

带你一次搞定 Java 多线程(IV)

3 线程同步

3.5 CAS

CAS(Compare And Swap)是由硬件实现的.

CAS 可以将 read- modify - write 这类的操作转换为原子操作.

i++自增操作包括三个子操作:

从主内存读取 i 变量值

对 i 的值加 1

再把加 1 之后 的值保存到主内存

CAS 原理: 在把数据更新到主内存时,再次读取主内存变量的值,如果现在变量的值与期望的值(操作起始时读取的值)一样就更新.

CAS
在把数据更新到主内存的
共享变量前,
再次读取主内存共享变量的值
如果现在读取的共享变量值与期
望的值一样就更新

主内存

count = 10 11

线程1 count++

- 1) 从主内存中读取 count 变量的值到线程的工作内存中, count 10
- 2) 对线程内存中的 count 的值加 1 11
- 3) 把线程工作内存中的 count 值更新到主内存中

再次读取主内存的共享变量 count 的值

11

如果读取的主内存共享变量的值与期望的值不一样, 撤销本次操作

线程2 count++

- 1) 从主内存中读取 count 变量的值到线程的工作内存中, count 10
- 2) 对线程内存中的 count 的值加 1 11
- 3) 把线程工作内存中的 count 值更新到主内存中

再次读取主内存的共享变量 count 的值

10

使用 CAS 实现线程安全的计数器

```
package com.wkcto.cas;

/**
 * 使用 CAS 实现一个线程安全的计数器
 *
 * Author: 老崔
 */
public class CASTest {
    public static void main(String[] args) {
        CASCounter casCounter = new CASCounter();

        for (int i = 0; i < 100000; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    System.out.println(casCounter.incrementAndGet());
                }
            }).start();
        }
    }
}

class CASCounter{

    //使用 volatile 修饰 value 值,使线程可见
    volatile private long value;

    public long getValue() {
        return value;
    }

    //定义 compare and swap 方法
    private boolean compareAndSwap(long expectedValue, long newValue){

        //如果当前 value 的值与期望的 expectedValue 值一样,就把当前的 Value 字段替换为
        newValue 值

        synchronized (this){
```

```
        if ( value == expectedValue){
            value = newValue;
            return true;
        }else {
            return false;
        }
    }
}

//定义自增的方法
public long incrementAndGet(){
    long oldValue ;
    long newValue;
    do {
        oldValue = value;
        newValue = oldValue+1;
    }while ( !compareAndSwap(oldvalue, newValue) );
    return newValue;
}
```

CAS 实现原子操作背后有一个假设：共享变量的当前值与当前线程提供的期望值相同，就认为这个变量没有被其他线程修改过。

实际上这种假设不一定总是成立.如有共享变量 `count = 0`

A 线程对 `count` 值修改为 10

B 线程对 `count` 值修改为 20

C 线程对 `count` 值修改为 0

当前线程看到 `count` 变量的值现在是 0,现在是否认为 `count` 变量的值没有被其他线程更新呢？这种结果是否能够接受??

这就是 CAS 中的 ABA 问题,即共享变量经历了 A->B->A 的更新。

是否能够接收 ABA 问题跟实现的算法有关.

如果想要规避 ABA 问题,可以为共享变量引入一个修订号(时间戳), 每次修改共享变量时,相应的修订号就会增加 1. ABA 变量更新过程变量: $[A,0] \rightarrow [B,1] \rightarrow [A,2]$, 每次对共享变量的修改都会导致修订号的增加,通过修订号依然可以准确判断变量是否被其他线程修改过. AtomicStampedReference 类就是基于这种思想产生的.

3.6 原子变量类

原子变量类基于 CAS 实现的, 当对共享变量进行 read-modify-write 更新操作时,通过原子变量类可以保障操作的原子性与可见性.对变量的 read-modify-write 更新操作是指当前操作不是一个简单的赋值,而是变量的新值依赖变量的旧值,如自增操作 $i++$. 由于 volatile 只能保证可见性,无法保障原子性, 原子变量类内部就是借助一个 Volatile 变量,并且保障了该变量的 read-modify-write 操作的原子性, 有时把原子变量类看作增强的 volatile 变量. 原子变量类有 12 个,如:

分组	原子变量类
基础数据型	AtomicInteger, AtomicLong, AtomicBoolean
数组型	AtomicIntegerArray, AtomicLongArray,AtomicReferenceArray
字段更新器	AtomicIntegerFieldUpdater,

	AtomicLongFieldUpdater, AtomicReferenceFieldUpdater
引用型	AtomicReference, AtomicStampedReference, AtomicMarkableReference

3.6.1 AtomicLong

```
package com.wkcto.atomics.atomiclong;

import java.util.concurrent.atomic.AtomicLong;

/**
 * 使用原子变量类定义一个计数器
 *
 * 该计数器,在整个程序中都能使用,并且所有的地方都使用这一个计数器,这个计数器可以
 * 设计为单例
 *
 * 北京动力节点老崔
 */
public class Indicator {

    //构造方法私有化
    private Indicator(){}

    //定义一个私有的本类静态的对象
    private static final Indicator INSTANCE = new Indicator();

    //3)提供一个公共静态方法返回该类唯一实例
    public static Indicator getInstance(){
        return INSTANCE;
    }
}
```

```
//使用原子变量类保存请求总数,成功数,失败数

private final AtomicLong requestCount = new AtomicLong(0); //记录请求总数

private final AtomicLong successCount = new AtomicLong(0); //处理成功总数

private final AtomicLong fialureCount = new AtomicLong(0); //处理失败总数


//有新的请求

public void newRequestReceive(){
    requestCount.incrementAndGet();
}

//处理成功

public void requestProcessSuccess(){
    successCount.incrementAndGet();
}

//处理失败

public void requestProcessFailure(){
    fialureCount.incrementAndGet();
}


//查看总数,成功数,失败数

public long getRequestCount(){
    return requestCount.get();
}

public long getSuccessCount(){
    return successCount.get();
}

public long getFailureCount(){
    return fialureCount.get();
}

}

package com.wkcto.atomics.atomiclong;

import java.util.Random;
```

```
/**
 * 模拟服务器的请求总数, 处理成功数,处理失败数
 *
 * 北京动力节点老崔
 */
public class Test {
    public static void main(String[] args) {

        //通过线程模拟请求,在实际应用中可以在 ServletFilter 中调用 Indicator 计数器的相
        关方法

        for (int i = 0; i < 10000; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {

                    //每个线程就是一个请求,请求总数要加 1

                    Indicator.getInstance().newRequestReceive();
                    int num = new Random().nextInt();

                    if ( num % 2 == 0 ){           //偶数模拟成功

                        Indicator.getInstance().requestProcessSuccess();

                    }else {           //处理失败

                        Indicator.getInstance().requestProcessFailure();

                    }

                }
            }).start();

        }

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //打印结果

        System.out.println( Indicator.getInstance().getRequestCount());           //总的请求数
```

```
        System.out.println( Indicator.getInstance().getSuccessCount());    //成功数

        System.out.println( Indicator.getInstance().getFailureCount());    //失败数
    }
}
```

3.6.2 AtomicIntegerArray

原子更新数组

```
package com.wkcto.atomics.atomicarray;

import java.util.concurrent.atomic.AtomicIntegerArray;

/**
 * AtomicIntegerArray 的基本操作
 *
 * 原子更新数组
 *
 * 北京动力节点老崔
 */
public class Test {
    public static void main(String[] args) {

        //1)创建一个指定长度的原子数组

        AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(10);
        System.out.println( atomicIntegerArray );    //[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

        //2)返回指定位置的元素

        System.out.println( atomicIntegerArray.get(0));    //0
        System.out.println( atomicIntegerArray.get(1));    //0

        //3)设置指定位置的元素

        atomicIntegerArray.set(0, 10);

        //在设置数组元素的新值时，同时返回数组元素的旧值
    }
}
```



```
System.out.println( atomicIntegerArray.getAndSet(1, 11) ); //0
System.out.println( atomicIntegerArray );    //[10, 11, 0, 0, 0, 0, 0, 0, 0]

//4)修改数组元素的值,把数组元素加上某个值

System.out.println( atomicIntegerArray.addAndGet(0, 22) ); //32
System.out.println( atomicIntegerArray.getAndAdd(1, 33));    //11
System.out.println( atomicIntegerArray );    //[32, 44, 0, 0, 0, 0, 0, 0, 0]

//5)CAS 操作

//如果数组中索引值为 0 的元素的值是 32 , 就修改为 222

System.out.println( atomicIntegerArray.compareAndSet(0, 32, 222)); //true
System.out.println( atomicIntegerArray );    //[222, 44, 0, 0, 0, 0, 0, 0, 0]
System.out.println( atomicIntegerArray.compareAndSet(1, 11, 333)); //false
System.out.println(atomicIntegerArray);

//6)自增/自减

System.out.println( atomicIntegerArray.incrementAndGet(0) );    //223, 相当于前缀

System.out.println( atomicIntegerArray.getAndIncrement(1));    //44, 相当于后缀

System.out.println( atomicIntegerArray );    //[223, 45, 0, 0, 0, 0, 0, 0, 0]
System.out.println( atomicIntegerArray.decrementAndGet(2));    //-1
System.out.println( atomicIntegerArray);    //[223, 45, -1, 0, 0, 0, 0, 0, 0]
System.out.println( atomicIntegerArray.getAndDecrement(3));    //0
System.out.println( atomicIntegerArray );    //[223, 45, -1, -1, 0, 0, 0, 0, 0]
    }
}
```

```
package com.wkcto.atomics.atomicarray;

import java.util.concurrent.atomic.AtomicIntegerArray;

/**
 * 在多线程中使用 AtomicIntegerArray 原子数组
 *
 * 北京动力节点老崔
 */
public class Test02 {

    //定义原子数组
```

```
static AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(10);

public static void main(String[] args) {

    //定义线程数组
    Thread[] threads = new Thread[10];

    //给线程数组元素赋值
    for (int i = 0; i < threads.length; i++) {
        threads[i] = new AddThread();
    }

    //开启子线程
    for (Thread thread : threads) {
        thread.start();
    }

    //在主线程中查看自增完以后原子数组中的各个元素的值,在主线程中需要在所有
    子线程都执行完后再查看

    //把所有的子线程合并到当前主线程中
    for (Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println( atomicIntegerArray );
}

//定义一个线程类,在线程类中修改原子数组
static class AddThread extends Thread{
    @Override
    public void run() {

        //把原子数组的每个元素自增 1000 次
        for (int j = 0; j < 100000; j++) {
            for (int i = 0; i < atomicIntegerArray.length(); i++) {
                atomicIntegerArray.getAndIncrement(i % atomicIntegerArray.length());
            }
        }
    }
}
```

```
    }  
    }  
    /* for (int i = 0; i < 10000; i++) {  
        atomicIntegerArray.getAndIncrement(i % atomicIntegerArray.length());  
    }*/  
}  
}
```

3.6.2 AtomicIntegerFieldUpdater

AtomicIntegerFieldUpdater 可以对原子整数字段进行更新,要求:

- 1) 字符必须使用 volatile 修饰,使线程之间可见
- 2) 只能是实例变量,不能是静态变量,也不能使用 final 修饰

```
package com.wkcto.atomics.atominintegerfiled;  
  
/**  
 * 使用 AtomicIntegerFieldUpdater 更新的字段必须使用 volatile 修饰  
 * 北京动力节点老崔  
 */  
public class User {  
    int id;  
    volatile int age;  
  
    public User(int id, int age) {  
        this.id = id;  
        this.age = age;  
    }  
  
    @Override  
    public String toString() {  
        return "User{" +  
            "id=" + id +
```

```
        ", age=" + age +  
        '}';  
    }  
}  
  
package com.wkcto.atomics.atominintegerfiled;  
  
import java.util.concurrent.atomic.AtomicIntegerFieldUpdater;  
  
/**  
 * 线程类,  
  
 * 北京动力节点老崔  
 */  
public class SubThread extends Thread {  
    private User user;          //要更新的 User 对象  
    //创建 AtomicIntegerFieldUpdater 更新器  
    private AtomicIntegerFieldUpdater<User> updater =  
        AtomicIntegerFieldUpdater.newUpdater(User.class, "age");  
  
    public SubThread(User user) {  
        this.user = user;  
    }  
  
    @Override  
    public void run() {  
        //在子线程中对 user 对象的 age 字段自增 10 次  
        for (int i = 0; i < 10; i++) {  
            System.out.println( updater.getAndIncrement(user));  
        }  
    }  
}  
  
package com.wkcto.atomics.atominintegerfiled;  
  
/**  
 * 北京动力节点老崔
```

```
*/  
public class Test {  
    public static void main(String[] args) {  
        User user = new User(1234, 10);  
        //开启 10 个线程  
        for (int i = 0; i < 10; i++) {  
            new SubThread(user).start();  
        }  
  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println( user );  
    }  
}
```

3.6.3 AtomicReference

可以原子读写一个对象

```
package com.wkcto.atomics.atomicreference;  
  
import java.util.Random;  
import java.util.concurrent.atomic.AtomicReference;  
  
/**  
 * 使用 AtomicReference 原子读写一个对象  
 * 北京动力节点老崔  
 */  
public class Test01 {  
    //创建一个 AtomicReference 对象  
    static AtomicReference<String> atomicReference = new AtomicReference<>("abc");
```

```
public static void main(String[] args) throws InterruptedException {  
    //创建 100 个线程修改字符串  
    for (int i = 0; i < 100; i++) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                try {  
                    Thread.sleep(new Random().nextInt(20));  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                if (atomicReference.compareAndSet("abc","def")){  
                    System.out.println(Thread.currentThread().getName() + "把字符串  
abc 更改为 def");  
                }  
            }  
        }).start();  
    }  
    //再创建 100 个线程  
    for (int i = 0; i < 100; i++) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                try {  
                    Thread.sleep(new Random().nextInt(20));  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                if (atomicReference.compareAndSet("def","abc")){  
                    System.out.println(Thread.currentThread().getName() + "把字符串  
还原为 abc");  
                }  
            }  
        }).start();  
    }  
}
```

```
        Thread.sleep(1000);
        System.out.println(atomicReference.get());
    }
}

package com.wkcto.atomics.atomicreference;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

/**
 * 演示 AtomicReference 可能会出现 CAS 的 ABA 问题
 *
 * 北京动力节点老崔
 */
public class Test02 {
    private static AtomicReference<String> atomicReference = new AtomicReference<>("abc");
    public static void main(String[] args) throws InterruptedException {

        //创建第一个线程,先把 abc 字符串改为"def",再把字符串还原为 abc

        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                atomicReference.compareAndSet("abc", "def");
                System.out.println(Thread.currentThread().getName() + "--" +
atomicReference.get());
                atomicReference.compareAndSet("def", "abc");
            }
        });

        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println( atomicReference.compareAndSet("abc", "ghg"));
            }
        });
    }
}
```

```
});  
t1.start();  
t2.start();  
t1.join();  
t2.join();  
System.out.println( atomicReference.get());  
}  
}
```

```
package com.wkcto.atomics.atomicreference;
```

```
import java.util.concurrent.TimeUnit;
```

```
import java.util.concurrent.atomic.AtomicReference;
```

```
import java.util.concurrent.atomic.AtomicStampedReference;
```

```
/**
```

```
 * AtomicStampedReference 原子类可以解决 CAS 中的 ABA 问题
```

```
 * 在 AtomicStampedReference 原子类中有一个整数标记值 stamp, 每次执行 CAS 操作时,需  
要对比它的版本,即比较 stamp 的值
```

```
 * 北京动力节点老崔
```

```
 */
```

```
public class Test03 {
```

```
 // private static AtomicReference<String> atomicReference = new AtomicReference<>("abc");
```

```
 //定义 AtomicStampedReference 引用操作"abc"字符串,指定初始化版本号为 0
```

```
 private static AtomicStampedReference<String> stampedReference = new  
AtomicStampedReference<>("abc", 0);
```

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

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

```
         @Override
```

```
         public void run() {
```

```
             stampedReference.compareAndSet("abc", "def",
```

```
 stampedReference.getStamp(), stampedReference.getStamp()+1);
```

```
             System.out.println(Thread.currentThread().getName() + "--"
```

```
 +stampedReference.getReference());
```

```
             stampedReference.compareAndSet("def", "abc",
```

```
 stampedReference.getStamp(), stampedReference.getStamp()+1);
```

```
         }
```



```
});  
Thread t2 = new Thread(new Runnable() {  
    @Override  
    public void run() {  
  
        int stamp = stampedReference.getStamp();           //获得版本号  
  
        try {  
            TimeUnit.SECONDS.sleep(1);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println( stampedReference.compareAndSet("abc", "ggg", stamp,  
stamp+1));  
    }  
});  
  
t1.start();  
t2.start();  
t1.join();  
t2.join();  
  
System.out.println( stampedReference.getReference() );  
}  
}
```

4 线程间的通信

4.1 等待/通知机制

4.1.1 什么是等待通知机制

在单线程编程中,要执行的操作需要满足一定的条件才能执行,可以把这个操作放在 if 语句块中.

在多线程编程中,可能 A 线程的条件没有满足只是暂时的,稍后其他的线程 B 可能会更新条件使得 A 线程的条件得到满足. 可以将 A 线程暂停,直到它的条件得到满足后再将 A 线程唤醒.它的伪代码:

```
atomics{           //原子操作

    while( 条件不成立 ){

        等待

    }

}
```

当前线程被唤醒条件满足后,继续执行下面的操作

```
}
```

4.1.2 等待/通知机制的实现

Object 类中的 wait()方法可以使执行当前代码的线程等待,暂停执行,直到接到通知或被中断为止.

注意:

- 1) wait()方法只能 在同步代码块中由锁对象调用
- 2) 调用 wait()方法,当前线程会释放锁

其伪代码如下:

```
//在调用 wait()方法前获得对象的内部锁
```

```
synchronized( 锁对象 ){
```

```
    while( 条件不成立 ){
```

```
        //通过锁对象调用 wait()方法暂停线程,会释放锁对象
```

```
        锁对象.wait();  
    }  
    //线程的条件满足了继续向下执行  
}
```

Object 类的 notify()可以唤醒线程,该方法也必须在同步代码块中由锁对象调用. 没有使用锁对象调用 wait()/notify() 会抛出 IllegalMonitorStateException 异常. 如果有多个等待的线程,notify()方法只能唤醒其中的一个. 在同步代码块中调用 notify()方法后,并不会立即释放锁对象,需要等当前同步代码块执行完后才会释放锁对象,一般将 notify()方法放在同步代码块的最后. 它的伪代码如下:

```
synchronized( 锁对象 ){  
    //执行修改保护条件 的代码  
    //唤醒其他线程  
    锁对象.notify();  
}
```

```
package com.wkcto.wait;  
  
/**  
 * 需要通过 notify()唤醒等待的线程  
 * 北京动力节点老崔  
 */  
public class Test03 {  
    public static void main(String[] args) throws InterruptedException {
```

```
String lock = "wkcto";    //定义一个字符串作为锁对象

Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {
        synchronized (lock) {

            System.out.println("线程 1 开始等待: " + System.currentTimeMillis());

            try {

                lock.wait();    //线程等待,会释放锁对象,当前线程转入
                                blocked 阻塞状态

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

            System.out.println("线程 1 结束等待:" + System.currentTimeMillis());

        }
    }
});

//定义第二个线程,在第二个线程中唤醒第一个线程

Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {

        //notify()方法也需要在同步代码块中,由锁对象调用

        synchronized (lock){

            System.out.println("线程 2 开始唤醒 : " + System.currentTimeMillis());

            lock.notify();    //唤醒在 lock 锁对象上等待的某一个线程

            System.out.println("线程 2 结束唤醒 : " + System.currentTimeMillis());

        }
    }
});

t1.start();    //开启 t1 线程,t1 线程等待
```

```
Thread.sleep(3000);    //main 线程睡眠 3 秒,确保 t1 入睡

t2.start();           //t1 线程开启 3 秒后,再开启 t2 线程唤醒 t1 线程
}
}
```

4.1.3 notify()方法后不会立即释放锁对象