Introduction to Parallel Computing (Java Concurrency)

2.5-hour intensive lecture

Shuhao Zhang

Huazhong University of Science and Technology (HUST)

shuhao_zhang[at]hust.edu.cn

shuhaozhangtony.github.io

January 2026

Logistics

- Format: lecture-only (no labs, no tutorials)
- Assumed background: basic Java, basic data structures
- Goal: build a practical mental model for writing correct concurrent Java programs
- We focus on Java 17+ APIs; no deep Java Memory Model (JMM) theory

Agenda

1. Motivation & mental model (parallel vs concurrent)
2. Threads: lifecycle, creating threads, joining
3. Correctness: races, `synchronized`, visibility
4. Coordination: `wait/notify` and blocking
5. Executors & thread pools: safe structure for concurrency
6. Patterns & pitfalls: producer-consumer, deadlock, starvation
7. More tools: locks & executor patterns
8. Synchronizers & concurrent collections
9. Debugging & operational tips
10. Performance intuition & wrap-up

Motivation & Mental Model

Why Parallel / Concurrent Computing?

- More cores are common; single-core speed is limited (power/heat)
- We want either:
  - **Lower latency**: respond faster
  - **Higher throughput**: do more work per second
- Real systems: web services, data processing, simulations, AI pipelines

Three Related Words

Concurrency

Multiple tasks make progress *overlapping in time* (may run on 1 core).

Parallelism

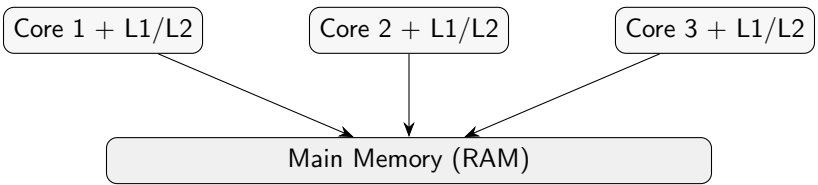
Multiple tasks run *at the same time* (requires multiple cores).

Distributed

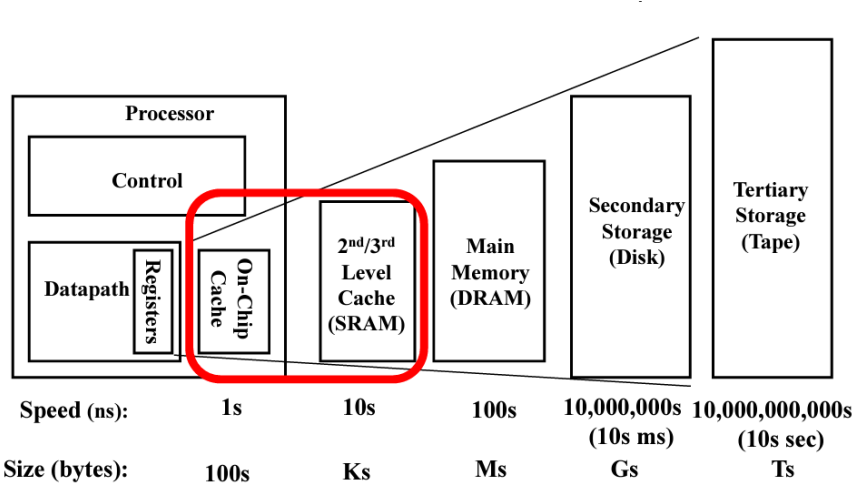
Tasks run on *different machines* (network + partial failures).

A Simple Hardware Picture

- CPU cores execute instructions
- Each core has private caches; all cores share main memory
- Key implication: shared memory is not instantly consistent



Memory Hierarchy (Why Performance is Non-Linear)



What Makes Concurrency Hard?

- **Non-determinism**: many valid interleavings
- **Shared mutable state**: races + visibility problems
- **Liveness hazards**: deadlock, starvation
- Debugging is difficult because bugs may disappear when you add logs

What We Will Cover (and Skip)

Cover

- Threads and interruption
- `synchronized`, `volatile`, atomics
- Executors and thread pools
- Common patterns + pitfalls

Skip / mention only

- Deep JMM formalism
- Fork/Join details
- Lock-free algorithm design
- Distributed systems topics

Core

A Rule of Thumb

Prefer simplicity over cleverness

If you can avoid sharing state, do it.

- Immutability and confinement are the easiest correctness tools
- When you must share state, synchronize deliberately

Java Threading Fundamentals

Creating Threads: Two Common Ways

- Extend `Thread` (simple, but less flexible)
- Implement `Runnable` / `Callable<T>` (preferred)

Key idea

A thread runs code; a task is the code to run.

Example: Minimal Thread Creation

```
public class HelloThreads {
  public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread(() ->
        System.out.println("Hello from " + Thread.currentThread().getName()));
    t.start();
    t.join();
    System.out.println("Done");
  }
}
```

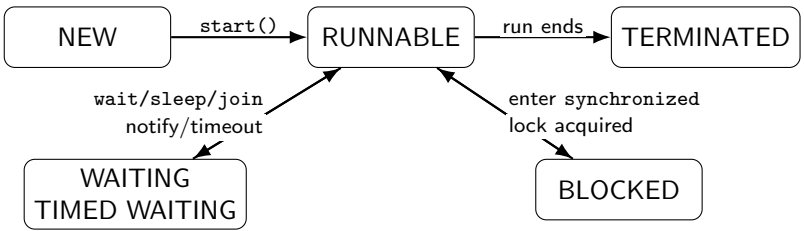
Thread Lifecycle (Practical View)

- **NEW** → created, not started
- **RUNNABLE** → eligible to run (or running)
- **BLOCKED / WAITING** → waiting for a lock/condition
- **TERMINATED** → finished

Debug hint

If your program "hangs", identify which threads are waiting, and on what.

Thread Lifecycle: State Diagram



Race Condition: The Classic Counter

```
public class RaceCondition {
  private static int counter = 0;

  public static void main(String[] args) throws InterruptedException {
    Runnable task = () -> {
      for (int i = 0; i < 100_000; i++) counter++; // not atomic
    };

    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);
    t1.start();
    t2.start();
    t1.join();
    t2.join();

    System.out.println("Expected 200000, actual " + counter);
  }
}
```

Why `counter++` Loses Updates

Thread 1
1) read c (=0)
2) add 1 (local=1)
3) write c (=1)

Thread 2
1) read c (=0)
2) add 1 (local=1)
3) write c (=1)

One possible interleaving:
T1: read 0 T2: read 0 T1: write 1 T2: write 1
Result: two increments, but final $c = 1$ (lost update)

Two Categories of Concurrency Bugs

Safety (Wrong result)

Race conditions, visibility issues, broken invariants.

Liveness (No progress)

Deadlock, starvation, livelock, thread leaks.

Cancellation and Interruption

- Java uses **cooperative cancellation**
- `Thread.interrupt()` sets an interrupt flag
- Blocking calls may throw `InterruptedException`

Rule

If you catch `InterruptedException`, either propagate it or restore the flag.

Interruption Pattern

```
try {
  while (!Thread.currentThread().isInterrupted()) {
    // do work
    Thread.sleep(50);
  }
} catch (InterruptedException e) {
  Thread.currentThread().interrupt(); // restore
}
```

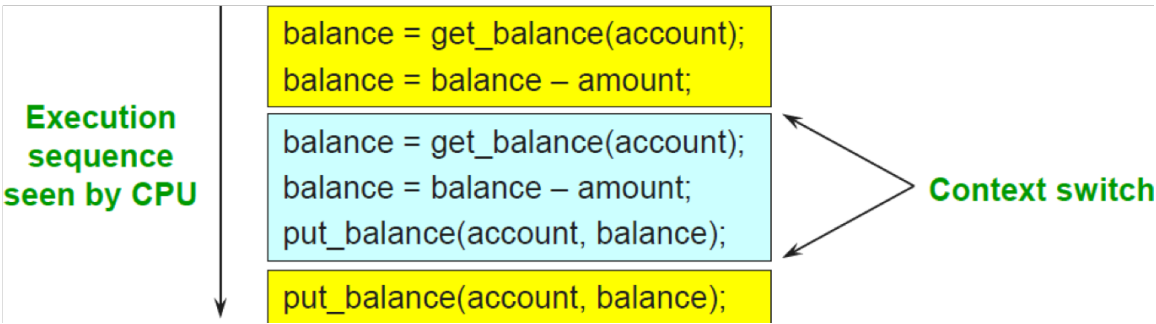
Synchronization Essentials

What Does Synchronization Solve?

- **Mutual exclusion**: prevent conflicting updates
- **Visibility**: make writes by one thread visible to others
- **Ordering**: constrain reordering across threads

A Typical Synchronization Bug (Diagram)

- Symptom: **occasionally wrong results** or **hangs**
- Cause: missing mutual exclusion / missing condition coordination



The Monitor Idea: `synchronized`

- Every Java object can act as a lock (monitor)
- `synchronized` enforces:
  - only one thread executes a critical section at a time
  - a visibility boundary when entering/exiting the monitor

Fixing the Counter with synchronized

```
public class SynchronizedCounter {
  private int counter = 0;

  public synchronized void inc() {
    counter++;
  }

  public synchronized int get() {
    return counter;
  }
}
```

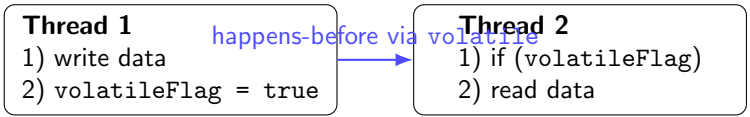
`volatile`: Visibility Without Mutual Exclusion

- `volatile` makes reads/writes visible across threads
- It does not make compound actions atomic
- Typical use: stop flags, configuration snapshots

Anti-pattern

`volatile int x; x++;` is still not atomic.

Visibility and Ordering: A Minimal Picture



Takeaway: `volatile` gives visibility + ordering for the flag
but does not make compound updates (like `counter++`) atomic.

Atomic Variables

- `AtomicInteger`, `AtomicLong`, `AtomicReference<T>`
- Provide atomic read-modify-write operations (CAS based)
- Often good for counters, IDs, statistics

Performance note

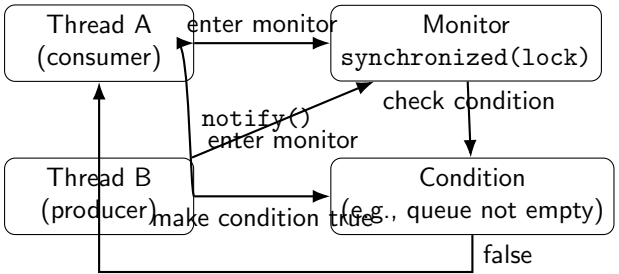
For highly contended counters, consider `LongAdder`.

Waiting for a Condition: `wait/notify`

- Use when a thread must wait until a condition becomes true
- Always call `wait()` inside a loop checking the condition

```
synchronized (lock) {
  while (!condition) {
    lock.wait();
  }
  // proceed
}
```

`wait()` and `notify()` (Conceptual Flow)



Rule: always `wait()` in a `while` loop
(spurious wakeups + condition may change)

From `wait/notify` to Higher-Level Tools

- Higher-level concurrency utilities reduce mistakes:
  - `BlockingQueue` (producer-consumer)
  - `CountDownLatch`, `Semaphore`
  - `ReentrantLock` + `Condition`
- Principle: prefer library abstractions over custom locking

ExecutorService and Thread Pools

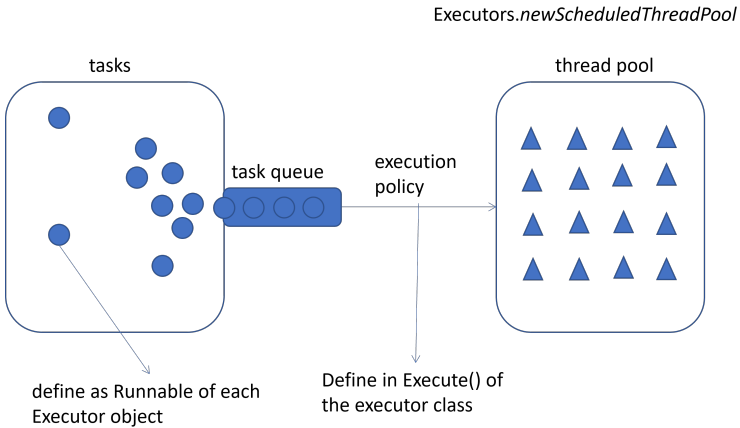
Why Not Create Threads Everywhere?

- Threads are not free: stack memory, context switching, scheduling overhead
- Unbounded thread creation can crash a service
- We want a **policy**: queueing, limits, naming, shutdown

Executor Framework: Core Idea

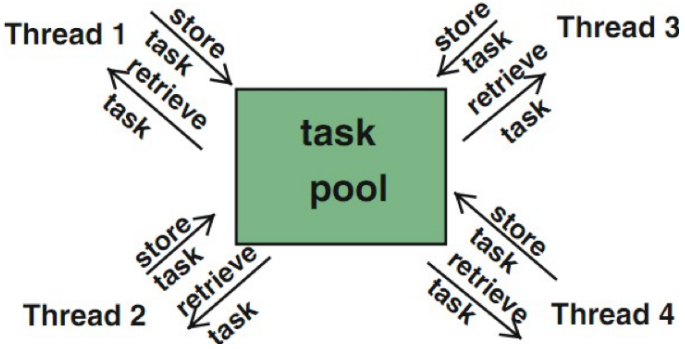
- Separate **task submission** from **task execution**
- Use a thread pool to reuse worker threads
- Main APIs:
  - `Executor` (`execute`)
  - `ExecutorService` (`submit`, `shutdown`)

Executor Framework (Diagram)

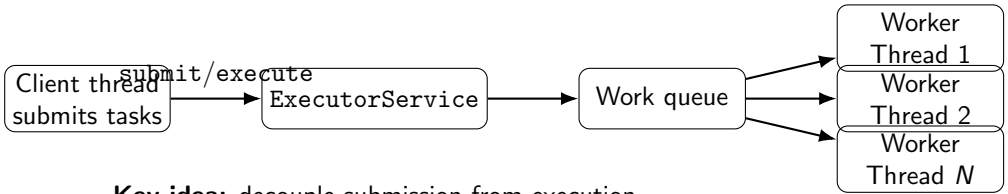


Task Pool Intuition (Diagram)

- A fixed number of workers repeatedly **pull tasks** from a queue
- This gives you a global place to enforce: limits, naming, shutdown, monitoring



Thread Pool in One Picture



Key idea: decouple submission from execution and reuse a bounded set of worker threads

Example: Parallel Work with `invokeAll`

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

public class ExecutorInvokeAll {
  public static void main(String[] args) throws Exception {
    ExecutorService pool = Executors.newFixedThreadPool(4);
    try {
      List<Callable<Integer>> tasks = new ArrayList<>();
      for (int i = 0; i < 8; i++) {
        int id = i;
        tasks.add(() -> id * id);
      }
      for (Future<Integer> f : pool.invokeAll(tasks)) {
        System.out.println(f.get());
      }
    } finally {
      pool.shutdown();
    }
  }
}
```

Shutdown Correctly

- `shutdown()` stops accepting new tasks
- `shutdownNow()` also attempts to interrupt workers
- Always make a plan for:
  - timeouts
  - cancellation
  - resource cleanup

Thread Pool Tuning (Rules of Thumb)

- CPU-bound tasks: pool size $\approx$ number of cores
- I/O-bound tasks: larger pools may help, but measure
- Prefer bounded queues in services (backpressure)

Warning

A large pool can increase tail latency (p95/p99) due to contention.

A Quick Note: CompletableFuture

- Useful for asynchronous pipelines and composition
- Avoid blocking inside callbacks
- Use custom executors for control

```
CompletableFuture.supplyAsync(this::fetch)
  .thenApply(this::parse)
  .thenAccept(this::store);
```

Patterns & Pitfalls

Thread-Safety by Design: The Easy Wins

- **Confinement**: keep mutable state inside one thread
- **Immutability**: share read-only objects
- **Minimize sharing**: reduce shared data surface area

Concurrent Collections (When Sharing is Needed)

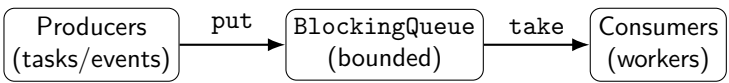
- `ConcurrentHashMap`: scalable map
- `CopyOnWriteArrayList`: read-mostly lists
- `BlockingQueue`: safe handoff between threads

Rule

Do not wrap a non-thread-safe collection and assume it is safe.

Producer–Consumer: Safe Handoff + Backpressure

- Producers put work items into a bounded queue
- Consumers take items and process them
- If producers are faster, the queue fills up ⇒ producers block (backpressure)



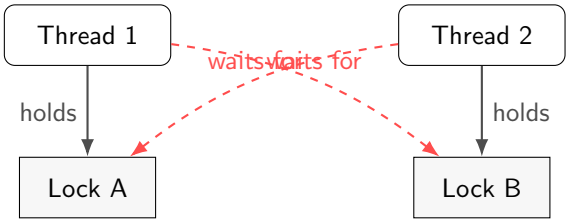
If the queue is full, `put` blocks
⇒ **backpressure**

Producer–Consumer with `BlockingQueue`

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

public class ProducerConsumerBlockingQueue {
  public static void main(String[] args) throws Exception {
    BlockingQueue<Integer> q = new ArrayBlockingQueue<>(2);

    Thread p = new Thread(() -> {
      try { q.put(1); q.put(2); q.put(-1); } // -1 = poison pill
      catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    });

    Thread c = new Thread(() -> {
      try {
        for (;;) {
          int x = q.take();
          if (x == -1) break;
          System.out.println("c " + x);
        }
      } catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    });

    p.start(); c.start(); p.join(); c.join();
  }
}
```

Deadlock: How It Happens

- Two (or more) threads wait forever for each other
- Common cause: inconsistent lock ordering



Cycle $\Rightarrow$ no progress (deadlock)

Prevention

Define a global lock order and always acquire locks in that order.

Common Pitfalls Checklist

- Holding a lock while doing I/O or calling unknown code
- Forgetting timeouts in blocking operations
- Using `Thread.sleep()` for coordination
- Ignoring interruption and cancellation
- Measuring only average latency (ignore tail)

Locks & Executor Patterns

Optional

How to Debug a "Hanging" Program (Checklist)

- Identify which threads are **RUNNABLE** vs **BLOCKED** vs **WAITING**
- Look for locks: `synchronized` monitors and `ReentrantLock`
- Symptoms to recognize:
  - deadlock: circular waiting on locks
  - starvation: one thread never gets CPU/lock
  - thread leak: threads keep growing over time

Optional

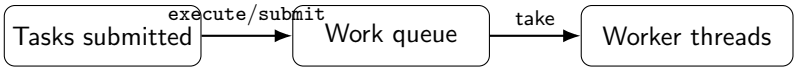
synchronized vs ReentrantLock (When to Use Which)

- synchronized: simplest; auto-unlock on exceptions
- ReentrantLock: try-lock, timed lock, multiple Condition objects

```
Lock lock = new ReentrantLock();
Condition notEmpty = lock.newCondition();
lock.lock();
try {
  while (q.isEmpty()) notEmpty.await();
  // ...
} finally {
  lock.unlock();
}
```

Optional

Thread Pool Tuning: What You Actually Control



Rule of thumb: bounded queue + clear rejection policy (protects latency and memory)

Knobs (`ThreadPoolExecutor`): `corePoolSize`, `maximumPoolSize`, `keepAliveTime`, `workQueue`, `RejectedExecutionHandler`

Optional

ExecutorService: Patterns Worth Remembering

- **Batch tasks**: `invokeAll` for "same kind" tasks
- **First result wins**: `invokeAny` (with timeouts)
- **Completion order**: `CompletionService` for streaming results

```
var cs = new ExecutorCompletionService<Result>(pool);
for (var t : tasks) cs.submit(t);
for (int i = 0; i < tasks.size(); i++) {
  Result r = cs.take().get(); // as they finish
}
```

Optional

CompletableFuture: Two Practical Rules

- Pass a custom executor if you care about isolation
- Handle exceptions explicitly (`exceptionally` / `handle`)

```
ExecutorService pool = Executors.newFixedThreadPool(4);
try {
  CompletableFuture.supplyAsync(this::fetch, pool)
    .thenApply(this::parse)
    .exceptionally(ex -> fallback());
} finally {
  pool.shutdown();
}
```

Optional

ConcurrentHashMap: Safe Patterns

- Prefer atomic methods: `computeIfAbsent`, `merge`
- Avoid "check-then-act" with external locks
- Values still need to be thread-safe (immutability helps)

Optional

Performance Notes (Only If Asked)

- Contended counters: `LongAdder` beats `AtomicLong` under high contention
- More threads can increase tail latency (p95/p99)
- Beware false sharing (adjacent hot fields on one cache line)

Synchronizers

Optional

Synchronizers: A Small Taxonomy

- **Mutual exclusion**: monitor lock, `ReentrantLock`
- **One-shot coordination**: `CountDownLatch`
- **Reusable coordination**: `CyclicBarrier`, `Phaser`
- **Permits / throttling**: `Semaphore`
- **Handoff / queues**: `BlockingQueue`, `SynchronousQueue`

Optional

CountDownLatch: One-shot Gate

- Use it to wait for **N events** to happen
- After it reaches zero, it stays open forever

```
var done = new CountDownLatch(tasks.size());
for (var t : tasks) {
  pool.submit(() -> { try { t.run(); } finally { done.countDown(); } });
}
done.await();
```

Optional

CyclicBarrier: Reusable Rendezvous

- Use it when **K threads** must reach the same point
- Optional "barrier action" runs when the last party arrives

```
var barrier = new CyclicBarrier(k, this::merge);
for (int i = 0; i < k; i++) {
  pool.submit(() -> { step1(); barrier.await(); step2(); return null; });
}
```

Optional

Semaphore: Limit Concurrency

- Use it to protect a scarce resource (DB connections, rate-limited API)
- Acquire before, release after

```
var sem = new Semaphore(10);
void handle() throws InterruptedException {
  sem.acquire();
  try { callRemote(); }
  finally { sem.release(); }
}
```

Optional

ReadWriteLock: Many Readers, Few Writers

- Multiple readers can proceed concurrently
- Writers are exclusive
- Works best when reads dominate and the protected data is large
- Simpler alternative: immutable snapshots + volatile reference

Condition Variables: Multiple Wait Sets

- A monitor has a single wait set; `Condition` allows multiple
- Always wait in a **loop** (spurious wakeups)

```
lock.lock();
try {
  while (q.isEmpty()) notEmpty.await();
  var x = q.remove();
  notFull.signal();
} finally {
  lock.unlock();
}
```

Optional

BlockingQueue as a Synchronizer

- Producer-consumer is usually easiest with `BlockingQueue`
- Blocking operations: `put` and `take`
- Bounded queues provide **backpressure**
- Prefer it over manual `wait/notify` in application code

Optional

SynchronousQueue: Direct Handoff

- Capacity is 0: a `put` waits for a `take`
- Useful for handoff designs and certain thread pool queues
- Bad fit for batching (no buffering)

Optional

Phaser / Exchanger (Mention Only)

- `Phaser`: dynamic parties + phased computation (advanced)
- `Exchanger`: two threads swap objects at a rendezvous
- If you do not already need them, prefer simpler tools

Optional

Which Synchronizer Should I Use?

- Need to protect shared state? → lock / synchronized
- Need to wait for N tasks? → CountDownLatch
- Need K threads to meet repeatedly? → CyclicBarrier
- Need to bound concurrency? → Semaphore
- Need safe handoff / buffering? → BlockingQueue

Concurrent Collections

Optional

Why Concurrent Collections Exist

- Correct synchronization is hard to get right everywhere
- Built-in concurrent data structures:
  - reduce locking mistakes
  - offer better scalability than a single global lock
- Still requires a mental model: atomic methods, iteration semantics

Optional

Synchronized Wrapper vs Concurrent Collection

- `Collections.synchronizedList(list)`: one lock around every method
- Concurrent collections often use finer-grained coordination
- Wrapper iteration still needs external synchronization
- Rule: prefer **concurrent collections** in multi-threaded code

Optional

ConcurrentHashMap: What You Can Assume

- Safe for concurrent access without external locks
- Iteration is **weakly consistent** (no `ConcurrentModificationException`)
- Avoid check-then-act on multiple operations
- Values should be immutable or thread-safe

Optional

ConcurrentHashMap: Atomic Update Patterns

- Prefer atomic map operations over manual locking

```
map.computeIfAbsent(k, key -> expensiveInit(key));
map.merge(k, 1L, Long::sum);
map.compute(k, (key, v) -> v == null ? 1 : v + 1);
```

Optional

CopyOnWriteArrayList: Read-mostly Workloads

- Writes copy the whole array (expensive)
- Reads are fast and iteration is snapshot-based
- Great when: many readers, few writers, small-ish list
- Bad when: frequent writes or very large lists

Optional

ConcurrentLinkedQueue vs BlockingQueue

- `ConcurrentLinkedQueue`: non-blocking, unbounded, no backpressure
- `BlockingQueue`: can block and can be bounded
- If you need producer-consumer, start with `BlockingQueue`

Optional

BlockingQueue Families (Quick Guide)

- `ArrayBlockingQueue`: bounded, array-based, predictable
- `LinkedBlockingQueue`: optionally bounded, linked nodes
- `PriorityBlockingQueue`: priority ordering (unbounded)
- `DelayQueue`: elements become available after a delay

Optional

ConcurrentSkipListMap/Set: Sorted and Concurrent

- Provides sorted keys with concurrent access
- Useful for range queries and ordered maps
- Higher overhead than hash-based maps

Optional

Counters Under Contention

- `AtomicLong`: one hot memory location (contention hotspot)
- `LongAdder`: striped counters, better throughput under contention
- Trade-off: `LongAdder.sum()` is not a single atomic snapshot

Optional

Common Mistakes

- Using a concurrent map but storing mutable, non-thread-safe values
- Treating weakly-consistent iteration as a strict snapshot
- Mixing external locks with concurrent container locks (risk of deadlock)
- Using unbounded queues where backpressure is required

Debugging & Tools

Optional

When Concurrency Breaks: What You Usually See

- Program **hangs**: no progress, CPU near 0%
- Program **spins**: CPU near 100% (busy-wait, livelock)
- **Wrong results**: lost updates, stale reads, out-of-order actions
- **Slowdown**: contention, oversized thread pools, queue buildup

Optional

Minimal Observability Checklist

- Add **thread names**: make logs readable ("pool-3-thread-7" is not enough)
- Log at the **boundary**: task submit/start/end + latency
- Always include: **request id / job id** to correlate events
- Prefer **counters** over println spam (rate limits, queue sizes, timeouts)

Optional

Thread Dumps: A Useful Mental Model

- A thread dump is a snapshot of **where each thread is blocked**
- Look for: **BLOCKED** (monitor lock), **WAITING** (condition/park), **RUNNABLE**
- The stack trace answers: *"What am I waiting for?"*
- Repeating patterns across dumps indicate a real bottleneck

Optional

Hanging Program: What to Inspect First

- **All threads WAITING**: likely missing `notify`/`signal`
- **Many threads BLOCKED**: lock contention or deadlock
- **One hot RUNNABLE thread**: busy loop (missing `sleep`/blocking call)
- **Thread count grows**: thread leak, unbounded executor creation

Optional

Deadlock: The Signature

- A deadlock is **circular waiting**: T1 holds A, waits for B; T2 holds B, waits for A
- A thread dump often shows **a cycle of locks**
- Common causes:
  - inconsistent lock ordering
  - calling into unknown code while holding a lock

Optional

Timeouts as a Design Tool

- Timeouts prevent infinite waits and turn hangs into errors
- Prefer time-bounded operations in production code

```
Future<Result> f = pool.submit(this::work);
try {
  Result r = f.get(200, TimeUnit.MILLISECONDS);
} catch (TimeoutException ex) {
  f.cancel(true); // interrupt if running
}
```

Optional

Interrupts: The Cancellation Mechanism

- `interrupt()` is a **request** to stop, not a force-kill
- Blocking calls (sleep/wait/join/queue ops) typically respond to interrupts
- Best practice: catch `InterruptedException` and **restore the flag**

Optional

Reproducibility: Reduce Non-determinism

- Shrink the problem: fewer threads, fewer inputs, shorter run time
- Control scheduling a bit: fixed thread pool size, fixed seeds
- Add **assertions** for invariants (counts, bounds, ordering where required)
- Prefer deterministic tests, but accept that some bugs are "Heisenbugs"

Optional

Typical Root Causes (Fast Map)

- Hang → missing notification, deadlock, forgotten `shutdown()`
- Spin → busy wait, broken condition loop, wrong volatile/atomic usage
- Wrong result → race condition, non-atomic compound action
- Slow → lock contention, too many threads, tiny tasks (overhead)

Optional

What Not to Do

- Do not "fix" by adding random `sleep` calls
- Do not add heavy logging inside hot locks (can change timing a lot)
- Do not use `Thread.stop()` or force-kill threads
- Do not block inside `synchronized` for long operations (I/O, network)

Optional

Tooling (Mention Only)

- Thread dumps: JVM can print stacks for all threads (platform dependent)
- Profiling: VisualVM, Java Flight Recorder (JFR), async-profiler
- Metrics: queue sizes, pool utilization, timeouts, error rates
- Rule: measure first, optimize second

Performance & Case Studies

Optional

Amdahl's Law (Reality Check)

- If fraction p is parallelizable, speedup is bounded by:

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}}$$

- Even with infinite cores: $S(\infty) = \frac{1}{1-p}$
- Optimize the sequential bottleneck first

Optional

Throughput vs Latency

- Throughput: tasks per second
- Latency: time per task (often care about p95/p99)
- Adding threads can increase throughput but **hurt tail latency**
- Choose based on product requirements

Optional

CPU-bound vs I/O-bound Thread Pools

- CPU-bound: start near **#cores** threads (avoid oversubscription)
- I/O-bound: can use more threads, but watch queueing and memory
- Rule: measure and tune; defaults are rarely perfect

Optional

Contention: Make the Critical Section Small

- Reduce shared mutable state
- Partition: sharding / lock striping
- Use atomic data structures (where they fit)
- Move slow work outside locks (I/O, allocations, callbacks)

Optional

False Sharing (Cache Lines)

- Two independent hot fields on the same cache line $\rightarrow$ performance drops
- Symptoms: scaling stops early, CPU cycles wasted on cache coherence
- Fixes: padding, separate objects, restructure arrays

Optional

Work Distribution: Avoid Tiny Tasks

- Too-small tasks $\rightarrow$ overhead dominates (scheduling, context switches)
- Chunk work: process ranges / batches
- For uneven tasks, use a queue (work stealing is a deeper topic)

Optional

Backpressure: Bounded Queues

- Unbounded queues hide overload until you run out of memory
- Bounded queues force the system to slow down early
- Choose a rejection/overflow strategy (drop, block, retry, fail fast)

Optional

Profiling vs Guessing

- The bottleneck is usually not where you expect
- Measure: CPU, allocations, lock contention, queue sizes
- Optimize one thing at a time; keep correctness tests

Optional

Microbenchmark Pitfalls (Mention Only)

- JVM warm-up and JIT compilation change performance over time
- Dead-code elimination can remove work you think you measured
- Use JMH for serious microbenchmarks

Optional

Case Study: Web Request Handling

- Task: serve many independent requests concurrently
- Use a bounded thread pool; avoid creating threads per request
- Protect shared caches with concurrent maps
- Add timeouts for upstream calls (avoid thread starvation)

Optional

Case Study: Batch Processing Pipeline

- Producer stage reads input; worker stage transforms; sink stage writes output
- Use `BlockingQueue` between stages (backpressure)
- Separate pools for I/O vs CPU stages (isolation)

Optional

Takeaways

- Correctness first: avoid shared state, use the right primitives
- Use executors and concurrent collections for safer defaults
- Measure performance; tune threads/queues based on workload

Wrap-up

Key Takeaways

1. Concurrency is about correctness first, then performance
2. Prefer **no sharing** $\rightarrow$ **immutability** $\rightarrow$ **synchronization**
3. Use **ExecutorService** for lifecycle and policy control
4. Learn the standard patterns (producer-consumer, cancellation)

Recommended References

- *Java Concurrency in Practice* (Goetz et al.)
- Official Java docs: docs.oracle.com/en/java/
- Concurrency utilities overview: java.util.concurrent

Questions?

- Shuhao Zhang(Huazhong University of Science and Technology (HUST))
- Email: shuhao_zhang[at]hust.edu.cn
- Homepage: shuhaozhangtony.github.io