多线程高级

目录 Contents

实现多线程

线程状态

意外状态

婴儿

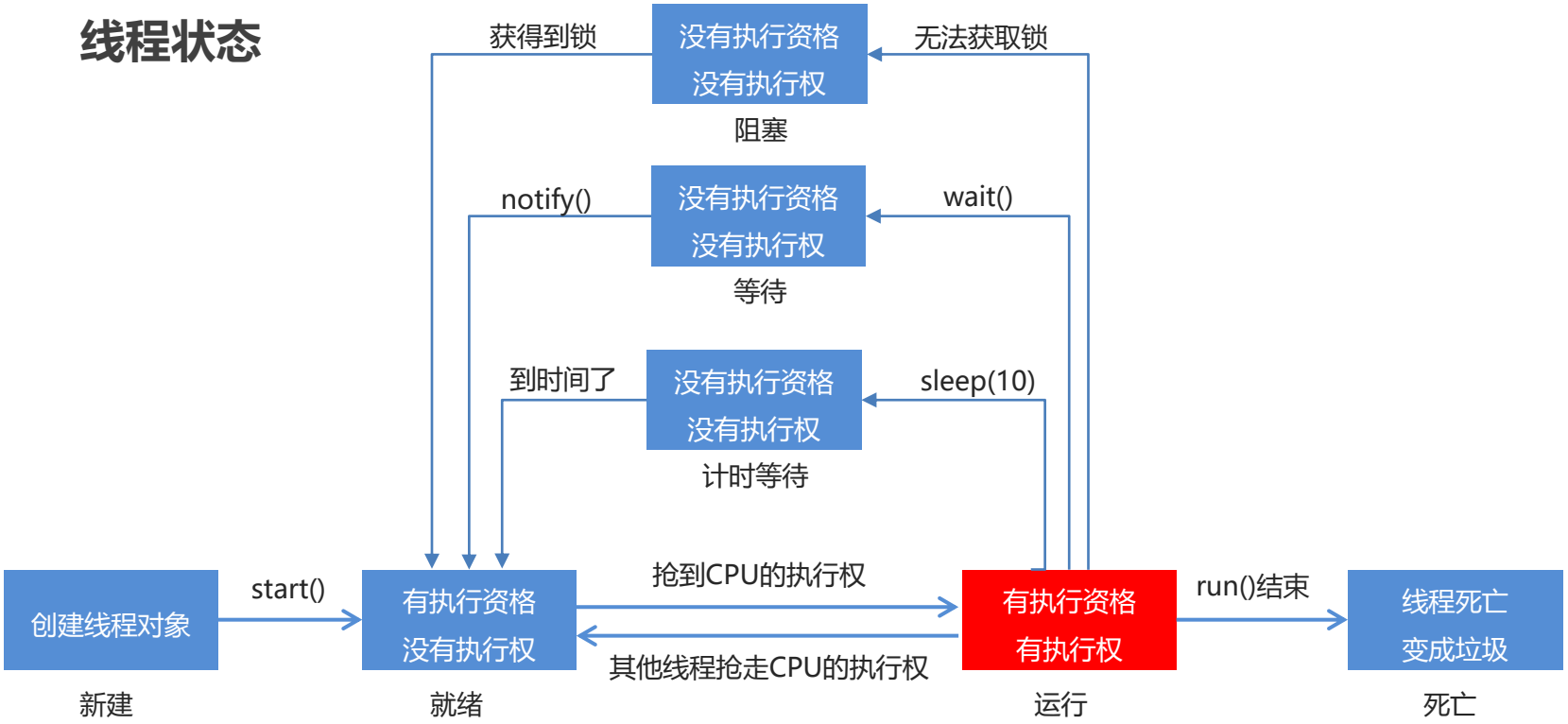
单身

婚后

GG

实现多线程

线程状态



实现多线程

小结

虚拟机中线程的六种状态：

新建状态（NEW）	→	创建线程对象
就绪状态（RUNNABLE）	→	start方法
阻塞状态（BLOCKED）	→	无法获得锁对象
等待状态（WAITING）	→	wait方法
计时等待（TIMED_WAITING）	→	sleep方法
结束状态（TERMINATED）	→	全部代码运行完毕

实现多线程

练习

1，编写一段代码，依次显示线程的状态：

NEW --- RUNNABLE --- TIME_WAITING --- RUNNABLE --- TERMINATED

2，编写一段代码，依次显示线程的状态：

NEW --- RUNNABLE --- WAITING --- RUNNABLE --- TERMINATED

3，编写一段代码，依次显示线程的状态：

NEW --- RUNNABLE --- BLOCKED --- RUNNABLE --- TERMINATED

目录 Contents

线程池

吃饭买碗故事

跑到小贾同学开的超市去买了个碗

小皮同学

线程池

吃饭买碗故事

又跑到小贾同学开的超市去买了个碗

小皮同学

线程池

故事中的问题

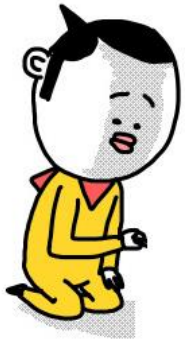
1，每次都要买碗，浪费时间

2，每次吃完都把碗摔了，浪费资源

线程池

故事的解决方案

跑到超市去买个碗



线程池

故事的解决方案

线程池

故事的解决方案

一根饭后烟思考人生

线程池

故事的解决方案

线程池

故事的解决方案

一根饭后烟思考人生

故事的解决方案

1，找一个柜子放碗，此时柜子是空的

2，第一次吃饭之前，还是要去买碗

3，吃完之后，把碗放回柜子

4，第二次吃饭的时候，就不需要买碗了。直接从柜子里面拿就可以了。

5，吃完再次将碗放回柜子。

线程池

以前写多线程的弊端

```
class MyThread extends Thread{
    @Override
    public void run() {
        多线程执行的一些代码
    }
}
```

```
MyThread t1 = new MyThread();
MyThread t2 = new MyThread();

t1.start();
t2.start();
```

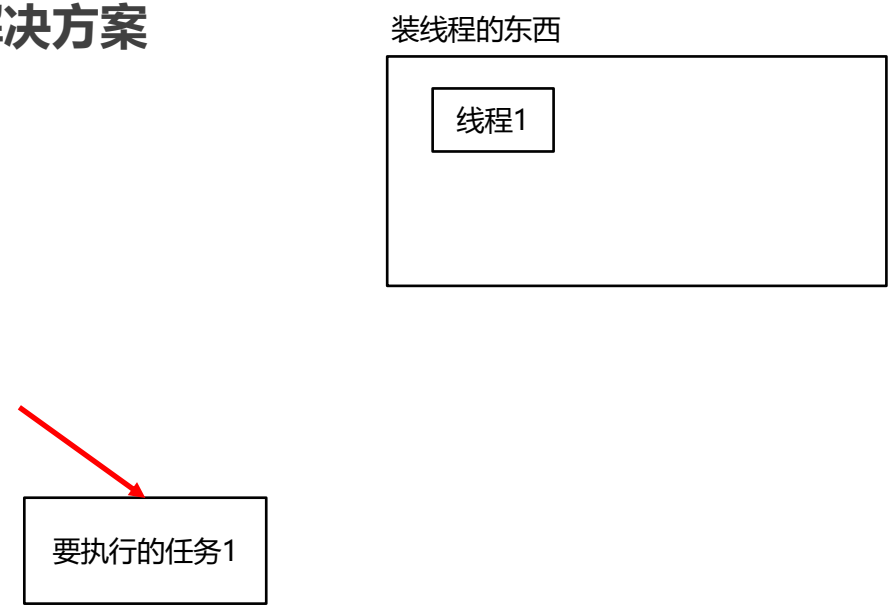
弊端：

1，用到线程的时候就创建。

2，用完之后线程消失。

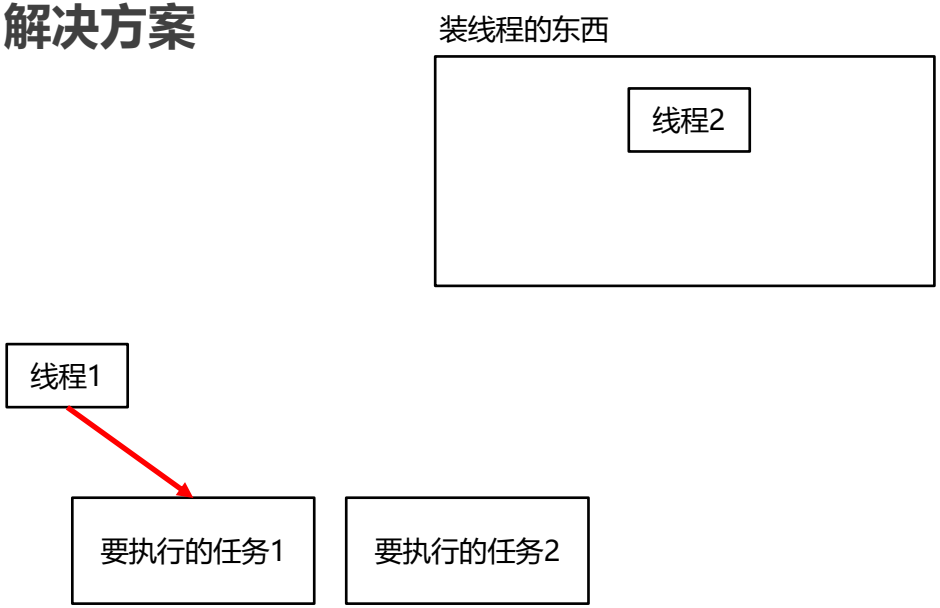
线程池

解决方案



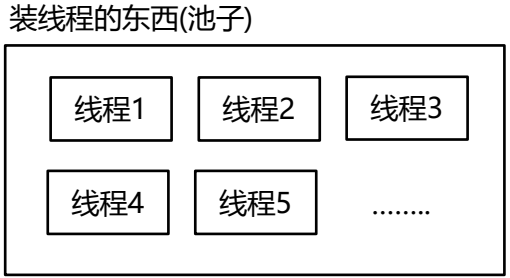
线程池

解决方案



线程池

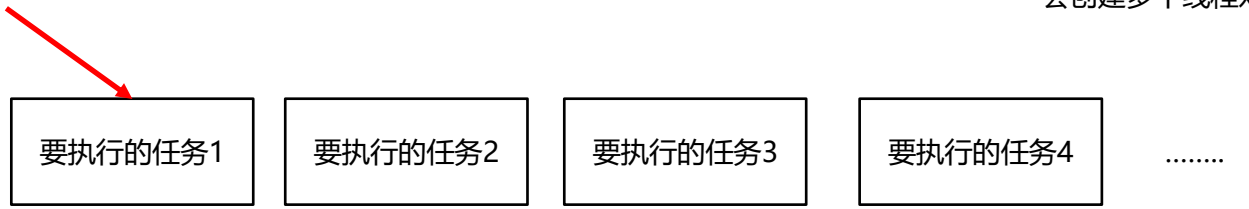
步骤总结



1，创建一个池子，池子中是空的

2，有任务需要执行时，才会创建线程对象。
当任务执行完毕，线程对象归还给池子。

3，当有多个任务需要同时执行时，
会创建多个线程对象。



线程池

代码实现

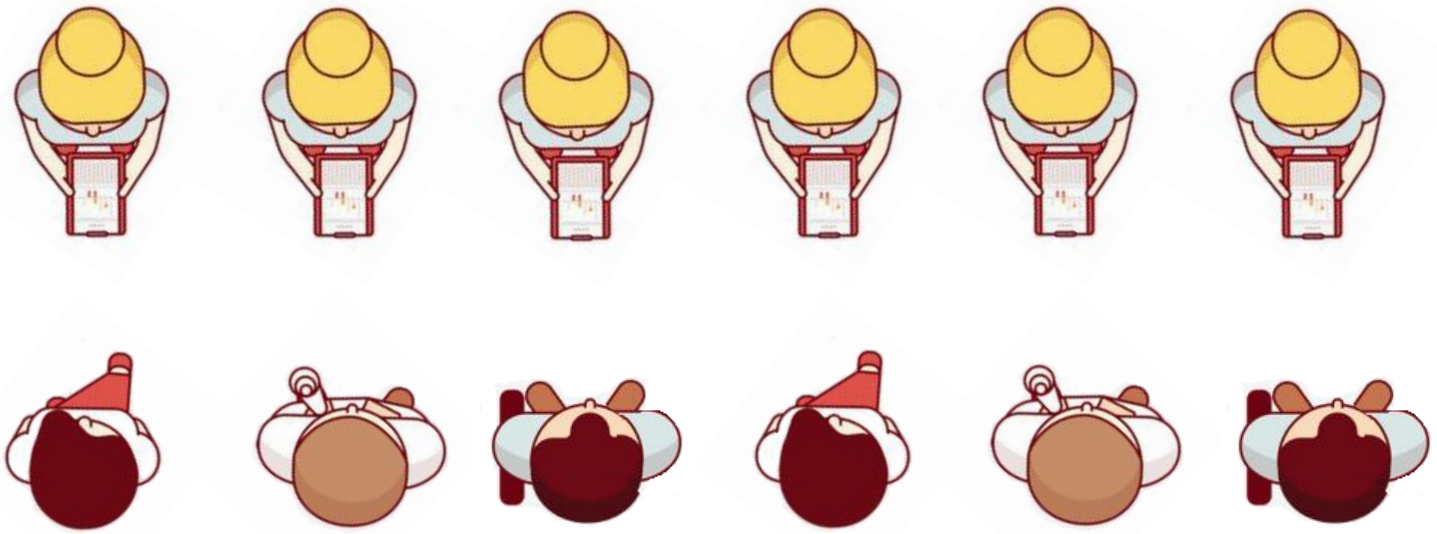
1，创建一个池子，池子中是空的 → 创建Executors中的静态方法

2，有任务需要执行时，创建线程对象。
任务执行完毕，线程对象归还给池子。 → submit方法

池子会自动的帮我们创建对象，
任务执行完毕，也会自动把线程对象归还池子。

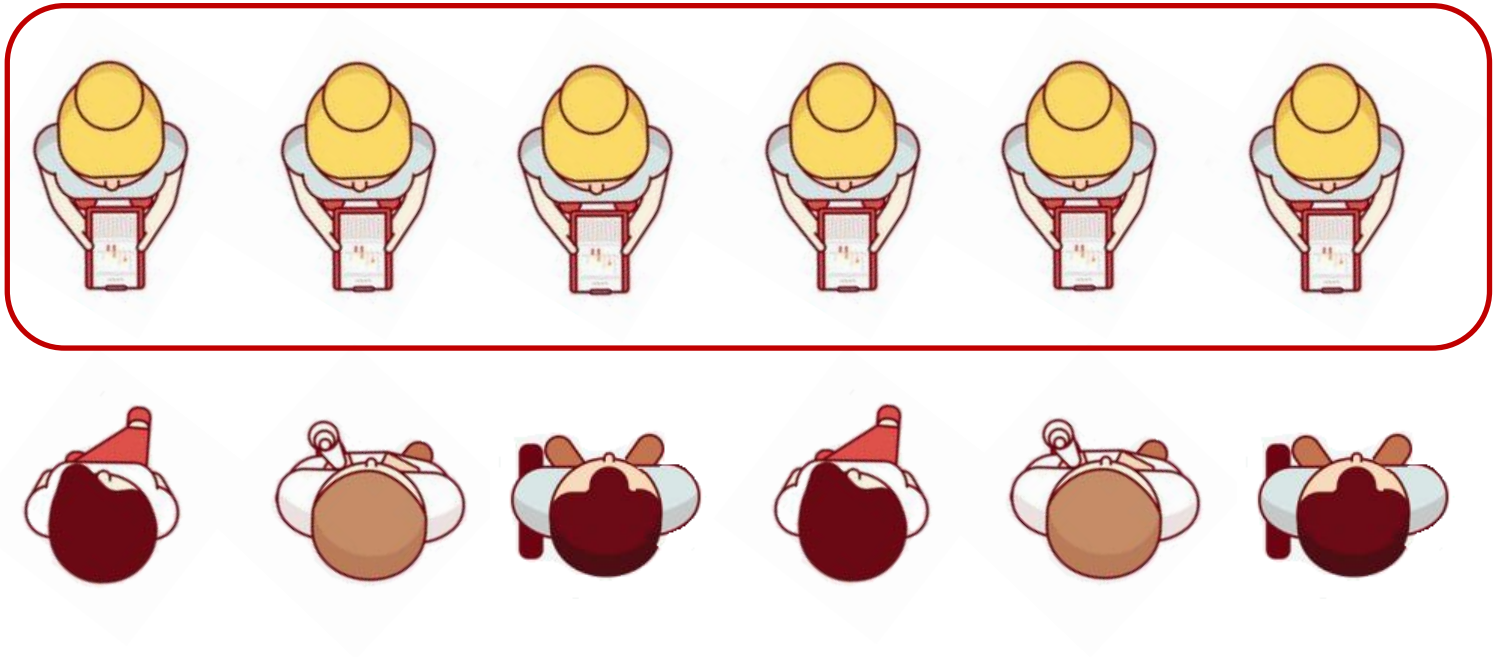
3，所有的任务全部执行完毕，关闭连接池 → shutdown方法

线程池

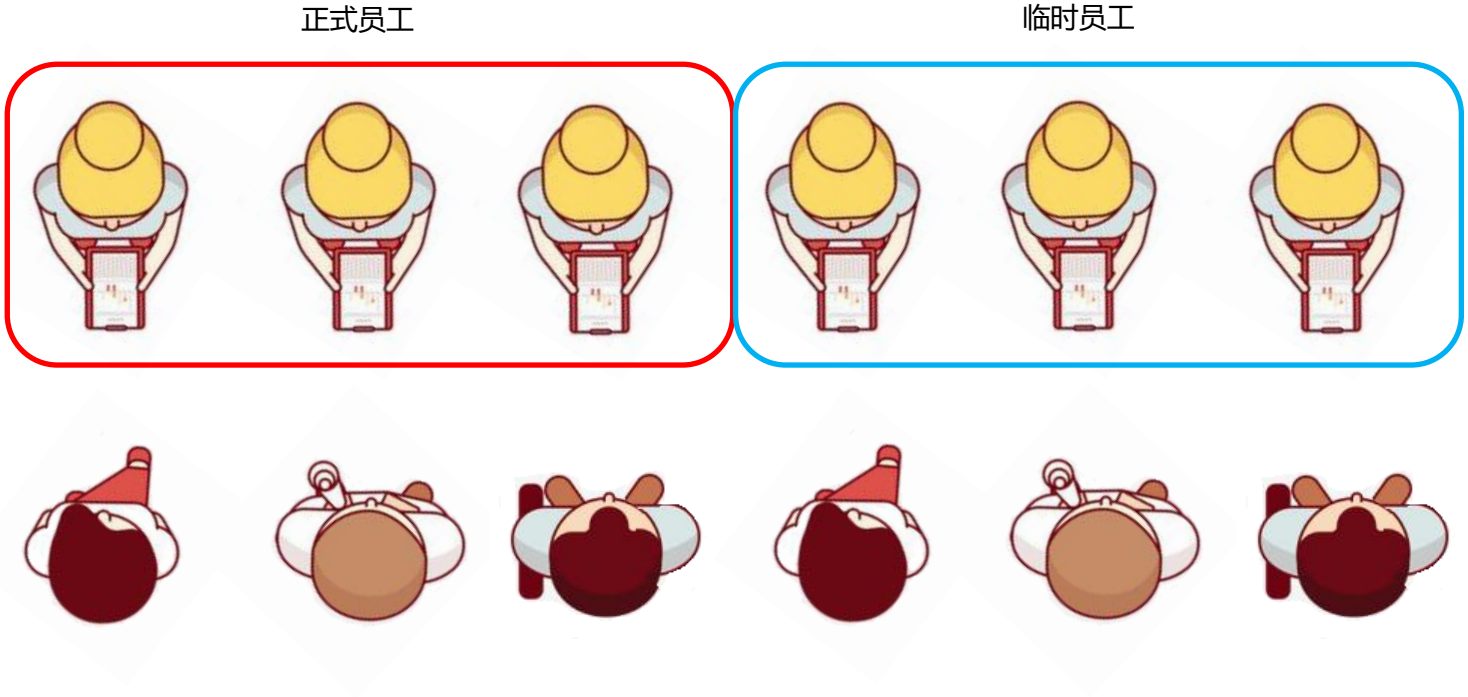


线程池

饭馆能容纳最多的服务员



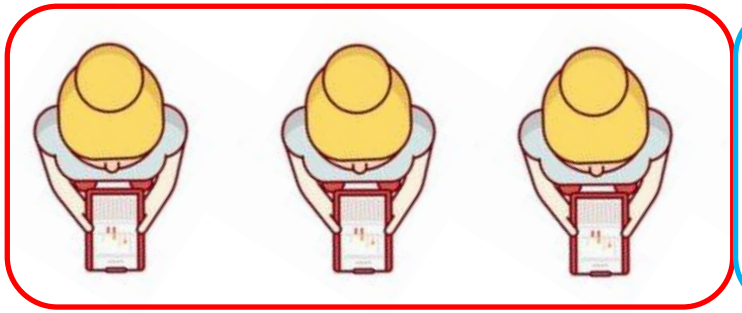
线程池



线程池

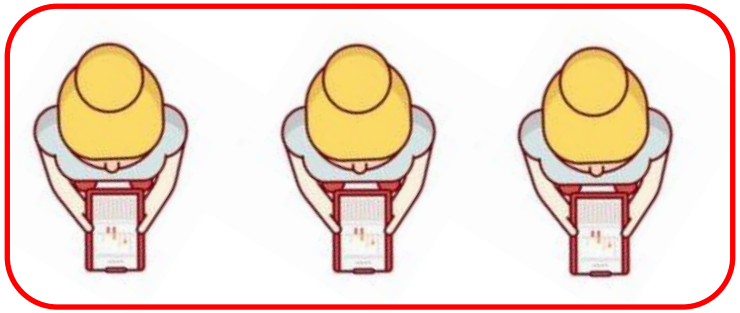
正式员工

临时员工



线程池

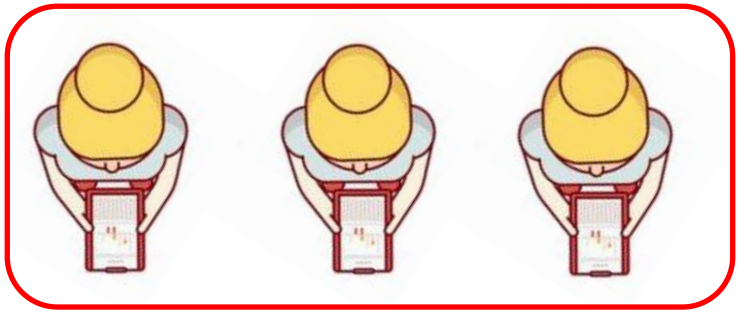
正式员工



当临时员工有一段时间闲着了，为了节约成本，
老板就需要把临时员工给辞掉

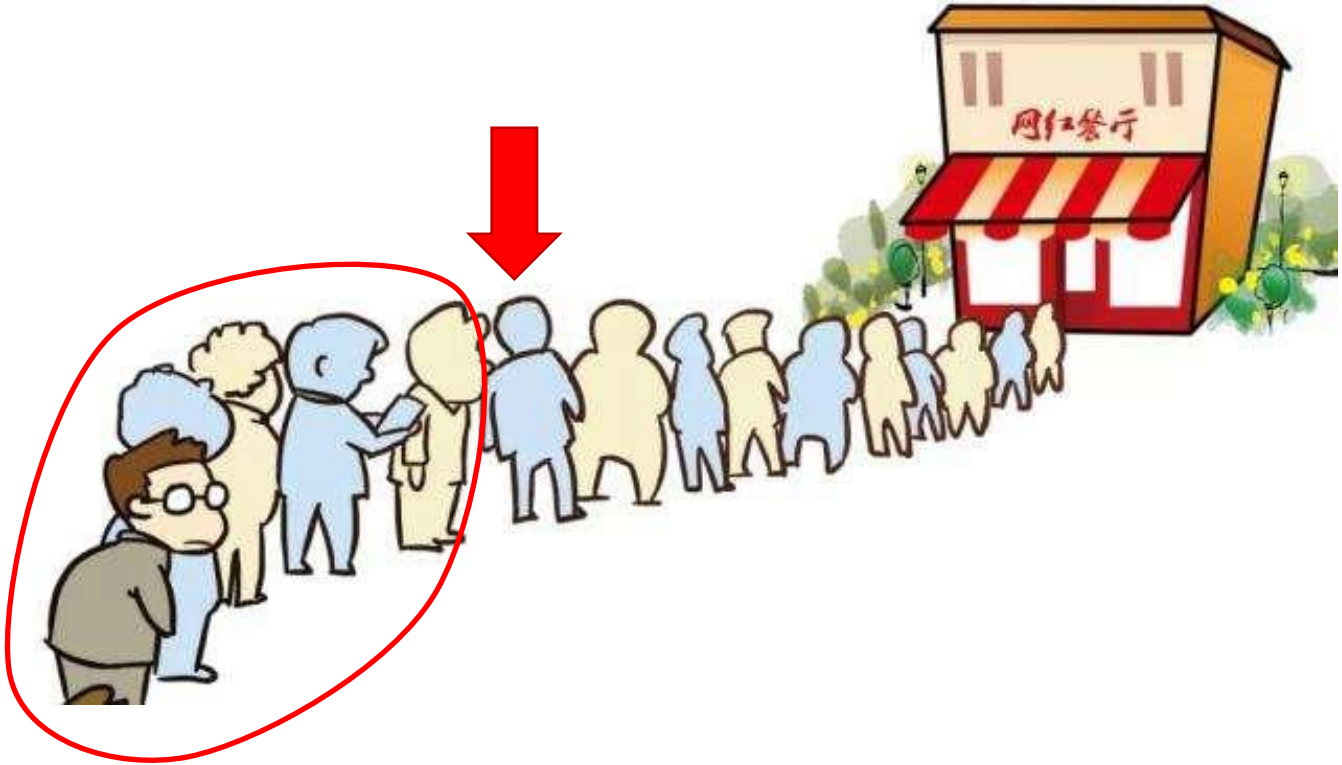
线程池

正式员工



正式员工只要招来了，哪怕没有顾客，都不能辞掉。
除非饭馆倒闭。

线程池



故事中的核心元素

核心元素一：正式员工数量

核心元素二：餐厅最大员工数

核心元素三：临时员工空闲多长时间被辞退（值）

核心元素四：临时员工空闲多长时间被辞退（单位）

核心元素五：排队的客户

核心元素六：从哪里招人

核心元素七：当排队人数过多，超出顾客请下次再来（拒绝服务）

故事中的核心元素

核心元素一：正式员工数量	→	核心线程数量
核心元素二：餐厅最大员工数	→	线程池中最大线程的数量
核心元素三：临时员工空闲多长时间被辞退（值）	→	空闲时间（值）
核心元素四：临时员工空闲多长时间被辞退（单位）	→	空闲时间（单位）
核心元素五：排队的客户	→	阻塞队列
核心元素六：从哪里招人	→	创建线程的方式
核心元素七：当排队人数过多，超出顾客请下次再来（拒绝服务）	→	要执行的任务过多时的解决方案

线程池

创建线程池对象

```
ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor
(核心线程数量,最大线程数量,空闲线程最大存活时间,任务队列,创建线程工厂,任务的拒绝策略);
```

参数一：核心线程数量 → 不能小于0

参数二：最大线程数 → 不能小于等于0，最大数量 >= 核心线程数量

参数三：空闲线程最大存活时间 → 不能小于0

参数四：时间单位 → 时间单位

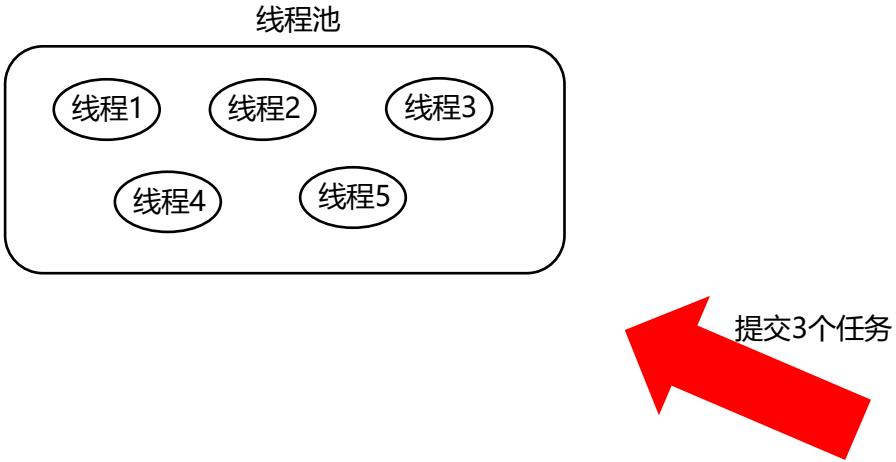
参数五：任务队列 → 不能为null

参数六：创建线程工厂 → 不能为null

参数七：任务的拒绝策略 → 不能为null

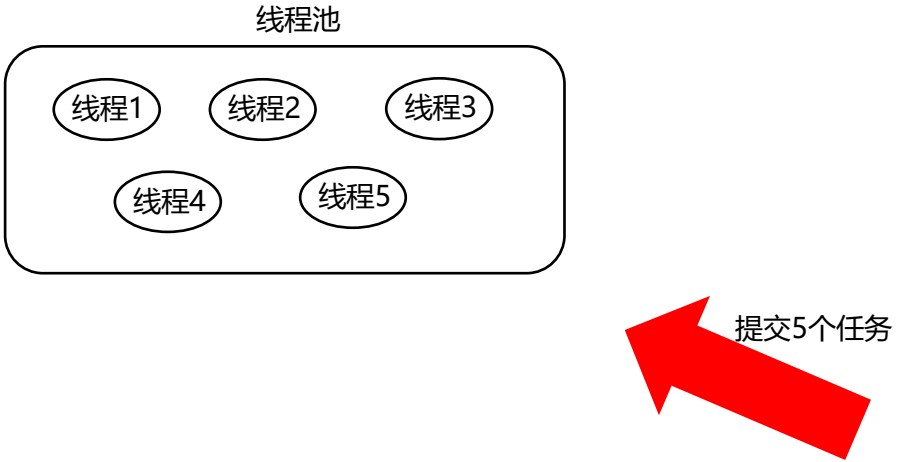
线程池

任务拒绝策略



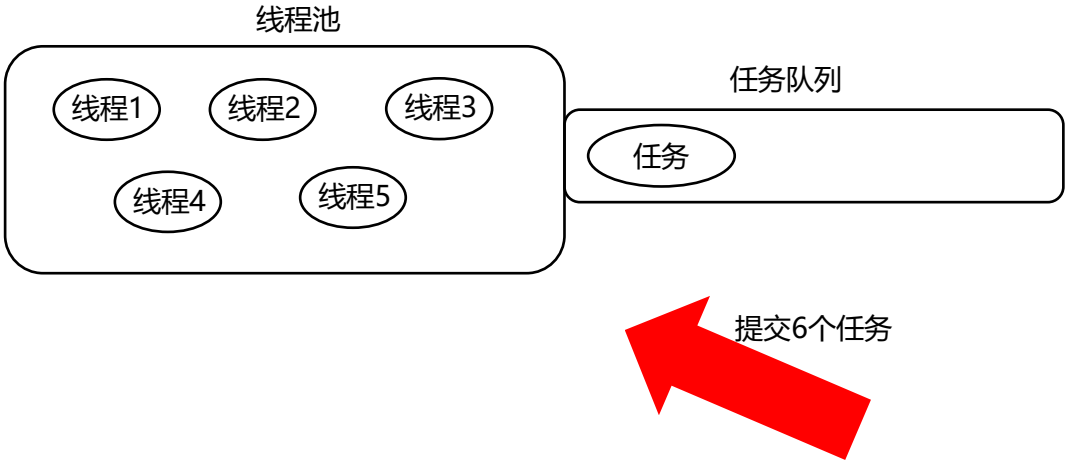
线程池

任务拒绝策略



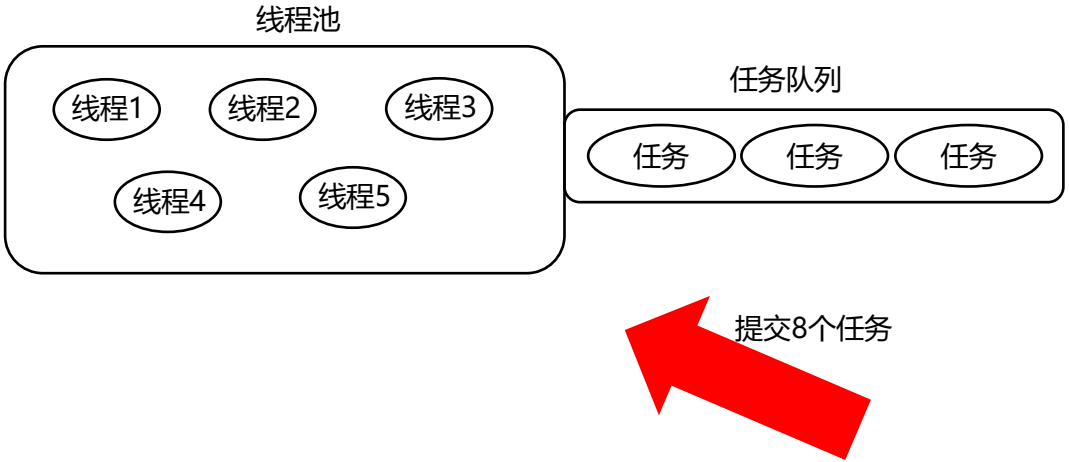
线程池

任务拒绝策略



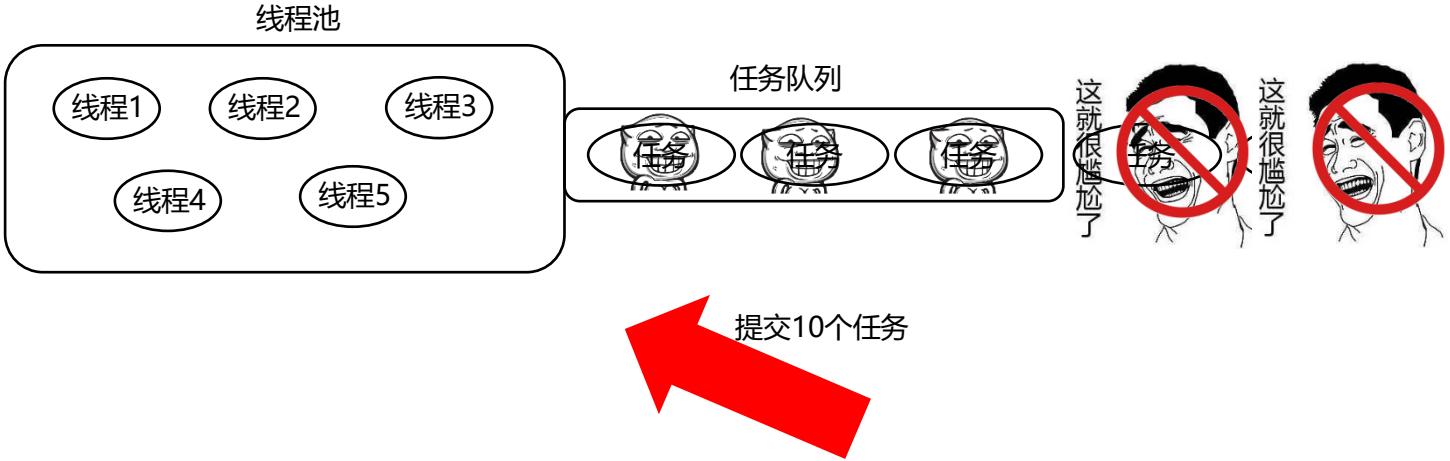
线程池

任务拒绝策略



线程池

任务拒绝策略



任务拒绝策略

ThreadPoolExecutor.AbortPolicy:	丢弃任务并抛出RejectedExecutionException异常。**是默认的策略。**
ThreadPoolExecutor.DiscardPolicy：	丢弃任务，但是不抛出异常 这是不推荐的做法。
ThreadPoolExecutor.DiscardOldestPolicy：	抛弃队列中等待最久的任务 然后把当前任务加入队列中。
ThreadPoolExecutor.CallerRunsPolicy:	调用任务的run()方法绕过线程池直接执行。

目录 Contents

TCP通信程序

结婚基金的故事

小皮同学

存在银行的结婚基金

小路同学

TCP通信程序

结婚基金的故事

TCP通信程序

结婚基金的故事

TCP通信程序

结婚基金的故事

取一万充值到游戏

TCP通信程序

结婚基金的故事

故事的问题

女孩虽然知道结婚基金是十万，但是当基金的余额发生变化的时候，女孩无法知道最新的余额。

TCP通信程序

多线程中的问题

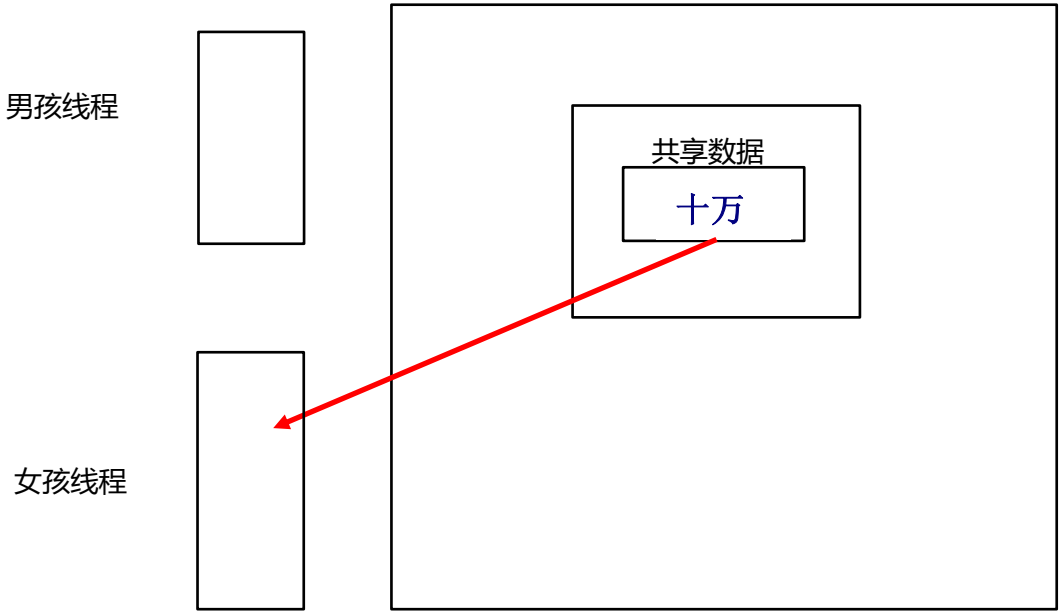
共享数据

A线程

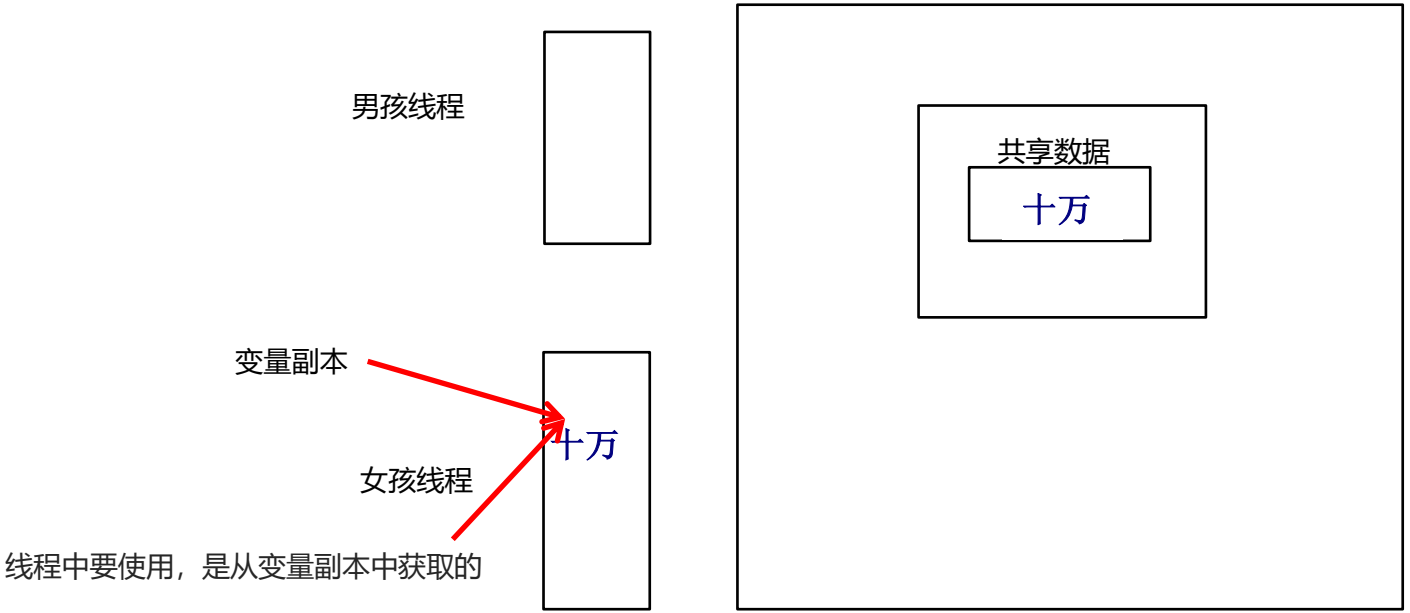
B线程

当A线程修改了共享数据时，
B线程没有及时获取到最新的值，
如果还在使用原先的值，就会出现问题

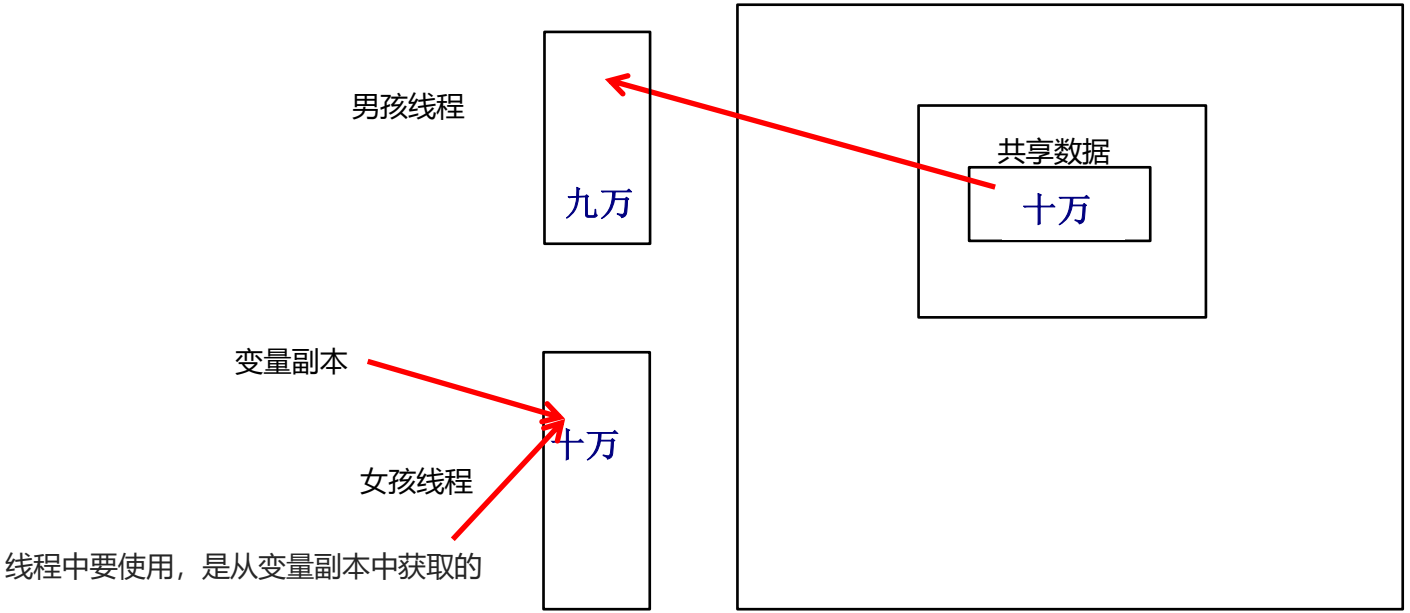
JMM



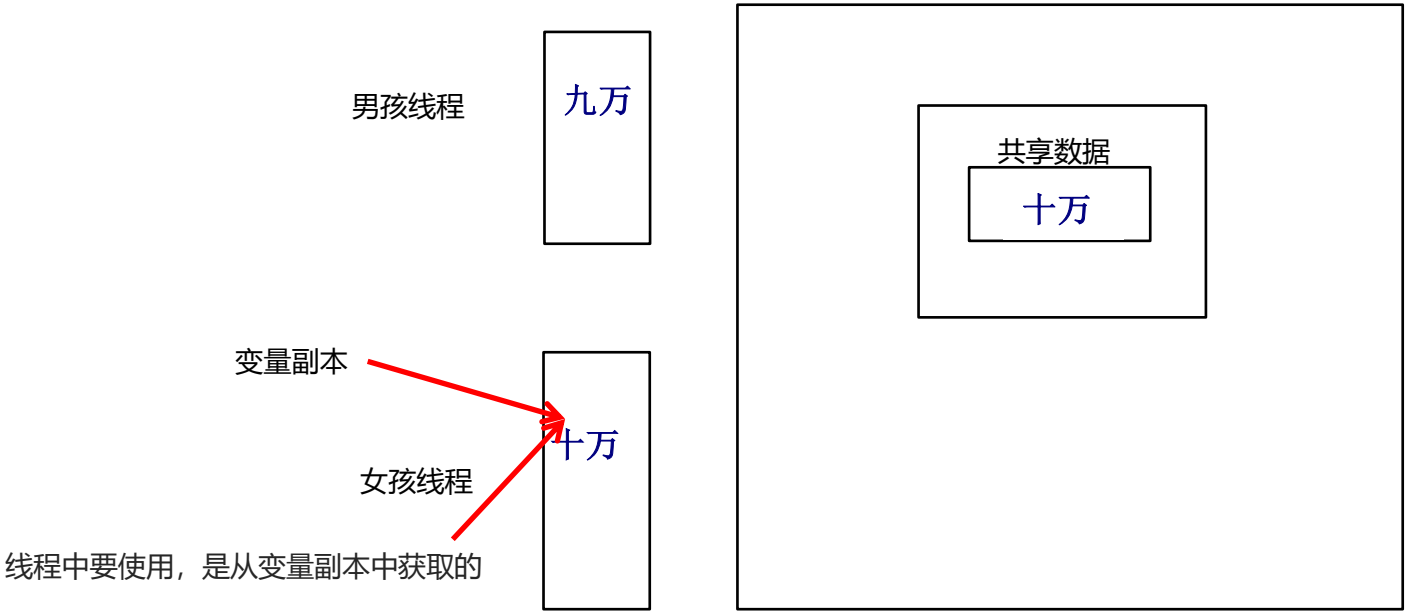
JMM



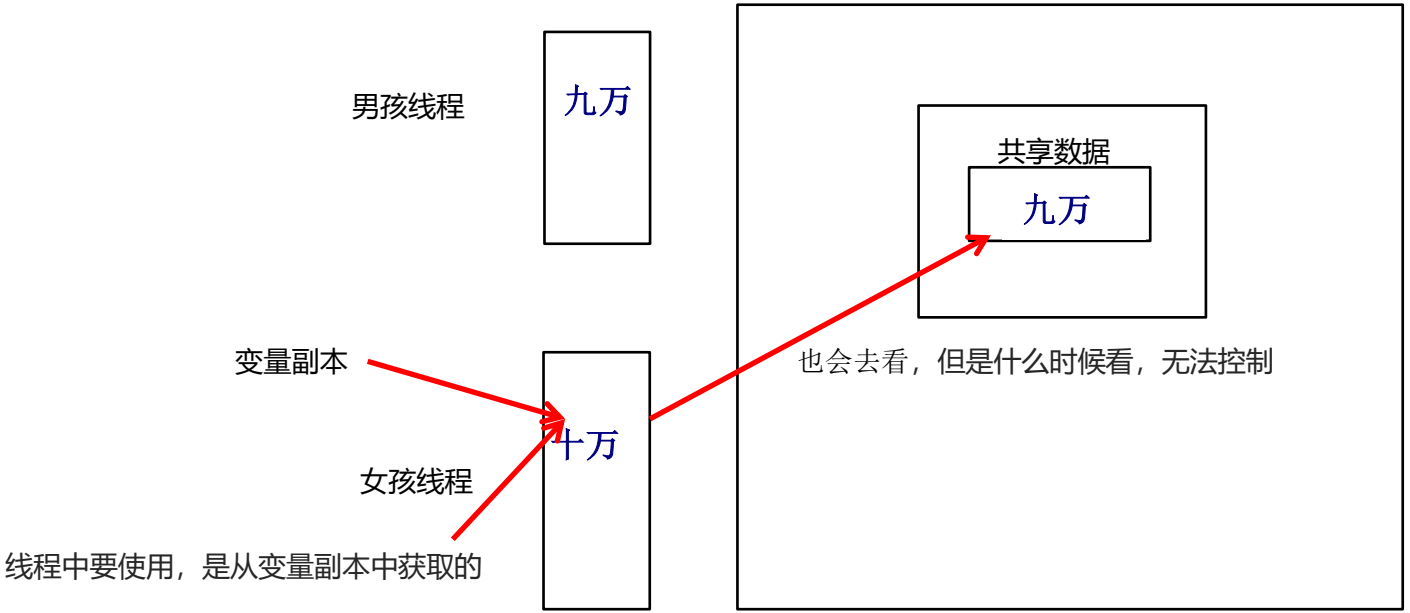
JMM



JMM



JMM



小结

1，堆内存是唯一的，每一个线程都有自己的线程栈。

2，每一个线程在使用堆里面变量的时候，都会先拷贝一份到变量的副本中。

3，在线程中，每一次使用是从变量的副本中获取的。

问题

如果A线程修改了堆中共享变量的值。

那么其他线程不一定能及时使用最新的值。

Volatile关键字

强制线程每次在使用的时候，都会看一下共享区域最新的值

Synchronized同步代码块

1，线程获得锁

2，清空变量副本

3，拷贝共享变量最新的值到变量副本中

4，执行代码

5，将修改后变量副本中的值赋值给共享数据

6，释放锁

目录 Contents

原子性

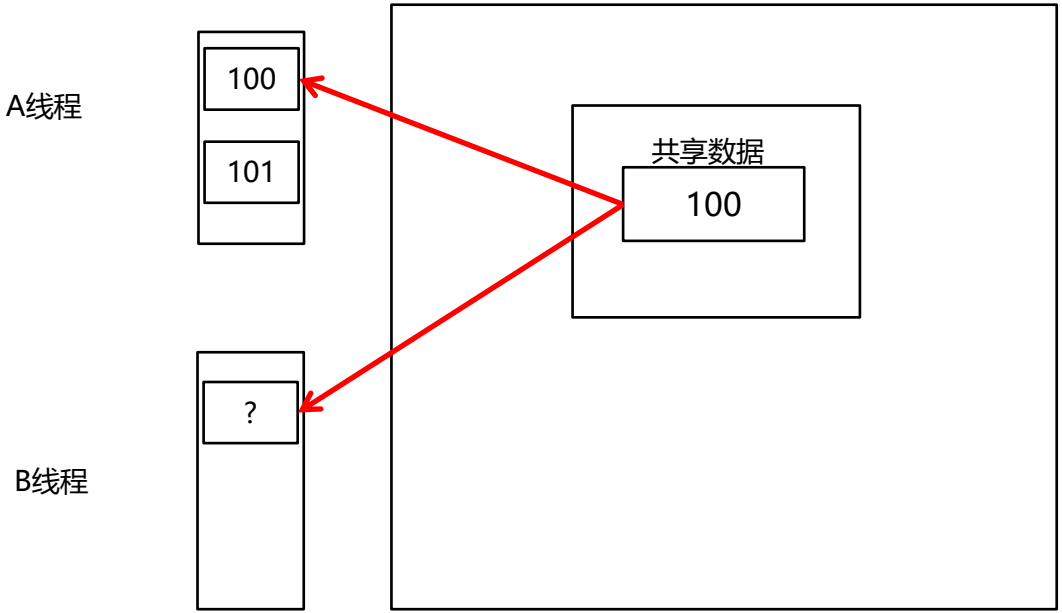
原子性

所谓的原子性是指在一次操作或者多次操作中，要么所有的操作全部都得到了执行并且不会受到任何因素的干扰而中断，要么所有的操作都不执行，**多个操作是一个不可以分割的整体。**

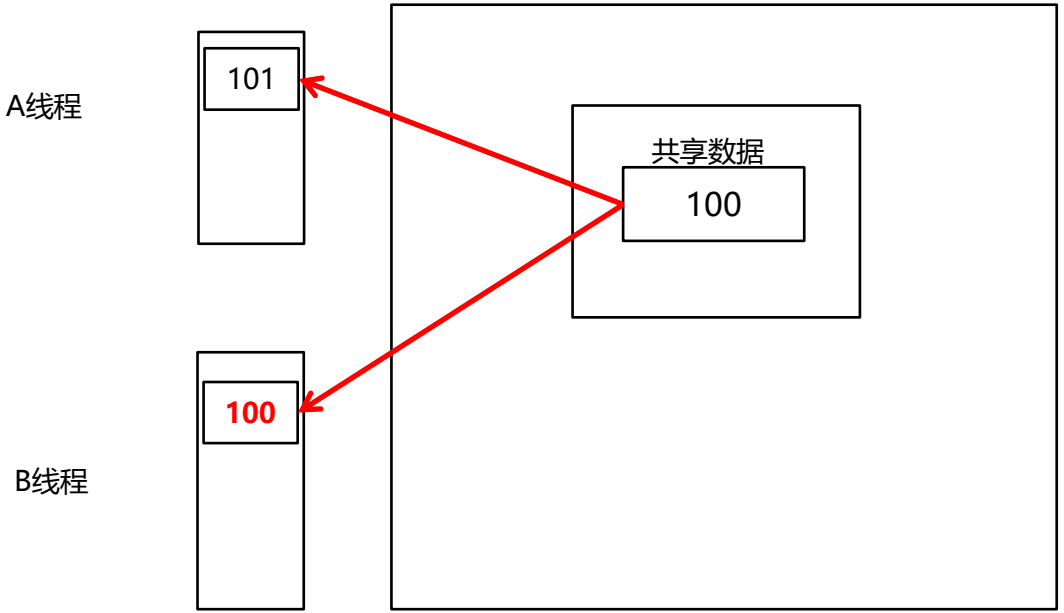
再比如：

小贾同学要给小皮同学汇款1000元。

JMM



JMM



JMM

volatile关键字：

只能保证线程每次在使用共享数据的时候是最新值。

但是不能保证原子性。

原子类AtomicInteger

public AtomicInteger():	初始化一个默认值为0的原子型Integer
public AtomicInteger(int initialValue):	初始化一个指定值的原子型Integer
int get():	获取值
int getAndIncrement():	以原子方式将当前值加1，注意，这里返回的是自增前的值。
int incrementAndGet():	以原子方式将当前值加1，注意，这里返回的是自增后的值。
int addAndGet(int data):	以原子方式将输入的数值与实例中的值（AtomicInteger里的value）相加，并返回结果。
int getAndSet(int value):	以原子方式设置为newValue的值，并返回旧值。

AtomicInteger原理

自旋锁 + CAS 算法

CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）

当旧的预期值A == 内存值 此时修改成功，将V改为B

当旧的预期值A！=内存值 此时修改失败，不做任何操作

并重新获取现在的最新值（这个重新获取的动作就是自旋）

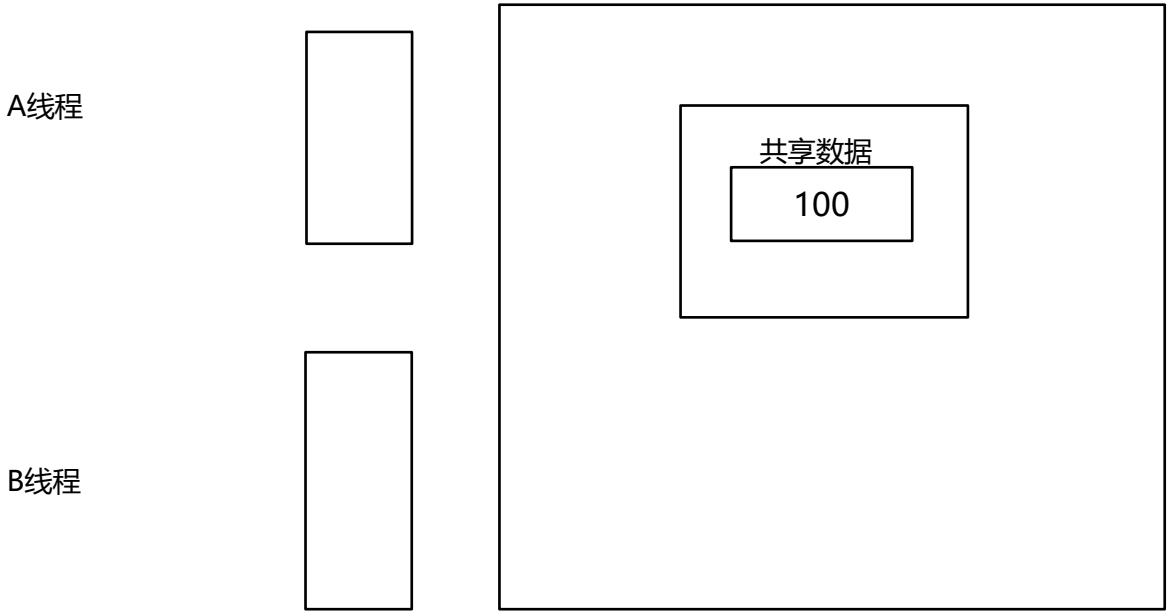
自旋+CAS

CAS算法：有3个操作数（内存值， 旧的预期值A，要修改的值B）

当旧的预期值A == 内存值 此时修改成功，将V改为B

当旧的预期值A！=内存值 此时修改失败，不做任何操作

并重新获取现在的最新值（这个重新获取的动作就是自旋）



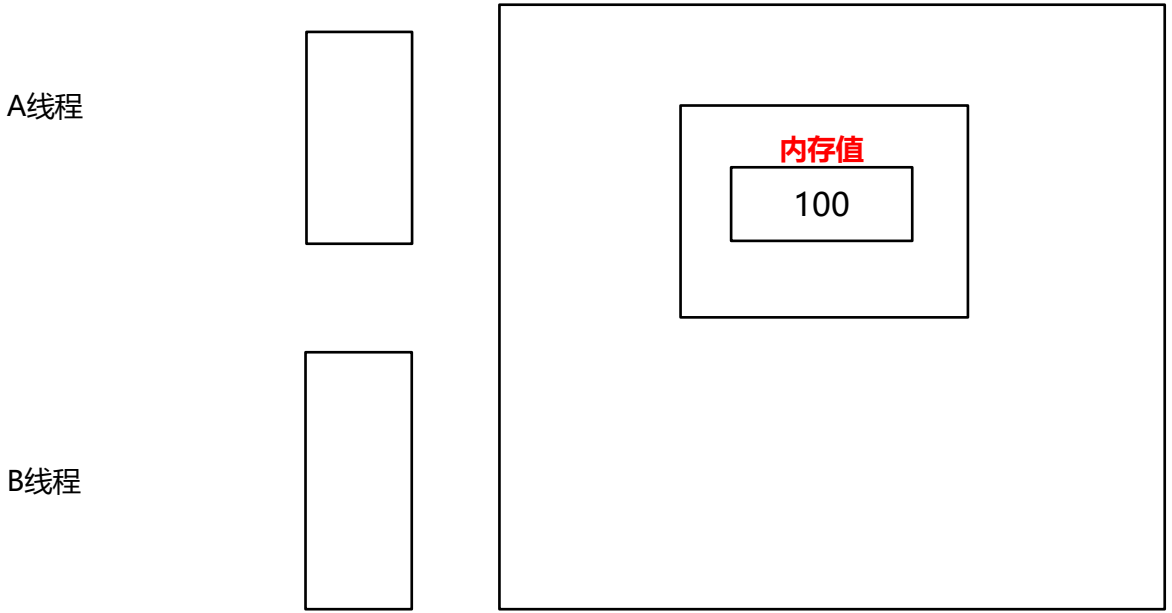
自旋+CAS

CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）

当旧的预期值A == 内存值 此时修改成功，将V改为B

当旧的预期值A！=内存值 此时修改失败，不做任何操作

并重新获取现在的最新值（这个重新获取的动作就是自旋）



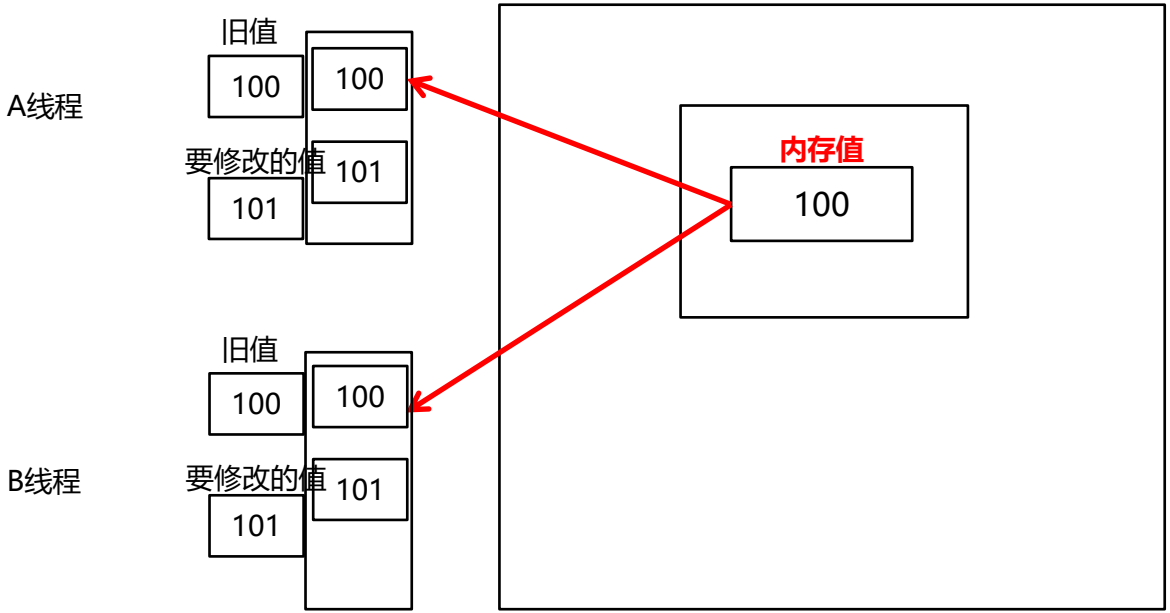
CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）

当旧的预期值A == 内存值 此时修改成功，将V改为B

当旧的预期值A！=内存值 此时修改失败，不做任何操作

并重新获取现在的最新值（这个重新获取的动作就是自旋）

自旋+CAS



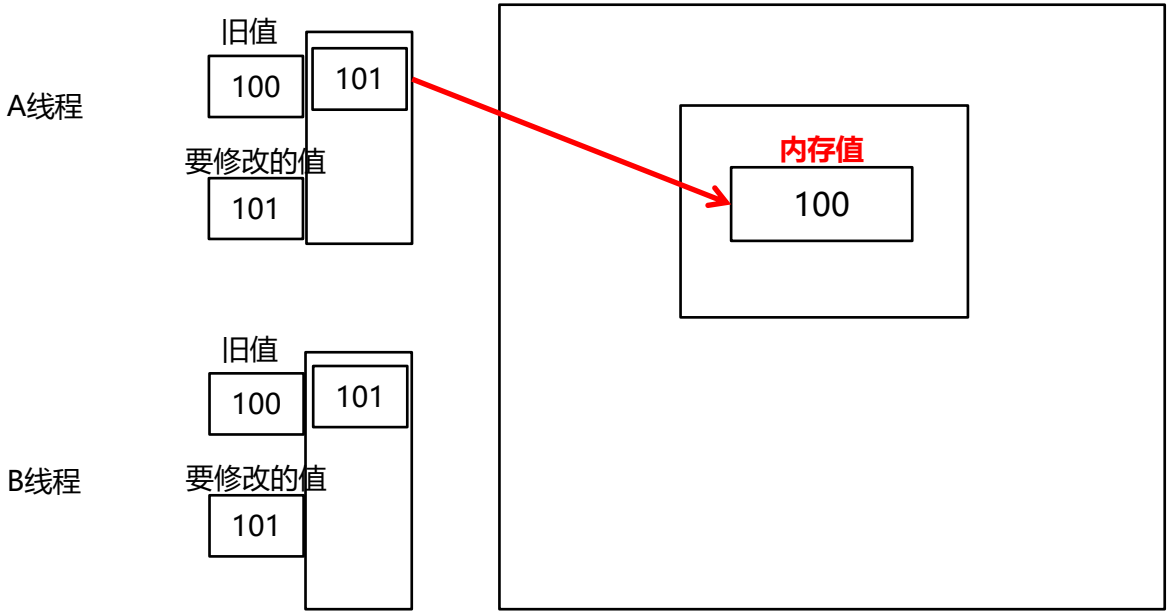
CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）

当旧的预期值A == 内存值 此时修改成功，将V改为B

当旧的预期值A！=内存值 此时修改失败，不做任何操作

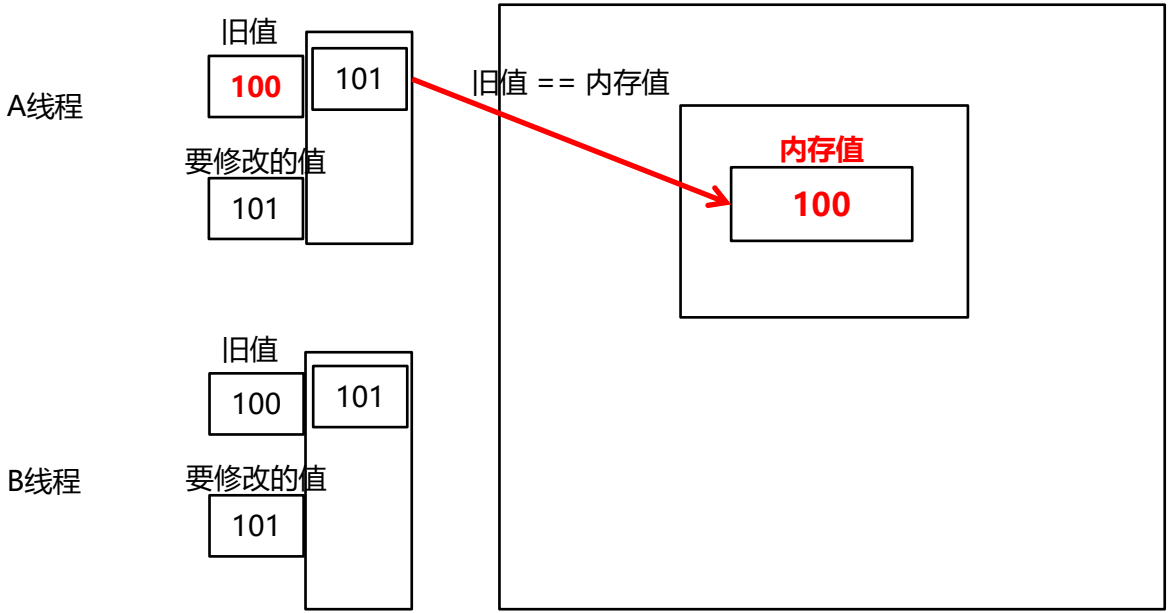
并重新获取现在的最新值（这个重新获取的动作就是自旋）

自旋+CAS



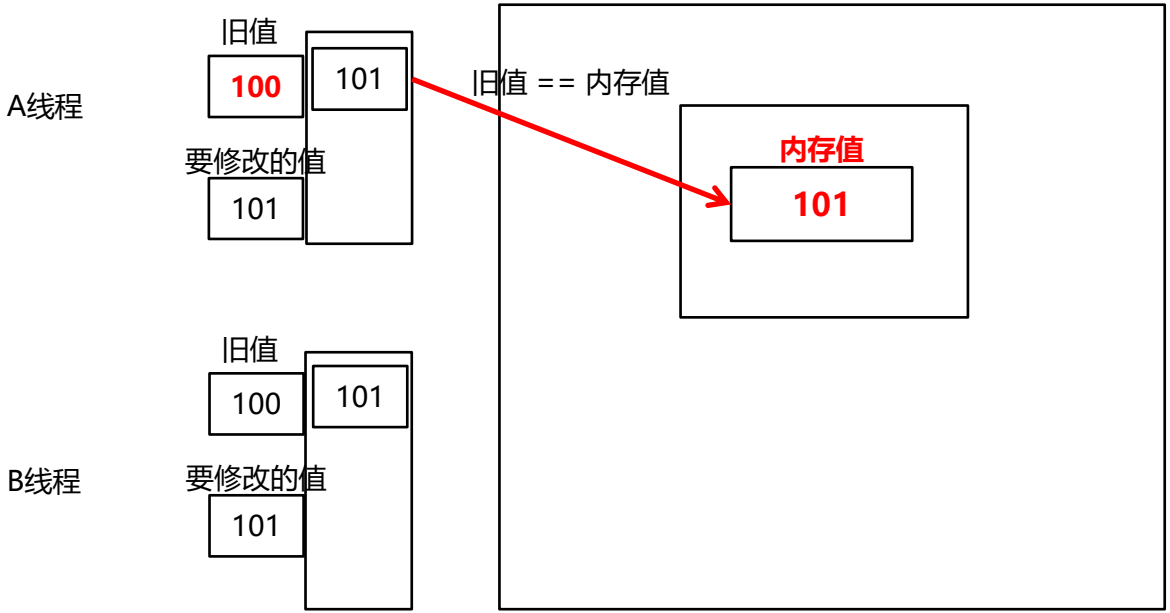
CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）
当旧的预期值A == 内存值 此时修改成功，将V改为B
当旧的预期值A！=内存值 此时修改失败，不做任何操作
并重新获取现在的最新值（这个重新获取的动作就是自旋）

自旋+CAS



CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）
当旧的预期值A == 内存值　此时修改成功，将V改为B
当旧的预期值A！=内存值　此时修改失败，不做任何操作
并重新获取现在的最新值（这个重新获取的动作就是自旋）

自旋+CAS



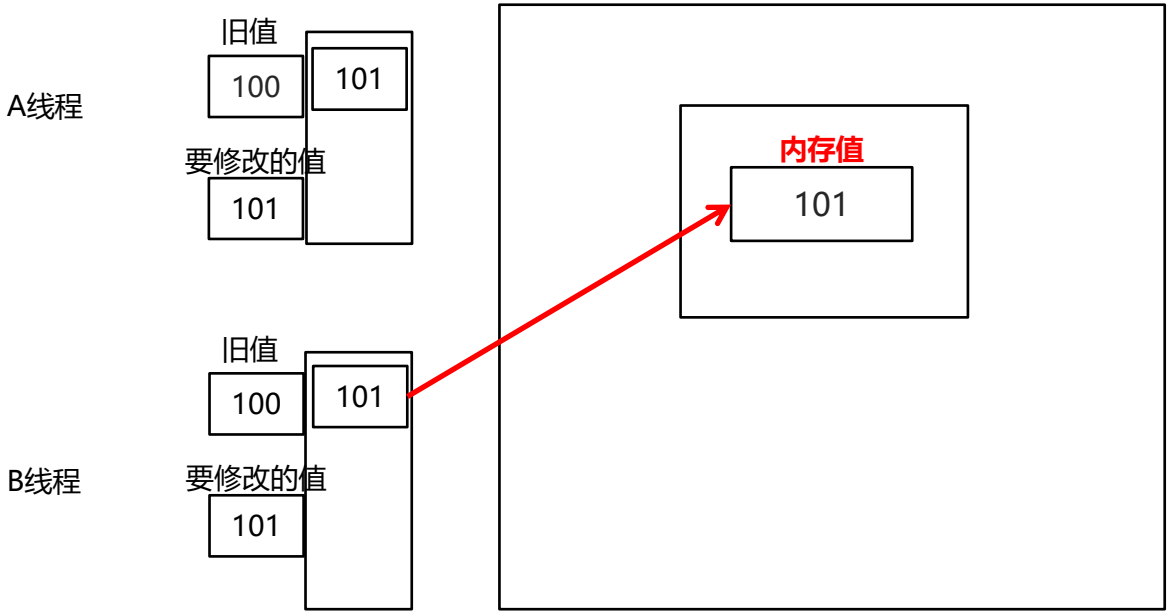
CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）

当旧的预期值A == 内存值 此时修改成功，将V改为B

当旧的预期值A！=内存值 此时修改失败，不做任何操作

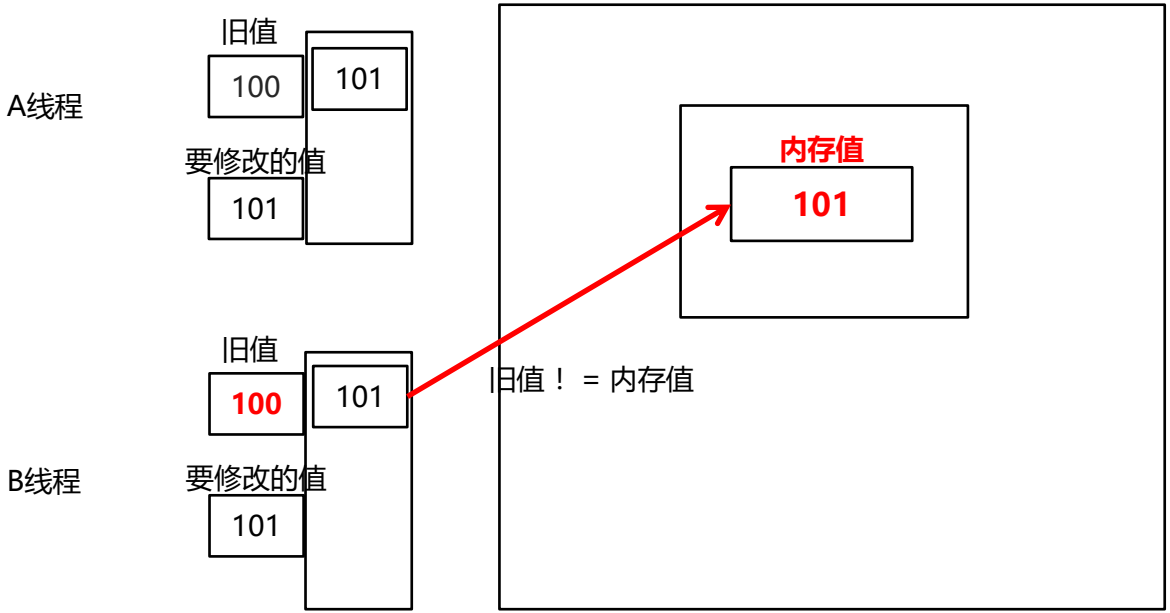
并重新获取现在的最新值（这个重新获取的动作就是自旋）

自旋+CAS



CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）
当旧的预期值A == 内存值 此时修改成功，将V改为B
当旧的预期值A！=内存值 此时修改失败，不做任何操作
并重新获取现在的最新值（这个重新获取的动作就是自旋）

自旋+CAS



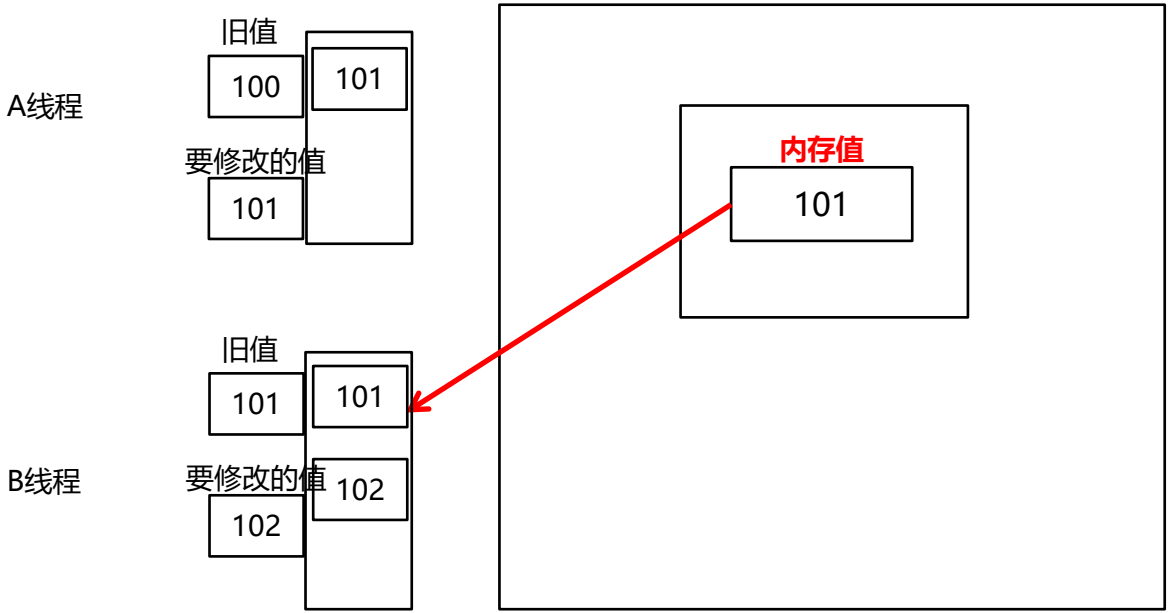
CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）

当旧的预期值A == 内存值 此时修改成功，将V改为B

当旧的预期值A！=内存值 此时修改失败，不做任何操作

并重新获取现在的最新值（这个重新获取的动作就是自旋）

自旋+CAS



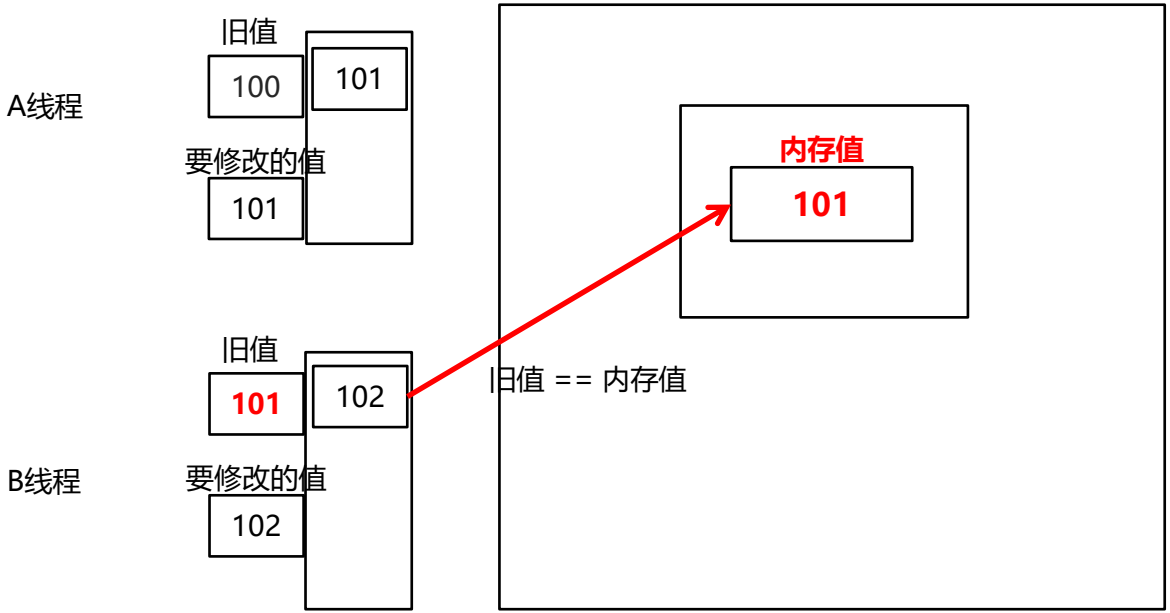
CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）

当旧的预期值A == 内存值 此时修改成功，将V改为B

当旧的预期值A！=内存值 此时修改失败，不做任何操作

并重新获取现在的最新值（这个重新获取的动作就是自旋）

自旋+CAS



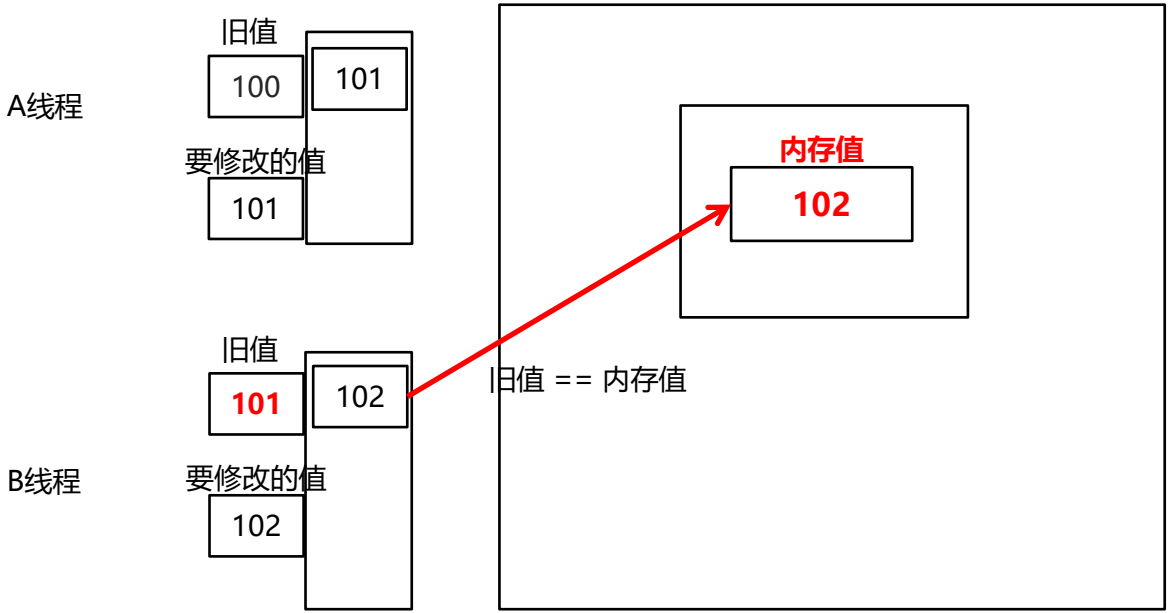
CAS算法：有3个操作数（内存值V， 旧的预期值A，要修改的值B）

当旧的预期值A == 内存值 此时修改成功，将V改为B

当旧的预期值A！=内存值 此时修改失败，不做任何操作

并重新获取现在的最新值（这个重新获取的动作就是自旋）

自旋+CAS



小结

CAS 算法：

在修改共享数据的时候，把原来的旧值记录下来了。
如果现在内存中的值跟原来的旧值一样，证明没有其他线程操作过内存值，则修改成功。

如果现在内存中的值跟原来的旧值不一样了，证明已经有其他线程操作过内存值了。
则修改失败，需要获取现在最新的值，再次进行操作，这个重新获取就是自旋。

synchronized和CAS的区别

相同点：在多线程情况下，都可以保证共享数据的安全性。

不同点：

synchronized总是从最坏的角度出发，认为每次获取数据的时候，别人都有可能修改。所以在每次操作共享数据之前，都会上锁。 **（悲观锁）**

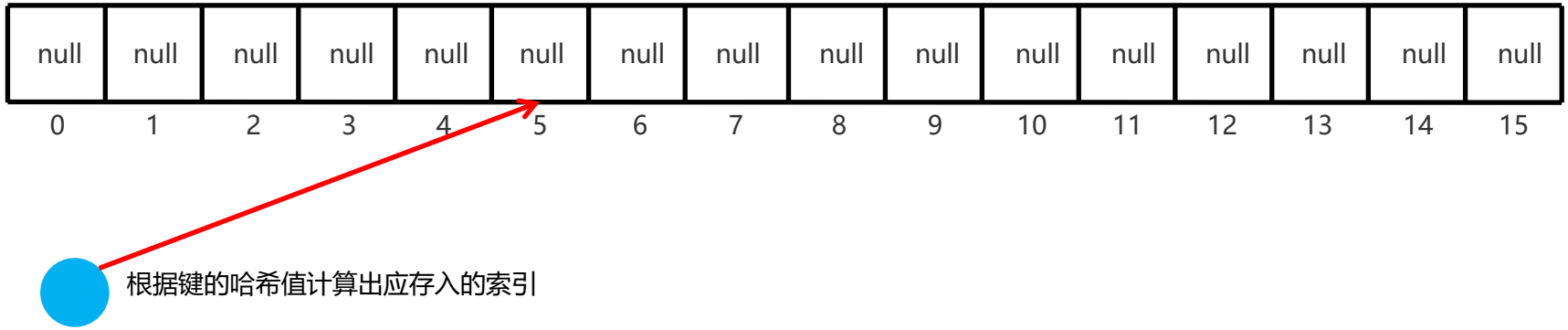
cas是从乐观的角度出发，假设每次获取数据别人都不会修改，所以不会上锁。只不过在修改共享数据的时候，会检查一下，别人有没有修改过这个数据。
如果别人修改过，那么我再次获取现在最新的值。
如果别人没有修改过，那么我现在直接修改共享数据的值。
（乐观锁）

目录 Contents

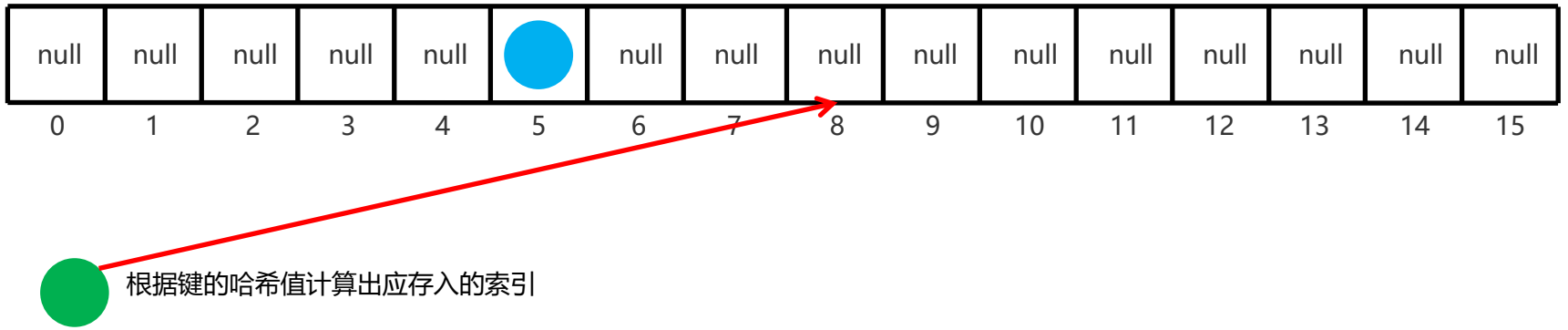
Hashtable

HashMap是线程不安全的(多线程环境下可能会存在问题)。

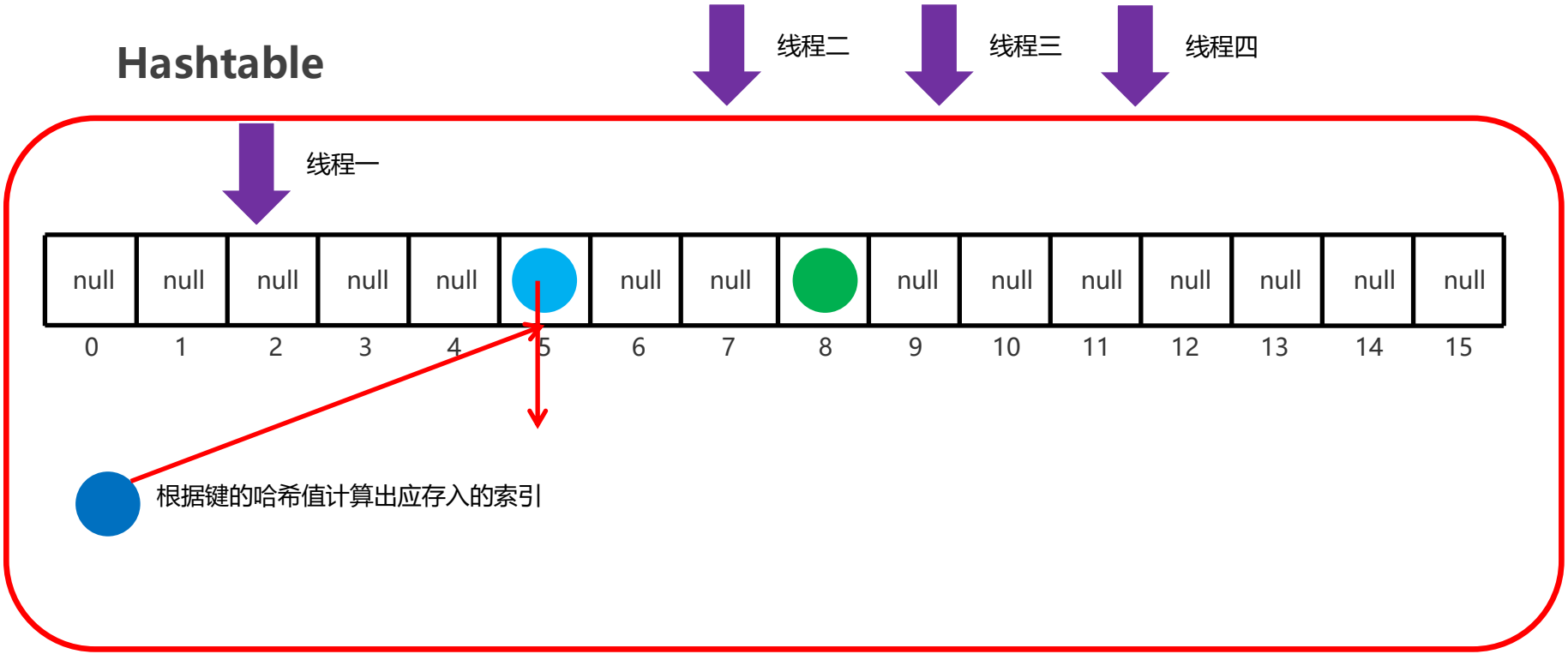
为了保证数据的安全性我们可以使用Hashtable，但是Hashtable的效率低下。



Hashtable



并发工具类



并发工具类

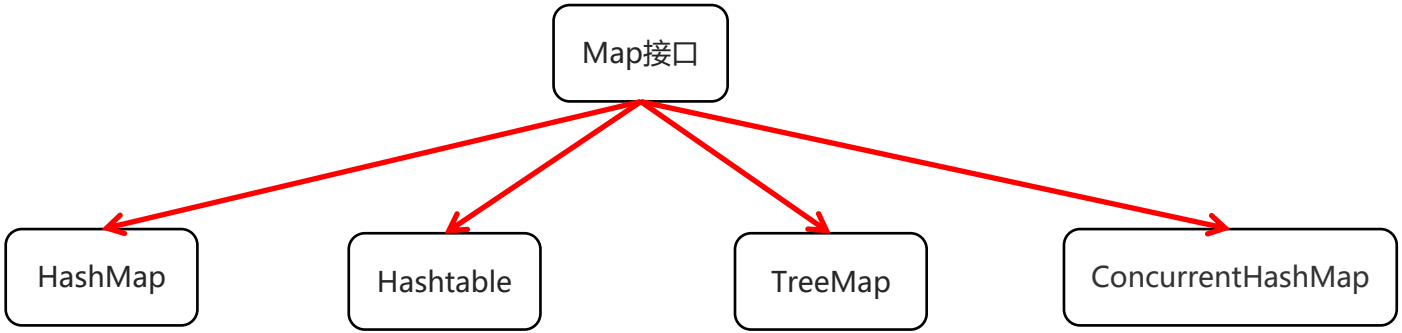
Hashtable

1，Hashtable采取悲观锁synchronized的形式保证数据的安全性

2，只要有线程访问， 会将整张表全部锁起来， 所以Hashtable的效率低下。

并发工具类

ConcurrentHashMap



小结

1，HashMap是线程不安全的。多线程环境下会有数据安全问题

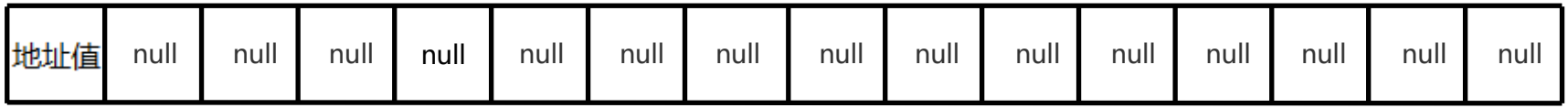
2，Hashtable是线程安全的，但是会将整张表锁起来，效率低下

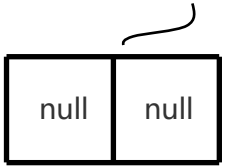
3，ConcurrentHashMap也是线程安全的，效率较高。

在JDK7和JDK8中，底层原理不一样。

ConcurrentHashMap1.7版本原理解析

Segment[]





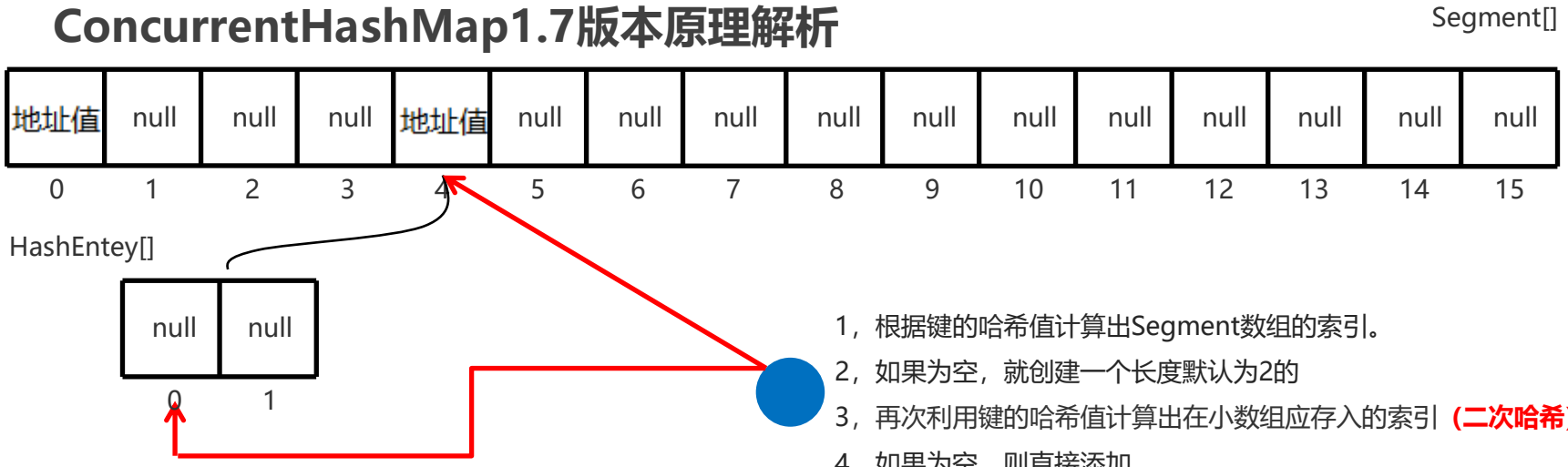

```
ConcurrentHashMap<String, String> hm = new ConcurrentHashMap<>();
```

小结：

1，创建一个默认长度16，默认加载因为0.75的数组，数组名Segment

2，再创建一个长度为2的小数组，把地址值赋值给0索引，其他索引都是null

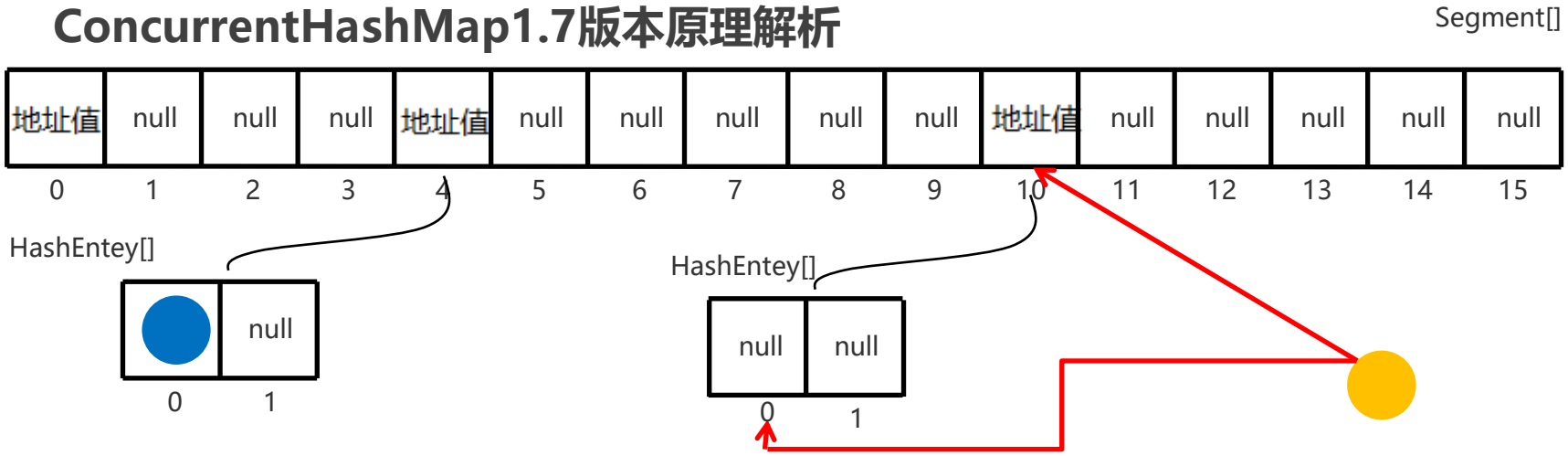
ConcurrentHashMap1.7版本原理解析



1，根据键的哈希值计算出Segment数组的索引。
2，如果为空，就创建一个长度默认为2的
3，再次利用键的哈希值计算出在小数组应存入的索引**（二次哈希）**
4，如果为空，则直接添加。

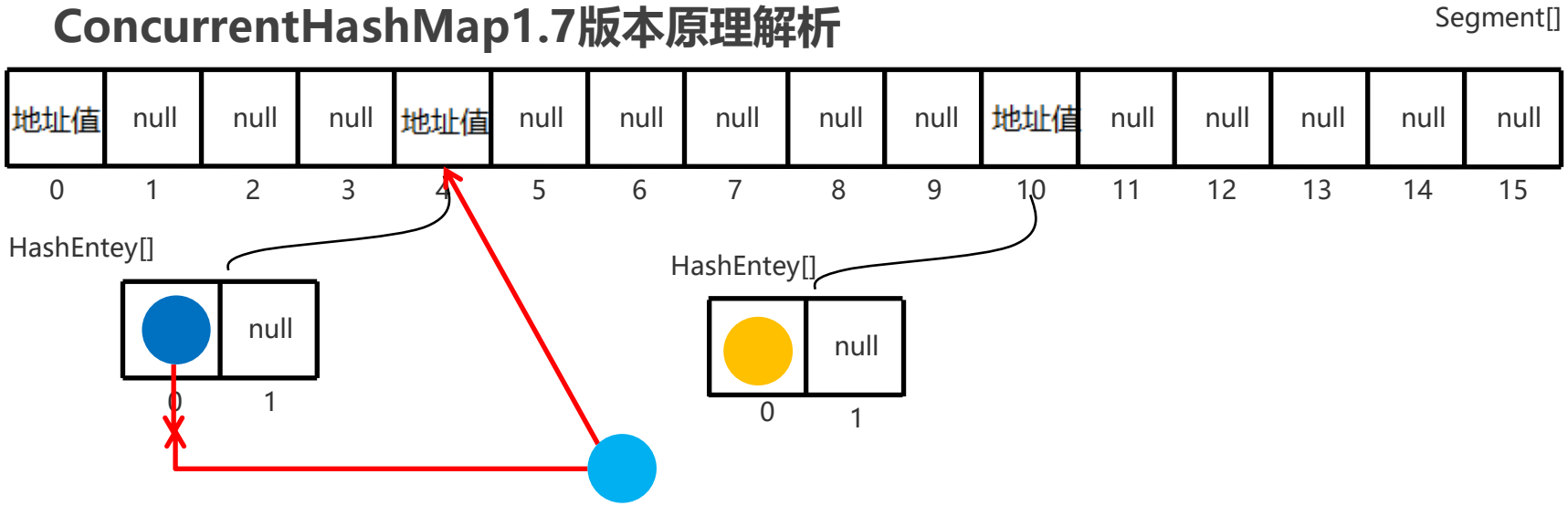
并发工具类

ConcurrentHashMap1.7版本原理解析

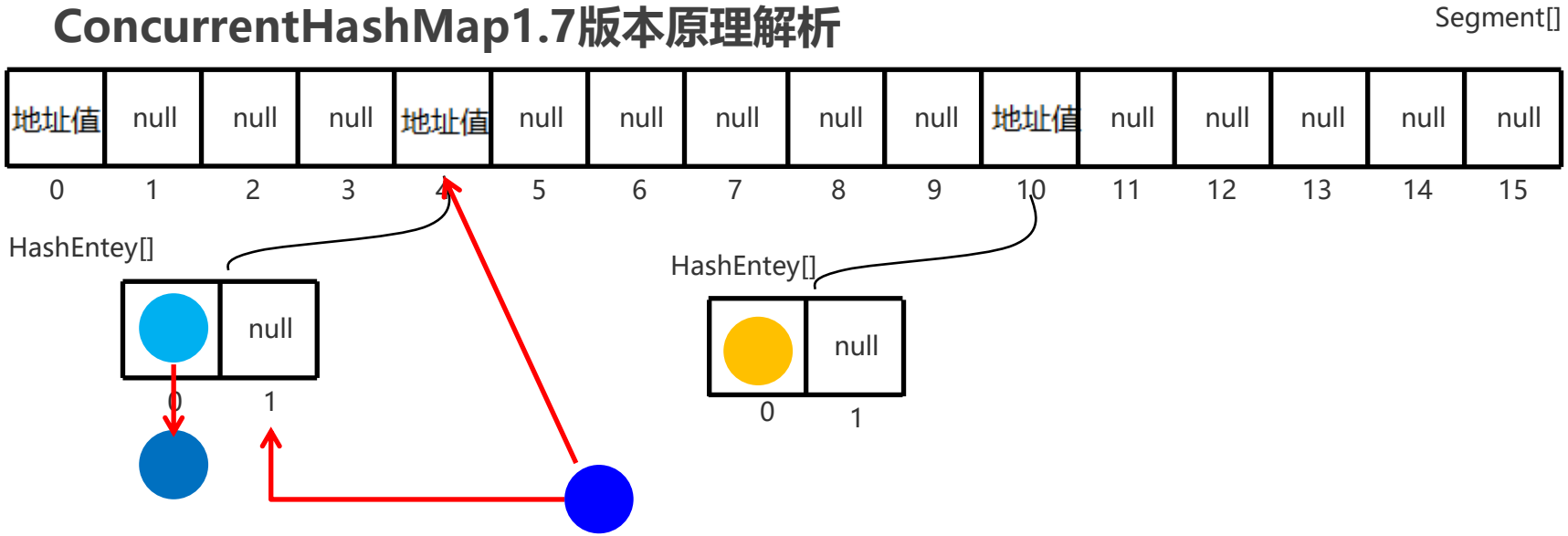


并发工具类

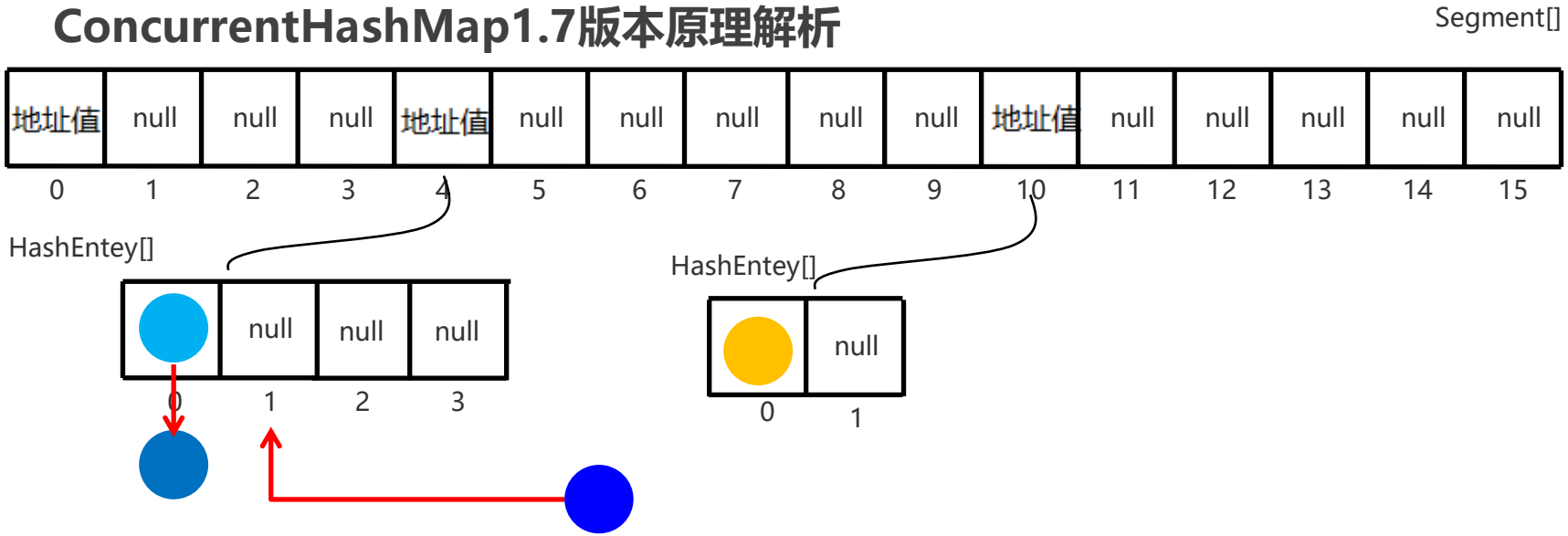
ConcurrentHashMap1.7版本原理解析



ConcurrentHashMap1.7版本原理解析

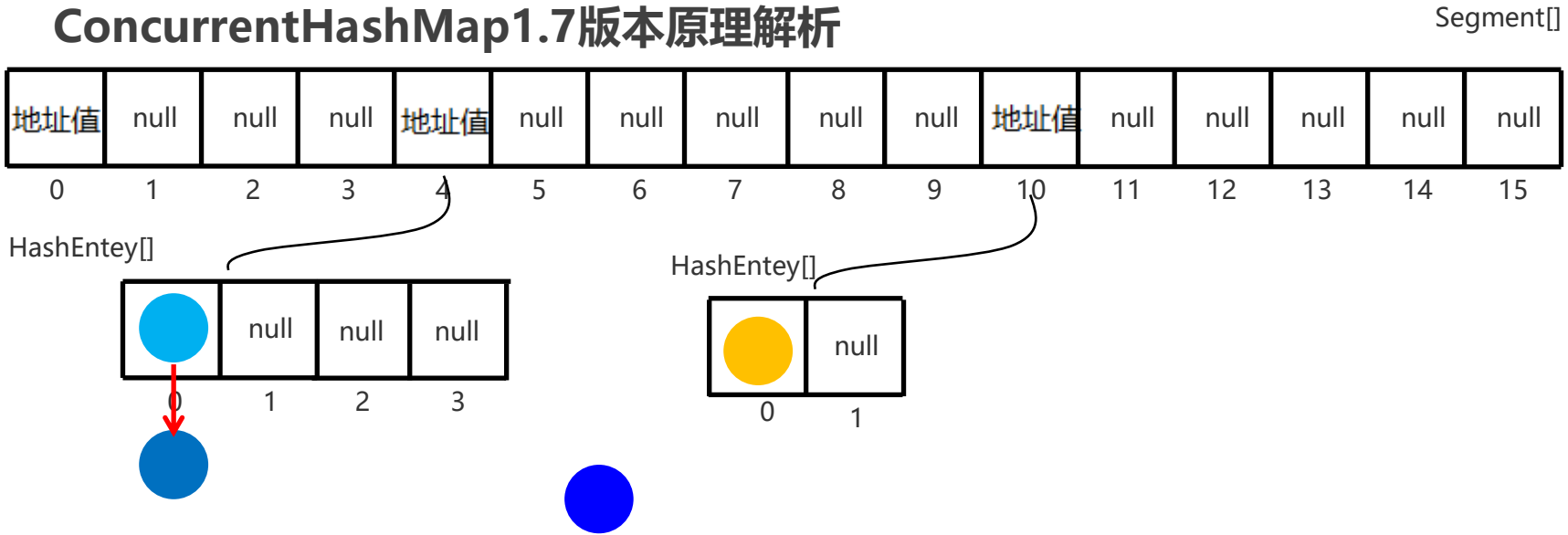


ConcurrentHashMap1.7版本原理解析

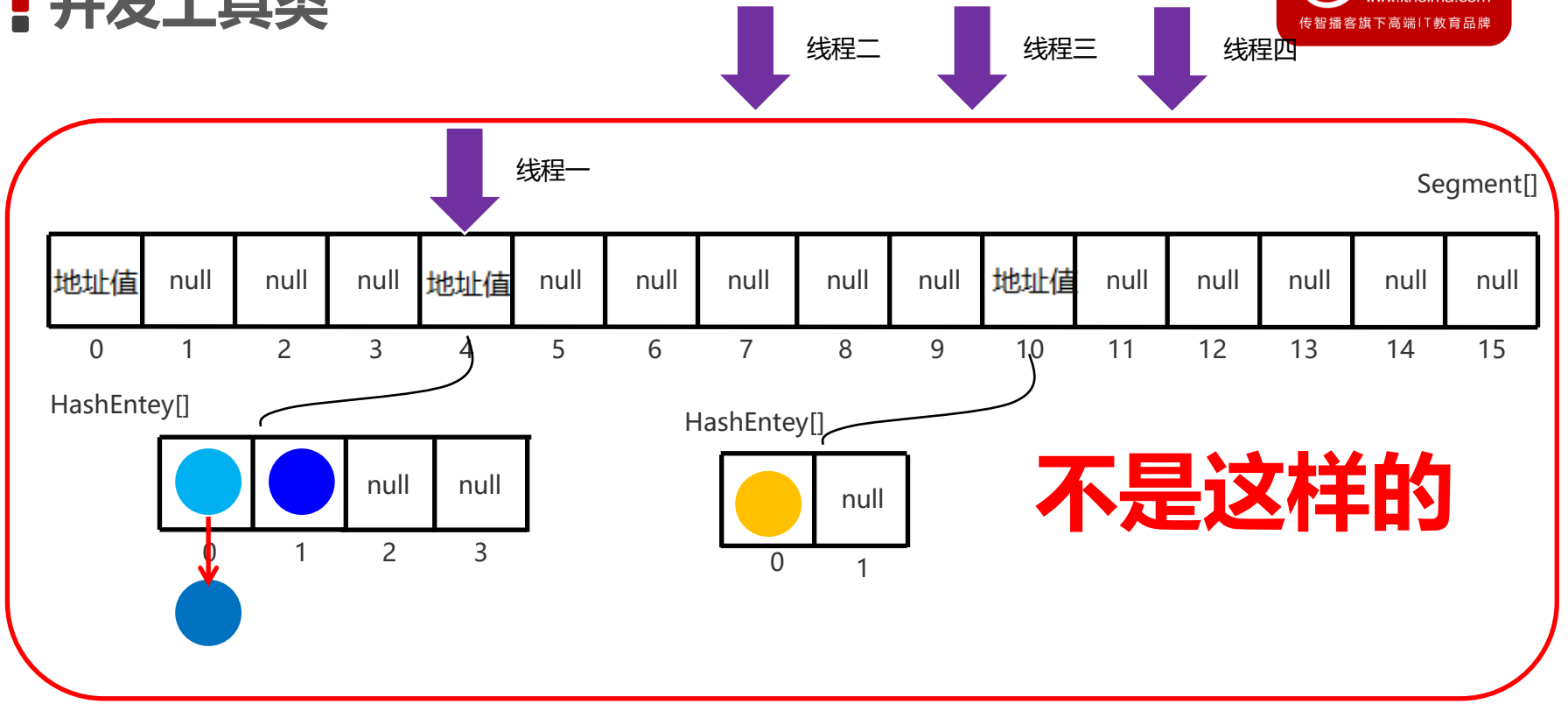


并发工具类

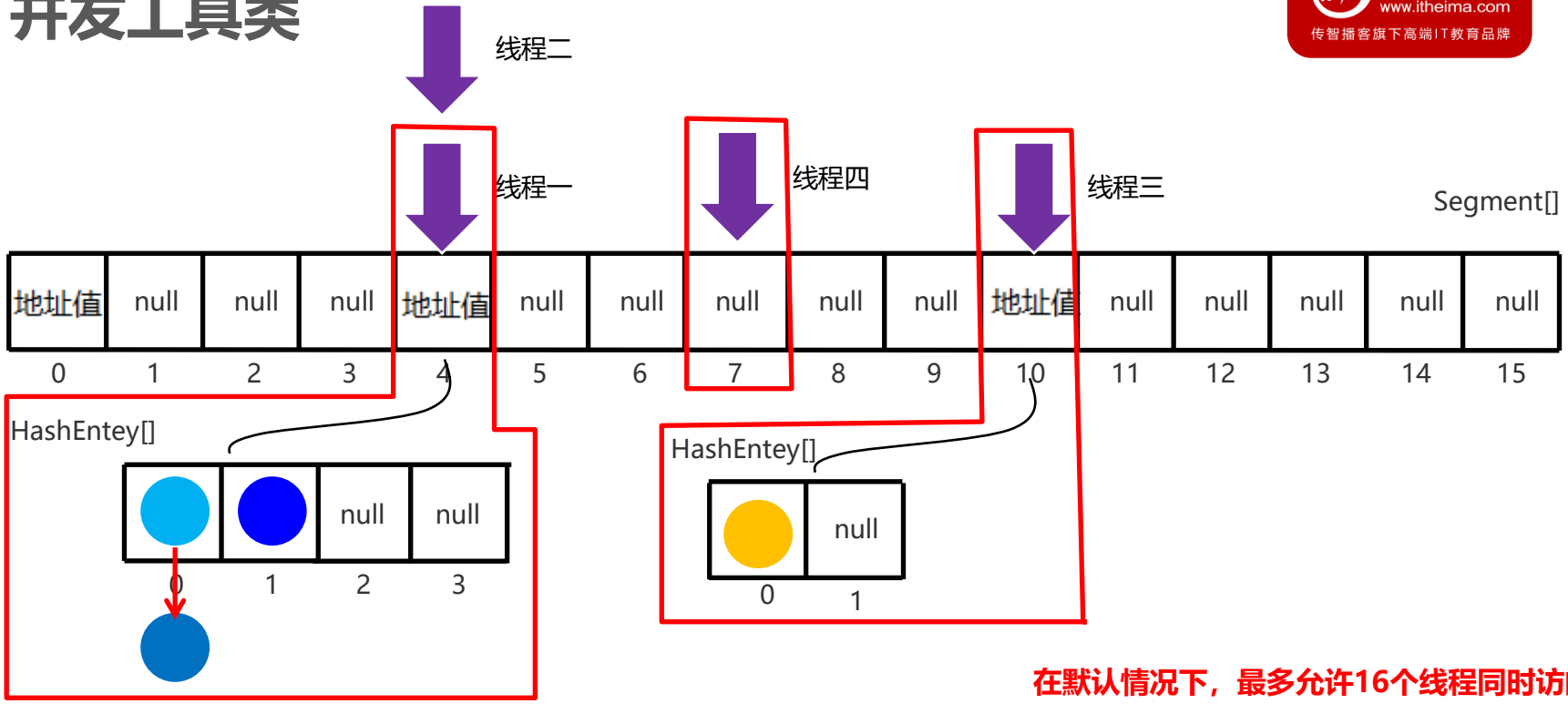
ConcurrentHashMap1.7版本原理解析



并发工具类

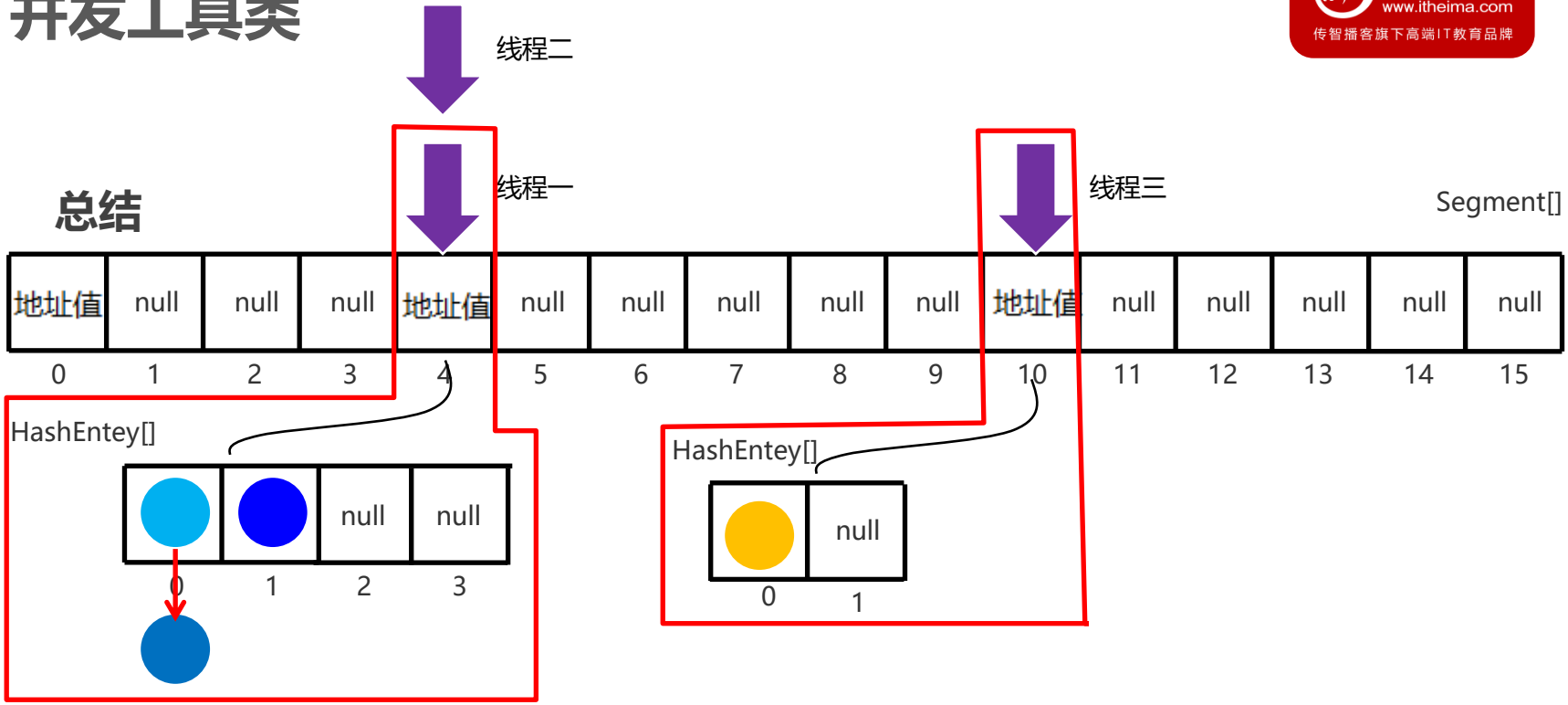


并发工具类



在默认情况下，最多允许16个线程同时访问

并发工具类

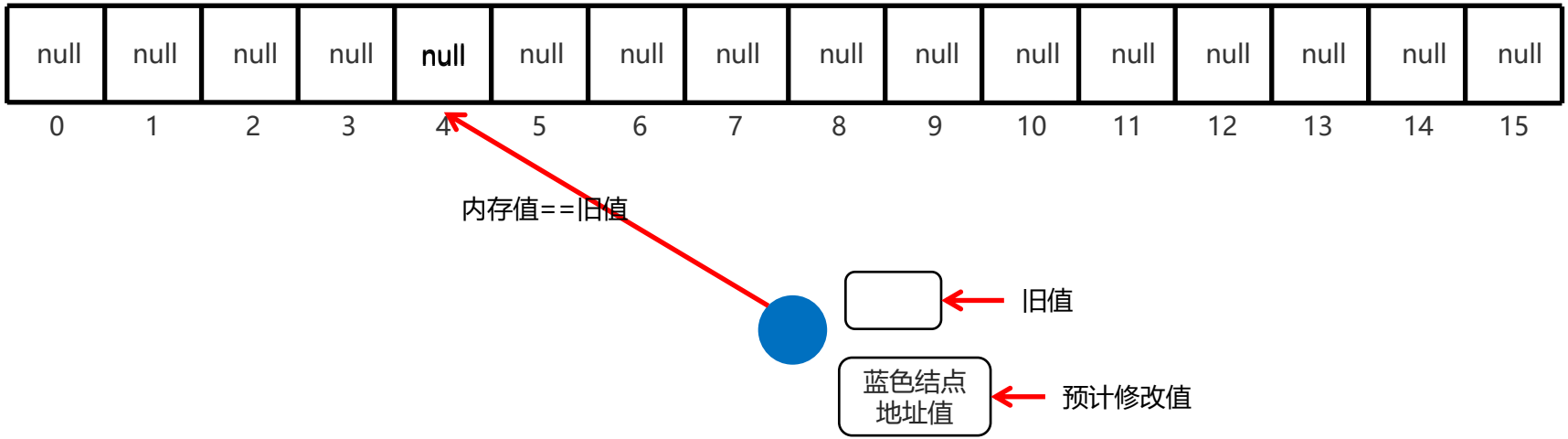


ConcurrentHashMap1.8版本原理解析

底层结构：哈希表。（数组、链表、红黑树的结合体）。

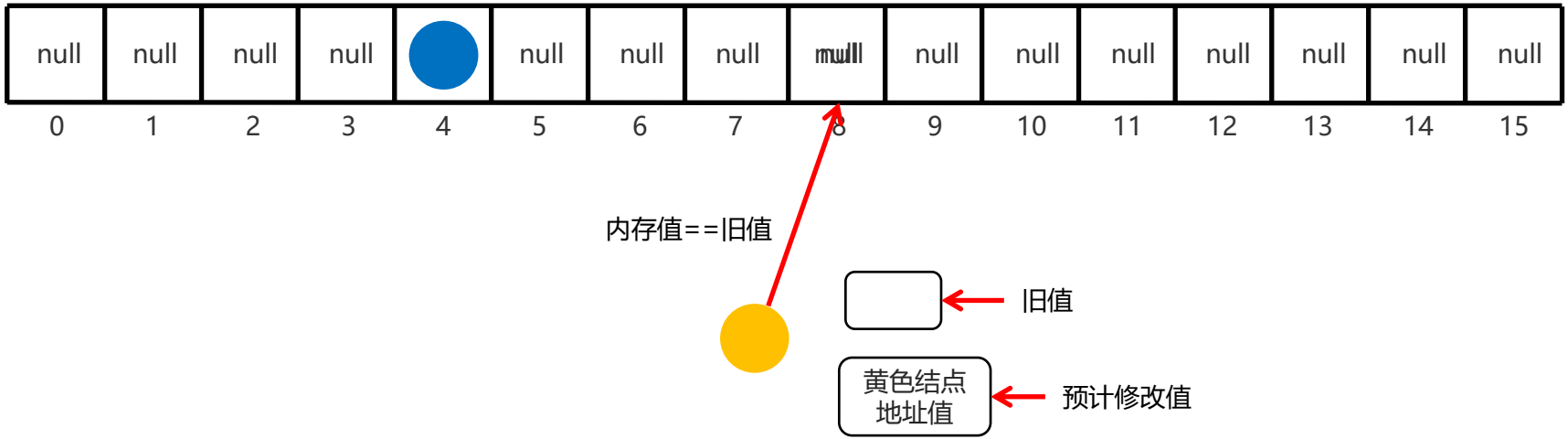
结合CAS机制 + synchronized同步代码块形式保证线程安全。

ConcurrentHashMap1.8版本原理解析

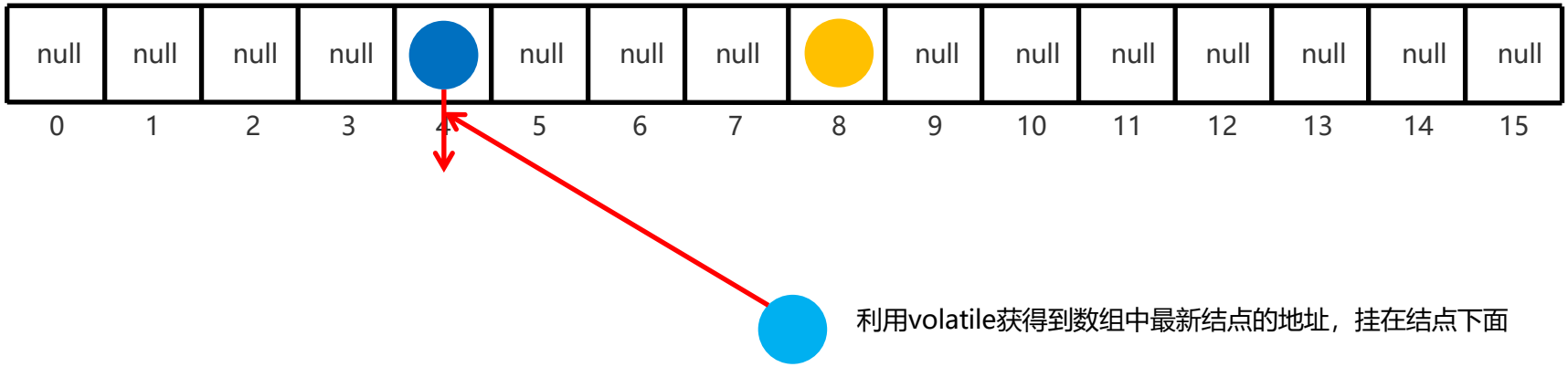


并发工具类

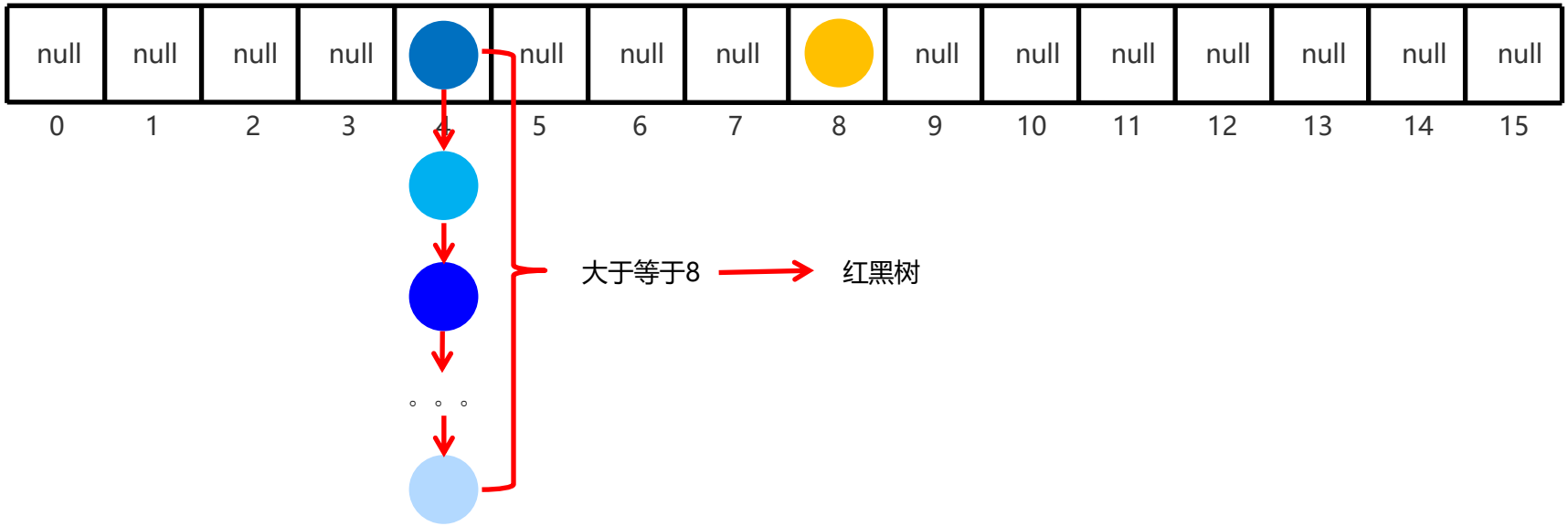
ConcurrentHashMap1.8版本原理解析



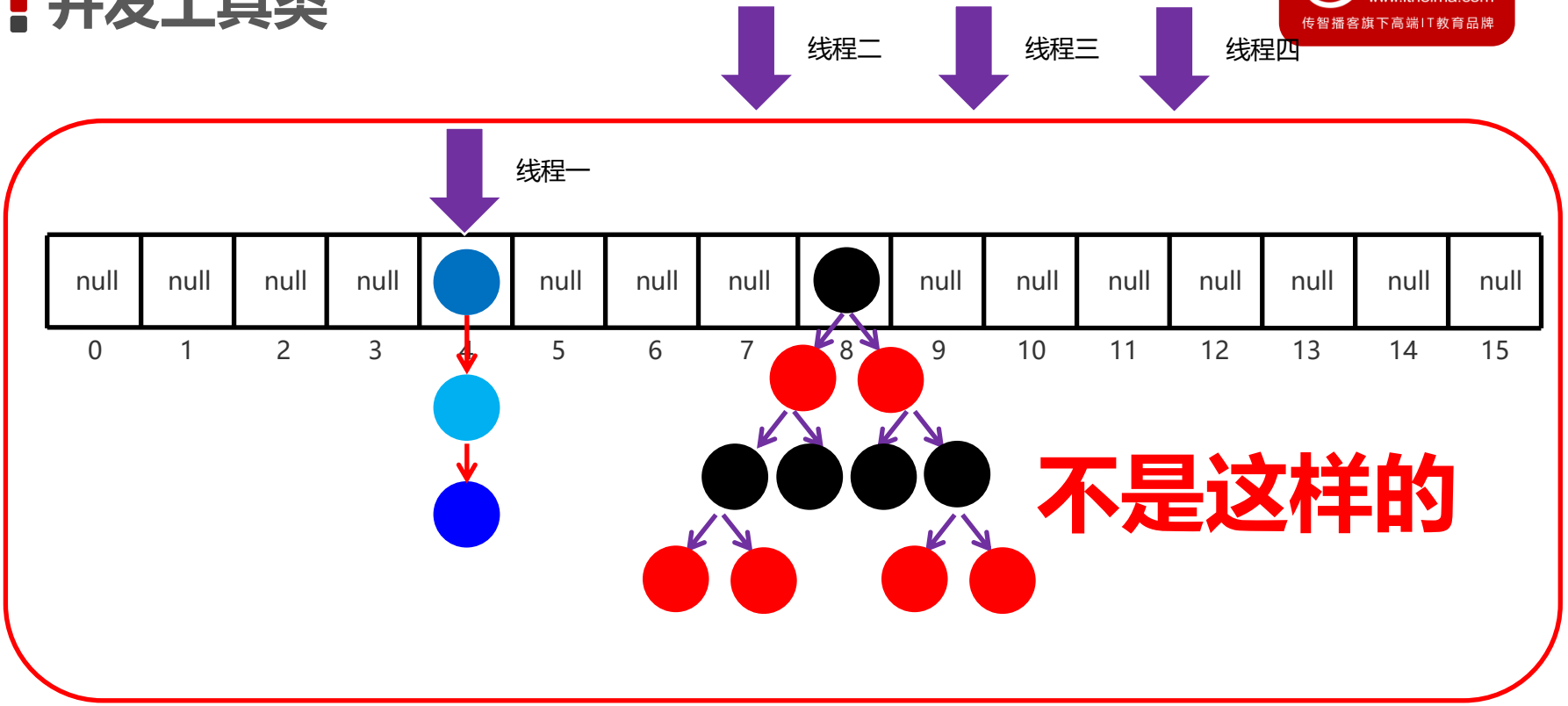
ConcurrentHashMap1.8版本原理解析



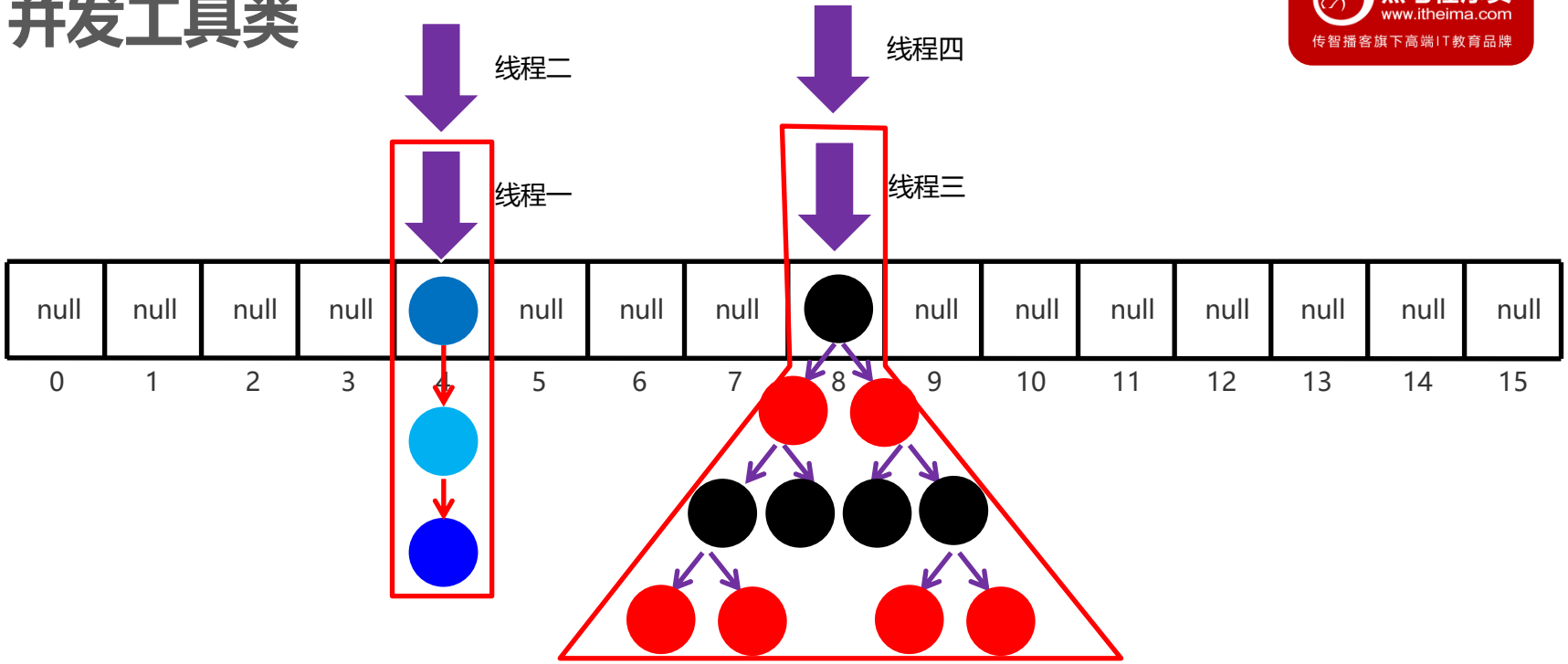
ConcurrentHashMap1.8版本原理解析



并发工具类



并发工具类



总结

1，如果使用空参构造创建ConcurrentHashMap对象，则什么事情都不做。
在第一次添加元素的时候创建哈希表

2，计算当前元素应存入的索引。

3，如果该索引位置为null，则利用cas算法，将本结点添加到数组中。

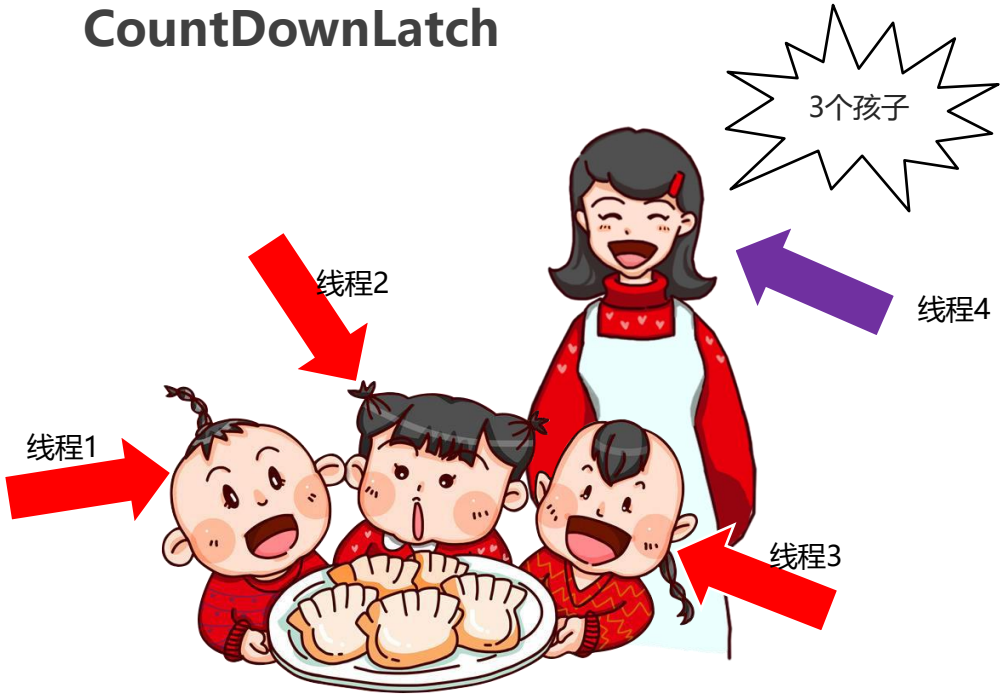
4，如果该索引位置不为null，则利用volatile关键字获得当前位置最新的结点地址，挂在他下面，变成链表。

5，当链表的长度大于等于8时，自动转换成红黑树

6，以链表或者红黑树头结点为锁对象，配合悲观锁保证多线程操作集合时数据的安全性。

并发工具类

CountDownLatch



使用场景：

让某一条线程等待其他线程执行完毕之后再执行。

并发工具类

CountDownLatch

孩子线程：

吃饺子

说一声自己吃完了

妈妈线程：

等待

收拾碗筷

测试类：

开启4条线程

创建CountDownLatch的对象传递给4条线程

CountDownLatch

方法	解释
public CountDownLatch(int count)	参数传递线程数，表示等待线程数量
public void await()	让线程等待
public void countDown()	当前线程执行完毕

CountDownLatch小结

使用场景：

让某一条线程等待其他线程执行完毕之后再执行。

CountDownLatch(int count)：参数写等待线程的数量。并定义了一个计数器。

await()：让线程等待，当计数器为0时，会唤醒等待的线程

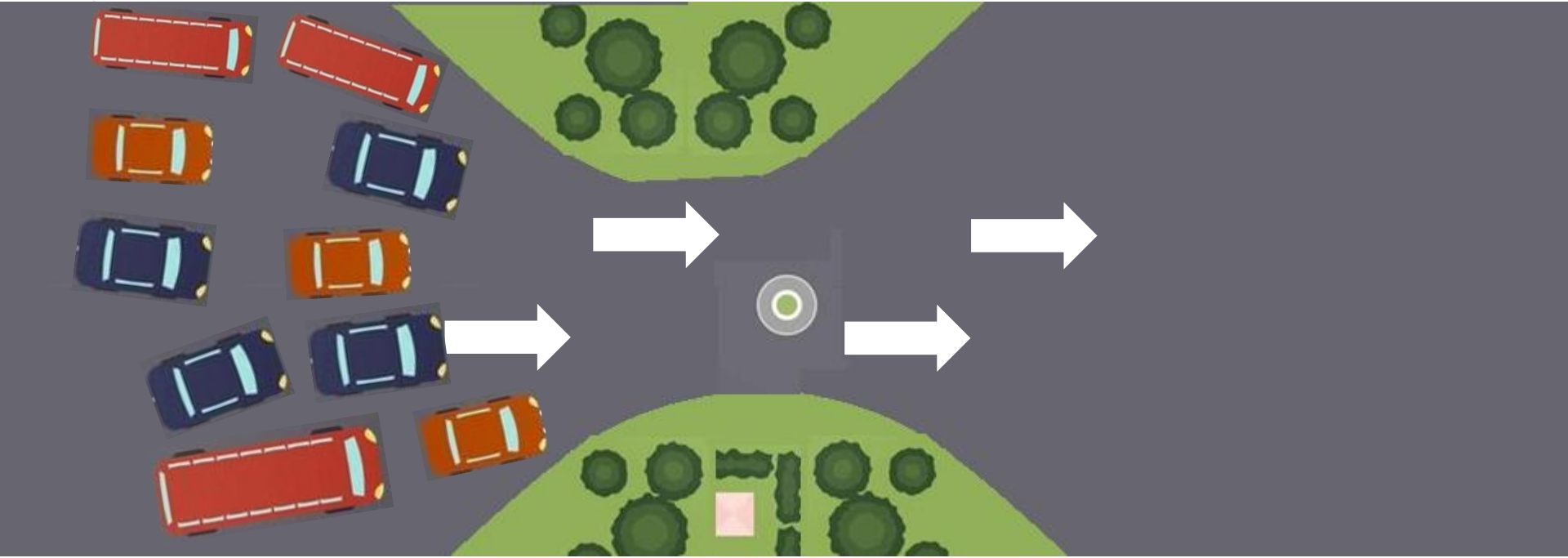
countDown()： 线程执行完毕时调用，会将计数器-1。

Semaphore

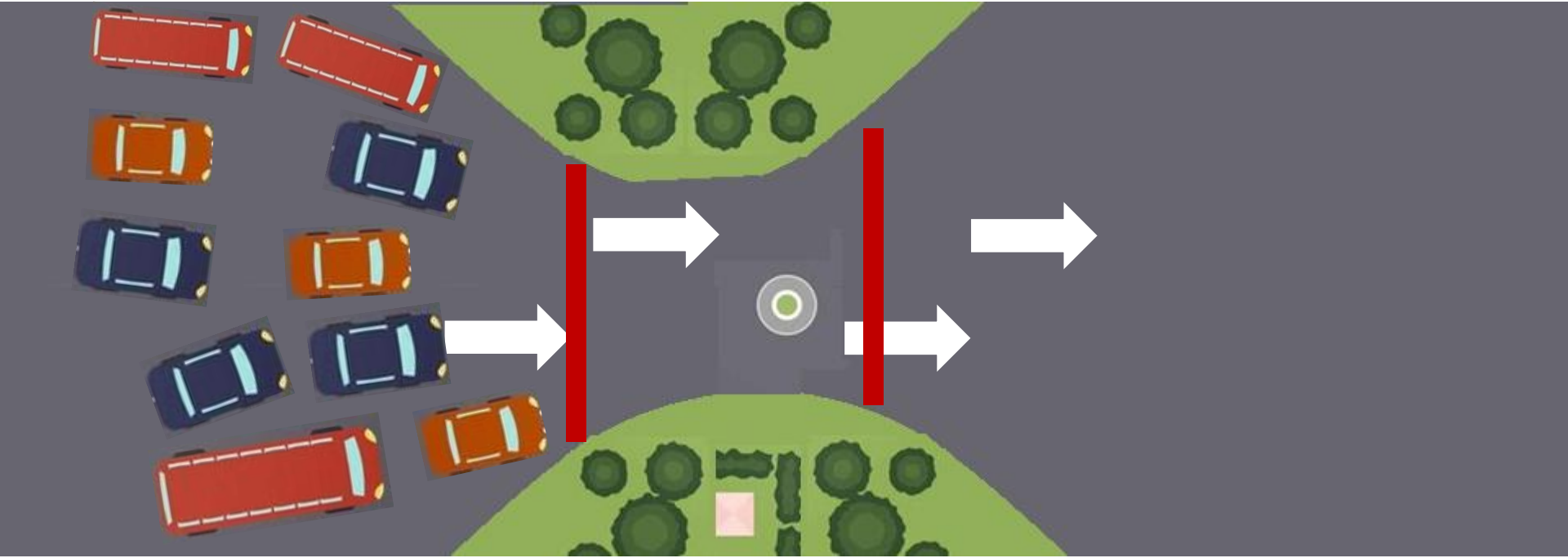
使用场景：

可以控制访问特定资源的线程数量。

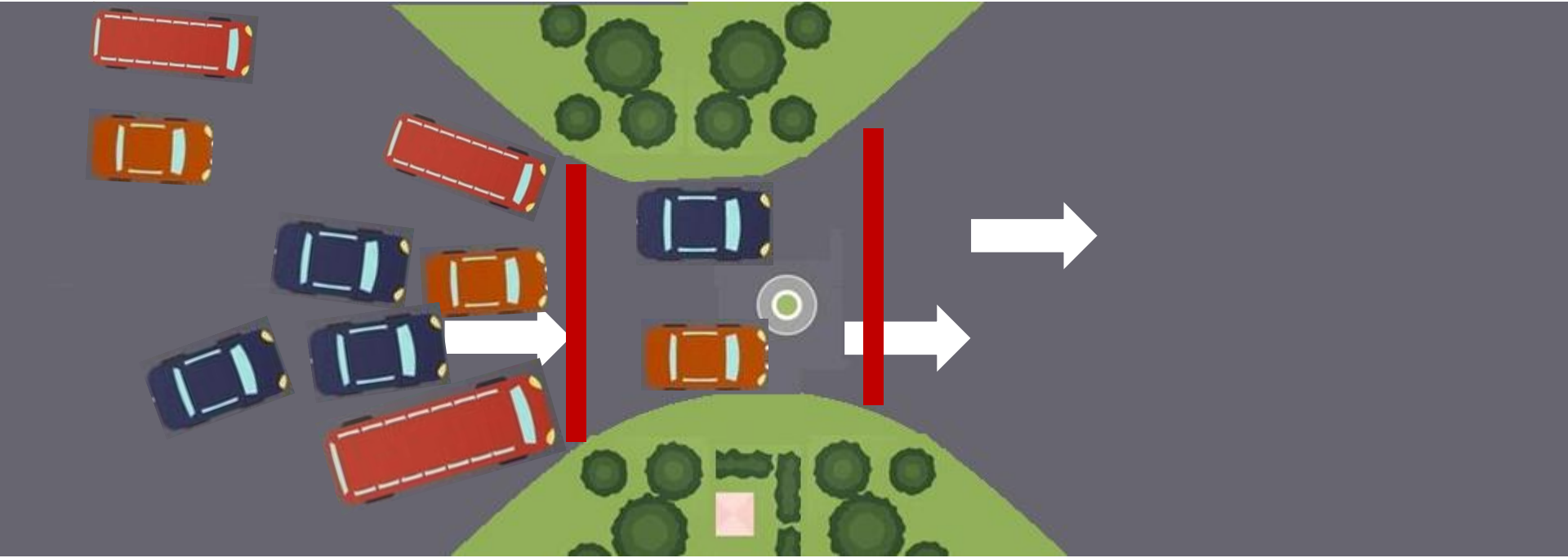
并发工具类



并发工具类



并发工具类



Semaphore

步骤：

1，需要有人管理这个通道 → 创建Semaphore对象

2，当有车进来了，发通行许可证 → acquire()发通行证

3，当车出去了，收回通行许可证 → release()收回通行证

4，如果通行许可证发完了，那么其他车辆只能等着

传智播客旗下高端IT教育品牌