并发编程实践

# 并发编程实践基础

## 线程安全

线程安全的概念：当多个线程访问一个类，这个类始终都能表现出正确的行为，那么这个类就是线程安全的。

Synchronized:可以加在任意对象或者方法上加锁，这个加锁的对象称为互斥区或者临界区。

**package** com.guigu.base.sysc001;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.sysc001.MyThread</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日上午8:48:04 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** MyThread **extends** Thread {

**private** **int** count = 5;

**public** **synchronized** **void** run() {

count--;

System.***out***.println(**this**.*currentThread*().getName() + ",count=" + count);

}

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

/\*\*

\* 分析： 当多个线程访问MyThread的run方法 以排队的方式进入（这里的排队是按照cpu分配的先后顺序而定）

\* 一个线程如果想要执行synchronized修饰的代码

\* 1.尝试获得锁

\* 2.如果拿到锁，执行synchronized代码的具体内容，拿不到锁，这个线程不会断的尝试获取这把锁，直到拿到为止

\* 而且是多个线程同时竞争这把锁， 也就说会有锁竞争的问题

\*

\*/

MyThread mt = **new** MyThread();

Thread t1 = **new** Thread(mt, "t1");

Thread t2 = **new** Thread(mt, "t2");

Thread t3 = **new** Thread(mt, "t3");

Thread t4 = **new** Thread(mt, "t4");

Thread t5 = **new** Thread(mt, "t5");

t1.start();

t2.start();

t3.start();

t4.start();

t5.start();

}

}

## 多个线程多个锁

多个线程多个锁，每个线程拿到自己指定的锁，分别获取到锁之后再执行synchronized 方法的具体内容。

**package** com.guigu.base.sysc001;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.sysc001.MultiThread</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日上午8:59:15 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** MultiThread {

/\*\*

\* 1.关键字synchronized获取的都是对象锁， 而不是一段代码(方法)， 当做锁

\* 2.代码中哪个线程先执行synchronized关键字的方法，哪个多线程就持有该方法所有对象的锁。

\* 3.静态方法上加上synchronized关键字 代表锁定类的字节码.class 是类一级的锁 独占.class类

\*

\*/

**private** **static** **int** *num* = 0;

**public** **static** **synchronized** **void** printNum(String msg) {

**try** {

**if** (msg.equals("a")) {

*num* = 100;

System.***out***.println("msg a ,set msg over");

Thread.*sleep*(1000);

} **else** {

*num* = 200;

System.***out***.println("msg b,set msg over");

}

System.***out***.println("msg " + msg + ",msg=" + *num*);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

// 注意观察run方法输出顺序

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

// 定义两个不同的对象

MultiThread m1 = **new** MultiThread();

MultiThread m2 = **new** MultiThread();

//两个线程两个对象都能进入那个printNum方法

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

m1.*printNum*("a");

}

});

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

m2.*printNum*("b");

}

});

t1.start();

t2.start();

}

}

## 对象锁的同步和异步

同步synchronized

同步的概念就是共享，如果不是共享的资源就没有必要同步

异步:asynchronized

异步就是独立，相互之间不受任何限制，ajax发起请求，可以继续浏览网页或者其他操作。

同步的目的是保证线程安全，对于线程安全来说需要满足两个特性：

**原子性**

**可见性**

**package** com.guigu.base.sysc001;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.sysc001.MyObject</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日上午9:17:26 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** MyObject {

**public** **synchronized** **void** method1() {

**try** {

System.***out***.println(Thread.*currentThread*().getName());

Thread.*sleep*(4000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**public** **synchronized** **void** method2() {

System.***out***.println(Thread.*currentThread*().getName());

}

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

/\*\*

\* 分析：

\* t1线程先持有Object对象的Lock锁 ，t2线程是可以以异步的方式调用对象中的非synchronized修饰的方法

\* t1线程先持有Object对象的Lock锁 ,t2线程如果在这个时候调用对象同步synchronized方法则需要等待也就是同步。

\*

\*/

**final** MyObject mo = **new** MyObject();

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

mo.method1();

}

});

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

mo.method2();

}

});

t1.start();

t2.start();

}

}

## 脏读

**package** com.guigu.base.sysc002;

**public** **class** DirtyRead {

/\*\*

\* 业务需要保持完整性 保持业务的原子性

\*/

**private** String username = "guigu";

**private** String password = "123";

**public** **synchronized** **void** setValues(String username, String password) {

**this**.username = username;

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**this**.password = password;

System.***out***.println("setValues 的最终结果是:" + username + ",password" + password);

}

**public** **synchronized** **void** getValues() {

System.***out***.println("getValues方法得到的Username是：" + **this**.username + ",password:" + **this**.password);

}

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

**final** DirtyRead dr = **new** DirtyRead();

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

dr.setValues("liulaoshi", "456");

}

});

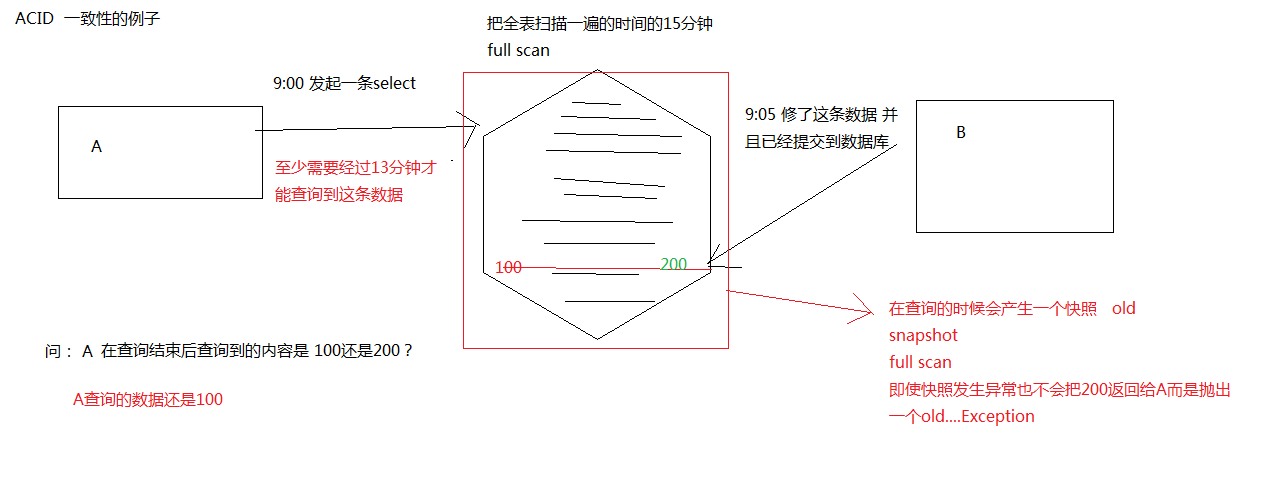
t1.start();

Thread.*sleep*(1000);

dr.getValues();

}

}



## Synchronized其他概念

Synchronized 锁重入：

关键字synchronized拥有锁重入的功能，也就是在使用synchronized时，当一个线程获取一个对象锁之后，再次请求次对象时是可以再次得到该对象的锁。

如果出现异常将自动释放锁。

**package** com.guigu.base.sysc002;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.sysc002.SyncDemo</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日上午10:15:27 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** SyncDemo {

**public** **synchronized** **void** method1() {

System.***out***.println("method1...");

method2();

}

**public** **synchronized** **void** method2() {

System.***out***.println("method2...");

method3();

}

**public** **synchronized** **void** method3() {

System.***out***.println("method3...");

}

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

SyncDemo sd =**new** SyncDemo();

Thread t1 =**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

sd.method1();

}

});

t1.start();

}

}

**package** com.guigu.base.sysc002;

**public** **class** SyncDemo2 {

**static** **class** Father{

**public** **int** i=10;

//锁重入

**public** **synchronized** **void** operSup() {

**try** {

i--;

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

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**static** **class** Child **extends** Father{

**public** **synchronized** **void** operSub() {

**while**(i>0) {

**try** {

i--;

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

Thread.*sleep*(100);

**this**.operSup();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

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

Thread t1 =**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

Child c =**new** Child();

c.operSub();

}

});

t1.start();

}

}

## Synchronized代码块

使用Synchronized会在某些情况下产生弊端，比如A线程调用同步方法执行一个很长时间的任务，那么B线程必须等到比较长时间才能执行，这种情况下可以去优化代码执行时间，也就是通常所说的**减少锁的粒度**。

### 通过减少锁的粒度

**package** com.guigu.base.sysc003;

**public** **class** Optimize {

**public** **void** doLongTimeTask() {

**try** {

System.***out***.println("当前线程开始执行..." + Thread.*currentThread*().getName() + "正在执行一个比较长时间的内容...");

Thread.*sleep*(5000);

**synchronized** (**this**) {

System.***out***.println("当前线程执行同步代码..." + Thread.*currentThread*().getName());

Thread.*sleep*(100);

}

System.***out***.println("当前线程执行结束...." + Thread.*currentThread*().getName() + "执行完毕...");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

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

**final** Optimize op = **new** Optimize();

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

op.doLongTimeTask();

}

}, "t1");

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

op.doLongTimeTask();

}

}, "t2");

t1.start();

t2.start();

}

}

### 对象锁 、类锁 、任意对象锁、减少锁的粒度

**package** com.guigu.base.sysc003;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.sysc003.ObjectLock</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日上午10:54:40 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** ObjectLock {

**public** **void** method1() {

**synchronized** (**this**) { // 对象锁

**try** {

System.***out***.println("do method1....");

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **void** method2() {

**synchronized** (ObjectLock.**class**) { // 类锁

**try** {

System.***out***.println("do method2....");

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**private** Object lock = **new** Object();

**public** **void** method3() {

**synchronized** (lock) { // 任意对象锁

**try** {

System.***out***.println("do method3....");

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

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

**final** ObjectLock objLock = **new** ObjectLock();

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

objLock.method1();

}

});

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

objLock.method2();

}

});

Thread t3 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

objLock.method3();

}

});

t1.start();

t2.start();

t3.start();

}

}

### 字符串常量锁

**package** com.guigu.base.sysc003;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.sysc003.StingLock</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日上午11:00:10 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** StringLock {

/\*\*

\* synchronized 代码块对字符串的锁 注意String常量池的缓存功能

\*/

**public** **void** method() {

**synchronized** (**new** String("guigu")) {

**try** {

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

System.***out***.println("当前线程：" + Thread.*currentThread*().getName() + "开始");

Thread.*sleep*(1000);

System.***out***.println("当前线程:" + Thread.*currentThread*().getName() + "结束");

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

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

**final** StringLock sl = **new** StringLock();

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

sl.method();

}

}, "t1");

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

sl.method();

}

}, "t2");

t1.start();

t2.start();

}

}

## Volatile关键字

Volatile关键字的主要作用是使变量在多个线程之间可见。

**package** com.guigu.base.sysc004;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.sysc004.RunThread</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日下午1:23:54 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** RunThread **extends** Thread {

**public** **volatile** **boolean** isRunning = **true**;

**public** **void** setRunning(**boolean** isRunning) {

**this**.isRunning = isRunning;

}

@Override

**public** **void** run() {

System.***out***.println("进入run方法...");

**while**(isRunning==**true**) {

//....

}

System.***out***.println("线程停止.....");

}

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

RunThread rt =**new** RunThread();

rt.start();

Thread.*sleep*(1000);

//更改状态

rt.setRunning(**false**);

System.***out***.println("isRunning的值已经被设置为false");

Thread.*sleep*(1000);

System.***out***.println(rt.isRunning);

}

}

总结： java中每一个线程都有一个工作内存区，其中存放着所有线程共享的主内存中的变量值的拷贝，当线程执行时，他是在自己的工作内存中操作这些变量，为了存取一个共享的变量，一个线程通常先获取锁定并去清除他的内存工作区，把这些共享的变量从所有线程的共享内存区中正确的装入到自己的工作内存区中。当线程解锁时，保证该工作区中变量的值写回到共享内存区。

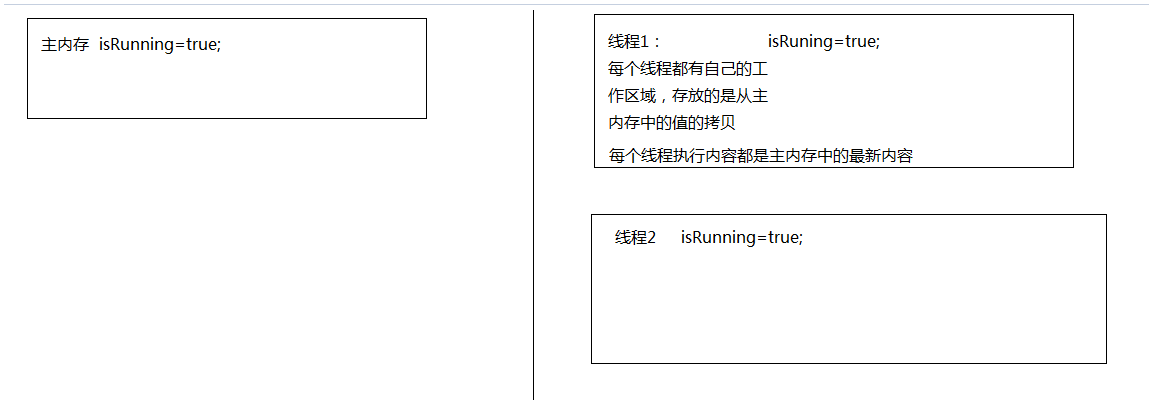
**Volatile的作用：就是强制线程到主内存(共享内存)去读取变量，而不是在线程工区的内存区域中读取，从而实现了多个线程间的变量的可见性，也就是满足线程安全的可见性。**

多线程中一个线程修改了变量 另外一个线程能否立马感知

一个线程能够执行的操作 使用(use) 赋值(assign) 装载(load) 存储(Store) 锁定(lock)

解锁(unlock)

主内存的操作 读(read) 写(write) 锁定(lock) 解锁(unlock) 每个操作都是原子操作



### Volatile关键字非原子性

Volatile关键字虽然拥有多个线程之前的可见性，但是却不具备同步性(也就是原子性)，可以说是一个轻量级锁(Synchronized)的应用，但是性能比Synchronized强大很多，不会造成阻塞，很多开源框中大量使用Volatile关键字比如netty ，但是要注意的是volatile仅仅用于多个线程之间变量的操作，并不能替代Synchronized同步功能。

**package** com.guigu.base.sysc004;

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

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.sysc004.VolatileAtomic</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日下午1:51:22 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** VolatileAtomic **extends** Thread{

// private static volatile int count;

**private** **static** AtomicInteger *count* =**new** AtomicInteger(0);

**public** **static** **void** addCount() {

**for** (**int** i = 0; i < 1000; i++) {

// count++;

*count*.incrementAndGet();

}

System.***out***.println(*count*);

}

@Override

**public** **void** run() {

*addCount*();

}

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

VolatileAtomic [] arr =**new** VolatileAtomic[100];

**for** (**int** i = 0; i < 100; i++) {

arr[i]=**new** VolatileAtomic();

}

**for**(**int** i=0;i<100;i++) {

arr[i].start();

}

}

}

**package** com.guigu.base.sysc004;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.sysc004.AtomicUse</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日下午2:00:05 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**import** java.util.ArrayList;

**import** java.util.List;

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

**public** **class** AtomicUse {

// 形参是指定初始值

**private** **static** AtomicInteger *count* = **new** AtomicInteger(0);

// addAndGet本身是一个原子性操作 但是多个addAndGet在一个方法内是非原子性的操作 所以需要对这个整体的方法再次加锁

**public** **synchronized** **int** multiAdd() {

**try** {

// Thread.sleep(1000);

*count*.addAndGet(1);

*count*.addAndGet(2);

*count*.addAndGet(3);

*count*.addAndGet(4);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** *count*.get();

}

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

**final** AtomicUse au = **new** AtomicUse();

List<Thread> ts = **new** ArrayList<>();

**for** (**int** i = 0; i < 100; i++) {

ts.add(**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

System.***out***.println(au.multiAdd());

}

}));

}

**for** (Thread t : ts) {

t.start();

}

}

}

# 同步容器

## 同步类容器

同步类容器都是线程安全的。但是在某些场景下可能需要**再次加锁保护复合操作**。复合类操作如：迭代(反复访问元素，遍历完集合中所有的元素)，跳转（根据指定的顺序找到当前元素的下一个元素），以及条件运算，这些复合操作在多线程并发修改容器时，都可能会表现出意外的行为。 最常见的经典的错误**ConcurrentModificationException** 原因是当容器迭代的过程中 被并发的修改了内容。

同步容器：比如古老的Vector Hashtable等这些容器的同步其实都是jdk加了Collections.sychrinized等工厂方法创建实现。 也就是说底层无非就是synchronized关键字。

互联网时代完全不能满足 高并发的需求、在保证线程安全的同时也必须用于足够好的性能。

**package** com.guigu.base.coll005;

**import** java.util.Vector;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.coll005.Tickets</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日下午2:54:49 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** Tickets {

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

// 初始化 火车票池 并且添加火车票

**final** Vector<String> tickets = **new** Vector<>();

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

tickets.add("火车票:" + i);

}

**for** (**int** i = 1; i <= 10; i++) {

**new** Thread("线程" + i) {

**public** **void** run() {

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

**if** (tickets.isEmpty()) {

**break**;

}

//ConcurrentModificationException

System.***out***.println(Thread.*currentThread*().getName() + "---" + tickets.remove(0));

}

};

}.start();

}

}

}

## 并发容器

Jdk1.5之后提供了多种并发容器类替代同步容器 从而改善性能，同步容器类的状态都是串行化的，他们虽然都是线程安全的，但是严重减低了并发性，在多线程环境下，严重减低了应用程序的吞吐量。

并发类容器就是针对并发设计的

使用**CurrentHashMap**来替代传统的Hashtable --->HashMap

使用**CopyOnWriteArrayList**替代Vctor ----> ArrayList

使用**CopeOnWriteArraySet**替代Set ----->HashSet

还有并发的队列

Queue

使用 **ConcurrentLinkedQueue** 替代Queue

**ArrayBlockingQueue**

**PriorityBlockingQueue**

**SynchronousQueue**

### ConcurrentMap

**源码解读 必须自我解读。**

ConcurrentMap接口有两个重要的实现

**ConcurrentHashMap**

**ConcurrentSkipListMap(**支持并发排序功能 弥补CurrentHashMap的不足)

ConcurrentHashMap 内部使用段来表示这些不同的部分，每个段其实就是一个小的Hashtable,他们都有自己的锁，只要多个修改操作发生在不同的段上，他们就可以并发的执行，把一个整体分为16个段，也就是最高支持16个线程并发的修改的操作，这也是多线程在场景下**减少锁的粒度**

从而降低锁竞争，并且代码中有大量共享变量使用volatile关键字声明，目的是第一时间获取修改的内容，所以性能非常好。

**package** com.guigu.base.coll006;

**import** java.util.concurrent.ConcurrentHashMap;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.coll006.UseConcurrentHashMap</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日下午3:12:22 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** UseConcurrentHashMap {

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

ConcurrentHashMap<String, String> chm =**new** ConcurrentHashMap<>();

//使用方式和传统的map用法基本相同

chm.put("username", "guigu");

chm.put("age", "15");

chm.put("gender", "female");

System.***out***.println(chm.get("username"));

System.***out***.println(chm.size());

chm.forEach((key,value)->System.***out***.println(key+"---"+value));

}

}

### CopyOnWrite

Copy-On-Write 简称COW 是一种程序中的优化策略。

JDK中的COW容器有两种 CopyOnWriteArrayList 和CopyOnWriteArraySet

什么是CopyOnWrite容器：

CopyOnWrite容器是即写时复制的容器，通俗说，当我们往一个容器中添加元素的时候 ，不是直接向当前容器中添加，而是先将当前容器进行Copy，复制出一个新的容器，然后向新的容器中添加元素，添加完成元素后，在将原来容器的引用指向新的容器，这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何数据，所有CopyOnWrite容器也是一种读写分离的思想，读和写是不同的容器。

**package** com.guigu.base.coll006;

**import** java.util.concurrent.CopyOnWriteArrayList;

**import** java.util.concurrent.CopyOnWriteArraySet;

/\*\*

\* <p>project\_name:ThreadMulti</p>

\* <p>package\_name:com.guigu.base.coll006.UseCopyOnWrite</p>

\* <p>description：</p>

\* <p>@author：刘老师<p>

\* <p> date:2018年3月15日下午3:28:30 </p>

\* <p>comments： </p>

\* <p>@version jdk1.8</p>

\*

\* <p>Copyright (c) 2018, 980991634@qq.com All Rights Reserved. </p>

\*/

**public** **class** UseCopyOnWrite {

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

//用法和ArrayList基本一致

CopyOnWriteArrayList<String> cwal =**new** CopyOnWriteArrayList<>();

//用法基本和Set基本一致

CopyOnWriteArraySet<String> cwas =**new** CopyOnWriteArraySet<>();

}

}

### 并发的Queue