专题27：内存泄漏和内存溢出（史上最全、定期更新）

本文版本说明：V2

> 此文的格式，由markdown 通过程序转成而来，由于很多表格，没有来的及调整，出现一个格式问题，尼恩在此给大家道歉啦。
>
> 由于社群很多小伙伴，在面试，不断的交流最新的面试难题，所以，《Java面试红宝书》， 后面会不断升级，迭代。
>
> 本专题，作为 《Java面试红宝书》专题之一， 《Java面试红宝书》一共**30个面试专题，后续还会增加**

《Java面试红宝书》升级的规划为：

后续基本上，**每一个月，都会发布一次**，最新版本，可以扫描扫架构师尼恩微信， 发送 “领取电子书” 获取。

尼恩的微信二维码在哪里呢？

具体可以百度搜索 **疯狂创客圈 总目录**

面试问题交流说明：

如果遇到面试难题，或者职业发展问题， 或者中年危机问题，都可以来 疯狂创客圈社群交流，

加入交流群，加尼恩微信即可，

入交流群，加尼恩微信即可， 发送**“入群”**

内存泄漏和内存溢出

内存溢出和内存泄露的区别与联系

内存溢出：（out of memory）通俗理解就是内存不够，指程序要求的内存超出了系统所能分配的范围，通常在运行大型软件或游戏时，软件或游戏所需要的内存远远超出了你主机内安装的内存所承受大小，就叫内存溢出。比如申请一个int类型，但给了它一个int才能存放的数，就会出现内存溢出，或者是创建一个大的对象，而堆内存放不下这个对象，这也是内存溢出。

内存泄漏：（Memory Leak）是指程序中己动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

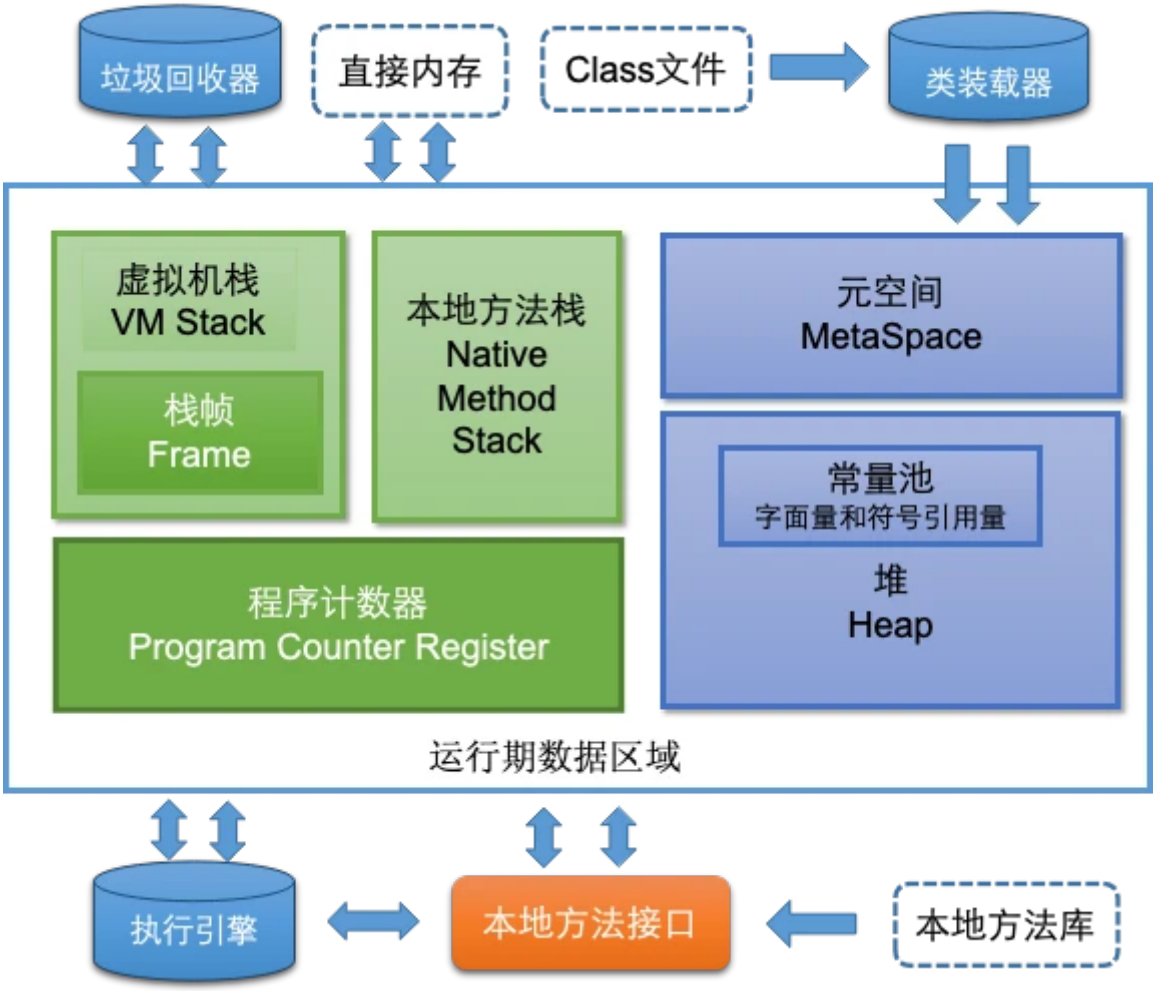
因此，我们从上面也可以推断出内存泄露可能会导致内存溢出。

二者的关系：

> 内存溢出会抛出异常，内存泄露不会抛出异常，大多数时候程序看起来是正常运行的。

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。

3. 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。

4. 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，一个处理器（对于多内核来说是一个内核）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。

内存泄露8种情况

由于java的JVM引入了垃圾回收机制，垃圾回收器会自动回收不再使用的对象，了解JVM回收机制的都知道JVM是使用引用计数法和可达性分析算法来判断对象是否是不再使用的对象，**本质都是判断一个对象是否还被引用。**那么对于这种情况下，由于代码的实现不同就会出现很多种内存泄漏问题（让JVM误以为此对象还在引用中，无法回收，造成内存泄漏）。

1、静态集合类

如HashMap、LinkedList等等。如果这些容器为静态的，那么它们的生命周期与程序一致，则容器中的对象在程序结束之前将不能被释放，从而造成内存泄漏。简单而言，长生命周期的对象持有短生命周期对象的引用，尽管短生命周期的对象不再使用，但是因为长生命周期对象持有它的引用而导致不能被回收。

2、各种连接，如数据库连接、网络连接和IO连接等。

在对数据库进行操作的过程中，首先需要建立与数据库的连接，当不再使用时，需要调用close方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器才会回收对应的对象。否则，如果在访问数据库的过程中，对Connection、Statement或ResultSet不显性地关闭，将会造成大量的对象无法被回收，从而引起内存泄漏。

3、变量不合理的作用域。

一般而言，一个变量的定义的作用范围大于其使用范围，很有可能会造成内存泄漏。另一方面，如果没有及时地把对象设置为null，很有可能导致内存泄漏的发生。

```
public class UsingRandom {

        private String msg;

        public void receiveMsg(){

        readFromNet();// 从网络中接受数据保存到msg中

        saveDB();// 把msg保存到数据库中
```

```
}

}
123456789101112131415
```

如上面这个伪代码，通过readFromNet方法把接受的消息保存在变量msg中，然后调用saveDB方法把msg的内容保存到数据库中，此时msg已经就没用了，由于msg的生命周期与对象的生命周期相同，此时msg还不能回收，因此造成了内存泄漏。

实际上这个msg变量可以放在receiveMsg方法内部，当方法使用完，那么msg的生命周期也就结束，此时就可以回收了。还有一种方法，在使用完msg后，把msg设置为null，这样垃圾回收器也会回收msg的内存空间。

4、内部类持有外部类

如果一个外部类的实例对象的方法返回了一个内部类的实例对象，这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持有外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露。

5、改变哈希值

当一个对象被存储进HashSet集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段了，否则，对象修改后的哈希值与最初存储进HashSet集合中时的哈希值就不同了，在这种情况下，即使在contains方法使用该对象的当前引用作为的参数去HashSet集合中检索对象，也将返回找不到对象的结果，这也会导致无法从HashSet集合中单独删除当前对象，造成内存泄露

6、过期引用

内存泄漏的第一个常见来源是存在过期引用。

举个例子-看你能否找出内存泄漏

```
import java.util.Arrays;

public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }
```

```
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}

12345678910111213141516171819202122232425262728 29
```

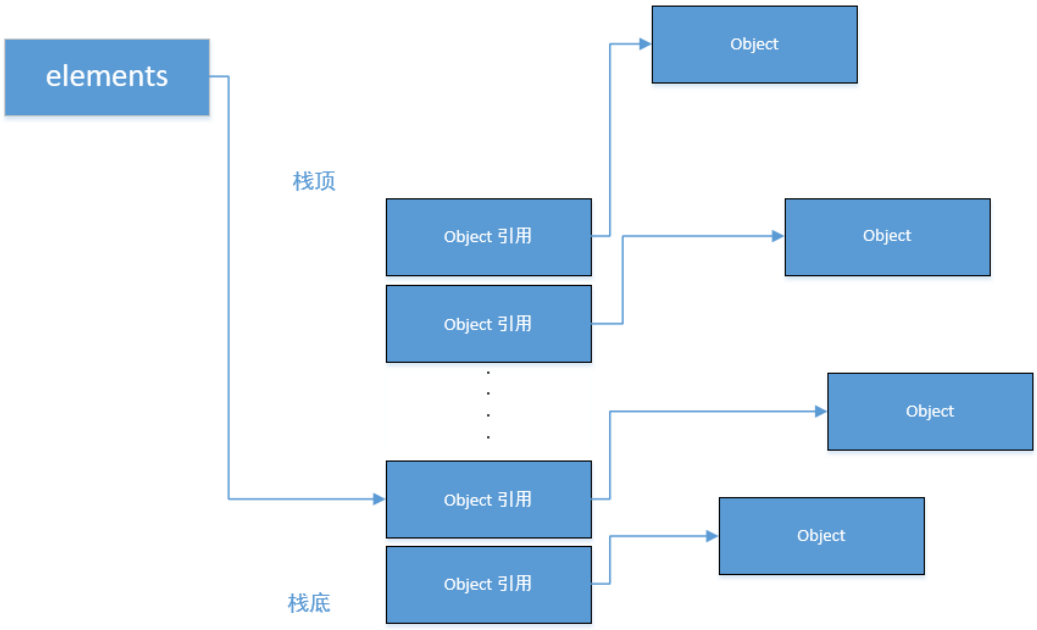
如果一个栈先是增长，然后再收缩，从栈中弹出来的对象不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，它们也不会被回收。因为栈内部维护着对这些对象的过期引用（obsolete reference）。过期引用指永远也不会再被解除的引用。在本例中，在elements数组的“活动部分（active portion）”之外的任何引用都是过期的。活动部分指elements中下标小于size的那些元素。

如果一个对象引用被无意识地保留了，垃圾回收机制不仅不会回收这个对象，而且不会回收被这个对象所引用的所有其他对象。解决方法：一旦对象引用已经过期，只需清空这些引用即可。在本例中，只要一个元素被弹出栈，指向它的引用就过期了。

6.1原因分析

上述程序并没有明显的错误，但是这段程序有一个内存泄漏，随着GC活动的增加，或者内存占用的不断增加，程序性能的降低就会表现出来，严重时可导致内存泄漏，但是这种失败情况相对较少。
代码的主要问题在pop函数，下面通过这张图示展现
假设这个栈一直增长，增长后如下图所示

当进行大量的pop操作时，由于引用未进行置空，gc是不会释放的，如下图所示

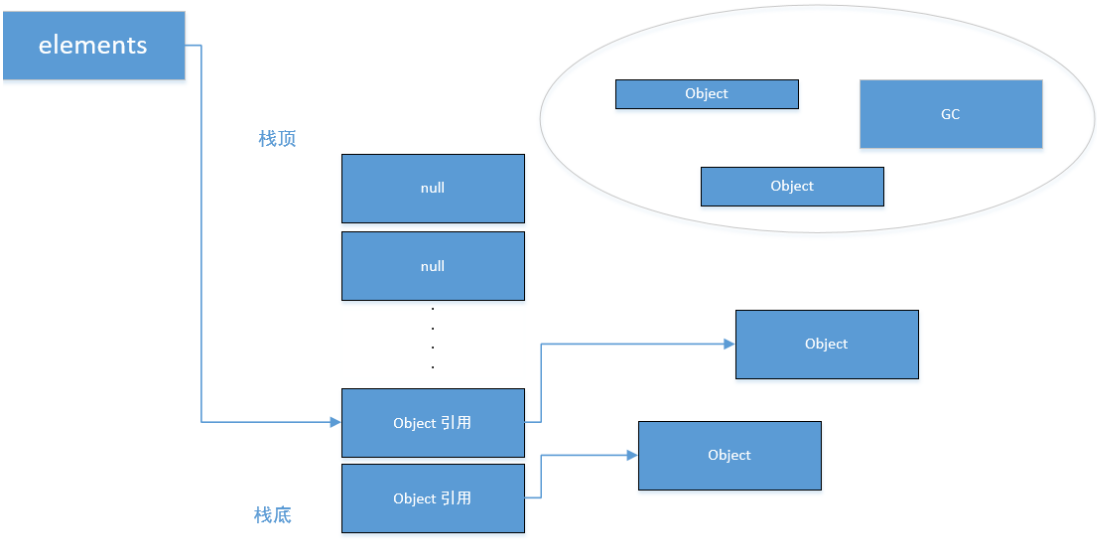


从上图中看以看出，如果栈先增长，在收缩，那么从栈中弹出的对象将不会被当作垃圾回收，即使程序不再使用栈中的这些队象，他们也不会回收，因为栈中仍然保存这对象的引用，俗称过期引用，这个内存泄露很隐蔽。

6.2解决方法

```
public Object pop() {
    if (size == 0)
    throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null;
    return result;
}
1234567
```

一旦引用过期，清空这些引用，将引用置空。



7.缓存泄漏

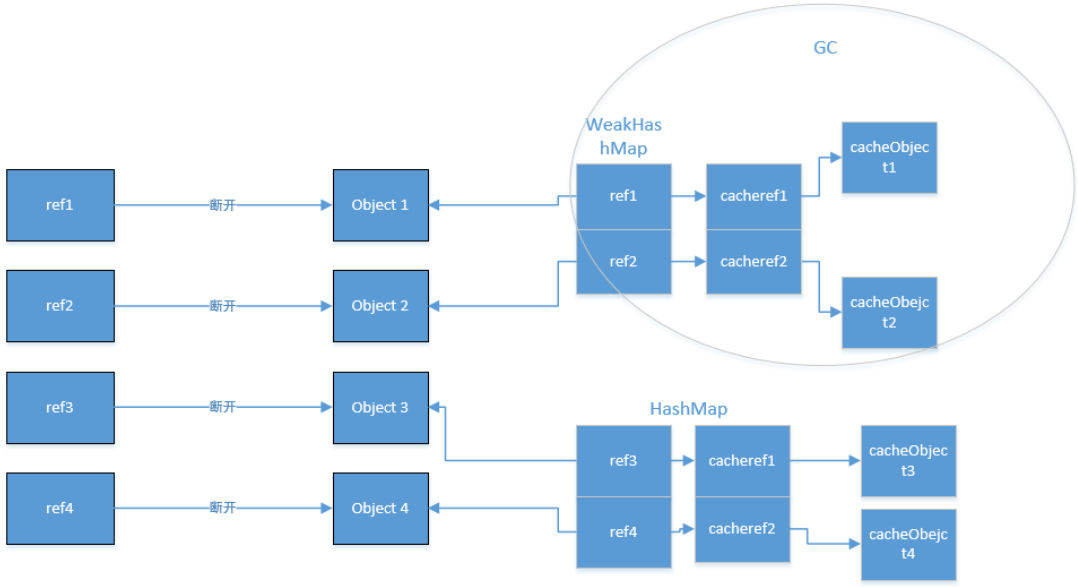
内存泄漏的另一个常见来源是缓存，一旦你把对象引用放入到缓存中，他就很容易遗忘，对于这个问题，可以使用WeakHashMap代表缓存，此种Map的特点是，当除了自身有对key的引用外，此key没有其他引用那么此map会自动丢弃此值

7.1代码示例

```
package com.ratel.test;

/**
 * @业务描述:
 * @package_name:  com.ratel.test
 * @project_name:  ssm
 * @author:  ratelfu@qq.com
 * @create_time:  2019-04-18 20:20
 * @copyright (c) ratelfu 版权所有
 */
import java.util.HashMap;
import java.util.Map;
import java.util.WeakHashMap;
import java.util.concurrent.TimeUnit;

public class MapTest {
    static Map wMap = new WeakHashMap();
    static Map map = new HashMap();
    public static void main(String[] args) {
```

```
        init();
        testWeakHashMap();
        testHashMap();
    }



    public static void init(){
        String ref1= new String("obejct1");
        String ref2 = new String("obejct2");
        String ref3 = new String ("obejct3");
        String ref4 = new String ("obejct4");
        wMap.put(ref1, "chaheObject1");
        wMap.put(ref2, "chaheObject2");
        map.put(ref3, "chaheObject3");
        map.put(ref4, "chaheObject4");
        System.out.println("String引用ref1，ref2，ref3，ref4 消失");

    }
    public static void testWeakHashMap(){

        System.out.println("WeakHashMap GC之前");
        for (Object o : wMap.entrySet()) {
            System.out.println(o);
        }
        try {
            System.gc();
            TimeUnit.SECONDS.sleep(20);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("WeakHashMap GC之后");
        for (Object o : wMap.entrySet()) {
            System.out.println(o);
        }
    }
    public static void testHashMap(){
        System.out.println("HashMap GC之前");
        for (Object o : map.entrySet()) {
            System.out.println(o);
        }
        try {
            System.gc();
            TimeUnit.SECONDS.sleep(20);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("HashMap GC之后");
        for (Object o : map.entrySet()) {
            System.out.println(o);
        }
    }

}
/** 结果
 String引用ref1，ref2，ref3，ref4 消失
```

```
 WeakHashMap GC之前
 obejct2=chaheObject2
 obejct1=chaheObject1
 WeakHashMap GC之后
 HashMap GC之前
 obejct4=chaheObject4
 obejct3=chaheObject3
 Disconnected from the target VM, address: '127.0.0.1:51628', transport:
'socket'
 HashMap GC之后
 obejct4=chaheObject4
 obejct3=chaheObject3
 **/

12345678910111213141516171819202122232425262728293031323334353637383940414243444
54647484950515253545556575859606162636465666768697071727374757677787980818283848
58687888990
```



上面代码和图示主演演示WeakHashMap如何自动释放缓存对象，当init函数执行完成后，局部变量字符串引用weakd1,weakd2,d1,d2都会消失，此时只有静态map中保存中对字符串对象的引用，可以看到，调用gc之后，hashmap的没有被回收，而WeakHashmap里面的缓存被回收了。

8.监听器和回调

内存泄漏第三个常见来源是监听器和其他回调，如果客户端在你实现的API中注册回调，却没有显示的取消，那么就会积聚。需要确保回调立即被当作垃圾回收的最佳方法是只保存他的若引用，例如将他们保存成为WeakHashMap中的键。

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

编译以下代码，执行时jvm参数设置为-Xms20m -Xmx20m

```
1  package com.zhujiukeji.oom;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  //堆溢出
6  public class HeapOomError {
7      public static void main(String[] args) {
8          List<byte[]> list = new ArrayList<>();
9          int i = 0;
10         while (true) {
11             list.add(new byte[5 * 1024 * 1024]);
12             System.out.println("count is: " + (++i));
13         }
14     }
15 }
16
```

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

```
1  package com.zhujiukeji.oom;
2  import java.util.HashMap;
4  //内存泄漏导致的内存溢出
5  public class MemoryLeakOomError {
6      static class Key {
7          Integer id;
8          Key(Integer id) {
9              this.id = id;
10         }
11         @Override
12         public int hashCode() {
13             return id.hashCode();
14         }
15         @Override
16         public boolean equals(Object o) {
17            boolean response = false;
18            if (o instanceof Key) {
19               response = (((Key)o).id).equals(this.id);
20            }
21            return response;
22         }
23     }
24     @SuppressWarnings({ "unchecked", "rawtypes" })
25     public static void main(String[] args) {
26         Map m = new HashMap();
27         while (true) {
28             for (int i = 0; i < 10000; i++) {
29                 if (!m.containsKey(new Key(i))) {
30                     m.put(new Key(i), "Number:" + i);
31                 }
32             }
33         }
34     }
35 }
```

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

对第三方包，一定要慎重选择，不需要的包就去掉。这样既有助于提高编译打包的速度，也有助于提高远程部署的速度。

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

```java
package com.zhujiukeji.oom;

import sun.misc.Unsafe;
import java.lang.reflect.Field;

public class DirectoryMemoryOomError {

    static int ONE_MB = 1024 * 1024;
    static int index = 0;

    @SuppressWarnings("restriction")
    public static void main(String[] args) {
        try {
            Field field = Unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            Unsafe unsafe = (Unsafe) field.get(null);
            while (true) {
                index++;
                unsafe.allocateMemory(ONE_MB);
            }
        } catch (Exception e) {
            System.out.println("index  : " + index);
            e.printStackTrace();
        } catch (Error e) {
            System.out.println("index  : " + index);
            e.printStackTrace();
        }
    }
}

```

3.解决办法

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

1.问题描述

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

```
1  package com.zhujiukeji.oom;
2
3  import java.util.concurrent.Executor;
4  import java.util.concurrent.Executors;
5
6  //模拟无法创建本地线程，抛出异常
7  public class UnableCreateNativeThreadError {
8
9      public static void main(String[] args) {
10         while(true) {
11             Executor pool=Executors.newCachedThreadPool();
12             pool.execute(()->System.out.println("aaaa"));
13         }
14     }
15
16 }
17
```

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

增加系统交换区的大小，我个人认为，如果使用了交换区，性能会大大降低，不建议采用这种方式，生产环境尽量避免最大内存超过系统的物理内存。其次，去掉系统交换区，只使用系统的内存，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

1.问题概述

在描述该问题之前，先熟悉一点操作系统的知识：操作系统是建立在进程的概念之上，这些进程在内核中作业，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。

内存溢出或泄露原因分析

分析堆内存溢出的原因可能如下：

使用了大量的递归或无限递归（递归中用到了大量的新建的对象）

使用了大量循环或死循环（循环中用到了大量的新建的对象）

类中和引用变量过多使用了Static修饰 如 public staitc Student s；在类中的属性中使用 static 修饰的最好只用基本类型或字符串。如public static int i = 0; //public static String str;

数组，List，Map中存放的是对象的引用而不是对象，因为这些引用会让对应的对象不能被释放，会大量存储在内存中。

分析栈内存溢出的原因可能如下：

使用了大量的递归或无限递归

使用了大量循环或死循环（如循环中不停调用方法）

list，map，数组等长度过大等。

出现内存溢出或内存泄露的解决方案

1.修改JVM启动参数(-Xms，-Xmx)，直接增加虚拟机内存。

2.检查错误日志。

3.使用内存查看工具查看内存使用情况(如jconsole)

4.对代码进行仔细分析，找出可能发生内存溢出的位置。

详细排查方案如下:

检查在数据库中取的数据量是否超过内存

检查是否有过大的集合或对象

检查是死循环或递归是否会导致溢出

检查是否有大量对象的创建是否会出现内存问题

检查是否有大量的连接对象或监听器等未关闭

……

在开发中应如何避免出现内存泄露

1.尽量少使用枚举

2.尽量使用静态内部类而不是内部类

3.尽量使用轻量级的数据结构

4.养成关闭连接和注销监听器的习惯

5.谨慎使用static关键字

6.谨慎使用单例模式

……

硬核推荐：尼恩Java硬核架构班

又名疯狂创客圈社群 VIP

详情：

https://www.cnblogs.com/crazymakercircle/p/9904544.html

架构班（社群 VIP）的起源：

最初的视频，主要是给读者加餐。很多的读者，需要一些高质量的实操、理论视频，所以，我就围绕书，和底层，做了几个实操、理论视频，然后效果还不错，后面就做成迭代模式了。

架构班（社群 VIP）的功能：

提供高质量实操项目整刀真枪的架构指导、快速提升大家的:

- 开发水平
- 设计水平
- 架构水平

弥补业务中 CRUD 开发短板，帮助大家尽早脱离具备 3 高能力，掌握：

- 高性能
- 高并发
- 高可用

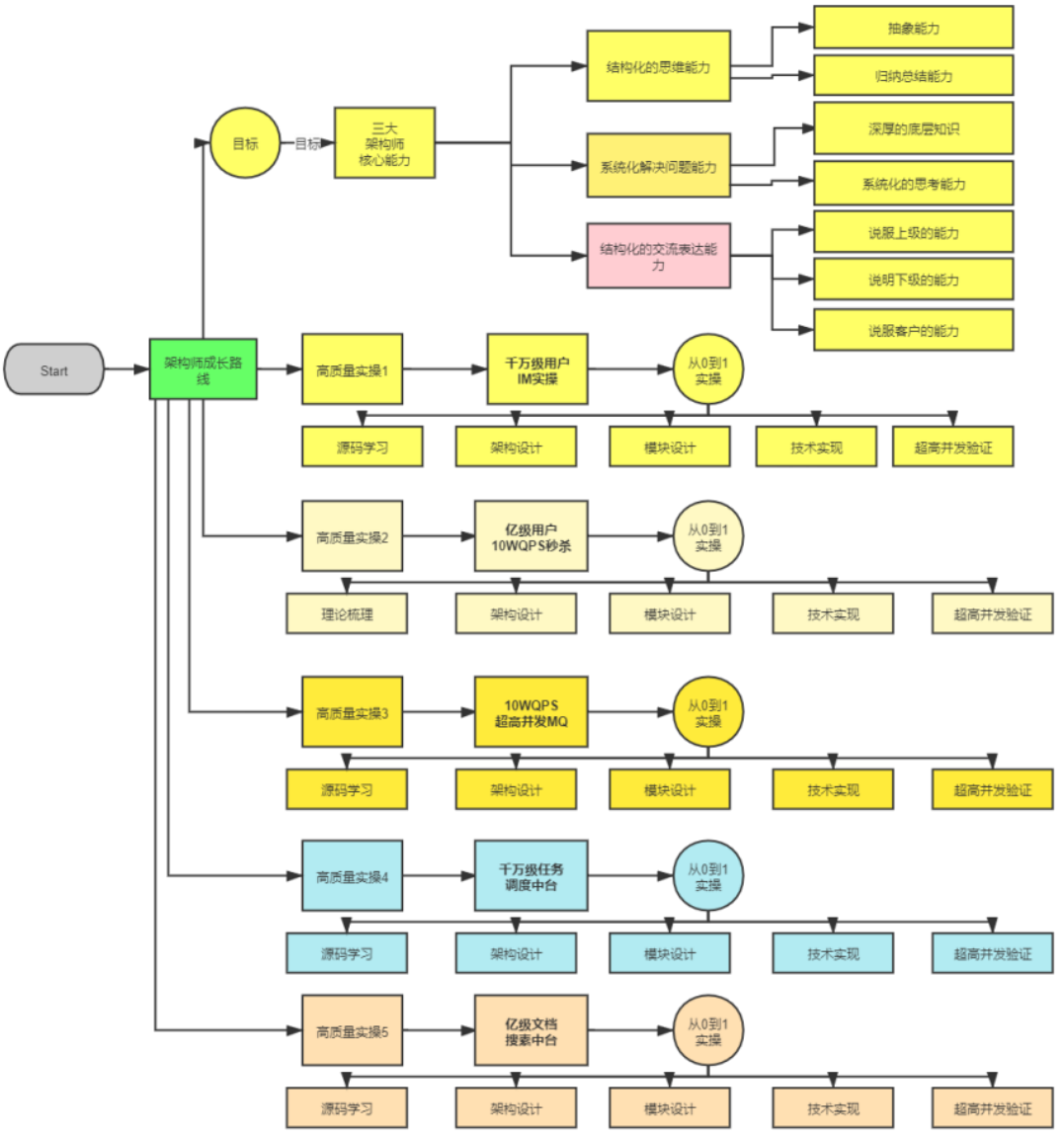
作为一个高质量的架构师成长、人脉社群，把所有的卷王聚焦起来，一起卷：

- 卷高并发实操
- 卷底层原理
- 卷架构理论、架构哲学
- 最终成为顶级架构师，实现人生理想，走向人生巅峰

架构班（社群 VIP）的目的：

- 高质量的实操，大大提升简历的含金量，吸引力，增强面试的召唤率
- 为大家提供九阳真经、葵花宝典，快速提升水平
- 进大厂、拿高薪
- 一路陪伴，提供助学视频和指导，辅导大家成为架构师
- 自学为主，和其他卷王一起，卷高并发实操，卷底层原理、卷大厂面试题，争取狠卷 3 月成高手，狠卷 3 年成为顶级架构师

N 个超高并发实操项目：简历压轴、个顶个精彩



【样章】第 17 章:横扫全网 Rocketmq 视频第 2 部曲: 工业级 rocketmq 高可用(HA)底层原理和实操

工业级 rocketmq 高可用底层原理，包含：消息消费、同步消息、异步消息、单向消息等不同消息的底层原理和源码实现；消息队列非常底层的主从复制、高可用、同步刷盘、异步刷盘等底层原理。

工业级 rocketmq 高可用底层原理和搭建实操，包含：高可用集群的搭建。

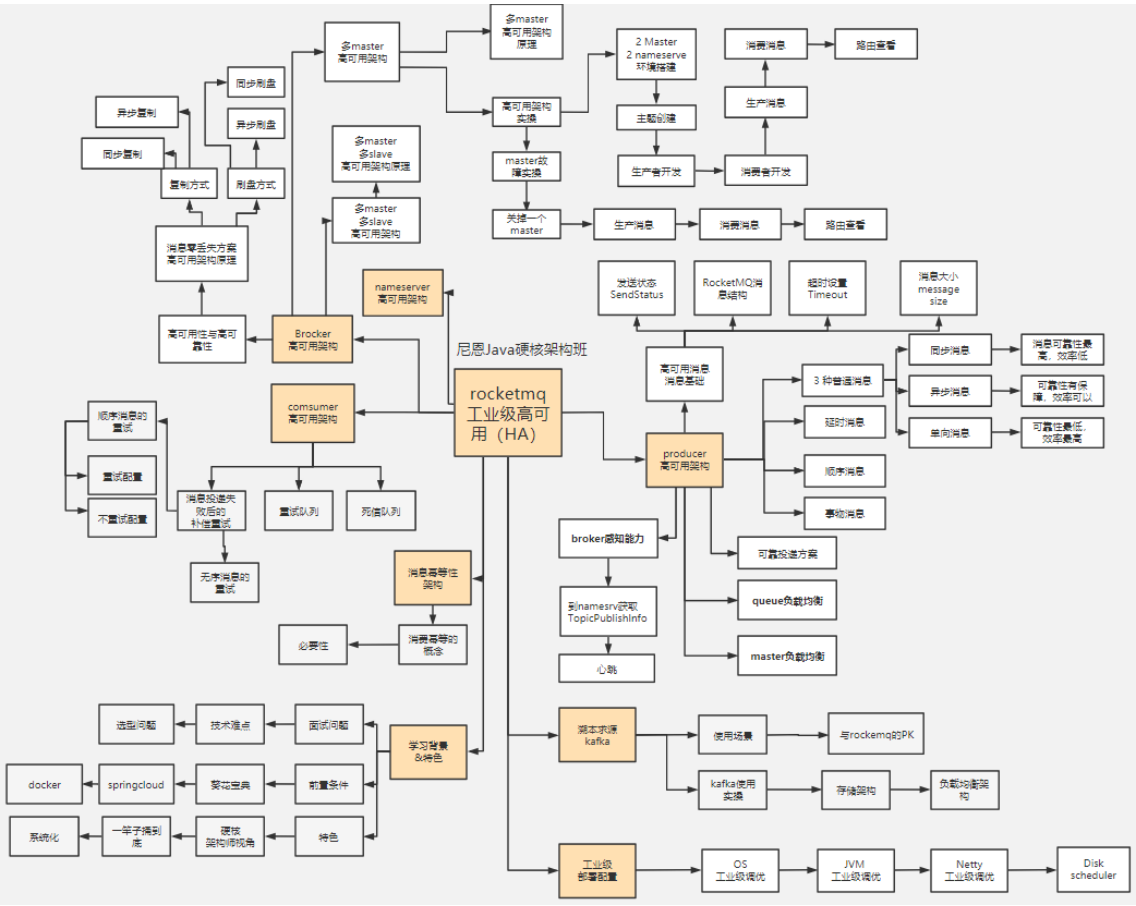
解决以下难题：

1、技术难题：RocketMQ 如何最大限度的保证消息不丢失的呢？RocketMQ 消息如何做到高可靠投递？

2、技术难题：基于消息的分布式事务，核心原理不理解

3、选型难题： kafka or rocketmq ，该娶谁？

下图链接：https://www.processon.com/view/6178e8ae0e3e7416bde9da19



成功案例：2 年翻 3 倍，35 岁卷王成功转型为架构师

详情：http://topcoder.cloud/forum.php?mod=forumdisplay&fid=43&page=1

最新 最后发表 热门 精华

- 成功案例：[1057号卷王] 3年小伙拿到外企offer，薪酬涨了200% — 卷王1号 超级版主 前天 17:41
- 成功案例：[645号卷王] 4年经验卷王逆袭，被毕业后，反涨24W — 卷王1号 超级版主 2022-9-21
- 成功案例：[878号卷王] 小伙8年经验，年薪60W — 卷王1号 超级版主 2022-8-13
- 年薪70W案例：通过尼恩的指导，小伙伴年薪从40W涨到70W — 卷王1号 超级版主 2022-2-11
- 成功案例：[493号卷王] 5年小伙拿满意offer，就业寒冬季逆涨30% — 卷王1号 超级版主 前天 17:43
- 成功案例：[250号卷王] 就业极寒时代，收offer 涨25% — 卷王1号 超级版主 前天 17:38
- 成功案例：[612号卷王] 就业极寒时代，从外包到自研 — 卷王1号 超级版主 前天 17:15
- 成功案例：[913号卷王] 热烈祝贺6年经验卷王，年薪40W — 卷王1号 超级版主 2022-9-21
- 成功案例：[959号卷王] 4年经验卷王，喜获百度、Boss直聘等N个优质offer，最高涨100% — 卷王1号 超级版主 2022-9-21
- 成功案例：[529号卷王] 5年经验卷王喜收2大offer，最高涨5K — 卷王1号 超级版主 2022-9-21
- 成功案例：[811号卷王] 热烈祝贺7年经验卷王，薪酬涨30% — 卷王1号 超级版主 2022-9-21
- 成功案例：[287号卷王] 不惧大寒潮，卷王逆市收4 offer，涨30%，可喜可贺 — 卷王1号 超级版主 2022-5-30
- 成功案例：[1002号卷王] 5月份"被毕业"，改简历后，斩获顶级央企Offer，涨薪7000+ — 卷王1号 超级版主 2022-7-5
- 成功案例：[7号卷王] 热烈祝贺小伙伴涨薪120% — 卷王1号 超级版主 2022-8-13
- 成功案例：[134号卷王] 大三小伙卷1年，斩获顶级央企Offer，成功逆袭 — 卷王1号 超级版主 2022-7-6
- 成功案例：[1008号卷王] 5年经验卷王收42W offer，月涨8000，可喜可贺 — 卷王1号 超级版主 2022-5-30
- 成功案例：[453号卷王] 非全日制 6年卷王喜提3 offer，年薪30W，可喜可贺 — 卷王1号 超级版主 2022-5-21
- 成功案例：[924号卷王] 6年卷王喜提4 offer，最高涨薪9000，可喜可贺 — 卷王1号 超级版主 2022-5-21
- 成功案例：[15号卷王] 4年卷王入职 微软，涨薪50%，可喜可贺 — 卷王1号 超级版主 2022-5-12
- 成功案例：[527号卷王] 4年卷王喜提2 offer，涨薪50%，可喜可贺 — 卷王1号 超级版主 2022-5-13
- 成功案例：[788号卷王] 3年卷王喜提优质Offer，涨薪60% — 卷王1号 超级版主 2022-5-11
- 成功案例：热烈祝贺：非全日制卷王，喜提2个心仪offer，面3家过2家 — 卷王1号 超级版主 2022-4-21

最新 最后发表 热门 精华

- 成功案例：[693号卷王] 二线城市6年卷王喜提4大优质Offer，含央企offer，最高薪酬35W — 卷王1号 超级版主 2022-4-16
- 成功案例：[85号卷王] 双非2本小伙，春招大捷，喜提9个offer，最高薪酬近30万 — 卷王1号 超级版主 2022-4-14
- 成功案例：[741号卷王] 卷王逆袭！6年小伙从很少面试机会到搞定35K*14薪Offer — 卷王1号 超级版主 2022-4-12
- 成功案例：[642号卷王] 热烈祝贺，6年卷王喜提优质国企offer — 卷王1号 超级版主 2022-4-7
- 成功案例：[796号卷王] 热烈祝贺，36岁卷王喜提52万优质offer — 卷王1号 超级版主 2022-3-25
- 成功案例：[15号卷王] 小伙卷1年，涨薪9K+，喜收ebay等多个优质offer — 卷王1号 超级版主 2022-3-24
- 成功案例：[821号卷王] 小伙狠卷3个月，喜提10多个offer — 卷王1号 超级版主 2022-3-21
- 成功案例：[736号卷王] 3年半经验收22k offer，但是小伙志存高远，冲击25k+ — 卷王1号 超级版主 2022-3-20
- 成功案例：热烈祝贺1群小卷王offer拿到手软，甚至拒了阿里offer — 卷王1号 超级版主 2022-3-16
- 简历案例：简历一改，腾讯的邀请就来了！热烈祝贺，小伙收到一大堆面试邀请 — 卷王1号 超级版主 2022-3-10
- 成功案例：祝贺我圈两大超级卷王，一个过了阿里HR面，一个过了阿里2面 — 卷王1号 超级版主 2022-3-10
- 成功案例：小伙伴php转Java，卷1.5年Java，涨薪50%，喜收多个优质offer — 卷王1号 超级版主 2022-3-10
- 成功案例：4年小伙狠卷半年，拿到 移动、京东 两大顶级offer — 尼恩 超级版主 2022-3-5
- 成功案例：[267号卷王] 助力3年经验卷王，拿到蜂巢的17k x 14薪的offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[143号卷王] 二本院校00后卷神，毕业没到一年跳到宇节，年薪45W — 卷王1号 超级版主 2022-2-27
- 成功案例：[494号卷王] 尼恩分布式事务助力卷王拿到 中信银行offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[76号卷王] 2线城市卷王，狠卷1.5年，喜收22K offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[429号卷王] 小伙伴在社群卷5个月，涨8k+ — 卷王1号 超级版主 2022-2-27
- 成功案例：[154号卷王] 双非学校毕业卷王，连拿 京东到家&滴滴 两个大厂Offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[232号卷王] 涨薪10K，继续卷向食物链顶端 — 卷王1号 超级版主 2022-2-27
- 成功案例：狠卷1年技术，喜收 腾讯、阿里、微软三大Offer，最高年薪56W — 卷王1号 超级版主 2022-2-27
- 成功案例：[449号卷王] 应届毕业卷王喜收 滴滴offer，年薪33W — 卷王1号 超级版主 2022-2-27

- 成功案例：[732号卷王] 尼恩助力3年经验卷王收获 京东offer，年薪35W — 卷王1号 超级版主 2022-2-27
- 成功案例：[558号卷王] 2年经验卷王，喜收 网易和阿里子公司两个优质offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[569号卷王] 双非应届生卷王，喜收字节跳动实习offer — 卷王1号 超级版主 2022-2-25
- 成功案例：[420号卷王] 狠卷1年，卷王涨薪80%，涨薪12000元！ — 卷王1号 超级版主 2022-2-25
- 成功案例：[76号卷王] 通过尼恩1年半的指导，专科学历小伙伴从0.8K涨到22K — 卷王1号 超级版主 2022-2-10

- 成功案例：[551号卷王] 小伙伴学完后，成功进入大厂，并且推荐自己的朋友加VIP学习 — 卷王1号 超级版主 2022-2-10
- 成功案例：[214号卷王] 助力2年经验卷王，成功拿到17K月薪 — 卷王1号 超级版主 2022-2-10
- 成功案例：[92号卷王] 课程实操助力社群小伙伴喜收 喜马拉雅Offer — 卷王1号 超级版主 2022-2-10
- 成功案例：社群卷王小伙伴成功过了滴滴三面 获滴滴Offer — 卷王1号 超级版主 2022-2-10
- [612号卷王]滴滴小伙伴，蹲点考察半年，觉得靠谱后加入 疯狂创客圈 — 卷王1号 超级版主 2022-2-10

简历优化后的成功涨薪案例（VIP 含免费简历优化）

修改简历找尼恩（资深简历优化专家）

- 如果面试表达不好，尼恩会提供 简历优化指导
- 如果项目没有亮点，尼恩会提供 项目亮点指导
- 如果面试表达不好，尼恩会提供 面试表达指导

作为 40 岁老架构师，尼恩长期承担技术面试官的角色：

- **从业以来，“阅历”无数，对简历有着点石成金、改头换面、脱胎换骨**的指导能力。
- **尼恩指导过刚刚就业的小白，也指导过 P8 级的老专家，都指导他们上岸。**

如何联系尼恩。尼恩微信，请参考下面的地址：

语雀：https://www.yuque.com/crazymakercircle/gkkw8s/khigna
码云：https://gitee.com/crazymaker/SimpleCrayIM/blob/master/疯狂创客圈总目录.md