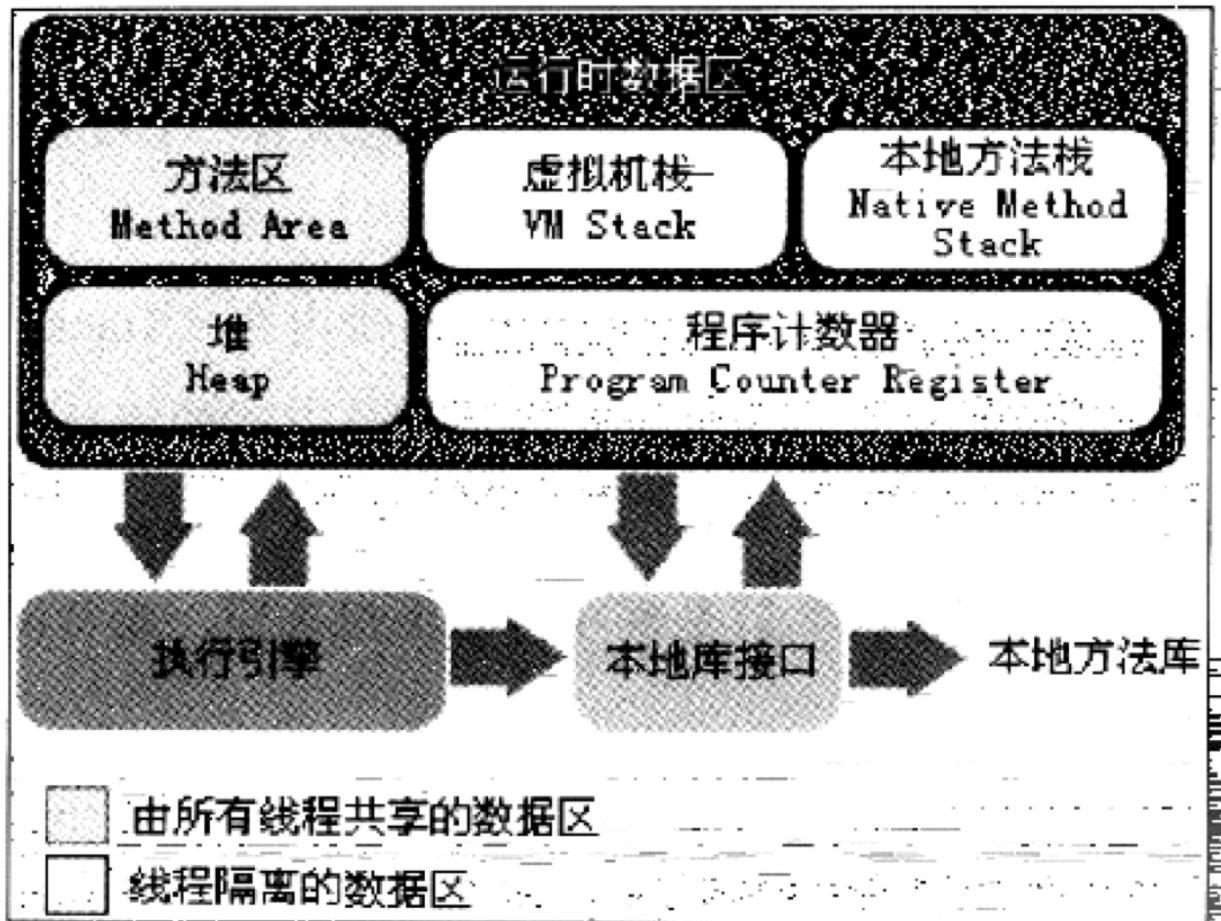


JVM



一：jvm内存结构

1.程序计数器 (pc)

1.1作用：当前线程所执行的字节码的行号指示器

1.2注： 1>每个线程都有一条独立的pc

2>此区域时唯一一个在虚拟机规范中没有规定任何OutOfMemoryError的区域

2.虚拟机栈

虚拟机栈也是线程私有的，每个方法在执行同时都会创建一个栈帧，每一个方法从调用到执行的过程就对应着一个栈帧在虚拟机栈中从出栈入栈的过程

在 Java 虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 `StackOverflowError` 异常；如果虚拟机栈可以动态扩展（当前大部分的 Java 虚拟机都可动态扩展，只不过 Java 虚拟机规范中也允许固定长度的虚拟机栈），如果扩展时无法申请到足够的内存，就会抛出 `OutOfMemoryError` 异常。

3..本地方法栈

本地方法栈（Native Method Stack）与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务

4.堆

4.1 堆被所有线程共享，用于存放对象实例

4.2 会抛出 OOMError 异常

5.方法区

元空间是方法区的实现

方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然 Java 虚拟机

内存溢出案例：

```
/*
 * 演示元空间内存溢出 java.lang.OutOfMemoryError: Metaspace
 * -XX:MaxMetaspaceSize=8m
 */
public class Demo1_8 extends ClassLoader { // 可以用来加载类的二进制字节码
    public static void main(String[] args) {
        int j = 0;
        try {
            Demo1_8 test = new Demo1_8();
            for (int i = 0; i < 10000; i++, j++) {
                // ClassWriter 作用是生成类的二进制字节码
                ClassWriter cw = new ClassWriter(0);
                // 版本号, public, 类名, 包名, 父类, 接口
                cw.visit(Opcodes.V1_8, Opcodes.ACC_PUBLIC, "Class" + i, null, "java/lang/Object", null);
                // 返回 byte[]
                byte[] code = cw.toByteArray();
                // 执行了类的加载
                test.defineClass("Class" + i, code, 0, code.length); // Class 对象
            }
        } finally {
            System.out.println(j);
        }
    }
}
```

spring mybatis 动态加载类，使用不当就会导致 OOM

6.运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

运行时常量池相对于 Class 文件常量池的另外一个重要特征是具备动态性，Java 语言并不要求常量一定只有编译期才能产生，也就是并非预置入 Class 文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是 String 类的 intern() 方法。

既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

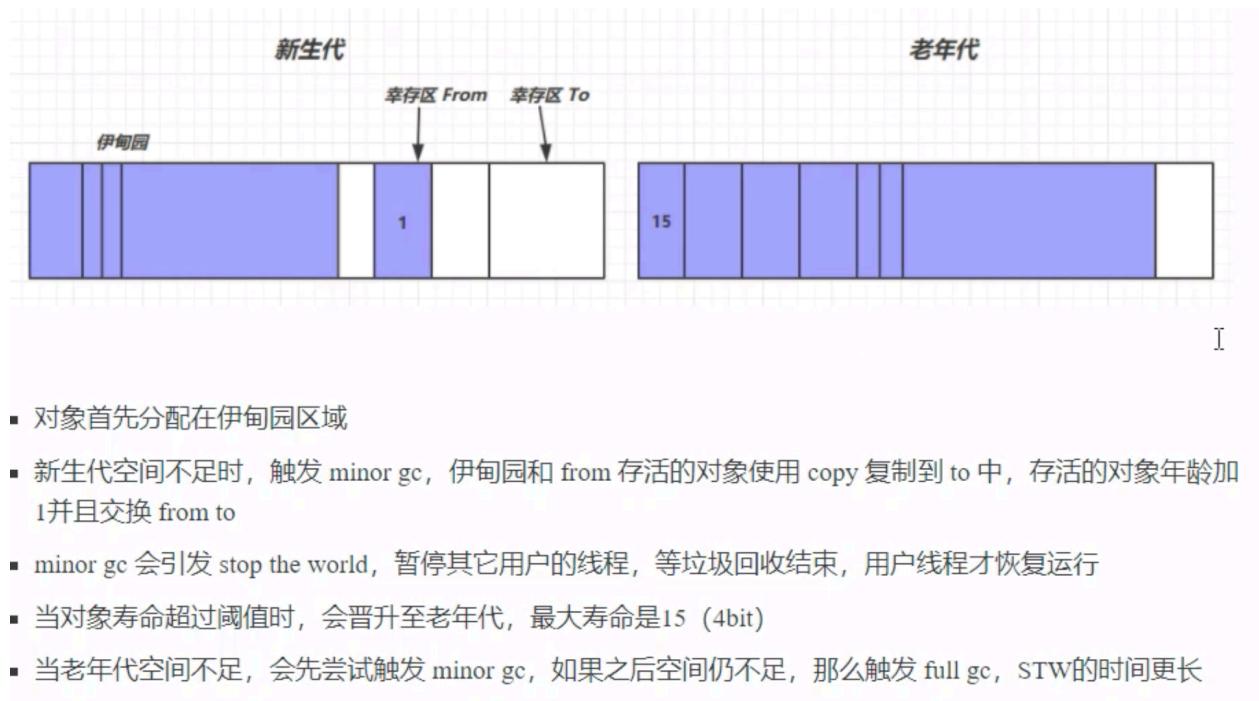
7. 直接内存

二：GC

(守护线程最典型的应用)

1. 分代回收算法

1.1 图示：

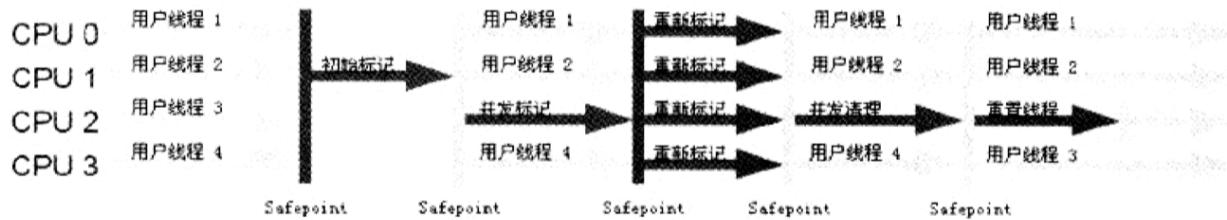


1.2 VM 参数

| 含义 | 参数 |
|------------------|---|
| 堆初始大小 | -Xms |
| 堆最大大小 | -Xmx 或 -XX:MaxHeapSize=size |
| 新生代大小 | -Xmn 或 (-XX:NewSize=size + -XX:MaxNewSize=size) |
| 幸存区比例 (动态) | -XX:InitialSurvivorRatio=ratio 和 -XX:+UseAdaptiveSizePolicy |
| 幸存区比例 | -XX:SurvivorRatio=ratio |
| 晋升阈值 | -XX:MaxTenuringThreshold=threshold |
| 晋升详情 | -XX:+PrintTenuringDistribution |
| GC详情 | -XX:+PrintGCDetails -verbose:gc |
| FullGC 前 MinorGC | -XX:+ScavengeBeforeFullGC |

2.垃圾回收器

2.1CMS垃圾回收器



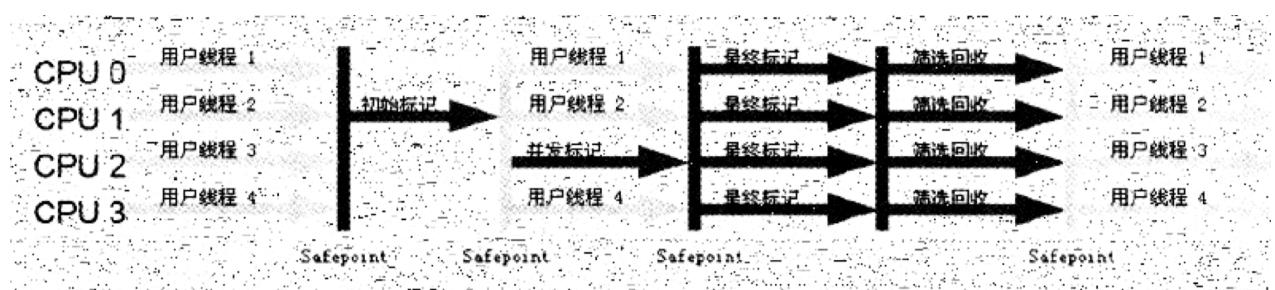
2.1.1缺点

- 1>总吞吐量降低
- 2>无法处理浮动垃圾
- 3>产生内部碎片（标记清除算法）
- 4>当碎片太多时会并发失败，退化为串行SerialOld

敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说 CPU 资源）而导致应用程序变慢，总吞吐量会降低。CMS 默认启动的回收线程数是 $(CPU\ 数量 + 3) / 4$ ，也就是当 CPU 在 4 个以上时，并发回收时垃圾收集线程不少于 25% 的 CPU 资源，并且随着 CPU 数量的增加而下降。但是当 CPU 不足 4

】CMS 收集器无法处理浮动垃圾（Floating Garbage），可能出现“Concurrent Mode Failure”失败而导致另一次 Full GC 的产生。由于 CMS 并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS 无法在当次收集中处理掉它们，只好留待下一次 GC 时再清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，那就还需要预留有足够的内存空间给用户线程使用，因此 CMS 收集器不能像其他收

2.2G1收集器



2.2.1使用的算法

整体上使用标记整理算法，两个Region之间使用复制算法

2.2.2回收阶段

young collection

young collection+cm

mixed collection

三：类加载与字节码技术

1.类文件结构

根据 JVM 规范，类文件结构如下

```
ClassFile {
    u4          magic;
    u2          minor_version;
    u2          major_version;
    u2          constant_pool_count;
    cp_info     constant_pool[constant_pool_count-1];
    u2          access_flags;  I
    u2          this_class;
    u2          super_class;
    u2          interfaces_count;
    u2          interfaces[interfaces_count];
    u2          fields_count;
    field_info   fields[fields_count];
    u2          methods_count;
    method_info  methods[methods_count];
    u2          attributes_count;
    attribute_info attributes[attributes_count];
}
```

access_flag: 修饰信息 (public abstract)

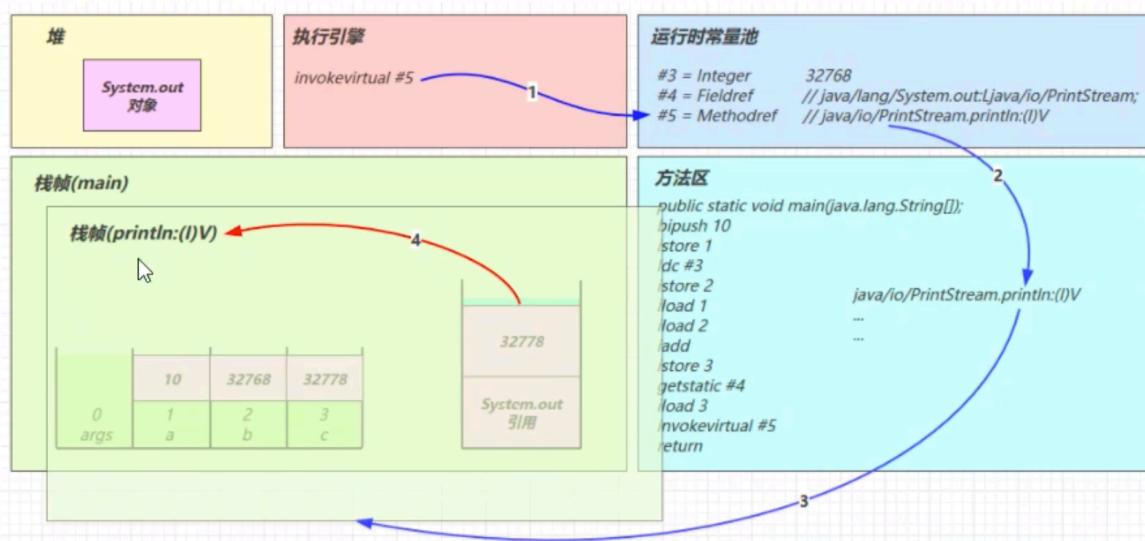
attribute_count: 附加的属性信息

2.字节码指令

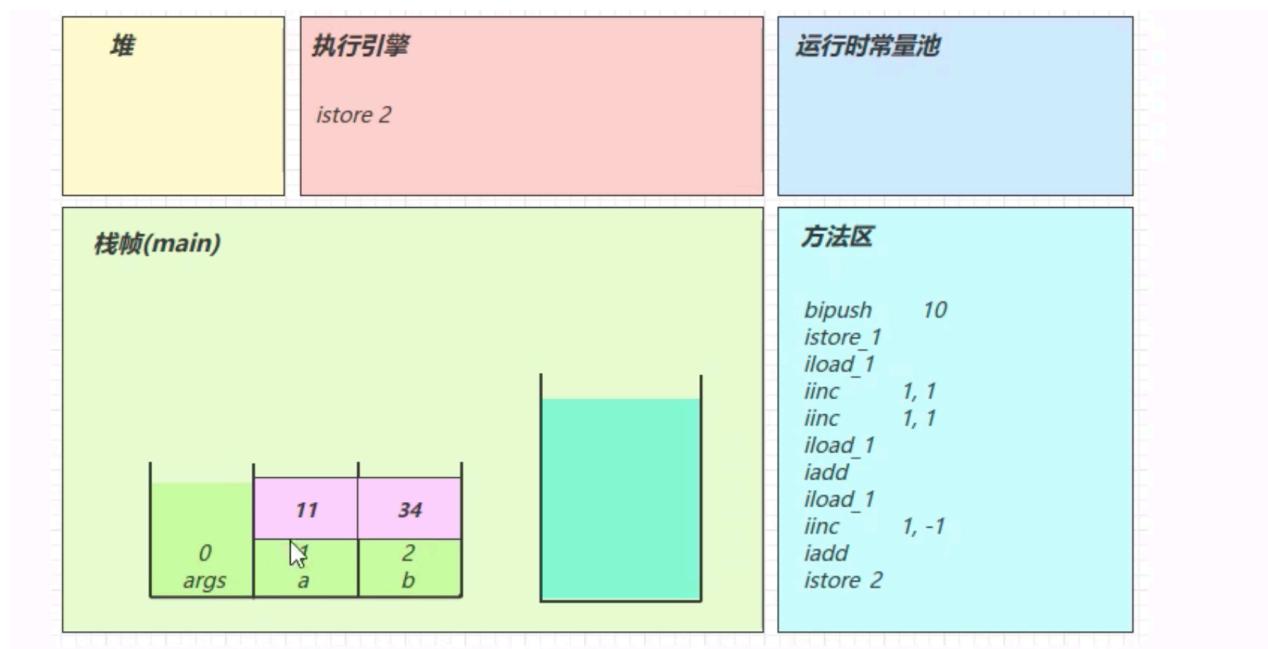
2.1 运行流程：

invokevirtual #5

- 找到常量池 #5 项
- 定位到方法区 `java/io/PrintStream.println:(I)V` 方法
- 生成新的栈帧 (分配 locals、stack 等)
- 传递参数，执行新栈帧中的字节码



`b=a++ + ++a + a--` 的执行流程 (a++ 和 ++a 的区别就是先 iload 还是先 iinc)



2.2 if判断指令的执行流程

```
public class Demo3_3 {
    public static void main(String[] args) {
        int a = 0;
        if(a == 0) {
            a = 10;
        } else {
            a = 20;
        }
    }
}
```

字节码:

```
0:  iconst_0
1:  istore_1
2:  iload_1
3:  ifne      12
6:  bipush    10
8:  istore_1
9:  goto     15
12: bipush    20
14: istore_1
15: return
```

2.3 循环控制指令

2.3.1 while

其实循环控制还是前面介绍的那些指令，例如 while 循环：

```
public class Demo3_4 {
    public static void main(String[] args) {
        int a = 0;
        while (a < 10) {
            a++;
        }
    }
}
```

字节码是：

```
0: iconst_0
1: istore_1
2: iload_1
3: bipush      10
5: if_icmpge   14
8: iinc        1, 1
11: goto       2
14: return
```

2.2.3 do-while

再比如 do while 循环：

```
public class Demo3_5 {
    public static void main(String[] args) {
        int a = 0;
        do {
            a++;
        } while (a < 10);
    }
}
```

字节码是：

```
0: iconst_0
1: istore_1
2: iinc        1, 1
5: iload_1
6: bipush      10
8: if_icmpgt   2
11: return
```

先做的iinc在做的比较

2.2.4 for

最后再看看 for 循环：

```
public class Demo3_6 {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {

        }
    }
}
```

字节码是：

```
0:  iconst_0
1:  istore_1
2:  iload_1
3:  bipush      10
5:  if_icmpge   14
8:  iinc        1, 1
11: goto       2
14: return
```

2.4 构造方法

2.4.1 ()V: 在类加载的初始化阶段被调用

```
public class Demo3_8_1 {

    static int i = 10;

    static {
        i = 20;
    }

    static {
        i = 30;
    }
}
```

编译器会按从上至下的顺序，收集所有 static 静态代码块和静态成员赋值的代码，合并为一个特殊的方法 <cinit>()V：

```
0: bipush      10
2: putstatic   #2                      // Field i:I
5: bipush      20
7: putstatic   #2                      // Field i:I
10: bipush     30
12: putstatic   #2                     // Field i:I
15: return
```

2.4.2 ()V: 在实例化对象时被调用

```
private String a = "s1";

{
    b = 20;
}

private int b = 10;

{
    a = "s2";
}

public Demo3_8_2(String a, int b) {
    this.a = a;
    this.b = b;
}

public static void main(String[] args) {
    Demo3_8_2 d = new Demo3_8_2("s3", 30);
    System.out.println(d.a);
    System.out.println(d.b);
}
```

编译器会按从上至下的顺序，收集所有 {} 代码块和成员变量赋值的代码，形成新的构造方法，但原始构造方法内的代码总是在最后

黑马程序员 -

Pc

```
public cn.itcast.jvm.t3.bytecode.Demo3_8_2(java.lang.String, int);
descriptor: (Ljava/lang/String;I)V
flags: ACC_PUBLIC
Code:
stack=2, locals=3, args_size=3
  0: aload_0
  1: invokespecial #1    // super.<init>()V
  4: aload_0
  5: ldc           #2    // <- "s1"
  7: putfield       #3    // -> this.a
 10: aload_0
 11: bipush        20   // <- 20
 13: putfield       #4    // -> this.b
 16: aload_0
 17: bipush        10   // <- 10
 19: putfield       #4    // -> this.b
 22: aload_0
 23: ldc           #5    // <- "s2"
 25: putfield       #3    // -> this.a
 28: aload_0
 29: aload_1
 30: putfield       #3    // -> this.a
 33: aload_0
 34: iload_2
 35: putfield       #4    // -> this.a
```

2.5方法调用

```

public class Demo3_9 {
    public Demo3_9() { }

    private void test1() { }

    private final void test2() { }

    public void test3() { }

    public static void test4() { }

    public static void main(String[] args) {
        Demo3_9 d = new Demo3_9();
        d.test1();
        d.test2();
        d.test3();
        d.test4();
        Demo3_9.test4();
    }
}

```

```

0: new           #2                  // class cn/itcast/jvm/t3/bytocode/Demo3_9
3: dup
4: invokespecial #3                // Method "<init>":()V
7: astore_1
8: aload_1
9: invokespecial #4 I             // Method test1:()V
12: aload_1
13: invokespecial #5                // Method test2:()V
16: aload_1
17: invokevirtual #6                // Method test3:()V
20: aload_1
21: pop
22: invokestatic #7                // Method test4:()V
25: invokestatic #7                // Method test4:()V
28: return

```

对象.静态方法会产生下面两条没必要的虚拟机指令(20,21):

aload 1

pop

2.6异常处理

2.6.1: 一个catch

```
public static void main(String []args){  
    int i=0;  
    try{  
        i=10;  
    }catch(Exception e){  
        i=20;  
    }  
}
```

```
Descriptor: <CL>java/lang/String, J  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=1, locals=3, args_size=1  
  0:  iconst_0  
  1:  istore_1  
  2:  bipush      10  
  4:  istore_1  
  5:  goto       12  
  8:  astore_2  
  9:  bipush      20  
 11:  istore_1  
 12:  return  
Exception table:  
  from   to target type I  
    2     5     8  Class java/lang/Exception  
LineNumberTable: ...  
LocalVariableTable:  
  Start  Length  Slot  Name   Signature  
    9       3     2    e   Ljava/lang/Exception;  
    0       13    0  args  [Ljava/lang/String;  
    2       11    1    i    I
```

2.6.2: 多个catch

```
0:  iconst_0
1:  istore_1
2:  bipush      10
4:  istore_1
5:  goto       26
8:  astore_2
9:  bipush      30
11: istore_1
12: goto       26
15: astore_2
16: bipush      40
18: istore_1
19: goto       26
22: astore_2
23: bipush      50
25: istore_1
26: return
```

Exception table:

| from | to | target | type |
|------|----|--------|--------------------------------------|
| 2 | 5 | 8 | Class java/lang/ArithmeticException |
| 2 | 5 | 15 | Class java/lang/NullPointerException |
| 2 | 5 | 22 | Class java/lang/Exception |

2.6.3 finally

```
public static void main(String []args){
    int i=0;
    try{
        i=10;
    }catch(Exception e){
        i=20;
    }finally{
        i=30;
    }
}
```

```

stack=1, locals=4, args_size=1
 0: iconst_0
 1: istore_1          // 0 -> i
 2: bipush      10   // try -----
 4: istore_1          // 10 -> i
 5: bipush      30   // finally |
 7: istore_1          // 30 -> i
 8: goto      27   // return -----
11: astore_2          // catch Exception -> e -----
12: bipush      20   // |
14: istore_1          // 20 -> i
15: bipush      30   // finally |
17: istore_1          // 30 -> i
18: goto      27   // return -----
21: astore_3          // catch any -> slot 3 -----
22: bipush      30   // finally |
24: istore_1          // 30 -> i
25: aload_3           // <- slot 3
26: athrow          // throw -----
27: return

Exception table:
  from    to  target type
    2      5    11  Class java/lang/Exception
    2      5    21  any    // 剩余的异常类型, 比如 Error
   11     15    21  any    // 剩余的异常类型, 比如 Error

LineNumberTable: ...
LocalVariableTable:

```

finally中的代码复制到try-catch和catch中不包括的异常类型中，保证其执行。
finally 中的return会吞掉athrow指令！

2.7finally面试题

```

public static void main(String []args){
    int i=0;
    try{
        i=10;
        return i;
    }finally{
        i=20;
    }
}
//此时输出返回值为10

```

```

0: bipush      10    // <- 10 放入栈顶
2: istore_0      // 10 -> i
3: iload_0       // <- i(10)
4: istore_1      // 10 -> slot 1, 暂存至 slot 1, 目的是为了固定返回值
5: bipush      20    // <- 20 放入栈顶
7: istore_0      // 20 -> i
8: iload_1       // <- slot 1(10) 载入 slot 1 暂存的值
9: ireturn      // 返回栈顶的 int(10)
10: astore_2
11: bipush      20
13: istore_0
14: aload_2
15: athrow      T

```

try中的代码执行完会弹出栈顶的原因是固定返回值，执行完finally后再加载到栈顶并返回

2.8synchronized

```

public static void main(String []args){
    Object obj=new Object();
    synchronized(obj){
        System.out.print("OK");
    }
}

```

```

0: new          #2    // new Object
3: dup
4: invokespecial #1   // invokespecial <init>:()V
7: astore_1
8: aload_1       // <- lock (synchronized开始)
9: dup
10: astore_2      // lock引用 -> slot 2
11: monitorenter
12: getstatic     #3   // <- System.out
15: ldc          #4   // <- "ok"
17: invokevirtual #5   // invokevirtual println:(Ljava/lang/String;)V
20: aload_2       // <- slot 2(lock引用)
21: monitorexit
22: goto          30
25: astore_3
26: aload_2       // <- slot 2(lock引用)
27: monitorexit
28: aload_3
29: athrow
30: return
Exception table:
  from   to   target type
    12    22    25  any
    25    28    25  any
LineNumberTable: ...
LocalVariableTable:

```

3.语法糖

4.类加载

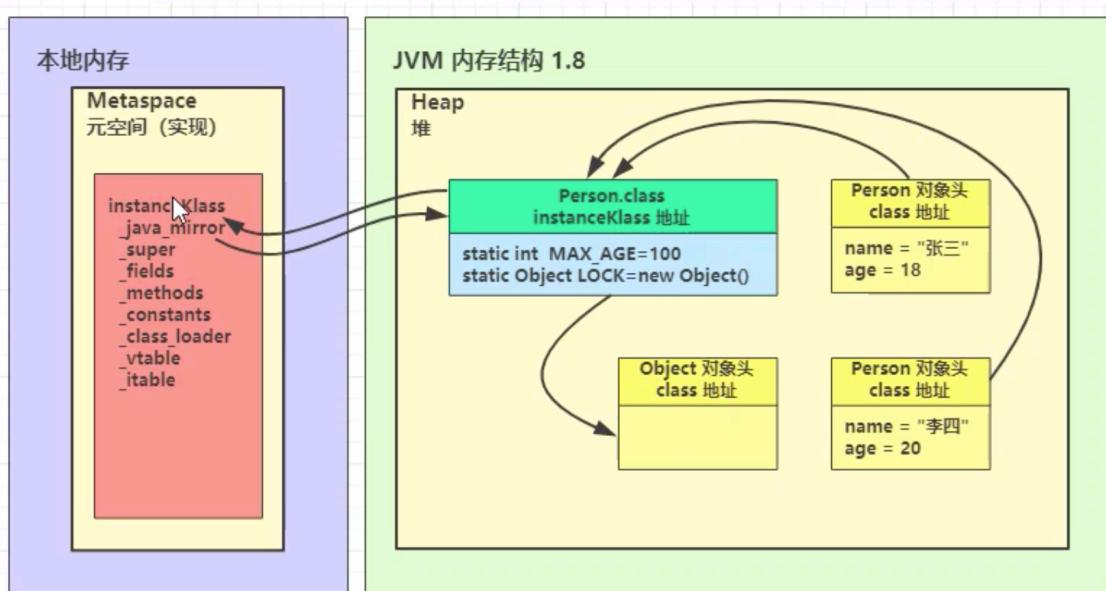
类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）7个阶段。其中验证、准备、解析3个部分统称为连接（Linking），这7个阶段的发生顺序如图 7-1 所示。



图 7-1 类的生命周期

4.1加载

- instanceKlass 这样的【元数据】是存储在方法区（1.8 后的元空间内），但 _java_mirror 是存储在堆中
- 可以通过前面介绍的 HSDB 工具查看



4.3 初始化

发生的时机

概括得说，类初始化是【懒惰的】

- main方法所在的类，总会被首先初始化
 - 首次访问这个类的静态变量或静态方法时
 - 子类初始化，如果父类还没初始化，会引发
 - 子类访问父类的静态变量，只会触发父类的初始化
 - Class.forName
 - new 会导致初始化

不会导致类初始化的情况

- 访问类的 static final 静态常量（基本类型和字符串）不会触发初始化
 - 类对象.class 不会触发初始化
 - 创建该类的数组不会触发初始化
 - 类加载器的 loadClass 方法
 - Class.forName 的参数 2 为 false 时

5. 类加载器

5.1 层级关系

| 名称 | 加载哪的类 | 说明 |
|-------------------------|-----------------------|-------------------------|
| Bootstrap ClassLoader | JAVA_HOME/jre/lib | 无法直接访问 |
| Extension ClassLoader | JAVA_HOME/jre/lib/ext | 上级为 Bootstrap, 显示为 null |
| Application ClassLoader | classpath | 上级为 Extension |
| 自定义类加载器 | 自定义 | 上级为 Application |

5.4 双亲委派

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
```

```

// 检查此类是否已被加载过
Class<?> c = findLoadedClass(name);
//如果没被加载
if (c == null) {
    long t0 = System.nanoTime();
    try {
        //如果有上级类加载器
        if (parent != null) {
            //看上级类加载器是否加载
            c = parent.loadClass(name, false);
        } else {
            //如果没有，看启动类加载器是否加载
            c = findBootstrapClassOrNull(name);
        }
    } catch (ClassNotFoundException e) {
        //捕捉异常，但是不做处理
    }
    //若启动类加载器仍不加载此类
    if (c == null) {

        long t1 = System.nanoTime();
        //在classpath下找到此类进行加载
        c = findClass(name);

        // this is the defining class loader; record the stats

sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);

sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
sun.misc.PerfCounter.getFindClasses().increment();
    }
}
if (resolve) {
    resolveClass(c);
}
//加载完成后返回Class
return c;
}
}

```

5.5自定义类加载器

包名，类名，类加载器都一致的类才是同一个类

问问自己，什么时候需要自定义类加载器

- 1) 想加载非 classpath 随意路径中的类文件
- 2) 都是通过接口来使用实现，希望解耦时，常用在框架设计
- 3) 这些类希望予以隔离，不同应用的同名类都可以加载，不冲突，常见于 tomcat 容器

I

步骤：

1. 继承 ClassLoader 父类
2. 要遵从双亲委派机制，重写 findClass 方法
 - 注意不是重写 loadClass 方法，否则不会走双亲委派机制
3. 读取类文件的字节码
4. 调用父类的 defineClass 方法来加载类
5. 使用者调用该类加载器的 loadClass 方法

```
class MyClassLoader extends ClassLoader {

    @Override // name 就是类名称
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        String path = "e:\\\\myclasspath\\\\" + name + ".class";

        try {
            ByteArrayOutputStream os = new ByteArrayOutputStream();
            Files.copy(Paths.get(path), os);

            // 得到字节数组
            byte[] bytes = os.toByteArray();

            // byte[] -> *.class
            return defineClass(name, bytes, 0, bytes.length);
        } catch (IOException e) {
            e.printStackTrace();
            throw new ClassNotFoundException("类文件未找到", e);
        }
    }
}
```

6.运行期优化

6.1即时编译

6.1.1分层编译



6.1.2方法内联

方法内联

(Inlining)

```
private static int square(final int i) {  
    return i * i;  
}
```

java

```
System.out.println(square(9));
```

java

如果发现 square 是热点方法，并且长度不太长时，会进行内联，所谓的内联就是把方法内代码拷贝、粘贴到调用者的位置：

```
System.out.println(9 * 9);
```

java

还能够进行常量折叠 (constant folding) 的优化

```
System.out.println(81);
```

6.2.3字段优化

6.2反射优化

control+f4查看实现类

```
public Object invoke(Object var1, Object[] var2) throws  
IllegalArgumentException, InvocationTargetException {  
    if (++this.numInvocations > ReflectionFactory.inflationThreshold() &&  
        !ReflectUtil.isVMAnonymousClass(this.method.getDeclaringClass()))  
    {  
        MethodAccessorImpl var3 = (MethodAccessorImpl)(new  
MethodAccessorGenerator()).generateMethod(this.method.getDeclaringClass(),  
this.method.getName(), this.method.getParameterTypes(),  
this.method.getReturnType(), this.method.getExceptionTypes(),  
this.method.getModifiers());  
        this.parent.setDelegate(var3);  
    }  
  
    return invoke0(this.method, var1, var2);  
}
```

当反射生成相同类超过阈值（默认15次）时，会动态生成MethodAccessorGenerator类优化生成速度

四：Java内存模型

Java 虚拟机规范中试图定义一种 Java 内存模型^① (Java Memory Model, JMM) 来屏蔽

掉各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果。在此之前，主流程序语言（如 C/C++ 等）直接使用物理硬件和操作系统的

Java 内存模型规定了所有的变量都存储在主内存 (Main Memory) 中 (此处的主内存与介绍物理硬件时的主内存名字一样，两者也可以互相类比，但此处仅是虚拟机内存的一部分)。每条线程还有自己的工作内存 (Working Memory，可与前面讲的处理器高速缓存类比)，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝^②；线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量^③。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成，线程、主内存、工作内存三者的交互关系如图 12-2 所示。

1. 原子性 可见性 有序性

原子性：由 JMM 来直接保证的原子性变量操作包括 `read`、`load`、`assign`、`use`、`store`、和 `write`，大致可以认为基本数据类型的访问读写是具备原子性的

可见性：当一个线程修改了共享变量的值，其他线程可以立即得知这个修改

有序性：在本线程内观察，所有线程都是有序的，从一个线程观察另一个线程，所有线程都是无序的。
(前半句指‘线程内表现为串行的语义’，后半句指‘指令重排和主内存和工作内存同步延迟’)

2. 先行发生原则(happen-before)

现在就来看看“先行发生”原则指的是什么。先行发生是 Java 内存模型中定义的两项操作之间的偏序关系，如果说操作 A 先行发生于操作 B，其实就是说在发生操作 B 之前，操作 A 产生的影响能被操作 B 观察到，“影响”包括修改了内存中共享变量的值、发送了消息、

- 程序次序规则 (Program Order Rule)：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说，应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。
- 管程锁定规则 (Monitor Lock Rule)：一个 unlock 操作先行发生于后面对同一个锁的 lock 操作。这里必须强调的是同一个锁，而“后面”是指时间上的先后顺序。
- volatile 变量规则 (Volatile Variable Rule)：对一个 volatile 变量的写操作先行发生于后面对这个变量的读操作，这里的“后面”同样是指时间上的先后顺序。
- 线程启动规则 (Thread Start Rule)：Thread 对象的 start() 方法先行发生于此线程的每一个动作。
- 线程终止规则 (Thread Termination Rule)：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过 Thread.join() 方法结束、Thread.isAlive() 的返回值等手段检测到线程已经终止执行。
- 线程中断规则 (Thread Interruption Rule)：对线程 interrupt() 方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过 Thread.interrupted() 方法检测到是否有中断发生。
- 对象终结规则 (Finalizer Rule)：一个对象的初始化完成（构造函数执行结束）先行发生于它的 finalize() 方法的开始。
- 传递性 (Transitivity)：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那就可以得出操作 A 先行发生于操作 C 的结论。

3.线程状态

Java 语言定义了 5 种线程状态，在任意一个时间点，一个线程只能有且只有其中的一种状态，这 5 种状态分别如下。

- 新建 (New)：创建后尚未启动的线程处于这种状态。
- 运行 (Runnable)：Runnable 包括了操作系统线程状态中的 Running 和 Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着 CPU 为它分配执行时间。
- 无限期等待 (Waiting)：处于这种状态的线程不会被分配 CPU 执行时间，它们要等待被其他线程显式地唤醒。以下方法会让线程陷入无限期的等待状态：
 - 没有设置 Timeout 参数的 Object.wait() 方法。
 - 没有设置 Timeout 参数的 Thread.join() 方法。
 - LockSupport.park() 方法。

□ 限期等待（Timed Waiting）：处于这种状态的线程也不会被分配 CPU 执行时间，不过无须等待被其他线程显式地唤醒，在一定时间之后它们会由系统自动唤醒。以下方法会让线程进入限期等待状态：

- Thread.sleep() 方法。
- 设置了 Timeout 参数的 Object.wait() 方法。
- 设置了 Timeout 参数的 Thread.join() 方法。
- LockSupport.parkNanos() 方法。
- LockSupport.parkUntil() 方法。

□ 阻塞（Blocked）：线程被阻塞了，“阻塞状态”与“等待状态”的区别是：“阻塞状态”在等待着获取到一个排他锁，这个事件将在另外一个线程放弃这个锁的时候发生；而“等待状态”则是在等待一段时间，或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。

□ 结束（Terminated）：已经终止线程的线程状态，线程已经结束执行。

上述 5 种状态在遇到特定事件发生的时候将会互相转换，它们的转换关系如图 12-6 所示。

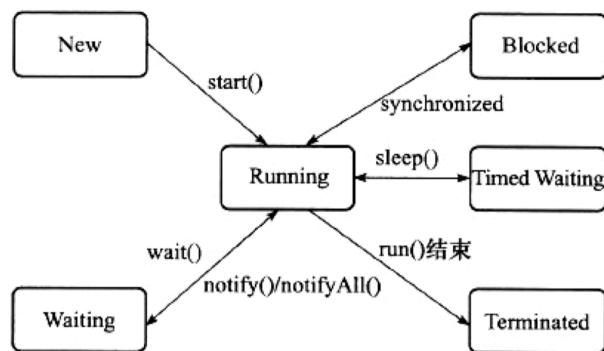


图 12-6 线程状态转换关系

图：

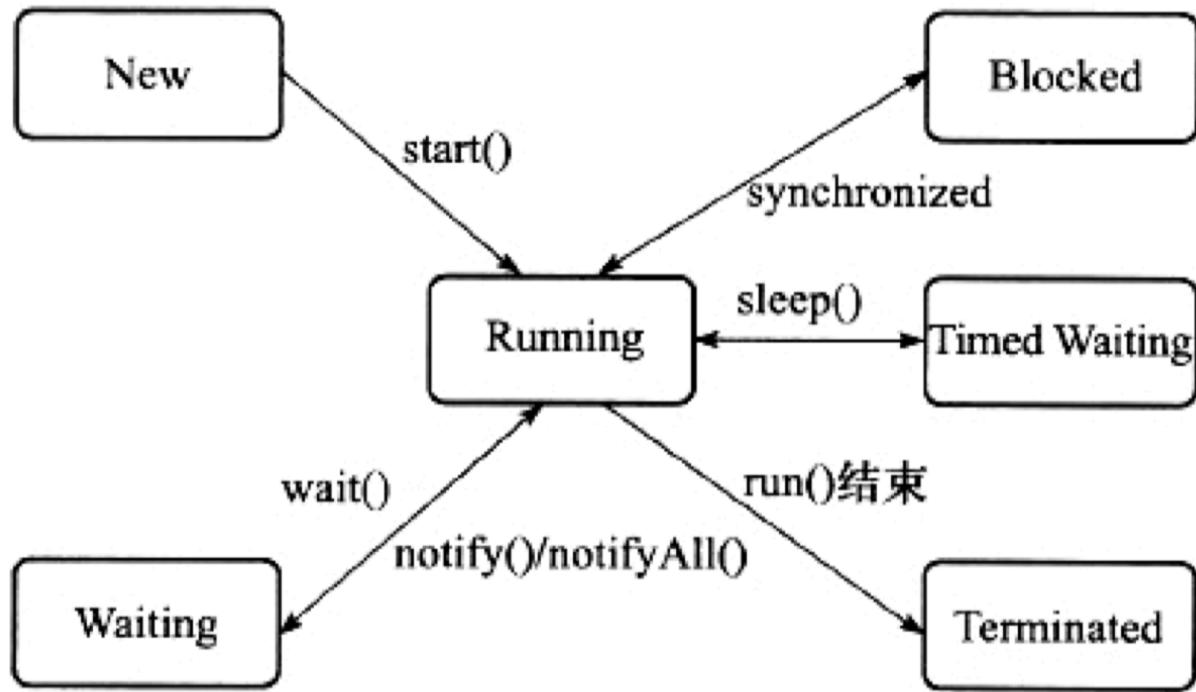


图 12-6 线程状态转换关系

4. 乐观锁与悲观锁

- CAS 是基于乐观锁的思想：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
- synchronized 是基于悲观锁的思想：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。

CAS 指令需要有 3 个操作数，分别是内存位置（在 Java 中可以简单理解为变量的内存地址，用 V 表示）、旧的预期值（用 A 表示）和新值（用 B 表示）。CAS 指令执行时，当且仅当 V 符合旧预期值 A 时，处理器用新值 B 更新 V 的值；否则它就不执行更新，但是无论是否更新了 V 的值，都会返回 V 的旧值。上述的处理过程是一个原子操作。

CAS 的全称为 Compare-And-Swap，它是一条 CPU 并发原语。

它的功能是判断内存某个位置的值是否为预期值，如果是则更改为新的值，这个过程是原子的。

CAS 并发原语体现在 JAVA 语言中就是 sun.misc.Unsafe 类中的各个方法。调用 Unsafe 类中的 CAS 方法，JVM 会帮我们实现 CAS 汇编指令。这是一种完全依赖于硬件的功能，通过它实现了原子操作。再次强调，由于 CAS 是一种系统原语，原语属于操作系统语义范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断，也就是说 CAS 是一条 CPU 的原子指令，不会造成所谓的数据不一致问题。

4.1 unsafe

1 Unsafe

是 CAS 的核心类，由于 Java 方法无法直接访问底层系统，需要通过本地（native）方法来访问，Unsafe 相当于一个后门，基于该类可以直接操作特定内存函数。Unsafe 类存在于 sun.misc 包中，其内部方法操作可以像 C 的指针一样直接操作内存，因为 Java 中 CAS 操作的执行依赖于 Unsafe 类的方法。

注意 Unsafe 类中的所有方法都是 native 修饰的，也就是说 Unsafe 类中的方法都直接调用操作系统底层资源执行相应任务

2 变量valueOffset，表示该变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的。

```
/*
 * Atomically increments by one the current value.
 *
 * @return the previous value
 */
public final int getAndIncrement() {
    return unsafe.getAndAddInt(0, this, valueOffset, 1);
}
```

3 变量value用volatile修饰，保证了多线程之间的内存可见性。

5. 锁优化

1. 锁消除

对于被检测出不可能存在竞争的共享数据的锁进行消除

2. 锁粗化

如果一系列的连续操作都对同一个对象反复加锁和解锁，频繁的加锁操作就会导致性能损耗。

对同一个对象只加一次锁

3. 自旋锁

线程访问已经被锁定的资源时，先自旋一段时间，若成功则获取资源，不成功则阻塞

4. 偏向锁

线程偏向于让已经获得过的资源的线程得到锁

5. 轻量级锁

四个状态

- 无锁状态
- 偏向锁状态
- 轻量级锁状态
- 重量级锁状态

附1：juc面试

1. 公平锁与非公平锁

公平锁 是指多个线程按照申请锁的顺序来获取锁，类似排队打饭，先来后到。

非公平锁 是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁
在高并发的情况下，有可能会造成优先级反转或者饥饿现象

1.1 二者区别

公平锁 / 非公平锁

并发包中ReentrantLock的创建可以指定构造函数的boolean类型来得到公平锁或非公平锁，默认是非公平锁

关于两者区别：

公平锁： Threads acquire a fair lock in the order in which they requested it

公平锁，就是很公平，在并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空，或者当前线程是等待队列的第一个，就占有锁，否则就会加入到等待队列中，以后会按照FIFO的规则从队列中取到自己

非公平锁： a nonfair lock permits barging: threads requesting a lock can jump ahead of the queue of waiting threads if the lock happens to be available when it is requested.

非公平锁比较粗鲁，上来就直接尝试占有锁，如果尝试失败，就再采用类似公平锁那种方式。

非公平锁的优点在于吞吐量比公平锁大

ReentrantLock 默认非公平锁 synchronized也是非公平锁

2. 可重入锁（递归锁）

可重入锁（也叫做递归锁）

指的是同一线程外层函数获得锁之后，内层递归函数仍然能获取该锁的代码，在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁

也即是说，线程可以进入任何一个它已经拥有的锁所同步着的代码块。

最大的作用：避免死锁

ReentrantLock和synchronized都是可重入锁

3. 自旋锁

自旋锁（spinlock）

是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU

3.1 手写自旋锁

```
class SpinLock{
    private AtomicReference<Thread> ar=new AtomicReference<>();
    public void myLock(){
        System.out.println("come in lock");
        Thread thread=Thread.currentThread();
        while(!ar.compareAndSet(null,thread)){
            }
        System.out.println(thread.getName()+"获得锁");
    }
}
```

```
}

public void myUnlock(){
    Thread thread=Thread.currentThread();
    ar.compareAndSet(thread,null);
    System.out.println(thread.getName()+"解锁");
}

}
```

4.读写锁

独占锁，共享锁，互斥锁

独占锁：指该锁一次只能被一个线程所持有。对ReentrantLock和Synchronized而言都是独占锁

共享锁：指该锁可被多个线程所持有。

对ReentrantReadWriteLock其读锁是共享锁，其写锁是独占锁。

读锁的共享锁可保证并发读是非常高效的，读写，写读，写写的过程是互斥的。

5.线程池

5.1线程池的优势，为什么使用

线程池做的工作主要是控制运行的线程的数量，**处理过程中将任务放入队列**，然后在线程创建后启动这些任务，**如果线程数量超过了最大数量超出数量的线程排队等候**，等其它线程执行完毕，再从队列中取出任务来执行。

他的主要特点为：**线程复用；控制最大并发数；管理线程。**

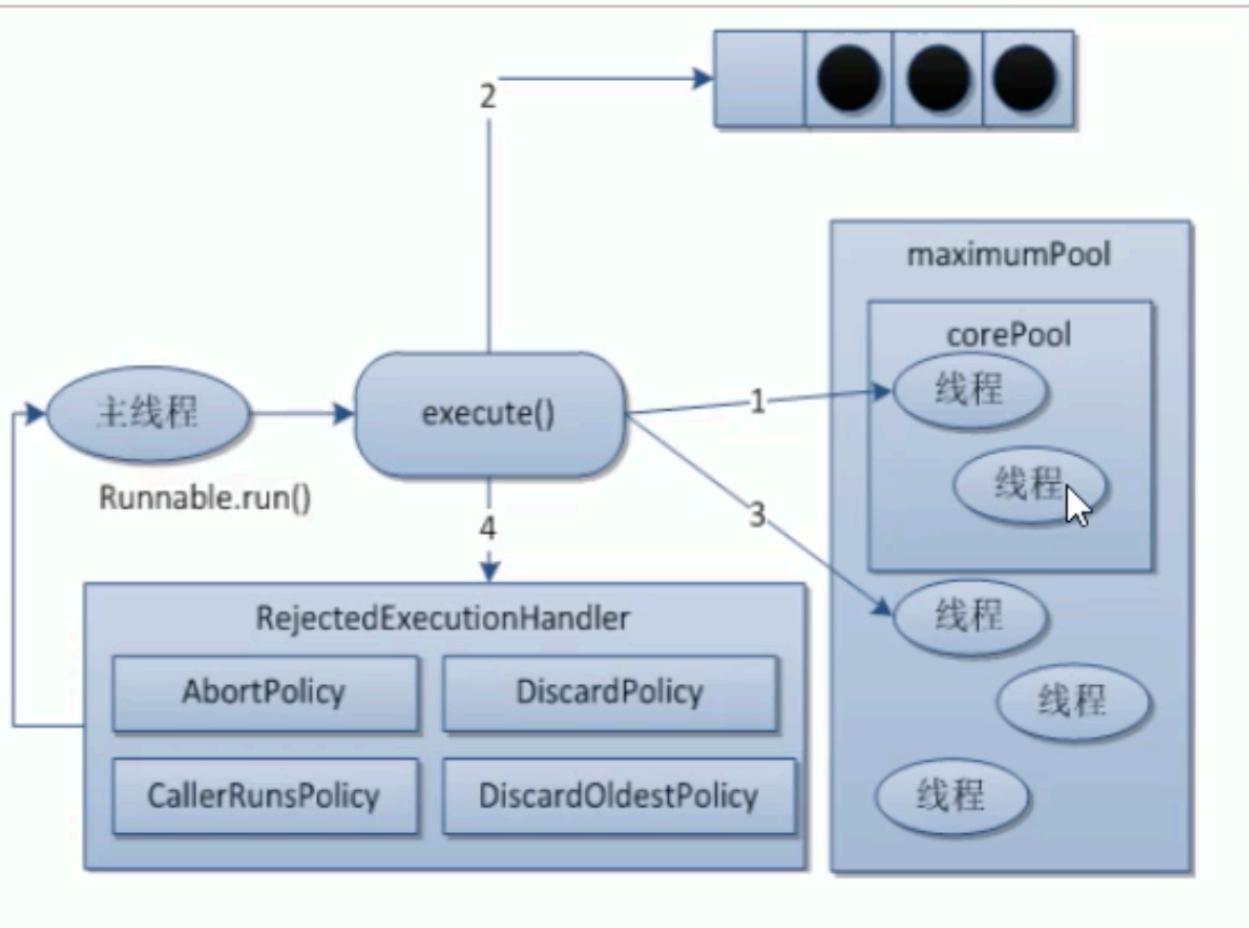
I

第一：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。

第二：提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。

第三：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。

5.2线程池的工作流程



1. 在创建了线程池后，等待提交过来的任务请求。
2. 当调用 `execute()` 方法添加一个请求任务时，线程池会做如下判断：
 - 2.1 如果正在运行的线程数量小于 `corePoolSize`, 那么马上创建线程运行这个任务;
 - 2.2 如果正在运行的线程数量大于或等于 `corePoolSize`, 那么将这个任务放入队列;
 - 2.3 如果这时候队列满了且正在运行的线程数量还小于 `maximumPoolSize`, 那么还是要创建非核心线程立刻运行这个任务;
 - 2.4 如果队列满了且正在运行的线程数量大于或等于 `maximumPoolSize`, 那么线程池会启动饱和拒绝策略来执行。
3. 当一个线程完成任务时，它会从队列中取下一个任务来执行。
4. 当一个线程无事可做超过一定的时间 (`keepAliveTime`) 时，线程池会判断：

```

int c = ctl.get();
//如果线程数小于核心线程数
if (workerCountOf(c) < corePoolSize) {
    //运行线程
    if (addWorker(command, true))
        return;
    c = ctl.get();
}
//大于核心线程数，加入队列
if (isRunning(c) && workQueue.offer(command)) {
    int recheck = ctl.get();
    if (! isRunning(recheck) && remove(command))
        //执行拒绝策略
        reject(command);
    else if (workerCountOf(recheck) == 0)
        addWorker(null, false);
}

```

```
    }
    else if (!addWorker(command, false))
        reject(command);
}
```

5.3四大拒绝策略

AbortPolicy(默认): 直接抛出 RejectedExecutionException 异常阻止系统正常运行。

CallerRunsPolicy: "调用者运行"一种调节机制, 该策略既不会抛弃任务, 也不会抛出异常, 而是将某些任务回退到调用者线程上执行。

DiscardOldestPolicy: 抛弃队列中等待最久的任务, 然后把当前任务加入队列中尝试再次提交当前任务。

DiscardPolicy: 直接丢弃任务, 不予任何处理也不抛出异常。如果允许任务丢失, 这是最好的一种方案。

5.4使用何种方式创建线程池, 为什么

4. 【强制】线程池不允许使用 Executors 去创建, 而是通过 ThreadPoolExecutor 的方式, 这样的处理方式让写的同学更加明确线程池的运行规则, 规避资源耗尽的风险。

说明: Executors 返回的线程池对象的弊端如下:

1) FixedThreadPool 和 SingleThreadPool:

允许的请求队列长度为 Integer.MAX_VALUE, 可能会堆积大量的请求, 从而导致 OOM。

2) CachedThreadPool 和 ScheduledThreadPool:

允许的创建线程数量为 Integer.MAX_VALUE, 可能会创建大量的线程, 从而导致 OOM。

5.5线程池关闭

我们知道, 使用 shutdownNow 方法, 可能会引起报错, 使用 shutdown 方法可能会导致线程关闭不了。

所以当我们使用 shutdownNow 方法关闭线程池时, 一定要对任务里进行异常捕获。

当我们使用 shutdown 方法关闭线程池时, 一定要确保任务里不会有永久阻塞等待的逻辑, 否则线程池就关闭不了

interrupt():如果有阻塞, 则抛出异常, 无阻塞, 则标志一个中断位

isInterrupted():判断标志位, 若标记过则为 true

interrupted():判断标志位, 若标记过则为 true, 并清除标志位, 连续判断下次则为 false

6.手写死锁及其排查

6.1手写死锁

```
class DeadLock implements Runnable{

    String lock1;
    String lock2;

    public DeadLock(String lock1, String lock2) {
        this.lock1 = lock1;
        this.lock2 = lock2;
    }

    @Override
```

```

public void run() {
    synchronized (lock1){
        System.out.println("已持有"+lock1+",想获取"+lock2);
        try {
            TimeUnit.SECONDS.sleep(2); }catch(InterruptedException e){
e.printStackTrace();}
        synchronized (lock2){
            System.out.println("已持有"+lock2+",想获取"+lock1);
        }
    }
}

public class TestDeadLock {
    public static void main(String[] args) {
        String s1="lock1";
        String s2="lock2";
        DeadLock deadLock=new DeadLock(s1,s2);
        DeadLock deadLock2=new DeadLock(s2,s1);
        new Thread(deadLock).start();
        new Thread(deadLock2).start();
    }
}

```

6.2排查

6.2.1: jps找到进程id

```

yinwenbodeMacBook-Pro:intetview yinwenbo$ jps
627 Jps
597 RemoteMavenServer
616 KotlinCompileDaemon
586
622 Launcher
623 TestDeadLock
yinwenbodeMacBook-Pro:intetview yinwenbo$ jstack 623

```

6.2.2: jstack排查问题

```

Java stack information for the threads listed above:
=====
"Thread-1":
    at interview.deadlock.DeadLock.run(TestDeadLock.java:24)
    - waiting to lock <0x0000000795789090> (a java.lang.String)
    - locked <0x00000007957890c8> (a java.lang.String)
    at java.lang.Thread.run(Thread.java:745)
"Thread-0":
    at interview.deadlock.DeadLock.run(TestDeadLock.java:24)
    - waiting to lock <0x00000007957890c8> (a java.lang.String)
    - locked <0x0000000795789090> (a java.lang.String)
    at java.lang.Thread.run(Thread.java:745)

```

Found 1 deadlock.

7.线程安全的单例模式

double check

```

//volatile 禁止指令重排
/*
由于singletonTest = new SingletonTest()操作并不是一个原子性指令，会被分为多个指令：

memory = allocate(); //1: 分配对象的内存空间
ctorInstance(memory); //2: 初始化对象
instance = memory; //3: 设置instance指向刚分配的内存地址
但是经过重排序后如下：

memory = allocate(); //1: 分配对象的内存空间
instance = memory; //3: 设置instance指向刚分配的内存地址，此时对象还没被初始化
ctorInstance(memory); //2: 初始化对象
若有A线程进行完重排后的第二步，且未执行初始化对象。此时B线程此时B线程来取
singletonTest时，
发现singletonTest不为空，于是便返回该值，但由于没有初始化完该对象，此时返回的对象是
有问题的。
上述代码的改进方法：将singletonTest声明为volatile类型即可（volatile有内存屏障的
功能）。
*/
public class Singleton{
    private static volatile Singleton singleton;
    private Singleton(){}
    public static Singleton getInstance(){
        if(singleton==null){
            synchronized(Singleton.class){
                if(singleton==null){
                    singleton=new Singleton();
                }
            }
        }
    }
}

```

```
    }
    return singleton;
}

}
```

饿汉式

```
public class Singleton{
    private static Singleton INSTANCE=new Singleton();
    private Singleton(){}
    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

8.线程交替打印

1.wait、notify

```
public class AlterPrint implements Runnable{
    private int i=1;
    public void run(){
        while(true){
            synchronized(this){
                notify();
                if(i<=100){
                    System.out.println(Thread.currentThread().getName() + ":" + i);
                    i++;
                    try{
                        wait();
                    }catch(Exception e){
                        e.printStackTrace();
                    }
                }else break;
            }
        }
    }
}
```

2.await、signal

```
class TestNewComm{
    private int i=1;
    private Lock lock=new ReentrantLock();
    Condition c1=lock.newCondition();
    Condition c2=lock.newCondition();
    public void print1() {
```

```
lock.lock();
try {
    while ((i & 1) == 0) {
        try {
            c1.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println("1:" + i);
    i++;
    c2.signal();
}finally {
    lock.unlock();
}
}

public void print2() {
    lock.lock();
    try {
        while ((i & 1) == 1) {
            try {
                c2.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("2:" + i);
        i++;
        c1.signal();
    }finally {
        lock.unlock();
    }
}
}

public static void main(String[] args) {

    TestNewComm t=new TestNewComm();
    new Thread(()->{
        for (int i = 0; i <5; i++)
            t.print1();
    }, "1").start();
    new Thread(()->{
        for (int i = 0; i <5; i++)
            t.print2();
    }, "2").start();
}
```

附2: jvm面试

1.类加载器

1.1什么是类加载器

Java类加载器是Java运行时环境的一部分，负责动态加载Java类到Java虚拟机的内存空间中。在经过Java编译器编译之后就被转换成Java的.class文件。类加载器负责读取Java字节代码，并转换成`java.lang.Class`类的一个实例。

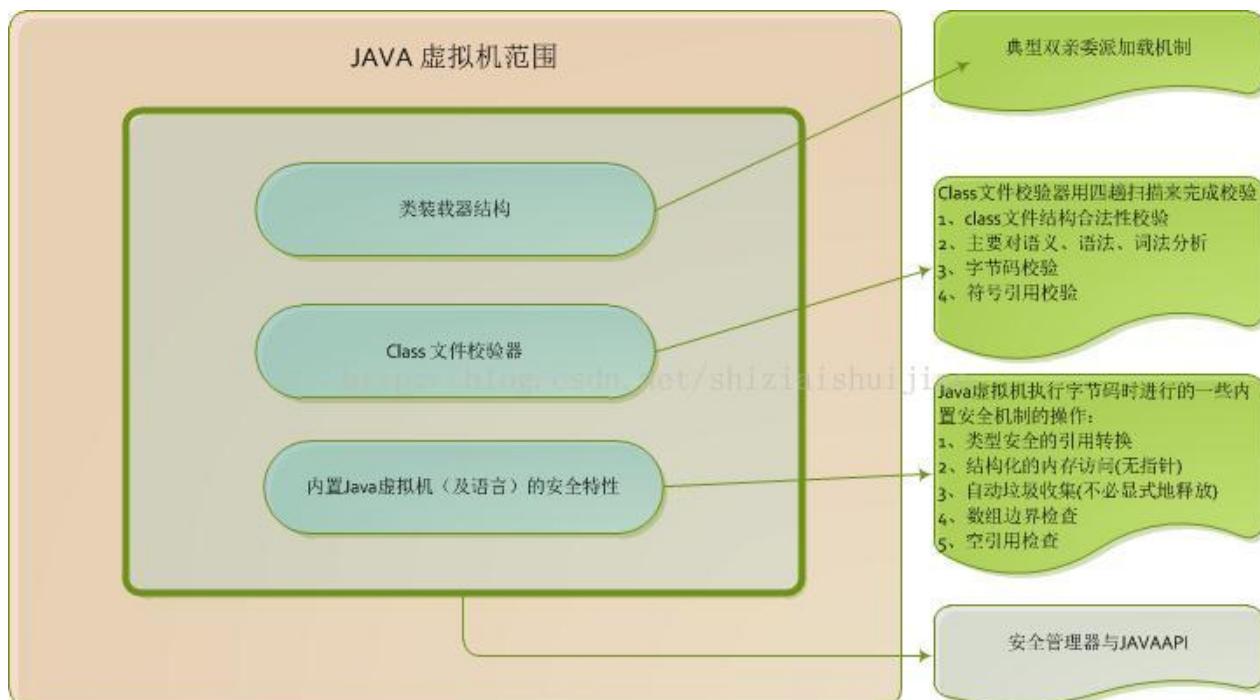
1.2类加载器种类

1.3双亲委派

当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载器中，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的Class），子类加载器才会尝试自己去加载。

采用双亲委派的一个好处是比如加载位于rt.jar包中的类`java.lang.Object`，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载器最终得到的都是同样一个Object对象。

1.4沙箱安全模型



2.GC Root

2.1哪些对象可以回收

在GC Root Set中以gc root为起点进行链路访问，如果对象可达，则不被回收，如果引用不可达，则可以回收

2.2哪些对象可以作为GC Root

- 1.虚拟机栈（局部变量表）中引用的对象
- 2.方法区中的类静态属性引用的对象
- 3.方法区中常量引用的对象
- 4.本地方法栈中引用的对象

3.jvm参数

3.1.参数类型

3.1.1 标配参数

-version
-help

3.2.2 X参数（了解）

-Xint (解释执行)
-Xcomp (编译执行)
-Xmixed (混合执行)

3.2.3 XX参数

1.boolean类型

-XX: +或者- 某个属性值
+表示开启-表示关闭

例：

-XX: +PrintGCDetails (是否打印GC收集细节)
-XX: -UseSerialGC (是否打印GC收集细节)

2.KV设值类型

-XX: 属性key=属性值value

例：

-XX:MetaspaceSize=128m (设置元空间大小为128m)

3.jinfo

jinfo -flag 配置项 进程编号（查看当前运行程序的配置）

4.-Xms和-Xmx

-Xms等价于 -XX:InitialHeapSize
-Xms等价于 -XX:MaxHeapSize

3.2查看JVM默认值

1.java -XX:+PrintFlagsInitial (查看初始默认值)

2.java -XX:+PrintFlagsFinal (主要查看修改更新)

3.java -XX:+PrintCommandLineFlags

4.常用参数

1.-Xms

初始大小内存， 默认为物理内存1/64

等价于-XX:InitialHeapSize

2.-Xmx

最大分配内存， 默认为物理内存1/4

等价于-XX:MaxHeapSize

3.-Xss

设置单个线程的大小， 一般默认为512K~1024K

等价于-XX:ThreadStackSize

4.-Xmn

设置年轻代大小

5.-XX:MetaspaceSize

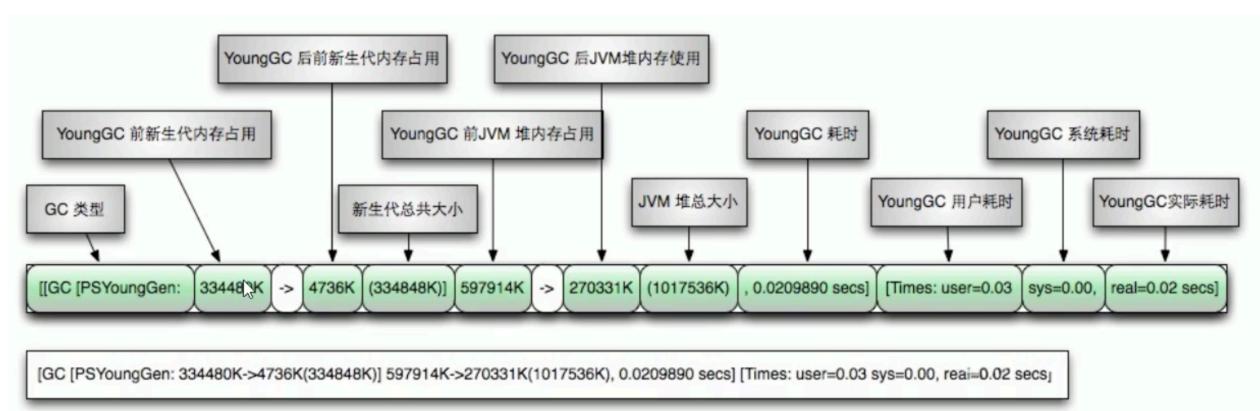
设置元空间大小

-Xms10m -Xmx10m -XX:MetaspaceSize=1024m -XX:+PrintFlagsFinal

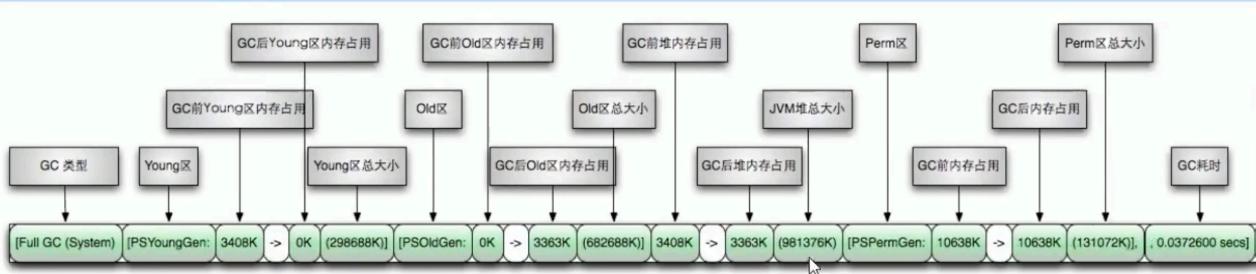
6.-XX:+PrintGCDetails

输出详细GC收集日志信息

MINIOR GC:



FULL GC:



7.-XX:SurvivorRatio

设置新生代中eden和s0/s1空间的比例
默认

-XX:SurvivorRatio=8, Eden:S0:S1 =8:1:1

假如

-XX:SurvivorRatio=4, Eden:S0:S1 =4:1:1

SurvivorRatio值就是设置eden区的比例占多少， s0/s1相同

8.-XX:NewRatio

配置年轻代与老年代在堆结构的占比

默认

-XX:NewRatio=2新生代占1,老年代2, 年轻代占整个堆的1/3

假如

-XX:NewRatio=4新生代占1,老年代4, 年轻代占整个堆的1/5

NewRatio值就是设置老年代的占比, 剩下的1给新生代

9.-XX:MaxTenuringThreshold

设置垃圾最大年龄 (必须为0-15)

-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, 直接进入老年代。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率。

5.java.lang.ref

1.强引用

直接new的对象, 只要有引用, 就不会被GC

2.软引用

SoftReference, GC && 空间不足时就会被回收

3.弱引用

WeakReference, 只要GC就会被回收

/*WeakHashMap的使用 (用于缓存)

```

当已经put的键值对的key被置为null时,
下次gc会回收次键值对

*/
public static void main(String[] args) {

    WeakHashMap<Integer, String> map=new WeakHashMap<>();

    Integer i=new Integer(1);
    String name="weakhashmap";

    map.put(i, name);
    System.out.println(map); // {1=weakhashmap}

    i=null;
    System.out.println(map); // {1=weakhashmap}

    System.gc();

    System.out.println(map+" "+i); // {} null

}

```

4.软弱的应用场景

假如有一个应用需要读取大量的本地图片：

- * 如果每次读取图片都从硬盘读取则会严重影响性能，
- * 如果一次性全部加载到内存中又可能造成内存溢出。

此时使用软引用可以解决这个问题。|

设计思路是：用一个**HashMap**来保存图片的路径和相应图片对象关联的软引用之间的映射关系，在内存不足时，JVM会自动回收这些缓存图片对象所占用的空间，从而有效地避免了**OOM**的问题。

```
Map<String, SoftReference<Bitmap>> imageCache = new HashMap<String, SoftReference<Bitmap>>();
```

5.虚引用

PhantomReference，随时会被回收，必须配合引用队列使用，被回收时会被添加到引用队列中，允许被回收的引用做一些后续的通知

*java提供了4种引用类型，在垃圾回收的时候，都有自己各自的特点。
ReferenceQueue是用来配合引用工作的，没有ReferenceQueue一样可以运行。*

*创建引用的时候可以指定关联的队列，当GC释放对象内存的时候，会将引用加入到引用队列，
如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动
这相当于是一种通知机制。*

*当关联的引用队列中有数据的时候，意味着引用指向的堆内存中的对象被回收。通过这种方式，JVM允许我们在对象被销毁后，
做一些我们自己想做的事情。*

```

/*
 * java.lang.Object@60e53b93
 * null
 * null
 * =====
 * null
 * null
 * java.lang.ref.PhantomReference@5e2de80c
 */
public static void main(String[] args) {

    Object o=new Object();
    ReferenceQueue queue=new ReferenceQueue();
    PhantomReference pr=new PhantomReference(o,queue);
    System.out.println(o);
    System.out.println(pr.get());
    System.out.println(queue.poll());

    System.out.println("=====");

    o=null;
    System.gc();
    System.out.println(o);
    System.out.println(pr.get());
    System.out.println(queue.poll());

}

}

```

6.OOM

1.StackOverflowError

```

public static void main(String[] args) {
    stackOverflowError();
}

private static void stackOverflowError() {
    stackOverflowError();
}

```

2.Java heap space

```

public static void main(String[] args) {
    String s="1";
    while (true){
        s+="1"+ new Random().nextInt(10000000);
    }
}

```

3.GC overhead limit exceeded

超过98%的时间去GC但回收了不到2%的堆内存

- * GC回收时间过长时会抛出`OutOfMemoryError`。过长的定义是，超过98%的时间用来做GC并且回收了不到2%的堆内存
- * 连续多次GC都只回收了不到2%的极端情况下才会抛出。假如不抛出`GC overhead limit`错误会发生什么情况呢？
- * 那就是GC清理的这么点内存很快会再次填满，迫使GC再次执行。这样就形成恶性循环，
- * CPU使用率一直是100%，而GC却没有任何成果

```
public static void main(String[] args) {  
    int i=0;  
    List list=new ArrayList();  
    try{  
        while (true){  
            list.add(String.valueOf(i++).intern());  
        }  
    }catch(Throwable e){  
        System.out.println("i="+i);  
        e.printStackTrace();  
        throw e;  
    }  
}
```

4.Direct buffer Memory

NIO的allocateDirect()分配过大导致的直接内存溢出

故障现象：
`Exception in thread "main" java.lang.OutOfMemoryError: Direct buffer memory`

- * 导致原因：
- * 写NIO程序经常使用`ByteBuffer`来读取或者写入数据，这是一种基于通道(Channel)与缓冲区(Buffer)的I/O方式，
- * 它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆里面的`DirectByteBuffer`对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。
- *
- * `ByteBuffer.allocate(capability)` 第一种方式是分配JVM堆内存，属于GC管辖范围，由于需要拷贝所以速度相对较慢
- *
- * `ByteBuffer.allocateDirect(capability)` 第一种方式是分配OS本地内存，不属于GC管辖范围，由于不需要内存拷贝所以速度相对较快。
- *
- * 但如果不断分配本地内存，堆内存很少使用，那么JVM就不需要执行GC，`DirectByteBuffer`对象们就不会被回收，这时候堆内存充足，但本地内存可能已经使用光了，再次尝试分配本地内存就会出现`OutOfMemoryError`，那程序就直接崩溃了。
- */

```
public static void main(String[] args) {  
    // -Xms10m -Xmx10m -XX:MetaspaceSize=5m  
    ByteBuffer bf=ByteBuffer.allocateDirect(6*1024*1024);  
}
```

5.unable to create new native thread

高并发请求服务器时,经常出现如下异常:`java.lang.OutOfMemoryError: unable to create new native thread`
准确的讲该native thread异常与对应的平台有关

导致原因:

- 1 你的应用创建了太多线程了,一个应用进程创建多个线程,超过系统承载极限
- 2 你的服务器并不允许你的应用程序创建这么多线程,Linux系统默认允许单个进程可以创建的线程数是1024个,你的应用创建超过这个数量,就会报`java.lang.OutOfMemoryError: unable to create new native thread`

解决办法:

- 1.想办法降低你应用程序创建线程的数量,分析应用是否真的需要创建这么多线程,如果不是,改代码将线程数降到最低
- 2.对于有的应用,确实需要创建很多线程,远超过linux系统的默认1024个线程的限制,可以通过修改linux服务器配置,扩大linux默认限制

```
public static void main(String[] args) {  
    int i=0;  
    while (true){  
        i++;  
        new Thread(()->{  
            try {  
  
                TimeUnit.SECONDS.sleep(Integer.MAX_VALUE);}  
            catch(InterruptedException e){  
                e.printStackTrace();}  
        }).start();  
        System.out.println(i);  
    }  
}
```

6.Metaspace

Java 8及之后的版本使用Metaspace来替代永久代。

Metaspace是方法区在HotSpot中的实现,它与持久代最大的区别在于: Metaspace并不在虚拟机内存中而是使用本地内存也即在java8中,`class metadata(the virtual machines internal presentation of Java class)`,被存储在叫做Metaspace的native memory

永久代(java8后被原空间Metaspace取代了)存放了以下信息:

虚拟机加载的类信息
常量池
静态变量
即时编译后的代码

模拟Metaspace空间溢出,我们不断生成类往元空间灌,类占据的空间总是会超过Metaspace指定的空间大小的

```
/**  
 * 演示元空间内存溢出 java.lang.OutOfMemoryError: Metaspace  
 * -XX:MaxMetaspaceSize=8m  
 */  
public class Demo1_8 extends ClassLoader { // 可以用来加载类的二进制字节码  
    public static void main(String[] args) {  
        int j = 0;  
        try {  
            Demo1_8 test = new Demo1_8();  
            for (int i = 0; i < 10000; i++, j++) {  
                // ClassWriter 作用是生成类的二进制字节码  
                ClassWriter cw = new ClassWriter(0);  
                // 版本号, public, 类名, 包名, 父类, 接口  
                cw.visit(Opcodes.V1_8, Opcodes.ACC_PUBLIC, "Class" + i, null, "java/lang/Object", null);  
                // 返回 byte[]  
                byte[] code = cw.toByteArray();  
                // 执行了类的加载  
                test.defineClass("Class" + i, code, 0, code.length); // Class 对象  
            }  
        } finally {  
            System.out.println(j);  
        }  
    }  
}
```

7.GC算法与收集器的关系

GC垃圾回收算法和垃圾收集器的关系?分别是什么请你谈谈

关系：GC算法（标清，标整，复制，分代回收）是内存回收的理论，垃圾收集器就是算法落地实现

四种垃圾收集器：

- 1.串行垃圾回收器（Serial）：它为单线程环境设计并且只使用一个线程进行垃圾回收，会暂停所有的用户线程。（一个服务员清理多个桌子）
- 2.并行垃圾回收器（Parallel）：多个垃圾回收线程并行工作，此时用户线程是暂停的。（多个服务员清理多个桌子）
- 3.并发垃圾回收器（CMS）：用户线程和垃圾收集线程同时执行（不一定是并行，可能交替执行），不需要停顿用户线程。适用于对响应时间有要求的场景
- 4.G1垃圾回收器：G1垃圾回收器将堆内存分割成不同的区域然后并发的对其进行垃圾回收

8.垃圾收集器的配置和理解

怎么查看服务器默认的垃圾收集器是那个?生产上如何配置垃圾收集器的?谈谈你对垃圾收集器的理解?

1.查看默认的垃圾收集器：

JVM参数：

```
java -XX:+PrintCommandLineFlags -version
```

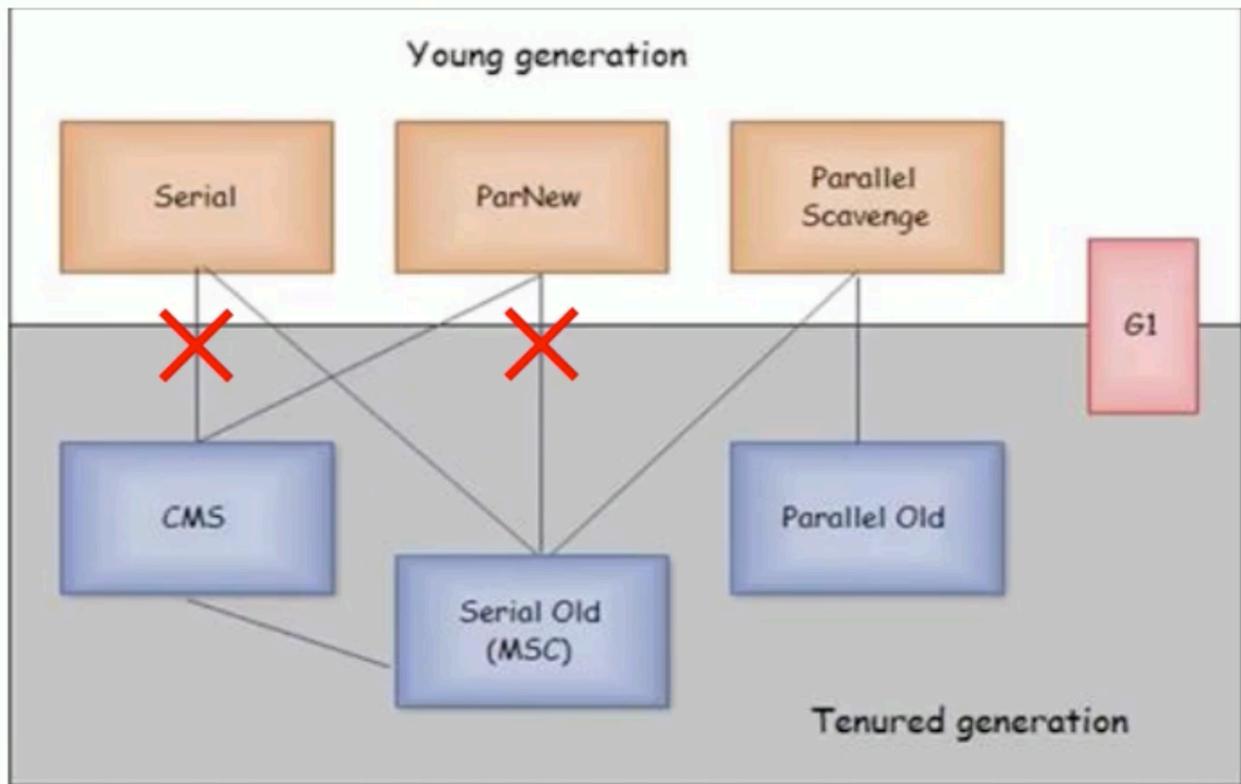
下图红色就是默认垃圾收集器

I

```
-XX:InitialHeapSize=265755648 -XX:MaxHeapSize=4252090368 -XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers  
-XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC  
java version "1.8.0_111"  
Java(TM) SE Runtime Environment (build 1.8.0_111-b14)  
Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)
```

2.生产上如何配置垃圾收集器：

红色表示java8版本开始，对应的垃圾收集器Deprecated，不推荐使用。



3.对垃圾收集器的理解

串行：

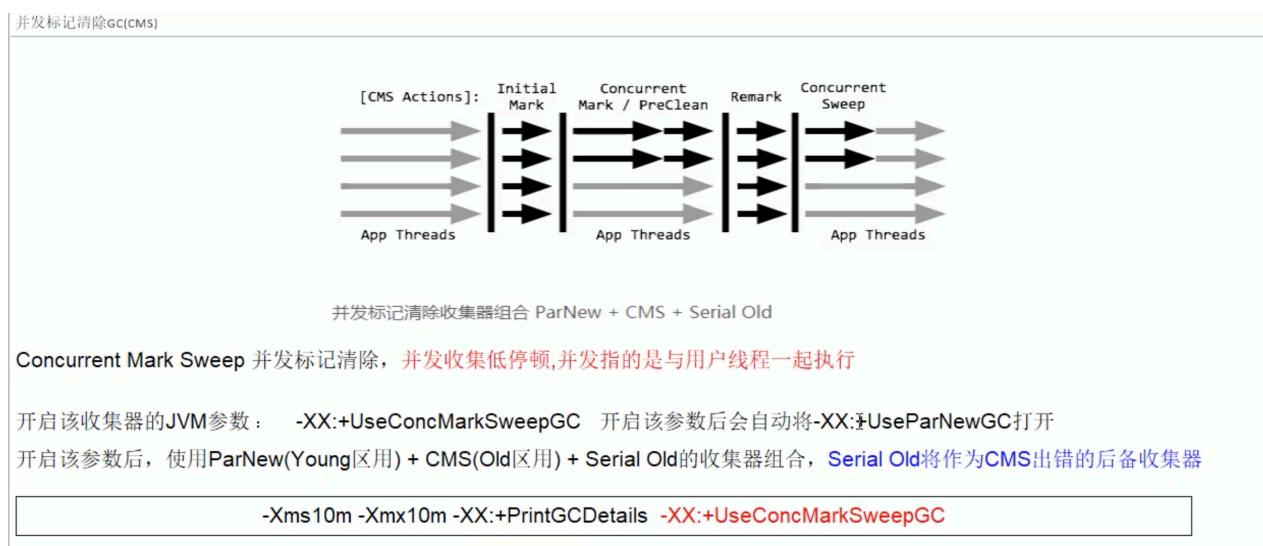
串行收集器是最古老，最稳定以及效率高的收集器，只使用一个线程去回收但其在进行垃圾收集过程中可能会产生较长的停顿（“Stop-The-World”状态）。虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限定单个CPU环境来说，**没有线程交互的开销可以获得最高的单线程垃圾收集效率**，因此**Serial**垃圾收集器依然是**java虚拟机运行在Client模式下默认的新生代垃圾收集器**。

对应JVM参数是：-XX:+UseSerialGC

开启后会使用：**Serial(Young区用) + Serial Old(Old区用)**的收集器组合

表示：新生代、老年代都会使用串行回收收集器，新生代使用复制算法，老年代使用标记-整理算法

CMS：



1.初始标记：只是标记一下GC roots能直接关联的对象，**会STW**，速度很快

2.并发标记：进行GC ROOTS跟踪的过程，和用户线程一起工作，**不会STW**。主要标记过程，标记全部对象

3.重新标记：（二次确认过程）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，**会STW**。

4.并发清除：清除GC ROOT不可达对象，和用户线程一起工作，**不会STW**。

优缺点：

优点：并发收集低停顿

缺点：

1>并发执行，对CPU资源压力大。(CMS在收集与应用程序会同时增加对堆内存的占用，也就是说，CMS必须在老年代堆内存用尽之前完成立即回收，否则CMS回收失败时，将会触发担保机制，退化为串行收集器，进行较长时间的STW)

2>标记清除产生碎片

4.各个垃圾收集器算法

| 参数 | 新生代垃圾收集器 | 新生代算法 | 老年代垃圾收集器 | 老年代算法 |
|--|--------------------|---------------------|---|-------|
| -XX:+UseSerialGC | SerialGC | 复制 | SerialOldGC | 标整 |
| -XX:+UseParNewGC | ParNew | 复制 | SerialOldGC | 标整 |
| -XX:+UseParallelGC/-XX:+UseParallelOldGC | Parallel[Scavenge] | 复制 | Parallel Old | 标整 |
| -XX:+UseConcMarkSweepGC | ParNew | 复制 | CMS + Serial Old的收集器组合 (Serial Old作为CMS出错的后备收集器) | 标清 |
| -XX:+UseG1GC | G1整体上采用标记-整理算法 | 局部是通过复制算法，不会产生内存碎片。 | | |

9.G1

1.G1是什么

像CMS收集器一样，能与应用程序线程并发执行。

整理空间更快。

需要更多的时间来预测GC停顿时间。

不希望牺牲大量的吞吐性能。

不需要更大的Java Heap。

G1收集器的设计目标是取代CMS收集器，它同CMS相比，在以下方面表现的更出色：

G1是一个有整理内存过程的垃圾收集器，不会产生很多内存碎片。

G1的Stop The World(STW)更可控，**G1在停顿时间上添加了预测机制，用户可以指定期望停顿时间。**

CMS垃圾收集器虽然减少了暂停应用程序的运行时间，但是它还是存在着内存碎片问题。于是，为了去除内存碎片问题，同时又保留CMS垃圾收集器低暂停时间的优点，JAVA7发布了一个新的垃圾收集器 - G1垃圾收集器。

G1是在2012年才在jdk1.7u4中可用。**oracle官方计划在jdk9中将G1变成默认的垃圾收集器以替代CMS**。它是一款面向服务端应用的收集器，主要应用在多CPU和大内存服务器环境下，极大的减少垃圾收集的停顿时间，全面提升服务器的性能，逐步替换java8以前的CMS收

2.底层原理

1.Region区域化垃圾收集器

最大好处是化整为零，避免全内存扫描，只需要按照区域来进行扫描即可

2.回收过程

初始标记：只标记GC Roots能直接关联到的对象

并发标记：进行GC Roots Tracing的过程

最终标记：修正并发标记期间，因程序运行导致标记发生变化的那一部分对象

筛选回收：根据时间来进行价值最大化的回收

形如：

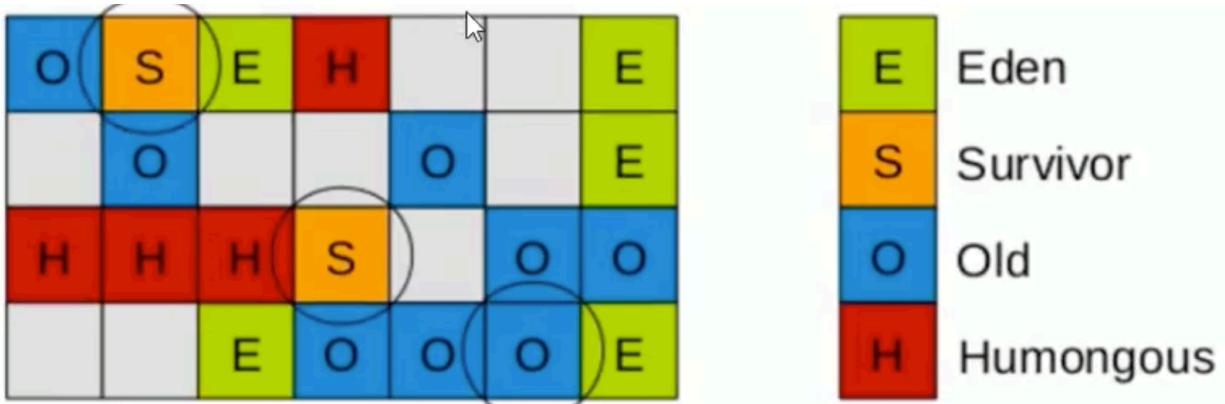
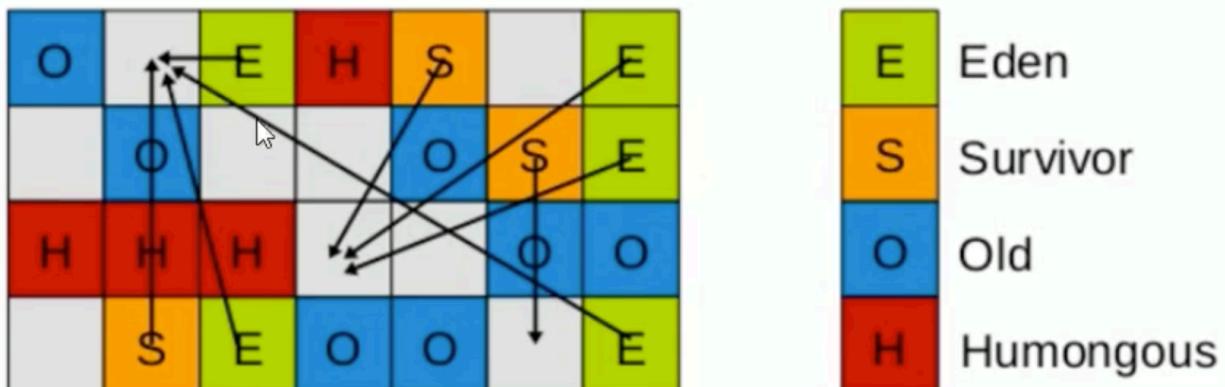


3. 回收步骤

G1收集器下的Young GC

针对Eden区进行收集，Eden区耗尽后会被触发，主要是小区域收集 + 形成连续的内存块，避免内存碎片

- * Eden区的数据移动到Survivor区，假如出现Survivor区空间不够，Eden区数据会晋升到Old区
- * Survivor区的数据移动到新的Survivor区，部分数据会晋升到Old区
- * 最后Eden区收拾干净了，GC结束，用户的应用程序继续执行。

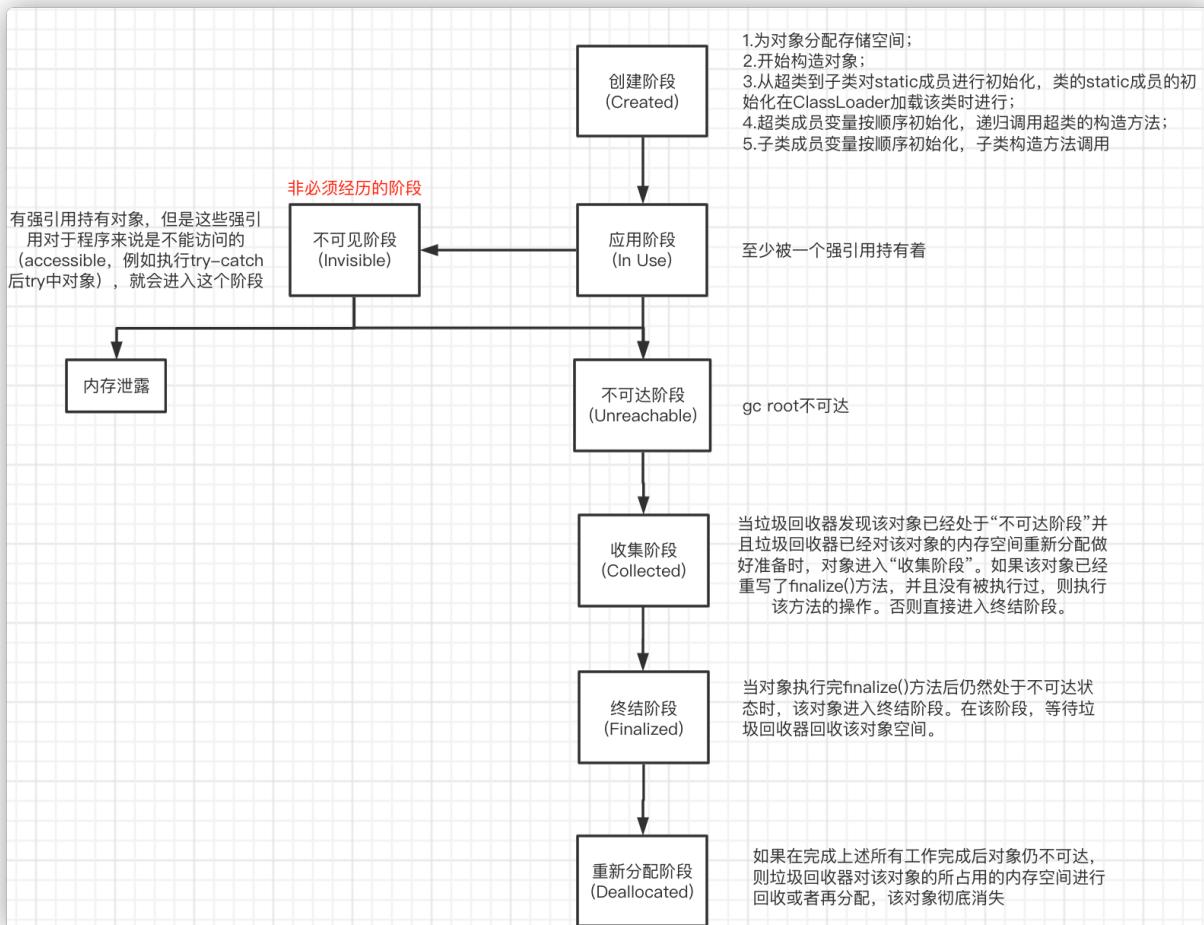


3. 较CMS的优势

比起cms有两个优势：

- 1) G1不会产生内存碎片。
- 2) 是可以精确控制停顿。该收集器是把整个堆（新生代、老生代）划分成多个固定大小的区域，每次根据允许停顿的时间去收集垃圾最多的区域。

10.java对象的生命周期



文献：https://web.archive.org/web/20120626144027/http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html