

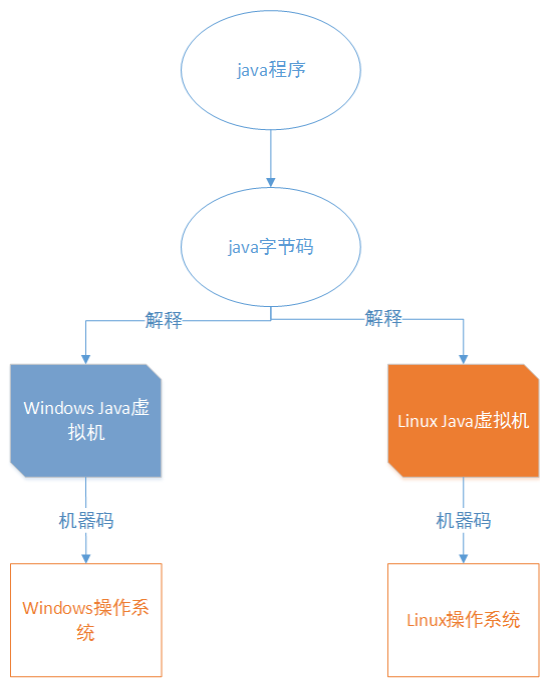
JVM浅谈

啥是JVM

JVM就是java vary mass的缩写，意为Java混乱而复杂，用以示意初学者别学java。JVM就是Java Virtual Machine，也就是java虚拟机的意思，let's 康康JVM的作用和结构

JVM的作用

java有一句非常强悍的宣传语：编译一次，到处运行。这里的到处运行的含义在于java可以在不同的操作系统进行运行，Windows和Linux都有属于自己的jvm，这样的话我们的java程序直接面对的就是java虚拟机，每一个java程序都是在找jvm，而不同平台的jvm对我们的代码的兼容是性是相同的，而具体的与系统层面的交互则由jvm完成，我感觉可以说java是一种jvm虚拟机上面的语言，用什么系统与java并无太大关系，只要你有这个系统平台的jvm，java就能上，具体的和系统的交互交给jvm就好了。当然，JVM平台上也并不只Java一门语言，正在折磨我的Kotiln也是JVM平台的红人。



Java的编译过程

在探索java的编译过程之前来看两个术语：

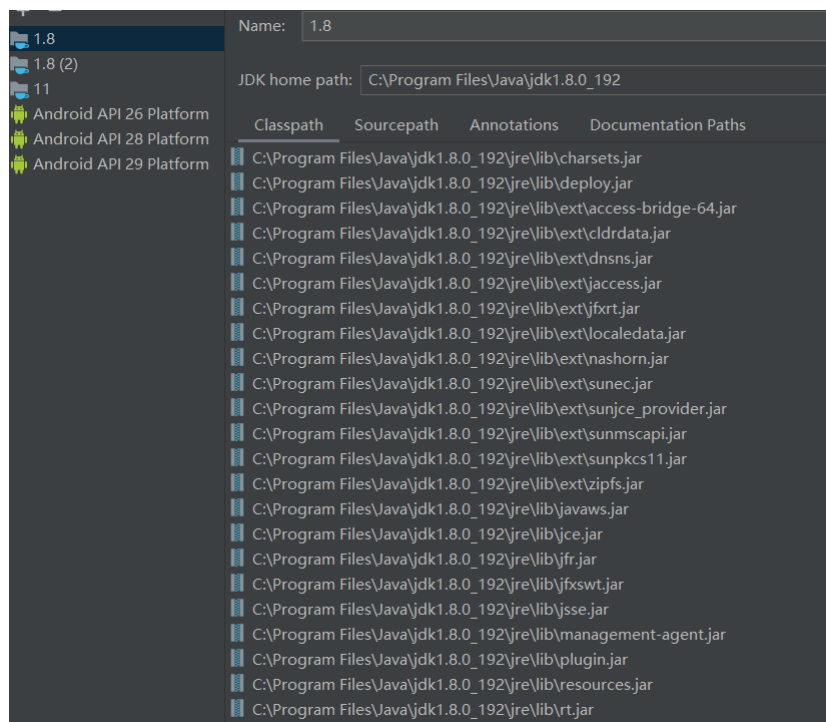
JRE: Java Runtime Environment, java运行环境

JDK: Java Development Kit, java开发工具包

这个是我的电脑的JDK安装目录，其中的bin文件夹可以理解为JVM的所在地，lib则是JVM所需的类库

bin	2019-09-13 9:08	文件夹	
lib	2019-09-13 9:08	文件夹	
COPYRIGHT	2019-09-13 9:08	文件	4 KB
LICENSE	2019-09-13 9:08	文件	1 KB
README.txt	2019-09-13 9:08	文本文档	1 KB
THIRDPARTYLICENSEREADME.txt	2019-09-13 9:08	文本文档	152 KB
THIRDPARTYLICENSEREADME-JAVAF...	2019-09-13 9:08	文本文档	106 KB
Welcome.html	2019-09-13 9:08	Chrome HTML D...	1 KB

在IDEA的Project Structure里面，我们可以看到大量的jar包存放在lib文件夹下，JVM+lib就构成了JRE。



接下来看一下Java自代码变为可执行的程序的过程，先展示一段复杂而深刻的代码：

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Hello world");  
    }  
}
```

现在我们点击一下build按钮，编译过后将在工程对应的out文件夹下找到Main.class文件

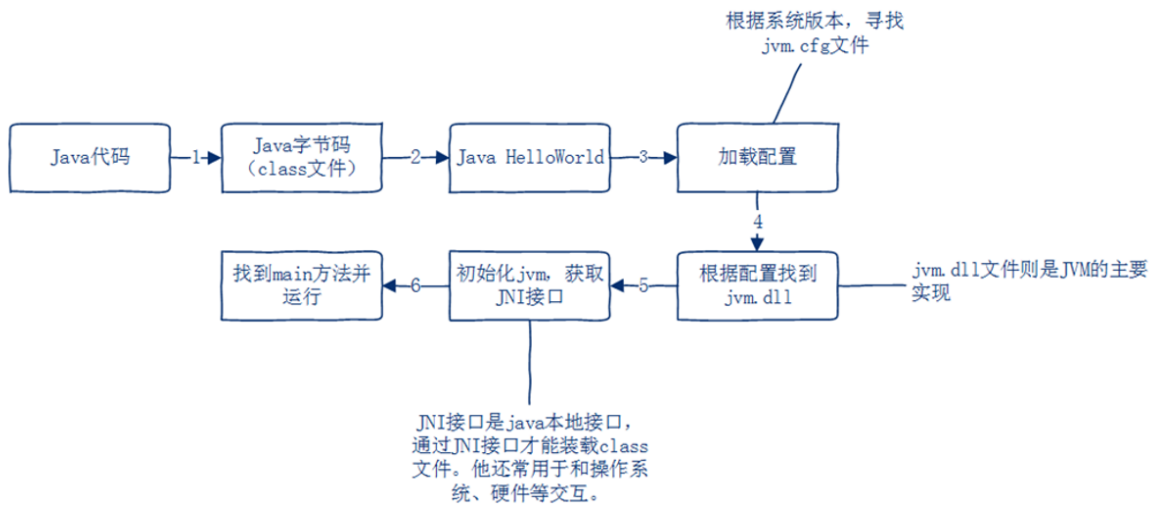
此电脑 > Data (D:) > RayleighZ_javaPrj > Jvm_test > out > production > Jvm_test				
名称	修改日期	类型	大小	
Main.class	2020-07-06 16:48	CLASS 文件	2 KB	

之后这个class文件需要找到JVM所在地，他找到JVM的向导就是前文截图中在bin文件夹里面的java.exe程序，他会通过自己的规则找到JVM的位置，将class文件托付给jvm

之后会经过jvm.cfg文件，此文件将配置JVM的一些参数，比如运行状态是-server还是-client，这两个模式在启动速度和启动后的运行效率上有所不同，GC机制上也有所不同。

此电脑 > Windows-SSD (C:) > Program Files > Java > jre1.8.0_251 > lib > amd64				
名称	修改日期	类型	大小	
jvm.cfg	2020-04-20 10:43	CFG 文件	1 KB	

之后我们的JVM就被启动运行了，初始化完成后他会找到main方法并进行运行，余下的就不再赘述(我也不会)。大体的一个流程见下图：



这里先提一嘴，学长说以后会有通识课讲这个，咱就不班门弄斧了

JVM的结构

基本的内存分区

简书上嫖的图片：



让我们来具体的看一看内存分区

- Java栈：Java栈中存储的是栈帧，栈帧中存储的则是**局部变量表**(Local Variables)、**操作数栈**(Operand Stack)、**指向当前方法所属的类的运行时常量池的引用**(Reference to runtime constant pool)、**方法返回地址**(Return Address)和一些额外的附加信息，说白了就是基本数据类型的变量和对象的引用，栈帧会伴随着线程对于方法的调用而产生并入栈，它有以下至关重要的特性：
 1. 被线程所私有，每一个线程拥有一个独立的栈，线程之间不可互相访问。
 2. 只存储基本数据类型(不包含包装类)和对象的引用。
 3. 不受JVM_GC的回收处理。

关于为什么要用栈这种存储结构，我的理解是栈这种结构天生适合进行描述函数的嵌套使用，诸如递归等算法感觉底层实现上使用的数据结构就是栈

- Java堆：堆内存里面存储的是**对象**和**数组**，注意，String以及基本类型的包装类也被存储在其中（这里有些说道，大家先记着，后面咱们再讨论），java的堆是线程共享的，所有的线程公用同一个堆，堆内存是会受JVM_GC的处理的。

- 方法区：在JVM中，**类型信息**和**类静态变量**都保存在方法区中，类型信息是由类加载器在类加载的过程中从类文件中提取出来的信息，不知道大家还记不记得扎金花，当时让玩的反射就是要获取Class对象，解析Class对象中的Field然后进行操作的，编译产生的Class对象被classloader提取出的类型信息就存储在方法区之中，此区域也是被线程共享的，并且同堆一样会被GC处理。

以下是类型信息的大致包含内容

- 1、类型的全名（The fully qualified name of the type）
- 2、类型的父类型全名（除非没有父类型，或者父类型是java.lang.Object）（The fully qualified name of the type's direct superclass）
- 3、该类型是一个类还是接口（class or an interface）（Whether or not the type is a class）
- 4、类型的修饰符（public, private, protected, static, final, volatile, transient等）（The type's modifiers）
- 5、所有父接口全名的列表（An ordered list of the fully qualified names of any direct superinterfaces）
- 6、类型的字段信息（Field information）
- 7、类型的方法信息（Method information）
- 8、所有静态类变量（非常量）信息（All class (static) variables declared in the type, except constants）
- 9、一个指向类加载器的引用（A reference to class ClassLoader）
- 10、一个指向Class类的引用（A reference to class Class）
- 11、基本类型的常量池（The constant pool for the type）

- 本地方法栈：在调用一些非Java语言的本地方法接口时使用此栈，比如那些底层实现使用的是C的接口
- 程序计数器：它是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。这里先了解一下，待会再谈这个东西的具体作用

稍微写两行代码理解一下上面这些东西

```
public class Main {
    public static void main(String[] args) {
        int a = 1;
        int b = 1;
        System.out.println( a == b );//true, 值比较
        Integer c = 1;
        Integer d = 1;
        Integer e = new Integer(1);
        System.out.println( c == d );//true, 引用比较
        System.out.println( c == e );//false, 引用比较
        System.out.println( c.equals(e) );//true, 值比较
        d = 2;
        System.out.println( c == d );//false, 引用比较
    }
}
```

让我从前往后的捋一捋，第一个true和第五个true是显然的，第二个true显得有点怪异（当然在座的诸位大佬看起来可能也是非常显然的），第三个false用法有点怪怪的，但是仔细一想也对，最后一个false就又非常怪异了，我们就着重于看这三行怪异的比较

首先==号对于引用数据类型而言相当于比较指向的对象的内存地址，c==d这里的true说明c和d两个不同的引用指向的是同一个对象，这是我们得出的结论，但是比较神奇的是如果我将c和d同时赋值为128，那么结果就一定是false，问题就变得更加奇怪了，同样的Integer初始化方式居然得到了相反的结果..... 这里小弟就不卖关子了，其实上学期也有学长讲过了，我也就是在这儿拾人牙慧一下罢了，直接呈上源码给各位瞧瞧

```
/**
 * Cache to support the object identity semantics of autoboxing for values
 * between
 * -128 and 127 (inclusive) as required by JLS.
 *
 * The cache is initialized on first usage. The size of the cache
 * may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} option.
 * During VM initialization, java.lang.Integer.IntegerCache.high property
 * may be set and saved in the private system properties in the
 * sun.misc.VM class.
 */

private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[]; //注意啊。。这里的Integer数组是final定义的

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =

sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high"); //获得人为定
义的缓存上界
        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); //防止上限比下限还低
            } 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() {}
}
```

所以说啊，在默认情况下-128 ~ 127是缓存在常量池之中的，如果Integer被赋值到这个区间内的值的话就会指向相同的地址区间，这样为什么c==d的输出结果的true以及为什么改成128之后是false就迎刃而解了。

再让我们仔细看看下面的代码

```
/**
 * Returns an {@code Integer} instance representing the specified
 * {@code int} value. If a new {@code Integer} instance is not
 * required, this method should generally be used in preference to
 * the constructor {@link #Integer(int)}, as this method is likely
 * to yield significantly better space and time performance by
 * caching frequently requested values.
 *
 * This method will always cache values in the range -128 to 127,
 * inclusive, and may cache other values outside of this range.
 *
 * @param i an {@code int} value.
 * @return an {@code Integer} instance representing {@code i}.
 * @since 1.5
 */
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

这个方法就说明了Integer对象被声明的方法区别，如果是在缓存区之内，就直接将缓存数组里面的对应单元返回过去，反之就new一个新对象（这个Integer的构造方法就是将value赋值为i，没啥特殊的）

好！到此为止我们似乎就get到了Integer声明的全部知识点！但是可能有的兄弟就要问了：“你这贴的是valueOf这个类方法，它和Integer对象的声明有啥关系啊，我只看见了Integer a = 1; 你贴的这行代码我没看见啊”。

与其说是有的兄弟问，其实是因为我写稿的时候自己想不明白🤔，其实很简单啦，就是Java的自动装箱啦，大家应该都懂，这里贴一下没有必要的手动装箱写法

```
Integer a = Integer.valueOf(250);
```

其实不光Integer，String也有类似的性质，也就是说除非使用String a = new String("XXXXXX")这中写法强行开辟堆内存，不然的话Java中相同的String只会声明一次，之后都会使用常量池里面声明过的内存空间，其根源也是因为String底层是被final修饰的，所以复习课上磊磊学姐对String的描述是字符串常量

```
/** The value is used for character storage. */
private final char value[];
```

String b = "TheShy"; a==b的输出结果还是true的，跟.equals()的结果是相同的。

总结一下，对于Integer而言，对应数值在缓存区间之内的对象引用拿到的是在常量池里面的对象，强行使用new Integer()或者是对应的数值不在缓存区间之内的就会在堆内开辟内存存储对象

然后咱们再来扯扯有关方法区与堆的区别

方法区与堆

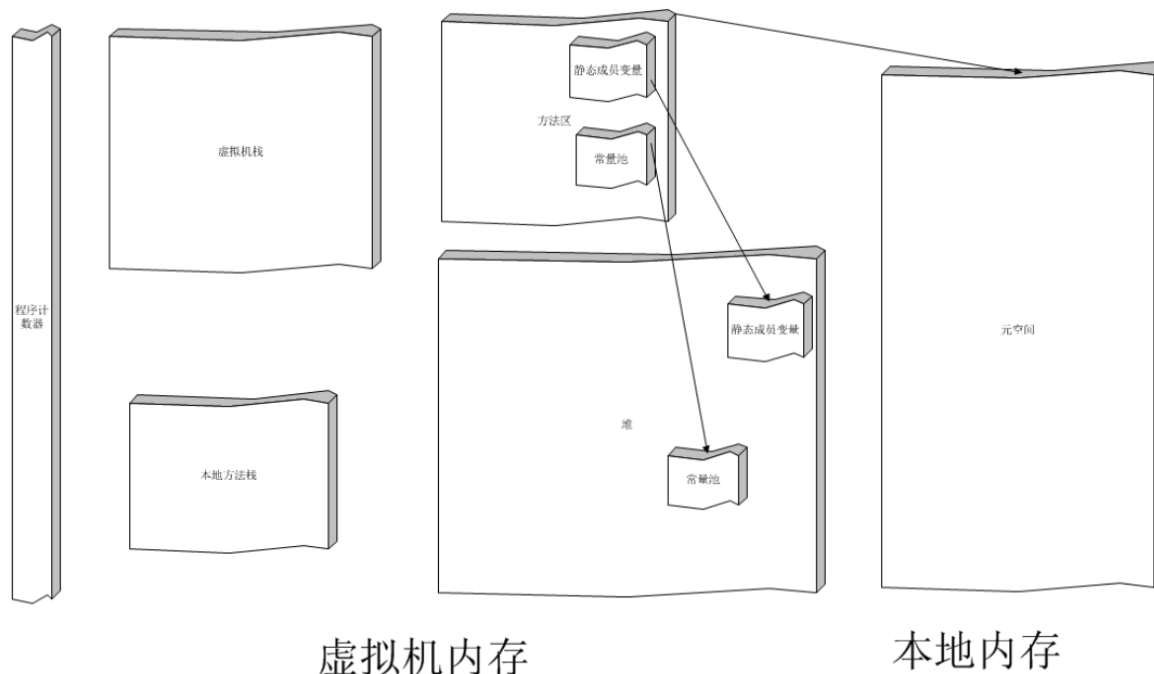
为啥我要浪费大家的时间去区分这两个概念？为啥不谈其他区之间的区别而只谈方法区和堆？当然是为了凑时长啊！——关于为啥要专挑方法区和堆进行区分，不知大家刚刚有没有产生一种疑惑，堆是用来存储对象的，那么final这样的常量属性归谁存储呢？它是对象的一部分，按理说应该存储在堆中，但是从方法区中又有一个叫常量池的东西，这个属性又好像应该存储在常量池中.....emmm，好像出了一些问

题。🤔🤔🤔🤔

其实这个问题并不复杂，常量准确的说应该用final修饰的成员变量，利用final修饰的局部变量应该称之为不可变量，final关键字的加成并没有让他一跃而成为常量池的高贵住户，它还是存储在栈之中的。而用final修饰的成员变量就完全符合进入常量池的条件了，自然可以入往常量池之中

可以理解为方法区是面向类的，而堆是面向对象的

说到这里应该就已经解决了疑惑了，但是我在查资料的时候却发现了一个诡异的图片：



这张图带给了我不少的冲击力，首先是方法区的常量池和静态成员变量居然存储的仅仅是引用，真正的数据居然存储在堆之中？然后就是方法区居然可以引用虚拟机内存之外的内存？

讲真，这个问题还是比较复杂的，关于常量池在JDK8之后的归属问题以及常量池到底要存储什么，网络上存在着一些不同的说法，我就找了一种我认同度比较高的解释给大家分享一下，我先把相关链接贴在这里，看不懂我写的东西的话可以去看看这些文章：

[JAVA常量池讲解](#) [静态成员变量的存储位置与JVM的内存划分](#) [元空间与永久代](#)（我只是一个莫的感情的链接搬运机器）

先吹吹概念，聊两句常量池

常量池

常量池分为静态常量池和运行时常量池，我们常说的常量池指的是运行时常量池，静态常量池存储的是字符串字面量，还包含类、方法的信息，占用class文件绝大部分空间。而运行时常量池存储的是经过ClassLoader加载过后的类的信息，存储的是编译期间产生的各种**字面量**和**符号引用**。

停停停，字面量和符号引用又是啥东西啊，解释清楚了再往下讲啊

- 字面量：Java语法层面的常量，加final修饰的变量以及String和Integer的部分区间这种Java语法层面的常量被称为是字面量
- 符号引用常量：JVM有符号引用和直接引用这两个概念，符号引用是编译原理方面的概念，我是这样理解的：符号引用存在于Class文件中，是编译期间产生的，这个时候还没有经过ClassLoader的加载，这些变量或者常量对应的内存存储空间还不确定，所以就像我们编写程序一样，Java用符号引用来表示这些变量和常量，而一旦加载进内存之中，得到的就是指向地址的直接引用了。当然了，符号引用还被用于存储类和接口的全限定名、字段名称和描述符、方法名称和描述符，后三者就是符号引用常量，关于这三个东西是什么，才疏学浅，讲不明白。

《java虚拟机规范》中写道：“确切来说，java虚拟机为每个类型都维护着一个常量池”。

然后就是JDK8之后存储规则的变化（如果理解有误，请各位及时指出）

JDK8前后JVM的变化

可以不负责任的说之前给出的有关JVM区域的划分是基于JDK8之前版本给出的。早在JDK7之中，JVM的方法区就已经开始分裂了（牢不可破的联盟）**符号引用存储在了native heap中**（你别看它长得像本地方法栈的英文名，其实这玩意儿在堆中），**字符串常量和静态类型变量存储在普通的堆区中**，如此而言，不论是new String还是直接赋值，字符串常量的存储位置都是在堆之中的，JDK8保留了JDK7关于方法区进行的这些分裂，并且引入了元空间的概念，原本的方法区依托于永久代，JDK8取消了永久代，改做方法区。为什么要这样做？永久代毕竟还是JVM内部的内存，面对一些会动态生成类的问题（好高端，之前完全没听说过）是很容易出现OOM的，简而言之，永久代是有极限的，只有超出JVM内存的限制才能修成正果，所以它不做JVM内存了，化而为计算机外部内存并以元空间这个崭新的名字降临到了JDK8之中。我们暂时不用关心元空间和方法区内存分配的问题，只需要记住常量池的转移就可以理解之前的疑惑了。

程序计数器

程序计数器存储的是当前正在执行的字节码指令的地址，也就是记住当前程序正在干什么。

当时查资料看到这个定义的时候我满脸？？？，这玩意有啥用啊，后来冷静了一下，感觉是不是用来记录诸如递归调用中的函数断点之类的信息的……我当时以为我找到了答案，但是转念一想，诸如分支选择这些结构应该是在字节码之中被控制的，而不是用计数器进行控制，而递归结构又完全可以通过JAVA栈FILO的特性进行替代，好像程序计数器真的没啥作用了，我又大开脑洞，是不是为了在程序半途终止的时候用来记录一下，到程序恢复的时候就知道上次搞到什么位置了？这样的解释看似很合理，也很贴合函数计数器的功能，但是这种所谓的程序被终止不能是如同关机和重新运行那种终止，而更像是暂停，Java啥时候要用到暂停呢？我想到了以下几个方面

- **多线程工作**：上学期梓漩学姐（如果我没有记错）讲过，单核CPU实现多线程的方法是在多项任务之间快速切换，假设这里要进行AB两个线程，CPU先去跑了线程A，然后需要立刻去执行线程B，这个时候就需要程序计数器去为CPU记下线程A跑到那里了，方便CPU跑完线程B之后回来执行线程A能找到上次运行到什么位置了。
- **Stop The World**：JVM GC中的一个概念，后面会提，这里会使用程序计数器也只是我的一个猜测。

接下来搞一个历史遗留问题，虽然说与JVM关系不大，但是感觉用JVM可以更加深入的了解这个问题

内部类与final

废话不多说，先上一段代码：


```
public class Main {  
  
    int b = 1; //外部类的属性  
  
    public void outerFunction() {  
  
        int aFromOuterClass = 1; //基本数据类型，存储在栈中  
        God godTao = new God(); //引用类型，存储在堆中（我在没有暗示涛神）  
        int [] ints = new int[1]; //数组类型，存储在堆中  
  
        class innerClass{
```




```

        public void innerFunction(){
            aFromOuterClass++;//错误，内部类使用外部类的局部基本数据类型应该加final
            b++;//正确，可以正常的访问和修改
            godTao.name = "Zhang_Tao";//正确，可以正常的访问和修改
            ints[0]++;//正确，可以正常的访问和修改
        }
    }
}

```




不知大家还记不记得我们可爱而又美丽的丁德桥学姐在红岩第三节课讲的有关内部类的东西，当时ds  提到过一句内部类对外部类局部变量的引用要加final关键字进行修饰，那么为啥呢，丁神表示由于篇幅限制没能成功说完

哦豁， 还是不明白，好的，那么你继续听我慢慢分析
首先我们定义一个接口，做准备准备工作。

不准备了，不准备了，自己百度看教程吧，再准备这课件起码到20000字了。

今天小弟我就书接上文，狗尾续貂一波。

首先，非静态内部类会自然而然的持有一个外部类的引用（之前学长在将有关Handler的内存溢出的时候提到过一次），在编译过后的内部类的构造方法中会传进来一个外部类的实例，

 God.class	2020-07-07 14:30	CLASS 文件	1 KB
 Main\$1innerClass.class	2020-07-09 15:19	CLASS 文件	1 KB
 Main.class	2020-07-09 15:19	CLASS 文件	1 KB

图中的第二个类就是编译后的局部内部类，点开它（将错误部分删除了之后的编译结果，请自动忽略背景图片）

```

class Main$1innerClass {
    Main$1innerClass(Main this$0, God var2, int[] var3) {
        this.this$0 = this$0;
        this.val$godTao = var2;
        this.val$ints = var3;
    }

    public void innerFunction() {
        ++this.this$0.b;
        this.val$godTao.name = "Zhang_Tao";
        int var10002 = this.val$ints[0]++;
    }
}

```

局部内部类调用的这些变量和引用肯定是存储在栈中的。而如果调用的是一个引用数据类型。那么内部类拿到的就是此引用数据类型的引用的复制（如果说的不这么拗口，就是如果调用的是对象，内部类就可以拿到这个对象的一个引用的复制），这两个引用指向的是同一块堆内存，内部类就可以对这个对象进行正常的修改，而如果传递进来的是基本数据类型，传递进去的就是一个这个基本数据类型的一个复制，类似于c语言的值传递，内部类对这个变量进行修改将无法引起外部类变量的更改，为了防止这种

事情发生，Java就强制要求外部变量不可变，这样就算是传递进去了，内部类也不能对他进行修改，就解决了不同步变化的问题。

如果调用的是外部类的成员变量，那就不一样了，属性变量就算是基本数据类型，也是存储在堆内存之中的，可以通过包含着它的对象进行传递，这样就可以通过引用的方式访问此变量，不存在内部改变而外部不改变的情况。数组也是如此。

ds有提到闭包的概念，这个我并不了解，大家如果有了解的可以教一下我

GC机制

谈完上面这些有的没的的，咱们还是来点更实用的知识点，聊一聊GC机制吧，这个地方预备分为两个部分去讲，一个部分是纯粹的JVM GC机制，另一个部分则是基于Android平台谈内存泄漏。

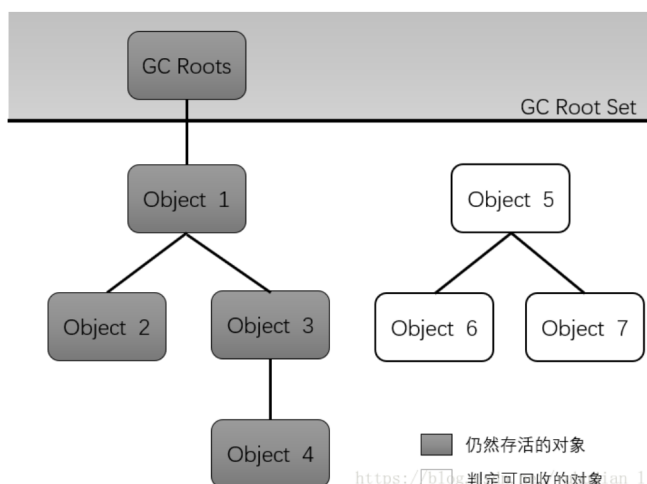
JVM 的GC机制

JVM堆内存的GC机制

堆内存中存储着几乎所有的对象，JVM对这些对象是否可以回收的判断依据是**对象是否死亡**，也就是是否还存在任何可能性使此对象被引用到，判断方法有以下几种

回收谁

- **引用计数算法**：每当一个对象被引用时，计数器数值就+1，每当一个引用失效时，计数器数值就减一，**任何时刻计数器数值为0的对象就是不可再被引用的**。怎么理解？最初声明一个对象，我们一定会拥有它的一个引用，而如果一块地址的引用数为0，完全可以认为这块地址是不可能再被引用到的。可以这样比喻，假如我就是个对象，我居住在深山里，我有很多住山外面的朋友，他们知道我的住址，我的朋友就相当于我的引用，当我在这个世界上的所有朋友都消失的时候，山外面不再有人知道我，更不会知道我住哪儿，那我对于山外的人就是没有意义的，我就可以被GC掉。那么这样的GC机制存在怎样的问题呢？如果两个对象互相持有对方的引用，并且与外界不再有任何联系，他们始终是有引用的，故无法被GC，但是对于我们而言，这样的两个对象是没有意义的，应该被GC，这就体现了引用计数算法的一个缺陷
- **可达性分析算法**：这个算法的基本思路就是通过一系列名为GC Roots的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，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，下图对象object5, object6, object7虽然有互相判断，但它们到GC Roots是不可达的，所以它们将会判定为是可回收对象。



https://blog.csdn.net/qq_41592261/article/details/104444444

一般来说，如下情况的对象可以作为GC Roots：

虚拟机栈（栈帧中的本地变量表）中的引用的对象

方法区中的类静态属性引用的对象

方法区中的常量引用的对象

本地方法栈中JNI（Native方法）的引用的对象

那么可达性算法分析又存在那些问题呢？

1、**需要消耗大量的时间**。GC Roots肯定要是那种全局性的结点，或者换句话说，GC Roots选取之后应该是具备全域检索能力的，打个不合理的比喻，如果GC Roots是涛哥，那么涛哥及涛哥庞大的大佬圈子都不可能GC掉，而如果GC Roots是我，我和我的垃圾圈子一共就那三四个人，让我们不被GC掉显然是没有意义的。而排查全局性的GC Roots的所有可达结点需要消耗大量的时间。

2、**GC停顿**。在进行可达性算法分析的时候需要保证对象之间的引用关系不发生改变，导致进行可达性分析算法的时候需要暂停所有的线程（称为"Stop The World"）



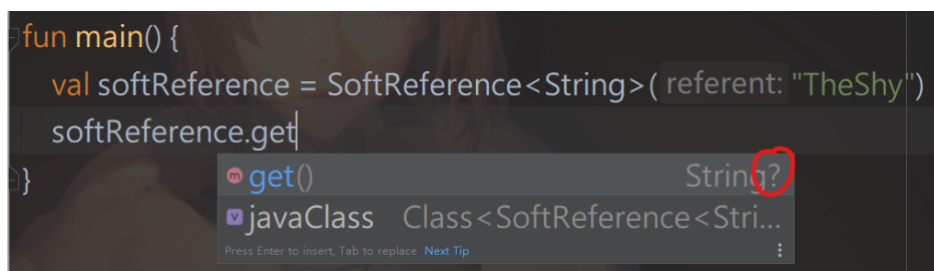
- **引用类型**：JDK1.2之后对引用的概念进行了扩充，将引用分为强引用，软引用，弱引用，虚引用几种类型，其引用强度依次减弱

1. 强引用（StrongReference）：Object a = new Object(); 这种引用，只要强引用存在，就不会回收被引用的对象
2. 软引用（SoftReference）：被软引用的对象将在内存不足的时候被回收，可以通过JDK1.2之后提供的SoftReference类进行实现
3. 弱引用（WeakReference）：被弱引用的对象只能存活到下一次GC来临之前，当GC的时候，不论当前内存是否已满，都会将被弱引用的对象GC掉
4. 虚引用（PhantomReference）：有没有虚引用不影响此对象是否会被GC，通过虚引用也不能得到一个对象的实例，设置虚引用的唯一目的就是在对象被GC掉之后系统会发送一条通知

具体怎么使用后三种引用，这里举个例子：

```
SoftReference<String>stringSoftReference = new SoftReference<>("TheShy");
System.out.println(stringSoftReference.get());
```

stringSoftReference就相当于之前的变量名，对它所指向的对象进行调用只需要使用get方法就可以（由于有可能会被GC掉，所以需要进行判空）



Kotlin中可以很清晰的看到这里的返回值是可能为空的

啥时候回收

- CPU空闲的时候（给自己找点事情做）
- 堆满了的时候（后面会具体讲YGC 与 FGC的触发条件）
- 手动调用System.gc();（哲哥上节课多次操作过了）

怎么回收（垃圾收集算法）

JVM有以下回收策略，我先copy一些网上的简述

- 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。
- 分代收集算法：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。

接下来仔细的看着

- **标记清除算法**：顾名思义，标记清除算法分为两个部分，标记和清除，首先标记出所有需要进行回收的对象，在标记完成后统一进行回收，标记算法就是之前有提到过的可达性分析算法，以及引用计数算法。在判断过对象是否要进行回收之后，调用清除算法进行清除。这是一种相当基础的工作机制，就是简单的遍历查找和清除，它存在两个不足之处，1、标记和清除算法的效率都不高 2、清除之后容易产生大量的内存碎片，引起清除后空间碎片化的原因是被清除的对象的分布极有可能是很分散的，这样引起的后果是如果在清楚之后需要对大对象进行存储，很有可能找不到一块足够大的连续的内存块，因为可用的内存都是分散的，这样就会再引起一次GC，而如之前所说，GC是有代价的，反复发生GC对程序的运行效果影响还是很大的
- **复制算法**：这种算法解决了效率问题和空间分配问题，具体的操作方法是**将内存划分为两部分，先使用其中的一部分**，发生GC的时候先将不能被GC的对象移动到另一块内存进行避难，**之后一次性对之前的内存进行GC清理**，在提升了效率的同时也解决了内存碎片化的问题。打一个比较怪异的比喻，就相当于是在小镇A上出了一堆恐怖分子，但是小镇上也有平民，标记清除算法就相当于是在先挨家挨户的摸查谁是恐怖分子谁是平民，然后对恐怖分子进行挨家挨户的枪决，复制算法就相当于是在先将小镇A上的平民转移到小镇B，然后直接对小镇A进行轰炸，以达到消灭恐怖分子的目的，这样自然要快很多。由于堆内存之中存储使用的数据结构是堆（我在说什么大实话）所以移动只需要放到另一块内存的堆顶就ok，转移是工作还是比较轻松的。然而复制算法的劣势在于由于一次只能使用部分的内存空间，必然会导致内存空间的浪费，这里可以通过调整存储幸存者的区域和存储new的崭新变量的内存空间的占比来降低浪费掉的内存的占比。然后就是如果幸存的对象过多，对这些对象进行转移将花费较多的时间
- **标记整理算法**：这种算法理解起来很简单，记得刚开始提到的标记清理算法吗，标记整理算法就是在标记之后不先急着进行清理，而是先将要保留的对象移向内存的同一端，然后再进行统一清理
- **分代收集算法**：分代收集，自然要先进行分代，堆内存被分为**年轻代和老年代**，年轻代又被分为**Eden区（伊甸）**，**From区与To区**，新建对象时首先会存储在Eden区，当Eden区满了之后就会发生Young GC，首先将GC后幸存的对象转移到From区之中，之后对Eden区进行一次全方位GC，在这之后建立的新的对象还是会被存储在Eden区之中，当Eden区再次满掉之后**会将Eden区与From区还在被使用的对象复制到To区**，然后再进行一次同样的GC，JVM会保证From区和To区至少有一个是空闲的，即发生youngGC的时候会将对象自Eden区以及From/To区转移到To/From区，经过若干次转移之后（默认为15次），JVM认为那些在From和To区之间游荡的对象应该转移次数过多了，就一并打入老年代，当老年带满掉之后就会进行大扫除（Full GC/Major GC），就是前文简述中提到的**新生代基本采用复制算法，老年代采用标记整理算法**。

Tip:

- 大对象直接进入老年代：可以通过-XX:PretenureSizeThresholdsh控制JVM的参数，超过这个参数限制的大对象直接进入老年代
- JVM存在空间分配担保策略，在发生YongGC之前，JVM会检查老年代的最大可用连续空间是否大于新生代的总空间，如果是这样那么YongGC就是安全的，如果小于，JVM将会查看HandlePromotionFailure值来判断是否允许担保失败，如果允许，将继续检查老年代最大可用的连续空间是否大于历次进入老年代的对象的平均大小，如果大于，将尝试着进行一次

YongGC，如果小于，则进行FullGC。

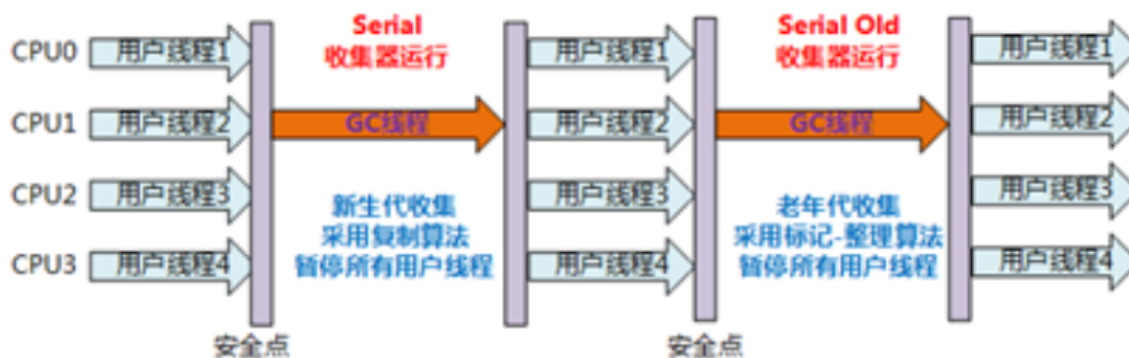
稍微总结一下，就是JVM需要进行YongGC了，先预判一手，如果这波稳，就直接GC，如果不稳，就判断一下能不能担保GC引起的风险，如果能，就比对剩余的连续空间与进入老年代的对象的平均大小，如果大于就试着GC以下，如果小于就直接FullGC

垃圾收集器

上面全是理论上的算法，那么真正去执行这些算法的工作者又是谁呢？那就是垃圾收集器了，接下来让我们了解一下一些常见的垃圾收集器的特性和工作方法，我先将一篇博客链接贴在这里，大家看不懂我写的东西或者想要了解更多的可以点击[这里](#)，我精力能力实在有限，没有办法将所有的知识点全部吸收并转化给大家，这部分讲的比较粗略，希望大家谅解。

Serial收集器（串行收集器）

特点：针对新生代，对新生代采用复制算法，是一个单线程从收集器，只会沿着一条线路进行垃圾收集，并且在垃圾收集的时候要Stop The World



上图是它的工作流程图，这里的**安全点**是指线程工作状态确定的点，也就是引用关系暂不发生变化的点，只有到达安全点才能暂停，具体的查找和实现方法我才疏学浅不好说明，大家可以课后自己去查查看看（是一个挺重要的概念），大家注意后面的是Serial Old收集器，两者并不相同，后面会提到Serial Old收集器

安全点常取的位置

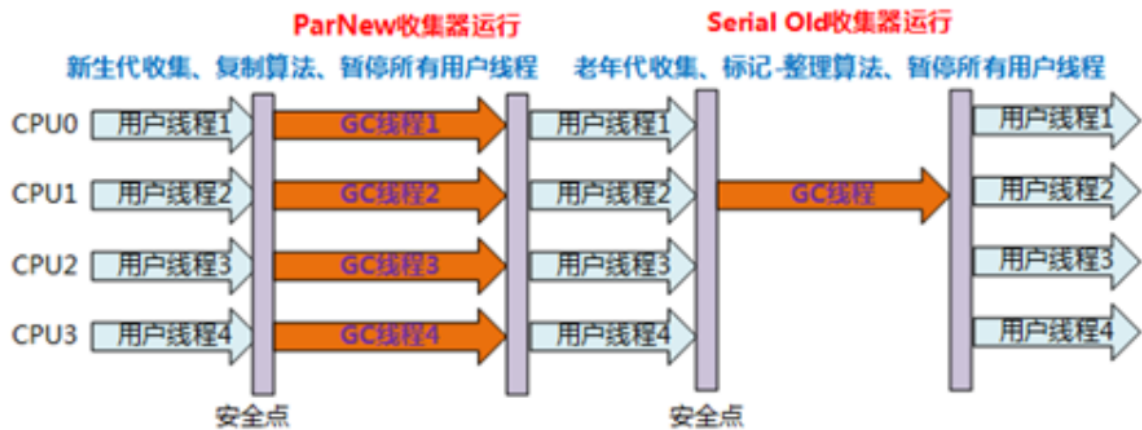
- 1、循环的末尾
- 2、方法临返回前
- 3、调用方法之后
- 4、抛异常的位置

这种收集器适用于单核的CPU，可以将单个线程效率发挥到最好，同时也是Client模式下默认的新生代收集器。（我在课件的最开始介绍过Client模式和Server模式）

ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和Serial收集器完全一样。也可以认为是针对于新生代

虚拟机在处于Server模式下会首选这种收集器来执行新生代的GC，下图是ParNew/SerialOld收集器



ParNew是唯一一种可以和CMS收集器兼容公用的收集器（后者针对老年代，具有非常好的特性，后面会提到），故在Server模式下具有举足轻重的作用，其缺点在于如果被使用于单核CPU，线程之间的调度将降低其运行效率，这时候不如使用串行收集器。

Parallel Scavenge收集器

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器。

Parallel Scavenge收集器关注点是**吞吐量**（如何高效率的利用CPU）。所谓吞吐量就是CPU中用于运行用户代码的时间与CPU总消耗时间的比值。（吞吐量：CPU用于用户代码的时间/CPU总消耗时间的比值，即=运行用户代码的时间/(运行用户代码时间+垃圾收集时间)。比如，虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。）这样就提升了CPU的利用率，让我们的时间更多的花在运行代码上而不是等待GC上

Serial Old收集器

Serial收集器的老年代版本，单线程，利用标志整理算法对老年代进行GC。

Parallel Old收集器

Parallel Scavenge收集器的老年代版本，用于缓解Parallel Scavenge收集器尴尬的收集器，因为在它出现之前如果新生代使用了Parallel Scavenge收集器，老年代只能使用Serial Old收集器，后者在Server模式下效果一般，而ParNew收集器确可以搭配高贵的CMS收集器，那要Parallel Scavenge收集器有何用？而Parallel Old收集器就缓解了这种尴尬，为Parallel Scavenge收集器提供了一个多线程的老年代收集方法，让Parallel Scavenge收集器有了用武之地。

CMS收集器

达到了最短停顿时间的目的，针对于老年代的收集，只能于ParNew收集器兼容，GC线程和工作线程并发执行

工作流程：

- 初始标记(CMS initial mark)
- 并发标记(CMS concurrent mark)
- 重新标记 (CMS remark)
- 并发清除 (CMS concurrent sweep)

其在进行初始标记和重新标记的时候需要Stop The World，其余时候可以于工作线程并发进行

总结

在都介绍了一遍之后，我们来稍微总结一下

- CMS收集器和Parallel Scavenge收集器的对比：CMS专注于尽可能减少因GC而产生的暂停事件，PS收集器则专注于让GC时间相对于程序运行时间而言占比更短
- 收集器的选择方案：如果是Client模式，新生代和老年代最好均选用串行的收集器，以充分发挥单核性能，如果是Server模式，可以选择PN+CMS或者PS+PO这两种组合

G1收集器

JDK7中开始完全支持的性能超高的收集器，对飙CMS收集器，大家可以了解一下

Android的内存泄漏

其实这个罗杰学长讲过，昨天哲哥他们也讲过，但是由于我理解的不是很到位，所以想借着写课件的机会补充一下自己的知识，在这里我就只稍微的总结一下，大家如果已经很了解了就可以玩玩手机或者赶一赶自己的考核项目

为什么会发生内存泄漏

Android为每个应用分配的内存空间是有限的，而如果因为一些原因导致JVM无法对堆中的内存进行回收，或者对象占用的内存过大，就有可能引起内存泄漏

常见的导致Android出现内存泄漏的写法与解决方案

分析了一下，感觉从宏观上可以分为两个原因：1 声明周期短的对象持有了生命周期长的对象的引用，导致JVM无法进行回收 2 本身定义的一些集合类越聚越多，没有及时清理

先来看看可能1

- Handler使用不当：作为内部类的Handler持有外部Activity的引用，显然Activity的生命周期长于Handler的预计生命周期，容易导致JVM无法对Handler进行回收。 解决方案：将Handler改为静态内部类，并使用弱引用来获得Activity的实例
- 持有Context变量：如果Activity的Context被生命周期更长的对象所持有，将会导致Activity无法被回收。解决方案：如果生命周期太长的话可以去持有Application的引用

再来看看可能2

- 集合对象没有在合适的时候进行清理：顾名思义，集合装的太多，内存受不住了。解决方案：集合用过之后及时的clear然后赋值为null。
- 资源对象未释放：数据连接、IO、Socket连接等等，它们必须显示释放（用代码 close 掉），否则不会被 GC 回收。

罗杰学长（如果我没有记错）之前上课的时候有讲过内存泄漏，我把他的课件贴在下面，点击就可下载

[罗杰学长的课件](#)

最后

可能存在很多错误，也可能说了许多没有意义的东西，希望各位多多包涵