

Многопоточное программирование в Java

Croc Java School



Многопоточность

Процесс может состоять из нескольких потоков, выполняющихся без определенного порядка во времени.

Разделение позволяет эффективнее использовать ресурсы.

- Процесс имеет минимум один главный поток.
- Все потоки разделяют адресное пространство процесса.
- Стек у каждого потока свой.



Синхронизация потоков

Потоки редко выполняются изолированно. Часто различным потокам приходится останавливать/возобновлять свое исполнение в зависимости от состояния других потоков.

Причины синхронизации:

- Совместный доступ к ресурсам (память, файлы, устройства ввода-вывода).
- Ожидание результата вычислений.



Многопоточность в Java

1. Базовая поддержка на уровне языка (Thread, synchronized).
2. Высокоуровневые API `java.util.concurrent`.

Базовые примитивы



Интерфейс Runnable

```
public interface Runnable {  
    public abstract void run();  
}
```

Сама по себе реализация интерфейса Runnable не обеспечивает выполнение в отдельном потоке.



Thread

Для запуска инструкций в отдельном потоке необходимо используется класс Thread.

```
public class Thread implements Runnable {  
  
    private Runnable target;  
  
    public void run() {  
        if (target != null) {  
            target.run();  
        }  
    }  
}
```



Два способа создания потока

1. Создать класс, наследующий `Thread` и переопределить метод `run()`.
2. Создать класс, реализующий интерфейс `Runnable` и передать его экземпляр в конструктор `Thread`.

**Пример. Создать поток,
печатающий в цикле
заданный символ**

—



Создание потока. Способ 1

```
public class CharThread extends Thread {  
    private final char c;  
  
    public CharThread(char c) {  
        this.c = c;  
    }  
  
    public void run() {  
        while (true) {  
            System.out.print(c);  
        }  
    }  
}
```

ДИСКЛЕЙМЕР 🙄

**Создавать потоки с помощью
класса Thread стоит только в
учебных целях**



Создание потока. Способ 2

```
public class CharRunnable implements Runnable {  
    private final char c;  
  
    public CharRunnable(char c) {  
        this.c = c;  
    }  
  
    public void run() {  
        while (true) {  
            System.out.print(c);  
        }  
    }  
}
```



Запуск потока

Для запуска потока используется метод `Thread.start()`. Поток начинает исполняться параллельно с текущим.

```
Thread t1 = new CharThread('a');
```

```
Thread t2 = new Thread(new CharRunnable('b'));
```

```
t1.start();
```

```
t2.start();
```

```
t1.join();
```

```
t2.join();
```

```
[out]
```

```
bbbbbbbaabbaabbaabbbbaabbaabbaabbaabbaabbaabbaabbaabbaabbaa  
bbaabbaabbaa...
```



Запуск потока

После вызова метода `start()` вызывающий поток продолжает свое исполнение, не дожидаясь завершения метода.

Параллельно начинает исполняться метод `run()` нового потока.

run() не приводит к параллельному исполнению

Методы `Runnable.run()` и `Thread.run()` выполняют инструкции в текущем потоке!

```
Thread t1 = new CharThread('a');
```

```
Runnable t2 = new CharRunnable('b');
```

```
t1.run();
```

```
t2.run(); // will never run
```

[out]

[illegible]

**Пример. Печатать символы с
разной частотой**

—



Thread.sleep()

Для приостановки исполнения текущего потока используется метод `Thread.sleep()`.

```
public static native void sleep(long millis)  
    throws InterruptedException;
```

Javadoc:

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors.



Вывод символа с заданной частотой

```
public class CharRunnable implements Runnable {
    private final char c;
    private final float frequency;
    public CharRunnable(char c, float frequency) {
        this.c = c;
        this.frequency = frequency;
    }
    public void run() {
        while (true) {
            System.out.print(c);
            try {
                Thread.sleep((long)(1e3 / frequency));
            } catch (InterruptedException e) {
                break; // stop execution if thread was interrupted
            }
        }
    }
}
```



Символ 'a' будем печатать в два раза чаще, чем 'b'

```
Thread t1 = new Thread(new CharRunnable('a', 10));  
Thread t2 = new Thread(new CharRunnable('b', 5));  
t1.start();  
t2.start();
```

[out]

```
ababaabaabaabaabaabaabaabaabaabaabaabaabaabaabaabaabaab  
aabaabaabaabaabaabaabaabaabaabaabaab...
```

**Пример. Каждые 16 символов
добавлять перевод строки**

—



Выравнивание

```
public class CharRunnable implements Runnable {  
    private static int numPrinted = 0;  
  
    public void run() {  
        while (true) {  
            System.out.print(c);  
            numPrinted++;  
            if (numPrinted % 16 == 0)  
                System.out.println();  
            // sleep...  
        }  
    }  
}
```



Посмотрим на вывод

[out]

ababaabaabaabaab

aabaabaabaabaaba

abaabaabaabaabaa

baabaabaabaabaab

aabaabaabaabaaba

abaabaababaabaab

aabaabaabaabaaba

...

Вроде бы все хорошо, но, что если печатать символы чаще?



Печатаем символы быстрее

```
Thread t1 = new Thread(new CharRunnable('a', 10_000));  
Thread t2 = new Thread(new CharRunnable('b', 5_000));  
t1.start();  
t2.start();
```



Смотрим на вывод

[out]

babababababab

abababababababab

abababababaaaaba

bababababababababbbbbbbbbbbbbbb

babababababababb

babababababababbbbbbbbbbbbbbb

aa

bbbbbbbbbbbbbbbbbb

...

Почему все сломалось?



Потокобезопасность

Класс `CharRunnable` не потокобезопасный.

Класс называется потокобезопасным, если его поведение корректно вне зависимости от способа использования в многопоточной среде.



Классы проблем многопоточных приложений

Thread interference

Атомарность чтения и записи, неопределенность параллелизма.

Memory consistency

Синхронизация изменений в данных между разными потоками: happens-before.



Как бороться с этими проблемами:

1. не использовать общие переменные в разных потоках;
2. обеспечить неизменяемость общих переменных (immutability);
3. синхронизировать доступ к общим переменным, так что одновременно с ними будет работать только один поток.



Критическая секция

Блок инструкций, в котором производится доступ к общим ресурсам, который не должен одновременно исполняться более, чем одним потоком.



BRIAN GOETZ

WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA



JAVA CONCURRENCY IN PRACTICE



Java Concurrency in Practice

Brian Goetz + команда `java.util.concurrent`

synchronized



Состояние гонки (неопределенность параллелизма)

Race conditions are where the correctness of an algorithm is dependent on lucky timing in the ordering of actions taken by each thread.

The most common race condition is check-then-act.

Not all data races are race conditions and not all race conditions are data races. check-then-act is when you make an observation about the system then take an action, however by the time you take the action the observation could have become invalid.



check-then-act в классе CharRunnable

Общее состояние: переменная numPrinted.

check:

```
if (numPrinted % 16 == 0)
```

act:

```
System.out.println(); // условие check уже может оказаться нарушенным
```

check-then-act в классе CharRunnable

Поток 1

```
// numPrinted: 15
1 System.out.print(c); // print a
2 numPrinted++; // 16
3 if (numPrinted % 16 == 0)
.
.
6 System.out.println();
```

Поток 2

```
// numPrinted: 15
.
.
.
4 System.out.print(c); // print b
5 numPrinted++; // 17
.
7 if (numPrinted % 16 == 0)
8     System.out.println();
```

На самом деле все еще
сложнее из-за проблем с
memory consistency

—



Memory consistency

При чтении данных из памяти поток сначала копирует блок памяти в “рабочую” память потока, откуда в свою очередь читается значение.

При записи, значение сначала фиксируется в рабочей памяти, и только оттуда переносится в основную.

Рабочие блоки потоков при этом изолированы друг от друга.

Если значение уже присутствует в рабочей памяти, то чтение может не инициировать копирование основной памяти в целях оптимизации. Аналогично запись не обязательно должна приводить к записи блока в основную память.



synchronized

Простой способ устранить гонку и обеспечить memory consistency — использовать ключевое слово `synchronized`.

`synchronized` может быть применен:

- к методам, в том числе и статическим;
- отдельным блокам инструкций.



synchronized

Когда мы используем ключевое слово `synchronized` Java обеспечивает сериализацию (эксклюзивное исполнение) критических секций различными потоками. Для этого в платформе предусмотрены мониторы (intrinsic locks), связанные с экземплярами классов.

У каждого объекта есть свой intrinsic/monitor lock.



Исправление CharRunnable

Ок, должно быть достаточно пометить метод как `synchronized` попробуем.

```
public synchronized void run() {  
    while (true) {  
        System.out.print(c);  
        numPrinted++;  
        if (numPrinted % 16 == 0)  
            System.out.println();  
        try {  
            Thread.sleep((long)(1e3 / frequency));  
        } catch (InterruptedException e) {  
            break;  
        }  
    }  
}
```



Не помогло

[out]

aabbaabbbbaabbbbaabbb

abbaabbaabbbba

bbbaaabbaaabbbbaabb

abbaabbaabbaab

aabbaabbbbaabbbbaabb


abbaabbaabbaab

aabbbaabbaabbabb

aabbabbabbbaabba

. . .

Почему не помогло?



`synchronized` упорядочивает блоки в контексте конкретных экземпляров (или экземпляра `CharRunnable.class` в случае статических методов). Так как мы используем два разных экземпляра `CharRunnable`:

```
Thread t1 = new Thread(new CharRunnable('a', 10_000));  
Thread t2 = new Thread(new CharRunnable('b', 5_000));
```

методы `run` синхронизируются для этих экземпляров, но между разными объектами по-прежнему допускаются пересечения.



Общий контекст синхронизации

```
public class CharRunnable implements Runnable {  
    private static final Object lock = new Object();  
  
    public void run() {  
        while (true) {  
            synchronized (lock) {  
                System.out.print(c);  
                numPrinted++;  
                if (numPrinted % 16 == 0)  
                    System.out.println();  
            }  
            // sleep...  
        }  
    }  
}
```



Теперь все хорошо

[out]

babbaabbbaabbbaa

babbabbbaabbbaaaa

bbaaabbbaabbbaabb

aabbabbabbbaaabba

abbaabbaabbaabab

aabbbaabbaabbbaaa

abbaabaaabbbbaaab

bbbababaaabbbaaab

. . .

Побочные эффекты синхронизации

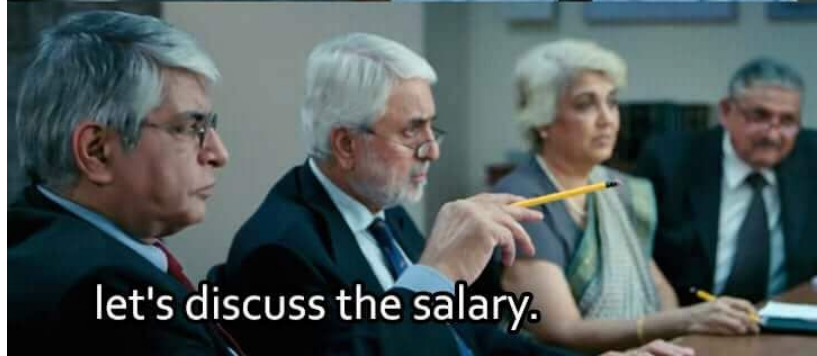
Thread Contention

Потоки могут перейти в состояние взаимной блокировки.

- Deadlock
- Livelock

Из-за конкуренции за доступ к ресурсу и частых переключений контекста может наблюдаться замедление работы.

- Starvation



volatile



volatile

По умолчанию чтение и запись атомарны для ссылок и примитивных типов, за исключением `double` и `long`.

Чтение и запись атомарны для всех переменных, помеченных ключевым словом `volatile`.

Дополнительно ключевое слово `volatile` гарантирует выполнение `happens-before`, что позволяет устранить проблемы `memory consistency`.



Использование переменной состояния для завершения потока

```
static class Task implements Runnable {  
    private boolean cancelled;  
  
    public void cancel() {  
        cancelled = true;  
    }  
  
    public void run() {  
        while (!cancelled) {  
            System.out.println("running...");  
        }  
    }  
}
```



Использование переменной состояния для завершения потока

```
Task task = new Task();  
new Thread(task).start();  
// sleep...  
task.cancel();
```




В общем случае измененное значение переменной `cancelled` другие потоки могут не “увидеть”.



Использование переменной состояния для завершения потока

```
static class Task implements Runnable {  
    private volatile boolean cancelled;  
  
    public void cancel() {  
        cancelled = true;  
    }  
  
    public void run() {  
        while (!cancelled) {  
            System.out.println("running...");  
        }  
    }  
}
```



Ключевое слово `volatile` не обеспечивает синхронизацию доступа к ресурсам, но гарантирует видимость изменений между потоками.



Thread.interrupt()

Метод `interrupt()` сигнализирует потоку о необходимости завершиться.

Тем не менее, для корректного завершения поток должен обработать этот сигнал.

- Обработать `InterruptedException` и завершать работу.
- Периодически проверять статус `Thread.currentThread().isInterrupted()`, если он не вызывает методы, генерирующие `InterruptedException`.



Прерывание потока

```
static class Task implements Runnable {  
  
    public void run() {  
        while (!Thread.currentThread().isInterrupted()) {  
            System.out.println("running...");  
            // do some work...  
            try {  
                Thread.sleep(1_000L);  
            } catch (InterruptedException e) {  
                Thread.currentThread().interrupt();  
                break;  
            }  
        }  
    }  
}
```



Прерывание потока

```
Thread t = new Thread(new Task());  
t.start();  
// sleep...  
t.interrupt();
```




Важные особенности

Вызов метода `Thread.interrupted()` сбрасывает статус прерывания потока.

`Thread.currentThread().isInterrupted()` — нет.

Если поток не пробрасывает `InterruptedException`, необходимо явно восстанавливать статус `interrupted` потока: `Thread.currentThread().interrupt();`

ReentrantLock



ReentrantLock

```
private byte[] bytes;  
private ReentrantLock lock = new ReentrantLock();  
  
public void append(byte b) {  
    lock.lock();  
    try {  
        // add b to bytes array  
    } finally {  
        lock.unlock();  
    }  
}
```



Condition await

```
private byte[] bytes;
private ReentrantLock lock = new ReentrantLock();
private Condition appended = lock.newCondition();

public void process() throws InterruptedException {
    lock.lock();
    try {
        while (bytes.length < 64 * 1024)
            appended.await();
        // process array
    } finally {
        lock.unlock();
    }
}
```



Condition signal/signalAll

```
private byte[] bytes;
private ReentrantLock lock = new ReentrantLock();
private Condition appended = lock.newCondition();

public void append(byte b) {
    lock.lock();
    try {
        // add b to bytes array
        appended.signalAll();
    } finally {
        lock.unlock();
    }
}
```



Spurious wakeup

Javadoc одно из условий пробуждения потока:

A "spurious wakeup" occurs.

Завершение ожидания не гарантирует выполнения условия, даже если сигнал отправляется при наступлении этого условия.

```
if (bytes.length >= 64 * 1024)
    appended.signalAll();
```

```
appended.await();
// нет гарантий относительно размера bytes
```

Intrinsic Locks



Intrinsic Lock

Каждый объект Java имеет встроенный объект блокировки и один связанный с ним Condition.

```
synchronized (obj) {  
    // ...  
}
```

То же самое, что и

```
obj.lock.lock();  
try {  
    // ...  
} finally {  
    lock.unlock();  
}
```




Object wait

```
synchronized (obj) {  
    while (/* check condition holds */) {  
        obj.wait(timeout);  
    }  
}
```



Object notify/notifyAll

```
synchronized (obj) {  
    obj.notify(); // notify any waiting thread  
    // or  
    obj.notifyAll(); // notify all awaiting objects  
}
```

Чем пользуемся на практике



java.util.concurrent

- *Locks*: Lock, Condition, ReadWriteLock
- *Atomics*: AtomicInteger, AtomicReference, ...
- *Executors*: Callable, ExecutorService, Future, CompletableFuture
- *Queues*: BlockingQueue, DelayQueue
- *Synchronizers*: Semaphore, CountdownLatch, CyclicBarrier
- *Concurrent collections*: ConcurrentHashMap, CopyOnWriteArrayList



ExecutorService

Создание и управление потоками дорого обходится (по памяти и времени создания).
Эффективно переиспользовать потоки при их освобождении, а не пересоздавать под каждую новую задачу.

Этим занимаются пулы потоков.



Callable

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

Абстракция для представления задачи.

Отличия от Runnable:

- задача возвращает значение
- предусматривает возможность выбросить исключение



Пример задачи

Посчитать количество слов в массиве, длина которых не меньше заданной.

```
public class CountWords implements Callable<Long> {  
    private String[] words;  
    private int threshold;  
  
    // конструктор опущен  
  
    public Long call() {  
        long count = 0;  
        for (String word : words) {  
            if (word.length() >= threshold)  
                count++;  
        }  
        return count;  
    }  
}
```



ExecutorService

```
public interface ExecutorService extends Executor {  
    <T> Future<T> submit(Callable<T> task);  
    // ...  
}
```

Метод `submit` — предпочтительный способ порождения параллельных задач в Java.



Future

Абстракция для задачи, которая может быть еще не завершена (исполняется в параллельном потоке).

```
public interface Future<V> {  
    boolean cancel(boolean mayInterruptIfRunning);  
    boolean isCancelled();  
    boolean isDone();  
    V get() throws InterruptedException, ExecutionException;  
    V get(long timeout, TimeUnit unit)  
        throws InterruptedException, ExecutionException, TimeoutException;  
}
```



Пример параллельных вычислений

Подсчитать суммарное количество слов с заданным ограничением по минимальной длине в двух массивах

```
private ExecutorService pool = Executors.newFixedThreadPool(3);

public long countWordsParallel(String[] words1, String[] words2, int threshold)
    throws ExecutionException, InterruptedException {

    Future<Long> result1 = pool.submit(new CountWords(words1, threshold));
    Future<Long> result2 = pool.submit(new CountWords(words2, threshold));
    return result1.get() + result2.get();
}
```



Executors

Распространенные реализации ExecutorService

```
Executors.newFixedThreadPool(nThreads)
```

```
Executors.newSingleThreadExecutor()
```

```
Executors.newCachedThreadPool()
```

```
Executors.newScheduledThreadPool(corePoolSize)
```



Освобождение потоков пула

```
ExecutorService pool = Executors.newCachedThreadPool();
```

// прекратить принимать задачи, но не прерывать текущие

```
pool.shutdown();
```

// прервать в том числе текущие задачи

```
pool.shutdownNow();
```