Java Concurrency
на практике

Chapter-13:

Explicit Locks
Явные замки (блокировки)

Upcode Software
Engineer Team

-2023-

CONTENT

1. Что такое **Lock и ReentrantLock ?**
2. **Опрашиваемое и хронометрируемое приобретение блокировки**
3. **Прерываемое приобретение блокировки**
4. **Не-блочно структурированная замковая защита**
5. **Соображения по поводу производительности**
6. **Справедливость**
7. **Выбор между synchronized и ReentrantLock**
8. **Замки чтения-записи**
9. **Доп. блокировки**
10. **Ресурсы**

Что такое **Lock** и **ReentrantLock ?** (1/1)

Lock - это интерфейс в Java, который предоставляет базовые методы для работы с блокировкой.

Он включает методы **lock()** для получения блокировки и **unlock()** для освобождения блокировки.

Lock предоставляет более гибкий и мощный механизм управления блокировкой, чем традиционные синхронизированные блоки.

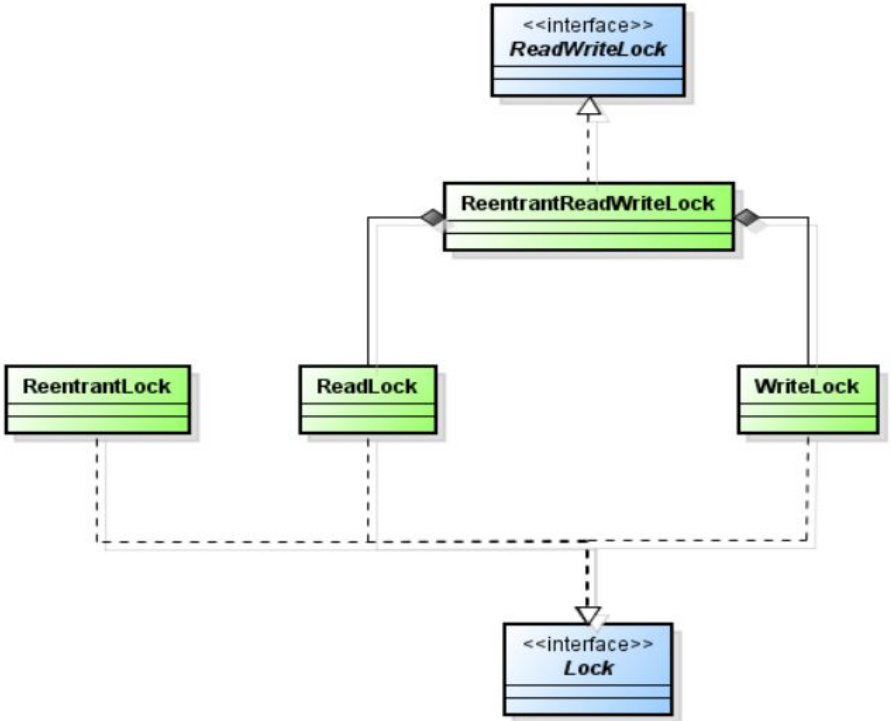
Что такое **Lock** и **ReentrantLock ?** (2/1)

ReentrantLock - это конкретная реализация интерфейса **Lock**.

Он расширяет функциональность обычной блокировки, предоставляя "**реентерабельность**" блокировки, что означает, что один и тот же поток может захватить блокировку несколько раз.

Это может быть полезно в некоторых сценариях, когда потоку требуется занять блокировку несколько раз перед тем, как ее освободить.

Что такое **Lock** (3/1)



Что такое **Lock** и **ReentrantLock ?** (3/1)

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Printer {
    private Lock lock = new ReentrantLock();

    public void printDocument(String document, int copies) {
        lock.lock(); // Захватываем блокировку

        try {
            for (int i = 0; i < copies; i++) {
                // Критическая секция: печать документа
                System.out.println("Печать документа: " + document);
            }
        } finally {
            lock.unlock(); // Освобождаем блокировку
        }
    }

    public void photocopyDocument(String document, int copies) {
        lock.lock(); // Захватываем блокировку

        try {
            for (int i = 0; i < copies; i++) {
                // Критическая секция: копирование документа
                System.out.println("Копирование документа: " + document);
            }
        } finally {
            lock.unlock(); // Освобождаем блокировку
        }
    }
```

Что такое **Lock** и **ReentrantLock ?** (4/1)

```
32
33      public static void main(String[] args) {
34          Printer printer = new Printer();
35
36          // Создаем два потока для работы с принтером
37          Thread thread1 = new Thread(() -> printer.printDocument("Договор", 3));
38          Thread thread2 = new Thread(() -> printer.photocopyDocument("Заявление", 2));
39
40          // Запускаем потоки
41          thread1.start();
42          thread2.start();
43      }
44  }
45
```

Что такое **Lock** и **ReentrantLock ?** (5/1)

```
Печать документа: Договор
Печать документа: Договор
Печать документа: Договор
Копирование документа: Заявление
Копирование документа: Заявление
```

Опрашиваемое и хронометрируемое приобретение блокировки (1/2)

Опрашиваемое приобретение блокировки означает, что **поток, пытающийся получить блокировку, будет продолжать пытаться ее получить**, пока она не будет доступна. Это отличается от неблокирующих алгоритмов, которые позволяют потоку продолжать работу, даже если требуемый ресурс недоступен.

Хронометрируемое приобретение блокировки означает, что **поток, пытающийся получить блокировку, будет ожидать определенный период времени, прежде чем считать, что блокировка недоступна**. Если блокировка не будет получена в течение этого периода, поток может принять решение о продолжении работы без блокировки или повторной попытке получения блокировки.

Прерываемое приобретение блокировки (1/2)

Прерываемое приобретение блокировки **(interruptible lock acquisition)** относится к возможности прерывания потока в процессе ожидания блокировки. Это связано с использованием **механизма прерывания потоков в Java.**

В **Java** блокировки можно использовать для синхронизации доступа к общим ресурсам. Обычно, **когда поток пытается получить блокировку**, **и она уже занята другим потоком, он переходит в состояние ожидания** до тех пор, пока блокировка не освободится.

Прерываемое приобретение блокировки дает возможность прерывания этого ожидания, если другой поток вызывает метод `interrupt()` для ожидающего потока. Это позволяет эффективно управлять ситуациями, когда длительное ожидание блокировки не является желательным или когда поток должен быть прерван извне.

Не-блочно структурированная замковая защита (1/4)

Неблокирующая замковая защита - это метод синхронизации доступа к общим ресурсам в многопоточной среде, который не использует традиционные блокировки. Вместо этого, он использует атомарные операции и другие механизмы, которые позволяют потокам работать параллельно без блокировки друг друга.

Этот подход обеспечивает более высокую производительность и масштабируемость, поскольку избегает проблем, связанных с блокировками, такими как взаимная блокировка и ожидание. Неблокирующие алгоритмы обычно используются в приложениях, где производительность и эффективное использование ресурсов являются критически важными.

Соображения по производительности (1/5)

- **Потокобезопасность:**

Убедитесь, что ваш код обеспечивает безопасность потоков. Используйте синхронизацию, встроенные механизмы блокировки (lock) и другие средства, чтобы предотвратить состояние гонки и обеспечить корректность работы в многопоточной среде.

- **Использование правильных структур данных:**

Выбирайте структуры данных, которые предназначены для многопоточной работы. Например, **ConcurrentHashMap в Java** обеспечивает безопасное чтение и запись без блокировки всей карты данных.

Соображения по производительности (2/5)

- **Избегайте блокировок во время выполнения:**

Стремитесь к минимизации блокировок и использованию атомарных операций. Это может уменьшить вероятность взаимоблокировок и улучшить производительность.

- **Понимание Thread Pools:**

Используйте пулы потоков (Thread Pools) для эффективного управления потоками. Это может уменьшить накладные расходы на создание и уничтожение потоков.

Соображения по производительности (3/5)

- **Оптимизация кода:**

Профилируйте ваш код, выявляйте узкие места и оптимизируйте их. Однако, помните о золотом правиле оптимизации: "Не оптимизируйте заранее" (Don't optimize prematurely).

- **Использование Volatile и атомарных типов:**

Если вам нужна переменная, доступ к которой должен быть атомарным, рассмотрите использование ключевого слова volatile или атомарных типов данных.

Соображения по производительности (4/5)

- **Мониторинг и анализ:**

Используйте инструменты мониторинга и профилирования, такие как VisualVM, jConsole, или различные агенты профилирования, чтобы отслеживать производительность и искать проблемы.

Справедливость (1/6)

Справедливость (**fairness**) относиться к использованию справедливых блокировок (**fair locks**) или справедливых очередей (**fair queues**).

Справедливость в данном случае означает, что потоки будут получать доступ к ресурсам в порядке их запроса или в соответствии с другим справедливым механизмом.

Справедливость (2/6)

Например, если используется **ReentrantLock** с параметром **fair** в значении **true**, это означает, что блокировки будут предоставляться потокам в том порядке, в котором они запрашивают доступ.

```
ReentrantLock fairLock = new
ReentrantLock(true);
```

Это может быть полезным в сценариях, где важна справедливая очередь доступа к ресурсам, чтобы избежать "**голодающих**" потоков.

Synchronized или ReentrantLock (1/7)

1. **Простота использования:**

synchronized является более простым и удобным в использовании. Инструкции synchronized применяются автоматически, и не требуется явного использования объектов блокировки.

```
public synchronized void
synchronizedMethod() {
    // код, защищенный блокировкой
}
```

Synchronized или **ReentrantLock** (2/7)

В случае **ReentrantLock**, вы должны явно создать объект блокировки, вызывать **lock()** и **unlock()**:

```
private final ReentrantLock lock = new ReentrantLock();

public void reentrantLockMethod() {
    lock.lock();
    try {
        // код, защищенный блокировкой
    } finally {
        lock.unlock();
    }
}
```

Synchronized или **ReentrantLock** (3/7)

1. **Гибкость**: ReentrantLock предоставляет большую гибкость. Например, у вас есть больше возможностей для настройки поведения блокировки, таких как использование справедливой блокировки (**fair**), установка времени ожидания (**tryLock(long timeout, TimeUnit unit))** и т. д.

2. **Причина выбора**: В большинстве случаев использование synchronized будет предпочтительным из-за своей простоты и легкости в использовании. Однако в случаях, когда вам нужна дополнительная гибкость и возможность управления, ReentrantLock может быть более подходящим выбором.

Замки чтения-записи (1/8)

Замки чтения и записи - это механизмы синхронизации, которые позволяют разделять доступ к данным между потоками. Они предоставляют более высокую производительность, чем простые блокировки чтения-записи, потому что они разрешают параллельное чтение, но блокируют запись в тот момент, когда какой-то поток пишет.

В Java для этого может использоваться **интерфейс ReadWriteLock** с двумя реализациями:

ReentrantReadWriteLock и StampedLock.

Замки чтения-записи (2/8)

Пример использования **ReentrantReadWriteLock**:

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class SharedResource {
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private int data;

    public int readData() {
        lock.readLock().lock();
        try {
            // Чтение данных
            return data;
        } finally {
            lock.readLock().unlock();
        }
    }
```

```
    public void writeData(int newData) {
        lock.writeLock().lock();
        try {
            // Запись данных
            data = newData;
        } finally {
            lock.writeLock().unlock();
        }
    }
}
```

Замки чтения-записи (3/8)

В этом примере метод **readData()** использует блокировку чтения, что позволяет множеству потоков параллельно читать данные. Метод **writeData()** использует блокировку записи, блокируя доступ ко всему ресурсу во время записи.

Важно заметить, что использование замков чтения и записи имеет смысл только в тех ситуациях, где частые операции чтения превалируют над операциями записи. В противном случае, использование обычных блокировок может быть более эффективным.

Synchronized (1/9)

В Java существует несколько различных механизмов блокировки, каждый со своими особенностями, плюсами и минусами. Давай рассмотрим некоторые из них:

synchronized:

Плюсы:

- Прост в использовании: не требует явного создания объекта Lock.
- Автоматическое освобождение блокировки при выходе из блока кода.

Минусы:

- Не является реентерабельным в том смысле, что поток не может повторно захватить блокировку, которую уже удерживает.
- Могут возникнуть проблемы с производительностью в некоторых случаях.

ReentrantLock (2/9)

ReentrantLock:

Плюсы:

- Реентерабельность: поток может повторно захватывать блокировку.
- Больше гибкости и дополнительные функции, такие как возможность установки таймаута для ожидания блокировки.

Минусы:

- Больше кода для написания и поддержки.

ReadWriteLock (3/9)

ReadWriteLock (ReentrantReadWriteLock):

Плюсы:

- Разделяемая блокировка для чтения и эксклюзивная блокировка для записи.
- Эффективно при большом количестве операций чтения и небольшом количестве операций записи.

Минусы:

- Больше потребление памяти и сложность в сравнении с обычной блокировкой.

StampedLock (4/9)

StampedLock:

Плюсы:

- Предоставляет три режима блокировки: чтение, запись и оптимистическое чтение.
- Оптимистическое чтение позволяет избежать блокировок в некоторых случаях.

Минусы:

- Сложнее в использовании, чем обычные блокировки.

Semaphore (5/9)

Semaphore:

Плюсы:

- Позволяет контролировать доступ к ресурсам, определяя количество потоков, которым разрешен доступ.

Минусы:

- Может быть сложным в использовании и поддержке, особенно в более сложных сценариях.

Condition (6/9)

Condition (условия):

Плюсы:

- Позволяет потокам ждать определенных условий перед выполнением определенных действий.

Минусы:

- Требует использования внутри блокировки, что может усложнить код.

Resources (1/10)

References

1. Java Concurrency in Practice (page 96- 120) in russian
2. Разница между Data Race и Race Condition / Хабр (habr.com)
3. Race Condition и Data Race | German Gorelkin Blog (ubiklab.net)
4. https://coderlessons.com/articles/java/uchebnik-po-parallelizmu-java-blokirovka-iavnye-blokirovki

Thank you!

Presented by **Nodirkhuja Tursunov**

(ameriqano@gmail.com)