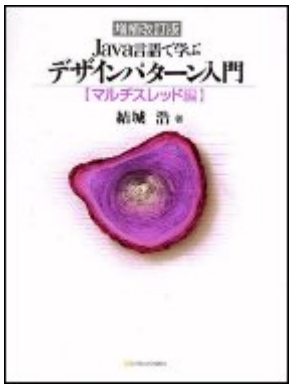
経営情報システム学特論１
１１～１２．Javaマルチスレッドパターン

SS専攻 経営情報システム学講座 客員
石川 冬樹
f-ishikawa@nii.ac.jp

参考書

- 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
  - 12個のパターンを解説
  - 同時にJavaの並行性に関する仕様と向き合うことにもなる



- http://www.hyuki.com/dp/dp2.html
- 今回APIはJava 8を参照

補足：デザインパターン

- しばしば現れる問題に対する，典型的な設計の考え方を再利用
  - 設計そのものではない（パターン）
- GoFパターンが有名（常識）
  - Erich Gammaら”Gang of Four”による，オブジェクト指向設計のための23のパターン

補足：GoFパターンの例

- 左右どちらの設計がよいだろう？
  （どういう状況，基準で）

```
class Mathematic {

  public Data sort(Data data){
    switch(settings) {
    case QUICK:
      return quickSort(data);
    case BUBBLE:
      return bubbleSort(data);
    default: ...
  }
}
```

```
class Mathematic {
  Sorter sorter;
  public Data sort(Data data){
    return sorter.sort(data);
  }
  public void setSorter(Sorter s){
    sorter = s;
  }
}
```

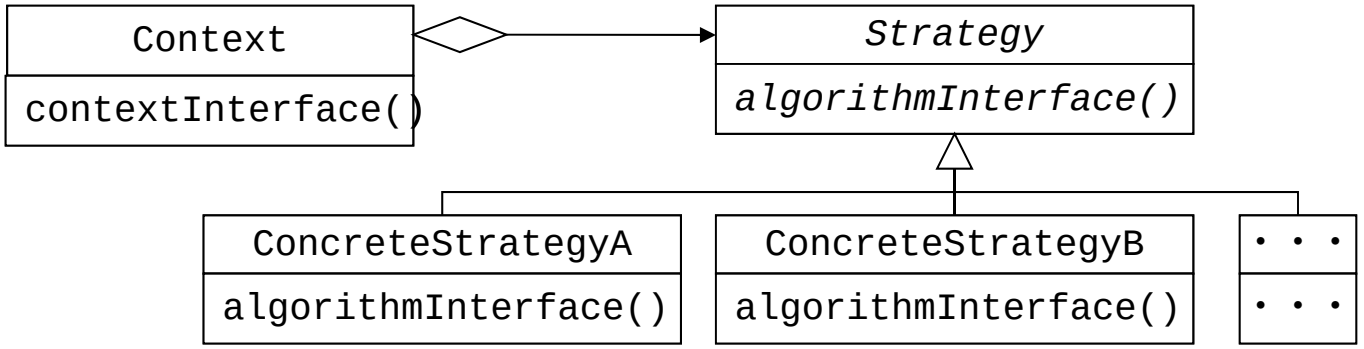
```
abstract class Sorter {
  public abstract Data sort(Data);
```

```
class QuickSorter extends Sorter {
  public Data sort(Data) { ... }
```

```
class BubbleSorter extends Sorter {
  public Data sort(Data) { ... }
```

補足：GoFパターンの例

- Strategyパターン
  - 多数のアルゴリズムがある
  - それらを利用するクラスに組み込んでしまわず，独立に定義や追加ができるようにしたい



目次

- 基本的な排他制御
  - Single Threaded Executionパターン
  - Javaのメモリに関する処理仕様
  - 関連するJava標準ライブラリ
- 基本的な同期制御
- 読み書きを区別した制御
- タスクの実行

Single Threaded Execution： 概要

- Single Threaded Executionパターン
  - 目的： 共有資源にアクセスする動作区間（*critical section/region*）が，同時に高々1つのスレッドのみに実行されるようにする
  - 防ぎたい問題： 同一データへの複数スレッドの干渉（*data race*・*race condition*）により共有資源（変数）の値が予期せぬものとなる
  - 対応策： 複数のスレッドから呼び出すと問題がある（*スレッドセーフ*ではない）処理に対し，ロックの確保により排他制御を行う

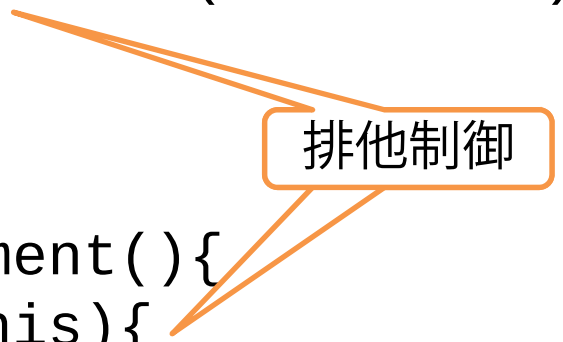
Single Threaded Execution： 実装

- Javaでの典型的な実装： synchronizedキーワード（メソッドまたはブロック）

```
private int x;

public synchronized add(int value){
  x = x + value;
}

public void increment(){
   synchronized(this){
     x++;
   }
}
```



Single Threaded Execution： 考察

- 使う状況
  - 複数スレッドから起動される処理あり，状態の変更を含む場合
- 注意点
  - 異なる順序で複数ロックを確保するようにすると，デッドロックが生じる
  - ロック処理のオーバーヘッドおよびスレッド間の衝突により性能が低下するため，共有資源の利用とクリティカルセクションの長さを最小限にする
  - 何をどのロックで保護しているかを意識する
  - 継承の際の徹底に留意する（後述）

Java詳細： 継承とsynchronized

- 継承した際に，メソッドがsynchronizedであることが強制的に引き継がれるわけではない
  - メソッド定義にsynchronizedを付けても，内部ではブロックに展開されるため，オーバーライドされる

```
... method1() {synchronized(this){...}}
```

  - 強制するには「中身」だけオーバーライドさせる

```
public synchronized final ... method1(...){
   return method1_impl(...);
}

protected ... method1_impl(...){
   ...
}
```

継承不可

原子性

- 排他制御は，ある処理を*原子的*（*atomic*）にするという意味があるとも見なせる
  - その処理の実行中に他のスレッドによる処理が挿入されることはない
  - 例： x := x+1（カウンターのインクリメント）がもしも原子的でないと・・・（第3回より）
  - 別の例： if(x==false){ x=true; }（空きを確認して確保）がもしも原子的でないと・・・（*test-and-set*として知られる処理）

Java詳細： 原子性

- 値に対する単一の代入や参照は基本的に原子的
- long/doubleにおいてはそれらも原子的ではない
  - 処理系依存だが典型的には複数回のメモリアクセスになるため，仕様で原子的でないと決めている
- java.utilパッケージにおけるCollection（set/list）やMapは，スレッドセーフでない
  - 複数スレッドから要素の追加などを行うと，本講義で見てきたものと同様な不整合が生じる
  - イテレータによる順次アクセスを行うループの途中で，対象に変更があると例外が投げられる（ConcurrentModificationException, ただしこの例外は単一スレッドでも起きる）

目次

- 基本的な排他制御
  - Single Threaded Executionパターン
  - Javaのメモリに関する処理仕様
  - 関連するJava標準ライブラリ
- 基本的な同期制御
- 読み書きを区別した制御
- タスクの実行

Java詳細：メモリに関する処理仕様

- *可視性（visibility）*
  - スレッドごとにキャッシュ（ヒープ領域）を持つため，他のスレッドが持つ変数の値について，最新の値が見えているとは限らない
  - 他スレッドにすぐに値の変更が見えるように強制するためにはその指示が必要（後述）
  - 処理系の実装が様々な最適化を行えるよう，この仕様になっている
  - 例：他スレッドが持つフラグを監視していても，フラグの古い値が見えたままという可能性はある（滅多に起きない実装になっているだろうが）

Java詳細：メモリに関する処理仕様

- 強制読み出し： キャッシュを破棄し，スレッド間での共有メモリ部分から
  - synchronizedブロックに入るとき
  - Java標準ライブラリが提供するロックを確保したとき
  - volatileと宣言された変数を読んだとき
- 強制書き出し： キャッシュの変更内容を，スレッド間の共有メモリ部分へ
  - synchronizedブロックから出たとき
  - Java標準ライブラリが提供するロックを解放したとき
  - volatileと宣言された変数に書き込んだとき

Java詳細：メモリに関する処理仕様

- 共有変数に対し，これらの強制読み出しと強制書き出しが徹底されていれば，
  - 共有変数にアクセスする際に，最新の値を読むことになる（古い値を処理に使ってしまうことはない）
    - アクセス前に最新の結果を強制読み出し
  - 共有変数のアクセス後には，自身が行った処理の結果を共有変数に反映している（新しい値をローカルにとどめておくことはない）
    - アクセス後に結果を強制書き出し
- 共有変数では，前述のいずれかの仕組みを用いるべき（synchronized/lock/volatile）

Java詳細：メモリに関する処理仕様

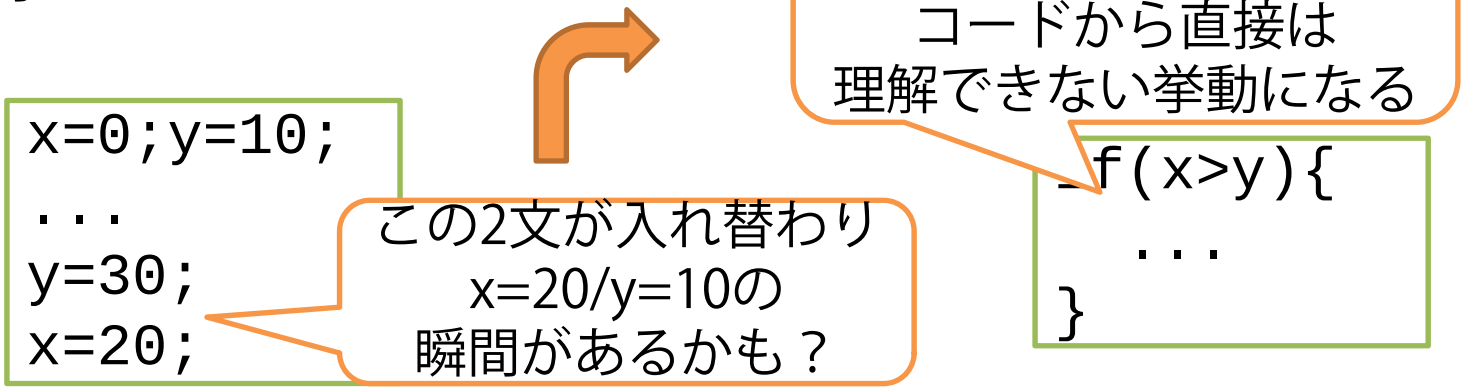
- volatile変数

  ※ volatile: 揮発性，変わりやすい
  （キャッシュを使わないということのイメージ）

  - 先の挙動により，boolean フラグ変数や，一つのスレッドだけが更新する変数（他は読むだけ）には十分（synchronizedなしで使うことができる）
  - long/doubleの単一参照・代入も原子性を保証
  - ただしインクリメントやtest-and-setなど「今の値を見て書き換える」場合は原子的ではなく使えない（書き込み側だけsynchronizedしてもよい）
  - synchronizedよりコードが単純で，オーバーヘッドは小さいことが多い（それなりにかかる）

Java詳細：メモリに関する処理仕様

- 実行順序の入れ替え（reorder）
  - 実行順序により結果が変わらない複数の処理は，順序の入れ替えや並列実行を行ってよい
  - （処理が適切に原子的になっていない）マルチスレッドでは予期せぬ状況を生むことがある
  - 処理系の実装が様々な最適化を行えるよう，この仕様になっている
  - 例：



目次

- 基本的な排他制御
  - Single Threaded Executionパターン
  - Javaのメモリに関する処理仕様
  - 関連するJava標準ライブラリ
- 基本的な同期制御
- 読み書きを区別した制御
- タスクの実行

単純な値における原子性の保証

- AtomicBoolean，AtomicInteger，AtomicIntegerArray，AtomicReference<V>など
  - get/set： volatile変数の読み書き同等の効果
  - lazySet： 書き込み順序を強制しない書き込み
  - compareAndSet / weakCompareAndSet： 現在の値が指定された値ならば書き換えることを原子的に行う（volatile同等の効果あり・なし）
  - そのほかクラスごとのメソッド（AtomicIntegerクラスであれば，incrementAndGet/addAndGetなど）

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

Strong/Weak Compare-And-Set

- 再掲：compareAndSet / weakCompareAndSet
  - 現在の値が指定された値ならば書き換えることを原子的に行う（volatile同等の効果あり・なし）？
  - 強制読み出しをして最新値を読み取り，比較対象変数へのアクセスを比較をして，等しいならば更新，強制書き出し（原子的に）

    OR
  - 値を読み出して等しく，かつ他のスレッドが値の更新を試みてなければ更新，そうでなければ失敗（何度も試してもよい）
    - ログなど実行順序を気にしなくてよいときくらい

Collectionのスレッドセーフ化（１）

- java.util.Collectionsクラスにおける synchronizedSet/List/Mapメソッド
  - 既存のSet/List/Mapをラップするスレッドセーフな Set/List/Mapを返すので，それにアクセスさせる
  - ただしイテレーターの利用時には，synchronizedする必要がある

https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

Collectionのスレッドセーフ化（２）

- java.util.concurrentパッケージでの，スレッドセーフなCollection（Set/List/Map）クラス
  - ConcurrentSkipListSet，ConcurrentLinkedDeque，ConcurrentHashMapなどのクラス（次頁）
  - すべてを一様にsynchronizedするよりも，特定の不整合や特定の仮定の下での性能向上

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

Collectionのスレッドセーフ化（３）

- CopyOnWriteArraySet/Listクラス
  - 内部的にコピーを作成して利用することにより，スレッドセーフにしている
  - イテレーターは取得時の要素を用いる（古いかもしれないが，部分的な更新で不整合があることもない）
  - 変更のオーバーヘッドが大きいが，イテレーターになどよる全体走査（多少古くてもよい）が多い場合には効率がよい

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

Collectionのスレッドセーフ化（４）

- ConcurrentSkipListSet/Mapクラスや，ConcurrentLinkedQueue/Dequeクラスなど
  - 一要素の追加や削除などはスレッドセーフ
  - 内部構造を分割し可能なら並列処理するため速い
  - sizeやaddAllなど全体にかかわる操作は，原子的でなく不整合がある結果が含まれる場合がある
  - イテレーターはある時点での状況を反映した要素を順次返し，前述の例外を投げない

    （*弱一貫性*：十分に速く更新は反映するが，反映待ちの更新があるかもしれず必ず最新とは限らない）

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

ロック

- ReentrantLockクラス
  - lock/unlockメソッド
  - ブロックしない，あるいはタイムアウト指定可能なtryLockメソッドも
  - synchronizedと異なり解放忘れに注意

http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html

```
lc.lock();
if(...){
  return ...
}
try{
  ...
  lc.unlock();
}
catch(...){
  ...
}
```

複数のreturn文での解放忘れ

例外発生時の解放忘れ（finallyを使うべき）

目次

- 基本的な排他制御
- 基本的な同期制御
  - Guarded Suspensionパターン
  - Balkingパターン
  - Producer-Consumerパターン
  - 関連するJava標準ライブラリ
- 読み書きを区別した制御
- タスクの実行

GuardedSuspention： 概要

- GuardedSuspentionパターン
  - 目的： ある条件が満たされたときだけ実行を進めるようにする（*ガード条件*が成り立つまでブロックする）
  - 防ぎたい問題： 不適切な状況で処理を実行すると不整合がある状態や実行エラーが生じる
  - 対応策： 条件が成り立つまで待つようにし，また条件を成り立たせた側はそのことを知らせるようにする（条件の成否が変えられないよう排他制御も行う）

GuardedSuspention： 実装

- Javaでの典型的な実装： wait/notifyAll

```
synchronized(this){
  while( !cond ){
    wait();
  }

  // process that requires cond;
}
```

排他制御が必要

ループが必要
（条件が成り立ち起こされた後だが，
他のスレッドが状況を変えているかも）

```
synchronized(this){
  // process that lets cond hold;

  notifyAll();
}
```

目次

- 基本的な排他制御
- 基本的な同期制御
  - Guarded Suspensionパターン
  - Balkingパターン
  - Producer-Consumerパターン
  - 関連するJava標準ライブラリ
- 読み書きを区別した制御
- タスクの実行

Balking： 概要

- Balkingパターン
  - 目的： ある条件が満たされたときだけ実行を進めるようにする（*ガード条件*が成り立っていない場合，成り立つまで待つのではなく中断する）
  - 防ぎたい問題： 不適切な状況で処理を実行すると不整合がある状態や実行エラーが生じるが，ブロックし続けると応答性が悪くなる
  - 対応策： 条件が成り立つか確認し，成り立っていなかったら中断する（条件の成否が変えられないよう排他制御も行う）

Balking： 実装

- Javaでの典型的な実装： ifでの条件確認

```
synchronized(this){
  if( !cond ){
    return false; // or wait(3000);
    // or throw exception
  }

  // process that requires cond;
}
```

タイムアウト指定も可能

戻り値が元々なければ
return falseで表現可能だが,
そうでない場合例外で区別

Balking： 概要

- 使う状況
  - 処理の必要がないことがある場合
  - 長時間ブロックせず，ユーザーへの応答など他の処理を進めたい場合
  - ガード条件が成り立つのが1回だけの場合
    （例：もし「初期化されていない」ならば）

目次

- 基本的な排他制御
- 基本的な同期制御
  - Guarded Suspensionパターン
  - Balkingパターン
  - Producer-Consumerパターン
  - 関連するJava標準ライブラリ
- 読み書きを区別した制御
- タスクの実行

Producer-Consumer： 概要

- Producer-Consumerパターン
  - 目的： ある資源についてその生産と消費をつなぎ対応させて実行する
  - 防ぎたい問題： 互いを待つようにすると生産と消費の速度が異なる場合に多くの待ち時間が生じ，また不適切な状況で生産や消費を実行すると不整合がある状態や実行エラーが生じる
  - 対応策： 中間に仲介役となるバッファーやチャンネルを設け，適切に排他制御・同期制御を行いつつ資源を管理する

Producer-Consumer： 実装

- Javaでの典型的な実装： concurrent対応のキュー（自分でwait/notifyしなくても）

```
BlockingQueue queue = new ...
```

```
// produce resource
queue.put(resource);
```

```
queue.take(resource);
// produce resource
```

ブロッキング動作は内部に入っている

目次

- 基本的な排他制御
- 基本的な同期制御
  - Guarded Suspensionパターン
  - Balkingパターン
  - Producer-Consumerパターン
  - 関連するJava標準ライブラリ
- 読み書きを区別した制御
- タスクの実行

待ち合わせの機構

- CyclicBarrierクラス
  - 指定数だけawaitが呼び出されると皆起こされ進む
  - 例： 全員が特定の実行箇所に到達するのを待つ
- CountDownLatchクラス
  - 各スレッドはこのオブジェクトに対してawaitメソッドを呼び出して待ち，初期化時に設定された回数だけcountDownメソッドが呼ばれるとawaitしている全スレッドが起こされる
  - 例： 作業者スレッドは準備ができたらcountDown, 数が揃ったら調整者スレッドは起こされ動き始める

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

様々なキュー（一部）

- ArrayBlockingQueueクラス： 固定長（putもブロックしうる，いわゆる*BoundedBuffer*）
- LinkedBlockingQueueクラス： 要素数上限なし
- DelayQueueクラス： 一定時間が経つまで取り出せず，古いものから取り出される
- SynchronousQueueクラス： put/takeが同期（ランデブー）
- ConcurrentLinkedQueueクラス： 弱一貫性に基づきブロックしない

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

セマフォ

- *セマフォ（semaphore）*：資源の個数を管理
  - Semaphoreクラス
    - acquireメソッドにて資源を1個確保（個数が1つ減る，tryAcquireの場合ブロックしない）
    - releaseメソッドにて資源を1個解放

http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html

目次

- 基本的な排他制御
- 基本的な同期制御
- 読み書きを区別した制御
  - Immutableパターン
  - Read-Write Lockパターン
- タスクの実行

Immutable： 概要

- Immutableパターン
  - 目的：複数スレッドに読み出しのみの変数を共有させたい
  - 防ぎたい問題： 共有変数の内部外部いずれかにおいて，誤って変更をしてしまう
  - 対応策： 変数の値を変更する手段を提供しない

Immutable： 実装

- Javaでの典型的な実装： 変数をfinal privateとし，getterのみを提供する

```
public final class AddressData{
  private final String name;
  private final String address;

  public AddressData(String nm, String adr){
    this.name = nm;
    this.address = adr;
  }

  public String getName(){
    return name
  };
  public String getAddress(){ ... }
```

変更不可

Immutable： 考察

- 使う状況
  - インスタンスの生成後値が変化しない場合
  - インスタンスが共有されて頻繁にアクセスされる（その際の性能を保ちたい）場合
- 注意点
  - 該当クラスにsetterがなくとも，そのフィールドが指している参照先が書き換えられることがないかに注意する必要がある

    例： getしてそれを操作すると書き換えられる

Immutableの活用

- JavaのStringやIntegerクラスなどはImmutable
  - String str1 = str1 + str2 は，文字列の値を書き換えているわけではなく，新しく文字列オブジェクトを作り，それを指すように参照を書き換えている
  - MutableなStringBufferクラスとの使い分け
- CollectionsクラスのunmodifiableSet/List/Mapメソッド
  - Set/List/Mapの変更不可能なビューを返す
  - 返されたものに対し，変更メソッドを呼び出しても実行できない（UnsupportedOperationException ）

https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

目次

- 基本的な排他制御
- 基本的な同期制御
- 読み書きを区別した制御
  - Immutableパターン
  - Read-Write Lockパターン
- タスクの実行

Read-Write Lock： 概要

- Read-Write Lockパターン
  - 目的：共有変数に対し読み出しと書き出しのタスクが分かれている
  - 防ぎたい問題：同一データへの複数スレッドの干渉により共有資源（変数）の値が予期せぬものとなることを避けたいが，読み書き一様に排他制御するとオーバーヘッドが大きい
  - 対応策： 複数の読み出しは同時に行えるようにし，読み出しと書き出し，複数の書き出しの間のみで排他制御を行う
  - 使う状況
    - 読みの方が頻度が高い・処理が大きい場合

Read-Write Lock： 実装

- Javaでの典型的な実装： ReentrantReadWriteLockを用いる

```
final ReentrantReadWriteLock rwl
  = new ReentrantReadWriteLock();

rwl.readLock().lock();
// read
rwl.readLock().unlock();

rwl.writeLock().lock();
// write
rwl.writeLock().unlock();
```

補足： Read-Write Lockの自前実装例

```
public synchronized void readLock() ... {
  while(writing > 0 ||
        (writepriority && waitingwriter > 0){
    wait();
  }
  reading++;
}

public synchronized void readUnlock(){
  reading--;
  writepriority = true;
  notifyAll();
}
```

（続く）

補足： Read-Write Lockの自前実装例

```
public synchronized void writeLock() ... {
  waitingWriter++;
  try{
    while(reading > 0 || wrting > 0){
      wait();
    }
  }
  finally{ waitingWriter--; }
  writing++;
}

public synchronized void writeUnlock(){
  writing--;
  writepriority = false;
  notifyAll();
}
```

目次

- 基本的な排他制御
- 基本的な同期制御
- 読み書きを区別した制御
- タスクの実行
  - Thread-Per-Messageパターン
  - Worker Threadパターン
  - Futureパターン
  - その他のパターン

Thread-Per-Message： 概要

- Thread-Per-Messageパターン
  - 目的： 個々のリクエストに随時応答しつつ，それぞれに対応する処理を進めたい
  - 防ぎたい問題： 1つのリクエストに対する処理が終わるまでリクエストに対して応答することができない
  - 対応策： リクエストに対する実際の処理を新しいスレッドに委ねる
  - 使う状況
    - 処理に時間がかかる
    - 処理の順序は気にしない
    - 処理の戻り値は不要である

Thread-Per-Message： 実装

- Javaでの典型的な実装： スレッド立ち上げ
  （下記では実際の処理が書かれたhelperオブジェクトと，匿名Threadクラスを利用）

```
public void request(...){
  new Thread(){
    public void run(){
      helper.work(...);
    }
  }.start();
}
```

ThreadFactory

- ThreadFactoryインターフェース
  - newThreadメソッドを定義している
  - スレッドの生成方法をここで定義しておき，都度呼び出すと便利

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadFactory.html

目次

- 基本的な排他制御
- 基本的な同期制御
- 読み書きを区別した制御
- タスクの実行
  - Thread-Per-Messageパターン
  - Worker Threadパターン
  - Futureパターン
  - その他のパターン

Worker Thread： 概要

- Worker Threadパターン
  - 目的： 個々のリクエストに随時応答しつつ，それぞれに対応する処理を進めたい
  - 防ぎたい問題： Thread-Per-Messageでは，スレッド生成のオーバーヘッドが生じ，リクエストが多い場合にメモリ等が多数使われる
  - 対応策： 予め作業者スレッド群（*スレッドプール*）を作り，それが順次リクエストに対する処理を実行する
    （呼び出しと実行が分離され，作業者の数を制御することができる）

Worker Thread： 実装

- Javaでの典型的な実装： 固定数のスレッド立ち上げと，その中での繰り返し処理

```
WorkerThread[] threadPool;
...
for(WorkerThread th: threadPool){
  th.start();
}
```

```
public class WorkerThread ... {
  public void run(){
    Request req = queue.take();
    req.execute();
  }
}
```

ThreadPoolExecutor

- ThreadPoolExecutorクラス
  - プールの大きさやスレッドの作り方（予めか必要に応じ），使われないスレッドの終了時間などを設定
  - executeメソッドにて実行指示（すぐあるいはそのうちスレッドが割り当てられて実行される）
- Executorsクラスで十分なものが作成可能
  - newFixedThreadPoolメソッド： 固定数
  - newCachedThreadPoolメソッド： 必要に応じて作られ，一定時間使われないと消える
  - newScheduledThreadPool： 定期実行など

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html

目次

- 基本的な排他制御
- 基本的な同期制御
- 読み書きを区別した制御
- タスクの実行
  - Thread-Per-Messageパターン
  - Worker Threadパターン
  - Futureパターン
  - その他のパターン

Future： 概要

- Futureパターン
  - 目的：個々のリクエストに随時応答しつつ，それぞれに対応する処理を進め，その結果を得たい
  - 防ぎたい問題： 結果が戻り値として得られるまで待つと応答性が悪くなる
  - 対応策： 戻り値が入る先となる*Future*オブジェクトを返し，後の任意のタイミングで戻り値を取り出せるようにする

http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html

Future： 実装

- Javaでの典型的な実装： FutureTaskクラス（Future インターフェースの実装の1つ）を用いる

```
Callable task = new ...;

FutureTask<String> future
  = new FutureTask(task);

// do other things

String result = future.get();
```

ブロックしない

ブロックする

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html

Future

- Callableインターフェース
  - Runnableと同様に任意の処理をcallメソッドとして持つクラスを定めるが，runと異なり戻り値がある
- Futureインターフェース
  - この参照自体はすぐに返される
  - getメソッドで結果を得る際にブロッキングする
  - isDoneメソッドで終了確認
  - cancelメソッドでキャンセル
- FutureTaskクラス
  - Futureインターフェースの基本的な実装

目次

- 基本的な排他制御
- 基本的な同期制御
- 読み書きを区別した制御
- タスクの実行
  - Thread-Per-Messageパターン
  - Worker Threadパターン
  - Futureパターン
  - その他のパターン

その他のパターン

- Two-Phase Terminationパターン
  - スレッドを停止したい場合，外部からは強制停止できない（正確には昔はできたが非推奨）
  - 停止する旨を伝え，そのスレッド自身がそのことを検知し，終了処理を正しく終えてから終了する

その他のパターン

- Thread-Specific Storageパターン
  - マルチスレッド対応でなかったプログラムにおける変数をスレッドごとにするため，スレッド名ごとに保存先を変えるような変数に置き換える
- Active Objectパターン
  - 複数クライアント（マルチスレッド）対応でなかったプログラムに対し，リクエストをバッファし1つずつ呼び出すようなスケジューラーを間にはさむことによりマルチスレッド対応とする

まとめ

- Javaにおけるマルチスレッドプログラミングパターン
  - これまで扱ったものと概念は同等で，Javaに限るものでもない
  - ただしJava固有の記法や，処理仕様を考慮する必要がある
  - 多くの典型的な処理は，標準ライブラリを使いこなすだけで実現することができる
    （ただし理屈を知らないと正しく使いこなせない）

- 次回： 関連する最新の話題