Java Lecture #05

Concurrency

Saint Petersburg, 2012

T · · Systems

Agenda

- Примитивы синхронизации
- Thread-safe коллекции
- Планировщики и пулы потоков
- Fork/Join Framework
- Утилитные классы

Lock

- Lock – интерфейс, обозначающий мьютекс в явном виде
- При этом гораздо более гибкий, чем стандартные java-мониторы
- Основные отличия от synchronized-блоков
  - Вы сами создаете объект-мьютекс
  - Вы сами решаете какие ресурсы защищать
  - Вы сами ответственны за освобождение мьютекса
  - Можно захватывать мьютекс в одном контексте, а отпускать в другом
- Реализация синхронизации на Lock'ах во многих случаях эффективнее synchronized-блоков
- Из за большей гибкости она позволяет накладывать более слабые условия на взаимодействия потоков
- Lock'и тяжелее отлаживать и диагностировать проблемы, ведь с точки зрения JVM это рядовые объекты. Для synchronized-блокировок всегда можно запросить у JVM Thread dump, который покажет все потоки и взятые ими synchronized-блокировки.

Lock

- Рассмотрим реализацию thread-safe счетчика с использованием Lock
- В отличие от synchronized Lock является не средством языка, а обычным объектом с набором методов
- В этом случае критическую секцию ограничивают операции `lock()` и `unlock()`

```
public class Counter{

  private Lock lock = new ReentrantLock();
  private int count = 0;

  public int increment(){
    lock.lock();
    int newCount = ++count;
    lock.unlock();
    return newCount;
  }
}
```

- Вызов lock() блокирует, если Lock в данный момент занят, поэтому удобно использовать метод `tryLock()`, который сразу вернет управление и результат
- При использовании Lock не будут работать стандартные методы `wait()`, `notify()` и `notifyAll()`, ведь монитор как таковой не используется
- Вместо них используются реализации интерфейса Condition, ассоциированные с Lock: необходимо вызвать `Lock.newCondition()` и уже у Condition вызывать методы `await()`, `signal()` и `signalAll()`
- С одним Lock можно ассоциировать несколько Condition

Lock

- Наиболее распространенный паттерн для работы с Lock'ами представлен справа
- Он гарантирует, что Lock будет отпущен в любом случае, даже если при работе с ресурсом будет выброшено исключение

```
Lock l =...;
l.lock();
try {
    // работаем с ресурсом, который
    // защищен этим мьютексом
} finally {
    l.unlock();
}
```

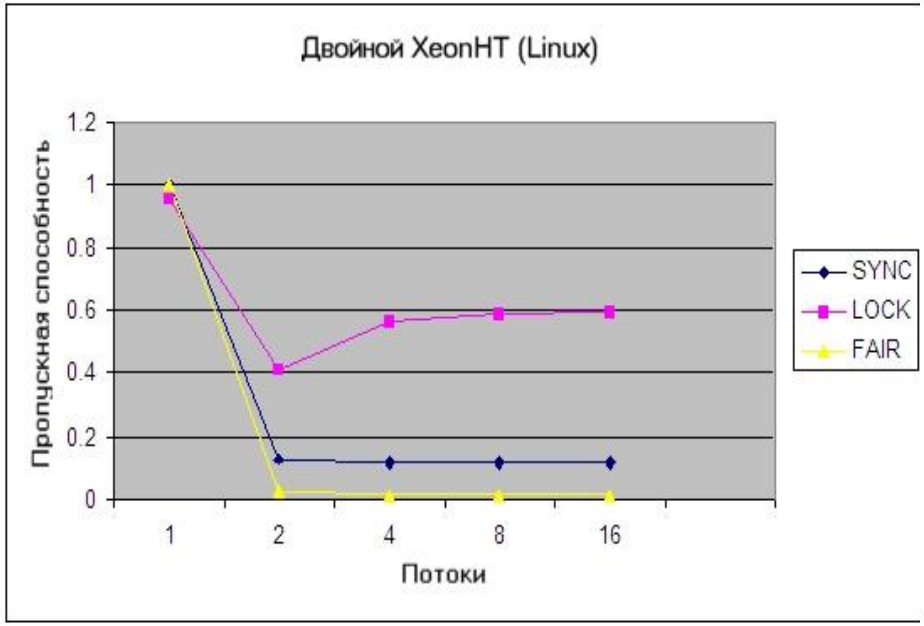
- Для synchronized этот подход неактуален – там средствами языка предоставляется гарантия, что мьютекс будет отпущен
- Этот паттерн весьма полезен в любой ситуации, требующей обязательного освобождения ресурсов
- Широко используются две основные реализации Lock:
  - ReentrantLock допускает вложенные критические секции
  - ReadWriteLock имеет разные механизмы блокировки на чтение и запись, позволяя уменьшить накладные расходы

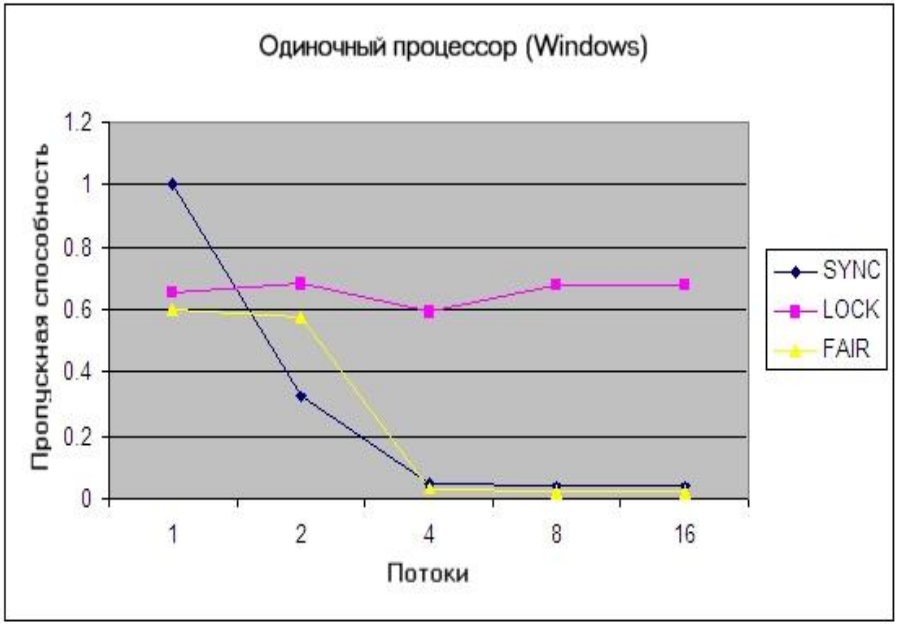
Lock fairness

- Fairness (равнодоступность) – свойство Lock'а, при котором при освобождении управление отдается тому из ожидающих потоков, который ждет дольше всех
- Fairness не распространяется на действия собственно планировщика потоков
- Fair Locks менее производительны, но более предсказуемы, чем Unfair
- В действительности равнодоступность блокировок - очень сильное требование и достигается за счет значительных потерь в производительности
- Учет использования системных ресурсов и синхронизация, необходимые для обеспечения равнодоступности означают, что соперничающие равнодоступные блокировки будут иметь гораздо более низкую пропускную способность, чем неравнодоступные
- По умолчанию следует установить для равнодоступности значение false, если для правильности вашего алгоритма не критично, чтобы потоки обслуживались точно в порядке очереди.
- Блокировки на synchronized изначально unfair и нет способа изменить это поведение

Lock – масштабируемость

- Тест для измерения относительной масштабируемости synchronized в сравнении с Lock, использует генератор псевдослучайных чисел (PRNG).
- Диаграммы показывают пропускную способность в вызовах в секунду, нормализованную до случая synchronized с одним потоком для различных реализаций.
- Как видно, реализация основанная на Lock гораздо лучше масштабируется
- Тест наглядно показывает, что Fair Lock – достаточно дорогое удовольствие





T · · Systems

Compare-And-Set

- Как обеспечить atomicity и visibility без memory barrier'а?
- Compare-and-set (compare-and-swap, CAS) – инструкция, поддерживаемая на уровне процессора (lock:cmpxchg)
- Она позволяет сравнить значение с содержимым памяти и при совпадении выполнить запись
- Эта инструкция позволяет применять оптимистичные блокировки без переключения контекста потока при занятом ресурсе
- Все Atomic-обертки содержат метод `compareAndSet(...)`

- Принцип работы CAS в псевдокоде:
- CAS на многопроцессорных машинах будет дороже из-за аппаратной реализации атомарности операции

```
Boolean CAS(int *ptr, int old, int new) {
    atomically {
        if (*ptr == old) {
            *ptr = new;
            return true;
        } else {
            return false;
        }
    }
}
```

Atomic wrappers

- Обертки над примитивными типами
  - `AtomicInteger`
  - `AtomicLong`
  - `AtomicBoolean`
  - `AtomicReference<T>`
- Предоставляют реализации CAS-операций
- Содержат набор полезных атомарных операций
  - `boolean compareAndSet(int expect, int update)`
  - `int incrementAndGet()`
  - `int getAndIncrement()`
  - `int getAndSet(int newValue)`
  - `int addAndGet(int delta)`
  - `boolean weakCompareAndSet(int expect, int update)`

T · · Systems

Atomic wrappers - пример

```
public class ConcurrentStack <E> {
    AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();

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

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

    private static class Node <E> {
        public final E item;
        public Node<E> next;

        public Node(E item) {
            this.item = item;
        }
    }
```

Agenda

- Примитивы синхронизации
- Thread-safe коллекции
- Планировщики и пулы потоков
- Fork/Join Framework
- Утилитные классы

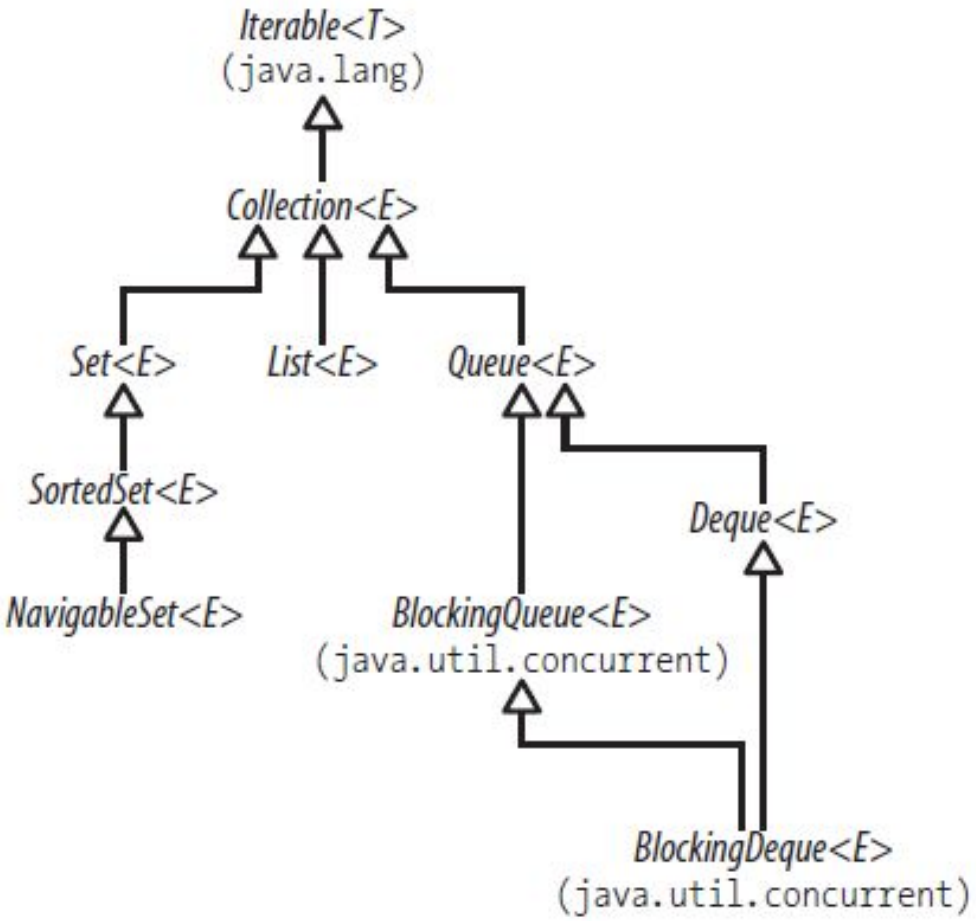
Collections.synchronized…()

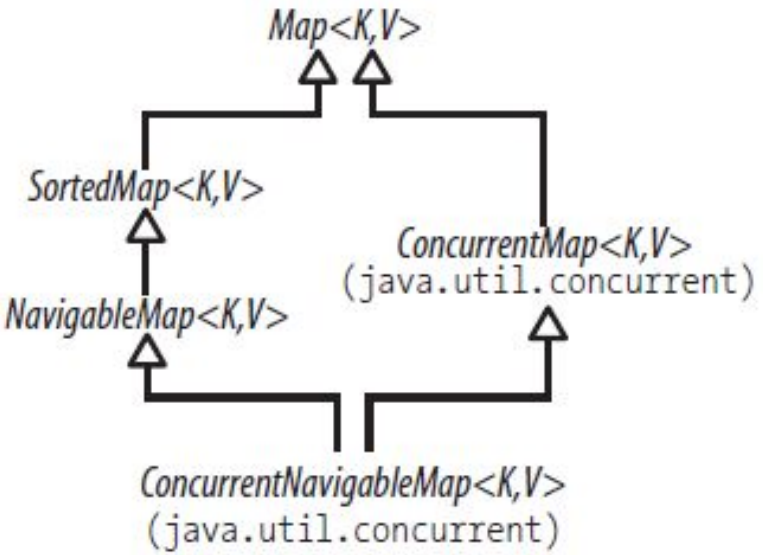
- Класс Collections содержит среди прочих методы
  - `Collections.synchronizedCollection(Collection<T> c)`
  - `Collections.synchronizedList(List<T> list)`
  - `Collections.synchronizedMap(Map<K,V> m)`
  - `Collections.synchronizedSet(Set<T> s)`
- Они возвращают обертки над коллекциями-аргументами с синхронизированными методами
- Этими методами очень удобно оборачивать уже существующие коллекции
- Содержимое можно небезопасно менять путем модификации коллекции-источника
- Итерирование требует внешней синхронизации на коллекции
- Не очень хорошо масштабируются

Legacy implementations

- HashTable<K, V> – синхронизированная реализация интерфейса Map
  - Все методы синхронизированы
  - Потребляет заметно меньше памяти, чем ConcurrentHashMap
  - Плохо масштабируется
  - Последовательности операций на HashTable могут нуждаться в дополнительной внешней синхронизации, если требуется атомарность
- Vector<E> – синхронизированная реализация интерфейса List
  - Все методы синхронизированы
  - Не осуществляет копирования при записи
  - Не дает значительного overhead'а по памяти
  - Процесс итерирования требует внешней синхронизации на самой коллекции

Java.util.concurrent – новые интерфейсы





T-Systems

java.util.ConcurrentMap

- Любые попытки сделать реализацию Map thread-safe упираются в необходимость атомарности группы операций
- Например: «Если в Map нет такого ключа, то положить его»
  - Требует двух операций, которые должны выполняться атомарно
  - Для достижения атомарности придется самостоятельно писать внешние средства синхронизации
  - Непонятно как увязать их с синхронизацией самой коллекции
- ConcurrentMap добавляет к Map методы для обработки часто встречающихся связанных операций на Map:
  - `V putIfAbsent(K key, V value)`

    что эквивалентно

```
if (!map.containsKey(key))
    return map.put(key, value);
else
    return map.get(key);
```

java.util.ConcurrentMap

- `boolean remove(Object key, Object value)`

```
if (map.containsKey(key) && map.get(key).equals(value)) {
    map.remove(key);
    return true;
} else {
    return false;
}
```

- `boolean replace(K key, V oldValue, V newValue)`

```
if (map.containsKey(key) && map.get(key).equals(oldValue)) {
    map.put(key, newValue);
    return true;
} else {
    return false;
}
```

- `V replace(K key, V value)`

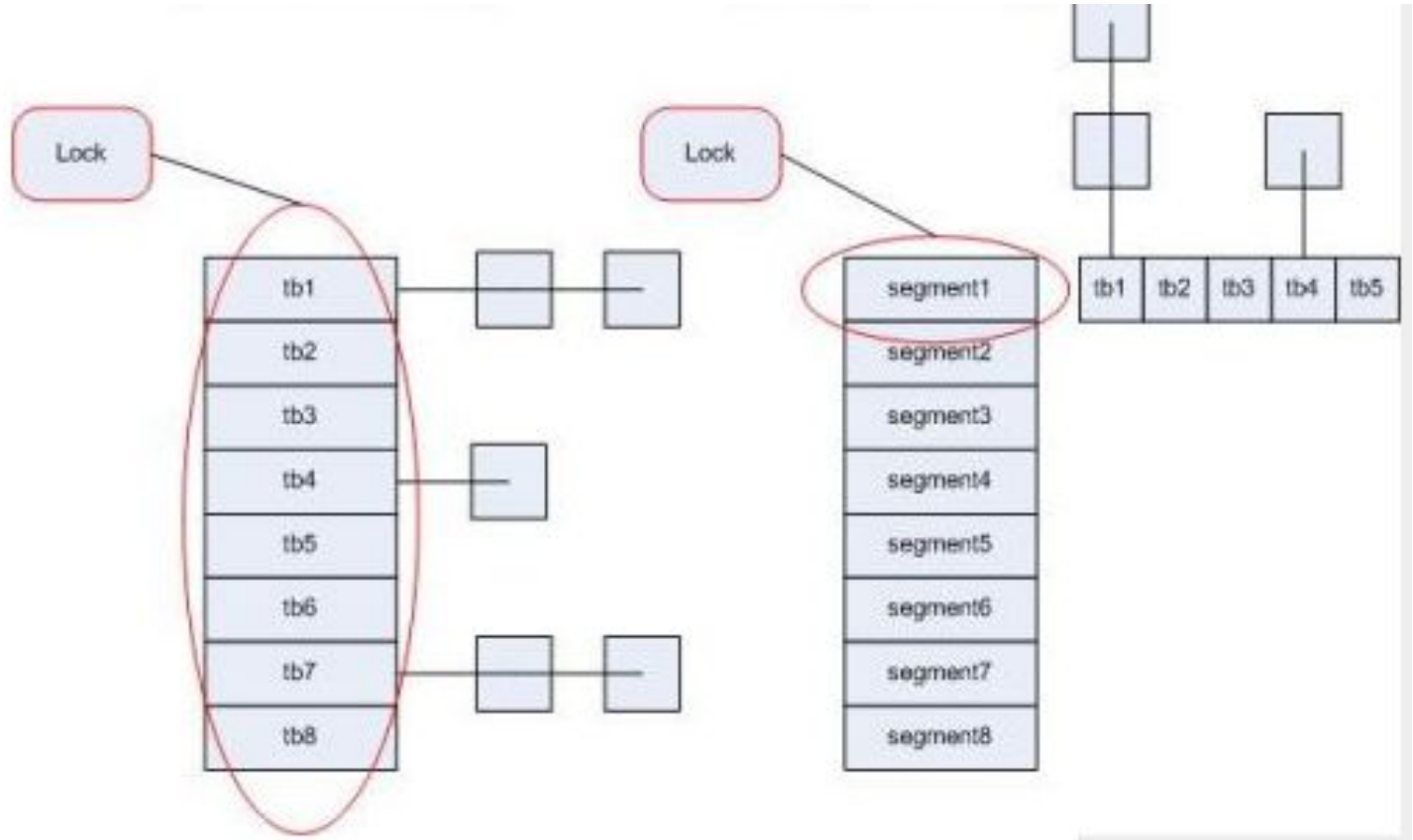
```
if (map.containsKey(key)) {
    return map.put(key, value);
} else {
    return null;
}
```

ConcurrentHashMap

- Основная thread-safe реализация интерфейса Map<K,V>
- Реализует также ConcurrentMap
- Внутри похожа на HashMap, но имеет дополнительные механизмы синхронизации
- Масштабируется гораздо лучше HashTable, практически линейно
- Не синхронизирует операции чтения
- Операции чтения отражают результат последней завершенной операции записи, не учитывая те, что еще в процессе
- Итераторы отображают состояние коллекции на момент создания итератора
- Позволяет задавать concurrency level – размер сегмента хэш-таблицы, блокируемого на запись
- Потребляет заметно больше памяти, чем HashTable

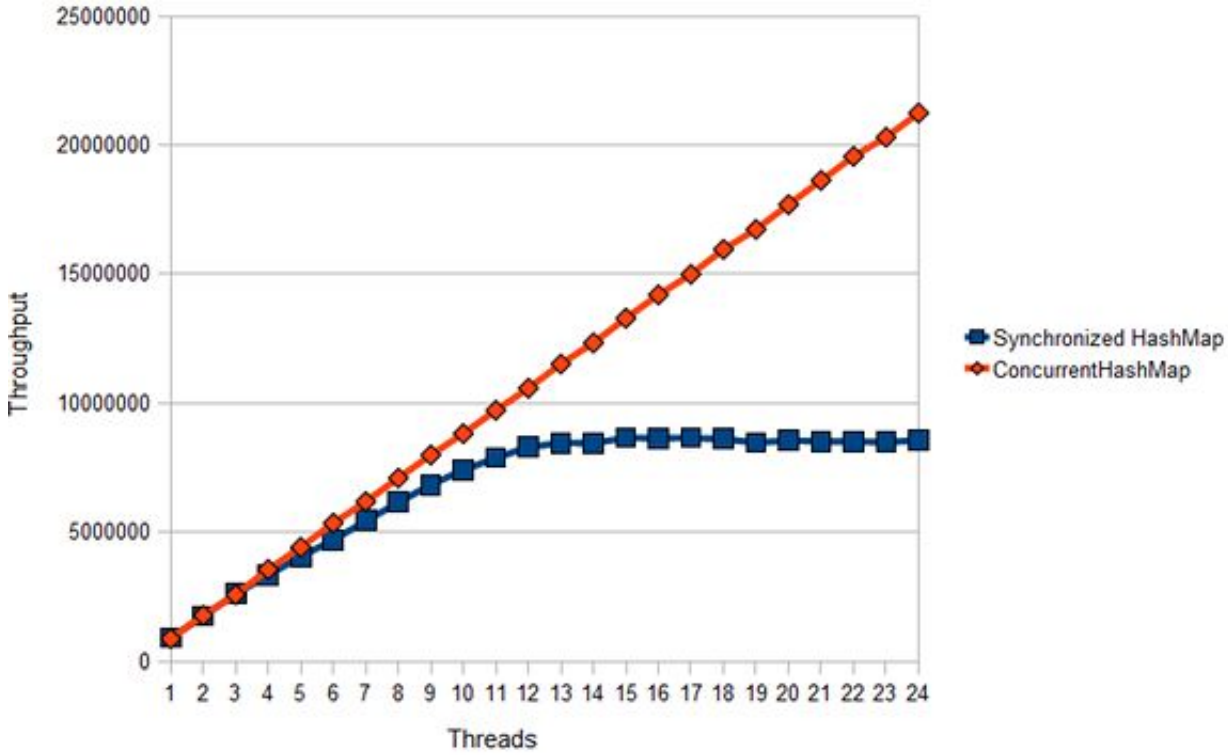
ConcurrentHashMap

- Версии реализации от 7 и ниже используют сегментированную структуру
- При записи блокируется не весь Map, а один сегмент
- Разрабатываемая версия 8 будет блокироваться уже на конкретных bucket'ах, а сегменты исчезнут



ConcurrentHashMap vs HashTable

- HashTable блокирует всю таблицу целиком, ограничивая вертикальную масштабируемость
- С какого-то момента добавление новых ядер уже не дает прироста производительности



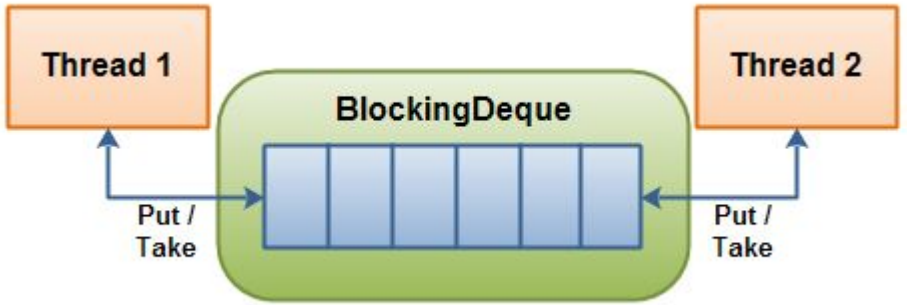
Threads	ConcurrentHashMap	Hashtable
1	1.00	1.03
2	2.59	32.40
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41

T · · Systems

Blocking Queues

- Отлично подходят для реализации шаблона Producer-Consumer
- Добавляют набор блокирующих методов для работы с очередью
- Могут быть Fair по отношению к использующим потокам
- `ArrayBlockingQueue<E>`
  - Ограниченная очередь на базе массива
- `PriorityBlockingQueue<E>`
  - Очередь с сортировкой элементов по Comparator'у
  - Неограниченная очередь
- `SynchronousQueue<E>`
  - Очередь из **одного(!)** элемента
  - Операция добавления блокирует до соответствующей операции чтения из другого потока

- Интерфейс BlockingDeueue расширяет BlockingQueue методами работы с обоими концами структуры



T · · Systems

Blocking queues: API reference

	Throws Exception	Special Value	Blocks	Times Out
Insert	`addFirst(o)`	`offerFirst(o)`	`putFirst(o)`	`offerFirst(o, timeout, timeunit)`
Remove	`removeFirst(o)`	`pollFirst(o)`	`takeFirst(o)`	`pollFirst(timeout, timeunit)`
Examine	`getFirst(o)`	`peekFirst(o)`		

	Throws Exception	Special Value	Blocks	Times Out
Insert	`addLast(o)`	`offerLast(o)`	`putLast(o)`	`offerLast(o, timeout, timeunit)`
Remove	`removeLast(o)`	`pollLast(o)`	`takeLast(o)`	`pollLast(timeout, timeunit)`
Examine	`getLast(o)`	`peekLast(o)`		

Copy-on-write

- CopyOnWriteArrayList и CopyOnWriteArraySet основаны на массиве, копируемом при операции записи
- Уже открытые итераторы при этом не увидят изменений в коллекции
- Эти коллекции следует использовать только когда 90+% операций являются операциями чтения
- При частых операциях модификации большая коллекция способна убить производительность
- Сортировка этих коллекций не поддерживается, т.к. она подразумевает O(n) операций вставки
- Итераторы по этим коллекциям не поддерживают операций модификации

Copy-on-write - Реализация

- Операция добавления элемента в список:

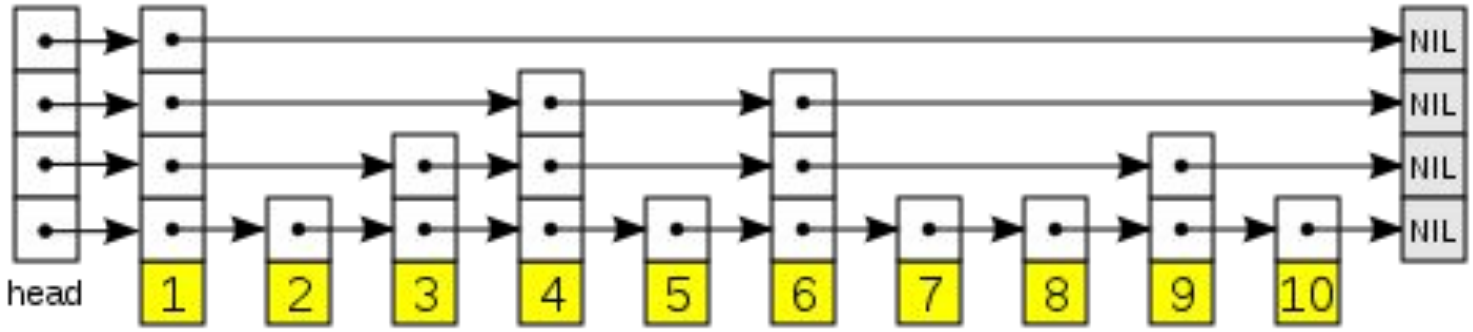
```
CopyOnWriteArrayList cow = new CopyOnWriteArrayList();
cow.add(new Object());
```

- И её реализация в CopyOnWriteArrayList:

```
public boolean add(E e) {
final ReentrantLock lock = this.lock;
lock.lock();
try {
    Object[] elements = getArray();
    int len = elements.length;
    Object[] newElements = Arrays.copyOf(elements, len + 1);
    newElements[len] = e;
    setArray(newElements);
    return true;
} finally {
    lock.unlock();
}
}
```

Skip Lists

- ConcurrentSkipListMap и ConcurrentSkipListSet основаны на Skip List'ах
- Это единственные доступные thread-safe реализации NavigableSet и NavigableMap
- Skip List, как правило, занимает больше памяти, чем хэш-таблица
- Гарантирует O(log(n)) для большинства операций
- ConcurrentSkipListMap, в отличие от ConcurrentHashMap, не предоставляет средств для performance-тюнинга
- ConcurrentSkipListMap также реализует ConcurrentMap
- Это единственные упорядоченные thread-safe коллекции



Итераторы

- Как правило итераторы коллекций из java.util.concurrent не бросают ConcurrentModificationException
- Они не являются fail-fast
- Они гарантированно отражают состояние коллекции на момент создания итератора
- Итераторы не блокируют другие операции или итераторы на исходной коллекции
- При этом они могут содержать и более поздние изменения, но это не гарантируется

Итераторы

- Выполнение этого кода приводит к ConcurrentModificationException
- Если заменить реализацию на CopyOnWriteArrayList, то исключения не будет

```
List<String> myList = new ArrayList<String>();
myList.add("1");
myList.add("2");
Iterator<String> it = myList.iterator();
while (it.hasNext()) {
    String value = it.next();
    System.out.println("List Value:" + value);
    if (value.equals("2")) myList.remove(value);
}
```

Agenda

- Примитивы синхронизации
- Thread-safe коллекции
- Планировщики и пулы потоков
- Fork/Join Framework
- Утилитные классы

Callable

- Имеет единственный метод V call()
- По принципу действия схож с Runnable

```
FutureTask<String> future =
      new FutureTask<String>(new Callable<String>() {
        public String call() {
          return getBatch("ALL_EMPLOYEES");
      }});
    executor.execute(future);
if(future.isDone()){
    future.get();
}
```

Executor

- Executor – интерфейс, обозначающий абстрактную систему для асинхронного исполнения задач
- В него передают исполняемый код, а он заботится о выборе потока для исполнения
- При этом он может содержать несколько потоков, обеспечивая их эффективное переиспользование
- Класс Executors представляет собой фабрику для создания Executor'ов
- Эта фабрика позволяет создавать разнообразные очереди и пулы потоков, избавляя программиста от необходимости писать однообразный инфраструктурный код
- Простой пример использования Executor'а представлен ниже

```
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
    public void run() {
        // этот код будет выполнен в отдельном потоке
    }
});
```

ExecutorService

- Представляет собой расширение Executor'а с дополнительными возможностями
- Способен создавать объекты Future<T>, представляющие собой результаты выполнения асинхронных операций
- Основные методы:
  - `Submit(...)` – различные варианты этого метода принимают задачу на выполнение
  - `invokeAll()` – метод выполнит переданный в него список задач и вернет управление тогда, когда все задачи будут завершены или наступит таймаут
  - `invokeAny()` – метод выполнит переданный в него список задач и вернет управление тогда, когда хотя бы одна задача будет завершена или наступит таймаут
  - `shutdown()` – при вызове этого метода ExecutorService закончит выполнение текущих задач, но новых принимать уже не будет
- Многие Executor'ы, возвращаемые фабрикой Executors на самом деле являются реализациями ExecutorService

Future<T>

- Future – интерфейс, семантически обозначающий результат выполнения асинхронной операции
- Тип-параметр – это тип результата операции
- Важные методы интерфейса
  - `get()` позволяет получить результат операции, блокируя, если результата еще нет
  - `cancel()` останавливает выполнение задачи, если только она еще не завершена
  - `isDone()` позволяет определить, завершена ли задача
- Все операции в рамках выполнения задачи happens-before любых операций после вызова метода get()
- Самой распространенной реализацией Future является FutureTask
- FutureTask также реализует Runnable, так что его экземпляры удобно передавать в Thread или Executor

Future<T> - Пример

```
interface ArchiveSearcher {
    String search(String target);
}
class App {
    ExecutorService executor;
    ArchiveSearcher searcher;

    void showSearch(final String target)  throws InterruptedException {
        Future<String> future  = executor.submit(new Callable<String>() {
            public String call() {
                return searcher.search(target);
            }});
        displayOtherThings(); // выполнение других задач
        try {
            displayText(future.get()); // пытаемся получить результат
        } catch (ExecutionException ex) {
            cleanup();
        }
    }
}
```

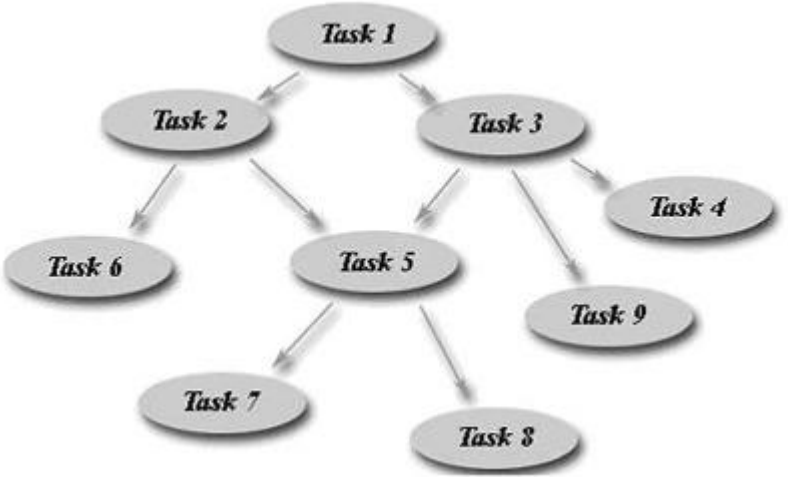
Agenda

- Примитивы синхронизации
- Thread-safe коллекции
- Планировщики и пулы потоков
- Fork/Join Framework
- Утилитные классы

Fork/Join

- Fork/Join – это подход к написанию многопоточных программ, основанный на следующем рекурсивном алгоритме:
  - Если задача достаточно мала – выполнить её
  - Если нет – разбить на несколько и выполнять их, а результаты агрегировать
  - Для подзадач вернуться к пункту 1.
- Этот подход отлично работает для большого количества однотипных задач

Схема справа представляет собой типовое дерево задач, полученное в результате декомпозиции



Fork/Join Framework

- Начиная с Java 7 в стандартные библиотеки Java включен Fork/Join Framework, который предоставляет инфраструктуру для подобной декомпозиции
- Изначально он входил в JSR-166, который описывал практически все содержимое пакета java.util.concurrent
- Тем не менее, ему потребовалось еще 6 лет, чтобы попасть в мейнстрим
- При этом существует много сторонних реализаций Fork/Join, например Tymeac

Fork/Join Framework API

- Интерфейс ForkJoinTask<T> представляет собой небольшую задачу как результат декомпозиции на этапе Fork. Есть две реализации:
  - RecursiveAction – не возвращает результат работы для этапа Join
  - RecursiveTask<T> - возвращает результата типа T для использования на этапе Join
- ForkJoinPool – реализация Executor'а для ForkJoinTask
- В конструкторе ему можно задать размер пула, по умолчанию он равен количеству процессоров в системе
- С каждым потоком связана очередь задач, пополняемая вызовами fork()
- Поток исполняет их, начиная с самых новых
- Если очередь заканчивается, то поток старается украсть задачи из очередей других потоков пула

Fork/Join Framework - Пример

- Пример вычисляет N-ый член последовательности Фибоначи методом Fork/Join
- Одна часть выполняется в текущем потоке, вторая – шедулится на Thread pool

```
class Fibonacci extends RecursiveTask<Integer> {
    final int n;

    Fibonacci(int n) {
        this.n = n;
    }

    Integer compute() {
        if (n <= 1)
            return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        return f2.compute() + f1.join();
    }
}
```

Agenda

- Примитивы синхронизации
- Thread-safe коллекции
- Планировщики и пулы потоков
- Fork/Join Framework
- Утилитные классы

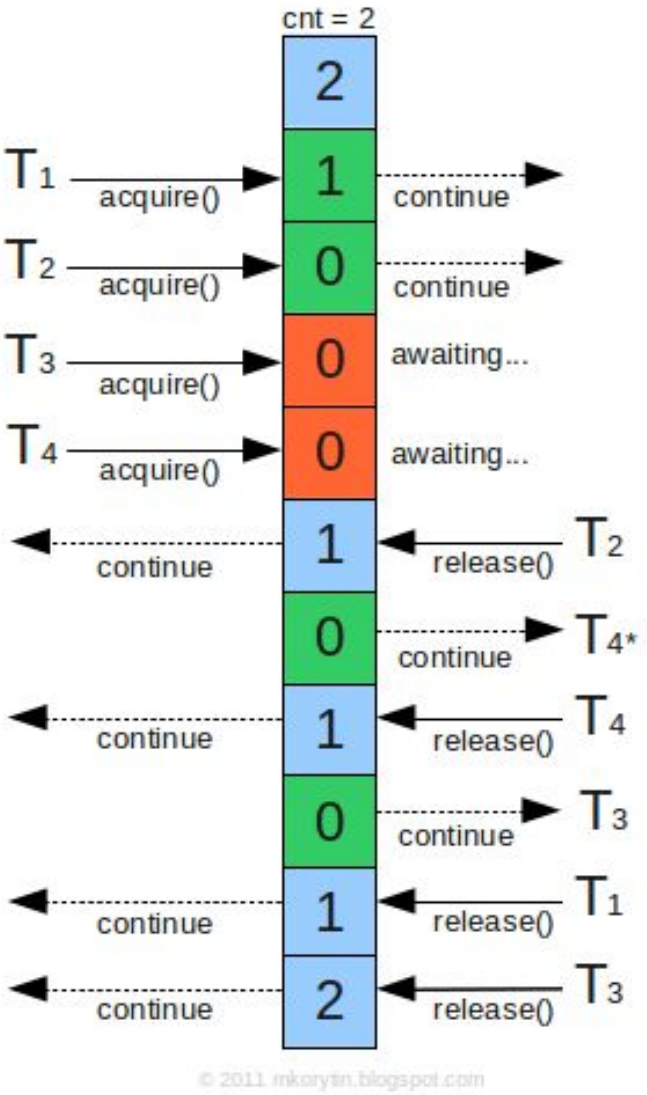
ThreadLocal<T>

- ThreadLocal – типизированный контейнер для объектов, ассоциирующий содержимое с текущим потоком.
- Проще говоря, ThreadLocal возвращает каждому потоку свой экземпляр объекта
- Пример ниже иллюстрирует самую распространенную схему использования ThreadLocal: ассоциация объекта с потоком
- В данном случае с потоком ассоциируется уникальный идентификатор

```
class UniqueThreadIdGenerator {

    private static final AtomicInteger uniqueId = new AtomicInteger(0);

    private static final ThreadLocal < Integer > uniqueNum =
        new ThreadLocal < Integer > () {
            @Override protected Integer initialValue() {
                return uniqueId.getAndIncrement();
            }
    };

    public static int getCurrentThreadId() {
        return uniqueNum.get();
    }
}
```

Semaphore

- Объект, позволяющий войти в заданный участок кода не более чем *n* потокам одновременно
- N определяется параметром конструктора
- При N=1 по действию аналогичен Lock
- Fairness – гарантия очередности потоков

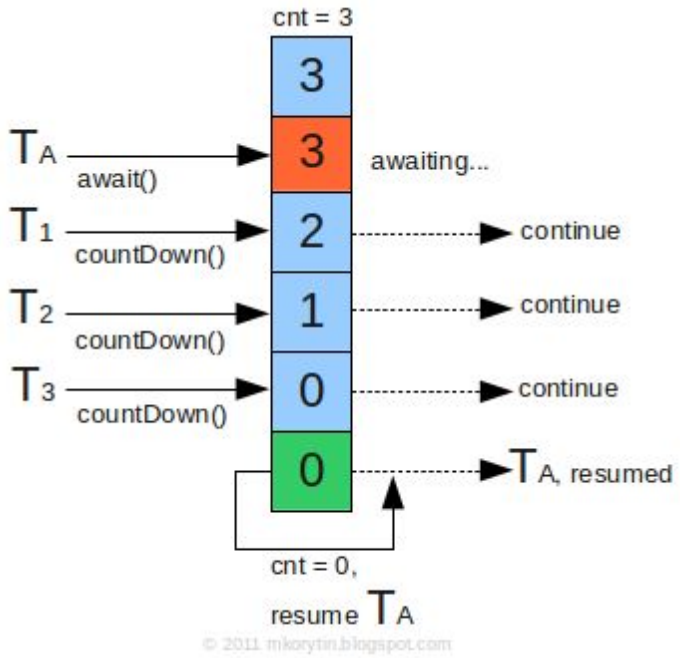



```
Semaphore semaphore = new Semaphore(4);
semaphore.acquire();
semaphore.release();
boolean available = semaphore.tryAcquire();
```

CountDownLatch

- Утилитный класс для синхронизации действий потоков
- Представляет собой барьер, на котором потоки ждут некоторого события
- В основе лежит счетчик, который можно уменьшать от начального значения до нуля

- Он предоставляет две основные операции:
  - `countDown()` – уменьшает значение счетчика на единицу
  - `await()` – текущий поток будет заблокирован пока значение счетчика не упадет до нуля
- Перезапустить CountDownLatch нельзя



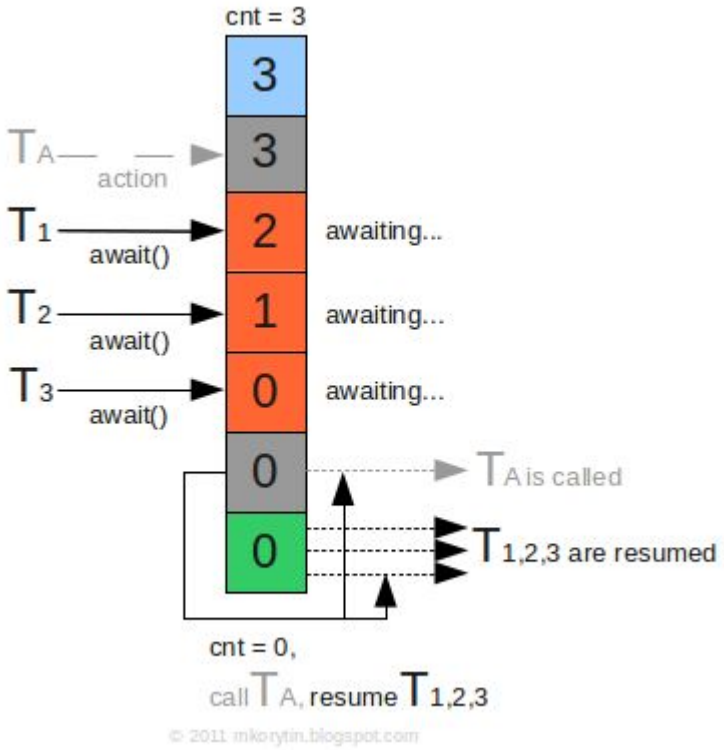
CountDownLatch - Пример

```
CountDownLatch latch = new CountDownLatch(1);
Waiter  waiter  = new Waiter(latch);
new Thread(waiter).start();
new Thread(waiter).start();
Thread.sleep(10000);
latch.countDown();
```

```
public class Waiter  implements Runnable{
    CountDownLatch latch = null;
    public Waiter(CountDownLatch latch) {
        this.latch = latch;
    }

    public void run() {
        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Waiter Released");
    }
}
```

CyclicBarrier

- Позволяет N потокам дождаться друг друга в некоторой точке выполнения
- N задается параметром конструктора
- Как только все N потоков вызовут `await()` их разом отпустит
- Опционально можно задать некий Runnable, который будет выполнен в момент разблокировки
- CyclicBarrier можно перезапустить методом `reset()`



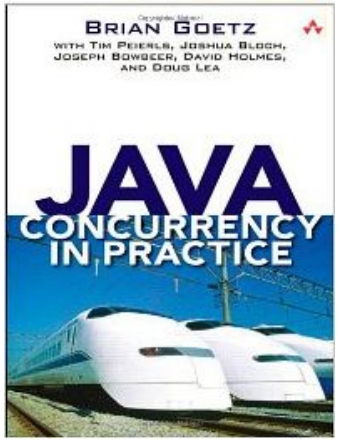
CyclicBarrier - Пример

```
int N = 10;
final CyclicBarrier cb = new CyclicBarrier(N);
for (int i = 0; i < N; i++) {
    final int idx = i;
    new Thread(new Runnable() {
        public void run() {
            System.out.println("T" + idx + ": await");
            try {
                cb.await();
            } catch (InterruptedException ex) {
                System.out.println("T" + idx + ": interrupted");
                return;
            } catch (BrokenBarrierException ex) {
                System.out.println("T" + idx + ": broken");
                return;
            }
            System.out.println("T" + idx + ": continue");
        }
    }).start();
}
```

T · · Systems

Library

- Brian Goetz. Java concurrency in practice

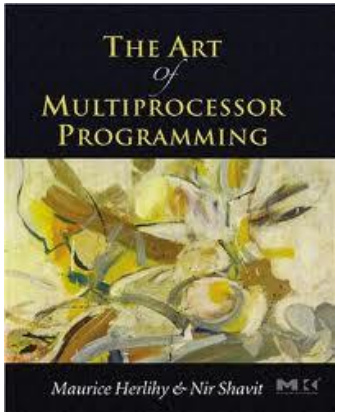


- Java Language Specification, глава 17

  http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html

- Maurice Herlihy , Nir Shavit.

  The art of multiprocessor programming



- Статьи Brian'a Goetz'a на http://www.ibm.com/developerworks

  (например http://www.ibm.com/developerworks/ru/library/j-jtp10185/index.html)

T · · Systems