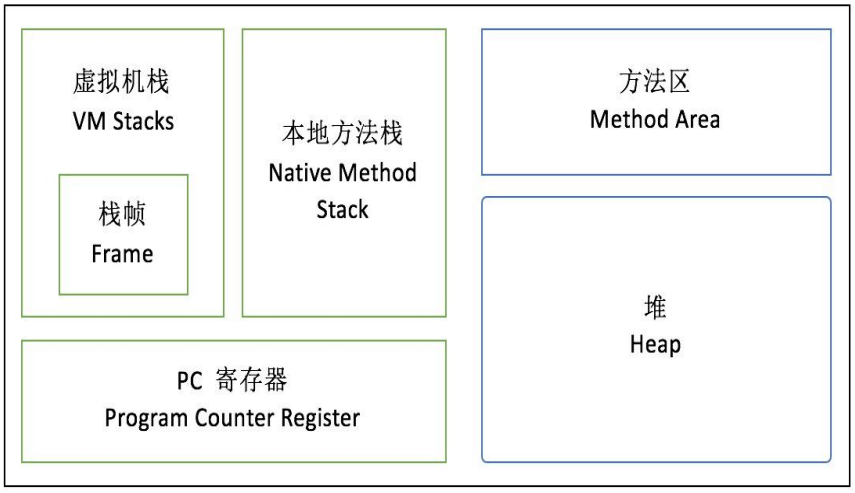
liuxiaopeng

Java8内存模型—永久代(PermGen)和元空间(Metaspace)

一、JVM 内存模型

根据 JVM 规范，JVM 内存共分为虚拟机栈、堆、方法区、程序计数器、本地方法栈五个部分。



1、虚拟机栈：每个线程有一个私有的栈，随着线程的创建而创建。栈里面存着的是一种叫“栈帧”的东西，每个方法会创建一个栈帧，栈帧中存放了局部变量表（基本数据类型和对象引用）、操作数栈、方法出口等信息。栈的大小可以固定也可以动态扩展。当栈调用深度大于JVM所允许的范围，会抛出StackOverflowError的错误，不过这个深度范围不是一个恒定的值，我们通过下面这段程序可以测试一下这个结果：

栈溢出测试源码：

```
1  package com.paddx.test.memory;
2
3  public class StackErrorMock {
4      private static int index = 1;
```

公告

昵称： liuxiaopeng
园龄： 6年5个月
粉丝： 570
关注： 2
+加关注

搜索

找找看
谷歌搜索

最新随笔

1.Spring Boot实战：模板引擎
2.Spring Boot实战：拦截器与过滤器
3.Spring Boot实战：静态资源处理
4.Spring Boot实战：集成Swagger2
5.Spring Boot实战：Restful API的构建
6.Spring Boot实战：数据库操作
7.Spring Boot实战：逐行释义HelloWorld

```
5
6       public void call(){
7           index++;
8           call();
9       }
10
11      public static void main(String[] args) {
12          StackErrorMock mock = new StackErrorMock();
13          try {
14              mock.call();
15          }catch (Throwable e){
16              System.out.println("Stack deep : "+index);
17              e.printStackTrace();
18          }
19      }
20  }
```

代码段 1

运行三次，可以看出每次栈的深度都是不一样的，输出结果如下。

```
liuxpdeMacBook-Pro:classes liuxp$ java  com.paddx.test.memory.StackErrorMock
Stack deep : 22029
java.lang.StackOverflowError
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
liuxpdeMacBook-Pro:classes liuxp$ java  com.paddx.test.memory.StackErrorMock
Stack deep : 21964
java.lang.StackOverflowError
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
liuxpdeMacBook-Pro:classes liuxp$ java  com.paddx.test.memory.StackErrorMock
Stack deep : 18882
java.lang.StackOverflowError
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
        at com.paddx.test.memory.StackErrorMock.call(StackErrorMock.java:11)
```

至于红色框里的值是怎么出来的，就需要深入到 JVM 的源码中才能探讨，这里不作详细阐述。

虚拟机栈除了上述错误外，还有另一种错误，那就是当申请不到空间时，会抛出 OutOfMemoryError。这里有一个小细节需要注意，catch 捕获的是 Throwable，而不是 Exception。因为 StackOverflowError 和 OutOfMemoryError 都不属于 Exception 的子类。

2、本地方法栈：

这部分主要与虚拟机用到的 Native 方法相关，一般情况下， Java 应用程序员并不需要关心这部分的内容。

3、PC 寄存器：

PC 寄存器，也叫程序计数器。JVM支持多个线程同时运行，每个线程都有自己的程序计数器。倘若当前执行的是 JVM 的方法，则该寄存器中保存当前执行指令的地址；倘若执行的是native 方法，则PC寄存器中为空。

4、堆

8.Java集合类：AbstractCollection源码解析

9.Java集合：整体结构

10.Java 并发编程：volatile的使用及其原理

我的标签

Java(16)

spring boot(8)

Spring(8)

并发编程(4)

Rest(2)

jsp(1)

Freemaker(1)

thymeleaf(1)

模板引擎(1)

拦截器(1)

更多

积分与排名

积分 - 56087

排名 - 24262

随笔档案

2018年5月(1)

2018年1月(5)

堆内存是 JVM 所有线程共享的部分，在虚拟机启动的时候就已经创建。所有的对象和数组都在堆上进行分配。这部分空间可通过 GC 进行回收。当申请不到空间时会抛出 OutOfMemoryError。下面我们简单的模拟一个堆内存溢出的情况：

```
package com.paddx.test.memory;

import java.util.ArrayList;
import java.util.List;

public class HeapOomMock {
    public static void main(String[] args) {
        List<byte[]> list = new ArrayList<byte[]>();
        int i = 0;
        boolean flag = true;
        while (flag){
            try {
                i++;
                list.add(new byte[1024 * 1024]);//每次增加一个1M大小的数组对象
            }catch (Throwable e){
                e.printStackTrace();
                flag = false;
                System.out.println("count="+i);//记录运行的次数
            }
        }
    }
}
```

代码段 2

运行上述代码，输出结果如下：

```
liuxpdeMacBook-Pro:classes liuxp$ java -Xms16m -Xmn8m -Xmx16m com.paddx.test.memory.HeapOomMock
java.lang.OutOfMemoryError: Java heap space
        at com.paddx.test.memory.HeapOomMock.main(HeapOomMock.java:17)
count=14
liuxpdeMacBook-Pro:classes liuxp$
```

注意，这里我指定了堆内存的大小为16M，所以这个地方显示的count=14（这个数字不是固定的），至于为什么会是14或其他数字，需要根据 GC 日志来判断，具体原因会在下篇文章中给大家解释。

5、方法区：

方法区也是所有线程共享。主要用于存储类的信息、常量池、方法数据、方法代码等。方法区逻辑上属于堆的一部分，但是为了与堆进行区分，通常又叫“非堆”。 关于方法区内存溢出的问题会在下文中详细探讨。

二、PermGen（永久代）

绝大部分 Java 程序员应该都见过 "java.lang.OutOfMemoryError: PermGen space "这个异常。这里的 “PermGen space”其实指的就是方法区。不过方法区和“PermGen space”又有着本质的区别。前者是 JVM 的规范，而后者则是 JVM 规范的一种实现，并且只有 HotSpot 才有 “PermGen space”，而对于其他类型的虚拟机，如 JRockit（Oracle）、J9（IBM）并没有“PermGen space”。由于方法区主要存储类的相关信息，所以对于动态生成类的情况比较容易出现永久代的内存溢出。最典型的场景就是，在 jsp 页面比较多的情况，容易出现永久代内存溢出。我们现在通过动态生成类来模拟 “PermGen space”的内存溢出：

```
package com.paddx.test.memory;

public class Test {
```

2017年12月(1)
2016年6月(1)
2016年5月(3)
2016年4月(4)
2016年3月(3)

阅读排行榜

1. Spring Boot实战：拦截器与过滤器(220736)
2. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(196394)
3. Java并发编程：Synchronized及其实现原理(118146)
4. Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）(84133)
5. Java 并发编程：volatile的使用及其原理(61450)

评论排行榜

1. Java并发编程：Synchronized及其实现原理(34)
2. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(30)
3. Java 并发编程：volatile的使用及其原理(22)
4. Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）(19)
5. Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)(17)

```
}
```

代码段 3

```
package com.paddx.test.memory;

import java.io.File;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.ArrayList;
import java.util.List;

public class PermGenOomMock{
    public static void main(String[] args) {
        URL url = null;
        List<ClassLoader> classLoaderList = new ArrayList<ClassLoader>();
        try {
            url = new File("/tmp").toURI().toURL();
            URL[] urls = {url};
            while (true){
                ClassLoader loader = new URLClassLoader(urls);
                classLoaderList.add(loader);
                loader.loadClass("com.paddx.test.memory.Test");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

代码段 4

运行结果如下：

```
liuxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.7.0_75"
Java(TM) SE Runtime Environment (build 1.7.0_75-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.75-b04, mixed mode)
liuxpdeMacBook-Pro:classes liuxp$ java -XX:PermSize=8m -XX:MaxPermSize=8m  com.paddx.test.memory.PermGenOomMock
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
        at java.lang.ClassLoader.defineClass(ClassLoader.java:800)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:449)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:71)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
        at com.paddx.test.memory.PermGenOomMock.main(PermGenOomMock.java:19)
```

本例中使用的 JDK 版本是 1.7，指定的 PermGen 区的大小为 8M。通过每次生成不同URLClassLoader对象来加载Test类，从而生成不同的类对象，这样就能看到我们熟悉的 "java.lang.OutOfMemoryError: PermGen space " 异常了。这里之所以采用 JDK 1.7，是因为在 JDK 1.8 中， HotSpot 已经没有 “PermGen space”这个区间了，取而代之是一个叫做 Metaspace（元空间） 的东西。下面我们就来看看 Metaspace 与 PermGen space 的区别。

三、Metaspace（元空间）

其实，移除永久代的工作从JDK1.7就开始了。JDK1.7中，存储在永久代的部分数据就已经转移到了Java Heap或者是 Native Heap。但永久代仍存在于JDK1.7中，并没完全移除，譬如符号引用(Symbols)转移到了native heap；字面量(interned strings)转移到了

推荐排行榜

1. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(89)

2. Java并发编程：Synchronized及其实现原理(80)

3. Java 并发编程：核心理论(45)

4. Spring Boot实战：拦截器与过滤器(42)

5. Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)(33)

最新评论

1. Re:Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)

线程的5种状态：新建、就绪、运行、死亡、阻塞 wait(让出CPU、进入阻塞、释放锁)/notify(进入就绪)/notifyAll（进入就绪） synchronized sleep（让出CPU、不释...

--harryzhou6

2. Re:Java 并发编程：volatile的使用及其原理

@JohnNaruto 你的“a = 3、b = a”，看起来没用++，但这和++一样，是两步操作。这两操作之间存在被另一个线程插入的风险，因此只是保证了可见性，但却不是原子的。...

--fddgfgfj

3. Re:Spring Boot实战：Restful API的构建

你好，能分享一下工程么，谢谢

--pisceakin

4. Re:Java并发编程：Synchronized及其实现原理

java heap；类的静态变量(class statics)转移到了java heap。我们可以通过一段程序来比较 JDK 1.6 与 JDK 1.7及 JDK 1.8 的区别，以字符串常量为例：

```
package com.paddx.test.memory;

import java.util.ArrayList;
import java.util.List;

public class StringOomMock {
    static String  base = "string";
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        for (int i=0;i< Integer.MAX_VALUE;i++){
            String str = base + base;
            base = str;
            list.add(str.intern());
        }
    }
}
```

这段程序以2的指数级不断的生成新的字符串，这样可以比较快速的消耗内存。我们通过 JDK 1.6、JDK 1.7 和 JDK 1.8 分别运行：

JDK 1.6 的运行结果：

```
liuxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.6.0_65"
Java(TM) SE Runtime Environment (build 1.6.0_65-b14-466.1-11M4716)
Java HotSpot(TM) 64-Bit Server VM (build 20.65-b04-466.1, mixed mode)
liuxpdeMacBook-Pro:classes liuxp$ java -XX:PermSize=8m -XX:MaxPermSize=8m  -Xmx16m  com.paddx.test.memory.StringOomMock
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
        at java.lang.String.intern(Native Method)
        at com.paddx.test.memory.StringOomMock.main(StringOomMock.java:17)
```

JDK 1.7的运行结果：

```
liuxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.7.0_75"
Java(TM) SE Runtime Environment (build 1.7.0_75-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.75-b04, mixed mode)
liuxpdeMacBook-Pro:classes liuxp$ java -XX:PermSize=8m -XX:MaxPermSize=8m  -Xmx16m  com.paddx.test.memory.StringOomMock
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Arrays.java:2367)
        at java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:130)
        at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:114)
        at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:415)
        at java.lang.StringBuilder.append(StringBuilder.java:132)
        at com.paddx.test.memory.StringOomMock.main(StringOomMock.java:15)
```

JDK 1.8的运行结果：

```
liuxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.8.0_40"
Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)
liuxpdeMacBook-Pro:classes liuxp$ java -XX:PermSize=8m -XX:MaxPermSize=8m  -Xmx16m  com.paddx.test.memory.StringOomMock
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=8m; support was removed in 8.0
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=8m; support was removed in 8.0
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Arrays.java:3332)
        at java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:137)
        at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:121)
        at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:421)
        at java.lang.StringBuilder.append(StringBuilder.java:136)
        at com.paddx.test.memory.StringOomMock.main(StringOomMock.java:15)
```

从上述结果可以看出，JDK 1.6下，会出现“PermGen Space”的内存溢出，而在 JDK 1.7和 JDK 1.8 中，会出现堆内存溢出，并且 JDK 1.8中 PermSize 和 MaxPermGen 已经无效。因此，可以大致验证 JDK 1.7 和 1.8 将字符串常量由永久代转移到堆中，并且 JDK 1.8 中已经不存在永久代的结论。现在我们看看元空间到底是一个什么东西？

666

--蜗牛0121

5. Re:Spring Boot实战：拦截器与过滤器

@稚屿丶 我试了，确实是FilterRegistrationBean配置的过滤器在@WebFilter的过滤器之前的...

--fjdsklfj

元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制，但可以通过以下参数来指定元空间的大小：

-XX:MetaspaceSize，初始空间大小，达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过MaxMetaspaceSize时，适当提高该值。

-XX:MaxMetaspaceSize，最大空间，默认是没有限制的。

除了上面两个指定大小的选项以外，还有两个与 GC 相关的属性：

-XX:MinMetaspaceFreeRatio，在GC之后，最小的Metaspace剩余空间容量的百分比，减少为分配空间所导致的垃圾收集

-XX:MaxMetaspaceFreeRatio，在GC之后，最大的Metaspace剩余空间容量的百分比，减少为释放空间所导致的垃圾收集

现在我们在 JDK 8下重新运行一下代码段 4，不过这次不再指定 PermSize 和 MaxPermSize。而是指定 MetaSpaceSize 和 MaxMetaSpaceSize的大小。输出结果如下：

```
liuxpdeMacBook-Pro:classes liuxp$ java -version
java version "1.8.0_40"
Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)
liuxpdeMacBook-Pro:classes liuxp$ java -XX:MetaspaceSize=8m -XX:MaxMetaspaceSize=8m com.paddx.test.memory.PermGenOomMock
Exception in thread "main" java.lang.OutOfMemoryError: Metaspace
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:760)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:362)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
```

从输出结果，我们可以看出，这次不再出现永久代溢出，而是出现了元空间的溢出。

四、总结

通过上面分析，大家应该大致了解了 JVM 的内存划分，也清楚了 JDK 8 中永久代向元空间的转换。不过大家应该都有一个疑问，就是为什么要做这个转换？所以，最后给大家总结以下几点原因：

1、字符串存在永久代中，容易出现性能问题和内存溢出。

2、类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。

3、永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

4、Oracle 可能会将HotSpot 与 JRockit 合二为一。

标签: JVM, 内存模型

好文要顶 关注我 收藏该文

liuxiaopeng
关注 - 2
粉丝 - 570
+加关注

89 0

« 上一篇: 从字节码层面看“HelloWorld”

» 下一篇: 通过反编译深入理解Java String及intern

posted @ 2016-03-27 01:04 liuxiaopeng 阅读(196406) 评论(30) 编辑 收藏 举报

刷新评论 刷新页面 返回顶部

登录后才能查看或发表评论，立即 登录 或者 逛逛 博客园首页