在图 1-2 中，线程 A 已经持有了资源 2，它同时还想申请资源 1，线程 B 已经持有了资源 1，它同时还想申请资源 2，所以线程 1 和线程 2 就因为相互等待对方已经持有的资源，而进入了死锁状态。

那么为什么会产生死锁呢？学过操作系统的朋友应该都知道，死锁的产生必须具备以下四个条件。

- 互斥条件：指线程对已经获取到的资源进行排它性使用，即该资源同时只由一个线程占用。如果此时还有其他线程请求获取该资源，则请求者只能等待，直至占有资源的线程释放该资源。
- 请求并持有条件：指一个线程已经持有了至少一个资源，但又提出了新的资源请求，而新资源已被其他线程占有，所以当前线程会被阻塞，但阻塞的同时并不释放自己已经获取的资源。
- 不可剥夺条件：指线程获取到的资源在自己使用完之前不能被其他线程抢占，只有在自己使用完毕后才由自己释放该资源。
- 环路等待条件：指在发生死锁时，必然存在一个线程—资源的环形链，即线程集合{T0，T1，T2，…，T*n*} 中的 T0 正在等待一个 T1 占用的资源，T1 正在等待 T2 占用的资源，……T*n* 正在等待已被 T0 占用的资源。

下面通过一个例子来说明线程死锁。

```
public class DeadLockTest2 {

    // 创建资源
    private static Object resourceA = new Object();
    private static Object resourceB = new Object();

    public static void main(String[] args) {

        // 创建线程A
        Thread threadA = new Thread(new Runnable() {
            public void run() {
                synchronized (resourceA) {
                    System.out.println(Thread.currentThread() + " get ResourceA");

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

```
                    e.printStackTrace();
                }

                System.out.println(Thread.currentThread() + "waiting get sourceB");
                synchronized (resourceB) {
                    System.out.println(Thread.currentThread() + "get esourceB");
                }
            }
        }
    });

    // 创建线程B
    Thread threadB = new Thread(new Runnable() {
        public void run() {
            synchronized (resourceB) {
                System.out.println(Thread.currentThread() + " get ResourceB");

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

                System.out.println(Thread.currentThread() + "waiting get esourceA");
                synchronized (resourceA) {
                    System.out.println(Thread.currentThread() + "get ResourceA");
                }
            };
        }
    });

    // 启动线程
    threadA.start();
    threadB.start();
  }
}
```

输出结果如下。

```
DeadLockTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[Thread-0,5,main] get ResourceA
Thread[Thread-1,5,main] get ResourceB
Thread[Thread-0,5,main]waiting get ResourceB
Thread[Thread-1,5,main]waiting get ResourceA
```

下面分析代码和结果：Thread-0 是线程 A，Thread-1 是线程 B，代码首先创建了两个资源，并创建了两个线程。从输出结果可以知道，线程调度器先调度了线程 A，也就是把 CPU 资源分配给了线程 A，线程 A 使用 synchronized(resourceA) 方法获取到了 resourceA 的监视器锁，然后调用 sleep 函数休眠 1s，休眠 1s 是为了保证线程 A 在获取 resourceB 对应的锁前让线程 B 抢占到 CPU，获取到资源 resourceB 上的锁。线程 A 调用 sleep 方法后线程 B 会执行 synchronized(resourceB) 方法，这代表线程 B 获取到了 resourceB 对象的监视器锁资源，然后调用 sleep 函数休眠 1s。好了，到了这里线程 A 获取到了 resourceA 资源，线程 B 获取到了 resourceB 资源。线程 A 休眠结束后会企图获取 resourceB 资源，而 resourceB 资源被线程 B 所持有，所以线程 A 会被阻塞而等待。而同时线程 B 休眠结束后会企图获取 resourceA 资源，而 resourceA 资源已经被线程 A 持有，所以线程 A 和线程 B 就陷入了相互等待的状态，也就产生了死锁。下面谈谈本例是如何满足死锁的四个条件的。

首先，resourceA 和 resourceB 都是互斥资源，当线程 A 调用 synchronized(resourceA) 方法获取到 resourceA 上的监视器锁并释放前，线程 B 再调用 synchronized(resourceA) 方法尝试获取该资源会被阻塞，只有线程 A 主动释放该锁，线程 B 才能获得，这满足了资源互斥条件。

线程 A 首先通过 synchronized(resourceA) 方法获取到 resourceA 上的监视器锁资源，然后通过 synchronized(resourceB) 方法等待获取 resourceB 上的监视器锁资源，这就构成了请求并持有条件。

线程 A 在获取 resourceA 上的监视器锁资源后，该资源不会被线程 B 掠夺走，只有线程 A 自己主动释放 resourceA 资源时，它才会放弃对该资源的持有权，这构成了资源的不可剥夺条件。

线程 A 持有 objectA 资源并等待获取 objectB 资源，而线程 B 持有 objectB 资源并等待 objectA 资源，这构成了环路等待条件。所以线程 A 和线程 B 就进入了死锁状态。

1.9.2　如何避免线程死锁

要想避免死锁，只需要破坏掉至少一个构造死锁的必要条件即可，但是学过操作系统的读者应该都知道，目前只有请求并持有和环路等待条件是可以被破坏的。

造成死锁的原因其实和申请资源的顺序有很大关系，使用资源申请的有序性原则就可

以避免死锁，那么什么是资源申请的有序性呢？我们对上面线程 B 的代码进行如下修改。

```
// 创建线程B
    Thread threadB = new Thread(new Runnable() {
        public void run() {
            synchronized (resourceA) {
                System.out.println(Thread.currentThread() + " get ResourceB");

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

                System.out.println(Thread.currentThread() + "waiting get ResourceA");
                synchronized (resourceB) {
                    System.out.println(Thread.currentThread() + "get ResourceA");
                }
            };
        }
    });
```

输出结果如下。

```
<terminated> DeadLockTest2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[Thread-0,5,main] get ResourceA
Thread[Thread-0,5,main]waiting get ResourceB
Thread[Thread-0,5,main]get ResourceB
Thread[Thread-1,5,main] get ResourceB
Thread[Thread-1,5,main]waiting get ResourceA
Thread[Thread-1,5,main]get ResourceA
```

如上代码让在线程 B 中获取资源的顺序和在线程 A 中获取资源的顺序保持一致，其实资源分配有序性就是指，假如线程 A 和线程 B 都需要资源 1，2，3，...，n 时，对资源进行排序，线程 A 和线程 B 只有在获取了资源 n-1 时才能去获取资源 n。

我们可以简单分析一下为何资源的有序分配会避免死锁，比如上面的代码，假如线程 A 和线程 B 同时执行到了 synchronized (resourceA)，只有一个线程可以获取到 resourceA 上的监视器锁，假如线程 A 获取到了，那么线程 B 就会被阻塞而不会再去获取资源 B，线程 A 获取到 resourceA 的监视器锁后会去申请 resourceB 的监视器锁资源，这时候线程 A 是可以获取到的，线程 A 获取到 resourceB 资源并使用后会放弃对资源 resourceB 的持有，然后再释放对 resourceA 的持有，释放 resourceA 后线程 B 才会被从阻塞状态变为激活状态。

所以资源的有序性破坏了资源的请求并持有条件和环路等待条件，因此避免了死锁。

1.10　守护线程与用户线程

Java 中的线程分为两类，分别为 daemon 线程（守护线程）和 user 线程（用户线程）。在 JVM 启动时会调用 main 函数，main 函数所在的线程就是一个用户线程，其实在 JVM 内部同时还启动了好多守护线程，比如垃圾回收线程。那么守护线程和用户线程有什么区别呢？区别之一是当最后一个非守护线程结束时，JVM 会正常退出，而不管当前是否有守护线程，也就是说守护线程是否结束并不影响 JVM 的退出。言外之意，只要有一个用户线程还没结束，正常情况下 JVM 就不会退出。

那么在 Java 中如何创建一个守护线程？代码如下。

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

    Thread daemonThread = new Thread(new  Runnable() {
        public void run() {

        }
    });

    //设置为守护线程
    daemonThread.setDaemon(true);
    daemonThread.start();

}
```

只需要设置线程的 daemon 参数为 true 即可。

下面通过例子来理解用户线程与守护线程的区别。首先看下面的代码。

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

    Thread thread = new Thread(new  Runnable() {
        public void run() {
            for(;;){}
        }
    });

    //启动子线程
    thread.start();
```

```
        System.out.print("main thread is over");
    }
```

输出结果如下。

```
testDaemonThread [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年9月28日 下午10:05:43)
main thread is over
```

如上代码在 main 线程中创建了一个 thread 线程，在 thread 线程里面是一个无限循环。从运行代码的结果看，main 线程已经运行结束了，那么 JVM 进程已经退出了吗？在 IDE 的输出结果右上侧的红色方块说明，JVM 进程并没有退出。另外，在 mac 上执行 jps 会输出如下结果。

```
→ /Users/zhuizhumengxiang git:(master) ✗ jps
36705 TestUserThread
36795 Jps
720
```

这个结果说明了当父线程结束后，子线程还是可以继续存在的，也就是子线程的生命周期并不受父线程的影响。这也说明了在用户线程还存在的情况下 JVM 进程并不会终止。那么我们把上面的 thread 线程设置为守护线程后，再来运行看看会有什么结果：

```
//设置为守护线程
thread.setDaemon(true);
//启动子线程
thread.start();
```

输出结果如下。

```
<terminated> testDaemonThread [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年9月28日 下午10:1:
main thread is over
```

在启动线程前将线程设置为守护线程，执行后的输出结果显示，JVM 进程已经终止了，执行 ps -eaf |grep java 也看不到 JVM 进程了。在这个例子中，main 函数是唯一的用户线程，thread 线程是守护线程，当 main 线程运行结束后，JVM 发现当前已经没有用户线程了，就会终止 JVM 进程。由于这里的守护线程执行的任务是一个死循环，这也说明了如果当前进程中不存在用户线程，但是还存在正在执行任务的守护线程，则 JVM 不等守护线程

运行完毕就会结束 JVM 进程。

main 线程运行结束后，JVM 会自动启动一个叫作 DestroyJavaVM 的线程，该线程会等待所有用户线程结束后终止 JVM 进程。下面通过简单的 JVM 代码来证明这个结论。

翻看 JVM 的代码，能够发现，最终会调用到 JavaMain 这个 C 函数。

```
int JNICALL
JavaMain(void * _args)
{
    ...
    //执行Java中的main函数
    (*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);

    //main函数返回值
    ret = (*env)->ExceptionOccurred(env) == NULL ? 0 : 1;

    //等待所有非守护线程结束，然后销毁JVM进程
    LEAVE();
}
```

LEAVE 是 C 语言里面的一个宏定义，具体定义如下。

```
#define LEAVE() \
    do { \
        if ((*vm)->DetachCurrentThread(vm) != JNI_OK) { \
            JLI_ReportErrorMessage(JVM_ERROR2); \
            ret = 1; \
        } \
        if (JNI_TRUE) { \
            (*vm)->DestroyJavaVM(vm); \
            return ret; \
        } \
    } while (JNI_FALSE)
```

该宏的作用是创建一个名为 DestroyJavaVM 的线程，来等待所有用户线程结束。

在 Tomcat 的 NIO 实现 NioEndpoint 中会开启一组接受线程来接受用户的连接请求，以及一组处理线程负责具体处理用户请求，那么这些线程是用户线程还是守护线程呢？下面我们看一下 NioEndpoint 的 startInternal 方法。

```
  public void startInternal() throws Exception {

        if (!running) {
```

```
        running = true;
        paused = false;

        ...

        //创建处理线程
        pollers = new Poller[getPollerThreadCount()];
        for (int i=0; i<pollers.length; i++) {
            pollers[i] = new Poller();
            Thread pollerThread = new Thread(pollers[i], getName() +
             "-ClientPoller-"+i);
            pollerThread.setPriority(threadPriority);
            pollerThread.setDaemon(true);//声明为守护线程
            pollerThread.start();
        }
        //启动接受线程
        startAcceptorThreads();
    }

protected final void startAcceptorThreads() {
    int count = getAcceptorThreadCount();
    acceptors = new Acceptor[count];

    for (int i = 0; i < count; i++) {
        acceptors[i] = createAcceptor();
        String threadName = getName() + "-Acceptor-" + i;
        acceptors[i].setThreadName(threadName);
        Thread t = new Thread(acceptors[i], threadName);
        t.setPriority(getAcceptorThreadPriority());
        t.setDaemon(getDaemon());//设置是否为守护线程，默认为守护线程
        t.start();
    }
}

private boolean daemon = true;
public void setDaemon(boolean b) { daemon = b; }
public boolean getDaemon() { return daemon; }
```

在如上代码中，在默认情况下，接受线程和处理线程都是守护线程，这意味着当tomcat收到shutdown命令后并且没有其他用户线程存在的情况下tomcat进程会马上消亡，而不会等待处理线程处理完当前的请求。

总结：如果你希望在主线程结束后 JVM 进程马上结束，那么在创建线程时可以将其设置为守护线程，如果你希望在主线程结束后子线程继续工作，等子线程结束后再让 JVM 进程结束，那么就将子线程设置为用户线程。

1.11　ThreadLocal

多线程访问同一个共享变量时特别容易出现并发问题，特别是在多个线程需要对一个共享变量进行写入时。为了保证线程安全，一般使用者在访问共享变量时需要进行适当的同步，如图 1-3 所示。

同步的措施一般是加锁，这就需要使用者对锁有一定的了解，这显然加重了使用者的负担。那么有没有一种方式可以做到，当创建一个变量后，每个线程对其进行访问的时候访问的是自己线程的变量呢？其实 ThreadLocal 就可以做这件事情，虽然 ThreadLocal 并不是为了解决这个问题而出现的。

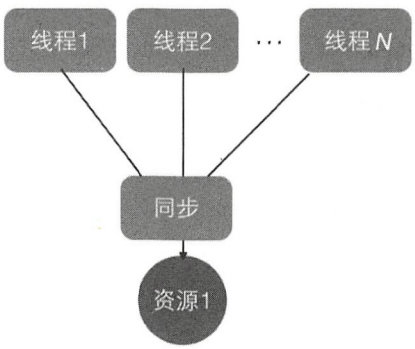


图 1-3

ThreadLocal 是 JDK 包提供的，它提供了线程本地变量，也就是如果你创建了一个 ThreadLocal 变量，那么访问这个变量的每个线程都会有这个变量的一个本地副本。当多个线程操作这个变量时，实际操作的是自己本地内存里面的变量，从而避免了线程安全问题。创建一个 ThreadLocal 变量后，每个线程都会复制一个变量到自己的本地内存，如图 1-4 所示。

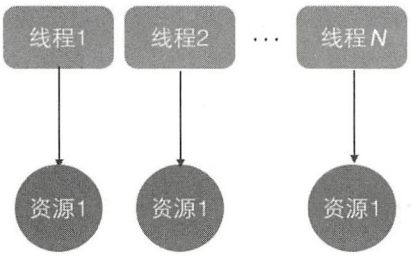


图 1-4

1.11.1 ThreadLocal 使用示例

本节介绍如何使用 ThreadLocal。本例开启了两个线程，在每个线程内部都设置了本地变量的值，然后调用 print 函数打印当前本地变量的值。如果打印后调用了本地变量的 remove 方法，则会删除本地内存中的该变量，代码如下。

```
public class ThreadLocalTest {

    //(1)print函数
    static void print(String str){
        //1.1 打印当前线程本地内存中localVariable变量的值
        System.out.println(str + ":" +localVariable.get());
        //1.2 清除当前线程本地内存中的localVariable变量
        //localVariable.remove();
    }
    //(2) 创建ThreadLocal变量
    static ThreadLocal<String> localVariable = new ThreadLocal<>();
    public static void main(String[] args) {

        //(3) 创建线程one
        Thread threadOne = new Thread(new  Runnable() {
            public void run() {
                //3.1 设置线程One中本地变量localVariable的值
                localVariable.set("threadOne local variable");
                //3.2 调用打印函数
                print("threadOne");
                //3.3 打印本地变量值
                System.out.println("threadOne remove after" + ":" +localVariable.get());

            }
        });
```

```
        //(4) 创建线程two
        Thread threadTwo = new Thread(new  Runnable() {
            public void run() {
                //4.1 设置线程Two中本地变量localVariable的值
                localVariable.set("threadTwo local variable");
                //4.2 调用打印函数
                print("threadTwo");
                //4.3 打印本地变量值
                System.out.println("threadTwo remove after" + ":" +localVariable.get());

            }
        });
        //(5)启动线程
        threadOne.start();
        threadTwo.start();
    }
```

运行结果如下。

```
threadOne:threadOne local variable
threadTwo:threadTwo local variable
threadOne remove after:threadOne local variable
threadTwo remove after:threadTwo local variable
```

代码（2）创建了一个 ThreadLocal 变量。

代码（3）和（4）分别创建了线程 One 和 Two。

代码（5）启动了两个线程。

线程 One 中的代码 3.1 通过 set 方法设置了 localVariable 的值，这其实设置的是线程 One 本地内存中的一个副本，这个副本线程 Two 是访问不了的。然后代码 3.2 调用了 print 函数，代码 1.1 通过 get 函数获取了当前线程（线程 One）本地内存中 localVariable 的值。

线程 Two 的执行类似于线程 One。

打开代码 1.2 的注释后，再次运行，运行结果如下。

```
threadOne:threadOne local variable
threadOne remove after:null
threadTwo:threadTwo local variable
threadTwo remove after:null
```

1.11.2 ThreadLocal 的实现原理

首先看一下 ThreadLocal 相关类的类图结构，如图 1-5 所示。

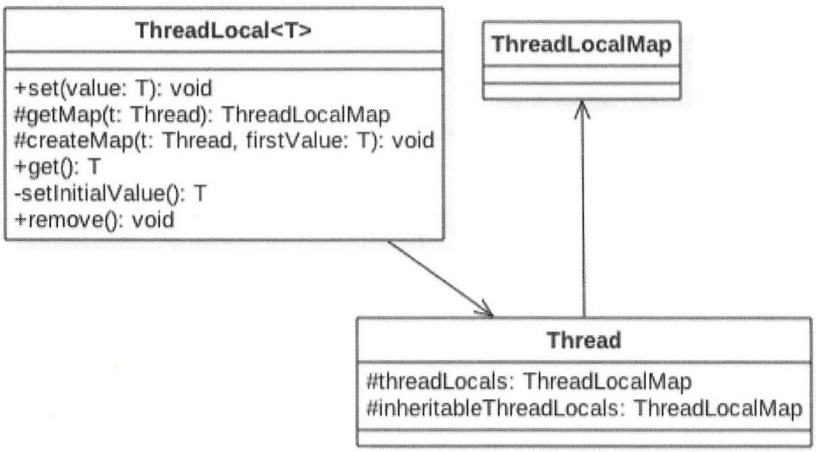


图 1-5

由该图可知，Thread 类中有一个 threadLocals 和一个 inheritableThreadLocals，它们都是 ThreadLocalMap 类型的变量，而 ThreadLocalMap 是一个定制化的 Hashmap。在默认情况下，每个线程中的这两个变量都为 null，只有当前线程第一次调用 ThreadLocal 的 set 或者 get 方法时才会创建它们。其实每个线程的本地变量不是存放在 ThreadLocal 实例里面，而是存放在调用线程的 threadLocals 变量里面。也就是说，ThreadLocal 类型的本地变量存放在具体的线程内存空间中。ThreadLocal 就是一个工具壳，它通过 set 方法把 value 值放入调用线程的 threadLocals 里面并存放起来，当调用线程调用它的 get 方法时，再从当前线程的 threadLocals 变量里面将其拿出来使用。如果调用线程一直不终止，那么这个本地变量会一直存放在调用线程的 threadLocals 变量里面，所以当不需要使用本地变量时可以通过调用 ThreadLocal 变量的 remove 方法，从当前线程的 threadLocals 里面删除该本地变量。另外，Thread 里面的 threadLocals 为何被设计为 map 结构？很明显是因为每个线程可以关联多个 ThreadLocal 变量。

下面简单分析 ThreadLocal 的 set、get 及 remove 方法的实现逻辑。

1．void set(T value)

```
public void set(T value) {
```

```
        //(1)获取当前线程
        Thread t = Thread.currentThread();
        //(2)将当前线程作为key，去查找对应的线程变量，找到则设置
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
        //(3)第一次调用就创建当前线程对应的HashMap
            createMap(t, value);
    }
```

代码（1）首先获取调用线程，然后使用当前线程作为参数调用 getMap(t) 方法，getMap(Thread t) 的代码如下。

```
ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }
```

可以看到，getMap(t) 的作用是获取线程自己的变量 threadLocals，threadlocal 变量被绑定到了线程的成员变量上。

如果 getMap(t) 的返回值不为空，则把 value 值设置到 threadLocals 中，也就是把当前变量值放入当前线程的内存变量 threadLocals 中。threadLocals 是一个 HashMap 结构，其中 key 就是当前 ThreadLocal 的实例对象引用，value 是通过 set 方法传递的值。

如果 getMap(t) 返回空值则说明是第一次调用 set 方法，这时创建当前线程的 threadLocals 变量。下面来看 createMap(t, value) 做什么。

```
    void createMap(Thread t, T firstValue) {
        t.threadLocals = new ThreadLocalMap(this, firstValue);
    }
```

它创建当前线程的 threadLocals 变量。

2．T get()

```
    public T get() {
        //(4) 获取当前线程
        Thread t = Thread.currentThread();
        //(5)获取当前线程的threadLocals变量
        ThreadLocalMap map = getMap(t);
        //(6)如果threadLocals不为null，则返回对应本地变量的值
        if (map != null) {
```

```
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        //(7)threadLocals为空则初始化当前线程的threadLocals成员变量
                return setInitialValue();
    }
```

代码（4）首先获取当前线程实例，如果当前线程的 threadLocals 变量不为 null，则直接返回当前线程绑定的本地变量，否则执行代码（7）进行初始化。setInitialValue() 的代码如下。

```
private T setInitialValue() {
    //(8)初始化为null
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    //(9)如果当前线程的threadLocals变量不为空
    if (map != null)
        map.set(this, value);
    else
    //(10)如果当前线程的threadLocals变量为空
        createMap(t, value);
    return value;
}

protected T initialValue() {
    return null;
}
```

如果当前线程的 threadLocals 变量不为空，则设置当前线程的本地变量值为 null，否则调用 createMap 方法创建当前线程的 createMap 变量。

3．void remove()

```
public void remove() {
     ThreadLocalMap m = getMap(Thread.currentThread());
     if (m != null)
         m.remove(this);
 }
```

如以上代码所示，如果当前线程的 threadLocals 变量不为空，则删除当前线程中指定 ThreadLocal 实例的本地变量。

总结：如图 1-6 所示，在每个线程内部都有一个名为 threadLocals 的成员变量，该变量的类型为 HashMap，其中 key 为我们定义的 ThreadLocal 变量的 this 引用，value 则为我们使用 set 方法设置的值。每个线程的本地变量存放在线程自己的内存变量 threadLocals 中，如果当前线程一直不消亡，那么这些本地变量会一直存在，所以可能会造成内存溢出，因此使用完毕后要记得调用 ThreadLocal 的 remove 方法删除对应线程的 threadLocals 中的本地变量。在高级篇要讲解的 JUC 包里面的 ThreadLocalRandom，就是借鉴 ThreadLocal 的思想实现的，后面会具体讲解。

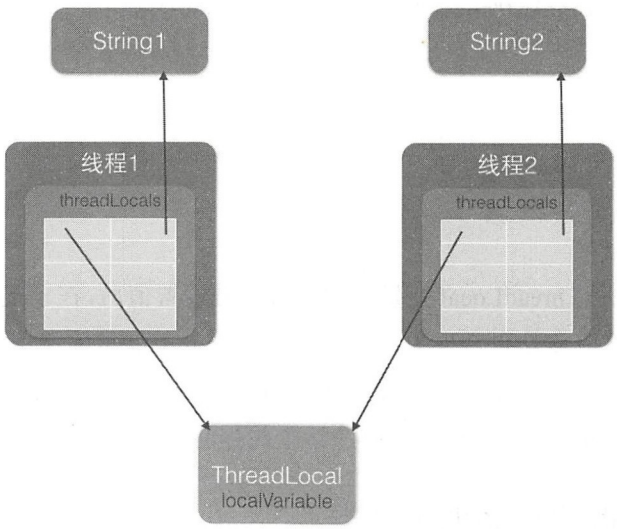


图 1-6

1.11.3　ThreadLocal 不支持继承性

首先看一个例子。

```
public class TestThreadLocal {

    //(1)创建线程变量
    public static ThreadLocal<String> threadLocal = new ThreadLocal<String>();
    public static void main(String[] args) {
```

```
        //(2) 设置线程变量
        threadLocal.set("hello world");
        //(3) 启动子线程
        Thread thread = new Thread(new  Runnable() {
            public void run() {
                //(4) 子线程输出线程变量的值
                System.out.println("thread:" + threadLocal.get());

            }
        });
        thread.start();

        //(5) 主线程输出线程变量的值
        System.out.println("main:" + threadLocal.get());

    }
}
```

输出结果如下。

```
main:hello world
thread:null
```

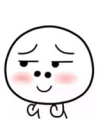
也就是说，同一个 ThreadLocal 变量在父线程中被设置值后，在子线程中是获取不到的。根据上节的介绍，这应该是正常现象，因为在子线程 thread 里面调用 get 方法时当前线程为 thread 线程，而这里调用 set 方法设置线程变量的是 main 线程，两者是不同的线程，自然子线程访问时返回 null。那么有没有办法让子线程能访问到父线程中的值？答案是有。

1.11.4 InheritableThreadLocal 类

为了解决上节提出的问题，InheritableThreadLocal 应运而生。InheritableThreadLocal 继承自 ThreadLocal，其提供了一个特性，就是让子线程可以访问在父线程中设置的本地变量。下面看一下 InheritableThreadLocal 的代码。

```
public class InheritableThreadLocal<T> extends ThreadLocal<T> {

    //(1)
    protected T childValue(T parentValue) {
        return parentValue;
    }
    //(2)
```



```
    ThreadLocalMap getMap(Thread t) {
       return t.inheritableThreadLocals;
    }
    //(3)
    void createMap(Thread t, T firstValue) {
        t.inheritableThreadLocals = new ThreadLocalMap(this, firstValue);
    }
}
```

由如上代码可知，InheritableThreadLocal 继承了 ThreadLocal，并重写了三个方法。由代码（3）可知，InheritableThreadLocal 重写了 createMap 方法，那么现在当第一次调用 set 方法时，创建的是当前线程的 inheritableThreadLocals 变量的实例而不再是 threadLocals。由代码（2）可知，当调用 get 方法获取当前线程内部的 map 变量时，获取的是 inheritableThreadLocals 而不再是 threadLocals。

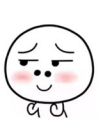
综上可知，在 InheritableThreadLocal 的世界里，变量 inheritableThreadLocals 替代了 threadLocals。

下面我们看一下重写的代码（1）何时执行，以及如何让子线程可以访问父线程的本地变量。这要从创建 Thread 的代码说起，打开 Thread 类的默认构造函数，代码如下。

```
    public Thread(Runnable target) {
     init(null, target, "Thread-" + nextThreadNum(), 0);
}

   private void init(ThreadGroup g, Runnable target, String name,
                     long stackSize, AccessControlContext acc) {
         ...
         //(4)获取当前线程
         Thread parent = currentThread();
         ...
         //(5)如果父线程的inheritableThreadLocals变量不为null
         if (parent.inheritableThreadLocals != null)
         //(6)设置子线程中的inheritableThreadLocals变量
         this.inheritableThreadLocals =
ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
     this.stackSize = stackSize;
     tid = nextThreadID();
}
```

如上代码在创建线程时，在构造函数里面会调用 init 方法。代码(4)获取了当前线程(这里是指 main 函数所在的线程，也就是父线程)，然后代码（5）判断 main 函数所在线程里



面的 inheritableThreadLocals 属性是否为 null，前面我们讲了 InheritableThreadLocal 类的 get 和 set 方法操作的是 inheritableThreadLocals，所以这里的 inheritableThreadLocal 变量不为 null，因此会执行代码（6）。下面看一下 createInheritedMap 的代码。

```
static ThreadLocalMap createInheritedMap(ThreadLocalMap parentMap) {
    return new ThreadLocalMap(parentMap);
}
```

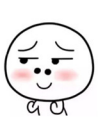
可以看到，在 createInheritedMap 内部使用父线程的 inheritableThreadLocals 变量作为构造函数创建了一个新的 ThreadLocalMap 变量，然后赋值给了子线程的 inheritableThreadLocals 变量。下面我们看看在 ThreadLocalMap 的构造函数内部都做了什么事情。

```
private ThreadLocalMap(ThreadLocalMap parentMap) {
            Entry[] parentTable = parentMap.table;
            int len = parentTable.length;
            setThreshold(len);
            table = new Entry[len];

            for (int j = 0; j < len; j++) {
                Entry e = parentTable[j];
                if (e != null) {
                    @SuppressWarnings("unchecked")
                    ThreadLocal<Object> key = (ThreadLocal<Object>) e.get();
                    if (key != null) {
                        //(7)调用重写的方法
                        Object value = key.childValue(e.value);//返回e.value
                        Entry c = new Entry(key, value);
                        int h = key.threadLocalHashCode & (len - 1);
                        while (table[h] != null)
                            h = nextIndex(h, len);
                        table[h] = c;
                        size++;
                    }
                }
            }
        }
```

在该构造函数内部把父线程的 inheritableThreadLocals 成员变量的值复制到新的 ThreadLocalMap 对象中，其中代码（7）调用了 InheritableThreadLocal 类重写的代码（1）。

总结：InheritableThreadLocal 类通过重写代码（2）和（3）让本地变量保存到了具体



线程的 inheritableThreadLocals 变量里面，那么线程在通过 InheritableThreadLocal 类实例的 set 或者 get 方法设置变量时，就会创建当前线程的 inheritableThreadLocals 变量。当父线程创建子线程时，构造函数会把父线程中 inheritableThreadLocals 变量里面的本地变量复制一份保存到子线程的 inheritableThreadLocals 变量里面。

把 1.11.3 节中的代码（1）修改为

```
    //(1) 创建线程变量
    public static ThreadLocal<String> threadLocal = new InheritableThreadLocal<Stri
ng>();
```

运行结果如下。

```
thread:hello world
main:hello world
```

可见，现在可以从子线程正常获取到线程变量的值了。

那么在什么情况下需要子线程可以获取父线程的 threadlocal 变量呢？情况还是蛮多的，比如子线程需要使用存放在 threadlocal 变量中的用户登录信息，再比如一些中间件需要把统一的 id 追踪的整个调用链路记录下来。 其实子线程使用父线程中的 threadlocal 方法有多种方式，比如创建线程时传入父线程中的变量，并将其复制到子线程中，或者在父线程中构造一个 map 作为参数传递给子线程，但是这些都改变了我们的使用习惯，所以在这些情况下 InheritableThreadLocal 就显得比较有用。

第2章

并发编程的其他基础知识

2.1 什么是多线程并发编程

首先要澄清并发和并行的概念，并发是指同一个时间段内多个任务同时都在执行，并且都没有执行结束，而并行是说在单位时间内多个任务同时在执行。并发任务强调在一个时间段内同时执行，而一个时间段由多个单位时间累积而成，所以说并发的多个任务在单位时间内不一定同时在执行。在单 CPU 的时代多个任务都是并发执行的，这是因为单个 CPU 同时只能执行一个任务。在单 CPU 时代多任务是共享一个 CPU 的，当一个任务占用 CPU 运行时，其他任务就会被挂起，当占用 CPU 的任务时间片用完后，会把 CPU 让给其他任务来使用，所以在单 CPU 时代多线程编程是没有太大意义的，并且线程间频繁的上下文切换还会带来额外开销。

图 2-1 所示为在单个 CPU 上运行两个线程，线程 A 和线程 B 是轮流使用 CPU 进行任务处理的，也就是在某个时间内单个 CPU 只执行一个线程上面的任务。当线程 A 的时间片用完后会进行线程上下文切换，也就是保存当前线程 A 的执行上下文，然后切换到线程 B 来占用 CPU 运行任务。

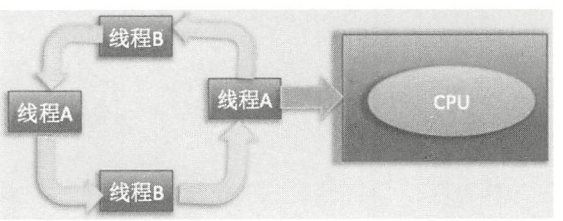


图 2-1

图 2-2 所示为双 CPU 配置，线程 A 和线程 B 各自在自己的 CPU 上执行任务，实现了真正的并行运行。

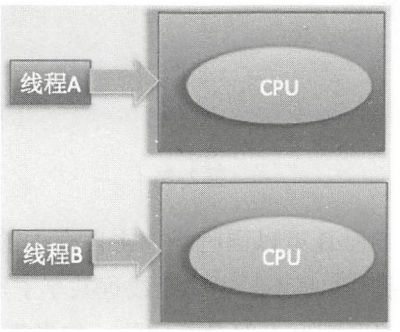


图 2-2

而在多线程编程实践中，线程的个数往往多于 CPU 的个数，所以一般都称多线程并发编程而不是多线程并行编程。

2.2　为什么要进行多线程并发编程

多核 CPU 时代的到来打破了单核 CPU 对多线程效能的限制。多个 CPU 意味着每个线程可以使用自己的 CPU 运行，这减少了线程上下文切换的开销，但随着对应用系统性能和吞吐量要求的提高，出现了处理海量数据和请求的要求，这些都对高并发编程有着迫切的需求。

2.3　Java 中的线程安全问题

谈到线程安全问题，我们先说说什么是共享资源。所谓共享资源，就是说该资源被多个线程所持有或者说多个线程都可以去访问该资源。

线程安全问题是指当多个线程同时读写一个共享资源并且没有任何同步措施时，导致出现脏数据或者其他不可预见的结果的问题，如图 2-3 所示。

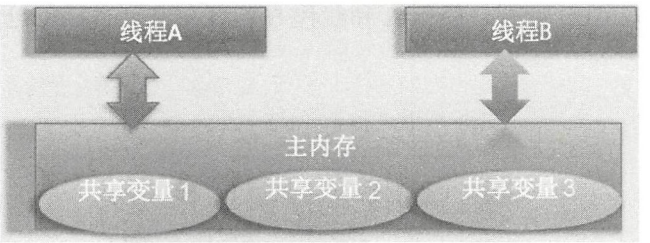


图2-3

在图2-3中，线程A和线程B可以同时操作主内存中的共享变量，那么线程安全问题和共享资源之间是什么关系呢？是不是说多个线程共享了资源，当它们都去访问这个共享资源时就会产生线程安全问题呢？答案是否定的，如果多个线程都只是读取共享资源，而不去修改，那么就不会存在线程安全问题，只有当至少一个线程修改共享资源时才会存在线程安全问题。最典型的就是计数器类的实现，计数变量count本身是一个共享变量，多个线程可以对其进行递增操作，如果不使用同步措施，由于递增操作是获取—计算—保存三步操作，因此可能导致计数不准确，如下所示。

	t1	t2	t3	t4
线程A	从内存读取count值到本线程	递增本线程count的值	写回主内存	
线程B		从内存读取count值到本线程	递增本线程count的值	写回主内存

假如当前count=0，在t1时刻线程A读取count值到本地变量countA。然后在t2时刻递增countA的值为1，同时线程B读取count的值0到本地变量countB，此时countB的值为0（因为countA的值还没有被写入主内存）。在t3时刻线程A才把countA的值1写入主内存，至此线程A一次计数完毕，同时线程B递增CountB的值为1。在t4时刻线程B把countB的值1写入内存，至此线程B一次计数完毕。这里先不考虑内存可见性问题，明明是两次计数，为何最后结果是1而不是2呢？其实这就是共享变量的线程安全问题。那么如何来解决这个问题呢？这就需要在线程访问共享变量时进行适当的同步，在Java中最常见的是使用关键字synchronized进行同步，下面会有具体介绍。

2.4 Java中共享变量的内存可见性问题

谈到内存可见性，我们首先来看看在多线程下处理共享变量时Java的内存模型，如图2-4所示。

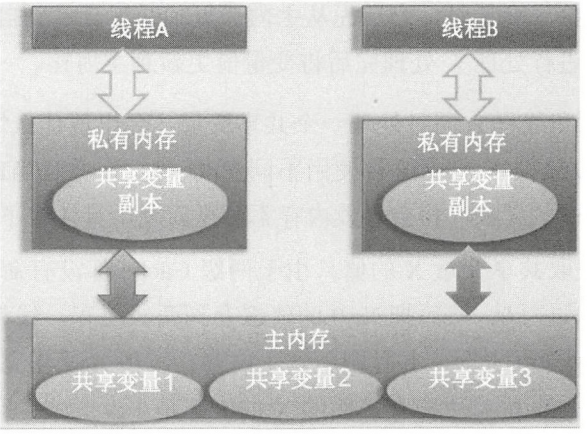


图 2-4

Java 内存模型规定，将所有的变量都存放在主内存中，当线程使用变量时，会把主内存里面的变量复制到自己的工作空间或者叫作工作内存，线程读写变量时操作的是自己工作内存中的变量。Java 内存模型是一个抽象的概念，那么在实际实现中线程的工作内存是什么呢？请看图 2-5。

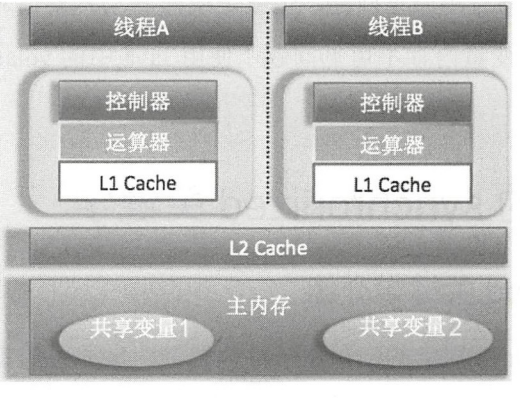


图 2-5

图中所示是一个双核 CPU 系统架构，每个核有自己的控制器和运算器，其中控制器包含一组寄存器和操作控制器，运算器执行算术逻辑运算。每个核都有自己的一级缓存，在有些架构里面还有一个所有 CPU 都共享的二级缓存。那么 Java 内存模型里面的工作内存，就对应这里的 L1 或者 L2 缓存或者 CPU 的寄存器。

当一个线程操作共享变量时，它首先从主内存复制共享变量到自己的工作内存，然后对工作内存里的变量进行处理，处理完后将变量值更新到主内存。

那么假如线程 A 和线程 B 同时处理一个共享变量，会出现什么情况？我们使用图 2-5 所示 CPU 架构，假设线程 A 和线程 B 使用不同 CPU 执行，并且当前两级 Cache 都为空，那么这时候由于 Cache 的存在，将会导致内存不可见问题，具体看下面的分析。

- 线程 A 首先获取共享变量 X 的值，由于两级 Cache 都没有命中，所以加载主内存中 X 的值，假如为 0。然后把 X=0 的值缓存到两级缓存，线程 A 修改 X 的值为 1，然后将其写入两级 Cache，并且刷新到主内存。线程 A 操作完毕后，线程 A 所在的 CPU 的两级 Cache 内和主内存里面的 X 的值都是 1。
- 线程 B 获取 X 的值，首先一级缓存没有命中，然后看二级缓存，二级缓存命中了，所以返回 X= 1；到这里一切都是正常的，因为这时候主内存中也是 X=1。然后线程 B 修改 X 的值为 2，并将其存放到线程 2 所在的一级 Cache 和共享二级 Cache 中，最后更新主内存中 X 的值为 2；到这里一切都是好的。
- 线程 A 这次又需要修改 X 的值，获取时一级缓存命中，并且 X=1，到这里问题就出现了，明明线程 B 已经把 X 的值修改为了 2，为何线程 A 获取的还是 1 呢？这就是共享变量的内存不可见问题，也就是线程 B 写入的值对线程 A 不可见。

那么如何解决共享变量内存不可见问题？使用 Java 中的 volatile 关键字就可以解决这个问题，下面会有讲解。

2.5 Java 中的 synchronized 关键字

2.5.1 synchronized 关键字介绍

synchronized 块是 Java 提供的一种原子性内置锁，Java 中的每个对象都可以把它当作一个同步锁来使用，这些 Java 内置的使用者看不到的锁被称为内部锁，也叫作监视器锁。线程的执行代码在进入 synchronized 代码块前会自动获取内部锁，这时候其他线程访问该同步代码块时会被阻塞挂起。拿到内部锁的线程会在正常退出同步代码块或者抛出异常后或者在同步块内调用了该内置锁资源的 wait 系列方法时释放该内置锁。内置锁是排它锁，也就是当一个线程获取这个锁后，其他线程必须等待该线程释放锁后才能获取该锁。

另外，由于 Java 中的线程是与操作系统的原生线程一一对应的，所以当阻塞一个线程时，需要从用户态切换到内核态执行阻塞操作，这是很耗时的操作，而 synchronized 的使用就会导致上下文切换。

2.5.2　synchronized 的内存语义

前面介绍了共享变量内存可见性问题主要是由于线程的工作内存导致的，下面我们来讲解 synchronized 的一个内存语义，这个内存语义就可以解决共享变量内存可见性问题。进入 synchronized 块的内存语义是把在 synchronized 块内使用到的变量从线程的工作内存中清除，这样在 synchronized 块内使用到该变量时就不会从线程的工作内存中获取，而是直接从主内存中获取。退出 synchronized 块的内存语义是把在 synchronized 块内对共享变量的修改刷新到主内存。

其实这也是加锁和释放锁的语义，当获取锁后会清空锁块内本地内存中将会被用到的共享变量，在使用这些共享变量时从主内存进行加载，在释放锁时将本地内存中修改的共享变量刷新到主内存。

除可以解决共享变量内存可见性问题外，synchronized 经常被用来实现原子性操作。另外请注意，synchronized 关键字会引起线程上下文切换并带来线程调度开销。

2.6　Java 中的 volatile 关键字

上面介绍了使用锁的方式可以解决共享变量内存可见性问题，但是使用锁太笨重，因为它会带来线程上下文的切换开销。对于解决内存可见性问题，Java 还提供了一种弱形式的同步，也就是使用 volatile 关键字。该关键字可以确保对一个变量的更新对其他线程马上可见。当一个变量被声明为 volatile 时，线程在写入变量时不会把值缓存在寄存器或者其他地方，而是会把值刷新回主内存。当其他线程读取该共享变量时，会从主内存重新获取最新值，而不是使用当前线程的工作内存中的值。volatile 的内存语义和 synchronized 有相似之处，具体来说就是，当线程写入了 volatile 变量值时就等价于线程退出 synchronized 同步块（把写入工作内存的变量值同步到主内存），读取 volatile 变量值时就相当于进入同步块（先清空本地内存变量值，再从主内存获取最新值）。

下面看一个使用 volatile 关键字解决内存可见性问题的例子。如下代码中的共享变量

value 是线程不安全的，因为这里没有使用适当的同步措施。

```
public class ThreadNotSafeInteger {

    private int value;

    public int get() {
        return value;
    }

    public void set(int value) {
        this.value = value;
    }
}
```

首先来看使用 synchronized 关键字进行同步的方式。

```
public class ThreadSafeInteger {

    private int value;

    public synchronized int get() {
        return value;
    }

    public synchronized  void set(int value) {
        this.value = value;
    }
}
```

然后是使用 volatile 进行同步。

```
public class ThreadSafeInteger {

    private volatile int value;

    public int get() {
        return value;
    }

    public void set(int value) {
        this.value = value;
    }
}
```

在这里使用 synchronized 和使用 volatile 是等价的，都解决了共享变量 value 的内存可见性问题，但是前者是独占锁，同时只能有一个线程调用 get() 方法，其他调用线程会被阻塞，同时会存在线程上下文切换和线程重新调度的开销，这也是使用锁方式不好的地方。而后者是非阻塞算法，不会造成线程上下文切换的开销。

但并非在所有情况下使用它们都是等价的，volatile 虽然提供了可见性保证，但并不保证操作的原子性。

那么一般在什么时候才使用 volatile 关键字呢？

- 写入变量值不依赖变量的当前值时。因为如果依赖当前值，将是获取—计算—写入三步操作，这三步操作不是原子性的，而 volatile 不保证原子性。
- 读写变量值时没有加锁。因为加锁本身已经保证了内存可见性，这时候不需要把变量声明为 volatile 的。

2.7　Java 中的原子性操作

所谓原子性操作，是指执行一系列操作时，这些操作要么全部执行，要么全部不执行，不存在只执行其中一部分的情况。在设计计数器时一般都先读取当前值，然后 +1，再更新。这个过程是读—改—写的过程，如果不能保证这个过程是原子性的，那么就会出现线程安全问题。如下代码是线程不安全的，因为不能保证 ++value 是原子性操作。

```
public class ThreadNotSafeCount {

    private  Long value;

    public Long getCount() {
        return value;
    }

    public void inc() {
        ++value;
    }
}
```

使用 Javap -c 命令查看汇编代码，如下所示。

```
public void inc();
  Code:
```

```
  0: aload_0
  1: dup
  2: getfield      #2                  // Field value:J
  5: lconst_1
  6: ladd
  7: putfield      #2                  // Field value:J
 10: return
```

由此可见，简单的 ++value 由 2、5、6、7 四步组成，其中第 2 步是获取当前 value 的值并放入栈顶，第 5 步把常量 1 放入栈顶，第 6 步把当前栈顶中两个值相加并把结果放入栈顶，第 7 步则把栈顶的结果赋给 value 变量。因此，Java 中简单的一句 ++value 被转换为汇编后就不具有原子性了。

那么如何才能保证多个操作的原子性呢？最简单的方法就是使用 synchronized 关键字进行同步，修改代码如下。

```
public class ThreadSafeCount {

    private  Long value;

    public synchronized Long getCount() {
        return value;
    }

    public synchronized void inc() {
        ++value;
    }
}
```

使用 synchronized 关键字的确可以实现线程安全性，即内存可见性和原子性，但是 synchronized 是独占锁，没有获取内部锁的线程会被阻塞掉，而这里的 getCount 方法只是读操作，多个线程同时调用不会存在线程安全问题。但是加了关键字 synchronized 后，同一时间就只能有一个线程可以调用，这显然大大降低了并发性。你也许会问，既然是只读操作，那为何不去掉 getCount 方法上的 synchronized 关键字呢？其实是不能去掉的，别忘了这里要靠 synchronized 来实现 value 的内存可见性。那么有没有更好的实现呢？答案是肯定的，下面将讲到的在内部使用非阻塞 CAS 算法实现的原子性操作类 AtomicLong 就是一个不错的选择。

2.8　Java 中的 CAS 操作

在 Java 中，锁在并发处理中占据了一席之地，但是使用锁有一个不好的地方，就是当一个线程没有获取到锁时会被阻塞挂起，这会导致线程上下文的切换和重新调度开销。Java 提供了非阻塞的 volatile 关键字来解决共享变量的可见性问题，这在一定程度上弥补了锁带来的开销问题，但是 volatile 只能保证共享变量的可见性，不能解决读—改—写等的原子性问题。CAS 即 Compare and Swap，其是 JDK 提供的非阻塞原子性操作，它通过硬件保证了比较—更新操作的原子性。JDK 里面的 Unsafe 类提供了一系列的 compareAndSwap* 方法，下面以 compareAndSwapLong 方法为例进行简单介绍。

- boolean compareAndSwapLong(Object obj,long valueOffset,long expect, long update) 方法：其中 compareAndSwap 的意思是比较并交换。CAS 有四个操作数，分别为：对象内存位置、对象中的变量的偏移量、变量预期值和新的值。其操作含义是，如果对象 obj 中内存偏移量为 valueOffset 的变量值为 expect，则使用新的值 update 替换旧的值 expect。这是处理器提供的一个原子性指令。

关于 CAS 操作有个经典的 ABA 问题，具体如下：假如线程 I 使用 CAS 修改初始值为 A 的变量 X，那么线程 I 会首先去获取当前变量 X 的值（为 A），然后使用 CAS 操作尝试修改 X 的值为 B，如果使用 CAS 操作成功了，那么程序运行一定是正确的吗？其实未必，这是因为有可能在线程 I 获取变量 X 的值 A 后，在执行 CAS 前，线程 II 使用 CAS 修改了变量 X 的值为 B，然后又使用 CAS 修改了变量 X 的值为 A。所以虽然线程 I 执行 CAS 时 X 的值是 A，但是这个 A 已经不是线程 I 获取时的 A 了。这就是 ABA 问题。

ABA 问题的产生是因为变量的状态值产生了环形转换，就是变量的值可以从 A 到 B，然后再从 B 到 A。如果变量的值只能朝着一个方向转换，比如 A 到 B，B 到 C，不构成环形，就不会存在问题。JDK 中的 AtomicStampedReference 类给每个变量的状态值都配备了一个时间戳，从而避免了 ABA 问题的产生。

2.9　Unsafe 类

2.9.1　Unsafe 类中的重要方法

JDK 的 rt.jar 包中的 Unsafe 类提供了硬件级别的原子性操作，Unsafe 类中的方法都是 native 方法，它们使用 JNI 的方式访问本地 C++ 实现库。下面我们来了解一下 Unsafe 提

供的几个主要的方法以及编程时如何使用 Unsafe 类做一些事情。

- long objectFieldOffset(Field field) 方法：返回指定的变量在所属类中的内存偏移地址，该偏移地址仅仅在该 Unsafe 函数中访问指定字段时使用。如下代码使用 Unsafe 类获取变量 value 在 AtomicLong 对象中的内存偏移。

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

- int arrayBaseOffset(Class arrayClass) 方法：获取数组中第一个元素的地址。
- int arrayIndexScale(Class arrayClass) 方法：获取数组中一个元素占用的字节。
- boolean compareAndSwapLong(Object obj, long offset, long expect, long update) 方法：比较对象 obj 中偏移量为 offset 的变量的值是否与 expect 相等，相等则使用 update 值更新，然后返回 true，否则返回 false。
- public native long getLongvolatile(Object obj, long offset) 方法：获取对象 obj 中偏移量为 offset 的变量对应 volatile 语义的值。
- void putLongvolatile(Object obj, long offset, long value) 方法：设置 obj 对象中 offset 偏移的类型为 long 的 field 的值为 value，支持 volatile 语义。
- void putOrderedLong(Object obj, long offset, long value) 方法：设置 obj 对象中 offset 偏移地址对应的 long 型 field 的值为 value。这是一个有延迟的 putLongvolatile 方法，并且不保证值修改对其他线程立刻可见。只有在变量使用 volatile 修饰并且预计会被意外修改时才使用该方法。
- void park(boolean isAbsolute, long time) 方法：阻塞当前线程，其中参数 isAbsolute 等于 false 且 time 等于 0 表示一直阻塞。time 大于 0 表示等待指定的 time 后阻塞线程会被唤醒，这个 time 是个相对值，是个增量值，也就是相对当前时间累加 time 后当前线程就会被唤醒。如果 isAbsolute 等于 true，并且 time 大于 0，则表示阻塞的线程到指定的时间点后会被唤醒，这里 time 是个绝对时间，是将某个时间点换算为 ms 后的值。另外，当其他线程调用了当前阻塞线程的 interrupt 方法而中断了当前线程时，当前线程也会返回，而当其他线程调用了 unPark 方法并且把当前线程作为参数时当前线程也会返回。

- void unpark(Object thread) 方法：唤醒调用 park 后阻塞的线程。

下面是 JDK8 新增的函数，这里只列出 Long 类型操作。

- long getAndSetLong(Object obj, long offset, long update) 方法：获取对象 obj 中偏移量为 offset 的变量 volatile 语义的当前值，并设置变量 volatile 语义的值为 update。

```
public final long getAndSetLong(Object obj, long offset, long update)
  {
    long l;
    do
    {
      l = getLongvolatile(obj, offset);//(1)
     } while (!compareAndSwapLong(obj, offset, l, update));
     return l;
  }
```

由以上代码可知，首先（1）处的 getLongvolatile 获取当前变量的值，然后使用 CAS 原子操作设置新值。这里使用 while 循环是考虑到，在多个线程同时调用的情况下 CAS 失败时需要重试。

- long getAndAddLong(Object obj, long offset, long addValue) 方法：获取对象 obj 中偏移量为 offset 的变量 volatile 语义的当前值，并设置变量值为原始值 +addValue。

```
public final long getAndAddLong(Object obj, long offset, long addValue)
{
  long l;
  do
  {
    l = getLongvolatile(obj, offset);
  } while (!compareAndSwapLong(obj, offset, l, l + addValue));
  return l;
 }
```

类似 getAndSetLong 的实现，只是这里进行 CAS 操作时使用了原始值 + 传递的增量参数 addValue 的值。

2.9.2　如何使用 Unsafe 类

看到 Unsafe 这个类如此厉害，你肯定会忍不住试一下下面的代码，期望能够使用 Unsafe 做点事情。

```
public class TestUnSafe {
```

```
    //获取Unsafe的实例（2.2.1）
    static final Unsafe unsafe = Unsafe.getUnsafe();

    //记录变量state在类TestUnSafe中的偏移值（2.2.2）
    static final long stateOffset;

    //变量(2.2.3)
    private volatile long state=0;

    static {

        try {
            //获取state变量在类TestUnSafe中的偏移值(2.2.4)
            stateOffset = unsafe.objectFieldOffset(TestUnSafe.class.
             getDeclaredField("state"));

        } catch (Exception ex) {

            System.out.println(ex.getLocalizedMessage());
            throw new Error(ex);
        }

    }

    public static void main(String[] args) {

        //创建实例，并且设置state值为1(2.2.5)
        TestUnSafe test = new TestUnSafe();
        //(2.2.6)
        Boolean sucess = unsafe.compareAndSwapInt(test, stateOffset, 0, 1);
        System.out.println(sucess);

    }
}
```

在如上代码中，代码（2.2.1）获取了 Unsafe 的一个实例，代码（2.2.3）创建了一个变量 state 并初始化为 0。

代码（2.2.4）使用 unsafe.objectFieldOffset 获取 TestUnSafe 类里面的 state 变量，在 TestUnSafe 对象里面的内存偏移量地址并将其保存到 stateOffset 变量中。

代码（2.2.6）调用创建的 unsafe 实例的 compareAndSwapInt 方法，设置 test 对象的

state 变量的值。具体意思是，如果 test 对象中内存偏移量为 stateOffset 的 state 变量的值为 0，则更新该值为 1。

运行上面的代码，我们期望输出 true，然而执行后会输出如下结果。

```
<terminated> TestUnSafe [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.SecurityException: Unsafe
        at sun.misc.Unsafe.getUnsafe(Unsafe.java:90)
        at com.zlx.con.program.example.TestUnSafe.<clinit>(TestUnSafe.java:17)
```

为找出原因，必然要查看 getUnsafe 的代码。

```
private static final Unsafe theUnsafe = new Unsafe();

 public static Unsafe getUnsafe()
{
    //(2.2.7)
    Class localClass = Reflection.getCallerClass();

   //(2.2.8)
    if (!VM.isSystemDomainLoader(localClass.getClassLoader())) {
      throw new SecurityException("Unsafe");
    }
    return theUnsafe;
}

  //判断paramClassLoader是不是BootStrap类加载器(2.2.9)
 public static boolean isSystemDomainLoader(ClassLoader paramClassLoader)
  {
    return paramClassLoader == null;
  }
```

代码（2.2.7）获取调用 getUnsafe 这个方法的对象的 Class 对象，这里是 TestUnSafe.class。

代码（2.2.8）判断是不是 Bootstrap 类加载器加载的 localClass，在这里是看是不是 Bootstrap 加载器加载了 TestUnSafe.class。很明显由于 TestUnSafe.class 是使用 AppClassLoader 加载的，所以这里直接抛出了异常。

思考一下，这里为何要有这个判断？ 我们知道 Unsafe 类是 rt.jar 包提供的，rt.jar 包里面的类是使用 Bootstrap 类加载器加载的，而我们的启动 main 函数所在的类是使用 AppClassLoader 加载的，所以在 main 函数里面加载 Unsafe 类时，根据委托机制，会委托

给 Bootstrap 去加载 Unsafe 类。

如果没有代码（2.2.8）的限制，那么我们的应用程序就可以随意使用 Unsafe 做事情了，而 Unsafe 类可以直接操作内存，这是不安全的，所以 JDK 开发组特意做了这个限制，不让开发人员在正规渠道使用 Unsafe 类，而是在 rt.jar 包里面的核心类中使用 Unsafe 功能。

如果开发人员真的想要实例化 Unsafe 类，那该如何做？

方法有多种，既然从正规渠道访问不了，那么就玩点黑科技，使用万能的反射来获取 Unsafe 实例方法。

```
public class TestUnSafe {

    static final Unsafe unsafe;

    static final long stateOffset;

    private volatile long state = 0;

    static {

        try {

            //使用反射获取Unsafe的成员变量theUnsafe
            Field field = Unsafe.class.getDeclaredField("theUnsafe");

            // 设置为可存取
            field.setAccessible(true);

            // 获取该变量的值
            unsafe = (Unsafe) field.get(null);

            //获取state在TestUnSafe中的偏移量
            stateOffset = unsafe.objectFieldOffset(TestUnSafe.class.
             getDeclaredField("state"));

        } catch (Exception ex) {

            System.out.println(ex.getLocalizedMessage());
            throw new Error(ex);
        }
```

```
    }

    public static void main(String[] args) {

        TestUnSafe test = new TestUnSafe();
        Boolean sucess = unsafe.compareAndSwapInt(test, stateOffset, 0, 1);
        System.out.println(sucess);

    }
}
```

在如上代码中，通过代码（2.2.10）、代码（2.2.11）和代码（2.2.12）反射获取 unsafe 的实例，运行后输出结果如下。

```
<terminated> TestUnSafe [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
true
```

2.10　Java 指令重排序

Java 内存模型允许编译器和处理器对指令重排序以提高运行性能，并且只会对不存在数据依赖性的指令重排序。在单线程下重排序可以保证最终执行的结果与程序顺序执行的结果一致，但是在多线程下就会存在问题。

下面看一个例子。

```
int a = 1;(1)
int b = 2;(2)
int c= a + b;(3)
```

在如上代码中，变量 c 的值依赖 a 和 b 的值，所以重排序后能够保证（3）的操作在（2）（1）之后，但是（1）（2）谁先执行就不一定了，这在单线程下不会存在问题，因为并不影响最终结果。

下面看一个多线程的例子。

```
public static class ReadThread extends Thread {
        public void run() {

            while(!Thread.currentThread().isInterrupted()){
                if(ready){//(1)
                    System.out.println(num+num);//(2)
```

```
                }
                System.out.println("read thread....");
            }

        }
    }

public static class Writethread extends Thread {
    public void run() {
        num = 2;//(3)
        ready = true;//(4)
        System.out.println("writeThread set over...");
    }
}

private static int num =0;
private static boolean ready = false;

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

    ReadThread rt = new ReadThread();
    rt.start();

    Writethread  wt = new Writethread();
    wt.start();

    Thread.sleep(10);
    rt.interrupt();
    System.out.println("main exit");
}
```

首先这段代码里面的变量没有被声明为 volatile 的，也没有使用任何同步措施，所以在多线程下存在共享变量内存可见性问题。这里先不谈内存可见性问题，因为通过把变量声明为 volatile 的本身就可以避免指令重排序问题。

这里先看看指令重排序会造成什么影响，如上代码在不考虑内存可见性问题的情况下一定会输出 4？答案是不一定，由于代码（1）（2）（3）（4）之间不存在依赖关系，所以写线程的代码（3）（4）可能被重排序为先执行（4）再执行（3），那么执行（4）后，读线程可能已经执行了（1）操作，并且在（3）执行前开始执行（2）操作，这时候输出结果为 0 而不是 4。

重排序在多线程下会导致非预期的程序执行结果，而使用 volatile 修饰 ready 就可以

避免重排序和内存可见性问题。

写 volatile 变量时，可以确保 volatile 写之前的操作不会被编译器重排序到 volatile 写之后。读 volatile 变量时，可以确保 volatile 读之后的操作不会被编译器重排序到 volatile 读之前。

2.11　伪共享

2.11.1　什么是伪共享

为了解决计算机系统中主内存与 CPU 之间运行速度差问题，会在 CPU 与主内存之间添加一级或者多级高速缓冲存储器（Cache）。这个 Cache 一般是被集成到 CPU 内部的，所以也叫 CPU Cache，图 2-6 所示是两级 Cache 结构。

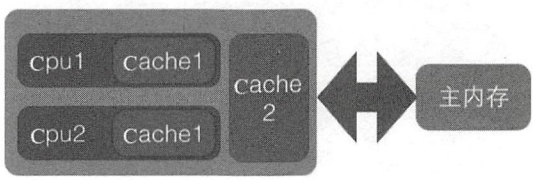


图 2-6

在 Cache 内部是按行存储的，其中每一行称为一个 Cache 行。Cache 行（如图 2-7 所示）是 Cache 与主内存进行数据交换的单位，Cache 行的大小一般为 2 的幂次数字节。

图 2-7

当 CPU 访问某个变量时，首先会去看 CPU Cache 内是否有该变量，如果有则直接从中获取，否则就去主内存里面获取该变量，然后把该变量所在内存区域的一个 Cache 行大小的内存复制到 Cache 中。由于存放到 Cache 行的是内存块而不是单个变量，所以可能会

把多个变量存放到一个 Cache 行中。当多个线程同时修改一个缓存行里面的多个变量时，由于同时只能有一个线程操作缓存行，所以相比将每个变量放到一个缓存行，性能会有所下降，这就是伪共享，如图 2-8 所示。

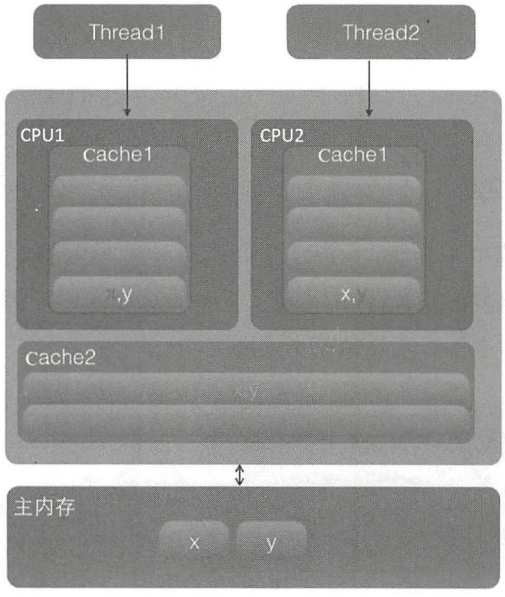


图 2-8

在该图中，变量 x 和 y 同时被放到了 CPU 的一级和二级缓存，当线程 1 使用 CPU1 对变量 x 进行更新时，首先会修改 CPU1 的一级缓存变量 x 所在的缓存行，这时候在缓存一致性协议下，CPU2 中变量 x 对应的缓存行失效。那么线程 2 在写入变量 x 时就只能去二级缓存里查找，这就破坏了一级缓存。而一级缓存比二级缓存更快，这也说明了多个线程不可能同时去修改自己所使用的 CPU 中相同缓存行里面的变量。更坏的情况是，如果 CPU 只有一级缓存，则会导致频繁地访问主内存。

2.11.2 为何会出现伪共享

伪共享的产生是因为多个变量被放入了一个缓存行中，并且多个线程同时去写入缓存行中不同的变量。那么为何多个变量会被放入一个缓存行呢？其实是因为缓存与内存交换数据的单位就是缓存行，当 CPU 要访问的变量没有在缓存中找到时，根据程序运行的局

部性原理，会把该变量所在内存中大小为缓存行的内存放入缓存行。

```
long a;
long b;
long c;
long d;
```

如上代码声明了四个 long 变量，假设缓存行的大小为 32 字节，那么当 CPU 访问变量 a 时，发现该变量没有在缓存中，就会去主内存把变量 a 以及内存地址附近的 b、c、d 放入缓存行。也就是地址连续的多个变量才有可能会被放到一个缓存行中。当创建数组时，数组里面的多个元素就会被放入同一个缓存行。那么在单线程下多个变量被放入同一个缓存行对性能有影响吗？其实在正常情况下单线程访问时将数组元素放入一个或者多个缓存行对代码执行是有利的，因为数据都在缓存中，代码执行会更快，请对比下面代码的执行。

代码（1）

```
public class TestForContent {

    static final int LINE_NUM = 1024;
    static final int COLUM_NUM = 1024;
    public static void main(String[] args) {

        long [][] array = new long[LINE_NUM][COLUM_NUM];

        long startTime = System.currentTimeMillis();
        for(int i =0;i<LINE_NUM;++i){
            for(int j=0;j<COLUM_NUM;++j){
                array[i][j] = i*2+j;
            }
        }
        long endTime = System.currentTimeMillis();
        long cacheTime = endTime - startTime;
        System.out.println("cache time:" + cacheTime);

    }
}
```

代码（2）

```
public class TestForContent2 {

    static final int LINE_NUM = 1024;
    static final int COLUM_NUM = 1024;
```

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

        long [][] array = new long[LINE_NUM][COLUM_NUM];

        long startTime = System.currentTimeMillis();
        for(int i =0;i<COLUM_NUM;++i){
            for(int j=0;j<LINE_NUM;++j){
                array[j][i] = i*2+j;
            }
        }
        long endTime = System.currentTimeMillis();

        System.out.println("no cache time:" + (endTime - startTime));

    }

}
```

在笔者的 mac 电脑上执行代码（1）多次，耗时均在 10ms 以下，执行代码（2）多次，耗时均在 10ms 以上。显然代码（1）比代码（2）执行得快，这是因为数组内数组元素的内存地址是连续的，当访问数组第一个元素时，会把第一个元素后的若干元素一块放入缓存行，这样顺序访问数组元素时会在缓存中直接命中，因而就不会去主内存读取了，后续访问也是这样。也就是说，当顺序访问数组里面元素时，如果当前元素在缓存没有命中，那么会从主内存一下子读取后续若干个元素到缓存，也就是一次内存访问可以让后面多次访问直接在缓存中命中。而代码（2）是跳跃式访问数组元素的，不是顺序的，这破坏了程序访问的局部性原则，并且缓存是有容量控制的，当缓存满了时会根据一定淘汰算法替换缓存行，这会导致从内存置换过来的缓存行的元素还没等到被读取就被替换掉了。

所以在单个线程下顺序修改一个缓存行中的多个变量，会充分利用程序运行的局部性原则，从而加速了程序的运行。而在多线程下并发修改一个缓存行中的多个变量时就会竞争缓存行，从而降低程序运行性能。

2.11.3 如何避免伪共享

在 JDK 8 之前一般都是通过字节填充的方式来避免该问题，也就是创建一个变量时使用填充字段填充该变量所在的缓存行，这样就避免了将多个变量存放在同一个缓存行中，例如如下代码。

```
public final static class FilledLong {
        public volatile long value = 0L;
        public long p1, p2, p3, p4, p5, p6;
    }
```

假如缓存行为64字节，那么我们在FilledLong类里面填充了6个long类型的变量，每个long类型变量占用8字节，加上value变量的8字节总共56字节。另外，这里FilledLong是一个类对象，而类对象的字节码的对象头占用8字节，所以一个FilledLong对象实际会占用64字节的内存，这正好可以放入一个缓存行。

JDK 8提供了一个sun.misc.Contended注解，用来解决伪共享问题。将上面代码修改为如下。

```
@sun.misc.Contended
  public final static class FilledLong {
        public volatile long value = 0L;
    }
```

在这里注解用来修饰类，当然也可以修饰变量，比如在Thread类中。

```
/** The current seed for a ThreadLocalRandom */
@sun.misc.Contended("tlr")
long threadLocalRandomSeed;

/** Probe hash value; nonzero if threadLocalRandomSeed initialized */
@sun.misc.Contended("tlr")
int threadLocalRandomProbe;

/** Secondary seed isolated from public ThreadLocalRandom sequence */
@sun.misc.Contended("tlr")
int threadLocalRandomSecondarySeed;
```

Thread类里面这三个变量默认被初始化为0，这三个变量会在ThreadLocalRandom类中使用，后面章节会专门讲解ThreadLocalRandom的实现原理。

需要注意的是，在默认情况下，@Contended注解只用于Java核心类，比如rt包下的类。如果用户类路径下的类需要使用这个注解，则需要添加JVM参数：-XX:-RestrictContended。填充的宽度默认为128，要自定义宽度则可以设置-XX:ContendedPaddingWidth参数。

2.11.4 小结

本节讲述了伪共享是如何产生的，以及如何避免，并证明在多线程下访问同一个缓存行的多个变量时才会出现伪共享，在单线程下访问一个缓存行里面的多个变量反而会对程序运行起到加速作用。本节的这些知识为后面高级篇讲解的 LongAdder 的实现原理奠定了基础。

2.12 锁的概述

2.12.1 乐观锁与悲观锁

乐观锁和悲观锁是在数据库中引入的名词，但是在并发包锁里面也引入了类似的思想，所以这里还是有必要讲解下。

悲观锁指对数据被外界修改持保守态度，认为数据很容易就会被其他线程修改，所以在数据被处理前先对数据进行加锁，并在整个数据处理过程中，使数据处于锁定状态。悲观锁的实现往往依靠数据库提供的锁机制，即在数据库中，在对数据记录操作前给记录加排它锁。如果获取锁失败，则说明数据正在被其他线程修改，当前线程则等待或者抛出异常。如果获取锁成功，则对记录进行操作，然后提交事务后释放排它锁。

下面我们看一个典型的例子，看它如何使用悲观锁来避免多线程同时对一个记录进行修改。

```
public int updateEntry(long id){
    //(1)使用悲观锁获取指定记录
    EntryObject entry = query("select * from table1 where id = #{id} for
update",id);

   //(2)修改记录内容，根据计算修改entry记录的属性
    String name = generatorName(entry);
    entry.setName(name);
    ......

   //(3)update操作
   int count = update("update table1 set name=#{name},age=#{age} where id
    =#{id}",entry);
   return count;
```

```
}
```

对于如上代码，假设 updateEntry、query、update 方法都使用了事务切面的方法，并且事务传播性被设置为 required。执行 updateEntry 方法时如果上层调用方法里面没有开启事务，则会即时开启一个事务，然后执行代码（1）。代码（1）调用了 query 方法，其根据指定 id 从数据库里面查询出一个记录。由于事务传播性为 requried，所以执行 query 时没有开启新的事务，而是加入了 updateEntry 开启的事务，也就是在 updateEntry 方法执行完毕提交事务时，query 方法才会被提交，就是说记录的锁定会持续到 updateEntry 执行结束。

代码（2）则对获取的记录进行修改，代码（3）把修改的内容写回数据库，同样代码（3）的 update 方法也没有开启新的事务，而是加入了 updateEntry 的事务。也就是 updateEntry、query、update 方法共用同一个事务。

当多个线程同时调用 updateEntry 方法，并且传递的是同一个 id 时，只有一个线程执行代码（1）会成功，其他线程则会被阻塞，这是因为在同一时间只有一个线程可以获取对应记录的锁，在获取锁的线程释放锁前（updateEntry 执行完毕，提交事务前），其他线程必须等待，也就是在同一时间只有一个线程可以对该记录进行修改。

乐观锁是相对悲观锁来说的，它认为数据在一般情况下不会造成冲突，所以在访问记录前不会加排它锁，而是在进行数据提交更新时，才会正式对数据冲突与否进行检测。具体来说，根据 update 返回的行数让用户决定如何去做。将上面的例子改为使用乐观锁的代码如下。

```
public int updateEntry(long id){
    //(1)使用乐观锁获取指定记录
    EntryObject entry = query("select * from table1 where id = #{id}",id);

   //(2)修改记录内容，version字段不能被修改
    String name = generatorName(entry);
    entry.setName(name);
    ……

   //(3)update操作
   int count = update("update table1 set name=#{name},age=#{age},version=${versi
on}+1 where id =#{id} and version=#{version}",entry);
   return count;
}
```

在如上代码中，当多个线程调用 updateEntry 方法并且传递相同的 id 时，多个线程可