目录

[**一、** **多线程** 2](#_Toc471153624)

[**二、** **集合** 8](#_Toc471153625)

[**三、** **Spring** 11](#_Toc471153626)

[**四、** **数据库** 12](#_Toc471153627)

[**五、** **JVM** 12](#_Toc471153628)

[**六、** **NIO** 17](#_Toc471153629)

[**七、** **Redis** 17](#_Toc471153630)

[**八、** **MQ** 17](#_Toc471153631)

[**九、** **其他** 17](#_Toc471153632)

1. **多线程**

#### 1.1 实现方式

##### 1.1.1 Thread

继承Thread类的方法尽管被我列为一种多线程实现方式，但Thread本质上也是实现了Runnable接口的一个实例，它代表一个线程的实例，并且，启动线程的唯一方法就是通过Thread类的start()实例方法。start()方法是一个native方法，它将启动一个新线程，并执行run()方法。这种方式实现多线程很简单，通过自己的类直接extend Thread，并复写run()方法，就可以启动新线程并执行自己定义的run()方法。例如：

public class MyThread extends Thread {

　　public void run() {

　　 System.out.println("MyThread.run()");

　　}

}

在合适的地方启动线程如下：

MyThread myThread1 = new MyThread();

MyThread myThread2 = new MyThread();

myThread1.start();

myThread2.start();

##### 1.1.2 Runnable

如果自己的类已经extends另一个类，就无法直接extends Thread，此时，必须实现一个Runnable接口，如下：

public class MyThread extends OtherClass implements Runnable {

　　public void run() {

　　 System.out.println("MyThread.run()");

　　}

}

为了启动MyThread，需要首先实例化一个Thread，并传入自己的MyThread实例：

MyThread myThread = new MyThread();

Thread thread = new Thread(myThread);

thread.start();

事实上，当传入一个Runnable target参数给Thread后，Thread的run()方法就会调用target.run()，参考JDK源代码：

public void run() {

　　if (target != null) {

　　 target.run();

　　}

}

##### 1.1.3 Future

ExecutorService、Callable、Future这个对象实际上都是属于Executor框架中的功能类。想要详细了解Executor框架的可以访问http://www.javaeye.com/topic/366591 ，这里面对该框架做了很详细的解释。返回结果的线程是在JDK1.5中引入的新特征，确实很实用，有了这种特征我就不需要再为了得到返回值而大费周折了，而且即便实现了也可能漏洞百出。

可返回值的任务必须实现Callable接口，类似的，无返回值的任务必须Runnable接口。执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了，再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。下面提供了一个完整的有返回结果的多线程测试例子，在JDK1.5下验证过没问题可以直接使用。代码如下：

import java.util.concurrent.\*;

import java.util.Date;

import java.util.List;

import java.util.ArrayList;

/\*\*

\* 有返回值的线程

\*/

@SuppressWarnings("unchecked")

public class Test {

public static void main(String[] args) throws ExecutionException,

InterruptedException {

System.out.println("----程序开始运行----");

Date date1 = new Date();

int taskSize = 5;

// 创建一个线程池

ExecutorService pool = Executors.newFixedThreadPool(taskSize);

// 创建多个有返回值的任务

List<Future> list = new ArrayList<Future>();

for (int i = 0; i < taskSize; i++) {

Callable c = new MyCallable(i + " ");

// 执行任务并获取Future对象

Future f = pool.submit(c);

// System.out.println(">>>" + f.get().toString());

list.add(f);

}

// 关闭线程池

pool.shutdown();

// 获取所有并发任务的运行结果

for (Future f : list) {

// 从Future对象上获取任务的返回值，并输出到控制台

System.out.println(">>>" + f.get().toString());

}

Date date2 = new Date();

System.out.println("----程序结束运行----，程序运行时间【"

+ (date2.getTime() - date1.getTime()) + "毫秒】");

}

}

class MyCallable implements Callable<Object> {

private String taskNum;

MyCallable(String taskNum) {

this.taskNum = taskNum;

}

public Object call() throws Exception {

System.out.println(">>>" + taskNum + "任务启动");

Date dateTmp1 = new Date();

Thread.sleep(1000);

Date dateTmp2 = new Date();

long time = dateTmp2.getTime() - dateTmp1.getTime();

System.out.println(">>>" + taskNum + "任务终止");

return taskNum + "任务返回运行结果,当前任务时间【" + time + "毫秒】";

}

}

代码说明：

上述代码中Executors类，提供了一系列工厂方法用于创先线程池，返回的线程池都实现了ExecutorService接口。

public static ExecutorService newFixedThreadPool(int nThreads)

创建固定数目线程的线程池。

public static ExecutorService newCachedThreadPool()

创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

public static ExecutorService newSingleThreadExecutor()

创建一个单线程化的Executor。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

ExecutoreService提供了submit()方法，传递一个Callable，或Runnable，返回Future。如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。

#### 1.2 线程池

##### 1.2.1 内部原理

使用部分：

http://www.cnblogs.com/easycloud/p/3726089.html

缺陷：

http://www.ibm.com/developerworks/cn/java/j-jtp0730/

好处：

1.减少在创建和销毁线程上所花的时间以及系统资源的开销

2.如不使用线程池，有可能造成系统创建大量线程而导致消耗完系统内存以及”过度切换”。

##### 1.2.2 常见类型

1、newFixedThreadPool创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

2、newCachedThreadPool创建一个可缓存的线程池。这种类型的线程池特点是：

1).工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

2).如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

3、newSingleThreadExecutor创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，如果这个线程异常结束，会有另一个取代它，保证顺序执行(我觉得这点是它的特色)。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的 。

4、newScheduleThreadPool创建一个定长的线程池，而且支持定时的以及周期性的任务执行，类似于Timer。

1. **集合**

#### 2.1 HashMap

##### 2.1.1 Hash算法

典型的哈希算法包括 MD2、MD4、MD5 和 SHA-1。它是一种单向密码体制,即它是一个从明文到密文的不可逆的映射,只有加密过程,没有解密过程。同时,哈希函数可以将任意长度的输入经过变化以后得到固定长度的输出。哈希函数的这种单向特征和输出数据长度固定的特征使得它可以生成消息或者数据。

##### 2.1.2 内部原理

http://www.cnblogs.com/chenssy/p/3521565.html

初始容量，加载因子。这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中桶的数量，初始容量是创建哈希表时的容量，加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，它衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。系统默认负载因子为0.75，一般情况下我们是无需修改的。

当我们想一个HashMap中添加一对key-value时，系统首先会计算key的hash值，然后根据hash值确认在table中存储的位置。若该位置没有元素，则直接插入。否则迭代该处元素链表并依此比较其key的hash值。如果两个hash值相等且key值相等(e.hash == hash && ((k = e.key) == key || key.equals(k))),则用新的Entry的value覆盖原来节点的value。如果两个hash值相等但key值不等 ，则将该节点插入该链表的链头。

面试部分：

http://www.admin10000.com/document/3322.html

##### 2.1.3 线程不安全的原因

addEntry（操作），resize（扩容：双倍）

#### 2.2 ConcurrentHashMap

##### 2.2.1 内部原理

查找segment中的位置，分段实现线程安全，HashEntry内部为final ，所以删除时，复制删除节点前所有元素在和后面的重新组成新segment；添加时在表头添加。

#### 2.3 ArrayList

##### 2.3.1 内部原理

(01) ArrayList 实际上是通过一个数组去保存数据的。当我们构造ArrayList时；若使用默认构造函数，则ArrayList的默认容量大小是10。

(02) 当ArrayList容量不足以容纳全部元素时，ArrayList会重新设置容量：新的容量=“(原始容量x3)/2 + 1”。

(03) ArrayList的克隆函数，即是将全部元素克隆到一个数组中。

(04) ArrayList实现java.io.Serializable的方式。当写入到输出流时，先写入“容量”，再依次写入“每一个元素”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”。

#### 2.4 BlockingQueue

##### 2.4.1 内部原理

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

阻塞队列提供了四种处理方法:

1.抛出异常：是指当阻塞队列满时候，再往队列里插入元素，会抛出IllegalStateException("Queue full")异常。当队列为空时，从队列里获取元素时会抛出NoSuchElementException异常 。

2.返回特殊值：插入方法会返回是否成功，成功则返回true。移除方法，则是从队列里拿出一个元素，如果没有则返回null

3.一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到拿到数据，或者响应中断退出。当队列空时，消费者线程试图从队列里take元素，队列也会阻塞消费者线程，直到队列可用。

4.超时退出：当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定的时间，生产者线程就会退出。

1. **Spring**

#### 3.1 注入方式

##### 3.1.1 构造器注入

##### 3.1.2 属性注入

##### 3.1.3 静态工厂注入

#### 3.2 Bean生命周期

#### 3.3 自定义标签

##### 3.3.1 如何解析

#### 3.4 Aop

##### 3.4.1 配置

#### 3.5 事务

##### 3.5.1 配置

1. **数据库**

#### 4.1 MySQL

##### 4.1.1 常见类型

##### 4.1.2 优化方式

1. **JVM**

#### 5.1 Java 内存模型

#### 5.2 OOM的类型及原因

##### 5.2.1 Heap

**java.lang.OutOfMemoryError: Java heap space**

**分析:**

此OOM是由于JVM中heap的最大值不满足需要，将设置heap的最大值调高即可，参数样例为：-Xmx2G

JVM堆的设置是指Java程序运行过程中JVM可以调配使用的内存空间的设置.JVM在启动的时候会自动设置Heap size的值，其初始空间(即-Xms)是物理内存的1/64，最大空间(-Xmx)是物理内存的1/4。可以利用JVM提供的-Xmn -Xms -Xmx等选项可进行设置。Heap size 的大小是Young Generation 和Tenured Generaion 之和。

提示：在JVM中如果98％的时间是用于GC且可用的Heap size 不足2％的时候将抛出此异常信息。

提示：Heap Size 最大不要超过可用物理内存的80％，一般的要将-Xms和-Xmx选项设置为相同，而-Xmn为1/4的-Xmx值。

**解决参考:**

调高heap的最大值，即-Xmx的值调大。

##### 5.2.2 StackOverflowError

**Exception in thread "main" java.lang.StackOverflowError**

**分析:**

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OutOfMemoryError异常。

一般在单线程程序情况下无法产生OutOfMemoryError异常，使用多线程方式也会出现OutOfMemeoryError，因为栈是线程私有的，线程多也会方法区溢出。

**解决参考:**

检查程序是否有深度递归。

##### 5.2.3 Perm

**java.lang.OutOfMemoryError: PermGen space**

**分析**:

PermGen space的全称是Permanent Generation space,是指内存的永久保存区域,这块内存主要是被JVM存放Class和Meta信息的,Class在被Loader时就会被放到PermGen space中,它和存放类实例(Instance)的Heap区域不同,GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的应用中有很多CLASS的话,就很可能出现PermGen space错误,这种错误常见在web服务器对JSP进行pre compile的时候。如果你的WEB APP下都用了大量的第三方jar, 其大小超过了jvm默认的大小那么就会产生此错误信息了。由于JVM在默认的情况下，Perm默认为64M，而很多程序需要大量的Perm区内 存，尤其使用到像spring等框架的时候，由于需要使用到动态生成类，而这些类不能被GC自动释放，所以导致OutOfMemoryError: PermGen space异常。解决方法很简单，增大JVM的 -XX:MaxPermSize 启动参数，就可以解决这个问题，如过使用的是默认变量通常是64M，改成128M就可以了，-XX:MaxPermSize=128m。如果已经是128m，就改成 256m。我一般在服务器上为安全起见，改成256m。

**解决参考:**

调高Perm的最大值，即-XX:MaxPermSize的值调大。

另外，注意一点，Perm一般是在JVM启动时加载类进来，如果是JVM运行较长一段时间而不是刚启动后溢出的话，很有可能是由于运行时有类被动态加载进来，此时建议用CMS策略中的类卸载配置。

如：-XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled

##### 5.2.4 GC

**java.lang.OutOfMemoryError: GC overhead limit exceeded**

**分析**:

此OOM是由于JVM在GC时，对象过多，导致内存溢出，建议调整GC的策略，在一定比例下开始GC而不要使用默认的策略，或者将新代和老代设置合适的大小，需要进行微调存活率。

**解决参考**:

改变GC策略，在老代80%时就是开始GC，并且将-XX:SurvivorRatio（-XX:SurvivorRatio=8）和-XX:NewRatio（-XX:NewRatio=4）设置的更合理。

##### 5.2.5 Native Thread Created

**java.lang.OutOfMemoryError: unable to create new native thread**

**分析**:

这个异常问题本质原因是我们创建了太多的线程，而能创建的线程数是有限制的，导致了异常的发生。

**解决参考:**

如果JVM内存调的过大或者可利用率小于20%，可以建议将heap及perm的最大值下调，并将线程栈调小，即-Xss调小，如：-Xss128k。

在JVM内存不能调小的前提下，将-Xss设置较小，如：-Xss:128k。

##### 5.2.6 Allocate Huge Array

**Exception in thread "main": java.lang.OutOfMemoryError: Requested array size exceeds VM limit**

**分析**:

此类信息表明应用程序试图分配一个大于堆大小的数组。例如，如果应用程序new一个数组对象，大小为512M，但是最大堆大小为256M，因此OutOfMemoryError会抛出，因为数组的大小超过虚拟机的限制。

**解决参考:**

（1）、首先检查heap的-Xmx是不是设置的过小

（2）、如果heap的-Xmx已经足够大，那么请检查应用程序是不是存在bug，例如：应用程序可能在计算数组的大小时，存在算法错误，导致数组的size很大，从而导致巨大的数组被分配。

##### 5.2.7 Small Swap

Exception in thread "main": java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space?

**分析**:

抛出这类错误，是由于从native堆中分配内存失败，并且堆内存可能接近耗尽。这类错误可能跟应用程序没有关系，例如下面两种原因也会导致错误的发生：

（1）操作系统配置了较小的交换区

（2）系统的另外一个进程正在消耗所有的内存

**解决参考:**

（1）、检查os的swap是不是没有设置或者设置的过小

（2）、检查是否有其他进程在消耗大量的内存，从而导致当前的JVM内存不够分配。

注意：虽然有时<reason>部分显示导致OOM的原因，但大多数情况下，<reason>显示的是提示分配失败的源模块的名称，所以有必要查看日志文件，如crash时的hs文件。

##### 5.2.8 Exhausted Native Memory

**java.lang.OutOfMemoryErr java.io.FileInputStream.readBytes(Native Method)**

**分析**:

从错误日志来看，在OOM后面没有提示引起OOM的原因，进一步查看stack trace发现，导致OOM的原因是由Native Method的调用引起的，另外检查[Java](http://lib.csdn.net/base/java)heap，发现heap的使用正常，因而需要考虑问题的发生是由于Native memory被耗尽导致的。

**解决参考:**

从根本上来说，解决此问题的方法应该通过检测发生问题时的环境下，native memory为什么被占用或者说为什么native memory越来越小，从而去解决引起Native memory减小的问题。但是如果此问题不容易分析时，可以通过以下方法或者结合起来处理。  
（1）、cpu和os保证是64位的，并且jdk也换为64位的。  
（2）、将java heap的-Xmx尽量调小，但是保证在不影响应用使用的前提下。  
（3）、限制对native memory的消耗，比如：将thread的-Xss调小，并且限制产生大量的线程；限制文件的io操作次数和数量；限制网络的使用等等。

#### 5.3 GC算法

#### 5.4 优化参数

1. **NIO**

#### 6.1 基本概念

1. **Redis**

#### 7.2 分布式锁实现

1. **MQ**

#### 7.3 概念及作用

实现系统解耦

1. **其他**