Don't Make it a Race

The Four Common Concurrency Data Control Patterns

Hotels.com™

What You Will Learn

- The overall concurrency landscape
- The common ways to handle shared mutable state
  - With detailed examples
- How it all hangs together
- Great places to go on holiday

Agenda

Topic
Concurrency Low/Medium/High level
Shared Mutable State
Four Concurrency Control Patterns
Pessimistic Locking
Optimistic Transaction
Queue To A Single Thread
Partitioning The Data
Putting it all together

Picture of Bellagio from Vincent Radikahl travelling in Italy
https://www.instagram.com/p/BdVkne9F-4N/?taken-by=hotelsdotcom

Concurrency Low/Medium/High Levels

Concurrency patterns|models|frameworks|techniques?

There are 3 levels

- High level – **Models** (how you will write your programs)
  - Eg
    - Threads & Locks
    - Actors
  - Implemented with **frameworks**
- Medium level – **Patterns** (how you will decide to handle shared mutable state)
  - Discussed in detail in this talk
- Low Level – basic concurrency programming building blocks and **techniques** (how to build stuff from scratch)
  - Eg
    - Synchronized blocks
    - StampedLock

Low Level: Concurrency Building Blocks

So many

- Runnable | Thread | ThreadGroup | synchronized | Object.wait() | Object.notify() | Object.notifyAll() | NIO | java.util.concurrent.locks | volatile | java.util.concurrent.atomic | java.util.concurrent | some of java.lang.invoke (eg VarHandles) | @Contended
- Collections
  - java.util | Clojure collections (all available in Java) | PCollections | Chronicle | Agrona | Guava | Eclipse | Fastutil | Vavr | Apache | Trove | ObjectLayout/StructuredArray | Roaring Bitmaps | LMAX Disruptor | JCTools | high-scale-lib
- Doubtless many more

Medium Level: Concurrency Patterns

Covered in this talk

- Built from those low level building blocks
- These are used to define the core data management in high level model frameworks
- Not mutually exclusive, often different ones used in the same framework/program for different data as appropriate
- Four common concurrency data control patterns:
  - pessimistic locking
  - optimistic transactions
  - queue to a single thread
  - partitioning the data

High Level: Concurrency Models

10 Common Concurrency Models (including the 7 from "Seven Concurrency Models in Seven Weeks" *Paul Butcher*)

- DIY - Threads & Locks (eg Thread class, Runnable, synchronized, java.util.concurrent.locks, Executors)
- Functional Programming (eg Streams & Lambda expressions, RxJava)
- Atomic & Thread-local (eg java.util.concurrent.atomic, Clojure collections, VarHandle, SoftwareTransactionalMemory)
- Actors (eg Akka framework)
- Communicating Sequential Processes (eg ParallelUniverse's Quasar framework, Project Loom, Apache Camel)
- Data Parallelism (eg using GPUs, "Java on the GPU" Dmitry Aleksandrov https://www.youtube.com/watch?v=BjdYRtL6qjg)
- MapReduce (eg Apache Spark framework)
- Event-driven
  - Single-threaded (eg Vert.x framework)
  - Multi-threaded (eg CompleteableFuture framework, Kafka framework)
- Grid Computing (eg Apache Ignite, Hazelcast)

Picture from Vicky travelling in Bosnia and Herzegovina
https://www.instagram.com/p/BdGPZPWFDQr/?taken-by=hotelsdotcom

Shared Mutable State

Shared Mutable State

What does this mean?

- Shared: data storage that is used by **more than one thread**
- Mutable: data storage that is **updated** at some point with new data
- State: data storage with data

EACH word matters because eliminating ANY word stops concurrency being difficult

Shared Mutable **STATE**

Get rid of state and you have no concurrency problem

- No state is the jackpot of concurrency management
- Unfortunately not many applications have no state
  - You can push all the state to a datastore, which makes the application dependent on coordination to that datastore instead of coordination internally
    - But that might be ideal and simple and fast enough for your application, so do consider it
    - These are different high level concurrency models, eg CRUD, CQRS.
      - But essentially the same as synchronizing all access+updates to shared state
- But you can certainly decouple the stateful parts of the application from the stateless parts
  - So letting you scale each part appropriately
  - Stateless scaling is easy, you just add hardware resources and use copies of the stateless components to utilise those resources

Shared **MUTABLE** State

If the data item doesn't need changing, there is no concurrency problem

- **Immutable** data is great, you can use it across all threads with no problems
- You can also use data that was mutated but will no longer be mutated after some point (eg after you initialized your application or built the structure) – **effectively immutable**
  - You may need to flush changes to "main" memory and fault the changes to all existing threads, but after that the data is the same as immutable
  - The JIT might not be able to optimize as much as for actually immutable data, but that's usually a small matter compared to having to apply concurrency control to access the data
- Immutable state is not as easy to **get right** as you might expect
  - Fields should be **final and private**
  - And if a field references an object rather than primitive data, that object needs to be **immutable too**
- Immutable state objects are very efficient to use, but you often find you need to change the state, and to do that you have to make a new object – which is a source if **inefficiency** and adds pressure to the garbage collections.
  - Efficiency for concurrency usually far outweighs the inefficiency of needing copies

SHARED Mutable State

If the data item is not shared across threads, there is no concurrency problem

- It’s thread local state, and sequential programming that we’re all comfortable with applies

Shared Mutable State

Some of the high level concurrency models listed earlier are designed to avoid one of the words

- Shared **Im**mutable State: Functional Programming; MapReduce
- **Un**shared Mutable State: Actors; Single-threaded Event-driven; Data Parallelism; MapReduce

Picture in Baraza from Fred Fisher travelling in Zanzibar, Tanzania
https://www.instagram.com/p/BiPyFy3g27S/?taken-by=hotelsdotcom

Four Concurrency Control Patterns

Example

To show the differences, we'll use a consistent simple real world example

- BusinessProcess object with identity and state and some processing capability
  - Processing takes current state and new information and does something to produce the new state
- Objects accessed by identity (so will use hash maps as natural storage)
- New information can be anything, for my examples I'll use a double

Single Threaded

```
private Map<String,Business> data = new HashMap<>();

public void store(Business biz) {
  this.data.put(biz.identity(), biz);
}


public void processUpdate(String identity, double update) {
  Business biz = this.data.get(identity);
  biz.processValueInPlace(update);
}
```

Picture of Antelop Canyon from Povilas travelling in Arizona, USA
https://www.instagram.com/p/BhMrwOUAzSL/?taken-by=hotelsdotcom

Pessimistic Locking

Single Threaded

```
private Map<String,Business> data = new HashMap<>();

public void store(Business biz) {
  this.data.put(biz.identity(), biz);
}


public void processUpdate(String identity, double update) {
  Business biz = this.data.get(identity);
  biz.processValueInPlace(update);
}
```

Pessimistic Locking

```
private Map<String,Business> data = new HashMap<>();

public synchronized void store(Business biz) {
  this.data.put(biz.identity(), biz);
}


public synchronized void processUpdate(String identity, double update) {
  Business biz = this.data.get(identity);
  biz.processValueInPlace(update);
}
```

Pessimistic Locking

Benefits

- Simple to understand and reason about: single-threaded reasoning is what we do really well
- Easy to write single-threaded code
- JVM understands locks, especially monitors, and can optimize them well

Disadvantages

- Single-threaded execution - the higher the concurrency the more time threads are blocked
- Letting the datastructure escape the class (eg "return this.data“) is an easy bug to make
- Difficult to minimize lock time without adding bugs

Picture of Koh Hong from Stephan Audiger travelling in Krabi, Thailand
https://www.instagram.com/p/Be3i5X0FveZ/?taken-by=hotelsdotcom

Optimistic Transaction

Single Threaded

```
private Map<String,Business> data = new HashMap<>();


public void store(Business biz) {
  this.data.put(biz.identity(), biz);
}



public void processUpdate(String identity, double update) {
  Business biz = this.data.get(identity);
  biz.processValueInPlace(update);
}
```

Optimistic Transaction – Bug1

```
private ConcurrentMap<String,Business> data = new ConcurrentHashMap<>();


public void store(Business biz) {
  this.data.put(biz.identity(), biz);
}



public void processUpdate(String identity, double update) {
  Business biz = this.data.get(identity);
  biz.processValueInPlace(update);  //BUG!! but could fix in Business class
}
```

Optimistic Transaction – Bug2

```
private ConcurrentMap<String,Business> data = new ConcurrentHashMap<>();


public void store(Business biz) {
  this.data.put(biz.identity(), biz);
}



public void processUpdate(String identity, double update) {
  Business biz = this.data.get(identity);
  this.data.put(identity, biz.processValueReturnCopy(update));  //BUG!!
}
```

Optimistic Transaction

```
private ConcurrentMap<String,Business> data = new ConcurrentHashMap<>();

public void store(Business biz) {
  this.data.put(biz.identity(), biz);
}



public void processUpdate(String identity, double update) {
  this.data.computeIfPresent(identity, (k,v)->v.processValueIdempotent(update));
}
```

Example Business Processing

```
public void processValueInPlace(double update) {
  this.state += update; //BUG for unsynchronized concurrent update
}


public Business processValueIdempotent(double update) {
  return new Business(this.identity, this.state + update);
}


public void processValueInPlaceConcurrent(double update) {
  this.state.addAtomic(update);
}
```

Optimistic Transaction – Software Transactional Memory

```
class BusinessWrapper {AtomicReference<ImmutableBusiness> biz; ... }

ConcurrentMap<String,BusinessWrapper> data = new ConcurrentHashMap<>();


public void processUpdate(String identity, double update1, double u2) {
  ImmutableBusiness wrapper = this.data.get(identity);
  boolean done = false;
  while (!done) {
    ImmutableBusiness currentBiz = wrapper.biz.get();
    ImmutableBusiness newBiz = currentBiz.processValueReturningNew(update, u2));
    done = wrapper.biz.compareAndSet(currentBiz , newBiz);
  }
}
```

Example ImmutableBusiness Processing

```
// No synchronization nor Atomics needed, just simple processing
public ImmutableBusiness processValueReturningNew(double update, double u2) {
  double newState1 = this.state1 + update;
  double newState2 = this.state2 + u2;
  return new ImmutableBusiness(this.identity, newState1, newState2);
}
```

Optimistic Transaction

Benefits

- Eliminates explicit synchronization problems (lock contention, scope)
- Much higher chance of avoiding blocking, so higher throughput across all threads

Disadvantages

- Write collisions mean retries
- Retries can be costly, and they're unpredictable
- Difficult to understand and work through – easy to get wrong and introduce bugs
- Retry code must be idempotent – easy to get wrong and introduce bugs
- More transient objects in general transactional implementations, so more GC pressure

Picture of St. Paul's Cathedral from Jo Toader travelling in London, UK
https://www.instagram.com/p/BfB2RU-lN9j/?taken-by=hotelsdotcom

Queue To A Single Thread

Single Threaded

```
private Map<String,Business> data = new HashMap<>();

public void store(Business biz) {
  this.data.put(biz.identity(), biz);
}


public void processUpdate(String identity, double update) {
  Business biz = this.data.get(identity);
  biz.processValueInPlace(update);
}
```

Queue To A Single Consumer Thread

```
public class SingleThreadConsumer extends SingleThreaded implements Runnable {
//Just use the Single Threaded implementation in the consumer
//and add in queue processing capability
...

  public void run() {
    Business biz = takeFromQueue();
    switch(biz.operationType()) {
      case STORE: this.store(biz); break;
      case PROCESS: this.processUpdate(biz.identity(), biz.value()); break;
      case FLUSH: synchronized(this) {this.store(biz);}; break;
  }
}
```

Queue On A Single Thread

```
public void store(Business biz) {
  addToQueue(biz);
}



public void processUpdate(String identity, double update) {
  Business biz = new Business(identity, update, PROCESS);
  addToQueue(biz);
}
```

Queue On A Single Thread

Benefits

- No shared state
- Decouples producers and consumers; Simple fault tolerance
- Inherent asynchronous processing support
- Easy to convert to distributed implementation
- Lowest latency of updates if you can use “fire and forget”

Disadvantages

- A lot of infrastructure
- Additional latency from queueing if you need to confirm the update has completed/get the new state
- Unidirectional is straightforward, bidirectional communication is harder

Picture of Kotor Bay from Denise travelling in Montenegro
https://www.instagram.com/p/Bd3E2gKlvZS/?taken-by=hotelsdotcom

Partitioning The Data

Single Threaded

```
private Map<String,Business> data = new HashMap<>();


public void store(Business biz) {
  this.data.put(biz.identity(), biz);
}



public void processUpdate(String identity, double update) {
  Business biz = this.data.get(identity);
  biz.processValueInPlace(update);
}
```

Partitioned Data

```
private ConcurrentMap<String,Business> data = new ConcurrentHashMap<>();

public void store(Business biz) {
  this.data.put(biz.identity(), biz);
}

//The partitioning is in the Business class, not here ... here it’s simple
public void processUpdate(String identity, double update) {
  Business biz = this.data.get(identity);
  biz.processValueInPlacePartitioned(update);
}
```

Example Business Processing

```
//Example here just uses 3 partitions for simplicity
//and non-balanced partitioning, but obviously it can all be improved
//See Striped64 implementation for a very sophisticated implementation
public void processValueInPlacePartitioned(double update) {
  switch(Thread.currentThread().getId()%3) {
    case 0: this.state1.addAtomic(update); break;
    case 1: this.state2.addAtomic(update); break;
    case 2: this.state3.addAtomic(update); break;
  }
}
```

Partitioned Data

Benefits

- The fastest concurrent updates out there (eg LongAdder)
- Also used for the fastest distributed systems (eg Cassandra)

Disadvantages

- So many things to think about to get it right, easy to have bugs

Picture from Enrica Wong travelling in Saint Lucia
https://www.instagram.com/p/BnWoyhmAkvt/?taken-by=hotelsdotcom

Putting it all together

How to think about these four patterns

- Pessimistic Locking is easy, simple to understand and probably fast enough at reasonable concurrency
- Optimistic Transaction is most likely the way to go if you need to be able to handle higher concurrency, but you need to work hard to avoid subtle bugs
- Partitioned Data is the fastest highly concurrent option but really hard to achieve so a very long way to go to get your implementation mature (ie low on bugs)
- Sending to a Queue is the easiest for adding asynchronous support and also for adding distributed support

But …

How to think about these four patterns

… These four patterns are NOT mutually exclusive

- ConcurrentHashMap uses **Optimistic Transactions** to enable updates to be shared at the highest rates, **Partitioned Data** to increase multi-threaded throughput, and **Pessimistic Locking** where unlikely, infrequent or complex actions need to be processed
- The Actor Model uses **Sending to a Queue** together with **Partitioned Data** to enable individual Actors to operate serial execution, while allowing multiple different actors to execute concurrently
- Functional Programming in Java uses the underlying ForkJoinPool to execute the pipelines; ForkJoinPool is implemented using all four patterns together (Sending to a Queue, Partitioned Data, Optimistic Transactions and Pessimistic Locking), using each pattern where it's strength is most appropriately applied

1. Determine Shared Mutable Data Early

This is NOT premature optimization

- You should try to identify all potential shared mutable state for your application as early as possible
  - And guesstimate the level of concurrency for reads and for writes against that data
  - Including for the general data store and for specific data items
- There are fundamental design and architectural decisions you need to make based on this information
  - These are often very expensive to correct later

2. Can You Eliminate Any Word: Shared|Mutable|State?

As mentioned previously, eliminate one word, eliminate concurrency as a problem

- This is the ideal solution
- And if you can't eliminate it, can you at least use something else that is built to handle it?
  - That's why there are a zillion different Datastores, it's much easier to use one than build in your own management

3. Can A Concurrency Model Do It For You?

Many of the common ones are gaining greater use despite being really hard to debug, often difficult to understand, and less efficient than do-it-yourself, for exactly this reason.

The most common Concurrency Models that reduce and often eliminate your need to use one of the Concurrency Data Control Patterns are (in alphabetical order since I have no idea about our community uptake):

- Actors
- Functional Programming
- MapReduce
- Single-threaded Event-driven

4. Understand Concurrency As Best You Can

If the last couple of decision points means you still need to build, you need to get as best an understanding of concurrency subtleties as you can.

Even a decade of use doesn’t stop new concurrency bugs being found in the Fork-Join framework, despite it being open source and having some of the best concurrency experts in the world working on it

5. Encapsulate and Minimize Touchpoints

You decide you need to build it? Remember

- 1. The patterns are not mutually exclusive, choose according to need, try to keep it as simple as possible
- 2. Good OOP helps - encapsulate and present component APIs for client classes to use, encapsulation is spectacularly important for maintainable concurrency management
- 3. Avoid letting any data structures escape the class, that's a recipe for creating concurrency bugs
  - Eg don't have `getMap(){return mySharedMap;}`
- 4. Use immutable (final everything) classes returning new instances for any change to minimize bugs creeping in during maintenance
  - It's fewer things that can get messed up, even if it is higher overhead
  - And the JIT can optimize concurrent use of immutable instances usage pretty well
- 5. Use "persistent" (Clojure style) or other effectively copy-on-write classes to avoid collections getting corrupted
- 6. Use mature data concurrent structures (eg ConcurrentHashMap) even if you have to mangle your model a bit and take an efficiency hit
  - But make sure you understand how to use them

Picture from Alessia travelling in Naples, Italy
https://www.instagram.com/p/BkS_yvDlbxd/?taken-by=hotelsdotcom

Who Am I? Jack Shirazi

- Working in Performance and Reliability Engineering Team at Hotels.com
  - Part of Expedia Group, handling over $100billion in bookings annually
  - World's largest travel agency
- Founder of JavaPerformanceTuning.com
- Author of Java Performance Tuning (O'Reilly)
- Published over 60 articles on Java Performance Tuning & a monthly newsletter for 15 years & around 10 000 tuning tips

Hotels.com™