|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |

**Практические МАТЕРИАЛЫ №3**

|  |  |
| --- | --- |
| **Операционные системы** | |
| *(наименование дисциплины (модуля) в соответствии с учебным планом)* | |
| Уровень | бакалавриат |
|  | *(бакалавриат, магистратура, специалитет)* |
| Форма обучения | очная |
|  | *(очная, очно-заочная, заочная)* |
| Направление(-я)  подготовки | 09.03.02 Информационные системы и технологии |
|  | *(код(-ы) и наименование(-я))* |
|  |  |
| Институт | комплексной безопасности и специального приборостроения |
|  | *(полное и краткое наименование)* |
| Кафедра | КБ-4 «Интеллектуальные системы информационной безопасности» |
|  | *(полное и краткое наименование кафедры, реализующей дисциплину (модуль))* |
| Лектор | Латыпова Ольга Валерьевна |
|  | *(сокращенно – ученая степень, ученое звание; полностью – ФИО)* |

|  |  |  |  |
| --- | --- | --- | --- |
| Используются в данной редакции с учебного года | 2020/2021 | | |
|  | *(учебный год цифрами)* | | |
| Проверено и согласовано «\_\_\_\_» \_\_\_\_\_\_\_\_20\_\_\_г. |  |  |  |
|  | *(подпись директора Института/Филиала с расшифровкой)* | | |

Москва 2020 г.

**Задание**

Разработать программу на любом высокоуровневом языке программирования выполняющую следующие условия:

Дано 3 производителя и 2 потребителя, все разные потоки и работают все одновременно. Есть очередь с 200 элементами. Производители добавляют случайное число от 1..100, а потребители берут эти числа. Если в очереди элементов >= 100 производители спят, если нет элементов в очереди - потребители спят. Если элементов стало <= 80 производители просыпаются. Все это работает до тех пор пока пользователь не нажал на кнопку "q", после чего производители останавливаются а потребители берут все элементы, только потом программа завершается.

**Теоретическая часть**

1.The Thread Pool

В Java потоки отображаются на потоки системного уровня, которые являются ресурсами операционной системы. Если вы создаете потоки неуправляемо, вы можете быстро исчерпать эти ресурсы.

Переключение контекста между потоками также выполняется операционной системой - чтобы эмулировать параллелизм. Упрощенное представление таково: чем больше потоков вы создаете, тем меньше времени тратит каждый поток на выполнение реальной работы.

Шаблон ThreadPool помогает экономить ресурсы в многопоточном приложении, а также содержать параллелизм в определенных предопределенных пределах.

Когда вы используете пул потоков, вы **пишете свой параллельный код в форме параллельных задач и отправляете их для выполнения в экземпляр пула потоков.** Этот экземпляр контролирует несколько повторно используемых потоков для выполнения этих задач.

Шаблон позволяет **контролировать количество потоков, создаваемых приложением**, их жизненный цикл, а также планировать выполнение задач и сохранять входящие задачи в очереди.

### ****2. Пулы потоков в Java****

#### **2.1.**Executors **,**Executor **и**ExecutorService

Вспомогательный класс *Executors*содержит несколько методов для создания предварительно сконфигурированных для вас экземпляров пула потоков. Эти классы - хорошее место для начала - используйте его, если вам не нужно применять какие-либо пользовательские настройки.

Интерфейсы Executor и ExecutorService используются для работы с различными реализациями пула потоков в Java. Как правило, вы должны **держать свой код отделенным от фактической реализации пула потоков**и использовать эти интерфейсы во всем приложении.

Интерфейс *Executor*имеет единственный метод *execute*для отправки *Runnable*экземпляров для выполнения.

Вот краткий пример того, как вы можете использовать API *Executors*для получения экземпляра *Executor*, поддерживаемого одним пулом потоков и неограниченной очередью для последовательного выполнения задач. Здесь мы выполняем одну задачу, которая просто печатает « *Hello World*» на экране. Задача представлена ​​как лямбда (функция Java 8), которая выводится как *Runnable*.

Executor executor = Executors.newSingleThreadExecutor();

executor.execute(() ->System.out.println("Hello World"));

Интерфейс ExecutorService содержит большое количество методов для контроля за ходом выполнения задач и управления завершением службы. Используя этот интерфейс, вы можете отправлять задачи на выполнение, а также контролировать их выполнение, используя возвращенный экземпляр Future .

В следующем примере создается *ExecutorService*, отправляем задачу, а затем используем возвращенный метод экземпляра класса *Future*  *get()*, чтобы дождаться завершения отправленной задачи и возврата значения:

ExecutorServiceexecutorService = Executors.newFixedThreadPool(10);

Future<String> future = executorService.submit(() ->"Hello World");//some operations

String result = future.get();

Конечно, в реальном сценарии вы обычно не хотите вызывать *future.get()* сразу, а откладываете вызов до тех пор, пока вам действительно не понадобится значение вычисления.

Метод *submit()*перегружен и принимает экземпляры либо *Runnable*, либо *Callable*, оба из которых являются функциональными интерфейсами и могут передаваться как лямбды (начиная с Java 8).

Одиночный метод *Runnable* не генерирует исключение и не возвращает значение. Интерфейс *Callable*может быть более удобным, поскольку он позволяет генерировать исключение и возвращать значение.

Наконец, чтобы позволить компилятору вывести тип *Callable*, просто верните значение из лямбда-выражения

#### **2.2.**ThreadPoolExecutor

*ThreadPoolExecutor*- это расширяемая реализация пула потоков с большим количеством параметров и хуков для тонкой настройки.

Основные параметры конфигурации, которые мы обсудим здесь:

* corePoolSize **,**maximumPoolSize **и**keepAliveTime.

Пул состоит из фиксированного числа основных потоков, которые все время находятся внутри, и нескольких избыточных потоков, которые могут порождаться и затем прерываться, когда они больше не нужны. Параметр *corePoolSize*- это количество основных потоков, которые будут созданы и сохранены в пуле. Если все основные потоки заняты и передано больше задач, то пулу разрешается расти до *maximumPoolSize*.

Параметр *keepAliveTime*- это интервал времени, в течение которого избыточные потоки (то есть потоки, экземпляры которых превышают *corePoolSize*) могут существовать в состоянии ожидания.

Эти параметры охватывают широкий спектр вариантов использования, но наиболее типичные конфигурации предопределены в статическихметодах Executors .

Например, метод *newFixedThreadPool()*создает *ThreadPoolExecutor*с равными значениями параметров *corePoolSize*и *maximumPoolSize*и нулевым *keepAliveTime.*Это означает, что количество потоков в этом пуле потоков всегда одинаково:

ThreadPoolExecutor executor =

(ThreadPoolExecutor) Executors.newFixedThreadPool(2);

executor.submit(() -> {

Thread.sleep(1000);

returnnull;

});

executor.submit(() -> {

Thread.sleep(1000);

returnnull;

});

executor.submit(() -> {

Thread.sleep(1000);

returnnull;

});

assertEquals(2, executor.getPoolSize());

assertEquals(1, executor.getQueue().size());

В приведенном выше примере мы создаем экземпляр *ThreadPoolExecutor*с фиксированным числом потоков 2. Это означает, что если количество одновременно выполняемых задач всегда меньше или равно двум, то они выполняются сразу же. В противном случае некоторые из этих задач могут быть помещены в очередь, чтобы дождаться своей очереди.

Мы создали три *Callable*задачи, которые имитируют тяжелую работу, спят в течение 1000 миллисекунд. Первые две задачи будут выполнены одновременно, а третьему придется ждать в очереди. Мы можем проверить это, вызвав методы *getPoolSize ()*и *getQueue (), size ()*сразу после отправки задач.

Другой предварительно настроенный *ThreadPoolExecutor*может быть создан с помощью метода *Executors.newCachedThreadPool()*. Этот метод вообще не получает несколько потоков. *CorePoolSize*фактически установлен на 0, а *maximumPoolSize*установлен на *Integer.MAX*VALUE\_\_ для этого экземпляра. *keepAliveTime*составляет 60 секунд.

Эти значения параметров означают, что **пул кэшированных потоков может расти без границ для размещения любого количества отправленных задач**. Но когда потоки больше не нужны, они удаляются через 60 секунд бездействия. Типичный вариант использования - это когда в вашем приложении много недолговечных задач.

ThreadPoolExecutor executor =

(ThreadPoolExecutor) Executors.newCachedThreadPool();

executor.submit(() -> {

Thread.sleep(1000);

returnnull;

});

executor.submit(() -> {

Thread.sleep(1000);

returnnull;

});

executor.submit(() -> {

Thread.sleep(1000);

returnnull;

});

assertEquals(3, executor.getPoolSize());

assertEquals(0, executor.getQueue().size());

Размер очереди в приведенном выше примере всегда будет нулевым, поскольку внутри используется экземпляр *SynchronousQueue*. В *SynchronousQueue* пары операций *insert*и *remove* всегда происходят одновременно, поэтому очередь никогда фактически не содержит ничего.

API *Executors.newSingleThreadExecutor()* создает другую типичную форму *ThreadPoolExecutor*, содержащую один поток. **Однопоточный исполнитель идеален для создания цикла событий.**Параметры *corePoolSize*и *maximumPoolSize*равны 1, а *keepAliveTime*равен нулю.

Задачи в приведенном выше примере будут выполняться последовательно, поэтому значение флага будет равно 2 после завершения задачи:

AtomicInteger counter = newAtomicInteger();

ExecutorService executor = Executors.newSingleThreadExecutor();

executor.submit(() -> {

counter.set(1);

});

executor.submit(() -> {

counter.compareAndSet(1, 2);

});

Кроме того, этот *ThreadPoolExecutor*украшен неизменяемой оболочкой, поэтому его нельзя перенастроить после создания. Обратите внимание, что это также причина, по которой мы не можем привести его к*ThreadPoolExecutor*.

#### **2.3.**ScheduledThreadPoolExecutor

*ScheduledThreadPoolExecutor* расширяет класс *ThreadPoolExecutor*, а также реализует интерфейс *ScheduledExecutorService*с помощью нескольких дополнительных методов:

* schedule метод позволяет выполнить задачу один раз после указанного

задержать;

* scheduleAtFixedRate метод позволяет выполнить задачу после. Указанная начальная задержка, а затем выполнить ее несколько раз с определенным период; аргумент *period*- это время **, измеренное между началом время выполнения заданий**, поэтому скорость выполнения является фиксированной;
* scheduleWithFixedDelay метод похож на scheduleAtFixedRate , он многократно выполняет заданную задачу, но указанная задержка **измеряется между окончанием предыдущей задачи и началом следующей**; скорость выполнения может варьироваться в зависимости от времени, которое требуется для выполнения любой данной задачи.

Метод Executors.newScheduledThreadPool() обычно используется для создания ScheduledThreadPoolExecutor с заданным  corePoolSize , неограниченным maximumPoolSize и нулевым keepAliveTime . Вот как можно запланировать выполнение задачи за 500 миллисекунд:

ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);

executor.schedule(() -> {

System.out.println("Hello World");

}, 500, TimeUnit.MILLISECONDS);

Следующий код показывает, как выполнить задачу с задержкой в ​​500 миллисекунд, а затем повторять ее каждые 100 миллисекунд. После планирования задачи мы ждем, пока она не сработает три раза, используя *countDownLatch lock* *,* а затем отменим ее, используя метод*Future.cancel()*.

CountDownLatch lock = newCountDownLatch(3);

ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);

ScheduledFuture<?> future = executor.scheduleAtFixedRate(() -> {

System.out.println("Hello World");

lock.countDown();

}, 500, 100, TimeUnit.MILLISECONDS);

lock.await(1000, TimeUnit.MILLISECONDS);

future.cancel(true);

#### **2.4.**ForkJoinPool

*ForkJoinPool* является центральной частью структуры *fork/join*, представленной в Java 7. Она решает общую проблему **порождения нескольких задач в рекурсивных алгоритмах**. Используя простой *ThreadPoolExecutor*, вы быстро исчерпаете потоки, поскольку для выполнения каждой задачи или подзадачи требуется собственный поток.

В рамках *fork/join*любая задача может порождать ( *fork*) несколько подзадач и ожидать их завершения, используя метод *join*. Преимущество структуры *fork/join*состоит в том, что она **не создает новый поток для каждой задачи или подзадачи**, вместо этого реализуя алгоритм Work Stealing.

Давайте рассмотрим простой пример использования *ForkJoinPool*для обхода дерева узлов и вычисления суммы всех значений листьев. Вот простая реализация дерева, состоящего из узла, значения *int*и набора дочерних узлов:

staticclassTreeNode{

int value;

Set<TreeNode> children;

TreeNode(int value, TreeNode... children) {

this.value = value;

this.children = Sets.newHashSet(children);

}

}

Теперь, если мы хотим суммировать все значения в дереве параллельно, нам нужно реализовать интерфейс*RecursiveTask <Integer>*. Каждая задача получает свой собственный узел и добавляет свое значение к сумме значений своего *children*. Чтобы вычислить сумму значений *children*, реализация задачи делает следующее:

* передает набор children ,
* сопоставляет этот поток, создавая новый CountingTask для каждого элемента,
* выполняет каждую подзадачу, разветвляя ее,
* собирает результаты, вызывая метод join для каждой разветвленной задачи,
* суммирует результаты, используя сборщик Collectors.summingInt

publicstaticclassCountingTaskextendsRecursiveTask<Integer>{

privatefinalTreeNode node;

publicCountingTask(TreeNode node){

this.node = node;

}

@Override

protected Integer compute(){

returnnode.value + node.children.stream()

.map(childNode ->newCountingTask(childNode).fork())

.collect(Collectors.summingInt(ForkJoinTask::join));

}

}

Код для выполнения вычисления на реальном дереве:

TreeNode tree = newTreeNode(5,

newTreeNode(3), newTreeNode(2,

newTreeNode(2), newTreeNode(8)));

ForkJoinPoolforkJoinPool = ForkJoinPool.commonPool();

int sum = forkJoinPool.invoke(newCountingTask(tree));