久

杨

珍

精

悟尽归原 几近于道。

Create By Yangcl

2018-03-15

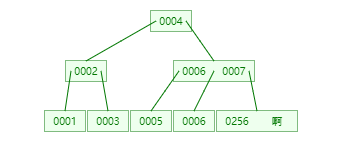
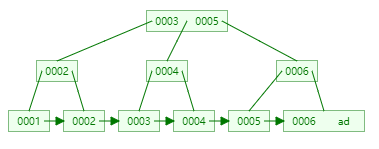
**Mysql高级特性**

**1 MySQL基础理论**

**1.1 B-Tree和B+Tree**

Mysql的InnoDB和MyISAM引擎都利用了B+tree作为索引数据结构，也就是说：表数据**文件**本身就是按B+Tree组织的一个索引结构；不同点在于InnoDB是聚集索引，MyISAM是非聚集索引。这个【表数据文件】叫做“页”。Innodb一页的大小=16Kb。这是默认的，可以人为自己设置。

B-Tree的数据结构如下： B+Tree的数据结构如下：

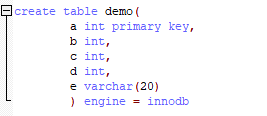
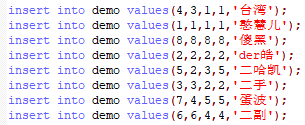
B+Tree的每一个节点就是InnoDB中的一个“页”。

他们两者最大的不同在于B+Tree的叶子节点带有指针，网上有不少博客画的图是错的。上面两幅图生成自：

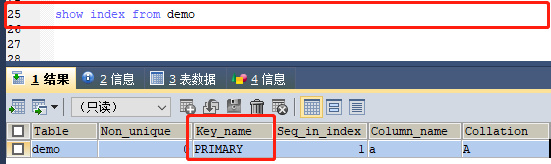
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html。MySQL对B+Tree还有改进，使用的是双向指针。

1.2 InnoDB中的“页”

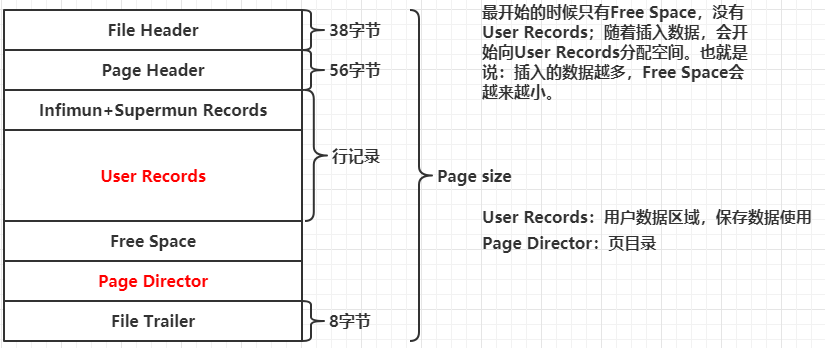
页（Page）是 Innodb 存储引擎用于管理数据的最小磁盘单位。常见的页类型有数据页、目录页、Undo页、系统页、事务数据页等，本文主要分析的是数据页和目录页。默认的页大小为16KB，每个页中至少存储有2条或以上的行记录。在MySQL中可通过如下命令查看页的大小：mysql> show variables like 'innodb\_page\_size'; 基于mysql5.7版本，创建一张表插入8条数据，用来做后续的说明。

 注意：上面插入数据时是无序的。

 主键索引。

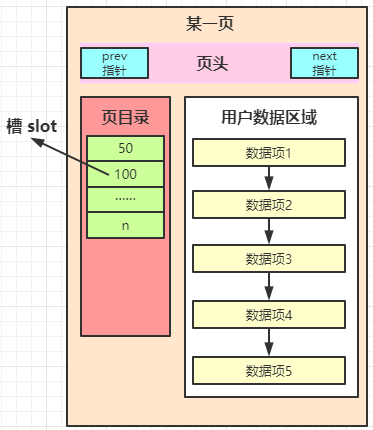
1.2.1页的结构



参考：https://dev.mysql.com/doc/internals/en/innodb-page-overview.html。

我们结合页目录和用户数据区这两个重要的属性内容来绘制一个【**数据页**的对象结构图】，用于后面阐述数据添加过程和索引结构。

注意！这里不是目录页！！



1.2.2 InnoDB数据插入过程模拟



根据自增ID排序插入，【用户数据区域】保存的内容呈现为一个有序链表结构。这也是为什么我们在插入数据的时候是无序的，但是在默认查询的时候却是根据主键ID排好序的。

1.3 存在的3个问题

通过1.2.2的模拟过程，产生了三个问题：1、没有自增ID怎么办？2、排序链表插入操作效率低3、链表效率问题如何解决？

1.3.1排序链表插入操作效率低

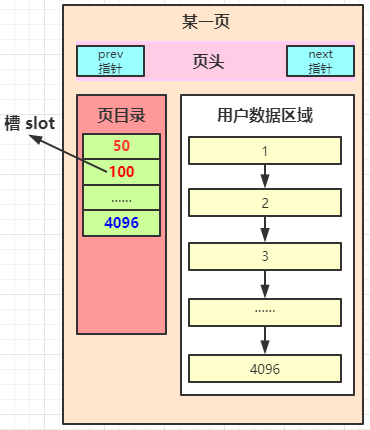
【用户数据区域】使用的是链表结构来保存新添加的数据，链表对于添加和删除操作效率是很高的；但是排序后插入这一点比较耗费性能。实际上这是一种“相互妥协”后的选择。因为作为数据库不仅要保存数据，更多的情况下是查询数据。排序后的记录，根据主键来查询效率更高；事实上无论是使用主动声明的自增主键还是MySQL启动隐藏的自增主键，都是为了这一步：链表排序插入。而且所有的索引结构也都是建立在这第一步之上的，后面会介绍索引的“回表”操作，回表行为就是在定位自增主键。

**1.3.2 无主键的解决方案**

如果没有主键，使用的是uuid，那么如何进行排序插入？MySQL给出的方案是首先检查是否存在唯一索引，否则启用隐藏的字段“row\_id”来解决这个问题，row\_id是自增的。唯一索引和自增主键的结构非常相似。

1.3.3 页目录 与 链表结构带来的查询效率问题

默认的页大小为16KB，现在有这样一种极端情况出现：我们有一张表，只有一个字段，int类型，占4比特，那么此页的【用户数据区域】中的链表最多保存：4096个记录。这非常非常的长，链表最大的缺点就是查询速度慢，这样就会出现比较大的性能损耗。那么Mysql是如何解决长链表查询速度慢的问题呢？答案是使用【页目录】，页目录就像书籍的目录一样。



那么**页目录**如何构成？

第一步：对链表中的数据进行分组，假如每组50条。

第二步：我们提取每一组中最大的那条记录(最后一条)的地址偏移量。

第三步：将地址偏移量保存到页目录的槽中。

页目录中保存 地址偏移量 的单位叫“槽”，页目录实际上是由槽组成的。

槽有一个属性：槽高，此属性主要用来进行二分查找。

**注**：实际分组时，每组的记录小于8条，这里写50仅仅是方便演示和计算。

这里和HashMap使用8作为链表的临界值是一样的，没有原因，就是大量测试

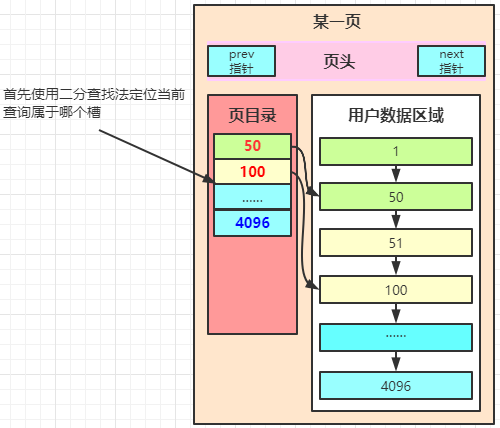
验证出来的最佳值。

实际上“页目录和用户数据区域”共同组成了一个Hash结构，使用的也是哈希思想。

1.4 MySQL如何完成一次查询

1.4.1 查询的基本设计思路

select \* from demo where a = 95; 结合1.3.3中提出的问题，面对这样一条sql语句，mysql是这样设计的：



**第一步**：页目录二分查找定位“槽”的位置。

页目录中的数据是排好序的，所以使用二分查找没问题。

**第二步**：定位到当前槽所在的分组，找到第一个元素从前到后

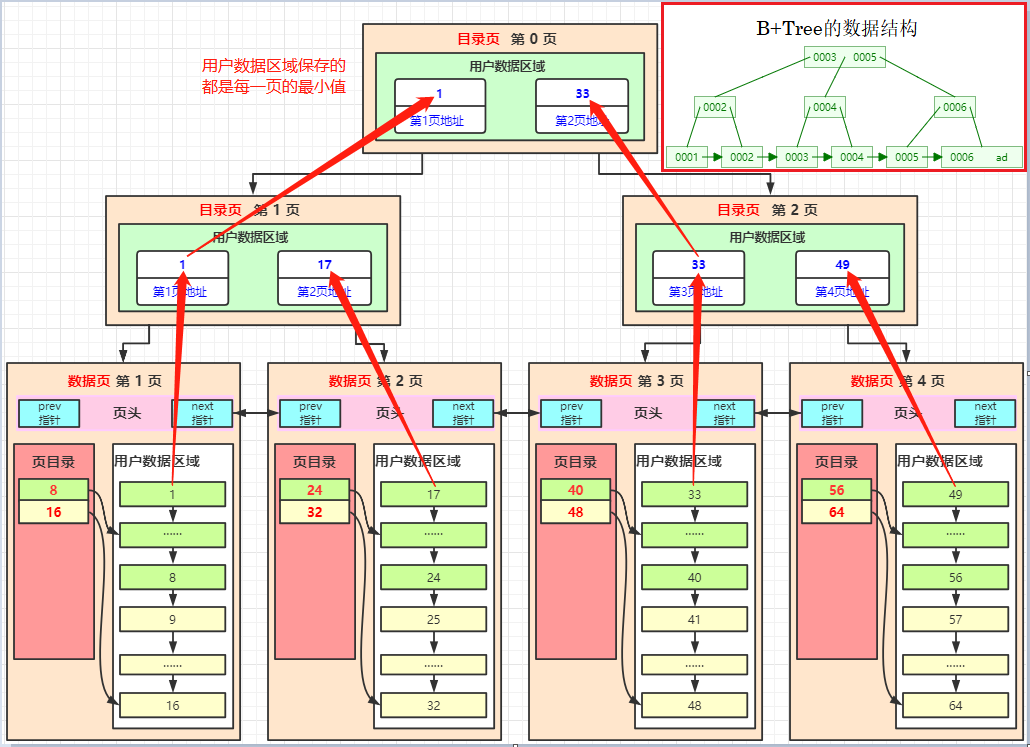
进行对比。

注意：这里a是主键，是默认的索引，所以会按照上述方式查找；

如果a不是主键，那么会进行全表扫描，直到找到结果为止。

1.4.2 查询的全局设计思路

当数据量越来越大，几十万条记录出现的时候，一页无法满足，此时就用到了B+Tree的数据结构，我们这里的每一个数据页都变成了叶子节点。解决思路就是“以空间换时间”。下图以自增ID为索引组织。



目录页的用户数据区保存的都是两个“页”的地址，数据页处于最底层，属于B+Tree的叶子节点，并且以指针相连。

那么问题来了，针对select \* from demo where a = 28; 这条sql语句，会如何执行？总体上采用二分查找的方式来定位数据。

第一步：在【目录页 第0页】对比，因为1< 28 <33，所以进入【目录页 第1页】

第二步：在【目录页 第1页】中，1 < 28 && 17 < 28，所以进入【数据页 第2页】

第三步：在【数据页 第2页】中的页目录继续进行二分查找，因为28 > 24，所以在32所在的槽分组进行链表遍历。

第四步：判断数据是否存在并返回。

从上图可以看出，新增、修改和删除都是比较消耗性能的操作。此处还会涉及到“页分裂和页合并”。

1.5 索引

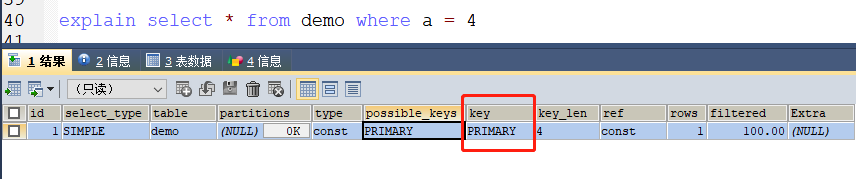
1.5.1 索引定义

索引是一种数据结构，是一种组织数据的方式。通常来讲，主键索引是默认的**一级索引** (如果没有主键也会有row\_id所支撑的索引)；主键索引又叫聚簇索引，他有二个特性：

1）按照主键排序。无论数据页还是目录页，都是按照主键进行排序的。

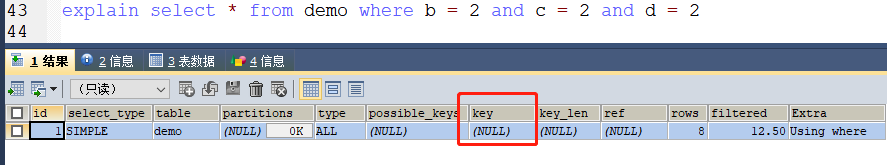
2）最底层的叶子节点存储了完整的数据信息，包含隐藏字段。

也就是说针对主键索引：索引结构即是保存好的数据，保存好的数据即索引；如果没有后续手动创建的索引，甚至可以认为它就是这张数据库表本身。我们手动创建的索引是二级索引，包括联合索引、哈希索引等；他们不保存完整的数据信息的(唯一索引除外)，只保存了其主键或row\_id；手动创建的索引都需要以一级索引作为查询依据，这个行为叫做：回表。我们看一个主键索引：

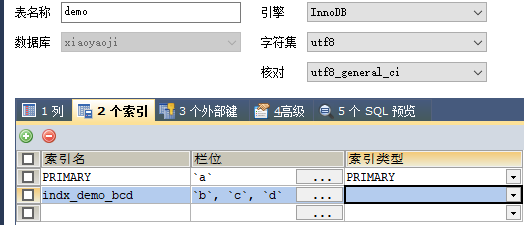


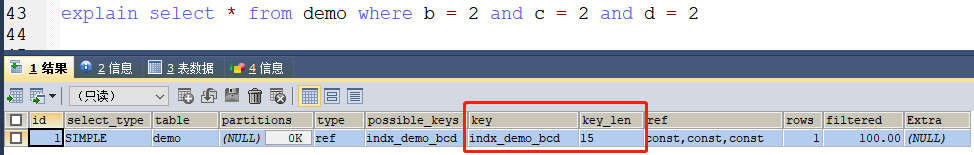
1.5.2 联合索引

我们执行一条sql：select \* from demo where b = 2 and c = 2 and d = 2；未创建索引之前，如果我们执行这条sql，MySQL会进行全表扫描，一条一条的找，直到定位到我们要找的数据。

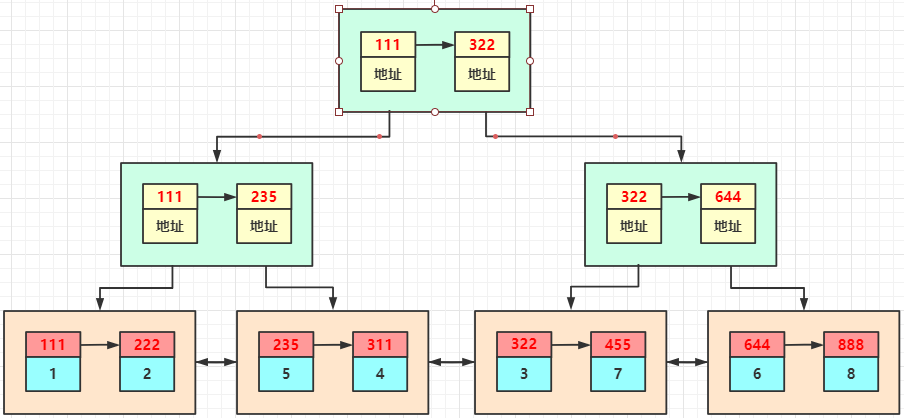


下面我们创建一个bcd联合索引：create index indx\_demo\_bcd on demo(b,c,d)，结果如下图：





我们创建的这个索引：indx\_demo\_bcd，也会生成一个B+Tree。它排序的方式是按照先b，后c，最后d的这种方式进行。也就是说：索引一旦建立它就有了一个**固定的顺序**，正是这个顺序导致了某些条件下索引不生效的问题。其对象数据结构展示如下：



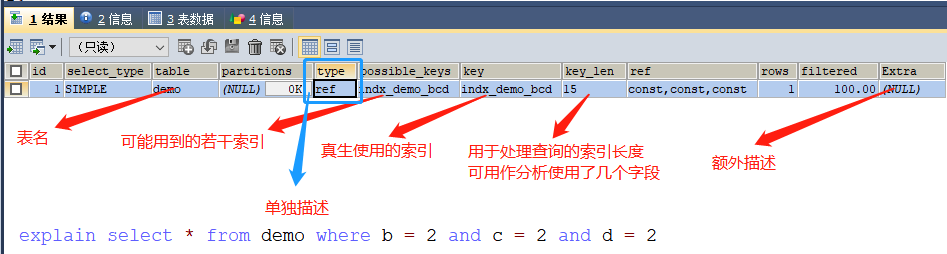
 对照数据表中的数据。

从bcd索引对象数据结构图中可以看出，我们创建的联合索引与主键索引数据结构上是一致的，都采用的B+Tree，只不过排序依据变的复杂了。不同点在于最底层的叶子节点没有保存全部数据信息，而是只保留了主键信息。当我们定位到222这条记录的时候，我们只拿到了主键是“2”，下一步我们还需要到“一级索引”中根据2找到全部的记录信息，这个过程叫：回表。MySQL5.7以前的版本，有出现多次回表的情况，效率较低；5.7以后的版本出现了“索引下沉”，将回表的此数降到了1次。效率提升很多。

在MySQL中，每新建一个索引就会出现一份新的索引数据结构；所以数据库中的表可以看成是由一个“一级索引”和多个二级索引共同构成。也就是说，索引建的越多，增删改的效率越低，这也是不建议多建索引的原因。通常建议扩充索引字段来达到提高查询效率的目的，但这是一个比较有难度且耗神的技术活儿。

1.6 索引的运行原理与生效条件分析

1.6.1 关于explain



**关于type字段**：依次从好到差：system，const，eq\_ref，ref，fulltext，ref\_or\_null，unique\_subquery，index\_subquery，range，index\_merge，index，ALL，除了all之外，其他的type都可以使用到索引，除了index\_merge之外，其他的type只可以用到一个索引。如果显示的是all，则代表没有使用索引，进行了全表查询。

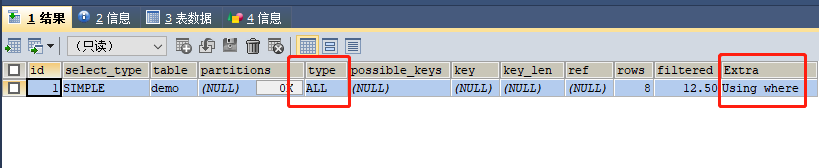
**关于ref字段**：与索引作比较的列；上图显示是3个。

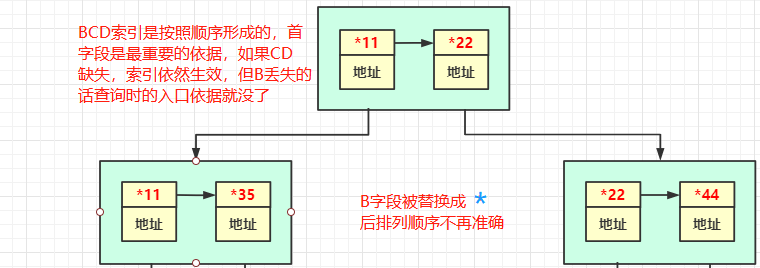
1.6.2 索引字段位置颠倒不会引起索引失效

我们创建索引的顺序是：bcd，那么实际开发中explain select \* from demo where d = 2 and b = 2 and c = 2；bcd的顺序变了，是否会有影响？答案是不会，因为索引执行器会自动判断和优化。

1.6.3 最左前缀原则：索引首字段缺失→索引失效

bcd顺序索引，如果没有b，那么索引失效。比如：explain select \* from demo where c = 2 and d = 2 执行结果如下：



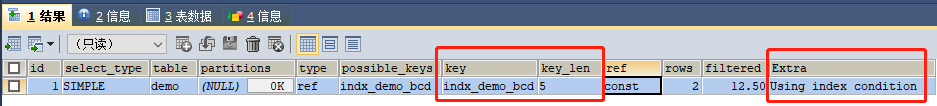


在使用二分查找时，由于缺少了B的存在，所以无法判断向左边查找还是右边查找。这就是**最左前缀原则**。

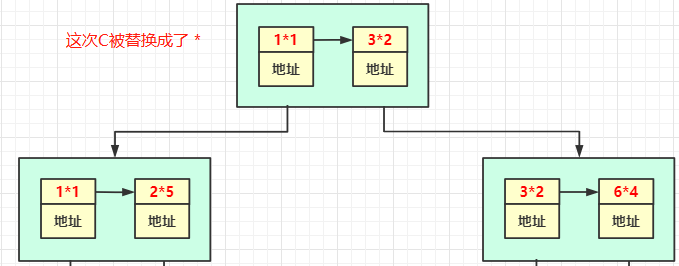
1.6.4 索引条件下推

分别检查如下两个索引执行结果：

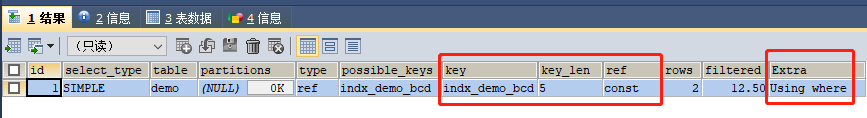
explain select \* from demo where b = 2 and d = 2



b和d都是索引组成的一部分，但是只用到了一个字段，所以key\_len=5，ref只有一个值。但是我们可以看到Extra的值是：Using index condition；使用了索引条件。



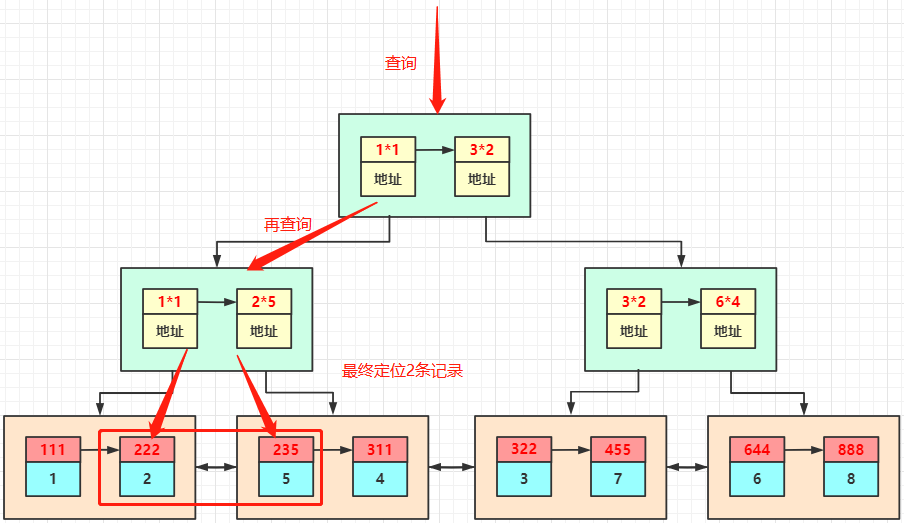
我们再来执行和观察：explain select \* from demo where b = 2 and e = "共和国万岁! "；这里的e字段不是联合索引字段的一部分。



可以看到Extra的值是：Using where。

我们以这条SQL的执行为例：select \* from demo where b = 2 and d = 2。

在5.7版本以前：



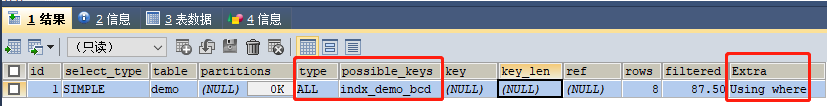
第一个索引的字段生效，我们最终会定位到2条记录，他们都是以2开头的；下一步会拿着这两条记录对应的主键2和5全部进行回表操作。拿到2条完成的记录以后，我们再根据d字段进行比较，确定最终的记录。会出现多次回表的过程。

在5.7版本以后，如果d=2的这个条件是索引的一部分，那么数据库会进行自动判断，对比得到正确值后，回表一次；不会全部回表，从而提升查询效率。但是如果这里的条件换成：e = "共和国万岁! "，由于e并不是索引字段的一部分，所以这里就不会触发索引条件下推的操作，而是全部回表完成对比，效率低很多。

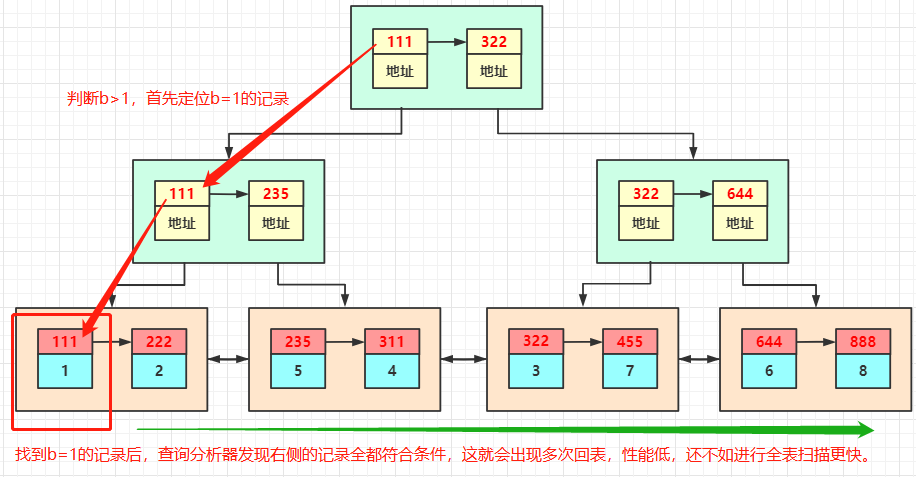
1.6.5 查询优化器的成本计算

因为回表行为的存在，也就带来了性能消耗的不确定性，所以有的时候虽然存在索引但未必会使用到。这些都由查询优化器自己来掌握，他会找到执行成本最低的方案。比如下列情况：

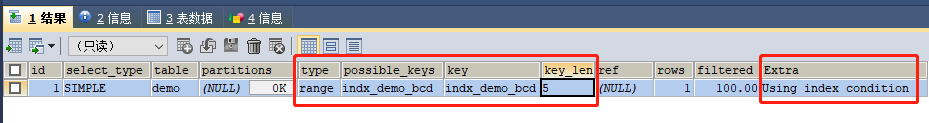
explain select \* from demo where b > 1；



possible\_keys = index\_demo\_bcd，但是type=ALL，Extra显示：Using where。也就是说：查询优化器可能执行index\_demo\_bcd索引，但最终选择了进行全表扫描。因为在索引对象图中：

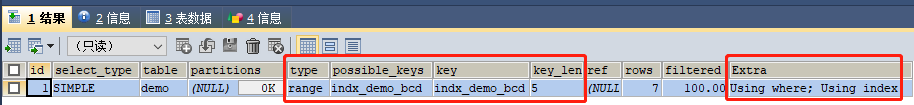


* 再看：explain select \* from demo where **b > 6**；



type=range代表使用了<、>、>、 >=、<、<=、IS NULL、BETWEEN、IN()或者like。索引又特么活了！

* 再看：explain select **c** from demo where b > 1；依然是条件b > 1，但不再select \* 查询全部字段了。



这次索引依然生效！这是因为字段b、字段c和字段d都在当前索引结构中，我们不需要回表就能拿到这个值。这就是**覆盖索引**。但是此处如果要是改成：explain select **e** from demo where b > 1；就不会使用索引，因为字段e不在索引内，需要进行回表才能找到e的值到底是什么，也就是说和select \* 没有区别。

1.7 Hash索引

Java高并发底层原理与实际应用

JMM: Java Memory Model

1) 主内存、工作内存与堆栈的关系 【理论基础】

JVM将内存组织为**主内存**和**工作内存**两个部分。主内存主要包括本地方法区和堆。每个线程都有一个工作内存，工作内存中主要包括两个部分，一个是**属于该线程**私有的栈和对主存部分变量拷贝的寄存器(包括程序计数器PC和cup工作的高速缓存区)。



1) 所有的变量都存储在主内存中(虚拟机内存的一部分)，对于所有线程都是共

享的。

2) 每条线程都有自己的工作内存，工作内存中保存的是主存中某些变量的拷贝。

3) 线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中

的变量。

4) 线程之间无法直接访问对方的工作内存中的变量，线程间变量的传递均需要

通过主内存来完成。

简而言之【工作内存：强调私有性；主内存：强调共享性】。进程代表一个软件，操作系统会为每一个运行的软件创建一个进程，比如：一个Tomcat进程代表了它在操作系统中运行的标识，这个Tomcat启动了一个线程池，包含200个线程，并发处理客户端传来的请求。每一个线程中会包含若干对象、字段、静态变量等等。

* **堆区(heap)：**

存储的全部是对象，每个对象都包含一个与之对应的class的信息。(class的目的是得到操作指令，注意：一个class可以创建多个对象)。jvm只有一个堆区被所有线程共享，堆中不存放基本类型和对象引用，只存放对象本身。

* **栈区**：

每个线程包含一个栈区，栈中只保存基础数据类型8个和自定义对象的引用(不是对象)，对象都存放在堆区中。每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问。栈分为3个部分：基本类型变量区、执行环境上下文、操作指令区(存放操作指令)。

* **方法区**：

又叫静态区，跟堆一样，被所有的线程共享。方法区包含所有的class和static变量。方法区中包含的都是在整个程序中永远唯一的元素，如class，static变量。（参考：<https://www.cnblogs.com/cmfwm/p/7671188.html>）

JAVA寄存器

所有进程都使用寄存器，Java开发组决定Java只使用四个寄存器，这是因为如果使用的寄存器数多于处理器端口数，那么处理器的效率将严重地降低。Java虚拟机使用下列寄存器管理系统堆栈：

**程序记数寄存器**：跟踪程序执行的准确位置

**堆栈指针寄存器**：指示操作栈项

**框架寄存器**：指向当前执行的环境

**变量寄存器**：指向当前执行环境中第一个本地变量

堆栈分离的本质：

在现代计算机系统中，读写速度分级：硬盘<内存<CPU；CPU的读写速度远高于内存，导致CPU会出现大量空闲时间（因为在等待从内存中读取数据），因此在内存和CPU之间增加了一套缓存机制，一般打开CPU-Z会显示你的CPU有3级缓存，6M左右。**栈**保存的是若干个对象在**堆**中的内存地址，当一个线程在运行的时候，当前这个线程会根据栈中保存的地址去堆中拷贝这个对象的副本中的字段和字段的值到工作内存中，降低CPU空闲等待时间，从而提升系统性能。

JVM规范定义了线程对内存间交互操作：

**交互规则**：

1. 锁定：Lock，作用于主内存中的变量，把一个变量标识为一条线程独占的状态。
2. 读取：Read，作用于主内存中的变量，把一个变量的值从主内存传输到线程的工作内存中。
3. 加载：Load，作用于工作内存中的变量，把read操作从主内存中得到的变量的值放入工作内存的变量副本中。
4. 使用：Use， 作用于工作内存中的变量，把工作内存中一个变量的值传递给执行引擎。
5. 赋值：Assign，作用于工作内存中的变量，把一个从执行引擎接收到的值赋值给工作内存中的变量。
6. 存储：Store，作用于工作内存中的变量，把工作内存中的一个变量的值传送到主内存中。
7. 写入：Write，作用于主内存中的变量，把store操作从工作内存中得到的变量的值放入主内存的变量中。
8. 解锁：Unlock，作用于主内存中的变量，把一个处于锁定状态的变量释放出来，之后可被其它线程锁定。

**并且这8种操作必须遵循以下规则**：

1、不允许read和load、store和write操作之一单独出现。即不允许一个变量从主内存被读取了，但是工作内存不接受，或者从工作内存回写了但是主内存不接受。

2、不允许一个线程丢弃它最近的一个assign操作，即变量在工作内存被更改后必须同步改更改回主内存。

3、工作内存中的变量在没有执行过assign操作时，不允许无意义的同步回主内存。

4、在执行use前必须已执行load，在执行store前必须已执行assign。

5、一个变量在同一时刻只允许一个线程对其执行lock操作，一个线程可以对同一个变量执行多次lock，但必须执行相同次数的unlock操作才可解锁。

6、一个线程在lock一个变量的时候，将会清空工作内存中的此变量的值，执行引擎在use前必须重新read和load。

7、线程不允许unlock其他线程的lock操作。并且unlock操作必须是在本线程的lock操作之后。

8、在执行unlock之前，必须首先执行了store和write操作。

可见，锁操作都是围绕主内存来进行。在将变量从主内存读取到工作内存中，必须顺序执行read、load；要将变量从工作内存同步回主内存中，必须顺序执行store、write。

2) JMM 【理论基础】

JVM是整个计算机虚拟模型，属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过插入特定类型的 **Memory Barrier**（内存屏障）来禁止特定类型的编译器重排序和处理器重排序，为上层提供一致的内存可见性保证；JMM 决定一个线程对共享变量(实例域、静态域和数组)的写入何时对其它线程可见（线程的可见性）。

由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存（也称为：**栈空间**），工作内存是每个线程的私有数据区域，而Java内存模型中规定所有变量都存储在**主内存**，主内存是共享内存区域，所有线程都可访问，但线程对变量的操作(读取赋值等)必须在栈空间中进行，首先要将变量从主内存拷贝到自己的栈空间，然后对变量进行操作，操作完成再将变量写回主内存，不能直接操作主内存中的变量，各个线程中的工作内存，储存着主内存中的变量副本拷贝，因为不同的线程无法访问对方的工作内存，因此线程间的通讯(传值) 必须通过主内存来完成,其简要访问过程如下图：

**JMM关于同步规定**：

1 线程解锁前，必须把共享变量的值刷新回主内存。

2 线程加锁前，必须读取主内存的最新值到自己的工作内存。

3 加锁解锁是同一把锁。

线程A和线程B之间需要通信，必须经过3个步骤：

1 线程A和线程B均从主内存中读取同一个共享变量到自己的工作内存中，各自进行计算。

2 线程A把自己工作内存中更新过的共享变量副本刷新到主内存中。

3 线程B到主内存中读取线程A之前更新过的共享变量。

3) Happen-Before规则【学术讨论】

Java的并发采用的是共享内存模型，使用happen-before规则实现共享变量的同步操作(简称hb规则)。该规则定义了Java多线程操作的有序性和可见性，防止了 编译器重排序 对程序结果的影响。按照官方的说法：当一个变量被多个线程读取并且至少被一个线程写入时，如果读操作和写操作没有HB关系，则会产生数据竞争问题。要想保证操作B的线程看到操作A的结果(无论A和B是否在一个线程)，那么在A和B之间必须满足HB原则，如果没有，将有可能导致重排序。当缺少HB关系时，就可能出现重排序问题。

**HB有哪些规则**：(http://ifeve.com/java-使用-happen-before-规则实现共享变量的同步操作/)

1. 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作。
2. 锁定规则：在监视器锁上的解锁操作必须在同一个监视器上的加锁操作之前执行。
3. volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作。
4. 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C。(*这个规则至关重要，如何熟练的使用传递规则是实现同步的关键*。)
5. 线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。
6. 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生。
7. 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行。
8. 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始。

4) JMM内存交互3个基本特性【学术讨论】

原子性：

原子性即一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。原子是世界上的最小单位，具有不可分割性。在 Java 中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

可见性：

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。JMM 是通过在线程 A 变量工作内存修改后将新值同步回主内存，线程 B 在变量读取前从主内存刷新变量值，这种依赖主内存作为传递媒介的方式来实现可见性。

有序性：

**线程内**，从某个线程的角度看方法的执行，指令会按照一种叫 "串行"（as-if-serial）的方式执行，此种方式已经应用于顺序编程语言。

**线程间**，这个线程 " 观察 " 到其他线程并发地执行非同步的代码时，由于指令重排序优化，任何代码都有可能交叉执行。

唯一起作用的约束是：对于同步方法，同步块（synchronized 关键字修饰）以及 volatile 字段的操作仍维持相对有序。JMM保证有序性的手段是：禁止指令重排序。JMM规定的这三个基本特性，是Java多线程的核心原理。

5) 指令重排序 与 内存屏障【理论基础】

指令重排序

**多线程环境中**线程交替执行，由于编译器优化重排的原因，两个线程中使用的变量能否保证一致性是无法确定的，结果无法预测。计算机在执行程序时，为了提高性能，编译器和处理器常常会做指令重排，一般分为以下三种重排方式：



单线程环境里面，不存在指令重排序所造成的的问题，程序最终执行结果和代码顺序执行的结果一致。在多核心CPU，多线程的环境下，处理器在进行重新排序的时候，必须要考虑指令之间的数据依赖性。

**数据依赖性**

数据依赖性是一种因果关系。如右图所示，源代码经过重排序后，可能出现的执行顺序如下：



上述源代码有三种可能出现的重排序方式，分别为：1234（排序不变）、2134和1324。由于数据依赖性的存在，第四种情况由于y需要x变量先赋值计算完成，所以不可能排到x变量参与计算的代码之前。直观感受这种情况：1243，也是违背数据依赖性原则的，所以也不会出现。

**结果无法预测** 代码如下：

int a = 0;

int b = 0;

int x = 0;

int y = 0;

|  |  |
| --- | --- |
| 线程1 | 线程2 |
| x = a | y = b |
| b = 1 | a = 2 |
| x = 0 y = 0 | |

如果编译器对这段代码进行执行重排优化后，可能出现下列情况：

|  |  |
| --- | --- |
| 线程1 | 线程2 |
| b = 1 | a = 2 |
| x = a | y = b |
| x = 2 y = 1 | |

内存屏障 - Memory Barrier

**指令重排序是编译器的一种优化策略，但有时候我们需要禁止他，此时需要使用内存屏障。**Memory Barrier又称内存栅栏，是一个 CPU 指令。可以分为：LoadLoad屏障、LoadStore屏障、StoreStore屏障和 StoreLoad屏障。内存屏障有两个作用：

特性1：保证特定操作的执行顺序。

特性2：保证某些变量的内存可见性（**volatile的内存可见性**）。

这其中**特性1**：阻止屏障两侧的指令重排序，插入一条 Memory Barrier 会告诉编译器和 CPU，不管什么指令都不能和这条 Memory Barrier 指令重排序；也就是说：可以通过插入内存屏障来禁止在内存屏障前后的指令被重排序优化。**特性二**：强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效。如一个Write-Barrier（写入屏障）将刷出所有在Barrier之前写入cache的数据，因此任何CPU上的线程都能读取到这些数据的最新版本，让每个线程栈重新去堆中再次拷贝变量副本。

内存屏障阻碍了CPU采用优化技术来降低内存操作延迟，因此必定会带来性能损失。Java中对内存屏障的使用常见的有 volatile 和 synchronized 关键字修饰的代码块，还可以通过 Unsafe 这个类来使用内存屏障。

volatile【实际运用】

volatile是Java虚拟机提供的**轻量级**同步机制，主要解决两个问题：【**保证可见性**】和【**禁止指令重排序**】，不保证原子性。是乞丐版的synchronized，阉割了线程安全性。其最底层在操作CPU的内存屏障。通过前面对JMM的铺垫，各个线程对主内存中共享变量的操作都是各个线程各自拷贝到自己的工作内存操作后再写回主内存中的。这就可能存在一个线程A修改了共享变量X的值还未写回主内存中时，另外一个线程B又对内存中的一个共享变量X进行操作，但此时A线程工作内存中的共享比那里X对线程B来说并不不可见。这种工作内存与主内存同步延迟现象就造成了可见性问题。禁止指令重排序则解决了顺序性问题。

可见性-验证代码

public class MyData {

// 变量未添加volatile关键字，没有可见性

public int number = 0;

// public volatile int number = 0;

public int getNumber() {

return number;

}

public void setNumber(int number) {

this.number = number;

}

}

public class VolatileMain {

public static void main(String[] args) {

MyData data = new MyData();

new Thread( () -> {

System.out.println(Thread.currentThread().getName() + " 进入");

// 线程等待3秒

try {TimeUnit.SECONDS.sleep(3);} catch (InterruptedException e) {e.printStackTrace();}

data.setNumber(10); // 设置一个新值 10

System.out.println(Thread.currentThread().getName() + " 更新 number 值为：" + data.getNumber());

} , "线程A" ).start();

while (data.getNumber() == 0) {

// main 线程在此处等待，直到number值不在等于0。但因为线程A修改number的值后，main线程没有得到通知，此处会一直循环。

}

// main 线程“想”打印出来，但打印不出来。

System.out.println(Thread.currentThread().getName() + " mission is over");

}

}

代码解读：

在VolatileMain.java中，线程A和Main线程同时操作一个MyData()对象；线程A在等待3秒后，将number设置成10。但由于JMM内存交互的可见性特点，number字段没有添加volatile关键字，故线程A改变了number的值但是没有通知其他线程，造成其他线程无限等待。

1. 线程A和Main线程同时操作MyData对象；
2. 线程A启动，栈中记录了一个对象引用地址：MyData data。Main线程进行了相同的操作。
3. Main线程开始执行自旋锁：while，Main线程在此处等待，直到number值不在等于0
4. 3秒钟过后，线程A开始在拷贝MyData的副本到【工作内存】中，同时开始计算number的值，变为10。
5. 线程A将新值写入到主内存的堆空间的目标对象中，MyData对象的值得到更新，线程执行完成并退出。
6. 由于没有通知其他线程，主内存发生改变，所以Main线程继续等待，直到外部强制其退出。



当我们为MyData.java添加的number字段添加了volatile关键字后，这个问题得到解决：



不保证原子性

验证代码

public class MyData {

// 变量未添加volatile关键字，没有可见性

public volatile int number = 0;

public void addNumber() {

this.number ++;

}

public int getNumber() {

return number;

}

public void setNumber(int number) {

this.number = number;

}

}

public class VolatileMain {

public static void main(String[] args) {

MyData data = new MyData();

for(int i = 1 ; i <= 20 ; i ++) {

new Thread( () -> {

for(int n = 0 ; n < 1000 ; n ++) {

data.addNumber();

}

} , "线程-" + i ).start();

}

while (Thread.activeCount() > 2) {

// 等待上面20个线程计算完成，在用Main线程获取最终结果

Thread.yield();

}

// 由于期待值是：20000，但由于volatile并不具有原子性，故这里打印出正确数值的概率极小。

System.out.println(Thread.currentThread().getName() + " finally number value is : " + data.getNumber());

}

}

运行结果如下：



底层原因分析

原子性即一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。但是volatile不保证这种特性，究其原因需要从底层字节码的角度来分析，在eclipse中启用javap –c后，得到MyData.java（已经去掉和此方法无关的字节码）中addNumber()方法的Java字节码如下：

Compiled from "MyData.java"

public class com.matrix.sxt.e02.MyData {

public volatile int number;

public void addNumber();

Code:

0: aload\_0

1: dup

2: getfield #12 // Field number:I

5: iconst\_1

6: iadd

**7: putfield** #12 // Field number:I

10: return

}

aload\_0：从局部变量0中装载引用类型值。在源码中：public volatile int number = 0; 这一步初始化了number=0的这个变量。

dup：复制栈顶部一个字长内容。JMM规定栈负责运行，会加载类中的实例变量和8种基本类型数据到栈内，代码中number被声明为了int类型，正好是实例变量又是8种基本类型之一。

number++：一共进行了4步操作：

getfield： 拿到原始值number。由于没有加锁所以多个线程间会抢夺，都会从主内存中拿到原始值：0。

其中：Field number:I，这里的I代表number是 int类型数据。

iconst\_1： 将拿到的值压入栈（当int取值-1~5时，JVM采用iconst指令将常量压入栈中）。

iadd：

int型，加1次“1”；因为number++ 等价于number = number + 1。

如果这里是number = number + 2，那么这里的语义解释为：int型，加1次“2”。

这一步是在各自的工作内存中执行，不在主内存中执行。

**putfield**：写回主内存。

在多线程环境下，无法保证原子性的原因就出现在了这4步操作上。因为线程在CPU中以纳秒为单位在急速切换，这四步每一步都可能在当前这个线程执行的时候被CPU挂起。在iadd和putfield这两步，当前线程在自己的工作内存中加1后，写回主内存的过程中会出现脏数据覆盖。比如：线程A计算完值为1，但在putfield执行之前被挂起；线程B计算完=1，写入主内存，线程B再次计算=2，其他线程依然挂起，线程B写入主内存。此时线程A被唤醒，写入主内存，number又变为1。

解决方案

解决这个问题最简单的方式是使用AtomicInteger，原子整形会进一步引申出CAS。代码修改如下：

public class MyData {

// 变量未添加volatile关键字，没有可见性

public volatile int number = 0;

public AtomicInteger atomicInteger = new AtomicInteger();

public void addNumber() {

this.number ++;

}

public void atomicAdd() {

atomicInteger.getAndIncrement();

}

public int getNumber() {

return number;

}

public AtomicInteger getAtomicValue() {

return atomicInteger;

}

}

public class VolatileMain {

public static void main(String[] args) {

MyData data = new MyData();

for(int i = 1 ; i <= 20 ; i ++) {

new Thread( () -> {

for(int n = 0 ; n < 1000 ; n ++) {

data.addNumber();

data.atomicAdd(); // 调用原子整形

}

} , "线程-" + i ).start();

}

while (Thread.activeCount() > 2) {

// 等待上面20个线程计算完成，在用Main线程获取最终结果

Thread.yield();

}

// 由于期待值是：20000，但由于volatile并不具有原子性，故这里打印出正确数值的概率极小。

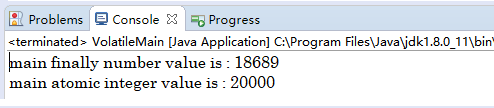
System.out.println(Thread.currentThread().getName() + " finally number value is : " + data.getNumber());

System.out.println(Thread.currentThread().getName() + " atomic integer value is : " + data.getAtomicValue());

}

}

运行结果如下：

DCL双端检锁机制单例优化 – 实例

public class SingletonVolatile {

private static **volatile** SingletonVolatile instance = null;

private SingletonVolatile() {

System.out.println(Thread.currentThread().getName() + " SingletonVolatile私有构造函数实例化");

}

// DCL Double check lock 双端捡锁

public static SingletonVolatile getInstance() {

if(instance == null) {

synchronized (SingletonVolatile.class) {

if(instance == null) {

instance = new SingletonVolatile(); // 双端捡锁问题点

}

}

}

return instance;

}

public static void main(String[] args) {

for (int i = 1; i <=1000; i++) {

new Thread(() ->{ SingletonVolatile.getInstance(); } , "当前线程：" + String.valueOf(i) ).start();

}

}

}

双端检锁机制不一定线程安全，原因是有指令重排的存在，加入volatile可以禁止指令重排从而达到线程安全。原因在于某一个线程在执行到第一次检测，读取到的instance不为null时，instance的引用对象可能没有完成初始化。instance = new SingletonVolatile(); 这段代码的执行可以分解为以下3个步骤：

1. memory = allocate(); 分配对象内存空间
2. instance(memory); 初始化对象
3. instance = memory; 设置instance的指向刚分配的内存地址,此时instance!=null

步骤2和步骤3不存在数据依赖关系。而且无论重排前还是重排后程序执行的结果在单线程中并没有改变，因此这种重排优化：1->3->2是允许的。但是指令重排只会保证串行语义的执行一致性(单线程) 并不会关心多线程间的语义一致性，所以当一条线程访问instance不为null时，由于instance实例未必完成初始化，也就造成了线程安全问题。

原子引用

原子引用是对volatile的进一步完善和封装，他提供了volatile不具备的原子性，但没有使用线程锁synchronized，所以系统并发性不受影响。

Compare and Swap(CAS)【实际运用】

CAS：CPU原子操作

即：**比较并交换**。多个线程工作内存中的“**一个**”变量A(堆中的副本拷贝)和主内存中的这个变量A进行比较，如果期望值与真实值一致，那么更新为当前线程指定的值；CAS只能操作一个变量，是一条CPU并发原语，它的功能是判断内存某个位置的值是否为预期值，如果是则更新为新的值，这个过程是原子的。

* **如何保证原子性**？

CAS并发原语体现在Java语言中是rt.jar包中的sun.misc.Unsafe.java（*oracle未公布rt.jar中的jdk包和sun包下的源代码*）类中的各个方法。调用Unsafe类中的CAS方法，JVM会帮我实现CAS汇编指令，这是一种完全依赖于硬件功能，通过它实现了原子操作。关于**系统原语**：原语属于操作系统用途范畴，是由若干条指令组成，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许中断（原语的执行必须是连续的，就像国家主席车队出行会出现交通管制，绝对不允许社会车辆插队）。也就是说CAS是一条原子指令，不会造成所谓的数据不一致的问题。

比如本段落中的“Unsafe类中的CAS方法”具体指：public final native boolean compareAndSwapInt(Object o, long offset, int expected, int x); 他是一个由native修饰的本地方法，该方法的实现位于：unsafe.cpp中，其汇编指令如下：



这其中：Atomic::cmpxchg…事实上在汇编层面，只要是以Atomic开头的，都是原子操作，不允许插队。可以查看JVM源码中的Atomic.hpp、Atomic.inline.hpp等相关文件内容，里面会有几个cmpxchg重载函数(compare and exchange 比较并交换)。该方法就是JVM原子操作的底层实现。由于每个平台架构实现都不一样所以无法把所有的实现代码都列举出来，这里只看atomic\_windows\_x86的相关内容：

// 如果是多处理机系统，添加0xF0前缀  
#define LOCK\_IF\_MP(mp) \_\_asm cmp mp, 0 \

\_\_asm je L0 \

\_\_asm \_emit 0xF0 \  
 \_\_asm L0:

inline jint **Atomic::cmpxchg** (jint exchange\_value, volatile jint\* dest, jint compare\_value) {  
 // 判断系统是否为多处理机系统(MP MultiProcessor)  
 int mp = os::is\_MP();  
 \_\_asm {  
 mov edx, dest  
 mov ecx, exchange\_value  
 mov eax, compare\_value  
 LOCK\_IF\_MP(mp)  
 cmpxchg dword ptr [edx], ecx  
 }  
}

AtomicInteger.java源代码解析

public class CasDemo {

public static void main(String[] args) {

AtomicInteger ai = new AtomicInteger(5); // 初始化的值为5(否则默认为0)，此值会放到主物理内存(堆中)

ai.getAndIncrement(); // 进行自增加1操作后，变为6。此方法极具CAS代表性，后面细聊。

boolean flag1 = ai.compareAndSet(6, 2020); // 堆中的变量为6，线程期望主内存的值为6，那么更新为新的值：2020

boolean flag2 = ai.compareAndSet(6, 2021); // 堆中的变量为2020，线程期望主内存的值为6，由于不一致，所以更新为2021的时候失败。

System.out.println(flag1 + " 当前值：" + ai.get());

System.out.println(flag2 + " 当前值：" + ai.get());

}

}

运行如下：



如果线程的期望值和主物理内存的真实值一致，那么则修改更新值；如果不一致，那么修改失败，重新获得主物理内存的真实值。CAS的底层原理是自旋锁和Unsafe.java。打开AtomicInteger.java底层源代码，基本是对Unsafe类的封装。

public class AtomicInteger extends Number implements java.io.Serializable {  
 private static final long serialVersionUID = 6214790243416807050L;

private volatile int value; // 提供可见性与顺序性保障

private static final Unsafe unsafe = Unsafe.getUnsafe(); // 提供原子性保障

private static final long valueOffset;  
static {  
 try {

// 使用Unsafe类获取一个字段的内存偏移量（即内存地址），同时赋值给valueOffset字段。此处为AtomicInteger.java的value字段

// 后面会通过获取valueOffset这个内存地址，操作该内存地址所对应的值。

valueOffset = unsafe.objectFieldOffset(AtomicInteger.class.getDeclaredField("value"));  
 } catch (Exception ex) {  
 throw new Error(ex);  
 }  
}

……

}

当查看AtomicInteger.java的源码的时候会发现，他首先获取了Unsafe类，然后使用volatile关键字修饰**一个**私有实例变量。AtomicInteger拥有JMM规定的3个基本特性：可见性、有序性和原子性；这其中可见性和有序性由volatile关键字提供保障，而原子性则由Unsafe类来完成。其中valueOffset是内存地址偏移量，他是Unsafe类要操作的指针。Unsafe类相当于一个后门，基于该类可以操作特定内存的数据，其内部方法均为native修饰，可以像C语言的指针一样直接操作物理内存。也就是说Unsafe类中的方法都直接调用操作系统底层资源执行相应任务。

AtomicInteger.java中的getAndIncrement()方法，非常具有CAS的代表性，开始进一步拆解：





这里的代码解析要结合多线程来理解。我们假设有2个线程A和B，这两个线程启动后，同时操作（AtomicInteger ai = new AtomicInteger(5)）同一个AtomicInteger()对象：ai，并且调用ai.getAndIncrement()方法。线程A先执行，它在操作ai对象+1时，ai对象使用自旋锁：

第1步：v = getIntVolatile(o , offset); 获取当前对象（即：ai）在主内存中的当前值作为“期望值”；

第2步：调用compareAndSwapInt方法，他的第二个参数是AtomicInteger的类变量value在堆中的地址，可以直接拿到最新值。但是这两步并不是原子操作，所以第1步中拿到值后，线程A可能被挂起，线程B会整个执行完成，从而将堆内存中的value值变成6。这时线程A继续执行第2步，compareAndSwapInt方法从堆中再次取出value的值，发现是6，不是线程A期望的值5，同时返回false，!false=true，故自旋锁do-while继续生效执行，线程A再次通过getIntVolatile方法，从堆中拿到最新的值6，再次调用compareAndSwapInt进行比较，发现value在堆中的值和线程A的期望值是一样的，那么将执行原子操作：v+1。

注意：

compareAndSwapInt方法是原子性的。

1. getIntVolatile方法和compareAndSwapInt方法前两个参数是一样的；都是在通过操作当前对象的类变量所在的内存地址的方式来获取值。他们都在尝试获取AtomicInteger的类变量private volatile int value的值。new AtomicInteger(5)的时候会在堆中为value分配一个地址，并指定值为5。
2. 这里有JMM线程交互的影子。
3. getAndAddInt方法使用了自旋锁。
4. 第1步和第2步存在时间差，会导致ABA问题。

CAS带来的问题

1、do-while循环，多次比较，时间开销大。

2、只能保证一个共享变量的原子性。当需要对多个共享变量操作时，自旋锁+CAS的方式无法保证操作的原子性，只能通过加锁来实现。

3、引发ABA问题，也称“调包问题”。

ABA问题

CAS算法实现的一个重要前提是需要取出内存中某个时刻的数据，并在当下时刻比较并交换，这里存在的时间差会导致数据变化。CAS操作只有主内存值与预期值相等才会更新主内存中的值，所有CAS操作可能会出现这种现象：原来内存值为A，线程1和线程2都获取该值，然后线程1使用CAS将内存值修改为B，然后又使用CAS将内存值修改回A；这时线程2使用CAS对内存值进行修改时发现内存值仍然是A，然后线程2修改成功。大多数情况下ABA问题并不会对程序造成什么影响。代码如下：

public class ABADemo {

private static AtomicReference<Integer> atomicReference = new AtomicReference<>(100);

public static void main(String[] args) {

System.out.println("=== 以下是ABA问题的产生 ===");

new Thread(()->{

// 线程-1模拟ABA问题

atomicReference.compareAndSet(100,101);

atomicReference.compareAndSet(101,100);

},"线程-1").start();

new Thread(()->{

// 暂停1秒 保证线程-1完成ABA

try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e) { e.printStackTrace(); }

System.out.println(atomicReference.compareAndSet(100 , 200) + " " + atomicReference.get());

},"线程-2").start();

}

}



原子引用实例

AtomicReference

AtomicReference可以实现自定义的原子类；当Java提供的基本原子类，如AtomicInteger无法满足现有业务需求，我们需要自己去定义一个原子类的时候就需要以AtomicReference来包装。它还有一个带版本号的加强类：AtomicStampedReference.java；与AtomicReference不同的是，它的每一次操作都会记录一个版本号，形成一个先后顺序，类似于Git和Svn。示例代码如下：

public class User {

private String name;

private Integer age;

public User(String name, Integer age) {

this.name = name;

this.age = age;

}

}

import java.util.concurrent.atomic.AtomicReference;

public class AtomicRefDemo {

public static void main(String[] args) {

AtomicReference<User> atomicReference = new AtomicReference<User>();

User bzh = new User("巴扎黑" , 22);

User ptq = new User("皮条强" , 44);

atomicReference.set(bzh);

System.out.println("比较替换后的结果：" + atomicReference.compareAndSet(bzh , ptq) + " 当前用户为：" + atomicReference.get().getName());

System.out.println("比较替换后的结果：" + atomicReference.compareAndSet(bzh , ptq) + " 当前用户为：" + atomicReference.get().getName());

System.out.println("比较替换后的结果：" + atomicReference.compareAndSet(ptq , bzh) + " 当前用户为：" + atomicReference.get().getName());

}

}



ABA问题的解决 - AtomicStampedReference

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 线程名 | 初始值 | 版本号 | 线程名 | 修改值 | 版本号 |
| 线程-1 | 100 | 1 |  |  |  |
| 线程2 | 100 | 1 |  |  |  |
|  |  |  | 线程1 | 101 | 2 |
|  |  |  | 线程1 | 100 | 3 |
|  |  |  | 线程2 | 100 | 2 |

表格中，线程B正好完成了一次ABA过程，当线程A被唤醒后，发现自己期待的值还是100，但是版本号却不是最新的。示例代码如下：

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicStampedReference;

public class AbaResolveDemo {

private static AtomicStampedReference<Integer> stampedReference = new AtomicStampedReference<>(100 , 1);

public static void main(String[] args) {

new Thread(()->{

int stamp = stampedReference.getStamp();

String name = Thread.currentThread().getName();

System.out.println(name + " 第1次版本号【" + stamp + "】 值是" + stampedReference.getReference());

// 暂停1秒钟

try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e) { e.printStackTrace(); }

System.out.println("");

System.err.println(name + " 开始模拟ABA过程");

stampedReference.compareAndSet(100,101, stampedReference.getStamp(), stampedReference.getStamp() + 1);

System.out.println(" " + name + " 第2次版本号【" + stampedReference.getStamp() + "】 值是" + stampedReference.getReference());

stampedReference.compareAndSet(101,100, stampedReference.getStamp(), stampedReference.getStamp() + 1);

System.out.println(" " + name + " 第3次版本号【" + stampedReference.getStamp() + "】 值是" + stampedReference.getReference());

System.err.println(name + " 结束模拟ABA过程");

} , "线程-1").start();

new Thread(()->{

int stamp = stampedReference.getStamp();

String name = Thread.currentThread().getName();

System.out.println(name + " 第1次版本号【" + stamp + "】 值是" + stampedReference.getReference());

// 保证线程-1 完成1次ABA

try { TimeUnit.SECONDS.sleep(3); } catch (InterruptedException e) { e.printStackTrace(); }

boolean result = stampedReference.compareAndSet(100, 2019, stamp, stamp + 1);

System.out.println("");

System.out.println(name + " 是否修改成功：" + result + " 最新版本号【" + stampedReference.getStamp() + "】");

System.out.println(name + " 获取的最新的值 " + stampedReference.getReference());

} , "线程-2").start();

}

}



代码中初始值为100，初始版本号为1。

AtomicStampedReference实际应用

AtomicStampedReference实现无锁二叉搜索树 https://github.com/arunmoezhi/LockFreeBST

Unsafe【实际运用】

https://www.jianshu.com/p/db8dce09232d

https://www.cnblogs.com/suxuan/p/4948608.html

**集合类与并发安全性【实际运用】**

**ArrayList**

多线程下ArrayList是不安全的，会报出java.util.Concurrent Modification(n. 修改|修正|改变) Exception异常。

public static void main(String[] args) {

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

for (int i = 1; i <= 30; i++) {

new Thread(() -> {

list.add( Thread.currentThread().getName() );

System.out.println(list);

}, String.valueOf("T-" +i)).start();

}

}

异常结果如下：



**基础解决方式 Vector**

Vector类是Java最早提供的List接口实现类，线程安全。ArrayList是在JDK1.2以后提出的。

public static void main(String[] args) {

List<String> list = new Vector<String>();

for (int i = 1; i <= 30; i++) {

new Thread(() -> {

list.add( Thread.currentThread().getName() );

System.out.println(list);

}, String.valueOf("T-" +i)).start();

}

}

Vector使用synchronized的方式来保证线程安全性，底层代码如下：



**基础解决方式 Collections**

在rt.jar包下提供了java.util.Collections工具集，用于操作集合类；其中包括：synchronizedList。

public static void main(String[] args) {

List<String> list = Collections.synchronizedList(new ArrayList<String>());

for (int i = 1; i <= 30; i++) {

new Thread(() -> {

list.add( Thread.currentThread().getName() );

System.out.println(list);

}, String.valueOf("T-" +i)).start();

}

}

Collections底层也用的synchronized的方式来保证线程安全性：



运行结果如下：



**升级解决方式：写时复制**

在java.util.concurrent包下提供了CopyOnWriteArrayList。写时复制copyOnWrite：容器即写时复制的容器。是一种读写分离的思想，读和写不同的容器。

public static void main(String[] args) {

List<String> list = new CopyOnWriteArrayList<>();

for (int i = 1; i <= 30; i++) {

new Thread(() -> {

list.add( Thread.currentThread().getName() );

System.out.println(list);

}, String.valueOf("T-" +i)).start();

}

}

CopyOnWriteArrayList.java源代码分析如下：

public class CopyOnWriteArrayList<E> implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

private static final long serialVersionUID = 8673264195747942595L;

/\*\* The lock protecting all mutators \*/

final transient ReentrantLock lock = new ReentrantLock();

/\*\* The array, accessed only via getArray/setArray. \*/

// transient关键字标记的成员变量【array】不参与序列化过程；volatile保证【array】变量的可见性和顺序性。

private transient volatile Object[] array;

/\*\*

\* Gets the array. Non-private so as to also be accessible

\* from CopyOnWriteArraySet class.

\*/

final Object[] getArray() {

return array;

}

/\*\*

\* Sets the array.

\*/

final void setArray(Object[] a) {

array = a;

}

……

/\*\*

\* Appends the specified element to the end of this list.

\*

\* @param e element to be appended to this list

\* @return {@code true} (as specified by {@link Collection#add})

\*/

public boolean add(E e) { // 添加元素的时候采用加锁操作

final ReentrantLock lock = this.lock;

lock.lock();

try {

Object[] elements = getArray(); // 获取当前对象保存的数组

int len = elements.length;

Object[] newElements = Arrays.copyOf(elements, len + 1); // 复制当前数组，同时数组长度+1，用于保存新加的元素。

newElements[len] = e; // 新数组赋值

setArray(newElements); // 替换当前对象保存的数组。这会导致一个情况：2个线程可能出现读出的数组内容不同，因为读和写是分离的。

return true;

} finally {

lock.unlock();

}

}

}

**HashMap**

多线程下HashMap是不安全的，也会报出java.util.Concurrent Modification Exception异常，和ArrayList报的异常一样。HashMap的Key和Value**允许**为null；Key允许重复但会覆盖。

public static void main(String[] args) {

Map<String , String> map = new HashMap<String, String>();

for (int i = 1; i <= 30; i++) {

new Thread(() -> {

String name = Thread.currentThread().getName();

map.put(name , name.split("-")[1]);

System.out.println(map);

}, String.valueOf("T-" +i)).start();

}

}



**基础解决方式 Collections.synchronizedMap**

Map<String , String> map = Collections.synchronizedMap(**new** HashMap<String , String>()); 和ArrayList一样，同样的配方，一样的味道。

**基础解决方式Hashtable**

线程安全，key与value都不允许null值。

public static void main(String[] args) {

Map<String , String> map = new Hashtable<String, String>();

for (int i = 1; i <= 30; i++) {

new Thread(() -> {

String name = Thread.currentThread().getName();

map.put(name , name.split("-")[1]);

System.out.println(map);

}, String.valueOf("T-" +i)).start();

}

}

**升级解决方式：ConcurrentHashMap**

ConcurrentHashMap的Key和Value都不允许为null；Key允许重复，但会覆盖。

public static void main(String[] args) {

Map<String , String> map = new ConcurrentHashMap<String, String>();

for (int i = 1; i <= 30; i++) {

new Thread(() -> {

String name = Thread.currentThread().getName();

map.put(name , name.split("-")[1]);

System.out.println(map);

}, String.valueOf("T-" +i)).start();

}

}

**ConcurrentHashMap1.7和1.8的不同实现**

Jdk1.7中采用Segment + HashEntry的方式进行实现。ConcurrentHashMap初始化时，计算出Segment数组的大小ssize和每个Segment中HashEntry数组的大小cap，并初始化Segment数组的第一个元素；其中ssize大小为2的幂次方，默认为16，cap大小也是2的幂次方，最小值为2，最终结果根据根据初始化容量initialCapacity进行计算。Segment在实现上继承了ReentrantLock，这样就自带了锁的功能。

**put实现**：

当执行put方法插入数据时，根据key的hash值，在Segment数组中找到相应的位置，如果相应位置的Segment还未初始化，则通过CAS进行赋值，接着执行Segment对象的put方法通过加锁机制插入数据。

场景：线程A和线程B同时执行相同Segment对象的put方法

1、线程A执行tryLock()方法成功获取锁，则把HashEntry对象插入到相应的位置；

2、线程B获取锁失败，则执行scanAndLockForPut()方法，在scanAndLockForPut方法中，会通过重复执行tryLock()方法尝试获取锁，在多处理器环境下，重复次数为64，单处理器重复次数为1，当执行tryLock()方法的次数超过上限时，则执行lock()方法挂起线程B；

3、当线程A执行完插入操作时，会通过unlock()方法释放锁，接着唤醒线程B继续执行；

1.8中放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保证并发安全进行实现。此类有6000+行源码，只截取部分。

**put实现**：

public V put(K key, V value) {

return putVal(key, value, false);

}

/\*\* Implementation for put and putIfAbsent \*/

final V putVal(K key, V value, boolean onlyIfAbsent) {

if (key == null || value == null)

throw new NullPointerException();

int hash = spread(key.hashCode());

int binCount = 0;

for (Node<K,V>[] tab = table; ;) {

Node<K,V> f; int n, i, fh;

if (tab == null || (n = tab.length) == 0)

tab = initTable();

else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {

if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))

break; // no lock when adding to empty bin

} else if ((fh = f.hash) == MOVED)

tab = helpTransfer(tab, f);

else {

V oldVal = null;

synchronized (f) { // Synchronized加锁

if (tabAt(tab, i) == f) {

if (fh >= 0) {

binCount = 1;

for (Node<K,V> e = f;; ++binCount) {

K ek;

if (e.hash == hash && ((ek = e.key) == key || (ek != null && key.equals(ek)))) {

oldVal = e.val;

if (!onlyIfAbsent)

e.val = value;

break;

}

Node<K,V> pred = e;

if ((e = e.next) == null) {

pred.next = new Node<K,V>(hash, key, value, null);

break;

}

}

}

else if (f instanceof TreeBin) {

Node<K,V> p;

binCount = 2;

if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key, value)) != null) {

oldVal = p.val;

if (!onlyIfAbsent)

p.val = value;

}

}

}

}

if (binCount != 0) {

if (binCount >= TREEIFY\_THRESHOLD)

treeifyBin(tab, i);

if (oldVal != null)

return oldVal;

break;

}

}

}

addCount(1L, binCount);

return null;

}

**HashSet**

多线程下HashSet是不安全的，也会报出java.util.Concurrent Modification Exception异常，和ArrayList报的异常一样。

public static void main(String[] args) {

Set<String> set = new HashSet<>();

for (int i = 1; i <= 30; i++) {

new Thread(() -> {

set.add( Thread.currentThread().getName() );

System.out.println(set);

}, String.valueOf("T-" +i)).start();

}

}



**基础解决方式Collections.synchronizedSet**

Set<String> set = Collections.synchronizedSet(new HashSet<String>()); 和ArrayList一样，同样的配方，一样的味道。

**HashSet底层结构：HashMap**

HashSet的底层数据结构使用的是HashMap，利用HashMap的特性：Key值不允许重复来实现，Map中的value值则填充一个默认对象。

源代码如下：

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, java.io.Serializable {

static final long serialVersionUID = 5024744406713321676L;

private transient HashMap<E ,Object> map;

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object(); // 作为map结构的默认Value值

/\*\*

\* Constructs a new, empty set; the backing <tt>HashMap</tt> instance has

\* default initial capacity (16) and load factor (0.75). // 默认初始容量(16)和负载因子(0.75)，标准的HashMap。

\*/

public HashSet() {

map = new HashMap<>();

}

public boolean add(E e) {

return map.put(e , PRESENT) == null;

}

……

}

**升级解决方式：CopyOnWriteArraySet**

Set<String> set = **new** CopyOnWriteArraySet<String>();

Java中的锁【实际运用】



部分内容参考：<https://www.cnblogs.com/jyroy/p/11365935.html>

乐观锁VS悲观锁

对于同一个数据的并发操作，**悲观锁**认为自己在使用数据的时候一定有别的线程来修改数据，因此在获取数据的时候会先加锁，确保数据不会被别的线程修改。Java中，synchronized关键字和Lock的实现类都是悲观锁。**乐观锁**认为自己在使用数据时不会有别的线程修改数据，所以不会添加锁，只是在更新数据的时候去判断之前有没有别的线程更新了这个数据。如果这个数据没有被更新，当前线程将自己修改的数据成功写入。如果数据已经被其他线程更新，则根据不同的实现方式执行不同的操作（例如报错或者自动重试）。

乐观锁在Java中是通过使用无锁编程来实现，最常采用的是CAS算法，Java原子类中的递增操作就通过CAS自旋实现的。悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确；乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升。

公平锁VS非公平锁

公平锁：

是指多个线程按照申请锁的顺序来获取锁，类似排队打饭 先来后到。在并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空或者当前线程是等待队列的第一个，就占有锁；否则就会加入到等待队列中，以后会按照FIFO的规则从队列中取到自己。

非公平锁：

是指在多线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取到锁，在高并发的情况下，有可能造成优先级反转或者饥饿现象。在并发环境中，线程尝试插队，直接占有锁，如果尝试插队失败则去排队。

并发包ReentrantLock的创建可以指定构造函数的boolean类型来得到公平锁或者非公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。对于synchronized而言也是一种非公平锁。

ReentrantLock对两种锁的实现机制

源代码如下：

/\*\*

\* Creates an instance of {@code ReentrantLock}.

\* This is equivalent to using {@code ReentrantLock(false)}.

\*/

public ReentrantLock() {

sync = new NonfairSync(); // 源代码中默认构造函数采用非公平锁

}

/\*\*

\* Creates an instance of {@code ReentrantLock} with the

\* given fairness policy.

\*

\* @param fair {@code true} if this lock should use a fair ordering policy

\*/

public ReentrantLock(boolean fair) {

sync = fair ? new FairSync() : new NonfairSync(); // 重载构造函数中采用可选的锁类型，true为公平锁，false为非公平锁

}

对于**NonfairSync**方法，是一个静态的成员内部类，使用static意味着随对象创建直接初始化到方法区中：

/\*\*

\* Sync object for non-fair locks

\*/

static final class NonfairSync extends Sync {

private static final long serialVersionUID = 7316153563782823691L;

/\*\*

\* Performs lock. Try immediate barge, backing up to normal

\* acquire on failure.

\*/

final void lock() {

// 此方法位于：public abstract class AbstractQueuedSynchronizer extends AbstractOwnableSynchronizer implements java.io.Serializable

if (compareAndSetState(0, 1))

setExclusiveOwnerThread(Thread.currentThread());

else

acquire(1);

}

protected final boolean tryAcquire(int acquires) {

return nonfairTryAcquire(acquires);

}

}

protected final boolean compareAndSetState(int expect , int update) {

// See below for intrinsics setup to support this | 底层调用了Unsafe类的CAS方法

return unsafe.compareAndSwapInt(this, stateOffset, expect, update);

}

对于**FairSync**方法：

static final class FairSync extends Sync {

private static final long serialVersionUID = -3000897897090466540L;

final void lock() {

acquire(1);

}

/\*\*

\* Fair version of tryAcquire. Don't grant access unless

\* recursive call or no waiters or is first.

\*/

protected final boolean tryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState(); // AbstractQueuedSynchronizer.java中的方法，获取当前队列中线程的状态

if (c == 0) {

if (!hasQueuedPredecessors() && compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current); // 锁准备交给当前线程

return true;

}

}

else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0)

throw new Error("Maximum lock count exceeded");

setState(nextc);

return true;

}

return false;

}

}

可重入锁(又叫递归锁)

概念描述

指的是同一个线程外层函数获得锁之后，内层递归函数（或其他被调用的加锁方法）仍然能获取该锁的代码，也就是说：线程可以进入任何一个他已经拥有的锁，所同步着的代码块。简单来说：主人回家开了第一道门，房间里的其他门都可以打开；因为默认情况下谁家房间内的门都不会锁着。

**典型**：

ReentrantLock / synchronized就是一个典型的可重入锁。

**作用**：

可重入锁最大的作用就是避免死锁。

synchronized代码示例

public class SynchronizedDemo {

public synchronized void sendSms(){

System.out.println(Thread.currentThread().getName() + " 发送短信完成");

sendEmail();

}

public synchronized void sendEmail(){

System.out.println(Thread.currentThread().getName() + " 邮件发送完成");

}

}

public class TestDemo {

public static void main(String[] args) {

SynchronizedDemo sd = new SynchronizedDemo();

new Thread(() -> { sd.sendSms(); } , "T1").start();

new Thread(() -> { sd.sendSms(); } , "T2").start();

}

}



ReentrantLock代码示例

public class ReentrantLockDemo implements Runnable{

private Lock lock = new ReentrantLock();

public void run() {

sendSms();

}

private void sendSms(){

lock.lock();

try {

System.out.println(Thread.currentThread().getName() + " 发送短信完成");

this.sendEmail();

}catch (Exception e){

e.printStackTrace();

}finally {

lock.unlock();

}

}

private void sendEmail(){

lock.lock();

try {

// 线程可以进入任何一个他已经拥有锁，所同步着的代码块

System.out.println(Thread.currentThread().getName() + " 邮件发送完成");

}catch (Exception e){

e.printStackTrace();

}finally {

lock.unlock();

}

}

}

public class TestDemo {

public static void main(String[] args) {

ReentrantLockDemo rld = new ReentrantLockDemo();

Thread thread3 = new Thread(rld, "T3");

Thread thread4 = new Thread(rld, "T4");

thread3.start();

thread4.start();

}

}



ReentrantLock 2层锁解一次





这种情况比较特殊，从运行表现上可以看到，T4线程并没有打印，这是因为T3线程还在等待。如果我们加锁2次，解锁也是2次，那么这种情况就不会发生。

自旋锁 - SpinLock

定义：

是指尝试获取锁的线程不会立即阻塞（自旋的对立面即：阻塞），而是采用循环的方式去尝试获取锁；这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。最常见的实现是AtomicInteger类中的一个方法：getAndIncrement方法。



实现一个自旋锁

import java.util.concurrent.atomic.AtomicReference;

/\*\*

\* @description: 手写一个自旋锁

\*

\* @author Yangcl

\* @home https://github.com/PowerYangcl

\* @date 2020年3月16日 下午3:14:13

\* @version 1.0.0.1

\*/

public class SpinLock {

AtomicReference<Thread> atomicReference = new AtomicReference<Thread>();

public void lock() {

Thread thread = Thread.currentThread();

System.out.println(Thread.currentThread().getName() + " 尝试获取锁");

while ( !atomicReference.compareAndSet(null, thread) ) { // 期望值是空，那么则设置成自己为当前线程，否则现在已经有其他线程了

// 开始自旋等待

}

}

public void unlock() {

Thread thread = Thread.currentThread();

atomicReference.compareAndSet(thread , null); // 如果当前锁用完了，那么置空让其他线程获取锁。

System.out.println(Thread.currentThread().getName() + " 开始释放锁");

}

}

public class SpinDemo {

public static void main(String[] args) {

SpinLock lock = new SpinLock();

new Thread(() -> {

lock.lock();

try {TimeUnit.SECONDS.sleep(5);} catch (InterruptedException e) {e.printStackTrace();} // 当前锁占用5秒，模拟该线程执行任务

lock.unlock();

} , "T1").start();

try {TimeUnit.SECONDS.sleep(2);} catch (InterruptedException e) {e.printStackTrace();} // 暂停main线程2秒，确保最先执行线程T1

new Thread(() -> {

lock.lock();

lock.unlock();

} , "T2").start();

}

}



独占锁(写锁) VS 共享锁(读锁) VS 互斥锁

**独占锁**：指该锁一次只能被一个线程所持有。ReentrantLock和Synchronized都是独占锁。

**共享锁**：指该锁可被多个线程所持有。

读写锁：独占锁+共享锁

既要保持数据一致性，也要保持并发性；读写锁适合于对数据结构的读次数比写次数多得多的情况。读写锁通常一次只有一个线程可以占有写模式的读写锁，但是多个线程可以同时占有读模式的读写锁（允许多个线程读但只允许一个线程写）

**读写锁的特点**：

1、如果有其它线程读数据，则允许其它线程执行读操作，但不允许写操作。

2、如果有其它线程写数据，则其它线程都不允许读、写操作。

3、如果某线程申请了读锁，其它线程可以再申请读锁，但不能申请写锁。

4、如果某线程申请了写锁，其它线程不能申请读锁，也不能申请写锁。

ReentrantReadWriteLock，读锁是共享锁，写锁是独占锁。其读的共享锁可保证并发读是非常高效的；读写、写读和写写的过程是互斥的，只有读读能够共存。

利用Java的读写锁实现缓存的设计【实际运用】

import java.util.HashMap;

import java.util.Map;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.ReentrantReadWriteLock;

public class BaseCache {

private volatile Map<String , Object> map = new HashMap<>();

private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

public void put(String key , Object value){

lock.writeLock().lock();

try {

System.out.println(Thread.currentThread().getName() + " 正在写入：" + key + " = " + value);

try {TimeUnit.SECONDS.sleep(2);} catch (InterruptedException e) {e.printStackTrace();}

map.put(key , value);

System.out.println(Thread.currentThread().getName() + " 写入完成：" + key + " = " + value);

} catch (Exception e) {

e.printStackTrace();

}finally {

lock.writeLock().unlock();

}

}

public Object get(String key){ // 获取缓存中的值

Object result =null;

lock.readLock().lock();

try {

result = map.get(key);

System.out.println(Thread.currentThread().getName() + " 读取完成：" + key + " = " + result);

return result;

} catch (Exception e) {

e.printStackTrace();

}finally {

lock.readLock().unlock();

}

return result;

}

public void remove(String key){ } // TODO

public void clearAll(){} // TODO

}

public class ReadWriteLock {

public static void main(String[] args) {

BaseCache myCaChe = new BaseCache();

for (int i = 1; i <= 5; i++) {

final int temp = i;

new Thread(() -> { myCaChe.put(temp + "", temp); }, String.valueOf(i)).start();

}

for (int i = 6; i <= 10; i++) {

int temp = i - 5;

new Thread(() -> { myCaChe.get(temp + ""); }, String.valueOf(i)).start();

}

}

}

加锁前： 加锁后写入操作被严格控：

CountDownLatch / CyclicBarrier / Semaphore

CountDownLatch【实际运用】

倒计时，减法计数。让一些线程阻塞直到另外一些完成后才被唤醒。CountDownLatch主要有两个方法，当一个或多个线程调用await方法时，调用线程会被阻塞；其他线程调用countDown方法计数器减1(调用countDown方法时线程不会阻塞)，当计数器的值变为0时，因调用await方法被阻塞的线程会被唤醒，继续执行。模拟代码秦灭六国：

public enum CountryEnmu {

One(1, "齐" , "姜小白"),

Two(2 ,"楚" ,"楚庄王"),

Three(3, "燕","燕昭王"),

Four(4, "赵","赵武灵王"),

Five(5, "魏","魏武卒"),

Six(6, "韩","韩非子");

public static CountryEnmu foreach(int index){

CountryEnmu[] arr = CountryEnmu.values();

for (CountryEnmu ele : arr){

if(index == ele.getId()){

return ele;

}

}

return null;

}

private Integer id;

private String name;

private String remark;

CountryEnmu(Integer id, String name, String remark) {

this.id = id;

this.name = name;

this.remark = remark;

}

…… 省略getter setter方法。

}

public class CountDownLatchDemo {

public static void main(String[] args) {

CountDownLatch countDownLatch = new CountDownLatch(6); // 初始一个线程数量

for (int i = 1; i <= 6 ; i ++){

final int sleep = i;

new Thread(() -> {

try {TimeUnit.SECONDS.sleep(sleep);} catch (InterruptedException e) {e.printStackTrace();}

System.out.println(Thread.currentThread().getName() + " 国，被灭");

countDownLatch.countDown(); // 此处进行减法计数

}, CountryEnmu.foreach(i).getName() ).start();

}

new Thread(() -> {

try { countDownLatch.await(); } catch (InterruptedException e) { e.printStackTrace(); }

System.out.println(Thread.currentThread().getName() + " 一统华夏！");

}, "秦国" ).start();

new Thread(() -> {

try { countDownLatch.await(); } catch (InterruptedException e) { e.printStackTrace(); }

System.out.println(Thread.currentThread().getName() + " 再次一统华夏！");

}, "汉朝" ).start();

}

}

运行结果如下：



CyclicBarrier【实际运用】

CyclicBarrier的字面意思是可循环（Cyclic）使用的屏障（Barrier）。他要做的事情是：让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程达到屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活，线程进入屏障通过CyclicBarrier的await方法。CyclicBarrier是加法计数，与CountDownLatch的模式相反。通俗点讲，类似于人到齐了我们开会。查看其构造函数如下：

/\*\*

\* Creates a new {@code CyclicBarrier} that will trip when the given number of parties (threads) are waiting upon it, and which

\* will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.

\*

\* @param parties the number of threads that must invoke {@link #await} before the barrier is tripped

\* @param barrierAction the command to execute when the barrier is tripped , or {@code null} if there is no action

\* @throws IllegalArgumentException if {@code parties} is less than 1

\*/

public CyclicBarrier(int parties, Runnable barrierAction) { // barrierAction代表屏障被触发后，开始启动一个线程，来处理收集到的结果。

if (parties <= 0) throw new IllegalArgumentException();

this.parties = parties;

this.count = parties;

this.barrierCommand = barrierAction;

}

/\*\*

\* Creates a new {@code CyclicBarrier} that will trip when the given number of parties (threads) are waiting upon it, and

\* does not perform a predefined action when the barrier is tripped.

\*

\* @param parties the number of threads that must invoke {@link #await} before the barrier is tripped

\* @throws IllegalArgumentException if {@code parties} is less than 1

\*/

public CyclicBarrier(int parties) {

this(parties, null);

}

CyclicBarrier的构造函数有2个，但通常使用第一个。收集龙珠的简单示例如下：

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierDemo {

public static void main(String[] args) {

CyclicBarrier cyclicBarrier = new CyclicBarrier(7, () -> {

System.*out*.println("所有线程都已到达屏障，开始召唤神龙"); // 所有线程都执行完成后，此处才开始执行计算操作

});

// CyclicBarrier cyclicBarrier = new CyclicBarrier(4, new Runnable() { // 这里第一个参数写成4，但是下面是7个线程在等待，会造成线程假死。

// public void run() {

// System.out.println("所有线程都已到达屏障，开始召唤神龙");

// }

// });

for (int i = 1; i <= 7; i++) {

final int temp = i;

new Thread(() -> {

System.*out*.println(Thread.*currentThread*().getName() + " 收集到第" + temp + "颗龙珠");

try {

cyclicBarrier.await(); // 先到达屏障的，被阻塞

} catch (InterruptedException |BrokenBarrierException e) {

e.printStackTrace();

}

},"线程-" + String.*valueOf*(i) ).start();

}

}

}



Semaphore【实际运用】

semaphore又名信号量，主要用于两个目的：1 用于多个共享资源的互斥使用；2 用于并发线程数控制。有减就会有增，有增就会有减。Java中加锁的方式很多，synchronized、Lock、ReentrantReadWriteLock，第四种加锁方式就是semaphore。他的特点就是“抢车位”。前三种加锁方式的特点是多个线程抢夺同一个资源，semaphore不同之处在于多个线程抢夺多个资源。

public class SemaphoreDemo {

public static void main(String[] args) {

Semaphore semaphore = new Semaphore(5); // 模拟5个停车位

for(int i = 1 ; i <= 10 ; i ++) { // 模拟10辆车

final int temp = i \* 10;

new Thread(() -> {

try {

semaphore.acquire();

System.out.println(Thread.currentThread().getName() + " 抢到车位");

try {TimeUnit.SECONDS.sleep( temp );} catch (InterruptedException e) {e.printStackTrace();}

System.out.println(Thread.currentThread().getName() + " 停车：" + temp + "秒后离开车位");

} catch (InterruptedException e) {

e.printStackTrace();

}finally {

semaphore.release();

}

},"车主-" + String.valueOf(i) ).start();

}

}

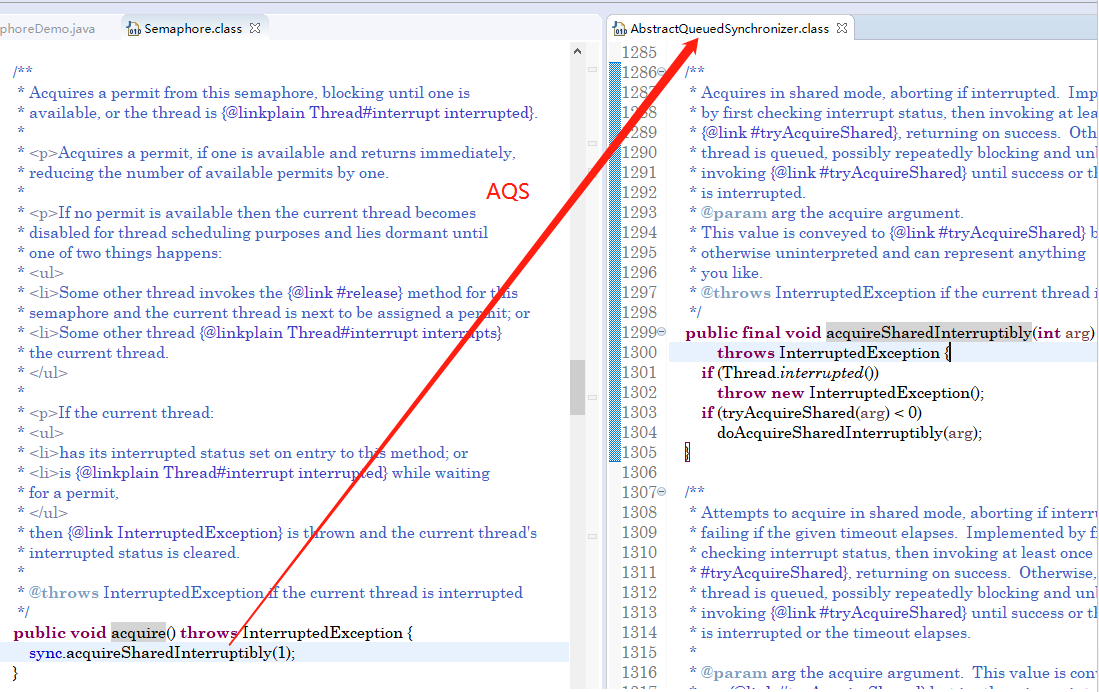
}

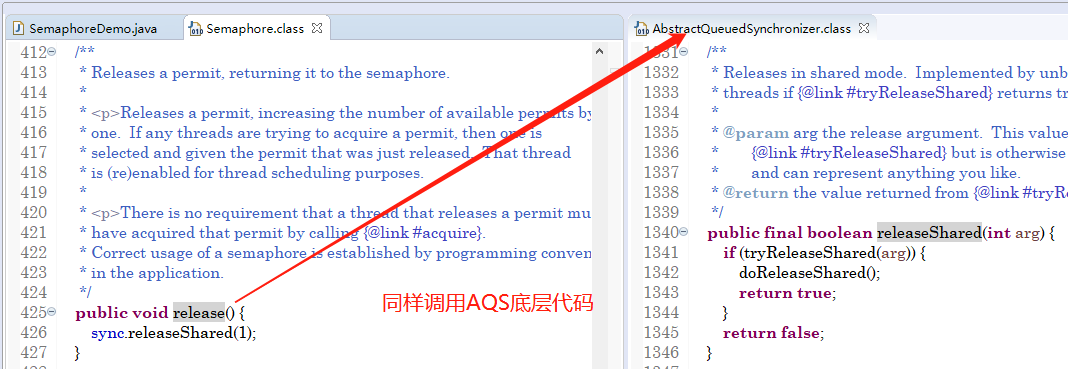
其底层实现：



对于acquire()方法，底层调用的是AQS。



对于release()方法：



BlockingQueue 阻塞队列【实际运用】

阻塞队列定义

顾名思义，它首先是一个队列，通常用在“生产者/消费者模式”中、消息队列等。一个阻塞队列在数据结构中所起的作用如下图所示：



线程1向阻塞队列中添加元素，线程2从阻塞队列中移除元素。

**特点**：

1、当阻塞队列是空时，从队列中获取元素的操作将会被阻塞。

2、当阻塞队列是满时，往队列里添加元素的操作将会被阻塞。

通俗来讲，蛋糕店买蛋糕的过程，与阻塞队列很相似。蛋糕店的柜子就是队列，厨师是生产者，做好了一个蛋糕放进柜子里，如果柜子已经满了，厨师不再做蛋糕，因为做了没地方摆放，厨师此时等待蛋糕被买走；客户就是消费者，柜子里面有蛋糕，客户付钱买走，如果柜子里面没有蛋糕，那么客户等待厨师生产蛋糕。如果等的时间太长，客户不耐烦走了，也就意味着消费者等待超时。

BlockingQueue类图结构：



标记为红色的是常用的三种队列

ArrayBlockingQueue：由数组结构组成的有界阻塞队列。

LinkedBlockingQueue：由链表结构组成的有界(但大小默认值Integer>MAX\_VALUE，是个巨大的隐患)阻塞队列。

SynchronousQueue：不存储元素的阻塞队列,也即是单个元素的队列。

PriorityBlockingQueue：支持优先级排序的无界阻塞队列。

DelayQueue：使用优先级队列实现的延迟无界阻塞队列。

LinkedTransferQueue：由链表结构组成的无界阻塞队列。

LinkedBlockingDeque：由链表结构组成的双向阻塞队列。

重点！为什么需要BlockingQueue?

在多线程领域，所谓阻塞：指在某些情况下会挂起线程（即阻塞），一旦条件满足，被挂起的线程又会自动被唤醒；在concurrent包发布以前，在多线程环境下，我们每个程序员都必须自己去控制这些细节，尤其还要兼顾效率和线程安全，而这会给我们的程序带来不小的复杂度。由于这些行为的控制需要开发人员进行人为代码干预，换句话说这相当于汽车是“手动挡的”。但当我们引入了BlockingQueue以后，我们不用再关心什么时候需要阻塞线程，什么时候需要唤醒线程，这一切BlockingQueue都一手包办了；以后开车就变成了自动挡了，你只需要关心油门刹车就行了。

BlockingQueue可以看做是除synchronized、Lock、ReentrantReadWriteLock和semaphore之外的第五种加锁方式。

BlockingQueue核心方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 方法类型 | 抛出异常 | 特殊值 | 阻塞 | 超时 |
| 插入 | add(e) | offer(e) | put(e) | offer(e , time , unit) |
|  |  |  | 返回空 |  |
| 移除 | remove() | poll() | take() | poll(time , unit) |
|  | 返回移除的对象 |  |  |  |
| 检查 | element() | peek() | 不可用 | 不可用 |
|  | 队列是否为空队首元素是谁 | 同左 |  |  |

**抛出异常**：当阻塞队列满时，再往队列里面add插入元素会抛IllegalStateException: Queue full当阻塞队列空时，再往队列Remove元素时候回抛出NoSuchElementException。

**特 殊 值**：插入方法,成功返回true 失败返回false。移除方法，成功返回元素，队列里面没有就返回null。

**一直阻塞**：当阻塞队列满时，生产者继续往队列里面put元素，队列会一直阻塞直到put数据or响应中断退出当阻塞队列空时，消费者试图从队列take元素，队列会一直阻塞消费者线程直到队列可用。

**超时退出**：当阻塞队列满时，队列会阻塞生产者线程一定时间，超过后限时后生产者线程就会退出。

ArrayBlockingQueue代码演示

抛出异常：脾气比较火爆

**public class** BlockingQueueDemo {

**public static void** main(String[] args) {

BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3);

System.out.println(blockingQueue.add("a"));

System.out.println(blockingQueue.add("b"));

System.out.println(blockingQueue.add("c"));

System.out.println(blockingQueue.add("d")); // 队列长度为3，加入第四个元素时抛出异常

}

}



**public class** BlockingQueueDemo {

**public static void** main(String[] args) {

BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3);

System.out.println(blockingQueue.add("a"));

System.out.println(blockingQueue.add("b"));

System.out.println(blockingQueue.add("c"));

System.out.println(blockingQueue.remove()); // 返回被移除的对象a

System.out.println(blockingQueue.remove()); // 返回被移除的对象b

System.out.println(blockingQueue.remove()); // 返回被移除的对象c

System.out.println(blockingQueue.remove()); // 队列中共三个对象，当移除第四个的时候会抛出异常

}

}



特 殊 值

**public class** BlockingQueueDemo {

**public static void** main(String[] args) {

BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3);

System.out.println("插入第1个元素，状态：" + blockingQueue.offer("a"));

System.out.println("插入第2个元素，状态：" + blockingQueue.offer("b"));

System.out.println("插入第3个元素，状态：" + blockingQueue.offer("c"));

System.out.println("插入第4个元素，状态：" + blockingQueue.offer("d"));

System.out.println("开始移除元素：" + blockingQueue.poll()); // 返回被移除的对象a

System.out.println("开始移除元素：" + blockingQueue.poll()); // 返回被移除的对象b

System.out.println("开始移除元素：" + blockingQueue.poll()); // 返回被移除的对象c

System.out.println("检查队列是否为空，同时返回队首元素：" + blockingQueue.peek());

System.out.println("继续移除第四个元素，返回：" + blockingQueue.poll()); // 队列中共三个对象，当移除第四个的时候返回null

}

}



一直阻塞

这组API威力强大，在代码中应当慎用。用的好，程序性能极高，用的不好bug也会很致命。官网使用的这组方法做演示。

**public class** BlockingQueueDemo {

**public static void** main(String[] args) {

BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3);

**try** {

blockingQueue.put("a"); // 返回void，sysout无法打印

blockingQueue.put("b");

blockingQueue.put("c");

// blockingQueue.put("d"); // 此时会一直阻塞

System.out.println("开始移除元素：" + blockingQueue.take()); // 返回被移除的对象a

System.out.println("开始移除元素：" + blockingQueue.take()); // 返回被移除的对象b

System.out.println("开始移除元素：" + blockingQueue.take()); // 返回被移除的对象c

System.out.println("继续移除第四个元素，返回：" + blockingQueue.take());

// 队列中共三个对象，当移除第四个的时候一直阻塞，知道队列中有第四个元素进入

} **catch** (InterruptedException e) {e.printStackTrace();}

}

}



超时退出

比阻塞更加合理的一种解决方式。

**public class** BlockingQueueDemo {

**public static void** main(String[] args) {

BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3);

**try** {

System.out.println("插入第1个元素，状态：" + blockingQueue.offer("a" , 2L , TimeUnit.SECONDS));

System.out.println("插入第2个元素，状态：" + blockingQueue.offer("b" , 2L , TimeUnit.SECONDS));

System.out.println("插入第3个元素，状态：" + blockingQueue.offer("c" , 2L , TimeUnit.SECONDS));

System.out.println("插入第4个元素，状态：" + blockingQueue.offer("d" , 2L , TimeUnit.SECONDS)); // 2秒后无法进入队列则返回false

System.out.println("开始移除元素：" + blockingQueue.poll(2L , TimeUnit.SECONDS)); // 返回被移除的对象a

System.out.println("开始移除元素：" + blockingQueue.poll(2L , TimeUnit.SECONDS)); // 返回被移除的对象b

System.out.println("开始移除元素：" + blockingQueue.poll(2L , TimeUnit.SECONDS)); // 返回被移除的对象c

System.out.println("继续移除第四个元素，返回：" + blockingQueue.poll(2L , TimeUnit.SECONDS)); // 2秒后断开

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}



SynchronousQueue代码演示

SynchronousQueue没有容量，与其他的BlockingQueue不同，他不存储元素。每个put操作必须要等待一个take操作，否则不能继续添加元素，反之亦然。

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.SynchronousQueue;

**import** java.util.concurrent.TimeUnit;

**public** **class** SynchronousQueueDemo {

**public static void** main(String[] args) {

BlockingQueue<String> blockingQueue = **new** SynchronousQueue<>();

// 模拟向队列中放入数据

**new** Thread(() -> {

**try** {

System.out.println(Thread.currentThread().getName() + " 放入 1");

blockingQueue.put("1");

System.out.println(Thread.currentThread().getName() + " 放入 2");

blockingQueue.put("2");

System.out.println(Thread.currentThread().getName() + " 放入 3");

blockingQueue.put("3");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}, "线程-A").start();

// 线程ABC模拟从队列中取数据

**new** Thread(() -> {

**try** {

TimeUnit.SECONDS.sleep(5);

System.out.println(Thread.currentThread().getName() + " 获取" + blockingQueue.take());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}, "线程-B").start();

**new** Thread(() -> {

**try** {

TimeUnit.SECONDS.sleep(5);

System.out.println(Thread.currentThread().getName() + " 获取" + blockingQueue.take());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}, "线程-C").start();

**new** Thread(() -> {

**try** {

TimeUnit.SECONDS.sleep(5);

System.out.println(Thread.currentThread().getName() + " 获取" + blockingQueue.take());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}, "线程-D").start();

}

}



**生产者/消费者模式**

**多线程交互的口诀**：

**线程**（多个外部线程） **操作**（具体到方法）**资源类**（本身提供被多个线程操作的基础方法）；

**判断 干活 唤醒通知**；严防多线程并发状态下的虚假唤醒。

**传统版：Lock-await-singal**



不再使用Synchronized的方式，这里以Lock的方式来演示。代码将演示一个初始值为0的变量，两个线程对其交替操作，一个加1，一个减1；循环5次。

**源码示例与分析**

**import** java.util.concurrent.locks.Condition;

**import** java.util.concurrent.locks.Lock;

**import** java.util.concurrent.locks.ReentrantLock;

/\*\*

\* @**description**: ShareData.java即为【资源类】

\*

\* @**author** Yangcl

\* @**home** https://github.com/PowerYangcl

\* @**date** 2020年3月19日 下午3:24:35

\* @**version** 1.0.0.1

\*/

**public class** ShareData {

**private int** number = 0;

**private** Lock lock = **new** ReentrantLock();

**private** Condition condition = lock.newCondition();

**public void** increment() {

lock.lock();

**try** {

// 1、判断

**while**(number != 0) { // while防止虚假唤醒。if判断的话多余2个线程就会出错

condition.await(); // 等待，不能生产。调用await方法后操作，线程的锁被“原子级地”释放

}

// 2、干活

number ++;

System.out.println(Thread.currentThread().getName() + " 增加 number = " + number);

// 3、通知唤醒

condition.signalAll();

} **catch** (InterruptedException e) {

e.printStackTrace();

}**finally**{

lock.unlock();

}

}

**public void** decrement() {

lock.lock();

**try** {

// 1、判断

**while**(number == 0) { // while防止虚假唤醒。if判断的话多余2个线程就会出错

condition.await(); // 等待，不能生产。调用await方法后操作，线程的锁被“原子级地”释放

}

// 2、干活

number --;

System.out.println(Thread.currentThread().getName() + " 消耗 number = " + number);

// 3、通知唤醒

condition.signalAll();

} **catch** (InterruptedException e) {

e.printStackTrace();

}**finally**{

lock.unlock();

}

}

}

**public class** ShareDateMain {

**public static void** main(String[] args) {

ShareData shareData = new ShareData();

**new** Thread(() -> {

**for**(**int** i = 1 ; i <= 5 ; i ++) {

shareData.increment();

}

}, "线程-A").start();

**new** Thread(() -> {

**for**(**int** i = 1 ; i <= 5 ; i ++) {

shareData.decrement();

}

}, "线程-B").start();

}

}

详细图解如下：



调用await方法后操作，线程的锁被“原子级地”释放。



此时，线程A执行完成后，你会发现线程B调用的decrement()方法会被重新执行：

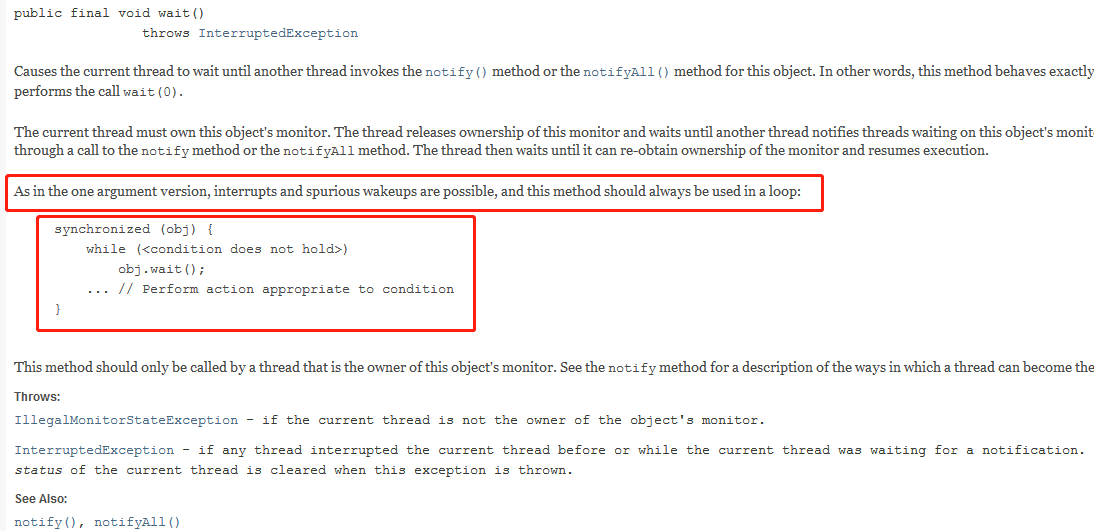


最终运行结果：



**虚假唤醒**

使用while，不可以用if。jdk官方文档中的解释和示例均定义使用while。比如对于Object类的wait方法：



**阻塞队列版**

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.TimeUnit;

**import** java.util.concurrent.atomic.AtomicInteger;

**import** org.apache.commons.lang3.StringUtils;

**public** **class** BlockingQueueResource {

**private** **volatile** **boolean** flag = **true**; // 生产者与消费者的方法运行标识，同生共死，默认开启

**private** AtomicInteger atomicInteger = **new** AtomicInteger(); // 默认值是0

**private** BlockingQueue<String> blockingQueue = **null**; // 模拟放蛋糕的柜子

**public** BlockingQueueResource(BlockingQueue<String> blockingQueue) {

**this**.blockingQueue = blockingQueue;

}

**public** **void** producer() {

String producer = "";

**boolean** offer;

String name = Thread.currentThread().getName();

**try** {

**while**(flag) { // 准备执行生产活动

producer = atomicInteger.incrementAndGet() + "";

offer = blockingQueue.offer(producer, 1L, TimeUnit.SECONDS);

**if**(offer) {

System.err.println(name + " 做完了第【" + producer + "】个蛋糕，放入柜台成功");

}**else** {

System.err.println(name + " 做完了第【" + producer + "】个蛋糕，放入柜台失败！！！！！");

}

TimeUnit.SECONDS.sleep(1);

}

System.err.println(name + " producer 生产蛋糕的行为已经停止");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**public** **void** consumer() {

String result = "";

String name = Thread.currentThread().getName();

**try** {

**while**(flag) {

result = blockingQueue.poll(3L, TimeUnit.SECONDS);

**if**(StringUtils.isBlank(result)) {

System.out.println(name +" 排队超过2m没有取到蛋糕，不再等待去别的店买了");

**return**;

}

System.out.println(name + " 从柜台中买走了第【" + result + "】个蛋糕");

}

System.out.println("蛋糕店不做蛋糕了，" + name + " 已经离开。");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**public** **void** stop() {

flag = **false**;

System.out.println("时间到，活动停止。当前柜台里还有：" + blockingQueue.size() + "个蛋糕");

}

}

**public** **class** BlockingQueueResourceMain {

**public** **static** **void** main(String[] args) {

BlockingQueueResource resource = **new** BlockingQueueResource(**new** ArrayBlockingQueue<String>(10));

**new** Thread(() -> {resource.producer();}, "厨子巴扎黑").start();

**new** Thread(() -> {resource.producer();}, "厨子傻憨黑").start();

**new** Thread(() -> {resource.consumer();}, "狸花猫吾皇").start();

**for**(**int** i = 1 ; i < 3 ; i ++) {

**new** Thread(() -> {resource.consumer();}, "买蛋糕路人-" + i).start();

}

// 设置运行5秒钟

**try** { TimeUnit.SECONDS.sleep(5); } **catch** (InterruptedException e) { e.printStackTrace(); }

resource.stop();

}

}



**顺序调用与精确唤醒问题**

多线程之间按照顺序调用，实现A->B->C三个线程顺序启动，要求如下：线程A打印1次，线程B打印2次，线程C打印3次，循环2次。

**import** java.util.concurrent.locks.Condition;

**import** java.util.concurrent.locks.Lock;

**import** java.util.concurrent.locks.ReentrantLock;

**public class** ShareData {

**private int** number = 1; // 1/2/3分别代表ABC三个线程。

**private** Lock lock = **new** ReentrantLock();

**private** Condition conditionA = lock.newCondition(); // 实现精确唤醒

**private** Condition conditionB = lock.newCondition(); // 实现精确唤醒

**private** Condition conditionC = lock.newCondition(); // 实现精确唤醒

**public void** printOne() {

lock.lock();

**try** {

// 1、判断

**while**(number != 1) {

conditionA.await();

}

// 2、干活

System.out.println(Thread.currentThread().getName() + " number = " + number);

// 3、通知唤醒

number = 2;

conditionB.signal();

} **catch** (InterruptedException e) {

e.printStackTrace();

}**finally**{

lock.unlock();

}

}

**public void** printTwo() {

lock.lock();

**try** {

// 1、判断

**while**(number != 2) {

conditionB.await();

}

// 2、干活

System.out.println(Thread.currentThread().getName() + " number = " + number);

System.out.println(Thread.currentThread().getName() + " number = " + number);

// 3、通知唤醒

number = 3;

conditionC.signal();

} **catch** (InterruptedException e) {

e.printStackTrace();

}**finally**{

lock.unlock();

}

}

**public void** printThree() {

lock.lock();

**try** {

// 1、判断

**while**(number != 3) {

conditionC.await();

}

// 2、干活

System.out.println(Thread.currentThread().getName() + " number = " + number);

System.out.println(Thread.currentThread().getName() + " number = " + number);

System.out.println(Thread.currentThread().getName() + " number = " + number);

// 3、通知唤醒

number = 1;

conditionA.signal();

} **catch** (InterruptedException e) {

e.printStackTrace();

}**finally**{

lock.unlock();

}

}

}

**public class** ShareDateMain {

**public static void** main(String[] args) {

ShareData shareData = **new** ShareData();

**new** Thread(() -> {

**for**(**int** i = 1 ; i <= 5 ; i ++) {

shareData.printOne();

}

}, "线程-A").start();

**new** Thread(() -> {

**for**(**int** i = 1 ; i <= 5 ; i ++) {

shareData.printTwo();

}

}, "线程-B").start();

**new** Thread(() -> {

**for**(**int** i = 1 ; i <= 5 ; i ++) {

shareData.printThree();

}

}, "线程-C").start();

}

}



线程池实战

线程池基础知识点

特点与优势

线程池做的工作主要是控制运行的线程的数量，处理过程中将任务加入队列，然后在线程创建后启动这些任务，如果先生超过了最大数量，超出的数量的线程排队等候，等其他线程执行完毕，再从队列中取出任务来执行。他的主要**特点**为：线程复用、控制最大并发数、管理线程。

第一：降低资源消耗。通过重复利用自己创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务不需要等待线程创建，可以立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配，调优和监控。

线程数量的合理配置

通过Runtime.getRuntime().availableProcessors(); 获取当前服务器的CPU核心数。

**CPU密集型**：

任务需要大量运算，且没有阻塞，CPU一直全速运行。CPU密集型任务只有在真正的多核心CPU服务器上才可能得到加速（通过多线程），而在单核CPU上无论开启几个模拟的多线程任务，都不可能得到加速，因为CPU总的运算能力就那些。

CPU密集型任务配置尽量少的线程数量，一般公式：CPU核心数+1。

**IO密集型**：

即该任务需要大量的阻塞（读取数据库、读取文件、访问其他Http接口或Rpc接口、访问Redis等等）。在单线程上运行IO密集型任务，会导致浪费大量的CPU运算能力（因为一直在等待计算资源就绪）。所以在IO密集型任务中，使用多线程可以大大加速程序运行；即使在单核CPU上，这种加速主要就是利用了被浪费掉的阻塞时间。

市面上通常有两种主流的配置方式（这两种配置方式需要根据自身具体业务，在生产环境中不断验证和测试，从而找到最适合自己的系统的方式）：

**第一种**：由于IO密集型任务线程并不是一直在执行任务，则应该配置尽可能多的线程，如：CPU核心数 \* 2。

**第二种**：CPU核心数 / (1 - 阻塞系数) 阻塞系数在0.8 ~ 0.9之间。比如8核CPU：8 / (1 – 0.9) = 80个线程数。

架构实现



Java中的线程池是通过Executor框架实现，该框架中用到了Executor，Executors，ExecutorService，ThreadPoolExecutor这几个类。

线程池底层工作原理



【1】在创建了线程池后，等待提交过来的任务请求。

【2】当调用execute()添加一个任务时，线程池做如下判断：

2.1 如果正在运行的线程数量小于corePoolSize，那么 马上创建线程运行这个任务。

2.2 如果正在运行的线程数量大于等于corePoolSize，那么将这个任务放入队列。

2.3 如果这时候队列满了，且正在运行的线程数量还小于maximumPoolSize，那么还是要创建非核心线程，立刻运行这个任务。

2.4 如果队列满了，且正在运行的线程数量大于或等于maximumPoolSize，那么线程池会启动饱和拒绝策略来执行。

【3】当一个线程完成任务时，他会从队列中取下一个任务来执行。

【4】当一个线程无事可做，超时一定时间(keepAliveTime)时，线程池会判断：如果当前运行的线程数大于corePoolSize，那么这个线程就会被停掉。

【5】线程池的所有任务完成后，他最终会收缩到corePoolSize的大小。

简单来说：线程任务进入以后，**1**先使用核心线程来处理；**2**核心线程数量达到阈值后，多余的线程任务进入到BlockingQueue阻塞队列中等等；**3**阻塞队列也满了，此时最大线程数开始创建线程，运行任务；**4**最大线程数也达到阈值，线程池开始启动饱和拒绝策略。

JDK1.8提供的默认线程池

JDK1.8原生提供的线程池是由Executors工具类封装好的，但在实际开发中通常都不会直接使用，因为“线程数量的合理配置”需要人为去判断；Executors提供的默认封装并不合理，但他们底层都调用了ThreadPoolExecutor.java，通常在实际开发中，我们会再次封装ThreadPoolExecutor.java形成适合自己业务的线程池工具类，而且通常会以如下5中线程池的实现思想作为参考和依据。

Executors.newCachedThreadPool();

适用于执行很多短期异步小程序或者负载较轻的服务器，其源码如下：

**public static** ExecutorService newCachedThreadPool() {

**return new** ThreadPoolExecutor(

0, // int corePoolSize, 线程池核心线程数

Integer.MAX\_VALUE, // int maximumPoolSize, 线程池最大数

60L, // keepAliveTime, 空闲线程存活时间

TimeUnit.SECONDS, // TimeUnit unit, 时间单位

new SynchronousQueue<Runnable>()); // 线程池所使用的缓冲队列**没有容量**！

}

它使用的是SynchronousQueue，SynchronousQueue没有容量，与其他的BlockingQueue不同，他不存储元素。也就是说来了任务就创建线程运行，线程不会进入队列，如果线程空闲超过60秒，就销毁线程。但他的最大线程数是21亿，这个设置极不合理，当大量的线程进入后，极易造成OOM。

Executors.newSingleThreadExecutor();

创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务都按照指定顺序执行。



Executors.newFixedThreadPool(10);

适合执行一个长期的多线程任务，其性能要好很多。



Executors. newScheduledThreadPool(int corePoolSize);

有计划性的线程池，就是在给定的延迟之后运行，或周期性地执行。用的不多，介绍略过。

Executors.newWorkStealingPool()

JDK1.8 版本加入的一种线程池，stealing 翻译为抢断、窃取的意思，它实现的一个线程池和上面4种都不一样，用的是 ForkJoinPool 类。它是一个并行的线程池，参数中传入的是一个线程并发的数量，这里和之前就有很明显的区别，前面4种线程池都有核心线程数、最大线程数等等，而newWorkStealingPool使用了一个并发线程数解决问题。从介绍中来看，此线程池不会保证任务的顺序执行，底层也没有再使用队列，也就是 WorkStealing 的意思，抢占式的工作。

由于能够合理的使用CPU进行对任务操作（并行操作），所以适用于很耗时的任务中。

**public static** ExecutorService newWorkStealingPool() {

**return new** ForkJoinPool(

Runtime.getRuntime().availableProcessors(), // 获取当前服务器核心数量

ForkJoinPool.defaultForkJoinWorkerThreadFactory,

**null**,

**true**);

}

线程池结合Callable接口使用示例

多线程任务执行入口类

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**import** com.alibaba.fastjson.JSONObject;

**import** com.matrix.annotation.Inject;

**import** com.matrix.base.BaseClass;

**import** com.matrix.dao.ITcLandedPropertyMapper;

**import** com.matrix.dao.IUserDemoMapper;

// 通常一个APP的首页会调用很多不相干的信息

**public class** AppHomePageService extends BaseClass{

@Inject // 模拟要访问的数据库

**private** ITcLandedPropertyMapper tcLandedPropertyMapper;

@Inject // 模拟要访问的数据库

**private** IUserDemoMapper userDemoMapper;

**public** JSONObject homePageInfo(JSONObject dto) {

JSONObject result = **new** JSONObject();

CountDownLatch countDownLatch = **new** CountDownLatch(3); // 初始一个线程数量，这里启动三个线程作为示例

ExecutorService executor = Executors.newCachedThreadPool();

Task1001Position callable1 = **new** Task1001Position("lat", "lng" , countDownLatch);

Future<JSONObject> callable1Task = executor.submit(callable1);

TaskUserDemoTop5 callable2 = **new** TaskUserDemoTop5(userDemoMapper , countDownLatch);

Future<JSONObject> callable2Task = executor.submit(callable2);

TaskLandedPropertyPriceTop5 callable3 = **new** TaskLandedPropertyPriceTop5(tcLandedPropertyMapper , countDownLatch);

Future<JSONObject> callable3Task = executor.submit(callable3);

**try** {

JSONObject json = callable1Task.get(15, TimeUnit.SECONDS); // 15秒后超时，TimeoutException。如果不用countDownLatch，这也是一种选择。

System.out.println("json = " + json);

countDownLatch.await();

result.put("callable1Task", callable1Task.get().getString("data"));

result.put("callable2Task", callable2Task.get().getString("data"));

result.put("callable3Task", callable3Task.get().getString("data"));

} **catch** (InterruptedException | ExecutionException | TimeoutException e) { // 一旦出现异常，则所有的线程任务都将被终止，无论执行完成与否

e.printStackTrace();

}**finally** {

executor.shutdown(); // 执行完成务必结束

}

System.out.println(result.toJSONString());

**return** result;

}

}

三个任务类：

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.TimeUnit;

**import** com.alibaba.fastjson.JSONObject;

/\*\*

\* @**description**: Task[表示这是一个多线程任务] 1001[表示调用了第三方的1001接口] Position[表示位置信息]

\*

\* @**author** Yangcl

\* @**home** https://github.com/PowerYangcl

\* @**version** 1.0.0.1

\*/

**public class** Task1001Position **implements** Callable<JSONObject> {

**private** CountDownLatch countDownLatch;

**private** String lat = "";

**private** String lng = "";

**public** Task1001Position(String lat, String lng , CountDownLatch countDownLatch) {

**this**.countDownLatch = countDownLatch;

**this**.lat = lat;

**this**.lng = lng;

}

@Override

**public** JSONObject call() **throws** Exception {

JSONObject result = **new** JSONObject();

// TODO 调用第三方接口，获取位置信息

**this**.lat = "123899012.123";

**this**.lng = "324124.213123";

**try** { TimeUnit.SECONDS.sleep(10); } **catch** (InterruptedException e) { e.printStackTrace(); } // 模拟查库耗时10秒

result.put("data", "北京市通州区梨园镇天使之城小区");

**this**.countDownLatch.countDown(); // 计数器减一

**return** result;

}

}

// 模拟地产信息前5名

**public class** TaskLandedPropertyPriceTop5 **implements** Callable<JSONObject> {

**private** CountDownLatch countDownLatch;

**private** ITcLandedPropertyMapper tcLandedPropertyMapper;

**public** TaskLandedPropertyPriceTop5(ITcLandedPropertyMapper tcLandedPropertyMapper , CountDownLatch countDownLatch) {

**this**.countDownLatch = countDownLatch;

**this**.tcLandedPropertyMapper = tcLandedPropertyMapper;

}

@Override

**public** JSONObject call() **throws** Exception {

JSONObject result = **new** JSONObject();

**try** { TimeUnit.SECONDS.sleep(12); } **catch** (InterruptedException e) { e.printStackTrace(); } // 模拟查库耗时12秒

result.put("data", "地产信息前5名接口查库耗时2秒");

**this**.countDownLatch.countDown(); // 计数器减一

**return** result;

}

}

// 模拟活跃用户前5名

**public class** TaskUserDemoTop5 **implements** Callable<JSONObject> {

**private** CountDownLatch countDownLatch;

**private** IUserDemoMapper userDemoMapper;

**public** TaskUserDemoTop5(IUserDemoMapper userDemoMapper , CountDownLatch countDownLatch) {

**this**.countDownLatch = countDownLatch;

**this**.userDemoMapper = userDemoMapper;

}

@Override

**public** JSONObject call() **throws** Exception {

JSONObject result = **new** JSONObject();

**try** { TimeUnit.SECONDS.sleep(15); } **catch** (InterruptedException e) { e.printStackTrace(); } // 模拟查库耗时15秒

result.put("data", "活跃用户前5名接口查库耗时3秒");

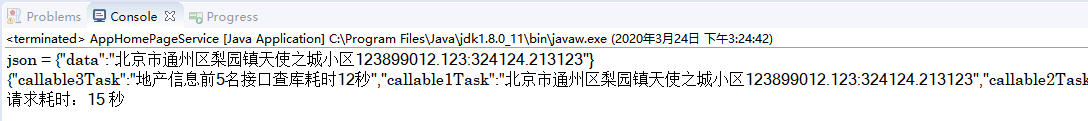
**this**.countDownLatch.countDown(); // 计数器减一

**return** result;

}

}

运行结果如下：



CompletableFuture

对于简单的业务场景使用Future完全没有，但是想将多个异步任务的计算结果组合起来，后一个异步任务的计算结果需要前一个异步任务的值等等，使用Future提供的那点API就囊中羞涩，处理起来不够优雅，这时候还是让CompletableFuture以声明式的方式优雅的处理这些需求。而且在Future编程中想要拿到Future的值然后拿这个值去做后续的计算任务，只能通过轮询的方式去判断任务是否完成这样非常占CPU并且代码也不优雅

<https://www.toutiao.com/i6779504565172568590/>

<https://www.jianshu.com/p/b3c4dd85901e>

<https://www.jianshu.com/p/6bac52527ca4>

synchronized-线程锁

synchronized锁的膨胀过程

也叫升级过程。理解此问题，首先要清楚偏向锁，轻量级锁和重量级锁。Java SE 1.6中为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁。Synchronized的升级顺序是：无锁→偏向锁→轻量级锁→重量级锁，顺序不可逆。

偏向锁

当一个线程访问同步代码块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，偏向锁是一个可重入的锁，以后该线程在进入和退出该同步代码块时不需要花费 CAS 操作来加锁和解锁，而只需简单的测试一下对象头的 Mark Word 里是否存储着指向当前线程的偏向锁(当前线程的线程ID)，如果测试成功，表示线程已经获得了锁，如果测试失败，则需要再测试下 Mark Word 中偏向锁的标识是否设置成 1（表示当前是偏向锁），如果偏向锁标识是1，则使用 CAS 进行锁获取，偏向锁标识不是1，则尝试使用CAS 将对象头的偏向锁指向当前线程，上述两种CAS获取锁的操作，如果CAS操作成功则获取到了偏向锁，失败则代表出现了锁竞争，需要**锁撤销**操作。

锁撤销

偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。偏向锁的撤销需要等待拥有偏向锁的线程到达全局安全点（在这个时间点上没有字节码正在执行），会首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否活着，如果线程不处于活动状态，则将锁的对象的对象头设置成无锁状态，如果线程仍然活着，拥有偏向锁的栈会被执行(判断是否需要持有锁)，遍历偏向对象的锁记录，查看使用情况，如果还需要持有偏向锁，则偏向锁升级为**轻量级锁**，如果不需要持有偏向锁了，则将锁对象恢复成无锁状态，最后唤醒暂停的线程。

轻量级锁

线程在执行同步块之前，JVM 会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的 Mark Word 复制到

锁记录中，官方称为 Displaced Mark Word。然后线程尝试使用 CAS 将对象头中的 Mark Word 替换为指向锁记录的指针。

如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁，自旋有一定次数，如果超过

设置自旋的次数则升级到重量级锁，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁升级为重量级锁，重量级

锁使除了拥有锁的线程以外的线程都阻塞，防止CPU空转。

重量级锁

synchronized是非公平锁，Synchronized在线程进入阻塞队列时，等待的线程会先尝试获取锁，如果获取不到就进入阻塞队列，这明显对于已经进入队列的线程是不公平的。轻量级锁膨胀之后，就升级为重量级锁了。重量级锁是依赖对象内部的monitor锁来实现的，而monitor又依赖操作系统的MutexLock(互斥锁)来实现的，所以重量级锁也被成为互斥锁。

*Mutex：/mu' tæ' ks/ n. 互斥；互斥元，互斥体；互斥量.*

* 为什么说重量级锁开销大?

因为此时需要系统内核状态切换。当系统检查到锁是重量级锁之后，会把等待想要获得锁的线程进行阻塞，被阻塞的线程不会消耗cup。但是阻塞或者唤醒一个线程时，都需要操作系统来帮忙，这就需要从用户态转换到内核态，而转换状态是需要消耗很多时间的，有可能比用户执行代码的时间还要长。这就是说为什么重量级线程开销很大的。

Java对象头

synchronized的锁是存在对象头里的，对象头由两部分数据组成：Mark Word（标记字段）、Klass Pointer（类型指针）Mark Word存储了对象自身运行时数据，如hashcode、GC分代年龄、锁状态标志、线程持有的锁、偏向锁ID等等。是实现轻量级锁和偏向锁的关键，Klass Pointer是Java对象指向类元数据的指针，jvm通过这个指针确定这个对象是哪个类的实例。

* About klass

Klass is the internal representation of a Java class (let's say A) & it will hold the basic information about the structure of the

class including the bytecode. It will be stored as an object itself.

monitor

每个Java对象从娘胎里出来就带着一把看不见的锁，叫做内部锁或者monitor锁，我们可以把它理解成一种同步机制，它是线程私有的数据结构。monitor的结构如下：

* Owner：

初始时为NULL表示当前没有任何线程拥有该monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为NULL；

* EntryQ:

关联一个系统互斥锁（semaphore），阻塞所有试图锁住monitor record失败的线程。

* RcThis:

表示blocked或waiting在该monitor record上的所有线程的个数。

* Nest(英 /nest/ n. 巢，窝；安乐窝；温床):

用来实现重入锁的计数。

* HashCode:

保存从对象头拷贝过来的HashCode值（可能还包含GC age）。

* Candidate(英 / kæn dɪ deɪt/ 候补者):

用来避免不必要的阻塞或等待线程唤醒，因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换（从阻塞到就绪然后因为竞争锁失败又被阻塞）从而导致性能严重下降。Candidate只有两种可能的值0表示没有需要唤醒的线程1表示要唤醒一个继任线程来竞争锁。

优缺点

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 场景 |
| 偏向锁 | 加解锁不需要过多的资源消耗，和非同步方法的相比仅仅是纳秒的差距 | 如果存在所竞争，会有额外的锁撤销操作 | 适用于只有一个线程访问的场景 |
|  |  |  |  |
| 轻量级锁 | 竞争线程不会阻塞，会自旋，减少了上线文切换 | 如果始终得不到锁，会消耗cpu资源 | 追求响应时间，同步代码块多为计算、执行快的场景 |
|  |  |  |  |
| 重量级锁 | 没优点 | 线程阻塞响应时间慢 | 同步代码块执行时间较长的场景使用 |
|  |  |  |  |

Java Virtual Machine实际应用

JVM体系结构

Java的内存区域主要分为五部分(此节参考：<https://www.cnblogs.com/duanxz/p/3732300.html>，文章写的极好。)：

1. 程序计数器(PC)。
2. Java 虚拟机栈(JVM Stack)。
3. 本地方法栈(Native Method Stack)。
4. Java 堆内存(Java Heap)。
5. 方法区(Method Area)。

按照功能：



按照内存是否共享：



类装载器（ClassLoader）（用来装载.class文件）。执行引擎（执行字节码，或者执行本地方法）。运行时数据区（方法区、堆、java栈、PC寄存器、本地方法栈），这其中：方法区和堆是GC的主要作用区域，如下图。



下面我们来解析这几个区域。

1、程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的信号指示器。每一条JVM线程都有自己的PC寄存器在任意时刻，一条JVM线程只会执行一个方法的代码。该方法称为该线程的当前方法（Current Method）如果该方法是java方法，那PC寄存器保存JVM正在执行的字节码指令的地址，如果该方法是native，那PC寄存器的值是undefined。

此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

2、Java 虚拟机栈

Java虚拟机栈也是线程私有的，每一条线程都拥有自己私有的Java虚拟机栈，它与线程同时创建。并且生命周期与线程相同。它描述了Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至完成的过程，都对应一个栈帧从入栈到出栈的过程。关于栈帧详细的内容在后边复习虚拟机字节码执行引擎的时候再说吧。

Java 虚拟机栈在方法调用和返回中也扮演了很重要的角色。因为除了栈帧的入栈和出栈之外，Java虚拟机栈不会再受其它因素的影响，因此栈帧可在系统的堆上分配内存（注意，是系统的Heap而不是Java Heap）。Java虚拟机栈所使用的内存不需要保证是连续的。

3、本地方法栈

Java可以通过java本地接口JNI（Java Native Interface）来调用其它语言编写（如C）的程序，在Java里面用native修饰符来描述一个方法是本地方法。本地方法栈就是虚拟机线程调用Native方法执行时的栈，它与虚拟机栈发挥类似的作用。但是要注意，虚拟机规范中没有对本地方法栈作强制规定，虚拟机可以自由实现，所以可以不是字节码。如果是以字节码实现的话，虚拟机栈本地方法栈就可以合二为一，事实上，OpenJDK和SunJDK所自带的HotSpot虚拟机就是直接将虚拟机栈和本地方法栈合二为一的。

Java虚拟机规范规定该区域也可抛出StackOverFlowError和OutOfMemoryError。

4、Java 堆

这个区域用来放置所有对象实例以及数组。不过在JIT（Just-in-time）情况下有些时候也有可能在栈上分配对象实例。堆也是java垃圾收集器管理的主要区域（所以很多时候会称它为GC堆）。从GC回收的角度看，由于现在GC基本都是采用的分代收集算法，所以堆内存结构还可以分块成：新生代和老年代；再细一点的有Eden空间、From Survivor空间、To Survivor空间等。如下图：



5、方法区

方法区是可供各条线程共享的运行时内存区域。存储了每一个类的结构信息，例如运行时常量池（Runtime Constant Pool）、字段和方法数据、构造函数和普通方法的字节码内容、还包括一些在类、实例、接口初始化时用到的特殊方法。

方法区创建时机：在虚拟机启动的时候创建。

方法区的容量：可以是固定大小的，也可以随着程序执行的需求动态扩展，并在不需要过多空间时自动收缩。方法区在实际内存空间中可以是不连续的。

6、直接内存

直接内存（Direct Memory）虽然不是程序运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但这部分内存也被频繁使用，而且它也可能导致OutOfMemoryError异常出现。

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方法库直接分配堆外内存，然后通过一个存储在Java堆里面的DirecByteBuffer对象作为这块内存的引用进行操作。这样能在某些应用场景中显著提高性能，因为它避免了在Java堆和Native堆中来回复制数据。

显然，本机直接内存的分配不会受到Java堆大小的限制，但是，还是会受到本机总内存（包括RAM及SWAP区或者分页文件）的大小及处理器寻址空间的限制，从而导致动态扩展时出现OutOfMemoryError异常。

7、执行引擎

将字节码即时编译、优化为本地代码， 然后执行。

类、对象、变量、方法在运行时的交互方式

在了解完这些知识以后，就可以知道：类和对象在运行时的内存里是怎么样的；以及各类型变量、方法在运行时是怎么交互的？

1、在程序运行时类是在方法区，实例对象本身在堆里面。

2、方法字节码在方法区。线程调用方法执行时创建栈帧并压栈，方法的参数和局部变量在栈帧的局部变量表。

3、对象的实例变量和对象一起在堆里，所以各个线程都可以共享访问对象的实例变量。

4、静态变量在方法区，所有对象共享。字符串常量等常量在运行时常量池。

5、各线程调用的方法，通过堆内的对象，方法区的静态数据，可以共享交互信息。

6、各线程调用的方法所有参数传递、方法返回值的返回，都是使用栈帧里的操作数栈来完成的。



为什么jdk1.8要把方法区从JVM里（永久代）移到直接内存（元空间）

原因一：

因为直接内存，JVM将会在IO操作上具有更高的性能，因为它直接作用于本地系统的IO操作。而非直接内存，也就是堆内存中的数据，如果要作IO操作，会先复制到直接内存，再利用本地IO处理。

从数据流的角度，非直接内存是下面这样的作用链：本地IO --> 直接内存 --> 非直接内存 --> 直接内存 --> 本地IO

而直接内存是：本地IO --> 直接内存 --> 本地IO

原因二：

整个永久代有一个 JVM 本身设置固定大小上线，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，并且永远不会得到java.lang.OutOfMemoryError。可以使用 -XX：MaxMetaspaceSize 标志设置最大元空间大小，默认值为 unlimited，这意味着它只受系统内存的限制。

-XX：MetaspaceSize 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。

GC Root

什么是垃圾？

简单的说就是内存中已经不再使用到的空间，就是垃圾。

如何判断一个对象是否可以被回收

1、引用计数法

简单来讲，给对象中添加一个引用计数器，当有一个地方引用它，计数器值+1，每当有一个引用失效时，计数器值-1。任何时刻，计数器值为零的对象，就是不可能再被使用的，那么这个对象就是可回收对象。但引用计数法目前已经没有人再使用了，因为他无法解决对象之间循环引用的问题。

2、枚举根节点做可达性分析（根搜索路径）

为了解决引用计数法的循环引用问题，Java使用了可达性分析的方法。

跟踪：Tracing

* + - * 复制：Copying
      * 标记-清楚：Mark-Sweep
* 标记-压缩：Mark-Compact



从根开始（GC引用遍历），根指向【对象B】，对象B又指向【对象B1】和【对象B2】，他们就是一条链上的蚂蚱，这时则说明他们对象可达。但是对象C，根并没有指向他，那么对象C就是对象不可达，是被回收的东西。

所谓GC roots或者说Tracing GC的“根集合”就是一组必须活跃的引用；言下之意，即：中间的链子不断。基本思路就是通过一系列名为“GC Roots”的对象作为起始点，从这个被称为GC Roots的对象开始向下搜索，如果一个对象到GC Roots没有任何引用链相连时，则说明此对象不可用。也就是说：给定一个集合的引用（也就是说有很多）作为根出发，通过引用关系遍历对象图，能被遍历到的（可到达的）对象就被判定为存活；没有被遍历到的，就自然被判定为死亡。

* 那么哪些对象可以作为GC Roots?

1. 虚拟机栈（栈帧中的局部变量区，也叫作局部变量表）。
2. 方法区中的类静态属性引用的对象。强引用，不容易被回收，使用需谨慎。
3. 方法区中常量引用的对象。强引用，不容易被回收，使用需谨慎。
4. 本地方法栈中JNI（native方法）引用的对象。比如多线程的start方法等。

举例如下：

**public** **class** GcRootsDemo {

**private** **byte**[] arr = **new** **byte**[100 \* 1024 \* 1024];

**private** **static** HoldThread holdThread; // 2) 方法区中的类静态属性引用的对象。强引用，不容易被回收，使用需谨慎。

**private** **static** **final** Task1001Position **task** = **new** Task1001Position("" , "" , **null**); // 3) 方法区中常量引用的对象。强引用，不容易被回收。

**public** **void** functions(String str) { // 1) 虚拟机栈方法要放到栈里，g1就是虚拟机栈中引用的对象。

GcRootsDemo gcRootsDemo = **new** GcRootsDemo();

System.gc();

}

**public** **static** **void** main(String[] args) {

**new** GcRootsDemo().functions("aaaa"); // 当被其他方法调用时，gcRootsDemo对象可达

}

}

XX参数

分类

Boolean类型



KV设值类型



jps + jinfo 如何查看一个运行中的Java程序，他的某个JVM参数是否开启？具体值是多少？

**第一步**：使用jps –l命令，获取正在运行的所有Java进程（包含名称、进程ID等等）



**第二步**：使用jinfo命令：jinfo –flag [具体jvm参数名称] [进程ID]。



我们尝试为这个程序添加一个JVM启动参数，在IDEA中，具体步骤如下：

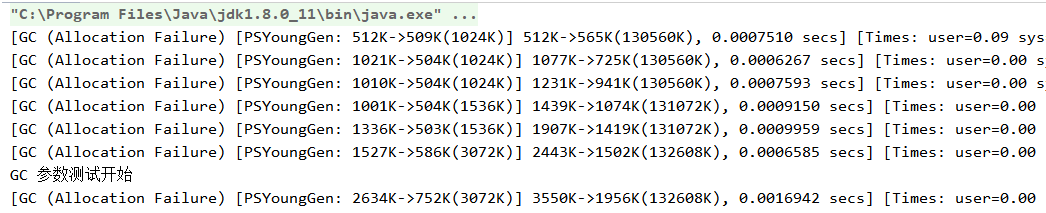




完成后，我们重新启动程序，再次按照上面的两个步骤在控制台输入命令，结果如下：



程序开启PrintGCDetails参数后，会在运行控制台打印出GC日志信息。



KV设值类型举例：MetaspaceSize

默认情况下：



21807104 ÷ 1024 ÷1024 = 20.796875M，默认约21M左右；当我们修改后：





计算结果：536870912 ÷ 1024 ÷1024 = 512M。类似的参数还有：MaxTenuringThreshold（最大任期阈值；Java对象在young区，多大的极限年龄可以升级为老年区，默认要活过15次）。

*Tenuring /ˈtenjər/任期; Threshold /ˈθreʃhoʊld/ 阈值*

程序还在执行，运行下命令：



JVM对MaxTenuringThreshold的默认值是15，这也是JVM可接受的最大值，通常这个值的范围在0到15之间，如果设置超过15会报如下错误：



jinfo -flags [进程ID]查看当前线程所有参数



Non-default VM flags代表系统默认的参数；Command line：代表人为调整后的参数。这其中，-XX:InitialHeapSize等价于我们常见的-Xms参数，默认初始值为当前机器物理内存的1/64 ；-MaxHeapSize等价于-Xmx参数，默认初始值为当前物理内存的1/4。

查看JVM默认值

-XX:+PrintFlagsInitial

主要用于查看JVM默认初始化的所有参数，调优是非常重要。





冒号等号是被修改过的标记。

-XX:+PrintFlagsFinal –version

主要查看JVM修改更新后的参数





当我们尝试运行java -XX:+PrintFlagsFinal -XX:MaxHeapSize=512m com.matrix.sxt.e14.HelloGc



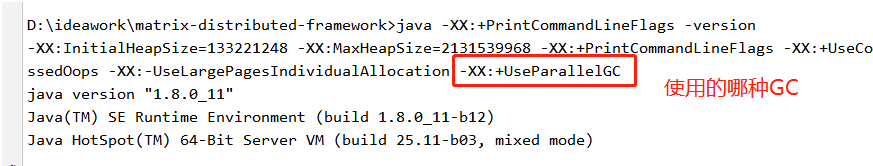
其结果如下：



计算结果：536870912 ÷ 1024 ÷1024 = 512M。

-XX:+PrintCommandLineFlags

打印HotSpotVM 采用的自动优化参数，默认的是并行GC回收器。



JVM调优常用参数

-Xms

初始内存大小，默认为物理内存的1/64；等价于：-XX:InitialHeapSize。

-Xmx

最大分配内存，默认为物理内存的1/4；等价于：-XX:MaxHeapSize。

-Xss

设置单个线程栈的大小，一般默认为512K ~ 1024K；等价于：-XX:ThreadStackSize。Windows系统默认的是0。



-Xmn

设置年轻代大小。

-XX:MetaspaceSize

JDK1.8替换了永久代，默认值是21M，相对较小；如果系统对象创建比较频繁，可能会导致OOM异常，故此参数是JVM调优的主要对象。

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。但他们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，元空间的大小受到本地内存的限制。最大值默认为本机内存的1/4。

-XX:+PrintGCDetails

输出详细的GC日志信息。

-XX:SurvivoRatio

设置新生代中eden和S0/S1空间的比例。默认情况下 –XX:SurvivorRatio=8，Eden:S0:S1=8:1:1；假如设置 –XX:SurvivorRatio=4，那么Eden:S0:S1=4:1:1。

SurvivorRatio的值就是设置eden区的比例占多少，S0/S1是相同的。

-XX:NewRatio

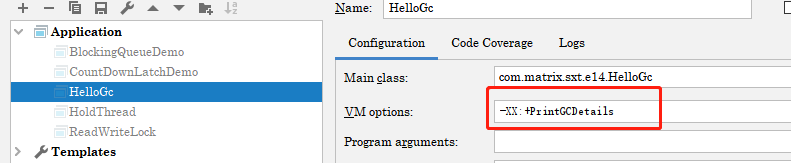
配置年轻代与老年代在堆结构的占比。默认情况下 –XX:NewRatio=2，新生代占1，老年代占2，年轻代占整个堆的1/3。假如设置成 –XX:NewRatio=4，新生代占1，老年代占4，年轻代占整个堆的1/5。NewRatio就是设置老年代的占比，剩下的1给新生代。

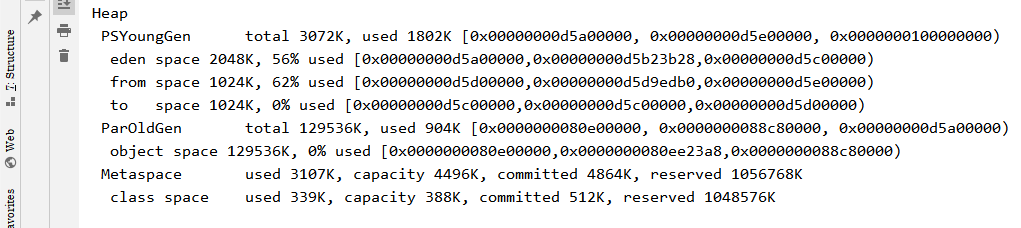
-XX:MaxTenuringThreshold

设置垃圾的最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入老年代。对于老年代比较多的应用，可以显著提高效率。如果将此值设置为一个较大的值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代的存活时间，增加在年轻代被回收的概率。

GC详细日志解读

程序运行之前加入参数：-XX:+PrintGCDetails，在程序执行完成后，会打印出详细的GC信息





PSYoungGen：新生代；ParOldGen：老年代；Metaspace：元数据。我们模拟一个OOM异常，以此来分析GC日志信息。修改VM optinos为：-Xms10m -Xmx10M -XX:+PrintGCDetails。初始内存大小和最大分配内存都设置成10M，如果模拟一个数组大小为50M ：byte[] array = new byte[50 \* 1024 \* 1024];

**程序运行后**：

[GC (Allocation Failure) [PSYoungGen: 2048K->509K(2560K)] 2048K->993K(9728K), 0.0008996 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[GC (Allocation Failure) [PSYoungGen: 2557K->496K(2560K)] 3041K->1384K(9728K), 0.0008834 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC (Allocation Failure) [PSYoungGen: 509K->0K(2560K)] [ParOldGen: 1068K->1271K(7168K)] 1577K->1271K(9728K), [Metaspace: 3068K->3068K

(1056768K)], 0.0052131 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] [ParOldGen: 1271K->1216K(7168K)] 1271K->1216K(9728K), [Metaspace: 3068K->3068K

(1056768K)], 0.0092008 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

at com.matrix.sxt.e14.HelloGc.main(HelloGc.java:7)

可见GC的详细日志可以分为黑色字体部分（GC (Allocation Failure）、蓝色字体部分（Full GC (Allocation Failure)）。这两部分我们分开来详细解读：

GC (Allocation Failure)

[GC (Allocation Failure) [PSYoungGen: 2048K->509K(2560K)] 2048K->993K(9728K), 0.0008996 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[GC (Allocation Failure)--------------------------------------GC类型 （分配失败）

[PSYoungGen:

2048K------------------------------------------------------YoungGC前新生代内存占用

->

509K--------------------------------------------------------YoungGC后新生代内存占用

(2560K)----------------------------------------------------新生代总共大小

]

2048K-----------------------------------------------------------YoungGC前JVM堆内存占用

->

993K-------------------------------------------------------------YoungGC后JVM堆内存占用

(9728K)---------------------------------------------------------JVM堆总大小

, 0.0008996 secs----------------------------------------------YoungGC耗时

]

[Times:

user=0.00--------------------------------------------------YoungGC用户耗时

sys=0.00,---------------------------------------------------YoungGC系统耗时

real=0.00 secs--------------------------------------------YoungGC实际耗时

]

Full GC (Allocation Failure)

[名称：GC前内存占用 -> GC后内存占用 (该区内存总大小)]

[Full GC (Allocation Failure) [PSYoungGen: 509K->0K(2560K)] [ParOldGen: 1068K->1271K(7168K)] 1577K->1271K(9728K), [Metaspace: 3068K->3068K

(1056768K)], 0.0052131 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

[Full GC (Allocation Failure) --------------------------------------GC类型 （分配失败）

[PSYoungGen: ------------------------------------------------------Young区

509K---------------------------------------------------------------GC前Young区内存占用

->

0K ----------------------------------------------------------------GC后Young区内存占用

(2560K) -----------------------------------------------------------Young区总大小

]

[ParOldGen: ---------------------------------------------------------Old区

1068K--------------------------------------------------------------GC前Old区内存占用

->

1271K--------------------------------------------------------------GC后Old区内存占用

(7168K) ------------------------------------------------------------Old区总大小

]

1577K-------------------------------------------------------------------GC前**堆**内存占用

->

1271K -------------------------------------------------------------------GC后**堆**内存占用

(9728K), ----------------------------------------------------------------JVM**堆内存总大小** 9728÷1024=9.5M，约等于10M

[Metaspace: -----------------------------------------------------------元空间（JDK1.7为PSPermGen，即：Perm区，永久带）

3068K---------------------------------------------------------------GC前元空间大小

->

3068K---------------------------------------------------------------GC后元空间大小

(1056768K) -------------------------------------------------------元空间总大小：1056768÷1024÷1024≈1M

],

0.0052131 secs--------------------------------------------------------GC耗时

]

[Times: user=0.00 sys=0.00, real=0.01 secs]

GC 和 Full GC 有什么区别？

GC（或**Minor** GC）：收集 生命周期短的区域(Young area)。

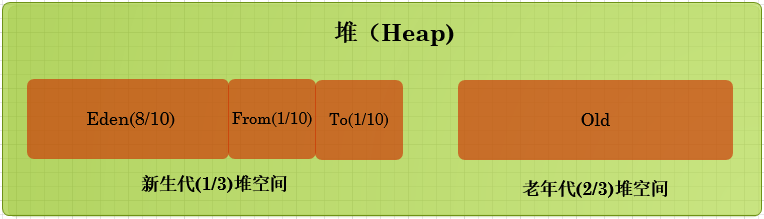
Full GC（或**Major** GC）：收集生命周期短的区域(Young area)和生命周期比较长的区域(Old area)，对整个堆进行垃圾收集。

Minor GC后，Eden是空的吗？

是的，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。GC 效率也会比较高，我们要尽量减少Full GC 的次数。当显示调用System.gc()时，gc does a full collection(both young generation and tenured generatio。显式调用system.gc()会触发full gc，对象在Eden出生每经历一次Minor GC后仍然存活，并且能被Survivor容纳，就会被移动到Survivor，并设定年龄为1.以后在Survivor每"熬过"一次Minor GC，年龄就增加一岁，当年龄超过一定值就被移动到老年代，MaxTenuringThreshold用于设置年龄阈值，但是如果Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄。大对象直接进入老年代，比如长数组或长字符串，-XX:PretenureSizeThreshold用于设置大于这个值直接在老年代分配。

Full GC是清理整个堆空间—包括年轻代、老年代和元空间(jdk1.7为永久代)。有时候系统会频繁的FullGC，这时候需要去服务器查一下原因。Full GC 老年代扛不住的时候，会报出OOM异常。

新生代与老年代的概念与结构



Java从GC的角度可以细分为新生代（Eden区、From Survivor区和To Survivor区）和老年代。

MinorGC的过程（复制->清空->互换）

Step：1 Eden和SurvivorFrom区的对象复制到SurvivorTo区，年龄+1

首先，当Eden区满的时候，会触发第一次GC，把还活着的对象拷贝到SurvivorFrom区；当Eden区再次触发GC的时候，会扫描Eden区和From区，对这两个区域进行垃圾回收，经过这次回收后还存活的对象，则直接复制到To区（如果有的对象年龄已经达到了老年的标准，则复制到Old区），同时把这些对象的年龄+1。

Step：2清空Eden、SurvivorFrom

然后，清空Eden和SurvivorForm中的对象，也即复制之后有交换，原来的From和To谁空了谁就变成To区。

Step：3 SurvivorTo和SurvivorFrom互换

最后，SurvivorTo和SurvivorFrom进行互换，原来的SurvivorTo成为下一次GC时的SurvivorFrom区。部分对象会在From和To区域中复制来复制去，如此交换15次（由JVM参数MaxTenuringThreshold来决定，JVM默认为15次），最终还是存活则放入老年代。

常见OOM异常及触发原因

1、StackOverflowError

经典的错误（不是异常），栈空间溢出。最常见的触发这个错误的方式是方法的深度调用：递归。

**public** **class** StackOverflowErrorDemo {

**public** **static** **void** main(String[] args) {

stackOverflowError();

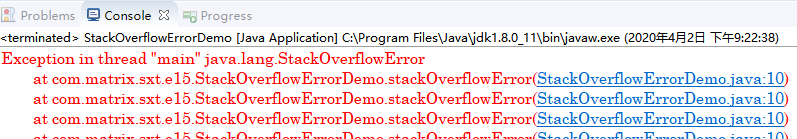
}

**private** **static** **void** stackOverflowError() {

stackOverflowError();

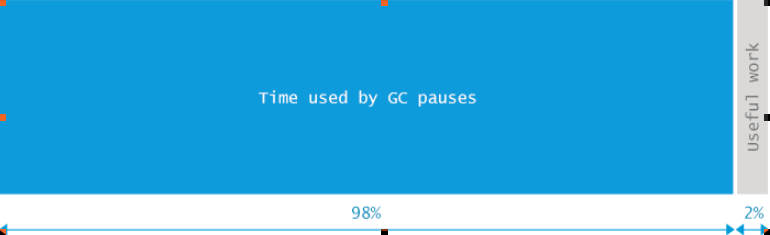
}

}



2、Java.lang.OutOfMemoryError: GC overhead limit exceeded

GC回收(次数、频率)超过最大极限值。也就是说目前运行的程序写的特别诡异，导致Java的内存极具上升，比如大量的对象被装载，被新建出来，而虚拟机资源98%以上都用作垃圾回收了，但是回收效果很糟糕，导致GC一直在运行，真正用作计算、处理的行为比例，少的可怜（往往少于2%），事倍功“微”。



GC回收时间过长时，会抛出Java.lang.OutOfMemoryError: GC overhead limit exceeded错误。过长的定义：超过98%的时间用来做GC但是回收了不到2%的内存，连续多次GC都只回收了不到2%的极端情况才会抛出此错误。假如Java虚拟机不抛出GC overhead limit exceeded错误会发生什么情况呢？那就是GC清理的这么点儿内存会很快再次填满，迫使GC再次执行，从而形成一个恶性循环：CPU使用率一直是100%，而GC却没有任何成果。导致原因多是死循环。

**public** **class** GcOverheadDemo {

**public** **static** **void** main(String[] args) {

**int** i = 0;

List<String> list = **new** ArrayList<String>();

**try** {

**while**(**true**) { // 此处只是利用死循环来模拟，实际开发切记不可这么写。

list.add(String.valueOf(++ i).intern());

}

} **catch** (Throwable e) { // Exception和Error的上层接口

System.**out**.println("$$$$$$$$$$$$$$$$$$$$ i = " + i);

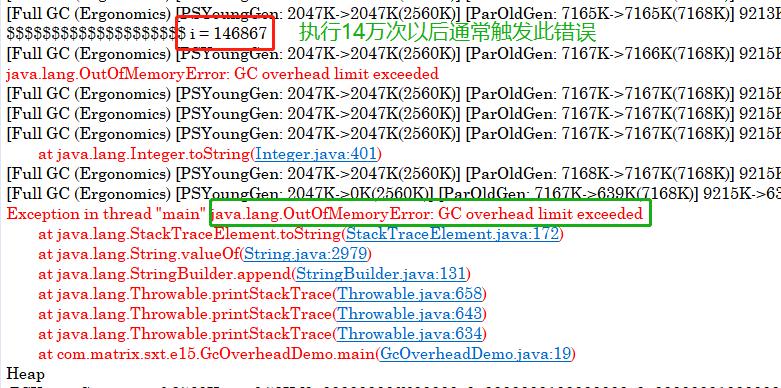
e.printStackTrace();

}

}

}

执行程序前配置VM参数：-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:MaxDirectMemorySize=5M，运行结果如下：

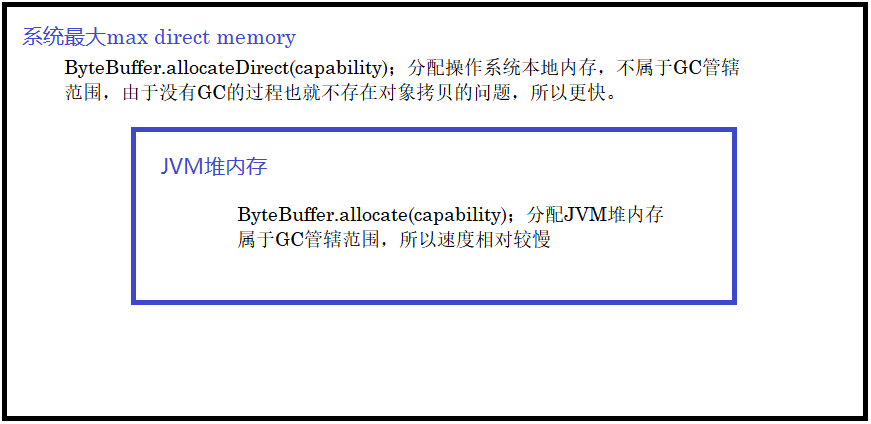


3、Java.lang.OutOfMemoryError: Direct buffer memory

此故障产生的原因：在写NIO程序时，经常使用ByteBuffer来读取或者写入数据，这是一种基于通道（Channel）与缓冲区（Buffer）的IO方式，他可以使用native函数库直接分配堆外内存，然后通过一个存储在Java堆里面的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。

第一种：ByteBuffer.allocate(capability)；分配JVM堆内存，属于GC管辖范围，由于MinorGC的过程（复制->清空->互换）需要拷贝，所以速度相对较慢。

第二种：ByteBuffer.allocateDirect(capability)；分配操作系统本地内存，不属于GC管辖范围，由于没有GC的过程也就不存在对象拷贝的问题，所以更快。



但如果不断分配本地内存，堆内存很少使用，那么JVM就不需要执行GC，DirectByteBuffer对象就不会被回收，这时候堆内存充足，但本地内存已经使用光了，再次尝试分配本地内存就会出现OOM，程序就会直接崩溃。

**import** java.nio.ByteBuffer;

**import** java.util.concurrent.TimeUnit;

**public** **class** DirectBufferMemory {

@SuppressWarnings("restriction")

**public** **static** **void** main(String[] args) {

**long** maxDirectMemory = sun.misc.VM.maxDirectMemory() / 1024 / 1024;

System.**out**.println("系统最大max direct memory = " + maxDirectMemory + " M");

// 默认系统最大max direct memory = 1808 M，本机系统8G内存，约1/4

**try** {TimeUnit.**SECONDS**.sleep(3);} **catch** (InterruptedException e) {e.printStackTrace();}

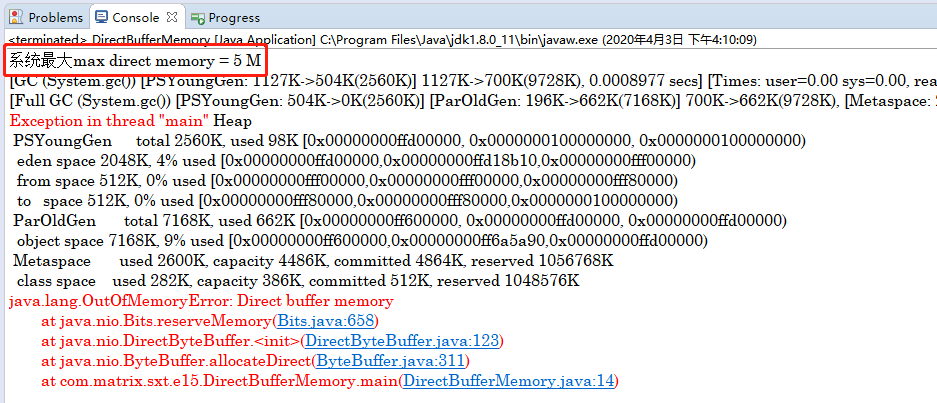
// -XX:MaxDirectMemorySize=5m 我们配置为5MB启动程序，但代码中实际分配6MB空间，从而引发Direct buffer memory错误

ByteBuffer bb = ByteBuffer.allocateDirect(6 \* 1024 \* 1024);

}

}

执行程序前配置VM参数：-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:MaxDirectMemorySize=5M，运行结果如下：



4、Java.lang.OutOfMemoryError: unable to create new native thread

高并发请求服务器时，经常出现的一种错误：无法创建新的线程；准确的讲native thread异常与对应的平台有关。

* 导致原因：

1、当前应用(注意！这里不是单指某一段代码或某个类，而是一个整体应用项目)创建了太多的线程，超过系统承载极限。

2、当前服务器不允许你的应用程序创建这么多线程，Linux系统默认可创建最大线程数为1024个。

* 解决办法：

1、想办法降低应用程序创建的线程数量，分析应用是否真的需要创建这么多线程，如果不是改代码，降低线程数量。

2、修改Linux服务器配置，扩大Linux默认限制（Linux系统中，Root用户创建线程不受限制）。

示例代码如下：

**public** **class** UnableCreateNewNativeThread {

**public** **static** **void** main(String[] args) {

**for**(**int** i = 0 ; i < 2048; i ++) {

System.**out**.println("i = " + i);

**new** Thread(() -> {

**try** {TimeUnit.**SECONDS**.sleep(Integer.**MAX\_VALUE**);} **catch** (InterruptedException e) {e.printStackTrace();}

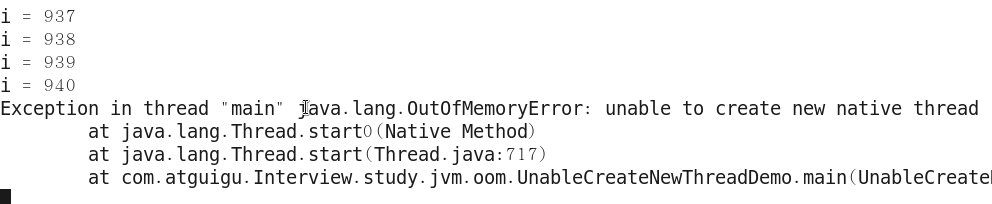
}, "线程：" + i).start();

}

}

}

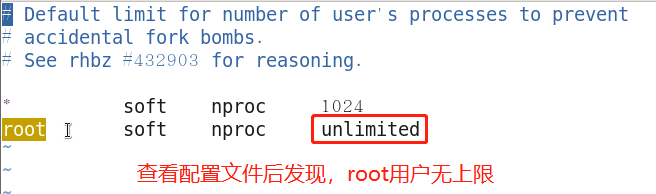
运行结果如下：



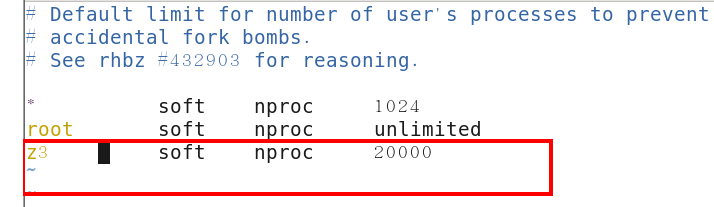
如何修改Linux默认线程数量限制？

在Linux终端或Xshell中：





图中【root】上面的\*号代表所有非Root用户，他们都被限制为1024个线程。如果们想修改、扩大这个配置，只需要在配置文件下面加入指定的用户即可：



但通常来讲，极少这样修改，不推荐修改默认配置。

实际生产环境会在哪儿出现？

当我们的生产环境中的下单服务，根据业务需求在执行下单的过程中，开启了10个线程分别用于计算营销活动、会员权益、拆单等，单台虚拟机的配置为8核心、8G内存，200G硬盘。如果按照IO密集型，阻塞系数0.1来计算，Tomcat线程池合理数量应该为80个；但如果Nginx负载到每个Tomcat上的请求是150个，那么Tomcat会超负荷运行，他会启动150个线程来应对请求，每个下单请求又会启动10个线程，即1500个线程会在同一个时刻被启动，这将会直接导致：Java.lang.OutOfMemoryError: unable to create new native thread错误出现。

也就是说Linux默认线程数是Nginx负载的一个重要依据。

5、Java.lang.OutOfMemoryError: Metaspace

在Java8及以后的版本，使用Metaspace来替代永久代。Metaspace是方法区在HotSpot虚拟机中的实现，它与持久代最大的区别在于：Metaspace并不在虚拟机内存中，而是使用本地内存。即：在java8以后的版本中，class metadata(The virtual machines internal presentation of java class)，被存储在叫做Metaspace

的native memory中。永久代（java8以后被Metaspace取代）存放了如下信息：

1虚拟机加载的类信息（比如rt.jar包下的类：String.java、Object.java、ArrayList.java等等），这些信息属于基础的模板和元数据，用于支持开发者使用。

2常量池

3静态变量

4即时编译后的代码

我们可以通过不断的生成类，往元空间里灌，类占据的空间总是会超过Metaspace指定的空间大小，从而模拟Metaspac空间溢出。代码如下：

**import** com.matrix.sxt.e14.HelloGc;

**import** org.springframework.cglib.proxy.Enhancer;

**import** org.springframework.cglib.proxy.MethodInterceptor;

**import** org.springframework.cglib.proxy.MethodProxy;

**import** java.lang.reflect.Method;

**public** **class** MetaspaceDemo {

**public** **static** **void** main(String[] args) {

**int** i = 0;

**try** {

**while** (**true**){

i++;

Enhancer enhancer = **new** Enhancer(); // cglib动态字节码技术，单独章节介绍

enhancer.setSuperclass(HelloGc.**class**);

enhancer.setUseCache(**false**);

enhancer.setCallback(**new** MethodInterceptor() {

@Override

**public** Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) **throws** Throwable {

**return** methodProxy.invokeSuper(o , args);

}

});

enhancer.create();

}

}**catch** (Throwable e){

System.**out**.println("类创建了【" + i + "】次后发生异常");

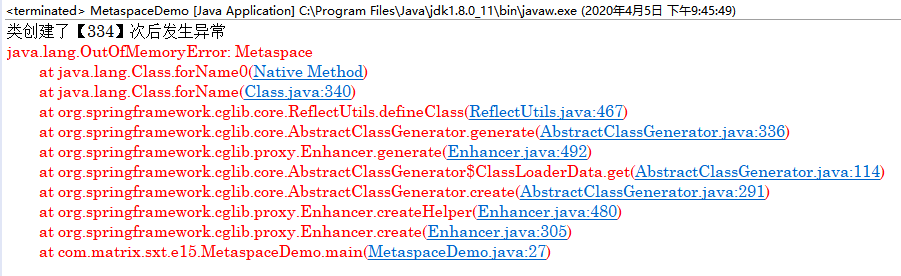
e.printStackTrace();

}

}

}

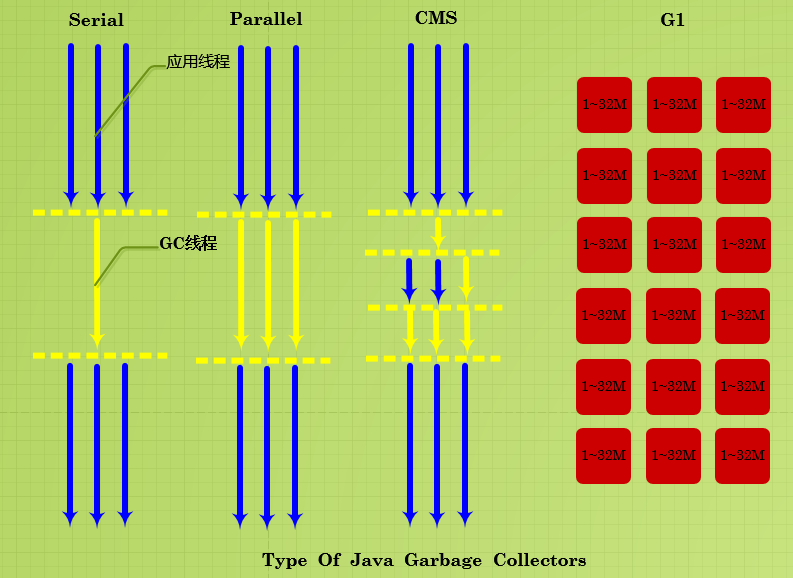
执行程序钱，加上如下Jvm参数：-XX:MetaspaceSize=8M -XX:MaxMetaspaceSize=8M。



垃圾回收算法与垃圾回收器

垃圾回收器基础图解

GC算法（引用计数、复制、标记清除、标记整理）是内存回收的方法论，是一种理论和算法，并不是具体的实现方式。垃圾回收器是这些理论和算法的落地实现。到目前为止（Jdk14）还没有完美的垃圾收集器出现，也没有万能的收集器，只是针对具体的应用业务，找到最合适的收集器，进行分代收集。比如复制算法用在新生代来使用，标记清除和标记整理两种算法通常用在老年代来使用。



* **串行垃圾回收器（Serial）**：

为单线程环境设计并且只使用一个线程进行垃圾回收，会暂停所有用户线程；所以不适合服务器环境。jdk早期版本，目前已经弃用。

* **并行垃圾回收器（Parallel）**：

多个垃圾回收线程并行工作，此时用户线程也是暂停的，适用于科学计算/大数据处理等弱交互场景。Serial的改进型，并行垃圾收集，效率提升，用户线程等待时间变短。Jdk1.8以前都是默认的回收器。

* **并发垃圾回收器（CMS）**：

用户线程和垃圾收集线程同时执行（不一定并行，可能交替执行），不需要停顿用户线程，互联网服务多用此种回收器，适用于对响应时间有要求的场景。

* **G1垃圾回收器**：

G1将堆内存分割成不同的区域，大小为1~32M之间，然后并发的对其进行垃圾回收。它是对CMS的改进和优化。对比CMS优点是不再产生内存碎片，同时效率更高。Jdk1.7开始推出，1.8正式推荐使用。Java9的默认垃圾回收器。

生产环境垃圾收集器的查看与配置

输入命令：java -XX:+PrintCommandLineFlags –version 然后回车，可以看到当前环境的GC回收器：



默认使用的是UseParallelGC。

常用的垃圾回收器有：

1、-XX:+UseSerialGC(串行垃圾回收器)；

2、-XX:UseParallelGC；

3、-XX:UseConcMarkSweepGC；

4、-XX:UseParNewGC；

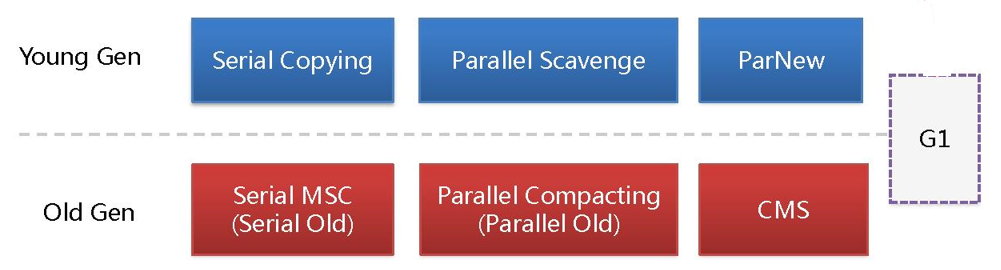
5、-XX:UseParallelOldGC；

6、-XX:UseG1GC；

垃圾收集器的一些简称与常识

常见缩写与对应全称

|  |  |  |
| --- | --- | --- |
| 简称 | 全称 | 中文释义 |
| DefNew | default new generation | 默认新生代使用的哪一种垃圾回收器。 |
| Tenured | old | 旧的，老的。Tenured Generation即老年代。 |
| ParNew | parallel new generation | 即：在新生代使用并行垃圾收集器。使用-XX:+UseParNewGC（新生代使问用并行收集器，老年代使用串行回收收集器）或者-XX:+UseConcMarkSweepGC(新生代使用并行收集器，老年代使用CMS) |
| PSYoungGen | parallel scavenge | 即：在新生代(Young generation)使用并行清除算法。使用-XX:+UseParallelOldGC（新生代，内老年代都使用并行回收收容集器）或者-XX:+UseParallelGC（新生代使用并行回收收集器，老年代使用串行收集器） |
| ParOldGen | parallel old generation | parallel scavenge对应的老年代算法。 |
|  |  |  |



Serial MSC已经废弃不用。除了G1以外，每种GC都有自己适合的使用情景，他们并不是通用的。

Server/Client模式

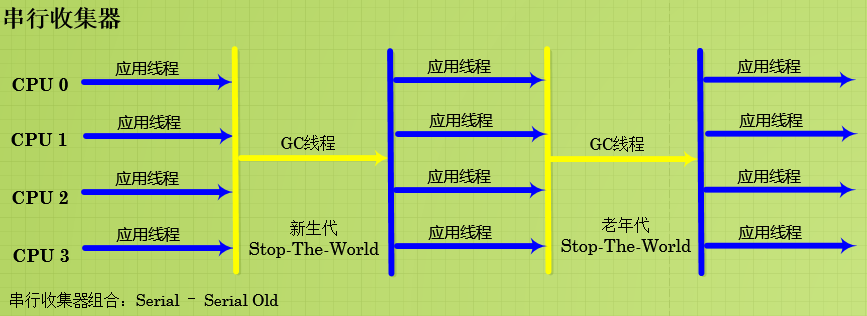
32位的Windows操作系统，不论硬件如何都默认使用Client的JVM模式。32位的其他操作系统，2G内存同时有2个CPU以上的机器用Server模式，低于该配置则使用Client模式。对于64位操作系统，只能用Server模式。

垃圾回收器的经典组合方式

新生代使用哪种收集器，则它的老年代使用的收集器是一一对应的；他们是被设计好的几种固定组合方式。

串行垃圾回收器（Serial / Serial Coping / Serial Old）

**特点**：新生代1个线程回收垃圾，老年代1个线程回收垃圾。



虽然在收集垃圾过程中需要暂停所有其他工作线程，但是他简单高效，对于限定单个CPU的环境来说（类似树莓派），没有线程交互的开销可以获得最高的单线程垃圾收集效率，因此Serial垃圾收集器依然是Java虚拟机运行在Client模式下的新生代垃圾收集器。对应的JVM参数是：-XX:+UseSerialGC。当开启这个参数后会使用：Serial(Young区用) + Serial Old(Old区用)的收集器组合；表示新生代、老年代都会使用串行垃圾收集器，新生代使用复制算法，老年代使用标记-整理算法。

示例代码如下：

**public** **class** GcSerialDemo {

**public** **static** **void** main(String[] args) {

**int** i = 0;

List<String> list = **new** ArrayList<String>();

**try** {

**while**(**true**) {

list.add(String.valueOf(++ i).intern());

}

} **catch** (Throwable e) { // Exception和Error的上层接口

e.printStackTrace();

}

}

}

运行前添加参数：-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseSerialGC

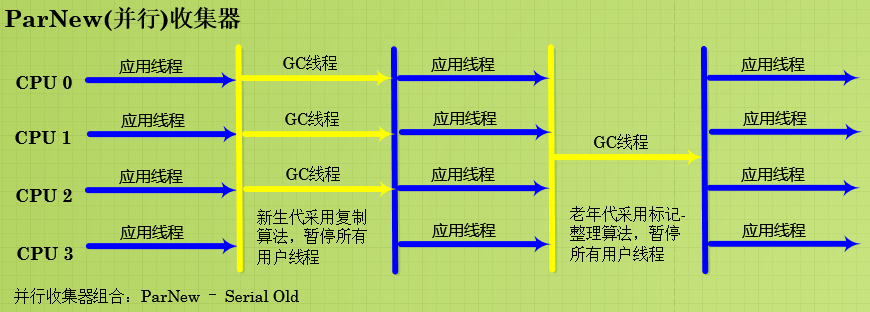
运行结果如下图：



**关于Serial Old收集器**：Serial收集器的老年代版本，同样也是单线程收集器，使用标记整理算法，他的主要用途是配合CMS收集器使用，为CMS做最后一层保障，不至于因为CMS的异常导致整个系统死机。

并行垃圾回收器（ParNew）

**特点**：新生代N个线程（默认CPU核心数个线程）回收垃圾，老年代1个线程回收垃圾。



ParNew收集器是Serial收集器在新生代的并行多线程版本，最常见的应用场景是配合老年代的CMS GC工作，其余的行为和Serial收集器完全一样，当他在收集垃圾的过程中，同样会暂停其他的工作线程，只不过他提供了更多的垃圾回收线程，效率更高了。在【垃圾回收器基础图解】中已经详细绘制了其算法过程。他是很多Java虚拟机运行在Server模式下**新生代**的默认垃圾收集器。

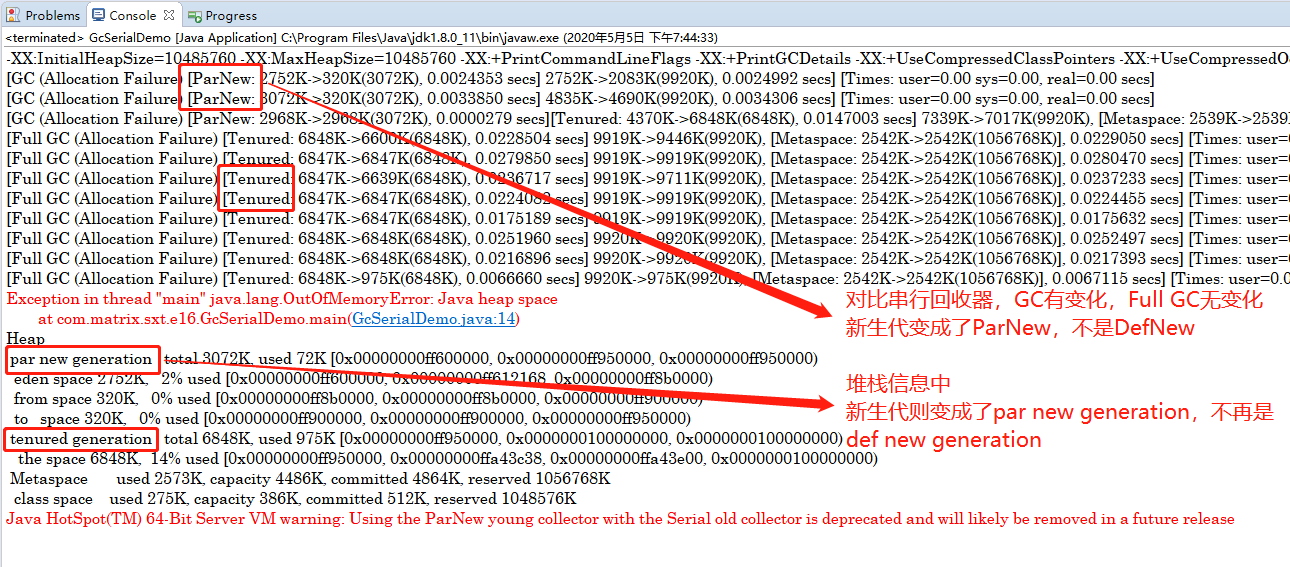
启用参数：-XX:+UseParNewGC 启用ParNew只影响新生代的垃圾收集，不影响老年代。开始此参数后，会使用ParNew（Young区）+Serial Old的收集器组合，新生代使用复制算法，老年代采用标记-整理算法。但从Java8开始ParNew+Tenured这样的默认组合方式已经不再推荐，后续Java更高版本会移除。

备注：

关于限制垃圾回收的线程数量问题，可以使用：-XX:ParallelGCThreads参数来调整；Java默认开启和CPU数目相同的线程数。

运行前添加参数：-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseParNewGC

运行结果如下图：



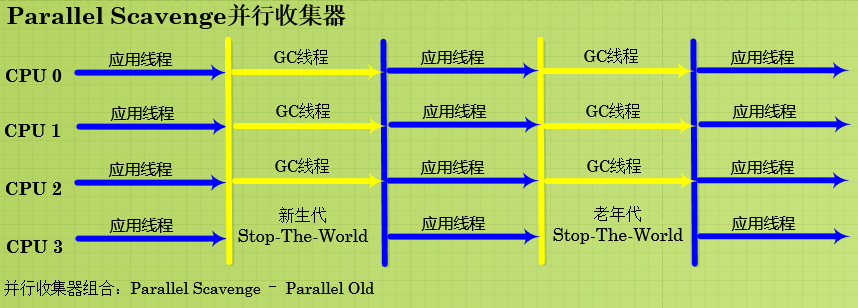
从上图中可以看到日志的最后打印出：

Java HotSpot(TM) 64-Bit Server VM warning: Using the ParNew young collector with the Serial old collector is deprecated and will likely be removed in a future release。

也就是说，后面会废弃掉。上文中提到ParNew收集器最常见的应用场景是配合老年代的CMS GC工作；当使用-XX：+UseConcMarkSweepGC时，-XX：UseParNewGC会自动开启。因此，如果年轻代的并行GC不想开启，可以通过设置-XX：-UseParNewGC来关掉。CMS后文详细拆解。

并行垃圾回收器（Parallel / Parallel Scavenge）

**特点**：新生代N个线程回收垃圾，老年代N个线程回收垃圾。



这是Java 8默认的垃圾回收器组合方式。Parallel Scavenge收集器类似于ParNew，也是一个新生代的垃圾收集器，使用复制算法，是一个并行的多线程的垃圾收集器，俗称吞吐量优先收集器，这种垃圾收集器在新生代和老年代都是并行化的。

Parallel Scavenge与ParNew不同的关注点：

**1、可控制的吞吐量**

Thoughput(美 /ˈθruːpʊt/某一时期内的生产量，接待人数，吞吐量) = 运行用户代码时间 / （运行用户代码时间+垃圾收集时间）；比如程序运行100分钟，垃圾收集时间1分钟，吞吐量等于99%；高吞吐量意味着高效利用CPU的时间，它多用于在后台运算而不需要太多交互的任务。

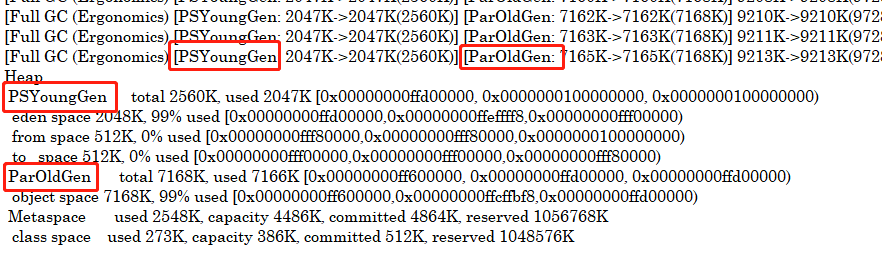
**2、自适应调节策略**

虚拟机会根据当前系统的运行情况，收集性能监控信息，动态调整这些参数从而提供最合适的停顿时间(-XX:MaxGCPauseMillis)或最大吞吐量。-XX:MaxGCPauseMillis=N，N表示启动多少个GC线程。其计算方式分为2种情况：A) cpu > 8，N=5/8；B) cpu < 8，N=实际个数。

启动参数：-XX:UseParallelGC 或 -XX:UseParallelOldGC，他们可以相互激活使用Parallel Scavenge收集器。

运行前添加参数：-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseParallelGC

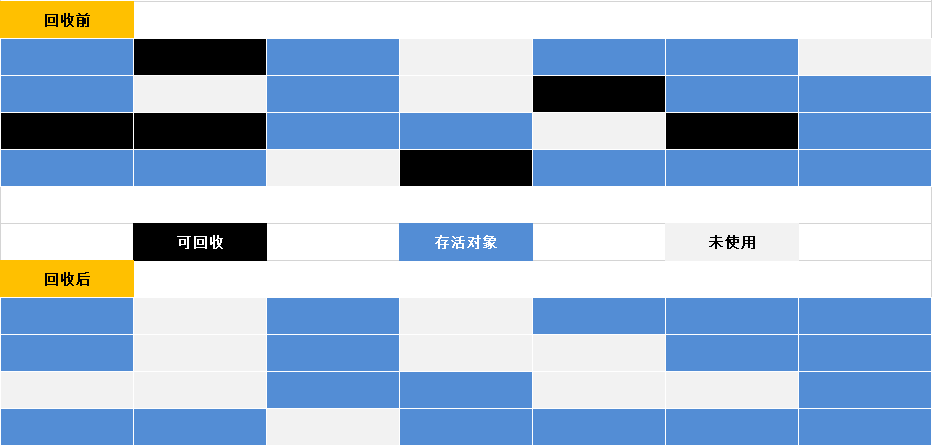
运行结果如下图：



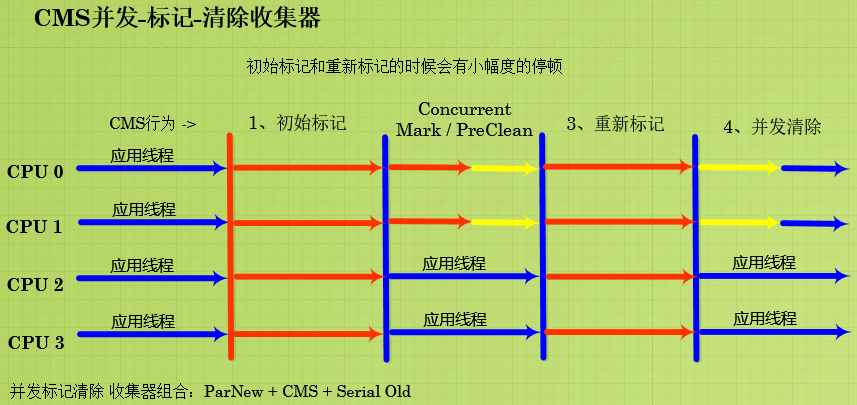
PSYoungGen即上面表格中的parallel scavenge；ParOldGen即上面表格中的parallel old generation。

CMS并发-标记-清除收集器

标记清除算法（Mark-Sweep）**特点**：算法分成标记和清除2个阶段，先标记出要回收的对象，然后统一回收这些对象，优点是节约内存空间，缺点是会产生较多的碎片。如下图：



这种算法的落地实现就是常说的CMS垃圾回收器（Concurrent Mark Sweep），他是一种以获取“最短回收停顿时间”为目标的收集器。适合应用在互联网或者其他高并发领域的服务器上，这类应用尤其重视服务器的响应速度，希望系统的停顿时间最短。CMS非常适合堆内存大、CPU核数多的服务器端应用，是G1出现之前的首选垃圾收集器。如下图所示，Concurrent Mark Sweep并发标记清除，并发标记低停顿其中“并发”指的是与用户线程一起执行。



上图中

第一步：

初始标记要回收的对象，此刻会造成短暂的暂停，这是第一次Full GC。

第二步：

Concurrent Mark/PreClean在进行GC Roots的过程中，和用户线程一起工作，不需要暂停工作线程，这一步是主要标记过程，标记全部对象。

第三步：

重新标记存在的意义是：由于前两步：初始标记和并发标记的过程中，有的对象又有可能不需要**收集**了，所以需要进行二次确认的过程。官方解释如下：为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，仍然需要暂停所有的工作线程，即此刻会造成短暂的暂停，这是第二次Full GC。

第四步：

清除GC Roots不可达对象，和用户线程一起工作，不需要暂停工作线程，基于标记结果，直接清理对象。注意：这一步并没有进行堆空间碎片压缩。

结合上述四步以及前面介绍的三种垃圾回收器组合方式，从总体上来看CMS收集器的内存回收和用户线程是近似一起并发的执行。前三种垃圾回收器的策略是要么回收垃圾，要么线程工作；CMS本质上是对前三种回收方式的优化和升级，将整个回收过程分成了四步，让其中的1、3两步短暂暂停，2、4两步并发执行，从而减少JVM在回收垃圾时，整体的暂停时间。在G1出现之前这是最优秀的解决方案，但他也有一个**弊端**：如果堆空间比较大，如何更好的提升效率？

启动参数：-XX:+UseConcMarkSweepGC；开启该参数后会自动将-XX:+UserParNewGC打开。当开启后，会使用：

ParNew(Young区) + CMS(Old区) + Serial Old的收集器组合方式；Serial Old将作为CMS出错后的备用收集器。

**优点**：并发收集停顿低。

**缺点**：

1）并发执行，对CPU资源压力大。由于并发执行，CMS的收集线程与程序应用线程会同时增加对堆内存的占用，也就是说，CMS必须要在老年代堆内存用尽之前，完成垃圾回收，否则CMS垃圾回收将会失败，然后将触发担保机制，Serial Old收集器将会以STW（Stop The World）的方式进行一次GC，从而造成较大的停顿时间。

2）采用标记清除算法会导致大量碎片（这是标记清除类算法的通病）。标记清除算法无法整理空间碎片，老年代空间会随着应用时长被逐步耗尽，最后将不得不通过担保机制对堆内存进行压缩。CMS也提供了参数：-XX:CMSFullGCsBeForeCompaction（默认为0，即每次都进行内存整理）来指定多少次CMS收集之后，进行一次压缩的Full GC。

运行前添加参数：-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseConcMarkSweepGC

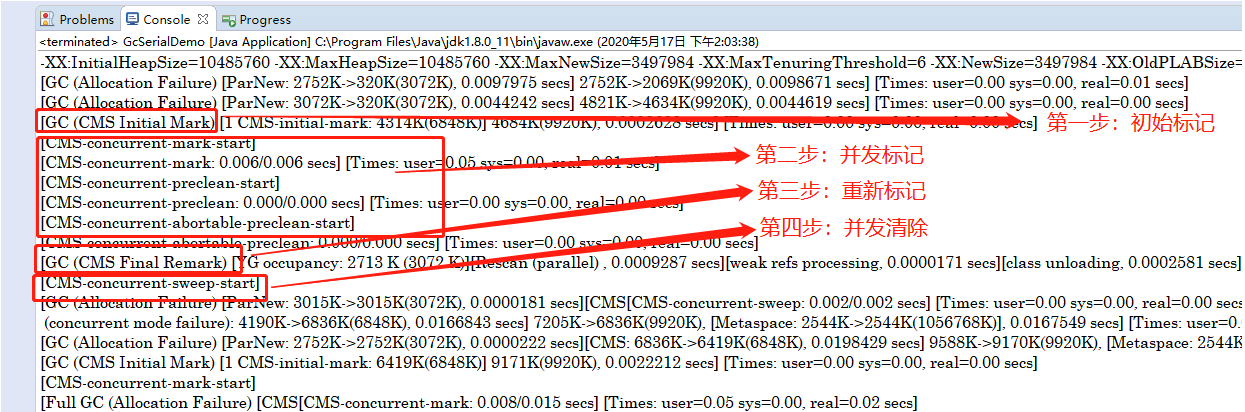
运行结果如下：

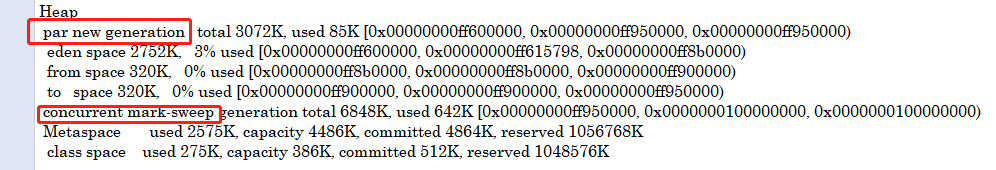
-XX:InitialHeapSize=10485760 -XX:MaxHeapSize=10485760 -XX:MaxNewSize=3497984 -XX:MaxTenuringThreshold=6

-XX:NewSize=3497984 -XX:OldPLABSize=16 -XX:OldSize=6987776 -XX:+PrintCommandLineFlags -XX:+PrintGCDetails

-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseConcMarkSweepGC

-XX:-UseLargePagesIndividualAllocation -XX:+UseParNewGC





Heap显示Young区：par new generation[eden space 2752K，from区 320K，to区 320K]，Old区：concurrent mark-sweep generation。

G1垃圾回收器

G1简介

G1（Garbage First）收集器是一款面向服务端应用程序的收集器，重点使用在多处理器和大容量内存环境中（在堆空间较大的应用程序中，CMS收集器存在效率问题），在实现高吞吐量的同时，尽可能的满足垃圾收集暂停时间“短”的要求。另外，它还具有一下特性：

1、像CMS一样，能够与应用程序一起并发执行。

2、整理空闲的碎片空间更快（CMS的使用Serial Old收集器的瓶颈之处）。

3、吞吐性能更高

4、不需要更大的Java Heap。

G1收集器的目标就是为了取代CMS收集器，他与CMS相比，在以下方面表现更加出色：1、G1在整理内存的过程中，只会产生较少的内存碎片；2、G1的STW更加可控，G1在停顿时间上添加了预测机制，用户可以指定期望的停顿时间；这是一个极其强悍的功能，更加细粒度的操作性意味着在进行JVM调优的时候可以更加合理的把控应用程序的性能指标。

G1首次发布在2012年的Java7中，其版本是jdk1.7u4开始携带。在Jdk9中，oracle已经将G1作为默认的垃圾收集器使用，并替代CMS。

* G1之前的垃圾回收器特点：

1、年轻代和老年代是各自独立且连续的内存块；

2、年轻代收集使用单Eden + S0 + S1进行操作回收的复制算法

3、老年代进行垃圾回收必须扫描整个老年代区域；

4、都是以尽可能少而快速地执行GC为设计原则。

G1的主要改变是Eden、Survivor和Tenured等内存区域不再是连续的，而是变成了一个个小的region，每个region从1M到32M不等。一个region可能属于Eden、Survivor或者Tenured的任意区域。

*region英 /ˈriːdʒən/ n. 地区；范围；块 | Tenured美 /ˈtenjərd/ 年老代*

G1特点

1、G1能充分利用多CPU、多核环境的硬件优势，尽量缩短STW。

2、G1整体上采用的是标记-整理算法，局部通过复制算法，基本不会产生内存碎片。

3、宏观上，G1不再区分年轻代和老年代，他将内存划分成多个独立的子区域(region)，类似于围棋的棋盘。

4、G1依然需要在小范围内进行年轻代和老年代的区分，他们分别对应一部分region的集合，而且这些集合不需要是连续的；

5、G1只有逻辑上的分代概念，每个region都可能随着G1的运行在年轻代和老年代之间进行切换。虽然G1宏观上不再区分年轻代和老年代，但依然保留了这两个概念，只不过他们不再像以前一样是物理隔离的。也就是说依然会采用不同的GC方式来处理不同的区域。

G1关键概念

* **Region**

G1里面的Region的概念不同于传统的垃圾回收算法中的分区的概念。G1默认把堆内存分为1024个分区，后续垃圾收集的单位都是以Region为单位的。Region是实现G1算法的基础，每个Region的大小相等，通过-XX:G1HeapRegionSize参数可以设置Region的大小。一个Region在某一个时刻是Eden，在另一个时刻就可能属于老年代。G1在进行垃圾清理的时候就是将一个Region的对象拷贝到另外一个Region中。

* **SATB**

SATB的全称是Snapchat-At-The-Beginning。SATB是维持并发GC的一种手段。G1并发的基础就是SATB。SATB可以理解成在GC开始之前对堆内存里的对象做一次快照，此时活的对象就认为是活的，从而形成一个对象图。在GC收集的时候，新生代的对象也认为是活的对象，除此之外其他不可达的对象都认为是垃圾对象。

问题1：如何找到在GC的过程中分配的对象呢：每个region记录着两个top-at-mark-start（TAMS）指针，分别为prevTAMS和nextTAMS。在TAMS以上的对象就是新分配的，因而被视为隐式marked。通过这种方式我们就找到了在GC过程中新分配的对象，并把这些对象认为是活的对象。

问题2：如何解决GC过程中引用发生变化的问题：G1给出的解决办法是通过Write Barrier。Write Barrier就是对引用字段进行赋值做了环切。通过Write Barrier就可以了解到哪些引用对象发生了什么样的变化。

* **RSet**

RSet全称是Remember Set，每个Region中都有一个RSet，记录的是其他Region中的对象引用本Region对象的关系(谁引用了我的对象)。G1里面还有另外一种数据结构就Collection Set(CSet)，CSet记录的是GC要收集的Region的集合，CSet里的Region可以是任意代的。在GC的时候，对于old->young和old->old的跨代对象引用，只要扫描对应的CSet中的RSet即可。

* **停顿预测模型**

G1收集器突出表现出来的一点是通过一个停顿预测模型来根据用户配置的停顿时间来选择CSet的大小，从而达到用户期待的应用程序暂停时间。

通过-XX:MaxGCPauseMillis参数来设置。这一点有点类似于ParallelScavenge收集器。关于停顿时间的设置并不是越短越好。设置的时间越短意味着每次收集的CSet越小，导致垃圾逐步积累变多，最终不得不退化成Serial GC；停顿时间设置的过长，那么会导致每次都会产生长时间的停顿，影响了程序对外的响应时间。

* **存活对象拷贝(Evacuation)**

G1垃圾回收分为两个阶段：1、全局并发标记阶段(Global Concurrent marking)，下面细讲；2、拷贝存活对象阶段(evacuation)。

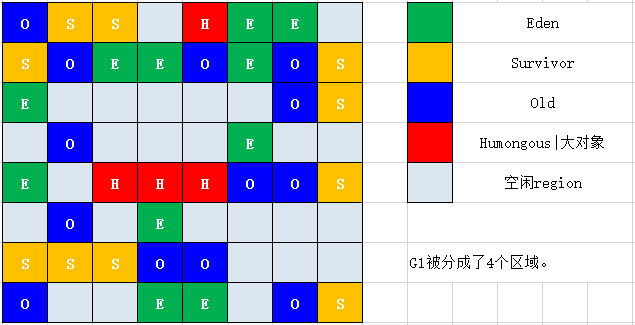
Evacuation阶段是全暂停的。该阶段把一部分Region里的活对象拷贝到另一部分Region中，从而实现垃圾的回收清理。Evacuation阶段从第一阶段选出来的Region中筛选出任意多个Region作为垃圾收集的目标，这些要收集的Region叫CSet，通过RSet实现。筛选出CSet之后，G1将并行的将这些Region里的存活对象拷贝到其他Region中，这点类似于ParalledScavenge的拷贝过程，整个过程是完全暂停的。关于停顿时间的控制，就是通过选择CSet的数量来达到控制时间长短的目标。

底层原理

G1的运行过程是这样的，会在Young GC和Mix GC之间不断的切换运行，同时定期的做全局并发标记，在实在赶不上回收速度的情况下使用Full GC(Serial GC)。初始标记是搭在YoungGC上执行的，在进行全局并发标记的时候不会做Mix GC，在做Mix GC的时候也不会启动初始标记阶段。当MixGC赶不上对象产生的速度的时候就退化成Full GC，这一点是需要重点调优的地方。

region区域化垃圾收集器最大的好处是：化整为零，避免全内存扫描，只需要按照区域来进行扫描即可。其核心思想是将整个堆内存区域分成大小相同的子区域(region)，在JVM启动时会自动设置这些子区域的大小；在堆的使用上，G1并不要求对象的存储一定是物理上连续的，只要逻辑上连续即可；每个分区也不会就是固定的某个代（比如这个分区就是新生代，或者这个分区就是老年代，不是这样固定的，是随着G1运行而产生变化的）；

启动的时候可以通过参数：-XX:G1HeapRegionSize=n，来指定每个分区的大小（1MB到32MB之间，且必须是2的幂），最多划分成2048个分区。也即能够支持的最大内存为：32MB \* 2048 = 65536MB = 64G内存。G1算法将堆划分成为若干个区域，如下图：



从上文的表述中可以看到有一个问题，即我们将每个region划分成为最大32M的一个区域，但是如果我们其中的一个对象所占用的内存区域大于32M怎么处理？G1非常聪明，他专门针对大对象进行联合处理，将多个region块儿联合起来处理，直到满足这个大对象需要的空间为止。

G1算法虽然将堆划分成若干个区域，但他仍然属于分代收集器。这些region的一部分包含新生代，新生代的垃圾收集依然采用暂停所有应用线程的方式，将存活线程拷贝到老年代或者Survivor空间。还有一部分region包含老年代，G1收集器通过将对象从一个区域复制到另一个区域，来完成清理工作；这就意味着在正常的处理过程中，G1完成了堆的压缩工作（至少是部分堆的压缩），从而避免了CMS造成内存碎片的问题。

在G1中，还有一个特殊的区域，叫做Humongous区(美 /hjuːˈmʌŋɡəs/ adj. 巨大无比的，极大的)，即：大对象。如果一个对象的空间占用了分区空间的50%以上，G1收集器就会认为这是一个巨型对象。这些大对象会默认被直接分配到老年代，但是如果他是一个短期存在的对象，这就会对垃圾收集器造成负面影响；为了解决这个问题，G1划分出了一个新的区：Humongous，用来专门存放巨型对象。G1会寻找连续的H分区来存储这些大对象。为了能够找到连续的H区，有时候不得不启动Full GC。就像餐厅的拼桌，人少的桌让一让去别的桌，人多的顾客团体将几张桌子拼到一起。

G1垃圾回收步骤

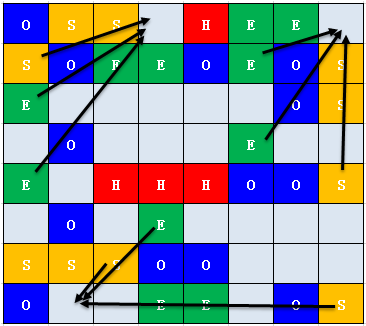
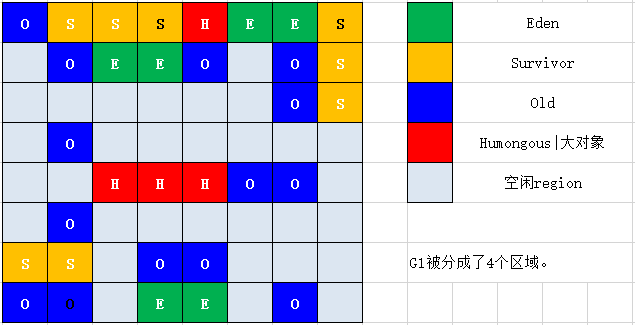
针对Eden区进行收集，Eden区耗尽后会被触发，主要是小区域收集 + 形成连续的内存块儿，避免内存碎片。

1、Eden区的数据移动到Survivor区，假如出现Survivor区空间不够的情况，Eden区的数据会被晋升到Old区。

2、Survivor区的数据移动到新的Survivor区，部分数据会晋升到Old区。

3、最后Eden去清理完成，GC行为结束，用户的应用程序继续执行。

其简要发生过程如下图所示：

G1运行的4步过程

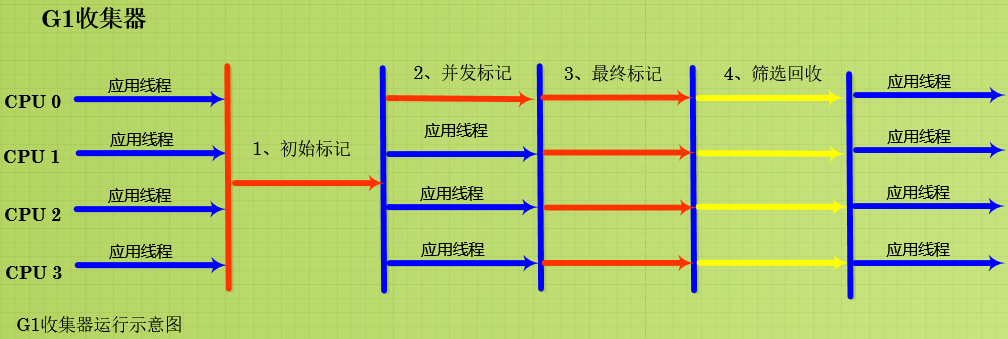
初始标记：只标记GC Roots能直接关联到的对象。

并发标记：进行GC Roots Tracing的过程。

最终标记：修正并发标记期间，因程序运行导致标记发生变化的那一部分对象。

筛选回收：根据时间来进行价值最大化的回收。

如下图所示：



五种垃圾回收器对比

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | 参数 | 新生代垃圾回收器 | 新生代算法 | 老年代垃圾收集器 | 老年代算法 |
| 串行垃圾回收器 | -XX:+UseSerialGC | Serial GC | 复制 | Serial Old GC | 标记整理 |
| 并行垃圾回收器 | -XX:+UseParNewGC | ParNew | 复制 | Serial Old GC | 标记整理 |
| 并行垃圾回收器 | -XX:UseParallelGC | Parallel[Scavenge] | 复制 | Parallel Old | 标记整理 |
| CMS垃圾回收器 | -XX:+UseConcMarkSweepGC | ParNew | 复制 | CMS+Serial Old(担保机制) | 标记清除 |
| G1垃圾回收器 | -XX:+UseG1GC | G1 | 标记整理 | G1 | 标记整理 |

G1最佳实践

性能优化与调配

* **不断调优暂停时间指标**

通过XX:MaxGCPauseMillis=x可以设置启动应用程序暂停的时间，G1在运行的时候会根据这个参数选择CSet来满足响应时间的设置。一般情况下这个值设置到100ms或者200ms都是可以的(不同情况下会不一样)，但如果设置成50ms就不太合理。暂停时间设置的太短，就会导致出现G1跟不上垃圾产生的速度。最终退化成Full GC。所以对这个参数的调优是一个持续的过程，逐步调整到最佳状态。

* **不要设置新生代和老年代的大小**

G1收集器在运行的时候会调整新生代和老年代的大小。通过改变代的大小来调整对象晋升的速度以及晋升年龄，从而达到我们为收集器设置的暂停时间目标。设置了新生代大小相当于放弃了G1为我们做的自动调优。我们需要做的只是设置整个堆内存的大小，剩下的交给G1自己去分配各个代的大小。

* **关注Evacuation Failure**

Evacuation Failure类似于CMS里面的晋升失败，堆空间的垃圾太多导致无法完成Region之间的拷贝，于是不得不退化成Full GC来做一次全局范围内的垃圾收集。

G1常见配置参数

|  |  |
| --- | --- |
| 参数名称 | 含义 |
| -XX:+UseG1GC | 使用 G1 垃圾收集器 |
| -XX:MaxGCPauseMillis=200 | 设置期望达到的最大GC停顿时间指标（JVM会尽力实现，但不保证达到） |
| -XX:InitiatingHeapOccupancyPercent=45 | 启动并发GC周期时的堆内存占用百分比. G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表示”一直执行GC循环”. 默认值为 45. |
| -XX:NewRatio=n | 新生代与老生代(new/old generation)的大小比例(Ratio). 默认值为 2. |
| -XX:SurvivorRatio=n | eden/survivor 空间大小的比例(Ratio). 默认值为 8. |
| -XX:MaxTenuringThreshold=n | 提升年老代的最大临界值(tenuring threshold). 默认值为 15. |
| -XX:ParallelGCThreads=n | 设置垃圾收集器在并行阶段使用的线程数,默认值随JVM运行的平台不同而不同. |
| -XX:ConcGCThreads=n | 并发垃圾收集器使用的线程数量. 默认值随JVM运行的平台不同而不同. |
| -XX:G1ReservePercent=n | 设置堆内存保留为假天花板的总量,以降低提升失败的可能性. 默认值是 10. |
| -XX:G1HeapRegionSize=n | 使用G1时Java堆会被分为大小统一的的区(region)。此参数可以指定每个heap区的大小. 默认值将根据 heap size 算出最优解. 最小值为 1Mb, 最大值为 32Mb. |

G1日志详细解读

// 新生代GC

[GC pause (G1 Humongous Allocation) (young) (initial-mark), 0.0035356 secs] // 初始标记，耗时0.0035秒

[Parallel Time: 2.4 ms, GC Workers: 8] // 并行8个线程，耗时2.4ms

[GC Worker Start (ms): Min: 813.1, Avg: 813.7, Max: 813.9, Diff: 0.7]

[Ext Root Scanning (ms): Min: 0.0, Avg: 1.1, Max: 1.5, Diff: 1.5, Sum: 9.1] // 每个扫描root的线程耗时

// 更新RS的耗时，G1中每块区域都有一个RS与之对应，RS记录了该区域被其他区域引用的对象。回收时，就把RS作为根集的一部分，从而加快回收

[Update RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]

[Processed Buffers: Min: 0, Avg: 0.0, Max: 0, Diff: 0, Sum: 0] // Processed Buffers就是记录引用变化的缓存空间

[Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0] // 扫描RS

[Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0] // 根扫描耗时

[Object Copy (ms): Min: 0.0, Avg: 0.5, Max: 1.3, Diff: 1.3, Sum: 3.6] // 对象拷贝

[Termination (ms): Min: 0.0, Avg: 0.2, Max: 0.2, Diff: 0.2, Sum: 1.2]

[Termination Attempts: Min: 1, Avg: 1.8, Max: 4, Diff: 3, Sum: 14]

[GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.1]

[GC Worker Total (ms): Min: 1.6, Avg: 1.8, Max: 2.3, Diff: 0.8, Sum: 14.1] // GC线程耗时

[GC Worker End (ms): Min: 815.4, Avg: 815.4, Max: 815.4, Diff: 0.0]

[Code Root Fixup: 0.0 ms]

[Code Root Purge: 0.0 ms]

[Clear CT: 0.1 ms] // 清空CardTable耗时，RS是依赖CardTable记录区域存活对象的

[Other: 1.1 ms]

[Choose CSet: 0.0 ms] // 选取CSet

[Ref Proc: 0.9 ms] // 弱引用、软引用的处理耗时

[Ref Enq: 0.0 ms] // 弱引用、软引用的入队耗时

[Redirty Cards: 0.1 ms]

[Humongous Register: 0.0 ms]

[Humongous Reclaim: 0.0 ms]

[Free CSet: 0.0 ms] // 释放被回收区域的耗时（包含他们的RS）

[Eden: 5120.0K(24.0M)->0.0B(12.0M) Survivors: 0.0B->2048.0K Heap: 16.0M(50.0M)->12.4M(50.0M)]

[Times: user=0.01 sys=0.00, real=0.01 secs]

// 根区域扫描

[GC concurrent-root-region-scan-start]

[GC concurrent-root-region-scan-end, 0.0012422 secs]

// 并发标记

[GC concurrent-mark-start]

[GC concurrent-mark-end, 0.0004063 secs]

// 重新标记又叫最终标记

[GC remark [Finalize Marking, 0.0003736 secs] [GC ref-proc, 0.0000533 secs] [Unloading, 0.0007439 secs], 0.0013442 secs]

[Times: user=0.00 sys=0.00, real=0.00 secs]

// 独占清理

[GC cleanup 13M->13M(50M), 0.0004002 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

这是一段完整的GC日志。从整体上看，并发标记周期和混合回收的前后都有可能穿插着新生代GC。其中并发标记周期主要是回收老年代空间，当然也包含了一次新生代GC。

三色标记算法



**黑色**：自身和成员变量均已标记完成

**灰色**：自身被标记完成，成员变量未被标记

**白色**：未被标记的对象（垃圾）

**漏标情况**：会将本来不是垃圾得对象，当作垃圾回收了。

在运行态中，1、黑色对象指向了白色对象，2、并且同时，灰色对象取消了白色对象的引用。

就会产生白色对象漏标的情况。因为黑色对象已经标记完毕，不会再去扫描，并且也没有灰色对象的指向，所以漏标了

即：在并发标记时，引用可能产生变化，白色对象有可能被错误回收。

* 漏标的应对，打破这两种条件之一即可：

1、Incremental update 增量更新，关注引用的增加，如果发现黑色指向了白色，把黑色重新标记为白色，remark过程将重新扫描属性。但是会造成重复扫描已扫描过的属性。（CMS对漏标的处理方式）

2、SATB snapshot at the begining 关注引用的删除，当灰色对象对白色对象的引用删除后，将这个引用推到GC的堆栈中，保证白色对象还是能够被扫描到（G1对漏标的处理方式）在扫描时拿到这个引用，由于有Rset存在，不需要扫描整个堆去查找指向白色的引用，效率比较高。SATB配合Rset，使得整个效率提高

CMS使用的是三色标记+Incremental Update算法

G1使用的是三色标记+snapshot at the begining （SATB）算法

ZGC： Colored Pointers+写屏障

Shenandoah ：Colored Pointers+读屏障

ZGC原理与实现分析

ZGC有别于上述所有的GC，是一种全新的收集器。目标：支持TB级堆内存（最大4T）；最大GC停顿10ms；对吞吐量影响最大不超过15%；在SPECjbb 2015基准测试中，128G堆内存，单次GC停顿最大1.68ms, 平均1.09ms。

https://www.jianshu.com/p/4e4fd0dd5d25

**线上故障排查与定位**

**异常代码模拟**

**public** **class** LinuxCheck { //后面的代码都将以此为例模拟。

**public** **static** **void** main(String[] args) {

**while**(**true**) {

System.***out***.println(**new** java.util.Random().nextInt(289289));

}

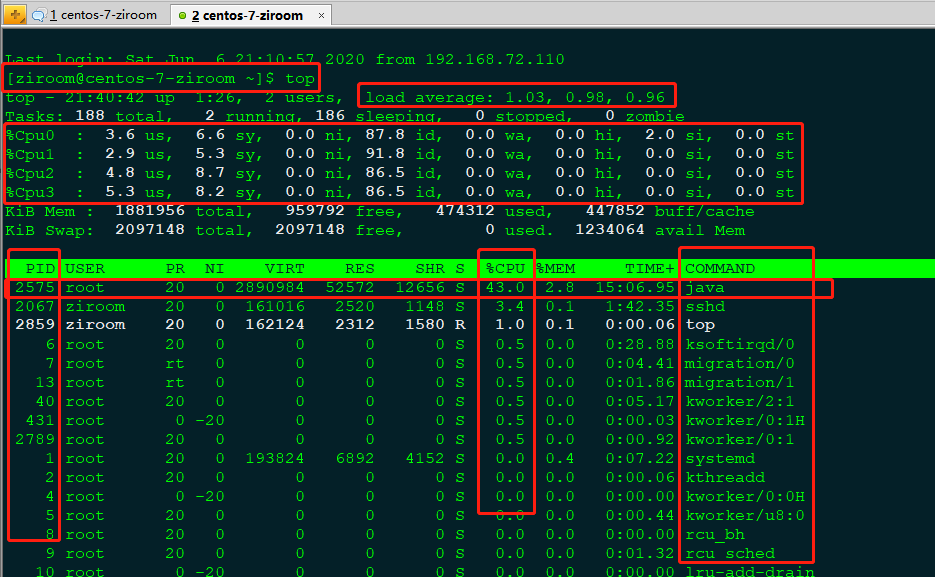
}

}

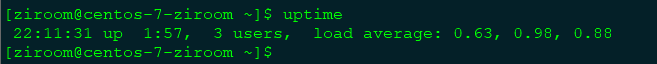
**线上常用Linux命令**

**1、Top命令查看整机运行情况**

Top命令类似于Windows的资源管理器，直接在命令行输入top命令。top命令还有个功能，在键盘上按“1”这个键可以看到每个cpu的详细情况，前提是你的CPU是处在多核心状态下的。



当我们运行了LinuxCheck.java后，代码会进入死循环的状态，不断在后台打印出信息；此时虚拟机系统会明显感觉反应变慢，输入top命令后能够简单的反馈出系统运行的整体信息。我们通常更关心CPU使用占比情况、PID、COMMAND和load average（系统负载均衡）。load average：1.03, 0.98， 0.96这三个值的的含义如下：系统1分钟、5分钟和15分钟，系统的平均负载值；这三个值相加的和除以逻辑CPU数量再乘以100%，如果计算结果高于60%，那么系统的负载压力过重。计算：(1.03 + 0.98 + 0.96) / 4 \* 100% = 74.25%；在top命令之外，如果你只想查看load average，可以使用uptime命令：



|  |
| --- |
| 面板信息详细解读 |
| 第一行，任务队列信息，同 uptime 命令的执行结果 |
| top – 21:40:42 up 1:26, 2 users, load average: 1.03, 0.98, 0.96 |
| 21:40:42代表系统时间；up 1:26代表运行时间1分26秒；当前登录用户：2；load average负载均衡(uptime) |
|  |
| 第二行，Task – 任务(进程) |
| Tasks: 188 total, 2 running, 186 sleeping, 0 stopped, 0 zombie |
| 总进程188；2个进程运行；186个进程休眠；0个进程停止；僵尸进程数：0个 |
|  |
| 第三行，CPU状态信息 |
| %Cpu0 : 3.6 us, 6.6 sy, 0.0 ni, 87.8 id, 0.0 wa, 0.0 hi, 2.0 si, 0.0 st |
| us【user space】用户空间占用CPU的百分比；sy【sysctl】内核空间占用CPU的百分比；ni【】改变过优先级的进程占用CPU的百分比；  id【idolt】空闲CPU百分比；wa【wait】IO等待占用CPU的百分比；hi【Hardware IRQ】硬中断占用CPU的百分比；  si【Software Interrupts】软中断占用CPU的百分比 |
|  |
| 第四行，内存状态 |
| Kib Mem : 1881956 total, 959792 free, 474312 used, 447852 buff/cache |
| 内存共计(total)：1881956Kb ≈ 1.97G；使用了(used)：474312Kb ≈ 463Mb；当前空闲(free)：959792Kb ≈ 937Mb；  缓存的内存量(buff/cache)：447852Kb ≈ 437M |
|  |
| 第五行，swap交换分区信息 | 这里需要重点关注，加个小星星！！ |
| Kib swap : 2097148 total, 2097148 free, 0 used, 1234064 avail Mem |
| avail Mem：缓冲区的交换总量。2097148Kb ≈ 1.999996G。 |

**1.1备注（此处针对第四行-内存状态 和 第五行-swap交换分区信息进行特殊说明）：**

1、可用内存 = free + buffer + cached。

2、对于内存监控，在top里我们要时刻监控第五行swap交换分区的used，如果这个数值在不断的变化，说明内核在不断进行内存和swap的数据交换，

代表内存不够用了。

3、第四行中使用中的内存总量（used）指的是现在系统内核控制的内存数。

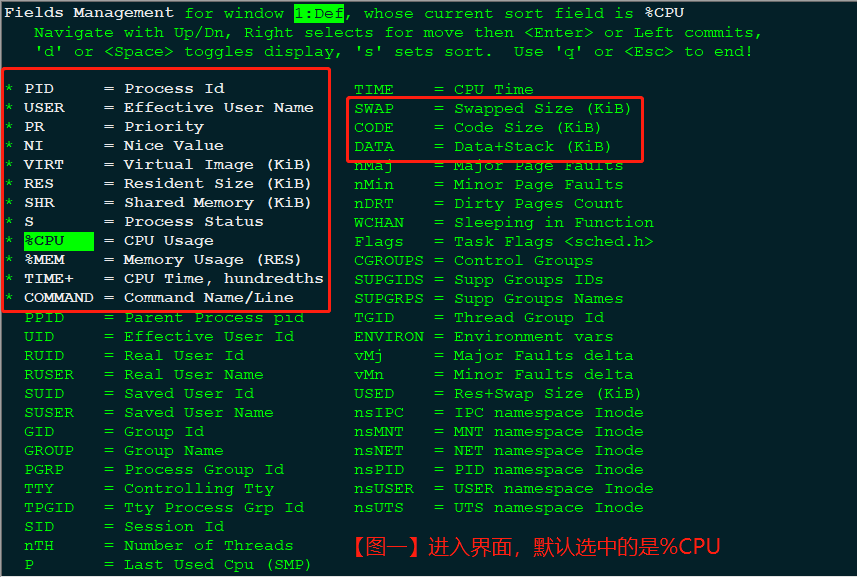
4、第四行中空闲内存总量（free）是内核还未纳入其管控范围的数量。

5、纳入内核管理的内存不见得都在使用中，还包括过去使用过的现在可以被重复利用的内存，内核并不把这些可被重新使用的内存交还到free中去，

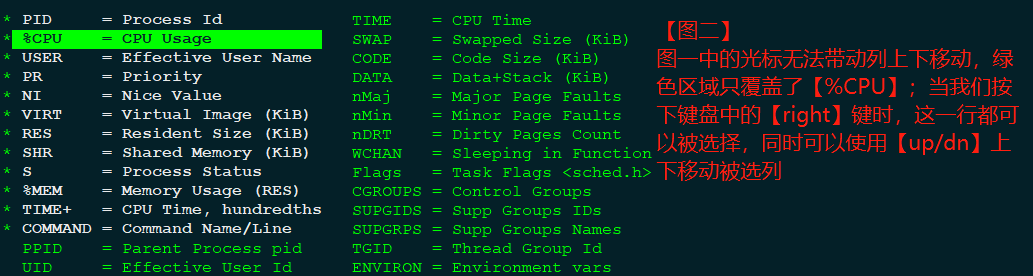
因此在linux上free内存会越来越少，但不用为此担心。

**1.2 Top命令其他参数**

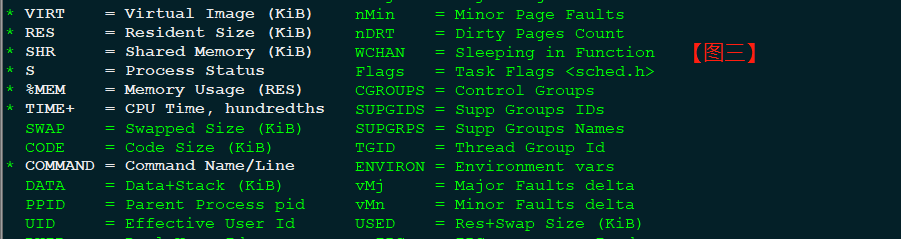
从上图【各进程的状态监控】面板中我们可以看到从“PID”到“COMMAND”的所有信息，但“DATA”就没有显示出来；实际上这个面板中还隐藏了其他的信息，只是他在top中没有显示出来，我们可以按下【f】键来让他显示：



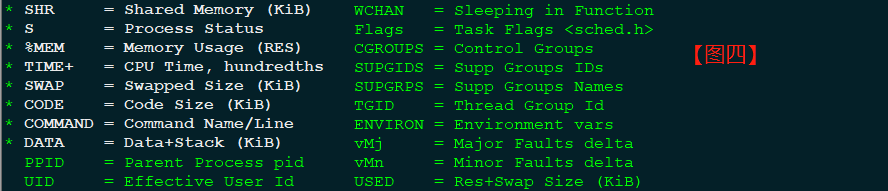
通过up和down可以选中某个字段，\*表示选中展示的列，可以通过空格或d来进行切换显示或隐藏，按选中后按右箭头可以选中该行，再执行up或down可以调整该字段的展示顺序位置，按q或esc键可以退出当前操作。举个列子：



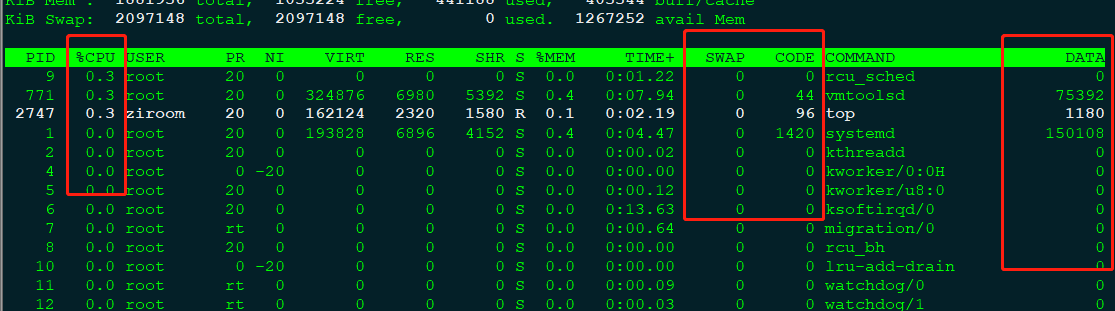
依照此方法，我们将【SWAP】【CODE】【DATA】三个列分别移动到【COMMAND】上和下。



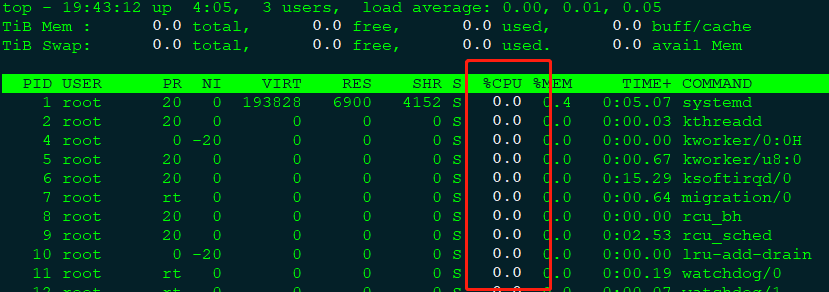
我们下一步通过【空格】或【d】键来显示这三个被选列，注意如果整行是被选中的状态是无法使用“显示/隐藏”功能的，即图二中的状态。



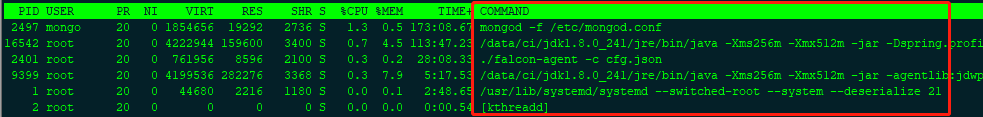
这里故意将【SWAP】和【CODE】放到了【COMMAND】之前，【DATA】放到了后面；然后我们按下【Esc】退出，查看新的界面。



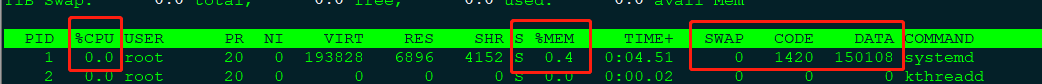
按下键盘的【x】键可以高亮显示%CPU这列，如下图：



按下【c】键，COMMAND可以显示详细路径，如下图：



**1.3 各进程（任务）的状态监控**



PID — 进程id。

USER — 进程所有者。

PR — 进程优先级。

NI — nice值。负值表示高优先级，正值表示低优先级。

VIRT — 进程使用的虚拟内存总量，单位kb。VIRT = SWAP + RES。

RES — 进程使用的、未被换出的物理内存大小，单位kb。RES = CODE + DATA。

SHR — 共享内存大小，单位kb。

S — 进程状态。D = 不可中断的睡眠状态 R = 运行 S = 睡眠 T = 跟踪/停止 Z = 僵尸进程。

%CPU — 上次更新到现在的CPU时间占用百分比。

%MEM — 进程使用的物理内存百分比。

TIME+ — 进程使用的CPU时间总计，单位1/100秒。

COMMAND — 进程名称（命令名/命令行）。

**VIRT：virtual memory usage 虚拟内存**。

1、进程“需要的”虚拟内存大小，包括进程使用的库、代码、数据等

2、假如进程申请100m的内存，但实际只使用了10m，那么它会增长100m，而不是实际的使用量

**RES：resident memory usage 常驻内存。**

1、进程当前使用的内存大小，但不包括swap out。

2、包含其他进程的共享。

3、如果申请100m的内存，实际使用10m，它只增长10m，与VIRT相反。

4、关于库占用内存的情况，它只统计加载的库文件所占内存大小。

**SHR：shared memory 共享内存。**

1、除了自身进程的共享内存，也包括其他进程的共享内存。

2、虽然进程只使用了几个共享库的函数，但它包含了整个共享库的大小。

3、计算某个进程所占的物理内存大小公式：RES – SHR。

4、swap out后，它将会降下来。

**DATA**

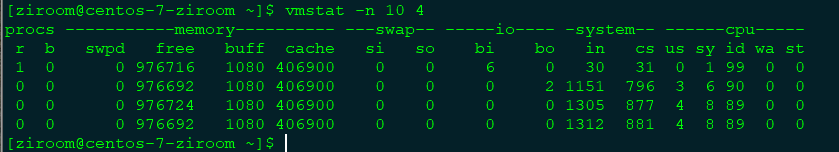
1、数据占用的内存。

2、真正的该程序要求的数据空间，是真正在运行中要使用的。

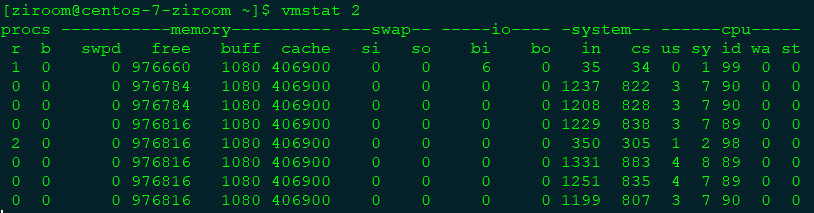
**2、CPU运行情况：vmstat**

**2.1 基本使用方式**

相比top这个命令更加详细的反应出了CPU的信息(包含但不限于查看CPU信息)，vmstat可以看到整个机器的CPU、内存和IO的使用情况，而不是单单看到各个进程的CPU使用率和内存使用率。这个命令的使用方法比较简单，常见的有2种方式来使用，第一种：vmstat -n 10 4；这条命令的含义是：每10秒钟采样一次，共计采样4次后结束。



另一种比较常见的使用方式是：vmstat 2；这条命令的含义是：每2秒钟采样一次，一直执行，直到命令被中断为止。



**2.2 vmstat命令详解**

这里开始介绍vmstat命令的高级用法。

vmstat [-a] [-n] [-S unit] [delay [ count]]

vmstat [-s] [-n] [-S unit]

vmstat [-m] [-n] [delay [ count]]

vmstat [-d] [-n] [delay [ count]]

vmstat [-p disk partition] [-n] [delay [ count]]

vmstat [-f]

vmstat [-V]

-a：显示活跃和非活跃内存。

-f：显示从系统启动至今的fork数量 。

-m：显示slabinfo

-n：只在开始时显示一次各字段名称。

-s：显示内存相关统计信息及多种系统活动数量。

delay：刷新时间间隔。如果不指定，只显示一条结果。

count：刷新次数。如果不指定刷新次数，但指定了刷新时间间隔，这时刷新次数为无穷。

-d：显示磁盘相关统计信息。

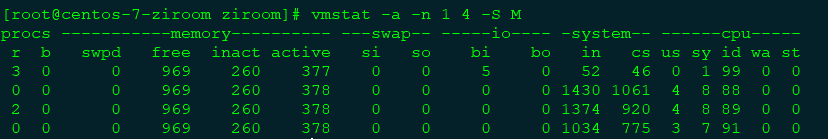
-p：显示指定磁盘分区统计信息

-S：使用指定单位显示。参数有 k 、K 、m 、M ，分别代表1000、1024、1000000、1048576字节（byte）。默认单位为K（1024 bytes）

-V：显示vmstat版本信息。

**2.3 使用范例**

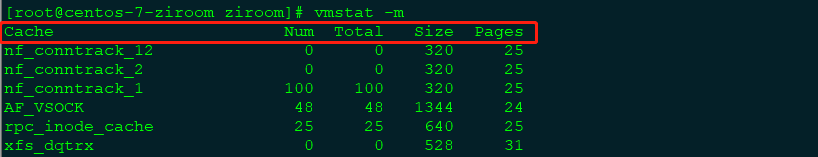
* **vmstat -a -n 1 4 -S M** 显示活跃和非活跃内存，每1秒钟采样一次，共计采样4次，显示单位使用M



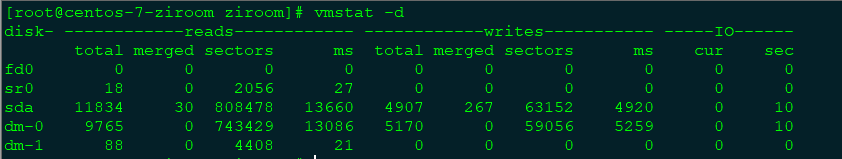
inact: 非活跃内存大小（当使用-a选项时显示）

active: 活跃的内存大小（当使用-a选项时显示）

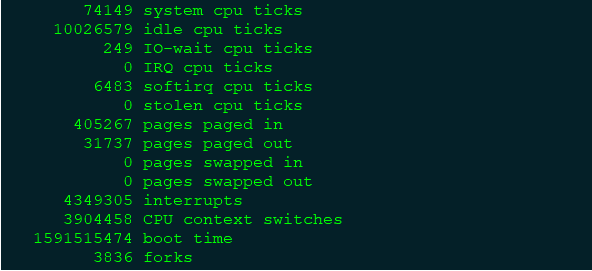
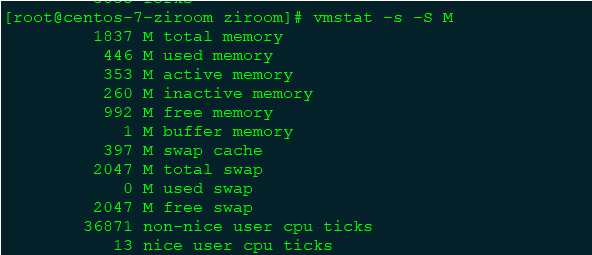
* **vmstat -m**查看系统的slab信息



* **vmstat -d** 查看磁盘信息



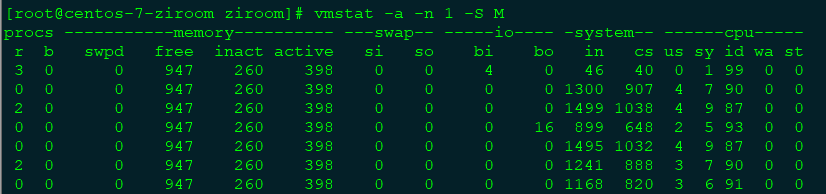
* **vmstat -s -S M** 查看内存使用的详细信息



* **vmstat -f 检查forks(复制)次数**



**2.4 vmstat 面板参数说明**



* **procs分组**

**r**：运行队列中的进程数量(runtime)；

这个值也可以判断是否需要增加CPU，原则上1核的CPU他的运行队列不要超过2，整个系统的运行队列不能超过总核心数的2倍，否则代表压力过大。

b：等待资源的进程数(blocking)；

比如正在等待磁盘IO、网络IO等。

根据图中数据，虚机分配4核心，日志面板中的最大值为3，小于8；3 + 2 + 2 = 7 / 7 = 1，平均值为1，没有超过2；所以系统运行依旧正常。

* **cpu分组**

**us**：用户进程消耗CPU时间百分比(user time)；us值高，用户进程消耗CPU时间多，如果长期大于50%则需要优化程序。

**sy**：内核进程消耗CPU时间百分比；sy的值高时，说明系统内核消耗的CPU资源多，这并不是良性表现，我们应该检查原因。

**id**：即idle，空闲时间百分比。此值越大越好。

wa：IO等待时间百分比。wa的值高时，说明IO等待比较严重，这可能由于磁盘大量作随机访问造成，也有可能磁盘出现瓶颈（块操作）。

st：来自于一个虚拟机偷取CPU时间的百分比。

这其中us + sy的参考值为80%，如果大于80%则说明存在CPU不足，处于高占用状态。参考图中的最大数值，4 + 9 = 13，远低于80。

* **memory分组**

swpd：使用虚拟内存大小。如果swpd的值不为0，但是SI，SO的值长期为0，这种情况不会影响系统性能。

free：空闲物理内存大小。

buff：用作缓冲的内存大小。

cache：用作缓存的内存大小。如果加入[-a]命令，则不会显示cache。如果cache的值大的时候，说明cache处的文件数多，如果频繁访问到的文件都能被cache处，那么磁盘的读IO bi会非常小。

* **swap分组**

si：每秒从交换区写到内存的大小，由磁盘调入内存。

so：每秒写入交换区的内存大小，由内存调入磁盘。

注意：内存够用的时候，这2个值都是0，如果这2个值长期大于0时，系统性能会受到影响，磁盘IO和CPU资源都会被消耗。有些朋友看到空闲内存（free）很少的或接近于0时，就认为内存不够用了，不能光看这一点，还要结合si和so，如果free很少，但是si和so也很少（大多时候是0），那么不用担心，系统性能这时不会受到影响的。

* **io分组**

bi：每秒读取的块数。

bo：每秒写入的块数。

注意：随机磁盘读写的时候，这2个值越大（如超出1024k)，能看到CPU在IO等待的值也会越大。

* **system分组**

in：每秒中断数，包括时钟中断。

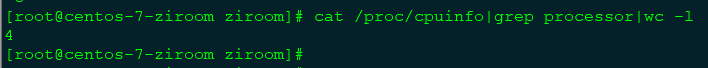
cs：每秒上下文切换数。

注意：上面2个值越大，会看到由内核消耗的CPU时间会越大。

**2.5 服务器性能监控主要度量依据**

目前说来，对于服务器监控有用处的度量主要有：r、us、sy、id。注意：如果r经常大于4，且id经常少于40，表示cpu的负荷很重。如果bi，bo 长期不等于0，表示内存不足。当us＋sy的值接近100的时候，表示CPU正在接近满负荷工作。但要注意的是，CPU 满负荷工作并不能说明什么，Linux总是试图要CPU尽可能的繁忙，使得任务的吞吐量最大化。唯一能够确定CPU瓶颈的还是r（运行队列）的值。

Linux下查看CPU核心数的命令：cat /proc/cpuinfo|grep processor|wc -l 。



当r值超过了CPU个数，就会出现CPU瓶颈，解决办法大体几种：

1、最简单的就是增加CPU个数和核数。

2、通过调整任务执行时间，如大任务放到系统不繁忙的情况下进行执行，进尔平衡系统任务。

3、调整已有任务的优先级。

* **关于memory分组中的swpd虚拟内存问题**。

数据库服务器都只有有限的RAM，出现内存争用现象是Oracle的常见问题。当内存的需求大于RAM的数量，服务器启动了虚拟内存机制，通过虚拟内存，可以将RAM段移到SWAP DISK的特殊磁盘段上，这样会 出现虚拟内存的页导出和页导入现象，页导出并不能说明RAM瓶颈，虚拟内存系统经常会对内存段进行页导出，但页导入操作就表明了服务器需要更多的内存了， 页导入需要从SWAP DISK上将内存段复制回RAM，导致服务器速度变慢。

这种情况到目前为止尚未遇到过，偶然机会看到了这个案例，故加入此处作为参考；其解决办法有如下几种：

1、加到RAM；

2、改小SGA，使得对RAM需求减少；

3、减少PGA，从而减少RAM的需求；

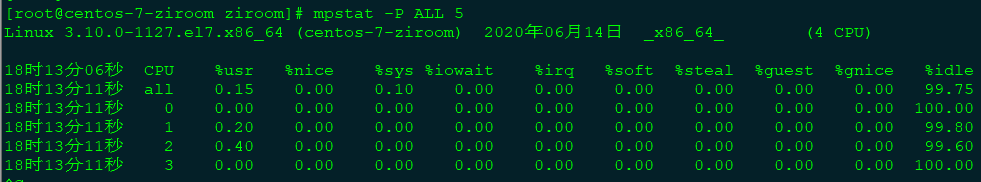
**2.6 其他查看CPU的方式 mpstat**

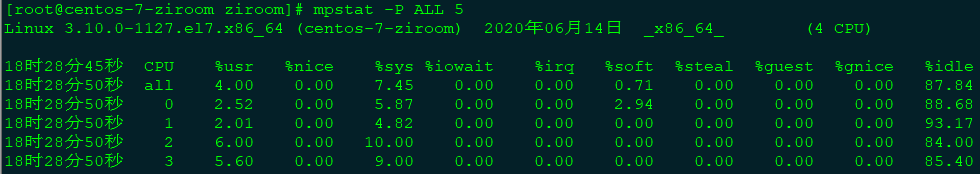
mpstat是 Multiprocessor(/ˌmʌltiˈprəʊsesə(r)/ n 多重处理器) Statistics的缩写，是实时系统监控工具。其报告与CPU的一些统计信息，这些信息存放在/proc/stat文件中。在多CPU系统里，其不但能查看所有CPU的平均状况信息，而且能够查看特定CPU的信息。mpstat的语法：mpstat [-P {cpu|ALL}] [internal [count]]。

其中，各参数含义如下：

|  |  |
| --- | --- |
| 参数 | 含义 |
| -P {cpu|ALL} | **表示监控哪个CPU， cpu在[0,cpu个数-1]中取值。** |
| internal | **相邻的两次采样的间隔时间。** |
| count | **采样的次数，count只能和delay一起使用。** |

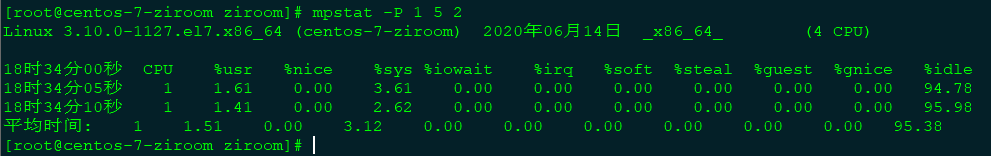
* **使用示例1**：mpstat -P ALL 5

 此图为无Java运行模式下



此图为模拟异常代码的模式

* **使用示例2**：mpstat -P 1 5 2；查看序号为1的CPU，每5秒采样一次，一共采样2次。



* **vmstat和mpstat 命令的差别**：

mpstat 可以显示每个处理器的统计，而 vmstat 显示所有处理器的统计。因此，编写糟糕的应用程序（不使用多线程体系结构）可能会运行在一个多处理器机器上，而不使用所有处理器。从而导致一个 CPU 过载，而其他 CPU 却很空闲。通过 mpstat 可以轻松诊断这些类型的问题。

**2.7 CPU用量分解信息 pidstat**

pidstat是sysstat工具的一个命令，用于监控全部或指定进程的cpu、内存、线程、设备IO等系统资源的占用情况。可以根据进程ID快速定位问题所在。

pidstat的用法：pidstat [ 选项 ] [ <时间间隔> ] [ <次数> ]。

常用的参数：

-u：默认的参数，显示各个进程的cpu使用统计。

-r：显示各个进程的内存使用统计。

-d：显示各个进程的IO使用情况。

-p：指定进程号。

-w：显示每个进程的上下文切换情况。

-t：显示选择任务的线程的统计信息外的额外信息。

-T { TASK | CHILD | ALL }

这个选项指定了pidstat监控的。TASK表示报告独立的task，CHILD关键字表示报告进程下所有线程统计信息。ALL表示报告独立的task和task下面的所有线程。注意：task和子线程的全局的统计信息和pidstat选项无关。这些统计信息不会对应到当前的统计间隔，这些统计信息只有在子线程kill或者完成的时候才会被收集。

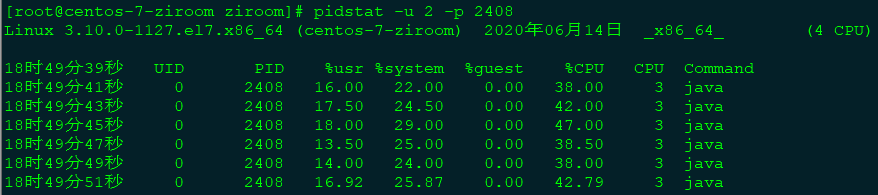
-V：版本号

-h：在一行上显示了所有活动，这样其他程序可以容易解析。

-I：在SMP环境，表示任务的CPU使用率/内核数量。

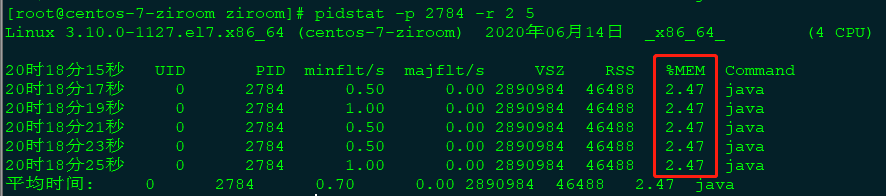
-l：显示命令名和所有参数。

* **使用示例1 查看CPU**：pidstat -u 2 -p 2408；对于2408号线程，每2秒钟采样一次。

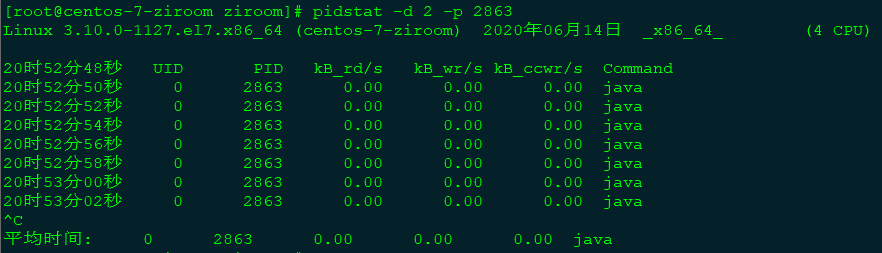


从图中可以看到，2408号线程都运行在了3号CPU上。因为LinuxCheck.java并不是多线程的应用程序，所以通常来讲这样的代码会运行在同一个CPU上，方便我们定位到问题所在。

* **使用示例2查看内存**：pidstat -p 2784 -r 2 5；对于2784号线程，每2秒钟采样一次，共计采样5次，同时显示内存使用大小。

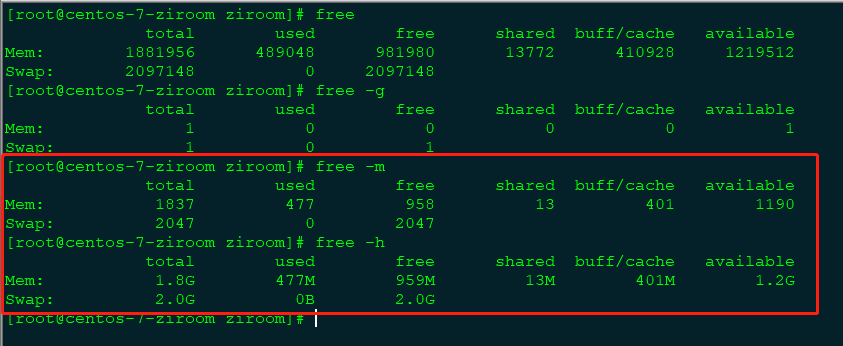


* **使用示例3查看磁盘IO**：pidstat -d 2 -p 2863；每2秒查看一次2863号线程的磁盘IO使用情况。

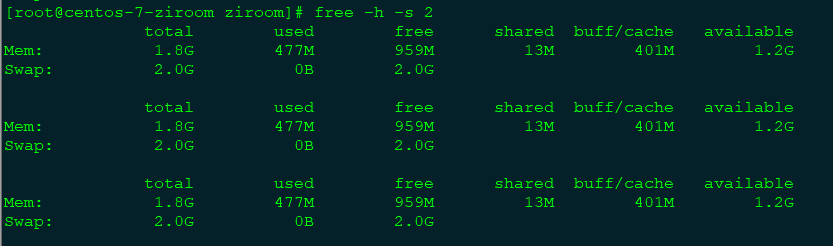


**3、内存占用情况：free**

free 命令显示系统内存的使用情况，包括物理内存、交换内存(swap)和内核缓冲区内存。如果加上 -h 选项，输出的结果会友好很多：



在实际开发中，我们通常会使用free -m 或 free -h来查看内存的使用情况。有时我们需要持续的观察内存的状况，此时可以使用 -s 选项并指定间隔的秒数：



上图为每2秒钟采样一次。

其面板输出含义如下：

Mem 行(第二行)是内存的使用情况。

Swap 行(第三行)是交换空间的使用情况。

total 列显示系统总的可用物理内存和交换空间大小。

used 列显示已经被使用的物理内存和交换空间。

free 列显示还有多少物理内存和交换空间可用使用。

shared 列显示被共享使用的物理内存大小。

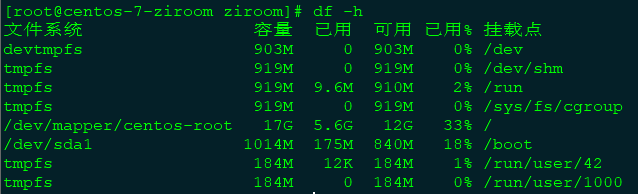
buff/cache 列显示被 buffer 和 cache 使用的物理内存大小。

available 列显示还可以被应用程序使用的物理内存大小。

**4、硬盘 | 磁盘IO | 网络IO命令**

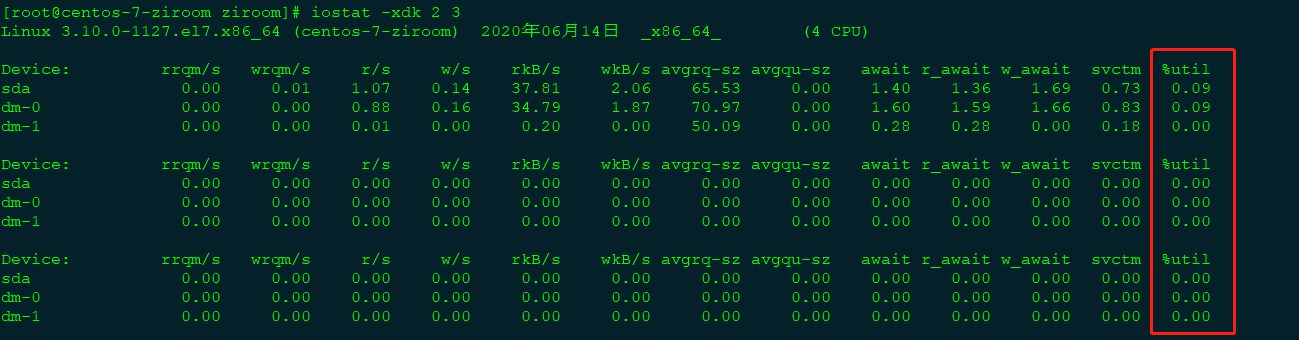
**4.1、硬盘使用情况：df**

df -h(-h代表：human 以人类能看懂的方式显示)



**4.2、磁盘IO情况：iostat**

在生产环境中，大概率导致一个系统运行缓慢的2个主要原因之一(另一个原因是CPU负载过高)。通常使用iostat -xdk命令来实现，下图每2秒采样一次，共采样三次。评估磁盘的重要依据是【%util】列。还有一种方式是使用上述的ipstat命令也可以定位磁盘IO的使用情况。



其面板输出含义如下：

rrqm/s 每秒对该设备的读请求被合并次数，文件系统会对读取同块(block)的请求进行合并

wrqm/s 每秒对该设备的写请求被合并次数

r/s 每秒完成的读次数

w/s 每秒完成的写次数

rkB/s 每秒读数据量(kB为单位)

wkB/s 每秒写数据量(kB为单位)

avgrq-sz 平均每次IO操作的数据量(扇区数为单位)

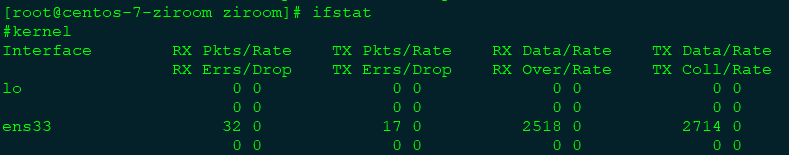
avgqu-sz 平均等待处理的IO请求队列长度

await 平均每次IO请求等待时间(包括等待时间和处理时间，毫秒为单位)；值越小性能越好。

svctm 平均每次IO请求的处理时间(毫秒为单位)；值越小性能越好。

%util 一秒中有百分之几的时间用于I/O操作。当数据值接近100%的时候，磁盘宽带跑满，需要优化程序或者增加磁盘。

**4.3、网络IO情况：ifstat**



**综合实例：生产环境故障定位CPU占用过高**

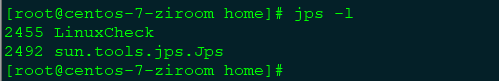
**1、Top命令定位CPU占比最高的程序**



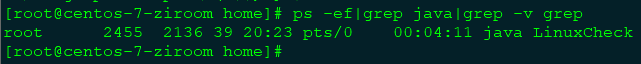
从图中可以看出，PID为2455的**进程**，CPU占比达到26.6%。

**2、jps或ps -ef进一步定位，找到是哪个后台程序在出问题**

**2.1 方式1：jsp命令定位**

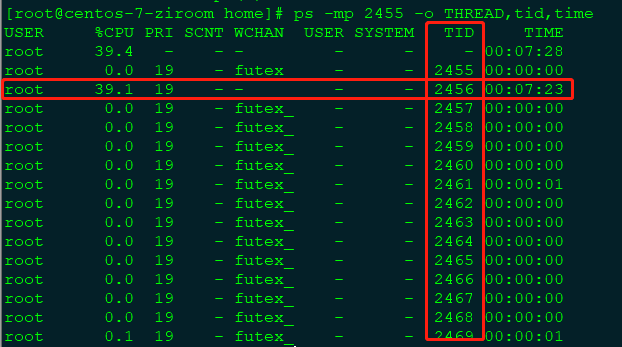


**2.2 方式2：ps -ef命令定位**



**3、定位到具体的线程和代码行**

一个进程由多个线程组成，故问题还是要定位到具体的线程上。首先我们使用ps -mp [进程ID] -o THREAD, tid, time 命令；注意这里用的tid，因为我们要从进程定位到线程，故这里从pid变成了tid。



命令显示，2456号线程CPU占比最高，故他有问题。

ps -mp [进程ID] -o THREAD, tid, time 命令参数示意：-m显示所有的线程；-p pid进程使用cpu的时间；-o该参数后是用户自定义格式。

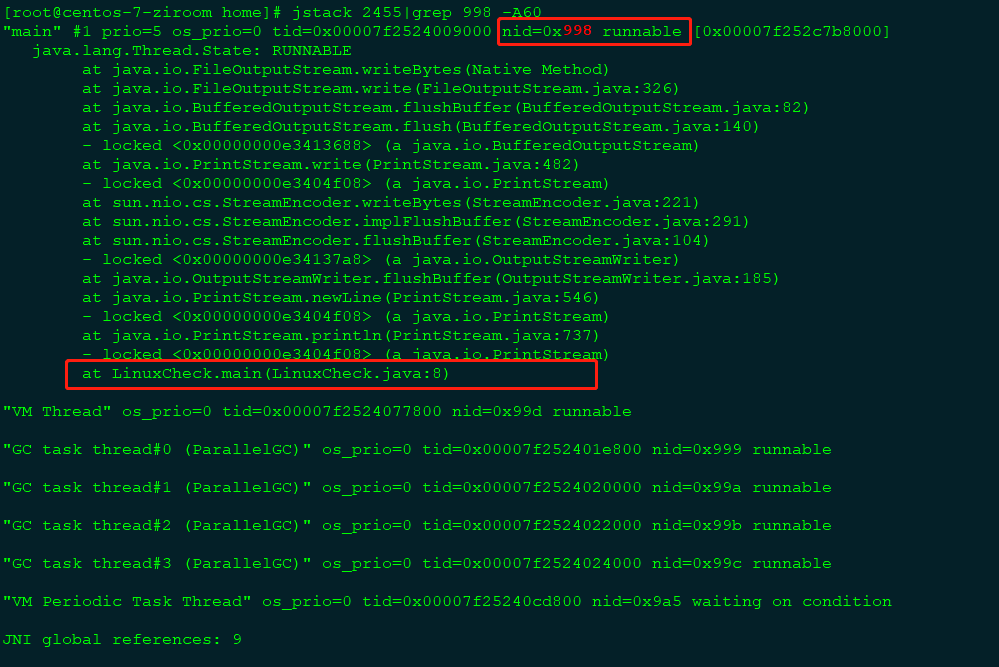
这其中：-o THREAD, tid, time代表我要展示【线程】的tid和执行时间。

**4、线程ID转换为16进制英文小写**

因为线程在内存中是以16进制来标识和运行的，故这里要将10进制转换为16进制：整数2456转化为16进制等于998。

**5、jstack定位**

基本命令为：jstack [进程ID] | grep tid(16进制线程id小写英文) -A60；-A60：打印出前60行。



第一个红框：nid=0x998 runnable，这个线程运行中。

第二个红框：“at LinuxCheck.main(LinuxCheck.java:8)”代表问题在LinuxCheck.java第八行，我们打开其源代码：

