十种JVM内存溢出的情况，你碰到过几种？

开物笔记 发布于 2018-12-02

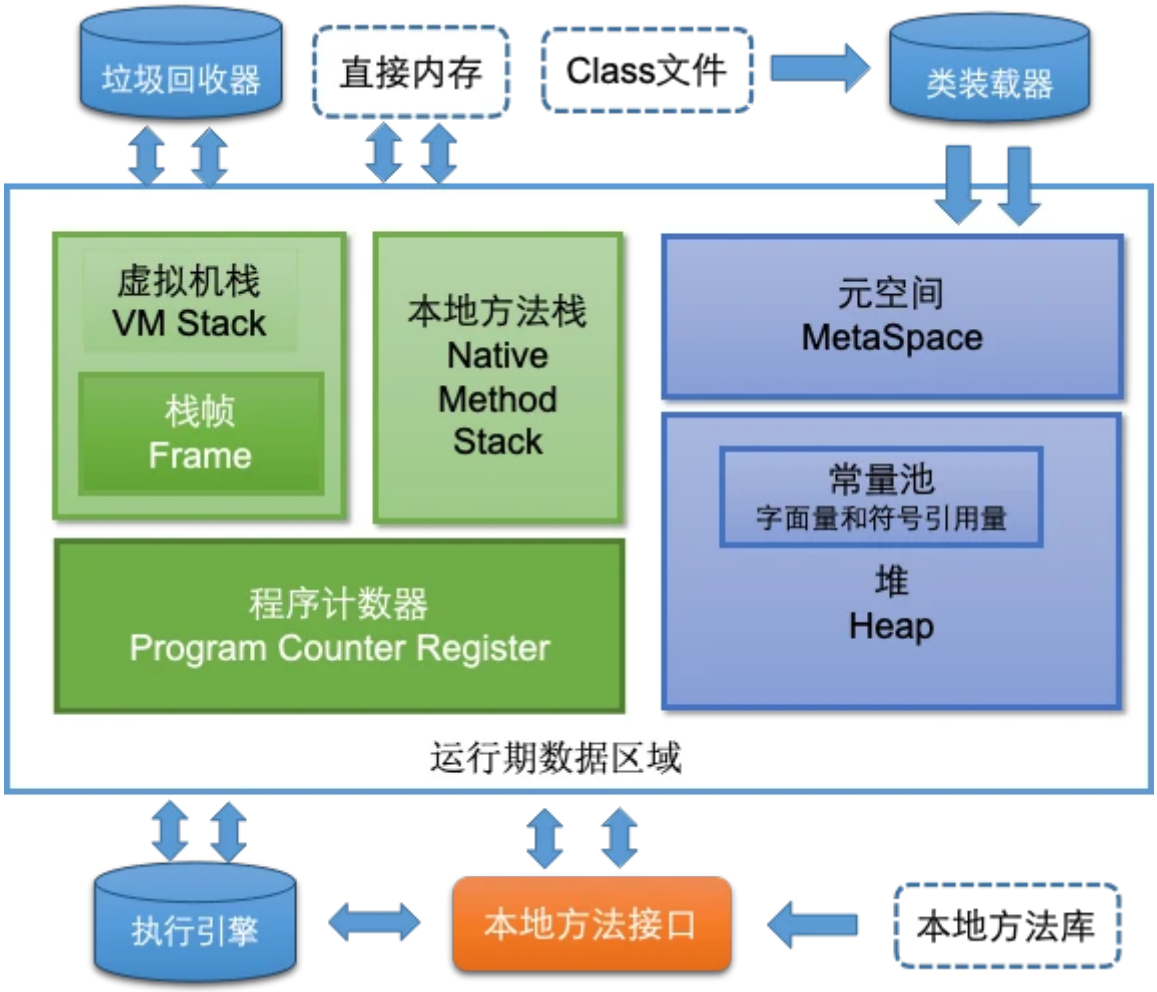
导言：

对于java程序员来说，在虚拟机自动内存管理机制的帮助下，不需要自己实现释放内存，不容易出现内存泄漏和内存溢出的问题，由虚拟机管理内存这一切看起来非常美好，但是一旦出现内存溢出或者内存泄漏的问题，对于不熟悉jvm虚拟机是怎么使用内存的话，那么排查错误将会是一项非常艰巨的任务。所以在了解内存溢出之前先要搞明白JVM的内存模型。

JVM（Java虚拟机）是一个抽象的计算模型。就如同一台真实的机器，它有自己的指令集和执行引擎，可以在运行时操控内存区域。目的是为构建在其上运行的应用程序提供一个运行环境。JVM可以解读指令代码并与底层进行交互：包括操作系统平台和执行指令并管理资源的硬件体系结构。

JVM内存模型

根据 JVM8 规范，JVM 运行时内存共分为虚拟机栈、堆、元空间、程序计数器、本地方法栈五个部分。还有一部分内存叫直接内存，属于操作系统的本地内存，也是可以直接操作的。



1. 元空间(Metaspace)

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。

2.虚拟机栈(JVM Stacks)

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。

1. 本地方法栈(Native Method Stack)

与虚拟机栈类似，区别是虚拟机栈执行java方法，本地方法站执行native方法。在虚拟机规范中对本地方法栈中方法使用的语言、使用方法与数据结构没有强制规定，因此虚拟机可以自由实现它。

1. 程序计数器(Program Counter Register)

程序计数器可以看成是当前线程所执行的字节码的行号指示器。在任何一个确定的时刻，一个处

复到正确的执行位置，每条线程都需要一个独立的程序计数器，我们称这类内存区域为“线程私有”内存。

5.堆内存(Heap)

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。堆是JVM内存占用最大，管理最复杂的一个区域。其唯一的用途就是存放对象实例：所有的对象实例及数组都在对上进行分配。jdk1.8后，字符串常量池从永久代中剥离出来，存放在队中。

6.直接内存(Direct Memory)

直接内存并不是虚拟机运行时数据区的一部分，也不是Java 虚拟机规范中农定义的内存区域。在JDK1.4 中新加入了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，然后通脱一个存储在Java堆中的DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。

内存溢出的十个场景

JVM运行时首先需要类加载器（classLoader）加载所需类的字节码文件。加载完毕交由执行引擎执行，在执行过程中需要一段空间来存储数据（类比CPU与主存）。这段内存空间的分配和释放过程正是我们需要关心的运行时数据区。内存溢出的情况就是从类加载器加载的时候开始出现的，内存溢出分为两大类：OutOfMemoryError和StackOverflowError。以下举出10个内存溢出的情况，并通过实例代码的方式讲解了是如何出现内存溢出的。

1.java堆内存溢出

当出现java.lang.OutOfMemoryError:Java heap space异常时，就是堆内存溢出了。

1.问题描述

1.设置的jvm内存太小，对象所需内存太大，创建对象时分配空间，就会抛出这个异常。

2.流量/数据峰值，应用程序自身的处理存在一定的限额，比如一定数量的用户或一定数量的数据。而当用户数量或数据量突然激增并超过预期的阈值时，那么就会峰值停止前正常运行的操作将停止并触发java . lang.OutOfMemoryError:Java堆空间错误

2.示例代码

```java
package com.zhujiukeji.oom;

import java.util.ArrayList;
import java.util.List;
//堆溢出
public class HeapOomError {
    public static void main(String[] args) {
        List<byte[]> list = new ArrayList<>();
        int i = 0;
        while (true) {
            list.add(new byte[5 * 1024 * 1024]);
            System.out.println("count is: " + (++i));
        }
    }
}
```

以上这个示例，如果一次请求只分配一次5m的内存的话，请求量很少垃圾回收正常就不会出错，但是一旦并发上来就会超出最大内存值，就会抛出内存溢出。

3.解决方法

首先，如果代码没有什么问题的情况下，可以适当调整-Xms和-Xmx两个jvm参数，使用压力测试来调整这两个参数达到最优值。

其次，尽量避免大的对象的申请，像文件上传，大批量从数据库中获取，这是需要避免的，尽量分块或者分批处理，有助于系统的正常稳定的执行。

最后，尽量提高一次请求的执行速度，垃圾回收越早越好，否则，大量的并发来了的时候，再来新的请求就无法分配内存了，就容易造成系统的雪崩。

2.java堆内存泄漏

1.问题描述

Java中的内存泄漏是一些对象不再被应用程序使用但垃圾收集无法识别的情况。因此，这些未使用的对象仍然在Java堆空间中无限期地存在。不停的堆积最终会触发java . lang.OutOfMemoryError。

2.示例代码

```java
package com.zhujiukeji.oom;
import java.util.HashMap;
//内存泄漏导致的内存溢出
public class MemoryLeakOomError {
    static class Key {
        Integer id;
        Key(Integer id) {
            this.id = id;
        }
        @Override
        public int hashCode() {
            return id.hashCode();
        }
        @Override
        public boolean equals(Object o) {
           boolean response = false;
           if (o instanceof Key) {
              response = (((Key)o).id).equals(this.id);
           }
           return response;
        }
    }
    @SuppressWarnings({ "unchecked", "rawtypes" })
    public static void main(String[] args) {
        Map m = new HashMap();
        while (true) {
            for (int i = 0; i < 10000; i++) {
                if (!m.containsKey(new Key(i))) {
                    m.put(new Key(i), "Number:" + i);
                }
            }
        }
    }
}
```

当执行上面的代码时，可能会期望它永远运行，不会出现任何问题，假设单纯的缓存解决方案只将底层映射扩展到10,000个元素，而不是所有键都已经在HashMap中。然而事实上元素将继续被添加，因为key类并没有重写它的equals()方法。

随着时间的推移，随着不断使用的泄漏代码，“缓存”的结果最终会消耗大量Java堆空间。当泄漏内存填充堆区域中的所有可用内存时，垃圾收集无法清理它，java . lang.OutOfMemoryError。

3.解决办法

相对来说对应的解决方案比较简单：重写equals方法即可：

```java
package com.zhujiukeji.oom;
import java.util.HashMap;
//内存泄漏导致的内存溢出
public class MemoryLeakOomError {
    static class Key {
        Integer id;
        Key(Integer id) {
            this.id = id;
        }
        @Override
        public int hashCode() {
            return id.hashCode();
        }
        @Override
        public boolean equals(Object o) {
           boolean response = false;
           if (o instanceof Key) {
              response = (((Key)o).id).equals(this.id);
           }
           return response;
        }
    }
    @SuppressWarnings({ "unchecked", "rawtypes" })
    public static void main(String[] args) {
        Map m = new HashMap();
        while (true) {
            for (int i = 0; i < 10000; i++) {
                if (!m.containsKey(new Key(i))) {
                    m.put(new Key(i), "Number:" + i);
                }
            }
        }
    }
}
```

3.垃圾回收超时内存溢出

1、问题描述

当应用程序耗尽所有可用内存时，GC开销限制超过了错误，而GC多次未能清除它，这时便会引发java.lang.OutOfMemoryError。当JVM花费大量的时间执行GC，而收效甚微，而一旦整个GC的过程超过限制便会触发错误(默认的jvm配置GC的时间超过98%，回收堆内存低于2%)。

2.示例代码

```java
package com.zhujiukeji.oom;

import java.util.Map;
import java.util.Random;

public class OverheadLimitOomError {
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static void main(String args[]) throws Exception {
        Map map = System.getProperties();
        Random r = new Random();
        while (true) {
          map.put(r.nextInt(), "微信公众号:xtech100");
        }
      }

}

```

3.解决方法

要减少对象生命周期，尽量能快速的进行垃圾回收。

4.Metaspace内存溢出

1.问题描述

元空间的溢出，系统会抛出java.lang.OutOfMemoryError: Metaspace。出现这个异常的问题的原因是系统的代码非常多或引用的第三方包非常多或者通过动态代码生成类加载等方法，导致元空间的内存占用很大。

2.示例代码

以下是用循环动态生成class的方式来模拟元空间的内存溢出的。

```java
package com.zhujiukeji.oom;
import java.lang.management.ClassLoadingMXBean;
public class MetaSpaceOomError{
    @SuppressWarnings("rawtypes")
    public static void main(String[] args) {
        ClassLoadingMXBean loadingBean = ManagementFactory.getClassLoadingMXBean();
        //循环动态产生class
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(MetaSpaceOomError.class);
            enhancer.setCallbackTypes(new Class[]{Dispatcher.class, MethodInterceptor.class});
            enhancer.setCallbackFilter(new CallbackFilter() {
                @Override
                public int accept(Method method) {
                    return 1;
                }
                @Override
                public boolean equals(Object obj) {
                    return super.equals(obj);
                }
            });
            Class clazz = enhancer.createClass();
            System.out.println(clazz.getName());
            //显示数量信息（共加载过的类型数目，当前还有效的类型数目，已经被卸载的类型数目）
            System.out.println("total: " + loadingBean.getTotalLoadedClassCount());
            System.out.println("active: " + loadingBean.getLoadedClassCount());
            System.out.println("unloaded: " + loadingBean.getUnloadedClassCount());
        }
    }
}

```

3.解决办法

默认情况下，元空间的大小仅受本地内存限制。但是为了整机的性能，尽量还是要对该项进行设置，以免造成整机的服务停机。

1）优化参数配置，避免影响其他JVM进程

-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。

2）慎重引用第三方包

3）关注动态生成类的框架

对于使用大量动态生成类的框架，要做好压力测试，验证动态生成的类是否超出内存的需求会抛出异常。

5.直接内存内存溢出

1.问题描述

在使用ByteBuffer中的allocateDirect()的时候会用到，很多javaNIO(像netty)的框架中被封装为其他的方法，出现该问题时会抛出java.lang.OutOfMemoryError: Direct buffer memory异常。

如果你在直接或间接使用了ByteBuffer中的allocateDirect方法的时候，而不做clear的时候就会出现类似的问题。

2.示例代码

```
1  package com.zhujiukeji.oom;
2
3  import sun.misc.Unsafe;
4  import java.lang.reflect.Field;
5
6  public class DirectoryMemoryOomError {
7
8      static int ONE_MB = 1024 * 1024;
9      static int index = 0;
10
11     @SuppressWarnings("restriction")
12     public static void main(String[] args) {
13         try {
14             Field field = Unsafe.class.getDeclaredField("theUnsafe");
15             field.setAccessible(true);
16             Unsafe unsafe = (Unsafe) field.get(null);
17             while (true) {
18                 index++;
19                 unsafe.allocateMemory(ONE_MB);
20             }
21         } catch (Exception e) {
22             System.out.println("index  : " + index);
23             e.printStackTrace();
24         } catch (Error e) {
25             System.out.println("index  : " + index);
26             e.printStackTrace();
27         }
28     }
29 }
30
```

如果经常有类似的操作，可以考虑设置参数：-XX:MaxDirectMemorySize，并及时clear内存。

6.栈内存溢出

1.问题描述

当一个线程执行一个Java方法时，JVM将创建一个新的栈帧并且把它push到栈顶。此时新的栈帧就变成了当前栈帧，方法执行时，使用栈帧来存储参数、局部变量、中间指令以及其他数据。

当一个方法递归调用自己时，新的方法所产生的数据(也可以理解为新的栈帧)将会被push到栈顶，方法每次调用自己时，会拷贝一份当前方法的数据并push到栈中。因此，递归的每层调用都需要创建一个新的栈帧。这样的结果是，栈中越来越多的内存将随着递归调用而被消耗，如果递归调用自己一百万次，那么将会产生一百万个栈帧。这样就会造成栈的内存溢出。

2.示例代码

```
1  package com.zhujiukeji.oom;
2  //栈溢出
3  public class StackOomError {
4      int num = 1;
5      public void testStack(){
6          num++;
7          this.testStack();
8      }
9      public static void main(String[] agrs){
10         StackOomError t = new StackOomError();
11         t.testStack();
12     }
13
14 }
15
```

3.解决办法

如果程序中确实有递归调用，出现栈溢出时，可以调高-Xss大小，就可以解决栈内存溢出的问题了。递归调用防止形成死循环，否则就会出现栈内存溢出。

7.创建本地线程内存溢出

线程基本只占用heap以外的内存区域，也就是这个错误说明除了heap以外的区域，无法为线程分配一块内存区域了，这个要么是内存本身就不够，要么heap的空间设置得太大了，导致了剩余的内存已经不多了，而由于线程本身要占用内存，所以就不够用了。

2.示例代码

```java
package com.zhujiukeji.oom;

import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

//模拟无法创建本地线程，抛出异常
public class UnableCreateNativeThreadError {

    public static void main(String[] args) {
        while(true) {
            Executor pool=Executors.newCachedThreadPool();
            pool.execute(()->System.out.println("aaaa"));
        }
    }

}
```

3.解决方法

首先检查操作系统是否有线程数的限制，使用shell也无法创建线程，如果是这个问题就需要调整系统的最大可支持的文件数。

日常开发中尽量保证线程最大数的可控制的，不要随意使用线程池。不能无限制的增长下去。

8.超出交换区内存溢出

1.问题描述

在Java应用程序启动过程中，可以通过-Xmx和其他类似的启动参数限制指定的所需的内存。而当JVM所请求的总内存大于可用物理内存的情况下，操作系统开始将内容从内存转换为硬盘。

一般来说JVM会抛出Out of swap space错误，代表应用程序向JVM native heap请求分配内存失败并且native heap也即将耗尽时，错误消息中包含分配失败的大小（以字节为单位）和请求失败的原因。

2.解决办法

存，保证应用的性能。

9.数组超限内存溢出

1.问题描述

有的时候会碰到这种内存溢出的描述Requested array size exceeds VM limit，一般来说java对应用程序所能分配数组最大大小是有限制的，只不过不同的平台限制有所不同，但通常在1到21亿个元素之间。当Requested array size exceeds VM limit错误出现时，意味着应用程序试图分配大于Java虚拟机可以支持的数组。JVM在为数组分配内存之前，会执行特定平台的检查：分配的数据结构是否在此平台是可寻址的。

2.示例代码

以下就是代码就是数组超出了最大限制。

```
1  package com.zhujiukeji.oom;
2
3  public class ArrayLimitOomError {
4
5      public static void main(String[] args) {
6          for (int i = 3; i >= 0; i--) {
7              try {
8                  int[] arr = new int[Integer.MAX_VALUE-i];
9                  System.out.format("zhujiukeji 初始化 with %,d elements.\n",
10                         Integer.MAX_VALUE-i);
11             } catch (Throwable t) {
12                 t.printStackTrace();
13             }
14         }
15     }
16 }
17
```

3.解决方法

因此数组长度要在平台允许的长度范围之内。不过这个错误一般少见的，主要是由于Java数组的索引是int类型。 Java中的最大正整数为2 ^ 31 - 1 = 2,147,483,647。 并且平台特定的限制可以非常接近这个数字，例如：我的环境上(64位macOS，运行Jdk1.8)可以初始化数组的长度高达2,147,483,645（Integer.MAX_VALUE-2）。若是在将数组的长度再增加1达到nteger.MAX_VALUE-1会出现的OutOfMemoryError。

10.系统杀死进程内存溢出

在内核中作业，其中有一个非常特殊的进程，称为“内存杀手（Out of memory killer）”。当内核检测到系统内存不足时，OOM killer被激活，检查当前谁占用内存最多然后将该进程杀掉。

一般Out of memory:Kill process or sacrifice child错会在当可用虚拟虚拟内存(包括交换空间)消耗到让整个操作系统面临风险时，会被触发。在这种情况下，OOM Killer会选择“流氓进程”并杀死它。

2.示例代码

```
1  package com.zhujiukeji.oom;
2
3  import java.util.List;
4
5  public class OsKillerOomError {
6
7      public static void main(String[] args){
8          List<int[]> l = new java.util.ArrayList();
9          for (int i = 10000; i < 100000; i++) {
10             try {
11                 l.add(new int[100000000]);
12             } catch (Throwable t) {
13                 t.printStackTrace();
14             }
15         }
16     }
17
18 }
19
```

3.解决方法

虽然增加交换空间的方式可以缓解Java heap space异常，还是建议最好的方案就是升级系统内存，让java应用有足够的内存可用，就不会出现这种问题。

总结

通过以上的10种出现内存溢出情况，大家在实际碰到问题时也就会知道怎么解决了，在实际编码中也要记得：

2.对于大的对象或者大量的内存申请，要进行优化，大的对象要分片处理，提高处理性能，减少对象生命周期。

3.尽量固定线程的数量，保证线程占用内存可控，同时需要大量线程时，要优化好操作系统的最大可打开的连接数。

4.对于递归调用，也要控制好递归的层级，不要太高，超过栈的深度。

5.分配给栈的内存并不是越大越好，因为栈内存越大，线程多，留给堆的空间就不多了，容易抛出OOM。JVM的默认参数一般情况没有问题（包括递归）。

内存溢出 java jvm调优

架构那点事

记录架构点滴，一起分享，煮酒论科技

关注专栏

开物笔记

10年架构经验，擅长微服务架构，物联网微服务架构。

373 声望 **348** 粉丝

关注作者

4 条评论

得票 最新

撰写评论 ...

15 9 3

阿杜：很有收获，谢谢

• 回复 • 2019-02-20

allen：关于堆内存描述最后一段，字符串常量池是1.6之后才从方法区移到堆当中，可以通过不同jdk版本测试出来，1.6版本的会抛出OOM:Premgen space，1.7和1.8则会抛出OOM:Java heap space

• 回复 • 2019-06-29

zazaluMonster：很有收货, 感觉写的比我的好, 如果有小白感觉看不明白的话, 可以先看看我的, 个人觉得我说的算通俗, 希望可以快速帮助你解决问题,
https://zazalu.space/2019/09/...

• 回复 • 2019-09-19

> **开物笔记**（作者）：@zazaluMonster 来做广告的吗?
>
> • 回复 • 2019-11-18

继续阅读

微服务中消息总线架构设计应用

当一个O2O电商系统到达一定规模之后，就需要考虑系统的可扩展性、松耦合和组件化。一般采用的都是基...

开物笔记　赞 13　阅读 8.5k　评论 11

微服务架构：如何用十步解耦你的系统?

耦合性，是对模块间关联程度的度量。耦合的强弱取决于模块间接口的复杂性、调用模块的方式以及通过界...

开物笔记　赞 54　阅读 7.4k　评论 9

记一次，jvm 内存溢出

3、当时的环境：打包成jar后，直接 运行 java -jar xx.jar。默认的jvm 运行参数 -Xms 。 因此给jvm分配的...

心无私天地宽　赞 3　阅读 2k

typescript 使用的几种情况

接口的创建 可以使用 type 和 interface 来创建类型 type 特有的优点: 声明基本类型别名，联合类型，元组...

Grewer　赞 2　阅读 851

JVM 栈(stack)溢出案例

Java内存溢出(OutOfMemoryError)

我需要对hive中的数据进行批量操作处理，对于没有了解过hive的同学来说，有点茫然了。于是按照常规思路...

博予liutxer　赞 1　阅读 2.6k

JVM 堆(heap)溢出案例

JDK 8的 JVM 在 JDK 7 的基础上从堆内存中移除了永久代（Perm Generation），替换为了堆内存之外的元...

Developer　阅读 3.4k

jvm学习一|jvm内存区域以及内存溢出

1、jvm内存区域 程序计数器 程序计数器中保存线程执行状态，在线程上下文切换时保存和恢复数据。 方法...

yinpursue　阅读 574