# BIO,NIO,AIO 有什么区别?

* BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。
* NIO：Non IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。
* AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。

详细回答

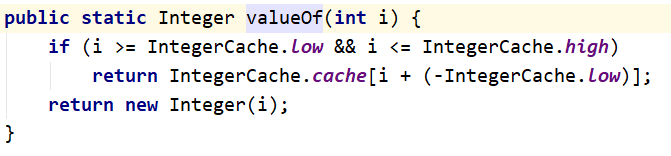
* BIO (Blocking I/O): 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
* NIO (New I/O): NIO是一种同步非阻塞的I/O模型，在Java 1.4 中引入了NIO框架，对应 java.nio 包，提供了 Channel , Selector，Buffer等抽象。NIO中的 N可以理解为Non-blocking，不单纯是New。它支持面向缓冲的，基于通道的I/O操作方法。 NIO提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
* AIO (Asynchronous I/O): AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

# 包装类缓存问题

Integer i =128;  
Integer i2=128;  
System.***out***.println(i==i2);

问题 i i2 = 127 结果是true 128 结果为false

valueOf方法源码



数值在缓存之内,返回缓存中的对象,如果不在返回新的对象

|  |
| --- |
| **private static class** IntegerCache {  **static final int *low*** = -128;  **static final int *high***;  **static final** Integer ***cache***[];   **static** {  *// high value may be configured by property* **int** h = 127;  String integerCacheHighPropValue =  sun.misc.VM.*getSavedProperty*(**"java.lang.Integer.IntegerCache.high"**);  **if** (integerCacheHighPropValue != **null**) {  **try** {  **int** i = *parseInt*(integerCacheHighPropValue);  i = Math.*max*(i, 127);  *// Maximum array size is Integer.MAX\_VALUE* h = Math.*min*(i, Integer.***MAX\_VALUE*** - (-***low***) -1);  } **catch**( NumberFormatException nfe) {  *// If the property cannot be parsed into an int, ignore it.* }  }  ***high*** = h;   ***cache*** = **new** Integer[(***high*** - ***low***) + 1];  **int** j = ***low***;  **for**(**int** k = 0; k < ***cache***.**length**; k++)  ***cache***[k] = **new** Integer(j++);   *// range [-128, 127] must be interned (JLS7 5.1.7)* **assert** IntegerCache.***high*** >= 127;  }   **private** IntegerCache() {} } |

还有哪些包装类有缓存

Byte short Long Integer Character 都有缓存

具体细节,翻阅源代码

# 简述线程生命周期(状态)

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中，它要经过新建(New)、就绪（Runnable）、运行（Running）、阻塞(Blocked)和死亡(Dead)5 种状态。尤其是当线程启动以后，它不可能一直"霸占"着 CPU 独自运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、阻塞之间切换

**新建状态（NEW）**

当程序使用 new 关键字创建了一个线程之后，该线程就处于新建状态，此时仅由 JVM 为其分配内存，并初始化其成员变量的值

**就绪状态（RUNNABLE）：**

当线程对象调用了 start()方法之后，该线程处于就绪状态。Java 虚拟机会为其创建方法调用栈和程序计数器，等待调度运行。

**运行状态（Running）**

线程运行,正在占用CUP运行的一个状态

**阻塞状态（BLOCKED）：**

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：

**等待阻塞（o.wait等待对列）：**

运行(running)的线程执行 o.wait()方法，JVM 会把该线程放入等待队列(waitting queue)中。

**同步阻塞(lock)**

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。

**其他阻塞(sleep/join)**

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。

**线程死亡（DEAD）**

线程会以下面三种方式结束，结束后就是死亡状态。

正常结束

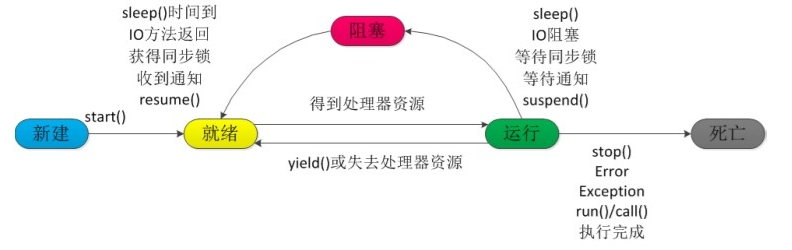
1. run()或 call()方法执行完成，线程正常结束。

异常结束

1. 线程抛出一个未捕获的 Exception 或 Error。

调用stop

1. 直接调用该线程的 stop()方法来结束该线程—该方法通常容易导致死锁，不推荐使用。



# 终止线程 4 种方式

## 正常运行结束

程序运行结束，线程自动结束。

## 使用退出标志退出线程

一般 run()方法执行完，线程就会正常结束，然而，常常有些线程是伺服线程。它们需要长时间的

运行，只有在外部某些条件满足的情况下，才能关闭这些线程。使用一个变量来控制循环，例如：

最直接的方法就是设一个 boolean 类型的标志，并通过设置这个标志为 true 或 false 来控制 while循环是否退出，代码示例：

public class ThreadSafe extends Thread {  
public volatile boolean exit = false;  
public void run() { while (!exit){  
//do something  
}  
}  
}

定义了一个退出标志 exit，当 exit 为 true 时，while 循环退出，exit 的默认值为 false.在定义 exit时，使用了一个 Java 关键字 volatile，这个关键字的目的是使 exit 同步，也就是说在同一时刻只能由一个线程来修改 exit 的值。

## Interrupt 方法结束线程

使用 interrupt()方法来中断线程有两种情况：

1线程处于阻塞状态：如使用了 sleep,同步锁的 wait,socket 中的 receive,accept 等方法时，会使线程处于阻塞状态。当调用线程的 interrupt()方法时，会抛出 InterruptException 异常。阻塞中的那个方法抛出这个异常，通过代码捕获该异常，然后 break 跳出循环状态，从而让我们有机会结束这个线程的执行。通常很多人认为只要调用 interrupt 方法线程就会结束，实际上是错的， 一定要先捕获 InterruptedException 异常之后通过 break 来跳出循环，才能正常结束 run 方法。

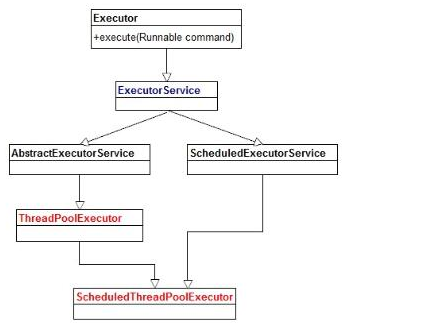
2线程未处于阻塞状态：使用 isInterrupted()判断线程的中断标志来退出循环。当使用interrupt()方法时，中断标志就会置 true，和使用自定义的标志来控制循环是一样的道理。

public class ThreadSafe extends Thread {  
public void run() {  
while (!isInterrupted()){ //非阻塞过程中通过判断中断标志来退出  
try{  
Thread.sleep(51000);//阻塞过程捕获中断异常来退出  
}catch(InterruptedException e){  
e.printStackTrace();  
break;//捕获到异常之后，执行 break 跳出循环  
}  
}  
}  
}

## stop 方法终止线程（线程不安全）

程序中可以直接使用 thread.stop()来强行终止线程，但是 stop 方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，不安全主要是：thread.stop()调用之后，创建子线程的线程就会抛出 ThreadDeatherror 的错误，并且会释放子线程所持有的所有锁。一般任何进行加锁的代码块，都是为了保护数据的一致性，如果在调用thread.stop()后导致了该线程所持有的所有锁的突然释放(不可控制)，那么被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。因此，并不推荐使用 stop 方法来终止线程。

# 线程池用过没?如何获得一个线程池?各个参数的含义?



new Thread 直接就是创建一个线程对象 线程的任务和线程对象合并到一起

new Runnable new Callable 创建一个任务对象 将任务对象放到一个线程对象中去执行 将任务和线程对象分开

## 线程池的好处

* 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。
* 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。
* 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。

1）核心参数

public ThreadPoolExecutor(int corePoolSize, // 核心线程数量大小     
int maximumPoolSize, // 线程池最大容纳线程数  
long keepAliveTime, // 线程空闲后的存活时长  
TimeUnit unit,  
//缓存异步任务的队列 //用来构造线程池里的worker线程  
BlockingQueue<Runnable> workQueue,  
ThreadFactory threadFactory,  
//线程池任务满载后采取的任务拒绝策略   
RejectedExecutionHandler handler)

* corePoolSize

线程池的核心线程数。在没有设置 allowCoreThreadTimeOut 为 true 的情况下，核心线程会在线程池中一直存活，即使处于闲置状态。

maximumPoolSize

线程池所能容纳的最大线程数。当活动线程(核心线程+非核心线程)达到这个数值后，后续任务将会根据 RejectedExecutionHandler 来进行拒绝策略处理。

* keepAliveTime

非核心线程闲置时的超时时长。超过该时长，**非核心线程**就会被回收。若线程池通设置核心线程也允许 timeOut，即 allowCoreThreadTimeOut 为 true，则该时长同样会作用于核心线程，在超过 aliveTime 时，核心线程也会被回收，AsyncTask 配置的线程池就是这样设置的。

* unit

keepAliveTime 时长对应的单位。Timeunit

* workQueue

线程池中的任务队列，通过线程池的 execute() 方法提交的 Runnable 对象会存储在该队列中。

* ThreadFactory

线程工厂，功能很简单，就是为线程池提供创建新线程的功能。这是一个接口，可以通过自定义，做一些自定义线程名的操作。

* RejectedExecutionHandler

当任务无法被执行时(超过线程最大容量 maximum 并且 workQueue 已经被排满了)的处理策略，这里有四种任务拒绝类型。

## 线程池工作原则

* 1、当线程池中线程数量小于 corePoolSize 则创建线程，并处理请求。
* 2、当线程池中线程数量大于等于 corePoolSize 时，则把请求放入 workQueue 中,随着线程池中的核心线程们不断执行任务，只要线程池中有空闲的核心线程，线程池就从 workQueue 中取任务并处理。
* 3 、当 workQueue 已存满，放不下新任务时则新建非核心线程入池，并处理请求直到线程数目达到 maximumPoolSize（最大线程数量设置值）。
* 4、如果线程池中线程数大于 maximumPoolSize 则使用 RejectedExecutionHandler 来进行任务拒绝处理。

## 任务队列 BlockingQueue

任务队列 workQueue 是用于存放不能被及时处理掉的任务的一个队列，它是 一个 BlockingQueue 类型。

关于 BlockingQueue，虽然它是 Queue 的子接口，但是它的主要作用并不是容器，而是作为线程同步的工具，他有一个特征，当生产者试图向 BlockingQueue 放入(put)元素，如果队列已满，则该线程被阻塞；当消费者试图从 BlockingQueue 取出(take)元素，如果队列已空，则该线程被阻塞。(From 疯狂Java讲义)

## 任务拒绝类型

* ThreadPoolExecutor.AbortPolicy:

当线程池中的数量等于最大线程数时抛 java.util.concurrent.RejectedExecutionException 异常，涉及到该异常的任务也不会被执行，线程池默认的拒绝策略就是该策略。

* ThreadPoolExecutor.DiscardPolicy():

当线程池中的数量等于最大线程数时,默默丢弃不能执行的新加任务，不报任何异常。

* ThreadPoolExecutor.CallerRunsPolicy():

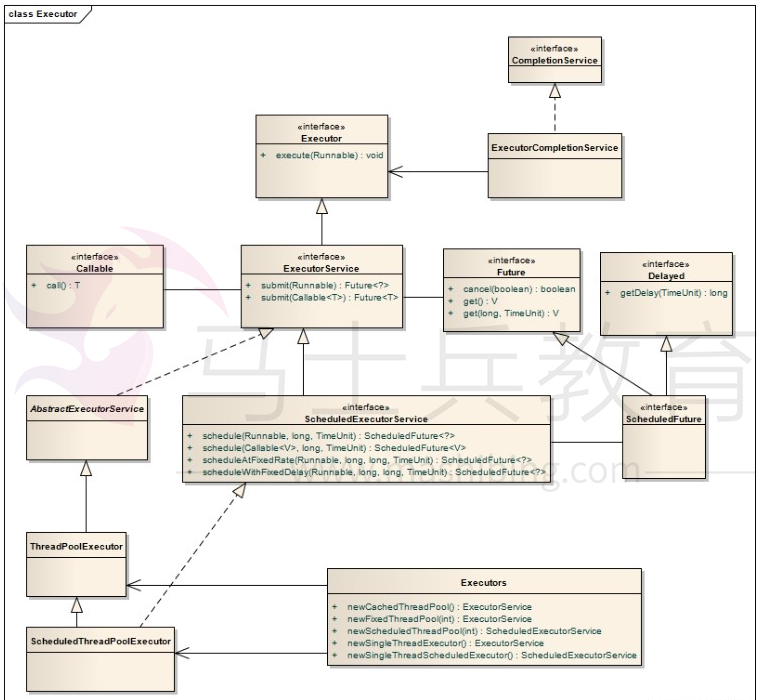
当线程池中的数量等于最大线程数时，重试添加当前的任务；它会自动重复调用execute()方法。

* ThreadPoolExecutor.DiscardOldestPolicy():

当线程池中的数量等于最大线程数时,抛弃线程池中工作队列头部的任务(即等待时间最久的任务)，并执行新传入的任务。

## 4种线程池

Java 里面线程池的顶级接口是 **Executor**，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 **ExecutorService**。



**Executors 线程池工具类给我们提供了一些API,用于返回一些不同的线程池**

**newCachedThreadPool**

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。

**newFixedThreadPool**

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。在任意点，在大多数 nThreads 线程会处于处理任务的活动状态。如果在所有线程处于活动状态时提交附加任务，则在有可用线程之前，附加任务将在队列中等待。如果在关闭前的执行期间由于失败而导致任何线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。在某个线程被显式地关闭之前，池中的线程将一直存在。

**newScheduledThreadPool**

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

\ScheduledExecutorService scheduledThreadPool= Executors.newScheduledThreadPool(3);  
scheduledThreadPool.schedule(newRunnable(){  
@Override  
public void run() { System.out.println("延迟三秒"); }  
}, 3, TimeUnit.SECONDS);  
scheduledThreadPool.scheduleAtFixedRate(newRunnable(){  
@Override  
public void run() {  
System.out.println("延迟 1 秒后每三秒执行一次");  
}  
},1,3,TimeUnit.SECONDS);

**newSingleThreadExecutor**

Executors.newSingleThreadExecutor()返回一个线程池（这个线程池只有一个线程）,这个线程池可以在线程死后（或发生异常时）重新启动一个线程来替代原来的线程继续执行下去！

# 为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗？

## 栈是运行时的单位，而堆是存储的单位。

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

1. 从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。
2. 堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。
3. 栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得动态增长成为可能，相应栈中只需记录堆中的一个地址即可。
4. 面向对象就是堆和栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

# 为什么不把基本类型放堆中呢？

因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

# 堆中存什么？栈中存什么？

堆中存的是对象。栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在栈中，一个对象只对应了一个4btye的引用（堆栈分离的好处：）。

# Java中的参数传递时传值呢？还是传引用？

|  |
| --- |
| */\*\*  \** ***@Author:*** *Ma HaiYang  \** ***@Description:*** *MircoMessage:Mark\_7001  \*/* **public class** Test1 {  **public static void** main(String[] args) {  */\*int[] arr ={0};  change2(arr);  System.out.println(arr[0]);\*/   /\* String s ="bbb";  changeStr(s);  System.out.println(s);\*/* Integer i =10;  *changeInteger*(i);  System.***out***.println(i);  }  **public static void** changeInteger(Integer i){  i=100;  }    **public static void** changeStr(String s){  s=**"aaaa"**;  }    **public static void** change(**int**[] arr){  arr[0]=100;  }  **public static void** change2(**int**[] arr){  arr=**new int**[]{100};  }    } |

参数是基本数据类型 按值传递

参数是引用类型 按引用传递

按引用传递在方法体重修改形参时,可能会对实参产生影响, 注意坑 修改的方式,String 包装类诶

# 如何确定垃圾

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。

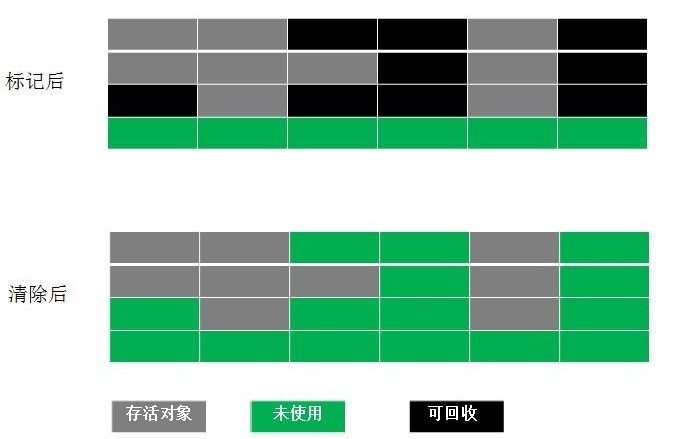
可达性分析

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。

# 如何清除垃圾

## 标记清除算法（Mark-Sweep）

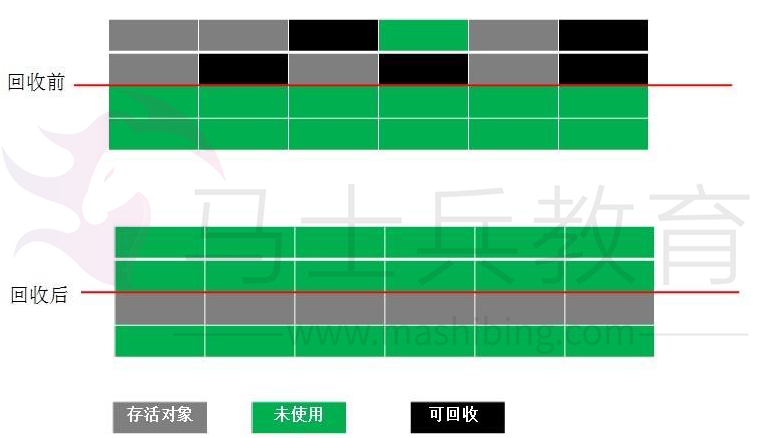
最基础的垃圾回收算法，分为两个阶段，标注和清除。标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间。如图



我们就可以发现，该算法最大的问题是内存碎片化严重，后续可能发生大对象不能找到可利用空间的问题。

## 复制算法（copying）

为了解决 Mark-Sweep 算法内存碎片化的缺陷而被提出的算法。按内存容量将内存划分为等大小的两块。每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用的内存清掉，如图：



算法虽然实现简单，内存效率高，不易产生碎片，但是最大的问题是可用内存被压缩到了原本的一半。且存活对象增多的话，Copying 算法的效率会大大降低。

## 标记整理算法(Mark-Compact)

结合了以上两个算法，为了避免缺陷而提出。标记阶段和 Mark-Sweep 算法相同，标记后不是清理对象，而是将存活对象移向内存的一端。然后清除端边界外的对象。如图：



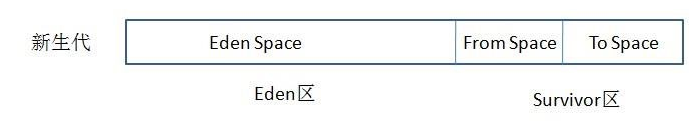
## 分代收集算法

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(Young Generation)。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。

### 新生代与复制算法

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。



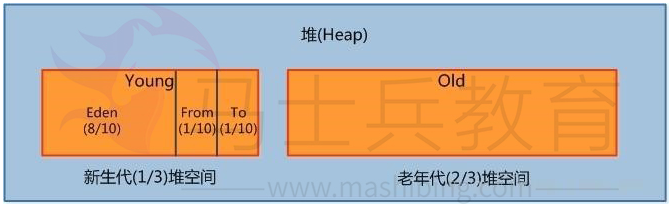
### 老年代与标记复制算法

而老年代因为每次只回收少量对象，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，Eden Space 和 From Space 区的存活对象会被挪到 To Space，然后将 Eden Space 和 From Space 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。

# java堆内存分代

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。



## 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。

1. 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。

2. ServivorFrom,上一次 GC 的幸存者，作为这一次 GC 的被扫描者。

3. ServivorTo 保留了一次 MinorGC 过程中的幸存者。

4. MinorGC 的过程（复制->清空->互换）

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）；

然后，清空 Eden 和 ServicorFrom 中的对象；

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 不会频繁执行。在进行 MajorGC 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。

MajorGC 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。MajorGC 的耗时比较长，因为要扫描再回收。MajorGC 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

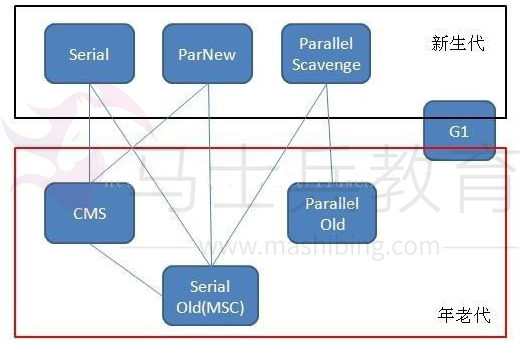
指内存的永久保存区域，主要存放 Class 和 Meta（元数据）的信息,Class 在被加载的时候被放入永久区域，它和和存放实例的区域不同,GC 不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的 Class 的增多而胀满，最终抛出 OOM 异常。

## 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。

# 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：



## 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。

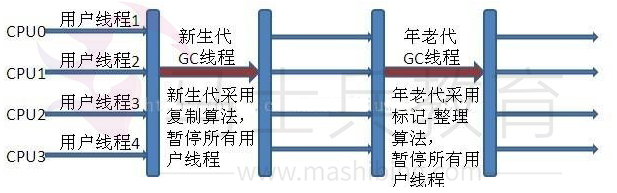
## Serial Old 收集器（单线程标记整理算法 ）

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：

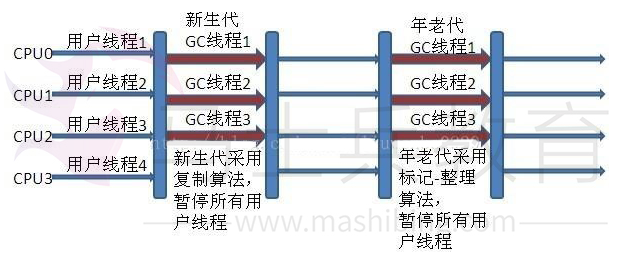
1在 JDK1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。

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：



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 Parallel Scavenge/ParNew 与年老代 Serial Old 搭配垃圾收集过程图：



## 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**

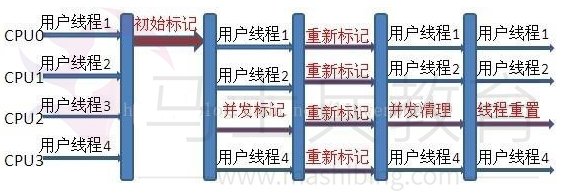
为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，仍然需要暂停所有的工作线程。

**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。

# 事务

什么是事务: 事务逻辑上的一组操作,组成这组操作的各个逻辑单元,要么一起成功,要么一起失败.

## 事务特性（4种）:

* 原子性 （atomicity）:强调事务的不可分割.
* 一致性 （consistency）:事务的执行的前后数据的完整性保持一致.
* 隔离性 （isolation）:一个事务执行的过程中,不应该受到其他事务的干扰
* 持久性（durability） :事务一旦结束,数据就持久到数据库

## 引发安全性问题:

脏读 :一个事务读到了另一个事务的未提交的数据

不可重复读 :一个事务读到了另一个事务已经提交的 update 的数据导致多次查询结果不一致.

虚幻读 :一个事务读到了另一个事务已经提交的 insert 的数据导致多次查询结果不一致.

## 解决读问题: 设置事务隔离级别

DEFAULT 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔 离级别.

未提交读（read uncommited） :脏读，不可重复读，虚读都有可能发生

已提交读 （read commited）:避免脏读。但是不可重复读和虚读有可能发生

可重复读 （repeatable read） :避免脏读和不可重复读.但是虚读有可能发生.

串行化的 （serializable） :避免以上所有读问题.

Mysql 默认:可重复读

Oracle 默认:读已提交

## 事务的传播行为

PROPAGION\_XXX :事务的传播行为

* 保证同一个事务中

PROPAGATION\_REQUIRED 支持当前事务，如果不存在 就新建一个(默认)

PROPAGATION\_SUPPORTS 支持当前事务，如果不存在，就不使用事务

PROPAGATION\_MANDATORY 支持当前事务，如果不存在，抛出异常

* 保证没有在同一个事务中

PROPAGATION\_REQUIRES\_NEW 如果有事务存在，挂起当前事务，创建一个新的事务

PROPAGATION\_NOT\_SUPPORTED 以非事务方式运行，如果有事务存在，挂起当前事务

PROPAGATION\_NEVER 以非事务方式运行，如果有事务存在，抛出异常

PROPAGATION\_NESTED 如果当前事务存在，则嵌套事务执行

# Mybatis 的一级缓存原理（sqlsession 级别）

第一次发出一个查询 sql，sql 查询结果写入 sqlsession 的一级缓存中，缓存使用的数据结构是一个 map。

key：MapperID+offset+limit+Sql+所有的入参

value：用户信息

同一个 sqlsession 再次发出相同的 sql，就从缓存中取出数据。如果两次中间出现 commit 操作（修改、添加、删除），本 sqlsession 中的一级缓存区域全部清空，下次再去缓存中查询不到所以要从数据库查询，从数据库查询到再写入缓存。

二级缓存原理（mapper 基本）

二级缓存的范围是mapper 级别（mapper同一个命名空间），mapper以命名空间为单位创建缓存数据结构，结构是 map。mybatis 的二级缓存是通过 CacheExecutor 实现的。CacheExecutor 其实是 Executor 的代理对象。所有的查询操作，在 CacheExecutor 中都会先匹配缓存中是否存在，不存在则查询数据库。

key：MapperID+offset+limit+Sql+所有的入参

具体使用需要配置：

1. Mybatis 全局配置中启用二级缓存配置
2. 在对应的 Mapper.xml 中配置 cache 节点
3. 在对应的 select 查询节点中添加 useCache=true

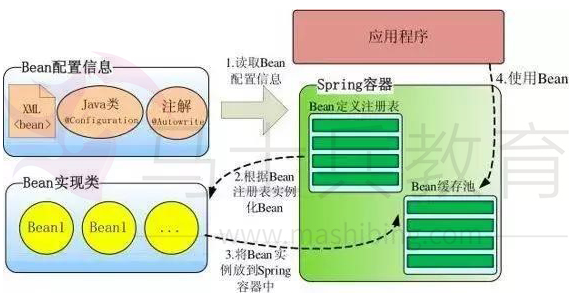
# Spring IOC 原理

**概念**

Spring 通过一个配置文件描述 Bean 及 Bean 之间的依赖关系，利用 Java 语言的反射功能实例化 Bean 并建立 Bean 之间的依赖关系。 Spring 的 IoC 容器在完成这些底层工作的基础上，还提供了 Bean 实例缓存、生命周期管理、 Bean 实例代理、事件发布、资源装载等高级服务。

**Spring容器高层视图**

Spring 启动时读取应用程序提供的 Bean 配置信息，并在 Spring 容器中生成一份相应的 Bean 配置注册表，然后根据这张注册表实例化 Bean，装配好 Bean 之间的依赖关系，为上层应用提供准备就绪的运行环境。其中 Bean 缓存池为 HashMap 实现



优点

ioc的思想最核心的地方在于，资源不由使用资源的双方管理，而由不使用资源的第三方管理， 这可以带来很多好处：

1）资源集中管理，实现资源的可配置和易管理

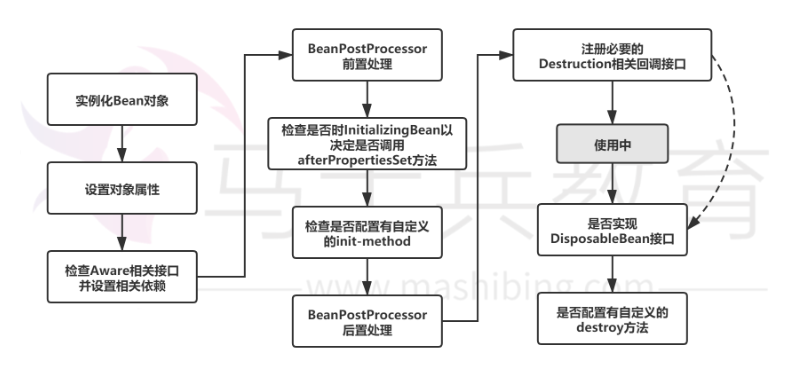
2）降低了使用资源双方的依赖程度，也就是我们说的耦合度

# Spring bean的作用域和生命周期；

## 作用域



## 生命周期



# Spring AOP 原理

**概念**

"横切"的技术，剖解开封装的对象内部，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，

并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，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，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

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，而各处基本相似，比如权限认证、日志、事物。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。

AOP 主要应用场景有：事务管理、安全检查、权限控制、数据校验、缓存、对象池管理等

1. Authentication 权限
2. Caching 缓存
3. Context passing 内容传递
4. Error handling 错误处理
5. Lazy loading 懒加载
6. Debugging 调试
7. logging, tracing, profiling and monitoring 记录跟踪 优化 校准
8. Performance optimization 性能优化
9. Persistence 持久化
10. Resource pooling 资源池
11. Synchronization 同步
12. Transactions 事务

## AOP 核心概念

1、切面（aspect）：类是对物体特征的抽象，切面就是对横切关注点的抽象

2、横切关注点：对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点。

3、连接点（joinpoint）：被拦截到的点，因为 Spring 只支持方法类型的连接点，所以在 Spring 中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器。

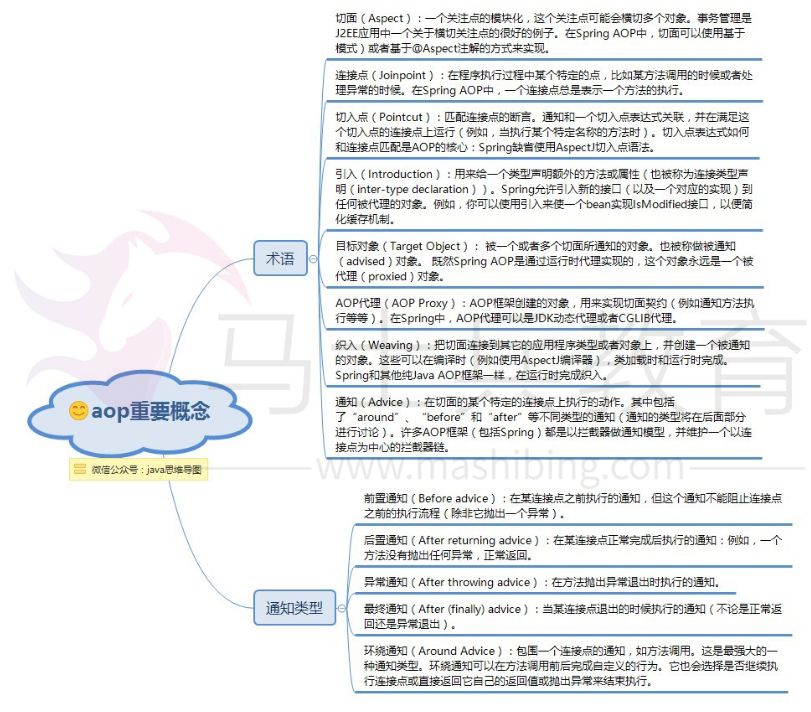
4、切入点（pointcut）：对连接点进行拦截的定义

5、通知（advice）：所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、最终、环绕通知五类。

6、目标对象：代理的目标对象

7、织入（weave）：将切面应用到目标对象并导致代理对象创建的过程

8、引入（introduction）：在不修改代码的前提下，引入可以在运行期为类动态地添加一些方法或字段。



## AOP 两种代理方式

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。

**JDK**动态接口代理

1. JDK 动态代理主要涉及到 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。

InvocationHandler是一个接口，通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态将横切逻辑和业务逻辑编制在一起。Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例，生成目标类的代理对象。

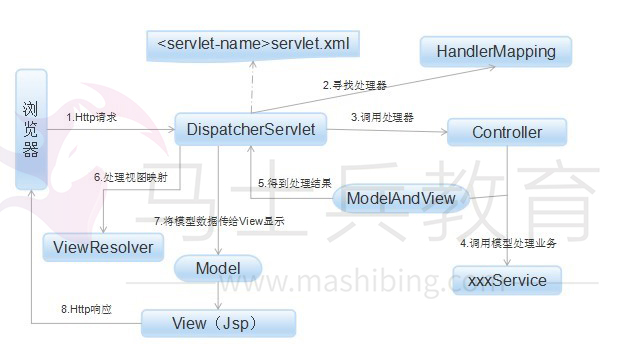
**CGLib** 动态代理

1. ：CGLib 全称为 Code Generation Library，是一个强大的高性能，高质量的代码生成类库，可以在运行期扩展 Java 类与实现 Java 接口，CGLib 封装了 asm，可以再运行期动态生成新的 class。和 JDK 动态代理相比较：JDK 创建代理有一个限制，就是只能为接口创建代理实例，而对于没有通过接口定义业务方法的类，则可以通过 CGLib 创建动态代理。

# Spring MVC 原理

Spring 的模型-视图-控制器（MVC）框架是围绕一个 DispatcherServlet 来设计的，这个 Servlet 会把请求分发给各个处理器，并支持可配置的处理器映射、视图渲染、本地化、时区与主题渲染等，甚至还能支持文件上传。

**MVC** 流程



**Http** 请求到**DispatcherServlet**

(1) 客户端请求提交到 DispatcherServlet。

**HandlerMapping** 寻找处理器

(2) 由 DispatcherServlet 控制器查询一个或多个 HandlerMapping，找到处理请求的

Controller。

调用处理器**Controller**

(3) DispatcherServlet 将请求提交到 Controller。

**Controller** 调用业务逻辑处理后，返回**ModelAndView**

(4)(5)调用业务处理和返回结果：Controller 调用业务逻辑处理后，返回 ModelAndView。

**DispatcherServlet** 查询**ModelAndView**

(6)(7)处理视图映射并返回模型： DispatcherServlet 查询一个或多个 ViewResoler 视图解析器，找到 ModelAndView 指定的视图。

**ModelAndView** 反馈浏览器**HTTP**

(8) Http 响应：视图负责将结果显示到客户端。

# Spring Boot比Spring做了哪些改进？

1）Spring Boot可以建立独立的Spring应用程序；

2）内嵌了如Tomcat，Jetty和Undertow这样的容器，也就是说可以直接跑起来，用不着再做 部署工作了；

3）无需再像Spring那样搞一堆繁琐的xml文件的配置；

4）可以自动配置Spring。SpringBoot将原有的XML配置改为Java配置，将bean注入改为使 用注解注入的方式(@Autowire)，并将多个xml、properties配置浓缩在一个appliaction.yml 配置文件中。

5）提供了一些现有的功能，如量度工具，表单数据验证以及一些外部配置这样的一些第三方功能；

6）整合常用依赖（开发库，例如spring-webmvc、jackson-json、validation-api和tomcat 等），提供的POM可以简化Maven的配置。当我们引入核心依赖时，SpringBoot会自引入其 他依赖。

# 关于intern

String a=**new** String(**"123"**)+**new** String(**"456"**);  
//String b=**new** String(**"123456"**);  
String intern = a.intern();  
System.***out***.println(intern==a);

注释输出true,取消注释 输出false

<https://blog.csdn.net/qq_41884976/article/details/83353389>

