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）

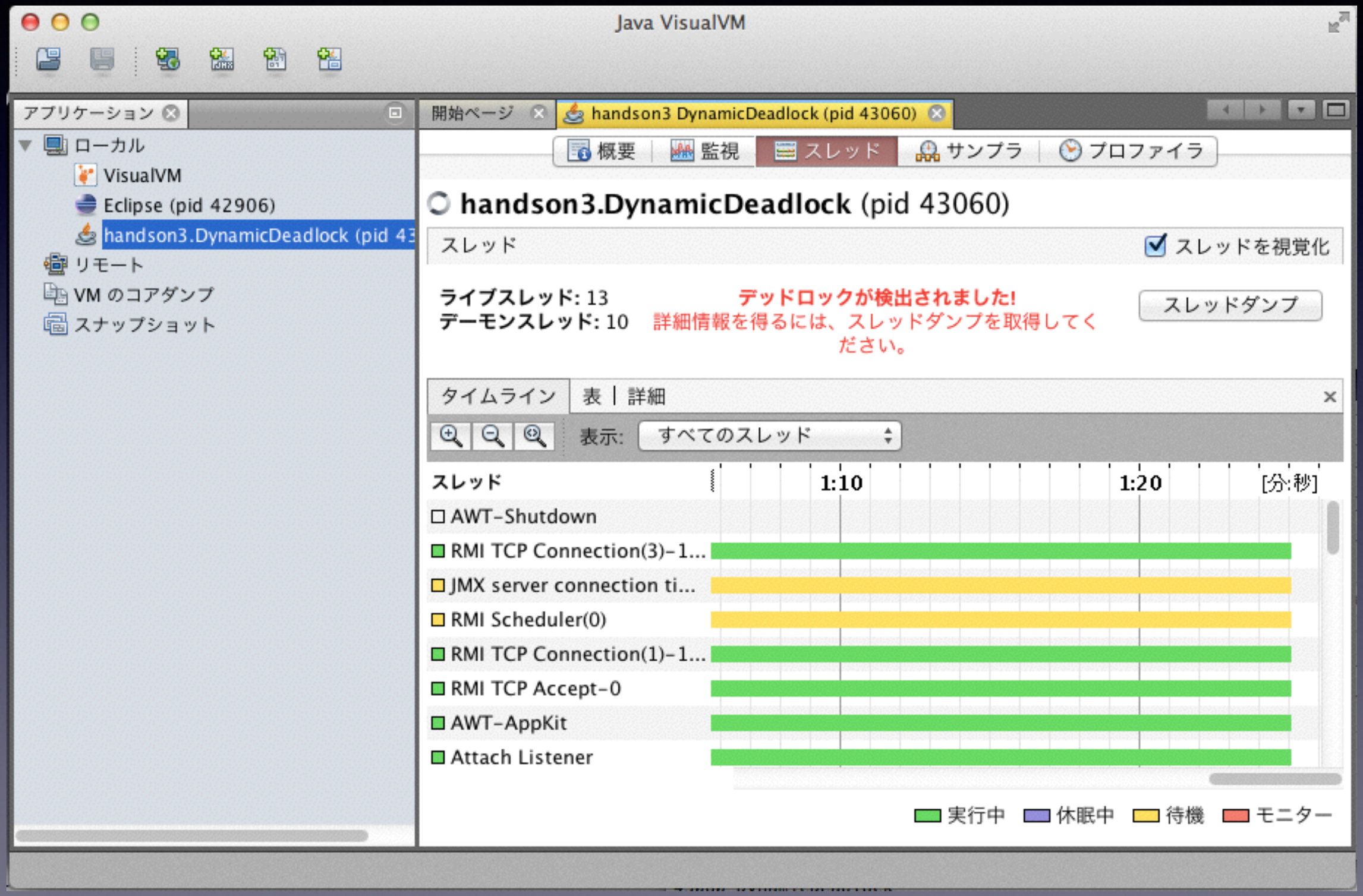
- デッドロックが起きてもJVMが例外を投げたりするわけではない（ダンマリ）。
  診断は以下のように行う。jps/jstackを使う場合。

  - DyamicDeadlock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DynamicDeadloc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
  - DyamicDeadlock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。

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

- 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。

タスクのキャンセル

タスクのキャンセルが必要な理由

- ユーザがキャンセルをリクエストした
- 時間制限のある活動
- アプリケーションイベント
- エラー
- シャットダウン

ユーザがキャンセルをリクエストした

- GUIの例を見てみる。
  - CancellableGUIを実行する。
  - clickボタンをクリックする。
    →3秒ほど待つと、ラベルが「clicked.」に変化する。
  - 次に、clickボタンをクリックし、3秒待たずにとなりのcancelボタンをクリックする。
    →cancelボタンをクリックしたタイミングで、ラベルが「cancelled.」に変化する。

時間制限のある活動（1）

- 5秒間FizzBuzzする例を見てみる。
  - TimerLimitTaskを実行する。
    →ひたすらFizzBuzzを表示し続け、5秒ほどすると停止する。

時間制限のある活動（2）

- TimeLimitTask.java

```
Runnable r = new Runnable() {
    public void run() {
        long i=1;
        while(true) {
            printFizzBuzz(i);
            i++;
            try {
                Thread.sleep(0);
            } catch(InterruptedException e) {
                // インタラプトされたらループ終了
                break;
            }
        }
    }
```

あえてThread.sleep(0)を入れることで、外部から割り込む余地を与えている。
Thread.sleep(0)がないと、キャンセルできない。

時間制限のある活動（3）

- TimeLimitTask.java

```
// スレッド開始
ExecutorService executor = Executors.newCachedThreadPool();
Future<?> task = executor.submit(r);

try {
    task.get(TIMEOUT, TimeUnit.SECONDS);
} catch(TimeoutException e) {
    // タスクはfinallyブロックでキャンセルされる
} catch(InterruptedException e) {
    // タスクはfinallyブロックでキャンセルされる
} catch(ExecutionException e) {
    // タスク中で投げられた例外：再投する
    throw new RuntimeException(e);
} finally {
    // タスクがすでに完了していたら無害
    task.cancel(true);// 実行中ならインタラプトする
}

executor.shutdown();
```

タイムアウト、正常終了に関わらず、cancelメソッドを呼び出す。
タスクが取り消せなかった場合（既に正常終了していた場合）はfalseを返す。
タスクが取り消せた場合はtrueを返す。（戻り値を無視しているため、いずれにしても正常扱い）

シャットダウン可能なwebサーバ（１）

- /stopをリクエストすると停止するwebサーバのサンプルを見てみる。
  →実用上はありえないが。。

  - CancellableWebServer.javaを実行する。

  - 「クライアントからの接続を待ち受けます。」が表示されたら、ブラウザから以下のURLを開く。
    http://localhost:8888/
    3秒ほど待つと、ブラウザに日時が表示される。

  - ブラウザから以下のURLを開く。
    http://localhost:8888/stop
    ブラウザに日時が表示され、webサーバも停止する。

シャットダウン可能なwebサーバ（2）

- CancellableWebServer.java

```
while(true) {
    System.out.println("クライアントからの接続を待ち受けます。");
    final Socket client = ss.accept();
    System.out.println("クライアントから接続されました。");
    Callable<Boolean> task = new Callable<Boolean>() {
        public Boolean call() throws IOException {
            return Boolean.valueOf(handleRequest(client));
        }
    };
    Future<Boolean> future = executor.submit(task);
    if(!future.get()) {
        // webサーバ終了
        break;
    }
    System.out.println("クライアントを切断しました。");
}

executor.shutdown();
```

Handlerから停止かどうかの判定結果を返すようにして、停止と判断されたら、無限ループ終了。（クライアントからの待ち受け終了）

shutdownとshutdownNowの違い

	shutdown	shutdownNow
shutdown後のタスクの受け付け	しない	しない
タスクを受け付けた後、未実施のタスク（※１）	実行する	実行しない（※２）

※１・・・スレッドプール数5に対して、8のタスクがsubmitされた場合、3つのタスクは実行待ちになる。
※２・・・List<Runnable>として戻り値で返される。

shutdownとawaitTermination

- ExecutorShutdownSample1.java

```
ExecutorService executor = Executors.newCachedThreadPool();
for(int i=1; i<=10; i++) {
    executor.submit(new MyTask(i));
}
executor.shutdown();

// シャットダウン後は新たなタスクは受け付けられない
try {
    executor.submit(new MyTask(100));
} catch(RejectedExecutionException e) {
    e.printStackTrace();
}

// 最大5秒シャットダウンの完了を待つ
try {
    executor.awaitTermination(5, TimeUnit.SECONDS);
} catch(Exception e) {
    e.printStackTrace();
}
```

ExecutorService#shutdownを呼んでも、各タスクにインタラプトを要求するだけで、すぐに終了するわけではない。
submitしたタスクが、完了／タイムアウト／インタラプトのいずれかするまで待つ。

UncaughtExceptionHandler

- UncaughtExceptionHandlerSample.java

```
// UncaughtExceptionHandlerを定義
UncaughtExceptionHandler ueh = new UncaughtExceptionHandler() {
    public void uncaughtException(Thread t, Throwable e) {
        // 通常、標準エラー出力に出るところ、標準出力に表示するよう変更
        e.printStackTrace(System.out);
    }
};

// カレントスレッドにHandlerをセット
Thread.setDefaultUncaughtExceptionHandler(ueh);

// ランタイム例外を（意図的に）スロー
throw new RuntimeException("Runtime Exception!");
```

通常、catchしていない例外は、標準エラー出力にスタックトレースが表示されるが、この例では、標準出力にスタックトレースが表示される。
アプリログへの出力や、メール通知などに利用すると良いらしい。

JVMのシャットダウンフック

- ShutdownHookSample.java

```
// シャットダウンフックを定義
Runnable shutdownHookTask = new Runnable() {
    public void run() {
        System.out.println("shutdown hook");
    }
};

// シャットダウンフックを追加
Runtime.getRuntime().addShutdownHook(new Thread(shutdownHookTask));
```

複数のシャットダウンフックを設定することができるが、並行に動作するので、スレッドセーフにする必要がある。一つだけ設定した方が分かりやすい。
→一時ファイルの削除や、OSがクリーンアップしない資源の削除に使うと良いらしい。

デーモンスレッド（1）

- スレッドには、正規のスレッドとデーモンスレッドの2種類がある。

- JVMが始動するとき、正規のスレッドはメインスレッドのみ。その他は全てデーモンスレッド。（ガーベッジコレクタ、ファイナライザなど）

- スレッドが終了したとき、JVM内に残っているスレッドがデーモンスレッドのみの場合、JVMのシャットダウンを開始する。

デーモンスレッド（2）

- どのようなスレッドが実行されているか見てみる。
  - FixedThreadPoolWebServerを実行する。
  - コマンドラインからjpsを実行。
    →FixedThreadPoolWebServerの左横に表示された数字を覚える。
  - コマンドラインからjstack <pid>を実行。
    →スレッドダンプが表示される。
  - jvisualvmコマンドを実行。
    →GUIが起動する。
  - ブラウザからhttp://localhost:8888へアクセスする。
    →スレッドが起動する様が確認できる。

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。