

## 基础知识



## 线程和进程

- 进程就是运行中的程序
- 一个进程往往可以包含多个线程，至少包含一个
- Java默认有2个线程：main、GC
- Java开启线程的方式：Thread、Runnable、Callable
- Java真的可以开启线程吗？
  - 开不了，java无法直接操作硬件，最后是调用的本地方法开启的线程：

```
private native void start0();
```

## 并发和并行

- 并发：多线程操作同一个资源
  - 在单核CPU下，模拟出多条线程（多线程快速交替）
- 并行：多线程同时运行
  - CPU多核情况下，多个线程可以同时执行

## 并发编程目的

并发编程的本质：**充分利用cpu的资源**

## 线程的六个状态

```
public enum State {  
    // 新生  
    NEW,  
  
    // 运行  
    RUNNABLE,  
  
    // 阻塞  
    BLOCKED,
```

```
// 等待，死死的等
WAITING,

// 超时等待
TIMED_WAITING,

// 终止
TERMINATED;
}
```

## wait/sleep区别

1. 来自不同的类
  - wait -> Object
  - sleep -> Thread
    - sleep 不常用，用的是: `TimeUnit`
2. 关于锁的释放

wait 会释放锁，sleep抱着锁睡觉，不会释放锁
3. 使用范围是不同的
  - wait 必须在同步代码块中使用
  - sleep 可以在任何地方睡
4. 是否需要捕获异常
  - wait 不需要捕获异常
  - sleep 必须要捕获异常
5. 是否需要被唤醒
  1. sleep不需要被唤醒
  2. wait需要被唤醒

## Lock 锁（重点）

### synchronized

synchronized 锁两个东西：对象和Class

```
package juc;

/**
 * @author miemiehoho
 * @date 2022/2/10 15:22
 */
public class SaleTicket {

    public static void main(String[] args) {
        Ticket ticket = new Ticket();
        new Thread(() -> {
            for (int i = 0; i < 50; i++) {
                ticket.sale();
            }
        }, "A").start();
    }
}
```

```

        new Thread(() -> {
            for (int i = 0; i < 50; i++) {
                ticket.sale();
            }
        }, "B").start();
        new Thread(() -> {
            for (int i = 0; i < 50; i++) {
                ticket.sale();
            }
        }, "C").start();
    }
}

class Ticket {
    // 票数
    private int number = 30;

    // 卖票
    public synchronized void sale() {
        if (number > 0) {
            System.out.println(Thread.currentThread().getName() + "卖出第" +
number-- + "张票, 剩余: " + number);
        }
    }
}

```

## Synchronized 和 Lock 区别

1. Synchronized 是内置的关键字，Lock 是一个Java类
2. Synchronized 无法判断获取锁的状态，Lock可以判断是否获取到了锁
3. Synchronized 会自动释放锁，lock必须手动释放锁，否则可能会产生死锁
4. synchronized 如果 线程1获得锁后阻塞，那么线程2会一直傻傻的等待；lock锁就不一定会等下去
5. synchronized 是可重入锁，是不可以的中断的，非公平的；lock 是可重入锁，可以判断锁，默认是非公平的，但可以自己设置为公平的
6. synchronized 适合锁少量的代码同步问题，lock适合锁大量的同步代码

## 什么是锁，如何判断锁的是谁

在计算机科学中，锁(lock)与互斥(mutex)是一种同步机制，用于在许多线程执行时对资源的限制

锁的是对象

锁保证了线程串行安全执行

锁是实现同步互斥的一种机制

## 生产者和消费者问题

面试高频：单例模式、8大排序算法、生产者消费者问题、死锁

## Synchronized 版

```
package juc;

/**
 * @author miemiehoho
 * @date 2022/2/10 16:20
 */
// 线程通信
public class Demo {
    public static void main(String[] args) {
        Data data = new Data();

        new Thread() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "A").start();

        new Thread() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "B").start();

        new Thread() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "C").start();

        new Thread() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "D").start();
    }
}
```

// 判断等待, 业务, 通知

```

class Data { // 资源类

    private int number = 0;

    public synchronized void increment() throws InterruptedException {
        while (number != 0) {
            // 等待
            this.wait();
        }
        number++;
        System.out.println(Thread.currentThread().getName() + "->" + number);
        // 通知其他线程
        this.notifyAll();
    }

    public synchronized void decrement() throws InterruptedException {
        while (number == 0) {
            // 等待
            this.wait();
        }
        number--;
        System.out.println(Thread.currentThread().getName() + "->" + number);
        // 通知其它线程
        this.notifyAll();
    }
}

```

## 虚假唤醒问题

线程也可以唤醒，而不会被通知，中断或超时，即所谓的**虚假唤醒**。虽然这在实践中很少会发生，但应用程序必须通过测试应该使线程被唤醒的条件来防范，并且如果条件不满足则**继续等待**。换句话说，**等待应该总是出现在循环中**，就像这样：

```

synchronized (obj) {
    while (<<condition does not hold>>)
        obj.wait(timeout);
    ... // Perform action appropriate to condition
}

```

防止虚假唤醒

造成虚假唤醒问题的代码：

```

if (number != 0) {
    // 等待
    this.wait();
}

```

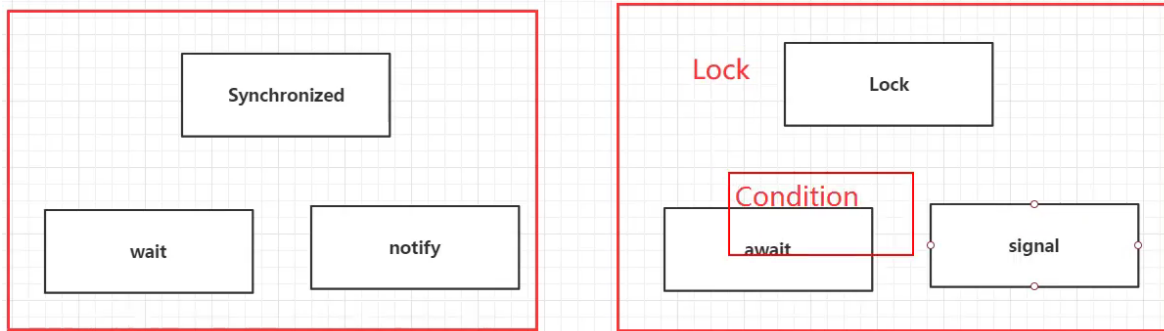
解决虚假唤醒问题：if -> while

```

while (number != 0) {
    // 等待
    this.wait();
}

```

## JUC版



```
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock(); try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }

    public Object take() throws InterruptedException {
        lock.lock(); try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally { lock.unlock(); }
    }
}
```

```
package juc;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @author miemiehoho
 * @date 2022/2/10 17:50
 */
```

```

public class LockDemo {
    public static void main(String[] args) {
        Data data = new Data();

        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "A").start();

        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "B").start();

        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "C").start();

        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "D").start();
    }
}

// 判断等待，业务，通知
class Data { // 资源类

    private int number = 0;
    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();

    public void increment() throws InterruptedException {
        try {
            lock.lock();
            while (number != 0) {
                // 等待
            }
        }
    }
}

```

```

        condition.await();
    }
    number++;
    System.out.println(Thread.currentThread().getName() + "->" +
number);
    // 通知其他线程
    condition.signalAll();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    lock.unlock();
}
}

public void decrement() throws InterruptedException {
    try {
        lock.lock();
        while (number == 0) {
            // 等待
            condition.await();
        }
        number--;
        System.out.println(Thread.currentThread().getName() + "->" +
number);
        // 通知其它线程
        condition.signalAll();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}

```

## condition 实现精准通知唤醒

**condition 同步监视器优点：**可以一个监视器只监视一个方法，来通过多个监视器监视多个方法实现精准通知

```

package juc;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @author miemiehoho
 * @date 2022/2/10 18:01
 */
public class ConditionDemo {
    public static void main(String[] args) {
        Data3 data = new Data3();
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                data.printA();
            }
        }, "A").start();
    }
}

```



```

        new Thread() -> {
            for (int i = 0; i < 10; i++) {
                data.printB();
            }
        }, "B").start();
        new Thread() -> {
            for (int i = 0; i < 10; i++) {
                data.printC();
            }
        }, "C").start();
    }
}

class Data3 {

    private Lock lock = new ReentrantLock();
    private Condition condition1 = lock.newCondition();
    private Condition condition2 = lock.newCondition();
    private Condition condition3 = lock.newCondition();
    private int flag = 1; // 标志位

    public void printA() {
        try {
            lock.lock();
            while (flag != 1) {
                // 等待
                condition1.await();
            }
            System.out.println(Thread.currentThread().getName() + " -> A");
            flag = 2;
            // 唤醒
            condition2.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void printB() {
        try {
            lock.lock();
            while (flag != 2) {
                // 等待
                condition2.await();
            }
            System.out.println(Thread.currentThread().getName() + " -> B");
            flag = 3;
            condition3.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void printC() {
        try {

```

```

        lock.lock();
        while (flag != 3) {
            // 等待
            condition3.await();
        }
        System.out.println(Thread.currentThread().getName() + " -> C");
        flag = 1;
        condition1.signal();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}

```

## 八锁现象，彻底理解锁

1、标准情况下先发短信还是先打电话？

发短信

原因：因为锁的存在，被synchronized修饰的方法，锁的对象是方法的调用者，sendSms() 和 call() 这两个方法是同一个对象 phone，用的是同一把锁，谁先拿到锁那么谁就先执行

sendSms() 方法先持有了锁

```

package juc.lock;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/10 19:25
 */
public class Test1 {

    public static void main(String[] args) throws InterruptedException {

        Phone phone = new Phone();
        new Thread(() -> {
            phone.sendSms();
        }).start();

        TimeUnit.SECONDS.sleep(1);

        new Thread(() -> {
            phone.call();
        }).start();

    }
}

```

```

class Phone {

    public synchronized void sendSms() {
        System.out.println("发短信");
    }

    public synchronized void call() {
        System.out.println("打电话");
    }
}

```

2、发短信方法 sleep 3 秒 情况下先发短信还是先打电话？

发短信

原因：因为锁的存在，被synchronized修饰的方法，锁的对象是方法的调用者，sendSms() 和 call() 这两个方法是同一个对象 phone，用的是同一把锁，谁先拿到锁那么谁就先执行

sendSms() 方法先持有了锁

```

package juc.lock;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/10 19:25
 */
public class Test1 {

    public static void main(String[] args) throws InterruptedException {

        Phone phone = new Phone();
        new Thread(() -> {
            phone.sendSms();
        }).start();

        TimeUnit.SECONDS.sleep(1);

        new Thread(() -> {
            phone.call();
        }).start();

    }
}

class Phone {

    public synchronized void sendSms() {
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发短信");
    }

    public synchronized void call() {

```

```
        System.out.println("打电话");
    }
}
```

3、增加一个普通方法情况下先发短信还是先Hello?

先 Hello

原因：普通方法不受锁的影响，synchronized 锁的是对象的调用者

```
package juc.lock;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/10 19:25
 */
public class Test1 {

    public static void main(String[] args) throws InterruptedException {

        Phone phone = new Phone();
        new Thread(() -> {
            phone.sendSms();
        }).start();

        TimeUnit.SECONDS.sleep(1);

        new Thread(() -> {
            phone.hello();
        }).start();

    }
}

class Phone {

    public synchronized void sendSms() {
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发短信");
    }

    public synchronized void call() {
        System.out.println("打电话");
    }

    public void hello() {
        System.out.println("Hello");
    }
}
```

4、两个不同的对象，分别发短信，打电话情况下先发短信还是先打电话？

先打电话

原因：两个不同的对象，两个不同的调用者，是两把不同的锁，不是同一把锁，所以是按时间先后来执行的

```
package juc.lock;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/10 19:25
 */
public class Test1 {

    public static void main(String[] args) throws InterruptedException {

        Phone phone1 = new Phone();
        Phone phone2 = new Phone();

        new Thread(() -> {
            phone1.sendSms();
        }).start();

        TimeUnit.SECONDS.sleep(1);

        new Thread(() -> {
            phone2.call();
        }).start();

    }
}

class Phone {

    public synchronized void sendSms() {
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发短信");
    }

    public synchronized void call() {
        System.out.println("打电话");
    }

    public void hello() {
        System.out.println("Hello");
    }
}
```

5、两个方法都是静态方法，情况下先发短信还是先打电话？

先发短信

原因：

1. synchronized 锁的是对象的调用者
2. 被 static 修饰的静态方法，类一加载就有了，锁的对象是 Class，是模板，是全局唯一的

```
package juc.lock;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/10 19:25
 */
public class Test1 {

    public static void main(String[] args) throws InterruptedException {

        Phone phone = new Phone();

        new Thread(() -> {
            phone.sendSms();
        }).start();

        TimeUnit.SECONDS.sleep(1);

        new Thread(() -> {
            phone.call();
        }).start();

    }
}

class Phone {

    public static synchronized void sendSms() {
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发短信");
    }

    public static synchronized void call() {
        System.out.println("打电话");
    }

}
```

6、两个不同的对象，两个方法都是静态方法，分别发短信，打电话情况下先发短信还是先打电话？  
先发短信

原因：两个方法都是 static 修饰的，所以锁的对象是 Class，两个对象的 Class 是同一个，

```
package juc.lock;

import java.util.concurrent.TimeUnit;
```

```

/**
 * @author miemiehoho
 * @date 2022/2/10 19:25
 */
public class Test1 {

    public static void main(String[] args) throws InterruptedException {

        Phone phone1 = new Phone();
        Phone phone2 = new Phone();

        new Thread(() -> {
            phone1.sendSms();
        }).start();

        TimeUnit.SECONDS.sleep(1);

        new Thread(() -> {
            phone2.call();
        }).start();

    }
}

class Phone {

    public static synchronized void sendSms() {
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发短信");
    }

    public static synchronized void call() {
        System.out.println("打电话");
    }

}

```

7、一个静态同步方法，一个普通同步方法，先执行哪个？

打电话

原因：一个静态同步方法，一个普通同步方法，锁不是同一个锁，静态同步方法锁的是Class 模板，普通同步方法锁的是phone实例对象

```

package juc.lock;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/10 19:25
 */
public class Test1 {

```

```

    public static void main(String[] args) throws InterruptedException {

        Phone phone = new Phone();

        new Thread(() -> {
            phone.sendSms();
        }).start();

        TimeUnit.SECONDS.sleep(1);

        new Thread(() -> {
            phone.call();
        }).start();

    }
}

class Phone {

    public static synchronized void sendSms() {
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发短信");
    }

    public synchronized void call() {
        System.out.println("打电话");
    }

}

```

8、两个不同的对象，一个静态同步方法，一个普通同步方法，先执行哪个？

打电话

原因：一个静态同步方法，一个普通同步方法，锁不是同一个锁，静态同步方法锁的是Class 模板，普通同步方法锁的是phone实例对象

```

package juc.lock;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/10 19:25
 */
public class Test1 {

    public static void main(String[] args) throws InterruptedException {

        Phone phone1 = new Phone();
        Phone phone2 = new Phone();
    }
}

```



```

        new Thread(() -> {
            phone1.sendSms();
        }).start();

        TimeUnit.SECONDS.sleep(1);

        new Thread(() -> {
            phone2.call();
        }).start();

    }
}

class Phone {

    public static synchronized void sendSms() {
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发短信");
    }

    public synchronized void call() {
        System.out.println("打电话");
    }

}

```

## 总结

锁有两种：

- new this 具体的一个实例对象
- static Class 唯一的一个模板

## 线程不安全的集合类

### List 不安全

List 是线程不安全的

### 实例

运行下面代码报 并发修改异常： `ConcurrentModificationException`

```

package juc.collection;

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

/**
 * @author miemiehoho
 * @date 2022/2/10 21:24
 */

```

```

public class ListTest {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                list.add(UUID.randomUUID().toString().substring(0, 5));
                System.out.println(list);
            }, String.valueOf(i)).start();
        }
    }
}

```

## 解决方案

### 方案一 Vector

Vector是线程安全的

```

package juc.collection;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
import java.util.Vector;

/**
 * @author miemiehoho
 * @date 2022/2/10 21:24
 */
public class ListTest {

    public static void main(String[] args) {
        Vector<String> vector = new Vector<>();

        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                vector.add(UUID.randomUUID().toString().substring(0, 5));
                System.out.println(vector);
            }, String.valueOf(i)).start();
        }
    }
}

```

### 方案二 Collections工具类

```

package juc.collection;

import java.util.*;

/**
 * @author miemiehoho
 * @date 2022/2/10 21:24
 */

```

```

public class ListTest {

    public static void main(String[] args) {
        List<String> list = Collections.synchronizedList(new ArrayList<>());
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                list.add(UUID.randomUUID().toString().substring(0, 5));
                System.out.println(list);
            }, String.valueOf(i)).start();
        }
    }
}

```

### 方案三 JUC包下的 CopyOnWriteArrayList

#### CopyOnWriteArrayList:

CopyOnWrite 写入时复制，COW思想，计算机程序设计领域的一种优化策略，

多个线程调用的时候，读取的时候是固定，但是对于写入，要COW，避免覆盖，造成数据不安全问题

#### CopyOnWriteArrayList 相比于 Vector的优势:

只要有 synchronized 的方法，效率相对就会比较低，而CopyOnWriteArrayList 的 add() 方法没有使用 synchronized 关键字，而是使用的 lock锁

源码:

```

public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        lock.unlock();
    }
}

```

```

package juc.collection;

import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;

/**
 * @author miemiehoho
 * @date 2022/2/10 21:24
 */
public class ListTest {

```

```

    public static void main(String[] args) {
        List<String> list = new CopyOnWriteArrayList<>();
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                list.add(UUID.randomUUID().toString().substring(0, 5));
                System.out.println(list);
            }, String.valueOf(i)).start();
        }
    }
}

```

## Set 不安全

### 实例

运行下面代码报 并发修改异常: `ConcurrentModificationException`

```

package juc.collection;

import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.UUID;
import java.util.concurrent.CopyOnWriteArrayList;

/**
 * @author miemiehoho
 * @date 2022/2/10 21:54
 */
public class SetTest {

    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                set.add(UUID.randomUUID().toString().substring(0, 5));
                System.out.println(set);
            }, String.valueOf(i)).start();
        }
    }
}

```

## 解决方案

### 方案一 Collections工具类

```

package juc.collection;

import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;

/**
 * @author miemiehoho
 * @date 2022/2/10 21:54
 */

```

```

*/
public class SetTest {

    public static void main(String[] args) {
        Set<String> set = Collections.synchronizedSet(new HashSet<>());
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                set.add(UUID.randomUUID().toString().substring(0, 5));
                System.out.println(set);
            }, String.valueOf(i)).start();
        }
    }
}

```

## 方案二 JUC包下的 CopyOnWriteSet

```

package juc.collection;

import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.CopyOnWriteArraySet;

/**
 * @author miemiehoho
 * @date 2022/2/10 21:54
 */
public class SetTest {

    public static void main(String[] args) {
        Set<String> set = new CopyOnWriteArraySet<>();
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                set.add(UUID.randomUUID().toString().substring(0, 5));
                System.out.println(set);
            }, String.valueOf(i)).start();
        }
    }
}

```

## HashSet 的底层是什么?

HashSet 的底层就是 HashMap, set的本质就是 hashMap的key, key是无法重复的

源码:

```

public HashSet() {
    map = new HashMap<>();
}

```

## HashSet 的 add() 方法

源码：

```
public boolean add(E e) {  
    return map.put(e, PRESENT) == null;  
}
```

PRESENT 是一个常量，固定的值

```
private static final Object PRESENT = new Object();
```

## HashMap 不安全

loadFactor 加载因子 0.75  
initialCapacity 初始容量 16

### 解决方案

#### 方案一 Collections工具类

```
Map<String, String> map = Collections.synchronizedMap(new HashMap<>());
```

#### 方案二 JUC包下的 ConcurrentHashMap

```
Map<String, String> map = new ConcurrentHashMap<>();
```

## ConcurrentHashMap 的原理？

## Callable

Callable接口类似于Runnable，因为它们都是为其实例可能由另一个线程执行的类设计的。然而，A Runnable不返回结果，也不能抛出被检查的异常。

该Executors类包含的实用方法，从其他普通形式转换为Callable类。

Callable 方式创建线程的特点：

1. 有返回值
2. 可以抛出异常
3. 需要实现的方法不同 run()/ call()

## 用 Callable接口 创建线程

FutureTask 是 Runnable 接口的实现类，含构造方法：

```
public FutureTask(Callable<V> callable) {
    if (callable == null)
        throw new NullPointerException();
    this.callable = callable;
    this.state = NEW;      // ensure visibility of callable
}
```

```
public FutureTask(Runnable runnable, V result) {
    this.callable = Executors.callable(runnable, result);
    this.state = NEW;      // ensure visibility of callable
}
```

## 实例

```
public static void main(String[] args) {
    FutureTask<Integer> task = new FutureTask<>(new Callable<Integer>() {
        @Override
        public Integer call() throws Exception {
            System.out.println("call()");
            return 123445;
        }
    });

    new Thread(task, "A").start();
    new Thread(task, "B").start();

    try {
        Integer i = task.get(); // get() 方法可能会产生阻塞
        System.out.println(i);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
```

- get() 方法可能会产生阻塞
- 启用多个方法 get()方法只会返回一次结果

原因：

- 由于JVM第二次调用 FutureTask对象所持有的线程时，由于此时FutureTask的state此时已非NEW状态（各个状态，这边不做详细解释），则此时会直接结束，不执行
- 这个看一眼FutureTask源码就清楚了，用的是一个state存放线程状态，outcome存放结果
- FutureTask的run方法源码：

```

public void run() {
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                       null, Thread.currentThread()))
        return;
    .....
}

```

## 常用的辅助类（必会）

### CountDownLatch

#### 一个减法计数器

- 等待计数器归零，然后才会再向下执行

```

public class CountDownLatch
    extends Object

```

允许一个或多个线程等待直到在其他线程中执行的一组操作完成的同步辅助。

A CountDownLatch用给定的 *N* 计数初始化。 *await* 方法阻塞，直到由于 *countDown()* 方法的调用而导致当前计数达到零，之后所有等待线程被释放，并且任何后续的 *await* 调用立即返回。 这是一个一次性的现象 - 计数无法重置。 如果您需要重置计数的版本，请考虑使用 *CyclicBarrier*。

A CountDownLatch是一种通用的同步工具，可用于多种用途。 一个CountDownLatch为一个计数的CountDownLatch用作一个简单的开/关锁存器，或者门：所有线程调用*await*在门口等待，直到被调用*countDown()*的线程打开。 一个CountDownLatch初始化*N*可以用来做一个线程等待，直到*N*个线程完成某项操作，或某些动作已经完成*N*次。

CountDownLatch一个有用的属性是，它不要求调用*countDown*线程等待计数到达零之前继续，它只是阻止任何线程通过*await*，直到所有线程可以通过。

### 实例

```

package juc.help;

import java.util.concurrent.CountDownLatch;

/**
 * @author miemiehoho
 * @date 2022/2/11 9:48
 */
public class TestCountDownLatch {

    public static void main(String[] args) throws InterruptedException {
        // 计数器，一般用在必须要执行任务的时候使用
        CountDownLatch countDownLatch = new CountDownLatch(6);
        for (int i = 1; i <= 6; i++) {
            new Thread() -> {
                System.out.println(Thread.currentThread().getName() + "go out");
                countDownLatch.countDown(); // 计数器减一
            }.start();
        }

        countDownLatch.await(); // 等待计数器归零，然后再向下执行

        System.out.println("关门");
    }
}

```



## 原理

```
countDownLatch.countDown();// 计数器减一
countDownLatch.await();// 等待计数器归零，然后再向下执行
```

每次有线程调用 `countDown()` ,计数器就会减一，当计数器归零后，`countDownLatch.await()` 就会被唤醒，然后再向下执行

## CyclicBarrier

一个线程加法计数器（实际是一个减法计数器，源码中是做的减法）

- 一个加法计数器，需要等待计数完成才会执行，否则线程就会被一直阻塞
- 这个辅助类是通过`await`来记数的，`await`之后本次线程会被阻塞，直到神龙召唤后才执行后面的
- 匿名内部类访问局部变量要加`final`，加`final`，变量会存在堆中的方法区里，子线程共享进程的堆，所以能读到。否则是存在另一个线程的栈中

```
public class CyclicBarrier
extends Object
```

允许一组线程全部等待彼此达到共同屏障点的同步辅助。循环阻塞在涉及固定大小的线程方的程序中很有用，这些线程必须偶尔等待彼此。屏障被称为循环，因为它可以在等待的线程被释放之后重新使用。

A `CyclicBarrier` 支持一个可选的 `Runnable` 命令，每个屏障点运行一次，在派对中的最后一个线程到达之后，但在任何线程释放之前。在任何一方继续进行之前，此屏障操作对更新共享状态很有用。

## 实例

```
package juc.help;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

/**
 * @author miemiehoho
 * @date 2022/2/11 10:03
 */
public class TestCyclicBarrier {

    public static void main(String[] args) {
        // 集齐七颗龙珠召唤神龙
        // 召唤神龙的线程
        CyclicBarrier cyclicBarrier = new CyclicBarrier(7, new Thread(() -> {
            System.out.println("召唤神龙成功!");
        }));
        for (int i = 1; i <= 7; i++) {
            // 匿名内部类访问局部变量要加final
            // 加final，变量会存在堆中的方法区里，子线程共享进程的堆，所以能读到。否则是存在
            // 另一个线程的栈中，不同线程读不到
            // 成员变量的生命周期更长，当成员变量中引用了局部变量，那么就需要加final，复制一
            // 份到堆内存中，否则引用的该变量就访问不到了
            final int temp = i;
            // lambda 无法操作到变量i，它的本质是new了一个类，无法操作主线程的局部变量
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName() + "拿到龙珠: "
                    + temp);

                try {
                    // 这个辅助类是通过await来记数的，await之后本次线程会被阻塞，直到神龙
                    // 召唤后才执行后面的
                } catch (BrokenBarrierException e) {
                    // ...
                }
            }).start();
        }
    }
}
```

```

        cyclicBarrier.await();// 等待
        System.out.println(Thread.currentThread().getName() +
"over");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        e.printStackTrace();
    }
    }, String.valueOf(i)).start();
}
}
}
}

```

## 原理

```

// 召唤神龙的线程
CyclicBarrier cyclicBarrier = new CyclicBarrier(7, new Thread(() -> {
    System.out.println("召唤神龙成功！");
}));
// 这个辅助类是通过await来记数的，await之后本次线程会被阻塞，直到神龙召唤后才执行后面的
cyclicBarrier.await();// 等待

```

## Semaphore

### 计数信号量

- 用于限流

```

public class Semaphore
extends Object
implements Serializable

```

一个计数信号量。在概念上，信号量维持一组许可证。如果有必要，每个acquire()都会阻塞，直到许可证可用，然后才能使用它。每个release()添加许可证，潜在地释放阻塞获取方。但是，没有使用实际的许可证对象；Semaphore只保留可用数量的计数，并相应地执行。

信号量通常用于限制线程数，而不是访问某些（物理或逻辑）资源。例如，这是一个使用信号量来控制对一个项目池的访问的类：

### 实例 - 抢车位

```

package juc.help;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/11 10:26
 */
public class TestSemaphore {

    public static void main(String[] args) {
        // 线程数量：停车位，常用在限流
        Semaphore semaphore = new Semaphore(3);

        for (int i = 0; i < 6; i++) {
            new Thread(() -> {
                try {

```

```

        // 抢到车位
        semaphore.acquire();
        System.out.println(Thread.currentThread().getName() + "抢到了
车位");

        TimeUnit.SECONDS.sleep(1);
        System.out.println(Thread.currentThread().getName() + "离开车
位");

    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        // 释放车位
        semaphore.release();
    }
}, String.valueOf(i)).start();

    }
}
}

```

因为数量满了，acquire方法会阻塞，直到有多余空间释放出来，才会继续执行

## 原理

```

// 抢到车位
semaphore.acquire();// 获得信号量，如果已经满了就会等待，直到有信号量被释放为止
// 释放车位
semaphore.release();// 释放信号量，当前的信号量释放，然后唤醒等待的线程

```

## 作用

1. 多个共享资源互斥的使用
2. 并发限流，控制最大线程数量

## 总结

CountDownLatch：指定线程执行完毕，再执行操作

CyclicBarrier：执行达到指定线程数再执行操作

Semaphore：同一时间只能有指定数量得到线程

## 读写锁

- 读写锁可以实现更细粒度的控制
- 写锁（独占锁）一次只能被一个线程占有
- 读锁（共享锁）可以被多个线程同时占有

### ReadWriteLock

读 - 读 可以共存  
 读 - 写 不能共存（必须等写入完才能读）  
 写 - 写 不能共存

所有已知实现类：

ReentrantReadWriteLock

读可以被多线程同时读  
写的时候只能有一个线程去写

```
public interface ReadWriteLock
```

A `ReadWriteLock` 维护一对关联的 `locks`，一个用于只读操作，一个用于写入。`read lock` 可以由多个阅读器线程同时进行，只要没有作者。`write lock` 是独家的。

所有 `ReadWriteLock` 实现必须保证的存储器同步效应 `writeLock` 操作（如在指定 `Lock` 接口）也保持相对于所述相关联的 `readLock`。也就是说，一个线程成功获取读锁定将会看到在之前发布的写锁定所做的所有更新。

读写锁允许访问共享数据时的并发性高于互斥锁所允许的并发性。它利用了这样一个事实：一次只有一个线程（写入线程）可以修改共享数据，在许多情况下，任何数量的线程都可以同时读取数据（因此 *读数据线程*）。从理论上讲，通过使用读写锁允许的并发性增加将导致性能改进超过使用互斥锁。实际上，并发性的增加只能在多处理器上完全实现，然后只有在共享数据的访问模式是合适的时才可以。

读写锁是否会提高使用互斥锁的性能取决于数据被读取的频率与被修改的频率相比，读取和写入操作的持续时间以及数据的争用 - 即是，将尝试同时读取或写入数据的线程数。例如，最初填充数据的集合，然后经常被修改的频繁搜索（例如某种目录）是使用读写锁的理想候选。然而，如果更新变得频繁，那么数据的大部分时间将被专门锁定，并且并发性增加很少。此外，如果读取操作太短，则读写锁定实现（其本身比互斥锁更复杂）的开销可以支配执行成本，特别是因为许多读写锁定实现仍将序列化所有线程通过小部分代码。最终，只有剖析和测量将确定使用读写锁是否适合您的应用程序。

## 实例

```
package juc.readwrite;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * @author miemiehoho
 * @date 2022/2/11 11:01
 */
public class ReadWriteLockDemo {

    public static void main(String[] args) {
        MyCache myCache = new MyCache();

        // 写入
        for (int i = 0; i < 5; i++) {
            final int temp = i;
            new Thread(() -> {
                myCache.put(String.valueOf(temp), String.valueOf(temp));
            }, String.valueOf(temp)).start();
        }

        // 读取
        for (int i = 0; i < 5; i++) {
            final int temp = i;
            new Thread(() -> {
                myCache.get(String.valueOf(temp));
            }, String.valueOf(temp)).start();
        }
    }
}

class MyCache {
```

```

private volatile Map<String, String> map = new HashMap<>();
// 读写锁，可以更细粒度的控制
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

// 写，希望只有一个线程可以写
public void put(String key, String value) {
    try {
        // 写锁
        readWriteLock.writeLock().lock();
        System.out.println(Thread.currentThread().getName() + "写入:" + key);
        map.put(key, value);
        System.out.println(Thread.currentThread().getName() + "写入ok");
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        readWriteLock.writeLock().unlock();
    }
}

// 读，可以多个线程同时读
public void get(String key) {
    try {
        readWriteLock.readLock().lock();
        System.out.println(Thread.currentThread().getName() + "读取:" + key);
        map.get(key);
        System.out.println(Thread.currentThread().getName() + "读取ok");
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        readWriteLock.readLock().unlock();
    }
}
}

```

## 阻塞队列 - BlockingQueue

阻塞

队列

FIFO

- 写入：如果队列满了，就必须阻塞等待
- 读取：如果队列是空的，就必须阻塞等待生产

java.util.concurrent

## Interface **BlockingQueue<E>**

参数类型

E - 此集合中保存的元素的类型

All Superinterfaces:

Collection <E>, Iterable <E>, Queue <E>

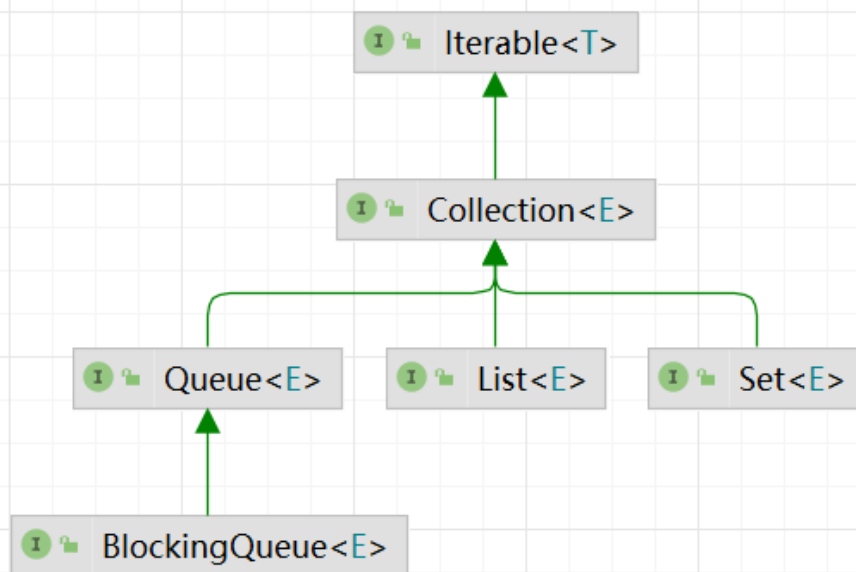
All Known Subinterfaces:

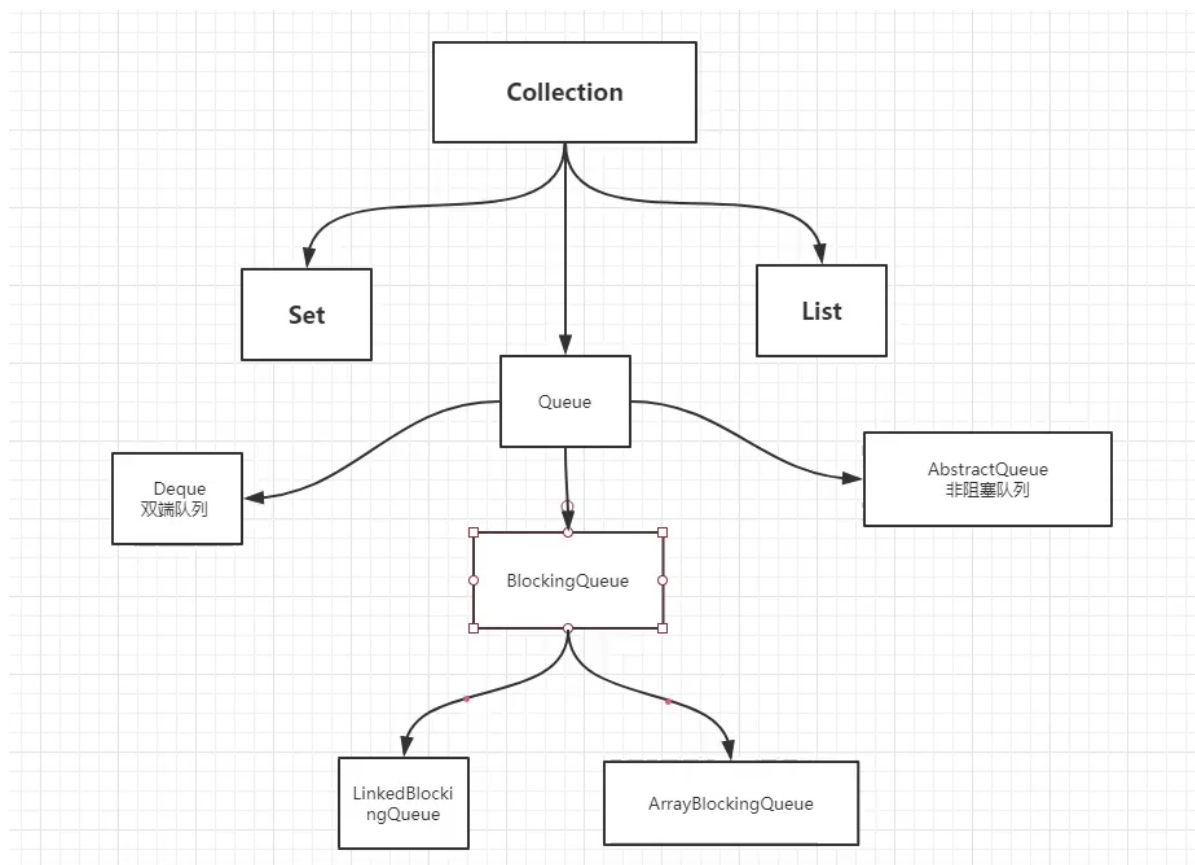
BlockingDeque <E>, TransferQueue <E>

所有已知实现类:

ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue

同步队列





## BlockingQueue

## 阻塞队列的使用场景

多线程并发处理、线程池

## 队列的常用操作

添加、移除

## 四组API

1. 抛出异常
2. 不会抛出异常
3. 阻塞等待
4. 超时等待

	抛出异常	不抛出异常	阻塞等待	超时等待
添加	add()	offer()	put()	offer()
移除	remove()	poll()	take()	poll()
判断队列首元素	element()	peek()	-	-

```
package juc.queue;

import java.util.List;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
```

```

/**
 * @author miemiehoho
 * @date 2022/2/11 11:53
 */
public class TestBlockingQueue {

    public static void main(String[] args) {
        try {
            test3();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // 抛出异常
    public static void test1() {
        // 设置队列容量
        ArrayBlockingQueue queue = new ArrayBlockingQueue(3);

        System.out.println(queue.add("a"));
        System.out.println(queue.add("b"));
        System.out.println(queue.add("c"));
        //      System.out.println(queue.add("d"));
        System.out.println(queue.element());
        System.out.println(queue.remove());
        System.out.println(queue.remove());
        System.out.println(queue.remove());
        //      System.out.println(queue.remove());
    }

    // 不抛出异常
    public static void test2() {
        // 设置队列容量
        ArrayBlockingQueue queue = new ArrayBlockingQueue(3);

        System.out.println(queue.offer("a"));
        System.out.println(queue.offer("b"));
        System.out.println(queue.offer("c"));
        //      System.out.println(queue.offer("d"));
        //      System.out.println(queue.peek());
        System.out.println(queue.poll());
        System.out.println(queue.poll());
        System.out.println(queue.poll());
        System.out.println(queue.poll());
    }

    // 阻塞等待
    public static void test3() throws InterruptedException {
        // 设置队列容量
        ArrayBlockingQueue queue = new ArrayBlockingQueue(3);

        queue.put("a");
        queue.put("b");
        queue.put("c");
        //      queue.put("d");// 一直阻塞

        System.out.println(queue.take());
        System.out.println(queue.take());
    }
}

```



```

        System.out.println(queue.take());
//        System.out.println(queue.take()); // 一直阻塞
    }

    // 阻塞超时
    public static void test4() throws InterruptedException {
        // 设置队列容量
        ArrayBlockingQueue queue = new ArrayBlockingQueue(3);

        System.out.println(queue.offer("a"));
        System.out.println(queue.offer("b"));
        System.out.println(queue.offer("c"));
//        System.out.println(queue.offer("d", 2, TimeUnit.SECONDS)); // 超时退出

        System.out.println(queue.poll());
        System.out.println(queue.poll());
        System.out.println(queue.poll());
        System.out.println(queue.poll(2, TimeUnit.SECONDS));
    }
}

```

## 同步队列 - SynchronousQueue

同步队列不同于其他的 BlockingQueue，SynchronousQueue 不存储元素，put 了一个元素，必须从里面先 take，否则不能再 put 进去

- 同步队列 SynchronousQueue put 了一个元素后该线程就会进入阻塞状态，直到调用了 take 或 poll 方法取出元素才会继续执行

```

package juc.queue;

import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/11 16:24
 */
public class TestSynchronousQueue {

    public static void main(String[] args) {
        // 同步队列
        SynchronousQueue<String> synchronousQueue = new SynchronousQueue<>();

        new Thread(() -> {
            try {
                System.out.println(Thread.currentThread().getName() + " -> put
1");
                synchronousQueue.put("1");
                System.out.println(Thread.currentThread().getName() + " -> put
2");
                synchronousQueue.put("2");
                System.out.println(Thread.currentThread().getName() + " -> put
3");
                synchronousQueue.put("3");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        })
    }
}

```

```

    }
    }, "A").start();

    new Thread(() -> {
        try {
            TimeUnit.SECONDS.sleep(2);
            System.out.println(Thread.currentThread().getName() + " get " +
synchronousQueue.take());
            TimeUnit.SECONDS.sleep(2);
            System.out.println(Thread.currentThread().getName() + " get " +
synchronousQueue.take());
            TimeUnit.SECONDS.sleep(2);
            System.out.println(Thread.currentThread().getName() + " get " +
synchronousQueue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }, "B").start();
}
}
}

```

## 线程池（重点）

线程池必会：三大方法、七大参数、四种拒绝策略

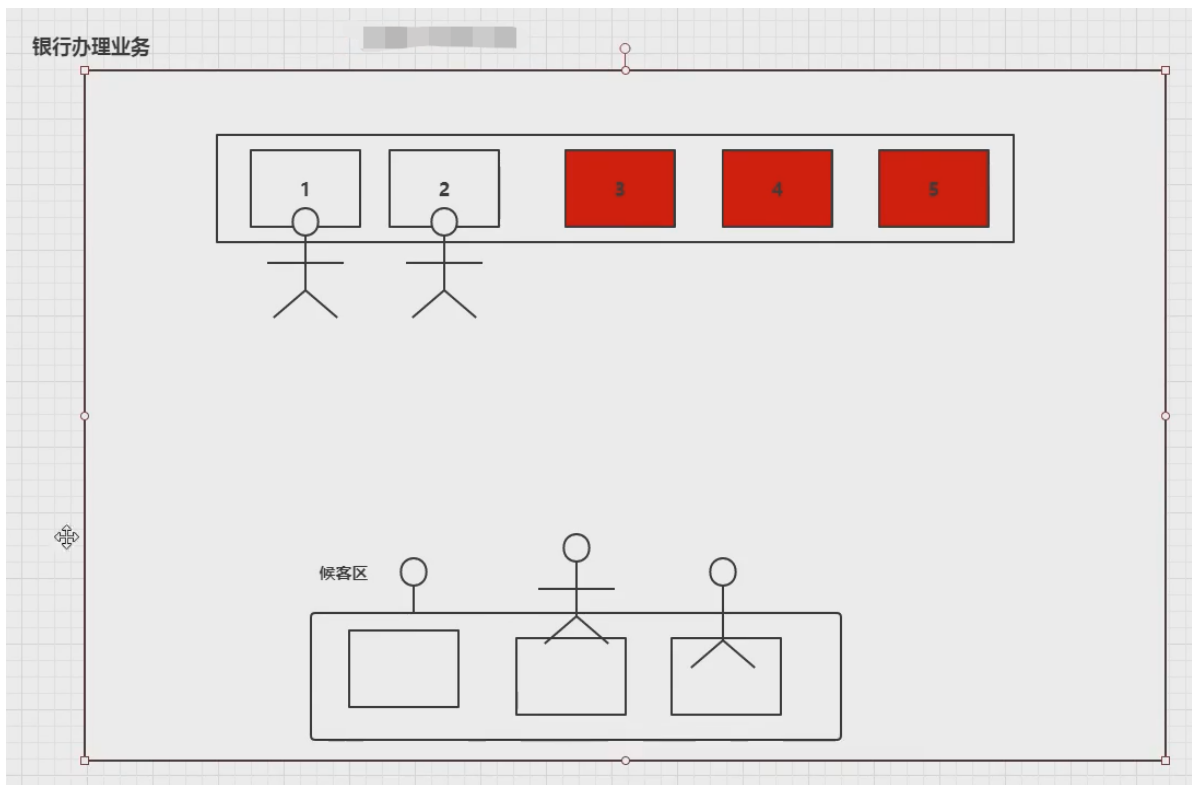
## 池化技术

程序运行的本质：占用系统资源 -> 优化资源的使用 = 》池化技术

线程池、JDBC连接池、内存池、对象池

池化技术：事先准备好一些资源

## 图解线程池



## 线程池的好处

创建、销毁线程十分浪费资源

1. 降低资源消耗
2. 提高响应速度
3. 方便管理

线程可复用、可以控制最大并发数、可以管理线程

## 线程池：三大方法

4. 【强制】线程池不允许使用 `Executors` 去创建，而是通过 `ThreadPoolExecutor` 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：`Executors` 返回的线程池对象的弊端如下：

- 1) `FixedThreadPool` 和 `SingleThreadPool`：  
允许的请求队列长度为 `Integer.MAX_VALUE`，可能会堆积大量的请求，从而导致 OOM。
- 2) `CachedThreadPool` 和 `ScheduledThreadPool`：  
允许的创建线程数量为 `Integer.MAX_VALUE`，可能会创建大量的线程，从而导致 OOM。

注意：线程池用完必须关闭

```
ExecutorService threadPool = Executors.newSingleThreadExecutor(); // 获得只包含一个线程的线程池
Executors.newFixedThreadPool(5); // 获得一个包含指定线程数的固定大小的线程池
Executors.newCachedThreadPool(); // 可伸缩的线程池
```

## 实例

```
package juc.pool;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * @author miemiehoho
 * @date 2022/2/11 17:31
 */
// Executors （工具类）
public class Demo {

    public static void main(String[] args) {
        //      ExecutorService threadPool = Executors.newSingleThreadExecutor();// 获得只包含一个线程的线程池
        //      ExecutorService threadPool = Executors.newFixedThreadPool(5);// 获得一个包含指定线程数的固定大小的线程池
        ExecutorService threadPool = Executors.newCachedThreadPool();// 可伸缩的线程池

        try {
            for (int i = 0; i < 10; i++) {
                threadPool.execute(() -> {
                    System.out.println(Thread.currentThread().getName());
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 线程池用完必须关闭
            threadPool.shutdown();
        }

    }
}
```

## 线程池：七大参数

### 源码分析

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}

public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                   0L, TimeUnit.MILLISECONDS,
                                   new LinkedBlockingQueue<Runnable>());
}

public static ExecutorService newCachedThreadPool() {
```

```

        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                       60L, TimeUnit.SECONDS,
                                       new SynchronousQueue<Runnable>());
    }

```

可以发现，这三个线程本质是调用了 `ThreadPoolExecutor`

```

public ThreadPoolExecutor(int corePoolSize, // 核心线程池大小
                          int maximumPoolSize, // 最大线程池大小
                          long keepAliveTime, // 存活时间（超过核心线程的线程闲置时的存
货时间）
                          TimeUnit unit, // 超时单位
                          BlockingQueue<Runnable> workQueue, // 阻塞队列
                          ThreadFactory threadFactory, // 线程工厂（创建线程的，一般不
用改）
                          RejectedExecutionHandler handler) { // 拒绝策略
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
        null :
        AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}

```

线程池中worker其实就是在某个线程中，不停的拿队列中的任务进行执行

## 手动创建线程池

```

ExecutorService threadPool = new ThreadPoolExecutor(2,
    5,
    3,
    TimeUnit.SECONDS,
    new LinkedBlockingQueue<>(3),
    Executors.defaultThreadFactory(),
    new ThreadPoolExecutor.DiscardOldestPolicy()
);

```

## 线程池：四种拒绝策略

- 默认拒绝策略: `new ThreadPoolExecutor.AbortPolicy()`
- `AbortPolicy()`
  - 满了，还进来，丢掉任务，抛出异常
- `CallerRunsPolicy()`
  - 哪里来的去哪里
- `DiscardPolicy()`
  - 满了，还进来，丢掉任务，也不抛出异常
- `DiscardOldestPolicy()`
  - 满了，还进来，抛弃队列里面最老的那个，代替他的位置进入队列里（源码的意思是：队列首部被poll出去，然后再调用execute方法尝试进入队列）

```
// 获取并忽略 executor 将执行的下一个任务，如果一个任务立即可用，然后重试任务 r 的执行，
// 除非 executor 被关闭，在这种情况下任务 r 被丢弃。
//参数：
//r - 请求执行的可运行任务
//e - 试图执行此任务的执行者
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    if (!e.isShutdown()) {
        e.getQueue().poll();
        e.execute(r);
    }
}
```

## 小结、拓展

### 最大线程到底该如何定义（调优）

1. CPU 密集型：几核就是几可以保证效率最高

- `System.out.println(Runtime.getRuntime().availableProcessors());` // 获取系统线程数（逻辑处理器）

2. IO 密集型：大于 当前程序中十分耗IO的线程的个数

## 四大函数式接口（必须掌握）

新时代程序员：lambda表达式、链式编程、函数式接口、Stream流式计算

**函数式接口：只有一个方法的接口**

例：

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

函数式接口可以简化编程模型，在新版本的框架底层大量应用

foreach 方法就是一个消费者类型的函数式接口

? compact3-package-summary  
? **Consumer**  
? DoubleBinaryOperator  
? DoubleConsumer  
? DoubleFunction  
? DoublePredicate  
? DoubleSupplier  
? DoubleToIntFunction  
? DoubleToLongFunction  
? DoubleUnaryOperator  
? **Function**  
? IntBinaryOperator  
? IntConsumer  
? IntFunction  
? IntPredicate  
? IntSupplier  
? IntToDoubleFunction  
? IntToLongFunction  
? IntUnaryOperator  
? LongBinaryOperator  
? LongConsumer  
? LongFunction  
? LongPredicate  
? LongSupplier  
? LongToDoubleFunction  
? LongToIntFunction  
? LongUnaryOperator  
? ObjDoubleConsumer  
? ObjIntConsumer  
? ObjLongConsumer  
? package-frame  
? package-summary  
? package-tree  
? package-us  
? **Predicate**  
? **Supplier**  
? ToDoubleBiFunction

**四大原生函数式接口**

## 实例

### Function 函数型接口

```
@FunctionalInterface
public interface Function<T, R> {

    Applies this function to the given argument
    Params: t – the function argument
    Returns: the function result

    R apply(T t);
}
```

```

/**
 * Function 函数型接口，有一个输入，有一个输出
 * 只要是 函数型接口，都可以用lambda表达式简化
 */
@param args
*/
public static void main(String[] args) {
    //      Function<String, String> function = new Function<String, String>() {
    //          @Override
    //          public String apply(String str) {
    //              return str;
    //          }
    //      };

    //      String str = function.apply("123");
    //      System.out.println(str);

    Function<String, String> function = (str) -> {return str;};
    //      Function<String, String> function = str -> {return str;};
    System.out.println(function.apply("123"));
}

```

## Predicate 函数式接口

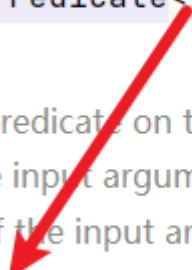
```

@FunctionalInterface
public interface Predicate<T> {

    Evaluates this predicate on the given a
    Params: t – the input argument
    Returns: true if the input argument m

    boolean test(T t);
}

```



```


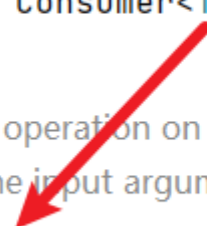
/**
 * 断定型接口，有一个输入参数，返回值只能是布尔值
 */
public static void main(String[] args) {
    //      Predicate<String> predicate = new Predicate<String>() {
    //          @Override
    //          public boolean test(String o) {
    //              return true;
    //          }
    //      };

    Predicate<String> predicate = (o)->{return true;};
    System.out.println(predicate.test("a"));
}

```



## Consumer 消费型接口

```
@FunctionalInterface
public interface Consumer<T> {
    
    
    Performs this operation on the given argument.
    Params: t – the input argument
    void accept(T t);
}
```

```
/**
 * Consumer 消费型接口，只有输入，没有返回值
 */
public static void main(String[] args) {
    // Consumer<String> consumer = new Consumer<String>() {
    //     @Override
    //     public void accept(String s) {
    //         System.out.println(s);
    //     }
    // };
    Consumer<String> consumer = (str) -> {
        System.out.println(str);
    };
    consumer.accept("abc");
}
```

## Supplier 供给型接口

```
@FunctionalInterface
public interface Supplier<T> {
    Gets a result.
    Returns: a result
    T get();
}
```

```

/**
 * Supplier 供给型接口，没有参数，只有返回值
 */
public static void main(String[] args) {
//      Supplier<String> supplier = new Supplier<String>() {
//          @Override
//          public String get() {
//              return "abc";
//          }
//      };
//      Supplier<String> supplier = ()->{return "abc";};
    System.out.println(supplier.get());
}

```

## Stream 流式计算

### 什么是Stream 流式计算

大数据 = 存储+计算

集合、MySQL 本质就是存储东西的；

计算都应该交给流来操作

### 实战

```

package juc.stream;

import java.util.Arrays;
import java.util.List;

/**
 * @author miemiehoho
 * @date 2022/2/11 21:39
 */
public class Test {

    /**
     * 题目要求：一分钟内完成此题目，只能用一行代码实现
     * 现在有5个用户，筛选：
     * 1.ID必须是偶数
     * 2.年龄必须大于23岁
     * 3.用户名转为大写字母
     * 4.用户名字母倒着排序
     * 5.只输出一个用户
     *
     * @param args
     */
    public static void main(String[] args) {
        User u1 = new User(1, "a", 21);
        User u2 = new User(2, "b", 22);
        User u3 = new User(3, "c", 23);
        User u4 = new User(4, "d", 24);
        User u5 = new User(6, "e", 25);
        // 集合就是存储
    }
}

```

```

List<User> users = Arrays.asList(u1, u2, u3, u4, u5);
// 计算交给Stream流
// lambda表达式、链式编程、函数式接口、Stream流式计算
users.stream()
    .filter((u) -> {return u.getId() % 2 == 0;})
    .filter((u)->{return u.getAge()>23;})
    .map((u)->{return u.getName().toUpperCase();})
    .sorted((uu1,uu2)->{return uu2.compareTo(uu1);})
    .limit(1)
    .forEach(System.out::println);
    }
}

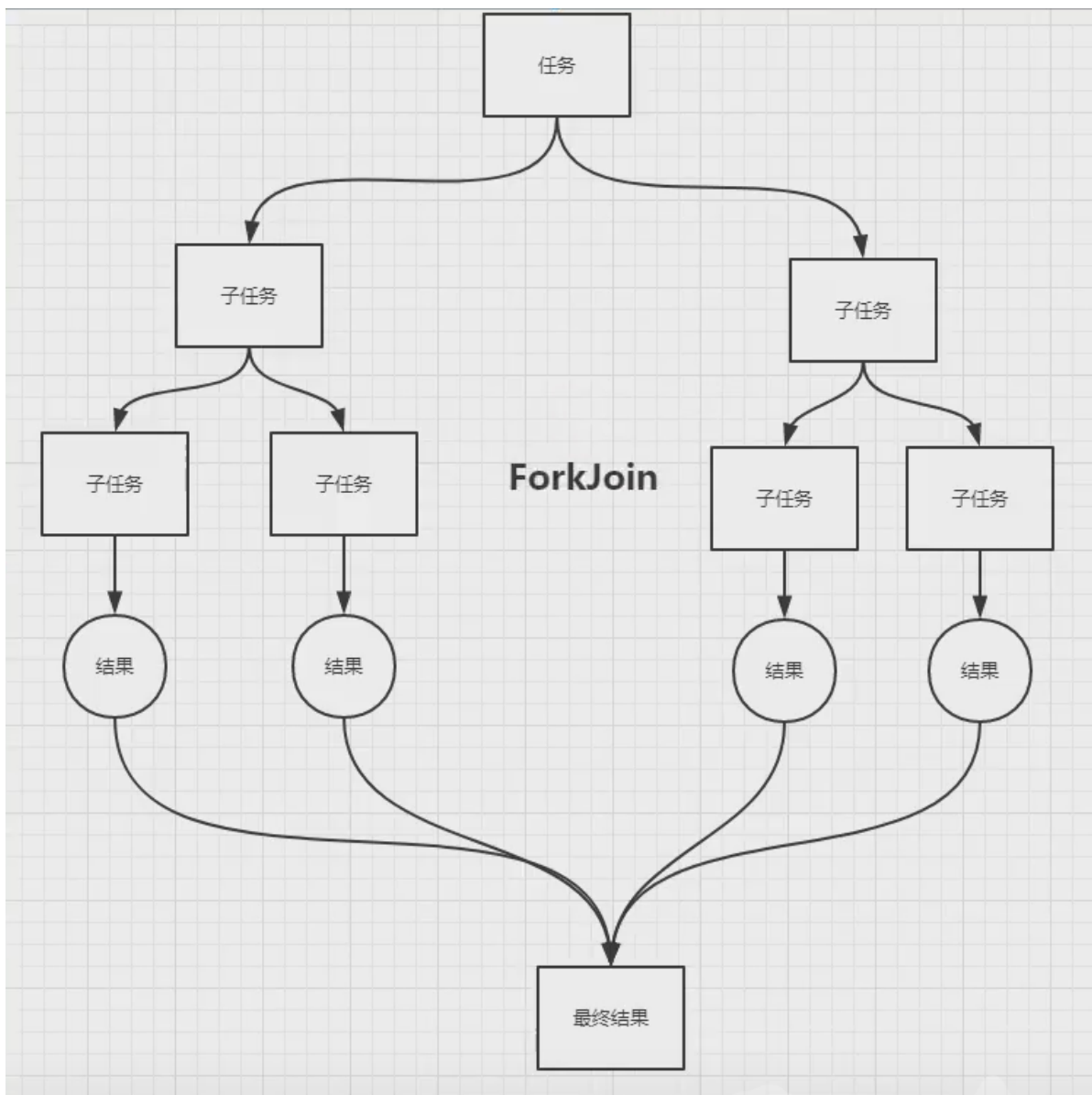
```

## ForkJoin

### 什么是 ForkJoin?

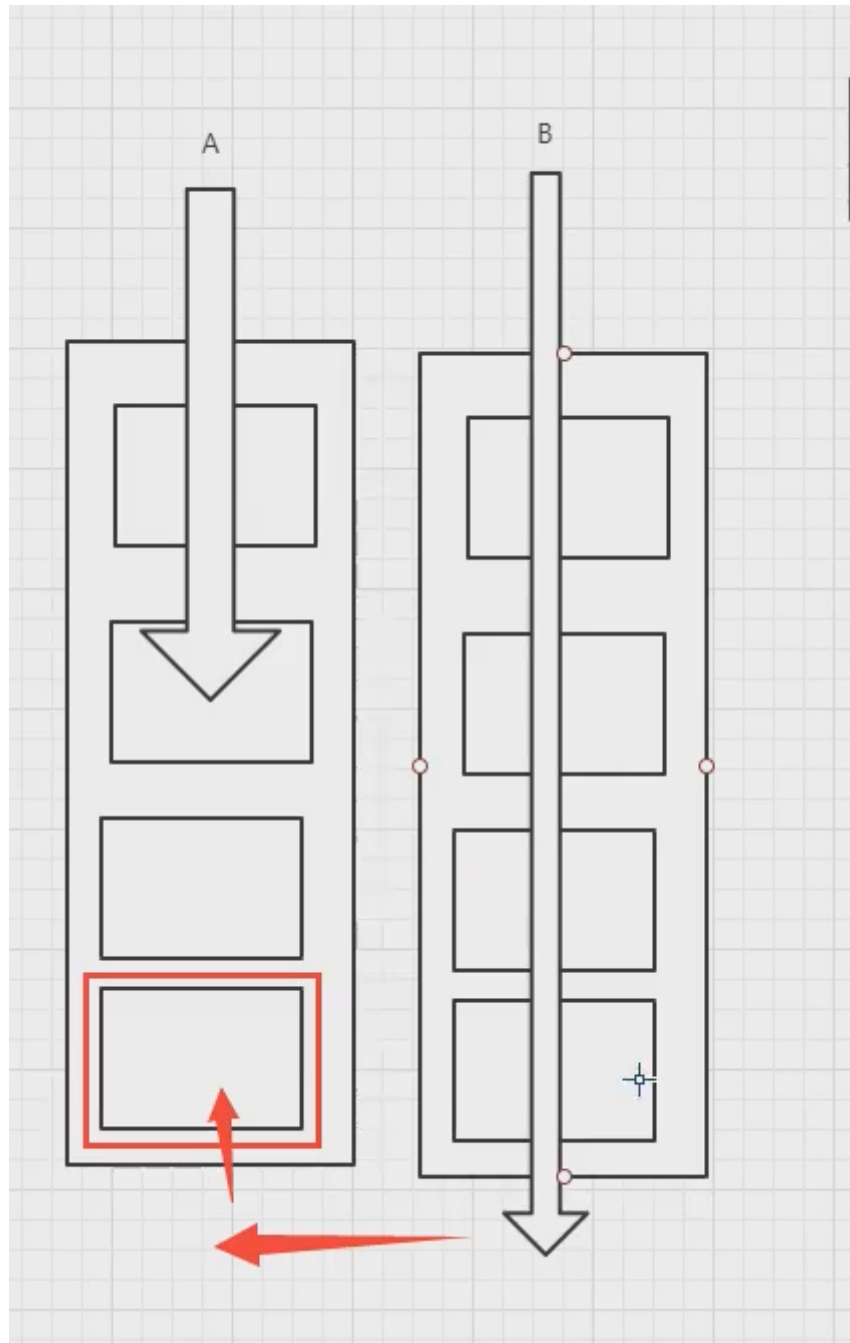
ForkJoin在JDK1.7，用于并行执行任务，提高效率，适用于**大数据量场景**

大数据：Map Reduce（把大任务拆分成小任务）



## ForkJoin 特点

ForkJoin 特点：工作窃取，这里面维护的都是双端队列



## ForkJoin 操作

### 如何使用 forkjoin?

- 1.使用 forkjionPool 执行
- 2.计算任务 forkjionPool.execute(ForkJoinTask<?> task)
- 3.计算类要继承RecursiveTask

void

`execute(ForkJoinTask<?> task)`

为异步执行给定任务的排列。

## Class `ForkJoinTask<V>`

`java.lang.Object`

`java.util.concurrent.ForkJoinTask<V>`

递归事件（没有返回值）

All Implemented Interfaces:

`Serializable` , `Future <V>`

已知直接子类:

`CountedCompleter` , `RecursiveAction` , `RecursiveTask`

递归任务（有返回值）

### 实例

```
package juc.forkjoin;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.stream.LongStream;
import java.util.stream.Stream;

/**
 * @author miemiehoho
 * @date 2022/2/11 23:32
 */
public class Test {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        long start = 0L;
        long end = 10_0000_0000L;
        test1(start, end);
        test2(start, end);
        test3(start, end);
    }

    public static void test1(long start, long end) {
        long sum = 0L;
        long begin = System.currentTimeMillis();
        for (long i = start; i <= end; i++) {
            sum += i;
        }
        long over = System.currentTimeMillis();
        System.out.println("test1(),sum = " + sum + "时间: " + (over - begin));
    }

    // forkjoin
    public static void test2(long start, long end) throws ExecutionException,
        InterruptedException {
        long begin = System.currentTimeMillis();
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        ForkJoinTask<Long> task = new ForkJoinDemo(start, end);
        ForkJoinTask<Long> result = forkJoinPool.submit(task);
        // forkJoinPool.execute();// 同步提交
        long sum = result.get();
        long over = System.currentTimeMillis();
    }
}
```

```

        System.out.println("test2(),sum = " + sum + "时间: " + (over - begin));
    }

    // stream 并行流
    public static void test3(long start, long end) {
        long begin = System.currentTimeMillis();
        // rangeClosed() 左右闭区间
        // Long::sum 代表调用 Long 的 sum 方法
        long sum = LongStream.rangeClosed(start, end).parallel().reduce(0,
Long::sum);
        long over = System.currentTimeMillis();
        System.out.println("test3(),sum = " + sum + "时间: " + (over - begin));
    }
}

```

```

package juc.forkjoin;

import java.util.List;
import java.util.concurrent.RecursiveTask;

/**
 * @author miemiehoho
 * @date 2022/2/11 23:09
 */
// 求和的计算任务

/**
 * 如何使用 forkjoin
 * 1. 使用 forkjionPool 执行
 * 2. 计算任务 forkjionPool.execute(ForkJoinTask<?> task)
 * 3. 计算类要继承RecursiveTask
 */
public class ForkJoinDemo extends RecursiveTask<Long> {

    private long start;
    private long end;
    private long temp = 10000L; // 临界值, 可以调节 (调优)

    public ForkJoinDemo(long start, long end) {
        this.start = start;
        this.end = end;
    }

    // 计算方法
    @Override
    protected Long compute() {
        if ((end - start) <= temp) {
            long sum = 0L;
            for (long i = start; i < end; i++) {
                sum += i;
            }
            return sum;
        } else {
            // forkjoin
            // 分支合并计算
            long middle = (start + end) / 2; // 中间值
            ForkJoinDemo task1 = new ForkJoinDemo(start, middle);

```

```

        task1.fork();// 拆分任务, 把任务压入线程队列
        ForkJoinDemo task2 = new ForkJoinDemo(middle + 1, end);
        task2.fork();// 拆分任务, 把任务压入线程队列
        return task1.join() + task2.join();
    }
}
}

```

## 异步回调

**Future** 的设计初衷, 对将来的某个事件的结果进行建模

java.util.concurrent

### Interface Future<V>

参数类型

V - 未来的 get方法返回的结果类型

All Known Subinterfaces:

Response <T>, RunnableFuture <V>, RunnableScheduledFuture <V>, ScheduledFuture <

所有已知实现类:

CompletableFuture, CountedCompleter, ForkJoinTask, FutureTask, RecursiveAction, RecursiveTask, SwingWorker

### Class CompletableFuture<T>

java.lang.Object

java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

CompletionStage <T>, Future <T>

```

public class CompletableFuture<T>
    extends Object
    implements Future<T>, CompletionStage<T>

```

用Future可以明确地完成(设定其值和状态), 并且可以被用作CompletionStage, 支持有关的功能和它的完成时触发动作。

当两个或多个线程试图complete, completeExceptionally, 或cancel一个CompletableFuture, 只有一个成功。

```

package juc.future;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/13 18:21
 */

/**
 * 异步调用: CompletableFuture<T>
 * 异步执行
 * 成功回调
 * 失败回调
 */
public class Demo {

```

```

        public static void main(String[] args) throws ExecutionException,
        InterruptedException {
            //          // 发起一个请求（没有返回结果的异步回调）
            //          CompletableFuture<Void> completableFuture =
            CompletableFuture.runAsync(() -> {
                //          try {
                //          TimeUnit.SECONDS.sleep(3);
                //
                System.out.println(Thread.currentThread().getName()+"Async=>Void");
                //          } catch (InterruptedException e) {
                //          e.printStackTrace();
                //          }
                //          });
                //
                //          System.out.println("main Thread");
                //          // 阻塞，获取执行结果
                //          completableFuture.get();

                // 有返回结果的异步回调
                CompletableFuture<Integer> completableFuture =
                CompletableFuture.supplyAsync(() -> {
                    System.out.println(Thread.currentThread().getName() +
                    "Async=>Integer");
                    int temp = 10 / 0;
                    return 1024;
                });

                System.out.println(completableFuture.whenComplete((t, u) -> {
                    System.out.println("t=>" + t); // 正常的返回结果
                    System.out.println("u=>" + u); // 错误信息
                }).exceptionally((e) -> {
                    System.out.println(e.getMessage());
                    return 222333; // 错误的返回结果
                }).get());

                // success code
                // error code

            }
        }
    }

```

## JMM

### 请你谈谈对 Volatile 的理解

Volatile 是 java 虚拟机提供的轻量级的 同步机制

1. 保证可见性
2. 不保证原子性
3. 禁止指令重排



# 什么是JMM(java memory model)

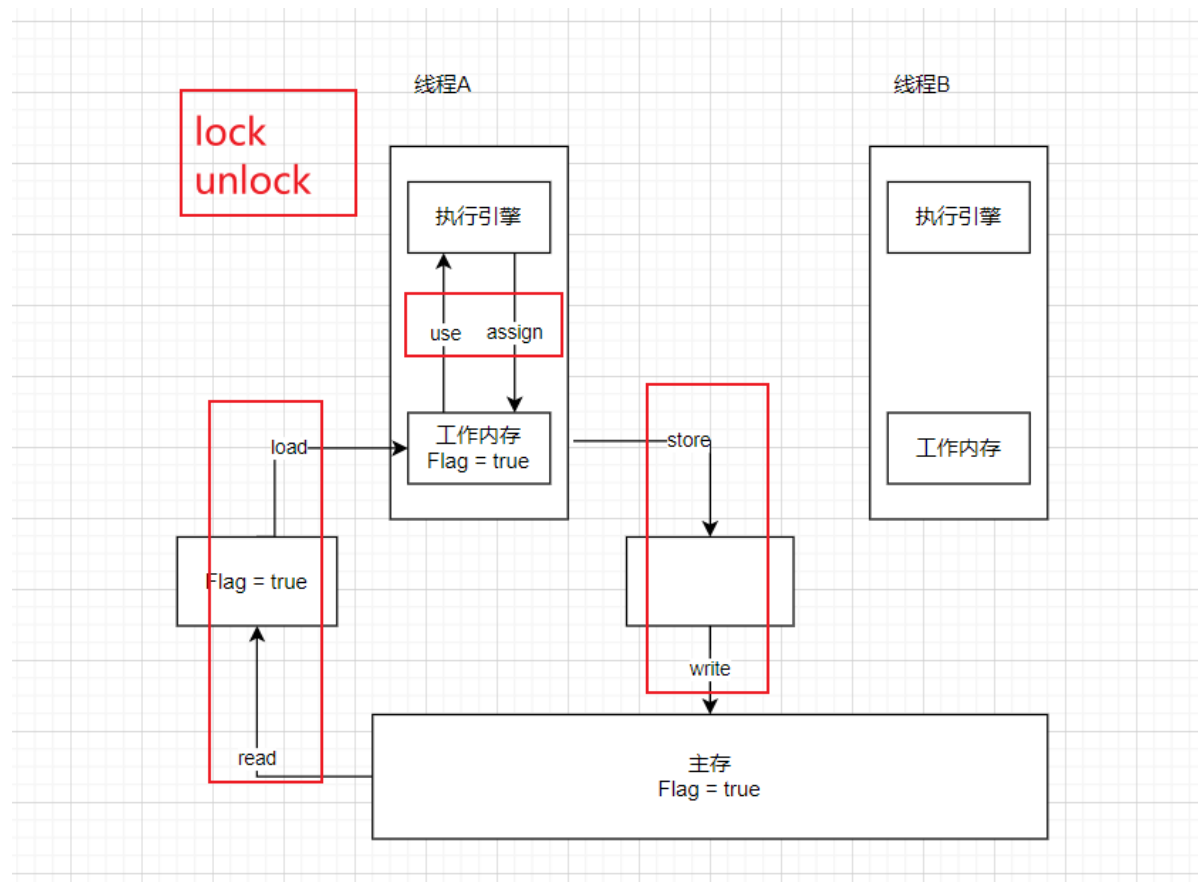
JMM: java内存模型, 不存在的东西, 是一种概念、约定

## 关于JMM的一些的同步约定

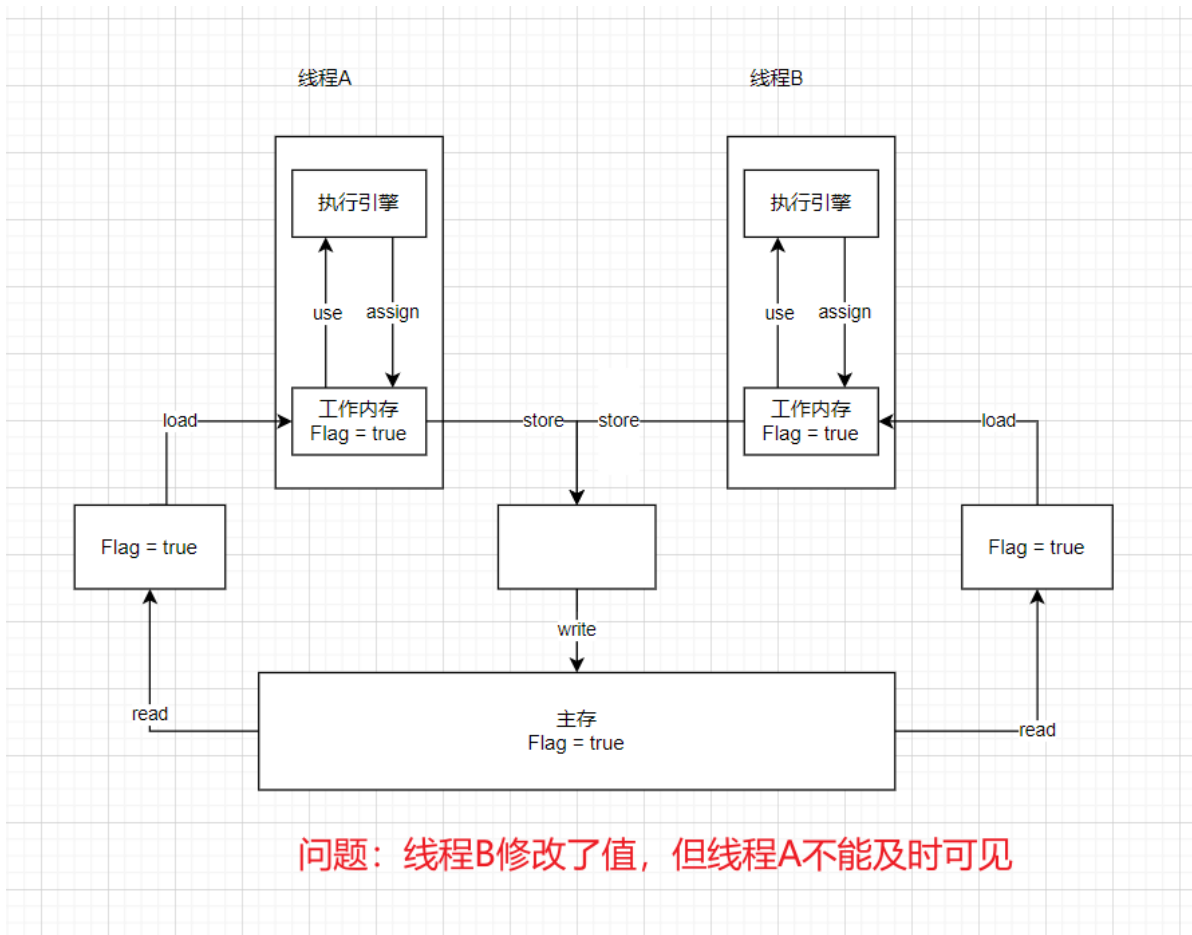
1. 线程解锁前, 必须把共享变量 **立刻** 刷回贮存
2. 线程加锁前, 必须读取主存中的最新值到工作内存中
3. 加锁和解锁是同一把锁

线程 工作内存 主存

8种操作:



存在的问题:



**内存交互操作有8种，虚拟机实现必须保证每一个操作都是原子的，不可在分的**

- lock（锁定）：作用于主内存的变量，把一个变量标识为线程独占状态
- unlock（解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
- read（读取）：作用于主内存变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用
- load（载入）：作用于工作内存的变量，它把read操作从主存中变量放入工作内存中
- use（使用）：作用于工作内存中的变量，它把工作内存中的变量传输给执行引擎，每当虚拟机遇到一个需要使用到变量的值，就会使用到这个指令
- assign（赋值）：作用于工作内存中的变量，它把一个从执行引擎中接受到的值放入工作内存的变量副本中
- store（存储）：作用于主内存中的变量，它把一个从工作内存中一个变量的值传送到主内存中，以便后续的write使用
- write（写入）：作用于主内存中的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中

**JMM对这八种指令的使用，制定了如下规则：**

- 不允许read和load、store和write操作之一单独出现。即使用了read必须load，使用了store必须write
- 不允许线程丢弃他最近的assign操作，即工作变量的数据改变了之后，必须告知主存
- 不允许一个线程将没有assign的数据从工作内存同步回主内存
- 一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。就是对变量实施use、store操作之前，必须经过assign和load操作

- 一个变量同一时间只有一个线程能对其进行lock。多次lock后，必须执行相同次数的unlock才能解锁
- 如果对一个变量进行lock操作，会清空所有工作内存中此变量的值，在执行引擎使用这个变量前，必须重新load或assign操作初始化变量的值
- 如果一个变量没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的变量
- 对一个变量进行unlock操作之前，必须把此变量同步回主内存

## 案例

**存在的问题：**程序不知道 主内存中的值已经被修改过了：

```
package juc.test_volatile;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/13 23:56
 */
public class Demo {

    private static int num = 0;

    public static void main(String[] args) {

        new Thread(() -> {
            while (num == 0) {

            }
        }).start();

        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        num = 1;
        System.out.println(num);
    }
}
```

## Volatile

### 1、保证可见性

MESI（缓存一致性协议）？

```
package juc.test_volatile;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/13 23:56
 */
```

```

*/
public class Demo {

    // 不加 volatile 程序就会死循环
    // 加 volatile 可以保证可见性
    private volatile static int num = 0;

    public static void main(String[] args) {

        new Thread(() -> {
            while (num == 0) {

            }
        }).start();

        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        num = 1;
        System.out.println(num);
    }
}

```

## 2、不保证原子性

### 什么是原子性？

原子性,也就是不可分割：线程A在执行任务的时候，不能被打扰的，要么同时成功，要么同时失败。

```

package juc.test_volatile;

/**
 * @author miemiehoho
 * @date 2022/2/14 16:28
 */
public class Demo2 {

    // volatile 不保证原子性
    private volatile static int num = 0;

    // synchronized 可以保证原子性
    public static void add() {
        num++;
    }

    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    add();
                }
            }).start();
        }
    }

    // java 中默认有两个线程在执行的：main、gc

```

```

while (Thread.activeCount() > 2) {
    Thread.yield(); // 线程礼让
    // yield就是当前线程把自己的cpu调度时间让给其它线程
    // yield就是礼让，让main线程放弃CPU给别的线程，这样main线程就不会执行下去，等到
    其它线程都死了，main线程再往下执行
    // Thread.yield()是在主线程中执行的，意思只要还有除了GC和main线程之外的线程在
    跑，主线程就让出cpu不往下执行
}

System.out.println(Thread.currentThread().getName() + " " + num);
}
}

```

## 如何不加 synchronized 或 lock 怎么保证原子性

*// volatile 不保证原子性*

```

public class VDemo02 {

    // volatile 不保证原子性
    private volatile static int num

    public static void add(){
        num++; // 不是获得这个值操作
    }

    public static void main(String[] args) {
        //理论上num结果应该为 2 万
    }
}

```

*2、+ 1*  
*3、写回这个值*

Bytecode for `add()`:

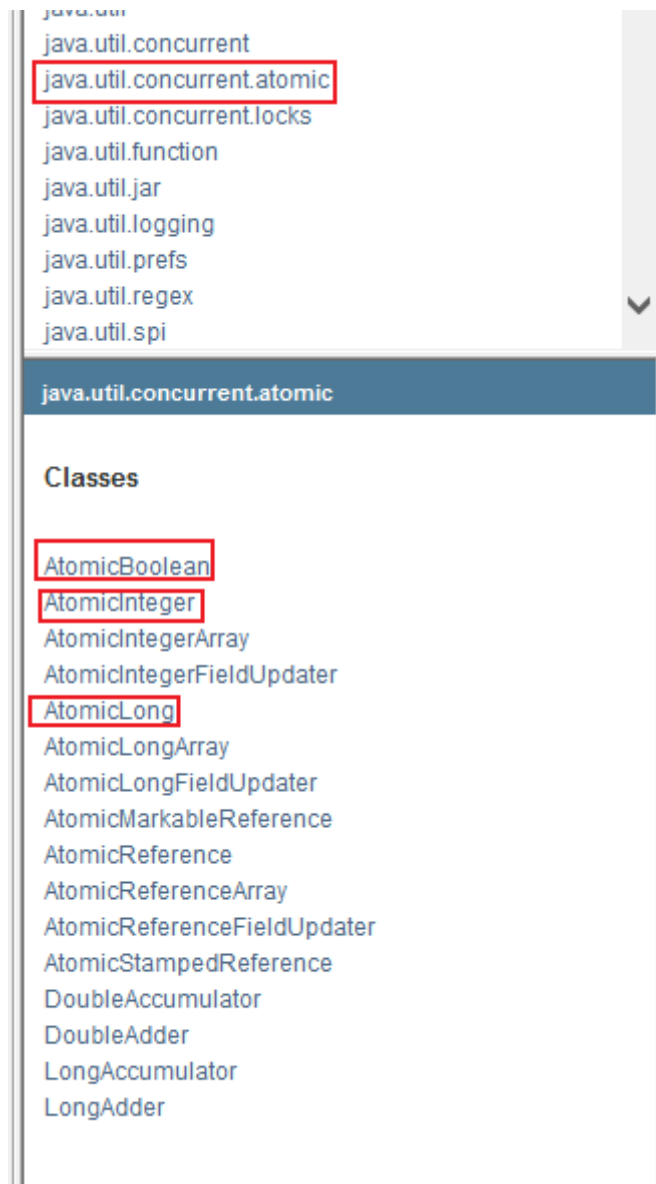
```

Code:
  0: aload_0
  1: invokespecial #1          // Method java/lang/Integer.valueOf:Ljava/lang/Integer;
  4: return

public static void add():
    Code:
      0: getstatic     #2          // Field num:I
      3: iconst_1
      4: iadd
      5: putstatic     #2          // Field num:I
      8: return

```

答案：使用 原子类解决原子性问题



```
package juc.test_volatile;

import java.util.concurrent.atomic.AtomicInteger;

/**
 * @author miemiehoho
 * @date 2022/2/14 16:28
 */
public class Demo2 {

    // volatile 不保证原子性
    private static AtomicInteger num = new AtomicInteger();

    // synchronized 可以保证原子性
    public static void add() {
        // num++; // num++ 并不是原子性操作
        num.getAndIncrement(); // AtomicInteger的 +1 方法,并不是简单的 +1操作, 底层用的是 CAS
    }

    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
```

```

        add();
    }
}).start();
}

// java 中默认有两个线程在执行的: main、gc
while (Thread.activeCount() > 2) {
    Thread.yield();// 线程礼让
    // yield就是当前线程把自己的cpu调度时间让给其它线程
    // yiled就是礼让, 让main线程放弃CPU给别的线程, 这样main线程就不会执行下去, 等到
    其它线程都死了, main线程再往下执行
    // Thread.yield()是在主线程中执行的, 意思只要还有除了GC和main线程之外的线程在
    跑, 主线程就让出cpu不往下执行
}

System.out.println(Thread.currentThread().getName() + " " + num);
}
}

```

这些原子类的底层都和操作系统挂钩, 直接在内存中修改值, Unsafe 类是一个很特殊的存在

## 原子类为什么这么高级?

### 3、禁止指令重排

#### 什么是指令重排?

你写的程序, 计算机并不是按照你写的那样去执行的:

源代码 -> 编译器优化的重排 -> 指令并行也可能会重排 -> 内存系统也会重排 -> 执行

处理器在进行指令重排的时候, 会考虑数据之间的依赖性

```

int x = 1; // 1
int y = 2; // 2
x = x + 5; // 3
y = x * x; // 4

```

我们所期望的执行顺序: 1234

可以运行的其他顺序: 2134、1324, 但不可能是 4123, 因为存在有数据依赖, 处理器在进行指令重排的时候, 会考虑数据之间的依赖性

指令重排就是指: 在不影响结果的前提下, 对某些代码优先执行, 用于提高效率

#### 可能造成影响的案例:

a、b、x、y, 默认都是 0

线程A	线程B
x = a	y = b
b = 1	a = 2

正常的结果: x = 0, y = 0, 但是可能由于指令重排

线程A	线程B
b = 1	a = 2
x = a	y = b

指令重排导致的诡异结果：x = 2,y = 1

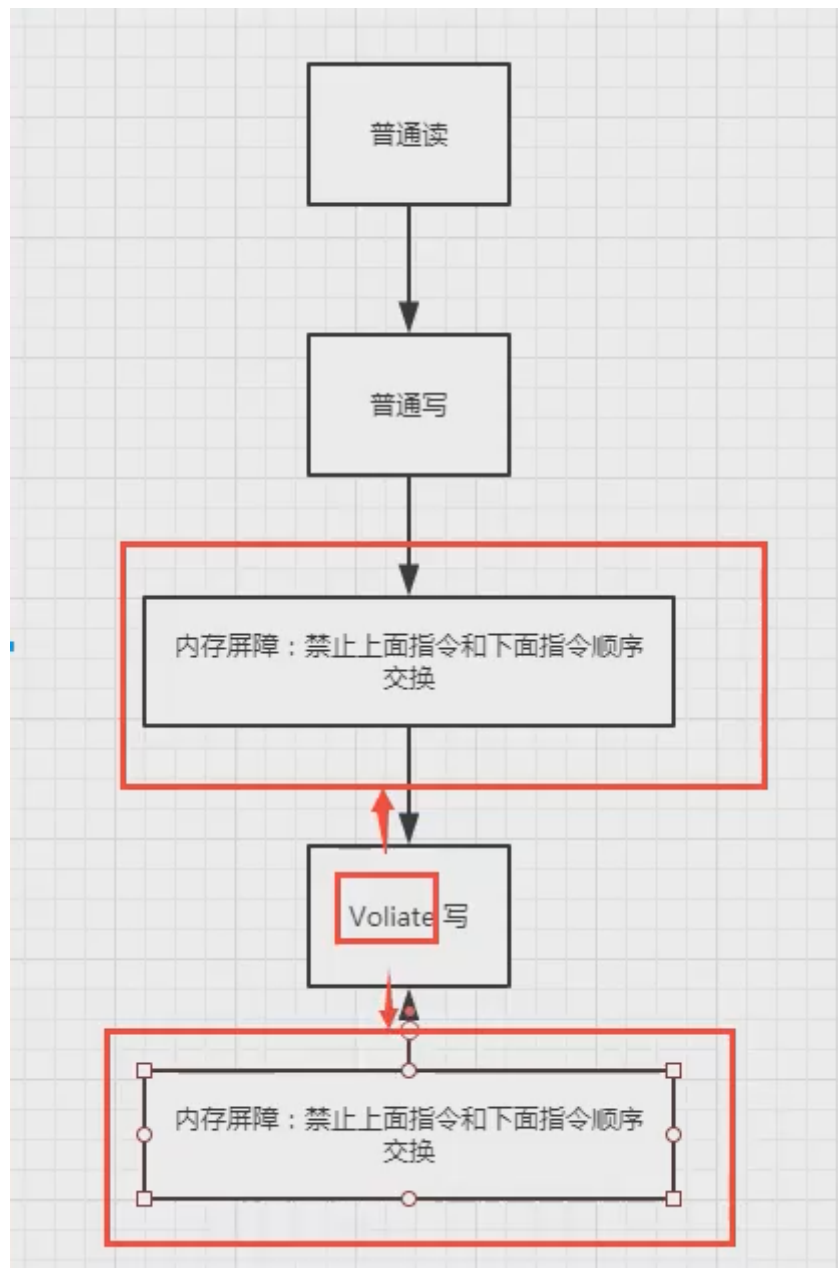
加上volatile会在编译后的代码块前后加上内存屏障，禁止指令重排

**volatile 可以避免指令重排：**

内存屏障（CPU指令），作用：

1. 保证特定的操作的执行顺序
2. 可以保证某些变量的内存可见性（利用这些特性volatile实现了可见性）

画图解释：对于执行不同操作的多个线程：



Volatile 可以保证可见性，不能保证原子性，由于内存屏障，可以避免指令重排的现象产生



volatile 内存屏障 在单例模式中使用的最多

# 彻底玩转单例模式

---

## 单例模式

### 创建单例模式的三种方式

1. 饿汉式
2. 懒汉式、DCL懒汉式（使用了 volatile）
3. 静态内部类

但是这三种方式都是不安全的，因为存在反射

### 什么是单例模式

在有些系统中，为了节省内存资源、保证数据内容的一致性，对某些类要求只能创建一个实例，这就是所谓的单例模式。

### 单例模式的定义、特点、应用

#### 定义

单例（Singleton）模式的定义：指一个类只有一个实例，且该类能自行创建这个实例的一种模式。例如，Windows 中只能打开一个任务管理器，这样可以避免因打开多个任务管理器窗口而造成内存资源的浪费，或出现各个窗口显示内容的不一致等错误。

#### 特点

单例模式有 3 个特点：

1. 单例类只有一个实例对象；
2. 该单例对象必须由单例类自行创建；
3. 单例类对外提供一个访问该单例的全局访问点。

#### 应用

在计算机系统中，还有 Windows 的回收站、操作系统中的文件系统、多线程中的线程池、显卡的驱动程序对象、打印机的后台处理服务、应用程序的日志对象、数据库的连接池、网站的计数器、Web 应用的配置对象、应用程序中的对话框、系统中的缓存等常常被设计成单例。

单例模式在现实生活中的应用也非常广泛，例如公司 CEO、部门经理等都属于单例模型。J2EE 标准中的 [ServletContext](#) 和 [ServletContextConfig](#)、[Spring](#) 框架应用中的 [ApplicationContext](#)、数据库中的连接池等也都是单例模式。

### 单例模式的优点和缺点

单例模式的优点：

- 单例模式可以保证内存里只有一个实例，减少了内存的开销。
- 可以避免对资源的多重占用。
- 单例模式设置全局访问点，可以优化和共享资源的访问。

单例模式的缺点：

- 单例模式一般没有接口，扩展困难。如果要扩展，则除了修改原来的代码，没有第二种途径，违背开闭原则。

- 在并发测试中，单例模式不利于代码调试。在调试过程中，如果单例中的代码没有执行完，也不能模拟生成一个新的对象。
- 单例模式的功能代码通常写在一个类中，如果功能设计不合理，则很容易违背单一职责原则。

单例模式看起来非常简单，实现起来也非常简单。单例模式在面试中是一个高频面试题。希望大家能够认真学习，掌握单例模式，提升核心竞争力，给面试加分，顺利拿到 Offer。

## 单例模式的应用场景

对于 [Java](#) 来说，单例模式可以保证在一个 JVM 中只存在单一实例。单例模式的应用场景主要有以下几个方面。

- 需要频繁创建的一些类，使用单例可以降低系统的内存压力，减少 GC。
- 某类只要求生成一个对象的时候，如一个班中的班长、每个人的身份证号等。
- 某些类创建实例时占用资源较多，或实例化耗时较长，且经常使用。
- 某类需要频繁实例化，而创建的对象又频繁被销毁的时候，如多线程的线程池、网络连接池等。
- 频繁访问数据库或文件的对象。
- 对于一些控制硬件级别的操作，或者从系统上来讲应当是单一控制逻辑的操作，如果有多个实例，则系统会完全乱套。
- 当对象需要被共享的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。

## 单例模式的结构

单例模式是[设计模式](#)中最简单的模式之一。通常，普通类的构造函数是公有的，外部类可以通过“new 构造函数()”来生成多个实例。但是，如果将类的构造函数设为私有的，外部类就无法调用该构造函数，也就无法生成多个实例。这时该类自身必须定义一个静态私有实例，并对外提供一个静态的公有函数用于创建或获取该静态私有实例。

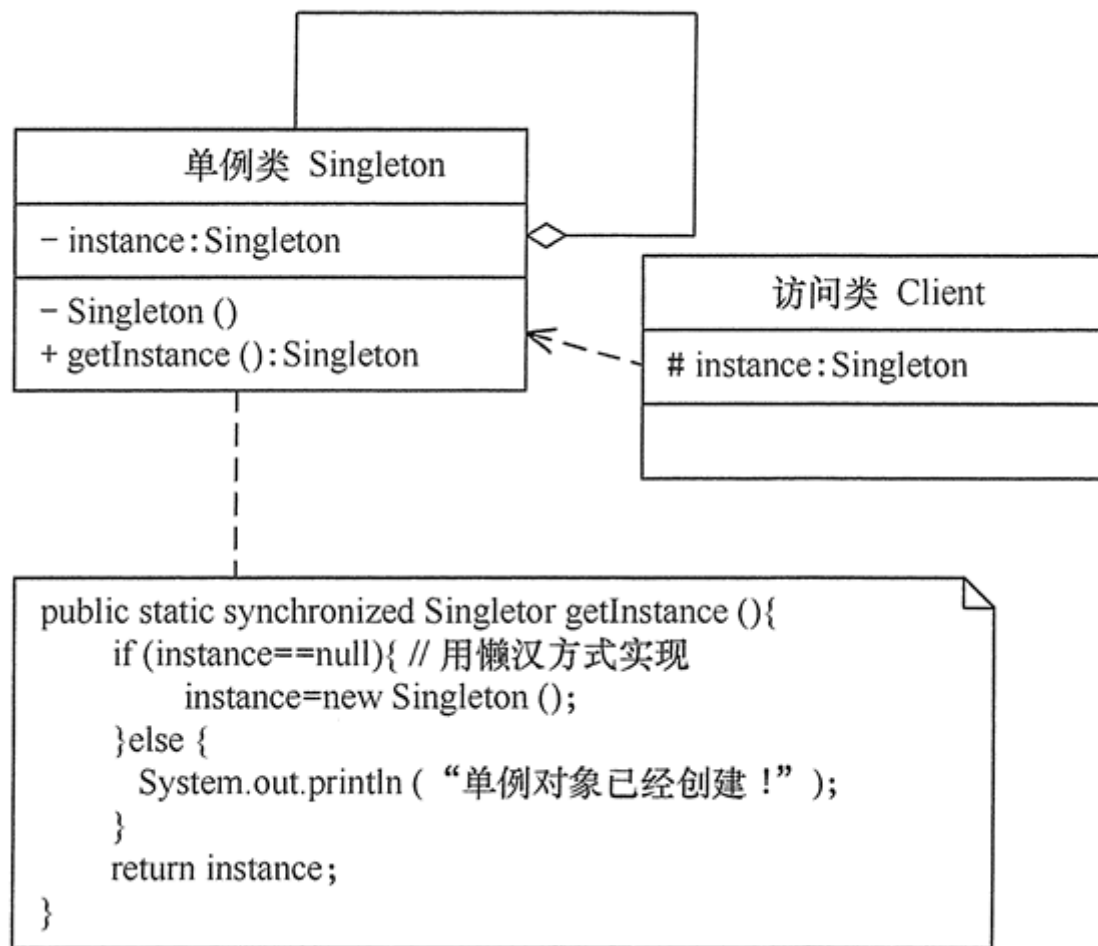
下面来分析其基本结构和实现方法。

### 1. 单例模式的结构

单例模式的主要角色如下。

- 单例类：包含一个实例且能自行创建这个实例的类。
- 访问类：使用单例的类。

其结构如图 1 所示。



## 2. 单例模式的实现

Singleton 模式通常有两种实现形式：饿汉式单例、懒汉式单例

### 饿汉式

该模式的特点是类一旦加载就创建一个单例，保证在调用 `getInstance` 方法之前单例已经存在了。

```
public class HungrySingleton {
    private static final HungrySingleton instance = new HungrySingleton();

    private HungrySingleton() {
    }

    public static HungrySingleton getInstance() {
        return instance;
    }
}
```

### 饿汉式单例为什么是线程安全的？

饿汉式单例在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以是线程安全的，可以直接用于多线程而不会出现问题。

## 饿汉式单例存在浪费内存的问题

在调用静态方法之前，确实没有初始化，但是由JVM类加载模式可以知道，符号引用在解析阶段就要变成直接引用，所以已经加载了该对象，如果后来不使用(即初始化)空间确实浪费了

```
package juc.single;

/**
 * @author miemiehoho
 * @date 2022/2/14 17:18
 */
// 饿汉式单例
public class Hungry {

    private byte[] data1 = new byte[1024 * 1024];
    private byte[] data2 = new byte[1024 * 1024];
    private byte[] data3 = new byte[1024 * 1024];
    private byte[] data4 = new byte[1024 * 1024];

    private Hungry() {

    }

    private final static Hungry HUNGRY = new Hungry();

    public static Hungry getInstance() {
        return HUNGRY;
    }
}
```

## 其他问题

- 饿汉式单例在类首次加载的时候才会创建；
- 在调用静态方法之前，确实没有初始化，但是由JVM类加载模式可以知道，符号引用在解析阶段就要变成直接引用，所以已经加载了该对象，如果后来不使用(即初始化)空间确实浪费了
- 静态变量是随着类的加载就已经实例化了，跟是否调用该静态方法没有关系

## 懒汉式

该模式的特点是类加载时没有生成单例，只有当第一次调用 getInstance 方法时才去创建这个单例。代码如下：

```
public class LazySingleton {
    private static volatile LazySingleton instance = null;    //保证 instance 在所有线程中同步

    private LazySingleton() {
    }    //private 避免类在外部被实例化

    public static synchronized LazySingleton getInstance() {
        //getInstance 方法前加同步
        if (instance == null) {
            instance = new LazySingleton();
        }
    }
}
```

```

    }
    return instance;
}
}

```

注意：如果编写的是多线程程序，则不要删除上例代码中的关键字 `volatile` 和 `synchronized`，否则将存在线程非安全的问题。如果不删除这两个关键字就能保证线程安全，但是每次访问时都要同步，会影响性能，且消耗更多的资源，这是懒汉式单例的缺点。

## DCL懒汉式

双重检测锁模式的懒汉式单例，即DCL懒汉式单例

```

package juc.single;

/**
 * @author miemiehoho
 * @date 2022/2/14 18:08
 */
// 懒汉式单例
public class LazyDemo {

    private LazyDemo() {

    }

    private volatile static LazyDemo lazyDemo = null;

    // 双重检测锁模式的懒汉式单例，即DCL懒汉式单例
    public synchronized LazyDemo getInstance() {
        if (lazyDemo == null) {
            synchronized (LazyDemo.class) { // 锁 class
                if (lazyDemo == null) {
                    lazyDemo = new LazyDemo(); // 这个代码在极端情况下是可能出现问题的，
                    // 因为这不是一个原子性操作，会经历三个步骤
                    /**
                     * 1. 分配内存空间
                     * 2. 执行构造方法，初始化对象
                     * 3. 把这个对象指向这个空间
                     */
                    // 在执行这三步的过程中，可能会发生指令重排的现象，会导致顺序可能变成
                    132,
                    // 对于多线程环境，在 A线程没有执行完 2 的时候，如果B线程运行，那就不会
                    走if 代码块中的代码，
                    // 此时 lazyDemo还没有完成构造，就可能产生问题，因此需要加 volatile

                }
            }
        }
        return lazyDemo;
    }
}

```

## DCL懒汉式单例为什么不直接在方法上加锁？

因为只有在创建的时候会抢夺锁，其他的线程仅仅一个读取的操作都需要等待锁，会造成效率低下的问题。所以就是说，锁的范围越小越好

## 为什么DCL懒汉式单例需要 volatile 关键字？

```
// 双重检测锁模式的懒汉式单例，即DCL 懒汉式单例
public synchronized LazyDemo getInstance() {
    if (lazyDemo == null) {
        synchronized (LazyDemo.class) { // 锁 class
            if (lazyDemo == null) {
                lazyDemo = new LazyDemo(); // 这个代
            }
        }
    }
    return lazyDemo;
}
```

这个代码在极端情况下是可能出现问题的，  
因为这不是一个原子性操作

代码 `lazyDemo = new LazyDemo();` 在极端情况下是可能出现问题的，因为这不是一个原子性操作，会经历三个步骤：

1. 分配内存空间
2. 执行构造方法，初始化对象
3. 把这个对象指向这个空间

在执行这三步的过程中，**可能会发生指令重排**的现象，会导致顺序可能变成 132，对于多线程环境，在A线程没有执行完 2 的时候，如果B线程运行，那就不会走 if 代码块中的代码，**此时 lazyDemo还没有完成构造**，就可能产生问题，因此需要加 volatile

注意第三步执行完对象就已将不为空了，所以多线程下，就会直接获取对象的值，但是对象还没有执行2这个操作

## 为什么DCL懒汉式单例需要 synchronized 关键字？

这里双重检测加锁是保证了操作原子性，只有一个线程能创建一个实例，其他线程无法创建第二个；volatile关键字是为了防止因为指令重排导致的多线程问题，有可能线程A创建一个实例，虚拟机只执行了分配空间，对象地址引用这两步，这是线程B过来发现对象已经被创建了，但是获取到的对象是还没有被初始化的

new关键字就是申请一个内存空间，构造方法就是利用该空间将对象给初始化，最后是等号把该空间的内存地址赋值给声明，这块空间是在堆内存中，而声明则在本地方法栈里。

## 静态内部类

```
package juc.single;

/**
 * @author miemiehoho
 * @date 2022/2/14 18:38
 */
```

```

*/
// 静态内部类
public class Holder {

    private Holder() {

    }

    public static Holder getInstance() {
        return InnerClass.HOLDER;
    }

    public static class InnerClass {
        private static final Holder HOLDER = new Holder();
    }
}

```

## 反射

### 问题一：

可以通过反射创建对象，破坏单例模式

```

public static void main(String[] args) throws NoSuchMethodException,
InvocationTargetException, InstantiationException, IllegalAccessException {
    LazyDemo instance1 = LazyDemo.getInstance();

    // 获得空参的构造器
    Constructor<LazyDemo> constructor =
LazyDemo.class.getDeclaredConstructor(null);
    constructor.setAccessible(true); // 通过这个方法可以无视私有的构造器，然后就可以通过反
射创建对象
    LazyDemo instance2 = constructor.newInstance();

    System.out.println(instance1.toString());
    System.out.println(instance2.toString());
}

```

解决方式：三重检测

```

package juc.single;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

/**
 * @author miemiehoho
 * @date 2022/2/14 18:08
 */
// 懒汉式单例
public class LazyDemo {

    private LazyDemo() {
        synchronized (LazyDemo.class) {

```

```

        if (lazyDemo != null) {
            throw new RuntimeException("不要试图使用反射破坏单例");
        }
    }
}

private volatile static LazyDemo lazyDemo = null;

// 为什么不直接在方法上加锁？ 因为只有在创建的时候会抢夺锁，其他的线程仅仅一个读取的操作都需要等待锁，会造成效率低下的问题。所以就是说，锁的范围越小越好

// 双重检测锁模式的懒汉式单例，即DCL懒汉式单例
public static synchronized LazyDemo getInstance() {
    if (lazyDemo == null) {
        synchronized (LazyDemo.class) { // 锁 class
            if (lazyDemo == null) {
                lazyDemo = new LazyDemo(); // 这个代码在极端情况下是可能出现问题的，
                // 因为这不是一个原子性操作，会经历三个步骤
            }
        }
    }
    return lazyDemo;
}
}

```

## 问题二：

不通过 getInstance() 方法创建对象

```

public static void main(String[] args) throws NoSuchMethodException,
InvocationTargetException, InstantiationException, IllegalAccessException {
    // 获得空参的构造器
    Constructor<LazyDemo> constructor =
        LazyDemo.class.getDeclaredConstructor(null);
    constructor.setAccessible(true); // 通过这个方法可以无视私有的构造器，然后就可以通过反
    // 射创建对象
    LazyDemo instance1 = constructor.newInstance();
    LazyDemo instance2 = constructor.newInstance();

    System.out.println(instance1.toString());
    System.out.println(instance2.toString());
}

```

## 解决方式：红绿灯

设置一个私有的关键字，作为红绿灯，如果对方不通过反编译是找不到这个私有的关键字的，而且这个关键字还可以进一步做一些加密处理

```

package juc.single;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

```



```

/**
 * @author miemiehoho
 * @date 2022/2/14 18:08
 */
// 懒汉式单例
public class LazyDemo {

    // 红绿灯
    private static boolean flag = false;

    private LazyDemo() {
        synchronized (LazyDemo.class) {
            if (flag == false) {
                flag = true;
            } else {
                throw new RuntimeException("不要试图使用反射破坏单例");
            }
        }
    }

    private volatile static LazyDemo lazyDemo = null;

    // 为什么不直接在方法上加锁？ 因为只有在创建的时候会抢夺锁，其他的线程仅仅一个读取的操作都需要等待锁，会造成效率低下的问题。所以就是说，锁的范围越小越好

    // 双重检测锁模式的懒汉式单例，即DCL懒汉式单例
    public static synchronized LazyDemo getInstance() {
        if (lazyDemo == null) {
            synchronized (LazyDemo.class) { // 锁 class
                if (lazyDemo == null) {
                    lazyDemo = new LazyDemo(); // 这个代码在极端情况下是可能出现问题的，
                    // 因为这不是一个原子性操作，会经历三个步骤
                }
            }
        }
        return lazyDemo;
    }
}

```

### 问题三：

红绿灯自动被破解

```

public static void main(String[] args) throws NoSuchMethodException,
InvocationTargetException, InstantiationException, IllegalAccessException,
NoSuchFieldException {
    // 拿到 flag 字段
    Field flag = LazyDemo.class.getDeclaredField("flag");
    flag.setAccessible(true); // 破坏私有权限

    // 获得空参的构造器
    Constructor<LazyDemo> constructor =
        LazyDemo.class.getDeclaredConstructor(null);
}

```

```

        constructor.setAccessible(true);// 通过这个方法可以无视私有的构造器，然后就可以通过反射创建对象
        LazyDemo instance1 = constructor.newInstance();

        flag.set(instance1, false);
        LazyDemo instance2 = constructor.newInstance();

        System.out.println(instance1.toString());
        System.out.println(instance2.toString());
    }

```

道高一尺魔高一丈啊!

## 到底如何解决反射问题？为什么枚举可以避免单例模式被破坏？

### 什么是枚举？

- Java 枚举是一个特殊的类，一般表示一组常量，比如一年的 4 个季节，一个年的 12 个月份，一个星期的 7 天，方向有东南西北等。
- Java 枚举类使用 enum 关键字来定义，各个常量使用逗号，来分割。
- 枚举产生于 JDK1.5, 自带单例模式

### 源码分析

constructor.newInstance() 方法源码分析：

**如果 类是枚举类型，抛出异常：不能使用反射破坏枚举**

```

@CallerSensitive
public T newInstance(Object ... initargs)
    throws InstantiationException, IllegalAccessException,
           IllegalArgumentException, InvocationTargetException
{
    if (!override) {
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
            Class<?> caller = Reflection.getCallerClass();
            checkAccess(caller, clazz, null, modifiers);
        }
    }
    // 如果 类是枚举类型，抛出异常：不能使用反射破坏枚举
    if ((clazz.getModifiers() & Modifier.ENUM) != 0)
        throw new IllegalArgumentException("Cannot reflectively create enum objects");
    ConstructorAccessor ca = constructorAccessor; // read volatile
    if (ca == null) {
        ca = acquireConstructorAccessor();
    }
    @SuppressWarnings("unchecked")
    T inst = (T) ca.newInstance(initargs);
    return inst;
}

```

实例：

enum 是什么？ 本身也是一个Class类

### 1、新建 枚举类

```
/**
 * @author miemiehoho
 * @date 2022/2/14 19:15
 */
// enum 是什么？ 本身也是一个Class类
public enum EnumSingle{

    INSTANCE;

    public EnumSingle getInstance(){
        return INSTANCE;
    }
}
```

### 2、利用IDEA 查看 新建的枚举类生成的 .class 文件

```
public enum EnumSingle {
    INSTANCE;

    private EnumSingle() {
    }

    public EnumSingle getInstance() {
        return INSTANCE;
    }
}
```

可以看到 存在 无参的构造方法，

### 3、尝试利用反射创建 EnumSingle 实例

```
public static void main(String[] args) throws InvocationTargetException,
InstantiationException, IllegalAccessException, NoSuchMethodException {
    EnumSingle instance1 = EnumSingle.INSTANCE;
    Constructor<EnumSingle> constructor =
EnumSingle.class.getDeclaredConstructor(null);
    constructor.setAccessible(true);// 破除私有权限
    EnumSingle instance2 = constructor.newInstance();
    System.out.println(instance1);
    System.out.println(instance2);
}
```

输出：

报 不存在无参构造方法的异常

```
Exception in thread "main" java.lang.NoSuchMethodException:
juc.single.EnumSingle.<init>()
    at java.lang.Class.getConstructor0(Class.java:3082)
    at java.lang.Class.getDeclaredConstructor(Class.java:2178)
    at juc.single.Test.main(EnumSingle.java:24)
```

#### 4、反编译查看 EnumSingle.class 文件

```
javap -p EnumSingle.class
```

查看反编译代码可以发现，仍存在一个空参的构造方法

```
D:\13686\OneDrive -
365word\miemiehoho\Project\practice\out\production\practice\juc\single>javap -p
EnumSingle.class
Compiled from "EnumSingle.java"
public final class juc.single.EnumSingle extends
java.lang.Enum<juc.single.EnumSingle> {
    public static final juc.single.EnumSingle INSTANCE;
    private static final juc.single.EnumSingle[] $VALUES;
    public static juc.single.EnumSingle[] values();
    public static juc.single.EnumSingle valueOf(java.lang.String);
    private juc.single.EnumSingle();
    public juc.single.EnumSingle getInstance();
    static {};
}
```

#### 5、利用专业的jad 工具反编译

<https://varaneckas.com/jad/>

```
jad -sjava EnumSingle.class
```

反编译生成源码：

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   EnumSingle.java

package juc.single;

public final class EnumSingle extends Enum
{

    public static EnumSingle[] values()
    {
        return (EnumSingle[])$VALUES.clone();
    }

    public static EnumSingle valueOf(String name)
    {
        return (EnumSingle)Enum.valueOf(juc/single/EnumSingle, name);
    }

    private EnumSingle(String s, int i)
```

```

{
    super(s, i);
}

public EnumSingle getInstance()
{
    return INSTANCE;
}

public static final EnumSingle INSTANCE;
private static final EnumSingle $VALUES[];

static
{
    INSTANCE = new EnumSingle("INSTANCE", 0);
    $VALUES = (new EnumSingle[] {
        INSTANCE
    });
}
}

```

发现，不存在无参的构造方法，存在一个含参的构造方法：

```

private EnumSingle(String s, int i)
{
    super(s, i);
}

```

## 6、尝试利用反射创建含参的 EnumSingle 实例

```

public static void main(String[] args) throws InvocationTargetException,
InstantiationException, IllegalAccessException, NoSuchMethodException {
    EnumSingle instance1 = EnumSingle.INSTANCE;
    Constructor<EnumSingle> constructor =
EnumSingle.class.getDeclaredConstructor(String.class, int.class);
    constructor.setAccessible(true);// 破除私有权限
    EnumSingle instance2 = constructor.newInstance();
    System.out.println(instance1);
    System.out.println(instance2);
}

```

报错：

```

Exception in thread "main" java.lang.IllegalArgumentException: Cannot
reflectively create enum objects
    at java.lang.reflect.Constructor.newInstance(Constructor.java:417)
    at juc.single.Test.main(EnumSingle.java:26)

```

**Cannot reflectively create enum objects**，即不能利用反射创建枚举单例

## 总结:

枚举的最终反编译源码:

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   EnumSingle.java

package juc.single;

public final class EnumSingle extends Enum
{

    public static EnumSingle[] values()
    {
        return (EnumSingle[])$VALUES.clone();
    }

    public static EnumSingle valueOf(String name)
    {
        return (EnumSingle)Enum.valueOf(juc/single/EnumSingle, name);
    }

    private EnumSingle(String s, int i)
    {
        super(s, i);
    }

    public EnumSingle getInstance()
    {
        return INSTANCE;
    }

    public static final EnumSingle INSTANCE;
    private static final EnumSingle $VALUES[];

    static
    {
        INSTANCE = new EnumSingle("INSTANCE", 0);
        $VALUES = (new EnumSingle[] {
            INSTANCE
        });
    }
}
```

## 深入理解 CAS

---

# 什么是 CAS

CAS：比较当前工作内存中的值和主内存中的值，如果这个值是期望的，那么则执行操作；如果不是就一直循环（底层是自旋锁）

缺点：

1. 由于底层是自旋锁，所以循环会耗时
2. 一次性只能保证一个共享变量的原子性
3. 存在ABA问题

java 层面的CAS： `compareAndSet` 比较并交换

底层的 CAS： `compareAndSwapInt` 比较并交换

```
package juc.cas;

import java.util.concurrent.atomic.AtomicInteger;

/**
 * @author miemiehoho
 * @date 2022/2/14 21:33
 */
public class Demo {

    // java 层面的cas: compareAndSet    比较并交换

    // CAS 是 CPU的并发原语
    public static void main(String[] args) {
        // 原子类底层用了 CAS
        AtomicInteger atomicInteger = new AtomicInteger(2020);

        // 期望 更新
        // public final boolean compareAndSet(int expect, int update) {
        System.out.println(atomicInteger.compareAndSet(2020, 2021)); // 如果期望的值
        // 达到了，就更新，否则不更新
        System.out.println(atomicInteger.get());

        System.out.println(atomicInteger.compareAndSet(2020, 2021));
        System.out.println(atomicInteger.get());
    }
}
```

## Unsafe类

Java无法操作内存，java 可以调用 C++，C++可以操作内存

Unsafe类相当于java的后门，可以通过这个类操作内存

```

public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;

```

Java无法操作内存,  
java 可以调用 C++,  
C++可以操作内存

Unsafe类相当于java的后门  
可以通过这个类操作内存

## 实例:

```

// CAS 是 CPU的并发原语
public static void main(String[] args) {
    // 原子类底层用了 CAS
    AtomicInteger atomicInteger = new AtomicInteger(2020);
    atomicInteger.getAndIncrement();
}

```

AtomicInteger 对应的方法源码:

```

// setup to use Unsafe.compareAndSwapInt for updates
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;

static {
    try {
        valueOffset = unsafe.objectFieldOffset// 获取内存地址偏移量
            (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
}

public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}

```

Unsafe 对应的方法源码:



```
// var1 当前对象
// var2 当前对象内存的值
// var4 1
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2); // 获取var1内存地址中的值
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

- `compareAndSwapInt(var1, var2, var5, var5 + var4)` 是内存操作，效率很高
- `compareAndSwapInt(var1, var2, var5, var5 + var4)`：如果当前对象var1 当前内存地址中的值依然是 var5,则更新为 var5 + var4
- `compareAndSwapInt(var1, var2, var5, var5 + var4)` 是底层的 CAS
- `public final int getAndAddInt(Object var1, long var2, int var4)` 方法源码是一段标准的自旋锁：不停旋转，直到值能够更新为止

## CAS：ABA问题（狸猫换太子）

这种问题如果是基本类型不会产生影响，但是如果是引用类型，Java是值传递，就会有影响了，传递的值可能没变但是引用的对象可能变了

ABA问题解决要靠加版本号来解决

对数值没影响，对引用有影响

Mysql乐观锁，不是说上了一把锁，其实没有上锁，只是用的老版本数据

乐观锁就是他认为别人没改过，悲观锁，他认为别人都改过

乐观锁是一种思想，就是数据更新的时候检测是否产生并发冲突。CAS是具体的实现

乐观锁有版本号、CAS这两种实现方式

乐观锁的实现方式：CAS和版本号机制

不解决ABA问题不解决，其他线程会改变你想改变线程的值

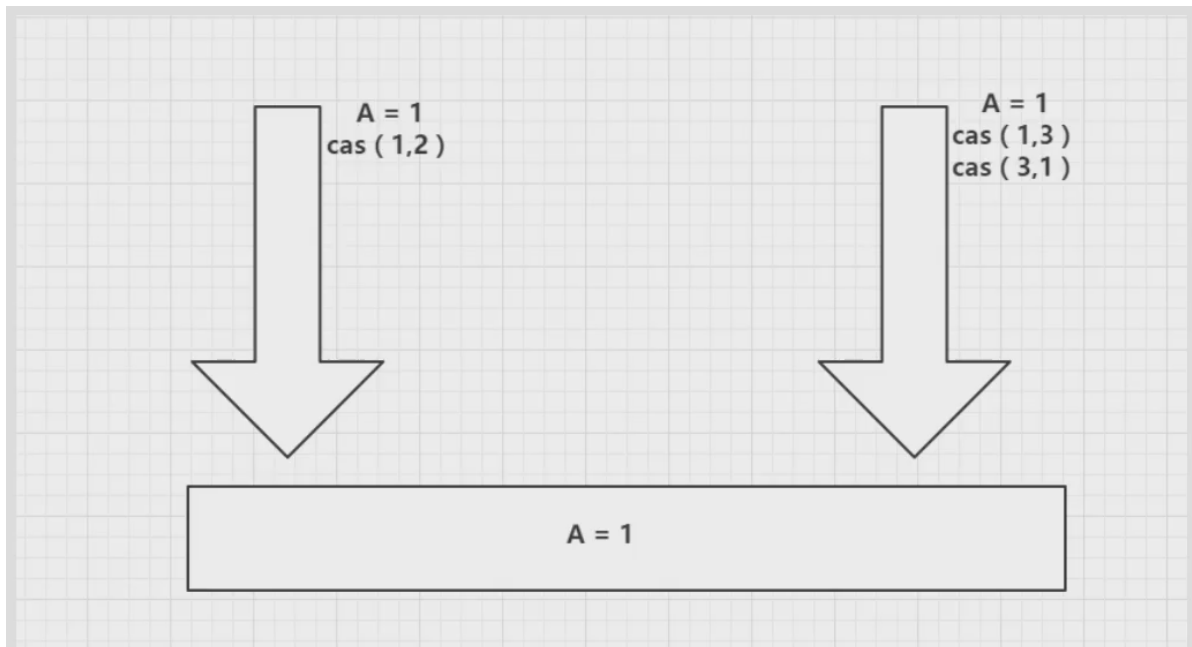
aba不解决资源会被提前挪用，这不是我们希望的

周志明的JVM实践中说：ABA问题大多时候不影响结果，如果确实要解决，我为什么不加锁解决呢？

悲观锁：

**实例：**

**ABA问题：**



```
public static void main(String[] args) {  
    // 原子类底层用了 CAS  
    AtomicInteger atomicInteger = new AtomicInteger(2020);  
  
    // 期望 更新  
    // public final boolean compareAndSet(int expect, int update) {  
    // ===== 捣乱的线程  
    System.out.println(atomicInteger.compareAndSet(2020, 2021)); // 如果期望的值达到了，就更新，否则不更新  
    System.out.println(atomicInteger.get());  
  
    System.out.println(atomicInteger.compareAndSet(2021, 2020)); // 如果期望的值达到了，就更新，否则不更新  
    System.out.println(atomicInteger.get());  
  
    // ===== 期望的线程  
    System.out.println(atomicInteger.compareAndSet(2020, 666666));  
    System.out.println(atomicInteger.get());  
}
```

## 原子引用 解决 ABA问题

### 带版本号的原子操作

解决 ABA 问题，引入原子引用。对应的思想：乐观锁

Integer 使用了对象缓存机制，默认范围是 -128 ~ 127，**推荐使用静态工厂方法 valueOf 获取对象实例**，而不是 new，因为 valueOf 使用缓存，而 new 一定会创建新的对象分配新的内存空间。

**Alibaba开发手册规约：**

【强制】所有的相同类型的包装类对象之间值的比较，全部使用 `equals` 方法比较。

说明：对于 `Integer var = ?` 在-128 至 127 范围内的赋值，`Integer` 对象是在 `IntegerCache.cache` 产生，会复用已有对象，这个区间内的 `Integer` 值可以直接使用`==`进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 `equals` 方法进行判断

## 代码

### 如果解决的ABA问题？

由于已经被修改过了，版本号发生了变化，所以 修改失败，无法改成期望值

```
package juc.cas;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicStampedReference;

/**
 * @author miemiehoho
 * @date 2022/2/14 21:33
 */
public class Demo {

    // java 层面的cas: compareAndSet    比较并交换

    // CAS 是 CPU的并发原语
    public static void main(String[] args) {
        // 原子类底层用了 CAS
        // AtomicInteger atomicInteger = new AtomicInteger(2020);
        // V initialRef    初始值
        // int initialStamp    初始时间戳（类似于版本号）
        // 每次修改都需要更新时间戳，以声明已经被修改了
        // 注意：如果泛型是一个包装类，注意引用问题
        // 正常业务中，这里比较的都是对象
        AtomicStampedReference<Integer> atomicInteger = new
AtomicStampedReference<>(1, 1);

        new Thread(() -> {
            int stamp = atomicInteger.getStamp(); // 获得版本号
            System.out.println("a1->" + stamp);
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            atomicInteger.compareAndSet(1, 2, atomicInteger.getStamp(),
atomicInteger.getStamp() + 1);
            System.out.println();
            System.out.println("a2->" + atomicInteger.getStamp());

            System.out.println(atomicInteger.compareAndSet(2, 1,
atomicInteger.getStamp(), atomicInteger.getStamp() + 1));
            System.out.println("a3->" + atomicInteger.getStamp());

        }, "a").start();
    }
}
```

```

// 由于已经被修改过了，版本号发生了变化，所以 修改失败，无法改成期望值 6
// 类似于乐观锁
new Thread(() -> {
    int stamp = atomicInteger.getStamp(); // 获得版本号
    System.out.println("b1->" + stamp);
    try {
        TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("修改: " + atomicInteger.compareAndSet(1, 6,
stamp, stamp + 1));
    System.out.println("b2->" + atomicInteger.getStamp());

}, "b").start();

}
}

```

Integer类，如果值的范围不在-128和127之间，就是不同的对象，由于这里比较的是对象的内存地址(==)，所以肯定是false

## 对各种锁的理解

通过上面学习，就是两个大锁，一个叫乐观锁，一个叫悲观锁

### 1、公平锁、非公平锁

- 公平锁：非常公平，不能够插队，必须先来后到
- 非公平锁：非常不公平，可以插队，默认都是非公平的

lock、synchronized默认也是非公平锁

```

public ReentrantLock() {
    sync = new NonfairSync();
}

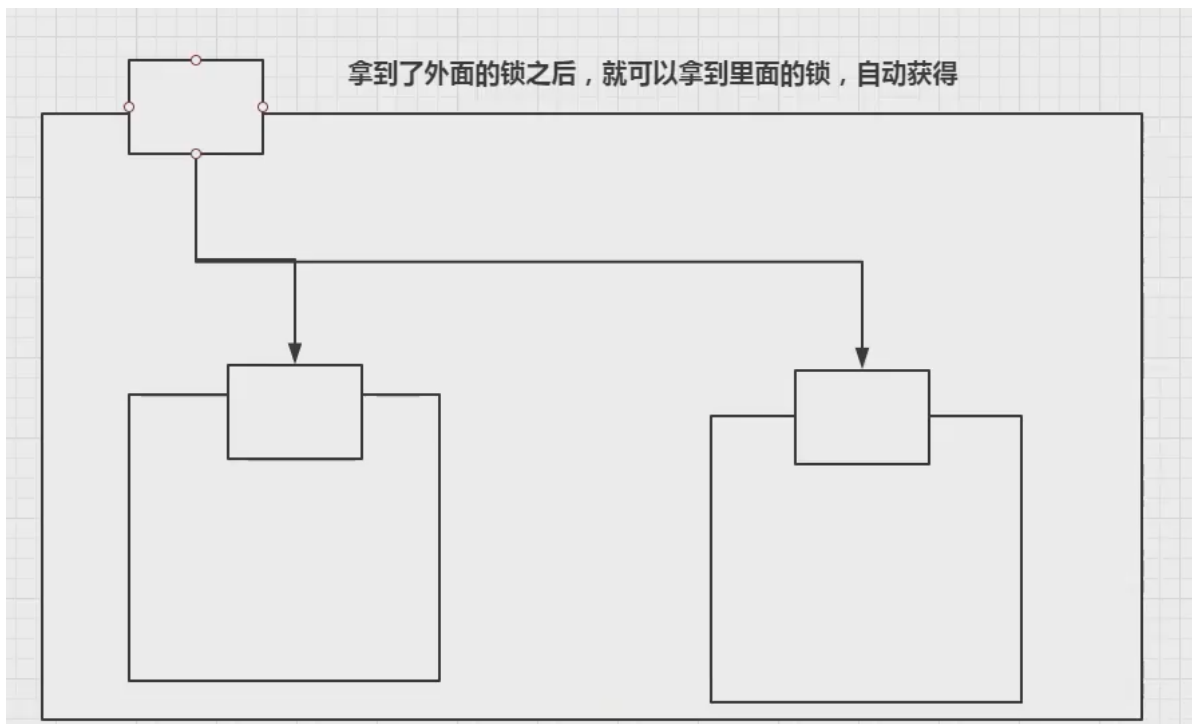
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}

```

### 2、可重入锁（递归锁）

所有的锁都是可重入锁

拿到外面的锁，就可以拿到里面的锁



## synchronized 可重入锁

```
package juc.lock;

/**
 * @author miemiehoho
 * @date 2022/2/15 12:40
 */
public class Demo1 {
    public static void main(String[] args) {
        iPhone iPhone = new iPhone();
        new Thread(() -> {
            iPhone.sms();
        }, "A").start();

        new Thread(() -> {
            iPhone.sms();
        }, "B").start();
    }
}

class iPhone {

    public synchronized void sms() {
        System.out.println(Thread.currentThread().getName() + " sms");
        call(); // 这里也有锁
    }

    public synchronized void call() {
        System.out.println(Thread.currentThread().getName() + " call");
    }
}
```

## lock 可重入锁

- 这就是一把锁，锁的就是ReentrantLock这个对象，ReentrantLock中有AQS，AQS可以判断两次lock方法都是同一个线程，这才是可重入锁
  - 可以对一个ReentrantLock对象多次执行lock()加锁和unlock()释放锁，也就是可以对一个锁加多次，叫做可重入加锁
  - 某一个线程已经获得了锁，还可以再次获得锁而不会出现死锁
  - 可重入锁是指：某个线程已经获取了某个锁，那么他可以再次获取该锁而不陷入死锁
  - 底层就是个计数器 每次获得锁就会计数器+1 解锁就是计数器-1 当计数器为0时就是空闲了
- 
- 锁的计数必须归零
  - lock锁必须显式的开锁和关锁 同时存在

```
package juc.lock;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @author miemiehoho
 * @date 2022/2/15 12:40
 */
public class Demo2 {
    public static void main(String[] args) {
        iPhone2 iphone = new iPhone2();
        new Thread(() -> {
            iphone.sms();
        }, "A").start();

        new Thread(() -> {
            iphone.sms();
        }, "B").start();
    }
}

class iPhone2 {
    Lock lock = new ReentrantLock();

    public void sms() {
        try {
            lock.lock(); // lock.lock(); lock.unlock(); lock锁必须配对，否则就会死锁
            System.out.println(Thread.currentThread().getName() + " sms");
            call(); // 这里也有锁
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void call() {
        try {
```

```

        lock.lock();
        System.out.println(Thread.currentThread().getName() + " call");
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}

```

### 3、自旋锁 (spin lock)

自旋锁会不断的去尝试，直到成功为止

源码示例：

Unsafe类的 `getAndAddInt` 方法

```

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}

```

### 自定义自旋锁

底层用 CAS实现自旋锁

测试代码中的两条线程：

- T1获得锁，T2自旋一直尝试获得。T1解锁后，T2才可以获得锁结束自旋，然后解锁
- 线程2在自旋，进来lock的时候，不是null，一直死循环
- T1加锁后，T2进入自旋，只有T1解锁后，T2才能加锁成功

```

package juc.lock;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

/**
 * @author miemiehoho
 * @date 2022/2/15 14:45
 */
// 自旋锁
// 底层是内存屏障保证一致性
// 底层使用的自旋锁（CAS）
public class SpinLockDemo {
    // Thread 引用类型，默认空值为 null
    AtomicReference<Thread> atomicReference = new AtomicReference<>();
}

```

```

// 加锁
public void myLock() {
    Thread thread = Thread.currentThread();
    System.out.println(thread.getName() + "-> myLock");
    // 自旋锁
    while (!atomicReference.compareAndSet(null, thread)) {

    }
}

// 解锁
public void myUnLock() {
    Thread thread = Thread.currentThread();
    System.out.println(thread.getName() + "-> myUnLock");
    atomicReference.compareAndSet(thread, null);
}

}

class Test {

    // T1获得锁, T2自旋一直尝试获得。T1解锁后, T2才可以获得锁结束自旋, 然后解锁
    // 线程2在自旋, 进来lock的时候, 不是null, 一直死循环
    // T1加锁后, T2进入自旋, 只有T1解锁后, T2才能加锁成功
    public static void main(String[] args) throws InterruptedException {
        SpinLockDemo lock = new SpinLockDemo();
        new Thread() -> {
            lock.myLock();
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                lock.myUnLock();
            }
        }, "T1").start();

        TimeUnit.SECONDS.sleep(1);

        new Thread() -> {
            lock.myLock();
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                lock.myUnLock();
            }
        }, "T2").start();

    }
}

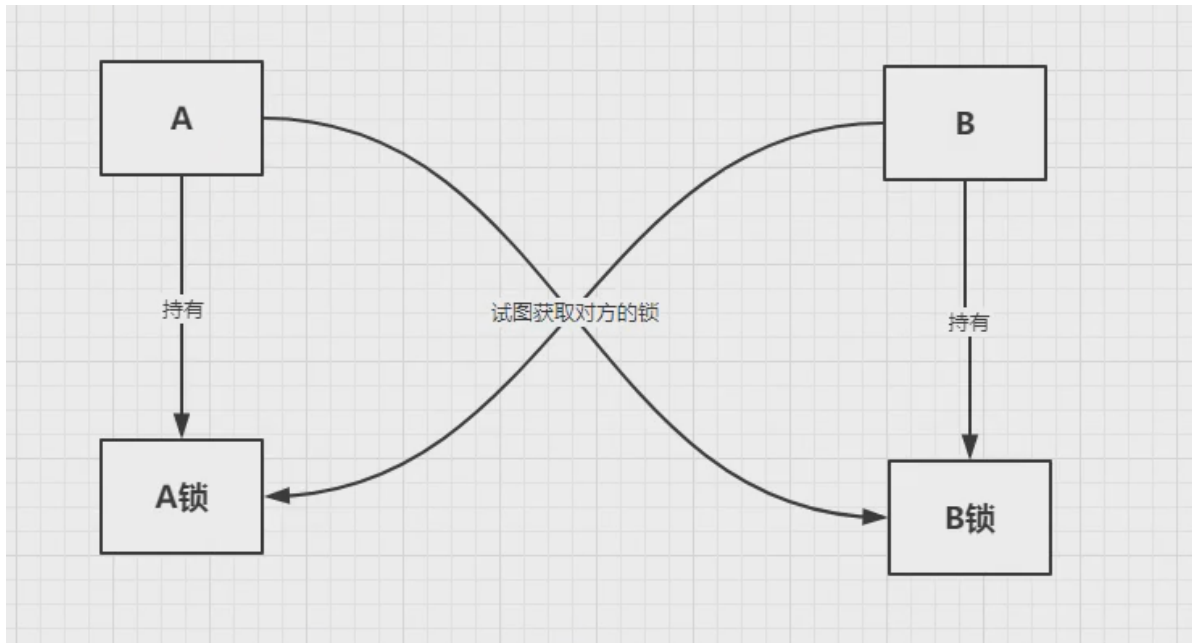
```



## 4、死锁

### 什么是死锁？

死锁：多个线程互相抱着对方的资源，形成僵持



### 死锁四要素、死锁产生的4个必要条件

死锁产生的4个必要条件：

- 1、**互斥**：某种资源一次只允许一个进程访问，即该资源一旦分配给某个进程，其他进程就不能再访问，直到该进程访问结束。
- 2、**占有且等待**：一个进程本身占有资源（一种或多种），同时还有资源未得到满足，正在等待其他进程释放该资源。
- 3、**不可抢占**：别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来。
- 4、**循环等待**：存在一个进程链，使得每个进程都占有下一个进程所需的至少一种资源。

### 死锁 案例

这里虽然new了两个Tread，但是因为锁的是string，在常量池（方法区）（可以理解为整个程序共享），所以两个线程锁的是同一个string对象，也就是同一把锁。（想到这个例子真厉害！！）

因为锁的对象是String类型的，是在常量池里面找。所以说**一般不推荐使用String作为锁的对象**

```
package juc.lock;

import java.util.concurrent.TimeUnit;

/**
 * @author miemiehoho
 * @date 2022/2/15 15:35
 */
public class DeadLockDemo {
    public static void main(String[] args) {
        String apple = "apple";
        String banana = "banana";
        new Thread(new Eat(apple, banana), "T1").start();
        new Thread(new Eat(banana, apple), "T2").start();
    }
}
```

```

    }
}

class Eat implements Runnable {

    String str1;
    String str2;

    public Eat(String str1, String str2) {
        this.str1 = str1;
        this.str2 = str2;
    }

    @Override
    public void run() {
        synchronized (str1) {
            System.out.println(Thread.currentThread().getName() + " " + str1 + "
-> get " + str2);
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (str2) {
                System.out.println(Thread.currentThread().getName() + " " + "
get " + str2);
            }
        }
    }
}

```

## 怎么解决问题

1. 使用jdk的bin目录下 **jps** 定位进程号: `jps -l`
2. 使用 **jstack** 查看进程信息, 找到死锁问题:

```

Java stack information for the threads listed above:
=====
"T2":
    at juc.lock.Eat.run(DeadLockDemo.java:40)
    - waiting to lock <0x000000076e290740> (a java.lang.String)
    - locked <0x000000076e290778> (a java.lang.String)
    at java.lang.Thread.run(Thread.java:748)
"T1":
    at juc.lock.Eat.run(DeadLockDemo.java:40)
    - waiting to lock <0x000000076e290778> (a java.lang.String)
    - locked <0x000000076e290740> (a java.lang.String)
    at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.

```

3.

## 面试、工作中如何排查问题？

1. 查看日志
2. 查看堆栈信息

## 堆栈中经常出现的问题有哪几种？

SOF和OOM

## 怎么排除死锁？

死锁预防 ----- 确保系统永远不会进入死锁状态

## 小结

---