参考：<http://blog.csdn.net/u011116672/article/details/51065711>

# [Java并发编程系列之十八：读写锁](http://blog.csdn.net/u011116672/article/details/51065711)

版权声明：本文为博主原创文章，转载请注明出处。

在之前提到的synchronized的互斥锁和ReentrantLock都属于排他锁，这些锁在同一时刻只能允许一个线程进行访问。而读写锁允许同一时刻有多个读线程进行访问，但是在有写线程的时候，所有的读线程和其他所有的写线程都将阻塞。读写锁维护了一对锁，一个读锁和一个写锁，这种分离提高了并发性，因为在使用排他锁的时候，读线程也是被阻塞的，相比之下确实提高了并行度。

读写锁除了保证写操作对读操作的可见性和读操作的并行度的提升外，还能够简化读写交互的编程场景。试想在一个读多写少的场景下，如果没有读写锁，正常的交互机制就是等待/通知(wait/notify或者await/signal)机制了，就是在写操作开始时，所有晚于写操作的读操作都会陷入阻塞状态，只有当写操作完成操作并进行通知之后，读操作才能得到执行的机会。使用读写锁则可以大大简化编程难度，在读操作时获取读锁，因为读锁可以被多个读线程获取，所以提高了并行度，写操作时获取写锁即可。读写锁适用于读大于写的场景，Java中读写锁的重要实现是ReentrantReadWriteLock。ReentrantReadWriteLock的上层接口是ReadWriteLock。ReadWriteLock接口只有两个方法：readLock()和writeLock()。

下面通过一个例子演示ReentrantReadWriteLock的使用：代码实现了一个缓存示例

package com.rhwayfun.concurrency.r0405;

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantReadWriteLock;

/\*\*

\* Created by rhwayfun on 16-4-5.

\*/

public class Cache {

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

static ReentrantReadWriteLock rw = new ReentrantReadWriteLock();

static Lock readLock = rw.readLock();

static Lock writeLock = rw.writeLock();

static DateFormat format = new SimpleDateFormat("HH:mm:ss");

/\*\*

\* 获取一个key对应的value

\* @param key

\* @return

\*/

public static final Object get(String key){

readLock.lock();

try {

return map.get(key);

}finally {

readLock.unlock();

}

}

/\*\*

\* 设置key对应的value，并返回旧的value

\* @param key

\* @param value

\*/

public static final Object put(String key,Object value){

writeLock.lock();

try {

return map.put(key,value);

}finally {

writeLock.unlock();

}

}

/\*\*

\* 清空所有的内容

\*/

public static final void clear(){

writeLock.lock();

try {

map.clear();

}finally {

writeLock.unlock();

}

}

/\*\*

\* 写线程

\*/

static class Writer implements Runnable{

private Cache cache;

public Writer(Cache cache) {

this.cache = cache;

}

public void run() {

long start = System.currentTimeMillis();

int i = 0;

for (;;i++){

if (System.currentTimeMillis() - start > 1000 \* 5){

break;

}

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException e) {

e.printStackTrace();

}

String value = format.format(new Date());

cache.put(String.valueOf(i),value);

System.out.println(Thread.currentThread().getName() + ":is writing data "

+ String.valueOf(i) + "=" + value + " at "

+ format.format(new Date()));

}

System.out.println(Thread.currentThread().getName() + ":finish writing data at "

+ format.format(new Date()));

}

}

/\*\*

\* 读线程

\*/

static class Reader implements Runnable{

private Cache cache;

public Reader(Cache cache) {

this.cache = cache;

}

public void run() {

int i = 0;

for (;;i++){

Object obj = cache.get(String.valueOf(i));

if (obj instanceof String){

System.out.println(Thread.currentThread().getName() + ":is reading data "

+ String.valueOf(i) + "=" + obj +" at "

+ format.format(new Date()));

}

}

}

}

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

Cache cache = new Cache();

new Thread(new Writer(cache),"Writer").start();

TimeUnit.SECONDS.sleep(5);

for (int i = 0; i < 5; i++){

new Thread(new Reader(cache),"Reader" + i).start();

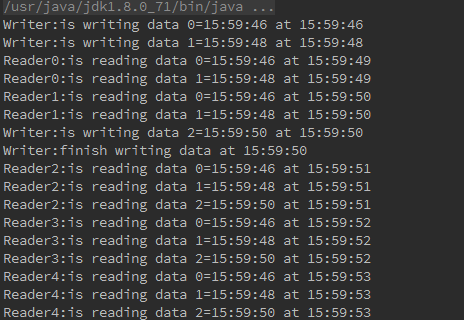
TimeUnit.SECONDS.sleep(1);

}

}

}

程序运行的结果如下：



上述示例中Cache组合使用一个非线程安全的HashMap作为缓存的实现，同时使用读锁和写锁来保证Cache是线程安全的。在读操作get方法中使用读锁，使得并发访问该方法的时候不会阻塞。在写操作put方法中使用写锁，在更新HashMap的时候必须提前获取写锁，获取写锁的线程将会阻塞读锁和写锁的获取，而只有在写锁被释放后其他读写操作才能继续执行。