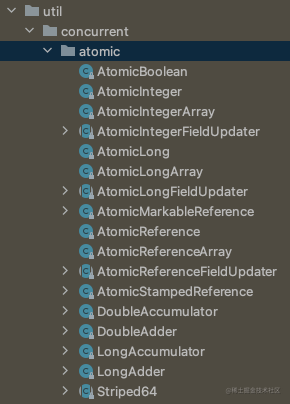
从JDK1.5开始，Java在java.util.concurrent.atomic包下引入了一些Atomic相关的原子操作类，这些类避免使用加锁来实现同步，从而更加方便、高效的实现原子操作。atomic包下的所有类如下图所示：



Atomic包下所有的原子类都只适用于单个元素，即只能保证一个基本数据类型、对象、或者数组的原子性。根据使用范围，可以将这些类分为四种类型，分别为原子更新基本类型、原子更新数组、原子更新引用、原子更新属性。

1. 原子更新基本类型

atomic包下原子更新基本数据类型包括AtomicInteger、AtomicLong、AtomicBoolean三个类，分别提供了原子更新整数类型、原子更新长整数类型和原子更新布尔类型的功能。这里，我们以AtomicInteger为例来学习如何使用。

AtomicInteger中提供了很多方法供我们调用，如：

// 获取当前值，然后自加，相当于i++

getAndIncrement()

// 获取当前值，然后自减，相当于i--

getAndDecrement()

// 自加1后并返回，相当于++i

incrementAndGet()

// 自减1后并返回，相当于--i

decrementAndGet()

// 获取当前值，并加上预期值

getAndAdd(int delta)

// 获取当前值，并设置新值

int getAndSet(int newValue)

// ...

需要注意的是这些方法都是原子操作，在多线程下也能够保证原子性以incrementAndGet方法为例：

AtomicInteger atomicInteger = new AtomicInteger();

private int index;

public void increase() throws InterruptedException {

new Thread(() -> {

for (int i = 0; i < 10000; i++) {

index = atomicInteger.incrementAndGet();

}

}).start();

new Thread(() -> {

for (int i = 0; i < 10000; i++) {

index = atomicInteger.incrementAndGet();

}

}).start();

Thread.sleep(1000);

System.out.println("-----" + index); // 输出结果20000

}

在increase方法中开启了两个线程并使用AtomicInteger对index进行自增操作，每次的输出结果都为20000.

1. 原子更新引用类型

基本类型的原子类只能更新一个变量，如果需要原子更新多个变量，则需要使用引用类型原子类。引用类型的原子类包括AtomicReference、AtomicStampedReference、AtomicMarkableReference三个。

AtomicReference 引用原子类

AtomicStampedReference 原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。（关于CAS及ABA问题后文详细分析）

AtomicMarkableReference 原子更新带有标记的引用类型。该类将 boolean 标记与引用关联起来。

接下来以AtomicReference为例来分析，首先看下AtomicReference的类结构：

public class AtomicReference<V> implements java.io.Serializable {

public final V get() {

return value;

}

public final void set(V newValue) {

value = newValue;

}

public final boolean compareAndSet(V expectedValue, V newValue) {

return VALUE.compareAndSet(this, expectedValue, newValue);

}

// ...省略其他

}

可以看到AtomicReference是一个泛型类，内部设置及更新引用类型数据的方法。以compareAndSet方法为例来看如何使用。

public class Book {

public String name;

public int price;

public Book(String name, int price) {

this.name = name;

this.price = price;

}

}

AtomicReference<Book> atomicReference = new AtomicReference<>();

Book book1 = new Book("三国演义", 42);

atomicReference.set(book1);

Book book2 = new Book("水浒传", 40);

atomicReference.compareAndSet(book1, book2);

System.out.println("Book name is " + atomicReference.get().name + ",价格是" + atomicReference.get().price);

输出结果为：

Book name is 水浒传,价格是40

上述代码首先将book1关联AtomicReference，接着又实例化了book2。调用compareAndSet方法传入book1和book2两个参数，通过CAS更新book。首先判断期望的是不是book1，如果是则更新为book2.否则继续自旋知道更新成功。

3.原子更新数组

这里原子更新数组并不是对数组本身的原子操作，而是对数组中的元素。主要包括3个类：AtomicIntegerArray、AtomicLongArray及AtomicReferenceArray，分别表示原子更新整数数组的元素、原子更新长整数数组的元素以及原子更新引用类型数组的元素。我们以AtomicIntegerArray为例来看：

public class AtomicIntegerArray implements java.io.Serializable {

// final类型的int数组

private final int[] array;

// 获取数组中第i个元素

public final int get(int i) {

return (int)AA.getVolatile(array, i);

}

// 设置数组中第i个元素

public final void set(int i, int newValue) {

AA.setVolatile(array, i, newValue);

}

// CAS更改第i个元素

public final boolean compareAndSet(int i, int expectedValue, int newValue) {

return AA.compareAndSet(array, i, expectedValue, newValue);

}

// 获取第i个元素，并加1

public final int getAndIncrement(int i) {

return (int)AA.getAndAdd(array, i, 1);

}

// 获取第i个元素并减1

public final int getAndDecrement(int i) {

return (int)AA.getAndAdd(array, i, -1);

}

// 对数组第i个元素加1后再获取

public final int incrementAndGet(int i) {

return (int)AA.getAndAdd(array, i, 1) + 1;

}

// 对数组第i个元素减1后再获取

public final int decrementAndGet(int i) {

return (int)AA.getAndAdd(array, i, -1) - 1;

}

// ... 省略

}

可以看到，在AtomicIntegerArray内部维护了一个final修饰的int数组，且类中所有的操作都是针对数组元素的操作。同时，这些方法都是原子操作，可以保证多线程下数据的安全性。

1. 原子更新对象属性

如果直选哟更新某个对象中的某个字段，可以使用更新对象字段的原子类。包括三个类，AtomicIntegerFieldUpdater、AtomicLongFieldUpdater以及AtomicReferenceFieldUpdater。需要注意的是这些类的使用需要满足以下条件才可。

被操作的字段不能是static类型；

被操纵的字段不能是final类型；

被操作的字段必须是volatile修饰的；

属性必须对于当前的Updater所在区域是可见的。

下面以AtomicIntegerFieldUpdater为例，结合前例中的Book类来更新Book的价格，注意将price用volatile修饰。

public class Book {

public String name;

public volatile int price;

public Book(String name, int price) {

this.name = name;

this.price = price;

}

}

AtomicIntegerFieldUpdater<Book> updater = AtomicIntegerFieldUpdater.newUpdater(Book.class, "price");

Book book = new Book("三国演义", 42);

updater.set(book, 50);

System.out.println( "更新后的价格是" + updater.get(book));

实例化一个Book，价格为42，通过AtomicIntegerFieldUpdater可以将价格修改为50。