

本日のお品書き

- ・ API には何があるの？

- ・ JSpider のオブジェクトモデル
 - 概要図
 - オブジェクトモデルの実装
 - Site interface
 - Resource interface

- ・ Tasking & Threading
 - 予備知識： タスクの概念を使ったスレッド・モデルについて
 - Task と WorkerTask
 - Task の実装
 - WorkerThread
 - Scheduler

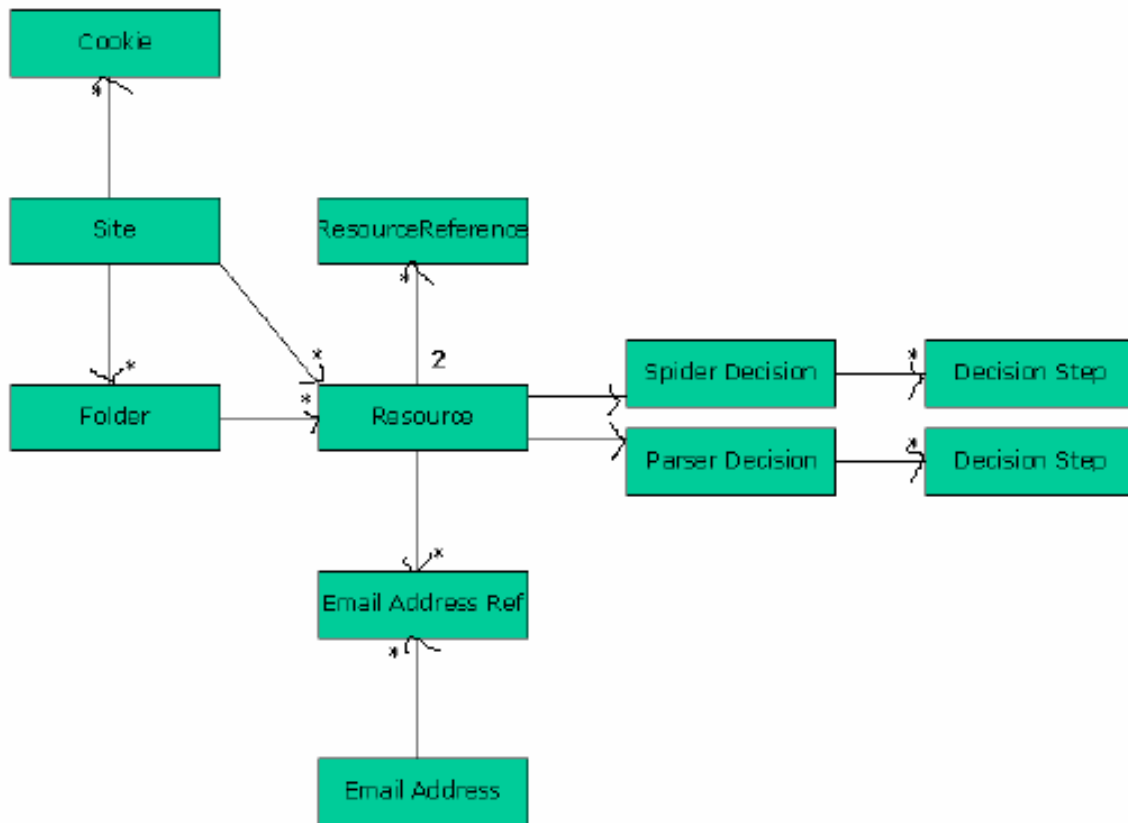
- ・ API には何があるの？

SPI と Core に挟まれてその位置づけがいまいち分からないですね、API って。ドキュメントとソースを読む限りでは、JSpider の内部的なオブジェクトモデルが API に該当するっぽいです。JSpider では実世界の実体に対応するクラスが丁寧に実装されています。たとえば一つの Web サイトに対応するとして Site interface (`¥net¥javacoding¥jspider¥api¥model¥Site.java`) があり、一つのメールアドレスに対応するクラスとして EMailAddress interface (`¥net¥javacoding¥jspider¥api¥model¥EMailAddress.java`) があります。で、これらを継承して細々としたクラスが実装されている、というような具合です。これらのクラスを総じて、API としているのだとおもわれます。

なのでそれらのクラスを紹介していけば良いのですが、それだけだとネタが持たないっつーか30分もしないで終わってしまうので、第一回目でちょっとだけ取り上げた micro-tasks の仕組み (イコール、JSpider の Tasking と Threading の機構) を見ていこうとおもいます。これらはプログラム全体を横断するもので、どこのレイヤに属するというものではないので (強いて挙げるなら Core ですかね)、ここで扱ってしましましょう。

・ JSpider のオブジェクトモデル

概要図



まずオブジェクトモデルの概要図を載せます。公式ドキュメントの p23 にあるものに、所有/被所有の関係を矢印で書き加えたものです。指している側が所有する側、指されている側が所有される側になります。

このなかでは、"Site"と"Resource"が、他のクラスを集約する中心的な位置づけとなっています。読むときにはこのあたりから手を着けるのが宜しいでしょう。

オブジェクトモデルの実装

流れとしてはこれらのクラスの実装を見ていくことになるわけですが、こいつらはとりたてて紹介するような特別なテクニックを使って実装されているわけではなく、ソースコ

ードをこのレジュメに貼り付けて、みんなで読むほどのものではありません。そのクラスに関連する情報をまとめて管理するだけの、構造的な実装になっているものが多いです。

基本的に、個々の実装は読みたい人が勝手に読むということにして、ここでは Site interface と Resource interface のみを取り上げることにします。

Site interface

ここで注目なのは public static final で宣言されてる5つの定数です。それ以外は割とどうでもよいです。これら5つの定数は、そのサイトの robots.txt に関する状態を表します。クラス(Site interface を継承して実装されたクラスね)がインスタンスングされた時点ではこれは STATE_DISCOVERED に初期化されており、その後、robots.txt をチェックし、他の4つのいずれかに遷移します。

- ・ STATE_DISCOVERED

まだ robots.txt をチェックしていない状態

- ・ STATE_ROBOTSTXT_HANDLED

robots.txt をダウンロードした状態

以後、JSpider は robots.txt の指示に従って動作する

- ・ STATE_ROBOTSTXT_ERROR

robots.txt のダウンロードに失敗した状態

このサイトから一切のリソースをダウンロードできないとする(何が書いてあるかわかんないから?)

- ・ STATE_ROBOTSTXT_UNEXISTING

robots.txt が存在しなかった状態

このサイトに存在するすべてのリソースをダウンロードできる

- ・ STATE_ROBOTSTXT_SKIPPED

設定ファイルの指定によって、robots.txt を扱わないようにしてあった場合、この状態になる

このサイトに存在するすべてのリソースをダウンロードできる

これらの定数ですが、FetchRobotsTXTTaskImpl クラス (¥net¥javacoding¥jspider¥core¥task¥work¥FetchRobotsTXTTaskImpl.java) が robots.txt をダウンロードし、AgentImpl クラス (¥net¥javacoding¥jspider¥core¥impl¥AgentImpl.java) を経て適当なものが設定されるようになっています。また、ダウンロードされた robots.txt は RobotsTXTRule クラス (¥net¥javacoding¥jspider¥core¥rule¥impl¥RobotsTXTRule.class) RobotsTXTRule クラス (¥net¥javacoding¥jspider¥core¥util¥html¥RobotsTXTRule.java) RobotsTXTRule クラス (¥net¥javacoding¥jspider¥core¥util¥html¥RobotsTXTRule.java) の3つのクラスで解釈されます。このあたりのコードを弄くると、robots.txt の扱いを思うがままにすることができます:-)

作成された SiteInternal クラスのインスタンスは、は SiteDAOImpl クラス (¥net¥javacoding¥jspider¥core¥storage¥impl¥SiteDAOImpl.class) によって管理されます。このクラスはたぶん次回に扱います。

```
/**
 *
 * $Id: Site.java,v 1.23 2003/04/10 16:19:03 vanrogu Exp $
 *
 * @author G?nther Van Roey
 */
public interface Site {

    /**
     * if the site is newly discovered, we're not going to fetch resources
     * until we have interpreted the site's robots.txt rules.
     */
    public static final int STATE_DISCOVERED = 0;

    /**
     * if the robots.txt was handled (interpreted or missing), we can
     * fetch resources from it.
     */
    public static final int STATE_ROBOTSTXT_HANDLED = 1;

    /**
     * if robots.txt couldn't be fetched (but seems to be there), no resources
     * will be fetched from the site!
     */
    public static final int STATE_ROBOTSTXT_ERROR = 2;

    /**
     * if robots.txt was not found, all resources can be fetched !
     */
    public static final int STATE_ROBOTSTXT_UNEXISTING = 3;

    /**
     * if robots.txt was skipped, all resources can be fetched !
     */
    public static final int STATE_ROBOTSTXT_SKIPPED = 4;

    public int getState();
}
```

```
public String getHost();

public int getPort();

public boolean isRobotsTXTHandled();

public boolean getObeyRobotsTXT();

public boolean getFetchRobotsTXT();

public URL getURL();

public Folder[] getRootFolders();

public Folder getRootFolder(String name);

public Resource[] getRootResources();

public Resource[] getAllResources();

public Cookie[] getCookies();

public String getCookieString();

public boolean getUseCookies();

public boolean getUseProxy();

public String getUserAgent ( );

public boolean isBaseSite ( );

public boolean mustHandle ( );

}
```

Resource interface

ここで重要なのは STATE_ で始まる 8 つの定数です。Resource は Web サイトにあるリソースを、データの形式を問わずに表現する interface です。JSpider ではクローリングの過程を spidering と parsing に分けており、それぞれについて Rule を設定するようになっています。Resource interface では、これらの定数によって、リソースがクローリングの過程のなかのどこにいるか、またそのリソースがどういう扱いになっているか (=Rule にどう判断されたか) を表します。

STATE_DISCOVERED は、HTML がパースされ、その URL が発見された状態を表します。インスタンス化された時点では、クラスはこの状態に設定されます。ダウンロードに成功すると STATE_FETCHED に、パースまで成功すると STATE_PARSED に遷移します。それぞれの処理に失敗するか、Rule によって弾かれると、それに応じた状態が設定

されます。どうして"STATE_PARSE_FORBIDDEN"が無いのかはよくわかんない。なんだろう。

```
/**
 *
 * $Id: Resource.java,v 1.8 2003/04/09 17:08:03 vanrogu Exp $
 *
 * @author G?nther Van Roey
 */
public interface Resource {

    public static final int STATE_DISCOVERED = 1;
    public static final int STATE_FETCH_ERROR = 2;
    public static final int STATE_FETCH_IGNORED = 3;
    public static final int STATE_FETCH_FORBIDDEN = 4;
    public static final int STATE_FETCHED = 5;
    public static final int STATE_PARSE_ERROR = 6;
    public static final int STATE_PARSE_IGNORED = 7;
    public static final int STATE_PARSED = 8;

    public int getState();

    public String getStateName ();

    public URL getURL();

    public String getFileName();

    public Site getSite();

    public Folder getFolder();

    public String getName();

    public Date getDiscoveryTime();

    public Resource[] getReferers();

    public Decision getSpiderDecision ( );

    public Decision getParseDecision ( );

}
```

・Tasking & Threading

予備知識： タスクの概念を使ったスレッド・モデルについて

ええと。予備知識として、軽く「タスク」の概念を取り入れたスレッドモデルについて触れておきたいと思います。

プログラム中で行われる処理を小さな単位に切り分け、これを「タスク」と呼びます。この「タスク」の考え方をを使ったスレッド管理は、不定期に必要な処理が発生するようなプログラム、特にサーバの実装で顕著に見られるものです。これらのスレッド管理について、主立ったスレッドモデルを紹介します。

第一に。もっとも単純なモデルとしては、タスクが発生するたびに、新しいスレッドを作成し、アサインするというものがあります。このモデルでは、タスクのインスタンスとスレッドのインスタンスは1対1で作成され、タスクの処理が終了すれば、そのスレッドは破棄されます。

(ポンチ絵)

このモデルは単純明快で実装も容易ですが、いくつかの問題があります。

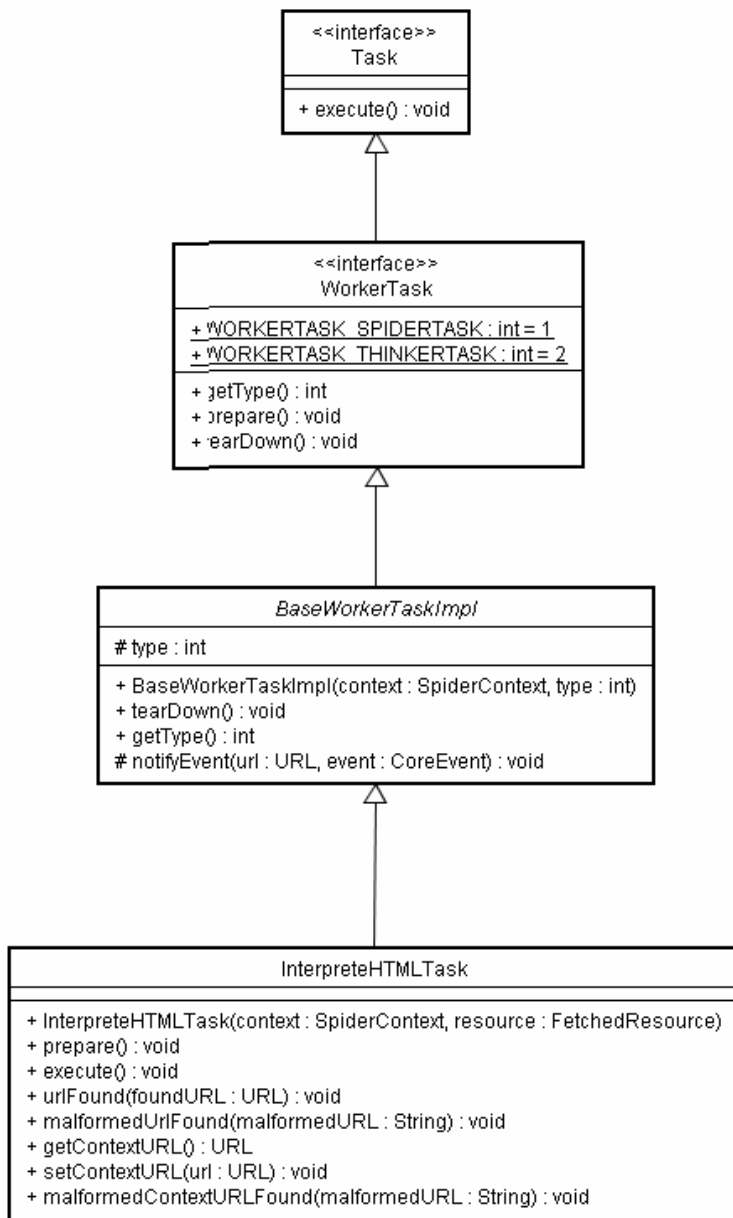
- ・ スレッドの作成と破棄に大きなコストが掛かる。スレッドの作成と破棄に要するコストは思いのほか大きく、スレッドの作成と破棄のために、タスクを処理するよりも多くの時間を費やすこともあります。
- ・ スレッドを多く作りすぎると、リソースの食い過ぎで全体的なパフォーマンスが落ちる。
- ・ セキュリティ・ホールになる

.....で。この問題を解決するスレッドモデルとして、とりあえず2つ紹介します

- ・ Thread Pool (スレッド・プール)
- ・ Worker Thread(ワーカ・スレッド)

(ポンチ絵)

Task と WorkerTask



JSpider では、クローリングに関わるほぼすべての処理を、Task という interface を継承したクラスに実装しています。

```

/**
 * Interface that will be implemented upon each Task to be carried out
 * by a Thread.
 *
 * $Id: Task.java,v 1.4 2003/04/25 21:29:02 vanrogu Exp $
 */
    
```

```
* @author G?nther Van Roey
*/
public interface Task {

    /**
     * Has the task executed. The thread calling this method will do it's
     * time in there :).
     */
    public void execute();
}
```

Task interface は、execute 関数が宣言されただけのシンプルな interface です。クローリングに必要な処理をこの Task を継承したクラスとして実装し、execute 関数をスレッドが実行することによって、クローリングが行われます。

また、Task を継承した interface である WorkerTask interface があります。

```
/**
 * Interface that will be implemented upon each class that represents a
 * JSpider workertask that needs to be executed by a Worker Thread.
 *
 * JSpider has two types of tasks: spider tasks (that fetch data from a
 * web server), and thinker tasks (that interpret data, take decision,
 * etc...)
 *
 * $Id: WorkerTask.java,v 1.5 2003/04/25 21:29:02 vanrogu Exp $
 *
 * @author G?nther Van Roey
 */
public interface WorkerTask extends Task {

    /**
     * Task type that is used for every task that will require the fetching
     * of data from a site.
     */
    public static final int WORKERTASK_SPIDERTASK = 1;

    /**
     * Task type used for all tasks that don't require any fetching of data
     */
    public static final int WORKERTASK_THINKERTASK = 2;

    /**
     * Returns the type of the task - spider or thinker.
     * @return the type of the task
     */
    public int getType ( );

    /**
     * Allows some work to be done before the actual Task is carried out.
     * During the invocation of prepare, the WorkerThread's state will be
     * WORKERTHREAD_BLOCKED.
     */
    public void prepare ( );

    /**
     * Allows us to put common code in the abstract base class.
     */
    public void tearDown ( );
}
```

```
}
```

Task の実装

WorkerTask を実装したクラスとして BaseWorkerTaskImpl クラスがあります。JSpi
der で実装されているタスクは、実際にはすべてこの BaseWorkerTaskImpl を継承するこ
とで実装されています。Task, WorkerTask を直接 implements しているものはひとつもあ
りません。

```
/**
 *
 * $Id: BaseWorkerTaskImpl.java,v 1.4 2003/04/10 16:19:08 vanrogu Exp $
 *
 * @author G?nther Van Roey
 */
public abstract class BaseWorkerTaskImpl implements WorkerTask {

    protected Log log;
    protected int type;
    protected SpiderContext context;

    public BaseWorkerTaskImpl(SpiderContext context, int type) {
        this.log = LogFactory.getLog(this.getClass());
        this.type = type;
        this.context = context;
    }

    public void tearDown() {
        context.getAgent().flagDone(this);
    }

    public int getType() {
        return type;
    }

    protected void notifyEvent(URL url, CoreEvent event) {
        if ( event == null ) {
            log.error("PANIC! event is null!");
            log.error("URL: " + url);
        } else {
            context.getAgent().registerEvent(url, event);
        }
    }
}
```

最終的なタスクの実装は¥net¥javacoding¥jspider¥core¥task¥work 以下にあります。

- DecideOnSpideringTask

処理対象となっているリソースについて、Rule に問い合わせを行う

• DecideOnParsingTask

同上

• FetchRobotsTXTTaskImpl

robots.txt をダウンロードする

また、その結果に応じて、適切な Event を発行する

• SpiderHttpURLTask

指定された URL から HTTP プロトコルを用いてリソースをダウンロードする

• InterpretHTMLTask

HTML をパースし、URL の抽出といった諸々の処理を行う

WorkerThread

タスクは Java の Thread クラスを継承した WorkerThread クラス (¥net¥javacoding¥j spider¥core¥threading¥WorkerThread.java) によって実行されます。がんばって読みましょう。

```
/**
 * Implementation of a Worker Thread.
 * This thread will accept WorkerTasks and execute them.
 *
 * $Id: WorkerThread.java,v 1.11 2003/04/02 20:55:26 vanrogu Exp $
 *
 * @author G?nther Van Roey
 */
class WorkerThread extends Thread {

    public static final int WORKERTHREAD_IDLE = 0;
    public static final int WORKERTHREAD_BLOCKED = 1;
    public static final int WORKERTHREAD_BUSY = 2;

    /** the current state of this thread - idle, blocked, or busy. */
    protected int state;

    /** Whether this instance is assigned a task. */
    protected boolean assigned;

    /** Whether we should keep alive this thread. */
    protected boolean running;

    /** Threadpool this worker is part of. */
    protected WorkerThreadPool stp;
```

```
/** Task this worker is assigned to. */
protected WorkerTask task;

/**
 * Public constructor.
 * @param stp thread pool this worker is part of
 * @param name name of the thread
 * @param i index in the pool
 */
public WorkerThread(WorkerThreadPool stp, String name, int i) {
    super(stp, name + " " + i);
    this.stp = stp;
    running = false;
    assigned = false;
    state = WORKERTHREAD_IDLE;
}

/**
 * Tests whether this worker thread instance can be assigned a task.
 * @return whether we're capable of handling a task.
 */
public boolean isAvailable() {
    return (!assigned) && running;
}

/**
 * Determines whether we're occupied.
 * @return boolean value representing our occupation
 */
public boolean isOccupied() {
    return assigned;
}

/**
 * Method that allows the threadPool to assign the worker a Task.
 * @param task WorkerTask to be executed.
 */
public synchronized void assign(WorkerTask task) {
    if ( !running ) {
        //SHOULDN'T HAPPEN WITHOUT BUGS
        throw new RuntimeException("THREAD NOT RUNNING, CANNOT ASSIGN TASK !!!");
    }
    if (assigned) {
        //SHOULDN'T HAPPEN WITHOUT BUGS
        throw new RuntimeException("THREAD ALREADY ASSIGNED !!!");
    }
    this.task = task;
    assigned = true;
    notify();
}

/**
 * Tells this thread not to accept any new tasks.
 */
public synchronized void stopRunning() {
    if ( ! running ) {
        throw new RuntimeException ( "THREAD NOT RUNNING - CANNOT STOP !");
    }
    if ( assigned ) {
        try {
            this.wait();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

```
    }
    running = false;
    notify();
}

/**
 * Returns the state of this worker thread (idle, blocked or busy).
 * @return
 */
public int getState ( ) {
    return state;
}

/**
 * Thread's overridden run method.
 */
public synchronized void run() {
    running = true;

    Log log = LogFactory.getLog(WorkerThread.class);
    log.debug("Worker thread (" + this.getName() + ") born");

    synchronized (stp) {
        stp.notify();
    }

    while (running) {
        if (assigned) {
            state = WORKERTHREAD_BLOCKED;
            task.prepare();
            state = WORKERTHREAD_BUSY;
            try {
                task.execute();
                task.tearDown();
            } catch (Exception e) {
                log.fatal("PANIC! Task " + task + " threw an excpation!", e);
                System.exit(1);
            }

            synchronized (stp) {
                assigned = false;
                task = null;
                state = WORKERTHREAD_IDLE;
                stp.notify();
                this.notify(); // if some thread is blocked in stopRunning();
            }

        }
        try {
            wait();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    /* notify the thread pool that we died. */
    log.debug("Worker thread (" + this.getName() + ") dying");
}
}
```

WorkerThreadPool

作成された WorkerThread クラスのインスタンスは、WorkerThreadPool クラス (¥net ¥javacoding ¥jspider ¥core ¥threading ¥WorkerThreadPool.java) によって管理されます。がんばって読みましょう。

```
/**
 * Thread Pool implementation that will be used for pooling the spider and
 * parser threads.
 *
 * $Id: WorkerThreadPool.java,v 1.7 2003/02/27 16:47:49 vanrogu Exp $
 *
 * @author G?nther Van Roey
 */
public class WorkerThreadPool extends ThreadGroup {

    /** Task Dispatcher thread associated with this threadpool. */
    protected DispatcherThread dispatcherThread;

    /** Array of threads in the pool. */
    protected WorkerThread[] pool;

    /** Size of the pool. */
    protected int poolSize;

    /**
     * Public constructor
     * @param poolName name of the threadPool
     * @param threadName name for the worker Threads
     * @param poolSize number of threads in the pool
     */
    public WorkerThreadPool(String poolName, String threadName, int poolSize) {
        super(poolName);

        this.poolSize = poolSize;

        dispatcherThread = new DispatcherThread(this, threadName + " dispatcher", this);
        pool = new WorkerThread[poolSize];
        for (int i = 0; i < poolSize; i++) {
            pool[i] = new WorkerThread(this, threadName, i);
            synchronized (this) {
                try {
                    pool[i].start();
                    wait();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }

    /**
     * Assigns a worker task to the pool. The threadPool will select a worker
     * thread to execute the task.
     * @param task the WorkerTask to be executed.
     */
    public synchronized void assign(WorkerTask task) {
        while (true) {
            for (int i = 0; i < poolSize; i++) {
                if (pool[i].isAvailable()) {
                    pool[i].assign(task);
                    return;
                }
            }
        }
    }
}
```

```
        try {
            wait();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

/**
 * Assigns a DispatcherTask to the threadPool. The dispatcher thread
 * associated with the threadpool will execute it.
 * @param task DispatcherTask that will keep the workers busy
 */
public void assignGroupTask(DispatcherTask task) {
    dispatcherThread.assign(task);
}

/**
 * Returns the percentage of worker threads that are busy.
 * @return int value representing the percentage of busy workers
 */
public int getOccupation() {
    int occupied = 0;
    for (int i = 0; i < poolSize; i++) {
        WorkerThread thread = pool[i];
        if (thread.isOccupied()) {
            occupied++;
        }
    }
    return (occupied * 100) / poolSize;
}

public int getBlockedPercentage() {
    int counter = 0;
    for (int i = 0; i < poolSize; i++) {
        WorkerThread thread = pool[i];
        if (thread.getState() == WorkerThread.WORKERTHREAD_BLOCKED ) {
            counter++;
        }
    }
    return (counter * 100) / poolSize;
}

public int getBusyPercentage ( ) {
    int counter = 0;
    for (int i = 0; i < poolSize; i++) {
        WorkerThread thread = pool[i];
        if (thread.getState() == WorkerThread.WORKERTHREAD_BUSY) {
            counter++;
        }
    }
    return (counter * 100) / poolSize;
}

public int getIdlePercentage ( ) {
    int counter = 0;
    for (int i = 0; i < poolSize; i++) {
        WorkerThread thread = pool[i];
        if (thread.getState() == WorkerThread.WORKERTHREAD_IDLE ) {
            counter++;
        }
    }
    return (counter * 100) / poolSize;
}
}
```

```
/**
 * Causes all worker threads to die.
 */
public void stopAll() {
    for (int i = 0; i < pool.length; i++) {
        WorkerThread thread = pool[i];
        thread.stopRunning();
    }
}

/**
 * Returns the number of worker threads that are in the pool.
 * @return the number of worker threads in the pool
 */
public int getSize ( ) {
    return poolSize;
}
}
```

Scheduler

Scheduler の機能は次の 2 つです。

- ・タスクを一時的に貯めておくキューとしてはたらく
- ・URL を指定してタスクの実行をブロックする (未実装かも)

Scheduler は Spider タスク用、Thinker タスク用の 2 つのリストを持ち、それぞれのタスクを管理します。タスクを put & get する機能と、それに付随するタスクの個数取得などの機能を持ちます。また、URL 単位でタスクをブロックする機能を実装ですが、ブロック対象となっている URL を格納するリストは宣言されているんですがそれがどこでも使われておらず、未実装なんじゃないかとおもわれます。

Scheduler インターフェース

```
/**
 * Interface that will be implemented upon each object that will act as a task
 * scheduler.
 * The Task scheduler will keep track of all work that is being done and all
 * tasks that still have to be carried out.
 *
 * $Id: Scheduler.java,v 1.10 2003/04/25 21:28:59 vanrogu Exp $
 *
 * @author G?nther Van Roey
 */
public interface Scheduler {

    /**
     * Schedules a Worker Task to be executed. The scheduler will keep a
     * reference to the task and return it later on to be processed.
     * @param task task to be scheduled
     */
}
```

```
*/
public void schedule(WorkerTask task);

/**
 * Block a task for a certain siteURL. This is used to block any
 * resource handling for a site for which we didn't interpret the
 * robots.txt file yet.
 * @param siteURL the site for which the task is
 * @param task the task to be temporarily blocked
 */
public void block( URL siteURL, DecideOnSpideringTask task);

/**
 * Returns all tasks that were blocked for the specified site, and
 * removes them from the blocked resources pool.
 * @param siteURL the site we want to unblock all resources for
 * @return array with all tasks that were blocked for this site
 */
public DecideOnSpideringTask[] unblock(URL siteURL);

/**
 * Flags a task as done. This way, we are able to remove the task from
 * the in-process list.
 * @param task task that was completed
 */
public void flagDone(WorkerTask task);

/**
 * Returns a thinker task to be processed
 * @return Task to be carried out
 * @throws TaskAssignmentException if all the work is done or no suitable
 * items are found for the moment.
 */
public WorkerTask getThinkerTask() throws TaskAssignmentException;

/**
 * Returns a fetch task to be processed
 * @return Task to be carried out
 * @throws TaskAssignmentException if all the work is done or no suitable
 * items are found for the moment.
 */
public WorkerTask getFethTask() throws TaskAssignmentException;

/**
 * Determines whether all the tasks are done. If there are no more tasks
 * scheduled for process, and no ongoing tasks, it is impossible that new
 * work will arrive, so the spidering is done.
 * @return boolean value determining whether all work is done
 */
public boolean allTasksDone();

/**
 * Statistics method.
 * @return blocked jobs counter
 */
public int getBlockedCount( );

/**
 * Statistics method.
 * @return assigned jobs counter
 */
public int getAssignedCount( );

/**
 * Statistics method.
```

```
    * @return total jobs counter
    */
    public int getJobCount ( );

    /**
     * Statistics method.
     * @return total thinker jobs counter
     */
    public int getThinkerJobCount ( );

    /**
     * Statistics method.
     * @return total spider jobs counter
     */
    public int getSpiderJobCount ( );

    /**
     * Statistics method.
     * @return jobs finished counter
     */
    public int getJobsDone ( );

    /**
     * Statistics method.
     * @return finished spider jobs counter
     */
    public int getSpiderJobsDone ( );

    /**
     * Statistics method.
     * @return finished thinker jobs counter
     */
    public int getThinkerJobsDone ( );
}
```

SchedulerImpl クラス

```
/**
 * Default implementation of a Task scheduler
 *
 * $Id: SchedulerImpl.java,v 1.17 2003/04/25 21:29:04 vanrogu Exp $
 *
 * @author G?nther Van Roey
 */
public class SchedulerImpl implements Scheduler {

    /** List of fetch tasks to be carried out. */
    protected List fetchTasks;

    /** List of thinker tasks to be carried out. */
    protected List thinkerTasks;

    /** Set of tasks that have been assigned. */
    protected Set assignedSpiderTasks;
    protected Set assignedThinkerTasks;

    protected int spiderTasksDone;
    protected int thinkerTasksDone;

    protected Map blocked;

    int blockedCount = 0;

    public int getBlockedCount( ) {
        return blockedCount;
    }

    public int getAssignedCount( ) {
        return assignedSpiderTasks.size() + assignedThinkerTasks.size();
    }

    public int getJobCount() {
        return getThinkerJobCount() + getSpiderJobCount();
    }

    public int getThinkerJobCount() {
        return thinkerTasksDone + assignedThinkerTasks.size ( ) + thinkerTasks.size();
    }

    public int getSpiderJobCount() {
        return spiderTasksDone + assignedSpiderTasks.size ( ) + fetchTasks.size();
    }

    public int getJobsDone() {
        return getSpiderJobsDone() + getThinkerJobsDone();
    }

    public int getSpiderJobsDone() {
        return spiderTasksDone;
    }

    public int getThinkerJobsDone() {
        return thinkerTasksDone;
    }

    /**
     * Public constructor?
     */
}
```

```
public SchedulerImpl() {
    fetchTasks = new ArrayList();
    thinkerTasks = new ArrayList();
    assignedThinkerTasks = new HashSet();
    assignedSpiderTasks = new HashSet();
    blocked = new HashMap();
}

public void block(URL siteURL, DecideOnSpideringTask task) {
    ArrayList al = (ArrayList)blocked.get(siteURL);
    if ( al == null ) {
        al = new ArrayList();
        blocked.put(siteURL, al);
    }
    int before = al.size();
    al.add(task);
    int after = al.size();

    int diff = after-before;
    blockedCount+=diff;
}

public DecideOnSpideringTask[] unblock(URL siteURL) {
    ArrayList al = (ArrayList)blocked.remove(siteURL);
    if ( al == null ) {
        return new DecideOnSpideringTask[0];
    } else {
        blockedCount-=al.size();
        return (DecideOnSpideringTask[]) al.toArray(new DecideOnSpideringTask[al.size()]);
    }
}

/**
 * Schedules a task to be processed.
 * @param task task to be scheduled
 */
public synchronized void schedule(WorkerTask task) {
    if (task.getType() == WorkerTask.WORKERTASK_SPIDERTASK ) {
        fetchTasks.add(task);
    } else {
        thinkerTasks.add(task);
    }
}

/**
 * Flags a task as done.
 * @param task task that was complete
 */
public synchronized void flagDone(WorkerTask task) {
    if (task.getType() == WorkerTask.WORKERTASK_THINKERTASK ) {
        assignedThinkerTasks.remove(task);
        thinkerTasksDone++;
    }else{
        assignedSpiderTasks.remove(task);
        spiderTasksDone++;
    }
}

public synchronized WorkerTask getThinkerTask() throws TaskAssignmentException {
    if (thinkerTasks.size() > 0) {
        WorkerTask task = (WorkerTask) thinkerTasks.remove(0);
        assignedThinkerTasks.add(task);
        return task;
    }
    if (allTasksDone()) {
```

```
        throw new SpideringDoneException();
    } else {
        throw new NoSuitableItemFoundException();
    }
}

/**
 * Returns a fetch task to be carried out.
 * @return WorkerTask task to be done
 * @throws TaskAssignmentException notifies when the work is done or there
 * are no current outstanding tasks.
 */
public synchronized WorkerTask getFethTask() throws TaskAssignmentException {
    if (fetchTasks.size() > 0) {
        WorkerTask task = (WorkerTask) fetchTasks.remove(0);
        assignedSpiderTasks.add(task);
        return task;
    }
    if (allTasksDone()) {
        throw new SpideringDoneException();
    } else {
        throw new NoSuitableItemFoundException();
    }
}

/**
 * Determines whether all the tasks are done. If there are no more tasks
 * scheduled for process, and no ongoing tasks, it is impossible that new
 * work will arrive, so the spidering is done.
 * @return boolean value determining whether all work is done
 */
public synchronized boolean allTasksDone() {
    return (fetchTasks.size() == 0 &&
            thinkerTasks.size() == 0 &&
            assignedSpiderTasks.size() == 0 &&
            assignedThinkerTasks.size() == 0 &&
            blocked.size() == 0);
}

/**
public synchronized String toString ( ) {
    StringBuffer sb = new StringBuffer();
    Iterator it = this.thinkerTasks.iterator();
    while ( it.hasNext() ) {
        System.out.println("TH . " + it.next().getClass());
    }
    it = this.assignedThinkerTasks.iterator();
    while ( it.hasNext() ) {
        System.out.println("TH A " + it.next().getClass());
    }
    it = this.fetchTasks.iterator();
    while ( it.hasNext() ) {
        System.out.println("SP . " + it.next().getClass());
    }
    it = this.assignedSpiderTasks.iterator();
    while ( it.hasNext() ) {
        System.out.println("SP A " + it.next().getClass());
    }
    return sb.toString();
}
*/
}
```