## 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。

### 21 原子类：无锁工具类的典范

前面我们多次提到一个累加器的例子，示例代码如下。在这个例子中，add10K()这个方法不是线程安全的，问题就出在变量count的可见性和count+=1的原子性上。可见性问题可以用volatile来解决，而原子性问题前面一直采用的是互斥锁的方案。

|  |
| --- |
| public class Test{  long count = 0;  void add 10k(){  int idx = 0;  while(idx ++ < 10000){  count += 1;  }  }  } |

其实对于简单的原子性，还有一种无锁的方案。Java SDK并发包将这种无锁方案封装提炼之后，实现了一系列的原子类。不过，在深入介绍原子类的实现之前，我们先看看如何利用原子类解决累加器问题。

在下面的代码中，我们将原来的long类型变量count替换成为了原子类AtomicLong，原来的Count+1 替换成了count.getAndIncrement()，仅需要这两处简单的改动就能使得add10K()方法编程线程安全的，原子类的使用还是很简单的。

|  |
| --- |
| public class Test{  AtomicLong count = new AtomicLong(0);  void add10K(){  int idx = 0;  while(idx++ < 10000){  count.getAndIncrement();  }  }  } |

无锁方案相对互斥锁的方案，最大的好处就是性能。互斥锁方案为了保证互斥性，需要执行加锁、解锁操作，而加锁、解锁操作本身就消耗性能；同时拿不到锁的线程还会进入阻塞状态，进而触发线程切换，线程切换对性能的消耗也很大。相比之下，无所方案则完全没有加锁、解锁的性能消耗，同时还能保证互斥性。

无锁方案的实现原理

其实原子类性能高的原因很简单，主要是硬件支持。CPU为了解决并发问题，提供了CAS指令（CAS全称是Compare And Swap，即“比较并交换”)。CAS指令包含3个参数：共享变量的内存地址A，用于比较的值B和共享变量的新增C；并且只有当内存中地址A处的值等于B时，才能将内存中地址A处的值更新为新值C。作为一条CPU指令，CAS指令本身是能够保证原子性的。

可以使用下面的CAS指令的模拟代码来理解CAS的工作原理。在下面的模拟程序中有两个参数，一个是期望值expect，另外一个是需要写入的新值newValue，只有当目前的count值和期望值expect相等的时候，才会将count更新为newValue。

|  |
| --- |
| class SimulatedCAS{  int count；  synchronized int cas(int expect,int newValue){  //du读muqian读目前countde的的zhi的值  int curValue = count;  //bijiao比较muqian比较的值是否==期望值  if(curValue == expect){  //如果是，则更新count的值  count = newValue；  }  返回写入前的值  return curValue;  }  } |

只有当目前count的值和期望的值expect相等的时候，才会将count更新为newValue。

对于前面提到的累加器的例子，count+=1的一个核心问题：基于内存中的count的当前值A计算出来的count+=1为A+1，在A+1写入内存的时候，很可能此时内存中count已经被其他线程更新过了，这样导致错误地覆盖其他线程写入的值。也就是说，只有当内存中的count的值等于期望值A时，才能将内存中Count的值更新为计算结果A+1。

使用CAS来解决并发问题，一般都会伴随着自旋，而所谓的自旋，其实就是循环尝试。例例如实现一个线程安全的count+=1操作，”CAS+自旋”的实现方案如下所示，首先计算newValue=count+1，如果cas(count,newValue）返回的值不等于Count，则意味着线程在执行完代码1处之后，执行代码2之前，count的值被其他的线程更新过。这时候可以采用自旋方案，入下面代码所示，可以重新读count最新的值来计算newValue并尝试再次更新，直到成功。

|  |
| --- |
| class SimulatedCAS{  volatile int count;  //实现count +=1;  addOne(){  do{  newValue = count +1; //1  }while(count != cas(count,newValue) //2  }  //模拟实现CAS  synchronzied int cas(int expect,int newValue){  //读目前count的值  int curValue = count;  //比较目前count值是否==期望值  if(curValue == expect){  //如果是，则更新count的值  count = newValue;  }  //返回写入前的值  return curValue;  }  } |

通过上面的示例代码，相比你已经发现了，CAS这种无锁的方案，完全没有加锁、解锁操作，即便两个线程完全同时执行addOne()方法，也不会有线程被阻塞，所以相对于互斥方案来说，性能好了很多。

但是在CAS方案中，有一个问题可能会常被忽略，那就是ABA问题，什么是ABA问题呢？

前面我们提到“如果cas(count,newValue)返回的值不等于count，意味着线程在执行完代码之后，执行代码2之前，count的值被其他的线程更新过”，那如果cas(count,newValue)返回的值等于count，是否就能够认为count的值没有被其他线程更新过呢？显然不是的，假设count的原本值是A，线程T1在执行完代码1处之后，执行代码2之前，有可能count被线程T2更新成B，之后又被T3更新回了A，这样线程T1虽然看到的一直是A，但是已经被其他的线程更新过，这就是ABA的问题。

可能大多数情况下，我们并不关心ABA问题，例如数值的原子递增，但也不能所有的情况都不关心，例如原子化的更新对象很可能就需要关心ABA问题，因为两个A虽然相等，但是第二个A的属性可能已经发生变化了。所以在使用CAS方案的时候，一定要先Check一下。

看Java如何实现原子化的count+=1

在本文开始部分，我们使用原子类AtomicLong的getAndIncrement()方法替代了count+=1，从而实现了线程安全。原子类AtomicLong的getAndIncrement()方法内部就是基于CAS实现的，下面我们来看看Java是如何使用CAS来实现原子化的count+=1的。

在JAVA1.8版本中，getAndIncrement()方法会转调unsafe.getAndAddLone()方法。这里this和valueOffset两个参数可以确定共享变量的内存地址。

|  |
| --- |
| final long getAndIncrement(){  return unsafe.getAndAddLong(this,valueOffset,1L);  } |

Unsafe.getAndAddLong()方法的源码如下，该方法首先会在内存中读取共享的变量的值，之后循环调用compareAndSwapLong()方法来尝试设置共享变量的值，知道成功为止。compareAndSwapLong()是一个native方法，只有当内存中共享变量的值等于expected时，才会将共享变量的值更新为x，并且返回true，否则返回false。compareAndSwapLong的语义和CAS指令的语义的差别仅仅是返回值不同而已。

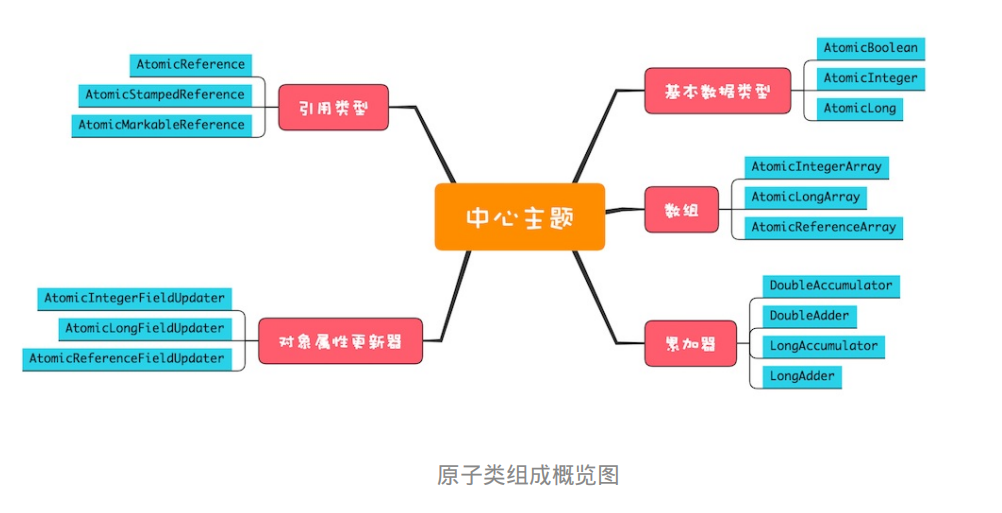
|  |
| --- |
| public final long getAndAddLong(Object o,long offset,long dalta){  long v;  do{  //读取内存的值  v = getLongVolatile(o,offset);    }while(!comapreaAndSeapLong(o,offesetmv,v\_delta));  return v;  }  //原子性地将变量更新为X  //条件是内存中的值等于expected  //更新成功则返回true  native boolean compareAndSwapLong(  Object o, long offset,  long expected,  long x); |

另外，需要你注意的是，getAndAddLong()方法的实现，基本上就是CAS使用的经典范例。Java提供的原子类里面CAS一般被实现为compareAndSet()，compareAndSet()的语义和CAS指令的语义的差别仅仅是返回值不同而已，compareAndSet()的语义和CAS指令的语义的差别仅仅是返回值不同而已，compareAndSet()里面如果更新成功，则返回true，否则返回false。

|  |
| --- |
| do{  //获取当前值  oldV = xxxx;  //根据当前值计算新值  newV = ...oldV...  }while(!compareAndSet(oldV,newV)); |

原子类概览

Java SDK并发包里提供的原子类内容很丰富，我们可以将它们分为五个类别：原子化的基本数据类型、原子化的对象引用类型、原子化数组、原子化对象属性更新器和原子化累加器。这五个类别提供的方法基本上是相似的，并且每个类别都有若干原子类，你可以通过下面的原子类组成的概览图获得一个全局的印象。



1. 原子化的基本类型

相关实现有AtomicBoolean、AtomicInteger和AtomicLong，提供的方法主要有以下这些。



1. 原子化的相关对象引用类型

相关实现有AtomicReference、AtomicStampedReference和AtomicMarkableReference，利用它们可以实现对象引用的原子化更新。AtomicReference提供的方法和原子化的基本数据类型差不多。不过需要注意的是，对象引用的更新需要重点关注ABA问题，AtomicStampedReference和AtomicMarkableReference这两个原子类可以解决ABA问题。

解决ABA问题的思路其实很简单，增加一个版本号维度就可以了，每次执行CAS操作，附加再更新一个版本号，只要保证版本号是递增的，那么即便A变成B在变回A，版本号也不会变回来。

1. 原子化数组

相关实现有AtomicIntegerArray、AtomicLongArray和AtomicReferenceArray，利用这些原子类，我们可以原子化地更新数组里的每一个元素。这些类提供的方法和原子化的基本数据类型仅仅的区别是：每个方法多了一个数组的索引引用。

1. 原子化对象属性更新器

相关实现有AtomicIntegerFiedldUpdater、AtomicLongFieldUpdater和AtomicReferenceFieldUpdater，利用它们可以实现原子化地更新对象的属性，这三个方法都是利用反射机制实现的，创建更新器的方法如下：

|  |
| --- |
| public static <U> AtomicXXXFieldUpdater<U>  newUpdater(Class<U> tclass,String fieldName) |

需要注意的是，对象属性必须是volatile类型的，只有这样才能保证可见性；如果对象属性不是volatile类型的，newUpdater()方法会抛出IllegalArgumentException这个运行时异常。

你会发现newUpdater()的方法参数只有类的信息，没有对象引用，而更新对象的属性，一定需要对象的引用，那这个参数是在那里引用的呢？ 是在原子操作方法参数中传入的。例如compareAndSet()这个原子操作，相比原子化的基本类型多了一个对象引用的obj。原子化对象属性更新器相关的方法，相比原子化的基本数据类型仅仅多了对象引用参数。

|  |
| --- |
| boolean compareAndSet(T obj,int expect,int update) |

1. 原子化的累加器

DoubleAccumulator、DoubleAdder、LongAccumulator和LongAdder，这四个类仅仅用来执行累加操作，相比于原子化的基本类型，速度更快，但是不支持compareAndSet()方法。如果你仅仅需要累加操作，使用原子化的累加器性能会更好。

总结

Java提供的原子类能够解决一些简单的原子性问题，但你可能会发现，上面我们所有的原子类方法都是针对一个共享变量的，如果你需要解决多个变量的原子性问题，建议还是使用互斥锁的方案。

### 22 Executor与线程池

线程是一个重量级的对象，应该避免频繁创建和销毁。可以使用线程池来解决。

普通的池化资源，都是下面这样，当你需要资源的时候就调用acquire()方法来申请资源，用完之后就调用release()释放资源。但是线程池里压根就没有申请线程和释放线程的方法。

|  |
| --- |
| class XXXPool{  //获取池化资源  XXX acquire(){  }  //释放池化资源  void release(XXX x){  }  } |

线程池是一种生产者-消费者模式

为什么线程池没有采用一般意义上池化资源的设计方法呢？如果线程池采用一般意义上池化资源的设计方法，应该是下面示例代码这样。你可以来思考一下，假设我们获取到一个空闲线程T1，然后该如何使用T1呢？你期望的可能是这样：通过调用T1的execute()方法，传入一个Runnable对象来执行具体的业务逻辑，就像通过构造函数Thread(Runnable target)创建线程一样。

|  |
| --- |
| //采用一般意义上池化资源的设计方法  class ThreadPool{  //获取空闲线程  Thread acquire(){  }  //释放线程  void release(Thread t){  }  }  //期望的使用  ThreadPool pool;  Thread T1 = pool.acquire();  //传入Runnable对象  T1.execute(() -> {  //具体业务逻辑  ...  }); |

所以，线程池的设计，没有办法直接采用一般意义上池化资源的设计方法，那线程池该如何设计呢？目前业界线程池的设计，普遍采用的是生产者-消费者模式。线程池的使用方是生产者，线程池本身是消费者。在下面的示例代码中，我们创建了一个简单的线程池MyThreadPool，你可以通过它来理解线程池的工作原理。

|  |
| --- |
| //简化的线程池，仅用来说明工作原理  class MyThreadPool{  //利用阻塞队列实现生产者-消费者模式  BlockingQueue<Runnable> workQueue;  //保存内部工作线程  List<WolerThread> threads = new ArrayList();  //构造方法  MyThreadPool(int poolSize,BolockingQueue<Runnable> workQueue){  this.workQueue = workQueue;  //创建工作线程  for(int idx=0; idx < poolSize; idx++){  WorkerThread work = new WorkerThread();  work.start();  threads.add(work);  }  }  //提交任务  void execute(Runnable command){  workQueue.put(command);  }  //工作线程负责消费任务，并执行任务  class WorkerThread extends Thread{  public void run(){  //循环取任务来执行  while(true){  Runnable task = workQueue.take();  task.run();  }  }  }  }  //下面使用示例  //创建有界阻塞队列  BlockingQueue<Runnable> workQueue = new LinkedBlockingQueue<2>;  //创建线程池  MyThreadPool pool = new MyThreadPool(10,workQueue);  //提交任务  pool.execute(() -> {System.out.println("hello");}); |

在MyThreadPool 的内部，我们维护了一个阻塞队列workQueue和一组工作线程，工作线程的个数由构造函数中的poolSize来指定。用户通过调用execute()方法来提交Runnable任务，execute()方法的内部实现仅仅是将任务加入workQueue中。MyThreadPool内部维护的工作线程会消费workQueue中的任务并执行任务。

如何使用Java中的线程池

Java并发包里提供的线程池相关的工具类中，最核心的是ThreadPoolExecutor的构造函数非常复杂。如下图所示，一共7个参数。

|  |
| --- |
| ThreadPoolExecutor(  int corePoolSize,  int maximumPoolSize,  long keepAliveTime,  TimeUnit unit,  BlockingQueue<Runnable> workQueue,  ThreadFactory threadFactory,  RejectedExecutionHandler handler) |

CorePoolSize：表示线程池保有的最小线程数。有些项目很闲，但是也不能把人都撤了，至少要留CorePoolSize个人坚守阵地

MaximumPoolSize：表示线程池创建的最大线程数。当项目很忙的时候，就需要加人，但是也不能无限制地加，最多就加到maximunmPoolSize个人。当项目闲下来时，就要撤人了，最多能撤到corePoolSize个人。

KeepAliveTime&Unit:上面提到项目根据忙闲来增减人员，那在编程世界里，如何定义忙和闲呢？很简单，一个线程如果在一段时间内，都没有执行执行任务，说明很闲，keepAliveTime和unit就是用来定义这个“一段时间”的参数，也就是说，如果一段时间内，都没有执行任务，说明很闲，keepAliveTime和unit就是用来定义这个“一段时间”的参数。也就是说，如果一个线程空闲了keepAliveTime&unit这么久，而且线程池的线程数大于corePoolSize，那么这个空闲的线程就要被回收了。

workQueue：工作队列，和上面的示例代码的工作队列同义

ThreadFactory:通过这个参数你可以自定义如何创建线程，例如你可以给线程指定一个有意义的名字。

Handler:通过这个参数你可以自定义任务的拒绝策略。如果线程池中的所有的线程都在忙碌，并且工作队列也满了（前提是工作队列是有界队列），那么此时提交任务，线程池就会拒绝接受。至于拒绝策略，你可以通过handler这个参数来指定。ThreadPoolExecutor已经提供了以下4种策略。

CallerRunsPolicy:提交任务的线程自己去执行该任务

AbortPolicy：默认的拒绝策略，会throws RejectedExecutionException

DiscardPolicy:直接丢弃任务，没有任何异常抛出

DiscardOldestPolicy：丢弃最老的任务，其实就是把最早进入工作队列的任务丢弃，然后把新任务加入到工作队列。

Java在1.6版本还增加了allowCoreThreadTimeout(boolean value)方法，他可以让所有的线程都支持超时，这意味着如果线程很闲，就会将项目组的成员都撤走。

使用线程池要注意什么

考虑到ThreadPoolExecutor的构造函数比较复杂，所以Java并发包里提供了一个线程池的静态工厂类Exectors，利用Executors你可以快速创建线程池。不过不建议使用，主要原因是：Executors提供的很多方法默认使用的都是无界的LinkedBlockingQueue，高负载的情况下，无界队列很容易导致OOM，而OOM会导致所有的请求都无法处理。所以强烈建议使用有界队列。

使用有界队列，当任务过多时，线程池会触发执行拒绝策略，线程池默认的拒绝策略会throw RejectedExecutionExecution这是个运行时异常，对于线程池的任务非常重要，建议自定义自己的拒绝策略；并且在实际工作中，自定义的拒绝策略往往和降级策略配合使用。

使用线程池还要注意异常处理的问题，例如通过ThreadPoolExecutor对象的Execute()方法提交任务时，如果任务在执行的过程中出现运行时异常，会导致执行任务的线程终止；不过，最致命的是任务虽然异常了，但是你却获取不到任何通知，这会让你误认为任务都执行得很正常。虽然线程池提供了很多用于异常处理的方法，但是最稳妥的和简单的方案还是补货所有的异常并按需处理。

|  |
| --- |
| try{  //业务逻辑  }catch(){  //按需处理  }catch(){  //按需处理  } |

### 23 Future：如何用多线程实现最优的“烧水泡茶”程序

ThreadPoolExecutor的void execute（Runable command）方法，利用这个方法虽然可以提交任务，但是却没有办法获取的执行结果（execute()方法没有返回值）。而很多场景下，我们又都是需要获取任务的执行结果的。下面来介绍一下使用ThreadPoolExecutor的时候，如何获取执行结果。

如何获取任务执行结果

Java通过ThreadPoolExecutor提供的3个submit()方法和1个FutureTask工具类来支持获得任务执行结果的需求。下面我们先来介绍这3个submit()方法，这3个方法的方法签名如下。

|  |
| --- |
| //提交Runnable任务  Future<?> submit(Runnable task);  //提交Callable任务  <T> Future<T> submit(Callable<T> task);  //提交Runnable任务及结果引用  <T> Future<T> submit(Runnbale task,T result); |

你会发现它们的返回值都是Future接口，Future接口有5个方法，它们分别是取消任务的方法cancel()、判断任务是否已取消的方法isCancelled()、判断任务是否已结束的方法isDone()以及2个获得任务执行结果的get()和get(timeout,unit)，其中最后一个get(timeout,unit)支持超时机制。通过Future接口的这5个方法你会发现，我们提交的任务不但能够后去任务执行结果，还可以取消任务。不过需要注意的是：这两个get()方法都是阻塞的，如果被调用的时候，任务还没有执行完，那么调用get()方法的线程会阻塞，直到任务执行完才被唤醒。

|  |
| --- |
| //取消任务  boolean cancel(boolean mayInterruptIfRunning);  //判断任务是否取消  boolean isCanceleld();  //判读任务是否已经结束  boolean isDone();  //获取任务结果  get();  //获取任务执行结果，支持超时  get(long timeout,TimeUnit unit); |

这3个submit()方法之间的区别在于方法参数不同，下面我们简要介绍一下。

1. 提交Runnable任务submit(Runnable task)：这个方法的参数是一个Runnable接口，Runnable接口的run()方法是没有返回值的，所以submit(Runnable task)这个方法返回的Future仅可以用来断言任务已经结束了，类似于Thread.join()。
2. 提交Callable任务submit(Callable<T> task)：这个方法的参数是一个Callable接口，它只有一个call()方法，并且这个方法是有返回值的，所以这个方法返回的Future对象可以通过调用其get()方法来获取任务的执行结果。
3. 提交Runnable任务及结果引用submit(Runnable task,T result):这个方法很有意思，假设这个方法返回的Future对象是f，f.get()的返回值就是传给submit()方法的参数result。这个方法该怎么用呢？下面这段示例代码展示了它的经典用法。需要你注意的是Runnable接口的实现类Task声明了一个有参构造Task(Result r)，创建Task对象的时候传入了result对象，这样就能在类Task的run()方法中对result进行各种操作了。result相当于主线程和子线程之间的桥梁，通过它主子线程可以共享数据。

|  |
| --- |
| ExecutorService executor = Executors.newFiexedThreadPool(1);  //创建Result对象r  Result r = new Result();  r.setAAA(a)  //提交任务  Future<Result> future = executor.submit(new Task(r),r);  Result fr = future.get();  //下面的灯饰成立  fr === r;  fr.getAAA() = a;  fr.getXXX() === x  class Task implements Runnable{  Result r;  //通过构造函数传入result  Task(Result r){  this.r = r;  }  void run(){  //可以操作rsult  a = r.getAAA();  r.setXXX(x);  }  } |

下面我们再来介绍Futuretask工具类。前面提到的Future是一个接口，而Futuretask是一个工具类，这个工具类有两个构造函数，它们的参数和前面介绍的submit()方法类似。

|  |
| --- |
| FutureTask(Callable<V> callable);  FutureTask(Runnable runnable,V result); |

那如何使用FutureTask呢？其实很简单，FutureTask实现了Runnable和Future接口，由于实现了Runnable接口，所以可以将FutureTask对象作为任务提交给ThreadPoolExecutor去执行，也可以直接被Thread执行；又因为实现了Future接口，所以也能用来获取任务的执行结果。下面的示例代码是将FutureTask对象提交给ThreadPoolExecutor去执行。

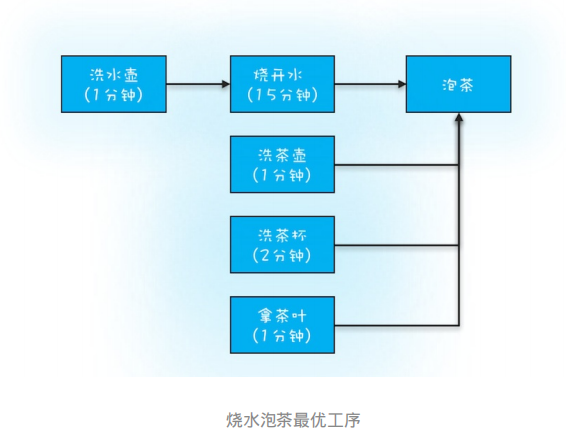
|  |
| --- |
| //创建FutureTask  FutureTask<Integer> futureTask = new FutureTask<>(() -> 1+2 );  //创建线程池  ExecutorService es = Executors.newCachedThreadPool();  //提交FutureTask  es.submit(futureTask);  //获取计算结果  Integer result = futureTask.get(); |

FutureTask对象直接被Thread执行的示例代码如下所示。

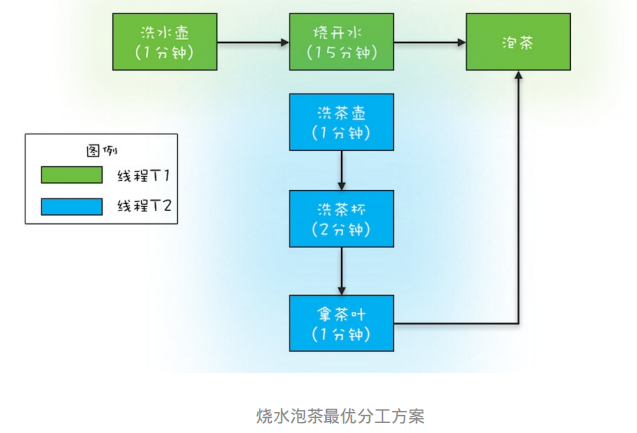
|  |
| --- |
| //创建FutureTask  FutureTask<Integer> futureTask = new FutureTask<>(() -> 1+2 );  //创建并启动线程  Thread T1 = new Thread(futureTask);  T1.start();  //提交FutureTask  es.submit(futureTask);  //获取计算结果  Integer result = futureTask.get(); |

实现最优的“烧水泡茶”程序

最优工序应该是下面的这样：



下面用程序来模拟一下这个最优的工序。并发编程可以总结为三个核心问题：分工、同步和互斥。编写并发程序，首先要做的就是分工，所谓的分工指的是如何高效地拆解任务并分配给线程。对于烧茶这个程序，一种最优的分工方案可以如下图所示的这样：用两个线程T1和T2来完成烧水泡茶程序，T1负责洗水壶、烧开水、泡茶这三道工序，T2负责洗茶壶、洗茶杯、拿茶叶三道工序，其中T1在执行泡茶这道工序时需要等待T2完成拿茶叶的工序。对于T1的这个等待动作，你应该可以想出很多种办法，例如Thread.join()、CountDownLatch，甚至阻塞队列都可以解决，这次使用Future特性来实现。



下面的示例代码就是用Future特性来实现。首先，我们创建了两个FutureTask--ft1和ft2,ft1完成洗水壶、烧开水、泡茶的任务，Ft2完成洗茶壶、洗茶杯、拿茶叶的任务；这里需要注意的是ft1这个任务在执行泡茶任务前，需要等待ft2把茶叶拿来，所以ft1内部需要引用ft2，并在泡茶之前，调用ft2的get()方法实现等待。

|  |
| --- |
| //创建任务T2的FutureTask  FutureTask<String> ft2 = new Futuretask<>(new T2task());  //创建任务T1的FutureTask  FutureTask<String> ft1 = new FutureTask<>(new T1Task(ft2));  //线程T1执行任务ft1  Thread T1 = new Thread(ft1);  T1.start();  //线程T2执行任务ft2  Thread T2 = new Thread(ft2);  T2.start();  //等待线程T1的执行结果  System.out.println(ft1.get());  //T1Task需要执行的任务  //洗水壶、烧开水、泡茶  class T1Task implements Callable<String>{  FutureTask<String> ft2;  //T1任务需要T2任务的FutureTask  T1Task(FutureTask<String> ft2){  this.ft2 = ft2;  }  String call() throws Exception{  System.out.println("T1:洗水壶...");  TimeUnit.SECONDS.sleep(1);  System.out.println("T2:烧开水...");  TimeUnit.SECONDS.sleep(15);  //获取T2线程的茶叶  String tf = ft2.get();  System.out.println("T1:拿到茶叶..." + tf);  System.out.println("T1:泡茶...");  return "上茶:" + tf;  }  class T1Task implements Callable<String>{  FutureTask<String> ft2;  //T1任务需要T2任务的FutureTask  T1Task(FutureTask<String> ft2){  this.ft2 = ft2;  }  String call() throws Exception{  System.out.println("T2:洗茶壶...");  TimeUnit.SECONDS.sleep(1);  System.out.println("T2:洗茶杯...");  TimeUnit.SECONDS.sleep(2);  System.out.println("T2:拿茶叶...");  TimeUnit.SECONDS.sleep(1);  return "龙井";  }  } |

### 24 CompletaBleFuture：异步编程没有那么难

串行转换成并行的过程中，一定会涉及到异步化，例如下面的代码，现在是串行的，为了提升性能，我们得把它并行化。

|  |
| --- |
| //以下两个方法都是耗时操作  doBizA();  doBizB(); |

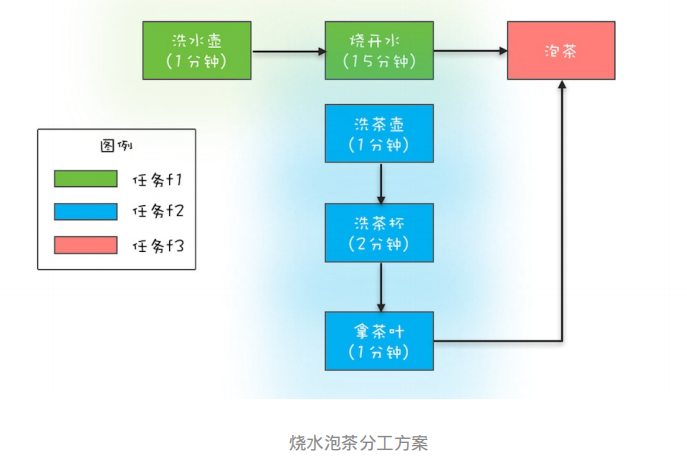
比较简单，就像下面代码中这样，创建两个子线程去执行就可以了。你会发现下面的并行方案，主线程无需等待doBizaA()和doBizB()的执行结果，也就是说doBizA()和doBizB()两个操作被异步化了。

|  |
| --- |
| new Thread(() -> doBizA()).start();  new Thread(() -> doBizB()).start(); |

异步化，是并行方案的得以实施的基础，更深入地讲其实就是：利用多线程优化性能这个核心方案得以实施的基础。Java在1.8版本提供了CompletableFuture来支持异步编程。

CompletableFuture的核心优势

我们利用CompletableFuture重新实现前面体积的烧水泡茶程序。首先还是需要先完成分工方案，下下面的程序中，我们分为3个任务：任务1负责洗水壶、烧开水，任务2负责洗茶壶、洗水杯和拿茶叶，任务3负责泡茶。其中任务3要等任务1和任务2都完成后才能开始。这个分工如下图所示。



下面是代码实现：

1. 无需手工维护线程，没有反锁的手工维护线程的工作，给任务分配线程的工作也不需要我们关注；
2. 语义更清晰，例如f3=f1.thenCombine(f2,()->{})能够清晰的表述“任务3要等待任务1和任务2都完成后才能开始”；
3. 代码更简练并且专注于业务逻辑，几乎所有代码都是业务逻辑相关的。

|  |
| --- |
| //任务1：洗水壶->烧开水  CompletableFuture<Void> f1 = CompletableFuture.runAsync(() -> {  System.out.printl("T1:洗水壶...");  sleep(1,TimeUnit.SECONDS);  Sytem.out.println("T1:烧开水...");  sleep(15,TimeUnit.SECONDS);  });  //任务2： 洗茶壶->洗茶杯->拿茶叶  CompletableFuture<Void> f2 = CompletableFuture.supplyAsync(() -> {  System.out.printl("T2:洗茶壶...");  sleep(1,TimeUnit.SECONDS);  Sytem.out.println("T2:烧茶杯...");  sleep(2,TimeUnit.SECONDS);  Sytem.out.println("T2:拿茶叶...");  sleep(1,TimeUnit.SECONDS);  return "龙井";  });  //任务3： 任务1和任务2完成后执行：泡茶  CompletableFuture<Void> f3 =  f1.thenCombine(f2,(\_,tf) -> {  System.out.println("T1:拿到茶叶:" + tf);  System.out.println("T1:泡茶");  return "上茶:" + tf;  });  //等待任务3执行结果  System.out.println(f3.join());  void sleep(int t, TimeInit u){  try{  u.sleep(t);  }catch(InterruptedException e){}  } |

创建CompletableFuture对象

创建CompletableFuture 对象主要靠下面代码中展示的4个静态方法，我们先看前两个。在烧茶泡茶的例子中，我们已经使用了runAsync(Runnable runnable)和supplyAsync(supplier<U> supplier)，它们之间的区别是：Runnable接口的run()方法没有返回值，而Supplier接口的get()方法是有返回值的。

前两个方法和后两个方法的区别在于：后两个方法可以指定线程池的参数。

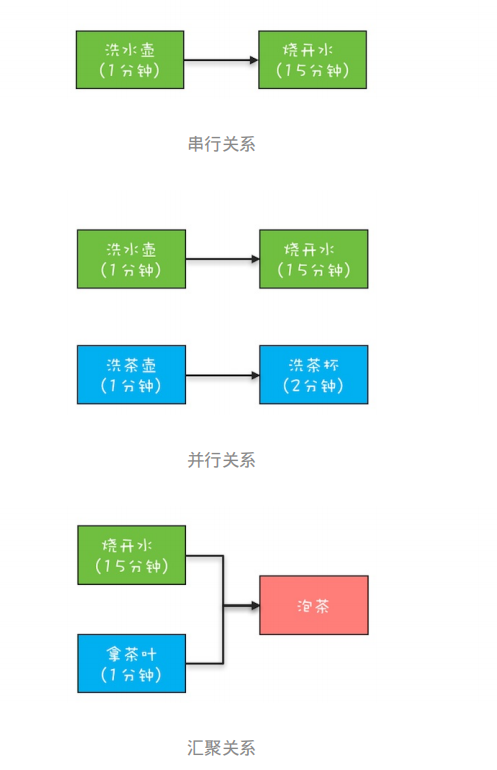
默认的情况下COmpletableFuture会使用公共的ForkJoinPool线程池，这个线程池默认创建的线程数是CPU的核数（也可以通过JVM option:-Djava.util.concurrent.ForkJoinPool. common.parallelism来设置ForkJoinPool线程池的线程数）。如果所有CompletableFuture共享一个线程池，一旦有任务执行一些慢的I/O操作，就会导致线程池中所有线程都阻塞在I/O 操作上，从而造成线程饥饿，进而影响整个系统的性能。所以，强烈建议你要根据不同的业务类型创建不同的线程池，以避免互相干扰。

|  |
| --- |
| //使用默认线程池  static CompletableFuture<void>  runAsync(Runnable runnable)  static CompletableFuture<U>  supplyAsync(Supplier<U> suppier)  //可以指定线程池  static CompletableFuture<Void>  runAsync(Runnable runnable,Executor executor)  static CompletableFuture<U>  supplyAsync(Supplier<U> supplier,Executor executor) |

创建完CompletableFuture 对象之后，会自动地异步执行runnable.run()方法或者supplier.get()方法，对于一个异步操作，你需要关注两个问题：一个是异步操作什么时候结束，另外一个是如何获取异步操作的执行结果。因为CompletableFuture类实现了Future接口，所以这两个问题你都可以通过Future接口来解决。另外，CompletableFuture类还实现了CompletionStage接口。

如何理解CompletionStage接口

可以站在分工的角度类比一下工作流。任务是有时序关系的，比如串行关系、并行关系、汇聚关系等。以烧水泡茶为例，其中烧水壶和烧开水就是串行关系，洗水壶、烧开水和洗茶壶、洗茶杯这两组任务之间就是并行关系，而烧开水、拿茶叶和泡茶就是汇聚关系。



CompletionStage接口可以清晰地描述任务之间的这种时序关系，例如前面提到的f3 = f1.thenCombine(f2,()->{}) 描述的就是一种汇聚关系。烧水泡茶程序中的汇聚关系是一种AND聚合关系，这里的AND指的是所有的依赖的任务（烧开水和拿茶叶）都完成后才开始执行当前任务（泡茶）。既然有AND聚合关系，那就一定还有OR聚合关系，所谓OR指的是依赖的任务只要有一个完成就可以执行当前任务。

在编程领域，还有一个问题就是异常处理，CompletionStage接口也可以方便地描述异常处理。

下面一一介绍，CompletionStage 接口如何描述串行关系，AND聚合关系、OR关系以及异常处理。

描述串行关系

CompletionStage接口里面描述串行关系，主要是thenApply、thenAccept、thenRun和tehnCompose这四个系列的接口

ThenApply系列函数里参数fn的类型是接口Function<T,,R>，这个接口里与CompletionStage相关的方法是R apply(T t)，这个方法既能接受参数，也支持返回值，所以thenApply系列方法返回的是CompletionStage<R>。

而thenAccept系统方法里参数consumer的类型是接口Consumer<T>，这个接口里与CompletionStage相关的方法是void accept(T t)，这个方法虽然支持参数，但却不支持返回值，所以thanAccept系列方法返回的是CompletionStage<Void>。

thenRun系列方法里的action的参数是Runnable，所以action既不能接受参数也不支持返回值，所以thenRun系列方法返回的也是 CompletionStage<Void>。

这些方法里面Async代表的是异步执行fn、consumer或者action。其中，需要你注意的是thenCompose系列方法，这个系列方法会创建出一个子流程，最终结果和thenApply系列是相同的。

通过下面的示例代码，你可以看一下thenApply()方法是如何使用的。首先通过suypplyAsync()启动一个异步流程，之后是两个串行操作，整体看起来还是很简单的。不过，虽然这是一个异步流程，但任务1,2,3却是串行执行的，2依赖1的执行结果，3依赖2的执行结果。

|  |
| --- |
| **CompletableFuture<String> f0**  **= CompletableFuture.supplyAsync(() -> "Hello World") //1**  **.thenApply(s -> s+ " QQ") //2**  **.thenApply(String::toUpperCase): //3**  **System.out.println(f0.join());** |

2.描述AND汇聚关系

CompletionStage接口里面描述AND汇聚关系，主要thenCombine、thenAcceptBoth和runAfterBoth 系列的接口，这些接口的区别也是源自fn、consumer、action 这三个核心参数不同。它们的使用你可以参考上面烧水泡茶的实现程序。

3.描述OR汇聚关系

CompletionStage 接口里面描述OR汇聚关系，主要是applyToEither、acceptEither和runAfterEither 系列的接口，这些接口的区别也是源自fn、consumer、action这三个核心参数不同。

下面的示例代码展示了如何使用applyToEither()方法来描述一个OR汇聚关系。

1. 异常处理

虽然上面我们提到的fn、consumer、action它们的核心方法都不允许抛出可检查异常，但是却无法限制它们抛出运行时异常，例如下面的代码，执行7/0就会出现除零错误这个运行时异常。

|  |
| --- |
| CompletableFuture<Integer> f0 = CompletableFuture  .supplyAsync(() -> (7/0))  .thenApply(r -> r\*10);  System.out.println(f0.join()); |

CompletionStage接口给我们提供的方案非常简单，比如try{}catch{} 还要简单，下面是相关的方法，使用这些方法进行异常处理和串行操作是一样的，都支持链式编程方式。

下面的示例代码展示了如何使用exceptionally()方法来处理异常，exceptionally()的使用非常类似于try{}catch{}中的catch{}，但是由于支持链式编程方式，所以相对更简单。既然有try{}catch{}，那就一定还有try{}finally{}，whenComplete()和handle()系列方法就类似于try{}finally{}中的finally{}，无论是否发生异常都会执行whenComplete()中的回调函数consumer和 handle()中的回调函数fn。whenComplete()和handle()的区别在于whenComplete()不支持返回结果，而handle()支持返回结果。

|  |
| --- |
| CompletableFuture<Integer> f0 = CompletableFuture  .supplyAsync(() -> (7/0))  .thenApply(r -> r\*10)  .exceptionally(e -> 0);  System.out.println(f0.join()); |

### 25 CompletionService：如何批量执行异步任务

如何优化一个询价应用的核心代码？如果采用“ThreadPoolExecutor+Future”的方案，你的优化结果很可能是下面示例代码这样：用三个线程异步执行询价，通过三次调用Future的get()方法获取询价结果，之后询价结果保存在数据库里

|  |
| --- |
| //创建线程池  ExecutorService executor =  Executors.newFixedThreadPool(3);  //异步向电商1询价  Future<Integer> f1 =  executor.submit(  ()->getPriceByS1());  //异步向电商2询价  Future<Integer> f2 =  executor.submit(  ()->getPriceByS2());  //异步向电商3询价  Future<Integer> f3 =  executor.submit(  ()->getPriceByS3());  //获取电商S1报价并异步保存  executor.execute(() -> save(f1.get()));  //获取电商S2报价并异步保存  executor.execute(() -> save(f2.get()));  //获取电商S3报价并异步保存  executor.execute(() -> save(f3.get())); |

上面的报价本身没有太大的问题，但是有个地方的处理需要注意，那如果获取电商S1报价的耗时很长，那么即便获取电商S2报价的耗时很短，也无法让保存S2报价的操作先执行，因为这个主线程都阻塞在了f1.get()操作上。怎么解决呢？

估计你也能想到，增加阻塞队列，获取到S1、S2、S3的报价都进入阻塞队列，然后在主线程中消费阻塞队列，这样就能保证先获取到的报价先保存到数据库了。下面的示例代码展示了如何利用阻塞队列实现先获取到的败家先保存到数据库。

|  |
| --- |
| //创建阻塞队列  BlockingQueue bq =new LinkedBlockingQueue<>();  //电商S1报价异步进入阻塞队列  executor.execute(() -> b1.put(f1.get());  //电商S2报价异步进入阻塞队列  executor.execute(() -> b1.put(f2.get());  //电商S3报价异步进入阻塞队列  executor.execute(() -> b1.put(f3.get());  //异步保存所有报价  for(int i =0 ; i < 3 ; i++){  Integer r = bq.take();  executor.executor(()->save(r));  } |

利用CompletionService实现询价系统

不过实际项目中，并不建议你这样做，因为Java SDK并发包里已经提供了设计精良的CompletionService。利用CompletionService不但能帮你解决先获取到的报价先保存到数据库的问题，而且还能让代码更简练。

CompletionService 的实现原理也是内部维护一个阻塞队列，当任务执行结束就把任务的执行结果加入到阻塞队列中，不同的是CompletionService是把任务结果的Future对象加入到阻塞队列中，而上面的示例代码是把任务最终结果放入到阻塞队列中。

那到底如何创建CompletionService 呢？

CompletionService 接口的实现类是ExecutorCompletionService，这个实现类的构造方法有两个，分别是：

|  |
| --- |
| ExecutorCompletionService(Executor executor);  EXecutorCompletionService(Executor executor,BlockingQueue<Future<V>> completionQueue) |

这两个构造方法都需要传入一个线程池，如果不指定completionQueue，那么默认会使用无界的LinkedBlockingQuueue，任务执行结果的Future对象就是加入到completionQueue中。

下面的示例代码完整展示了如何利用CompletionService来实现高性能的询价系统。其中，我们没有指定completionQueue，因此默认使用无界的LinkedBlockingQueue。之后通过CompletionServicey异步执行。最后通过CompletionService接口提供的take()方法获取一个Future对象，调用Future对象的get()方法就能返回询价操作的执行结果。

|  |
| --- |
| //创建线程池  ExecutorService executor = Executors.newFixedThreadPool(3);  //创建CompletionService  CompletionService<Integer> cs = new ExecutorCompletionService(executor);  //异步向S1询价  cs.submit(() -> getPricebyS1());  //异步向S2询价  cs.submit(() -> getPricebyS2());  //异步向S3询价  cs.submit(() -> getPricebyS3());  //异步保存所有报价  for(int i =0 ; i < 3 ; i++){  Integer r = cs.take().get();  executor.executor(()->save(r));  } |

CompletionSrevice接口说明

下面我们详细地地介绍一下ComoletionService接口提供的方法，ComoletionService接口方法提供的方法有五个。

Submit()相关的方法有两个。一个方法参数是Callable<V> task，前面利用CompletionServcie实现询价系统的示例代码中，我们提交任务就是用到它。另外一个方法有两个个参数，分别是Runnable task 和 V result，这个方法类似于ThreadPoolExecutor的<T> Future<T> submit(Runnable task,T result）。

CompletionService接口的其余3个方法，都是阻塞队列相关的，take()、poll()都是从阻塞队列中获取并且移除一个元素；它们的区别在于如果阻塞队列是空的，那么调用take()方法的线程会被阻塞，而poll()方法会返回null值。Poll(long timeout,TimeUnit unit)方法支持以超时的方式获取并移除阻塞队列头部的一个元素，如果等待了timeout unit时间，阻塞队列还是空的，那么该方法返回null值。

|  |
| --- |
| Future<V> submit(Callable<V> task);  Future<V> submit(Runnable task,V result);  Future<V> take() throws InterruptedException;  Future<V> poll;  Future<V> poll(long timeout,TimeUnit unit) throws InterruptedException; |

利用CompletionService实现Dubbo中的ForKing Cluster

Dubbo 中有一个叫做Forking的集群模式，这种集群模式下，并行支持地调用多个查询任务，只要有一个成功返回杰锅锅，整个服务就可以反悔了。例如你需要提供一个地址转坐标的服务，为了保证该服务的高可用和性能，你可以并行地调用3个地图服务商的API，然后只要1个正确返回了结果r，那么地址转坐标这个服务就可以直接返回r了。这种集群模式可以容忍2个地图服务商的异常，但是消耗资源偏多。

|  |
| --- |
| geocoder(addr){  //并行执行以下三个查询服务  r1 = gecoderByS1(addr);  r2 = gecoderByS2(addr);  r3 = gecoderByS3(addr);  //只要r1、r2、r3有一个返回，则返回  return r1|r2|r3  } |

利用CompletionService可以快速实现Forking这种集群模式，比如下面的示例代码展示了具体如何实现的。首先我们创建了一个线程池Executor、一个CompletionService对象cs和一个Future<Integer>类型的列表futures，每次通过调用CompletionService的submit()方法提交一个异步任务，会返回一个Future对象，我们把这些Future对象保存在列表futures中。通过调用cs.take().get()，我们能够拿到最快返回的执行结果，只要拿到一个，就可以取消所有任务并返回最终结果了。

|  |
| --- |
| //创建线程池  ExecutorService executor = Executors.newFixedThreadPool(3);  //创建CompletionService  CompletionService<Integer> cs = new ExecutorCompletionService<>(executor);  //用于保存Future对象  List<Future<Integer>> futures = new ArrayList<>(3);  //提交异步任务,并保存future到futures  futures.add(cs.submit(() -> geocoderByS1());  futures.add(cs.submit(() -> geocoderByS2());  futures.add(cs.submit(() -> geocoderByS3());  //获取最快返回的任务执行结果  Integer r = 0;  try{  //只要有一个成功返回，则break  for(int i = 0; i < 3; ++i){  r = cs.take().get();  //简单地通过判空来检查是否成功返回  if(r != null）{  break;  }  }  }fianlly{  //取消所有任务  for(Future<Integer> f : futures)  f.cancel(true);  }  //返回结果  return r; |

### 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

在线程池中使用ThreadLocal为什么可能导致内存泄漏呢？原因就出在线程池中线程存活的时间太长，往往都是和程序同生共死，这意味着Thread持有的ThreadLocalMap一直都不会被回收，再加上ThreadLocalMap中的Entry中的Entry对ThreadLocal的是弱引用，所以只要ThradLocal结束自己的生命周期是可以被回收掉的。但是Entry中的value却是被Entry强引用的，所以即便Value的生命周期结束，value也是无法被回收的

### 31 Guarded Suspension模式：等待唤醒机制的规范实现

背景：需要开发一个web项目，web版的文件浏览器，通过它用户可以在浏览器里查看服务器上的目录和文件。这个项目依赖运维部门提供的文件浏览服务，而这个文件浏览服务值支持消息队列（MQ）方式接入。消息队列的应用场景很多，主要用作流量的消峰和解耦。在这种接入方式中，发送消息和消费结果这两个操作之间是异步的。

在这个web项目中，用户通过浏览器发过来一个请求，会被转换成一个异步消息发送给MQ，等MQ返回结果后，再将这个结果返回至浏览器。但是现在的问题是：给MQ发送消息的线程是处理Web 请求的线程T1，但消息MQ的结果的线程并不是线程T1，那线程T1如何等待MQ返回结果呢？

Guarded Suspension 模式

Guarded Suspension直译过来就是“保护性地暂停”。

下图就是Guarded Suspension 模式的结构图，非常简单，一个对象GuardedObject，内部有一个成员变量--受保护的对象，以及两个成员方法--get（Predicate<T> p）和 onChanged（T obj）方法。参数p就是用来描述这个前提条件的；通过onChanged()方法可以fire一个事件，而这个事件往往能改变前提条件p的计算结果。

Guarded Object 的内部实现非常简单，是管程的一个经典用法，如下面代码所示。

get()方法通过变量的await()方法实现等待，onChanged()方法通过条件变量的signalAll()方法实现唤醒功能。逻辑还是很简单。

扩展Guarded Suspension 模式

Guarded Suspension 模式里Guarded Object 有两个核心方法，一个是get()方法，一个是onChanged()方法。很显然，在处理Web请求的方法handleWebReq()方法中，可以调用GuardedObject的get()方法来实现等待；在MQ消息的消费方法onMessage()中，可以调用GuardedObject的onChanged()方法来实现唤醒。

但是在实现的时候会遇到一个问题，handleWebReq()里面创建了GuardedObject对象的实例go，并调用其get()方法等待结果，那么在onMessage()方法中，如何才能够找到匹配的GuardedObject对象呢。

来扩展一下Guarded Suspension模式，从而使它能够很方便地解决这个问题。在这个程序中，每个发送到MQ的消息，都有唯一性的属性id，所以我们可以维护一个MQ消息的id和GuardedObject对象实例的关系。

|  |
| --- |
| class GuardedObject<T>{  //受保护的对象  T obj;  final Lock lock = new ReentrantLock();  final Condition done = lock.newCondition();  final int timeout = 2;  //保存所有GuardedObject  final static Map<Object,GuardedObject> gos = new ConcurrentHashMap<>();  //静态方法创建GuardedObject  static <K> GuardedObject create(K key){  GuardedObject go = new GuardedObject();  gos.put(key,go);  return go;  }  static<k,T> void fireEvent(K key,T obj){  GuardedObject go = gos.remove(key);  if(go != null){  go.onChanged(obj);  }  }  T get(Predicate<T> p){  lock.lock();  try{  //MESA管程推荐写法  while(!p.test(obj)){  done.await(timeout,TimeUnit.SECONDS);  }  }catch(InterruptedException e){  throw new RuntimeException(e);  }finally{  lock.unlock();  }  }  } |

定义了这个关系，看看具体怎么实现。下面代码扩展了Guarded Suspension模式的实现，扩展后的GuardedObject 内部维护了一个Map，其中Key是MQ消息id，而value是GuardObject对象实例，同时增加了静态方法create()和fireEvent()；create()方法用来创建一个GuardedObject对象的实例，并根据key值将其加入到Map中。

|  |
| --- |
| //处理浏览器发来的请求  Respond handleWebReq(){  int id = 序号生成器.get();  //创建一消息  Message msg1 = new Message(id,"{...}");  //创建GuardedObject实例  GuardedObject<Message> go = GuardedObject.create(id);  //发送消息  send(msg1);  //等待MQ消息  Message r = go.get(  t -> t != null);  }  void onMessage(Message msg){  //唤醒等待的线程  GuardedObject.fireEvent(  msg.id,msg)  } |

### 32 BalKing模式：再谈线程安全的单例模式

可以用多线程版本的if，来理解Guarded Suspension 模式，不同于单线程的if，这个多线程版本的if是需要等待的，必须条件为真。但是很显然在这个世界上，不是所有的场景都需要这么执着，有时候需要快速放弃。

需要快速放弃的一个常见的例子是各种编码器提供的自动保存功能。自动保存功能的实现逻辑一般都是隔一段时间自动存盘操作，存盘操作的前提是文件修改，如果没有做过文件修改，就需要快速放弃存盘操作。下面的示例代码将自动保存的功能代码化了，很显然AutoSaveEditor这个类不是线程安全的，因为共享变量changed的读写没有使用同步，那如何保证AutoSaveEditor的线程安全性呢？

|  |
| --- |
| Class AutoSaveEditor{  //文件是否被修改过  Boolean changed = false;  //定时任务线程池  ScheduledExecutorService ses = Excutors.newSingleThreadScheduledExecutor();  //定时执行保存操作  void startAutoSave(){  Ses.scheduleWithFixedDelay(()->{  autoSave();  },5,5,TimeUnit.SECONDS);  }  //自动存盘操作  void autoSave(){  if(!changed){  return;  }  changed = false;  //执行存盘操作  //省略且实现  this.execSave();  }  void edit(){  //省略编辑逻辑  .....  changed = true;  }  } |

读写共享变量changed的方法autoSave()和edit()都加互斥锁就可以了。这样做虽然简单，但是性能很差，因为是锁的范围太大了。那我们可以将锁的范围缩小，只在读写共享变量changed的地方加锁，实现代码如下所示。

|  |
| --- |
| //自动存盘操作  void autoSave(){  synchronized(this){  if(!changed){  return;  }  changed = false;  }  //执行存盘操作  //省略且实现  this.execSave();  }  //编辑操作  void edit(){  //省略编辑操作  sychronied(this){  changed = true;  }  } |

如果你深入地分析一下这个示例程序，你会发现，示例中的共享变量是一个状态变量。业务逻辑依赖于这个状态变量的状态：当状态满足这个条件时，执行某个业务逻辑，其实本质其实不过就是一个if而已，放到多线程场景里，就是一种“多线程版的if”。这种”多线程版的if”的应用场景还是很多的，所以也有人把它总结成了一种设计模式，叫做Balking模式。

Balking模式的经典实现

Balking模式本质上是一种规范化地解决：”多线程版本的if”的方案，对于上面自动保存的例子，使用Balking模式规范化之后的写法如下所示，你会发现仅仅是将edit()方法中对共享变量的changed的赋值操作抽取到了change()中，这样的好处是将并发处理逻辑和业务逻辑分开。

|  |
| --- |
| boolean changed = false;  //自动存盘操作  void autoSave(){  sychronized(this){  if(!changed){  return;  }  changed = false;  }  //执行存盘操作  //省略具体的实现  this.execSave()；  }  //编辑操作  void edit(){  //省略编辑逻辑  ...  changed();  }  void change(){  sychronized(this){  changed = true;  }  } |

用volatile实现Balking模式

在特定场景下，也可以使用volatile来实现，但使用valatile的前提是对原子性没有要求。

在《Coy-on-Write模式：不是延迟策略的COW》中，有一个RPC框架路由器表的案例，在RPC框架中，本地路由表是要和注册中心进行信息同步的，应用启动时候，会将应用依赖的服务的路由表从注册中心同步到本地路由表中，如果应用重新启动的时候注册中心宕机，那么会导致该应用依赖的服务均不可用，因为找不到依赖服务的路由表。为了防止这种极端的情况出现，RPC框架可以将本地路由表自动保存在本地文件中，如果启动的时候注册中心宕机，那么就从本地文件中恢复重启前的路由表。这其实也是一种降级的方案。

自动保存路由表和前面介绍的编辑器自动保存原理是一样的，也可以使用Balking模式实现，不过我们这里采用的是volatile来实现的，实现的代码如下。之所以使用volatile实现，是因为对共享变量changed和rt的写操作不存在原子性的要求，而且采用scheduleWithFixedDelay（）这种调度方式能够保证同一时刻只有一个线程执行autoSave()方法。

|  |
| --- |
| //路由表的信息  public class RouterTable{  //key: 接口名  //value:路由接口  ConcurrentHashMap<String,CopyOnWriteArraySet<Router>>  rt = new ConcurrnetHashMap<>();  //路由表是否发生变化  volatile boolean changed;  //路由表写入本地文件的线程池  ScheduledExcutorService ses =  Executors.newSingleThreadScheduledExecutor();  //启动定时任务  // 将变更后的路由表写入本地文件  public void startLocalSaver(){  ses.scheduleWithFixedDelay(()->{  autoSave();  },1,1,MINUTES);  }  //保存路由表到本地文件  void autoSave(){  if(!changed){  return;  }  changed = false;  //将路由表写入本地文件  //省略其方法实现  this.save2Local();  }  //删除路由  public void remove(Router router){  Set<Router> set = rt.get(router.iface);  if(set != null){  set.remove(router);  //路由表发生变化  changed = true;  }  }  public void add(Router router){  Set<Router> set = rt.computeIfAbsent(  route.iface,r ->  new CopyOnWriteArraySet<>());    set.add(router);  //路由表已经发生变化  changed = true;    }    } |

Balking 模式有一个非常典型的应用场景就是单次初始化，下面的示例代码是它的实现。这个实现方法中，我们将init()声明为一个同步方法，这样统一时刻就只有一个线程能够执行init()方法；init()方法在第一次执行完时会将inited设置为true，这样后续执行的init()方法的线程就不会再执行doInit()了。

|  |
| --- |
| class InitTest{  boolean inited = fales;  synchronized void init(){  if(inited){  return;  }  //省略doInit的实现  doInit();  inited = true;  }  } |

线程安全的单例模式本质上其实是单次初始化，所以可以用Balking模式来实现线程安全的单例模式，下面的示例代码是其实现。这个实现虽然功能上没有问题，但是性能很差，因为互斥锁synchronized将getInstance()方法串行化了，那有没有办法可以优化一下它的性能呢？

|  |
| --- |
| class Singleton{  private static Singleton singleton;  //构造方法私有化  private Singleton(){}  //获取实例（单例）  public synchronized static  Singleton getInstance(){  if(singleton == null){  singleton = new Singleton();  }  return singleton;  }  } |

办法当然是有的，那就是经典的双重检查（Double Check）方案，下面得代码是其详细实现。在双重检查的方案中，一旦Singleton对象被成功创建以后，就不会执行synchronized(Single.class){}相关的代码，也就是是，此时getInstance()方法是无锁的，从而解决了性能问题。不过需要注意的是，这个方案中使用了volatile来禁止编译优化，其原因之前有说明过。至于获取锁之后的二次检查，则是出于对安全性负责。

|  |
| --- |
| class Singleton{  private static Singleton singleton;  //构造方法私有化  private Singleton(){}  //获取实例（单例）  public static  Singleton getInstance(){  if(singleton == null){  synchronized(Singleton.class){  if(singleton == null){  singleton = new Singleton();  }  }  }  return singleton;  }  } |

总结

Balking模式和Guarded Suspension模式从实现上看似乎没有多大的关系，Balking模式只需要用互斥锁就能够解决，而Guarded Suspension 模式则要用到管程这种高级的并发原语；但是从应用的角度来看，它们解决的都是“线程安全的if”语义，不同之处，Guarded Suspension模式会等待if条件为真，而balking模式不会等待。

Balking模式的经典实现是使用互斥锁，你可以使用java内置的synchronized，也可以使用SDK提供的Lock；如果你对互斥锁的性能不满意，也可以采用volatile方案，不过使用volatile方案需要谨慎。

### 33 Thread-Per-Message模式：最简单使用的分工方法

并发编程领域的问题总结为三个核心的问题：分工、同步和互斥。其中，同步和互斥相关的问题更多的源自微观，而分工问题则是源自宏观。我们解决的问题，往往从宏观入手。解决并发问题，首要问题是解决宏观的分工问题。

在并发编程领域里，解决分工问题也有一系列的设计模式，比较常用的主要有Thread-Per-Message模式、Worker Thread模式、生产者-消费者模式。今天重点介绍Thread-Per-Message 模式。

如何理解Thread-Per-Message模式

Thread-Per-Message 模式是指为每个任务分配一个独立的线程。

用Thread实现Thread-Per-Message模式

Thread-Per-Message模式的一个最简单的应用场景是网络编程里服务端的实现，服务端为每个客户端请求创建一个独立的线程，当线程处理完成请求之后，自动销毁，这是一个最简单的并发处理网络请求的方法。

网络编程里最简单的程序当数echo程序可，echo程序的服务端会原封不动地将客户端的请求发送回客户端。例如，客户端发送TCP请求“Hello World”，那么服务端也返回“Hello World”。

下面我们就以echo程序的服务端为例，介绍如何实现Thread-Per-Message模式。

示例代码如下：

|  |
| --- |
| final ServerSocketChannel ssc =  ServerSocketChannel.open().bind(  new InetSocketAddress(8080));  //处理请求  try{  while(true){  //接受请求  SocketChannel sc = ssc.accpet();  //每个请求都创建一个线程  new Thread(()->{  try{  ByteBuffer rb = ByteBuffer.allocateDirect(1024);  sc.read(b);  //模拟请求处理  Thread.sleep(2000);  //写Socket  ByteBuffer wb = (ByteBuffer)rb.flip();  sc.write(wb);  //关闭socket  sc.close();  }catch(Exception e){  throw new UncheckedIoException(e);  }  }).start();  }finally{  ssc.close();  }  } |

上面的方案不具备可行性，因为线程是一个重量级的对象，创建成本很高，一方面创建线程比较耗时，另外一方面线程占用的内存比较大。所以，为每一个请求创建一个新的线程并不适合高并发的场景。

为了解决这个问题，java并发包中提供了线程池等工具类，这个方式是一个很稳妥的方案，但是不是唯一的方案。

业界还有另外一种方案，叫做轻量级线程。这个方案在java领域知名度并不高，在Go、Lua语言用的较多。轻量级的线程，创建成本很低，基本上和创建一个普通的对象成本相似；并且创建的速度和内存占用相比操作系统线程至少是一个数量级的提升，所以轻量级的线程实现Thread-Per-Message模式完全没有问题。

OpenJdk中有一个Loom项目，就是解决Java语言的轻量级线程的问题，在这个项目中，轻量级的线程被叫做Fiber。下面基于Fiber实现Thread-per-Message模式。

|  |
| --- |
| final ServerSocketChannel ssc =  ServerSocketChannel.open().bind(  new InetSocketAddress(8080));  //处理请求  try{  while(true){  //接受请求  SocketChannel sc = ssc.accpet();  //每个请求都创建一个线程  Fiber.scheduled(()->{  try{  ByteBuffer rb = ByteBuffer.allocateDirect(1024);  sc.read(b);  //模拟请求处理  LockSupport.parkNanos(2000\*1000000);  //写Socket  ByteBuffer wb = (ByteBuffer)rb.flip();  sc.write(wb);  //关闭socket  sc.close();  }catch(Exception e){  throw new UncheckedIoException(e);  }  });  }finally{  ssc.close();  }  } |

### 34 WorkerThread模式：如何避免重复创建线程

上一节中，我们介绍了一种最简单的分工模式--Thread-per-Message模式，对应到现实世界，其实就是委托代办。这种分工模式如果用Java Thread实现，频繁地创建、销毁线程非常影响性能，同时无限制地创建线程还可能导致OMM，所以在Java领域的使用场景就受限了。这节介绍Java领域使用最多的Worker Thread模式。

Worker Thread 模式及其实现

以echo 程序为例，看看如何用线程池实现。

下面的示例代码是用线程池实现的Echo服务端。

|  |
| --- |
| ExecutorService es =  Executors.newFixedThreadPool(500);  final ServerSocketChannel ssc =  ServerSocketChannel.open().bind(  new InteSocketAddress(8080));  //处理请求  try{  while(true){  //接受请求  SocketChannel sc = ssc.accept();  //将请求处理任务提交给线程池  es.execute(() -> {  try{  //读Socket  ByteBuffer rb = ByteBuffer  .allocateDirect(1024);  sc.read(rb);  //模拟处理请求  Thread.sleep(2000);  //写Socket  ByteBuffer wb = (ByteBuffer)rb.flip();  sc.write(wb);  //关闭Socket  sc.close();  }catch(Exception e){  throw new UncheckedIOexception(e);  }  });  }  }finally{  ssc.close();  es.shutdown();  } |

正确的创建线程池

Java的线程池既能够避免无限制地创建线程导致OMM，也能避免无限制地接受任务导致OMM。所以建议你用创建有界的队列来接受任务。

当请求量大于有界队列的容量时，就需要合理地拒绝请求。如何合理的拒绝，可以结合具体的业务场景来制定，即便线程池默认的拒绝策略能够满足你的需求，也同样建议你在创建线程池时，清晰地指明拒绝策略。

同时也建议在你实际工作中给线程赋予一个业务相关的名字。

下面是新的示例代码

|  |
| --- |
| ExecutorService es = new ThreadPoolExecutor(  50,500,60L,TimeUnit.SECONDS,  //注意要创建有界队列  new LinkedBlockingQueue<Runnable>(2000),  //建议根据业务需求实现ThreadFactory  r -> {  return new Thread(r,"echo-" + r.hashCode());  },  //建议根据业务需求实现RejectedExecutionHandler  new ThreadPoolExecutor.CallerRunsPolicy()); |

避免线程死锁

使用线程池过程中，还要注意一种线程死锁的场景。如果提交到相同线程池的任务不是相互独立的，而是有依赖关系的，那么就有可能导致线程死锁。

相关的逻辑精简后没如下图所示，该应用将一个大型的计算任务分为两个阶段，第一个阶段的任务会等待第二个阶段的子任务完成。在这个应用里，每一个阶段都是用了线程池，而且两个阶段是用的是同一个线程池。

使用下面的示例代码模拟应用，如果执行下面的代码会发现永远执行不到最后一行。执行过程中没有任何异常，但是应用已经停止响应了。

|  |
| --- |
| ExecutorService es = Executors.  newFixedThreadPool(2);  //L1阶段的闭锁  CountDownLatch l1 = new CountDownLatch(2);  for(int i = 0; i < 2 ; i++0){  System.out.println("L1");  //执行L1阶段任务  es.execute(() -> {  //L2 阶段的闭锁  CountDownLatch l2 = new CountDownLatch(2);  //执行L2阶段的任务  for(int j = 0; j < 2; j++){  es.execute(() -> {  System.out.println("L2");  l2.countDown();  });  }  //等待L2阶段任务执行完  l2.await();  l1.countDown();  });  }  //等待着L1阶段任务完成  l1.await();  System.out.println("end"); |

当应用出现类似问题时，首选的诊断方法是查看线程栈。下图是上面示例代码停止响应后的线程栈。下图是上面示例代码停止响应后的线程栈，你会发现线程池中两个线程都阻塞在l2.await() 这行代码上，也就是说，线程池里所有的线程都在等待L2阶段的任务整完，那L2阶段的子任务什么时间能够执行完呢？永远都执行不完，因为线程池里的线程都阻塞了，没有空闲的线程执行L2阶段的任务了。

原因找到了，那如何解决就简单了，最简单的方法就是将线程池的最大线程数调大，如果能够确认任务的数量不是非常多的话，这个方法是可行的，否则这个方法就行不通。其实这种问题通用的解决方法就是为不同的的任务。

最后总结一下：提交到相同线程池的任务一定是相互独立的，否则就一定要慎重。

### 35 两阶段的终止模式：如何优雅的终止线程

前面的两篇文章，从纯技术的角度看，都是启动多线程去执行一个异步任务。既启动，那如何终止呢？本节介绍如何优雅地终止线程。

在《Java线程的生命周期》中，我们说过：线程执行完或者出现异常就会进入终止状态。这样看终止一个线程终止，只需要执行完自己的任务，自己进入终止状态，这样很简单。今天我们讲的是“如何优雅的终止线程”，是在一个线程T1中，终止线程T2；这里所谓的“优雅”，指的是让线程T2执行完自己的任务。

Java语言的Thread类早期提供了一个stop()方法，用来终止线程，但是现在已经不建议使用了。既然不建议使用了，那在Java领域，如何优雅地终止线程呢？  
如何理解两阶段终止模式

两阶段终止模式。顾明思议，就是将终止过程分为两个阶段，其中第一个阶段主要是线程T1向线程T2发终止命令，而第二个阶段则是线程T2响应终止指令。

那在Java语言中，终止指令是什么呢。这个要从Java线程的状态转换说起。下面是Java线程的状态转换图。

这个图里你可以发现，Java线程进入终止状态的前提是线程进入RUNNABLE状态，而实际上线程也可能处在休眠状态，也就是说，我么想要终止一个线程，首先要把线程的状态从休眠状态转换到RUNNABLE状态。如何做到呢？这个要靠Java Thread 类提供的interrupt()方法，它可以将休眠状态的线程的线程转换到RUNNABLE状态。

线程转换到RUNNABLE状态以后，我们如何终止呢？RUNNABLE状态转换为终止状态，优雅的方式是让Java线程自己执行完run()方法，所以一般我们采用的方法是设置一个标志位，然后线程会在合适的时机检查这个标志位，如果发现符合终止条件，则自动退出run()方法。这个过程其实就是我们前面提到的第二阶段：响应终止条件。

综合上面的这两点，我们总结出终止指令，其实包括两方面的内容：interrupt()方法和线程终止的标志位。

理解了两阶段终止模式之后，下面我们看一个实际工作中的案例。

用两阶段终止模式终止监控操作

实际工作中，有些监控系统需要动态地采集一些数据，一般都是监控系统发送采集指令给被监控系统的监控代理，监控代理接受到指令之后，从监控目标收集数据，然后回传给监控系统，详细过程如下图所示。处于对性能的考虑（有些监控项对性能的影响很大，所以不能一直持续监控），动态采集功能一般都会有终止操作。

下面的示例代码是监控代理简化之后的实现，start()方法会启动一个新的线程rptThread来执行监控数据的采集和回传功能，stop()方法需要优雅地终止线程rptThread，那stop()相关的功能该如何实现呢？

### 38 案例分析一：高性能限流器GuavaRateLimiter

Guava RateLimiter是如何解决高并发场景下的限流问题的。Guava是Google开源的java类库，提供一个工具列RateLimiter。我们先来看看RateLimiter的使用。假设有一个线程池，他每秒只能处理两个任务，如果提交的任务过快，可能导致任务过快，可能导致系统不稳定，这个时候就需要用到限流。

在下面的代码中，我们创建了一个流速为2个请求/秒的限流器，这里的流速怎么理解呢？直观的看，2个请求/秒指的是每秒最多允许2个请求通过限流器，其实在Guava中，流速还有更深一层的意思：是一种匀速的概念，2个请求/秒等价于1个请求/500毫秒。

在向线程池提交任务之前，调用acquire()方法就能起到限流的作用。通过示例代码的执行结果，任务提交到线程池的时间间隔基本上稳定在500毫秒。

经典限流算法：令牌桶算法

Guavade限流器使用上还是很简单的，那他是如何实现的呢？Guava采用的是令牌桶算法，其核心是要想通过限流器，必须拿到令牌。也就是说，只要我们能够限制发放令牌的速率，那么就能控制流速了。令牌桶算法的详细描述如下：

1. 令牌以固定的速率添加到令牌桶中，假设限流的速率是r/秒,则令牌会没1/r秒添加一个；
2. 假设令牌桶的容量是b，如果令牌桶已满，则新令牌会被丢弃；
3. 请求能够通过限流器的前提是令牌桶中有令牌