Common Java Concurrency pitfalls

Presented by : Kunal Sinha

Austin Java Users Group

03/26/2019

About me

Seasoned software engineer primarily working in the area of Identity and access management. Spent nearly 7 years at Sun Microsystems, where I contributed to the development of Sun LDAP directory and OpenDS LDAP directory servers. At Oracle, I was responsible for architecting and developing Oracle Unified Directory (OUD) LDAP directory. At present, I am responsible for designing and developing interesting features of Okta Universal Directory.

Overview

- Need for concurrency
- Common pitfalls and how to avoid them
- Q&A

Concurrency

- Concurrency comes with a cost
  - Cost of context-switching
  - Wrong use of CPU time
  - Synchronization prohibits compiler optimization
  - Flushing of memory caches
- Extreme caution is required (user errors)
- The bugs may almost never show up in production

Common Pitfalls

```
public final class Example1 {

  public static void main(String[] args) {
    final MyTask mytask = new MyTask();
    Thread t = new Thread(mytask);
    t.start();
    mytask.stop();
  }

  private static final class MyTask implements Runnable {
    private boolean shutdown;

    MyTask() {
      this.shutdown = false;
    }

    public void stop() {
      shutdown = true;
    }

    public void run() {
      while(!shutdown) {
        //Some network I/O
        //sleep
      }
    }
  }
}
```

Learning: understand visibility guarantee

- JVM specification allows optimization by reordering instructions, local caching with thread-level visibility
- Dependent actions should ensure a happens-before guarantee
- Piggybacking is possible but should be reserved for experts
- Use Atomic variables or volatile
- Use synchronized/locks
- Use immutable classes

```
public class Example2 {

  public static void main(String ...args) {
   final Shared shared = new Shared();
   Thread t1 = new Thread(() -> System.out.println(shared.getData()));
   //Guarantees happens-before here.
   t.start();

   //main thread is setting the data
   shared.setData(5);
   //t1 is getting the data at this point.
 }


 private static final class Shared {
  private int data;

  public int getData() {
    return data;
  }
  public synchronized void setData(int data) {
    this.data = data;
  }
}
}
```

Learning: happens-before guarantee isn’t obvious all time

```
public class CompoundActionExample{

  private final ConcurrentHashMap<String, String> map = new
ConcurrentHashMap<>();

  private final AtomicInteger counter = new AtomicInteger(0);

  public void method(String key, String value) {
    map.putIfAbsent(key, value);
    counter.incrementAndGet();
  }
```

Learning

- Atomicity and visibility are two different things
- Some optimizations are lost when using compound actions but it is for safety.

```
public class CompoundActionExample{
  private Map<String, String> map = new HashMap<>();
  private volatile int counter;
  private final ReentrantReadWriteLock rwLock = new
ReentrantReadWriteLock();

  public void method(String key, String value) {
    rwLock.writeLock().lock();
    try {
    if (!map.containsKey(key)) {
      map.put(key, value);
    }
    counter++;
    } finally {
     //Use finally to avoid any exceptions
      rwLock.writeLock().unlock();
    }
  }

  public int getCounter() {
    return counter;
  }

  public String getValue(String key) {
    rwLock.readLock().lock();
    try {
      return map.get(key);
    } finally {
      rwLock.readLock().unlock();
    }
  }
}
```

```
public class ImmutabilityExample {
  private int data1;
  private int data2;
  private boolean isDone;

  public ImmutabilityExample(int data1, int data2,
boolean isDone) {
    this.data1 = data1;
    this.data2 = data2;
    this.isDone = isDone;
  }

  public void setDone(boolean isDone) {
    this.isDone = isDone;
  }
```

Learning: Write immutable classes whenever you can

```
public class ImmutabilityExample {
  private final int data1;
  private final int data2;
  private volatile boolean isDone;

  public ImmutabilityExample(int data1, int
data2, boolean isDone) {
    this.data1 = data1;
    this.data2 = data2;
    this.isDone = isDone;
  }

  public void setDone(boolean isDone) {
    this.isDone = isDone;
  }
}
```

```
public class LockExample {
  private final ConcurrentHashMap<String, String> hashMap = new ConcurrentHashMap<>();
  private final List<Integer> list = new Vector<>();

  public void method1(String key, String value) {
    synchronized(hashMap) {
      //Get the value
      //Get the mapping count.
      //Some other stuff on the hashmap.
    }
  }

  public synchronized void putIfAbsent(int value) {
     if (!list.contains(value)) {
        list.add(value);
     }
  }
}
```

Learning: Use an appropriate lock

```
public class CMEExample {
 private final Collection<Integer> collection =
Collections.synchronizedList(new ArrayList<>());

 public void method() {
   Iterator<Integer> it = collection.iterator();
   while(it.hasNext()) {
     //iterate.
   }
 }
}
```

Learning: Design to avoid fail-fast

- Hold a lock till the time iterator is iterating
- Clone it to avoid the error
- Use a collection from util.concurrent package that use weakly-consistent iterators

Interruptions

```
public class InterruptionExample implements Callable<Integer> {

 public Integer call() {
   try {
     //some blocking method.
   } catch (Exception e) {
     LOG.warn("Exception occurred", e);
   }

   return //some value.
 }
}
```

Lesson: Handle Interruptions better

- Respect InterruptedException in method signatures
- Propagate it back to the caller
- Restore the interruption status
- If you deal with AWS SDK, handle AbortException

```
public class InterruptionExample implements
Callable<Integer> {
 public Integer call() {
   try {
     //some blocking method.
   } catch (InterruptedException e) {
     //Restore the interruption status.
     Thread.currentThread().interrupt();
   }

   return //some value.
 }
```

Using ExecutorService

```
public class ExecutorServiceExample {
  public static void main(String ...args) {
    ExecutorService executorService =
Executors.newFixedThreadPool(1);
    executorService.submit(()-> {
      while (true) {
        //Simulate long-running job
        Thread.sleep(1000);
      }
    });

    //JVM shutting down so stop the
service.
    executorService.shutdown();
  }
}
```

Lesson: do a proper shutdown by using interruptions

```
public class ExecutorServiceExample {
  public static void main(String ...args) {
    ExecutorService executorService = Executors.newFixedThreadPool(1);
    executorService.submit(()-> {
      while (true) {
        //Simulate long-running job
        Thread.sleep(1000);
      }
    });

    executorService.shutdown();
    try {
        if (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {
            executorService.shutdownNow();
        }
    } catch (InterruptedException e) {
        executorService.shutdownNow();
    }
  }
}
```

```
public class LazyInitExample {
  private static LazyInitExample instance;

  public static LazyInitExample getInstance0() {
    if (instance == null) {
      instance = new LazyInitExample();
    }
    return instance;
  }

  public static LazyInitExample getInstance1() {
    if (instance == null) {
      synchronized(LazyInitExample.class) {
        if (instance == null) {
          instance = new LazyInitExample();
        }
      }
    }
    return instance;
  }
}
```

Learning: Still a long way to go ☺

Q & A