СПб АУ НОЦНТ РАН

Java 07

18.11.2015

Fork-Join

```
void mergeSort(int begin, int end) {
  int m = (begin + end) / 2;

  // recursive subtasks
  mergeSort(begin, m);
  mergeSort(m, end);

  // join result
  ...
}
```

Fork-Join. Implementation of RecursiveTask<T>.compute

```
T compute() {
    List<MyTask> subTasks = new LinkedList<>();

    for (TaskData subTaskData : getSubTasksData()) {
        MyTask task = new MyTask(subTaskData);
        task.fork();
        subTasks.add(task);
    }

    T result = initial;
    for(MyTask task : subTasks) {
        accumulator = accumulate(
          accumulator, task.join()
        );
    }
    return result;
}
```

Fork-Join. Run

```
import java.util.concurrent.ForkJoinPool;

public class Main {
    public static void main(String[] args) {
        System.out.println(
            new ForkJoinPool().invoke(new MyTask())
        );
    }
}
```

Fork-Join. Implementation of RecursiveTask<T>.compute

```
T compute() {
    List<MyTask> subTasks = new LinkedList<>();

    for (TaskData subTaskData : getSubTasksData()) {
        MyTask task = new MyTask(subTaskData);
        task.fork(); // Magic
        subTasks.add(task);
    }

    T result = initial;
    for(MyTask task : subTasks) {
        accumulator = accumulate(
          accumulator, task.join() // Magic
        );
    }
    return result;
}
```

Опять Singleton

```
class Singleton {
  private static Singleton instance;

  public synchronized static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

Опять Singleton

```
class Singleton {
  private static Singleton instance;

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
          if (instance == null) {
            instance = new Singleton();
          }
          return instance;
      }
    }

    return instance;
  }
}
```

Опять Singleton

```
class Singleton {
  private static Singleton instance;

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
          if (instance == null) {
            instance = new Singleton();
          }
          return instance;
      }
    }

    return instance; // may be null or uninitialized
  }
}
```

Опять Singleton

```
class Singleton {
  private static volatile Singleton instance;

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
          if (instance == null) {
            instance = new Singleton();
          }
          return instance;
      }
    }

    return instance;
  }
}
```

Singleton. Еще одно возможное решение

```
class Singleton {
  final String x = "final";

  private static Singleton instance;

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
          if (instance == null) {
            instance = new Singleton();
          }
          return instance;
      }
    }

    return instance; // can't be uninitialized
  }
}
```

Singleton. Еще одно возможное решение

```
class Singleton {
  final String x = "final";

  private static Singleton instance;

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
          if (instance == null) {
            instance = new Singleton();
          }
          return instance;
      }
    }
    return instance; // can't be uninitialized,
                     // but still may be null
  }
}
```

Singleton. Еще одно возможное решение

```
class Singleton {
  final String x = "final";
  private static Singleton instance;

  public static Singleton getInstance() {
    Singleton result = instance;
    if (result == null) {
      synchronized (Singleton.class) {
          if (instance == null) {
            instance = new Singleton();
          }
          return instance;
      }
    }

    return result;
  }
}
```