Java - multithreading

Anastasiya Solodkaya, Denis Stepulenok

LevelUP

31 января 2017 г.

Многопоточность

- Совместно изменяемое состояние объекта
- Потокобезопасность объекта
  - Объект корректно себя ведет себя в однопоточном приложении
  - Объект сохраняет корректность поведения в многопоточном приложении
- Лучше вкладывать в дизан как можно раньше

Атомарность действий

- Действие неделимо
- Что будет, если это не так?
  - *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*

Встроенный механизм блокировки

synchronyzed block

```
Object obj = new Object();

...

synchronyzed (obj) {
...
}
```

Встроенный механизм блокировки

synchronyzed methods

```
public class MyClass {

    // by current instance
    public synchronized void doSomething(){
        ...
    }

    // by current class
    public static synchronized void doSomething2(){
        ...
    }

}
```

synchronized - свойства

- reentrancy?
- эффективность?

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

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)
- **2 и более** общедоступных переменных

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.
- **volitile** - слабая форма синхронизации, компилятору дают указание, что переменная будет использоваться из многих потоков, и потому ее изменения должны быть видны сразу.

Не дай убежать ссылкам!

- Необходимо всегда внимательно следить за тем, что мы открываем для других потоков.
- Не получил ли другой поток нелегальный доступ к изменению внутреннего состояния?
- Не получил ли другой поток доступ к **this** из конструктора (когда объект еще не до конца сформирован)?
- Не получил ли другой поток нелегальный достуа к **this** через вложенные классы?

Доступность - недоступные для других потоков объекты

- **thread-confinement** - мы не отдаем объект вовне, у других потоков просто нет на него ссылки.
- **ad-hoc thread-confinement** - мы можем отдать объект вовне, но документируем, что его не надо брать извне.
- **stack confinement** - хранение в локальных переменных. Мы можем перекопировать разделяемый объект в локальную переменную.
- **ThreadLocal<T>** - класс, который позволяет хранить какие-то значения для каждого текущего потока отдельно. Использовать с осторожностью!

Доступность - недоступные для модификации из других потоков объекты

- **immutable** объекты
- **final** - гарантированная initialization safety
- оставлять поля не **final** только в том случае, если они действительно изменяемые.
- есть так же логически неизменяемые объекты (по сути проектировки)

Доступность - неправильная публикация объекта

- Даже хороший объект может сломаться, если будет неправильно опубликован для других потоков.

```
public class Publisher {
    public Holder instance;
    public void initialize() { instance = new
        Holder(10); }
}

public class Holder {
    private int n;
    public Holder(int n) { this.n = n; }
    public void assertSanity() {
        if (n != n)
            throw new AssertionError("This statement is
                false.");
        }
    }
}
```

Доступность - неправильная публикация объекта

- Мы можем увидеть устаревшее значение по ссылке
- Мы можем увидеть актуальное значение по ссылке, но устаревшее состояние.

Доступность - безопасная публикация объекта

И ссылка на объект, и сам объект должны быть опубликованы одновременно!

- Инициализация из статического инициализатора
- Поле volitile
- Поле AtomicReference
- Поле final
- Поле, защищенное блокировкой

Общие правила публикации объекта

- **immutable** объекты могут быть опубликованы с помощью любого механизма.
- **effectively immutable** объекты должны быть опубликованы безопасно
- **immutable** объекты должны быть опубликованы безопасно, а так же должны быть защищены с помощью lock или же должны иметь такую защиту.

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.

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

ConcurrentHashMap

- Weakly-consistent iterator
- Методы size() и isEmpty() возвращают **примерный** результат.
- Lock stripping:
  - Может иметь много читателей.
  - Может иметь ограниченное количество писателей.
  - Безопасная совместная работа читателей и писателей.
- Разные нужные составные действия типа **put-if-absent** уже реализованы.
- Невозможно использовать client-side locking (не блокируется эксклюзивно)

CopyOnWriteArrayList

- Безопасное итерирование по элементам, которые не видят изменений, сделанных из других потоков.
- Используется внутренний блокиратор
- При записи создается копия массива и по окончанию записи записывается

Queues

- BlockingQueue - блокирующая
- ConcurrentLinkedQueue - FIFO
- PriorityBlockingQueue
- SynchronousQueue - вообще по сути не та очередь, это для управления очередями потоков

BlockingQueue

- Блокирующая - если у нас есть ограничения на размер, то в случае пустой коллекции операция **poll** блокируется до поступления нового объекта, а для полной блокируется **put**.
- Паттерн **producer-consumer**
- Отличный инструмент для управления ресурсами
- **BlockingDeque** - реализует паттерн **work-stealing**.

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.

Executor framework: Executor interface

```
public interface Executor {
    void execute(Runnable command);
}
```

Executor framework: Executors(Thread Pools)

Класс **Executors** содержит набор фабричных методов для создания различных пулов потоков

- **newFixedThreadPool** - фиксированный размер. Если поток умирает, то он добавляет новый. Пытается соответствовать заявленному размеру
- **newCachedThreadPool** - не ограничивает размер, более гибкий
- **newSingleThreadExecutor** - ограничивает размер одним потоком, это гарантировано и не может быть никем изменено.
- **newScheduledThreadPool** - фиксированный размер, позволяет откладывать исполнение потока или повторять.

Executor framework: Executors(Thread Pools)

На самом деле все эти методы возвращают **ExecutorService**, который имеет расширенную функциональность и позволяет управлять жизненным циклом пула потоков.

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException;
    // ... additional convenience methods for task
        submission
}
```

Executor framework: Executors(Thread Pools)

- **shutdown()** - постепенный останов, задачи завершаются естественным образом, новые не завершаются
- **shutdownNow()** - немедленный останов, все задачи отменяются (вернее, мы **пытаемся** отменить все задачи)

Callable и Future

- **Runnable** - базовый интерфейс, который принимается Executor. Не возвращает ничего (**run()**), не бросает исключений.
- **Callable** (или **Callable<Void>**) - больше подходит для вычислительных задач и задач, которые могут выбросить исключение.
- **Future** - предоставляет уже больше возможностей для управления. Здесь можно проверить, завершена ли задача, отменить ее, и так далее.

Callable и Future

```
public interface Runnable {
    public abstract void run();
}
public interface Callable<V> {
    V call() throws Exception;
}
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException,
        ExecutionException, CancellationException;
    V get(long timeout, TimeUnit unit) throws
        InterruptedException, ExecutionException,
        CancellationException, TimeoutException;
}
```

CompletionService: Executor + BlockingQueue

- Позволяет задавать множество задач, и получать их результат по мере завершения их работы.

CompletionService: Executor + BlockingQueue

```
public class Renderer {
    private final ExecutorService executor;
    Renderer(ExecutorService executor) { this.executor =
        executor; }
    void renderPage(CharSequence source) {
        final List<ImageInfo> info = scan(source);
        CompletionService<ImageData> c =
            new ExecutorCompletionService<>(executor);
        for (final ImageInfo imageInfo : info)
            c.submit(() -> imageInfo.downloadImage());
        renderText(source);
        try {
            for (int t = 0, n = info.size(); t < n; t++) {
                Future<ImageData> f = c.take();
                ImageData imageData = f.get();
                renderImage(imageData);
            }
        } catch (InterruptedException | ExecutionException
            e) { ... }
    }
```

Завершение задач и потоков

- Нет такого механизма, который позволил бы немедленно завершить задачу или поток
- Отмена задачи может быть реализована с помощью простого поля **cancel** (volitile):

```
public class MyTask implements Runnable {
    private volatile boolean cancelled;
    public void run() {
        while (!cancelled ) { ... }
    }
    public void cancel() {
        cancelled = true;
    }
    ...
}
```

Прерывание потока

- Для потока можно вызвать прерывание
- Реализация действий по получения прерывания - на совести того, кто пишет код

```
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    public static boolean interrupted() { ... }
    ...
}
```

- Метод **interrupted()** должен использоваться аккуратно, так как сбрасывает флаг "прервано"

Прерывание потока - соглашения

- Всегда, когда вы используете инструменты для работы с потоками, необходимо учитывать **interruprion policy**, так как в одном пуле прерывание потока будет означать отмену задачи, а в другом - полностью завершить работу пула.
- Проектируя свои задачи и потоки и задачи, не стоит полагаться на знания **interruption policy** классов, с которыми они работают.

Стратегия работы с прерыванием (1)

Перепрокинуть исключение

```
BlockingQueue<Task> queue;
...
public Task getNextTask() throws InterruptedException {
    return queue.take();
}
```

Стратегия работы с прерыванием (2)

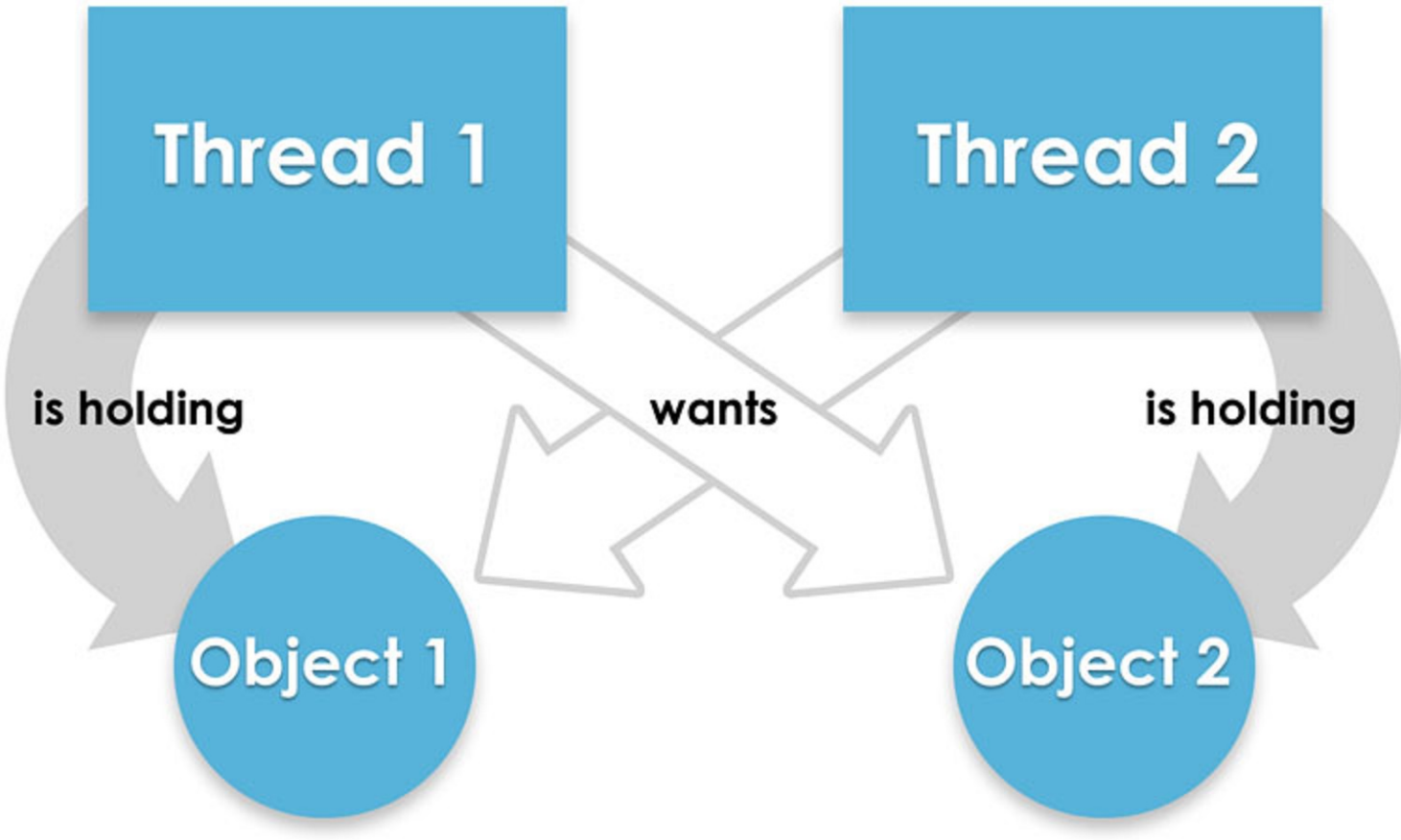
Восстановить статус "прервано"

```
// actually non-cancelable task
public Task getNextTask(BlockingQueue<Taskgt; queue) {
    boolean interrupted = false;
    try {
        while (true) {
            try {
                return queue.take();
            } catch (InterruptedException e) {
                interrupted = true;
                // fall through and retry
            }
        }
    } finally {
        if (interrupted)
            Thread.currentThread().interrupt();
    }
}
```

Стратегия работы с прерыванием (3)

Самое ужасное, что можно сделать - это просто "погасить"**InterruptedException** и забыть о нем.

Deadlock

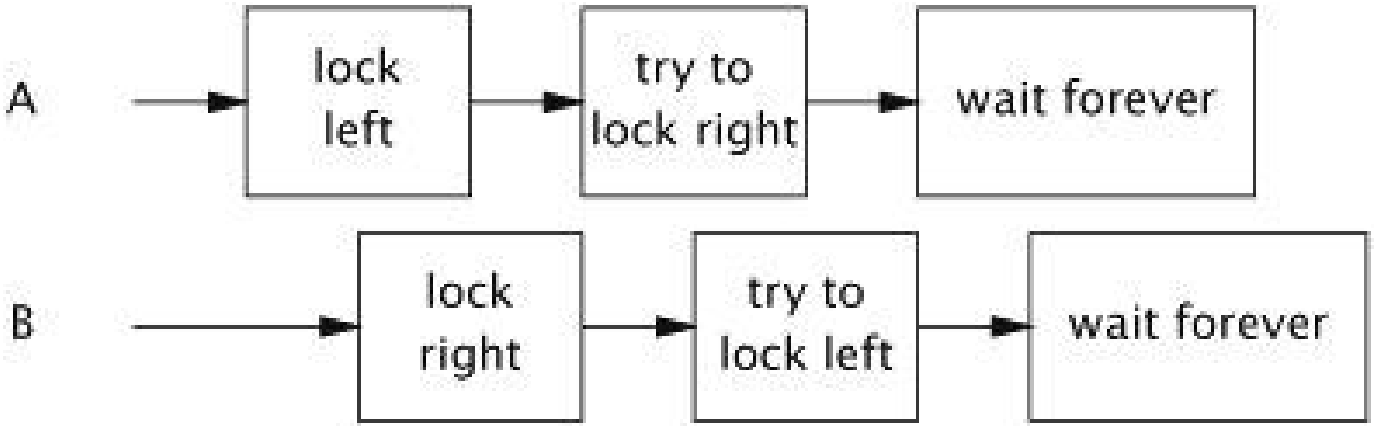

Thread 1
Thread 2
is holding
wants
is holding
Object 1
Object 2

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



Dynamic Lock-Ordering Deadlock

Порядок взятия блокировки на самом деле зависит от порядка переданных аргументов.

```
public void transferMoney(Account fromAccount, Account
    toAccount, DollarAmount amount) throws
    InsufficientFundsException {
     synchronized (fromAccount) {
         synchronized (toAccount) {
             if (fromAccount.getBalance().compareTo(amount)
                 < 0)
                 throw new InsufficientFundsException();
             else {
                 fromAccount.debit(amount);
                 toAccount.credit(amount);
             }
         }
     }
}
```

Cooperating Objects Deadlock

```
class Taxi {
    private Point location, destination;
    private final Dispatcher dispatcher;
    public Taxi(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }
    public synchronized Point getLocation() {
        return location;
    }
    public synchronized void setLocation(Point location) {
        this.location = location;
        if (location.equals(destination))
            dispatcher.notifyAvailable(this);
    }
}
```

Cooperating Objects Deadlock

```
class Dispatcher {
    private final Set<Taxi> taxis;
    private final Set<Taxi> availableTaxis;
    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }
    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }
    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

Как работать с deadlock'ами

- Использовать time-based блокировки
- Анализировать thread dumps

Deadlock: thread dump

```
Found one Java-level deadlock:
=============================
"ApplicationServerThread":
  waiting to lock monitor 0x080f0cdc (a MumbleDBConnection),
  which is held by "ApplicationServerThread"
"ApplicationServerThread":
  waiting to lock monitor 0x080f0ed4 (a
      MumbleDBCallableStatement),
  which is held by "ApplicationServerThread"

Java stack information for the threads listed above:
"ApplicationServerThread":
        at MumbleDBConnection.remove_statement
        - waiting to lock <0x650f7f30> (a
            MumbleDBConnection)
        at MumbleDBStatement.close
        - locked <0x6024ffb0> (a MumbleDBCallableStatement)
      ...
      ...
```

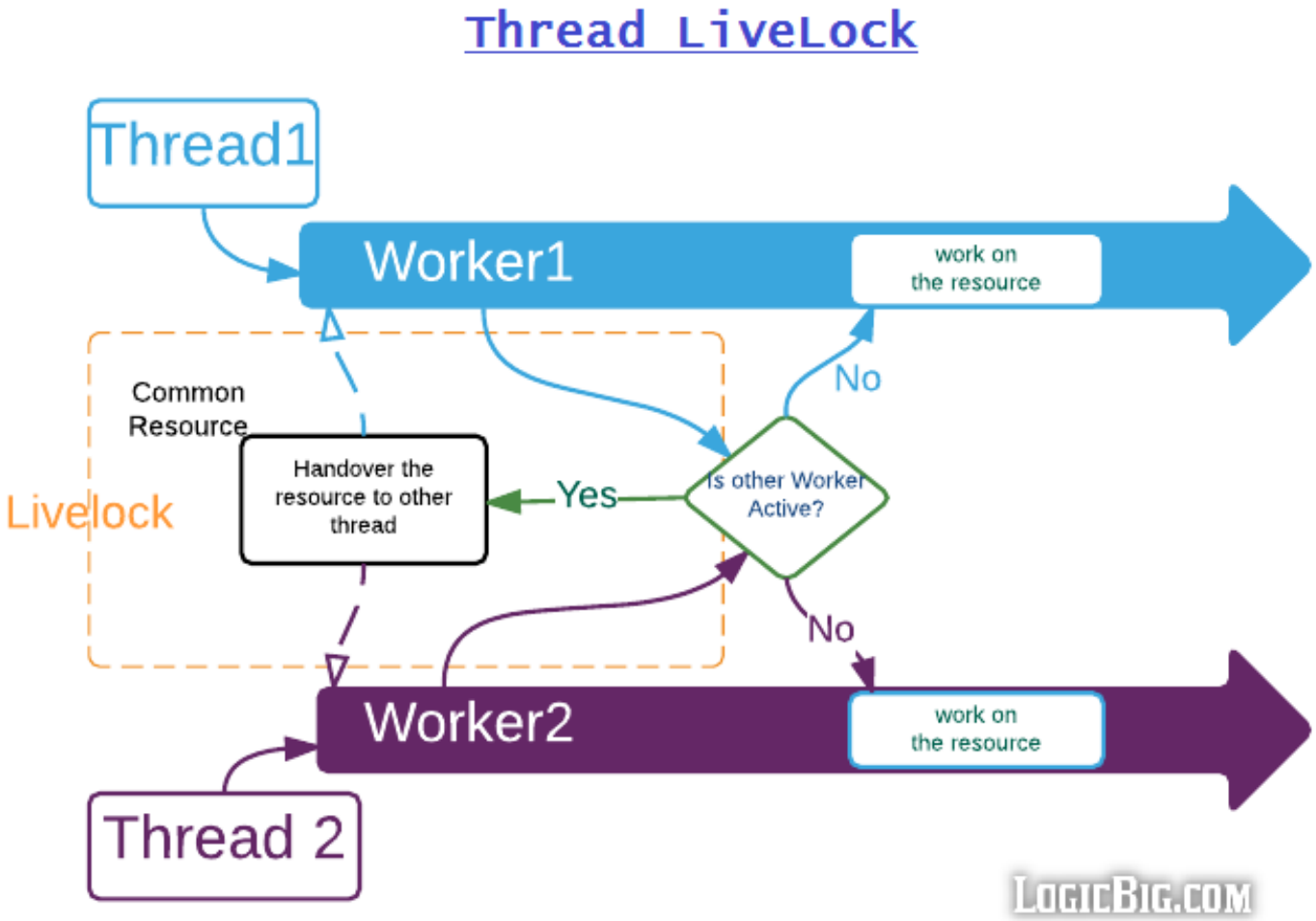
Deadlock: thread dump

```
"ApplicationServerThread":
        at MumbleDBCallableStatement.sendBatch
        - waiting to lock <0x6024ffb0> (a
            MumbleDBCallableStatement)
        at MumbleDBConnection.commit
        - locked <0x650f7f30> (a MumbleDBConnection)
      ...
```

Starvation


Running Java Thread
Starving Thread
Higher Priority Threads waiting...

Livelock


Thread LiveLock
Thread1
Worker1
work on the resource
Common Resource
Livelock
Handover the resource to other thread
Yes
Is other Worker Active?
No
No
Worker2
work on the resource
Thread 2
LogicBig.com

Шаблоны

- **Readers-writer lock** многопоточный доступ на чтение, однопоточный на запись
- **Thread-local-storage** - в джаве реализован в **ThreadLocal<T>**
- **Thread pool**

Singleton и Double-locking pattern

Однопоточная версия singleton'а

```
class Foo {
    private Foo instance;
    public Foo getInstance() {
        if (instance == null) {
            instance = new Foo();
        }
        return instance;
    }

    ...
}
```

Singleton и Double-locking pattern

Многопоточная, медленная версия singleton'а

```
class Foo {
    private Foo instance;
    public synchronized Foo getInstance() {
        if (instance == null) {
            instance = new Foo();
        }
        return instance;
    }

    ...
}
```

Singleton и Double-locking pattern

Двойная блокировка, антипаттерн (по версии Oracle). Может быть источником проблем (stale data)

```
class Foo {
    private Foo instance;
    public Foo getInstance() {
        if (instance == null) {
            synchronized(this) {
                if (instance == null) {
                    instance = new Foo();
                }
            }
        }
        return instance;
    }
    ...
}
```

Singleton и Double-locking pattern

Двойная блокировка, антипаттерн (по версии Oracle). Работает для JDK 1.5 и выше.

```
class Foo {
    private volatile Foo instance;
    public Foo getInstance() {
        if (instance == null) {
            synchronized(this) {
                if (instance == null) {
                    instance = new Foo();
                }
            }
        }
        return instance;
    }
    ...
}
```

Singleton и Double-locking pattern

Самый удачный пример реализации

```
class Bar {
    private static class FooHolder {
        public static final Foo instance = new Foo();
    }

    public static Foo getInstance() {
        return FooHolder.instance;
    }
}
```