03102022 Threads Потоки выполнения.

Все программы управляются на основе ОС и идея многозадачности (псевдо – поочередное быстрое переключение между программами(для 1-ядерных), реальная - ) имеет 2 реализации, в зависимости от ОС:

- основанная на потоках, конкуренция только за процессорное время

- основанная на процессах , сложный в поддержании, дорого и ресурсоемко, для взаимодействия процессов требуются дополнительные прослойки

Потоки(нити) полезная нагрузка процессов

P2

P1

P1 t

P2 t

P1

Промежуточные процессы

сохранения разделяемых ресурсов,

издержки для поддержания

многозадачной системы

Промежуточное ПО для взаимодействия процессов

Мультизадачность процесса обеспечивается потоками, возможность распараллелить саму программу (процесс), разные части кода будут запускаться – выполняться в отдельном потоке ,параллельно , в рамках одного процесса, что позволит повысить производительность системы, тк теперь потоки в рамках контекста процесса будут конкурировать только за процессорное время, а остальное – Heap динамическая память со всем его содержимым (для JAVA это статика) и другие ресурсы, как разделяемые, так и не разделяемые будут общими для всех потоков, своими у каждого процесса будут только локальный Stack и процессорное время. Планировщику в свою очередь легче переключать потоки чем процессы. Современные ОС реализуют многозадачность на процессах, один поток содержит минимум один поток.

В Java механизм многозадачности основан на потоках, тк сама JVM это процесс, а выполняемые ей программы – потоки. Память в JVM делится на STACK & HEAP- на динамическую память и статический контекст, куда подгружаются объекты класса Class, благодаря которым создаем экземпляры и POOL-String & Int литералы. Память для всех Java программ одна и та же, свой только Stack, что и обеспечивает масштабируемость. Перейдем к номенклатуре:

Все потоки в JVM объединяются в группы, из них выделяется главная main, которая является отправной точкой запуска, в группе в свою очередь тоже есть главный поток main , стартует с метода main. Каждый поток характеризуется:

JVM main

-Структурой, метаданные описывающие поток

-Поточная функция, код который будет выполняться в рамках данного потока

main

Для JVM все потки одинаковы, с точки зрения завершения, если ф-ция главного потока завершится, остальные – продолжат работу до завершения последнего активного.

Рассмотрим структуру потока подробнее:

ID- константа long(тлько для чтения), по которому планировщик, обращается к потокам для управления

Name – мб изменено, по своему усмотрению, по дефолту Thread1, Thread2, MainThread итд

Thread group-можно создавать свои дочерние потоки

Priority влияет на частоту выдачи квантов времени выделяемых потоку, от 1-самый низкий до 10- самый высокий, default -5, для изменения используются именованные константы MAX/MIN/NORMAL\_PRIORITY,

State : до момента запуска – инициализационное, после запуска метода start() выполнения, ожидание – процессорного времени или другого ресурса, блокировка – другим потоком, терминальное – выполнился был уничтожен, но осталась структура пока не будут прочитаны результирующие данные, Живой-не живой, является ли демоном.

Классификация потоков Foreground-интерактивный & Background – потоки демоны, сервисные, работают параллельно, закрываются.

Программа работает пока жив хоть один Foreground поток.

Рассмотрим функцию потоков:

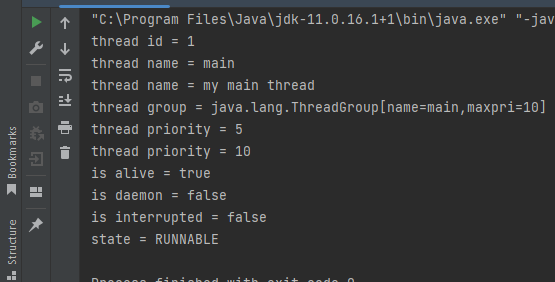
Main() запуск главного потока

Run()запуск параллельного выполнения потока, декларируется в (функциональном – содержит только одно описание метода)IF Runnable, в нем описывается код, который должен выполнятся параллельно с другим методом.

Перейдем к практике, задачи:

1 рассмотреть метаданные

2 научиться управлять главным потоком

Реализуем проект с архитектурой MVC, создаем первый контроллер, теперь мы хотим получить ссылку на поток , котрый будет выполнять код метода mainСоздаем ссылку класса Thread и в инициализации на классе вызываем статический метод currentThread, который позволяет получить ссылку на текущий поток. Теперь посмотрим какую информацию о потоке мы можем получить:Id, Name-можем изменить, group -тж можем изменить, но только не для главного потока, priority- тж можно изменить, isAlive, isDaemon, is interrupted- приостановлен, state-NEW,RUNNABLE,WAITING…( на базе перечислений).

package itStep.yandr.javaLessons.lesson42.controller;  
public class Controller {  
 public static void main(String[] args) {  
 Thread thread = Thread.*currentThread*();  
 System.*out*.println("thread id = " + thread.getId());  
 System.*out*.println("thread name = " + thread.getName());  
 thread.setName("my main thread");  
 System.*out*.println("thread name = " + thread.getName());  
 System.*out*.println("thread group = " + thread.getThreadGroup());  
 System.*out*.println("thread priority = " + thread.getPriority());  
 thread.setPriority(Thread.*MAX\_PRIORITY*);  
 System.*out*.println("thread priority = " + thread.getPriority());  
 System.*out*.println("is alive = " + thread.isAlive());  
 System.*out*.println("is daemon = " + thread.isDaemon());  
 System.*out*.println("is interrupted = " + thread.isInterrupted());  
 System.*out*.println("state = " + thread.getState());  
 }  
}

Данный поток метаданных характеризует объект потока.\* Вопрос собеседование:

1Есть группа потоков с максимальным приоритетом 6, добавляя поток с приоритетом 10, снизится его приоритете до приоритета группы

2 если добавляемый поток по приоритету ниже максимального в группе, то сохранится индивидуальный приоритет добавленного.

Заглянем в класс Thread

У любого потока есть цель target, ссылка на объект в котором сосредоточена поточная функция, есть ссылка на ThreadGroup, ClassLoader, встроенный enum состояний потока.

Перейдем непосредственно к созданию потока :

**1 способ**, наследуясь от Thread, таким образом мы полностью наследуем инфраструктуру и создавая экземпляры данного класса они полностью готовы к запуску, переопределив метод run, которая ничего не принимает , ничего не возвращает, тк используется она не клиентом, а JVM, клиент только инициирует запуск и создание потока. Для обеспечения параллельной работы функции используем бесконечный цикл.

package itStep.yandr.javaLessons.lesson42.model;  
  
public class FirstThread extends Thread {  
 @Override  
 public void run() {  
 while (true) {  
 System.*out*.printf("\nid = %d, name = %s", getId(), getName());  
 }  
 }  
}

Описав поточную ф-цию, создаем еще один контроллер в котором опишем сценарий параллельной работы главного и первого потоков. Создаем ссылку на первый поток и запускаем его при помощи метода start, получаем ссылку на главный (текущий поток) через currentThread

package itStep.yandr.javaLessons.lesson42.controller;  
  
import itStep.yandr.javaLessons.lesson42.model.FirstThread;  
  
public class FirstController {  
 public static void main(String[] args) {  
 FirstThread thread = new FirstThread();  
 thread.start();  
 Thread main = Thread.*currentThread*();  
 while (true) {  
 System.*out*.printf("\nid = %d, name = %s", main.getId(), main.getName());  
 }  
 }  
}

Однако такая архитектура противоречит принципу единственной ответственности, тк главный поток, содержит и инфракстуру создания объекта и саму поточную ф-цию

**2.способ**,класс реализует только поточную ф-цию, имплементит Runnable. В run создаем ссылку на текущий поток

package itStep.yandr.javaLessons.lesson42.model;  
public class SecondThread implements Runnable {   
 @Override  
 public void run() {  
 Thread thread = Thread.*currentThread*();  
 while (true) {  
 System.*out*.printf("\nid = %d, name = %s", getId(), getName());  
 }  
 }  
}

Реализуем второй контроллер, создаем ссылку на наш объект, а для вызова функциаонала потока, создаем новый поток, передавая ему target наш объект( может также принимать имя потока или группу потоков)

package itStep.yandr.javaLessons.lesson42.controller;  
  
import itStep.yandr.javaLessons.lesson42.model.SecondThread;  
  
public class SecondController {  
 public static void main(String[] args) {  
 SecondThread threadLogic = new SecondThread();  
 Thread thread = new Thread(threadLogic);  
 thread.start();  
 }  
}

Оба способа имеют значимый минус, мы нагружаем пользователя процессом создания потока….

\*Рассмотрим способы создания на лету:

Анонимный объект:

new FirstThread().start();

SecondThread threadLogic = new SecondThread();  
new Thread(threadLogic).start();

new Thread(new SecondThread()).start();

Pассмотрим взаимодействия и манипуляцию потоками, начнем с потоков демонов. Для приостановки потока есть 2 способа метод sleep(-устарел) класса Thread(+ 2 перегруженные версии, миллисек, наносек), генерирует interrupted exception (как и в других методах останавливающих работу потока), для возможности освободить ресурс, при уничтожении извне, это исключение требует обязательной обработки или делегирования.Более современным методом для приостановки работы потока TimeUnit.SECOND.sleep(более читабельный)

Зассмотрим жизненные циклы наших потоков, в 3м контроллере, создаем объект поточной функции, и на его основе 2 потока, запускаем, приостанавливаем на 5 секунд и выводим сообщение – маркер об окончании работы главного потока

package itStep.yandr.javaLessons.lesson42.controller;  
  
import itStep.yandr.javaLessons.lesson42.model.SecondThread;  
  
import java.util.concurrent.TimeUnit;  
  
public class ThirdController {  
 public static void main(String[] args) throws InterruptedException {  
 System.*out*.println("main thread started");  
 SecondThread threadLogic = new SecondThread();  
 Thread thread1 = new Thread(threadLogic);  
 Thread thread2 = new Thread(threadLogic);  
  
 thread1.start();  
 thread2.start();  
 TimeUnit.*SECONDS*.sleep(5);  
 System.*out*.println("\nmain thread closed");  
  
 }  
}

тепрь попробуем повторить , но с конечной функцией потоков, поменяем ее

package itStep.yandr.javaLessons.lesson42.model;  
  
public class SecondThread implements Runnable {  
  
 @Override  
 public void run() {  
 Thread thread = Thread.*currentThread*();  
 for (int i = 0; i < 1000; i++) {  
 System.*out*.printf("\nid = %d, name = %s", thread.getId(), thread.getName());  
 }  
 System.*out*.printf("\n%s thread closed", thread.getName());  
 }  
}

немного притормозим ф-цию

package itStep.yandr.javaLessons.lesson42.model;  
  
import java.util.concurrent.TimeUnit;  
  
public class SecondThread implements Runnable {  
  
 @Override  
 public void run() {  
 Thread thread = Thread.*currentThread*();  
  
 for (int i = 0; i < 1000; i++) {  
 try {  
 TimeUnit.*MILLISECONDS*.sleep(200);  
 } catch (InterruptedException e) {  
 System.*err*.println(e);  
 }  
 System.*out*.printf("\nid = %d, name = %s", thread.getId(), thread.getName());  
 }  
 System.*out*.printf("\n%s thread closed", thread.getName());  
 }  
}

\*Главный поток завершил свою работу. А остальные потоки продолжили

Перейдем к потоку демону

package itStep.yandr.javaLessons.lesson42.controller;  
import itStep.yandr.javaLessons.lesson42.model.SecondThread;  
import java.util.concurrent.TimeUnit;  
  
public class ThirdController {  
 public static void main(String[] args) throws InterruptedException {  
 System.*out*.println("main thread started");  
  
 SecondThread threadLogic = new SecondThread();  
 Thread thread1 = new Thread(threadLogic);  
 Thread thread2 = new Thread(threadLogic);  
 thread1.setDaemon(true);  
 thread2.setDaemon(true);  
  
 thread1.start();  
 thread2.start();  
 TimeUnit.*SECONDS*.sleep(5);  
 System.*out*.println("\nmain thread closed");  
  
 }  
}\* на этот раз после завершения работы главного потока, программа завершилась

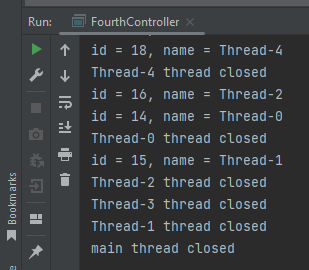
Создаем группу потоков, помещая их в контейнер, в данном случае – массив

package itStep.yandr.javaLessons.lesson42.controller;  
  
import itStep.yandr.javaLessons.lesson42.model.SecondThread;  
  
public class FourthController {  
 public static void main(String[] args) {  
 SecondThread threadLogic = new SecondThread();  
 int count = 5;  
 Thread[] threads = new Thread[count];  
 for (int i = 0; i < threads.length; i++) {  
 threads[i] = new Thread(threadLogic);  
 threads[i].start();  
 }

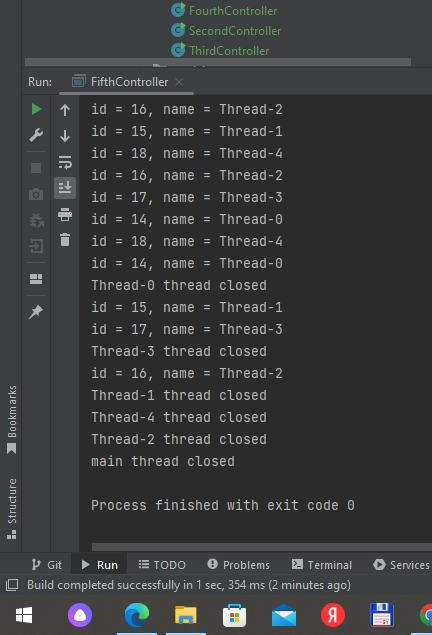
System.*out*.println("\nmain thread closed");

}  
}

Составим сценарий в котором главный поток будет проверять дочерние потоки на “жив” При каждой итерации главный поток, будет пробегать по массиву, проверяя жив ли поток, если хоть один жив, главный поток засыпает на 1 секунду иначе выходим из цикла.

package itStep.yandr.javaLessons.lesson42.controller;  
  
import itStep.yandr.javaLessons.lesson42.model.SecondThread;  
  
import java.util.concurrent.TimeUnit;  
  
public class FourthController {  
 public static void main(String[] args) throws InterruptedException {  
 SecondThread threadLogic = new SecondThread();  
 int count = 5;  
 Thread[] threads = new Thread[count];  
 for (int i = 0; i < threads.length; i++) {  
 threads[i] = new Thread(threadLogic);  
 threads[i].start();  
 }  
 while (true) {  
  
 if (threads[0].isAlive() || threads[1].isAlive() || threads[2].isAlive() || threads[3].isAlive()  
 || threads[4].isAlive()) {  
 TimeUnit.*SECONDS*.sleep(1);  
 } else {  
 break;  
 }  
 }  
 System.*out*.println("\nmain thread closed");   
 }  
}

Так мы сможем получить грамотную реализацию многопоточности, когда главный поток дожидается выполнения всех остальных и только потом заканчивает свою работу. Цель , заблокировать главный поток до момента окончания работы остальных потоков, эту возможность дает метод join, который блокирует поток выполнения(главный ), при выполнении потока на котором вызван.

package itStep.yandr.javaLessons.lesson42.controller;  
  
import itStep.yandr.javaLessons.lesson42.model.SecondThread;  
  
  
public class FifthController {  
 public static void main(String[] args) throws InterruptedException {  
 SecondThread threadLogic = new SecondThread();  
 int count = 5;  
 Thread[] threads = new Thread[count];  
 for (int i = 0; i < threads.length; i++) {  
 threads[i] = new Thread(threadLogic);  
 threads[i].start();  
 }  
 for (int i = 0; i < threads.length; i++) {  
 threads[i].join();  
 }  
 System.*out*.println("\nmain thread closed");  
 }  
}