Чёрная метка

StampedLock и его друзья

Дмитрий Чуйко

Java SE Performance Team

24 апреля 2016

Содержание

1. Введение

2. Анархия на борту

3. Честный, жадный, хитрый

4. Грядущий сиквел

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracles products remains at the sole discretion of Oracle.

Введение

Напоминание: 2016 год

- JDK 9 Early Access
  https://jdk9.java.net/
- JDK 8u
- JDK 7 End of Public Updates in April 2015

Измерение: Performance

Приводимые результаты

- Intel® Xeon® E5-2680 (2x8x2, 2.7 GHz)
- Linux x64, kernel 2.6
- JDK 8
- OpenJDK JMH

Анархия на борту

Не пытайтесь покинуть порт: Отплыть из Омска

```
class Point {
  private double x, y;
  ...
  void move(double deltaX, double deltaY) {
    x += deltaX;
    y += deltaY;
  }
  double distanceFromOrigin() {
    return Math.hypot(x, y);
  }
  void moveIfAtOrigin(double newX, double newY) {
    if(x == 0.0 && y == 0.0) {
      x = newX;
      y = newY;
    }
  }
 }
```

Не пытайтесь покинуть порт: 2 капитана

Честный, жадный, хитрый

Locks: Всех под замок!

```
class Point {
  private double x, y;
  ...
  synchronized void move(double deltaX, double deltaY) {
    x += deltaX;
    y += deltaY;
  }
  synchronized double distanceFromOrigin() {
    return Math.hypot(x, y);
  }
  synchronized void moveIfAtOrigin(double newX, double newY)
    if(x == 0.0 && y == 0.0) {
      x = newX;
      y = newY;
    }
  }
 }
```

Locks: Чего бы хотелось

- Блокировка на чтение, на запись
- Чтение без блокировки
- Upgrade/downgrade (RL↔WL)
- Простота использования
- И чтобы быстро работало

Locks: ReentrantReadWriteLock

@since 1.5

- Lock, блокировка на чтение, на запись
- Рекурсивный захват (reentrancy)
- Пробная блокировка, таймауты, прерываемость

Locks: ReentrantReadWriteLock

@since 1.5

- Lock, блокировка на чтение, на запись
- Рекурсивный захват (reentrancy)
- Пробная блокировка, таймауты, прерываемость
- Conditions
- Fair/unfair
- Serializable...
- Memory effects

ReentrantReadWriteLock: Point

```
class Point {
   private double x, y;
   private final ReentrantReadWriteLock rrwl = new ReentrantReadWriteLock();
   ...
```

ReentrantReadWriteLock: Запись

```
void move(double deltaX, double deltaY) { // exclusive
  rrwl.writeLock().lock();
  try {
    x += deltaX;
    y += deltaY;
  } finally {
    rrwl.writeLock().unlock();
  }
}
```

ReentrantReadWriteLock: Чтение

```
double distanceFromOrigin() {
  rrwl.readLock().lock();
  try {
    return Math.hypot(x, y);
  } finally {
    rrwl.readLock().unlock();
  }
}
```

Locks: ReentrantReadWriteLock

- `readLock()` на чтение, `writeLock()` на запись
- Попробуйте написать оптимистичное чтение, гарантировать memory effects
- Downgrade WL→RL: захват RL под WL
- Накладные расходы на `ThreadLocal`

Locks: ReentrantReadWriteLock

"Most people know that I consider acceding to requests to support per-thread read counts in JDK6 RRWL to be the worst decision I've ever made in j.u.c. But still not as bad as someone's decision not to make Thread.getId() a final method."
Doug Lea

Locks: ReentrantReadWriteLock

Что внутри

- Полная инкапсуляция
- `ThreadLocal<HoldCounter>`
- `AbstractQueuedSynchronizer`
  - Атомарное состояние: `volatile int state`
  - Очередь ожидания с атомарными обновлениями (CLH - Craig, Landin, Hagersten)

Locks: AbstractQueuedSynchronizer

```
private transient volatile Node head;
private transient volatile Node tail;
```

Locks: StampedLock

@since 1.8

- Пример из javadoc

StampedLock: Point

```
class Point {
   private double x, y;
   private final StampedLock sl = new StampedLock();
   ...
```

StampedLock: Запись

```
void move(double deltaX, double deltaY) { // exclusive
  long stamp = sl.writeLock();
  try {
    x += deltaX;
    y += deltaY;
  } finally {
    sl.unlockWrite(stamp);
  }
}
```

StampedLock: Чтение

```
double distanceFromOrigin() {
  long stamp = sl.readLock();
  try {
    return Math.hypot(x, y);
  } finally {
    sl.unlockRead(stamp);
  }
}
```

StampedLock: Оптимистичное чтение

```
double distanceFromOrigin() { // A read-only method
  long stamp = sl.tryOptimisticRead();
  double currentX = x;
  double currentY = y;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentX = x;
      currentY = y;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return Math.hypot(currentX, currentY);
}
```

StampedLock: Апгрейд

```
void moveIfAtOrigin(double newX, double newY) {
  long stamp = sl.writeLock();
  try {
    if(x == 0.0 && y == 0.0) {
      x = newX;
      y = newY;
    }
  } finally {
    sl.unlockWrite(stamp);
  }
}
```

StampedLock: Апгрейд

```
void moveIfAtOrigin(double newX, double newY) {
  long stamp = sl.readLock();
  try {
    if(x == 0.0 && y == 0.0) {
      sl.unlockRead(stamp);
      // competitors come
      stamp = sl.writeLock();
      if(x == 0.0 && y == 0.0) {
        x = newX;
        y = newY;
      }
    }
  } finally {
    sl.unlock(stamp);
  }
}
```

StampedLock: Апгрейд

```
void moveIfAtOrigin(double newX, double newY) { // upgrade
  long stamp = sl.readLock(); // Could start with optimistic
  try {
    while (x == 0.0 && y == 0.0) {
      long ws = sl.tryConvertToWriteLock(stamp);
      if (ws != 0L) {
        stamp = ws;
        x = newX;
        y = newY;
        break;
      }
      else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally {
    sl.unlock(stamp);
  }
}
```

StampedLock: Хитрый

- Оптимистичное чтение
- Если проверки неудачные, можно захватить блокировку
  - Обычно удачные

StampedLock: Хитрый

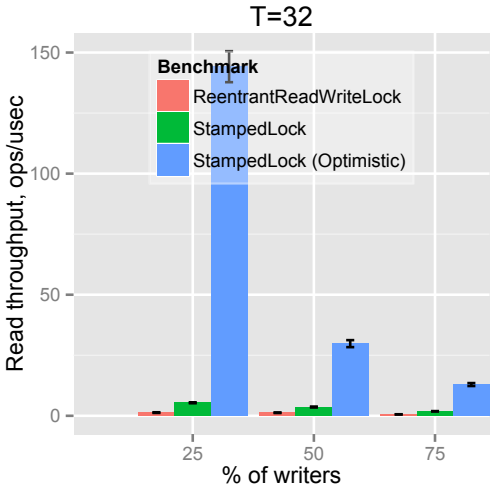
- Оптимистичное чтение
- Если проверки неудачные, можно захватить блокировку
  - Обычно удачные
- Выигрыш
  - Мало записей $\Rightarrow$ в разы
  - Нет записей $\Rightarrow$ на порядки
  - Только проверенное состояние является консистентным

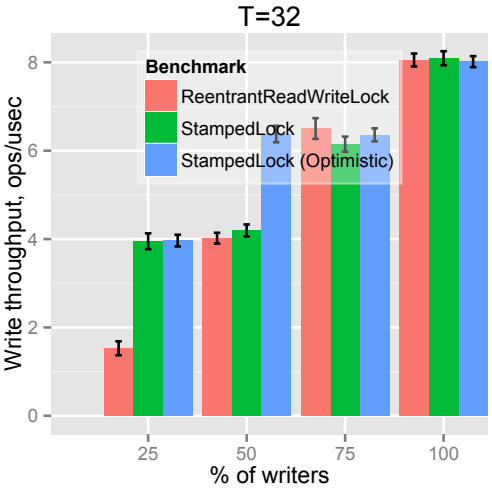
StampedLock: Оптимистичное чтение

Тест производительности

- Читатели пытаются читать без блокировки и проверять
- Читатели читают под блокировкой на чтение
- Писатели пишут под блокировкой на запись

StampedLock: Оптимистичное чтение





StampedLock: Оптимистичное чтение

- 0 писателей, 32 читателя
  - StampedLock (Optimistic) → ≈**1500x**
  - StampedLock → ≈2x
- 1 писатель, 31 читатель
  - StampedLock (Optimistic) → ≈200x
  - StampedLock → ≈3x

StampedLock: Другие свойства

- Повышение уровня блокировки
  - Может не быть выигрыша, если не ограничивать запись
  - Корректное состояние

StampedLock: Другие свойства

- Повышение уровня блокировки
  - Может не быть выигрыша, если не ограничивать запись
  - Корректное состояние
- В штатном варианте может быть в разы быстрее RRWL
  - При этом потребляет меньше ресурсов
  - Можно передавать метку

StampedLock: Другие свойства

- Повышение уровня блокировки
  - Может не быть выигрыша, если не ограничивать запись
  - Корректное состояние
- В штатном варианте может быть в разы быстрее RRWL
  - При этом потребляет меньше ресурсов
  - Можно передавать метку
- Если очень хочется
  - `asReadWriteLock()`

StampedLock: Что внутри

- Атомарное состояние: `volatile long state`
- Очередь ожидания с атомарным обновлением (CLH)
  - co-waiters
- Состояние – метка для использования снаружи

StampedLock: Как работает

```
/* StampedLock sl: volatile long state */
```

Sparrow

```
stamp = sl.tryOptimisticRead(); /* A: s = state */
double currentX = x;            /* B: x */
double currentY = y;            /* B: y */
if (!sl.validate(stamp)) {      /* C: see: state,x,y */
```

Barbossa

```
stamp = sl.writeLock();         /* 0: publish: state */
x = newX;                       /* 1: x */
y = newY;                       /* 1: y */
sl.unlock(stamp);               /* 2: publish: x,y,state */
```

StampedLock: Наблюдаемое поведение

Reordering

validate	B: read C: validate	B: validate C: read	Side Effect
∅	Y	**Y**	N
`volatile read`	Y	**Y**	N
`volatile write`	Y	N	**write**
`CAS`	Y	N	**write**
`U.getLongVolatile()`	Y	**Y(doc)**/N(hs)	N
`U.loadFence()`	Y	N	order

StampedLock: StampedLock

Как работает (@since 1.8)

```
/* StampedLock sl: volatile long state */
```

Sparrow

```
stamp = sl.tryOptimisticRead(); /* s = state */
double currentX = x; double currentY = y;
if (!sl.validate(stamp)) {      /* unsafe.loadFence() */
```

Barbossa

```
stamp = sl.writeLock();         /* CAS(state, next1) */
x = newX; y = newY;
sl.unlock(stamp);               /* CAS(state, next2) */
```

Грядущий сиквел

StampedLock: Unsafe

```
private static final sun.misc.Unsafe U =
        sun.misc.Unsafe.getUnsafe();

// JDK 9
private static final jdk.internal.misc.Unsafe U =
        jdk.internal.misc.Unsafe.getUnsafe();
```

StampedLock: Unsafe

```
// JDK 8
private static final sun.misc.Unsafe U =
        sun.misc.Unsafe.getUnsafe();

// JDK 9
private static final jdk.internal.misc.Unsafe U =
        jdk.internal.misc.Unsafe.getUnsafe();
```

StampedLock: Unsafe

```
// JDK 8
STATE = U.objectFieldOffset(StampedLock.class.getDeclaredField("state")); // long
U.compareAndSwapLong(this, STATE, s, next = s + WBIT)

// Faster Atomic*FieldUpdaters
STATE = AtomicLongFieldUpdater.newUpdater(StampedLock.class, "state");
STATE.compareAndSet(this, STATE, s, next = s + WBIT)

// VarHandles
STATE = MethodHandles.lookup().findVarHandle( // VarHandle
        StampedLock.class, "state", long.class);
STATE.compareAndExchangeVolatile( // or even compareAndExchangeAcquire
        this, s, next = s + WBIT);
```

Concurrency: Ресурсы

- *Java Concurrency in Practice*
  Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea
- Concurrency JSR-166 Interest Site
  http://g.oswego.edu/dl/concurrency-interest/
- JMH
  http://openjdk.java.net/projects/code-tools/jmh/

Спасибо: Q & A