

一、Java内存区域 (JDK1.8)

- 1.1 程序计数器
- 1.2 Java虚拟机栈
- 1.3 本地方法栈
- 1.4 Java堆
- 1.5 方法区
- 1.6 运行时常量池
- 1.7 直接内存

二、OutOfMemoryError

- 2.1 Java堆溢出
- 2.2 虚拟机栈和本地方法栈溢出
- 2.3 方法区和运行时常量池溢出
- 2.4 String.intern()引发的思考
 - 2.4.1 分析1
 - 2.4.2 分析2
 - 2.4.3 分析3
 - 2.4.4 分析4
 - 2.4.5 分析5
- 2.5 本机直接内存溢出
- 2.6 内存溢出和内存泄漏的区别

三、判断对象的生死

- 3.1 引用计数算法
- 3.2 可达性分析算法
- 3.3 Java中的引用
 - 3.3.1 强引用
 - 3.3.2 软引用
 - 3.3.3 弱引用
 - 3.3.4 虚引用
- 3.4 真正的死亡
- 3.5 回收方法区

四、垃圾回收算法

- 4.1 Java堆的划分
- 4.2 垃圾回收算法
 - 4.2.1 标记-清除算法
 - 4.2.2 复制算法
 - 4.3.3 标记-整理算法
 - 4.3.4 分代收集算法
- 4.3 Minor GC和Full GC区别
- 4.4 Client模式和Server模式下的GC算法区别

五、JVM参数配置

- 5.1 常见参数配置
- 5.2 堆内存大小配置
- 5.3 设置新生代比例参数
- 5.4 设置新生代与老年代比例参数

六、垃圾收集器

- 6.1 Serial收集器
- 6.2 ParNew收集器
- 6.3 Parallel Scavenge收集器
- 6.4 Serial Old收集器
- 6.5 Parallel Old收集器
- 6.6 CMS收集器

6.7 G1收集器

七、内存分配策略

7.1 对象优先在Eden分配

7.2 大对象直接进入老年代

7.3 长期存活的对象将进入老年代

7.4 动态对象年龄判定

7.5 空间分配担保

八、JDK可视化工具

九、字节码技术

十、类加载机制

10.1 类加载的时机

10.2 类加载过程

10.2.1 加载

10.2.2 连接过程

10.2.3 初始化

10.3 类加载器

10.3.1 加载器分类

10.3.2 双亲委派模型

10.3.3 破坏双亲委派

10.3.4 类加载器常用方法

10.4 热部署

10.4.1 热部署原理

10.4.2 热部署与热加载

10.4.3 实现

一、Java内存区域 (JDK1.8)

Java虚拟机再执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。主要包含以下几个运行时数据区域：

JVM运行时内存区域概览



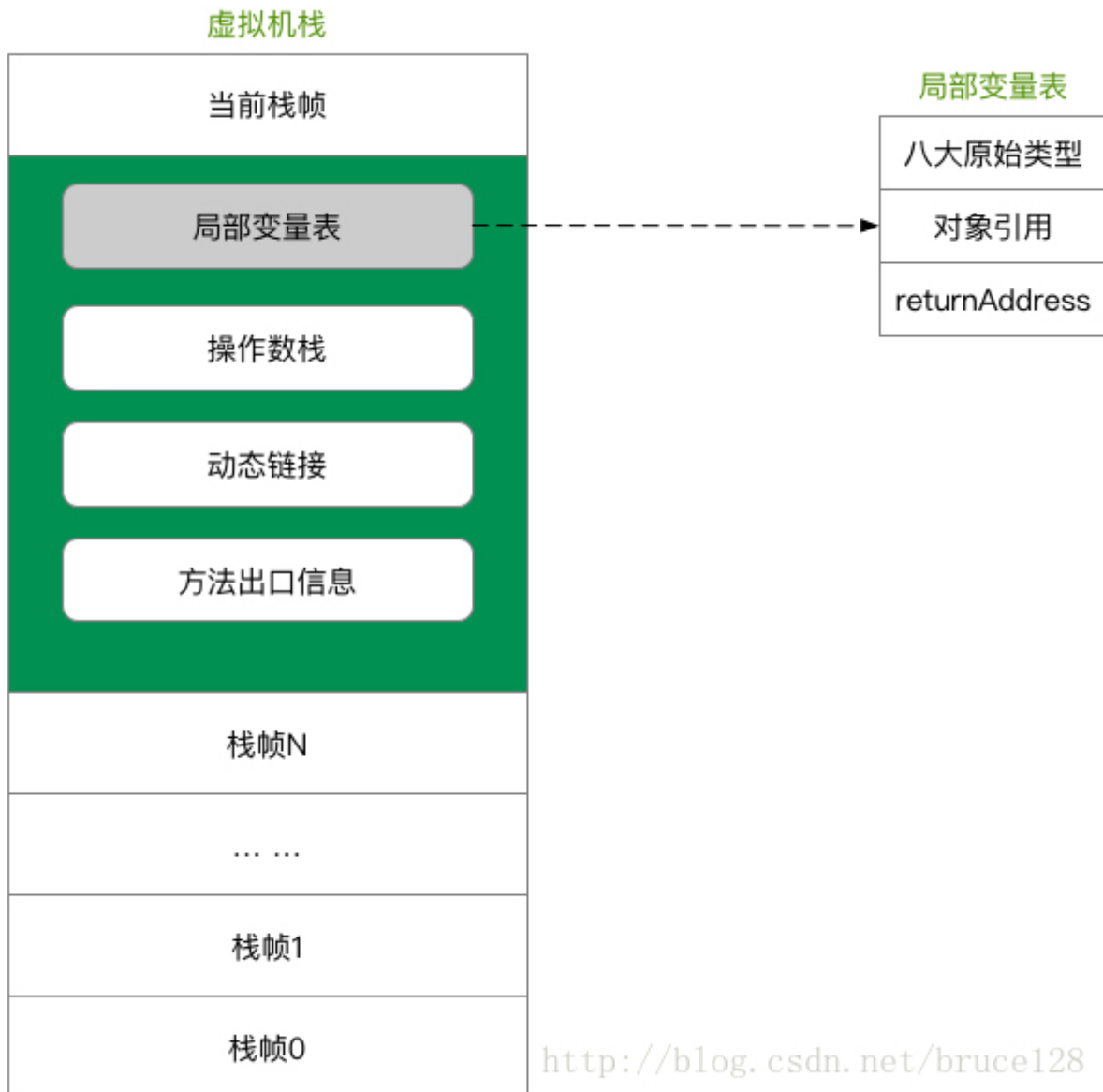
1.1 程序计数器

程序计数器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。

由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，一个处理器都只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都有一个独立的程序计数器，各个线程之间计数器互不影响，独立存储。称之为“线程私有”的内存。程序计数器内存区域是虚拟机中唯一没有规定OutOfMemoryError情况的区域。

1.2 Java虚拟机栈

java虚拟机也是线程私有的，它的生命周期和线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法在运行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。



咱们常说的堆内存、栈内存中，栈内存指的就是虚拟机栈。局部变量表存放了编译期可知的各种基本数据类型（8个基本数据类型）、对象引用（地址指针）、returnAddress类型。

局部变量表所需的内存空间在编译期间完成分配。在运行期间不会改变局部变量表的大小。

这个区域规定了两种异常状态：如果线程请求的栈深度大于虚拟机所允许的深度，则抛出StackOverflowError异常；如果虚拟机栈可以动态扩展，在扩展是无法申请到足够的内存，就会抛出OutOfMemoryError异常。

1.3 本地方法栈

本地方法栈与虚拟机栈所发挥作用非常相似，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。本地方法栈也是抛出两个异常。

1.4 Java堆

java堆是java虚拟机所管理的内存中最大的一块，是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，这一点在Java虚拟机规范中的描述是：**所有的对象实例以及数组都要在堆上分配。**

java堆是垃圾收集器管理的主要区域，因此也成为“GC堆”（Garbage Collected Heap）。从内存回收角度来看java堆可分为：新生代和老生代。从内存分配的角度看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer, TLAB）。无论怎么划分，都与存放内容无关，无论哪个区域，存储的都是对象实例，进一步的划分都是为了更好的回收内存，或者更快的分配内存。

根据Java虚拟机规范的规定，java堆可以处于物理上不连续的内存空间中。当前主流的虚拟机都是可扩展的（通过 -Xmx 和 -Xms 控制）。如果堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

1.5 方法区

方法区与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。它有个别名叫Non-Heap（非堆）。当方法区无法满足内存分配需求时，抛出OutOfMemoryError异常。

JDK1.7中将放在永久代中的常量池移出到Java堆中

JDK1.8已经不存在永久代，使用元空间来实现方法区。

元数据区取代了1.7版本及以前的永久代。元数据区和永久代本质上都是方法区的实现。方法区存放虚拟机加载的类信息，静态变量，常量等数据。

符号引用(Symbols)转移到了native heap;字面量(interned strings)转移到了java heap;类的静态变量(class statics)转移到了java heap。

方法区和永久代的区别：

什么是方法区？

方法区（Method Area）是**jvm规范**里面的运行时数据区的一个组成部分，jvm规范中的运行时数据区还包含了：pc寄存器、虚拟机栈、堆、方法区、运行时常量池、本地方法栈。

方法区存储东西？

主要用来存储class、运行时常量池、字段、方法、代码、JIT代码等。

注意：

- (1) **运行时数据区跟内存不是一个概念。**
- (2) **方法区是运行时数据区的一部分**
- (3) 方法区是**jvm规范**中的一部分，**并不是实际的实现**，切忌将规范跟实现[混为一谈](#)。

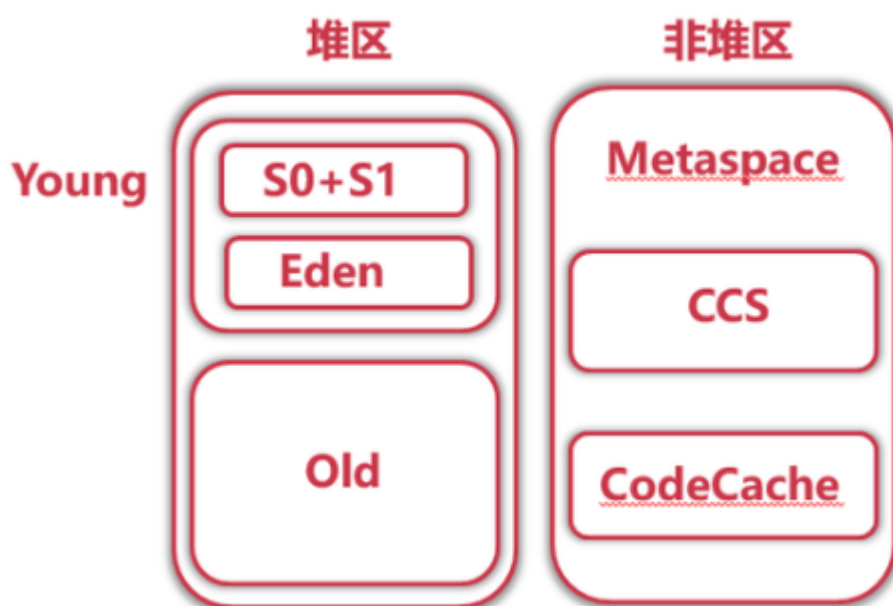
我们再来看下永久带（Perm区）：

永久带又叫Perm区，只存在于hotspot jvm中，并且只存在于jdk7和之前的版本中，jdk8中已经彻底移除了永久带，jdk8中引入了一个新的内存区域叫metaspace。

- (1) 并不是所有的jvm中都有永久带，[ibm](#)的j9，[oracle](#)的JRocket都没有永久带。
- (2) 永久带是**实现层面**的东西。
- (3) 永久带里面存的东西基本上就是方法区规定的那些东西。

因此，我们可以说，永久带是方法区的一种实现，当然，在hotspot jdk8中metaspace可以看成是方法区的一种实现。

下面我们来看下hotspot jdk8中移除了永久带以后的内存结构：



结论：

- (1) 方法区是规范层面的东西，规定了这一个区域要存放哪些东西
- (2) 永久带或者是metaspace是对方法区的不同实现，是实现层面的东西。

1.6 运行时常量池

运行时常量池是方法区的一部分。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，用于存放编译期生成的各种字面量和符号引用，这部分内容将在加载后进入方法区的运行时常量池中存放。

1.7 直接内存

直接内存不是虚拟机运行时数据区的一部分，也不是java虚拟机规范中定义的内存区域。但这部分区域也频繁使用，而且也可能导致OutOfMemoryError异常

在JDK1.4中新加入的NIO（New Input/Output）类，引入了一种基于通道（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堆中的DirectByteBuffer对象作为这块内存的引用进行操作。

二、OutOfMemoryError

2.1 Java堆溢出

Java堆用于存储对象实例，只要不断的创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清楚这些对象，那么在对象数量达到最大堆的容量限制后就会产生内存溢出异常。

设置运行参数: `-Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError`

将堆的最小值和最大值设置为一样可避免自动扩展, 通过参数 `-XX:+HeapDumpOnOutOfMemoryError` 可以让虚拟机在出现内存溢出异常时Dump出当前的内存堆转储快照以便事后进行分析。

代码:

```
1  package com.jvm.outofmemory;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6
7  Author: 98050
8  Time: 2019-01-10 20:40
9  Feature: Java堆溢出 -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
10
11 public class HeapOOM {
12     static class OOMObject{
13     }
14
15     public static void main(String[] args) {
16         List<OOMObject> list = new ArrayList<OOMObject>();
17         while (true){
18             list.add(new OOMObject());
19         }
20     }
21 }
22
```

结果:

```
.jar;D:\jdk\jre\lib\plugin.jar;D:\jdk\jre\lib\resources.jar;D:\jdk\jre\lib\rt.jar;E:\J
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid23488.hprof ...
Heap dump file created [28337808 bytes in 0.107 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
-----
at java.util.Arrays.copyOf(Arrays.java:3210)
at java.util.Arrays.copyOf(Arrays.java:3181)
at java.util.ArrayList.grow(ArrayList.java:265)
at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:239)
at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:231)
at java.util.ArrayList.add(ArrayList.java:462)
at com.jvm.outofmemory.HeapOOM.main(HeapOOM.java:20)
-----
Process finished with exit code 1
```

2.2 虚拟机栈和本地方法栈溢出

由于在HotSpot虚拟机中并不区分虚拟机栈和本地方法栈, 因此只能由-Xss参数设置栈容量。关于虚拟机栈和本地方法栈, 在Java虚拟机中描述了两种异常:

- 如果线程请求的栈深度大于虚拟机所允许的最大深度, 将抛出StackOverflowError异常
- 如果虚拟机在扩展栈时无法申请到足够的内存空间, 则抛出OutOfMemory异常

```

1 age com.jvm.outofmemory;
2
3
4 Author: 98050
5 Time: 2019-01-10 21:05
6 Feature: -Xss128k
7
8 public class JavaVMStackSOF {
9
10 private int stackLength = 1;
11
12 public void stackLeak(){
13     stackLength ++;
14     stackLeak();
15 }
16
17 public static void main(String[] args) throws Throwable {
18     JavaVMStackSOF oom = new JavaVMStackSOF();
19     try{
20         oom.stackLeak();
21     }catch (Throwable e){
22         System.out.println("stack length:" + oom.stackLength);
23         throw e;
24     }
25 }
26
27

```

结果:

不设置栈容量:

```

D:\jdk\jre\lib\charsets.jar;D:\jdk\jre\lib\deploy.jar;D:\jdk\jre\lib\ext\access-bridge-6
D:\jdk\jre\lib\ext\jaccess.jar;D:\jdk\jre\lib\ext\jfxrt.jar;D:\jdk\jre\lib\ext\localedat
D:\jdk\jre\lib\ext\sunjsce_provider.jar;D:\jdk\jre\lib\ext\sunmscapi.jar;D:\jdk\jre\lib\
D:\jdk\jre\lib\jce.jar;D:\jdk\jre\lib\jfr.jar;D:\jdk\jre\lib\jfxswt.jar;D:\jdk\jre\lib\j
D:\jdk\jre\lib\resources.jar;D:\jdk\jre\lib\rt.jar;E:\Java_Demo\jvm\target\classes com.j
Exception in thread "main" java.lang.StackOverflowError
stack length:38018
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:14)
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:14)
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:14)
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:14)
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:14)

```

设置栈容量为128k

```

D:\jdk\jre\lib\jce.jar;D:\jdk\jre\lib\jfr.jar;D:\jdk\jre\lib\jfxswt.jar;D:\jdk\jre\lib\jsse.jar
D:\jdk\jre\lib\resources.jar;D:\jdk\jre\lib\rt.jar;E:\Java_Demo\jvm\target\classes com.jvm.outo
stack length:996
Exception in thread "main" java.lang.StackOverflowError
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:13)
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:14)
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:14)
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:14)
—>at com.jvm.outofmemory.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:14)

```


单个线程下无论由于栈帧太大还是虚拟机容量太小，当内存无法分配的时候，虚拟机抛出的都是StackOverflowError异常。

```
1 age com.jvm.outofmemory;
2
3
4 Author: 98050
5 Time: 2019-01-10 21:15
6 Feature: -Xss200M 创建线程导致内存溢出异常
7
8 public class JavaVMStackOOM {
9     private void dontStop(){
10         while (true){
11
12         }
13     }
14
15     public void stackLeakByThread(){
16         while (true){
17             Thread thread = new Thread(new Runnable() {
18                 public void run() {
19                     dontStop();
20                 }
21             });
22             thread.start();
23         }
24     }
25
26     public static void main(String[] args) {
27         JavaVMStackOOM oom = new JavaVMStackOOM();
28         oom.stackLeakByThread();
29     }
30
31 }
```

2.3 方法区和运行时常量池溢出

运行时常量池是方法区的一部分。

JDK1.7之前方法区也叫永久代，使用 `-XX:PermSize=2m -XX:MaxPermSize=2m` 来设置方法区的大小

JDK1.8使用元空间（Metaspace）替代了永久代（PermSize），因此使用 `-xx:MetaspaceSize=2m -xx:MaxMetaspaceSize=2m` 来设置元空间的大小

代码：

```
1 age com.jvm.outofmemory;
2
3 rt java.util.ArrayList;
4 rt java.util.List;
5
6
```

```
7 Author: 98050
8 Time: 2019-01-10 21:47
9 Feature: JDK1.7之前:-XX:PermSize=10M -XX:MaxPermSize=10M      JDK1.8:-XX:MetaspaceSize=2m
      MaxMetaspaceSize=2m
10
11 public class RuntimeConstantPoolOOM {
12     public static void main(String[] args) {
13         List<String> list = new ArrayList<String>();
14         int i = 0;
15         while (true){
16             list.add(String.valueOf(i++).intern());
17         }
18     }
19
20 }
```

结果:

```
D:\jdk\bin\java.exe -XX:MetaspaceSize=2M -XX:MaxMetaspaceSize=2M
-Dfile.encoding=UTF-8 -classpath D:\jdk\jre\lib\charsets.jar;D:\j
D:\jdk\jre\lib\ext\dnsns.jar;D:\jdk\jre\lib\ext\jaccess.jar;D:\jd
D:\jdk\jre\lib\ext\sunec.jar;D:\jdk\jre\lib\ext\sunec_provider.j
.jar;D:\jdk\jre\lib\javaws.jar;D:\jdk\jre\lib\jce.jar;D:\jdk\jre\
.jar;D:\jdk\jre\lib\plugin.jar;D:\jdk\jre\lib\resources.jar;D:\jd
Error occurred during initialization of VM
OutOfMemoryError: Metaspace

Process finished with exit code 1
|
```

2.4 String.intern()引发的思考

String.intern()是一个Native方法，它的作用是：如果字符串常量池中已经包含一个等于此String对象的字符串，则返回代表池中这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。

知识点

JDK 1.7 和 1.8 将字符串常量由永久代转移到堆中。JDK 1.7以后的intern()实现不会再复制实例到永久代中，因为永久代已经被移除，用元空间来实现方法区。

字符串池里的内容是在类加载完成，经过验证，准备阶段之后在堆中生成字符串对象实例，然后将该字符串对象实例的引用值存到string pool中（记住：string pool中存的是引用值而不是具体的实例对象，具体的实例对象是在堆中开辟的一块空间存放的）。

在HotSpot VM里实现的string pool功能的是一个StringTable类，它是一个哈希表，里面存的是驻留字符串(也就是我们常说的用双引号括起来的)的引用（而不是驻留字符串实例本身），也就是说在堆中的某些字符串实例被这个StringTable引用之后就等同被赋予了“驻留字符串”的身份。这个StringTable在每个HotSpot VM的实例只有一份，被所有的类共享。

JDK 1.7后，intern方法还是会先去查询常量池中是否有已经存在的字符串引用，如果存在，则返回常量池中的引用，这一点与之前没有区别，区别在于，如果在常量池找不到对应的字符串，则不会再将字符串拷贝到常量池，而只是在常量池中生成一个对原字符串的引用。简单的说，就是往常量池放的东西变了：原来在常量池中找不到时，复制一个副本放到常量池，1.7后则是将在堆上的地址引用复制到常量池。

2.4.1 分析1

```
1 String str3 = new String("hello");
2 tem.out.println(str3.intern() == str3);
3
4 String str4 = new String("java");
5 tem.out.println(str4.intern() == str4);
```

运行结果都为false。为什么？

str3指向的是堆上的对象引用，str3.intern()返回的是常量池中"hello"的引用；str4同理。

```
1 String temp = new String("hello");
2 temp.intern();
3 String temp2 = "hello";
4 System.out.println(temp == temp2);
```

返回false

2.4.2 分析2

```
1 String temp = new String("hello");
2 String temp2 = "hello";
3 System.out.println(temp == temp2);
4 System.out.println(temp2 == temp.intern());
```

通过字面量赋值创建字符串时，会先在常量池中查找是否存在相同的字符串，若存在，则将栈中的引用直接指向该字符串；若不存在，则在常量池中生成一个字符串，再将栈中的引用指向该字符串。

第一个返回结果是false，第二个是true。在直接赋值创建temp2时会扫描常量池中是否有"hello"，因为有第一步，所以在创建temp2的时候常量池中已经有"hello"了，那么temp2就返回"hello"的引用即可。所以第一个输出false，因为temp在堆上。temp2返回的是常量池中的引用，所以第二条输出就返回true了。

2.4.3 分析3

```
1 String str5 = "hello";
2 System.out.println(str5.intern() == str5);
3
4 String str6 = "java";
5 System.out.println(str6.intern() == str6);
```

运行结果都为true。这个就容易理解了，调用intern方法返回的是常量池中的引用，即指向同一个地址。

2.4.4 分析4

```

1      String str1 = new StringBuilder("he").append("llo").toString();
2      System.out.println(str1.intern() == str1);
3
4      String str2 = new StringBuilder("ja").append("va").toString();
5      System.out.println(str2.intern() == str2);

```

第一个返回true，第二个返回false。

看一下StringBuilder的源码：

```

@NotNull @Contract(pure=true) @Override
public String toString() {
    // Create a copy, don't share the array
    return new String(value, offset: 0, count);
}

```

调用toString的时候，也就是调用String的构造函数：

```

@Contract(pure=true) public String( @NotNull char value[], int offset, int count) {
    if (offset < 0) {
        throw new StringIndexOutOfBoundsException(offset);
    }
    if (count <= 0) {
        if (count < 0) {
            throw new StringIndexOutOfBoundsException(count);
        }
        if (offset <= value.length) {
            this.value = "".value;
            return;
        }
    }
    // Note: offset or count might be near -1>>>1.
    if (offset > value.length - count) {
        throw new StringIndexOutOfBoundsException(offset + count);
    }
    this.value = Arrays.copyOfRange(value, offset, to: offset+count);
}

```

与普通的构造函数还是有差别的：

```

/*
 * A {@code String}
 */
@Contract(pure=true) public String( @NotNull String original) {
    this.value = original.value;
    this.hash = original.hash;
}

```

所以在调用完toString方法后，常量池中还没有"hello"，当调用intern方法时，放入常量池中的是str1对象的引用，所以第一个返回结果为true。

```
1 String str1 = new StringBuilder("he").append("llo").toString();
2 String temp = "hello";
3 System.out.println(temp == str1);
```

返回结果为false，也可以间接证明。

还有一个问题，为什么第二个会返回false。说明常量池中已经有“java”了，所以两个对象地址不同。

常量池中存放编译期生成的各种字面量和符合引用。

2.4.5 分析5

```
1 String str7 = new String("ja")+new String("va");
2 System.out.println(str7 == str7.intern());
3
4 String str8 = new String("he")+new String("llo");
5 System.out.println(str8 == str8.intern());
```

第一个返回结果是false，第二个返回结果是true。为什么？关键在于“+”这个连接符，它的底层是调用StringBuilder实现的，所以一切就了然于胸了。

2.5 本机直接内存溢出

DirectMemory可以通过-xx:MaxDirectMemorySize指定，如果不指定，则与默认Java堆最大值(-Xmx)一样。

jdk1.4之后，引入nio概念，加入了通道，缓冲区。可以直接使用native方法库直接分配堆外内存。在某些场景下，可以避免，在堆中来回复制对象，而是直接通过DirectByteBuffer直接操作该内存区域，从而提高性能。直接内存溢出：是在程序使用了NIO后发生OutOfMemoryError异常。

代码：

```
1 package com.jvm.outofmemory;
2
3 import sun.misc.Unsafe;
4
5 import java.lang.reflect.Field;
6
7
8 Author: 98050
9 Time: 2019-01-10 22:36
10 Feature: 直接内存溢出 -Xmx20m -XX:MaxDirectMemorySize=10M
11
12 public class DirectMemoryOOM {
13     private static final int _1MB = 1024*1024;
14
15     public static void main(String[] args) throws IllegalAccessException {
16         Field unsafeField = Unsafe.class.getDeclaredFields()[0];
17         unsafeField.setAccessible(true);
18         Unsafe unsafe = (Unsafe) unsafeField.get(null);
19         while (true){
20             unsafe.allocateMemory(_1MB);
21         }
```

```
22 }
23
24
```

结果:

```
D:\jdk\jre\lib\ext\dnsns.jar;D:\jdk\jre\lib\ext\jaccess.jar;D:\jdk\jre\lib\ext\jfxrt.jar;D:
D:\jdk\jre\lib\ext\sunec.jar;D:\jdk\jre\lib\ext\sunjce_provider.jar;D:\jdk\jre\lib\ext\sunm
.jar;D:\jdk\jre\lib\javaws.jar;D:\jdk\jre\lib\jce.jar;D:\jdk\jre\lib\jfr.jar;D:\jdk\jre\lib
.jar;D:\jdk\jre\lib\plugin.jar;D:\jdk\jre\lib\resources.jar;D:\jdk\jre\lib\rt.jar;E:\Java_D
Exception in thread "main" java.lang.OutOfMemoryError
  at sun.misc.Unsafe.allocateMemory(Native Method)
  at com.jvm.outofmemory.DirectMemoryOOM.main(DirectMemoryOOM.java:20)

Process finished with exit code 1
```

2.6 内存溢出和内存泄漏的区别

Java内存泄漏就是没有及时清理内存垃圾，导致系统无法再给你提供内存资源（内存资源耗尽）；

Java内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。

内存溢出，这个好理解，说明存储空间不够大。就像倒水倒多了，从杯子上面溢出了来了一样。内存泄漏，原理是，使用过的内存空间没有被及时释放，长时间占用内存，最终导致内存空间不足，而出现内存溢出。

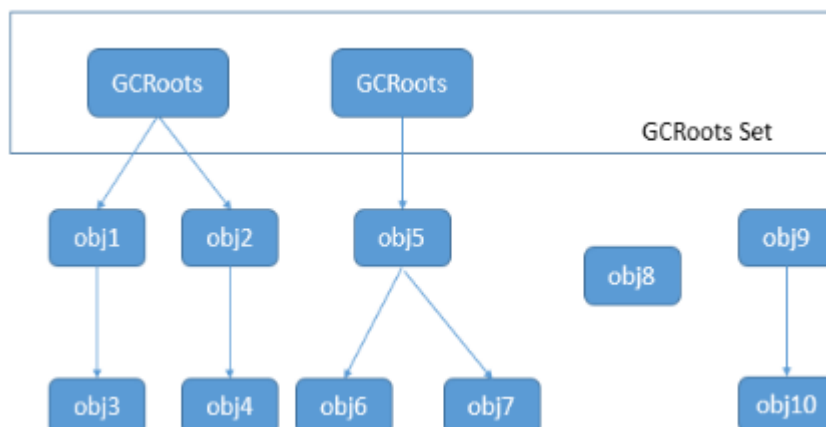
三、判断对象的生死

3.1 引用计数算法

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值加1；当引用失效时，计数器值减1.任何时刻计数器值为0的对象就是不可能再被使用的。但是它很难解决对象之间相互循环引用的问题。

3.2 可达性分析算法

其基本思路**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。



如何选取GC Roots对象呢？在Java语言中，可以作为GC Roots的对象包括下面几种：

(1). 虚拟机栈（栈帧中的局部变量区，也叫做局部变量表）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。

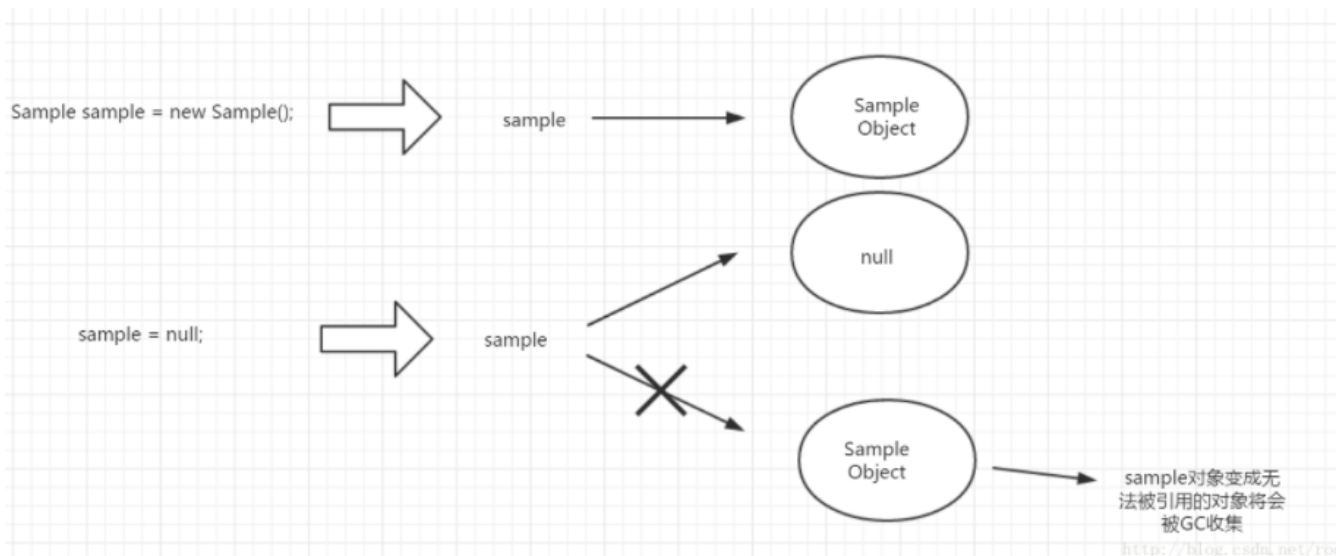
(4). 本地方法栈中JNI(Native方法)引用的对象。

3.3 Java中的引用

3.3.1 强引用

只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象

```
1  package com.jvm.references;
2
3
4  Author: 98050
5  Time: 2019-01-11 21:03
6  Feature: 强引用
7
8  public class Test {
9
10     public static void main(String[] args) {
11         //创建一个对象，new出来的对象都是分配在java堆中的
12         Sample sample = new Sample();    //sample这个引用就是强引用
13
14         sample = null;                    //将这个引用指向空指针，
15         //那么上面那个刚new来的对象就没用任何其它有效的引用指向它了
16         //也就说该对象对于垃圾收集器是符合条件的
17         //因此在接下来某个时间点 GC进行收集动作的时候，该对象将会被销毁，内存被释放
18     }
19
20     Sample{
21
22
23
```

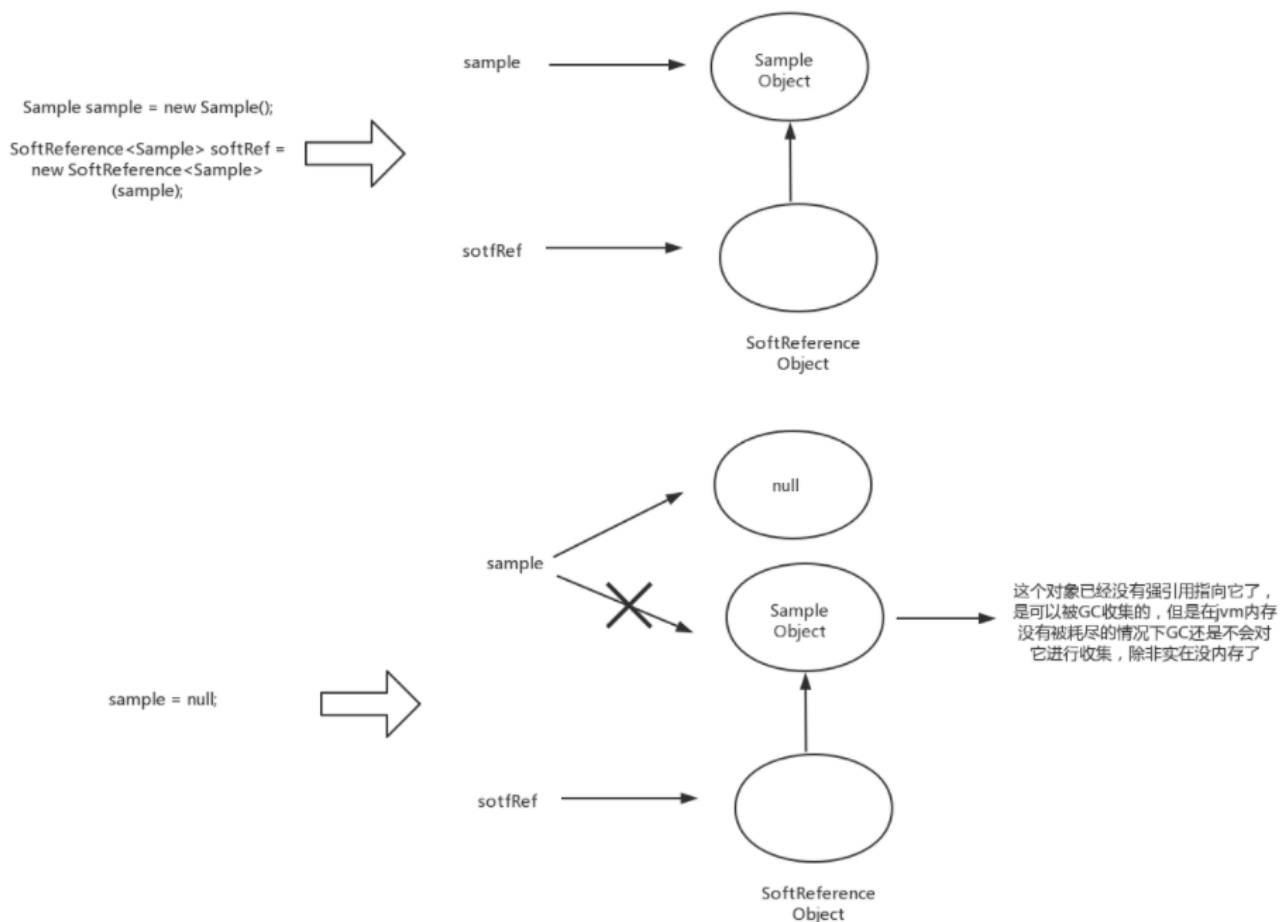


3.3.2 软引用

软引用在java.lang.ref包中有与之对应的类java.lang.ref.SoftReference。

重点： 被软引用指向的对象不会被垃圾收集器收集(即使该对象没有强引用指向它)，除非jvm使用内存不够了，才会对这类对象进行销毁，释放内存。举个简单的例子：

```
1  age com.jvm.references;
2
3  rt java.lang.ref.SoftReference;
4
5
6  Author: 98050
7  Time: 2019-01-11 21:08
8  Feature: 软引用
9
10 ic class Test2 {
11
12 public static void main(String[] args) {
13     //创建一个对象，new出来的对象都是分配在java堆中的
14     Sample sample = new Sample();    //sample这个引用就是强引用
15
16     //创建一个软引用指向这个对象    那么此时就有两个引用指向Sample对象
17     SoftReference<Sample> softRef = new SoftReference<Sample>(sample);
18
19     //将强引用指向空指针 那么此时只有一个软引用指向Sample对象
20     //注意：softRef这个引用也是强引用，它是指向SoftReference这个对象的
21     //private T referent; 这个才是软引用， 只被jvm使用
22     sample = null;
23
24     //可以重新获得Sample对象，并用一个强引用指向它
25     sample = softRef.get();
26 }
27
28
```

测试一下当jvm内存不足的情况下，软引用的回收情况。

```

1  age com.jvm.references;
2
3  rt java.lang.ref.SoftReference;
4  rt java.util.ArrayList;
5  rt java.util.List;
6
7
8  Author: 98050
9  Time: 2019-01-11 21:08
10 Feature: 软引用 -Xmx100m
11
12 ic class Test2 {
13
14 private static final List<Object> TEST_DATA = new ArrayList<>();
15
16 public static void main(String[] args) throws InterruptedException {
17     //创建一个对象，new出来的对象都是分配在java堆中的
18     Sample sample = new Sample(); //sample这个引用就是强引用
19
20     //创建一个软引用指向这个对象 那么此时就有两个引用指向Sample对象
21     SoftReference<Sample> softRef = new SoftReference<Sample>(sample);
22
23     //将强引用指向空指针 那么此时只有一个软引用指向Sample对象
24     //注意：softRef这个引用也是强引用，它是指向SoftReference这个对象的

```

```

25 //private T referent; 这个才是软引用, 只被jvm使用
26 sample = null;
27
28 //可以重新获得Sample对象, 并用一个强引用指向它
29 sample = softRef.get();
30
31 new Thread(new Runnable() {
32     @Override
33     public void run() {
34         while (true) {
35             System.out.println(softRef.get());
36             try {
37                 Thread.sleep(1000);
38             } catch (InterruptedException e) {
39                 e.printStackTrace();
40             }
41             TEST_DATA.add(new byte[1024 * 1024 * 5]);
42         }
43     }
44 }).start();
45 }
46
47

```

```

1  age com.jvm.references;
2
3
4  Author: 98050
5  Time: 2019-01-11 21:20
6  Feature:
7
8  ic class Sample {
9
10 private final byte[] data;
11
12 public Sample(){
13     data = new byte[1024 * 1024 * 10];
14 }
15
16

```

Java堆设置100M

结果:

```
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
com.jvm.references.Sample@3b94507a
null
null
Exception in thread "Thread-0" java.lang.OutOfMemoryError: Java heap space
    at com.jvm.references.Test2$1.run(Test2.java:42)
    at java.lang.Thread.run(Thread.java:748)
```

当堆内存耗尽的时候，会将软引用回收，（Sample对象大小为10M），那么就会输出两个null。（线程内分配内存大小为5M）

当然在创建软引用时，还可以传入ReferenceQueue，这个队列有啥用呢？当jvm回收某个软引用对象之后会将该SoftReference对象(例子中的softRef对象)添加进这个队列，因此就可以知道这个对象啥时候被回收了，可以做一些其它操作。

```
1  package com.jvm.references;
2
3  import java.lang.ref.Reference;
4  import java.lang.ref.ReferenceQueue;
5  import java.lang.ref.SoftReference;
6  import java.util.ArrayList;
7  import java.util.List;
8
9
10 Author: 98050
11 Time: 2019-01-11 21:08
12 Feature: 软引用 -Xmx100m
13
14 public class Test2 {
15
16     private static final List<Object> TEST_DATA = new ArrayList<>();
17
18     private static final ReferenceQueue<Sample> QUEUE = new ReferenceQueue<>();
19
20
21     public static void main(String[] args) throws InterruptedException {
22         //创建一个对象，new出来的对象都是分配在java堆中的
23         Sample sample = new Sample(); //sample这个引用就是强引用
```

```

24
25 //创建一个软引用指向这个对象 那么此时就有两个引用指向Sample对象
26 SoftReference<Sample> softRef = new SoftReference<Sample>(sample,QUEUE);
27
28 //将强引用指向空指针 那么此时只有一个软引用指向Sample对象
29 //注意: softRef这个引用也是强引用, 它是指向SoftReference这个对象的
30 //那么这个软引用在哪呢? 可以跟一下java.lang.Reference的源码
31 //private T referent; 这个才是软引用, 只被jvm使用
32 sample = null;
33
34 //可以重新获得Sample对象, 并用一个强引用指向它
35 sample = softRef.get();
36
37 new Thread(new Runnable() {
38     @Override
39     public void run() {
40         while (true) {
41             System.out.println(softRef.get());
42             try {
43                 Thread.sleep(1000);
44             } catch (InterruptedException e) {
45                 e.printStackTrace();
46             }
47             TEST_DATA.add(new byte[1024 * 1024 * 5]);
48         }
49     }
50 }).start();
51
52 new Thread(){
53     @Override
54     public void run() {
55         while (true) {
56             Reference<? extends Sample> poll = QUEUE.poll();
57             if (poll != null) {
58                 System.out.println("--- 软引用对象被jvm回收了 ---- " + poll);
59                 System.out.println("--- 回收对象 ---- " + poll.get());
60             }
61         }
62     }
63 }.start();
64 }
65
66

```

运行结果:

```

D:\jdk\jre\lib\resources.jar;D:\jdk\jre\lib\rt.jar;E:\Java_Demo\jvm\target\classes;com
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
com.jvm.references.Sample@4bde525f
null
--- 软引用对象被jvm回收了 ---- java.lang.ref.SoftReference@220af032
--- 回收对象 ---- null
null
Exception in thread "Thread-0" java.lang.OutOfMemoryError: Java heap space
---at com.jvm.references.Test2$1.run(Test2.java:47)
---at java.lang.Thread.run(Thread.java:748)

```

3.3.3 弱引用

弱引用会被jvm忽略，也就说在GC进行垃圾收集的时候，如果一个对象只有弱引用指向它，那么和没有引用指向它是一样的效果，jvm都会对它就行果断的销毁，释放内存。其实这个特性是很有用的，jdk也提供了java.util.WeakHashMap这么一个key为弱引用的Map。比如某个资源对象你要释放(比如 db connection), 但是如果被其它map作为key强引用了，就无法释放，被jvm收集。

```

1 |kReference<Sample> softRef = new weakReference<Sample>(sample, QUEUE);

```

结果：

```

com.jvm.references.Sample@5c570a66
com.jvm.references.Sample@5c570a66
com.jvm.references.Sample@5c570a66
com.jvm.references.Sample@5c570a66
com.jvm.references.Sample@5c570a66
com.jvm.references.Sample@5c570a66
com.jvm.references.Sample@5c570a66
com.jvm.references.Sample@5c570a66
com.jvm.references.Sample@5c570a66
com.jvm.references.Sample@5c570a66
--- 弱引用对象被jvm回收了 ---- null
Exception in thread "Thread-1" java.lang.NullPointerException
null
at com.jvm.references.Test3$2.run(Test3.java:58)
null
null
null
null

```

3.3.4 虚引用

虚引用和弱引用的回收机制差不多，都是可以被随时回收的。但是不同的地方是，它的构造方法必须强制传入ReferenceQueue，因为在jvm回收前(重点：对，就是回收前，软引用和弱引用都是回收后)，会将PhantomReference对象加入ReferenceQueue中；还有一点就是PhantomReference.get()方法永远返回空，不管对象有没有被回收。

```

1  package com.jvm.references;
2
3  import java.lang.ref.PhantomReference;
4  import java.lang.ref.Reference;
5  import java.lang.ref.ReferenceQueue;
6  import java.lang.ref.WeakReference;
7  import java.util.ArrayList;
8  import java.util.List;
9
10
11  Author: 98050
12  Time: 2019-01-11 21:53
13  Feature: 虚引用
14
15  public class Test4 {
16      private static final List<Object> TEST_DATA = new ArrayList<>();
17
18      private static final ReferenceQueue<Sample> QUEUE = new ReferenceQueue<>();
19
20
21      public static void main(String[] args) throws InterruptedException {
22          //创建一个对象，new出来的对象都是分配在java堆中的

```

```

23 Sample sample = new Sample(); //sample这个引用就是强引用
24
25 //创建一个弱引用指向这个对象 那么此时就有两个引用指向Sample对象
26 PhantomReference<Sample> softRef = new PhantomReference<Sample>(sample, QUEUE);
27
28 //将强引用指向空指针 那么此时只有一个软引用指向Sample对象
29 //注意: softRef这个引用也是强引用, 它是指向SoftReference这个对象的
30 //那么这个软引用在哪呢? 可以跟一下java.lang.Reference的源码
31 //private T referent; 这个才是软引用, 只被jvm使用
32 sample = null;
33
34 //可以重新获得Sample对象, 并用一个强引用指向它
35 sample = softRef.get();
36
37 new Thread(new Runnable() {
38     @Override
39     public void run() {
40         while (true) {
41             System.out.println(softRef.get());
42             try {
43                 Thread.sleep(1000);
44             } catch (InterruptedException e) {
45                 e.printStackTrace();
46             }
47             TEST_DATA.add(new byte[1024 * 1024 * 5]);
48         }
49     }
50 }).start();
51
52 new Thread(){
53     @Override
54     public void run() {
55         while (true) {
56             Reference<? extends Sample> poll = QUEUE.poll();
57             if (poll != null) {
58                 System.out.println("--- 虚幻引用对象被jvm回收了 ---- " + poll);
59                 System.out.println(poll.isEnqueued());
60                 System.out.println("--- 回收对象 ---- " + poll.get());
61             }
62         }
63     }
64 }.start();
65 }
66
67

```



```
null
null
null
null
null
null
null
null
null
null
null
null
--- 虚幻引用对象被jvm回收了 ---- java.lang.ref.PhantomReference@78541db1
false
--- 回收对象 ---- null
null
null
null
null
null
null
```

3.4 真正的死亡

Java技术使用finalize()方法在垃圾收集器将对象从内存中清除出去前，做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在Object类中定义的，因此所有的类都继承了它。子类覆盖finalize()方法以整理系统资源或者执行其他清理工作。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

即使在可达性分析算法中的不可达对象，也并非式“非死不可”的。要真正宣告一个对象的死亡，至少要经历两次标记过程：

标记的前提是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。

1)第一次标记并进行一次筛选

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”。

2)第二次标记

如果这个对象被判定为有必要执行finalize（）方法，那么这个对象将会被放置在一个名为：F-Queue的队列之中，并在稍后由一条虚拟机自动建立的、低优先级的Finalizer线程去执行。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finalize（）方法中执行缓慢，或者发生死循环（更极端的情况），将很可能会导致F-Queue队列中的其他对象永久处于等待状态，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的一个对象建立关联即可，譬如把自己赋值给某个类变量或对象的成员变量，那在第二次标记时它将移除出“即将回收”的集合。如果对象这时候还没逃脱，那基本上它就真的被回收了。

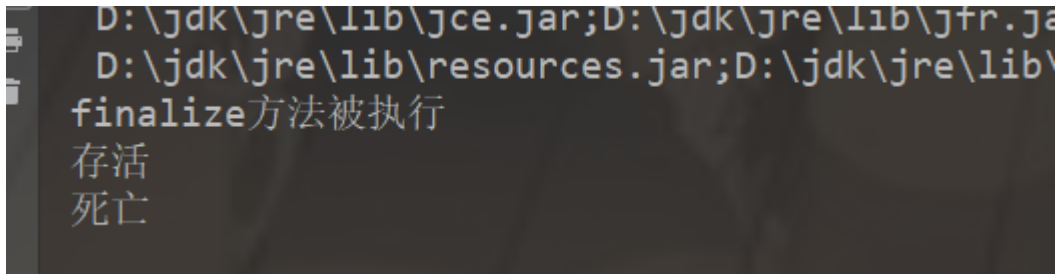
例子：

```
1 age com.jvm.finalize;
2
3
```



```
4 Author: 98050
5 Time: 2019-01-11 22:19
6 Feature: 1.对象可以在被GC时自我拯救 2.自救的机会只有一次, 因为一个对象的finalize()方法最多只会被系统调用一次
7
8 public class FinalizeEscapeGC {
9
10 public static FinalizeEscapeGC SAVE_HOOK = null;
11
12 public void isAlive(){
13     System.out.println("存活");
14 }
15
16 @Override
17 protected void finalize() throws Throwable {
18     super.finalize();
19     System.out.println("finalize方法被执行");
20     //自我拯救
21     FinalizeEscapeGC.SAVE_HOOK = this;
22 }
23
24 public static void main(String[] args) throws InterruptedException {
25     SAVE_HOOK = new FinalizeEscapeGC();
26
27     //第一次拯救
28     SAVE_HOOK = null;
29     System.gc();
30     //因为finalize方法优先级很低, 所以暂停0.5秒
31     Thread.sleep(500);
32     if (SAVE_HOOK != null){
33         SAVE_HOOK.isAlive();
34     }else {
35         System.out.println("死亡");
36     }
37
38     //第二次拯救
39     SAVE_HOOK = null;
40     System.gc();
41     //因为finalize方法优先级很低, 所以暂停0.5秒
42     Thread.sleep(500);
43     if (SAVE_HOOK != null){
44         SAVE_HOOK.isAlive();
45     }else {
46         System.out.println("死亡");
47     }
48 }
49
50
```

结果:



任何一个对象的finalize()方法都只会被系统自动调用一次。

3.5 回收方法区

方法区的垃圾回收“性价比”比较低，在堆中，尤其是在新生代中，常规应用进行一次垃圾收集一般可以回收70%~95%的空间。

永久代主要回收两部分内容：**废弃常量**和**无用类**。

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。请参考下Java8：从永久代到元数据区(注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)

四、垃圾回收算法

4.1 Java堆的划分

Java 中的堆是 JVM 所管理的最大的一块内存空间，主要用于存放各种类的实例对象。

在 Java 中，堆被划分成两个不同的区域：新生代 (Young)、老年代 (Old)。新生代 (Young) 又被划分为三个区域：Eden、From Survivor、To Survivor。

这样划分的目的是为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。

堆的内存模型大致为：



默认的，新生代 (Young) 与老年代 (Old) 的比例的值为 1:2 (该值可以通过参数 -XX:NewRatio 来指定)，即：新生代 (Young) = 1/3 的堆空间大小。老年代 (Old) = 2/3 的堆空间大小。

其中，新生代 (Young) 被细分为 Eden 和两个 Survivor 区域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分。默认的，Eden : from : to = 8 : 1 : 1 (可以通过参数 -XX:SurvivorRatio 来设定)，即：Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。

根据垃圾回收机制的不同，Java堆有可能拥有不同的结构，最为常见的就是将整个Java堆分为新生代和老年代。其中**新生代存放新生的对象或者年龄不大的对象，老年代则存放老年对象。**

新生代分为den区、s0区、s1区，s0和s1也被称为from和to区域，他们是两块大小相等并且可以互相角色的空间。

绝大多数情况下，对象首先分配在eden区，在新生代回收后，如果对象还存活，则进入s0或s1区，之后每经过一次新生代回收，如果对象存活则它的年龄就加1，对象达到一定的年龄后，则进入老年代。

可以通过配置以下参数改变整个JVM堆的配置比例：

```
1 |x20m -Xms20m
```

4.2 垃圾回收算法

4.2.1 标记-清除算法

概念

该算法有两个阶段。

1. 标记阶段：找到所有可访问的对象，做个标记
2. 清除阶段：遍历堆，把未被标记的对象回收

应用场景

该算法一般应用于老年代,因为老年代的对象生命周期比较长。

优缺点

标记清除算法的优点和缺点

1. 优点
 - 是可以解决循环引用的问题
 - 必要时才回收(内存不足时)
2. 缺点
 - 回收时，应用需要挂起，也就是stop the world。
 - 标记和清除的效率不高，尤其是要扫描的对象比较多的时候。
 - 空间问题，标记清除后会产生大量不连续的内存碎片，空间碎片太多可能导致以后在程序运行过程中需要分配大对象时，无法找到足够的连续内存而不得不提前触发一次垃圾收集。

4.2.2 复制算法

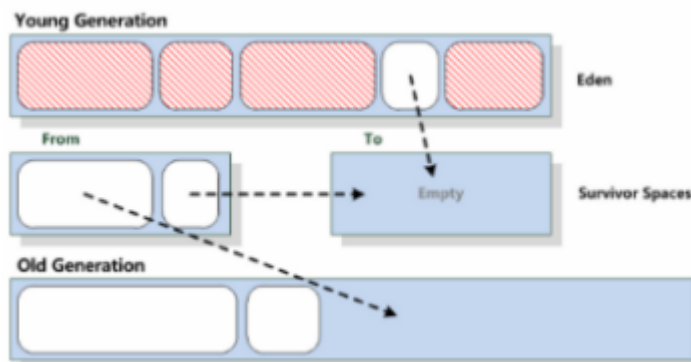
概念

如果jvm使用了coping算法，一开始就会将可用内存分为两块，from域和to域，每次只是使用from域，to域则空闲着。当from域内存不够了，开始执行GC操作，这个时候，会把from域存活的对象拷贝到to域,然后直接把from域进行内存清理。

应用场景

复制算法一般是使用在新生代中，因为新生代中的对象一般都是朝生夕死的，存活对象的数量并不多，这样使用复制算法进行拷贝时效率比较高。jvm将Heap 内存划分为新生代与老年代，又将新生代划分为Eden(伊甸园) 与2块 Survivor Space(幸存者区) ,然后在Eden ->Survivor Space 以及From Survivor Space 与To Survivor Space 之间实行 Copying 算法。不过jvm在应用coping算法时，并不是把内存按照1:1来划分的，这样太浪费内存空间了。一般的jvm都是8:1 。也就是每次新生代中可用空间为整个新生代容量的90%。Eden区:From区:To区域的比例是 8:1:1

始终有90%的空间是可以用来创建对象的,而剩下的10%用来存放回收后存活的对象。



1、当Eden区满的时候,会触发第一次young gc,把还活着的对象拷贝到Survivor From区；当Eden区再次触发young gc的时候,会扫描Eden区和From区域,对两个区域进行垃圾回收,经过这次回收后还存活的对象,则直接复制到To区域,并将Eden和From区域清空（90%）。

2、当后续Eden又发生young gc的时候,会对Eden和To区域进行垃圾回收,存活的对象复制到From区域,并将Eden和To区域清空。

3、可见部分对象会在From和To区域中复制来复制去,如此交换15次(由JVM参数MaxTenuringThreshold决定,这个参数默认是15),最终如果还是存活,就存入到老年代

注意: 万一存活对象数量比较多，那么To域的内存可能不够存放，这个时候会借助老年代的空间。

优缺点

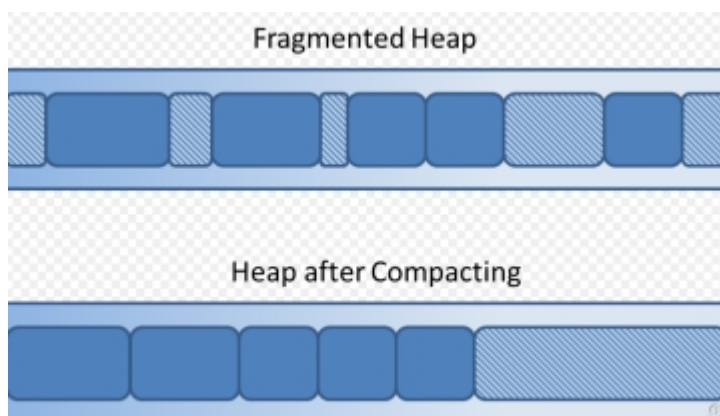
优点:在存活对象不多的情况下，性能高，能解决内存碎片和java垃圾回收算法之-标记清除 中导致的引用更新问题。

缺点: 会造成一部分的内存浪费。不过可以根据实际情况，将内存块大小比例适当调整；**如果存活对象的数量比较大，coping的性能会变得很差。**

4.3.3 标记-整理算法

标记清除算法和标记整理算法非常相同，但是标记整理算法在标记清除算法之上解决内存碎片化

概念



整理算法简单介绍

任意顺序：即不考虑原先对象的排列顺序，也不考虑对象之间的引用关系，随意移动对象；

线性顺序：考虑对象的引用关系，例如a对象引用了b对象，则尽可能将a和b移动到一块；

滑动顺序：按照对象原来在堆中的顺序滑动到堆的一端。

优缺点

优点:解决内存碎片问题

缺点：压缩阶段，由于移动了可用对象，**需要去更新引用。**

知识点

到底什么是堆？

4.3.4 分代收集算法

新生代：复制算法（每次垃圾回收时都会发现大量对象死亡，只有少量存活）

老年代：标记整理、标记清除（对象存活率高）

4.3 Minor GC和Full GC区别

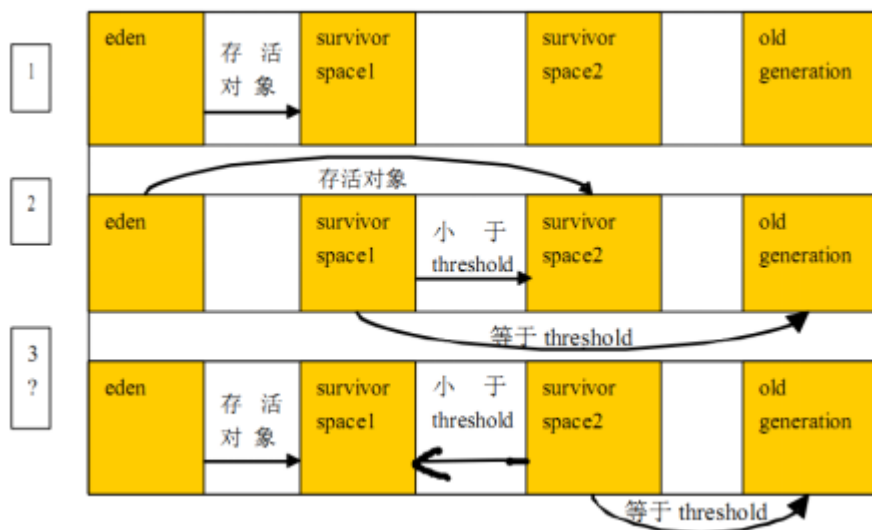
新生代 GC (Minor GC)：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。

老年代 GC (Major GC / Full GC)：指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC（但非绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程）。MajorGC 的速度一般会比 Minor GC 慢 10倍以上。

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。

Full GC触发机制：当老年代满时会引发Full GC，Full GC将会同时回收年轻代、年老代，当永久代满时也会引发Full GC，会导致Class、Method元信息的卸载其中Minor GC

虚拟机给每个对象定义了一个对象年龄（Age）计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1。对象在 Survivor 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁）时，就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold (阈值)来设置。



4.4 Client模式和Server模式下的GC算法区别

client模式下，新生代选择的是串行gc，老年代选择的是串行gc server模式下，新生代选择的是并行回收gc，老年代选择的是并行gc

五、JVM参数配置

5.1 常见参数配置

```

1 :+PrintGC          每次触发GC的时候打印相关日志
2 :+UseSerialGC      串行回收
3 :+PrintGCDetails   更详细的GC日志
4 s                  堆初始值
5 x                  堆最大可用值
6 n                  新生代堆最大可用值
7 :SurvivorRatio      用来设置新生代中eden空间和from/to空间的比例。
8 //-XX:SurvivorRatio=eden/from=eden/to
9 :NewRatio           配置新生代与老年代占比
  
```

总结:在实际工作中，可以直接将初始的堆大小与最大堆大小相等，这样的好处是可以减少程序运行时垃圾回收次数，从而提高效率。

5.2 堆内存大小配置

```

1 x20m -xms5m        当下Java应用最大可用内存为20M， 初始内存为5M
  
```

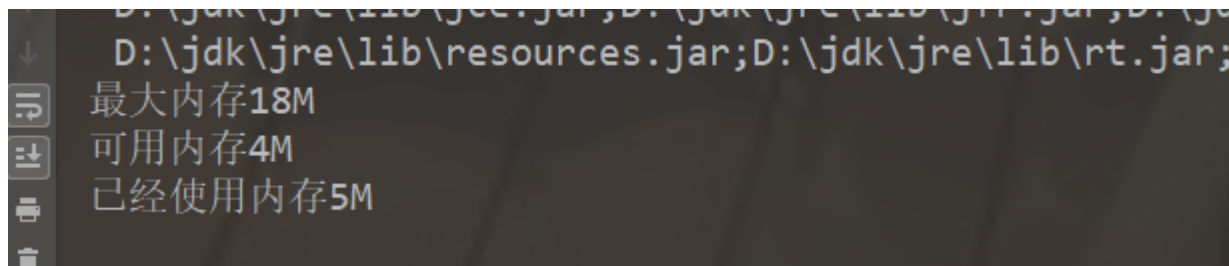
```

1 age com.jvm.propertysetting;
2
3
4 Author: 98050
5 Time: 2019-01-14 19:28
6 Feature: 堆内存大小配置
  
```

```

7 Xmx20m -Xms5m
8
9 ic class Heap {
10 public static void main(String[] args) {
11     System.out.print("最大内存");
12     System.out.println(Runtime.getRuntime().maxMemory() / 1024 / 1024 + "M");
13     System.out.print("可用内存");
14     System.out.println(Runtime.getRuntime().freeMemory() / 1024 / 1024 + "M");
15     System.out.print("已经使用内存");
16     System.out.println(Runtime.getRuntime().totalMemory() / 1024 / 1024 + "M");
17 }
18

```



D:\jdk\jre\lib\resources.jar;D:\jdk\jre\lib\rt.jar;
 最大内存18M
 可用内存4M
 已经使用内存5M

存在一点误差。

5.3 设置新生代比例参数

```

1 s20m -Xmx20m -Xmn1m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC
2 说明: 堆内存初始化值20m,堆内存最大值20m, 新生代最大值可用1m, eden空间和from/to空间的比例为2/1
3 XX:SurvivorRatio=2          Survivor:Eden = 2:2

```

```

1 age com.jvm.propertysetting;
2
3
4 Author: 98050
5 Time: 2019-01-14 19:44
6 Feature: 设置新生代比例参数
7 Xms20m -Xmx20m -Xmn1m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC
8
9 ic class Young {
10 public static void main(String[] args) {
11     byte[] b = null;
12     for (int i = 0; i < 10; i++) {
13         b = new byte[1 * 1024 * 1024];
14     }
15 }
16
17

```



```

- jar, D:\jdk\jre\lib\rt.jar, D:\java_demo\jvm\target\classes\com\jvm\propertysetting\Young
[GC (Allocation Failure) [DefNew: 511K->256K(768K), 0.0305082 secs] 511K->432K(20224K), 0.0305562 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
[GC (Allocation Failure) [DefNew: 766K->170K(768K), 0.0013377 secs] 942K->601K(20224K), 0.0013581 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [DefNew: 676K->55K(768K), 0.0006439 secs] 1108K->655K(20224K), 0.0006625 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation      total 768K, used 501K [0x00000000fec00000, 0x00000000fed00000, 0x00000000fed00000)
  eden space 512K,  87% used [0x00000000fec00000, 0x00000000fec6f668, 0x00000000fec80000)
  from space 256K,  21% used [0x00000000fec00000, 0x00000000feccdf30, 0x00000000fed00000)
  to   space 256K,   0% used [0x00000000fec80000, 0x00000000fec80000, 0x00000000fec00000)
 tenured generation      total 19456K, used 10840K [0x00000000fed00000, 0x0000000100000000, 0x0000000100000000)
  the space 19456K,   55% used [0x00000000fed00000, 0x00000000ff796000, 0x00000000ff796000, 0x0000000100000000)
 Metaspace               used 3452K, capacity 4496K, committed 4864K, reserved 1056768K
 class space              used 376K, capacity 388K, committed 512K, reserved 1048576K
Process finished with exit code 0

```

5.4 设置新生代与老年代比例参数

```

1  s20m -Xmx20m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC
2  :NewRatio=2
3  说明：堆内存初始值20m,堆内存最大值20m, 新生代最大值可用1m, eden空间和from/to空间的比例为2/1, 新生代和
   老年代的占比为1/2
4  XX:NewRatio=2                新生代:老年代=1:2, 新生代占堆的1/3

```

```

1  age com.jvm.propertysetting;
2
3
4  Author: 98050
5  Time: 2019-01-14 20:09
6  Feature: 设置新生代与老年代比例参数
7  -Xms20m -Xmx20m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC
8  XX:NewRatio=2
9
10  ic class YoungVsOld {
11  public static void main(String[] args) {
12      byte[] b = null;
13      for (int i = 0; i < 10; i++) {
14          b = new byte[1 * 1024 * 1024];
15      }
16  }
17

```

```

[GC (Allocation Failure) [DefNew: 2926K->1664K(5120K), 0.0017055 secs] 2926K->1711K(18816K), 0.00
[GC (Allocation Failure) [DefNew: 4838K->1026K(5120K), 0.0016315 secs] 4835K->1713K(18816K), 0.00
[GC (Allocation Failure) [DefNew: 4180K->1024K(5120K), 0.0003086 secs] 4867K->1713K(18816K), 0.00
Heap
 def new generation      total 5120K, used 4216K [0x00000000fec00000, 0x00000000ff2a0000, 0x00000000
  eden space 3456K,  92% used [0x00000000fec00000, 0x00000000fef1e078, 0x00000000fef60000)
  from space 1664K,  61% used [0x00000000ff100000, 0x00000000ff200110, 0x00000000ff2a0000)
  to   space 1664K,   0% used [0x00000000fef60000, 0x00000000fef60000, 0x00000000ff100000)
 tenured generation      total 13696K, used 689K [0x00000000ff2a0000, 0x0000000100000000, 0x00000001
  the space 13696K,   5% used [0x00000000ff2a0000, 0x00000000ff34c580, 0x00000000ff34c600, 0x000
 Metaspace               used 3365K, capacity 4496K, committed 4864K, reserved 1056768K
 class space              used 369K, capacity 388K, committed 512K, reserved 1048576K

```

六、垃圾收集器

串行回收: JDK1.5前的默认算法 缺点是只有一个线程, 执行垃圾回收时程序停止的时间比较长

并行回收: 多个线程执行垃圾回收适合于吞吐量的系统, 回收时系统会停止运行

6.1 Serial收集器

最基本、发展历史最悠久的收集器, 是一个 **单线程** 收集器。进行垃圾收集时必须暂停其它所有工作线程, 直到收集结束。

特点

- 针对**新生代**;
- 采用**复制算法**;
- 单线程收集;
- 进行垃圾收集时, 必须暂停所有工作线程, 直到完成;

应用场景

依然是HotSpot在Client模式下默认的新生代收集器;

优点

简单高效 (与其他收集器的单线程相比) ;

对于限定单个CPU的环境来说, Serial收集器没有线程交互 (切换) 开销, 可以获得最高的单线程收集效率;

在用户的桌面应用场景中, 可用内存一般不大 (几十M至一两百M), 可以在较短时间内完成垃圾收集 (几十MS至一百多MS), 只要不频繁发生, 这是可以接受的

设置参数

"-XX:+UseSerialGC": 添加该参数来显式的使用串行垃圾收集器;

6.2 ParNew收集器

除了多线程外, 其余的行为、特点和Serial收集器一样; 如Serial收集器可用控制参数、收集算法、Stop The World、内存分配规则、回收策略等; 两个收集器共用了不少代码;

特点

- **多线程**
- **并行收集**

应用场景

在Server模式下, ParNew收集器是一个非常重要的收集器, 因为除Serial外, 目前只有它能与**CMS收集器配合工作**; 但在单个CPU环境中, 不会比Serial收集器有更好的效果, 因为存在线程交互开销。

设置参数

"-XX:+UseConcMarkSweepGC": 指定使用CMS后, 会默认使用ParNew作为新生代收集器; "-

XX:+UseParNewGC": 强制指定使用ParNew;

"-XX:ParallelGCThreads": 指定垃圾收集的线程数量, ParNew默认开启的收集线程与CPU的数量相同;

6.3 Parallel Scavenge收集器

吞吐量优先收集器。Parallel Scavenge收集器的目标是达到一个可控制的吞吐量，所谓吞吐量就是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)

特点

- 针对新生代
- 使用复制算法
- 多线程
- 并行收集
- 关注吞吐量
- 自适应策略

应用场景

- 主要适合在后台运算而不需要太多交互的任务

优点

- 高效地利用CPU时间，尽快完成程序的运算任务

设置参数

-XX:MaxGCPauseMillis：（大于0的毫秒数）控制最大垃圾收集停顿时间，收集器将尽可能地保证内存回收花费的时间不超过设定值。**GC停顿时间的缩短是以牺牲吞吐量和新生代空间换来的。**

-XX:GCTimeRatio：设置吞吐量大小（0~100）。垃圾收集时间占总时间的比率：1/（1+n）n为设置的参数

-XX:+UseAdaptiveSizePolicy：是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）。

6.4 Serial Old收集器

Serial收集器的老年代版本

特点

- 单线程
- 并行收集
- 使用标记-整理算法

应用场景

- Client模式下的虚拟机使用
- Server模式下：在JDK1.5之前的版本中搭配Parallel Scavenge收集器使用；作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。

6.5 Parallel Old收集器

Parallel Scavenge收集器的老年代版本

特点

- 多线程
- 并行收集

- 使用标记-整理算法

应用场景

- 注重吞吐量及CPU资源敏感的场所，可以优先考虑Parallel Scavenge + Parallel Old

优点

- 吞吐量优先

设置参数

-XX:+UseParallelOldGC

6.6 CMS收集器

CMS收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为4个步骤，包括：

- 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)

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行。

老年代收集器（新生代使用ParNew）

特点

- 以获取最短回收停顿时间为目标
- **多线程**
- **并发收集**

应用场景

- 网站
- B/S系统的服务端

缺点

- CMS收集器对CPU资源非常敏感，并发阶段吞吐量会降低
- CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败导致Full GC的产生。
- 会产生大量碎片

设置参数

-XX:+UseConcMarkSweepGC

-XX:CMSInitiatingOccupancyFraction = 70 CMS收集器不能像其它收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运行使用。老年代使用70%后开始CMS收集。但是不能设置的太高，因为这样留给程序运行的内存就变少了，就会出现“Concurrent Mode Failure”，这时就会启用Serial Old进行老年代垃圾的回收，等待时间更长。

-XX:CMSFullGCsBeforeCompaction：设置执行多少次不压缩的Full GC后，来一次带压缩的

6.7 G1收集器

运行步骤：

- 初始标记
- 并发标记
- 最终标记
- 筛选回收

特点

- 并行与并发
 - 在主线程暂停的情况下，使用并行收集
 - 在主线程运行的情况下，使用并发收集
- 支持多CPU和垃圾回收线程
- 分代收集
- 空间整合：整体上基于标记-整理算法，Region之间基于复制算法
- 可预测的停顿：建立可预测的停顿时间模型，可以指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。（G1跟踪各个Region里面的垃圾堆积价值大小，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region）
- 将整个堆划分为多个大小相等的独立区域（Region）
- 支持很大的堆，高吞吐量

设置参数

-XX:+UseG1GC

七、内存分配策略

7.1 对象优先在Eden分配

大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够空间进行分配时，虚拟机会发起一次Minor GC。

7.2 大对象直接进入老年代

通过-XX:PretenureSizeThreshold参数设置，令大于这个设置值的对象直接在老年代分配。避免在Eden区和两个Survivor区发生大量的内存复制。

这个参数只对Serial和ParNew两款收集器有效。

7.3 长期存活的对象将进入老年代

当对象年龄达到-XX:MaxTenuringThreshold设置的值时，就进入老年代，默认为15。

7.4 动态对象年龄判定

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

7.5 空间分配担保

发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。

如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC。

下面解释一下“冒险”是冒了什么风险，前面提到过，新生代使用复制收集算法，但为了内存利用率，只使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor GC后仍然存活的情况（最极端的情况就是内存回收后新生代中所有对象都存活），就需要老年代进行分配担保，把Survivor无法容纳的对象直接进入老年代。与生活中的贷款担保类似，老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，一共有多少对象会活下来在实际完成内存回收之前是无法明确知道的，所以只好取之前每一次回收晋升到老年代对象容量的平均大小值作为经验值，与老年代的剩余空间进行比较，决定是否进行Full GC来让老年代腾出更多空间。

取平均值进行比较其实仍然是一种动态概率的手段，也就是说，如果某次Minor GC存活后的对象突增，远远高于平均值的话，依然会导致担保失败（Handle Promotion Failure）。如果出现了HandlePromotionFailure失败，那就只好在失败后重新发起一次Full GC。虽然担保失败时绕的圈子是最大的，但大部分情况下都还是会将HandlePromotionFailure开关打开，避免Full GC过于频繁。

八、JDK可视化工具

jconsole

从Java 5开始 引入了JConsole。JConsole 是一个内置Java 性能分析器，可以从命令行或在 GUI shell 中运行。您可以轻松地使用JConsole（或者，它更高端的“近亲”VisualVM）来监控Java 应用程序性能和跟踪Java 中的代码。

如果是从命令行启动，使JDK在PATH上，运行jconsole即可。

如果从GUI shell启动，找到JDK安装路径，打开bin文件夹，双击jconsole。

当分析工具弹出时（取决于正在运行的Java版本以及正在运行的Java程序数量），可能会出现一个对话框，要求输入一个进程的URL来连接，也可能列出许多不同的本地Java进程（有时包含JConsole进程本身）来连接。

visualVm

VisualVM是一款免费的，集成了多个JDK命令行工具的可视化工具，它能为您提供强大的分析能力，对Java应用程序做性能分析和调优。这些功能包括生成和分析海量数据、跟踪内存泄漏、监控垃圾回收器、执行内存和CPU分析，同时它还支持在MBeans上进行浏览和操作。本文主要介绍如何使用VisualVM进行性能分析及调优。

九、字节码技术

十、类加载机制

10.1 类加载的时机

Java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) 七个阶段。其中准备、验证、解析3个部分统称为连接（Linking）：



10.2 类加载过程

10.2.1 加载

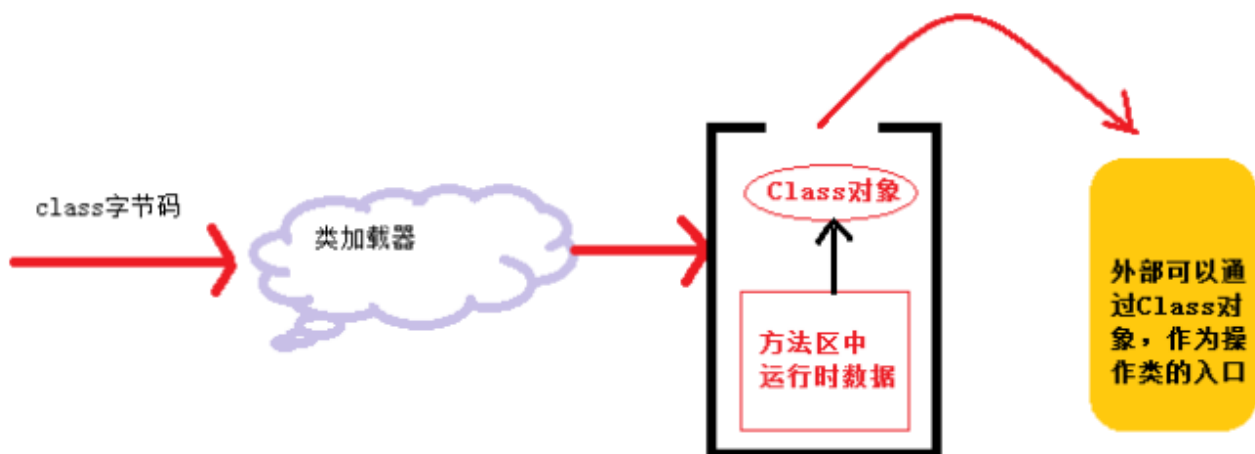
加载阶段，虚拟机要完成3件事情：

- 通过一个类的全限定名来获取定义此类的二进制字节流
- 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
- 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口

当系统运行时，**类加载器将.class文件的二进制数据从外部存储器（如光盘，硬盘）调入内存中**，CPU再从内存中读取指令和数据进行运算，并将运算结果存入内存中。内存在此过程中充当着"二传手"的作用，通俗的讲，如果没有内存，类加载器从外部存储设备调入.class文件二进制数据直接给CPU处理，而由于CPU的处理速度远远大于调入数据的速度，容易造成数据的脱节，所以需要内存起缓冲作用。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在**方法区**之中，方法区中的数据存储格式由虚拟机实现自行定义。然后在内存中实例化一个java.lang.Class类的对象（并没有明确规定是在Java堆中，对于HotSpot虚拟机而言，Class对象比较特殊，它虽然是对象，但是存放在方法区里面），这个对象将作为程序访问方法区中的这些类型数据的外部接口。

类将.class文件加载至运行时的方法区后，会在**堆**中创建一个java.lang.Class对象，用来封装类位于方法区内的数据结构，该Class对象是在加载类的过程中创建的，每个类都对应有一个Class类型的对象，Class类的构造方法是私有的，只有JVM能够创建。因此Class对象是反射的入口，使用该对象就可以获得目标类所关联的.class文件中具体的数据结构。



类加载的最终产物就是位于堆中的Class对象（注意不是目标类对象），该对象封装了类在方法区中的数据结构，并且向用户提供了访问方法区数据结构的接口，即Java反射的接口。

10.2.2 连接过程

将java类的二进制代码合并到JVM的运行状态之中的过程

- 验证：确保加载的类信息符合JVM规范，没有安全方面的问题
- 准备：正式为类变量（static变量）分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配
- 解析：虚拟机常量池的符号引用替换为字节引用过程

10.2.3 初始化

类初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才是真正开始执行类中定义的Java程序代码（字节码）。

初始化阶段是执行类构造器（）方法的过程。类构造器（）方法是由编译器自动收藏类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生，代码从上往下执行。

当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化

虚拟机会保证一个类的（）方法在多线程环境中被正确加锁和同步

当范围一个Java类的静态域时，只有真正声名这个域类才会被初始化

类初始化的时机（5种），即对一个类进行主动引用

1. 遇到new、getstatic、putstatic或invokestatic这四条字节码指令（注意在新建数组的时候要注意，被动引用）时，如果类没有进行过初始化，则需要先对其进行初始化。生成这四条指令的最常见的Java代码场景是：
 - 使用new关键字实例化对象的时候
 - 读取或设置一个类的静态字段（被final修饰，已在编译器把结果放入常量池的静态字段除外）的时候
 - 调用一个类的静态方法的时候
2. 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
3. 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
4. 当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

5. 当使用jdk1.7动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF_getstatic,REF_putstatic,REF_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

被动引用：

1. 通过子类引用父类的静态字段，不会导致子类初始化（对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类种定义的静态字段，只会触发父类的初始化而不会触发子类的初始化）
2. 通过数组定义来引用类，不会触发此类的初始化（比如，new String[]只会直接触发String[]类的初始化，也就是触发对类Ljava.lang.String的初始化，而直接不会触发String类的初始化）
3. 常量在编译阶段会进入常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化

10.3 类加载器

比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那么这两个类就一定不相等。这里说的相等包括代表类的Class对象的equals()方法、isAssignableFrom()方法、isInstance()方法的返回结果，也包括使用instanceof关键字做对象所属关系判读等情况。

10.3.1 加载器分类

启动 (Bootstrap) 类加载器

启动类加载器主要加载的是JVM自身需要的类，这个类加载使用C++语言实现的，是虚拟机自身的一部分，它负责将<JAVA_HOME>/lib路径下的核心类库或-Xbootclasspath参数指定的路径下的jar包加载到内存中，注意必由于虚拟机是按照文件名识别加载jar包的，如rt.jar，如果文件名不被虚拟机识别，即使把jar包丢到lib目录下也是没有作用的(出于安全考虑，**Bootstrap启动类加载器只加载包名为java、javax、sun等开头的类**)。

扩展 (Extension) 类加载器

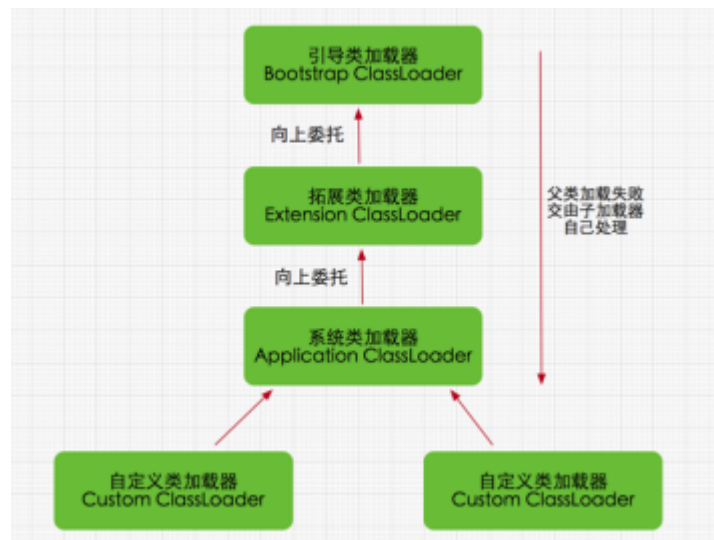
扩展类加载器是指Sun公司(已被Oracle收购)实现的sun.misc.Launcher\$ExtClassLoader类，由Java语言实现的，是Launcher的静态内部类，它负责加载<JAVA_HOME>/lib/ext目录下或者由系统变量-Djava.ext.dir指定路径中的类库，开发者可以直接使用标准扩展类加载器。

应用程序 (Application) 类加载器

也称应用程序加载器是指Sun公司实现的sun.misc.Launcher\$AppClassLoader。它负责加载系统类路径java -classpath或-D java.class.path 指定路径下的类库，也就是我们经常用到的classpath路径，开发者可以直接使用系统类加载器，一般情况下该类加载是程序中默认类加载器，通过ClassLoader#getSystemClassLoader()方法可以获取到该类加载器。如果程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认类加载器。

在Java的日常应用程序开发中，类的加载几乎是由上述3种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，需要注意的是，Java虚拟机对class文件采用的是按需加载的方式，也就是说当需要使用该类时才会将它的class文件加载到内存生成class对象，而且加载某个类的class文件时，Java虚拟机采用的是双亲委派模式即把请求交由父类处理，它一种任务委派模式。

10.3.2 双亲委派模型



双亲委派模式是在Java 1.2后引入的，其工作原理的是，如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式，即每个儿子都很懒，每次有活就丢给父亲去干，直到父亲说这件事我也干不了时，儿子自己想办法去完成，这不就是传说中的实力坑爹啊？那么采用这种模式有啥用呢？

采用双亲委派模式的好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，**通过这种层级关可以避免类的重复加载**，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。

其次是考虑到安全因素，java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。可能你会想，如果我们在classpath路径下自定义一个名为java.lang.SingleInterge类(该类是胡编的)呢？该类并不存在java.lang中，经过双亲委托模式，传递到启动类加载器中，由于父类加载器路径下并没有该类，所以不会加载，将反向委托给子类加载器加载，最终会通过系统类加载器加载该类。但是这样做是不允许，因为java.lang是核心API包，需要访问权限，强制加载将会报出如下异常

```
java.lang.SecurityException: Prohibited package name: java.lang
```

10.3.3 破坏双亲委派

OSGi实现模块化热部署的关键则是它自定义的类加载器机制实现的。每一个程序模块都有一个自己的类加载器，当需要更换一个Bundle时，就把Bundle连同类加载器一起换掉以实现代码的热替换。

10.3.4 类加载器常用方法

loadClass(String)

该方法加载指定名称（包括包名）的二进制类型，该方法在JDK1.2之后不再建议用户重写但用户可以直接调用该方法，loadClass()方法是ClassLoader类自己实现的，该方法中的逻辑就是双亲委派模式的实现，其源码如下，loadClass(String name, boolean resolve)是一个重载方法，resolve参数代表是否生成class对象的同时进行解析相关操作。

正如loadClass方法所展示的，当类加载请求到来时，先从缓存中查找该类对象，如果存在直接返回，如果不存在则交给该类加载去的父加载器去加载，倘若没有父加载则交给顶级启动类加载器去加载，最后倘若仍没有找到，则使用findClass()方法去加载（关于findClass()稍后会进一步介绍）。从loadClass实现也可以知道如果不想重新定义加载类的规则，也没有复杂的逻辑，只想在运行时加载自己指定的类，那么我们可以直接使用this.getClass().getClassLoader().loadClass("className")，这样就可以直接调用ClassLoader的loadClass方法获取到class对象。

findClass(String)

在JDK1.2之前，在自定义类加载时，总会去继承ClassLoader类并重写loadClass方法，从而实现自定义的类加载类，但是在JDK1.2之后已不再建议用户去覆盖loadClass()方法，而是建议把自定义的类加载逻辑写在findClass()方法中，从前面的分析可知，findClass()方法是在loadClass()方法中被调用的，当loadClass()方法中父加载器加载失败后，则会调用自己的findClass()方法来完成类加载，这样就可以保证自定义的类加载器也符合双亲委托模式。需要注意的是ClassLoader类中并没有实现findClass()方法的具体代码逻辑，取而代之的是抛出ClassNotFoundException异常，同时应该知道的是findClass方法通常是和defineClass方法一起使用的(稍后会分析)

defineClass(byte[] b, int off, int len)

defineClass()方法是用来将byte字节流解析成JVM能够识别的Class对象(ClassLoader中已实现该方法逻辑)，通过这个方法不仅能够通过class文件实例化class对象，也可以通过其他方式实例化class对象，如通过网络接收一个类的字节码，然后转换为byte字节流创建对应的Class对象，defineClass()方法通常与findClass()方法一起使用，一般情况下，在自定义类加载器时，会直接覆盖ClassLoader的findClass()方法并编写加载规则，取得要加载类的字节码后转换成流，然后调用defineClass()方法生成类的Class对象

resolveClass(Class<?> c)

使用该方法可以使用类的Class对象创建完成也同时被解析。前面我们说链接阶段主要是对字节码进行验证，为类变量分配内存并设置初始值同时将字节码文件中的符号引用转换为直接引用。

10.4 热部署

对于Java应用程序来说，热部署就是在运行时更新Java类文件。

10.4.1 热部署原理

想要知道热部署的原理，必须要了解java类的加载过程。一个java类文件到虚拟机里的对象，要经过如下过程：首先通过java编译器，将java文件编译成class字节码，类加载器读取class字节码，再将类转化为实例，对实例newInstance就可以生成对象。

类加载器ClassLoader功能，也就是将class字节码转换到类的实例。

在java应用中，所有的实例都是由类加载器，加载而来。

一般在系统中，类的加载都是由系统自带的类加载器完成，而且对于同一个全限定名的java类（如com.csjar.soc.HelloWorld），只能被加载一次，而且无法被卸载。

这个时候问题就来了，如果我们希望将java类卸载，并且替换更新版本的java类，该怎么做呢？

既然在类加载器中，java类只能被加载一次，并且无法卸载。那是不是可以直接把类加载器给换了？答案是可以的，我们可以自定义类加载器，并重写ClassLoader的findClass方法。想要实现热部署可以分以下三个步骤：

- 1、销毁该自定义ClassLoader
- 2、更新class类文件

3、创建新的ClassLoader去加载更新后的class类文件。

10.4.2 热部署与热加载

Java热部署与Java热加载的联系和区别

Java热部署与热加载的联系

- 1.不重启服务器编译/部署项目
- 2.基于Java的类加载器实现

Java热部署与热加载的区别

- 部署方式
 - 热部署在服务器运行时重新部署项目
 - 热加载在运行时重新加载class
- 实现原理
 - 热部署直接重新加载整个应用
 - 热加载在运行时重新加载class
- 使用场景
 - 热部署更多的是在生产环境使用
 - 热加载则更多的实在开发环境使用

10.4.3 实现

目标加载类1.0:

```
1  age com.example;  
2  
3  
4  Author: 98050  
5  Time: 2019-02-22 15:27  
6  Feature:  
7  
8  ic class User {  
9  
10 public void add(){  
11     System.out.println("user_____1.0");  
12 }  
13
```

编译好的替换类2.0 (.class文件，通过反编译工具查看)：

```

1  package com.example;
2
3  import java.io.PrintStream;
4
5  public class User
6
7  {
8      public void add()
9      {
10         System.out.println("user_____2.0");
11     }
12 }

```

自定义类加载器：

```

1  package com.example;
2
3  import java.io.IOException;
4  import java.io.InputStream;
5
6
7  Author: 98050
8  Time: 2019-02-22 17:42
9  Feature: 自定义类加载器
10
11 public class MyClassLoader extends ClassLoader {
12
13     @Override
14     protected Class<?> findClass(String name) throws ClassNotFoundException {
15         try {
16             //文件名称
17             String fileName = name.substring(name.lastIndexOf('.') + 1) + ".class";
18             //获取文件输入流
19             InputStream inputStream = this.getClass().getResourceAsStream(fileName);
20             //读取字节
21             byte[] bytes = new byte[inputStream.available()];
22             inputStream.read(bytes);
23             //将bytes解析成jvm可以识别的Class对象
24             return defineClass(name, bytes, 0, bytes.length);
25         } catch (IOException e) {
26             e.printStackTrace();
27         }
28         return super.findClass(name);
29     }
30 }

```

测试类：

```

1  package com.example;
2
3  import java.lang.reflect.InvocationTargetException;
4  import java.lang.reflect.Method;
5

```

```

6
7 Author: 98050
8 Time: 2019-02-22 17:39
9 Feature:
10
11 ic class HotSwap {
12
13 public static void main(String[] args) throws ClassNotFoundException,
    ClassNotFoundException, InvocationTargetException, InstantiationException,
    IllegalAccessException, InterruptedException {
14     User user = new User();
15     user.add();
16     loadUser();
17     Thread.sleep(10 * 1000);
18     loadUser();
19 }
20
21 public static void loadUser() throws ClassNotFoundException, IllegalAccessException,
    antiationException, NoSuchMethodException, InvocationTargetException {
22     MyClassLoader loader = new MyClassLoader();
23     Class<?> loaderClass = loader.findClass("com.example.User");
24     Object o = loaderClass.newInstance();
25     Method method = loaderClass.getMethod("add");
26     method.invoke(o);
27     System.out.println(o.getClass());
28     System.out.println(o.getClass().getClassLoader());
29 }
30

```

在线程睡眠期间，将要替换的类替换target目录下的class文件