Javaの理論と実践: JDK 5.0における、より柔軟でスケーラブルなロック

新しいロック・クラスがsynchronizedを改善、ただし、まだsynchronizedを捨てるべきではない

Brian Goetz
Principal Consultant
Quiotix

2004年 10月 26日

JDK 5.0では、ハイ・パフォーマンスの同時アプリケーションの開発用に、強力な選択肢が追加されています。例えば`java.util.concurrent.lock`の`ReentrantLock`クラスが、Java言語の`synchronized`機能の置き換えとして提供されています。これは、メモリーの意味体系もロックの意味体系も同じであり、競合状態下でもパフォーマンスが高く、`synchronized`には無い機能も持っています。では`synchronized`のことは忘れて、`ReentrantLock`だけを使うべきなのでしょうか。並行処理のエキスパートBrian Goetzが、その答えを持って夏休みから戻ってきました。

このシリーズの他の記事を見る

マルチスレッド処理や並行処理は新しいものではありませんが、Java言語設計で革新的だったのは、クロス・プラットフォームのスレッド・モデルと正式なメモリー・モデルを言語仕様の中に直接取り入れた、最初の主流言語がJavaだったという点です。Javaのコア・クラス・ライブラリーには`Thread`クラスがあり、これがスレッドを作り、開始し、操作します。またJava言語には`synchronized`と`volatile`という、スレッドにまたがる並行性制約と通信するための構成体が含まれています。これによってプラットフォームに依存しない並行クラスの開発が単純になりますが、並行クラスを書くことが些末なことに成り下がるわけではなく、単に少し易しくなるというだけです。

synchronizedを簡単に復習する

あるコード・ブロックを「同期させる」と宣言すると、一般的に原子性（atomicity）と可視性（visibility）という、2つの重要な結果が現れます。原子性というのは、あるモニター・オブジェクトで保護されたコードを実行できるのは一度に一つのスレッドのみであり、共有状態を更新する時に複数のスレッドが衝突するのを防げるようになっていることを言います。可視性というのはもっと微妙なもので、メモリー・キャッシュやコンパイラー最適化における不測の変化を処理します。変数のキャッシュ値は普通、（レジスターにある場合にしろ、プロセッサー固

有のキャッシュにある場合にしろ、また命令再配列やその他のコンパイラー最適化を受けている場合にしろ）他のスレッドからすぐには見えないようになっており、スレッドはキャッシュ値とは無関係です。ところが下記のコードで示すように同期化を使用すると、ランタイムは、一つのスレッドが`synchronized`ブロックを出る前に行った変数の更新が、別のスレッドがその同じモニターで保護された`synchronized`ブロックに入る時に、そのスレッドから確実に見えるようにするのです。`volatile`変数にも同じような規則があります（同期化とJavaのメモリー・モデルに関しては参考文献を見てください）。

```
synchronized (lockObject) { // update object state
}
```

ですから同期化は、（同期化境界(synchronization boundaries)が正しく設定されている限り）レース条件やデータ破損を引き起こすことなく、複数の共有変数を確実に更新するために必要なことをすべて行ってくれるのです。そして、こうした変数の最新値を、適切に同期化される他のスレッドが確実に見るようにしてくれるのです。ですから、明確でクロス・プラットフォームのメモリー・モデル（最初の定義にあった一部の誤りがJDK 5.0では修正されています）を定義すれば、下記の単純な規則に従うだけで「一度書けばどこででも実行できる（Write Once, Run Anywhere）」並行クラスを作ることが可能になります。

> 次に別のスレッドが読み込む可能性のある変数を書く場合、または先ほど別のスレッドが書き込んだ可能性のあるスレッドを読み込む場合には、必ず同期化を行う必要がある。

さらに良いことに、最近のJVMでは、競合無しの同期化（あるスレッドが既にロックを保持している場合には、他のスレッドはロックを取得しようとしない）のパフォーマンス・コストはごく僅かなものです。（これは常に正しいわけではありません。初期のJVMでの同期化は最適化されていなかったため・・今は迷信と思われていますが・・競合の有無によらず同期化には大きなパフォーマンス・コストがかかるという説が生まれてしまいました。）

同期化を改善する

ということで、同期化は大変結構に思えます。ではなぜJSR 166グループが`java.util.concurrent.lock`フレームワークの開発にあれほどの時間をかけたのでしょう。答えは単純です。同期化は良いものですが、完全ではないのです。同期化にはいくつか、機能的な制限があります。つまり、ロックを取得しようと待っているスレッドに割り込むことはできず、永遠に待ち続けるつもりでないと、ロックをポーリングしたりロックを取得したりすることはできません。また同期化では、ロックは、取得されたのと同じスタック・フレームで解放されることが必要です。これはほとんどの場合には正しい（そして例外処理とうまくやり取りできる）のですが、非ブロック構造のロックの方がずっと良いという場合が、僅かとはいえ存在するのです。

ReentrantLockクラス

`java.util.concurrent.lock`の`Lock`フレームワークはロックの抽象化であり、これによってロック実装が言語の機能ではなく、Javaクラスとして実装できるようになります。これによって、異なったスケジューリング・アルゴリズムやパフォーマンス特性、意味体系を持つ、複数の`Lock`が実装できるようになります。`Lock`を実装する`ReentrantLock`クラスは、並行処理やメモリーの意味体系が`synchronized`と同じですが、その他にもロック・ポーリング(lock polling)、タイム・ロック・

ウエイト(timed lock waits)、割り込み可能ロック・ウエイト(interruptible lock waits)などの機能が追加されています。その上、激しい競合状況下でのパフォーマンスが、はるかに高くなっています。（つまり、多くのスレッドが共有リソースに対してアクセスしようとしている場合に、JVMがスレッドのスケジューリングのために費やす時間が減り、より多くの時間がスレッドの実行に使われるのです。）

では、再入可能ロック（reentrant lock）とは何を意味するのでしょうか。単純に言うと、ロックに関連付けられた取得カウント（acquisition count）があり、ロックを保持するスレッドが再度ロックを取得すると取得カウントが増加され、ロックを本当に解放するには2回解放する必要があるのです。これは`synchronized`の意味体系と対応します。つまりあるスレッドが、そのスレッドが既に所有する、モニターで保護された同期ブロックに入ると、そのスレッドは進むことが許されますが、このスレッドが2番目の（あるいは、後から入った）`synchronized`ブロックを出る時にはロックは解放されず、そのモニターで保護されて入った最初の`synchronized`ブロックを出る時に解放されるのです。

リスト1のコード例を見ると、`Lock`と同期との違いが一つ、すぐに分かります。つまりロックはfinallyブロックで解放される必要があるのです。そうしないと、保護されたコードが例外を投げると、ロックは永遠に解放されないかも知れないのです！ この差は些細なことに思えるかも知れませんが、実は非常に重要なのです。finallyブロックにあるロックを解放し忘れると、プログラムに時限爆弾を仕掛けたようになり、遂に爆発したとしても追跡は非常に困難なのです。一方同期ではJVMが、ロックが確実に、自動的に解放されるようにするのです。

リスト1. ReentrantLockでコード・ブロックを保護する

```
Lock lock = new ReentrantLock();
lock.lock();
try { // update object state
}
finally {
  lock.unlock();
}
```

追加として、競合下での`ReentrantLock`の実装は現在のsynchronizedの実装よりも、はるかにスケーラブルです。（将来のバージョンのJVMでは、競合下でのsynchronizedのパフォーマンスが改善される可能性があります。）これはつまり、多くのスレッドが全て同じロックに対して競合している場合には、`synchronized`よりも`ReentrantLock`を使った方が全体的なスループットは高くなる、ということを意味しています。

ReentrantLockと同期のスケーラビリティを比較する

Tim Peierlsが、線形合同法（linear congruence）による疑似乱数生成（pseudorandom number generator: PRNG）を使って、`Lock`に対する`synchronized`の相対的なスケーラビリティを比較する、単純なベンチマークを構築しています。PRNGは`nextRandom()`が呼ばれる度に現実の作業をするので、このベンチマークは（いわゆるベンチマークの多くとは違い）、不自然なタイミングにしたり、実際には何もしないコードであったりするものではなく、`synchronized`と`Lock`を現実の世界に適用したものを測定していると言うことができます。

このベンチマークには`PseudoRandom`に対するインターフェースがあり、これは`nextRandom(int bound)`という一つのメソッドを持っています。このインターフェースは`java.util.Random`クラス

の機能と非常に似ています。PRNGは次の乱数を生成する時の入力として、最後に生成された番号を使い、最後に生成された数字はインスタンス変数として維持されるので、この状態を更新するコード部分は他のスレッドに先取りされないようにすることが重要です。これを確実にするために、ここではある形式のロックを使うことにします。（これは`java.util.Random`クラスでも同じです。）ここでは2つの`PseudoRandom`実装を作りました。1つは同期を使うもので、もう1つは`java.util.concurrent.ReentrantLock`を使います。ドライバー・プログラムは幾つかのスレッドを作り出し、それぞれが狂ったようにサイコロを振ります。その後で2つの実装が、サイコロを毎秒何回振ることができたかを計算するのです。その結果は図1と図2で、スレッド数に対する値として要約されています。このベンチマークは完璧なものではなく、また2台のシステムで実行されたに過ぎません（ハイパースレッド機能(hyperthreading)がありLinuxを実行するデュアルXeonと、単一プロセッサーのWindowsマシン）。それでも、`ReentrantLock`の方が`synchronized`よりもスケーラビリティが高いことを示すには充分と言えるでしょう。

図1. 単一CPUでの、同期とロックのスループット比較

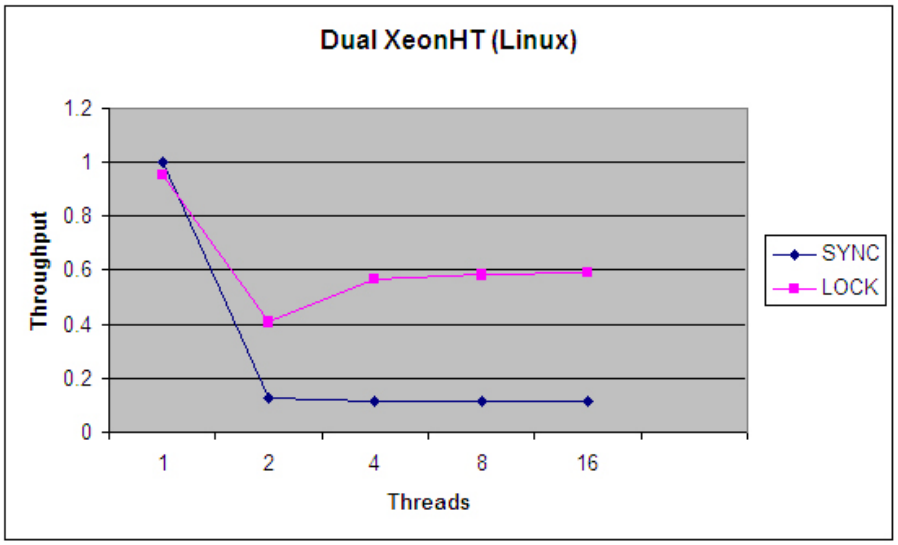


図2. 4 CPUでの、同期とロックの（正規化した）スループット比較

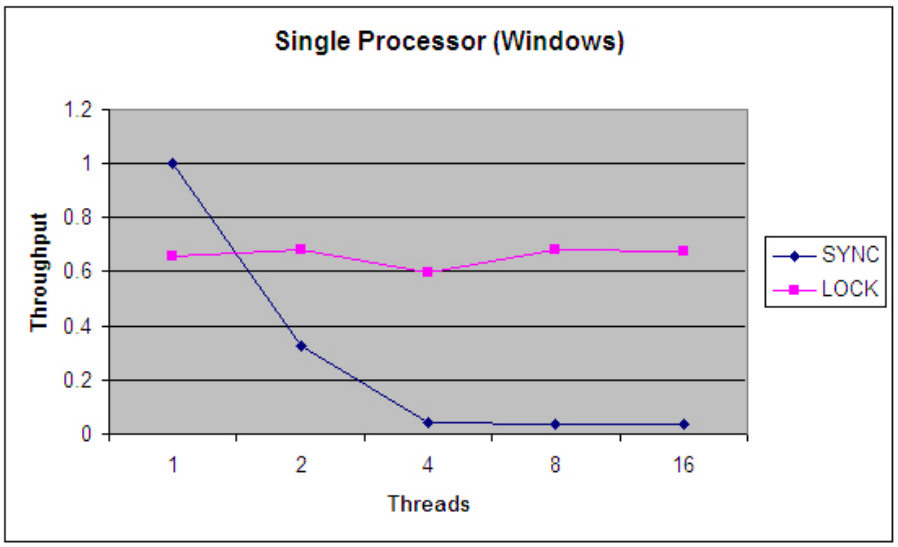


図1と図2は、様々な実装を1スレッドのsynchronizedの場合に正規化して、毎秒のコール数でスループットを示したものです。それぞれの実装は、比較的早く定常状態のスループットに収束しています。これは一般的に、プロセッサーが完全利用されていること、CPU時間の一部は実際の作業（乱数計算）に使用され、また別の一部はオーバーヘッドのスケジューリングに使われていることを示しています。同期化を使っている方は競合が少しでもあると大幅にスループットが悪くなりますが、Lockを使っている方はオーバーヘッドのスケジューリングに使う時間はずっと少なく、さらなるスループット向上やCPU利用率向上の余地があることを示しています。

条件変数

ルートのObjectクラスには、スレッドにまたがって通信するための特別な幾つかのメソッド、つまりwait()やnotify()、notifyAll()があります。これらは高度な並行処理機能であり、多くの開発者はほとんど使いません。これらは非常に繊細なもので、間違って使いがちなことを考えると、あまり使われないのは良いことかも知れません。幸いJDK 5.0にjava.util.concurrentが追加されたことによって、このようなメソッドを使わざるを得ない場面はさらに少なくなります。

通知とロックの間には、やり取りがあります。つまりオブジェクトに対してwaitしたりnotifyしたりするには、そのオブジェクトに対するロックを保持する必要があります。Lockが同期の一般化であるのと同様、Lockフレームワークには、Conditionと呼ばれる、waitとnotifyの一般化が含まれています。Lockオブジェクトは、そのロックに結合された条件変数に対するファクトリー・オブジェクトとして動作し、標準的なwaitやnotifyソッドとは違って、与えられたLockに関連する条件変数は一つ以上ある可能性があります。これによって、多くの並行アルゴリズムの開発が簡単になります。例えばConditionのJavadocでは、「not full」と「not empty」という2つの条件変数を使った境界付きバッファー実装の例を示していますが、これはロック毎に単一のwaitを使った等価な実装よりも読みやすく（そして効率的に）なっています。wait、notify、notifyAllと似たConditionメソッドにはawaitやsignalそれにsignalAllといった名前がつけられていますが、これはObjectにある、対応したメソッドをオーバーライドできないためです。

それはフェアではない

Javadocを熟読すると、ReentrantLockのコンストラクターに対する引き数の一つが、fair（フェア）ロックにするかunfair（アンフェア）ロックにするかというブール値であることに気がつくでしょう。フェアなロックというのは、スレッドが、そのスレッドが要求したのと同じ順序でロックを取得するものです。一方、アンフェアなロックというのは、ロックを最初に要求したスレッドがそのロックを取得する前に、別のスレッドがそのロックを取得してしまうというバージング(barging)を許すものです。

なぜ全てのロックをフェアにしないのでしょうか。フェアなのは良いことであり、アンフェアは悪いことのはずです。（偶然かも知れませんが、子供が大人に決定を求める時には、ほとんど必ず「それじゃ不公平だ」という状況が生まれます。私達は公平性が重要であることを知っており、子供達もそれを知っているのです。）現実には、ロックに対して公平性を保証するのは非常に強力な保証を与えることであり、大きなパフォーマンス・コストがかかるのです。公平性の保証のために帳尻合わせと同期が必要だということは、競合状態にあるフェアなロックはアンフェアなロックよりもスループットがずっと低いことを意味します。ですから、キュー・アップした通りにスレッドをサービスすることがアルゴリズム的に致命的に重要である場合を除いて、fairのデフォルトはfalseに設定すべきです。

では同期はどうでしょう。組み込みのモニター・ロックはフェアなのでしょうか。その答えを聞くと多くの人は驚くでしょうが、フェアではなく、フェアであったこともないのです。JVMが全てのスレッドに対して、いずれは待っていたロックが与えられることを保証するので、誰もスレッドが足りないと不満は言わないのです。大部分の場合には統計的に公平性を保証するだけで充分であり、その方が、決定的な公平性の保証よりもずっとコストが安くすみます。ですからReentrantLocksがデフォルトで「アンフェア」であるという事実は、同期化ではずっと真実であったことを、単に明示的に示しているに過ぎません。同期に関して心配したことがないのであれば、ReentrantLockにも心配する必要はないのです。

図3と図4は図1と図2と同じデータですが、デフォルトのバージング・ロックの代わりにフェア・ロックを使った、新しい乱数ベンチマークのデータが追加されています。ご覧の通り、公平性はタダではありません。本当に必要であれば公平性のコストをかけるべきですが、デフォルトにすべきではありません。

図3. 4 CPUでの、同期とバージング・ロック、フェア・ロックそれぞれのスループット比較

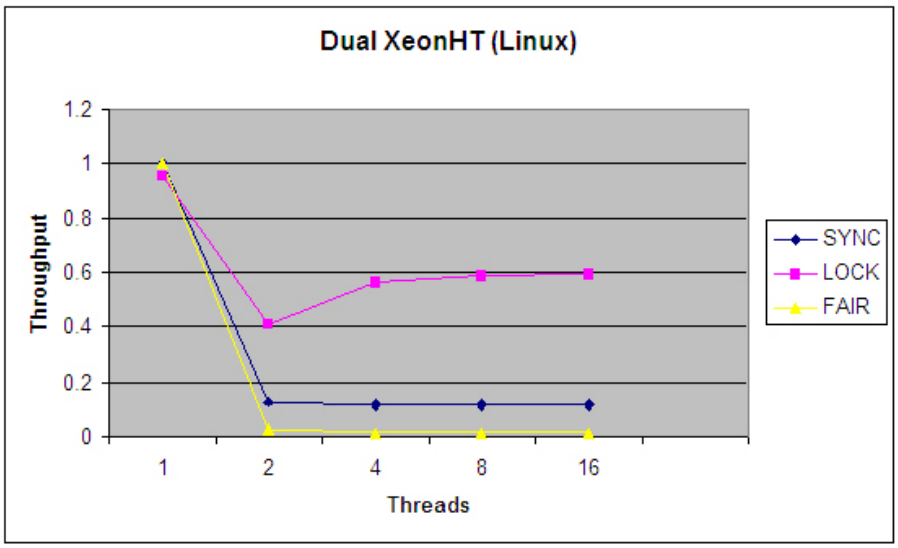
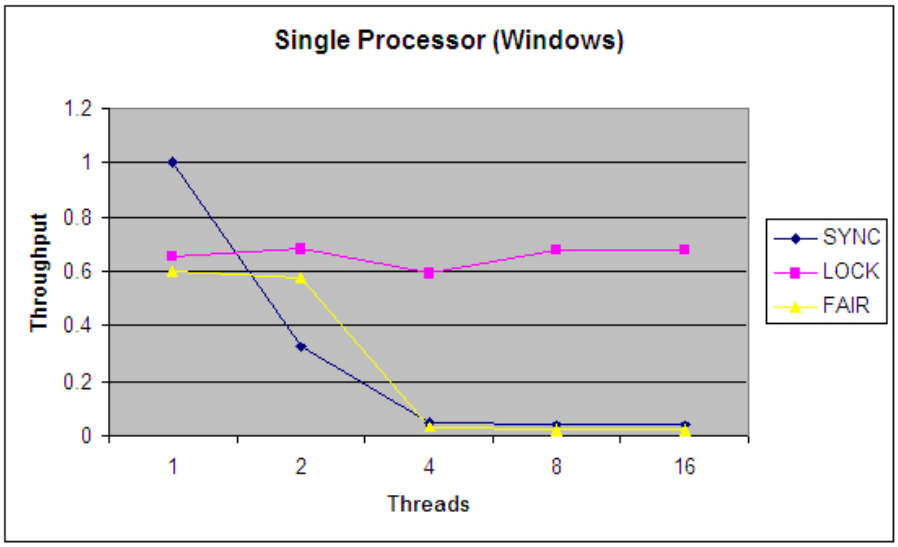


図4. 単一CPUでの、同期とバージング・ロック、フェア・ロックそれぞれのスループット比較



全ての面で良いのか？

ReentrantLockは全ての面でsynchronizedよりも良いと思われるかも知れません。ReentrantLockはsynchronizedにできることは何でもでき、メモリー意味体系と並行処理の意味体系は同じであり、synchronizedにない機能を持ち、負荷状態でのパフォーマンスも高いのです。ではsynchronizedを、これまで改善されてきて結局廃案にされたスクラップの山の一部として格下げし、全く忘れてしまうべきなのでしょうか。あるいはもっと極端に、既存のsynchronizedコードをReentrantLockで書き直すべきなのでしょうか。実際、Javaプログラミングの入門書の幾つかはマルチスレッドに関する章の説明をそのように進めており、関連事項として同期化に触れるのみで、全ての例をLockで説明しています。これは行き過ぎであると私は思っています。

まだ同期化を捨て去るべきではない

ReentrantLockは非常に素晴らしい実装であり、同期よりも大きな利点が幾つかあることは確かですが、だからといって同期化を、使用すべきでない機能として捉えるのは大きな間違いだと思います。java.util.concurrent.lockのロック・クラスは、上級ユーザーが高度な条件で使用すべき、高度なツールなのです。一般的に、Lockの持つ高度な機能に特別な必要性がある場合、または具体的な状況下で同期がスケーラビリティのボトルネックになることが（単に推測ではなく）実証できる場合を除いて、同期化に踏みとどまるべきだと言えるでしょう。

明らかに「良い」実装を採用することに対して、なぜ私はそう保守的なのでしょうか。同期には相変わらず、java.util.concurrent.lockのロック・クラスに比べて幾つかの利点があるのです。例えば、同期を使っている時にロックを解放し忘れるのは不可能です。synchronizedブロックから出る時に、JVMが解放してくれるからです。ところがロックを解放するためにfinallyブロックを使うことは忘れがちであり、これはプログラムにとって大きな問題になります。プログラムはテストをパスしますが、フィールドでロックしてしまいます。そうすると、なぜそうなったかを割り出すのが非常に困難になります（これ自体が、初心者には絶対にLockを使わせないようにする良い理由となります）。

もう一つの理由として、JVMが同期を使ってロック取得と解放を管理する時には、JVMがスレッド・ダンプを生成する時にロック情報を含めることができるのです。これらはデッドロックやその他予期せぬ振る舞いの原因を判別できるので、デバッグには非常に貴重なものです。ところが`Lock`クラスは単に普通のクラスであり、どのスレッドがどの`Lock`オブジェクトを所有しているか、JVMには（まだ）分かりません。さらに同期化はほとんど全てのJava開発者が知っており、またどのバージョンのJVMでも動作します。JDK 5.0が標準となるまでには恐らく最低2年間はかかるでしょうが、その間`Lock`クラスを使うということは、一部のJVMでは使えない機能、一部の開発者は理解していない機能を使うことになります。

どういう場合にsynchronizedではなくReentrantLockを選択すべきなのか

ではどういう場合に`ReentrantLock`を使うべきなのでしょう。答えは極めて単純です。synchronizedではできない何か、たとえばタイム・ロック・ウエイト(timed lock waits)、割り込み可能ロック・ウエイト(interruptible lock waits)、非ブロック構造化ロック(non-block-structured locks)、複数の条件変数、あるいはロック・ポーリング(lock polling)などが、実際に必要な場合に使えばよいのです。`ReentrantLock`にはスケーラビリティの利点もあるので、高度な競合が見られる状況が実際にあるならば、`ReentrantLock`を使うべきです。ただし、大部分の同期ブロックにはほとんど競合が見られず、ましてや高度な競合が見られることは稀なことは忘れないでください。`ReentrantLock`を使えば「パフォーマンスが高くなるはず」と単純に思いこまず、同期が不適当だと証明できるまでは同期を使って開発することを、私としてはお勧めしたいと思います。繰り返しますが、これは上級ユーザー向けの上級ツールなのです。（そして真に上級のユーザーであれば、単純なツールでは不適当だと確認できるまでは、一番単純なツールを好むものです。）いつものことですが、まず正しく動くようにし、その後で、さらに早くすべきかどうかを考えるべきです。

まとめ

`Lock`フレームワークは同期と互換性を持つ置き換えであり、`synchronized`には無い多くの機能を持ち、競合状態下でも高いパフォーマンスが得られる実装が可能です。ところがこうした明確な利点だけでは、`synchronized`の代わりに常に`ReentrantLock`を使うべきだということにはなりません。むしろ、`ReentrantLock`の力が本当に必要かどうかに基づいて判断を下すべきです。圧倒的大部分の場合において、`ReentrantLock`は必要ないはずです。同期でも充分に用が足りるのであり、同期ならばどんなJVMでも動作し、大多数の開発者が理解している上、間違いも起こしにくいのです。本当に必要になるまでLockはしまっておくべきです。そして本当に必要になった時には、Lockがあって良かったと思うでしょう。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。