Main-Components-Utilities-Concurrent-Package

Executor

`Executor` is an interface in the `java.util.concurrent` package that provides a framework for asynchronous execution of tasks in Java. It abstracts away the details of thread creation, management, and scheduling, allowing developers to focus on the task logic rather than low-level concurrency details.

The `Executor` interface defines a single method, `execute(Runnable command)`, that takes a `Runnable` object as input and executes it asynchronously on a new thread or an existing thread from a thread pool, depending on the implementation of the `Executor`.

ExecutorService

`ExecutorService` is a sub-interface of the `Executor` interface in the `java.util.concurrent` package. It extends the `Executor` interface by providing additional methods to manage and control the execution of tasks submitted to an `Executor`.

The `ExecutorService` interface defines methods for submitting tasks, querying the status of running tasks, and controlling the execution of tasks. It allows you to create a pool of threads to execute tasks asynchronously and provides methods to manage the pool, such as setting the maximum number of threads, specifying a queue for tasks that can't be immediately executed, and shutting down the executor when no longer needed.

The `ExecutorService` interface also provides a mechanism for returning the result of a task after it completes execution. The `submit()` method can be used to submit a `Callable` object that returns a result. The `submit()` method returns a `Future` object that can be used to retrieve the result of the `Callable` object once it has finished execution.

ScheduledExecutorService

`ScheduledExecutorService` is a sub-interface of the `ExecutorService` interface, It extends the functionality of `ExecutorService` by providing methods for scheduling the execution of tasks to run after a delay or to run periodically at a fixed rate or delay.
The `ScheduledExecutorService` interface provides several methods for scheduling tasks, including `schedule()`, `scheduleAtFixedRate()`, and `scheduleWithFixedDelay()`. These methods take a `Runnable` or `Callable` object as input and schedule it to run after a specified delay or at a fixed rate or delay.

The `schedule()` method schedules the execution of a task after a specified delay. The `scheduleAtFixedRate()` method schedules a task to run periodically at a fixed rate, while the `scheduleWithFixedDelay()` method schedules a task to run periodically with a fixed delay between the end of the previous execution and the start of the next execution.

The `ScheduledExecutorService` interface also provides methods for canceling scheduled tasks and shutting down the executor when no longer needed.

CountDownLatch

`CountDownLatch` is a synchronization mechanism provided in the `java.util.concurrent` package that allows one or more threads to wait until a set of operations being performed in other threads completes.

The `CountDownLatch` is initialized with a count of the number of events that must occur before the waiting thread or threads can proceed. Each time an event occurs, the count is decremented. When the count reaches zero, the waiting threads are released.

The `CountDownLatch` provides two main methods:

1. `countDown()`: This method decrements the count of the `CountDownLatch`. It is called by the threads performing the events to signal that an event has occurred.
2. `await()`: This method waits until the count of the `CountDownLatch` reaches zero. It is called by the threads that need to wait for the events to complete.

CyclicBarrier

`CyclicBarrier` is a synchronization mechanism provided in the `java.util.concurrent` package that allows a group of threads to wait for each other to reach a common barrier point before continuing execution.

The `CyclicBarrier` is initialized with a count of the number of threads that must reach the barrier before the waiting threads can proceed. Each thread that reaches the barrier awaits the other threads. When the required number of threads has arrived, the threads are released and can continue execution.

The `CyclicBarrier` provides two main methods:

1. `await()`: This method waits until all the threads have reached the barrier. When the required number of threads has arrived, the method returns, and the threads can continue execution.
2. `reset()`: This method resets the barrier to its initial state, allowing the threads to wait for the required number of threads to arrive again.

Semaphore

`Semaphore` is a synchronization mechanism provided in the `java.util.concurrent` package that allows limiting the number of threads that can access a shared resource simultaneously.

The `Semaphore` is initialized with a count of the number of permits available to access the shared resource. Threads can acquire permits from the `Semaphore` before accessing the shared resource. If all permits are already taken, the thread will wait until a permit becomes available. Once a thread has finished accessing the shared resource, it releases the permit back to the `Semaphore`, allowing another thread to acquire it.

The `Semaphore` provides two main methods:

1. `acquire()`: This method acquires a permit from the `Semaphore`, blocking the thread if all permits are already taken.
2. `release()`: This method releases a permit back to the `Semaphore`, allowing another thread to acquire it.

`ThreadFactory` is an interface provided in the `java.util.concurrent` package that allows customizing the creation of new threads in an executor or thread pool.

BlockingQueue

Java `BlockingQueue` is an interface provided in the `java.util.concurrent` package that represents a queue that blocks when attempting to add elements to a full queue or remove elements from an empty queue. It provides a thread-safe way to transfer data between threads, making it useful in multi-threaded applications.

The `BlockingQueue` interface extends the `Queue` interface and provides additional methods for adding, removing, and inspecting elements in a blocking manner. It includes methods such as `put()`, which adds an element to the queue, blocking if the queue is full, and `take()`, which removes and returns an element from the queue, blocking if the queue is empty.

Java provides several concrete implementations of the `BlockingQueue` interface, such as `ArrayBlockingQueue`, `LinkedBlockingQueue`, and `SynchronousQueue`, each with their own characteristics and behaviors. By using a `BlockingQueue`, you can create a producer-consumer pattern, where one or more threads produce data and add it to the queue, and one or more threads consume data from the queue and process it, without the need for explicit synchronization or locks.

DelayQueue

Java `DelayQueue` is a class provided by the `java.util.concurrent` package that implements a blocking queue where elements are ordered according to their expiration

time. Elements are stored in the queue until their expiration time is reached, at which point they can be retrieved from the queue. This class is useful for scheduling tasks that need to be executed after a certain delay, such as reminders, notifications, or timeouts. The `DelayQueue` class provides several methods for adding, removing, and inspecting elements in the queue, as well as for blocking until an element is available or the queue is empty.

Lock

Java `Lock` is an interface provided in the `java.util.concurrent.locks` package that provides a more advanced mechanism for thread synchronization compared to the traditional `synchronized` keyword in Java. It allows multiple threads to access a shared resource concurrently while ensuring that only one thread can modify the resource at a time. The `Lock` interface provides methods for acquiring and releasing the lock and can be used to implement more complex synchronization patterns, such as read-write locks, fair locks, and reentrant locks.

Phaser

Java `Phaser` is a synchronization mechanism provided by the `java.util.concurrent` package that allows coordinating the execution of multiple threads in phases. It works by allowing threads to register themselves with the `Phaser` and wait for other threads to arrive at a common barrier point before continuing execution. The `Phaser` provides methods to advance to the next phase and to wait for other threads to arrive at the barrier point. It is useful in situations where you need to coordinate the execution of multiple threads in a complex workflow that requires synchronization between phases.