拉勾教育

— 互联网人实战大学 —

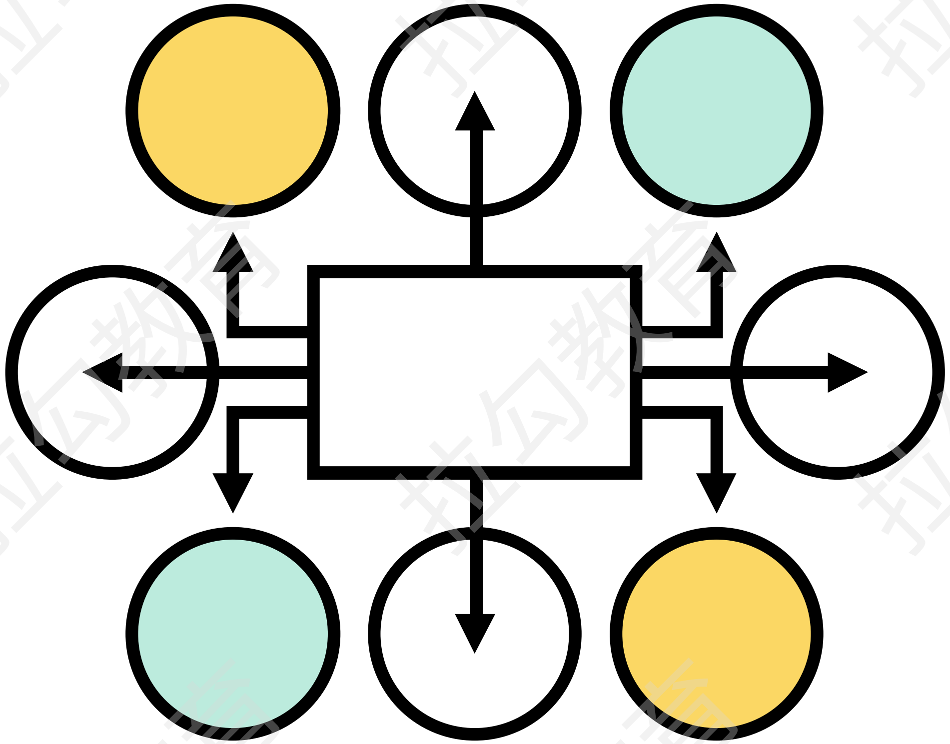
《 Java性能优化与面试21讲》

李国

— 拉勾教育出品 —

12 | 案例分析：并行计算让代码“飞”起来

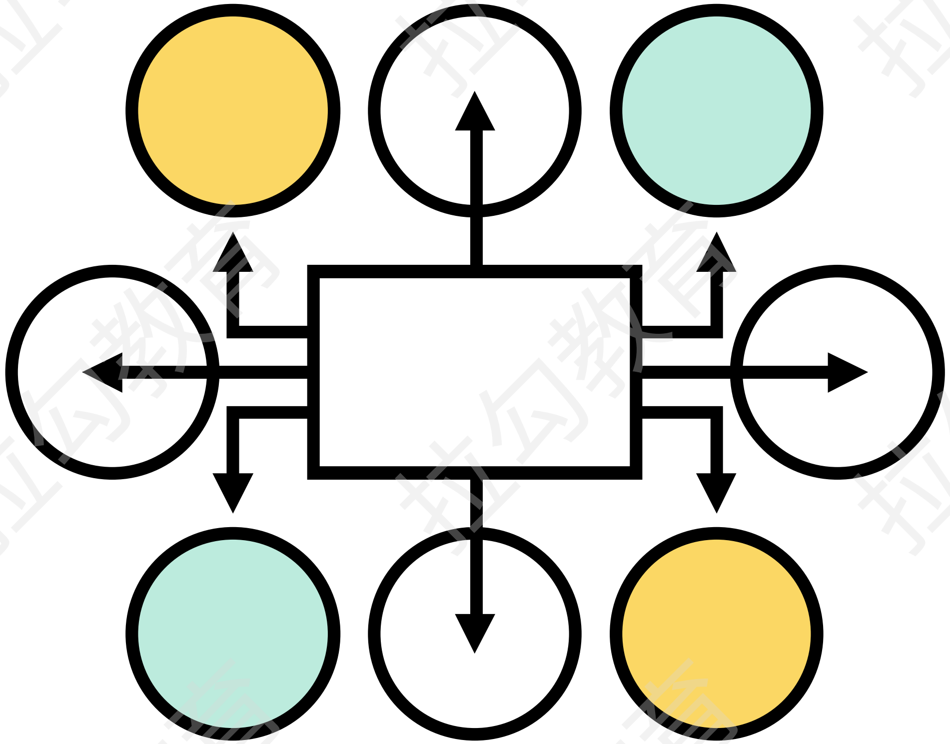
前言



通过**多进程**和**多线程**的手段

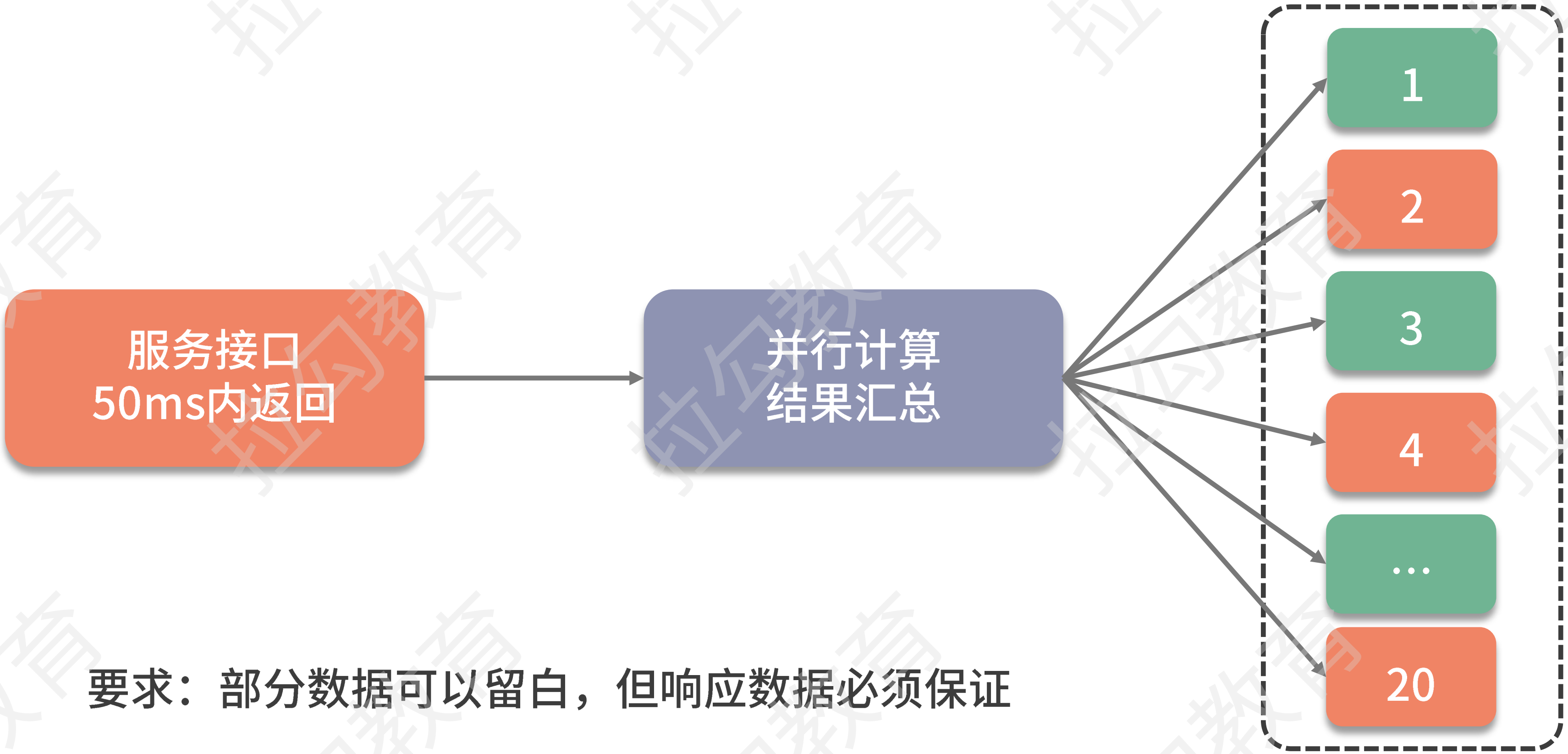
可以让多个 CPU 核同时工作

加快任务的执行



多线程是Java程序员面试和工作中必备的技能

并行获取数据



要求：部分数据可以留白，但响应数据必须保证

并行获取数据

```java
public class ParallelFetcher {
  final long timeout;
  final CountDownLatch latch;
  final ThreadPoolExecutor executor = new ThreadPoolExecutor(100, 200, 1,
      TimeUnit.HOURS, new ArrayBlockingQueue<>(100));
  public ParallelFetcher(int jobSize, long timeoutMill) {
    latch = new CountDownLatch(jobSize);
    timeout = timeoutMill;
  }
  public void submitJob(Runnable runnable) {
    executor.execute(() -> {
      runnable.run();
      latch.countDown();
    });
  }
  public void await() {
    try {
      this.latch.await(timeout, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
      throw new IllegalStateException();
```

并行获取数据

拉勾教育

— 互联网人实战大学 —

```
    latch = new CountDownLatch(jobSize);
    timeout = timeoutMill;
  }
  public void submitJob(Runnable runnable) {
    executor.execute(() -> {
      runnable.run();
      latch.countDown();
    });
  }
  public void await() {
    try {
      this.latch.await(timeout, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
      throw new IllegalStateException();
    }
  }
  public void dispose() {
    this.executor.shutdown();
  }
}
```

并行获取数据


```java
public static void main(String[] args) {
  final String userid = "123";
  final SlowInterfaceMock mock = new SlowInterfaceMock();
  ParallelFetcher fetcher = new ParallelFetcher(20, 50);
  final Map<String, String> result = new HashMap<>();

  fetcher.submitJob(() -> result.put("method0", mock.method0(userid)));
  fetcher.submitJob(() -> result.put("method1", mock.method1(userid)));
  fetcher.submitJob(() -> result.put("method2", mock.method2(userid)));
  fetcher.submitJob(() -> result.put("method3", mock.method3(userid)));
  fetcher.submitJob(() -> result.put("method4", mock.method4(userid)));
  fetcher.submitJob(() -> result.put("method5", mock.method5(userid)));
  fetcher.submitJob(() -> result.put("method6", mock.method6(userid)));
  fetcher.submitJob(() -> result.put("method7", mock.method7(userid)));
  fetcher.submitJob(() -> result.put("method8", mock.method8(userid)));
  fetcher.submitJob(() -> result.put("method9", mock.method9(userid)));
  fetcher.submitJob(() -> result.put("method10", mock.method10(userid)));
  fetcher.submitJob(() -> result.put("method11", mock.method11(userid)));
  fetcher.submitJob(() -> result.put("method12", mock.method12(userid)));
  fetcher.submitJob(() -> result.put("method13", mock.method13(userid)));
  fetcher.submitJob(() -> result.put("method14", mock.method14(userid)));
  fetcher.submitJob(() -> result.put("method15", mock.method15(userid)));
```

并行获取数据

拉勾教育

— 互联网人实战大学 —

```
fetcher.submitJob(() -> result.put("method7", mock.method7(userid)));
fetcher.submitJob(() -> result.put("method8", mock.method8(userid)));
fetcher.submitJob(() -> result.put("method9", mock.method9(userid)));
fetcher.submitJob(() -> result.put("method10", mock.method10(userid)));
fetcher.submitJob(() -> result.put("method11", mock.method11(userid)));
fetcher.submitJob(() -> result.put("method12", mock.method12(userid)));
fetcher.submitJob(() -> result.put("method13", mock.method13(userid)));
fetcher.submitJob(() -> result.put("method14", mock.method14(userid)));
fetcher.submitJob(() -> result.put("method15", mock.method15(userid)));
fetcher.submitJob(() -> result.put("method16", mock.method16(userid)));
fetcher.submitJob(() -> result.put("method17", mock.method17(userid)));
fetcher.submitJob(() -> result.put("method18", mock.method18(userid)));
fetcher.submitJob(() -> result.put("method19", mock.method19(userid)));

fetcher.await();

System.out.println(fetcher.latch);
System.out.println(result.size());
System.out.println(result);

fetcher.dispose();
}
```

并行获取数据

拉勾教育

— 互联网人实战大学 —

I/O 密集型任务

对于常见的互联网服务来说，大多数是属于 I/O 密集型的

比如等待数据库的 I/O，等待网络 I/O 等

计算密集型任务

计算密集型的任务却正好相反，比如一些耗时的算法逻辑

CPU 要想达到最高的利用率，提高吞吐量，最好的方式就是

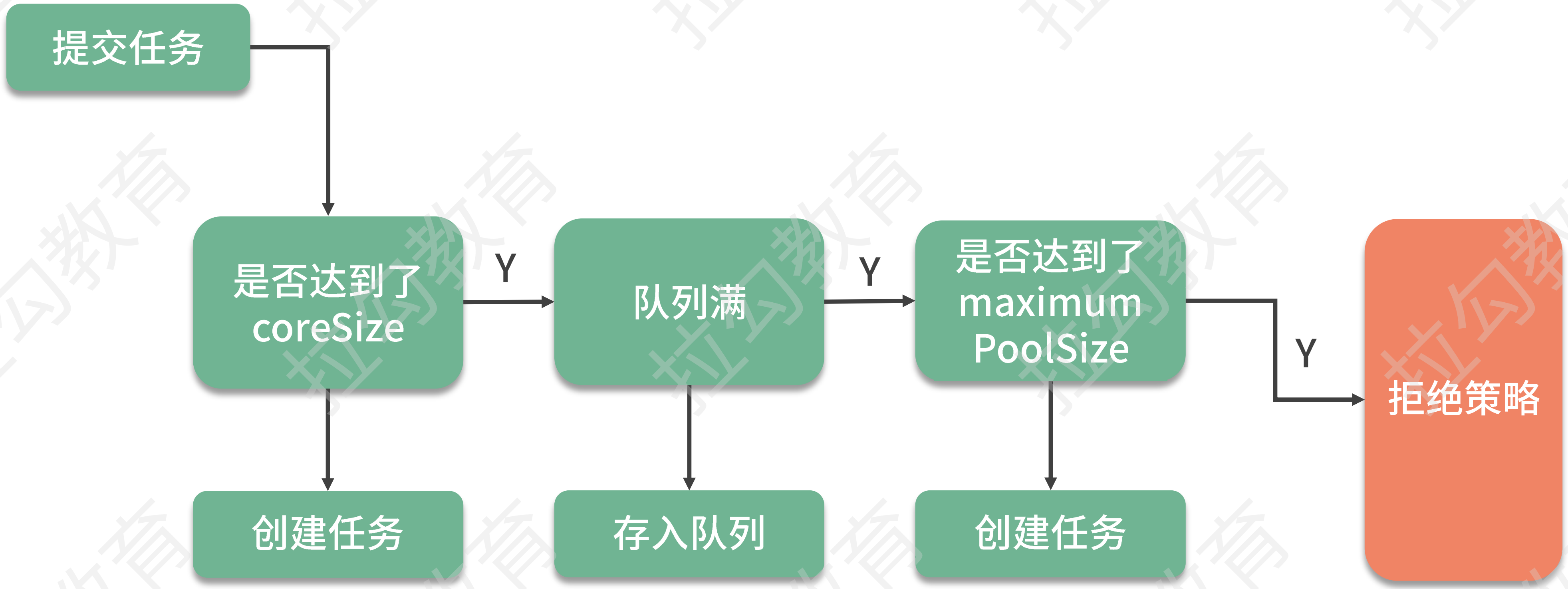
让它尽量少地在任务之间切换，此时线程数等于 CPU 数量，是效率最高的

从池化对象原理看线程池

拉勾教育
— 互联网人实战大学 —

```
public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

从池化对象原理看线程池



从池化对象原理看线程池

固定大小线程池

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

从池化对象原理看线程池

拉勾教育

— 互联网人实战大学 —

无限大小线程池

```
public static ExecutorService newCachedThreadPool() {
  return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
    60L, TimeUnit.SECONDS,
    new SynchronousQueue<Runnable>());
}
```

从池化对象原理看线程池

拉勾教育
— 互联网人实战大学 —

无限大小线程池

```java
public static ExecutorService newCachedThreadPool() {
  return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
    60L, TimeUnit.SECONDS,
    new SynchronousQueue<Runnable>());
}
```

- 如果任务可以接受一定时间的延迟，那么使用 LinkedBlockingQueue 指定一个队列的上限，缓存一部分任务是合理的
- 如果任务对实时性要求很高，比如 RPC 服务，就可以使用 SynchronousQueue 队列对任务进行传递，而不是缓存它们

从池化对象原理看线程池

拒绝策略

默认的拒绝策略是抛出异常的 AbortPolicy，与之类似的是 DiscardPolicy，**非常不推荐**

CallerRunsPolicy，当线程池饱和时，它会使用用户的线程执行任务

DiscardOldestPolicy，它在遇到线程饱和时，会先弹出队列里最旧的任务，然后把当前的任务添加到队列中

在 SpringBoot 中如何使用异步?

```java
@SpringBootApplication
@EnableAsync
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}

@Component
@Async
public class AsyncJob {
    public String testJob() {
        try {
            Thread.sleep( millis: 1000 * 3);
            System.out.println(Thread.currentThread().getName()
        } catch (InterruptedException e) {
            throw new IllegalStateException();
        }
        return "aaa";
    }

    public Future<String> testJob2() {
        String result = this.testJob();
        return new AsyncResult<>(result);
    }
}
```

在 SpringBoot 中如何使用异步?


```
@Bean
public ThreadPoolTaskExecutor getThreadPoolTaskExecutor() {
  ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
  taskExecutor.setCorePoolSize(100);
  taskExecutor.setMaxPoolSize(200);
  taskExecutor.setQueueCapacity(100);
  taskExecutor.setKeepAliveSeconds(60);
  taskExecutor.setThreadNamePrefix("test-");
  taskExecutor.initialize();
  return taskExecutor;
}
```

多线程资源盘点

注意

下面的每一个对比，**都是面试中的知识点**

想要更加深入地理解，你需要阅读 JDK 的源码

多线程资源盘点

线程安全的类

- StringBuilder 对应着 StringBuffer

  后者主要是通过 synchronized 关键字实现了线程的同步

  值得注意的是，在单个方法区域里两者是没有区别的，JIT 的编译优化会去掉 synchronized 关键字的影响

- HashMap 对应着 ConcurrentHashMap

  ConcurrentHashMap 的话题很大，这里提醒一下 JDK1.7 和 1.8 之间的实现已经不一样了

  1.8 已经去掉了分段锁的概念（锁分离技术），并且使用 synchronized 来代替了 ReentrantLock

多线程资源盘点

线程安全的类

- ArrayList 对应着 CopyOnWriteList

  后者是写时复制的概念，适合读多写少的场景

- LinkedList 对应着 ArrayBlockingQueue

  ArrayBlockingQueue 对默认是不公平锁，可以修改构造参数，将其改成公平阻塞队列

  它在 concurrent 包里使用的非常频繁

- HashSet 对应着 CopyOnWriteArraySet

多线程资源盘点

拉勾教育

— 互联网人实战大学 —

线程安全的类

```java
public class FaultDateFormat {
    SimpleDateFormat format = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss");

    public static void main(String[] args) {
        final FaultDateFormat faultDateFormat = new FaultDateFormat();
        ExecutorService executor = Executors.newCachedThreadPool();
        for(int i=0;i<1000;i++){
            executor.submit(()-> {
                try {
                    System.out.println(faultDateFormat.format.parse( source: "2020-07-25 08:56:40"));
                } catch (ParseException e) {
                    throw new IllegalStateException();
                }
            });
        }
        executor.shutdown();
    }
}
```

多线程资源盘点

拉勾教育

— 互联网人实战大学 —

线程安全的类

```
Thu May 01 08:56:40 CST 618104
Thu May 01 08:56:40 CST 618104
Mon Jul 26 08:00:04 CST 1
Tue Jun 30 08:56:00 CST 2020
Thu Oct 01 14:45:20 CST 16
Sun Jul 13 01:55:40 CST 20220200
Wed Dec 25 08:56:40 CST 2019
Sun Jul 13 01:55:40 CST 20220200
```

多线程资源盘点

线程安全的类

```java
public class GoodDateFormat {
    ThreadLocal<SimpleDateFormat> format = new ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue() {
            return new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss");
        }
    };
    public static void main(String[] args) {
        final GoodDateFormat faultDateFormat = new GoodDateFormat();
        ExecutorService executor = Executors.newCachedThreadPool();
        for(int i=0;i<1000;i++){
            executor.submit(()-> {
                try {
                    System.out.println(faultDateFormat.format.get().parse( source: "2020-07-25 08:56:40"));
                } catch (ParseException e) {
                    throw new IllegalStateException();
                }
            });
        }
        executor.shutdown();
```

多线程资源盘点

线程的同步方式

- 使用 Object 类中的 wait、notify、notifyAll 等函数。由于这种编程模型非常复杂，现在已经很少用了

  这里有一个关键点，那就是对于这些函数的调用，必须放在同步代码块里才能正常运行

- 使用 ThreadLocal 线程局部变量的方式，每个线程一个变量

- 使用 synchronized 关键字修饰方法或者代码块

  这是 Java 中最常见的方式，有锁升级的概念

多线程资源盘点

线程的同步方式

- 使用 Concurrent 包里的可重入锁 ReentrantLock。使用 CAS 方式实现的可重入锁
- 使用 volatile 关键字控制变量的可见性，这个关键字保证了变量的可见性，但不能保证它的原子性
- 使用线程安全的阻塞队列完成线程同步

  比如使用 LinkedBlockingQueue 实现一个简单的生产者消费者
- 使用原子变量。**Atomic***系列方法，也是使用 CAS 实现的
- 使用 Thread 类的 join 方法，可以让多线程按照指定的顺序执行

多线程资源盘点

线程的同步方式

```java
public class ProducerConsumer {
    private static final int Q_SIZE = 10;
    private LinkedBlockingQueue<String> queue = new LinkedBlockingQueue<String>(Q_SIZE);
    private volatile boolean stop = false;

    Runnable producer = () -> {
        while (!stop) {
            try {
                queue.offer(UUID.randomUUID().toString(), timeout: 1, TimeUnit.SECONDS);
            } catch (InterruptedException e) {
                //noop
            }
        }
    };
    Runnable consumer = () -> {
        while (!stop) {
            try {
                String value = queue.take();
```

多线程资源盘点

线程的同步方式

```
Runnable consumer = () -> {
    while (!stop) {
        try {
            String value = queue.take();
            System.out.println(Thread.currentThread().getName() + " | " + value);
        } catch (InterruptedException e) {
            //noop
        }
    }
};

void start() {
    new Thread(producer,  name: "Thread 1").start();
    new Thread(producer,  name: "Thread 2").start();
    new Thread(consumer,  name: "Thread 3").start();
    new Thread(consumer,  name: "Thread 4").start();
}
```

多线程资源盘点

FastThreadLocal

```
/**
 * Holder to support the {@code currentTransactionStatus()} method,
 * and to support communication between different cooperating advices
 * (e.g. before and after advice) if the aspect involves more than a
 * single method (as will be the case for around advice).
 */
private static final ThreadLocal<TransactionInfo> transactionInfoHolder =
    new NamedThreadLocal<>("Current aspect-driven transaction");
```

多线程资源盘点

FastThreadLocal

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    ...
}
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

多线程资源盘点

FastThreadLocal

```java
/**
 * Double the capacity of the table.
 */
private void resize() {
    Entry[] oldTab = table;
    int oldLen = oldTab.length;
    int newLen = oldLen * 2;
    Entry[] newTab = new Entry[newLen];
    int count = 0;

    for (int j = 0; j < oldLen; ++j) {
        Entry e = oldTab[j];
        if (e != null) {
            ThreadLocal<?> k = e.get();
            if (k == null) {
                e.value = null; // Help the GC
            } else {
                int h = k.threadLocalHashCode & (newLen - 1);
                while (newTab[h] != null)
                    h = nextIndex(h, newLen);
                newTab[h] = e;
                count++;
            }
        }
    }

    setThreshold(newLen);
    size = count;
    table = newTab;
}
```

多线程资源盘点

拉勾教育

— 互联网人实战大学 —

FastThreadLocal

```java
// Cache line padding (must be public)
// With CompressedOops enabled, an instance of this class should occupy at least 128 bytes.
public long rp1, rp2, rp3, rp4, rp5, rp6, rp7, rp8, rp9;

private InternalThreadLocalMap() {
    super(newIndexedVariableTable());
}

private static Object[] newIndexedVariableTable() {
    Object[] array = new Object[INDEXED_VARIABLE_TABLE_INITIAL_SIZE];
```

你在多线程使用中都遇到过哪些问题?

面试官会经常问你在**多线程使用中遇到的一些问题**，以此来判断你实际的应用情况

你在多线程使用中都遇到过哪些问题?

面试官会经常问你在**多线程使用中遇到的一些问题**，以此来判断你实际的应用情况

线程池的不正确使用，造成了资源分配的不可控

I/O 密集型场景下，线程池开的过小，造成了请求的频繁失败

线程池使用了 CallerRunsPolicy 饱和策略，造成了业务线程的阻塞

SimpleDateFormat 造成的时间错乱

你在多线程使用中都遇到过哪些问题?

拉勾教育

— 互 联 网 人 实 战 大 学 —

```
While (! isInterrupted () ) {
  try{
     ...
  }catch(Exception ex){
     ...
  }
}
```

你在多线程使用中都遇到过哪些问题?

拉勾教育
— 互联网人实战大学 —

```
ExecutorService executor = Executors.newCachedThreadPool();
executor.submit( ()-> {
    String s = null; s.substring(0);
});
executor.shutdown();
```

你在多线程使用中都遇到过哪些问题?


```java
while (task != null || (task = getTask()) != null) {
    w.lock();
    // If pool is stopping, ensure thread is interrupted;
    // if not, ensure thread is not interrupted.  This
    // requires a recheck in second case to deal with
    // shutdownNow race while clearing interrupt
    if ((runStateAtLeast(ctl.get(), STOP) ||
         (Thread.interrupted() &&
          runStateAtLeast(ctl.get(), STOP))) &&
        !wt.isInterrupted())
        wt.interrupt();
    try {
        beforeExecute(wt, task);
        try {
            task.run();
            afterExecute(task, t: null);
        } catch (Throwable ex) {
            afterExecute(task, ex);
            throw ex;
        }
    } finally {
```

你在多线程使用中都遇到过哪些问题?


```java
while (task != null || (task = getTask()) != null) {
    w.lock();
    // If pool is stopping, ensure thread is interrupted;
    // if not, ensure thread is not interrupted.  This
    // requires a recheck in second case to deal with
    // shutdownNow race while clearing interrupt
    if ((runStateAtLeast(ctl.get(), STOP) ||
         (Thread.interrupted() &&
          runStateAtLeast(ctl.get(), STOP))) &&
        !wt.isInterrupted())
        wt.interrupt();
    try {
        beforeExecute(wt, task);
        try {
            task.run();
            afterExecute(task, t: null);
        } catch (Throwable ex) {
            afterExecute(task, ex);
            throw ex;
        }
    } finally {
```

```java
protected void afterExecute(Runnable r, Throwable t) { }
```

关于异步

“异步，并没有减少任务的执行步骤，也没有算法上的改进，那么为什么说异步的速度更快呢？”

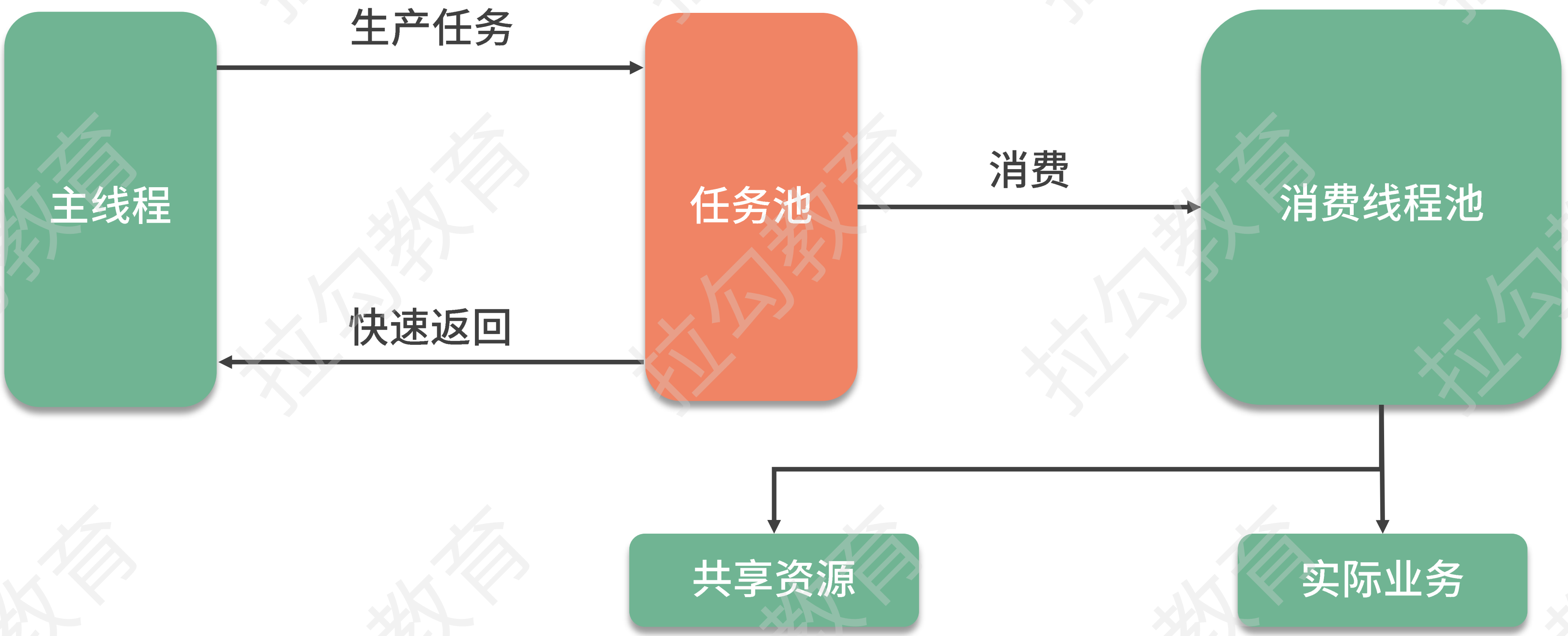
关于异步

“异步，并没有减少任务的执行步骤，也没有算法上的改进，那么为什么说异步的速度更快呢?”

异步是一种编程模型，它通过将耗时的操作转移到后台线程运行

从而减少对主业务的堵塞，所以说异步让速度变快了

关于异步



小结

本课时默认你已经有了多线程的基础知识（否则看起来会比较吃力）

所以从 CountDownLatch 的一个实际应用场景说起

谈到了线程池的两个重点：**阻塞队列**和**拒绝策略**

学习了如何在常见的框架 SpringBoot 中配置任务异步执行

对最常用的 ThreadLocal 进行了介绍，并了解了 Netty 对这个工具类的优化

Next：第13讲《案例分析：多线程锁的优化》