Java 底层学习

前言

不论学什么语言，都要从底层原理上理解！

Java 内置工具

（jps、jstack、jmap、jstat）

javap 命令分析 java 汇编指令

javap 是 jdk 自带的反解析工具。它的作用就是根据 class 字节码文件，反解析出当前类对应的 **code 区（汇编指令）**、本地变量表、异常表和代码行偏移量映射表、常量池等等信息。

当然这些信息中，有些信息（如本地变量表、指令和代码行偏移量映射表、常量池中方法的参数名称等等）需要在使用 javac 编译成 class 文件时，指定参数才能输出，比如，你直接 javac xx.java，就不会在生成对应的局部变量表等信息，如果你使用 **javac -g** xx.java 就可以生成所有相关信息了。

通过反编译生成的汇编代码，我们可以深入的了解 java 代码的工作机制
javap 的用法格式：
javap <options> <classes>

```
-help  --help  -?        输出此用法消息
-version                 版本信息，其实是当前javap所在jdk的版本信息，不是class在哪个jdk下生成的。
-v  -verbose             输出附加信息（包括行号、本地变量表，反汇编等详细信息）
-l                         输出行号和本地变量表
-public                    仅显示公共类和成员
-protected               显示受保护的/公共类和成员
-package                 显示程序包/受保护的/公共类 和成员 (默认)
-p  -private             显示所有类和成员
-c                       对代码进行反汇编
-s                       输出内部类型签名
-sysinfo                 显示正在处理的类的系统信息 (路径, 大小, 日期, MD5 散列)
-constants               显示静态最终常量
-classpath <path>        指定查找用户类文件的位置
-bootclasspath <path>    覆盖引导类文件的位置
```

javap -v classxx，不仅会输出行号、本地变量表信息、反编译汇编代码，还会输出当前类用到的常量池等信息。
javap -l 会输出行号和本地变量表信息。
javap -c 会对当前 class 字节码进行反编译生成汇编代码。

案例：

```
public class TestDate {

    private int count = 0;

    public static void main(String[] args) {
        TestDate testDate = new TestDate();
        testDate.test1();
    }

    public void test1(){
        Date date = new Date();
        String name1 = "wangerbei";
        test2(date,name1);
        System.out.println(date+name1);
    }

    public void test2(Date dateP,String name2){
        dateP = null;
        name2 = "zhangsan";
    }

    public void test3(){
        count++;
    }

    public void  test4(){
        int a = 0;
        {
            int b = 0;
            b = a+1;
        }
        int c = a+1;
    }
}
```

```
javap -c -l TestDate
```

```
Warning: Binary file TestDate contains
com.justest.test.TestDate
Compiled from "TestDate.java"
public class com.justest.test.TestDate {
  //默认的构造方法，在构造方法执行时主要完成一些初始化操作，包括一些成员变量的初始化赋值等操作
  public com.justest.test.TestDate();
    Code:
       0: aload_0 //从本地变量表中加载索引为 0 的变量的值，也即 this 的引用，压入栈
       1: invokespecial #10  //出栈，调用 java/lang/Object."<init>":()V 初始化对象，就是 this 指定的对象的 init()方法完成初始化
       4: aload_0  // 4 到 6 表示，调用 this.count = 0，也即为 count 复制为 0。这里 this 引用入栈
       5: iconst_0 //将常量 0，压入到操作数栈
       6: putfield      //出栈前面压入的两个值（this 引用，常量值 0），  将 0 取出，并赋值给 count
       9: return
//指令与代码行数的偏移对应关系，每一行第一个数字对应代码行数，第二个数字对应前面 code 中指令前面的数字
    LineNumberTable:
      line 5: 0
```

```
      line 7: 4
      line 5: 9
    //局部变量表，start+length 表示这个变量在字节码中的生命周期起始和结束的偏移位置（this 生命周期从头 0 到结尾 10），slot 就是这个变量在局部变量表中的槽位（槽位可复用），name 就是变量名称，Signatur 局部变量类型描述
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      10     0  this   Lcom/justest/test/TestDate;

  public static void main(java.lang.String[]);
    Code:
// new 指令，创建一个 class com/justest/test/TestDate 对象，new 指令并不能完全创建一个对象，对象只有在初，只有在调用初始化方法完成后（也就是调用了 invokespecial 指令之后），对象才创建成功，
       0: new  //创建对象，并将对象引用压入栈
       3: dup //将操作数栈定的数据复制一份，并压入栈，此时栈中有两个引用值
       4: invokespecial #20  //pop 出栈引用值,调用其构造函数，完成对象的初始化
       7: astore_1 //pop 出栈引用值，将其（引用）赋值给局部变量表中的变量 testDate
       8: aload_1  //将 testDate 的引用值压入栈，因为 testDate.test1();调用了 testDate，这里使用 aload_1 从局部变量表中获得对应的变量 testDate 的值并压入操作数栈
       9: invokevirtual #21 // Method test1:()V  引用出栈，调用 testDate 的 test1()方法
      12: return //整个 main 方法结束返回
    LineNumberTable:
      line 10: 0
      line 11: 8
      line 12: 12
    //局部变量表，testDate 只有在创建完成并赋值后，才开始声明周期
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      13     0  args   [Ljava/lang/String;
          8       5     1 testDate   Lcom/justest/test/TestDate;

  public void test1();
    Code:
       0: new           #27                 // 0 到 7 创建 Date 对象，并赋值给 date 变量
       3: dup
       4: invokespecial #29                 // Method java/util/Date."<init>":()V
       7: astore_1
       8: ldc           #30     // String wangerbei，将常量“wangerbei”压入栈
      10: astore_2  //将栈中的“wangerbei”pop 出，赋值给 name1
      11: aload_0 //11 到 14，对应 test2(date,name1);默认前面加 this.
      12: aload_1 //从局部变量表中取出 date 变量
      13: aload_2 //取出 name1 变量
      14: invokevirtual #32                 // Method test2:(Ljava/util/Date;Ljava/lang/String;)V  调用 test2 方法
  // 17 到 38 对应 System.out.println(date+name1);
      17: getstatic     #36                 // Field java/lang/System.out:Ljava/io/PrintStream;
  //20 到 35 是 jvm 中的优化手段，多个字符串变量相加，不会两两创建一个字符串对象，而使用 StringBuilder 来创建一个对象
      20: new           #42                 // class java/lang/StringBuilder
      23: dup
```

```
      24: invokespecial #44                 // Method
java/lang/StringBuilder."<init>":()V
      27: aload_1
      28: invokevirtual #45                 // Method
java/lang/StringBuilder.append:(Ljava/lang/Object;)Ljava/lang/
StringBuilder;
      31: aload_2
      32: invokevirtual #49                 // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/
StringBuilder;
      35: invokevirtual #52                 // Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
      38: invokevirtual #56                 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
invokevirtual 指令表示基于类调用方法
      41: return
    LineNumberTable:
      line 15: 0
      line 16: 8
      line 17: 11
      line 18: 17
      line 19: 41
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      42     0  this   Lcom/justest/test/TestDate;
          8      34     1  date   Ljava/util/Date;
         11      31     2 name1   Ljava/lang/String;

  public void test2(java.util.Date, java.lang.String);
    Code:
       0: aconst_null //将一个 null 值压入栈
       1: astore_1 //将 null 赋值给 dateP
       2: ldc           #66      // String zhangsan 从常量池中取
出字符串“zhangsan”压入栈中
       4: astore_2 //将字符串赋值给 name2
       5: return
    LineNumberTable:
      line 22: 0
      line 23: 2
      line 24: 5
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       6     0  this   Lcom/justest/test/TestDate;
          0       6     1 dateP   Ljava/util/Date;
          0       6     2 name2   Ljava/lang/String;

  public void test3();
    Code:
       0: aload_0 //取出 this，压入栈
       1: dup   //复制操作数栈栈顶的值，并压入栈，此时有两个 this 对
象引用值在操作数组栈
       2: getfield #12// Field count:I this 出栈，并获取其 count
字段，然后压入栈，此时栈中有一个 this 和一个 count 的值
       5: iconst_1 //取出一个 int 常量 1,压入操作数栈
       6: iadd  // 从栈中取出 count 和 1，将 count 值和 1 相加，结果入
栈
       7: putfield      #12 // Field count:I  一次弹出两个，第一个
弹出的是上一步计算值，第二个弹出的 this，将值赋值给 this 的 count 字段
      10: return
    LineNumberTable:
      line 27: 0
      line 28: 10
    LocalVariableTable:
```

```
      Start  Length  Slot  Name   Signature
          0      11     0  this   Lcom/justest/test/TestDate;
  public void test4();
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
       3: istore_2
       4: iload_1
       5: iconst_1
       6: iadd
       7: istore_2
       8: iload_1
       9: iconst_1
      10: iadd
      11: istore_2
      12: return
    LineNumberTable:
      line 33: 0
      line 35: 2
      line 36: 4
      line 38: 8
      line 39: 12
    //看下面，b和c的槽位slot一样，这是因为b的作用域就在方法块中，方法块结束，局部变量表中的槽位就被释放，后面的变量就可以复用这个槽位
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      13     0  this   Lcom/justest/test/TestDate;
          2      11     1     a   I
          4       4     2     b   I
         12       1     2     c   I
}
```

Java 指令集

https://blog.csdn.net/github_35983163/article/details/52945845

常量入栈指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x01	aconst_null		null值入栈。
0x02	iconst_m1		-1(int)值入栈。
0x03	iconst_0		0(int)值入栈。
0x04	iconst_1		1(int)值入栈。
0x05	iconst_2		2(int)值入栈。
0x06	iconst_3		3(int)值入栈。
0x07	iconst_4		4(int)值入栈。
0x08	iconst_5		5(int)值入栈。
0x09	lconst_0		0(long)值入栈。
0x0a	lconst_1		1(long)值入栈。
0x0b	fconst_0		0(float)值入栈。
0x0c	fconst_1		1(float)值入栈。
0x0d	fconst_2		2(float)值入栈。
0x0e	dconst_0		0(double)值入栈。
0x0f	dconst_1		1(double)值入栈。
0x10	bipush	valuebyte	valuebyte值带符号扩展成int值入栈。
0x11	sipush	valuebyte1 valuebyte2	(valuebyte1 << 8) | valuebyte2 值带符号扩展成int值入栈。
0x12	ldc	indexbyte1	常量池中的常量值（int, float, string reference, object reference）入栈。
0x13	ldc_w	indexbyte1 indexbyte2	常量池中常量（int, float, string reference, object reference）入栈。
0x14	ldc2_w	indexbyte1 indexbyte2	常量池中常量（long, double）入栈。

局部变量值转载到栈中指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x19	(wide)aload	indexbyte	从局部变量indexbyte中装载引用类型值入栈。
0x2a	aload_0		从局部变量0中装载引用类型值入栈。
0x2b	aload_1		从局部变量1中装载引用类型值入栈。
0x2c	aload_2		从局部变量2中装载引用类型值入栈。
0x2d	aload_3		从局部变量3中装载引用类型值入栈。
0x15	(wide)iload	indexbyte	从局部变量indexbyte中装载int类型值入栈。
0x1a	iload_0		从局部变量0中装载int类型值入栈。
0x1b	iload_1		从局部变量1中装载int类型值入栈。
0x1c	iload_2		从局部变量2中装载int类型值入栈。
0x1d	iload_3		从局部变量3中装载int类型值入栈。
0x16	(wide)lload	indexbyte	从局部变量indexbyte中装载long类型值入栈。
0x1e	lload_0		从局部变量0中装载int类型值入栈。
0x1f	lload_1		从局部变量1中装载int类型值入栈。
0x20	lload_2		从局部变量2中装载int类型值入栈。
0x21	lload_3		从局部变量3中装载int类型值入栈。
0x17	(wide)fload	indexbyte	从局部变量indexbyte中装载float类型值入栈。
0x22	fload_0		从局部变量0中装载float类型值入栈。
0x23	fload_1		从局部变量1中装载float类型值入栈。
0x24	fload_2		从局部变量2中装载float类型值入栈。
0x25	fload_3		从局部变量3中装载float类型值入栈。
0x18	(wide)dload	indexbyte	从局部变量indexbyte中装载double类型值入栈。
0x26	dload_0		从局部变量0中装载double类型值入栈。
0x27	dload_1		从局部变量1中装载double类型值入栈。
0x28	dload_2		从局部变量2中装载double类型值入栈。
0x29	dload_3		从局部变量3中装载double类型值入栈。
0x32	aaload		从引用类型数组中装载指定项的值。
0x2e	iaload		从int类型数组中装载指定项的值。
0x2f	laload		从long类型数组中装载指定项的值。
0x30	faload		从float类型数组中装载指定项的值。
0x31	daload		从double类型数组中装载指定项的值。
0x33	baload		从boolean类型数组或byte类型数组中装载指定项的值（先转换为int类型值，后压栈）。
0x34	caload		从char类型数组中装载指定项的值（先转换为int类型值，后压栈）。
0x35	saload		从short类型数组中装载指定项的值（先转换为int类型值，后压栈）。

将栈顶值保存到局部变量中指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x3a	(wide)astore	indexbyte	将栈顶引用类型值保存到局部变量indexbyte中。
0x4b	astroe_0		将栈顶引用类型值保存到局部变量0中。
0x4c	astore_1		将栈顶引用类型值保存到局部变量1中。
0x4d	astore_2		将栈顶引用类型值保存到局部变量2中。
0x4e	astore_3		将栈顶引用类型值保存到局部变量3中。
0x36	(wide)istore	indexbyte	将栈顶int类型值保存到局部变量indexbyte中。
0x3b	istore_0		将栈顶int类型值保存到局部变量0中。
0x3c	istore_1		将栈顶int类型值保存到局部变量1中。
0x3d	istore_2		将栈顶int类型值保存到局部变量2中。
0x3e	istore_3		将栈顶int类型值保存到局部变量3中。
0x37	(wide)lstore	indexbyte	将栈顶long类型值保存到局部变量indexbyte中。
0x3f	lstore_0		将栈顶long类型值保存到局部变量0中。
0x40	lstore_1		将栈顶long类型值保存到局部变量1中。
0x41	lstore_2		将栈顶long类型值保存到局部变量2中。
0x42	lstroe_3		将栈顶long类型值保存到局部变量3中。
0x38	(wide)fstore	indexbyte	将栈顶float类型值保存到局部变量indexbyte中。
0x43	fstore_0		将栈顶float类型值保存到局部变量0中。
0x44	fstore_1		将栈顶float类型值保存到局部变量1中。
0x45	fstore_2		将栈顶float类型值保存到局部变量2中。
0x46	fstore_3		将栈顶float类型值保存到局部变量3中。
0x39	(wide)dstore	indexbyte	将栈顶double类型值保存到局部变量indexbyte中。
0x47	dstore_0		将栈顶double类型值保存到局部变量0中。
0x48	dstore_1		将栈顶double类型值保存到局部变量1中。
0x49	dstore_2		将栈顶double类型值保存到局部变量2中。
0x4a	dstore_3		将栈顶double类型值保存到局部变量3中。
0x53	aastore		将栈顶引用类型值保存到指定引用类型数组的指定项。
0x4f	iastore		将栈顶int类型值保存到指定int类型数组的指定项。
0x50	lastore		将栈顶long类型值保存到指定long类型数组的指定项。
0x51	fastore		将栈顶float类型值保存到指定float类型数组的指定项。
0x52	dastore		将栈顶double类型值保存到指定double类型数组的指定项。
0x54	bastroe		将栈顶boolean类型值或byte类型值保存到指定boolean类型数组或byte类型数组的指定项。
0x55	castore		将栈顶char类型值保存到指定char类型数组的指定项。
0x56	sastore		将栈顶short类型值保存到指定short类型数组的指定项。

wide指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0xc4	wide		使用附加字节扩展局部变量索引（iinc指令特殊）。

通用（无类型）栈操作指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x00	nop		空操作。
0x57	pop		从栈顶弹出一个字长的数据。
0x58	pop2		从栈顶弹出两个字长的数据。
0x59	dup		复制栈顶一个字长的数据，将复制后的数据压栈。
0x5a	dup_x1		复制栈顶一个字长的数据，弹出栈顶两个字长数据，先将复制后的数据压栈，再将弹出的两个字长数据压栈。
0x5b	dup_x2		复制栈顶一个字长的数据，弹出栈顶三个字长的数据，将复制后的数据压栈，再将弹出的三个字长的数据压栈。
0x5c	dup2		复制栈顶两个字长的数据，将复制后的两个字长的数据压栈。
0x5d	dup2_x1		复制栈顶两个字长的数据，弹出栈顶三个字长的数据，将复制后的两个字长的数据压栈，再将弹出的三个字长的数据压栈。
0x5e	dup2_x2		复制栈顶两个字长的数据，弹出栈顶四个字长的数据，将复制后的两个字长的数据压栈，再将弹出的四个字长的数据压栈。
0x5f	swap		交换栈顶两个字长的数据的位置。Java指令中没有提供以两个字长为单位的交换指令。

类型转换指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x86	i2f		将栈顶int类型值转换为float类型值。
0x85	i2l		将栈顶int类型值转换为long类型值。
0x87	i2d		将栈顶int类型值转换为double类型值。
0x8b	f2i		将栈顶float类型值转换为int类型值。
0x8c	f2l		将栈顶float类型值转换为long类型值。
0x8d	f2d		将栈顶float类型值转换为double类型值。
0x88	l2i		将栈顶long类型值转换为int类型值。
0x89	l2f		将栈顶long类型值转换为float类型值。
0x8a	l2d		将栈顶long类型值转换double类型值。
0x8e	d2i		将栈顶double类型值转换为int类型值。
0x90	d2f		将栈顶double类型值转换为float类型值。
0x8f	d2l		将栈顶double类型值转换为long类型值。
0x91	i2b		将栈顶int类型值截断成byte类型，后带符号扩展成int类型值入栈。
0x92	i2c		将栈顶int类型值截断成char类型值，后带符号扩展成int类型值入栈。
0x93	i2s		将栈顶int类型值截断成short类型值，后带符号扩展成int类型值入栈。

整数运算			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x60	iadd		将栈顶两int类型数相加，结果入栈。
0x64	isub		将栈顶两int类型数相减，结果入栈。
0x68	imul		将栈顶两int类型数相乘，结果入栈。
0x6c	idiv		将栈顶两int类型数相除，结果入栈。
0x70	irem		将栈顶两int类型数取模，结果入栈。
0x74	ineg		将栈顶int类型值取负，结果入栈。
0x61	ladd		将栈顶两long类型数相加，结果入栈。
0x65	lsub		将栈顶两long类型数相减，结果入栈。
0x69	lmul		将栈顶两long类型数相乘，结果入栈。
0x6d	ldiv		将栈顶两long类型数相除，结果入栈。
0x71	lrem		将栈顶两long类型数取模，结果入栈。
0x75	lneg		将栈顶long类型值取负，结果入栈。
0x84	(wide)iinc	indexbyte constbyte	将整数值constbyte加到indexbyte指定的int类型的局部变量中。

浮点运算			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x62	fadd		将栈顶两float类型数相加，结果入栈。
0x66	fsub		将栈顶两float类型数相减，结果入栈。
0x6a	fmul		将栈顶两float类型数相乘，结果入栈。
0x6e	fdiv		将栈顶两float类型数相除，结果入栈。
0x72	frem		将栈顶两float类型数取模，结果入栈。
0x76	fneg		将栈顶float类型值取反，结果入栈。
0x63	dadd		将栈顶两double类型数相加，结果入栈。
0x67	dsub		将栈顶两double类型数相减，结果入栈。
0x6b	dmul		将栈顶两double类型数相乘，结果入栈。
0x6f	ddiv		将栈顶两double类型数相除，结果入栈。
0x73	drem		将栈顶两double类型数取模，结果入栈。
0x77	dneg		将栈顶double类型值取负，结果入栈。

逻辑运算—移位运算			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x78	ishl		左移int类型值。
0x79	lshl		左移long类型值。
0x7a	ishr		算术右移int类型值。
0x7b	lshr		算术右移long类型值。
0x7c	iushr		逻辑右移int类型值。
0x7d	lushr		逻辑右移long类型值。

逻辑运算—按位布尔运算			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x73	iand		对int类型按位与运算。
0x7f	land		对long类型的按位与运算。
0x80	ior		对int类型的按位或运算。
0x81	lor		对long类型的按位或运算。
0x82	ixor		对int类型的按位异或运算。
0x83	lxor		对long类型的按位异或运算。

控制流指令—条件跳转指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x99	ifeq	branchbyte1 branchbyte2	若栈顶int类型值为0则跳转。
0x9a	ifne	branchbyte1 branchbyte2	若栈顶int类型值不为0则跳转。
0x9b	iflt	branchbyte1 branchbyte2	若栈顶int类型值小于0则跳转。
0x9e	ifle	branchbyte1 branchbyte2	若栈顶int类型值小于等于0则跳转。
0x9d	ifgt	branchbyte1 branchbyte2	若栈顶int类型值大于0则跳转。
0x9c	ifge	branchbyte1 branchbyte2	若栈顶int类型值大于等于0则跳转。
0x9f	if_icmpeq	branchbyte1 branchbyte2	若栈顶两int类型值相等则跳转。
0xa0	if_icmpne	branchbyte1 branchbyte2	若栈顶两int类型值不相等则跳转。
0xa1	if_icmplt	branchbyte1 branchbyte2	若栈顶两int类型值前小于后则跳转。
0xa4	if_icmple	branchbyte1 branchbyte2	若栈顶两int类型值前小于等于后则跳转。
0xa3	if_icmpgt	branchbyte1 branchbyte2	若栈顶两int类型值前大于后则跳转。
0xa2	if_icmpge	branchbyte1 branchbyte2	若栈顶两int类型值前大于等于后则跳转。
0xc6	ifnull	branchbyte1 branchbyte2	若栈顶引用值为null则跳转。
0xc7	ifnonnull	branchbyte1 branchbyte2	若栈顶引用值不为null则跳转。
0xa5	if_acmpeq	branchbyte1 branchbyte2	若栈顶两引用类型值相等则跳转。
0xa6	if_acmpne	branchbyte1 branchbyte2	若栈顶两引用类型值不相等则跳转。

控制流指令—比较指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0x94	lcmp		比较栈顶两long类型值，前者大，1入栈；相等，0入栈；后者大，-1入栈。
0x95	fcmpl		比较栈顶两float类型值，前者大，1入栈；相等，0入栈；后者大，-1入栈；有NaN存在，-1入栈。
0x96	fcmpg		比较栈顶两float类型值，前者大，1入栈；相等，0入栈；后者大，-1入栈；有NaN存在，-1入栈。
0x97	dcmpl		比较栈顶两double类型值，前者大，1入栈；相等，0入栈；后者大，-1入栈；有NaN存在，-1入栈。
0x98	dcmpg		比较栈顶两double类型值，前者大，1入栈；相等，0入栈；后者大，-1入栈；有NaN存在，-1入栈。

控制流指令—无条件跳转指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0xa7	goto	branchbyte1 branchbyte2	无条件跳转到指定位置。
0xc8	goto_w	branchbyte1 branchbyte2 branchbyte3 branchbyte4	无条件跳转到指定位置（宽索引）。

控制流指令—表跳转指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0xaa	tableswitch	<0-3bytepad> defaultbyte1 defaultbyte2 defaultbyte3 defaultbyte4 lowbyte1 lowbyte2 lowbyte3 lowbyte4 highbyte1 highbyte2 highbyte3 highbyte4 jump offsets...	通过索引访问跳转表，并跳转。
0xab	lookupswitch	<0-3bytepad> defaultbyte1 defaultbyte2 defaultbyte3 defaultbyte4 npairs1 npairs2 npairs3 npairs4 match offsets	通过键值访问跳转表，并跳转。

控制流指令—异常和finally			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0xbf	athrow		抛出异常。
0xa8	jsr	branchbyte1 branchbyte2	跳转到子例程序。
0xc9	jsr_w	branchbyte1 branchbyte2 branchbyte3 branchbyte4	跳转到子例程序（宽索引）。
0xa9	(wide)ret	indexbyte	返回子例程序。
对象操作指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0xbb	new	indexbyte1 indexbyte2	创建新的对象实例。
0xc0	checkcast	indexbyte1 indexbyte	类型强转。
0xc1	instanceof	indexbyte1 indexbyte2	判断类型。
0xb4	getfield	indexbyte1 indexbyte2	获取对象字段的值。
0xb5	putfield	indexbyte1 indexbyte2	给对象字段赋值。
0xb2	getstatic	indexbyte1 indexbyte2	获取静态字段的值。
0xb3	putstatic	indexbyte1 indexbyte2	给静态字段赋值。

数组操作指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0xbc	newarray	atype	创建type类型的数组。
0xbd	anewarray	indexbyte1 indexbyte2	创建引用类型的数组。
0xbe	arraylength		获取一维数组的长度。
0xc5	multianewarray	indexbyte1 indexbyte2 dimension	创建dimension维度的数组。

方法调用指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0xb7	invokespecial	indexbyte1 indexbyte2	编译时方法绑定调用方法。
0xb6	invokevirtual	indexbyte1 indexbyte2	运行时方法绑定调用方法。
0xb8	invokestatic	indexbyte1 indexbyte2	调用静态方法。
0xb9	invokeinterface	indexbyte1 indexbyte2 count 0	调用接口方法。

方法返回指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0xac	ireturn		返回int类型值。
0xad	lreturn		返回long类型值。
0xae	freturn		返回float类型值。
0xaf	dreturn		返回double类型值。
0xb0	areturn		返回引用类型值。
0xb1	return		void函数返回。

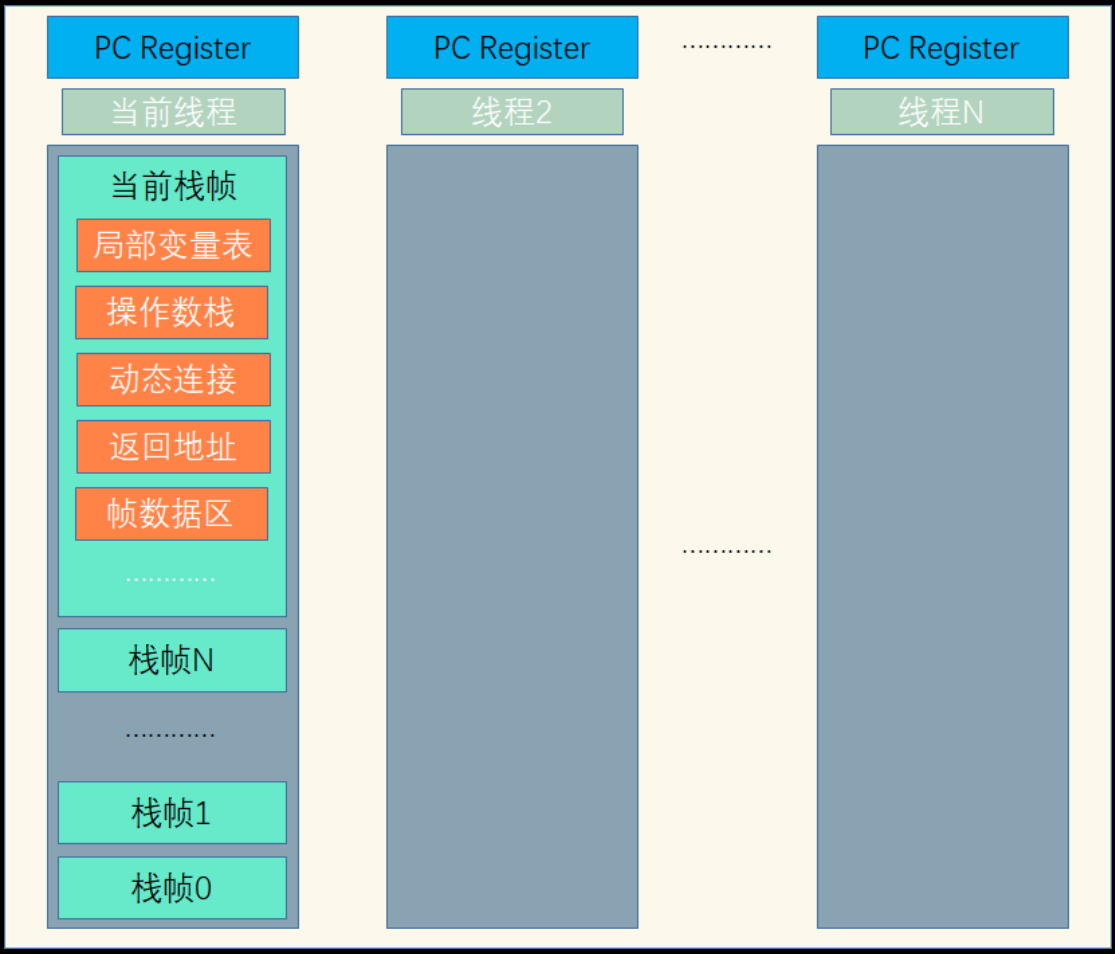
线程同步指令			
指令码	操作码（助记符）	操作数	描述（栈指操作数栈）
0xc2	monitorenter		进入并获得对象监视器。
0xc3	monitorexit		释放并退出对象监视器。

栈帧

一个线程对应一个 JVM Stack。JVM Stack 中包含一组 Stack Frame。线程每调用一个方法就对应着 JVM Stack 中 Stack Frame 的入栈，方法执行完毕或者异常终止对应着出栈（销毁）。

当 JVM 调用一个 Java 方法时，它从对应类的类型信息中得到此方法的**局部变量区和操作数栈**的大小，并据此分配栈帧内存，然后压入 JVM 栈中。

在活动线程中，只有位于栈顶的栈帧才是有效的，称为当前栈帧，与这个栈帧相关联的方法称为当前方法。



一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

局部变量表（Local Variable Table）

编译程序代码的时候就可以确定栈帧中需要多大的局部变量表，具体大小可在编译后的 Class 文件中看到。

局部变量表的容量以 Variable Slot（变量槽）为最小单位，每个变量槽都可以存储 32 位长度的内存空间。

操作数栈（Operand Stack）

同样也可以在编译期确定大小。

Frame 被创建时，操作栈是空的。操作栈的每个项可以存放 JVM 的各种类型数据，其中 long 和 double 类型（64 位数据）占用两个栈深。

方法执行的过程中，会有各种字节码指令往操作数栈中写入和提取内容，也就是出栈和入栈操作

操作栈调用其它有返回结果的方法时，会把结果 push 到栈上

动态链接（Dynamic Linking）

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态链接。

在类加载阶段中的解析阶段会将符号引用转为直接引用，这种转化也称为静态解析。另外的一部分将在运行时转化为直接引用，这部分称为动态链接。

返回地址（Return Address）

方法开始执行后，只有 2 种方式可以退出 ：

1. 方法返回指令，
2. 异常退出。

帧数据区（Stack Data）

帧数据区的大小依赖于 JVM 的具体实现。

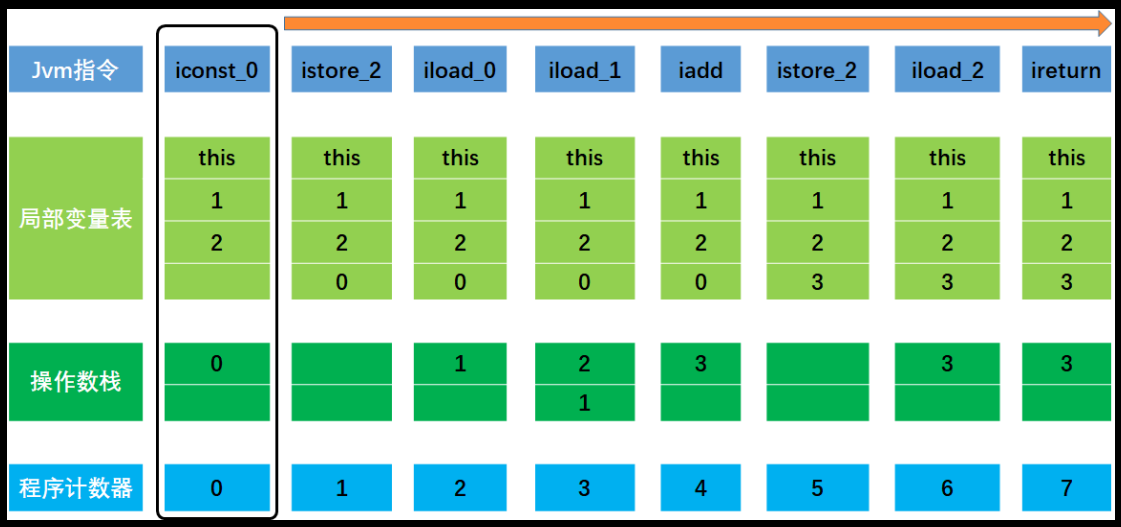
例子：

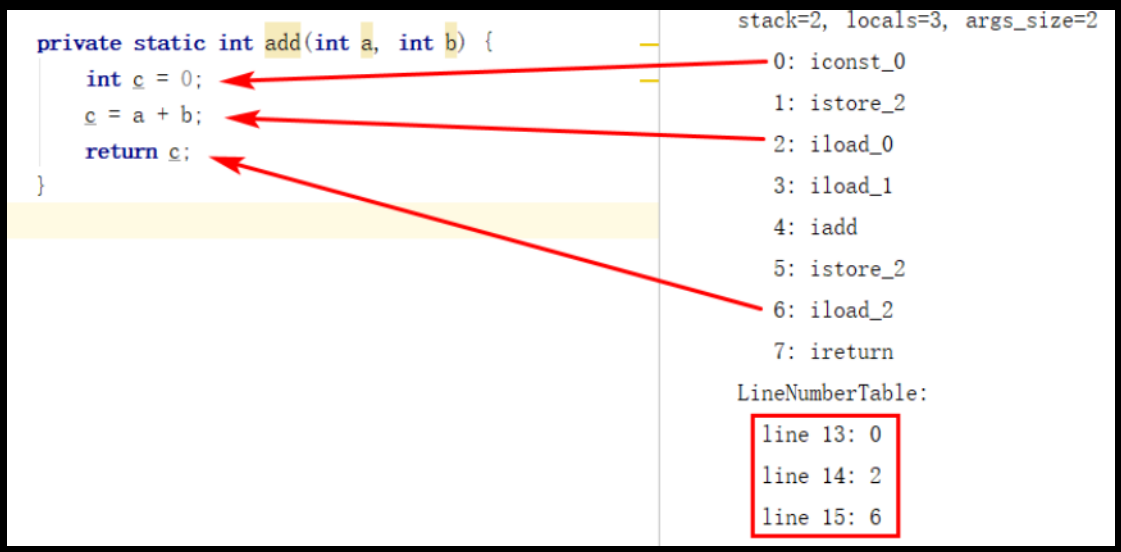
```
  private static int add(int a, int b) {
      int c = 0;
      c = a + b;
      return c;
  }
```

```
private static int add(int, int);
  descriptor: (II)I
  flags: ACC_PRIVATE, ACC_STATIC
  Code:
    stack=2, locals=3, args_size=2
       0: iconst_0
       1: istore_2
       2: iload_0
       3: iload_1
       4: iadd
       5: istore_2
       6: iload_2
       7: ireturn
```

```
# 操作数栈为 2
# 本地变量容量为 3
# 入参个数为 2
stack=2, locals=3, args_size=2
```

add(1,2) 的过程：

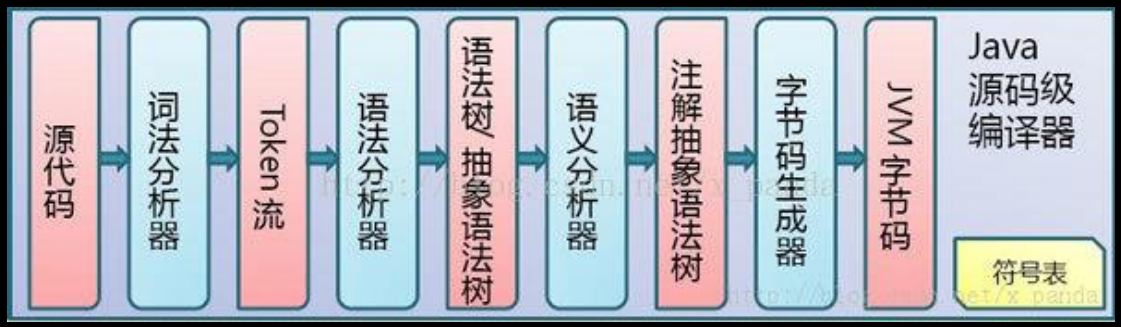




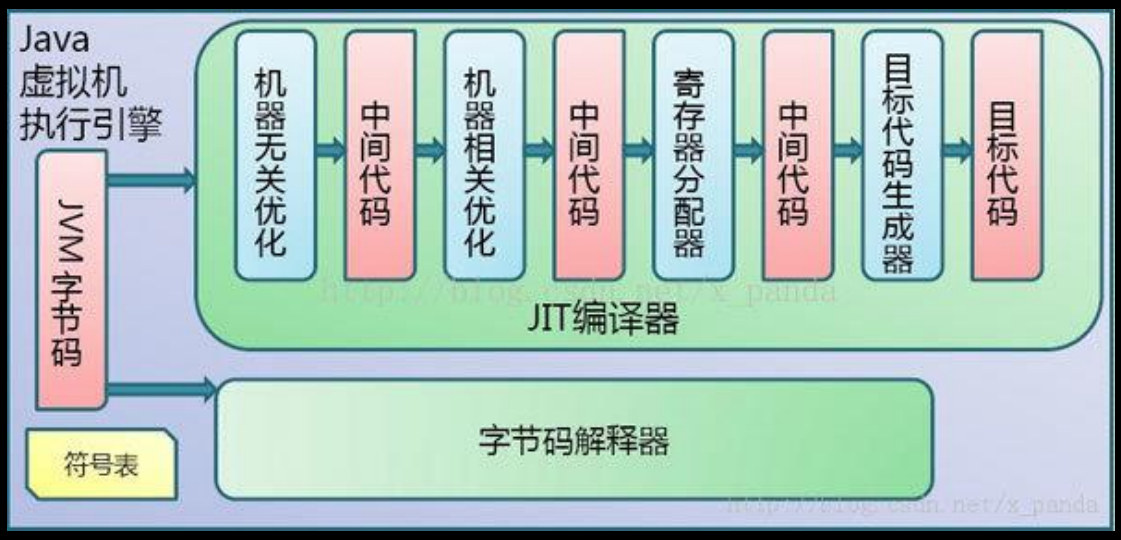
Java 代码编译、执行过程

编译过程：

通过 java 源码编译器完成：



Java 字节码的执行是由 JVM 执行引擎：



Java 代码编译和执行的整个过程包含了以下三个重要的机制：

- Java 源码编译机制
- 类加载机制
- 类执行机制

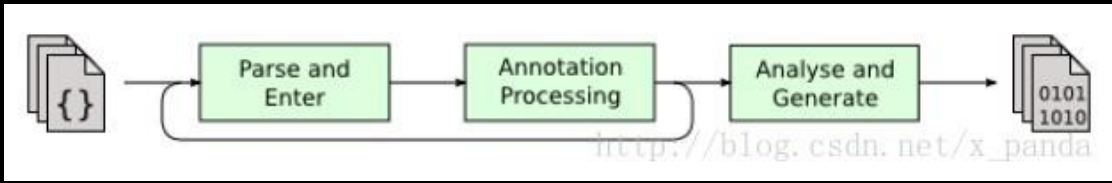
Java 源码编译机制

Java 源码编译由以下三个过程组成：

①分析和输入到符号表

②注解处理

③语义分析和生成 class 文件



最后生成的 class 文件由以下部分组成：

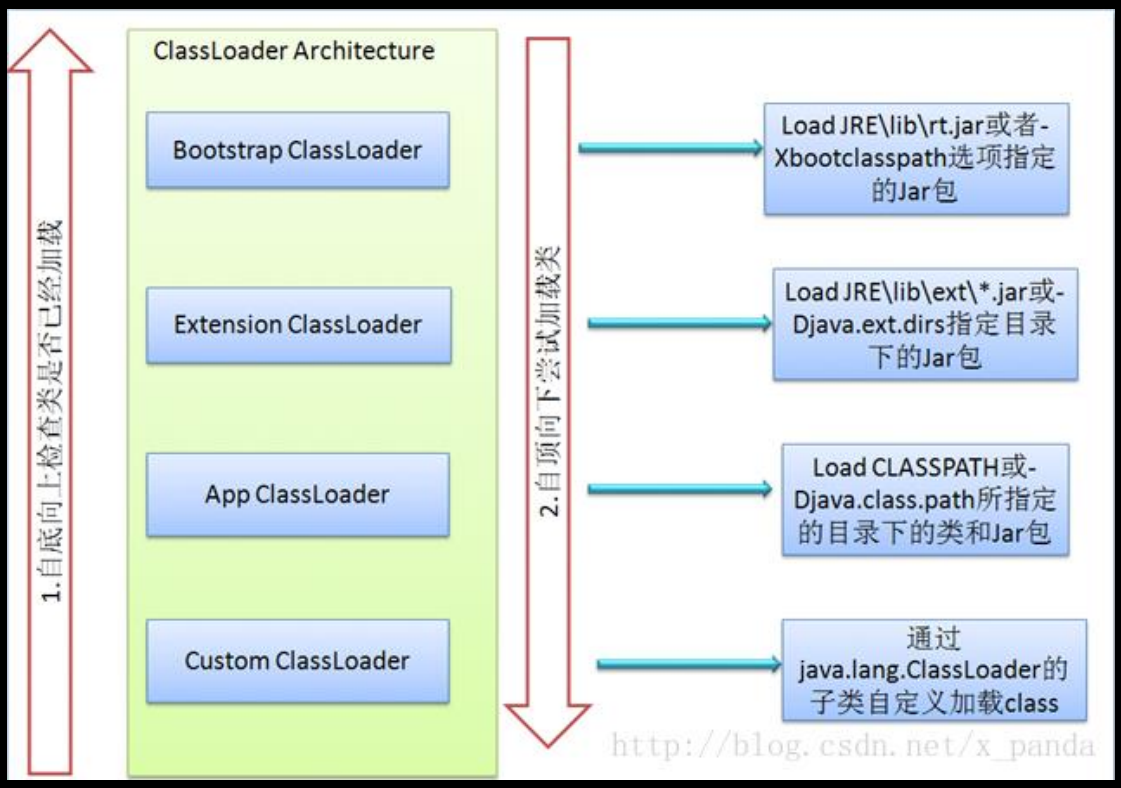
① 结构信息：包括 class 文件格式版本号及各部分的数量与大小的信息

② 元数据：对应于 Java 源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池

③ 方法信息：对应 Java 源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息

类加载机制

JVM 的类加载是通过 ClassLoader 及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



① Bootstrap ClassLoader: 负责加载$JAVA_HOME 中 jre/lib/rt.jar 里所有的 class，由 C++实现，不是 ClassLoader 子类

② Extension ClassLoader: 负责加载 java 平台中扩展功能的一些 jar 包，包括$JAVA_HOME 中 jre/lib/*.jar 或-Djava.ext.dirs 指定目录下的 jar 包

③ App ClassLoader: 负责记载 classpath 中指定的 jar 包及目录中 class

④ Custom ClassLoader: 属于应用程序根据自身需要自定义的 ClassLoader，如 tomcat、jboss 都会根据 j2ee 规范自行实现 ClassLoader

加载过程中会先检查类是否被已加载，检查顺序是自底向上，从 Custom ClassLoader 到 BootStrap ClassLoader 逐层检查，只要某个 classloader 已加载就视为已加载此类，保证此类只所有 ClassLoader 加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

类执行机制

JVM 是基于堆栈的虚拟机。JVM 为**每个新创建的线程都分配一个堆栈**.也就是说，对于一个 Java 程序来说，它的运行就是通过对堆栈的操作来完成的。堆栈以帧为单位保存线程的状态。JVM 对堆栈只进行两种操作：以帧为单位的压栈和出栈操作。

JVM 执行 class 字节码，线程创建后，都会产生程序计数器(PC)和栈(Stack)，程序计数器存放下一条要执行的指令在方法内的偏移量，***栈中存放一个个栈帧，每个栈帧对应着每个方法的每次调用***，而栈帧又是由局部变量区

和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果。栈的结构如下图所示：

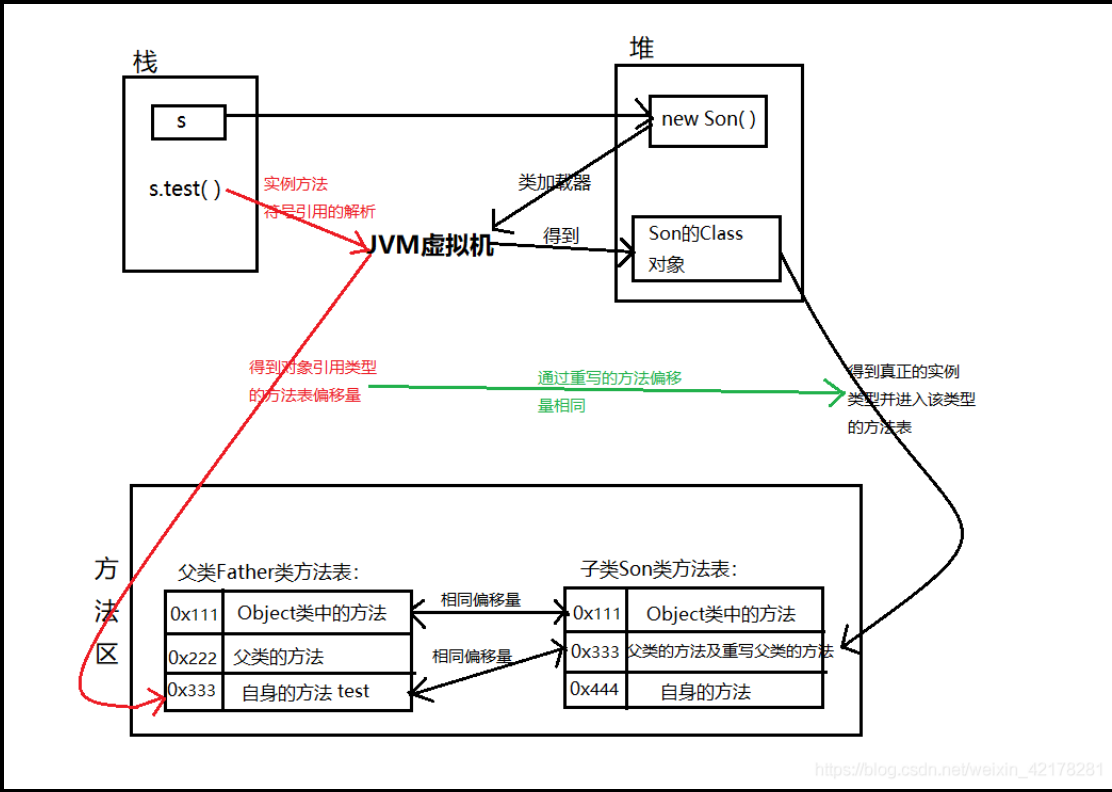
多态的底层实现

多态的底层实现是依靠**动态绑定**

```
class Father {
    public void test(){
        System.out.println("This is Father");
    }
}

 class Son extends Father {
    @Override
    public void test(){
        System.out.println("This is Son");
    }
}

public class TestDemo {
        public static void main(String[] args) {
            Father s = new Son();
            s.test();
        }
    }
```



如上图所示，当我们在执行代码的时候，首先根据我们所写的语法在栈内存上会创建相对应的引用变量 s，相应地在堆内存上开辟空间创建 Son 的实例对象，并且引用 s 指向它的实例 Son，由类的加载过程我们可知道我们所编写的 Class 文件会在 JVM 方法区上建立储存它所含有的类型信息（成员变量、类变量、方法等）并且还会得到一个 Class 对象（通过反射机制）建立在堆区上，该 Class 对象会作为方法区访问数据的入口。

方法区中有方法表，并且方法表中的每一项都是指向相应方法的指针。
JAVA 语言是单继承机制，一个类只能继承一个父类，而所有的类又都继承自 Object 类。方法表有自己的存储机制：**方法表中最先存放的是 Object 类的方法，接下来是父类的方法，最后才是自身特有的方法**。这里的关键点在于如果子类重写了父类的方法，那么子类和父类的同名方法共享一个方法表项，都被认作是父类的方法（仅仅只有非私有的实例方法才行，静态方法是不行的），**即同名（子类重写的）方法在相对应类的方法表中的偏移量是相同的。（和C++类似，虚函数表中重写方法偏移相同）**

结合同名方法偏移量相同且是固定的，则在调用方法时，首先会对实例方法的符号引用进行解析，解析的结果就是方法表的偏移量。当我们把子类对象声明为父类类型时，明面上虚拟机通过对象引用的类型得到该类型方法区中类型信息的入口，去查询该类型的方法表（即例中的 Father），得到的是父类型的方法表中的 test 方法的偏移量，但实际上编译器通过类加载过程获取到 Class 对象知道了实例对象 s 的真正类型，转而进入到了真正的子类类型（例中的 Son）的方法表

中用偏移量寻找方法，恰好两者偏移量是相等的，我们就顺利成章的拿到了 Son 类型方法表中的 test 方法进而去指向 test 方法入口。

态的好处：
1. 应用程序不必为每一个派生类编写功能调用，只需要对抽象基类进行处理即可。大大提高程序的可复用性。（继承保证）
2. 派生类的功能可以被基类的方法或引用变量所调用，这叫向后兼容，
3. 可以提高可扩充性和可维护性。（多态保证）

不定参数的底层实现

```
1      public static void main(String[] args) {
2          test("a", "b", "c");
3      }
4
5      public static void test(String... strs) {
6          System.out.println(strs.getClass().getSimpleName());
7          System.out.println(strs);
8      }
```

反编译.class源码可看到

```
1      public static void main(String[] args) {
2          test(new String[] { "a", "b", "c" });
3      }
4
5      public static void test(String... strs) {
6          System.out.println(strs.getClass().getSimpleName());
7          System.out.println(strs);
8      }
```

（个人感觉类似语法糖，编译器理解为一个数组）

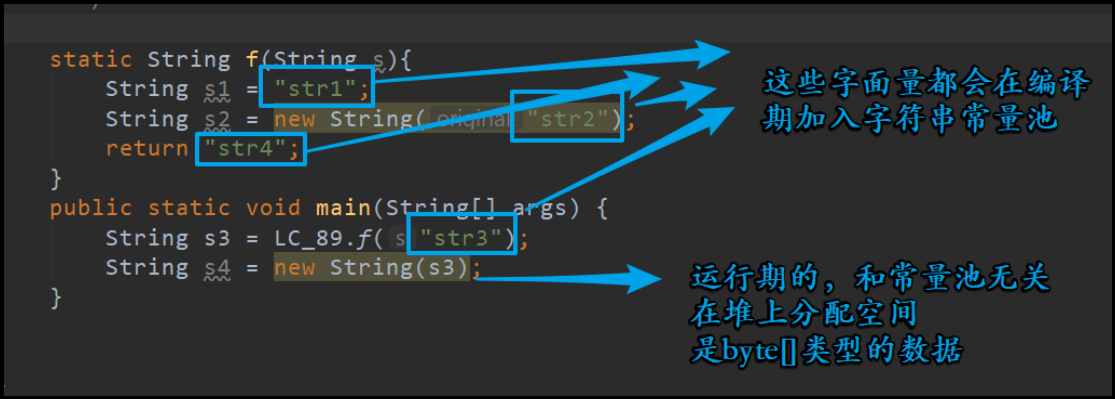
session：每次随机创建一个
设计更新 name 的 api
- c/s 采用什么传输协议

-
hashmap
concurentHashmap: 不是 syncrized
类加载器、双亲委派模型，在字节码上怎么实现的

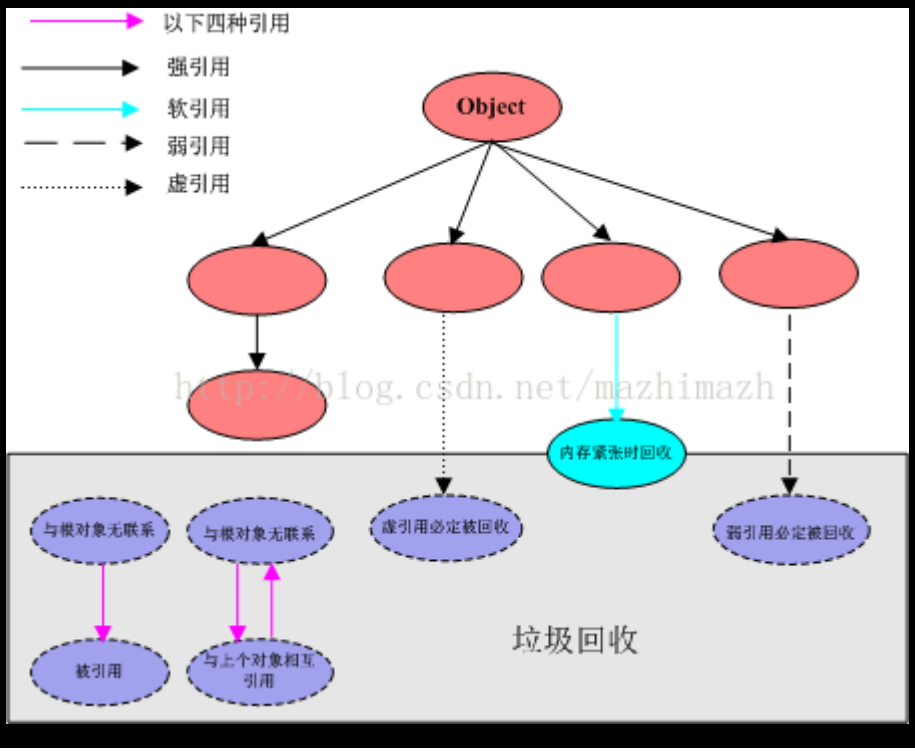
String\字符串常量池

字符串常量池放在 堆 中
普通常量池放在方法区

在编译时期，编译器会将所有的字面量处理掉，**如果是字符串字面量，就加入字符串常量池**，比如，函数调用中传入字符串字面量，编译期就会加入字符串常量池。



引用



引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时终止
软引用	在内存不足时	对象缓存	内存不足时终止
弱引用	在垃圾回收时	对象缓存	gc运行后终止
虚引用	Unknown	Unknown	Unknown

当垃圾回收器回收时，某些对象会被回收，某些不会被回收。垃圾回收器会从根对象 Object 来标记存活的对象，然后将某些不可达的对象和一些引用的对象进行回收

虚引用（PhantomReference）

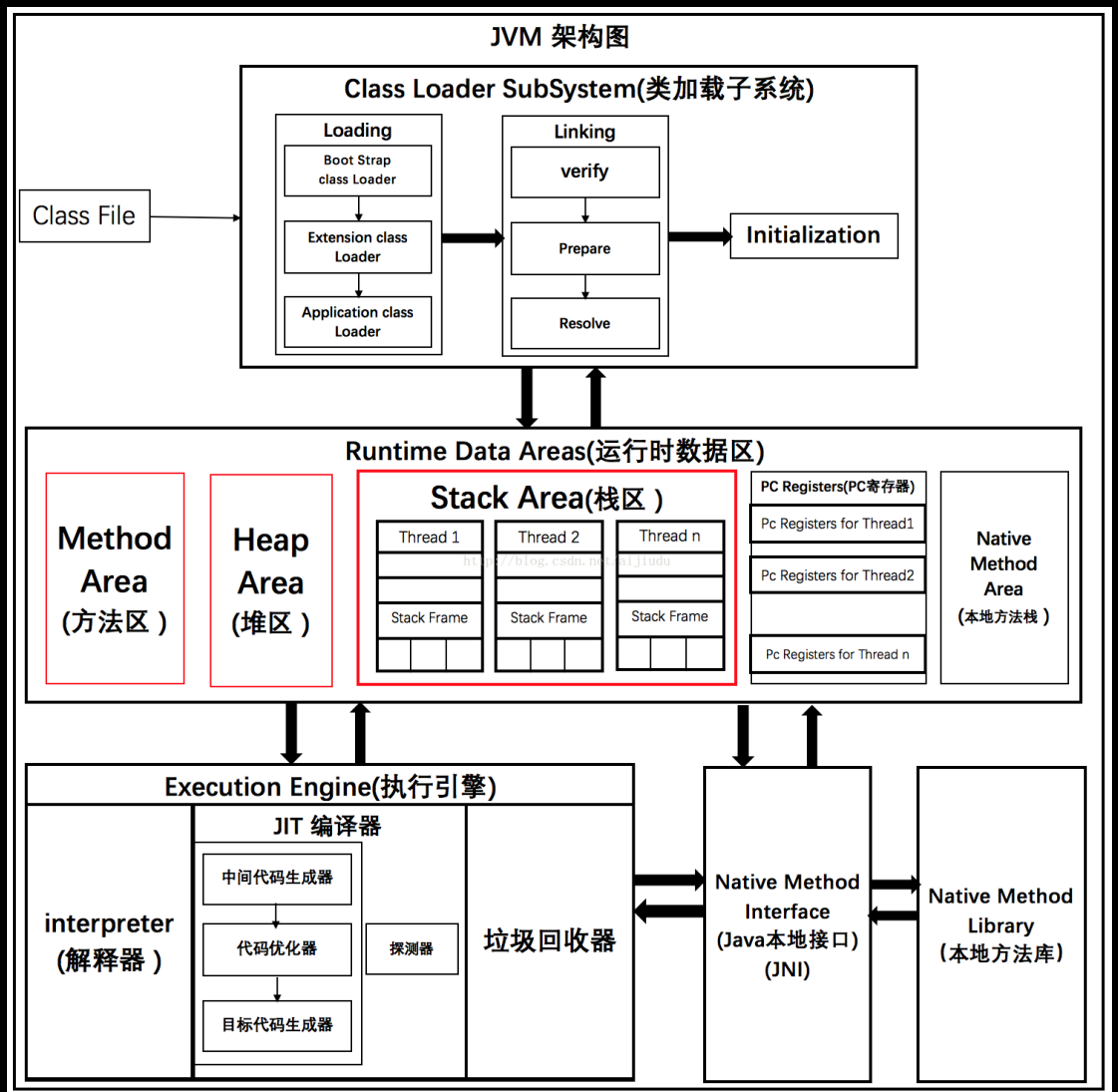
“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

虚引用**主要用来跟踪对象被垃圾回收器回收的活动**。

虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收

一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。你声明虚引用的时候是要传入一个 queue 的。当你的虚引用所引用的对象已经执行完 finalize 函数的时候，就会把对象加到 queue 里面。你可以通过判断 queue 里面是不是有对象来判断你的对象是不是要被回收了

JVM 架构图



虚拟机栈 vs. 操作数栈

虚拟机栈以 栈帧为单位出入栈，而操作数栈是栈帧中的。

直接指针 vs. 句柄

栈中的对象引用如何指向堆中的对象？

1. 直接指针：直接指向堆中对象地址
   缺点：如果 GC 导致对象地址变更，需要改变该指针
2. 句柄：在堆中开辟一段空间，维护一张句柄表，栈中引用指向句柄表项，表项指向堆中的对象。
   GC 后对象地址变化后只需要维护句柄表就行，缺点是空间上开销，时间上间接访问。

SpotHot 只有直接引用？

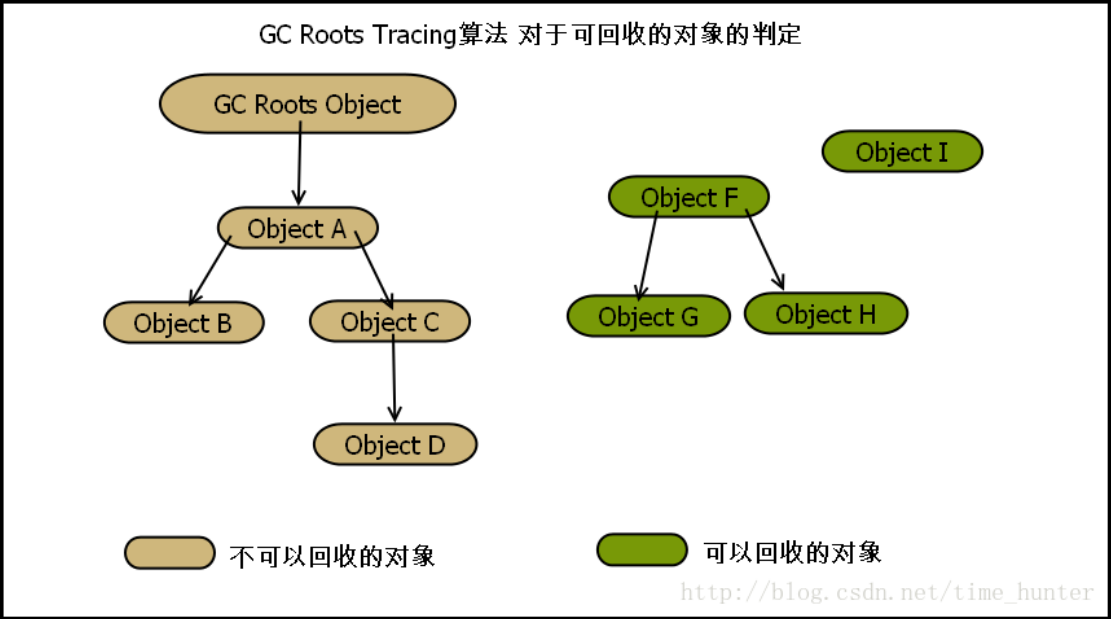
JVM 调优参数

-Xmx3550m：设置 JVM 最大可用内存为 3550M。
-Xms3550m：设置 JVM 促使内存为 3550m。此值可以设置与-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。
-Xmn2g：设置年轻代大小为 2G。整个 JVM 内存大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为 64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。
-Xss128k：设置每个线程的堆栈大小。JDK5.0 以后每个线程堆栈大小为 1M，以前每个线程堆栈大小为 256K。
-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）。设置为 4，则年轻代与年老代所占比值为 1：4，年轻代占整个堆栈的 1/5
-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:MaxPermSize=16m:设置持久代大小为 16m。
-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。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。

GC Roots Tracing

JVM 中对内存进行回收时，需要判断对象是否仍在使用中，可以通过 GC Roots Tracing 辨别
通过一系列名为”GCRoots”的对象作为起始点，从这个节点向下搜索，搜索走过的路径称为 ReferenceChain，当一个对象到 GCRoots 没有任何 ReferenceChain 相连时，(图论：这个对象不可到达)，则证明这个对象不可用。

----可达性分析算法



可以作为 GC Root 引用点的是:

1. Java Stack 中的引用的对象。
2. 方法区中静态引用指向的对象。
3. 方法区中常量引用指向的对象。
4. Native 方法中 JNI 引用的对象。

GC 管理的主要区域是 Java 堆，一般情况下只针对堆进行垃圾回收。方法区、栈和本地方法区不被 GC 所管理,因而选择这些区域内的对象作为 GC roots,被 GC roots 引用的对象不被 GC 回收。

被 GC 判断为”垃圾”的对象一定会回收吗?

即使在可达性分析算法中不可达的对象,也并非是“非死不可”的,这时候它们暂时处于“缓刑”阶段,要真正宣告一个对象死亡,至少要经历两次标记过程:如果对象在进行可达性分析后发现没有与 GC Roots 相连接的引用链,那它将会被第一次标记并且进行一次筛选,筛选的条件是此对象是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法,或者 finalize()方法已经被虚拟机调用过,虚拟机将这两种情况都视为“没有必要执行”。(即意味着直接回收)

如果这个对象被判定为有必要执行 finalize()方法,那么这个对象将会放置在一个叫做 F-Queue 的队列之中,并在稍后由一个由虚拟机自动建立的、低优先级的 Finalizer 线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法,但并不承诺会等待它运行结束,这样做的原因是,如果一个对象在 finalize()方法中执行缓慢,或者发生了死循环(更极端的情况),将很可能会导致 F-Queue 队列中其他对象永久处于等待,甚至导致整个内存回收系统崩溃。

finalize()方法是对象逃脱死亡命运的最后一次机会,稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记,如果**对象要在 finalize()中成功拯救自己**

——只要重新与引用链上的任何一个对象建立关联即可,譬如把自己(this 关键字)赋值给某个类变量或者对象的成员变量,那在第二次标记时它将被移除出“即将回收”的集合;如果对象这时候还没有逃脱,那基本上它就真的被回收了。

finalize 流程：当对象变成(GC Roots)不可达时，GC 会判断该对象是否覆盖了 finalize 方法,若未覆盖,则直接将其回收。否则,若对象未执行过 finalize 方法，将其放入 F-Queue 队列，由一低优先级线程执行该队列中对象的 finalize 方法。执行 finalize 方法完毕后，GC 会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”。

垃圾回收器准备释放内存的时候，会先调用 finalize()。
(1).对象不一定会被回收。
(2).垃圾回收不是析构函数。
(3).垃圾回收只与内存有关。
(4).垃圾回收和 finalize()都是靠不住的，只要 JVM 还没有快到耗尽内存的地步，它是不会浪费时间进行垃圾回收

之所以要使用 finalize()，是存在着垃圾回收器不能处理的特殊情况。假定你的对象（并非使用 new 方法）获得了一块“特殊”的内存区域，由于垃圾回收器只知道那些显示地经由 new 分配的内存空间，所以它不知道该如何释放这块“特殊”的内存区域,那么这个时候 Java 允许在类中定义一个由 finalize()方法。

特殊的区域例如：1）由于在分配内存的时候可能采用了类似 C 语言的做法，而非 JAVA 的通常 new 做法。这种情况主要发生在 native method 中，比如 native method 调用了 C/C++方法 malloc()函数系列来分配存储空间，但是除非调用 free()函数，否则这些内存空间将不会得到释放，那么这个时候就可能造成内存泄漏。但是由于 free()方法是在 C/C++中的函数,所以 finalize()中可以用本地方法来调用它。以释放这些“特殊”的内存空间。2）又或者打开的文件资源，这些资源不属于垃圾回收器的回收范围。

换言之，finalize()的主要用途是释放一些其他做法开辟的内存空间，以及做一些清理工作。因为在 JAVA 中并没有提够像“析构”函数或者类似概念的函数，要做一些类似清理工作的时候，必须自己动手创建一个执行清理工作的普通方法，也就是 override Object 这个类中的 finalize()方法。例如，假设某一个对象在创建过程中会将自己绘制到屏幕上,如果不是明确地从屏幕上将其擦出,它可能永远都不会被清理。如果在 finalize()加入某一种擦除功能，当 GC 工作时，finalize()得到了调用，图像就会被擦除。要是 GC 没有发生，那么这个图像就会被一直保存下来。

一旦垃圾回收器准备好释放对象占用的存储空间，首先会去调用 finalize()方法进行一些必要的清理工作。只有到下一次再进行垃圾回收动作的时候，才会真正释放这个对象所占用的内存空间。

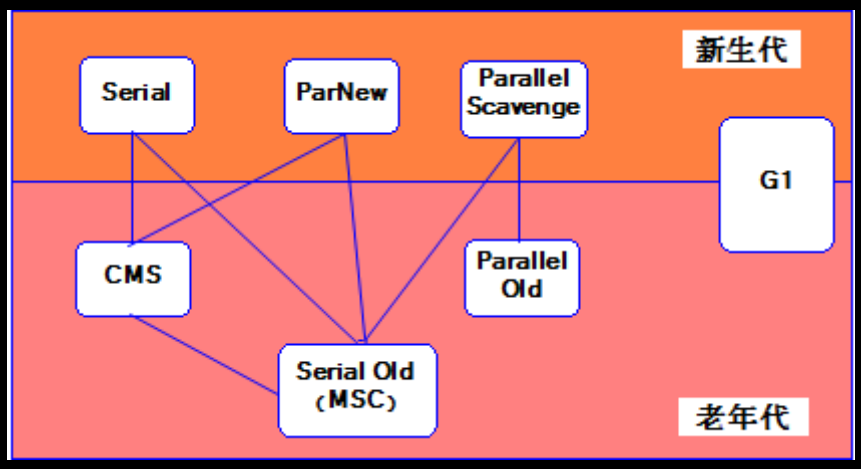
finalize()函数是在垃圾回收器准备释放对象占用的存储空间的时候被调用的，绝对不能直接调用 finalize()，所以应尽量避免用它

GC

在 GC 之前对象是存在 Eden 和 from 中的，进行 GC 的时候 Eden 中的对象被拷贝到 To 这样一个 survive 空间（survive（幸存）空间：包括 from 和 to，他们的空间大小是一样的，又叫 s1 和 s2）中（有一个拷贝算法），From 中的对象（算法会考虑经过 GC 幸存的次数）到一定次数（阈值（如果说每次 GC 之后这个对象依旧在 Survive 中存在，GC 一次他的 Age 就会加 1，默认 15 就会放到 OldGeneration。但是实际情况比较复杂，有可能没有到阈值就从 Survive 区域直接到 Old Generation 区域。在进行 GC 的时候会对 Survive 中的对象进行判断,Survive 空间中有一些对象 Age 是一样的,也就是经过的 GC 次数一样，年龄相同的这样一批对象的总和大于等于 Survive 空间一半的话，这组对象就会进入 old Generation 中，（是一种动态的调整））），会被复制到 OldGeneration，如果没到次数 From 中的对象会被复制到 To 中，复制完成后 To 中保存的是有效的对象，Eden 和 From 中剩下的都是无效的对象，这个时候就把 Eden 和 From 中所有的对象清空。在复制的时候 Eden 中的对象进入 To 中，To 可能已经满了，这个时候 Eden 中的对象就会被直接复制到 Old Generation 中，From 中的对象也会直接进入 Old Generation 中。就是存在这样一种情况，To 比较小，第一次复制的时候空间就满了，直接进入 old Generation 中。复制完成后，To 和 From 的名字会对调一下，因为 Eden 和 From 都是空的，对调后 Eden 和 To 都是空的，下次分配就会分配到 Eden。一直循环这个流程。好处：使用对象最多和效率最高的就是在 Young Generation 中，通过 From to 就避免过于频繁的产生 FullGC（Old Generation 满了一般都会产生 FullGC）

垃圾回收器

HotSpot 虚拟机所包含的收集器：



如果两个收集器之间存在连线，则说明它们可以搭配使用。

1. 新生代收集器：Serial、ParNew、Parallel Scavenge
2. 老年代收集器：CMS、Serial Old、Parallel Old
3. 整堆收集器： G1

相关概念：

1. 并行收集：指多条垃圾收集线程并行工作，但此时用户线程仍处于等待状态。
2. 并发收集：指用户线程与垃圾收集线程同时工作（不一定是并行的可能会交替执行）。用户程序在继续运行，而垃圾收集程序运行在另一个 CPU 上。
3. 吞吐量：即 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值（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)）。例如：虚拟机共运行 100 分钟，垃圾收集器花掉 1 分钟，那么吞吐量就是 99%
4. Java 中 Stop-The-World 机制简称 STW，是在执行垃圾收集算法时，Java 应用程序的其他所有线程都被挂起（除了垃圾收集帮助器之外）。Java 中一种全局暂停现象，全局停顿，所有 Java 代码停止，native 代码可以执行，但不能与 JVM 交互；这些现象多半是由于 gc 引起。

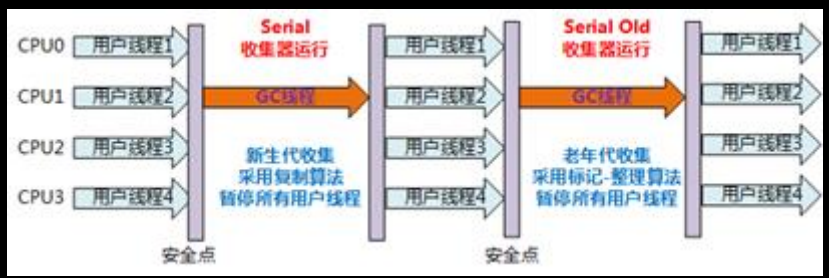
Serial 收集器

最基本的、发展历史最悠久

1. 单线程：无线程切换开销
2. Stop the world

应用场景：适用于 Client 模式下的虚拟机

Serial / Serial Old 收集器运行示意图：



ParNew 收集器

其实就是Serial收集器的多线程版本。除了使用多线程外其余行为均和Serial

收集器一模一样

1. 多线程：默认开启的收集线程数与 CPU 的数量相同
2. 存在 Stop The World 问题

应用场景：ParNew 收集器是许多运行在 Server 模式下的虚拟机中首选的新生代收集器，因为它是除了 Serial 收集器外，唯一一个能与 CMS 收集器配合工作的。

ParNew/Serial Old 组合收集器运行示意图如下：

Parallel Scavenge 收集器

与吞吐量关系密切，故也称为吞吐量优先收集器。

Scavenge：打扫；排除废气

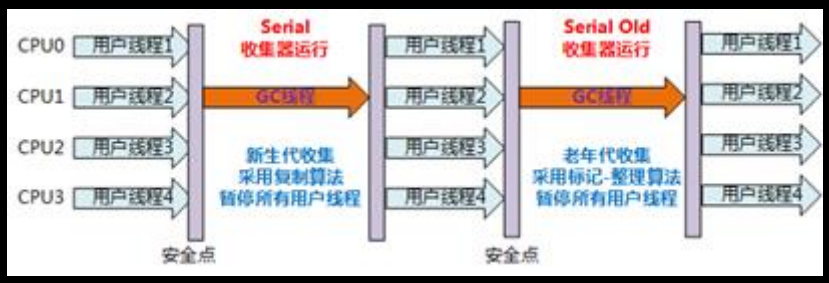
1. 属于新生代收集器
2. 也是采用复制算法的收集器
3. 并行的多线程收集器
4. GC 自适应调节策略：Parallel Scavenge 收集器可设置 -XX:+UseAdptiveSizePolicy 参数。当开关打开时不需要手动指定新生代的大小（-Xmn）、Eden 与 Survivor 区的比例（-XX:SurvivorRation）、晋升老年代的对象年龄（-XX:PretenureSizeThreshold）等，虚拟机会根据系统的运行状况收集性能监控信息，动态设置这些参数以提供最优的停顿时间和最高的吞吐量，这种调节方式称为 GC 的自适应调节策略。

Serial Old 收集器

1. Serial Old 是 Serial 收集器的老年代版本。
2. 同样是单线程收集器，
3. 采用标记-整理算法。

应用场景：主要也是使用在 Client 模式下的虚拟机中。也可在 Server 模式下使用。

Serial / Serial Old 收集器工作过程图（Serial 收集器图示相同）：

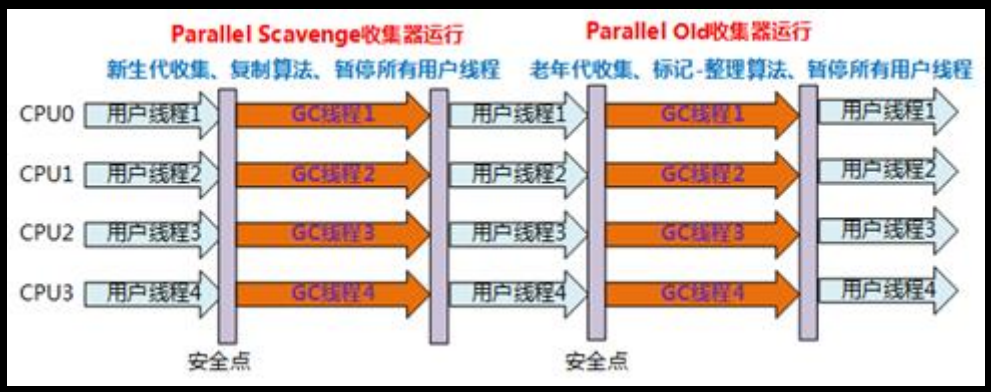


Parallel Old 收集器

1. 是 Parallel Scavenge 收集器的老年代版本。
2. 多线程
3. 采用标记-整理算法。

应用场景：注重高吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge+Parallel Old 收集器。

Parallel Scavenge/Parallel Old 收集器工作过程图：



CMS 收集器

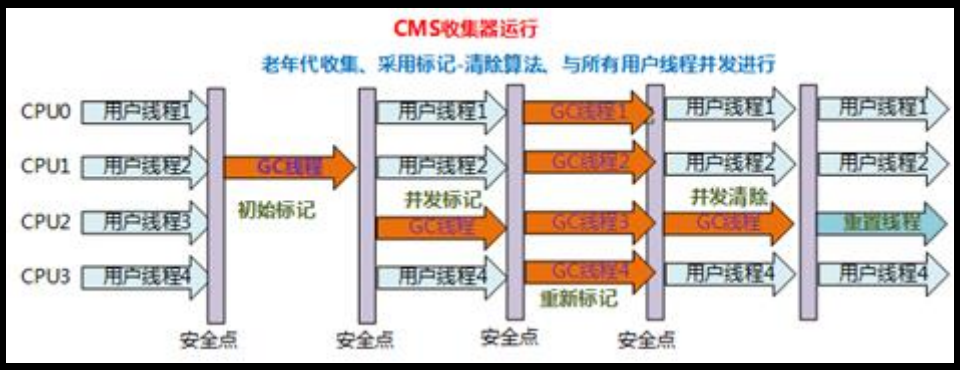
一种以获取**最短回收停顿时间**为目标的收集器。

1. 基于标记-清除算法实现。
2. 并发收集、
3. 低停顿/低延迟

应用场景：适用于注重服务的响应速度，希望系统停顿时间最短、低延迟，给用户带来更好的体验等场景下。如 web 程序、b/s 服务。

CMS 收集器的运行过程分为下列 4 步：

1. 初始标记：标记 GC Roots 能直接到的对象。速度很快但是仍存在 Stop The World 问题。
2. 并发标记：进行 GC Roots Tracing 的过程，找出存活对象且用户线程可并发执行。
3. 重新标记：为了修正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。仍然存在 Stop The World 问题。
4. 并发清除：对标记的对象进行清除回收。内存回收过程是与用户线程一起并发执行的。



缺点：

1. 对 CPU 资源非常敏感。
2. 无法处理浮动垃圾，可能出现 Concurrent Model Failure 失败而导致另一次 Full GC 的产生。
3. 因为采用标记-清除算法所以会存在空间碎片的问题，导致大对象无法分配空间，不得不提前触发一次 Full GC。

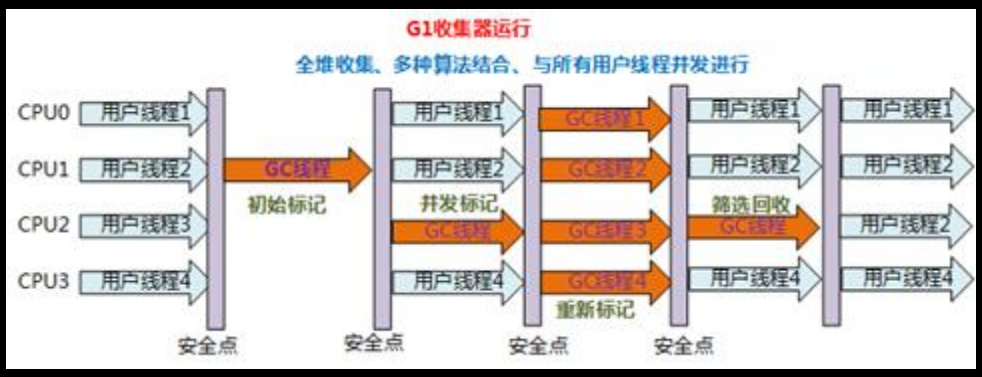
G1 收集器

一款**面向服务端应用**的垃圾收集器。

1. 并行与并发：G1 能充分利用多 CPU、多核环境下的硬件优势，使用多个 CPU 来缩短 Stop-The-World 停顿时间。部分收集器原本需要停顿 Java 线程来执行 GC 动作，G1 收集器仍然可以通过并发的方式让 Java 程序继续运行。
2. 分代收集：G1 能够独自管理整个 Java 堆，并且采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次 GC 的旧对象以获取更好的收集效果。
3. 空间整合：G1 运作期间不会产生空间碎片，收集后能提供规整的可用内存。
4. 可预测的停顿：G1 除了追求低停顿外，还能建立可预测的停顿时间模型。能让使用者明确指定在一个长度为 M 毫秒的时间段内，消耗在垃圾收集上的时间不得超过 N 毫秒。

G1 收集器大致可分为如下步骤：

1. 初始标记：仅标记 GC Roots 能直接到的对象，并且修改 TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的 Region 中创建新对象。（需要线程停顿，但耗时很短。）
2. 并发标记：从 GC Roots 开始对堆中对象进行可达性分析，找出存活对象。（耗时较长，但可与用户程序并发执行）
3. 最终标记：为了修正在并发标记期间因用户程序执行而导致标记产生变化的那一部分标记记录。且对象的变化记录在线程 Remembered Set Logs 里面，把 Remembered Set Logs 里面的数据合并到 Remembered Set 中。（需要线程停顿，但可并行执行。）
4. 筛选回收：对各个 Region 的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。（可并发执行）



Minor GC、Major GC、Full GC

年轻代内存垃圾回收叫做 Minor GC。
大多数新建的对象都位于 Eden 区。当 Eden 区被对象填满时，就会执行 Minor GC。并把所有存活下来的对象转移到其中一个 survivor 区。

Minor GC 触发机制：
当**年轻代满时就会触发 Minor GC,这里的年轻代满指的是 Eden** 代满,Survivor 满不会引发 GC。

年老代内存里包含了长期存活的对象和经过多次 Minor GC 后依然存活下来的对象。
老年代 GC（**Major GC / Full GC**）：指发生在老年代的 GC，出现了 Major GC
Full GC 触发机制：
（1）调用 System.gc 时，系统建议执行 Full GC，但是不必然执行
（2）老年代空间不足
（3）方法区空间不足
（4）通过 Minor GC 后进入老年代的平均大小大于老年代的可用内存
（5）由 Eden 区、survivor space1（From Space）区向 survivor space2（To Space）区复制时，对象大小大于 To Space 可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

类名.class、object.getClass()、Class.forName()

都是获取一个类的 Class 对象
区别：
1. 类的加载方式不同
   a) Class.forName()属于**动态加载**类，在代码运行时加载指定类；

b) Class.class 属于静态加载类，在代码编译时加载指定类；
c) object.getClass()取决于对象的产生方式：既可以是静态加载类（通过new创建的对象），也可以是动态加载类（通过Class.forName(xx.xx)创建的对象，对象可能不存在）。

```
Class.forName("bbb").newInstance().getClass();
```

2. Class 对象的创建方式不同
   a) Class.forName()
      运行阶段：JVM 使用类装载器，将类装入内存中，并对类进行初始化（静态代码块、非静态代码块、构造函数调用及静态变量）。最后返回 Class 的对象。
   b) 类名.class
      编译阶段：JVM 使用类装载器，将类装入内存中，并对类进行初始化操作
   c) 实例对象.getClass()
      当调用 getClass()方法时，该对象的类已经被加载到内存中并且完成了初始化操作；直接返回 Class 的对象，没有其它操作。

异常的底层原理

```
Code:
  stack=1, locals=7, args_size=1
     0: iconst_1              // 将int型常量1入栈
     1: istore_1              // 将栈顶int型的数赋予第二个局部变量（即将常量1赋值到局部变量a）
     2: iconst_3              // 将int型常量3入栈
     3: istore_3              // 将栈顶的数赋予第四个局部变量（即将常量3赋值到局部变量c）
     4: goto          23      // 跳转到偏移量为21的指令
     7: astore        5       // 栈顶ref对象存入第1局部变量
     9: iconst_2              // 将int型常量2入栈
    10: istore_2              // 将栈顶的数赋予第三个局部变量（即将常量3赋值到局部变量b）
    11: iconst_3              // 将int型常量3入栈
    12: istore_3              // 将栈顶的数赋予第四个局部变量（即将常量3赋值到局部变量c）
    13: goto          23
    16: astore        6
    18: iconst_3
    19: istore_3
    20: aload         6
    22: athrow
    23: iconst_4
    24: istore        4
    26: return
```

异常表

```
Exception table:
   from    to  target type
       0     2     7   Class java/lang/Exception
       0     2    16   any
       7    11    16   any
      16    18    16   any
```

```
public class Exception1{
    public static void main(String[] args){
        int a,b,c,d;
        try{
            a =1 ;

        }catch(Exception e){
            b = 2;
        }finally{
            c = 3 ;
        }
         d = 4;
    }
}
```

异常表：

1. Exception table：异常处理信息表
2. from 异常处理开始的位置
3. to 异常处理结束的位置
4. from和to结合起来就是异常处理的位置。在上述例子中，Exception table的第一行from和to分别表示为0,2，这里代表着异常是从0开始，到2结束(不包括2)，表示的是try包含的代码块。
5. target 异常处理器的起始位置，即catch开始处理的位置
6. type 异常类型，any表示所有类型

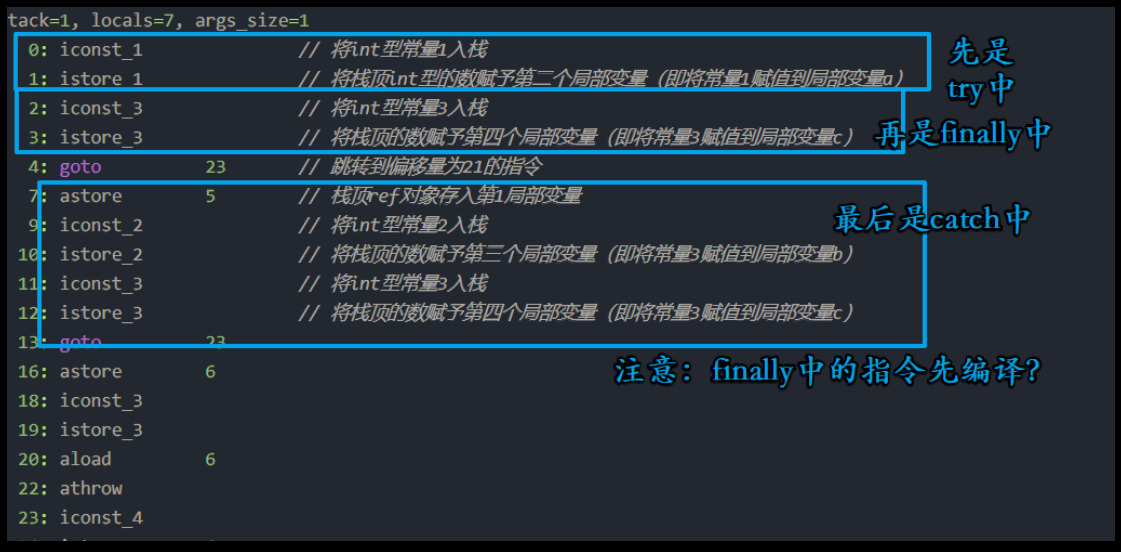
异常处理的过程：

当程序触发异常时，Java虚拟机会遍历异常表中的所有条目（即try里面的所有代码）。如果异常找到异常发生的字节码条目，则会跟catch要捕捉的异常匹配，如果匹配，则开始执行catch里面的代码。

如果没有匹配到，那么它会弹出当前方法对应的 Java 栈帧，并且在调用者（caller）中重复上述操作。在最坏情况下，Java 虚拟机需要遍历当前线程 Java 栈上所有方法的异常表

如果在catch中发生了异常，那么Java虚拟机会抛弃第一个异常，尝试捕获并处理新的异常。这个在编码中是很不利于调试的。

(应该是在某一行X发生了异常，然后去查异常表，找到X在From到To之间的一个条目，跳转到target处。)



finally总会被执行的原理：

catch处理完后会跳转到finally处

还需要仔细搞搞这个上面几行代码？？？？？？？？？？？？？？？？

Edwin Xu