Future と ThreadPool

Shigeki Shoji

@takesection

クラウド、GPGPU

- マルチコアプロセッサの性能を効率よく使って並行計算
- 大量のコアを持つGPUをGeneral-purposeに使った機械学習
- リアクティブシステム（リアクティブ宣言）

ThreadPool

Executors

- newCachedThreadPool
- newFixedThreadPool
- newSingleThreadExecutor
- newWorkStealingPool

- newScheduledThreadPool
- newSingleThreadScheduledExecutor

ForkJoinPool

- commonPool

CachedThreadPool

- 新しいタスクが送信されると、スレッドに直接タスクを渡す(handoffs)
  - キャッシュされたスレッドがないときは新規スレッドを作成する
  - プールサイズの制限を超えて新規スレッドが作成できないときは、例外（RejectedExecutionException）がスローされる
  - アイドル状態のスレッドをキャッシュする期間は、KeepAliveTimeで設定（デフォルトは60秒）

FixedThreadPoolとSingleThreadExecutor

- 新しいタスクをキューに追加して、固定数のスレッドを再利用する
  - SingleThreadExecutorは固定数が1で変更不可

WorkStealingPool(ForkJoinPool)

- ForkJoinTaskを実行するためのExecutorService
  - タスクをさらにfork、joinして複数の小さなタスクに分割できるForkJoinTaskを実行することができる
  - プール内のスレッドがプールに送信されたタスクを見つけて実行しようとするため、プロセッサの性能を効率よく利用できる

ScheduledThreadPoolとSingleThreadScheduledExecutor

- 指定された時間経過後や周期的にタスクの実行をスケジュールする
  - SingleThreadScheduledExecutorはスレッド数の変更不可

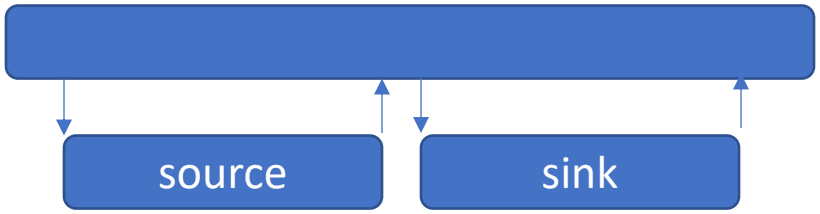
Future

Future

- ExecutorServiceのsubmit、invokeAllで取得
- getメソッドで結果を返すまでブロックすることができる

Futureの結果を後続のFutureで使いたい場合

```
Future<Integer> source =
    executor.submit(new Callable<Integer>() { … });
int res = source.get();
Future<Void> sink =
    executor.submit(new Callable<Void>() { … });
sink.get();
```



CompletableFutureとCompletionStage

- これまでのFutureのように計算結果を取得した値を後続の計算する場合にFuture.get()のようなブロックを発生させないように、Futureのままsupplier、 function、 consumerに計算結果を渡していくことができる

```
private CompletableFuture<Void> process(int n) throws InterruptedException {
    return CompletableFuture.
        supplyAsync(() -> {
            logger.info(String.format("supplier: %d %s", n,
                Thread.currentThread().toString()));
            return n;
        }, executor).
        thenApply(x -> {
            logger.info(String.format("function: %d %s", x,
                Thread.currentThread().toString()));
            return x * 2;
        }).
        thenAccept(x ->
            logger.info(String.format("consumer: %d %s", x,
                Thread.currentThread().toString()))
        );
}
```

バックプレッシャーとFlow

- reactive-streams 仕様に対応したバックプレッシャーにより Push ベースの通信によるリソース管理の問題を回避できるインターフェース

デッドロック

- CachedThreadPoolは、スレッドを作成できないときに例外が発生する

- ForkJoinPool、 FixedThreadPool、 SingleThreadExecutorは、タスクはキューに入る。
  - スレッドは、Thread.sleep() や、IO待ちあるいは、Future.get()のように別のスレッドの計算結果を待つと、実際は何も処理をしていないにもかかわらず、他のタスクがそのスレッドを使うことができなくなる。
  - すべてのスレッドがアクティブな時に、スレッドが同じスレッドプールから別のスレッドを要求して計算結果を待つとデッドロックが発生する。

デッドロックの回避

- ノンブロッキングIOを使う
- CompletableFuture(CompletionStage)を使い計算結果をCompletionStageにとどまり続けるようにする
- 異なるスレッドプールのスレッドに移譲する
- スレッドプールがForkJoinPoolの場合は、ForkJoinPool.ManagedBlockerを実装したブロッキングタスクを使用する
- 十分な数のスレッドプールを算出して使用する

ご清聴ありがとうございました