Обзор java.util.concurrent.\*

[Блог компании Luxoft](https://habr.com/ru/company/luxoft/blog/)[Программирование \*](https://habr.com/ru/hub/programming/)[Java \*](https://habr.com/ru/hub/java/)

Tutorial

Работать с потоками напрямую через создание одиночных *thread-ов* неудобно и чревато ошибками. Поэтому в 2004 году в Java 5 добавили Concurrency API. Он находится в пакете java.util.concurrent и содержит большое количество полезных классов и методов для многопоточного программирования.

В повседневной работе не так уж часто приходится сталкиваться с пакетом для многопоточности java.util.concurrent. Иногда существуют проектные ограничения по использованию java 1.4.2, где нет данного пакета, но чаще всего хватает обычной синхронизации и не требуется ничего сверхъестественного. К счастью, периодически возникают задачи, заставляющие немного пораскинуть мозгами и либо написать велосипед, либо порыться в javadoc'ах и найти что-то более подходящее. С велосипедом проблем нет — просто берешь и пишешь, благо ничего суперсложного в многопоточности нет. С другой стороны, меньше кода — меньше багов. Тем более, что на многопоточность никто в здравом уме юнит тестов не пишет, т.к. это уже полноценные интеграционные тесты получаются со всеми вытекающими последствиями.

Что выбрать для конкретного случая? В условиях запарки и deadline'ов довольно сложно охватить весь java.util.concurrent. Выбирается что то похожее и вперед! Так, постепенно, в коде появляются ArrayBlockingQueue, ConcurrentHashMap, AtomicInteger, Collections.synchronizedList(new LinkedList()) и другие интересности. Иногда правильно, иногда нет. В какой то момент времени начинаешь осознавать, что более 95% стандартных классов в java вообще не используются при разработке продукта. Коллекции, примитивы, перекладывание байтиков с одного места на другое, hibernate, spring или EJB, еще какая то библиотека и, вуаля, приложение готово.

Чтобы хоть как то упорядочить знания и облегчить вхождение в тему, ниже идет обзор классов для работы с многопоточностью. Пишу прежде всего как шпаргалку для себя. А если еще кому сгодится — вообще замечательно.

Для затравки

Сразу приведу пару интересных ссылок. Первая для тех, кто немного плавает в многопоточности. Вторая для «продвинутых» программеров — возможно тут найдётся что-нибудь полезное.

* [Java Concurrency Tutorials](http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html)
* [Concurrent Programming in Java by Doug Lea](http://www.cs.umd.edu/class/fall2002/cmsc433-0201/lectures/cpjslides.pdf)

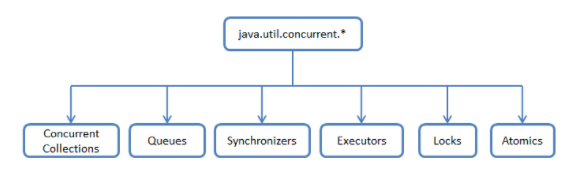
Немного об авторе пакета java.util.concurrent

Если кто хоть когда-нибудь открывал исходники классов java.util.concurrent, не могли не заметить в авторах Doug Lea (Даг Ли), профессора Oswego (Осуиго) университета штата Нью Йорк. В список наиболее известных его разработок попали java [collections](http://gee.cs.oswego.edu/dl/classes/collections/index.html) и [util.concurrent](http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html), которые в том или ином виде отразились в существующих JDK. Также им была написана [dlmalloc](http://en.wikipedia.org/wiki/Malloc" \l "dlmalloc) имплементация для динамического выделения памяти. Среди литературы отметилась книга по многопоточности [Concurrent Programming in Java: Design Principles and Pattern, 2nd Edition](http://www.informit.com/store/concurrent-programming-in-java-design-principles-and-9780201310092). Более подробно можно ознакомиться на его [домашней страничке](http://g.oswego.edu/).

Выступление Doug Lea на [JVM Language Summit](http://wiki.jvmlangsummit.com/2010_Main_Page) в 2010 году.

По вершкам

Наверное у многих возникало чувство некоторого хаоса при беглом взгляде на java.util.concurrent. В одном пакете намешаны разные классы с совершенно разным функционалом, что несколько затрудняет понимание что к чему относится и как это работает. Поэтому, можно схематично поделить классы и интерфейсы по функциональному признаку, а затем пробежаться по реализации конкретных частей.

  
  
**Concurrent Collections** — набор коллекций, более эффективно работающие в многопоточной среде нежели стандартные универсальные коллекции из java.util пакета. Вместо базового враппера Collections.synchronizedList с блокированием доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных по [wait-free](http://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B0%D1%8F_%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) алгоритмам.

**Queues** — неблокирующие и блокирующие очереди с поддержкой многопоточности. Неблокирующие очереди заточены на скорость и работу без блокирования потоков. Блокирующие очереди используются, когда нужно «притормозить» потоки «Producer» или «Consumer», если не выполнены какие-либо условия, например, очередь пуста или перепонена, или же нет свободного «Consumer»'a.

**Synchronizers** — вспомогательные утилиты для синхронизации потоков. Представляют собой мощное оружие в «параллельных» вычислениях.

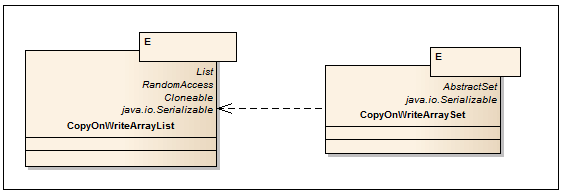
**Executors** — содержит в себе отличные фрейморки для создания пулов потоков, планирования работы асинхронных задач с получением результатов.

**Locks** — представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми synchronized, wait, notify, notifyAll.

**Atomics** — классы с поддержкой атомарных операций над примитивами и ссылками.

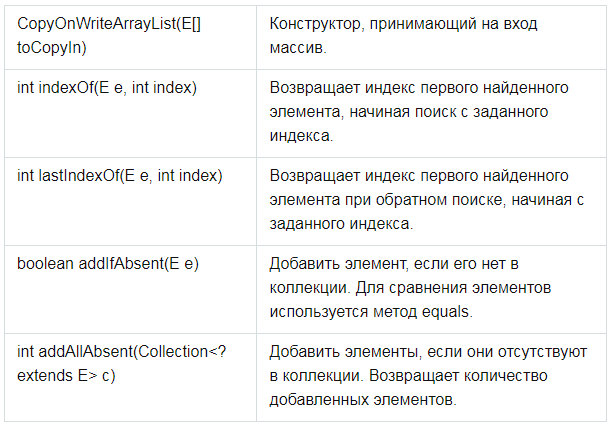
1. Concurrent Collections

**CopyOnWrite коллекции**



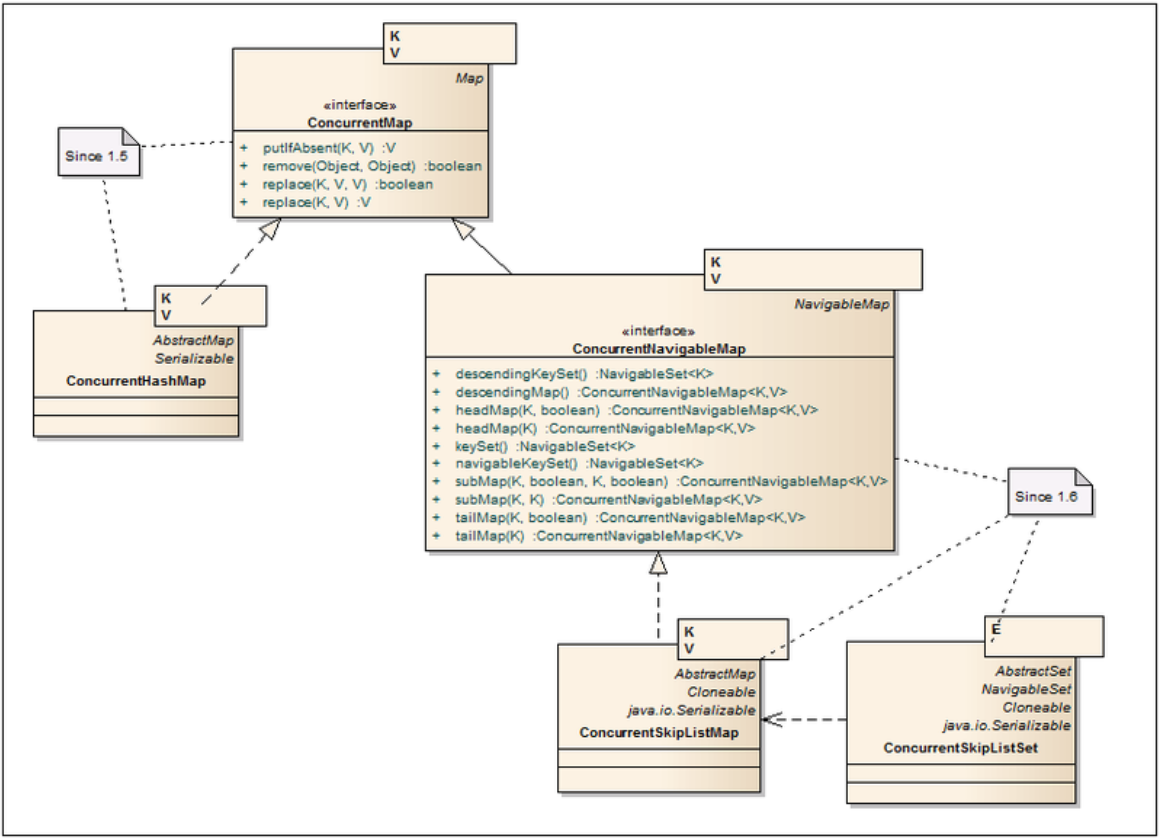
Название говорит само за себя. Все операции по изменению коллекции (add, set, remove) приводят к созданию новой копии внутреннего массива. Тем самым гарантируется, что при проходе итератором по коллекции не кинется ConcurrentModificationException. Следует помнить, что при копировании массива копируются только референсы (ссылки) на объекты (shallow copy), т.ч. доступ к полям элементов не thread-safe. CopyOnWrite коллекции удобно использовать, когда write операции довольно редки, например при реализации механизма подписки listeners и прохода по ним.  
  
**CopyOnWriteArrayList<E>** — Потокобезопасный аналог ArrayList, реализованный с CopyOnWrite алгоритмом.

**Дополнительные методы и конструктор**



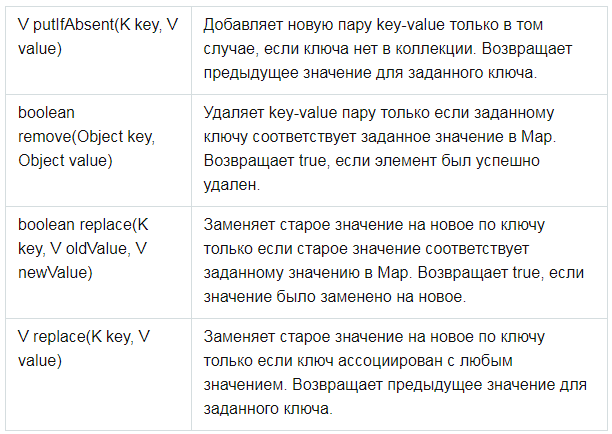
**CopyOnWriteArraySet<E>**  — Имплементация интерфейса Set, использующая за основу CopyOnWriteArrayList. В отличии от CopyOnWriteArrayList, дополнительных методов нет.

Scalable Maps

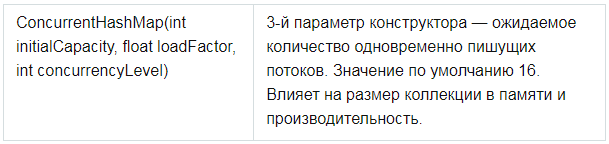


Улучшенные реализации HashMap, TreeMap с лучшей поддержкой многопоточности и масштабируемости.  
  
**ConcurrentMap<K, V>** — Интерфейс, расширяющий Map несколькими дополнительными атомарными операциями.

**Дополнительные методы**

  
**ConcurrentHashMap<K, V>** — В отличие от Hashtable и блоков synhronized на HashMap, данные представлены в виде сегментов, разбитых по hash'ам ключей. В результате, для доступ к данным лочится по сегментам, а не по одному объекту. В дополнение, итераторы представляют данные на определенный срез времени и не кидают ConcurrentModificationException. Более детально ConcurrentHashMap описан в хабратопике [тут](http://habrahabr.ru/post/132884/).

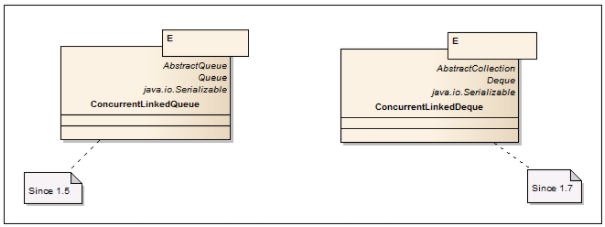
**Дополнительный конструктор**



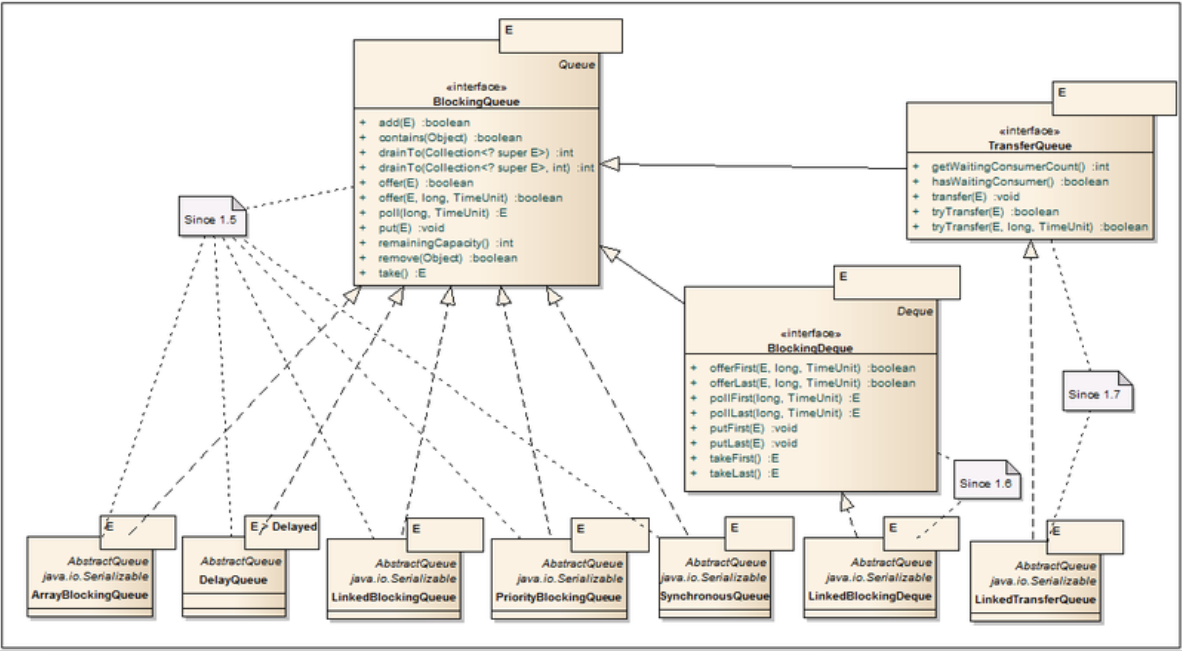
**ConcurrentNavigableMap<K,V>** — Расширяет интерфейс NavigableMap и вынуждает использовать ConcurrentNavigableMap объекты в качестве возвращаемых значений. Все итераторы декларируются как безопасные к использованию и не кидают ConcurrentModificationException.  
  
**ConcurrentSkipListMap<K, V>** — Является аналогом TreeMap с поддержкой многопоточности. Данные также сортируются по ключу и гарантируется усредненная производительность log(N) для containsKey, get, put, remove и других похожих операций. Алгоритм работы SkipList описан на [Wiki](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D1%81_%D0%BF%D1%80%D0%BE%D0%BF%D1%83%D1%81%D0%BA%D0%B0%D0%BC%D0%B8) и [хабре](http://habrahabr.ru/post/139870/).  
  
**ConcurrentSkipListSet<E>** — Имплементация Set интерфейса, выполненная на основе ConcurrentSkipListMap.

2. Queues

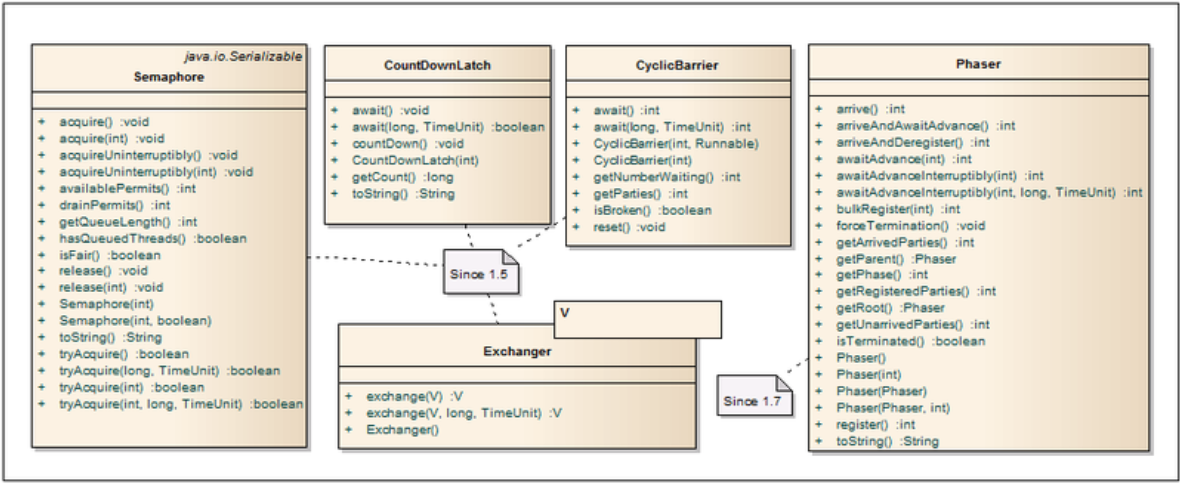
**Non-Blocking Queues**

  
  
Потокобезопасные и неблокирующие имплементации Queue на связанных нодах (linked nodes).  
  
**ConcurrentLinkedQueue<E>** — В имплементации используется wait-free алгоритм от Michael & Scott, адаптированный для работы с garbage collector'ом. Этот алгоритм довольно эффективен и, что самое важное, очень быстр, т.к. построен на [CAS](http://en.wikipedia.org/wiki/Compare-and-swap). Метод size() может работать долго, т.ч. лучше постоянно его не дергать. Детальное описание алгоритма можно посмотреть тут [тут](http://www.cs.rochester.edu/u/michael/PODC96.html).  
  
**ConcurrentLinkedDeque<E>** — Deque расшифровывается как Double ended queue и читается как «Deck». Это означает, что данные можно добавлять и вытаскивать с обоих сторон. Соответственно, класс поддерживает оба режима работы: FIFO (First In First Out) и LIFO (Last In First Out). На практике, ConcurrentLinkedDeque стоит использовать только, если обязательно нужно LIFO, т.к. за счет двунаправленности нод данный класс проигрывает по производительности на 40% по сравнению с ConcurrentLinkedQueue.

**Blocking Queues**

  
  
**BlockingQueue<E>** — При обработке больших потоков данных через очереди становится явно недостаточно использования ConcurrentLinkedQueue. Если потоки, разгребающие очередь перестанут справляться с наплывом данных, то можно довольно быстро схлопотать out of memory или перегрузить IO/Net настолько, что производительность упадет в разы пока не настанет отказ системы по таймаутам или из за отсутствия свободных дескрипторов в системе. Для таких случаев нужна queue с возможностью задать размер очереди или с блокировками по условиям. Тут то и появляется интерфейс BlockingQueue, открывающий дорогу к целому набору полезных классов. Помимо возможности задавать размер queue, добавились новые методы, которые реагируют по-разному на незаполнение или переполнение queue. Так, например, при добавлении элемента в переполненную queue, один метод кинет IllegalStateException, другой вернет false, третий заблокирует поток, пока не появится место, четвертый же заблокирует поток с таймаутом и вернет false, если место так и не появится. Также стоит отметить, что блокирующие очереди не поддерживают null значения, т.к. это значение используется в методе poll как индикатор таймаута.  
  
**ArrayBlockingQueue<E>** — Класс блокирующей очереди, построенный на классическом кольцевом буфере. Помимо размера очереди, доступна возможность управлять «честностью» блокировок. Если fair=false (по умолчанию), то очередность работы потоков не гарантируется. Более подробно о «честности» можно посмотреть в описании ReentrantLock'a.  
  
**DelayQueue<E extends Delayed>** — Довольно специфичный класс, который позволяет вытаскивать элементы из очереди только по прошествии некоторой задержки, определенной в каждом элементе через метод getDelay интерфейса Delayed.  
  
**LinkedBlockingQueue<E>** — Блокирующая очередь на связанных нодах, реализованная на «two lock queue» алгоритме: один лок на добавление, другой на вытаскивание элемента. За счет двух локов, по сравнению с ArrayBlockingQueue, данный класс показывает более высокую производительность, но и расход памяти у него выше. Размер очереди задается через конструктор и по умолчанию равен Integer.MAX\_VALUE.  
  
**PriorityBlockingQueue<E>** — Является многопоточной оберткой над PriorityQueue. При вставлении элемента в очередь, его порядок определяется в соответствии с логикой Comparator'а или имплементации Comparable интерфейса у элементов. Первым из очереди выходит самый наименьший элемент.  
  
**SynchronousQueue<E>** — Эта очередь работает по принципу один вошел, один вышел. Каждая операция вставки блокирует «Producer» поток до тех пор, пока «Consumer» поток не вытащит элемент из очереди и наоборот, «Consumer» будет ждать пока «Producer» не вставит элемент.  
  
**BlockingDeque<E>** — Интерфейс, описывающий дополнительные методы для двунаправленной блокирующей очереди. Данные можно вставлять и вытаскивать с двух сторон очереди.  
  
**LinkedBlockingDeque<E>** — Двунаправленная блокирующая очередь на связанных нодах, реализованная как простой двунаправленный список с одним локом. Размер очереди задается через конструктор и по умолчанию равен Integer.MAX\_VALUE.  
  
**TransferQueue<E>** — Данный интерфейс может быть интересен тем, что при добавлении элемента в очередь существует возможность заблокировать вставляющий «Producer» поток до тех пор, пока другой поток «Consumer» не вытащит элемент из очереди. Блокировка может быть как с таймаутом, так и вовсе может быть заменена проверкой на наличие ожидающих «Consumer»ов. Тем самым появляется возможность реализации механизма передачи сообщений с поддержкой как синхронных, так и асинхронных сообщений.  
  
**LinkedTransferQueue<E>** — Реализация TransferQueue на основе алгоритма Dual Queues with Slack. Активно использует [CAS](http://en.wikipedia.org/wiki/Compare-and-swap) и [парковку](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7-b147/sun/misc/Unsafe.java#Unsafe.park%28boolean%2Clong%29) потоков, когда они находятся в режиме ожидания.

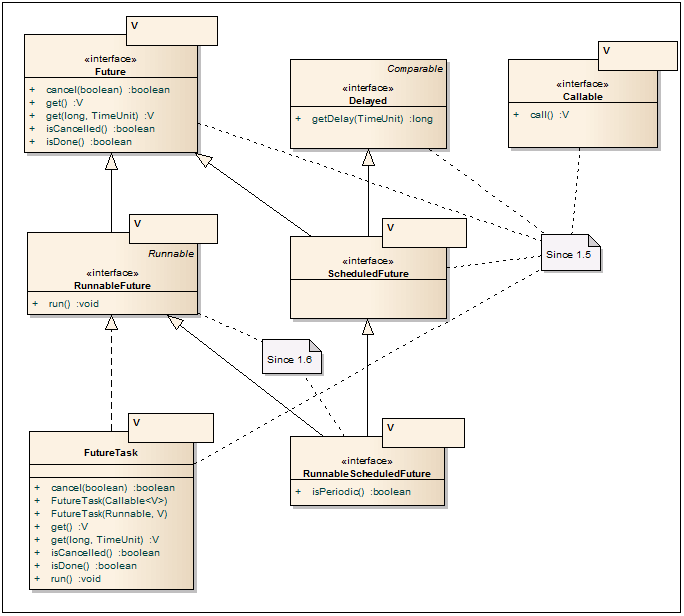
3. Synchronizers

  
  
В данном разделе представлены классы для активного управления потоков.  
  
**Semaphore** — [Семафоры](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) чаще всего используются для ограничения количества потоков при работе с аппаратными ресурсами или файловой системой. Доступ к общему ресурсу управляется с помощью счетчика. Если он больше нуля, то доступ разрешается, а значение счетчика уменьшается. Если счетчик равен нулю, то текущий поток блокируется, пока другой поток не освободит ресурс. Количество разрешений и «честность» освобождения потоков задается через конструктор. Узким местом при использовании семафоров является задание количества разрешений, т.к. зачастую это число приходится подбирать в зависимости от мощности «железа».  
  
**CountDownLatch** — Позволяет одному или нескольким потокам ожидать до тех пор, пока не завершится определенное количество операций, выполняющих в других потоках. Классический пример с драйвером довольно неплохо описывает логику класса: Потоки, вызывающие драйвер, будут висеть в методе await (с таймаутом или без), пока поток с драйвером не выполнит инициализацию с последующим вызовом метода countDown. Этот метод уменьшает счетчик count down на единицу. Как только счетчик становится равным нулю, все ожидающие потоки в await продолжат свою работу, а все последующие вызовы await будут проходить без ожиданий. Счетчик count down одноразовый и не может быть сброшен в первоначальное состояние.  
  
**CyclicBarrier** — Может использоваться для синхронизации заданного количества потоков в одной точке. Барьер достигается когда N-потоков вызовут метод await(...) и заблокируются. После чего счетчик сбрасывается в исходное значение, а ожидающие потоки освобождаются. Дополнительно, если нужно, существует возможность запуска специального кода до разблокировки потоков и сброса счетчика. Для этого через конструктор передается объект с реализацией Runnable интерфейса.  
  
**Exchanger<V>** — Как видно из названия, основное предназначение данного класса — это обмен объектами между двумя потоками. При этом, также поддерживаются null значения, что позволяет использовать данный класс для передачи только одного объекта или же просто как синхронизатор двух потоков. Первый поток, который вызывает метод exchange(...) заблокируется до тех пор, пока тот же метод не вызовет второй поток. Как только это произойдет, потоки обменяются значениями и продолжат свою работу.  
  
**Phaser** — Улучшенная реализация барьера для синхронизации потоков, которая совмещает в себе функционал CyclicBarrier и CountDownLatch, вбирая в себя самое лучшее из них. Так, количество потоков жестко не задано и может динамически меняться. Класс может повторно переиспользоваться и сообщать о готовности потока без его блокировки. Более подробно можно почитать в хабратопике [тут](http://habrahabr.ru/post/117185/).

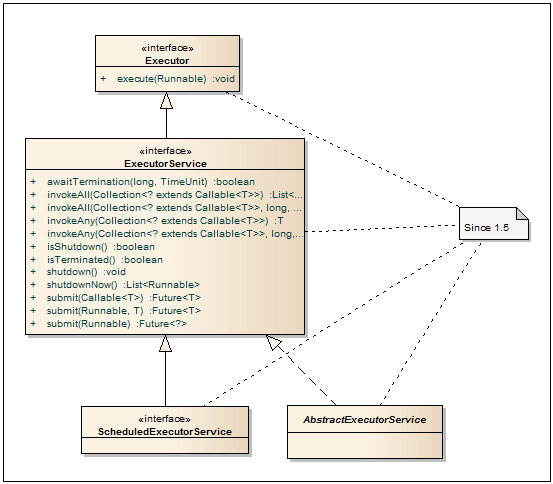
4. Executors

Вот мы и подобрались к самой большой части пакета. Здесь будут описаны интерфейсы для запуска асинхронных задач с возможностью получения результатов через Future и Callable интерфейсы, а также сервисы и фабрики для создания thread pools: ThreadPoolExecutor, ScheduledPoolExecutor, ForkJoinPool. Для лучшего понимания, сделаем небольшую декомпозицию интерфейсов и классов.

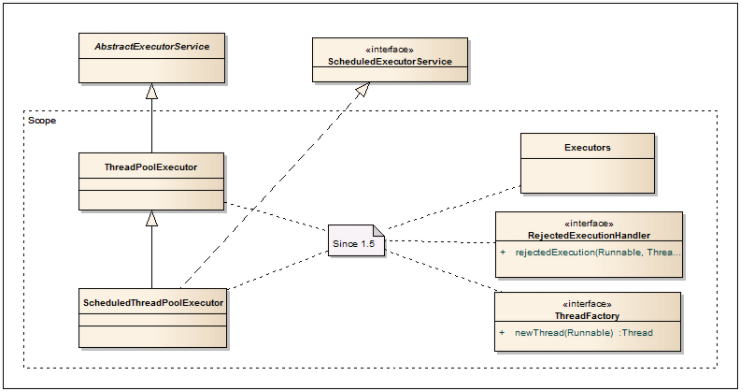
**Future and Callable**

  
  
**Future<V>** — Замечательный интерфейс для получения результатов работы асинхронной операции. Ключевым методом здесь является метод get, который блокирует текущий поток (с таймаутом или без) до завершения работы асинхронной операции в другом потоке. Также, дополнительно существуют методы для отмены операции и проверки текущего статуса. В качестве имплементации часто используется класс FutureTask.  
  
**RunnableFuture<V>** — Если Future — это интерфейс для Client API, то интерфейс RunnableFuture уже используется для запуска асинхронной части. Успешное завершение метода run() завершает асинхронную операцию и позволяет вытаскивать результаты через метод get.  
  
**Callable<V>** — Расширенный аналог интерфейса Runnable для асинхронных операций. Позволяет возвращать типизированное значение и кидать checked exception. Несмотря на то, что в этом интерфейсе отсутсвует метод run(), многие классы java.util.concurrent поддерживают его наряду с Runnable.  
  
**FutureTask<V>** — Имплементация интерфейса Future/RunnableFuture. Асинхронная операция принимается на вход одного из конструкторов в виде Runnable или Callable объектов. Сам же класс FutureTask предназначен для запуска в worker потоке, например через new Thread(task).start(), или через ThreadPoolExecutor. Результаты работы асинхронной операции вытаскиваются через метод get(...).  
  
**Delayed** — Используется для асинхронных задач, которые должны начаться в будущем, а также в DelayQueue. Позволяет задавать время до начала асинхронной операции.  
  
**ScheduledFuture<V>** — Маркерный интерфейс, объединяющий Future и Delayed интерфейсы.  
  
**RunnableScheduledFuture<V>** — Интерфейс, объединяющий RunnableFuture и ScheduledFuture. Дополнительно можно указывать является ли задача одноразовой или же должна запускаться с заданной периодичностью.

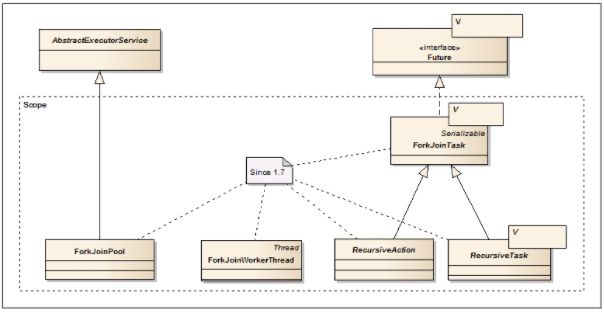
**Executor Services**

  
  
**Executor** — Представляет собой базовый интерфейс для классов, реализующих запуск Runnable задач. Тем самым обеспечивается развязка между добавлением задачи и способом её запуска.  
  
**ExecutorService** — Интерфейс, который описывает сервис для запуска Runnable или Callable задач. Методы submit на вход принимают задачу в виде Callable или Runnable, а в качестве возвращаемого значения идет Future, через который можно получить результат. Методы invokeAll работают со списками задач с блокировкой потока до завершения всех задач в переданном списке или до истечения заданного таймаута. Методы invokeAny блокируют вызывающий поток до завершения любой из переданных задач. В дополнении ко всему, интерфейс содержит методы для graceful shutdown. После вызова метода shutdown, данный сервис больше не будет принимать задачи, кидая RejectedExecutionException при попытке закинуть задачу в сервис.  
  
**ScheduledExecutorService** — В дополнении к методам ExecutorService, данный интерфейс добавляет возможность запускать отложенные задачи.  
  
**AbstractExecutorService** — Абстрактный класс для построения ExecutorService'a. Имплементация содержит базовую имплементацию методов submit, invokeAll, invokeAny. От этого класса наследуются ThreadPoolExecutor, ScheduledThreadPoolExecutor и ForkJoinPool.

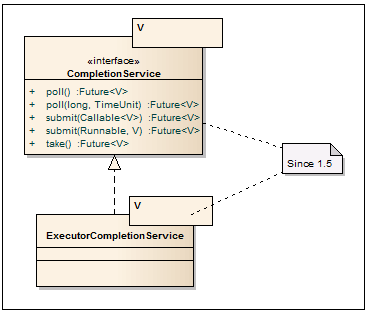
**ThreadPoolExecutor & Factory**

  
  
**Executors** — Класс-фабрика для создания ThreadPoolExecutor, ScheduledThreadPoolExecutor. Если нужно создать один из этих пулов, эта фабрика именно то, что нужно. Также, тут содержатся разные адаптеры Runnable-Callable, PrivilegedAction-Callable, PrivilegedExceptionAction-Callable и другие.  
  
**ThreadPoolExecutor** — Очень мощный и важный класс. Используется для запуска асинхронных задач в пуле потоков. Тем самым практически полностью отсутствует оверхэд на поднятие и остановку потоков. А за счет фиксируемого максимума потоков в пуле обеспечивается прогнозируемая производительность приложения. Как было ранее сказано, создавать данный пул предпочтительно через один из методов фабрики Executors. Если же стандартных конфигураций будет недостаточно, то через конструкторы или сеттеры можно задать все основые параметры пула. Более подробно можно ознакомиться в этом [топике](http://habrahabr.ru/post/116363/).  
  
**ScheduledThreadPoolExecutor** — В дополнении к методам ThreadPoolExecutor, позволяет запускать задачи после определенной задержки, а также с некоторой периодичностью, что позволяет реализовать на базе этого класса Timer Service.  
  
**ThreadFactory** — По умолчанию, ThreadPoolExecutor использует стандартную фабрику потоков, получаемую через Executors.defaultThreadFactory(). Если нужно что-то больше, например задание приоритета или имени потока, то можно создать класс с реализацией этого интерфейса и передать его в ThreadPoolExecutor.  
  
**RejectedExecutionHandler** — Позволяет определить обработчик для задач, которые по каким то причинам не могут быть выполнены через ThreadPoolExecutor. Такой случай может произойти, когда нет свободных потоков или сервис выключается или выключен (shutdown). Несколько стандартных имплементаций находятся в классе ThreadPoolExecutor: CallerRunsPolicy — запускает задачу в вызывающем потоке; AbortPolicy — кидает эксцепшен; DiscardPolicy — игнорирует задачу; DiscardOldestPolicy — удаляет самую старую незапущенную задачу из очереди, затем пытается добавить новую задачу еще раз.

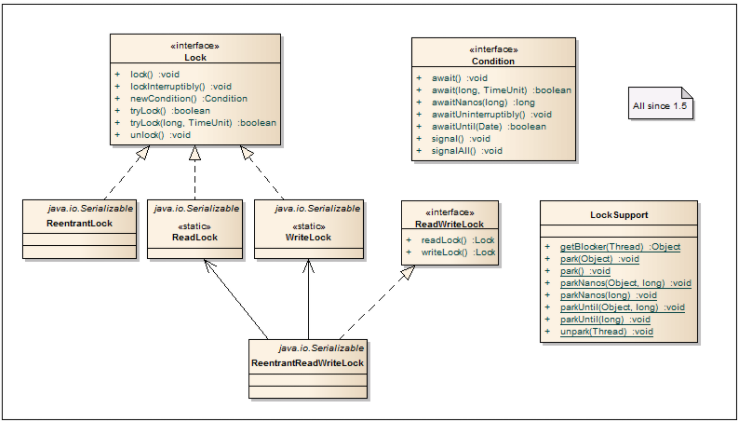
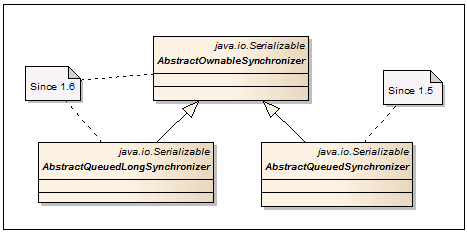
**Fork Join**

  
  
В java 1.7 появился новый Fork Join фреймворк для решения рекурсивных задач, работающих по алгоритмам [разделяй и влавствуй](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm) или [Map Reduce](http://ru.wikipedia.org/wiki/MapReduce). Чтобы было более наглядней, можно привести визуальный пример алгоритма сортировки quicksort:  
  
Так, за счет разбиения данных строки/столбца на части, можно добиться их параллельной обработки в разных потоках. Для решения этой задачи можно использовать и обычный ThreadPoolExecutor, но за счет частого переключения контекста и отслеживания контроля исполнения все это не очень эффективно работает. Тут то нам приходит на помощь Fork Join framework в основу которого используется [work-stealing](http://classes.engineering.wustl.edu/cse566s/presentations/WorkStealing.pdf) алгоритм. Наиболее хорошо раскрывает себя в системах с большим количеством процессоров. Подробнее можно ознакомиться в блоге [тут](http://www.igvita.com/2012/02/29/work-stealing-and-recursive-partitioning-with-fork-join/) или [публикации](http://gee.cs.oswego.edu/dl/papers/fj.pdf) Doug Lea. Про производительность и масштабируемость можно почитать [тут](http://letitcrash.com/post/17607272336/scalability-of-fork-join-pool).  
  
**ForkJoinPool** — Представляет собой точку входа для запуска корневых (main) ForkJoinTask задач. Подзадачи запускаются через методы задачи, от которой нужно отстрелиться (fork). По умолчанию создается пул потоков с количеством потоков равным количеству доступных для JVM процессоров (cores).  
  
**ForkJoinTask** — Базовый класс для всех Fork Join задач. Из ключевых методов можно отметить: fork() — добавляет задачу в очередь текущего потока ForkJoinWorkerThread для асинхронного выполнения; invoke() — запускает задачу в текущем потоке; join() — ожидает завершения подзадачи с возвращением результата; invokeAll(...) — объединяет все три предыдущие предыдущие операции, выполняя две или более задач за один заход; adapt(...) — создает новую задачу ForkJoinTask из Runnable или Callable объектов.  
  
**RecursiveTask** — Абстрактный класс от ForkJoinTask, с объявлением метода compute, в котором должна производиться асинхронная операция в наследнике.  
  
**RecursiveAction** — Отличается от RecursiveTask тем, не возвращает результат.  
  
**ForkJoinWorkerThread** — Используется в качестве имплементации по умолчанию в ForkJoinPoll. При желании можно отнаследоваться и перегрузить методы инициализации и завершения worker потока.

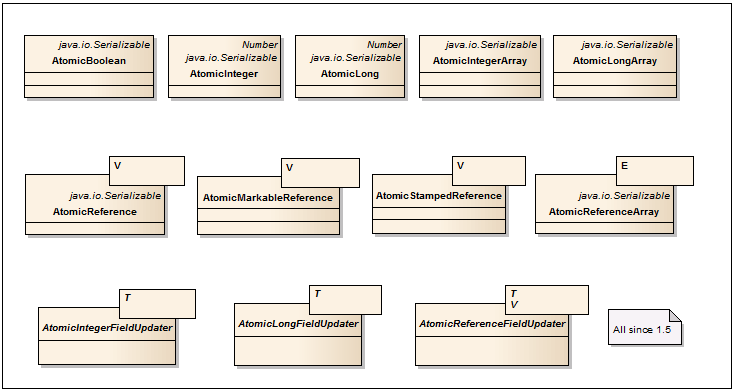
**Completion Service**

  
  
**CompletionService** — Интерфейс сервиса с развязкой запуска асинхронных задач и получением результатов. Так, для добавления задач используются методы submit, а для вытаскивания результатов *завершенных* задач используются блокирующий метод take и неблокирующий poll.  
  
**ExecutorCompletionService** — По сути является враппером над любым классом, реализующим интерфейс Executor, например ThreadPoolExecutor или ForkJoinPool. Используется преимущественно тогда, когда хочется абстрагироваться от способа запуска задач и контроля за их исполнением. Если есть завершенные задачи — вытаскиваем их, если нет — ждем в take пока что-нибудь не завершится. В основе сервиса по умолчанию используется LinkedBlockingQueue, но может быть передана и любая другая имплементация BlockingQueue.

5. Locks

  
  
**Condition** — Интерфейс, который описывает альтернативные методы стандарным wait/notify/notifyAll. Объект с условием чаще всего получается из локов через метод lock.newCondition(). Тем самым можно получить несколько комплектов wait/notify для одного объекта.  
  
**Lock** — Базовый интерфейс из lock framework, предоставляющий более гибкий подход по ограничению доступа к ресурсам/блокам нежели при использовании synchronized. Так, при использовании нескольких локов, порядок их освбождения может быть произвольный. Плюс имеется возможность пойти по альтернативному сценарию, если лок уже кем то захвачен.  
  
**ReentrantLock** — Лок на вхождение. Только один поток может зайти в защищенный блок. Класс поддерживает «честную» (fair) и «нечестную» (non-fair) разблокировку потоков. При «честной» разблокировке соблюдается порядок освобождения потоков, вызывающих lock(). При «нечестной» разблокировке порядок освобождения потоков не гарантируется, но, как бонус, такая разблокировка работает быстрее. По умолчанию, используется «нечестная» разблокировка.  
  
**ReadWriteLock** — Дополнительный интерфейс для создания read/write локов. Такие локи необычайно полезны, когда в системе много операций чтения и мало операций записи.  
  
**ReentrantReadWriteLock** — Очень часто используется в многопоточных сервисах и кешах, показывая очень хороший прирост производительности по сравнению с блоками synchronized. По сути, класс работает в 2-х взаимоисключающих режимах: много reader'ов читают данные в параллель и когда только 1 writer пишет данные.  
  
**ReentrantReadWriteLock.ReadLock** — Read lock для reader'ов, получаемый через readWriteLock.readLock().  
  
**ReentrantReadWriteLock.WriteLock** — Write lock для writer'ов, получаемый через readWriteLock.writeLock().  
  
**LockSupport** — Предназначен для построения классов с локами. Содержит методы для парковки потоков вместо устаревших методов Thread.suspend() и Thread.resume().  
  
  
  
**AbstractOwnableSynchronizer** — Базовый класс для построения механизмов сихнронизации. Содержит всего одну пару геттер/сеттер для запоминания и чтения эксклюзивного потока, который может работать с данными.  
  
**AbstractQueuedSynchronizer** — Используется в качестве базового класса для механизма синхронизации в FutureTask, CountDownLatch, Semaphore, ReentrantLock, ReentrantReadWriteLock. Может применяться при создании новых механизмов синхронизации, полагающихся на одиночное и атомарное значение int.  
  
**AbstractQueuedLongSynchronizer** — Разновидность AbstractQueuedSynchronizer, которая поддерживает атомарное значение long.

6. Atomics

  
  
  
**AtomicBoolean**, **AtomicInteger**, **AtomicLong**, **AtomicIntegerArray**, **AtomicLongArray** — Что если в классе нужно синхронизировать доступ к одной простой переменной типа int? Можно использовать конструкции с synchronized, а при использовании атомарных операций set/get, подойдет также и volatile. Но можно поступить еще лучше, использовав новые классы Atomic\*. За счет использования [CAS](http://en.wikipedia.org/wiki/Compare-and-swap), операции с этими классами работают быстрее, чем если синхронизироваться через synchronized/volatile. Плюс существуют методы для атомарного добавления на заданную величину, а также инкремент/декремент.  
  
**AtomicReference** — Класс для атомарных операцией с ссылкой на объект.  
  
**AtomicMarkableReference** — Класс для атомарных операцией со следующей парой полей: ссылка на объект и битовый флаг (true/false).  
  
**AtomicStampedReference** — Класс для атомарных операцией со следующей парой полей: ссылка на объект и int значение.  
  
**AtomicReferenceArray** — Массив ссылок на объекты, который может атомарно обновляться.  
  
**AtomicIntegerFieldUpdater**, **AtomicLongFieldUpdater**,**AtomicReferenceFieldUpdater** — Классы для атомарного обновления полей по их именам через reflection. Смещение полей для CAS определяется в конструкторе и кешируются, т.ч. тут нет сильного падения производительности из за reflection.

Вместо заключения

Спасибо, что дочитали до конца или, по крайней мере, пролистнули статью в конец. Сразу хочется подчернуть, что здесь находится лишь краткое описание классов без каких либо примеров. Это сделано специально, чтобы не загромождать статью чрезмерными вставками кода. Основная мысль: дать быстрый обзор классов, чтобы знать в каком направлении двигаться и что можно использовать. Примеры использования классов можно легко найти в интернете или в исходниках самих классов. Надеюсь данный пост поможет быстрее и эффективней решать интересные задачи с многопоточностью.