一、多线程

2. synchronized

在并发编程中，synchronized大家都肯定用过，一般情况下，我们会把synchronized称为重量级锁。主要原因，是因为JDK1.6之前，synchronized是一个重量级锁相比于JUC的锁显得非常笨重，存在性能问题。JDK1.6及之后，Java对synchronized进行的了一系列优化，性能与JUC的锁不相上下

synchronized保证方法或者代码块在运行时，同一时刻只有一个线程执行代码块，还可以保证共享变量的内存可见性，也可以保证修饰的代码块重排序也不会影响其执行结果。

一句话：synchronized可以保证并发程序的**原子性，可见性，有序性。**

synchronized可以修饰方法和代码块。

- 方法：可修饰静态方法和非静态方法
- 代码块：同步代码块的锁对象可以为当前实例对象、字节码对象（class）、其他实例对象

2.1 如何解决可见性问题？

JMM关于synchronized的两条规定：

- 线程解锁前：必须把自己本地内存中共享变量的最新值刷新到主内存中
- 线程加锁时：将清空本地内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值

在可见性案例中，做如下修改：

```
while (flag) {
    //在死循环中添加同步代码块，可以解决可见性问题
    synchronized (this) {
    }
}
```

synchronized实现可见性的过程

1. 获得互斥锁（同步获取锁）
2. 清空本地内存
3. 从主内存拷贝变量的最新副本到本地内存
4. 执行代码
5. 将更改后的共享变量的值刷新到主内存
6. 释放互斥锁

2.2 同步原理剖析

synchronized是如何实现同步的呢?

同步操作主要是monitorenter和monitorexit这两个jvm指令实现的，先写一段简单的代码：

```
public class Demo05Synchronized {
    public synchronized void increase(){
        System.out.println("synchronized 方法");
    }
    public void syncBlock(){
        synchronized (this){
            System.out.println("synchronized 块");
        }
    }
}

```

在cmd命令行执行javac编译和`javap -c ***.class`生成class文件对应的字节码指令

```
javac Demo05Synchronized.java
javap -c Demo05Synchronized.class
```

从结果可以看出，同步代码块使用的是monitorenter和monitorexit这两个jvm指令

```
//同步方法
public synchronized void increase();
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED //ACC_SYNCHRONIZED标记
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2  // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3  // String synchronized 方法
         5: invokevirtual #4  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
         8: return

 //同步块
  public void syncBlock();
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=1
         0: aload_0
         1: dup
         2: astore_1
         3: monitorenter      //monitorenter指令进入同步块
         4: getstatic     #2  // Field
java/lang/System.out:Ljava/io/PrintStream;
         7: ldc           #5  // String synchronized 块
         9: invokevirtual #4  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
        12: aload_1
        13: monitorexit       //monitorexit指令退出同步块
        14: goto          22
```

```
        17: astore_2
        18: aload_1
        19: monitorexit       //monitorexit指令退出同步块
        20: aload_2
        21: athrow
        22: return
```

从上述字节码指令看的到，**同步代码块**和**同步方法**的字节码是不同的

- 对于synchronized同步块，对应的monitorenter和monitorexit指令分别对应synchronized同步块的进入和退出。
    - **为什么会多一个monitorexit?** 编译器会为同步块添加一个隐式的try-finally，在finally中会调用monitorexit命令释放锁
- 对于synchronized方法，对应ACC_SYNCHRONIZED关键字，JVM进行方法调用时，发现调用的方法被ACC_SYNCHRONIZED修饰，则会先尝试获得锁，方法调用结束了释放锁。在JVM底层，对于这两种synchronized的实现大致相同。**都是基于monitorenter和monitorexit指令实现**，底层还是使用 标记字段MarkWord和**Monitor（管程）**来实现重量级锁。

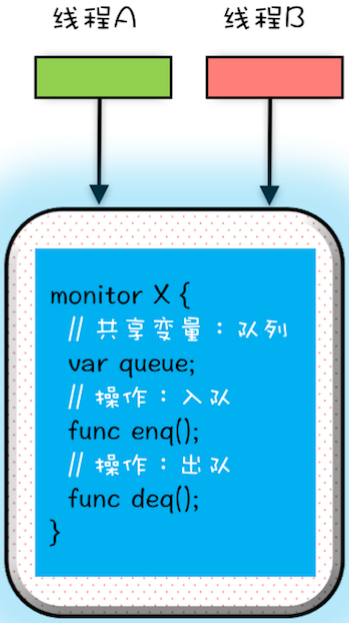
2.3 什么是Monitor?

- Monitor中文翻译为管程，也有人称之为“监视器”，管程指的是管理共享变量以及对共享变量的操作过程，让他们支持并发。
- Java中的所有对象都可以作为锁，每个对象都与一个 monitor 相关联，线程可以对 monitor 执行 lock 和 unlock 操作。
- Java并没有把lock和unlock操作直接开放给用户使用，但是却提供了两个指令来隐式地使用这两个操作：moniterenter和moniterexit。moniterenter对应lock操作，moniterexit对应unlock操作，通过这两个指锁定和解锁 monitor 对象来实现同步。
- 当一个monitor对象被线程持有后，它将处于锁定状态。对于一个 monitor 而言，同时只能有一个线程能锁定monitor，其它线程试图获得已被锁定的 monitor时，都将被阻塞。当monitor被释放后，阻塞中的线程会尝试获得该 monitor锁。一个线程可以对一个 monitor 反复执行 lock 操作，对应的释放锁时，需要执行相同次数的 unlock 操作。

> 详细了解Monitor请参考：极客时间专栏《Java并发编程实战-王宝令》管程：并发编程的万能钥匙
> 文章：https://time.geekbang.org/column/article/86089

Monitor如何解决了线程安全问题?

管程解决互斥问题的思路：就是将共享变量及其对共享变量的操作统一封装起来。



2.4 什么是锁优化?

- 在JDK 1.6之前，synchronized使用传统的锁（重量级锁）实现。它依赖于操作系统（互斥量）的同步机制，涉及到用户态和内核态的切换、线程的上下文切换，性能开销较高，所以给开发者留下了synchronized关键字性能不好的印象。
- 如果**只有一个线程运行时**并没有发生资源竞争、或**两个线程交替执行**，使用传统锁机制无疑效率是会比较低的。
- JDK1.6中为了减少这两个场景，获得锁和释放锁带来的性能消耗，同步锁进行优化引入：**偏向锁和轻量级锁。**
- 同步锁一共有四种状态，级别从低到高依次是：**无锁，偏向锁，轻量级锁，重量级锁。**这四种状态会随着竞争激烈情况逐渐升级。

偏向锁

偏向锁则是基这样一个想法：只有一个线程访问锁资源（无竞争）的话，偏向锁就会把整个同步措施都消除，并记录当前持有锁资源的线程和锁的类型。

轻量级锁

轻量级锁是基于这样一个想法：只有两个线程交替运行时，如果线程竞争锁失败了，先不立即挂起，而是让它飞一会儿（自旋），在等待过程中，可能锁就被释放了，这时该线程就可以重新尝试获取锁，同时记录持有锁资源的线程和锁的类型。

那锁信息存储在哪?

例如：锁类型，当前持有线程

偏向锁标记	锁状态标识	锁状态
0	01	无锁
1	01	偏向锁
无	00	轻量锁
无	10	重量锁
无	11	GC标记

同步锁锁定资源是对象，那无疑存储在对象信息中，由对象直接携带，是最方便管理和操作的。

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁状态	对象Hashcode、对象GC分代年龄				01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空，不需要记录信息				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01

> 32位操作系统的Markword

3. volatile

通过前面的内容，咱们了解了synchronized同步代码块，同步代码块在多线程场景下存在性能问题。接下来介绍一个轻量级的线程安全问题解决方案 **volatile** ，它比使用synchronized的成本更加低。

Java语言对volatile的定义：**Java允许线程访问共享变量，为了确保共享变量能被准确和一致地更新，线程应该确保通过排他锁单独获得这个变量。**

说人话：volatile可以保证多线程场景下变量的**可见性和有序性。**如果某变量用volatile修饰，则可以确保所有线程看到变量的值是一致的。

- 可见性：保证此变量的修改对所有线程的可见性。
- 有序性：禁止指令重排序优化，编译器和处理器在进行指令优化时，不能把在volatile变量操作(读/写)后面的语句放到其前面执行，也不能将volatile变量操作前面的语句放在其后执行。**遵循了JMM的happens-before规则**

> 注：volatile虽然看起来比较简单，无非就是在某个变量前加上volatile，但要用好并不容易！

3.1 解决内存可见性问题

在可见性案例中，做如下修改：

```
// 添加volatile关键词
private volatile boolean flag = true;
```

volatile实现内存可见性的过程

线程写volatile变量的过程：

1. 改变线程本地内存中volatile变量副本的值；
2. 将改变后的副本的值从本地内存刷新到主内存

线程读volatile变量的过程：

1. 从主内存中读取volatile变量的最新值到线程的本地内存中
2. 从本地内存中读取volatile变量的副本

3.2 volatile实现原理-源码分析

volatile实现内存可见性原理：内存屏障（Memory Barrier）

内存屏障（Memory Barrier）是一种**CPU指令**，用于控制特定条件下的重排序和内存可见性问题。Java编译器也会根据内存屏障的规则禁止重排序

- 写操作时，通过在写操作指令后加入一条store屏障指令，让本地内存中变量的值能够刷新到主内存中
- 读操作时，通过在读操作前加入一条load屏障指令，及时读取到变量在主内存的值

我们可以从源码角度，来理解volatile的可见性和有序性。

```
{
  public volatile boolean flag;
    descriptor: Z
    flags: ACC_PUBLIC, ACC_VOLATILE

  Demo05JmmVolatile$JmmDemo();
    descriptor: ()V
    flags:
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: iconst_1
         6: putfield      #2                  // Field flag:Z
         9: return
      LineNumberTable:
        line 14: 0
        line 15: 4

  public void run();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #4                  // String 子线程执行。。。
```

有volatile修饰的变量，通过javap可以看到volatile字节码有个关键字ACC_VOLATILE，通过这个关键字定位到hotspot JVM源码文件 `src\share\vm\utilities\accessFlags.hpp` 文件，代码如下：

```
 public:
  // Java access flags
  bool is_public      () const         { return (_flags & JVM_ACC_PUBLIC
) != 0; }
  bool is_private     () const         { return (_flags & JVM_ACC_PRIVATE
  ) != 0; }
  bool is_protected   () const         { return (_flags & JVM_ACC_PROTECTED
  ) != 0; }
  bool is_static      () const         { return (_flags & JVM_ACC_STATIC
) != 0; }
```

```
  bool is_final       () const         { return (_flags & JVM_ACC_FINAL
  ) != 0; }
  bool is_synchronized() const         { return (_flags &
JVM_ACC_SYNCHRONIZED) != 0; }
  bool is_super       () const         { return (_flags & JVM_ACC_SUPER
  ) != 0; }
  bool is_volatile    () const         { return (_flags & JVM_ACC_VOLATILE
) != 0; }
  bool is_transient   () const         { return (_flags & JVM_ACC_TRANSIENT
  ) != 0; }
  bool is_native      () const         { return (_flags & JVM_ACC_NATIVE
) != 0; }
  bool is_interface   () const         { return (_flags & JVM_ACC_INTERFACE
  ) != 0; }
  bool is_abstract    () const         { return (_flags & JVM_ACC_ABSTRACT
) != 0; }
  bool is_strict      () const         { return (_flags & JVM_ACC_STRICT
) != 0; }
```

可以看到

```
bool is_volatile () const { return (_flags & JVM_ACC_VOLATILE ) != 0; }
```

再根据关键字is_volatile搜索，在`src\share\vm\interpreter\bytecodeInterpreter.cpp`可以看到如下代码：

```
//
// Now store the result
//
int field_offset = cache->f2_as_index();
if (cache->is_volatile()) {
    if (tos_type == itos) {
        obj->release_int_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == atos) {
        VERIFY_OOP(STACK_OBJECT(-1));
        obj->release_obj_field_put(field_offset, STACK_OBJECT(-1));
        OrderAccess::release_store(&BYTE_MAP_BASE[(uintptr_t)obj >>
CardTableModRefBS::card_shift], 0);
    } else if (tos_type == btos) {
        obj->release_byte_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == ltos) {
        obj->release_long_field_put(field_offset, STACK_LONG(-1));
    } else if (tos_type == ctos) {
        obj->release_char_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == stos) {
        obj->release_short_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == ftos) {
        obj->release_float_field_put(field_offset, STACK_FLOAT(-1));
    } else {
        obj->release_double_field_put(field_offset, STACK_DOUBLE(-1));
    }
    OrderAccess::storeload();
}
```

在这段代码中，会先判断tos_type（volatile变量类型），后面有不同的基础类型的调用，比如int类型就调用release_int_field_put，byte就调用release_byte_field_put等等。

我们可以看到后面执行的语句是

```
OrderAccess::storeload();
```

可以在`src\share\vm\runtime\orderAccess.hpp`找到对应的实现方法：

```
static void storeload();
```

实际上这个方法的实现针对不同的CPU有不同的实现的，在`src/os_cpu`目录下可以看到不同的实现，以`src\os_cpu\linux_x86\vm\orderAccess_linux_x86.inline.hpp`为例，是这么实现的：

```
inline void OrderAccess::loadload()   { acquire(); }
inline void OrderAccess::storestore() { release(); }
inline void OrderAccess::loadstore()  { acquire(); }
inline void OrderAccess::storeload()  { fence(); }
```

fence()函数的实现：

```
inline void OrderAccess::fence() {
  if (os::is_MP()) {
    // always use locked addl since mfence is sometimes expensive
#ifdef AMD64
    __asm__ volatile ("lock; addl $0,0(%%)" : : : "cc", "memory");
#else
    __asm__ volatile ("lock; addl $0,0(%%esp)" : : : "cc", "memory");
#endif
  }
}
```

通过这面代码可以看到`lock; addl`，其实这个就是内存屏障。`lock; addl $0,0(%%esp)`作为cpu的一个内存屏障。

`addl $0,0(%%rsp)`表示：将数值0加到rsp寄存器中，而该寄存器指向栈顶的内存单元。加上一个0，rsp寄存器的数值依然不变。即这是一条无用的汇编指令。在此利用addl指令来配合lock指令，用作cpu的内存屏障。

Java编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。为保证在任意处理器平台下能得到正确的 volatile 内存操作语义，JMM 采取保守策略，下面是基于保守策略的 JMM 内存屏障插入策略：

- 在每个 volatile 写前，插入StoreStore 屏障。
- 在每个 volatile 写后，插入StoreLoad 屏障。
- 在每个 volatile 读后，插入LoadLoad 屏障。
- 在每个 volatile 读后，插入LoadStore 屏障。

结合JMM中四类内存屏障的作用，我们可以得出下面的结论：

屏障类型	示例	说明
LoadLoad	Load1;LoadLoad;Load2	确保装载动作Load1，先于 Load2及其后所有Load操作。 对于屏障前后的Store操作并无影响。
StoreStore	Store1;StoreStore;Store2	确保Store1刷新数据到内存(使数据对其他处理器可见)的操作，先于Store2及其后所有Store指令的执行。 对于屏障前后的Load操作并无影响。
LoadStore	Load1;LoadStore;Store2	确保屏障指令之前的所有Load操作，先于屏障之后所有Store操作(刷新数据到主存)。
StoreLoad	Store1;StoreLoad;Load2	确保屏障之前的所有内存访问操作(包括Store和Load)完成之后，才执行屏障之后的内存访问操作。 全能型屏障，会屏蔽屏障前后所有指令的重排。

重排序规则表：

是否允许重排序	第二个操作		
第一个操作	普通读/写	Volatile读	Volatile写
普通读	Yes	Yes	Yes
普通写	Yes	Yes	No
Volatile读	No	No	No
Volatile写	No	No	No

1. 当第一个操作是**volatile读**时，不管第二个操作是什么，都不能重排序
    - 确保 volatile 读到的是最新值：volatile 读之后的操作不会被编译器重排序到 volatile 读之前
2. 当第一个操作是**volatile写**时，不管第二个操作是什么，都不能重排序
    - 确保 volatile 写操作对之后的操作可见
3. 当第二个操作是**volatile写**时，第一个操作是普通写时，不能重排序

3.3 volatile缺陷：原子性Bug

原子性的问题：虽然volatile可以保证可见性，但是不能满足原子性

```
package com.hero.multithreading;

public class Demo06Volatile {
    public static void main(String[] args) throws InterruptedException {
        VolatileDemo demo = new VolatileDemo();

        for (int i = 0; i < 2; i++) {
```

```
            Thread t = new Thread(demo);
            t.start();
        }

        Thread.sleep(1000);
        System.out.println("count = "+demo.count);
    }

    static class VolatileDemo implements Runnable {
        public volatile int count;
        //public volatile AtomicInteger count = new AtomicInteger(0);

        public void run() {
            addCount();
        }

        public void addCount() {
            for (int i = 0; i < 10000; i++) {
                count++;//但是实际情况是三条汇编指令
            }
        }
    }
}

```

结果：count = 12205

不应该是20000吗？问题分析：

以上出现原子性问题的原因是count++并不是原子性操作。

count = 5 开始，流程分析：

1. 线程1读取count的值为5
2. 线程2读取count的值为5
3. 线程2加1操作
4. 线程2最新count的值为6
5. 线程2写入值到主内存的最新值为6
6. 线程1执行加1count=6，写入到主内存的值是6。
7. 结果：对count进行了两次加1操作，主内存实际上只是加1一次。结果为6

解决方案：

1. 使用synchronized
2. 使用ReentrantLock（可重入锁）
3. 使用AtomicInteger（原子操作）

使用synchronized

```
public synchronized void addCount() {
    for (int i = 0; i < 10000; i++) {
        count++;
    }
}
```

使用ReentrantLock（可重入锁）

```
//可重入锁
private Lock lock = new ReentrantLock();

public void addCount() {
    for (int i = 0; i < 10000; i++) {
        lock.lock();
        count++;
        lock.unlock();
    }
}
```

使用AtomicInteger（原子操作）

```
public static AtomicInteger count = new AtomicInteger(0);
public void addCount() {
    for (int i = 0; i < 10000; i++) {
        //count++;
        count.incrementAndGet();
    }
}
```

3.4 volatile适合使用场景

变量真正独立于其他变量和自己以前的值，在单独使用的时候，适合用volatile

- 对变量的写入操作不依赖其当前值：例如++和--运算符的场景则不行
- 该变量没有包含在具有其他变量的不变式中

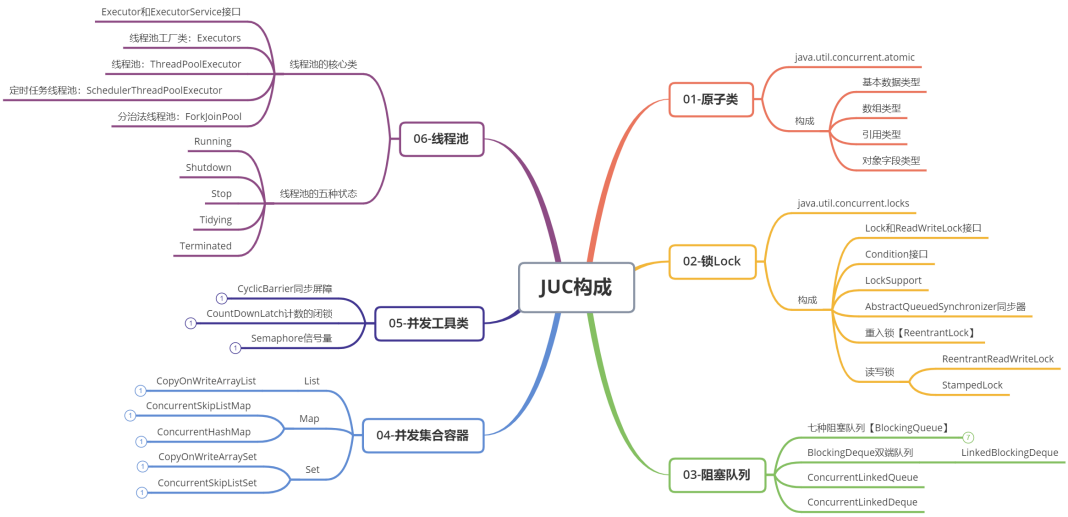
3.5 synchronized和volatile比较

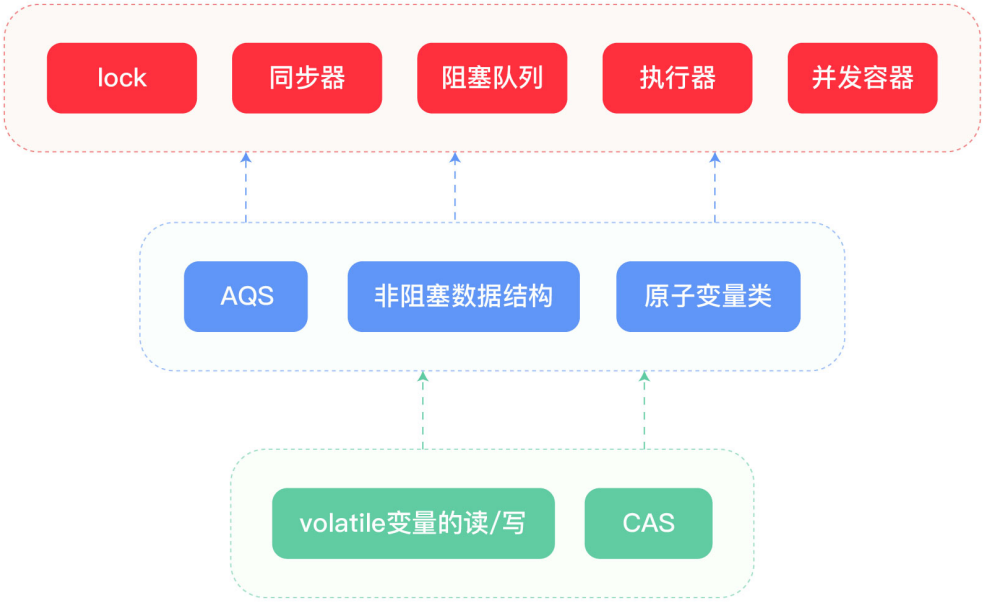
- volatile不需要加锁，比synchronized更轻便，不会阻塞线程
- synchronized既能保证可见性，又能保证原子性，而volatile只能保证可见性，无法保证原子性
- 与synchronized相比volatile是一种非常简单的同步机制

二、并发编程

1. JUC简介

从JDK1.5起，Java API 中提供了java.util.concurrent（简称JUC）包，在此包中定义了并发编程中很常用的工具，比如：线程池、阻塞队列、同步器、原子类等等。JUC是 JSR 166 标准规范的一个实现，JSR 166 以及 JUC 包的作者是同一个人 Doug Lea 。





2. 原子类与CAS

通过上面学习volatile，我们发现volatile修饰的变量存在原子性的BUG，这个问题怎么解决呢？难道只能使用Synchronized吗?

2.1 Atomic包

java.util.concurrent.atomic包

从JDK 1.5开始提供了java.util.concurrent.atomic包（以下简称Atomic包），这个包中的原子操作类提供了一种**用法简单、性能高效、线程安全地更新一个变量的方式。**可以解决volatile原子性操作变量的问题。

因为变量的类型有很多种，所以在Atomic包里一共提供了13个类，属于4种类型的原子更新方式，分别是原子更新基本类型、原子更新数组、原子更新引用和原子更新属性（字段）。**Atomic包里的类基本都是使用Unsafe实现的包装类。**

Atomic里的类主要包括：

- **基本类型-使用原子的方式更新基本类型**
  - AtomicInteger：整形原子类
  - AtomicLong：长整型原子类
  - AtomicBoolean ： 布尔型原子类
- **引用类型**
  - AtomicReference：引用类型原子类
  - AtomicStampedReference：原子更新引用类型里的字段原子类
  - AtomicMarkableReference ： 原子更新带有标记位的引用类型
- **数组类型-使用原子的方式更新数组里的某个元素**
  - AtomicIntegerArray：整形数组原子类
  - AtomicLongArray：长整形数组原子类
  - AtomicReferenceArray ： 引用类型数组原子类
- **对象的属性修改类型**
  - AtomicIntegerFieldUpdater:原子更新整形字段的更新器
  - AtomicLongFieldUpdater：原子更新长整形字段的更新器
  - AtomicReferenceFieldUpdater ： 原子更新引用类形字段的更新器
- JDK1.8新增类
  - DoubleAdder：双浮点型原子类
  - LongAdder：长整型原子类
  - DoubleAccumulator：类似DoubleAdder，但要更加灵活(要传入一个函数式接口)
  - LongAccumulator：类似LongAdder，但要更加灵活(要传入一个函数式接口)

AtomicInteger主要API如下：

```
get()                //直接返回值
getAndAdd(int)       //增加指定的数据，返回变化前的数据
getAndDecrement()    //减少1，返回减少前的数据
getAndIncrement()    //增加1，返回增加前的数据
getAndSet(int)       //设置指定的数据，返回设置前的数据

addAndGet(int)       //增加指定的数据后返回增加后的数据
decrementAndGet()    //减少1，返回减少后的值
incrementAndGet()    //增加1，返回增加后的值
lazySet(int)         //仅仅当get时才会set

compareAndSet(int, int)//尝试新增后对比，若增加成功则返回true否则返回false
```

用AtomicInteger解决可见性案例中的问题！

```
package com.hero.multithreading;

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

```
4
5  public class Demo07Volatile {
6      public static void main(String[] args) throws InterruptedException {
7          VolatileDemo demo = new VolatileDemo();
8
9          for (int i = 0; i < 2; i++) {
10             Thread t = new Thread(demo);
11             t.start();
12         }
13
14         Thread.sleep(1000);
15         System.out.println("count = "+demo.count);
16     }
17
18     static class VolatileDemo implements Runnable {
19         public AtomicInteger count = new AtomicInteger(0);
20
21         public void run() {
22             addCount();
23         }
24
25         public void addCount() {
26             for (int i = 0; i < 10000; i++) {
27                 count.incrementAndGet();
28             }
29         }
30     }
31 }
```

虽然涉及到的类很多，但是原理都是差不多，也都是使用CAS进行的原子操作。接下来，我们看一下CAS的原理。

2.2 CAS介绍

CAS是什么?

CAS即compare and swap（比较再替换）

同步组件中大量使用CAS技术实现了Java多线程的并发操作。整个AQS同步组件、Atomic原子类操作等等都是以CAS实现的，甚至ConcurrentHashMap在1.8的版本中也调整为了CAS+Synchronized。可以说CAS是整个JUC的基石。

CAS并不难以理解，**本质上是一个方法调用了一行CPU原子指令。**

```
1  执行函数：CAS(V,E,N)
```

CAS操作涉及到三个操作数：

- V：要读写的内存地址
- E：进行比较的值 （预期值）
- N：拟写入的新值

当且仅当**内存地址V**中的值等于 **预期值E** 时，将内存V中的值改为N，否则会进行自旋操作（一般情况下），即不断的重试。

CAS本质是一条CPU的原子指令，可以保证共享变量修改的原子性。

2.3 CAS原理详解

Java中对CAS的实现

Java不能像C/C++那样直接操作内存区域，需要通过本地方法(native 方法)来访问。JAVA中的CAS操作都是通过sun包下Unsafe类实现，而Unsafe类中的方法都是native方法。

Unsafe类，全限定名是sun.misc.Unsafe，位于在 sun.misc 包下，不属于Java 标准API。

Unsafe对CAS操作的实现有三个

```
/**
 * Atomically update Java variable to x if it is currently holding expected.
 * @return true if successful
 */
public final native boolean compareAndSwapInt(Object o, long offset, int expected, int x);
/**
 * Atomically update Java variable to x if it is currently holding expected.
 * @return true if successful
 */
public final native boolean compareAndSwapObject(Object o, long offset,Object expected,Object x);
/**
 * Atomically update Java variable to x if it is currently holding expected.
 * @return true if successful
 */
public final native boolean compareAndSwapLong(Object o, long offset, long expected, long x);
```

我们以其中的compareAndSwapInt为例，来说明

compareAndSwapInt

- 方法作用：如果当前时刻，**待更新的原值** 与 预期值 expected相等，则将 **待更新的原值** 的 值更新为x。如果更新成功，则返回 true，否则返回 false。
- compareAndSwapInt是 Unsafe 类中提供的一个原子操作。方法一共有四个参数：
  - o：需要改变的对象
  - offset：内存偏移量，offset 为o对象所属类中，某个属性在类中的内存地址偏移量
  - expected ： 预期值
  - x：拟替换的新值

内存偏移量offset的作用是什么?

- 计算出对象中，待更新的原值的准确内存地址
- Java对象在内存中会占用一段内存区域，Java对象的属性会按照一定的顺序在对象内存中存储。根据对象this就可以定位到this对象在内存的起始地址，然后在根据属性state(相对this)的offset内存偏移量，就可以精确的定位到state的内存地址，从而得到当前时刻state在内存中的值。

```
public final native boolean compareAndSwapInt(Object o, long offset, int expected, int x);
```

为了弄清楚CAS原子性的实现原理，查看openJDK8的源码，在`hotspot\src\share\vm\prims\unsafe.cpp`中可以找到compareAndSwapInt的实现（行1213）

```
/*
*jobject obj：java对象【AtomicInteger】
*jlong offset：内存偏移量
*jint e：预期值
*jint x：拟替换的新值
*/
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);//将Java对象解析成JVM的oop（普通对象指针）
  //根据对象p内存地址和内存地址偏移量计算拟修改对象属性的地址
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  //基于cas比较并替换，x表示拟更新的值， addr表示要操作的内存地址， e表示预期值
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

从上面的代码，我们很明显可以看出，Unsafe_CompareAndSwapInt 函数实现的关键在于最后一句，使用了 `cmpxchg` 指令，进一步，我们来看看 Windows 平台下 Atomic::cmpxchg 函数：

可以在在`hotspot\src\os_cpu\windows_x86\vm\atomic_windows_x86.inline.hpp`找到

```
//atomic_windows_x86.inline.hpp 66行.LOCK_IF_MP预编译机器码
/*
*
注意：_emit 0xF0中的_emit 并不是真正的指令，可以把它理解为伪指令,它的作用很简单,就是把操作数直接写入到二进制文件里,正常写汇编程序,比如mov eax,1这条，编译器会把这条指令翻译成机器码,然后写入二进制文件，而_emit 0xF0这条指令，编译器直接就会把0xF0写入二进制文件，0xF0是什么呢?其实是lock前缀.
*/
#define LOCK_IF_MP(mp) __asm cmp mp, 0  \
                       __asm je L0      \
                       __asm _emit 0xF0 \//相当于lock前缀指令。
                       __asm L0:
/*
 * jint exchange_value：拟替换的新值
 * jint* dest：内存地址
 * jint compare_value：预期值
 */
//216行
inline jint Atomic::cmpxchg (jint exchange_value, volatile jint* dest, jint compare_value) {
  // alternative for InterlockedCompareExchange
  int mp = os::is_MP();
  __asm {
    mov edx, dest
    mov ecx, exchange_value
    mov eax, compare_value
    LOCK_IF_MP(mp)    /*
     * cmpxchg比较并交换。这是一个汇编指令:
     *   cmpxchg: 即“比较并交换”指令
```

```
     *   dword: 全称是 double word
     *          在 x86/x64 体系中，一个word = 2 byte，dword = 4 byte = 32 bit，
dword = 8 byte = 64 bit
     *   ptr: 全称是 pointer，与前面的 dword 连起来使用，表明访问的内存单元是一个双字
单元
     *   [edx]: [...] 表示一个内存单元，edx 是寄存器，dest 指针值存放在 edx 中。那么
[edx] 表示内存地址为 dest 的内存单元
     *
     * 这一条指令的意思就是，将 eax 寄存器中的值（compare_value）与 [edx] 双字内存单
元中的值
     * 进行对比，如果相同，则将 ecx 寄存器中的值（exchange_value）存入 [edx] 内存单元
中。
     */
    cmpxchg dword ptr [edx], ecx
  }
}
```

2.4 CAS缺陷

CAS虽然高效地解决了原子操作，但是还是存在一些缺陷的，主要表现在三个地方：**循环时间太长、只能保证一个共享变量原子操作、ABA问题。**

- **循环时间太长**：如果CAS一直不成功呢？如果自旋CAS长时间地不成功，则会给CPU带来非常大的开销。
    - 原子类AtomicInteger#getAndIncrement()的方法
- **只能保证一个共享变量原子操作**：看了CAS的实现就知道这只能针对一个共享变量，如果是多个共享变量就只能使用锁了。
- **ABA问题**：CAS需要检查操作值有没有发生改变，如果没有发生改变则更新。但是存在这样一种情况：如果一个值原来是A，变成了B，然后又变成了A，那么在CAS检查的时候会发现没有改变，但是实质上它已经发生了改变，这就是所谓的ABA问题。对于ABA问题其解决方案是加上版本号，即在每个变量绑定一个版本号，每次改变时加1，即A —> B —> A，变成1A —> 2B —> 3A。

下面我们将通过一个例子可以可以看到AtomicStampedReference和AtomicInteger的区别。我们定义两个线程，线程1负责将100 —> 101 —> 100，线程2执行 100 —>2022，看两者之间的区别。

```
package com.hero.multithreading;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicStampedReference;

public class Demo08ABA {

    static AtomicInteger ar = new AtomicInteger(100);
    static AtomicStampedReference<Integer> asr = new
AtomicStampedReference<>(100, 1);

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

```
        System.out.println("======ABA问题的产生======");

        Thread t1 = new Thread(() -> {
            ar.compareAndSet(100, 101);
            ar.compareAndSet(101, 100);
        }, "t1");
        t1.start();

        Thread t2 = new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();

            }
            System.out.println(ar.compareAndSet(100, 2022) + "\t" +
ar.get());
        }, "t2");
        t2.start();

        //顺序执行，AtomicInteger案例先执行
        t1.join();
        t2.join();

        System.out.println("======ABA问题的解决======");
        new Thread(() -> {
            int stamp = asr.getStamp();
            System.out.println(Thread.currentThread().getName() + "\t第一次版
本号：  " + stamp);

            try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException
e) { e.printStackTrace(); }

            asr.compareAndSet(100,101, asr.getStamp(), asr.getStamp()+1);
            System.out.println(Thread.currentThread().getName() + "\t第二次版
本号：  " + asr.getStamp());

            asr.compareAndSet(101,100, asr.getStamp(), asr.getStamp()+1);
            System.out.println(Thread.currentThread().getName() + "\t第三次版
本号：  " + asr.getStamp());
        }, "t3").start();

        new Thread(() -> {

            int stamp = asr.getStamp();
            System.out.println(Thread.currentThread().getName() + "\t第一次版
本号：  " + stamp);

            try { TimeUnit.SECONDS.sleep(3); } catch (InterruptedException
e) { e.printStackTrace(); }

            boolean result= asr.compareAndSet(100,2022, stamp,stamp+1);
            System.out.println(Thread.currentThread().getName()+"\t修改成功与
否："+result+"  当前最新版本号"+ asr.getStamp());
```

```
59              System.out.println(Thread.currentThread().getName()+"\t当前实际
值："+ asr.getReference());
60          }, "t4").start();
61      }
62  }
63
```

运行结果充分展示了AtomicInteger的ABA问题和AtomicStampedReference解决ABA问题。

```
C:\develop\java\jdk1.8.0_171\bin\java.exe ...
======ABA问题的产生======
true	2022
======ABA问题的解决======
t3	第一次版本号：1
t4	第一次版本号：1
t3	第二次版本号：2
t3	第三次版本号：3
t4	修改成功与否：false	当前最新版本号3
t4	当前实际值：100
```

3. Lock锁与AQS

3.1 Java锁简介

Java提供了种类丰富的锁，每种锁因其特性的不同，在适当的场景下能够展现出非常高的效率。

JUC包中提供的锁：

- ReentrantLock重入锁，它是一种可重入的独享锁，具有与使用 synchronized 相同的一些基本行为和语义，但是它的API功能更强大，ReentrantLock 相当于synchronized 的增强版，具有synchronized很多所没有的功能。
- ReentrantReadWriteLock读写锁
    - synchronized和ReentrantLock都是同步互斥锁，不管是读操作的线程还是写操作的线程，同时只能有一个线程获得锁，也就是在进行写操作的时候，在写线程进行访问的时候，所有的线程都会被阻塞。但是其实，读操作是不需要加锁访问的。互斥锁不区分读写，全部加锁实现起来简单，但是性能会大打折扣。
    - ReentrantReadWriteLock维护了一对关联锁：ReadLock和WriteLock，由词知意，一个读锁一个写锁，合称“读写锁”。一个是ReadLock(读锁)用于读操作的，一个是WriteLock(写锁)用于写操作，这两个锁都实现了Lock接口。**读写锁适合于读多写少的场景，基本原则是读锁可以被多个线程同时持有进行访问，而写锁只能被一个线程持有。**
- **StampedLock重入读写锁**，JDK1.8引入的锁类型，是对读写锁ReentrantReadWriteLock的增强版

在Java中往往会按照是否含有某一特性来定义锁，下面我们按照锁的特性将锁进行分组归类，帮助大家更系统的理解Java中的锁。

锁的分类：按上锁方式划分

- **隐式锁：synchronized**

    - synchronized为Java的关键字，是Java提供的同步机制，当它用来修饰一个方法或一个代码块时，能够保证在同一时刻最多只能有一个线程执行该代码。当使用synchronized修饰代码时，并不需要显式的执行加锁和解锁过程，所以它也被称之为隐式锁。
- **显式锁：JUC包中提供的锁**
    - JUC中提供的锁都提供了常用的锁操作，加锁和解锁的方法都是显式的，我们称他们为显式锁。

锁的分类：按特性划分

- **乐观锁/悲观锁**：按照线程在使用共享资源时，**要不要锁住同步资源**，划分为：乐观锁和悲观锁。
    - 悲观锁：比较悲观，总是假设最坏的情况，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。
        - 实现：JUC的锁、Synchronized
    - 乐观锁：比较乐观，总是假设最好的情况，对于同一个数据的并发操作，**乐观锁认为自己在使用数据时不会有别的线程修改数据，所以在获取数据的时候不会添加锁。**只有在更新数据的时候才会去判断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，则会根据不同的情况执行不同的操作（例如：报错或自动重试）
        - 实现：CAS算法，关系型数据库的版本号机制
- **可重入锁/不可重入锁**：按照同一个线程是否可以重复获取同一把锁，划分为：可重入锁和不可重入锁。
    - 重入锁：一个线程可以重复获取同一把锁，不会因为之前已经获取了该锁未释放而被阻塞。在获得一个锁之后未释放锁之前，再次获得同一把锁时，只会增加获得锁的次数，当释放锁时，会同时减少锁定次数。可重入锁的一个优点是可一定程度避免死锁。
        - 实现：ReentrantLock、synchronized
    - 非重入锁：不可重入锁，与可重入锁相反，同一线程获得锁之后不可再次获取，重复获取会发生死锁。
- **公平锁/非公平锁**：按照多个线程竞争同一锁时需不需要排队，能不能插队，划分为公平锁和非公平锁。
    - 公平锁：多个线程按照申请锁的顺序来获得锁
        - 实现：new ReentrantLock(true)
    - 非公平锁：多个线程获取锁的顺序并不是按照申请锁的顺序，允许“插队”，有可能后申请的线程比先申请的线程优先获取锁
        - 实现：new ReentrantLock(false)，synchronized
- **独享锁/共享锁**：按照多个线程能不能同时共享同一个锁，锁被划分为独享锁和排他锁。
    - 独享锁（写锁）：独享锁也叫排他锁，是指同一个锁同时只能被一个线程所持有。如果线程A对获得了锁S后，则其他线程只能阻塞等待线程A释放锁S后，才能获得锁S。
        - 实现：synchronized，ReentrantLock
    - 共享锁（读锁）：同一个锁可被多个线程同时持有。如果线程A对获得了共享锁S后，则其他线程无需等待可以获得共享锁S。
        - 实现：ReentrantReadWriteLock的读锁。
    - 在ReentrantReadWriteLock维护了一对关联锁：ReadLock和WriteLock，由词知意，一个读锁一个写锁，合称“读写锁”。ReadLock(读锁)用于读操作的，WriteLock(写锁)用于写操作，读锁是共享锁，写锁是独享锁，读锁可保证在读多写少的场景中，提高并发读的性能，增加程序的吞吐量。

锁的分类：其他常见的锁

- **自旋锁**：获取锁失败时，线程不会阻塞而是循环尝试获得锁，直至获得锁成功。
  - 实现：CAS，举例：AtomicInteger#getAndIncrement()
- **分段锁**：在并发程序中，使用独占锁时保护共享资源的时候，基本上是采用串行方式，每次只能有一个线程能访问它。串行操作是会降低可伸缩性，在某些情况下我们可以将锁按照某种机制分解为一组独立对象上的锁，这成为分段锁。
  - 说的简单一点：容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率。ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。
  - 实现：ConcurrentHashMap
- **无锁/偏向锁/轻量级锁/重量级锁**：
  - 这四个锁是synchronized独有的四种状态，级别从低到高依次是：无锁、偏向锁、轻量级锁和重量级锁。它们是JVM为了提高synchronized锁的获取与释放效率而做的优化。四种状态会随着竞争的情况逐渐升级，而且是不可逆的过程，即不可降级。

用ReentrantLock解决可见性案例中的问题！

```
package com.hero.multithreading;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicStampedReference;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Demo09ReentrantLock {

    public static void main(String[] args) throws InterruptedException {
        VolatileDemo demo = new VolatileDemo();

        for (int i = 0; i < 2; i++) {
            Thread t = new Thread(demo);
            t.start();
        }

        Thread.sleep(1000);
        System.out.println("count = "+demo.count);
    }

    static class VolatileDemo implements Runnable {
        public  int count = 0;
        public Lock lock = new ReentrantLock();

        public void run() {
            addCount();
        }

        public void addCount() {
```

```
            lock.lock();
            for (int i = 0; i < 10000; i++) {
                count++;
            }
            lock.unlock();
        }
    }
}
```

3.2 synchronized和JUC的锁对比

Java已经提供了synchronized，为什么还要使用JUC的锁呢？重复造轮子？

synchronized同步锁提供了一种排他式的同步机制，当多个线程竞争锁资源时，同时只能有一个线程持有锁，当一个线程获取了锁，其他线程就会被阻塞只有等到占有锁的线程释放锁后，才能重新进行锁竞争。

使用synchronized同步锁，线程会三种情况下释放锁：

1. 线程执行完了同步代码块/方法，释放锁；
2. 线程执行时发生异常，此时JVM会让线程自动释放锁；
3. 在同步代码块/方法中，锁对象执行了wait方法，线程释放锁。

从以上synchronized的特点，我们可以总结出两个不足之处：

第一：**synchronized同步锁的线程阻塞，存在有两个致命的缺陷：无法控制阻塞时长；阻塞不可中断。**

- 使用synchronized同步锁，假如占有锁的线程被长时间阻塞（IO阻塞，sleep方法，join方法等），由于线程在阻塞时没有释放锁，如果其他线程尝试获取锁，就会被阻塞只能一直等待下去，甚至会发生死锁，这样就会造成大量线程的堆积，严重的影响服务器的性能。
- JUC的锁可以解决这两个缺陷：
    - tryLock(long time, TimeUnit unit)
    - lockInterruptibly()

第二：读多写少的场景中，当多个读线程同时操作共享资源时，读操作和读操作不会对共享资源进行修改，所以读线程和读线程是不需要同步的。如果这时采用synchronized关键字，就会导致一个问题，当多个线程都只是进行读操作时，所有线程都只能同步进行，只能有一个读线程可以进行读操作，其他读线程只能等待锁的释放而无法进行读操作。

- **在上述场景中，我们需要实现一种机制，当多个线程都都只是进行读操作时，使得线程可以同时进行读操作（共享锁）。synchronized同步锁，不支持这种操作。**
- JUC的ReentrantReadWriteLock锁可以解决以上问题

总结

01-线程异步变同步Synchronized

Synchronized解决内存可见性问题

- 线程解锁前：必须把自己本地内存中共享变量的最新值刷新到主内存中
- 线程加锁时：将清空本地内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值

Synchronized同步实现原理：同步操作主要是monitorenter和monitorexit两个指令实现，背后原理是Monitor（管程）

管程(Monitor)：是**管**理共享变量及对共享变量操作的过**程**，**将共享变量和对共享变量的操作统一封装起来**，让这个过程可以并发执行。

为什么所有对象都可以作为锁？因为每个对象都都有一个Monitor对象与之关联。然后线程对monitor执行lock和unlock操作，相当于对对象执行上锁和解锁操作。

锁优化：同步锁一共有四个状态：无锁，偏向锁，轻量级锁，重量级锁，JVM会视情况来逐渐升级锁，而不是上来就加重量级锁

- **偏向锁：只有一个线程访问锁资源**，偏向锁就会把整个同步措施消除
- **轻量级锁：只有两个线程交替竞争锁资源**，如果线程竞争锁失败了不立即挂起，而是让它飞一会（自旋），在等待过程中可能锁就会被释放出来，这时尝试重新获取锁

02-volatile关键字

volatile可以保证多线程场景下共享变量的可见性、有序性。

- **可见性：**保证对此共享变量的修改，所有线程的可见性
- **有序性：**禁止指令重排序的优化，遵循JMM的happens-before规则

内存屏障（Memory Barrier）是CPU的一种指令，用于控制特定条件下的重排序和内存可见性问题。Java编译器会根据内存屏障的规则禁止重排序。

volatile缺陷：原子性问题

适合场景：变量真正独立于其他变量和自己以前的值，在单独使用的时适合用volatile

- 对变量的写入操作不依赖其当前值：例如++和--运算符的场景则不行
- 该变量没有包含在具有其他变量的不变式中