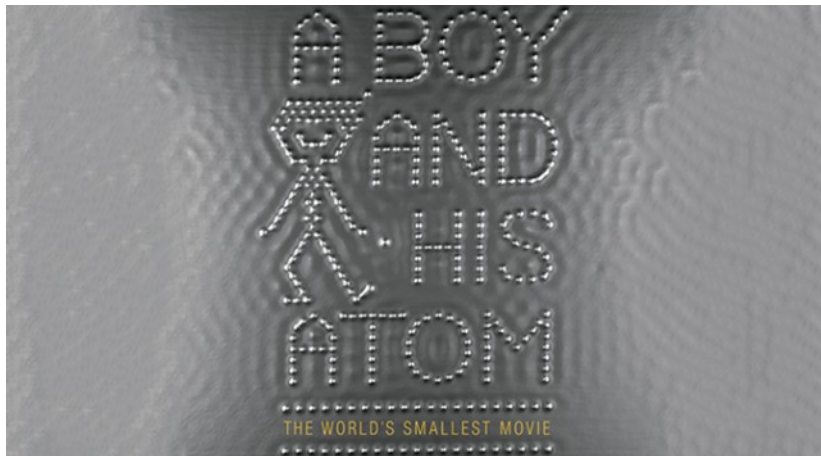


# Разработка многопоточных приложений на Java

Алексей Владыкин

2 декабря 2013

- 1 Атомарные типы
- 2 Примитивы синхронизации
- 3 Коллекции
- 4 Executors



- Пакет `java.util.concurrent.atomic`
- `AtomicBoolean`  
`AtomicInteger`  
`AtomicLong`  
`AtomicReference<V>`
- Операции:  
`V get()`  
`void set(V newValue)`  
`boolean compareAndSet(V expect, V update)`

- Примитив `compareAndSet` позволяет реализовывать другие операции
- Пример из `AtomicInteger`:

```
public final int incrementAndGet() {  
    for (;;) {  
        int current = get();  
        int next = current + 1;  
        if (compareAndSet(current, next))  
            return next;  
    }  
}
```



# Semaphore

- Класс `java.util.concurrent.Semaphore`
- Ограничивает одновременный доступ к ресурсу
- В отличие от `synchronized`-блока, одновременно могут работать несколько потоков (но не более заданного N)
- Операции:  
`void acquire()`  
`void release()`

```
Semaphore semaphore = new Semaphore(10);

semaphore.acquire();
try {
    // up to 10 threads may
    // execute this code concurrently
} finally {
    semaphore.release();
}
```



# CountDownLatch

- Класс `java.util.concurrent.CountDownLatch`
- Обеспечивает точку синхронизации между  $N$  потоками (несколько потоков могут дожидаться друг друга и потом стартовать одновременно)
- Операции:  
`void await()`  
`void countDown()`

```
CountDownLatch latch = new CountDownLatch(10);  
  
// this call blocks until latch.countDown()  
// is called at least 10 times  
latch.await();
```

# CyclicBarrier

- Класс `java.util.concurrent.CyclicBarrier`
- Вариант `CountDownLatch`, допускающий повторное ожидание

# ReentrantLock

- Класс `java.util.concurrent.locks.ReentrantLock`
- Обеспечивает взаимное исключение потоков, аналогичное `synchronized`-блокам
- Операции:  
`lock()`  
`unlock()`

```
Lock lock = new ReentrantLock();  
  
lock.lock();  
try {  
    doSomething();  
} finally {  
    lock.unlock();  
}
```

# Condition

- Класс `java.util.concurrent.locks.Condition`
- Аналог `wait/notify`
- Привязан к `Lock`'у
- У одного `Lock`'а может быть много `Condition`'ов

```
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();

lock.lock();
try {
    while (!conditionSatisfied()) {
        condition.await();
    }
} finally { lock.unlock(); }

// somewhere else in our program
lock.lock();
try {
    condition.signal();
} finally { lock.unlock(); }
```

# ReentrantReadWriteLock

- Класс `java.util.concurrent.locks.ReentrantReadWriteLock`
- Поддерживает разделение доступа на чтение и на запись



```
ReadWriteLock lock = new ReentrantReadWriteLock();

// somewhere in our program
lock.readLock().lock();
try {
    readOnlyOperation();
} finally {
    lock.readLock().unlock();
}

// somewhere else in our program
lock.writeLock().lock();
try {
    modifyingOperation();
} finally {
    lock.writeLock().unlock();
}
```



- Пакет `java.util.concurrent`
- Многопоточные варианты стандартных коллекций:
  - `ConcurrentHashMap`
  - `ConcurrentSkipListMap`
  - `ConcurrentSkipListSet`
  - `CopyOnWriteArrayList`
  - `CopyOnWriteArraySet`
- Более эффективны, чем полностью синхронизованные коллекции `java.util.Collections.synchronizedCollection()`

# ConcurrentLinkedQueue

- Класс `java.util.concurrent.ConcurrentLinkedQueue<E>`
- Реализация очереди, поддерживающая одновременный доступ из многих потоков, при этом не использующая блокировки
- Операции:
  - `boolean offer(E e)`
  - `E poll()`
  - `E peek()`

# BlockingQueue

- Интерфейс `java.util.concurrent.BlockingQueue<E>`
- Очередь, поддерживающая ограничение по размеру и операции ожидания
- Операции:  
`void put(E e)`  
`E take()`
- Реализации:  
`LinkedBlockingQueue`, `ArrayBlockingQueue`, ...



- Класс `java.util.concurrent.ExecutorService` и его соседи
- Инфраструктура для выполнения задач в несколько потоков
- Инкапсулирует создание потоков, организацию очереди задач, распределение задач по потокам

# ExecutorService

- `Future<?> submit(Runnable task)`
- `<T> Future<T> submit(Callable<T> task)`
- `void shutdown()`
- `List<Runnable> shutdownNow()`



# Executors

- Класс `java.util.concurrent.Executors`
- `ExecutorService newSingleThreadExecutor()`
- `ExecutorService newFixedThreadPool(int nThreads)`
- `ExecutorService newCachedThreadPool()`

# ForkJoinPool

- Класс `java.util.concurrent.ForkJoinPool`
- Вариант `ExecutorService`, в котором выполняющиеся задачи могут динамически порождать подзадачи
- Принимает на исполнение `ForkJoinTask`

## Что сегодня узнали

- Какие есть примитивы синхронизации, помимо встроенных мониторов
- Какие коллекции использовать в многопоточных программах
- Как организовать параллельное выполнение множества задач, не занимаясь низкоуровневым программированием потоков