マルチスレッド ハンズオン

第３回　デッドロックと実行性能

アジェンダ

- 前回のおさらい
- デッドロック
- 実行性能

前回のおさらい

デッドロック

単純なデッドロックの例（1）

- SimpleDeadlock.java

```
final Object left = new Object();
final Object right = new Object();

// left -> rightの順にロックを取得するタスク
Runnable leftRightTask = new Runnable() {
    public void run() {
        synchronized(left) {
            try {
                Thread.sleep(SLEEP_TIME);
            } catch(InterruptedException e) {
                // 何もしない
            }

            synchronized(right) {
                System.out.println("leftRight");
            }
        }
    }
};
```

単純なデッドロックの例（2）

- SimpleDeadlock.java

```
// right -> leftの順にロックを取得するタスク
Runnable rightLeftTask = new Runnable() {
    public void run() {
        synchronized(right) {
            try {
                Thread.sleep(SLEEP_TIME);
            } catch(InterruptedException e) {
                // 何もしない
            }

            synchronized(left) {
                System.out.println("rightLeft");
            }
        }
    }
};
```

単純なデッドロックの例（3）

Thread1
(leftRightTask)

synchronized
(left)

synchronized
(right)

Thread2が保持しているrightのロック解放待ち

Thread2
(rightLeftTask)

synchronized
(right)

synchronized
(left)

Thread1が保持しているleftのロック解放待ち

Thread1、Thread2お互いが、お互いの保持するロック解放待ちとなり、デッドロックとなる。
これを防ぐには、ロックの取得順をあらかじめ決めておく必要がある。（left→rightか、right→leftのどちらか）

動的なデッドロックの例（1）

- DyamicDeadlock.java

```
/**
 * 取引を表すクラス
 */
private static class Deal {
    /**
     * fromAccountからtoAccountへ指定したamount送金する
     * @param fromAccount
     * @param toAccount
     * @param amount
     */
        public void transferMoney(Account fromAccount, Account toAccount, BigDecimal
amount) {
            synchronized(fromAccount) {
                synchronized(toAccount) {
                    fromAccount.debit(amount);
                    toAccount.credit(amount);
                }
            }
        }
    }
```

fromAccount→toAccountの順にロックを取得している。
順番にロックを取得しているにも関わらず、デッドロックになる場合がある。
なぜか。

動的なデッドロックの例（2）

- DyamicDeadlock.java

```
final Account accountA = new Account();
final Account accountB = new Account();

// 口座Aから口座Bに10,000円送金するタスク
Runnable transferAtoBTask = new Runnable() {
    public void run() {
        for(int i=0; i<LOOP_COUNT; i++) {
            new Deal().transferMoney(accountA, accountB, new BigDecimal(10000));
        }
    }
};

// 口座Bから口座Aに20,000円送金するタスク
Runnable transferBtoATask = new Runnable() {
    public void run() {
        for(int i=0; i<LOOP_COUNT; i++) {
            new Deal().transferMoney(accountB, accountA, new BigDecimal(20000));
        }
    }
};
```

transferMoneyの引数に渡す引数の順番を入れ替えるとデッドロックが起きうる。

動的なデッドロックの例（3）

Thread1
(AtoBTask)

synchronized
(accountA)

synchronized
(accountB)

Thread2が保持しているaccountAのロック解放待ち

Thread2
(BtoATask)

synchronized
(accountB)

synchronized
(accountA)

Thread1が保持しているaccountBのロック解放待ち

Thread1、Thread2お互いが、お互いの保持するロック解放待ちとなり、デッドロックとなる。
これを防ぐには、次ページ参照。

動的なデッドロックの例（４）

- DyamicDeadlock2.java

```
int fromHash = System.identityHashCode(fromAccount);
int toHash = System.identityHashCode(toAccount);
if(fromHash < toHash) {
    synchronized(fromAccount) {
        synchronized(toAccount) {
            transferMoney(fromAccount, toAccount, amount);
        }
    }
} else if(toHash < fromHash) {
    synchronized(toAccount) {
        synchronized(fromAccount) {
            transferMoney(fromAccount, toAccount, amount);
        }
    }
} else {
    synchronized(tieLock) {
        synchronized(fromAccount) {
            synchronized(toAccount) {
                transferMoney(fromAccount, toAccount, amount);
            }
        }
    }
}
```

ロック対象オブジェクトのハッシュコードを求めて、ハッシュコードの小さい順にロックする。二つのオブジェクトで同じハッシュコードが得られた場合は、tieLockを使って同期化する。

オープンコールを使ってデッドロックを避ける（1）

- CoObjectDeadlock.java
  - Taxi#setLocationと、Dispatcher#getImageを複数スレッドから同時に呼ぶと、デッドロックを起こすことがある。
  - Taxi#setLocationは、Taxi→Dispatcherの順にロックを取得する。
  - Dispatcher#getImageは、Dispatcher→Taxiの順にロックを取得する。

オープンコールを使ってデッドロックを避ける（2）

- CoObjectDeadlock2.java
  - オープンコールとは、他のロックを保有しないでメソッドを呼び出すこと。
  - Taxi#setLocationは、以下のように改良されている。
    1) Taxiのロックを取得
    2) 必要な情報をコピー
    3) Taxiのロックを解放
    4) Dispatcherのメソッド呼び出し（ロックを取得；Taxiのロックを持たずにメソッド呼び出ししているのでオープンコールとなる）
  - Dispatcher#getImageも同様。

tryLockを使ってデッドロックを避ける（1）

- TryLock.java

```
Runnable leftRightTask = new Runnable() {
...
if(left.tryLock()) {
    try {
        Thread.sleep(SLEEP_TIME);

        if(right.tryLock()) {
            try {
                System.out.println("leftRight");
                break;
            } finally {
                right.unlock();
            }
        } else {
            System.out.println(Thread.currentThread().getName() + ": Lock right failed.");
        }
    } catch(InterruptedException e) {
        // 何もしない
    } finally {
        left.unlock();
    }
} else {
    System.out.println(Thread.currentThread().getName() + ": Lock left failed.");
}
```

left→rightの順にロックを取得するところは、SimpleDeadlock.javaと同じだが、ロックの取得に失敗した場合は、ロックを解放して処理終了する。（実用的にはリトライする）

デッドロックの診断（1）

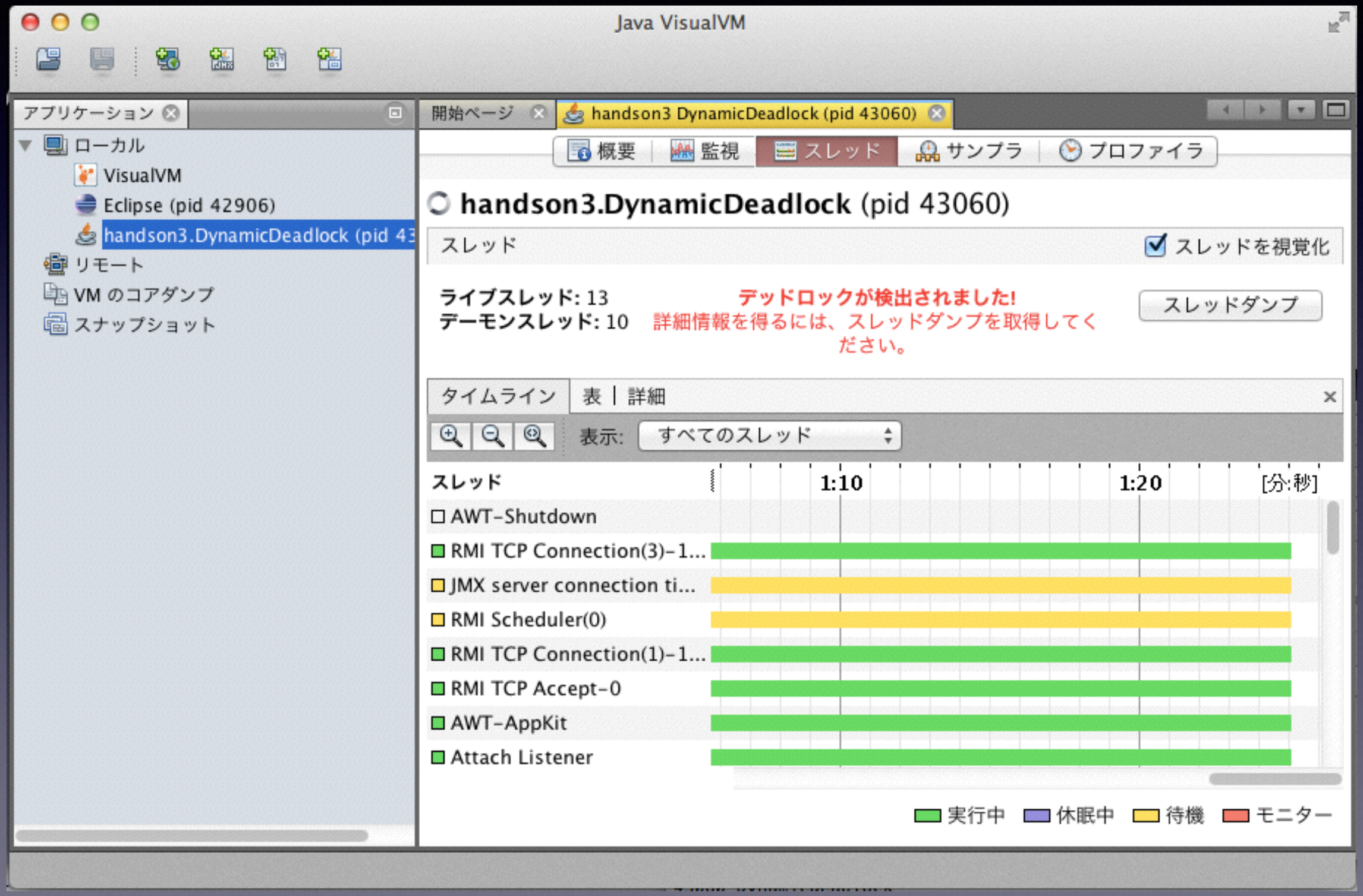
- デッドロックが起きてもJVMが例外を投げたりするわけではない（ダンマリ）。
  診断は以下のように行う。jps/jstackを使う場合。

  - DynamicDeadlockを実行する。
  - コマンドラインからjpsを実行。
  - コマンドラインからjstackを実行。引数に、jpsコマンドで取得したDynamicDeadlockのpidを指定する。
  - スレッドダンプが表示され、最下行に
    ”Found 1 deadlock.”が表示される。

デッドロックの診断（2）

- jvisualvmを使う場合
  - DynamicDeadlockを実行する。
  - コマンドラインからjvisualvmを実行。
  - アプリケーションペインから、DyamicDeadlockのプロセスをダブルクリック。
  - スレッドをクリック。
    “デッドロックが検出されました!”のメッセージが表示される。
  - （補足）「jvisualvmは、jconsole、jstat、jinfo、jstack、jmapなんかを全部まとめたすごいやつ。」だそうです。
    http://ameblo.jp/wataru420/entry-11250099243.html

デッドロックの診断（3）


Java VisualVM
アプリケーション
ローカル
VisualVM
Eclipse (pid 42906)
handson3.DynamicDeadlock (pid 43
リモート
VM のコアダンプ
スナップショット
開始ページ
handson3 DynamicDeadlock (pid 43060)
概要
監視
スレッド
サンプラ
プロファイラ
handson3.DynamicDeadlock (pid 43060)
スレッド
スレッドを視覚化
ライブスレッド: 13
デーモンスレッド: 10
デッドロックが検出されました!
詳細情報を得るには、スレッドダンプを取得してください。
スレッドダンプ
タイムライン
表 | 詳細
表示:
すべてのスレッド
スレッド
1:10
1:20
[分:秒]
AWT-Shutdown
RMI TCP Connection(3)-1...
JMX server connection ti...
RMI Scheduler(0)
RMI TCP Connection(1)-1...
RMI TCP Accept-0
AWT-AppKit
Attach Listener
実行中
休眠中
待機
モニター

ハンズオン１

- 3スレッド間でデッドロックを起こすサンプルを作成せよ。

- tryLockを使って、上記サンプルでデッドロックが起こらないようにせよ。（可能であればリトライさせる。）

- ヒント：共有オブジェクトは3つ用意する。スレッド1はA→B、スレッド2はB→C、スレッド3はC→Aの順にロックを取得する。

デッドロック以外の生存事故 - 飢餓状態（1）

- StarvationSample.java

```
final Object sharedObject = new Object();

// 一旦ロックをつかんだら二度と解放しないタスク
Runnable infiniteLoopTask = new Runnable() {
    public void run() {
        synchronized(sharedObject) {
            while(true) {
            }
        }
    }
};

// ロックを取ろうとするが、取得できないタスク
Runnable starvationTask = new Runnable() {
    public void run() {
        while(true) {
            synchronized(sharedObject) {
                System.out.println("starvationTask");
            }
        }
    }
};
```

infiniteLoopTaskが一度sharedObjectのロックを取得すると、二度と解放しないので、starvationTaskはロックを取得することができず、ロック解放待ちのままウエイト状態となる。

デッドロック以外の生存事故 - 飢餓状態（2）

- 飢餓状態の診断は以下のように行う。
  - StarvationSampleを実行する。
  - コマンドラインからjpsを実行。
  - コマンドラインからjstackを実行。引数に、jpsコマンドで取得したStarvationSampleのpidを指定する。

デッドロック以外の生存事故 - 飢餓状態（3）

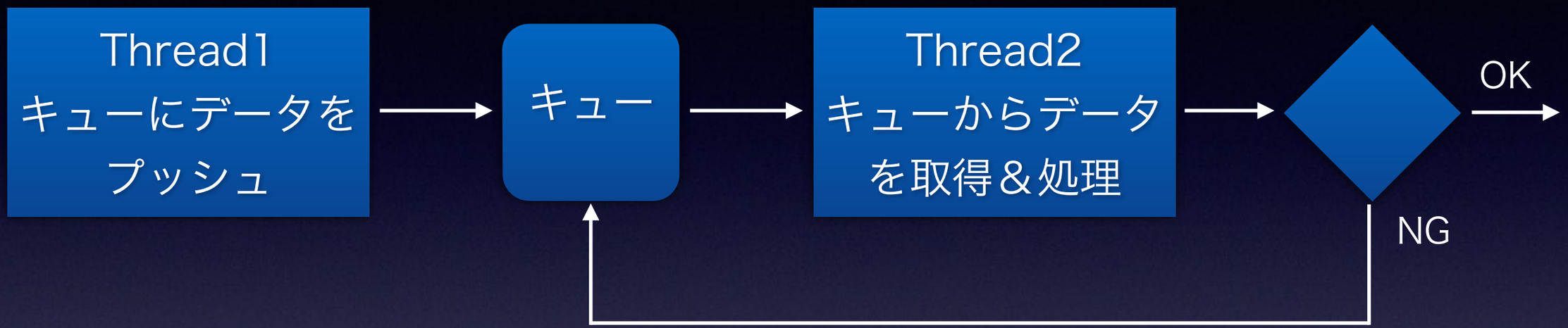
- jstackの出力（抜粋）

```
"pool-2-thread-1" prio=5 tid=7fdf8c8f0000 nid=0x1092a4000 waiting for monitor entry [1092a3000]
   java.lang.Thread.State: BLOCKED (on object monitor)
   at handson3.StarvationSample$2.run(StarvationSample.java:29)
   - waiting to lock <7f3120bd0> (a java.lang.Object)
   at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:439)
   at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:303)
   at java.util.concurrent.FutureTask.run(FutureTask.java:138)
   at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
   at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
   at java.lang.Thread.run(Thread.java:695)

"pool-1-thread-1" prio=5 tid=7fdf898c8800 nid=0x1091a1000 runnable [1091a0000]
   java.lang.Thread.State: RUNNABLE
   at handson3.StarvationSample$1.run(StarvationSample.java:19)
   - locked <7f3120bd0> (a java.lang.Object)
   at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:439)
   at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:303)
   at java.util.concurrent.FutureTask.run(FutureTask.java:138)
   at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access
$301(ScheduledThreadPoolExecutor.java:98)
   at java.util.concurrent.ScheduledThreadPoolExecutor
$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:206)
   at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
   at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
   at java.lang.Thread.run(Thread.java:695)
```

Thread1が7f3120bd0をロックしており、Thread2が7f3120bd0のロック取得待ちとなっていることが分かる。

デッドロック以外の生存事故 - ライブロック



エラーのとき、リトライのため、キューにデータを戻す

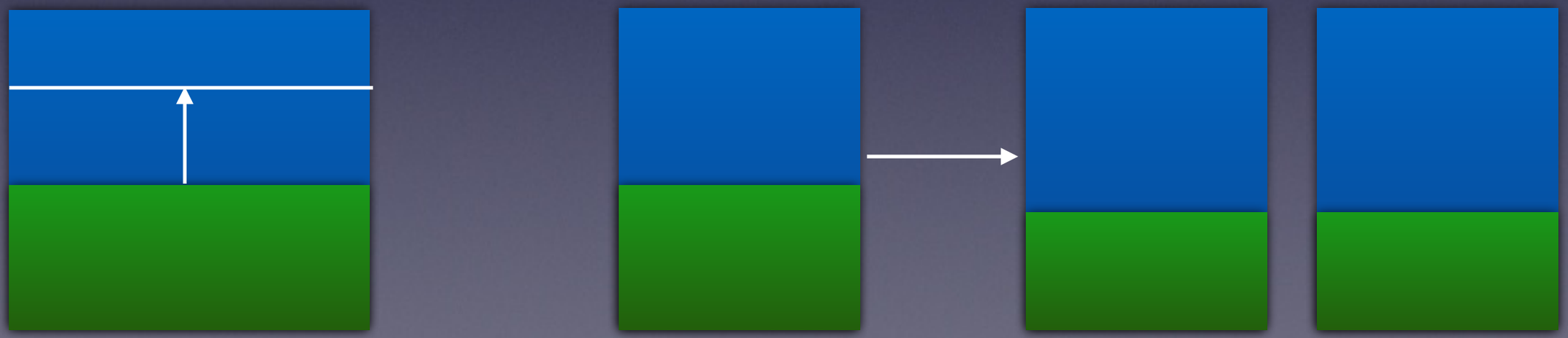
何回リトライしてもエラーになるデータの場合、無限ループになる

リトライすべきエラーと、そうでないエラーを、区別する必要がある。

実行性能

実行性能の向上を目指して並行処理を使う目的

- 既存の処理資源をより有効に利用すること（余ってるCPUパワーの有効活用など）。
- 今後さらに処理資源（CPU、メモリ、IOの帯域幅）を増やすことに意味があるシステムを作りあげること。



実行性能の向上とスケーラビリティの向上

- 実行性能の向上
    - 計算結果のキャッシュや、アルゴリズムのチューニング
      O（n^2）→O（n log n）を行うこと。
- スケーラビリティとは
    - 計算資源（CPU、メモリ、I/Oの帯域幅など）を増やしたときにスループットや処理能力がどれぐらい向上するかしないかを計る測度のこと。
- スケーラビリティの向上
    - 処理資源を増やして処理能力を向上させること。

実行性能の向上を目指す際の注意

- 最適化することよりも、正しいコードを書くことを優先すること。高速化は、必要な速さが得られないときだけ取り組みべき。（バグリやすい）

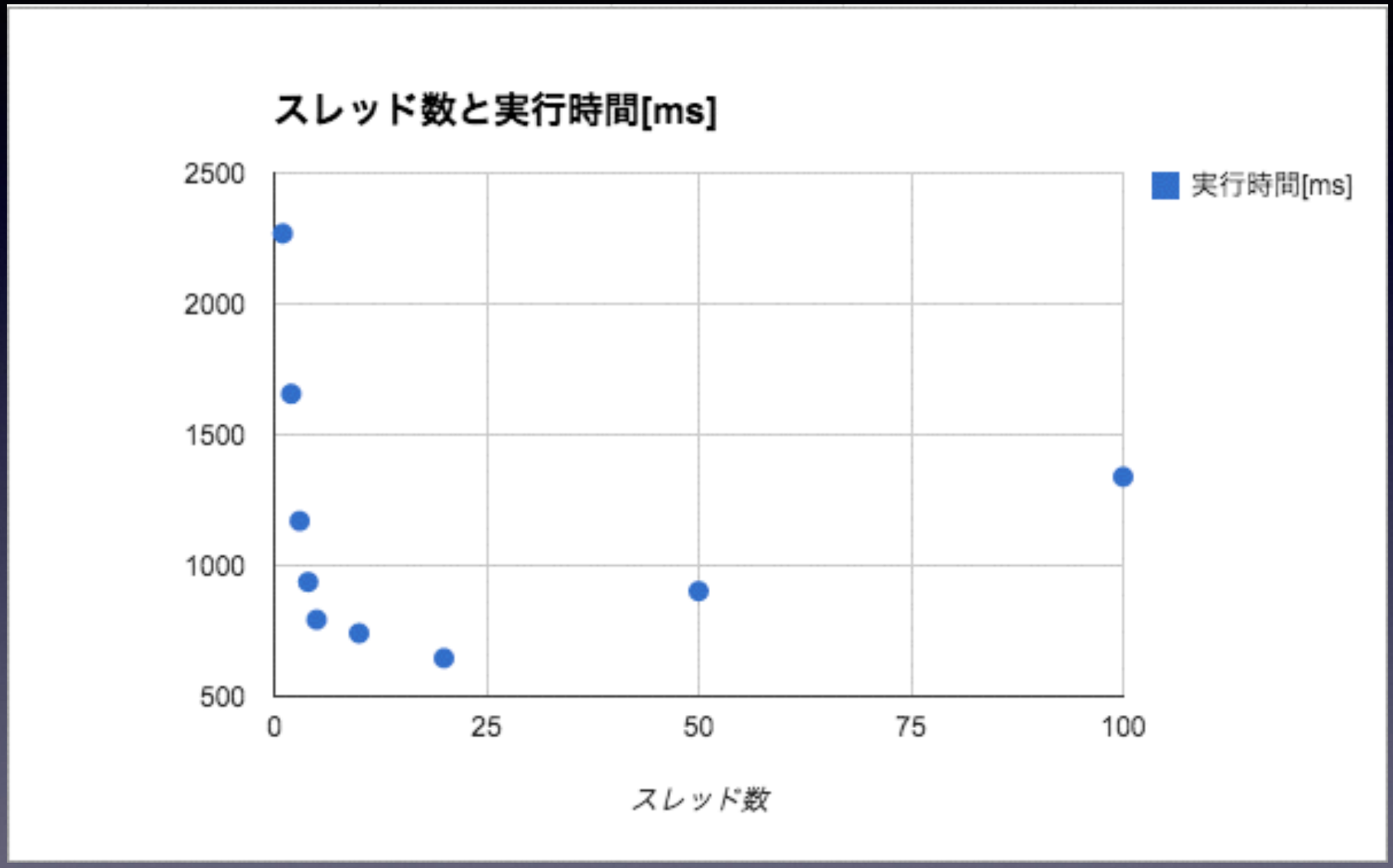
- 実測せよ、推測するな。

- 大きなデータ集合の場合は、バブルソートよりも、クイックソートの方が効率的だが、
  小さなデータ集合の場合は、バブルソートの方が効率的。
  →処理するデータの性質などによるため、実測が大事になる

実行性能の向上（1）

- ファイルを読み込んで行数を返すタスクを作成し、スレッド数に応じて実行性能がどのように変化するか見る。
- jvisualvmを実行。
- NThreadsを実行。
- コンソールに実行時間、jvisualvmにCPU、メモリ使用量の経過が表示される。

実行性能の向上（2）



20スレッドまでは、スレッド数を増やすほど実行時間が短くなっているが、50スレッドより増やしても、むしろ実行時間が長くなっている。
→スレッドのコストが増大したことや、I/Oの競合が原因として考えられる。

ハンズオン２

- 元本（円）と利率（%）を与えると、複利計算を行い、n年後の運用結果を求めるタスクを作成せよ。（n年分の運用結果を全て保持します）
- 1秒以内で出来るだけ未来分を含んだ運用結果を表示するプログラムを作成せよ。
- ヒント：n年後の運用結果は、元本+(1+利率)^nで求まる。

まとめ

- タスクの実行
  - Executorフレームワークを使うとタスクの依頼と実行ポリシーを分離でき、豊富な種類の実行ポリシーを使えます。
- タスクのキャンセル
  - Javaにはキャンセルのために利用できるインタラプトの仕組みがありますが、それを正しく使うのはデベロッパの責任です。Executorフレームワークを使うと、キャンセルできるタスクやサービスを簡単に構築できます。

リソース

- ハンズオンソースコード
  https://github.com/taka2/MultiThreadHandsOn

- Java並行処理プログラミング ―その「基盤」と「最新API」を究める―
  ISBN: 978-4797337204

宿題

- 指定した数までの素数を生成するタスクを作成し、指定秒数内で出来るだけ素数を表示するプログラムを作成せよ。
- JDKのソースツリーを再起的に探索し、ディレクトリに含まれるJavaソースのファイル名と行数を返すタスクを作成せよ。
  Executorフレームワークを使ってタスクを並行に実行するように変更せよ。