Многопоточность

Лекция 11: Параллельное выполнение

Преподаватель: [Ваше имя]

Группа: 203

Семестр: Осенний 2024

План лекции

1. **Что такое многопоточность?**
2. **Threads и Runnable**
3. **Синхронизация**
4. **Concurrent collections**
5. **Executor Framework**
6. **Практический пример: Многопоточность в игре**

Что такое многопоточность?

Определение:

Многопоточность — способность программы выполнять несколько задач одновременно.

Преимущества:

- **Повышение производительности** на многоядерных системах
- **Отзывчивость** пользовательского интерфейса
- **Эффективное использование** ресурсов
- **Параллельная обработка** данных

Применение в играх:

Threads и Runnable

Создание потока:

```
// Способ 1: Наследование от Thread
public class GameAIThread extends Thread {
    private GameEngine gameEngine;
    private boolean running;

    public GameAIThread(GameEngine gameEngine) {
        this.gameEngine = gameEngine;
        this.running = true;
    }

    @Override
    public void run() {
        while (running) {
            try {
                // Выполнение ИИ
                gameEngine.performAITurn();

                // Пауза между ходами
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }
    }

    public void stopThread() {
        running = false;
        interrupt();
    }
```

Runnable интерфейс

```
// Способ 2: Реализация Runnable
public class ResourceLoader implements Runnable {
    private String resourcePath;
    private ResourceLoadCallback callback;

    public ResourceLoader(String resourcePath, ResourceLoadCallback callback) {
        this.resourcePath = resourcePath;
        this.callback = callback;
    }

    @Override
    public void run() {
        try {
            // Загрузка ресурса
            Resource resource = loadResource(resourcePath);

            // Уведомление о завершении
            callback.onResourceLoaded(resource);

        } catch (Exception e) {
            callback.onResourceLoadError(e);
        }
    }

    private Resource loadResource(String path) {
        // Имитация загрузки
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return new Resource(path);
    }
}

// Использование
Thread loaderThread = new Thread(new ResourceLoader("textures/unit.png", callback));
loaderThread.start();
```

Lambda выражения для потоков

```
// Современный способ создания потоков
public class ThreadManager {

    public void startAITurn(GameEngine gameEngine) {
        Thread aiThread = new Thread(() -> {
            try {
                while (!Thread.currentThread().isInterrupted()) {
                    gameEngine.performAITurn();
                    Thread.sleep(1500);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        aiThread.setName("AI-Thread");
        aiThread.start();
    }

    public void startResourceLoading(List<String> resources, ResourceLoadCallback callback) {
        Thread loaderThread = new Thread(() -> {
            resources.forEach(resource -> {
                try {
                    Resource loadedResource = loadResource(resource);
                    callback.onResourceLoaded(loadedResource);
                } catch (Exception e) {
                    callback.onResourceLoadError(e);
                }
            });
        });

        loaderThread.setName("Resource-Loader");
        loaderThread.start();
    }
}
```

Синхронизация потоков

Проблема гонки данных:

```
public class GameState {
    private int gold = 1000;
    private int units = 0;

    // ❌ Проблема: гонка данных
    public void addGold(int amount) {
        gold += amount; // Не атомарная операция
    }

    public void removeGold(int amount) {
        gold -= amount; // Не атомарная операция
    }

    public int getGold() {
        return gold; // Может вернуть устаревшее значение
    }
```

Volatile и атомарные типы

Volatile переменные:

```
public class GameEngine {
    private volatile boolean gameRunning = true;
    private volatile int currentTurn = 1;

    public void stopGame() {
        gameRunning = false; // Изменение видно всем потокам
    }

    public boolean isGameRunning() {
        return gameRunning; // Всегда актуальное значение
    }

    public void nextTurn() {
        currentTurn++; // Атомарная операция
    }
}
```

Concurrent collections

Thread-safe коллекции:

```
import java.util.concurrent.*;

public class GameWorld {
    // Потокобезопасные коллекции
    private ConcurrentHashMap<Position, Unit> unitPositions = new ConcurrentHashMap<>();
    private CopyOnWriteArrayList<GameEvent> eventQueue = new CopyOnWriteArrayList<>();
    private BlockingQueue<GameCommand> commandQueue = new LinkedBlockingQueue<>();

    public void addUnit(Unit unit, Position position) {
        unitPositions.put(position, unit);
    }

    public void removeUnit(Position position) {
        unitPositions.remove(position);
    }

    public Unit getUnitAt(Position position) {
        return unitPositions.get(position);
    }

    public void addEvent(GameEvent event) {
        eventQueue.add(event);
    }

    public void addCommand(GameCommand command) {
        try {
            commandQueue.put(command); // Блокирующая операция
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public GameCommand getNextCommand() {
        try {
            return commandQueue.take(); // Блокирующая операция
```

Executor Framework

ThreadPoolExecutor:

```
public class GameThreadManager {
    private ExecutorService aiExecutor;
    private ExecutorService resourceExecutor;
    private ScheduledExecutorService gameLoopExecutor;

    public GameThreadManager() {
        // Пул потоков для ИИ
        aiExecutor = Executors.newFixedThreadPool(2);

        // Пул потоков для загрузки ресурсов
        resourceExecutor = Executors.newCachedThreadPool();

        // Планировщик для игрового цикла
        gameLoopExecutor = Executors.newScheduledThreadPool(1);
    }

    public void startAITurn(GameEngine gameEngine) {
        aiExecutor.submit(() -> {
            try {
                gameEngine.performAITurn();
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }

    public void loadResource(String resourcePath, ResourceLoadCallback callback) {
        resourceExecutor.submit(() -> {
            try {
                Resource resource = loadResource(resourcePath);
                callback.onResourceLoaded(resource);
            } catch (Exception e) {
                callback.onResourceLoadError(e);
            }
        });
    }

    public void startGameLoop(Runnable gameLoop) {
        gameLoopExecutor.scheduleAtFixedRate(gameLoop, 0, 16, TimeUnit.MILLISECONDS);
    }

    public void shutdown() {
        aiExecutor.shutdown();
        resourceExecutor.shutdown();
        gameLoopExecutor.shutdown();

        try {
            if (!aiExecutor.awaitTermination(5, TimeUnit.SECONDS)) {
```

CompletableFuture для асинхронных операций

```
public class AsyncGameManager {

    public CompletableFuture<GameState> loadGameAsync(String filename) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(2000); // Имитация загрузки
                return new GameState(filename);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new RuntimeException("Загрузка прервана", e);
            }
        });
    }

    public CompletableFuture<List<Resource>> loadResourcesAsync(List<String> resourcePaths) {
        List<CompletableFuture<Resource>> futures = resourcePaths.stream()
            .map(path -> CompletableFuture.supplyAsync(() -> loadResource(path)))
            .collect(Collectors.toList());

        return CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
            .thenApply(v -> futures.stream()
                .map(CompletableFuture::join)
                .collect(Collectors.toList()));
    }

    public void processGameAsync(String filename, GameLoadCallback callback) {
        loadGameAsync(filename)
            .thenAccept(gameState -> {
                // Обработка в UI потоке
                Platform.runLater(() -> callback.onGameLoaded(gameState));
            })
            .exceptionally(throwable -> {
                Platform.runLater(() -> callback.onGameLoadError(throwable));
                return null;
            });
    }

    private Resource loadResource(String path) {
        try {
            Thread.sleep(1000); // Имитация загрузки
            return new Resource(path);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("Загрузка ресурса прервана", e);
        }
    }
}
```

Практический пример: Многопоточность в игре

```
public class MultiThreadedGameEngine {
    private final GameState gameState;
    private final ExecutorService aiExecutor;
    private final ExecutorService animationExecutor;
    private final ScheduledExecutorService gameLoopExecutor;
    private final BlockingQueue<GameCommand> commandQueue;

    private volatile boolean gameRunning = true;
    private volatile boolean aiThinking = false;

    public MultiThreadedGameEngine() {
        this.gameState = new GameState();
        this.aiExecutor = Executors.newFixedThreadPool(2);
        this.animationExecutor = Executors.newCachedThreadPool();
        this.gameLoopExecutor = Executors.newScheduledThreadPool(1);
        this.commandQueue = new LinkedBlockingQueue<>();

        startGameLoop();
        startAIThread();
        startCommandProcessor();
    }

    private void startGameLoop() {
        gameLoopExecutor.scheduleAtFixedRate(() -> {
            if (gameRunning) {
                updateGame();
                renderGame();
            }
        }, 0, 16, TimeUnit.MILLISECONDS); // 60 FPS
    }

    private void startAIThread() {
        aiExecutor.submit(() -> {
            while (gameRunning && !Thread.currentThread().isInterrupted()) {
                if (!aiThinking && gameState.isAITurn()) {
                    aiThinking = true;
                    performAITurn();
                    aiThinking = false;
                }
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        });
    }

    private void startCommandProcessor() {
        aiExecutor.submit(() -> {
            while (gameRunning && !Thread.currentThread().isInterrupted()) {
                try {
                    GameCommand command = commandQueue.take();
                    processCommand(command);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        });
    }

    public void submitCommand(GameCommand command) {
        try {
            commandQueue.put(command);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void processCommand(GameCommand command) {
        switch (command.getType()) {
            case MOVE:
                processMoveCommand((MoveCommand) command);
                break;
            case ATTACK:
                processAttackCommand((AttackCommand) command);
                break;
            case BUILD:
                processBuildCommand((BuildCommand) command);
                break;
        }
    }

    private void processMoveCommand(MoveCommand command) {
        Unit unit = command.getUnit();
        Position target = command.getTarget();

        if (gameState.isValidMove(unit, target)) {
            // Анимация движения в отдельном потоке
            animationExecutor.submit(() -> {
                animateUnitMove(unit, unit.getPosition(), target);
                gameState.moveUnit(unit, target);
            });
        }
    }

    private void animateUnitMove(Unit unit, Position from, Position to) {
        // Анимация движения
        try {
            Thread.sleep(500); // Имитация анимации
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void performAITurn() {
        // ИИ думает в отдельном потоке
        List<Unit> aiUnits = gameState.getAIUnits();

        for (Unit unit : aiUnits) {
            if (unit.isAlive()) {
                Position bestMove = calculateBestMove(unit);
                if (bestMove != null) {
                    gameState.moveUnit(unit, bestMove);
                }
            }
        }

        gameState.endAITurn();
    }

    private Position calculateBestMove(Unit unit) {
        // Простой алгоритм ИИ
        try {
            Thread.sleep(100); // Имитация размышлений
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Логика выбора лучшего хода
        return findNearestEnemyPosition(unit);
    }

    private Position findNearestEnemyPosition(Unit unit) {
        // Поиск ближайшего врага
        return gameState.getEnemyPositions().stream()
            .min(Comparator.comparingDouble(pos ->
                unit.getPosition().getDistanceTo(pos)))
            .orElse(null);
    }

    public void shutdown() {
        gameRunning = false;
        aiExecutor.shutdown();
        animationExecutor.shutdown();
        gameLoopExecutor.shutdown();

        try {
            if (!aiExecutor.awaitTermination(5, TimeUnit.SECONDS)) {
                aiExecutor.shutdownNow();
            }
        } catch (InterruptedException e) {
            aiExecutor.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}
```

Обработка ошибок в многопоточности

```
public class ThreadExceptionHandler {

    public static void setDefaultExceptionHandler() {
        Thread.setDefaultUncaughtExceptionHandler((thread, throwable) -> {
            System.err.println("Необработанное исключение в потоке " + thread.getName());
            throwable.printStackTrace();

            // Логирование ошибки
            GameLogger.getInstance().logError("Thread error: " + throwable.getMessage(), throwable);
        });
    }

    public static void handleThreadException(Thread thread, Throwable throwable) {
        if (throwable instanceof InterruptedException) {
            // Обработка прерывания потока
            Thread.currentThread().interrupt();
        } else if (throwable instanceof RuntimeException) {
            // Обработка runtime ошибок
            GameLogger.getInstance().logError("Runtime error in " + thread.getName(), throwable);
        } else {
            // Обработка других ошибок
            GameLogger.getInstance().logError("Unexpected error in " + thread.getName(), throwable);
        }
    }
}

// Использование
public class SafeGameThread extends Thread {

    public SafeGameThread(Runnable target) {
        super(target);
        setUncaughtExceptionHandler(ThreadExceptionHandler::handleThreadException);
    }
}
```

Лучшие практики многопоточности

✅ Что делать:

- **Использовать Executor Framework** вместо прямого создания потоков
- **Применять потокобезопасные коллекции** для общих данных
- **Синхронизировать доступ** к изменяемым данным
- **Обрабатывать InterruptedException** корректно
- **Использовать volatile** для флагов состояния

❌ Чего избегать:

- **Создавать слишком много потоков** вручную
- **Игнорировать синхронизацию** при доступе к общим данным
- **Использовать Thread.stop()** (устаревший метод)

Домашнее задание

Задача 1:

Реализовать многопоточный ИИ для игры

Задача 2:

Создать систему асинхронной загрузки ресурсов

Задача 3:

Реализовать потокобезопасную очередь команд

Что дальше?

На следующей лекции:

- **Сетевое программирование**
- **TCP/UDP протоколы**
- **Клиент-сервер архитектура**
- **Многопользовательская игра**

Подготовка:

- Изучить главу 21-22 из учебника
- Выполнить домашнее задание
- Подготовить вопросы по текущей теме

Вопросы?

Контакты:

- **Email:** [ваш.email@university.edu]
- **Telegram:** [@username]
- **Офис:** [номер кабинета]

Следующая лекция: Сетевое программирование