方彦昆

内容

1. java中的多线程创建
2. Synchronized关键字
3. 自旋锁和自适应自旋锁
4. 公平锁 VS 非公平锁
5. 可重入锁和非可重入锁
6. 共享锁和独享锁
7. php中的锁和一些问题

PA
RT

java中多线程创建

第 一 章

01 多线程创建

```
class ThreadM extends Thread{  // 继承Thread类，作为线程的实现类
    private String name ;       // 表示线程的名称
    public ThreadM(String name){
        this.name = name ;      // 通过构造方法配置name属性
    }

    public void run(){  // 覆写run()方法，作为线程 的操作主体
        for(int i=0;i<10;i++){
            System.out.println(name + "运行，i = " + i) ;
        }
    }
}
```

01 多线程创建

```
class RunnableThread implements Runnable{
    // 实现Runnable接口，作为线程的实现类
    private String name ;       // 表示线程的名称
    public RunnableThread(String name){
        this.name = name ;      // 通过构造方法配置name属性
    }

    public void run(){  // 覆写run()方法，作为线程 的操作主体
        for(int i=0;i<10;i++){
            System.out.println(name + "运行，i = " + i) ;
        }
    }
};
```

PA
RT

Synchronized关键字

第 二 章

2 Synchronized

1. 修饰一个代码块，被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象；

```
public void run() {
        synchronized(this) {
            for (int i = 0; i < 5; i++) {
                try {
                        System.out.println(Thread.currentThread().getName() + ":" + (count++));
                        Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

2 Synchronized

2. 修饰一个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象；

```
public synchronized void run() {
    for (int i = 0; i < 5; i ++) {
        try {
            System.out.println(Thread.currentThread().getName() + ":" + (count++));
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

2 Synchronized

3. 修饰一个静态的方法，其作用的范围是整个静态方法，作用的对象是这个类的所有对象；

```
public synchronized static void method() {
        for (int i = 0; i < 5; i ++) {
            try {
                System.out.println(Thread.currentThread().getName() + ":" + (count++));
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
}
```

2 Synchronized

4. 修饰一个类，其作用的范围是synchronized后面括号括起来的部分，作用主的对象是这个类的所有对象。

```
class SyncThread implements Runnable {
  private static int count;

  public SyncThread() {count = 0 ;  }

  public static void method() {
    synchronized(SyncThread.class) {
      for (int i = 0; i < 5; i ++) {
        try {
          System.out.println(Thread.currentThread().getName() + ":" + (count++));
          Thread.sleep(100);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
  }

  public synchronized void run() { method(); }
}
```

总结

A. 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。
B. 每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。
C. 实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

PART

自旋锁和自适应自旋锁

第 三 章

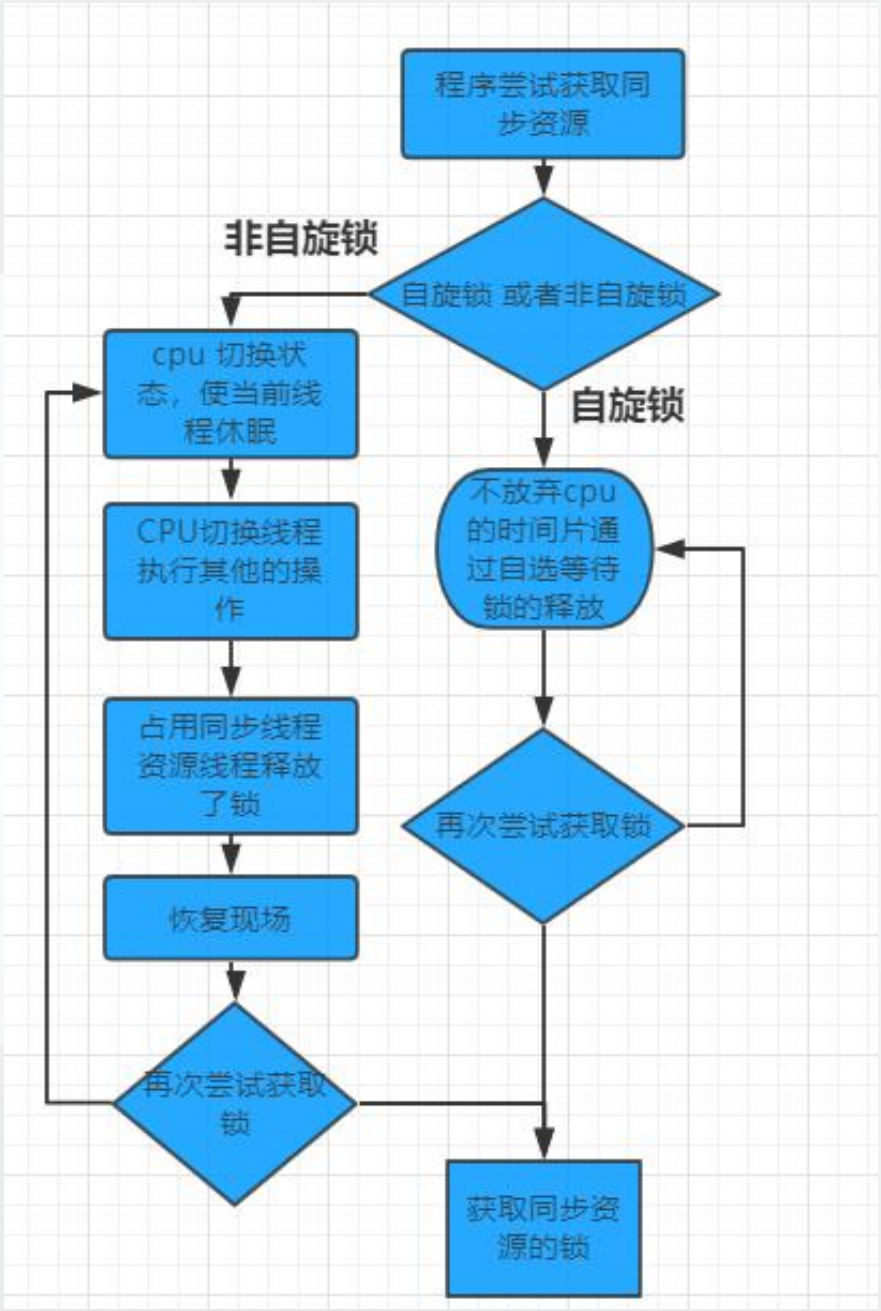
3 自旋锁和自适应自旋锁

自旋锁：

同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复现场的花费可能会让系统得不偿失。如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃CPU的执行时间，看看持有锁的线程是否很快就会释放锁，让当前线程“稍等一下”，如果在等一下后，前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不必阻塞而是直接获取同步资源，从而避免切换线程的开销。这就是自旋锁。

缺点：

自旋等待虽然避免了线程切换的开销，但它要占用处理器时间。如果锁被占用的时间很短，自旋等待的效果就会非常好。反之，如果锁被占用的时间很长，那么自旋的线程只会白浪费处理器资源。所以，自旋等待的时间必须要有一定的限度，如果自旋超过了限定次数没有成功获得锁，就应当挂起线程。



3 自旋锁和自适应自旋锁

自适应自旋锁：

自适应意味着自旋的时间（次数）不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源。

PART

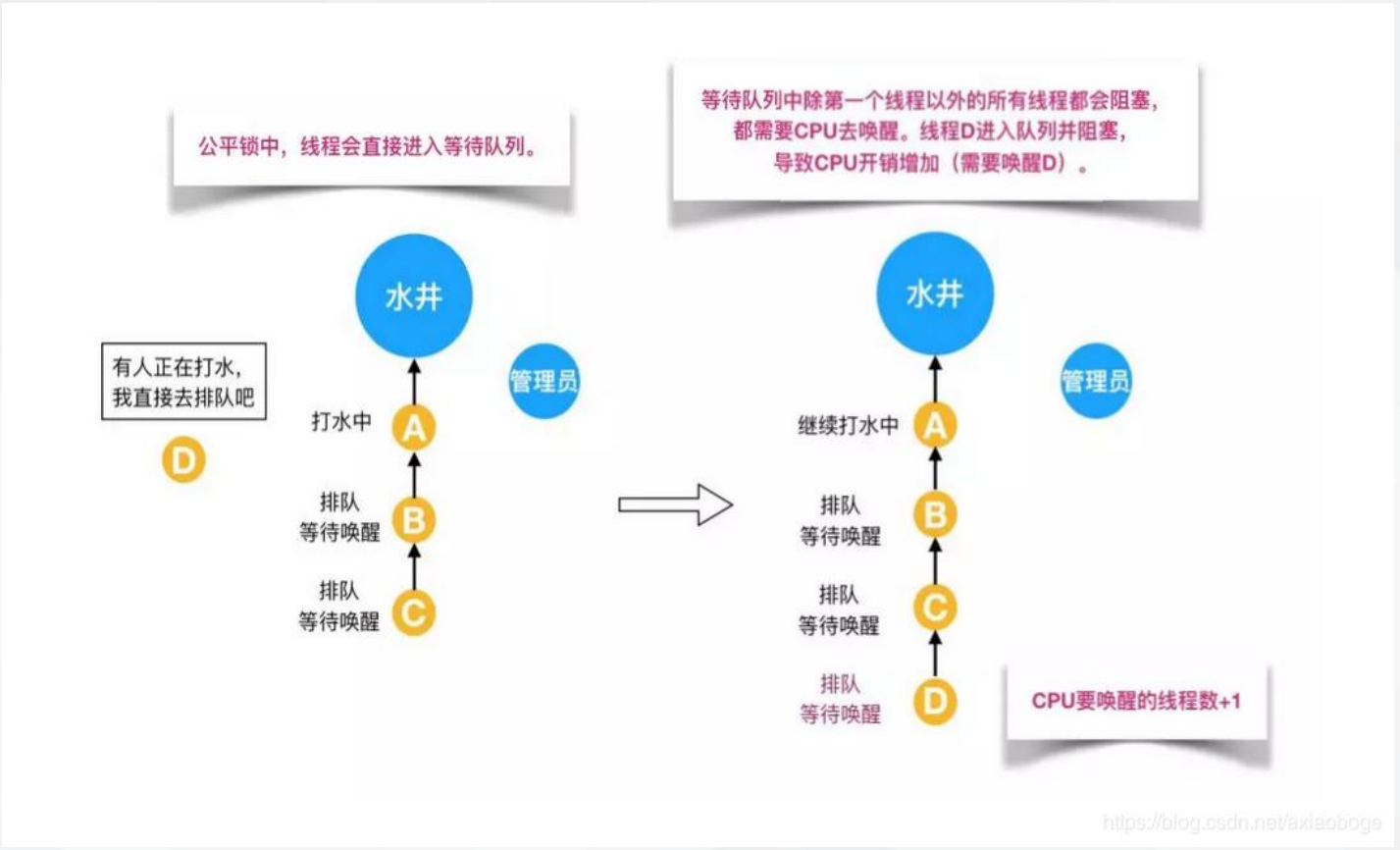
公平锁 VS 非公平锁

第 四 章

4 公平锁和非公平锁

公平锁：

公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁。公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU唤醒阻塞线程的开销比非公平锁大。

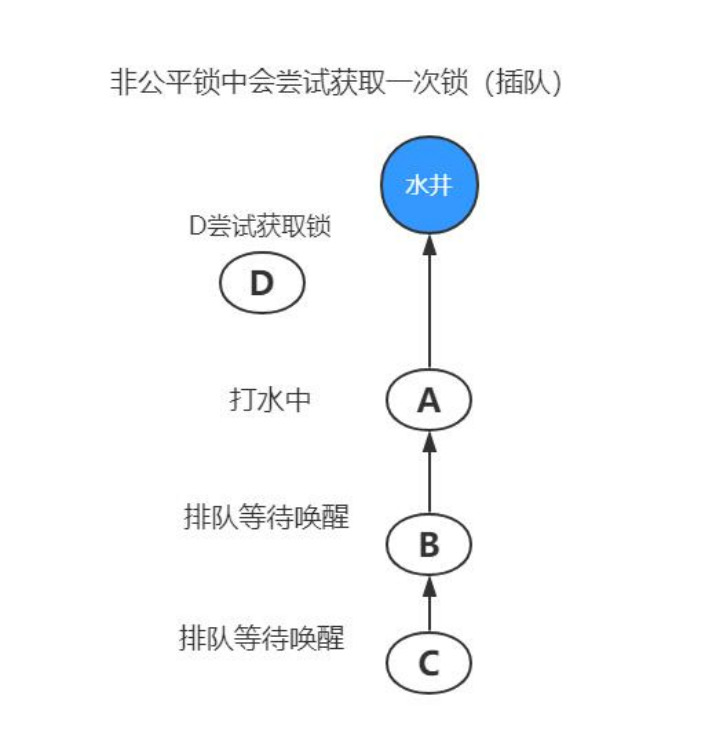


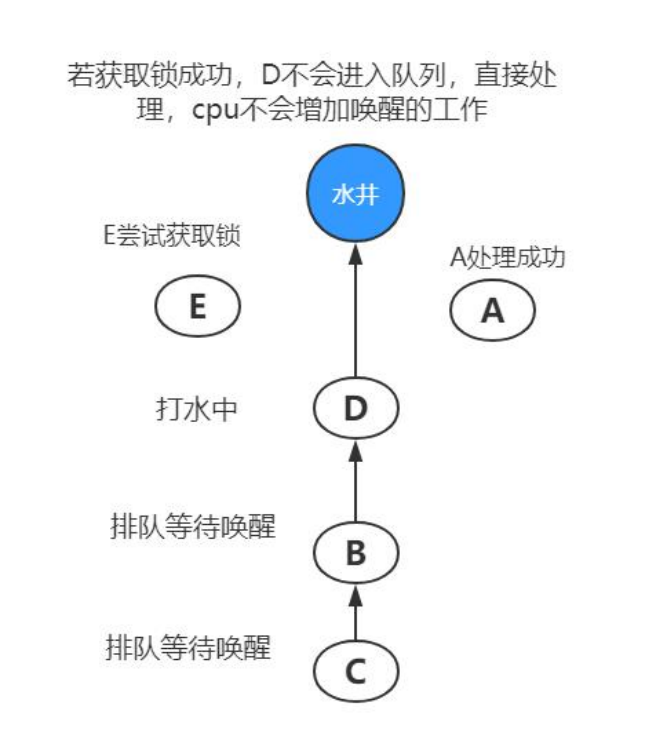
4 公平锁和非公平锁

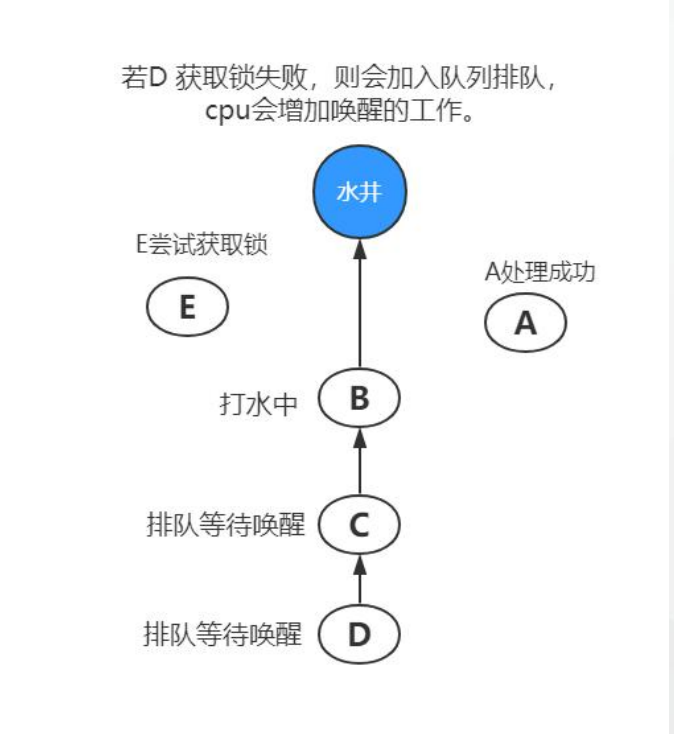
非公平：

非公平锁是多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。但如果此时锁刚好可用，那么这个线程可以无需阻塞直接获取到锁，所以非公平锁有可能出现后申请锁的线程先获取锁的场景。非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁。

饿死（starvation）是一个线程长时间得不到需要的资源而不能执行的现象。 有线程饿死并不代表着出现了死锁。







PART

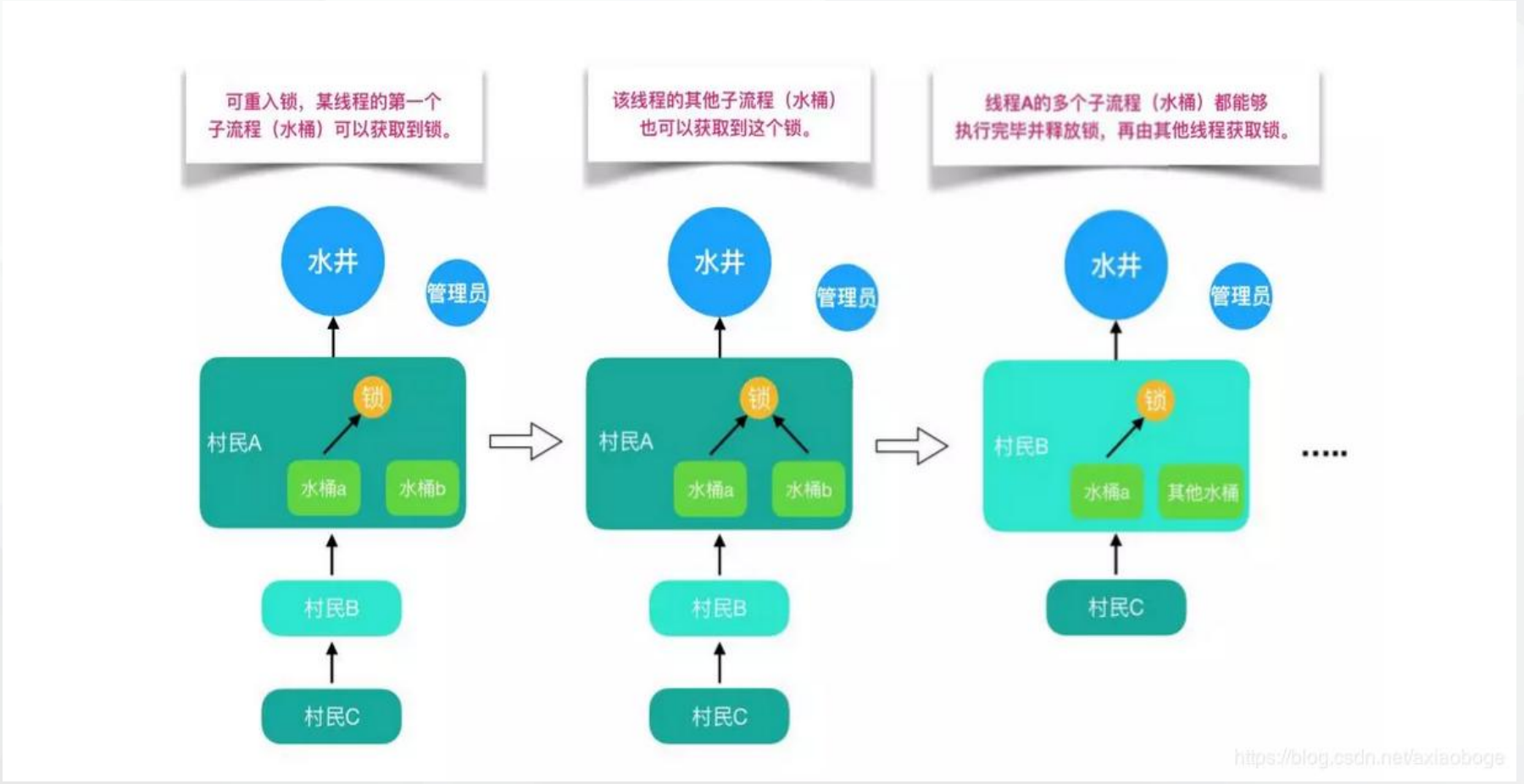
可重入锁和非可重入锁

第 五 章

4 重入锁和非重入锁

可重入锁：

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁之后，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象或者class），不会因为之前已经获取过还没释放而阻塞。Java中ReentrantLock和synchronized都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。



4 重入锁和非重入锁

```
 */
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

4 重入锁和非重入锁

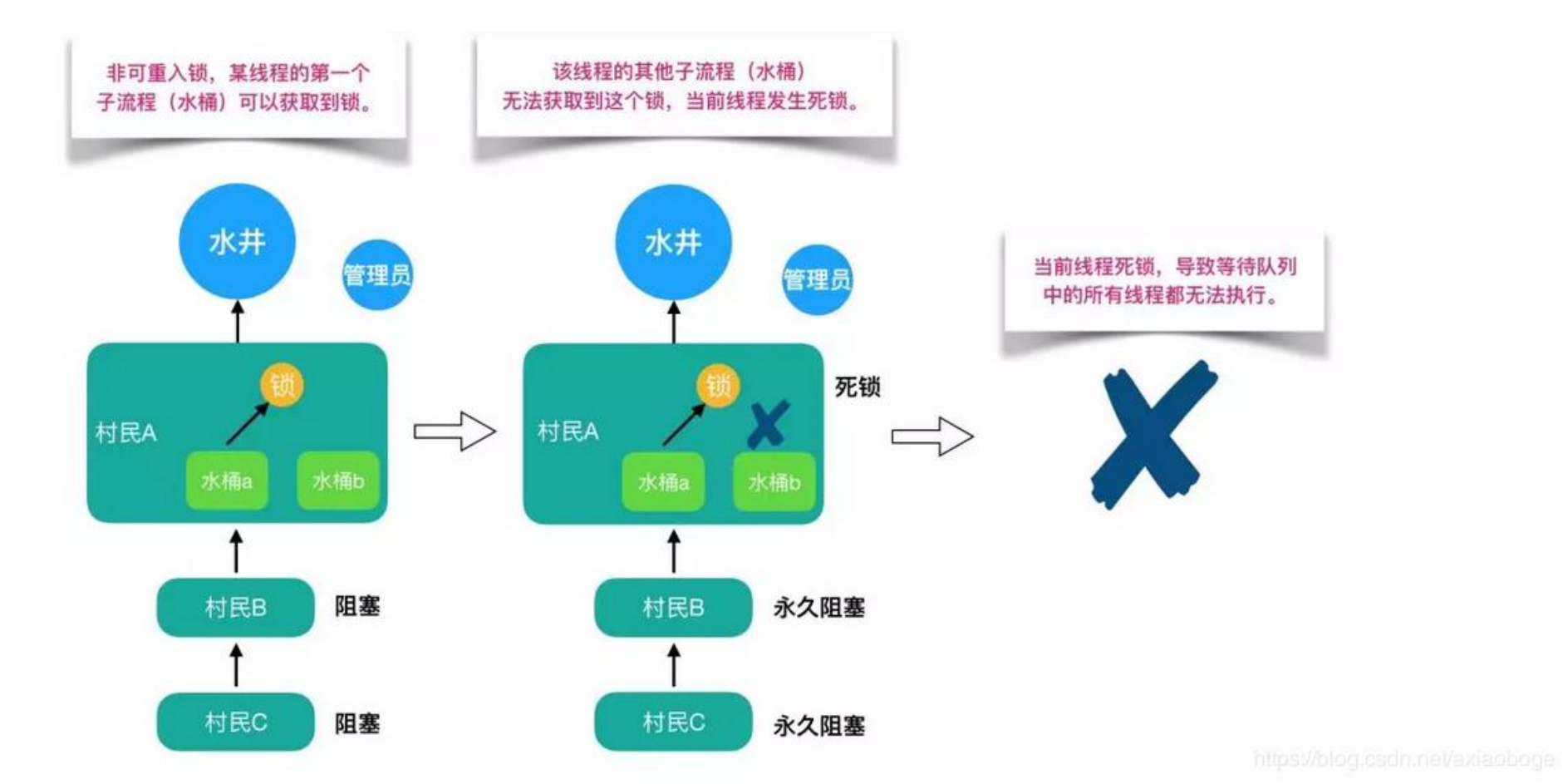
重入锁维护了一个同步状态status来计数重入次数，status初始值为0。当线程尝试获取锁时，可重入锁先尝试获取并更新status值，如果status == 0表示没有其他线程在执行同步代码，则把status置为1，当前线程开始执行。如果status != 0，则判断当前线程是否是获取到这个锁的线程，如果是的话执行status+1，且当前线程可以再次获取锁；释放锁时，可重入锁同样先获取当前status的值，判断在当前线程是持有锁，如果有锁释放锁并 status 减少，直达status-1 == 0，则表示当前线程所有重复获取锁的操作都已经执行完毕，然后该线程才会真正释放锁。

4 重入锁和非重入锁

非重入锁：

一个线程在外层方法获取锁的时候，再进入该线程的内层方法再次获取锁（前提锁对象得是同一个对象或者class），会引起死锁。

java 中的Atomic 包就是非重入锁。



PA
RT

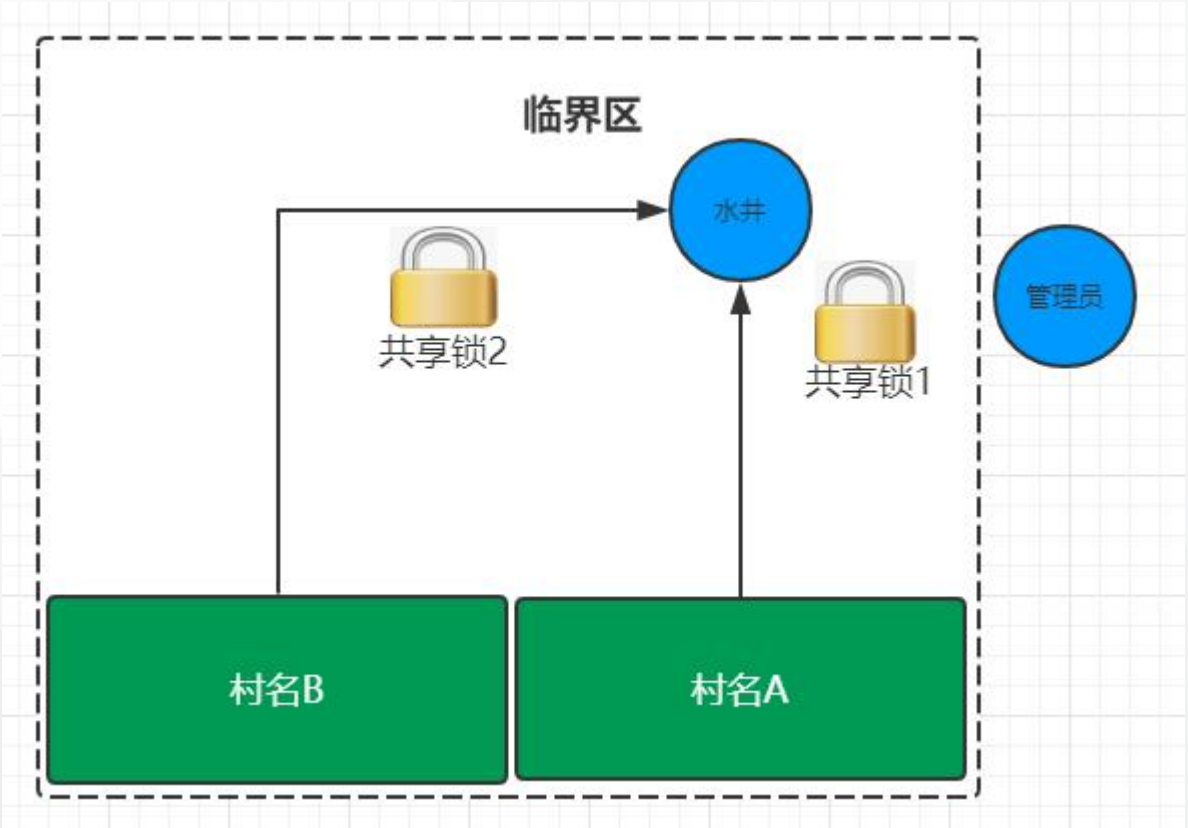
共享锁和独享锁

第 六 章

6 共享锁和独享锁

共享锁：

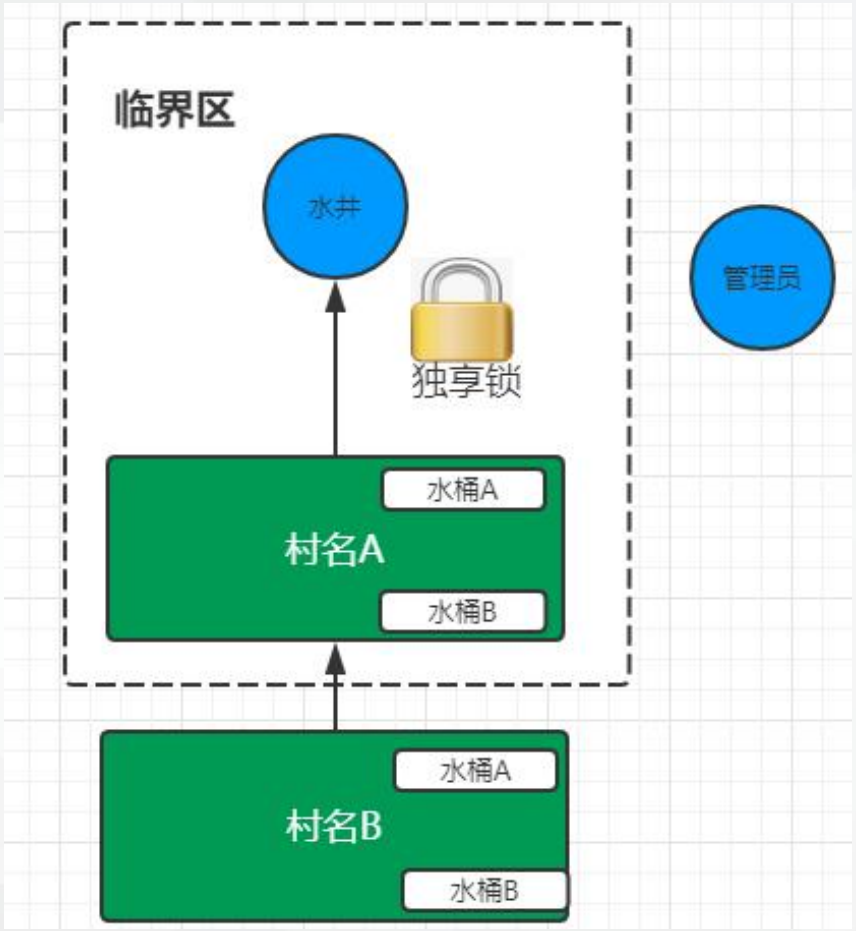
共享锁是指该锁可被多个线程所持有。如果线程T对数据A加上共享锁后，则其他线程只能对A再加共享锁，不能加排它锁。获得共享锁的线程只能读数据，不能修改数据



6 共享锁和独享锁

独享锁：

独享锁也叫排他锁，是指该锁一次只能被一个线程所持有。如果线程T对数据A加上排它锁后，则其他线程不能再对A加任何类型的锁。获得排它锁的线程即能读数据又能修改数据。



PA
RT

php中的锁和一些问题

第 七 章

6 php中的锁

PHP 的锁

```
$fp = fopen("/tmp/lock.txt", "r+");
if (flock($fp, LOCK_EX)) {  // 进行排它型锁定
    ftruncate($fp, 0);      // truncate file
    fwrite($fp, "Write something here\n");
    fflush($fp); // flush output before releasing the lock
    flock($fp, LOCK_UN);    // 释放锁定
    LOCK_SH;// 共享锁
    LOCK_EX;// 排它锁
    LOCK_UN;//解锁
} else {
    echo "Couldn't get the lock!";
}
```

6 php中的锁

进程间锁 Lock

`PHP` 代码中可以很方便地创建一个锁，用来实现数据同步。 `Lock` 类支持 `5` 种锁的类型

锁类型	说明
SWOOLE_FILELOCK	文件锁
SWOOLE_RWLOCK	读写锁
SWOOLE_SEM	信号量
SWOOLE_MUTEX	互斥锁
SWOOLE_SPINLOCK	自旋锁

请勿在 onReceive 等回调函数中创建锁，否则内存会持续增长，造成内存泄漏。

谢谢