Advanced Thread Management

Synchronization

Synchronization techniques in Java

Synchronization in Java is essential for ensuring that multiple threads can work together without interfering with each other.

Here are the key synchronization techniques used in Java:

- **Synchronized Methods**

  When a method is declared as synchronized, the thread holds the monitor for that method's object while it is executing. Other threads that try to call any synchronized method on the same object will be blocked until the monitor is released.

  ```
  public synchronized void synchronizedMethod() {
      // critical section code
  }
  ```

- **Synchronized Blocks**

  Synchronized blocks are used to synchronize only a part of the method. This can help improve performance by reducing the scope of the synchronized code.

  ```
  public void method() {
      synchronized (this) {
          // critical section code
      }
  }
  ```

- **Static Synchronization**

  Static methods can also be synchronized. When a static method is synchronized, the monitor is on the Class object associated with the class.

  ```
  public static synchronized void staticSynchronizedMethod() {
      // critical section code
  }
  ```

- **Locks (java.util.concurrent.locks)**

  Java provides more sophisticated control over synchronization with the Lock interface and its implementations like ReentrantLock.

  ReentrantLock

  Allows for more flexible locking, including the ability to try to acquire a lock, to acquire a lock interruptibly, and to acquire a lock with a timeout.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Example {
    private final Lock lock = new ReentrantLock();

    public void method() {
        lock.lock();
        try {
            // critical section code
        } finally {
            lock.unlock();
        }
    }
}
```

- **ReadWriteLock**

  Allows a pair of associated locks, one for read-only operations and one for writing. Multiple threads can acquire the read lock if no thread holds the write lock.

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Example {
    private final ReadWriteLock lock = new
ReentrantReadWriteLock();

    public void readMethod() {
        lock.readLock().lock();
        try {
            // read-only critical section code
        } finally {
            lock.readLock().unlock();
        }
    }

    public void writeMethod() {
        lock.writeLock().lock();
        try {
            // write critical section code
        } finally {
            lock.writeLock().unlock();
        }
    }
}
```

- **Volatile Keyword**

  The volatile keyword in Java ensures that the value of a variable is always read from and written to the main memory, not from the thread's local cache.

```
private volatile boolean flag = true;

public void method() {
    while (flag) {
        // do something
    }
}
```

- **Atomic Variables (java.util.concurrent.atomic)**

  Atomic classes provide a way to perform atomic operations on single variables without using synchronization. Examples include AtomicInteger, AtomicLong, and AtomicReference.

```
import java.util.concurrent.atomic.AtomicInteger;

public class Example {
    private AtomicInteger counter = new AtomicInteger(0);

    public void increment() {
        counter.incrementAndGet();
    }

    public int getValue() {
        return counter.get();
    }
}
```

- **Concurrent Collections**

  Java provides thread-safe collections in the java.util.concurrent package, like ConcurrentHashMap, CopyOnWriteArrayList, and CopyOnWriteArraySet.

```
import java.util.concurrent.ConcurrentHashMap;

public class Example {
    private ConcurrentHashMap<String, Integer> map = new
ConcurrentHashMap<>();

    public void put(String key, Integer value) {
        map.put(key, value);
    }

    public Integer get(String key) {
        return map.get(key);
    }
}
```

Java offers various mechanisms to handle synchronization and ensure thread safety, ranging from simple synchronized blocks to more advanced constructs like locks and atomic variables. Choosing the right synchronization technique depends on the specific requirements and the need for flexibility and performance in your application.

Locks, Monitors, and synchronized blocks

Java provides several mechanisms to manage synchronization and ensure thread safety in concurrent programming. The primary tools are locks, monitors, and synchronized blocks.

Here's a detailed look at each of these:

1. **Monitors**

   In Java, each object can act as a monitor. A monitor is a synchronization mechanism that allows threads to have mutual exclusion and the ability to wait for a certain condition to become true. Monitors are used implicitly in synchronized methods and blocks.

   Key Points:

   - Each object in Java has an intrinsic lock or monitor.
   - The monitor is automatically associated with an object when a synchronized method or synchronized block is used.
   - Only one thread can hold the monitor of an object at any given time.

2. **Synchronized Blocks**

   A synchronized block in Java is a way to control access to a shared resource by multiple threads. It uses the intrinsic lock of an object to ensure that only one thread can execute the synchronized block of code at any given time.

```
public void method() {
    synchronized (this) {
        // critical section code
    }
}
```

   Advantages:

   - Allows for more granular control over synchronization compared to synchronized methods.
   - Can synchronize on any object, not just the current instance (this).

```
public class Counter {
    private int count = 0;

    public void increment() {
        synchronized (this) {
            count++;
        }
    }

    public int getCount() {
        return count;
    }
}
```

3. **Locks**

Locks provide a more flexible and sophisticated mechanism for synchronization compared to synchronized blocks. The java.util.concurrent.locks package provides several lock implementations, including ReentrantLock.

ReentrantLock:

A ReentrantLock is a type of lock that allows the thread that currently holds the lock to re-acquire it without causing a deadlock.

Key Methods:

- lock(): Acquires the lock.
- unlock(): Releases the lock.
- tryLock(): Attempts to acquire the lock without waiting indefinitely.
- lockInterruptibly(): Acquires the lock unless the thread is interrupted.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Counter {
    private final Lock lock = new ReentrantLock();
    private int count = 0;

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }

    public int getCount() {
        return count;
    }
}
```

Advantages:

-
- More flexible lock acquisition and release compared to synchronized blocks.
- Can attempt to acquire the lock with a timeout.
- Allows for interruptible lock acquisition.

Comparison: Synchronized Blocks vs. Locks

Feature	**Synchronized Blocks**	**Locks (ReentrantLock)**
Scope	Method-level or block-level	Block-level only
Flexibility	Less Flexible	More flexible
Lock Acquisition	Automatic	Manual
Interruptible	No	Yes (‘lockInterruptibly()’)
Timeout	No	Yes (‘tryLock(long timeout, TimeUnit unit)’)
Fairness	No	Yes (fair mode)

Monitors are intrinsic locks associated with every Java object, used implicitly in synchronized methods and blocks.

Synchronized blocks provide a simple way to ensure mutual exclusion, allowing fine-grained control over synchronization.

Locks, particularly ReentrantLock, offer greater flexibility and more control over synchronization, including features like interruptible and timed lock acquisition.

Understanding and choosing the appropriate synchronization mechanism is crucial for designing robust, efficient, and deadlock-free concurrent applications in Java.

Java Concurrent Application

Introduction to Java's concurrency utilities (java.util.concurrent package)

Java's java.util.concurrent package provides a robust set of utilities to handle concurrency, making it easier to manage multiple threads, synchronize access to shared resources, and improve the overall performance and reliability of concurrent applications.

Here's an introduction to the key components of this package:

1. **Executors Framework**

   The Executors framework simplifies the creation and management of thread pools. It provides factory methods for creating different types of executor services.

   Key Interfaces and Classes:

   - Executor: A simple interface for executing tasks.
   - ExecutorService: An extension of Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.
   - ScheduledExecutorService: An ExecutorService that can schedule commands to run after a given delay, or to execute periodically.

   Example:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor =
Executors.newFixedThreadPool(2);

        Runnable task1 = () -> System.out.println("Task 1
executed");
        Runnable task2 = () -> System.out.println("Task 2
executed");
```

```
        executor.submit(task1);
        executor.submit(task2);

        executor.shutdown();
    }
}
```

2. **Concurrent Collections**

   The java.util.concurrent package includes thread-safe collection classes that help prevent synchronization issues.

   Key Classes:

   - ConcurrentHashMap: A thread-safe version of HashMap.
   - CopyOnWriteArrayList: A thread-safe variant of ArrayList where all mutative operations are implemented by making a fresh copy of the underlying array.
   - CopyOnWriteArraySet: A thread-safe variant of Set.

   Example:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor =
Executors.newFixedThreadPool(2);

        Runnable task1 = () -> System.out.println("Task 1
executed");
        Runnable task2 = () -> System.out.println("Task 2
executed");

        executor.submit(task1);
        executor.submit(task2);

        executor.shutdown();
    }
}
```

3. **Locks and Synchronizers**

Java provides more advanced locking mechanisms beyond the synchronized keyword.

Key Classes:

- ReentrantLock: A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.
- ReentrantReadWriteLock: A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing.
- StampedLock: Offers an advanced locking mechanism with three modes for controlling access: writing, reading, and optimistic reading.

Example:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockExample {
    private final Lock lock = new ReentrantLock();

    public void performTask() {
        lock.lock();
        try {
            // critical section
        } finally {
            lock.unlock();
        }
    }
}
```

4. **Atomic Variables**

The java.util.concurrent.atomic package provides a set of classes that support lock-free thread-safe programming on single variables.

Key Classes:

- AtomicInteger: An int value that may be updated atomically.
- AtomicLong: A long value that may be updated atomically.
- AtomicReference: An object reference that may be updated atomically.

Example:

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicExample {
    private AtomicInteger counter = new AtomicInteger(0);

    public void increment() {
        counter.incrementAndGet();
    }

    public int getValue() {
        return counter.get();
    }
}
```

5. **Synchronizers**

Synchronizers are used to control the flow of multiple threads.

Key Classes:

- CountDownLatch: A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
- CyclicBarrier: A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.
- Semaphore: A counting semaphore that restricts the number of threads that can access a resource.
- Exchanger: A synchronization point where threads can exchange objects.

Example:

```
import java.util.concurrent.CountDownLatch;

public class LatchExample {
    public static void main(String[] args) throws
InterruptedException {
        CountDownLatch latch = new CountDownLatch(3);

        Runnable task = () -> {
            System.out.println("Task executed");
            latch.countDown();
        };

        new Thread(task).start();
        new Thread(task).start();
        new Thread(task).start();

        latch.await();  // Wait until the count reaches zero
        System.out.println("All tasks are completed");
    }
}
```

Java's java.util.concurrent package provides a comprehensive set of tools for handling concurrency, from managing thread pools with the Executors framework to providing thread-safe collections, advanced locking mechanisms, atomic variables, and various synchronizers. These utilities help in writing efficient, reliable, and maintainable concurrent code.

Using ExecutorService for managing threads

The ExecutorService interface in Java provides a higher-level replacement for working with threads directly. It manages a pool of worker threads and allows you to run tasks asynchronously.

Here's how to use ExecutorService for managing threads effectively.

1. **Creating an ExecutorService**

   You can create different types of ExecutorService instances using the factory methods in the Executors class:

   - Fixed Thread Pool: A fixed number of threads.
   - Cached Thread Pool: Creates new threads as needed but reuses previously constructed threads when available.
   - Single Thread Executor: A single worker thread.
   - Scheduled Thread Pool: A pool that can schedule commands to run after a given delay or periodically.

   Example:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceExample {
    public static void main(String[] args) {
        // Creating a fixed thread pool with 2 threads
        ExecutorService executor =
Executors.newFixedThreadPool(2);

        Runnable task1 = () -> System.out.println("Task 1
executed by " + Thread.currentThread().getName());
        Runnable task2 = () -> System.out.println("Task 2
executed by " + Thread.currentThread().getName());
        Runnable task3 = () -> System.out.println("Task 3
executed by " + Thread.currentThread().getName());

        executor.submit(task1);
        executor.submit(task2);
        executor.submit(task3);

        executor.shutdown();  // Initiates an orderly shutdown
    }
}
```

2. **Submitting Tasks**

You can submit tasks to the ExecutorService using the submit method. This method returns a Future object, which can be used to retrieve the result of the computation or to check the status of the task.

Example:

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor =
Executors.newFixedThreadPool(2);

        Callable<String> task = () -> {
            Thread.sleep(2000);
            return "Task's execution";
        };

        Future<String> future = executor.submit(task);

        System.out.println("Task submitted");
        String result = future.get();  // Blocks until the
task is completed
        System.out.println("Task result: " + result);

        executor.shutdown();
    }
}
```

3. **Scheduling Tasks**

   The ScheduledExecutorService can be used to schedule tasks to run after a delay or periodically.

   Example:

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(1);

        Runnable task = () -> System.out.println("Scheduled
task executed at " + System.currentTimeMillis());

        // Schedule a task to run after 5 seconds
        scheduler.schedule(task, 5, TimeUnit.SECONDS);

        // Schedule a task to run periodically every 3 seconds
with an initial delay of 1 second
        scheduler.scheduleAtFixedRate(task, 1, 3,
TimeUnit.SECONDS);
    }
}
```

4. **Shutting Down ExecutorService**

   It's important to shut down the ExecutorService to free up system resources. There are two main methods for shutting down:

   - shutdown(): Initiates an orderly shutdown in which previously submitted tasks are executed but no new tasks will be accepted.
   - shutdownNow(): Attempts to stop all actively executing tasks and halts the processing of waiting tasks.

Example:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ShutdownExample {
    public static void main(String[] args) {
        ExecutorService executor =
Executors.newFixedThreadPool(2);

        Runnable task = () -> {
            try {
                Thread.sleep(2000);
                System.out.println("Task executed");
            } catch (InterruptedException e) {
                System.err.println("Task interrupted");
            }
        };

        executor.submit(task);
        executor.submit(task);

        executor.shutdown();

        try {
            if (!executor.awaitTermination(3,
TimeUnit.SECONDS)) {
                executor.shutdownNow();
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
        }
    }
}
```

Using ExecutorService simplifies thread management by providing a higher-level API for running tasks asynchronously. It handles the creation, management, and reuse of threads, and allows for scheduling and controlling task execution. This results in more readable, maintainable, and efficient concurrent code.

Concurrent Collections

Overview of concurrent collections (ConcurrentHashMap, ConcurrentLinkedQueue)

Java's java.util.concurrent package provides several thread-safe collection classes that are designed to be used in concurrent environments. These concurrent collections handle synchronization internally to provide better performance and avoid common concurrency issues such as race conditions.

Two of the most commonly used concurrent collections are ConcurrentHashMap and ConcurrentLinkedQueue.

1. **ConcurrentHashMap**

   ConcurrentHashMap is a thread-safe implementation of the Map interface. It allows concurrent read and write operations without the need for explicit synchronization.

   Key Features:

   - Thread Safety: Multiple threads can safely access and modify the map.
   - High Performance: Internally uses a technique called lock striping to minimize contention and improve performance.
   - Non-blocking Reads: Reads do not require locking and are generally non-blocking.

   Common Methods:

   - put(K key, V value): Associates the specified value with the specified key in the map.
   - get(Object key): Returns the value to which the specified key is mapped, or null if the map contains no mapping for the key.
   - remove(Object key): Removes the key (and its corresponding value) from the map.
   - replace(K key, V oldValue, V newValue): Replaces the entry for a key only if currently mapped to a given value.
   - computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction): Computes a value if the specified key is not already associated with a value (or is mapped to null).

Example:

```
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new
ConcurrentHashMap<>();

        // Adding elements
        map.put("one", 1);
        map.put("two", 2);
        map.put("three", 3);

        // Retrieving elements
        System.out.println("Value for key 'one': " +
map.get("one"));

        // Removing elements
        map.remove("two");

        // Replacing elements
        map.replace("three", 3, 33);

        // Iterating over elements
        map.forEach((key, value) -> System.out.println(key +
": " + value));
    }
}
```

2. **ConcurrentLinkedQueue**

ConcurrentLinkedQueue is a thread-safe implementation of the Queue interface. It is an unbounded, thread-safe, non-blocking FIFO (first-in-first-out) queue.

Key Features:

- Non-blocking: Uses lock-free algorithms to ensure high throughput and low latency.
- Thread Safety: Multiple threads can safely add and remove elements concurrently.
- Unbounded: The queue grows dynamically as needed, without any predefined size limit.

Common Methods:

- offer(E e): Inserts the specified element at the tail of the queue.
- poll(): Retrieves and removes the head of the queue, or returns null if the queue is empty.

- peek(): Retrieves, but does not remove, the head of the queue, or returns null if the queue is empty.
- size(): Returns the number of elements in the queue.

Example:

```
import java.util.concurrent.ConcurrentLinkedQueue;

public class ConcurrentLinkedQueueExample {
    public static void main(String[] args) {
        ConcurrentLinkedQueue<String> queue = new
ConcurrentLinkedQueue<>();

        // Adding elements
        queue.offer("first");
        queue.offer("second");
        queue.offer("third");

        // Retrieving elements
        System.out.println("Head of queue: " + queue.peek());

        // Removing elements
        System.out.println("Removed element: " +
queue.poll());
        System.out.println("Removed element: " +
queue.poll());

        // Checking remaining elements
        System.out.println("Head of queue after removals: " +
queue.peek());
        System.out.println("Size of queue: " + queue.size());

        // Iterating over elements
        queue.forEach(System.out::println);
    }
}
```

In conclusion.

- **ConcurrentHashMap**: A high-performance, thread-safe implementation of the Map interface, ideal for scenarios where concurrent read and write access to a map is required.
- **ConcurrentLinkedQueue**: A non-blocking, thread-safe implementation of the Queue interface, suitable for scenarios where a FIFO ordering of elements is necessary and multiple threads need to add or remove elements concurrently.

Both collections are designed to handle high levels of concurrency with minimal contention and are a vital part of building robust and scalable concurrent applications in Java.