

Aplicações concorrentes em Java

Desenvolvendo aplicações concorrentes estáveis e escaláveis

Implemente códigos com maior facilidade, em menos tempo, mais confiáveis, escaláveis e de manutenção simples, com a nova API Concurrent

RONALDO BLANC ROCHA

De que se trata o artigo:

Desenvolvimento de software com processamento concorrente usando a API `java.util.concurrent`. E a comparação entre o esforço utilizado para desenvolver um software concorrente utilizando as primitivas de concorrência e utilizando a API.

Para que serve:

Aumentar a qualidade, o desempenho e a escalabilidade de software com processamento concorrente. As classes da API são bem testadas e confiáveis, e tornam o código mais compreensivo, facilitando atividades de manutenção.

Em que situação o tema é útil:

Com a evolução dos computadores de múltiplos núcleos, e a crescente demanda por processamento, desenvolver softwares concorrentes com eficiência, facilidade, escalabilidade e boa manutenibilidade são alguns dos objetivos mais comuns para empresas de desenvolvimento de software.

Aplicações concorrentes em Java:

O Java, assim como outras linguagens, escolheu o modelo de threads para tratar tarefas que podem ser executadas simultaneamente. Esse modelo tem suas vantagens e desvantagens (assim como qualquer modelo). O Java sempre trouxe a concorrência como um recurso da linguagem, não algo externo. Mesmo assim, desenvolver aplicações concorrentes sempre foi uma tarefa muito complexa. Com a introdução da `java.util.concurrent`, a complexidade para desenvolver tais aplicações diminuiu muito. Utilizando essa ferramenta, mesmo programadores menos experientes podem escrever aplicações concorrentes com qualidade e segurança, obtendo ótimos resultados.

A evolução dos processadores com um único núcleo baseada em *clocks* com maior frequência atingiu o limite do aquecimento suportado e um custo/benefício inviável. Esses processadores enfrentavam ainda outros problemas, como alto consumo de energia e gargalo no acesso à memória. Portanto, novas alternativas para a melhoria de desempenho e consumo ganharam destaque, entre elas o paralelismo de *threads* e *cores*. Os processadores de múltiplos núcleos (MC) e a tecnologia SMT (*Simultaneous Multithreading*) permitem que tarefas sejam executadas em paralelo melhorando o aproveitamento dos recursos.

Para tirar-se maior proveito desses avanços, é necessário mudar o modo de pensar e escrever aplicações, substituindo-se o paradigma seqüencial pelo concorrente. Dentre os desafios que esse paradigma apresenta, alguns merecem destaque: identificar o que pode ser executado em paralelo (vamos chamar de *thread* cada parte do código com essa característica) e a sincronização dos threads.

O que são *Threads*?

Os sistemas operacionais, em sua maioria, suportam processos, isto é, programas independentes que rodam com algum grau de isolamento. *Thread* é um recurso que permite múltiplas atividades ocorrendo ao mesmo tempo em um mesmo processo. O Java foi uma das primeiras linguagens, com boa aceitação no mercado, que trouxe as threads como parte da linguagem, não as tratando apenas como um recurso controlado pelo sistema operacional.

As threads assim como os processos, são independentes e concorrentes, tendo variáveis locais e pilha (*stack*) próprias, mas compartilham a memória e estado do processo em que participam. Dessa forma, elas têm acesso aos mesmos objetos alocados no heap. Apesar de facilitar a troca de dados, deve-se tomar cuidado para garantir que isto não resulte em erros de concorrência (*data races*).

Todos os programas escritos em Java rodam *threads*, pelo menos uma (*main*). A JVM cria outras threads que geralmente não são percebidas, como: Garbage Collection (GC), *threads* da AWT/Swing; em servidores de aplicação conectores de HTTP, EJB etc.

A API do Java para *threads* é muito simples. Entretanto, escrever programas que as utilizem com eficiência é um trabalho complexo. Então, porque utilizá-las se são tão complicadas? Algumas das principais razões são:

- Aproveitar as vantagens dos computadores multiprocessados;
- Executar tarefas assíncronas ou em *background*;
- Tornar a interface com o usuário mais fluida;
- Simplificar a modelagem.

Antes da API **java.util.concurrent**, trabalhar com *threads* em Java exigia um grande esforço, pois os recursos disponíveis são muito simples e de baixo nível. No exemplo da (**Listagens 1 a 4**), apresenta-se a implementação de um problema clássico de produtor e consumidor, utilizando os recursos primitivos de concorrência.

A classe principal **ProducerConsumer** instancia alguns **Producers**, que geram dados, e **Consumers**, que os utilizam. Ambos se comunicam por intermédio da fila customizada (representada na classe **Queue**), implementada utilizando recursos primitivos do Java como **synchronized**, além dos algoritmos específicos a este tipo de fila. Por exemplo, o método **add()** não pode inserir elementos se a fila já estiver “cheia”, neste caso o recurso de bloqueio seguido de retry (ver o loop **while(isFull())** com um **wait()**) garante a integridade; assim, se a fila estiver cheia, a thread-produtor que está tentando inserir ficará em **wait()** até que alguma thread-consumidor remova um elemento com **remove()**, que ao final faz um **notifyAll()**, acordando o produtor para o próximo retry. Todo este código é bastante trabalhoso e complexo, além de limitado, em escalabilidade e outras qualidades.

A JDK 5 agregou melhorias significativas para o desenvolvimento de aplicações concorrentes, incluindo modificações na JVM, novos recursos de baixo nível para sincronização, classes de alto nível com bom desempenho e *thread-safe* (como thread pools), coleções concorrentes, semáforos,

locks e barreiras. Um dos objetivos desse artigo é, auxiliar na compreensão de como essas novas classes podem melhorar a produtividade, desenvolvendo códigos mais escaláveis, seguros e mais simples para dar manutenção.

Utilizando-se da nova API, pode-se trabalhar com recursos testados e confiáveis. Os problemas enfrentados com a programação concorrente, em sua maioria, são parecidos e, as soluções para os aspectos mais comuns fazem parte da nova API. Dessa forma, pode-se juntá-las como blocos para construir-se a aplicação.

Listagem 1. Produtor/Consumidor com Semáforo <Queue.java>

```
package br.com.jm.concurrent;

public class Queue {
    private static final Integer MAX_ITENS = 5000;
    private Integer itens = new Integer(0);
    private boolean firstThousand = false;

    private synchronized boolean isFull() {
        if (this.itens >= MAX_ITENS) {
            if (!this.firstThousand) this.firstThousand = true;
            return true;
        }
        else {
            return false;
        }
    }

    private synchronized boolean isEmpty() {
        if (this.itens <= 0) {
            return true;
        }
        else {
            return false;
        }
    }

    public synchronized void add() {
        while (isFull()) {
            try {
                synchronized (this) {
                    wait();
                    System.out.println(Thread.currentThread().getName() + " waiting");
                }
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        itens++;
        System.out.println(Thread.currentThread().getName()
            + " produziu um valor. Total [" + this.itens + "] ");
        synchronized (this) {
            notifyAll();
        }
    }

    public synchronized void remove() {
        while (isEmpty()) {
            try {
                synchronized (this) {
                    wait();
                    System.out.println(Thread.currentThread().getName() + " waiting");
                }
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        itens--;
        System.out.println(Thread.currentThread().getName()
            + " consumiu um valor. Total [" + this.itens + "]");
        synchronized (this) {
```

```

        notifyAll();
    }
}

public synchronized boolean isRemovable() {
    return this.firstThousand;
}
}

```

Listagem 2. Produtor/Consumidor com Semáforo <Producer.java>

```

package br.com.jm.concurrent;

public class Producer extends Thread {

    private Queue queue;

    public Producer(Queue queue, String string) {
        this.queue = queue;
        this.setName(string);
    }

    public void run() {
        while (true) {
            try {
                sleep(1);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
            queue.add();
        }
    }
}

```

Listagem 3. Produtor/Consumidor com Semáforo <Consumer.java>

```

package br.com.jm.concurrent;

public class Consumer extends Thread {
    private Queue queue;

    public Consumer(Queue queue, String string) {
        this.queue = queue;
        this.setName(string);
    }

    public void run() {
        while (true) {
            try {
                sleep(random());
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (queue.isRemovable())
                queue.remove();
        }
    }

    private long random() {
        Double random = Math.random() * 100;
        return (long) (random.intValue() % 10);
    }
}

```

Listagem 4. Produtor/Consumidor com Semáforo <ProducerConsumer.java>

```

package br.com.jm.concurrent;

public class ProducerConsumer {

    private static final int MAX_PRODUCER = 2;
    private static final int MAX_CONSUMER = 6;

    public static void main(String args[]) {
        Queue queue = new Queue();
        // Producers
    }
}

```

```

Producer producer = null;
for (int i = 0; i < MAX_PRODUCER; i++) {
    producer = new Producer(queue, "Producer " + i);
    producer.start();
}
Consumer consumer = null;
// Consumers
for (int i = 0; i < MAX_CONSUMER; i++) {
    consumer = new Consumer(queue, "Consumer " + i);
    consumer.start();
}
}
}

```

O conceito *Thread-safe*

Como é difícil definir se uma classe é *thread-safe*, alguns pontos devem ficar claros:

Para uma classe ser considerada *thread-safe*, primeiro ela deve funcionar bem em um ambiente não concorrente, ou seja, deve ser implementada de acordo com as especificações e qualquer operação (leitura ou alteração de valores utilizando membros ou métodos públicos) realizada em objetos dessas classes não devem colocá-los em estado inválido. Além disso, a classe deve funcionar bem em ambientes *multi-thread*. Nesse cenário, a estratégia de agendamento e tempo de execução, ou sincronizações adicionais necessárias ao código, não devem influenciar o bom funcionamento da classe. Para que as operações executadas por um objeto *thread-safe* atendam esses critérios, elas devem estar bem encapsuladas e a integridade dos dados deve ser mantida de forma transparente.

Antes do Java 5, o único mecanismo para garantir que as classes atendiam o conceito *thread-safe* era a primitiva **synchronized**. Assim, as variáveis compartilhadas entre múltiplas *threads* precisavam ser sincronizadas para que as operações de leitura e alteração de valores fossem coordenadas.

Mesmo que não exista nenhuma linha de código que indique explicitamente o uso de *threads* em sua aplicação, o uso de *frameworks* e alguns recursos podem exigir que as classes que os utilizam sejam *thread-safe*. Mas desenvolver classes *thread-safe* exige muito mais atenção e análise do que classes não *thread-safe*.

Servlets containers, por exemplo, criam muitas *threads* e uma determinada Servlet pode ter acessos simultâneos para atender múltiplas requisições, por isso, uma classe Servlet deve ser *thread-safe*.

A **Listagem 5** mostra o exemplo de uma Servlet não *thread-safe* (a princípio o código parece correto) que salva o endereço das máquinas que acessam o servidor. Entretanto, a Servlet não é *thread-safe*, pois a classe **HashSet** também não é. Nessas condições, poderíamos perder algum dado ou corromper o estado da coleção. O modo simples de corrigir esse problema, utilizando a nova API, seria usar uma das coleções seguras, criando o objeto **ipAddressSet** da seguinte forma:

```
private Set<String> ipAddressSet = Collections.synchronizedSet(new HashSet<String>());
```

É muito interessante observar que a nova API simplifica o trabalho do desenvolvedor, ao mesmo tempo em que exige um maior conhecimento sobre concorrência, mesmo para um exemplo tão simples quanto esse.

Listagem 5. Servlet não *thread-safe* <ServletNaoSeguro.java>

```

public class ServletNaoSeguro extends HttpServlet{

    private Set<String> ipAddressSet = new HashSet<String>();

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        String ipAddress = request.getRemoteAddr();
        if (ipAddressSet != null)
            ipAddressSet.add(ipAddress);
    }
}

```

}

Coleções Seguras

Na plataforma Java (desde a JDK 1.0) existem coleções *thread-safe* como **Hashtable** e **Vector**. A API de Collections (introduzida no JDK 1.2) introduz muitas coleções sem esta característica (como **ArrayList**, **HashSet** etc.), mas podemos transformá-las em *thread-safe*, com o uso das *factories*:

- **Collections.synchronizedSet**;
- **Collections.synchronizedMap**;
- **Collections.synchronizedList**.

Essas coleções geralmente têm baixa performance e alguns problemas, principalmente quando o processamento depende de dados processados anteriormente. Chamadas aos métodos **Iterator.hasNext()** e **Iterator.next()** podem lançar uma **ConcurrentModificationException**, pois alguma thread pode ter alterado a coleção. Analogamente, testar se um objeto existe na lista antes de inseri-lo, ou recuperar o tamanho da coleção (por exemplo um **List.size()**), pode provocar “*data races*”. Esses cenários são conhecidos pelo nome “*conditionally thread-safe*”.

Além desses recursos, a **java.util.concurrent** traz novas coleções como: **ConcurrentHashMap**, **CopyOnWriteArrayList** e **CopyOnWriteArraySet**. Os propósitos dessas classes são alta performance e alta escalabilidade para classes *thread-safe*. O uso dessas classes cumprirá seus objetivos em todas as aplicações que não tenham a necessidade de bloquear toda a coleção para evitar alterações. A **ConcurrentHashMap**, por exemplo, é feita para cenários de alta concorrência e leva tão a sério essa idéia que muito raramente bloqueia em locks, mesmo por pouco tempo.

As coleções da nova API retornam “*weakly consistent iterators*” ao contrario das coleções da **java.util** que retornam “*fast-fail iterators*”. Os “*fast-fail iterators*” lançam uma **ConcurrentModificationException** (como explicado anteriormente) pois consideram que a lista não pode ser alterada, seja por outra thread ou pelo mesmo, enquanto está sendo percorrida. Já para os “*weakly consistent iterators*” se um objeto foi removido, ele não será retornado e se um objeto foi adicionado, ele pode ou não ser retornado. Existe, porém, uma garantia de que o mesmo objeto não será retornado mais de uma vez em uma mesma iteração. Para coleções com ordem estável, os objetos retornados virão na ordem correta, não importa como a coleção foi alterada.

As listas baseadas em arrays (como **Vector**) ou criadas com uma das *factories* disponíveis na nova API (como **Collections.synchronizedList()**) retornam “*fast-fail iterators*” e para evitar o cenário “*conditionally thread-safe*” a *thread* que está iterando sobre a lista teria que copiar toda a lista ou bloqueá-la, sendo qualquer uma dessas soluções muito custosa. Desde a introdução da **java.util.concurrent**, você deve encarar estes recursos como meros quebra-galhos – talvez para código legado que você não tenha tempo de revisar mais profundamente com a nova API.

As classes **CopyOnWriteArrayList** e **CopyOnWriteArraySet** sempre criam uma cópia da lista quando ela é alterada (**add()**, **set()** ou **remove()**), assim as threads que estão iterando sobre a lista continuarão trabalhando sobre a mesma lista que começaram. Mesmo com o custo de copiar a lista, na maioria dos casos, o numero de iterações é bem maior que o de modificações. E o custo de tais cópias é menor do que o normal devido à otimização *copy-on-write*: grosso modo, cada thread só copia de forma incremental os elementos que alterou, não a coleção inteira. Para esses cenários as novas classes oferecem melhor performance e permitem maior concorrência que as outras.

A classe **ConcurrentHashMap** permite maior concorrência que as outras alternativas de mapas *thread-safe*. O modo utilizado pelas outras implementações é sincronizar todos os métodos, o que nos leva a dois problemas:

1. **Falta de escalabilidade**, pois uma única thread pode acessar a coleção durante o mesmo período;
2. **A possibilidade de falha na segurança do acesso** (“*conditionally thread-safe*”) que exige sincronização externa às classes.

Com a técnica de sincronizar tudo, independente das operações que outras threads queiram realizar na coleção, estas precisam esperar pela primeira thread terminar sua execução. Já a **ConcurrentHashMap** permite várias operações de leitura quase concomitantes, operações de leitura e gravação simultâneas (pode-se ler e gravar dados ao mesmo tempo) e múltiplas gravações concorrentes (várias threads gravando ao mesmo tempo).

A JDK 5 incluiu uma nova estrutura de dados, a **Queue**, que é uma interface bem simples. A **java.util.concurrent** traz a **BlockingQueue**, que facilita a utilização das filas tornando simples o tratamento de erros comuns em relação à essa estrutura de dados, como: limites máximo e mínimo ultrapassados (no caso de filas com limite de elementos definido) e consumo excessivo de memória (no caso de filas sem limites). Simplesmente bloquear a *thread* também permite maior controle sobre o fluxo de dados (se os produtores estão colocando objetos na fila muito mais rápido que os consumidores, bloquear os produtores tornará o consumo mais rápido). As implementações dessa interface são:

- **LinkedBlockingQueue** – Fila FIFO com ou sem limites baseada em nós ligados (como se fosse uma **LinkedList**);
- **PriorityBlockingQueue** – Fila sem limites e com prioridades(não FIFO);
- **ArrayBlockingQueue** – Fila FIFO com limites baseada num array;
- **SynchronousQueue** – Uma fila “degenerada” que não comporta nem um único elemento, mas facilita a sincronização entre threads (permite o *hand-off*, ou transferência direta, de elementos diretamente do produtor para o consumidor).

No exemplo da **Listagens 6 a 8** apresenta-se a implementação de um produtor/consumidor utilizando **BlockingQueue**.

Esse exemplo tem praticamente o mesmo nível de complexidade que o das **Listagens 1 a 4**, mas o código em si é muito mais simples. Na classe principal **ProducerConsumer** temos uma **ArrayBlockingQueue** com tamanho fixo que controla os números máximos e mínimos de elementos na fila (assim como a fila da **Listagem 1**), temos ainda um **Executor (CachedThreadPoo)** que lida com a concorrência, utilizando um pool de threads que podem ser reutilizadas, maximizando assim a performance, pois alocar recursos para uma thread não é um processo simples. A fila comporta no máximo 10 elementos e o produtor só pode inserir dados se existir espaço livre (assim como o primeiro exemplo), a própria **BlockingQueue** controla esses limites. O consumidor utilizando o método **take()** não corre riscos de exceções por falta de objetos na fila, pois esse método espera até que exista um elemento na fila para retornar. Todo o processo ocorre sem a utilização de qualquer primitiva de concorrência, muito utilizadas em todas as classes das **Listagens 1 a 4**, deixando claro como a nova API pode facilitar o nosso trabalho como desenvolvedores.

FIFO – Define a ordem de entrada e saída dos elementos da fila, nesse caso o primeiro a entrar é o primeiro a sair (first in, first out).

Listagem 6. Producer/Consumer utilizando BlockingQueue <ProducerConsumer.java>

```
package br.com.jm.concurrent.bq;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ProducerConsumer {
    public static void main(String... args) throws Exception {
        BlockingQueue<String> q = new ArrayBlockingQueue<String>(10);
        ExecutorService executor = Executors.newCachedThreadPool();
        //2 produtores
        executor.execute(new Producer(q));
        executor.execute(new Producer(q));
        //6 consumidores
        executor.execute(new Consumer(q));
        executor.execute(new Consumer(q));
    }
}
```

```

        executor.execute(new Consumer(q));
        executor.execute(new Consumer(q));
        executor.execute(new Consumer(q));
        executor.execute(new Consumer(q));
    }
}

```

Listagem 7. *Producer/Consumer utilizando BlockingQueue* <Producer.java>

```

package br.com.jm.concurrent.bq;

import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable {
    private final BlockingQueue<String> queue;

    public Producer(BlockingQueue<String> q) {
        queue = q;
    }

    public void run() {
        while (true) {
            try {
                queue.put("Java Magazine");
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

Listagem 8. *Producer/Consumer utilizando BlockingQueue* <Consumer.java>

```

package br.com.jm.concurrent.bq;

import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable {
    private final BlockingQueue<String> queue;

    public Consumer(BlockingQueue<String> q) {
        queue = q;
    }

    public void run() {
        while (true) {
            try {
                System.out.println(queue.take());
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

O framework Executor e Thread Pools

As implementações de *thread pools* disponíveis na API são bem flexíveis e existe um framework para controlar a execução de *tasks* que implementam a interface **Runnable** (ou uma nova variante, **Callable**). No framework Executor existe a interface **Executor**, que é bem simples. O contrato dessa interface define um método para executar implementações de **Runnable**. Em qual *thread* a task será executada não faz parte do contrato, dependerá da implementação de **Executor** sendo utilizada, ou seja, a forma de execução é desacoplada da forma de chamada. Como a interface **Executor** se preocupa com o modo da chamada, fica muito mais fácil ajustar os parâmetros de execução (alterando valores, como: tamanho do pool, prioridade, timeouts, etc.) com pequenas alterações no código.

Dica: Thread pool é uma boa solução para os problemas de alocação e liberação de recursos enfrentados em aplicações multi-thread. Como nos thread pools as threads são reutilizadas, o custo de alocação de cada thread é dividido entre as tarefas que ela executará. Como a thread já está criada a aplicação pode responder uma requisição (ou seja, iniciar uma tarefa) imediatamente.

Existem várias implementações de **Executors**, cada uma com sua estratégia de execução. Elas são criadas por *factories* disponíveis em métodos estáticos da classe **Executors**, os principais são:

- **Executors.newCachedThreadPool()**: Um *pool* sem limite de tamanho, mas que reutiliza *threads* quando disponíveis. Se não existem *threads* disponíveis, cria uma nova e adiciona ao *pool*. Se uma *thread* não for utilizada durante um minuto, será finalizada e removida do *pool*;
- **Executors.newFixedThreadPool(int n)**: Um *pool* que reutiliza um certo grupo de *threads* com tamanho definido. Se o número de *threads* ultrapassar o número definido, as *threads* esperam até um espaço ser liberado;
- **Executors.newSingleThreadExecutor()**: Um *pool* que utiliza uma única *thread* para executar as tarefas. Dessa forma, as tarefas são executadas de forma sequencial. A *thread* de eventos do Swing, por exemplo, seria um **SingleThreadExecutor**.

A estratégia de execução define em qual thread uma task será executada. Define o modo de execução, ou seja, como os recursos serão utilizados (ex.: memória) e como se comportar em casos de sobrecarga.

Dica: Uma pergunta muito comum é: Qual tamanho deve ter o Thread pool?

A resposta depende, principalmente, do hardware e o tipo de tarefa (task) que será executada. Existe uma lei que pode ajudar bastante: A lei de Amdahl. Para aplicá-la, precisa-se do número de processadores (N) existentes na máquina que executará a tarefa. Precisa-se ainda de aproximações dos tempos médio de processamento (TP) e de espera (TE) para a tarefa. O TP é o tempo gasto para a conclusão da tarefa e o TE é o tempo gasto pela thread esperando para iniciar a execução. Assim, o tamanho do Thread pool deve ser algo em torno de:

$$N * (1 + TE / TP)$$

Obs.: Os tempos TP e TE, não precisam ser precisos, para muitos valores de TP e TE, pode-se atingir um bom desempenho.

A interface Future

Essa interface pode representar uma *task* que completou a execução, que está em execução, ou que ainda não começou. Pode-se ainda cancelar uma *task* que ainda não terminou, descobrir quando a *task* terminou ou foi cancelada e receber ou esperar pelos valores retornados.

Uma implementação dessa interface é a **FutureTask**, que encapsula **Runnable** ou **Callable** obedecendo aos contratos das interfaces **Future** e **Runnable**, assim, pode ser executada facilmente por um **Executor**. Alguns métodos do **Executor** (como **ExecutorService.submit()**) retornam **Future** além de executar a *task*.

Para recuperar os resultados de uma *task*, usamos **Future.get()**, que lança uma **ExecutionException** caso a execução da *task* tenha provocado uma exceção; caso a *task* ainda não tenha terminado o método ficará esperando o retorno do resultado, se a execução já acabou retorna o valor imediatamente.

*As diferenças entre as interfaces **Runnable** e **Callable** são: Na interface **Callable<T>** o método **T call()**, além de devolver um resultado do tipo definido **T**, lança exceção. O método **void run()** da interface **Runnable** não retorna valores e não lança exceção.*

A interface `CompletionService`

Permite que o processamento dos resultados seja desacoplado da execução da *task*. Sua interface define o método `submit()` para indicar as *tasks* que serão executadas e define o método `take()` para recuperar as *tasks* terminadas. Os dois métodos retornam uma implementação de `Future`, o que facilita o processo de recuperação de resultados. Usando a interface `CompletionService` a aplicação pode ser estruturada facilmente com o pattern *Produtor/Consumidor* (utilizado como exemplo nas **Listagens 1 a 4 e 6 a 8**), onde o produtor cria e submete as *tasks* e o consumidor recupera os resultados para processá-los.

Uma implementação dessa interface é a `ExecutorCompletionService`, que usa um `Executor` para processar as *tasks*. No exemplo das **Listagens 9 a 11** apresenta-se a implementação de um produtor/consumidor utilizando essa interface.

Esse exemplo usa muitos recursos da nova API para resolver praticamente o mesmo problema que os outros exemplos, mas nele as interfaces `ExecutorCompletionService`, `Future` e `Callable` deixam bem claro como o código pode ser bem simples e executar tarefas complexas, como submeter tarefas e recuperar os valores ao final da execução das mesmas. Para esse exemplo foi escolhido o `Executor` `FixedThreadPool`, com tamanho máximo de 10 threads. O produtor submete uma tarefa para a `ExecutorCompletionService` chamando o método `addTask()`, enquanto o consumidor espera para obter o resultado com o método `getResult()`. O `getResult()` retorna uma instância da interface `Future`, onde o método `get()` retorna o resultado imediatamente, se ele existir, ou espera até o final da execução tarefa para retornar.

Listagem 9. *Producer/Consumer* utilizando `CompletionService` <ProducerConsumer.java>

```
package br.com.jm.concurrent.cs;

public class ProducerConsumer {
    private static AtomicInteger count = new AtomicInteger(0);

    public static void main(String args[]) throws InterruptedException,
        ExecutionException
    {
        CompletionService<String> completionService = new ExecutorCompletionService<String>(
            Executors.newFixedThreadPool(10));
        Producer producer = new Producer(completionService);
        Consumer consumer = new Consumer(completionService);
        while (true) {
            producer.addTask("Java Magazine [" + safeIncrement() + "]");
            System.out.println(consumer.getResult().get());
        }
    }

    private static int safeIncrement() {
        if (count.equals(Integer.MAX_VALUE)) {
            return 0;
        }
        else {
            return count.addAndGet(1);
        }
    }
}
```

Listagem 10. *Producer/Consumer* utilizando `CompletionService` <Producer.java>

```
package br.com.jm.concurrent.cs;

public class Producer {

    private CompletionService<String> completionService;

    public Producer(CompletionService<String> completionService) {
        this.completionService = completionService;
    }

    public void addTask(final String value) {
        completionService.submit(new Callable<String>() {
            public String call() throws Exception {
                return value;
            }
        });
    }
}
```

```
    }  
    });  
}
```

Listagem 11. *Producer/Consumer* utilizando *CompletionService* <Consumer.java>

```
package br.com.jm.concurrent.cs;  
  
public class Consumer {  
  
    private CompletionService<String> completionService;  
  
    public Consumer(CompletionService<String> completionService) {  
        this.completionService = completionService;  
    }  
  
    public Future<String> getResult() throws InterruptedException {  
        return completionService.take();  
    }  
}
```

Conclusões

Pode-se perceber que desenvolver com a nova API é muito mais simples, apesar de exigir ainda conhecimentos sólidos sobre programação concorrente e a necessidade de códigos específicos que indiquem para a JVM como lidar com essas situações. A mesma segurança que existe em um código não concorrente em Java (ou seja, que a memória será alocada e desalocada de modo correto e eficiente, as melhorias que o JIT Compiler proporciona, entre outros) é o que a **java.util.concurrent** proporciona para os códigos concorrentes. Claro que seria melhor não existir nenhuma preocupação em deixar explícitos os códigos para o controle da sincronização e concorrência e deixar para a JVM resolver, assim como ela resolve os problemas de alocação de memória, mas não é tão simples (como alocação de memória), controlar a sincronização e concorrência de um processo. Esse é um dos motivos para o assunto ser um dos mais discutidos hoje em dia. Com essa API podemos substituir a maior parte das primitivas de concorrência por classes de alto nível, bem testadas e de bom desempenho.

Links

<http://www.jcp.org/jsr/detail/166.jsp>

JSR 166: Concurrency Utilities

<http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>

Tutorial da Sun sobre concorrência em Java

<http://gee.cs.oswego.edu/dl/concurrency-interest/>

Site do Doug Lea sobre concorrência

http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=practice:

Coluna do Brian Goetz no developerWorks

Livros (opcional)

Concurrent Programming in Java, Doug Lea, Addison-Wesley, 1999

Livro do professor Doug Lea, que escreveu a API que serviu de base para a `java.util.concurrent`.

Java Concurrency In Practice, Brian Goetz, Addison-Wesley, 2006

Livro do engenheiro da Sun Brian Goetz, muito conhecido e comentado.

Saiba mais

www.devmedia.com.br/articles/viewcomp.asp?comp=10017

Vídeo - Novidades do NetBeans 6.5 - Parte 1

www.devmedia.com.br/articles/viewcomp.asp?comp=8471

Java Magazine 53 - Perspectivas para um Mundo Paralelo

www.devmedia.com.br/articles/viewcomp.asp?comp=9420

Java Magazine 57 - Programando com Pools

www.devmedia.com.br/articles/viewcomp.asp?comp=10197

Java Magazine 59 - Quick Update: A Crise dos Threads



Ronaldo Blanc Rocha (ronaldo@summa.com.br) é Consultor Java EE pela Summa Technologies do Brasil, atuando há 9 anos com Java/Java EE.