day20_NIO和JVM

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.掌握缓冲流的基本过程实现
2.掌握缓冲流的实现原理
3.掌握标准流的基本使用
4.掌握转换流的基本使用
5.掌握打印流的基本使用
6.掌握序列化流的基本使用
7.了解装饰设计模式
8.了解编码问题
9.了解Properties的使用
```

1.2 教学难点:

```
1.装饰设计模式的原理
2.编码问题的理解
```

二 教学目标

```
1.掌握NIO的原理
2.掌握NIO的基本使用
3.掌握JVM的工作原理
4.掌握JVM各区块的基本工作原理
5.掌握堆区的分区使用
6.掌握堆内存优化
```

三 教学导读

3.1. NIO

3.1.1. 概念

NIO： New IO。 Non-Blocking IO。

NIO 是JDK1.4的时候出现了一个新的IO， 用来替代传统的IO流。 NIO与IO有着相同的功能， 但是操作的方法不同。Java提供了一些改进输入/输出处理的新功能,这些新功能被统称为新IO(New IO 简称NIO),新增了许多用于处理输入/输出的类,这些类都被放在java.nio包以及子包下,并对原java.io中的很多类都以NIO为基础进行改写,新增了满足NIO的功能

NIO是基于通道(Channel)， 面向缓冲区(Buffer)的。

在JDK1.7的时候， 为NIO添加了一些新的特定。 被称为 NIO.2

Java NIO 由以下几个核心部分组成:

Channels：通道 Buffer：缓冲区 Selectors：选择器

Channels和Buffer是新IO中的两个核心对象,Channel是对传统的输入/输出系统的模拟,在新IO系统中所有的数据都需要通过通道传输。

Channels与传统的InputStream,OutputStrem最大的区别在于它提供了一个map()方法,通过该map()方法可以直接将"一块数据"映射到

内存中,如果说传统的输入/输出系统是面向流处理,则新IO则是面向块的处理

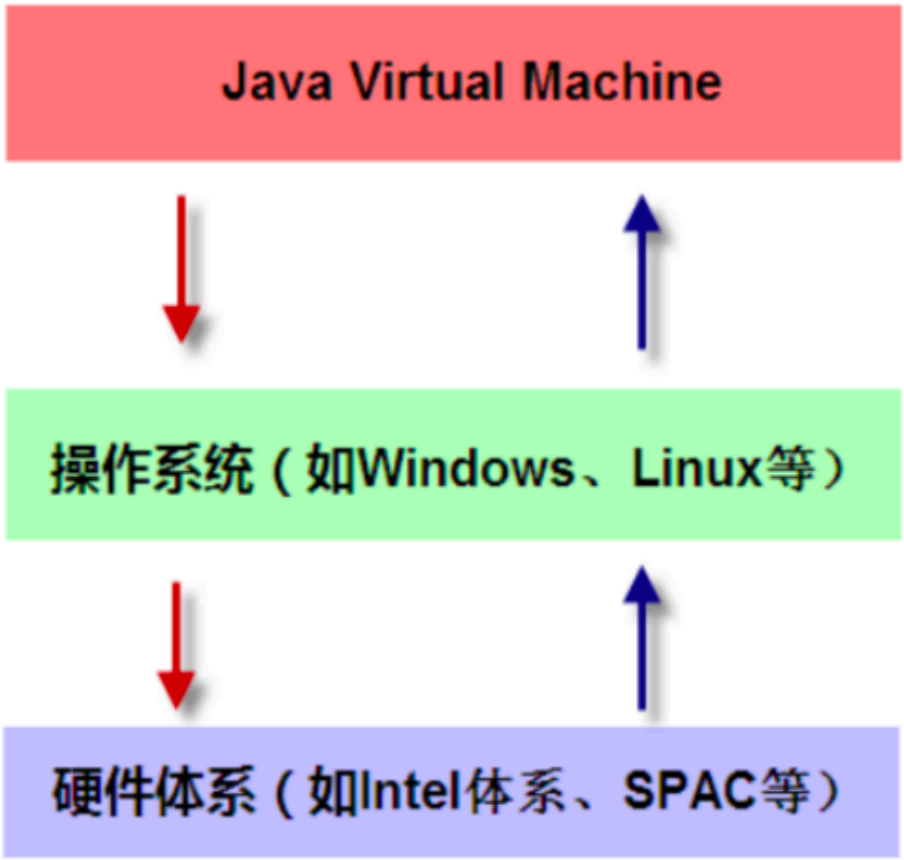
Buffer可以被理解为一个容器(缓冲区,数组),发送到Channel中的所有对象都必须首先放到Buffer中,而从Channel中读取的数据也必须

放到Buffer中,也就是说数据可以从Channel读取到Buffer中,也可以从Buffer写到Channel中

3.1.2. 和传统的IO的区别

```
1.IO面向流,NIO面向缓冲区
Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性.
2.IO是阻塞式的,NIO有非阻塞式的
Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。
3.IO没有选择器,NIO有选择器
Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道.
```

3.2. JVM



JVM是运行在操作系统之上的，它与硬件没有直接的交集。 JVM直接翻译为Java虚拟机但实际应该是Java虚拟机规范。

3.2.1. 三种JVM

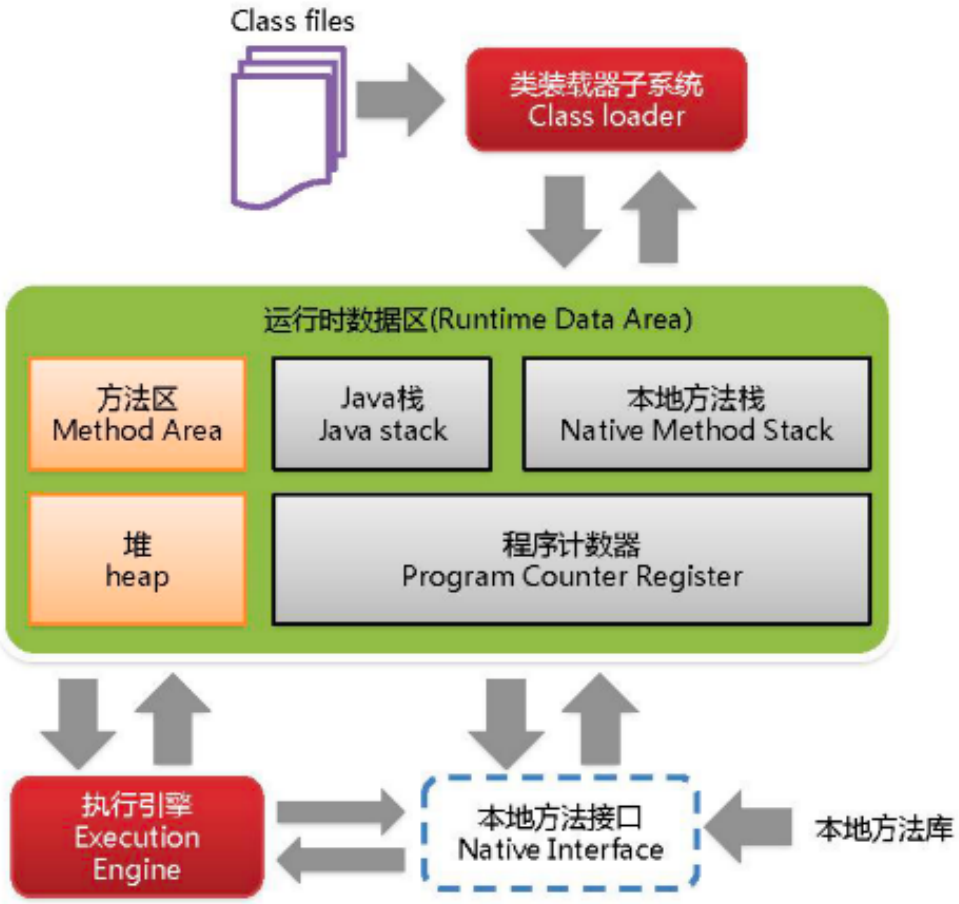
Sun公司的HotSpot、 BEA公司的JRockit、 IBM公司的J9 VM

提起HotSpot VM，相信所有Java程序员都知道，它是Sun JDK和OpenJDK中所带的虚拟机，也是目前使用范围最广的Java虚拟机。但不一定所有人都知道的是，这个目前看起来“血统纯正”的虚拟机在最初并非由Sun公司开发， 而是由一家名为“Longview Technologies”的小公司设计的； 甚至这个虚拟机最初并非是为Java语言而开发的，它来源于Strongtalk VM，而这款虚拟机中相当多的技术又是来源于一款支持Self语言实现“达到C语言50%以上的执行效率”的目标而设计的虚拟机，Sun公司注意到了这款虚拟机在JIT编译上有许多优秀的理念和实际效果，在1997年收购了Longview Technologies公司，从而获得了HotSpot VM。

BEA公司02年从Appeal Virtual Machines收购获得，专注于服务端应用，曾经号称世界上速度最快的虚拟机在2008年和2009年，Oracle公司分别收购了BEA公司和Sun公司，这样Oracle就同时拥有了两款优秀的Java虚拟机： JRockit VM和HotSpot VM。Oracle公司宣布在不久的将来（大约应在发布JDK 8的时候）会完成这两款虚拟机的整合工作，使之优势互补。整合的方式大致上是在HotSpot的基础上，移植JRockit的优秀特性，譬如使用JRockit的垃圾回收器与MissionControl服务，使用HotSpot的JIT编译器与混合的运行时系统。

J9是IBM公司开发的虚拟机，其为IBM公司各种产品的执行平台。

3.2.2. JVM体系结构概览



四 教学内容

4.1. NIO

4.1.1. Buffer缓冲区(会)

4.1.1.1. 缓冲区的概念

其实是一个用来存储基本数据类型的一个容器， 类似于一个数组。

缓冲区， 可以按照存储的数据类型不同， 将缓冲区分为:

ByteBuffer、 ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer、CharBuffer

但是， 要注意， **并没有BooleanBuffer**！

这些缓冲区， 虽然是不同的类， 但是他们拥有的相同的属性和方法。 因为他们都继承自相同的父类 Buffer。

4.1.1.2. Buffer的几个属性

- **capacity:** 容量。 代表一个缓冲区的最大的容量， 缓冲区一旦开辟完成， 将无法修改。
- **limit:** 限制。 表示缓冲区中有多少数据可以操作。
- **position:** 位置。 表示当前要操作缓冲区中的哪一个下标的数据。
- **mark:** 标记。 在缓冲区中设计一个标记， 配合 reset() 方法使用， 修改position的值。

> mark <= position <= limit <= capacity

4.1.1.3. Buffer常用的方法

方法	描述
allocate(int capacity)	开辟一个缓冲区， 并设置缓冲区的容量。
put(byte b)	将一个字节的数据存入缓冲区， position向后挪动一位。
put(byte[] arr)	将一个字节数组的数据存入缓冲区， position向后挪动数组长度位。
flip()	将缓冲区切换成读模式。
get()	获取一个字节的数据。
get(byte[] arr)	将缓冲区中的数据读取到数组中。
mark()	在缓冲区中添加一个标记， 将mark属性的值设置为当前的position的值。
reset()	将属性position的值， 修改为mark记录的值。
rewind()	倒回， 将position的值重置为0， 同时清空mark的标记的值。
clear()	重置所有的属性为缓冲区刚刚被开辟时的状态。

注意事项

- 在向缓冲区中写数据的时候， 要注意： 不要超出缓冲区的范围。 如果超出范围了， 会出现BufferOverflowException异常， 且本次put的所有数据都不会存入到缓冲区中。

> 缓冲区， 其实有两种模式： 分别是读模式和写模式。
>
> **读模式：** 就是从缓冲区中读取数据。
>
> **写模式:** 就是将数据写入到缓冲区。
>
> 一个刚刚被开辟的缓冲区， 默认处于写模式。

4.1.1.4. 缓冲区的详解

其实， 缓冲区中， 并没有所谓的读模式和写模式。 其实所谓“读模式”和“写模式”， 只是逻辑上的区分。

一个处于“读模式”下的缓冲区， 依然可以写数据。 一个处理“写模式”下的缓冲区， 依然可以读取数据。

上述方法中， 基本所有的方法， 都是围绕着缓冲区中的几个属性进行的。

```
1  import java.nio.ByteBuffer;
2
3  /**
4   * @Author 千锋大数据教学团队
5   * @Company 千锋好程序员大数据
6   * @Description “读模式”与“写模式”误区
7   */
8  public class Demo2 {
9    public static void main(String[] args) {
10     /*
11      * 在创建buffer对象的时候传递的参数就是capacity
12      * 容量为1024的缓冲区
13      * 此时buffer的limit和capacity都为1024
14      * 此时的position是0
15      */
16     ByteBuffer buffer = ByteBuffer.allocate(1024);//开辟
   容量1024字节
```

```
17
System.out.println("position:"+buffer.position());//0
18      System.out.println("limit:"+buffer.limit());//1024
19
20      /*
21       * position是5,说明写入了5个字节,position指向的是当前内容的结尾,方便接着往下写
22       */
23      buffer.put("hello".getBytes());
24      System.out.println(buffer.position());//5
25      System.out.println(buffer.limit());//1024
26
27      //可以继续写
28      buffer.put("world".getBytes());
29      System.out.println(buffer.position());//10
30
31      //切换为读模式
32      /*这一步很重要 flip可以理解为模式切换 之前的代码实现的是写入操作
33       *当调用这个方法后就变成读取操作,那么position和limit的值就要发生变换
34       *此时capacity为1024不变
35       *此时limit就移动到原来position所在的位置,相当于把buffer中没有数据的空间
36       *"封印起来"从而避免读取Buffer数据的时候读到null值
37       *相当于 limit = position  limit = 10
38       *此时position的值相当于 position = 0
39       *
40       */
41
42      buffer.flip();
43      System.out.println("3:"+buffer.position());//0
44      System.out.println("3:"+buffer.limit());//10
```

```
45 //    //获取单个字节
46 //    //buffer.get();
47 //    //获取多个字节
48     byte[] data=new byte[buffer.limit()];
49     buffer.get(data);
50     System.out.println("data:"+new String(data));
51     System.out.println("读取data
后:"+buffer.position());//从0变成了10,position相当于读字节
的指针,内容读完了指到了结尾
52     System.out.println("读取data后:"+buffer.limit());
53
54     //将position设回0，所以你可以重读Buffer中的所有数据。
limit保持不变，仍然表示能从Buffer中读取多少个元素
55 //    buffer.rewind();
56 //    byte[] data1=new byte[buffer.limit()];
57 //    buffer.get(data1);
58 //    System.out.println(new String(data1));
59 //    System.out.println(buffer.position());//从0变成了
10,position相当于读字节的指针,内容读完了指到了结尾
60 //    System.out.println(buffer.limit());
61
62     /*
63      * clear():
64      * API中的意思是清空缓冲区,但是实际是没有删除缓冲区数据的
65      * 而是将缓冲区中limit和position恢复到初始状态
66      * 即limit和capacity都为1024 position是0
67      * 此时可以完成写入模式
68      */
69 //    buffer.clear();
70 //    System.out.println("clear后:"+buffer.position());
71 //    System.out.println("clear后:"+buffer.limit());
72
73     //可以继续写
```

```
74 //    buffer.put("temp".getBytes());
75     //继续读
76 //    buffer.flip();
77 //    byte[] arr = new byte[buffer.limit()];
78 //    buffer.get(arr);
79 //    System.out.println("temp:"+new String(arr));
80   }
81 }
```

4.1.1.5. 直接缓冲区(了解)

非直接缓冲区， 是在JVM中开辟的空间。 allocate(int capacity)

直接缓冲区， 是直接在物理内存上开辟的空间。 allocateDirect(int capacity)

> 对于一个已经存在的缓冲区， 可以使用 isDirect() 方法， 判断是否是直接缓冲区。 这个方法的返回值类型是boolean类型的， 如果是true, 就表示是一个直接缓冲区。 如果是false， 就表示不是一个直接缓冲区。

4.1.2. Channel通道(会)

4.1.2.1. 通道的简介

通道Channel， 就是文件和程序之间的连接。 在NIO中， 数据的传递， 是由缓冲区实现的， 通道本身并不负责数据的传递。

通道在 java.nio.channels 包中， 常见的Channel子类：

- 本地文件: FileChannel
- 网络文件: SocketChannel, ServerSocketChannel, DatagramChannel

Channel类似于传统的流对象,但与传统的流对象有两个主要的区别:

- Channel可以直接将我指定文件的部分或全部直接映射成Buffer
- 程序不能直接访问Channel中的数据,包括读取,写入都不行.Channel只能与Buffer进行交互

4.1.2.2. 通道的打开方式

1. 可以使用支持通道的类， getChannel()方法获取通道。
    1. 本地文件: FileInputStream、FileOutputStream
    2. 网络文件: Socket、ServerSocket、DatagramSocket
2. 在NIO.2中， 使用FileChannel.open()方法打开通道。
3. 在NIO.2中， 使用Files工具类newXXX()方法打开通道。

4.1.2.3. 使用FileChannel读写数据

在使用FileChannel之前，必须先打开它。但是，我们无法直接打开一个FileChannel，需要通过使用一个InputStream、OutputStream的getChannel()方法来返回对应的Channel

- 写数据

```
1  public class Demo3 {
2    public static void main(String[] args) throws
   IOException {
3      /*
4       * 写入文本文件
5       */
6      //创建文件写出流
7      FileOutputStream fileOutputStream = new
   FileOutputStream("temp1.txt");
```

```
8       //获取通道
9       FileChannel fileChannel =
    fileOutputStream.getChannel();
10      //创建缓冲流
11      ByteBuffer buffer = ByteBuffer.allocate(1024);
12      //向缓冲流中放入数据
13      buffer.put("helloworld".getBytes());
14      //转换模式为读取内容
15      buffer.flip();
16      //利用通道写入
17      fileChannel.write(buffer);
18      //关闭资源
19      fileChannel.close();
20      fileOutputStream.close();
21      System.out.println("写入完毕");
22    }
23  }
24
```

- 读数据

```
1   public class Demo4 {
2     public static void main(String[] args) throws
    IOException {
3       /*
4        * 文件读取
5        */
6       //创建文件输入流
7       FileInputStream fileInputStream = new
    FileInputStream("temp1.txt");
8       //获取通道
```

```
        FileChannel fileChannel =
fileInputStream.getChannel();
        //创建缓冲流对象
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        //利用通道读取内容
        int num = fileChannel.read(buffer);
        System.out.println(num);
        //转换模式为读取模式
        buffer.flip();
        //利用缓冲流读取数据到控制台
        System.out.println(new
String(buffer.array(),0,num));
        //关闭资源
        fileChannel.close();
        fileInputStream.close();
        System.out.println("读完了");
    }
}

```

课上练习一:

单文件的复制

```
package com.qf.NIO;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.OpenOption;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

```

```java

public class Demo5 {
  public static void main(String[] args) throws IOException {
    /*
     * 文件的复制
     * //打开或创建一个文件，返回一个文件通道来访问该文件
       //StandardOpenOption是枚举里面显示的是文件打开方式
     */
    //创建通道
    FileChannel inChannel = FileChannel.open(Paths.get("temp1.txt"), StandardOpenOption.READ);
    FileChannel outChannel = FileChannel.open(Paths.get("temp2.txt"), StandardOpenOption.WRITE,StandardOpenOption.CREATE);
    //创建缓冲流
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    int num = 0;
    //复制
    while ((num = inChannel.read(buffer)) != -1) {
      buffer.flip();
      outChannel.write(buffer);
      buffer.clear();//让指针重新回到0
    }

    inChannel.close();
    outChannel.close();
    System.out.println("复制完毕");
  }
}
```

4.1.2.4. 分散聚合(了解)

分散写， 聚合读。

将文件中的数据分散的写入到多个缓冲区， 这些缓冲区按照一定的顺序进行数据的写入。 在读取的时候， 也按照相同的顺序进行数据的读操作。

示例代码

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Program2 {
    public static void main(String[] args) {
        try (FileChannel srcChannel = FileChannel.open(Paths.get("file\\day28\\src\\src"), StandardOpenOption.READ);
             FileChannel dstChannel = FileChannel.open(Paths.get("file\\day28\\src\\target"), StandardOpenOption.WRITE, StandardOpenOption.APPEND, StandardOpenOption.CREATE)) {
            // 做多个缓冲区，存入一个数组中
            ByteBuffer[] buffers = new ByteBuffer[4];
            for (int i = 0; i < buffers.length; i++) {
                buffers[i] = ByteBuffer.allocate(128);
            }
```

```
            while (srcChannel.read(buffers) != -1) {
                // 从源文件读取数据
                for (ByteBuffer buffer : buffers) {
                    buffer.flip();
                }
                dstChannel.write(buffers);
                // 重置状态
                for (ByteBuffer buffer : buffers) {
                    buffer.clear();
                }
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4.1.4. NIO2(会)

4.1.4.1. 简介

Java7对原有NIO进行了重大改进,改进主要包括如下两个方面的内容

1.提供了全面的文件IO和文件系统访问支持

2.基于异步Channel的IO

Path、Paths和Files

之前学习中学习过File类来访问文件系统,但是File类的功能比较有限,NIO.2为了弥补这种不足,引入了一个Path接口还提供了Files和Paths两个工具类,其中Files封装了包含大量静态方法来操作文件,Paths则包含了返回Path的静态工厂方法

4.1.4.2. Files使用示例代码

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class FilesUsage {
    public static void main(String[] args) throws IOException {
        // 1. 创建一级文件夹(如果这个文件夹存在，会抛异常)
        // Files.createDirectory(Paths.get("C:\\Users\\luds\\Desktop\\abc"));
        // 2. 创建多级文件夹(如果需要创建的文件夹存在，不会抛出异常)
        // Files.createDirectories(Paths.get("C:\\Users\\luds\\Desktop\\abc\\a\\b\\c"));
        // 3. 创建一个文件(如果这个文件已存在，重复创建会异常)
```

```
19          //
Files.createFile(Paths.get("C:\\Users\\luds\\Desktop\\a
bc\\file"));
20          // 4. 删除一个文件或者空文件夹(如果要删除的文件或空文件
夹不存在，会抛异常；如果删除的不是一个空文件夹也会抛异常)
21          //
Files.delete(Paths.get("C:\\Users\\luds\\Desktop\\abc\\
file"));
22          // 5. 删除一个文件或者空文件夹(如果删除失败，不会抛异
常，返回false)
23          //
Files.deleteIfExists(Paths.get("C:\\Users\\luds\\Deskto
p\\abc\\file"));
24          // 6. 移动文件(重命名)
25          // Files.move(Paths.get("file\\day28\\source"),
Paths.get("file\\day28\\dst\\source"));
26
27          // 7. 拷贝文件
28
Files.copy(Paths.get("file\\day28\\dst\\source"),
Paths.get("file\\day28\\source"));
29      }
30  }
```

4.1.4.3. NIO2使用示例代码

```
1  public class Demo7 {
2      public static void main(String[] args) throws
IOException {
3          //Path就是用来替代File
4          //构建Path对象的两种方式
5          //传一个路径
6          Path path1 = Paths.get("D:\\123");
```

```
7         //第一个参数是盘符 ,第二个参数是可变参数 下面有多少文件路径就写多少
8         Path path2 = Paths.get("D:\\", "123","456.txt");
9         //Path是结合Files工具类使用的
10        //创建目录
11        Files.createDirectories(path1);
12        //判断文件是否存在
13        if(!Files.exists(path2)) {
14          //创建文件
15          Files.createFile(path2);
16        }
17
18        //复制文件
19        //第一个参数原文件路径, 第二个参数目标文件路径
20  //    Files.copy(new FileInputStream(new File("D:\\123\\456.txt")), Paths.get("D:\\", "123","222.txt"));
21  //     Files.copy(path2, new FileOutputStream(new File("D:\\123\\789.txt")));
22  //     Files.copy(path2, Paths.get("D:\\", "123","111.txt"));
23       //一次读取文件中所有的行
24        List<String> readAllLines = Files.readAllLines(Paths.get("src/com/qiangfeng/test/Demo1.java"));
25          for (String str : readAllLines) {
26        System.out.println("haha:"+str);
27      }
28          //将集合中的内容写入到文件中
29          Files.write(Paths.get("D:\\", "123","Demo.java"), readAllLines);
30          /*
```

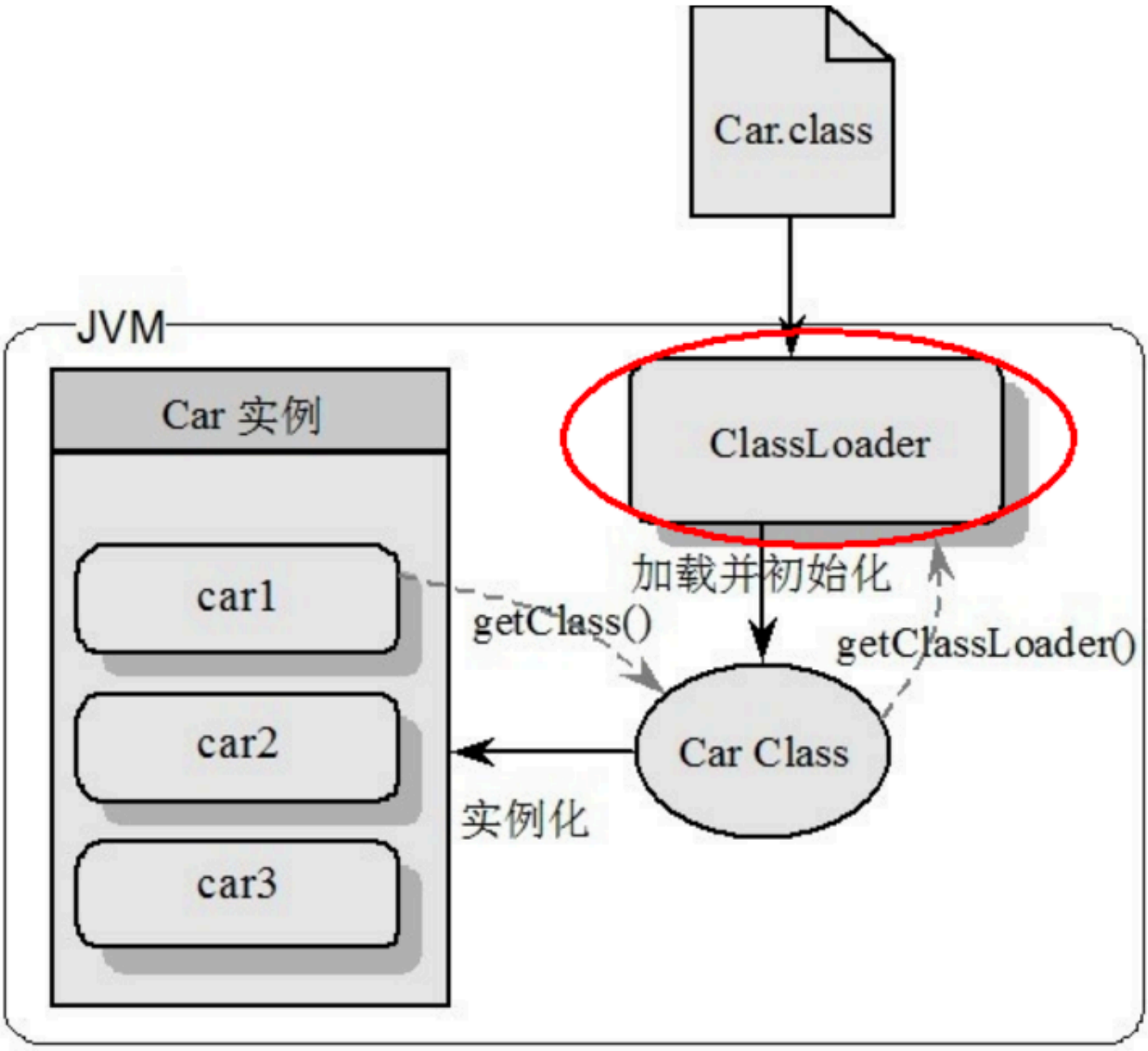
```
31          static BufferedReader newBufferedReader(Path
    path)
32      打开一个文件进行读取，返回一个 BufferedReader以有效方式从
    文件中读取文本。
33      static BufferedReader newBufferedReader(Path path,
    Charset cs)
34      打开一个文件进行读取，返回一个 BufferedReader可以用来有效
    方式从文件中读取文本。
35      static BufferedWriter newBufferedWriter(Path path,
    Charset cs, OpenOption... options)
36      打开或创建一个文件写入，返回一个 BufferedWriter，可以有效
    的方式将文件写入文本。
37      static BufferedWriter newBufferedWriter(Path path,
    OpenOption... options)
38      打开或创建一个文件写入，返回一个 BufferedWriter能够以有效
    的方式的文件写入文本。
39
40      直接创建缓冲流对象 可以执行 字符集 和访问权限
41           */
42    }
43  }
44
```

4.2. JVM(了解)

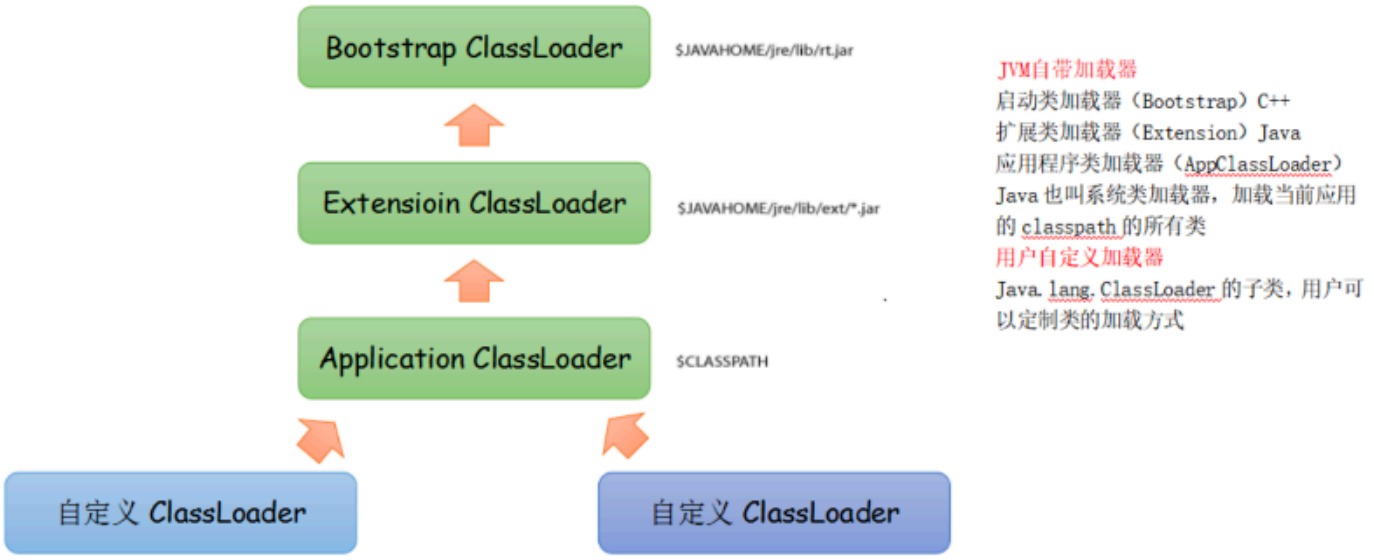
4.2.1. 类加载器ClassLoader

类加载器， 负责加载class文件。 class文件在文件开头有特定的文件标示。

ClassLoader只负责class文件的加载， 至于它是否可以运行， 则由Execution Engine决定。




```
Car.class 相当于是我们编写的类模板， 封装着属性和行为， 但是.class文件是存储在物理内存中的。 我们通过ClassLoader类的加载，加载到JVM中， 此时可以得到Car Class这里的Car。 Class就相当于是JVM中的模板， 那么创建Car实例都是一样的。 我们之前学过反射， 那么我们知道通过反射可以获取属性和行为， 就是因为我们获取到了JVM中的Car Class。
```



1. 启动类加载器:

   这个类加载器负责放在 `<JAVA_HOME>\lib` 目录中的， 或者被 `-Xbootclasspath` 参数所指定的路径中的， 并且是虚拟机识别的类库。用户无法直接使用。 就相当于是为什么我们可以通过new创建出来对象。

2. 扩展类加载器:

   这个类加载器由 `sun.misc.Launcher$AppClassLoader` 实现。它负责 `<JAVA_HOME>\lib\ext` 目录中的，或者被 `java.ext.dirs` 系统变量所指定的路径中的所有类库。用户可以直接使用。

3. 应用程序类加载器:

   这个类由 `sun.misc.Launcher$AppClassLoader` 实现。是ClassLoader中 `getSystemClassLoader()` 方法的返回值。它负责用户路径 `ClassPath` 所指定的类库。用户可以直接使用。如果用户没有自己定义类加载器，默认使用这个。

4. 自定义加载器:

   用户自己定义的类加载器（只有做框架才会用到）

```
1 public class Demo {
```

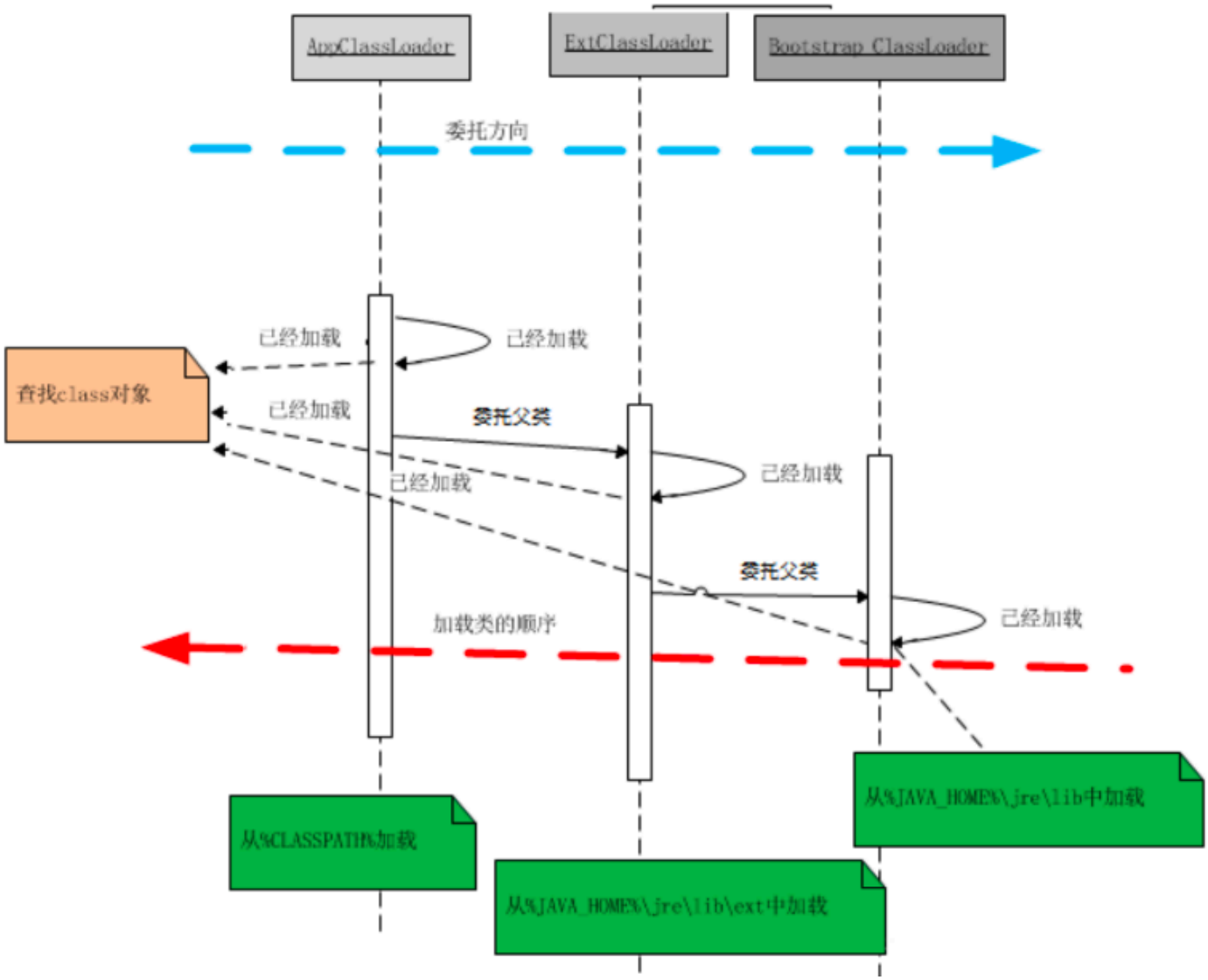
```
    public static void main(String[] args) {
        //Object类是系统提供的
        Object obj = new Object();
        //Demo是自己创建的
        Demo demo = new Demo();
        //分别获取Object和Demo类中给的ClassLoader
        //获取反射对象的三种方式
        //1.通过对象.getClass
        //2.通过类名.class
        //3.Class.forName(类的全限定名)
        System.out.println("ClassLoader is:"+obj.getClass().getClassLoader());
        System.out.println("-------------------------------分割线-------------------------------------");
        System.out.println("ClassLoader is:"+demo.getClass().getClassLoader());
        System.out.println("ClassLoader father is:"+demo.getClass().getClassLoader().getParent());
        System.out.println("ClassLoader grandfather is:"+demo.getClass().getClassLoader().getParent().getParent());
        /**
         * getClassLoader打印是null的原因
         * 提到这里不得不提一下jvm的类加载机制。自上而下加载，自下而上检查。
         * Bootstrap ClassLoader是由C++编写的，它本身是虚拟机的一部分，所以它并不是一个JAVA类，也就是无法在java代码中获取它的引用，
         * JVM启动时通过Bootstrap类加载器加载rt.jar等核心jar包中的class文件，所以当系统类通过.getClass或是.class都是由它加载。
```

```
22              * 然后呢，我们通过执行自定义类的打印发现，JVM初始化sun.misc.Launcher并创建AppClassLoader和 Extension ClassLoader实例。
23              * ExtClassLoader设置为AppClassLoader的父加载器。Bootstrap没有父加载器，但是它却可以作用一 个ClassLoader的父加载器。
24              * 比如ExtClassLoader。这也可以解释之前通过ExtClassLoader的getParent方法获取为Null的现象。
25              * 具体为什么是这个原因,下面就给大家解释一下
26              *
27              * ps:这里需要注意的是父加载器不是父类
28              * 这些类可以在jdk安装目录中的jre文件夹中lib文件夹下有一个rt.jar包可以通过压缩的方式打开然后通过包名的形式找到对应和也会发现Object.class也是存在的,
29              * 这就是为什么说我们创建的类默认继承于Object
30              * Object类在于java.lang.Object下边
31              */
32      }
33  }
```

4.2.2. JVM双亲委派机制和沙箱机制

4.2.2.1. 双亲委派

一个类加载器查找class和resource时，是通过“委托模式”进行的，它首先判断这个class是不是已经加载成功，如果没有的话它并不是自己进行查找，而是先通过父加载器，然后递归下去，直到Bootstrap ClassLoader，如果Bootstrap classloader找到了，直接返回，如果没有找到，则一级一级返回，最后到达自身去查找这些对象。这种机制就叫做双亲委托。



可以看到2根箭头，蓝色的代表类加载器向上委托的方向，如果当前的类加载器没有查询到这个class对象已经加载就请求父加载器（不一定是父类）进行操作，然后以此类推。直到Bootstrap ClassLoader。如果Bootstrap ClassLoader也没有加载过此class实例，那么它就会从它指定的路径中去查找，如果查找成功则返回，如果没有查找成功则交给子类加载器，也就是ExtClassLoader,这样类似操作直到终点，也就是我上图中的红色箭头示例。 用序列描述一下：

1. 一个AppClassLoader查找资源时，先看看缓存是否有，缓存有从缓存中获取，否则委托给父加载器。
2. 递归，重复第1部的操作。
3. 如果ExtClassLoader也没有加载过，则由Bootstrap ClassLoader出面，它首先查找缓存，如果没有找到的话，就去找自己的规定的路径

下，也就是sun.mic.boot.class下面的路径。找到就返回，没有找到，让子加载器自己去找。

4. Bootstrap ClassLoader如果没有查找成功，则ExtClassLoader自己在java.ext.dirs路径中去查找，查找成功就返回，查找不成功，再向下让子加载器找。
5. ExtClassLoader查找不成功，AppClassLoader就自己查找，在java.class.path路径下查找。找到就返回。如果没有找到就让子类找，如果没有子类会怎么样？抛出各种异常。

4.2.2.2. 沙箱机制

沙箱机制也就是双亲委派模型的安全机制。

```
// 在写代码中,系统会提供对应的系统类给我们使用,比如String字符串,那么我们来进行一个模拟操作, 自定义一个java.lang.String类
package java.lang;
/**
 * 创建一个String类包名和类名完全和系统中的String一致
 */
public final class String {
    public static void main(String[] args) {
        System.out.println("谈恋爱吗?做监狱的那种!");
    }
}
// 当我们执行的时候回发现:
// 错误: 在类 java.lang.String 中找不到 main 方法, 请将main 方法定义为: public static void main(String[] args)否则JavaFX应用程序类必须扩展javafx.application.Application
// 代码中是明明有main的为什么会出错就是时双亲委托中的沙箱机制了
// 自定义的java.lang.String类永远都不会被加载进内存。
```

```
15 // 因为首先是最顶端的类加载器加载系统的java.lang.String 类，最终自定义的类加载器无法加载java.lang.String类这样一来的好处就是可以保证不被恶意的修改系统中原有的类
```

> 了解:
>
> 双亲委托和沙箱机制不是绝对安全的因为可以写自定义ClassLoader,自定义的类加载器里面强制加载自定义的 java.lang.String 类，不去通过调用父加载器 ,完成类的加载. 当ClassLoader加载成功后,Execution Engine执行引擎负责解释命令，提交操作系统执行。

4.2.3. 本地方法栈和本地方法接口

我们可以发现Object类中有很多方法是使用native修饰,并且没有方法体,那说明这些方法超出了Java的范围。 所以底层实现需要使用JNI--->Java Native Interface。

native修饰的方法就是告诉java本身,此时需要调用外部的本地类库即C/C++类库来执行这个方法

4.2.3.1. Native Interface本地接口

本地接口的作用是融合不同的编程语言为 Java 所用，它的初衷是融合C/C++程序，Java 诞生的时候是 C/C++横行的时候，要想立足，必须有调用 C/C++程序，于是就在内存中专门开辟了一块区域处理标记为native的代码，它的具体做法是 Native Method Stack中登记 native方法，在Execution Engine 执行时加载native libraies。目前该方法使用的越来越少了，除非是与硬件有关的应用，比如通过Java程序驱动打印机或者Java系统管理生产设备，在企业级应用中已经比较少见。因为现在的异构领域间的通信很发达，比如可以使用 Socket通信，也可以使用WebService等等，不多做介绍。

4.2.3.2. Native Method Stack

它的具体做法是Native Method Stack中登记native方法，在Execution Engine 执行时加载本地方法库。

问题:现在我们发现虚拟机中有Java栈和本地方法栈两个栈区,那么我们在创建对象或是声明变量的时候对应的变量或对象是在Java栈还是本地方法栈中呢?

Person p = new Person();

那么对象p是在Java栈还是本地方法栈?

不带native的进入到Java栈中,只有带native的进入到本地方法栈,所以p对象是在Java栈中,而我们平时所说的栈就是Java栈。

4.2.4. PC寄存器

每个线程都有一个程序计数器，是线程私有的,就是一个指针，指向方法区中的方法字节码（用来存储指向下一条指令的地址,也即将要执行的指令代码），由执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不记。

pc寄存器的作用:

```
public class Demo {
    public static void show1() {}
    public static void show2() {}
    public static void show3() {}
    public static void main(String[] args) {
        /*
         *当执行代码的时候都是到是从上至下顺序执行
```

```
8               *可以发现代码中是调用了三个方法,那么这个三个方法也是顺序执行
9               *那么这个三个方法是如何计入顺序执行的呢?
10              *就是PC寄存器
11              */
12              show1();
13              show2();
14              show3();
15          }
16      }
```

4.2.5. Method Area方法区

方法区是被所有线程共享，所有字段和方法字节码，以及一些特殊方法如构造函数，接口代码也在此定义。简单说，**所有定义的方法的信息都保存在该区域，此区属于共享区间**。

存放在方法区的:静态变量+常量+类信息(构造方法/接口定义)+运行时常量池存在方法区中

ps:只要是被线程私有独享的一律没有会后,只有是线程共享才能有回收

4.2.6. Stack栈是什么

> ps:有一句对但是是废话的一句话, 程序=算法+数据结构 实际应该是 程序= 框架+业务逻辑

介绍两个数据结构:

- 栈:先进后出(FILO)
- 队列:先进先出(FIFO)

> ps:栈中main方法一定是在栈底

栈也叫栈内存，主管Java程序的运行，是在线程创建时创建，它的生命期是跟随线程的生命期，线程结束栈内存也就释放，对于栈来说不存在垃圾回收问题，只要线程一结束该栈就Over，生命周期和线程一致，是线程私有的。

栈中存储什么: 8种基本类型的变量+对象的引用变量+实例方法都是在函数的栈内存中分配

4.2.6.1. 栈存储什么

栈帧中主要保存3 类数据:

- 本地变量（Local Variables）:输入参数和输出参数以及方法内的变量;
- 栈操作（Operand Stack）:记录出栈、入栈的操作;
- 栈帧数据（Frame Data）:包括类文件、方法等等

4.2.6.2. 栈运行原理

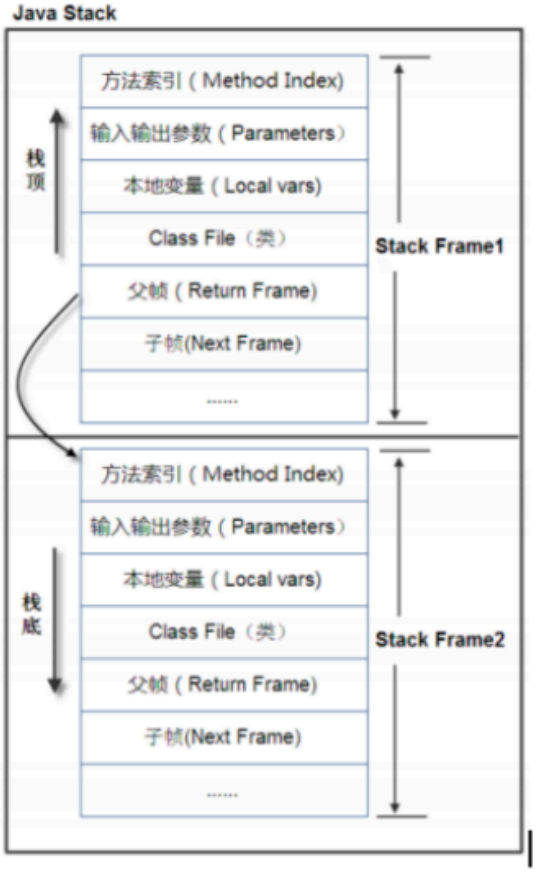
栈中的数据都是以栈帧（Stack Frame）的格式存在，栈帧是一个内存区块，是一个数据集，是一个有关方法(Method)和运行期数据的数据集，当一个方法A被调用时就产生了一个栈帧 F1，并被压入到栈中。

A方法又调用了 B方法，于是产生栈帧 F2 也被压入栈，

B方法又调用了 C方法，于是产生栈帧 F3 也被压入栈，

……

执行完毕后，先弹出F3栈帧，再弹出F2栈帧，再弹出F1栈帧……



图示在一个栈中有两个栈帧：

栈帧 2 是最先被调用的方法，先入栈，

然后方法 2 又调用了方法 1，栈帧 1 处于栈顶的位置，

栈帧 2 处于栈底，执行完毕后，依次弹出栈帧 1 和栈帧 2，

线程结束，栈释放。

每执行一个方法都会产生一个栈帧，保存到栈(后进先出)的顶部，顶部栈就是当前的方法，该方法执行完毕 后会自动将此栈帧出栈

4.2.6.3. StackOverflowError

StackOverflflowError:抛出这个错误是因为递归太深。 其实真正的原因是因为Java线程操作是基于栈的，当调用方法内部方法也就是进行一次递归的时候就会把当前方法压入栈直到方法内部的方法执行完全之后，就会返回上一个方法，也就是出栈操作执行上一个方法。

4.2.6.4. 总结

只要我们知道错误发生的原因了。当出现相就的问题就可以快速定位问题，迅速解决问题。

- StackOverflflowError：递归过深，递归没有出口。
- OutOfMemoryError：JVM空间溢出，创建对象速度高于GC回收速度。

4.2.7. Heap堆

一个JVM实例只存在一个堆内存，堆内存的大小是可以调节的。类加载器读取了类文件后，需要把类、方法、常变量放到堆内存中，保存所有引用类型的真实信息，以方便执行器执行，堆内存分为三部分：

- Young Generation Space 新生区 Young/New
- Tenure generation space 养老区 Old/ Tenure
- Permanent Space 永久区 Perm

4.2.7.1. 新生区

新生区是类的诞生、成长、消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分： 伊甸区（Eden space）和幸存者区（Survivor pace） ，所有的类都是在伊甸区被new出来的。幸存区有两个： 0区（Survivor 0 space）和1区（Survivor 1 space） 。当伊甸园的空间用完时，程序又需要创建对象，JVM的垃圾回收器将对伊甸园区进行垃圾回收(Minor GC)即轻量垃圾回收，将伊甸园区中的不再被其他对象所引用的对象进行销毁。然后将伊甸园中的剩余对象移动到幸存 0区。若幸存 0区也满了，再对该区进行垃圾回收，然后移动到 1区。那如果1 区也满了呢？再移动到养老区。若养老区也满了，那么这个时候将产生MajorGC（FullGC)即重量垃圾回收，进行养老区的内存清理。若养老区执行了Full GC之后发现依然无法进行对象的保存，就会产生OOM异常“OutOfMemoryError”。

4.2.7.2. OutOfMemoryError

如果出现java.lang.OutOfMemoryError: Java heap space异常，说明Java虚拟机的堆内存不够。原因有二：

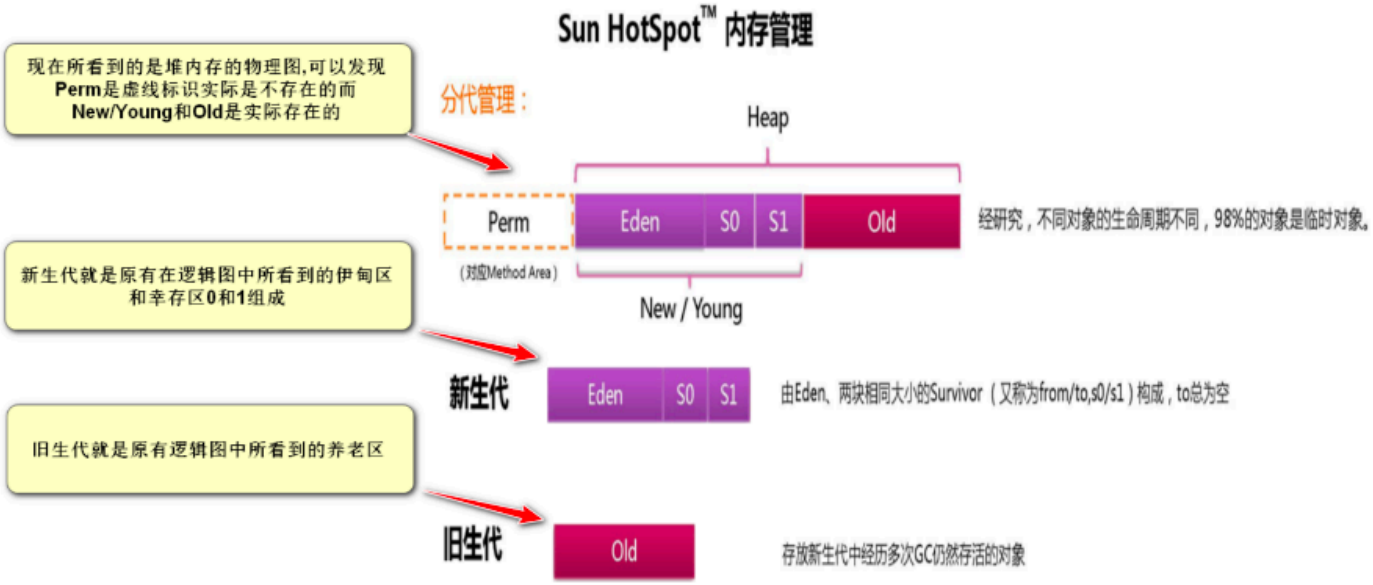
- Java虚拟机的堆内存设置不够，可以通过参数-Xms、-Xmx来调整。
- 代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）。

4.2.7.3. 永久区

永久存储区是一个常驻内存区域，用于存放JDK自身所携带的Class,Interface 的元数据，也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭 JVM 才会释放此区域所占用的内存。

如果出现java.lang.OutOfMemoryError: PermGen space，说明是Java虚拟机对永久代Perm内存设置不够。一般出现这种情况，都是程序启动需要加载大量的第三方jar包。例如：在一个Tomcat下部署了太多的应用。或者大量动态反射生成的类不断被加载，最终导致Perm区被占满。

- Jdk1.6及之前： 有永久代, 常量池1.6在方法区
- Jdk1.7： 有永久代，但已经逐步“去永久代”，常量池1.7在堆
- Jdk1.8及之后： 无永久代，常量池1.8在元空间

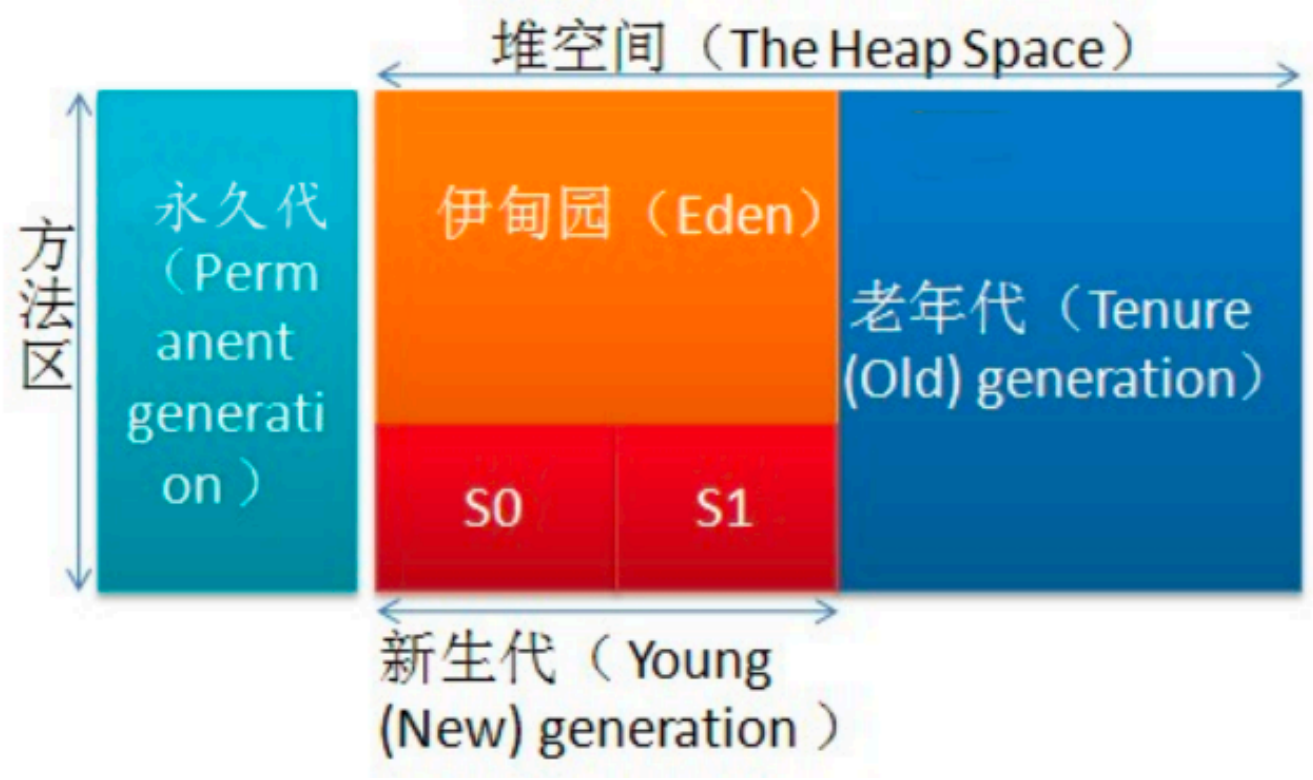


实际而言，方法区（Method Area）和堆一样，是各个线程共享的内存区域，它用于存储虚拟机加载的：类信息+普通常量+静态常量+编译器编译后的代码等等，虽然JVM规范将方法区描述为堆的一个逻辑部分，但它却还有一个别名叫做Non-Heap(非堆)，目的就是要和堆分开。

对于HotSpot虚拟机，很多开发者习惯将方法区称之为“永久代(Parmanent Gen)”，但严格本质上说两者不同，或者说使用永久代来实现方法区而已，永久代是方法区(相当于是一个接口interface)的一个实现，jdk1.7的版本中，已经将原本放在永久代的字符串常量池移走。

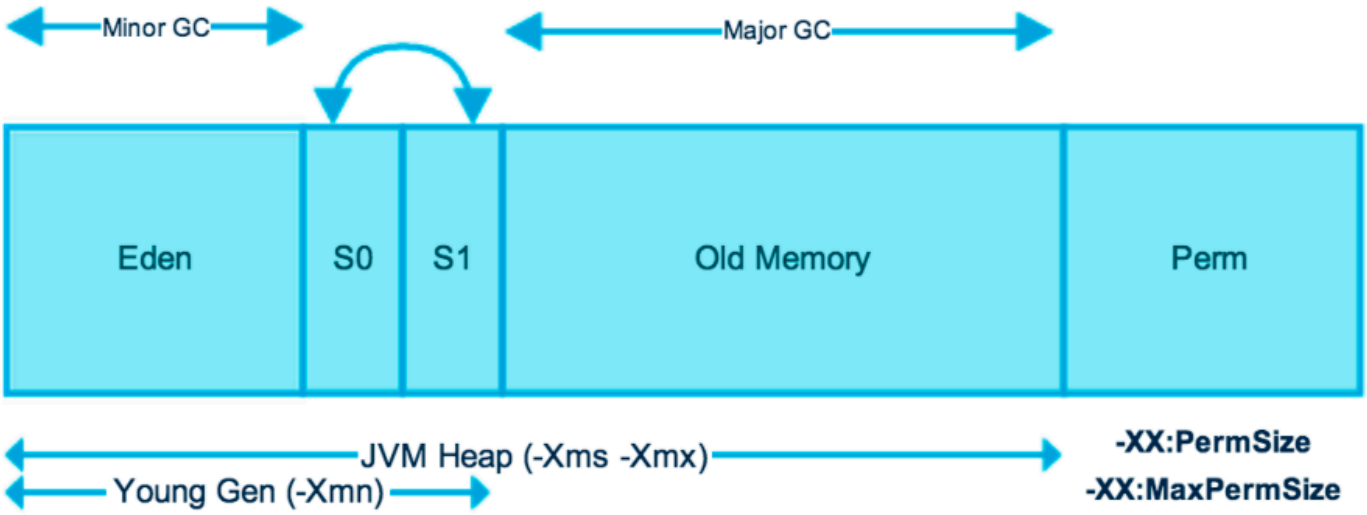
> ps:直白一点其实就是方法区就是永久带的一种落地实现

常量池（Constant Pool）是方法区的一部分，Class文件除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，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。



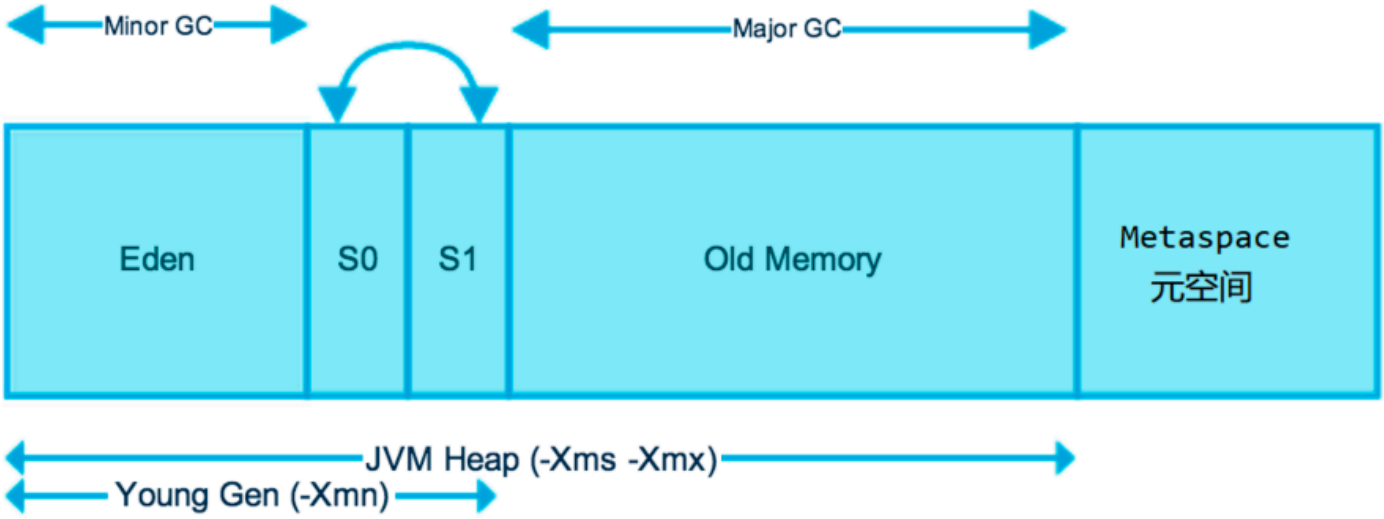
4.2.8. 堆内存调优

4.2.8.1. Java7



4.2.8.2. Java8

Java8之后将最初的永久代取消了，由元空间取代。



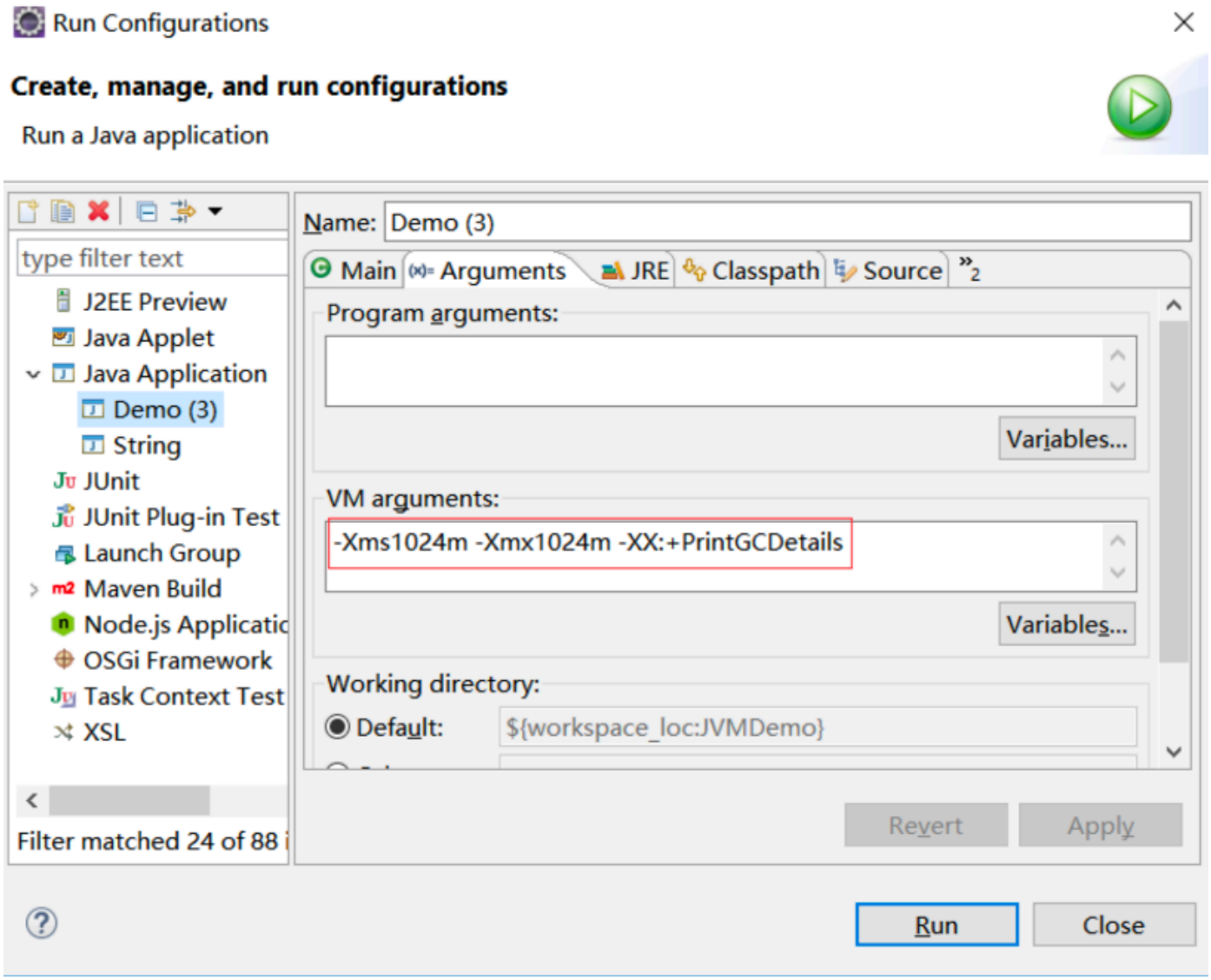
4.2.8.3. Java调优内存量计算

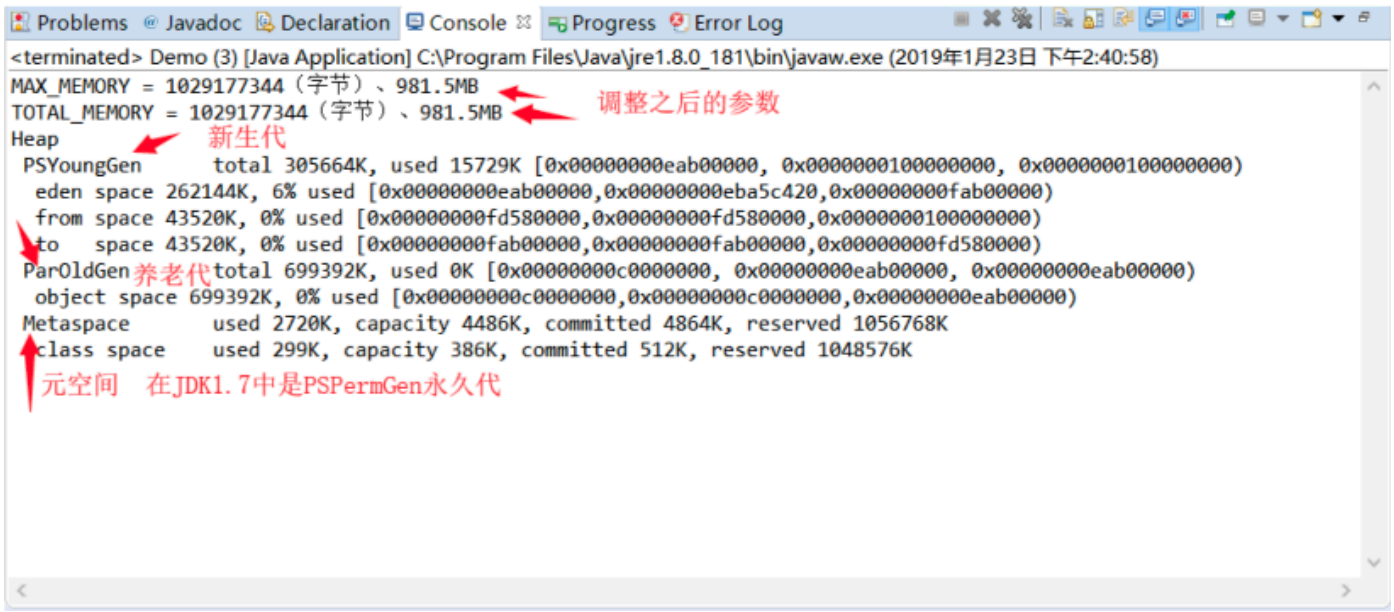
-Xms	设置初始分配大小，默认为物理内存的“1 / 64”
-Xmx	最大分配内存，默认为物理内存的 “1 / 4”
-XX:+PrintGCDetails	输出详细的GC处理日志

```java
public class Demo {
    public static void main(String[] args) {
        long maxMemory = Runtime.getRuntime().maxMemory() ;
        //返回 Java 虚拟机试图使用的最大内存量。
        long totalMemory = Runtime.getRuntime().totalMemory() ;
        //返回 Java 虚拟机中的内存总量。
        System.out.println("MAX_MEMORY = " + maxMemory + "（字节）、" + (maxMemory / (double)1024 / 1024) + "MB");
        System.out.println("TOTAL_MEMORY = " + totalMemory + "（字节）、" + (totalMemory / (double)1024 / 1024) + "MB");
    }
}
// 发现默认的情况下分配的内存是总内存的“1 / 4”、而初始化的内存为“1 / 64”
// MAX_MEMORY = 3799515136（字节）、3623.5MB
// TOTAL_MEMORY = 257425408（字节）、245.5MB
```

VM参数

- -Xms1024m : 最大内存
- -Xmx1024m : 初始化内存
- -XX：+printGCDetails : 打印内存日志信息





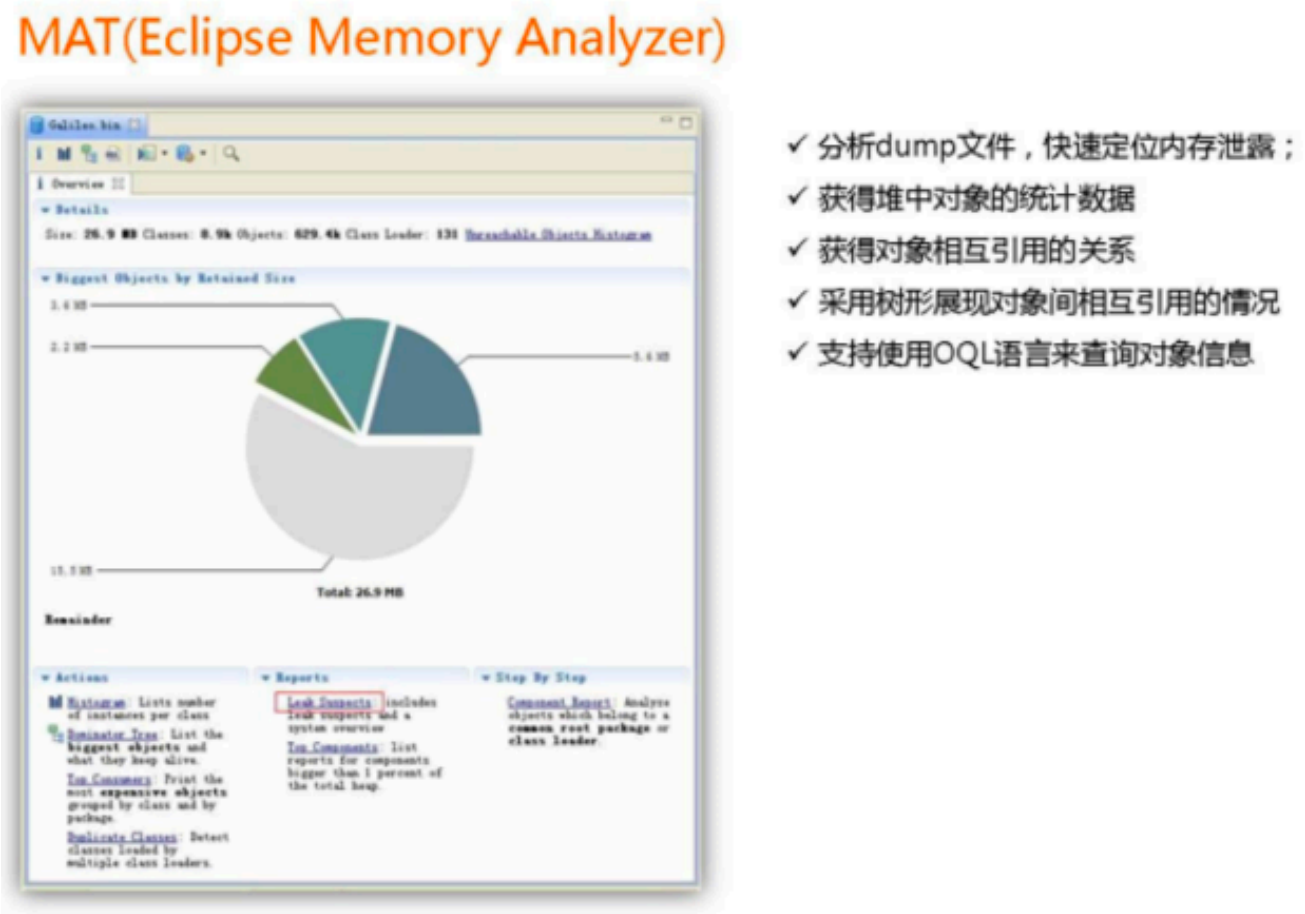
用新生代大小/1024+养老代大小/1024 = 初始化的大小

来看一下OOM

先修改VM参数：-Xms8m -Xmx8m -XX:+PrintGCDetails

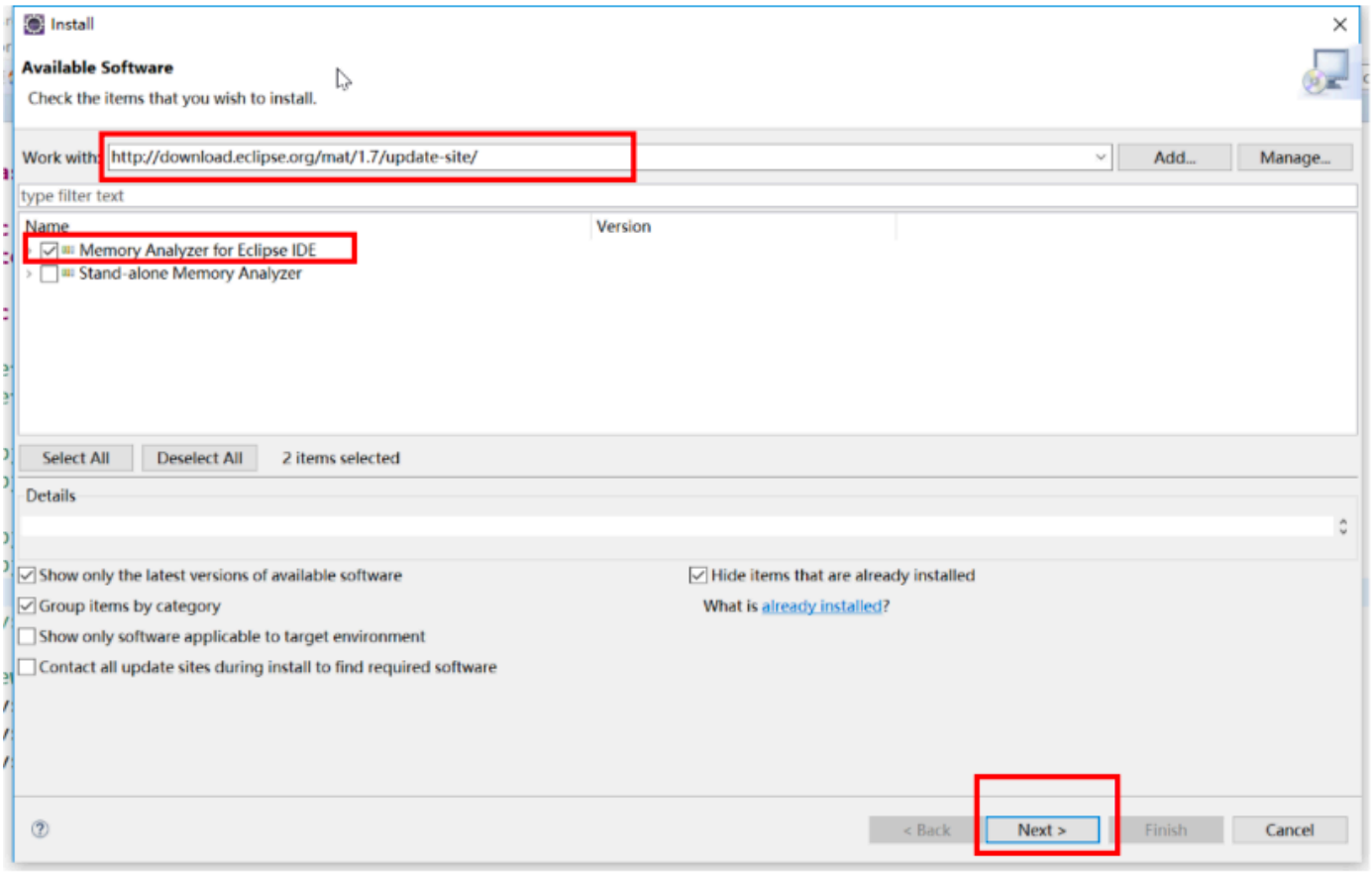
```
import java.util.Random;
public class Demo {
    public static void main(String[] args) {
        String str = "1234567" ;
        while(true) {
            str += str + new Random().nextInt(88888888)
+ new Random().nextInt(999999999) ;
        }
    }
}
```

4.2.9. MAT



在Eclipes中Help--->Install New Software中

下载地址: http://download.eclipse.org/mat/1.7/update-site/




```
import java.util.ArrayList;
public class Demo {
    byte[] byteArray = new byte[1024*1024];
    public static void main(String[] args) {
        int count = 1;
        ArrayList<Demo> list = new ArrayList<>();
        try {
            while(true) {
                list.add(new Demo());
                count++;
            }
        } catch(Exception e) {

System.out.println("*****************count"+count);
```

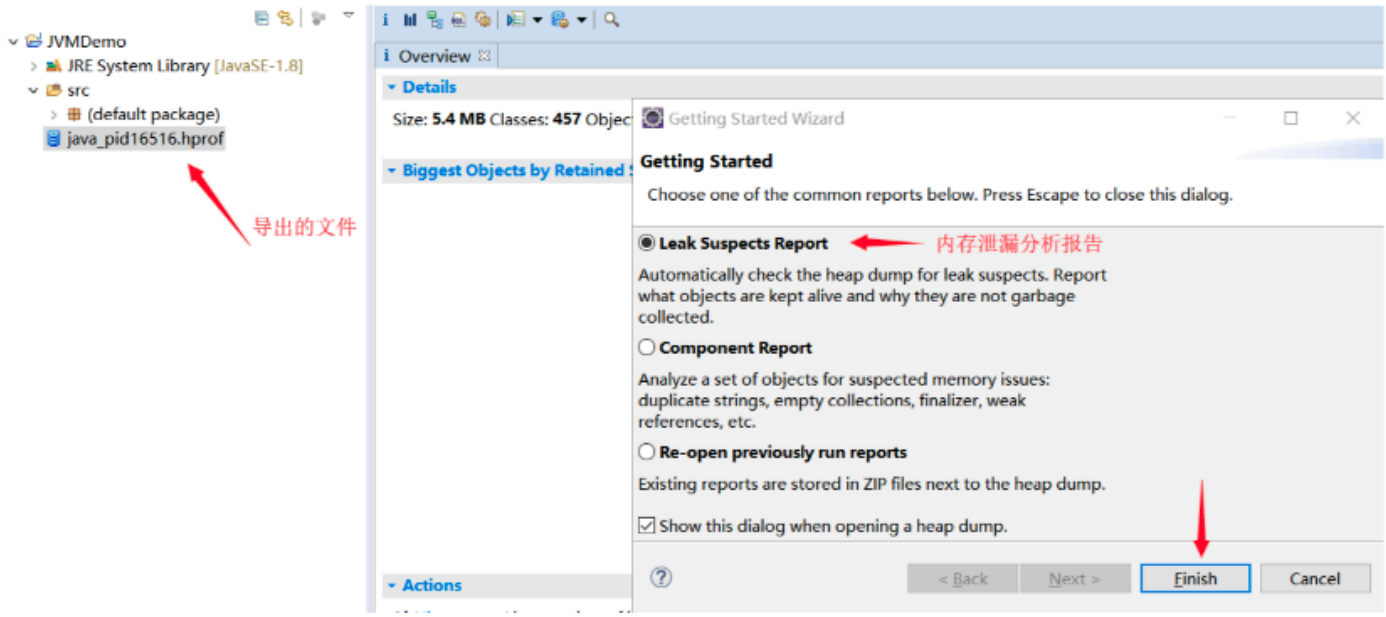
```
14                  e.printStackTrace();
15              }
16          }
17      }
```

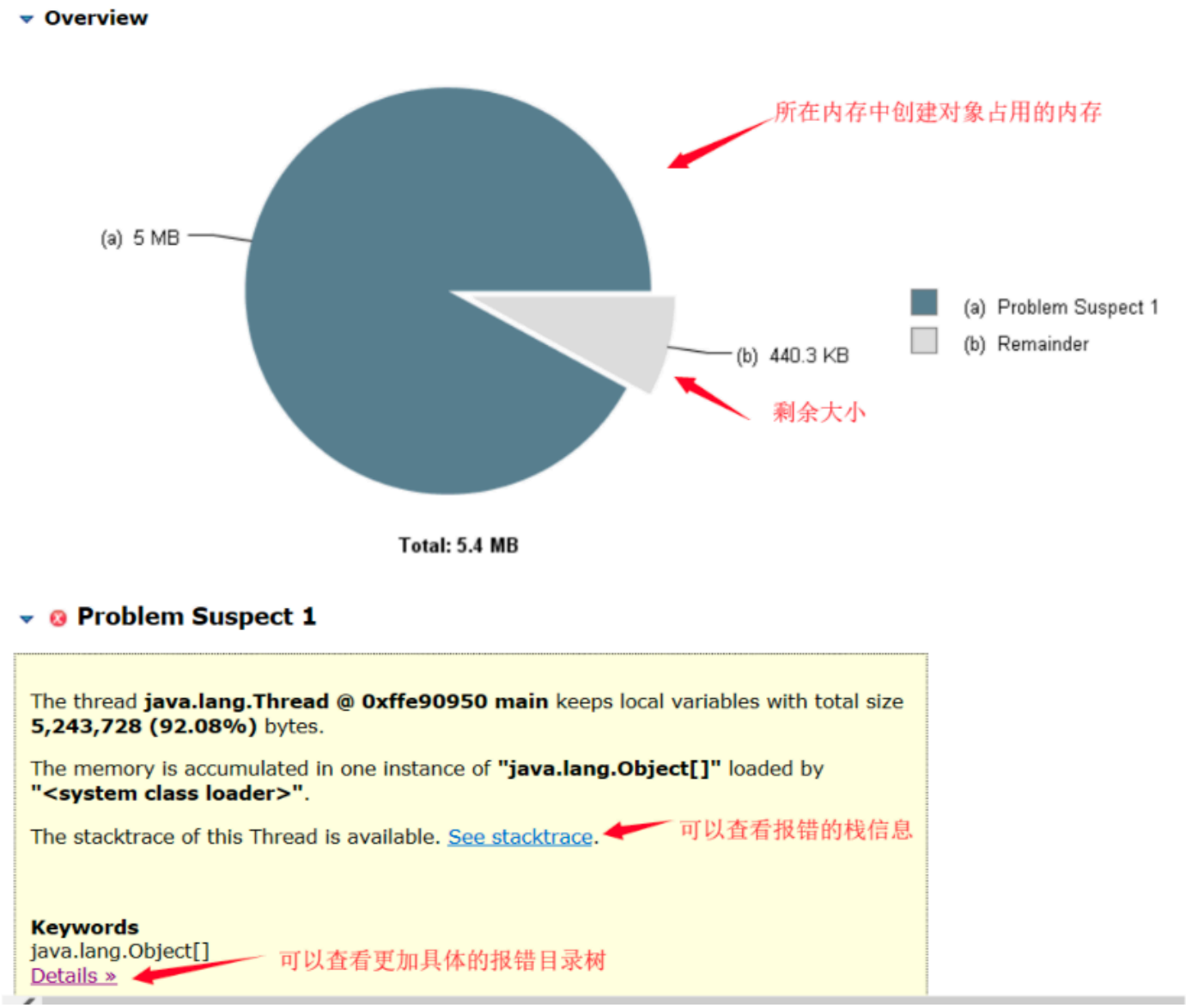
-XX:+HeapDumpOnOutOfMemoryErrorOOM时导出堆到文件。

运行设置

-Xms1m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError

执行完代码后刷新工程出现





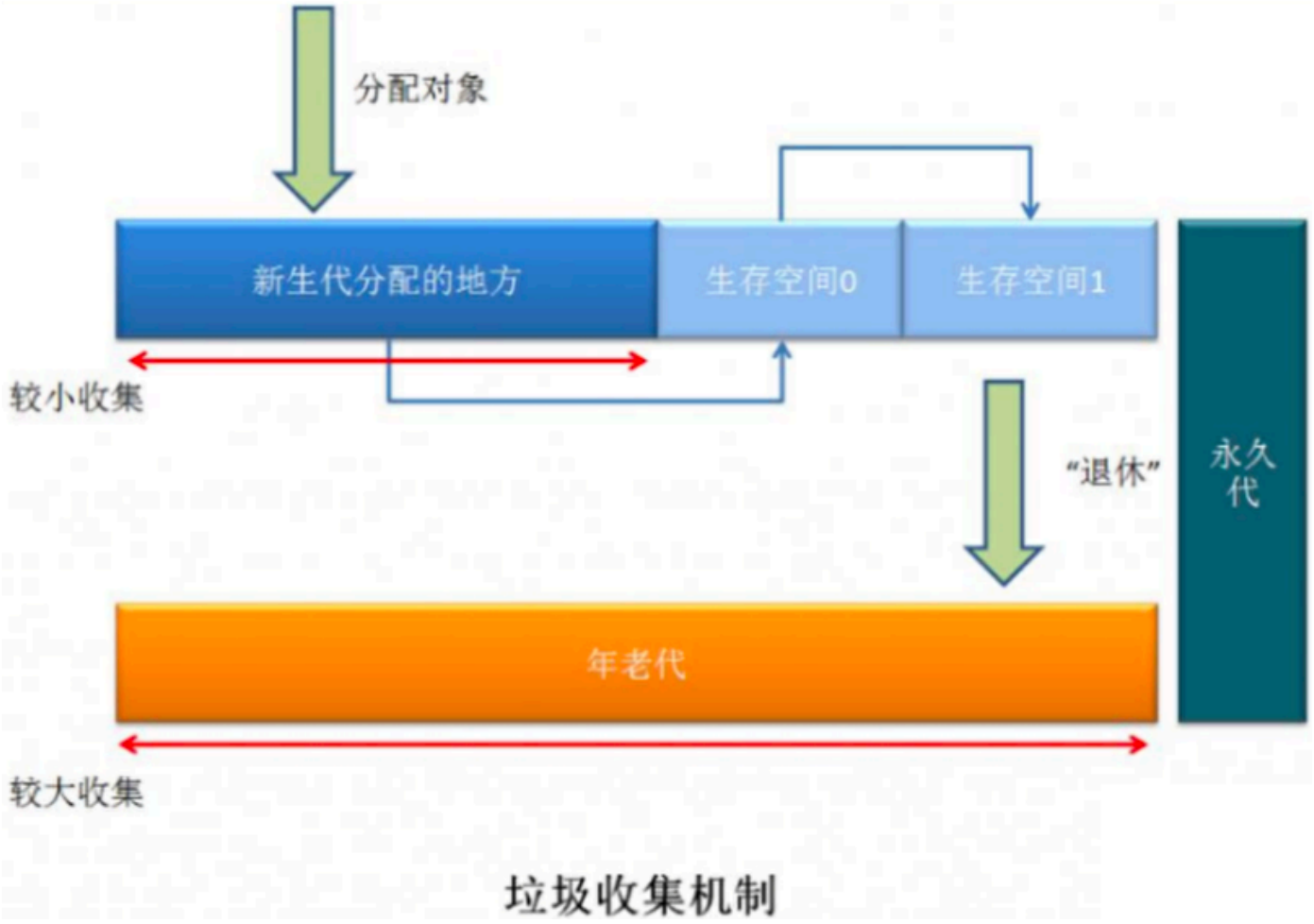
回去看代码会发现在catch中的输出是没有输出,而是一场打印是出来了。所以这个就是一个小技巧了就是通过其父类进行抓起将Exception修改成Throwable即可。

4.2.10. 垃圾回收机制GC

4.2.10.1. GC是什么

Garbage Collection， 简称GC， 是分代垃圾收集算法。 频繁收集Yong区， 较少收集Old区， 基本不动Perm区。

4.2.10.2. GC算法的总体概述



垃圾收集机制

JVM在进行GC时，并非每次都对上面三个内存区域一起回收的，大部分时候回收的都是指新生代。 因此GC按照回收的区域又分了两种类型，一种是普通GC（minor GC），一种是全局GC（major GC or Full GC）， 普通GC（minor GC）：只针对新生代区域的GC。 全局GC（major GC or Full GC）：针对年老代的GC，偶尔伴随对新生代的GC以及对永久代的GC。

4.2.10.3. GC4大算法

4.2.10.3.1. 引用计数法（了解）

无法解决循环引用的问题， 不被Java采纳。

4.2.10.3.2. 复制算法（Copying）

年轻代中使用的是MinorGC,这种GC算法采用的是复制算法。

Minor GC会把Eden中的所有活的对象都移到Survivor区域中，如果Survivor区中放不下，那么剩下的活的对象就被移到Old generation中，也即一旦收集后，Eden是就变成空的了。 当对象在 Eden (包括一个 Survivor 区域，这里假设是 from 区域) 出生后，在经过一次 Minor GC 后，如果对象还存活，并且能够被另外一块 Survivor 区域所容纳(上面已经假设为 from 区域，这里应为 to 区域，即 to 区域有足够的内存空间来存储 Eden 和 from 区域中存活的对象)，则使用复制算法将这些仍然还存活的对象复制到另外一块 Survivor 区域 (即 to 区域) 中，然后清理所使用过的 Eden 以及 Survivor 区域 (即 from 区域)，并且将这些对象的年龄设置为1，以后对象在 Survivor 区每熬过一次 Minor GC，就将对象的年龄 + 1，当对象的年龄达到某个值时 (默认是 15 岁，通过-XX:MaxTenuringThreshold 来设定参数)，这些对象就会成为老年代。

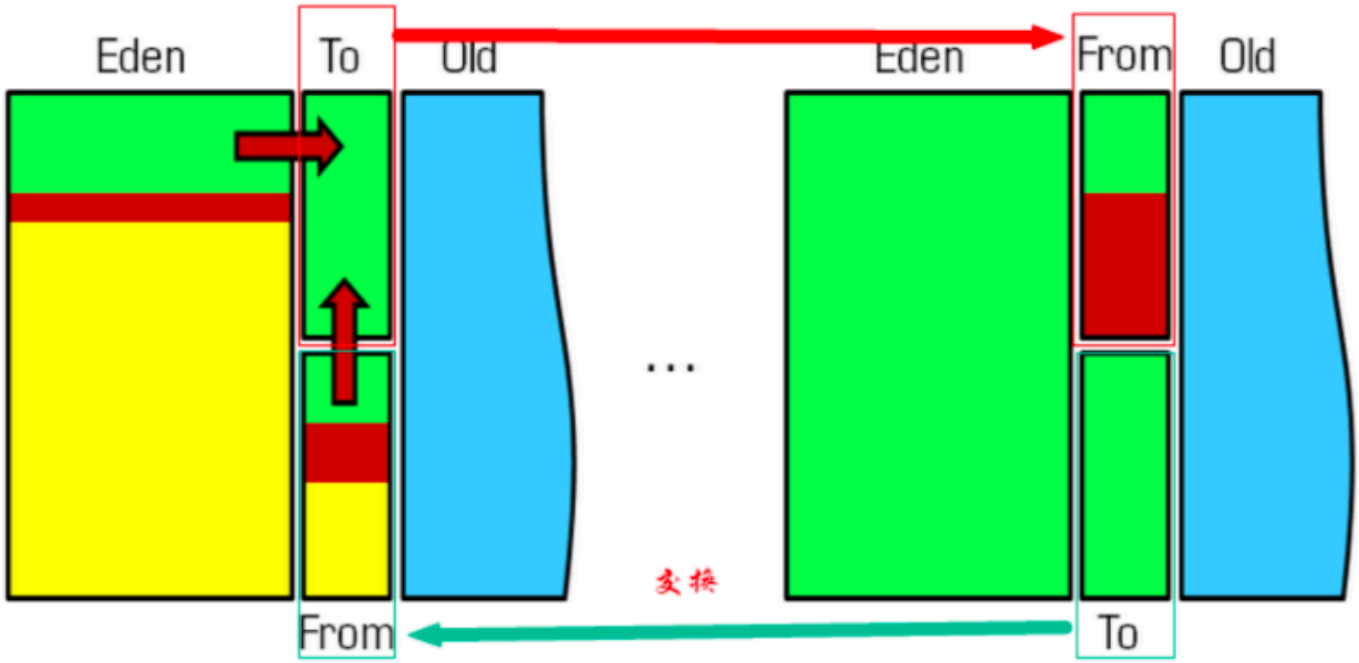
-XX:MaxTenuringThreshold — 设置对象在新生代中存活的次数

年轻代中的GC,主要是复制算法（Copying）

HotSpot JVM把年轻代分为了三部分：1个Eden区和2个Survivor区（分别叫from和to）。默认比例为8:1:1, 一般情况下，新创建的对象都会被分配到Eden区(一些大对象特殊处理),这些对象经过第一次Minor GC后，如果仍然存活，将会被移到Survivor区。对象在Survivor区中每熬过一次Minor GC，年龄就会增加1岁，当它的年龄增加到一定程度时，就会被移动到年老代中。因为年轻代中的对象基本都是朝生夕死的(90%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

复制 (Copying)

在GC开始的时候，对象只会存在于Eden区和名为“From”的Survivor区，Survivor区“To”是空的。紧接着进行GC，Eden区中所有存活的对象都会被复制到“To”，而在“From”区中，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过-XX:MaxTenuringThreshold来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。经过这次GC后，Eden区和From区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新的“To”就是上次GC前的“From”，新的“From”就是上次GC前的“To”。不管怎样，都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。



因为Eden区对象一般存活率较低，一般的，使用两块10%的内存作为空闲和活动区间，而另外80%的内存，则是用来给新建对象分配内存的。一旦发生GC，将10%的from活动区间与另外80%中存活的eden对象转移到10%的to空闲区间，接下来，将之前90%的内存全部释放，以此类推。

缺点:

复制算法它的缺点也是相当明显的。 1、它浪费了一半的内存，这太要命了。 2、如果对象的存活率很高，我们可以极端一点，假设是100%存活，那么我们需要将所有对象都复制一遍，并将所有引用地址重置一遍。复制这一工作所花费的时间，在对象存活率达到一定程度时，将会变的不可忽视。 所以从以上描述不难看出，复制算法要想使用，最起码对象的存活率要非常低才行，而且最重要的是，我们必须要克服50%内存的浪费。

4.2.10.3.3. 标记清除法（Mark-Sweep）

老年代一般是由标记清除或者是标记清除与标记整理的混合实现。

当堆中的有效内存空间（available memory）被耗尽的时候，就会停止整个程序（也被称为stop the world），然后进行两项工作，第一项则是标记，第二项则是清除。 标记：从引用根节点开始标记所有被引用的对象。标记的过程其实就是遍历所有的GC Roots，然后将所有GC Roots可达的对象 标记为存活的对象。 清除：遍历整个堆，把未标记的对象清除。 缺点：此算法需要暂停整个应用，会产生内存碎片用通俗的话解释一下标记/清除算法，就是当程序运行期间，若可以使用的内存被耗尽的时候，GC线程就会被触发并将程序暂停，随后将依旧存活的对象标记一遍，最终再将堆中所有没被标记的对象全部清除掉，接下来便让程序恢复运行。

缺点:

1. 首先，它的缺点就是效率比较低（递归与全堆对象遍历），而且在进行GC的时候，需要停止应用程序，这会导致用户体验非常差劲
2. 其次，主要的缺点则是这种方式清理出来的空闲内存是不连续的，这点不难理解，我们的死亡对象都是随即的出现在内存的各个角落的，现在把它们清除之后，内存的布局自然会乱七八糟。而为了应付这一点，JVM就不得不维持一个内存的空闲列表，这又是一种开销。而且在分配

数组对象的时候，寻找连续的内存空间会不太好找。

4.2.10.3.4. 标记压缩（Mark-Compact）

老年代一般是由标记清除或者是标记清除与标记整理的混合实现。

在整理压缩阶段，不再对标记的对像做回收，而是通过所有存活对像都向一端移动，然后直接清除边界以外的内存。 可以看到，标记的存活对象将会被整理，按照内存地址依次排列，而未被标记的内存会被清理掉。如此一来，当我们需要给新对象分配内存时，JVM只需要持有一个内存的起始地址即可，这比维护一个空闲列表显然少了许多开销。

标记/整理算法不仅可以弥补标记/清除算法当中，内存区域分散的缺点，也消除了复制算法当中，内存减半的高额代价

缺点:

标记/整理算法唯一的缺点就是效率也不高，不仅要标记所有存活对象，还要整理所有存活对象的引用地址。从效率上来说，标记/整理算法要低于复制算法。

4.2.10.3.5. 标记清除压缩（Mark-Sweep-Compact)

4.2.10.3.6. 总结

内存效率：复制算法>标记清除算法>标记整理算法（此处的效率只是简单的对比时间复杂度，实际情况不一定如此）。 内存整齐度：复制算法=标记整理算法>标记清除算法。 内存利用率：标记整理算法=标记清除算法>复制算法。

可以看出，效率上来说，复制算法是当之无愧的老大，但是却浪费了太多内存，而为了尽量兼顾上面所提到的三个指标，标记/整理算法相对来说更平滑一些，但效率上依然不尽如人意，它比复制算法多了一个标记的阶段，又比标记/清除多了一个整理内存的过程

难道就没有一种最优算法吗？ 猜猜看，下面还有

回答：无，没有最好的算法，只有最合适的算法。==========>分代收集算法。

年轻代(Young Gen)

年轻代特点是区域相对老年代较小，对像存活率低。这种情况复制算法的回收整理，速度是最快的。复制算法的效率只和当前存活对像大小有关，因而很适用于年轻代的回收。而复制算法内存利用率不高的问题，通过hotspot中的两个survivor的设计得到缓解。

老年代(Tenure Gen)

老年代的特点是区域较大，对像存活率高。

这种情况，存在大量存活率高的对像，复制算法明显变得不合适。一般是由标记清除或者是标记清除与标记整理的混合实现。

Mark阶段的开销与存活对像的数量成正比，这点上说来，对于老年代，标记清除或者标记整理有一些不符，但可以通过多核/线程利用，对并发、并行的形式提标记效率。

Sweep阶段的开销与所管理区域的大小形正相关，但Sweep“就地处决”的特点，回收的过程没有对像的移动。使其相对其它有对像移动步骤的回收算法，仍然是效率最好的。但是需要解决内存碎片问题。

Compact阶段的开销与存活对像的数据成开比，如上一条所描述，对于大量对像的移动是很大开销的，做为老年代的第一选择并不合适。

基于上面的考虑，老年代一般是由标记清除或者是标记清除与标记整理的混合实现。以hotspot中的CMS回收器为例，CMS是基于Mark-Sweep实现的，对于对像的回收效率很高，而对于碎片问题，CMS采用基于Mark-Compact算法的Serial Old回收器做为补偿措施：当内存回收不佳（碎片导致的Concurrent Mode Failure时），将采用Serial Old执行Full GC以达到对老年代内存的整理。

4.2.11. 扩展堆外内存

4.2.11.1. 内存分类

1. 堆内内存（on-heap memory)

   回顾

   堆外内存和堆内内存是相对的二个概念，其中堆内内存是平常中接触比较多的，我们在jvm参数中只要使用-Xms，-Xmx等参数就可以设置堆的大小和最大值，理解jvm的堆还需要知道下面这个公式：

   堆内内存 = 新生代+老年代+持久代(元空间)

   使用堆内内存（on-heap memory）的时候，完全遵守JVM虚拟机的内存管理机制，采用垃圾回收器（GC）统一进行内存管理，GC会在某些特定的时间点进行一次彻底回收，也就是Full GC，GC会对所有分配的堆内内存进行扫描，在这个过程中会对JAVA应用程序的性能造成一定影响，还可能会产生Stop The World。

2. 堆外内存（off-heap memory）

   和堆内内存相对应，堆外内存就是把内存对象分配在Java虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机），这样做的结果就是能够在一定程度上减少垃圾回收对应用程序造成的影响。

4.2.11.2. 为什么使用堆外内存

1. 减少了垃圾回收

   使用堆外内存的话，堆外内存是直接受操作系统管理(而不是虚拟机)。这样做的结果就是能保持一个较小的堆内内存，以减少垃圾收集对应用的影响。

2. 提升复制速度(io效率)

   堆内内存由JVM管理，属于“用户态”；而堆外内存由OS管理，属于“内核态”。如果从堆内向磁盘写数据时，数据会被先复制到堆外内存，即内核缓冲区，然后再由OS写入磁盘，使用堆外内存避免了这个操作。

> ps:堆外内存是可以通过java中的未公开的Unsafe和NIO包下ByteBuffffer来创建堆外内存

五 实战应用