斥锁；当患者被叫到时，类似线程已经获取到锁。
2. 大夫让患者去做检查（缺乏检测报告不能诊断病因），类似于线程要求的条件没有满足。
3. 患者去做检查，类似于线程进入等待状态；然后大夫叫下一个患者，这个步骤我们在前面等待-通知机制忽视了，这个步骤对应到程序里，本质是线程释放持有的互斥锁。
4. 患者做完检查，类似于线程要求的条件已经满足；患者拿检测报告重新分诊，类似于线程重新获取互斥锁，这个步骤在我们前面的等待-通知机制中忽视了。

所以完整的等待-通知机制：线程首先获取互斥锁，当线程要求的条件不能满足时，释放互斥锁，进入等待状态；当要求满足的时，通知等待的线程，重新获取互斥锁。

用synchronized实现等待-通知机制

Java语言中内置的synchronized配置wait()、notify()、notifyAll()这三个方法就能轻松实现。

用synchronized实现互斥锁。在下面的图中，左边有一个等待队列，同一时刻，只允许一个线程进入synchronized保护的临界区，当一个线程进入临界区后，其他线程就只能进入图中左边的等待队列等待。这个等待队列和互斥锁是一对一的关系，每个互斥锁都有自己独立的等待队列。

在并发程序中，当一个线程进入临界区后，由于某些条件不满足，需要进入等待状态，Java对象的wait()方法就能满足这种需求。入上图所示，当调用wait()之后，当前的线程就会被阻塞，并且进入到右边的等待队列中，这个等待队列也是互斥锁的等待队列。线程在进入等待队列的同时，会释放持有的互斥锁，线程释放锁后，其他的线程就有机会获得锁，进入临界区了。

那线程要求的条件满足时，该怎么通知这个等待的线程呢？很简单，就是Java对象的notify()和notifyAll()方法。下面图中所示，当条件满足时调用notify()，会通知等待队列（互斥锁的等待队列）中的线程，告诉它条件曾经满足过。

为什么说，曾经满足过？因为notify()只能保证在通知的时间点，条件满足的。而被通知线程的执行时间点和通知时间点基本上不会重合，所以当线程执行的时候，很可能条件已经不满足了（保不齐有其他线程插队）。这一点需要格外注意。

除此之外，还有一点需要注意，被通知的线程要想重新执行，仍然需要获取到互斥锁（因为曾经获取的锁在调用wait()时已经释放了）。

上面我们一直强调wait()、notify()、notifyAll()方法操作的等待队列是互斥锁的等待队列，所以如果synchronized锁定的是this，那么对应的一定是this.wait()、this.notify()、this.notifyAll()。

而且wait()、notify()、notifyAll()这三个方法能够调用的前提是已经获取了相应的互斥锁，所以我们 会发现这三个都在Synchronized{}内部调用的。如果在Synchronized{}外部调用，或者锁定this，而调用target.wait()调用的话，JVM会抛出一个运行时异常：java.lang.IllegalMonitorStateException。

一个更好的资源分配器

如何解决一次性申请转出账户和转入账户的问题。在这个等待-通知机制中，我们需要考虑以下四个要素。

1. 互斥锁：上一节中我们提到Allcator需要是单利的，所以我们可以用this作为互斥锁。
2. 线程要求的条件：转出账户和转入账户都没有被分配过。
3. 何时等待：线程要求的条件不满足就等待。
4. 何时通知：当线程释放账户时就通知。

范式

While(条件不满足){

wait()；

}

利用这个范式可以解决上面提到的条件曾经满足过这个问题。因为当wait()返回时，有可能条件已经发生变化了，曾经满足条件，但是现在已经不满足了，所以要重新先检验条件是否满足。

尽量使用notifyAll()

为什么不使用notify()呢？这两者室友区别的，notify()是随机地通知等待队列中的一个线程，而notifyAll()会通知等待队列中的所有的线程。从感觉上来讲，应该是notify()更好，因为即便通知所有的线程，也只有一个线程能够进入临界区。但是存在风险，它的风险可能导致某些线程永远不会被通知到。

假设我们有资源A、B、C、D，线程1申请到了AB，线程2申请到了CD，此时线程3申请AB，会进入等待队列，线程4申请CD也会进入等待队列。我们再假设之后线程1归还了AB，如果使用notify()来通知等待队列中的线程，有可能被通知的是线程4，但线程4申请的是CD，所以此时线程4还是会继续等待，而真正该唤醒的线程3就再也没有机会被唤醒。

所以除非深思熟虑，否则尽量使用notifyAll()。

Wait和sleep的区别

1. Sleep是Thread的方法，wait是Object类的方法；
2. Sleep方法调用的时候必须指定时间
3. wait会释放锁而sleep不会释放资源
4. Wait只能在同步方法和同步块中使用，而sleep任何地方都可以
5. wait无需捕捉异常，而sleep需要

### 7 安全性、活跃性以及性能问题

并发编程中我们需要注意的问题，主要有三个方面，分别是：安全性问题，活跃性问题和性能问题。

安全性问题

什么是线程安全呢，本质上就是真确性。而正确性的含义就是程序按照我们期望的执行，不要让我们感到意外。那如何写出线程安全的程序呢？在第一节已经介绍了并发Bug的三个来源：原子性问题、可见性问题和有序性问题。

那是不是所有的代码都需要分析是否存在这三个问题呢？当然不是，其实只有一种情况：存在共享数据并且该数据会发生变化，通俗的讲就是多个线程同时读写同一数据。那如果能够做到不共享数据和数据状态不发生变化，不就能够保证线程安全性了么。例如线程本地存储（thread Local Storage,TLS）、不变模式等等。

但是现实生活中，必须共享会发生变化的数据。当多个线程同时访问同一个数据，并且至少有一个线程会写这个数据的时候，如果我们不采取保护措施，那么就会导致并发Bug，即数据竞争。

那是不是在访问数据的地方，我们加个锁保护一下就可以解决所有的问题呢。

这个问题叫做竞态条件（Race Condition）。所谓的竞态条件。所谓竞态条件，指的是程序的执行结果依赖线程的执行顺序。

活跃性问题

所谓活跃性问题，指的是某个操作无法执行下去，我们常见的“死锁”就是一种典型的活跃性问题，当然除了死锁外，还有两种情况，分别是“活锁”和“饥饿”。

之前的章节讲过，发生“死锁”后线程会互相等待，而且会一直等待下去，在技术上的表现形式是线程的永久地“阻塞”了。

但有时线程虽然没有发生阻塞，但仍然会存在执行不下去的情况，这就是所谓的活锁。可以类比现实世界里的例子，路人甲从左手门出门，路人乙从右手门进门，两人为了不相撞，互相谦让，路人甲让路走右手边，路人乙也让路走左手边，结果两人又相撞了。这种情况，基本谦让几次就解决了，因为人会交流。但是在程序世界里，就有可能会一直没完没了的“谦让”下去，导致“活锁”。

解决“活锁”的方案很简单，谦让时，尝试等待一个随机的时间就可以了。“等待一个随机的时间”的方案虽然简单，却非常有效。

性能问题

使用“锁”要非常小心，但是如果小心过度，也可能出现“性能问题”。“锁”的过度使用可能导致串行化的范围过大，这样就不能够发挥多线程的优势了，而我们之所以使用多线程搞并发程序，为的就是提升性能。

所以我们要尽量减少串行，那串行对性能的影响是怎么样的呢？假设串行百分比是5%，我们用多核多线程相比单核单线程能提速多少。

### 10 JAVA线程（中）：创建多少线程才是合适的？

在java领域，实现并发程序的主要手段就是多线程，使用多线程还是比较简单的，但是使用多少个线程却是困哪的问题。

要解决这个问题，首先要分析以下两个问题：

1. 为什么要使用多线程？
2. 多线程的应用场景有哪些？

为什么要使用多线程？

使用多线程，本质上就是提升程序性能。但是首要的问题：如何度量性能。

度量性能的指标有很多，但是两个指标是最核心的，它们就是延迟和吞吐量。延迟指的是发出请求到收到响应这个过程的时间；延迟越短，意味着程序执行得越快，性能也越好。吞吐量指的是在单位时间内能处理请求的数量；吞吐量越大，意味着程序执行得越快，性能也越好。吞吐量指的是在单位时间内能够处理请求数量；吞吐量越大，意味着程序能够处理的请求越多，性能也越好。这两个指标内部有一定的关联（同等条件下，延迟越短，吞吐量越大），但是由于他们隶属于不同的维度（一个是时间维度，一个是空间维度），并不能互相转换。

我们所谓的性能，从度量的角度，主要是降低延迟，提高吞吐量。这个也是我们使用多线程的原因。

多线程的应用场景

要想“降低延迟，提高吞吐量”，对应的方法呢？基本有两个方向，一个方向是优化算法，另外一个是将硬件的性能发挥到极致。前者属于算法的范畴，后者则是和并发编程息息相关了。那计算机主要有哪些硬件呢？主要是两类：一个是I/O，一个是CPU。简而言之，在并发领域，提升性能的本质上就是提升硬件的利用率，再具体点来说，就是提升I/O的利用率和CPU的利用率。

下面用一个简单的示例来说明：如何利用多线程来提升CPU和I/O的利用率？假设程序按照CPU计算和I/O操作交叉的方式运行，而且CPU计算和I/O操作的耗时是1:1。

如下图所示，如果只有一个线程，执行CPU计算的时候，I/O设备空闲；执行I/O操作的时候，CPU空闲，所以CPU的利用率和I/O设备的利用率都是50%。

如果有两个线程，如下图所示，当线程A执行CPU计算的时候，线程B执行I/O操作；当线程A执行I/O操作的时候，线程B执行CPU计算，这样的CPU的利用率和I/O设备的利用率就都达到了100%。

因此如果CPU和I/O设备的利用率都很低，那么可以尝试通过增加线程来提高吞吐量。

在单核时代，多线程主要就是用来平衡CPU和I/O设备的。如果程序只有CPU计算，而没有I/O设备的话，多线程不但不会提升性能，还会使性能变得更差，原因是增加了线程切换的成本。但是在多核时代，这种纯计算型的程序也可以利用多线程来提升性能。为什么呢？因为利用多线程可以降低响应时间。

为了便于理解，这里举个简单的例子说明一下：计算1+2+... ...+ 10亿的值，如果在4核的CPU上利用4个线程执行，线程A计算[1,25亿)，线程B计算[25亿,50亿)，线程C计算[50,75亿)，线程D计算[75亿,100亿)，之后汇总，那么理论上应该比一个线程计算[1,100亿]快将近4倍，响应时间能够降低到25%。一个线程，对于4核的CPU，CPU的利用率只有25%，而4个线程利用率为100%。

创建多少个线程合适？

创建多少线程合适，要看多线程具体的应用场景。我们程序一般都是CPU计算和I/O操作交叉执行，由于I/O设备的速度相对于CPU来说很慢，所以大部分情况下，I/O操作执行的时间相对于CPU计算来说都非常长，这种场景我们一般称之为I/O密集型计算；和I/O密集型计算相对的就是CPU密集型计算了，CPU密集型计算大部分场景下都是纯CPU计算。I/O密集型程序和CPU密集型程序，计算最佳线程数的方法是不同的。

对于CPU密集型计算，多线程本质上是提升多核CPU的利用率，所以对于一个4核的CPU，每个核一个线程，理论上创建4个线程就可以了，再多创建线程也只是增加线程切换的成本。所以，对于CPU密集型的计算场景，理论上“线程的数量=CPU核数”就是最合适的。不过工程上，线程的数量一般会设置为“CPU核数+1”，这样的话，当线程因为偶尔的内存页失效或其他原因导致阻塞时，这个额外的线程可以顶上，从而保证CPU的利用率。

对于I/O密集型的计算场景，比如前面我们的例子中，如果CPU计算和I/O操作的耗时是1：1，那么2个线程是最合适的。如果CPU计算和I/O操作的耗时是1:2，那多少个线程合适呢？是3个线程，如下图所示：CPU在A、B、C三个线程之间切换，对于线程A，当CPU从B、C切换回来时，线程A正好执行完I/O操作。这样CPU和I/O设备的利用率都达到了100%。

通过上面的例子，我们会发现，对于I/O密集型计算的场景，最佳的线程数是与程序中CPU计算和I/O操作的耗时相关的，我们可以总结出这样的公式：

最佳线程数=1+（I/O耗时/CPU耗时）

我们令R=I/O耗时，综合上图，可以解释为：当线程A执行I/O操作时，另外R个线程正好执行完各自的CPU计算。这样CPU的利用率就达到了100%。

不过上面的公式针对单核CPU的，至于多核CPU，也很简单，只需要等比扩大就可以了，计算公式如下:

最佳的线程数=CPU核数\*[1+(I/O耗时/CPU耗时)]

Java 线程（下）：为什么局部变量是线程安全的？

前面小节讲到，多个线程同时访问共享变量的时候，会导致问题。在Java语言里，java方法里面的局部变量是否存在并发问题呢？以下为例。

比如：下面的代码里的fibonacci()方法，会根据传入的参数n，返回1到n的斐波那契数列，斐波那契数列类似这样：1、1、2、3、5、8、13、21、34... ...。在这个方法里面，有个局部变量：数组r用来保存数列的结果，每次计算完一项，都会更新数组r对应位置中的值。那么当多个线程调用fibonacci()这个方法的时候，数组r是否存在数据竞争(Data Race)呢？

方法是如何被执行的

高级语言里的普通语句，例如上面的r[i]=r[i-2]+r[i-1]；翻译成CPU的指令相对简单，可方法调用比较复杂了。例如下面这三行代码：第一行，声明一个int变量a;第2行，调用方法fibonacci(a)；第3行，将b赋值给c。

局部变量存哪里？

方法内的局部变量存哪里？

局部变量的作用域是方法内部，也就是是当方法执行完，局部变量就没用了，局部变量应该和方法同生共死。此时你应该会想到调用栈的栈帧，调用栈栈就是和方法同生共死的，所以局部变量放到调用栈里相当的合理。事实上，的确这样的，局部变量就是放到调用栈里。于是调用栈的结构就变成下图这样。

调用栈与线程

连个线程可以同时用不同的参数调用相同的方法，那调用栈和线程之间是什么关系呢？答案是：每个线程都有自己独立的调用栈。因为如果不是这样，那两个线程就相互干扰了。入下图所示，线程A、B、C每个线程都有自己独立的调用栈。

线程封闭

方法里的局部变量，因为不会和其他线程共享，所以没有并发问题，这个思路很好，已经成为解决并发问题的一个重要技术，就做线程封闭，即仅在单线程内访问数据。由于不存在共享，所以即便不同步也不会有并发问题。

采用线程封闭技术的案例非常多，例如从数据库连接池里获取的链接Connection，在JDBC规范里并没有要求这个Connection必须是线程安全的。数据库连接池通过线程封闭技术，保证一个Connection一旦被一个线程获取之后，在线程关闭Connection之前的这段时间里，不会再分配给其他线程，从而保证了Connection不会有并发的问题。

### 12 如何用面向对象的思想写好并发程序

在Java语言里，面向对象思想能够让并发编程变得更简单。

那如何才能用面向对象思想写好并发程序呢？可以从封装共享变量、识别共享变量间约束条件和制定并发访问策略这三方面下手。

1. 封装共享变量

并发程序，核心问题是解决多线程同时访问共享变量的问题。

面向对象思想里有一个很重要的特性是封装，封装的通俗解释就是将属性和实现细节封装在对象内部，外届对象只能通过目标对象提供的公共方法来间接访问这些内部属性。利用面向对象思想些并发程序的思路，很简单：将共享变量作为对象属性封装在内部，对所有公共方法制定并发访问策略。

### 30 如线程本地存储模式

如何避免共享呢，其实思路很简单，就是每个线程都拥有自己的变量，彼此之间不共享，也就没有并发问题。

我们在《为什么局部变量是线程安全的》中提到过线程封闭，其本质上就是避免共享。你已经知道通过局部变量可以做到避免共享，那还有没有其他方法做到呢？有的，Java语言提供的线程本地存储（ThreadLocal）就能做到。下面我们先看看ThreadLocal到底该如何使用。

ThreadLocal的使用方法

下面的静态类ThreadId会为每个线程分配一个唯一的线程Id，如果一个线程前后两次调用ThreadId的get()方法，两次get()方法的返回值是相同的。但如果是两个线程分别调用ThreadId的get()方法，那么线程看到的get()方法的返回值是不同的。

在Java中SimpleDateFormat不是线程安全的，那如果需要在并发场景下使用它，该怎么办呢。

其实有一个办法就是用ThreadLocal来解决，下面的示例代码就是ThreadLocal解决方案的具体实现。

ThreadLocal的工作原理

如果让你自己实现ThreadLocal的功能，你会怎么设计呢？ThreadLocal的目标是让不同的线程有不同的变量V，那直接的方法就是创建一个Map，它的Key是线程，Value是每个线程拥有的变量V，ThreadLocal内部持有这样一个Map就可以了。

然而Java不是这样设计的，Java的实现里面也有一个Map，叫做ThreadLocalMap，不过持有ThreadLocalMap，不过持有ThreadLocalMap的不是ThreadLocal，而是Thread。Thread这个类的内部有一个私有属性threadLocals，其类型就是ThreadLocalMap，ThreadLocalMap的Key这个类的内部有一个私有属性threadLocals，其类型就是ThreadLocalMap，ThreadLocalMap的Key是ThreadLocal。

在Java实现方案里面，ThreadLocal仅仅是一个代理的工具类，内部并不持有任何与线程相关的数据，所有和线程相关的数据都存储在Thread里面，这样的设计更容易理解。而从数据的亲缘性上来讲，ThreadLocalMap属于Thread也更合理。

当然还有一个更深次的原因，那就是不容易产生内存泄漏，在我们的设计方案中，ThreadLocal持有的Map会有Thread对象的引用，这就意味着，只要ThreadLocal对象存在，那么Map中Thread对象就永远不会被回收。ThreadLocal的生命周期往往都比线程要长，所以这种设计方案容易导致内存泄漏。而Java的实现中Thread持有ThreadLocalMap，而ThreadLocalMap里对ThreadLocal的引用还是弱引用（Weakreference），所以只要Thread对象可以被回收，那么ThreadLocalMap就能被回收。Java这样的实现方案虽然看上去复杂一些，但是更加安全。

但是即使如此也不能保证TreadLocal百分之一百的不内存泄漏。

ThreadLocal与内存泄漏