

Lock. ThreadLocal. Executors



ПРЕПОДАВАТЕЛЬ



Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа



TEL-RAN
by Starta Institute

1

ПОВТОРЕНИЕ ИЗУЧЕННОГО

Повторение

1. Для чего нужны атомарные классы?
2. Чем блокирующие очереди отличаются от неблокирующих?
3. Какой алгоритм Concurrent Collection обеспечивает их работу в многопоточной программе?



Повторение

1. Для чего нужны атомарные классы?
 2. Чем блокирующие очереди отличаются от неблокирующих?
 3. Какой алгоритм Concurrent Collection обеспечивает их работу в многопоточной программе?
-
1. Для обеспечения многопоточного чтения и записи примитивных и ссылочных переменных без синхронизации и блокировки ресурса.
 2. Блокирующие очереди в отличие от неблокирующих при вызове некоторых методов останавливают (блокируют) выполнение текущего потока до тех пор, пока не появилось место в очереди (при операции записи в очередь) или не появился элемент в очереди (при извлечении элемента).
 3. Алгоритм CopyOnWrite. При изменении данных коллекции создаётся полная копия внутреннего хранилища коллекции. Копия заменяет оригинал, когда его никто не читает.

Повторение

Как переделать класс под использование в
многопоточной программе?

```
import java.util.ArrayList;
import java.util.List;

public class ResultHolder {
    private int copiesCount;
    private List<Double> results;

    public ResultHolder() {
        results = new ArrayList<>();
        copiesCount = 0;
    }

    public void addResult(double result) {
        results.add(result);
    }

    public List<Double> getResultCopy() {
        copiesCount++;
        return new ArrayList<>(results);
    }
}
```


Повторение

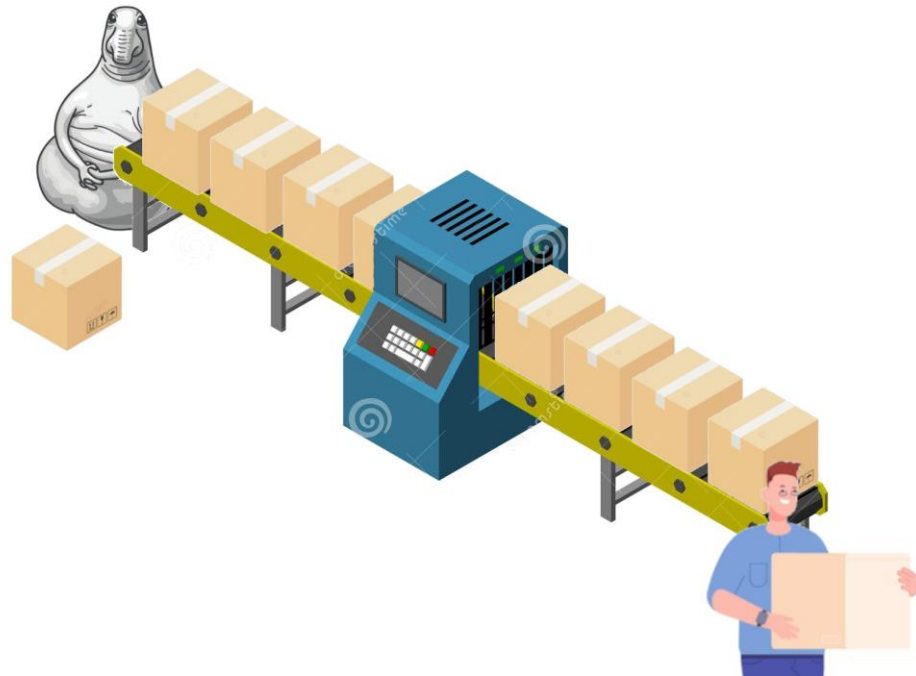
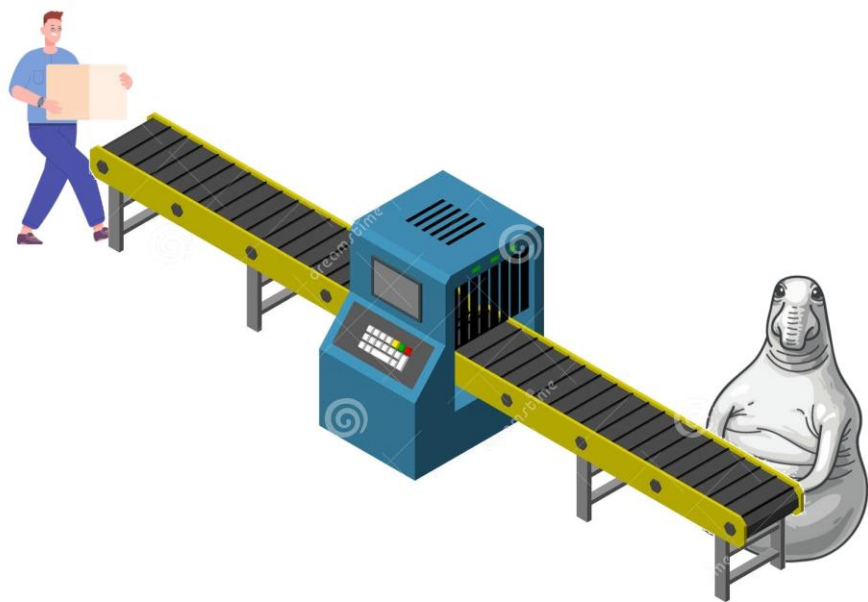
Как переделать класс под использование в
многопоточной программе?

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicInteger;

public class MultithreadedResultHolder {
    private AtomicInteger copiesCount;
    private List<Double> results;
    public MultithreadedResultHolder() {
        results = new CopyOnWriteArrayList<>();
        copiesCount = new AtomicInteger(0);
    }
    public void addResult(double result) {
        results.add(result);
    }
    public List<Double> getResultCopy() {
        copiesCount.incrementAndGet();
        return new ArrayList<>(results);
    }
    public int getCopiesCount() {
        return copiesCount.get();
    }
}
```

Повторение

Какая программная сущность изображена на рисунках?



2

ОСНОВНОЙ БЛОК

Введение

- Лаки локи
- Местный
- Запрос в будущее
- Экзекуция



Проблема

Пакет *java.util.concurrent* предоставляет API более высокого уровня для синхронизации и управления потоками.

Пакет *java.util.concurrent.locks* включает классы, которые существенно отличаются от встроенной синхронизации и мониторов, и которые можно использовать для блокировки ресурсов с определенными условиями. Этот пакет дает намного большую гибкость в использовании блокировок без условий и с условием.



Лаки локи

Интерфейс Lock

Lock – базовый интерфейс из пакета *java.util.concurrent.locks*, предоставляющий более гибкий подход при ограничении доступа к ресурсам/блокам по сравнению с использованием *synchronized*, *wait()* и *notify()*:

- возможность блокировки потока до тех пор, пока текущий поток не прервётся (с простой синхронизацией невозможно прервать поток, ожидающий блокировки).
- возможность одной блокировке иметь несколько условий, чтобы ждать или пробуждаться.
- при использовании нескольких блокировок, порядок их освобождения может быть произвольный (при простой синхронизации – только в порядке приобретения).



Лаки локи

Интерфейс Lock

- есть возможность перехода к альтернативному сценарию, если блокировка уже захвачена.
- возможность запрашивать блокировку, если она свободна в течение заданного времени ожидания и если текущий поток не был прерван.
- возможность делать блокировку не в блоке кода. С одной стороны, это даёт гибкость, но с другой – нужно помнить, что блокировку требуется снять. При использовании *synchronized* JVM снимает блокировку сама при окончании синхронизованного блока кода.

```
Lock lk = ...;
lk.lock();
try {
    // доступ к защищенному блокировкой ресурсу
} finally {
    // освобождение блокировки
    lk.unlock();
}
```

Лаки локи

Методы Lock

Метод	Описание
<i>lock()</i>	Получение блокировки
<i>lockInterruptibly()</i>	Получение блокировки, если текущий поток не прерывается
<i>newCondition()</i>	Получение нового <i>Condition</i> , связанного с блокировкой <i>Lock</i>
<i>tryLock()</i>	Получение блокировки, если она свободна во время вызова
<i>tryLock(long time, TimeUnit unit)</i>	Получение блокировки в течение заданного времени
<i>unlock()</i>	Освобождение блокировки

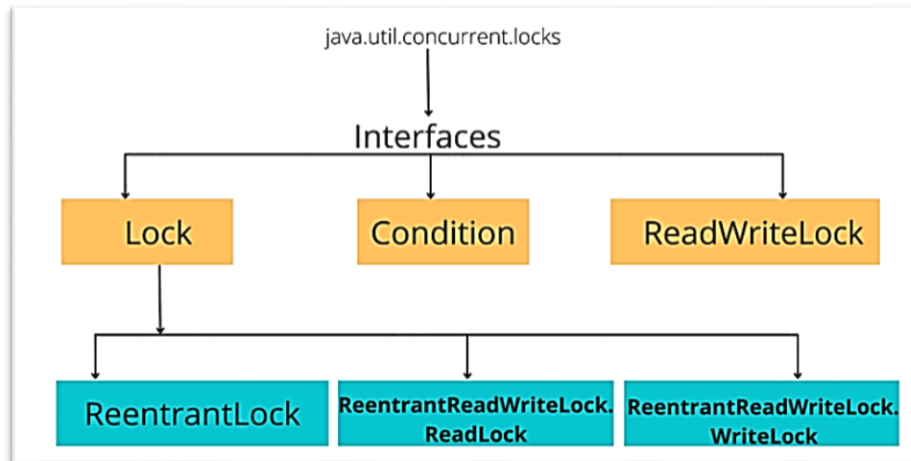
Запрос блокировки с возможностью прерывания процесса ожидания, когда поток запрашивает блокировку методом *lockInterruptibly()* и не получает ее сразу же, переходит в процесс ожидания. Методом *interrupt()* работу потока можно прервать. Тогда ожидающий блокировки поток просыпается, и генерируется *InterruptedException*. После этого попыток доступа к защищенному ресурсу (получения блокировок) не делается и освобождать блокировку не требуется.

Лаки локи

ReentrantLock

Класс **ReentrantLock** (*reentrant* – повторный вход, повторное использование), реализующий интерфейс *Lock*, также, как и *synchronized*, обеспечивает многопоточность, но имеет дополнительные возможности, связанные с опросом о блокировании (*lock polling*), ожиданием блокирования в течение определенного времени и прерыванием ожидания блокировки.

ReentrantLock более эффективен в условиях жесткой состязательности.



Лаки локи

Пример использования `lockInterruptibly`

Внутренний блок *try-finally* получает блокировку и доступ к защищенным ресурсам; после завершения работы блокировка освобождается.

Внешний блок *try-catch* обрабатывает исключительные ситуации запроса блокировки.

Если поток прерван в результате исключительной ситуации, то выполняется перехват *catch* (*InterruptedException*) и метод снятия блокировки *unlock* не вызывается.

```
Lock l = new ReentrantLock();
try {
    l.lockInterruptibly();
    try {
        // работа с защищенным ресурсом
    } finally {
        l.unlock();
    }
} catch (InterruptedException e) {
    System.err.println("Interrupted wait");
}
```

Лаки локи

Пример использования ReentrantLock



ThreadReentrantLockExample.zip



Задание

Разработайте программу, которая имитирует работу банка с несколькими банкоматами. Создайте класс `BankAccount`, содержащий поле `int balance` и методы `withdraw` и `deposit`, которые изменяют значение счета.

Создайте класс `ATM` (банкомат), представляющий поток. Каждый банкомат должен иметь доступ к общему объекту `BankAccount` и выполнять операции снятия и внесения денег. Используйте `ReentrantLock` для обеспечения атомарности операций счета и избежания состояний гонки.



ThreadReentrantLockTaskAtm.zip

Лаки локи

Condition

Интерфейс *Lock* позволяет осуществлять более гибкое структурирование и поддерживает многократно связанный условный объект *Condition*.

Интерфейсное условие *Condition* (очередь условия) в сочетании с блокировкой *Lock* позволяет заменить методы монитора/мьютекса (*wait*, *notify* и *notifyAll*) объектом, управляющим ожиданием событий. Блокировка *Lock* заменяет использование *synchronized*, а *Condition* – объектные методы монитора.

Condition предоставляет средство управления для одного потока, чтобы приостановить его выполнение до тех пор, пока он не будет уведомлен другим потоком. Чтобы получить *Condition* для блокировки *Lock* используют метод *newCondition()*.

```
ReentrantLock locker = new ReentrantLock();  
Condition condition = locker.newCondition();
```

Методы Condition

Метод	Описание
<i>await()</i>	Переводит поток в состояние ожидания до тех пор, пока не будет выполнено некоторое условие или пока другой поток не вызовет методы <i>signal/signalAll</i>
<i>await(long time, TimeUnit unit)</i>	Переводит поток в состояние ожидания на определенное время пока не будет выполнено некоторое условие или пока другой поток не вызовет методы <i>signal/signalAll</i>
<i>signal()</i>	Сигнализирует потоку, у которого ранее был вызван метод <i>await()</i> , о возможности продолжения работы. Применение аналогично использованию метода <i>notify</i> класса <i>Object</i>
<i>signalAll()</i>	Сигнализирует всем потокам, у которых ранее был вызван метод <i>await()</i> , о возможности продолжения работы. Применение аналогично использованию метода <i>notifyAll</i> класса <i>Object</i>

Чтобы перевести поток в ожидание, если определенное условие не выполняется, используется метод *await*: *while (условие) condition.await();*

После завершения всех действий в потоке (при выходе) подается сигнал об изменении условия другим потокам: *condition.signalAll();*

Лаки локи

Использование Condition



ThreadReentrantCondExample.zip



Лаки локи

ReadWriteLock

ReadWriteLock – интерфейс создания *read/write* блокировок, который реализует один единственный класс **ReentrantReadWriteLock**. Блокировку чтение-запись следует использовать при длительных и частых операциях чтения и редких операциях записи. Тогда при доступе к защищенному ресурсу используются разные методы блокировки, как показано ниже:

```
ReadWriteLock rwl = new ReentrantReadWriteLock();  
Lock readLock    = rwl.readLock();  
Lock writeLock   = rwl.writeLock();
```

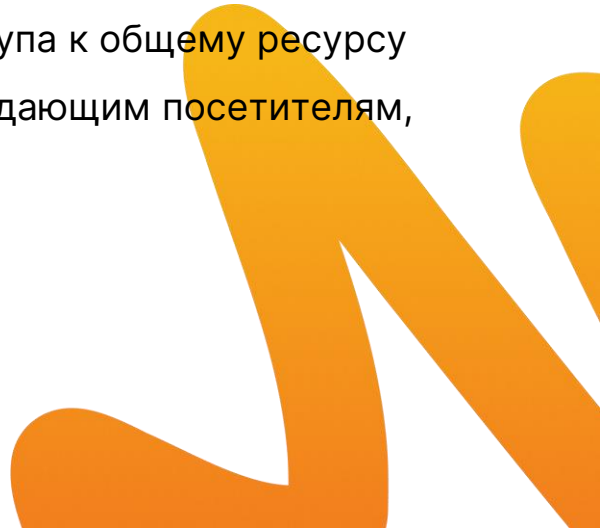
ReentrantReadWriteLock позволяет только одному потоку записывать в ресурс одновременно, при этом, когда наступает очередь чтения, читать могут многие потоки.

Необходимо соблюдать разумное количество читателей, иначе у потока-писателя наступает «голодание» (недоступность общего ресурса длительное время). Как альтернативу можно применить [StampedLock](#).

Задание

Разработайте программу для кафе, где ограничено количество посадочных мест. Кафе может принимать новых посетителей, если есть свободные места, и обслуживать их. Каждый посетитель может сделать заказ и занять место. Если все места заняты, новые посетители должны подождать, пока кто-то не освободит место.

Используйте `ReentrantLock` для обеспечения безопасности доступа к общему ресурсу (количеству свободных мест) и `Condition` для сигнализации ожидающим посетителям, когда появится свободное место.



Проблема

Предположим, что в приложение приходят запросы по сети от многих пользователей.

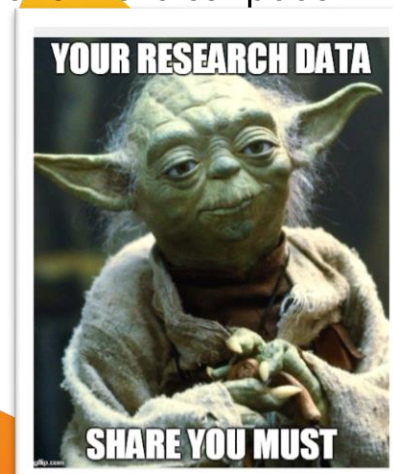
Каждый пользователь обрабатывается в своём потоке.

Приложение состоит из многих классов, каждый из которых выполняет свою задачу.

Например, один класс проверяет данные пользователя, второй – обрабатывает сам запрос, третий – накапливает статистику запросов, четвёртый – рассчитывает биллинг за предоставленную услугу и т.д. Может оказаться, что данные о пользователе и его запросе потребуются всем этим классам.

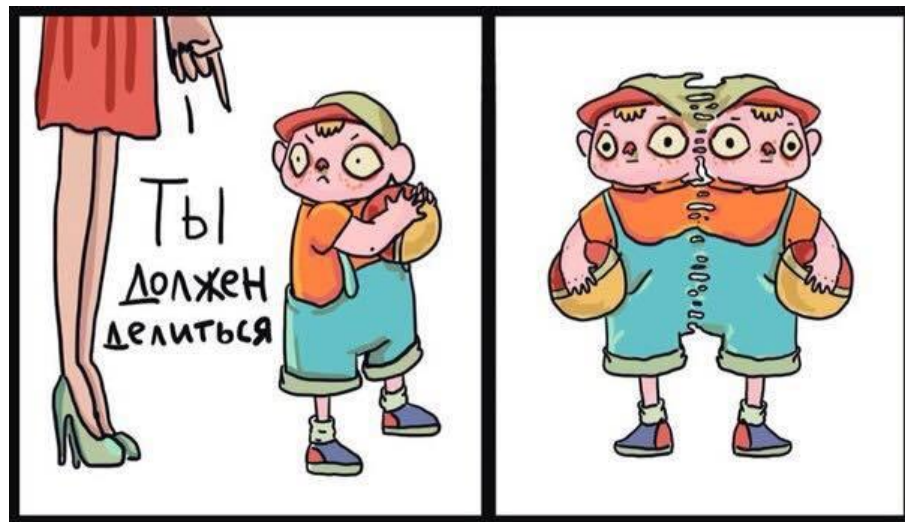
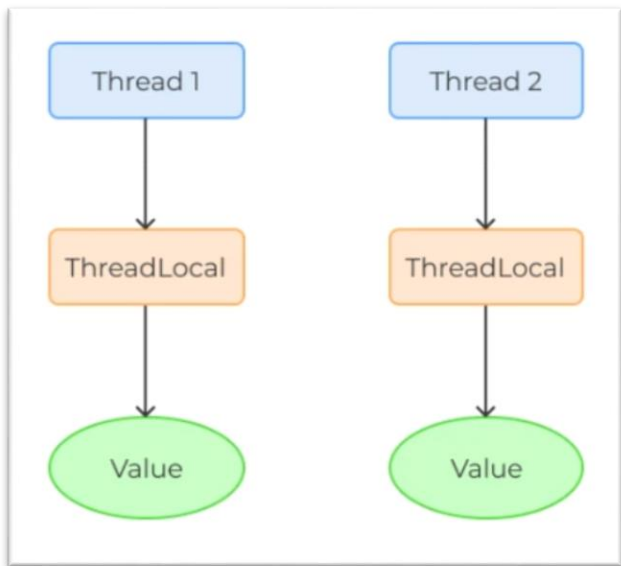
Можно передавать данные в конструктор или метод соответствующего класса, но это сильно увеличивает объём кода.

Глобальных переменных в Java нет. Как передать нужную информацию всем классам?



Местный ThreadLocal

ThreadLocal – класс обёртка, который позволяет передавать данные между методами разных классов, вызываемых в одном потоке. Такая переменная будет уникальна для каждого потока. Переменная *ThreadLocal* одного потока не доступна в другом потоке.



Местный Методы ThreadLocal

ThreadLocal() // Создает пустую переменную в Java

get() // Возвращает значение локальной переменной текущего потока

set() // Устанавливает значение локальной переменной для текущего потока

remove() // Удаляет значение локальной переменной текущего потока

ThreadLocal.withInitial() // Дополнительный фабричный метод, который устанавливает начальное значение

Местный Использование ThreadLocal



ThreadLocalExample.zip



Задание

Разработайте приложение, моделирующее работу системы авторизации. Используйте `ThreadLocal` для хранения информации о текущем пользователе в контексте потока. Реализуйте класс `UserContext` с методами `login(String username)` и `logout()`. При успешном входе пользователя, информация о нем должна быть сохранена в `ThreadLocal`. При выходе пользователя, соответствующая информация должна быть удалена из `ThreadLocal`.

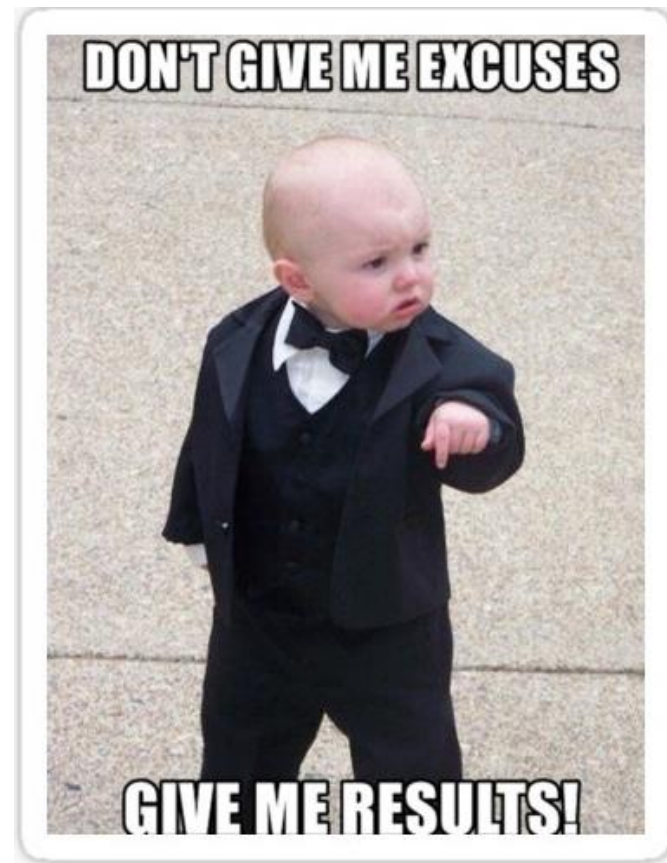


Проблема

При работе многопоточного приложения часто необходимо получение от потока результата его деятельности в виде некоторого объекта.

Интерфейс *Runnable* имеет единственный метод *run*, который не возвращает результата.

Каким образом можно получить результат выполнения потока?

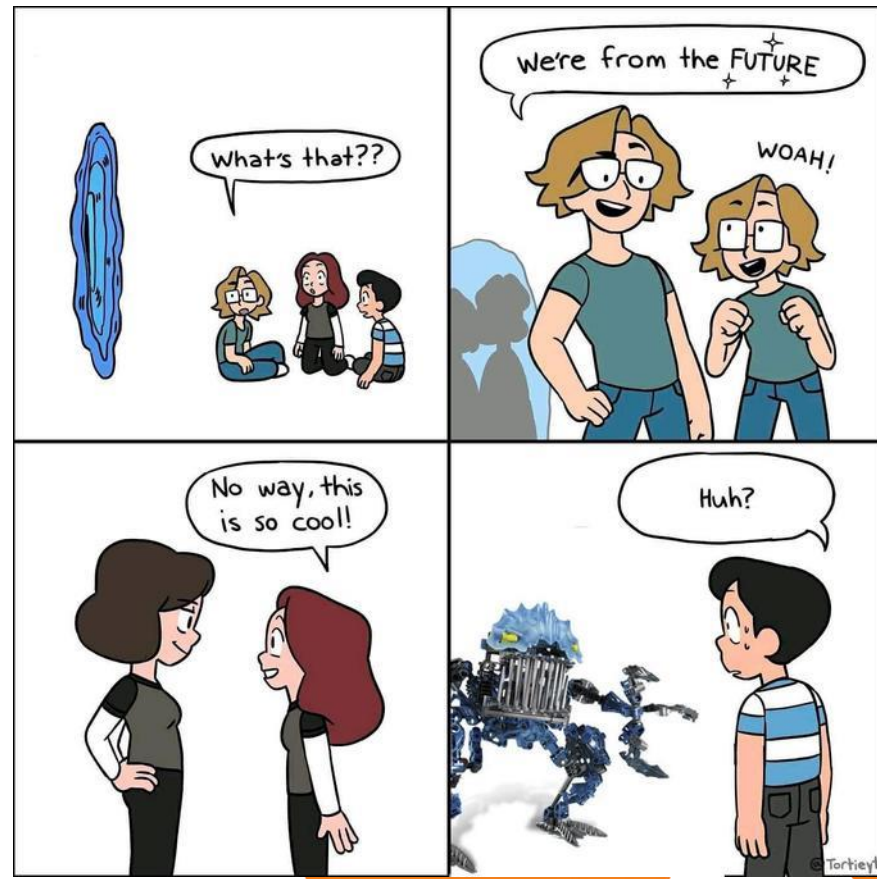


Запрос в будущее

Интерфейсы Callable, Future

Эту задачу можно решить с использованием интерфейсов *Callable<V>* и *Future<V>*.

Совместное использование двух реализаций данных интерфейсов позволяет получить результат в виде некоторого объекта.



Запрос в будущее

Callable

Интерфейс **Callable** очень похож на интерфейс *Runnable*. Объекты, реализующие данные интерфейсы, исполняются другим потоком. Однако, в отличие от *Runnable*, интерфейс *Callable* использует *Generic*'и для определения типа возвращаемого объекта. *Runnable* содержит метод *run()*, описывающий действие потока во время выполнения, а *Callable* – метод *call()*.



Запрос в будущее

Future



Интерфейс **Future** параметризирован. Методы интерфейса можно использовать для проверки завершения работы потока, ожидания завершения и получения результата.

Метод	Описание
<i>cancel (boolean mayInterruptIfRunning)</i>	попытка завершения задачи
<i>V get()</i>	ожидание (при необходимости) завершения задачи, после чего можно будет получить результат
<i>V get(long timeout, TimeUnit unit)</i>	ожидание (при необходимости) завершения задачи в течение определенного времени, после чего можно будет получить результат
<i>isCancelled()</i>	вернет true, если выполнение задачи будет прервано прежде завершения
<i>isDone()</i>	вернет true, если задача завершена

Запрос в будущее

Использование Callable и Future



ThreadCallableFutureExample.zip



Запрос в будущее

FutureTask

Класс-оболочка **FutureTask** базируется на конкретной реализации интерфейса *Future*.

Чтобы создать реализацию данного класса необходим объект *Callable*.

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*.

Если посмотреть в исходники класса *FutureTask*, то можно увидеть, что он реализует интерфейс *RunnableFuture*, который, в свою очередь, наследует свойства интерфейсов *Runnable* и *Future<V>*.

Задание

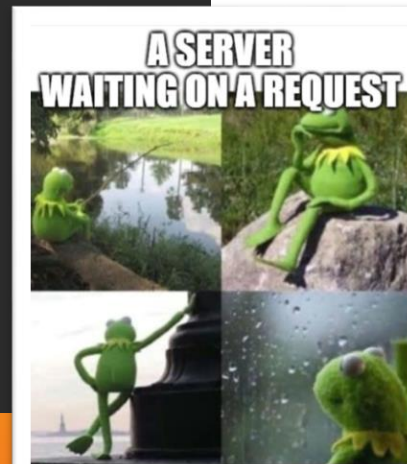
Напишите программу для вычисления факториала числа с использованием Callable и Future. Каждый поток должен вычислять факториал для своего уникального числа.



Проблема

Предположим, что наше клиент-серверное приложение принимает запросы на присоединение клиентов. Для обработки каждого соединения (сокета) логично запускать отдельный поток, в котором будет происходить взаимодействие с клиентом

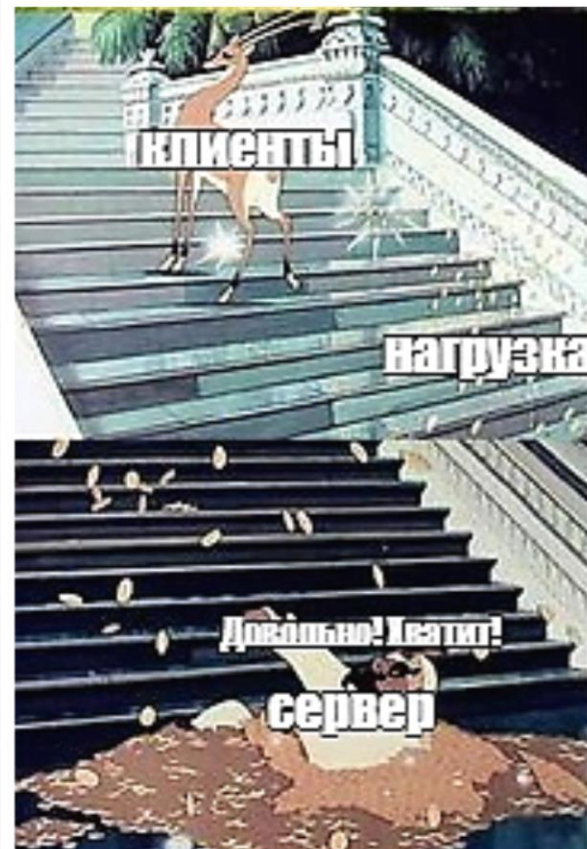
```
public void startServer() {  
    try (ServerSocket serverSocket = new ServerSocket(property.getPort())){  
        while (!isStop) {  
            Socket socket = serverSocket.accept(); // поток ждёт подключения нового клиента  
            Thread clientHandle = new Thread(() -> {  
                User user = new User(new Connection(socket));  
            });  
            clientHandle.start();  
        }  
    } catch (IOException e) {  
        throw new ServerException("Ошибка запуска сервера ", e);  
    }  
}
```



Проблема

При росте нагрузки (количества клиентов) в какой-то момент окажется, что в системе слишком много потоков и между ними существенная конкуренция. Т.е. многие потоки ждут процессорного времени, регулярно пытаются получить общие ресурсы и, не получив их, снова ждут. Каждый поток в этот момент занимает часть оперативной памяти (оверхед) и требует ресурсов процессора на переключение между потоками. Таким образом, ресурсы заняты потоком, но простаивают.

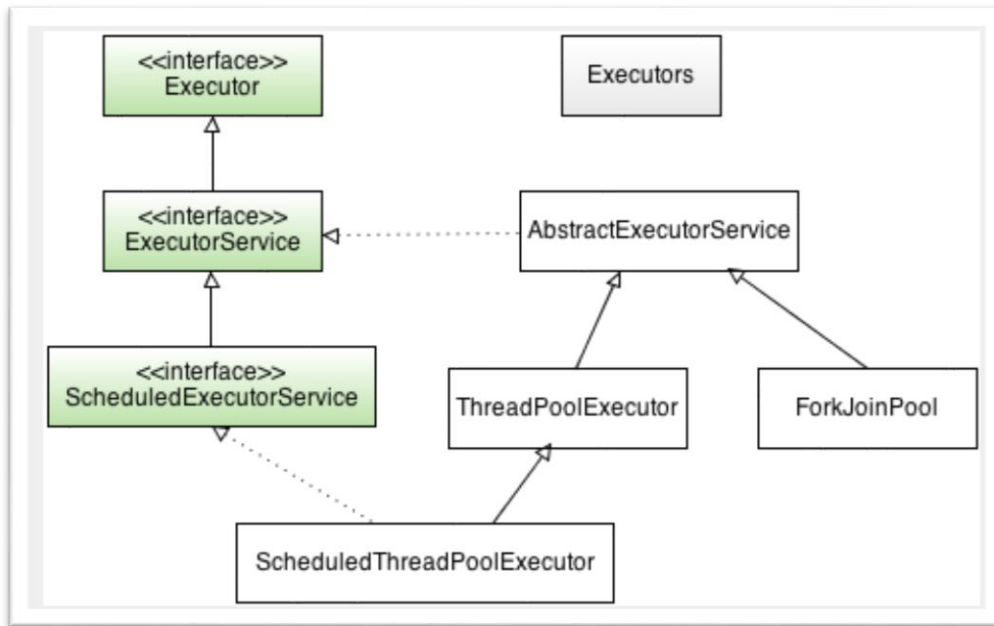
Каким образом можно ограничить рост числа потоков в системе каким-то разумным пределом?



Экзекуция

Сервисы исполнения

Сервисы исполнения (executors) – позволяющие управлять потоковыми задачами с возможностью получения результатов через интерфейсы *Future* и *Callable*. Сервисы исполнения служат альтернативой классу *Thread* и позволяют работать с набором (пулом потоков).

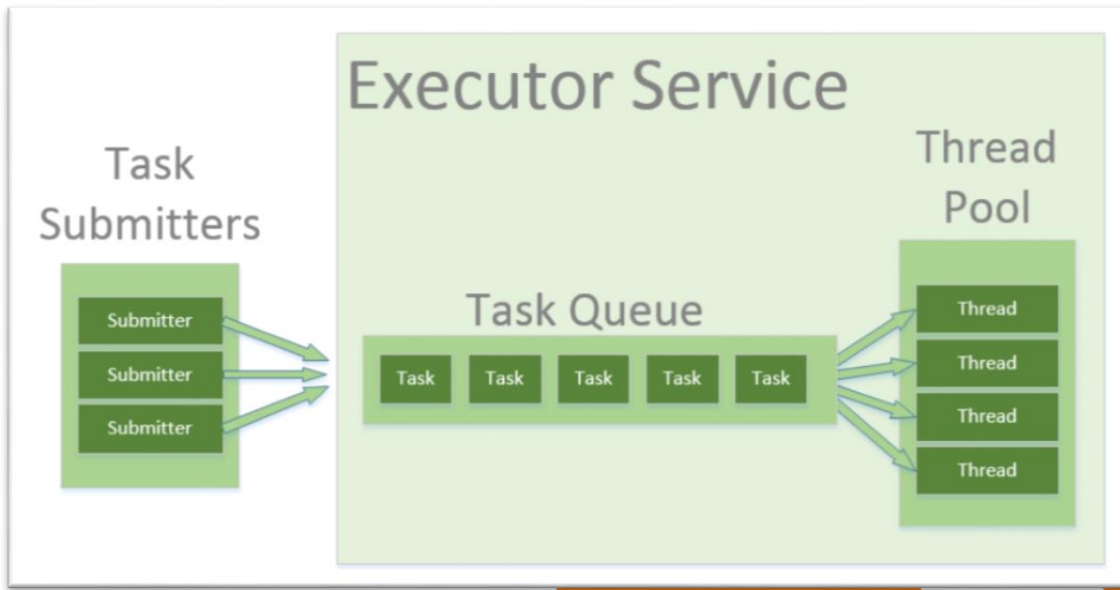


Экзекуция ExecutorService

ExecutorService – основной интерфейс, представляющий работу *сервисов исполнения*, имплементация которого используется для запуска потоков. *ExecutorService* состоит из очереди входных заданий и пула потоков, которые эти задачи обрабатывают. Если все потоки пула заняты, то задачи ждут в очереди. Очередь может быть ограничена по размеру при необходимости.

Размер пула потоков может быть фиксирован или изменяться динамически в заданных пределах.

Реализации интерфейса *Executor* при создании позволяют установить настройки для пула потоков и очереди задач.



Экзекуция

Методы ExecutorService

Потоки можно запускать, используя методы *execute* и *submit*. Оба метода в качестве параметра принимают объекты *Runnable* или *Callable*.

```
void execute(Runnable thread);
```

submit возвращает значение типа *Future*, позволяющий получить результат выполнения потока.

invokeAll работает со списками задач с блокировкой потока до завершения всех задач в переданном списке или до истечения заданного времени.

invokeAny блокирует вызывающий поток до завершения любой из переданных задач.

shutdown позволяет завершить все принятые на исполнение задачи и блокирует поступление новых.

boolean awaitTermination (long timeout, TimeUnit unit)	Блокировка до тех пор, пока все задачи не завершат выполнение после запроса на завершение работы или пока не наступит тайм-аут или не будет прерван текущий поток, в зависимости от того, что произойдет раньше
List<Future<T>> invokeAll (Collection<? extends Callable<T>> tasks)	Выполнение задач с возвращением списка задач с их статусом и результатами завершения
List<Future<T>> invokeAll (Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Выполнение задач с возвращением списка задач с их статусом и результатами завершения в течение заданного времени
T invokeAny (Collection<? extends Callable<T>> tasks)	Выполнение задач с возвращением результата успешно выполненной задачи (т. е. без создания исключения), если таковые имеются
T invokeAny (Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Выполнение задач в течение заданного времени с возвращением результата успешно выполненной задачи (т. е. без создания исключения), если таковые имеются

boolean isShutdown()	Возвращает true, если исполнитель сервиса остановлен (shutdown)
boolean isTerminated()	Возвращает true, если все задачи исполнителя сервиса завершены по команде остановки (shutdown)
void shutdown()	Упорядоченное завершение работы, при котором ранее отправленные задачи выполняются, а новые задачи не принимаются
List<Runnable> shutdownNow()	Остановка всех активно выполняемых задач, остановка обработки ожидающих задач, возвращение списка задач, ожидающих выполнения
Future<T> submit (Callable<T> task)	Завершение выполнения задачи, возвращающей результат в виде объекта Future
Future<?> submit (Runnable task)	Завершение выполнения задачи, возвращающей объект Future, представляющий данную задачу
Future<T> submit (Runnable task, T result)	Завершение выполнения задачи, возвращающей объект Future, представляющий данную задачу


Экзекуция

Пример использования `ExecutorService`

Достаточно создать объект типа *ExecutorService* с нужными свойствами и передать ему на исполнение задачу типа *Callable*. Впоследствии можно легко просмотреть результат выполнения этой задачи с помощью объекта *Future*.

```
// Вместо следующего кода
new Thread(new RunnableTask()).start();

// можно использовать
ExecutorService executor;
// . . .
executor.execute(new CallableSample1());
Future<String> f1 = executor.submit(new CallableSample2());
```



Экзекуция

Пул потоков

ThreadPool в Java представляет собой набор потоков, предназначенный для повторного использования. Он обеспечивает управление потоками и распределение задач между ними, что улучшает производительность и эффективность работы с большим количеством задач. Таки образом, многопоточность отделена от логики задач.

Фиксированный пул потоков – пул, имеющий фиксированный размер, указываемый при создании. Например,

```
ThreadPool threadPool = new ThreadPool(100);
```

Если поток в пуле завершается, то он автоматически заменяется новым потоком, ожидающим задачи. Это уменьшает время реакции приложения на новую входную задачу.

Кешированный пул потоков – пул, создающий новые потоки по мере необходимости и переиспользующий ранее созданные места в пуле.

Однопоточный пул – пул с одним активным потоком и неограниченной очередью. При остановке потока, он будет перезапущен.

Экзекуция

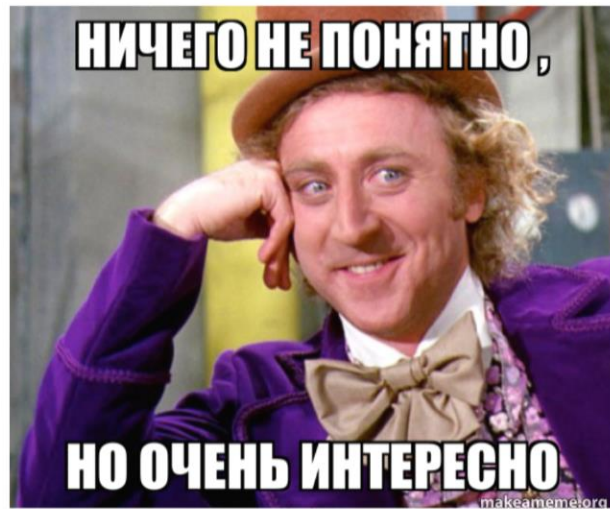
Реализации пулов потоков

Кроме класса *ThreadPool* существуют и другие реализации пулов потоков (для информации).

MemoryAwareThreadPoolExecutor – пул потоков с ограничением по памяти, который блокирует отправку задачи, когда в очереди слишком много задач.

JMXEnabledThreadPoolExecutor – пул потоков, который регистрирует JMX-компонент* для контроля и настройки размера пула в runtime.

** Java Management Extensions (JMX) — это технология, позволяющая управлять и мониторить приложения, разработанные на языке Java. JMX API предоставляет набор инструментов для взаимодействия с ресурсами управления*



Экзекуция

Размер пула потоков

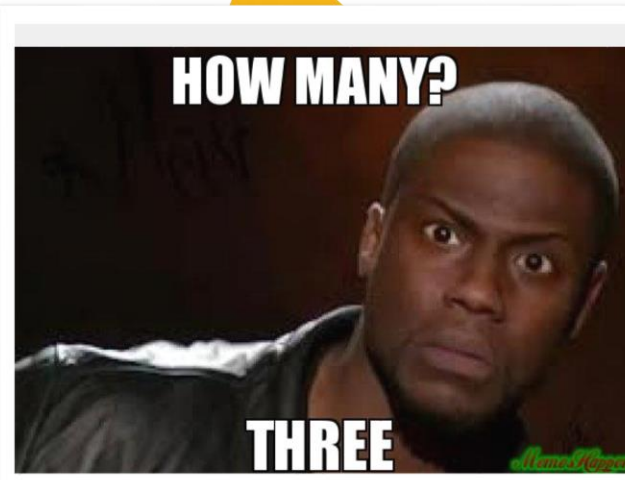
Определить размер пула на первом этапе создания программы можно по следующему эмпирическому правилу:

1. Для чисто вычислительных задач размер пула потоков равен количеству процессоров системы **N**.
2. Для сетевых задач размер пула можно оценить так:

$$\text{Размер пула} = N * (1 + WT/ST),$$

где

- N - количество процессоров (ядер)
- WT - время ожидания ресурса
- ST - время обслуживания запроса



Экзекуция **ScheduledExecutorService**



Интерфейс **ScheduledExecutorService** расширяет свойства *ExecutorService* для поддержки планирования потоков исполнения. Он применяется, когда требуется выполнение кода *асинхронно* и *периодически* (через некоторое время). Интервалом может быть время между двумя последовательными запусками или время между окончанием одного выполнения и началом другого.

Методы *ScheduledExecutorService* возвращают *ScheduledFuture*, который также содержит значение отсрочки для выполнения *ScheduledFuture*.

Экзекуция

Создание ScheduledExecutorService

```
// если требуется отложить выполнение на 5 секунд
ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();
service.schedule(new Runnable() { ... }, 5, TimeUnit.SECONDS);

// если требуется назначить выполнение каждую секунду
ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();
service.scheduleAtFixedRate(new Runnable() { ... }, 0, 1, TimeUnit.SECONDS);

// если требуется назначить выполнение кода с промежутком 1 секунда между выполнениями
ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();
service.scheduleWithFixedDelay(new Runnable() { ... }, 0, 1, TimeUnit.SECONDS);
```

Экзекуция Реализации ExecutorService



В пакет *concurrent* включены три предопределенных класса исполнителей:

ThreadPoolExecutor, *ScheduledThreadPoolExecutor* и *ForkJoinPool*.

ThreadPoolExecutor реализует интерфейс *ExecutorService* и обеспечивает поддержку управляемого пула потоков исполнения. Класс *ScheduledThreadPoolExecutor* реализует интерфейс *ScheduledExecutorService* для поддержки планирования пула потоков исполнения. А класс *ForkJoinPool* реализует интерфейс *ExecutorService* и применяется в каркасе *Fork/Join Framework*.

Чтобы получить реализацию данных объектов необходимо использовать класс-фабрику *Executors*.

Экзекуция **ThreadPoolExecutor**

ThreadPoolExecutor – это конкретная реализация *ExecutorService*, которая представляет собой гибкий и мощный пул потоков.



Экзекуция

Применение ThreadPoolExecutor



ThreadPoolExample.zip



Задание

1 В фирме есть несколько отделов, каждый из которых обрабатывает разные типы задач. Разработайте программу, используя `ThreadPoolExecutor`, чтобы распределить выполнение задач по отделам фирмы параллельно. Каждая задача представляет собой обработку запроса от клиента, и у каждого отдела есть свои силы для выполнения задач.

2 На предприятии требуется регулярное техническое обслуживание оборудования. Разработайте программу с использованием `ScheduledThreadPoolExecutor`, чтобы планировать выполнение технического обслуживания каждого устройства через определенные промежутки времени.

3

Домашнее задание

Домашнее задание

1 Разработайте программу, эмулирующую работу нескольких пользователей (поток) с единым электронным документом. Все пользователи могут добавлять, редактировать и удалять записи (одна запись – одна строка). Каждый новый пользователь получает актуальную копию документа и работает с ней. После внесения изменений пользователь сохраняет изменения. Для оптимизации доступа к документу используйте `ReentrantReadWriteLock`.

2 Создайте классы Клиент, Товар, Задание. Создайте единый механизм для генерации уникальных идентификаторов для каждого из классов (у Клиентов свои идентификаторы, у Товаров – свои, у Заданий – свои). Используйте обёртку `ThreadLocal` для хранения счетчика уникальных идентификаторов в каждом потоке. Реализуйте класс `UniqueldGenerator`, который предоставляет метод `generateUniqueld()`, возвращающий уникальный идентификатор.

3 В интернет-магазине есть два процесса: обработка заказов и доставка товаров. На фоне обработки заказов, нужно периодически планировать доставку готовых товаров. Разработайте программу, используя `ThreadPoolExecutor` и `ScheduledThreadPoolExecutor`, чтобы эффективно управлять обработкой заказов и доставкой.

ЗАКЛЮЧЕНИЕ

