

0.1. Android アプリケーションとマルチスレッド

Android アプリケーションの中で、ユーザーからのキー入力と、それに応答した画面の描画(動きのある画面)を伴うようなものを作成する場合には、今まで学んだ基礎的なウィジェットの知識に加えて Java の Thread クラスを利用した仕組みが必要不可欠になります。この理由を簡単に記すと以下のようになります。

今まで記してきた基礎的なアプリケーションは、スレッドという実行単位の中で動作していたのですが、1 つしかないスレッドを利用してリアルタイムに動きのある画面などを描画しつづけると、描画処理にアプリケーションが集中してしまい、ユーザーからのキー入力などを、スムーズにアプリケーションに取り込むことができなくなってしまいます。

かといって、ユーザーからのキー入力等の監視に集中したスレッドの中から、定期的な画面の描画を行おうとすると、今度はスムーズな画面の描画が実現できなくなってしまいます。

特に、GAME アプリなどで、リアルタイムに画面表示が変化していくようなアプリケーションを作る際には、こうした点を解決する必要があります。

そこで、プログラムのメインスレッドとは別に、新たなスレッドを生成して、定期的な画面の描画などは、新たに生成したスレッド側に任せるという考え方が生まれます。

実は、これは Android に限った話題ではなくて、一般の Java のアプリケーションを作成する場合などにも共通する考え方になっています。

さて、スレッドの重要性はざっと以上ようになりますが、Thread クラスは Android に固有のクラスではなく、あくまで Java 言語の API にそもそも存在する汎用のクラスです。

1.1. スレッドとは

Windows や Unix などと同時に複数のプログラムを実行させることが出来ます。これは実行の単位である「プロセス」を OS が管理しており、OS が「プロセス」を制御し、同時に複数のプロセスを実行可能な「マルチ・プロセス」の仕組みを提供しています。

□ スレッド

通常のプログラムは開始→処理→終了の流れを持つ逐次制御フローです。

スレッドとは、一言で言えば逐次的な制御フローのことです。

スレッド (Thread) はもともと「糸」という意味ですが、糸をたぐるように順番に命令が実行されていくため、このような名前が付いています。

参考 — OS から見れば Java は一つのプロセスという単位で管理されています。

Java のスレッドは OS 上の JVM で動く仮想的な「プロセス」と考えると分かりやすい。

スレッドとは上記のことからプログラムの処理単位です。Java でのスレッドは **main** メソッドを実行している処理も一つのスレッドであり、そのスレッドのことをメインスレッドと呼びます。従い、メインメソッドが終了するとプログラム (Java プロセス) も終了します。

参考 — メインスレッドは JVM によって自動的に起動されるため、起動するコードを書く必要はありません。今まで例題、演習で作成してきたプログラムは全てメインメソッドにて動作している、シングルスレッドです。

マルチスレッド

マルチスレッドとは名前のとおり複数のスレッドと言う意味です。

マルチスレッドはメインスレッドとは別に処理単位（スレッド）を複数生成して、平行して複数の処理を行うことを言います。

例えば、通信処理を行いながら同時に計算処理などの別の処理を行うことが出来ます。

参考 — `java.awt`（GUI を構築するためのパッケージ）や、`Servlet`（サーバー側のシステムを構築する際のテクノロジー）等はいくつかのスレッドが自動で起動されています。

Java ではこの様なマルチスレッドが言語仕様としてサポートされています。つまり、一つの Java プログラムが複数のスレッドを持つことが可能であり、そのプロセス内のスレッドに対する資源管理を JVM が実行してくれます。

※他の言語でも、マルチスレッドを使うことは可能です。ただ、資源割り当てなどを自分で実装する必要があり、非常に高度なスキルが求められます。また、ローカルコンピュータに依存するコードが必要になるため、移植時の変更コストも大です。

参考 — 多くのコンピュータは CPU を 1 つしか持っていません。そのようなコンピュータでは、厳密には 1 つの処理しか同時には実行することができません。従い、「マルチスレッド」は、通常「時分割処理」という方法が用いられています。複数の処理を頻繁に切り替えて実行することで、仮想的に複数の処理を同時に実行しているように見せかけています。

※Java はさらに OS から見れば一つのプロセスにすぎないことも忘れてはいけません。

図 7-1 スレッドの状態イメージ



1.2.スレッドの実行

Java でマルチスレッドプログラムを作成するのは非常に簡単です。この項では2つのスレッドを作成する方法を説明します。

1.2.1.Threadクラスを継承する方法

Thread クラスを継承したクラスを利用して新しいスレッドを実行する方法です。
処理の流れは以下です。

- ① Thread クラスを継承するクラスを作成する。
- ② 継承したクラスで `run()` メソッドをオーバーライドする。
- ③ 継承したクラスのインスタンスを作成する。
- ④ 作成されたインスタンスの `start()` メソッド (Thread) を呼び出す。

一般的にスレッドは、Thread クラスかそのサブクラスのオブジェクトとして生成され、実行されます。実際に実行される処理は `run()` メソッドであり、自分で作成する場合は、`run()` メソッドをオーバーライドして、目的の処理を記述することになります。

※後述するもう一つのやり方も基本的には変わりません。

注 - `run()` メソッドを直接呼び出してはいけません。

`start()` メソッドを呼び出すことで、適切に `run()` メソッドが呼び出されます。

□ コーディング例

①、②

```
class TestThread extends Thread {  
    public void run() {  
        // このスレッドの処理を記述  
    }  
}
```

③、④

```
TestThread tThread = new TestThread();    //スレッドクラスのインスタンス化  
tThread.start();                          //スレッド実行要求
```

例題 7-1

- スレッドクラスを2つ生成し、それぞれを実行し結果を確認する。
スレッドクラスは単純に10回分のループ処理の状況を標準出力するだけ。

```
package thread;

//同時並行で動く処理クラス
class NewThread extends Thread {

    //コンストラクタ
    public NewThread(String name) {
        super(name);
    }

    //オーバーライドして処理を記述
    public void run() {
        for(int i = 0; i < 500; i++) {
            System.out.println
                ("○スレッド名称=" + this.getName() + " 処理カウント数 = " + i);
        }
    }
}

//-----

public class SampleThread {

    public static void main(String[] args) {
        //並行処理スタート
        NewThread nThread = new NewThread("NEW");
        nThread.start();

        // メインスレッド固有の処理
        for(int i = 0; i < 500; i++) {
            System.out.println
                ("●スレッド名称=MAIN" + " 処理カウント数 = " + i);
        }
    }
}

} //終わり
```

交互に呼び出されいかにも並行に実行されている動きです。

1.2.2. Runnable インタフェースを実装する方法

Java は一つのスーパークラスしか継承することができません。従ってあるクラスを継承しているサブクラスをスレッドとして実行する場合は、**Thread** クラスを継承して作成することは出来ません。従い、Runnable を実装して実現する方法も用意されています。

Runnable インタフェースを実装したクラスを利用して新しいスレッドを実行する処理の流れは以下です。

- ① Runnable インタフェースを実装したクラスを作成する。
- ② 実装したクラスで `run()` メソッドをオーバーライドする。
- ③ 実装したクラスのインスタンスを作成する。
- ④ そのインスタンスを引数として、Thread クラスのコンストラクタを呼び、Thread クラスのインスタンスを作成する。
- ⑤ 作成された Thread クラスのインスタンスの `start()` メソッドを呼び出す。

Thread クラスを継承する方法より多少手順は増えますが、特に難しくはありません。

実装クラスをインスタンス化して作ったオブジェクトの参照を、Thread クラスのコンストラクタの引数にして Thread クラスを生成します。実際に実行される処理は `run()` メソッドであり、`run()` メソッドを実装する方法は Thread クラスを継承する方法と変わりありません。

□ コーディング例

①、②

```
class TestRunnable implements Runnable {  
    public void run() {  
        // このスレッドの処理を記述  
    }  
}
```

③、④、⑤

```
TestRunnable tRunnable = new TestRunnable();    //スレッド対象クラスのインスタンス化  
Thread thread = new Thread(tRunnable);        //スレッドクラスのインスタンス化  
thread.start();                                //スレッド実行要求 (TestRunnable クラスの run メソッドが実行)
```

例題 7-2

- 例題 7-1 と同様の処理を Runnable インタフェースを実装した方法で行う。

```
package thread;

//同時並行で動く処理クラス
class NewRunnable implements Runnable {

    // オーバーライドして処理を記述
    public void run() {
        for (int i = 0; i < 500; i++) {
            System.out.println("○スレッド名称="
                + this.getClass().getName() + " 処理カウント数 = " + i);
        }
    }
}

public class SampleRunnableThread {

    /**
     * @param args
     */
    public static void main(String[] args) {

        // NewRunnable クラスを生成
        NewRunnable nRunnable = new NewRunnable();
        // Thread クラスを生成
        Thread thread = new Thread(nRunnable);
        thread.start();

        // メインスレッド固有の処理
        for(int i = 0; i < 500; i++) {
            System.out.println
                ("●スレッド名称=MAIN" + " 処理カウント数 = " + i);
        }

    }
}

//終わり
```

例題 7-1 とスレッドの実行順が異なります。交互に実行されているのではなく、スレッドの処理順序は都度変わります。

1.2.3.2つのスレッド方法の比較

紹介した2つの方法をどの様に使い分けたら良いのでしょうか？

Java2 SDK の API 仕様書には「クラス Thread のメソッドのうち、メソッド run だけをオーバーライドして使用する場合は、インタフェース Runnable を使用してください。Thread クラスを継承する方法は、run 以外のメソッドもオーバーライドしてスレッドの基本的な動作を変えたい場合にのみ使用する事が推奨されています。」という意味のことが書かれています。

また幾つかの書籍ではいろいろな見解が述べられています。

Runnable の使用の項で述べた様に、既にスーパークラスを継承しているクラスをスレッド処理として実装するには、Runnable インタフェースを実装する方法しかありません。が、それ以外の場合はどの様に考えれば良いのでしょうか？

□ Runnable インタフェースを実装する方法を最優先に検討する。

「オブジェクト指向」という観点から見た場合、Thread を継承する方法はあまり良い方法とは言えません。サブクラスと言うのは本来どういう目的で作成されるかと言うことを考えるべきです。Thread ジョブとして作成するクラスは、Thread 処理とは関係無いはずで

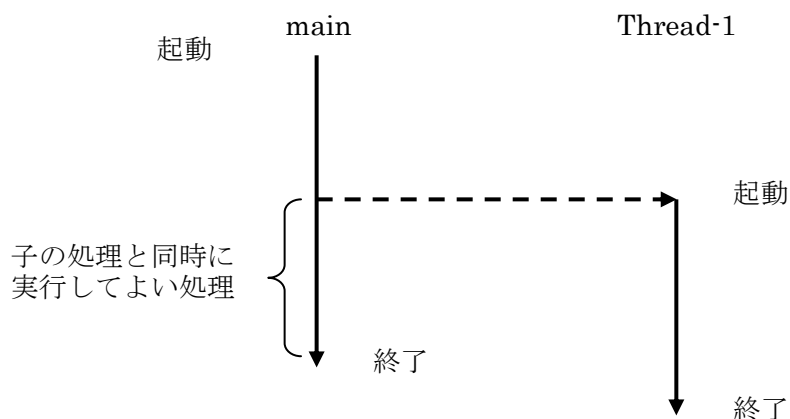
スーパークラスとサブクラスがあまりにもかけ離れたものになってしまうため、やはり Runnable インタフェースを実装する方法が望ましいはずで

1.2.4.スレッドのライフサイクル

Java のスレッドは、作成されてから消滅するまでいくつかの状態を遷移します。

□ メインスレッドと子スレッドの関係

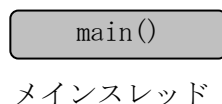
図 7-2 メインスレッドと子スレッドの関係



スタックされるイメージ

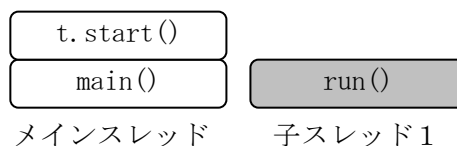
- ① JVM が main() メソッドを呼び出す。

```
public static void main(String[] args) {  
    :  
}
```

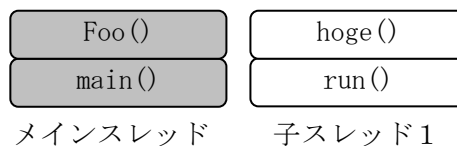


- ② main() メソッドが新たなスレッドを起動する。

```
Runnable r = new MyThreadJob();  
new Thread(r).start();  
Foo f = new Foo();
```



- ③ JVM はメインスレッドと子スレッド 1 の処理を、両者の処理が終了するまで交互に進める。



□ スレッドスケジューラ

スケジューラは「実行可能」状態から「実行中」状態に、また逆に「実行中」から「実行可能」状態にスレッドを制御します。このような状態の制御はすべて **Java** のスレッドスケジューラが行います。従い、プログラマがスケジューラをコントロールすることは出来ませんし、そのためのメソッドも用意されていません。

※スケジューラを直接コントロールすることは出来ませんが、間接的にコントロールすることは可能です。(後述します)

注 — スレッドのスケジューラはこう動くという予測のもとにプログラムを組んではいけません。

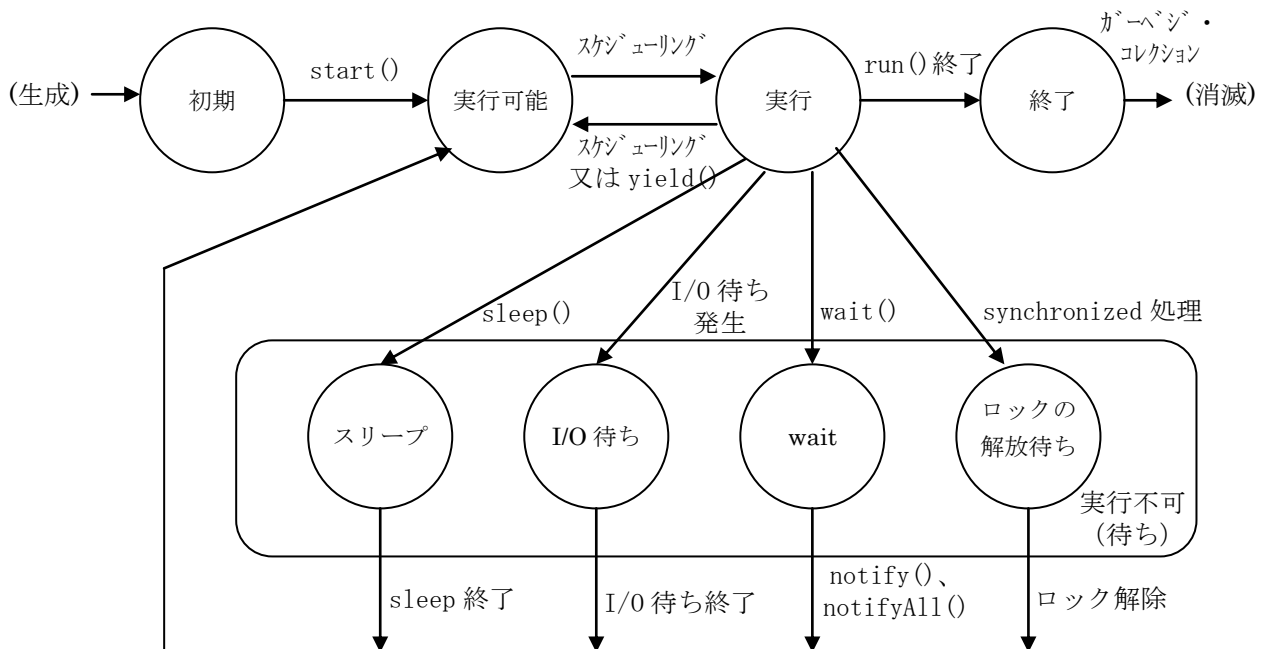
Java のスレッドのスケジューリングは、プライオリティ（実行優先度）に基づいて行われるように規定されています。各スレッドはプライオリティを持ち、通常プライオリティの高いスレッドがプライオリティの低いスレッドより優先して実行されます。

参考 — 子スレッドのプライオリティの初期値は、親スレッドのプライオリティと同じ値になります。

クラス `Thread` には、プライオリティを操作するメソッドとして、`getPriority`、`setPriority` が用意されています。

□ スレッドのライフサイクル

図 7-3 スレッドの状態遷移



- 初期
クラス Thread（又はそれを継承したクラス）のオブジェクトが生成された (new) 状態。
- 実行可能
メソッド start が実行されることにより、スレッドは「実行可能」状態になります。
※スレッドオブジェクトに資源が割り当てられます。
- 実行
スケジューリング機構によって、「実行可能」状態にあるスレッドのうち 1 つが選ばれ、実行されます。 ※run メソッドが実行されます。
 - ・ より高いプライオリティのスレッドが「実行可能」状態になった時や、そのスレッドに割り当てられている実行時間が終了した場合は「実行中」から「実行可能」状態に戻ります。
 - ・ メソッド yield が呼び出された場合も「実行中」から「実行可能」状態に戻ります。
※プライオリティ値に依存します。

○ 実行不可（待ち）

「実行中」のスレッドは以下のような契機で「実行不可」（待ち）状態になります。

- ・メソッド `sleep`
- ・周辺機器などの I/O 待ちの発生
- ・メソッド `wait` の実行
- ・他のスレッドにロックされているオブジェクトに対する `synchronized` 処理の実行

○ 終了

メソッド `run` の処理が終了するとスレッドも「終了」状態になります。

□ Thread クラスのメソッド

Thread クラスにはライフサイクルを明示的に操作するメソッドがあります。

以下代表的なメソッドを紹介します。

① `sleep` メソッド

`sleep` メソッドは Thread クラスのクラス（static）メソッドです。

`sleep` メソッドは指定時間だけ現在実行中のスレッドを休止させます。`sleep` メソッドはクラスメソッドのため、`sleep` メソッドを実行するスレッド以外の別のスレッドを休止することはありません。

- ・ `public static void sleep(long millis) throws InterruptedException`

使用例

- ・ `thread.sleep(1000);` // 1 秒停止後復帰

注 — `sleep` メソッドを呼び出したスレッドは、取得しているロックを解放しないことに注意してください。

参考 — `sleep` メソッドは、休止中に他のメソッドから割り込みがかけられた時に `InterruptedException` が発生しますので例外処理を記述する必要があります。

② yield メソッド

yield メソッドは、現在処理中のスレッドを一時休止し、他のスレッドに実行の機会をあたえます。即ち自信は「実行可能」状態に戻ります。yeild メソッドも sleep メソッドと同じくクラス (static) メソッドです。休止されるスレッドを指定することは出来ません。

- `public static void yeild()`

使用例

- `thread.yeild();` //実行可能状態に遷移し他のスレッドに実行権を譲ります。

参考 - yield() は sleep(0) と似たような効果と理解すると分かりやすいです。

③ interrupt メソッド

interrupt メソッドは休止中のスレッドに割り込みを入れるメソッドです。

割り込みをいれることの出来るスレッドは、join メソッドや sleep メソッドの実行により待機中のメソッド (Object の wait メソッドも対象) です。

- `public void interrupt()`

④ getPriority/setPriority メソッド

全てのスレッドはプライオリティ (優先度) を持っています。

このスレッド毎のプリオリティ値を取得したり、変更したりする事が出来ます。

- `public final void setPriority(int newPriority)`
- `public final int getPriority()`

優先度が指定出来る値は 1 ~ 10 です。数値が大きい方が優先度が高いことになります。

また、スレッド同期を行うメソッドも存在します。

マルチスレッドでは、別のスレッドにある処理を任せ、そのスレッドが終了した時に、自分のスレッドの処理を再開したいという場合があります。このような時、Thread クラスに以下のメソッドが用意されています。

① join メソッド

このメソッドはスレッドに対して呼び出すと、当該スレッドが終了するまで呼び出し元のスレッドが待機します。join メソッドには、引数の異なる 3 つのメソッドがあります。

- `public final void join() throws InterruptedException`
- `public final void join(long millis) throws InterruptedException`
- `public final void join(long millis, int nanos) throws InterruptedException`

※引数なしは該当スレッドが終了するのを永久に待ち続けます。引数のあるものは、終了しなくても、指定時間経過すれば処理を再開します。(join() と join(0) は同じ)

例えば、例題 7-1 を生成したスレッドを永久に待つ様にする場合は

```
//スレッド実行要求
System.out.println("スレッド実行要求開始");
sThread1.start();
sThread2.start();
System.out.println("スレッド実行要求終了。run メソッドが呼ばれる。");

この部分を
System.out.println("スレッド実行要求開始");
sThread1.start();
sThread2.start();
try {
    sThread1.join();
    sThread2.join();
} catch (InterruptedException ie) {
}
System.out.println("スレッド実行要求終了。run メソッドが呼ばれる。");
```

とすることで、sThread1 と sThread2 の処理が終了するまで待つことになります。

さらに推奨されていないメソッドは以下の通りであり、基本的にはスレッドを停止させるようなメソッドです。

① stop メソッド

スレッドを停止させるためのメソッドです。

② suspend メソッド

スレッドを直ちに中断させるメソッドです。resume メソッドによって再開します。

③ resume メソッド

suspend メソッドによって中断されたスレッドを再開するメソッドです。

演習 7-1

例題 7-2 のプログラムを使用して、スレッド名称が”スレッド 2”のスレッドのみ run メソッドの先頭で 2 秒間休止させ、必ずスレッド 1 が先に終了させるようにしなさい。

演習 7-2

run メソッドの処理を永久ループさせる処理 (while(true)) を実装し、stop メソッドを使用せずに外部から対象スレッドを終了させる処理を実装しなさい。

1.3.スレッドの排他制御

いままでのスレッドは、それぞれ別々のオブジェクトを作成しマルチスレッドで起動していました。それでは、複数のスレッドが同じオブジェクトを操作する場合はどうなるのでしょうか。

結論から言うと、プログラムは予想外の動作をします。では、実際に動かして問題になることを確認してみたいと思います。

例題 7-3

- ・ 口座情報を管理するクラス : Kouza
- ・ 振込み処理をスレッドで実装するクラス : FurikomiShori
- ・ 同一の口座に対し、2 人が同時に振込みを行う動作クラス : DoujiFurikomiMain

```
package thread;
public class Kouza{
    int zandaka;//残高

    //残高を設定するコンストラクタ
    public Kouza(int zandaka){
        this.zandaka = zandaka;
    }

    //入金する(ここが排他制御を必要とする箇所!)
    public void nyukin(int kingaku){//synchronized

        //ここは、わざと長い時間がかかる処理にしています!
        int tmp = zandaka + kingaku;
        try{
            Thread.sleep(500);//休止
        }catch(InterruptedException e){}
        zandaka = tmp;//残高に入金額を加算する
        //入金時の現残高を表示する

        System.out.println("現在の残高:" + this.zandaka);
    }//nyukin()の終わり
}//終わり
```

残高を保持する口座情報クラスであり、残高に入金加算するメソッドが存在します。

```

package thread;

//振込み処理（複数のA T Mから同時入金される可能性があるのでマルチスレッドにする）
public class FurikomiShori extends Thread {

    //振込み対象の銀行口座
    //(同一口座が、数人の振込み処理から同時刻に振込みが行われる可能性あり)
    private Kouza kouza;

    public FurikomiShori(Kouza kouza){
        this.kouza = kouza;
    }

    //同時並行処理の対象となる部分(注！---意図的に時間のかかる処理にしている)
    public void run(){
        //計10万円を10回に分けて入金
        for(int i = 1; i <= 10;i++){
            System.out.println(getName() + "入金中");
            kouza.nyukin(10000); //入金メソッドを呼び出す
        }
    }
}

```

口座情報を確認し、一定の金額 100.000 円を入金するクラスであり、口座情報をインスタンス変数として保持しています。

このクラスに同一口座情報を渡し、2つの入金処理スレッドを生成し処理を実行してみます。

```

package thread;

//同時振込みデモ
public class DoujiFurikomiMain{

    public static void main(String[] args){

        //口座を生成
        Kouza kouzaOoyasan = new Kouza(1000000); //残高 100 万円の大家さんの口座

        //0さんの振込み処理スレッドを生成(振込み先は大家さんの口座)
        FurikomiShori frikomi0 = new FurikomiShori(kouzaOoyasan);
        //1さんの振込み処理スレッドを生成(振込み先は大家さんの口座)
        FurikomiShori frikomil = new FurikomiShori(kouzaOoyasan);

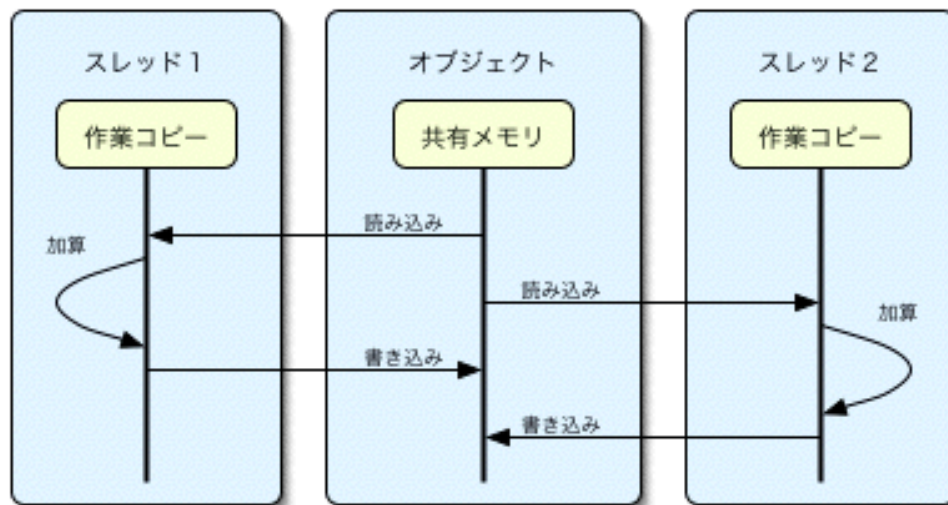
        //同時振込み開始
        frikomi0.start(); //0さんが10万円振り込み
        frikomil.start(); //1さんが10万円振り込み

        //振込み終了後、大家さんの口座残高合計が1200000円になるはず。。。

    }
} //終わり

```


では実際に実行してみましょう。



1.4. スレッドの同期(待合い)

共有のデータを複数のスレッドが扱うと、データの競合が発生する可能性があります。それは以下の処理で、あるスレッドが処理を開始し (2) の状態にあるときに、別のスレッドが (1) の処理を開始するような場合です。これはまさに前述のスレッドセーフでない状況です。

- (1) 共有域のデータを取り込む
- (2) 取り込んだデータを加工する
- (3) 加工したデータを共有域に書き込む

1.4.1. ロック

このようなことが起こらないようにするには、あるスレッドが上記の (1) ~ (3) の処理をしている間は、他のスレッドはこの処理ができないようにロックすれば良いのです。このように排他的な操作によって複数のスレッド間で矛盾が起こらないようにすることを、同期をかける (synchronize) と言います。

1.4.2. synchronized キーワード

同期をかけるには、次のようにメソッドに `synchronized` キーワードを付けます。するとこのメソッドを含むインスタンスに同期がかかり、他のスレッドはそのインスタンスの `synchronized` メソッドは実行できなくなります (実行が終わるまで待機状態で待たされます)。そのインスタンスの `synchronized` でないメソッドは自由に実行できます。また、同じクラスから生成されたものでも、別インスタンスのメソッドは、実行できます。

```
synchronized void methodA() { ...           // メソッド単位で設定
```

メソッド全体でなく、メソッドの一部分だけで同期をかけることができます。それには次のように `synchronized` ブロックを使います。この場合同期をかけるオブジェクトを指定します。

```
synchronized (obj) { ... } // ブロック単位で設定
```

同期をかけると、他の処理をロックすることになるので、一般に性能の低下をもたらします。ロックは必要最小限にしましょう。