# 虚拟机中的对象

## 对象的分配

虚拟机遇到一条new指令时：根据new的参数是否能在常量池中定位到一个类的符号引用,如果没有，说明还未定义该类，抛出ClassNotFoundException；

### 1）检查加载

先执行相应的类加载过程。如果没有，则进行类加载

### 分配内存

根据方法区的信息确定为该类分配的内存空间大小

#### 指针碰撞 (java堆内存空间规整的情况下使用)

接下来虚拟机将为新生对象分配内存。为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。

如果Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“**指针碰撞**”。

#### 空闲列表 (java堆空间不规整的情况下使用)

如果Java堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“**空闲列表**”。

选择哪种分配方式由Java堆是否规整决定，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。

#### 并发安全

除如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，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。

##### CAS机制

解决这个问题有两种方案，一种是对分配内存空间的动作进行同步处理——实际上虚拟机采用CAS配上失败重试的方式保证更新操作的原子性；

##### 分配缓冲

另一种是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私有内存，也就是本地线程分配缓冲（Thread Local Allocation Buffer,TLAB），如果设置了虚拟机参数 -XX:+UseTLAB，在线程初始化时，同时也会申请一块指定大小的内存，只给当前线程使用，这样每个线程都单独拥有一个Buffer，如果需要分配内存，就在自己的Buffer上分配，这样就不存在竞争的情况，可以大大提升分配效率，当Buffer容量不够的时候，再重新从Eden区域申请一块继续使用。

TLAB的目的是在为新对象分配内存空间时，让每个Java应用线程能在使用自己专属的分配指针来分配空间（Eden区，默认Eden的1%），减少同步开销。

TLAB只是让每个线程有私有的分配指针，但底下存对象的内存空间还是给所有线程访问的，只是其它线程无法在这个区域分配而已。当一个TLAB用满（分配指针top撞上分配极限end了），就新申请一个TLAB。

### 3）内存空间初始化

（注意不是构造方法）内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值(如int值为0，boolean值为false等等)。这一步操作保证了对象的实例字段在Java代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

### 4）设置

接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。这些信息存放在对象的对象头之中。

### 5）对象初始化

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从Java程序的视角来看，对象创建才刚刚开始，所有的字段都还为零值。所以，一般来说，执行new指令之后会接着把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

# 对象的内存布局

在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。

对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

第三部分对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。由于HotSpot VM的自动内存管理系统要求对对象的大小必须是8字节的整数倍。对象正好是9字节的整数，所以当对象其他数据部分（对象实例数据）没有对齐时，就需要通过对齐填充来补全。

# 对象的访问定位

建立对象是为了使用对象，我们的Java程序需要通过栈上的reference数据来操作堆上的具体对象。目前主流的访问方式有使用句柄和直接指针两种。

## 句柄

如果使用句柄访问的话，那么Java堆中将会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。

## 直接指针

如果使用直接指针访问， reference中存储的直接就是对象地址。

这两种对象访问方式各有优势，使用句柄来访问的最大好处就是reference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reference本身不需要修改。

使用直接指针访问方式的最大好处就是速度更快，它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一项非常可观的执行成本。

对Sun HotSpot而言，它是使用直接指针访问方式进行对象访问的。

# 堆内存分配策略

新生代

Eden区

Survivor(from)区：

设置Survivor是为了减少送到老年代的对象

Survivor(to)区：

设置两个Survivor区是为了解决碎片化的问题（复制回收算法）

## 对象优先在Eden区分配

虚拟机参数：

-Xms20m

-Xmx20m

-Xmn10m

-XX:+PrintGCDetails

-XX:+PrintGCDetails 打印垃圾回收日志，程序退出时输出当前内存的分配情况

注意：新生代初始时就有大小

大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够空间分配时，虚拟机将发起一次Minor GC。

## 大对象直接进入老年代

-Xms20m

-Xmx20m

-Xmn10m

-XX:+PrintGCDetails

-XX:PretenureSizeThreshold=4m

-XX:+UseSerialGC

PretenureSizeThreshold参数只对Serial和ParNew两款收集器有效。

最典型的大对象是那种很长的字符串以及数组。这样做的目的：1.避免大量内存复制,2.避免提前进行垃圾回收，明明内存有空间进行分配。

## 长期存活对象进入老年区

如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并将对象年龄设为1，对象在Survivor区中每熬过一次 Minor GC，年龄就增加1，当它的年龄增加到一定程度(默认为15)\_时，就会被晋升到老年代中。

## 对象年龄动态判定

如果在 Survivor空间中相同年龄所有对象大小的综合大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代

## 空间分配担保

在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的，如果担保失败则会进行一次Full GC；如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC。

# Java中的泛型

## 泛型是什么

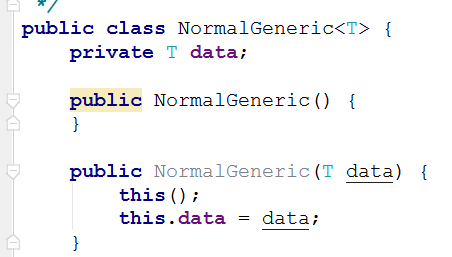
泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？

顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

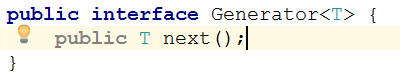
引入一个类型变量T（其他大写字母都可以，不过常用的就是T，E，K，V等等），并且用<>括起来，并放在类名的后面。泛型类是允许有多个类型变量的。

### 泛型类



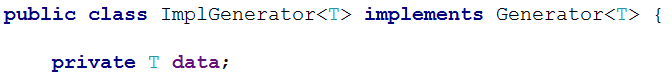
### 泛型接口

泛型接口与泛型类的定义基本相同。



而实现泛型接口的类，有两种实现方法：

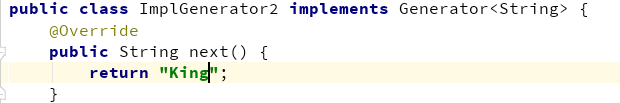
1、未传入泛型实参时：



在new出类的实例时，需要指定具体类型：



2、传入泛型实参



在new出类的实例时，和普通的类没区别。

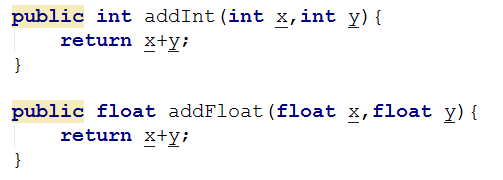
### 泛型方法



泛型方法，是在调用方法的时候指明泛型的具体类型 ，泛型方法可以在任何地方和任何场景中使用，包括普通类和泛型类。

## 为什么我们需要泛型？

通过两段代码我们就可以知道为何我们需要泛型



实际开发中，经常有数值类型求和的需求，例如实现int类型的加法, 有时候还需要实现long类型的求和, 如果还需要double类型的求和，需要重新在重载一个输入是double类型的add方法。

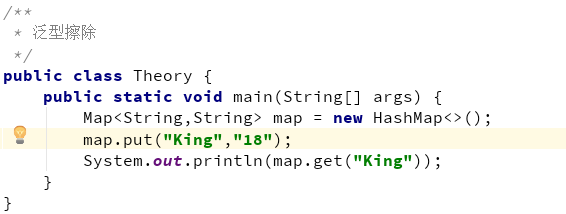
所以泛型的好处就是：

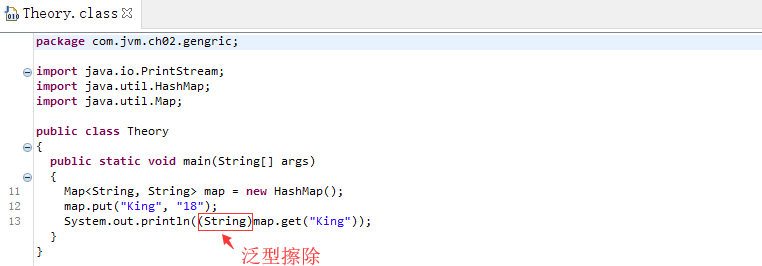
* 适用于多种数据类型执行相同的代码
* 泛型中的类型在使用时指定，不需要强制类型转换

## 虚拟机是如何实现泛型的？

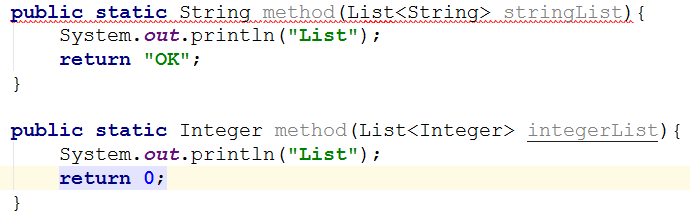
Java语言中的泛型，它只在程序源码中存在，在编译后的字节码文件中，就已经替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了强制转型代码，因此，对于运行期的Java语言来说，ArrayList＜int＞与ArrayList＜String＞就是同一个类，所以泛型技术实际上是Java语言的一颗语法糖，Java语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型称为伪泛型。

将一段Java代码编译成Class文件，然后再用字节码反编译工具进行反编译后，将会发现泛型都不见了，程序又变回了Java泛型出现之前的写法，泛型类型都变回了原生类型（因为）





### 使用泛型注意事项（小甜点，了解即可，装B专用）



上面这段代码是不能被编译的，因为参数List＜Integer＞和List＜String＞编译之后都被擦除了，变成了一样的原生类型List＜E＞，擦除动作导致这两种方法的特征签名变得一模一样（注意在IDEA中是不行的，但是jdk的编译器是可以，因为jdk是根据方法返回值+方法名+参数）。

JVM版本兼容性问题：JDK1.5以前，为了确保泛型的兼容性，JVM除了擦除，其实还是保留了泛型信息(Signature是其中最重要的一项属性，它的作用就是存储一个方法在字节码层面的特征签名，这个属性中保存的参数类型并不是原生类型，而是包括了参数化类型的信息)----弱记忆

另外，从Signature属性的出现我们还可以得出结论，擦除法所谓的擦除，仅仅是对方法的Code属性中的字节码进行擦除，实际上元数据中还是保留了泛型信息，这也是我们能通过反射手段取得参数化类型的根本依据。