共享模型之无锁

0. 概述

设计一个转账的类，此时余额属于是临界区，因此按照以往学到的知识，需要对那些操作余额的代码加锁，也就是如下代码

```
package com.ecifics.concurrent.part6;

/**
 * @author Ecifics
 * @Description TODO
 * @date 4/25/2022-3:28 PM
 */
```

```
import lombok.extern.slf4j.Slf4j;

import java.util.ArrayList;
import java.util.List;

/**
 * Description: 使用重量级锁synchronized来保证多线程访问共享资源发生的安全问题
 *
 * @author guizy
 * @date 2020/12/27 16:23
 */
@Slf4j(topic = "guizy.Test1")
public class TestAccount {

    public static void main(String[] args) {
        Account account = new AccountUnsafe(10000);
        Account.demo(account);
    }
}

class AccountUnsafe implements Account {
    private Integer balance;

    public AccountUnsafe(Integer balance) {
        this.balance = balance;
    }

    @Override
    public Integer getBalance() {
        synchronized (this) {
            return balance;
        }
    }

    @Override
    public void withdraw(Integer amount) {
        // 通过这里加锁就可以实现线程安全，不加就会导致线程安全问题
        synchronized (this) {
            balance -= amount;
        }
    }
}
```

```
interface Account {
    // 获取余额
    Integer getBalance();

    // 取款
    void withdraw(Integer amount);

    /**
     * Java8之后接口新特性，可以添加默认方法
     * 方法内会启动 1000 个线程，每个线程做 -10 元 的操作
     * 如果初始余额为 10000 那么正确的结果应当是 0
     */
    static void demo(Account account) {
        List<Thread> ts = new ArrayList<>();
        long start = System.nanoTime();
        for (int i = 0; i < 1000; i++) {
            ts.add(new Thread(() -> {
                account.withdraw(10);
            }));
        }
        ts.forEach(thread -> thread.start());
        ts.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        long end = System.nanoTime();
        System.out.println(account.getBalance()
                + " cost: " + (end - start) / 1000_000 + " ms");
    }
}
```

除了加锁之外，还可以不加锁实现线程安全

```
@Slf4j(topic = "guizy.Test1")
public class TestAccount {

    public static void main(String[] args) {
        Account account = new AccountCas(10000);
        Account.demo(account);
    }
```

```
}

class AccountCas implements Account {
    private AtomicInteger balance;

    public AccountCas(int balance) {
        this.balance = new AtomicInteger(balance);
    }

    @Override
    public Integer getBalance() {
        return balance.get();
    }

    @Override
    public void withdraw(Integer amount) {
        while (true) {
            int previous = balance.get();
            int next = previous - amount;
            if (balance.compareAndSet(previous, next)) {
                break;
            }

        }
    }
}


interface Account {
    // 获取余额
    Integer getBalance();

    // 取款
    void withdraw(Integer amount);

    /**
     * Java8之后接口新特性，可以添加默认方法
     * 方法内会启动 1000 个线程，每个线程做 -10 元 的操作
     * 如果初始余额为 10000 那么正确的结果应当是 0
     */
    static void demo(Account account) {
        List<Thread> ts = new ArrayList<>();
        long start = System.nanoTime();
        for (int i = 0; i < 1000; i++) {
            ts.add(new Thread(() -> {
```

```
                account.withdraw(10);
            }));
        }
        ts.forEach(thread -> thread.start());
        ts.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        long end = System.nanoTime();
        System.out.println(account.getBalance()
                + " cost: " + (end - start) / 1000_000 + " ms");
    }
}
```

`compareAndSet()` 会比较并设置，会比较第一个参数和调用他的数值是否相同（也就是上述例子中的balance和previous的值是否相同），如果相同，会将调用这个方法的变量的值设置为第二个参数值

1. CAS与volatile

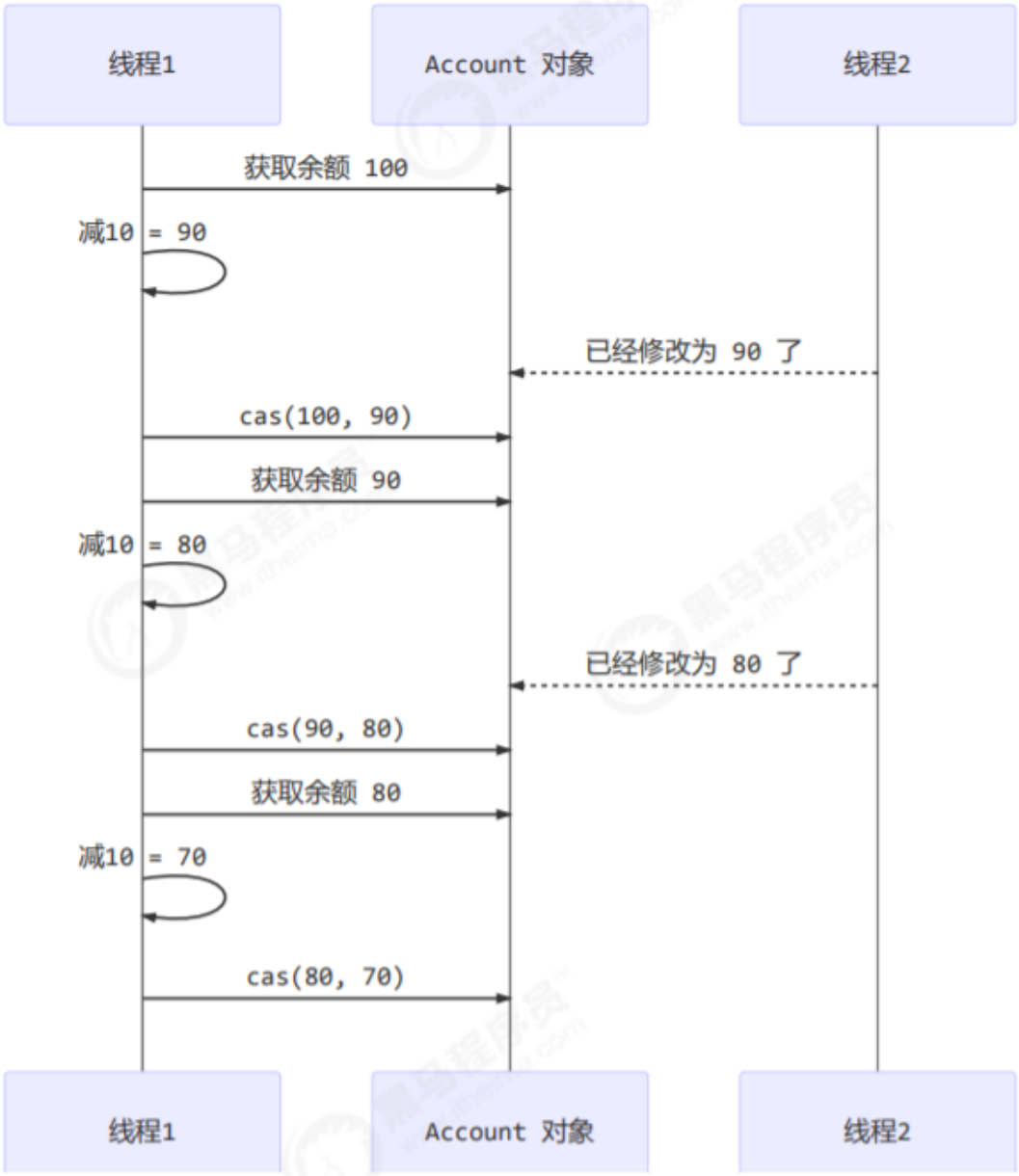
1.1 回顾转账示例

前面看到的 `AtomicInteger` 的解决方法，内部并 没有用锁 来保护 共享变量 的线程安全。那么它是如何实现的呢?

```
@Override
public void withdraw(Integer amount) {
    // 核心代码
    // 需要不断尝试，直到成功为止
    while (true){
        // 比如拿到了旧值 1000
        int prev = balance.get();
        // 在这个基础上 1000-10 = 990
        int next = prev - amount;
        /*
         compareAndSet 保证操作共享变量安全性的操作:
         ① 线程A首先获取balance.get(),拿到当前的balance值prev
         ② 根据这个prev值 - amount值 = 修改后的值next
```

```
            ③ 调用compareAndSet方法， 首先会判断当初拿到的prev值,是否和现在的
                balance值相同;
                3.1、如果相同,表示其他线程没有修改balance的值， 此时就可以将
next值
                    设置给balance属性
                3.2、如果不相同,表示其他线程也修改了balance值， 此时就设置next
值失败,
                    然后一直重试， 重新获取balance.get()的值,计算出next值,
                    并判断本次的prev和balnce的值是否相同...重复上面操作
            */
            if (atomicInteger.compareAndSet(prev,next)){
                break;
            }
        }
}
```

其中的关键是 `compareAndSwap（比较并设置值）`，它的简称就是 `CAS` （也有 Compare And Swap 的说法），它必须是原子操作。



流程：

当一个线程要去修改Account对象中的值时，先获取值prev（调用get方法），然后再将其设置为新的值next（调用cas方法）。在调用cas方法时，会将previous与Account中的余额进行比较。

- 如果两者 相等 ，就说明该值还未被其他线程修改，此时便可以进行修改操作。
- 如果两者 不相等 ，就不设置值，重新获取值prev（调用get方法），然后再将其设置为新的值next（调用cas方法），直到修改成功为止。

1.2 volatile在CAS中的作用

在上面代码中的 `AtomicInteger类` ，**保存值的value属性使用了volatile 修饰。**获取共享变量时，为了 保证该变量的可见性 ，需要使用 **volatile 修饰。**

volatile可以用来修饰 **成员变量和静态成员变量**，他可以避免线程从自己的工作缓存中查找变量的值，必须到主存中获取它的值，线程操作 volatile 变量都是直接操作主存。**即一个线程对 volatile 变量的修改，对另一个线程可见。**

> 注意：volatile 仅仅保证了共享变量的可见性，让其它线程能够看到最新值，但不能解决指令交错问题（不能保证原子性）

CAS 必须借助 volatile 才能读取到共享变量的最新值来实现【比较并交换】的效果

1.3 为什么无锁效率高

- 使用CAS---无锁情况下，即使循环失败，**线程始终在高速运行，没有停歇**，而 `synchronized` 会让线程在没有获得锁的时候，发生上下文切换（会保存线程相关数据再将其停止，下次运行时还需要恢复之前运行时的数据），进入阻塞。
  - 打个比喻：线程就好像高速跑道上的赛车，高速运行时，速度超快，一旦发生上下文切换，就好比赛车要减速、熄火，等被唤醒又得重新打火、启动、加速... 恢复到高速运行，代价比较大
- 但无锁情况下，因为线程要保持运行，需要额外 CPU 的支持，CPU 在这里就好比高速跑道，没有额外的跑道，线程想高速运行也无从谈起，虽然不会进入阻塞，但由于没有分到时间片，仍然会进入可运行状态，还是会导致上下文切换。

> 注：线程数少于核心数，CAS效率高，如果高于核心数，效率也会降低

1.4 CAS的特点

结合 CAS 和 volatile 可以实现 无锁并发 ，适用于**线程数少、多核 CPU**的场景下。

- CAS 是基于乐观锁的思想：**最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。**
- synchronized是基于悲观锁的思想：**最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。**
- CAS 体现的是**无锁并发、无阻塞(使线程一直运行，不发生上下文切换)并发**
  - 因为没有使用 synchronized，所以线程不会陷入阻塞，这是效率提升的因素之一
  - 但如果竞争激烈(写操作多)，可以想到重试必然频繁发生，**反而效率会受影响**

2.原子整数

java.util.concurrent.atomic 并发包提供了一些并发工具类，这里把它分成五类：

- 使用原子的方式(共享数据为基本数据类型原子类)
  - AtomicInteger：整型原子类
  - AtomicLong：长整型原子类
  - AtomicBoolean ： 布尔型原子类

上面三个类提供的方法几乎相同，所以我们将以 AtomicInteger为例子来介绍。

```
public class AtomicIntegerTest {
    public static void main(String[] args) {
        AtomicInteger i = new AtomicInteger(1);

        //false
        System.out.println(i.compareAndSet(2, 1));
        //true
        System.out.println(i.compareAndSet(1, 2));
        //2
        System.out.println(i);

        //自增再获取值，输出值为3
        System.out.println(i.incrementAndGet());
        System.out.println("After invoking incrementAndGet: i = "
+ i);

        //获取再自增，输出为3
        System.out.println(i.getAndIncrement());
        System.out.println("After invoking getAndIncrement: i = "
+ i);

        System.out.println(i.get());

        //获取并且增加相应的值（值为传入的参数值）
        System.out.println(i.getAndAdd(10));
        System.out.println("After invoking getAndAdd: i = " + i);

        i.updateAndGet(x -> x * 10);
        System.out.println(i.get());
    }
}
```

上面的例子中withdraw方法

```
@Override
public void withdraw(Integer amount) {
    while (true) {
        int previous = balance.get();
        int next = previous - amount;
        if (balance.compareAndSet(previous, next)) {
            break;
        }

    }
}
```

可以修改为

```
@Override
public void withdraw(Integer amount) {
    balance.getAndAdd(-1 * amount);
}
```

3. 原子引用

3.1 概述

原子引用的作用: **保证引用类型的共享变量是线程安全的(确保这个原子引用没有引用过别人**

为什么需要原子引用类型? **保证引用类型的共享变量是线程安全的（确保这个原子引用没有引用过别人）。**

原子引用类型，基本类型原子类只能更新一个变量，如果需要原子更新多个变量，需要使用引用类型原子类：

- **AtomicReference**：引用类型原子类
- **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。**
- **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，也可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

3.2 转账问题

当转账余额的类型变为BigDecimal之类的可以采用AtomicReference

```
public class AtomicReferenceTest {
    public static void main(String[] args) {
        DecimalAccount.demo(new
DecimalAccountCas(BigDecimal.valueOf(10000)));
    }
}

class DecimalAccountCas implements DecimalAccount {
    private AtomicReference<BigDecimal> balance;

    public DecimalAccountCas(BigDecimal balance) {
        this.balance = new AtomicReference<>(balance);
    }

    @Override
    public BigDecimal getBalance() {
        return balance.get();
    }

    @Override
    public void withdraw(BigDecimal amount) {
        while (true) {
            BigDecimal prev = balance.get();
            BigDecimal next = prev.subtract(amount);
            if (balance.compareAndSet(prev, next)) {
                break;
            }
        }
    }
}

interface DecimalAccount {
    // 获取余额
    BigDecimal getBalance();

    // 取款
    void withdraw(BigDecimal amount);

    /**
```

```
     * 方法内会启动 1000 个线程，每个线程做 -10 元 的操作
     * 如果初始余额为 10000 那么正确的结果应当是 0
     */
    static void demo(DecimalAccount account) {
        List<Thread> ts = new ArrayList<>();
        for (int i = 0; i < 1000; i++) {
            ts.add(new Thread(() -> {
                account.withdraw(BigDecimal.TEN);
            }));
        }
        ts.forEach(Thread::start);
        ts.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        System.out.println(account.getBalance());
    }
}
```

3.3 ABA问题

如下程序所示，虽然 在other方法中存在两个线程对共享变量进行了修改，但是修改之后又变成了原值，main线程对修改过共享变量的过程是不可见的，这种操作这对业务代码并无影响。

```
public class Test1 {

    static AtomicReference<String> ref = new AtomicReference<>
("A");

    public static void main(String[] args) {
        new Thread(() -> {
            String pre = ref.get();
            System.out.println("change");
            try {
                other();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Sleeper.sleep(1);
```

```
            //把ref中的A改为C
            System.out.println("change A->C " +
ref.compareAndSet(pre, "C"));
        }).start();
    }

    static void other() throws InterruptedException {
        new Thread(() -> {
            // 此时ref.get()为A,此时共享变量ref也是A,没有被改过，此时
CAS
            // 可以修改成功，B
            System.out.println("change A->B " +
ref.compareAndSet(ref.get(), "B"));
        }).start();
        Thread.sleep(500);
        new Thread(() -> {
            // 同上，修改为A
            System.out.println("change B->A " +
ref.compareAndSet(ref.get(), "A"));
        }).start();
    }
}
```

3.4 ABA问题解决方法-AtomicStampedReference

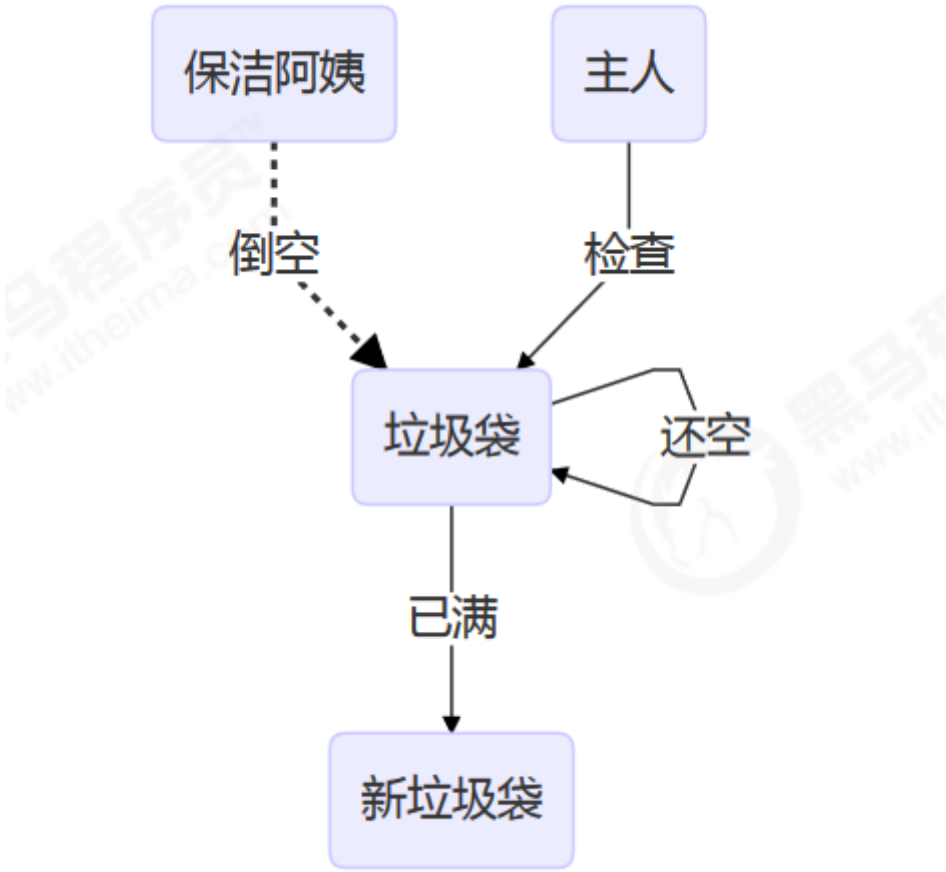
主线程仅能判断出共享变量的值与最初值 A是否相同，不能感知到这种从 A 改为 B 又改回 A 的情况，如果主线程希望：只要有其它线程动过共享变量，那么自己的 cas 就算失败，这时，仅比较值是不够的，需要再加一个版本号。使用AtomicStampedReference来解决。

```
public class Test1 {
    //指定版本号
    static AtomicStampedReference<String> ref = new
AtomicStampedReference<>("A", 0);

    public static void main(String[] args) {
        new Thread(() -> {
            String pre = ref.getReference();
            //获得版本号
            int stamp = ref.getStamp(); // 此时的版本号还是第一次获取
的
```

```
            System.out.println("change");
            try {
                other();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //把ref中的A改为C,并比对版本号，如果版本号相同，就执行替换，并让版本号+1
            System.out.println("change A->C stamp " + stamp + ref.compareAndSet(pre, "C", stamp, stamp + 1));
        }).start();
    }

    static void other() throws InterruptedException {
        new Thread(() -> {
            int stamp = ref.getStamp();
            System.out.println("change A->B stamp " + stamp + ref.compareAndSet(ref.getReference(), "B", stamp, stamp + 1));
        }).start();
        Thread.sleep(500);
        new Thread(() -> {
            int stamp = ref.getStamp();
            System.out.println("change B->A stamp " + stamp + ref.compareAndSet(ref.getReference(), "A", stamp, stamp + 1));
        }).start();
    }
}
```

3.5 AtomicMarkableReference (标记cas的共享变量是否被修改过)

- `AtomicStampedReference` 可以给 原子引用 加上 版本号 ，**追踪原子引用整个的变化过程**，如：A -> B -> A ->C，通过AtomicStampedReference，我们可以知道，引用变量中途被更改了几次。
- 但是有时候， 并不关心引用变量更改了几次，只是单纯的关心是否更改过 ，所以就有了 `AtomicMarkableReference`




```
@Slf4j(topic = "guizy.TestABAAtomicMarkableReference")
public class TestABAAtomicMarkableReference {
    public static void main(String[] args) throws
InterruptedException {
        GarbageBag bag = new GarbageBag("装满了垃圾");

        // 参数2 mark 可以看作一个标记，表示垃圾袋满了
        AtomicMarkableReference<GarbageBag> ref = new
AtomicMarkableReference<>(bag, true);
        log.debug("主线程 start...");

        GarbageBag prev = ref.getReference();
        log.debug(prev.toString());

        new Thread(() -> {
            log.debug("打扫卫生的线程 start...");
            bag.setDesc("空垃圾袋");
            while (!ref.compareAndSet(bag, bag, true, false)) {
            }
            log.debug(bag.toString());
        }).start();
```

```
        Thread.sleep(1000);
        log.debug("主线程想换一只新垃圾袋? ");

        boolean success = ref.compareAndSet(prev, new
GarbageBag("空垃圾袋"), true, false);
        log.debug("换了么? " + success);
        log.debug(ref.getReference().toString());
    }
}

class GarbageBag {
    String desc;

    public GarbageBag(String desc) {
        this.desc = desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }

    @Override
    public String toString() {
        return super.toString() + " " + desc;
    }
}
```

```
23:00:24.062 guizy.TestABAAtomicMarkableReference [main] - 主线程
start...
23:00:24.069 guizy.TestABAAtomicMarkableReference [main] -
com.guizy.cas.GarbageBag@2be94b0f 装满了垃圾
23:00:24.312 guizy.TestABAAtomicMarkableReference [Thread-0] - 打
扫卫生的线程 start...
23:00:24.313 guizy.TestABAAtomicMarkableReference [Thread-0] -
com.guizy.cas.GarbageBag@2be94b0f 空垃圾袋
23:00:25.313 guizy.TestABAAtomicMarkableReference [main] - 主线程想
换一只新垃圾袋?
23:00:25.314 guizy.TestABAAtomicMarkableReference [main] - 换了么?
false
23:00:25.314 guizy.TestABAAtomicMarkableReference [main] -
com.guizy.cas.GarbageBag@2be94b0f 空垃圾袋
```

3.6 AtomicStampedReference和AtomicMarkableReference两者的区别

- `AtomicStampedReference` 需要我们传入 **整型变量** 作为 版本号 ， 来判定是否被更改过
- `AtomicMarkableReference` ***需要我们传入*** **布尔变量** 作为 标记 ， 来判断是否被更改过

4.原子数组

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

上面三个类提供的方法几乎相同，所以我们这里以 `AtomicIntegerArray` 为例子来介绍。实例代码

普通数组内元素, 多线程访问造成安全问题

```
public class AtomicArrayTest {
    public static void main(String[] args) {
        demo(
                () -> new int[10],
                array -> array.length,
                (array, index) -> array[index]++,
                array ->
System.out.println(Arrays.toString(array))
        );
    }

    /**
     * 参数1，提供数组、可以是线程不安全数组或线程安全数组
     * 参数2，获取数组长度的方法
     * 参数3，自增方法，回传 array，index
     * 参数4，打印数组的方法
     */
    // supplier 提供者 无中生有 ()->结果
    // function 函数 一个参数一个结果 (参数)->结果 , BiFunction (参数1,
参数2)->结果
    // consumer 消费者 一个参数没结果 (参数)->void, BiConsumer (参数1,
参数2)->void
```

```
    private static <T> void demo(Supplier<T> arraySupplier,
Function<T, Integer> lengthFun,
                                 BiConsumer<T, Integer>
putConsumer, Consumer<T> printConsumer) {
        List<Thread> ts = new ArrayList<>();
        T array = arraySupplier.get();
        int length = lengthFun.apply(array);

        for (int i = 0; i < length; i++) {
            // 创建10个线程，每个线程对数组作 10000 次操作
            ts.add(new Thread(() -> {
                for (int j = 0; j < 10000; j++) {
                    putConsumer.accept(array, j % length);
                }
            }));
        }

        ts.forEach(t -> t.start()); // 启动所有线程
        ts.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }); // 等所有线程结束

        printConsumer.accept(array);
    }
}
```

```
[9870, 9862, 9774, 9697, 9683, 9678, 9679, 9668, 9680, 9698]
```

- 使用 `AtomicIntegerArray` 来创建安全数组

```
demo(
        ()-> new AtomicIntegerArray(10),
        (array) -> array.length(),
        (array, index) -> array.getAndIncrement(index),
        array -> System.out.println(array)
);

demo(
        ()-> new AtomicIntegerArray(10),
        AtomicIntegerArray::length,
        AtomicIntegerArray::getAndIncrement,
        System.out::println
);
```

```
[10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000]
```

7. 字段更新器

字段更新器保护对象的某个成员变量进行原子操作，只能配合volatile修饰的字段使用，否则会出现异常

保证多线程访问同一个对象的成员变量时, 成员变量的线程安全性

- AtomicReferenceFieldUpdater —引用类型的属性
- AtomicIntegerFieldUpdater —整形的属性
- AtomicLongFieldUpdater —长整形的属性

示例代码:

```
@Slf4j(topic = "guizy.AtomicFieldTest")
public class AtomicFieldTest {
    public static void main(String[] args) {
        Student stu = new Student();
        AtomicReferenceFieldUpdater updater =
AtomicReferenceFieldUpdater.newUpdater(Student.class,
String.class, "name");
        System.out.println(updater.compareAndSet(stu, null, "张
三"));
```

```
        System.out.println(updater.compareAndSet(stu, stu.name,
"王五"));
        System.out.println(stu);
    }
}

class Student {
    volatile String name;

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

运行结果

```
true
true
Student{name='王五'}
```

7. 原子累加器

7.1累加器性能比较 AtomicLong, LongAddr

```
@Slf4j(topic = "guizy.Test")
public class Test {
    public static void main(String[] args) {
        System.out.println("----AtomicLong----");
        for (int i = 0; i < 5; i++) {
            demo(() -> new AtomicLong(), adder ->
adder.getAndIncrement());
        }

        System.out.println("----LongAdder----");
        for (int i = 0; i < 5; i++) {
            demo(() -> new LongAdder(), adder ->
adder.increment());
        }
    }
```

```
    private static <T> void demo(Supplier<T> adderSupplier,
Consumer<T> action) {
        T adder = adderSupplier.get();
        long start = System.nanoTime();
        List<Thread> ts = new ArrayList<>();
        // 4 个线程，每人累加 50 万
        for (int i = 0; i < 40; i++) {
            ts.add(new Thread(() -> {
                for (int j = 0; j < 500000; j++) {
                    action.accept(adder);
                }
            }));
        }
        ts.forEach(t -> t.start());
        ts.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        long end = System.nanoTime();
        System.out.println(adder + " cost:" + (end - start) /
1000_000);
    }
}
```

```
----AtomicLong----
20000000 cost:646
20000000 cost:707
20000000 cost:689
20000000 cost:713
20000000 cost:657

----LongAdder----
20000000 cost:148
20000000 cost:139
20000000 cost:130
20000000 cost:122
20000000 cost:116
```

性能比较：

- LongAdder性能提升的原因很简单，就是在有竞争时，设置多个 累加单元 (但不会超过cpu的核心数)，Therad-0 累加 Cell[0]，而 Thread-1 累加Cell[1]... 最后将结果汇总。这样它们在累加时操作的不同的 Cell 变量，因此减少了 CAS 重试失败，从而提高性能。
- 之前AtomicLong等都是在一个共享资源变量上进行竞争，`while(true)` 循环进行CAS重试, 性能没有LongAdder高

7.2 LongAdder源码

LongAdder关键域

```
// 累加单元数组，懒惰初始化
transient volatile Cell[] cells;

// 基础值，如果没有竞争，则用CAS累加这个域
transient volatile long base;

// 在cells创建或扩容时，置为1，表示加锁
transient volatile int cellsBusy;
```

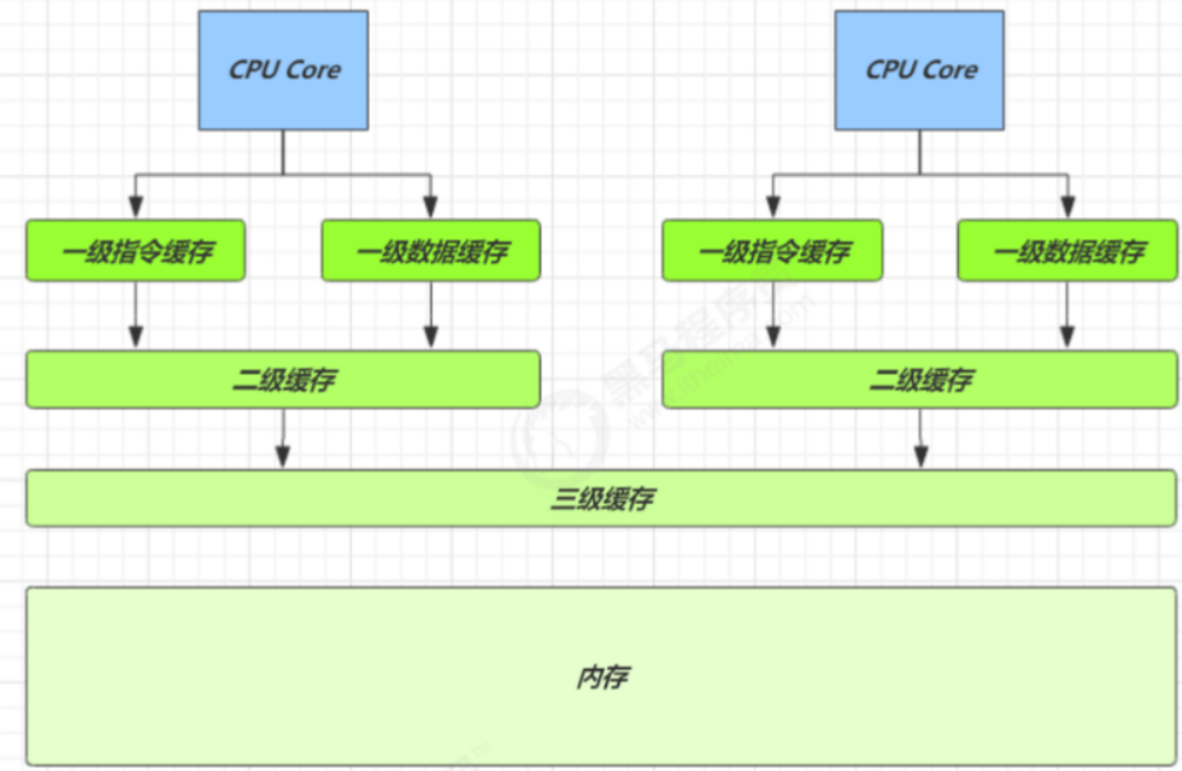
7.3 伪共享

LongAdder中Cell类源代码，其中@sun.misc.Contended注解是为了**防止缓存行共享**

```
@sun.misc.Contended
static final class Cell {
    volatile long value;
    Cell(long x) { value = x; }
    final boolean cas(long cmp, long val) {
        return UNSAFE.compareAndSwapLong(this, valueOffset, cmp,
val);
    }

    //下面的是不重要的代码
    private static final sun.misc.Unsafe UNSAFE;
    private static final long valueOffset;
    static {
        try {
```

```
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class<?> ak = Cell.class;
            valueOffset = UNSAFE.objectFieldOffset
                (ak.getDeclaredField("value"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
}
```

缓存和内存关系图

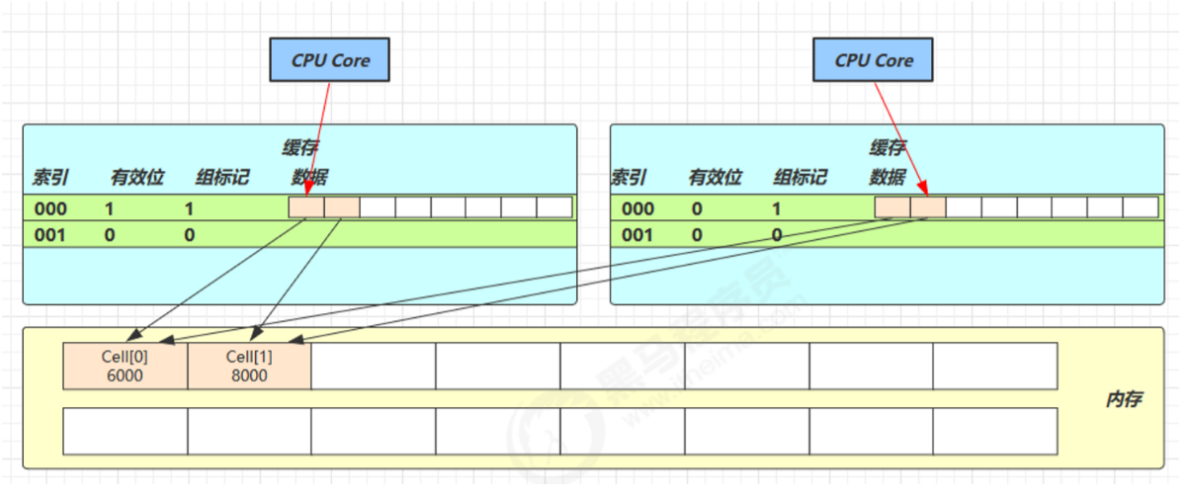


因为 CPU 与 内存的速度差异很大，需要靠预读数据至缓存来提升效率。

而缓存以缓存行为单位，每个缓存行对应着一块内存，一般是 64 byte（8 个 long）

缓存的加入会造成数据副本的产生，即同一份数据会缓存在不同核心的缓存行中

CPU 要保证数据的一致性(缓存一致性)，如果某个 CPU 核心更改了数据，其它 CPU 核心对应的整个缓存行必须失效

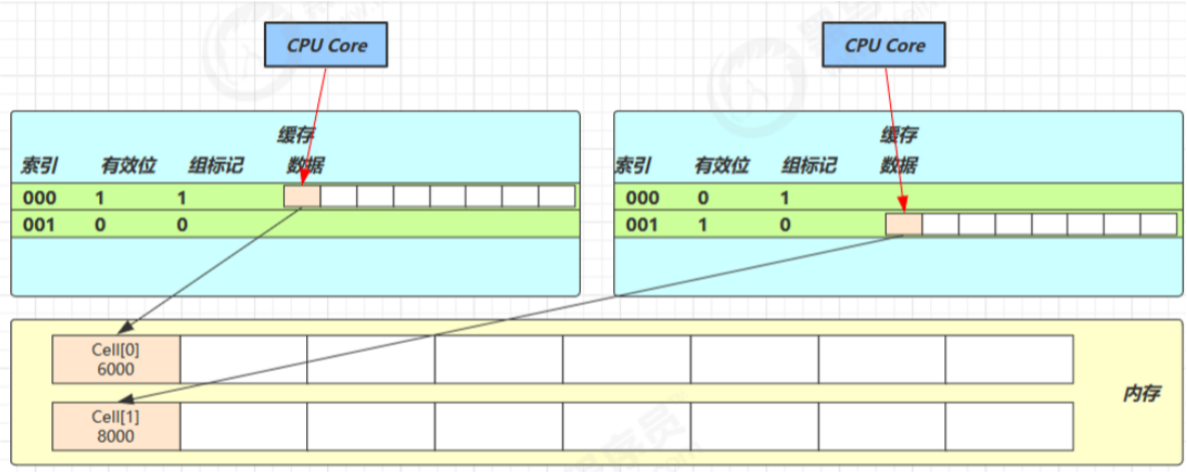


因为 Cell 是数组形式，在内存中是连续存储的，一个 Cell 为 24 字节（16 字节的对象头和 8 字节的 value），因 此缓存行可以存下 2 个的 Cell 对象。这样问题来了：

- Core-0 要修改 Cell[0]
- Core-1 要修改 Cell[1]

无论谁修改成功，都会导致对方 Core 的缓存行失效，

- 比如 Core-0 中 Cell[0]=6000, Cell[1]=8000 要累加 Cell[0]=6001, Cell[1]=8000，这时会让 Core-1 的缓存行失效
- @sun.misc.Contended 用来解决这个问题，它的原理是在使用此注解的对象或字段的**前后各增加 128 字节大小的 padding**（空白），从而让 CPU 将对象预读至缓存时**占用不同的缓存行**，这样，不会造成对方缓存行的失效



8. Unsafe

8.1 概述

Unsafe对象提供了非常底层的，操作内存、线程的方法，Unsafe对象不能直接调用，只能通过反射获得