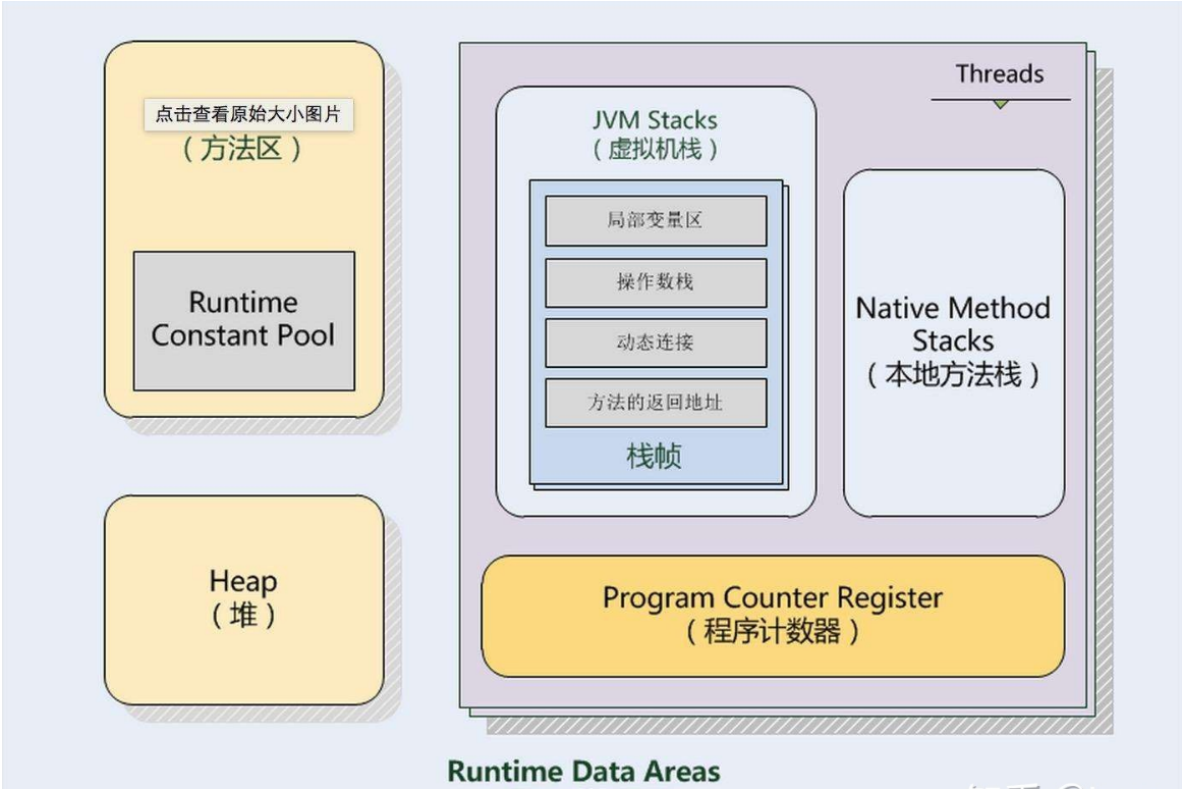
内存
    虚拟机内存分配
    JVM垃圾回收算法
    JVM垃圾收集器有哪些？以及优劣势比较？
关于并发编程
        线程之间的通信
        线程之间的同步
        Java内存模型和硬件架构之间的桥接
        线程安全性
        比较并交换 （CAS）
        解决并发冲突问题
        锁
            AQS
            CountDownLatch
            Semaphore
            CyclicBarrier
            ReentrantLock
                synchronized
                voliate
                volatile与synchronized
        进程、线程、协程
    并发模型
        1. 单进（线）程·循环处理请求
        2. 多进程
        4. 单线程·回调（callback）和事件轮询
            Nginx
            Node.js
        3. 多线程
        总结
        PS:函数式编程
线程
    线程的六种状态
    Java线程中断
callable & future
    Future模式(FutureTask)
ForkJoin
    一、简介
    二、工作窃取算法
    java线程池实现原理

内存

虚拟机内存分配

根据 JVM 规范，JVM 内存共分为虚拟机栈、堆、方法区、程序计数器、本地方法栈五个部分。



1.堆：存放对象实例，几乎所有的对象实例都在这里分配内存

- 堆得内存由-Xms指定，默认是物理内存的1/64；最大的内存由-Xmx指定，默认是物理内存的1/4。
- 默认空余的堆内存小于40%时，就会增大，直到-Xmx设置的内存。具体的比例可以由-XX:MinHeapFreeRatio指定
- 空余的内存大于70%时，就会减少内存，直到-Xms设置的大小。具体由-XX:MaxHeapFreeRatio指定。

2.虚拟机栈

虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息本地方法栈：本地方法栈则是为虚拟机使用到的Native方法服务。

3.方法区：存储已被虚拟机加载的类元数据信息(元空间)

1）有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载
2）方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。
该区域是被线程共享的。
3）方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

4.程序计数器：当前线程所执行的字节码的行号指示器

JVM垃圾回收算法

1.标记-清除: 这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：1.效率不高，标记和清除的效率都很低；2.会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次GC动作。

2.复制算法: 为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一般的内存。 于是将该算法进行了改进，内存区域不再是按照1：1去划分，而是将内存划分为8:1:1三部分，较大那份内存交Eden区，其余是两块较小的内存区叫Survior区。每次都会优先使用Eden区，若Eden区满，就将对象复制到第二块内存区上，然后清除Eden区，如果此时存活的对象太多，以至于Survivor不够时，会将这些对象通过分配担保机制复制到老年代中。(java堆又分为新生代和老年代)

\3. 标记-整理 该算法主要是为了解决标记-清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。

4.分代收集 现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用复制算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用标记-整理 或者 标记-清除。

JVM垃圾收集器有哪些？以及优劣势比较?

1.串行收集器

**
**串行收集器是最简单的，它设计为在单核的环境下工作（32位或者windows），你几乎不会使用到它。它在工作的时候会暂停整个应用的运行，因此在所有服务器环境下都不可能被使用。

使用方法：-XX:+UseSerialGC

2.并行收集器

这是JVM默认的收集器，跟它名字显示的一样，它最大的优点是使用多个线程来扫描和压缩堆。缺点是在minor和full GC的时候都会暂停应用的运行。并行收集器最适合用在可以容忍程序停滞的环境使用，它占用较低的CPU因而能提高应用的吞吐（throughput）。

使用方法：-XX:+UseParallelGC

3.CMS收集器

CMS是Concurrent-Mark-Sweep的缩写，并发的标记与清除。

这个算法使用多个线程并发地（concurrent）扫描堆，标记不使用的对象，然后清除它们回收内存。在两种情况下会使应用暂停（Stop the World, STW）：

- 1. 当初次开始标记根对象时initial mark。
- 2. 当在并行收集时应用又改变了堆的状态时，需要它从头再确认一次标记了正确的对象final remark。

这个收集器最大的问题是在年轻代与老年代收集时会出现的一种竞争情况（race condition），称为提升失败promotion failure。对象从年轻代复制到老年代称为提升promotion，但有时侯老年代需要清理出足够空间来放这些对象，这需要一定的时间，它收集的速度可能赶不上不断产生的要提升的年轻代对象的速度，这时就需要做STW的收集。STW正是CMS想避免的问题。为了避免这个问题，需要增加老年代的空间大小或者增加更多的线程来做老年代的收集以赶上从年轻代复制对象的速度。

除了上文所说的内容之外，CMS最大的问题就是内存空间碎片化的问题。CMS只有在触发FullGC的情况下才会对堆空间进行compact。如果线上应用长时间运行，碎片化会非常严重，会很容易造成promotion failed。为了解决这个问题线上很多应用通过定期重启或者手工触发FullGC来触发碎片整理。

对比并行收集器它的一个坏处是需要占用比较多的CPU。对于大多数长期运行的服务器应用来说，这通常是值得的，因为它不会导致应用长时间的停滞。但是它不是JVM的默认的收集器。

4.G1收集器

如果你的堆内存大于4G的话，那么G1会是要考虑使用的收集器。它是为了更好支持大于4G堆内存在JDK 7 u4引入的。G1收集器把堆分成多个区域，大小从1MB到32MB，并使用多个后台线程来扫描这些区域，优先会扫描最多垃圾的区域，这就是它名称的由来，垃圾优先Garbage First。

如果在后台线程完成扫描之前堆空间耗光的话，才会进行STW收集。它另外一个优点是它在处理的同时会整理压缩堆空间，相比CMS只会在完全STW收集的时候才会这么做。

使用过大的堆内存在过去几年是存在争议的，很多开发者从单个JVM分解成使用多个JVM的微服务（micro-service）和基于组件的架构。其他一些因素像分离程序组件、简化部署和避免重新加载类到内存的考虑也促进了这样的分离。

除了这些因素，最大的因素当然是避免在STW收集时JVM用户线程停滞时间过长，如果你使用了很大的堆内存的话就可能出现这种情况。另外，像Docker那样的容器技术让你可以在一台物理机器上轻松部署多个应用也加速了这种趋势。

使用方法： - XX:+UseG1GC

关于并发编程

在并发编程领域，有两个关键问题：线程之间的通信和同步。

线程之间的通信

线程的通信是指线程之间以何种机制来交换信息。在命令式编程中，线程之间的通信机制有两种共享内存和消息传递。

在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过共享对象进行通信。

在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在java中典型的消息传递方式就是wait()和notify()。

线程之间的同步

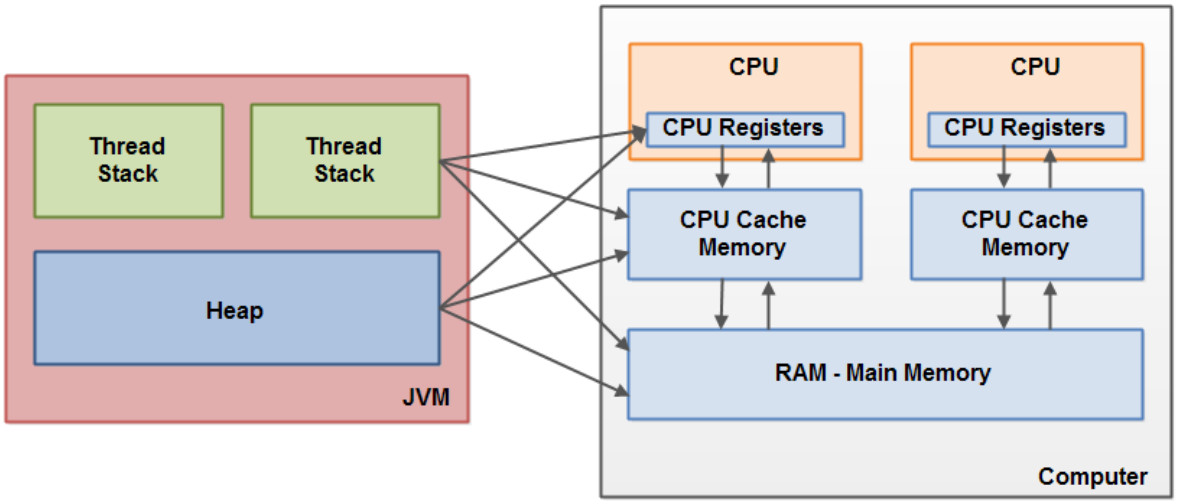
同步是指程序用于控制不同线程之间操作发生相对顺序的机制。

在共享内存并发模型里，同步是显式进行的。程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。

在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

Java内存模型和硬件架构之间的桥接

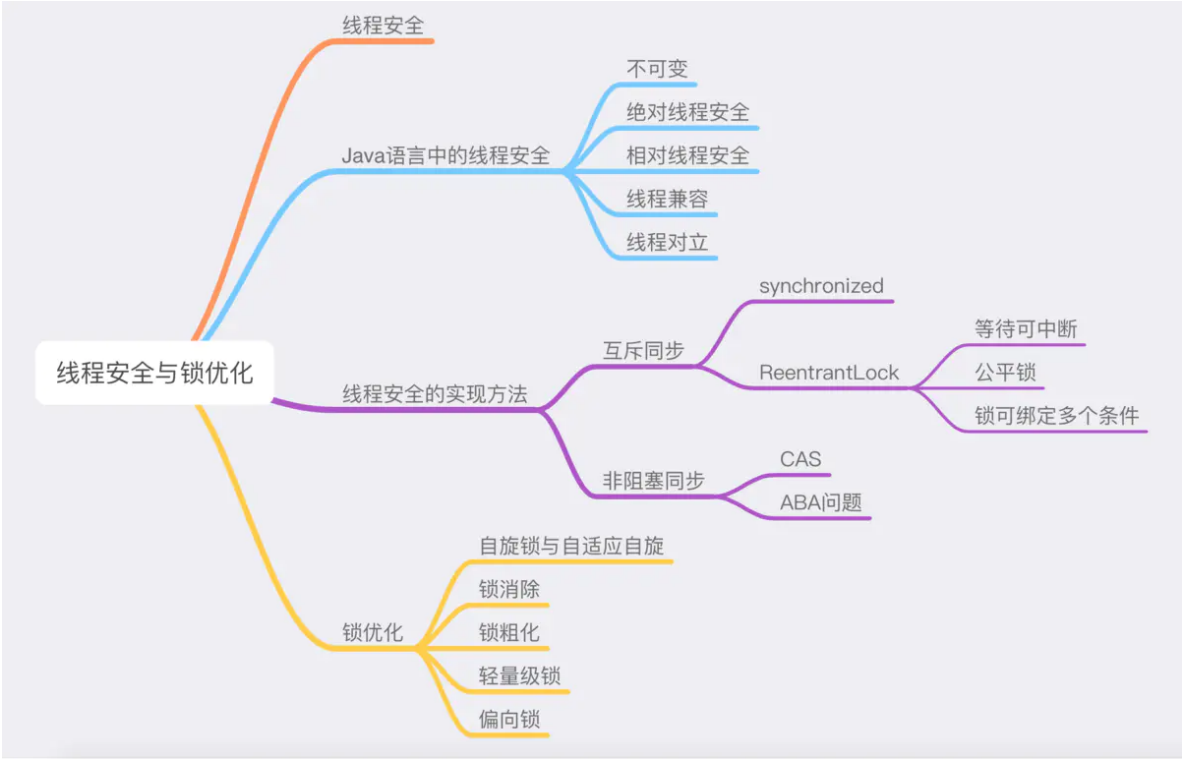
正如上面讲到的，Java内存模型和硬件内存架构并不一致。硬件内存架构中并没有区分栈和堆，从硬件上看，不管是栈还是堆，大部分数据都会存到主存中，当然一部分栈和堆的数据也有可能会存到CPU寄存器中，如下图所示，Java内存模型和计算机硬件内存架构是一个交叉关系：



当对象和变量存储到计算机的各个内存区域时，必然会面临一些问题，其中最主要的两个问题是：

- 1. 共享对象对各个线程的可见性
- 2. 共享对象的竞争现象

线程安全性



比较并交换 （CAS）

- CAS 指令需要有 3 个操作数，分别是内存位置（在 Java 中可以简单理解为变量的内存地址，用 V 表示）、旧的预期值（用 A 表示）和新值（用 B 表示）。CAS 指令执行时，当且仅当 V 符合旧值预期值 A 时，处理器用新值 B 更新 V 的值，否则它就不执行更新，但是无论是否更新了 V 的值，都会返回 V 的旧值，上述的处理过程是一个原子操作。

  在 JDK 1.5 之后，Java 程序中才可以使用 CAS 操作，该操作由 sun.misc.Unsafe 类里面的 compareAndSwapInt() 和 compareAndSwapLong() 等几个方法包装提供，虚拟机在内部对这些方法做了特殊处理，即时编译出来的结果就是一条平台相关的处理器 CAS 指令，没有方法调用的过程，或者可以认为是无条件内联进去了。（注：这种被虚拟机特殊处理的方法称为固有函数（intrinsics），类似的固有函数还有 Math.sin() 等）

  由于 Unsafe 类不是提供给用户程序调用的类（Unsafe.getUnsafe() 的代码中限制了只有启动类加载器（Bootstrap ClassLoader）加载的 Class 才能访问它），因此，如果不采用反射手段，我们只能通过其他的 Java API 来间接使用它，如 J.U.C 包里面的整数原子类，其中的 compareAndSet() 和 getAndIncrement() 等方法都使用了 Unsafe 类的 CAS 操作。

- ABA 问题
  - incrementAndGet()方法在一个无限循环中，不断尝试将一个比当前值大1的新值赋给自己。
  - 如果失败了，那说明在执行“获取-设置”操作的时候值已经有了修改，于是再次循环进行下一次操作，直到设置成功为止。
  - 尽管CAS看起来很美，但显然这种操作无法涵盖互斥同步的所有使用场景，并且CAS从语义上来说并不是完美的，
  - 存在这样的一个逻辑漏洞：如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然为A值，
  - 那我们就能说它没有被其他线程改变过了吗？如果在这段期间它的值曾经被改成了B，后来又被改回A，
  - 那CAS操作就会误认为它从来没有被改变过。这个漏洞称为CAS操作的“ABA”问题。J.U.C包为了解决这个问题，

    - 提供了一个带有标记的原子引用类“AtomicStampedReference”，它可以通过控制变量值的版本来保证CAS的正确性。
    - 不过目前来说这个类比较“鸡肋”，大部分情况下ABA问题不会影响程序并发的正确性，
    - 如果需要解决ABA问题，改用传统的互斥同步可能会比原子类更高效。

- AQS的基础也是cas

解决并发冲突问题

- 原子类
    - CAS
- 不可变对象

Collections.unmodifiableXXX

JDK中提供了一系列方法方便我们创建不可变集合

- java.util.Collections#unmodifiableCollection
- java.util.Collections#unmodifiableSet
- java.util.Collections#unmodifiableSortedSet
- java.util.Collections#unmodifiableNavigableSet
- java.util.Collections#unmodifiableList
- java.util.Collections#unmodifiableMap
- java.util.Collections#unmodifiableSortedMap
- java.util.Collections#unmodifiableNavigableMap

Guava

可变集合类型	可变集合源：JDK or Guava?	Guava不可变集合
Collection	JDK	ImmutableCollection

可变集合类型	可变集合源：JDK or Guava?	Guava不可变集合
List	JDK	ImmutableList
Set	JDK	ImmutableSet
SortedSet/NavigableSet	JDK	ImmutableSortedSet
Map	JDK	ImmutableMap
SortedMap	JDK	ImmutableSortedMap
Multiset	Guava	ImmutableMultiset
SortedMultiset	Guava	ImmutableSortedMultiset
Multimap	Guava	ImmutableMultimap
ListMultimap	Guava	ImmutableListMultimap
SetMultimap	Guava	ImmutableSetMultimap
BiMap	Guava	ImmutableBiMap
ClassToInstanceMap	Guava	ImmutableClassToInstanceMap
Table	Guava	ImmutableTable

- **ArrayList为什么会出现并发问题?**

  ArrayList是线程不安全的，在多线程并发访问的时候可能会出现问题，如果想使用线程安全的集合类，java自带有vector,也就是说vector是线程安全的。但是arayList的底层是数组实现的，而且可以自动扩容，获得元素或者在数组尾段插入元素的效率高，所以说ArrayList有其独特的优势。

  ArrayList并发可能出现的问题

  - 结果不正确
  - null
  - ArrayIndexOutOfBoundsException
- **HashMap为什么会出现并发问题?**

  大多数javaer都知道HashMap是线程不安全的，多线程环境下数据可能会发生错乱，一定要谨慎使用。这个结论是没错，可是HashMap的线程不安全远远不是数据脏读这么简单，它还有可能会发生死锁，造成内存飙升100%的问题（JDK1.8修复了这个问题）

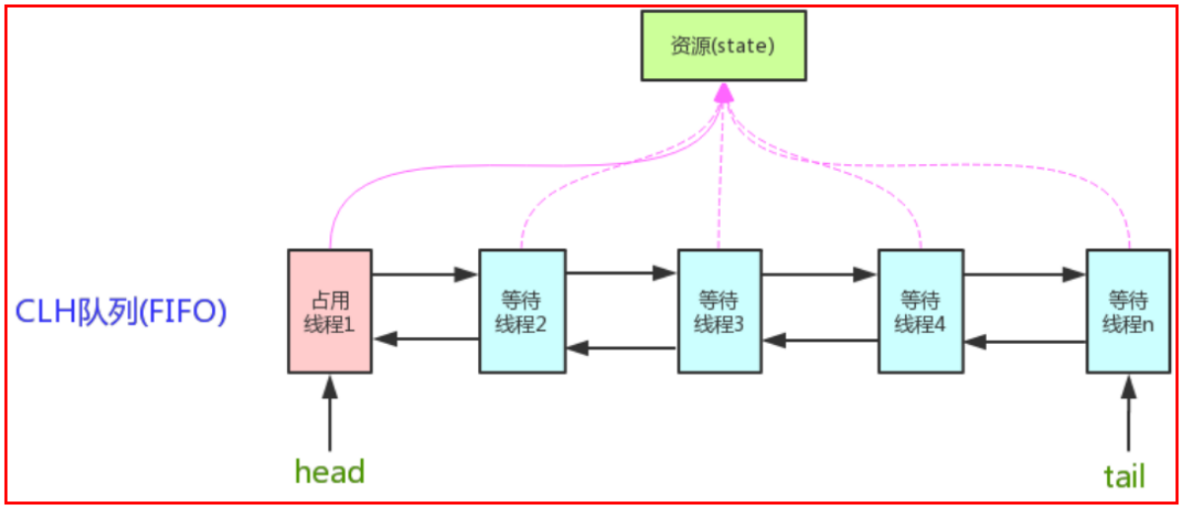
  - 脏数据
  - 死锁，造成内存飙升100%的问题
- **解决方案**

  - java.util.Collections#synchronizedCollection(java.util.Collection)
  - java.util.Collections#synchronizedSet(java.util.Set)
  - java.util.Collections#synchronizedSortedSet
  - java.util.Collections#synchronizedNavigableSet
  - java.util.Collections#synchronizedList(java.util.List)
  - java.util.Collections#synchronizedMap

- **并发容器（ConcurrentHashMap，CopyOnWriteArrayList，CopyOnWriteArraySet）**

锁

AQS



同步器的核心方法是acquire和release操作，其背后的思想也比较简洁明确。acquire操作是这样的：

while (**当前同步器的状态**不允许获取操作) {

```
如果当前**线程**不在**队列**中，则将其插入队列

**阻塞**当前线程
```

}

如果线程位于队列中，则将其移出队列

release操作是这样的：

更新**同步器的状态**

if (新的状态允许某个被阻塞的线程获取成功)

```
**解除队列**中一个或多个**线程**的**阻塞**状态
```

从这两个操作中的思想中我们可以提取出三大关键操作：**同步器的状态变更、线程阻塞和释放、插入和移出队列。**所以为了实现这两个操作，需要协调三大关键操作引申出来的三个基本组件：

·同步器状态的原子性管理；

·线程阻塞与解除阻塞；

·队列的管理；

由这三个基本组件，我们来看j.u.c是怎么设计的。

CountDownLatch

CountDownLatch是并发包中用来控制一个或者多个线程等待其他线程完成操作的并发工具类

Semaphore

Semaphore用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量。还可以用来实现某种资源池限制，或者对容器施加边界。

CyclicBarrier

栅栏类似于闭锁，它能阻塞一组线程直到某个事件的发生。栅栏与闭锁的关键区别在于，所有的线程必须同时到达栅栏位置，才能继续执行。闭锁用于等待事件，而栅栏用于等待其他线程。

CyclicBarrier可以使一定数量的线程反复地在栅栏位置处汇集。当线程到达栅栏位置时将调用await方法，这个方法将阻塞直到所有线程都到达栅栏位置。如果所有线程都到达栅栏位置，那么栅栏将打开，此时所有的线程都将被释放，而栅栏将被重置以便下次使用。

ReentrantLock

synchronized

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性

voliate

当一个变量定义为volatile后，它将具备两种特性：1. 可见性，2. 禁止指令重排序。

首先，我们要意识到有这样的现象：编译器为了加快程序运行速度，对一些变量的写操作会现在寄存器或CPU缓存上进行，最后写入内存。而在这个过程中，变量的新值对其它线程是不可见的。

可见性：当对volatile标记的变量进行修改时，会将其它缓存中存储的修改前的变量清除，然后重新读取。这里从哪读尚未明确，一般来说应该是先在进行修改的缓存A中修改为新值，然后通知其它缓存清除掉此变量，当其它缓存B中的线程读取此变量时，会向总线发送消息，这是存储新值的缓存A获取到消息，将新值传给B，最后将新值写入内存。

volatile与synchronized

volatile本质是在告诉jvm当前变量在寄存器中的值是不确定的,需要从主存中读取,synchronized则是锁定当前变量,只有当前线程可以访问该变量,其他线程被阻塞住. volatile仅能使用在变量级别,synchronized则可以使用在变量,方法. volatile仅能实现变量的修改可见性,但不具备原子特性,而synchronized则可以保证变量的修改可见性和原子性. volatile不会造成线程的阻塞,而synchronized可能会造成线程的阻塞. volatile标记的变量不会被编译器优化,而synchronized标记的变量可以被编译器优化.

进程、线程、协程

进程是系统进行资源分配的一个独立单位。这些资源包括：用户的地址空间，实现进程（线程）间同步和通信的机制，已打开的文件和已申请到的I/O设备，以及一张由核心进程维护的地址映射表。内核通过**进程控制块**（PCB，process control block）来感知进程。

线程是调度和分派的基本单位。内核通过**线程控制块**（TCB，thread control block）来感知线程。

线程本身不拥有系统资源，而是仅有一点必不可少的、能保证独立运行的资源，如TCB、程序计数器、局部变量、状态参数、返回地址等寄存器和堆栈。同一进程的所有线程具有相同的地址空间，线程可以访问进程拥有的资源。多个线程可并发执行，一个进程含有若干个相对独立的线程，但至少有一个线程。

线程的有不同的实现方式，分**内核支持线程**（KST，Kernel Supported Threads）和**用户级线程**(UST, User Supported Threads)。内核级线程的 TCB 保存在内核空间，其创建、阻塞、撤销、切换等活动也都是在内核空间实现的。用户级线程则是内核无关的，用户级线程的实现在用户空间，内核感知不到用户线程的存在。用户线程的调度算法可以是进程专用的，不会被内核调度，但同时，用户线程也无法利用多处理机的并行执行。而一个拥有多个用户线程的进程，一旦有一个线程阻塞，该进程所有的线程都会被阻塞。内核的切换需要转换到内核空间，而用户线程不需要，所以前者开销会更大。但用户线程也需要内核的支持，一般是通过运行时系统或内核控制线程来连接一个内核线程，有 1:1、1:n、n:m 的不同实现。

在分时操作系统中，处理机的调度一般基于时间片的轮转（RR, round robin），多个就绪线程排成队列，轮流执行时间片。而为保证交互性和实时性，线程都是以抢占的方式（Preemptive Mode）来获得处理机。而抢占方式的开销是比较大的。有抢占方式就有非抢占方式（Nonpreemptiv Mode），在非抢占式中，除非某正在运行的线程执行完毕、因系统调用（如 I/O 请求）发生阻塞或主动让出处理器，不会被调度或暂停。

而**协程**（Coroutine）就是基于非抢占式的调度来实现的。进程、线程是操作系统级别的概念，而协程是编译器级别的，现在很多编程语言都支持协程，如 Erlang、Lua等。准确来说，协程只是一种用户态的轻量线程。它运行在用户空间，不受系统调度。它有自己的调度算法。在上下文切换的时候，协程在用户空间切换，而不是陷入内核做线程的切换，减少了开销。简单地理解，就是编译器提供一套自己的运行时系统（而非内核）来做调度，做上下文的保存和恢复，重新实现了一套"并发"机制。系统的并发是时间片的轮转，单处理器交互执行不同的执行流，营造不同线程同时执行的感觉；而协程的并发，是单线程内控制权的轮转。相比抢占式调度，协程是主动让权，实现协作。协程的优势在于，相比回调的方式，写的异步代码可读性更强。缺点在于，因为是用户级线程，利用不了多核机器的并发执行。

线程的出现，是为了分离进程的两个功能：资源分配和系统调度。让更细粒度、更轻量的线程来承担调度，减轻调度带来的开销。但线程还是不够轻量，因为调度是在内核空间进行的，每次线程切换都需要陷入内核，这个开销还是不可忽视的。协程则是把调度逻辑在用户空间里实现，通过自己（编译器运行时系统/程序员）模拟控制权的交接，来达到更加细粒度的控制。

并发模型

1. 单进（线）程·循环处理请求

单进程和单线程其实没有区别，因为一个进程至少有一个线程。循环处理请求应该是最初级的做法。当大量请求进来时，单线程一个一个处理请求，请求很容易就积压起来，得不到响应。这是无并发的做法。

---

2. 多进程

主进程监听和管理连接，当有客户请求的时候，fork 一个子进程来处理连接，父进程继续等待其他客户的请求。但是进程占用服务器资源是比较多的，服务器负载会很高。

4. 单线程·回调（callback）和事件轮询

Nginx

Nginx 采用的是多进程（单线程） & 多路IO复用模型:

Node.js

Node.js 也是单线程模型。

3. 多线程

和多进程的方式类似，只不过是替换成线程。主线程负责监听、`accept()`连接，子线程（工作线程）负责处理业务逻辑和流的读取。子线程阻塞，同一进程内的其他线程不会被阻塞。

缺点是：

1. 会频繁地创建、销毁线程，这对系统也是个不小的开销。这个问题可以用线程池来解决。线程池是预先创建一部分线程，由线程池管理器来负责调度线程，达到线程复用的效果，避免了反复创建线程带来的性能开销，节省了系统的资源。
2. 要处理同步的问题，当多个线程请求同一个资源时，需要用锁之类的手段来保证线程安全。同步处理不好会影响数据的安全性，也会拉低性能。
3. 一个线程的崩溃会导致整个进程的崩溃。

多线程的适用场景是：提高响应速度，让IO和计算相互重叠，降低延时。虽然多线程不能提高绝对性能，但是可以提高平均响应性能。

这种其实是比较容易想到的，特别是对于刚刚学习多线程和操作系统的计算机学生而言。在请求量不高的时候，是足够的。来多少连接开多少线程，就看服务器的硬件性能能不能承受。但高并发并不是线性地堆砌硬件或加线程数就能达到的。100个线程也许能够达到1000的并发，但10000的并发下，线程数乘以10也许就不行，比如线程调度带来的开销、同步成为了瓶颈。

总结

高并发的关键在于实现异步非阻塞，更加高效地利用 CPU。多线程可以达到非阻塞，但占用资源多，切换开销大。协程用栈的动态增长、用户态的调度来避免多线程的两个问题。事件驱动用单线程的方式，避免了占用太多系统资源，不需要关心线程安全，但无法利用多核。具体要采用哪种模型，还是要看需求。模型或技术只是工具，条条大陆通罗马。

比较优雅的还是 CSP 和 Actor 模型，因为能够符合人的思维习惯，避免了锁的使用。个人觉得加锁和多线程的方式，很容易被滥用，这是一种从微观出发和线性的思维方式，不够高屋建瓴。不如用消息通信来的耦合性更低。

高并发编程很有必要性。一方面，很多应用都需要高并发支持，网络的用户越来越多，业务场景会越来越复杂，需要有稳定和高效的服务器支持。另一方面，现代的计算机性能都是比较高的，但如果软件设计得不够好，就不能够把性能都给发挥出来。这就很浪费了。

PS:函数式编程

函数式编程也是一个可以用来解决并发问题的模型。

线程

在`Java`中，创建线程一般有两种方式，一种是继承`Thread`类，一种是实现`Runnable`接口。

线程的六种状态

线程状态枚举：

java.lang.Thread.State

- NEW： 新建状态，线程对象已经创建，但尚未启动
- RUNNABLE:就绪状态，可运行状态，调用了线程的start方法，已经在java虚拟机中执行，等待获取操作系统资源如CPU，操作系统调度运行。
- BLOCKED:堵塞状态。线程等待锁的状态，等待获取锁进入同步块/方法或调用wait后重新进入需要竞争锁
- WAITING：等待状态。等待另一个线程以执行特定的操作。调用以下方法进入等待状态。Object.wait(), Thread.join(),LockSupport.park
- TIMED_WAITING: 线程等待一段时间。调用带参数的Thread.sleep, objct.wait,Thread.join，LockSupport.parkNanos,LockSupport.parkUntil
- TERMINATED:进程结束状态。

其中，Thread.sleep(long)使线程暂停一段时间，进入TIMED_WAITING时间，并不会释放锁，在设定时间到或被interrupt后抛出InterruptedException后进入RUNNABLE状态； Thread.join是等待调用join方法的线程执行一段时间(join(long))或结束后再往后执行，被interrupt后也会抛出异常，join内部也是wait方式实现的。

wait方法是object的方法，线程释放锁，进入WAITING或TIMED_WAITING状态。等待时间到了或被notify/notifyall唤醒后，回去竞争锁，如果获得锁，进入RUNNABLE，否则进步BLOCKED状态等待获取锁。

Java线程中断

interrupted()是Java提供的一种中断机制，要把中断搞清楚，还是得先系统性了解下什么是中断机制。

callable & future

Future模式(FutureTask)

Future模式的核心在于：去除了主函数的等待时间，并使得原本需要等待的时间段可以用于处理其他业务逻辑。

`Futrure`模式:对于多线程，如果线程A要等待线程`B`的结果，那么线程`A`没必要等待`B`，直到`B`有结果，可以先拿到一个未来的`Future`，等`B`有结果是再取真实的结果。

ForkJoin

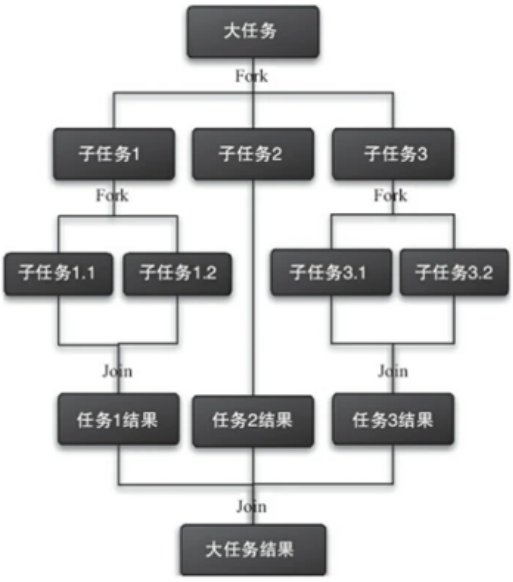
一、简介

算法领域有一种基本思想叫做“**分治**”，所谓“分治”就是将一个难以直接解决的大问题，分割成一些规模较小的子问题，以便各个击破，分而治之。

比如：对于一个规模为N的问题，若该问题可以容易地解决，则直接解决；否则将其分解为K个规模较小的子问题，这些子问题互相独立且与原问题性质相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解，这种算法设计策略叫做分治法。

许多基础算法都运用了“分治”的思想，比如二分查找、快速排序等等。

基于“分治”的思想，J.U.C在JDK1.7时引入了一套**Fork/Join**框架。Fork/Join框架的基本思想就是将一个大任务分解（**Fork**）成一系列子任务，子任务可以继续往下分解，当多个不同的子任务都执行完成后，可以将它们各自的结果合并（**Join**）成一个大结果，最终合并成大任务的结果：



二、工作窃取算法

从上述Fork/Join框架的描述可以看出，我们需要一些线程来执行Fork出的任务，在实际中，如果每次都创建新的线程执行任务，对系统资源的开销会很大，所以Fork/Join框架利用了线程池来调度任务。

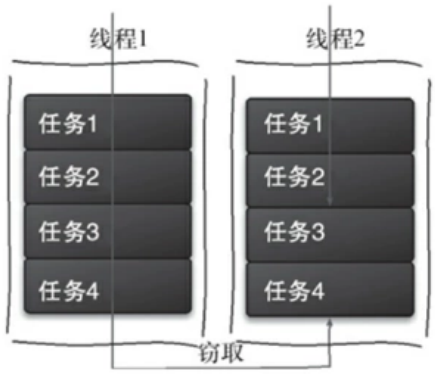
另外，这里可以思考一个问题，既然由线程池调度，必然存在两个要素：

- 工作线程
- 任务队列

一般的线程池只有一个任务队列，但是对于Fork/Join框架来说，由于Fork出的各个子任务其实是平行关系，为了提高效率，减少线程竞争，应该将这些平行的任务放到不同的队列中去，如上图中，大任务分解成三个子任务：子任务1、子任务2、子任务3，那么就创建三个任务队列，然后再创建3个工作线程与队列一一对应。

由于线程处理不同任务的速度不同，这样就可能存在某个线程先执行完了自己队列中的任务的情况，这时为了提升效率，我们可以让该线程去“窃取”其它任务队列中的任务，这就是所谓的***工作窃取算法***。

“工作窃取”的示意图如下，当线程1执行完自身任务队列中的任务后，尝试从线程2的任务队列中“窃取”任务：



对于一般的队列来说，入队元素都是在“队尾”，出队元素在“队首”，要满足“工作窃取”的需求，任务队列应该支持从“队尾”出队元素，这样可以减少与其它工作线程的冲突（因为正常情况下，其它工作线程从“队首”获取自己任务队列中的任务），满足这一需求的任务队列其实就是我们在juc-collections框架中LinkedBlockingDeque

当然，出于性能考虑，J.U.C中的Fork/Join框架并没有直接利用LinkedBlockingDeque作为任务队列，而是自己重新实现了一个。

java线程池实现原理

线程池由两个核心数据结构组成：

1）线程集合（workers）：存放执行任务的线程，是一个HashSet；

2）任务等待队列（workQueue）：存放等待线程池调度执行的任务，是一个阻塞式队列BlockingQueue；

拒绝策略（RejectedExecutionHandler）

当线程集合和等待队列都满时线程无法调度任务，这时线程池会执行一个默认的或使用者指定的拒绝策略。

JDK内置的拒绝策略主要有下面几种：

1）调用线程执行（CallerRunsPolicy），任务被线程池拒绝后，任务会被调用线程执行；

2）终止执行（AbortPolicy），任务被拒绝时，抛出RejectedExecutionException异常报错

3）丢弃任务（DiscardPolicy），任务被直接丢弃，不会抛异常报错；

4）丢失老任务（DiscardOldestPolicy），把等待队列中最老的任务删除，删除后重新提交当前任务。

除了这些内置的拒绝策略，使用者还可以实现RejectedExecutionHandler接口自定义拒绝策略

关闭线程池

关闭线程池时有两个关键步骤：

1）修改线程池状态到SHUTDOWN，这时新提交到线程池的任务都会被直接拒绝；

2）中断线程池中的所有线程，中断任务执行回收线程集合中所有线程。