Concorrência em Java

ACH 2003 — Computação Orientada a Objetos

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

Objetos Imutáveis

- Vimos que é preciso sincronizar o acesso à memória compartilhada quando usamos múltiplas threads
- Se o estado de um objeto não puder mudar após sua criação, eles não poderiam ser corrompidos pelas threads nem serem observados em um estado inconsistente
- Objetos assim são ditos objetos imutáveis

Estratégias para definir objetos imutáveis

1. não forneça métodos "*setters*"
2. torne todos os campos `final` e `private`
3. não permita que subclasses sobrescrevam os métodos. Você pode fazer isso definindo a classe como `final` e fazendo com que o construtor seja privado (instâncias são criadas por um `factory method`)
4. se sua instância fizer referência a outros objetos mutáveis, não permita que esses objetos sejam modificados:
   - não forneça métodos que permitem mudar os objetos mutáveis
   - não compartilhe as referências aos objetos mutáveis. Nunca armazene referência a objetos mutáveis que você não criou (passados no construtor). Se necessário, crie cópias e guarde referência às cópias. Do mesmo modo, crie cópias de objetos internos antes de devolvê-los em seus métodos

String é um exemplo de classe imutável

Objetos de alto nível para concorrência

- Até aqui vimos as construções de baixo nível que permitem a criação de problemas concorrentes
- Elas existem desde a criação de Java e funcionam bem, mas são mais difíceis de usar e é mais difícil de conseguir alto desempenho com elas
- Java (≥ 5) introduziu novas funcionalidades de alto nível para concorrência (pacote `java.util.concurrent`) e estruturas de dados concorrentes (no arcabouço de coleções):

  Objetos lock permitem escrever expressões idiomáticas que simplificam muitos apps concorrentes

  Executors API de alto nível para criar e gerenciar *threads*

  Coleções concorrentes que facilitam o gerenciamento de grandes coleções de dados e que reduzem a necessidade de sincronização

  Variáveis atômicas com funcionalidades que diminuem a sincronização e ajudam a evitar problemas de inconsistência da memória

  ThreadLocalRandom provê a geração eficiente de números aleatórios para múltiplas *threads*

Objetos lock

- O pacote `java.util.concurrent.locks` implementa objetos que fornecem formas diferentes de expressar o controle de concorrência em Java
- Oferecem a mesma funcionalidade dos monitores: só uma *thread* pode obter um `Lock` por vez, são reentrantes, também permitem uso de `wait`/`notify` com os objetos `Condition`
- Permitem que o código tome outra ação caso não seja possível obter um *lock* imediatamente

Corrigindo o exemplo de Deadlock

Situação onde duas ou mais *threads* ficam bloqueadas para sempre porque uma fica esperando a outra:

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }
```

```
    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

É extremamente provável que ambas as *threads* fiquem bloqueadas. Ambas ficarão esperando (para sempre) até que uma das *threads* saia de **bow**

Corrigindo o exemplo de Deadlock (II)

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

public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();

        public Friend(String name) { this.name = name; }
        public String getName() { return this.name; }
```

```
public boolean impendingBow(Friend bower) {
    Boolean myLock = false;
    Boolean yourLock = false;
    try {
        myLock = lock.tryLock();
        yourLock = bower.lock.tryLock();
    } finally {
        if (! (myLock && yourLock)) {
            if (myLock) {
                lock.unlock();
            }
            if (yourLock) {
                bower.lock.unlock();
            }
        }
    }
    return myLock && yourLock;
}
```

```
public void bow(Friend bower) {
    if (impendingBow(bower)) {
        try {
            System.out.format("%s: %s has"
                + " bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        } finally {
            lock.unlock();
            bower.lock.unlock();
        }
    } else {
        System.out.format("%s: %s started"
            + " to bow to me, but saw that"
            + " I was already bowing to"
            + " him.%n",
            this.name, bower.getName());
    }
}
```

Executors

- Em programas maiores, pode ser melhor separar o gerenciamento e criação de *threads* do resto da aplicação
- O pacote `java.util.concurrent` define 3 interfaces para classes que ajudam essa separação:

  Executor: uma interface para permitir lançar novas tarefas

  ExecutorService: uma subinterface de `Executor` que permite controlar o ciclo de vida das tarefas e do próprio executor

  ScheduledExecutorService: uma subinterface de `ExecutorService` que permite a execução de tarefas futuras e de tarefas periódicas

Executor

- Provê o método `execute` para facilitar a criação de uma *thread* para a execução de uma tarefa
- Seja `r` um `Runnable` e `e` um `Executor`:
  - o código `(new Thread(r)).start();`
  - pode ser substituído por `e.execute(r);`
- A ideia geral é que o executor encontre/crie uma *thread* e execute o `Runnable` imediatamente, mas como isso será feito dependerá da classe que implementar o `Executor`

ExecutorService

- Além do método `execute`, provê um método mais versátil chamado `submit`
  - `submit` aceita como argumento tanto objetos `Runnable`, quanto objetos `Callable`, que permitem que a tarefa devolva um valor
  - O método `submit` devolve uma instância de `Future`, que é usada para recuperar o valor devolvido pela tarefa `Callable`
- Provê métodos para invocar coleções de objetos `Callable` simultaneamente
- Também provê métodos para terminar o executor (para isso as tarefas devem tratar corretamente as interrupções)

ScheduledExecutorService

- Além dos métodos de `ExecutorService`, provê:
  - o método `schedule`, que executa um `Runnable` ou `Callable` após um tempo especificado
  - o método `scheduleAtFixedRate`, que executa uma tarefa pela primeira vez após o tempo (`initialDelay`) especificado, e depois a executa novamente periodicamente após o intervalo de tempo `period` (a tarefa é executada pela primeira vez após `initialDelay`, a segunda após `initialDelay+period`, a terceira após `initialDelay+2*period`, etc.
  - o método `scheduleWithFixedDelay` que executam tarefas repetidamente, com um atraso dado entre o término da última execução da tarefa e o início da próxima execução

Thread Pools

- A maioria das implementações de `Executor` em `java.util.concurrent` usa *thread pools*: um conjunto de *threads* pré-instanciadas
    - essas *threads* são chamadas também de *workers* e são independentes de seus `Runnable` ou `Callable`
- Usar *worker threads* minimiza o sobrecusto (*overhead*) da criação das *threads* (em uma aplicação grande, criar e destruir muitas *threads* causam um grande *overhead*), mas gastam mais memória
- É comum usarmos uma *thread pool* de tamanho fixo: há sempre um número fixo de *threads* em execução e, se uma *thread* for terminada durante a execução de uma tarefa, outra é instanciada em seu lugar
    - se houver mais tarefas do que *threads* disponíveis, as tarefas são colocadas em uma fila interna até que algum *worker* fique disponível

Criação de Executors

`Executors.newSingleThreadExecutor()` cria uma única *thread* em *background*

`Executors.newFixedThreadPool(int nThreads)` cria um número fixo de *threads* em *background*

`Executors.newCachedThreadPool()` *pool* que aumenta conforme a demanda

Fork/Join

- Implementação de `ExecutorService` projetada para trabalhos que podem ser quebrados recursivamente em pedaços menores
- Distribui as tarefas entre *workers* de uma *thread pool*, usando um algoritmo de *work stealing*
- Implementado pela classe `ForkJoinPool`, que executa processos do tipo `ForkJoinTask`

Esquema de uso básico

```
se (meu pedaço do trabalho for pequeno o suficiente)
  faça a tarefa toda de uma vez
senão
  divida meu trabalho em dois pedaços
  invoque a execução dos dois pedaços e espere pelos resultados
```

Exemplo de ForkJoinTask

```
static class SortTask extends RecursiveAction {
   static final int THRESHOLD = 1000;
   final long[] array; final int lo, hi;

   SortTask(long[] array, int lo, int hi) {
     this.array = array; this.lo = lo; this.hi = hi;
   }

   SortTask(long[] array) { this(array, 0, array.length); }

   protected void compute() {
     if (hi - lo < THRESHOLD)
       Arrays.sort(array, lo, hi); // ordena sequencialmente
     else {
       int mid = (lo + hi) >>> 1;
       invokeAll(new SortTask(array, lo, mid),
                 new SortTask(array, mid, hi));
       merge(lo, mid, hi); // como o merge de MergeSort
     }
   }
 }
```

Para executar: `new ForkJoinPool().invoke(new SortedTask(umArray));` (cria uma *thread* por processador disponível e executa a ordenação em paralelo)

Coleções Concorrentes

O pacote `java.util.concurrent` estende o arcabouço de coleções do Java com os seguintes tipos de coleções:

`BlockingQueue`: define uma estrutura FIFO que bloqueia (ou espera até um *timeout*) quando você tenta adicionar um item a uma fila cheia, ou recuperar um item de uma fila vazia — implementada por classes como `ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingDeque`, etc.

`ConcurrentMap`: uma subinterface de `java.util.Map` que define operações atômicas — implementada por `ConcurrentHashMap`

`ConcurrentNavigableMap`: uma subinterface de `ConcurrentMap` que permite buscas por respostas aproximadas — implementada por `ConcurrentSkipListMap`, uma versão concorrente de `TreeMap`

Variáveis atômicas

- O pacote `java.util.concurrent.atomic` define classes que implementam operações atômicas em variáveis
- As classes implementam `get` e `set` que funcionam como leituras e escritas em variáveis `volatile` e que estabelecem relações de *happens-before* entre sequências de chamadas
- Oferecem também o método `boolean compareAndSet(expectedValue, updateValue)` e operações aritméticas atômicas

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() { c.incrementAndGet(); }

    public void decrement() { c.decrementAndGet(); }

    public int value() { return c.get(); }
}
```

Números aleatórios concorrentes

- A implementação de `Math.random()` é devidamente sincronizada e pode ser usada por múltiplas *threads*
  - mas com muitas *threads* pode ficar meio lenta
- A classe `ThreadLocalRandom` é otimizada para reduzir a contenção em programas concorrentes
- Ex:

```
int r = ThreadLocalRandom.current().nextInt(4, 77);
```

Referências

- The Java Tutorials: Concurrency: `http://docs.oracle.com/javase/tutorial/essential/concurrency/`

E agora, José?

Lições a serem tiradas de COO

COO é sobre se preparar para se tornar um excelente programador

- Leia **muito** código de outras pessoas: são nos códigos bem escritos que aprendemos os melhores truques. Em particular, leia o código das classes básicas de Java
- Não se contente com o superficial: saber a sintaxe de uma linguagem de programação / conhecer uma API não implica em programar bem
- Código funcionando $\neq$ código pronto. Incorpore testes automatizados do início ao fim do processo de desenvolvimento e assegure-se de que seu código está legível (senão, refatore-o!)
- Programação é um trabalho para ser feito em equipe. É preciso aprender a se comunicar bem (UML) e a compartilhar código (git)

The Pragmatic Programmer i

O livro *The Pragmatic Programmer: From Journeyman to Master* sugere que você construa seu portfólio de conhecimentos assim:

- Aprenda uma nova linguagem por ano: linguagens diferentes resolvem os problemas de formas diferentes e ajuda a expandir seu modo de pensar e evitar cair na rotina
- Leia um livro técnico por trimestre: livrarias estão cheias de livros técnicos sobre os tópicos relacionados aos seus projetos. Depois que você dominar as tecnologias que estiver usando, comece a estudar coisas que não estejam relacionadas ao seus projetos
- Leia livros não-técnicos também: é importante se lembrar que computadores são usados por **pessoas**, pessoas cujas necessidades você está tentando solucionar

The Pragmatic Programmer ii

- Faça cursos: procure por cursos interessantes dentro e fora da universidade
- Participe de grupos de usuários e meetups: isolar-se pode ser fatal pra sua carreira, descubra no que as pessoas estão trabalhando fora do seu dia a dia
- Experimente ambientes diferentes: se você só trabalha com Windows, brinque com um Unix em casa. Se você está acostumado apenas com IDEs, experimente usar diferentes editores e makefile (ou vice-versa)
- Atualize-se: acompanhe revistas e jornais especializados como: Communications of the ACM[1], ACM Queue[2], IEEE Computer[3], IEEE Software[4], SBC Horizontes[5], Computação Brasil[6]

The Pragmatic Programmer iii

- Fique ligado: acompanhe fóruns de discussão de qualidade sobre tendências em tecnologia. Sugestões: https://news.ycombinator.com/, https://www.reddit.com/r/programming, https://www.reddit.com/r/compsci, https://lobste.rs/.

[1] https://cacm.acm.org/
[2] https://queue.acm.org/
[3] https://www.computer.org/csdl/magazine/co
[4] https://www.computer.org/csdl/magazine/so
[5] https://horizontes.sbc.org.br/
[6] https://www.sbc.org.br/publicacoes-2/298-computacao-brasil