新语言，新思维

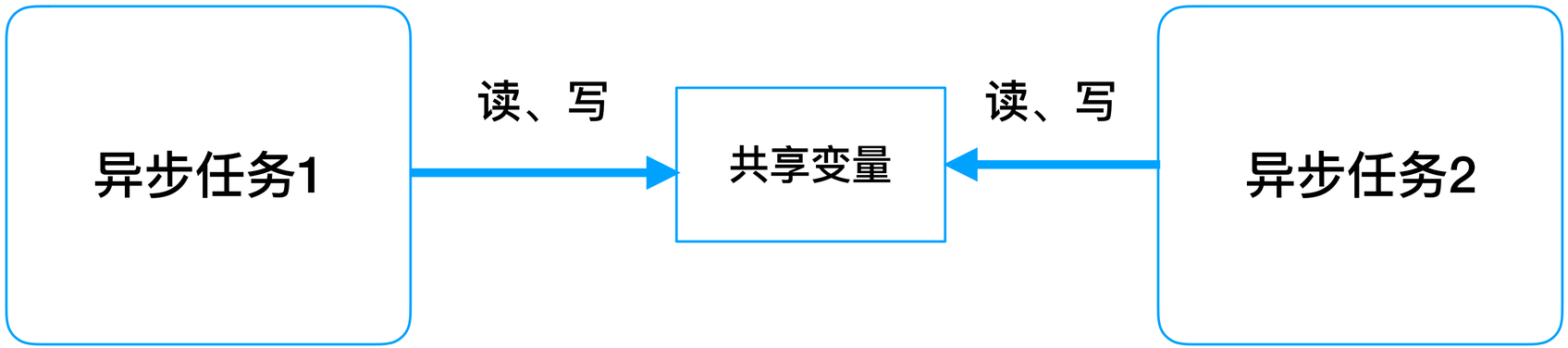
解读一个并发问题的多种实现

陶召胜

next:

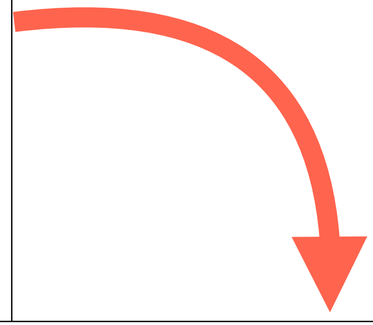
异步编程的问题

变量读写冲突



IO阻塞

```scala
object ExampleBlockingScala extends App {
  //所有并发任务在这个拥有10个线程的线程池中执行
  implicit val ec = ExecutionContext.fromExecutorService(Executors.newFixedThreadPool(10))

  //创建100个并发任务，每个任务都会阻塞5秒
  (1 to 100).map(i => {
    println(s"Calling blocking Future: ${i}")
    Future {
      Thread.sleep(5000) //阻塞5秒，模拟I/O阻塞
      println(s"Blocking future finished ${i}")
    }
  })

  //这是另一个并发任务，因为线程被大量I/O阻塞导致这个任务很少有机会得到执行
  Future {
    (1 to 100).map(i => println(s"another a future,print: ${i}"))
  }

  println(">>> Press ENTER to exit <<<")
  StdIn.readLine()
  ec.shutdown()
}
```



名称	18:23:05	18:23:10	18:23:1 运行		总计
Attach Listener			97,268 ms	(100%)	97,268 ms
Finalizer			0 ms	(0%)	97,268 ms
JMX server connection timeout 25			0 ms	(0%)	97,268 ms
main			97,268 ms	(100%)	97,268 ms
Monitor Ctrl-Break			97,268 ms	(100%)	97,268 ms
pool-1-thread-1			0 ms	(0%)	97,268 ms
pool-1-thread-10			0 ms	(0%)	97,268 ms
pool-1-thread-2			0 ms	(0%)	97,268 ms
pool-1-thread-3			0 ms	(0%)	97,268 ms
pool-1-thread-4			0 ms	(0%)	97,268 ms
pool-1-thread-5			0 ms	(0%)	97,268 ms
pool-1-thread-6			0 ms	(0%)	97,268 ms
pool-1-thread-7			0 ms	(0%)	97,268 ms
pool-1-thread-8			0 ms	(0%)	97,268 ms
pool-1-thread-9			0 ms	(0%)	97,268 ms
Reference Handler			0 ms	(0%)	97,268 ms
RMI Scheduler(0)			0 ms	(0%)	97,268 ms
RMI TCP Accept-0			97,268 ms	(100%)	97,268 ms

回调地狱

```
userService.getFavorites(userId, new Callback<List<String>>() { 1
  public void onSuccess(List<String> list) { 2
    if (list.isEmpty()) { 3
      suggestionService.getSuggestions(new Callback<List<Favorite>>() {
        public void onSuccess(List<Favorite> list) { 4
          UiUtils.submitOnUiThread(() -> { 5
            list.stream()
                .limit(5)
                .forEach(uiList::show); 6
            });
        }

        public void onError(Throwable error) { 7
          UiUtils.errorPopup(error);
        }
      });
    } else {
      list.stream() 8
          .limit(5)
          .forEach(favId -> favoriteService.getDetails(favId, 9
            new Callback<Favorite>() {
              public void onSuccess(Favorite details) {
                UiUtils.submitOnUiThread(() -> uiList.show(details));
              }

              public void onError(Throwable error) {
                UiUtils.errorPopup(error);
              }
            }
          ));
    }
  }

  public void onError(Throwable error) {
    UiUtils.errorPopup(error);
  }
});
```

Future也有不足

- get 很容易导致另一个对象阻塞
- 不支持多值、高级错误处理

next:

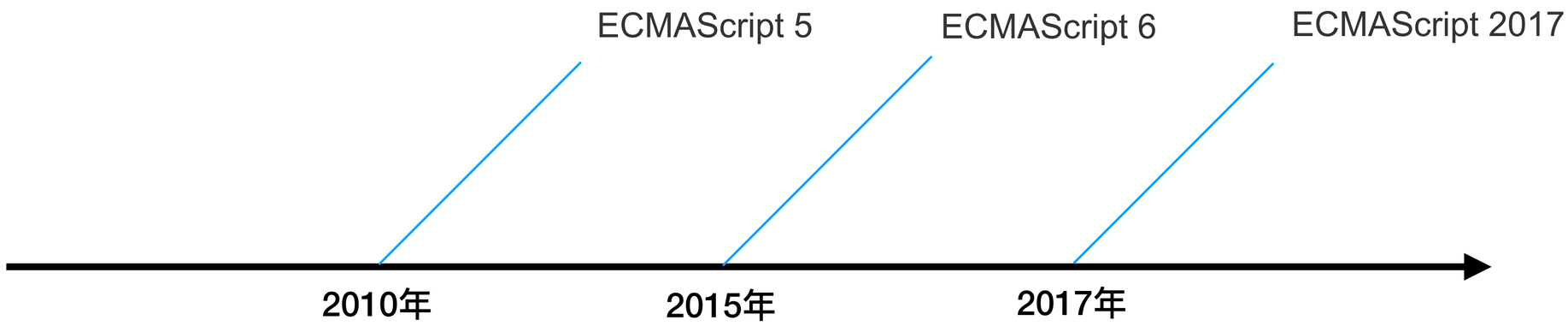
多任务求解1-10,000,000的和

序号	语言	关键点
1	JavaScript	不再有回调地狱，变异步为顺序化思维，程序更加可读
2	Go	高并发调度，通道让异步编程更简单
3	Scala	（1）简洁的异步编程 （2）AKKA：分布式计算框架
4	Java	（1）fork/join （2）CompletableFuture （3）反应式编程（Reactive Programming）

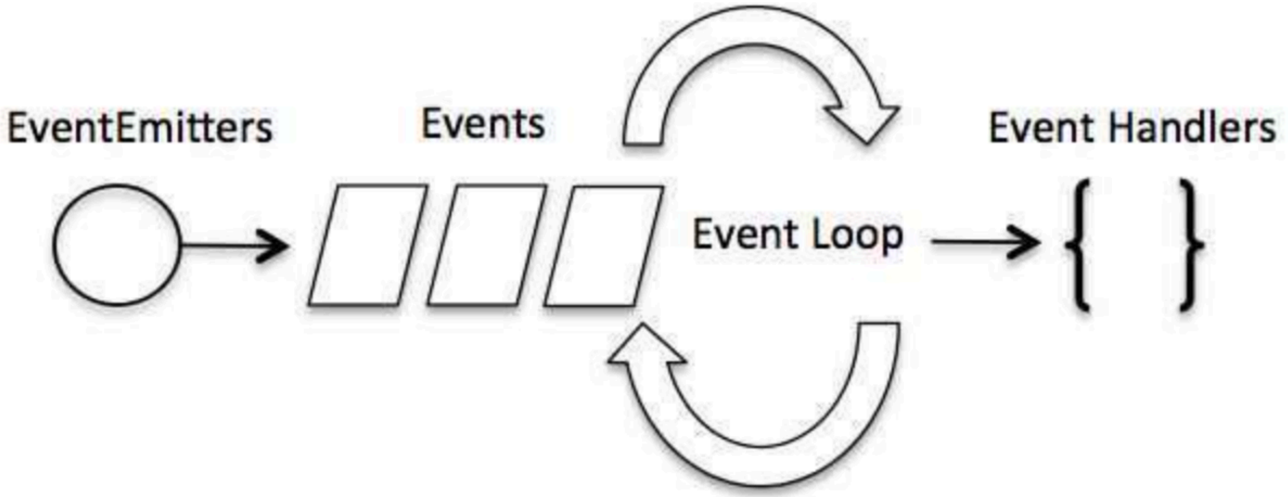
next:

JavaScript

关键点：不再有回调地狱，变异步为顺序化思维，程序更加可读



单线程事件循环



异步，callback

```
1  const fs = require('fs');
2
3  fs.readFile('readme.txt', 'utf-8', (err, data) => {
4      //数据准备好后，这个函数被异步调用
5      if(err) {
6          console.error(err);
7      } else {
8          console.log(data);
9      }
10 });
11 //这个先输出，然后才会输出文件内容
12 console.log('end.');
```

例子JavaScript实现(Promise)

```
12 //函数返回Promise对象
13 const calculate = function(start, end) {
14     return new Promise(function(resolve) {
15         process.nextTick(function() {
16             //模拟异步任务
17             let result = 0;
18             for(let i = start; i <= end; i++) {
19                 result = result + i;
20             };
21             resolve(result); //把异步任务的结果传递给下一个环节
22         });
23     });
24 };
```

```
25 const target = 10000000; //求1-target的和
26
27 //异步执行多个任务，然后(then)收集所有任务的计算结果(results)
28 Promise.all([calculate(1, target / 2), calculate(target / 2 + 1, target)]).then(function(results) {
29     let sum = 0;
30     for(let result of results) {
31         sum = sum + result;
32     }
33     console.log(sum); //最终结果
34 });
```

例子JavaScript实现(Generator)

```
15 //函数返回Promise对象
16 const calculate = function(start, end) {
17     return new Promise(function(resolve) {
18         process.nextTick(function() {
19             //模拟异步任务
20             let result = 0;
21             for(let i = start; i <= end; i++) {
22                 result = result + i;
23             };
24             resolve(result); //把异步任务的结果传递给下一个环节
25         });
26     });
27 };
```

```
28 const target = 10000000; //求1-target的和
29
30 //定义Generator
31 const gen = function*() {
32     const result1 = yield calculate(1, target / 2);
33     const result2 = yield calculate(target / 2 + 1, target);
34     return result1 + result2;
35 }
36
37 const resultPromise = co(gen);//使用co库,需要: const co = require('co')
38 resultPromise.then(results => console.log(results));
```

例子JavaScript实现(async/await)

```
13 //函数返回Promise对象
14 const calculate = function(start, end) {
15     return new Promise(function(resolve) {
16         process.nextTick(function() {
17             //模拟异步任务
18             let result = 0;
19             for(let i = start; i <= end; i++) {
20                 result = result + i;
21             };
22             resolve(result); //把异步任务的结果传递给下一个环节
23         });
24     });
25 };
```

```
26 const target = 10000000; //求1-target的和
27
28 //使用async、await,无需使用co库
29 const aggregateResult = async function() {
30     const result1 = await calculate(1, target / 2);
31     const result2 = await calculate(target / 2 + 1, target);
32     return result1 + result2;
33 };
34
35 aggregateResult().then(results => console.log(results));
```

多进程，利用多核

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
if(cluster.isMaster) {
    console.log(`Master ${process.pid} is running`);
    // 主进程Fork出更多工作进程
    for(let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }
    //有工作进程退出时，收到信号
    cluster.on('exit', (worker, code, signal) => {
        console.log(`worker ${worker.process.pid} died`);
    });
} else {
    // 多个工作进程可以共享TCP连接
    // 在这种情况下，它是一个HTTP服务器
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end('hello world\n');
    }).listen(8000);
    console.log(`Worker ${process.pid} started`);
}
```

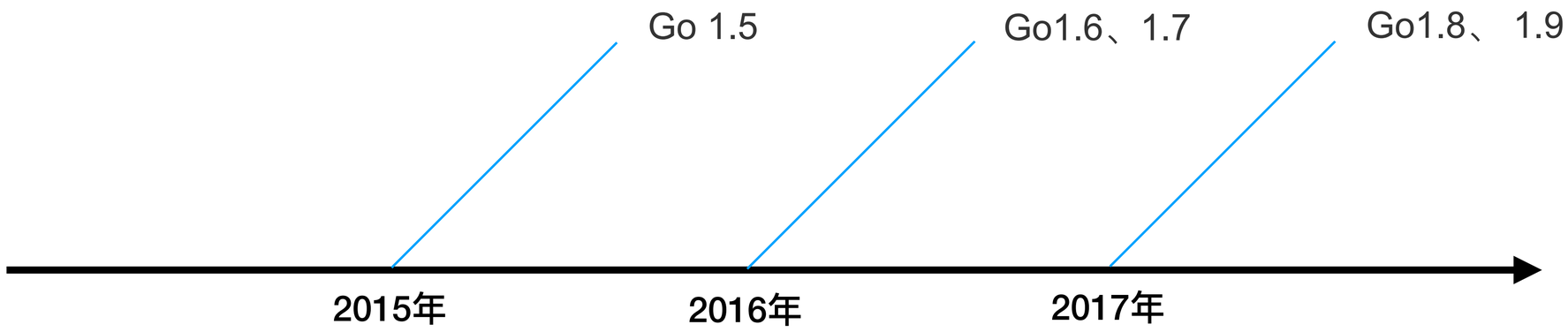
输出：

```
taoxuefeideMacBook-Pro-3:nodejs taoxuefei$ node mycluster.js
Master 1273 is running
Worker 1274 started
Worker 1275 started
Worker 1277 started
Worker 1276 started
```

next:

Go

关键点：高并发调度，通道让异步编程更简单

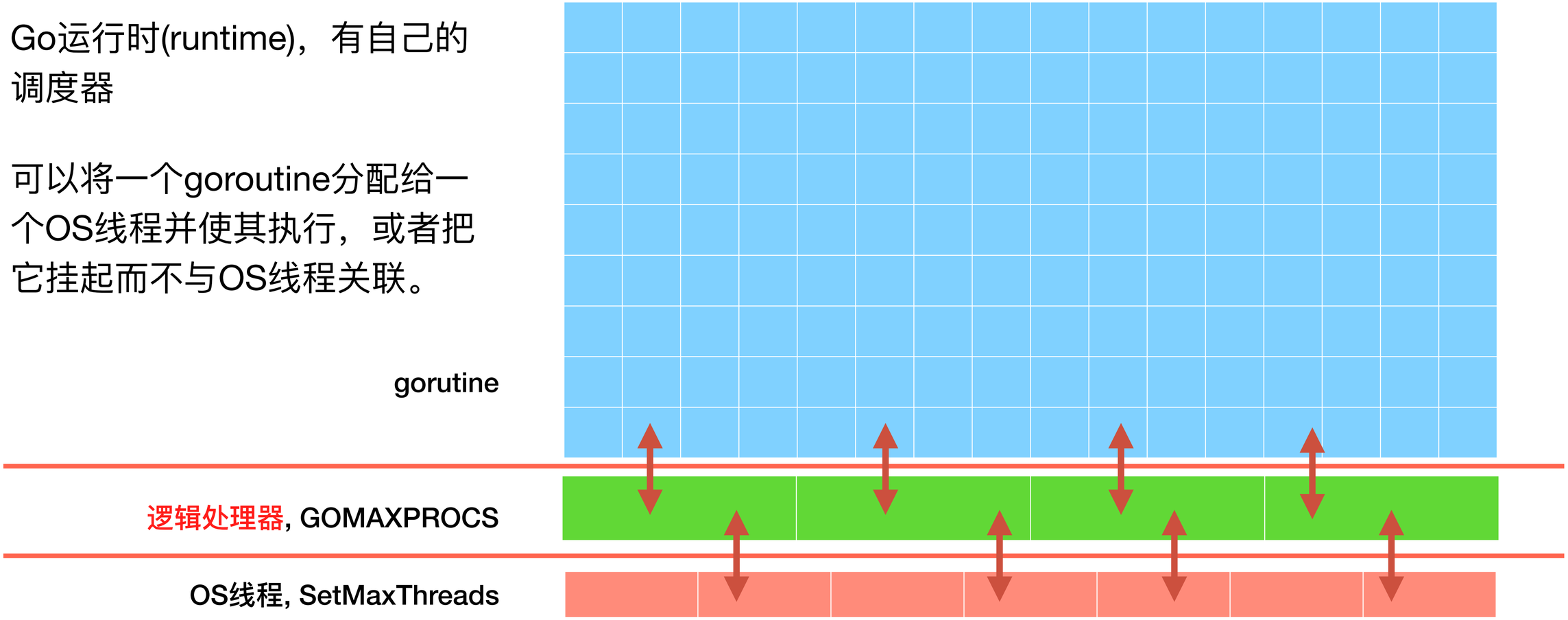


goroutine

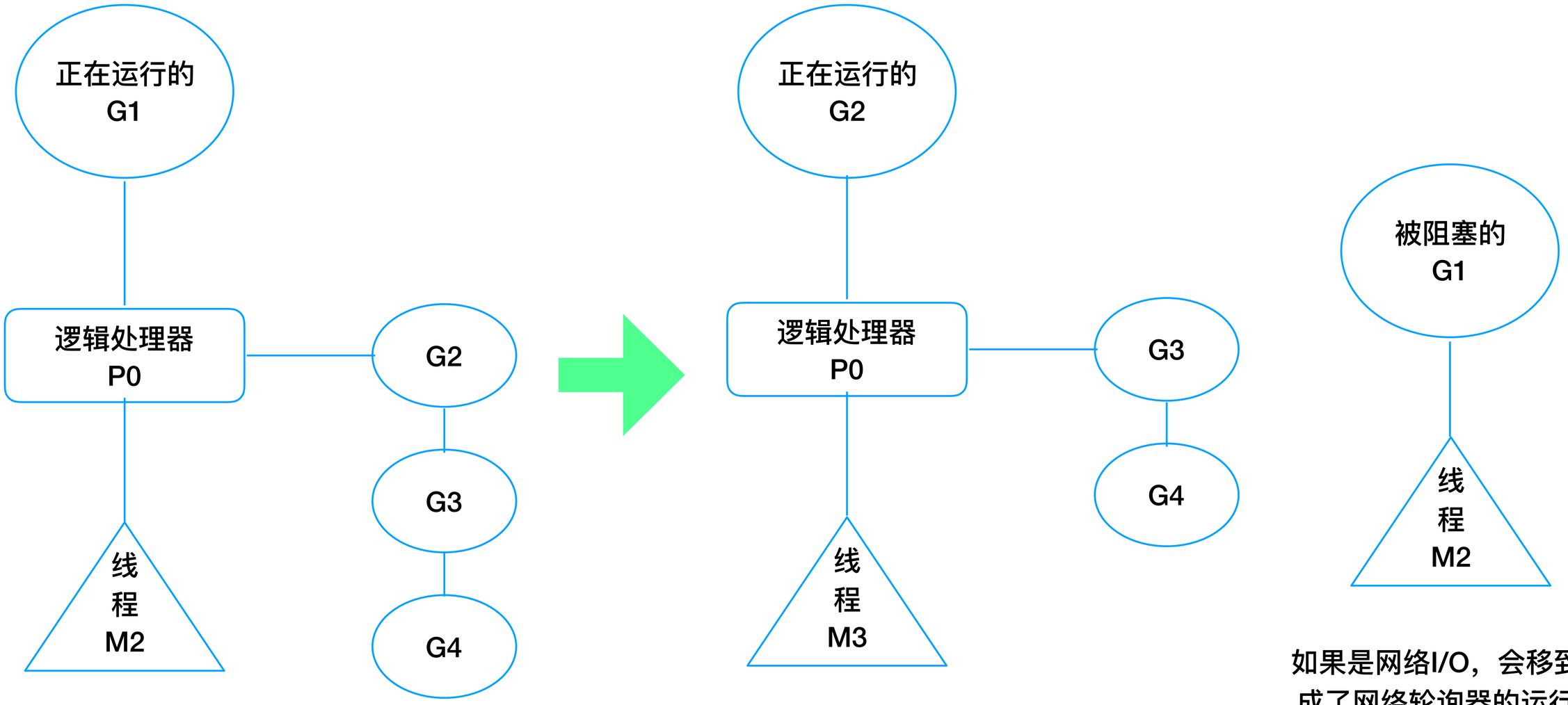
```go
func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }

    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err)
            continue
        }
        go handleConn(conn)
    }
}

func handleConn(c net.Conn) {
    defer c.Close()
    for {
        _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
        if err != nil {
            return
        }
        time.Sleep(1 * time.Second)
    }
}
```

gorutine在逻辑处理器上执行

- Go运行时(runtime)，有自己的调度器
- 可以将一个goroutine分配给一个OS线程并使其执行，或者把它挂起而不与OS线程关联。

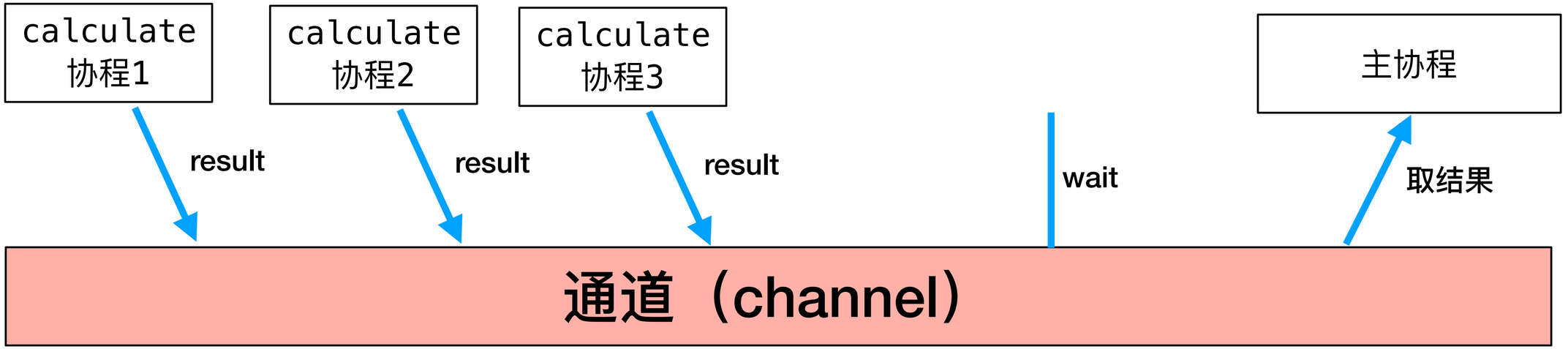


处理阻塞



如果是网络I/O，会移到集成了网络轮询器的运行时

通道是gorutine沟通的桥梁



例子Go实现

```go
const (
    target    = 10000000 //目标，求1-TARGET的和
    threshold = 1000000  //判断是否需要分解子任务的阈值
)

func calculate(waitFinished *sync.WaitGroup, channel chan<- int64, start int64, end int64) {
    defer waitFinished.Done()
    result := int64(0);
    if end-start <= threshold {
        //模拟计算任务，进行计算
        for i := start; i <= end; i++ {
            result += i
        }
        channel <- result //将计算结果放到通道中
    } else {
        //继续分解子任务
        middle := start + ((end - start) / 2)
        waitFinished.Add(2)
        go calculate(waitFinished, channel, start, middle)
        go calculate(waitFinished, channel, middle+1, end)
    }
}

func main() {
    waitFinished := &sync.WaitGroup{}
    channel := make(chan int64, 1000)
    waitFinished.Add(1)
    go calculate(waitFinished, channel, 1, target) //创建协程（主任务）
    waitFinished.Wait() //等待所有任务结束
    close(channel)
    sum := int64(0)
    //从通道中取子任务的计算结果并汇总输出
    for result := range channel {
        sum += result
    }
    fmt.Println(sum)
}
```

next:

Scala

关键点： (1)简洁的异步编程

(2)AKKA：分布式计算框架

创建异步任务就是这么简洁

```scala
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.io.StdIn
import scala.util.Random

object ExampleSimpleFuture extends App {
  //创建一个异步任务
  val future1 = Future {
    //模拟异步任务执行
    Thread.sleep(Random.nextInt(1000))
    "Result for Future1"//任务执行结果
  }

  //创建另一个异步任务
  val future2 = Future {
    Thread.sleep(Random.nextInt(1000))
    "Result for Future2"//任务执行结果
  }

  //处理两个异步任务的结果，注意也是异步的
  future1 foreach println
  future2 foreach println

  println(">>> Press ENTER to exit <<<")
  StdIn.readLine()
}
```

异步任务之间的连续处理也是这么简洁

```scala
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.io.StdIn
import scala.util.Random

object ExampleFutureMap extends App {
  //第一个异步任务
  val future1 = Future {
    //模拟异步任务执行
    Thread.sleep(Random.nextInt(1000))
    "Hello" + "World" //任务执行结果
  }

  //第二个异步任务，处理第一个异步任务的结果
  val future2 = future1 map { x =>
    //模拟异步任务执行
    Thread.sleep(Random.nextInt(1000))
    x.length //任务执行结果
  }

  //处理第二个异步任务的结果（输出），也是异步的
  future2 foreach println

  println(">>> Press ENTER to exit <<<")
  StdIn.readLine()
}
```

例子Scala实现

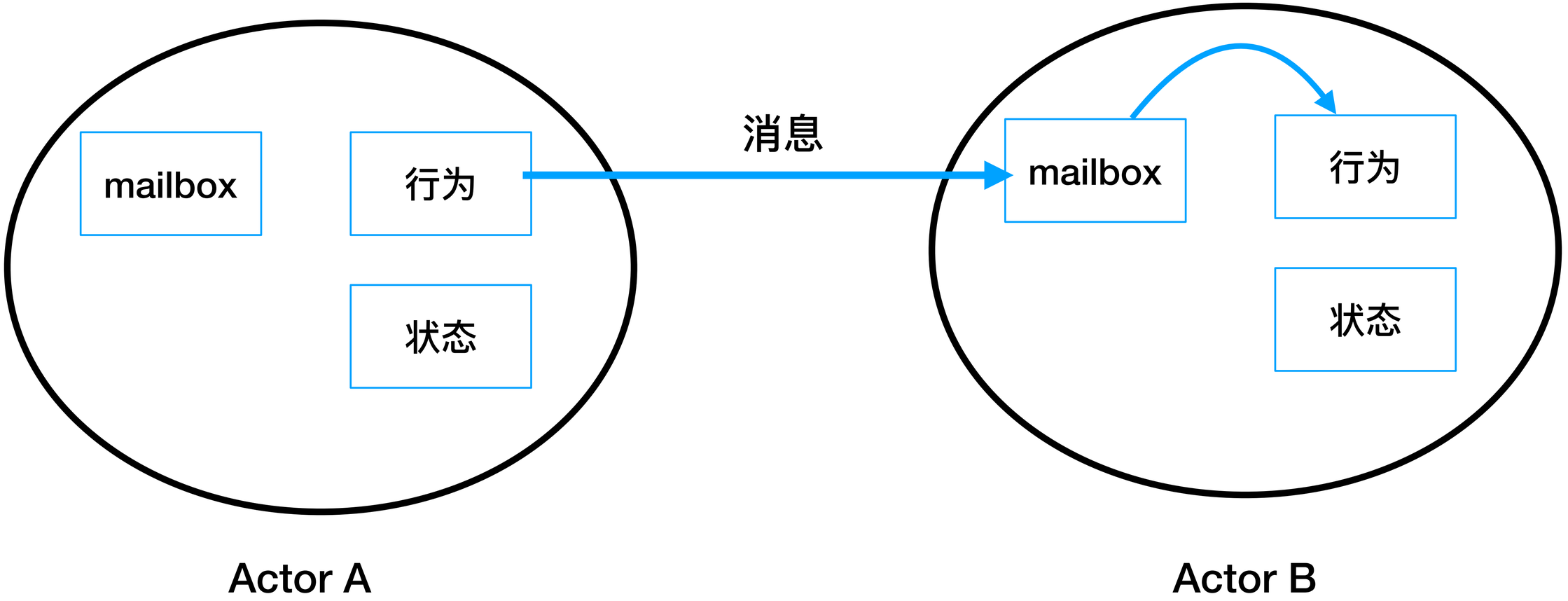
```scala
object ExampleScala extends App {
  //所有并发任务在这个拥有10个线程的线程池中执行
  implicit val ec = ExecutionContext.fromExecutorService(Executors.newWorkStealingPool(10))

  val target = 10000000 //目标，求1-TARGET的和
  val threshold = 1000000 //每个子任务的区间段
  val times = target / threshold //分解成10子个任务

  //使用Future创建10个异步子任务
  val futures = for (i <- 1 to times) yield Future {
    val start = threshold.asInstanceOf[Long] * (i - 1) + 1
    val end = threshold * i
    //模拟子任务执行，此处只是使用了scala的数字序列简单求和函数代替一个耗时计算
    (start to end).sum
  }

  //获取10个异步子任务的结果，并将这些结果做reduce求和操作，注意结果还是Future
  val results = Future.reduceLeft(futures)(_ + _)

  //将Future的结果打印出来，注意这一步也是异步执行
  results foreach println

  println(">>> Press ENTER to exit <<<")
  StdIn.readLine()
```

对比：Java8 CompletableFuture实现

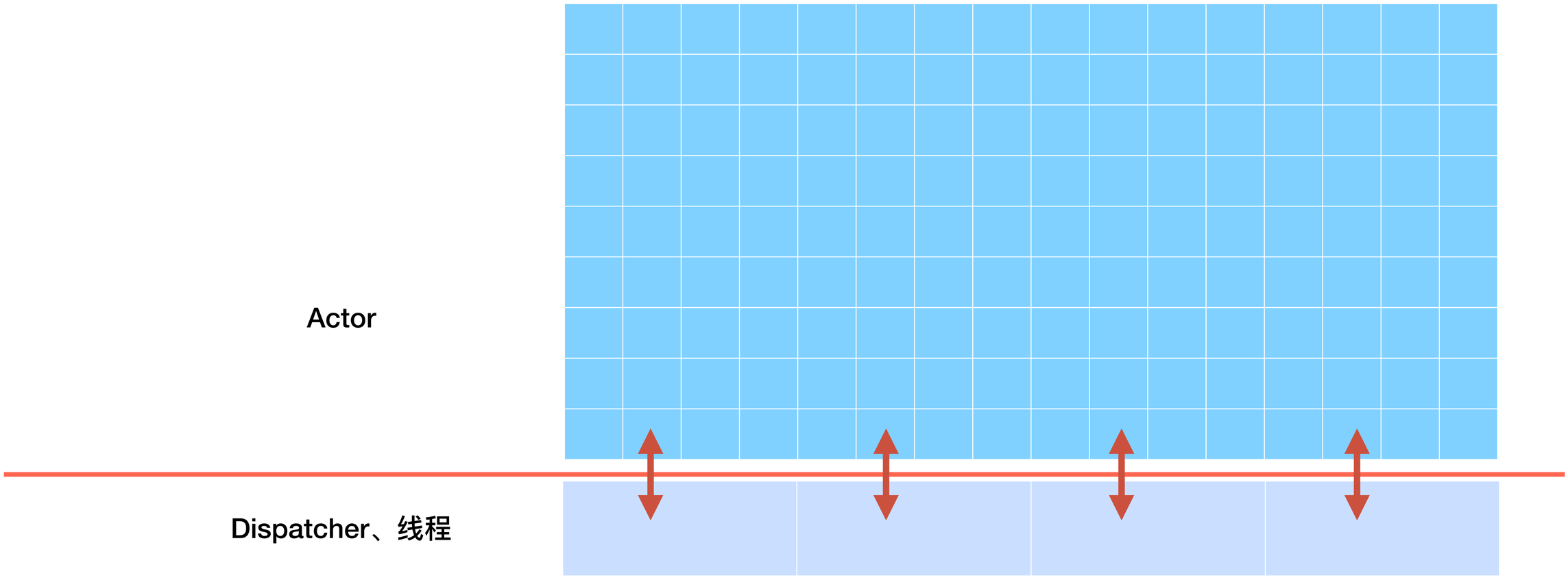
```java
public class ExampleCompletableFuture {
    private static final long THRESHOLD = 1_000_000;

    public static void main(String[] args) {
        //模拟10个子任务
        Stream<Integer> tasks = Stream.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
        //生成10个异步子任务
        List<CompletableFuture<Long>> futureList = tasks.map(i -> {
            return CompletableFuture.supplyAsync(() -> {
                //模拟异步子任务执行
                return LongStream.range(i * THRESHOLD, (i + 1) * THRESHOLD + 1).sum();
            });
        }).collect(Collectors.toList());

        //转成数组
        CompletableFuture<Long>[] combinationArray = futureList.toArray(new CompletableFuture[futureList.size()]);
        //创建等待所有异步子任务完成的CompletableFuture
        CompletableFuture<Void> allDone = CompletableFuture.allOf((combinationArray));
        //所有异步子任务完成后要做的事，获取结果、生成列表
        CompletableFuture<List<Long>> results = allDone.thenApply(v -> {
            return futureList.stream().map(CompletableFuture::join).collect(Collectors.toList());
        });

        //汇总子任务的计算结果，输出最后结果
        System.out.println(results.join().stream().mapToLong(i -> i).sum());
    }
}
```

AKKA分布式计算框架

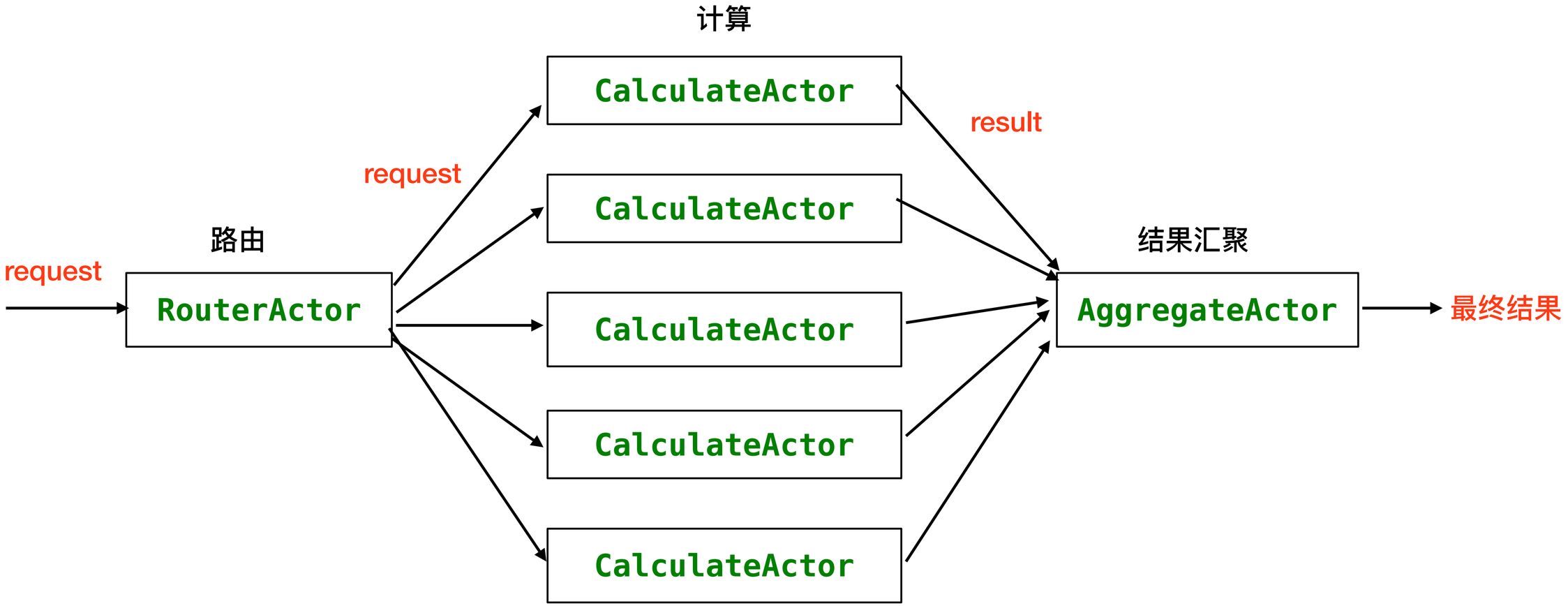
- 核心概念：Actor，是行为和状态的容器



Actor与线程



例子Actor设计



Dispatcher，线程池

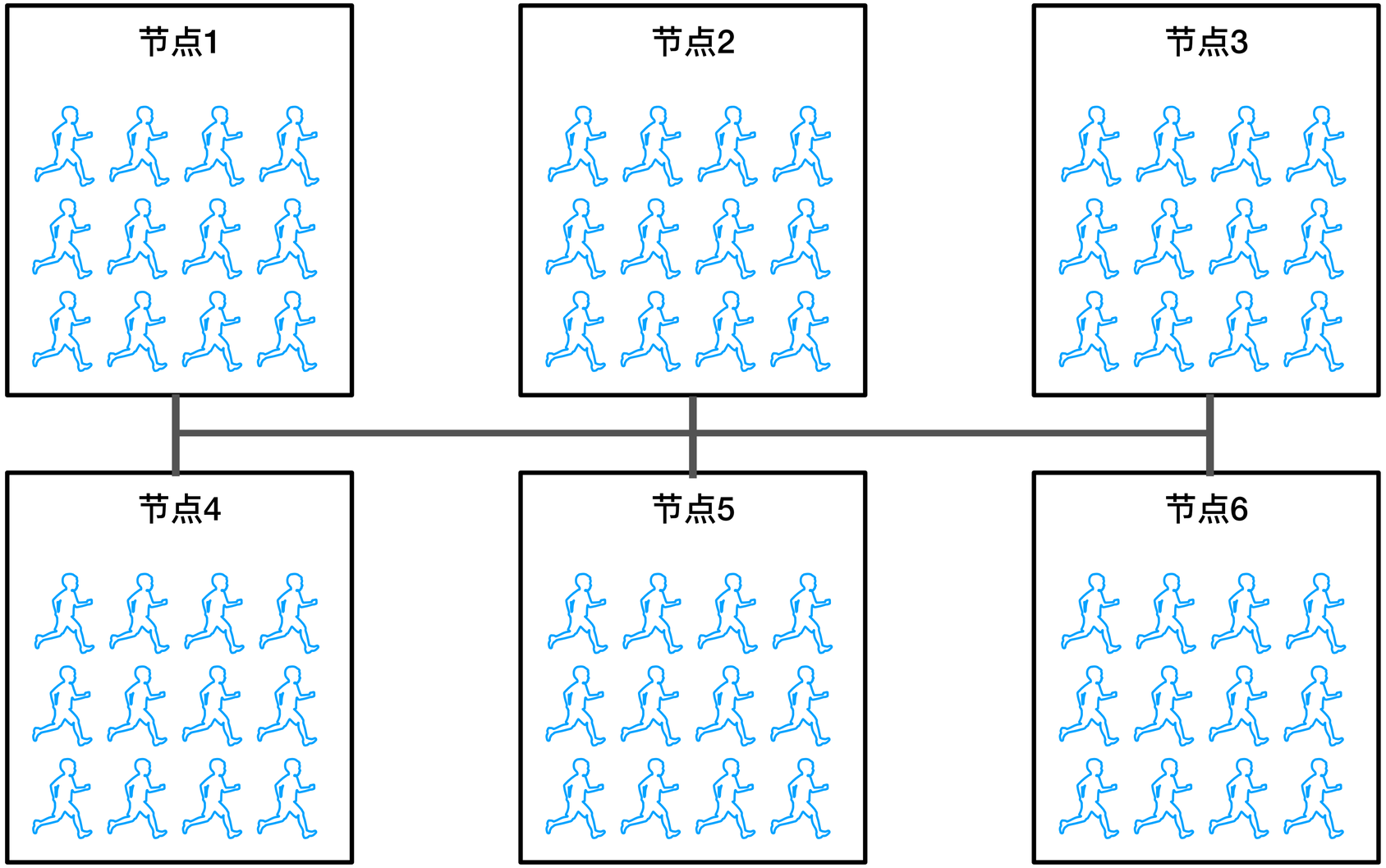
例子AKKA实现

```scala
object CalculateActor {
  def props: Props = Props[CalculateActor]
  final case class Task(start: Long, end: Long)
  final case class Result(result: Long)
}

//这个Actor负责处理计算任务
class CalculateActor extends Actor with ActorLogging {
  import CalculateActor._

  def receive = {
    case Task(start, end) =>
      log.info(s"start:$start ; end: $end")
      sender() ! Result((start to end).sum) //此处简单模拟计算任务
  }
}

object AggregateActor {
  def props(times: Int): Props = Props(new AggregateActor(times))
}

//这个Actor负责汇聚"计算Actor"发来的计算结果
class AggregateActor(times: Int) extends Actor with ActorLogging {
  import CalculateActor._

  val results = ListBuffer[Long]()

  def receive = {
    case Result(result) =>
      results.+=(result)
      if (results.size == times) {
        log.info(s"result:${results.sum}")
      }
  }
}
```

例子AKKA实现(续)

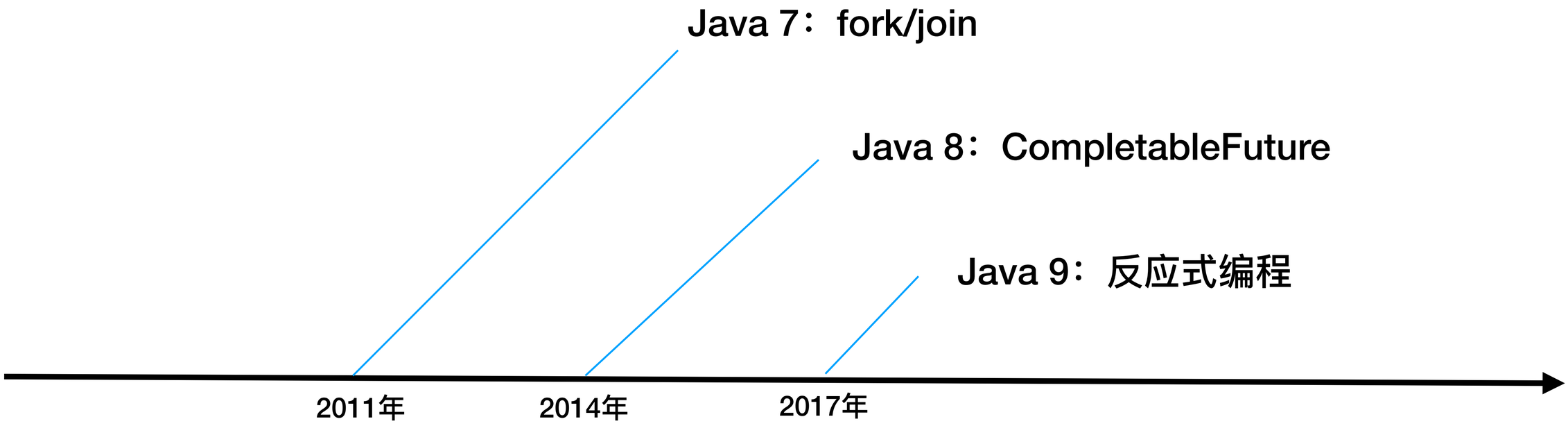
```
46  object ExampleAkka extends App {
47
48    import CalculateActor._
49
50    val system: ActorSystem = ActorSystem("ExampleAkka")
51    try {
52      val target = 10000000 //目标，求1-TARGET的和
53      val threshold = 1000000 //每个子任务的区间段
54      val times = target / threshold //分解成10子个任务
55
56      //这个Actor负责收集"计算Actor"（CalculateActor）的结果，并最终汇总输出
57      val aggregateActor: ActorRef = system
58        .actorOf(AggregateActor.props(times), "aggregateActor")
59
60      //这个路由Actor可以将任务轮询转给它的10个子Actor并发处理（CalculateActor）
61      val routerActor: ActorRef = system
62        .actorOf(RoundRobinPool(10).props(Props[CalculateActor]), "router")
63
64      //向路由Actor发出10个计算任务
65      (1 to times).foreach(i => {
66        routerActor.tell(Task(threshold * (i - 1) + 1, threshold * i), aggregateActor)
67      })
68
69      println(">>> Press ENTER to exit <<<")
70      StdIn.readLine()
71    } finally {
72      system.terminate()
73    }
74  }
```

AKKA的分布式

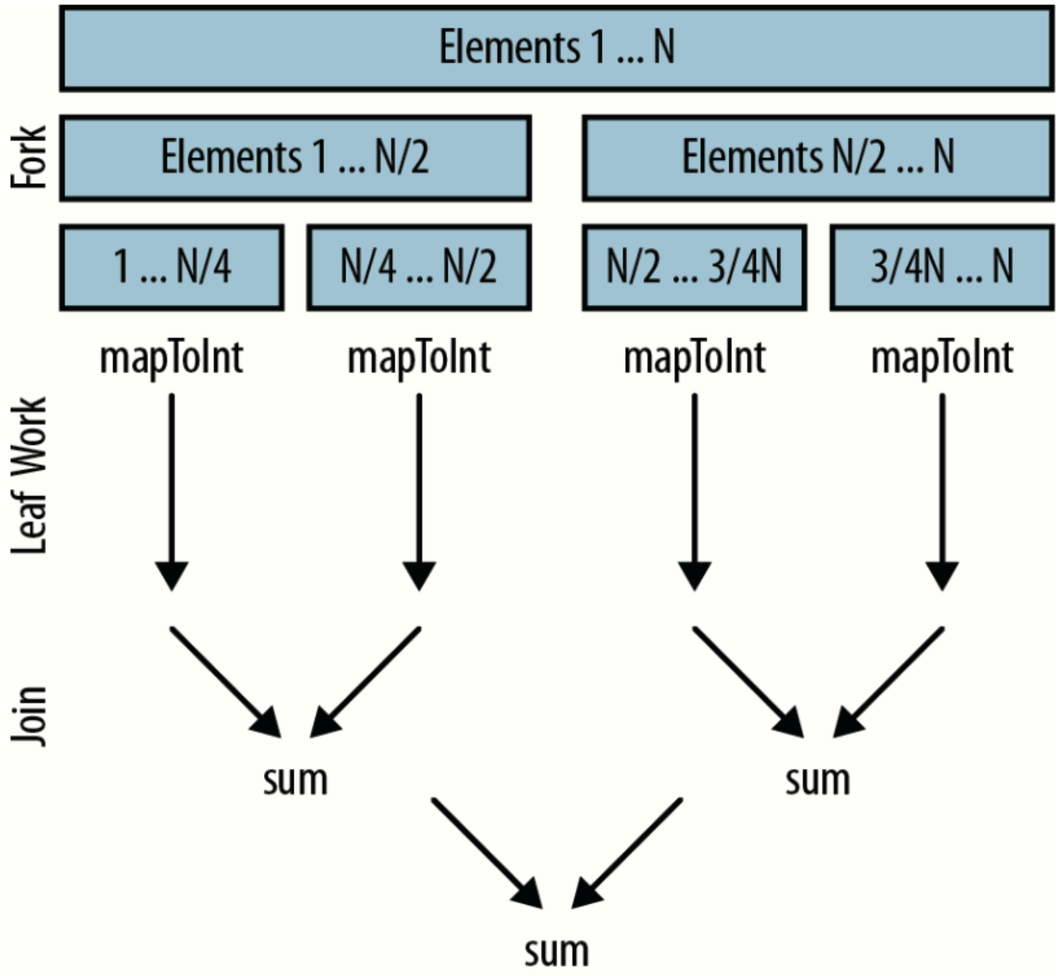


next:

Java


Java 7：fork/join
Java 8：CompletableFuture
Java 9：反应式编程
2011年
2014年
2017年

Java7 fork/join 分解合并问题



例子fork/join实现

```java
class CalculateTask extends RecursiveTask<Long> {
    private static final long THRESHOLD = 1_000_000;//判断是否需要分解子任务的阈值
    private long start,end;

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

    @Override
    protected Long compute() {
        long sum = 0;
        if (end - start <= THRESHOLD) {
            for (long i = start; i <= end; i++) {
                sum += i;
            }
        } else {
            //fork更小的子任务
            long middle = (start + end) / 2;
            CalculateTask leftTask = new CalculateTask(start, middle);
            CalculateTask rightTask = new CalculateTask(middle + 1, end);
            leftTask.fork();
            rightTask.fork();

            //汇聚子任务的计算结果
            long leftResult = leftTask.join();
            long rightResult = rightTask.join();
            sum = leftResult + rightResult;
        }
        return sum;
    }
}
```

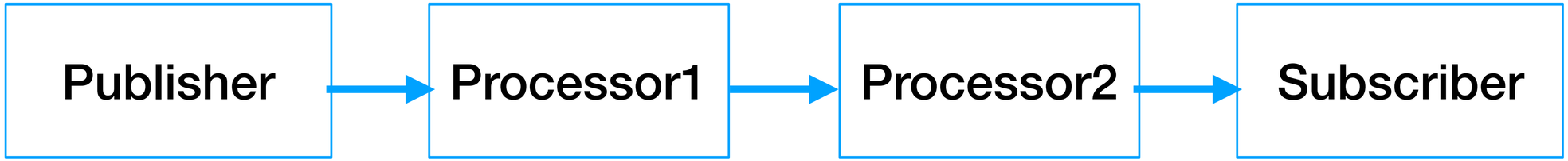
```java
public class ExampleForkJoin {
    private static final long TARGET = 10_000_000;//求1-TARGET的和

    public static void main(String[] args) {
        //创建forkjoin线程池
        ForkJoinPool forkJoinPool = (ForkJoinPool) Executors.newWorkStealingPool(10);
        CalculateTask task = new CalculateTask(1, TARGET);//创建主计算任务
        Future<Long> result = forkJoinPool.submit(task);//任务提交到线程池
        try {
            System.out.println(result.get());//获取结果，同步，测试用
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

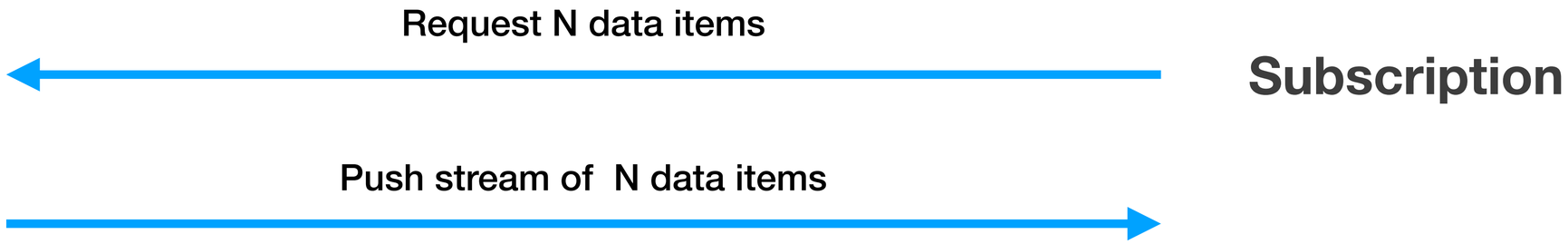
例子Java8 CompletableFuture实现

```java
public class ExampleCompletableFuture {
    private static final long THRESHOLD = 1_000_000;

    public static void main(String[] args) {
        //模拟10个子任务
        Stream<Integer> tasks = Stream.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
        //生成10个异步子任务
        List<CompletableFuture<Long>> futureList = tasks.map(i -> {
            return CompletableFuture.supplyAsync(() -> {
                //模拟异步子任务执行
                return LongStream.range(i * THRESHOLD, (i + 1) * THRESHOLD + 1).sum();
            });
        }).collect(Collectors.toList());

        //转成数组
        CompletableFuture<Long>[] combinationArray = futureList.toArray(new CompletableFuture[futureList.size()]);
        //创建等待所有异步子任务完成的CompletableFuture
        CompletableFuture<Void> allDone = CompletableFuture.allOf((combinationArray));
        //所有异步子任务完成后要做的事，获取结果、生成列表
        CompletableFuture<List<Long>> results = allDone.thenApply(v -> {
            return futureList.stream().map(CompletableFuture::join).collect(Collectors.toList());
        });

        //汇总子任务的计算结果，输出最后结果
        System.out.println(results.join().stream().mapToLong(i -> i).sum());
    }
}
```

Java9 反应式编程

- Reactive Programming。
- 异步、非阻塞。
- 4个基本概念





接口定义

```
3   public final class Flow {
4
5       private Flow() {}
6
7       @FunctionalInterface
8       public static interface Publisher<T> {
9           public void subscribe(Subscriber<? super T> subscriber);
10      }
11
12      public static interface Subscriber<T> {
13          public void onSubscribe(Subscription subscription);
14          public void onNext(T item);
15          public void onError(Throwable throwable);
16          public void onComplete();
17      }
18
19      public static interface Subscription {
20          public void request(long n);
21          public void cancel();
22      }
23
24      public static interface Processor<T,R> extends Subscriber<T>, Publisher<R> {
25      }
26  }
```

Reactor

- 两个基本概念：Flux 和 Mono。

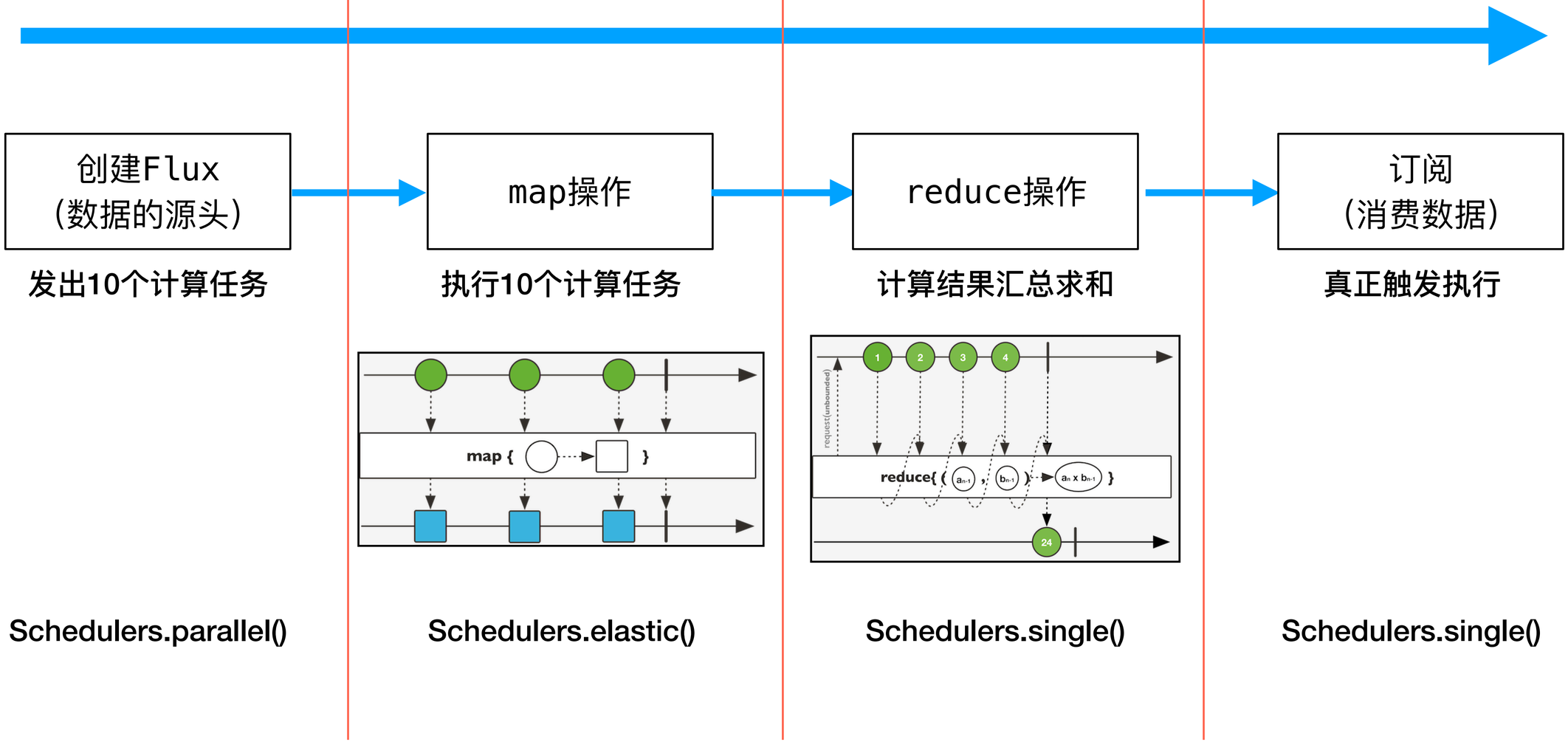
  Flux：包含 0 到 N 个元素的异步序列。

  Mono：包含 0 或者 1 个元素的异步序列。

  消息：正常的包含元素的消息、序列结束的消息和序列出错的消息。

  操作符（Operator）：对流上元素的操作。

例子代码执行过程


创建Flux
(数据的源头)
map操作
reduce操作
订阅
(消费数据)
发出10个计算任务
执行10个计算任务
计算结果汇总求和
真正触发执行
map { }
request(unbounded)
reduce{ (,) }
24
Schedulers.parallel()
Schedulers.elastic()
Schedulers.single()
Schedulers.single()

例子Reactor实现

```
3   import reactor.core.publisher.Flux;
4   import reactor.core.scheduler.Schedulers;
5
6   import java.io.IOException;
7   import java.util.stream.LongStream;
8
9   public class ExampleReactor {
10      private static final long TARGET = 10_000_000;//求1-TARGET的和
11      private static final long THRESHOLD = 1_000_000;
12
13      public static void main(String[] args) throws IOException {
14          Flux.<Integer>create(sink -> {
15              //创建Flux，产生数据序列，模拟发出10个任务
16              for (int i = 0; i < 10; i++) {
17                  sink.next(i);//发出数据
18              }
19              sink.complete();//发出结束信号
20          })
21                  .subscribeOn(Schedulers.parallel())//指定发出数据使用的调度器
22                  .publishOn(Schedulers.elastic())//切换随后operator使用的调度器
23                  .map(task -> {
24                      //模拟子任务执行
25                      return LongStream.range(task * THRESHOLD, (task + 1) * THRESHOLD + 1).sum();
26                  })
27                  .publishOn(Schedulers.single())//切换随后operator使用的调度器
28                  .reduce((x, y) -> x + y)//对数据流做reduce操作
29                  .subscribe(System.out::println);//订阅数据，真正执行，订阅之前什么也不会发生
30          System.out.println(">>> Press ENTER to exit <<<");
31          System.in.read();
32      }
33  }
```

总结

序号	语言	关键点
1	JavaScript	不再有回调地狱，变异步为顺序化思维，程序更加可读
2	Go	高并发调度，通道让异步编程更简单
3	Scala	（1）简洁的异步编程 （2）AKKA：分布式计算框架
4	Java	（1）fork/join （2）CompletableFuture （3）反应式编程（Reactive Programming）

谢谢！

陶召胜