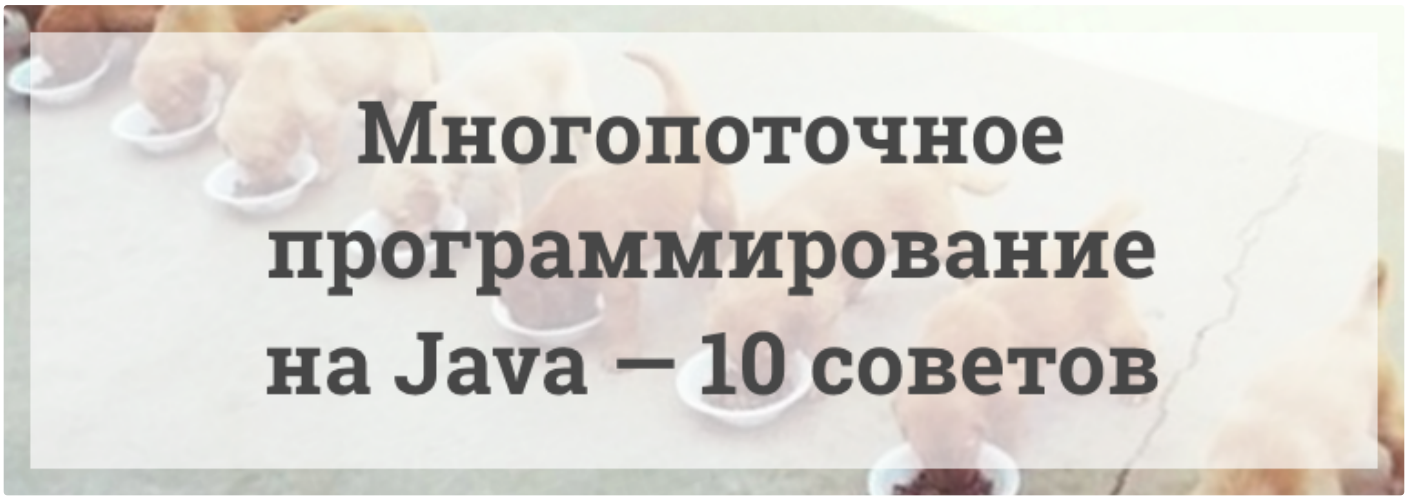
10 советов по многопоточному программированию на Java

31 мая 2015 в 14:35, Переводы (https://tproger.ru/category/translations/) 9 минут 20 587



(https://tproger2.azureedge.net/wp-content/uploads/2015/05/pic06.png)

Рассказывает Дж. Пол, автор блога Java Revisited

Написание параллельного кода – непростая задача, а проверка его корректности – задача еще сложнее. Несмотря на то, что Java предоставляет обширную поддержку многопоточности и синхронизации на уровне языка и API, на деле же оказывается, что написание корректного многопоточного Java-кода зависит от опыта и усердности конкретного программиста. Ниже изложен набор советов, которые помогут вам качественно повысить уровень вашего многопоточного кода на Java. Некоторые из вас, возможно, уже знакомы с этими советами, но никогда не помешает освежать их в памяти раз в пару лет.

Многие из этих советов появились в процессе обучения и практического программирования, а также после прочтения книг «Java concurrency in practice» (https://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20) и «Effective Java» (https://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20). Я советую прочитать первую каждому Java-программисту два раза; да, всё правильно, ДВА раза.

Параллелизм – запутанная и сложная для понимания тема (как, например, для некоторых – рекурсия), и после однократного прочтения вы можете не до конца всё понять.

Единственная цель использования параллелизма – создание масштабируемых и быстрых приложений, но при этом всегда следует помнить, что скорость не должна становиться помехой корректности. Ваша Java-програма должна удовлетворять своему инварианту независимо от того, запущена ли она в однопоточном или многотопочном виде. Если вы новичок в параллельном программировании, для начала ознакомьтесь с различными проблемами, возникающими при параллельном запуске программ (например: взаимная блокировка, состояние гонки, ресурсный голод и т.д.).

1. Используйте локальные переменные

Всегда старайтесь использовать локальные переменные вместо полей класса или статических полей. Иногда разработчики используют поля класса, чтобы сэкономить память и переиспользовать переменные, полагая, что создание локальной переменной при каждом вызове метода может потребовать большого количества дополнительной памяти. Одним из примеров такого использования может послужить коллекция (Collection), объявленная как статическое поле и переиспользуемая с помощью метода clear(). Это переводит класс в общее состояние, которого он иметь не должен, т.к. изначально создавался для параллельного использования в нескольких потоках. В коде ниже метод `execute()` вызывается из разных потоков, а для реализации нового функционала потребовалась временная коллекция. В оригинальном коде была использована статическая коллекция (List), и намерение разработчика были ясны — очищать коллекцию в конце метода `execute()`, чтобы потом можно было её заново использовать. Разработчик полагал, что его код потокобезопасен, потому что CopyOnWriteArrayList потокобезопасен. Но это не так — метод `execute()` вызывается из разных потоков, и один из потоков может получить доступ к данным, записанным другим потоком в общий список. Синхронизация, предоставляемая CopyOnWriteArrayList в данном случае недостаточна для обеспечения инвариантности метода `execute()`.

```
1  public static class ConcurrentTask {
2      private static List temp = Collections.synchronizedList(new ArrayList());
3
4      @Override
5      public void execute(Message message) {
6          // Используем локальный временный список
7          // List temp = new ArrayList();
8
9          // Добавим в список что-нибудь из сообщения
10         temp.add("message.getId()");
11         temp.add("message.getCode()");
12
13         temp.clear(); // теперь можно переиспользовать
14     }
15 }
```

Проблема: Данные одного сообщения попадут в другое, если два вызова `execute()` «пересекаются», т.е. первый поток добавит id из первого сообщения, затем второй поток добавит id из второго сообщения (это произойдёт ещё до очистки списка), таким образом данные одного из сообщений будут повреждены.

Варианты решений:

1. Добавить блок синхронизации (https://java67.blogspot.sg/2013/01/difference-between-synchronized-block-vs-method-java-example.html) в ту часть кода, где поток добавляет что-то во временный список и очищает его. Таким образом, другой поток не сможет получить доступ к списку, пока первый не закончит работу с ним. В таком случае эта часть кода будет однопоточной, что уменьшит производительность приложения в целом.
2. Использовать локальный список в место поля класса. Да, это увеличит затраты памяти, но избавит от блока синхронизации и сделает код более читаемым. Также вам не придётся беспокоиться о временных объектах – о них позаботится сборщик мусора.

Здесь представлен только один из случаев, но при написании параллельного кода лично я предпочитаю локальные переменные полям класса, если последних не требует архитектура приложения.

2. Предпочитайте неизменяемые классы изменяемым

Самая широко известная практика в многопоточном программировании на Java – использование неизменяемых *(immutable)* классов. Неизменяемые классы, такие как String, Integer и другие упрощают написание параллельного кода в Java, т.к. вам не придётся беспокоиться о состоянии объектов данных классов. Неизменяемые классы уменьшают количество элементов синхронизации в коде (https://javarevisited.blogspot.sg/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html). Объект неизменяемого класса, будучи однажды созданным, не может быть изменён. Самый лучший пример такого класса – строка (`java.lang.String`). Любая операция изменения строки в Java (перевод в верхний регистр, взятие подстроки и пр.) приведёт к созданию нового объекта String для результата операции, оставив исходную строку нетронутой.

3. Сокращайте области синхронизации

Любой код внутри области синхронизации не может быть исполнен параллельно, и если в вашей программе 5% кода находится в блоках синхронизации, то, согласно закону Амдала, производительность всего приложения не может быть улучшена более, чем в 20 раз. Главная причина этого в том, что 5% кода всегда выполняется последовательно. Вы можете уменьшить это количество, сокращая области синхронизации – попробуйте использовать их только для критических секций. Лучший пример сокращения областей синхронизации – блокировка с двойной проверкой, которую можно реализовать в Java 1.5 и выше с помощью volatile переменных.

4. Используйте пул потоков

Создание потока *(Thread)* — дорогая операция. Если вы хотите создать масштабируемое Java-приложение, вам нужно использовать пул потоков. Помимо тяжеловесности операции создания, управление потокам вручную порождает много повторяющегося кода, который, перемешиваясь с бизнес-логикой, уменьшает читаемость кода в целом. Управление потоками – задача фреймворка, будь то инструмент Java или любой другой, который вы захотите использовать. В JDK есть хорошо организованный, богатый и полностью протестированный фреймворк, известный как Executor framework (https://javarevisited.blogspot.sg/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html), который можно использовать везде, где потребуется пул потоков.

5. Используйте утилиты синхронизации вместо wait() и notify()

В Java 1.5 появилось множество утилит синхронизации, таких как CyclicBarrier, CountDownLatch и Semaphore. Вам всегда следует сначала изучить, что есть в JDK для синхронизации, прежде чем использовать `wait()` и `notify()`. Будет намного проще реализовать шаблон читатель-писатель с помощью BlockingQueue, чем через `wait()` и `notify()`. Также намного проще будет подождать 5 потоков (https://javarevisited.blogspot.sg/2012/07/countdownlatch-example-in-java.html) для завершения вычислений, используя CountDownLatch, чем реализовывать то же самое через `wait()` и `notify()`. Изучите пакет `java.util.concurrent`, чтобы писать параллельный код на Java лучшим образом.

6. Используйте BlockingQueue для реализации Producer-Consumer

Этот совет следует из предыдущего, но я выделил его отдельно ввиду его важности для параллельных приложений, используемых в реальном мире. Решение многих проблем многопоточности основано на шаблоне Producer-Consumer, и BlockingQueue – лучший способ реализации его в Java. В отличие от Exchanger, который может быть использовать в случае одного писателя и читателя, BlockingQueue может быть использована для правильной обработки нескольких писателей и читателей.

7. Используйте потокобезопасные коллекции вместо коллекций с блокированием доступа

Потокобезопасные коллекции предоставляют большую масштабируемость и производительность, чем их аналоги с блокированием доступа (`Collections.synchronizedCollection` и пр.). ConcurrentHashMap, которая, по моему мнению, является самой популярной потокобезопасной коллекцией, демострирует лучшую производительность, чем блокировочные HashMap или Hashtable, в случае, когда количество читателей превосходит количество писателей. Другое преимущество

потокобезопасных коллекций состоит в том, что они реализованы с помощью нового механизма блокировки (`java.util.concurrent.locks.Lock`) и используют нативные механизмы сихнронизации, предоставленные низлежащим аппаратным обеспечением и JVM. Вдобавок используйте CopyOnWriteArrayList вместо `Collections.synchronizedList`, если чтение из списка происходит чаще, чем его изменение.

8. Используйте семафоры для создания ограничений

Чтобы создать надёжную и стабильную систему, у вас должны быть ограничения на ресурсы (базы данных, файловую систему, сокеты и т.д.). Ваш код ни в коем случае не должен создавать и/или использовать бесконечное количество ресурсов. Семафоры (`java.util.concurrent.Semaphore`) — хороший выбор для создания ограничений на использование дорогих ресурсов, таких как подключения к базе данных (кстати, в этом случае можно использовать пул подключений). Семафоры помогут создать ограничения и заблокируют потоки в случае недоступности ресурса.

9. Используйте блоки синхронизации вместо блокированных методов

Данный совет расширяет совет по сокращению областей синхрониации. Использование блоков синхронизации – один из методов сокращения области синхронизации, что также позволяет выполнить блокировку на объекте, отличном от текущего, представленного указателем `this`. Первым кандитатом должна быть атомарная переменная, затем `volatile` переменная, если они удовлетворяют ваши требования к синхронизации. Если вам требуется взаимное исключение, используйте в первую очередь ReentrantLock, либо блок `synchronized`. Если вы новичок в параллельном программировании, и не разрабатываете какое-либо жизненно важное приложение, можете просто использовать блок `synchronized` — так будет безопаснее и проще.

10. Избегайте использования статических переменных

Как показано в первом совете, статические переменные, будучи использованными в параллельном коде, могут привести к возникновению множества проблем. Если вы всё-таки используете статическую переменную, убедитесь, что это константа либо неизменяемая коллекция. Если вы думаете о том, чтобы переиспользовать коллекцию с целью экономии памяти, вернитесь ещё раз к первому совету.

11. Используйте Lock вместо synchronized

Последний, бонусный совет, следует использовать с осторожностью. Интерфейс Lock — мощный инструмент, но его сила влечёт и большую ответственность. Различные объекты Lock на операции чтения и записи позволяют реализовывать масштабируемые структуры данных, такие как ConcurrentHashMap, но при этом требуют большой осторожности при своём программировании. В отличие от блока synchronized, поток не

освобождает блокировку автоматически. Вам придётся явно вызывать `unlock()`, чтобы снять блокировку. Хорошей практикой является вызов этого метода в блоке finally, чтобы блокировка завершалась при любых условиях:

```
lock.lock();
try {
    //do something ...
} finally {
    lock.unlock();
}
```

Заключение

Вам были представлены советы по написанию многопоточного кода на Java. Ещё раз повторюсь, никогда не помешает перечитывать «Java concurrency in practice» и «Effective Java» время от времени. Также можно вырабатывать нужный для параллельного програмирования способ мышления, просто читая чужой код и пытаясь визуализировать проблемы во время разработки. В завершение спросите себя, каких правил вы придерживаетесь, когда разрабатываете многопоточные приложения на Java?

Перевод статьи Top 10 Java Multithreading and Concurrency Best Practices (https://javarevisited.blogspot.ru/2015/05/top-10-java-multithreading-and.html)

Java (https://tproger.ru/tag/java/), Лучшая практика (https://tproger.ru/tag/best-practice/), Многопоточность (https://tproger.ru/tag/multithreading/)