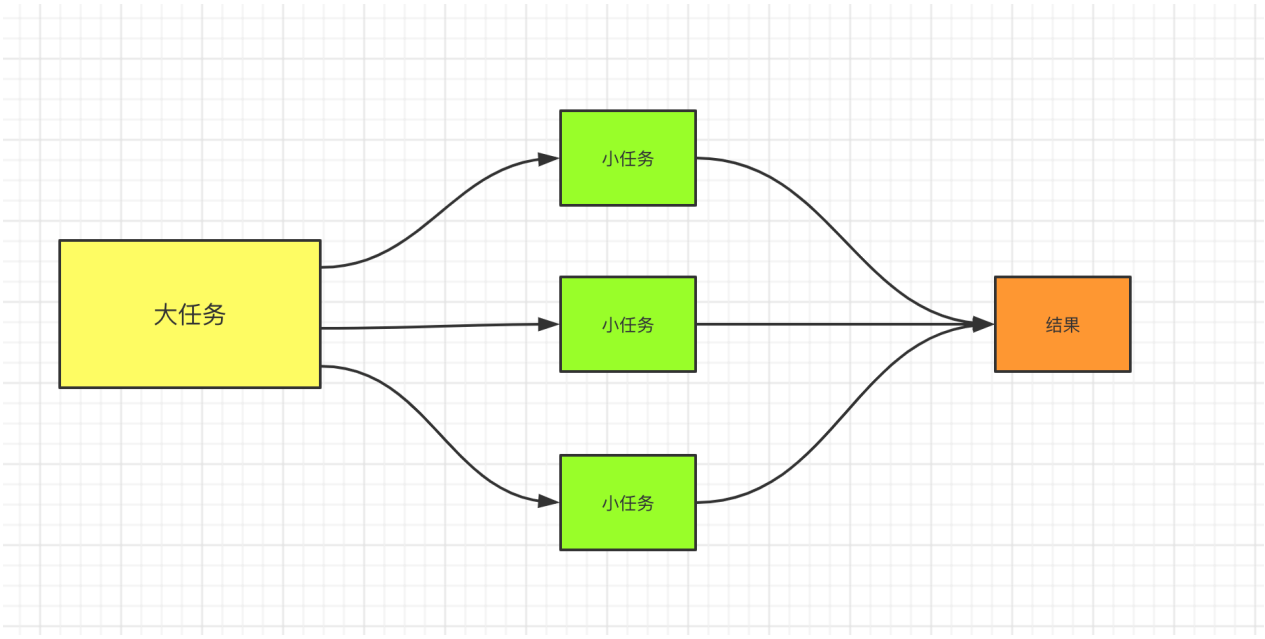
ForkJoin 框架

ForkJoin 是 JDK 1.7 后发布的多线程并发处理框架，功能上和 JUC 类似，JUC 更多时候是使用单个类完成操作，ForkJoin 使用多个类同时完成某项工作，处理上比 JUC 更加丰富，实际开发中使用的场景并不是很多，互联网公司真正有高并发需求的情况才会使用，**面试时候会加分**

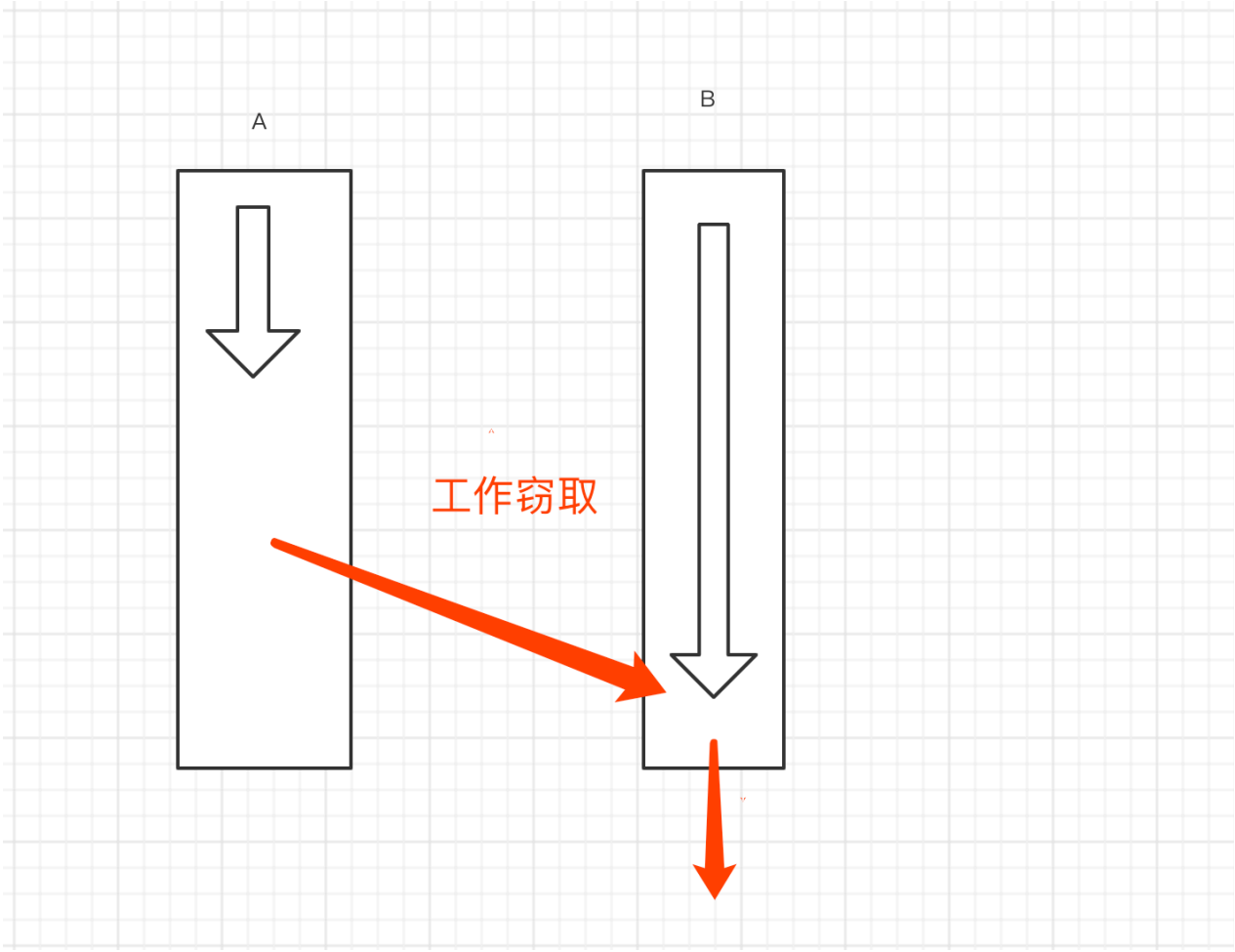
本质上是对线程池的一种的补充，对线程池功能的一种扩展，基于线程池的，它的核心思想就是将一个大型的任务拆分成很多个小任务，分别执行，最终将小任务的结果进行汇总，生成最终的结果。



本质就是把一个线程的任务拆分成多个小任务，然后由多个线程并发执行，最终将结果进行汇总。

比如 A B 两个线程同时还执行，A 的任务比较多，B 的任务相对较少，B 先执行完毕，这时候 B 去帮助 A 完成任务（将 A 的一部分任务拿过来替 A 执行，执行完毕之后再把结果进行汇总），从而提高效率。

工作窃取

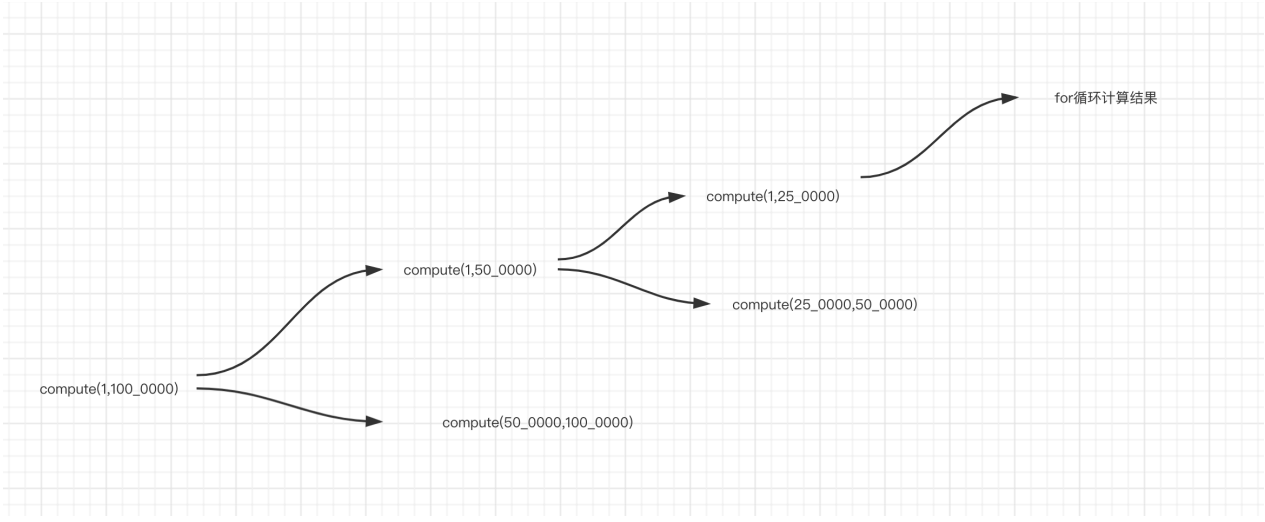


ForkJoin 框架，核心是两个类

- ForkJoinTask （描述任务）
- ForkJoinPool（线程池）提供多线程并发工作窃取

使用 ForkJoinTask 最重要的就是要搞清楚如何拆分任务，这里用的是**递归**思想。

1、需要创建一个 ForkJoinTask 任务，ForkJoinTask 是一个抽象类，不能直接创建 ForkJoinTask 的实例化对象，开发者需要自定义一个类，继承 ForkJoinTask 的子类 RecursiveTask ，Recursive 就是递归的意思，该类就提供了实现递归的功能。



```java
package com.southwind.demo;

import java.util.concurrent.RecursiveTask;

/**
 * 10亿求和
 */
public class ForkJoinDemo extends RecursiveTask<Long> {

    private Long start;
    private Long end;
    private Long temp = 100_0000L;

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

    @Override
    protected Long compute() {
        if((end-start)<temp){
            Long sum = 0L;
            for (Long i = start; i <= end; i++) {
                sum += i;
            }
            return sum;
        }else{
            Long avg = (start+end)/2;
            ForkJoinDemo task1 = new ForkJoinDemo(start,avg);
            task1.fork();
            ForkJoinDemo task2 = new ForkJoinDemo(avg,end);
            task2.fork();
            return task1.join()+task2.join();
        }
    }
}
```

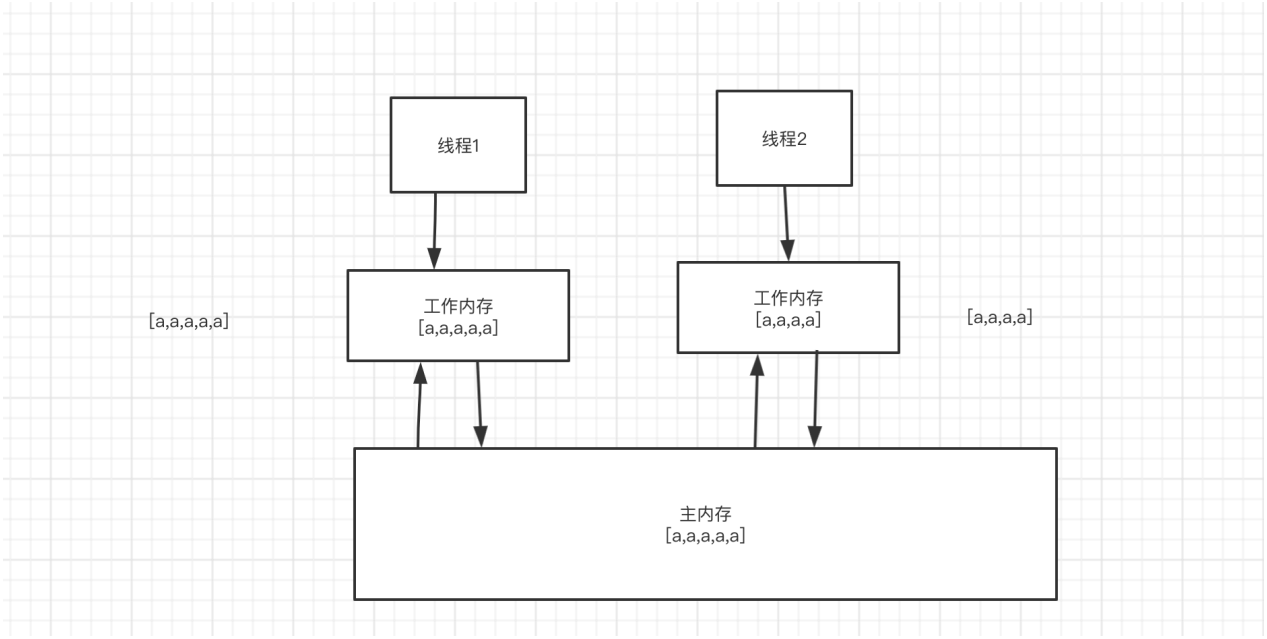
```java
package com.southwind.demo;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

public class Test {
    public static void main(String[] args) {
        Long startTime = System.currentTimeMillis();
```

```
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        ForkJoinTask<Long> task = new ForkJoinDemo(0L,10_0000_0000L);
        forkJoinPool.execute(task);
        Long sum = 0L;
        try {
            sum = task.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        Long endTime = System.currentTimeMillis();
        System.out.println(sum+", 供耗时"+(endTime-startTime));
    }
}
```

Volatile 关键字



Volatile 是 JVM 提供的轻量级同步机制，**可见性**，主内存对象线程可见。

一个线程执行完任务之后还，会把变量存回到主内存中，并且从主内存中读取当前最新的值，如果是一个空的任务，则不会重新读取主内存中的值

```
package com.southwind.demo2;

import java.util.concurrent.TimeUnit;

public class Test {
    private static int num = 0;

    public static void main(String[] args) {
        /**
         * 循环
```

```
         */
        new Thread(()->{
            while(num == 0){
                System.out.println("---Thread---");
            }
        }).start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        num = 1;
        System.out.println(num);
    }
}
```

```
package com.southwind.demo2;

import java.util.concurrent.TimeUnit;

public class Test {
    private static volatile int num = 0;

    public static void main(String[] args) {
        /**
         * 循环
         */
        new Thread(()->{
            while(num == 0){

            }
        }).start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        num = 1;
        System.out.println(num);
    }
}
```

线程池 workQueue

一个阻塞队列，用来存储等待执行的任务，常用的阻塞队列有以下几种：

- ArrayBlockingQueue：基于数组的先进先出队列，创建时必须指定大小。
- LinkedBlockingQueue：基于链表的先进先出队列，创建时可以不指定大小，默认值时Integer.MAX_VALUE。
- SynchronousQueue：它不会保持提交的任务，而是直接新建一个线程来执行新来的任务。
- PriorityBlockingQueue：具有优先级的阻塞队列。