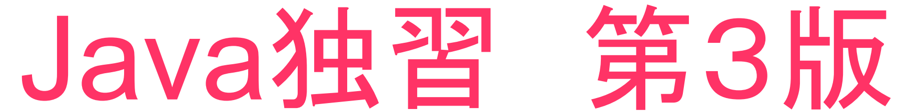
Java独習　第3版



9.3 同期
9.4 デッドロック
9.5 スレッドの通信

2006年6月21日(水)　　南　慶典

9.3 同期

共有データへの読み書きの同期

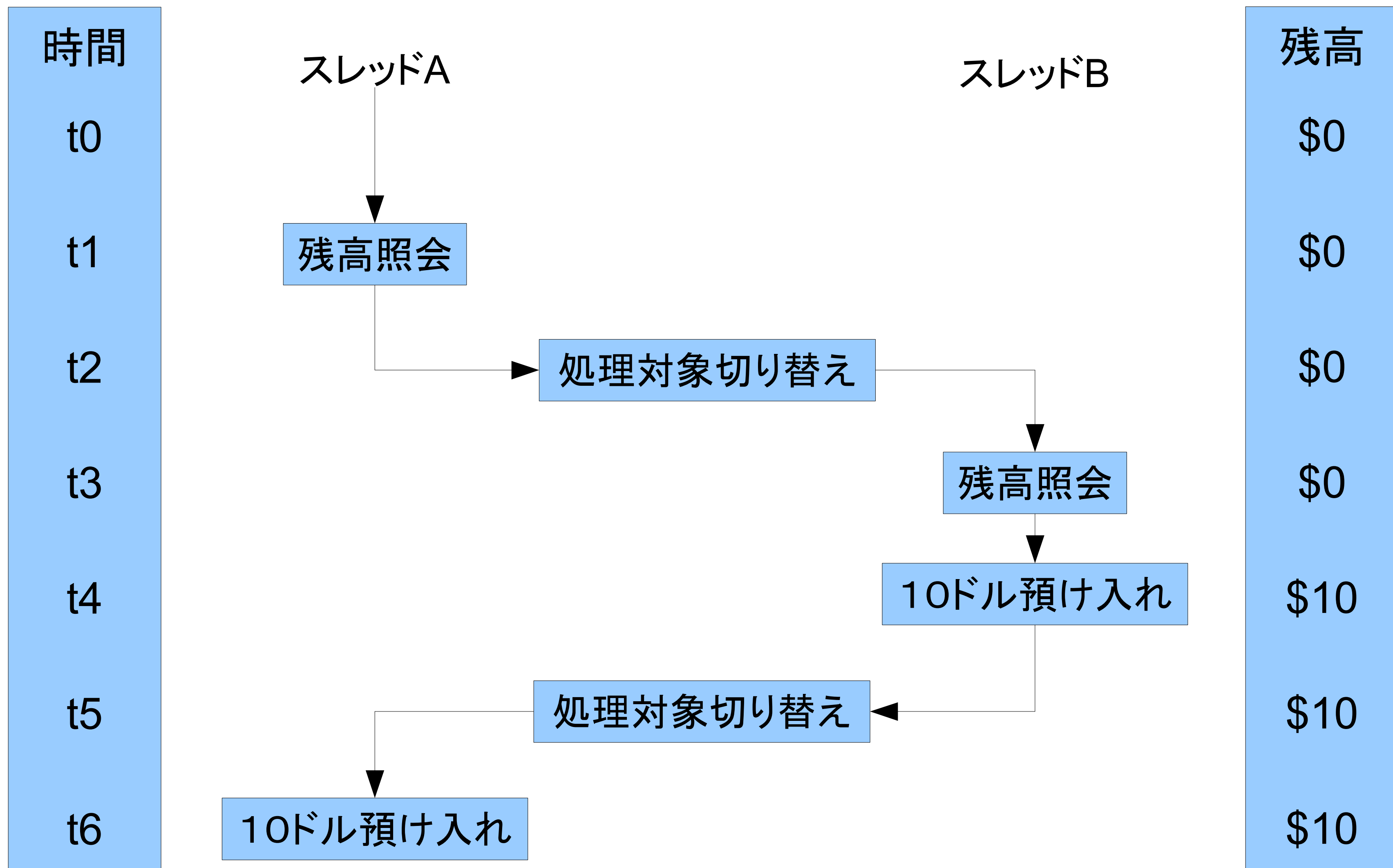
複数のスレッドから共有データを読み書きするときに発生する問題について

一つのフィールドに対して複数のスレッドが同時にアクセスする可能性がある場合、その順番によっては整合性が保てなくなる可能性があるので、スレッドの制御フローが独立していては困ることがある。

次のスライドに一例として、複数の顧客が共有する銀行口座があると仮定した図を用意。顧客はみな、この口座に現金を預け入れしたり引き出したりすることができる。アプリケーションでは、顧客ごとにスレッドを用意して入金や出勤を処理することになる。

t0時に口座の残高は0に初期設定される。スレッドAが実行され、口座に10ドルを預け入れようとする。t1時の残高は0ドルである。ただし、t2時にはスレッドAからスレッドBに処理対象が切り替えられる。t3時にはスレッドBが残高を読み取る。スレッドBは、t4時に口座に10ドルを預け入れる。t5時には処理対象が切り替えられ、スレッドに制御が戻される。t6時に、スレッドAは残高を10ドルに設定する。

この一連の操作の結果、最終的な口座の残高は10ドルにしかならない。



共有データへの読み書きの同期

この問題を解決するには、共有データへの読み書きを同期させる必要がある。

共有データへの読み書きを同期させる2つの方法

1．　メソッドの宣言にsynchronized修飾子を指定して、メソッドを同期させる
2．　synchronizedブロックの使用

スレッドによって実行されたsynchronizedインスタンスメソッドでは、まずそのオブジェクトのロックが自動的に取得される。別のスレッドが同じオブジェクトのsynchronizedインスタンスメソッドを実行しようとすると、JVMは、現在のスレッドがロックを解放するまで2番目のスレッドを待たせる。

スレッドによって実行が開始されたsynchronized静的メソッドでは、関連付けられたClassオブジェクトのロックが自動的に取得される。別のスレッドが同じクラスのsynchronized静的メソッドを実行しようとすると、JVMは、現在のスレッドがロックを解放するまで2番目のスレッドを待たせる。

ロックはどちらもメソッドの処理が完了すると自動的に解放。また実行できるのは一度に1つのスレッドだけとなる。

共有データへの読み書きの同期

synchronizedブロックの使用

```
synchronized ( obj ) {
    //  処理ブロック
}
```

obj はロック対象のオブジェクトである。インスタンスデータを保護する必要がある場合は、そのオブジェクトをロックする。クラスデータを保護する場合は、対応するClassオブジェクトをロックする。

例として、複数の顧客が共有口座に現金を預け入れられる操作をシミュレートしたプログラムを次のスライドに載せる。

```java
class Account {
	private int balance = 0;

	synchronized void deposit ( int amount ) {
		balance += amount;
	}
	int getBalance() {
		return balance;
	}
}

class Customer extends Thread {
	Account account;

	Customer ( Account account )  {
		this.account = account;
	}

	public void run() {
		try {
			for ( int i=0; i<100000; i++ ) {
				account.deposit(10);
			}
		}
		catch ( Exception e ) {
			e.printStackTrace();
		}
	}
}
```

```java
class BankDemo {
	private final static int NUMCUSTOMERS = 10;
	public static void main(String[] args) {
		//  口座の作成
		Account account = new Account();

		//  スレッドの作成と起動
Customer customers[] = new Customer[NUMCUSTOMERS];
		for (int i=0; i < NUMCUSTOMERS; i++){
			customers[i] = new Customer(account);
			customers[i].start();
		}

		//  スレッドの完了を待機する
		for (int i=0; i< NUMCUSTOMERS;i++){
			try{
				customers[i].join();
			}
			catch(InterruptedException e){
				e.printStackTrace();
			}
		}
	// 口座の残高を表示する
	System.out.println(account.getBalance());
	}
}
```

実行結果
100000

9.4 デッドロック

デッドロック

デッドロックとは、マルチスレッドプログラムで発生するエラーの一種で、複数のスレッドが互いにロックの解放を永久に待ち続けている状態のことである。

　例えば、スレッド１ではオブジェクト１のロックを保持し、オブジェクト2のロック解放を待機するとする。その一方で、スレッド2では、オブジェクト2のロックを保持し、オブジェクト1のロック解放を待機している。この場合、どちらのスレッドも処理を進めることができず、必要なロックがもう一方のスレッドから解放されるのを、永遠に待ち続ける。

```java
class A {
    B b;

    synchronized void a1(){
        System.out.println("Starting a1");
        b.b2();
    }

    synchronized void a2(){
        System.out.println("Starting a2");
    }
}

class B {
    A a;

    synchronized void b1(){
        System.out.println("Starting b1");
        a.a2();
    }

    synchronized void b2(){
        System.out.println("Starting b2");
    }
}
```

```java
class Thread1 extends Thread {
    A a;

    Thread1(A a){
        this.a = a;
    }

    public void run() {
        for(int i=0; i<100000;i++)
            a.a1();
    }
}

class Thread2 extends Thread {
    B b;

    Thread2(B b){
        this.b = b;
    }

    public void run(){
        for(int i=0; i<100000; i++)
            b.b1();
    }
}
```

```
class DeadlockDemo {

    public static void main(String[] args) {

        // オブジェクトの作成
        A a = new A();
        B b = new B();
        a.b = b;
        b.a = a;

        // スレッドの作成
        Thread1 t1 = new Thread1(a);
        Thread2 t2 = new Thread2(b);
        t1.start();
        t2.start();

        // スレッドの完了を待つ
        try {
            t1.join();
            t2.join();
        }
        catch(Exception e) {
            e.printStackTrace();
        }

        // メッセージの表示
        System.out.println("Done!");
    }

}
```

実行結果

```
Starting a1
Starting b2
Starting a1
Starting b2
Starting a1
Starting b1
```

1つ目のスレッドでは、4回のループが正常に実行される。ループのたびにa1()メソッドからb2()メソッドが呼び出され、制御が返ってくる、さらに1つ目のスレッドがa1()メソッドを再び呼び出します。

その後で処理対象が切り替えられ、2つ目のスレッドがb1()メソッドを呼び出します。2つ目のスレッドはAオブジェクトのロックが解放されるのを待ちますが、1つ目のスレッドがBオブジェクトのロックの解放を待っているので、デッドロックに陥ります。

9.5 スレッド間の通信

スレッド間の通信

スレッドの動作を協調させる。スレッドから一時的にロックを解放し、他のスレッドにsynchronizedブロックを実行するチャンスを与えることができる。このロックは後でまた取得できる。

Objectクラスには、メソッド間の通信に使用できるメソッドが3つ用意されている。

wait() メソッド

```
void wait() throws InterruptedException
void wait(long msec) throws InterruptedException
void wait(long msec, int nsec) throws InterruptedException
```

1つのwait()メソッドは、synchronizedメソッドまたはsynchronizedブロックを実行しているスレッドでロックを解放し、ほかのスレッドからの通知を待つ。

1つ目の構文のwait()メソッドを使うと、現在のスレッドは無期限に待機する。2つ目の構文を使うと、msecミリ秒間待機する。3つ目の構文を使うと、msecミリ秒とnsecナノ秒を足した時間の間待機する。

スレッド間の通信

notify( )メソッド

void notify( )

notify()メソッドは、synchronizedメソッドまたはsynchronizedブロックを実行しているスレッドから、そのオブジェクトのロック解放を待機しているスレッドに対して通知を送る。どういった基準でスレッドを1つ選ぶかは、JVMの実装元によって決められる。

notifyAll( )メソッド

void notifyAll( )

notifyAll()メソッドは、synchronizedメソッドまたはsynchronizedブロックを実行しているスレッドから、そのオブジェクトのロック解放を待機している全てのスレッドに対して通知を送る。

これらのメソッドを呼び出してもロックは解放されない。ロックが解放されるのはsynchronizedメソッドまたはsynchronizedブロックを抜けるとき。notify()メソッドやnotifyAll()メソッドによって、1つのスレッドのsynchronizedメソッドまたはsynchronizedブロックの実行が再開される。スレッドはwait()メソッドから戻ってきて、次のコードから実行を再開する。

```
class Producer extends Thread {
	Queue queue;

	Producer(Queue queue){
		this.queue = queue;
	}

	public void run(){
		int i = 0;
		while(true){
			queue.add(i++);
		}
	}
}
class Consumer extends Thread {
	String str;
	Queue queue;

	Consumer(String str, Queue queue){
		this.str = str;
		this.queue = queue;
	}
	public void run(){
		while(true){
			System.out.println(str + ": " +
queue.remove());
		}
	}
}
```

```
class Queue {
	private final static int SIZE = 10;
	int array[] = new int[SIZE];
	int r = 0;
	int w = 0;
	int count = 0;

	synchronized void add(int i){

		// 待ち行列がいっぱいの場合は待機する
		while(count == SIZE){
			try{
				wait();
			}
			catch(InterruptedException ie) {
				ie.printStackTrace();
				System.exit(0);
			}
		}

		// 配列にデータを追加して書き込みポインタを更
新する
		array[w++] = i;
		if(w >= SIZE)
			w = 0;

		// countカウントを1つ増やす
		++count;

		// 待機中のスレッドに通知する
		notifyAll();
	}
```

```
synchronized int remove(){

        // 待ち行列が空の場合は待機する
        while(count == 0){
            try{
                wait();
            }
            catch(InterruptedException ie){
                ie.printStackTrace();
                System.exit(0);
            }
        }

// 配列からデータを読み取って読み取りポインタを更新
        int element = array[r++];
        if (r >= SIZE)
            r = 0;

// countを1つ減らす
        --count;

// 待機中のスレッドに通知する
        notifyAll();

        // データを返す
        return element;
    }
}
```

```
class ProducerConsumers {

    public static void main(String[] args) {
        Queue queue = new Queue();
        new Producer(queue).start();
        new Consumer("ConsumerA", queue).start();
        new Consumer("ConsumerB", queue).start();
        new Consumer("ConsumerC", queue).start();
    }
}
```

実行結果

```
ConsumerA: 0
ConsumerA: 1
ConsumerA: 2
ConsumerA: 3
ConsumerA: 4
ConsumerC: 5
ConsumerC: 6
ConsumerC: 7
ConsumerC: 8
ConsumerC: 9
ConsumerC: 10
ConsumerC: 11
ConsumerC: 12
ConsumerA: 14
ConsumerA: 15
ConsumerA: 16
ConsumerA: 17
ConsumerA: 18
ConsumerA: 19
ConsumerA: 20
ConsumerC: 13
ConsumerC: 21
ConsumerC: 22
ConsumerC: 23
ConsumerC: 24
ConsumerC: 25
    ・
    ・
    ・
```

練習問題 ①

問題1

次のプログラムは、デッドロックが起こるかどうか説明しなさい。

```
class Q{
	synchronized void q1(){
		q2();
	}
	synchronized void q2(){
	}
}

class ThreadQ extends Thread {
	Q q;

	ThreadQ(Q q) {
		this.q = q;
	}

	public void run(){
		for(int i=0;i<100000;i++)
			q.q1();
	}
}
```

```
class DeadlockQ {
	private final static int NUMTHREADS = 10;
	public static void main(String[] args) {
		// オブジェクトの作成
		Q q = new Q();

		// スレッドの作成
		ThreadQ threads[] = new ThreadQ[NUMTHREADS];
		for(int i=0;i<NUMTHREADS;i++){
			threads[i] = new ThreadQ(q);
			threads[i].start();
		}

		// スレッドの完了を待つ
		for (int i=0;i<NUMTHREADS;i++)
			try{
				threads[i].join();
			}
			catch(Exception e){
				e.printStackTrace();
			}

		//メッセージの表示
		System.out.println("完了");
	}
}
```

練習問題 ②

問題2

１０匹のねずみが箱を出入りするとする。ねずみは10秒以上20秒未満の間、箱の外で過ごした後、箱に入って5秒以上8秒未満の間そこで過ごし、また外に出る。箱の中に入れるねずみは4匹までとする。箱がいっぱいの場合、入ろうとするねずみを待たせる必要がある。このような状態をシミュレートするマルチスレッドプログラムを作成しなさい。ただし各ねずみの動作を管理するためにスレッドを使う。ねずみが出入りするたびに、箱の中にいるねずみの数を表示しなさい。