Приёмы параллельного программирования в Java

Виды параллельных вычислений

Распараллеливание вычислений может быть на уровне:

- систем (распределенные системы, облачные вычисления, grid, cluster, etc.)
- процессов (процессы ОС)
- потоков

Взаимодействие потоков выполнения может быть через:

- общую память
- общее хранилище данных (базу данных)

Мотивация

- Увеличение производительности за счет равномерного использования нескольких процессоров или разнородных систем:
  - Обработка больших объемов данных
    (Построение отчетов, бизнес аналитика, индексирование больших хранилищ)
  - Долгие вычисления
    (Шифрование, переборные алгоритмы (расчет маршрутов, минимизация издержек, решение математических задач, игры и т.д.)

- Масштабируемость системы (если эффективно распараллеливается – можно задействовать больше машин)

Мотивация

Может исходить из функциональных требований к системе, например:

- Система должна быть многопользовательской
- Многозадачность
- Выполнение каких-либо действий в фоновом режиме или асинхронно
- Выполнение каких-либо действий по расписанию
- Отзывчивый пользовательский интерфейс

Подходы к распараллеливанию вычислений

- Разделяемый доступ к общей памяти
  - блокировки, синхронизация критических секций, барьеры памяти, атомарные инструкции типа CAS, базы данных, транзакции
- Обмен сообщениями
  - Erlang (модель акторов), Occam (CSP) – потоки не имеют доступа к общей памяти, но могут обмениться сообщениями синхронно или асинхронно
  - Можно реализовать и в Java с использованием Immutable типов, глубокого копирования данных и обмена сообщениями (например JMS)
- Полная изолированность потоков вычислений (Одни потоки могут порождать другие при этом передавая на вход данные. Поток обрабатывает исходные данные и возвращает результат породившему потоку)
  - MapReduce (AppEngine Mapper API, Apache Hadoop),
  - J2EE WorkManager

Возникающие проблемы

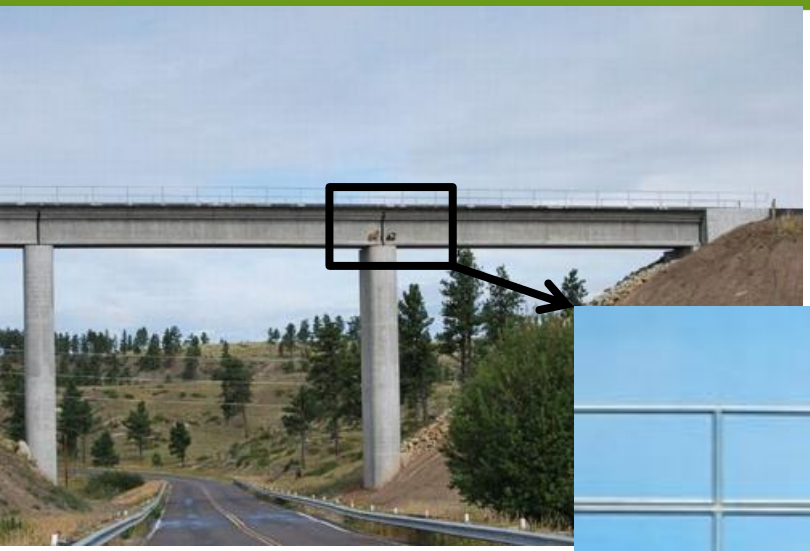
- Разделяемый доступ к общей памяти
  - Сохранение целостности данных при одновременном доступе (thread-safety),
    - непредсказуемость фактического порядка исполнения инструкций (race conditions)
  - Конечность вычислений
    - взаимная блокировка потоков (deadlock)
    - зависание потока (starvation)
    - тупиковая ситуация (livelock)
    - инверсия приоритетов (priority inversion)
  - Низкая производительность
    - ожидание очереди на блокировку
    - переключение контекста (context switching)

Плюсы и минусы подходов

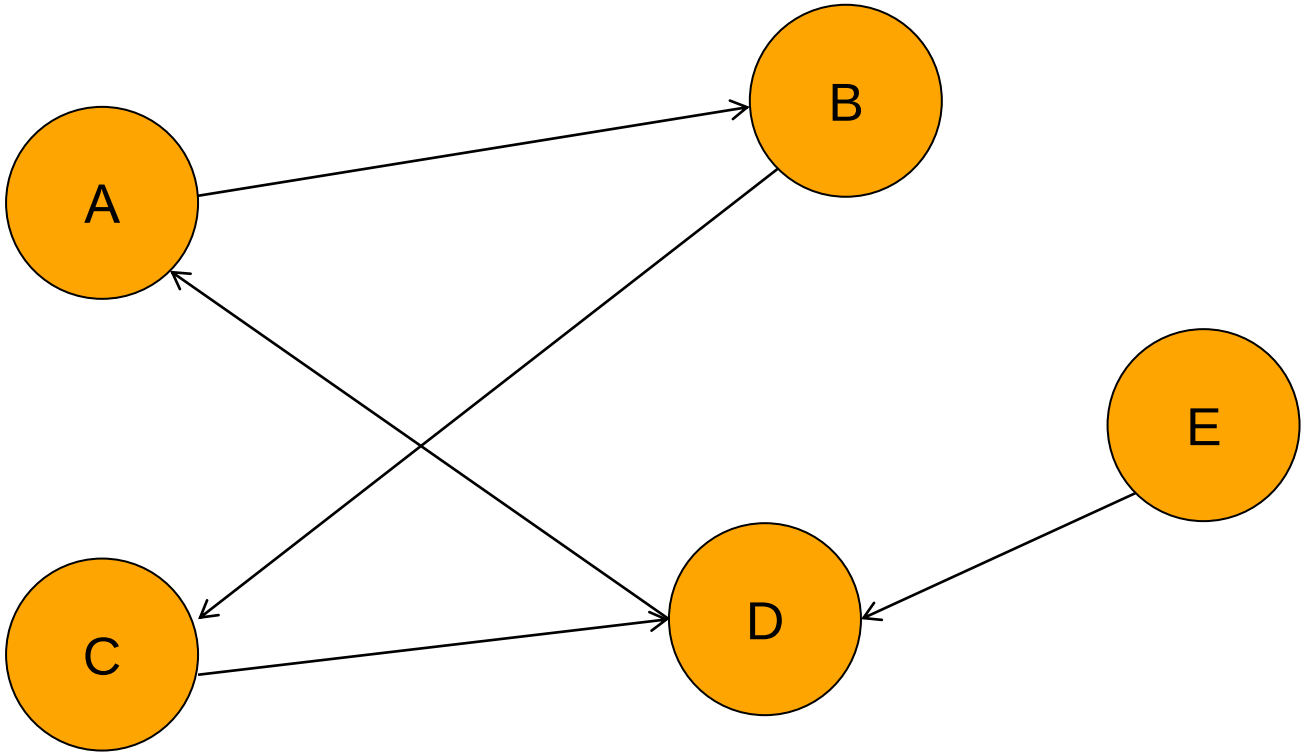
- Обмен сообщениями
  - код всегда потокобезопасен
  - легче масштабировать
  - дополнительные затраты на коммуникации
  - сложнее строить алгоритмы

- Изолированные потоки
  - сложность алгоритмического распараллеливания обработки данных
  - легко масштабируется
  - нет всех вышеперечисленных проблем

Взаимная блокировка



Граф блокировок потоков



Взаимная блокировка.

```
final Object lockA = new Object();
final Object lockB = new Object();
new Thread() {
        public void run() {
                synchronized (lockA) {
                        synchronized (lockB) {
                                // Do something
                        }
                }
        }
}.start();
new Thread() {
        public void run() {
                synchronized (lockB) {
                        synchronized (lockA) {
                                // Do something
                        }
                }
        }
}.start();
```

Дамп потоков

Full thread dump Java HotSpot(TM) 64-Bit Server VM (20.1-b02 mixed mode):

"Thread-1" prio=6 tid=0x0000000006998800 nid=0x1a48 waiting for monitor entry [0x00000000076af000]
  java.lang.Thread.State: BLOCKED (on object monitor)
        at DeadlockDemo$2.run(DeadlockDemo.java:27)
        - waiting to lock <0x00000007d5d0a6e8> (a java.lang.Object)
        - locked <0x00000007d5d0a6f8> (a java.lang.Object)

"Thread-0" prio=6 tid=0x0000000006998000 nid=0x1620 waiting for monitor entry [0x000000000748f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
        at DeadlockDemo$1.run(DeadlockDemo.java:16)
        - waiting to lock <0x00000007d5d0a6f8> (a java.lang.Object)
        - locked <0x00000007d5d0a6e8> (a java.lang.Object)

…

Вывод информации о блокировках

```
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x000000000058ca80 (object 0x00000007d5d0a6e8, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x000000000058f330 (object 0x00000007d5d0a6f8, a java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
                at DeadlockDemo$2.run(DeadlockDemo.java:27)
                - waiting to lock <0x00000007d5d0a6e8> (a java.lang.Object)
                - locked <0x00000007d5d0a6f8> (a java.lang.Object)
"Thread-0":
                at DeadlockDemo$1.run(DeadlockDemo.java:16)
                - waiting to lock <0x00000007d5d0a6f8> (a java.lang.Object)
                - locked <0x00000007d5d0a6e8> (a java.lang.Object)

Found 1 deadlock.
```

Взаимная блокировка взаимодействующих объектов

```
class A {
            public synchronized void callB(B b){
                        b.process();
            }
            public synchronized void process(){}
}
class B {
            public synchronized void callA(A a){
                        a.process();
            }
            public synchronized void process(){}
}

final A a = new A();
final B b = new B();

//Thread1
a.callB(b);

//Thread 2
b.callA(a);
```

Дамп потоков

```
"Thread-1" prio=6 tid=0x0000000006ab1000 nid=0x18a8 waiting for monitor entry [0x00000000076ef000]
   java.lang.Thread.State: BLOCKED (on object monitor)
                at DeadlockDemo1$A.process(DeadlockDemo1.java:16)
                - waiting to lock <0x00000007d5d0bac0> (a DeadlockDemo1$A)
                at DeadlockDemo1$B.callA(DeadlockDemo1.java:26)
                - locked <0x00000007d5d0cc30> (a DeadlockDemo1$B)
                at DeadlockDemo1$2.run(DeadlockDemo1.java:45)

"Thread-0" prio=6 tid=0x0000000006ab0000 nid=0x17e0 waiting for monitor entry [0x00000000075ef000]
   java.lang.Thread.State: BLOCKED (on object monitor)
                at DeadlockDemo1$B.process(DeadlockDemo1.java:31)
                - waiting to lock <0x00000007d5d0cc30> (a DeadlockDemo1$B)
                at DeadlockDemo1$A.callB(DeadlockDemo1.java:11)
                - locked <0x00000007d5d0bac0> (a DeadlockDemo1$A)
                at DeadlockDemo1$1.run(DeadlockDemo1.java:39)
```

Случай с аппаратом лучевой терапии Therac-25

Этот аппарат работал в трёх режимах:

- Электронная терапия: электронная пушка напрямую облучает пациента; компьютер задаёт энергию электронов от 5 до 25 МэВ.
- Рентгеновская терапия: электронная пушка облучает вольфрамовую мишень, и пациент облучается рентгеновскими лучами, проходящими через конусообразный рассеиватель. В этом режиме энергия электронов одна: 25 МэВ.
- В третьем режиме никакого излучения не было. На пути электронов (на случай аварии) располагается стальной отражатель, а излучение имитируется светом. Этот режим применяется для того, чтобы точно навести пучок на больное место.

Модель памяти Java. Зачем это нужно знать?

```
public class NoVisibility {
        private static boolean ready;
        private static int number;

        private static class ReaderThread extends Thread {
                public void run() {
                        while (!ready)
                                Thread.yield();
                        System.out.println(number);
                }
        }

        public static void main(String[] args) {
                new ReaderThread().start();
                number = 42;
                ready = true;
        }
}
```

Модель памяти Java

Что это?

Это *часть спецификации языка Java* (JLS chapter 17), описывающая на низком уровне принципы хранения переменных в памяти (полей объекта, класса, элементов массива и т.д.) и доступа потоков к общей памяти.

Описывает поведение многопоточных программ в терминах:

- Атомарности операций
- Видимости данных
- Порядка выполнения инструкций

Модель памяти Java. Атомарность операций

Какие операции являются атомарными?

Атомарные операции – эффект которых на систему семантически такой же как если бы они выполнялись мгновенно и результат не зависел от работы других потоков.

Согласно модели памяти Java:

- Чтение и запись значений в память соответствующую переменной (кроме типов long и double) является атомарной операцией
- Чтение и запись указателей атомарно вне зависимости от разрядности указателя (32 или 64 бита)
- Чтение и запись volatile long или double переменных атомарно

Модель памяти Java. Видимость данных

Видимость данных - это правила, описывающие при каких обстоятельствах изменение переменных в одном потоке становится видимым для других потоков.

Изменение данных одним поток становится видим для других, если:

- Пишущий поток освобождает блокировку, которую тут же получает другой поток
- Для **volatile** поля все изменения сделанные в одном потоке сразу же видны другим потокам.
- При первом обращении к полю объекта поток увидит либо его значение по умолчанию, либо значение записанное каким-либо потоком.

Вторая семантика synchronized

```
//Thread 1
x = 1;
synchronized(this){
  y = 2;
}
```

Все переменные записаны в память

Все переменные читаются из памяти

```
//Thread 2
synchronized(this){
  a = x;
}
b = y;
```

Модель памяти Java. Видимость данных

- После окончания работы конструктора все **final** поля становятся видимыми другим потокам.
- Все модификации сделанные в конструкторе видны в **finalize()**
- При запуске **нового потока**, все модификации текущего потока становятся видны
- При **завершении потока**, все модификации переменных становятся видимыми другим потокам

Модель памяти Java.

Изменение порядка выполнения инструкций

- Компилятором (может менять инструкции местами для оптимизации, если это не нарушает логики выполнения программы*)
- Процессором
- Процессором может быть изменен порядок записи переменных из кэша в оперативную память (может производится в порядке отличном от того как они изменялись программой)

Основной принцип:

Программа должна сохранять семантику последовательного и однопоточного выполнения.

Модель памяти Java. Работа над ошибками.

До Java 5:

- Не гарантированности видимости final полей класса после завершения конструктора.
  - Immutable объекты не были потокобезопасными

    Можно было получить ошибку в подобном коде:

```
String s1 = "string";
String s2 =  s1.substring(2);
if (s2 != s2) throw new AssertionError();
```

Модель памяти Java. Работа над ошибками.

Чтение и запись **volatile** переменных были упорядочены между собой, но не упорядочены относительно других операций.

Следующий код не работал:

```
Map configOptions;
char[] configText;
volatile boolean initialized = false;
. . .
// In thread A
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;
. . .
// In thread B
while (!initialized)
  sleep();
// use configOptions
```

Volatile переменные

```
volatile int value;
```

Семантически эквивалентно

```
private int value;
final synchronized void set(int value) {
       this.value = value;
}
final synchronized float get() {
       return value;
}
```

Volatile переменные

Использовать **volatile** можно, если:

- Доступ к полю происходит из разных потоков, но вне других блоков синхронизации (необходимых для других нужд)
- Поле не входит ни в какие инварианты для состояния объекта
- Модификация поля не зависит от его текущего значения
- В поле могут быть записаны только семантически корректные значения

Например, один поток может модифицировать поле, а другие только читать.

Volatile массивы

Рассмотрим массив:

```
volatile int [] arr = new int[SIZE];


arr = arr;
int x = arr[0];
arr[0] = 1;
```

Сама ссылка на массив – volatile, но элементы массива - нет. Это же применимо и к коллекциям ArrayList, LinkedList и т.д.

Есть атомарные реализации массивов с volatile элементами:

AtomicIntegerArray, AtomicLongArray и AtomicReferenceArray<T>

Безопасная публикация объектов

До завершения конструктора объект может быть не целостным (не все поля инициализированы, не все инварианты состояния выполняются)

Источник неприятностей

- ссылка на объект доступна другому потоку до создания объекта и нет никаких специальных синхронизаций

Пример небезобасной инициализации

```
public Holder holder;
public void initialize() {
        holder = new Holder(42);
}
...
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.");
        }
}
```

```
//Thread1
initialize();

//Thread2
if (holder != null){
        holder.assertSanity();
}
```

Антипаттерн – публикация this в конструкторе

```
public class ThisEscape {
...
        public ThisEscape(EventSource source) {
                source.registerListener(new EventListener() {
                        public void onEvent(Event e) {
                                doSomething(e);
                        }
                });
                ...
        }
}
```

Решение – использовать фабричный метод

```
public class SafeListener {
        private final EventListener listener;
        private SafeListener() {
                listener = new EventListener() {
                        public void onEvent(Event e) {
                                doSomething(e);
                        }
                };
        }
        public static SafeListener newInstance(EventSource source) {
                SafeListener safe = new SafeListener();
                source.registerListener(safe.listener);
                return safe;
        }
}
```

Принципы потокобезопасной инициализации

- Инициализировать ссылку на объект в статической инициализации
- Делать объект доступным другим потокам через ссылку, которая:
  - volatile
  - AtomicReference
  - final поле потокобезопасно инициализированного объекта
  - поле, доступ к которому синхронизирован

Неизменяемые (immutable) объекты

- Объекты, состояние которых не изменяется в течение жизни объекта

Например: String, Integer, Double и т.д.

Объект эффективно считается неизменяемым, если:

- его состояние не может быть изменено после создания
- все поля определяющие состояние – **final***
- объект инициализирован потокобезопасно (ссылка на него недоступна другим потокам до завершения конструктора)

Immutable объекты всегда потокобезопасны.

Пример immutable класса

```
public class ImmutablePoint {
        private final double x;
        private final double y;

        public ImmutablePoint(double x, double y) {
                this.x = x;
                this.y = y;
        }
}
```

Реализация потокобезопасной инициализации синглтона

Задача: есть синглтон. Надо реализовать его отложенную потокобезопасную инициализацию.

```
public class Singleton {
        private static Singleton instance;

        public static synchronized Singleton getInstance() {
                if (instance == null) {
                        instance = new Singleton();
                }
                return instance;
        }

        private Singleton(){
                ...
        }
}
```

Антипаттерн – Блокировка с двойной проверкой (DCL)

```
public static Singleton getInstance() {
        if (instance == null) {
                synchronized (Singleton.class) {                    // 1
                        if (instance == null) {                     // 2
                                instance = new Singleton();         // 3
                        }
                }
        }
        return instance;
}
```

Вариант исправления

```
public static Singleton getInstance() {
        if (instance == null) {
                synchronized (Singleton.class) { // 1
                        Singleton inst = instance; // 2
                        if (inst == null) {
                                synchronized (Singleton.class) { // 3
                                        inst = new Singleton(); // 4
                                }
                                instance = inst; // 5
                        }
                }
        }
        return instance;
}
```

Потокобезопасное решение начиная с Java 5

Так как поменялась семантика **volatile** переменных, можно определить **instance** так:

```
private static volatile Singleton instance;
```

тогда в строке

```
instance = new Singleton();
```

не может произойти перестановки операций вызова конструктора и присвоения значения переменной.

Но все же использовать не рекомендуется.

Корректная реализация отложенной инициализации синглтона

```
public class Something {
        private static class LazySomethingHolder {
                public static Something something = new Something();
        }

        private Something(){
                ...
        }

        public static Something getInstance() {
                return LazySomethingHolder.something;
        }
        ...
}
```

Базовые элементы аппаратной синхронизации

Помимо синхронизации для потокобезопасного доступа к разделяемыми переменным можно использовать атомарные инструкция вида **сравнение-с обменом (Compare-and-Swap, CAS):**

```
boolean compareAndSwap(reference, expectedValue, newValue)
```

Изменение значения переменной происходит, только если текущее значение совпадает с ожидаемым. В противном случае ничего не происходит

CAS поддерживается на аппаратном уровне всеми современными процессорами.

Сравнение с обменом (CAS)

CAS семантически эквивалентна следующему коду

```
public class SimulatedCAS {
          private int value;

          public synchronized int getValue() { return value; }

          public synchronized boolean compareAndSwap(int expectedValue, int newValue) {
                    if (value == expectedValue) {
                              value = newValue;
                              return true;
                    }
                    return false;
     }
}
```

Атомарные типы данных

Начиная с **Java 5** появились реализации **AtomicInteger**, **AtomicBoolean**, **AtomicReference** и др. (`java.util.concurrent.atomic.*`), которые предоставляют атомарные операции основанные на CAS.

Атомарные типы ведут себя так же, как **volatile** переменные.

Пример методов класса AtomicInteger:

- `int addAndGet(int i, int delta)`
- `int decrementAndGet()`
- `int getAndDecrement()`
- `int getAndIncrement()`

Реализация потокобезопасного счётчика

```
public static Counter counter = new Counter();

public class Counter {
    private long value = 0;

    public synchronized long getValue() {
        return value;
    }

    public synchronized long increment() {
        return ++value;
    }
}
```

```
//Thread1
print(counter.getValue());

//Thread2
counter.increment();
```

Реализация счётчика с помощью AtomicLong

```
public class Counter {
        private AtomicLong value = new AtomicLong();

        public long getValue() {
                return value.get();
        }

        public long increment() {
                return value.incrementAndGet();
        }
}
```

```
private volatile long value;

public final long get() {
        return value;
}
```

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

Неблокирующие алгоритмы

Потокобезопасные алгоритмы, использующие для доступа к общим данным CAS, называются неблокирующими.

Основной принцип:

некоторый шаг алгоритма выполняется гипотетически, с осознанием того, что он должен быть выполнен повторно, если операция CAS закончилась неудачно.

Неблокирующие алгоритмы часто называют *оптимистическими*, потому что они выполняются с предположением о том, что не будет конфликтных ситуаций

Потокобезопасный стек. Алгоритм Трайбера

```
import java.util.concurrent.atomic.AtomicReference;
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();
    static class Node<E> {
        final E item;
        Node<E> next;
        public Node(E item) { this.item = item; }
    }

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));
    }
```

Потокобезопасный стек. Алгоритм Трайбера.

```
public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = head.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!head.compareAndSet(oldHead,  newHead));
        return oldHead.item;
    }
}
```

Неблокирующая очередь. Алгоритм Майкла-Скота

```
class LinkedQueue<E> implements Queue<E> {
        private static class Node<E> {
                E item;
                final AtomicReference<Node<E>> next;

                Node(E item, Node<E> next) {
                        this.item = item;
                        this.next = new AtomicReference<Node<E>>(next);
                }
        }

        private AtomicReference<Node<E>> head = new AtomicReference<Node<E>>(
                                                        new Node<E>(null, null));
        private AtomicReference<Node<E>> tail = head;
```

Неблокирующая очередь. Добавление элемента.

```
public boolean offer(E item) {
        Node<E> newNode = new Node<E>(item, null);
        for (;;) {
                Node<E> curTail = tail.get();
                Node<E> residue = curTail.next.get();
                if (curTail == tail.get()) {
                        if (residue == null) {
                                if (curTail.next.compareAndSet(null, newNode)) {
                                        tail.compareAndSet(curTail, newNode);
                                        return true;
                                }
                        } else {
                                tail.compareAndSet(curTail, residue);
                        }
                }
        }
}
```

Неблокирующая очередь. Получение первого элемента.

```
public E poll() {
        for (;;) {
                Node<E> curHead = head.get();
                Node<E> curTail = tail.get();
                Node<E> newHead = curHead.next.get();
                if (curHead == head.get()) {
                        if (curHead == curTail) {
                                if (newHead == null) {
                                        return null;
                                } else {
                                        tail.compareAndSet(curTail, newHead);
                                }
                        } else if (head.compareAndSet(curHead, newHead)) {
                                E item = newHead.item;
                                if (item != null) {
                                        newHead.item = null;
                                        return item;
                                }
                        }
                }
        }
}
```

Неблокирующие коллекции в Java

С Java 5:

- ConcurrentHashMap<K,V> (Использует CAS + гранулярные блокировки для модификации)
- ConcurrentLinkedQueue<E>

С Java 6:

- ConcurrentSkipListMap<K,V>
- ConcurrentSkipListSet<E>

С Java 7:

- ConcurrentLinkedDeque<E>

Плюс: нет взаимных блокировок и ошибок одновременной модификации

Минусы: ниже производительность при интенсивной модификации. Операция size() линейная по времени.

Копирование при модификации

Другой подход – копировать данные при изменении коллекции.

Реализации:

- CopyOnWriteArrayList<E>
- CopyOnWriteArraySet<E>

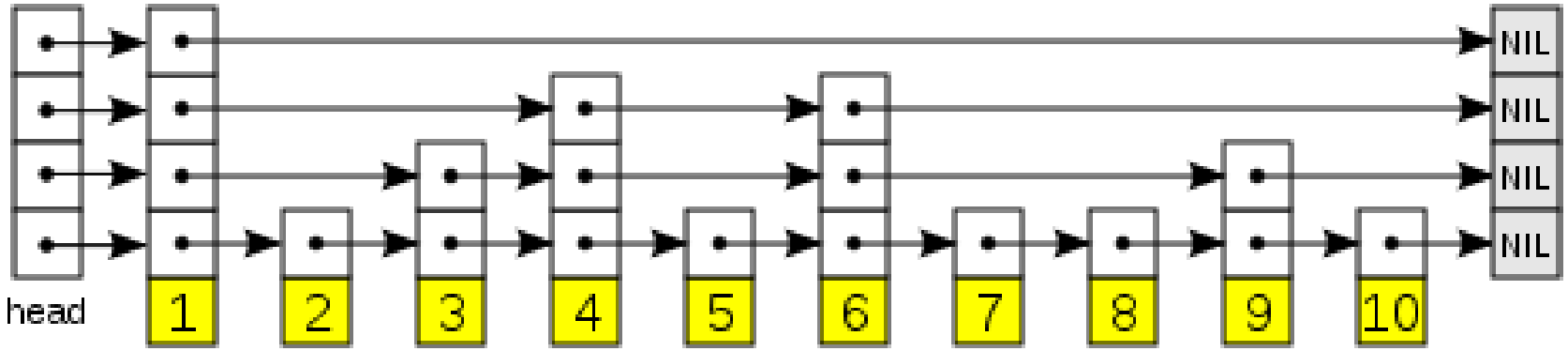
Плюсы:

- нет блокировок при чтении
- iterator() работает со своей копией коллекции на момент вызова метода
- Операции записи синхронизированы и делают копию массива с данными

Минусы:

- экстенсивное потребление памяти

Список с пропусками



Поиск, вставка, удаление – O(log n)

Fork/Join

Fork/Join - фреймворк для многопоточного программирования в Java 7 основанный на легковесных потоках (задачах).

Есть бэкпорт фреймворка для Java 6.

Основные классы:

- ForkJoinPool – управляет распределением задач по потока, управляет запуском, аналог ExecutorServer
- ForkJoinTask/RecursiveTask/RecursiveAction – аналоги Thread/Callable/Runnable

Fork/Join на примере

```
interface TreeNode {
    Iterable<TreeNode> getChildren(); //Retrieve values sequentially by net from some service
    long getValue();
}

public class MaxCounter extends RecursiveTask<Long>{

            private final TreeNode root;

            public MaxCounter(TreeNode root) {
                        this.root = root;
            }

            public static long maxValue(TreeNode root){
                return new ForkJoinPool(100).invoke(new MaxCounter(root));
            }
```

Fork/Join на примере

```
    @Override
    protected Long compute() {
        List<MaxCounter> subTasks = new ArrayList<>();

        for(TreeNode child : root.getChildren()) {
        MaxCounter task = new MaxCounter(child);
            task.fork();
            subTasks.add(task);
        }

        long max = 0;
        for(MaxCounter task : subTasks) {
           long result = task.join();
           if (result > max) max = result;
        }

        return max;
    }
}
```

Fork/Join – управление блокировками

```
class ManagedLocker implements ManagedBlocker {
        final ReentrantLock lock;
        boolean hasLock = false;

        ManagedLocker(ReentrantLock lock) {
                this.lock = lock;
        }

        public boolean block() {
                if (!hasLock)
                        lock.lock();
                return true;
        }

        public boolean isReleasable() {
                return hasLock || (hasLock = lock.tryLock());
        }
}
…
ForkJoinPool.managedBlock(new ManagedLocker(lock));
```

Многопоточное программирование в J2EE

EJB контейнер берет на себя управление системными ресурсами: безопасность, управление потоками, управление ресурсами (pooling, управление транзакциями), позволяет масштабировать приложение.

Вследствие этого возникают множество ограничений на использование API в приложениях.

Enterprise bean’ы не должны:

1. управлять потоками выполнения
2. использовать средства синхронизации для разграничения доступа к ресурсам между экземплярами EJB
3. работать с файловой системой
4. использовать reflection, native вызовы

И т.д.

Средства параллельного программирования в J2EE

- WorkManager API (JSR-237)
- Messaging (JMS, MDB)
- Асинхронный вызов EJB (@Asynchronous в EJB 3.1)
- Вызов EJB по расписанию (Timer Service, @Timeout, @Schedule в EJB 3.1)
- Синхронизация методов в EJB (@Lock в EJB 3.1)
- Асинхронный вызов веб сервисов (JAX-WS)
- Асинхронный ответ в сервлете (Servlet 3.0)

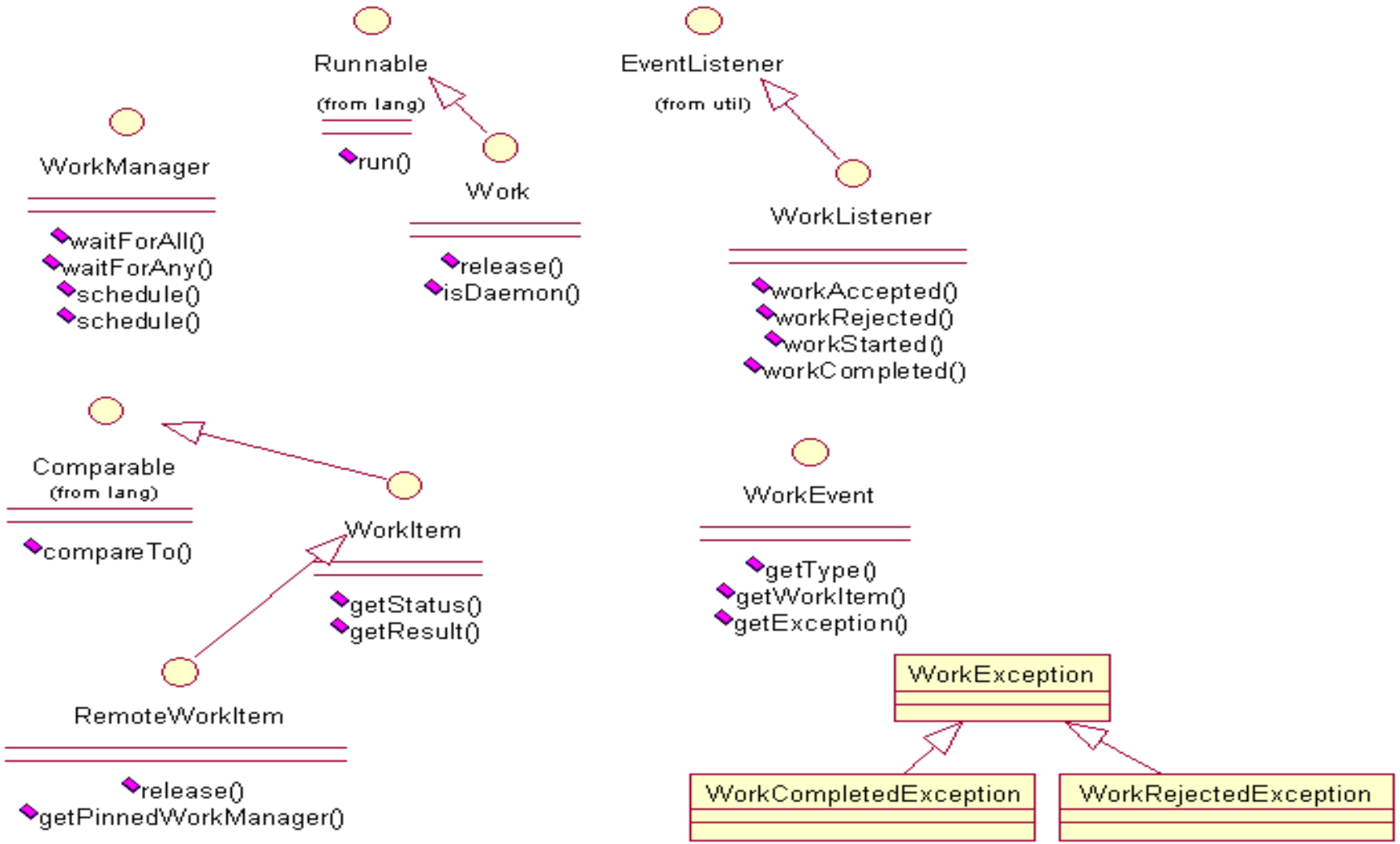
WorkManager API

JSR-237 - API для параллельного выполнения задач в среде J2EE.

Реализации:

- В **Oracle Weblogic** и **IBM WebShpere** используется реализация JSR-237 CommonJ
- В **JBoss** – WorkManager API из JCA (JSR-322)

WorkManager API



Использование WorkManager’а

- Реализовать интерфейс Work. Фактические действия содержатся здесь
- Реализовать интерфейс WorkListener для получения результата выполнения
- Настроить WorkManager в сервере приложений (параметры pool’а потоков, сервер или кластер для выполнения и др.)
- Добавить ресурс в ejb-jar.xml или web.xml

Реализация интерфейса Work

```
import commonj.work.Work;


public class WorkImpl implements Work {

	@Override
	public void run() {
		// Собственно сами действия
	}

	@Override
	public void release() {
		// WorkManager собирается остановить выполнение
	}

	@Override
	public boolean isDaemon() {
		// true если задача требует много времени на выполнение
		return false;
	}

}
```

Реализация интерфейса WorkListener

```
import commonj.work.WorkEvent;
import commonj.work.WorkListener;

public class WorkListenerImpl implements WorkListener {

      @Override
      public void workAccepted(WorkEvent workEvent) {
      }

      @Override
      public void workCompleted(WorkEvent workEvent) {
      }

      @Override
      public void workRejected(WorkEvent workEvent) {
      }

      @Override
      public void workStarted(WorkEvent workEvent) {
      }

}
```

Пример создания задачи

```
            //#1. Получить WorkManager из JNDI
            WorkManager workManager;
            InitialContext ctx = new InitialContext();
            this.workManager =
(WorkManager)ctx.lookup("java:comp/env/wm/TestWorkManager");

            //#2. Создать Work and WorkListener
            Work work = new WorkImpl("HelloWorld");
            WorkListener workListener=new WorkListenerImpl();

            //#3. Назначить выполнение задачи
            WorkItem workItem = workManager.schedule(work,
workListener);

            List<WorkItem> workItemList=new ArrayList<WorkItem>();
            workItemList.add(workItem);

            //#4. Дождаться выполнения всех задач
            this.workManager.waitForAll(workItemList,
WorkManager.INDEFINITE);

            //#5. Получить результаты выполнения, если требуется
            WorkImpl workImpl= (WorkImpl)workItem.getResult();
```

Контекст выполнения задач

Для **POJO** задач

- Задача выполняется с тем же контекстом безопасности
- Java:comp вызывающей компоненты доступен в задаче
- Можно использовать UserTransaction
- Можно использовать удаленное выполнение

Для **EJB** задач

- Это обязательно должна быть @Local ссылка на @Stateless session bean
- Задача выполняется с тем же контекстом безопасности и он может быть переопределен в EJB
- Можно использовать декларативные транзакции
- Используется java:comp enterprise bean’a

Удаленное выполнение задач

JSR-237 WorkManager может выполнять задачи на удалённых JVM:

- Work объект должен реализовывать Serializable
- Эта функциональность зависит от сервера приложений
- Реализовано в CommonJ

Java Messaging System

Java Message Service (JMS) — стандарт middleware для рассылки сообщений, позволяющий приложениям, выполненным на платформе J2EE, создавать, посылать, получать и читать сообщения.

Может использоваться как транспорт для

- вызова внутренних компонет
- удаленного вызова
- вызова SOAP веб сервисов

Можно использовать как синхронно, так и асинхронно

Java Messaging System

Преимущества JMS:

- Надежность
- Гарантированность доставки (поддержка транзакций)
- Масштабируемость
- Асинхронность

Отправка JMS сообщения

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(mappedName="jms/Queue")
private static Queue queue;


connection = connectionFactory.createConnection();
session = connection.createSession(true,
Session.SESSION_TRANSACTED);
messageProducer = session.createProducer(queue);

message = session.createTextMessage();

for (int i = 0; i < NUM_MSGS; i++) {
    message.setText("This is message " + (i + 1));
    System.out.println("Sending message: " +
message.getText());
    messageProducer.send(message);
}
```

Получение JMS сообщения

```
@MessageDriven(mappedName="jms/Queue", activationConfig =  {
        @ActivationConfigProperty(propertyName =
"destinationType",
                                  propertyValue =
"javax.jms.Queue")
    })
public class SimpleMessageBean implements MessageListener {

     public void onMessage(Message inMessage) {
         TextMessage msg = null;

         try {
             if (inMessage instanceof TextMessage) {
                 msg = (TextMessage) inMessage;
                 logger.info("MESSAGE BEAN: Message received: " +
                     msg.getText());
             } else {
                 logger.warning("Message of wrong type: " +
                     inMessage.getClass().getName());
             }
         } catch (JMSException e) {
             e.printStackTrace();
             mdc.setRollbackOnly();
         } catch (Throwable te) {
             te.printStackTrace();
         }
     }
                                 }
```

Асинхронный вызов EJB

В EJB 3.1 появилась простая возможность делать асинхронные вызовы:

добавить **@Asynchronous** аннотацию к методу или всему классу.

При вызове такого метода клиент не будет дожидаться окончания выполнения.

Асинхронный вызов EJB

Существует два вида асинхронных методов:

- Метод возвращает **void**.
- Метод возвращает **Future<?>**

```
@Stateless
//@Asynchronous
public class AsyncBean {
  @Asynchronous
  public void ignoreResult(int a, int b) {
      // …
  }

  @Asynchronous
  public Future<Integer> longProcessing(int a, int b) {
      return new AsyncResult<Integer>(a * b);
  }
}
```

EJB Timer Services

EJB контейнер предоставляет возможность запускать методы по рассписанию

- Можно запускать методы по расписанию, через определенное время, через определенные интервалы
- Можно создавать программно, аннотациями или в deployment конфигурации
- Таймеры можно создавать в @Stateless, @Singleton и @MessageDriven bean’ax
- Таймеры persistent по умолчанию

Пример таймера

```
@Singleton
@Startup
public class DummyTimer3 {
  @Resource TimerService timerService;

  @PostConstruct
  public void initTimer() {
    if (timerService.getTimers() != null) {
      for (Timer timer : timerService.getTimers()) {
        if (timer.getInfo().equals("dummyTimer3.1") ||
            timer.getInfo().equals("dummyTimer3.2"))
          timer.cancel();
      }
    }
    timerService.createCalendarTimer(
      new ScheduleExpression().
          hour("*").minute("*").second("*/10"), new
TimerConfig("dummyTimer3.1", true));
    timerService.createCalendarTimer(
      new ScheduleExpression().
hour("*").minute("*").second("*/45"), new TimerConfig("dummyTimer3.2",
true));
  }
  @Timeout
  public void timeout(Timer timer) {
    System.out.println(getClass().getName() + ": " + new Date());
  }
}
```

Пример таймера EJB 3.1

```
@Schedules({
      @Schedule(hour="*", minute="*", second="*/10",
info="every tenth"),
      @Schedule(hour="*", minute="*", second="*/45",
info="every 45th")
    })
    public void printTime(Timer timer) {
      System.out.println(getClass().getName() + ": " +
          new Date() + ":" +
          timer.getInfo());
    }
```

Синхронизация методов в EJB

Новые возможности в EJB 3.1:

- **@Singleton** – enterprise bean, который реализован как singleton session bean. Существует только один экземпляр bean'а. Можно использовать межпоточную синхронизацию.

Для @Singleton bean'ов можно указывать порядок создания:

- **@Startup** – bean инициализируется при запуске приложения
- **@DependsOn**("MyBean1", "MyBean2") – указанные bean'ы должны быть инициализированы до

Синхронизация управляемая контейнером

```
@Singleton
@AccessTimeout(value=120000)
public class BufferSingletonBean {
  private StringBuffer buffer;

  @Lock(READ)
  private String getContent(){
      return buffer.toString();
  }

  @Lock(WRITE)
  public void append(String string) {
      buffer.append(string);
  }

  @Lock(WRITE)
  @AccessTimeout(value=360000)
  public void flush {
    //Save to db ...
    buffer = new StringBuffer();
  }
}
```

Программная синхронизация

```
@ConcurrencyManagement(BEAN)
@Singleton
public class BufferSingletonBean {
  private StringBuffer buffer;
  private ReadWriteLock lock = new ReentrantReadWriteLock();

  private String getContent(){
       try{
            lock.readLock().lock();
            return buffer.toString();
       }finally{
            lock.readLock().unlock();
       }
  }
  public void append(String string) {
       try{
            lock.writeLock().lock();
            buffer.append(string);
       }finally{
            lock.writeLock().unlock();
       }
  }
  public void flush {
       try{
            lock.writeLock().lock();
            //Save to db ...
            buffer = new StringBuffer();
       }finally{
            lock.writeLock().unlock();
       }
  }
}
```

Асинхронный вызов веб сервисов

Начиная с JAX_WS 2.0 асинхронный вызов веб сервиса можно просто реализовать на стороне клиента.

Клиентский прокси будет сам управлять асинхронным вызовом.

Для этого надо:

1. Добавить пользовательский binding
2. Сгенерировать асинхронный клиентский код
3. Вызывать веб сервис через асинхронные методы

Генерация клиентского кода для асинхронного вызова

Указать пользовательский binding при вызове wsimport

<bindings node="wsdl:definitions"> <package name="com.company.jaxws.stockquote.client"/>
<enableAsyncMapping>true</enableAsyncMapping>

</bindings>

Ant задача:

<wsimport debug=*"${debug}" verbose="${verbose}" keep="${keep}"*

extension=*"${extension}" destdir="${build.classes.home}" wsdl="${client.wsdl}"*>

<binding dir=*"${basedir}/etc" includes="${client.binding.async}"* />

</wsimport>

Клиентский код

```
@WebService(name = "StockQuote")
public interface StockQuote {
    …
    public Response<GetQuoteResponse>getQuoteAsync(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0);
    …
    public Future<?>getQuoteAsync(String arg0,
        AsyncHandler<GetQuoteResponse>asyncHandler);

    …
    public double getQuote(
        String arg0);
}
```

Вызов веб сервиса

```
        //#1
        Response<GetQuoteResponse>resp = port.getQuoteAsync (code);
        while( ! resp.isDone()){
                //some client side processes
        }
        GetQuoteResponse output = resp.get();

        //#2

        port.getQuoteAsync ("MHP", new GetQuoteCallbackHandler (){
             private GetQuoteResponse output;
               public void handleResponse
(Response<GetQuoteResponse>response) {
                        try {
                          output = response.get ();
                    } catch (ExecutionException e) {
                      e.printStackTrace ();
                    } catch (InterruptedException e) {
                      e.printStackTrace ();
                    }
             }
        );
```

Асинхронный ответ в сервлете

```
@WebServlet("/foo" asyncSupported=true)
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest req,
HttpServletResponse res) {
        ...
        AsyncContext aCtx = request.startAsync(req,
res);
        ScheduledThreadPoolExecutor executor = new
ThreadPoolExecutor(10);
        executor.execute(new AsyncWebService(aCtx));
    }
}

public class AsyncWebService implements Runnable {
    AsyncContext ctx;
    public AsyncWebService(AsyncContext ctx) {
        this.ctx = ctx;
    }
    public void run() {
        // Какие-либо действия
        ctx.dispatch("/render.jsp");
    }
}
```

Полезная литература по теме

- **Concurrency in Practice,** Brian Goetz, Doug Lea, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes
- **Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects**, F. Buschmann, K.Henney, D. Schmidt, M. Stal, H. Rohnert
- **Patterns in java**, Mark Grand
- **Concurrent Programming in Java, Design Principles and Patterns**, Doug Lea
- **Patterns for Parallel Programming**, T.Mattson, B.Sanders, B. Massingill

Интернет ресурсы

- **Leveraging EJB Timers for J2EE Concurrency**
  http://www.devx.com/Java/Article/33694/1954
- **The Work Manager API: Parallel Processing Within a J2EE Container**
  http://www.devx.com/Java/Article/28815/1954
- **Simple Asynchronous methods in EJB 3.1**
  http://javahowto.blogspot.com/2010/03/simple-asynchronous-methods-in-ejb-31.html
- **Java Memory Model**
  http://gee.cs.oswego.edu/dl/cpj/jmm.html

Интернет ресурсы

- **Double-checked locking**
  http://www.ibm.com/developerworks/java/library/j-dcl/index.html
- **Non-blocking algorithms**
  http://www.ibm.com/developerworks/java/library/j-jtp04186/
- **Fork/Join framework**
  http://www.oracle.com/technetwork/articles/java/fork-join-422606.html
  http://www.ibm.com/developerworks/java/library/j-jtp03048/index.html

Ваши вопросы?

Спасибо за внимание!