

Java - multithreading

Anastasiya Solodkaya, Denis Stepulenok

LevelUP

3 марта 2017 г.

1 Основы многопоточности

2 Thread и Runnable

3 Проектирование приложений

4 Механизм synchronized

5 Как синхронизировать доступ к общедоступному состоянию?

6 Visibility

7 Практики проектирования объектов

8 Многопоточность в java

Что такое многопоточность

- Способность программы выполнять разные задачи в разных потоках.
- Потоки выполняются без предписанного порядка

Зачем нужна многопоточность?

- Возможность (вероятно) увеличить продуктивность приложения.
- Возможность распределить приложение (использовать больше вычислительных ресурсов).

Поток vs. процесс

- **Процесс** - абстракция запущенной программы. Процессы могут общаться друг с другом, не имеют общей используемой памяти.
- **Поток** - единица выполнения вычислений внутри потока. Обычно несколько потоков имеют общие данные (память), которые разделяют друг с другом.

Опасности многопоточности

- Использование общих данных
- Ограничения памяти
- Возможные внутренние оптимизации

Ошибки многопоточности

- Их очень трудно поймать и повторить
- Вы можете не наблюдать их годами
- Их трудно отлаживать

Многопoтoчнoсть: пpoектиpoвaниe

- Лyчшe зaклaдывaть "пpaвильный" дизaйн зaрaнee
- Oчeнь мнoгие oшибки "исчeзaют" пpи пpaвильнoм пpoектиpoвaнии c т.з. OOP, и пpи пpaвильнoм paспpeдeлeнии oбязaннoстей мeждy клaccaми.

- 1 Основы многопоточности
- 2 Thread и Runnable
- 3 Проектирование приложений
- 4 Механизм synchronized
- 5 Как синхронизировать доступ к общедоступному состоянию?
- 6 Visibility
- 7 Практики проектирования объектов
- 8 Многопоточность в java

То, на чем все основано - Thread и Runnable

- **Thread** - класс, представляющий собой поток.
- **Runnable** - класс, который представляет собой задачу, которая может (в частности) выполняться в потоке.

Thread + run()

```
Thread t = new Thread(){  
    @Override  
    public void run() {  
        System.out.println("Hi! I'm thread!");  
    }  
};  
t.start();
```

Thread + Runnable

```
Thread t = new Thread(new Runnable() {  
    public void run() {  
        System.out.println("Hi! I'm thread!");  
    }  
});  
t.start();
```

Thread.run()

- По умолчанию вызывает метод **run()** переданного **Runnable** (если есть)
- Можно переопределить
- Значимое ограничение - метод не может возвращать значения и не определяет никаких возможных исключений.
- **Thread** реализует **Runnable**
- Не перепутайте методы **start** и **run** у потока. Метод **run** не стартует отдельный поток!

Свойства Thread

- **name** - имя потока.
- **id** - идентификатор потока
- **daemon** - маркер, является ли поток демоном
- **priority** - приоритет потока

Методы Thread

- **start** - стартует новый поток. Если ваш поток уже выполняется, то вызов этого метода спровоцирует **IllegalThreadStateException**
- Методы **sleep** говорят java-машине, что этому потоку нечего выполнять, поэтому пока что (в течении какого-то времени) его не надо ставить в очередь выполнения.
- Методы **join** используются для ожидания завершения потока.
- Статический метод **currentThread()** позволяет определить, какой поток сейчас исполняется.

Прерывание потока

- Гарантированного метода для завершения потока не существует
- С помощью метода **interrupt()** мы посылаем потоку информацию, что мы хотим его прервать.
- Обработка - на совести разработчика потока. Вы можете определить, что ваш поток хотят прервать так:
 - У вас возникло исключение **InterruptedException**
 - Вызов метода **isInterrupted()** возвращает **true**.
 - Вызов метода **interrupted()** возвращает **true**. Осторожно! Этот метод очистит флаг прерывания (можно восстановить снова с помощью метода **interrupt()**)

- 1 Основы многопоточности
- 2 Thread и Runnable
- 3 Проектирование приложений
- 4 Механизм synchronized
- 5 Как синхронизировать доступ к общедоступному состоянию?
- 6 Visibility
- 7 Практики проектирования объектов
- 8 Многопоточность в java

Что такое "потокобезопасный объект"?

- Объект корректно себя ведет себя в однопоточном приложении
- Объект сохраняет корректность поведения в многопоточном приложении

Совместное изменение общих данных

Необходимо гарантировать атомарность действий (там, где они логически атомарны).

- *LostAtomicity.java*
- *GoodAtomicity.java*

Race condition

- Это когда корректность результата зависит от "удачного" времени.
- Очень частое явление для неатомарных действий:
 - read-modify-write
 - check-then-act
- Ленивая инициализация - один из известных примеров **check-then-act**
- *LazyInitialization.java*

- 1 Основы многопоточности
- 2 Thread и Runnable
- 3 Проектирование приложений
- 4 Механизм synchronized
- 5 Как синхронизировать доступ к общедоступному состоянию?
- 6 Visibility
- 7 Практики проектирования объектов
- 8 Многопоточность в java

Встроенный механизм блокировки

synchronized block

```
Object obj = new Object();
```

```
...
```

```
synchronized (obj) {
```

```
    ...
```

```
}
```

Встроенный механизм блокировки

synchronized methods

```
public class MyClass {  
  
    // by current instance  
    public synchronized void doSomething(){  
        ...  
    }  
  
    // by current class  
    public static synchronized void doSomething2(){  
        ...  
    }  
}
```

synchronized - свойства

- reentrancy?
- эффективность?

- 1 Основы многопоточности
- 2 Thread и Runnable
- 3 Проектирование приложений
- 4 Механизм synchronized
- 5 Как синхронизировать доступ к общедоступному состоянию?
- 6 Visibility
- 7 Практики проектирования объектов
- 8 Многопоточность в java

Общедоступные переменные

- Для каждой переменной с многопоточным доступом:
 - Собственный объект для блокировки
 - Доступ только через синхронизированный блок
- То есть:

```
public class MyClass {  
    int counter; Object lock;  
  
    public void incrementAndGet(){  
        synchronized(lock){  
            return counter++;  
        }  
    }  
  
    public void decrementAndGet(){  
        synchronized(lock){  
            return counter--;  
        }  
    }  
}
```

Общедоступные переменные

- А что, если для нескольких не связанных между собой переменных использовать один объект для блокировки?

```
public class MyClass {  
    int counter0, counter1; Object lock;  
  
    public void increment0AndGet(){  
        synchronized(lock){ return counter0++; }  
    }  
    public void decrement0AndGet(){  
        synchronized(lock){ return counter0--; }  
    }  
    public void increment1AndGet(){  
        synchronized(lock){ return counter1++; }  
    }  
    public void decrement1AndGet(){  
        synchronized(lock){ return counter1--; }  
    }  
}
```

Общедоступные переменные

Связанные переменные

Если переменные логически связаны, то для них необходимо использовать один и тот же блокировщик:

- *InvariantAtomicityNonSafe.java*
- *InvariantAtomicityNonSafe.java*

Потокобезопасность общедоступных переменных

- Нет общедоступных переменных
- 1 общедоступная переменная
- 2 и более общедоступных переменных

- 1 Основы многопоточности
- 2 Thread и Runnable
- 3 Проектирование приложений
- 4 Механизм synchronized
- 5 Как синхронизировать доступ к общедоступному состоянию?
- 6 **Visibility**
- 7 Практики проектирования объектов
- 8 Многопоточность в java

Видимость изменений

- Изменения в переменной, сделанные в одном потоке, должны быть видны в другом потоке
- Java может делать различные оптимизации и может возникнуть проблема перестановки (reordering)

Видимость изменений

- Stale data
- **out-ofthin-air safety** - мы видим данные, может и устаревшие, но хотя бы актуальные на какой-то момент в прошлом.
- 64-битные операции (long, double) - мы можем увидеть данные, которые даже не существовали никогда.
- **intrinsic lock** - если два потока входят в блоки, охраняемые одним и тем же блокировщиком, то второй *гарантированно* увидит изменения, сделанные первым.
- **volatile** - слабая форма синхронизации, компилятору дают указание, что переменная будет использоваться из многих потоков, и потому ее изменения должны быть видны сразу.
- **final** - гарантируется, что после создания объекта с final-полями значения, находящиеся в этих полях, всем видны.

- 1 Основы многопоточности
- 2 Thread и Runnable
- 3 Проектирование приложений
- 4 Механизм synchronized
- 5 Как синхронизировать доступ к общедоступному состоянию?
- 6 Visibility
- 7 **Практики проектирования объектов**
- 8 Многопоточность в java

Перед тем, как проектировать объект

- 1 определить переменные, которые представляют собой состояние объекта
- 2 определить инварианты, пост-условия и ограничения на переменные
- 3 определить политику для управления доступом к объекту

Instance confinement

- инкапсуляция упрощает процесс разработки.
- при правильно реализованной инкапсуляции, легче реализовать политику безопасности

```
public class MySet {  
    private final Set<MyClass> mySet = new  
        HashSet<MyClass>();  
    public synchronized void add(MyClass c) {  
        mySet.add(c);  
    }  
    public synchronized boolean contains(MyClass c) {  
        return mySet.contains(c);  
    }  
}
```

Java Monitor Pattern

- Используется во многих объектах.

```
public class MyPrivateLock {  
    private final Object myLock = new Object();  
    void someMethod() {  
        synchronized(myLock) {  
            // ...  
        }  
    }  
}
```

- Есть минусы (например, трудно расширять объект)

Композиция потокобезопасных объектов?

- Не всегда композиция потокобезопасна (зависит от действий над ними): см. *InvariantAtomicityNonSafe.java*, *IntRange.java*
- Однако, если внутреннее поле-состояние не участвует в инвариантах и не имеет "неправильных" состояний, то его можно даже публиковать.

Расширение объектов?

- Например, мы хотим добавить функциональность к **Vector**:

```
public class Vector1<E> extends Vector<E> {  
    public synchronized boolean putIfAbsent(E x) {  
        boolean absent = !contains(x);  
        if (absent)  
            add(x);  
        return absent;  
    }  
}
```

- Но что получится, если тот, кто реализовал класс, использовал **java monitor pattern**? То же самое не сработает с синхронизированной версией **ArrayList**!
- В таком случае может быть проще использовать **делегирование** вместо расширения.

Расширение объектов?

```
public class ImprovedList<T> implements List<T> {  
    private final List<T> list;  
    public ImprovedList(List<T> list) { this.list = list; }  
    public synchronized boolean putIfAbsent(T x) {  
        boolean contains = list.contains(x);  
        if (contains)  
            list.add(x);  
        return !contains;  
    }  
    public synchronized void clear() { list.clear(); }  
    // ... similarly delegate other List methods  
}
```

Документирование

Рекомендуется документировать все, что связано с thread-safety policy.

- 1 Основы многопоточности
- 2 Thread и Runnable
- 3 Проектирование приложений
- 4 Механизм synchronized
- 5 Как синхронизировать доступ к общедоступному состоянию?
- 6 Visibility
- 7 Практики проектирования объектов
- 8 Многопоточность в java

Коллекции

- Synchronized коллекции: Vector, Stack, Hashtable
 - проблемы с составными действиями
 - проблемы с производительностью
 - проблемы с итераторами (в т.ч. и скрытыми)
- Concurrent коллекции: ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue

Синхронизаторы

- **Latch** - откладывает работу потока до какого-то время (например, на основе счетчика). Работает как ворота. Ждет **события**. См. *CountDownLatchDemo.java*.
- **Barrier** - похож на **latch**, однако ждет достижения определенной точки всеми потоками. Ждет **остальные потоки**.
- **Semaphore** - управляет доступом к определенным ресурсам. Например, можно с помощью него реализовать pool ресурсов.
- **FutureTask** - блокирующая сущность, позволяет ожидать какого-то действия. См. *FutureTaskDemo.java*

Как реализовать множество задач?

- Последовательно? Медленно.
- Самим создавать потоки? Тяжело управлять потоками, а так же следить за тем, сколько потоков у вас уже есть.
- Рекомендуется иметь не более **nCPU + 1** потоков.

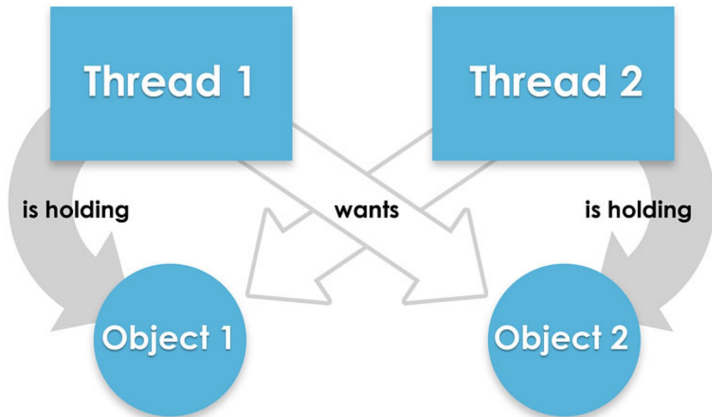
```
int cores = Runtime.getRuntime().availableProcessors();
```

Executor framework

- Достаточно мощный фреймворк для управления потоками.
- Предоставляет возможность создавать thread pools, управлять ими, реализует полноценный жизненный цикл.

- 1 Основы многопоточности
- 2 Thread и Runnable
- 3 Проектирование приложений
- 4 Механизм synchronized
- 5 Как синхронизировать доступ к общедоступному состоянию?
- 6 Visibility
- 7 Практики проектирования объектов
- 8 Многопоточность в java

Deadlock

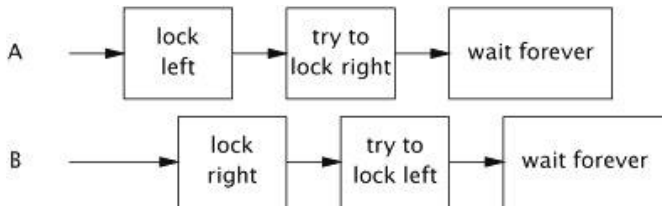


Lock-Ordering Deadlock

```
public class LeftRightDeadlock {  
    private final Object left = new Object();  
    private final Object right = new Object();  
    public void leftRight() {  
        synchronized (left) {  
            synchronized (right) {  
                doSomething();  
            }  
        }  
    }  
    public void rightLeft() {  
        synchronized (right) {  
            synchronized (left) {  
                doSomethingElse();  
            }  
        }  
    }  
}
```


Lock-Ordering Deadlock

Чтобы избежать, нужен фиксированный глобальный порядок локов



Starvation

Running Java Thread



Starving Thread



Higher Priority Threads waiting...

Livelock

Thread LiveLock

