Java Concurrency and Performance

TRAINING

contents

mobile:+91.9880951838
mailto:mohit.riverstone@gmail.com
website:www.movaatechnologies.com

Java Concurrency and Performance

TRAINING

course contents

- Producer Consumer(Basic Hand-Off) Day1
- Common Issues with thread
- Java Memory Model(JMM)
- Applied Threading techniques
- Building Blocks for Highly Concurrent Design
- Highly Concurrent Data Structures-Part1 Day2
- Designing For Concurrency
- Sharing Objects
- Composing Objects
- Canned Synchronizers
- Structuring Concurrent Applications
- Cancellation and Shutdown Day3
- Applying Thread Pools
- Liveness, Performance, and Testing
- Performance and Scalability
- Explicit Locks
- Building Custom Synchronizers Day4
- Atomic Variables and Nonblocking Synchronization
- Fork and Join Framework
- Crash course in Mordern hardware
- Designing for multi-core/processor environment
- Highly Concurrent Data Structures-Part2

mobile:+91.9880951838
mailto:mohit.riverstone@gmail.com
website:www.movaatechnologies.com

Description:

- With the advent of multi-core processors the usage of single threaded programs is soon becoming obsolete.Java was built to be able to do many things at once. In computer lingo, we call this "concurrency". This is the main reason why Java is so useful. Today we see a lot of our applications running on multiple cores, concurrent java programs with multiple threads is the answer for effective performance and stability on multi-core based applications. Concurrency is among the utmost worries for newcomers to Java programming but there's no reason to let it deter you. Not only is excellent documentation available but also pictorial representations of each topic to make understanding much graceful and enhanced. Java threads have become easier to work with as the Java platform has evolved. In order to learn how to do multithreaded programming in Java 6 and 7, you need some building blocks. Our training expert with his rich training and consulting experience illustrates with real application based case studies.

Intended Audience:

- The target group is programmers who want to know foundations of concurrent programming and existing concurrent programming environments, in order, now or in future, to develop multithreaded applications for multi-core processors and shared memory multiprocessors.

Key Skills:

- Dealing with threads and collections on a multi-core/ multiprocessor.
- To quickly identify the root cause of poor performance in your applications.
- Eliminate conditions that will prevent you from finding performance bottlenecks.
- JDK 5, 6, 7 which have features to harness the power of the underlying hardware.

Prerequisites:

- Basic knowledge of Java (introductory course or equivalent practical experience).

Instructional Method:

- This is an instructor led course provides lecture topics and the practical application of JEE5.0 and the underlying technologies. It pictorially presents most concepts and there is a detailed case study that strings together the technologies, patterns and design.

mobile:+91.9880951838
mailto:mohit.riverstone@gmail.com
website:www.movaatechnologies.com

Java Concurrency and Performance

- ***Producer Consumer(Basic Hand-Off)***
  - Why wait-notify require Synchronization
    - notifyAll used as work around
    - Structural modification to hidden queue by wait-notify
    - locking handling done by OS
    - use cases for notify-notifyAll
    - Hidden queue
    - design issues with synchronization
- ***Common Issues with thread***
  - Uncaught Exception Handler
  - problem with stop
  - Dealing with InterruptedStatus
- ***Java Memory Model(JMM)***
  - Real Meaning and effect of synchronization
  - Volatile
  - Sequential Consistency would disallow common optimizations
  - The changes in JMM
  - Final
  - Shortcomings of the original JMM
    - Finals not really final
    - Prevents effective compiler optimizations
    - Processor executes operations out of order
    - Compiler is free to reorder certain instructions
    - Cache reorders writes
    - Old JMM surprising and confusing
  - Instruction Reordering
    - What is the limit of reordering
    - Programmatic Control
    - super-scalar processors

mobile:+91.9880951838
mailto:mohit.riverstone@gmail.com
website:www.movaatechnologies.com

- heavily pipelines processors
- As-if-serial-semantics
- Why is reordering done

- Cache Coherency
  - Write-back Caching explained
  - What is cache Coherence.
  - How does it effect java programs.
  - Software based Cache Coherency
  - NUMA(Non uniform memory access)
  - Caching explained
  - Cache incoherency
- New JMM and goals of JSR-133
  - Simple,intuitive and, feasible
  - Out-of-thin-air safety
  - High performance JVM implementations across architectures
  - Minimal impact on existing code
  - Initialization safety
  - Preserve existing safety guarantees and type-safety

- ***Applied Threading techniques***
  - Safe Construction techniques
  - Thread Local Storage
  - Thread safety levels
  - UnSafe Construction techniques
- ***Building Blocks for Highly Concurrent Design***
  - CAS
    - Wait-free Queue implementation
    - Optimistic Design
    - Wait-free Stack implementation
    - Hardware based locking
      - ABA problem
        - Markable reference

        - weakCompareAndSet
        - Stamped reference
    - Reentrant Lock
        - ReentrantReadWriteLock
        - ReentrantLock
    - Lock Striping
        - Lock Striping on LinkNodes
        - Lock Striping on table
    - Indentifying scalability bottlenecks in java.util.Collection
        - segregating them based on Thread safety levels
    - Lock Implementation
        - Multiple user conditions and wait queues
        - Lock Polling techniques
        - Based on CAS
        - Design issues with synchronization
- ***Highly Concurrent Data Structures-Part1***
    - Weakly Consistent Iterators vs Fail Fast Iterators
    - ConcurrentHashMap
        - Structure
        - remove/put/resize lock
        - Almost immutability
        - Using volatile to detect interference
        - Read does not block in common code path
- ***Designing For Concurrency***
    - Atomicity
    - Confinement
    - Immutability
    - Visibility
    - Almost Immutability
    - Restructuring and refactoring
- ***Sharing Objects***

- Thread confinement
  - Stack confinement
  - ThreadLocal
  - Unshared objects are safe
  - Ad-hoc thread confinement
- Visibility
  - Synchronization and visibility
  - Non-atomic 64-bit numeric operations
  - Problems that state data can cause
  - Volatile vs synchronized
  - Single-threaded write safety
  - Volatile flushing
  - Making fields visible with volatile
  - Reason why changes are not visible
- Immutability
  - Definition of immutable
  - Immutable is always thread safe
  - Immutable containing mutable object
  - Final fields
- Safe publication
  - Making objects and their state visible
  - Safe publication idioms
  - How to share objects safely
  - "Effectively" immutable objects
- Publication and escape
  - Publishing objects to alien methods
  - Publishing objects as method returns
  - Implicit links to outer class
  - Ways we might let object escape
  - Publishing objects via fields
- ***Composing Objects***

mobile:+91.9880951838
mailto:mohit.riverstone@gmail.com
website:www.movaatechnologies.com

- Instance confinement
  - Split locks
  - Example of fleet management
  - Java monitor pattern
  - Lock confinement
  - Encapsulation
  - How instance confinement is good
  - State guarded by private fields
- Documenting synchronization policies
  - Examples from the JDK
  - Documentation checklist
  - What should be documented
  - Synchronization policies
  - Interpreting vague documentation
- Adding functionality to existing thread-safe classes
  - Benefits of reuse
  - Using composition to add functionality
  - Subclassing to add functionality
  - Modifying existing code
  - Client-side locking
- Designing a thread-safe class
  - Pre-condition
  - Thread-safe counter with invariant
  - Primitive vs object fields
  - Encapsulation
  - Post-conditions
  - Waiting for pre-condition to become true
- Delegating thread safety
  - Independent fields
  - Publishing underlying fields
  - Delegating safety to ConcurrentMap

- Invariables and delegation
- Using thread safe components
- Delegation with vehicle tracker

- ***Canned Synchronizers***
  - Semaphore
  - Latches
  - SynchronousQueue
  - Future
  - Exchanger
  - Synchronous Queue Framework
  - Mutex
  - Barrier
- ***Structuring Concurrent Applications***
  - Finding exploitable parallelism
    - Callable controlling lifecycle
    - CompletionService
    - Limitations of parallelizing heterogeneous tasks
    - Callable and Future
    - Time limited tasks
    - Example showing page renderer with future
    - Sequential vs parallel
    - Breaking up a single client request
  - The Executor framework
    - Memory leaks with ThreadLocal
    - Delayed and periodic tasks
    - Thread pool structure
    - Motivation for using Executor
    - Executor lifecycle, state machine
    - Difference between java.util.Timer and ScheduledExecutor
    - ThreadPoolExecutor
    - Decoupling task submission from execution
    - Shutdown() vs ShutdownNow()

- Executor interface
- Thread pool benefits
- Standard ExecutorService configurations

  - Execution policies
    - Various sizing options for number of threads and queue length
    - In which order? (FIFO, LIFO, by priority)
    - Who will execute it?

  - Executing tasks in threads
    - Disadvantage of unbounded thread creation
    - Single-threaded vs multi-threaded
    - Explicitely creating tasks
    - Indepence of tasks
    - Identifying tasks
    - Task boundaries

- ***Cancellation and Shutdown***

  - Stopping a thread-based service
    - Graceful shutdown
    - ExecutorService shutdown
    - Providing lifecycle methods
    - Asynchronous logging caveats
    - Example: A logging service
    - Poison pills
    - One-shot execution service

  - Task cancellation
    - Cancellation policies
    - Using flags to signal cancellation
    - Reasons for wanting to cancel a task
    - Cooperative vs preemptive cancellation

    - Interruption
      - Origins of interruptions
      - WAITING state of thread

mobile:+91.9880951838
mailto:mohit.riverstone@gmail.com
website:www.movaatechnologies.com

- How does interrupt work?
- Methods that put thread in WAITING state
- Policies in dealing with InterruptedException
- Thread.interrupted() method

- Dealing with non-interruptible blocking
  - Interrupting locks
  - Reactions of IO libraries to interrupts
- Responding to interruption
  - Letting the method throw the exception
  - Saving the interrupt for later
  - Ignoring the interrupt status
  - Restoring the interrupt and exiting
- Interruption policies
  - Task vs Thread
  - Different meanings of interrupt
  - Preserving the interrupt status
- Example: timed run
  - Telling a long run to eventually give up
  - Canceling busy jobs

- Handling abnormal thread termination
  - Using UncaughtExceptionHandler
  - Dealing with exceptions in Swing
  - ThreadGroup for uncaught exceptions
- JVM shutdown
  - Shutdown hooks
  - Orderly shutdown
  - Daemon threads
  - Finalizers
  - Abrupt shutdown

- ***Applying Thread Pools***
  - Configuring ThreadPoolExecutor

- Thread factories
- corePoolSize
- Customizing thread pool executor after construction
- Using default Executors.new* methods
- Managing queued tasks
- maximumPoolSize
- keepAliveTime
- PriorityBlockingQueue

  - Saturation policies
    - Discard
    - Caller runs
    - Abort
    - Discard oldest

- Sizing thread pools
  - Examples of various pool sizes
  - Determining the maximum allowed threads on your operating system
  - CPU-intensiv vs IO-intensive task sizing
  - Danger of hardcoding worker number
  - Problems when pool is too large or small
  - Formula for calculating how many threads to use
  - Mixing different types of tasks
- Tasks and Execution Policies
  - Long-running tasks
  - Homogenous, independent and thread-agnostic tasks
  - Thread starvation deadlock
- Extending ThreadPoolExecutor
  - terminate
  - Using hooks for extension
  - afterExecute
  - beforeExecute

- Parallelizing recursive algorithms
  - Using Fork/Join to execute tasks
  - Converting sequential tasks to parallel
- ***Liveness, Performance, and Testing***
  - Avoiding Liveness Hazards
    - Other liveness hazards
      - Poor responsiveness
      - Livelock
      - Starvation
        - ReadWriteLock in Java 5 vs Java 6
        - Detecting thread starvation
    - Avoiding and diagnosing deadlocks
      - Adding a sleep to cause deadlocks
      - "TryLock" with synchronized
      - Using open calls
      - Verifying thread deadlocks
      - Avoiding multiple locks
      - Timed lock attempts
      - Stopping deadlock victims
      - DeadlockArbitrator
      - Deadlock analysis with thread dumps
      - Unit testing for lock ordering deadlocks
    - Deadlock
      - Thread-starvation deadlocks
      - Discovering deadlocks
      - Checking whether locks are held
      - Resource deadlocks
      - The drinking philosophers
      - Lock-ordering deadlocks
      - Defining a global ordering
      - Resolving deadlocks

mobile:+91.9880951838
mailto:mohit.riverstone@gmail.com
website:www.movaatechnologies.com

- Causing a deadlock amongst philosophers
- Deadlock between cooperating objects
- Imposing a natural order
- Dynamic lock order deadlocks
- Defining order on dynamic locks

  - Open calls and alien methods
    - Example in Vector

- ***Performance and Scalability***
  - Thinking about performance
    - Mistakes in traditional performance optimizations
    - 2-tier vs multi-tier
    - Evaluating performance tradeoffs
    - Performance vs scalability
    - Effects of serial sections and locking
    - How fast vs how much
  - Reducing lock contention
    - How to monitor CPU utilization
    - Performance comparisons
    - ReadWriteLock
    - Using CopyOnWrite collections
    - Immutable objects
    - Atomic fields
    - Using ConcurrentHashMap
    - Narrowing lock scope
    - Avoiding "hot fields"
    - Hotspot options for lock performance
    - Reasons why CPUs might not be loaded
    - How to find "hot locks"
    - Lock splitting
    - Dangers of object pooling
    - Safety first!
    - Reducing lock granularity

- Exclusive locks
  - Lock striping
    - In ConcurrentHashMap
    - In ConcurrentLinkedQueue
- Amdahl's and Little's laws
  - Formula for Amdahl's Law
  - Problems with Amdahl's law in practice
  - Applying Little's Law in practice
  - Utilization according to Amdahl
  - Maximum useful cores
  - How threading relates to Little's Law
  - Formula for Little's Law
- Costs introduced by threads
  - Context switching
  - Locking and unlocking
  - Cache invalidation
  - Spinning before actual blocking
  - Lock elision
  - Memory barriers
  - Escape analysis and uncontended locks

- ***Explicit Locks***
  - Lock and ReentrantLock
    - Using try-finally
    - Memory visibility semantics
    - Using try-lock to avoid deadlocks
    - tryLock and timed locks
    - Interruptible locking
    - Non-block-structured locking
    - ReentrantLock implementation
    - Using the explicit lock
  - Synchronized vs ReentrantLock

- Memory semantics
- Prefer synchronized
- Ease of use

- Performance considerations
  - Heavily contended locks
  - Java 5 vs Java 6 performance
  - Throughput on contended locks
  - Uncontended performance
- Fairness
  - Standard non-fair mechanisms
  - Throughput of fair locks
  - Round-robin by OS
  - Barging
  - Fair explicit locks in Java
- Read-write locks
  - ReadWriteLock interface
  - Understanding system to avoid starvation
  - ReadWriteLock implementation options
    - Release preference
    - Downgrading
    - Reader barging
    - Upgrading
    - Reentrancy

- ***Building Custom Synchronizers***
  - Explicit condition objects
    - Condition interface
    - Timed conditions
    - Benefits of explicit condition queues
  - AbstractQueuedSynchronizer (AQS)
    - Basis for other synchronizers
  - Managing state dependence

- Exceptions on pre-condition fails
- Structure of blocking state-dependent actions
- Crude blocking by polling and sleeping
- Example using bounded queues
- Single-threaded vs multi-threaded

  - Introducing condition queues
    - With intrinsic locks

- Using condition queues
  - Waking up too soon
  - Conditional waits
  - Condition queue
  - Encapsulating condition queues
  - State-dependence
  - notify() vs notifyAll()
  - Condition predicate
  - Lock
  - Waiting for a specific timeout

  - Missed signals
    - InterruptedException

- ***Atomic Variables and Nonblocking Synchronization***
  - Hardware support for concurrency
    - Using "Unsafe" to access memory directly
    - CAS support in the JVM
    - Compare-and-Set
    - Performance advantage of padding
    - Nonblocking counter
    - Simulation of CAS
    - Managing conflicts with CAS
    - Compare-and-Swap (CAS)
    - Shared cache lines
    - Optimistic locking

- Atomic variable classes
  - Optimistic locking classes
  - How do atomics work?
  - Atomic array classes
  - Performance comparisons: Locks vs atomics
  - Cost of atomic spin loops
  - Very fast when not too much contention
  - Types of atomic classes
- Disadvantages of locking
  - Priority inversion
  - Elimination of uncontended intrinsic locks
  - Volatile vs locking performance
- Nonblocking algorithms
  - Scalability problems with lock-based algorithms
  - Atomic field updaters
  - Doing speculative work
  - AtomicStampedReference
  - Nonblocking stack
  - Definition of nonblocking and lock-free
  - Highly scalable hash table
  - The ABA problem
- Using sun.misc.Unsafe
  - Dangers
  - Reasons why we need it

- ***Fork and Join Framework***
  - Fork -join decomposition
  - Fork and Join
  - ParallelArray
  - Divide and conquer
  - Hardware shapes programming idiom
  - Exposing fine grained parallelism

mobile:+91.9880951838
mailto:mohit.riverstone@gmail.com
website:www.movaatechnologies.com

- Anatomy of Fork and Join
- Limitations
- Work Stealing

- ***Crash course in Mordern hardware***
  - Amdahl's Law
  - Cache
    - cache controller
    - write
    - Direct mapped
    - read
    - Address mapping in cache
  - Memory Architectures
    - NUMA
    - UMA
- ***Designing for multi-core/processor environment***
  - Concurrent Stack
  - Harsh Realities of parallelism
  - Parallel Programming
  - Concurrent Objects
    - Sequential Consistency
    - Linearizability
    - Concurrency and Correctness
    - Progress Conditions
    - Quiescent Consistency
  - Concurrency Patterns
    - Lazy Synchronization
    - Lock free Synchronization
    - Optimistic Synchronization
    - Fine grained Synchronization
  - Priority Queues
    - Heap Based Unbounded Priority Queue

- Skiplist based Unbounded priority Queue
- Array Based bounded Priority Queue
- Tree based Bounded Priority Queue

- Lists
  - Coarse Grained Synchronization
  - Lazy Synchronization
  - Optimistic Synchronization
  - Non Blocking Synchronization
  - Fine Grained Synchronization
- Skiplists
- Spinlocks
  - Lock suitable for NUMA systems
- Concurrent Queues
  - Unbounded lock-free Queue
  - Bounded Partial Queue
  - Unbounded Total Queue
- Concurrent Hashing
  - Open Address Hashing
  - Closed Address Hashing
  - Lock Free Hashing

- ***Highly Concurrent Data Structures-Part2***
  - CopyOnWriteArray(List/Set)
- NonBlockingHashMap
  - For systems with more than 100 cpus/cores
  - State based Reasoning
  - all CAS spin loop bounded
  - Constant Time key-value mapping
  - faster than ConcurrentHashMap
  - no locks even during resize
- Queue interfaces
  - Queue

- BlockingQueue
- Deque
- BlockingDeque

- Queue Implementations
  - ArrayDeque and ArrayBlockingDeque
    - WorkStealing using Deques
  - LinkedBlockingQueue
  - LinkedBlockingDeque
  - ConcurrentLinkedQueue
    - GC unlinking
    - Michael and Scott algorithm
    - Tails and heads are allowed to lag
    - Support for interior removals
    - Relaxed writes
  - ConcurrentLinkedDeque
    - Same as ConcurrentLinkedQueue except bidirectional pointers
  - LinkedTransferQueue
    - Internal removes handled differently
    - Heuristics based spinning/blocking on number of processors
    - Behavior differs based on method calls
    - Usual ConcurrentLinkedQueue optimizations
    - Normal and Dual Queue
- Skiplist
  - Lock free Skiplist
  - Sequential Skiplist
  - Lock based Concurrent Skiplist
  - ConcurrentSkipListMap(and Set)
    - Indexes are allowed to race
    - Iteration
    - Problems with AtomicMarkableReference
    - Probabilistic Data Structure

- Marking and nulling
- Different Way to mark