Xone 開発ガイド

Xone Personal Version1.0.4

(有)ファイバーワークス

■はじめに

この開発ガイドでは、Xone をインストールしたフォルダを XONE_HOME として説明します。実際にインストールしたフォルダに適宜読みかえてください。XONE_HOME は環境変数として定義する必要はありません。

目次

1 Xoneモデル

Xoneのモデルは、もともとAIなどで使われていた素性構造がベースになっています。素性構造では、個々の素性（属性）を名前・型・値で表します。その素性を複数組み合わせることで、オブジェクトを定義します。たとえば、

製品A
　　型番、string、xxxxx
　　価格、int、10000

は、2つの素性を組み合わせて製品Aを表現したものです。これは、C言語の構造体にも良く似ています。Xoneモデルでも、これと同様に複数の素性を組み合わせて、オブジェクトを構成します。Xoneでは個々の素性をエレメント（Element）と呼んでいます。エレメントは名前がユニークであれば、1つのオブジェクトにいくつあってもかまいません。
このエレメントの他に、エレメントリストというものも含めることができます。エレメントリストにも名前が付けられており、これもユニークな名前であれば、いくつあってもかまいません。個々のエレメントリストには、複数のエレメントを格納できます。また、エレメントリストはその名前のとおり、リストですから、この中に入れるエレメントは名前では同じでもかまいません。

例として、簡単な注文データを考えてみます。1つの注文には、複数の製品を含めることができるものとします。

注文A
　　顧客名、string、○野△男
　　注文日、date、2004-4-12
　　送付先、string、東京都調布市XX町1-1-1
　　list:注文内容
　　　　型番、string、xxxxx
　　　　個数、int、1
　　　　型番、string、yyyyy
　　　　個数、int、2
　　　　型番、string、zzzzz
　　　　個数、int、3

この例では、「注文内容」という名前のエレメントリストで、xxxxx、yyyyy、zzzzzという製品をそれぞれ1、2、3個注文したことを示しています。エレメントリストも名前がユニークであれば、いくつでも含めることができるので、たとえば「送付先リスト」として複数の送付先を入れることもできます。

以上がXoneモデルの骨子であり、シンプルなモデルです。

1.1 エレメント

エレメントは、名前・型・値で構成されます。値は省略（nullとなる）できますが、名前と型は省略できません。エレメントの名前は**Java**の識別子と同じ命名規則（※）にしたがいます。先頭を数字にしたり、アンダースコアやドル記号以外の記号を使うことはできませんが、アルファベットだけではなく日本語ももちろん使えます。

※命名規則　エレメント名だけではなく、後ほど説明する型名、オブジェクト名、フォルダ名も同じ命名規則に従います。Javaの識別子（変数名やメソッド名など）と同じ命名規則です。ですから、先頭の文字を数字にしたり、_や$以外の記号を使うことはできません。日本語を使うこともできます。大文字と小文字は区別されます。

・プリミティブ型
型には、デフォルトで用意されている型（プリミティブな型）として、14種類とその配列の14種類で

合計 28 種類あります。これらの型は、com.fiverworks.xone.model.XoneModel に定数として定義されています。また、それぞれ次のように Java のクラスに対応します。

プリミティブ型	定義されている定数	**Java** クラスとの対応
boolean	XoneModel.BOOLEAN	java.lang.Boolean
byte	XoneModel.BYTE	java.lang.Byte
short	XoneModel.SHORT	java.lang.Short
int	XoneModel.INT	java.lang.Integer
long	XoneModel.LONG	java.lang.Long
float	XoneModel.FLOAT	java.lang.Float
double	XoneModel.DOUBLE	java.lang.Double
integer	XoneModel.INTEGER	java.math.BigInteger
decimal	XoneModel.DECIMAL	java.math.BigDecimal
string	XoneModel.STRING	java.lang.String
time	XoneModel.TIME	java.sql.Time
date	XoneModel.DATE	java.sql.Date
dateTime timestamp	XoneModel.DATE_TIME XoneModel.TIMESTAMP	java.sql.Timestamp

この プリミティブ型の型名は XMLSchema に合わせたものですが、Java でも SQL でも日付・時間は timestamp と表すことがあるので、DATE_TIME と TIMESTAMP のどちらでも同じものとしています。それぞれの型が取りうる値は、対応する Java クラスと同じです。DATE は、java.util.Date ではなく、java.sql.Date であることに注意してください。また、プリミティブ型の INT と INTEGER は、それぞれ java.lang.Integer と java.math.BigInteger に対応します。

プリミティブ型には、すべてその配列があります。配列は、各型に_ARRAY を付けたものです（例: XoneModel.INT_ARRAY）。配列は内部では[]を付けて表現されます（XoneModel.INT_ARRAY は int[]）。

・ユーザ定義型
プリミティブ型以外の型もユーザが自由に定義できます。プリミティブ型については値がその型の範囲外であればエラーになりますが、プリミティブ型以外の型についてはチェックされません。また、型名は、命名規則に従う必要があります。型名の後に"[]"を付けると、配列と見なします。

・正しいユーザ定義型の例
"point"
"matrix33[]"
"容積"

・間違った例
"3D" （先頭が数字なので命名規則違反）
"x-y" （-記号は使えない。命名規則違反）

なお、"[]"を付けて配列とすることはできますが、これはあくまで1次元のみで、"[][]"で2次元配列にするといったことはできません（エラーになります）。

ユーザ定義型として、クラス名を指定することもできます。これについては後ほど説明します。

1.2 Xone オブジェクト

エレメントとエレメントリストを複数組み合わせて、ひとつのオブジェクトを構成します。これを Xone オブジェクトと呼びます。Xone オブジェクトには、クラス、インスタンス、Any インスタンスの3種類あります。このうちクラスとインスタンスはオブジェクト指向と同様に、クラスはテンプレートのようなものであり、インスタンスはその実体のことです。Any インスタンスは、クラスのないインスタンスのことです。
どの Xone オブジェクトも、エレメントとエレメントリストの組み合わせであることは変わりません。それぞれの違いは次のとおりです。

・クラス
エレメントとエレメントリストを定義しておくもの。
エレメントに値を持つと、インスタンスを生成するときはデフォルトの値となる。
個々のエレメントに制約条件を持つことができる。
エレメントリストの中のエレメントはインスタンスにもそのまま入いる。

・インスタンス
クラスを元にして作られる。クラスで定義されているエレメントやエレメントリストに対してのみアクセスでき、それ自体を追加・削除することはできない。
クラスで定義されているエレメントリストには自由にエレメントを追加・削除できる。

・**Any** インスタンス
クラスは必要ない。エレメントやエレメントリストは自由に追加・削除できる。

以上のように Xone でのクラス、インスタンス、Any インスタンスの関係は、XMLでのスキーマ、XMLインスタンス、整形式の関係とよく似ています。

2 最初のサンプルプログラム

ここで、Xone モデルを使った簡単なサンプルプログラムを紹介します。サンプルプログラムはすべて、XONE_HOME の samples フォルダに格納されています。最初はデータベースとのデータのやりとりはないので、インストールガイドにしたがって CL（クライアント層）のクラスパスの設定をしてください。
Xone モデルは、com.fiverworks.xone.model パッケージにすべて集約されています。この中の、主なクラスの役割は次のとおりです。

クラス名	役割
XoneModel	Xone モデルの中心的なクラスで、これを使ってインスタンスの生成や型のチェックなどを行う。
XoneObject	Xone のクラスやインスタンスのスーパークラス
XoneClass	Xone のクラス
XoneInstance	Xone のインスタンス
XoneAnyInstance	Xone の Any インスタンス
XoneElement	Xone のエレメント
XoneElementList	Xone のエレメントリスト

次のプログラムは「1　Xone モデル」で示した例をプログラムにしたものです。

・プログラム１

```
package samples.model;
import com.fiverworks.xone.model.*;
public class Sample1 {
  public static void main(String[] args) {
    XoneAnyInstance xai = XoneModel.newXoneAnyInstance("製品", "製品 A");
    xai.addElement(new XoneElement("型番", XoneModel.STRING, "xxxxx"));
    // 上の行は簡略化して以下のようにも書ける
    // xai.addElement("型番", XoneModel.STRING, "xxxxx");
    xai.addElement(new XoneElement("価格", XoneModel.INT, "10000"));
    System.out.println(xai);
  }
}
```

このプログラムでは Any インスタンスを作っています。Any インスタンスは、

XoneModel.newXoneAnyInstance(クラス名, インスタンス名)

で作ります。Any インスタンスには「クラスは必要ない」と書きましたが、このクラス名はあくまでも識別のための名前だけで、実際のクラスは必要ありません。クラス名、インスタンス名はそれぞれ命名規則に従う必要があります。
Any インスタンスはエレメントを自由に追加・削除（ただし、名前がユニークであること）できるので、ここでは2つのエレメントを追加しています。エレメントのプリミティブ型は、XoneModel に定義されています。エレメントの値は文字列で指定します。この値が、その型に合わなかったり、範囲外の場合は実行時例外が投げられます。

コメントにもあるように、
xai.addElement(new XoneElement("型番", XoneModel.STRING, "xxxxx"));
という文は、
xai.addElement("型番", XoneModel.STRING, "xxxxx");
と簡略して書くこともできます。
同じ名前のエレメントを追加した場合、後から追加したもので上書きされます。

プログラム 1 の実行結果は以下のようになります。

```
class:製品
type:XoneAnyInstance
name:製品 A
path:製品 A
hint:null
  #型番,string,xxxxx
  #価格,int,10000
```

実行結果の「class:」にはクラス名、「type:」 にはオブジェクトのタイプ（XoneClass、XoneInstance、XoneAnyInstance のいずれか）、「name:」にはオブジェクト名が表示されています。この他、「path:」や「hint:」は後ほど説明します。#が先頭に付いているものは、エレメントを表しています。それぞれ、カンマで区切られて、エレメントの名前・型・値が表示されています。
次のプログラムは「1　Xone モデル」で示したもうひとつの例をプログラムにしたものです。

・プログラム２

```
package samples.model;
import com.fiverworks.xone.model.*;
public class Sample2 {
	public static void main(String[] args) {
		XoneAnyInstance xai = XoneModel.newXoneAnyInstance("注文", "注文 A");
		xai.addElement("顧客名", XoneModel.STRING, "xxxxx");
		xai.addElement("注文日", XoneModel.DATE, "2004-4-12");
		xai.addElement("送付先", XoneModel.STRING, "東京都調布市 XX 町 1-1");

		XoneElementList xel = xai.addElementList("注文内容");
		xel.add("型番", XoneModel.STRING, "xxxxx");
		xel.add("個数", XoneModel.INT, "1");
		xel.add("型番", XoneModel.STRING, "yyyyy");
		xel.add("個数", XoneModel.INT, "2");
		xel.add("型番", XoneModel.STRING, "zzzzz");
		xel.add("個数", XoneModel.INT, "3");

		System.out.println(xai);
	}
}
```

このプログラムでも Any インスタンスを作っています。Any インスタンスには、名前がユニークであればいくつでもエレメントリストを追加することができます。ここでは"注文内容"というエレメントリストを追加しています。エレメントリストの中にはいくつでもエレメントを追加できます(リストなので、エレメントの名前は同じでもかまいません)。
注文日という DATE 型のエレメントを追加していますが、DATE、TIME、DATE_TIME (TIMESTAMP)のフォーマットは次のとおりです。

型名	フォーマット
DATE	yyyy-MM-dddd 例:2000-04-08(2000-4-8 でもよい) ただし、存在しない日付(例えば 2000-2-31 など)はエラー
TIME	HH:mm:ss 例:15:18:40 ただし、範囲外の値(例えば 15:62:18 など)はエラー
DATE_TIME TIMESTAMP	yyyy-MM-dd HH:mm:ss 例:2000-4-8 15:18:40

なお、yyyy、MM などのフォーマット指定子については java.text.SimpleDateFormat のドキュメントを参照してください。また、これらのフォーマットはそれぞれ java.sql.Date、java.sql.Time、java.sql.Timestamp の toString()で得られる形式と同です。

・配列
プログラム3は配列を使ったサンプルプログラムです。

・プログラム３

```
package samples.model;

import com.fiverworks.xone.model.*;

public class Sample3 {
	public static void main(String[] args) {
		XoneAnyInstance xai = XoneModel.newXoneAnyInstance("LINE3D", "lineA");
		xai.addElement("start", XoneModel.FLOAT_ARRAY,
				new String[]{"0.0f","0.0f", "0.0f"});
		xai.addElement("end", XoneModel.FLOAT_ARRAY,
				new String[]{"1.0", "2.0", "3.0"});
		System.out.println(xai);
	}
}
```

配列の場合、要素中に null があるときはエラーになります。配列は CSV 形式の文字列として格納されます。

・ユーザ定義型
プログラム4はユーザ定義型を使ったサンプルプログラムです。

・プログラム４

```
package samples.model;
import com.fiverworks.xone.model.*;
public class Sample4 {
	public static void main(String[] args) {
		XoneAnyInstance xai = XoneModel.newXoneAnyInstance("LINE3D", "lineA");
		xai.addElement("start", "point3d", "1.0, 2.0, 3.0");
		xai.addElement("end", "point3d[]", new String[]{"1.0", "2.0", "3.0"});
		// 配列ではないのに配列の値を指定するとエラーになります（次のような文）。
		// xai.addElement("end", "point3d", new String[]{"1.0", "2.0", "3.0"});
		System.out.println(xai);
	}
}
```

ユーザ定義型の値は、文字列で与えなければなりません。また、"[]"を付けて配列とすることもできます。配列のときは、文字列配列で値を設定します。

2.1 クラスとインスタンス

プログラム5はクラスとインスタンスを使ったサンプルプログラムです。クラスを定義し、そのクラスからインスタンスを生成します。

・プログラム５

```
package samples.model;
import com.fiverworks.xone.model.*;
public class Sample5 {
	public static void main(String[] args) {
		XoneElement xe1 = new XoneElement("顧客名", XoneModel.STRING);
		XoneElement xe2 = new XoneElement("注文日", XoneModel.DATE, "2004-4-12");
		XoneElement xe3 = new XoneElement("送付先", XoneModel.STRING);
		XoneElementList xel = new XoneElementList("注文内容");
		xel.add("デフォルトの要素", XoneModel.STRING, "");
		XoneClass xc = XoneModel.newXoneClass("注文",
					new XoneElement[]{xe1, xe2, xe3},
					new XoneElementList[]{xel});

		XoneInstance xi = XoneModel.newXoneInstance(xc, "注文A");
		xi.setElementValue("顧客名", "xxxxx");
		xi.setElementValue("送付先", "東京都調布市XX町1-1-1");
		// クラスに定義されていないエレメントに値を設定するとエラー（次のような文）
		// xi.setElementValue("未定義", "abc");

		XoneElementList xel1 = xi.getElementList("注文内容");
		xel1.add("型番", XoneModel.STRING, "xxxxx");
		xel1.add("個数", XoneModel.INT, "1");
		xel1.add("型番", XoneModel.STRING, "yyyyy");
		xel1.add("個数", XoneModel.INT, "2");

		System.out.println(xc);
		System.out.println("-----");
		System.out.println(xi);
	}
}
```

プログラム5の出力結果は次のとおりです。

```
class:注文
type:XoneClass
name:注文
path:注文
hint:null
  #顧客名,string,null
  #注文日,date,2004-4-12
  #送付先,string,null
 list:注文内容
  #デフォルトの要素,string,
 list:_restrictionList
 list:_restrictionMessageList
-----
class:注文
type:XoneInstance
name:注文 A
path:注文 A
hint:null
  #顧客名,string,xxxxx
  #注文日,date,2004-4-12
  #送付先,string,東京都調布市 XX 町 1-1-1
 list:注文内容
  #デフォルトの要素,string,
  #型番,string,xxxxx
  #個数,int,1
  #型番,string,yyyyy
  #個数,int,2
```

クラスとインスタンスには、以下のような関係があります。

・インスタンスでは、クラスで定義されているエレメントにのみ値を設定できる。
・インスタンスでは、クラスに定義されているエレメントを削除したり、新たにエレメントを追加することはできない。
・クラスで値を指定したエレメントは、インスタンスのデフォルト値になる
・クラスでエレメントリストに追加したエレメントは、インスタンスにも入る
・クラスには、_restrictionList と_restrictionMessageList というエレメントリストが自動的に追加される（これらは制約に関連するもので、後ほど解説します）

2.2 エレメント

プログラム5では、オブジェクトが保持しているエレメントに値を設定するために、

```
xi.setElementValue("顧客名", "xxxxx");
```

のような文を実行しています。これでもよいのですが、オブジェクトからではなくエレメントを取得してからの方がエレメントの充実したメソッドを利用できます。
エレメントの値は内部では文字列として保持されます。ただし、プリミティブ型の場合は、その型に合わせた値を取得することができます。次のプログラムはエレメントの値の取得方法を示したものです。

・プログラム６

```
package samples.model;
import com.fiverworks.xone.model.*;
public class Sample6 {
	public static void main(String[] args) {
		XoneAnyInstance xai = XoneModel.newXoneAnyInstance("製品", "製品A");
		xai.addElement("価格", XoneModel.INT, "10000");

		String p1 = xai.getElementValue("価格");
		Integer p2 = (Integer)xai.getElementValueAsType("価格");

		XoneElement xe = xai.getElement("価格");
		String p3 = xe.getValue();
		Integer p4 = (Integer)xe.getValueAsType();
		int p5 = xe.getValueAsInt();
	}
}
```

ユーザ定義型の場合、エレメントの値は文字列だけなので、エレメントの
getValueAsObject()
getValueAsString()
getValue()
では、どれもString型で同じ値を取得します。

・エレメントの値について
エレメントの値は文字列として格納されますが、値がnullの場合、内部的には"~null;"と表現されます。また、セパレータとして"~ls;"という文字列も使います。そのため、これらとまったく同じシーケンスの文字列を値の中に含めないようにしてください。

3 フォルダとオブジェクト

Xone は、Xone のオブジェクトを RDB で管理する仕組みを提供しています。オブジェクトはツリー状になったフォルダで管理されます。これは OS においてファイルがフォルダで管理されるのと同様のものです。これには以下のようなルールがあります。

・フォルダにはいくつでも子のフォルダを作成できる
・子のフォルダの名前はユニークでなければならない
・フォルダにはいくつでもオブジェクトを格納できる
・1つのフォルダ内のオブジェクトの名前はユニークでなければならない
・フォルダやオブジェクトの名前は、前述した命名規則に従う必要がある（名前を数字で始めたり、スペースなどの記号を使うことはできない）

上記のように、フォルダとオブジェクトに関連するルールは OS でのファイル管理とほぼ同様のものです。

3.1 フォルダの管理

システムの初期化時が終わっていれば、Xone のデータベースにはルートフォルダ（root）とその下に system という名前のフォルダが作られます。フォルダの管理には、com.fiverworks.xone.model パッケージの XoneNode、XoneFolder クラスを使います。ルートフォルダは XoneFolder に定義されており、パス区切り文字は XoneNode に定義されています。ルートフォルダは"root"、パス区切り文字は"/"として定義しています。
たとえば、

```
XoneFolder.ROOT_FOLDER + XoneNode.SEPARATOR + “test"
```

というのは、

```
root/test
```

というパスを示しています。

Xone のデータベースとやりとりするときは、すべて com.fiverworks.xone.mw パッケージの MwMain クラスを通して行います（mw はミドルウェアの略です）。MwMain にはフォルダやオブジェクトの管理、ユーザやセッションの管理などのメソッドが用意されています。これらのメソッドを利用するには、最初にログインする必要があります。

例：

```
MwMain main = new MwMain();
main.login("name", "password".toCharArray());
```

ここで login メソッドには、インストールガイドで設定したユーザの名前とパスワードを指定してください。

3.1.1 フォルダの作成

フォルダを作るには、MwMain の newFolder メソッドを使います。このメソッドには、

・XoneFolder newFolder(String parent, String name)
・XoneFolder newFolder(String parent, String name, String description)
・XoneFolder newFolder(String parent, XoneFolder folder)

の3種類あります。もっともシンプルなのは、親フォルダのパス名と新規のフォルダ名を指定するものです。

例:

```
main.newFolder("root/classes", "master");
```

（注:ここで上記の main は MwMain のインスタンスを表します。以下同じ）

この例では、"root/classes"というフォルダ内に"master"というフォルダを作成します。ここで、親フォルダが存在しなかったり、不正なフォルダ名を指定すると例外が投げられます。

newFolder メソッドは新規に作成されたフォルダを XoneFolder で返します。

3.1.2 フォルダの取得

フォルダ情報を取得するには、getFolder メソッドを使います。

例:

```
XoneFolder xf = main.getFolder("root/classes/master");
```

フォルダ情報を取得せずに、単にフォルダが存在するかどうか調べるときは、existsFolder メソッドを使います。

例:

```
boolean exists = main.existsFolder("root/classes/master");
```

この例では、exists が true ならば root/classes/master フォルダが存在することになります。

3.1.3 フォルダの更新

フォルダの名前や説明などを後で変更するときは、folderProperty メソッドを使います。これには、newFolder メソッドと同様に、

・XoneFolder folderProperty(String path, String name)
・XoneFolder folderProperty(String path, String name, String description)
・XoneFolder folderProperty(String path, XoneFolder folder)

の3種類あります。

例:

```
main.folderProprty("root/classes/master","test");
```

この例は、root/classes/master フォルダを root/classes/test に名前を変更するものです。
folderProperty メソッドは変更後のフォルダ情報を XoneFolder で返します。

3.1.4 フォルダの削除・復活

フォルダの削除には、deleteFolder メソッドを使います。

例:

```
int result = main.deleteFolder("root/classes/master");
```

これは、root/classes/master を削除するものです。result には削除したフォルダ数の合計が返されます。たとえば、root/classes/master の下にフォルダが 2 個あるとすれば、合計で 3 個なので result は 3 になります。

フォルダを削除するといっても、実際は削除マークが付けられるだけで、直後なら（上書きなどされ

ていなければ）復活することもできます。復活には、reviveFolder メソッドを使います。

例：

```
int result = main.reviveFolder("root/classes/master");
```

復活できた場合は、さきほどと同様に復活したフォルダの合計数が返されます。もちろん、フォルダ内のオブジェクトも復活されます。

削除マークを付けるだけではなく、完全にデータを消したいときは、次の deleteFolder メソッドで、

・int deleteFolder(String path, boolean completely)

completely に true を指定します。これで削除すると、復活はできません。また、この場合すでに削除マークの付いているフォルダでも完全に削除することができます。

3.1.5 フォルダの移動・コピー

フォルダの移動やコピーには、それぞれ moveFolder、copyFolder メソッドを使います。

・int moveFolder(String source, String dest)
・int copyFolder(String source, String dest)

source フォルダを dest フォルダの下に移動またはコピーします。このとき dest 側にすでに同じ名前のフォルダがあると例外が投げられます。また、メソッドが成功した場合、移動またはコピーしたフォルダの合計数が返されます。

3.2 オブジェクトの管理

オブジェクトの保存や読み込みなども、すべて com.fiverworks.xone.mw パッケージの MwMain クラスを通して行います。

3.2.1 オブジェクトの保存

オブジェクトの保存には、save メソッドを使います。save メソッドには、

・int save(String parent, XoneObject[] objs)
・int save(String parent, XoneObject[] objs, String writeMode)
・int save(String parent, Transformable[] objs)
・int save(String parent, Transformable[] objs, String writeMode)

の 4 種類あります。Transformable というのは、com.fiverworks.xone.model パッケージにあるインターフェイスであり、これについては「6　Transformable なオブジェクト」を参照してください。また、writeMode というパラメータには、com.fiverworks.xone.mw.Commands で定義されている、NEW または OVERWRITE のいずれかを指定します。

例：

```
XoneAnyInstance xa1 = XoneModel.newXoneAnyInstance(...);
                :
XoneAnyInstance xa2 = XoneModel.newXoneAnyInstance(...);
                :
main.save("root/instances", new XoneObject[]{xa1, xa2}, Commands.OVERWRITE);
```

この例では、root/instances フォルダに xa1 と xa2 というオブジェクトを Commands.OVERWRITE で書き込みます。このモードでは同じ名前のオブジェクトがあっても上書きします。一方、

Commands.NEW は同じ名前のオブジェクトがあるときは保存されません。writeMode を省略すると、Commands.NEW で保存されます。save メソッドは保存できたオブジェクトの数を返します。

次のプログラムはログインした後、ルートフォルダの下に test というフォルダを作ってオブジェクトを保存するものです。login メソッドには、すでに設定してあるログイン名とパスワードを入れて実行してください(パスワードは暗号化されてサーバに送られます)。

・プログラム７

```
ppackage samples.model;
import com.fiverworks.xone.model.*;
import com.fiverworks.xone.mw.*;
import com.fiverworks.xone.*;
public class Sample7 {
	private void test() {
		MwMain main = new MwMain();
		try {
			main.login("name", "password".toCharArray());
			String fname = "test";
			main.newFolder(XoneFolder.ROOT_FOLDER, "test");
			String parent = XoneFolder.ROOT_FOLDER + XoneNode.SEPARATOR + fname;
			XoneObject xo = makeObj();
			main.save(parent, new XoneObject[]{xo});
		} catch (XoneException ex) {
			ex.printStackTrace();
		} catch (XoneRuntimeException ex) {
			ex.printStackTrace();
		} finally {
			if (main.isLogin()) main.logout();
		}
	}

	private XoneObject makeObj() {
		XoneAnyInstance xai = XoneModel.newXoneAnyInstance("製品", "製品A");
		xai.addElement("型番", XoneModel.STRING, "xxxxx");
		xai.addElement("価格", XoneModel.INT, "10000");
		return xai;
	}

	public static void main(String[] args) {
		new Sample7().test();
	}
}
```

※二重にログインすることはできないので、もしログインしてからエラーなどでログインしたままになってしまった場合、セッションマネージャを利用して強制的にログアウトしてください。セッションマネー

ジャについては、ツールガイドを参照してください。

3.2.2 オブジェクトの読み込み

オブジェクトを読み込むメソッドには、次の種類があります。

1.XoneObject[] load(String parent)
2.XoneObject[] load(String parent, String[] names)
3.XoneObject[] load(String[] paths)
4.XoneObject[] load(String parent, String orderBy, int limit, int offset)
5.XoneObject[] load(String parent, String orderBy, String elementCondition, int limit, int offset)
6.XoneObject[] load(String parent, String where)
7.XoneObject[] load(String parent, String where, String elementCondition)

1.は指定した親フォルダ内のオブジェクトをすべて読み込みます。ただし、オブジェクトの数が多すぎるときは例外が投げられます。この一度に読み込める最大数は、xone.properties の db.limit で設定できます(「14.2　設定」参照)。

2.は親フォルダと読み込むオブジェクトの名前の配列を指定します。
例:

```
XoneObject[] xos = main.load("root/test", new String[]{"abc", "xyz"});
```

この例では root/test フォルダから、abc と xyz という名前のオブジェクトを読み込みます。

3.は複数のパスから読み込む方法です。
例:

```
XoneObject [] xos = main.load(new String[]{"root/test/abc", "root/test1/xyz"});
```

この例では、root/test フォルダの abc というオブジェクトと、root/test1 フォルダの xyz というオブジェクトを読み込みます。

4.は SQL の select 文と同様に、並び替えるフィールド(orderBy)、一度に読み込む最大数(limit)、何番目から読み込むか(offset)を指定します。並び替えるフィールドには、次の表の MwMain に定義されたフィールド名を指定できます。これらのフィールドは、オブジェクトを保存しているデータベースのテーブルのフィールド名です。これが null ならば、NAME_FIELD で並べ替えます(並び替えは昇順)。

フィールド名	型	内容
NAME_FIELD	文字列	オブジェクト名が格納されているフィールド
TYPE_FIELD	文字列	オブジェクトのタイプが格納されているフィールド。 内容は XoneClass、XoneInstance、XoneClass のいずれかになっている
CLASSNAME_FIELD	文字列	オブジェクトのクラス名が格納されているフィールド。 XoneClass の場合はオブジェクト名と同じ(
TIMESTAMP_FIELD	日時	オブジェクトが作成または更新された日時
OBJECTID_FIELD	数値	オブジェクトの ID フィールド
FOLDERID_FIELD	数値	格納されているフォルダを示すフォルダの ID
HINT_FIELD	文字列	検索の hint のためのフィールド(後述)

読み込む最大数(limit)を指定したくないときは-1(または com.fiverworks.xone.mw.Commands ク

ラスの Commands.ALL)）を指定してください。この場合、xone.properties の db.limit の値よりオブジェクトの数が多くても例外は投げられません。
何番目から読み込むか（offset）は、必要なければ 0（または Commands.ZERO_OFFSET）を指定します。

例：

```
XoneObject [] xos = main.load("root/test", MwMain.NAME_FIELD, 20, 10);
```

この例は、root/test フォルダ内でオブジェクト名（NAME_FIELD）で並べ替え、その 10 番目から最大 20 個のオブジェクトを読み込みます。

5.は 4.にエレメントの値による条件式(elementCondition)を指定するものです。

例：

```
XoneObject[] xos = main.load("root/test",  null, "#price < 1500;",20,10);
```

この例では、オブジェクト名で並べ替え（orderby のパラメータが null なので）、さらに price という名前のエレメントの値が 1500 未満のオブジェクトで、その 10 番目から最大 20 個のオブジェクトを読み込ます 。この例に示したようにシャープ記号(#)とエレメント名を指定すると、そのエレメントの値に置き換えられます。式の最後にセミコロンを忘れやすいので注意してください。エレメントの条件式にはさまざまな関数を利用することもできますし、ユーザが定義した関数も利用できます。条件式については、後の「10　式について」を参照してください。

6.は、SQL の where 句に相当する条件式を指定できます。前述の NAME_FIELD、TYPE_FIELD、CLASSNAME_FIELD、TIMESTAMP_FIELD、OBJECTID_FIELD、FOLDERID_FIELD、HINT_FIELD のフィールドを使った指定ができます。

例：

```
XoneObject [] xos = main.load("root/test",  MwMain.NAME_FIELD + "  LIKE 'xxx'");
```

この例では、オブジェクト名が LIKE 'xxx'の条件に一致するオブジェクトが読み込まれます。フィールドの型が文字列のときは、次の例のようにシングルクオートで囲むのを忘れないでください。

例：

```
XoneObject [] xos = main.load("root/test",  MwMain.CLASSNAME_FIELD + "=
'root/classes/Book'");
```

この例では、クラス名が root/classes/Book のオブジェクトを読み込みます。

where はそのまま SQL の select 文の where 句に渡されるので、この中に記述できる内容は RDB によって異なります。また、条件式の中にセミコロン（;）を含めることはできません。

7.は 6.の where 句の条件に加え、さらにエレメントの値による条件式(elementCondition)を指定できます。

例：

```
XoneObject[] xos = main.load("root/test",  MwMain.CLASSNAME_FIELD + "=
'root/classes/Book'",  "#price < 1500;");
```

この例では、前述の条件に合致し、さらに price という名前のエレメントの値が 1500 未満のオブジェクトだけを読み込みます。 elementCondition は RDB による違いはなく、どの RDB でも同様に動作します。elementCondition の条件だけで読み込みたい場合は、パラメータの where を null にします。

なお、ここで示した読み込みの指定方法は、オブジェクトの削除・復活・カウントでもほぼ同様に利用できます。

3.2.3 オブジェクトの削除・復活・カウント

オブジェクトの削除には、delete メソッドを使います。delete メソッドには、次の種類があります。

・int delete(String parent)
・int delete(String parent, boolean completely)
・int delete(String parent, String[] names)
・int delete(String parent, String[] names, boolean completely)
・int delete(String[] paths)
・int delete(String[] paths, boolean completely)
・int delete(String parent, String where)
・int delete(String parent, String where, boolean completely)
・int delete(String parent, String where, String elementCondition)
・int delete(String parent, String where, String elementCondition, boolean completely)

パラメータに completely の付くものとそうでないものがあります。これはフォルダの削除と同様、削除マークを付けるだけか、完全に削除するかを指定するものです。completely が true の場合は完全に削除します。これが false だったり、このパラメータのないメソッドでは削除マークを付けるだけです。
それぞれのパラメータはオブジェクトの読み込みと同様なので、そちらを参照してください。delete メソッドでは、削除したオブジェクトの合計数を返します。

例:

```
int result = main.delete("root/test");
```

この例では、root/test フォルダ内のオブジェクトをすべて削除します（削除マークを付けるだけ）。フォルダ内にオブジェクトがいくつあっても削除します。

例:

```
int result = main.delete(new String[]{"root/test/abc", "root/test1/xyz"}, true);
```

この例では、root/test フォルダの abc というオブジェクトと、root/test1 フォルダの xyz というオブジェクトを完全に削除します。
なお、delete メソッド（および次の revive メソッドも）で where を指定できるものがありますが、これも load メソッドと同様に SQL の select 文の where 句に渡されます。

削除マークを付けたオブジェクトの場合は、revive メソッドで復活できます。ただし、削除マークを付けても、同じ名前のオブジェクトを保存（書き込みモードが NEW でも）すると上書きされていまい、復活はできません。あやまって削除してしまった場合は、速やかに復活(revive)してください。
revive メソッドには、次の種類があります。

・int revive(String parent)
・int revive(String parent, String[] names)
・int revive(String[] paths)
・int revive(String parent, String where)
・int revive(String parent, String where, String elementCondition)

パラメータは delete メソッドと同様です。フォルダ名を指定した場合は、そのフォルダ内の削除マークの付いたオブジェクトだけを復活します。オブジェクト名やオブジェクトのパス名を指定した場合、そのオブジェクトが削除マークの付いたものでなければ例外が投げられます。

例:

```
int result = main.delete(new String[]{"root/test/abc", "root/test1/xyz"}, true);
```

この例では、root/test フォルダの abc というオブジェクトと、root/test1 フォルダの xyz というオブジェクトを復活します。ただし、root/test/abc に削除マークが付いていても、root/test/xyz には付いていなかった場合、例外が投げられて両方とも復活されません。revive メソッドでは、復活できたオブジェクトの合計数を返します。

オブジェクトを削除したり、読み込む前にオブジェクト数を調べたいときは、count メソッドを使います。

・int count(String parent)
・int count(String parent, String where)
・int count(String parent, String where, String elementCondition)

それぞれのパラメータは、delete メソッドと同様です。

例:

```
int result = main.count("root/test");
```

この例では、root/test フォルダ内のオブジェクト(削除されているオブジェクトは含まれません)の数を取得しています。

オブジェクトが存在するかどうかを調べるときは、exists メソッドを使います。

・boolean exists(String path)

例:

```
boolean result = main.exists("root/test/abc");
```

この例では、root/test フォルダ内の abc というオブジェクトがあるかどうか調べています。結果が true であれば、存在することになります。ただし、削除されているオブジェクトは存在しないものとして、結果は false が返ります。

3.2.4 オブジェクトの移動・コピー

オブジェクトの移動とコピーは、それぞれ move と copy メソッドを使います。

・int copy(String source, String dest)
・int copy(String source, String dest, String writeMode)
・int copy(String source, String[] names, String dest)
・int copy(String source, String[] names, String dest, String writeMode)
・int move(String source, String dest)
・int move(String source, String dest, String writeMode)
・int move(String source, String[] names, String dest)
・int move(String source, String[] names, String dest, String writeMode)

パラメータの writeMode は保存のときのモードと同じで、同じ名前のオブジェクトがあったときに上書き(Commands.OVERWRITE)または新規(Commands.NEW)のいずれかを指定します。このパラメータがないメソッドでは、新規(Commands.NEW)で書き込みます。

例:

```
int result = main.copy("root/test", "root/dest");
```

この例では、root/test フォルダ内のオブジェクトを root/dest フォルダ内にコピーします。同じ名前のオブジェクトが root/dest フォルダ内にあれば上書きしません。copy や move メソッドでは、結果として実際に移動やコピーできたオブジェクトの数が返されます。
例:

```
int result = main.move("root/test", new String[]{"abc", "xyz"}, "root/dest");
```

この例では、root/test フォルダ内の abc と xyz という名前のオブジェクトを root/dest フォルダ内に移動します。同じ名前のオブジェクトが root/dest フォルダ内にあれば上書きしません。move の場合、移動されなかったオブジェクトはソース側のフォルダに残ります。たとえば、この例で root/dest フォルダ内にすでに abc という名前のオブジェクトがあった場合、xyz だけが移動され、abc は root/test フォルダ内に残ります。

3.2.5 オブジェクトのメタ情報

オブジェクトはその内容を変更せずに、名前、更新(作成)時間、ヒント属性を変更することができます。これら 3 種類の情報をオブジェクトのメタ情報と呼んでおり、メタ情報の変更には objectProperty メソッドを使います。

・XoneObjectInfo objectProprty(String path, String name)

・XoneObjectInfo objectProprty(String path, String name, Timestamp timestamp)

・XoneObjectInfo objectProprty(String path, String name, Timestamp timestamp, String hint)

例:

```
XoneObjectInfo xoi = main.objectProperty("root/test/abc", “xyz”);
```

この例では、root/test フォルダ内の abc という名前のオブジェクトを xyz という名前に変更します。このメソッドの結果は、com.fiverworks.xone.model パッケージの XoneObjectInfo で返されます。この XoneObjectInfo はオブジェクトのメタ情報を表す Java クラスです。

3.3 ノードについて

Xone では、サーバで管理されるフォルダやオブジェクトのメタ情報をノードと総称しています。ノードは com.fiverworks.xone.model パッケージの XoneNode という Java クラスで表されます。XoneNode はフォルダを表す XoneFolder、オブジェクトのメタ情報を表す XoneObjectInfo のスーパークラスになっており、これらは典型的なコンポジットパターンになっています。

サーバ上のオブジェクトやフォルダには、通常の状態(アンロック)、削除、無効の4つの状態があります。こうした状態は、XoneNode で取得できます。オブジェクトやフォルダの一覧を返す list メソッドでは、XoneNode の配列が取得できます。

・XoneNode[] list(String path)

list メソッドは、XoneNode の配列を返します(path で指定されたフォルダが空の場合は長さ 0 の配列で返されます)。

例:

```
XoneNode[] xns = main.list(“root/test”);
```

この例では、root/test フォルダ内のフォルダとオブジェクトの一覧を取得します。このとき、削除マー

クの付いたフォルダやオブジェクトも含まれます。
個々のノードの状態は、XoneNode のメソッドで調べることができます。無効という状態は、親のフォルダが削除されて、それを単独では復活できないことを示します。サンプルプログラム 8 は、list メソッドと XoneNode を使った例です。

・プログラム 8

```
package samples.model;
import com.fiverworks.xone.model.*;
import com.fiverworks.xone.mw.*;
import com.fiverworks.xone.*;
public class Sample8 {
  private MwMain main;
  private void login() {
    main = new MwMain();
    try {
      main.login("name", "password".toCharArray());
    } catch (XoneException ex) {
      ex.printStackTrace();
    }
  }
  private void test() {
    login();
    try {
      String parent = XoneFolder.ROOT_FOLDER + XoneNode.SEPARATOR + "test";
      XoneNode[] nodes = main.list(parent);
      for (int i = 0; i < nodes.length; i++) {
        XoneNode node = nodes[i];
        System.out.println(i + ":"+ node.getName());
        System.out.println("  path:"+ node.getPath());
        System.out.println("  type:" + node.getType());
        System.out.println("  status:" + node.getStatusString());
        System.out.println("  timestamp:" + node.getTimestamp());
      }
    } catch (XoneRuntimeException ex) {
      ex.printStackTrace();
    } finally {
      if (main.isLogin()) main.logout();
    }
  }
  public static void main(String[] args) {
    new Sample8().test();
  }
}
```

4 制約式

クラスを定義するときには、そのインスタンスのエレメントが取る値に対して、制約を定義できます。たとえば、次の式は

equals('#性別', '男') || equals('#性別', '女') ;

という制約式は、性別という名前のエレメントの値は男または女でなければならない、という意味です。制約式は、オブジェクトの読み込みで記述したエレメントの条件式と同じ文法です。制約式の付いたクラスを定義した例を次に示します。

・プログラム９

```
package samples.model;
import com.fiverworks.xone.model.*;
public class Sample9 {
	public static void main(String[] args) {
		XoneElement xe1 = new XoneElement("性別", XoneModel.STRING);
		XoneElement xe2 = new XoneElement("年齢", XoneModel.INT);

		// 制約式用のエレメントリスト
		XoneElementList xel = new XoneElementList(XoneClass.RESTRICTION_LIST);
		xel.add("性別", XoneModel.STRING,
				"equals('#@', '男') || equals('#性別', '女');");
		xel.add("年齢", XoneModel.STRING,
				"equals('#性別', '男') ?  #@ >= 18 : #@ >= 16;");

		XoneClass xc = XoneModel.newXoneClass("社員",
				new XoneElement[]{xe1, xe2},
				new XoneElementList[]{xel});
		System.out.println(xc);
	}
}
```

制約式は、このサンプルにあるように XoneClass.RESTRICTION_LIST という名前のエレメントリストの中に定義します。XoneClass.RESTRICTION_LIST というのは"_restrictionList"と定義されており、予約名になっています。このエレメントリストの中に、エレメントとして

new XoneElement("性別", XoneModel.STRING, "equals('#@', '男') || equals('#@', '女');"));

のような制約式を指定しています。これは"性別"というエレメントの制約式を定義したものです。ここで、#@はこのエレメント自身の値を示すもので、ここでは#性別と書いても同じことです。もう一つの制約式、

new XoneElement("年齢", XoneModel.STRING, "equals('#性別', '男') ? #@ >= 18 : #@ >= 16;")

は、「年齢というエレメントの値は、もし性別というエレメントの値が'男'ならば 18 以上、そうでなければ 16 以上でなければならない」という意味になります。
制約式に合わない値が指定されたときのメッセージもクラスに定義できます。メッセージを定義した

例を次に示します。

```
・プログラム10
package samples.model;
import com.fiverworks.xone.model.*;
public class Sample10 {
	public static void main(String[] args) {
		XoneElement xe1 = new XoneElement("性別", XoneModel.STRING);
		XoneElement xe2 = new XoneElement("年齢", XoneModel.INT);

		// 制約式用のエレメントリスト
		XoneElementList xel = new XoneElementList(XoneClass.RESTRICTION_LIST);
		xel.add("性別", XoneModel.STRING,
			"equals('#@', '男') || equals('#性別', '女');");
		xel.add("年齢", XoneModel.STRING,
			"equals('#性別', '男') ?  #@ >= 18 : #@ >= 16;");
		// 制約式のメッセージ用のエレメントリスト
		XoneElementList xel1 =
			new XoneElementList(XoneClass.RESTRICTION_MESSAGE_LIST);
		xel1.add("性別", XoneModel.STRING,
			"性別の欄は'男'または'女'でなければなりません");
		xel1.add("年齢", XoneModel.STRING,
			"年齢の欄は性別が'男'の場合は18以上、そうでなければ16以上の数値を指
定してください");

		XoneClass xc = XoneModel.newXoneClass("社員",
			new XoneElement[]{xe1, xe2},
			new XoneElementList[]{xel, xel1});
		System.out.println(xc);
	}
}
```

制約式のメッセージは、XoneClass.RESTRICTION_MESSAGE_LIST という名前のエレメントリストの中に定義します。XoneClass.RESTRICTION_MESSAGE_LIST というのは"_restrictionMessageList"と定義されており、予約名になっています。このエレメントリストの中に、エレメントとしてメッセージを定義します。

new XoneElement("性別", XoneModel.STRING, "性別の欄は'男'または'女'でなければなりません")

これは、性別というエレメントに対する制約式のメッセージです。このようにメッセージを定義しておけば、アプリケーションプログラムはユーザの間違った入力値に対する適切なメッセージを表示することができます。

制約式やメッセージが定義されたクラスのインスタンスを作って、それが制約式に合致するかどうかを調べるには validate メソッドを使います。プログラム 11 にその例を示します。なお、制約式ついて詳しくは後の「10　式について」で記述します。

・プログラム11

```
package samples.model;
import com.fiverworks.xone.model.*;
public class Sample11 {
  private void test() {
    XoneClass xc = makeClass();
    XoneInstance xi = XoneModel.newXoneInstance(xc, "社員A");
    xi.setElementValue("性別", "男");
    xi.setElementValue("年齢", Integer.toString(18));
    ValidateResult[] vr = xi.validate();
    printResult(xi, vr);
    xi.setElementValue("性別", "男");
    xi.setElementValue("年齢", Integer.toString(15));
    vr = xi.validate();
    printResult(xi, vr);
  }
  private void printResult(XoneObject instance, ValidateResult[] vr) {
    System.out.println("--  評価結果  ------");
    for (int i = 0; i < vr.length; i++) {
      System.out.println(vr[i]);
    }
  }
  private XoneClass makeClass() {
    XoneElement xe1 = new XoneElement("性別", XoneModel.STRING);
    XoneElement xe2 = new XoneElement("年齢", XoneModel.INT);
    XoneElementList xel = new XoneElementList(XoneClass.RESTRICTION_LIST);
    xel.add("性別", XoneModel.STRING, "equals('#@', '男') || equals('#性別', '女');");
    xel.add("年齢", XoneModel.STRING, "equals('#性別', '男') ?  #@ >= 18 : #@ >= 16;");
    XoneElementList xel1 = new XoneElementList(XoneClass.RESTRICTION_MESSAGE_LIST);
    xel1.add("性別", XoneModel.STRING, "性別の欄は'男'または'女'でなければなりません");
    xel1.add("年齢", XoneModel.STRING, "年齢の欄は性別が'男'の場合は18以上、そうでなければ16以上の数値を指定してください");
     XoneClass xc = XoneModel.newXoneClass("社員", new XoneElement[]{xe1, xe2}, new XoneElementList[]{xel, xel1});
    return xc;
  }
  public static void main(String[] args) {
    new Sample11().test();
  }
}
```

5 クラスとインスタンス

これまでのサンプルプログラムでは、クラスを作ってすぐにそのインスタンスを作りましたが、通常はクラスは作成したらサーバに保存し、その上で、サーバのクラスを読み込んでインスタンスを作ります。プログラム 12 は、サーバのクラスを読み込んでインスタンスを作るまでを示したものです。

・プログラム 12

```
package samples.model;
import com.fiverworks.xone.model.*;
import com.fiverworks.xone.mw.*;
import com.fiverworks.xone.*;
public class Sample12 {
  private MwMain main;
  private void login() {
    main = new MwMain();
    try {
      main.login("name", "password".toCharArray());
    } catch (XoneException ex) {
      ex.printStackTrace();
    }
  }
  private void test() {
    login();
    try {
      String cname = "root/test/注文";
      XoneObject[] xos = main.load(new String[]{cname});
      XoneClass xc = (XoneClass)xos[0];
      // 上の文は次の文でもよい
      //XoneClass xc = XoneModel.getXoneClass(cname);
      XoneInstance xi = XoneModel.newXoneInstance(xc, "注文 A");
      System.out.println(xi);
    } catch (XoneRuntimeException ex) {
      ex.printStackTrace();
    } finally {
      if (main.isLogin()) main.logout();
    }
  }
  public static void main(String[] args) {
    new Sample12().test();
  }
}
```

クラスは読み込まれると内部にキャッシュされるので、クラスを取得するには

```
XoneClass xc = XoneModel.getXoneClass(cname);
```

としてもかまいません。キャッシュされていないときは、null が返ります。
アプリケーションを開発する際には、まず必要なクラスを定義します。その上で、アプリケーションの起動時に、必要なクラスを読み込んでおくとよいでしょう。

RDB では製品マスターや顧客マスターなど、一般にマスターテーブルを作ります。そして、そのマスターの ID を使って、他のテーブルから参照します。Xone ではすべてオブジェクトですが、オブジェクトの中から他のオブジェクトを参照するには、ユーザ定義型のエレメントを使うとよいでしょう。たとえば、次の注文クラスでは、顧客というエレメントを"root/test/顧客"型と定義しています。

```
class:root/test/注文
type:XoneClass
name:注文
path:root/test/注文
hint:
  #顧客,root/test/顧客,null
  #注文日,date,null
  #送付先,string,null
 list:_restrictionList
 list:_restrictionMessageList
 list:注文内容
```

このクラスのインスタンスを作るときは、顧客のエレメントの値として顧客インスタンスのパスを設定します。RDB になぞらえて言えば、顧客インスタンスはマスターデータ、顧客インスタンスのパスは ID ということになります。このようにエレメントの型は/で区切ることができるので、パスを表現できます。区切られた個々の要素は命名規則に従う必要があります。

5.1 クラスを変更する

あるクラスのインスタンスを、別のクラスのインスタンスにすることができます。インスタンスを作ってから、クラスを変更するときは XoneObject の adaptClass メソッドを使います。adaptClass はインスタンスにのみ使えるメソッドです。これは以下のように使います。

```
XoneClass xc = ・・・
XoneInstance xi = ・・・
xi.adaptClass（xc);
```

クラスを変更しても、インスタンスはサーバから読み込んだだけでは前のままです。そこで、上記のように adaptClass を実行すると、インスタンスはそのクラス定義の内容によって、次のルールで変更されます。

1.インスタンスにはなく、クラスにはあるエレメントやエレメントリストは追加される
2.インスタンスにあって、クラスにはないエレメントやエレメントリストは削除される
3.同じ名前・型のエレメントは、インスタンスの内容がそのまま残る
4.同じ名前で、異なる型のエレメントがクラスにあるとき、インスタンスのエレメントの値がその型に入れることができればその内容は残る。そうでなければ、クラスのデフォルト値が入る。

注文というクラスが次のように定義されているとします（制約関連は省略しています）。

```
class:root/test/注文
type:XoneClass
name:注文
path:root/test/注文
```

hint:
#合計価格,int,null
#単価,float,null
#税,float,null

また、注文 A というクラスは次のように定義されているとします。

class:root/test/注文 A
type:XoneClass
name:注文 A
path:root/test/注文 A
hint:
#合計価格,decimal,null
#単価,int,null
#値引率,float,3.0

ここで、プログラム 13 を実行すると、結果は次のようになります。

class:root/test/注文
type:XoneInstance
name:注文 1
path:注文 1
hint:null
#合計価格,int,10000
#単価,float,10.5
#税,float,null

class:root/test/注文 A
type:XoneInstance
name:注文 1
path:注文 1
hint:null
#合計価格,decimal,10000
#単価,int,null
#値引率,float,3.0

前述したルールにしたがってインスタンスが変更されています。ルール1によって、値引率というエレメントが追加されています（クラスで定義しているデフォルト値も入ります）。ルール2によって税というエレメントは削除されました。
合計価格は、int から decimal なので、そのまま値が残ります。一方、単価は float から int であり、この例では値が入らないので値は残りません（クラスにはデフォルト値がないので null になります）。

・プログラム 13

```
package samples.model;
import com.fiverworks.xone.model.*;
import com.fiverworks.xone.mw.*;
import com.fiverworks.xone.*;
public class Sample13 {
  private MwMain main;

  private void login() {
    main = new MwMain();
    try {
      main.login("name", "password".toCharArray());
    } catch (XoneException ex) {
      ex.printStackTrace();
    }
  }
  private void test() {
    login();
    try {
      String cname = "root/test/注文";
      String cname1 = "root/test/注文 A";

      XoneObject[] xos = main.load(new String[]{cname, cname1});
      XoneClass xc = XoneModel.getXoneClass(cname);
      XoneInstance xi = XoneModel.newXoneInstance(xc, "注文 1");
      xi.setElementValue("単価", "10.5");
      xi.setElementValue("合計価格", "10000");
      System.out.println(xi);

      XoneClass xc1 = XoneModel.getXoneClass(cname1);
      xi.adaptClass(xc1);
      System.out.println(xi);
    } catch (XoneRuntimeException ex) {
      ex.printStackTrace();
    } finally {
      if (main.isLogin()) main.logout();
    }
  }

  public static void main(String[] args) {
    new Sample13().test();
  }
}
```

6 Transformable なオブジェクト

com.fiverworks.xone.model.Transformable はシンプルなインターフェースであり、次の2個のメソッドが定義されています。

```
void set(XoneObject xo);
XoneObject toXoneObject();
```

このインターフェースを使うと、Xone のオブジェクトをほとんど Java のオブジェクト(POJO)と同等に扱うことができます。このため、システムの内部でも多用しており、これまで出てきた XoneNode、XoneFolder などもこれをインプリメントしています。Transformable は「変換可能な」という意味で、これは Xone と Java のオブジェクトが相互に変換できることから付けたものです。プログラム 14 は、Transformable インターフェースを実装した例です。

・プログラム 14

```
package samples.model;
import com.fiverworks.xone.model.*;
import com.fiverworks.xone.mw.*;
import com.fiverworks.xone.*;
public class Sample14 implements Transformable {
  private String modelId;
  private int price;
  private String objName;
  public Sample14(String objName) {
    this.objName = objName;
  }

  public java.lang.String getModelId() {return modelId;}
  public void setModelId(java.lang.String modelId) {this.modelId = modelId;}
  public int getPrice() {return price;}
  public void setPrice(int price) {this.price = price;}

  //Transformable
  public void set(XoneObject xo) {
    this.objName = xo.getName();
    this.modelId = xo.getElementValue("型番");
    this.price = xo.getElement("型番").getValueAsInt();
  }
  public XoneObject toXoneObject() {
    XoneAnyInstance xai = XoneModel.newXoneAnyInstance("Sample", objName);
    xai.addElement("型番", XoneModel.STRING, modelId);
    xai.addElement("価格", XoneModel.INT, Integer.toString(price));
    return xai;
  }
}
```

このプログラムでは、一般的な Java のオブジェクトと同様にアクセッサ（set/get メソッド）でオブジェクトの状態の取得と変更ができます（注：このサンプルでは、set(XoneObject xo)でエラー処理を行っていませんが、実際にはオブジェクトのクラス名や値を調べるべきでしょう）。

com.fiverworks.xone.mw.MwMain には Transformable なオブジェクトを直接データベースに保存する save メソッドがあります。これを使うと、さきほどのサンプルのオブジェクトの場合、作成から保存までは次のようになります。

```
// オブジェクトを作る
Sample14 obj1 = new Sample14(“name1");
Sample14 obj2 = new Sample14(“name2");
        ・
        ・
        ・
// 保存する
MwMain main = new MwMain();
main.login("name", "password".toCharArray());
main.save(“root/folder1", new Transfromable[]{obj1, obj2});
```

6.1 オブジェクト名とニックネーム

Transformable なオブジェクトと Java のオブジェクトとの大きな違いはオブジェクトの名前です。Xone ではデータベース上で、オブジェクト名をオブジェクトの識別に使います。このとき、1つのフォルダ内では、オブジェクト名はユニークでなければなりません。そのため、Transformable なオブジェクトも適切な名前を付ける必要があります。
ところが、複数のユーザが同時にオブジェクトを保存するような状況では、ユニークな名前を付けるのが困難な場合があります（特にオブジェクト名を自動的にプログラムで付けるようなとき）。このようなときは、ユーザのニックネームを利用してオブジェクト名を付けると良いかもしれません。ユーザ情報には「ニックネーム」という項目があり、これに各ユーザに固有の名前を入れておきます。そうしておいて、ニックネームをオブジェクト名の接頭辞に使います。MwMain では、ログインしたユーザ情報を取得できますから、それからニックネームを取り出します。

```
MwMain mwmain = new MwMain();
XoneUser xu = mwmain.login("name", "password".toCharArray());
String nickName = xu.getNickName();
```

こうしておいて、オブジェクト名をたとえば nickName＋番号（日時などから生成）としておけば、個々のユーザアプリケーションでユニークなオブジェクト名を作ることができます。この命名法は、Transformable なオブジェクトに限らず Xone のオブジェクトで利用できます。ユーザ情報では「ニックネーム」という項目は必須ではありませんが、できれば入力しておくと良いでしょう。このように利用するときはニックネームを次の点に注意して設定してください。

・ユニークにすること
・命名規則に従うため、先頭は数字にしない、アンダースコアやドル以外の記号を使わない

7 バッチ処理

com.fiverworks.xone.mw.MwMain にはデータベースとやりとりするさまざまなメソッドが用意されていますが、内部的にはたった1つのメソッドだけがやりとりしています。そのメソッドとは、

```
public XoneObject[] execute(XoneObject command, XoneObject[] args)
```

です。このメソッドでは command も XoneObject ですし、そのパラメータ（args）と返値も XoneObject の配列です。これ以外の load や delete などのメソッドはすべて、この execute メソッドを使っています。load や delete などのメソッドでは、基本的に

1.パラメータに合わせた command オブジェクトを作る
2.execute メソッドを呼び出す
3.（必要があれば返値を加工する）

という処理を行っています。たとえば、load のメソッドの一つを擬似的に書くと

```
public XoneObject[] load(String parent) {
        XoneObject command = Commands.getLoadCommand(parent);
        return   execute(command, null)の実行結果
}
```

となっています。command オブジェクトはすべて com.fiverworks.xone.mw.Commands で作っています。
この command オブジェクトを一度に複数実行するのがバッチ処理です。プログラム 15 は、バッチ処理の例です。

・プログラム 15

```
package samples.model;
import com.fiverworks.xone.model.*;
import com.fiverworks.xone.mw.*;
import com.fiverworks.xone.*;
public class Sample15 {
  private MwMain main;

  private void login() {
    main = new MwMain();
    try {
      main.login("name", "password".toCharArray());
    } catch (XoneException ex) {
      ex.printStackTrace();
    }
  }
  private void test() {
    login();
    try {
      String parent = "root/test";
      BatchCommand bc = new BatchCommand();
      XoneObject command = Commands.getDeleteFolderCommand(parent + "/xonetest2");
      bc.add(command);
      command = Commands.getDeleteCommand(parent, new String[]{"製品A"});
      bc.add(command);
      main.executeBatch(bc);
    } catch (XoneRuntimeException ex) {
      ex.printStackTrace();
    } finally {
      if (main.isLogin()) main.logout();
    }
  }

  public static void main(String[] args) {
    new Sample15().test();
  }
}
```

このプログラムでは、フォルダの削除とオブジェクトの削除とをバッチで実行しています。バッチ処理ではまず、BatchCommand のインスタンスを作り、それに必要なコマンドを追加していきます。そしてそのインスタンスを MwMain の executeBatch メソッドに渡します（これも内部的には execute メソッドで実行しています）。バッチ処理では途中でエラーが発生した場合、すべてロールバックされます（何も実行しなかったと同じことになります※）。

※Xone の現在のバージョンでは、MySQL 版に関してはロールバックされず、エラーが起きる直前までの処理が実行されてしまいます。

Commands には、getXXXCommand というメソッドがいくつもあり、この XXX の部分は MwMain でのメソッド名と同じものに対応しています。

BatchCommand にコマンドを追加するメソッドには、

・void add(XoneObject command)
・void add(XoneObject command, XoneObject arg)
・void add(XoneObject command, XoneObject[] args)
・void add(XoneObject command, Transformable arg)
・void add(XoneObject command, Transformable[] args)

の 5 種類あります。前述のプログラムでは、第2パラメータ（args）をとらない add メソッドを使いましたが、第2パラメータが必要なコマンドは次のようなものがあります。

MwMain のメソッド名	**Commands** でのメソッド名	第2パラメータの内容
save	getSaveCommand	保存するオブジェクト
newFolder	getNewFolderCommand	作成フォルダ関連の情報。パラメータは XoneFolder
folderProperty	getFolderPropertyCommand	変更するフォルダのプロパティの情報。パラメータは XoneFolder
addUser	getAddUserCommand	追加するユーザ情報。パラメータは XoneUser
updateUser	getUpdateUserCommand	変更するユーザ情報。パラメータは XoneUser

第2パラメータの必要なコマンドでは、まず Commands でコマンドを取得し、パラメータを付けて BatchCommand に追加します。BatchCommand では、Tranformable なパラメータを伴うコマンドも追加できます。ですから、Tranformable なオブジェクトをバッチ処理の中で保存することもできます。XoneFolder や XoneUser は Tranformable を実装しているので、次のようにバッチに追加できます。

例：

```
BatchCommand bc = new BatchCommand();
XoneObject command = Commands.getNewFolderCommand("root/test");
XoneFolder xf = new XoneFolder("work");
bc.add(command, xf);
XoneObject[] xos = makeObjs();  // XoneObject を作る仮のメソッド
command = Commands.getSaveCommand("root/test/work");
bc.add(command, xos);
main.executeBatch(bc);
```

この例では、 root/test フォルダ内に work というフォルダを新規に作成し、その中にオブジェクトを保存します。

BatchCommand の実行結果は、個々のバッチコマンドで変更されたオブジェクトおよびフォルダの数が int 配列で返ってきます。そのため、オブジェクトの読み込みなどはバッチ処理で実行はできますが、実質的に意味はありません（読み込んだオブジェクトを受け取ることができないため）。

8 コマンドとビジネスロジック

「7 バッチ処理」で記述したように、com.fiverworks.xone.mw.MwMain で実質的にサーバとやりとりするメソッドは execute メソッドだけです。このメソッドに渡すパラメータが同じであれば、どんなコマンドでも実行できます。また、サーバ側でそのコマンドを処理するプログラム(ビジネスロジック)も自由に作ることができます。

コマンドは Xone のオブジェクトであり、システムで利用するコマンドは Commands クラスで作っています。コマンドを作るときは、まずは Commands の getBasicCommand メソッドを利用して基本的なコマンドを作ります。これは、単に"クラス名を"XoneCommand"、インスタンス名を"xoneCommand"という Any インスタンスを作っているだけです。これで、取得したコマンド(Any インスタンス)に、必要なパラメータを設定してコマンドとしています。たとえば、次の例は

```
XoneObject cmd = Commands.getBasicCommand();
cmd.addElement("command", XoneModel.STRING, "abc");
cmd.addElement("arg1", XoneModel.STRING, "xxx");
cmd.addElement("arg2", XoneModel.STRING, "yyy");
```

コマンドとするオブジェクト(cmd)に対して、command に abc、arg1 に xxx、arg2 に yyy というエレメントを追加したものです。getBasicCommand メソッドで得られるのは、Any インスタンスなので自由にエレメントやエレメントリストの追加が可能です。ここで重要なのが、command というエレメントです。サーバ側(ビジネスロジック層)では、この値を元に必要なプログラムを実行するからです。

一方のビジネスロジック側は、作るときには以下のルールに従う必要があります。
1.パラメータのないコンストラクタを持つこと
2.com.fiverworks.xone.bl.AbstractBizLogic を継承すること
3.public XoneObject[] execute(XoneObject command, XoneObject[] args)を実装すること

3 の execute メソッドは AbstractBizLogic で抽象メソッドとして定義されているので、かならず実装しなければなりません。ビジネスロジックの例をプログラム 16 に、コマンドの例をプログラム 17 にそれぞれ示します。

・プログラム 16

```
package samples.bl;
import com.fiverworks.xone.model.*;
import com.fiverworks.xone.bl.*;
public class Abc extends com.fiverworks.xone.bl.AbstractBizLogic {

  public Abc() {}

  public XoneObject[] execute(XoneObject command, XoneObject[] args) {
    System.out.println("--- Abc コマンド ---¥n" + command);
    return null;
  }
}
```

・プログラム 17

```
package samples.bl;
import com.fiverworks.xone.model.*;
import com.fiverworks.xone.mw.*;
import com.fiverworks.xone.*;
public class CommandTest {
  private MwMain main;

  private void login() {
    main = new MwMain();
    try {
      main.login("name", "password".toCharArray());
    } catch (XoneException ex) {
      ex.printStackTrace();
    }
  }
  private void test() {
    login();
    XoneObject cmd = Commands.getBasicCommand();
    cmd.addElement("command", XoneModel.STRING, "abc");
    cmd.addElement("arg1", XoneModel.STRING, "xxx");
    cmd.addElement("arg2", XoneModel.STRING, "yyy");
    try {
      main.execute(cmd);
    } catch (XoneRuntimeException ex) {
      ex.printStackTrace();
    } finally {
      if (main.isLogin()) main.logout();
    }
  }

  public static void main(String[] args) {
    new CommandTest().test();
  }
}
```

この例では返値はありませんが、必要であれば XoneObject の配列にすればどんな値でも返すことができます。プログラムとして必要なのはこれだけです。後は設定ファイルに、作成したビジネスロジックの情報を入れておきます。

ビジネスロジック用の設定ファイルは、XONE_HOME の config フォルダにある bizLogic-config.xml という XML 形式のファイルです。この設定ファイルには、以下のようになっており、システムのコマンドも定義されています。

```
<config>
	<bizLogics basePath="com.fiverworks.xone.bl.cmd">
		<role> <!-- すべての role で実行可能 -->
			<!-- Data Management -->
			<command name='exists' class='Exists'/>
			<command name='existsFolder' class='ExistsFolder'/>
			（中略）
		</role>
	</bizLogics>

	<!-- user def command
	<bizLogics basePath="???">
		<role name="???">
			<command name='???' class='???'/>
		</role>
	</bizLogics>
	-->
</config>
```

この中の user def command という記述のコメントをはずし、以下のような記述を入れます。

```
	<bizLogics basePath="samples.bl">
		<role>
			<command name='abc' class='Abc'/>
		</role>
	</bizLogics>
```

ここで、command 要素の name 属性にさきほどの command エレメントに定義した値を入れ、class 属性にビジネスロジックのクラスを指定します。bizLogics 要素の basePath 属性を省略したいときは、class 属性にパス名も入れて指定してください（この場合は、class='samples.bl .Abc'とします）。

これで、さきほど作成したコマンドを実行できるようになります。実行するには、プログラム 16 の Abc.java をコンパイルし、パスの通ったフォルダに入れておいてください。これで、プログラム 17 の CommandTest.java を実行すれば、ビジネスロジックが実行されるはずです。

※設定ファイルを書き換えたら、サーバ側を再起動するか、モデルマネージャの「再初期化」ボタンをクリックしてください。モデルマネージャについては、ツールガイドを参照してください。

8.1 AbstractBizLogic とトランザクション

AbstractBizLogic では、データベースに実際にアクセスする DbProxy のインスタンスを作成しています。この DbProxy には実際にデータベースにアクセスするためのメソッドが用意されています（メソッドの詳細については、API ドキュメントを参照してください）。新規に作成するビジネスロジックでは、この DbProxy のメソッドを使っても良いし、あるいは JDBC や O/R マッピングツールを使ってデータベースにアクセスしても良いし、データベースではなくファイルなどにアクセスしてもかまいません。データベースにアクセスする場合、DbProxy の getConnection を使うと、コネクションプールからのコネクションを取得できます。これには、ビジネスロジックの execute メソッドの冒頭で、次のように記述すればよいでしょう。

```
public XoneObject[] execute(XoneObject command, XoneObject[] args) {
    Connection conn = super.dbProxy.getConnection();
    ・・・
```

バッチ処理の中でも使う場合、このようにして取得したコネクションに対して、コミットやロールバックはしないでください。してもかまいませんが、バッチ処理の中で使うと原子性が保てなくなります。また、RDB にテーブルを用意しておいて、ビジネスロジックで JDBC や O/R マッピングツールを使ってアクセスすればパフォーマンスの良いアプリケーションを開発できます。Xone を利用してプロトタイプを作り、その後チューニングするというときに有効な方法です。

8.2 例外について

ビジネスロジックの例外は、すべて XoneRuntimeException を投げるようになっています。これ以外の例外の場合も XoneRuntimeException でラップして投げるようになっています。この例外には、例外のレベル、リトライ可能かどうか、エラー番号などを設定できます。これにはいくつかのコンストラクタがありますが、すべてのパラメータを設定するのは次のコンストラクタです。

```
public   XoneRuntimeException(int level, String message, Throwable cause, boolean retry)
```

ここで level には、XoneRuntimeException で定義されている WARNING、ERROR、FATAL、UNDEFINED のいずれかを指定します。これらは例外の深刻さを表しています。また、message の先頭に 4 桁の数値とコロンを指定しておくと、それをエラー番号とみなします。

例:

```
2000:指定されたパラメータが不正です
```

このエラー番号は、getNumber メソッドで取得できます。エラー番号はシステムで利用している番号と重複しないように 2000 以降を使用してください（付録参照）。retry というパラメータは true にすると、自動的に少々待ってから再度実行するようになっています。この待ち時間とリトライ回数は変更できます（「14.2　設定」参照）。ただし、バッチ処理の中ではリトライされず、すぐに例外が投げられます。

9 XML 対応

Xone は XML によるオブジェクトの入出力をサポートしています。これによって、XML ファイルでのデータ交換を容易に行うことができます。XML での入出力には、com.fiverworks.xone.model.XmlIo を使います。これにはさまざまな入力ストリームやファイルに対して XML で入出力するメソッドがあります。使い方は API ドキュメントを参照してください。出力した例を次に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xoneObjectList SYSTEM
"http://www.fiverworks.com/xone/dtd/xoneObjectList.dtd">

<xoneObjectList xmlns="http://www.fiverworks.com/xone/dtd" ver="1.0">
  <xoneObject name="注文" type="XoneClass">
    <property />
    <xoneElement name="顧客名" type="string">~null;</xoneElement>
    <xoneElement name="注文日" type="date">2004-4-12</xoneElement>
    <xoneElement name="送付先" type="string">~null;</xoneElement>
    <xoneElementList name="注文内容">
      <xoneElement name="デフォルトの要素" type="string" />
    </xoneElementList>
    <xoneElementList name="_restrictionList" />
    <xoneElementList name="_restrictionMessageList" />
  </xoneObject>

  <xoneObject name="注文 A" type="XoneInstance">
    <property>
      <className>注文</className>
    </property>
    <xoneElement name="顧客名" type="string">xxxxx</xoneElement>
    <xoneElement name="注文日" type="date">2004-4-12</xoneElement>
    <xoneElement name="送付先" type="string">東京都調布市 XX 町 1-1-1</xoneElement>
    <xoneElementList name="注文内容">
      <xoneElement name="デフォルトの要素" type="string" />
      <xoneElement name="型番" type="string">xxxxx</xoneElement>
      <xoneElement name="個数" type="int">1</xoneElement>
      <xoneElement name="型番" type="string">yyyyy</xoneElement>
      <xoneElement name="個数" type="int">2</xoneElement>
    </xoneElementList>
  </xoneObject>
</xoneObjectList>
```

なお、出力のときのエンコードは UTF-8 のみです。そのため、出力した XML ファイルをテキストエディタなどで表示するときは、UTF-8 に対応していないと日本語が正しく表示されません。なお、この XML に対応する DTD は、XONE_HOME の dtd フォルダに xoneObjectList.dtd として格納されています。

10 式について

Xone は式を処理するプロセッサを内蔵しています。式はオブジェクトを検索するときの条件式、クラスを定義するときの制約式で利用できますし、単独でも利用できます。また、このプロセッサは java.lang.Math クラスが備えている関数と同じものに加え、いくつかの関数を組み込み関数として持っています。さらにユーザが独自に作成した関数を追加することもできます。

式の記述形式は、基本的に Java と同じです。式の最後には必ずセミコロン**(;)**を付けてください。また、//で始まるとコメントと見なします。
例:

```
sqrt(1.0 * 1.0 + 2.0 * 2.0); //  底辺が 1.0、高さが 2.0 の直角三角形の斜辺の長さ
```

文ではなく式ですから、値を持ちます。値として利用できる型は、double(数値)、文字列、boolean (true と false のみ)を利用できます。

数値定数

数値定数には、以下の 2 つがあります。

PI - 円周率にもっとも近い double 値

E - 自然対数の底 e にもっとも近い double 値

例:

```
2 * PI * 3.0;    // 半径 3.0 の円の円周
```

数値表現

数値表現として 10 進数、16 進数、8 進数、指数表現が使えます。これらは基本的に Java と同じ表現です。

例:

1226	(10 進数)
0x3f	(16 進数)
024	(8 進数)
0.12e23	(指数表現)

文字列

文字列はシングルクオート(')で囲みます。

例:

```
'文字列'
```

シングルクオート自身を含めるには、直前にバックスラッシュ(円記号)を置きます。

例:

```
'\'クオート'
```

論理定数

論理定数には、true と false があります。大文字・小文字の区別があるので、True や FALSE などはエラーになります。

演算子

以下の演算子が利用できます。

・数値演算子(四則演算子と mod 演算子） ： +、-、*、/、%

・関係演算子 ： <、>、<=、>=

・等価演算子 ： ==、!=

・条件 AND 演算子、条件 OR 演算子、NOT 演算子 ： &&、||、!

・条件演算子（三項演算子）: ？ ：

・文字列連結演算子 ： +

'ab' + 'cd' は'abcd'となります。

'ab'+2 や'ab'+true は型の異なる演算と見なし、エラーになります。

組み込み関数

組み込み関数（内部関数）には、大きく文字列関数、boolean 関数、数値関数の3種類あります。これらは、それぞれ結果が文字列、boolean、数値（double）になる関数です。

・文字列関数

関数の書式	内容
toUpperCase(String value)	value で指定された文字列を大文字に変換する
toLowerCase(String value)	value で指定された文字列を小文字に変換する
substring(String value, double start)	value で指定された文字列の start 以降の部分文字列を返す
substring(String value, double start, double end)	value で指定された文字列の start から end までの部分文字列を返す
doubleToString(double value)	value で指定された値を文字列表現にして返す
booleanToString(boolean value)	value で指定された値を文字列表現にして返す
concat(String s1, String s2, ...)	s1 から sn までの文字列を連結して返す
trim(String value)	value で指定された文字列を trim して返す
currentTime()	現在の時刻を返す
currentDate()	現在の日付を返す
currentTimestamp()	現在の日付・時刻を返す

concat(String s1, String s2, ...)は、いくらでもパラメータを指定できます。

例:

concat('ab', 'cd', 'ef, 'gh');

は 'abcdefgh' になります。

currentTime()、currentDate()、currentTimestamp()はそれぞれ java.sql の Date、Time、Timestamp に System.currentTimeMillis()で生成し、それを toString メソッドで取り出したものです。そのため、それぞれの形式は、前述したエレメントでの形式と同じになります。

doubleToString(double value)は double 型の文字列表現で、String.valueOf(double d)と同じ表現形式になります。

・**boolean** 関数

関数の書式	内容
equals(String target, String arg)	target 文字列と arg 文字列が等しいときに true、そうでなければ false
matches(String target, String regex)	target 文字列が正規表現（regex）にマッチすれば true、そうでなければ false
endsWidth(String s1, String s2)	s1 で指定された文字列が、s2 で指定された接尾辞で終るかどうかを判定する
startsWidth(String s1, String s2)	s1 で指定された文字列が、s2 で指定された接頭辞で始まるかどうかを判定する
isNumber(String s1)	s1 で指定された文字列が数値に変換できるかどうかを判定する
isEmail(String value)	value で指定された文字列がメールアドレスの形式かどうか判定する。
after(String d1, String d2)	d1 が d2 より後ならば true、そうでないなら false
before(String d1, String d2)	d1 が d2 より前ならば true、そうでないなら false
between(String d1, String d2, String d3)	d1 が d2 より前で d3 より後ならば true、そうでないなら false

matches(String target, String regex)は String の mathces メソッドを使っています。正規表現については Java の APIドキュメントを参照してください。

isNumber(String s1)は、Double.parseDouble(String s1)でパースできれば数字であるとしています。ですから、数字であっても桁を超えたりすると false になります。

after、before、between はそれぞれ、同じ形式の日付や時刻同士で比較しないとエラーになります。

例:

after('2004-12-26', '2004-2-26'); 日付同士の比較で、結果は ture

after('2004-12-26', '2004-12-26'); 同じ日付なので結果は false

after('12:00:23', '08:23:40'); 時刻同士の比較で、結果は true

after('2004-12-26 12:00:23', '2004-2-26 08:23:40'); 日付・時刻同士の比較で、結果は true

after('2004-12-26 12:00:23', '2004-2-26'); 一方が日付・時刻で他方が日付なのでエラー

・数値関数

関数の書式	内容
val(String value)	value で指定された文字列を数値(double)に変換する
length(String value)	value で指定された文字列の長さを返す

関数の書式	内容
indexOf(String s1, String s2)	1 文字列内で、s2 部分文字列が最初に出現する位置のインデックスを返す
lastIndexOf(String s1, String s2)	s1 文字列内で、s2 部分文字列が最後に出現する位置のインデックスを返します
sum(double d1, double d2, ...)	d1 から dn までの合計を返す
average(double d1, double d2, ...)	d1 から dn までの平均を返す
arraysize(String value)	value で指定された CSV 形式の文字配列の長さを返す
floor(double value)	パラメータの値以下で整数と等しい最大の値を返す （これ以降は java.lang.Math の関数とほぼ同じです）
ceil(double value)	パラメータの値以上で整数と等しい最小の値を返す
abs(double value)	パラメータの絶対値を返す
sin(double value)	正弦 (サイン) を返す
cos(double value)	余弦 (コサイン) を返す
tan(double value)	正接 (タンジェント) を返す
log(double value)	自然対数値 (底は e) を返す
exp(double value)	パラメータで累乗した指数値を返す
sqrt(double value)	平方根を返す
pow(double value1, double value2)	第 1 パラメータを、第 2 パラメータで累乗した値を返す
min(double d1, double d2, ...)	d1 から dn までの値のうちもっとも小さい値を返す
max(double d1, double d2, ...)	d1 から dn までの値のうちもっとも大きい値を返す
acos(double value)	逆余弦 (アークコサイン) を 0.0 ～ PI の範囲で返す
asin(double value)	逆正弦 (アークサイン) を -PI/2 ～ PI/2 の範囲で返す
atan(double value)	逆正接 (アークタンジェント) を -PI/2 ～ PI/2 の範囲で返す
atan2(double value1, double value2)	直交座標 (b, a) を極座標 (r, theta) に変換す
ramdom()	0.0 以上で 1.0 より小さい乱数を返す
rint(double value)	最も近い整数を double 値で返す
toDegrees(double value)	ラジアンを度に換算する
toRadians(double value)	度をラジアンに換算する

val(String value)は isNumber(String s1)と同様に Double.parseDouble(String s1)でパースした値としています。ですから、桁を超えたりすると「数値ではない」というエラーになります。

sum(double d1, double d2, ...)、average(double d1, double d2, ...)、min(double d1, double d2, ...)、max(double d1, double d2, ...)は、いくらでもパラメータを指定できます。

例：

sum(1,2,3,4,5);

は 15.0 になります。

java.lang.Math の min た max は、2つのパラメータしか指定できません。

arraysize(String value)は value を CSV 形式の文字列として扱い、それを個々の文字列の配列としたときの個数を返します。配列型のエレメントで利用できます。

例:

arraysize('ab,cd,3,4,5');

は 5.0 になります。

10.1 式の処理

式の処理には、com.fiverworks.xone.exp.Processor クラスを使います。この Processor クラスのメソッドは、次の1つだけです。

public Object evaluate(String expression)

式を文字列として与えると、Object 型の結果を返します。Object 型ですが、実際には式の値の型である java.lang.Double、java.lang.Boolean、java.lang.String のいずれかの型になります。
Processor クラスを使ったサンプルをプログラム 18 に示します。

・プログラム 18

```
package samples.exp;
import com.fiverworks.xone.exp.*;
public class Sample1 {
  public static void main(String[] args) {
    Processor proc = new Processor();
    System.out.println(proc.evaluate("sqrt(1.0 * 1.0 + 2.0 * 2.0);"));
    System.out.println(proc.evaluate("2 * PI * 3.0; // 半径3.0の円の円周"));
    System.out.println(proc.evaluate("'abc' + 'bcv';"));
    System.out.println(proc.evaluate("after(currentDate(), '2004-2-26');"));
    System.out.println(proc.evaluate("matches('12323231', '[1-3]+');"));
  }
}
```

式を処理するとき、「パラメータを入れた式を定義しておき、実行するときにはそのパラメータを実際の値で置き換えてから実行する」というのは利用する機会が多いものです。これを SQL 文で行うのが java.sql.PreparedStatement ですが、これと同様のことを行う com.fiverworks.xone.exp.PreparedExpression というクラスがあります。java.sql.PreparedStatement ではパラメータの代わりとして'?'を使いますが、Xone では'?'は 条件演算子 で使うのでこの代わりに'$'を使います。PreparedExpression クラスを使ったサンプルをプログラム 19 に示します。

・プログラム 19

```
package samples.exp;
import com.fiverworks.xone.exp.*;
public class Sample2 {
  public static void main(String[] args) {
    PreparedExpression pe =
              new PreparedExpression("2.0 * PI * $; // 半径$の円周");
    double x = 5.0;
    pe.setDouble(1, x);
    pe.setDouble(2, x);

    System.out.println(pe.getExpression());
    System.out.println("半径" + x + "の円周の長さは" + pe.evaluate() + "です.");
  }
}
```

PreparedExpression ではコンストラクタに式を渡します。このときセミコロンを忘れないでください。式の中で置き換える$はいくつであってもかまいません。このサンプルでは'$'は 2 個使ってコメントも置き換えています。

10.2 関数を作る

ユーザが独自に関数を追加できます(外部関数)。関数の追加方法は次のとおりです。

1.1つの関数に対して、1つの Java のクラスを作る

2.パラメータのないコンストラクタを作る

3.com.fiverworks.xone.exp.func.IFunction をインプリメントする

4.外部関数の定義ファイル(XML 形式)に定義を追加する

IFunction には、

Object evaluate(Object[] args)

という1個のメソッドが定義されているので、これを実装します。args というパラメータには、パラメータの配列が渡されます。ですから、その関数ではまず、パラメータの数が合っているかどうかを調べます。次に、この配列の要素の型は java.lang.Double、java.lang.Boolean、java.lang.String のいずれかのはずなので、その関数に型が合っているかどうか調べます。最後に関数の値を返します。関数の値も Double、Boolean、String のいずれかの型で返します。作成した関数の例をプログラム 20 に示します。

・プログラム 20

```
package samples.exp;
import com.fiverworks.xone.*;
public class Triangle implements com.fiverworks.xone.exp.func.IFunction {

  public Triangle() {}

  public Object evaluate(Object[] args) {
    // パラメータの個数を調べる
    if (args.length != 2)
      throw new XoneRuntimeException("パラメータの数が違います");

    // パラメータの型を調べ、数値を取得する
    double base, height;
    try {
      base = ((Double)args[0]).doubleValue();
      height = ((Double)args[1]).doubleValue();
    } catch (Exception ex) {
      throw new XoneRuntimeException("パラメータの型が違います");
    }

    // 計算
    double result = (base * height) / 2;
    return new Double(result);
  }
}
```

外部関数の定義ファイル は、XONE_HOME の config フォルダに outer-function.xml というファイルです。このファイルは次のようになっています（XML 宣言などは省略しています）。

```
<root>
        <!--functions basePath="samples.exp">
                <function name="triangle" class="Triangle"/>
                <function name="sort" class="Sort"/>
        </functions-->
</root>
```

このコメント記号を取り去り、外部関数の名前とその関数に対応する Java のクラスを function 要素に記述します。この要素の name 属性には関数名、class 属性には Java のクラス名を指定します。関数名は、組み込み関数や他の外部関数と名前が同じにならないようにしてください。functions 要素の basePath 属性には、Java のクラスのあるパスを指定します。最後に Java のクラスをコンパイルして、それをクラスパスに追加します。（注:この時点ではまだ sort 関数は作っていないはずなので、この部分はコメントにしてください）。

これで、triangle という関数は組み込み関数とまったく同様に利用でき、たとえば次のような式を処理できるようになります。

triangle(5.0, sqrt(3.0));

関数の返す値はDouble、Boolean、Stringのいずれかの型でなければならない、と書きましたが、他の関数の中や式で使わないという条件であれば返す型は何でもかまいません。プログラム21は、その関数の例です。

・プログラム21

```
package samples.exp;
import com.fiverworks.xone.*;
public class Sort implements com.fiverworks.xone.exp.func.IFunction {

  public Sort() {}

  public Object evaluate(Object[] args) {
    // パラメータの個数を調べる
    if (args.length == 0)
      throw new XoneRuntimeException("パラメータの数が違います");

    // パラメータの型を調べ、数値を取得する
    double[] values = new double[args.length];
    try {
      for (int i = 0; i < args.length; i++) {
        values[i] = ((Double)args[i]).doubleValue();
      }
    } catch (Exception ex) {
      ex.printStackTrace();
      throw new XoneRuntimeException("パラメータの型が違います");
    }

    // 処理
    java.util.Arrays.sort(values);
    return values;
  }
}
```

これを先ほどの定義ファイルに

<function name="sort" class="Sort"/>

として追加すると、sort関数ができます。この関数は、返す型がdouble[]です。プログラム22は、sort関数の使用例です。

・プログラム 22

```
package samples.exp;
import com.fiverworks.xone.exp.*;
public class Sample3 {

  public static void main(String[] args) {
    Processor proc = new Processor();
    double[] result = (double[])proc.evaluate("sort(PI, 10/3, 3.1, sqrt(10));");
    for (int i = 0; i < result.length; i++) {
      System.out.println(result[i]);
    }
  }
}
```

このプログラムのように

```
sort(PI, 10/3, 3.1, sqrt(10));
```

といった使い方はできますが、次のように他の関数や式の中に入れて使うとエラーになります。

```
min(sort(PI, 10/3, 3.1), 2.5);
sort(PI, 10/3, 3.1) + 3;
```

なお、外部関数を定義する outer-function.xml ファイルは、その定義内容が間違っている(たとえば、関数のクラス名を間違えたとかクラスパスが設定されていないなど)と、すべての外部関数は無効になります。

10.3 関数マネージャ

com.fiverworks.xone.exp.func.FunctionManager は関数を管理するクラスです。内部関数や外部関数を取得するために、次のメソッドが用意されています。

・Map getInnerFunctions()

・Map getOuterFunctions()

これらは変更不可能(unmodifiable) な関数のマップを返します。関数のマップは、関数名をキーに、関数（IFunction）を値として持っています。これを使って、関数を評価することもできます。

例:

```
Map funcMap = FunctionManager.getInnerFunctions();
IFunction func = (IFunction)funcMap.get("toUpperCase");
System.out.println(func.evaluate(new Object[]{"abc"}));
```

この例では、内部関数のマップを取得し、そのマップから toUpperCase という内部関数を取得しています。さらに、その evaluate メソッドで"abc”というパラメータで評価しています。toUpperCase は小文字を大文字に変換する関数ですから、この結果は"ABC”と表示されます。

個々の関数は

・IFunction getInnerFunction(String name)

・IFunction getOuterFunction(String name)

で取得できます。返されるのは IFunction で、そのまま関数を評価することもできます。

例:

```
IFunction func = FunctionManager.getInnerFunction("toUpperCase");
System.out.println(func.evaluate(new Object[]{"abc"}));
```

この例では、先ほどと同様に結果として"ABC”と表示されます。

関数を新たに作成したり、あるいは変更した場合、Xone を再起動しなくても init メソッドを呼び出すと再初期化できます。

例:

```
FunctionManager.init();
```

これによって、outer-function.xml が再度読み込まれ、初期化されます。外部関数の開発時などに使うとよいでしょう。

11 ユーティリティクラス

com.fiverworks.xone.ut パッケージには、次の4つのユーティリティクラスがあります。

・ValueUt

・TextUt

・MatchUt

・GuiUt

11.1 ValueUt

エレメントの値を扱うときに、内部でも利用しているクラスです。このクラスには、値の変換、CSV(カンマセパレーティッドバリュー。カンマ区切り値)関連のメソッドがあります。CSV にはいくつかバリエーションがありますが、ValueUt で保証しているのはあくまでもこのクラスで作成した CSV 形式の文字列と文字列配列の変換です(ほとんどの CSV 形式は扱えると思います)。

プログラム 23 は、ValueUt の CSV 関連のメソッドを使用した例です。

・プログラム 23

```
package samples.ut;
import com.fiverworks.xone.ut.*;
public class Sample1 {

  public static void main(String[] args) {
    String[] values = new String[] {"abc", "de,ef","gh"};
    String csv = ValueUt.toCsv(values);
    System.out.println(csv);

    String[] array = ValueUt.toArray(csv);
    for (int i = 0; i < array.length; i++) {
      System.out.println(array[i]);
    }
  }
}
```

この出力結果は次のようになります。

```
abc,"de,ef",gh
abc
de,ef
gh
```

プログラム 24 は、ValueUt のさまざまなメソッドを使用した例です。

・プログラム24

```
package samples.ut;

import com.fiverworks.xone.ut.ValueUt;
import com.fiverworks.xone.model.XoneModel;

public class Sample2 {

	public static void main(String[] args) {
		// convertメソッド
		Float f1 = (Float)ValueUt.convert("2.0", XoneModel.FLOAT);
		Float f2 = (Float)ValueUt.convert("3.2f", Float.class);
		System.out.println(f1);
		System.out.println(f2);

		// convertメソッド
		String[] iarr = new String[]{"2","3","4"};
		Integer[] ia1 = (Integer[])ValueUt.convert(iarr, XoneModel.INT_ARRAY);
		Integer[] ia2 = (Integer[])ValueUt.convert(iarr, Integer[].class);
		System.out.println(ia1);
		System.out.println(ia2);

		// toStringArrayメソッド
		String[] sarr = ValueUt.toStringArray(new int[]{2,3,4});
		for (int i = 0; i < sarr.length; i++) {
			System.out.println(sarr[i]);
		}

		// toXxxArrayメソッド
		String[] sfarr = new String[]{"2.0","3.2f","4.5"};
		float[] farr = ValueUt.toFloatArray(sfarr);
		for (int i = 0; i < farr.length; i++) {
			System.out.println(farr[i]);
		}

		// isLegalXxxFormatメソッド
		System.out.println(ValueUt.isLegalDateFormat("2004-12-26"));
		System.out.println(ValueUt.isLegalDateFormat("2004-15-1"));
		System.out.println(ValueUt.isLegalTimeFormat("12:31:01"));
		System.out.println(ValueUt.isLegalTimeFormat("12:71:01"));
		System.out.println(ValueUt.isLegalTimestampFormat("2004-12-26 12:31:01"));
	}
}
```

11.2 TextUt

テキスト関連のユーティリティクラスです。MD5 で文字列を暗号化したり、テキストの圧縮・Base64 でのエンコード、テキスト内の文字列の置き換えなどのメソッドがあります。

プログラム 25 は、TextUt の圧縮・Base64 関連のメソッドを使用した例です。

・プログラム 25

```
package samples.ut;
import com.fiverworks.xone.ut.*;
public class Sample3 {

  public static void main(String[] args) {
    byte[] values = new byte[] {12, 34, 56};
    String packed = TextUt.pack(values);
    System.out.println(packed);

    byte[] unpacked = TextUt.unpackToByteArray(packed);
    for (int i = 0; i < unpacked.length; i++) {
      System.out.println(unpacked[i]);
    }
  }
}
```

この例では、圧縮する前のデータが短いので圧縮の効果はありませんが、画像や音楽などのバイトデータをエレメントの値としたり、逆にそれから取り出すときなどに利用できます。データが大きいときは、byte 配列でエレメントに入れるよりも、いったん圧縮して文字列としてエレメントに入れた方が効率がよいです。

11.3 MatchUt

エレメントの条件式に関連するユーティリティクラスです。エレメントの条件式では、オブジェクトの読み込みの条件式と同じで、エレメント名に#を付けるとその値と見なされます。プログラム 26 は、select メソッドの使用例です。

11.4 GuiUt

GUI を利用するアプリケーション用のユーティリティクラスです。ダイアログ関連、表示位置の設定などのメソッドがあります。

・プログラム 26

```
package samples.ut;
import java.util.*;
import com.fiverworks.xone.ut.*;
import com.fiverworks.xone.model.*;

public class Sample4 {

  private static XoneObject[] makeObj() {
    java.util.List objs = new ArrayList();
    String[] colors  = new String[]{"赤", "白"};
    for (int i = 0; i < 5; i++) {
      String name = "temp" + Integer.toString(i);
      XoneAnyInstance xai = XoneModel.newXoneAnyInstance("Test", name);
      xai.addElement("id", XoneModel.INT, Integer.toString(i));
      xai.addElement("色", XoneModel.STRING, colors[i % 2]);
      System.out.println(xai);
      objs.add(xai);
    }
    return (XoneObject[])objs.toArray(new XoneObject[0]);
  }

  public static void main(String[] args) {
    XoneObject[] xos = makeObj();
    XoneObject[] selected = MatchUt.select(xos, "#id >= 2 && #id <= 4 && equals('#色', '
赤');");
    for (int i = 0; i < selected.length; i++) {
      System.out.println(selected[i]);
    }
  }
}
```

12 オブジェクトチューザ

javax.swing にはファイルを選択するためのダイアログである JFileChooser というクラスがあります。Xone でもこれと同様にサーバ上のオブジェクトを選択する ObjectChooser というクラスが com.fiverworks.xone.tools.man パッケージにあります。これは、モデルマネージャを作る過程で作られたものですが、通常の GUI アプリケーションからも利用できます。プログラム 27 は ObjectChooser を使ったサンプルプログラムです。

実行すると、次の図のようなダイアログが表示されるので、適当なオブジェクトを選択して「開く」ボタンをクリックしてみてください。選択したオブジェクトが読み込まれて表示されます。

ObjectChooser は内部で MwMain を使うので、ObjectChooser のコンストラクタには必ずログイン済みの MwMain のインスタンスを渡します。また、このプログラムでは、GuiUt の setCenterOfParent メソッドを使って、親コンポーネントの真ん中に表示されるように位置を設定しています。この例のように親を null にするとフルスクリーンの真ん中に表示されます。

・プログラム27

```
package samples.tools;
import com.fiverworks.xone.mw.*;
import com.fiverworks.xone.model.*;
import com.fiverworks.xone.*;
import com.fiverworks.xone.tools.man.*;
import com.fiverworks.xone.ut.*;

public class Sample1 {
  private MwMain main;

  private void login() {
    main = new MwMain();
    try {
      main.login("name", "password".toCharArray());
    } catch (XoneException ex) {
      ex.printStackTrace();
    }
  }
  private void test() {
    login();
    try {
      java.awt.Frame parent = null;
      ObjectChooser oc = new ObjectChooser(parent, main);
      GuiUt.setCenterOfParent(oc, parent);
      int result = oc.showOpenDialog();
      if (result == ObjectChooser.APPROVE) {
        String objPath = oc.getSelectedObject();
        XoneObject[] xos = main.load(new String[]{objPath});
        System.out.println(xos[0]);
      }
    } catch (XoneRuntimeException ex) {
      ex.printStackTrace();
    } finally {
      if (main.isLogin()) main.logout();
    }
  }

  public static void main(String[] args) {
    new Sample1().test();
  }
}
```

13 ログイン／ログアウトとセッション管理

これまでログインするには MwMain の login メソッドを使ってきましたが、MwMain には GUI アプリケーションのための showLoginDialog メソッドも用意されています。メソッドは次のとおりです。こ

・showLoginDialog(java.awt.Frame parent)

このメソッドのパラメータ parent には、親のフレームを渡します。そうすると、親フレームの中央に次のようなダイアログが表示されます。

このフレームに null を渡すとフルスクリーンの中央に表示されます。

login や showLoginDialog メソッドは、ログインしたユーザを XoneUser で返します（これはログインした後に getLoginUser メソッドでも取得できます）。

13.1 MwMain の例外

Xone には、例外として XoneException と XoneRuntimeException の 2 種類あります。そのうち、XoneException はキャッチしなければならない例外で、XoneRuntimeException は実行時例外です。

Xone では、ほとんど XoneRuntimeException を使っており、XoneException を投げるのは、MwMain の login と showLoginDialog メソッドだけです。「8.2　例外について」で記述したように、XoneRuntimeException はエラー番号、エラーのレベル、リトライ可能かどうかといった属性を持っています。エラーのレベルには、XoneRuntimeException で次の定数で定義されています。

・WARNING

深刻ではないエラー。たとえば、指定されたフォルダが見つからない等。

・ERROR

深刻なエラーの可能性があるもの。Xone 以外のライブラリ等から投げられるエラーで深刻かどうかの判断ができないもの。

・FATAL

深刻なエラー。システムが正常に動作を続けられないと思われるエラー。

アプリケーションを開発する際は、これらのエラーレベルによってエンドユーザが適切な処置ができるようにしてください。

エラー番号は、1000 番台が使われています（付録を参照してください）。リトライ可能かどうかというのは、ビジネスロジックに関連するもので、MwMain を利用するアプリケーションでは気にする必要はありません。

13.2 セッション管理

MwMain にログインした後は、サーバ側でそのセッションが管理されます。このセッション情報は、MwMain で logout メソッドでログアウトすると削除されます。そのため、アプリケーションは例外などが発生した際でも、必ずログアウトしてから終了してください。パーソナル版の場合、再度ログインすると前回のセッションは削除されます。

14 パフォーマンスについて

Xone は階層的にオブジェクトを管理していること、LOB（Large Object）でオブジェクトを保存していることなどから、RDB を素直に使ったアプリケーションよりどうしてもパフォーマンスが落ちる場合があります。ここでは、パフォーマンスを上げるためのさまざまな設定や方法について解説します（「8 コマンドとビジネスロジック」も参照してください）。

14.1 hint 属性

Xone のオブジェクトには hint という文字列の属性があります。これはサーバ上のオブジェクトを検索（読み込みなど）する際に利用できます。「3.2.2　オブジェクトの読み込み方法」で SQL の where 句に相当する条件式とエレメントの条件式を記述できる

```
load(String parent, String where, String elementCondition)
```

というメソッドについて記述していますが、このときに hint に適切な文字列を入れてあれば、HINT_FIELD として where 句に相当する条件式に指定できます。この検索の処理は、where 句に相当する条件式は RDB 上で行われ、エレメントの条件式はビジネスロジック層で行われます。RDB で検索したものをビジネスロジック層でさらに絞り込んでいます。このため場合によっては、ビジネスロジック層に負荷がかかります。hint に入れた文字列は RDB で検索できるので、その分ビジネスロジック層の負荷を低減できます。
なお、hint で使っている RDB のフィールドは 255 文字までの可変長文字列（varchar）です。これ以上の長さの文字列を指定すると、SQL の例外が投げられます。

14.2 設定

Xone の設定情報は、XONE_HOME の config フォルダ内の xone.properties に記述されています。この設定内容によっても、パフォーマンスは変わることがあります。設定内容は次のとおりです。

・invoke.compression.size

クライアントとサーバの間ではデータを圧縮して送受信しています。この属性は、圧縮を開始するデータサイズをバイト単位で指定するものです（このサイズより小さい場合は圧縮しません）。比較的遅いネットワークの場合は、この値を小さめにしたほうがよいでしょう。反面、あまり小さくすると少量のデータでも圧縮してしまい、そのためパフォーマンスが落ちることがあります。デフォルトは 16000（バイト）です。

・db.limit

RDB からロードするとき、一度に読み込めるオブジェクトの数（limit 値）を指定します。デフォルトは 30 です。

・db.user.cache.size

ビジネスロジック側でキャッシュするユーザの数を指定します。デフォルトは 20 です。

・db.fixed.folders

この属性に設定されたフォルダは、システムの起動時にフォルダ情報がキャッシュされてアクセス

は高速になります。ただし、ここに指定したフォルダは削除したり、移動したりすることはできなくなります。カンマで区切って複数のフォルダを指定できます。

例:

```
db.fixed.folders=root/classes,root/instances/master
```

この例では、root/classes と root/instances/master という2つのフォルダを固定フォルダとして起動時に読み込まれます。また、root と root/system は必ず固定フォルダになります。起動時に存在しなかったフォルダはキャッシュされません。

・db.retry.count、db.retry.interval

はリトライ可能な例外が投げられた場合、サーバ側は db.retry.count で設定したリトライの回数だけリトライします。リトライする間隔は db.retry.interval で m 秒単位で設定します。

ユーザにとっては、すぐに例外を返されるよりはこの方が使いやすい場合があります。リトライする必要がなければ、回数を 0 に設定します。リトライ回数は 20 以上の数値を指定しても 20 となります。また、間隔は 20000（20 秒）以上を指定しても 20000 になります。デフォルトはそれぞれ 3 と 2000（2 秒）です。

・router

router については、インストールマニュアルを参照してください。

・check.session

セッションのチェックを厳密に行うかどうか指定します。厳密に行うと、サーバの負荷が高くなります。デフォルトは true（厳密に行う）です。

・auto.session.control、session.send.interval、session.check.interval、session.time.out

auto.session.control を true にすると、ログアウトせずにクライアントを終了してしまった、あるいはクライアントのプログラムがハングアップしたときなど、ある一定の時間がたつと自動的にログアウト処理を行うようになっています。これを false にした方がサーバの負荷が低減されます。

session.send.interval、session.check.interval、session.time.out は、auto.session.control が true のときだけ有効です。session.send.interval にはクライアントからセッション情報を送る間隔（秒単位）、session.check.interval はサーバでセッション情報を調べる間隔（秒単位）、session.time.out はセッションのタイムアウト（秒単位）をそれぞれ設定します。session.time.out には、session.send.interval の 2～5 倍程度の値を設定してください。

なお、システムの設定値はクライアントとサーバの両方が参照します。ホストが異なる場合は、必ず xone.properties を同じ値に設定してください。

14.3 システム構成

Xone ではさまざまな形態の多層システムに対応しています。その分柔軟性はありますが、ネットワークを経由することが多くなると、パフォーマンスはどうしても悪くなります。アプリケーションサーバや Web サービスを利用するときは、データベースサーバと物理的に同じマシンにするとパフォーマンスは上がります。

なお、アプリケーションサーバや Web サービスのサーバをクラスタリングする場合、MwMain の reInit メソッドで再初期化はうまくいかないことがあります。クラスタリングされている場合、すべての

マシンで reInit が実行さるわけではないからです。このメソッドは開発時にのみ利用し、実運用では使わないでください。

14.4 クライアントアプリケーション

クライアントアプリケーションでパフォーマンスを上げたいときは、次のような方針で開発するとよいでしょう。

・アプリケーションに必要な Xone のクラスは最初に読み込んでしまう

・MwMain のメソッドを何度も呼ぶ出すよりもバッチで処理する方法を考える

付録　エラー番号一覧

システムで定義されているエラー（例外）とその番号は以下のとおりです。

1000:ログイン名またはパスワードが違います
1001:このフォルダはすでにロックされています
1002:すでにログインしています
1003:このフォルダはロックされていません
1004:コマンドが見つかりません。または実行権限がありません
1005:config ファイルに間違いがあります
1006:このグループはユーザのデフォルトのグループになっているので削除できません
1007:デフォルトのグループが見つかりません
1008:このフォルダはすでに削除されたフォルダです
1009:受け側が送り側のサブフォルダです
1010:DB の例外です
1011:DB の初期化ができません
1012:指定された DB が見つかりません
1020:クラスの Element のタイプと異なるタイプです
1021:0 で除算しました
1022:すでに同じ名前のコマンドが定義されています
1023:すでに同じ名前の XoneElement があります
1024:すでに同じ名前の XoneElementList があります
1025:すでに同じ名前のフォルダがあります
1026:すでに同じ名前のグループがあります
1027:すでに同じ名前でログインしています
1028:すでに同じ名前のオブジェクトがあります
1029:親フォルダが 2 個以上あります。システムに矛盾があります
1030:すでに同じ名前のユーザがいます
1031:指定された XoneElement が見つかりません
1032:指定された XoneElementList が見つかりません
1033:固定フォルダが含まれているのでこの操作はできません
1034:フォルダが見つかりません
1035:関数が見つかりません
1036:グループが見つかりません
1037:クラス名が不正です
1038:この文字は条件式に含めることはできません
1039:日付の形式が違います（yyyy-mm-dd で指定してください）
1040:XoneElement の名前が不正です
1041:式が不正です
1042:XoneFolder のクラスではありません
1043:形式が違います
1044:グループ名が不正です
1045:XoneGroup のクラスではありません
1046:XoneElementList の名前が不正です
1047:ロックのモードが不正です
1048:名前が不正です
1049:XoneObject のステータスが不正です
1050:XoneObject のタイプが不正です
1051:XoneObjectInfo のクラスではありません
1052:配列が空です
1053:orderBy の指定が不正です
1054:パラメータの指定が違います

1055:パラメータの数が不正です
1056:パラメータのタイプが不正です
1057:パスワードが不正です
1058:パスが不正です
1059:リード・ライトモードが不正です
1060:結果のクラスが不正です
1061:結果の値が不正です
1062:ロール名は Admin か User でなければなりません
1063:XoneSession のクラスではありません
1064:タイプが不正です
1065:ユーザ名が不正です
1066:XoneUser のクラスではありません
1067:値が不正です
1068:バージョンが違います
1069:ライトモードが不正です
1070:xml の名前空間が違います
1071:このバージョンの xml(xoml)は処理できません
1072:同じ名前のオブジェクトが含まれています
1073:指定されたオブジェクトに無効なオブジェクトが含まれています
1074:フォルダ構造に矛盾があります
1075:ステータスに矛盾があります
1076:初期化エラーです
1077:内部エラーです
1078:IO(入出力)エラーです
1079:Java のクラスが見つかりません
1080:JDOM の例外です
1081:このフォルダは他からロックされています
1082:他からロックされたオブジェクトがあります
1083:この親フォルダは他からロックされています
1084:ミドルウェア(MwMain)が null です
1085:結果が帰ってきません
1086:タイプが配列ではありません
1087:削除されたフォルダではないか、無効なフォルダです
1089:このフォルダは編集不可です
1090:未実装
1091:論理式ではありません
1092:数値ではありません
1093:プリミティブなタイプではありません
1094:読み込み権限がありません
1095:書き込み権限がありません
1096:コマンドオブジェクトが null です
1097:パラメータが null または空です
1098:パスワードが null です
1099:セッション ID が null です
1100:ユーザ名が null です
1101:null 値は許されません
1102:指定されたオブジェクトが見つかりません(あるいは無効なオブジェクトです)
1103:O/R マッピングツールの例外です
1104:O/R マッピングツールの初期化エラーです
1105:範囲を超えています
1106:自身のセッションは削除できません
1107:自分自身は削除できません
1108:親フォルダが見つかりません。システムに矛盾があります

1109:文法に間違いがあります
1110:remove はサポートしていません
1111:送り側と受け側が同じフォルダです
1112:ログインしていません
1113:システムの例外です
1114:システムフォルダなのでこの操作はできません
1115:システムのエラーです
1116:ロードするオブジェクトが多すぎます。
1117:セッションが多すぎます
1118:XoneClass をロードできませんでした
1119:以下の例外が発生したので、固定フォルダに登録できませんでした
1120:この操作はサポートしていません
1121:ユーザが見つかりません
1122:XoneClass が null です
1123:必要な XoneClass がロードされていません
1124:XoneElement が見つかりません
1125:XoneFolder が null です
1126:Xone システムのエラーです
1130:ルータの指定が不正です
1131:ルータのインスタンスを生成できません
1132:以下の例外が発生したので、サーバと通信できません。
1133:DTD ファイルが見つかりません
1134:この SystemID の DTD が見つかりません
1135:クラスにアクセスできません
1140:xone.properties ファイルのプロパティが不正です(プロパティ名 :
1141:config 関連のファイルが見つかりません。
1142:ユーザ数が多すぎます。
1143:他からロックされているフォルダがあります
1144:削除されたオブジェクトではないか、無効なオブジェクトです
1145:フォルダの所有者になっているので削除できません
1146:null の要素が含まれています
1147:XoneObject ではない要素が含まれています
1148:ログイン中です