



JAVA 达摩班

Concurrency

季 鑫

Ashton

30 March, 2018





并发 与并行

1

理解并发和并行

并发(Concurrency): 是一个系统属性，它指运行程序，算法或者问题是可以拆分成多个独立的单元，并且可以任意顺序或者部分无序的同时执行，同时不影响运行结果

并行(Parallelism): 是一个计算类型，它运行计算过程或进程可以并行（独立的）执行

并发: 强调计算是发生在同一个时间范围内（same time frame），他们之间存在一定的依赖性

并行: 强调计算是发生在同一个时间点上（simultaneously）



并发和并行的关系

并发是泛化的并行

并行是一种并发，但并发不一定是并行

并发可以在单核处理器上实现，例如多线程，而并行必须依赖于多核（不同进程运行在不同处理器上）

并发描述的是一种问题，即两件任务如何同时发生

并行描述的是一种解决方案，即通过多核处理器并行运行

并发有多种实现方式，包括多核并行执行，时间片任务切换（task switch, time slice）



并发和多线程的关系

多线程是解决并发问题的一种方式，一种技术，一种工具
多线程是底层实现，更接近操作系统

如果是单核处理器，多线程通过线程分时占用资源的方式实现并发
如果是共享内存行多核处理器，每个线程单独运行在一个内核上，通过并行实现并发





多线程 概念及详解

2

线程概念

线程 (thread) 是一个轻量级的进程，用于执行特定的任务，并且不单独占用资源
进程 (process) 是占用资源的最小单位，它至少会有一个主线程

线程包括两种类型：用户线程 (user) 和守护线程 (daemon)

线程属于底层接口，它受线程调度器 (thread scheduler) 的管理，它属于操作系统的一部分

线程的执行是由内在管理工作由操作系统完成而非开发者，JVM不能管理线程的执行



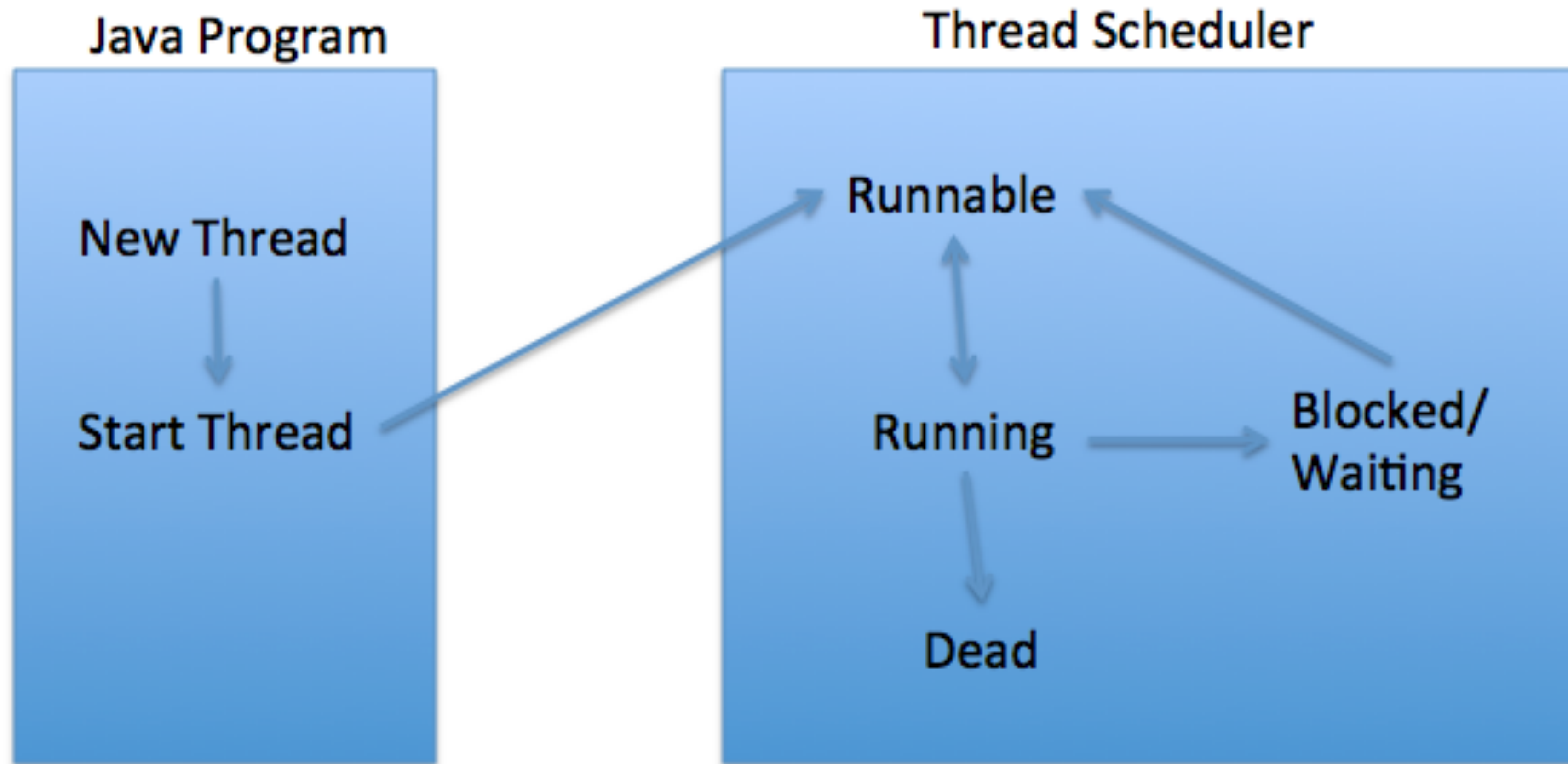
线程概念

- 创建线程需要更少的时间和资源
- 线程共享主进程的资源（数据和代码）
- 线程间上下文的切换消耗更少
- 线程间通信比进程间更容易实现

多线程：指两个或者多个线程同时运行在同一个程序中，并发的执行去解决一个问题



Thread 生命周期



线程实现

创建线程的两种方式：

1. 实现Runnable接口

2. 继承Thread类

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable()))  
            .start();  
    }  
}
```

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

Runnable vs Thread

如果类包括很多功能，那么应该实现Runnable

如果只提供单一功能，那么继承Thread

推荐通过实现Runnable的方式实现Thread，因为它java支持多接口，但不支持多继承



线程休眠（暂停）

Thread.sleep方法用于暂停当前正在运行的线程，暂停时间以毫秒为单位

休眠方法会通知线程调度器去将当前线程改为等待状态（wait），并设置一个结束时间。暂停结束后线程状态会改为运行（runnable），“继续”等待获取CPU资源继续运行。

休眠恢复（准确）时间依赖于线程调度器，等待系统分配CPU资源。



线程连接

Thread.join方法用于暂停当前线程直到指定线程结束

它相当于让指定线程等待另一线程结束后再执行，类似顺序执行



守护线程

守护线程（daemon thread）用于在后台运行线程，属于“隐形”线程

当主线程结束，进程就会结束，同时JVM会强制结束守护线程

反之，如果是线程是普通线程，当主线程结束后，进程不会结束直到普通线程完成。

即守护线程随着主进程的结束被自动关闭，而普通线程不会

守护线程一般用于非关键的系统功能，或非业务功能，例如日志，监控等，用于捕获系统资源细节和状态。

守护线程不要有IO操作，因为当进程结束后它也会结束，导致资源非正常关闭以及资源泄漏



线程安全

线程安全是指由于**线程间共享资源管理不当**导致的数据不一致性问题

数据不一致是因为更新字段不是原子操作，它包括三个步骤：读取当前值，获取更新后的值，更新值


线程安全要解决的，就是保证程序在多线程环境下的安全运行



死锁

当线程之前相互持有对方需要的资源，都不会释放，且都在等待

避免死锁方法：

1. **避免嵌套锁**：最常见的死锁。避免持有一个锁，而去申请另一个锁。一般同时只持有一个锁不会产生死锁。
 2. **按需取锁**：只申请当前（工作）需要的锁资源
 3. **避免无限等待**：设定线程死锁时间上限
- 

线程安全

实现线程安全有如下方案：

1. **同步锁**：最简单和广泛使用的线程安全方案
2. **原子包装类**：继承自`java.util.concurrent.atomic`，例如`AtomicInteger`
3. **锁机制**：定义自`java.util.concurrent.locks`
4. **线程安全类**：集合类如`ConcurrentHashMap`
5. **挥发方法**：使用`volatile`变量保证线程从内存读取数据，而不是线程缓存



线程安全威胁

如下代码会锁住myObject，当它被锁住，myObject的方法doSomething也会被阻塞，进入死锁导致 Denial of Service (DoS) 攻击

```
public class MyObject {  
    public synchronized void doSomething() {  
        // ...  
    }  
}
```

```
// Hackers code  
MyObject myObject = new MyObject();  
synchronized (myObject) {  
    while (true) {  
        Thread.sleep(Integer.MAX_VALUE);  
    }  
}
```


线程安全威胁

如下代码也存在DoS攻击风险，因为它锁住了MyObject类而不释放，导致死锁

```
public class MyObject {  
    public static synchronized void doSomething() {  
        // ...  
    }  
}
```

```
synchronized (MyObject.class) {  
    while (true) {  
        Thread.sleep(Integer.MAX_VALUE);  
    }  
}
```

线程安全威胁

如果lock对象是公共变量，并且更改它的引用，就能在多个线程上并行执行lock。

```
public class MyObject {  
    public Object lock = new Object();  
  
    public void doSomething() {  
        synchronized (lock) {  
            // ...  
        }  
    }  
}
```

//untrusted code

```
MyObject myObject = new MyObject();  
myObject.lock = new Object();
```

线程转储

线程转储（Thread dump）列出当前线程所有信息，它可以用于分析应用瓶颈和死锁状态。

Java自带分析工具：

1. `jstack <PID>`
2. `jcmd <PID> Thread.print`



线程转储

线程分析步骤：

1. ps -A | grep java

或者

jconsole

#获取进程PID

#提供可视化界面查看java运行进程，包括PID

2.1. kill -QUIT <PID>

#此步骤用于死锁分析，杀死进程并列分析结果

2.2. jstack <PID>

或者

jcmd <PID> Thread.print

#直接查看进程线程信息

#java 8自带命令行工具



Java并发 工具及框架

3

并发工具包

Java并发开发包，`java.util.concurrent`，提供了大量辅助创建并发应用的工具

Executor

Future

Semaphore

DelayQueue

ThreadLocal

ExecutorService

CountDownLatch

ThreadFactory

Locks

Atomic

ScheduledExecutorService

CyclicBarrier

BlockingQueue

Phaser

...



Executor

Executor是一个接口，它代表执行给定任务的对象。

解耦任务执行流和实际的任务执行机制

Executor并不要求任务异步执行

```
public class Invoker implements Executor {  
    @Override  
    public void execute(Runnable r) {  
        r.run();  
    }  
}
```

```
public void execute() {  
    Executor executor = new Invoker();  
    executor.execute( () -> {  
        // task to be performed  
    });  
}
```

ExecutorService

ExecutorService是一个完整的异步处理解决方案
它管理内存队列，并且基于线程可用性调度提交的任务

```
ExecutorService executor = Executors.newFixedThreadPool(10);  
executor.submit(() -> {  
    new Task();  
});
```

它提供方法终结任务： shutdown(), shutdownNow()

ScheduledExecutorService和ExecutorService类似，它可以实现定期执行任务。
它提供方法： schedule, scheduleAtFixedRate和scheduleWithFixedDelay

线程池ThreadPool

线程池用于管理工作线程，它包含一个等待执行的队列。

它可以由ThreadPoolExecutor创建，一个继承于AbstractExecutorService的类
它也可以由Executors工厂类创建

```
ExecutorService executor = Executors.newFixedThreadPool(5);  
for (int i = 0; i < 10; i++) {  
    Runnable worker = new WorkerThread("" + i);  
    executor.execute(worker);  
}  
executor.shutdown();  
while (!executor.isTerminated()) {  
}
```


Atomic

使用锁会带来性能问题

上下文的切换（运行，暂停，继续）的消耗大于计算本身，例如计数器
业界倾向于用非阻塞算法减少性能消耗，例如原子操作

Compare-and-swap 比较交换法(CAS)

M: 当前运行中的内存位置

A: 当前已有的变量值

B: 准备设置的新变量值

如果M和A匹配，将M的值改为B；否则，不更新M



Atomic

思路：合并三个步骤为一个原子不可分的操作，当多个线程通过CAS算法操作，不会产生暂停的上下文切换，只是被通知不能更新，继续执行其他任务。

常用的类：AtomicInteger, AtomicLong, AtomicBoolean 和 AtomicReference

常用的方法：get(), set(), compareAndSet(), incrementAndGet()等



Lock

Java 1.5引入到java.util.concurrent.lock包中的接口，提供了更丰富和扩展的功能

Lock和Synchronized block区别

- 同步锁完全包含在一个方法中，而锁通过lock()和unlock()是两个独立的方法
- 同步锁不能保证公平性，当一个线程释放掉锁后，谁都可以取得，无法估计到等待最久的线程等情况
- 同步锁在线程无法获取资源时会进入阻塞状态，而锁提供了tryLock方法，减少不必要等待
- 同步锁的线程进入阻塞后无法停止，而锁提供了lockInterruptibly方法，强制结束当前线程



Lock接口

常用方法：

`void lock()`

`void lockInterruptibly()` 允许阻塞的线程终止，继续执行抛出异常`java.lang.InterruptedException`

`boolean tryLock()` 非阻塞锁，尝试立刻获取锁

`boolean tryLock(long timeout, TimeUnit timeUnit)`

`void unlock()`

```
Lock lock = ...;
```

```
lock.lock();
```

```
try {
```

```
    // access to the shared resource
```

```
} finally {
```

```
    lock.unlock();
```

```
}
```



几种Lock实现

1. ReentrantLock

这是最常用Lock接口实现类，它和synchronized类似，但提供更多工具方法。例如持有锁，等待锁

什么是重入锁？

同步锁天生就具有重入属性。

如果一个线程通过一个同步锁锁住一个监视器对象（monitor object），而另外一个同步锁需要锁在同一个监视器对象上，那么该线程也能操作该同步锁。

2. ReentrantReadWriteLock

它包括只读锁和写锁组成，只读锁可以被多个线程同时拥有，而写锁同时只能被一个线程拥有。

3. StampedLock

Java 8引入，和ReadWriteLock类似，增加stamp标记

4. Condition

类似wait-notify模型，条件对象由锁对象创建。await方法类似wait方法，signal方法和signalAll方法类似notify和notifyAll方法

```
public class Test{

    public synchronized foo(){
        //do something
        bar();
    }

    public synchronized bar(){
        //do some more
    }
}
```

几种Lock实现

1. ReentrantLock

```
public class SharedObject {  
    //...  
    ReentrantLock lock = new ReentrantLock();  
    int counter = 0;  
  
    public void perform() {  
        lock.lock();  
        try {  
            // Critical section here  
            count++;  
        } finally {  
            lock.unlock();  
        }  
    }  
    //...  
}
```

```
public void performTryLock(){  
    //...  
    boolean isLockAcquired = l  
        ock.tryLock(1, TimeUnit.SECONDS);  
  
    if(isLockAcquired) {  
        try {  
            //Critical section here  
        } finally {  
            lock.unlock();  
        }  
    }  
    //...  
}
```

几种Lock实现

```
Map<String,String> syncHashMap = new HashMap<>();
ReadWriteLock lock = new ReentrantReadWriteLock();
```

```
Lock writeLock = lock.writeLock();
```

```
public void put(String key, String value) {
    try {
        writeLock.lock();
        syncHashMap.put(key, value);
    } finally {
        writeLock.unlock();
    }
}
```

```
public String remove(String key){
    try {
        writeLock.lock();
        return syncHashMap.remove(key);
    } finally {
        writeLock.unlock();
    }
}
```

2. ReentrantReadWriteLock

```
Lock readLock = lock.readLock();
```

```
public String get(String key){
    try {
        readLock.lock();
        return syncHashMap.get(key);
    } finally {
        readLock.unlock();
    }
}
```

```
public boolean containsKey(String key) {
    try {
        readLock.lock();
        return syncHashMap.containsKey(key);
    } finally {
        readLock.unlock();
    }
}
```

几种Lock实现

3. StampedLock

```
Map<String,String> map = new HashMap<>();
private StampedLock lock = new StampedLock();

public void put(String key, String value){
    long stamp = lock.writeLock();
    try {
        map.put(key, value);
    } finally {
        lock.unlockWrite(stamp);
    }
}

public String get(String key) throws InterruptedException {
    long stamp = lock.readLock();
    try {
        return map.get(key);
    } finally {
        lock.unlockRead(stamp);
    }
}
```

```
long stamp = lock.tryOptimisticRead();
String value = map.get(key);

if(!lock.validate(stamp)) {
    stamp = lock.readLock();
    try {
        return map.get(key);
    } finally {
        lock.unlock(stamp);
    }
}
return value;
```


Folk / Join

Folk/Join框架从java 7开始引入，其思想是分治（**divide and conquer**），利用尽可能多的处理器内核

folk，是只将任务迭代分割成多个独立的小任务，直到小到能异步执行；
join，或者将所有完成的子任务合并为最终结果，或者等待所有子任务全部完成

ForkJoinPool是框架的核心，它是ExecutorService的实现，用于管理工作线程，并且提供了获取线程池状态和性能的信息。

ForkJoinTask：返回值为void的任务 RecursiveTask<V>：有返回值的任务

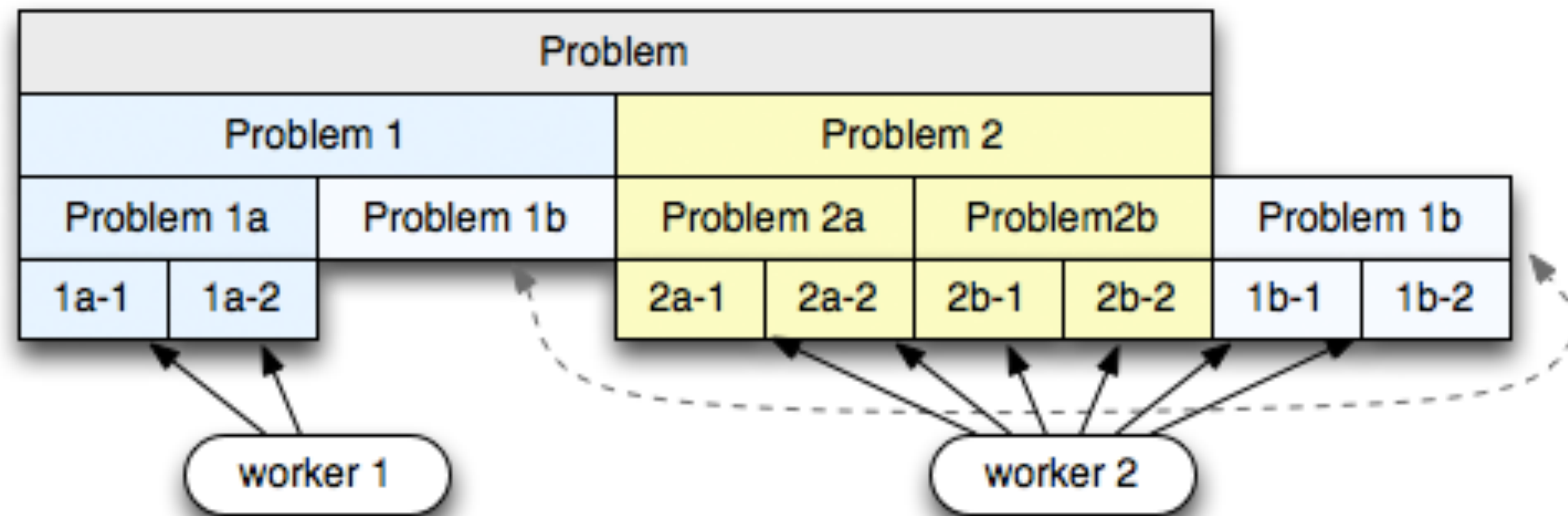


Folk / Join

每个线程都有自己的双向队列（**deque**），存储着它负责执行的任务。

工作窃取算法（work-stealing）是Folk/Join中负责平衡线程工作量的重要算法。

工作线程首先按照从头到尾的顺序完成自己的双向队列，如果它提前完成，那么就从当前繁忙的线程队列，或者全局队列的尾部“偷”任务过来帮忙完成。



并发问题方案

1. Thread

程序好坏严重依赖程序员素质，例如停止，但可以拥有绝对控制权，更接近系统
线程的数量不好掌控
快速，简单

2. Executors

ExecutorService和CompletionServices

抽象了Thread的复杂性，提供了大量封装好的实现，简化开发流程，适合大型系统

3. ForkJoinPool (FJP) with parallel streams

需要经验，评估系统复杂度

虽然并行由JVM管理，但开发者需要写一部分并行的代码

4. Actor

当前最流行的消息模型，框架丰富Akka

作业

1. 杂货店程序多线程版
2. 查阅和理解volatile变量，以及在线程操作的应用
3. 总结executorService的execute, submit和schedule方法





Thanks!

Any questions?

