В пакете **java.util.concurrent** определяются следующие средства параллелизма:

* Синхронизаторы
* Исполнители (Executors)
* Параллельные коллекции
* Каркас Fork/Join Framework

**СИНХРОНИЗАТОРЫ**

**Синхронизаторы** предоставляют высокоуровневые способы синхронизации взаимодействия нескольких потоков. Основные из них:

**Semaphore –** реализует классический семафор. Управляет доступом к общему ресурсу с помощью счетчика.

Когда потоку нужен доступ к ресурсу, он проверяет счетчик семафора. Если счетчик больше 0, то поток получит разрешение, а значение счетчика уменьшиться на 1. В противном случае поток будет заблокирован, пока счетчик будет равен 0. Если потоку больше не нужен доступ к ресурсу, он освобождает его, в результате чего счетчик семафора увеличивается на 1.

**Конструторы:**



Первый параметр **значение счетчика**. Второй определяет, **в каком порядке будет предоставляться доступ**. По умолчанию false – доступ предоставляется в неопределенном порядке. Если true – потокам предоставляется доступ в том порядке, в котором они его запрашивали.

**Методы:**

* **sem.acquire() –** запросить одно разрешение (уменьшить счетчик на 1)
* **sem.acquire(int число) –** запросить несколько разрешений
* **sem.release() –** освободить одно разрешение
* **sem.release(int число) –** освободить несколько разрешений

**CountDownLatch –** ожидает до тех пор, пока не произойдет определенное количество событий. Объект этого класса создается с количеством событий, которые должны произойти до того момента, как будет снята самоблокировка.

**Конструктор:**



**Методы:**

* **cdl.await() –** вызывает блокировку потока, пока счетчик CountDownLatch не достигнет нуля.
* **cdl.await(long ожидание, TimeUnit единица\_времени) –** блокировка пока счетчик не достигнет 0, либо пока не истечет таймаут. Возвращает false если достигнут таймаут, либо true если счетчик достиг 0.
* **cdl.countDown() –** метод оповещения о событии. Уменьшает счетчик на 1.

**CyclicBarrier –** позволяет группе потоков исполнения войти в режим ожидания в предварительно заданной точке выполнения. То есть все потоки останавливаются в определенном месте до тех пор, пока все потоки не достигнут этой точки (барьера).

**Конструторы:**



Первый параметр – количество потоков, которые должны достичь барьера. Второй параметр – поток, который будет исполнен по достижении барьера.

Барьер в каждом потоке определяется вызовом метода **await()** объекта CyclicBarrier**.** В итоге исполнение потока будет приостановлено до тех пор , пока **await()** не будет вызван во всех потоках. Если указать второй параметр конструктора, то будет также запущен этот поток.

Этот класс можно использовать повторно. То есть после того как барьер будет достигнут, выполнение продолжиться, а счетчик потоков вернется в начальное значение.

**Методы:**

* **await() –** ожидать, пока указанное число потоков не достигнет барьерной точки
* **await(long ожидание, TimeUnit единица\_времени) –** ожидание будет происходить пока не истечет таймаут. В обеих формах возвращается порядок, в котором потоки будут достигать барьерной точки. Первый поток вернет значение, равное количеству ожидаемых потоков минус 1, а последний поток возвращает нулевое значение.

**Exchanger –** упрощает обмен данными между двумя потоками исполнения.

Он ожидает до тех пор, пока два отдельных потока вызовут его метод **exchange()**. После этого, Exchanger произведет обмен данными, предоставляемыми обоими потоками.

Exchanger является обобщенным, где обобщение указывает тип обмениваемых данных.

**Методы:**

* **exchange(V буфер) –** буфер это ссылка на обмениваемые данные. Возвращаются данные, полученные из другого потока исполнения.
* **exchange(V буфер, long ожидание, TimeUnit единица времени) –** как и первая только с таймаутом.

Главная особенность метода exchange состоит в том, что он не завершается успешно до тех пор, пока не будет вызван для одного и того же объекта Exchanger из двух отдельных потоков.

**Phaser –** синхронизирует потоки исполнения, которые представляют одну или несколько стадий (фаз) выполнения действия.

Обычно сначала создается новый экземпляр класса Phaser. Затем синхронизатор фаз регистрирует одну или несколько сторон, вызывая метод **register()** или указывая нужное количество сторон в конструкторе. Синхронизатор фаз ожидает до тех пор, пока все зарегистрированные стороны не завершат фазу.

Стороны извещают об этом, вызывая например методы **arrive()** или **arriveAndAwaitAdvance().** Как только все стороны достигнут данной фазы, синхронизатор фаз может перейти к следующей фазе (если она имеется) или завершить свою работу.

**Методы:**

* **register() –** зарегистрировать сторону в фазе. Возвращается номер регистрируемой фазы.
* **arrive() –** сторона вызывает его, чтобы сообщить о завершении фазы. Возвращает текущий номер фазы. Если работа синхронизатора фаз завершена, возвращает отрицательное значение. Этот метод не приостанавливает выполнения потока (то есть он не ждет пока другие потоки завершат фазу)
* **arriveAndAwaitAdvance()** – как и arrive(), но поток будет ожидать пока другие потоки завершат фазу.
* **arriveAndDeregister()** – сообщить о завершении фазы и отменить регистрацию в синхронизаторе.

# **EXECUTERS(**ExecutorService)

**Executors (исполнители)** управляют исполнением потоков. На вершине иерархии находится интерфейс **Executor**, предназначенный для запуска потока исполнения.

В Executor определяется метод **execute(Runnable поток).** В результате вызова этого метода запускается указанный поток.

Интерфейс **ExecutorService** расширяет интерфейс Executor и предоставляет методы, управляющие исполнением.

**Методы:**

* **execute(Runnable поток) –** унаследован от Executer, запускает поток и не ожидает его завершения.
* **submit(Callable callable) –** запускает поток или callable на выполнение, и ожидает результата. Возвращает Future.
* **shutdown() -** закрыть пул. Текущие задачи продолжат выполняться, но новых запущено не будет.
* **awaitTermination(timeout, unit) –** подождать завершения потоков после shutdown

Метод **shutdown()** останавливает останавливает все потоки исполнения, находящиеся в данный момент под управлением данного ExecutorService.

Имеет три реализации в классах:

**ThreadPoolExecutor, ScheduledThreadPoolExecutor** и **ForkJoinPool.**

# **EXECUTORS**

**Executors –** служебный класс, содержащий несколько статических методов, упрощающих создание различных исполнителей.

**Методы:**

* **newCachedThreadPool() –** создает пул потоков, который не только вводит потоки в исполнение по мере необходимости, но и по возможности повторно использует их.
* **newFixedThreadPool(int количесвто\_потоков) –** пул потоков фиксированной длины. В конструкторе принимает количество потоков. Когда достигнут лимит, другие потоки не выполняются, а ждут очереди.
* **newScheduleThreadPool(int количесвто\_потоков) –** пул в котором можно осуществлять планирование потоков исполнения. Позволяет запускать потоки с задержкой или периодически.  
  **schedule(Runnable command, long delay, TimeUnit unit)** – запустить поток отложено  
  **scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit) –** запускать поток периодически

# **FUTURE**

Интерфейс **Future<V>** содержит значение, возвращаемое потоком после исполнения. V определяет тип возвращаемого результата. Таким образом это значение определяется “на будущее”, когда поток завершит свое исполнение.

**Методы:**

* **get() –** получить значение из Future. Поток блокируется, пока не завершится соответствующий Callable.
* **get(long ожидание, TimeUnit единица времени) –** аналогичен предыдущему, но ждет только в течении указанного времени
* **cancel() –** отменить выполнение задачи.
* **isCancelled() –** была ли задача отменена.
* **isDone()**  - была ли задача выполнена

**CALLABLE**

Интерфейс **Callable<V>** определяет поток исполнения, возвращающий значение. **V –** обозначает тип данных, возвращаемых потоком исполнения.

В нем определяется единственный метод **call().** Внутри него определяется код, который нужно выполнить. Если результат нельзя вычислить, метод call() генерирует исключение.

Запускается Callable вызовом метода **submit(Callable<T> callable)** определенного в интерфейсе **ExecutorService.** Результат возвращается через объект типа Future.

# **RUNNABLEFUTURE**

Интерфейс **RunnableFuture** наследует как интерфейс Future, так и интерфейс Runnable, и возвращает результат выполнения при успешном выполнении метода run. Класс реализации этого интерфейса – **FutureTask**.

FutureTask может быть выполнен как Runable, а также может получить возвращаемое значение Callable как Future.

# **FUTURETASK**

Таким образом, **FutureTask** представляет удобный механизм для превращения Callable одновременно в Future и Runnable, реализуя оба интерфейса. Объект класса FutureTask может быть передан на выполнение классу, реализующему интерфейс Executor, либо запущен в отдельном потоке, как класс, реализующий интерфейс Runnable.

Можно считать его адаптером старой модели работы с задачами в потоках и новой модели.

# **COMPLETABLEFUTURE**

**CompletableFuture –** класс для асинхронной работы, который дает возможность комбинировать шаги обработки, соединяя их в цепочку. Реализовывает интерфейс CompletionStage и Future. Он расширяет обычный Future и имеет следующие преимущества:

* CompletableFuture можно завершать вручную.
* Future имеет только метод get, поэтому если мы хотим выполнить некоторые действия над результатом мы должны его дождаться. В CompletableFuture мы можем передать результат куда-то еще (даже если он еще не готов), и продолжить выполнение. Так же в CompletableFuture мы можем повесить callback.
* Можно выполнить множество CompletableFuture один за другим.
* Имеет механизм обработки исключений

Для создания можно воспользоваться методом

* CompletableFuture.**supplyAsync(Supplier<U> supplier, Executor executor) –** первый параметр - это действие, второй Executor в котором выполнится. По умолчанию используется общий ForkJoinPool. Этот метод позволяет получить результат.
* CompletableFuture.**runAsync(Runnable runnable, Executor executor) –** запускает runnable (асинхронную задачу) и не возвращает результат. Нам не нужно ждать результата.
* **boolean complete(T value) –** завершить CompletableFuture вручную, записав значение
* **boolean completeExceptionally(Throwable ex) –** записать исключение, тем самым тоже завершив CompletableFuture
* **get() –** получить результат, блокирующий метод
* **thenAccept(Consumer action) –** обработать результат в callback. Не возвращает результата
* **thenRun** **(Runnable action) –** вызвать runnable, так же не возвращает результата и не имеет доступа к результату CompletableFuture.
* **thenApply(Function func) –** передать результат в function. Вызывается в том же потоке где и вызывается. Возвращает новый CompletionStage, который выполняется с результатом этого этапа. Это и есть по сути объединение в цепочку.
* **thenApplyAsync(Function func) –** как и предыдущий, но исполняется в отдельном потоке.
* **anyOf(CompletableFuture<?>... cfs) –** возвращает новый фьючерс, результатом которого будет результат первого завершившегося фьючерса.
* **allOf(CompletableFuture<?>... cfs) –** используется когда есть список задач, и нужно запустить какое-то действие после выполнения всех. Возвращает новый CompletableFuture<Void>.
* **thenCombine(anotherFuture, biFunction) –** по завершению двух задач выполнить третью.
* **thenCompose(function)** – используется когда function также возвращает CompletableFuture, чтобы избежать вложенности объектов CompletableFuture друг в друга.
* **handle() –** метод, позволяющий удобно обрабатывать исключения. Вызывается независимо от того возникли ошибки или нет.
* **Exceptionally() –** обрабатывает ошибки, если они возникли. Ошибка не будет распространяться дальше, и цепочка полностью отработает

Если в одной задаче возникает ошибка, то последующие задачи в цепочки выполнены не будут.

# **ATOMICS**

Средства пакета java.util.concurrent.atomic упрощают присваивание переменным и чтение из них без применения блокировок.

Атомарные операции выполняются с помощью классов **AtomicInteger** и **AtomicLong**.

**Методы этих классов:**

* **get() -** получение
* **set() –** установка
* **compareAndSet() –** сравнение и установка
* **decrementAndGet() –** декремент и получение
* **getAndSet() –** получение и установка

# Работаю по принципу **Compare-and-Swap(CAS).**

Берется старое значение и запоминается, оно называется ожидаемым. Затем на основе ожидаемого рассчитывается новое значение. Ожидаемое значение сравнивается с тем что лежит в памяти, и если они совпадают, то записывается новое значение.

**FORK/JOIN FRAMEWORK**

Набор новых классов и интерфейсов для поддержки параллельного программирования (когда используется несколько процессоров).

Fork/Join Framework упрощает создание и использование нескольких потоков исполнения, и автоматизирует использование нескольких процессоров.

**ForkJoinTask<V>** - абстрактный класс, определяющий выполняемую задачу. V – тип результата выполняемой задачи. Этот класс отличается от Thread тем, что он представляет облегченную абстракцию задачи, а не поток исполнения.

Задача ForkJoinTask выполняется потоками, управляемыми из пула потоков ForkJoinPool. Такой механизм позволяет управлять выполнением большого количества задач, фактически используя небольшое число потоков исполнения.

**Основные методы:**

* **fork()** – передает вызывающую задачу для асинхронного выполнения. Поток исполнения, из которого вызывается метод fork(), продолжает выполняться. Как только задача будет запланирована для выполнения, метод fork() возвратит ссылку this на объект этой задачи.
* **join()** – ожидает завершение задачи, для которой он вызван. Возвращается результат выполнения задачи.
* **invoke() –** запускает задачу на выполнение, а затем ожидает ее завершения.
* **invokeAll() –** позволяет запустить сразу несколько задач. Вызывающий поток ожидает завершения всех задач.
* **cancel(boolean прерывание) –** отменить задачу. Возвращает true если задача успешно отменена. False – если задача уже отменена, завершена или не может быть отменена. В настоящее время параметр прерывание не используется. Как правела вызов cancel() происходит за пределами задачи, поскольку задача может легко отменить себя с помощью return
* **isCancelled() –** проверить, была ли задача отменена.
* **isCompletedNormally() –** проверить была ли задача завершена нормально (без исключений и без cancel()).
* **isCompletedAnnormally() –** была ли задача завершена ненормально.
* **reinitialize() –** устанавливает задачу в исходное состояние до запуска. Но любые изменения внешних объектов не будут отменены.
* **tryUnfork() –** попытаться отменить вызов задачи (исключить ее из плана выполнения)

**RecursiveAction –** Является производным от класса ForkJoinTask<V> для выполнения задач, не возвращающих значения. Расширяется в прикладном коде, чтобы сформировать задачу возвращающую void.

**Методы:**

* **compute() –** абстрактный метод, который переопределяют и описывают в нем код выполняемой задачи.

**RecursiveTask<V> -** Является производным от класса ForkJoinTask<V> для выполнения задач, возвращающих значения.

Так же имеет метод **compute()** определяющий выполняемую задачу.

**ForkJoinPool –** управляет выполнением задач типа ForkJoinTask.

Можно создать его явно, а можно воспользоваться так называемым **общим пулом.**

**Общий пул –** это статический объект типа ForkJoinPool, автоматически доступный для применения.

**Конструкторы:**



Первый конструктор создает пул, обеспечивающий уровень параллелизма, равный количеству процессоров, доступных в системе. Второй конструктор позволяет задать уровень параллелизма. Уровень параллелизма определяет количество потоков, которые могут исполняться одновременно.

На самом деле ForkJoinPool может управлять намного большим количеством задач, чем уровень его параллелизма.

**Методы:**

* **invoke(ForkJoinTask<T> задача) –** запускает указанную задачу, ждет ее завершения и возвращает результат ее выполнения.
* **execute(ForkJoinTask<T> задача) –** запустить задачу на выполнение и не ждать ее завершения.
* **shutdown() –** закрыть пул. Текущие задачи продолжат выполняться, но новых запущено не будет.
* **shutdownNow() –** закрыть пул с попыткой отменить текущие задачи.
* **isShutdown() –** проверить, закрыт ли пул.
* **isTerminated() –** проверить, закрыт ли пул и все ли задачи завершены.

Запустить задачу на выполнение из общего пула можно двумя способами:

1. вызвав статический метод ForkJoinPool.**commonPool()** можно получить ссылку на пул и вызвать у нее метод invoke() или execute().
2. В любой части задачи, кроме вычислительной, можно вызвать метод fork() или invoke() из класса ForkJoinTask.

# FORKJOINTASK `ForkJoinTask` - это абстрактный класс в Java, который представляет задачу, которая может быть разделена на более мелкие подзадачи и выполняться параллельно в рамках пула потоков. Он является частью фреймворка Fork/Join, предоставляемого в пакете `java.util.concurrent`.

`ForkJoinTask` предоставляет два основных подкласса:

1. `RecursiveAction`: представляет задачу, которая не возвращает результат. Она используется, когда задача должна быть разделена на более мелкие подзадачи, которые выполняются параллельно, но результаты не требуются. Метод `compute()` в этом случае не возвращает значение.

2. `RecursiveTask`: представляет задачу, которая возвращает результат. Он используется, когда задача должна быть разделена на более мелкие подзадачи, которые выполняются параллельно, и результаты каждой подзадачи требуются для вычисления окончательного результата. Метод `compute()` возвращает значение результата.

Основные методы `ForkJoinTask`:

- `fork()`: этот метод используется для разделения задачи на более мелкие подзадачи. Он помещает подзадачи в очередь для выполнения в пуле потоков.

- `join()`: этот метод используется для ожидания завершения подзадачи и получения ее результата. Он блокирует текущий поток до тех пор, пока подзадача не завершится и не вернет результат.

- `invoke()`: этот метод используется для запуска задачи и ожидания ее завершения. Он эквивалентен вызову метода `fork()` и затем `join()` для той же задачи.

`ForkJoinTask` также предоставляет методы для работы с результатами и управления выполнением задачи:

- `isDone()`: этот метод возвращает `true`, если задача завершена, и `false` в противном случае.

- `cancel()`: этот метод пытается отменить выполнение задачи. Если задача еще не началась или уже завершилась, отмена будет успешной.

- `getException()`: этот метод возвращает исключение, которое было выброшено во время выполнения задачи, если такое было.

- `complete()`: этот метод ручным образом завершает выполнение задачи и устанавливает ее результат.

После создания объекта `ForkJoinTask` и установки его свойств, вы можете передать его в `ForkJoinPool` для выполнения. `ForkJoinPool` автоматически управляет выполнением задачи в рамках своего пула потоков.

Для выполнения задачи с использованием `ForkJoinPool`, вы можете использовать методы `invoke()` или `fork()` в сочетании с `join()`.

Метод `invoke()` запускает задачу и блокирует текущий поток, ожидая ее завершения и возвращая результат. Например:

java

ForkJoinPool forkJoinPool = new ForkJoinPool();

MyTask task = new MyTask();

result = forkJoinPool.invoke(task);

Метод `fork()` разделяет задачу на более мелкие подзадачи и помещает их в очередь для выполнения в пуле потоков. Затем можно использовать метод `join()` для ожидания завершения подзадачи и получения ее результата. Например:

java

ForkJoinPool forkJoinPool = new ForkJoinPool();

MyTask task = new MyTask();

task.fork();

result = task.join();

Обратите внимание, что метод `fork()` может быть вызван только изнутри метода `compute()` в подклассах `RecursiveAction` и `RecursiveTask`.

При разделении задачи на более мелкие подзадачи, обычно используется условие выхода из рекурсии, когда задача достигает некоторого минимального размера, который может быть выполнен последовательно. Это позволяет избежать избыточной параллелизации.

Важно отметить, что использование `ForkJoinTask` и `ForkJoinPool` имеет смысл только для задач, которые могут быть разделены на более мелкие подзадачи и выполняются параллельно. В противном случае, для простых задач, `ExecutorService` и `Runnable` могут быть более подходящими вариантами.