

Parasoft Jtest 8.4 技術資料

マルチスレッドプログラミング



テクマトリックス株式会社

今回は、2009年夏に日本語版のリリースを予定している Jtest 8.4 で強化されたマルチスレッドプログラム開発をサポートする静的解析ルールについて紹介します。マルチスレッドプログラムは、マルチコア CPU が普及している現在では CPU リソースを有効に活用でき、パフォーマンス向上を期待できるメリットがありますが、開発に注意を要することでも知られています。

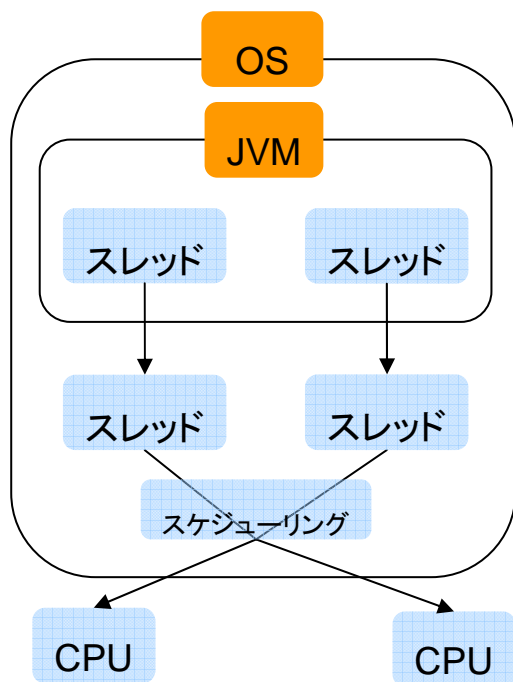
また、安全なマルチスレッドプログラムを作成することは、セキュリティの観点でも重要なポイントです。不適切なスレッド同期は同時実行性の問題を引き起こし、アプリケーションの予期しない振る舞いを招く可能性があり、それがセキュリティに影響を与える可能性もあります。

背景

シングルコア CPU の性能向上（クロック数の増加）は限界を迎え、CPU ベンダはコアを増やす戦略を選択しました。私たちの周りでも、マルチコア CPU を搭載したマシンは珍しくない状況です。

とはいえ、CPU クロック数があまり変わらないことから考えると、CPU コアが増えたからといって既存のプログラムの処理速度が向上するわけではありません。なぜならば、そのプログラムがマルチスレッドで並列に動作しなければ、マルチコア CPU のメリットを得ることができないからです。

つぎはマルチスレッドプログラムの動作イメージです。



たとえば、既存のプログラム（シングルスレッド）をマルチコア CPU 搭載の最新マシンで実行したとしても、CPU 資源を有効に活用できず、処理速度の向上はあまり見込めないかもしれません。

将来的に、あらたに作成するプログラム、または既存のプログラムはマルチスレッド化する必要に迫られる可能性があり、今後、開発者が直面する問題のひとつとして考えられます。

注意すべき点

マルチスレッドプログラミングは、つぎのような問題を伴う可能性があります。これらの問題は適切に設計、コーディングされていれば防げる問題です。

デッドロック

排他制御が正しくないために、スレッド間で依存するロックを保持したまま、互いがもつリソースの解除を待ち続けることで、処理を続けられなくなる事。

レース・コンディション

データ・レースともいう。排他制御が正しくないために、複数のスレッドが同一の共有リソースにアクセスした場合に、意図しないデータ変更、振る舞いが発生すること。セキュリティ脆弱性の観点からは、情報のインテグリティが損なわれる問題（「IPA セキュアプログラミング講座」による）として捉えることができる。

notify 通知の消失

例) notifyAll()でなく notify()による通知を行ったために、本来意図したスレッドに通知が届かず、結果的にその通知が失われてしまうこと。

無限ループ

例) ループ内でキャッチされた例外が適切にハンドルされないことで、そのループからブレイクできなくなり、無限ループに陥ること。

今後の課題は、マルチスレッドプログラミングに取り組む開発者がいかに既存のプログラムを安全なプログラムに作り変えるかということです。

Jtest 8.4 は、マルチスレッドプログラミングを支援する多くのルールを搭載することで、この問題へのソリューションを提供します。

ここでは、いくつかの例を用いて開発時に直面する可能性のある問題と Jtest を使用してどのようにその問題を検出するか説明していきます。

デッドロックの可能性

つぎのプログラムを見てみましょう。

```
package examples.rules.trs;

public class USL {
    int count;
    private Object LOCK = new Object();
    synchronized void updateOnce() {
        count++;
    }
    void updateTwice() {
        synchronized(LOCK) {
            count = count + 2;
        }
    }
}
```

例 1 デッドロックの可能性

クラス USL では インスタンスメソッドである updateOnce が synchronized として宣言されています。これは変数 count をプロテクトするためです。インスタンスメソッド updateTwice 内では synchronized ブロックで、オブジェクト LOCK を使用して count 変数をプロテクトしています。

この場合は、前者は This インスタンス上にロックを取り、後者は LOCK インスタンス上にロックを取ることを表しています。

Jtest 8.4 ではこのプログラムに対して、つぎのような違反を検出します。

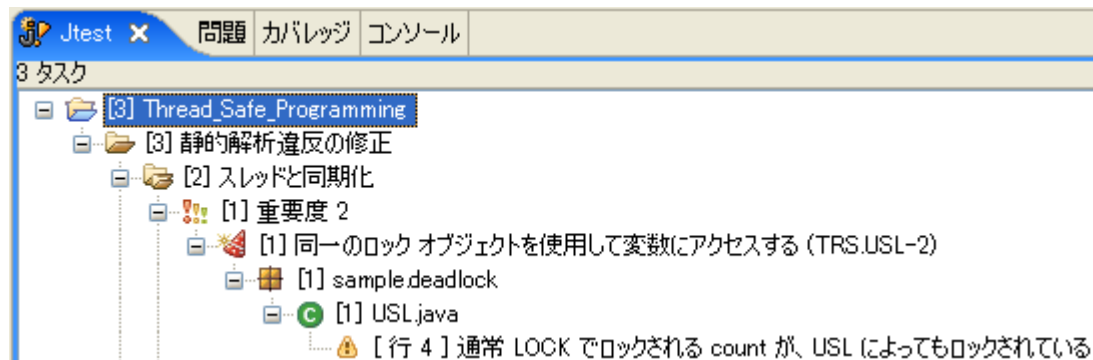


図 1 Jtest によるデッドロックの可能性の指摘

同じ変数を異なるロックでプロテクトしようとする、複数のスレッドがアクセスした場合にタイミングによってデッドロックになる可能性があります。

この場合、つぎのようにいずれも synchronized なインスタンスメソッドとして定義するほうが自然です。そうすれば、USL インスタンス上に同一のロックオブジェクトを使用して count 変数をプロテクトすることができます。

○修正例

```
package examples.rules.trs;

public class USLFixed {
    int count;
    synchronized void updateOnce() {
        count++;
    }
    synchronized void updateTwice() { //FIXED
        count = count + 2;
    }
}
```

例 2 修正した例 1 のプログラム

レース・コンディション

レース・コンディションの例として、『今月のピックアップ』2008年12月バックナンバーで紹介していますので、こちらをご参照ください。この記事では、Webアプリケーションにおける final でない static 変数を、複数スレッドで同時アクセスした場合に生じる問題を扱っています。

Atomic 変数について

Javaでは基本型・参照型の代入・参照はアトミックな操作ですが、longとdoubleはアトミックに扱われません。このため、long, doubleをスレッド間でデータ共有する場合、アトミックな操作にするために、同期ブロック内で使用するか、volatileとして宣言する必要があります。

ただし、アトミックな操作はロック取得のオーバーヘッドが伴いますので、必要以上に行うとパフォーマンスの低下を招く可能性があります。Javaのマルチスレッドサポートの理解不足やマルチスレッドであることを意識するあまりに、このようなコーディングをする可能性があります。

たとえば、つぎのコード例ではアトミックな操作を可能にする AtomicLong クラスを使用していますが、この変数は同期ブロック内にあるため、二重に同期化されています。

```

package sample.atomic;

import java.util.concurrent.atomic.*;

public class AIL {
    private AtomicLong _long = new AtomicLong();
    private static final Object LOCK = new Object();

    public void setLong(long l) {
        synchronized(LOCK) {
            _long.set(l);
        }
    }
    public long getLong() {
        synchronized(LOCK) {
            return _long.get();
        }
    }
}

```

例 3 同期が二重に行われる例

もし、この変数が同期ブロック内だけでアクセスされることが明白であれば、非アトミックな変数を使用するか、このまま AtomicLong クラスを使用する場合は、同期ブロックを削除することができます。

この場合、Jtest ではつぎのようにレポートします。

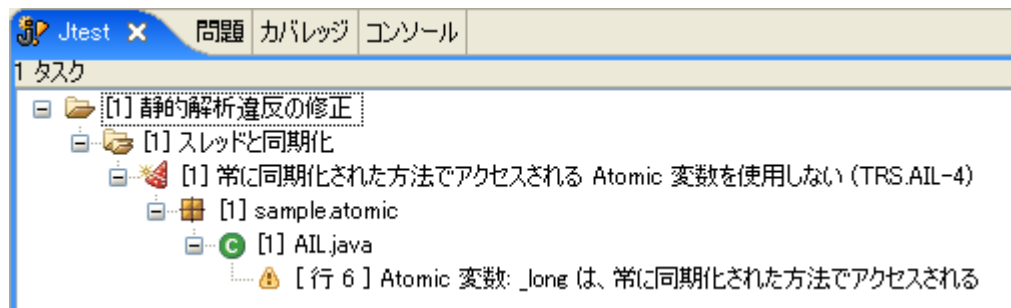


図 2 Jtest による二重の同期化の検出

このように Jtest 8.4 では、マルチスレッドプログラミング開発をサポートする静的解析ルールが多数追加されました。

スレッドにまつわるトラブルには困難な調査がつきものです。コーディング段階からこれらのルールを適用することで、より早く確実なソフトウェア開発が促進されます。

最後に Jtest 8.4 であらたに追加された「Thread Safe Programming」ルールセットに含まれるルールの一覧を掲載します。

「スレッドと同期化」カテゴリルールのほかにも、マルチスレッドプログラミングの作法と関連の深いルールが多数含まれていますので、さまざまなトラブルを予防するのに役立ちます。

Thread Safe Programming ルールセット

カテゴリ	ルール
ガーベッジ コレクション [GC]	際限なく大きくなる可能性のある static なコレクションまたはマップを避ける [GC.STV-3]
初期化 [INIT]	遅延初期化のための "if" チェックでは適切な演算子を使用する [INIT.AULI-3]
最適化 [OPT]	1 つのメソッド内でだけアクセスされる private フィールドを使用しない。ローカル変数に変更する [OPT.CTLV-3]
	配列のコピーには、ループではなく System.arraycopy () を使用する [OPT.IRB-2]
	ローカル変数には同期データ構造を使用しない [OPT.SDLS-3]
	ループ本体で synchronized メソッドを呼び出さない [OPT.SYN-3]
API の使用と実装 [PB.API]	"Collections" または "Maps" で "URL" オブジェクトを使用しない [PB.API.IUMS-1]
紛らわしいまたは意図的でない振り舞い [PB.CUB]	for ループの内部でループ制御変数に代入しない [PB.CUB.FLVA-2]
論理エラー [PB.LOGIC]	無限ループを使用しない [PB.LOGIC.AIL-1]
	配列または List 要素へアクセスするのに誤った添え字変数を使用しない [PB.LOGIC.AMOI-4]
	ループの条件で使用されないループ変数を避ける [PB.LOGIC.AULV-1]
	読み込みまたはスキップを行うメソッドの戻り値を確認する [PB.LOGIC.CRRV-1]
	戻り値が null ではないことをチェックした後に readLine() の結果を破棄してはいけない [PB.LOGIC.DJNCR-1]
	無限の再帰呼び出しを行わない [PB.LOGIC.FLRC-1]
	ループ変数の使用でのバグを防止する [PB.LOGIC.INDEX-2]
	ネストされた複数の for ループにおいて同じループ変数をインクリメント/デクリメントしてはいけない [PB.LOGIC.JI-1]
	ループの条件で使用されるオブジェクトは適切にループ本体でアクセスされるようにする [PB.LOGIC.OAMC-2]
	実行時例外 [PB.RE]
タイプミス [PB.TYPO]	条件内で代入演算子を使用しない [PB.TYPO.ASI-1]
	switch 文の case のフォールスルーで同じ変数に代入してはならない [PB.TYPO.DAV-3]
不要なコード [PB.USC]	到達しない "else if" と "else" を避ける [PB.USC.UIF-1]
シリアライゼーション [SERIAL]	readObject() メソッドを synchronized として宣言してはいけない [SERIAL.SROS-3]
サーブレット [SERVLET]	Servlet クラスでインスタンス フィールドを定義してはいけない [SERVLET.IF-3]
	サーブレット内の同期化を最小限にする [SERVLET.SYN-2]
スレッドと同期化 [TRS]	常に同期化された方法でアクセスされる Atomic 変数を使用しない [TRS.AIL-4]
	すべての待ちスレッドに通知されるように notify () ではなく notifyAll () を使用する [TRS.ANF-3]
	Thread.interrupted() を誤って使用しない [TRS.ATI-3]

安全ではない java.lang.ThreadGroup 型の変数を使用しない [TRS.AUTG-3]
VM によって異なる振る舞いをするかもしれないため Thread.yield () を使用しない [TRS.AUTY-3]
静的な java.text.DateFormat および java.util.Calendar を使用してメソッドを呼び出してはならない [TRS.CDF-3]
可能な限り Hashtable および synchronizedMap でラップされた HashMap の代わりに ConcurrentHashMap を使用する [TRS.CHM-5]
Thread を拡張するクラス以外で InterruptedException をキャッチしない [TRS.CIET-4]
同期化されたコレクションに対する複合アクションによるアクセスは原子性に違反するため避ける [TRS.CMA-3]
デッドロック発生の原因となる synchronized メソッドからの synchronized メソッドの呼び出しを避ける [TRS.CSFS-3]
コンストラクタからスレッドの start メソッドを呼び出さない [TRS.CSTART-3]
コンストラクタで this 参照を流出させない [TRS.CTRE-3]
安全ではない “ダブルチェック ロッキング” パターンの実装を避ける [TRS.DCL-3]
PriorityBlockingQueue で DiscardOldestPolicy を使用しない [TRS.DOPQ-3]
デバッグ目的の場合を除いて、getState を使用しない [TRS.GSD-4]
同期化が必要な可能性のある static フィールドへのアクセスを検査する [TRS.IASF-2]
スレッドセーフな怠惰な初期化にする [TRS.ILI-4]
設計上の欠陥を示す可能性があるため IllegalMonitorStateException をキャッチしてはいけない [TRS.IMSE-3]
java.lang.Thread を拡張するクラスまたは java.lang.Runnable を実装するクラスで run() を直接呼び出さない [TRS.IRUN-3]
関連のある複数の Atomic 変数には synchronized ブロック内でアクセスする [TRS.MRAV-4]
別のスレッドして実行できるようにスレッドのサブクラスには run () を入れる [TRS.MRUN-2]
スレッドに必ず名前を付ける [TRS.NAME-3]
synchronized でないメソッドで wait (), notify (), notifyAll() を呼び出さない [TRS.NSYN-2]
finally ブロックでロックを解放する [TRS.RLF-2]
Runnable.run() を実装するメソッドで同期化を使用する [TRS.RUN-5]
定数 String を同期化しない [TRS.SCS-1]
static の synchronized メソッドと static でない synchronized メソッドを混合させない [TRS.SNSM-4]
java.util.concurrent.locks.Lock の実装に “synchronized” キーワードを使用して同期化を実行してはいけない [TRS.SOL-3]
デッドロックを引き起こす可能性があるため、public フィールドを同期化しない [TRS.SOPF-2]
相互排除の保証が難しくなるため final ではないフィールドを同期化してはいけない [TRS.SOUF-3]
set メソッドを同期化している場合、get メソッドも同期化する [TRS.SSUG-3]
Object の this 参照で同期化を実行したりセマフォ メソッドを呼び出したりしてはいけない [TRS.STR-3]
安全ではない非推奨のメソッド Thread および Runtime を呼び出さない [TRS.THRD-1]

	ロックを保持している間に Thread.sleep() を呼び出さない。さもないとパフォーマンスを劣化させたりデッドロックを引き起こしたりする [TRS.TSHL-2]
	Collections.synchronized でラップされた Collection の同期化されていないアクセスを避ける [TRS.UACS-3]
	安全な場合にだけ同期化されていない Collection または Map を使用する [TRS.UCM-3]
	同一のロック オブジェクトを使用して変数にアクセスする [TRS.USL-2]
	run() メソッドを指定せずにスレッドを開始してはならない [TRS.UT-2]
	liveness 状態をテストするループの内部でだけ wait () と await() を呼び出す [TRS.UWIL-1]
	ポーリング ループではなく wait () および notifyAll () を使用する [TRS.UWNA-2]
	"java.util.concurrent.locks.Condition" オブジェクトで正しいメソッド呼び出しを使用する [TRS.WOC-3]
	他のメソッドが同期化されていない場合、writeObject() メソッドを同期化してはいけない [TRS.WOS-4]
未使用コード [UC]	空の synchronized 文を避ける [UC.SNE-1]
その他 [MISC]	あまり多くの 非 final な static フィールドを使用しない [MISC.MSF-4]

参考文献

『増補改訂版 Java 言語で学ぶデザインパターン入門マルチスレッド編』
 結城浩 著 ソフトバンククリエイティブ刊 (ISBN4-7973-3162-3)

【お問い合わせ先】

TechMatrix

テクマトリックス(株)

システムエンジニアリング事業部

ソフトウェアエンジニアリング営業部

ソフトウェアエンジニアリング営業課

TEL 03-5792-8606 FAX 03-5792-8706

E-MAIL parasoft-info@techmatrix.co.jp

URL <http://www.techmatrix.co.jp/products/quality/jtest/>