Das Java Memory Modell

Klaus Kreft & Angelika Langer

Diese pdf-Datei enhält die Manuskripte einer Artikelserie zum Thema "Java Memory Modell", die wir für das JavaMagazin in den Jahren 2008 und 2009 geschrieben haben.

Urheber: Klaus Kreft & Angelika Langer
Website: www.AngelikaLanger.com
Twitter: @AngelikaLanger
Kontakt: http://www.angelikalanger.com/Forms/Contact.html

Die Artikelserie ist online verfügbar unter http://www.angelikalanger.com/Articles/EffectiveJava.html#JMM.
Diese pdf-Datei befindet sich unter: http://www.AngelikaLanger.com/Articles/PDF/Java-Memory-Model.pdf.

Java Memory Modell

Teil 1: Einführung: Wozu braucht man volatile?

Klaus Kreft & Angelika Langer

Mit diesem Beitrag wollen wir eine kleine Reihe über Aspekte des Java-Memory-Modells (JMM) beginnen. Kenntnisse des Memory-Modells werden für die Programmierung mit mehreren Threads gebraucht und detaillierte Kenntnisse werden insbesondere durch die zunehmende Verwendung von Multicore-Prozessoren immer wichtiger. Deshalb wollen wir einige der wesentlichen Aspekte des Memory-Modells erläutern. Wir beginnen mit dem `volatile`**-Schüsselwort. Was bedeutet** `volatile`**? Wofür braucht man es? Worauf muss man achten?**

Unter Java-Programmierern ist allgemein bekannt, dass man bei der Programmierung mit mehreren Threads besonders aufpassen muss, wenn diese parallel ablaufenden Threads auf gemeinsam verwendete, veränderliche Daten (engl. *shared mutable data*) zugreifen. Dabei gibt es eine Reihe von Aspekten, die man als Programmierer im Auge behalten muss. Das bekannteste Problem ist die *Race Condition:* wenn der Zugriff auf die gemeinsam verwendeten Daten in mehreren Schritten erfolgt, dann ist der Zugriff unterbrechbar. Ein Thread etwa macht die ersten Schritte des Zugriff, wird mittendrin vom Thread-Scheduler verdrängt, ein anderer Thread kommt zum Zuge, greift auf die halbveränderten Daten zu und modifizert sie womöglich, wird seinerseits unterbrochen, der erste Thread kommt wieder dran. Er macht weiter, als sei nichts gewesen. Das Resultat dieser konkurrierenden Zugriffe ist unvorhersehbar.

Die übliche Abhilfe ist *Synchronisation*. Mit Hilfe von einem *Lock* (auch *Mutex* genannt) wird dafür gesorgt, dass der Zugriff auf die gemeinsam verwendeten veränderlichen Daten ununterbrechbar ist. Das geht so: alle beteiligten Threads, die auf die Daten zugreifen wollen, benutzen ein bestimmtes Lock-Objekts. Vor jedem Zugriff auf die kritischen Daten wird das Lock angefordert, nach Beendigung aller Zugriffsschritte wird das Lock wieder frei gegeben. Da ein Lock immer nur einen Thread als Besitzer haben kann, muss beim Anfordern gewartet werden, bis der aktuelle Besitzer das Lock wieder frei gibt. Auf dieser Weise ist gesichert, dass immer nur ein Thread zu einer Zeit auf die gemeinsam verwendeten veränderlichen Daten zugreifen kann, weil das Lock besetzt ist und alle anderen Threads warten müssen, bis der Thread fertig ist mit seinem Zugriff.

Diese Grundlagen sind sicher jedem Java-Entwickler geläufig, denn die Instrumente für die Synchronisation sind in Java direkt in die Sprache eingebettet worden in Form des `synchronized`-Schlüsselworts und durch die Tatsache, dass an jedem Objekt automatisch ein Lock dranhängt, das man zwar nicht sieht, das aber immer vorhanden ist und implizit über das `synchronized`-Schlüsselwort angesprochen wird. Als Alternative gibt es seit Java 5 auch noch die etwas flexibleren, expliziten Locks, siehe Interface `Lock` im Package `java.util.concurrent.locks`.

Nun ist die Verwendung von Locks aber teuer und deswegen kommt als Optimierung das `volatile`-Schlüsselwort ins Spiel. Die Kosten der Synchronisation mit Hilfe von Locks bestehen zum einen im Aufwand, den das Anfordern und Freigeben von Locks für die Virtuelle Maschine und das Betriebsystem bedeuten und zum anderen in der Tatsache, dass Synchronisation Wartezustände auslöst, die den Durchsatz der Anwendung reduzieren.

Beim Anfordern und Freigeben von Locks hat das Laufzeitsystem nämlich ein Menge zu tun: es werden Systemressourcen angelegt und weggeräumt, es werden Threads in Warteschlangen gestellt oder aus Wartezuständen aufgeweckt, es passieren Kontextwechsel, Daten-Caches müssen abgeglichen werden - all das führt dazu, dass Synchronisation Zeit und Aufwand kostet, der die Performance der Anwendung reduziert.

Daneben wirkt sich Synchronisation nachteilig auf den Durchsatz der Anwendung aus. Es kann passieren, dass der synchronisierte Zugriff auf gemeinsam verwendete veränderliche Daten zu einem echten Engpass werden kann. Wenn viele Threads gleichzeitig auf gemeinsam verwendete Daten zugreifen wollen und immer nur einer das Lock bekommt und alle anderen warten müssen, dann entsteht ein Stau, der sich negativ auf den Durchsatz der Anwendung auswirkt. Mit anderen Worten, Synchronisation skaliert nicht beliebig und je weniger Synchronisation gebraucht wird, desto besser ist es. Also ist das Motto: Synchronisation reduzieren, wo immer es geht.

Atomare Zugriffe

Wie oben erläutert wird die Synchronisation gebraucht, um komplexere Zugriffe auf gemeinsam verwendete veränderliche Daten ununterbrechbar zu machen. Wenn die betreffenden Daten aber elementar sind und der Zugriff darauf schon von Natur aus ununterbrechbar (man sagt auch *atomar*) ist, dann braucht man doch keine Synchronisation, richtig? Also, wenn zum Beispiel die gemeinsam verwendeten Daten lediglich aus einem `boolean` bestehen, dann sind Zugriffe wie Lesen oder Schreiben atomar. Das garantiert die Sprachdefinitionin Kapitel 17 (siehe /JLS/). Dort ist festgelegt, dass lesende und schreibende Zugriffe auf Variablen von primitivem Typ (außer `long` und `double`) atomar sind. Deshalb wird in der Praxis vielfach die Synchronisation weggelassen, wenn es um konkurriende Zugriffe auf Variablen von primitivem Typ geht.

Hier ist ein typisches Beispiel:

```
public class Processor {
    private boolean connectionPrepared = false;
```

```
    public void prepareConnection() {
        // ... open connection ...
        connectionPrepared = true;
    }
    public void start() throws InterruptedException {
        // ... various initializations ...
        while (!connectionPrepared )
            Thread.sleep(500);
        // ... start actual processing ...
    }
}
```

Wir haben hier zwei Methoden der Klasse `Processor`, die beide auf das Feld `connectionPrepared` zugreifen. Die Idee ist, dass die `start`-Methode in einem Thread ausgeführt wird, der alle vorbereitenden Arbeiten anstößt und dann abwartet, bis alle Vorbereitungen abgeschlossen sind, ehe er die eigentliche Verarbeitung beginnt. Die beiden Threads kommunizieren miteinander über gemeinsam verwendete veränderliche Daten, nämlich das `boolean` Feld `connectionPrepared`. Der eine Thread setzt `connectionPrepared` auf `true`, wenn er fertig ist, und der andere Thread beobachtet, ob `connectionPrepared` auf `true` gesetzt wurde, um dann mit der eigentlichen Arbeit zu beginnen. Das Lesen und Verändern des `boolean` Feldes ist garantiert atomar, deshalb wird keine Synchronisation verwendet.

Leider wurde hier ein wesentlicher Aspekt übersehen.

Sequential Consistency

Der Autor dieser kleinen Klasse ist augenscheinlich davon ausgegangen, dass der eine Thread irgendwann einmal das `boolean` Feld `connectionPrepared` setzen wird und dass der andere Thread den veränderten Wert dann sehen kann. Eine derartige Garantie gibt es in Java aber gar nicht. Es ist keineswegs so, dass ein Thread immer sofort sehen kann, was ein anderer Thread im Speicher gemacht hat. Es kann passieren, dass der eine Thread das `boolean` Feld `connectionPrepared` auf `true` gesetzt hat und der andere Thread diese Änderung nie zu sehen bekommt.

Das Schwierige an der Sache ist, dass der beschriebene Effekt nicht auftreten muss, aber auftreten kann. Das heißt, die Klasse hat einen Fehler, der sich aber unter Umständen gar nicht bemerkbar macht. In manchen existierenden Anwendungen schlummern derartige Fehler, die bisher einfach noch nicht entdeckt wurden. Nun ist es so, dass auf Maschinen mit nur einem Single-Core-Prozessor die Fehler häufig tatsächlich nicht auftreten. Aber in einer Multi-Prozessor- oder Multicore-Umgebung kann der Fehler dann plötzlich doch passieren. Da heute Dual-Core-Prozessoren Standard sind, kann man in der Tat beobachten, wie Anwendungen, die auf einer Single-Core-Maschine tadellos funktioniert haben, plötzlich ganz seltsame Fehler aufweisen, sobald sie auf einer Maschine mit einem Multi-Core-Prozessor ablaufen.

Welchen Fehler hat der Autor der Klasse gemacht? Er hat Annahmen über das Verhalten der Virtuellen Maschine gemacht, die unzutreffend sind. Was er unterstellt hat, wird allgemein als *Sequential Consistency* bezeichnet. Sequential Consistency bedeutet, dass ein Thread, der später drankommt, sehen kann, was die Threads vor ihm im Speicher an den gemeinsam verwendeten Daten gemacht haben. Das ist ein wunderbar simples mentales Modell, das aber leider von Java nicht unterstützt wird. *Wir haben a priori keine Sequential Consistency in Java.*

Es gibt in der Sprachspezifikation stattdessen eine Reihe von Garantien für die Reihenfolge von Operationen und auch für die Sichtbarkeit von Memory-Modifikationen, aber sie sind wesentlich schwächer als unserer Intuition entsprechende Sequential Consistency. Wir wollen jetzt nicht das gesamte Memory-Modell von Java aufrollen, sondern nur eine einzige der Regeln aus der Sprachspezifikation herausgreifen, nämlich die Garantien für `volatile`-Variablen.

Sichtbarkeitsregeln für Volatile-Variablen

Für eine `volatile`-Variable ist garantiert, dass ein Thread, der die Variable liest, den Wert bekommt, den zuletzt zuvor ein anderer Thread derselben Variablen zugewiesen hat. Es ist außerdem garantiert, dass der Wert, den ein Thread in einer `volatile`-Variablen ablegt, allen anderen Threads zugänglich gemacht wird. Das heißt, für `volatile`-Variablen haben wir die gewünschte Sequential Consistency. Allerdings gilt dies nur für die Variable selbst: bei einer Variablen von einem Referenztyp gilt es nur für die Adresse, nicht für das referenzierte Objekt. Die Garantien sind streng genommen sogar noch umfangreicher: beim Schreiben auf eine `volatile`-Variable wird nicht nur der neue Inhalt eben jener `volatile`-Variablen sichtbar gemacht, sondern alle Modifikation, die der Thread zuvor im Speicher gemacht hat. Das interessiert uns aber im Moment nicht. Da das Thema etwas umfangreicher ist, werden wir die Details in einem anderen Beitrag näher erläutern. Kehren wir lieber zu unserem fehlerhaften Beispielcode zurück.

Man könnte den Fehler ganz einfach dadurch beheben, dass man das `boolean` Feld `overHeated` als `volatile` deklariert:

```
public class Processor {
    private volatile boolean connectionPrepared = false;
    public void prepareConnection() {
        // ... open connection ...
        connectionPrepared = true;
    }
    public void start() throws InterruptedException {
        // ... various initializations ...
        while (!connectionPrepared )
            Thread.sleep(500);
        // ... start actual processing ...
    }
}
```

Jetzt ist die Klasse korrekt, weil nun garantiert ist, dass die Modifikation, die der eine Thread mit Hilfe der `prepareConnection`-Methode am `volatile boolean connectionPrepared` vornimmt, dem anderen Thread sichtbar gemacht wird. Mit dieser `volatile`-Deklaration kann es nicht mehr passieren, dass der eine Thread das `boolean`-Feld ändert und der andere Thread es gar nicht mitbekommt.

Sichtbarkeitsregeln für Synchronisation

Wir wollen nicht verschweigen, dass man das Problem auch anders lösen kann, nämlich indem man den Zugriff auf das `connectionPrepared`-Feld synchronisiert:

```
public class Processor {
    private boolean connectionPrepared = false;
    public synchronized void prepareConnection() {
        // ... open connection ...
        connectionPrepared = true;
    }
    public synchronized void start() throws InterruptedException {
        // ... various initializations ...
        while (!connectionPrepared )
            Thread.sleep(500);
        // ... start actual processing ...
    }
}
```

Synchronisation hat nicht nur Auswirkungen auf die Ununterbrechbarkeit einer Sequenz von Operationen, sondern hat zusätzlich Auswirkungen auf die Sichtbarkeit von Speichermodifikationen. Wenn ein Lock freigegeben wird, dann ist garantiert, dass alle Modifikationen, die der betreffende Thread im Speicher gemacht hat, allen anderen Threads sichtbar gemacht werden. Umgekehrt ist garantiert, dass ein Thread beim Erhalt eines Locks alle Modifkationen im Speicher zu sehen bekommt, die ein anderer Thread bewirkt hat, der dasselbe Lock vorher gehalten hat. Das heißt, im Falle von Synchronisation gilt wieder die intuitive Sequential Consistency. Wer immer und überall korrekt synchronisiert, braucht kein `volatile` und wird nie Probleme damit haben, dass ein Thread nicht sehen kann, was ein anderer gemacht hat.

Die fehlende Sequential Consistency macht sich erst dann bemerkbar, wenn es unsynchronisierte Zugriffe auf Variablen gibt, die nicht `volatile` sind.

Unsere ursprüngliche Überlegung war aber, dass Synchronisation teuer ist und wenn möglich vermieden werden sollte. Da hier die Zugriffe auf das `boolean`-Feld atomar sind, wird die Synchronisation nicht gebraucht, um für Ununterbrechbarkeit des Datenzugriffs zu sorgen. Deshalb hatten wir die Synchronisation absichtlich weggelassen. Wenn man eine solche Optimierung vornimmt, dann muss man aber das `boolean`-Feld als `volatile` deklarieren, um für die Sichtbarkeit der Modifikationen an diesem Feld zu sorgen.

Zusammenfassung

Wir haben in diesem Beitrag gesehen, dass man Synchronisation eliminiert, um die Performance- und Scalability-Kosten der Synchronisation zu vermeiden. Das kommt immer dann in Frage, wenn der Zugriff auf eine gemeinsam verwendete veränderliche Variable sowieso schon atomar ist und deshalb für die Ununterberechbarkeit des Zugriff keine Synchronisation gebraucht wird. Sobald man aber unsynchronisiert auf eine gemeinsam verwendete veränderliche Variable zugreift, ist nicht mehr gesichert, dass ein Thread sehen kann, was ein anderer Thread zuvor in der Variablen abgelegt hat. Solche Variablen,

auf die unsynchronisiert zugegriffen wird, müssen als `volatile` deklariert werden, damit Modifikationen, die ein Thread an der Variablen vornimmt, garantiert allen anderen Threads sichtbar gemacht werden.

In den nächsten Beitrag wollen wir das Thema vertiefen und uns ansehen, was eigentlich hinter diesen Sichtbarkeitsregeln steckt und wie das mit `volatile`-Referenzvariablen ist, welche anderen Garantien das Memory-Model sonst noch zu bieten hat, wie das mit `final`-Variablen ist, was atomic-Variablen sind und was sie wiederum mit `volatile` zu tun haben.

Verweise

/JLS/ Java Language Specification, 3rd Edition
Chapter 17: Threads and Locks
http://java.sun.com/docs/books/jls/third_edition/html/memory.html

---

Klaus Kreft arbeitet als Entwickler und Consultant, derzeit für SEN (Siemens Enterprise Communications GmbH & Co. KG). Kontakt: klaus.kreft@siemens.com

Angelika Langer arbeitet selbständig als Trainer mit einem eigenem Curriculum von Java- und C++-Kursen. Kontakt: www.AngelikaLanger.com

---

Allgemeine Einleitung zur Kolumne:

Liebe Leser!

Wir freuen uns, in Zukunft regelmäßig im JavaMagazin unter dem Titel "Effective Java" über Fallstricke und aktuelle Themen aus dem Bereich des Java-Sprachkerns berichten zu dürfen. Wir setzen damit eine Artikelserie fort, die wir im Jahr 2002 unter dem Titel "Effective Java" begonnen haben und die bis vor Kurzem im JavaSpektrum erschienen ist. Wer sich für die bereits 30-40 schon erschienenen Beiträge interessiert, kann sie unter www.AngelikLanger.com/Articles/EffectiveJava.html finden. Thematisch konzentrieren wir uns auf den Kern der Sprache und des JDK, also auf den Teil von Java, der jeden Java-Entwickler betrifft. Oft machen wir auf Stolpersteine in der Sprache aufmerksam und erläutern, wie man damit umgeht und wie die entsprechenden Best-Practice-Tipps dazu aussehen. Wir erläutern neue Sprachmittel, wenn welche hinzukommen, und betrachten gelegentlich auch andere Java-Themen, die nicht so geläufig sind, weil sie nicht täglich verwendet werden. Über Feedback unserer Leser freuen wir uns natürlich. Wir hoffen, dass auch für die Leser des JavaMagazins der eine oder andere Beitrag nützlich für die Arbeit sein wird und wünschen viel Vergnügen beim Lesen.

Klaus Kreft & Angelika Langer

Java Memory Modell

Teil 2: Das Java-Memory-Modell im Überblick

Klaus Kreft & Angelika Langer

Im letzten Beitrag unserer Reihe über Aspekte des Java-Memory-Modells (JMM) haben wir erläutert, dass man das `volatile`-Schüsselwort zum Zwecke der Optimierung verwendet, um die relative teure Synchronisation von konkurrierenden Zugriffen auf gemeinsam verwendete veränderliche Daten zu vermeiden. Dabei haben wir die Sichtbarkeitsregeln erwähnt, die sich im Zusammenhang mit `volatile` und Synchronisation aus dem Java-Memory-Modell ergeben (siehe /EFF/). In diesem Beitrag wollen wir einen Überblick über das Memory-Modell geben.

Im Zusammenhang mit `volatile` und Synchronisation haben wir den Begriff der *Sequential Consistency* erwähnt. Das ist ein Modell, mit dem sich viele Java-Entwickler das Multithreading in Java vorstellen, obwohl Java gar keine Sequential Consistency unterstützt. Wenn man dennoch so programmiert, als gäbe es Sequential Consistency in Java, dann können Fehler entstehen wie jener, den wir im letzten Beitrag besprochen haben.

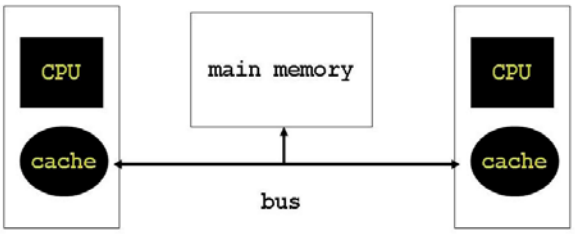
Das Modell der Sequential Consistency ist ein relativ einfaches Datenkonsistenzmodell. Es ähnelt der Funktionsweise von Multithreading auf einer Single-CPU-Umgebung. Die Vorstellung ist: die Threads laufen nicht wirklich parallel, sondern es gibt einen Thread-Scheduler, der den einzelnen Threads abwechselnd Zeitscheiben der CPU zuteilt. Ein Thread darf ein paar Operationen ausführen, wird dann verdrängt, es kommt ein anderer Threads dran, der wiederum ein paar Operationen machen darf, usw., so dass die einzelnen Threads ihre Operationen in einer sequenziellen Reihenfolge ausführen. Damit verbunden ist die Vorstellung, dass Threads, die später drankommen, Modifikationen im Speicher sehen können, die von Threads vorgenommen wurden, die vorher dran waren. Das ist ein einfaches Konsistenzmodell, das aber in Java nicht unterstützt wird.

Bei einem Konsistenzmodell geht es ganz allgemein (unabhängig von Java) um Regeln für die Zugriffe auf den Speicher. Wenn sich der Programmierer an die Regeln hält, dann gibt ihm das System (in unserem Falle Java und seine virtuelle Maschine) Garantien für die Effekte von Speicherzugriffen, damit der Programmierer weiss, was zur Laufzeit geschehen wird, und er die Effekte seiner Speicheroperationen vorhersehen kann. In High-Level-Sprachen wie Java müssen der Compiler und das Laufzeitsystem dafür sorgen, dass die High-Level-Sprachkonstrukte gemäß den Regeln des Konsistenzmodells in Low-Level-Operationen umgesetzt werden.

Auch Java hat ein Konsistenzmodell. Es ist aber nicht das Modell der Sequential Consistency, sondern Java hat ein eigenes Memory-Modell, das als JMM (= Java Memory Model) bezeichnet wird. Seine Regeln sind deutlich anders und schwächer als die der Sequential Consistency.

Wir wollen in diesem Beitrag einen Überblick über die Regeln des JMM geben. Das Thema ist ein wenig theoretisch und es ist nicht unmittelbar einsichtig, was all die Regeln für die Praxis der Java-Programmierung bedeuten. Trotzdem wollen wir erst einmal einen Überblick über das Modell als solches geben, ehe wir in nachfolgenden Beiträgen dieser Reihe die einzelnen Aspeke noch einmal näher auf ihre Bedeutung für die Praxis untersuchen.

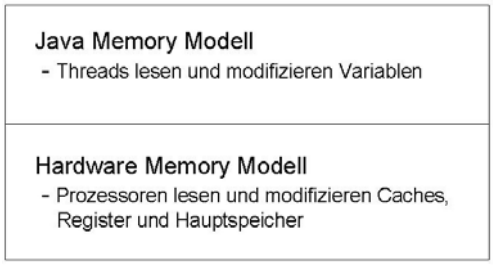
Das Java-Memory-Modell



Das Memory-Modell in Java ähnelt einer abstrakten SMP (= symmetric multi processing)-Maschine: die Threads laufen parallel und konzeptionell haben alle Threads Zugriff auf einen gemeinsamen Hauptspeicher (main memory), in dem die gemeinsam verwendeten Variablen abgelegt sind. Daneben hat jeder Thread einen eigenen lokalen Speicherbereich (cache), in den er Variablen hineinladen und lokal bearbeiten kann. Das Zurückschreiben der lokalen Daten in den Hauptspeicher (flush) und das Hereinladen von Daten aus dem Hauptspeicher (refresh) muss nach den Regeln des JMM geschehen. Das JMM beschreibt nun, in welcher Reihenfolge Aktionen passieren und welche Aktionen einen Flush oder Refresh auslösen.

Eine dieser Regeln besagt zum Beispiel, dass beim Start eines Threads alle relevanten Daten aus dem Hauptspeicher in den lokalen Arbeitsspeicher des Threads geladen werden. Dann darf der Thread mit diesen lokalen Daten arbeiten und muss gar nicht mehr ins Main Memory schauen, weil er die Daten im Cache hat. Am Ende des Threads muss der gesamte lokale Arbeitsspeicher des Threads wieder in den Hauptspeicher zurückgeschrieben werden. Daraus ergibt sich das Verhalten, dass wir auch intuitiv erwarten. Wenn ein Thread mit `join` auf das Ende eines anderen Threads wartet, kann der wartende Thread sehen, welche Modifikationen der andere, bereits beendete Thread gemacht hat.

Das JMM ist auch wieder nur ein Modell, mit dem sich der Java-Programmierer das Verhalten von Threads in einer JVM erklären kann. In Wirklichkeit muss die virtuelle Maschine die Regeln des JMM auf die Hardware abbilden, die ihr eigenes Hardware-Memory-Modell hat. Die heutigen Multicore-Prozessoren arbeiten nicht nur mit einem Cache pro Prozessorkern, sondern teilweise mit mehreren Ebenen von Caches und komplexeren Caching-Mechanismen, als sie das JMM vorsieht. Deshalb fällt der virtuellen Maschine die Aufgabe zu, mit geeigneten Anweisungen an die Hardware die Regeln des JMM zu implementieren.



Sichtbarkeitsregeln im JMM

Das Memory-Modell von Java regelt drei Dinge:

- *Atomicity*. Welche Operationen sind atomar, d.h. werden nicht durch andere Threads unterbrochen?
- *Ordering*. In welcher Reihenfolge passieren die Aktionen?
- *Visibility*. Wann werden Modifikationen im Speicher anderen Threads sichtbar gemacht?

Wir wollen an dieser Stelle nicht das komplette Memory-Modell erläutern. Nur ganz kurz:

Bei der Atomicity geht es zum Beispiel darum, dass der Zugriff auf Variablen von primitivem Typ (außer `long` und `double`) sowie auf Referenzvariablen ununterbrechbar ist. Gleiches gilt für `volatile`-Variablen (diesmal inklusive `long` und `double`). Die Operationen auf atomaren Variablen im Package `java.util.concurrent.atomic` sind ununterbrechbar. Gleiches gilt für einige der Operationen der Concurrent Collections im Package `java.util.concurrent`, z.B. die Methode `putIfAbsent` der `ConcurrentMap`. Bei Referenzvariablen darf man nicht vergessen, dass stets nur der Zugriff auf die Referenz selbst, d.h. auf die Adresse, atomar ist, nicht etwa der Zugriff auf das referenzierte Objekt. Ansonsten sind die Regeln zur Atomarität relativ einfach zu verstehen.

Beim Ordering geht es darum, unter welchen Umständen ein Thread die Effekte von Operationen, die ein anderer Thread ausführt, in der Reihenfolge sehen kann, in der der andere Thread die Operationen vorgenommen hat. Im allgemeinen ist nämlich ein weitrechendes Re-Ordering erlaubt, an dem sich der Compiler, die virtuelle Maschine und die Hardware beteiligen. Das Re-Ordering führt dazu, dass zwar innerhalb eines Threads klar ist, in welcher Reihenfolge die Effekte der Operationen entstehen, aber ein anderer Thread, der sich das anschaut, sieht die Effekte u.U. in einer anderen Reihenfolge, als sie produziert wurden. Es gibt im JMM Regeln für das Ordering, aber sie sind komplex und damit schwer zu verstehen. Deshalb sehen wir uns das Ordering nicht jetzt, sondern in einem späterem Beitrag genauer an.

Bei der Visibility geht es darum, ob und wann Modifikation am Speicher, die ein Thread gemacht hat, den anderen Threads sichtbar werden. Die Sichtbarkeitsregeln des JMM geben eine Reihe von Garantien, von denen wir einige auch schon im letzten Artikel erwähnt haben:

- *Threadstart- und -ende*. Der Start eines Threads löst einen Refresh des threadlokalen Arbeitsspeichers aus dem Hauptspeicher aus, das Ende des Thread einen Flush. Das entspricht der Intuition. Wenn zum Beispiel ein Thread mit `join` auf das Ende eines anderen Threads wartet, kann der wartende Thread sehen, welche Modifikationen der andere, bereits beendete Thread an gemeinsam verwendeten Daten gemacht hat.

- *Synchronisation*. Der Erhalt eines Locks löst einen Refresh aus, das Freigeben des Locks löst einen Flush aus. Das gilt für die alten impliziten, aber auch für die neuen expliziten Locks. Auch dieses Verhalten entspricht unserer Intuition. Alle Threads, die dasselbe Lock verwenden, durchlaufen die Sequenz der mit diesem Lock synchronisierten Anweisungen nicht parallel sondern nacheinander. Das heißt, es ist klar, dass zunächst ein Thread auf gemeinsam verwendete Daten zugreift und erst danach ein anderer. Da beim Freigeben des Locks der eine Thread alle lokalen Daten in den Hauptspeicher zurück schreiben muss und der nächste Thread beim Erhalt des Locks seinen lokalen Arbeitsspeicher aus dem Hauptspeicher auffrischen muss, sieht der zweite Thread alle Modifikationen, die der erste Thread an den gemeinsam verwendeten Daten vorgenommen hat.

- *Lese- und Schreibzugriff auf `volatile`-Variablen*. Das Lesen einer `volatile`-Variablen löst einen Refresh aus, das Modifizieren einer `volatile`-Variablen löst einen Flush aus. Das bedeutet, dass ein Thread, der den Inhalt einer `volatile`-Variablen liest, den Wert nicht aus seinem Arbeitsspeicher holen darf, sondern ihn aus dem Main Memory holen muss. Dabei muss er nicht nur den Inhalt der betreffenden `volatile`-Variablen holen, sondern er muss seinen

gesamten Arbeitsspeicher auffrischen. Auf diese Weise sieht er alle Modifikation an gemeinsam verwendeten Variablen, die zuvor ein anderer Thread gemacht hat, der auf dieselbe `volatile`-Variablen vorher schreibend zugegriffen hat. Denn beim schreibenden Zugriff muss der andere Thread seinen gesamten Arbeitspeicher in den Hauptspeicher zurückgeschrieben haben. Was das genau in der Praxis bedeutet, sehen wir uns in einem späteren Beitrag noch einmal genauer an.

- *Lese- und Schreibzugriff auf atomare Variablen.* Mit den atomaren Variablen sind die Abstraktionen aus dem Package `java.util.concurrent.atomic` gemeint. Das Lesen einer atomaren Variablen per `get`-Methode löst einen Refresh aus, das Modifizieren per `set`-Methode löst einen Flush aus. Darüber hinaus gibt es Methoden, die beides bewirken, z.B. `compareAndSet` und `getAndIncrement`. Mit anderen Worten, atomare Variablen haben Speichereffekte wie die `volatile`-Variablen und haben darüber hinaus atomare read-and-modify-Operationen, die die `volatile`-Variablen nicht haben. Auch dazu später mehr.

- *Erstmaliger Lesezugriff auf `final`-Variablen.* `final`-Variablen werden bekanntlich spätestens im Konstruktor mit ihrem konstanten Wert initialiert. Das Ende der Konstruktion löst einen partiellen Flush aus, bei dem die `final`-Variablen und alle "abhängigen" Objekte in den Hauptspeicher zurückgeschrieben werden. Die "abhängigen" Objekte sind jene, die von einer `final`-Variablen aus per Referenz erreichbar sind. Der erste lesende Zugriff auf eine `final`-Variable löst einen partiellen Refresh aus, bei dem die `final`-Variable und alle "abhängigen" Objekte in den Arbeitsspeicher geladen werden. Ein erneuter Refresh erfolgt nicht, weil die Variable einen konstanten Inhalt hat, der sich nicht mehr ändert. Auch das wollen wir uns in einem späteren Beitrag im Detail ansehen.

Wie man sieht, gibt es ein ganze Reihe von Sichtbarkeitsregeln, die man kennen sollte. Nun mag sich der eine oder andere Leser fragen, was ihn `volatile`-, `final`- oder atomare Variablen angehen. Möglicherweise im Moment nichts, aber das muss nicht so bleiben. Wer ausschließlich mit Synchronisation arbeitet und `volatile`-, `final`- oder atomare Variablen nicht verwendet, der muss sich über die Sichtbarkeitsregeln nicht den Kopf zerbrechen. Wie wir im letzten Beitrag aber schon erwähnt haben, kann Synchronisation negative Auswirkungen auf die Performance und die Skalierbarkeit von Anwendungen haben. Wenn das der Fall ist, dann wird man versuchen, die Synchronisation durch andere Mittel zu ersetzen, und dann kommen `volatile`-, `final`- oder atomare Variablen als Optimierungstechniken ins Spiel.

Zusammenfassung

Wir haben uns angesehen, was das Java-Memory-Modell überhaupt ist; es geht dabei um Regeln für Speicherzugriffe. Wir haben daran erinnert, dass wir keine Sequential Consistency in Java haben, sondern stattdessen das Java-Memory-Modell (JMM) mit seiner SMP-artigen Vorstellung von Hauptspeicher und threadlokalen Arbeitsspeichern. Das JMM regelt Ununterbrechbarkeit von Operationen, Sichtbarkeit von Speichereffekten und die Reihenfolge von Operationen. Wir werden uns bei nächsten Mal überlegen, warum es überhaupt wichtig ist, das JMM und seine Regeln zu kennen - ehe wir uns in nachfolgenden Beiträgen in die Details vertiefen.

Verweise

/EFF/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/JLS/ Java Language Specification, 3rd Edition
Chapter 17: Threads and Locks
http://java.sun.com/docs/books/jls/third_edition/html/memory.html

Java Memory Modell

Teil 3: Die Kosten der Synchronisation

Klaus Kreft & Angelika Langer

In den bisherigen beiden Beitrag unserer Reihe über Aspekte des Java-Memory-Modells (JMM) haben wir uns die Grundzüge des Modells und seine Sichtbarkeitsregeln angesehen (siehe /EFF2/). Weiter haben wir ein Beispiel für einen typischen Fehler betrachtet, der sich aus der Mißachtung des JMM ergeben kann (siehe /EFF1/). In diesem Beitrag wollen wir nun genauer darauf eingehen, warum man die Regeln des JMM überhaupt kennen sollte. Vielfach wird ausschließlich mit Synchronisation gearbeitet, so dass volatile-, final- und atomare Variablen kaum eine Rolle spielen. Wozu soll man sich also mit den JMM-Regeln dafür befassen? In diesem Beitrag wollen wir uns ansehen, welche Schwachpunkte Synchronisation hat und warum sie in Zukunft in zunehmendem Maße durch andere Techniken, die volatile-, final- und atomare Variablen benötigen, ersetzt werden wird.

Synchronisation von konkurrienden Zugriffen auf gemeinsam verwendete veränderliche Daten mit Hilfe von Locks ist in vielen Fällen unerlässlich, um die Konsistenz der Daten sicher zu stellen. Leider sind mit der Synchronisation Kosten verbunden. Zum einen sind es Performance-Kosten, die einfach dadurch entstehen, dass Synchronisation mit Arbeit verbunden ist, die das Laufzeitsystem erbringen muss. Zum anderen sind es Kosten, die durch einen Mangel an Skalierbarkeit entstehen

Performance-Einbußen durch Synchronisation

Das Anfordern und Freigeben eines Locks löst eine Reihe von Aktionen im Laufzeitsystem aus, die CPU-Zeit und damit Ablauf-Performance kosten.

So muss beim Anfordern eines Locks die zu dem Lock gehörende Warteschlange gefunden, abgefragt und ggf. modifiziert werden. In der Warteschlange stehen all die Threads, die darauf warten, das Lock nach seiner Freigabe zu bekommen. Die Verwaltung dieser Warteschlange kostet CPU-Zeit und Ressourcen.

Die tatsächliche Zuteilung eines Locks an einen Thread löst nicht nur den Update von Verwaltungsinformation in der virtuellen Maschine aus (das Lock ist jetzt besetzt und diese Information muss vermerkt werden), sondern auch *Memory Barriers*, die dafür sorgen, dass der nun aktivierte Thread seinen Cache auffrischt. Eine solche Memory Barrier ist eine Anweisung, die die virtuelle Maschine an die Hardware der Maschine erteilt, damit Prozessoren ihre Caches mit dem Hauptspeicher abgleichen. Dieser Abgleich ist erforderlich, weil er von den Sichtbarkeitsregeln des JMM verlangt wird, wie wir im letzten Beitrag erläutert haben (siehe /EFF2/). Eine solche Memory Barrier reduziert die Effizienz der Prozessoren, weil sie in den Caching-Algorithmus der Hardware eingreift. Das heißt, die mit der Synchronisation verbundenen Speichereffekte kosten Performance.

Die Freigabe eines Locks löst erneut Zugriffe auf die zu dem Lock gehörende Warteschlange aus, denn einer der wartenden Threads muss aufgeweckt werden. Die Freigabe des Locks löst außerdem erneut eine Memory Barrier aus, damit der freigebende Thread seine Cache-Inhalte den anderen Threads sichtbar macht.

All dies sind Aufwände, die allein durch die Synchronisation entstehen. In einer Single-CPU-Umgebung sind dies zusätzliche Laufzeitkosten, die nicht durch Laufzeitgewinn an anderer Stelle kompensiert werden. In einer Multiprozessor-Umgebung wird der Overhead in der Regel durch die erhöhte Ausnutzung der Prozessorleistung ausgeglichen oder sogar überkompensiert.

Wie hoch die Kosten der Synchronisation sind, läßt sich allgemein nicht sagen, sondern kann nur durch eine entsprechende Messung in der jeweils relevanten Umgebung verlässlich bestimmt werden, weil der Overhead von der Implementierung der virtuellen Maschine und deren Zusammenspiel mit dem Betriebsystem und der Hardware abhängt. Dass die Implementierungen unterschiedlich sind, kann man bereits daran sehen, dass in der virtuellen Maschine von Sun die alten impliziten Locks dahingehend optimiert sind, dass sie relativ preiswert sind, wenn es keine Konkurrenz beim Anfordern des Locks gibt, wohingegen die neuen expliziten Locks dahingehend optimiert sind, dass sie effizient sind, gerade wenn es sehr viel Konkurrenz gibt.

Die Laufzeiteinbußen durch Memory Barriers, die Verwaltung der Warteschlange, etc. machen aber nur einen Teil der Kosten der Synchronisation aus. Hinzu kommt noch die Einbuße an Skalierbarkeit.

Skalierbarkeitseinbußen durch Synchronisation

Synchronisation mit Hilfe von Locks führt dazu, dass gewisse Teile des Programms immer nur von einem Thread zu einer Zeit ausgeführt werden können. Das führt unter Umständen dazu, dass viele Threads warten und nur wenige Threads aktiv sind. Insgesamt ergibt sich daraus eine schlechte Ausnutzung der Prozessorleistung. Das ist in Multicore- und Multi- Prozessor-

Umgebungen nachteilig, weil die Leistung der vielen Kerne bzw. Prozessoren nicht angemessen genutzt wird. Wenn im ungünstigsten Falle nur ein einziger Thread aktiv ist und alle anderen warten, dann wird auch nur ein einziger Kern bzw. Prozessor ausgelastet und alle anderen bleiben ungenutzt. Das heißt, Programme mit viel Synchronisation skalieren schlecht in dem Sinne, dass sie zusätzliche Ressourcen in Form von zusätzlichen parallelen CPUs nur unzureichend ausnutzen.

Üblicherweise wird von einer Anwendung aber erwartet, dass sie auf einer Architektur mit höherer Prozessorleistung schneller abläuft. Dies wird auch erwartet, wenn die höhere Prozessorleistung nicht durch eine erhöhte Prozessorgeschwindigkeit, sondern durch durch eine erhöhte Anzahl von Prozessoren bzw. Prozessorkernen erbracht wird. Dabei ist die naive Erwartung an ein Programm meist, dass es beim Ablauf auf einem System mit zwei CPUs doppelt so schnell läuft wie auf einem System mit einer einzelnen CPU vom gleichen Typ. Diese einfache Rechnung ist aber falsch.

Amdahl's Law

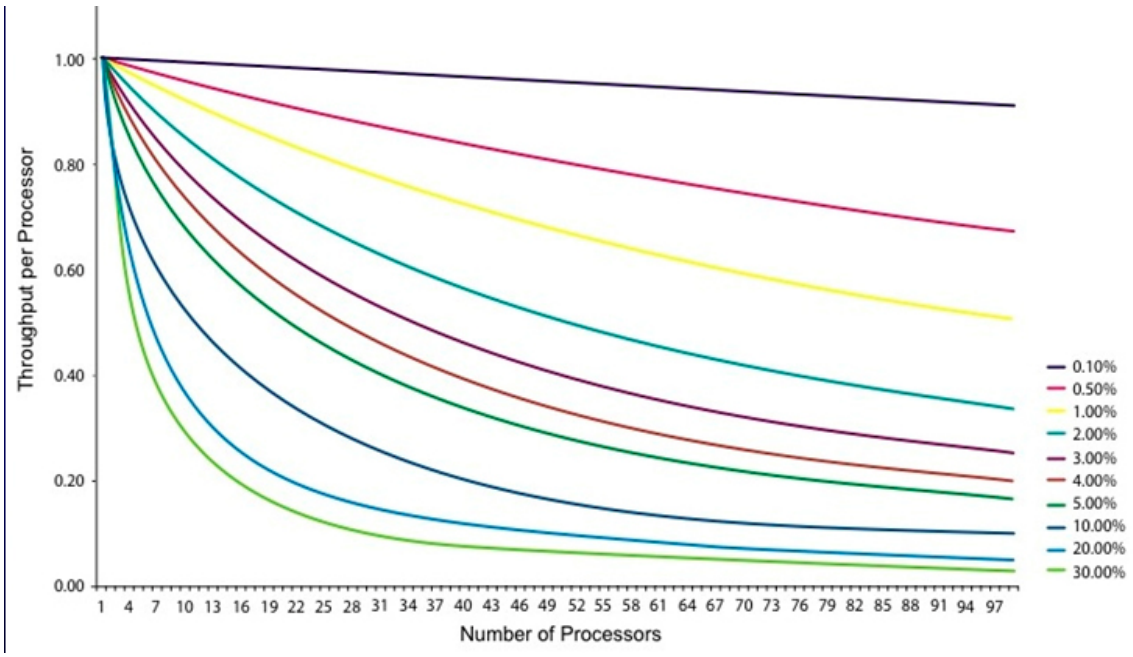
Der theoretisch erzielbare Performance-Gewinn durch zusätzliche Prozessoren wird nach dem Gesetz von Amdahl (siehe /AMDL/) bestimmt. Dazu ermittelt man den Prozentsatz des Programms, der parallel abläuft, und den Teil des Programms, der nicht parallelisierbar ist. Zu dem nicht parallellisierbaren Teil des Programms gehören alle synchronisierten Teile des Programms.

Wenn F der Bruchteil ist, der nicht parallel ablaufen kann, und N die Zahl der Prozessoren, dann ist der maximal erreichbare Performance-Zugewinn begrenzt durch:

$$speedup \leq \frac{1}{\left(F + \frac{(1 - F)}{N}\right)}$$

Bei unendlich vielen Prozessoren, konvergiert der maximale Performance-Zugewinn gegen 1/F. Wenn also 50% eines Programms parallel ablaufen und 50% sequentiell, dann wird das Programm maximal (bei unendlich vielen Prozessoren) doppelt so schnell. Beim Umstieg von einem auf zwei Prozessoren wird es lediglich um maximal 25% schneller. Das heißt, die naive Rechnung "doppelt so viele CPUs = doppelt so schnell" ist meist höchst unzutreffend.

Der Zugewinn wächst mit der Zahl der verfügbaren Prozessoren, aber auch mit dem Prozentsatz der Parallelverarbeitung in der Anwendung. Synchronisation jedoch reduziert den Anteil, der parallel ablaufen kann. Hier ist eine ernüchternde Grafik, die den Performance-Gewinn je zusätzlichem Prozessor zeigt abhängig vom nicht-parallelisierbaren Anteil des Programms:



Wenn nur 0,1% nicht-parallel ablaufen, dann ist der Zugewinn mit jedem zusätzlichen Prozessor ungefähr gleich. Das ist erfreulich, aber welche Anwendung läuft zu 99,9% parallel ab? Kaum eine. Wenn aber 30% nicht-parallel ablaufen, dann bringen der 5. oder 6. Prozessor nur noch 30-40% an Performance-Gewinn. Den 10. oder 12. Prozessor kann man sich eigentlich schon schenken, weil er kaum noch etwas bringt für die Performance.

Vollständig parallel?

Nun ist es in der Praxis schwierig, den Prozentsatz an Parallelverarbeitung in einer Anwendung präzise zu bestimmen. Bestenfalls kann man schätzen. Auch dabei leitet die Intuition gelegentlich in die Irre. Selbst Programmteile, die hochgradig parallel aussehen, haben sequentielle Anteile, was oftmals übersehen wird. Sehen wir uns dazu ein Beispiel an: eine Consumer-Producer-Situation, in der Produzenten-Threads Daten in eine `LinkedBlockingQueue` stecken, die von Konsumenten-Threads aus der Queue geholt und verarbeitet werden.

Das sieht nach einer hochgradigen Parallelverarbeitung aus und man könnte auf den Gedanken kommen, dass F, der Prozentsatz des nicht-parallelen Anteils, gleich Null ist. Das ist aber nicht der Fall, wie der Blick auf die Implementierung der `LinkedBlockingQueue` zeigt. Hier ist ein Ausschnitt aus dem Source-Code der Implementierung, nämlich die `offer`-Methode, mit der die Produzenten Objekte in der Queue ablegen:

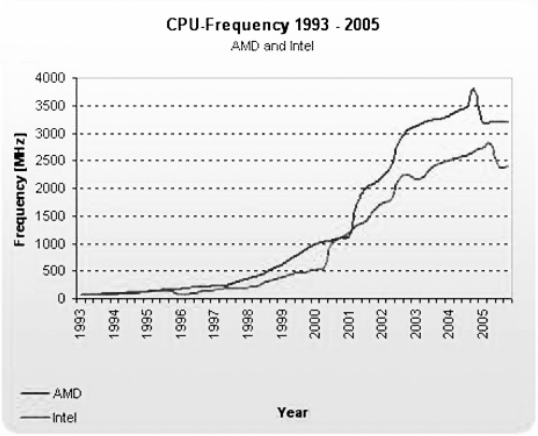
```
public boolean offer(E o) {
     if (o == null) throw new NullPointerException();
     final AtomicInteger count = this.count;
     if (count.get() == capacity)
         return false;
     int c = -1;
     final ReentrantLock putLock = this.putLock;
     putLock.lock();
     try {
         if (count.get() < capacity) {
             insert(o);
             c = count.getAndIncrement();
             if (c + 1 < capacity)
                 notFull.signal();
         }
     } finally {
         putLock.unlock();
     }
     if (c == 0)
         signalNotEmpty();
     return c >= 0;
 }
```

Man kann sehen, dass es in dieser Methode einen Anteil gibt, der nicht parallel abläuft, sondern der durch ein Lock sequentialisiert ist. Das heißt, um die Konsistenz der von den Threads gemeinsam verwendeten Queue zu sichern, ist ein gewisser Anteil an Sequentialisierung der Zugriffe unerlässlich.

Die Sequentialisierung in dieser Methode ist dabei keineswegs überflüssig oder ungeschickt implementiert. Im Gegenteil, die Implementierung der `LinkedBlockingQueue` ist hochgradig perfektioniert und nach allen Regeln der Kunst optimiert. Beispielsweise wird *Lock-Splitting* eingesetzt: statt eines einzigen Locks für die gesamte Datenstruktur gibt es ein `putLock` und ein `takeLock`, damit sich Produzenten und Konsumenten nicht gegenseitig behindern. Außerdem werden die Zugriffe auf das Feld `count`, das die jeweils aktuelle Zahl der Objekte in der Queue angibt, nicht mit Synchronisation gemacht, sondern *lock-free* mit Hilfe eines `AtomicInteger`. Zusätzlich werden konkurrierende Zugriffe auf Felder der Datenstruktur von vornherein vermieden, indem *lokale Kopien* auf dem Stack angelegt und verwendet werden. Aber selbst mit einer so hochoptimierten Implementierung, die sich bemüht, Synchronisation nach Möglichkeit zu vermeiden, bleibt ein kleiner Anteil an Synchronisation unvermeidlich. Die `take`-Methode sieht so ähnlich aus; auch dort gibt es einen sequentiellen Anteil. Man gelangt also zu der Erkenntnis, dass selbst hochgradig parallele Verarbeitungen geringe sequentielle Anteile haben, die die Skalierbarkeit der Anwendung reduzieren.

Wenn man dann noch überlegt, dass wir in der Regel Frameworks und andere Third-Party-Komponenten (wie hier die `LinkedBlockingQueue` aus dem JDK) einsetzen, die nicht-parallele Anteilen mitbringen, dann wird langsam klar, dass ein Anteil von 99,9% Parallelverarbeitung in einer Anwendung zwar theoretisch denkbar ist, in der Praxis aber wohl kaum vorkommen dürfte.

Sequentialisierung bekämpfen



Rücken wir die Aussagen des Gesetzes von Amdahl einmal in die rechte Perspektive. Alles was wir oben zu Skalierbarkeit bei mehreren Prozessoren gesagt haben, gilt uneingeschränkt auch für die Skalierbarkeit bei mehreren Kernen in einem Prozessor. Multicore-Prozessoren sind heutzutage die Norm. Seit das Gesetz von Moore nicht mehr gilt, dem gemäß sich die Prozessorgeschwindigkeit alle 2 Jahre verdoppelt, wird Prozessorleistung nicht mehr durch höhere Leistung eines einzelnen Prozessors erzielt, sondern es werden Prozessoren mit immer mehr Kernen gebaut. Letztes Jahr waren Dual-Core-Prozessoren Stand der Technik. Dieses Jahr sind es Quad-Core-Prozessoren und für die nächste Zukunft sind Prozessoren mit 6-12 Kernen angekündigt. De facto müssen Anwendungen heute so gebaut sein, dass sie die vielen Prozessorkerne auch nutzen. Der Trend geht aufgrund der Hardware-Gegebenheiten dahin, dass Software so geschrieben sein muss, dass sie auch zukünftige Mehrkern-Architekturen ausnutzen kann und mit der Zahl der Prozessorkerne skaliert. Das bedeutet aber, dass Synchronisation reduziert werden muss, weil sie diesem Ziel grundsätzlich im Wege steht.

Was kann man also tun, um die sequentiellen Anteile der eigenen Software zu reduzieren?

1. **Dauer der Synchronisation reduzieren**. Ein Lock sollte immer nur für eine möglichste kurze Zeitspanne gehalten werden, d.h. länger dauernde Operationen (z.B. I/O) gehören generell nicht in einen `synchronized`-Block, sondern sollten ausgelagert werden. Analog kann man überlegen, ob man eine Methode als Ganzes `synchronized` deklararieren muss oder ob man nicht besser einen kleineren `synchronized`-Block verwenden könnte, der wirklich nur die zu schützenden Anweisungen enthält und nichts Überflüssiges.
2. **`volatile`- und atomic Variablen benutzen.** Man kann Synchronisation durch Verwendung von atomaren Variablen vermeiden und beispielsweise als Zähler einen `AtomicInteger` anstelle eines gewöhnlichen `int` verwenden, weil der `AtomicInteger` atomare `increment`- und `decrement`-Methoden hat und ohne Synchronisation auskommt. Man kann Sichtbarkeit von Datenmodifikationen auch durch `volatile` anstelle von Synchronisation erzielen, wenn lediglich einfache Zugriffe auf primitive Daten nötig sind (siehe /EFF1/).
3. **Lock-Splitting / Lock-Striping verwenden**. Statt alle Zugriffe auf gemeinsam verwendete veränderliche Daten mit nur einem einzigen Lock zu schützen, kann man überlegen, ob sich separate Teile der Daten mit separaten Locks schützen lassen. Damit wird die Wahrscheinlichkeit von Kollisionen reduziert, die Threads warten insgesamt seltener und tendenziell steigt die Parallelität.
4. **Unveränderliche Daten verwenden.** Die Notwendigkeit für Sequentialisierung mit Hilfe von Locks besteht nur für gemeinsam verwendete Daten, die veränderlich sind. Wenn die gemeinsam verwendeten Daten sich nicht ändern können, dann wird keine Synchronisation benötigt. Das heißt, wo immer es möglich ist, sollten Immutable Types verwendet werden. Wenn keine adequaten Immutable Types vordefiniert sind, dann wird man bei Bedarf solche Typen selber implementieren müssen. Dabei spielen die JMM-Garantien für `final`-Variablen eine Rolle (siehe /EFF2/).
5. **Gemeinsamverwendung vermeiden.** Statt Daten mit anderen Threads gemeinsam zu verwenden, kann man überlegen, ob es nicht sinnvoll wäre, eine thread-lokale Kopie zu ziehen und diese zu verwenden. Dann wird für diese Daten keine Synchroniation mehr gebraucht, weil sie nicht mehr gemeinsam verwendet werden. Die `CopyOnWriteArrayList` im JDK verwendet eine solche Technik.
6. **Lock-Free Programming.** Man kann versuchen, Algorithmen zu verwenden oder zu entwickeln, die so geschickt auf gemeinsam verwendete veränderliche Daten zugreifen, dass kein Lock gebraucht wird, weil ausschließlich atomare Operationen ausgeführt werden. Solche Algorithmen zu entwickeln ist heute aber noch eher ein Forschungsthema als eine in der Praxis häufig eingesetzte Technik. Oft lassen sich die Datenzugriffe auch gar nicht so gestalten, dass lock-free programmiert werden kann. Es gibt im JDK Beispiele von Datenstrukturen, nämlich die Concurrent Collections im

Package `java.util.concurrent`, die mit solchen Algorithmen implementiert sind. Man sollte also versuchen, Concurrent Collections anstelle von Synchronized Collections oder selbstgeschriebenen Collections zu verwenden.

Zusammenfassung

In diesem Beitrag haben wir gesehen, dass Synchronisation teuer ist und worin die Kosten bestehen. In Multi-Prozessor- oder Multicore-Umgebungen wirkt sich die Synchronisation negativ auf die Skalierbarkeit des Programms aus. Tendenziell wird man also versuchen, Synchronisation zu reduzieren oder nach Möglichkeit ganz zu vermeiden. Techniken dafür haben wir kurz angesprochen. Dabei spielen `volatile`-, `final`- und atomare Variablen eine Rolle. Deshalb werden wir uns in den nächsten Beiträgen diese Variablen genauer ansehen.

Verweise

/EFF1/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/EFF2/ JMM - Überblick über das Java Memory Modell (JMM)
Klaus Kreft & Angelika Langer
JavaMagazin 8.08

/AMDL/ "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities"
Gene Amdahl
AFIPS Conference Proceedings, (30), pp. 483-485, 1967

Java Memory Modell

Teil 4: Details zu volatile-Variablen

Klaus Kreft & Angelika Langer

Wir haben im letzten Beitrag diskutiert, dass Synchronisation mit Hilfe von Locks Performance kosten kann. Um dies zu vermeiden, kann die Synchronisation unter Umständen durch andere Techniken ersetzt werden. Eine Alternative zur Synchronisation ist die Verwendung von `volatile`-Variablen. Deshalb wollen wir uns in diesem Beitrag näher mit `volatile`-Variablen befassen.

`volatile` als Alternative zur Synchronisation

Wir haben schon in einem früheren Beitrag (siehe /EFF1/) ein Beispiel gezeigt, in dem bewußt auf Synchronisation verzichtet wurde:

```
public class Processor {
    private volatile boolean connectionPrepared = false;
    public void prepareConnection() {
        // ... open connection ...
        connectionPrepared = true;
    }
    public void start() throws InterruptedException {
        // ... various initializations ...
        while (!connectionPrepared )
            Thread.sleep(500);
        // ... start actual processing ...
    }
}
```

Hier greifen zwei Methoden einer Klasse konkurrierend auf das `boolean`-Feld `connectionPrepared` zu. Die Idee ist, dass die `start`-Methode in einem Thread ausgeführt wird, der alle vorbereitenden Arbeiten anstößt und dann abwartet, bis alle Vorbereitungen abgeschlossen sind, ehe er die eigentliche Verarbeitung beginnt. Die beiden Threads kommunizieren miteinander über gemeinsam verwendete veränderliche Daten, nämlich das `boolean`-Feld `connectionPrepared`. Der eine Thread setzt `connectionPrepared` auf `true`, wenn er fertig ist, und der andere Thread beobachtet, ob `connectionPrepared` auf `true` gesetzt wurde, um dann mit der eigentlichen Arbeit zu beginnen.

Das Lesen und Verändern des `boolean` Feldes ist nicht unterbrechbar. Das garantiert die Sprachspezifikation mit der Regel, dass der Zugriff auf Variablen von primitiven Typ (außer `long` und `double`) und auf Referenzvariablen atomar ist. Deshalb braucht man in dem Beispiel keine Synchronisation, um den Zugriff ununterbrechbar zu machen.

Wir haben an diesem Beispiel diskutiert, dass das `boolean`-Feld `connectionPrepared` aber als `volatile` deklariert werden muss, wenn auf Synchronisation verzichtet wird, damit die Modifikation, die der Initialisierungsthread macht, dem wartenden Thread sichtbar gemacht wird. Diese Deklaration ist notwendig, weil das Java-Memory-Modell (JMM) nur unter bestimmten Bedingungen garantiert, dass Speichermodifikation anderen Threads sichtbar gemacht werden. Welche Bedingungen das sind, haben wir in einem Beitrag über das JMM (siehe /EFF2/) diskutiert. Unter anderem gibt es Sichtbarkeitgarantien für Synchronisation, aber auch für `volatile`-Variablen. Hier noch mal eine kurze Wiederholung:

Das Memory-Modell in Java ähnelt einer abstrakten SMP (= symmetric multi processing)-Maschine: die Threads laufen parallel und konzeptionell haben alle Threads Zugriff auf einen gemeinsamen Hauptspeicher (main memory), in dem die gemeinsam verwendeten Variablen abgelegt sind. Daneben hat jeder Thread einen eigenen lokalen Speicherbereich (cache), in den er Variablen hineinladen und lokal bearbeiten kann. Das Zurückschreiben der lokalen Daten in den Hauptspeicher (flush) und das Hereinladen von Daten aus dem Hauptspeicher (refresh) muss nach den Regeln des JMM geschehen.

- *Synchronisation*. Der Erhalt eines Locks löst einen Refresh aus, das Freigeben des Locks löst einen Flush aus. Alle Threads, die dasselbe Lock verwenden, durchlaufen die Sequenz der mit diesem Lock synchronisierten Anweisungen nicht parallel sondern nacheinander. Das heißt, es ist klar, dass zunächst ein Thread auf die gemeinsam verwendeten Daten zugreift und erst danach ein anderer. Da beim Freigeben des Locks der eine Thread alle lokalen Daten in den Hauptspeicher zurück schreiben muss und der nächste Thread beim Erhalt des Locks seinen lokalen Arbeitsspeicher aus dem Hauptspeicher auffrischen muss, sieht der zweite Thread alle Modifikationen, die der erste Thread an den gemeinsam verwendeten Daten vorgenommen hat.

- *Lese- und Schreibzugriff auf `volatile`-Variablen*. Das Lesen einer `volatile`-Variablen löst einen Refresh aus, das Modifizieren einer `volatile`-Variablen löst einen Flush aus. Das bedeutet, dass ein Thread, der den Inhalt einer `volatile`-Variablen liest, den Wert nicht aus seinem Arbeitsspeicher holen darf, sondern ihn aus dem Main Memory holen muss. Dabei muss er nicht nur den Inhalt der betreffenden `volatile`-Variablen holen, sondern er muss seinen gesamten Arbeitsspeicher auffrischen. Auf diese Weise sieht er alle Modifikation an gemeinsam verwendeten

Variablen, die zuvor ein anderer Thread gemacht hat, der auf dieselbe `volatile`-Variablen vorher schreibend zugegriffen hat. Denn beim schreibenden Zugriff muss der andere Thread seinen gesamten Arbeitspeicher in den Hauptspeicher zurückgeschrieben haben.

Da sowohl Synchronisation als auch `volatile` Speichereffekte haben, konnten wir in unserem Beispiel auf Synchronisation verzichten und sie durch die Verwendung von `volatile` ersetzen. In welchen Situationen funktioniert diese Technik? Immer dann, wenn Synchronisation nur wegen der Sichtbarkeit verwendet werden würde und aus sonst keinem anderen Grunde (z.B. für die Ununterbrechbarkeit des Zugriffs) gebraucht wird. Voraussetzung dafür ist, dass der Zugriff auf die betreffenden gemeinsam genutzten Daten bereits an sich atomar ist, d.h. die Technik funktioniert nur für Daten von primitivem Typ oder für Referenzvariablen.

Mißverständnis

Bei solchen Optimierungen können sich Fehler einschleichen. Schauen wir uns einen solchen Fehler an:

```
public class IntStack {  // falsch; nicht nachmachen !!!
  // attributes:
  private int[] array;
  private int cnt = 0;

  // constructor:
  public int_stack(int capacity) { array = new int[capacity]; }

  // methods:
  public synchronized void push(int elm)  { array[cnt++] = elm; }
  public synchronized int pop()           { return(array[--cnt]); }
  public              int size()          { return cnt; }
}
```

Es ist das Beispiel eines vereinfachten Stacks von Integerzahlen. Die Methoden `push` und `pop` sind synchronisiert, die Methode `size` nicht. Die Methode `size` kommt ohne Synchronisation aus, weil sie den Wert von `cnt` zurück liefert. Das Feld `cnt` ist vom primitiven Typ `int` und der lesende Zugriff ist deshalb atomar. Für die Ununterbrechbarkeit der `size`-Methode wird die Synchronisation also nicht gebraucht; deshalb wurde sie weggelasen.

Wie ist das aber nun mit der Sichtbarkeit der Modifikationen am `cnt`-Feld? Das `cnt`-Feld ist nicht `volatile`, aber es sind alle modifzierenden Zugriffe auf das Feld synchronisiert. Da Synchronisation Flushes auslöst, werden die Änderungen am `cnt`-Feld sichtbar, die in den Methoden `push` und `pop` gemacht wurden. Also ist die Implementierung in Ordnung - so lautet zumindest ein gelegentlich anzutreffendes Mißverständnis. Leider ist es nicht ganz so.

Es genügt nicht, dass die Modifikationen am `cnt`-Feld wegen der Synchronisation der Methoden `push` und `pop` gemäß JMM-Regeln in den Hauptspeicher geflusht werden. Wenn die `size`-Methode unsynchronisiert auf das `cnt`-Feld zugreift, dann ist nicht garantiert, dass vor dem Lesezugriff ein Refresh gemacht wurde. Es kann also passieren, dass ein Thread durch zahlreiche `pop`-Aufrufe das `cnt`-Feld inkrementiert und ein anderer Thread, der periodisch mit der `size`-Methode das `cnt`-Feld abfragt, diese Änderung nicht zu sehen bekommt, weil der `cnt`-Wert stets aus dem Cache des Threads und nicht aus dem Hauptspeicher geholt wird.

Es genügt also bereits ein einziger Zugriff ohne Speichergarantien auf eine Variable und die Sichtbarkeit ist nicht mehr gewährleistet. In dem Beispiel muss daher das `cnt`-Feld als `volatile` deklariert werden.

```
public class IntStack {  // jetzt ist es in Ordnung
  // attributes:
  private int[] array;
  private volatile int cnt = 0;

  // constructor:
  public int_stack(int capacity) { array = new int[capacity]; }

  // methods:
  public synchronized void push(int elm)  { array[cnt++] = elm; }
  public synchronized int pop()           { return(array[--cnt]); }
  public              int size()          { return cnt; }
}
```

Kosten von `volatile`

Wir haben `volatile` als preiswerte Alternative zur Synchronisation vorgestellt, aber natürlich sind auch mit der Verwendung von `volatile` Kosten verbunden. Zwar löst der Zugriff auf `volatile`-Variablen keine Aufwände für die Verwaltung von Warteschlangen oder das Aufwecken von Threads aus Wartezuständen aus. Es kann auch keinen Stau von Threads geben, die auf den Erhalt eines Locks warten. Aber der Zugriff auf `volatile`-Variablen löst Flushes und Refreshes aus, die ungünstig in die Caching-Mechanismen der Prozessoren eingreifen, und ist damit teurer als der Zugriff auf normale Variablen. Man sollte also auch `volatile` nicht gänzlich gedankenlos verwenden.

So kann zum Beispiel ein sehr häufiger Zugriff auf eine `volatile`-Variable, zum Beispiel in einer Schleife, in Summe teurer sein, als die einmalige Synchronisation der gesamten Schleife, weil jeder einzelen Zugriff auf die `volatile`-Variable einen Refresh oder Flush auslöst, die Synchronisation aber nur jeweils einen Refresh und Flush. Wo jeweils der Trade-off ist, muss man im Einzelfall mit einer Benchmark-Messung bestimmen. Wir wollen an dieser Stelle nur darauf hinweisen, dass man `volatile` auch aus Versehen zu ungünstig verwenden kann, dass sich u.U. negative Performance-Effekte einstellen.

Natürlich ist die Benutzung von `volatile` und Synchronisation mit Hilfe von Locks nicht in allen Belangen gleichwertig. Die Locks sind zusammen mit Conditions in einen Framework eingebettet, der auch komplexere Synchronisation mit Hilfe von `wait()`/`await()` und `notify()`/`signal()` erlaubt. Bei `volatile` ist das nicht so. Hier bleibt nur das mehrmalige Versuchen (Pollen), bis sich die Situation eingestellt hat, auf die man wartet. Das muss nicht immer zu schlechten Lösungen führen. Dies kann man an dem ersten Beispiel in diesem Artikel sehen. Eine Implementierung des Latch-Pattern auf Basis von `volatile`, bei dem der Startthread wartet, bis die Kommunikationsfunktionalität initialisiert ist, macht durchaus Sinn. In anderen Situationen können sich aber ernste Nachteile aus dem erhöhten CPU-Verbrauch (auf Grund des Pollens) und der lose gekoppelten Synchronisation beider Threads ergeben. In diesem Sinne muss man auch einige Beispiele in diesem Artikel sehen. Sie haben einen didaktischen, theoretischen Hintergrund: um an möglichst einfachen Beispielen die Speichereffekte von `volatile` zu diskutieren. Deshalb wrden wir in unserem nächsten Artikel das Thema noch mal von der praktischen Seite am Beispiel des Double-Check-Idioms aufrollen.

Speichereffekte von `volatile` auf andere Variablen

Bislang haben wir nur Beispiele betrachtet, in denen es um den Zugriff auf eine einzige Variable ging. Wenn der Zugriff darauf atomar war und die Variable keine Abhängigkeiten zu anderen Variablen hatte, dann konnten wir die Synchronisation durch die Verwendung von `volatile` reduzieren. Wie ist das bei mehreren Variablen? Schauen wir uns ein Beispiel an:

```
class FutureResult {   // korrekt, aber nicht unbedingt empfehlenswert
  private volatile boolean ready = false;
  private Object data = null;

  public Object getResult() {
    if (ready);
    return data;
  }
  public boolean isReady() {
    return ready;
  }
  // only one thread may ever call putResult()
  public void putResult(Object o) {
    data = o;
    ready = true;
  }
}
```

Hier ist die Idee, dass ein `FutureResult` von einem Thread verwendet wird, um anderen Threads ein Ergebnis zu übergeben. Ob das Ergebnis vorliegt und abgeholt werden kann, wird über das `boolean`-Feld `ready` angezeigt. Die empfangenden Threads pollen das Feld mit Hilfe der `isReady`-Methode und holen das Ergebnis mit `getResult` ab, sobald `isReady` `true` liefert.

Auf Synchronisation wird komplett verzichtet. Für die Methoden `getResult` und `isReady` geht das, weil sie lediglich atomare Operationen ausführen und deshalb nicht unterbrechbar sind. Die Method `putResult` braucht keine Synchronisation, weil unterstellt wird, dass nur ein einziger Thread diese Methode einmal aufruft. Wegen dieser Benutzungskonvention kann es keine konkurrierenden `putResult`-Aufrufe geben. Die konkurrierenden Aufrufe von `getResult` und `isReady` stellen auch kein Problem dar. Wenn eine dieser Methoden mitten in der `putResult`-Methode auf die Felder `ready` und/oder `data` zugreift, dann liefert `isReady` schlimmstenfalls noch `false` zurück, obwohl das Resultat bereits im Feld `data` abgelegt ist.

Diese vorübergehende Inkonsistenz der Daten ist aber kein Problem, weil der abfragende Thread dann beim nächsten Aufruf von `isReady` die konsistente Information erhält.

Da auf Synchronisation verzichtet wurde, haben wir unsynchronisierte Zugriffe auf die beiden Felder `ready` und `data` und wir müssen für die Sichtbarkeit der Modifikationen an diesen Feldern sorgen. Das wird getan, indem das `boolean`-Feld `ready` als `volatile` deklariert ist. Das Referenzfeld `data` ist hingegen nicht `volatile`. Ist das korrekt oder müssen beide Felder `volatile` sein?

Wenn man die Sichtbarkeitsregeln für `volatile`-Variablen genau ansieht, stellt man fest, dass es in der Tat genügt, wenn das `ready`-Feld `volatile` ist. Der schreibende Zugriff auf `ready` in der Methode `putResult` löst nämlich einen Flush aus. Dabei wird nicht nur der Inhalt von `ready` geflusht, sondern es werden alle Speichermodifikationen sichtbar, die der Thread bis dahin gemacht hat, also auch die Modifikation an der Referenzvariablen `data`. Die seltsame `if`-Abfrage in der Methode `getResult` mit dem leeren Statement im true-Fall dient einem ganz ähnlichen Zweck und ist keineswegs überflüssig. Der Lesezugriff auf die `volatile`-Variable `ready` löst einen Refresh aus, der nicht nur den aktuellen Inhalt von `ready` aus dem Hauptspeicher beschafft, sondern auch alle anderen Variablen auffrischt, auf die der Thread zugreifen wird. Damit wird auch der aktuelle Inhalt der Referenzvariablen `data` dem lesenden Thread sichtbar.

Nun ist das ganze Beispiel etwas seltsam, weil es um maximale Optimierung bemüht ist. Der Verzicht auf die Synchronisation ist nur wegen der Benutzungskonvention möglich und dann wird auch noch versucht, mit möglichst wenig `volatile`-Variablen auszukommen. Im Hinblick auf die Optimierung ist es gut, in Hinblick auf die Verständlichkeit des Codes muss man sich allerdings fragen, ob es hier nicht sinnvoller wäre, beide Felder als `volatile` zu erklären. Der Verzicht auf die `volatile`-Deklaration für die Referenzvarable `data` führt schließlich zu einem subtilen und damit recht fragilen Code, der bei geringfügigen Änderungen bereits inkorrekt wird. Der Leser des Source-Code muss folgende Aspekte verstanden haben:

– Die Reihenfolge der Anweisungen in der Methode `putResult` ist wichtig, denn der Zugriff auf das `volatile`-Feld muss ganz am Ende nach allen anderen Modifikationen geschehen, sonst werden die anderen Modifikationen nicht sichtbar.
– Der doch recht künstlich anmutende Lesezugriff auf das `volatile`-Feld `ready` in der Method `getResult` ist ebenfalls wichtig, weil sonst die Sichtbarkeit des anderen non-`volatile` Feldes fehlen würde.

Einfacher und vermutlich nicht einmal nennenswert langsamer wäre eine Lösung, in der beide Felder als `volatile` erklärt sind:

```
class FutureResult {   // korrekt
  private volatile boolean ready = false;
  private volatile Object data = null;

  public Object getResult() {
    return data;
  }
  public boolean isReady() {
    return ready;
  }
  // only one thread may ever call putResult()
  public void putResult(Object o) {
    data = o;
    ready = true;
  }
}
```

Wir haben das Beispiel bewußt ausgewählt, um zu demonstrieren, dass Zugriffe auf `volatile`-Variablen Speichereffekte haben, die nicht nur den Inhalt der `volatile`-Variablen betreffen, sondern dass alle im Cache eines Threads gehaltenen Variablen durch die Flushes und Refreshes betroffen sind.

`volatile`-Referenzvariablen

Für eine `volatile`-Referenzvariable gelten dieselben Garantien wie für `volatile`-Variablen von einem primitiven Typ. Allerdings beziehen sich alle Regeln stets nur auf die Referenz selbst, also die Adresse des Objekts, nicht aber auf das referenzierte Objekt. Sehen wir uns das einmal an einem Beispiel genauer an. Ändern wir die obige Klasse `FutureResult` so, dass die nicht ein einzelnes Objekt als Resultat enthält, sondern ein Paar von Resultaten:

```
public class Pair {
     private Object first;
     private Object second;

     public void addFirst(Object o) {
            first = o;
```

```
      }
      public void addSecond(Object o) {
             second = o;
      }
      public Object[] toArray() {
             return new Object[] {first,second};
      }
}
public class FutureResult {
    private volatile Pair data = null;

    public Object[] getResult() {
        return (data==null)?null:data.toArray();
    }
    public void isReady() [
        return data != null;
    }
    // only one thread may ever call putResult()
    public void putResult(Object o1,Object o2) {
        Pair tmp = new Pair();
        tmp.addFirst(o1);
        tmp.addSecond(o2);
        data = tmp;
    }
}
```

Die `FutureResult`-Klasse hat dieselben Benutzungskonventionen wie zuvor: nur ein Thread darf die `putResult`-Methode einmal aufrufen und die Empfänger des Resultats dürfen `getResult` erst aufrufen, wenn `isReady` `true` geliefert hat. In diesem Falle kommen wir wieder ganz ohne Synchronisation aus. Die Frage ist nun, ob es genügt, dass die Referenzvariable `data` als `volatile` erklärt ist. Schließlich interessieren den Empfanger des Resultats die Inhalte des `Pair`-Objekts und nicht dessen Adresse.

Die Methode `putResult` erzeugt und füllt ein temporäres `Pair`-Objekt und weist danach der `volatile`-Variablen `data` die Adresse dieses temporären Objekts zu. Diese Adressezuweisung löst den Flush aus und dabei werden alle ggf. im Cache des Threads gemachten Speichermodifikationen in den Hauptspeicher geschrieben. Damit werden garantiert auch die Inhalte des referenzierten `Pair`-Objekts sichtbar. Die Methoden `getResult` und `isReady` müssen lesend auf die `volatile`-Variablen `data` zu, ehe sie den Inhalt des referenzierten `Pair`-Objekts anschauen können. Das Lesen der Adresse löst den Refresh aus, der auch die Inhalte des referenzierten `Pair`-Objekts sichtbar macht.

Wichtig ist hierbei, dass die Methode `putResult` die Modifikation der Adresse *nach* der Modifikation der Inhalte des referenzierten Objekts macht. Folgende Implementierung wäre daher falsch:

```
    public void putResult(Object o1,Object o2) {
        data = new Pair();
        data.addFirst(o1);
        data.addSecond(o2);
    }
```

Hier erfolgt die Modifikation auf die `volatile`-Referenzvariable *vor* den Modifikationen des Objekts. Geflusht werden daher die Default-Inhalte des konstruierten `Pair`-Objekts. Ob die danach erfolgten Modifikationen am `Pair`-Objekt in den Hauptspeicher geschrieben werden, ist nicht gesichert. Es kann also passieren, dass der empfangende Thread das Ergebnis nie zu Gesicht bekommt.

Damit die `putResult()` Methode von oben korrekt funktioniert, kann die `Pair` Klasse so geändert werde, dass die beiden Felder `first` und `second` auch `volatile` sind:

```
public class Pair {
     private volatile Object first;
     private volatile Object second;

     public void addFirst(Object o) {
            first = o;
     }
     public void addSecond(Object o) {
            second = o;
     }
     public Object[] toArray() {
            return new Object[] {first,second};
     }
}
```

Jetzt löst auch die Zuweisung in `data.addFirst(o1)` bzw. `data.addSecond(o2)` jeweils einen eigenen Flush aus und macht damit die Änderungen für andere Threads sichtbar.

Der Ansatz, weitere geschachtelt enthaltene Felder `volatile` zu machen, hat natürlich seine Limitationen: Wenn wir keinen Zugriff auf den Sourcecode der Klasse (in unserem Fall `Pair`) haben, ist dies nicht möglich. Ein anderes Beispiel sind Java Arrays. Hier ist es auch nicht möglich, die Elemente des Arrays explizit `volatile` zu machen., sodass die korrekte Implementierung unseres Beispiels mit einem `Object`-Array so aussieht:

```
public class FutureResult {
    private volatile Object[] data = null;

    public Object[] getResult() {
        return data;
    }
    public void isReady() [
        return data != null;
    }
    // only one thread may ever call putResult()
    public void putResult(Object o1,Object o2) {
        Object[] tmp = new Object[2];
        tmp[0] = o1;
        tmp[1] = o2;
        data = tmp;
    }
}
```

Zum Abschluss noch ein Hinweis auf ein gelegentlich anzutreffendes Missverständnis: die Sichtbarkeitsregeln, die wir in den diskutierten Beispielen ausgenutzt haben, gelten immer nur dann, wenn die Zugriffe auf die `volatile`-Variablen zusammen passen. Das heißt, wenn ein Thread durch den schreibenden Zugriff auf eine `volatile` Variable einen Flush auslöst, dann werden die Speichermmodifikationen nicht allen Threads sichtbar, sondern nur denjenigen, die einen lesenden Zugriff auf diesselbe `volatile` Variable machen. Diese Randbedingung wird gelegentlich übersehen, ist aber eigentlich nicht besonders überraschend. Für die Speichereffekte im Zusammenhang mit dem Anfordern und Freigeben von Locks gilt genau dasgleiche: wenn ein Thread durch durch das Freigeben eines Locks einen Flush auslöst, dann werden die Speichermodifikationen nicht allen Threads sichtbar, sondern nur denjenigen, die dasselbe Lock anschließend anfordern und bekommen.

Zusammenfassung

In diesem Beitrag haben wir die Speichereffekte von `volatile` Variablen diskutiert. Der Gedanke dahinter ist: man möchte Datensynchronization, wo möglich und erwünscht, durch `volatile` zu ersetzen. Dadurch lassen sich Java Programme stärker parallelisieren, sodass sie auf Multicore- und Multi-Prozessor-Architekturen besser skalieren. Beim nächsten Mal wollen wir uns das Ganze an einem Beispiel aus der Praxis, dem Double-Check-Idiom, ansehen.

Verweise

/EFF1/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/EFF2/ JMM - Überblick über das Java Memory Modell (JMM)
Klaus Kreft & Angelika Langer
JavaMagazin 8.08

/EFF3/ JMM - Die Kosten der Synchronisation
Klaus Kreft & Angelika Langer
JavaMagazin 9.08

Java Memory Modell

Teil 5: volatile und das Double-Check-Idiom

Klaus Kreft & Angelika Langer

Wir haben in den letzten Beiträgen diskutiert, dass Synchronisation mit Hilfe von Locks die Skalierbarkeit bei Mulit-Core- und Multiprozessor-Architekturen einschränkt (/EFF3/). Um dies zu vermeiden, kann die Daten-Synchronisation durch die Verwendung von `volatile`-Variablen erreicht werden. Dazu haben wir uns im letzten Beitrag im Detail angesehen, welche Speichereffekte der lesende bzw. schreibende Zugriff auf `volatile` Variablen hat(/EFF4/). Die dort verwendeten Beispiele hatten aber eher didaktischen Charakter. Was uns also bis heute noch fehlt, ist ein überzeugendes Beispiel aus der Praxis, das zeigt, wie die Verwendung von `volatile` die Skalierbarkeit verbessert. Das liefern wir in diesem Artikel nach.

Ausgangssituation

Wir hatten in unserem letzen Beitrag schon erwähnt, dass `volatile` und Synchronisation mit Locks keine völlig gleichwertigen Konzepte sind (/EFF4/). Deshalb gibt es auch kein einfaches ‚Kochrezept', das man anwenden kann, um Locks durch `volatile`-Variablen zu ersetzen, um so die parallele Ausführbarkeit des Programms zu verbessern. Im Allgemeinen wird man nach Einführung der `volatile`-Variablen auch nicht immer vollständig auf das Locking verzichten können, sondern nur bei einem Teil der Zugriffe. Sind dies aber die Zugriffe, die in der Praxis am häufigsten vorkommen, so hat man das Wesentliche (nämlich eine Performanceverbesserung) schon erreicht.

Kommen wir zu unserem Beispiel. Es geht darum, ein privates Feld in einer Klasse nicht bei der Konstruktion, sondern beim ersten Zugriff zu initialisieren (*lazy initialization*):

```
public class MyClass {
    private MyField lazyField;
    ...

    public synchronized MyField getMyField() {
        if (lazyField == null)
            lazyField = new MyField( ... );

        return lazyField;
    }

    ...
}
```

Um fehlerhafte Mehrfach-Initialisierung auszuschließen, ist die `getMyField`-Methode `synchronized`, d.h. wir verwenden das mit `this` assoziierte Lock, um die Initialisierung zu schützen.

Erst einmal ist nichts gegen diese Implementierung einzuwenden: sie tut das, was sie soll, fehlerfrei. Zweifel bezüglich der Performance kommen einem aber, wenn man sich überlegt, dass das Lock im wesentlich nur für die Initialisierung benötigt wird. Nur beim allerersten Aufruf, wenn das `MyField`-Objekt erzeugt und seine Adresse im Feld `lazyField` abgelegt wird, kann eine Race Condition auftreten, die man per Synchronisation auflöst. Bei allen weiteren Aufrufen wird nur noch die Adresse abgefragt und zurückgegeben, wofür keine Synchronisation gebraucht wird. Trotzdem wird das Lock bei jedem Aufruf von `getMyField()` wieder benutzt.

Welche Optimierungsmöglichkeiten haben wir? Die kritische Region können wir noch unwesentlich verkleinern und das `return` herausziehen. Um die Benutzung des Locks selbst kommen wir so einfach aber nicht herum.

Das Double-Check-Idiom

Doug Schmitt hat sich vor mehr als zwölf Jahren schon mal Gedanken zu diesem Problem gemacht und ist damals auf das *Double-Check-Idiom* gekommen (die Details zu seiner Veröffentlichung sowie zur Historie des Idioms finden sich im Insert). Übertragen auf unser Ausgangsproblem sieht das Double-Check-Idiom so aus:

```
public class MyClass {
    private volatile MyField lazyField;
    ...

    public MyField getMyField() {
        if (lazyField == null) {  // Zeile 2
            synchronized (this) {
                if (lazyField == null) {
                    lazyField = new MyField( ... );
                }
            }

        return lazyField;         // Zeile 8
```

```
    }

    ...
}
```

In der Methode `getMyField()` wird ohne jede Synchronisation auf das Feld `lazyField` zugegriffen. Da es `volatile` ist, sehen wir auch, ob andere Threads es bereits initialisiert haben (warum das so ist haben wir im letzten Artikel im Detail diskutiert /EFF4/). Falls die Initialisierung bereits erfolgt ist, geben wir `lazyField` zurück und die Methode ist beendet. Und wenn die Initialisierung noch nicht erfolgt ist? Dann erfolgt sie in einem `synchronized`-Block. Dabei wird in dem Block noch einmal geprüft, ob `lazyField` immer noch `null` ist. Schließlich kann in der Zeit zwischen dem ersten Prüfen und dem Eintreten in den `synchronized`-Block die Initialisierung durch einen anderen Thread erfolgt sein.

Woher der Name ‚Double-Check-Idiom' stammt, ist wohl offensichtlich: das relevante Feld `lazyField` wird zweimal geprüft. Dabei kommt im Englischen noch ein gewisser Sprachwitz zum tragen: *double check* bedeutet idiomatisch so etwas wie *genau prüfen* und meint damit nicht unbedingt die zweimalige Prüfung.

Aber zurück zum Technischen: trotz der zweimaligen Prüfung hat sich die Performance verbessert. Die zweite Prüfung erfolgt nämlich nur bei der Initialisierung; möglicherweise sogar in mehreren Threads, wenn diese gerade zur gleichen Zeit die Methode `getMyField()` zum ersten Mal aufrufen. Dieser minimale Performanceverlust bei der Initialisierung wird durch den völligen Verzicht auf Locking im weiteren Ablauf des Programms mehr als wettgemacht. Das gilt schon bei einer Ein-Kern- / Ein-Prozessor-Architektur. Bei einer Multi-Kern- / Multi-Prozessor-Architektur ist der Gewinn noch höher, da verschiedene Threads die Methode `getMyField()` (nach der Initialisierung) echt parallel ausführen können.

Was bedeutet das Beispiel im Allgemeinen? Die Verwendung von `volatile` kann helfen, die Performance bei konkurrierenden Threadzugriffen, die bisher mit Locks synchronisiert wurden, zu verbessern. Es gibt kein ‚Kochrezept' für das Vorgehen. Ein Tipp ist aber, dass man nicht versuchen muss, das Locking ganz zu vermeiden. Für die Performanceverbesserung reicht es auch, dass das Locking im Programmablauf signifikant weniger durchlaufen wird.

Eine Optimierung

Als wir das Konzept zu diesem Artikel zusammengestellt haben, ist zufällig gerade die neueste Ausgabe vom Joshua Blochs *Effective Java* (siehe /BLCH/) herausgekommen. Dort im Item 71 (Use lazy initialization judiciously) hat Joshua Bloch seine Version des Double-Check-Idioms vorgestellt. Diese Version lässt die Speichereffekte von `volatile` noch subtiler in die Implementierung einfließen; deshalb wollen wir sie hier auch diskutieren.

Auf unser Beispiel angewandt sieht Joshua Bloch's Implementierung so aus:

```
public class MyClass {
    private volatile MyField lazyField;
    ...

    public MyField getMyField() {
        MyField tmp = lazyField;
        if (tmp == null) {
            synchronized (this) {
                tmp = lazyField
                if (tmp == null) {
                    lazyField = tmp = new MyField( ... );
                }
             }

        return tmp;
    }

    ...
}
```

Der offensichtliche Unterschied liegt in der zusätzlichen lokalen Variablen `tmp`. Welche Aufgabe hat sie?

Wir erinnern uns noch mal daran, was beim Lesen einer `volatile` Variable passiert (wir hatten das im letzten Artikel detailliert diskutiert /EFF4/): der Wert der Variable kann nicht einfach aus dem lokalen Arbeitsspeicher eines Thread genommen werden, sondern muss neu aus dem Hauptspeicher gelesen werden, um sicher zu sein, dass Updates, die andere Threads möglicherweise gemacht haben, sichtbar werden.

In der ersten Implementierung des Double-Check-Idioms wird die `volatile` Variable `lazyField` im Nicht-Initialisierungsfall zweimal gelesen: einmal beim Vergleich (Zeile 2) und einmal beim `return` (Zeile 8). In beiden Fällen wird ein ‚Refresh' vom Hauptspeicher gemacht. Der zweite ‚Refresh' ist aber im Nicht-Initialisierungsfall überflüssig, da wir wissen, dass der Wert von `lazyField` sich nach der Initialisierung nicht mehr ändert. Das heißt, das Feld `lazyField` ist sowas wie ‚semi final'. Es wird genau einmal gesetzt. Da es sich um eine *lazy initialization* handelt, erfolgt das Setzen aber nicht im Konstruktor sondern in `getMyField()`. Deshalb kann das Feld nicht wirklich als `final` deklariert werden. Trotzdem ändert es sich im weiteren Ablauf des Programms nicht mehr. Da Java-Compiler und Java-Laufzeitsystem nichts von dieser ‚semi final' Eigenschaft wissen, können sie keine Optimierung vornehmen und auf den zweiten ‚Refresh' vom Hauptspeicher nicht verzichten. Das bedeutet, wir müssen die Optimierung selbst machen. So kommt dann die Variante von Joshua Bloch heraus. Dort wird im Nicht-Initialisierungsfall nur einmal lesend auf `lazyField` zugegriffen, nämlich bei der Zuweisung an die lokale Variable `tmp`. Die Performance-Verbesserung auf Grund dieser Optimierung beträgt laut Joshua Bloch rund 25%.

Noch zwei Kommentare zum Double-Check-Idiom:

Wenn man sich die drei verschiedenen Implementierungen ansieht:

- Lock ohne `volatile`
- Double-Check-Idiom, d.h. Lock und `volatile`
- optimiertes Double-Check-Idiom, d.h. Lock, `volatile` und lokale Variable

dann fällt auf, dass es immer mehr Code wird, je performanter die Lösung ist. Dabei stört nicht so sehr die größere Menge an Code, als vielmehr, dass er auch immer unintuitiver wird. Aber damit muss man sich wohl heute in Java abfinden; das Performance-Modell von Java ist nun mal an einigen Stellen in hohem Maße unintuitiv.

Vollständigkeitshalber sei noch erwähnt, dass das Double-Check-Idiom nicht die beste und einzige Lösung ist, wenn es darum geht, ein `static` Feld *lazy* zu initialisieren. Solche verzögerten Initialisierungen von statischen Feldern treten zum Beispiel im Zusammenhang mit *Singletons* auf. Hier eignet sich das *Holder-Class-Idiom* besser. Es basiert im wesentlichen darauf, das Problem der einmaligen Initialisierung an das Laufzeitsystem der JVM zu delegieren. Wir wollen diese Lösung hier aber nicht diskutieren, weil sie überhaupt gar nichts mit dem Thema `volatile` und dem Java Memory Model zu tun hat. Details zum *HolderClass-Idiom* finden sich aber auch im Item 71 von Joshua Blochs Buch.

Single-Check-Idiome

Stattdessen wollen wir uns noch zwei Varianten des *Double-Check-Idioms* ansehen, an denen man Effekte des Memory Models diskutieren kann: *Single-Check-Idiom* und *Racy-Single-Check-Idiom* (beide sind auch in Joshua Bochs Buch erwähnt). Es geht dabei darum, ob und unter welchen Umständen man die Datensynchronisation verringern kann, d.h. den `synchronized`-Block und das `volatile` weglassen, um so vielleicht die Performance zu verbessern.

Das *Single-Check-Idiom* sieht so aus, wobei wir hier die Version mit nur einem ‚Refresh' unter Verwendung einer temporären Variablen betrachten:

```
public class MyClass {
    private volatile MyField lazyField;
    ...

    public MyField getMyField() {
        MyField tmp = lazyField;
        if (tmp == null)
            lazyField = tmp = new MyField();

        return tmp;
    }

    ...
}
```

Da der `synchronized`-Block fehlt, kann es hierbei natürlich zu mehrfachen Initialisierungen kommen. Das heißt, für mehrere Threads könnte `tmp == null` sein und jeder von ihnen würde dann das Feld `lazyField` initialisieren. Das Objekt, dessen Referenz als letztes an `lazyField` zugewiesen wird, ist dann das Objekt, welches anschließend allen Threads sichtbar ist, da das Feld `lazyField` als `volatile` deklariert ist.

Diese Mehrfachinitialisierung muss aber kein Problem sein. Es gibt Situationen, wo dies toleriert werden kann. Zum Beispiel, wenn in jedem Thead das gleiche Objekt erzeugt wird und `MyField` ein nicht veränderbarer (immutable) Typ ist.

Von der Performance her ist das *Single-Check-Idiom* dem *Double-Check-Idiom* nicht wirklich überlegen. Da sich beide nur während der Initialisierung unterscheiden, dürften die Performanceunterschiede für den gesamten Programmablauf nicht signifikant sein. Für den Fall von Thread-Kollisionen bei der Initialisierung haben beide Lösungen ihre spezifischen Nachteile, die sich schlecht gegeneinander aufrechnen lassen:

- Beim *Single-Check-Idiom* werden überflüssige Objekte konstruiert, was zusätzliche CPU-Zeit kostet.
- Die Initialisierung beim *Double-Check-Idiom* wird sequentialisiert und damit skaliert sie nicht so gut.

Vorteil des *Single-Check-Idiom* ist, dass es knapper in der Implementierung ist und zusätzlich dokumentiert, dass Mehrfach-Initialisierungen hier toleriert werden können .

Bei der zweiten Variante des des *Double-Check-Idioms*, dem *Racy-Single-Check-Idiom*, fällt nicht nur der `synchronized`-Block weg, sondern auch noch das `volatile`. Ein Beispiel für das *Racy-Single-Check-Idiom* (mit `int`) sieht dann so aus:

```
public class MyClass {
    private int lazyField;
    ...

    public int getMyField() {
        if (lazyField == 0)
            lazyField = 10000;

        return lazyField;
    }

    ...
}
```

Da das `lazyField` nicht mehr `volatile` ist, braucht man auch keine temporäre Variable mehr zur Optimierung. Von der Performance her ist diese Implementierung optimal, da hier gar keine expliziten Speichereffekte mehr getriggert werden. Sie ist aber nur sehr eingeschränkt verwendbar.

Hier ist nämlich nicht mehr gesichert, dass ein Thread die Initialisierung sieht, die von einem anderen Thread zuvor durchgeführt wurde. Ausgeschlossen ist es aber auch nicht, da die Sichtbarkeit durch andere Effekte hergestellt werden kann, zum Beispiel durch benutzerseitige Synchronisation. Das wäre der Fall, wenn der Aufruf der Methode `getMyField()` in einem `synchronized`-Block erfolgt, der von beiden Threads durchlaufen wird. Dann passiert die Synchronisation von Außen und nicht in der Klasse `MyClass` selbst. In so einer Situation ist das *Racy-Single-Check-Idiom* durchaus sinnvoll.

Natürlich kann es auch beim *Racy-Single-Check-Idiom* (wie beim *Single-Check-Idiom*) passieren, dass das Feld mehrmals initialisiert wird, weil es keine Synchronisation mehr gibt. Beim *Racy-Single-Check-Idiom* gibt es aber noch zwei andere Probleme.

Wenn das Feld, das *lazy* initialisiert werden soll, kein `int`-Wert ist, sondern vom Typ `long` oder `double`, dann ist der Zugriff darauf nicht atomar. Es könnte also passieren, dass das Lesen des *lazy*-Felds einen sinnlosen Wert liefert. Andere Probleme gibt es, wenn das Feld eine Referenz auf ein Objekt ist. Dann ist zwar der Zugriff auf die Adresse atomar, aber es gibt keine Garantie, dass der lesende Thread das referenzierte Objekt in einem konsistenten Zustand sieht. Schließlich ist ohne Synchronisation oder `volatile` nicht gewährleistet, dass die Felder des Objekts jemals sichtbar gemacht werden.

Kann man bei all den Einschränkungen mit dem *Racy-Single-Check-Idiom* überhaupt etwas anfangen? In der Theorie macht es Sinn, sich das *Racy-Single-Check-Idiom* zur Abgrenzung vom *Single-Check-Idiom* einmal vor Augen zu führen. In der Praxis sind Anwendungsfälle, bei denen das *Racy-Single-Check-Idiom* sinnvoll zum Einsatz kommt, eher selten. Es gibt aber Anwendungsfälle, zum Beispiel den HashCode im String. Da wird das `int`-Feld in der Klasse `java.lang.String`, das den Hashcode des String cacht, mit dem *Racy-Single-Check-Idiom* in der Methode `hashCode()` lazy-initialisiert - ohne Synchronisation und ohne `volatile`.

Hier ist der relevante Auszug aus der Implementierung der Klasse `java.lang.String`:

```
public final class String
{
    /** Cache the hash code for the string */
    private int hash; // Default to 0

    public String() {
        this.offset = 0;
        this.count = 0;
```

```
            this.value = new char[0];
        }

        public int hashCode() {
            int h = hash;
            if (h == 0) {
                int off = offset;
                char val[] = value;
                int len = count;

                for (int i = 0; i < len; i++) {
                    h = 31*h + val[off++];
                }
                hash = h;
            }
            return h;
        }
    }
```

In diesem Anwendungsfall macht es keine Probleme, dass der eine Thread, der `hashCode()` aufruft, u.U. nicht sehen kann, was ein anderer Thread zuvor in dem `hash`-Feld abgelegt hat. Da die HashCode-Berechnung sowieso immer dasselbe Ergebnis liefert, ist es egal, ob ein Thread den Wert sieht, den er gerade selber ausgerechnet hat, oder den Wert, den zuvor ein anderer Thread berechnet hat. Es macht auch nichts, dass das Feld eventuell zweimal initialisiert wird, weil sich der HashCode eines Strings nicht ändern kann. Also kann es nicht passieren, dass die zweite Initialisierung einen vom Default- und Initialwert abweichenden "aktuellen" Wert des `hash`-Feldes überschreibt. Das *Racy-Single-Check-Idiom* funktioniert hier natürlich nur, weil `java.lang.String` ein unveränderlicher (immutable) Typ ist; sonst wäre der HashCode nämlich nicht immer gleich und dann wäre eine Lösung ohne Synchronisation und ohne `volatile` falsch.

In einem der nächsten Beitrag wollen wir uns dann einen weiteren Anwendungsfall für das *Racy-Single-Check-Idiom* ansehen. Wie wir oben schon erwähnt habe, ist *Racy-Single-Check-Idiom* problematisch, wenn das lazy-initialisierte Feld eine Referenz auf ein Objekt ist. Das gibt aber nur, wenn das referenzierte Objekt veränderlich ist. Bei der Verwendung eines unveränderlichen (immutable) Typs ist die Initialisierung ohne Synchronisation und ohne `volatile` durchaus sinnvoll. Dazu muss der unveränderliche Typ aber korrekt implementiert sein und dazu werden wiederum die Speichergarantien für `final`-Felder gebraucht - und das wollen wir uns in den nächsten Beiträgen genauer ansehen.

Zusammenfassung

In diesem Beitrag haben wir die Speichereffekte von `volatile` Variablen am konkreten Beispiel des *Double-Check-Idioms* und einiger seiner Varianten diskutiert. In unserem nächsten Beitrag wollen wir uns noch einmal typische Anwendungsidiome für `volatile` ansehen.

Verweise

/EFF3/ JMM - Die Kosten der Synchronisation
Klaus Kreft & Angelika Langer
JavaMagazin 9.08

/EFF4/ JMM – Details zu `volatile` Variablen
Klaus Kreft & Angelika Langer
JavaMagazin 10.08

/DCS/ Reality Check
Douglas C. Schmidt
http://www.cs.wustl.edu/~schmidt/editorial-3.html

/DCLB/ The “Double-Checked Locking is Broken” Declaration
David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, John D. Mitchell, Kelvin Nilsen, Bill Pugh Emin Gun Sirer
http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

/BLCH/ Effective Java, 2nd Ed.
Joshua Bloch
Addison-Wesley, 2008

Insert: Das Double-Check-Idiom und seine Geschichte

Fangen wir erst einmal beim Namen an, denn für den gibt es einige Varianten. Für den vorderen Teil gibt es *Double-Check* und *Double-Checked*, für den hinteren Namensbestandteil *Idiom*, *Pattern* und *Locking*. In Kombination ergeben sich damit sechs Möglichkeiten von: *Double-Check-Idiom* bis *Double-Checked Locking*. Wir haben, weil es am kürzesten ist, *Double-Check-Idiom* verwendet.

Unseres Wissen nach ist das Idiom zum ersten Mal von Doug Schmidt in seinem Editorial der März-Ausgabe des C++-Reports 1996 beschrieben worden worden. Der Originaltext befindet sich heute in Doug Schmidts Archiv (/DCS/).

Das Idiom ist dann, wie viele andere Programmiertechniken, von C++ nach Java übernommen worden. Irgendwann ist einigen schlauen Leuten (namentlich: David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, John D. Mitchell, Kelvin Nilsen, Bill Pugh und Emin Gun Sirer) aufgefallen, dass das Verhalten von `volatile` Variablen, wie es in der damaligen Java Language Specification (JLS) beschrieben war, nicht ausreichend ist, um ein Funktionieren des Double-Check-Idioms in Java zu garantieren. Die Details sowie weitere nichtfunktionierende und funktionierende Varianten/Alternativen haben sie unter dem Titel *The "Double-Checked Locking is Broken" Declaration* (/DCLB/) veröffentlicht.

Gleichzeitig hat sich die Java Community gefragt, ob dieses Problem nicht auf Grund eines Defizits in Java entstanden ist. Das heißt, sollte man Java nicht so ändern, dass so etwas wie das Double-Check-Idiom funktioniert? Aus diesen Überlegungen entstand dann der JSR 133: *Java Memory Model and Thread Specification Revision* (Spec Lead: Bill Pugh). Das Ergebnis dieses JSRs war dann das mit Java 5.0 neu eingeführte Memory Model, dessen Bedeutung für `volatile` wir im letzten Artikel ausführlich theoretisch diskutiert haben und dessen Bedeutung für die Praxis wir hier am Beispiel des Double-Check-Idioms aufzeigen. So schließt sich der Kreis.

Java Memory Modell

Teil 6: Regeln für die Verwendung von volatile

Klaus Kreft & Angelika Langer

Wir haben in den vorangegangenen Beiträgen dieser Kolumne erläutert, welche Garantien das Java Memory Modell bezüglich der Verwendung von volatile-Feldern gibt und dass der Einsatz von `volatile` eine Alternative zur Verwendung von Locks darstellt. Es stellt sich die Frage: wann ist es sinnvoll, `volatile` zu verwenden und wann sollte man besser Locks oder andere Synchronisationsmechanismen benutzen? In diesem Beitrag wollen dieser Frage ein Stück weit nachgehen.

Effekte von Synchronisation

Synchronisationsmechanismen haben in der Regel zwei Effekte:

- *Atomicity*. Synchronisationsmechanismen garantieren, dass gewisse Operationen oder Sequenzen von Operationen ununterbrechbar (atomar) sind. Das bedeutet, dass von den Threads, die an der Synchronisation beteiligt sind, immer nur ein Thread zu einer Zeit exklusiv die atomare Sequenz von Operationen ausführt; alle anderen an der Synchronisation beteiligten Thread müssen warten. Die atomare Sequenz wird von den Threads also stete nacheinander, aber nie gleichzeitig oder ineinander verschränkt ausgeführt.
- *Visibility*. Synchronisationsmechanismen bieten Sichtbarkeitsgarantieren, d.h. dass Änderungen, die die an der Synchronisation beteiligten Threads im Speicher gemacht haben, anderen beteiligten Threads sichtbar werden.

Wir haben Synchronisation mit Locks und Synchronisation mit `volatile` in den letzten Artikel ausführlich diskutiert.

Locks sorgen dafür, dass in einem Thread die Operationen zwischen Anfordern und Freigeben desselben Locks nicht konkurrierend mit Operationen anderer Threads ausgeführt werden, die ebenfalls dasselbe Lock verwenden. Das bezeichnet man als Mutual Exclusion und es führt dazu, dass Sequenzen von Operationen atomar werden. Daneben hat das Anfordern und Freigeben von Locks Speichereffekte: das Anfordern eines Lock löst einen Refresh des thread-spezifischen Caches aus, das Freigeben löst einen Flush aus. Dadurch werden Änderungen, die ein Thread unter dem Schutz eines Locks im Speicher gemacht hat, anderen Threads, die dasselbe Lock verwenden, sichtbar.

`volatile` hat die gleichen Speichereffekte wie Locks, aber es hat wesentlich schwächere Atomicity-Eigenschaften. Das Lesen einer `volatile` -Variable löst einen Refresh des thread-spezifischen Caches aus, das Schreiben löst einen Flush aus. Dadurch werden Änderungen, die ein Thread vor und mit dem Schreibzugriff auf eine `volatile` -Variable im Speicher gemacht hat, anderen Threads, die lesend auf dieselbe `volatile` -Variable zugreifen, sichtbar. Der lesende und schreibende Zugriff auf `volatile` -Variable ist atomar, Sequenzen von solchen Zugriffen sind es aber nicht. Die Atomicity-Garantie gilt also nur für den einfachen lesenden oder schreibenden Zugriff auf die einzelen `volatile` -Variable. Inkrement und Dekrement von numerischen `volatile` -Variable sind zum Beispiel keine atomaren Operationen. Bei `volatile` -Referenzvariablen beziehen sich die Garantieren nur auf die Referenz, nicht auf das referenzierte Objekt (oder Array).

Es ist außerdem zu beachten, dass die Garantien immer nur dann gelten, wenn die Zugriffe zusammenpassen: Sichtbarkeit und Ununterbrechbarkeit gibt es nur für die Threads, die dasselbe Lock verwenden oder auf dieselbe `volatile` -Variable zugreifen. Anfordern/Freigeben von Lock A verhindert nicht, dass ein Thread konkurrierend läuft, der Lock B anfordert/freigibt. Analog, der lesende Zugriff auf `volatile`-Variable A macht nicht sichtbar, was ein anderer Thread bis zum schreibenden Zugriff auf `volatile`-Variable B im Speicher geändert hat. Der Mismatch passiert oft versehentlich und führt dann unvermeidlich zu Fehlern.

Regeln für den Einsatz von volatile

Wir haben in vorangegangenen Artikeln die Verwendung von `volatile` als Alternative zur Synchronisation mithilfe von Locks besprochen. Da die Garantien für `volatile`-Variablen geringer sind als für Locks, sind `volatile`-Variablen natürlich nicht immer eine Alternative. Es geht nur, wenn die Ununterbrechbarkeit, die die Locks liefern, nicht gebraucht wird. Das ist in folgenden Fällen der Fall:

Man kann den Zugriff auf Variablen mit `volatile` statt mit Locks synchronisieren, wenn:	
Regel 1:	– der zu schreibende neue Wert der Variablen unabhängig vom gegenwärtigen Wert ist, und
Regel 2:	– die Variable unabhängig von anderen Variablen ist.

Unabhängigkeit von gegenwärtigen Wert

Warum ist es wichtig, dass der neue Wert vom alten unabhängig ist? Das liegt daran, dass bei einer solchen Abhängigkeit erst einmal der alte Wert gelesen werden muss, dann muss darauf basierend der neue Wert errechnet werden und dann muss der neue Wert geschrieben werden. Das ist eine Read-Modify-Write-Sequenz, die als Ganzes ununterbrechbar sein muss, sonst kommt bei der Operation was anderes heraus, als beabsichtigt ist. `volatile`-Variablen bieten aber keine ununterbrechbaren Sequenzen von Operationen an. Also geht das mit `volatile` nicht.

Hier ist ein Beispiel:

```
public class Counter {  // Vorsicht: falsch !!!
  private volatile int value;
  public int getValue()  { return   value; }
  public int increment() { return ++value; }
  public int decrement() { return --value; }
}
```

Dieser angeblich thread-sichere Zähler ist nicht thread-sicher, weil die Inkrementierung und Dekrementierung unterbrechbar sind. Ein benutzender Thread bekommt unter Umständen beim Inkrementieren vom Zähler mit Wert 2 als neuen Wert 4 heraus, oder irgendeinen anderen von 3 verschiedenen Wert, weil beim Inkrementieren ein anderer Thread dazwischen gekommen ist.

Das obige Problem kann man elegant mit atomaren Variablen lösen. Der `AtomicInteger` zum Beispiel hat ununterbrechbare Inkement- und Dekrement-Operationen. Das wollen wir an dieser Stelle aber nicht näher betrachten. Atomare Variablen werden wir uns in einem der nächsten Beiträge ansehen.

Unabhängigkeit von anderen Variablen

Warum ist es wichtig, dass die `volatile`-Variable unabhängig von anderen Variablen ist? Das liegt daran, dass sie dann Teil einer Invarianten ist und die Invariante nur als Ganzes geändert werden kann. Die Invariante eines Objekts beschreibt, wann die Felder des Objekts in einem konsistenten und gültigen Zustand sind. Wenn nur ein Teil der Invarianten geändert wird, dann entsteht in der Regel ein inkonsistenter und ungültiger Zustand des Objekts. Man kann also nicht nur einen Teil des Objekts ändern, sondern man muss dafür sorgen, dass alle zur Invarianten gehörenden Teile auf einmal geändert werden und keine ungültigen Zwischenzustände sichtbar werden. Es muss also eine ganze Sequenz von Modifikationen auf den verschiedenen Teilen der Invarianten ununterbrechbar sein. Das leisten `volatile`-Variablen leider nicht.

Hier ist ein Beispiel:

```
public class NumberRange {  // Vorsicht: falsch !!!
  private volatile int lower;
  private volatile int upper;
  public void setLower(int i) {
        if (i > upper) throw new IllegalArgumentException(…);
        lower = i;
  }
  public boolean isInRange(int i) {
        return (i >= lower && i <= upper);
  }
  …
}
```

Die Invariante hier ist: `lower` muss kleiner als `upper` sein. Damit diese Invariante erhalten bleibt, muss in der Methode `setLower()` der Wert von `upper` gelesen werden, verglichen werden, und dann muss ein neuer Wert für `lower` geschrieben werden. Jede dieser Operationen ist einzeln betrachtet atomar, aber die Sequenz ist es nicht. Das müsste sie aber sein, sonst könnte folgendes passieren: Nehmen wir an, der Bereich ist `[2;5]`. Der Wert von `upper` (also `5`) wird gelesen und mit dem neuen `lower` (z.B. `4`) verglichen wird. Dann kommt aber ein anderer Thread dazwischen, der `upper` auf `3` ändert, was der erste Thread nicht merkt, so dass am Ende der ungültige Bereich `[4;3]` herauskommt.

Beispiele für den erfolgreichen Einsatz von volatile

Wie man an den Beispielen sehen kann, ist der Einsatz von `volatile` fehleranfällig und unter Umständigen schwieriger als die Synchronisation mit Locks. Man kann sich an den oben genannten Regeln orientieren. Im Folgenden wollen wir ein paar typische Idiome zeigen, in denen der Einsatz von `volatile` sinvoll und korrekt ist.

Einmalige Ereignisse

Variablen, die genau einmal geändert werden und sich danach nie wieder ändern, können gut mit `volatile`-Variablen ausgedrückt werden. Bei solchen Abstraktionen, die nach der Änderung in dem neuen Zustand hängen bleiben, spricht man von einem *Latch*. Solche Latches werden gerne als Start- oder Ende-Signale verwendet.

Hier ist ein Beispiel:

```
class Service {
  private volatile boolean shutdownRequested;
  ...
  public void shutdown() {
    shutdownRequested = true;
  }
  public void doWork() {
    while (!shutdownRequested) {
        // do stuff
    }
  }
}
```

Hier ist das Latch eine Boolesche Variable, die als Ende-Signal verwendet wird. Das Status-Flag `shutdownRequested` kann nur in eine Richtung (von `false` auf `true`) gesetzt werden. Danach ändert es sich nicht mehr.

Hier ist das Flag nicht Teil einer Invarianten; das Flag ist unabhängig vom restlichen Zustand des Objekts. Der neue Wert des Flags hängt nicht vom alten Wert des Flags ab. Synchronisation wird auch aus sonst keinem Grunde gebraucht, weil der Zugriff auf den primitiven Typ `boolean` sowieso atomar ist. Man braucht hier nur die Garantie, dass der Wert, den der Shutdown-Thread per Aufruf von `shutdown()` gesetzt hat, dem Worker-Thread sichtbar wird. Diese Sichtbarkeit garantiert die `volatile`-Deklaration. Also kann man hier auf Synchronisation verzichten.

Wichtig ist bei diesem Idiom, dass es sich bei der Modifikation der `volatile`-Variablen um ein Latch handelt. Wenn die Variable toggelt, also zwischen `true` und `false` hin- und herpendelt, dann wird man möglicherweise andere Synchronisationsmechanismen (wie Locks oder atomare Variablen) brauchen, weil man zum Beispiel verhindern will, dass Zustandsänderungen verloren gehen. `volatile` sorgt nur für die Sichtbarkeit. Wer sich die Variable wann und wie oft anschaut oder ändert oder über Änderungen informiert wird, ist über `volatile` nicht geregelt.

Hier noch ein Beispiel für ein solches One-Time-Event:

```
public class Precursor {
  public volatile Service theService;

  public void initInBackground() {
    ... do lots of stuff ...
    // this is the only write to theService
    theService = new Service();
  }
}
public class Worker {
  public void doWork() {
    while (true) {
      ... do some stuff ...
      // use the service, but only if it is ready
      if (backgroundLoader.theService != null)
        doSomething(backgroundLoader.theService);
    }
  }
}
```

Hier ist das Latch keine Boolesche Variable, sondern eine Referenz, die ihren Wert einmal von `null` auf einen von `null` verschiedenen Wert ändert. Diese Änderung wird als Startsignal betrachtet: der Worker-Thread fängt erst an zu arbeiten, wenn der Service verfügbar ist.

Ohne die `volatile`-Deklaration könnte es sein, dass die Adresse nicht sichtbar wird, oder nur die Adresse, nicht aber die Inhalte des referenzierten Objekts. Man braucht hier die Garantie, dass die Referenz, die der vorbereitende Thread gesetzt hat, dem Worker-Thread sichtbar wird, inklusive des Inhalts des referenzierten Objekts. Diese Sichtbarkeit garantiert die `volatile`-Deklaration der Referenzvariablen `theService`. Beim Schreibzugriff wird alles sichtbar gemacht, was bis zu

diesem Zeitpunkt im Speicher gemacht wurde, d.h. es wird die Adresse des neuen `Service`-Objekts sichtbar und dessen Inhalt, weil das Objekt vor der Adresszuweisung erzeugt wurde.

Die `volatile`-Deklaration genügt, weil die Referenz auf das `Service`-Objekt nicht Teil einer Invarianten, sondern unabhängig von allem anderen ist. Der neue Wert der Referenz hängt auch nicht vom alten Wert der Referenz ab. Synchronisation wird auch aus sonst keinem Grunde für die Veröffentlich des `Service`-Objekts gebraucht, weil der Zugriff auf Referenzen sowieso atomar ist.

Die `volatile`-Deklaration bewirkt aber nicht, dass die nachfolgenden Zugriffen auf das `Service`-Objekt sicher sind. Gesichert ist hier nur, dass die erstmalige Veröffentlichung im Rahmen der Initialisierung des `Service`-Objekts funktioniert, d.h. dass seine Adresse und seine Inhalte den Worker-Threads sichtbar werden. Was danach passiert, ist unklar; jedenfalls hat es nichts mit der `volatile`-Deklaration zu tun, sondern für die Sicherheit der nachfolgenden Zugriffen auf das `Service`-Objekt muss mit anderen Mitteln gesorgt werden.

Mehrfach-Veröffentlichung von Information

Betrachten wir den Fall, dass Informationen nicht nur einmal zur Verfügung gestellt werden soll, sondern immer wieder neue Information anderen Threads gegenüber veröffentlicht werden soll. Kann man dafür `volatile` gebrauchen? Unter Umständen geht es - und zwar immer dann, wenn sich die veröffentlichte Information selbst nicht verändert, sondern stets durch komplett neue ersetzt wird.

Hier ist ein Beispiel:

```
class TemperatureSensor {
    private volatile double theTemperature;

    public void setTemperature(double temperature) {
        theTemperature = temperature;
    }
    public void displayTemperature() {
        double currentTemparature = theTemperature;
        ... display currentTemparature ...
    }
}
```

Hier legt ein Sensor-Thread Temperaturinformation in einer `volatile`-Variablen ab, die andere Threads lesen und anzeigen.

Hier ist der `double`-Wert nicht Teil einer Invarianten; es gibt keine weiteren Felder. Der neue Temperaturwert hängt nicht vom alten Wert ab. Der Zugriff auf den `double`-Wert wäre ohne die `volatile`-Deklaration nicht atomar, aber wenn die `double`-Variable als `volatile` deklariert ist, dann ist der Zugriff atomar; Locks werden für Atomarität jedenfalls nicht gebraucht. Man braucht allerdings die Garantie, dass der Wert, den der Sensor-Thread per Aufruf von `setTemperature()` gesetzt hat, dem Display-Thread sichtbar wird. Diese Sichtbarkeit garantiert ebenfalls die `volatile`-Deklaration. Also kann man hier auf Synchronisation verzichten.

Allerdings muss man beachten, dass sich der Inhalt des der `double`-Variablen jederzeit ändern kann. Es können Änderungen verloren gehen, in dem Sinne, dass der Display-Thread sie nicht anzeigt, weil er sie verpasst hat. Es kann auch passieren, dass sich gar nichts geändert hat und der Display-Thread denselben Wert mehrmals anzeigt. Solange das akteptabel ist, funktioniert das Idiom. Sobald aber Werte nicht verloren gehen dürfen oder Mehrfachverwertungen desselben Werts unerwünscht sind, braucht man andere Synchronisationsmechanismen (wie Locks oder atomare Variablen, vielleicht sogar Conditions).

Kombination von Lock und volatile

Interessant ist die Verwendung von `volatile` in Kombination mit Locks. Ein Beispiel dafür ist das Double-Check-Locking, das wir im letzten Beitrag (siehe /EFF5/) besprochen haben. Es gibt aber auch andere Beispiele für die Kombination von `volatile` und Locks.

Betrachten wir noch einmal das anfängliche Beispiel der `Counter`-Klasse. Eine Lösung mit `volatile` allein ist nicht möglich, weil Inkrement und Dekrement unterbrechbare Operationen sind. Wenn man für die Ununterbrechbarkeit von `increment()` und `decrement()` mit Hilfe von Locks sorgt, dann bleibt aber immer noch die `getValue()`-Methode, die bereits atomar ist

und eigentlich keine Locks braucht. Man könnte jetzt alle Methoden als `synchronized` deklarieren oder man versucht eine preiswertere Lösung, indem man Synchronisation per Lock und per `volatile` kombiniert. Das sähe so aus:

```
class Counter {
  private volatile int value;
  public             int getValue()  { return   value; }
  public synchronized int increment() { return ++value; }
  public synchronized int decrement() { return --value; }
}
```

`value` ist nicht Teil einer Invarianten, weil es keine weiteren Felder gibt. Der neue Wert hängt zwar vom alten Wert des Flags ab; deshalb brauchen wir Synchronisation in `increment()` und `decrement()`. Aber die Methode `getValue()` macht keine Sequenz von Operationen, die von alten Wert abhinge; `getValue()` ist eine rein lesende Methode und ändert sowieso nichts am Objekt. Es spricht also nichts dagegen, sich in der Methode `getValue()` allein auf die Garantien von `volatile` zu verlassen. Atomarität haben wir schon, weil value vom Typ `int` ist, und die Sichtbarkeit bekommen wir durch die `volatile`-Deklaration. Also kann man in der Methode `getValue()` auf Synchronisation verzichten.

Dieses Beispiel ist natürlich etwas komplex, weil es gleich zwei Synchronisationsmechnismen gleichzeitig einsetzt. In komplizierten Fällen kann die gleichzeitige Verwendung beider Mechanismen leicht zu Fehlern führen.

Ziel des Einsatzes von volatile

Wie die Beispiele gezeigt haben, gibt es eine Reihe von Situationen, in denen `volatile` als Synchronisationsmechanismus einsetzbar ist. Wenn man sich nicht sicher ist, dann hat man generell immer die Möglichkeit, an allen fraglichen Stellen Locks zu verwenden. Warum dann also überhaupt `volatile` einsetzen?

Wir haben dazu in der vorletzten Ausgabe /EFF3/ besprochen, dass das Anfordern und Freigeben von Locks tendenziell für die Virtuelle Maschine aufwändiger ist als der Zugriff auf `volatile`-Variablen und deshalb mehr Performance kostet. `volatile` ist aber auch nicht kostenlos, weil es sogenannte Memory Barriers auslöst, die sich ungünstig auf den Caching-Mechanismus der Hardware auswirken. Generell ist das Anfordern und Freigeben von Locks weniger performant als der Zugriff auf `volatile`-Variablen und der ist wiederum weniger performant als der Zugriff auf non-`volatile`-Variablen, wobei die Peformance-Unterschiede je nach Plattform anders sein können. Das spricht für den Einsatz von `volatile` als Syncrhonisationsmechanismus zum Zwecke der Performance-Verbesserung.

Andererseits wenden die Virtuellen Maschinen ein ganze Reihe Optimierungstechniken für Locks an. Bei der Sun JVM ab Version 6.0 sind es zum Beispiel (siehe /PERF/ und /JTP/):

- *Lock Coarsening*. Dabei wird ein Lock über längere Zeit (z.B. für eine ganze Schleife) gehalten, statt es mehrfach für eine kürzere Zeit (z.B. je Schleifendurchlauf) anzufordern und freizugeben. (siehe Option `-XX:-EliminateLocks`)
- *Biased Locking*. Das ist eine Optimierung für Locks, die nicht oder nur von einem einzigen Thread angefordert werden. (siehe Option `-XX:-UseBiasedLocking`)
- *Adaptive Locking*. Mit dieser Optimierung wird versucht, die Kosten des Auslagerns und Wiederaktivierens eines Thread, der auf ein Lock wartet, zu reduzieren.
- *Lock Elision*. Dabei wird durch eine sogenannte *Escape Analysis* festgestellt, ob ein Lock-Objekt nur lokal in einer Methode benutzt wird. Dann wäre das Locking überflüssig und wird entfernt. (siehe Option `-XX:-DoEscapeAnalysis`)

Was für die Performance am Ende am günstigsten ist, muss man - wie eigentlich immer, wenn es um Performance-Aussagen geht - per Micro-Benchmark bestimmen. Es überlagern sich im Einzelfall so viele Effekte, dass man allgemein gesprochen nur Tendenzen nennen kann.

Selbst wenn der Performance-Gewinn durch die Verwendung von `volatile` nicht groß ist, dann hat `volatile` als Synchronisationstechnik immer noch einen Vorteil gegenüber den Locks: es skaliert besser, weil immer ein Thread erfolgreich zugreifen kann. Es gibt keine Wartezustände und deshalb kann es auch keine Deadlocks geben. Das heißt, Situationen, in denen alle an der Synchronisation beteiligten Threads aufeinander warten, können mit `volatile` nicht passieren.

Zusammenfassung

In diesem Beitrag haben wir uns detailliert angesehen, worauf man achten muss, wenn man `volatile`-Variablen als Synchronisationsmechanismus anstelle von Locks einsetzen will. Ziel einer solche Maßnahme ist in der Regel die

Verbesserung der Performance und/oder der Skalierbarkeit. `volatile` ist als Synchronisationsmechanismus aber schwächer als Locks, weil es zwar die gleichen Sichtbarkeitsgarantien gibt, aber kaum für Ununterbrechbarkeit von Zugriffen sorgt; lediglich einfache Lese- und Schreibzugriffe auf Referenzvariablen und Variablen von primitivem Typ sind atomar. Deshalb kann der Zugriff auf Variablen nur dann mit `volatile` statt mit Locks synchronisiert werden, wenn es keine Sequenzen von Operationen gibt, die ununterbrechbar sein müssen. Dazu muss der zu schreibende neue Wert der Variablen unabhängig vom gegenwärtigen Wert sein und die Variable darf nicht Teil einer Invarianten sein, sondern muss unabhängig von anderen Variablen sein.

In den nächsten Beiträgen sehen wir uns an, welche Garantien das Java Memory Modell für `final`-Felder gibt und wozu man diese Garantien braucht.

Verweise

/EFF1/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/EFF2/ JMM - Überblick über das Java Memory Modell (JMM)
Klaus Kreft & Angelika Langer
JavaMagazin 8.08

/EFF3/ JMM - Die Kosten der Synchronisation
Klaus Kreft & Angelika Langer
JavaMagazin 9.08

/EFF4/ JMM – Details zu `volatile` Variablen
Klaus Kreft & Angelika Langer
JavaMagazin 10.08

/EFF5/ JMM – Double-Check Locking
Klaus Kreft & Angelika Langer
JavaMagazin 11.08

/PERF/ Java SE 6 Performance White Paper
Sun Microsystems, Inc.
URL: http://java.sun.com/performance/reference/whitepapers/6_performance.html#2.1

/JTP/ Synchronization optimizations in Mustang
Brian Goetz
IBM developerWorks, Oktober 2005
URL: http://www.ibm.com/developerworks/java/library/j-jtp10185/index.html

Java Memory Modell

Teil 7: Die Initialisation-Safety-Garantie

Klaus Kreft & Angelika Langer

Wir haben in einem der vorangegangenen Beiträge [EFF2] erwähnt, dass es besondere Garantieren für die Sichtbarkeits- und Speichereffekte im Zusammenhang mit `final` gibt. Diese Garantien wollen wir uns in diesem Beitrag genauer ansehen.

Racy-Single-Check mit einem primitiven Typ (außer `long` und `double`)

Als wir im letzten Artikel [EFF5] die Verwendung von `volatile` am Beispiel des Double-Check-Idioms besprochen haben, haben wir uns auch das sogenannte *Racy-Single-Check*-Idiom angesehen. Bei diesem Idiom wird weder Synchronisation noch `volatile` genutzt. Entsprehend ist seine Anwendung sehr eingeschränkt, aber es gibt Fälle, in denen es korrekt und unproblematisch ist, nämlich wenn das zu initialisierende Feld einen kostanten Wert zugewiesen bekommt und von einem primitiven Typ (außer `long` und `double`).

Hier noch einmal unser Beispiel des *Racy-Single-Check*-Idioms aus unserem letzten Artikel:

```
public class MyClass {
    private int lazyField;
    ...

    public int getMyField() {
        if (lazyField == 0)
            lazyField = 10000;

        return lazyField;
    }

    ...
}
```

Für die korrekte Verwendung des *Racy-Single-Check*-Idiom im Beispiel oben sind folgende Überlegungen wichtig:

– *Der Zugriff auf das fragliche Feld muss atomar sein.* Sonst wäre der Zugriff unterbrechbar und das Feld könnte (aus Sicht des lesenden Threads) sinnlose Werte enthalten. Die Ununterbrechbarkeit des Zugriffs auf primitive Typen (außer `long` und `double`) ist in Java garantiert.

– *Das Feld `lazyField` ändert sich nach der Initialisierung nicht mehr*. Man würde es gerne als `final` deklarieren, um diese Eigenschaft sichtbar zu machen, aber das geht nicht, weil die Initialisierung nicht im Konstruktor, sondern "lazy" in der Methode `getMyField()` erfolgt. Semantisch ist das Feld aber `final`, sonst wäre die Verwendung des *Racy-Single-Check*-Idiom falsch. Es könnte sonst passieren, das zwei Thread gleichzeitig "`lazyField == 0`" sehen und beide nacheinander den Initialwert zuweisen. Wenn zwischendrin ein dritter Thread das Feld bereits verändert hätte, dann ginge diese Veränderung verloren. Es ist also wichtig, dass der Inhalt des Feldes unveränderlich ist.

Ein bekanntes Anwendungsbeispiel für das *Racy-Single-Check*-Idiom ist der Hashcodewert in der Klasse `java.lang.String`. Er wird genau mit dieser Technik *lazy* initialisiert.

Racy-Single-Check mit einem Referenztyp

Wie funktioniert nun das *Racy-Single-Check*-Idiom mit einem Feld, dass von einem Referenztyp ist?

Hier ist ein Beispiel mit einer Referenz auf einen Integer vom Typ `java.lang.Integer`:

```
public class MyClass {
    private Integer lazyField = null;  // default value
    ...

    public Integer getMyField() {
        if (lazyField == null)
            lazyField = new Integer(10000);

        return lazyField;
    }

    ...
}
```

Hier gehen nun drei Überlegungen ein, die die korrekte Verwendung des *Racy-Single-Check*-Idiom garantieren:

– *Der Zugriff auf das fragliche Feld muss atomar sein.* Das ist auch hier der Fall; die Ununterbrechbarkeit des Zugriffs auf Adressen, also Variablen von einem Referenztyp, ist in Java garantiert.

– *Die Referenz `lazyField` ändert sich nach der Initialisierung nicht mehr.* Auch hier würde man das Feld gerne als `final` deklarieren, aber es geht wieder nicht wegen der "lazy" Initialisierung. Die `final`-Deklaration der Referenz bezöge sich ohnehin nur auf die Adresse des referenzierten Objekts und nicht auf das Objekt und seine Inhalte. Ein unveränderliche Adresse reicht aber für das *Racy-Single-Check*-Idiom nicht aus; es muss eine dritte Voraussetzungen erfüllt sein.

– *Das referenzierte Objekt und seine Inhalte ändern sich nach der Initialisierung nicht mehr.* Wenn das Objekt veränderlich wäre, dann wäre die Verwendung des *Racy-Single-Check*-Idiom falsch, denn das Idiom lässt Mehrfach-Initialisierung zu. Es könnte passieren, das zwei Thread gleichzeitig "`lazyField == null`" sehen und beide nacheinander den Initialwert zuweisen. Eine Veränderung des referenzierten Objekts, die zwischendrin ein dritter Thread vorgenommen hätte, ginge verloren. Es ist also wichtig, dass nicht nur die Referenz, sondern auch das referenzierte Objekt unveränderlich ist.

Das bedeutet, dass das *Racy-Single-Check*-Idiom nur sinnvoll ist, wenn das Feld eine unveränderliche Referenz auf einen unveränderlichen (immutable) Typ ist.

Anforderungen an unveränderliche Typen

Nun ist der Typ `java.lang.Integer` bekanntlich ein unveränderlicher Typ und er ist auch so implementiert, dass das *Racy-Single-Check*-Idiom mit einem `Integer` funktioniert, aber das gilt nicht für jeden Typ, der von sich behauptet unveränderlich zu sein. Es genügt nämlich nicht, dass es in einem unveränderlichen Typ keine modifizierenden Methoden gibt.

Ein unveränderlicher Typ muss auch für die Sichtbarkeit seiner Inhalte sorgen, das heißt, er muss sicher stellen, dass die unveränderlichen Inhalte des Objekts nach der Konstruktion allen benutzenden Threads sichtbar werden. Denn was nützt es uns, wenn die Threads zwar die Adresse des `Integers` nach der Lazy-Initialisierung sehen, aber nicht die Inhalte des `Integers`?

Um für die Sichtbarkeit zu sorgen, braucht man bei der Implementierung eines unveränderlichen Typs die sogenannte "*Initialization-Safety*"-Garantie des Java Memory Modells. Das ist eine Regel für die Sichtbarkeit der Inhalte der `final`-Felder von Objekten.

Sehen wir uns also die "*Initialization-Safety*"-Garantie des Java Memory Modells mal genauer an.

Speichereffekte im Zusammenhang mit final-Feldern

Es geht bei der "*Initialization Safety*"-Garantie des Java Memory Modells darum, dass stets die Initialwerte von `final`-Feldern und niemals die Defaultwerte sichtbar sind. Wenn also ein Thread ein Objekt mit `final`-Feldern zu sehen bekommt, weil er die Adresse des Objekts sehen kann, dann sieht er die `final`-Felder des Objekts stets im Initialzustand *nach* der Konstruktion und nie im Defaultzustand *vor* der Konstruktion.

Was heißt das genau? Betrachten wir ein erstes einfaches Beispiel einer Klasse mit einem `final`-Feld:

```
public class Immutable {
  private final int field;
  public Immutable (int init) {
    field = init;
  }
  public String toString() {
    return "["+ field +"]";
  }
}
```

Die Klasse ist dem unveränderlichen Typ `java.lang.Integer` nachempfunden. Sie könnte beispielsweise als Typ des `lazyField` in unserem *Racy-Single-Check*-Idiom vorkommen.

```
public class MyClass {
    private Immutable lazyField = null;  // default value
    ...

    public Immutable getMyField() {
        if (lazyField == null)
            lazyField = new Immutable(10000);
```

```
        return lazyField;
    }
    ...
}
```

Nehmen wir nun einmal an, dass zwei Threads gleichzeitig auf ein Objekt vom Typ `Immutable` zugreifen.

```
public class Test {
  private static MyClass globalRef = new MyClass();

  public static void main(String[] args) {
    Runnable r = new Runnable() {
                 public void run() {
                   System.out.println(globalRef.getMyField().toString());
                 }
               };
    new Thread(r).start();
    new Thread(r).start();
  }
}
```

Beide Threads holen sich über die Methode `getMyField()` der Klasse `MyClass` die Referenz auf das `Immutable`-Feld des `MyClass`-Objekts und rufen anschließend auf dem `Immutable`-Feld die `toString()`-Methode der Klasse `Immutable` auf. Dann könnte es so auskommen, dass der eine Thread in der Methode `getMyField()` die Referenz `lazyField` auf das `Immutable`-Feld noch als `null` vorfindet, weil noch niemand die *lazy*-Initialisierung für das Feld gemacht hat. Der andere Thread findet möglicherweise schon eine von `null` verschiedene Referenz vor und greift über diese Referenz auf das `Immutable`-Objekt zu und ruft dessen `toString()`-Methode auf. In dieser Situation stellt sich die Frage, in welchem Zustand der zweite Thread den Inhalt des referenzierten `Immutable`-Objekts zu sehen bekommt.

Wenn das Feld `field` in der Klasse `Immutable` nicht `final` wäre, dann könnte es in dieser Situation passieren, dass der zweite Thread das referenzierte `Immutable`-Objekt sieht und das Feld `field` in diesem Objekt entweder den Wert `0` oder `10000` hat. Welchen der beiden Werte das Feld hat, ist undefiniert. Es kann sogar passieren, dass der zweite Thread, wenn er mehrmals liest, erst den Wert `0` und später den Wert `10000` zu sehen bekommt. Das sieht dann so aus, als sei das unveränderliche Objekt vom Typ `Immutable` gar nicht unveränderlich, weil sich sein Inhalt augenscheinlich ändert. Wie kann das sein?

Solche Effekte können bei fehlender `final`-Deklaration entstehen, weil es ohne die `final`-Deklaration keinerlei Garantien für die Sichtbarkeit des Feldes `field` gibt.

Sichtbarkeitsprobleme im Detail

Sichtbarkeitsprobleme sind generell ein bischen schwierig zu verstehen, weil sie immer Probleme zwischen Threads sind und nie innerhalb eines Threads entstehen. Ein Sichtbarkeitsproblem gibt es nur dann, wenn ein Thread beobachtet, was ein anderer Thread im Speicher macht. In unserem Beispiel ist es so, dass der erste Thread eine `null`-Referenz vorfindet und dann die *lazy*-Initialisierung macht, also das `Immutable`-Objekt konstruiert und dessen Adresse in der Referenzvariablen `lazyField` ablegt. So sieht es innerhalb des ersten Threads aus und es entspricht unserer Intuition: erst wird das Objekt alloziert (dann ist es in einem Defaultzustand), dann konstruiert (dann ist es in seinem Initialzustand) und danach wird seine Adresse dem Feld `lazyField` zugewiesen.

Für den zweiten Thread sieht die Sache u.U. fundamental anders aus. In der oben geschilderten Situation haben wir angenommen, dass das `int`-Feld in der Klasse `Immutable` nicht `final` ist und es daher keine Sichtbarkeitsgarantien gibt. Dann ist undefiniert, welche von den Speichermodifikationen, die der erste Thread gemacht hat, überhaupt sichtbar wird. Wir haben mal angenommen, dass ein Teil sichtbar wird, nämlich die Adresse des konstruierten `Immutable`-Objekts, aber nicht dessen Inhalt. Das ist ein denkbares Szenario; es kann auch passieren, dass der zweite Thread gar nichts zu sehen bekommen, auch nicht die Adresse des neu erzeugten Objekts.

Aber nehmen wir mal an, die Adresse wird sichtbar und der Rest nicht. Dann sieht der zweite Thread zwar das neu erzeugte `Immutable`-Objekt, aber er sieht den Inhalt des Objekts in seinem Defaultzustand vor der Initialisierung der Felder, d.h. er sieht für das `int`-Feld den Wert `0`. Der erste Thread hat zwar das `Immutable`-Objekt ordnungsgemäß initialisiert, ehe dessen Adresse in der Referenzvariablen `lazyField` abgelegt wurde, und in dem `int`-Feld des `Immutable`-Objekts steht auch der beabsichtigte Initialwert `10000` drin, aber dieser Wert `10000` existiert nur im Cache des ersten Threads und nicht im

Hauptspeicher (oder er steht im Hauptspeicher und der zweite Thread hat seinen Cache nicht aufgefrischt). Wie auch immer die Konstellation genau sein mag, der Effekt ist, dass es für den zweiten Thread so aussieht, als sähe er das Objekt vor seiner Konstruktion. Im weiteren zeitlichen Verlauf kann es dann noch passieren, dass der zweite Thread vielleicht noch einmal auf des Objekt schaut und in der Zwischenzeit Flushes und Refreshes erfolgt sind, so dass der Initialwert `10000` mittlerweile sichtbar geworden ist. Dann sieht es für den zweiten Thread so aus, als habe sich das Immutable-Objekt von seinem Defaultzustand in seinen Initialzustand geändert. Wie gesagt, es sieht nur so aus. Dahinter stehen die mehr oder weniger zufällig passierenden Speichereffekte.

Weil "mehr oder weniger zufällig passierende Speichereffekte" keine verlässlich funktionierenden Programme ergeben, will man genau solche undefinierten Situationen nicht haben. Die beschriebenen Effekte sind in der Regel höchst unerwünscht und deshalb haben wir das `int`-Feld ganz bewußt als `final` deklariert. Dann profitieren wir nämlich von der "*Initialization Safety*"-Garantie des Java Memory Modells. Es garantiert, dass das `final`-Feld erst nach seiner Initialisierung sichtbar gemacht wird und niemals vorher. Das heißt, wenn ein anderer Thread die Adresse des neu konstruierten Objekts zu sehen bekommt, dann sind garantiert alle `final`-Felder des neuen Objekts bereits in ihrer initialisierten Form sichtbar.

Für die Implementierung eines unveränderlichen Typs bedeutet es, dass grundsätzlich alle seine Felder als `final` deklariert sein müssen.

Unser `Immutable`-Typ im Beispiel ist korrekt implementiert: er hat keine verändernden Methoden und alle seine Felder sind als `final` deklariert. Man kann ihn ohne Bedenken als Typ eines Felds verwenden, das mit dem *Racy-Single-Check*-Idiom initialisiert wird. Das gilt aber, wie gesagt, nicht für alle Typen, die von sich behaupten, unveränderlich zu sein. Wenn ein angeblich unveränderlicher Typ non-`final`-Felder hat, dann ist Vorsicht geboten, weil die oben ausführlich beschriebenen Sichtbarkeitsprobleme auftreten können.

Mögliche Missverständnisse

An dieser Stelle ist vielleicht noch ein Hinweis auf mögliche Mißverständnisse angebracht: Es ist zu beachten, dass die "*Initialization Safety*"-Garantie nur für die `final`-Felder eines Objekts gilt. Wenn das Objekt noch weitere Felder hat, die nicht als `final` deklariert sind, dann gibt es für diese non-`final`-Felder keine Garantien. Hier ein Beispiel zur Illustration:

```
public class NoLongerImmutable {
  private final int finalField;
  private       int nonFinalField;
  public NoLongerImmutable(int init) {
    finalField = init;
    nonFinalField = init;
  }
  public String toString() {
    return "["+ finalField + "/" + nonFinalField +"]";
  }
  …
}
```

Die Klasse `NoLongerImmutable` hat ein `final`- und ein non-`final`-Feld. Selbst wenn die Klasse nur lesende Methoden hat, ist sie ist unveränderlicher Typ nicht mehr wirklich brauchbar, zumindest nicht im Zusammenhang mit dem *Racy-Single-Check*-Idiom. Wenn wir sie so verwenden wie zuvor die Klasse `Immutable`, dann gibt es Sichtbarkeitsprobleme für das non-`final`-Feld.

```
public class MyClass {
    private NoLongerImmutable lazyField = null;  // default value
    ...

    public NoLongerImmutable getMyField() {
        if (lazyField == null)
            lazyField = new NoLongerImmutable(10000);

        return lazyField;
    }
    ...
}

public class Test {
  private static MyClass globalRef = new MyClass();

  public static void main(String[] args) {
    Runnable r = new Runnable() {
                 public void run() {
```

```
                    System.out.println(globalRef.getMyField().toString());
                  }
                };
    new Thread(r).start();
    new Thread(r).start();
  }
}
```

Hier könnte der zweite Thread `[10000/10000]` ausgeben, oder aber auch `[10000/0]`, weil das zweite Feld nicht `final` ist und deshalb unklar ist, ob sein Default- oder sein Initialwert sichtbar wird.

Sichtbarkeitsgarantieren für abhängige Objekte

Wie ist das nun, wenn das `final`-Feld in einem unveränderlichen Typ kein Wert von einem primitiven Typ wie `int` ist, sondern eine Referenz auf ein Objekt? Hier ein Beispiel, in dem die Klasse `Immutable` kein `final`-Feld vom Typ `int`, sondern eine `final`-Referenz auf ein Array enthält:

```
public class Immutable {
  private final int[] finalArrayRef;
  public Immutable(int start, int size) {
    finalArrayRef = new int[size];
    for (int i=0;i<size;i++)
       finalArrayRef[i] = start+i;
  }
  public String toString() {
    return "["+ Arrays.toString(finalArrayRef) +"]";
  }
}
```

Zunächst einmal ist klar, dass der zweite Thread die Adresse des Arrays sehen wird (und nicht etwa `null`), weil die Referenzvariable `finalArrayRef` als `final` deklariert ist. Es stellt sich aber die Frage, ob der lesende Thread auch die Elemente in dem Array zu sehen bekommt.

Glücklicherweise gibt das Java Memory Modell in der Tat derartige Garantien für die Array-Elemente. Die Garantie für `final`-Felder bezieht sich nämlich nicht nur auf die `final`-Felder selbst. Für `final`-Felder, die von einem Referenztyp sind, ist garantiert, dass die Referenz *und alle "abhängigen" Objekte* sichtbar gemacht werden.

Die "abhängigen" Objekte sind jene, die von einem `final`-Feld aus per Referenz erreichbar sind, und alle Objekte, die wiederum von dort aus per Referenz erreichbar sind. Gemeint ist also die gesamte transitive Hülle aller erreichbaren Objekte. Das würde in unserem Beispiel alle Array-Elemente einschließen. In dem SMP-Modell für das Java Memory, das wir in [EFF2] beschrieben haben, kann man es sich so vorstellen, als ob am Ende der Konstruktion eines Objekts mit `final`-Feldern ein *partieller* Flush ausgelöst würde, bei dem die `final`-Felder des Objekts *und alle "abhängigen" Objekte* in den Hauptspeicher zurückgeschrieben werden. In jedem Falle ist garaniert, dass alle `final`-Felder eines Objekts und alle von diesen `final`-Feldern aus erreichbaren Objekte anderen Thread sichtbar gemacht sind, ehe die Adresse des Objekts sichtbar wird und die anderen Threads auf die `final`-Felder zugreifen können.

Unser `Immutable`-Typ mit dem Array wäre also korrekt implementiert: er hat keine verändernden Methoden, alle seine Felder sind als `final` deklariert und die Initialisation-Safety-Garantie sorgt dafür, dass auch die Array-Elemente sichtbar werden. Man kann ihn ohne Bedenken als Typ eines Felds verwenden, das mit dem *Racy-Single-Check*-Idiom initialisiert wird.

Wie ist das nun, wenn der unveränderliche Typ eine Referenz auf ein Array mit Referenzen (statt primitiven Elementen) enthält, also kein `int[]`, sondern ein `Integer[]`? Die Initialisation-Safety-Garantie sorgt dann dafür, dass auch die von den Array-Elementen referenzierten Objekte und deren Inhalte sichtbar werden. Damit der äußere Typ `Immutable` unververänderlich ist, müssen die Array-Elemente natürlich wiederum von einem korrekt implementierten unveränderlichen Typ (wie z.B. `Integer` oder `String`) sein.

Was wir hier am Beispiel einer `final`-Referenz auf ein Array erläutert haben, gilt analog für `final`-Referenzen auf Objekte. Die Initialisation-Safety-Garantie sorgt dafür, dass die gesamte transitive Hülle aller erreichbaren Objekte am Ende der Konstruktion sichtbar wird. Dabei müssen in dem unveränderlichen Typ alle abhängigen Objekte ihrerseits wieder unveränderlich sein.

Mögliche Mißverständnisse

Mit der Initialisation-Safety-Garantie muss man übrigens manchmal etwas vorsichtig umgehen. Im lesenden Thread ist nur gewährleistet, dass er sich beim ersten Zugriff auf das Objekt vom Typ `Immutable` alle Werte der `final`-Felder aus dem Hauptspeicher holt. Dabei holt er sich auch die Werte aller abhängigen Objekte; er macht also einen partiellen Refresh seines Arbeitsspeichers aus dem Hauptspeicher. Danach muss er aber keinen Refresh mehr machen. Das ist auch sinnvoll so. Für die `final`-Referenzvariable braucht er sowieso keinen Refresh mehr, weil die Adresse des Arrays konstant ist und sich nicht mehr ändern kann. In einem unveränderlichen Typ ändern sich auch die Inhalte des Arrays und seiner Elemente nicht; sonst wäre der Typ veränderlich.

Die Unveränderbarkeit des Arrays ist aber durch nichts gewährleistet, weil die `final`-Deklaration der Array-Referenz nur sagt, dass die Adresse des Arrays konstant ist, nicht aber der Inhalt des Array selbst. Es wäre also prinzipiell möglich, dass sich das Array und seine Elemente ändern. Für die Sichtbarkeit dieser späteren Änderungen gibt das Java Memory Modell keine Garantien. Sehen wir uns das im Beispiel mal an:

```
public class NoLongerImmutable {
  private final int[] finalArrayRef;
  public NoLongerImmutable(int start, int size) {
    finalArrayRef = new int[size];
    for (int i=0;i<size;i++)
       finalArrayRef[i] = start+i;
  }
  public String toString() {
    return "["+ Arrays.toString(finalArrayRef) +"]";
  }
  public void update(int idx, int val) {
    finalArrayRef[idx] = val;
  }
}
```

Hier ist nun nicht garantiert, dass lesende Threads die Modifikation an den Array-Elementen zu sehen bekommt, die andere Threads mit Hilfe von `update()` nach der Konstruktion gemacht haben. Lesende Threads müssen nur ein einziges Mal einen Refresh ihres Arbeitsspeichers machen, nämlich beim ersten Zugriff auf das `final`-Feld und alle seine abhängigen Objekte. Veränderungen an den abhängigen Objekten, die später noch passieren, müssen nicht - aber könnten (zum Beispiel durch weitere Synchronisationspunkte an ganz anderen Stellen) - sichtbar werden. Wenn sie garantiert sichtbar werden sollen, dann muss man mit anderen Mitteln (zum Beispiel explizite Synchronisation) für die Sichtbarkeit sorgen.
Man beachte, dass dieses Missverständnis bei unveränderlichen Typen nicht auftreten kann, weil in ein einem unveränderlichen Typ alle abhängigen Objekte ebenfalls unveränderlich sind. Es genügt also der Refresh aller abhängigen Objekte beim allerersten Zugriff, weil sich danach nichts mehr an den abhängigen Objekten ändert.

Unterschiede zu volatile

Man beachte, dass die Speichereffekte bei `final`-Feldern ganz anders sind als bei `volatile`-Referenzvariablen. Bei `volatile`-Variablen löst jeder schreibende oder lesende Zugriff einen Flush bzw. Refresh des gesamten Arbeitsspeichers aus (siehe [EFF4]). Bei `final`-Referenzvariablen löst nur das Ende des Konstruktors einen partiellen Flush und nur der erstmalige lesende Zugriff in jedem Thread einen partiellen Refresh aus. `volatile` ist also "teuer" als `final`, weil mehr Memory Barriers ausgelöst werden, und `volatile` hat keine Garantie für abhängige Objekte.

final Variablen vs. final Felder

Noch ein Hinweis auf mögliche Missverständnisse: "`final`" ist nicht gleich "`final`". Die Garantien des Java Memory Modells im Zusammenhang mit `final` beziehen sich nur auf `final`-Felder von Objekten, nicht auf `final`-Variablen. Die Speichereffekte für `final` werden nicht ohne Grund als "*Initialization Safety*"-Garantie bezeichnet. Die Garantie besagt lediglich, dass `final`-Felder eines Objekts stets in ihrer fertig initialisierten Form sichtbar werden, niemals vorher. Für andere Verwendungen von `final` (zum Beispiel als Parameter und lokalen Variablen von Methoden) gibt es keine Garantien.

Schauen wir uns zur Illustration einen Fall an, in dem zwar `final` verwendet wird, seine Verwendung aber nichts mit *Initialization Safety* zu tun hat.

```
public class ActiveObject{
  public ActiveObject(long times) throws InterruptedException {
    final long[] argumentAndResult = new long[2];
    argumentAndResult[0] = times;
```

```
    Runnable r = new Runnable() {
      public void run() {
        long start = System.nanoTime();
        for (int i=0;i<argumentAndResult[0];i++)
           System.out.print("*");
        argumentAndResult[1] = System.nanoTime()-start;
      }
    };
    Thread t = new Thread(r,"printer");
    t.start();
    t.join();
    System.out.println(argumentAndResult[1]);
  }
}
```

Das Beispiel zeigt ein gängiges Idiom für die Parametrisierung von Runnables: da die `run()`-Methode keine Argumente haben darf, implementiert man Runnables gerne als anonyme innere Klassen und gibt ihnen Zugriff auf `final`-Variablen des umgebenden Kontextes. Auf diese Art erreicht man eine indirekte Parametrisierung der `run()`-Methode.
In diesem Beispiel haben der main-Thread und der printer-Thread gemeinsam und konkurrierend Zugriff auf das `final`-Array `argumentAndResult`. Das erste Array-Element ist der Parameter für den printer-Thread; in dem zweiten Array-Element legt der printer-Thread das Ergebnis ab, d.h. die Zeit, die für das Ausdrucken von `times` vielen `"*"` gebraucht wurde.

Hier hat die Verwendung von `final` überhaupt nichts mit den oben besprochenen Speichereffekten für `final`-Felder zu tun, denn hier brauchen wir gar keine Garantien im Zusammenhang mit `final`-Variablen.

Die benötigten Speichererffekte werden hier durch Thread-Start und Thread-Ende geliefert, nicht durch die `final`-Deklaration der Array-Referenz. Der Start des printer-Threads löst einen Flush im main-Thread und einen Refresh im printer-Thread aus. Das heißt, der neu gestartete printer-Thread kann sehen, was der ihn startende main-Thread gemacht hat. Analog beim Thread-Ende: der main-Thread wartet mit `Thread.join()` auf das Ende des printer-Threads und kann dann alles sehen, was der printer-Thread gemacht hat.

Und eine letzter Hinweis auf mögliche Mißverständnisse:
Für die Thread-Ende-Garantie ist übrigens wichtig, dass sich der eine Thread über das Ende des anderen Threads aktiv informiert hat (per `join()` oder `isAlive()`). Wenn der printer-Thread nur zufällig schon gerade fertig ist, dann ist nicht garantiert, dass der main-Thread sieht, was der printer-Thread gemacht hat, weil der Refresh im main-Thread erst durch `join()` oder `isAlive()` ausgelöst wird.

Zusammenfassung

In diesem Beitrag haben wir uns die "*Initialization Safety*"-Garantie für `final`-Felder von einem Referenztyp angesehen. Die Garantie besagt, dass `final`-Felder eines Objekts einem andern Thread stets in ihrer fertig initialisierten Form sichtbar werden, niemals vorher. Dabei werden auch alle abhängigen Objekte in ihrer fertig initialisierten Form sichtbar.

Diese Garantie gibt es nur für `final`-Felder und nicht für `final`-Variablen. Die "*Initialization Safety*"-Garantie wird für die Implementierung von unveränderlichen Typen gebraucht: in einem unveränderlichen Typ müssen alle Felder als `final` deklariert sein und alle abhängigen Objekte müssen ihrerseits unveränderlich sein.

Im nächsten Beitrag sehen wir uns, was man trotz Verwendung von `final` bei der Implementierung von unveränderlichen Typen falsch machen kann. Die "*Initialization Safety*"-Garantie gilt nämlich nur, wenn der sogenannte *Object Escape* verhindert wird. Das schauen wir uns im nächsten Beitrag genauer an.

Verweise

/EFF1/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/EFF2/ JMM - Überblick über das Java Memory Modell (JMM)
Klaus Kreft & Angelika Langer
JavaMagazin 8.08

/EFF3/ JMM - Die Kosten der Synchronisation
Klaus Kreft & Angelika Langer
JavaMagazin 9.08

/EFF4/ JMM - Details zu volatile
Klaus Kreft & Angelika Langer
JavaMagazin 10.08

/EFF5/ JMM - Double-Check Locking
Klaus Kreft & Angelika Langer
JavaMagazin 11.08

Java Memory Modell

Teil 8: Die Initialisation-Safety-Garantie (für final-Felder von primitivem Typ)

Klaus Kreft & Angelika Langer

Wir haben in einem der vorangegangenen Beiträge [EFF2] erwähnt, dass es besondere Garantieren für die Sichtbarkeits- und Speichereffekte im Zusammenhang mit `final` gibt. Diese Garantien wollen wir uns in diesem Beitrag genauer ansehen.

Racy-Single-Check mit einem primitiven Typ (außer `long` und `double`)

Als wir im letzten Artikel [EFF5] die Verwendung von `volatile` am Beispiel des Double-Check-Idioms besprochen haben, haben wir uns auch das sogenannte *Racy-Single-Check*-Idiom angesehen. Bei diesem Idiom wird weder Synchronisation noch `volatile` genutzt. Entsprehend ist seine Anwendung sehr eingeschränkt, aber es gibt Fälle, in denen es korrekt und unproblematisch ist, nämlich wenn das zu initialisierende Feld einen kostanten Wert zugewiesen bekommt und von einem primitiven Typ (außer `long` und `double`).

Hier noch einmal unser Beispiel des *Racy-Single-Check*-Idioms aus unserem letzten Artikel:

```
public class MyClass {
    private int lazyField;
    ...

    public int getMyField() {
        if (lazyField == 0)
            lazyField = 10000;

        return lazyField;
    }

    ...
}
```

Für die korrekte Verwendung des *Racy-Single-Check*-Idiom im Beispiel oben sind folgende Überlegungen wichtig:

– *Der Zugriff auf das fragliche Feld muss atomar sein.* Sonst wäre der Zugriff unterbrechbar und das Feld könnte (aus Sicht des lesenden Threads) sinnlose Werte enthalten. Die Ununterbrechbarkeit des Zugriffs auf primitive Typen (außer `long` und `double`) ist in Java garantiert.

– *Das Feld `lazyField` ändert sich nach der Initialisierung nicht mehr*. Man würde es gerne als `final` deklarieren, um diese Eigenschaft sichtbar zu machen, aber das geht nicht, weil die Initialisierung nicht im Konstruktor, sondern "lazy" in der Methode `getMyField()` erfolgt. Semantisch ist das Feld aber `final`, sonst wäre die Verwendung des *Racy-Single-Check*-Idiom falsch. Es könnte sonst passieren, das zwei Thread gleichzeitig "`lazyField == 0`" sehen und beide nacheinander den Initialwert zuweisen. Wenn zwischendrin ein dritter Thread das Feld bereits verändert hätte, dann ginge diese Veränderung verloren. Es ist also wichtig, dass der Inhalt des Feldes unveränderlich ist.

Ein bekanntes Anwendungsbeispiel für das *Racy-Single-Check*-Idiom ist der Hashcodewert in der Klasse `java.lang.String`. Er wird genau mit dieser Technik *lazy* initialisiert.

Racy-Single-Check mit einem Referenztyp

Wie funktioniert nun das *Racy-Single-Check*-Idiom mit einem Feld, dass von einem Referenztyp ist?

Hier ist ein Beispiel mit einer Referenz auf einen Integer vom Typ `java.lang.Integer`:

```
public class MyClass {
    private Integer lazyField = null;  // default value
    ...

    public Integer getMyField() {
        if (lazyField == null)
            lazyField = new Integer(10000);

        return lazyField;
    }

    ...
}
```

Hier gehen nun drei Überlegungen ein, die die korrekte Verwendung des *Racy-Single-Check*-Idiom garantieren:

– *Der Zugriff auf das fragliche Feld muss atomar sein.* Das ist auch hier der Fall; die Ununterbrechbarkeit des Zugriffs auf Adressen, also Variablen von einem Referenztyp, ist in Java garantiert.

– *Die Referenz `lazyField` ändert sich nach der Initialisierung nicht mehr.* Auch hier würde man das Feld gerne als `final` deklarieren, aber es geht wieder nicht wegen der "lazy" Initialisierung. Die `final`-Deklaration der Referenz bezöge sich ohnehin nur auf die Adresse des referenzierten Objekts und nicht auf das Objekt und seine Inhalte. Ein unveränderliche Adresse reicht aber für das *Racy-Single-Check*-Idiom nicht aus; es muss eine dritte Voraussetzungen erfüllt sein.

– *Das referenzierte Objekt und seine Inhalte ändern sich nach der Initialisierung nicht mehr.* Wenn das Objekt veränderlich wäre, dann wäre die Verwendung des *Racy-Single-Check*-Idiom falsch, denn das Idiom lässt Mehrfach-Initialisierung zu. Es könnte passieren, das zwei Thread gleichzeitig "`lazyField == null`" sehen und beide nacheinander den Initialwert zuweisen. Eine Veränderung des referenzierten Objekts, die zwischendrin ein dritter Thread vorgenommen hätte, ginge verloren. Es ist also wichtig, dass nicht nur die Referenz, sondern auch das referenzierte Objekt unveränderlich ist.

Das bedeutet, dass das *Racy-Single-Check*-Idiom nur sinnvoll ist, wenn das Feld eine unveränderliche Referenz auf einen unveränderlichen (immutable) Typ ist.

Anforderungen an unveränderliche Typen

Nun ist der Typ `java.lang.Integer` bekanntlich ein unveränderlicher Typ und er ist auch so implementiert, dass das *Racy-Single-Check*-Idiom mit einem `Integer` funktioniert, aber das gilt nicht für jeden Typ, der von sich behauptet unveränderlich zu sein. Es genügt nämlich nicht, dass es in einem unveränderlichen Typ keine modifizierenden Methoden gibt.

Ein unveränderlicher Typ muss auch für die Sichtbarkeit seiner Inhalte sorgen, das heißt, er muss sicher stellen, dass die unveränderlichen Inhalte des Objekts nach der Konstruktion allen benutzenden Threads sichtbar werden. Denn was nützt es uns, wenn die Threads zwar die Adresse des `Integers` nach der Lazy-Initialisierung sehen, aber nicht die Inhalte des `Integers`?

Um für die Sichtbarkeit zu sorgen, braucht man bei der Implementierung eines unveränderlichen Typs die sogenannte "*Initialization-Safety*"-Garantie des Java Memory Modells. Das ist eine Regel für die Sichtbarkeit der Inhalte der `final`-Felder von Objekten.

Sehen wir uns also die "*Initialization-Safety*"-Garantie des Java Memory Modells mal genauer an.

Speichereffekte im Zusammenhang mit final-Feldern

Es geht bei der "*Initialization Safety*"-Garantie des Java Memory Modells darum, dass stets die Initialwerte von `final`-Feldern und niemals die Defaultwerte sichtbar sind. Wenn also ein Thread ein Objekt mit `final`-Feldern zu sehen bekommt, weil er die Adresse des Objekts sehen kann, dann sieht er die `final`-Felder des Objekts stets im Initialzustand *nach* der Konstruktion und nie im Defaultzustand *vor* der Konstruktion.

Was heißt das genau? Betrachten wir ein erstes einfaches Beispiel einer Klasse mit einem `final`-Feld:

```
public class Immutable {
  private final int field;
  public Immutable (int init) {
    field = init;
  }
  public String toString() {
    return "["+ field +"]";
  }
}
```

Die Klasse ist dem unveränderlichen Typ `java.lang.Integer` nachempfunden. Sie könnte beispielsweise als Typ des `lazyField` in unserem *Racy-Single-Check*-Idiom vorkommen.

```
public class MyClass {
    private Immutable lazyField = null;  // default value
    ...

    public Immutable getMyField() {
        if (lazyField == null)
            lazyField = new Immutable(10000);
```

```
        return lazyField;
    }
    ...
}
```

Nehmen wir nun einmal an, dass zwei Threads gleichzeitig auf ein Objekt vom Typ `Immutable` zugreifen.

```
public class Test {
  private static MyClass globalRef = new MyClass();

  public static void main(String[] args) {
    Runnable r = new Runnable() {
                 public void run() {
                   System.out.println(globalRef.getMyField().toString());
                 }
               };
    new Thread(r).start();
    new Thread(r).start();
  }
}
```

Beide Threads holen sich über die Methode `getMyField()` der Klasse `MyClass` die Referenz auf das `Immutable`-Feld des `MyClass`-Objekts und rufen anschließend auf dem `Immutable`-Feld die `toString()`-Methode der Klasse `Immutable` auf. Dann könnte es so auskommen, dass der eine Thread in der Methode `getMyField()` die Referenz `lazyField` auf das `Immutable`-Feld noch als `null` vorfindet, weil noch niemand die *lazy*-Initialisierung für das Feld gemacht hat. Der andere Thread findet möglicherweise schon eine von `null` verschiedene Referenz vor und greift über diese Referenz auf das `Immutable`-Objekt zu und ruft dessen `toString()`-Methode auf. In dieser Situation stellt sich die Frage, in welchem Zustand der zweite Thread den Inhalt des referenzierten `Immutable`-Objekts zu sehen bekommt.

Wenn das Feld `field` in der Klasse `Immutable` nicht `final` wäre, dann könnte es in dieser Situation passieren, dass der zweite Thread das referenzierte `Immutable`-Objekt sieht und das Feld `field` in diesem Objekt entweder den Wert `0` oder `10000` hat. Welchen der beiden Werte das Feld hat, ist undefiniert. Es kann sogar passieren, dass der zweite Thread, wenn er mehrmals liest, erst den Wert `0` und später den Wert `10000` zu sehen bekommt. Das sieht dann so aus, als sei das unveränderliche Objekt vom Typ `Immutable` gar nicht unveränderlich, weil sich sein Inhalt augenscheinlich ändert. Wie kann das sein?

Solche Effekte können bei fehlender `final`-Deklaration entstehen, weil es ohne die `final`-Deklaration keinerlei Garantien für die Sichtbarkeit des Feldes `field` gibt.

Sichtbarkeitsprobleme im Detail

Sichtbarkeitsprobleme sind generell ein bisschen schwierig zu verstehen, weil sie immer Probleme zwischen Threads sind und nie innerhalb eines Threads entstehen. Ein Sichtbarkeitsproblem gibt es nur dann, wenn ein Thread beobachtet, was ein anderer Thread im Speicher macht. In unserem Beispiel ist es so, dass der erste Thread eine `null`-Referenz vorfindet und dann die *lazy*-Initialisierung macht, also das `Immutable`-Objekt konstruiert und dessen Adresse in der Referenzvariablen `lazyField` ablegt. So sieht es innerhalb des ersten Threads aus und es entspricht unserer Intuition: erst wird das Objekt alloziert (dann ist es in einem Defaultzustand), dann konstruiert (dann ist es in seinem Initialzustand) und danach wird seine Adresse dem Feld `lazyField` zugewiesen.

Für den zweiten Thread sieht die Sache u.U. fundamental anders aus. In der oben geschilderten Situation haben wir angenommen, dass das `int`-Feld in der Klasse `Immutable` nicht `final` ist und es daher keine Sichtbarkeitsgarantien gibt. Dann ist undefiniert, welche von den Speichermodifikationen, die der erste Thread gemacht hat, überhaupt sichtbar wird. Wir haben mal angenommen, dass ein Teil sichtbar wird, nämlich die Adresse des konstruierten `Immutable`-Objekts, aber nicht dessen Inhalt. Das ist ein denkbares Szenario; es kann auch passieren, dass der zweite Thread gar nichts zu sehen bekommen, auch nicht die Adresse des neu erzeugten Objekts.

Aber nehmen wir mal an, die Adresse wird sichtbar und der Rest nicht. Dann sieht der zweite Thread zwar das neu erzeugte `Immutable`-Objekt, aber er sieht den Inhalt des Objekts in seinem Defaultzustand vor der Initialisierung der Felder, d.h. er sieht für das `int`-Feld den Wert `0`. Der erste Thread hat zwar das `Immutable`-Objekt ordnungsgemäß initialisiert, ehe dessen Adresse in der Referenzvariablen `lazyField` abgelegt wurde, und in dem `int`-Feld des `Immutable`-Objekts steht auch der beabsichtigte Initialwert `10000` drin, aber dieser Wert `10000` existiert nur im Cache des ersten Threads und nicht im

Hauptspeicher (oder er steht im Hauptspeicher und der zweite Thread hat seinen Cache nicht aufgefrischt). Wie auch immer die Konstellation genau sein mag, der Effekt ist, dass es für den zweiten Thread so aussieht, als sähe er das Objekt vor seiner Konstruktion. Im weiteren zeitlichen Verlauf kann es dann noch passieren, dass der zweite Thread vielleicht noch einmal auf des Objekt schaut und in der Zwischenzeit Flushes und Refreshes erfolgt sind, so dass der Initialwert `10000` mittlerweile sichtbar geworden ist. Dann sieht es für den zweiten Thread so aus, als habe sich das Immutable-Objekt von seinem Defaultzustand in seinen Initialzustand geändert. Wie gesagt, es sieht nur so aus. Dahinter stehen die mehr oder weniger zufällig passierenden Speichereffekte.

Weil "mehr oder weniger zufällig passierende Speichereffekte" keine verlässlich funktionierenden Programme ergeben, will man genau solche undefinierten Situationen nicht haben. Die beschriebenen Effekte sind in der Regel höchst unerwünscht und deshalb haben wir das `int`-Feld ganz bewußt als `final` deklariert. Dann profitieren wir nämlich von der "*Initialization Safety*"-Garantie des Java Memory Modells. Es garantiert, dass das `final`-Feld erst nach seiner Initialisierung sichtbar gemacht wird und niemals vorher. Das heißt, wenn ein anderer Thread die Adresse des neu konstruierten Objekts zu sehen bekommt, dann sind garantiert alle `final`-Felder des neuen Objekts bereits in ihrer initialisierten Form sichtbar.

Für die Implementierung eines unveränderlichen Typs bedeutet es, dass grundsätzlich alle seine Felder als `final` deklariert sein müssen.

Unser `Immutable`-Typ im Beispiel ist korrekt implementiert: er hat keine verändernden Methoden und alle seine Felder sind als `final` deklariert. Man kann ihn ohne Bedenken als Typ eines Felds verwenden, das mit dem *Racy-Single-Check*-Idiom initialisiert wird. Das gilt aber, wie gesagt, nicht für alle Typen, die von sich behaupten, unveränderlich zu sein. Wenn ein angeblich unveränderlicher Typ non-`final`-Felder hat, dann ist Vorsicht geboten, weil die oben ausführlich beschriebenen Sichtbarkeitsprobleme auftreten können.

Mögliche Missverständnisse

An dieser Stelle ist vielleicht noch ein Hinweis auf mögliche Mißverständnisse angebracht: Es ist zu beachten, dass die "*Initialization Safety*"-Garantie nur für die `final`-Felder eines Objekts gilt. Wenn das Objekt noch weitere Felder hat, die nicht als `final` deklariert sind, dann gibt es für diese non-`final`-Felder keine Garantien. Hier ein Beispiel zur Illustration:

```
public class NoLongerImmutable {
  private final int finalField;
  private       int nonFinalField;
  public NoLongerImmutable(int init) {
    finalField = init;
    nonFinalField = init;
  }
  public String toString() {
    return "["+ finalField + "/" + nonFinalField +"]";
  }
  …
}
```

Die Klasse `NoLongerImmutable` hat ein `final`- und ein non-`final`-Feld. Selbst wenn die Klasse nur lesende Methoden hat, ist sie ist unveränderlicher Typ nicht mehr wirklich brauchbar, zumindest nicht im Zusammenhang mit dem *Racy-Single-Check*-Idiom. Wenn wir sie so verwenden wie zuvor die Klasse `Immutable`, dann gibt es Sichtbarkeitsprobleme für das non-`final`-Feld.

```
public class MyClass {
    private NoLongerImmutable lazyField = null;  // default value
    ...

    public NoLongerImmutable getMyField() {
        if (lazyField == null)
            lazyField = new NoLongerImmutable(10000);

        return lazyField;
    }
    ...
}

public class Test {
  private static MyClass globalRef = new MyClass();

  public static void main(String[] args) {
    Runnable r = new Runnable() {
                 public void run() {
```

```
                    System.out.println(globalRef.getMyField().toString());
                  }
                };
      new Thread(r).start();
      new Thread(r).start();
    }
  }
```

Hier könnte der zweite Thread `[10000/10000]` ausgeben, oder aber auch `[10000/0]`, weil das zweite Feld nicht `final` ist und deshalb unklar ist, ob sein Default- oder sein Initialwert sichtbar wird.

Zusammenfassung

In diesem Beitrag haben wir uns die "*Initialization Safety*"-Garantie für `final`-Felder angesehen. Die Garantie besagt, dass `final`-Felder eines Objekts einem andern Thread stets in ihrer fertig initialisierten Form sichtbar werden, niemals vorher. Für non-`final`-Felder gibt es keine Garantien.

Die "*Initialization Safety*"-Garantie wird für die Implementierung von unveränderlichen Typen gebraucht: in einem unveränderlichen Typ müssen alle Felder als `final` deklariert sein, sonst kann es Sichtbarkeitsprobleme geben, beispielsweise wenn der unveränderliche Typ für eine *lazy*-Initialisierung mit dem *Racy-Single-Check*-Idiom verwendet wird.

Wir haben die gesamte Diskussion auf `final`-Felder von primitivem Typ beschränkt. Wie ist das, wenn das `final`-Feld von einem Referenztyp ist? Das sehen wir uns beim nächsten Mal an.

Verweise

/EFF1/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/EFF2/ JMM - Überblick über das Java Memory Modell (JMM)
Klaus Kreft & Angelika Langer
JavaMagazin 8.08

/EFF3/ JMM - Die Kosten der Synchronisation
Klaus Kreft & Angelika Langer
JavaMagazin 9.08

/EFF4/ JMM - Details zu volatile
Klaus Kreft & Angelika Langer
JavaMagazin 10.08

/EFF5/ JMM - Double-Check Locking
Klaus Kreft & Angelika Langer
JavaMagazin 11.08

Java Memory Modell

Teil 9: Die Initialisation-Safety-Garantie
(für final-Felder von einem Referenztyp)

Klaus Kreft & Angelika Langer

Wir haben im letzten Beitrag [EFF6] die Initialisation-Safety-Garantien des Java Memory Modells für `final`-Felder besprochen. Dabei geht es um die Garantie, dass alle `final`-Felder eines Objekts stets in ihrem Zustand nach der Konstruktion und nie in ihrem Defaultzustand vor der Konstruktion sichtbar werden. Wir haben dabei ein `final`-Feld vom Typ `int`, also einem primitiven Typ, betrachtet. Wie sehen die Garantien für `final`-Felder von einem Referenztyp aus? Werden auch die referenzierten Objekte sichtbar oder nur die Referenz selbst? Das wollen wir uns in diesem Beitrag ansehen.

Ausgangspunkt unserer Diskussion war die Lazy-Initialisierung eines Feldes mit Hilfe des *Racy-Single-Check*-Idioms (siehe [EFF5]). Bei diesem Idiom wird weder Synchronisation noch `volatile` genutzt. Hier ist ein Beispiel mit einer Referenz auf einen Integer vom Typ `java.lang.Integer`:

```
public class MyClass {
    private Integer lazyField = null;  // default value
    ...

    public Integer getMyField() {
        if (lazyField == null)
            lazyField = new Integer(10000);

        return lazyField;
    }

    ...
}
```

Für die korrekte Verwendung des *Racy-Single-Check*-Idiom müssen folgende Voraussetzungen gegeben sein:

- *Der Zugriff auf das fragliche Feld muss atomar sein.* Das ist hier der Fall, weil das Feld von einem Referenztyp ist.

- *Die Referenz `lazyField` ändert sich nach der Initialisierung nicht mehr*. Um das sicherzustellen, würde man das Feld gerne als `final` deklarieren, aber es geht nicht wegen der "lazy" Initialisierung. Die `final`-Deklaration der Referenz bezöge sich ohnehin nur auf die Adresse des referenzierten Objekts und nicht auf das Objekt und seine Inhalte. Deshalb muss eine dritte Voraussetzungen erfüllt sein.

- *Das referenzierte Objekt und seine Inhalte ändern sich nach der Initialisierung nicht mehr.* Wenn das Objekt veränderlich wäre, dann wäre die Verwendung des *Racy-Single-Check*-Idiom falsch, denn das Idiom lässt Mehrfach-Initialisierung zu. Es könnte passieren, das zwei Thread gleichzeitig "`lazyField == null`" sehen und beide nacheinander den Initialwert zuweisen. Eine Veränderung des referenzierten Objekts, die zwischendrin ein dritter Thread vorgenommen hätte, ginge verloren. Es ist also wichtig, dass nicht nur die Referenz, sondern auch das referenzierte Objekt unveränderlich ist.

Das bedeutet, dass das *Racy-Single-Check*-Idiom nur sinnvoll ist, wenn das betreffende Feld eine unveränderliche Referenz auf einen unveränderlichen (immutable) Typ ist. Deshalb haben wir über unveränderliche Typen gesprochen und festgestellt, dass alle Felder eines unveränderlichen Typs als `final` deklariert sein müssen; sonst gibt es Sichtbarkeitsprobleme. Die Sichtbarkeitsprobleme haben wir beim letzten Mal ausführlich diskutiert (siehe [EFF5]).

Anforderungen an unveränderliche Typen

Es genügt nicht, dass es in einem unveränderlichen Typ keine modifizierenden Methoden gibt. Ein unveränderlicher Typ muss auch für die Sichtbarkeit seiner Inhalte sorgen, das heißt, er muss sicherstellen, dass die unveränderlichen Inhalte des Objekts nach der Konstruktion allen benutzenden Threads sichtbar werden. Um für die Sichtbarkeit zu sorgen, braucht man bei der Implementierung eines unveränderlichen Typs die sogenannte "*Initialization-Safety*"-Garantie des Java Memory Modells.

Bei der "*Initialization Safety*"-Garantie des Java Memory Modells geht es darum, dass stets die Initialwerte von `final`-Feldern und niemals die Defaultwerte sichtbar sind. Wenn also ein Thread ein Objekt mit `final`-Feldern zu sehen bekommt, weil er die Adresse des Objekts sehen kann, dann sieht er die `final`-Felder des Objekts stets im Initialzustand *nach* der Konstruktion und nie im Defaultzustand *vor* der Konstruktion.

Wir haben dazu ein Beispiel mit einer Klasse mit einem `final`-Feld betrachtet:

```
public class Immutable {
  private final int field;
  public Immutable (int init) {
```

```
    field = init;
  }
  public String toString() {
    return "["+ field +"]";
  }
}
```

Der unveränderlichen Typ `Immutable` wird für ein Feld verwendet, das mit dem *Racy-Single-Check*-Idiom "lazy" initialisiert wird:

```
public class MyClass {
    private Immutable lazyField = null;  // default value
    ...

    public Immutable getMyField() {
        if (lazyField == null)
            lazyField = new Immutable(10000);

        return lazyField;
    }
    ...
}
```

Dann nehmen wir an, dass zwei Threads gleichzeitig auf ein Objekt vom Typ `Immutable` zugreifen:

```
public class Test {
  private static MyClass globalRef = new MyClass();

  public static void main(String[] args) {
    Runnable r = new Runnable() {
                 public void run() {
                   System.out.println(globalRef.getMyField().toString());
                 }
               };
    new Thread(r).start();
    new Thread(r).start();
  }
}
```

Beide Threads holen sich über die Methode `getMyField()` der Klasse `MyClass` die Referenz auf das `Immutable`-Feld des `MyClass`-Objekts und rufen anschließend auf dem `Immutable`-Feld die `toString()`-Methode der Klasse `Immutable` auf. Dann könnte es so auskommen, dass der eine Thread in der Methode `getMyField()` die Referenz `lazyField` auf das `Immutable`-Feld noch als `null` vorfindet, weil noch niemand die *lazy*-Initialisierung für das Feld gemacht hat. Der andere Thread findet möglicherweise schon eine von `null` verschiedene Referenz vor und greift über diese Referenz auf das `Immutable`-Objekt zu und ruft dessen `toString()`-Methode auf. In dieser Situation stellt sich die Frage, in welchem Zustand der zweite Thread den Inhalt des referenzierten `Immutable`-Objekts zu sehen bekommt.

Da das `int`-Feld `field` in der Klasse `Immutable` als `final` deklariert ist, garantiert die Initialisation-Safety-Garantie, dass der zweite Thread das `Immutable`-Objekt in seinem fertig initialisierten Zustand zu sehen bekommt. Der Wert des `int`-Feld `field` ist `10000` und nicht etwa `0`, wie es bei fehlender `final`-Deklaration der Fall sein könnte.

Sichtbarkeitsgarantieren für abhängige Objekte

Wie ist das nun, wenn das `final`-Feld in einem unveränderlichen Typ kein Wert von einem primitiven Typ wie `int` ist, sondern eine Referenz auf ein Objekt? Hier ein Beispiel, in dem die Klasse `Immutable` kein `final`-Feld vom Typ `int`, sondern eine `final`-Referenz auf ein Array enthält:

```
public class Immutable {
  private final int[] finalArrayRef;
  public Immutable(int start, int size) {
    finalArrayRef = new int[size];
    for (int i=0;i<size;i++)
       finalArrayRef[i] = start+i;
  }
  public String toString() {
    return "["+ Arrays.toString(finalArrayRef) +"]";
  }
}
```

Zunächst einmal ist klar, dass der zweite Thread die Adresse des Arrays sehen wird (und nicht etwa `null`), weil die Referenzvariable `finalArrayRef` als `final` deklariert ist. Es stellt sich aber die Frage, ob der lesende Thread auch die Elemente in dem Array zu sehen bekommt.

Glücklicherweise gibt das Java Memory Modell in der Tat derartige Garantien für die Array-Elemente. Die Garantie für `final`-Felder bezieht sich nämlich nicht nur auf die `final`-Felder selbst. Für `final`-Felder, die von einem Referenztyp sind, ist garantiert, dass die Referenz *und alle "abhängigen" Objekte* sichtbar gemacht werden.

Die "abhängigen" Objekte sind jene, die von einem `final`-Feld aus per Referenz erreichbar sind, und alle Objekte, die wiederum von dort aus per Referenz erreichbar sind. Gemeint ist also die gesamte transitive Hülle aller erreichbaren Objekte. Das würde in unserem Beispiel alle Array-Elemente einschließen. In dem SMP-Modell für das Java Memory, das wir in [EFF2] beschrieben haben, kann man es sich so vorstellen, als ob am Ende der Konstruktion eines Objekts mit `final`-Feldern ein *partieller* Flush ausgelöst würde, bei dem die `final`-Felder des Objekts *und alle "abhängigen" Objekte* in den Hauptspeicher zurückgeschrieben werden. In jedem Falle ist garaniert, dass alle `final`-Felder eines Objekts und alle von diesen `final`-Feldern aus erreichbaren Objekte anderen Thread sichtbar gemacht sind, ehe die Adresse des Objekts sichtbar wird und die anderen Threads auf die `final`-Felder zugreifen können.

Unser `Immutable`-Typ mit dem Array wäre also korrekt implementiert: er hat keine verändernden Methoden, alle seine Felder sind als `final` deklariert und die Initialisation-Safety-Garantie sorgt dafür, dass auch die Array-Elemente sichtbar werden. Man kann ihn ohne Bedenken als Typ eines Felds verwenden, das mit dem *Racy-Single-Check*-Idiom initialisiert wird.

Wie ist das nun, wenn der unveränderliche Typ eine Referenz auf ein Array mit Referenzen (statt primitiven Elementen) enthält, also kein `int[]`, sondern ein `Integer[]`? Die Initialisation-Safety-Garantie sorgt dann dafür, dass auch die von den Array-Elementen referenzierten Objekte und deren Inhalte sichtbar werden. Damit der äußere Typ `Immutable` unververänderlich ist, müssen die Array-Elemente natürlich wiederum von einem korrekt implementierten unveränderlichen Typ (wie z.B. `Integer` oder `String`) sein.

Was wir hier am Beispiel einer `final`-Referenz auf ein Array erläutert haben, gilt analog für `final`-Referenzen auf Objekte. Die Initialisation-Safety-Garantie sorgt dafür, dass die gesamte transitive Hülle aller erreichbaren Objekte am Ende der Konstruktion sichtbar wird.

Mögliche Mißverständnisse

Mit der Initialisation-Safety-Garantie muss man übrigens manchmal etwas vorsichtig umgehen. Im lesenden Thread ist nur gewährleistet, dass er sich beim ersten Zugriff auf das Objekt vom Typ `Immutable` alle Werte der `final`-Felder aus dem Hauptspeicher holt. Dabei holt er sich auch die Werte aller abhängigen Objekte; er macht also einen partiellen Refresh seines Arbeitsspeichers aus dem Hauptspeicher. Danach muss er aber keinen Refresh mehr machen. Das ist auch sinnvoll so. Für die `final`-Referenzvariable braucht er sowieso keinen Refresh mehr, weil die Adresse des Arrays konstant ist und sich nicht mehr ändern kann. In einem unveränderlichen Typ ändern sich auch die Inhalte des Arrays und seiner Elemente nicht; sonst wäre der Typ veränderlich.

Die Unveränderbarkeit des Arrays ist aber durch nichts gewährleistet, weil die `final`-Deklaration der Array-Referenz nur sagt, dass die Adresse des Arrays konstant ist, nicht aber der Inhalt des Array selbst. Es wäre also prinzipiell möglich, dass sich das Array und seine Elemente ändern. Für die Sichtbarkeit dieser späteren Änderungen gibt das Java Memory Modell keine Garantien. Sehen wir uns das im Beispiel mal an:

```
public class NoLongerImmutable {
  private final int[] finalArrayRef;
  public NoLongerImmutable(int start, int size) {
    finalArrayRef = new int[size];
    for (int i=0;i<size;i++)
       finalArrayRef[i] = start+i;
  }
  public String toString() {
    return "["+ Arrays.toString(finalArrayRef) +"]";
  }
  public void update(int idx, int val) {
    finalArrayRef[idx] = val;
  }
}
```

Hier ist nun nicht garantiert, dass lesende Threads die Modifikation an den Array-Elementen zu sehen bekommt, die andere Threads mit Hilfe von `update()` nach der Konstruktion gemacht haben. Lesende Threads müssen nur ein einziges Mal einen Refresh ihres Arbeitsspeichers machen, nämlich beim ersten Zugriff auf das `final`-Feld und alle seine abhängigen Objekte. Veränderungen an den abhängigen Objekten, die später noch passieren, müssen nicht - aber könnten (zum Beispiel durch weitere Synchronisationspunkte an ganz anderen Stellen) - sichtbar werden. Wenn sie garantiert sichtbar werden sollen, dann muss man mit anderen Mitteln (zum Beispiel explizite Synchronisation) für die Sichtbarkeit sorgen.
Man beachte, dass dieses Missverständnis bei unveränderlichen Typen nicht auftreten kann, weil in ein einem unveränderlichen Typ alle abhängigen Objekte ebenfalls unveränderlich sind. Es genügt also der Refresh aller abhängigen Objekte beim allerersten Zugriff, weil sich danach nichts mehr an den abhängigen Objekten ändert.

Unterschiede zu volatile

Man beachte, dass die Speichereffekte bei `final`-Feldern ganz anders sind als bei `volatile`-Referenzvariablen. Bei `volatile`-Variablen löst jeder schreibende oder lesende Zugriff einen Flush bzw. Refresh des gesamten Arbeitsspeichers aus (siehe [EFF4]). Bei `final`-Referenzvariablen löst nur das Ende des Konstruktors einen partiellen Flush und nur der erstmalige lesende Zugriff in jedem Thread einen partiellen Refresh aus. `volatile` ist also "teuer" als `final`, weil mehr Memory Barriers ausgelöst werden, und `volatile` hat keine Garantie für abhängige Objekte.

final Variablen vs. final Felder

Noch ein Hinweis auf mögliche Missverständnisse: "`final`" ist nicht gleich "`final`". Die Garantien des Java Memory Modells im Zusammenhang mit `final` beziehen sich nur auf `final`-Felder von Objekten, nicht auf `final`-Variablen. Die Speichereffekte für `final` werden nicht ohne Grund als "*Initialization Safety*"-Garantie bezeichnet. Die Garantie besagt lediglich, dass `final`-Felder eines Objekts stets in ihrer fertig initialisierten Form sichtbar werden, niemals vorher. Für andere Verwendungen von `final` (zum Beispiel als Parameter und lokalen Variablen von Methoden) gibt es keine Garantien.

Schauen wir uns zur Illustration einen Fall an, in dem zwar `final` verwendet wird, seine Verwendung aber nichts mit *Initialization Safety* zu tun hat.

```
public class ActiveObject{
  public ActiveObject(long times) throws InterruptedException {
    final long[] argumentAndResult = new long[2];
    argumentAndResult[0] = times;

    Runnable r = new Runnable() {
      public void run() {
        long start = System.nanoTime();
        for (int i=0;i<argumentAndResult[0];i++)
           System.out.print("*");
        argumentAndResult[1] = System.nanoTime()-start;
      }
    };
    Thread t = new Thread(r,"printer");
    t.start();
    t.join();
    System.out.println(argumentAndResult[1]);
  }
}
```

Das Beispiel zeigt ein gängiges Idiom für die Parametrisierung von Runnables: da die `run()`-Methode keine Argumente haben darf, implementiert man Runnables gerne als anonyme innere Klassen und gibt ihnen Zugriff auf `final`-Variablen des umgebenden Kontextes. Auf diese Art erreicht man eine indirekte Parametrisierung der `run()`-Methode.
In diesem Beispiel haben der main-Thread und der printer-Thread gemeinsam und konkurrierend Zugriff auf das `final`-Array `argumentAndResult`. Das erste Array-Element ist der Parameter für den printer-Thread; in dem zweiten Array-Element legt der printer-Thread das Ergebnis ab, d.h. die Zeit, die für das Ausdrucken von `times` vielen `"*"` gebraucht wurde.

Hier hat die Verwendung von `final` überhaupt nichts mit den oben besprochenen Speichereffekten für `final`-Felder zu tun, denn hier brauchen wir gar keine Garantien im Zusammenhang mit `final`-Variablen.

Die benötigten Speichererffekte werden hier durch Thread-Start und Thread-Ende geliefert, nicht durch die `final`-Deklaration der Array-Referenz. Der Start des printer-Threads löst einen Flush im main-Thread und einen Refresh im printer-Thread aus. Das heißt, der neu gestartete printer-Thread kann sehen, was der ihn startende main-Thread gemacht hat. Analog beim

Thread-Ende: der main-Thread wartet mit `Thread.join()` auf das Ende des printer-Threads und kann dann alles sehen, was der printer-Thread gemacht hat.

Und eine letzter Hinweis auf mögliche Mißverständnisse:
Für die Thread-Ende-Garantie ist übrigens wichtig, dass sich der eine Thread über das Ende des anderen Threads aktiv informiert hat (per `join()` oder `isAlive()`). Wenn der printer-Thread nur zufällig schon gerade fertig ist, dann ist nicht garantiert, dass der main-Thread sieht, was der printer-Thread gemacht hat, weil der Refresh im main-Thread erst durch `join()` oder `isAlive()` ausgelöst wird.

Zusammenfassung

In diesem Beitrag haben wir uns die "*Initialization Safety*"-Garantie für `final`-Felder von einem Referenztyp angesehen. Die Garantie besagt, dass `final`-Felder eines Objekts einem andern Thread stets in ihrer fertig initialisierten Form sichtbar werden, niemals vorher. Dabei werden auch alle abhängigen Objekte in ihrer fertig initialisierten Form sichtbar.

Diese Garantie gibt es nur für `final`-Felder und nicht für `final`-Variablen. Die "*Initialization Safety*"-Garantie wird für die Implementierung von unveränderlichen Typen gebraucht: in einem unveränderlichen Typ müssen alle Felder als `final` deklariert sein und alle abhängigen Objekte müssen ihrerseits unveränderlich sein.

Im nächsten Beitrag sehen wir uns, was man trotz Verwendung von `final` bei der Implementierung von unveränderlichen Typen falsch machen kann. Die "*Initialization Safety*"-Garantie gilt nämlich nur, wenn der sogenannte *Object Escape* verhindert wird. Das schauen wir uns im nächsten Beitrag genauer an.

Verweise

/EFF1/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/EFF2/ JMM - Überblick über das Java Memory Modell (JMM)
Klaus Kreft & Angelika Langer
JavaMagazin 8.08

/EFF3/ JMM - Die Kosten der Synchronisation
Klaus Kreft & Angelika Langer
JavaMagazin 9.08

/EFF4/ JMM - Details zu volatile
Klaus Kreft & Angelika Langer
JavaMagazin 10.08

/EFF5/ JMM - Double-Check Locking
Klaus Kreft & Angelika Langer
JavaMagazin 11.08

/EFF6/ JMM - Initialisation-Safety-Garantie (für final Felder von primitivem Typ)
Klaus Kreft & Angelika Langer
JavaMagazin 12.08

/EFF7/ JMM - Initialisation-Safety-Garantie (für final Felder von einem Referenztyp)
Klaus Kreft & Angelika Langer
JavaMagazin 1.09

Java Memory Modell

Teil 10: Über die Gefahren allzu aggressiver Optimierungen

Klaus Kreft & Angelika Langer

Wir haben in den letzten beiden Beiträge [EFF6 / EFF7] die Initialisation-Safety-Garantie für `final`-Felder am Beispiel des Racy-Single-Check-Idioms besprochen. Das Racy-Single-Check-Idiom ist eine Variante des Double-Check-Idioms [EFF5] und verzichtet aus Performance-Gründen bewußt auf jegliche Form von Synchronisation oder die Verwendung von `volatile`. Es handelt sich also um eine (relativ aggressive) Optimierung, die unter gewissen Randbedingungen manchmal sinnvoll einsetzbar ist. Um solche Optimierungen ins rechte Licht zu rücken und zu zeigen, wie fragwürdig der Verzicht auf Synchronisation und `volatile` im Allgemeinen ist, wollen wir in diesem Beitrag einige typische Missverständnisse und Fehlerfälle diskutieren.

Racy-Single-Check und unveränderlichen Typen

Sehen wir uns das Racy-Single-Check-Beispiel noch einmal an. Beim Double- oder Single-Check-Idiom geht es um die verzögerte Initialisierung (*lazy initialisation*) eines Feldes. Beim Racy-Single-Check-Idiom muss das Feld, das "lazy" initialisiert wird, unveränderlich sein. Wenn es eine Referenz ist, dann muss die Referenz selbst unverändert bleiben und muss auf ein Objekt von einem unveränderlichen Typ verweisen. Nehmen wir mal den Typ `java.lang.Integer`. Dann sieht das Idiom so aus:

```
public class MyClass {
    private Integer lazyField = null;
    ...

    public Integer getMyField() {
        if (lazyField == null)
            lazyField = new Integer(10000);

        return lazyField;
    }
    ...
}
```

In der Klasse `MyClass` wird im Zusammenhang mit der Initialisierung des Feldes `lazyField` vom Typ `java.lang.Integer` weder Synchronisation noch `volatile` verwendet, obwohl konkurrierende Zugriffe von mehreren Threads aus erlaubt sind. Das ist die Optimierung, die beim Racy-Single-Check-Idiom gewünscht ist. Sie ist auch korrekt. Wegen des Fehlens von Synchronisation und `volatile` gibt es hier zwar keine Sichtbarkeitsgarantien. Es könnte deshalb sein, dass ein Thread gar nicht sieht, dass ein anderer Thread bereits die Lazy-Initialisierung gemacht hat. Das stört aber nicht; dann macht der Thread die Initialisierung eben noch einmal selber. Racy-Single-Check läßt also Mehrfach-Initialisierungen zu.

Voraussetzung für dieses Idiom ist, dass das `lazyField` nach der Initialisierung nicht mehr geändert wird und außerdem auf ein Objekt von einem unveränderlichen Typ zeigt; sonst funktioniert das Idiom nicht. Wenn sich an dem Feld oder dem referenzierten Objekt nach der Initiasierung noch etwas ändern könnte, dann wären Mehrfach-Initialisierungen problematisch: das Feld oder das referenzierte Objekt könnten nach der Initialisierung modifiziert werden und eine erneute Initialisierung würde die bereits erfolgte Modifikation wieder zunichte machen. Damit ergäbe sich eine problematische Race Condition. Aber wenn das `lazyField` sich nicht ändert und auf ein Objekt von einem unveränderlichen Typ verweist, braucht man in der Tat im oben gezeigten Beispiel weder Synchronisation noch `volatile`.

Dabei ist es wichtig, dass der Typ des referenzierten Objekts unveränderlich in einem sehr spezifischen Sinne ist. Wir haben in den letzten beiden Beiträgen (siehe [EFF6] und [EFF7]) erläutert, was von einem unveränderlichen Typ erwartet wird. Nicht jeder Typ, der von sich behauptet, er sei unveränderlich, ist ohne Synchronisation und `volatile` in einem Racy-Single-Check-Idiom verwendbar. Für einen unveränderlichen Typ genügt es nämlich nicht, dass er nur lesende und keine modifizierenden Methoden hat und alle seine Felder ihrerseits wiederum unveränderlich sind. Ein unveränderlicher Typ muss außerdem für die Sichtbarkeit seiner Inhalte sorgen; sonst kann man ihn nicht ohne Synchronisation und `volatile` in einem Racy-Single-Check-Idiom verwenden. Das bedeutet, ein unveränderlicher Typ muss sicherstellen, dass die unveränderlichen Inhalte des Objekts nach der Konstruktion allen benutzenden Threads sichtbar werden. Zu diesem Zweck werden alle Felder eines unveränderlichen Typs als `final` deklariert, damit die Initialisation-Safety-Garantie des Java Memory Modells für die Sichtbarkeit sorgt

Wenn alle diese (teilweise subtilen) Randbedingungen erfüllt sind, dann kann man aus Performancegründen für eine Lazy-Initialisierung eines Feldes nach dem Racy-Single-Check-Idiom sowohl auf Synchronisation als auch auf die Verwendung von `volatile` verzichten. Das ist eine hoch-optimierte Lösung und es stellt sich die Frage: Sind solche Optimierungen sinnvoll? Wie oft kommt sowas vor? Braucht man für den konkurrierenden Zugriff auf ein unveränderliches Objekt grundsätzlich keine Synchronisation und niemals `volatile`?

Wir hatten im letzten Beitrag komplett auf Synchronisation und `volatile` verzichtet - im wesentlichen, weil wir die Initialisation-Safety-Garantie für `final`-Felder diskutieren wollten, die das Java Memory Modell gibt. In diesem Beitrag wollen wir demonstrieren, wie fragil eine solch hoch-optimierte Lösung sein kann.

Genereller Verzicht auf Synchronisation/volatile bei Verwendung von unveränderlichen Typen ?

Manchmal unterstellen Java-Programmierer, dass beim Zugriff auf Objekte von einem unveränderlichen Typ grundsätzlich keine Synchronisation und auch kein `volatile` erforderlich wäre weil, konkurrierende Zugriffe auf unveränderliche Objekte prinzipiell unproblematisch sind. Das ist leider ein Irrtum.

Man muss in dem obigen Beispiel nur eine Kleinigkeit ändern und schon ist es falsch. In dem Racy-Single-Check-Beispiel sind Mehrfach-Initialisierungen harmlos und es ist deshalb egal, dass es keinerlei Garantien für die Sichtbarkeit der Referenz `lazyField` und des referenzierten Objekts gibt.

Im Allgemeinen wird es aber wahrscheinlich doch so sein, dass andere Threads garantiert sehen sollen, was ein initialisierender Thread gemacht hat. Das wäre beispielsweise der Fall, wenn die Mehrfachinitialisierungen nicht akzeptabel sind und die Initialisierung nur einmal gemacht werden sollte. Ein Beispiel wäre eine Initialisierung, bei der die Kommunikationsverbindung zu einem anderen Service aufgebaut wird.

Unter solchen geringfügig veränderten Randbedingungen wird plötzlich eine Sichtbarkeitsgarantie für das `lazyField` gebraucht, damit die anderen Threads nach der ersten Initialisierung sehen, dass das Feld nicht mehr `null` ist und keine weitere Initialisierung mehr gemacht werden darf. Die oben gezeigte Lösung, die komplett auf Synchronisation und `volatile` verzichtet, wäre dann falsch.

Die Mehrfachinitialisierungen ist auch dann inakzeptabel, wenn sich beispielsweise die Referenz auf das unveränderliche Objekt ändern kann, weil es eine Methode gibt, die zuläßt, dass die Referenz neu belegt wird und dann auf ein anderes (auch wieder unveränderliches) Objekt verweist. Hier ist ein Beispiel für diese Situation: das `lazyField` ist nicht mehr unveränderlich, weil es eine Methode `setMyField()` gibt, die die Veränderung des Feldes erlaubt.

```
public class MyClass {
    private volatile Integer lazyField = null;
    ...

    public Integer getMyField() {
        if (lazyField == null)
            lazyField = new Integer(10000);

        return lazyField;
    }
    public void setMyField(int i) {
        lazyField = new Integer(i);
    }
}
```

Nun ist der Racy-Single-Check ohne Synchronisation und `volatile` nicht mehr möglich, weil Mehrfachinitialisierungen zu Fehlern führen könnten. Nach der ersten Initialisierung könnte das `lazyField` bereits geändert worden sein, die anderen Threads sähen aber möglicherweise immer noch den Wert `null` und würden eine erneute Initialisierung machen, die die bereits erfolgte Änderung des `lazyField` überschreiben würde.

Um das zu verhindern, haben wir das Feld als `volatile` deklariert. Die Speichereffekte von `volatile` (siehe [EFF4]) sorgen dafür, dass der Inhalt des Feldes `lazyField` (also die Adresse des neu erzeugten `Integer`-Objekts) allen anderen Threads, die die Methode `getMyField()` aufrufen, sichtbar wird.

Man kann die Sichtbarkeit auch mit Hilfe von Synchronisation garantieren. Das könnte dann so aussehen:[1]

[1] Die Annotation `@GuardedByLock` ist übrigens keine vordefinierte Standard-Annotation wie zum Beispiel `@Override` oder `@SuppressWarning`. Es handelt sich hingegen um eine Annotation, die man sich selbst zu Zwecken der besseren Dokumentation des Source-Codes definieren kann. Die Idee dafür stammt von Brian Goetz, der die Annotation in den Code-Beispielen in seinem Buch "Java Concurrency in Practice" (siehe [JCP]) verwendet hat.

```
public class MyClass {
    @GuardedBy("lock") private Integer lazyField = null;
    private Object lock = new Object();

    ...

    public Integer getMyField() {
        synchronized(lock) {
            if (lazyField == null)
                lazyField = new Integer();
        }
        return lazyField;
    }
    public void setMyField(int i) {
        synchronized(lock) {
            lazyField = new Integer(i);
        }
    }
    ...
}
```

Wie auch immer man für die Sichtbarkeit des konkurrierend verwendeten Feldes `lazyField` sorgt, das Beispiel zeigt, dass auch bei der Verwendung von unveränderlichen Typen sehr wohl Synchronisation oder `volatile` gebraucht wird und dass der völlige Verzicht darauf nur in seltenen Fällen überhaupt möglich ist. Das Racy-Single-Check-Idiom ist ein solcher seltener Fall, der aber so viele subtile Randbedingungen hat, dass eine winzige Änderung des Kontextes bereits dazu führt, dass man die im letzten Beitrag besprochene Initialisation-Safety-Garantie gar nicht braucht, weil man sowieso andere Mittel des Java Memory Modells wie Synchronisation oder `volatile` benutzen muss.

Es gibt aber noch mehr Irrtümer "überflüssige" Synchronisation betreffend, die zu Fehlern führen können.

Race Conditions bei der Konstruktion von Objekten

Irrtümlicherweise wird gelegentlich vermutet, die Konstruktion von Objekten sei atomar und es könne keine Race Conditions in Konstruktoren geben. Das ist nicht so, wie das folgende Beispiel zeigt. Es geht um eine Klasse mit einem Feld, einem Konstruktor, einer lesenden Zugriffsmethode und weiteren Methoden, die hier aber nicht gezeigt sind; darunter sind auch modifizierende Methoden.

```
class Sizes {
    private int[] sizes ;
    public Sizes(int... args) {
        System.arraycopy(args, 0, sizes=new int[args.length], 0, args.length);
    }
    public String toString() {
        return "Sizes: "+Arrays.toString(sizes);
    }
    // … further methods, including modifying methods …
}
```

Nun kann es vorkommen, dass der Konstruktor konkurrierend mit der `toString`-Methode abläuft, etwa in der folgenden Situation:

```
class Test { // falsch

    private static Sizes ref = null;

    public static void main(String[] args) {
      Runnable publisher = new Runnable() {
        public void run(){
           ref = new Sizes(512, 1024, 2048);
           // … do some more stuff …
        }
      };
      new Thread(publisher,"Publisher").start();

      Runnable spy = new Runnable() {
        public void run(){
          if (ref != null)
             System.out.println(ref);
        }
      };
      new Thread(spy,"Spy").start();
    }
```

```
}
```

Auf die Referenzvariable `ref` vom Typ `Sizes` wird von zwei Threads mit Namen "Publisher" und "Spy" konkurrierend zugegriffen. Der Publisher-Thread ruft den Konstruktor auf und die Referenz auf das neu konstruierte Objekt wird an die gemeinsam verwendete Referenzvariable `ref` zugewiesen. Der andere Thread wiederum greift über die Referenzvariable `ref` auf das Objekt zu und will es ausdrucken. Dann kann es passieren, dass die Adresse des neuen Objekts dem Spy-Thread sichtbar wird, die Felder des referenzierten, neu erzeugten Objekts jedoch nicht oder nur teilweise sichtbar sind. Das heißt, für den Spy-Thread sieht es so aus, als wäre das neue Objekt noch nicht fertig konstruiert. Wie kann so etwas geschehen?

Die Situation ist zugegebenerweise ein wenig ungewöhnlich, denn hier ist es nicht so, dass erst die Referenzvariable `ref` mit einer Adresse belegt wird, ehe ein oder mehrere Threads gestartet werden, die dann gemeinsam auf das referenzierte Objekt zugreifen. In dem Falle liefe die Konstruktion des neuen Objekts und die Zuweisung der Adresse an die gemeinsam verwendete Referenzvariable `ref` vor dem Start der konkurrierend zugreifenden Threads ab. Dann würde der Start der jeweiligen Threads zu einem Refresh der Caches der Threads führen, so dass die Threads die Adresse und das referenzierte Objekt zu sehen bekämen. Es gäbe also ein klare Happens-Before-Beziehung: das Objekt wird erzeugt und über die gemeinsam verwendete Referenzvariable zugänglich gemacht, ehe andere, neu gestartete Threads darauf zugreifen.

Stattdessen werden in der oben gezeigten Situation die beiden Threads gestartet, noch ehe die die gemeinsam verwendete Referenzvariable `ref` initialisiert ist. Die beiden Threads warten auch nicht aufeinander. Deshalb laufen hier die Konstruktion des neuen Objekts und die Zuweisung der Adresse an die gemeinsam verwendete Referenzvariable `ref` (beides im Publisher-Thread) konkurrierend zum lesenden Zugriff (im Spy-Thread) ab. Da es hier keine Sichtbarkeitsgarantien gibt, kann es wie oben beschrieben passieren, dass der Spy-Thread das neue Objekt während der Konstruktion sieht, noch ehe der Konstruktor fertig ist, weil nicht für Synchronisation gesorgt wird.

Korrekt wäre folgende Implementierung, die Thread-Synchronisation nutzt:

```
class Test { // okay

    @GuardedBy ("lock") private static Sizes ref = null;
    private static Object lock = new Object();

    public static void main(String[] args) {
      Runnable publisher = new Runnable() {
            public void run() {
                synchronized (lock) {
                    ref = new Sizes(512, 1024, 2048);
                }
                // … do some more stuff …
            }
      };
      new Thread(publisher,"Publisher").start();

      Runnable spy = new Runnable() {
            public void run() {
                synchronized (lock) {
                    System.out.println(ref);
                }
            }
      };
      new Thread(spy,"Spy").start();
    }
}
```

Die Zugriffe auf die gemeinsam verwendete Referenzvariable `ref` sind nun sequenzialisiert und es gibt keine Race Condition mehr. Der Spy-Thread kann die Adresse des neuen Objekts entweder vor oder nach dem entsprechenden `synchronized`-Block im Publisher-Thread sehen. Da das Anfordern und das Freigeben von Locks einen Refresh bzw. Flush des Arbeitsspeichers auslöst, ist gesichert, dass der Spy-Thread entweder die Referenz *und* den Inhalt des neuen Objekts oder `null` zu sehen bekommt.

Würde es hier genügen, die gemeinsam verwendete Referenzvariable `ref` als `volatile` zu deklarieren und auf die Synchronisation zu verzichten? Das sähe dann so aus:

```
class Test { // falsch

    private static volatile Sizes ref = null;

    public static void main(String[] args) {
      Runnable publisher = new Runnable() {
```

```
          public void run(){
             ref = new Sizes(512, 1024, 2048);
             // … do some more stuff …
          }
        };
        new Thread(publisher,"Publisher").start();

        Runnable spy = new Runnable() {
          public void run(){
            System.out.println(ref);
          }
        };
        new Thread(spy,"Spy").start();
      }
  }
```

Nein, `volatile` würde hier nicht ausreichen. Der Typ `Sizes` ist ein veränderlicher Typ, der nicht einmal thread-sicher ist, was man daran sehen kann, dass seine Methoden (siehe z.B. `toString()`) nicht `synchronized` deklariert sind und auch sonst keine Synchronisation in der Implementierung des Typs verwendet wird. Die benutzerseitige Synchronisierung ist also zwingend erforderlich.

Was wäre, wenn der Typ `Sizes` unveränderlich wäre, so wie `java.lang.Integer` aus dem ersten Beispiel? Würde es dann genügen, die gemeinsam verwendete Referenzvariable `ref` als `volatile` zu deklarieren und auf die Synchronisation zu verzichten?

Ja, das wäre möglich. Da es dann nur lesende Zugriffe auf das `Sizes`-Objekt gäbe, machte es keine Probleme, wenn die Zugriffe konkurrierend statt sequentiell erfolgten. Man müsste dann nur noch für die Sichtbarkeit des Objekts sorgen und dafür würde es genügen, die Referenzvariable `ref` als `volatile` zu deklarieren.

Spielt es dann eine Rolle, ob der unveränderliche Typ `Sizes` korrekt implementiert ist und alle seine Felder als `final` deklariert hat, so wir es in den letzten beiden Beiträgen (siehe [EFF6] und [EFF7]) besprochen haben?

Nein, es ist egal, vorausgesetzt wir haben die Adresse auf das Objekt als `volatile` deklariert, denn dann sorgen die Speichereffekte von `volatile` für die Sichtbarkeit des Objekts und man braucht die `final`-Deklaration der Felder und somit die Initialisation-Safety Garantie nicht mehr.

Die `final`-Deklaration der Felder eines unveränderlichen Typs wird nur gebraucht, wenn ansonsten weder Synchronisation noch `volatile` verwendet wird, so wie es im Racy-Single-Check-Idiom der Fall ist.

Zusammenfassung

Das Weglassen von Synchronisation und `volatile` (d.h. beides gleichzeitig weggelassen) ist eine aggressive Optimierung, die nur selten (z.B. beim Racy-Single-Check-Idiom) angewandt werden kann. Bereits geringfügige Änderungen an der Racy-Single-Check-Situation können dazu führen, dass eine `volatile`-Deklaration oder gar Synchronisation gebraucht wird. Eigentlich ist es eine Binsenweisheit: bei jeder Optimierung muss zuvor überlegt werden, ob die Optimierung möglich und korrekt ist, und insbesondere Optimierungen, die auf Synchronisation von konkurrierenden Zugriffen verzichten, sind diffizil und fehleranfällig.
Im Zusammenhang mit typischen Missverständnissen über den Verzicht auf Synchronisation haben wir außerdem gezeigt, dass gelegentlich sogar der Aufruf eines Konstruktors synchronisiert werden muss.
Im den nächsten Beiträgen sehen wir uns ein weiteres Optimierungsinstrument der Java Concurrency an: die atomaren Variablen.

Verweise

/EFF1/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/EFF2/ JMM - Überblick über das Java Memory Modell (JMM)
Klaus Kreft & Angelika Langer
JavaMagazin 8.08

/EFF3/ JMM - Die Kosten der Synchronisation
Klaus Kreft & Angelika Langer
JavaMagazin 9.08

/EFF4/ JMM - Speichereffekte von volatile
Klaus Kreft & Angelika Langer
JavaMagazin 10.08

/EFF5/ JMM - Double-Check Locking
Klaus Kreft & Angelika Langer
JavaMagazin 11.08

/EFF6/ JMM - Initialisation-Safety-Garantie
Klaus Kreft & Angelika Langer
JavaMagazin 12.08

/JCP/ Java Concurrency in Practice
Brian Goetz et.al.
Addison-Wesley 2006

Java Memory Modell

Teil 11: Atomic Scalars

Klaus Kreft & Angelika Langer

Wir haben in einem der Beiträge in dieser Kolumne [EFF4] über die Performance-Vorzüge und Speichereffekte von `volatile` im Kontext von konkurrierenden Threads geschrieben. Leider hat `volatile` gewisse Einschränkungen. So ist es beispielsweise nicht möglich, eine atomare Read-Modify-Write-Sequenz auf einer `volatile` Variablen auszuführen; es sind nur die einzelnen Lese- und Schreibzugriffe atomar, nicht aber die ganze Sequenz. Als logische Verallgemeinerung von `volatile` Variablen gibt es seit Java 5.0 atomare Variablen, die ununterbrechbare Read-Modify-Write-Sequenzen unterstützen und ansonsten dieselben Vorzüge und Speichereffekte wie `volatile` haben. In diesem Beitrag sehen wir uns die atomaren skalaren Variablen an.

Es gibt immer mal wieder Situationen, in denen man die Thread-Synchronisation mit Hilfe von Locks durch die Verwendung von `volatile`-Variablen ersetzt möchte, um die Performance zu verbessern. Das ist möglich, wenn der Zugriff auf eine gemeinsam verwendete Variable sowieso schon ununterbrechbar ist, wie zum Beispiel beim lesenden oder schreibenden Zugriff auf primitive Typen (außer `long` und `double`) und Referenzen. In solchen Fällen wird die Synchronisation nicht gebraucht, um den Zugriff auf die Variable ununterbrechbar zu machen, sondern nur noch, um die Variable sichtbar zu machen. Da es seit Java 5.0 aber Sichtbarkeitsgarantien für `volatile`-Variablen gibt, kann man hier auf die Synchronisation verzichten und stattdessen `volatile` verwenden. Mit `long` und `double` funktioniert die Lösung auch. Im Unterschied zu den übrigen primitiven Typen ist der Zugriff auf non-`volatile`-Variablen vom Typ `long` und `double` unterbrechbar, aber wenn eine Variablen vom Typ `long` und `double` als `volatile` erklärt ist, dann ist der Zugriff atomar. Diese Ununterbrechbarkeit ist bei `long` und `double` also ein zusätzlicher Effekt von `volatile`, den man bei dieser Situation ausnutzt.

Ein threadsicherer Zähler

Leider ist bei einer solchen Optimierung mit `volatile` nur der einzelne Lese- bzw. Schreibzugriff atomar. Dessen sollte man sich bewusst sein, sonst ist die Optimierung auf einmal gar nicht mehr threadsicher. Hier ist ein Beispiel für eine Abstraktion, in der aus Optimierungsgründen auf Synchronisation verzichtet wird und stattdessen `volatile` verwendet wird:

```
public class VolatileCounter {  // Vorsicht: falsch !!!
  private volatile int value;
  public int getValue()  { return   value; }
  public int increment() { return ++value; }
  public int decrement() { return --value; }
}
```

Leider hat sich ein Fehler eingeschlichen. Der völlige Verzicht auf Synchronisation ist hier nicht korrekt, denn nicht alle Zugriffe auf das `int`-Feld `value` sind ununterbrechbar. Die Sprachspezifikation garantiert zwar, dass der lesende und schreibende Zugriff auf Variablen von primitivem Typ ununterbrechbar ist, wenn sie als `volatile` deklariert sind. Wie oben bereits erwähnt, gilt das auch für `volatile`-Variablen vom Typ `long` und `double`. Für das Inkrementieren und Dekrementieren gilt es aber nicht. Ein Inkrement ist eine Sequenz von Lesen, Modifizieren und Schreiben. Das ist bereits eine so komplexe Operation, dass sie unterbrechbar ist.

Man muss also die Methoden `increment()` und `decrement()` synchronisieren, damit sie ununterbrechbar sind. Lediglich die Methode `getValue()` kann ohne Synchronisation auskommen. Dann sähe die `Counter`-Klasse so aus:

```
public class SynchronizedCounter {  // korrekt, aber nicht optimal
  private volatile int value;
  public int getValue()  { return   value; }
  public synchronized int increment() { return ++value; }
  public synchronized int decrement() { return --value; }
}
```

CAS - Compare-and-Swap

Prozessoren haben im allgemeinen ein atomaren Befehl, der eine sogenannte atomare Compare-and-Swap-Operation (auch CAS genannt) ausführt. Basierend auf solchen atomaren CAS-Operationen der Hardware kann man ununterbrechbare Read-Modify-Write-Sequenzen durchführen, die ganz ohne Synchronisation auskommen.

Leider war es bis Java 5.0 nicht möglich, diese Funktionalität von Java einfach auszunutzen. Seit Java 5.0 gibt es im Package `java.util.concurrent.atomic` verschiedene Klassen, die basierend auf CAS-Operationen ununterbrechbare Read-Modify-Write-Sequenzen anbieten. Dazu gehören Typen, wie `AtomicInteger`, `AtomicLong` und `AtomicBoolean`.

Eine Compare-and-Swap-Operation hat 3 Operanden:

- eine Speicherstelle

- den erwarteten alten Wert an dieser Speicherstelle
- einen neuen Wert für die Speicherstelle

Der Prozessor liest den alten Wert, vergleicht ihn mit dem erwarteten Wert und schreibt den neuen Wert an die Speicherstelle, wenn der gelesene Inhalt dem erwarteten Wert entspricht; wenn die Erwartung nicht zutrifft, wird nichts geändert. Die ganze Sequenz von Lesen, Vergleichen und Ändern ist ununterbrechbar.

Damit kann man ohne Synchronisation Modifikationen an Variablen vornehmen, die von mehreren Thread gemeinsam verwendet werden. Das Vorgehen ist dabei folgendes: Man verfolgt eine optimistische Strategie und versucht einfach mal, die Variable zu ändern. Wenn kein anderer Thread konkurrierend zugreift, dann klappt die CAS-Operation. Wenn sie scheitert, dann gab es wohl konkurrierende Zugriffe und ein anderer Thread war schneller. In diesem Fall wird der Änderungsversuch solange wiederholt, bis er erfolgreich ist.

Solche atomaren CAS-Operationen haben den Vorteil, dass sie bei wenig Konkurrenz um den Variablenzugriff performanter sind als synchronisierte Zugriffe, weil der gesamte Synchronisationsaufwand wegfällt. Bei sehr viel Konkurrenz kann unter Umständen die Synchronisation wieder günstiger sein, weil die CAS-Zugriffe dann mit hoher Wahrscheinlicheit scheitern und wiederholt werden müssen. Das kostet dann zusätzliche CPU-Zeit. Atomare Operationen haben aber gegenüber Synchronisation stets den Vorteil, dass zumindest ein Thread immer erfolgreich sind wird. So etwas wie Deadlocks, wo kein einziger Thread mehr arbeitet, kann mit atomaren Operationen nicht vorkommen.

Neben der Ununterbrechbarkeit haben die Zugriffe auf atomare Variablen dieselben Speichereffekte wie die Zugriffe auf `volatile`-Variablen. Deshalb werden die atomaren Variable auch als logische Verallgemeinerung von `volatile` angesehen.

Ein threadsicherer Zähler unter Verwendung von atomaren Variablen

Kehren wir zu unserem threadsicheren Zähler von oben zurück. Wie würde er aussehen, wenn wir ihn mit einer atomaren Variablen aus dem Package `java.util.concurrent.atomic` implementieren? Offensichtlich bietet sich hier die Verwendung des `AtomicInteger` an. Damit sieht die `Counter`-Abstraktion dann so aus:

```
public class AtomicCounter {  // korrekt und optimiert
  private AtomicInteger value = new AtomicInteger();
  public int getValue() { return value.get(); }
  public int increment() {
   int oldValue = value.get();
   while (!value.compareAndSet(oldValue, oldValue + 1))
     oldValue = value.get();
   return oldValue + 1;
  }
  public int decrement() {
   int oldValue = value.get();
   while (!value.compareAndSet(oldValue, oldValue - 1))
     oldValue = value.get();
   return oldValue - 1;
  }
}
```

Der Zähler ist jetzt kein `int` mehr, sondern ein `AtomicInteger`. Die Methode `getValue()` liefert in der atomaren Operation `AtomicInteger.get()` den Wert des Integers. Da atomare Variablen diesselben Speichereffekte wie `volatile` haben, ist das Lesen mit einem Refresh verbunden.

Das Inkrementieren erfolgt nun so: zunächst wird der aktuelle Wert des Integers gelesen, um den neuen Wert `oldValue+1` zu berechnen. Dann soll die Modifikation gemacht werden. Zwischen Lesen und Modifizieren ist der Thread unterbrechbar; es könnte also ein anderer Thread den Integer in der Zwischenzeit modifiziert haben. Deshalb wird die atomare Operation `compareAndSet()` aufgerufen. Sie prüft, ob der zuvor gelesene Wert noch aktuell ist. Wenn ja, dann ist die Modifikation erfolgreich und es kommt `true` zurück. Wenn nein, dann kommt `false` zurück und der ganze Vorgang von "Wert auslesen", "neuen Wert berechnen" und `compareAndSet()` wird wiederholt. Da `compareAndSet()` mit Lesen und Schreiben der Variablen verbunden ist, wird sowohl ein Refresh als auch ein Flush ausgelöst. Das Dekrementieren funktioniert analog.

Typisch für die Verwendung von atomaren Variablen ist die Schleife, in der die Operation solange wiederholt wird, bis sie erfolgreich ist. Insgesamt sieht die `Counter`-Implementierung mit Hilfe von atomaren Variablen deutlich komplizierter aus als die einfache Implementierung mit Synchronisation. Die Klasse `AtomicInteger` hat zwar spezielle `incrementAndGet`- und `getAndIncrement`-Methoden, die dem Postfix- und Prefix-Inkrement entsprechen; analog fürs Dekrementieren. Deutlich

einfacher sieht die Lösung damit aber auch nicht aus. Zum anderen wollten wir aus didaktischen Gründen hier auch das typische Vorgehen bei der Benutzung von atomaren Variablen zeigen.

Zusammenfassung

In diesem Beitrag haben wir uns angesehen, wie man atomare Skalare aus dem Package `java.util.concurrent.atomic` als bessere `volatile` Variablen für die Optimierung von konkurrierenden Zugriffen auf skalare Variablen einsetzen kann. Von zentraler Bedeutung sind dabei die sogenannten CAS(Compare-and-Swap)-Sequenzen, die im allgemeinen von der Hardware als ununterbrechbare Operationen unterstützt werden und über Java-Abstraktionen wie `AtomicInteger`, `AtomicLong` und `AtomicBoolean` nutzbar gemacht werden. Neben den atomare skalaren Variablen gibt es auch noch atomare Referenzvariablen. Die besprechen wir im nächsten Beitrag.

Verweise

/EFF1/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/EFF2/ JMM - Überblick über das Java Memory Modell (JMM)
Klaus Kreft & Angelika Langer
JavaMagazin 8.08

/EFF3/ JMM - Die Kosten der Synchronisation
Klaus Kreft & Angelika Langer
JavaMagazin 9.08

/EFF4/ JMM - Speichereffekte von volatile
Klaus Kreft & Angelika Langer
JavaMagazin 10.08

/EFF5/ JMM - Double-Check Locking
Klaus Kreft & Angelika Langer
JavaMagazin 11.08

/EFF6/ JMM - Die Initialisation-Safety-Garantie (für final Felder von primitivem Typ)
Klaus Kreft & Angelika Langer
JavaMagazin 12.08

/EFF7/ JMM - Initialisation-Safety-Garantie (für final Felder von einem Referenztyp)
Klaus Kreft & Angelika Langer
JavaMagazin 1.09

/EFF8/ JMM - Über die Gefahren allzu aggressiven Optimierens
Klaus Kreft & Angelika Langer
JavaMagazin 2.09

Java Memory Modell

Teil 12: Atomare Referenzvariablen

Klaus Kreft & Angelika Langer

Wir haben im letzten der Beiträge dieser Kolumne [EFF9] über atomare skalare Variablen gesprochen. In diesem Beitrag sehen wir uns die atomaren Referenzvariablen an.

Wir haben im letzten Beitrag gezeigt, wie man atomare Skalare aus dem Package `java.util.concurrent.atomic` als bessere `volatile` Variablen für die Optimierung von konkurrierenden Zugriffen auf skalare Variablen einsetzen kann. Erläutert haben wir es am Beispiel einer Counter-Abstraktion, in der ein Integer konkurrierend inkrementiert und dekrementiert werden kann. Da Inkrement und Dekrement keine atomaren Operationen sind, sind sie unterbrechbar. Eine Möglichkeit, die Counter-Abstraktion thread-safe zu machen, besteht nun darin, den konkurrierenden Zugriff auf diese Operationen mit Hilfe von Locks zu synchronisieren . Als Alternative zur Synchronisation kann man statt eines normalen `int` einen `AtomicInteger` verwenden. Er hat nämlich atomare Inkrement- und Dekrement-Operationen, deren Verwendung dann den Verzicht auf Synchronisation ermöglicht.

Ganz allgemein ist die Idee hinter einem atomaren Inkrement und Dekrement die sogenannte Compare-and-Swap-Sequenz, auch CAS genannt. Solch ein atomarer CAS-Befehl wird üblicherweise von dem jeweiligen Prozessor bereits unterstützt und die atomaren Variablen in Java nutzen die Möglichkeiten der Hardware, um ununterbrechbare Read-Modify-Write-Sequenzen auf Java-Objekten zu unterstützen. Solche Unterstützung bieten die atomaren, skalaren Variablen `AtomicInteger`, `AtomicLong` und `AtomicBoolean`. Nun beschränkt sich das Bedürfnis nach atomaren Read-Modify-Write-Sequenzen nicht nur auf skalare Typen. Deshalb gibt es neben den atomaren Skalaren auch noch atomare Referenzen wie `AtomicReference<V>`, `AtomicMarkableReference<V>` und `AtomicStampedReference<V>`.

Fallstudie zu atomaren Referenzen

Diese atomaren Referenzen wollen wir uns in diesem Beitrag genauer ansehen. Dazu betrachten wir ein Beispiel:

```
public final class StringUpdater { // Vorsicht: falsch !!!
    private String theString;
    private boolean changed = false;

    public void setText(String txt)  {
        theString = txt;
        changed = true;
    }
    public boolean displayText() {
        if (changed) {
            String currentText = theString;
            ... display currentText ...
            changed = false;
        }
        return true;
    }
}
```

Die Klasse `StringUpdater` enthält einen String und ein Boolesches Flag, welches anzeigt, ob der String seit der letzten Anzeige geändert wurde. Eine solche Abstraktion könnte zum Beispiel verwendet werden, um die Ergebnisse eines Temperatur-Sensors anzuzeigen: ein oder mehrere Sensor-Threads legen in dem `StringUpdater` mit Hilfe der `setText()`-Methode einen neuen Wert ab, wenn sich die Temperatur verändert hat, und markieren den String als "neu", indem das Flag gesetzt wird. Ein oder mehrere andere Threads zeigen den Wert über die `displayText()`-Methode an, aber nur, wenn er "neu" ist. Die Idee ist, dass der Wert nur angezeigt werden soll, wenn er sich geändert hat; wiederholtes Anzeigen desselben Wertes soll vermieden werden. Hingegen ist es kein Problem, wenn mal eine Änderung gar nicht angezeigt wird. Dieses Vorgehen macht Sinn, wenn das Anzeigen teuer oder aufwändig ist. Wenn die "Anzeige" beispielsweise darin besteht, dass jemanden eine Email geschickt wird mit dem neuen Wert, dann möchte man sicher nicht überflüssige Emails schicken, die gar keine neue Information enthalten. Soweit die Randbedingungen. Sehen wir uns nun an, ob die Abstraktion `StringUpdater` korrekt ist.

Sichtbarkeitsprobleme

Die oben gezeigte Lösung hat gravierende Defizite. Der erste Mangel sind Sichtbarkeitsprobleme. Es ist nicht gewährleistet, dass der Anzeige-Thread überhaupt zu sehen bekommt, was der Sensor-Thread in dem `StringUpdater` hinterlegt hat.

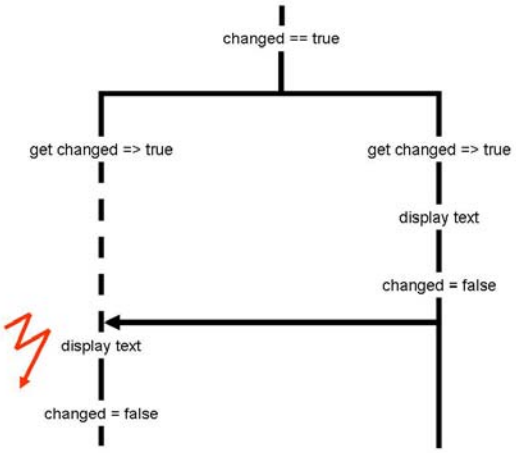
Das Sichtbarkeitsproblem könnte man lösen, indem die beiden Felder der Klasse `StringUpdater` als `volatile` deklariert werden. Das sähe dann so aus:

```
public final class StringUpdater { // Vorsicht: immer noch falsch !!!

    private volatile String theString;
```

```
    private volatile boolean changed = false;

    public void setText(String txt)  {
        theString = txt;
        changed = true;
    }
    public void displayText() {
        if (changed) {
            String currentText = theString;
            ... display currentText ...
            changed = false;
        }
    }
}
```

Race Condition

Der zweite Mangel ist damit aber nicht behoben: in der Methode `displayText()` wird das Boolesche Feld gelesen, ausgewertet und anschließend geändert. Diese Sequenz von Operationen ist unterbrechbar. Es könnte passieren (siehe Diagramm), dass ein Anzeige-Thread das Boolesche Feld `changed` liest, den Wert `true` vorfindet, deshalb den Text anzeigen will, aber vorher unterbrochen wird. Ein anderer Anzeige-Thread liest ebenfalls das Boolesche Feld und findet ebenfalls den Wert `true`. Dieser zweite Anzeige-Thread wird nicht unterbrochen und zeigt den Wert an. Wenn der erste Anzeige-Thread wieder arbeiten darf, wird er den bereits nicht mehr aktuellen Wert trotzdem anzeigen. Wir bekämen dann die redundante Mehrfach-Anzeige, die eigentlich vermieden werden sollte.



Das Problem ist, dass die Sequenz von Lesen-Auswerten-Verändern des Booleschen Feldes `changed` ununterbrechbar sein müsste. Wir brauchen also Synchronisation, um die Mehrfach-Anzeige zu verhindern. Das sähe dann so aus:

```
public final class StringUpdater {  // korrekt, aber nicht optimal

    private String theString;
    private boolean changed = false;

    public synchronized void setText(String txt) {
        theString = txt;
        changed = true;
    }

    public synchronized void displayText() {
        if (changed) {
            String currentText = theString;
            ... display currentText ...
            changed = false;
        }
    }
}
```

Jetzt ist die Lösung zwar korrekt und thread-sicher, aber wegen der Synchronisation relativ teuer, was die Performance betrifft.

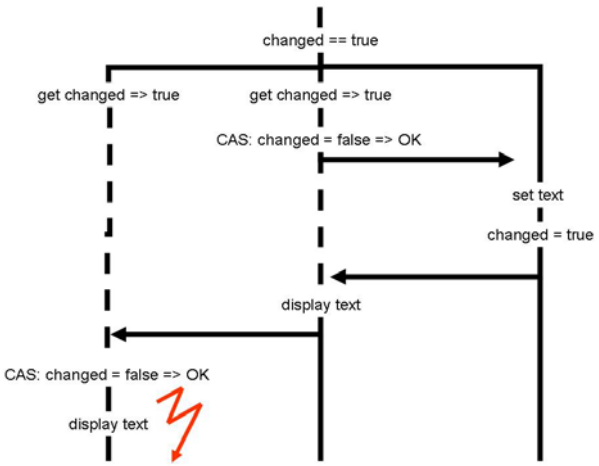
Optimierung mit Atomaren Referenzen

Wie wir gesehen haben, löst `volatile` einen Teil des Problems (das Sichtbarkeitsproblem), bietet aber nicht die Ununterbrechbarkeit der Read-Modify-Write-Sequenz, die wir brauchen. Atomare Variablen haben die ununterbrechbare Read-Modify-Write-Operationen, die den `volatile`-Variablen fehlen, und lösen außerdem die gleichen Speichereffekte aus wie `volatile`. Versuchen wir eine Lösung mit einer `AtomicReference` und einem `AtomicBoolean`.

```
public class StringUpdater {  // Vorsicht: falsch !!!

    private AtomicReference<String> theString = new AtomicReference<String>(null);
    private AtomicBoolean           changed   = new AtomicBoolean(false);

    public void setText(String txt) {
        theString.set(txt);
        changed.set(true);
    }
    public void displayText() {
        boolean chg = changed.get();
        if (chg) {
            if (changed.compareAndSet(true, false)) {
                String currentText = theString.get();
                ... display currentText ...
                return;
            } else  {
                displayText();
                return;
            }
        }
    }
}
```

Jetzt haben wir eine atomare Read-Modify-Write-Sequenz verwendet, um das Boolesche Feld changed zu bearbeiten. Wenn nun zwei Anzeige-Threads das Boolesche Feld changed lesen und beide den Wert `true` vorfinden, dann wird einer von beiden Threads erfolgreich das Boolesche Feld modifzieren und anschließend den Text anzeigen. Im jeweils anderen Thread würde das Modifizieren des Booleschen Feld changed scheitern, weil das Boolesche Feld nicht mehr den erwarteten Wert `true` hat. Das wäre in Ordnung; dann würde der betreffende Thread einfach noch einmal versuchen, das Feld erneut zu lesen, zu ändern und dann den Text anzuzeigen. Leider reicht das aber nicht aus, um die Mehrfach-Anzeigen zu verhindern.

Es könnte nämlich auch sein, dass nach dem Ändern des Booleschen Feld changed im ersten Anzeige-Thread ein Sensor-Thread den Text und das Boolesche Feld ändert (siehe Diagramm). Dann würde der eine Anzeige-Thread den neuen Text anzeigen. Der alte Text wäre verloren gegangen, aber das ist im gegebenen Anwendungsfall kein Prblem.



Danach würde aber im anderen Anzeige-Thread das Modifzieren des Booleschen Feldes erfolgreich verlaufen, weil sich der Boolesche Wert zwar in der Zwischenzeit geändert hat, aber schon wieder den erwarteten Wert `true` hat. Der neue Text würde also noch einmal angezeigt - und wir hätte dann doch eine unerwünschte Mehrfach-Anzeige.

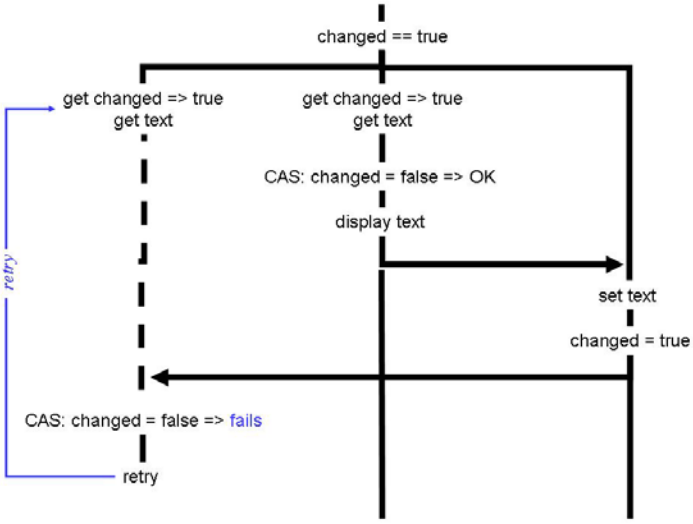
Das Problem hier ist, dass wir das Boolesche Feld changed nur ändern dürfen, wenn sich der Inhalt des Textfeldes nicht geändert hat. Es genügt nicht, wenn wir das Boolesche Feld in Isolation betrachten, sondern wir müssen hier die Kombination von String und Boolean gemeinsam verarbeiten.

Für diese Kombination von einem Objekt und einem Booleschen Wert gibt es im Package `java.util.concurrent.atomic` eine Abstraktion, nämlich die `AtomicMarkableReference<V>`. Die korrekte Lösung für unser Problem sieht also so aus:

```
public final class StringUpdater {  // endlich korrekt und effizient

    private AtomicMarkableReference<String> amr = new AtomicMarkableReference<String>(null, false);

    public void setText(String txt) {
        amr.set(txt, true);
    }
      public void displayText() {
        boolean[] changed = new boolean[1];
        String currentText = amr.get(changed);
        if (changed[0]) {
            if (amr.compareAndSet(currentText,currentText,true,false)) {
```

```
                ... display currentText ...
                return;
            } else {
                displayText();
                return;
            }
        }
    }
}
```

Jetzt kann es nicht mehr passieren, dass derselbe Text redundant mehrfach angezeigt wird. Die Compare-And-Set-Operation klappt nun nicht mehr, wenn sich der Text in der Zwischenzeit geändert hat, weil die `AtomicMarkableReference<V>` sowohl für die Referenz als auch für den Booleschen Wert prüft, ob sie die erwarteten Werte haben. In unserem Beispiel würde die CAS-Operation scheitern (siehe Diagramm) und der zweite Anzeige-Thread würde den gesamten Vorgang von "Werte lesen, vergleichen und ändern" erneut versuchen.



Wir haben anhand der Fallstudie illustriert, dass die Verwendung von atomaren Referenzen zur Verweidung von Synchronisation verwendet werden kann. Die atomaren Referenzen haben diesselben Speichereffekte wie `volatile` Referenzen und haben zusätzlich ununterbrechbare CAS-Operationen. In diesem Sinne sind die atomaren Referenzen eine logische Verallgemeinerung von `volatile` Referenzen. Wie das Beispiel aber auch gezeigt hat, kann man bei Verwendung von atomaren Referenzen natürlich Fehler machen. Die Verwendung von atomaren Variablen funktioniert nur dann als Ersatz für Synchronisation, wenn sich die kritischen Zugriffe auf eine einzige Variable beziehen. Wir haben in der Fallstudie den Fehler gemacht, dass wir zunächst zwei atomare Variablen verwendet haben. Auch das kann in Spezialfällen mal korrekt sein, nämlich wenn die beiden Variablen voneinander unabhängig sind. Sobald jedoch die beiden Variablen in einer Beziehung zueinander stehen und konsistent zueinander sein müssen, dann ist die Verwendung von zwei atomaren Variablen falsch. Wichtig ist beim Programmieren mit atomaren Variablen, dass man sich wirklich auf eine einzige (dann atomare) Variable beschränken kann. In unserem Beispiel war die `AtomicMarkableReference<V>`, also die Kombination von Referenz und Boolschem Wert, das richtige Instrument.

Atomare Referenzen - Übersicht

Werfen wir zu Schluss noch einen Blick auf die übrigen atomaren Referenz-Abstraktionen. Das Package `java.util.concurrent.atomic` hat, wie schon erwähnt, gleich 3 Arten von atomaren Referenzen zu bieten: `AtomicReference<V>`, `AtomicMarkableReference<V>` und `AtomicStampedReference<V>`. Die `AtomicReference<V>` ist die "normale" atomare Referenz auf ein Objekt. Die `AtomicMarkableReference<V>` ist eine Referenz auf ein Objekt zusammen mit einem Booleschen Wert; dabei ist der Boolesche Wert typischerweise sowas wie eine Gültigkeitsinformation zu dem Objekt. Die `AtomicStampedReference<V>` ist eine Referenz zusammen mit einem Integer-Wert; dabei ist der Integer häufig eine Art Versionsnummer für das referenzierte Objekt.

Alle atomaren Referenzen haben dieselben Speichereffekte wir `volatile`, damit es keine Sichtbarkeitsprobleme gibt. Die wesentlichen Methoden und deren Speichereffekte sind:

- `get()` entspricht dem Lesen einer `volatile`-Variablen
- `set()` entspricht dem schreibenden Zugriff auf eine `volatile`-Variable
- `compareAndSet()` entspricht dem lesenden und schreibenden Zugriff auf eine `volatile`-Variable
- `weakCompareAndSet()` ist lediglich ununterbrechbar, hat aber keine Speichereffekte, entspricht also dem Zugriff auf eine non-`volatile`-Variable (dient zur Optimierung, wenn man zum

Beispiel weiss, dass die relevanten Speichereffekte aufgrund anderer Operationen ausgeführt werden)

Neben den oben genannten Abstraktionen gibt es noch die schon im letzten Beitrag erwähnten atomaren Skalare `AtomicBoolean`, `AtomicInteger`, und `AtomicLong`. Sie haben zusätzliche Inkrement- und Dekrement-Methoden.

Außerdem gibt es atomare Arrays: `AtomicIntegerArray`, `AtomicLongArray` und `AtomicReferenceArray`. Sie bieten atomare Zugriffe auf die Elemente des Arrays - eine Funktionalität, die es für normale Arrays nicht gibt, weil man mit den Java-Sprachmitteln gar nicht deklarieren kann, dass man ein Array von `volatile`-Elementen haben möchte.

Daneben gibt es noch die sogenannten atomaren *Updater*: `AtomicReferenceFieldUpdater<T,V>`, `AtomicIntegerFieldUpdater<T>` und `AtomicLongFieldUpdater<T>`. Das sind Abstraktionen, die atomare CAS-Zugriffe auf `volatile`-Felder von Objekten anbieten. Während die atomaren Variablen jeweils ein Objekt (oder skalaren Wert, Array, Objekt + Boolean, Objekt + Version) kapseln und für das Objekt atomare CAS-Zugriffe ermöglichen, bieten die Updater atomare CAS-Zugriffe für `volatile`-Felder von Objekten. Dabei kann man einen einzigen Updater verwenden, um auf ein bestimmtes Feld in verschiedenen Objekten zuzugreifen. Man hat also zum Beispiel nur einen Updater für das `next`-Feld einer Klasse `Node` und kann mit diesem einen Updater auf das `next`-Feld aller Objekte vom Typ `Node` zugreifen. Das spart Resourcen, weil nicht jedes einzelne `next`-Feld in jedem einzelen `Node`-Objekt in eine `AtomicReference<Node>` eingepackt werden muss. Dafür sind die Garantien schwächer, weil neben den sicheren Zugriffen über den Updater immer noch direkt Zugriffe auf die `volatile`-Fehler gemacht werden können, die dann nicht atomar und ohne Speichereffekte sind.

Praxis-Relevanz

Atomare Variablen und Updater sind neben `volatile` eine weitere Möglichkeit, konkurrierende Zugriffe auf gemeinsam verwendete Objekte zu optimieren, indem man anstelle von synchronisierten Zugriffen atomare Zugriffe macht. Dieses Vermeiden von Synchronisation wird oft als *Lockfree-Programming* bezeichnet. Lockfree-Programming ist aber keine Technik für den Hausgebrauch, denn der Verzicht auf Synchronisation ist nur speziellen Fällen möglich. Voraussetzung für die Nutzung von atomaren Variablen (und den Verzicht auf Synchronisation) ist, dass die wesentlichen Zugriffe sich auf eine *einzige* Variable beschränken lassen, auf die man dann atomar zugreifen kann. Sobald mehrere Variablen konsistent zueinander geändert werden müssen, reichen atomare Variablen nicht aus; Synchronisation ist dann zwingend erforderlich.

Beispiele für den Einsatz atomarer Variablen findet man im JDK: die Klasse `ConcurrentLinkedQueue` wie auch die SkipList-Implementierungen sind Beispiele für Abstraktionen, die thread-sicher sind und die Thread-Sicherheit ohne die Verwendung von Synchronisation erreicht. Die Implementierung der `ConcurrentLinkedQueue` zum Beispiel verwendet atomare Updater, um den Link auf den jeweils nächsten Knoten in der Liste zu verwalten. Die Algorithmen, die für diese Art von Programmierung benötigt werden, sind in der Regeln nicht trivial und typischerweise das Ergebnis jahrelanger Forschung. Es gibt einige Standardalgorithmen für Stacks und Listen. Wer sich für Details interessiert, der muss sich mit der entsprechenden Fachliteratur beschäftigen. Eine gute (wenn auch sehr, sehr knappe) Einführung in die Prinzipien des Lockfree-Programming findet man in Kapitel 15.4. "Nonblocking Algorithms" in "Java Concurrency in Practice" von Brian Goetz (siehe [JCP]).

Insgesamt ist die Verwendung von atomaren Variablen kein Instrument, mit dem man in beliebigen Situationen optimieren kann, indem synchronisierte Zugriffe durch atomare Zugriffe ersetzt werden.

Zusammenfassung

In diesem Beitrag haben wir uns die atomaren Referenzvariablen angesehen. Sie sind eine logische Verallgemeinerung von `volatile` Referenzen. Die atomaren Referenzen haben diesselben Speichereffekte wie `volatile` Referenzen und stellen zusätzlich ununterbrechbare CAS-Operationen zur Verfügung. Atomare Referenzen ermöglichen Optimierungen, die darin bestehen, dass auf Synchronisation mit Locks verzichtet wird und stattdessen *lockfree* programmiert wird. Wichtig ist beim Programmieren mit atomaren Variablen, dass man die kritischen Zugriffe auf eine einzige Variable beschränken kann.

Verweise

/EFF1/ JMM - Einführung: Wozu braucht man volatile?
Klaus Kreft & Angelika Langer
JavaMagazin 7.08

/EFF2/ JMM - Überblick über das Java Memory Modell (JMM)
Klaus Kreft & Angelika Langer
JavaMagazin 8.08

/EFF3/ JMM - Die Kosten der Synchronisation
Klaus Kreft & Angelika Langer
JavaMagazin 9.08

/EFF4/ JMM - Speichereffekte von volatile
Klaus Kreft & Angelika Langer
JavaMagazin 10.08

/EFF5/ JMM - Double-Check Locking
Klaus Kreft & Angelika Langer
JavaMagazin 11.08

/EFF6/ JMM - Die Initialisation-Safety-Garantie
Klaus Kreft & Angelika Langer
JavaMagazin 12.08

/EFF7/ JMM - Über die Gefahren allzu aggressiven Optimierens
Klaus Kreft & Angelika Langer
JavaMagazin 1.09

/EFF8/ JMM - Atomare skalare Variablen
Klaus Kreft & Angelika Langer
JavaMagazin 2.09

/JCP/ Java Concurrency in Practice
Brian Goetz et.al.
Addison-Wesley 2006

AtomicReference vs. AtomicMarkableReference

Die `AtomicMarkableReference<V>` und `AtomicStampedReference<V>` sind reine Convenience-Klassen. Man kann die Kombination von einer Referenz mit einem Boolean oder einem Integer auch selber bauen. Um zu illustrieren, wie die `AtomicMarkableReference<V>` funktioniert, haben wir unser Beispiel hier noch einmal mit einer einfachen `AtomicReference<V>` gebaut.

Man braucht eine weitere Klasse, die die Kombination von Referenz und Boolean repräsentiert; wir haben sie `MarkedText` genannt. Dann sieht die Lösung so aus:

```
public final class StringUpdater {  // korrekt und effizient,
                                    // aber mit AtomicReference anstelle von AtomicMarkableReference
    private static class MarkedText {
        private final String text;
        private final boolean flag;

        public MarkedText(String txt) {
            this.text = txt;
            flag = true;
        }
        public MarkedText(MarkedText mt) {
            this.text = mt.text;
            flag = false;
        }
        public String getText() {
            return text;
        }
        public boolean isNew() {
            return flag;
```

```
            }
        }

        private AtomicReference<MarkedText> theText
        = new AtomicReference<MarkedText>(new MarkedText(new MarkedText((String)null)));

        public void setText(String txt) {
            theText.set(new MarkedText(txt));
        }
        public void displayText() {
            MarkedText currentText = theText.get();
            if (currentText.isNew()) {
                if (theText.compareAndSet(currentText, new MarkedText(currentText))) {
                    ... display currentText.getText() ...
                    return;
                }
                else {
                    displayText();
                    return;
                }
            }
        }
    }
```

Java Memory Modell

Teil 13: CopyOnWriteArrayList

Klaus Kreft & Angelika Langer

In diesem Beitrag wollen wir die `CopyOnWriteArrayList` aus dem JDK hernehmen und an ihrem Beispiel das Zusammenspiel der Speichereffekte von Synchronisation, `volatile`, und atomaren Referenzen noch einmal abschließend erläutern.

In unserer Serie über das Java Memory Modells haben wir eine Reihe von Aspekten angeschaut, mit denen sich die Verwendung von Synchronisation reduzieren läßt. Dabei soll mit der Vermeidung der Synchronisation in der Regel die Performance der betroffenen Abstraktion verbessert und der Durchsatz erhöht werden, weil ohne Synchronisation mehrere Threads ungehindert parallel arbeiten können, ohne aufeinander warten zu müssen. Wir haben die Effekte von `final`-, `volatile`- und atomaren Variablen betrachtet und an kleineren Beispielen demonstriert. Dieses Mal wollen wir zeigen, wie diese Sprachmittel in einer realistischen Abstraktion in der Praxis genutzt werden. Dazu wollen wir uns die Implementierung der `CopyOnWriteArrayList` aus dem JDK-Package `java.util.concurrent` genauer ansehen, weil sie ein geschicktes Zusammenspiel von `volatile` und Synchronisation zeigt.

Was ist eine CopyOnWriteArrayList?

Die `CopyOnWriteArrayList` ist eine thread-sichere Implementierung einer Liste, die man - ähnlich wie eine synchonisierte Liste - mehreren Thread zum gemeinsamen, konkurrierenden Zugriff zur Verfügung stellen kann. Während die synchronisierte Liste, die man über `Collections.synchronizedList()` bekommt, alle Zugriffsmethoden per Synchronisation auf dem Mutex des Listenobjekts schützt, ist die `CopyOnWriteArrayList` dahingehend optimiert, dass sie einen schnelleren Lesezugriff bietet, der ganz ohne Synchronisation auskommt. Im Gegenzug sind die modifizierenden Methoden relativ teuer, weil sie die Modifikation auf einer Kopie des unterliegenden Arrays ausführen und anschließend das alte gegen das neue modifizierte Array austauschen. Man bezahlt also die schnellen Lesezugriffe durch das relativ teure Kopieren bei den Schreibzugriffen. Wenn Modifikationen an der Liste selten und Lesezugriffe wesentlich häufiger vorkommen, dann ist die `CopyOnWriteArrayList` performanter als die normale synchronisierte Liste.

Wie funktioniert eine CopyOnWriteArrayList?

Die `CopyOnWriteArrayList` besteht aus einem Array, in dem die Referenzen auf die Elemente der Liste abgelegt sind. Die wesentliche Idee bei der `CopyOnWriteArrayList` ist, dass dieses Array unveränderlich ist. Es wird von keiner Methode der `CopyOnWriteArrayList` modifiziert und deshalb können alle lesenden Zugriffe ohne Synchronisation erfolgen. Hier ist ein Ausschnitt aus der Implementierung, der zunächst einmal nur eine lesende Methode, nämlich die `get()`-Methode, und die relevanten Felder der `CopyOnWriteArrayList` zeigt:

```
public class CopyOnWriteArrayList<E> implements List<E> {

    private volatile transient Object[] array;

    private final Object[] getArray() {
        return array;
    }

    public E get(int index) {
        return (E)(getArray()[index]);
    }
}
```

Man sieht, dass die Referenz auf das Array `volatile` ist und dass die `get()`-Methode über diese Referenz auf das benötigte Array-Element zugreift. Man sieht außerdem, dass die `get()`-Methode nicht synchronisiert ist. Sie kann also gleichzeitig mit anderen lesenden oder modifizierenden Methoden auf das Array der `CopyOnWriteArrayList` zugreifen. Warum das keine Race Conditions ergibt, nicht einmal wenn die `get()`-Methode gleichzeitig mit einer modifizierenden Methode wie `set()` abläuft, sehen wir uns später im Zusammenhang mit der `set()`-Methode an.

Das Fehlen jeglicher Synchronisation bedeutet, dass wir in der `get()`-Methode einen unsynchronisierten Zugriff auf die Array-Referenz haben. Bei unsynchronisierten Zugriffen stellt sich stets die Frage nach der Sichtbarkeit von Modifikationen, die ein anderer Thread gemacht hat. Kann die `get()`-Methode alle Modifikationen am Array sehen, die beispielsweise eine `set()`-Methode zuvor in einem anderen Thread ausgelöst hat?

Die Referenz auf das Array der `CopyOnWriteArrayList` ist als `volatile` deklariert und die `get()`-Methode greift lesend auf die Referenz zu (über die Hilfs-Methode `getArray()`). Weil die Array-Referenz `volatile` ist, ist garantiert, dass die `get()`-Methode alle Modifikationen sehen kann, die ein anderer Thread vor einem schreibenden Zugriff auf die Array-Referenz

gemacht hat. Sehen wir uns also eine modifizierende Methode näher an. Hier ist ein größerer Ausschnitt aus der Implementierung, der zusätzlich zur `get()`-Methode auch die `set()`-Methode zeigt:

```
public class CopyOnWriteArrayList<E> implements List<E> {

    /** The lock protecting all mutators */
    transient final ReentrantLock lock = new ReentrantLock();

    /** The array, accessed only via getArray/setArray. */
    private volatile transient Object[] array;

    private final Object[] getArray() {
        return array;
    }

    private final void setArray(Object[] a) {
        array = a;
    }

    public E get(int index) {
        return (E)(getArray()[index]);
    }

    public E set(int index, E element) {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            Object[] elements = getArray();
            Object oldValue = elements[index];

            if (oldValue != element) {
               int len = elements.length;
               Object[] newElements = Arrays.copyOf(elements, len); // 1
               newElements[index] = element;                        // 2
               setArray(newElements);                               // 3
            } else {
               // Not quite a no-op; ensures volatile write semantics
               setArray(elements);                                  // 3
            }
            return (E)oldValue;
        } finally {
            lock.unlock();
        }
    }
}
```

Sehen wir uns als erstes einmal die Funktionalität der `set()`-Methode an. Sie kopiert das Array (siehe Zeile //1), modifiziert die Kopie (siehe //2) und legt am Ende die Adresse der Kopie in der `volatile` Referenzvariablen `array` ab (siehe //3). Dieser schreibende Zugriff (über die Methode `setArray()`) auf die `volatile` Referenz löst eine Memory Barrier aus, bei der alle Modifikationen, die in der `set()`-Methode bis dahin vorgenommen wurden, im Hauptspeicher sichtbar gemacht werden für Methoden die später lesend auf die `volatile` Referenzvariable `array` zugreifen. Hierbei ist wichtig, dass der schreibende Zugriff auf die `volatile` Array-Referenz erst ganz am Ende nach der Modifikation des kopierten Arrays erfolgt, sonst wäre nicht gewährleistet, dass neben der Array-Adresse auch die Array-Elemente für die anderen Threads sichtbar werden.

Man kann nun auch sehen, warum der konkurriende unsynchronisierte Ablauf von `set()`- und `get()`-Methode keine Race Condition ist. Das Array, das die `get()`-Methode liest, wird von der `set()`-Methode nicht verändert, denn sie verändert nur eine lokale Kopie des Arrays. Die einzige Modifikation, die die `set()`-Methode an den Feldern der `CopyOnWriteArrayList` vornimmt, ist die Zuweisung der Adresse des neu erstellten Arrays an die Referenzvariable `array` (siehe //3). Diese Adresszuweisung ist atomar, also unkritisch, weil sie ununterbrechbar ist. Die `get()`-Methode erwischt also die Adresse des unveränderten Arrays vor dem Aufruf von `setArray()` oder die Adresse der modifizierten Kopie nach dem Aufruf von `setArray()`. Auf irgendwelche inkonsistenten Zwischenzustände des Arrays hat die `get()`-Methode keinen Zugriff.

Der konkurrierende Ablauf von zwei `set()`-Methode wäre hingegen sehr wohl eine Race Condition. Wenn sich beide `set()`-Methoden die Adresse des Arrays holen, beide eine lokale Kopie anlegen und ihre jeweilige Modifikation an der lokalen Kopie machen, dann wird die eine `set()`-Methode die Modifikation der anderen `set()`-Methode überschreiben, wenn sie die Referenzvariable `array` überschreibt. Nach einem unsynchronisierten Ablauf beider `set()`-Methoden bliebe nur eine der beiden Modifikationen übrig; wir hätte also ein *Lost-Update*-Problem. Die modifizierenden Methoden der `CopyOnWriteArrayList` sind deshalb alle gegeneinander synchronisiert. Dazu wird ein explizites Lock verwendet.

Die übrigen lesenden und modifizierenden Methoden der `CopyOnWriteArrayList` sehen analog aus. Interessant ist noch der Iterator. Die Beschreibung sagt, der Iterator iteriert über die Liste, so wie im Augenblick der Konstruktion des Iterators ausgesehen hat. Deshalb ist von einem *Snapshot* die Rede. Was hat es mit dem Snapshot auf sich?

Wie funktioniert der Iterator der CopyOnWriteArrayList

Der Iterator der `CopyOnWriteArrayList` ist ein reiner Lese-Iterator. Modifikationen läßt er nicht zu; modifizierende Iterator-Methoden wie `remove()` werfen immer eine `UnsupportedOperationException`. Sämtliche Methoden des Iterators sind unsynchronisiert, das heißt, der Iterator iteriert konkurrierend mit lesenden und schreibenden Methoden und anderen Iteratoren. Er ist - anders als der Iterator einer synchronisierten Liste - kein *Fast-Fail-Iterator* und wirft keine `ConcurrentModificationException`, wenn während der Iteratierung Modifikationen an der Liste vorgenommen werden.

Hier ist ein Auszug aus der Implementierung des Iterators der `CopyOnWriteArrayList`:

```
public class CopyOnWriteArrayList<E>

    private volatile transient Object[] array;

    private final Object[] getArray() {
        return array;
    }
    public Iterator<E> iterator() {
        return new COWIterator<E>(getArray(), 0);
    }

    private static class COWIterator<E> implements ListIterator<E> {
        /* Snapshot of the array **/
        private final Object[] snapshot;
        /* Index of element to be returned by subsequent call to next.*/
        private int cursor;

        private COWIterator(Object[] elements, int initialCursor) {
            cursor = initialCursor;
            snapshot = elements;
        }

        public boolean hasNext() {
            return cursor < snapshot.length;
        }

        public E next() {
            if (! hasNext())
                throw new NoSuchElementException();
            return (E) snapshot[cursor++];
        }

        /* Not supported. Always throws UnsupportedOperationException. */
        public void remove() {
            throw new UnsupportedOperationException();
        }
       ...
   }
   ...
}
```

Der Iterator enthält zwei Felder: die Referenz auf das Array der `CopyOnWriteArrayList` und einen Index, der seine Anfangsposition repräsentiert. Beide Felder werden bei der Konstruktion des Iterators initialisiert. Das heißt, der Iterator iteriert über das Array, das zum Zeitpunkt der Konstruktion des Iterators gerade aktuell war. Solange die `CopyOnWriteArrayList` nicht modifiziert wird, ist klar, dass die unsynchronisierte Iterierung kein Problem ist und keine Race Condition produziert. Wie ist das aber, wenn die `CopyOnWriteArrayList` modifiziert wird, beispielsweise mit der `set()`-Methode?

Wie wir schon gesehen haben, legt die `set()`-Methode eine Kopie des Arrays an, modifiziert die Kopie und überschreibt dann die Array-Referenz in der `CopyOnWriteArrayList` mit der Adresse der Kopie. Der Iterator bekommt davon nichts mit. Er wird weiterhin auf dem alten Array iterieren. In diesem Sinne arbeitet der Iterator der `CopyOnWriteArrayList` auf einem Snapshot und dieser Snapshot ist, nachdem eine Änderung an der Liste vorgenommen wurde, eine "alte" Kopie der Liste.

Weil das Array der `CopyOnWriteArrayList` nie verändert wird, wird im Iterator keine Synchronisation gebraucht. Auch der konkurrierende Ablauf einer Iteration und einer Modifikation der Liste ist kein Problem. Auf diese Weise hat die

`CopyOnWriteArrayList` einen höchst effizienten Iterator, der ein Maximum an konkurrierenden Zugriffen auf die Liste zuläßt.

Allerdings hat der Iterator der `CopyOnWriteArrayList` eine andere Semantik als der Fast-Fail-Iterator einer synchronisierten Liste. Er zeigt unter Umständen einen alten Stand (*stale value*) der Liste - eine Situation, die beim Iterieren über eine synchronisierte Liste nicht auftreten kann. Bei der synchronisierten Liste ist die konkurrierende Modifikation ein Fehler, der sich in der `ConcurrentModificationException` des Iterators ausdrückt. Bei der `CopyOnWriteArrayList` ist die konkurrierende Modifikation hingegen ein Feature.

Eine andere Anwendung desselben Prinzips

Die grundlegende Idee der Implementierung der `CopyOnWriteArrayList` besteht darin, dass sich das Array selbst nie ändert; es wird lediglich bei Bedarf durch ein neues ersetzt. Betrachten wir zur Illustration dieses Prinzip ein weiteres Beispiel: ein Interval. Hier ein Auszug aus einer denkbaren Implementierung, die aber nicht thread-safe ist:

```
public class Interval {  // Vorsicht: nicht thread-safe !!!
  // INVARIANT: lower <= upper
  private volatile int lower = 0
  private volatile int upper = 0;

  public void setLower(int i) {
        if (i > upper.)
            throw new IllegalArgumentException(
                    "can't set lower to " + i + " > upper");
        lower = i;
  }

    public int getLower() {
        return lower;
    }

  public boolean isInRange(int i) {
        return (i >= lower && i <= upper);
  }
  …
}
```

Diese Implementierung ist nicht thread-safe, weil die beiden Felder `lower` und `upper` eine Invariante bilden: `lower` muss immer kleiner als upper sein. Für den Erhalt der Invariante ist es erforderlich, dass Read-Check-Modify-Sequenzen wie in der `setLower()`-Methode ununterbrechbar sind: erst wird der aktuelle Wert von `upper` gelesen, dann wird er mit dem neuen Wert von `lower` verglichen, und nur wenn alles in Ordnung ist, wird der neue Wert für `lower` gesetzt. Wenn diese Sequenz unterbrochen würde, könnten sinnlose Intervalle entstehen, bei denen die Untergrenze größer als die Obergrenze ist.

Um diese Interval-Klasse thread-safe zu machen, kann man dasselbe Prinzip anwenden, das auch in der `CopyOnWriteArrayList` verwendet wird: man sorgt dafür, dass die eigentlichen Daten des Intervals (also das Paar von Unter- und Obergrenze) unveränderlich sind und lediglich bei Bedarf neue Daten erzeugt und gegen die alten ausgetauscht werden.

Hier ist eine korrigierte Fassung:

```
@ThreadSafe
public class Interval {

    @Immutable
    private static class IntPair {
        private final int lower;
        private final int upper;
        public IntPair(int l, int u) {
            lower =l;
            upper =u;
        }
    }

    private final AtomicReference<IntPair> values =
        new AtomicReference<IntPair>(new IntPair(0, 0));


    public void setLower(int i) {
        while (true) {
            IntPair oldv = values.get();
```

```
            if (i > oldv.upper)
                throw new IllegalArgumentException(
                   "Can't set lower to " + i + " > upper");
            IntPair newv = new IntPair(i, oldv.upper);
            if (values.compareAndSet(oldv, newv))
                return;
        }
    }

    public int getLower() {
        return values.get().lower;
    }

    public boolean isInRange(int i) {
        IntPair v = values.get();
        return (i >= v.lower && i <= v.upper);
    }

    …
}
```

Das Paar von Unter- und Obergrenze wird nie verändert. Wenn das Interval geändert werden soll, wie etwa in der `setLower()`-Methode, dann wird ein neues Paar erzeugt und gegen das alte ausgetauscht. Dazu genügt es aber nicht, lediglich die Referenz auf das Paar auszutauschen, sondern wir brauchen wegen der Invarianten noch immer die ununterbrechbare Read-Check-Modify-Sequenz - ähnlich wie man in der `set()`-Methode der `CopyOnWriteArrayList` eine ununterbrechbare Read-Calculate-Modify-Sequenz brauchte. Das könnte man wie in der `CopyOnWriteArrayList` mit Synchronisation erreichen, aber um eine effizientere Lösung zu bekommen, verwenden wir eine `AtomicReference`, die eine ununterbrechbare CAS (= compare-and-swap) Methode hat.

Das Verhalten ist bei dieser Lösung ähnlich wie bei der `CopyOnWriteArrayList`: konkurreirende Lesezugriffe sind sowieso kein Problem; Modifikationen, die konkurrierend zu lesenden Zugriffen passieren, sind auch kein Problem. Der Leser sieht stets ein gültiges Interval, das zu irgendeinem Zeitpunkt einmal existiert hat (und niemals ein fehlerhaftes, bei dem die Untergrenze größer als die Obergrenze ist); schlimmstenfalls sieht er einen älteren Zustand des Intervals (vor der letzten aktuellen Änderung). Sogar konkurrierende Schreibzugriffe kommen wegen der `AtomicReference` ohne Synchronisation aus; schlimmstenfalls muss ein Thread seinen Modifikationsversuch mehrmals wiederholen.

Zusammenfassung

Die Implementierung der `CopyOnWriteArrayList` illustriert, wie man den Bedarf an Synchronisatoin senken und die Zahl der konkurrierenden Zugriffe auf eine Abstraktion erhöhen kann. Wir haben dasselbe Prinzip auch noch am Beispiel einer Interval-Klasse gezeigt. Das Idiom hat folgende Elemente:

- Die Implementierung der Abstraktion hält lediglich eine Referenz auf alle Daten der Abstraktion und diese Daten sind unveränderlich; lediglich die Referenz kann sich ändern. Die Referenz ist wegen der Sichtbarkeitsanforderungen entweder `volatile` oder atomar.
- Alle Modifikationen werden auf einer Kopie der aktuellen Daten gemacht und am Ende wird die Referenz auf die alten Daten gegen eine Referenz auf die neuen Daten ausgetauscht. Weil die eigentlichen Daten der Abstraktion unveränderlich sind, können lesende Zugriffe miteinander und mit den modifizierenden Zugriffen zusammen parallel ohne Synchronisation ablaufen. Diesen Gewinn an Parallelität bezahlt man mit den Kosten für das Kopieren. Das Idiom lohnt sich also in erster Linie bei Abstraktionen, die wesentlich häufiger gelesen als modifiziert werden.
- Für die Modifikationen ist es erforderlich, dass eine Sequenz von "Lesen der aktuellen Daten" - "Bewerten der gelesenen Daten (Vergleich, Berechnung, etc.) - "Verändern der Daten" ununterbrechbar abläuft. Die Atomarität kann entweder per Synchronisation erreicht werden, wie bei der `CopyOnWriteArrayList`, oder mit Hilfe einer atomaren Referenz, wie im Interval-Beispiel.

Damit haben wir an einem realen Beispiel aus dem JDK gesehen, wie die Elemente des Java Memory Modells in der Praxis verwendet werden können.

Verweise

/EFF1/ JMM - Speichereffekte von volatile
Klaus Kreft & Angelika Langer
JavaMagazin 10.08

/EFF2/ JMM - Atomare Referenz-Variablen
Klaus Kreft & Angelika Langer
JavaMagazin 10.09

Training by Angelika Langer & Klaus Kreft, 1995 - 2015

Java

Seminars

Course Curriculum

Overview

JAVA

Effective Java	Discusses traps and pitfalls in Java as well as advanced Java features such as inner classes, generics and reflection.
Concurrent Java	In-depth coverage of multithread programming in Java, up-to-date, including the fork.join-framework.
High-Performance Java	Programming for performance: covers programming, monitoring, profiling, and tuning techniques.
Lambdas & Streams	In-depth coverage of programming with lambdas and streams in Java 8.
Java 8	An overview of new language features and JDK extensions in Java 8.
Java For Beginners	An introduction to Java for programmers without any experience in an object-oriented language

Effective Java

- *Best Practice Programming Idioms*
- *Avoiding Traps and Pitfalls*
- *Advanced Language Features*

Course Description

Java opens quite a number of trapdoors despite of its alleged reputation as an "easy-to-learn" programming language. For the more ambitious and professional Java programmer it is essential to understand the subtleties of the language and to know what to expect of Java, which language feature to use for which reason, and what to avoid in order to improve the quality of the resulting program.

Objective

This course aims to shed some light on the more "interesting" areas of Java: it addresses pitfalls and helps avoiding common Java errors and it explains less commonly known, yet indispensable language features.

Audience

Professional programmers who want to explore Java in greater depth.

Prerequisite

The seminar should ideally be attended after some initial experience with Java and builds on elementary Java knowledge.

Duration

3-5 days

Language

English or German

Course Overview

1. CONSTRUCTION AND FINALIZATION
   - Construction & Initialization
   - Destruction vs. Finalization
2. OBJECT INFRASTRUCTURE
   - Object Equality and Comparison
   - Copying Objects (clone())
3. IMMUTABILITY
   - final vs. Constness
   - Dual Class Idiom
4. INNER CLASSES
   - Nested Classes & Non-Static Inner Classes
   - Local & Anonymous Classes
5. LAMBDAS & STREAMS
   - Overview of Lambdas & Method References
   - Overview of Stream API
6. GENERICS
   - Generic Types and Methods
   - Type Erasure and WIldcards
7. REFLECTION & DYNAMIC PROXIES
   - Reflection API
   - Dynamic Proxies
8. WEAK & SOFT REFERENCES
   - Soft References & Caching
   - Weak References & Memory Leaks
9. “NEW” LANGUAGE FEATURES
   - Enum Types, Varargs, Autoboxing Pitfalls
   - Project “Coin” (JDK 7)
   - Default & Static Interface Methods
10. ANNOTATIONS & COMPILER PLUGINS
    - Declaring and Using Annotation Types
    - Compiler-Plugins for Annotation Processing
12. SERIALIZATION
    - Default and Custom Serialization
    - Object Stream Support
13. CLASS LOADING
    - Class Loader Basics
    - Custom Class Loader

Location

Open enrollment courses are conducted in collaboration with local partner companies. Public courses are offered at regular intervals.

The course schedule can be found at http://www.AngelikaLanger.com/Courses/Schedule.html.

Courses can also be held at your company site. Duration and content will then be tailored to your specific needs and prerequisites.

Contact

For availability and enrollment send e-mail to contact@AngelikaLanger.com. For further information see http://www.AngelikaLanger.com/Courses.html.

Concurrent Java

- *Multithread Programming in Java*
- *Synchronisation and Thread-Safety Issues*
- *Task and Thread Control*

Course Description

"Writing correct programs is hard; writing correct concurrent programs is harder. There are simply more things that can go wrong in a concurrent program than in a sequential one." (Brian Goetz)

Java supports concurrent programming with multiple threads directly by means of built-in language features. This is one of the many reasons why Java programmers need to grasp the essence of concurrent programming no matter whether they actively create multiple threads or simply prepare their classes for safe use by multi threads.

Professional Java programmers in general should be familiar with terms such as race conditions, synchronization, deadlocks, thread-safety, memory model, etc.

Objective

This seminar aims to give a comprehensive introduction to concurrent programming in Java, exposes students to programming techniques and idioms that have been proven useful in practice, and alerts them to commonly known pitfalls.

Audience

Software engineers who intend to build applications that are executed concurrently in multiple threads and programmers who need to prepare their classes for use in multi-threaded environments.

Prerequisite

Sound knowledge of Java. Experience with concurrent programming helpful, but not strictly required.

Duration

4 days

Language

English or German

Course Overview

1. MULTI-THREADING BASICS
   - Thread Safety
   - Thread Creation
2. CONCURRENCY CONTROL
   - Implicit and Explicit Locks
   - State-Dependent Operations
   - Synchronizers & - Blocking Queues
   - Synchronized vs. Concurrent Collections
   - Thread Local Memory
   - Atomic Scalars
3. TESTING CONCURRENT PROGRAMS
   - Atomic Invariant Checks
   - Test for Liveness & Blocking State
   - Test for Thread Safety & Race Conditions
   - Tools, Harnesses, Analyzers
4. THREAD CONTROL
   - Thread
   - ThreadPoolExecutor
     - Principles & Configuration
     - Scheduled Tasks
   - ForkJoinPool
     - Principles & Configuration
     - Managed Blocker

   CompletableFuture
5. ADVANCED TOPICS
   - JMM - Java Memory Model
   - Double Checked Locking

Location

Open enrollment courses are conducted in collaboration with local partner companies. Public courses are offered at regular intervals.

The course schedule can be found at http://www.AngelikaLanger.com/Courses/Schedule.html.

Courses can also be held at your company site. Duration and content will then be tailored to your specific needs and prerequisites.

Contact

For availability and enrollment send e-mail to contact@AngelikaLanger.com. For further information see http://www.AngelikaLanger.com/Courses.html.

High-Performance Java

- *Programming for Better Performance*
- *Avoiding Performance Pitfalls*
- *Profiling and Tuning Techniques*

Course Description

Application performance is an important issue in every software development project. This seminar explores strategies for improving the performance of Java programs. The focus is on core Java – the language itself and its platform libraries (JDK). Attendants study programming techniques for efficient use of the Java language and its runtime environment, including tips for avoiding performance pitfalls. Benchmarking and profiling techniques are explained and practiced in hands-on labs. The goal is to enable attendants to identify and analyze performance bottlenecks and apply appropriate tuning techniques.

Objective

This course aims to explain best practice programming techniques for high-performance-Java programs and practices performance profiling and analysis including appropriate tuning techniques.

Audience

Professional programmers with an interest in producing high-performance Java software. The focus is on programming and tuning, not on high-level design and testing.

Prerequisite

The seminar should ideally be attended by Java programmers with sound Java knowledge.

Duration

4 days

Language

English or German

Course Overview

1. PERFORMANCE & DEVELOPMENT
   a) Programming
      - Elementary Issues
      - Data Structures & I/O
   b) Benchmarking
      - Benchmark Pitfalls
      - JIT Compilation
      - Statistics
      - Caliper & JMH
2. PROFILING, MONITORING & TUNING
   a) JVM Profiling & Tuning
      - Tool Architecture
      - Profiling Strategies and Tactics
      - Performance Hot Spots
      - Memory Allocation Hot Spots
   b) Memory Leak Detection
      - Live Detection Strategies
      - Post-Mortem Analysis
   c) GC Profiling (Sun/Oracle)
      - Classic Garbage Collection
      - Garbage First (G1) Collector
      - GC Profiling
      - Throughput Tuning
      - Pause Tuning
      - G1 Tuning
   d) JVM Monitoring

Location

Open enrollment courses are conducted in collaboration with local partner companies. Public courses are offered at regular intervals.

The course schedule can be found at http://www.AngelikaLanger.com/Courses/Schedule.html.

Courses can also be held at your company site. Duration and content will then be tailored to your specific needs and prerequisites.

Contact

For availability and enrollment send e-mail to contact@AngelikaLanger.com. For further information see http://www.AngelikaLanger.com/Courses.html.

Java Programming with Lambdas & Streams

- *Java 8 - Language Features & APIs*
- *Lambda Expressions*
- *Parallel Streams*

Course Description

Major extensions to the language and the JDK have been released with Java 8 - most prominently "Lambdas & Streams".

The new language features include lambda expressions and method references, both of which support programming techniques known from functional languages. Now that Java has multiple inheritance via default interface methods programming techniques similar to mixins and traits are possible. In this seminar attendants will gain an overview of all new language features and will practice their use in hand-on labs.

Mastering lambdas is the prerequisite for successfully using streams. Streams are an extension to the JDK collection, which underwent a major overhaul in Java 8. Streams provide a functional API for sequential and parallel bulk operations on sequences of elements. The seminar provides an overview of the stream API from basics such as forEach-filter-map-reduce to advanced operations such as flatMap and collectors. Attendants will practice the use of streams in numerous labs.

The key motivation for leaning about streams is the convenient way in which they support parallelized access to sequences. In order to understand the complex performance model of parallel stream operations the seminar provides insights into the inner workings of streams.

Objective

This seminar aims to give a comprehensive introduction to all new language features in Java 8 including programming techniques and idioms. In addition, the course covers in-depth the new stream API from basics to advanced usage including the complex performance model of parallel streams.

Audience

Software engineers who intend to use parallel streams for performance reason or are generally interested in keeping their Java skills up-to--date.

Prerequisite

Sound knowledge of Java.

Duration

3 days

Language

English or German

Course Overview

1. LAMBDAS
   - Lambdas Expressions
   - Method/Constructor References
   - Functional Interfaces
   - Default & Static Interface Methods
   - Programming with Lambdas
   - Runtime Representation of Lambdas
2. STREAM API
   - Streams & Collections
   - ForEach-Filter-Map-Reduce
   - Fluent Programming
   - Intermediate & Terminal Operations
   - Mappers & Collectors
3. STREAM INTERNALS
   - Pipelining
   - Stateless & Stateful Operations
   - Non-Interference Requirement
   - Sequential & Parallel Execution
   - Performance Model
4. STREAM EXTENSIONS
   - User-Defined Stream Sources

Location

Open enrollment courses are conducted in collaboration with local partner companies. Public courses are offered at regular intervals.

The course schedule can be found at http://www.AngelikaLanger.com/Courses/Schedule.html.

Courses can also be held at your company site. Duration and content will then be tailored to your specific needs and prerequisites.

Contact

For availability and enrollment send e-mail to contact@AngelikaLanger.com. For further information see http://www.AngelikaLanger.com/Courses.html.

New Features in Java 8

- *Java 8 - Language Features & APIs*
- *Lambda Expressions & Parallel Streams*
- *Concurrency Utilities & Date/Time*

Course Description

Major extensions to the language and the JDK have been released with Java 8 - most prominently "Lambdas & Streams".

The new language features include lambda expressions and method references as well as default and static interface methods. In this seminar attendants will gain an overview of all new language features and will practice their use in hand-on labs.

Mastering lambdas is the prerequisite for successfully using streams. Streams are an extension to the JDK collection framework. They provide a functional API for sequential and parallel bulk operations on sequences of elements. The seminar provides an overview of the stream API from basics such as filter-map-reduce to advanced operations such as collectors. Attendants will practice the use of streams in numerous labs.

The key motivation for leaning about streams is the convenient way in which they support parallelized access to sequences. In order to understand the complex performance model of parallel stream operations the seminar provides insights into the inner workings of streams.

The most relevant new concurrency utility is CompletableFuture. It supports asynchronous result processing in a fluent style - yet another API that relies on lambdas. The remaining new concurrency features are for expert users who strive for better performance of their multi-threaded applications.

The seminar also gives an overview of the Date/ Time API, type annotations, and the metaspace.

Objective

This seminar aims to give a comprehensive introduction to all new language features in Java 8 including programming techniques and idioms. In addition, the course covers in-depth the new stream API and provides an overview of further JDK extensions.

Audience

Software engineers who intend to use parallel streams for performance reason or are generally interested in keeping their Java skills up-to--date.

Prerequisite

Sound knowledge of Java.

Duration

3 days

Language

English or German

Course Overview

1. LAMBDAS
   - Lambdas Expressions
   - Method/Constructor References
   - Default & Static Interface Methods
   - Programming with Lambdas
2. STREAM API
   - Streams & Collections
   - ForEach-Filter-Map-Reduce
   - Intermediate & Terminal Operations
   - Mappers & Collectors
3. STREAM INTERNALS
   - Pipelining
   - Sequential & Parallel Execution
   - Performance Model
4. CONCURRENCY UTILITIES
   - Completable Future
   - Stamped Lock
   - Accumulators
   - @Contended
5. MISCELLANEOUS
   - Date/Time API
   - Type Annotations
   - Metaspace vs. PermGen

Location

Open enrollment courses are conducted in collaboration with local partner companies. Public courses are offered at regular intervals.

The course schedule can be found at http://www.AngelikaLanger.com/Courses/Schedule.html.

Courses can also be held at your company site. Duration and content will then be tailored to your specific needs and prerequisites.

Contact

For availability and enrollment send e-mail to contact@AngelikaLanger.com. For further information see http://www.AngelikaLanger.com/Courses.html.

Introduction to Java

- *Language, Tools and APIs*
- *Programming Idioms and Best Practice*
- *Traps and Pitfalls*

Course Description

This course focuses on giving you an understanding of exactly what Java is and how to build, compile, and distribute effective stand-alone Java applications. The course is designed to be a comprehensive overview of the Java language and runtime model for developers experienced with another programming language. Upon completion, you will be able to program using Java. You will know how to create Java applications.

Audience

Professional programmers who want to learn Java.

Prerequisite

To fully benefit from this course, attendants should be experienced computer programmers who understand the rudiments of both procedural and object-oriented programming.

Duration

5 days

Language

English or German

Course Overview

Basic Java Language Features
- Java SDK
- Language Basics
- Arrays

Classes
- Classes
- References
- Access Modifiers
- Static Fields and Methods

Inheritance
- Inheritance
- Dynamic Binding

Initialization
- Construction
- Final Variables

Interfaces and Enumerations
- Interfaces
- Enumeration Types

Error Handling
- Exception Handling
- Assertions

Object Infrastructure
- Class Object
- Clone
- Equals

Parameterized and Nested Types
- Parameterized Types
- Nested Types

Reflection
- Reflection

Utilities
- Collections
- I/O

Concurrent Programming
- Threads
- Synchronization
- Concurrency Utilities

Java Platform Library APIs
- Swing
- Networking
- JDBC

Tools
- JavaDoc
- Jar

Location

Open enrollment courses are conducted in collaboration with local partner companies. Public courses are offered at regular intervals.

The course schedule can be found at http://www.AngelikaLanger.com/Courses/Schedule.html.

Courses can also be held at your company site. Duration and content will then be tailored to your specific needs and prerequisites.

Contact

For availability and enrollment send e-mail to contact@AngelikaLanger.com. For further information see http://www.AngelikaLanger.com/Courses.html.

BIO

Angelika Langer

Trainer / Mentor / Author / Consultant

Biography

I work as an independent trainer and consultant with a course curriculum of challenging Java and C++ seminars. My work is backed by 25+ years of practical experience as a software engineer in Germany and the US.

Currently my preferred fields of interest are training, coaching, and mentoring in the area of Java concurrency, Java performance, new Java features, and advanced core Java in general. I invest a considerable amount of my time on development of explanatory material, including course material, multimedia training, books, and articles.

I conduct lectures, seminars, and workshops worldwide, mainly in Europe and North America. I enjoy speaking at conferences all over the world. As a Java champion I support the Java Users Groups by joining their meetings and giving presentations. Open enrollment courses are organized at regular intervals, usually in collaboration with partner companies in Germany and the USA. On-site training is delivered world-wide.

The training business leaves me little time for consulting or extended periods of contractor work. The consulting jobs that I accept are mainly mentoring assignments including code reviews, audits, performance analysis, special purpose workshops, etc.

Course Curriculum

Currently I teach Java and C++ related topics based on my own course material. All courses are challenging; the target audience is practicing programmers; I rarely teach introductory courses although I can deliver introductions to C++ and Java on request. My curriculum includes Concurrent Java, High-Performance Java, Effective Java, Lambdas & Streams as well as Reliable C++, and C++ Templates. For further information see http://www.angelikalanger.com/Courses.html.

Publications

The list of my publications is too long to be mentioned here. For a complete list see www.angelikalanger.com/Publications.html.

I have been presenting technical sessions at numerous conferences around the world. Some are mentioned at lanyrd.com/. For a comprehensive list see www.angelikalanger.com/Conferences.html.

In brevity: I am author of the “Java Generics FAQ”, and the "Lambdas Tutorial & Reference", both being online resources devoted to language features in Java. In addition, I am co-author of the authoritative book "Standard C++ IOStreams and Locales" (Addison-Wesley, 2000) and “Java Core Programmierung” (entwickler.Press, 2011). I served as a columnist for C++ Report and C/C++ Users Journal and have been writing the “Effective Java” series (currently published in JavaMagazin) since 2002.

Public Speaking

I have been speaking at more conferences world-wide than I can remember. Some are mentioned at lanyrd.com/. For a complete list see www.angelikalanger.com/Conferences.html.

Past lectures were held at JavaOne, OOPLSA, TOOLS, JAX, OOP (Germany), ROOTS (Norway), JavaZone (Norway), JFokus (Sweden), C++ World (USA), Software Development (USA), ACCU (UK), SDC (Sweden), JDD (Poland), GeeCon (Poland), Ch/Open (Switzerland), many JUG meetings, and several other events.

Contact Info

Angelika Langer Training & Consulting
Neumarkter Straße 86d
81673 München, Germany
email: contact@AngelikaLanger.com
URL: http://www.AngelikaLanger.com
twitter:@AngelikaLanger