

Конспект по мультипоточному программированию в Java

1. Первая лекция: база

Определения процесс, поток, создание потока (по классике), проблемы многопоточного программирования (взаимная блокировка и гонка с обсуждением), синхронизация (synchronized методы).

2. Вторая лекция: практика и дополнения

2.1. Понятие и синтаксис лямбда-функций

Лямбда-выражения в Java — это сокращённый синтаксис для реализации функциональных интерфейсов (интерфейсов с одним абстрактным методом). Они появились в Java 8. Синтаксис: (параметры) -> тело. Пример: Runnable r = () -> System.out.println("Привет из лямбды"); Если параметров нет — скобки пустые; если один параметр — скобки можно опустить; если несколько — обязательно в скобках. Тело может быть блоком: (x, y) -> { return x + y; } Важно: лямбда захватывает только effectively final переменные (те, что не меняются после инициализации).

2.1.1. Аннотация @FunctionalInterface и стандартные функциональные интерфейсы

Функциональные интерфейсы рекомендуется помечать аннотацией `@FunctionalInterface`, чтобы гарантировать наличие ровно одного абстрактного метода.

Пример:

```
@FunctionalInterface  
interface MyAction {  
    void execute();  
}
```

Java предоставляет встроенные функциональные интерфейсы (`java.util.function`):

- `Supplier<T>` — возвращает значение без аргументов: `() -> 42`
- `Consumer<T>` — принимает аргумент, ничего не возвращает: `x -> System.out.println(x)`
- `Function<T, R>` — принимает `T`, возвращает `R`: `x -> x.length()`
- `Predicate<T>` — возвращает `boolean`: `x -> x > 0`

2.2. Runnable

`Runnable` — стандартный интерфейс для задач без возвращаемого значения: `public interface Runnable { void run(); }` Используется повсеместно для передачи кода в потоки или исполнители. Пример:

```
Runnable task = new Runnable() {  
    public void run() {  
        System.out.println("Работаю!");  
    }  
};
```

Или короче через лямбду:

```
Runnable task = () -> System.out.println("Работаю!");
```

2.3. Создание потоков на базе Runnable

Чтобы запустить задачу в отдельном потоке, оберачиваем `Runnable` в `Thread`:

```
Thread t = new Thread(runnable);  
t.start(); // запускает новый поток
```

Пример:

```
Thread worker = new Thread(() -> {
    for (int i = 0; i < 3; i++) {
        System.out.println("Поток: " + i);
    }
});
worker.start();
```

Важно: вызов `run()` напрямую НЕ создаёт поток — это обычная синхронная функция.

2.4. Атомарные переменные

Атомарные классы (из `java.util.concurrent.atomic`) позволяют выполнять потокобезопасные операции без `synchronized`. Примеры:

```
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet(); // атомарно увеличивает на 1
counter.addAndGet(5); // атомарно прибавляет 5
counter.get(); // получает текущее значение
```

Эти операции используют CAS (compare-and-swap) на уровне процессора и не блокируют другие потоки.

2.4.1. LongAdder и LongAccumulator

При очень высокой нагрузке `AtomicLong` может быть менее эффективен, поэтому используют:

- `LongAdder` — быстрее при частых инкрементах:

```
LongAdder adder = new LongAdder();
adder.increment();
System.out.println(adder.sum());
```

- `LongAccumulator` — позволяет задавать операцию, например максимум:

```
LongAccumulator max = new LongAccumulator(Long::max, Long.MIN_VALUE);
max.accumulate(10);
max.accumulate(50);
System.out.println(max.get()); // 50
```

2.5. Специальные реализации коллекций

Обычные коллекции (`ArrayList`, `HashMap`) не потокобезопасны. Вместо них — специальные реализации:

- `ConcurrentHashMap`: `Map<String, Integer> map = new ConcurrentHashMap<>();`
- `CopyOnWriteArrayList`: `List<String> list = new CopyOnWriteArrayList<>();`
- Обёртка: `List<String> syncList = Collections.synchronizedList(new ArrayList<>());`

Пример использования:

```
ConcurrentHashMap<String, Integer> stats = new ConcurrentHashMap<>();
stats.compute("requests", (k, v) -> v == null ? 1 : v + 1); // потокобезопасно
```

Особенности `CopyOnWriteArrayList`: Эффективен при редком добавлении и частом чтении, так как при изменении создаётся новая копия массива. Плохо подходит для частых модификаций.

2.6. Диспетчеры потоков

`ExecutorService` — это пул потоков, управляющий выполнением задач. Создаётся через фабрику `Executors`. Основные типы:

- `newFixedThreadPool(n)` — фиксированный пул из n потоков.
- `newSingleThreadExecutor()` — один поток, задачи в очереди.
- `newCachedThreadPool()` — динамический пул, создаёт потоки по мере необходимости.
- `newScheduledThreadPool(n)` — пул с поддержкой отложенного и периодического выполнения.
- `newSingleThreadScheduledExecutor()` — однопоточный планировщик.

Примеры:

```
Executors.newFixedThreadPool(4);
Executors.newSingleThreadExecutor();
Executors.newCachedThreadPool();
ScheduledExecutorService s = Executors.newScheduledThreadPool(2);
Executors.newSingleThreadScheduledExecutor();

ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);

// Однократное выполнение через 1 секунду
scheduler.schedule(() -> System.out.println("Выполнено через 1 сек"), 1,
```

```
// Периодическое выполнение каждые 2 секунды (с задержкой 0)
scheduler.scheduleAtFixedRate(() -> System.out.println("Повтор каждые 2

// Завершение через 10 секунд (для примера)
scheduler.schedule(() -> scheduler.shutdown(), 10, TimeUnit.SECONDS);
```

Важно: всегда вызывать `shutdown()` после завершения работы, иначе JVM не завершится.

2.7. Future и получение результата задачи

Когда задача должна вернуть результат, вместо `Runnable` используется интерфейс `Callable<T>`, у которого метод `T call()` может возвращать значение и выбрасывать проверяемые исключения. Метод `submit(Callable)` возвращает объект `Future<T>`, через который можно получить результат (блокируя поток) или проверить статус.

Пример:

```
ExecutorService exec = Executors.newFixedThreadPool(2);
Callable<Integer> task = () -> { Thread.sleep(1000); return 42; };
Future<Integer> future = exec.submit(task);
try {
    Integer result = future.get(); // блокируется до завершения
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
exec.shutdown();
```

Дополнительно:

- `future.isDone()` — проверяет, завершена ли задача.
- `future.cancel(true)` — пытается отменить выполнение.
- `get(timeout, unit)` — версия с таймаутом, чтобы избежать бесконечного ожидания.

2.7.1. CompletableFuture: современное расширение Future

Позволяет строить цепочки задач и асинхронно обрабатывать результаты:

```

CompletableFuture.supplyAsync(() -> 21)
    .thenApply(x -> x * 2)
    .thenAccept(System.out::println);

ExecutorService pool = Executors.newFixedThreadPool(4);
CompletableFuture.supplyAsync(() -> 21, pool);

int result = future.join();
System.out.println(result); // 52

```

supplyAsync() без Executor использует ForkJoinPool.commonPool() (обычно параллельный пул потоков JVM).

2.8. Виртуальные потоки и структурированная конкурентность (Java 19+)

Виртуальные потоки (Thread.ofVirtual()) позволяют запускать тысячи лёгких потоков:

```

Thread t = Thread.ofVirtual().start(() -> System.out.println("Лёгкий поток"));
t.join();

ExecutorService virtualPool = Executors.newThreadPerTaskExecutor(Thread.ofVirtual());

virtualPool.submit(() -> System.out.println("Виртуальный поток работает"));
virtualPool.submit(() -> {
    try { Thread.sleep(500); } catch (InterruptedException e) {}
    System.out.println("Вторая задача");
});

virtualPool.shutdown();
virtualPool.awaitTermination(1, TimeUnit.SECONDS);

CompletableFuture.runAsync(() -> System.out.println("Виртуальный поток работает"));
    .thenAccept(() -> Executors.newThreadPerTaskExecutor(Thread.ofVirtual()));

```

```
ExecutorService carrierPool = Executors.newFixedThreadPool(5);
ExecutorService virtualPool = Executors.newThreadPerTaskExecutor(
    Thread.ofVirtual().factory().withExecutor(carrierPool)
);
```

С Java 21 в режиме превью появилась структурированная конкурентность (`StructuredTaskScope`) — удобная альтернатива ручному управлению пулами потоков.

2.9. Пример

Комплексный пример: подсчёт суммы с использованием пула потоков и атомарной переменной.

```
AtomicLong total = new AtomicLong(0);
ExecutorService pool = Executors.newFixedThreadPool(3);
int[] data = {10, 20, 30, 40};

for (int x : data) {
    pool.submit(() -> total.addAndGet(x));
}

pool.shutdown();
pool.awaitTermination(5, TimeUnit.SECONDS);
System.out.println("Итог: " + total.get());
```

Обратите внимание: этот код корректен, но неэффективен при большом числе элементов (лучше группировать задачи). Также напомните, что `x` — effectively final, поэтому захват в лямбду корректен.