

# マルチスレッド ハンズオン

第2回 タスクの実行とキャンセル

# アジェンダ

- ・ 前回のおさらい
- ・ 並行コレクション
- ・ タスクの実行
- ・ タスクのキャンセル

前回のおさらい

# 宿題の解答

- java.text.DateFormat.javaの先頭の方
- ```
26 /*
27  * (C) Copyright Taligent, Inc. 1996 - All Rights Reserved
28  * (C) Copyright IBM Corp. 1996 - All Rights Reserved
29  *
30  * The original version of this source code and documentation is copyrighted
31  * and owned by Taligent, Inc., a wholly-owned subsidiary of IBM. These
32  * materials are provided under terms of a License Agreement between
33  * Taligent
34  * and Sun. This technology is protected by multiple US and International
35  * patents. This notice and attribution to Taligent may not be removed.
36  * Taligent is a registered trademark of Taligent, Inc.
37  *
38  */
```



# 宿題の解答

ページ ノート

閲覧

編集

履歴表示

検索



## Taligent

**Taligent**(タリジェント)は、1992年にアップルコンピュータとIBMが共同で設立した会社、およびその会社の開発していた**オブジェクト指向**の次世代オペレーティングシステム (OS) の名称である<sup>[1][2][3]</sup>。これは1991年のIBMとアップルの包括的提携の実現化の1つ<sup>[4]</sup>で、1994年にはヒューレット・パッカードも資本参加した<sup>[5]</sup>。

### 目次 [非表示]

- 1 概要
- 2 参照
- 3 関連項目
- 4 外部リンク

# synchronized(this)の.NETでの問題

- ・ stackoverflowで該当するトピックを見つけた。  
<http://stackoverflow.com/questions/251391/why-is-lockthis-bad>
- ・ MSDNにも記載されている。  
<http://msdn.microsoft.com/en-us/library/c5kehkc%28v=vs.110%29.aspx>
- ・ thisをロックする他の処理があるかも知れないので、thisをlock文で使うのはよくない。
- ・ Javaでも同様の問題が起こりうる。

# synchronized(this)の.NETでの問題

## ダメな例

- SynchronizedThis.java

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

```
new Thread(new Runnable() {  
    public void run() {  
        // 外部からインスタンスのロックを取得  
        synchronized(sharedObject) {
```

...

```
static class SharedObject {  
    public void method1() {  
        synchronized(this) {  
            System.out.println("method1 called.");  
        }  
    }  
}
```

sharedObjectに対するロックがクラス内部、クラス外部の両方から取得できてしまう。

# synchronized(this)の.NETでの問題

## 良い例

- PrivateLockObject.java

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

```
new Thread(new Runnable() {  
    public void run() {  
        // 外部からインスタンスのロックを取得  
        synchronized(sharedObject) {
```

```
... 
```

```
static class SharedObject {  
    private Object lockObject = new Object();  
    public void method1() {  
        synchronized(lockObject) {  
            System.out.println("method1 called.");  
        }  
    }  
}
```

プライベートメンバlockObjectを用意することで、外部からインスタンスのロックに関わらずmethod1の同期化が可能。



# クラスメソッドの同期の話

- SynchronizedStaticMethod.java

```
public synchronized static void method1() {  
    try {  
        System.out.println("method1 called.");  
        Thread.sleep(3000);  
    } catch (InterruptedException e) {  
        // 何もしない  
    }  
}  
  
public synchronized static void method2() {  
    try {  
        System.out.println("method2 called.");  
        Thread.sleep(3000);  
    } catch (InterruptedException e) {  
        // 何もしない  
    }  
}
```

method1とmethod2はクラスに対するロックを取得するので、method1とmethod2を複数スレッドから同時にコールしても、メソッドの実行が完了してから、もう一方のメソッドが実行される。

並行コレクション

# MapのiterateでConcurrentModificationExceptionが発生する問題（１）

- NonThreadSafeMap1.java

```
final Map<Integer, Integer> map = Collections.synchronizedMap(new
HashMap<Integer, Integer>());

new Thread(new Runnable() {
    public void run() {
        for(int i=0; i<NUM_DATA; i++) {
            map.put(i, i);
        }
    }
}).start();

new Thread(new Runnable() {
    public void run() {
        for(int i=0; i<NUM_LOOP; i++) {
            Iterator<Integer> ite = map.keySet().iterator();
            while(ite.hasNext()) {
                ite.next();
            }
        }
    }
}).start();
```

java.util.ConcurrentModificationExceptionが発生してしまう。なぜか。

# MapのiterateでConcurrentModificationExceptionが発生する問題（2）

・ NonThreadSafeMap2.java

```
final Map<Integer, Integer> map = Collections.synchronizedMap(new HashMap<Integer, Integer>());
```

```
new Thread(new Runnable() {
    public void run() {
        for(int i=0; i<NUM_DATA; i++) {
            map.put(i, i);
        }
    }
}).start();
new Thread(new Runnable() {
    public void run() {
        for(int i=0; i<NUM_LOOP; i++) {
            synchronized(map) {
                Iterator<Integer> ite = map.keySet().iterator();
                while(ite.hasNext()) {
                    ite.next();
                }
            }
        }
    }
}).start();
```

iterateしている間、mapをロックしてしまえば、問題は回避できるが、サイズの大きいMapを処理する場合、処理効率が悪い。（ループしている間、mapへの操作ができなくなる。）



# MapのiterateでConcurrentModificationExceptionが発生する問題（3）

- ThreadSafeMap.java

```
final Map<Integer, Integer> map = new ConcurrentHashMap<Integer, Integer>();

new Thread(new Runnable() {
    public void run() {
        for(int i=0; i<NUM_DATA; i++) {
            map.put(i, i);
        }
    }
}).start();

new Thread(new Runnable() {
    public void run() {
        for(int i=0; i<NUM_LOOP; i++) {
            Iterator<Integer> ite = map.keySet().iterator();
            while(ite.hasNext()) {
                ite.next();
            }
        }
    }
}).start();
```

ConcurrentHashMapを使うことで、ConcurrentModificationExceptionが発生してしまう問題と、処理効率の問題を同時に解決できる。

# ConcurrentHashMapの動作（１）

- ・ ConcurrentHashMapは、なぜConcurrentModificationExceptionを発生させずに処理出来るのか。
  - ・ HashMapの場合は、イテレータの作成後に、イテレータ自体のremove メソッド以外の方法でマップが構造的に変更されると、イテレータは ConcurrentModificationException をスローします。
  - ・ ConcurrentHashMapの場合は、ある時点またはイテレータ/列挙の作成以降のハッシュテーブルの状態を反映する要素を返します。これらが ConcurrentModificationException をスローすることはありません。しかし、一度に１つのスレッドのみがイテレータを使用するように設計されています。

# ConcurrentHashMapの動作 (2)

- ConcurrentHashMapSample.java

```
new Thread(new Runnable() {
    public void run() {
        for(int i=0; i<NUM_DATA; i++) {
            map.put(i, i);
        }
    }
}).start();
new Thread(new Runnable() {
    public void run() {
        System.out.println("map.size(iterator取得前) = " + map.size());
        Iterator<Integer> ite = map.keySet().iterator();
        System.out.println("map.size(iterator取得後) = " + map.size());
        int count = 0;
        while(ite.hasNext()) {
            ite.next();
            count++;
        }
        System.out.println("iteratorループカウント = " + count);
    }
}).start();
```

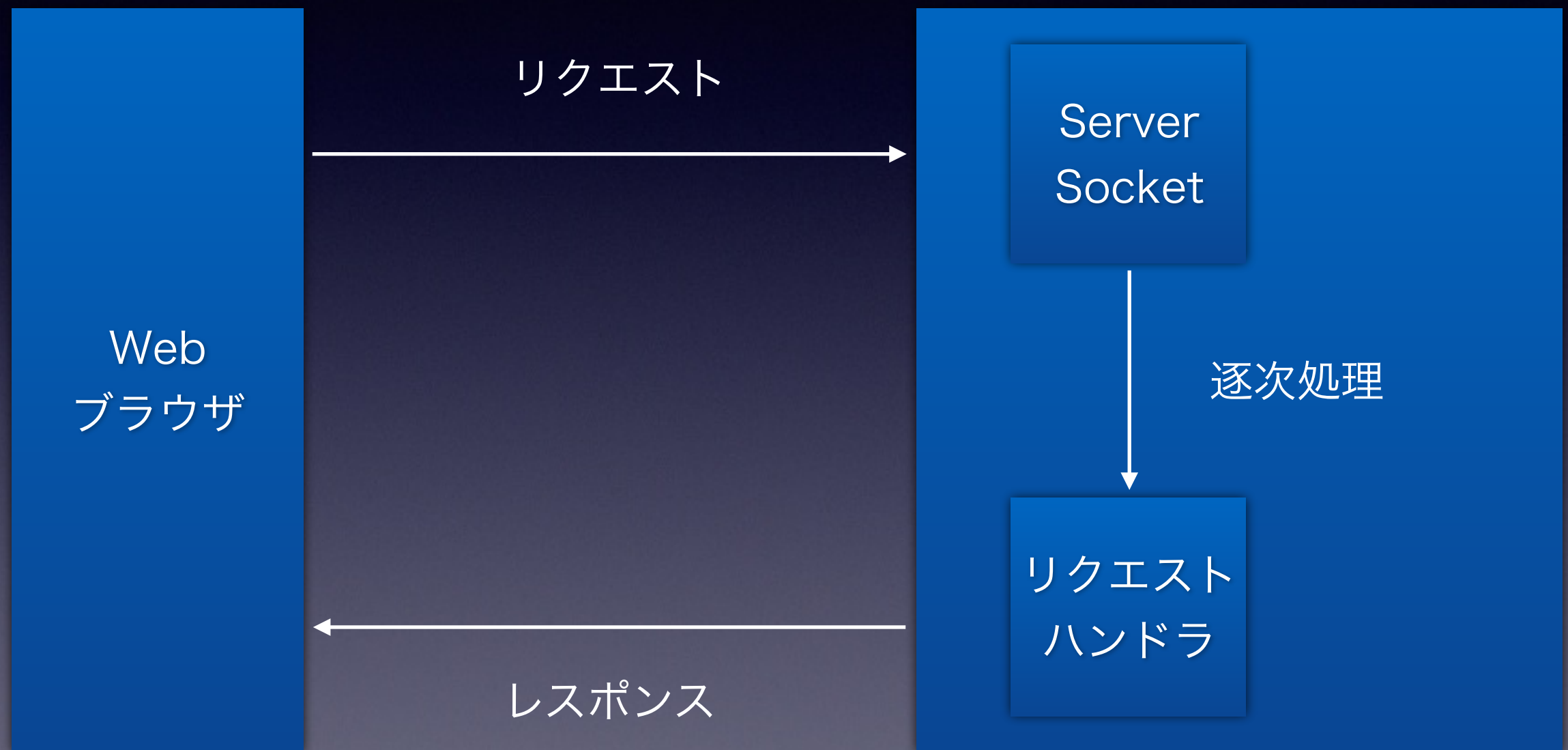
iteratorを呼んだからといって、mapの更新が出来なくなるわけではない。  
iteratorの結果を安定させるためには、mapに対する同期化が必要。

# タスクの実行



# 逐次処理webサーバの例（１）

- SequentialWebServer.java



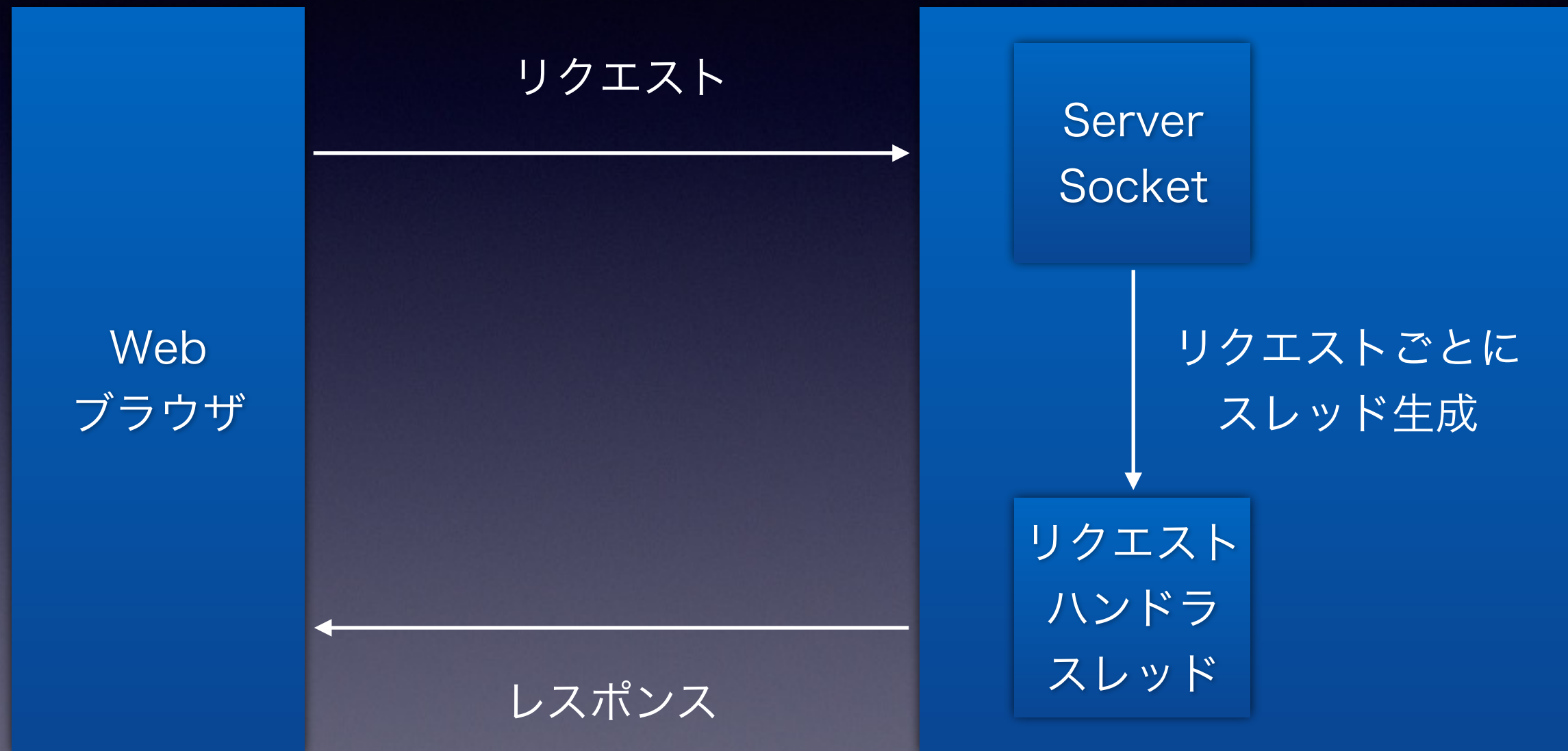
一つずつリクエストを処理する。前のリクエストが終わらないと、次のリクエストを処理できない。サーバ側が時間がかかることで、接続エラーになったり、クライアント側でタイムアウトしたりする。

# 逐次処理webサーバの例 (2)

- ・ 逐次処理webサーバの動作を試みる。
  - ・ SequentialWebServer.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。
  - ・ WebClient.java (100スレッドでアクセスするwebクライアント) を実行する。  
→接続エラーになったり、クライアント側でタイムアウトが発生したりするはず。
- ・ SequentialWebServerを停止する。

# 無制限スレッド生成webサーバの例（１）

- UnlimitedThreadWebServer.java



リクエストを受け付けるたびに、リクエストハンドラスレッドを生成する。  
前のリクエストの処理が終わっていなくても、次のリクエストを受け付けることが可能。**サーバ側のリソースを無制限に使用する。**



# 無制限スレッド生成webサーバの例（2）

- ・ 無制限スレッド生成webサーバの動作を試みる。
  - ・ UnlimitedThreadWebServer.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。
  - ・ WebClient.java（100スレッドでアクセスするwebクライアント）を実行する。  
→クライアント側にコネクションエラーが出たり、サーバ側でOutOfMemoryError発生したりするはず。
- ・ UnlimitedThreadWebServerを停止する。

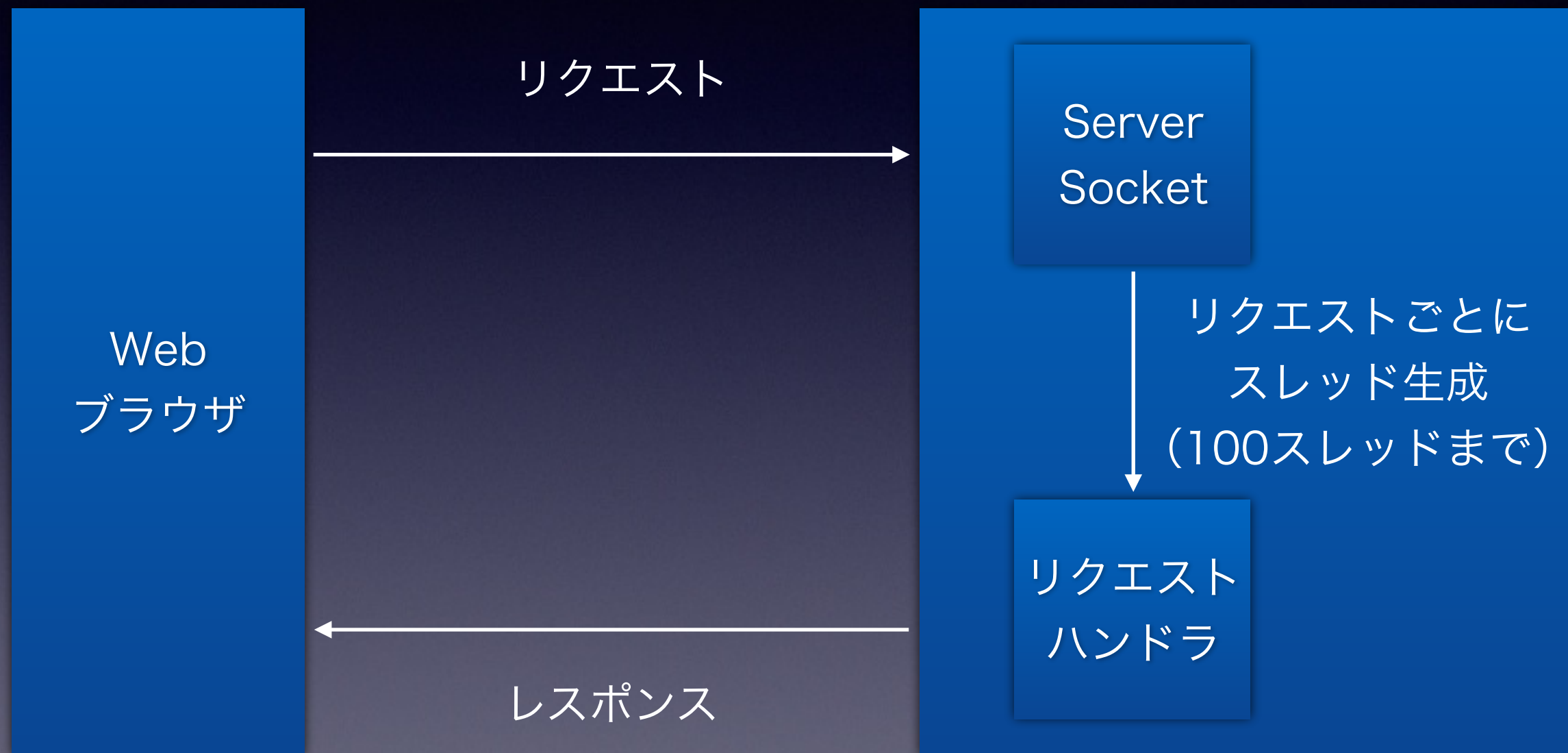


# 無制限スレッド生成webサーバの例（3）

- ・ スレッドを無制限に生成することによる弊害
  - ・ 資源の消費
  - ・ 安定性
- ・ スレッドのライフサイクルのオーバーヘッド  
→スレッドの生成／破棄にかかるコスト

# Executorを使ったwebサーバの例（1）

- FixedThreadPoolWebServer.java



リクエストを受け付けるたびに、リクエストハンドラスレッドを生成する。  
前のリクエストの処理が終わっていなくても、次のリクエストを受け付けることが可能。スレッドは最大100まで生成される。

# Executorを使ったwebサーバの例（2）

- ・ Executorを使ったwebサーバの動作を試みる。
  - ・ FixedThreadPoolWebServer.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。
  - ・ WebClient.java（100スレッドでアクセスするwebクライアント）を実行する。  
→正常終了するはず。（サーバから取得した日時が100行出力されればOK）
- ・ FixedThreadPoolWebServerを停止する。



# 4種類のスレッドプール

- ・ `newFixedThreadPool`
  - ・ 固定数のスレッドを再利用するスレッドプールを作成します。  
→リソースの消費量を一定量に抑える。
- ・ `newCachedThreadPool`
  - ・ 必要に応じ、新規スレッドを作成するスレッドプールを作成しますが、利用可能な場合には以前に構築されたスレッドを再利用します。（スレッド数は無制限）  
→短期間のタスクをたくさん起動するアプリケーションに向く。
- ・ `newSingleThreadExecutor`
  - ・ 単一のワーカースレッドを使用する `executor` を作成します。
- ・ `newScheduledThreadPool`
  - ・ 指定された遅延時間後、または周期的にコマンドの実行をスケジュールできる、スレッドプールを作成します。



# Timerと

## ScheduledThreadPoolExecutor (1)

- ・ TimerとScheduledThreadPoolExecutorの動作を試みる。
- ・ TimerSample1.javaを実行する。
- ・ TimerSample2.javaを実行する。
- ・ ScheduledThreadPoolSample.javaを実行する。
- ・ (補足) タスクの中身は、MyTimerTask.javaに記述されている。
- ・ どういう違いがあったか。

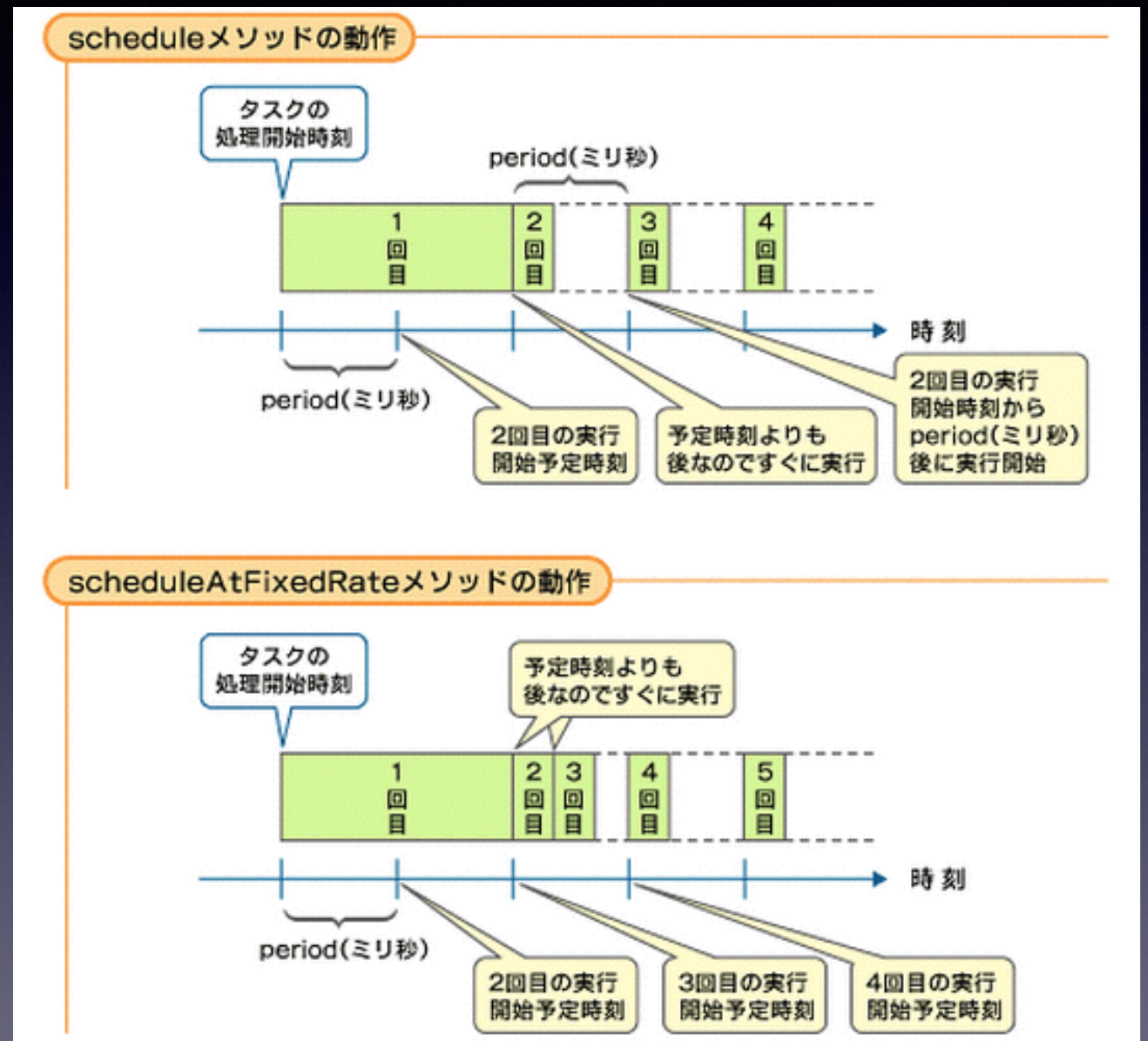
# Timerと

## ScheduledThreadPoolExecutor (2)

- Timer#scheduleとTimer#scheduleAtFixedRateの違い

- 下記、@ITの記事参照。

[http://  
www.atmarkit.co.  
jp/fjava/  
javatips/  
078java008.html](http://www.atmarkit.co.jp/fjava/javatips/078java008.html)



# Timerと

## ScheduledThreadPoolExecutor (3)

- ・ TimerとScheduledThreadPoolExecutorの違い
  - ・ Timerは単一スレッドで実行されるのに対し、ScheduledThreadPoolExecutorは内部でスレッドプールを利用する。
  - ・ タスクが例外を発生させても、実行元に伝搬されない。
  - ・ ※動作自体は同じ。（遅延タスクがあっても、次のタスクが追い越すことはない；APIドキュメントに記載あり）



# 結果を返すタスクCallableとFuture（1）

- ・ java.lang.Runnableインタフェース  
public void run()
  - ・ 結果を返せないし、例外も投げられない。
- ・ java.util.concurrent.Callableインタフェース  
public V call()
  - ・ 結果を返せるし、例外も投げられる。



# 結果を返すタスクCallableとFuture (2)

- CallableWebClient.java

```
// タスク
```

```
Callable<String> task = new Callable<String>() {  
    public String call() throws IOException {  
        return CallableWebClient.sendRequest();  
    }  
};
```

```
// リクエスト
```

```
ExecutorService executor = Executors.newCachedThreadPool();  
List<Future<String>> futures = new ArrayList<Future<String>>();  
for(int i=0; i<NUM_THREADS; i++) {  
    Future<String> future = executor.submit(task);  
    futures.add(future);  
}
```

```
// 結果受け取り
```

```
for(int i=0; i<NUM_THREADS; i++) {  
    Future<String> future = futures.get(i);  
    try {  
        System.out.println(future.get());  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

# ハンズオン 1

- ・ 指定したディレクトリのファイル一覧を取得するCallableタスクを作成せよ。
- ・ FixedThreadPoolExecutorを使って、複数ディレクトリのファイル一覧を並行で取得するプログラムを作成せよ。
- ・ ヒント：指定ディレクトリのファイル一覧は、以下のコードで取得できる。  

```
File[] files = new File("ディレクトリ").listFiles();  
ex) File[] files = new File("src/handson").listFiles();
```
- ・ ヒント 2：例えば、1スレッド目はsrc/handson配下、2スレッド目はsrc/handson2配下を読み取るなど、タスク分割するよう実装する。

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の例を見してみる。
- ・ 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試みる。  
→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を確認できる。



# 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、 $\text{元本} + (\text{元本} \times \text{利率})^n$ 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](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。