

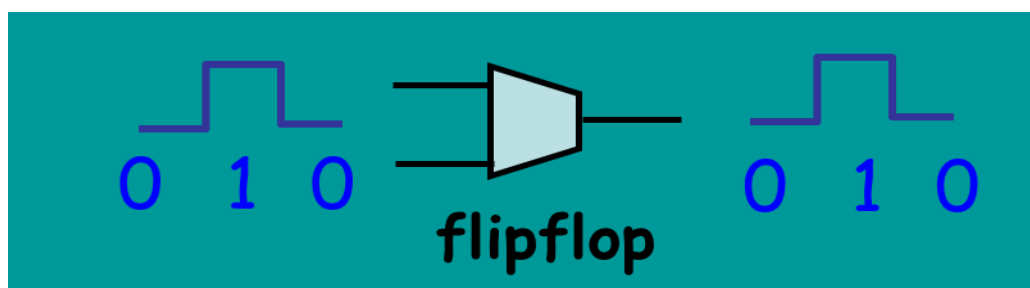
第四讲 共享内存基础

一. 引言

并发计算有两种主要的模式，共享内存和消息传递模式。两种模式各有自己的优缺点，一般而言，共享内存的计算模式更为直观，效率更高。课程默认的计算模式为共享内存的并发计算模式。首先摆在我们面前的问题是，共享内存并发计算的模型是什么？

图灵机是串行计算的标准计算模型，它关注的是哪些问题是可以计算的，哪些问题是不可计算的，如停机问题。至于图灵机的效率问题，一般不会多加关注。本讲和后面几讲的任务，就是提出一个和图灵机类似的共享内存的计算模型，考察在此模型下，哪些问题可以在特定的并发性质下是可计算的，哪些问题在此性质下是不可计算的。

并发计算和串行计算的最大不同在于，对于内存单元（图灵机的纸带上的单元格）的访问是并发的，这样原来满足串行计算要求的内存单元，无法满足并发计算的要求。对于串行计算来说，内存单元（历史原因称为寄存器）可以看成是一个 D 触发器，



D 触发器的输出电平需要一段时间后才能稳定.这一特性在串行计算是没问题的，单线程的程序总是在输出稳定后再执行下一条指令，单线程的读指令可以正确读到最近写指令的新值。所以这种寄存器被称为 **safe** 的。但是在多个线程同时访问一个寄存器的情况下，由于线程访问寄存器的时机是无法控制的，会出现某个线程改变了寄存器的值后，在电平还未稳定的情况下，另一个线程就开始读这个寄存器的值。原本在串行计算下 **safe** 的寄存器在并发计算下不再 **safe**!

那有没有满足并发计算要求的寄存器呢？答案是肯定的。

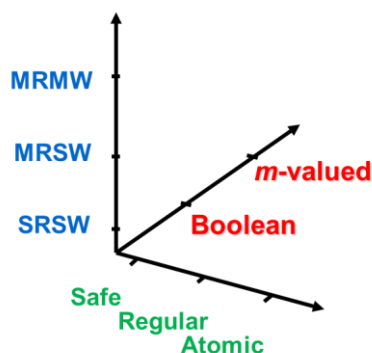
寄存器的定义和分类

寄存器的 Java 描述如下：

```
public interface Register<T> {  
    public T read(); //T 表示内存单元的类型，通常为布尔或者 m 位的整型  
    public void write(T v);  
}
```

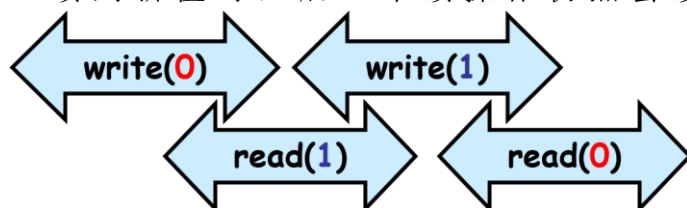
}

寄存器可从三个维度进行分类，如下图所示。按照寄存器存储的值这一维度划分，可以分为布尔类型和多值两种，多值寄存器可以视为布尔寄存器的一个数组。



按照线程的访问情况分为：单读单写（Single-Reader Single-Writer），多读单写（Multi-Reader Single-Writer）和 MRMW 为多读多写（Multi-Reader Multi-Writer）三类，这里的多是指寄存器可以被多个线程同时访问。按照寄存器满足的性质划分，可以分为安全的（Safe），正规的（Regular）和原子的（Atomic）三类。下面我们详细解释 Safe，Regular 和 Atomic 的含义。

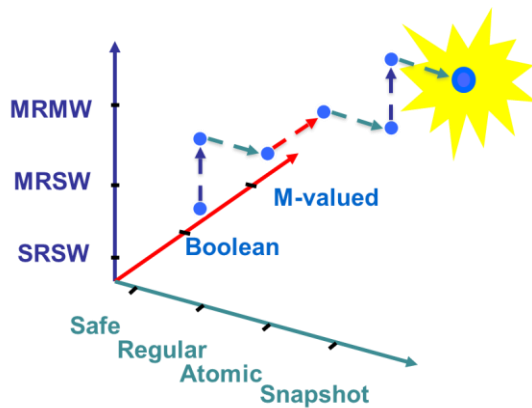
对于 safe 寄存器来说，如果写操作和读操作不重叠，读操作会正确读出之前一个写操作的值；如果写操作和读操作重叠，读操作可以读到该寄存器所允许的任意值。Regular 寄存器在此基础上增加了一条限制，当读写操作重叠时，读操作要么读到当前重叠的写操作的值，要么读到在读操作之前完成的最后一个写操作的值。Regular 寄存器的问题在于，对于连续的两个读操作，当前一个读操作已读到新值时，后一个读操作仍然会读到旧值，如下图所示。



这个现象仍然违反我们的直觉。为此我们再增加一条限制：当前一个读操作读到新值后，后续的读操作不再被允许读到旧值。满足这些性质的寄存器是原子的。满足原子性的寄存器在本质上就是可线性化的寄存器。

二. 从安全 SRSW 布尔寄存器构建原子 MRMW 多值寄存器

看起来 safe 寄存器和 atomic 寄存器的性质差别很大，我们却可以用 safe 寄存器，一步步地构建出 atomic 寄存器，甚至可以构建寄存器组的原子快照。



在这里，我们不能用前几讲里介绍的各种互斥协议或者是互斥锁来构建，因为我们讨论的是共享内存基础，这个共享内存的底层就是硬件提供的寄存器，所以只能用寄存器来设计互斥协议，不能用互斥协议来构建寄存器。

具体的路线图如下：SRSW safe Boolean—>MRSW safe Boolean—>MRSW regular Boolean—>MRSW regular—>MRSW atomic—>MRMW atomic
在构建过程中，要求构建的所有方法必须是无等待（wait free）的，这样所有寄存器相关的动作可以做到与系统的调度策略相独立。

1. 安全布尔单读单写至安全布尔多读单写

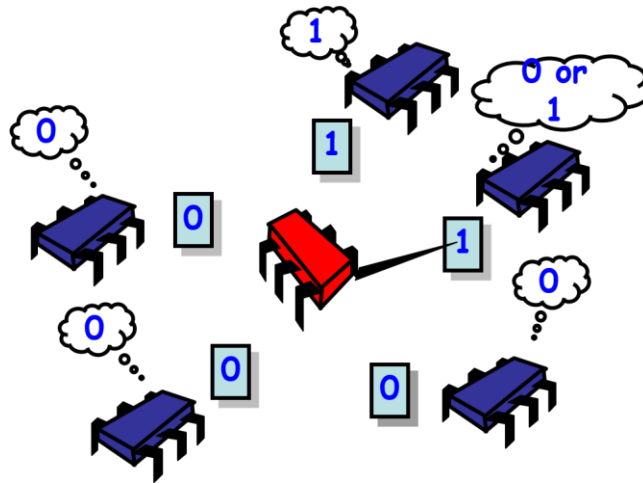
//内存单元命名规则：性质+类型+读写类型+Register

```
public class SafeBoolMRSWRegister
    implements Register<Boolean> {
    public boolean read() { ... }
    public void write(boolean x) { ... }
}
```

```
public class SafeBoolMRSWRegister
    implements Register<Boolean> {
```

//每个进程都有自己的安全单读单写的内存单元

```
private SafeBoolSRSWRegister[] r = new SafeBoolSRSWRegister[N];
public void write(boolean x) {
    for (int j = 0; j < N; j++)
        r[j].write(x); //写内存
}
public boolean read() {
    int i = ThreadID.get();
    return r[i].read(); //读自己的内存
}}
```



实现方式很简单，用单读单写寄存器数组构建一个多读单写寄存器。写线程依次向所有寄存器数组元素写新值，读线程读取自己所在数组元素的寄存器的值。由于每个寄存器都是 safe 的布尔寄存器，那么在写线程的写操作执行期间，读操作要么读到 0，要么读到 1。从宏观上看，整个多读单写寄存器的行为是 safe 的。

2. 安全布尔多读单写至正规布尔多读单写

这里分两种情况。一种情况是写入新值，那么安全和常规的布尔寄存器的行为是相同的，因为并发读取所观察到的任意值只能是 0 或 1，只不过一个是旧值，另一个是新值。如果将相同的值重复写入寄存器，接下来的事情就比较诡异了。安全性表示并发读取可以返回任何值，因此即使写入 0 覆盖 0，读线程也可以读取 1。对于正规寄存器来说，这是不可接受的，因为 regular 要求读线程要么读到旧值 0，要么读到新值，仍然是 0！读到 1 肯定违反正规性。解决的方式很简单，通过预存所写的最后一个值来避免这个问题。当写线程试图再次写相同的值时，不对寄存器进行写操作即可。

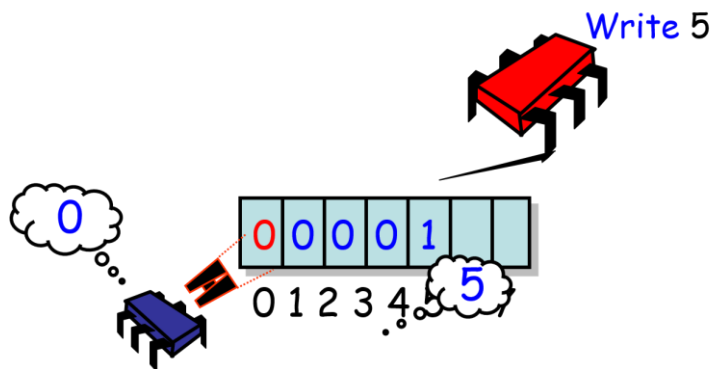
```
public class RegBoolMRSWRegister implements Register<Boolean> {
    private boolean old; // 线程所写最后一位
    private SafeBoolMRSWRegister value; // 实际的值
    public void write(boolean x) {
        if (old != x) { // 判断新旧值
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

那么能否用类似的思路用安全多值多读单写寄存器构建常规多值多读单写寄存器呢？答案是否定的。当写入新值时，正规性要求读操作只能返回旧值或新值，而安全性可以返回寄存器允许的任何值，不能满足正规性的要求。

3. 正规布尔多读单写至常规多值多读单写

解决的办法是再次扩充寄存器，用一个寄存器数组表示一个寄存器。

```
public class RegMRSWRegister implements Register{
    RegBoolMRSWRegister[M] bit;
    public void write(int x) {
        this.bit[x].write(true);
        for (int i=x-1; i>=0; i--)
            this.bit[i].write(false);
    }
    public int read() {
        for (int i=0; i < M; i++)
            if (this.bit[i].read())
                return i;
    }
}
```



布尔数组的每一个元素均为多值寄存器的值特征函数，写操作首先将对应本值的元素置 1，然后向下移动，沿途将所经过的数组元素值均置为 0。读操作从最低位开始读起，遇到第一个非 0 元素中止。这种方法的核心思想是利用写的可覆盖性对不同写操作的结果进行排序，保证读操作的正规性。

4. 正规单读单写至原子单读单写

为了原子性的要求，我们需要引入对逻辑时间进行计量的数据结构：时间戳（time stamp）。其直觉在于：原子寄存器就是可线性化的寄存器，需要用逻辑时间确定各种操作的先后顺序（线性化点）。

```
public class AtomicSRSWRegister implements Register{
    ThreadLocal<long> lastStamp;
    ThreadLocal<StampValue> lastRead;
    StampValue r_value;
    //StampValue 有两个分量，分别是 regular 时间戳
    //和 regular 寄存器的值

    public void write(T v) {
        stamp = lastStamp.get() + 1;
        r_value = new StampValue(stamp, v);
        lastStamp.set(stamp);
    }
}
```

```

}
public T read() {
    value = r_value;
    last = lastRead.get();
    result = StampValue.max(value, last);
    lastRead.set(result);
    return result.value;
}

```

5. 原子单读单写至原子多读单写

简单地扩充成一维数组是不行的，见下图例：

stamp	value
1:45	1234
1:45	1234
1:45	1234
1:45	1234

One per reader

写线程 A 从上往下写 2: 00 5678

Writer starts write...

stamp	value
2:00	5678
1:45	1234
1:45	1234
1:45	1234

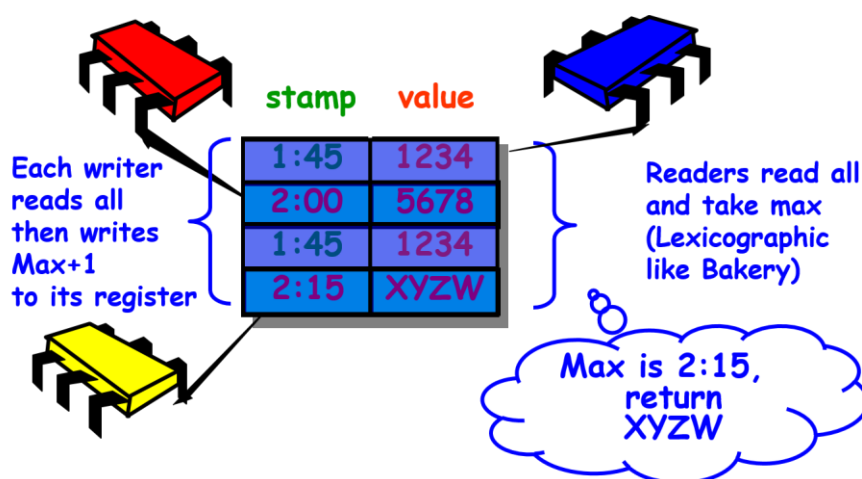
线程 B 读到 2: 00 5678。线程 C 在线程 B 完成之后读，读自己对应索引 3 的元素值为 1:45 1234，读到了旧值，不满足线性化。

解决办法是扩充成二维数组。写操作写入自己对应的整个列，每个读操作读取自己对应索引的行，然后比较哪个是最新的，把自己对应列的所有元素都更新为最新值，然后返回最新的值。这样就能保证后续的读操作在自己的行内一定能读到最近写操作的值。

	1	2	3
2:00	5678	1:45	1234
2:00	5678	1:45	1234
2:00	5678	1:45	1234

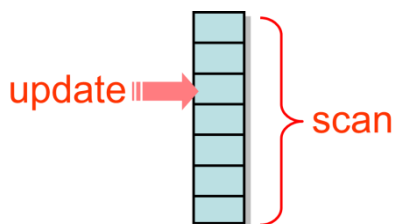
6. 原子多读单写至原子多读多写

还是采用扩充数组的方式构建。每个写线程在写入值之前先读取寄存器数组的所有时间戳，然后选出最大值并加 1 作为本次写操作的时间戳，写入本线程对应的数组位置。每个多读单写寄存器都是原子的（可线性化的），根据可线性化的组合性质和构造算法，时间戳的大小顺序和写操作的完成顺序严格一致。每个读操作读取所有的数组元素，选出时间戳最大的对应值作为读出的数值。显然，这个多读多写寄存器满足原子性。



三. 原子快照

对原子多读单写寄存器数组的访问还可以有另一种方式：原子快照。在原子快照中，每个寄存器对应一个线程，每个写线程在自己对应的索引位置写入数据，称为 `update` 方法。每个读线程使用 `scan` 方法扫描整个数组。原子快照构造了一个原子内存组的瞬间视图。



快照实现：

```
public interface Snapshot {  
    public int update(int v); // 写入数组中与调用者线程相对的内存中  
    public int[] scan(); // 返回数组的原子操作  
}
```

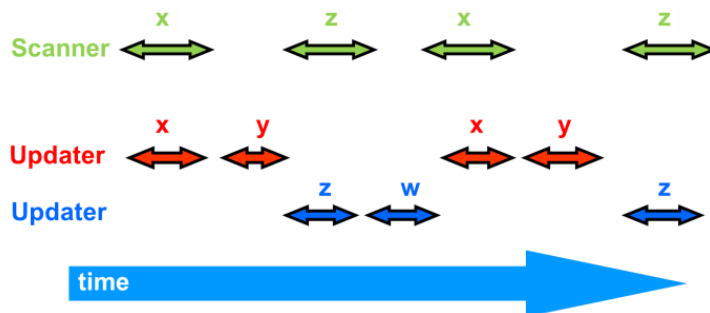
`Scan` 方法可以用收集（collect）的方式实现，即顺序读取寄存器数组。由于顺序读取整个数组不可能原子完成，中间可能会有并发的 `update` 操作，导致收集的

结果是不可线性化的。

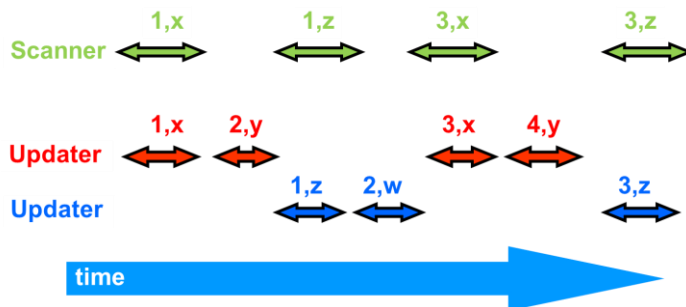
1. 简单快照

解决的办法也很直接，在收集完成后检测中间有没有并发的 `update` 操作，如果没有那么就是干净的收集。具体做法是：进行两次收集，对比两次收集的每个数组元素的值是否相同。如果所有数组元素的值均相同，即完成，否则重试。这种方法虽然简单，但是存在两个问题。一个是重复更新问题。

例：两个写线程红线程和蓝线程。当红线程写入 `x` 时，scan 收集到 `x`，然后红线程写入 `y`，紧接着蓝线程写入 `z`，scan 收集到 `z`。



实际上，一次收集 `x` 与 `z` 是不可能同时出现的，因为 `z` 在 `y` 之后写入，`y` 此时已经把 `x` 覆盖了。第二次收集获取同样的 `x` 和 `z`，表面看起来是干净的收集，但实际上并不是。通过添加时间戳（标签）可以解决此类问题，可说明是否为干净收集。读取不同时间戳的 `x` 和 `z`，可以识别不干净的收集。如下图所示。两次快照是不一样的。



实现：

```
public class SimpleSnapshot implements Snapshot {
    private AtomicMRSWRegister[] register;

    public void update(int value) {
        int i = Thread.myIndex();
        LabeledValue oldValue = register[i].read();
        LabeledValue newValue =
            new LabeledValue(oldValue.label+1, value);
        register[i].write(newValue);
    }
    private LabeledValue[] collect() {
        LabeledValue[] copy =
```



```

    new LabeledValue[n];
    for (int j = 0; j < n; j++)
        copy[j] = this.register[j].read();//读内存单元
    return copy;
}
public int[] scan() {
    LabeledValue[] oldCopy, newCopy;
    oldCopy = collect();          //第一次收集
    collect: while (true) {
        newCopy = collect(); //第二次收集
        if (!equals(oldCopy, newCopy)) { //不一样
            oldCopy = newCopy;
            continue collect;
        }
        return getValues(newCopy); //一样
    }
}

```

这种简单快照的实现方法是可线性化的，而且 Update 是无等待的。但是 Scan 方法不是无等待的，如果某次 scan 期间总是有并发的 update 打断，该 scan 方法可能会永远都无法完成。

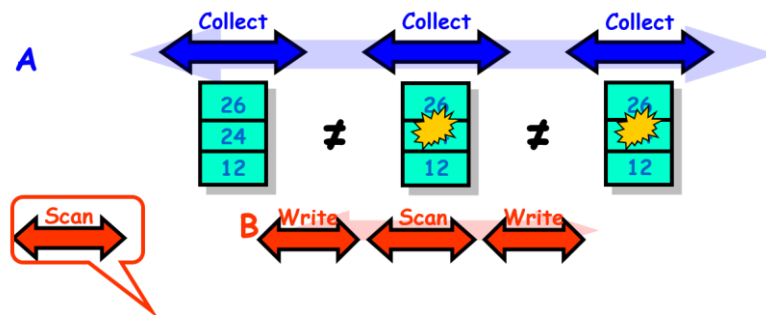
2. 无等待快照

下面是本讲的压轴：怎样构造无等待的 scan 方法，使得整个原子快照都是无等待的。按照常规的思路，我们会考虑加入某些互斥机制，把 scan 操作变成不可中断的。问题是目前还没有任何互斥的机制。我们实际上是用了多核编程中非常重要的一种思路，助人为乐的思路。我们在每次 update 之前添加一个 scan，将生成的快照与更新值一起写入，如果某个扫描持续被一些更新中断，扫描可以获取这些更新中的 scan 得到的快照。换句话说，update 会主动帮助其它被打断的 scan，提供那些 scan 想要的快照。

这里的直觉是：若一个正在 scan 的线程 A 在它进行重复收集的时候，要么直接拍到快照（其他线程均未 update）；要么一定能发现一个线程 B，线程 B 对应的数组元素值变更了两次（B 写入了两次），则 B 必定在 A 的 scan 的过程中执行了一次完整的 update 方法。A 可以直接使用 B 的 update 中的快照。

标注 1：线程 A 用的是线程 B 第二次时的 update。这个 update 的整个执行时段都被 A 的 scan 执行时段覆盖。线程 A 不能将线程 B 的最近一次快照作为他自己的快照，因为 B 的第一次 update 时快照有可能不是在 A 的扫描期间内拍摄的。

见下图：假设 B 在 A 扫描时执行了两次写入，那么 B 的第一次写入发生在 A 第一次收集后；B 的第二次写入发生在 A 第三次收集前。A 不能使用 B 第一次的快照，因为在 B 第一次快照后，可能会有另一个线程的更新发生在 A 第一次收集前，破坏 B 的快照的完整性。



标注 2: 如果 B 的第二次 update 中的 scan 被线程 C 打断, 那么我们就用 C 的快照, 然后以此类推。由于 A 和 B 是在 scan 过程中被打断, 它们会陷在 scan 过程中, 不会再对寄存器数组进行任何写操作, 这样在类推的过程中, 能进行写操作的线程会越来越少。由于总线程数是有穷的, 必然有一个线程能够完成 update 中的 scan 操作。

实现:

```
public class SnapValue {
    public int    label; //对每个快照进行计数
    public int    value; //实际值
    public int[]  snap; //最近的快照
}

public void update(int value) {
    int i = Thread.myIndex();
    int[] snap = this.scan();
    SnapValue oldValue = r[i].read();
    SnapValue newValue =
        new SnapValue(oldValue.label+1, value, snap); // 扫描计数值
    r[i].write(newValue);
}

public int[] scan() {
    SnapValue[] oldCopy, newCopy;
    boolean[] moved = new boolean[n]; //记录迁移的线程
    oldCopy = collect();
    collect: while (true) {
        newCopy = collect(); //重复收集
        for (int j = 0; j < n; j++) {
            if (oldCopy[j].label != newCopy[j].label) { //未能匹配
                if (moved[j]) { //如果线程迁移了两次, 就会获取第二个快照
                    return newCopy[j].snap;
                } else {
                    moved[j] = true; //记录迁移
                    oldCopy = newCopy;
                    continue collect;
                }
            }
        }
    }
    return getValues(newCopy);
}
```