JVM

数据类型

基本类型

byte

short

int

long

char

float

double

boolean

引用类型

类类型

接口类型

数组

堆与栈

栈

栈是运行时的单位

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

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

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

栈中存的是基本数据类型和堆中对象的引用。

在栈中，一个对象只对应了一个4btye的引用（堆栈分离的

好处：））。

堆

堆是存储的单位

堆解决的是数据存储的问题，即数据怎么放、放在哪儿

堆是所有线程共享的

堆只负责存储对象信息

堆中存的是对象

一个对象的大小是不可估计的，或者说是可以动态变化的

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？

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。

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。

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。

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？

占用的空间一般是1~8个字节——需要空间比较少

因为是基本类型，所以不会出现动态增长的情况——长度固定

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

Java中的参数传递时传值呢？还是传引用?

不要试图与C进行类比，Java中没有指针的概念

程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

堆和栈中，栈是程序运行最根本的东西。程序运行可以没有堆，但是不能没有栈。而堆是为栈进行数据存储服务，说白了堆就是一块共享的内存。不过，正是因为堆和栈的分离的思想，才使得Java的垃圾回收成为可能。

Java中，栈的大小通过-Xss来设置，当栈中存储数据比较多时，需要适当调大这个值，否则会出现java.lang.StackOverflowError异常。常见的出现这个异常的是无法返回的递归，因为此时栈中保存的信息都是方法返回的记录点。

Java对象的大小

基本数据的类型的大小是固定的

非基本类型的Java对象

在Java中，一个空Object对象的大小是8byte，这个大小只是保存堆中一个没有任何属性的对象的大小

它所占的空间为：4byte+8byte。4byte是上面部分所说的Java栈中保存引用的所需要的空间。而那8byte则是Java堆中对象的信息。因为所有的Java非基本类型的对象都需要默认继承Object对象，因此不论什么样的Java对象，其大小都必须是大于8byte。

在栈中，一个对象只对应了一个4btye的引用（堆栈分离的好处：）

如何分代

年轻代

Eden区

Survivor区

Survivor区

年老代

在年轻代经历了n次垃圾回收仍然存活的对象将被复制到老年代。因此可以认为，老年代存放的对象都是生命周期较长的对象。

持久代

用于存放静态问题，如java类和方法，常量池。持久代对垃圾回收没有显著影响。

GC

Scavenge GC

一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使Eden去能尽快空闲出来。

Full GC

对整个堆进行整理，包括新生代，老年代和持久代。Full GC要对整个堆进行整理，因此要比Scvaenge GC慢，因此应该尽可能的减少Full GC 的次数。在对JVM调优的工作中，很大一部分就是对Full GC的调节。

垃圾回收算法

按照基本回收策略分

引用计数（Reference Counting）

比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为0的对象。此算法最致命的是无法处理循环引用的问题。

此算法执行分两阶段。第一阶段从引用根节点开始标记所

有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

复制（Copying）

此算法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。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。次算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。

此算法的缺点也是很明显的，就是需要两倍内存空间。

标记-整理（Mark-Compact）

此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。

此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

按分区对待的方式分

增量收集（Incremental Collecting）

实时垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因JDK5.0中的收集器没有使用这种算法的。

分代收集（Generational Collecting）

基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从J2SE1.2开始）都是使用此算法的。

按系统线程分

串行收集

串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。

其局限性也比较明显，即无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小

数据量（100M左右）情况下的多处理器机器上。

并行收集

并行收集使用多线程处理垃圾回收工作，因而速度快，效率高。而且理论上CPU数目越多，越能体现出并行收集器的优势。

并发收集

相对于串行收集和并行收集而言，前面两个在进行垃圾回收工作时，需要暂停整个运行环境，而只有垃圾回收程序在运行，因此，系统在垃圾回收时会有明显的暂停，而且暂停时间会因为堆越大而越长。