

Java で XMPP #1

Smack の使い方(1): EchoBot

PTaaS Trebuchet ではツールの遠隔実行・管理・監視といった機能を実現するため XMPP という通信プロトコルを利用します。本稿では Java 言語による XMPP プログラミングを扱った日本語ドキュメントが少ないことから、サンプル・コードを挙げて使い方を簡単に説明してみたいと思います。

本稿では Java 言語で XMPP のクライアントを作成するためのライブラリである Smack についてサンプル・コードを挙げて使い方を簡単に説明してみたいと思います。サンプルとしては非常に簡単な自動応答プログラム（ボットと呼ばれる）の例として EchoBot というプログラムを作成します。

開発言語としてはJava (Ver.6)¹、XMPPライブラリとしてはSmack²、XMPPサーバとしてはOpenfire³を想定しています。動作確認はWindows 7 環境、ブラウザはFirefox 5で行いました。

¹ <http://java.sun.com/javase/ja/6/download.html>

² <http://www.igniterealtime.org/projects/smack/index.jsp>

³ <http://www.igniterealtime.org/projects/openfire/index.jsp>

目次

Java で XMPP #1	1
1 対象読者	3
2 環境設定	3
Openfire	3
Smack	8
3 EchoBot	8
3-1 EchoBot の動作	8
3-2 EchoBot のプログラム	11
プログラムの構成	11
main() メソッド	13
プロパティの読み込み	15
EchoBot コンストラクタ	17
新規チャットの開設	20
プレゼンス情報の更新	21
時報の発信	24
EchoBot クラスの全ソース	26
4 次回予告	30

1 対象読者

本稿は Java のプログラミングを一通り取得し、開発環境の設定が一通りできる人を対象とします。開発に利用した Java の SDK は Version 6 です。

2 環境設定

Openfire

まず Openfire サーバをインストールし、ユーザ・アカウントを 2 つ以上作成します。一つは通常のログイン用でもう一つは EchoBot 用です。Openfire のインストールでは、小規模テスト環境と言うことで外部 DB ではなく Openfire サーバに組み込まれている DB を利用しています。設定は Web ブラウザから行うことができます。

実際に画面を見てみましょう、設定が全て終えた後、サーバを再起動してみます。Windows 環境の場合 openfire.exe というプログラムからサーバの起動、再起動、終了、管理画面 (Web ブラウザが立ちあがりま) すことができます。「Launch Admin」ボタンで管理画面を立ち上げると以下の様な画面が表示されます：

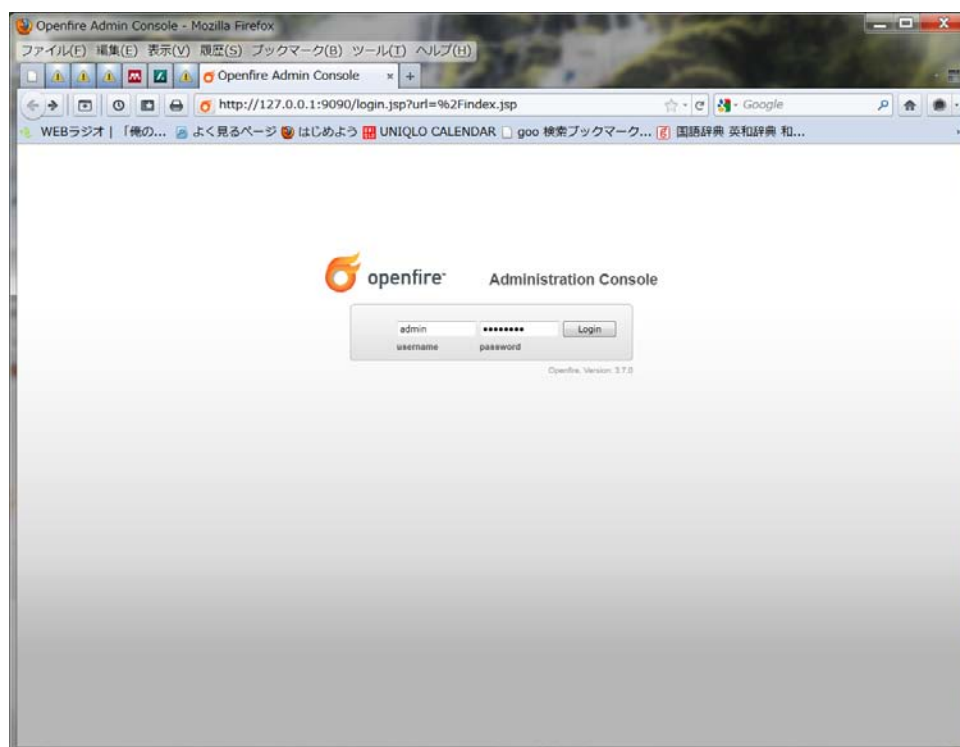


図 1 : Openfire ログイン画面

サーバのインストール時に設定した admin アカウントについてパスワードを入力してログインすると以下の様な画面が表示されます：

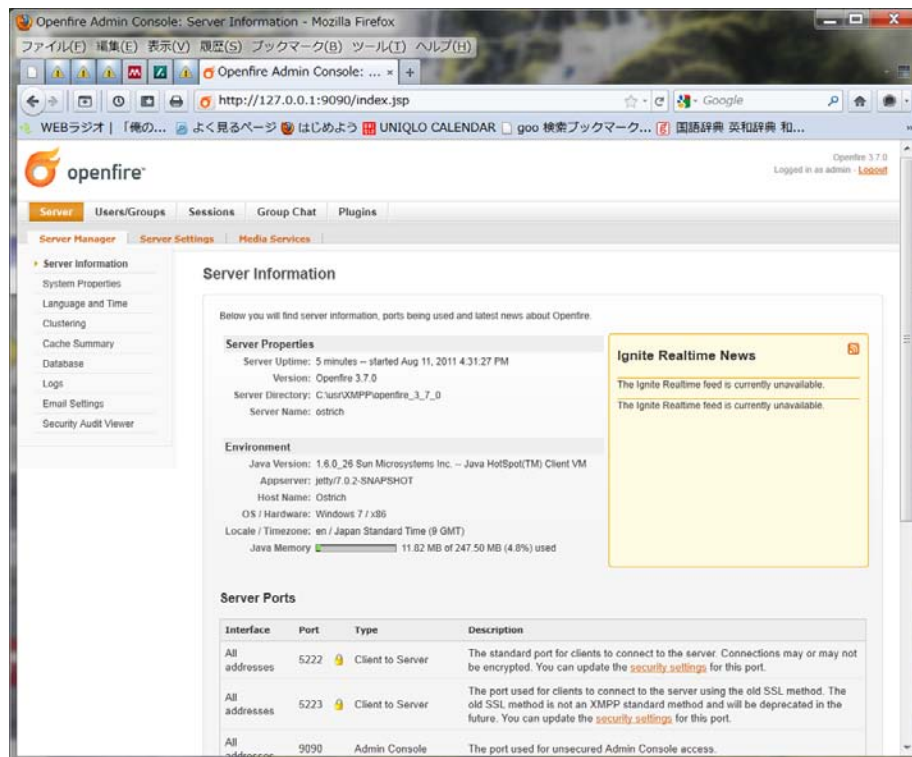


図 2 : Openfire 初期画面

ここでデータベース設定を見ることができます。左の Database という項目をクリックすると以下の様な画面が表示されます：

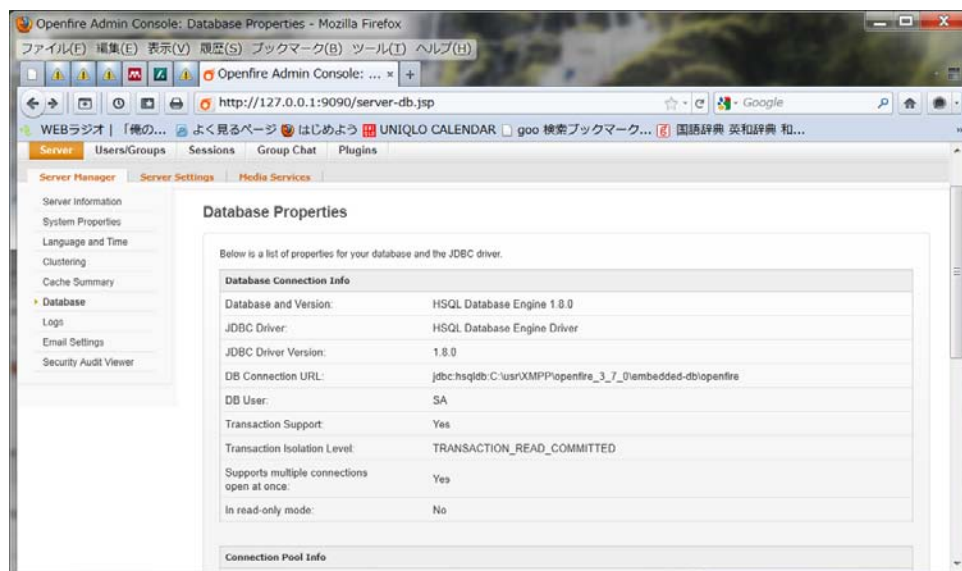


図 3 : Openfire データベース設定の確認

コネクション URL を見ると組み込み DB を利用していることが確認できます。インストール時に組み込み (Embedded) DB を選択するとパフォーマンスがよくないと警告が出ますが、プログラミングの練習用に自

分のマシンにサーバをインストールして利用する程度なら組み込み DB で問題はないでしょう。

実際にクライアント・ソフトウェアから接続するにはまずアカウントを作る必要があります。上に並んだ項目から「Users/Groups」を選んで、右の項目から「Create New User」を選ぶと以下の様な新規ユーザ作成画面が現れます：

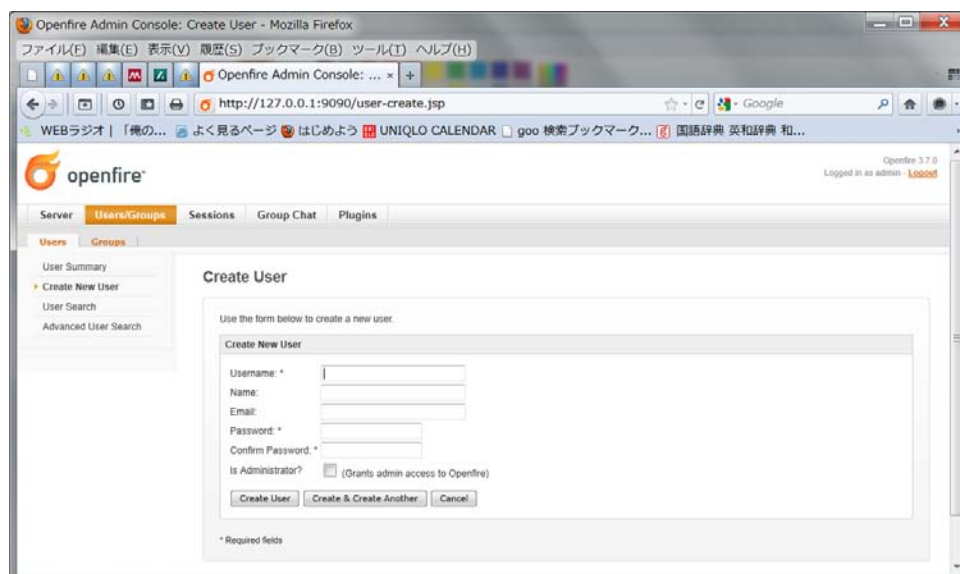


図 4：Openfire 新規ユーザ作成

ここで適当なユーザ ID とパスワードを設定してユーザを作成します。一通りユーザを作成したら作成したユーザの連絡先リスト（ロースター）を設定します。ロースターの設定はメッセージャー・クライアントから行うこともできますが、ロースターはサーバに保存されていますのでサーバの管理画面からも編集できます。プログラミングの際には作成中のプログラムがロースターの内容の操作を誤って壊すこともあり得るので、サーバ上での編集に慣れておきましょう。一通りアカウントを作成して「Users/Groups」の「User Summary」画面を見ると作成したユーザの一覧が見られます：

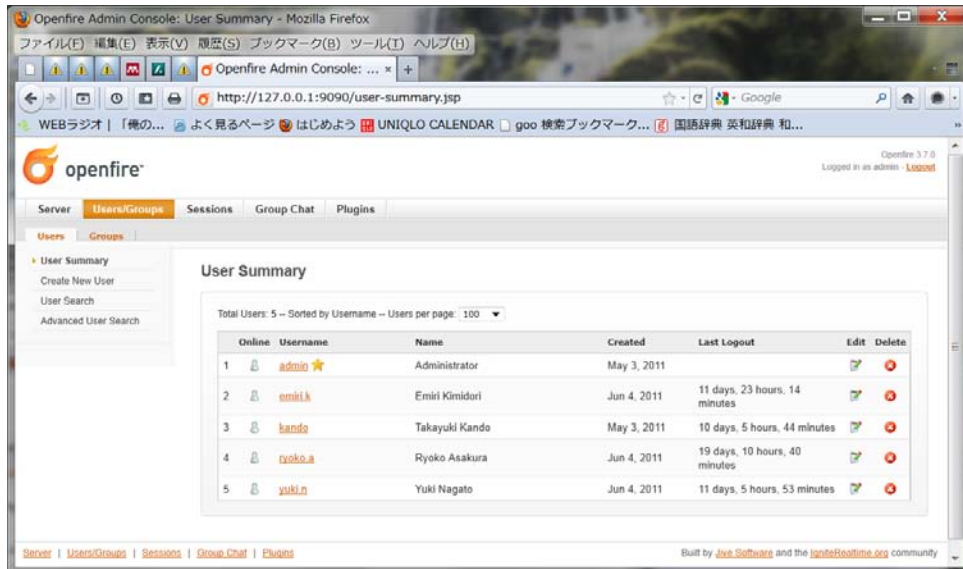


図 5 : Openfire ユーザー一覧

一覧の「Username」の列にあるユーザ名をクリックすると左側の詳細メニューに項目が増えます。「Roster」を選べると各ユーザの連絡先リスト（ロースター）を編集できます。ここでは emiri.k の連絡先を編集します。ロースターへの追加元承認制なので一方のユーザがロースターに追加しただけでは終わりませんので、各ユーザについて右側の Edit 列のアイコンをクリックして subscription を both（お互いに承認済みの意味）にしておきましょう。一通り編集してロースターに他のユーザを追加して subscription も変更し終わった状態が下の画面です：

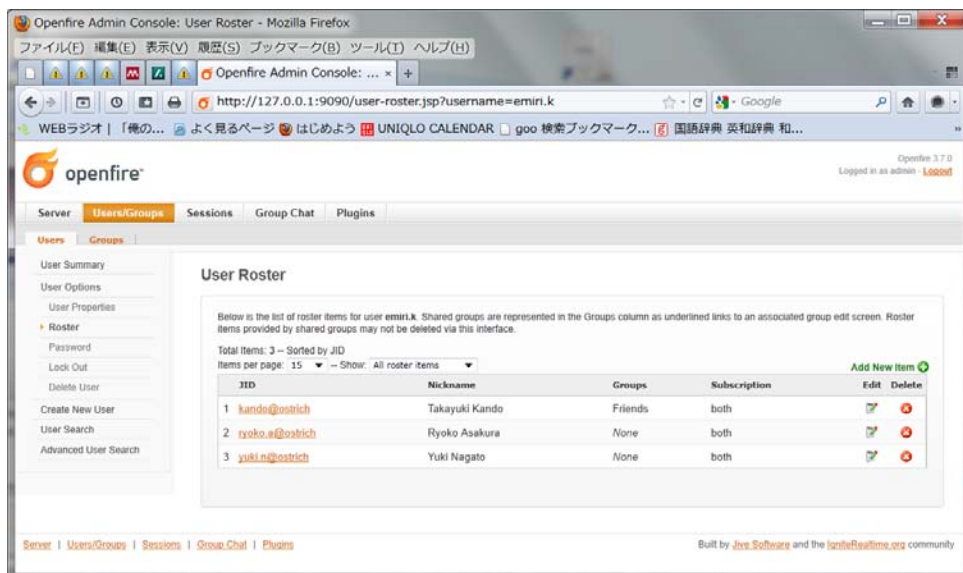


図 6 : Openfire ユーザの連絡先一覧

この状態でクライアントを 3 つ立ち上げてみましょう。Openfire の開発元が提供しているクライアントである Spark を使ってもいいのですが、Spark は Windows 環境では一つしか立ちあげられないので、ここでは

筆者が製作した Sample XMPP Client（リソース名は Sample Messenger）というクライアントを起動しています（このクライアントは現在開発中で別途解説文書を作成予定です。）。以下のように yuki.n、ryoko.a、emiri.k についてクライアントを起動、ログインすると以下のように 3 つのクライアントが起動されます：

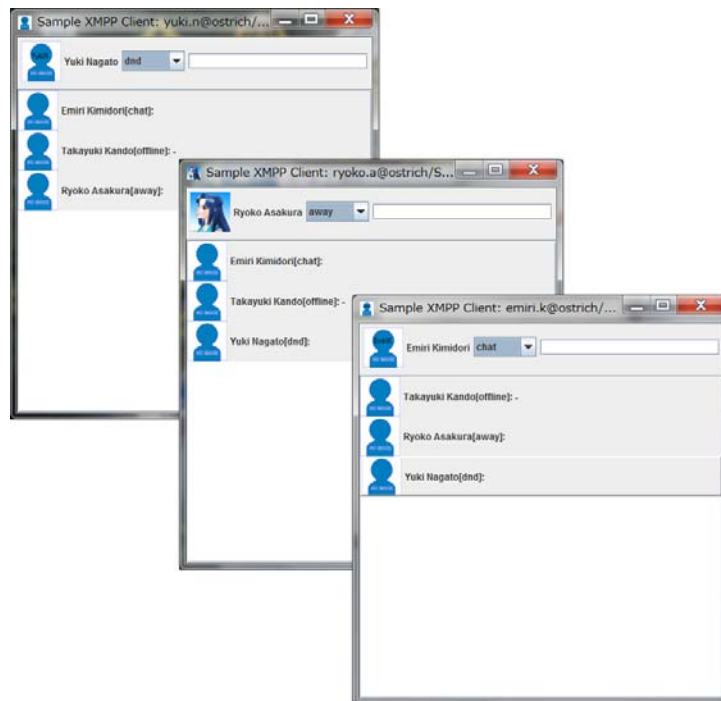


図 7：3 ユーザについて Sample Messenger を起動した様子

各クライアントのウィンドウには一番上に自分のステイタスを表示&操作する部分があり、その下には連絡先リストに登録されているユーザのリストが表示されています。yuki.n は Don't Disturb（邪魔しないで）を意味する”dnd”を、ryoko.a は離席を意味する”away”を、emiri.k はチャット歓迎を意味する”chat”を現在の mode として選択した状態になっています。それぞれのユーザ・リストも各ユーザの選択を反映した表示になっています。

Openfire の管理画面から「Sessions」の「Client Sessions」を選ぶと、以下の様な画面で各クライアントがサーバに接続している情報を見ることができます：

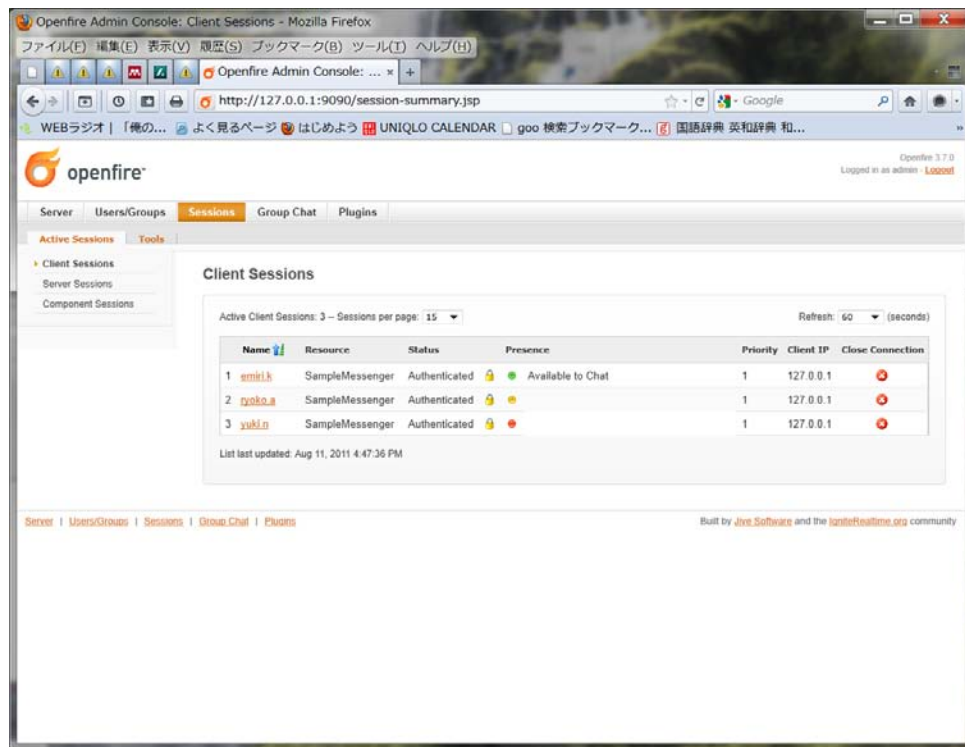


図 8 : Openfire セッション管理画面

クライアントの設定したリソース名”SampleMessenger”、接続の認証、暗号化、状態（各ユーザの mode を反映した緑、黄、赤色のアイコンで表示されている）、各接続の優先順位、各クライアントが稼働しているマシンの IP アドレス（サーバと同じマシンの上から接続しているので localhost を意味する特別な IP アドレス 127.0.0.1 になっている）などを見ることができます。一番右の○に×のアイコンはサーバ側から強制的に接続を切断する際にクリックします。右上の Refresh を設定すると表示されている状況が定期的に更新されます。

Smack

Smack ライブラリをインストールして `javac` や `java` コマンドを実行する際のクラスパス指定に追加します。Eclipse のような IDE（統合開発環境）を利用している場合は適宜設定してください。Eclipse の場合はプロジェクトのプロパティを選んで、ビルドパスに外部 jar ファイルとしてそのパスを追加します。

3 EchoBot

3-1 EchoBot の動作

EchoBot は非常に簡単な自動応答型のクライアント・プログラム（ボットと呼ばれる）で、以下のような動作をします：

- 1) プロパティ・ファイルに指定された設定でサーバに接続&ログインする
- 2) 接続すると **available** という **mode**、現在時刻を **status** にしたプレゼンス情報を一定時間（30 秒）毎に発信する
- 3) 通常のチャットで他のクライアントから話しかけられると以下の動作をする：
 - (ア) 話しかけてきたアカウントが時報発信用のリストになれば追加
 - (イ) チャットで送られてきたテキストをそのまま返す
- 4) 一定時間（5 分）毎に時報発信用のリストに登録された相手に時報メッセージを発信

以下、実際に動作を見てみましょう。ここでは **emiri.k** のユーザとして **EchoBot** をログインさせ、**ryoko.a** について先ほどの **SampleMessenger** というメッセンジャー・クライアントを立ち上げています。メッセンジャー・クライアントのロースターは以下のように見えます：

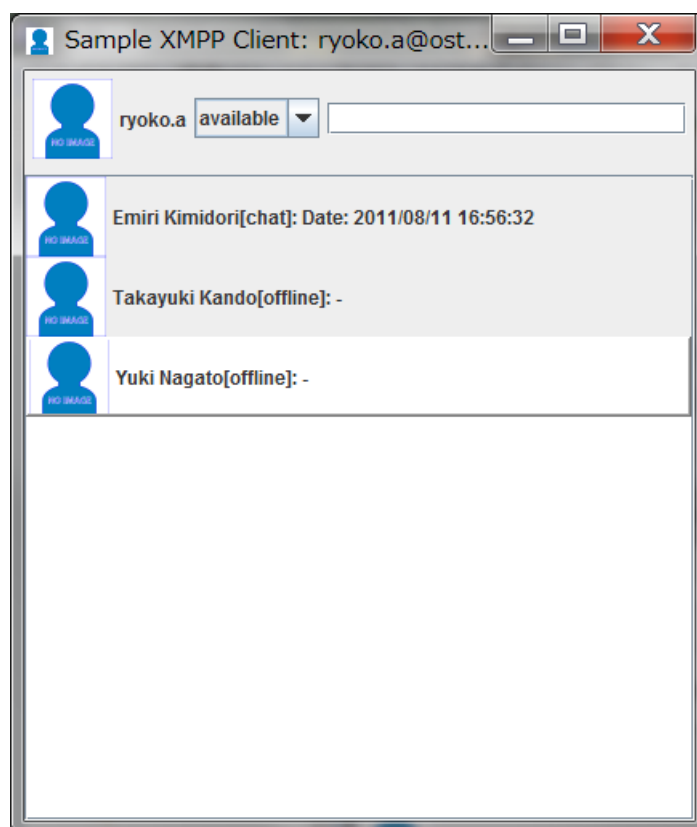


図 9：クライアントから見える **EchoBot** のステイタス

emiri.k としてログインしている **EchoBot** から送信されたプレゼンス情報を反映して、**emiri.k** の **mode** としては”chat”が、**status** としては現在の日付と時刻が表示されているのが分ります。

この時、**Openfire** の管理画面でセッションを眺めると以下のように表示されています：

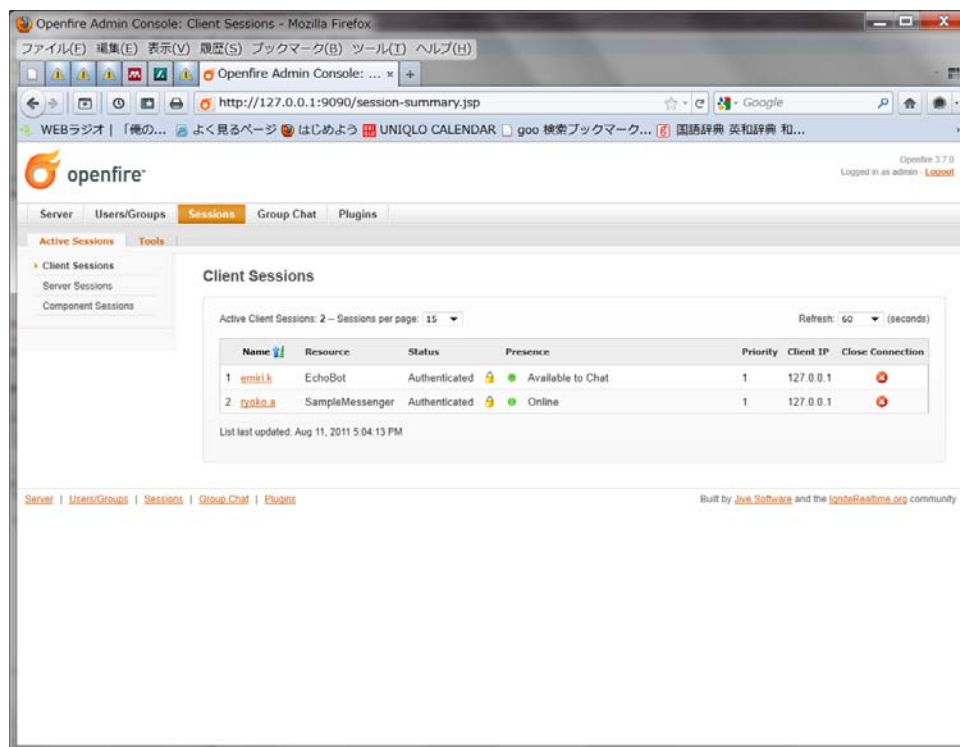


図 10 : Openfire から見る EchoBot のステイタス

EchoBot はリソース名として”EchoBot”を設定するのでそれが反映されています。また mode が”chat”になっていることもわかります。

その状態で実際に ryoko.a としてログインしているメッセージャー・クライアントから話しかけた場合のログは以下のようになります：



図 1 1 : クライアント側のチャット・ログ

チャットで ryoko.a が話しかけると同じテキストがオウム返しされています。また最初に話しかけて以降、時刻の分の末尾が 5 になる時に時報メッセージが発信されているのも分ります。

では以下の節でこのプログラムについて見ていくことにしましょう。

3-2 EchoBot のプログラム

プログラムの構成

EchoBotは非常に単純なプログラムなので、一個のトップレベル・クラスに全体がまとめられています。このトップレベル・クラスの中に定期的なプレゼンス情報の更新や時報メッセージの発信のためにTimerTaskインターフェースを実装する無名クラス⁴の定義が二つ含まれています。

⁴名前の付いていないクラス、匿名クラスとも呼ばれる。インナークラスの一種。Java ではクラスの中でクラスを入れ子に定義することができる。入れ子の「内側」に定義されたクラスは (static として宣言されていない限り)「外側」のクラスのオブジェクトの値を参照したり、メソッドを呼び出ししたりすることができる。無名クラスも同様 (無名クラスは static にはできない)。

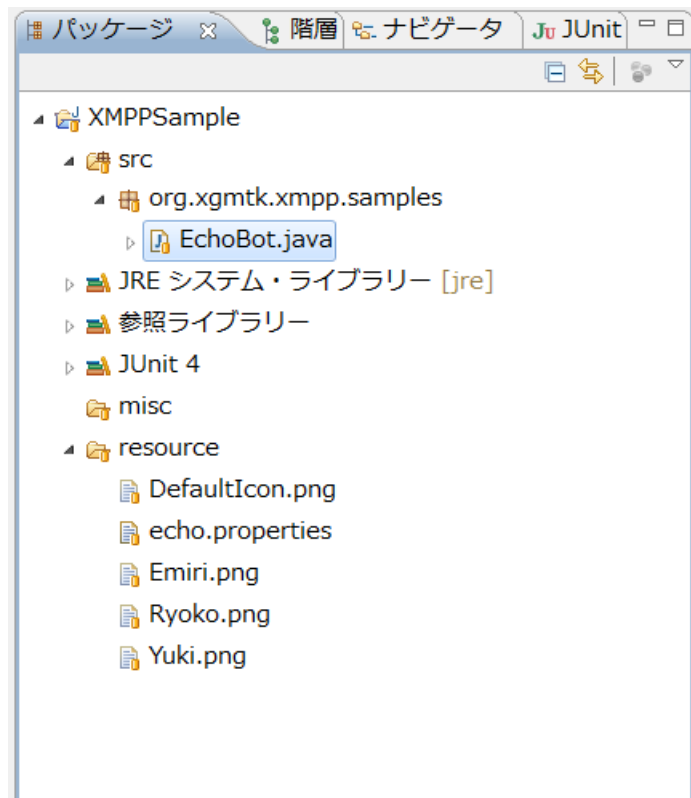


図 1 2 : EchoBot.java の位置

パッケージは `jp.or.isit.trebuchet.xmpp.samples` です。`isit.or.jp` は ISIT が保持しているドメインですので、Java におけるパッケージ名の慣習に則って `jp.or.isit.trebuchet` パッケージは ISIT の Trebuchet プロジェクトで専用に利用することができます。

下の図は Eclipse IDE で EchoBot クラスのアウトラインを表示したものです。

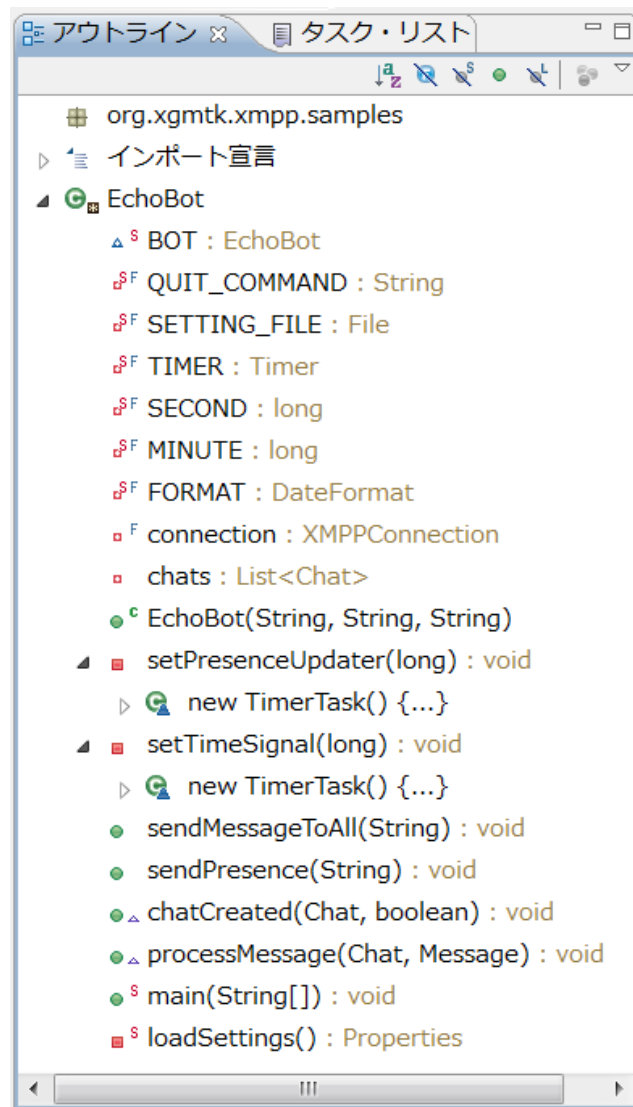


図 13 : EchoBot クラスのアウトライン

EchoBot クラスはそれ自身が Java アプリケーションとして動作するよう、アプリケーション開始の起点となる main() メソッドを含んでいます。まずはこの main() メソッドから見て行きましょう。

main() メソッド

```
public static void main(String[] args) throws Exception{
    Properties settings = loadSettings();
    String service = settings.getProperty("service");
    String id = settings.getProperty("id");
    String passwd = settings.getProperty("passwd");
```

```

BOT = new EchoBot(service, id, passwd);

System.out.println("Type ¥"+QUIT_COMMAND
    +"¥" and enter key to quit");
BufferedReader lineReader = new BufferedReader(
    new InputStreamReader(System.in));
String lineInput = lineReader.readLine();
while(lineInput != null){
    if(lineInput.equals(QUIT_COMMAND)){
        break;
    }
    lineInput = lineReader.readLine();
}
System.exit(0);
}

```

図 14 : EchoBot の main() メソッド

Java アプリケーションの実行は main() メソッドから開始されます。この main() メソッドではまず loadSettings() というメソッドでプロパティ・ファイルから情報を読み込んで Properties クラスのオブジェクトを作成しています。loadSettings() メソッドについては「プロパティの読み込み」(p.15) を参照してください。java.util.Properties クラスは Java の標準 API に含まれるクラスで、Java アプリケーションの設定情報ファイルを取り扱う目的でよく利用されます。

このプロパティ・ファイルは以下のような形式のテキストファイルです。各行の行頭から”=”の前までの文字列がキー、”=”の後ろから開業の前までの文字列がそのキーに対応する値となります。今回の例ではプロパティ・ファイルの中身は以下の様になっています：

```

service=localhost
id=emiri.k
passwd=Haruhi

```

図 15 : echo.properties

その後、service、id、passwd という String 型変数へをそれぞれ”service”、”id”、”passwd”というキーを使って Properties オブジェクトから値を取り出しています。プロパティ・ファイルの中身が図 18 のようになっているので service、id、passwd という変数の値はそれぞれ、”localhost”、”emiri.k”、”Haruhi”となります。これらはこの後 EchoBot クラスのコンストラクタの中でコネクションの接続とログインに使われますので、「環境設定」の「Openfire」(p.3) で設定したものと一致する必要があります。

`echo.properties` ファイルはサンプル・コードでは誰でも読めるような設定のファイルになっていますが、パスワードが含まれることから、実用に供するボットでは最低限、権限を適切に設定してボットと管理者以外には読み取れないようにしておくべきでしょう。

そして `main0` のもっとも重要な仕事として `EchoBot` クラスのオブジェクトを作成します。`EchoBot` クラスのコンストラクタについては「`EchoBot` コンストラクタ」(p.17) で詳細を述べますが、このコンストラクタの実行が終わると `EchoBot` は外からのチャットを受け付ける準備ができたことになります。

ここで `BOT` は `EchoBot` 型の `static` 変数です。処理が終わるまで GC されてしまわないよう念のため `static` 変数に保管しています。

その後は終了処理です。`EchoBot` オブジェクトが行う XMPP 関係の処理は `main0` メソッドとは別のスレッドで行われます。よって `main0` にはもう終了する以外にやることはないのでありますが、そのまま最終行の `System.exit(0)` に突っ込んでしまうと当然全てのスレッドが終了してしまうので困ります。かといって `System.exit(0)` をなくしてただ `main0` を終了させた場合、もし XMPP の処理をするスレッドがデーモン・スレッド⁵と設定されていた場合はやはり終了してしまいますし、そうでない場合は穏やかに終了させる簡単な方法がありません。XMPP のイベントを処理するスレッドは `XMPPConnection` クラスのオブジェクトが管理していますが、それがデーモン・スレッドであるかどうかはマニュアル等のドキュメントで仕様として明記されてはいませんので、それがデーモン・スレッドであるにせよないにせよ、そのことに依存することは避けた方がよいでしょう。

そこでここでは終わるべき時がくる前に `main0` のスレッドが終わってしまうことがないように、条件付きの無限ループをまわすこととしました。そのためまず、`System.in` オブジェクトを `BufferedReader` クラスのオブジェクト `lineReader` でラップします。これは `BufferedReader` クラスが持っている一行入力のための `readLine()` メソッドを利用するためです。ループでは `lineReader.readLine()` で一行読み込み、その結果が `"quit"` (このクラスで設定している定数 `QUIT_COMMAND` の値です。) と一致するまで無限にループが回ります⁶。ここで `"quit"` に続いて改行が入力されると無限ループが終わり、`System.exit(0);` が実行されてプログラム全体が終了します。

プロパティの読み込み

プロパティの読み込みは下に示す `loadSettings()` メソッドで行います：

```
private static Properties loadSettings()
```

⁵ デーモン・スレッドはプログラムの補助的な機能を担うスレッドに使われる機能で、スレッドを作成後、デーモン・スレッドだと設定することでスレッドはデーモン・スレッドになります。Java ではプログラムで動いているスレッドがデーモン・スレッドだけになるとプログラム全体が終了することになっています。

⁶ 入力がない場合は `lineReader.readLine()` のところで `main0` のスレッドはブロックされ、入力が一行あるまで待ちますのでこの無限ループは CPU 時間を無駄遣いはしません。


```
throws FileNotFoundException, IOException {  
  
    InputStream is = new BufferedInputStream(  
        new FileInputStream(SETTING_FILE));  
  
    Properties settings = new Properties();  
  
    settings.load(is);  
  
    is.close();  
  
    return settings;  
  
}
```

図 16 : loadSettings()メソッド

SETTING_FILE 定数が指すファイルを InputStream 型オブジェクト is を通じて読むために FileInputStream を作成し、それを BufferedInputStream でラップして is 変数を初期化しています。そして Properties 型の settings 変数にオブジェクトを作成し、settings.load()メソッドへ InputStream 型オブジェクト is を渡して読も込ませています。終わったら is.close()し、settings を返り値にしてリターンしています。

ここで、SETTING_FILE は”resource/echo.properties”ファイルを指す File 型定数です：

```
private static final File SETTING_FILE = new File("resource/echo.properties");
```

図 17 : SETTING_FILE 定数

Eclipse IDE で標準的なプロジェクト設定している場合の resource/echo.properties ファイルの位置はここです：

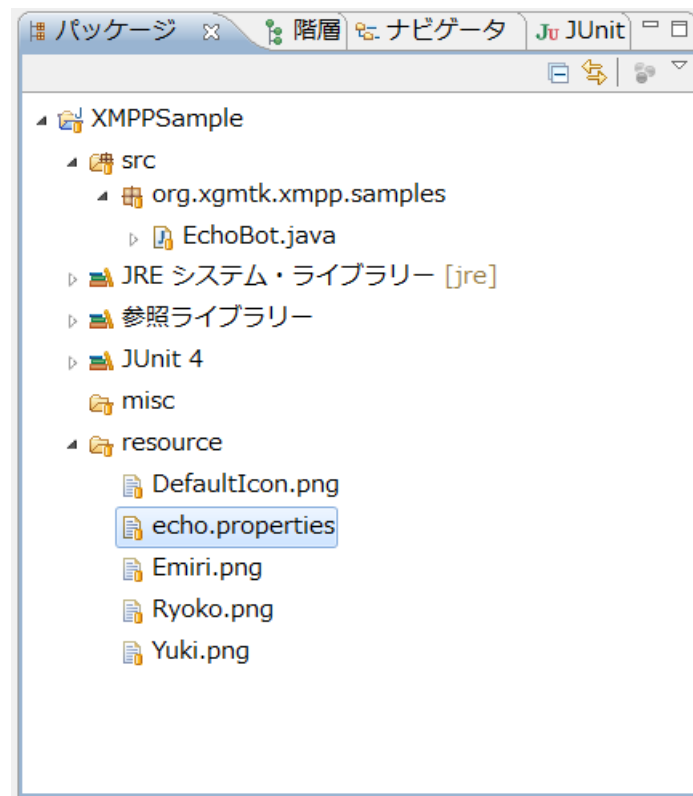


図 18 : echo.properties ファイルの位置

EchoBot コンストラクタ

EchoBot クラスのコンストラクタは EchoBot の機能を実現する中心部分です。3 つの部分に分けて説明します。

```
public EchoBot(String service, String id, String passwd)
    throws XMPPException {
    this.chats = new ArrayList<Chat>();
```

図 19 : EchoBot コンストラクタ序盤

最初に Chat オブジェクトを管理するためのリストを作成し、chats フィールドを初期化します。Chat オブジェクトはメッセージが送られてくるたびに渡されるもので、返事を返すために必要です。

ただ、オウム返しに返事を返すだけならメッセージが送られてくるたびにチャット・オブジェクトが渡されるので保管する必要はありませんが、EchoBot は話しかけてくれた人全員に自発的に時報のメッセージを送るためリストに集めて覚えておく必要があります。

```

this.connection = new XMPPConnection(service);
this.connection.connect();
this.connection.login(id, passwd,
    EchoBot.class.getSimpleName());
this.connection.getChatManager().addChatListener(this);

```

図 20 : EchoBot コンストラクタ中盤

続いて XMPP サーバに接続し、ログインします。クライアントから発信されるあらゆる XMPP パケット (XMPP 用語ではスタンザ) はまずサーバに送られてサーバが処理あるいは適切な宛先に配送します。逆にクライアントが受信するあらゆるパケットはサーバからやってきます。また一つリクエストを送ってレスポンスを得るたびに接続が切れる HTTP とは異なり、XMPP ではセッションの開始から終了までずっと一つのコネクションを利用します。このため XMPP ではサーバとの接続を表現する XMPPConnection クラスのオブジェクトがあらゆる処理の中心になります。また XMPP の接続は基本的にユーザ認証されるので、ログインが必要です。

手順は簡単で、まず XMPPConnection クラスのオブジェクトを作成します。この時に接続先のサービスを選択します。通常はサーバ名 (多くの場合は DNS で引ける名前) です。ここでは main() から渡された "localhost" になります。"localhost" は特別な名前が発信側と同一のマシンの上の XMPP サーバを指します。実際にそのマシンがどういう名前であっても同一マシン内からの接続は "localhost" を指定すれば接続できます。"localhost" で接続しても XMPPConnection の getUser() メソッドでユーザ名を取得した際のサービス名の部分は Openfire インストール時に設定した名前 (通常はサーバが稼働しているマシンの名前) になります。

次いで、作成した XMPPConnection オブジェクトの connect() メソッドを呼び出して接続します。指定したサービス名でサービスが見つからない場合はこの段階で失敗します。

そしてユーザ名とパスワード、リソース名を指定して login() メソッドを呼び出してログインします。ここでユーザ名とパスワードはそれぞれ main() から渡された "emiri.k" と "Haruhi" になります。リソース名には EchoBot クラスの単純名 "EchoBot" を渡しています。ここで直接 "EchoBot" という文字列リテラルで指定しなかった理由は EchoBot のクラス名を変更した時にリソース名を変え忘れないようにするためです。login() にはリソース名を省略した版のメソッドもありますが、その場合 "Smack_+Smack のバージョン番号" というリソース名になります。

ログインが終わったら ChatManager に自分自身を ChatManagerListener として登録します。これを可能にするために EchoBot は ChatManagerListener インターフェースをインプリメントして chatCreated() メソッドを備えています。ChatManager はある他のクライアントからメッセージが初めて送られてきたとき、ここで登録した ChatManagerListener 型オブジェクトの chatCreated() メソッドを呼び出します。chatCreated() メソッドでの処理については「最後にタイマーを二つ設定して、準備完了を標準出力に報告し

ます。一つめの `setTimeSignal()` メソッドでは時報の間隔を 5 分に、二つ目の `setPresenceUpdater()` メソッドではプレゼンス情報の更新間隔を 30 秒に設定しています。この二つのメソッドの時間指定の範囲はミリ秒です。EchoBot クラスでは読みやすくするため定数 `MINUTE` と `SECOND` 以下の様に定義しています：

```
private static final long SECOND = 1000;
private static final long MINUTE = 60 * SECOND;
```

図 25：定数 `SECOND` と `MINUTE`

`setTimeSignal()` メソッドについては「時報の発信」(p.54) で、`setPresenceUpdater()` メソッドについては「プレゼンス情報の更新」(p.51) で詳しく説明します。

新規チャットの開設」(p.19) で詳しく述べます。

`XMPPConnection` はマルチ・スレッド対応です。そして `XMPPConnection` オブジェクト自身が複数のスレッドを持ち並行動作しています。ソースをざっと眺めて確認した範囲だけでも、`XMPPConnection` は内部にサーバからのパケットを読むスレッド、サーバに向けてパケットを書きだすスレッド、そして（例えば `ChatManager` 経由で `ChatListener` を呼び出すといった形で）ユーザにイベントを伝えるメソッドという少なくとも三つのスレッドを備えています。この EchoBot のような簡単なアプリケーションではスレッドをそれほどははっきり意識することはありません（せいぜい先に述べた `main()` の終了の部分くらいです）が、GUI を備えたメッセージング・クライアントなど複雑なアプリケーションではそれを意識する必要があるでしょう。

一般に `XMPP` でメッセージを処理する機能を拡張する際はその拡張された処理を行うオブジェクトに `XMPPConnection` オブジェクトを渡して初期化する場合が多いですがチャットの開設を司る `ChatManager` や連絡先リストの管理を司る `Roster` は基本的な機能なので `XMPPConnection` オブジェクトに内蔵されており、それぞれ `getChatManager()` や `getRoster()` というメソッドでアクセスすることができます。

```
this.setTimeSignal(5*MINUTE);
this.setPresenceUpdater(30*SECOND);
System.out.println("Echo back service started. (user: ¥" +
    this.connection.getUser()+"¥")");
}
```

図 21：EchoBot コンストラクタ終盤

最後にタイマーを二つ設定して、準備完了を標準出力に報告します。一つめの `setTimeSignal()` メソッドでは時報の間隔を 5 分に、二つ目の `setPresenceUpdater()` メソッドではプレゼンス情報の更新間隔を 30 秒に設定しています⁷。この二つのメソッドの時間指定の範囲はミリ秒です。EchoBot クラスでは読みやすくするた

⁷ ここではあまり複雑にしても分りにくいと考え、実施しませんでした。これらの時間間隔も `echo.properties` ファイル

め定数MINUTEとSECOND以下の様に定義しています：

```
private static final long SECOND = 1000;
private static final long MINUTE = 60 * SECOND;
```

図 2 2 : 定数 SECOND と MINUTE

setTimeSignal()メソッドについては「時報の発信」(p.24) で、setPresenceUpdater()メソッドについては「プレゼンス情報の更新」(p.21) で詳しく説明します。

新規チャットの開設

「EchoBot コンストラクタ」(p.17) で述べたように、EchoBot クラスは ChatManagerListener インターフェースをインプリメントして ChatManager に EchoBot オブジェクト自身を渡しています。その結果他のクライアントから初めてメッセージが送られてきたタイミングで下のような chatCreated()メソッドが ChatManager によって呼び出されます：

```
@Override
public void chatCreated(Chat chat, boolean createdLocally) {
    chat.addMessageListener(this);
    this.chats.add(chat);
}
```

図 2 3 : chatCreated()メソッド

第一引数はChatオブジェクトでどこのどんなアドレスのクライアントからの新規Chatかという情報が収められています⁸。またChatオブジェクトはこのChatの一環として送られてくる実際のメッセージが到着する度に通知をしてくれるマネージャーとしての機能も備えています。

従って chatCreated()メソッドでは 2 つのを行います。1 つめは Chat オブジェクトに個々のメッセージの到着を知らせてもらうために MessageListner 型のオブジェクトを登録することです。ここでは ChatManager に EchoBot オブジェクトを登録したのと同じ方法を再び使います。このため EchoBot クラスは MessageListener インターフェースも実装します。

2 つ目は話しかけてきてくれたアカウントに時報を発信するために Chat オブジェクトを EchoBot オブジェクトが保持する Chat のリストである chats に追加することです。

に記述し、それを読み込んで設定する方がボットとしての使い勝手はよくなるでしょう。

⁸ Smack はスレッドをサポートしていて、Chat オブジェクトは実はスレッド毎に別になっています。このため実際には同一のクライアントから複数の Chat が開設される場合があります。

さて EchoBot クラスが MessageListner インターフェースを実装するには、EchoBot クラスに下の様な processMessage()メソッドを定義する必要があります：

```
@Override
public void processMessage(Chat chat, Message message) {
    String sender = chat.getParticipant();
    System.out.println("Recieved a message from: ¥"
        +sender+"¥", message text:¥"+message.getBody());
    try {
        chat.sendMessage(message.getBody());
    } catch (XMPPException e) {
        System.out.flush();
        System.err.println(
            "** Failed to send a message to ¥"
            +chat.getParticipant()+"¥".  "**");
        e.printStackTrace();
    }
}
```

図 24 : processMessage()メソッド

processMessage()スレッドは対象となる Chat にメッセージが到着する度に呼び出されます。ここでは、第一引数はメッセージが到着した Chat オブジェクトで、第二引数は到着したメッセージのパケットを表す Message オブジェクトです。

ここではまず標準出力へメッセージの受信ログを出すために Chat オブジェクトの getParticipant()メソッドで送り主のアドレスを取り出して String 型の sender 変数に設定します。そして送られてきたメッセージの本文であるテキストを取り出すために Message オブジェクトの getBody()メソッドを呼び出します。

ついでこのメソッド本来の仕事であるオウム返しを実現するため Chat オブジェクトの sendMessage()メソッドへ、送られてきた Message オブジェクトの本文（再び getBody()メソッドを使います。）を渡してメッセージを発信します。Chat オブジェクトは発信元のアドレスを知っているので、Chat オブジェクトは正しい宛先として送られてきたメッセージの送り主を設定したメッセージ・パケットをサーバに送ることができます。

sendMessage()メソッドは失敗して例外が投げられる場合がありますので、キャッチしてエラー・メッセージとスタックトレースを標準エラー出力へ表示しています。

プレゼンス情報の更新

プレゼンス情報を定期的に発信すると言った定期的な動作を実現するには Java の標準 API の Timer オブジェクトを利用します。下は EchoBot クラスにある Timer 型の static 変数 TIMER の定義です：

```
private static final Timer TIMER = new Timer();
```

図 25 : static 変数 TIMER

「EchoBot コンストラクタ」(p.17) で述べたようにプレゼンス情報を定期発信するように設定をするメソッドが下の `setPresenceUpdater()` です。

```
private void setPresenceUpdater(long interval) {
    TIMER.scheduleAtFixedRate(new TimerTask() {
        @Override
        public void run() {
            String status = "Date: "
                + FORMAT.format(Calendar.getInstance().getTime());
            sendPresence(status);
        }
    }, Calendar.getInstance().getTime(), interval);
}
```

図 26 : `setPresenceUpdater()`メソッド

先ほどの `TIMER` 変数に格納された `Timer` 型オブジェクトの `scheduleAtFixedRate()`メソッドを使って `TimerTask` 型オブジェクトを登録することで定期的なタスクが実行されます。`scheduleAtFixedRate()`メソッドの第一引数は実行すべきタスクを表す `TimerTask` 型オブジェクト、第二引数は開始時刻を表す `Date` オブジェクト、第三引数は実行間隔を表す `long` 型整数で単位はミリ秒です。

ここで第一引数には `TimerTask` インターフェースに基づく無名クラスのオブジェクトを定義&作成し、それを渡しています。こうすると時間が来たときにこの無名クラスの `run()`メソッドが呼び出されます。無名クラスはその無名クラスが含まれるクラスのフィールド、メソッド、`static` 変数、定数を利用することができます。ここでは `FORMAT` 変数に格納された日付と時刻をフォーマットする `DateFormat` オブジェクトを利用して、`run()`メソッドが実行された際の現在時刻を表す文字列を作成して `String` 型の `status` 変数に格納し、`EchoBot` オブジェクトの `sendPresence()`メソッドを呼び出してそれを渡しています。

`EchoBot` クラスの `static` 変数 `FORMAT` の定義は以下の通りです：

```
private static final DateFormat FORMAT
    = DateFormat.getDateInstance();
```

図 27 : static 変数 FORMAT

これによって `Date` 型オブジェクトから日付と時刻を表す文字列が作成できます。

`scheduleAtFixedRate()`メソッドの第二引数には現在時刻、第三引数には `EchoBot` コンストラクタから渡さ

れた値（ここでは $30 \times 60 \times 1000$ ミリ秒）を渡しています。

TimerTask オブジェクトの run() メソッドから呼び出される sendPresence() メソッドは下のようになります：

```
public void sendPresence(String status) {  
    Presence.Mode m = Presence.Mode.chat;  
    Presence.Type type = Presence.Type.available;  
    Presence p = new Presence(type, status, 1, m);  
    this.connection.sendPacket(p);  
}
```

図 28 : sendPresence() メソッド

sendPresence() メソッドはプレゼンス情報を表す Presence オブジェクトを作成して、XMPPConnection オブジェクトである EchoBot のフィールド connection について sendPacket() メソッドを呼び出し、プレゼンス・パケットをサーバに送りつけるという仕事をします。プレゼンス・パケットはサーバが適当に必要なクライアントへ配送するのであて先を指定する必要はありません。

Presence オブジェクトのコンストラクタにはユーザのサーバへの登録状況を表す Presence.Type 列挙型の値（ここではサーバ利用可能を表す available）、ユーザのクライアント利用状況を示す Presence.Mode 列挙型の値（ここではチャット歓迎を表す chat）、接続クライアントの優先順位（1、よく使われるデフォルト値）、ユーザが設定した短いテキストである status（ここでは TimerTask から渡された現在時刻を表す文字列）を渡しています。

時報の発信

「EchoBot コンストラクタ」(p.17) で述べたように時報を定期発信するように設定をするメソッドが下の `setTimeSignal()` です:

```
private void setTimeSignal(long interval) {
    Calendar local_calendar = Calendar.getInstance();
    local_calendar.set(Calendar.MINUTE, 0);
    local_calendar.set(Calendar.SECOND, 0);
    local_calendar.set(Calendar.MILLISECOND, 0);
    Date startTime = local_calendar.getTime();
    TIMER.scheduleAtFixedRate(new TimerTask(){
        @Override
        public void run() {
            String message
                = "* Time signal: "
                +FORMAT.format(Calendar.getInstance().getTime());
            sendMessageToAll(message);
        }
    }, startTime, interval);
    System.out.println(
        "Start time signal(interval: "
        +((double)interval / MINUTE)+" minutes) at : "
        +FORMAT.format(startTime));
}
```

図 29 : setTimeSignal()メソッド

`setTimeSignal()`の動作は `TIMER` や `FORMAT`、`TimerTask` 型の無名クラスの使い方に関して言えば「プレゼンス情報の更新」(p.21) の `setPresenceUpdater()`メソッドとほとんど同じです。違いは3つだけです。

1 つめは `Presence` の定期更新は開始時刻をあまり気にせず現在時刻にしていますが、時報はきっちりした時間にしたいので開始時刻を、現在時刻の分、秒、ミリ秒を0に設定した時刻に変えていることです。例えば11時48分25秒と225ミリ秒が現在時刻であったとすると、11時ジャストが開始時刻になります。このようにして `scheduleAtFixedRate()`メソッドを使うと11時50分から以降5分毎（この例では `interval` が5分に設定されているので）に `TimerTask` オブジェクトの `run()`メソッドが起動されるようになります。

2 つめは `TimerTask` 型のオブジェクトの `run()`メソッドで呼び出される `EchoBot` のメソッドが `sendPresence()`ではなくて `sendMessageToAll()`メソッドであることです、これは目的がプレゼンス情報の更新でなくてこれまでに話しかけてきた全員に時報のメッセージを送ることなので当然だと言えます。

3つめは些細なことですが、ログとしてタイマー設定終了後に開始時刻を標準出力に出力していることです。

時報を実際に送るために呼び出される EchoBot クラスのメソッド `sendMessageToAll()` は下のようになります：

```
public void sendMessageToAll(String message) {
    for(Chat c : this.chats){
        try {
            c.sendMessage(message);
        } catch (XMPPException e) {
            System.out.flush();
            System.err.println(
                "*** Failed to send a message to ¥" +
                c.getParticipant()+"¥".  ***");
            e.printStackTrace();
        }
    }
}
```

図 30 : `sendMessageToAll()`メソッド

既に「」(p.) で説明した方法により、これまでに開設された全ての Chat オブジェクトが `chats` フィールドのリストに記録されているのでこれを利用します。即ち `chats` の全要素をイテレートしてそれら Chat オブジェクトの `sendMessage()`メソッドを呼び出すことで、全員に向かって `TimaerTask` の `run()`メソッドから渡されたメッセージ（つまり現在時刻を表す文字列）を送信します。

`sendMessage()`メソッドは失敗して例外が投げられる場合がありますので、キャッチしてエラー・メッセージとスタックトレースを標準エラー出力へ表示して、引き続き残りの Chat オブジェクトに対する送信を行います。

EchoBot クラスの全ソース

最後に EchoBot クラスの全ソース (EchoBot.java) を載せておきます。

```
package jp.or.isit.trebuchet.xmpp.samples;

import java.io.*;
import java.text.DateFormat;
import java.util.*;

import org.jivesoftware.smack.*;

public class EchoBot implements ChatManagerListener, MessageListener{
    static EchoBot BOT;
    private static final String QUIT_COMMAND = "quit";
    private static final File SETTING_FILE
        = new File("resource/echo.properties");
    private static final Timer TIMER = new Timer();
    private static final long SECOND = 1000;
    private static final long MINUTE = 60 * SECOND;
    private static final DateFormat FORMAT
        = DateFormat.getDateTimeInstance();

    private final XMPPConnection connection;
    private List<Chat> chats;

    public EchoBot(String service, String id, String passwd)
        throws XMPPException {
        this.chats = new ArrayList<Chat>();

        this.connection = new XMPPConnection(service);
        this.connection.connect();
        this.connection.login(id, passwd,
            EchoBot.class.getSimpleName());
        this.connection.getChatManager().addChatListener(this);

        this.setTimeSignal(5*MINUTE);
    }
}
```

```

        this.setPresenceUpdater(30*SECOND);
        System.out.println("Echo back service started. (user: ¥" +
            this.connection.getUser()+"¥")");
    }

    private void setPresenceUpdater(long interval) {
        TIMER.scheduleAtFixedRate(new TimerTask() {
            @Override
            public void run() {
                String status = "Date: "
                    +FORMAT.format(Calendar.getInstance().getTime());
                sendPresence(status);
            }
        }, Calendar.getInstance().getTime(), interval);
    }

    private void setTimeSignal(long interval) {
        Calendar local_calendar = Calendar.getInstance();
        local_calendar.set(Calendar.MINUTE, 0);
        local_calendar.set(Calendar.SECOND, 0);
        local_calendar.set(Calendar.MILLISECOND, 0);
        Date startTime = local_calendar.getTime();
        TIMER.scheduleAtFixedRate(new TimerTask() {
            @Override
            public void run() {
                String message
                    = "* Time signal: "
                    +FORMAT.format(Calendar.getInstance().getTime());
                sendMessageToAll(message);
            }
        }, startTime, interval);
        System.out.println(
            "Start time signal(interval: "
            +((double)interval / MINUTE)+" minutes) at : "
            +FORMAT.format(startTime));
    }
}

```

```

public void sendMessageToAll(String message) {
    for(Chat c : this.chats){
        try {
            c.sendMessage(message);
        } catch (XMPPException e) {
            System.out.flush();
            System.err.println(
                "*** Failed to send a message to ¥"
                +c.getParticipant()+"¥".  ***");
            e.printStackTrace();
        }
    }
}

public void sendPresence(String status) {
    Presence.Mode m = Presence.Mode.chat;
    Presence.Type type = Presence.Type.available;
    Presence p = new Presence(type, status, 1, m);
    this.connection.sendPacket(p);
}

@Override
public void chatCreated(Chat chat, boolean createdLocally) {
    chat.addMessageListener(this);
    this.chats.add(chat);
}

@Override
public void processMessage(Chat chat, Message message) {
    String sender = chat.getParticipant();
    System.out.println("Recieved a message from: ¥"
        +sender+"¥", message text:¥"+message.getBody());
    try {
        chat.sendMessage(message.getBody());
    } catch (XMPPException e) {

```

```

        System.out.flush();
        System.err.println(
            "*** Failed to send a message to ¥"
            +chat.getParticipant()+"¥".  ***");
        e.printStackTrace();
    }
}

public static void main(String[] args) throws Exception{
    Properties settings = loadSettings();
    String service = settings.getProperty("service");
    String id = settings.getProperty("id");
    String passwd = settings.getProperty("passwd");

    BOT = new EchoBot(service, id, passwd);

    System.out.println("Type ¥"+QUIT_COMMAND
        +"¥" and enter key to quit");
    BufferedReader lineReader = new BufferedReader(
        new InputStreamReader(System.in));
    String lineInput = lineReader.readLine();
    while(lineInput != null){
        if(lineInput.equals(QUIT_COMMAND)){
            break;
        }
        lineInput = lineReader.readLine();
    }
    System.exit(0);
}

private static Properties loadSettings()
    throws FileNotFoundException, IOException {
    InputStream is = new BufferedInputStream(
        new FileInputStream(SETTING_FILE));
    Properties settings = new Properties();
    settings.load(is);
}

```



```
        is.close();  
        return settings;  
    }  
}
```

図 3 1 : EchoBot.java

4 予告

次回はメッセージアプリケーションらしい GUI を備え実際にメッセージをやり取りできるクライアントをサンプル・コードについて解説します。規模が大きいのですが、前半では特に GUI とメッセージのやり取りではスレッドを意識する必要があるのでそのあたりを詳しく解説する予定です。後半ではユーザを表すアイコンであるユーザ・アバター (User Avatar) を例に Pub/Sub プロトコルの簡易版でユーザ・アバターを始めリッチ・プレゼンスの送受信によく利用されている PEP (Personal Eventing Protocol) の実装について紹介する予定です (Trebuchet では仮想マシンのパフォーマンス情報や資源利用情報の発信に利用する予定です)。