

Java后台面试 常见问题



时芥蓝 (/u/bd4811478d4b) [+ 关注](#)

2017.04.15 09:57* 字数 43884 阅读 6205 评论 7 喜欢 198 赞赏 1

(/u/bd4811478d4b)

从三月份找实习到现在，面了一些公司，挂了不少，但最终还是拿到小米、百度、阿里、京东、新浪、CVTE、乐视家的研发岗offer。我找的是java后台开发，把常见的问题分享给大家，有一些是自己的总结，有一些是网上借鉴的内容。希望能帮助到各位。预祝各位同学拿到自己心仪的offer！

Nginx负载均衡

- 轮询、轮询是默认的，每一个请求按顺序逐一分配到不同的后端服务器，如果后端服务器down掉了，则能自动剔除
- ip_hash、个请求按访问IP的hash结果分配，这样来自同一个IP的访客固定访问一个后端服务器，有效解决了动态网页存在的session共享问题。
- weight、weight是设置权重，用于后端服务器性能不均的情况，访问比率约等于权重之比
- fair(第三方)、这是比上面两个更加智能的负载均衡算法。此种算法可以依据页面大小和加载时间长短智能地进行负载均衡，也就是根据后端服务器的响应时间来分配请求，响应时间短的优先分配。Nginx本身是不支持fair的，如果需要使用这种调度算法，必须下载Nginx的upstream_fair模块。
- url_hash(第三方)此方法按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，可以进一步提高后端缓存服务器的效率。Nginx本身是不支持url_hash的，如果需要使用这种调度算法，必须安装Nginx 的hash软件包。

代理的概念

正向代理，也就是传说中的代理，简单的说，我是一个用户，我访问不了某网站，但是我能访问一个代理服务器，这个代理服务器呢，他能访问那个我不能访问的网站，于是我先连上代理服务器，告诉他我需要那个无法访问网站的内容，代理服务器去取回来，然后返回给我。从网站的角度，只在代理服务器来取内容的时候有一次记录，有时候并不知道是用户的请求，也隐藏了用户的资料，这取决于代理告不告诉网站。

反向代理：结论就是，反向代理正好相反，对于客户端而言它就像是原始服务器，并且客户端不需要进行任何特别的设置。客户端向反向代理的命名空间(name-space)中的内容发送普通请求，接着反向代理将判断向何处(原始服务器)转交请求，并将获得的内容返回给客户端，就像这些内容原本就是它自己的一样。

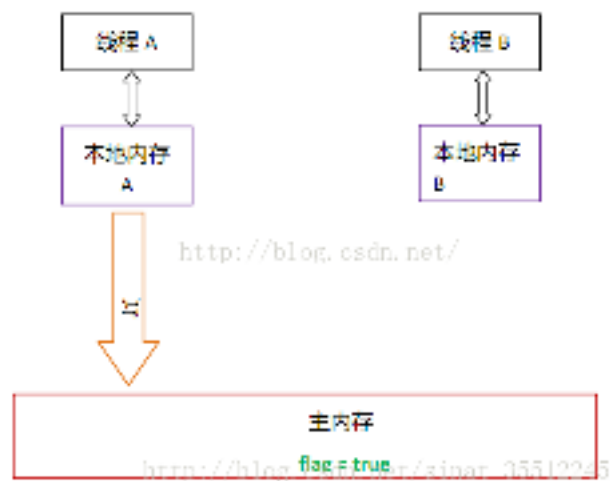
Volatile的特征：

- A、原子性：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作不具有原子性。
- B、可见性：对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。



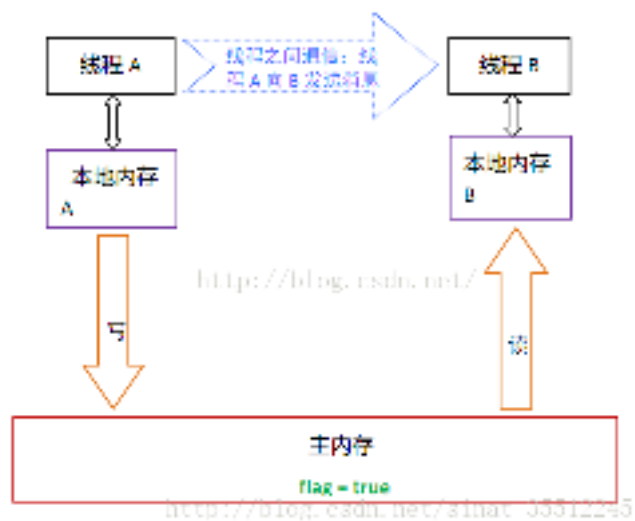
volatile的内存语义：

当写一个volatile变量时，JMM会把线程对应的本地内存中的共享变量值刷新到主内存。



这里写图片描述

当读一个volatile变量时，JMM会把线程对应的本地内存置为无效，线程接下来将从主内存中读取共享变量。



这里写图片描述

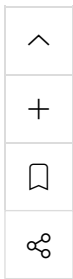
volatile的重排序

- 1、当第二个操作为volatile写操作时,不管第一个操作是什么(普通读写或者volatile读写),都不能进行重排序。这个规则确保volatile写之前的所有操作都不会被重排序到volatile之后;
- 2、当第一个操作为volatile读操作时,不管第二个操作是什么,都不能进行重排序。这个规则确保volatile读之后的所有操作都不会被重排序到volatile之前;
- 3、当第一个操作是volatile写操作时,第二个操作是volatile读操作,不能进行重排序。

这个规则和前面两个规则一起构成了:两个volatile变量操作不能够进行重排序;

除以上三种情况以外可以进行重排序。

比如：



- 1、第一个操作是普通变量读/写,第二个是volatile变量的读；
- 2、第一个操作是volatile变量的写,第二个是普通变量的读/写；

内存屏障/内存栅栏

内存屏障（Memory Barrier，有时叫做内存栅栏，Memory Fence）是一种CPU指令，用于控制特定条件下的重排序和内存可见性问题。Java (https://link.jianshu.com?t=http://lib.csdn.net/base/javase)编译器也会根据内存屏障的规则禁止重排序。（也就是让一个CPU处理单元中的内存状态对其它处理单元可见的一项技术。）

内存屏障可以被分为以下几种类型：

LoadLoad屏障：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。

StoreStore屏障：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。

LoadStore屏障：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。

StoreLoad屏障：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。

在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

内存屏障阻碍了CPU采用优化技术来降低内存操作延迟，必须考虑因此带来的性能损失。为了达到最佳性能，最好是把要解决的问题模块化，这样处理器可以按单元执行任务，然后在任务单元的边界放上所有需要的内存屏障。采用这个方法可以让处理器不受限的执行一个任务单元。合理的内存屏障组合还有一个好处是：缓冲区在第一次被刷后开销会减少，因为再填充改缓冲区不需要额外工作了。

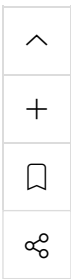
happens-before原则

如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系。

程序顺序规则：一个线程内，按源代码的顺序，46与在后面的操作先行发生于44与在后面的操作；
lock规则：一个unlock操作先行发生于任意到同一个lock操作；
volatile变量规则：对一个变量的写操作先行发生于任意读对这个变量的读操作；
代理规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C；
线程启动规则：Thread的start()方法先行发生于任意到该线程的每一个操作；
线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生；
线程终止规则：线程中所有操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法检测，Thread.isAlive()的返回值来判断该线程已经终止执行；
对象终结规则：一个对象的构造先行发生于它的finalize()方法的开始；

这里写图片描述

Java如何实现跨平台的？



跨平台是怎样实现的呢？这就要谈及Java虚拟机（Java (<https://link.jianshu.com?t=http://lib.csdn.net/base/java>)Virtual Machine，简称 JVM）。

JVM也是一个软件，不同的平台有不同的版本。我们编写的Java源码，编译后会生成一种 .class 文件，称为字节码文件。Java虚拟机就是负责将字节码文件翻译成特定平台下的机器码然后运行。也就是说，只要在不同平台上安装对应的 JVM，就可以运行字节码文件，运行我们编写的Java程序。

而在这个过程中，我们编写的Java程序没有做任何改变，仅仅是通过JVM这一“中间层”，就能在不同平台上运行，真正实现了“一次编译，到处运行”的目的。

JVM是一个“桥梁”，是一个“中间件”，是实现跨平台的关键，Java代码首先被编译成字节码文件，再由JVM将字节码文件翻译成机器语言，从而达到运行Java程序的目的。

注意：编译的结果不是生成机器码，而是生成字节码，字节码不能直接运行，必须通过JVM翻译成机器码才能运行。不同平台下编译生成的字节码是一样的，但是由JVM翻译成的机器码却不一样。

所以，运行Java程序必须有JVM的支持，因为编译的结果不是机器码，必须要经过JVM的再次翻译才能执行。即使你将Java程序打包成可执行文件（例如 .exe），仍然需要JVM的支持。

注意：跨平台的是Java程序，不是JVM。JVM是用C/C++开发的，是编译后的机器码，不能跨平台，不同平台下需要安装不同版本的JVM。

垃圾搜集器

1. 按照线程数量来分：

1. 串行 串行垃圾回收器一次只使用一个线程进行垃圾回收
2. 并行 并行垃圾回收器一次将开启多个线程同时进行垃圾回收。

2. 按照工作模式来分：

1. 并发 并发式垃圾回收器与应用程序线程交替工作，以尽可能减少应用程序的停顿时间
2. 独占 一旦运行，就停止应用程序中的其他所有线程，直到垃圾回收过程完全结束

3. 按照碎片处理方式：

1. 压缩式 压缩式垃圾回收器会在回收完成后，对存活对象进行压缩整消除回收后的碎片；
2. 非压缩式 非压缩式的垃圾回收器不进行这步操作。

4. 按工作的内存区间 可分为新生代垃圾回收器和老年代垃圾回收器

- 新生代串行收集器 serial 它仅仅使用单线程进行垃圾回收；第二，它独占式的垃圾回收。使用复制算法。
- 老年代串行收集器 serial old 年代串行收集器使用的是标记-压缩算法。和新生代串行收集器一样，它也是一个串行的、独占式的垃圾回收器
- 并行收集器 parnew 并行收集器是工作在新生代的垃圾收集器，它只简单地将串行回收器多线程化。它的回收策略、算法以及参数和串行回收器一样 并行回收器也是独占式的回收器，在收集过程中，应用程序会全部暂停。但由于并行回收器使用多线程进



行垃圾回收，因此，在并发能力比较强的 CPU 上，它产生的停顿时间要短于串行回收器，而在单 CPU 或者并发能力较弱的系统中，并行回收器的效果不会比串行回收器好，由于多线程的压力，它的实际表现很可能比串行回收器差。

- 新生代并行回收 (Parallel Scavenge) 收集器 新生代并行回收收集器也是使用复制算法的收集器。从表面上看，它和并行收集器一样都是多线程、独占式的收集器。但是，并行回收收集器有一个重要的特点：它非常关注系统的吞吐量。
- 老年代并行回收收集器 parallel old 老年代的并行回收收集器也是一种多线程并发的收集器。和新生代并行回收收集器一样，它也是一种关注吞吐量的收集器。老年代并行回收收集器使用标记-压缩算法，JDK1.6 之后开始启用。
- CMS 收集器 CMS 收集器主要关注于系统停顿时间。CMS 是 Concurrent Mark Sweep 的缩写，意为并发标记清除，从名称上可以得知，它使用的是标记-清除算法，同时它又是一个使用多线程并发回收的垃圾收集器。
 - CMS 工作时，主要步骤有：初始标记、并发标记、重新标记、并发清除和并发重置。其中初始标记和重新标记是独占系统资源的，而并发标记、并发清除和并发重置是可以和用户线程一起执行的。因此，从整体上来说，CMS 收集不是独占式的，它可以在应用程序运行过程中进行垃圾回收。

根据标记-清除算法，初始标记、并发标记和重新标记都是为了标记出需要回收的对象。并发清理则是在标记完成后，正式回收垃圾对象；并发重置是指在垃圾回收完成后，重新初始化 CMS 数据结构和数据，为下一次垃圾回收做好准备。并发标记、并发清理和并发重置都是可以和应用程序线程一起执行的。

- G1 收集器 G1 收集器是基于标记-压缩算法的。因此，它不会产生空间碎片，也没有必要在收集完成后，进行一次独占式的碎片整理工作。G1 收集器还可以进行非常精确的停顿控制。

网络基本概念

OSI模型

OSI 模型(Open System Interconnection model)是一个由国际标准化组织提出的概念模型,试图提供一个使各种不同的计算机和网络在世界范围内实现互联的标准框架。它将计算机网络体系结构划分为七层,每层都可以提供抽象良好的接口。了解 OSI 模型有助于理解实际上互联网络的工业标准——TCP/IP 协议。OSI 模型各层间关系和通讯时的数据流向如图所示：

OSI 模型.png

显然、如果一个东西想包罗万象、一般时不可能的；在实际的开发应用中一般时在此模型的基础上进行裁剪、整合！

七层模型介绍

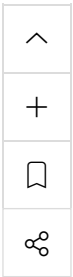
- 物理层：

物理层负责最后将信息编码成电流脉冲或其它信号用于网上传输；

eg：RJ45等将数据转化成0和1；
- 数据链路层：

数据链路层通过物理网络链路提供数据传输。不同的数据链路层定义了不同的网络协议特征,其中包括物理编址、网络拓扑结构、错误校验、数据帧序列以及流控；

可以简单的理解为：规定了0和1的分包形式，确定了网络数据包的形式；



- 网络层
网络层负责在源和终点之间建立连接;
可以理解为, 此处需要确定计算机的位置, 怎么确定? IPv4, IPv6!
- 传输层
传输层向高层口提供可靠的端到端的网络数据流服务。
可以理解为: 每一个应用程序都会在网卡注册一个端口号, 该层就是端口与端口的通信! 常用的 (TCP / IP) 协议;
- 会话层
会话层建立、管理和终止表示层与实体之间的通信会话;
建立一个连接 (自动的手机信息、自动的网络寻址) ;
- 表示层:
表示层口供多种功能用于应用层数据编码和转化,以确保以一个系统应用层发送的信息可以被另一个系统应用层识别;
可以理解为: 解决不同系统之间的通信, eg: Linux下的QQ和Windows下的QQ可以通信;
- 应用层:
OSI 的应用层协议包括文件的传输、访问及管理协议(FTAM) ,以及文件虚拟终端协议(VIP)和公用管理系统信息(CMIP)等;
规定数据的传输协议;

常见的应用层协议:

常见的应用层协议.png

互联网分层结构的好处: 上层的变动完全不影响下层的结构。

TCP/IP 协议基本概念

OSI 模型所分的七层,在实际应用中,往往有一些层被整合,或者功能分散到其他层去。
TCP/IP 没有照搬 OSI 模型,也没有 一个公认的 TCP/IP 层级模型,一般划分为三层到五层模型来口述 TCP/IP 协议。

- 在此描述用一个通用的四层模型来描述,每一层都和 OSI 模型有较强的相关性但是又可能会有交叉。
- TCP/IP 的设计,是吸取了分层模型的精华思想——封装。每层对上一层口供服务的时候,上一层的数据结构是黑盒,直接作为本层的数据,而不需要关心上一层协议的任何细节。

TCP/IP 分层模型的分层以以太网上传输 UDP 数据包如图所示;

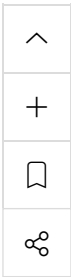
UDP 数据包.png

数据包

宽泛意义的数据包:每一个数据包都包含"标头"和"数据"两个部分."标头"包含本数据包的一些说明."数据"则是本数据包的内容。

细分数据包:

- 应用程序数据包: 标头部分规定应用程序的数据格式.数据部分传输具体的数据内容.**——对应上图中的数据! **
- TCP/UDP数据包:标头部分包含双方的发出端口和接收端口. UDP数据包:'标头'长度:8个字节,"数据包"总长度最大为65535字节,正好放进一个IP数据包. TCP数据包:理论上没有长度限制,但是,为了保证网络传输效率,通常不会超过IP数据长度,确保单个包不会被分割. ——对应上图中的UDP数据!
- IP数据包: 标头部分包含通信双方的IP地址,协议版本,长度等信息. '标头'长度:20~60字节,"数据包"总长度最大为65535字节. ——对应上图中的IP数据



- 以太网数据包: 最基础的数据包.标头部分包含了通信双方的MAC地址,数据类型等. '标头'长度:18字节,'数据'部分长度:46~1500字节. ——对应上图中的以太网数据

四层模型

1. 网络接口层

网络接口层包括用于协作IP数据在已有网络介质上传输的协议。

它定义像地址解析协议(Address Resolution Protocol,ARP)这样的协议,口供 TCP/IP 协议的数据结构和实际物理硬件之间的接口。

可以理解为：确定了网络数据包的形式 。

2. 网间层

网间层对应于 OSI 七层参考模型的网络层，本层包含 IP 协议、RIP 协议(Routing Information Protocol,路由信息协议),负责数据的包装、寻址和路由。同时还包含网间控制报文协议(Internet Control Message Protocol,ICMP)用来口供网络诊断信息；

可以理解为：该层时确定计算机的位置 。

3. 传输层

传输层对应于 OSI 七层参考模型的传输层,它口供两种端到端的通信服务。其中 TCP 协议(Transmission Control Protocol)口供可靠的数据流运输服务,UDP 协议(Use Datagram Protocol)口供不可靠的用户数据报服务。

TCP：三次握手、四次挥手；UDP：只发不管别人收不收到——任性哈

4. 应用层

应用层对应于 OSI 七层参考模型的应用层和表达层；

不明白的再看看7层参考模型 的描述 。

TCP/IP 协议族常用协议

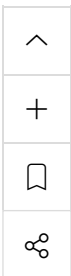
- 应用层：TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet 等等
- 传输层：TCP，UDP
- 网络层：IP，ICMP，OSPF，EIGRP，IGMP
- 数据链路层：SLIP，CSLIP，PPP，MTU

重要的 TCP/IP 协议族协议进行简单介绍:

- IP(Internet Protocol,网际协议)是网间层的主要协议,任务是在源地址和和目的地址之间传输数据。IP 协议只是尽最大努力来传输数据包,并不保证所有的包都可以传输到目的地,也不保证数据包的顺序和唯一。

- IP 定义了 TCP/IP 的地址,寻址方法,以及路由规则。现在广泛使用的 IP 协议有 IPv4 和 IPv6 两种:IPv4 使用 32 位二进制整数做地址,一般使用点分十进制方式表示,比如 192.168.0.1。
 - IP 地址由两部分组成,即网络号和主机号。故一个完整的 IPv4 地址往往表示为 192.168.0.1/24 或192.168.0.1/255.255.255.0 这种形式。
 - IPv6 是为了解决 IPv4 地址耗尽和其它一些问题而研发的最新版本的 IP。使用 128 位 整数表示地址,通常使用冒号分隔的十六进制来表示,并且可以省略其中一串连续的 0,如:fe80::200:1ff:fe00:1。
- 目前使用并不多！

- ICMP(Internet Control Message Protocol,网络控制消息协议)是 TCP/IP 的核心协议之一,用于在 IP 网络中发送控制消息,口供通信过程中的各种问题反馈。ICMP 直接使用 IP 数据包传输,但 ICMP 并不被视为 IP 协议的子协议。常见的联网状态诊断工具比如依赖于 ICMP 协议;
- TCP(TransmissionControlProtocol,传输控制协议)是一种面向连接的,可靠的, 基于字节流传输的通信协议。TCP 具有端口号的概念,用来标识同一个地址上的不同应用。口述 TCP 的标准文档是 RFC793。



- UDP(User Datagram Protocol,用户数据报协议)是一个面向数据报的传输层协 议。
UDP 的传输是不可靠的,简单的说就是发了不管,发送者不会知道目标地址 的数据通路是否发生拥塞,也不知道数据是否到达,是否完整以及是否还是原来的 次序。它同 TCP 一样有用来标识本地应用的端口号。所以应用 UDP 的应用,都能 够容忍一定数量的错误和丢包,但是对传输性能敏感的,比如流媒体、DNS 等。
- ECHO(Echo Protocol,回声协议)是一个简单的调试和检测工具。服务器会 原样回发它收到的任何数据,既可以使用 TCP 传输,也可以使用 UDP 传输。使用 端口号 7。
- DHCP(Dynamic Host Configuration Protocol,动态主机配置协议)是用于局域 网自动分配 IP 地址和主机配置的协议。可以使局域网的部署更加简单。
- DNS(Domain Name System,域名系统)是互联网的一项服务,可以简单的将用“.” 分隔的一般会有意义的域名转换成不易记忆的 IP 地址。一般使用 UDP 协议传输, 也可以使用 TCP,默认服务端口号 53。口
- FTP(File Transfer Protocol,文件传输协议)是用来进行文件传输的标准协议。 FTP 基于 TCP 使用端口号 20 来传输数据,21 来传输控制信息。
- TFTP(Trivial File Transfer Protocol,简单文件传输协议)是一个简化的文 件传输协议,其设计非常简单,通过少量存储器就能轻松实现,所以一般被用来通 过网络引导计算机过程中传输引导文件等小文件;
- SSH(Secure Shell,安全Shell),因为传统的网络服务程序比如TELNET本质上都极不安 全,明文传说数据和用户信息包括密码,SSH 被开发出来避免这些问题, 它其实是一个协议框架,有大量的扩展冗余能力,并且口供了加密压缩的通道可以 为其他协议使用。
- POP(Post Office Protocol,邮局协议)是支持通过客户端访问电子邮件的服务, 现在版本是 POP3,也有加密的版本 POP3S。协议使用 TCP,端口 110。
- SMTP(Simple Mail Transfer Protocol,简单邮件传输协议)是现在在互联网 上发送电子邮件的事实标准。使用 TCP 协议传输,端口号 25。
- HTTP(HyperText Transfer Protocol,超文本传输协议)是现在广为流行的WEB 网络的基 础,HTTPS 是 HTTP 的加密安全版本。协议通过 TCP 传输,HTTP 默认 使用端口 80,HTTPS 使用 443。

以上就是今天回顾的内容。
下篇回顾一下socket、TCP、UDP！

线程池

Executor 框架便是 Java 5 中引入的，其内部使用了线程池机制

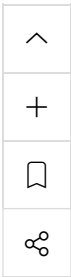
好处

第一：降低资源消耗 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

Java线程间的通信方式



wait()方法

wait()方法使得当前线程必须要等待，等到另外一个线程调用**notify()**或者**notifyAll()**方法。

当前的线程必须拥有当前对象的**monitor**，也即**lock**，就是锁。

线程调用**wait()**方法，释放它对锁的拥有权，然后等待另外的线程来通知它（通知的方式是**notify()**或者**notifyAll()**方法），这样它才能重新获得锁的拥有权和恢复执行。

要确保调用**wait()**方法的时候拥有锁，即，**wait()**方法的调用必须放在**synchronized**方法或**synchronized**块中。

一个小比较：

当线程调用了**wait()**方法时，它会释放掉对象的锁。

另一个会导致线程暂停的方法：**Thread.sleep()**，它会导致线程睡眠指定的毫秒数，但线程在睡眠的过程中是不会释放掉对象的锁的。

notify()方法

notify()方法会唤醒一个等待当前对象的锁的线程。

如果多个线程在等待，它们中的一个将会选择被唤醒。这种选择是随意的，和具体实现有关。（线程等待一个对象的锁是由于调用了**wait**方法中的一个）。

被唤醒的线程是不能被执行的，需要等到当前线程放弃这个对象的锁。

被唤醒的线程将和其他线程以通常的方式进行竞争，来获得对象的锁。也就是说，被唤醒的线程并没有什么优先权，也没有什么劣势，对象的下一个线程还是需要通过一般性的竞争。

notify()方法应该是被拥有对象的锁的线程所调用。

(This method should only be called by a thread that is the owner of this object's monitor.)

换句话说，和**wait()**方法一样，**notify**方法调用必须放在**synchronized**方法或**synchronized**块中。

wait()和**notify()**方法要求在调用时线程已经获得了对象的锁，因此对这两个方法的调用需要放在**synchronized**方法或**synchronized**块中。

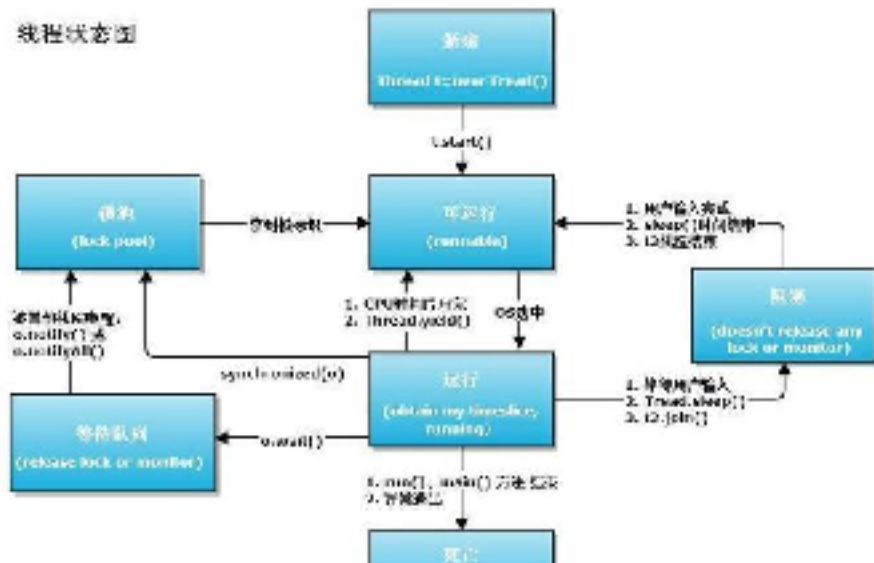
一个线程变为一个对象的锁的拥有者是通过下列三种方法：

1. 执行这个对象的**synchronized**实例方法。
2. 执行这个对象的**synchronized**语句块。这个语句块锁的是这个对象。
3. 对于Class类的对象，执行那个类的**synchronized**、**static**方法。

Java 线程有哪些状态，这些状态之间是如何转化的？



线程状态图



这里写图片描述

1. 新建(new): 新创建了一个线程对象。
2. 可运行(runnable): 线程对象创建后, 其他线程(比如main线程) 调用了该对象的 start()方法。该状态的线程位于可运行线程池中, 等待被线程调度选中, 获取cpu 的使用权。
3. 运行(running): 可运行状态(runnable)的线程获得了cpu 时间片 (timeslice) , 执行程序代码。
4. 阻塞(block): 阻塞状态是指线程因为某种原因放弃了cpu 使用权, 也即让出了cpu timeslice, 暂时停止运行。直到线程进入可运行(runnable)状态, 才有机会再次获得 cpu timeslice 转到运行(running)状态。阻塞的情况分三种:

(一). 等待阻塞: 运行(running)的线程执行o.wait()方法, JVM会把该线程放入等待队列(waiting queue)中。同时释放对象锁

(二). 同步阻塞: 运行(running)的线程在获取对象的同步锁时, 若该同步锁被别的线程占用, 则JVM会把该线程放入锁池(lock pool)中。

(三). 其他阻塞: 运行(running)的线程执行Thread.sleep(long ms)或t.join()方法, 或者发出了I/O请求时, JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时, 线程重新转入可运行(runnable)状态。

1. 死亡(dead): 线程run()、main() 方法执行结束, 或者因异常退出了run()方法, 则该线程结束生命周期。死亡的线程不可再次复生。

List接口、Set接口和Map接口的区别

1、List和Set接口自Collection接口, 而Map不是继承的Collection接口

接口提供3种集合的视图,Map的内容可以被当做一组key集合, 一组value集合, 或者一组key-value映射;



2、List接口

元素有放入顺序，元素可重复

List接口有三个实现类：LinkedList, ArrayList, Vector

LinkedList：底层基于链表实现，链表内存是散乱的，每一个元素存储本身内存地址的同时还存储下一个元素的地址；ArrayList和Vector的区别：ArrayList是非线程安全的，效率高；Vector是基于线程安全的，效率低

List是一种有序的Collection，可以通过索引访问集合中的数据，List比Collection多了10个方法，

1).所有的索引返回的方法都有可能抛出一个IndexOutOfBoundsException异常

2).subList(int fromIndex, int toIndex)返回的是包括fromIndex，不包括toIndex的子List

所有的List中只能容纳单个不同类型的对象组成的表，而不是Key-Value键值对。例如：[tom, koo, too, koo]；

所有的List中可以有相同的元素，例如Vector中可以有 [tom, koo, too, koo]；

所有的List中可以有null元素，例如[tom, null, 1]；

基于Array的List (Vector, ArrayList) 适合查询，而LinkedList (链表) 适合添加，删除

3、Set接口

元素无放入顺序，元素不可重复（注意：元素虽然无放入顺序，但是元素在set中的位置是有该元素的HashCode决定的）
Set接口有两个实现类：HashSet(底层由HashMap实现)，LinkedHashSet

SortedSet接口有一个实现类：TreeSet (底层由平衡二叉树实现)

Query接口有一个实现类：LinkList

Set具有与Collection完全一样的接口，因此没有任何额外的功能，不像前面有两个不同的List。实现Set

Set：存入Set的每个元素都必须是唯一的，因为Set不保存重复元素。加入Set的元素必须定义equals()

HashSet：为快速查找设计的Set。存入HashSet的对象必须定义hashCode()。

TreeSet：保存次序的Set，底层为树结构。使用它可以从Set中提取有序的序列。

LinkedHashSet：具有HashSet的查询速度，且内部使用链表维护元素的顺序(插入的次序)。于是

4、map接口

以键值对的方式出现的

Map接口有三个实现类：HashMap, Hashtable, LinkeHashMap

HashMap非线程安全，高效，支持null；

Hashtable线程安全，低效，不支持null

SortedMap有一个实现类：TreeMap

Session机制



一、术语session

session，中文经常翻译为会话，其本来的含义是指有始有终的一系列动作/消息，比如打电话时从拿起电话拨号到挂断电话这中间的一系列过程可以称之为一个session。有时候我们可以看到这样的话“在一个浏览器会话期间，...”，这里的会话一词用的就是其本义，是指从一个浏览器窗口打开到关闭这个期间^①。最混乱的是“用户（客户端）在一次会话期间”这样一句话，它可能指用户的一系列动作（一般情况下是同某个具体目的相关的一系列动作，比如从登录到选购商品到结账登出这样一个网上购物的过程，有时候也被称为一个transaction），然而有时候也可能仅仅是指一次连接，也有可能是指含义^①，其中的差别只能靠上下文来推断^②。

然而当session一词与网络协议相关联时，它又往往隐含了“面向连接”和/或“保持状态”这样两个含义，“面向连接”指的是在通信双方在通信之前要先建立一个通信的渠道，比如打电话，直到对方接了电话通信才能开始，与此相对的是写信，在你把信发出去的时候你并不能确认对方的地址是否正确，通信渠道不一定能建立，但对发信人来说，通信已经开始了。“保持状态”则是指通信的一方能够把一系列的消息关联起来，使得消息之间可以互相依赖，比如一个服务员能够认出再次光临的老顾客并且记得上次这个顾客还欠店里一块钱。这一类的例子有“一个TCP session”或者“一个POP3 session”^③。

而到了web服务器蓬勃发展的时代，session在web开发语境下的语义又有了新的扩展，它的含义是指一类用来在客户端与服务器之间保持状态的解决方案^④。有时候session也用来指这种解决方案的存储结构，如“把xxx保存在session里”^⑤。由于各种用于web开发的语言在一定程度上都提供了对这种解决方案的支持，所以在某种特定语言的语境下，session也被用来指代该语言的解决方案，比如经常把Java里提供的javax.servlet.http.HttpSession简称为session^⑥。

鉴于这种混乱已不可改变，本文中session一词的运用也会根据上下文有不同的含义，请大家注意分辨。

在本文中，使用中文“浏览器会话期间”来表达含义^①，使用“session机制”来表达含义^④，使用“session”表达含义^⑤，使用具体的“HttpSession”来表达含义^⑥

** 二、HTTP协议与状态保持**

HTTP协议本身是无状态的，这与HTTP协议本来的目的是相符的，客户端只需要简单的向服务器请求下载某些文件，无论是客户端还是服务器都没有必要纪录彼此过去的行为，每一次请求之间都是独立的，好比一个顾客和一个自动售货机或者一个普通的（非会员制）大卖场之间的关系一样。

然而聪明（或者贪心？）的人们很快发现如果能够提供一些按需生成的动态信息会使web变得更加有用，就像给有线电视加上点播功能一样。这种需求一方面迫使HTML逐步添加了表单、脚本、DOM等客户端行为，另一方面在服务器端则出现了CGI规范以响应客户端的动态请求，作为传输载体的HTTP协议也添加了文件上载、cookie这些特性。其中cookie的作用就是为了解决HTTP协议无状态的缺陷所作出的努力。至于后来出现的session机制则是另一种在客户端与服务器之间保持状态的解决方案。

让我们用几个例子来描述一下cookie和session机制之间的区别与联系。笔者曾经常去的一家咖啡店有喝5杯咖啡免费赠一杯咖啡的优惠，然而一次性消费5杯咖啡的机会微乎其微，这时就需要某种方式来纪录某位顾客的消费数量。想象一下其实也无外乎下面的几种方案：

- 1、该店的店员很厉害，能记住每位顾客的消费数量，只要顾客一走进咖啡店，店员就知道该怎么对待了。这种做法就是协议本身支持状态。
- 2、发给顾客一张卡片，上面记录着消费的数量，一般还有个有效期限。每次消费时，如果顾客出示这张卡片，则此次消费就会与以前或以后的消费相联系起来。这种做法就是在客户端保持状态。
- 3、发给顾客一张会员卡，除了卡号之外什么信息也不纪录，每次消费时，如果顾客出示该卡片，则店员在店里的纪录本上找到这个卡号对应的纪录添加一些消费信息。这种做法就是在服务器端保持状态。

由于HTTP协议是无状态的，而出于种种考虑也不希望使之成为有状态的，因此，后面两种方案就成为现实的选择。具体来说cookie机制采用的是在客户端保持状态的方案，而session机制采用的是在服务器端保持状态的方案。同时我们也看到，由于采用服务器端保持状态的方案在客户端也需要保存一个标识，所以



session机制可能需要借助于cookie机制来达到保存标识的目的，但实际上它还有其他选择。

**三、理解cookie机制 **

cookie机制的基本原理就如上面的例子一样简单，但是还有几个问题需要解决：“会员卡”如何分发；“会员卡”的内容；以及客户如何使用“会员卡”。

正统的cookie分发是通过扩展HTTP协议来实现的，服务器通过在HTTP的响应头中加上一行特殊的指示以提示浏览器按照指示生成相应的cookie。然而纯粹的客户端脚本如JavaScript或者VBScript也可以生成cookie。

而cookie的使用是由浏览器按照一定的原则在后台自动发送给服务器的。浏览器检查所有存储的cookie，如果某个cookie所声明的作用范围大于等于将要请求的资源所在的位置，则把该cookie附在请求资源的HTTP请求头上发送给服务器。意思是麦当劳的会员卡只能在麦当劳的店里出示，如果某家分店还发行了自己的会员卡，那么进这家店的时候除了要出示麦当劳的会员卡，还要出示这家店的会员卡。

cookie的内容主要包括：名字，值，过期时间，路径和域。

其中域可以指定某一个域比如.google.com，相当于总店招牌，比如宝洁公司，也可以指定一个域下的具体某台机器比如www.google.com

(https://link.jianshu.com?t=http://www.google.com/)或者froogle.google.com，可以用飘柔来做比。

路径就是跟在域名后面的URL路径，比如/或者/foo等等，可以用某飘柔专柜做比。路径与域合在一起就构成了cookie的作用范围。如果不设置过期时间，则表示这个cookie的生命期为浏览器会话期间，只要关闭浏览器窗口，cookie就消失了。这种生命期为浏览器会话期的cookie被称为会话cookie。会话cookie一般不存储在硬盘上而是保存在内存里，当然这种行为并不是规范规定的。如果设置了过期时间，浏览器就会把cookie保存到硬盘上，关闭后再次打开浏览器，这些cookie仍然有效直到超过设定的过期时间。

存储在硬盘上的cookie可以在不同的浏览器进程间共享，比如两个IE窗口。而对于保存在内存里的cookie，不同的浏览器有不同的处理方式。对于IE，在一个打开的窗口上按Ctrl-N（或者从文件菜单）打开的窗口可以与原窗口共享，而使用其他方式新开的IE进程则不能共享已经打开的窗口的内存cookie；对于Mozilla Firefox0.8，所有的进程和标签页都可以共享同样的cookie。一般来说是用javascript的window.open打开的窗口会与原窗口共享内存cookie。浏览器对于会话cookie的这种只认cookie不认人的处理方式经常给采用session机制的web应用程序开发者造成很大的困扰。

Cookie和Session的区别

HTTP请求是无状态的。



共同之处：

cookie和session都是用来跟踪浏览器用户身份的会话方式。

区别：

- cookie数据保存在客户端，session数据保存在服务器端。简单的说，当你登录一个网站的时候，如果web服务器端使用的是session，那么所有的数据都保存在服务器上，客户端每次请求服务器的时候会发送当前会话的sessionid，服务器根据当前sessionid判断相应的用户数据标志，以确定用户是否登录或具有某种权限。由于数据是存储在服务器上面，所以你不能伪造。
- sessionid是服务器和客户端链接时候随机分配的。如果浏览器使用的是cookie，那么所有的数据都保存在浏览器端，比如你登录以后，服务器设置了cookie用户名，那么当你再次请求服务器的时候，浏览器会将用户名一块发送给服务器，这些变量有一定的特殊标记。服务器会解释为cookie变量，所以只要不关闭浏览器，那么cookie变量一直是有效的，所以能够保证长时间不掉线。如果你能够截获某个用户的 cookie变量，然后伪造一个数据包发送过去，那么服务器还是认为你是合法的。所以，使用 cookie被攻击的可能性比较大。

如果设置了的有效时间，那么它会将 cookie保存在客户端的硬盘上，下次再访问该网站的时候，浏览器先检查有没有 cookie，如果有的话，就读取该 cookie，然后发送给服务器。如果你在机器上面保存了某个论坛 cookie，有效期是一年，如果有人入侵你的机器，将你的 cookie拷走，然后放在他的浏览器的目录下，那么他登录该网站的时候就是用你的的身份登录的。所以 cookie是可以伪造的。当然，伪造的时候需要主意，直接copy cookie文件到 cookie目录，浏览器是不认的，他有一个index.dat文件，存储了 cookie文件的建立时间，以及是否有修改，所以你必须先要有该网站的 cookie文件，并且要从保证时间上骗过浏览器

两个都可以用来存私密的东西，同样也都有有效期的说法,区别在于session是放在服务器上的，过期与否取决于服务期的设定，cookie是存在客户端的，过去与否可以在cookie生成的时候设置进去。

(1)cookie数据存放在客户的浏览器上，session数据放在服务器上

(2)cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗，如果主要考虑到安全应当使用session

(3)session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，如果主要考虑到减轻服务器性能方面，应当使用COOKIE

(4)单个cookie在客户端的限制是3K，就是说一个站点在客户端存放的COOKIE不能3K。

(5)所以：将登陆信息等重要信息存放为SESSION;其他信息如果需要保留，可以放在COOKIE中

Java中的equals和hashCode方法详解



`equals()` 方法是用来判断其他的对象是否和该对象相等。

`equals()`方法在`object`类中定义如下：

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

很明显是对两个对象的地址值进行的比较（即比较引用是否相同）。但是我们知道，`String`、`Math`、`Integer`、`Double`等这些封装类在使用`equals()`方法时，已经覆盖了`object`类的`equals()`方法。

比如在`String`类中如下：

[



复制代码

](javascript:void(0);)

```
public boolean equals(Object anObject) {  
    if (this == anObject) {  
        return true;  
    }  
    if (anObject instanceof String) {  
        String anotherString = (String)anObject;  
        int n = count;  
        if (n == anotherString.count) {  
            char v1[] = value;  
            char v2[] = anotherString.value;  
            int i = offset;  
            int j = anotherString.offset;  
            while (n- != 0) {  
                if (v1[i++] != v2[j++])  
                    return false;  
            }  
            return true;  
        }  
    }  
    return false;  
}
```

很明显，这是进行的内容比较，而已经不再是地址的比较。依次类推`Math`、`Integer`、`Double`等这些类都是重写了`equals()`方法的，从而进行的是内容的比较。当然，基本类型是进行值的比较。

它的性质有：

- **自反性**（reflexive）。对于任意不为 `null` 的引用值 `x`，`x.equals(x)` 一定是 `true`。
- **对称性**（symmetric）。对于任意不为 `null` 的引用值 `x` 和 `y`，当且仅当 `x.equals(y)` 是 `true` 时，`y.equals(x)` 也是 `true`。
- **传递性**（transitive）。对于任意不为 `null` 的引用值 `x`、`y` 和 `z`，如果 `x.equals(y)` 是 `true`，同时 `y.equals(z)` 是 `true`，那么 `x.equals(z)` 一定是 `true`。
- **一致性**（consistent）。对于任意不为 `null` 的引用值 `x` 和 `y`，如果用于 `equals` 比较的对象信息没有被修改的话，多次调用时 `x.equals(y)` 要么一致地返回 `true` 要么一致地返回 `false`。
- 对于任意不为 `null` 的引用值 `x`，`x.equals(null)` 返回 `false`。



对于 `Object` 类来说，`equals()` 方法在对象上实现的是差别可能性最大的等价关系，即，对于任意非 `null` 的引用值 `x` 和 `y`，当且仅当 `x` 和 `y` 引用的是同一个对象，该方法才会返回 `true`。

需要注意的是当`equals()`方法被`override`时，`hashCode()`也要被`override`。按照一般`hashCode()`方法的实现来说，相等的对象，它们的`hash code`一定相等。

hashCode() 方法详解

`hashCode()` 方法给对象返回一个`hash code`值。这个方法被用于`hash tables`，例如 `HashMap`。

它的性质是：

- 在一个Java应用的执行期间，如果一个对象提供给`equals`做比较的信息没有被修改的话，该对象多次调用 `hashCode()` 方法，该方法必须始终如一返回同一个 `integer`。
- 如果两个对象根据 `equals(Object)` 方法是相等的，那么调用二者各自的 `hashCode()` 方法必须产生同一个`integer`结果。
- 并不要求根据 `equals(java.lang.Object)` 方法不相等的两个对象，调用二者各自的 `hashCode()` 方法必须产生不同的`integer`结果。然而，程序员应该意识到对于不同的对象产生不同的`integer`结果，有可能会提高`hash table`的性能。

Java中CAS算法--乐观锁的一种实现方式



悲观者与乐观者的做事方式完全不一样，悲观者的人生观是一件事情我必须要百分之百完全控制才会去做，否则就认为这件事情一定会出问题；而乐观者的人生观则相反，凡事不管最终结果如何，他都会先尝试去做，大不了最后不成功。这就是悲观锁与乐观锁的区别，悲观锁会把整个对象加锁占为自有后才去做操作，乐观锁不获取锁直接做操作，然后通过一定检测手段决定是否更新数据。这一节将对乐观锁进行深入探讨。

上节讨论的**Synchronized**互斥锁属于悲观锁，它有一个明显的缺点，它不管数据存不存在竞争都加锁，随着并发量增加，且如果锁的时间比较长，其性能开销将会变得很大。有没有办法解决这个问题？答案是基于冲突检测的乐观锁。这种模式下，已经没有什么所谓的锁概念了，**每条线程都直接先去执行操作，计算完成后检测是否与其他线程存在共享数据竞争，如果没有则让此操作成功，如果存在共享数据竞争则可能不断地重新执行操作和检测，直到成功为止，可叫CAS自旋。**

乐观锁的核心算法是CAS（Compare and Swap，比较并交换），它涉及到三个操作数：内存值、预期值、新值。当且仅当预期值和内存值相等时才将内存值修改为新值。这样处理的逻辑是，**首先检查某块内存的值是否跟之前我读取时的一样，如不一样则表示期间此内存值已经被别的线程更改过，舍弃本次操作，否则说明期间没有其他线程对此内存值操作，可以把新值设置给此块内存。**如图2-5-4-1，有两个线程可能会差不多同时对某内存操作，线程二先读取某内存值作为预期值，执行到某处时线程二决定将新值设置到内存块中，如果线程一在此期间修改了内存块，则通过CAS即可以检测出来，假如检测没问题则线程二将新值赋予内存块。

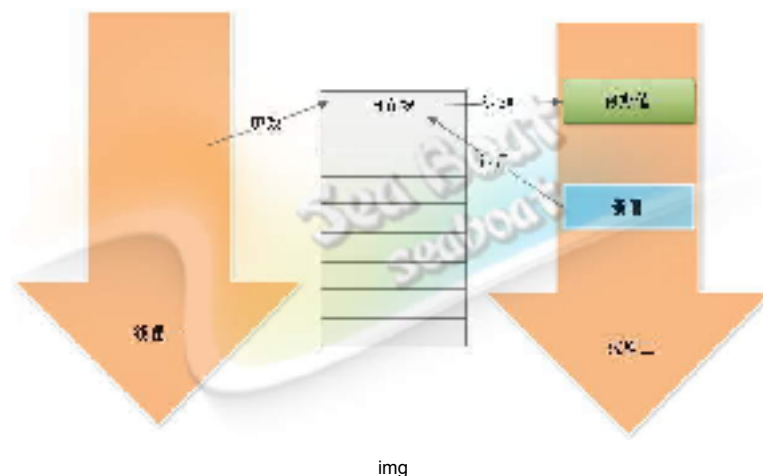


图2-5-4-1

假如你足够细心你可能会发现一个疑问，比较和交换，从字面上就有两个操作了，更别说实际CAS可能会有更多的执行指令，他们是原子性的吗？如果非原子性又怎么保证CAS操作期间出现并发带来的问题？我是不是需要用上节提到的互斥锁来保证他的原子性操作？**CAS肯定是具有原子性的**，不然就谈不上在并发中使用了，但这个原子性是由**CPU硬件指令实现保证的**，即使用JNI调用native方法调用由C++编写的硬件级别指令，jdk中提供了Unsafe类执行这些操作。另外，你可能想着CAS是通过互斥锁来实现原子性的，这样确实能实现，但用这种方式来保证原子性显示毫无意义。下面一个伪代码加深对CAS的理解：

```
public class AtomicInt {
    private volatile int value;
    public final int get() {
        return value;
    }
    public final int getAndIncrement() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return current;
        }
    }
    public final boolean compareAndSet(int expect, int update) {
        Unsafe类提供的硬件级别的compareAndSwapInt方法;
    }
}
```

其中最重要的方法是getAndIncrement方法，它里面实现了基于CAS的自旋。

现在已经了解乐观锁及CAS相关机制，乐观锁避免了悲观锁独占对象的现象，同时也提高了并发性能，但它也有缺点：

① 观锁只能保证一个共享变量的原子操作。如上例子，自旋过程中只能保证value变量的原子性，这时如果多一个或几个变量，乐观锁将变得力不从心，但互斥锁能轻易解决，不管对象数量多少及对象颗粒度大小。

② 长时间自旋可能导致开销大。假如CAS长时间不成功而一直自旋，会给CPU带来很大的开销。

③ ABA问题。CAS的核心思想是通过比对内存值与预期值是否一样而判断内存值是否被改过，但这个判断逻辑不严谨，假如内存值原来是A，后来被一条线程改为B，最后又被改成了A，则CAS认为此内存值并没有发生改变，但实际上是有被其他线程改过的，这种情况对依赖过程值的情景的运算结果影响很大。解决思路是引入版本号，每次变量更新都把版本号加一。

乐观锁是对悲观锁的改进，虽然它也有缺点，但它确实已经成为提高并发性能的主要手段，而且jdk中的并发包也大量使用基于CAS的乐观锁。

TimSort原理

comparable与comparator的区别



Comparable和Comparator的区别

([https://link.jianshu.com?](https://link.jianshu.com?t=http://www.cnblogs.com/szlbm/p/5504634.html)

[t=http://www.cnblogs.com/szlbm/p/5504634.html](https://link.jianshu.com?t=http://www.cnblogs.com/szlbm/p/5504634.html))

初次碰到这个问题是之前有一次电话面试，问了一个小时的问题，其中有一个问题就问到Comparable和Comparator的区别，当时没答出来。之后是公司入职时候做的一套Java编程题，里面用JUnit跑用例的时候也用到了Comparator接口，再加上JDK的大量的类包括常见的String、Byte、Char、Date等都实现了Comparable接口，因此要学习一下这两个类的区别以及用法。

Comparable

Comparable可以认为是一个**内比较器**，实现了Comparable接口的类有一个特点，就是这些类是可以和自己比较的，至于具体和另一个实现了Comparable接口的类如何比较，则依赖compareTo方法的实现，compareTo方法也被称为**自然比较方法**。如果开发者add进入一个Collection的对象想要Collections的sort方法帮你自动进行排序的话，那么这个对象必须实现Comparable接口。compareTo方法的返回值是int，有三种情况：

- 1、比较者大于被比较者（也就是compareTo方法里面的对象），那么返回正整数
- 2、比较者等于被比较者，那么返回0
- 3、比较者小于被比较者，那么返回负整数

写个很简单的例子：

```
public class Domain implements Comparable<Domain>
{
    private String str;

    public Domain(String str)
    {
        this.str = str;
    }

    public int compareTo(Domain domain)
    {
        if (this.str.compareTo(domain.str) > 0)
            return 1;
        else if (this.str.compareTo(domain.str) == 0)
            return 0;
        else
            return -1;
    }

    public String getStr()
    {
        return str;
    }
}
```

```
public static void main(String[] args)
{
    Domain d1 = new Domain("c");
    Domain d2 = new Domain("c");
    Domain d3 = new Domain("b");
    Domain d4 = new Domain("d");
    System.out.println(d1.compareTo(d2));
    System.out.println(d1.compareTo(d3));
    System.out.println(d1.compareTo(d4));
}
```

运行结果为：



```
0
1
-1
```

注意一下，前面说实现Comparable接口的类是可以支持和自己比较的，但是其实代码里面Comparable的泛型未必就一定要Domain，将泛型指定为String或者指定为其他任何任何类型都可以---只要开发者指定了具体的比较算法就行。

Comparator

Comparator可以认为是是一个**外比较器**，个人认为有两种情况可以使用实现Comparator接口的方式：

- 1、一个对象不支持自己和自己比较（没有实现Comparable接口），但是又想对两个对象进行比较
- 2、一个对象实现了Comparable接口，但是开发者认为compareTo方法中的比较方式并不是自己想要的那种比较方式

Comparator接口里面有一个compare方法，方法有两个参数T o1和T o2，是泛型的表示方式，分别表示待比较的两个对象，方法返回值和Comparable接口一样是int，有三种情况：

- 1、o1大于o2，返回正整数
- 2、o1等于o2，返回0
- 3、o1小于o3，返回负整数

写个很简单的例子，上面代码的Domain不变（假设这就是第2种场景，我对这个compareTo算法实现不满意，要自己写实现）：

```
public class DomainComparator implements Comparator<Domain>
{
    public int compare(Domain domain1, Domain domain2)
    {
        if (domain1.getStr().compareTo(domain2.getStr()) > 0)
            return 1;
        else if (domain1.getStr().compareTo(domain2.getStr()) == 0)
            return 0;
        else
            return -1;
    }
}
```

```
public static void main(String[] args)
{
    Domain d1 = new Domain("c");
    Domain d2 = new Domain("c");
    Domain d3 = new Domain("b");
    Domain d4 = new Domain("d");
    DomainComparator dc = new DomainComparator();
    System.out.println(dc.compare(d1, d2));
    System.out.println(dc.compare(d1, d3));
    System.out.println(dc.compare(d1, d4));
}
```

看一下运行结果：

```
0
1
-1
```



当然因为泛型指定死了，所以实现Comparator接口的实现类只能是两个相同的对象（不能一个Domain、一个String）进行比较了，因此实现Comparator接口的实现类一般都会以"待比较的实体类+Comparator"来命名

总结

总结一下，两种比较器Comparable和Comparator，后者相比前者有如下优点：

- 1、如果实现类没有实现Comparable接口，又想对两个类进行比较（或者实现类实现了Comparable接口，但是对compareTo方法内的比较算法不满意），那么可以实现Comparator接口，自定义一个比较器，写比较算法
- 2、实现Comparable接口的方式比实现Comparator接口的耦合性要强一些，如果要修改比较算法，要修改Comparable接口的实现类，而实现Comparator的类是在外部进行比较的，不需要对实现类有任何修改。从这个角度说，其实有些不太好，尤其在我们将实现类的.class文件打成一个.jar文件提供给开发者使用的时候。实际上实现Comparator接口的方式后面会写到就是一种典型的策略模式。

手写单例模式（线程安全）



解法一：只适合单线程环境（不好）

```
package test;
/**
 * @author xiaoping
 */
public class Singleton {
    private static Singleton instance=null;
    private Singleton(){

    }
    public static Singleton getInstance(){
        if(instance==null){
            instance=new Singleton();
        }
        return instance;
    }
}
```

注解:Singleton的静态属性instance中，只有instance为null的时候才创建一个实例，构造函数私有，确保每次都只创建一个，避免重复创建。

缺点：只在单线程的情况下正常运行，在多线程的情况下，就会出问题。例如：当两个线程同时运行到判断instance是否为空的if语句，并且instance确实没有创建好时，那么两个线程都会创建一个实例。

解法二：多线程的情况可以用。（懒汉式，不好）

```
public class Singleton {
    private static Singleton instance=null;
    private Singleton(){

    }
    public static synchronized Singleton getInstance(){
        if(instance==null){
            instance=new Singleton();
        }
        return instance;
    }
}
```

注解：在解法一的基础上加上了同步锁，使得在多线程的情况下可以用。例如：当两个线程同时想创建实例，由于在一个时刻只有一个线程能得到同步锁，当第一个线程加上锁以后，第二个线程只能等待。第一个线程发现实例没有创建，创建之。第一个线程释放同步锁，第二个线程才可以加上同步锁，执行下面的代码。由于第一个线程已经创建了实例，所以第二个线程不需要创建实例。保证在多线程的环境下也只有一个实例。

缺点：每次通过getInstance方法得到singleton实例的时候都有一个试图去获取同步锁的过程。而众所周知，加锁是很耗时的。能避免则避免。

解法三：加同步锁时，前后两次判断实例是否存在（可行）

```
public class Singleton {
    private static Singleton instance=null;
    private Singleton(){ }
    public static Singleton getInstance(){
        if(instance==null){
            synchronized(Singleton.class){
                if(instance==null){
                    instance=new Singleton();
                }
            }
        }
        return instance;
    }
}
```



注解：只有当instance为null时，需要获取同步锁，创建一次实例。当实例被创建，则无需试图加锁。

缺点：用双重if判断，复杂，容易出错。

解法四：饿汉式（建议使用）

```
public class Singleton {
    private static Singleton instance=new Singleton();
    private Singleton(){
    }
    public static Singleton getInstance(){
        return instance;
    }
}
```

注解：初试化静态的instance创建一次。如果我们在Singleton类里面写一个静态的方法不需要创建实例，它仍然会早早的创建一次实例。而降低内存的使用率。

缺点：没有lazy loading的效果，从而降低内存的使用率。

解法五：静态内部内。（建议使用）

```
public class Singleton {
    private Singleton(){
    }
    private static class SingletonHolder{
        private final static Singleton instance=new Singleton();
    }
    public static Singleton getInstance(){
        return SingletonHolder.instance;
    }
}
```

注解：定义一个私有的内部类，在第一次用这个嵌套类时，会创建一个实例。而类型为SingletonHolder的类，只有在Singleton.getInstance()中调用，由于私有的属性，他人无法使用SingleHolder，不调用Singleton.getInstance()就不会创建实例。

优点：达到了lazy loading的效果，即按需创建实例。

JVM参数初始值

初始堆大小： 1/64内存-Xms 最大堆大小： 1/4内存-Xmx

初始永久代大小： 1/64内存-XX:PermSize 最大堆大小： 1/4内存-XX:MaxPermSize

Java8的内存分代改进



JAVA 8持久代已经被彻底删除了

取代它的是另一个内存区域也被称为元空间。

元空间 —— 快速入门

- 它是本地内存中的一部分
- 最直接的表现就是OOM（内存溢出）问题将不复存在，因为直接利用的是本地内存。
- 它可以通过-XX:MetaspaceSize和-XX:MaxMetaspaceSize来进行调整
- 当到达XX:MetaspaceSize所指定的阈值后会开始进行清理该区域
- 如果本地空间的内存用尽了会收到java.lang.OutOfMemoryError: Metadata space的错误信息。
- 和持久代相关的JVM参数-XX:PermSize及-XX:MaxPermSize将会被忽略掉，并且在启动的时候给出警告信息。
- 充分利用了Java语言规范中的好处：类及相关的元数据的生命周期与类加载器的一致

元空间 —— 内存分配模型绝大多数的类元数据的空间都从本地内存中分配。用来描述类元数据的类也被删除了，分元数据分配了多个虚拟内存空间给每个类加载器分配一个内存块的列表，只进行线性分配。块的大小取决于类加载器的类型，sun/反射/代理对应的类加载器的块会小一些。不会单独回收某个类，如果GC发现某个类加载器不再存活了，会把相关的空间整个回收掉。这样减少了碎片，并节省GC扫描和压缩的时间。

元空间 —— 调优使用-XX:MaxMetaspaceSize参数可以设置元空间的最大值，默认是没有上限的，也就是说你的系统内存上限是多少它就是多少。使用-XX:MetaspaceSize选项指定的是元空间的初始大小，如果没有指定的话，元空间会根据应用程序运行时的需要动态地调整大小。一旦类元数据的使用量达到了“MaxMetaspaceSize”指定的值，对于无用的类和类加载器，垃圾收集此时会触发。为了控制这种垃圾收集的频率和延迟，合适的监控和调整Metaspace非常有必要。过于频繁的Metaspace垃圾收集是类和类加载器发生内存泄露的征兆，同时也说明你的应用程序内存大小不合适，需要调整。

** 快速过一遍JVM的内存结构，JVM中的内存分为5个虚拟的区域：（程序计数器、

虚拟机栈、本地方法栈、堆区、方法区）

Java8的JVM持久代 - 何去何从？

堆

- 你的Java程序中所分配的每一个对象都需要存储在内存里。堆是这些实例化的对象所存储的地方。是的——都怪new操作符，是它把你的Java堆都占满了的！
- 它由所有线程共享
- 当堆耗尽的时候，JVM会抛出java.lang.OutOfMemoryError 异常
- 堆的大小可以通过JVM选项-Xms和-Xmx来进行调整



堆被分为：

- Eden区 —— 新对象或者生命周期很短的对象会存储在这个区域中，这个区的大小可以通过-XX:NewSize和-XX:MaxNewSize参数来调整。新生代GC（垃圾回收器）会清理这一区域。
- Survivor区 —— 那些历经了Eden区的垃圾回收仍能存活下来的依旧存在引用的对象会待在这个区域。这个区的大小可以由JVM参数-XX:SurvivorRatio来进行调节。
- 老年代 —— 那些在历经了Eden区和Survivor区的多次GC后仍然存活下来的对象（当然了，是拜那些挥之不去的引用所赐）会存储在这个区里。这个区会由一个特殊的垃圾回收器来负责。老年代中的对象的回收是由老年代的GC（major GC）来进行的。

方法区

- 也被称为非堆区域（在HotSpot JVM的实现当中）
- 它被分为两个主要的子区域

持久代 —— 这个区域会 存储包括类定义，结构，字段，方法（数据及代码）以及常量在内的类相关数据。它可以通过-XX:PermSize及 -XX:MaxPermSize来进行调节。如果它的空间用完了，会导致java.lang.OutOfMemoryError: PermGen space的异常。

代码缓存——这个缓存区域是用来存储编译后的代码。编译后的代码就是本地代码（硬件相关的），它是由JIT（Just In Time）编译器生成的，这个编译器是Oracle HotSpot JVM所特有的。

JVM栈

- 和Java类中的方法密切相关
- 它会存储局部变量以及方法调用的中间结果及返回值
- Java中的每个线程都有自己专属的栈，这个栈是别的线程无法访问的。
- 可以通过JVM选项-Xss来进行调整

本地栈

- 用于本地方法（非Java代码）
- 按线程分配

PC寄存器

- 特定线程的程序计数器
- 包含JVM正在执行的指令的地址（如果是本地方法的话它的值则未定义）

好吧，这就是JVM内存分区的基础知识了。现在再说持久代这个话题吧。

对Java内存模型的理解以及其在并发当中的作用



概述

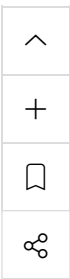
Java平台自动集成了线程以及多处理器技术，这种集成程度比Java以前诞生的计算机语言要厉害很多，该语言针对多种异构平台的平台独立性而使用的多线程技术支持也是具有开拓性的一面，有时候在开发Java同步和线程安全要求很严格的程序时，往往容易混淆的一个概念就是内存模型。究竟什么是内存模型？内存模型描述了程序中各个变量（实例域、静态域和数组元素）之间的关系，以及在实际计算机系统中将变量存储到内存和从内存中取出变量这样的底层细节，对象最终是存储在内存里面的，这点没有错，但是编译器、运行库、处理器或者系统缓存可以有特权在变量指定内存位置存储或者取出变量的值。【JMM】（Java Memory Model的缩写）允许编译器和缓存以数据在处理器特定的缓存（或寄存器）和主存之间移动的次序拥有重要的特权，除非程序员使用了final或synchronized明确请求了某些可见性的保证。在Java中应为不同的目的可以将java划分为两种内存模型：gc内存模型。并发内存模型。

gc内存模型

java与c++之间有一堵由内存动态分配与垃圾收集技术所围成的“高墙”。墙外面的人想进去，墙里面的人想出来。java在执行java程序的过程中会把它管理的内存划分若干个不同功能的数据管理区域。如图：

img

img



img

hotspot中的gc内存模型

整体上。分为三部分：栈，堆，程序计数器，他们每一部分有其各自的用途；虚拟机栈保存着每一条线程的执行程序调用堆栈；堆保存着类对象、数组的具体信息；程序计数器保存着每一条线程下一次执行指令位置。这三块区域中栈和程序计数器是线程私有的。也就是说每一个线程拥有其独立的栈和程序计数器。我们可以看看具体结构：

虚拟机/本地方法栈

在栈中，会为每一个线程创建一个栈。线程越多，栈的内存使用越大。对于每一个线程栈。当一个方法在线程中执行的时候，会在线程栈中创建一个栈帧(stack frame)，用于存放该方法的上下文(局部变量表、操作数栈、方法返回地址等等)。每一个方法从调用到执行完毕的过程，就是对应着一个栈帧入栈出栈的过程。

本地方法栈与虚拟机栈发挥的作用是类似的，他们之间的区别不过是虚拟机栈为虚拟机执行java(字节码)服务的，而本地方法栈是为虚拟机执行native方法服务的。

方法区/堆

在hotspot的实现中，方法区就是在堆中称为永久代的堆区域。几乎所有的对象/数组的内存空间都在堆上(有少部分在栈上)。在gc管理中，将虚拟机堆分为永久代、老年代、新生代。通过名字我们可以知道一个对象新建一般在新生代。经过几轮的gc。还存活的对象会被移到老年代。永久代用来保存类信息、代码段等几乎不会变的数据。堆中的所有数据是线程共享的。

- 新生代：应为gc具体实现的优化的原因。hotspot又将新生代划分为一个eden区和两个survivor区。每一次新生代gc时候。只用到一个eden区，一个survivor区。新生代一般的gc策略为mark-copy。
- 老年代：当新生代中的对象经过若干轮gc后还存活/或survivor在gc内存不够的时候。会把当前对象移动到老年代。老年代一般gc策略为mark-compact。
- 永久代：永久代一般可以不参与gc。应为其中保存的是一些代码/常量数据/类信息。在永久代gc。清楚的是类信息以及常量池。

JVM内存模型中分两大块，一块是 NEW Generation, 另一块是Old Generation. 在New Generation中，有一个叫Eden的空间，主要是用来存放新生的对象，还有两个Survivor Spaces (from,to)，它们用来存放每次垃圾回收后存活下来的对象。在Old Generation中，主要存放应用程序中生命周期长的内存对象，还有个Permanent Generation，主要用来放JVM自己的反射对象，比如类对象和方法对象等。

程序计数器

如同其名称一样。程序计数器用于记录某个线程下次执行指令位置。程序计数器也是线程私有的。

并发内存模型



java试图定义一个Java内存模型(Java memory model jmm)来屏蔽掉各种硬件/操作系统的内存访问差异，以实现让java程序在各个平台下都能达到一致的内存访问效果。java内存模型主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。模型图如下：

img

java并发内存模型以及内存操作规则

java内存模型中规定了所有变量都存贮到主内存（如虚拟机物理内存中的一部分）中。每一个线程都有一个自己的工作内存(如cpu中的高速缓存)。线程中的工作内存保存了该线程使用到的变量的主内存的副本拷贝。线程对变量的所有操作（读取、赋值等）必须在该线程的工作内存中进行。不同线程之间无法直接访问对方工作内存中变量。线程间变量的值传递均需要通过主内存来完成。

关于主内存与工作内存之间的交互协议，即一个变量如何从主内存拷贝到工作内存。如何从工作内存同步到主内存中的实现细节。java内存模型定义了8种操作来完成。这8种操作每一种都是原子操作。8种操作如下：

- lock(锁定)：作用于主内存，它把一个变量标记为一条线程独占状态；
- unlock(解锁)：作用于主内存，它将一个处于锁定状态的变量释放出来，释放后的变量才能够被其他线程锁定；
- read(读取)：作用于主内存，它把变量值从主内存传送到线程的工作内存中，以便随后的load动作使用；
- load(载入)：作用于工作内存，它把read操作的值放入工作内存中的变量副本中；
- use(使用)：作用于工作内存，它把工作内存中的值传递给执行引擎，每当虚拟机遇到一个需要使用这个变量的指令时候，将会执行这个动作；
- assign(赋值)：作用于工作内存，它把从执行引擎获取的值赋值给工作内存中的变量，每当虚拟机遇到一个给变量赋值的指令时候，执行该操作；
- store(存储)：作用于工作内存，它把工作内存中的一个变量传送给主内存中，以备随后的write操作使用；
- write(写入)：作用于主内存，它把store传送值放到主内存中的变量中。

Java内存模型还规定了执行上述8种基本操作时必须满足如下规则：

- 不允许read和load、store和write操作之一单独出现，以上两个操作必须按顺序执行，但没有保证必须连续执行，也就是说，read与load之间、store与write之间是可插入其他指令的。
- 不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。

^

+

□

⌘

- 不允许一个线程无原因地（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存中。
- 一个新的变量只能从主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量，换句话说就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。
- 一个变量在同一个时刻只允许一条线程对其执行lock操作，但lock操作可以被同一个线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。
- 如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。
- 如果一个变量实现没有被lock操作锁定，则不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定的变量。
- 对一个变量执行unlock操作之前，必须先把此变量同步回主内存（执行store和write操作）。

volatile型变量的特殊规则

关键字volatile可以说是Java虚拟机提供的最轻量级的同步机制，但是它并不容易完全被正确、完整的理解，以至于许多程序员都不习惯去使用它，遇到需要处理多线程的问题的时候一律使用synchronized来进行同步。了解volatile变量的语义对后面了解多线程操作的其他特性很有意义。Java内存模型对volatile专门定义了一些特殊的访问规则，当一个变量被定义成volatile之后，他将具备两种特性：

- 保证此变量对所有线程的可见性。第一保证此变量对所有线程的可见性，这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。而普通变量是做不到这点，普通变量的值在线程在线程间传递均需要通过主内存来完成，例如，线程A修改一个普通变量的值，然后向主内存进行会写，另外一个线程B在线程A回写完成了之后再从主内存进行读取操作，新变量值才会对线程B可见。另外，java里面的运算并非原子操作，会导致volatile变量的运算在并发下一样是不安全的。
- 禁止指令重排序优化。普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获得正确的结果，而不能保证变量赋值操作的顺序与程序中的执行顺序一致，在单线程中，我们是无法感知这一点的。

由于volatile变量只能保证可见性，在不符合以下两条规则的运算场景中，我们仍然要通过加锁来保证原子性。

- 1.运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。
- 2.变量不需要与其他的状态比阿尼浪共同参与不变约束。

原子性、可见性与有序性

Java内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性这三个特征来建立的，我们逐个看下哪些操作实现了这三个特性。

- **原子性（Atomicity）**：由Java内存模型来直接保证的原子性变量包括read、load、assign、use、store和write，我们大致可以认为基本数据类型的访问读写是具备原子性的。如果应用场景需要一个更大方位的原子性保证，Java内存模型还提供了lock和unlock操作来满足这种需求，尽管虚拟机未把lock和unlock操作直接开放给用户使用，但是却提供了更高层次的字节码指令monitorenter和monitorexit来隐式的使用这两个操作，这两个字节码指令反应到Java代码中就是同步块--synchronized关键字，因此在synchronized块之间的操作也具备原子性。



- **可见性 (Visibility)**：可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。上文在讲解volatile变量的时候我们已详细讨论过这一点。Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方式来实现可见性的，无论是普通变量还是volatile变量都是如此，普通变量与volatile变量的区别是，volatile的特殊规则保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。因此，可以说volatile保证了多线程操作时变量的可见性，而普通变量则不能保证这一点。除了volatile之外，Java还有两个关键字能实现可见性，即synchronized和final.同步快的可见性是由“对一个变量执行unlock操作前，必须先要把此变量同步回主内存”这条规则获得的，而final关键字的可见性是指：被final修饰的字段在构造器中一旦初始化完成，并且构造器没有把“this”的引用传递出去，那么在其他线程中就能看见final字段的值。
- **有序性 (Ordering)**：Java内存模型的有序性在前面讲解volatile时也详细的讨论过了，Java程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有的操作都是有序的：如果在一个线程中观察另外一个线程，所有的线程操作都是无序的。前半句是指“线程内表现为串行的语义”，后半句是指“指令重排序”现象和“工作内存与主内存同步延迟”现象。Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由“一个变量在同一个时刻只允许一条线程对其进行lock操作”这条规则获得的，这条规则决定了持有同一个锁的两个同步块只能串行的进入。

Arrays和Collections 对于sort的不同实现原理

1、Arrays.sort()

该算法是一个经过调优的快速排序，此算法在很多数据集上提供 $N \cdot \log(N)$ 的性能，这导致其他快速排序会降低二次型性能。

2、Collections.sort()

该算法是一个经过修改的合并排序算法（其中，如果低子列表中的最高元素效益高子列表中的最低元素，则忽略合并）。此算法可提供保证的 $N \cdot \log(N)$ 的性能，此实现将指定列表转储到一个数组中，然后再对数组进行排序，在重置数组中相应位置处每个元素的列表上进行迭代。这避免了由于试图原地对链接列表进行排序而产生的 $n^2 \log(n)$ 性能。

Java中object常用方法

- 1、clone()
- 2、equals()
- 3、finalize()
- 4、getClass()
- 5、hashCode()
- 6、notify()
- 7、notifyAll()
- 8、toString()

对于Java中多态的理解



所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。因为在程序运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

Java实现多态有三个必要条件：继承、重写、父类引用指向子类对象。

继承：在多态中必须存在有继承关系的子类 and 父类。

重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。

父类引用指向子类对象：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备技能调用父类的方法和子类的方法。

实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

多态的作用：消除类型之间的耦合关系。

Java序列化与反序列化是什么？为什么需要序列化与反序列化？如何实现Java序列化与反序列化

spring AOP 实现原理



什么是AOP

AOP（Aspect-OrientedProgramming，面向方面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需
要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

而AOP技术则恰恰相反，它利用一种称为“横切”的技术，剖解封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天工的妙手将这些剖开的切面复原，不留痕迹。

使用“横切”技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。正如Avanade公司的高级方案构架师Adam Magee所说，AOP的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

实现AOP的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

AOP使用场景

AOP用来封装横切关注点，具体可以在下面的场景中使用：

Authentication 权限

Caching 缓存

Context passing 内容传递

Error handling 错误处理

Lazy loading 懒加载

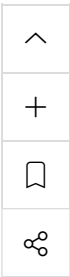
Debugging 调试

logging, tracing, profiling and monitoring 记录跟踪 优化 校准

Performance optimization 性能优化

Persistence 持久化

Resource pooling 资源池



AOP相关概念

方面（Aspect）：一个关注点的模块化，这个关注点实现可能另外横切多个对象。事务管理是J2EE应用中一个很好的横切关注点例子。方面用spring (<https://link.jianshu.com?t=http://lib.csdn.net/base/javaee>)的 Advisor或拦截器实现。

连接点（Joinpoint）：程序执行过程中明确的点，如方法的调用或特定的异常被抛出。

通知（Advice）：在特定的连接点，AOP框架执行的动作。各种类型的通知包括“around”、“before”和“throws”通知。通知类型将在下面讨论。许多AOP框架包括Spring都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器链。Spring中定义了四个advice: BeforeAdvice, AfterAdvice, ThrowAdvice和DynamicIntroductionAdvice

切入点（Pointcut）：指定一个通知将被引发的一系列连接点的集合。AOP框架必须允许开发者指定切入点：例如，使用正则表达式。Spring定义了Pointcut接口，用来组合MethodMatcher和ClassFilter，可以通过名字很清楚的理解，MethodMatcher是用来检查目标类的方法是否可以被应用此通知，而ClassFilter是用来检查Pointcut是否应该应用到目标类上

引入（Introduction）：添加方法或字段到被通知的类。Spring允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现 IsModified接口，来简化缓存。Spring中要使用Introduction, 可有通过 DelegatingIntroductionInterceptor来实现通知，通过DefaultIntroductionAdvisor来配置Advice和代理类要实现的接口

目标对象（Target Object）：包含连接点的对象。也被称作被通知或被代理对象。POJO

AOP代理（AOP Proxy）：AOP框架创建的对象，包含通知。在Spring中，AOP代理可以是JDK动态代理或者CGLIB代理。

织入（Weaving）：组装方面来创建一个被通知对象。这可以在编译时完成（例如使用AspectJ编译器），也可以在运行时完成。Spring和其他纯Java (<https://link.jianshu.com?t=http://lib.csdn.net/base/java>)AOP框架一样，在运行时完成织入。

Spring AOP组件

下面这种类图列出了Spring中主要的AOP组件



img

如何使用Spring AOP

可以通过配置文件或者编程的方式来使用Spring AOP。

配置可以通过xml文件来进行，大概有四种方式：

1. 配置ProxyFactoryBean，显式地设置advisors, advice, target等

2. 配置AutoProxyCreator，这种方式下，还是如以前一样使用定义的bean，但是从容器中获得

3. 通过<aop:config>来配置

4. 通过<aop:aspectj-autoproxy>来配置，使用AspectJ的注解来标识通知及切入点

也可以直接使用ProxyFactory来以编程的方式使用Spring AOP，通过ProxyFactory提供的方法可以设置target对象, advisor等相关配置，最终通过getProxy()方法来获取代理对象

具体使用的示例可以google. 这里略去

Spring AOP代理对象的生成

Spring提供了两种方式来生成代理对象: JDKProxy和Cglib，具体使用哪种方式生成由AopProxyFactory根据AdvisedSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。下面我们来研究一下Spring如何使用JDK来生成代理对象，具体的生成代码放在JdkDynamicAopProxy这个类中，直接上相关代码：

友情链接：Spring AOP 实现原理 (<https://link.jianshu.com?t=http://blog.csdn.net/moreevan/article/details/11977115/>)

```
/**
 * <ol>
 * <li>获取代理类要实现的接口,除了Advised对象中配置的,还会加上SpringProxy, Advised(
 * <li>检查上面得到的接口中有没有定义 equals或者hashcode的接口
 * <li>调用Proxy.newProxyInstance创建代理对象
 * </ol>
 */
public Object getProxy(ClassLoader classLoader) {
    if (logger.isDebugEnabled()) {
        logger.debug("Creating JDK dynamic proxy: target source is " +this.
    }
    Class[] proxiedInterfaces =AopProxyUtils.completeProxiedInterfaces(this
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
    return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
}
```

那这个其实很明了，注释上我也已经写清楚了，不再赘述。

下面的问题是，代理对象生成了，那切面是如何织入的？



我们知道InvocationHandler是JDK动态代理的核心，生成的代理对象的方法调用都会委托到InvocationHandler.invoke()方法。而通过JdkDynamicAopProxy的签名我们可以看到这个类其实也实现了InvocationHandler，下面我们就通过分析这个类中实现的invoke()方法来具体看下Spring AOP是如何织入切面的。



Servlet 工作原理

Servlet 工作原理解析

从 Servlet 容器说起

前面说了 Servlet 容器作为一个独立发展的标准化产品，目前它的种类很多，但是它们都有自己的市场定位，很难说谁优谁劣，各有特点。例如现在比较流行的 Jetty，在定制化和移动领域有不错的发展，我们这里还是以大家最为熟悉 Tomcat 为例来介绍 Servlet 容器如何管理 Servlet。Tomcat 本身也很复杂，我们只从 Servlet 与 Servlet 容器的接口部分开始介绍，关于 Tomcat 的详细介绍可以参考我的另外一篇文章《Tomcat 系统架构与模式设计分析》。

Tomcat 的容器等级中，Context 容器是直接管理 Servlet 在容器中的包装类 Wrapper，所以 Context 容器如何运行将直接影响 Servlet 的工作方式。

图 1 . Tomcat 容器模型

从上图可以看出 Tomcat 的容器分为四个等级，真正管理 Servlet 的容器是 Context 容器，一个 Context 对应一个 Web 工程，在 Tomcat 的配置文件中可以很容易发现这一点，如下：

清单 1 Context 配置参数

```
<Context path="/projectOne " docBase="D:\projects\projectOne"
reloadable="true" />
```

下面详细介绍一下 Tomcat 解析 Context 容器的过程，包括如何构建 Servlet 的过程。

Servlet 容器的启动过程

Tomcat7 也开始支持嵌入式功能，增加了一个启动类 org.apache.catalina.startup.Tomcat。创建一个实例对象并调用 start 方法就可以很容易启动 Tomcat，我们还可以通过这个对象来增加和修改 Tomcat 的配置参数，如可以动态增加 Context、Servlet 等。下面我们就利用这个 Tomcat 类来管理新增的一个 Context 容器，我们就选择 Tomcat7 自带的 examples Web 工程，并看看它是如何加到这个 Context 容器中的。

清单 2 . 给 Tomcat 增加一个 Web 工程

```
Tomcat tomcat = getTomcatInstance();
File appDir = new File(getBuildDirectory(), "webapps/examples");
tomcat.addWebapp(null, "/examples", appDir.getAbsolutePath());
tomcat.start();
ByteChunk res = getUrl("http://localhost:" + getPort() +
    "/examples/servlets/servlet/HelloWorldExample");
assertTrue(res.toString().indexOf("<h1>Hello World!</h1>") > 0);
```

清单 1 的代码是创建一个 Tomcat 实例并新增一个 Web 应用，然后启动 Tomcat 并调用其中的一个 HelloWorldExample Servlet，看有没有正确返回预期的数据。

Tomcat 的 addWebapp 方法的代码如下：

清单 3 .Tomcat.addWebapp



```

public Context addWebapp(Host host, String url, String path) {
    silence(url);
    Context ctx = new StandardContext();
    ctx.setPath( url );
    ctx.setDocBase(path);
    if (defaultRealm == null) {
        initSimpleAuth();
    }
    ctx.setRealm(defaultRealm);
    ctx.addLifecycleListener(new DefaultWebXmlListener());
    ContextConfig ctxCfg = new ContextConfig();
    ctx.addLifecycleListener(ctxCfg);
    ctxCfg.setDefaultWebXml("org/apache/catalin/startup/NO_DEFAULT_XML");
    if (host == null) {
        getHost().addChild(ctx);
    } else {
        host.addChild(ctx);
    }
    return ctx;
}

```

前面已经介绍了一个 Web 应用对应一个 Context 容器，也就是 Servlet 运行时的 Servlet 容器，添加一个 Web 应用时将会创建一个 StandardContext 容器，并且给这个 Context 容器设置必要的参数，url 和 path 分别代表这个应用在 Tomcat 中的访问路径和这个应用实际的物理路径，这个两个参数与清单 1 中的两个参数是一致的。其中最重要的一个配置是 ContextConfig，这个类将会负责整个 Web 应用配置的解析工作，后面将会详细介绍。最后将这个 Context 容器加到父容器 Host 中。

接下去将会调用 Tomcat 的 start 方法启动 Tomcat，如果你清楚 Tomcat 的系统架构，你会容易理解 Tomcat 的启动逻辑，Tomcat 的启动逻辑是基于观察者模式设计的，所有的容器都会继承 Lifecycle 接口，它管理者容器的整个生命周期，所有容器的修改和状态的改变都会由它去通知已经注册的观察者（Listener），关于这个设计模式可以参考《Tomcat 的系统架构与设计模式，第二部分：设计模式》。Tomcat 启动的时序图可以用图 2 表示。

图 2. Tomcat 主要类的启动时序图（查看大图

([https://link.jianshu.com?](https://link.jianshu.com?t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/image003.jpg)

[t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/image003.jpg](http://www.ibm.com/developerworks/cn/java/j-lo-servlet/image003.jpg)))

上图描述了 Tomcat 启动过程中，主要类之间的时序关系，下面我们将会重点关注添加 examples 应用所对应的 StandardContext 容器的启动过程。

当 Context 容器初始化状态设为 init 时，添加在 Context 容器的 Listener 将会被调用。ContextConfig 继承了 LifecycleListener 接口，它是在调用清单 3 时被加入到 StandardContext 容器中。ContextConfig 类会负责整个 Web 应用的配置文件的解析工作。

ContextConfig 的 init 方法将会主要完成以下工作：

1. 创建用于解析 xml 配置文件的 contextDigester 对象
2. 读取默认 context.xml 配置文件，如果存在解析它
3. 读取默认 Host 配置文件，如果存在解析它
4. 读取默认 Context 自身的配置文件，如果存在解析它
5. 设置 Context 的 DocBase

ContextConfig 的 init 方法完成后，Context 容器的会执行 startInternal 方法，这个方法启动逻辑比较复杂，主要包括如下几个部分：

1. 创建读取资源文件的对象
2. 创建 ClassLoader 对象
3. 设置应用的工作目录
4. 启动相关的辅助类如：logger、realm、resources 等



5. 修改启动状态，通知感兴趣的观察者（Web 应用的配置）
6. 子容器的初始化
7. 获取 ServletContext 并设置必要的参数
8. 初始化“load on startup”的 Servlet

Web 应用的初始化工作

Web 应用的初始化工作是在 ContextConfig 的 configureStart 方法中实现的，应用的初始化主要是要解析 web.xml 文件，这个文件描述了一个 Web 应用的关键信息，也是一个 Web 应用的入口。

Tomcat 首先会找 globalWebXml 这个文件的搜索路径是在 engine 的工作目录下寻找以下两个文件中的任一个 org/apache/catalin/startup/NO_DEFAULT_XML 或 conf/web.xml。接着会找 hostWebXml 这个文件可能会在 System.getProperty("catalina.base")/conf/\${EngineName}/\${HostName}/web.xml.default，接着寻找应用的配置文件 examples/WEB-INF/web.xml。web.xml 文件中的各个配置项将会被解析成相应的属性保存在 WebXml 对象中。如果当前应用支持 Servlet3.0，解析还将完成额外 9 项工作，这个额外的 9 项工作主要是为 Servlet3.0 新增的特性，包括 jar 包中的 META-INF/web-fragment.xml 的解析以及对 annotations 的支持。

接下去将会将 WebXml 对象中的属性设置到 Context 容器中，这里包括创建 Servlet 对象、filter、listener 等等。这段代码在 WebXml 的 configureContext 方法中。下面是解析 Servlet 的代码片段：

清单 4. 创建 Wrapper 实例



```

for (ServletDef servlet : servlets.values()) {
    Wrapper wrapper = context.createWrapper();
    String jspFile = servlet.getJspFile();
    if (jspFile != null) {
        wrapper.setJspFile(jspFile);
    }
    if (servlet.getLoadOnStartup() != null) {
        wrapper.setLoadOnStartup(servlet.getLoadOnStartup().intValue())
    }
    if (servlet.getEnabled() != null) {
        wrapper.setEnabled(servlet.getEnabled().booleanValue());
    }
    wrapper.setName(servlet.getServletName());
    Map<String,String> params = servlet.getParameterMap();
    for (Entry<String, String> entry : params.entrySet()) {
        wrapper.addInitParameter(entry.getKey(), entry.getValue());
    }
    wrapper.setRunAs(servlet.getRunAs());
    Set<SecurityRoleRef> roleRefs = servlet.getSecurityRoleRefs();
    for (SecurityRoleRef roleRef : roleRefs) {
        wrapper.addSecurityReference(
            roleRef.getName(), roleRef.getLink());
    }
    wrapper.setServletClass(servlet.getServletClass());
    MultipartDef multipartdef = servlet.getMultipartDef();
    if (multipartdef != null) {
        if (multipartdef.getMaxFileSize() != null &&
            multipartdef.getMaxRequestSize() != null &&
            multipartdef.getFileSizeThreshold() != null) {
            wrapper.setMultipartConfigElement(new
MultipartConfigElement(
                multipartdef.getLocation(),
                Long.parseLong(multipartdef.getMaxFileSize()),
                Long.parseLong(multipartdef.getMaxRequestSize()),
                Integer.parseInt(
                    multipartdef.getFileSizeThreshold()));
        } else {
            wrapper.setMultipartConfigElement(new
MultipartConfigElement(
                multipartdef.getLocation()));
        }
    }
    if (servlet.getAsyncSupported() != null) {
        wrapper.setAsyncSupported(
            servlet.getAsyncSupported().booleanValue());
    }
    context.addChild(wrapper);
}

```

这段代码清楚的描述了如何将 Servlet 包装成 Context 容器中的 StandardWrapper，这里有个疑问，为什么要将 Servlet 包装成 StandardWrapper 而不直接是 Servlet 对象。这里 StandardWrapper 是 Tomcat 容器中的一部分，它具有容器的特征，而 Servlet 为了一个独立的 web 开发标准，不应该强耦合在 Tomcat 中。

除了将 Servlet 包装成 StandardWrapper 并作为子容器添加到 Context 中，其它的所有 web.xml 属性都被解析到 Context 中，所以说 Context 容器才是真正运行 Servlet 的 Servlet 容器。一个 Web 应用对应一个 Context 容器，容器的配置属性由应用的 web.xml 指定，这样我们就能理解 web.xml 到底起到什么作用了。

回页首 (<https://link.jianshu.com?t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/#ibm-pcon>)

创建 Servlet 实例

前面已经完成了 Servlet 的解析工作，并且被包装成 StandardWrapper 添加在 Context 容器中，但是它仍然不能为我们工作，它还没有被实例化。下面我们将介绍 Servlet 对象是如何创建的，以及如何被初始化的。

创建 Servlet 对象



如果 Servlet 的 load-on-startup 配置项大于 0，那么在 Context 容器启动的时候就会被实例化，前面提到在解析配置文件时会读取默认的 globalWebXml，在 conf 下的 web.xml 文件中定义了一些默认的配置项，其定义了两个 Servlet，分别是：org.apache.catalina.servlets.DefaultServlet 和 org.apache.jasper.servlet.JspServlet 它们的 load-on-startup 分别是 1 和 3，也就是当 Tomcat 启动时这两个 Servlet 就会被启动。

创建 Servlet 实例的方法是从 Wrapper.loadServlet 开始的。loadServlet 方法要完成的的就是获取 servletClass 然后把它交给 InstanceManager 去创建一个基于 servletClass.class 的对象。如果这个 Servlet 配置了 jsp-file，那么这个 servletClass 就是 conf/web.xml 中定义的 org.apache.jasper.servlet.JspServlet 了。

创建 Servlet 对象的相关类结构图如下：

图 3. 创建 Servlet 对象的相关类结构

初始化 Servlet

初始化 Servlet 在 StandardWrapper 的 initServlet 方法中，这个方法很简单就是调用 Servlet 的 init 的方法，同时把包装了 StandardWrapper 对象的 StandardWrapperFacade 作为 ServletConfig 传给 Servlet。Tomcat 容器为何要传 StandardWrapperFacade 给 Servlet 对象将在后面做详细解析。

如果该 Servlet 关联的是一个 jsp 文件，那么前面初始化的就是 JspServlet，接下去会模拟一次简单请求，请求调用这个 jsp 文件，以便编译这个 jsp 文件为 class，并初始化这个 class。

这样 Servlet 对象就初始化完成了，事实上 Servlet 从被 web.xml 中解析到完成初始化，这个过程非常复杂，中间有很多过程，包括各种容器状态的转化引起的监听事件的触发、各种访问权限的控制和一些不可预料的错误发生的判断行为等等。我们这里只抓了一些关键环节进行阐述，试图让大家有个总体脉络。

下面是这个过程的一个完整的时序图，其中也省略了一些细节。

图 4. 初始化 Servlet 的时序图（查看大图 (<https://link.jianshu.com?t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/image007.jpg>)

回首页 (<https://link.jianshu.com?t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/#ibm-pcon>)

Servlet 体系结构

我们知道 Java Web 应用是基于 Servlet 规范运转的，那么 Servlet 本身又是如何运转的呢？为何要设计这样的体系结构。

图 5.Servlet 顶层类关联图

从上图可以看出 Servlet 规范就是基于这几个类运转的，与 Servlet 主动关联的是三个类，分别是 ServletConfig、ServletRequest 和 ServletResponse。这三个类都是通过容器传递给 Servlet 的，其中 ServletConfig 是在 Servlet 初始化时就传给 Servlet 了，而后两个是在请求达到时调用 Servlet 时传递过来的。我们很清楚 ServletRequest 和 ServletResponse 在 Servlet 运行的意义，但是 ServletConfig 和 ServletContext 对 Servlet 有何价值？仔细查看 ServletConfig 接口中声明的方法发现，这些方法都是为了获取这个 Servlet 的一些配置属性，而这些配置属性可能在 Servlet 运行时被用到。而 ServletContext 又是干什么的呢？Servlet 的运行模式是一个典型的“握手型的交互式”运行模式。所谓“握手型的交互式”就是两个模块为了交换数据通常都会准备一个交易场景，这个场景一直跟随这个交易过程



直到这个交易完成为止。这个交易场景的初始化是根据这次交易对象指定的参数来定制的，这些指定参数通常就会是一个配置类。所以对号入座，交易场景就由 ServletContext 来描述，而定制的参数集合就由 ServletConfig 来描述。而 ServletRequest 和 ServletResponse 就是要交互的具体对象了，它们通常都是作为运输工具来传递交互结果。

ServletConfig 是在 Servlet init 时由容器传过来的，那么 ServletConfig 到底是个什么对象呢？

下图是 ServletConfig 和 ServletContext 在 Tomcat 容器中的类关系图。

图 6. ServletConfig 在容器中的类关联图

上图可以看出 StandardWrapper 和 StandardWrapperFacade 都实现了 ServletConfig 接口，而 StandardWrapperFacade 是 StandardWrapper 门面类。所以传给 Servlet 的是 StandardWrapperFacade 对象，这个类能够保证从 StandardWrapper 中拿到 ServletConfig 所规定的的数据，而又不把 ServletConfig 不关心的数据暴露给 Servlet。

同样 ServletContext 也与 ServletConfig 有类似的结构，Servlet 中能拿到的 ServletContext 的实际对象也是 ApplicationContextFacade 对象。ApplicationContextFacade 同样保证 ServletContext 只能从容器中拿到它该拿的数据，它们都起到对数据的封装作用，它们使用的都是门面设计模式。

通过 ServletContext 可以拿到 Context 容器中一些必要信息，比如应用的工作路径，容器支持的 Servlet 最小版本等。

Servlet 中定义的两个 ServletRequest 和 ServletResponse 它们实际的对象又是什么呢？，我们在创建自己的 Servlet 类时通常使用的都是 HttpServletRequest 和 HttpServletResponse，它们继承了 ServletRequest 和 ServletResponse。为何 Context 容器传过来的 ServletRequest、ServletResponse 可以被转化为 HttpServletRequest 和 HttpServletResponse 呢？

图 7.Request 相关类结构图

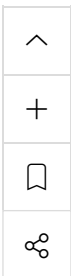
上图是 Tomcat 创建的 Request 和 Response 的类结构图。Tomcat 一接受到请求首先将会创建 org.apache.coyote.Request 和 org.apache.coyote.Response，这两个类是 Tomcat 内部使用的描述一次请求和相应的信息类它们是一个轻量级的类，它们作用就是在服务器接收到请求后，经过简单解析将这个请求快速的分配给后续线程去处理，所以它们的对象很小，很容易被 JVM 回收。接下去当交给一个用户线程去处理这个请求时又创建 org.apache.catalina.connector.Request 和 org.apache.catalina.connector.Response 对象。这两个对象一直穿越整个 Servlet 容器直到要传给 Servlet，传给 Servlet 的是 Request 和 Response 的门面类 RequestFacade 和 ResponseFacade，这里使用门面模式与前面一样都是基于同样的目的——封装容器中的数据。一次请求对应的 Request 和 Response 的类转化如下图所示：

图 8.Request 和 Response 的转变过程

回页首 (<https://link.jianshu.com?t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/#ibm-pcon>)

Servlet 如何工作

我们已经清楚了 Servlet 是如何被加载的、Servlet 是如何被初始化的，以及 Servlet 的体系结构，现在的问题就是它是如何被调用的。



当用户从浏览器向服务器发起一个请求，通常会包含如下信息：`http://hostname (https://link.jianshu.com?t=http://hostname): port /contextpath/servletpath`，`hostname` 和 `port` 是用来与服务器建立 TCP 连接，而后的 URL 才是用来选择服务器中那个子容器服务用户的请求。那服务器是如何根据这个 URL 来达到正确的 Servlet 容器中的呢？

Tomcat7.0 中这件事很容易解决，因为这种映射工作有专门一个类来完成的，这个就是 `org.apache.tomcat.util.http.mapper`，这个类保存了 Tomcat 的 Container 容器中的所有子容器的信息，当 `org.apache.catalina.connector.Request` 类在进入 Container 容器之前，`mapper` 将会根据这次请求的 `hostname` 和 `contextpath` 将 `host` 和 `context` 容器设置到 `Request` 的 `mappingData` 属性中。所以当 `Request` 进入 Container 容器之前，它要访问那个子容器这时就已经确定了。

图 9.Request 的 Mapper 类关系图

可能你有疑问，`mapper` 中怎么会有容器的完整关系，这要回到图 2 中 19 步 `MapperListener` 类的初始化过程，下面是 `MapperListener` 的 `init` 方法代码：

清单 5. MapperListener.init

```
public void init() {
    findDefaultHost();
    Engine engine = (Engine) connector.getService().getContainer();
    engine.addContainerListener(this);
    Container[] conHosts = engine.findChildren();
    for (Container conHost : conHosts) {
        Host host = (Host) conHost;
        if (!LifecycleState.NEW.equals(host.getState())) {
            host.addLifecycleListener(this);
            registerHost(host);
        }
    }
}
```

这段代码的作用就是将 `MapperListener` 类作为一个监听者加到整个 Container 容器中的每个子容器中，这样只要任何一个容器发生变化，`MapperListener` 都将会被通知，相应的保存容器关系的 `MapperListener` 的 `mapper` 属性也会修改。for 循环中就是将 `host` 及下面的子容器注册到 `mapper` 中。

图 10.Request 在容器中的路由图

上图描述了一次 Request 请求是如何达到最终的 Wrapper 容器的，我们现正知道了请求是如何达到正确的 Wrapper 容器，但是请求到达最终的 Servlet 还要完成一些步骤，必须要执行 Filter 链，以及要通知你在 `web.xml` 中定义的 listener。

接下去就要执行 Servlet 的 `service` 方法了，通常情况下，我们自己定义的 `servlet` 并不是直接去实现 `javax.servlet.servlet` 接口，而是去继承更简单的 `HttpServlet` 类或者 `GenericServlet` 类，我们可以有选择的覆盖相应方法去实现我们要完成的工作。

Servlet 的确已经能够帮我们完成所有的工作了，但是现在的 web 应用很少有直接将交互全部页面都用 `servlet` 来实现，而是采用更加高效的 MVC 框架来实现。这些 MVC 框架基本的原理都是将所有的请求都映射到一个 Servlet，然后去实现 `service` 方法，这个方法也就是 MVC 框架的入口。

当 Servlet 从 Servlet 容器中移除时，也就表明该 Servlet 的生命周期结束了，这时 Servlet 的 `destroy` 方法将被调用，做一些扫尾工作。

回页首 (<https://link.jianshu.com?t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/#ibm-pcon>)

Session 与 Cookie



前面我们已经说明了 Servlet 如何被调用，我们基于 Servlet 来构建应用程序，那么我们能从 Servlet 获得哪些数据信息呢？

Servlet 能够给我们提供两部分数据，一个是在 Servlet 初始化时调用 init 方法时设置的 ServletConfig，这个类基本上含有了 Servlet 本身和 Servlet 所运行的 Servlet 容器中的基本信息。根据前面的介绍 ServletConfig 的实际对象是 StandardWrapperFacade，到底能获得哪些容器信息可以看看这类提供了哪些接口。还有一部分数据是由 ServletRequest 类提供，它的实际对象是 RequestFacade，从提供的方法中发现主要是描述这次请求的 HTTP 协议的信息。所以要掌握 Servlet 的工作方式必须要很清楚 HTTP 协议，如果你还不清楚赶紧去找一些参考资料。关于这一块还有一个让很多人迷惑的 Session 与 Cookie。

Session 与 Cookie 不管是对 Java Web 的熟练使用者还是初学者来说都是一个令人头疼的东西。Session 与 Cookie 的作用都是为了保持访问用户与后端服务器的交互状态。它们有各自的优点也有各自的缺陷。然而具有讽刺意味的是它们优点和它们的使用场景又是矛盾的，例如使用 Cookie 来传递信息时，随着 Cookie 个数的增多和访问量的增加，它占用的网络带宽也很大，试想假如 Cookie 占用 200 个字节，如果一天的 PV 有几亿的时候，它要占用多少带宽。所以大访问量的时候希望用 Session，但是 Session 的致命弱点是不容易在多台服务器之间共享，所以这也限制了 Session 的使用。

不管 Session 和 Cookie 有什么不足，我们还是要用它们。下面详细讲一下，Session 如何基于 Cookie 来工作。实际上有三种方式能可以让 Session 正常工作：

1. 基于 URL Path Parameter，默认就支持
2. 基于 Cookie，如果你没有修改 Context 容器个 cookies 标识的话，默认也是支持的
3. 基于 SSL，默认不支持，只有 connector.getAttribute("SSLEnabled") 为 TRUE 时才支持

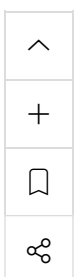
第一种情况下，当浏览器不支持 Cookie 功能时，浏览器会将用户的 SessionCookieName 重写到用户请求的 URL 参数中，它的传递格式如 /path/Servlet;name=value;name2=value2? Name3=value3，其中“Servlet;”后面的 K-V 对就是要传递的 Path Parameters，服务器会从这个 Path Parameters 中拿到用户配置的 SessionCookieName。关于这个 SessionCookieName，如果你在 web.xml 中配置 session-config 配置项的话，其 cookie-config 下的 name 属性就是这个 SessionCookieName 值，如果你没有配置 session-config 配置项，默认的 SessionCookieName 就是大家熟悉的“JSESSIONID”。接着 Request 根据这个 SessionCookieName 到 Parameters 拿到 Session ID 并设置到 request.setRequestedSessionId 中。

请注意如果客户端也支持 Cookie 的话，Tomcat 仍然会解析 Cookie 中的 Session ID，并会覆盖 URL 中的 Session ID。

如果是第三种情况的话将会根据 javax.servlet.request.ssl_session 属性值设置 Session ID。

有了 Session ID 服务器端就可以创建 HttpSession 对象了，第一次触发是通过 request.getSession() 方法，如果当前的 Session ID 还没有对应的 HttpSession 对象那么就创建一个新的，并将这个对象加到 org.apache.catalina.Manager 的 sessions 容器中保存，Manager 类将管理所有 Session 的生命周期，Session 过期将被回收，服务器关闭，Session 将被序列化到磁盘等。只要这个 HttpSession 对象存在，用户就可以根据 Session ID 来获取到这个对象，也就达到了状态的保持。

图 11.Session 相关类图



上从图中可以看出从 request.getSession 中获取的 HttpSession 对象实际上是 StandardSession 对象的门面对象，这与前面的 Request 和 Servlet 是一样的原理。下图是 Session 工作的时序图：

图 12.Session 工作的时序图（查看大图 (<https://link.jianshu.com?t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/image023.jpg>)

还有一点与 Session 关联的 Cookie 与其它 Cookie 没有什么不同，这个配置的配置可以通过 web.xml 中的 session-config 配置项来指定。

回首页 (<https://link.jianshu.com?t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/#ibm-pcon>)

Servlet 中的 Listener

整个 Tomcat 服务器中 Listener 使用的非常广泛，它是基于观察者模式设计的，Listener 的设计对开发 Servlet 应用程序提供了一种快捷的手段，能够方便的从另一个纵向维度控制程序和数据。目前 Servlet 中提供了 5 种两类事件的观察者接口，它们分别是：4 个 EventListeners 类型的，ServletContextAttributeListener、ServletRequestAttributeListener、ServletRequestListener、HttpSessionAttributeListener 和 2 个 LifecycleListeners 类型的，ServletContextListener、HttpSessionListener。如下图所示：

图 13.Servlet 中的 Listener（查看大图 (<https://link.jianshu.com?t=http://www.ibm.com/developerworks/cn/java/j-lo-servlet/image025.png>)

它们基本上涵盖了整个 Servlet 生命周期中，你感兴趣的每种事件。这些 Listener 的实现类可以配置在 web.xml 中的 <listener> 标签中。当然也可以在应用程序中动态添加 Listener，需要注意的是 ServletContextListener 在容器启动之后就不能再添加新的，因为它所监听的事件已经不会再出现。掌握这些 Listener 的使用，能够让我们的程序设计的更加灵活

Java NIO和IO的区别



下表总结了Java NIO和IO之间的主要差别，我会更详细地描述表中每部分的差异。

复制代码 (https://link.jianshu.com?t=undefined)代码如下:

IO NIO

面向流 面向缓冲

阻塞IO 非阻塞IO

无 选择器

面向流与面向缓冲

Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

阻塞与非阻塞IO

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么也不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

选择器（Selectors）

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

NIO和IO如何影响应用程序的设计

无论您选择IO或NIO工具箱，可能会影响您应用程序设计的以下几个方面：

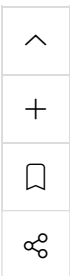
- 1.对NIO或IO类的API调用。
- 2.数据处理。
- 3.用来处理数据的线程数。

API调用

当然，使用NIO的API调用时看起来与使用IO时有所不同，但这并不意外，因为并不是仅从一个InputStream逐字节读取，而是数据必须先读入缓冲区再处理。

数据处理

使用纯粹的NIO设计相较IO设计，数据处理也受到影响。



在IO设计中，我们从InputStream或 Reader逐字节读取数据。假设你正在处理一基于行的文本数据流，例如：

复制代码 (https://link.jianshu.com?t=undefined)代码如下：

```
Name: Anna
Age: 25
Email: anna@mailserver.com (https://link.jianshu.com?t=mailto:anna@mailserver.com)
Phone: 1234567890
```

该文本行的流可以这样处理：

复制代码 (https://link.jianshu.com?t=undefined)代码如下：

```
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String nameLine = reader.readLine();
String ageLine = reader.readLine();
String emailLine = reader.readLine();
String phoneLine = reader.readLine();
```

请注意处理状态由程序执行多久决定。换句话说，一旦reader.readLine()方法返回，你就知道肯定文本行就已读完，readline()阻塞直到整行读完，这就是原因。你也知道此行包含名称；同样，第二个readline()调用返回的时候，你知道这行包含年龄等。正如你可以看到，该处理程序仅在有新数据读入时运行，并知道每步的数据是什么。一旦正在运行的线程已处理过读入的某些数据，该线程不会再回退数据（大多如此）。下图也说明了这条原则：

img

（Java IO: 从一个阻塞的流中读数据） 而一个NIO的实现会有所不同，下面是一个简单的例子：

复制代码 (https://link.jianshu.com?t=undefined)代码如下：

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

注意第二行，从通道读取字节到ByteBuffer。当这个方法调用返回时，你不知道你所需的所有数据是否在缓冲区内。你所知道的是，该缓冲区包含一些字节，这使得处理有点困难。

假设第一次 read(buffer)调用后，读入缓冲区的数据只有半行，例如，“Name:An”，你能处理数据吗？显然不能，需要等待，直到整行数据读入缓存，在此之前，对数据的任何处理毫无意义。



所以，你怎么知道是否该缓冲区包含足够的数据可以处理呢？好了，你不知道。发现的方法只能查看缓冲区中的数据。其结果是，在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：

复制代码 (<https://link.jianshu.com?t=undefined>)代码如下：

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while(! bufferFull(bytesRead) ) {
    bytesRead = inChannel.read(buffer);
}
```

bufferFull()方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是否已满。换句话说，如果缓冲区准备好被处理，那么表示缓冲区满了。

bufferFull()方法扫描缓冲区，但必须保持在bufferFull () 方法被调用之前状态相同。如果没有，下一个读入缓冲区的数据可能无法读到正确的位置。这是不可能的，但却是需要注意的又一问题。

如果缓冲区已满，它可以被处理。如果它不满，并且在你的实际案例中有意义，你或许能处理其中的部分数据。但是许多情况下并非如此。下图展示了“缓冲区数据循环就绪”：

img

3) 用来处理数据的线程数

NIO可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。

如果需要管理同时打开的成千上万个连接，这些连接每次只是发送少量的数据，例如聊天服务器，实现NIO的服务器可能是一个优势。同样，如果你需要维持许多打开的连接到其他计算机上，如P2P网络中，使用一个单独的线程来管理你所有出站连接，可能是一个优势。一个线程多个连接的设计方案如



img

Java NIO: 单线程管理多个连接

如果你有少量的连接使用非常高的带宽，一次发送大量的数据，也许典型的IO服务器实现可能非常契合。下图说明了一个典型的IO服务器设计：

img

Java IO: 一个典型的IO服务器设计- 一个连接通过一个线程处理

Java中堆内存和栈内存区别



Java把内存分成两种，一种叫做栈内存，一种叫做堆内存

在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配。当在一段代码块中定义一个变量时，java就在栈中为这个变量分配内存空间，当超过变量的作用域后，java会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用。

堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。

引用变量是普通变量，定义时在栈中分配内存，引用变量在程序运行到作用域外释放。而数组&对象本身在堆中分配，即使程序运行到使用new产生数组和对象的语句所在地代码块之外，数组和对象本身占用的堆内存也不会被释放，**数组和对象在没有引用变量指向它的时候，才变成垃圾，不能再被使用，但是仍然占着内存，在随后的一个不确定的时间被垃圾回收器释放掉。这个也是java比较占内存的主要原因，*****实际上，栈中的变量指向堆内存中的变量，这就是Java中的指针!**

java中内存分配策略及堆和栈的比较

1 内存分配策略

按照编译原理的观点,程序运行时的内存分配有三种策略,分别是静态的,栈式的,和堆式的.

静态存储分配是指在编译时就能确定每个数据目标在运行时刻的存储空间需求,因而在编译时就可以给他们分配固定的内存空间.这种分配策略要求程序代码中不允许有可变数据结构(比如可变数组)的存在,也不允许有嵌套或者递归的结构出现,因为它们都会导致编译程序无法计算准确的存储空间需求.

栈式存储分配也可称为动态存储分配,是由一个类似于堆栈的运行栈来实现的.和静态存储分配相反,在栈式存储方案中,程序对数据区的需求在编译时是完全未知的,只有到运行的时候才能够知道,但是规定在运行中进入一个程序模块时,必须知道该程序模块所需的数据区大小才能够为其分配内存.和我们在数据结构所熟知的栈一样,栈式存储分配按照先进后出的原则进行分配。

静态存储分配要求在编译时能知道所有变量的存储要求,栈式存储分配要求在过程的入口处必须知道所有的存储要求,而堆式存储分配则专门负责在编译时或运行时模块入口处都无法确定存储要求的数据结构的内存分配,比如可变长度串和对对象实例.堆由大片的可利用块或空闲块组成,堆中的内存可以按照任意顺序分配和释放.

2 堆和栈的比较

上面的定义从编译原理的教材中总结而来,除静态存储分配之外,都显得很呆板和难以理解,下面撇开静态存储分配,集中比较堆和栈:

从堆和栈的功能和作用来通俗的比较,**堆主要用来存放对象的，栈主要是用来执行程序的.**而这种不同又主要是由于堆和栈的特点决定的:

在编程中，例如C/C++中，所有的方法调用都是通过栈来进行的,所有的局部变量,形式参数都是从栈中分配内存空间的。实际上也不是什么分配,只是从栈顶向上用就行,就好像工厂中的传送带(conveyor belt)一样,Stack Pointer会自动指引你到放东西的位置,你所要做的只是把东西放下来就行.退出函数的时候，修改栈指针就可以把栈中的内容销毁.这样的模式速度最快,当然要用来运行程序了.需要注意的是,在分配的时候,比如为一个即将要调用的程序模块分配数据区时,应事先知道这个数据区的大小,也就说是虽然分配是在程序运行时进行的,但是分配的大小多少是确定的,不变的,而这个"大小多少"是在编译时确定的,不是在运行时。

堆是应用程序在运行的时候请求操作系统分配给自己内存，由于从操作系统管理的内存分配,所以在分配和销毁时都要占用时间，因此用堆的效率非常低.但是堆的优点在于,编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数



据要在堆里停留多长的时间,因此,用堆保存数据时会得到更大的灵活性。事实上,面向对象的多态性,堆内存分配是必不可少的,因为多态变量所需的存储空间只有在运行时创建了对象之后才能确定.在C++中,要求创建一个对象时,只需用 new命令编制相关的代码即可。执行这些代码时,会在堆里自动进行数据的保存.当然,为达到这种灵活性,必然会付出一定的代价:在堆里分配存储空间时会花掉更长的时间!这也正是导致我们刚才所说的效率低的原因,看来列宁同志说的好,人的优点往往也是人的缺点,人的缺点往往也是人的优点(晕~)。

3 JVM中的堆和栈

JVM是基于堆栈的虚拟机.JVM为每个新创建的线程都分配一个堆栈.也就是说,对于一个Java程序来说,它的运行就是通过对堆栈的操作来完成的。堆栈以帧为单位保存线程的状态。JVM对堆栈只进行两种操作:以帧为单位的压栈和出栈操作。

我们知道,某个线程正在执行的方法称为此线程的当前方法.我们可能不知道,当前方法使用的帧称为当前帧。当线程激活一个Java方法,JVM就会在线程的Java堆栈里新压入一个帧。这个帧自然成为了当前帧.在此方法执行期间,这个帧将用来保存参数,局部变量,中间计算过程和其他数据.这个帧在这里和编译原理中的活动纪录的概念是差不多的。

从Java的这种分配机制来看,堆栈又可以这样理解:堆栈(Stack)是操作系统在建立某个进程时或者线程(在支持多线程的操作系统中是线程)为这个线程建立的存储区域,该区域具有先进后出的特性。

每一个Java应用都唯一对应一个JVM实例,每一个实例唯一对应一个堆。应用程序在运行中所创建的所有类实例或数组都放在这个堆中,并由应用所有的线程共享.跟C/C++不同,Java中分配堆内存是自动初始化的。Java中所有对象的存储空间都是在堆中分配的,但是这个对象的引用却是在堆栈中分配,也就是说在建立一个对象时从两个地方都分配内存,在堆中分配的内存实际建立这个对象,而在堆栈中分配的内存只是一个指向这个堆对象的指针(引用)而已。

Java 中的堆和栈

Java把内存划分成两种:一种是栈内存,一种是堆内存。

在函数中定义的一些基本类型的变量和对象的引用变量都在函数的栈内存中分配。

当在一段代码块定义一个变量时,Java就在栈中为这个变量分配内存空间,当超过变量的作用域后,Java会自动释放掉为该变量所分配的内存空间,该内存空间可以立即被另作他用。

堆内存用来存放由new创建的对象和数组。

在堆中分配的内存,由Java虚拟机的自动垃圾回收器来管理。

在堆中产生了一个数组或对象后,还可以在栈中定义一个特殊的变量,让栈中这个变量的取值等于数组或对象在堆内存中的首地址,栈中的这个变量就成了数组或对象的引用变量。

引用变量就相当于为数组或对象起的一个名称,以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。

具体的说:

栈与堆都是Java用来在Ram中存放数据的地方。与C++不同,Java自动管理栈和堆,程序员不能直接地设置栈或堆。

Java的堆是一个运行时数据区,类的(对象从中分配空间。这些对象通过new、newarray、anewarray和multianewarray等指令建立,它们不需要程序代码来显式的释放。堆是由垃圾回收来负责的,堆的优势是可以动态地分配内存大小,生存期也不必事先告诉编译器,因为它是在运行时动态分配内存的,Java的垃圾收集器会自动收走这些不再使用的数据。但缺点是,由于要在运行时动态分配内存,存取速度较慢。

栈的优势是,存取速度比堆要快,仅次于寄存器,栈数据可以共享。但缺点是,存在栈中的数据大小与生存期必须是确定的,缺乏灵活性。栈中主要存放一些基本类型的变量(int, short, long, byte, float, double, boolean, char)和对象句柄。

栈有一个很重要的特殊性,就是存在栈中的数据可以共享。假设我们同时定义:

```
int a = 3;
int b = 3;
```

编译器先处理int a = 3;首先它会在栈中创建一个变量为a的引用,然后查找栈



中是否有3这个值，如果没找到，就将3存放进来，然后将a指向3。接着处理int b = 3;在创建完b的引用变量后，因为在栈中已经有3这个值，便将b直接指向3。这样，就出现了a与b同时均指向3的情况。这时，如果再令a=4;那么编译器会重新搜索栈中是否有4值，如果没有，则将4存放进来，并令a指向4;如果已经有了，则直接将a指向这个地址。因此a值的改变不会影响到b的值。要注意这种数据的共享与两个对象的引用同时指向一个对象的这种共享是不同的，因为这种情况a的修改并不会影响到b, 它是由编译器完成的，它有利于节省空间。而一个对象引用变量修改了这个对象的内部状态，会影响到另一个对象引用变量

反射讲一讲，主要是概念,都在哪需要反射机制，反射的性能，如何优化

反射机制的定义：

是在运行状态中，对于任意的一个类，都能够知道这个类的所有属性和方法，对任意一个对象都能够通过反射机制调用一个类的任意方法，这种动态获取类信息及动态调用类对象方法的功能称为java的反射机制。

反射的作用：

- 1、动态地创建类的实例，将类绑定到现有的对象中，或从现有的对象中获取类型。
- 2、应用程序需要在运行时从某个特定的程序集中载入一个特定的类

如何保证RESTful API安全性

友情链接： 如何设计好的RESTful API之安全性 (<https://link.jianshu.com?t=http://blog.csdn.net/ywk253100/article/details/25654101>)

如何预防MySQL注入



所谓SQL注入，就是通过把SQL命令插入到Web表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

我们永远不要信任用户的输入，我们必须认定用户输入的数据都是不安全的，我们都需要对用户输入的数据进行过滤处理。

1.以下实例中，输入的用户名必须为字母、数字及下划线的组合，且用户名长度为 8 到 20 个字符之间：

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches))
{
    $result = mysql_query("SELECT * FROM users
        WHERE username=$matches[0]");
}
else
{
    echo "username 输入异常";
}
```

让我们看下在没有过滤特殊字符时，出现的SQL情况：

```
// 设定$name 中插入了我们不需要的SQL语句
$name = "Qadir"; DELETE FROM users;";
mysql_query("SELECT * FROM users WHERE name={$name}");
```

以上的注入语句中，我们没有对 \$name 的变量进行过滤，\$name 中插入了我们不需要的SQL语句，将删除 users 表中的所有数据。

2.在PHP中的 mysql_query() 是不允许执行多个SQL语句的，但是在 SQLite 和 PostgreSQL 是可以同时执行多条SQL语句的，所以我们对这些用户的数据需要进行严格的验证。

防止SQL注入，我们需要注意以下几个要点：

- 1.永远不要信任用户的输入。对用户的输入进行校验，可以通过正则表达式，或限制长度；对单引号和 双"-"进行转换等。
- 2.永远不要使用动态拼装sql，可以使用参数化的sql或者直接使用存储过程进行数据查询存取。
- 3.永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。
- 4.不要把机密信息直接存放，加密或者hash掉密码和敏感的信息。
- 5.应用的异常信息应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装
- 6.sql注入的检测方法一般采取辅助软件或网站平台来检测，软件一般采用sql注入检测工具jsky，网站平台就有亿思网站安全平台检测工具。MDCSOFT SCAN等。采用MDCSOFT-IPS可以有有效的防御SQL注入，XSS攻击等。

3.防止SQL注入

在脚本语言，如Perl和PHP你可以对用户输入的数据进行转义从而防止SQL注入。

PHP的MySQL扩展提供了mysql_real_escape_string()函数来转义特殊的输入字符。

```
if (get_magic_quotes_gpc())
{
    $name = stripslashes($name);
}
$name = mysql_real_escape_string($name);
mysql_query("SELECT * FROM users WHERE name='{$name}'");
```



4. Like语句中的注入

like查询时，如果用户输入的值有"和"%"，则会出现这种情况：用户本来只是想查询"abcd"，查询结果中却有"abcd_"、"abcde"、"abcdf"等等；用户要查询"30%"（注：百分之三十）时也会出现问题。

在PHP脚本中我们可以使用addslashes()函数来处理以上情况，如下实例：

```
$sub = addslashes(mysql_real_escape_string("%something_"), "%_");  
// $sub == \%something\  
mysql_query("SELECT * FROM messages WHERE subject LIKE '{$sub}%')");
```

addslashes()函数在指定的字符前添加反斜杠。

语法格式：

addslashes(string,characters)

参数 描述

string 必需。规定要检查的字符串。

characters 可选。规定受 addslashes() 影响的字符或字符范围。

ThreadLocal(线程变量副本)

Synchronized实现内存共享，ThreadLocal为每个线程维护一个本地变量。

采用空间换时间，它用于线程间的数据隔离，为每一个使用该变量的线程提供一个副本，每个线程都可以独立地改变自己的副本，而不会和其他线程的副本冲突。

ThreadLocal类中维护一个Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值为对应线程的变量副本。

ThreadLocal在spring (<https://link.jianshu.com?t=http://lib.csdn.net/base/javaee>)中发挥着巨大的作用，在管理Request作用域中的Bean、事务管理、任务调度、AOP等模块都出现了它的身影。

Spring中绝大部分Bean都可以声明成Singleton作用域，采用ThreadLocal进行封装，因此有状态的Bean就能够以singleton的方式在多线程中正常工作了。

你能不能谈谈，Java (<https://link.jianshu.com?t=http://lib.csdn.net/base/java>)GC是在什么时候，对什么东西，做了什么事情？



在什么时候：

1. 新生代有一个Eden区和两个survivor区，首先将对象放入Eden区，如果空间不足就向其中的一个survivor区上放，如果仍然放不下就会引发一次发生在新生代的minor GC，将存活的对象放入另一个survivor区中，然后清空Eden和之前的那个survivor区的内存。在某次GC过程中，如果发现仍然又放不下的对象，就将这些对象放入老年代内存里去。

2. 大对象以及长期存活的对象直接进入老年区。

3. 当每次执行minor GC的时候应该对要晋升到老年代的对象进行分析，如果这些马上要到老年区的老年对象的大小超过了老年区的剩余大小，那么执行一次Full GC以尽可能地获得老年区的空间。

对什么东西：从GC Roots搜索不到，而且经过一次标记清理之后仍没有复活的对象。

做什么：

新生代：复制清理；

老年代：标记-清除和标记-压缩算法 (<https://link.jianshu.com?t=http://lib.csdn.net/base/datastructure>)；

永久代：存放Java中的类和加载类的类加载器本身。

GC Roots都有哪些：

- 1. 虚拟机栈中的引用的对象
- 2. 方法区中静态属性引用的对象，常量引用的对象
- 3. 本地方法栈中JNI（即一般说的Native方法）引用的对象。

Volatile和Synchronized四个不同点：

1 粒度不同，前者锁对象和类，后者针对变量

2 syn阻塞，volatile线程不阻塞

3 syn保证三大特性，volatile不保证原子性

4 syn编译器优化，volatile不优化

volatile具备两种特性：

1. 保证此变量对所有线程的可见性，指一条线程修改了这个变量的值，新值对于其他线程来说是可见的，但并不是多线程安全的。

2. 禁止指令重排序优化。

Volatile如何保证内存可见性：

1. 当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存。

2. 当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变量。

同步：就是一个任务的完成需要依赖另外一个任务，只有等待被依赖的任务完成后，依赖任务才能完成。

异步：不需要等待被依赖的任务完成，只是通知被依赖的任务要完成什么工作，只要自己任务完成了就算完成了，被依赖的任务是否完成会通知回来。（异步的特点就是通知）。

打电话和发短信来比喻同步和异步操作。

阻塞：CPU停下来等一个慢的操作完成以后，才会接着完成其他的工作。

非阻塞：非阻塞就是在这个慢的执行时，CPU去做其他工作，等这个慢的完成后，CPU才会接着完成后续的操作。

非阻塞会造成线程切换增加，增加CPU的使用时间能不能补偿系统的切换成本需要考虑。



线程池的作用：

在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。

常用线程池：ExecutorService 是主要的实现类，其中常用的有

Executors.newSingleThreadPool(),newFixedThreadPool(),newCachedThreadPool(),newScheduledThreadPool()。

一致性哈希：

Redis (<https://link.jianshu.com?t=http://lib.csdn.net/base/redis>)

数据结构: String—字符串 (key-value 类型)

索引: B+, B-,全文索引

MySQL (<https://link.jianshu.com?t=http://lib.csdn.net/base/mysql>)的索引是一个数据结构，旨在使数据库高效的查找数据。

常用的数据结构是B+Tree，每个叶子节点不但存放了索引键的相关信息还增加了指向相邻叶子节点的指针，这样就形成了带有顺序访问指针的B+Tree，做这个优化的目的是提高不同区间访问的性能。

什么时候使用索引：

1. 经常出现在group by,order by和distinct关键字后面的字段
2. 经常与其他表进行连接的表，在连接字段上应该建立索引
3. 经常出现在Where子句中的字段
4. 经常出现用作查询选择的字段

Spring IOC AOP（控制反转，依赖注入）



IOC容器：就是具有依赖注入功能的容器，是可以创建对象的容器，IOC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。**通常new一个实例，控制权由程序员控制，而“控制反转”是指new实例工作不由程序员来做而是交给Spring容器来做。。在Spring中BeanFactory是IOC容器的实际代表者。**

DI(依赖注入Dependency injection)：在容器创建对象后，处理对象的依赖关系。

Spring支持三种依赖注入方式，分别是属性（Setter方法）注入，构造注入和接口注入。

在Spring中，那些组成应用的主体及由Spring IOC容器所管理的对象被称之为Bean。

Spring的IOC容器通过反射的机制实例化Bean并建立Bean之间的依赖关系。

简单地讲，Bean就是由Spring IOC容器初始化、装配及被管理的对象。

获取Bean对象的过程，首先通过Resource加载配置文件并启动IOC容器，然后通过getBean方法获取bean对象，就可以调用他的方法。

Spring Bean的作用域：

Singleton：Spring IOC容器中只有一个共享的Bean实例，一般都是Singleton作用域。

Prototype：每一个请求，会产生一个新的Bean实例。

Request：每一次http请求会产生一个新的Bean实例。

AOP就是纵向的编程，如业务1和业务2都需要一个共同的操作，与其往每个业务中都添加同样的代码，不如写一遍代码，让两个业务共同使用这段代码。在日常有订单管理、商品管理、资金管理、库存管理等业务，都会需要到类似**日志记录、事务控制、****权限控制、性能统计、异常处理及事务处理等**。AOP把所有共有代码全部抽取出来，放置到某个地方集中管理，然后在具体运行时，再由容器动态织入这些共有代码。

Spring AOP应用场景

性能检测，访问控制，日志管理，事务等。

默认的策略是如果目标类实现接口，则使用JDK动态代理技术，如果目标对象没有实现接口，则默认会采用CGLIB代理

友情链接：Spring框架IOC容器和AOP解析 (<https://link.jianshu.com?t=http://www.cnblogs.com/xiaoxing/p/5836835.html>)

友情链接：浅谈Spring框架注解的用法分析 (<https://link.jianshu.com?t=http://www.importnew.com/23592.html>)

友情链接：关于Spring的69个面试问答——终极列表 (<https://link.jianshu.com?t=http://www.importnew.com/11657.html>)

代理的共有优点：业务类只需要关注业务逻辑本身，保证了业务类的重用性。



Java静态代理：

代理对象和目标对象实现了相同的接口，目标对象作为代理对象的一个属性，具体接口实现中，代理对象可以在调用目标对象相应方法前后加上其他业务处理逻辑。

缺点：一个代理类只能代理一个业务类。如果业务类增加方法时，相应的代理类也要增加方法。

Java动态代理：

Java动态代理是写一个类实现InvocationHandler接口，重写Invoke方法，在Invoke方法可以进行增强处理的逻辑的编写，这个公共代理类在运行的时候才能明确自己要代理的对象，同时可以实现该被代理类的方法的实现，然后在实现类方法的时候可以进行增强处理。

实际上：代理对象的方法 = 增强处理 + 被代理对象的方法

JDK和CGLIB生成动态代理类的区别：

JDK动态代理只能针对实现了接口的类生成代理（实例化一个类）。此时代理对象和目标对象实现了相同的接口，目标对象作为代理对象的一个属性，具体接口实现中，可以在调用目标对象相应方法前后加上其他业务处理逻辑

CGLIB是针对类实现代理，主要是对指定的类生成一个子类（没有实例化一个类），覆盖其中的方法。

SpringMVC运行原理

1. 客户端请求提交到DispatcherServlet
2. 由DispatcherServlet控制器查询HandlerMapping，找到并分发到指定的Controller中。
4. Controller调用业务逻辑处理后，返回ModelAndView
5. DispatcherServlet查询一个或多个ViewResolver视图解析器，找到ModelAndView指定的视图
6. 视图负责将结果显示到客户端

友情链接：Spring：基于注解的Spring MVC（上） (<https://link.jianshu.com?t=http://www.importnew.com/23015.html>)

友情链接：Spring：基于注解的Spring MVC（下） (<https://link.jianshu.com?t=http://www.importnew.com/23019.html>)

友情链接：SpringMVC与Struts2区别与比较总结 (<https://link.jianshu.com?t=http://blog.csdn.net/chenleixing/article/details/44570681>)

友情链接：SpringMVC与Struts2的对比 (<https://link.jianshu.com?t=http://blog.csdn.net/gstormspire/article/details/8239182/>)

TCP三次握手，四次挥手

TCP作为一种可靠传输控制协议，其核心思想：既要保证数据可靠传输，又要提高传输的效率，而用三次恰恰可以满足以上两方面的需求！****双方都需要确认自己的发信和收信功能正常，收信功能通过接收对方信息得到确认，发信功能需要发出信息—>对方回复信息得到确认。

三次握手过程：



1. 第一次握手：建立连接。客户端发送连接请求报文段，将 SYN 位置为1，Sequence Number 为x；然后，客户端进入 SYN_SEND 状态，等待服务器的确认；
2. 第二次握手：服务器收到客户端的 SYN 报文段，需要对这个 SYN 报文段进行确认，设置 ACK 为x+1(Sequence Number +1)；同时，自己还要发送 SYN 请求信息，将 SYN 位置为1，Sequence Number 为y；服务器端将上述所有信息放到一个报文段（即 SYN+ACK 报文段）中，一并发送给客户端，此时服务器进入 SYN_RECV 状态；
3. 第三次握手：客户端收到服务器的 SYN+ACK 报文段。然后将 Acknowledgment Number 设置为y+1，向服务器发送 ACK 报文段，这个报文段发送完毕以后，客户端和服务器端都进入 ESTABLISHED 状态，完成TCP三次握手。

TCP工作在网络OSI的七层模型中的第四层——Transport层，IP在第三层——Network层
◆ARP在第二层——Data Link层；在第二层上的数据，我们把它叫Frame，在第三层上的数据叫Packet，第四层的数据叫Segment。

四次挥手过程：

1. 第一次分手：主机1（可以使客户端，也可以是服务器端），设置 Sequence Number 和 Acknowledgment Number，向主机2发送一个 FIN 报文段；此时，主机1进入 FIN_WAIT_1 状态；这表示主机1没有数据要发送给主机2了；
2. 第二次分手：主机2收到了主机1发送的 FIN 报文段，向主机1回一个 ACK 报文段，Acknowledgment Number 为 Sequence Number 加1；主机1进入 FIN_WAIT_2 状态；主机2告诉主机1，我“同意”你的关闭请求；
3. 第三次分手：主机2向主机1发送 FIN 报文段，请求关闭连接，同时主机2进入 LAST_ACK 状态；
4. 第四次分手：主机1收到主机2发送的 FIN 报文段，向主机2发送 ACK 报文段，然后主机1进入 TIME_WAIT 状态；主机2收到主机1的 ACK 报文段以后，就关闭连接；此时，主机1等待2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，主机1也可以关闭连接了。

(2) 而关闭连接却是四次挥手呢？

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。

为什么建立连接是三次握手

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。

关闭连接却是四次挥手呢

而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

HTTPS和HTTP 为什么更安全，先看这些

http默认端口是80 https是443

http是HTTP协议运行在TCP之上。所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。



https是HTTP运行在SSL/TLS之上，SSL/TLS运行在TCP之上。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。此外客户端可以验证服务器端的身份，如果配置了客户端验证，服务器方也可以验证客户端的身份。HTTP(应用层) 和TCP(传输层)之间插入一个SSL协议。

一个Http请求

DNS域名解析 -> 发起TCP的三次握手 -> 建立TCP连接后发起http请求 -> 服务器响应http请求，浏览器得到html代码 -> 浏览器解析html代码，并请求html代码中的资源（如js、css、图片等） -> 浏览器对页面进行渲染呈现给用户

友情链接： HTTP与HTTPS的区别 (<https://link.jianshu.com?t=http://www.mahaixiang.cn/internet/1233.html>)

友情链接： HTTPS 为什么更安全，先看这些 (<https://link.jianshu.com?t=http://blog.jobbole.com/110373/>)

友情链接： HTTP请求报文和HTTP响应报文 (<https://link.jianshu.com?t=http://www.cnblogs.com/biyeymyhjob/archive/2012/07/28/2612910.html>)

友情链接： HTTP 请求方式: GET和POST的比较 (<https://link.jianshu.com?t=http://www.cnblogs.com/igeneral/p/3641574.html>)

Mybatis

每一个Mybatis的应用程序都以一个SqlSessionFactory对象的实例为核心。首先用字节流通过Resource将配置文件读入，然后通过SqlSessionFactoryBuilder().build方法创建SqlSessionFactory，然后再通过sqlSessionFactory.openSession()方法创建一个sqlSession为每一个数据库事务服务。

经历了Mybatis初始化 -> 创建SqlSession -> 运行SQL语句 返回结果三个过程

Servlet和Filter的区别：



整的流程是：Filter对用户请求进行预处理，接着将请求交给Servlet进行处理并生成响应，最后Filter再对服务器响应进行后处理。

Filter有如下几个用处：

Filter可以进行对特定的url请求和相应做预处理和后处理。

在HttpServletRequest到达Servlet之前，拦截客户的HttpServletRequest。

根据需要检查HttpServletRequest，也可以修改HttpServletRequest头和数据。

在HttpServletResponse到达客户端之前，拦截HttpServletResponse。

根据需要检查HttpServletResponse，也可以修改HttpServletResponse头和数据。

实际上Filter和Servlet极其相似，区别只是Filter不能直接对用户生成响应。实际上Filter里doFilter()方法里的代码就是从多个Servlet的service()方法里抽取的通用代码，通过使用Filter可以实现更好的复用。

Filter和Servlet的生命周期：

1.Filter在web服务器启动时初始化

2.如果某个Servlet配置了 1，该Servlet也是在Tomcat（Servlet容器）启动时初始化。

3.如果Servlet没有配置1，该Servlet不会在Tomcat启动时初始化，而是在请求到来时初始化。

4.每次请求，Request都会被初始化，响应请求后，请求被销毁。

5.Servlet初始化后，将不会随着请求的结束而注销。

6.关闭Tomcat时，Servlet、Filter依次被注销。

HashMap和TreeMap区别



HashMap：基于哈希表实现。使用HashMap要求添加的键类明确定义了hashCode()和equals() [可以重写hashCode()和equals()]，为了优化HashMap空间的使用，您可以调优初始容量和负载因子。适合查找和删除

(1)HashMap(): 构建一个空的哈希映像

(2)HashMap(Map m): 构建一个哈希映像，并且添加映像m的所有映射

(3)HashMap(int initialCapacity): 构建一个拥有特定容量的空的哈希映像

(4)HashMap(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空的哈希映像

TreeMap：基于红黑树实现。TreeMap没有调优选项，因为该树总处于平衡状态。适合按照自然顺序或者自定义的顺序排序遍历key

(1)TreeMap(): 构建一个空的映像树

(2)TreeMap(Map m): 构建一个映像树，并且添加映像m中所有元素

(3)TreeMap(Comparator c): 构建一个映像树，并且使用特定的比较器对关键字进行排序

(4)TreeMap(SortedMap s): 构建一个映像树，添加映像树s中所有映射，并且使用与有序映像s相同的比较器排序

友情链接：Java中HashMap和TreeMap的区别深入理解

(<https://link.jianshu.com?t=http://www.jb51.net/article/32652.htm>)

HashMap冲突

友情链接：HashMap冲突的解决方法以及原理分析 (<https://link.jianshu.com?t=http://www.cnblogs.com/peizhe123/p/5790252.html>)

友情链接：HashMap的工作原理 (<https://link.jianshu.com?t=http://www.importnew.com/7099.html>)

友情链接：HashMap和Hashtable的区别 (<https://link.jianshu.com?t=http://www.importnew.com/7010.html>)

友情链接：2种办法让HashMap线程安全 (<https://link.jianshu.com?t=http://flyfoxs.iteye.com/blog/2100120>)

HashMap, ConcurrentHashMap与LinkedHashMap的区别



1. ConcurrentHashMap是使用了锁分段技术来保证线程安全的，锁分段技术：首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问
2. ConcurrentHashMap 是在每个段（segment）中线程安全的
3. LinkedHashMap维护一个双链表，可以将里面的数据按写入的顺序读出
4. ConcurrentHashMap应用场景

1：ConcurrentHashMap的应用场景是高并发，但是并不能保证线程安全，而同步的HashMap和Hashtable的是锁住整个容器，而加锁之后ConcurrentHashMap不需要锁住整个容器，只需要锁住对应的Segment就好了，所以可以保证高并发同步访问，提升了效率。

2：可以多线程写。

ConcurrentHashMap把HashMap分成若干个Segment

1.get时，不加锁，先定位到segment然后在找到头结点进行读取操作。而value是volatile变量，所以可以保证在竞争条件时保证读取最新的值，如果读到的value是null，则可能正在修改，那么久调用ReadValueUnderLock函数，加锁保证读到的数据是正确的。

2.Put时会加锁，一律添加到hash链的头。

3.Remove时也会加锁，由于next是final类型不可改变，所以必须把删除的节点之前的节点都复制一遍。

4.ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术。它使用了多个锁来控制对Hash表的不同Segment进行的修改。

ConcurrentHashMap的应用场景是高并发，但是并不能保证线程安全，而同步的HashMap和Hashtable的是锁住整个容器，而加锁之后ConcurrentHashMap不需要锁住整个容器，只需要锁住对应的segment就好了，所以可以保证高并发同步访问，提升了效率。

友情链接：Java集合—ConcurrentHashMap原理分析 (<https://link.jianshu.com?t=http://www.cnblogs.com/ITtangtang/p/3948786.html>)

ThreadPoolExecutor 的内部工作原理

进程间的通信方式



1. 管道(pipe)：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。
2. 有名管道(named pipe)：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
3. 信号量(semaphore)：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
3. 消息队列(message queue)：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
5. 信号(sinal)：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
6. 共享内存(shared memory)：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。
7. 套接字(socket)：套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

死锁的必要条件

1. 互斥 至少有一个资源处于非共享状态
2. 占有并等待
3. 非抢占
4. 循环等待

解决死锁，第一个是死锁预防，就是不让上面的四个条件同时成立。二是，合理分配资源。

三是使用银行家算法，如果该进程请求的资源操作系统

(<https://link.jianshu.com?t=http://lib.csdn.net/base/operatingsystem>)剩余量可以满足，那么就分配。

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

赞赏支持



(/u/6bcc679bfb4f)

互联网公司-技术岗笔面试题 (/nb/11147513)

举报文章 © 著作权归作者所有



时芥蓝 (/u/bd4811478d4b) ♂

写了 96510 字，被 227 人关注，获得了 301 个喜欢
(/u/bd4811478d4b)

+ 关注

原来你是我最想留住的，幸运





被以下专题收入，发现更多相似内容

+ 收入我的专题



面试 (/c/3bbcc919ce49?utm_source=desktop&utm_medium=notes-included-collection)



NIU (/c/49f4e5542f91?utm_source=desktop&utm_medium=notes-included-collection)



面试题 (/c/de74a70723d8?utm_source=desktop&utm_medium=notes-included-collection)



java神作 (/c/6543daf9eb10?utm_source=desktop&utm_medium=notes-included-collection)



架构 (/c/0a9da375c721?utm_source=desktop&utm_medium=notes-included-collection)



编程知识点 (/c/9a56f7fc508d?utm_source=desktop&utm_medium=notes-included-collection)



java (/c/742ce65d61e7?utm_source=desktop&utm_medium=notes-included-collection)

展开更多

掘金 Java 文章精选合集 (/p/893b21dce761?utm_campaign=maleskine&...

Java 基础思维导图，让 Java 不再难懂 - 工具资源 - 掘金思维导图的好处 最近看了一些文章的思维导图，发现思维导图真是个强大的工具。了解了思维导图的作用之后，觉得把它运用到java上应该是个不错的想法...



掘金官方 (/u/5fc9b6410f4f?

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)

Java初级面试题 (/p/08153f5678de?utm_campaign=maleskine&utm_co...

1. Java基础部分 基础部分的顺序：基本语法，类相关的语法，内部类的语法，继承相关的语法，异常的语法，线程的语法，集合的语法，io的语法，虚拟机方面的语法。1、一个".java"源文件中是否可以包括多个...



叫我二叔呀 (/u/0ac615e458d5?

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)

百战程序员V1.2——尚学堂旗下高端培训_ Java1573题 (/p/49ad52bd5405...

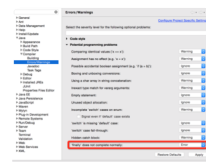
百战程序员_ Java1573题 QQ群：561832648489034603 掌握80%年薪20万掌握50%年薪10万 全程项目穿插，从易到难，含17个项目视频和资料持续更新，请关注www.itbaizhan.com 国内最牛七星级团队马士兵、...



Albert陈凯 (/u/185a3c553fc6?

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)


(/p/e3cd3095beb2?



utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation)


【转】Java面试题全集（上） (/p/e3cd3095beb2?utm_campaign=males...

转自: <http://blog.csdn.net/jackfrued/article/details/44921941> Java面试题全集 (中) :
<http://www.jianshu.com/p/cbc7fcb62951>Java面试题全集 (下) : <http://www...>

 王帅199207 (/u/d23e9f4acfea?
utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)

Java面试 (/p/d110b26685ac?utm_campaign=maleskine&utm_content=...

一. Java基础部分 21、一个".java"源文件中
是否可以包括多个...


 wy_sure (/u/803ab8e1a409?
utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)

(/p/0f9e92c58d66?



utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation)
一路溪声，扬首摘辰 (17) (/p/0f9e92c58d66?utm_campaign=maleskin...

远赴外省出差？你简直是在胡闹 文|水芋 《一路溪声，扬首摘辰》目录 “你简直是在胡闹！” “是，路总，但您不妨往好的方面想，那个合作公司的王总既然那么刁蛮难缠，说不定送过去一个御女一个清纯大学生轻轻...


 水芋 (/u/80c6741b14d4?
utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)

(/p/1157954882a1?



utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation)
挖掘潜能 师生同成长~厦门之旅D3 (/p/1157954882a1?utm_campaign=m...

正像人们说的那样，当班主任过去靠经验，今天靠科学。我们要塑造的是健康的学生，我们要塑造学生的追求、学生的思想和学生的情感，唤起学生主动学习，积极成长的内驱动力，即使孩子们离开了校园，离开了老师，...


 贺子兰 (/u/a1d986e9f168?
utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)

(/p/7fe08e5da514?

utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation)
容嬷嬷才是还珠格格里的隐藏大boss (/p/7fe08e5da514?utm_campaign=...

她还有着全剧里最多的——秘密。】有人说她是十八年前大明湖畔的夏雨荷，也有人说她是《甄嬛传》里擅长刺绣的安陵容。在大众印象里，容嬷嬷是个...



 路演去哪 (/u/a5d587239395?


utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)

(/p/94b9f3093e5a?



utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation)
日常矫情 (/p/94b9f3093e5a?utm_campaign=maleskine&utm_content=n...

还没开学的学校操场一如既往的安静 一点儿灯光都没有 我喜欢坐在看台上数星星 也不看台阶是不是干净 直接坐下来 蹭一屁股的灰也不在乎 和平常上课即使座位很干净也还要擦三遍的我判若两人 我很喜欢晒太阳 ...

 我才不会难过 (/u/c6e98560e97a?
utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)




(/p/0c122c0e85e8?



utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation)
《爱的意义》 (/p/0c122c0e85e8?utm_campaign=maleskine&utm_conte...

作者:快乐小散 “和什么样的人同行真的很重要”，十年前就曾看到过这句话，但没有深刻的理解。因为那时无权选择和谁在一起，只能被动的接受世界上的一切，好像自己的命运已经被安排好，如同带了枷锁的牛马...

 快樂小散 (/u/ce42dca412f5?

utm_campaign=maleskine&utm_content=user&utm_medium=seo_notes&utm_source=recommendation)