目录

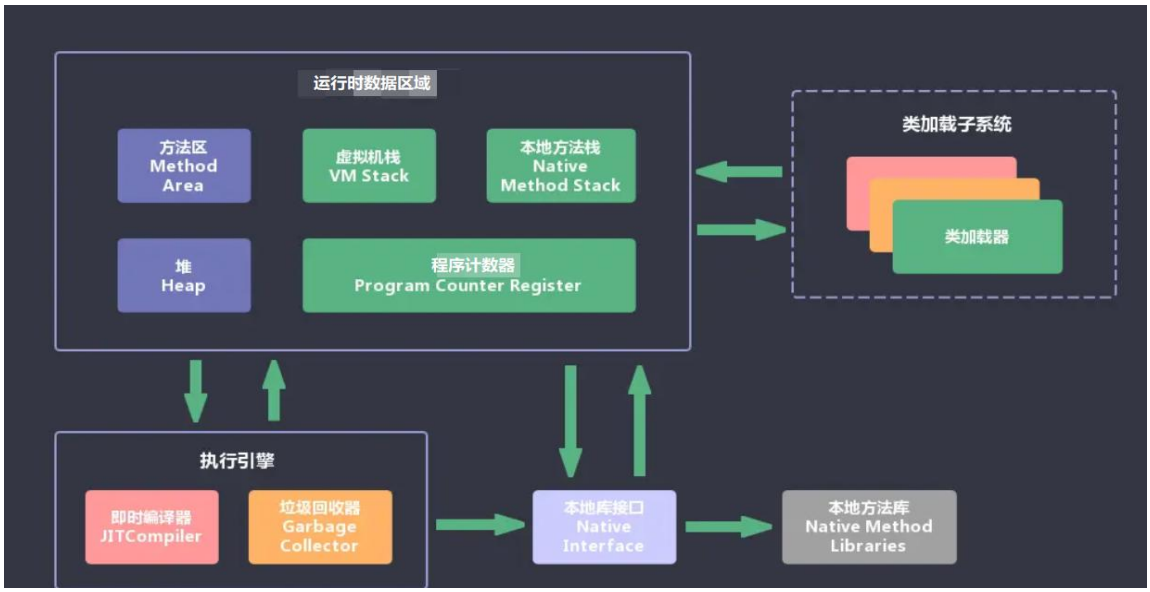
Java 面试题：JVM 篇

Java 内存区域

解释 Java 堆空间及 GC?

当通过 Java 命令启动 Java 进程的时候,会为它分配内存。内存的一部分用于创建堆空间,当程序中创建对象的时候,就从对空间中分配内存。GC 是 JVM 内部的一个进程,回收无效对象的内存用于将来的分配。

说一下 JVM 的主要组成部分及其作用?



JVM 包含两个子系统和两个组件,两个子系统为 Class loader(类装载)、Execution engine(执行引擎);两个组件为 Runtime data area(运行时数据区)、Native Interface(本地接口)。

Class loader(类装载):根据给定的全限定名类名(如:java.lang.Object)来装载 class 文件到 Runtime data area 中的 method area。

Execution engine (执行引擎):执行 classes 中的指令。

Native Interface(本地接口):与 native libraries 交互,是其它编程语言交互的接口。

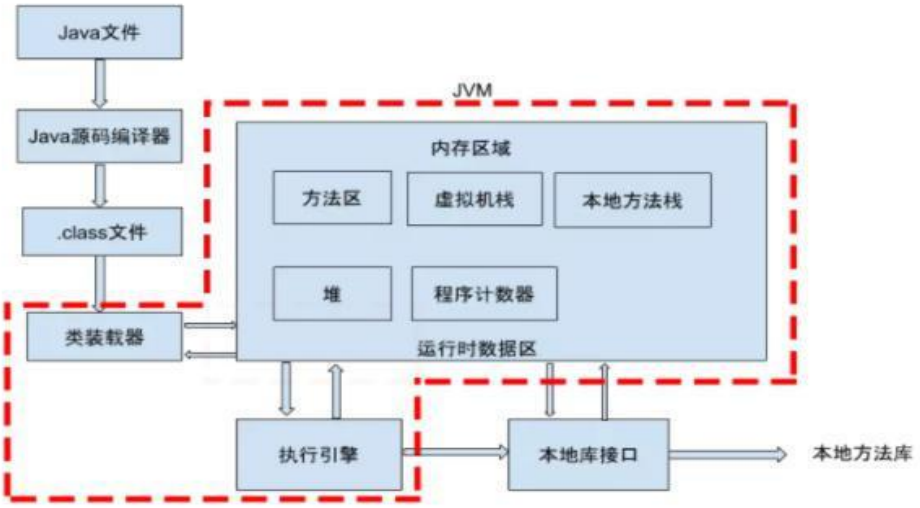
Runtime data area(运行时数据区域):这就是我们常说的 JVM 的内存。

作用：首先通过编译器把 Java 代码转换成字节码，类加载器（ClassLoader）再把字节码加载到内存中，将其放在运行时数据区（Runtime data area）的方法区内，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。

下面是 Java 程序运行机制详细说明

Java 程序运行机制步骤

- 首先利用 IDE 集成开发工具编写 Java 源代码，源文件的后缀为.java；
- 再利用编译器(javac 命令)将源代码编译成字节码文件，字节码文件的后缀名为.class；
- 运行字节码的工作是由解释器(java 命令)来完成的。



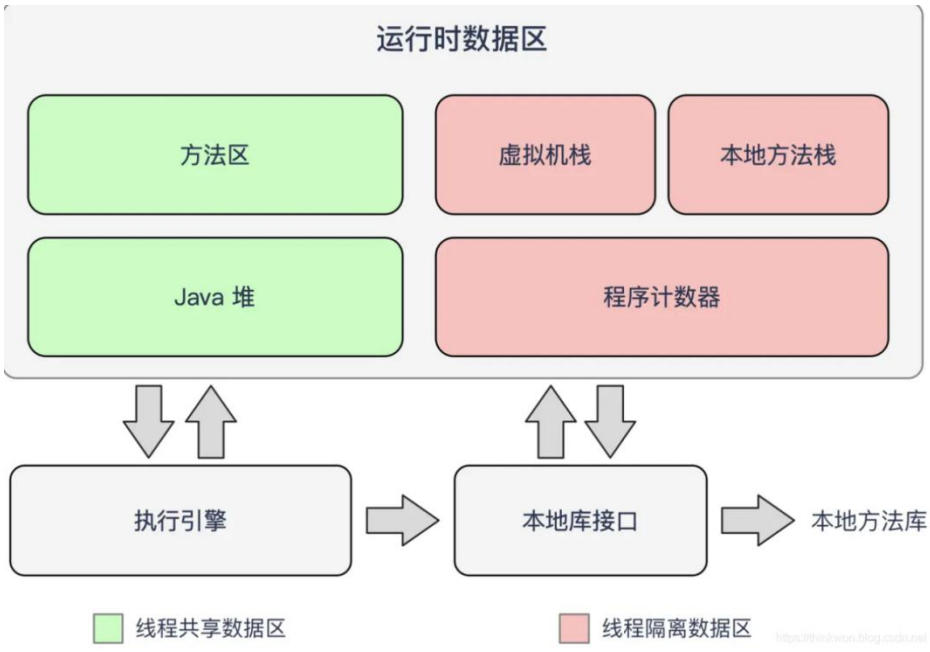
从上图可以看，java 文件通过编译器变成了.class 文件，接下来类加载器又将这些.class 文件加载到 JVM 中。

其实可以一句话来解释：类的加载指的是将类的.class 文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个 java.lang.Class 对象，用来封装类在方法区内的数据结构。

说一下 JVM 运行时数据区？ 或者：说一下 JVM 内存模型？

思路： 给面试官画一下 JVM 内存模型图，并描述每个模块的定义，作用，以及可能会存在的问题，如栈溢出等。

Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存区域划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有些区域随着虚拟机进程的启动而存在，有些区域则是依赖线程的启动和结束而建立和销毁。Java 虚拟机所管理的内存被划分为如下几个区域：



程序计数器（Program Counter Register）：当前线程所执行的字节码的行号指示器，字节码解析器的工作是通过改变这个计数器的值，来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能，都需要依赖这个计数器来完成；

Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作数栈、动态链接、方法出口等信息；

本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的；

- Java 堆（Java Heap）：Java 虚拟机中内存最大的一块，是被所有线程共享的，几乎所有的对象实例都在这里分配内存；
- 方法区（Methed Area）：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。

深拷贝和浅拷贝

浅拷贝（shallowCopy）只是增加了一个指针指向已存在的内存地址，

深拷贝（deepCopy）是增加了一个指针并且申请了一个新的内存，使这个增加的指针指向这个新的内存，

使用深拷贝的情况下，释放内存的时候不会因为出现浅拷贝时释放同一个内存的错误。

浅复制：仅仅是指向被复制的内存地址，如果原地址发生改变，那么浅复制出来的对象也会相应的改变。

深复制：在计算机中开辟一块**新的内存地址**用于存放复制的对象。

说一下堆栈的区别?

物理地址

堆的物理地址分配对对象是不连续的。因此性能慢些。在 GC 的时候也要考虑到不连续的分配，所以有各种算法。比如，标记-消除，复制，标记-压缩，分代（即新生代使用复制算法，老年代使用标记——压缩）

栈使用的是数据结构中的栈，先进后出的原则，物理地址分配是连续的。所以性能快。

内存分别

堆因为是不连续的，所以分配的内存是在运行期确认的，因此大小不固定。一般堆大小远远大于栈。

栈是连续的，所以分配的内存大小要在编译期就确认，大小是固定的。

存放的内容

堆存放的是对象的实例和数组。因此该区更关注的是数据的存储

栈存放：局部变量，操作数栈，返回结果。该区更关注的是程序方法的执行。

PS：

1.静态变量放在方法区

2.静态的对象还是放在堆。

程序的可见度

堆对于整个应用程序都是共享、可见的。

栈只对于线程是可见的。所以也是线程私有。他的生命周期和线程相同。

Java 中堆和栈有什么区别?

JVM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小，也不会在多个线程之间共享，而堆被整个 JVM 的所有线程共享。

队列和栈是什么？有什么区别?

队列和栈都是被用来预存储数据的。

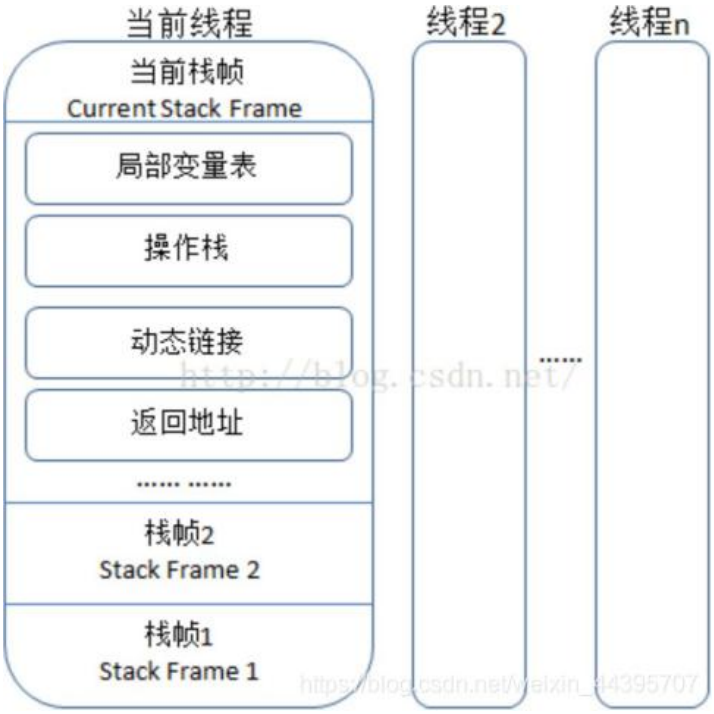
- 操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。
- 可操作的方式不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。
- 操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（不能从中间插入），每次离开的成员总是队列头上（不允许中途离队）。而栈为后进先出（LIFO）,即每次删除（出栈）的总是当前栈中最新的元素，即最后插入（进栈）的元素，而最先插入的被放在栈的底部，要到最后才能删除。

虚拟机栈(线程私有)

是描述 java 方法执行的内存模型，每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到

出栈的过程。

栈帧（ Frame）是用来存储数据和部分过程结果的数据结构，同时也被用来处理动态链接(Dynamic Linking)、 方法返回值和异常分派（ Dispatch Exception）。 栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完成（抛出了在方法内未被捕获的异常）都算作方法束。



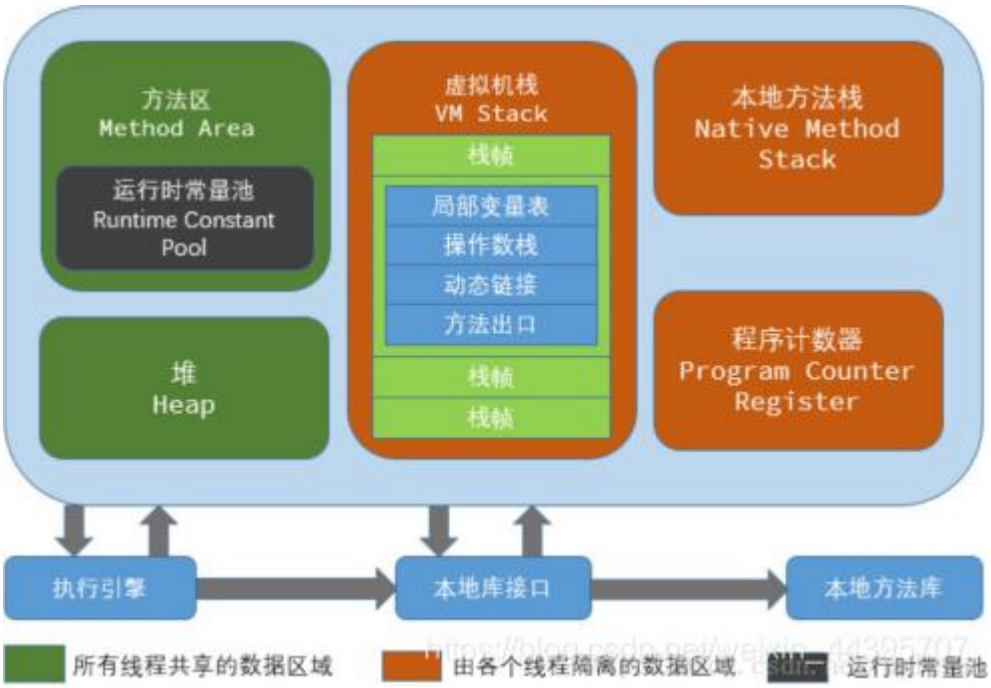
程序计数器(线程私有)

一块较小的内存空间，是当前线程所执行的字节码的行号指示器，每条线程都要有一个独立的程序计数器，这类内存也称为“线程私有” 的内存。

正在执行 java 方法的话，计数器记录的是虚拟机字节码指令的地址（当前指令的地址） 。如果还是 Native 方法，则为空。这个内存区域是唯一一个在虚拟机中没有规定任何 OutOfMemoryError 情况的区域。

什么是直接内存?

直接内存并不是 JVM 运行时数据区的一部分，但也会被频繁的使用：在 JDK 1.4 引入的 NIO 提供了基于 Channel 与 Buffer 的 IO 方式，它可以使用 Native 函数库直接分配堆外内存，然后使用 DirectByteBuffer 对象作为这块内存的引用进行操作(详见: Java I/O 扩展)，这样就避免了在 Java 堆和 Native 堆中来回复制数据，因此在一些场景中可以显著提高性能。



HotSpot 虚拟机对象探秘

对象的创建

Header	解释
使用 new 关键字	调用了构造函数
使用 Class 的 newInstance 方	调用了构造函数

法	
使用 Constructor 类的 newInstance 方法	调用了构造函数
使用 clone 方法	没有调用构造函数
使用反序列化	没有调用构造函数

说到对象的创建，首先让我们看看 Java 中提供的几种对象创建方式：

下面是对象创建的主要流程:

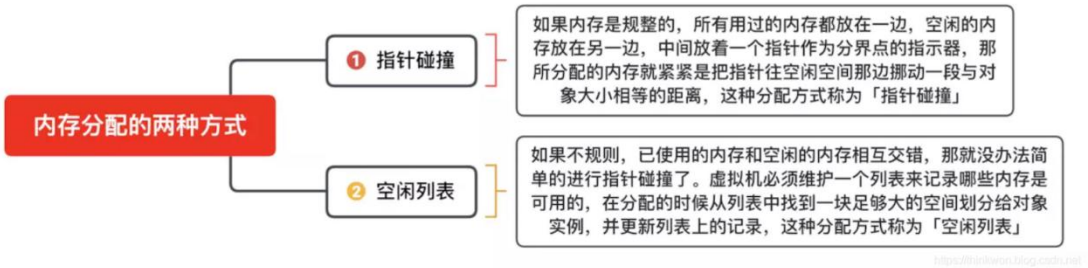
虚拟机遇到一条 new 指令时，先检查常量池是否已经加载相应的类，如果没有，必须先执行相应的类加载。类加载通过后，接下来分配内存。若 Java 堆中内存是绝对规整的，使用“指针碰撞“方式分配内存；如果不是规整的，就从空闲列表中分配，叫做”空闲列表“方式。划分内存时还需要考虑一个问题-并发，也有两种方式: CAS 同步处理，或者本地线程分配缓冲(Thread Local Allocation Buffer, TLAB)。然后内存空间初始化操作，接着是做一些必要的对象设置(元信息、哈希码...)，最后执行<init>方法。

为对象分配内存

类加载完成后，接着会在 Java 堆中划分一块内存分配给对象。内存分配根据 Java 堆是否规整，有两种方式：

- 指针碰撞：如果 Java 堆的内存是规整，即所有用过的内存放在一边，而空闲的的放在另一边。分配内存时将位于中间的指针指示器向空闲的内存移动一段与对象大小相等的距离，这样便完成分配内存工作。
- 空闲列表：如果 Java 堆的内存不是规整的，则需要由虚拟机维护一个列表来记录那些内存是可用的，这样在分配的时候可以从列表中查询到足够大的内存分配给对象，并在分配后更新列表记录。

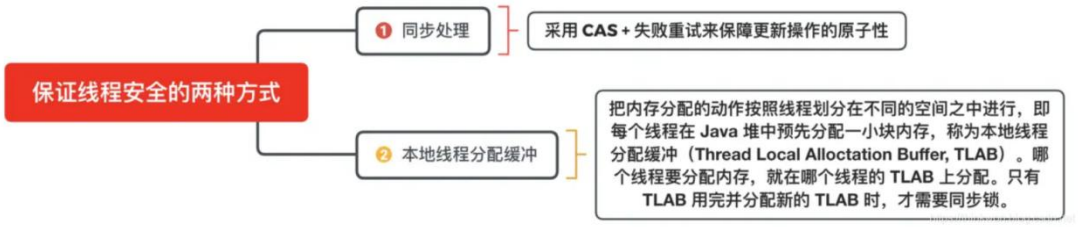
选择哪种分配方式是由 Java 堆是否规整来决定的，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。



处理并发安全问题

对象的创建在虚拟机中是一个非常频繁的行为，哪怕只是修改一个指针所指向的位置，在并发情况下也是不安全的，可能出现正在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针来分配内存的情况。解决这个问题有两种方案：

- 对分配内存空间的动作进行同步处理（采用 CAS + 失败重试来保障更新操作的原子性）；
- 把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer, TLAB）。哪个线程要分配内存，就在哪个线程的 TLAB 上分配。只有 TLAB 用完并分配新的 TLAB 时，才需要同步锁。通过 -XX:+/-UserTLAB 参数来设定虚拟机是否使用 TLAB。



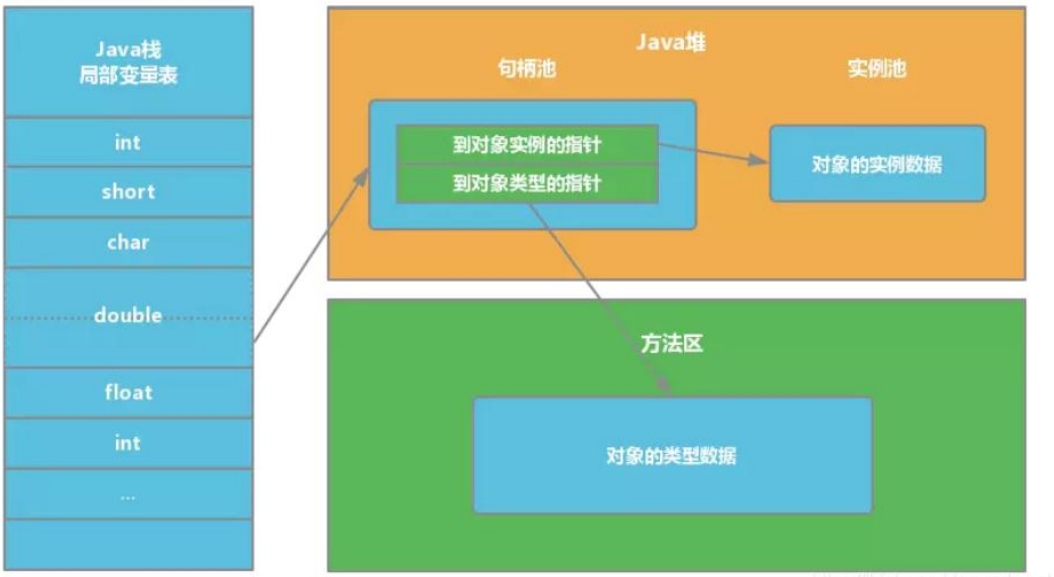
对象的访问定位

Java 程序需要通过 JVM 栈上的引用访问堆中的具体对象。对象的访问方式取决于 JVM 虚拟机的实现。目前主流的访问方式有 **句柄** 和 **直接指针** 两种方式。

> **指针：** 指向对象，代表一个对象在内存中的起始地址。
>
> **句柄：** 可以理解为指向指针的指针，维护着对象的指针。句柄不直接指向对象，而是指向对象的指针（句柄不发生变化，指向固定内存地址），再由对象的指针指向对象的真实内存地址。

句柄访问

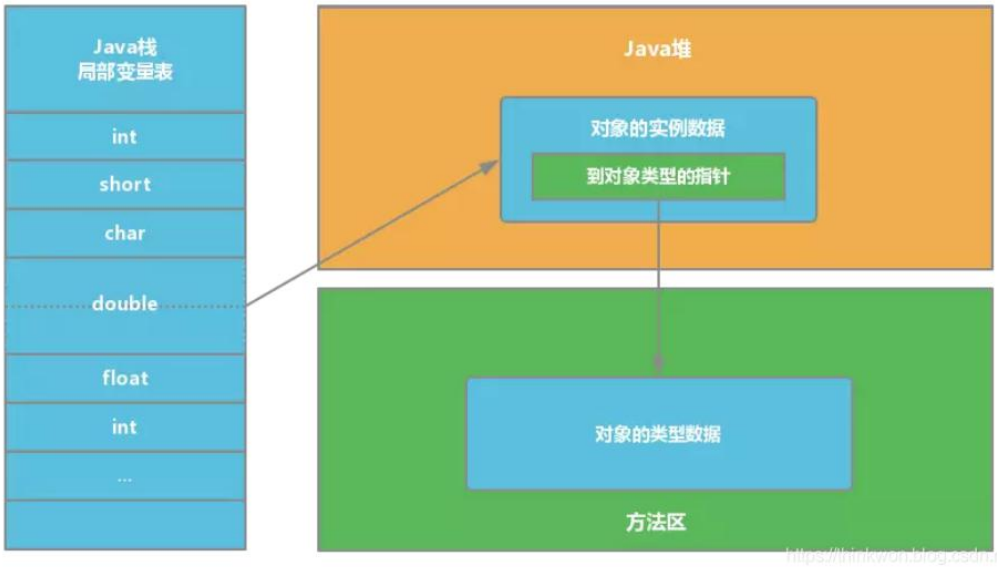
Java 堆中划分出一块内存来作为**句柄池**，引用中存储对象的**句柄地址**，而句柄中包含了**对象实例数据**与**对象类型数据**各自的**具体地址**信息，具体构造如下图所示：



优势：引用中存储的是**稳定**的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变**句柄中**的**实例数据指针**，而**引用**本身不需要修改。

直接指针

如果使用**直接指针**访问，**引用** 中存储的直接就是**对象地址**，那么 Java 堆对象内部的布局中就必须考虑如何放置访问**类型数据**的相关信息。



优势：速度更**快**，节省了**一次指针定位**的时间开销。由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是非常可观的执行成本。HotSpot 中采用的就是这种方式。

64 位 JVM 中，int 的长度是多数?

Java 中，int 类型变量的长度是一个固定值，与平台无关，都是 32 位。意思就是说，在 32 位和 64 位的 Java 虚拟机中，int 类型的长度是相同的。

32 位和 64 位的 JVM，int 类型变量的长度是多数?

32 位和 64 位的 JVM 中，int 类型变量的长度是相同的，都是 32 位或者 4 个字节。

怎样通过 Java 程序来判断 JVM 是 32 位 还是 64 位?

你可以检查某些系统属性如 sun.arch.data.model 或 os.arch 来获取该信息。

32 位 JVM 和 64 位 JVM 的最大堆内存分别是多数?

理论上说上 32 位的 JVM 堆内存可以到达 2^32，即 4GB，但实际上会比这个小很多。不同操作系统之间不同，如 Windows 系统大约 1.5GB，Solaris 大约 3GB。64 位 JVM 允许指定最大的堆内存，理论上可以达到 2^64，这是一个非常大的数字，实际上你可以指定堆内存大小到 100GB。甚至有的 JVM，如 Azul，堆内存到 1000G 都是可能的。

JRE、JDK、JVM 及 JIT 之间有什么不同?

JRE 代表 Java 运行时（Java run-time），是运行 Java 引用所必须的。

JDK 代表 Java 开发工具（Java development kit），是 Java 程序的开发工具，如 Java 编译器，它也包含 JRE。

JVM 代表 Java 虚拟机（Java virtual machine），它的责任是运行 Java 应用。

JIT 代表即时编译（Just In Time compilation），当代码执行的次数超过一定的阈值时，会将 Java 字节码转换为本地代码，如，主要的热点代码会被准换为本地代码，这样有利大幅度提高 Java 应用的性能。

内存溢出异常

Java 会存在内存泄漏吗？请简单描述

内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。理论上来说，Java 是有 GC 垃圾回收机制的，也就是说，不再被使用的对象，会被 GC 自动回收掉，自动从内存中清除。

但是，即使这样，Java 也还是存在着内存泄漏的情况，java 导致内存泄露的原因很明确：长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是 java 中内存泄露的发生场景。

什么情况下会发生栈内存溢出。

思路： 描述栈定义，再描述为什么会溢出，再说明一下相关配置参数，OK 的话可以给面试官手写是一个栈溢出的 demo。

参考答案：

- 栈是线程私有的，他的生命周期与线程相同，每个方法在执行的时候都会创建一个栈帧，用来存储局部变量表，操作数栈，动态链接，方法出口等信息。局部变量表又包含基本数据类型，对象引用类型
- 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常，方法递归调用产生这种结果。
- 如果 Java 虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是无法申请到足够的内存去完成扩展，或者在新建立线程的时候没有足够的内存去创建对应的虚拟机栈，那么 Java 虚拟机将抛出一个 OutOfMemory 异常。(线程启动过多)

- 参数 -Xss 去调整 JVM 栈的大小

垃圾收集器

简述 Java 垃圾回收机制

在 java 中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在 JVM 中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

GC 是什么？为什么要 GC

GC 是垃圾收集的意思（Gabage Collection）,内存处理是编程人员容易出现问题的地方，忘记或者错误的内存

回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动

回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

垃圾回收的优点和原理。并考虑 2 种回收机制

java 语言最显著的特点就是引入了垃圾回收机制，它使 java 程序员在编写程序时不再考虑内存管理的问题。

由于有这个垃圾回收机制，java 中的对象不再有“作用域”的概念，只有引用的对象才有“作用域”。

垃圾回收机制有效的防止了内存泄露，可以有效的使用可使用的内存。

垃圾回收器通常作为一个单独的低级别的线程运行，在不可预知的情况下对内存堆中已经死亡的或很长时间没有用过的对象进行清除和回收。

程序员不能实时的对某个对象或所有对象调用垃圾回收器进行垃圾回收。

垃圾回收有分代复制垃圾回收、标记垃圾回收、增量垃圾回收。

垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？

对于 GC 来说，当程序员创建对象时，GC 就开始监控这个对象的地址、大小以及使用情况。

通常，GC 采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当 GC 确定一些对象为"不可达"时，GC 就有责任回收这些内存空间。

可以。程序员可以手动执行 System.gc()，通知 GC 运行，但是 Java 语言规范并不保证 GC 一定会执行。

你能保证 GC 执行吗?

不能，虽然你可以调用 System.gc() 或者 Runtime.gc()，但是没有办法保证 GC 的执行。

Java 中都有哪些引用类型?

- 强引用：发生 gc 的时候不会被回收。
- 软引用：有用但不是必须的对象，在发生内存溢出之前会被回收。

- 弱引用：有用但不是必须的对象，在下一次 GC 时会被回收。
- 虚引用（幽灵引用/幻影引用）：无法通过虚引用获得对象，用 PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知。

介绍一下强引用、软引用、弱引用、虚引用的区别?

思路： 先说一下四种引用的定义，可以结合代码讲一下，也可以扩展谈到 ThreadLocalMap 里弱引用用处。

参考答案：

1）强引用

我们平时 new 了一个对象就是强引用，例如 Object obj = new Object();即使在内存不足的情况下，JVM 宁愿抛出 OutOfMemory 错误也不会回收这种对象。

2）软引用

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。

```
SoftReference<String> softRef=new SoftReference<String>(str);     // 软引用
```

用处： 软引用在实际中有重要的应用，例如浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

（1）如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建

（2）如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出

如下代码：

```
Browser prev = new Browser();                    // 获取页面进行浏览
SoftReference sr = new SoftReference(prev); // 浏览完毕后置为软引用          if(s
r.get()!=null){
    rev = (Browser) sr.get();              // 还没有被回收器回收，直接获取
}else{
    prev = new Browser();                  // 由于内存吃紧，所以对软引用的对象回收
了
    sr = new SoftReference(prev);          // 重新构建
}
```

3）弱引用

具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

```
String str=new String("abc");     WeakReference<String> abcWeakRef = new
WeakReference<String>(str);
```

```
str=null;

等价于

str = null;

System.gc();
```

4）虚引用

如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。虚引用主要用来跟踪对象被垃圾回收器回收的活动。

怎么判断对象是否可以被回收?

垃圾收集器在做垃圾回收的时候，首先需要判定的就是哪些内存是需要被回收的，哪些对象是「存活」的，是不可以被回收的；哪些对象已经「死掉」了，需要被回收。

一般有两种方法来判断：

- 引用计数器法：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题；
- 可达性分析算法：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。

在 Java 中，对象什么时候可以被垃圾回收

当对象对当前使用这个对象的应用程序变得不可触及的时候，这个对象就可以被回收了。

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免 Full GC 是非常重要的原因。

JVM 运行时堆内存如何分代?

Java 堆从 GC 的角度还可以细分为: 新生代(Eden 区、 From Survivor 区和 To Survivor 区)和老年代。

参考图 1:

参考图 2:

从图中可以看出： 堆大小 = 新生代 + 老年代。其中，堆的大小可以通过参数 –Xms、-Xmx 来指定。

默认的，新生代（Young）与老年代（Old）的比例的值为 **1:2**（ 该值可以通过参数 –XX:NewRatio 来指定 ），

即：**新生代（Young）= 1/3 的堆空间大小。老年代（Old）= 2/3 的堆空间大小。**

其中，新生代（Young）被细分为 Eden 和 两个 Survivor 区域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分

默认的，**Eden: from : to = 8 :1 : 1**（ 可以通过参数**–XX:SurvivorRatio** 来设定 ），即： **Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。**

JVM **每次只会使用 Eden 和其中的一块 Survivor 区域**来为对象服务，所以**无论什么时候，总是有一块 Survivor 区域是空闲着的。**

因此，新生代实际可用的内存空间为 9/10（ 即 90%)的新生代空间。

新生代

是用来存放新生的对象。一般占据堆的 1/3 空间。由于频繁创建对象，所以新生代会频繁触发 MinorGC 进行垃圾回收。新生代又分为 Eden 区、ServivorFrom、 ServivorTo 三个区。

Eden 区

Java 新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老年

代）。当 Eden 区内存不够的时候就会触发 MinorGC，对新生代区进行一次垃圾回收。

Servivor from 区

上一次 GC 的幸存者，作为这一次 GC 的被扫描者。

Servivor to 区

保留了一次 MinorGC 过程中的幸存者。

MinorGC 的过程（复制->清空->互换）

MinorGC 采用复制算法。

1. eden、 servicorFrom 复制到 ServicorTo，年龄+1

首先，把 Eden 和 ServivorFrom 区域中存活的对象复制到 ServicorTo 区域（如果有对象的年龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果 ServicorTo 不够位置了就放到老年区）；

2. 清空 eden、 servicorFrom

然后，清空 Eden 和 ServicorFrom 中的对象；

3. ServicorTo 和 ServicorFrom 互换

最后， ServicorTo 和 ServicorFrom 互换，原 ServicorTo 成为下一次 GC 时的 ServicorFrom 区。

老年代

主要存放应用程序中生命周期长的内存对象。

老年代的对象比较稳定，所以 MajorGC （常常称之为 FULL GC）不会频繁执行。在进行 FULL GC 前一般都先进行了一次 MinorGC，使得有新生代的

对象晋身入老年代，导致空间不够用时才触发。当无法找到足够大的连续空间分配给新创建的较大对象时也会提前触发一次 MajorGC 进行垃圾回收腾出空间。

FULL GC 采用标记清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。 ajorGC 的耗时比较长，因为要扫描再回收。 FULL GC 会产生内存碎片，为了减少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配。当老年代也满了装不下的时候，就会抛出 OOM（Out of Memory）异常。

永久代

指内存的永久保存区域，主要存放 Class 和 Meta（元数据）的信息,Class 在被加载的时候被放入永久区域， 它和和存放实例的区域不同,GC 不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的 Class 的增多而胀满，最终抛出 OOM 异常。

JVM 内存为什么要分成新生代，老年代，持久代。新生代中为什么要分为 Eden 和 Survivor。

思路： 先讲一下 JAVA 堆，新生代的划分，再谈谈它们之间的转化，相互之间一些参数的配置（如： –XX:NewRatio，–XX:SurvivorRatio 等），再解释为什么要这样划分，最好加一点自己的理解。

参考答案：

这样划分的目的是为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。

1）共享内存区划分

- 共享内存区 = 持久带 + 堆
- 持久带 = 方法区 + 其他
- Java 堆 = 老年代 + 新生代
- 新生代 = Eden + S0 + S1

2）一些参数的配置

- 默认的，新生代（Young）与老年代（Old）的比例的值为 1:2 ，可以通过参数 –XX:NewRatio 配置。
- 默认的，Eden : from : to = 8 : 1 : 1（ 可以通过参数 –XX:SurvivorRatio 来设定)
- Survivor 区中的对象被复制次数为 15(对应虚拟机参数 -XX:+MaxTenuringThreshold)

3)为什么要分为 Eden 和 Survivor?为什么要设置两个 Survivor 区?

- 如果没有 Survivor，Eden 区每进行一次 Minor GC，存活的对象就会被送到老年代。老年代很快被填满，触发 Major GC.老年代的内存空间远大于新生代，进行一次 Full GC 消耗的时间比 Minor GC 长得多,所以需要分为 Eden 和 Survivor。

- Survivor 的存在意义，就是减少被送到老年代的对象，进而减少 Full GC 的发生，Survivor 的预筛选保证，只有经历 16 次 Minor GC 还能在新生代中存活的对象，才会被送到老年代。
- 设置两个 Survivor 区最大的好处就是解决了碎片化，刚刚新建的对象在 Eden 中，经历一次 Minor GC，Eden 中的存活对象就会被移动到第一块 survivor space S0，Eden 被清空；等 Eden 区再满了，就再触发一次 Minor GC，Eden 和 S0 中的存活对象又会被复制送入第二块 survivor space S1（这个过程非常重要，因为这种复制算法保证了 S1 中来自 S0 和 Eden 两部分的存活对象占用连续的内存空间，避免了碎片化的发生）

JVM 中一次完整的 GC 流程是怎样的，对象如何晋升到老年代

思路：先描述一下 Java 堆内存划分，再解释 Minor GC，Major GC，full GC，描述它们之间转化流程。

我的答案：

- Java 堆 = 老年代 + 新生代
- 新生代 = Eden + S0 + S1
- 当 Eden 区的空间满了， Java 虚拟机会触发一次 Minor GC，以收集新生代的垃圾，存活下来的对象，则会转移到 Survivor 区。
- **大对象**（需要大量连续内存空间的 Java 对象，如那种很长的字符串）**直接进入老年态**；

- 如果对象在 Eden 出生，并经过第一次 Minor GC 后仍然存活，并且被 Survivor 容纳的话，年龄设为 1，每熬过一次 Minor GC，年龄+1，**若年龄超过一定限制（15），则被晋升到老年态。**即**长期存活的对象进入老年态。**
- 老年代满了而**无法容纳更多的对象**，Minor GC 之后通常就会进行 Full GC，Full GC 清理整个内存堆 – **包括年轻代和年老代。**
- Major GC **发生在老年代的 GC**，**清理老年区**，经常会伴随至少一次 Minor GC，**比 Minor GC 慢 10 倍以上。**

JVM 中的永久代中会发生垃圾回收吗

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免 Full GC 是非常重要的原因。请参考下 Java8：从永久代到元数据区
(译者注：Java8 中已经移除了永久代，新加了一个叫做元数据区的 native 内存区)

JAVA8 与元数据

在 Java8 中， 永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。元空间的本质和永久代类似，元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。 类的元数据放入 native memory，字符串池和类的静态变量放入 java 堆中， 这样可以加载多少类的元数据就不再由 MaxPermSize 控制，而由系统的实际可用空间来控制。

如何判断对象可以被回收?

判断对象是否存活一般有两种方式:

- 引用计数:

每个对象有一个引用计数属性，新增一个引用时计数加 1，引用释放时计数减 1，计数为 0 时可以回收。此方法简单，无法解决对象相互循环引用的问题。

- 可达性分析（Reachability Analysis）:

从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的，不可达对象。

引用计数法

在 Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单的办法是通过引用计数来判断一个对象是否可以回收。简单说，即一个对象如果没有任何与之关联的引用， 即他们的引用计数都不为 0， 则说明对象不太可能再被用到，那么这个对象就是可回收对象。

可达性分析

为了解决引用计数法的循环引用问题， Java 使用了可达性分析的方法。通过一系列的“GC roots”对象作为起点搜索。如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的。要注意的是，不可达对象不等价于可回收对象， 不可达对象变为可回收对象至少要经过两次标记过程。两次标记后仍然是可回收对象，则将面临回收。

Minor GC 与 Full GC 分别在什么时候发生?

新生代内存不够用时候发生 MGC 也叫 YGC，JVM 内存不够的时候发生 FGC

垃圾收集算法有哪些类型?

GC 最基础的算法有三类： 标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

标记 -清除算法，“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。

复制算法，“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

标记-压缩算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

分代收集算法，“分代收集”（Generational Collection）算法，把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法

说一下 JVM 有哪些垃圾回收算法?

- 标记-清除算法：标记无用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片。

- 复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半。
- 标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。
- 分代算法：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。

标记-清除算法

标记无用对象，然后进行清除回收。

标记-清除算法（Mark-Sweep）是一种常见的基础垃圾收集算法，它将垃圾收集分为两个阶段：

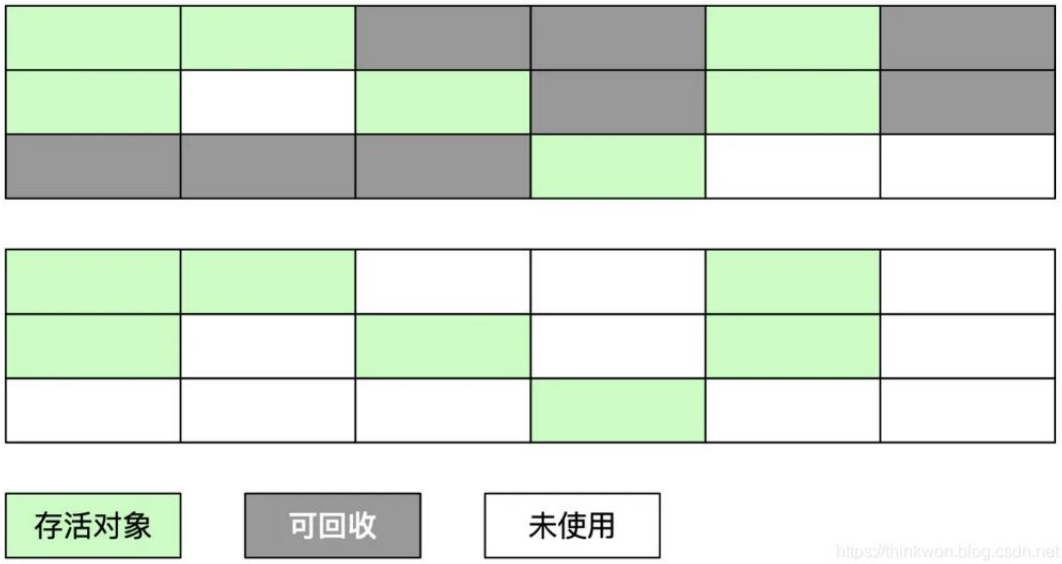
- 标记阶段：标记出可以回收的对象。
- 清除阶段：回收被标记的对象所占用的空间。

标记-清除算法之所以是基础的，是因为后面讲到的垃圾收集算法都是在此算法的基础上进行改进的。

优点：实现简单，不需要对象进行移动。

缺点：标记、清除过程效率低，产生大量不连续的内存碎片，提高了垃圾回收的频率。

标记-清除算法的执行的过程如下图所示



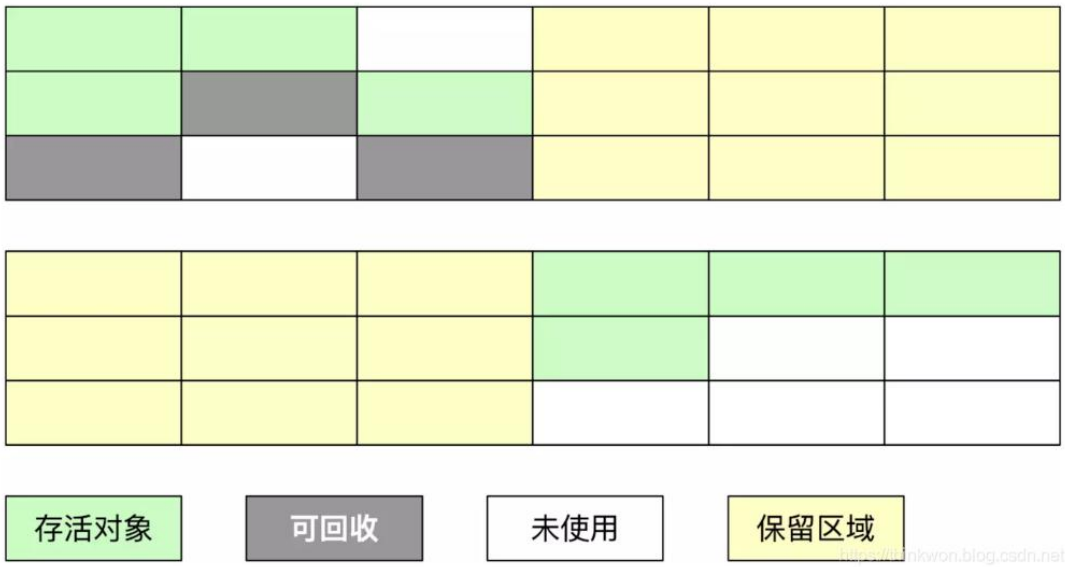
复制算法

为了解决标记-清除算法的效率不高的问题，产生了复制算法。它把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾收集时，遍历当前使用的区域，把存活对象复制到另外一个区域中，最后将当前使用的区域的可回收的对象进行回收。

优点：按顺序分配内存即可，实现简单、运行高效，不用考虑内存碎片。

缺点：可用的内存大小缩小为原来的一半，对象存活率高时会频繁进行复制。

复制算法的执行过程如下图所示



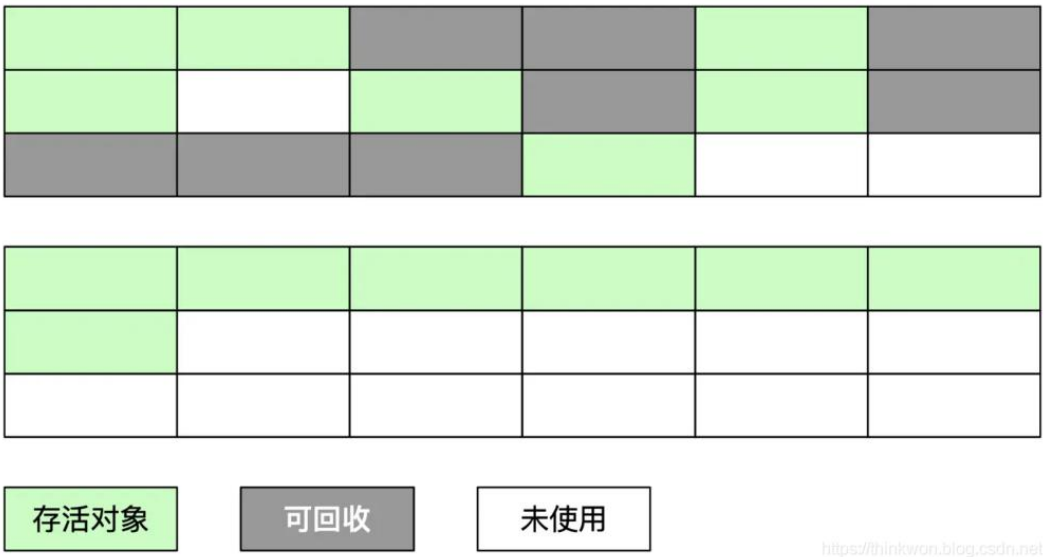
标记-整理算法

在新生代中可以使用复制算法，但是在老年代就不能选择复制算法了，因为老年代的对象存活率会较高，这样会有较多的复制操作，导致效率变低。标记-清除算法可以应用在老年代中，但是它效率不高，在内存回收后容易产生大量内存碎片。因此就出现了一种标记-整理算法（Mark-Compact）算法，与标记-整理算法不同的是，在标记可回收的对象后将所有存活的对象压缩到内存的一端，使他们紧凑的排列在一起，然后对端边界以外的内存进行回收。回收后，已用和未用的内存都各自一边。

优点：解决了标记-清理算法存在的内存碎片问题。

缺点：仍需要进行局部对象移动，一定程度上降低了效率。

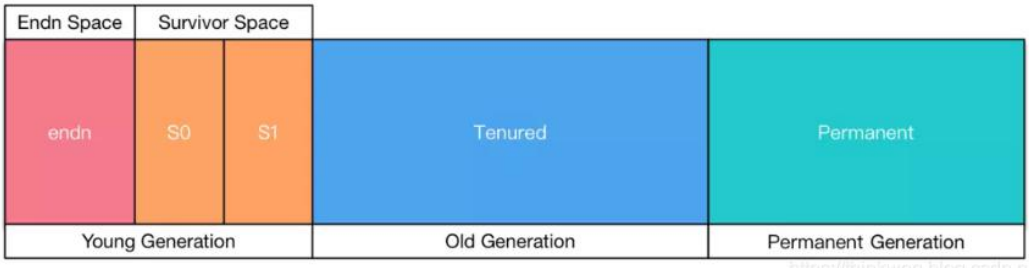
标记-整理算法的执行过程如下图所示



分代收集算法

分代收集法是目前大部分 JVM 所采用的方法，其核心思想是根据对象存活的不同生命周期将内存划分为不同的域，一般情况下将 GC 堆划分为老生代(Tenured/Old Generation)和新生代(YoungGeneration)。老生代的特点是每次垃圾回收时只有少量对象需要被回收，新生代的特点是每次垃圾回收时都有大量垃圾需要被回收，因此可以根据不同区域选择不同的算法。

当前商业虚拟机都采用**分代收集**的垃圾收集算法。分代收集算法，顾名思义是根据对象的**存活周期**将内存划分为几块。一般包括**年轻代**、**老年代** 和 **永久代**，如图所示：

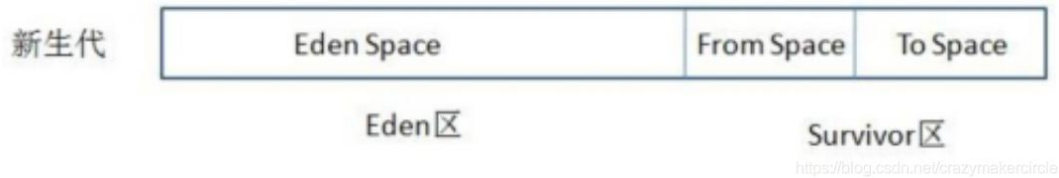


当前主流 VM 垃圾收集都采用“分代收集” (Generational Collection)算法，这种算法会根据对象存活周期的不同将内存划分为几块，如 JVM 中的 新生代、老年代、永久代， 这样就可以根据各年代特点分别采用最适当的 GC 算法

新生代与复制算法

每次垃圾收集都能发现大批对象已死，只有少量存活. 因此选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集

目前大部分 JVM 的 GC 对于新生代都采取 Copying 算法，因为新生代中每次垃圾回收都要回收大部分对象，即要复制的操作比较少，但通常并不是按照 1： 1 来划分新生代。一般将新生代划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用 Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另一块 Survivor 空间中。



老年代与标记复制算法

因为老年代对象存活率高、没有额外空间对它进行分配担保，就必须采用“标记—清理”或“标记—整理” 算法来进行回收，不必进行内存复制，且直接腾出空闲内存。因而采用 Mark-Compact 算法。

1. JAVA 虚拟机提到过的处于方法区的永生代(Permanet Generation)， 它用来存储 class 类，常量，方法描述等。对永生代的回收主要包括废弃常量和无用的类。

2. 对象的内存分配主要在新生代的 Eden Space 和 Survivor Space 的 From Space(Survivor 目前存放对象的那一块)，少数情况会直接分配到老生代。

3. 当新生代的 Eden Space 和 From Space 空间不足时就会发生一次 GC，进行 GC 后， EdenSpace 和 From Space 区的存活对象会被挪到 To Space，然后将 Eden Space 和 FromSpace 进行清理。

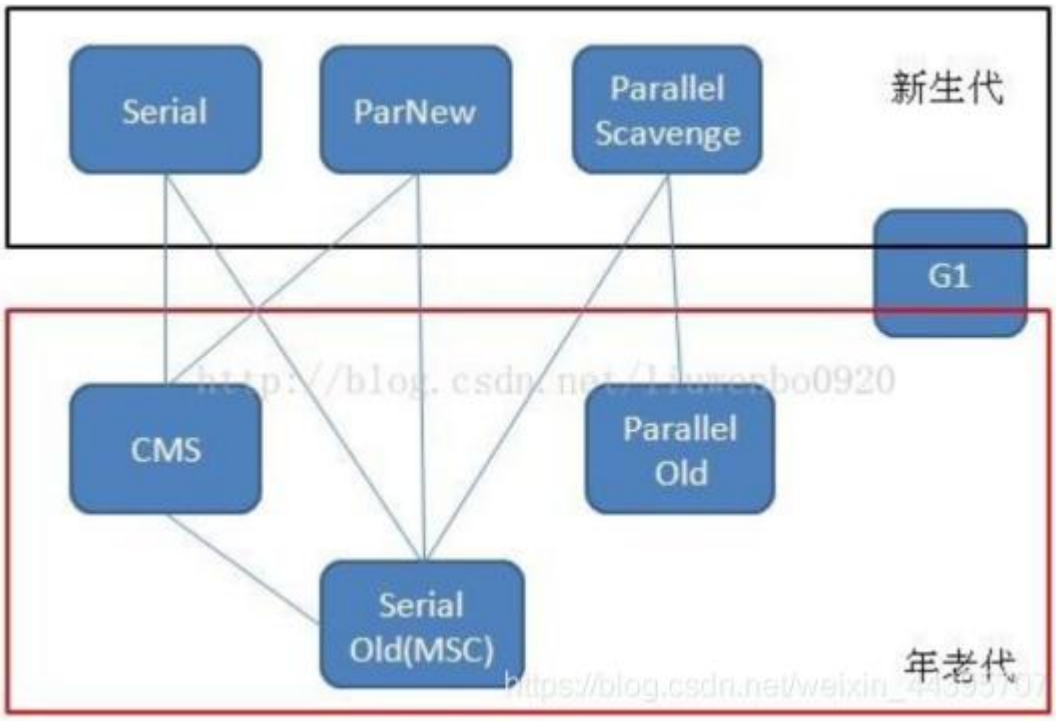
4. 如果 To Space 无法足够存储某个对象，则将这个对象存储到老生代。

5. 在进行 GC 后，使用的便是 Eden Space 和 To Space 了，如此反复循环。

6. 当对象在 Survivor 区躲过一次 GC 后，其年龄就会+1。 默认情况下年龄到达 15 的对象会被移到老生代中。

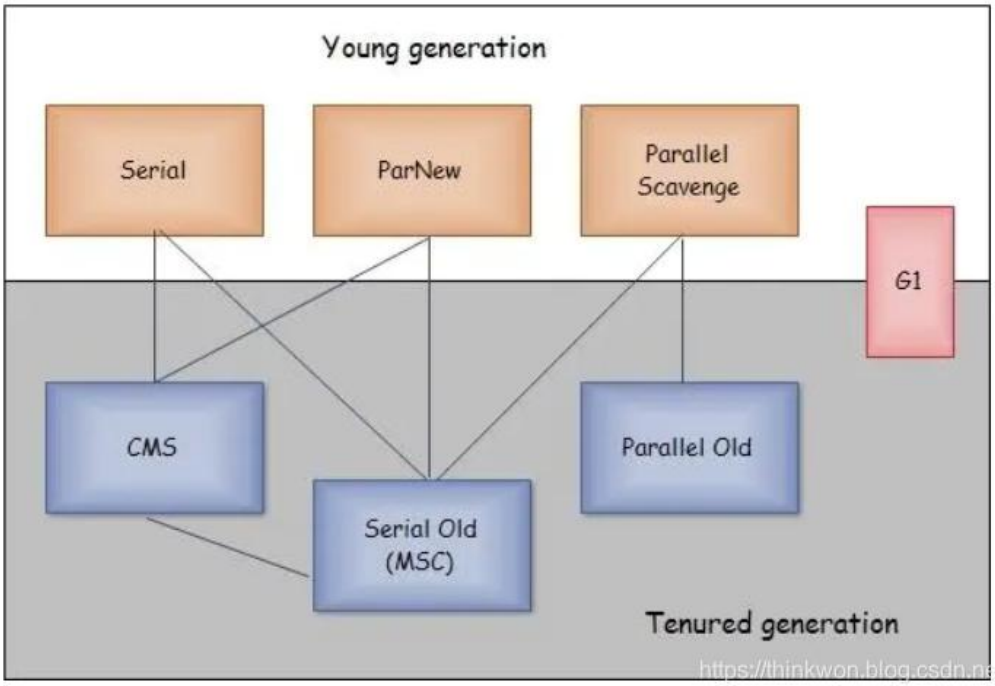
GC 垃圾收集器

Java 堆内存被划分为新生代和年老代两部分，新生代主要使用复制和标记-清除垃圾回收算法；年老代主要使用标记-整理垃圾回收算法，因此 java 虚拟中针对新生代和年老代分别提供了多种不同的垃圾收集器， JDK1.6 中 Sun HotSpot 虚拟机的垃圾收集器如下：



说一下 JVM 有哪些垃圾回收器?

如果说垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。下图展示了 7 种作用于不同分代的收集器，其中用于回收新生代的收集器包括 Serial、PraNew、Parallel Scavenge，回收老年代的收集器包括 Serial Old、Parallel Old、CMS，还有用于回收整个 Java 堆的 G1 收集器。不同收集器之间的连线表示它们可以搭配使用。



Serial 收集器（复制算法)：新生代单线程收集器，标记和清理都是单线程，优点是简单高效;

ParNew 收集器（复制算法)：新生代收并行集器，实际上是 Serial 收集器的多线程版本，在多核 CPU 环境下有着比 Serial 更好的表现;

Parallel Scavenge 收集器（复制算法)：新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC 线程时间)，高吞吐量可以高效率的利用 CPU 时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景;

Serial Old 收集器（标记-整理算法)：老年代单线程收集器，Serial 收集器的老年代版本;

Parallel Old 收集器（标记-整理算法)： 老年代并行收集器，吞吐量优先，Parallel Scavenge 收集器的老年代版本;

CMS(Concurrent Mark Sweep)收集器（标记-清除算法）： 老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短 GC 回收停顿时间。

G1(Garbage First)收集器（标记-整理算法）： Java 堆并行收集器，G1 收集器是 JDK1.7 提供的一个新收集器，G1 收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1 收集器不同于之前的收集器的一个重要特点是：G1 回收的范围是整个 Java 堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。

Serial 与 Parallel GC 之间的不同之处?

Serial 与 Parallel 在 GC 执行的时候都会引起 stop-the-world。它们之间主要不同 serial 收集器是默认的复制收集器，执行 GC 的时候只有一个线程，而 parallel 收集器使用多个 GC 线程来执行。

类似的问题：你知道哪几种垃圾收集器，各自的优缺点，重点讲下 cms 和 G1，包括原理，流程，优缺点。

思路： 一定要记住典型的垃圾收集器，尤其 cms 和 G1，它们的原理与区别，涉及的垃圾回收算法。

参考答案：

1）几种垃圾收集器：

- **Serial 收集器：** 单线程的收集器，收集垃圾时，必须 stop the world，使用复制算法。
- **ParNew 收集器：** Serial 收集器的多线程版本，也需要 stop the world，复制算法。
- **Parallel Scavenge 收集器：** 新生代收集器，复制算法的收集器，并发的多线程收集器，目标是达到一个可控的吞吐量。如果虚拟机总共运行 100 分钟，其中垃圾花掉 1 分钟，吞吐量就是 99%。
- **Serial Old 收集器：** 是 Serial 收集器的老年代版本，单线程收集器，使用标记整理算法。
- **Parallel Old 收集器：** 是 Parallel Scavenge 收集器的老年代版本，使用多线程，标记-整理算法。
- **CMS(Concurrent Mark Sweep) 收集器：** 是一种以获得最短回收停顿时间为目标的收集器，**标记清除算法，运作过程：初始标记，并发标记，重新标记，并发清除**，收集结束会产生大量空间碎片。
- **G1 收集器：** 标记整理算法实现，**运作流程主要包括以下：初始标记，并发标记，最终标记，筛选标记。**不会产生空间碎片，可以精确地控制停顿。

2）CMS 收集器和 G1 收集器的区别：

CMS 收集器是老年代的收集器，可以配合新生代的 Serial 和 ParNew 收集器一起使用；

G1 收集器收集范围是老年代和新生代，不需要结合其他收集器使用；

CMS 收集器以最小的停顿时间为目标的收集器；

G1 收集器可预测垃圾回收的停顿时间

CMS 收集器是使用“标记-清除”算法进行的垃圾回收，容易产生内存碎片

G1 收集器使用的是“标记-整理”算法，进行了空间整合，降低了内存空间碎片。

详细介绍一下 CMS 垃圾回收器?

CMS 是英文 Concurrent Mark-Sweep 的简称，是以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。对于要求服务器响应速度的应用上，这种垃圾回收器非常适合。在启动 JVM 的参数加上“-XX:+UseConcMarkSweepGC”来指定使用 CMS 垃圾回收器。

CMS 使用的是标记-清除的算法实现的，所以在 gc 的时候回产生大量的内存碎片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被降低。

Serial 垃圾收集器（单线程、 复制算法）

Serial（英文连续） 是最基本垃圾收集器，使用复制算法，曾经是 JDK1.3.1 之前新生代唯一的垃圾收集器。 Serial 是一个单线程的收集器， 它不但只会使用一个 CPU 或一条线程去完成垃圾收集工作，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，直到垃圾收集结束。

Serial 垃圾收集器虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限定单个 CPU 环境来说，没有线程交互的开销，可以获得

最高的单线程垃圾收集效率，因此 Serial 垃圾收集器依然是 java 虚拟机运行在 Client 模式下默认的新生代垃圾收集器。

ParNew 垃圾收集器（Serial+多线程）

ParNew 垃圾收集器其实是 Serial 收集器的多线程版本，也使用复制算法，除了使用多线程进行垃圾收集之外，其余的行为和 Serial 收集器完全一样，ParNew 垃圾收集器在垃圾收集过程中同样也要暂停所有其他的工作线程。ParNew 收集器默认开启和 CPU 数目相同的线程数，可以通过 -XX:ParallelGCThreads 参数来限制垃圾收集器的线程数。【Parallel：平行的】

ParNew 虽然是除了多线程外和 Serial 收集器几乎完全一样，但是 ParNew 垃圾收集器是很多 java 虚拟机运行在 Server 模式下新生代的默认垃圾收集器。

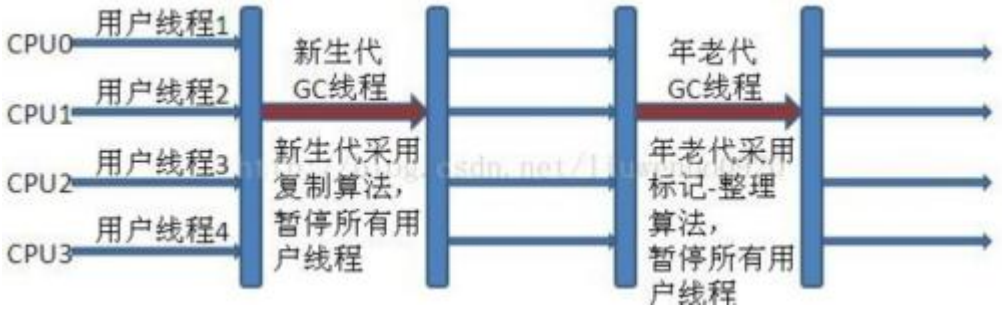
Parallel Scavenge 收集器（多线程复制算法、高效）

Parallel Scavenge 收集器也是一个新生代垃圾收集器，同样使用复制算法，也是一个多线程的垃圾收集器， 它重点关注的是程序达到一个可控制的吞吐量（Thoughput， CPU 用于运行用户代码的时间/CPU 总消耗时间，即吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)），高吞吐量可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适用于在后台运算而不需要太多交互的任务。 自适应调节策略也是 ParallelScavenge 收集器与 ParNew 收集器的一个重要区别。

Serial Old 收集器（单线程标记整理算法 ）

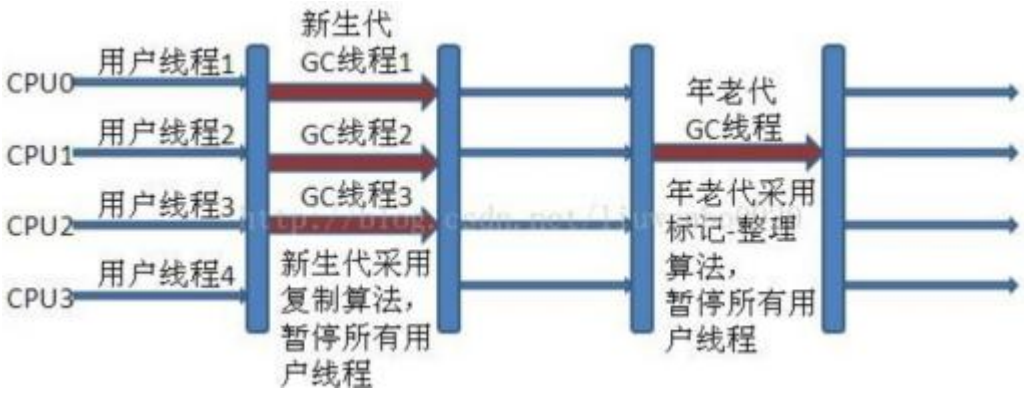
Serial Old 是 Serial 垃圾收集器年老代版本，它同样是个单线程的收集器，使用标记-整理算法，这个收集器也主要是运行在 Client 默认的 java 虚拟机默认的年老代垃圾收集器。在 Server 模式下，主要有两个用途：

1. 在 JDK1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。
2. 作为年老代中使用 CMS 收集器的后备垃圾收集方案。新生代 Serial 与年老代 Serial Old 搭配垃圾收集过程图：



新生代 Parallel Scavenge 收集器与 ParNew 收集器工作原理类似，都是多线程的收集器，都使用的是复制算法，在垃圾收集过程中都需要暂停所有的工作线程。

新生代 ParallelScavenge/ParNew 与年老代 Serial Old 搭配垃圾收集过程图：



Parallel Old 收集器（多线程标记整理算法）

Parallel Old 收集器是 Parallel Scavenge 的年老代版本，使用多线程的标记-整理算法，在 JDK1.6 才开始提供。

在 JDK1.6 之前，新生代使用 ParallelScavenge 收集器只能搭配年老代的 Serial Old 收集器，只能保证新生代的吞吐量优先，无法保证整体的吞吐量，Parallel Old 正是为了在年老代同样提供吞吐量优先的垃圾收集器， 如果系统对吞吐量要求比较高，可以优先考虑新生代 Parallel Scavenge 和年老代 Parallel Old 收集器的搭配策略。

新生代 Parallel Scavenge 和年老代 Parallel Old 收集器搭配运行过程图

CMS 收集器（多线程标记清除算法）

Concurrent mark sweep(CMS)收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间， 和其他年老代使用标记-整理算法不同，它使用多线程的标记-清除算法。最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验。CMS 工作机制相比其他的垃圾收集器来说更复杂。整个过程分为以下 4 个阶段：

初始标记

只是标记一下 GC Roots 能直接关联的对象，速度很快，仍然需要暂停所有

的工作线程。

并发标记

进行 GC Roots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程。

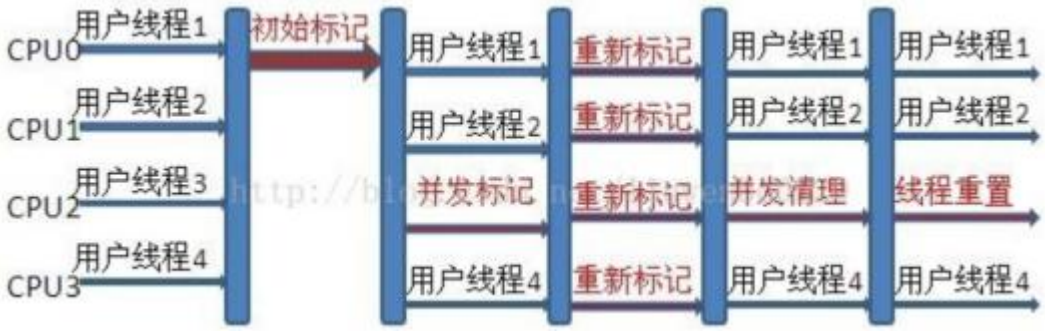
重新标记

为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，仍然需要暂停所有的工作线程。

并发清除

清除 GC Roots 不可达对象，和用户线程一起工作，不需要暂停工作线程。

由于耗时最长的并发标记和并发清除过程中，垃圾收集线程可以和用户现在一起并发工作， 所以总体上来看 CMS 收集器的内存回收和用户线程是一起并发地执行。CMS 收集器工作过程



G1 收集器

Garbage first 垃圾收集器是目前垃圾收集器理论发展的最前沿成果，相比与 CMS 收集器， G1 收集器两个最突出的改进是：

1.基于标记-整理算法，不产生内存碎片。

2.可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。

G1 收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区

域，并且跟踪这些区域的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间， 优先回收垃圾最多的区域。区域划分和优先级区域回收机制，确保 G1 收集器可以在有限时间获得最高的垃圾收集效率

新生代垃圾回收器和老年代垃圾回收器都有哪些？有什么区别？

- 新生代回收器：Serial、ParNew、Parallel Scavenge
- 老年代回收器：Serial Old、Parallel Old、CMS
- 整堆回收器：G1

新生代垃圾回收器一般采用的是复制算法，复制算法的优点是效率高，缺点是内存利用率低；老年代回收器一般采用的是标记-整理的算法进行垃圾回收。

简述分代垃圾回收器是怎么工作的？

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

- 把 Eden + From Survivor 存活的对象放入 To Survivor 区；
- 清空 Eden 和 From Survivor 分区；
- From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

老生代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程。

什么时候会触发 FullGC?

除直接调用 System.gc 外，触发 Full GC 执行的情况有如下四种。

1.旧生代空间不足

老生代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行 Full GC 后空间仍然不足，则抛出如下错误：

java.lang.OutOfMemoryError: Java heap space

为避免以上两种状况引起的 FullGC，调优时应尽量做到让对象在 Minor GC 阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。

2.Permanet Generation 空间满

PermanetGeneration 中存放的为一些 class 的信息等，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation

可能会被占满，在未配置为采用 CMS GC 的情况下会执行 Full GC。如果经过 Full GC 仍然回收不了，那么 JVM 会抛出如下错误信息：

java.lang.OutOfMemoryError: PermGen space

为避免 Perm Gen 占满造成 Full GC 现象，可采用的方法为增大 Perm Gen 空间或转为使用 CMS GC。

3.CMS GC 时出现 promotion failed 和 concurrent mode failure

对于采用 CMS 进行老生代 GC 的程序而言，尤其要注意 GC 日志中是否有 promotion failed 和 concurrent mode failure 两种状况，当这两种状况出现时可能会触发 Full GC。

promotionfailed 是在进行 Minor GC 时，survivor space 放不下、对象只能放入老生代，而此时老生代也放不下造成的；concurrent mode failure 是在执行 CMS GC 的过程中同时有对象要放入老生代，而此时老生代空间不足造成的。

应对措施为：增大 survivorspace、老生代空间或调低触发并发 GC 的比率，但在 JDK 5.0+、6.0+的版本中有可能会由于 JDK 的 bug29 导致 CMS 在 remark 完毕后很久才触发 sweeping 动作。对于这种状况，可通过设置 -XX:CMSMaxAbortablePrecleanTime=5（单位为 ms）来避免。

4.统计得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间

这是一个较为复杂的触发情况，Hotspot 为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行 Minor GC 时，做了一个判断，如果之前统计所得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发 Full GC。

例如程序第一次触发 MinorGC 后，有 6MB 的对象晋升到旧生代，那么当下一次 Minor GC 发生时，首先检查旧生代的剩余空间是否大于 6MB，如果小于 6MB，则执行 Full GC。

当新生代采用 PSGC 时，方式稍有不同，PS GC 是在 Minor GC 后也会检查，例如上面的例子中第一次 Minor GC 后，PS GC 会检查此时旧生代的剩余空间

是否大于 6MB，如小于，则触发对旧生代的回收。除了以上 4 种状况外，对于使用 RMI 来进行 RPC 或管理的 Sun JDK 应用而言，默认情况下会一小时执行一次 Full GC。可通过在启动时通过-java-Dsun.rmi.dgc.client.gcInterval=3600000 来设置 Full GC 执行的间隔时间或通过-XX:+ DisableExplicitGC 来禁止 RMI 调用 System.gc

内存分配策略

简述 java 内存分配与回收策率以及 Minor GC 和 Major GC

所谓自动内存管理，最终要解决的也就是内存分配和内存回收两个问题。前面我们介绍了内存回收，这里我们再来聊聊内存分配。

对象的内存分配通常是在 Java 堆上分配（随着虚拟机优化技术的诞生，某些场景下也会在栈上分配，后面会详细介绍），对象主要分配在新生代的 Eden 区，如果启动了本地线程缓冲，将按照线程优先在 TLAB 上分配。少数情况下也会直接在老年代上分配。总的来说分配规则不是百分百固定的，其细节取决于哪一种垃圾收集器组合以及虚拟机相关参数有关，但是虚拟机对于内存的分配还是会遵循以下几种「普世」规则：

对象优先在 Eden 区分配

多数情况，对象都在新生代 Eden 区分配。当 Eden 区分配没有足够的空间进行分配时，虚拟机将会发起一次 Minor GC。如果本次 GC 后还是没有足够的空间，则将启用分配担保机制在老年代中分配内存。

这里我们提到 Minor GC，如果你仔细观察过 GC 日常，通常我们还能从日志中发现 Major GC/Full GC。

- **Minor GC** 是指发生在新生代的 GC，因为 Java 对象大多都是朝生夕死，所有 Minor GC 非常频繁，一般回收速度也非常快；
- **Major GC/Full GC** 是指发生在老年代的 GC，出现了 Major GC 通常会伴随至少一次 Minor GC。Major GC 的速度通常会比 Minor GC 慢 10 倍以上。

大对象直接进入老年代

所谓大对象是指需要大量连续内存空间的对象，频繁出现大对象是致命的，会导致在内存还有不少空间的情况下提前触发 GC 以获取足够的连续空间来安置新对象。

前面我们介绍过新生代使用的是标记-清除算法来处理垃圾回收的，如果大对象直接在新生代分配就会导致 Eden 区和两个 Survivor 区之间发生大量的内存复制。因此对于大对象都会直接在老年代进行分配。

长期存活对象将进入老年代

虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须判断哪些对象应该放在新生代，哪些对象应该放在老年代。因此虚拟机给每个对象定义了一个对象年龄的计数器，如果对象在 Eden 区出生，并且能够被 Survivor 容纳，将被移动到 Survivor 空间中，这时设置对象年龄为 1。对象在 Survivor

区中每「熬过」一次 Minor GC 年龄就加 1，当年龄达到一定程度（默认 15）就会被晋升到老年代。

对象分配规则

1. 对象优先分配在 Eden 区，如果 Eden 区没有足够的空间时，虚拟机执行一次 Minor GC。
2. 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在 Eden 区和两个 Survivor 区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
3. 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了 1 次 Minor GC 那么对象会进入 Survivor 区，之后每经过一次 Minor GC 那么对象的年龄加 1，知道达到阀值对象进入老年区。
4. 动态判断对象的年龄。如果 Survivor 区中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。
5. 空间分配担保。每次进行 Minor GC 时，JVM 会计算 Survivor 区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次 Full GC，如果小于检查 HandlePromotionFailure 设置，如果 true 则只进行 Monitor GC，如果 false 则进行 Full GC

虚拟机类加载机制

简述 java 类加载机制?

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验，解析和初始化，最终形成可以被虚拟机直接使用的 java 类型。

描述一下 JVM 加载 Class 文件的原理机制

Java 中的所有类，都需要由类加载器装载到 JVM 中才能运行。类加载器本身也是一个类，而它的工作就是把 class 文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，像是反射，就需要显式的加载所需要的类。

类装载方式，有两种 ：

1.隐式装载， 程序在运行过程中当碰到通过 new 等方式生成对象时，隐式调用类装载器加载对应的类到 jvm 中，

2.显式装载， 通过 class.forname()等方法，显式加载需要的类

Java 类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类(像是基类)完全加载到 jvm 中，至于其他类，则在需要的时候才加载。这当然就是为了节省内存开销。

描述一下 JVM 加载 class 文件的原理机制

JVM 中类的装载是由类加载器 (ClassLoader) 和它的子类来实现的，Java 中的类加载器是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入类文件中的类。

由于 Java 的跨平台性，经过编译的 Java 源程序并不是一个可执行程序，而是一个或多个类文件。当 Java 程序需要使用某个类时，JVM 会确保这个类

已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class 文件中的数据读入到内存中，通常是创建一个字节数组读入.class 文件，然后产生与所加载类对应
的 Class 对象。

加载完成后，Class 对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后 JVM 对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。

类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader 的子类）。

从 Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM 更好的保证了 Java 平台的安全性，在该机制中，JVM 自带的 Bootstrap 是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM 不会向 Java 程序提供对 Bootstrap 的引用。下面是关于几个类
加载器的说明：

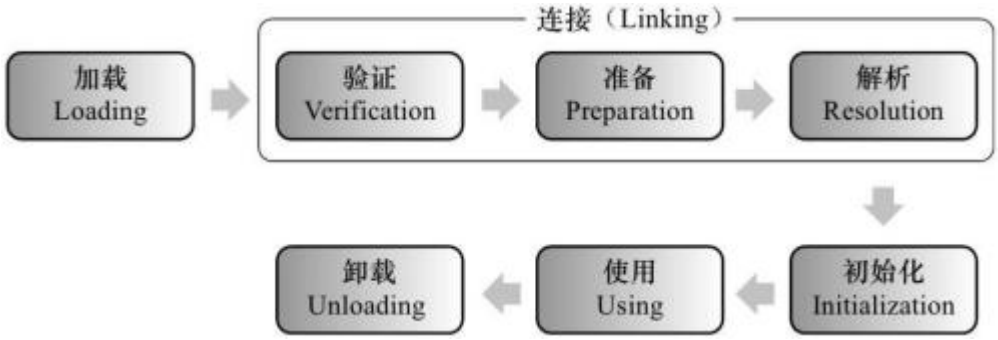
1. Bootstrap：一般用本地代码实现，负责加载 JVM 基础核心类库（rt.jar）；

2. Extension：从 java.ext.dirs 系统属性所指定的目录中加载类库，它的父加载器是 Bootstrap；

3. System：又叫应用类加载器，其父类是 Extension。它是应用最广泛的类加载器。它从环境变量 classpath 或者系统属性 java.class.path 所指定的目录中记载类，是用户自定义加载器的默认父加载器。

JVM 类加载机制

JVM 类加载机制分为五个部分：加载，验证，准备，解析，初始化，下面我们就分别来看一下这五个过程。



加载

加载是类加载过程中的一个阶段， 这个阶段会在内存中生成一个代表这个类的 java.lang.Class 对象， 作为方法区这个类的各种数据的入口。注意这里不一定非得要从一个 Class 文件获取，这里既可以从 ZIP 包中读取（比如从 jar 包和 war 包中读取），也可以在运行时计算生成（动态代理），也可以由其它文件生成（比如将 JSP 文件转换成对应的 Class 类）。

验证

这一阶段的主要目的是为了确保 Class 文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

准备

准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使用的内存空间。注意这里所说的初始值概念，比如一个类变量定义为：

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080， 将 v 赋值为 8080 的 put static 指令是程序被编译后， 存放于类构造器方法之中。

但是注意如果声明为：public static final int v = 8080;

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的：

```
public static int v = 8080;
```

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080， 将 v 赋值为 8080 的 put static 指令是程序被编译后， 存放于类构造器方法之中。但是注意如果声明为：

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v

赋值为 8080。解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的：

```
public static final int v = 8080;
```

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的：

1. CONSTANT_Class_info
2. CONSTANT_Field_info
3. CONSTANT_Method_info

等类型的常量。

符号引用

符号引用与虚拟机实现的布局无关， 引用的目标并不一定要已经加载到内存中。 各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在 Java 虚拟机规范的 Class 文件格式中。

直接引用

直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

初始化

初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载器以外，其它操作都由 JVM 主导。到了初始阶段，才开始

真正执行类中定义的 Java 程序代码。

类构造器

初始化阶段是执行类构造器方法的过程。 方法是由编译器自动收集类中的类变量的赋值操作和静态语句块中的语句合并而成的。虚拟机会保证子方法执行之前，父类的方法已经执行完毕， 如果一个类中没有对静态变量赋值也没有静态语句块，那么编译器可以不为这个类生成() 方法。注意以下几种情况不会执行类初始化：

1. 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
2. 定义对象数组，不会触发该类的初始化。
3. 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。
4. 通过类名获取 Class 对象，不会触发类的初始化。
5. 通过 Class.forName 加载指定类时，如果指定参数 initialize 为 false 时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
6. 通过 ClassLoader 默认的 loadClass 方法，也不会触发初始化动作。

什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。

2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

4. 用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现。

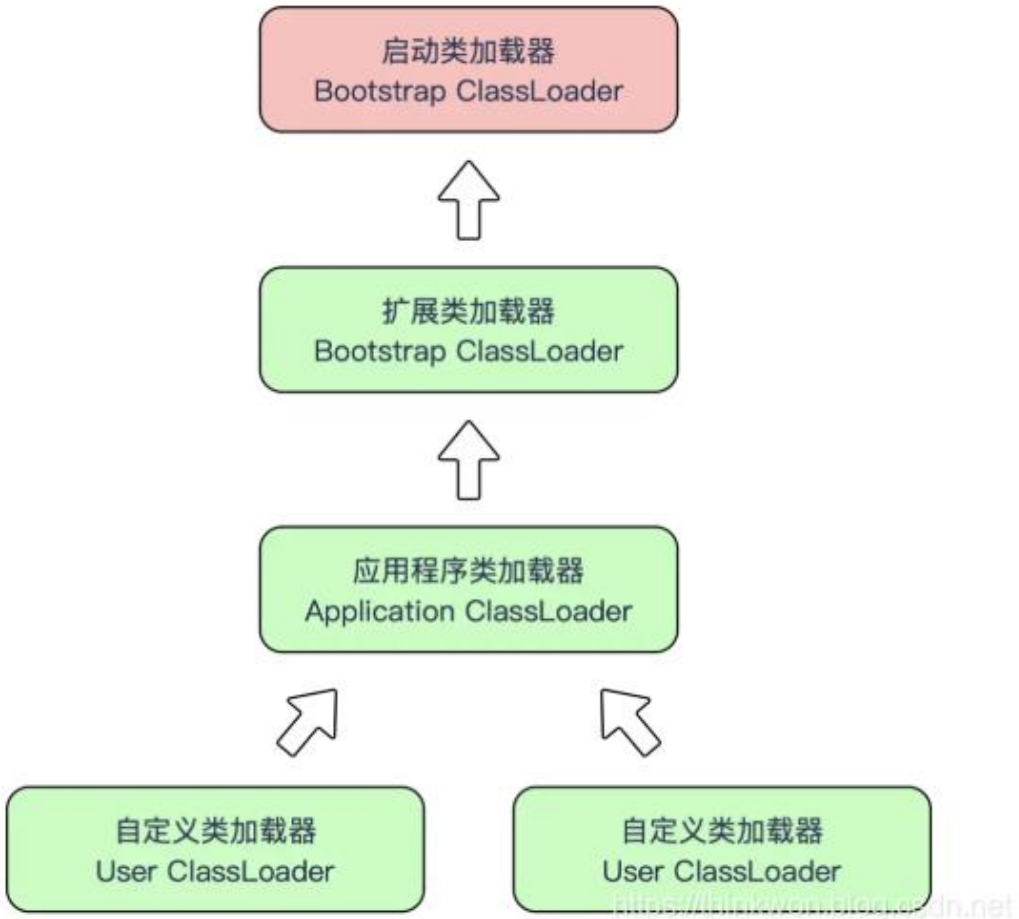
说一下类装载的执行过程?

类装载分为以下 5 个步骤：

- 加载：根据查找路径找到相应的 class 文件然后导入；
- 验证：检查加载的 class 文件的正确性；
- 准备：给类中的静态变量分配内存空间；
- 解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址；
- 初始化：对静态变量和静态代码块执行初始化工作。

什么是双亲委派模型?

在介绍双亲委派模型之前先说下类加载器。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立在 JVM 中的唯一性，每一个类加载器，都有一个独立的类名称空间。类加载器就是根据指定全限定名称将 class 文件加载到 JVM 内存，然后再转化为 class 对象。



类加载器分类：

- 启动类加载器（Bootstrap ClassLoader），是虚拟机自身的一部分，用来加载 Java_HOME/lib/目录中的，或者被 -Xbootclasspath 参数所指定的路径中并且被虚拟机识别的类库；
- 其他类加载器：

- 扩展类加载器(Extension ClassLoader):负责加载\lib\ext 目录或 Java. ext. dirs 系统变量指定的路径中的所有类库;
- 应用程序类加载器（Application ClassLoader)。负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

双亲委派模型：如果一个类加载器收到了类加载的请求，它首先不会自己去加载这个类，而是把这个请求委派给父类加载器去完成，每一层的类加载器都是如此，这样所有的加载请求都会被传送到顶层的启动类加载器中，只有当父加载无法完成加载请求（它的搜索范围中没找到所需的类）时，子加载器才会尝试去加载类。

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

简单说说你了解的类加载器，可以打破双亲委派么，怎么打破。

思路： 先说明一下什么是类加载器，可以给面试官画个图，再说一下类加载器存在的意义，说一下双亲委派模型，最后阐述怎么打破双亲委派模型。

参考的答案：

1. 什么是类加载器?

类加载器 就是根据指定全限定名称将 class 文件加载到 JVM 内存,转为 Class 对象。

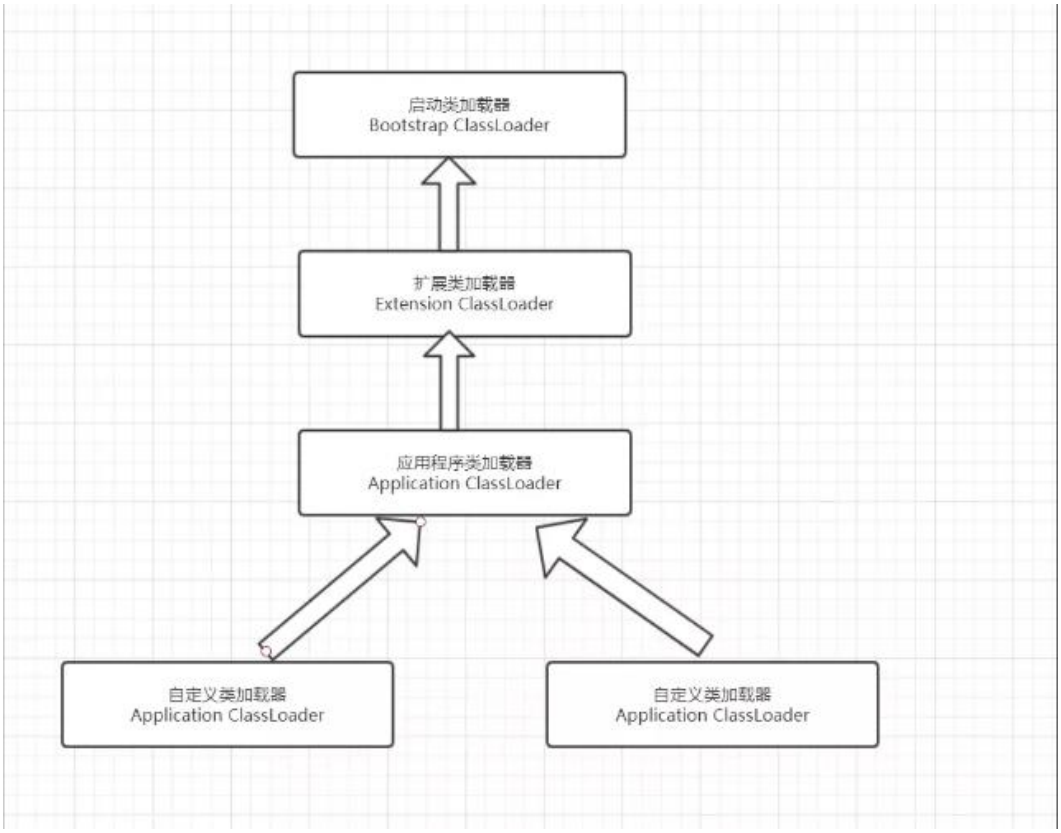
- 启动类加载器（Bootstrap ClassLoader）：由 C++语言实现（针对 HotSpot）,负责将存放在<JAVA_HOME>\lib 目录或 -Xbootclasspath 参数指定的路径中的类库加载到内存中。
- 其他类加载器：由 Java 语言实现，继承自抽象类 ClassLoader。如：
    - 扩展类加载器（Extension ClassLoader）：负责加载 <JAVA_HOME>\lib\ext 目录或 java.ext.dirs 系统变量指定的路径中的所有类库。
    - 应用程序类加载器（Application ClassLoader）。负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

2）双亲委派模型

双亲委派模型工作过程是：

如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即 ClassNotFoundException），子加载器才会尝试自己去加载。

双亲委派模型图：



3）为什么需要双亲委派模型？

在这里，先想一下，如果没有双亲委派，那么用户是不是可以**自己定义一个 java.lang.Object 的同名类**，**java.lang.String 的同名类**，并把它放到 ClassPath 中,那么**类之间的比较结果及类的唯一性将无法保证**，因此，为什么需要双亲委派模型？**防止内存中出现多份同样的字节码**

4）怎么打破双亲委派模型？

打破双亲委派机制则不仅**要继承 ClassLoader** 类，还要**重写 loadClass 和 findClass** 方法。

什么是 Java 虚拟机？为什么 Java 被称作是“平台无关的编程语言”？

Java 虚拟机是一个可以执行 Java 字节码的虚拟机进程。Java 源文件被编译成能被 Java 虚拟机执行的字节码文件。 Java 被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。Java 虚拟机让这个变为可能，因为它知道底层硬件平台的 指令长度和其他特性。

JVM 调优

说一下 JVM 调优的工具?

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 jconsole 和 jvisualvm 这两款视图监控工具。

- jconsole：用于对 JVM 中的内存、线程和类等进行监控；
- jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序死锁、监控内存的变化、gc 变化等。

常用的 JVM 调优的参数都有哪些?

- -Xms2g：初始化推大小为 2g；
- -Xmx2g：堆最大内存为 2g；
- -XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4；
- -XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2；
- –XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合；

- -XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组合；
- -XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合；
- -XX:+PrintGC：开启打印 gc 信息；
- -XX:+PrintGCDetails：打印 gc 详细信息。

调优命令有哪些?

Sun JDK 监控和故障处理命令有 jps jstat jmap jhat jstack jinfo

1. jps，JVM Process Status Tool,显示指定系统内所有的 HotSpot 虚拟机进程。
2. jstat，JVM statistics Monitoring 是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据。
3. jmap，JVM Memory Map 命令用于生成 heap dump 文件
4. jhat，JVM Heap Analysis Tool 命令是与 jmap 搭配使用，用来分析 jmap 生成的 dump，jhat 内置了一个微型的 HTTP/HTML 服务器，生成 dump 的分析结果后，可以在浏览器中查看
5. jstack，用于生成 java 虚拟机当前时刻的线程快照。
6. jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数

调优工具

常用调优工具分为两类,jdk 自带监控工具：jconsole 和 jvisualvm，第三方有：MAT(Memory AnalyzerTool)、GChisto。

1. jconsole，Java Monitoring and Management Console 是从 java5 开始，在 JDK 中自带的 java 监控和管理控制台，用于对 JVM 中内存， 线程和类等的监控
2. jvisualvm，jdk 自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC 变化等。
3. MAT，Memory Analyzer Tool，一个基于 Eclipse 的内存分析工具，是一个快速、功能丰富的 Javaheap 分析工具，它可以帮助我们查找内存泄漏和减少内存消耗
4. GChisto，一款专业分析 gc 日志的工具

说说你知道的几种主要的 JVM 参数

思路： 可以说一下堆栈配置相关的，垃圾收集器相关的，还有一下辅助信息相关的。

参考答案：

1）堆栈配置相关

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:MaxPermSize=16m -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxTenuringThreshold=0
```

-Xmx3550m： 最大堆大小为 3550m。

-Xms3550m： 设置初始堆大小为 3550m。

-Xmn2g： 设置年轻代大小为 2g。

-Xss128k： 每个线程的堆栈大小为 128k。

-XX:MaxPermSize： 设置持久代大小为 16m

-XX:NewRatio=4: 设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。

-XX:SurvivorRatio=4： 设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6

-XX:MaxTenuringThreshold=0： 设置垃圾最大年龄。如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。

2）垃圾收集器相关

```
-XX:+UseParallelGC-XX:ParallelGCThreads=20-XX:+UseConcMarkSweepGC  -XX:C
MSFullGCsBeforeCompaction=5-XX:+UseCMSCompactAtFullCollection：
```

-XX:+UseParallelGC： 选择垃圾收集器为并行收集器。

-XX:ParallelGCThreads=20： 配置并行收集器的线程数

-XX:+UseConcMarkSweepGC： 设置年老代为并发收集。

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection： 打开对年老代的压缩。可能会影响性能，但是可以消除碎片

3）辅助信息相关

```
-XX:+PrintGC-XX:+PrintGCDetails
```

-XX:+PrintGC 输出形式:

[GC 118250K->113543K(130112K), 0.0094143 secs] [Full GC 121376K->10414K(130112K), 0.0650971 secs]

-XX:+PrintGCDetails 输出形式:

[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs] [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs

怎么打出线程栈信息。

思路： 可以说一下 jps，top ，jstack 这几个命令，再配合一次排查线上问题进行解答。

参考答案：

- 输入 jps，获得进程号。
- top -Hp pid 获取本进程中所有线程的 CPU 耗时性能
- jstack pid 命令查看当前 java 进程的堆栈状态
- 或者 jstack -l > /tmp/output.txt 把堆栈信息打到一个 txt 文件。
- 可以使用 fastthread 堆栈定位，fastthread.io/

Spring 面试题 专题部分

Spring 框架中都用到了哪些设计模式?

1. 工厂模式：BeanFactory 就是简单工厂模式的体现，用来创建对象的实例；
2. 单例模式：Bean 默认为单例模式。
3. 代理模式：Spring 的 AOP 功能用到了 JDK 的动态代理和 CGLIB 字节码生成技术；
4. 模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。
5. 观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如 Spring 中 listener 的实现–ApplicationListener。

详细讲解一下核心容器（spring context 应用上下文）模块

这是基本的 Spring 模块，提供 spring 框架的基础功能，BeanFactory 是 任何以 spring 为基础的应用的核心。Spring 框架建立在此模块之上，它使 Spring 成为一个容器。

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从真正的应用代码中分离。最常用的就是 org.springframework.beans.factory.xml.XmlBeanFactory ，它根据 XML 文件中的定义加载 beans。该容器从 XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

Spring 框架中有哪些不同类型的事件

Spring 提供了以下 5 种标准的事件：

1. 上下文更新事件（ContextRefreshedEvent）：在调用 ConfigurableApplicationContext 接口中的 refresh()方法时被触发。

2. 上下文开始事件（ContextStartedEvent）：当容器调用 ConfigurableApplicationContext 的 Start()方法开始/重新开始容器时触发该事件。

3. 上下文停止事件（ContextStoppedEvent）：当容器调用 ConfigurableApplicationContext 的 Stop()方法停止容器时触发该事件。

4. 上下文关闭事件（ContextClosedEvent）：当 ApplicationContext 被关闭时触发该事件。容器被关闭时，其管理的所有单例 Bean 都被销毁。

5. 请求处理事件（RequestHandledEvent）：在 Web 应用中，当一个 http 请求（request）结束触发该事件。如果一个 bean 实现了 ApplicationListener 接口，当一个 ApplicationEvent 被发布以后，bean 会自动被通知。

Spring 应用程序有哪些不同组件?

Spring 应用一般有以下组件:

- 接口 - 定义功能。
- Bean 类 - 它包含属性，setter 和 getter 方法，函数等。
- Bean 配置文件 - 包含类的信息以及如何配置它们。
- Spring 面向切面编程（AOP） - 提供面向切面编程的功能。
- 用户程序 - 它使用接口。

使用 Spring 有哪些方式?

使用 Spring 有以下方式:

- 作为一个成熟的 Spring Web 应用程序。
- 作为第三方 Web 框架，使用 Spring Frameworks 中间层。
- 作为企业级 Java Bean，它可以包装现有的 POJO（Plain Old Java Objects）。
- 用于远程使用。

Spring 控制反转(IOC)

什么是 Spring IOC 容器?

控制反转即 IoC (Inversion of Control)，它把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的“控制反转”概念就是对组件对象控制权的转移，从程序代码本身转移到了外部容器。

Spring IOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

控制反转(IoC)有什么作用

- 管理对象的创建和依赖关系的维护。对象的创建并不是一件简单的事，在对象关系比较复杂时，如果依赖关系需要程序猿来维护的话，那是相当头疼的
- 解耦，由容器去维护具体的对象
- 托管了类的产生过程，比如我们需要在类的产生过程中做一些处理，最直接的例子就是代理，如果有容器程序可以把这部分处理交给容器，应用程序则无需去关心类是如何完成代理的

IOC 的优点是什么?

- IOC 或 依赖注入把应用的代码量降到最低。
- 它使应用容易测试，单元测试不再需要单例和 JNDI 查找机制。
- 最小的代价和最小的侵入性使松散耦合得以实现。
- IOC 容器支持加载服务时的饿汉式初始化和懒加载。

Spring IoC 的实现机制

Spring 中的 IoC 的实现原理就是工厂模式加反射机制。

示例：

```
interface Fruit {
```

```
    public abstract void eat();
}
class Apple implements Fruit {
    public void eat(){
        System.out.println("Apple");
    }
}
class Orange implements Fruit {
    public void eat(){
        System.out.println("Orange");
    }
}
class Factory {
    public static Fruit getInstance(String ClassName) {
        Fruit f=null;
        try {
            f=(Fruit)Class.forName(ClassName).newInstance();
        } catch (Exception e) {
```

```
                e.printStackTrace();

            }

            return f;

        }

    }

    class Client {

        public static void main(String[] a) {

            Fruit f=Factory.getInstance("io.github.dunwu.spring.Apple");

            if(f!=null){

                f.eat();

            }

        }

    }
```

Spring 的 IoC 支持哪些功能

Spring 的 IoC 设计支持以下功能：

- 依赖注入
- 依赖检查
- 自动装配

- 支持集合
- 指定初始化方法和销毁方法
- 支持回调某些方法（但是需要实现 Spring 接口，略有侵入）

其中，最重要的就是依赖注入，从 XML 的配置上说，即 ref 标签。对应 Spring RuntimeBeanReference 对象。

对于 IoC 来说，最重要的就是容器。容器管理着 Bean 的生命周期，控制着 Bean 的依赖注入。

BeanFactory 和 ApplicationContext 有什么区别?

BeanFactory 和 ApplicationContext 是 Spring 的两大核心接口，都可以当做 Spring 的容器。其中 ApplicationContext 是 BeanFactory 的子接口。

依赖关系

BeanFactory：是 Spring 里面最底层的接口，包含了各种 Bean 的定义，读取 bean 配置文档，管理 bean 的加载、实例化，控制 bean 的生命周期，维护 bean 之间的依赖关系。

ApplicationContext 接口作为 BeanFactory 的派生，除了提供 BeanFactory 所具有的功能外，还提供了更完整的框架功能：

- 继承 MessageSource，因此支持国际化。
- 统一的资源文件访问方式。
- 提供在监听器中注册 bean 的事件。

- 同时加载多个配置文件。
- 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的 web 层。

加载方式

BeanFactroy 采用的是延迟加载形式来注入 Bean 的，即只有在使用到某个 Bean 时(调用 getBean())，才对该 Bean 进行加载实例化。这样，我们就不能发现一些存在的 Spring 的配置问题。如果 Bean 的某一个属性没有注入，BeanFacotry 加载后，直至第一次使用调用 getBean 方法才会抛出异常。

ApplicationContext，它是在容器启动时，一次性创建了所有的 Bean。这样，在容器启动时，我们就可以发现 Spring 中存在的配置错误，这样有利于检查所依赖属性是否注入。ApplicationContext 启动后预载入所有的单实例 Bean，通过预载入单实例 bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

相对于基本的 BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置 Bean 较多时，程序启动较慢。

创建方式

BeanFactory 通常以编程的方式被创建，ApplicationContext 还能以声明的方式创建，如使用 ContextLoader。

注册方式

BeanFactory 和 ApplicationContext 都支持 BeanPostProcessor、BeanFactoryPostProcessor 的使用，但两者之间的区别是：BeanFactory 需要手动注册，而 ApplicationContext 则是自动注册。

Spring 如何设计容器的，BeanFactory 和 ApplicationContext 的关系详解

Spring 作者 Rod Johnson 设计了两个接口用以表示容器。

- BeanFactory
- ApplicationContext

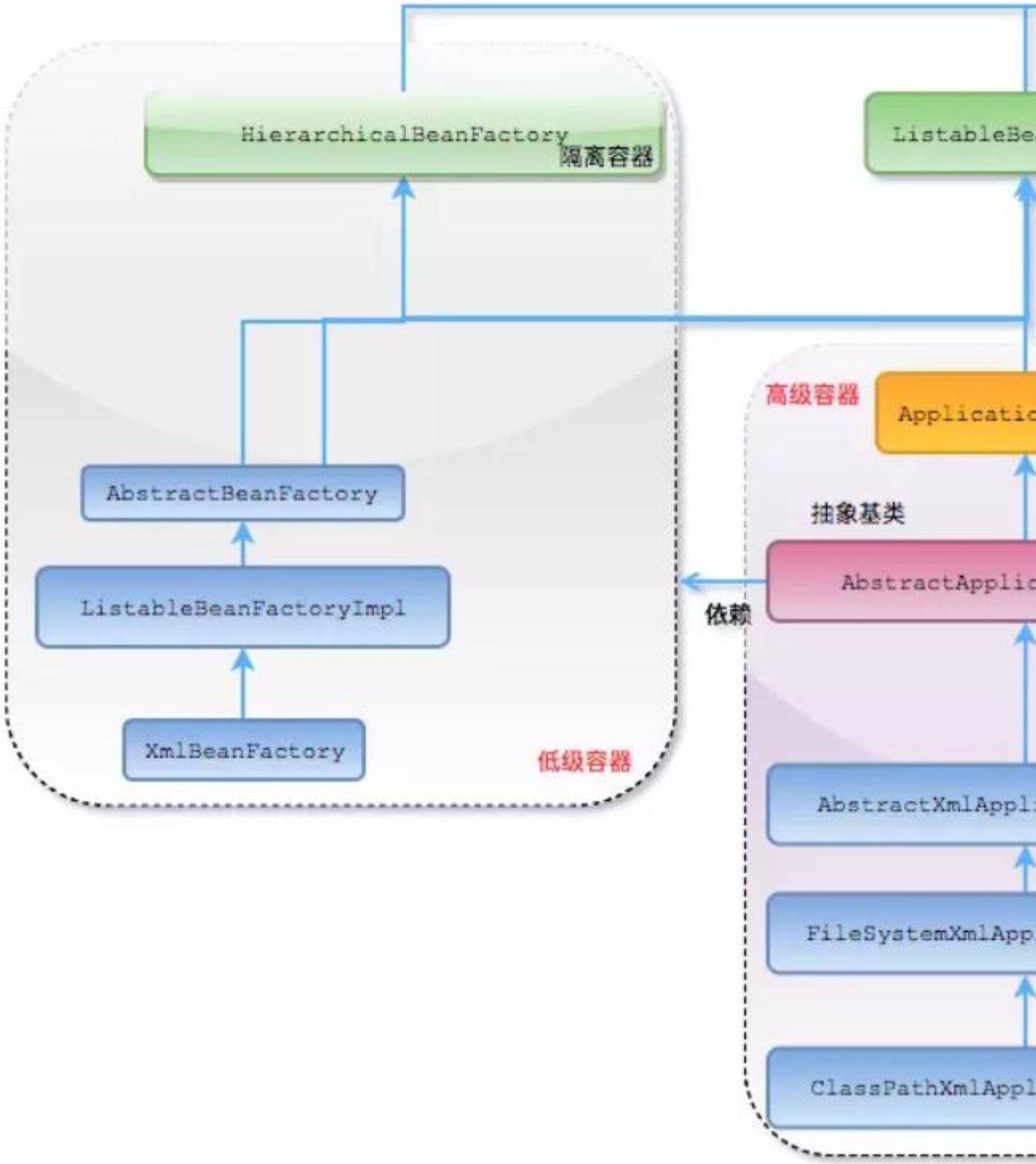
BeanFactory 简单粗暴，可以理解为就是个 HashMap，Key 是 BeanName，Value 是 Bean 实例。通常只提供注册（put），获取（get）这两个功能。我们可以称之为 **“低级容器”**。

ApplicationContext 可以称之为**“高级容器”**。因为他比 BeanFactory 多了更多的功能。他继承了多个接口。因此具备了更多的功能。例如资源的获取，支持多种消息（例如 JSP tag 的支持），对 BeanFactory 多了工具级别的支持等待。所以你看他的名字，已经不是 BeanFactory 之类的工厂了，而是 “应用上下文”， 代表着整个大容器的所有功能。该接口定义了一个 refresh 方法，此方法是所有阅读 Spring 源码的人的最熟悉的方法，用于刷新整个容器，即重新加载/刷新所有的 bean。

当然，除了这两个大接口，还有其他的辅助接口，这里就不介绍他们了。

BeanFactory 和 ApplicationContext 的关系

为了更直观的展示 “低级容器” 和 “高级容器” 的关系，这里通过常用的 ClassPathXmlApplicationContext 类来展示整个容器的层级 UML 关系。



有点复杂？ 先不要慌，我来解释一下。

最上面的是 BeanFactory，下面的 3 个绿色的，都是功能扩展接口，这里就不展开讲。

看下面的隶属 ApplicationContext 粉红色的 “高级容器”，依赖着 “低级容器”，这里说的是依赖，不是继承哦。他依赖着 “低级容器” 的 getBean 功能。而高级容器有更多的功能：支持不同的信息源头，可以访问文件资源，支持应用事件（Observer 模式）。

通常用户看到的就是 “高级容器”。 但 BeanFactory 也非常够用啦！

左边灰色区域的是 “低级容器”， 只负载加载 Bean，获取 Bean。容器其他的高级功能是没有的。例如上图画的 refresh 刷新 Bean 工厂所有配置，生命周期事件回调等。

小结

说了这么多，不知道你有没有理解 Spring IoC？ 这里小结一下：IoC 在 Spring 里，只需要低级容器就可以实现，2 个步骤：

1. 加载配置文件，解析成 BeanDefinition 放在 Map 里。
2. 调用 getBean 的时候，从 BeanDefinition 所属的 Map 里，拿出 Class 对象进行实例化，同时，如果有依赖关系，将递归调用 getBean 方法 —— 完成依赖注入。

上面就是 Spring 低级容器（BeanFactory）的 IoC。

至于高级容器 ApplicationContext，他包含了低级容器的功能，当他执行 refresh 模板方法的时候，将刷新整个容器的 Bean。同时其作为高级容器，包含了太多的功能。一句话，他不仅仅是 IoC。他支持不同信息源头，支持 BeanFactory 工具类，支持层级容器，支持访问文件资源，支持事件发布通知，支持接口回调等等。

ApplicationContext 通常的实现是什么?

FileSystemXmlApplicationContext：此容器从一个 XML 文件中加载 beans 的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。

ClassPathXmlApplicationContext：此容器也从一个 XML 文件中加载 beans 的定义，这里，你需要正确设置 classpath 因为这个容器将在 classpath 里找 bean 配置。

WebXmlApplicationContext：此容器加载一个 XML 文件，此文件定义了一个 WEB 应用的所有 bean。

什么是 Spring 的依赖注入?

控制反转 IoC 是一个很大的概念，可以用不同的方式来实现。其主要实现方式有两种：依赖注入和依赖查找

依赖注入：相对于 IoC 而言，依赖注入(DI)更加准确地描述了 IoC 的设计理念。所谓依赖注入（Dependency Injection），即组件之间的依赖关系由容器在应用系统运行期来决定，也就是由容器动态地将某种依赖关系的目标对象实例注入到应用系统中的各个关联的组件之中。组件不做定位查询，只提供普通的 Java 方法让容器去决定依赖关系。

依赖注入的基本原则

依赖注入的基本原则是：应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由 IoC 容器负责，“查找资源”的逻辑应该从应用组件的代码中抽取出来，交给 IoC 容器负责。容器全权负责组件的装配，它会把符合依赖关系的对象通过属性（JavaBean 中的 setter）或者是构造器传递给需要的对象。

依赖注入有什么优势

依赖注入之所以更流行是因为它是一种更可取的方式：让容器全权负责依赖查询，受管组件只需要暴露 JavaBean 的 setter 方法或者带参数的构造器或者接口，使容器可以在初始化时组装对象的依赖关系。其与依赖查找方式相比，主要优势为：

- 查找定位操作与应用代码完全无关。
- 不依赖于容器的 API，可以很容易地在任何容器以外使用应用对象。
- 不需要特殊的接口，绝大多数对象可以做到完全不必依赖容器。

有哪些不同类型的依赖注入实现方式?

依赖注入是时下最流行的 IoC 实现方式，依赖注入分为接口注入（Interface Injection），Setter 方法注入（Setter Injection）和构造器注入（Constructor Injection）三种方式。其中接口注入由于在灵活性和易用性比较差，现在从 Spring4 开始已被废弃。

构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。

Setter 方法注入：Setter 方法注入是容器通过调用无参构造器或无参 static 工厂 方法实例化 bean 之后，调用该 bean 的 setter 方法，即实现了基于 setter 的依赖注入。

构造器依赖注入和 Setter 方法注入的区别

构造函数注入	setter 注入
没有部分注入	有部分注入
不会覆盖 setter 属性	会覆盖 setter 属性

构造函数注入	setter 注入
任意修改都会创建一个新实例	任意修改不会创建一个新实例
适用于设置很多属性	适用于设置少量属性

两种依赖方式都可以使用，构造器注入和 Setter 方法注入。最好的解决方案是用构造器参数实现强制依赖，setter 方法实现可选依赖。

Spring Beans

什么是 Spring beans?

Spring beans 是那些形成 Spring 应用的主干的 java 对象。它们被 Spring IOC 容器初始化，装配，和管理。这些 beans 通过容器中配置的元数据创建。比如，以 XML 文件中 的形式定义。

一个 Spring Bean 定义 包含什么?

一个 Spring Bean 的定义包含容器必知的所有配置元数据，包括如何创建一个 bean，它的生命周期详情及它的依赖。

如何给 Spring 容器提供配置元数据? Spring 有几种配置方式

这里有三种重要的方法给 Spring 容器提供配置元数据。

- XML 配置文件。
- 基于注解的配置。
- 基于 java 的配置。

Spring 配置文件包含了哪些信息

Spring 配置文件是个 XML 文件，这个文件包含了类信息，描述了如何配置它们，以及如何相互调用。

Spring 基于 xml 注入 bean 的几种方式

Set 方法注入；

构造器注入：①通过 index 设置参数的位置；②通过 type 设置参数类型；

静态工厂注入；

实例工厂；

你怎样定义类的作用域?

当定义一个 在 Spring 里，我们还能给这个 bean 声明一个作用域。它可以通过 bean 定义中的 scope 属性来定义。如，当 Spring 要在需要的时候每次生产一个新的 bean 实例，bean 的 scope 属性被指定为 prototype。另一方面，一个 bean 每次使用的时候必须返回同一个实例，这个 bean 的 scope 属性 必须设为 singleton。

解释 Spring 支持的几种 bean 的作用域

Spring 框架支持以下五种 bean 的作用域：

- **singleton :** bean 在每个 Spring ioc 容器中只有一个实例。
- **prototype**：一个 bean 的定义可以有多个实例。
- **request**：每次 http 请求都会创建一个 bean，该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。

- **session**：在一个 HTTP Session 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。

- **global-session**：在一个全局的 HTTP Session 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。

注意： 缺省的 Spring bean 的作用域是 Singleton。使用 prototype 作用域需要慎重的思考，因为频繁创建和销毁 bean 会带来很大的性能开销。

Spring 框架中的单例 bean 是线程安全的吗?

不是，Spring 框架中的单例 bean 不是线程安全的。

spring 中的 bean 默认是单例模式，spring 框架并没有对单例 bean 进行多线程的封装处理。

实际上大部分时候 spring bean 无状态的(比如 dao 类)，所有某种程度上来说 bean 也是安全的，但如果 bean 有状态的话（比如 view model 对象），那就要开发者自己去保证线程安全了，最简单的就是改变 bean 的作用域，把“singleton”变更为“prototype”，这样请求 bean 相当于 new Bean()了，所以就可以保证线程安全了。

- 有状态就是有数据存储功能。

- 无状态就是不会保存数据。

Spring 如何处理线程并发问题?

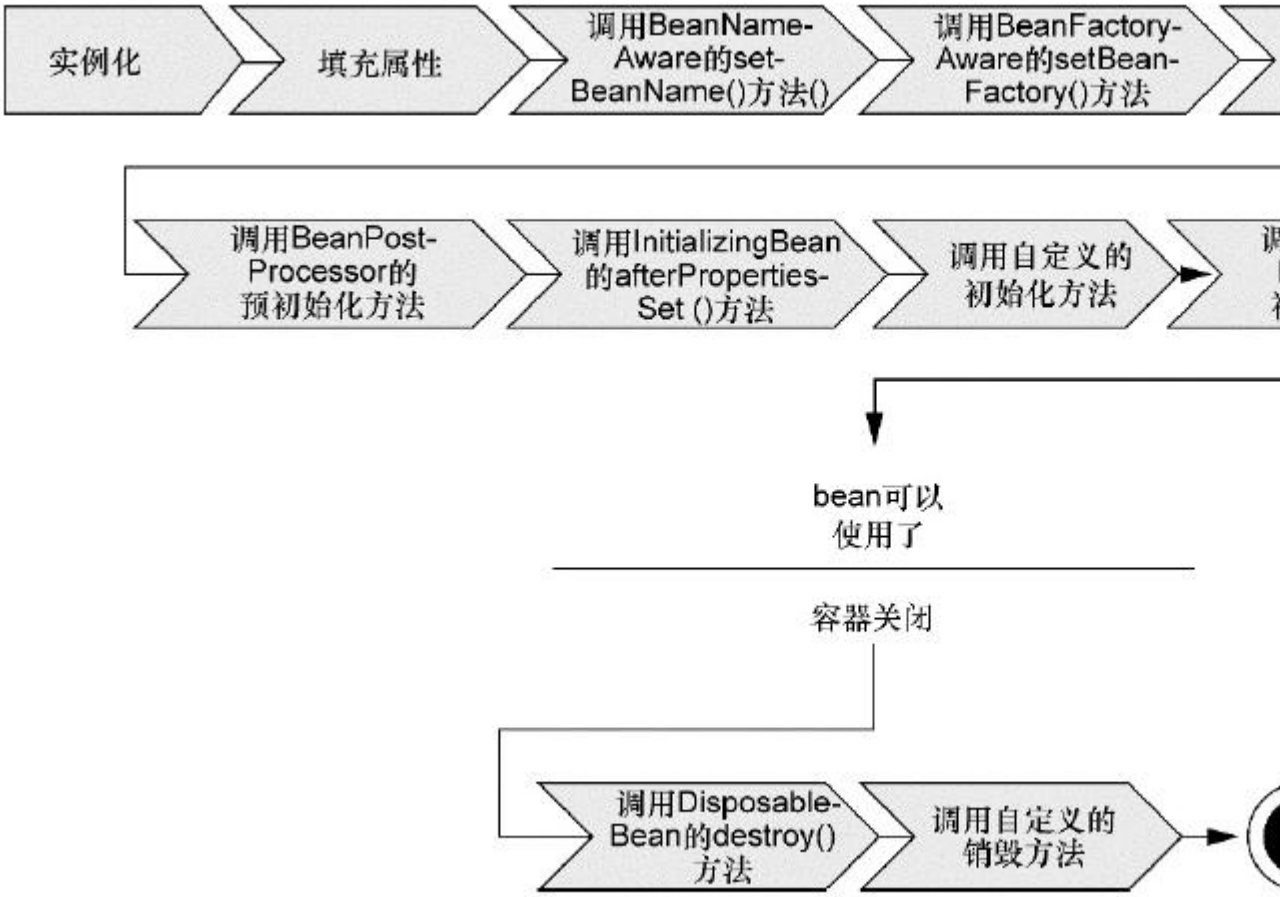
在一般情况下，只有无状态的 Bean 才可以在多线程环境下共享，在 Spring 中，绝大部分 Bean 都可以声明为 singleton 作用域，因为 Spring 对一些 Bean 中非线程安全状态采用 ThreadLocal 进行处理，解决线程安全问题。

ThreadLocal 和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而 ThreadLocal 采用了“空间换时间”的方式。

ThreadLocal 会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal 提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进 ThreadLocal。

解释 Spring 框架中 bean 的生命周期

在传统的 Java 应用中，bean 的生命周期很简单。使用 Java 关键字 new 进行 bean 实例化，然后该 bean 就可以使用了。一旦该 bean 不再被使用，则由 Java 自动进行垃圾回收。相比之下，Spring 容器中的 bean 的生命周期就显得相对复杂多了。正确理解 Spring bean 的生命周期非常重要，因为你或许要利用 Spring 提供的扩展点来自定义 bean 的创建过程。下图展示了 bean 装载到 Spring 应用上下文中的一个典型的生命周期过程。



bean 在 Spring 容器中从创建到销毁经历了若干阶段，每一阶段都可以针对 Spring 如何管理 bean 进行个性化定制。

正如你所见，在 bean 准备就绪之前，bean 工厂执行了若干启动步骤。

我们对上图进行详细描述：

Spring 对 bean 进行实例化；

Spring 将值和 bean 的引用注入到 bean 对应的属性中；

如果 bean 实现了 BeanNameAware 接口，Spring 将 bean 的 ID 传递给 setBean-Name() 方法；

如果 bean 实现了 BeanFactoryAware 接口，Spring 将调用 setBeanFactory()方法，将 BeanFactory 容器实例传入；

如果 bean 实现了 ApplicationContextAware 接口，Spring 将调用 setApplicationContext()方法，将 bean 所在的应用上下文的引用传入进来；

如果 bean 实现了 BeanPostProcessor 接口，Spring 将调用它们的 post-ProcessBeforeInitialization()方法；

如果 bean 实现了 InitializingBean 接口，Spring 将调用它们的 after-PropertiesSet()方法。类似地，如果 bean 使用 initmethod 声明了初始化方法，该方法也会被调用；

如果 bean 实现了 BeanPostProcessor 接口，Spring 将调用它们的 post-ProcessAfterInitialization()方法；

此时，bean 已经准备就绪，可以被应用程序使用了，它们将一直驻留在应用上下文中，直到该应用上下文被销毁；

如果 bean 实现了 DisposableBean 接口，Spring 将调用它的 destroy()接口方法。同样，如果 bean 使用 destroy-method 声明了销毁方法，该方法也会被调用。

现在你已经了解了如何创建和加载一个 Spring 容器。但是一个空的容器并没有太大的价值，在你把东西放进去之前，它里面什么都没有。为了从 Spring 的 DI(依赖注入)中受益，我们必须将应用对象装配进 Spring 容器中。

哪些是重要的 bean 生命周期方法？ 你能重载它们吗？

有两个重要的 bean 生命周期方法，第一个是 setup ， 它是在容器加载 bean 的时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。

bean 标签有两个重要的属性（init-method 和 destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct 和@PreDestroy）。

什么是 Spring 的内部 bean？什么是 Spring inner beans?

在 Spring 框架中，当一个 bean 仅被用作另一个 bean 的属性时，它能被声明为一个内部 bean。内部 bean 可以用 setter 注入“属性”和构造方法注入“构造参数”的方式来实现，内部 bean 通常是匿名的，它们的 Scope 一般是 prototype。

在 Spring 中如何注入一个 java 集合?

Spring 提供以下几种集合的配置元素：

类型用于注入一列值，允许有相同的值。

类型用于注入一组值，不允许有相同的值。

类型用于注入一组键值对，键和值都可以为任意类型。

类型用于注入一组键值对，键和值都只能为 String 类型。

什么是 bean 装配?

装配，或 bean 装配是指在 Spring 容器中把 bean 组装到一起，前提是容器需要知道 bean 的依赖关系，如何通过依赖注入来把它们装配到一起。

什么是 bean 的自动装配?

在 Spring 框架中，在配置文件中设定 bean 的依赖关系是一个很好的机制，Spring 容器能够自动装配相互合作的 bean，这意味着容器不需要和配置，能通过 Bean 工厂自动处理

bean 之间的协作。这意味着 Spring 可以通过向 Bean Factory 中注入的方式自动搞定 bean 之间的依赖关系。自动装配可以设置在每个 bean 上，也可以设定在特定的 bean 上。

解释不同方式的自动装配，spring 自动装配 bean 有哪些方式?

在 spring 中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用 autowire 来配置自动装载模式。

在 Spring 框架 xml 配置中共有 5 种自动装配：

- no：默认的方式是不进行自动装配的，通过手工设置 ref 属性来进行装配 bean。
- byName：通过 bean 的名称进行自动装配，如果一个 bean 的 property 与另一 bean 的 name 相同，就进行自动装配。
- byType：通过参数的数据类型进行自动装配。
- constructor：利用构造函数进行装配，并且构造函数的参数通过 byType 进行装配。
- autodetect：自动探测，如果有构造方法，通过 construct 的方式自动装配，否则使用 byType 的方式自动装配。

使用@Autowired 注解自动装配的过程是怎样的?

使用@Autowired 注解来自动装配指定的 bean。在使用@Autowired 注解之前需要在 Spring 配置文件进行配置，<context:annotation-config />。

在启动 spring IoC 时，容器自动装载了一个 AutowiredAnnotationBeanPostProcessor 后置处理器，当容器扫描到@Autowied、@Resource 或@Inject 时，就会在 IoC 容器自

动查找需要的 bean，并装配给该对象的属性。在使用@Autowired 时，首先在容器中查询对应类型的 bean：

- 如果查询结果刚好为一个，就将该 bean 装配给@Autowired 指定的数据；
- 如果查询的结果不止一个，那么@Autowired 会根据名称来查找；
- 如果上述查找的结果为空，那么会抛出异常。解决方法时，使用 required=false。

自动装配有哪些局限性?

自动装配的局限性是：

重写：你仍需用 和 配置来定义依赖，意味着总要重写自动装配。

基本数据类型：你不能自动装配简单的属性，如基本数据类型，String 字符串，和类。

模糊特性：自动装配不如显式装配精确，如果有可能，建议使用显式装配。

你可以在 Spring 中注入一个 null 和一个空字符串吗?

可以。

Spring 注解

什么是基于 Java 的 Spring 注解配置? 给一些注解的例子

基于 Java 的配置，允许你在少量的 Java 注解的帮助下，进行你的大部分 Spring 配置而非通过 XML 文件。

以@Configuration 注解为例，它用来标记类可以当做一个 bean 的定义，被 Spring IOC 容器使用。

另一个例子是@Bean 注解，它表示此方法将要返回一个对象，作为一个 bean 注册进 Spring 应用上下文。

```
@Configurationpublic class StudentConfig {
    @Bean
    public StudentBean myStudent() {
        return new StudentBean();
    }
}
```

怎样开启注解装配?

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在 Spring 配置文件中配置 `<context:annotation-config/>`元素。

@Component, @Controller, @Repository, @Service 有何区别?

@Component：这将 java 类标记为 bean。它是任何 Spring 管理组件的通用构造型。spring 的组件扫描机制现在可以将其拾取并将其拉入应用程序环境中。

@Controller：这将一个类标记为 Spring Web MVC 控制器。标有它的 Bean 会自动导入到 IoC 容器中。

@Service：此注解是组件注解的特化。它不会对 @Component 注解提供任何其他行为。您可以在服务层类中使用 @Service 而不是 @Component，因为它以更好的方式指定了意图。

@Repository：这个注解是具有类似用途和功能的 @Component 注解的特化。它为 DAO 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException。

@Required 注解有什么作用

这个注解表明 bean 的属性必须在配置的时候设置，通过一个 bean 定义的显式的属性值或通过自动装配，若@Required 注解的 bean 属性未被设置，容器将抛出 BeanInitializationException。示例：

```
public class Employee {

    private String name;

    @Required

    public void setName(String name){

        this.name=name;

    }

    public string getName(){

        return name;

    }

}
```

@Autowired 注解有什么作用

@Autowired 默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它 required 属性为 false）。@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required 一样，修饰 setter 方法、构造器、属性或者具有任意名称和/或多个参数的 PN 方法。

```
public class Employee {
    private String name;
    @Autowired
    public void setName(String name) {
        this.name=name;
    }
    public string getName(){
        return name;
    }
}
```

@Autowired 和@Resource 之间的区别

@Autowired 可用于：构造函数、成员变量、Setter 方法

@Autowired 和@Resource 之间的区别

- @Autowired 默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它 required 属性为 false）。

- @Resource 默认是按照名称来装配注入的，只有当找不到与名称匹配的 bean 才会按照类型来装配注入。

@Qualifier 注解有什么作用

当您创建多个相同类型的 bean 并希望仅使用属性装配其中一个 bean 时，您可以使用 @Qualifier 注解和 @Autowired 通过指定应该装配哪个确切的 bean 来消除歧义。

@RequestMapping 注解有什么用?

@RequestMapping 注解用于将特定 HTTP 请求方法映射到将处理相应请求的控制器中的特定类/方法。此注释可应用于两个级别：

- 类级别：映射请求的 URL
- 方法级别：映射 URL 以及 HTTP 请求方法

Spring 数据访问

解释对象/关系映射集成模块

Spring 通过提供 ORM 模块，支持我们在直接 JDBC 之上使用一个对象/关系映射映射(ORM)工具，Spring 支持集成主流的 ORM 框架，如 Hiberate，JDO 和 iBATIS，JPA，TopLink，JDO，OJB 。Spring 的事务管理同样支持以上所有 ORM 框架及 JDBC。

在 Spring 框架中如何更有效地使用 JDBC?

使用 Spring JDBC 框架，资源管理和错误处理的代价都会被减轻。所以开发者只需写 statements 和 queries 从数据存取数据，JDBC 也可以在 Spring 框架提供的模板类的帮助下更有效地被使用，这个模板叫 JdbcTemplate

解释 JDBC 抽象和 DAO 模块

通过使用 JDBC 抽象和 DAO 模块，保证数据库代码的简洁，并能避免数据库资源错误关闭导致的问题，它在各种不同的数据库的错误信息之上，提供了一个统一的异常访问层。它还利用 Spring 的 AOP 模块给 Spring 应用中的对象提供事务管理服务。

spring DAO 有什么用?

Spring DAO（数据访问对象） 使得 JDBC，Hibernate 或 JDO 这样的数据访问技术更容易以一种统一的方式工作。这使得用户容易在持久性技术之间切换。它还允许您在编写代码时，无需考虑捕获每种技术不同的异常。

spring JDBC API 中存在哪些类?

JdbcTemplate

SimpleJdbcTemplate

NamedParameterJdbcTemplate

SimpleJdbcInsert

SimpleJdbcCall

JdbcTemplate 是什么

JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

使用 Spring 通过什么方式访问 Hibernate? 使用 Spring 访问 Hibernate 的方法有哪些?

在 Spring 中有两种方式访问 Hibernate：

- 使用 Hibernate 模板和回调进行控制反转
- 扩展 HibernateDAOSupport 并应用 AOP 拦截器节点

如何通过 HibernateDaoSupport 将 Spring 和 Hibernate 结合起来?

用 Spring 的 SessionFactory 调用 LocalSessionFactory。集成过程分三步：

- 配置 the Hibernate SessionFactory
- 继承 HibernateDaoSupport 实现一个 DAO
- 在 AOP 支持的事务中装配

Spring 支持的事务管理类型， spring 事务实现方式有哪些?

Spring 支持两种类型的事务管理：

编程式事务管理：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。

声明式事务管理：这意味着你可以将业务代码和事务管理分离，你只需用注解和 XML 配置来管理事务。

Spring 事务的实现方式和实现原理

Spring 事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring 是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过 binlog 或者 redo log 实现的。

说一下 Spring 的事务传播行为

spring 事务的传播行为说的是，当多个事务同时存在的时候，spring 如何处理这些事务的行为。

① PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

③ PROPAGATION_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按 REQUIRED 属性执行。

说一下 spring 的事务隔离?

spring 有五大隔离级别，默认值为 ISOLATION_DEFAULT（使用数据库的设置），其他四个隔离级别和数据库的隔离级别一致：

1. ISOLATION_DEFAULT：用底层数据库的设置隔离级别，数据库设置的是什么我就用什么；

2. ISOLATION_READ_UNCOMMITTED：未提交读，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）；

3. ISOLATION_READ_COMMITTED：提交读，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读），SQL server 的默认级别；

4. ISOLATION_REPEATABLE_READ：可重复读，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读），MySQL 的默认级别；

5. ISOLATION_SERIALIZABLE：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。

脏读： 表示一个事务能够读取另一个事务中还未提交的数据。比如，某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。

不可重复读： 是指在一个事务内，多次读同一数据。

幻读：指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

Spring 框架的事务管理有哪些优点?

- 为不同的事务 API 如 JTA，JDBC，Hibernate，JPA 和 JDO，提供一个不变的编程模式。
- 为编程式事务管理提供了一套简单的 API 而不是一些复杂的事务 API
- 支持声明式事务管理。
- 和 Spring 各种数据访问抽象层很好得集成。

你更倾向用那种事务管理类型?

大多数 Spring 框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

Spring 面向切面编程(AOP)（13）

什么是 AOP

OOP(Object-Oriented Programming)面向对象编程，允许开发者定义纵向的关系，但并适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。

AOP(Aspect-Oriented Programming)，一般称为面向切面编程，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理等。

Spring AOP and AspectJ AOP 有什么区别？AOP 有哪些实现方式?

AOP 实现的关键在于 代理模式，AOP 代理主要分为静态代理和动态代理。静态代理的代表为 AspectJ；动态代理则以 Spring AOP 为代表。

（1）AspectJ 是静态代理的增强，所谓静态代理，就是 AOP 框架会在编译阶段生成 AOP 代理类，因此也称为编译时增强，他会在编译阶段将 AspectJ(切面)织入到 Java 字节码中，运行的时候就是增强之后的 AOP 对象。

（2）Spring AOP 使用的动态代理，所谓的动态代理就是说 AOP 框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个 AOP 对象，这个 AOP 对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

JDK 动态代理和 CGLIB 动态代理的区别

Spring AOP 中的动态代理主要有两种方式，JDK 动态代理和 CGLIB 动态代理：

- JDK 动态代理只提供接口的代理，不支持类的代理。核心 InvocationHandler 接口和 Proxy 类，InvocationHandler 通过 invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy 利用 InvocationHandler 动态创建一个符合某一接口的的实例，生成目标类的代理对象。

- 如果代理类没有实现 InvocationHandler 接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现 AOP。CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。

静态代理与动态代理区别在于生成 AOP 代理对象的时机不同，相对来说 AspectJ 的静态代理方式具有更好的性能，但是 AspectJ 需要特定的编译器进行处理，而 Spring AOP 则无需特定的编译器处理。

> InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy 是最终生成的代理实例; method 是被代理目标实例的某个具体方法; args 是被代理目标实例某个方法的具体入参，在方法反射调用时使用。

如何理解 Spring 中的代理?

将 Advice 应用于目标对象后创建的对象称为代理。在客户端对象的情况下，目标对象和代理对象是相同的。

Advice + Target Object = Proxy

解释一下 Spring AOP 里面的几个名词

（1）切面（Aspect）：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。在 Spring AOP 中，切面可以使用通用类（基于模式的风格）或者在普通类中以 @AspectJ 注解来实现。

（2）连接点（Join point）：指方法，在 Spring AOP 中，一个连接点 总是 代表一个方法的执行。 应用可能有数以千计的时机应用通知。这些时机被称为连接点。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

（3）通知（Advice）：在 AOP 术语中，切面的工作被称为通知。

（4）切入点（Pointcut）：切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。

（5）引入（Introduction）：引入允许我们向现有类添加新方法或属性。

（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。它通常是一个代理对象。也有人把它叫做 被通知（adviced） 对象。 既然 Spring AOP 是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

（7）织入（Weaving）：织入是把切面应用到目标对象并创建新的代理对象的过程。在目标对象的生命周期里有多少个点可以进行织入：

- 编译期：切面在目标类编译时被织入。AspectJ 的织入编译器是以这种方式织入切面的。
- 类加载期：切面在目标类加载到 JVM 时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5 的加载时织入就支持以这种方式织入切面。
- 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP 容器会为目标对象动态地创建一个代理对象。SpringAOP 就是以这种方式织入切面。

Spring 在运行时通知对象

通过在代理类中包裹切面，Spring 在运行期把切面织入到 Spring 管理的 bean 中。代理封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的目标 bean。当代理拦截到方法调用时，在调用目标 bean 方法之前，会执行切面逻辑。

直到应用需要被代理的 bean 时，Spring 才创建代理对象。如果使用的是 ApplicationContext 的话，在 ApplicationContext 从 BeanFactory 中加载所有 bean 的时候，Spring 才会创建被代理的对象。因为 Spring 运行时才创建代理对象，所以我们不需要特殊的编译器来织入 SpringAOP 的切面。

Spring 只支持方法级别的连接点

因为 Spring 基于动态代理，所以 Spring 只支持方法连接点。Spring 缺少对字段连接点的支持，而且它不支持构造器连接点。方法之外的连接点拦截功能，我们可以利用 Aspect 来补充。

在 Spring AOP 中，关注点和横切关注的区别是什么？在 spring aop 中 concern 和 cross-cutting concern 的不同之处

关注点（concern）是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

横切关注点（cross-cutting concern）是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

Spring 通知有哪些类型?

在 AOP 术语中，切面的工作被称为通知，实际上是程序执行时要通过 SpringAOP 框架触发的代码段。

Spring 切面可以应用 5 种类型的通知：

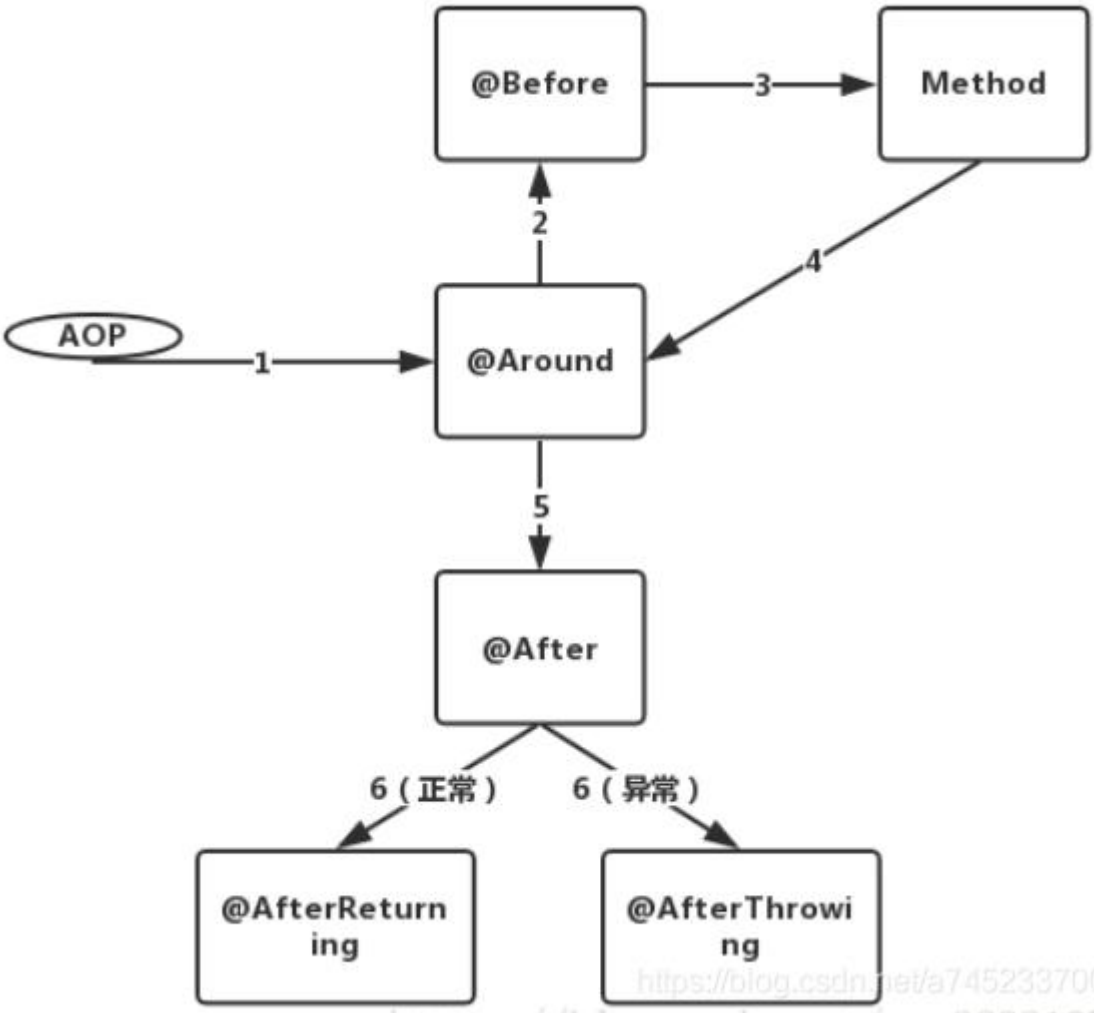
1. 前置通知（Before）：在目标方法被调用之前调用通知功能；

2. 后置通知（After）：在目标方法完成之后调用通知，此时不会关心方法的输出是什么；

3. 返回通知（After-returning ）：在目标方法成功执行之后调用通知；

4. 异常通知（After-throwing）：在目标方法抛出异常后调用通知；

5. 环绕通知（Around）：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的行为。



同一个 aspect，不同 advice 的执行顺序：

①没有异常情况下的执行顺序：

around before advice

before advice

target method 执行

around after advice

after advice

afterReturning

②有异常情况下的执行顺序:

around before advice

before advice

target method 执行

around after advice

after advice

afterThrowing:异常发生

java.lang.RuntimeException: 异常发生

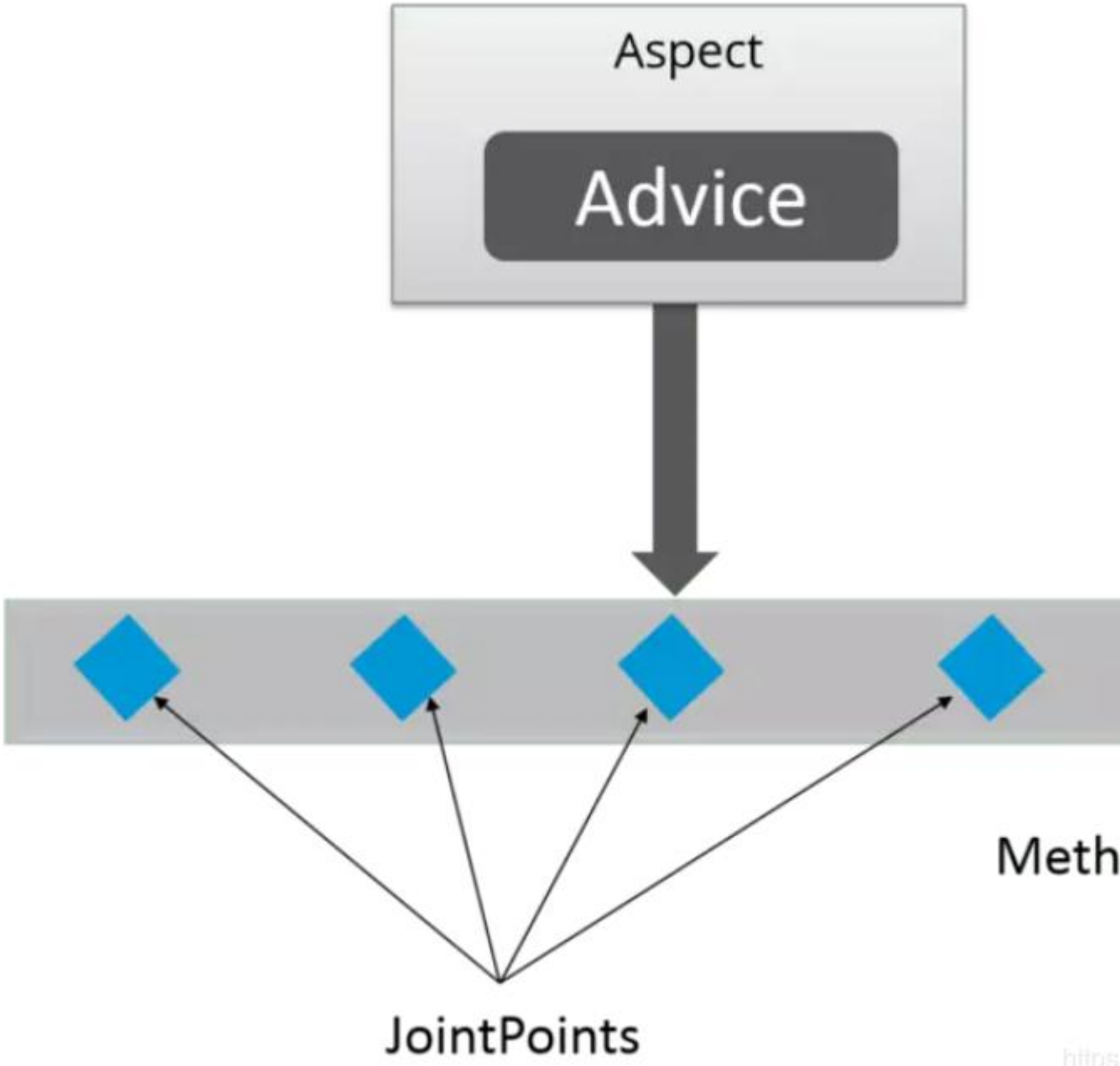
什么是切面 Aspect?

aspect 由 pointcount 和 advice 组成，切面是通知和切点的结合。 它既包含了横切逻辑的定义，也包括了连接点的定义. Spring AOP 就是负责实施切面的框架，它将切面所定义的横切逻辑编织到切面所指定的连接点中.

AOP 的工作重心在于如何将增强编织目标对象的连接点上，这里包含两个工作:

- 如何通过 pointcut 和 advice 定位到特定的 joinpoint 上
- 如何在 advice 中编写切面代码.

可以简单地认为，使用 @Aspect 注解的类就是切面.



解释基于 XML Schema 方式的切面实现

在这种情况下，切面由常规类以及基于 XML 的配置实现。

解释基于注解的切面实现

在这种情况下(基于@AspectJ 的实现)，涉及到的切面声明的风格与带有 java5 标注的普通 java 类一致。

有几种不同类型的自动代理?

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

Metadata autoproxying

Spring MVC 面试题 专题部分

什么是 Spring MVC? 简单介绍下你对 Spring MVC 的理解?

Spring MVC 是一个基于 Java 的实现了 MVC 设计模式的请求驱动类型的轻量级 Web 框架，通过把模型-视图-控制器分离，将 web 层进行职责解耦，把复杂的 web 应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

Spring MVC 的优点

（1）可以支持各种视图技术,而不仅仅局限于 JSP；

（2）与 Spring 框架集成（如 IoC 容器、AOP 等）；

（3）清晰的角色分配：前端控制器(dispatcherServlet)，请求到处理器映射（handlerMapping)，处理器适配器（HandlerAdapter)，视图解析器（ViewResolver）。

（4） 支持各种请求资源的映射策略。

核心组件

Spring MVC 的主要组件?

（1）前端控制器 DispatcherServlet（不需要程序员开发）

作用：接收请求、响应结果，相当于转发器，有了 DispatcherServlet 就减少了其它组件之间的耦合度。

（2）处理器映射器 HandlerMapping（不需要程序员开发）

作用：根据请求的 URL 来查找 Handler

（3）处理器适配器 HandlerAdapter

注意：在编写 Handler 的时候要按照 HandlerAdapter 要求的规则去编写，这样适配器 HandlerAdapter 才可以正确的去执行 Handler。

（4）处理器 Handler（需要程序员开发）

（5）视图解析器 ViewResolver（不需要程序员开发）

作用：进行视图的解析，根据视图逻辑名解析成真正的视图（view）

（6）视图 View（需要程序员开发 jsp）

View 是一个接口， 它的实现类支持不同的视图类型（jsp，freemarker，pdf 等等）

什么是 DispatcherServlet

Spring 的 MVC 框架是围绕 DispatcherServlet 来设计的，它用来处理所有的 HTTP 请求和响应。

什么是 Spring MVC 框架的控制器?

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring 用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

Spring MVC 的控制器是不是单例模式,如果是,有什么问题,怎么解决?

答: 是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

工作原理

请描述 Spring MVC 的工作流程？描述一下 DispatcherServlet 的工作流程?

（1）用户发送请求至前端控制器 DispatcherServlet；

（2） DispatcherServlet 收到请求后，调用 HandlerMapping 处理器映射器，请求获取 Handle；

（3）处理器映射器根据请求 url 找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet；

（4）DispatcherServlet 调用 HandlerAdapter 处理器适配器；

（5）HandlerAdapter 经过适配调用 具体处理器(Handler，也叫后端控制器)；

（6）Handler 执行完成返回 ModelAndView；

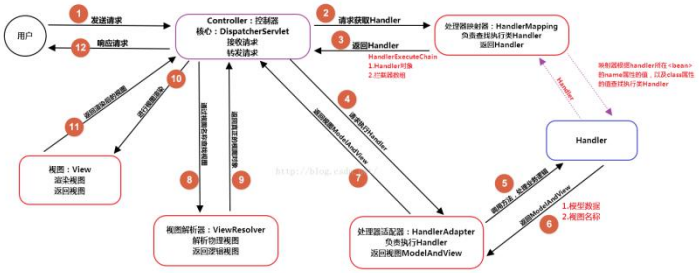
（7）HandlerAdapter 将 Handler 执行结果 ModelAndView 返回给 DispatcherServlet；

（8）DispatcherServlet 将 ModelAndView 传给 ViewResolver 视图解析器进行解析；

（9）ViewResolver 解析后返回具体 View；

（10）DispatcherServlet 对 View 进行渲染视图（即将模型数据填充至视图中）

（11）DispatcherServlet 响应用户。



MVC 框架

MVC 是什么？MVC 设计模式的好处有哪些

mvc 是一种设计模式（设计模式就是日常开发中编写代码的一种好的方法和经验的总结）。模型（model）-视图（view）-控制器（controller），三层架构的设计模式。用于实现前端页面的展现与后端业务数据处理的分离。

mvc 设计模式的好处

1.分层设计，实现了业务系统各个组件之间的解耦，有利于业务系统的可扩展性，可维护性。

2.有利于系统的并行开发，提升开发效率。

常用注解

注解原理是什么

注解本质是一个继承了 Annotation 的特殊接口，其具体实现类是 Java 运行时生成的动态代理类。我们通过反射获取注解时，返回的是 Java 运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用 AnnotationInvocationHandler 的 invoke 方法。

该方法会从 memberValues 这个 Map 中索引出对应的值。而 memberValues 的来源是 Java 常量池。

Spring MVC 常用的注解有哪些?

@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

@RequestBody：注解实现接收 http 请求的 json 数据，将 json 转换为 java 对象。

@ResponseBody：注解实现将 conreoller 方法返回对象转化为 json 对象响应给客户。

SpingMvc 中的控制器的注解一般用哪个,有没有别的注解可以替代?

答：一般用@Controller 注解,也可以使用@RestController,@RestController 注解相当于@ResponseBody + @Controller,表示是表现层,除此之外，一般不用别的注解代替。

@Controller 注解的作用

在 Spring MVC 中，控制器 Controller 负责处理由 DispatcherServlet 分发的请求，它把用户请求的数据经过业务处理层处理之后封装成一个 Model ，然后再把该 Model 返回给对应的 View 进行展示。在 Spring MVC 中提供了一个非常简便的定义 Controller 的方法，你无需继承特定的类或实现特定的接口，只需使用@Controller 标记一个类是 Controller ，然后使用@RequestMapping 和@RequestParam 等一些注解用以定义 URL 请求和 Controller 方法之间的映射，这样的 Controller 就能被外界访问到。此外 Controller 不会直接依赖于 HttpServletRequest 和 HttpServletResponse 等 HttpServlet 对象，它们可以通过 Controller 的方法参数灵活的获取到。

@Controller 用于标记在一个类上，使用它标记的类就是一个 Spring MVC Controller 对象。分发处理器将会扫描使用了该注解的类的方法，并检测该方法是否使用了@RequestMapping 注解。@Controller 只是定义了一个控制器类，而使用@RequestMapping 注解的方法才是真正处理请求的处理器。单单使用@Controller 标记在一个类上还不能真正意义上的说它就是 Spring MVC 的一个控制器类，因为这个时候 Spring 还不认识它。那么要如何做 Spring 才能认识它呢？这个时候就需要我们把这个控制器类交给 Spring 来管理。有两种方式：

- 在 Spring MVC 的配置文件中定义 MyController 的 bean 对象。
- 在 Spring MVC 的配置文件中告诉 Spring 该到哪里去找标记为@Controller 的 Controller 控制器。

@RequestMapping 注解的作用

RequestMapping 是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

RequestMapping 注解有六个属性，下面我们把她分成三类进行说明（下面有相应示例）。

value， method

value： 指定请求的实际地址，指定的地址可以是 URI Template 模式（后面将会说明）；

method： 指定请求的 method 类型， GET、POST、PUT、DELETE 等；

consumes，produces

consumes： 指定处理请求的提交内容类型（Content-Type），例如 application/json, text/html;

produces: 指定返回的内容类型，仅当 request 请求头中的(Accept)类型中包含该指定类型才返回；

params，headers

params： 指定 request 中必须包含某些参数值是，才让该方法处理。

headers： 指定 request 中必须包含某些指定的 header 值，才能让该方法处理请求。

@ResponseBody 注解的作用

作用： 该注解用于将 Controller 的方法返回的对象，通过适当的 HttpMessageConverter 转换为指定格式后，写入到 Response 对象的 body 数据区。

使用时机：返回的数据不是 html 标签的页面，而是其他某种格式的数据时（如 json、xml 等）使用；

@PathVariable 和@RequestParam 的区别

请求路径上有个 id 的变量值，可以通过@PathVariable 来获取 @RequestMapping(value = “/page/{id}”，method = RequestMethod.GET)

@RequestParam 用来获得静态的 URL 请求入参 spring 注解时 action 里用到。

其他

Spring MVC 与 Struts2 区别

相同点

都是基于 mvc 的表现层框架，都用于 web 项目的开发。

不同点

1.前端控制器不一样。Spring MVC 的前端控制器是 servlet：DispatcherServlet。struts2 的前端控制器是 filter：StrutsPreparedAndExcutorFilter。

2.请求参数的接收方式不一样。Spring MVC 是使用方法的形参接收请求的参数，基于方法的开发，线程安全，可以设计为单例或者多例的开发，推荐使用单例模式的开发（执行效率更高），默认就是单例开发模式。struts2 是通过类的成员变量接收请求的参数，是基于类的开发，线程不安全，只能设计为多例的开发。

3.Struts 采用值栈存储请求和响应的数据，通过 OGNL 存取数据，Spring MVC 通过参数解析器是将 request 请求内容解析，并给方法形参赋值，将数据和视图封装成 ModelAndView 对象，最后又将 ModelAndView 中的模型数据通过 reques 域传输到页面。Jsp 视图解析器默认使用 jstl。

4.与 spring 整合不一样。Spring MVC 是 spring 框架的一部分，不需要整合。在企业项目中，Spring MVC 使用更多一些。

Spring MVC 怎么样设定重定向和转发的?

（1）转发：在返回值前面加"forward:"，譬如"forward:user.do?name=method4"

（2）重定向：在返回值前面加"redirect:"，譬如"redirect:http://www.baidu.com"

Spring MVC 怎么和 AJAX 相互调用的?

通过 Jackson 框架就可以把 Java 里面的对象直接转化成 Js 可以识别的 Json 对象。具体步骤如下 ：

（1）加入 Jackson.jar

（2）在配置文件中配置 json 的映射

（3）在接受 Ajax 方法里面可以直接返回 Object,List 等,但方法前面要加上 @ResponseBody 注解。

如何解决 POST 请求中文乱码问题，GET 的又如何处理呢?

（1）解决 post 请求乱码问题：

在 web.xml 中配置一个 CharacterEncodingFilter 过滤器，设置成 utf-8；

```
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

（2）get 请求中文参数出现乱码解决方法有两个：

①修改 tomcat 配置文件添加编码与工程编码一致，如下：

<ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>

②另外一种方法对参数进行重新编码：

String userName = new

String(request.getParamter(“userName”).getBytes(“ISO8859-1”),“utf-8”)

ISO8859-1 是 tomcat 默认编码，需要将 tomcat 编码后的内容按 utf-8 编码。

Spring MVC 的异常处理?

答：可以将异常抛给 Spring 框架，由 Spring 框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

如果在拦截请求中，我想拦截 get 方式提交的方法,怎么配置

答：可以在@RequestMapping 注解里面加上 method=RequestMethod.GET。

怎样在方法里面得到 Request,或者 Session?

答：直接在方法的形参中声明 request,Spring MVC 就自动把 request 对象传入。

如果想在拦截的方法里面得到从前台传入的参数,怎么得到?

答：直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样。

如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象?

答：直接在方法中声明这个对象,Spring MVC 就自动会把属性赋值到这个对象里面。

Spring MVC 中函数的返回值是什么?

答：返回值可以有很多类型,有 String, ModelAndView。ModelAndView 类把视图和数据都合并的一起的，但一般用 String 比较好。

Spring MVC 用什么对象从后台向前台传递数据的?

答：通过 ModelMap 对象,可以在这个对象里面调用 put 方法,把对象加到里面,前台就可以通过 el 表达式拿到。

怎么样把 ModelMap 里面的数据放入 Session 里面?

答: 可以在类上面加上@SessionAttributes 注解,里面包含的字符串就是要放入 session 里面的 key。

Spring MVC 里面拦截器是怎么写的

有两种写法,一种是实现 HandlerInterceptor 接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在 Spring MVC 的配置文件中配置拦截器即可：

```
<!-- 配置 Spring MVC 的拦截器 --><mvc:interceptors>
    <!-- 配置一个拦截器的 Bean 就可以了 默认是对所有请求都拦截 -->
    <bean id="myInterceptor" class="com.zwp.action.MyHandlerInterceptor"></bean>
    <!-- 只针对部分请求拦截 -->
    <mvc:interceptor>
        <mvc:mapping path="/modelMap.do" />
        <bean class="com.zwp.action.MyHandlerInterceptorAdapter" />
    </mvc:interceptor></mvc:interceptors>
```

介绍一下 WebApplicationContext

WebApplicationContext 继承了 ApplicationContext 并增加了一些 WEB 应用必备的特有功能，它不同于一般的 ApplicationContext ，因为它能处理主题，并找到被关联的 servlet。

JUC 并发包与容器

内存可见性、指令有序性 理论

为什么代码会重排序?

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

在单线程环境下不能改变程序运行的结果；

存在数据依赖关系的不允许重排序

需要注意的是：重排序不会影响单线程环境的执行结果，但是会破坏多线程的执行语义。

as-if-serial 规则和 happens-before 规则的区别

- as-if-serial 语义保证单线程内程序的执行结果不被改变，happens-before 关系保证正确同步的多线程程序的执行结果不被改变。
- as-if-serial 语义给编写单线程程序的程序员创造了一个幻境：单线程程序是按程序的顺序来执行的。happens-before 关系给编写正确同步的多线程程序的程序员创造了一个幻境：正确同步的多线程程序是按 happens-before 指定的顺序来执行的。

- as-if-serial 语义和 happens-before 这么做的目的，都是为了在不改变程序执行结果的前提下，尽可能地提高程序执行的并行度。

volatile 内存可见性

volatile 关键字的作用

对于可见性，Java 提供了 volatile 关键字来保证可见性和禁止指令重排。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。当一个共享变量被 volatile 修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

从实践角度而言，volatile 的一个重要作用就是和 CAS 结合，保证了原子性，详细的可以参见 java.util.concurrent.atomic 包下的类，比如 AtomicInteger。

volatile 常用于多线程环境下的单次操作(单次读或者单次写)。

Java 中能创建 volatile 数组吗?

能，Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。意思是，如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。

volatile 变量和 atomic 变量有什么不同?

volatile 变量可以确保先行关系，即写操作会发生在后续的读操作之前，但它并不能保证原子性。例如用 volatile 修饰 count 变量，那么 count++ 操作就不是原子性的。

而 AtomicInteger 类提供的 atomic 方法可以让这种操作具有原子性如 getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

volatile 能使得一个非原子操作变成原子操作吗?

关键字 volatile 的主要作用是使变量在多个线程间可见，但无法保证原子性，对于多个线程访问同一个实例变量需要加锁进行同步。

虽然 volatile 只能保证可见性不能保证原子性，但用 volatile 修饰 long 和 double 可以保证其操作原子性。

所以从 Oracle Java Spec 里面可以看到：

- 对于 64 位的 long 和 double，如果没有被 volatile 修饰，那么对其操作可以不是原子的。在操作的时候，可以分成两步，每次对 32 位操作。
- 如果使用 volatile 修饰 long 和 double，那么其读写都是原子操作
- 对于 64 位的引用地址的读写，都是原子操作
- 在实现 JVM 时，可以自由选择是否把读写 long 和 double 作为原子操作
- 推荐 JVM 实现为原子操作

volatile 修饰符的有过什么实践?

单例模式

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：较复杂

描述：对于 Double-Check 这种可能出现的问题（当然这种概率已经非常小了，但毕竟还是有的嘛~），解决方案是：只需要给 instance 的声明加上 volatile 关键字即可 volatile 关键字的一个作用是禁止指令重排，把 instance 声明为 volatile 之后，对它的写操作就会有一个内存屏障（什么是内存屏障？），这样，在它的赋值完成之前，就不用会调用读操作。注意：volatile 阻止的不是 singleton = newSingleton()这句话内部[1-2-3]的指令重排，而是保证了在一个写操作（[1-2-3]）完成之前，不会调用读操作（if (instance == null)）。

```
public class Singleton7 {

    private static volatile Singleton7 instance = null;

    private Singleton7() {}

    public static Singleton7 getInstance() {
        if (instance == null) {
            synchronized (Singleton7.class) {
                if (instance == null) {
                    instance = new Singleton7();
                }
```

```
            }
        }

        return instance;
    }
}12345678910111213141516171819
```

synchronized 和 volatile 的区别是什么?

synchronized 表示只有一个线程可以获取作用对象的锁，执行代码，阻塞其他线程。

volatile 表示变量在 CPU 的寄存器中是不确定的，必须从主存中读取。保证多线程环境下变量的可见性；禁止指令重排序。

区别

- volatile 是变量修饰符；synchronized 可以修饰类、方法、变量。
- volatile 仅能实现变量的修改可见性，不能保证原子性；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；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；synchronized 标记的变量可以被编译器优化。

- **volatile 关键字**是线程同步的**轻量级实现**，所以 **volatile 性能肯定比 synchronized 关键字要好。**但是 **volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块。**synchronized 关键字在 JavaSE1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，**实际开发中使用 synchronized 关键字的场景还是更多一些。**

final

什么是不可变对象，它对写并发应用有什么帮助?

不可变对象(Immutable Objects)即对象一旦被创建它的状态（对象的数据，也即对象属性值）就不能改变，反之即为可变对象(Mutable Objects)。

不可变对象的类即为不可变类(Immutable Class)。Java 平台类库中包含许多不可变类，如 String、基本类型的包装类、BigInteger 和 BigDecimal 等。

只有满足如下状态，一个对象才是不可变的；

- 它的状态不能在创建后再被修改；
- 所有域都是 final 类型；并且，它被正确创建（创建期间没有发生 this 引用的逸出）。

不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行效率。

GC

Java 中垃圾回收有什么目的？什么时候进行垃圾回收?

垃圾回收是在内存中存在没有引用的对象或超过作用域的对象时进行的。

垃圾回收的目的是识别并且丢弃应用不再使用的对象来释放和重用资源。

如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占用的内存?

不会，在下一个垃圾回调周期中，这个对象将是被可回收的。

也就是说并不会立即被垃圾收集器立刻回收，而是在下一次垃圾回收时才会释放其占用的内存。

finalize()方法什么时候被调用？析构函数(finalization)的目的是什么?

1）垃圾回收器（garbage colector）决定回收某对象时，就会运行该对象的 finalize()方法；finalize 是 Object 类的一个方法，该方法在 Object 类中的声明 protected void finalize() throws Throwable { }

在垃圾回收器执行时会调用被回收对象的 finalize()方法，可以覆盖此方法来实现对其资源的回收。注意：一旦垃圾回收器准备释放对象占用的内存，将首先调用该对象的 finalize()方法，并且下一次垃圾回收动作发生时，才真正回收对象占用的内存空间

2）GC 本来就是内存回收了，应用还需要在 finalization 做什么呢？ 答案是大部分时候，什么都不用做(也就是不需要重载)。只有在某些很特殊的情况下，比如你调用了一些 native 的方法(一般是 C 写的)，可以要在 finaliztion 里去调用 C 的释放函数。

CAS 原子操作

什么是 CAS

CAS 是 compare and swap 的缩写，即我们所说的比较交换。

cas 是一种基于锁的操作，而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS 是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被 b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

java.util.concurrent.atomic 包下的类大多是使用 CAS 操作来实现的(AtomicInteger,AtomicBoolean,AtomicLong)。

CAS 的会产生什么问题?

1、ABA 问题：

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

2、循环时间长开销大：

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

3、只能保证一个共享变量的原子操作：

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

Lock 显示锁

Lock 接口(Lock interface)是什么？对比同步它有什么优势?

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

（1）可以使锁更公平

（2）可以使线程在等待锁的时候响应中断

（3）可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间

（4）可以在不同的范围，以不同的顺序获取和释放锁

整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

乐观锁和悲观锁的理解及如何实现，有哪些实现方式？

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如 Java 里面的同步原语 synchronized 关键字的实现也是悲观锁。

乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_condition 机制，其实都是提供的乐观锁。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

乐观锁的实现方式：

1、使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

2、java 中的 Compare and Swap 即 CAS ，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。 CAS 操作中包含三个操作数 —— 需要读写的内存位置（V）、进行比较的预期原值（A）和拟写入的新值(B)。如果内存位置 V 的值与预期原值 A 相匹配，那么处理器会自动将该位置值更新为新值 B。否则处理器不做任何操作。

ReentrantLock(重入锁)实现原理与公平锁非公平锁区别

什么是可重入锁（ReentrantLock）？

ReentrantLock 重入锁，是实现 Lock 接口的一个类，也是在实际编程中使用频率很高的一个锁，支持重入性，表示能够对共享资源能够重复加锁，即当前线程获取该锁再次获取不会被阻塞。

在 java 关键字 synchronized 隐式支持重入性，synchronized 通过获取自增，释放自减的方式实现重入。与此同时，ReentrantLock 还支持公平锁和非公平锁两种方式。那么，要想完完全全的弄懂 ReentrantLock 的话，主要也就是 ReentrantLock 同步语义的学习：1. 重入性的实现原理；2. 公平锁和非公平锁。

重入性的实现原理

要想支持重入性，就要解决两个问题：**1．在线程获取锁的时候，如果已经获取锁的线程是当前线程的话则直接再次获取成功；2．由于锁会被获取 n 次，那么只有锁在被释放同样的 n 次之后，该锁才算是完全释放成功。**

ReentrantLock 支持两种锁：**公平锁**和**非公平锁。何谓公平性，是针对获取锁而言的，如果一个锁是公平的，那么锁的获取顺序就应该符合请求上的绝对时间顺序，满足 FIFO。**

读写锁 ReentrantReadWriteLock 源码分析

ReadWriteLock 是什么

首先明确一下，不是说 ReentrantLock 不好，只是 ReentrantLock 某些时候有局限。如果使用 ReentrantLock，可能本身是为了防止线程 A 在写数据、线程 B 在读数据造成的数据不一致，但这样，如果线程 C 在读数据、线程 D 也在读数据，读数据是不会改变数

据的，没有必要加锁，但是还是加锁了，降低了程序的性能。因为这个，才诞生了读写锁 ReadWriteLock。

ReadWriteLock 是一个读写锁接口，读写锁是用来提升并发程序性能的锁分离技术，ReentrantReadWriteLock 是 ReadWriteLock 接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

而读写锁有以下三个重要的特性：

（1）公平选择性：支持非公平（默认）和公平的锁获取方式，吞吐量还是非公平优于公平。

（2）重进入：读锁和写锁都支持线程重进入。

（3）锁降级：遵循获取写锁、获取读锁再释放写锁的次序，写锁能够降级成为读锁。

高并发容器

并发容器之 ConcurrentHashMap 详解(JDK1.8 版本)与源码分析

什么是 ConcurrentHashMap?

ConcurrentHashMap 是 Java 中的一个**线程安全且高效的 HashMap 实现。**平时涉及高并发如果要用 map 结构，那第一时间想到的就是它。相对于 hashmap 来说，ConcurrentHashMap 就是线程安全的 map，其中利用了锁分段的思想提高了并发度。

那么它到底是如何实现线程安全的?

JDK 1.6 版本关键要素：

segment 继承了 ReentrantLock 充当锁的角色，为每一个 segment 提供了线程安全的保障；

segment 维护了哈希散列表的若干个桶，每个桶由 HashEntry 构成的链表。

JDK1.8 后，ConcurrentHashMap 抛弃了原有的 **Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。**

Java 中 ConcurrentHashMap 的并发度是什么?

ConcurrentHashMap 把实际 map 划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是 ConcurrentHashMap 类构造函数的一个可选参数，默认值为 16，这样在多线程情况下就能避免争用。

在 JDK8 后，它摒弃了 Segment（锁段）的概念，而是启用了一种全新的方式实现,利用 CAS 算法。同时加入了更多的辅助变量来提高并发度，具体内容还是查看源码吧。

什么是并发容器的实现?

何为同步容器：可以简单地理解为通过 synchronized 来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行。比如 Vector，Hashtable，以及 Collections.synchronizedSet，synchronizedList 等方法返回的容器。可以通过查看 Vector，Hashtable 等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字 synchronized。

并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩性，例如在 ConcurrentHashMap 中采用了一种粒度更细的加锁机制，可以称为分段锁，在这种锁机制下，允许任意数量的读线程并发地访问 map，并且执行读操作的线程和写操作的线程也

可以并发的访问 map，同时允许一定数量的写操作线程并发地修改 map，所以它可以在并发环境下实现更高的吞吐量。

Java 中的同步集合与并发集合有什么区别?

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在 Java1.5 之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java5 介绍了并发集合像 ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

SynchronizedMap 和 ConcurrentHashMap 有什么区别?

SynchronizedMap 一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为 map。

ConcurrentHashMap 使用分段锁来保证在多线程下的性能。

ConcurrentHashMap 中则是一次锁住一个桶。ConcurrentHashMap 默认将 hash 表分为 16 个桶，诸如 get，put，remove 等常用操作只锁当前需要用到的桶。

这样，原来只能一个线程进入，现在却能同时有 16 个写线程执行，并发性能的提升是显而易见的。

另外 ConcurrentHashMap 使用了一种不同的迭代方式。在这种迭代方式中，当 iterator 被创建后集合再发生改变就不再是抛出 ConcurrentModificationException，取而代之的是在改变时 new 新的数据从而不影响原有的数据，iterator 完成后再将头指针替换为新的数据 ，这样 iterator 线程可以使用原来老的数据，而写线程也可以并发的完成改变。

并发容器之 CopyOnWriteArrayList 详解

CopyOnWriteArrayList 是什么，可以用于什么应用场景？有哪些优缺点？

CopyOnWriteArrayList 是一个并发容器。有很多人称它是线程安全的，我认为这句话不严谨，缺少一个前提条件，那就是非复合场景下操作它是线程安全的。

CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出 ConcurrentModificationException。在 CopyOnWriteArrayList 中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

CopyOnWriteArrayList 的使用场景

通过源码分析，我们看出它的优缺点比较明显，所以使用场景也就比较明显。就是合适读多写少的场景。

CopyOnWriteArrayList 的缺点

1. 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致 young gc 或者 full gc。

2. 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个 set 操作后，读取到数据可能还是旧的，虽然 CopyOnWriteArrayList 能做到最终一致性,但是还是没法满足实时性要求。

3. 由于实际使用中可能没法保证 CopyOnWriteArrayList 到底要放置多少数据，万一数据稍微有点多，每次 add/set 都要重新复制数组，这个代价实在太高昂了。在高性能的互联网应用中，这种操作分分钟引起故障。

CopyOnWriteArrayList 的设计思想

1. 读写分离，读和写分开
2. 最终一致性
3. 使用另外开辟空间的思路，来解决并发冲突

并发容器之 BlockingQueue 详解

什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。

这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

JDK7 提供了 7 个阻塞队列。分别是：

ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。

PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。

DelayQueue：一个使用优先级队列实现的无界阻塞队列。

SynchronousQueue：一个不存储元素的阻塞队列。

LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

Java 5 之前实现同步存取时，可以使用普通的一个集合，然后在使用线程的协作和线程同步可以实现生产者，消费者模式，主要的技术就是用好，wait,notify,notifyAll,sychronized 这些关键字。而在 java 5 之后，可以使用阻塞队列来实现，此方式大大简少了代码量，使得多线程编程更加容易，安全方面也有保障。

BlockingQueue 接口是 Queue 的子接口，它的主要用途并不是作为容器，而是作为线程同步的的工具，因此他具有一个很明显的特性，当生产者线程试图向 BlockingQueue 放入元素时，如果队列已满，则线程被阻塞，当消费者线程试图从中取出一个元素时，如果队列为空，则该线程会被阻塞，正是因为它所具有这个特性，所以在程序中多个线程交替向 BlockingQueue 中放入元素，取出元素，它可以很好的控制线程之间的通信。

阻塞队列使用最经典的场景就是 socket 客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。

并发容器之 ConcurrentLinkedQueue 详解

ConcurrentLinkedQueue 非阻塞无界链表队列

ConcurrentLinkedQueue 是一个线程安全的队列，基于链表结构实现，是一个无界队列，理论上来说队列的长度可以无限扩大。

与其他队列相同，ConcurrentLinkedQueue 也采用的是先进先出（FIFO）入队规则，对元素进行排序。 （推荐学习：java 面试题目）

当我们向队列中添加元素时，新插入的元素会插入到队列的尾部；而当我们获取一个元素时，它会从队列的头部中取出。

因为 ConcurrentLinkedQueue 是链表结构，所以当入队时，插入的元素依次向后延伸，形成链表；而出队时，则从链表的第一个元素开始获取，依次递增；

值得注意的是，在使用 ConcurrentLinkedQueue 时，如果涉及到队列是否为空的判断，切记不可使用 size()==0 的做法，因为在 size()方法中，是通过遍历整个链表来实现的，在队列元素很多的时候，size()方法十分消耗性能和时间，只是单纯的判断队列为空使用 isEmpty()即可。

BlockingQueue 拯救了生产者、消费者模型的控制逻辑

经典的“生产者”和“消费者”模型中，在 concurrent 包发布以前，在多线程环境下，我们每个程序员都必须去自己控制这些细节，尤其还要兼顾效率和线程安全，而这会给我们的程序带来不小的复杂度。好在此时，强大的 concurrent 包横空出世了，而他也给我们带来了强大的 BlockingQueue。（在多线程领域：所谓阻塞，在某些情况下会挂起线程（即阻塞），一旦条件满足，被挂起的线程又会自动被唤醒）

BlockingQueue 的成员介绍

因为它隶属于集合家族，自己又是个接口。所以是有很多成员的，下面简单介绍一下

1. ArrayBlockingQueue

基于数组的阻塞队列实现，在 ArrayBlockingQueue 内部，维护了一个定长数组，以便缓

存队列中的数据对象，这是一个常用的阻塞队列，除了一个定长数组外，ArrayBlockingQueue 内部还保存着两个整形变量，分别标识着队列的头部和尾部在数组中的位置。 ArrayBlockingQueue 在生产者放入数据和消费者获取数据，都是共用同一个锁对象，由此也意味着两者无法真正并行运行，这点尤其不同于 LinkedBlockingQueue；按照实现原理来分析，ArrayBlockingQueue 完全可以采用分离锁，从而实现生产者和消费者操作的完全并行运行。 Doug Lea 之所以没这样去做，也许是因为 ArrayBlockingQueue 的数据写入和获取操作已经足够轻巧，以至于引入独立的锁机制，除了给代码带来额外的复杂性外，其在性能上完全占不到任何便宜。 ArrayBlockingQueue 和 LinkedBlockingQueue 间还有一个明显的不同之处在于，前者在插入或删除元素时不会产生或销毁任何额外的对象实例，而后者则会生成一个额外的 Node 对象。这在长时间内需要高效并发地处理大批量数据的系统中，其对于 GC 的影响还是存在一定的区别。而在创建 ArrayBlockingQueue 时，我们还可以控制对象的内部锁是否采用公平锁，默认采用非公平锁。

2.LinkedBlockingQueue

基于链表的阻塞队列，同 ArrayListBlockingQueue 类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到最大值缓存容量时（LinkedBlockingQueue 可以通过构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。而 LinkedBlockingQueue 之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。 作为开发者，我

们需要注意的是，如果构造一个 LinkedBlockingQueue 对象，而没有指定其容量大小，LinkedBlockingQueue 会默认一个类似无限大小的容量（Integer.MAX_VALUE），这样的话，如果生产者的速度一旦大于消费者的速度，也许还没有等到队列满阻塞产生，系统内存就有可能已被消耗殆尽了。

3. DelayQueue 延迟队列

DelayQueue 中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue 是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞，所以一定要注意内存的使用。

使用场景： DelayQueue 使用场景较少，但都相当巧妙，常见的例子比如使用一个 DelayQueue 来管理一个超时未响应的连接队列。

4. PriorityBlockingQueue

基于优先级的阻塞队列（优先级的判断通过构造函数传入的 Compator 对象来决定），但需要注意的是 PriorityBlockingQueue 并不会阻塞数据生产者，而只会在没有可消费的数据时，阻塞数据的消费者。因此使用的时候要特别注意，生产者生产数据的速度绝对不能快于消费者消费数据的速度，否则时间一长，会最终耗尽所有的可用堆内存空间。在实现 PriorityBlockingQueue 时，内部控制线程同步的锁采用的是公平锁。

5. SynchronousQueue

一种无缓冲的等待队列，类似于无中介的直接交易，有点像原始社会中的生产者和消费者，生产者拿着产品去集市销售给产品的最终消费者，而消费者必须亲自去集市找到所要商品的直接生产者，如果一方没有找到合适的目标，那么对不起，大家都在集市等待。相对于有缓冲的 BlockingQueue 来说，少了一个中间经销商的环节（缓冲区），如果有经销商，生产者直接把产品批发给经销商，而无需在意经销商最终会将这些产品卖给那些消费者，由于经

销商可以库存一部分商品，因此相对于直接交易模式，总体来说采用中间经销商的模式会吞吐量高一些（可以批量买卖）；但另一方面，又因为经销商的引入，使得产品从生产者到消费者中间增加了额外的交易环节，单个产品的及时响应性能可能会降低。

小结

BlockingQueue 不光实现了一个完整队列所具有的基本功能，同时在多线程环境下，他还自动管理了多线间的自动等待于唤醒功能，从而使得程序员可以忽略这些细节，关注更高级的功能。

原子操作类

什么是原子操作?

原子操作（atomic operation）意为”不可被中断的一个或一系列操作” 。

处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。 CAS 操作——Compare & Set，或是 Compare & Swap，现在几乎所有的 CPU 指令都支持 CAS 的原子操作。

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加 1 时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在 JDK1.5 之前我们可以使用同步技术来做到这一点。到 JDK1.5，java.util.concurrent.atomic 包提供了 int 和 long 类型的原子包装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

在 Java Concurrency API 中有哪些原子类(atomic classes)?

java.util.concurrent 这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择另一个线程进入，这只是一种逻辑上的理解。

原子类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference

原子数组：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray

原子属性更新器：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater

解决 ABA 问题的原子类：AtomicMarkableReference（通过引入一个 boolean 来反映中间有没有变过），AtomicStampedReference（通过引入一个 int 来累加来反映中间有没有变过）

说一下 atomic 的原理?

Atomic 包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

AtomicInteger 类的部分源码：

```
// setup to use Unsafe.compareAndSwapInt for updates（更新操作时提供“比较并替换”的作用）
private static final Unsafe unsafe = Unsafe.getUnsafe();private static final long valueOffset;
```

```
static {
        try {
                valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
}
private volatile int value;123456789101112
```

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS 的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个 volatile 变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

同步工具类

并发工具之 CountDownLatch 与 CyclicBarrier

常用的并发工具类有哪些?

- **Semaphore(信号量)-允许多个线程同时访问**：synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

- **CountDownLatch(倒计时器)：** CountDownLatch 是一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。

- **CyclicBarrier(循环栅栏)：** CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用 await()方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

在 Java 中 CycliBarriar 和 CountdownLatch 有什么区别?

CountDownLatch 与 CyclicBarrier 都是用于控制并发的工具类，都可以理解成维护的就是一个计数器，但是这两者还是各有不同侧重点的：

- CountDownLatch 一般用于某个线程 A 等待若干个其他线程执行完任务之后，它才执行；而 CyclicBarrier 一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；CountDownLatch 强调一个线程等多个线程完成某件事情。CyclicBarrier 是多个线程互等，等大家都完成，再携手共进。

- 调用 CountDownLatch 的 countDown 方法后，当前线程并不会阻塞，会继续往下执行；而调用 CyclicBarrier 的 await 方法，会阻塞当前线程，直到 CyclicBarrier 指定的线程全部都到达了指定点的时候，才能继续往下执行；

- CountDownLatch 方法比较少，操作比较简单，而 CyclicBarrier 提供的方法更多，比如能够通过 getNumberWaiting()，isBroken()这些方法获取当前多个线程的状态，并且 CyclicBarrier 的构造方法可以传入 barrierAction，指定当所有线程都到达时执行的业务功能；

- CountDownLatch 是不能复用的，而 CyclicLatch 是可以复用的。

并发工具之 Semaphore 与 Exchanger

Semaphore 有什么作用

Semaphore 就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore 有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果 Semaphore 构造函数中传入的 int 型整数 n=1，相当于变成了一个 synchronized 了。

Semaphore(信号量)-允许多个线程同时访问：synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

什么是线程间交换数据的工具 Exchanger

Exchanger 是一个用于线程间协作的工具类，用于两个线程间交换数据。它提供了一个交换的同步点，在这个同步点两个线程能够交换数据。交换数据是通过 exchange 方法来实现的，如果一个线程先执行 exchange 方法，那么它会同步等待另一个线程也执行 exchange 方法，这个时候两个线程就都达到了同步点，两个线程就可以交换数据。