久

杨

珍

精

悟尽归原 几近于道。

create by Yangcl

2018-03-15

1 Mysql基础篇

2 Mysql高级特性

2.1数据库索引与高性能查询

2.1.1 MySQL索引5大原则

**A：最左前缀匹配原则(重要的原则)**

mysql会一直向右匹配直到遇到范围查询（>、<、between、like）就停止匹配。比如a = 1 and b = 2 and c > 3 and d = 4，如果建立（a,b,c,d）顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。 一个查询中，如果第一个查询条件为索引，同时因为使用了>、<、between或like导致他失效了，那么索引整体失效。

**B：=和in可以乱序**

比如a = 1 and b = 2 and c = 3 建立（a,b,c）索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式。

**C：尽量选择区分度高的列作为索引**

区分度的公式是count(distinct column)/count(\*)。表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录。

**D：索引列不能参与计算，保持列“干净”**

比如from\_unixtime(create\_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create\_time = unix\_timestamp(‘2014-05-29’);

**E：尽量的扩展索引，不要新建索引**

比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。

2.1.2 索引使用的典型场景(使用索引优化查询)

A：匹配全值

对索引中所有列都指定具体值，即对索引中的所有列都有等值匹配的条件。

#设置组合索引（rental\_date,inventory\_id,customer\_id）为唯一索引。

EXPLAIN

SELECT \* FROM

rental

WHERE rental\_date = '2005-05-25 17:22:10'

AND inventory\_id = 373

AND customer\_id = 343 ;



B：匹配值的范围查询

对索引值进行范围查找，这种情况最常见。

#设置索引idx\_fk\_customer\_id(customer\_id)

EXPLAIN

SELECT

\*

FROM

rental

WHERE customer\_id >= 373

AND customer\_id < 400 ;



**C：匹配最左前缀**

仅仅使用索引中的最左边列进行查询。比如组合索引（col1,col2,col3）能够被col1，col1+col2，col1+col2+col3的等值查询利用到的。

#创建索引idx\_payment\_date(payment\_date,amount,last\_update);

EXPLAIN

SELECT \* FROM

payment

WHERE payment\_date = '2006-02-14 15:16:03'

AND last\_update = '2006-02-15 22:12:32' ;



从结果可以看出利用了索引，但又row为182行，所有只使用了部分索引。换一种方式，如下脚本，将不适用索引，请注意标红色的amount条件

EXPLAIN

SELECT \* FROM

payment

WHERE **amount** = 3.98

AND last\_update = '2006-02-15' ;



从结果看出，这次查询没有利用索引，进行了全表查找。因为我们所创建的索引中没有包含amount参数。

**D：所查询列全为索引**

当查询列都在索引字段中。即select中的列都在索引中。

#创建索引idx\_payment\_date(payment\_date,amount,last\_update);

EXPLAIN

SELECT

last\_update

FROM

payment

WHERE payment\_date = '2005-08-19 21:21:47'

AND amount = 4.99 ;



Extra内容显示为Using index,说明不需要通过索引回表，Using index就是平时说的覆盖索引扫描（即找到索引，就找到了要查询的结果，不用再回表查找了）。

**E：匹配列前缀**

仅仅使用索引的第一列，并且只包含索引第1列的开头部分进行查找。

#创建索引idx\_title\_desc\_part(title(10)，description(20));

EXPLAIN

SELECT

title

FROM

film\_text

WHERE title LIKE 'african%' ;



type=range，范围扫描，基于索引做范围扫描，为诸如BETWEEN，IN，>=，LIKE类操作提供支持。

为提高like的查询效率，请注意创建索引的代码中红色的部分，title(10)大意为匹配前10个字符。大表查询中极少使用LIKE这个关键字。

**F：索引部分等值匹配，部分范围匹配**

EXPLAIN

SELECT

inventory\_id

FROM

rental

WHERE rental\_date = '2006-02-14 15:16:03'

AND customer\_id >= 300

AND customer\_id <= 400 ;



type=ref，说明使用了索引。

**G：列名是索引，column\_name is null，使用索引**

EXPLAIN

SELECT

\*

FROM

payment

WHERE rental\_id IS NULL ;



2.1.3索引存在但不能使用索引的典型场景

**A：以%开头的like查询**

EXPLAIN

SELECT

\*

FROM

actor

WHERE last\_name LIKE '%NI%' ; **#百分号“%”只能在最右的情况索引才不会失效！**



**B：数据类型出现隐式转化，不会使用索引**

EXPLAIN

SELECT

\*

FROM

actor

WHERE last\_name = 1 ;



last\_name为varchar类型存储，但匹配值为int类型的数字，此时索引失效；正确如下：WHERE last\_name = '1' ;

**C：组合索引，不满足最左原则，不使用符合索引**

详见2.1.1 A中的说明。

**D：用索引比全表扫描还慢，则不要使用索引**

如查询以“S”开头的标题的电影，返回记录比例比较大，mysql预估索引扫描还不如全表扫描。

**E：用or分割条件，若or前后只要有一个查询条件的列没有索引，就都不会用索引**

EXPLAIN

SELECT

\*

FROM

payment

WHERE customer\_id = 203

OR amount = 3.96 ;



2.1.4 mysql explain结果中type的含义

|  |  |
| --- | --- |
| type值 | 含义 |
| system | CONST的特例，当表上只有一条元组匹配 |
| const | WHERE条件筛选后表上至多有一条元组匹配时，比如WHERE ID = 2 （ID是主键，值为2的要么有一条要么没有） |
| eq\_ref | 参与连接运算的表是内表（在代码实现的算法中，两表连接时作为循环中的内循环遍历的对象，这样的表称为内表）。  基于索引（连接字段上存在唯一索引或者主键索引，且操作符必须是“=”谓词，索引值不能为NULL）做扫描，使得对外表的一条元组，内表只有唯一一条元组与之对应。 |
| ref | 可以用于单表扫描或者连接。参与连接运算的表，是内表。基于索引（连接字段上的索引是非唯一索引，操作符必须是“=”谓词，连接字段值不可为NULL）做扫描，使得对外表的一条元组，内表可有若干条元组与之对应。 |
| ref\_or\_null | 类似REF，只是搜索条件包括：连接字段的值可以为NULL的情况，比如 where col = 2 or col is null |
| range | 范围扫描，基于索引做范围扫描，为诸如BETWEEN，IN，>=，LIKE类操作提供支持 |
| index\_scan | 索引做扫描，是基于索引在索引的叶子节点上找满足条件的数据（不需要访问数据文件） |
| all | 全表扫描或者范围扫描：**不使用索引**，顺序扫描，直接读取表上的数据（访问数据文件） |
| unique\_subquery | 在子查询中，基于唯一索引进行扫描，类似于EQ\_REF |
| index\_subquery | 在子查询中，基于除唯一索引之外的索引进行扫描 |
| index\_merge | 多重范围扫描。两表连接的每个表的连接字段上均有索引存在且索引有序，结果合并在一起。适用于作集合的并、交操作。 |
| ft | FULL TEXT，全文检索 |

你定义的索引在查询时是否生效取决于这里展示的type的值。

2.2数据库索引深层原理

2.1节从实用性角度总结了在开发中用到的索引知识点，尤其在重构某个功能或者项目模块的时候显得尤为重要。但是并没有从一个深层的角度来认识索引的本质。

这一节中将汇集所有优秀的文章，加以总结：什么是索引的本质。

2.2.1索引的数据结构及算法基础

**A：索引的本质**

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。提取句子主干，就可以得到索引的本质：索引是一种数据结构。数据库查询是数据库的主要功能之一，最基本的查询算法是顺序查找（linear search）时间复杂度为O(n)，显然在数据量很大时效率很低。优化的查找算法如二分查找（binary search）、二叉树查找（binary tree search）等，虽然查找效率提高了。但是各自对检索的数据都有要求：二分查找要求被检索数据有序，而二叉树查找只能应用于二叉查找树上，但是数据本身的组织结构不可能完全满足各种数据结构（例如，理论上不可能同时将两列都按顺序进行组织）。所以，在数据之外，数据库系统还维护着满足特定查找算法的数据结构。这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构就是索引。



上图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)的复杂度内获取到相应数据。虽然这是一个货真价实的索引，但是实际的数据库系统几乎没有使用二叉查找树或其进化品种红黑树（red-black tree）实现的，原因会在下文介绍。

**B：B-Tree**

为了描述B-Tree，首先定义一条数据记录为一个二元组[key, data]，key为记录的键值，对于不同数据记录，key是互不相同的；data为数据记录除key外的数据。那么B-Tree是满足下列条件的数据结构：

1. d>=2，即B-Tree的度；
2. h为B-Tree的高；
3. 每个非叶子结点由n-1个key和n个指针组成，其中d<=n<=2d；
4. 每个叶子结点至少包含一个key和两个指针，最多包含2d-1个key和2d个指针，叶结点的指针均为NULL；
5. 所有叶结点都在同一层，深度等于树高h；
6. key和指针相互间隔，结点两端是指针；
7. 一个结点中的key从左至右非递减排列；
8. 如果某个指针在结点node最左边且不为null，则其指向结点的所有key小于v(key1)，其中v(key1)为node的第一个key的值。
9. 如果某个指针在结点node最右边且不为null，则其指向结点的所有key大于v(keym)，其中v(keym)为node的最后一个key的值。
10. 如果某个指针在结点node的左右相邻key分别是keyi和keyi+1且不为null，则其指向结点的所有key小于v(keyi+1)且大于v(keyi)。



由于B-Tree的特性，在B-Tree中按key检索数据的算法非常直观：首先从根节点进行二分查找，如果找到则返回对应节点的data，否则对相应区间的指针指向的节点递归进行查找，直到找到节点或找到null指针，前者查找成功，后者查找失败。B-Tree上查找算法的伪代码如下：

BTree\_Search(node, key) {

if(node == null) return null;

foreach(node.key)

{

if(node.key[i] == key) return node.data[i];

if(node.key[i] > key) return BTree\_Search(point[i]->node);

}

return BTree\_Search(point[i+1]->node);

}

data = BTree\_Search(root, my\_key);

关于B-Tree有一系列有趣的性质，例如一个度为d的B-Tree，设其索引N个key，则其树高h的上限为logd((N+1)/2)，检索一个key，其查找结点个数的渐进复杂度为O(logdN)。从这点可以看出，B-Tree是一个非常有效率的索引数据结构。

https://www.cnblogs.com/tgycoder/p/5410057.html

**C：B+Tree**

2.2.2数据库事物隔离级别

https://www.cnblogs.com/wy697495/p/11025929.html

https://baijiahao.baidu.com/s?id=1611918898724887602&wfr=spider&for=pc

http://blog.codinglabs.org/articles/index-condition-pushdown.html

3 MyCat In Action

3.1 数据库集群基本术语

3.1.1 数据切分

Mycat的核心功能，通过某种特定的条件，将放在同一个数据库中的数据分散存放在多个数据库主机中，以达到分散到多台设备负载的效果。

数据切分有两种切分模式，一种是按照不同的表，将数据切分到不同的数据库主机中，这种切分方式称为数据的垂直切分，也叫纵向切分；另一种

则是根据表中的数据逻辑关系，将同一个表中的数据按照某种特定条件(比如：租户id)拆分到多个数据库主机中，这种方式称为数据的水平切分，也

叫做横向切分，当单表数据量达到800万条以上的时候，通常会对该表做横向切分。

字符串枚举分片

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。

**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。https://www.cnblogs.com/cmfwm/p/7671188.html

**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)。

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。并且这8种操作必须遵循以下规则：

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**：

1. compareAndSwapInt方法是原子性的。
2. 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。
3. 这里有JMM线程交互的影子。
4. getAndAddInt方法使用了自旋锁。
5. 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() );

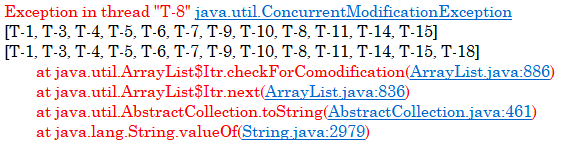
System.out.println(list);

}, 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() );

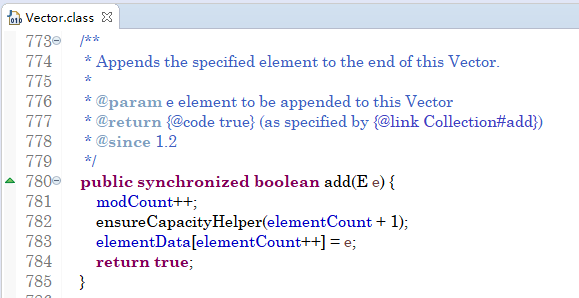
System.out.println(list);

}, 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() );

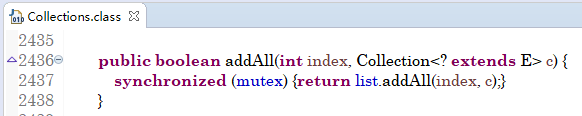
System.out.println(list);

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

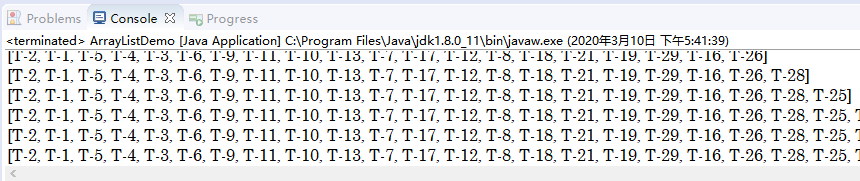
}

}

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. \*/

private transient volatile Object[] array; // 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。

/\*\*

\* 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();

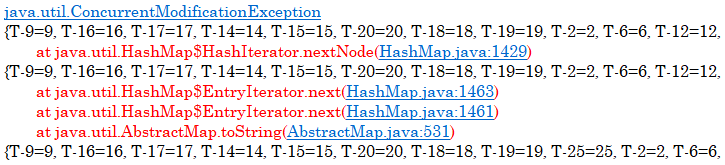
map.put(name , name.split("-")[1]);

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(() -> {

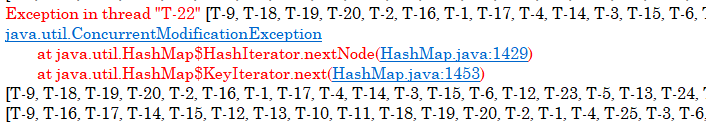
set.add( Thread.currentThread().getName() );

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】

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) {

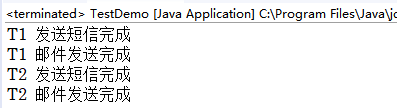
SynchronizedDemo sd = new SynchronizedDemo();

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");

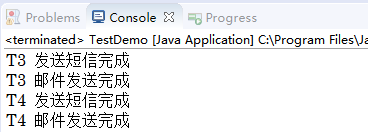
Thread thread4 = new Thread(rld, "T4");

thread3.start();

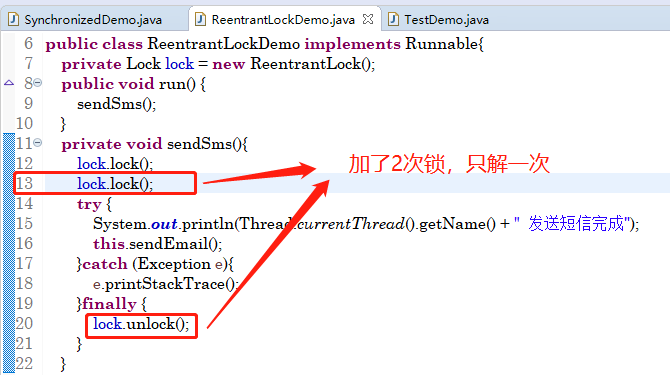
thread4.start();

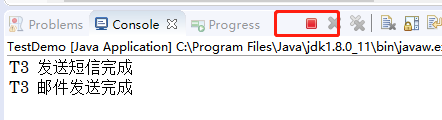
}

}



ReentrantLock 2层锁解一次



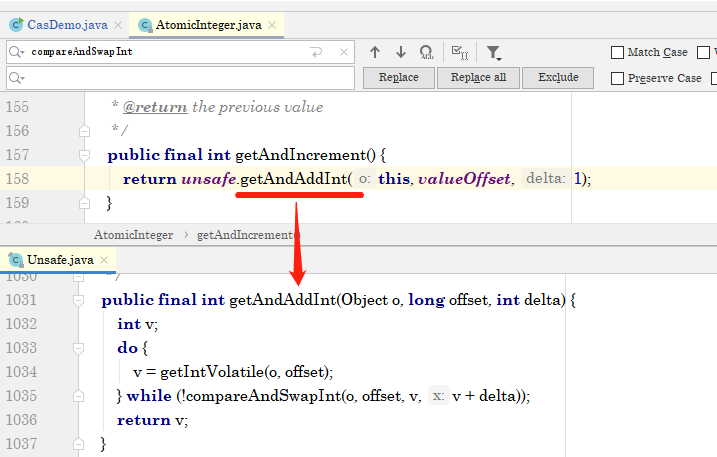


这种情况比较特殊，从运行表现上可以看到，T4线程并没有打印，这是因为T3线程还在等待。如果我们加锁2次，解锁也是2次，那么这种情况就不会发生。

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(() -> {

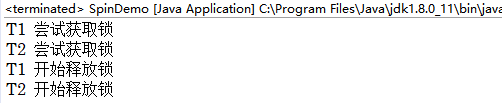
lock.lock();

lock.unlock();

} , "T2").start();

}

}



https://www.cnblogs.com/twoheads/p/10148598.html

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++) {

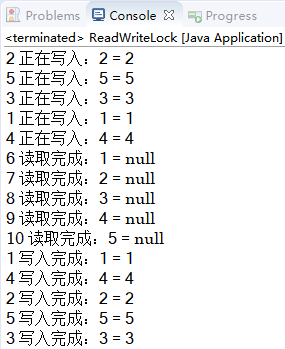
int temp = i - 5;

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(); }

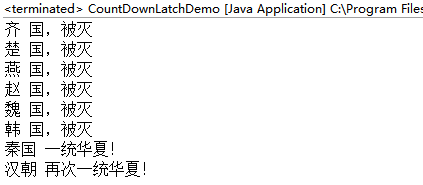
System.out.println(Thread.currentThread().getName() + " 再次一统华夏！");

}, "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();

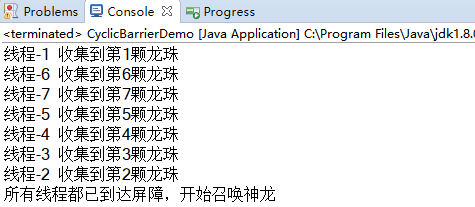
}

},"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();

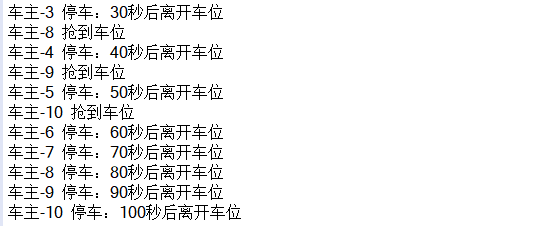
}

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

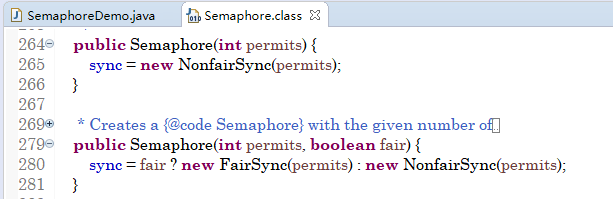
}

}

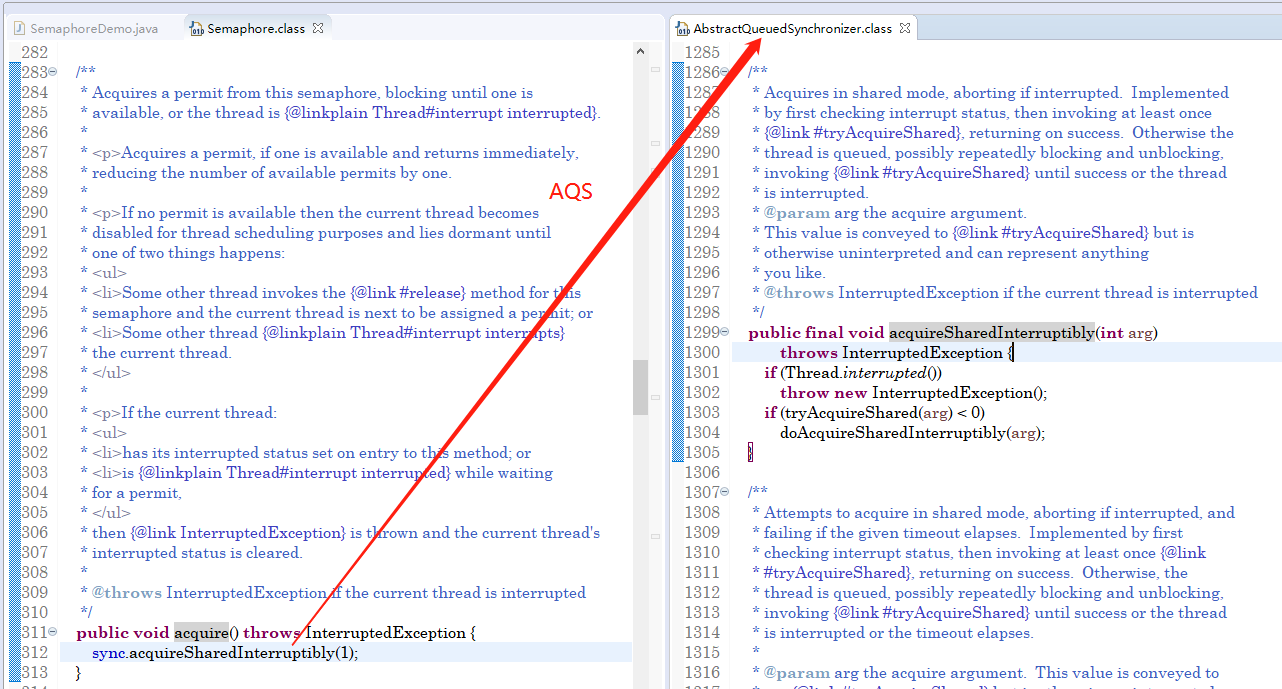
}

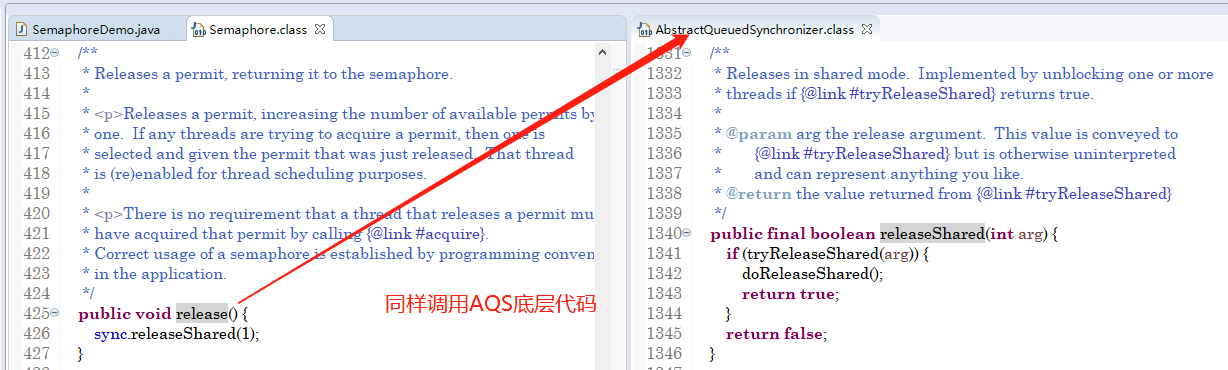
其底层实现：



对于acquire()方法，底层调用的是AQS。



对于release()方法：



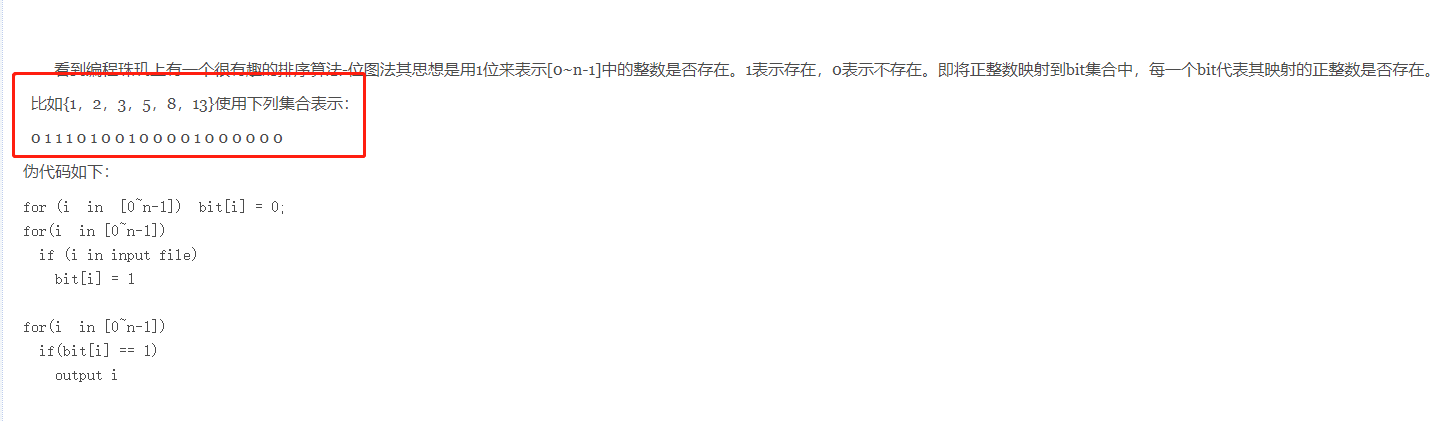
synchronized-线程锁

https://blog.csdn.net/zxd8080666/article/details/83214089



Java 位图法排序

https://www.cnblogs.com/Kingle/archive/2012/03/09/2994679.html



https://www.cnblogs.com/rrttp/p/7668773.html

https://blog.csdn.net/y999666/article/details/51220833

Java Virtual Machine实际应用

Html/Css/Javascript 篇

Servlet/J2EE篇

实用代码篇

* 1. Java代码段

1.1 十六进制和八进制

* 1. Javascript代码段
  2. Mysql代码段

实际问题收集

1 SpringMvc线程安全问题



springmvc是单例模式的框架,但它是线程安全的,因为springmvc没有成员变量,所有参数的封装都是基于方法的,属于当前线程的私有变量. 因此是线程安全的框架

关于java成员变量和局部变量

https://blog.csdn.net/weixin\_37012881/article/details/82699089



实例变量属于某个对象的属性，必须创建了实例对象，其中的实例变量才会被分配空间，才能使用这个实例变量。

https://blog.csdn.net/yong\_zi/article/details/81285049







2 ThreadLocal

https://www.jianshu.com/p/3c5d7f09dfbd

3 HashMap算法

4 Zookeeper

https://blog.csdn.net/qq\_36864672/article/details/78596499

5 Redis

https://mp.weixin.qq.com/s/f9N13fnyTtnu2D5sKZiu9w

6 SPI 技术

https://mp.weixin.qq.com/s/VrqOpyTTIzlyMAtJFda9fw

7 乐观锁和悲观锁



https://segmentfault.com/a/1190000015239603

JVM类的双亲加载

关键字transient是干啥的

https://blog.csdn.net/u010188178/article/details/83581506



AQS详解

https://blog.csdn.net/mulinsen77/article/details/84583716



秒杀系统设计

https://www.toutiao.com/i6802735487669764615/

内部类的使用场景和好处

https://www.runoob.com/w3cnote/java-inner-class-intro.html

代码样式

|  |  |  |
| --- | --- | --- |
| 方法名 | 返回值 | 功能描述 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
| 方法名 | 功能描述 |
|  |  |
|  |  |