Наборы данных для потоков

Классы коллекций

- До Java 2
  - Vector<E> is synchronized
  - Hashtable<K,V> is synchronized
  - Properties is synchronized
- Новые коллекции Java 2
  - . . . is not synchronized
  - synchronization wrapper
- Начиная с Java 5
  - Наборы для безопасной работы с потоками

Старые коллекции

- Vector
- Hashtable

Предназначены для работы из нескольких потоков

Все методы синхронизованы

Коллекции Java 2

- ArrayList <>
- HashMap <>
- . . .

Не обеспечивают безопасную работу с потоками

Необходима явная синхронизация

Особенности использования итераторов:

Если после создания итератора, набор данных изменился в другом потоке, генерируется исключение ConcurrentModificationException

```
List myList = new ArrayList<…>();

synchronized(myList){
    Iterator iter =myList.iterator();
    while (iter.hasNext()){ ….}
}
```

Синхронизующие оболочки

Позволяют обеспечить синхронизацию обычной коллекции там, где она необходима

```
ArrayList<…> usynch = new ArrayList<…>();

 List<. . .> synchAL =
    Collections.synchronizedList (usynch);
```

Синхронизующие оболочки

Получаем двойной доступ – с синхронизацией и без

usynch.add(. . .);

synchAL.clear();

Синхронизующие оболочки

Если нельзя допустить использование потоками несинхронизированных методов, то не сохраняем ссылку на базовый объект

```
Map<…>  synchM =
   Collections.synchronizedMap(
                  new HashMap<…>());
```

!!!

Синхронизующие оболочки управляют лишь методами коллекции

Их действие не распространяется на итераторы коллекций

При использовании итераторов и цикла for each необходима явная блокировка

Наборы для безопасной работы с потоками

Блокирующие очереди

Эффективные очереди

Эффективные хеш-таблицы

Массивы, копируемые при записи

Блокирующие очереди

Приостанавливает работу потока, если он пытается добавить элемент в заполненную очередь или извлечь элемент из пустой очереди

Предоставляют операции трех типов

- Генерирующие исключение при ошибочном использовании
- Возвращающие значение, свидетельствующее об ошибке
- Блокирующие

Операции

Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

Операции

- Генерирующие исключение

  add() - при переполнении очереди illegalStateException
  remove() - если очередь пуста NoSuchElementException
  element() если очередь пуста NoSuchElementException

- Возвращающие признак ошибки

  offer() -false, если очередь заполнилась
  poll()- null, если очередь пуста
  peek() - null, если очередь пуста

- Блокирующие

  put()
  take()

Классы блокирующих очередей

- LinkedBlockingQueue <E>
- ArrayBlockingQueue <E>
- DelayQueue <E extends Delayed>
- PriorityBlockingQueue <E >

SynchronousQueue<E>

- Реализует объединение потока-поставщика и потока-потребителя
- Данные передаются только в одном направлении
- Очередь не используется
- Если поток вызвал метод **`put()`**, его выполнение приостанавливается, пока другой поток не вызовет метод **`take()`**

Java 6 и Java 7

- 1.6 Добавлены блокирующие деки
  - **LinkedBlockingDeque<E>**
- 1.7 интерфейс *TransferQueue*
  - когда производитель отправляет сообщение потребителю с помощью метода *transfer ()* , производитель остается заблокированным, пока сообщение не будет использовано

Эффективные коллекции

- Эффективные наборы поддерживают большое количество читающих потоков и фиксированное число записывающих.
- Наборы данных используют специальные алгоритмы, позволяющие избежать блокирования всей структуры и минимизировать конфликты, допуская одновременный доступ к различным частям структуры
- Эффективные наборы данных представляют «слабо согласованные итераторы»
- Это означает, что итератор не обязательно отражает все изменения, произошедшие после их создания, но никогда не возвращают значение дважды и не вызывают исключение

Классы эффективных коллекций

- ConcurrentLinkedQueue <E>
- ConcurrentHashMap <E>
- CopyOnWriteArrayList <E>
- CopyOnWriteArraySet <E>

ConcurrentHashMap<K,V>

```
// изменена хеш-функция и способ организации карты

// добавить, если нет объекта value по ключу
   V putIfAbsent(K key, V value);
 // удалить, если имеется объект value с ключом
   boolean remove(K key, V value);
// заменить oldValue новым newValue объекта с ключом
   boolean replace(K key, V oldValue, V newValue);
// заменить новым значением newValue объект с ключом
    V replace(K key, V newValue);
```

Коллекции, копируемые при записи

- CopyOnWriteArrayList <E>
- CopyOnWriteArraySet <E>

Перед каждой модификацией коллекция копирует свое содержимое в новую, чтобы операции чтения содержимого коллекции выполнялись без синхронизации (так как они никогда не работают с изменяемыми данными)

Асинхронные вычисления

Интерфейсы Callable и Future

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

Например

Callable <Integer> - асинхронные вычисления, результат Integer

```
public class MyCallable implements Callable<Integer> {
    @Override
   public Integer call() throws Exception {
      Integer value;
            . . .
            return value;
   }
```

Интерфейсы
Callable и Future

- интерфейс Future описывает набор методов для работы с результатами асинхронных вычислений

```
public interface Future<V> {
    V get() throws…
    V get(long timeout, TimeUnit unit) throws…
    boolean cancel(boolean mayInterrupt) ;
    boolean  isCancelled();
    boolean isDone();

    }
```

Методы

– V get() throws…

– V get(long timeout, TimeUnit unit) throws…

Возвращает результат асинхронных вычислений, выполнение блокируется до окончания вычислений или до истечения указанного интервала времени

Методы

– boolean cancel(boolean mayInterrupt)

Пытается отменить выполнение задачи.

Если задача уже запущена и параметр mayInterrupt равен true, она прерывается.

Если вычисления еще не начаты, они не начнутся никогда. Если отмена задачи прошла успешно. Метод возвращает значение true.

Методы

– boolean isCancelled()

Возвращает true, если задача была отменена до ее нормального завершения

– boolean isDone()

Возвращает true, если выполнение задачи завершено, если выполнение было прекращено или если в процессе ее выполнения возникло исключение

Класс FutureTask

- Реализует интерфейсы
  - Runnable
  - Future <V>

Использование FutureTask

```
Callable<Integer> myComp = . . .;
FutureTask <Integer>  task=
   new FutureTask<Integer>(myComp);
Thread t = new Thread(task); //Runnable
t.start();
. . .
Integer result = task.get(); //Future
```

Пример (подсчет количества файлов)

```
class MyCounter  implements   Callable <Integer> {
MyCounter (File name){
…
}
. . .
public Integer call(){
 Integer count;
List<Future<Integer>> results = new ArrayList<>(); //Список для создаваемых FutureTask
   . . . // проверяем свойство файла  (следующий слайд)
         // или рекурсивно создаем новый  объект MyCounter
  return count;
}
}
```

Пример (подсчет количества файлов)

```
if (file.isDirectory()){
{// для каталогов перебираем элементы (elem) и создаем рекурсивно
. . .
  { MyCounter counter =new MyCounter(elem);
  FutureTask <Integer> task= new FutureTask<Integer>(counter);
  results.add(task); //добавили задание для подкаталога
  Thread t = new Thread(task);
  t.start();}
}
// формируем результат для каталога (следующий слайд)
} else  {   //для файла проверяем условие
     if (???(file)) count++;
}
```

Пример (подсчет количества файлов)

Результаты

- Для подкаталогов в методе call

```
for (Future <Integer> result : results)
  try {
          count += result.get();
      }
      catch (ExecutionException e) { . . .  }
```

Пример (подсчет количества файлов)

```
String directory = . . .;
MyCounter counter = new MyCounter(new File(directory));
FutureTask<Integer> task = new FutureTask<>(counter);
Thread t = new Thread(task);
t.start();
try {
        System.out.println(task.get() + " matching files.");
    }   catch (ExecutionException e)    {
            e.printStackTrace();
    }   catch (InterruptedException e)    {
            e.printStackTrace();
}
```

Интерфейсы
Executor и ExecutorService

- интерфейсы для планирования и управления Runnable объектами
- определено несколько реализаций Executor, которые предлагают различные характеристики планирования. Постановка задачи в очередь к обработчику делается методом execute()

Executor

```
Executor executor = . . .;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
...
```

Отдельный поток для каждого задания

```
class ThreadPerTaskExecutor
  implements Executor {
public void execute
  (Runnable r) {
     new Thread(r).start();
   }
}
```

Запуск задания в том же потоке

```
class DirectExecutor
  implements Executor {
    public void execute
        (Runnable r) {
        r.run();
    }
}
```

ExecutorService

- Интерфейс, наследующий Executor добавляет методы для управления завершением и методы, которые могут создавать Future для отслеживания хода выполнения одной или нескольких асинхронных задач

ExecutorService

- Методы
  - void shatdown()
  - List<Runnable> shatdownNow()
  - <T> Future<T> submit(Callable<T> task)
  - Future<?> submit (Runnable task)
  - <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
  - <T> T invokeAny(Collection<? extends Callable<T>> tasks)

ExecutorService

Интерфейс ExecutorService добавляет методы, позволяющие управлять выполняющимися задачами

Метод submit() позволяет получить объект Future, связанный с выполняемой задачей.

С его помощью можно опросить состояние задачи.

ExecutorService

```
ExecutorService executor = . . .

Callable<String> call = new MyCallable();
Future<String> fut = executor.submit(call);

if (fut.isDone()){
        System.out.println(fut.get());
```

Executors

- Объекты ExecutorService создаются через статические методы класса Executors
- Цель – уменьшить затраты на запуск нового потока
- Для этого сразу создаются наборы потоков (пулы), которые принимают задания на выполнение
- В пул помещаются объекты Runnable
- Потоки из пула запускают метод run()
- После завершения метода run() поток не удаляется, а остается готовым к выполнению новой задачи

Executors

Executors.newCachedThreadPool()

пул потоков немедленно начинает выполнение каждой задачи

новые потоки создаются по мере необходимости

бездействующие потоки удаляются через 60 секунд

Executors

Executors.newFixedThreadPool(n)

создает пул потоков фиксированного размера
если число задач превышает размер пула, то новые задачи ставятся в очередь

Executors.newSingleThreadPool()

создает пул с одним потоком, который последовательно выполняет задачи

Executors

Executors.newSheduledThreadPool()

Executors.newSingleThreadSheduledExecutor()

создает пулы, запускающие задачи по графику

график может предполагать запуск задачи через заданный интервал времени или периодическое выполнение задачи

Executors

Все методы возвращают экземпляр класса ThreadPoolExecutor, реализующий интерфейс ExecutorService

Передача объектов Runnable или Callable для их исполнения потоками в пуле осуществляется методом submit ()

Для завершения работы пула используется метод shutdown()

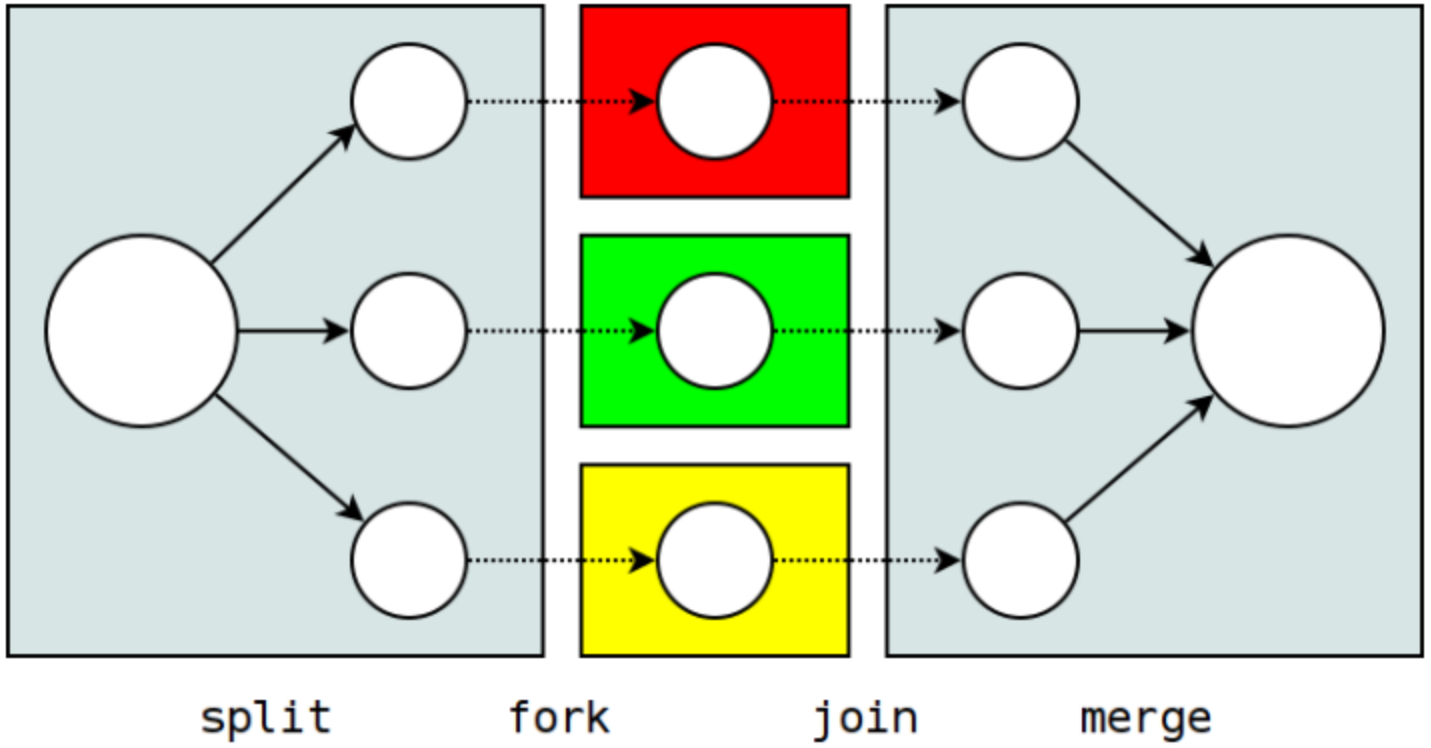
Метод invokeAll() позволяет передать коллекцию задач (возможно с установкой времени ожидания). По завершении всех задач получается список будущих действий с их статусом (завершена, прервана, выбросила исключение)

Метод invokeAny() возвращает результат первой успешно выполненной задачи в коллекции, остальные задачи отменяются

ForkJoinPool

- реализации ExecutorService для упрощения распараллеливания рекурсивных задач
- позволяет запускать ForkJoinTasks в пуле потоков
- В версии Java 8 включили fork/join framework

Классическая схема



```
ForkJoinPool common =                    ForkJoinPool.commonPool();
ForkJoinPool avialable = new ForkJoinPool ();

ForkJoinPool forkJoin2 = new ForkJoinPool(2);
// уровень параллелизма 2
```

Выполняемые задачи

- наследники класса ForkJoinTasks
  - RecursiveAction
  - RecursiveTask

Выполняемые задачи

https://docs.oracle.com/javase/tutorial/↵
essential/concurrency/forkjoin.html

```
if (my portion of the work is small enough)
    do the work directly
else
    split my work into two pieces invoke the two pieces and wait for the results
```

Базовая семантика методов

- fork()
  - Кладёт задачу в очередь, и возвращается
  - Кто-нибудь другой может эту задачу подхватить
- join()
  - Блокируется, пока задача не закончится
  - Но поток терять на этом нельзя!
  - FJP может дать ему что-нибудь выполнить другое

RecursiveTask

Дерево

```
public interface Node {
  Collection<Node> getChildren();
   Long getValue();
}
```

```
public class ValueSumCounter extends RecursiveTask<Long>{
  private final Node node;
@Override
 protected Long compute() {
     long sum = node.getValue();
     List<ValueSumCounter> subTasks = new LinkedList<>();
     for(Node child : node.getChildren()) {
         ValueSumCounter task = new ValueSumCounter(child);
         task.fork(); // запустим асинхронно
         subTasks.add(task);
    }
    for(ValueSumCounter task : subTasks) {
          sum += task.join();
       // дождёмся выполнения задачи и прибавим результат
   }
   return sum; }
```

```
public static void main(String[] args) {
    Node root = getRootNode();
     new ForkJoinPool().invoke(new ValueSumCounter(root));
}
```

Параллельные операции с массивами

Java 8

Arrays.parallelSort()

Arrays.parallelSetAll()

Arrays.parallelPrefix()