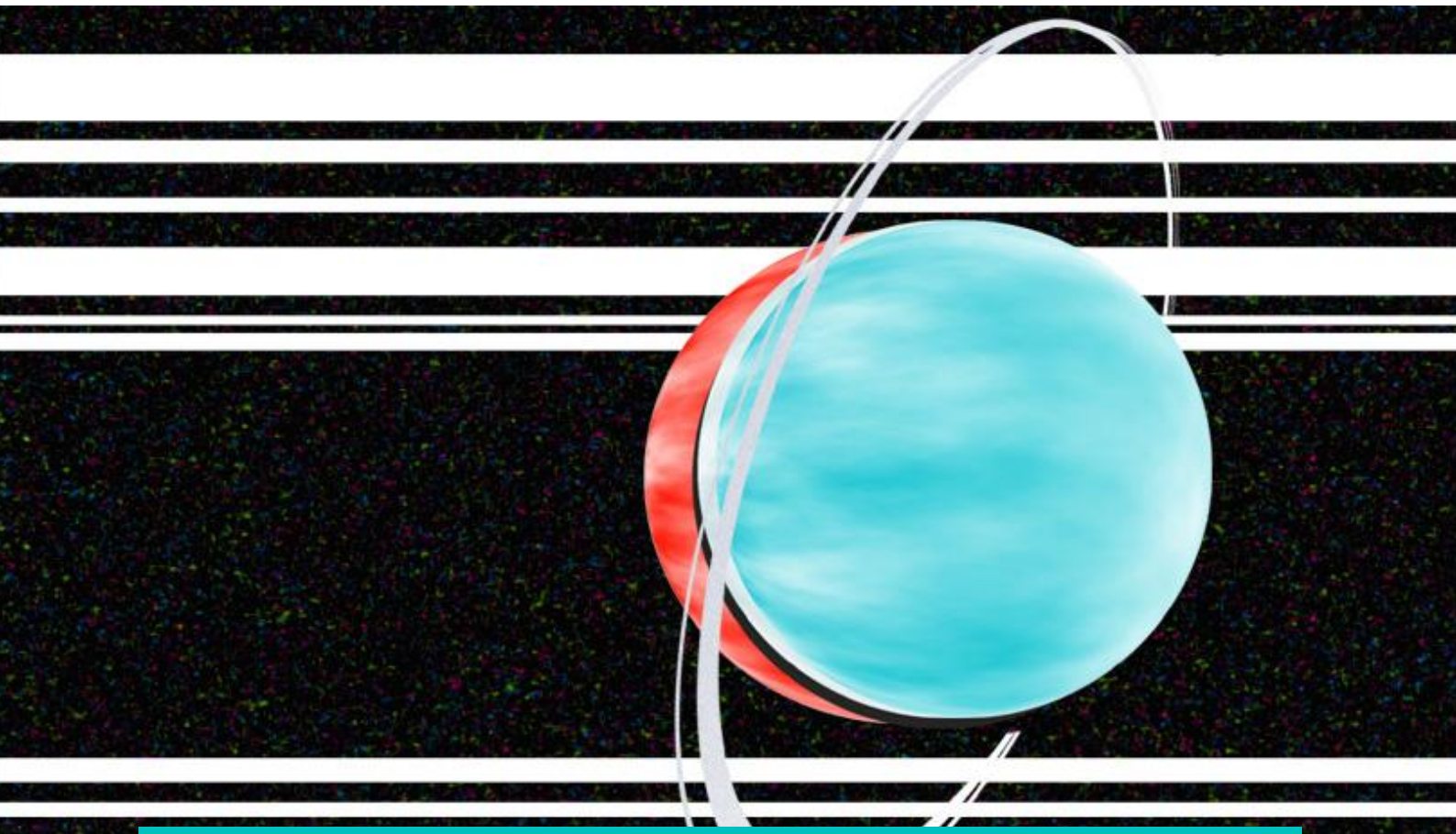
Recipes from the Uranoplums library



Uranoplums クックブック

syany 著

Uranoplums
クックブック

syany 著

まえがき

本書は、uranoplums（ウラノプラムスと読みます）に含まれているライブラリ群に関する情報を提供します。Uranoplumsは、2015 年 7 月に製作者が一般公開したばかりで、あまり世の中には知られておりません。

このコンポーネントには製作者たちが現場において経験し、実際に何度も実装を必要とされたクラスが集められています。標準的なJava API機能の拡張は、Jakarta Commonsを始めとした多くの世に広まっているコンポーネントによって解決できると考えていますが、今までのコンポーネントよりも自由度がなく、より現場よりです。

アーキテクチャ実装を頼まれて

中堅社員になったところで、あなたはプロジェクトマネージャにこういわれます。

「このシステムのアーキテクチャ設計とその実装を担当してもらいます。」

アプリケーションのアーキテクチャ設定のことです。他にも方式設計、ソフトウェアの基盤部分の設計、フレームワーク方式設計もしくはもっと広範囲の意味のシステムの共通部品の設計と実装などが同じ意味で使われています。

設計と実装の内容は、どのプロジェクトも大体決まっていて、ログ、メッセージ、バリデーション、例外、リソース、（Webシステムであれば更にフィルター、トークン、ハンドラ、セキュリティ）（DBMならば、トランザクション）などが代表的な作業スコープです。

同じような作業スコープである理由は、業務が直接関わりあわないためです。開発者や運用向けに考えられていることがほとんどです（今後の拡張もふくめ）。各機能も例えば以下のような要件がよくあがります。

- ログの出力形式にセッションIDを含めてほしい。
- メッセージはコードで管理し、コードに対応する内容はプロパティファイルや永続層に記載し、追加しやすくしたい。
- 日本独自の対応を行いたい（和暦対応、エンコード対応等）
- コーディングしやすいように、共通ユーティリティを増やしたい。（暗号化等）
- 管理しやすいように、Actionクラスの共通クラスを設けたい。
- 業務例外を作ること。

これらは、Jakarta Commons等の共通コンポーネントを用いて実現可能ですが、世に出ているものは様々な状況を加味しているせいか、とても汎用的で、どのプロジェクトでもほぼいちから作りなおすことになります。

そこでUranoplumsは、汎用的すぎている箇所を限定し、現場で起こりえる最低限の業務要件の違いに絞込み、いちから作り直すことなくアーキテクト設計、実装が行えるようになコンポーネントを提供しています。

本書の内容

本書は、Uranoplumsのコンポーネントについてを掘り下げて説明します。環境準備から始まり、プロジェクト毎に必要と思われるカテゴリ別に分けた以下の章で詳細に取り上げます。

第 1 章 開発の準備

本章では、uranoplumsを使うための開発環境を整えます。IDEにはEclipseを用いて行います。

第 2 章 標準の拡張

JDK、Jakarta Commonsを用いた機能、無かった標準機能の拡張機能について紹介します。

第 3 章 アーキテクト

アプリケーションのインフラ機能。ほとんどの現場で必要とされていましたが、今までのコンポーネントには提供されていなかったUranoplumsに含まれるインフラ機能についてを取り上げます。

第 4 章 日付・時刻

本章では、Java標準の日付、時間操作を拡張し、シンプルで、再拡張性の高いクラスについて紹介します。

第 5 章 文字列

本章では、Commons Lang、Seasar2、Terasolnaの各コンポーネントに含まれる文字列操作を拡張し、追加した機能の紹介や、国際化対応。文字エンコードに関するクラスについて紹介します。

第 6 章 その他

その他今までの章に入りきらなかった、もしくはカテゴリとしては入らなかった便利な機能についてを紹介します。

本書の記法

本書は、いくつか決まった表記を使用します。例えばコード内の文字を示す場合は等幅フォントを使用して表現します。また、特に重要なコードやコメントについては、太字を使用して目立たせます。サンプルコードの多くはすべてを記載せず、そのレシピに関連する部分だけを表示します。コードが省略されるときは、省略記号（…）を表記します。

本書では、以下の書体を使用します。

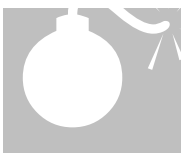
太字（ゴシック）：本文中で特に重要な箇所など、強調するときに使用します。

等幅：コマンド、オプション、コード、パラメータ、外部リソースファイル内容などを表します。

注意や警告を示します。レシピについての注意事項や実装する上で気をつけて置かなければならないことがある場合に、そのポイントについて説明しています。

インフォメーションを示します。一般的で、開示しておくべき情報であった場合、予めその内容を公開します。

引用を示します。本書、または引用文献に含まれている内容をそのまま記載する場合、こちらのアイコンで説明します。



エラー、罠を示します。対象のレシピや解説内容を使用した際に、起こりうるエラーや例外について、予防を促す意味合いとして予め記載しています。

ライセンス

本書で紹介しているUranoplumsはApache software License 2.0 に基いています。また、Uranoplumsが使用しているコンポーネントについては、次のライセンスにて公開しています。再配布の際の注意点とともに、各ライセンスについて説明します。

The Apache Software License, Version 2.0

Apache License（アパッチ・ライセンス）は、Apacheソフトウェア財団 (ASF) によるソフトウェア向けライセンス規定。Version2.0 では、全ファイルへのライセンス表示は不要としている。Uranoplumsで使用しているコンポーネントでは次が該当する。

- Google-gson
- Jakarta Commons Codec
- Jakarta Commons Collections
- Jakarta Commons Lang
- Seasar2
- Terasolna

Eclipse Public License v1.0

Eclipse Public License (EPL)は、ビジネス向けのフリーソフトウェアライセンスの１つとして設計され、GPLなどの同時期のライセンスより、提供の際のコピーレフト性が弱められている。EPLライセンスされたプログラムの受領者は、使用・修正・コピーや、修正したバージョンの配布ができる。 しかし、修正したバージョンを配布する場合はソースコードの入手方法を示すなどの義務が生じる。Uranoplumsで使用しているコンポーネントでは次が該当する。

- Junit
- Logback

MIT License

MIT License（エム・アイ・ティー ライセンス）は、マサチューセッツ工科大学を起源とする代表的なソフトウェアライセンスである。コピーレフトではなく、オープンソースであるかないかに関わらず再利用を認めている。

このソフトウェアを誰でも無償で無制限に扱って良い。ただし、著作権表示および本許諾表示をソフトウェアのすべての複製または重要な部分に記載しなければならない。しかし、作者または著作権者は、ソフトウェアに関してなんら責任を負わない。Uranoplumsで使用しているコンポーネントでは次が該当する。

- Slf4j

The BSD 3-Clause License

BSD License（ビーエスディー ライセンス）は、フリーソフトウェアで使われているライセンス体系のひとつ。カリフォルニア大学によって策定され、同大学のバークレー校内の研究グループ、Computer Systems Research Groupが開発したソフトウェア群であるBSDなどで採用されている。

「無保証」であることの明記と著作権およびライセンス条文自身の表示を再頒布の条件とするライセンス規定である。この条件さえ満たせば、BSDライセンスのソースコードを複製・改変して作成したオブジェクトコードをソースコードを公開せずに頒布できる。三条項BSDライセンスは以下の３条のことをいう。

- ソースコードを再頒布する場合、上記の著作権表示、本条件一覧、および下記免責条項を含めること。
- バイナリ形式で再頒布する場合、頒布物に付属のドキュメント等の資料に、上記の著作権表示、本条件一覧、および下記免責条項を含めること。
- 書面による特別の許可なしに、本ソフトウェアから派生した製品の宣伝または販売促進に、<組織>の名前またはコントリビューターの名前を使用してはならない。

Uranoplumsで使用しているコンポーネントでは次が該当する。

- Hamcrest

サンプルコードの利用

本書のコードは、大部分をそのまま使用するのでない限り、皆さんのプログラムや、ドキュメントに利用することが出来ます。

ただし、サンプルコードをCD-ROM媒体などを利用して販売したり、配布したりする場合には許可が必要です。本書についての質問に回答したり、サンプルコードを引用する場合の許可は必要ありません。

目次

第1章
開発の準備

さぁ、Uranoplumsを始めよう！

レシピ1.1 はじめに
レシピ1.2 Gradleを使用して環境構築する

要望

使用しているIDE環境（Gradle使用）に気軽にuranoplumsを導入したい。

解決

以下のURLを追加し、依存関係を更新します。
uranoplumsへのURL：
`https://github.com/uranonplums/uranoplums/tree/master/build/maven`

例 1-1 build.gradelの設定サンプル

```
// リポジトリURLの追加
repositories {
    maven {
        url "https://github.com/uranoplums/uranoplums/tree/master/build/maven"
    }
    mavenCentral()
    mavenLocal()
}
// 依存関係の設定方法
dependencies {
    compile group: 'org.uranoplums', name: 'uranoplums', version: '1.+'
}
```

（Eclipseならば、その後）プロジェクトを右クリックし、Gradle > 依存関係のリフレッシュ を実行する。

解説

この方法が、最も簡単に、且つ気軽にあなたの環境にuranoplumsを導入する推奨の方法です。製作者もこの方法でテストを行っています。

後は、設定したbuild.gradleの更新/依存関係のリフレッシュを行うことで、uranoplumsを利用するために必要な間接コンポーネントについても同時にダウンロードされます。

レシピ1.3 Mavenを使用して環境構築する

要望

使用しているIDE環境（Maven使用）でuranoplumsを導入したい。

解決

以下のURLを追加し、依存関係を更新します。
uranoplumsへのURL：
https://github.com/uranoplums/uranoplums/tree/master/build/maven

例 1-2 pom.xmlの設定サンプル

```
<dependency>
  <groupId>org.uranoplums</groupId>
  <artifactId>uranoplums</artifactId>
  <version>LATEST</version>
  <scope>compile</scope>
</dependency>
```

解説

Mavenを利用している場合も、レシピ 1.1 同様に設定ファイルにuranoplumsの依存関係を追加することで、必要情報がダウンロードされます。

レシピ1.4 手動でライブラリを指定する。

要望

使用しているIDE環境（Gradle, Mavenは未使用）でuranoplumsを道入したい。

解決

以下のURLからuranoplums開発環境をダウンロードします。
uranoplumsへのURL：
https://github.com/uranoplums/uranoplums/
次にダウンロードしたzipファイルを解凍し、以下のフォルダから、uranoplums.jarファイルを取得できます。
. ¥uranoplums¥build¥maven¥org¥uranoplums¥uranoplums¥1.x
こちらを、道入したいプロジェクトのビルド・パスに追加してください。
依存するjarは１パッケージ化されておりますので、不要です。

解説

uranoplums.jarの配置しているフォルダにソースファイルの入ったuranoplums-source.jarファイルを取得出来ます。

第2章
標準の拡張

基盤の基盤を充実させよう。

レシピ2.1 標準メソッドの自動化が既に実装済みのクラス

要望

equals, hashCode, toString, clone(シャローコピー)を自動生成されたオブジェクトが必要である。

解決

対象クラスの継承元をUraObjectにします。

例 2-1 UraObjectの継承

```
import org.uranoplums.typical.lang.UraObject;
…
public class BaseObject extends UraObject {
```

解説

tostring、equals、hashCodeはよく自動化が求められます。新規プロジェクトの度に～Builderを含めた共通クラスを作成するのは手間ですので、既に実装されているUraObjectを継承しましょう。

表 2-1 org.uranoplums.typical.lang.UraObjectのメソッド

メソッド名	内容
clone()	シャローコピー（superクラスのclone）を実行します。
equals(Object)	EqualsBuilder.reflectionEqualsを実行します。
hashCode()	HashCodeBuilder.reflectionHashCodeを実行します。
toString()	UraToStringBuilder クラスの toString メソッドを実行します。UraToStringBuilderは、出力されるオブジェクトを限定することが可能となります。
toStringFilter(String...)	引数に設定されているオブジェクト名称（正規表現可能）のみ表示します。
toMultiString()	整形されたtoStringを出力します。
toMultiStringFilter(String...)	toStringFilterの整形板です。

レシピ2.2 整形されたtoStringの生成

要望

整形されたtoStringオブジェクトが必要である。

解決

UraObjectを継承し、toMultiStringメソッドを実行します。

例 2-2 UraMultiLineStyleの使用

```
class ResultInfo extends UraObject {
    public String firstName = "テスト";
    public String lastName = "田中";
    public SubResultInfo sub = new SubResultInfo();
}
class SubResultInfo extends UraObject {
    public String comment = "趣味は散歩です。";
    public int length = 122;
}
…
ResultInfo info = new ResultInfo();
System.out.println(info.toMultiString());
```

出力結果

```
org.uranoplums.typical.lang.ResultInfo@7e959e[
  firstName=テスト
  lastName=田中
  sub=org.uranoplums.typical.lang.SubResultInfo@7e98bc[
    comment=趣味は散歩です。
    length=122
  ]
]
```

解説

Dumperのように整形されたtoStringはToStringBuilderでスタイルMultiLineStyleで実現可能ですが、対象オブジェクトのメンバにオブジェクトを持つような場合においても平坦に表示され、どのオブジェクトのメンバかわかりくくなります。

UraMultiStyleで出力した場合、オブジェクトのメンバがオブジェクトの場合、そのオブジェクトのメンバはちょうどDumperのようにインクリメントされた状態で出力されるようになります。

別解

UraMultiLineStyleをデフォルトスタイルにして使います。

例 2-3 setDefaultStyleでの使用

```
import org.uranoplums.typical.lang.builder.UraMultiLineToStringStyle;
…
class ResultInfo2 {
    public String firstName = "テスト2";
    public String lastName = "中村";
    public SubResultInfo2 sub = new SubResultInfo2();
    @Override
    public String toString() {
        // デフォルトスタイルを一時保管し、マルチスタイルに変更する
        ToStringStyle toStringStyle = ToStringBuilder.getDefaultStyle();
        ToStringBuilder.setDefaultStyle(UraMultiLineToStringStyle.INSTANCE);
        // toStringの実行
        String result = ToStringBuilder.reflectionToString(this);
        // 元に戻す。
        ToStringBuilder.setDefaultStyle(toStringStyle);
        return result;
    }
}
class SubResultInfo2 {
    public String comment = "趣味は登山です。";
    public int length = 122;
    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
```

```
}
…
ResultInfo2 info2 = new ResultInfo2();
System.out.println(info2.toString());
```

出力結果

```
org.uranoplums.typical.lang.ResultInfo2@836f27[
  firstName=テスト2
  lastName=中村
  sub=org.uranoplums.typical.lang.SubResultInfo2@83725e[
    comment=趣味は登山です。
    length=122
  ]
]
```

Common Langの提供するToStringBuilderクラスのreflectionToStringは、メンバ変数のtoStringはオブジェクトに依存します。メンバオブジェクトについても整形したい場合、メンバオブジェクトのtoStringメソッドにおいても、同様にreflectionToStringを実装する必要があります。

レシピ2.3 出力されるtoStringオブジェクトを限定

要望

出力されるtoStringオブジェクトを限定する必要がある。

解決

UraToStringBuilderのsetIncludeFieldNamesPerttern利用します。

例 2-4 setIncludeFieldNamesPerttern利用

```
import org.uranoplums.typical.lang.builder.UraToStringBuilder;
…
class PxxSearch {
    int pxxId = 890;
    long pxxMax = 60000;
    String pxxLimitstr = "制限付きオブジェクト";
    List<String> pxxList = newArrayList();
    {
        pxxList.add("345");
        pxxList.add("we3");
        pxxList.add("12r");
    }
}
…
final PxxSearch pxxSearch = new PxxSearch();
String output = new UraToStringBuilder(pxxSearch)
    .setIncludeFieldNamesPerttern("^pxx")
    .setExcludeFieldNames("pxxId")
    .toString();
```

出力結果

```
org.uranoplums.typical.lang.PxxSearch@81ab7f[pxxMax=60000,pxxLimitstr=制限付きオブジェクト,pxxList=[345, we3, 12r]]
```

解説

ToStringBuilderで自動化されたtoStringは出力しないものをsetExcludeFieldNamesで指定することが可能ですが、出力するものを指定することはできません。そこで、UraStringBuilderを利用しましょう。setIncludeFieldNamesPerttern は、出力対象を指定でき、さらにその中からsetExcludeFieldNamesで非出力対象

を指定することも可能です。
　また、setIncludeFieldNamesPerttern は、部分検索となっており、且つ正規表現を使用することができますので、接頭語、接尾語で限定することも出来、接頭語、接尾語の開発ルールのあるシステムで力を発揮します。

別解

　UraObject.toStringFilter(整形されたものを望むのであればtoMultiStringFilter)で同様の実装が可能。

例 2-5 toMultiStringFilterの利用

```
public class ScSearch extends UraObject {
    int scId = 890;
    long scMax = 60000;
    String scLimitstr = "制限付きオブジェクト";
    ScChildSearch scChild = new ScChildSearch();
    List<String> scList = newArrayList();
    {
        scList.add("345");
        scList.add("we3");
        scList.add("12r");
    }
}

public class ScChildSearch extends UraObject {
    int scAge = 98;
    String comment = "追い抜けない。。。";
}
…
final ScSearch scSearch = new ScSearch();
String output = scSearch.toMultiStringFilter("^sc");
System.out.println("res:");
System.out.println(output);
```

出力結果

```
res:
org.uranoplums.typical.lang.ScSearch@80c4bd[
  scId=890
  scMax=60000
  scLimitstr=制限付きオブジェクト
  scChild=org.uranoplums.typical.lang.ScChildSearch@80db1f[
    scAge=98
  ]
  scList=[345, we3, 12r]
]
```

レシピ2.4 簡易比較メソッドを生成する

要望

　自動生成された比較メソッドを使いたい。

解決

　UraDataObjectを継承し、compareToを利用します。

例 2-6 UraDataObjectを利用する。

```
import org.uranoplums.typical.lang.UraDataObject
…
class CarDataObject extends UraDataObject {
```

```
        int id = 3400;
        String name = "ken";
        UraLog log = UraLoggerFactory.getUraLog();
    }
    …
    final CarDataObject car01 = new CarDataObject();
    final CarDataObject car02 = new CarDataObject();
    System.out.println("CompareTo[car01 car02]:" + car01.compareTo(car02));
```

出力結果

```
CompareTo[car01 car02]:0
```

解説

比較はcompareToで比較します。比較メソッドについてもCommon LangのBuilderクラスで実現可能です。
比較メソッドは、特にデータオブジェクトで利用するため、UraObjectを継承した、UraDataObjectに実装されています。

UraDataObjectのcompareToは内部でCompareToBuilderを使用しています。
CompareToBuilderは、メンバ変数のcompateToを再帰的に呼び出し、全メンバの比較を行いますので、すべてのメンバ変数はComparableインタフェースを実装している必要があります。
インタフェースが実装されていない場合、例外が発生します。

レシピ2.5 簡易ディープコピーを生成する

要望

自動生成された簡易ディープコピーを使いたい。

解決

UraSerialDataObjectを継承しdeepCloneメソッドを実行します。

例 2-7 UraSerialDataObjectを利用する。

```
import org.uranoplums.typical.lang.UraSerialDataObject;
…
class ChildSerialDataObject extends UraSerialDataObject {
    private static final long serialVersionUID = -967693264092912433L;
    int id = 56;
    String name = "tom";
    UraLog log = UraLoggerFactory.getUraLog();
}

class CarSerialDataObject extends UraSerialDataObject {
    private static final long serialVersionUID = -6379534315287036367L;
    public int no = 500;
    public ChildSerialDataObject child = new ChildSerialDataObject();
    public Map<String, Object> map = newHashMap();
    {
        map.put("sheet", "parper");
        map.put("color", "orange");
    }
}
…
CarSerialDataObject car01 = new CarSerialDataObject();
CarSerialDataObject car02 = car01.deepClone();
CarSerialDataObject car03 = (CarSerialDataObject) car01.clone();
// car01 のmap内の内容だけ変更
```

```
System.out.println("car01.map.sheet: parper --> wood");
car01.map.put("sheet", "woods");
// 比較
System.out.println("equals[car01 car02]:" + car01.equals(car02) + "¥t:car02s
    sheet[" + car02.map.get("sheet") + "]");
System.out.println("equals[car01 car03]:" + car01.equals(car03) + "¥t:car03s
    sheet[" + car03.map.get("sheet") + "]");
```

出力結果

実行すると、次のように出力される。

```
car01.map.sheet: parper --> wood
equals[car01 car02]:false      :car02s sheet[parper]
equals[car01 car03]:true       :car03s sheet[woods]
```

解説

ディープコピーを自動化する手段は幾つかありますが、SerializationUtils.cloneは中でも特に簡易的な方法です。UraSerialDataObjectのdeepCloneメソッドで同様の処理を実現していまず。

SerializationUtils.clone を実現するためには、ユーティリティの名称にもあるように、オブジェクトがシリアライズ化できることが条件であるのでSerializableを実装しています。

deepCloneは、オブジェクトに存在するすべてのメンバをシリアライズでのコピーをしようと試みます。対象メンバでシリアライズ化出来ないものについては、対象外であるマークをしないと失敗しますので、**transient修飾子**をつけましょう。

レシピ2.6 業務システムでの使い分け方

要望

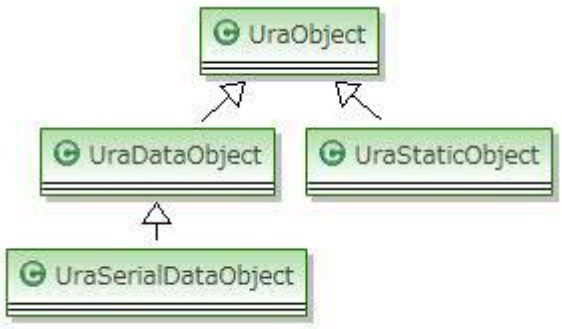
複数ある基盤オブジェクトの使い分けをしたい。

解決

例えば、次のように使い分けます。

- Bean/DTO/VOクラスのようなデータオブジェクトにはUraDataObjectを継承すると便利です。
- Actionクラスや、バッチのmainメソッドを内包しているのようなプロセスクラスには、UraObjectを継承します。
- ウエブアプリ向けのクラスで、Appサーバの再起動時のためにデシリアライズ/シリアライズでデータの消去を防ぎたい場合には、UraSerialDataObject継承します。

図 2-1 UraObjectクラスの継承関係



第3章
アーキテクト

拡張されたログやリソース操作を利用しよう

レシピ3.1 ログ名を自動生成するロガーファクトリーを使用する

要望

ログ名を自動で設定できるロガーファクトリーが必要である。

解決

UraLoggerFactoryを使用します。

```
import org.uranoplums.typical.log.UraLoggerFactory;
…
Logger logger = UraLoggerFactory.getLogger();
```

解説

ここからは、Uranoplumsが提供するアーキテクチャ機能について説明します。

最初はロギング機能です。Javaのロギング機能はlog4j、1.4 からある標準機能、Commons Loggingなどが有名ですが、Uranoplumsは最近よく使われているログlogback + slf4jを解析し、内包しています。

使用する側は、どのライブラリでもFactoryクラスを経由しインスタンス化するのが一般的ですが、ログ名を設定する必要があります（フィルターや設定などで利用するため）

名称は、使用するプロダクトごとに自由に決めても良いようにそうなっているはずなのですが、大体の現場で暗黙的にフルパッケージ名＋クラス名になってます。（機能別に設計されていればそれで十分です。）

UraLoggerFactory.getLoggerを使用することで、名前を指定することなくLoggerクラスをインスタンス化出来ます。ログ名は呼び出し元のフルパッケージ＋クラス名になります。

返却されるLoggerはSLF4JのLoggerインタフェース（実態はLogback）となっています。

レシピ3.2 メッセージでログの出力レベルを変える

要望

メッセージの内容で、ログレベルを指定するロガーが必要である。

解決

AbsUraCodeLog<E>を継承したクラスを作成し、toLevelメソッド内で出力を出し分けます。

例 3-1 AbsUraCodeLog<E>クラスの継承

```
import org.uranoplums.typical.log.AbsUraCodeLog;
…
class AutoLevelLog extends AbsUraCodeLog<String> {
...
    @Override
    public Level toLevel(String source, Level defaultLevel) {

        // デフォルトがnullならばOFF
```

```
        if (defaultLevel == null) {
            defaultLevel = Level.OFF;
        }
        // 対象が空ならば、デフォルト返却
        if (UraStringUtils.isEmpty(source)) {
            return defaultLevel;
        }

        // 先頭3文字比較
        String target = UraStringUtils.substring(source, 0, 3);
        if (UraStringUtils.equals(target, "ERR")) {
            return Level.ERROR;
        } else if (UraStringUtils.equals(target, "DBG")) {
            return Level.DEBUG;
        }
        return defaultLevel;
    }

    @Override
    public String getMessage(String source) {
        return source;
    }
}
```

例 3-2 logback.xml設定ファイルのサンプル

```
<configuration>
  <appender name="STDOUT"
    class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <param name="Pattern"
        value="TEST %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n" />
    </layout>
  </appender>

  <logger name="org.uranoplums.typical.log.AbsUraCodeLogTest" level="DEBUG">
  </logger>

  <root>
    <level value="TRACE" />
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

例 3-3 UraLoggerクラスの利用

```
// インスタンス化
UraLogger<String> logger = new AutoLevelLog(UraLoggerFactory.getLogger());
int id = 234;
// ログ出力例
logger.log("DBG デバッグ向けログです。[ID:{}]", id);
```

出力結果

実行すると、次のように出力される。

```
TEST 21:53:18.361 [Main Thread] DEBUG o.u.typical.log.AbsUraCodeLogTest - DBG デ
    バッグ向けログです。[ID:234]
```

解説

通常Loggerは、ログ出力用のメソッドがレベル別に存在し、開発者が出力レベルを指定します。これには 2 点懸念点があります。

まずは、メソッド名がレベル別に分かれていること。

次に、後から出力レベルを変えたい場合、ソースの修正が必要になってしまう。

AbsUraCodeLog<E>クラスはtoLevelメソッドによって、出力対象のメッセージから出力レベルを設定することが可能です。
上記の例は、メッセージの先頭３文字を解析してエラー、デバッグ、OFFに分けています。
その他にも、データベースを利用したりすることで、ソースコードを変えることなく出力レベルを変えることが出来ます。

表 3-1 AbsUraCodeLog<E>クラスの持つ主なメソッド

メソッド名称	内容
log(E, Object...)	第１引数の値を使用してログを出力します。
log(Marker, E, Object...)	第２引数の値を使用してログを出力します。MarkerはLogbackのMarkerクラスです。
log(Marker, Throwable, E, Object...)	第３引数の値を使用してログを出力します。
isDebugEnabled()	レベル判定の一つ。ここでは、デバッグレベルの出力が可能ならばtrueを返却します。
isEnabledFor(Level)	レベル判定。指定したレベルが出力可能であればtrueを返却します。

別解

メッセージの先頭1文字をから出力レベルを出し分けるので良いのであれば、予め作成されているUraStringCodeLogクラスを使用することで実現可能です。

例 3-4 UraStringCodeLogの利用

```
// インスタンス化
import org.uranoplums.typical.log.UraLoggerFactory;
…
UraLogger<String> logger = UraLoggerFactory.getUraStringCodeLog();
// または以下
//UraLogger<String> logger = new UraStringCodeLog(UraLoggerFactory.getLogger());
// ログ出力例
logger.log("D001 デバッグログ");
logger.log("E001 エラーログ");
logger.log("I001 インフォログ");
logger.log("W001 警告ログ");
```

出力結果

```
TEST 01:49:17.813 [Main Thread] DEBUG o.u.typical.log.UraStringCodeLogTest - D001
    デバッグログ
TEST 01:49:17.894 [Main Thread] ERROR o.u.typical.log.UraStringCodeLogTest - E001
    エラーログ
TEST 01:49:17.895 [Main Thread] INFO  o.u.typical.log.UraStringCodeLogTest - I001
    インフォログ
TEST 01:49:17.899 [Main Thread] WARN  o.u.typical.log.UraStringCodeLogTest - W001
    警告ログ
```

レシピ3.3 出力メッセージ内容の管理方法を自由にする

要望

指定したメッセージを利用して、出力メッセージの管理方法を変えたい。

解決

AbsUraCodeLog<E>を継承したクラスを作成し、getMessageメソッド内で外部で管理しているリソースを取得します。

例 3-5 getMessageの実装

```
import org.uranoplums.typical.log.AbsUraCodeLog;
import org.uranoplums.typical.resource.UraJSONResource;
…
```

```
class CodeMessageLog extends AbsUraCodeLog<String> {

    private UraJSONResource messageResource = new UraJSONResource("message");
…
    @Override
    public String getMessage(String source) {
        return this.messageResource.getResourceString(source);
    }
}
```

例 3-6 message.jsonの例

```
{
  ERR002 : "_/_/_/ ～～に失敗しました。[ID={0}] _/_/_/",
  DBG001 : "_/_/_/ ～～出力。[Object='{}'] _/_/_/"
}
```

例 3-7 logの実行

```
// インスタンス化
UraLogger<String> logger = new CodeMessageLog(UraLoggerFactory.getLogger());
int id = 234;
// ログ出力例
logger.log("DBG001", id);
logger.log("ERR002", logger.toString());
// 存在しないコード値
logger.log("ERR001", id);
```

出力結果

実行すると、次のように出力される。

```
TEST 21:05:14.730 [Main Thread] DEBUG o.u.typical.log.AbsUraCodeLogTest - _/_/_/ ～
    ～出力。[Object=234] _/_/_/
TEST 21:05:14.784 [Main Thread] ERROR o.u.typical.log.AbsUraCodeLogTest - _/_/_/ ～
    ～に失敗しました。[ID=Logger[org.uranoplums.typical.log.AbsUraCodeLogTest]]
    _/_/_/
TEST 21:05:14.817 [Main Thread] ERROR o.u.typical.log.AbsUraCodeLogTest -
```

解説

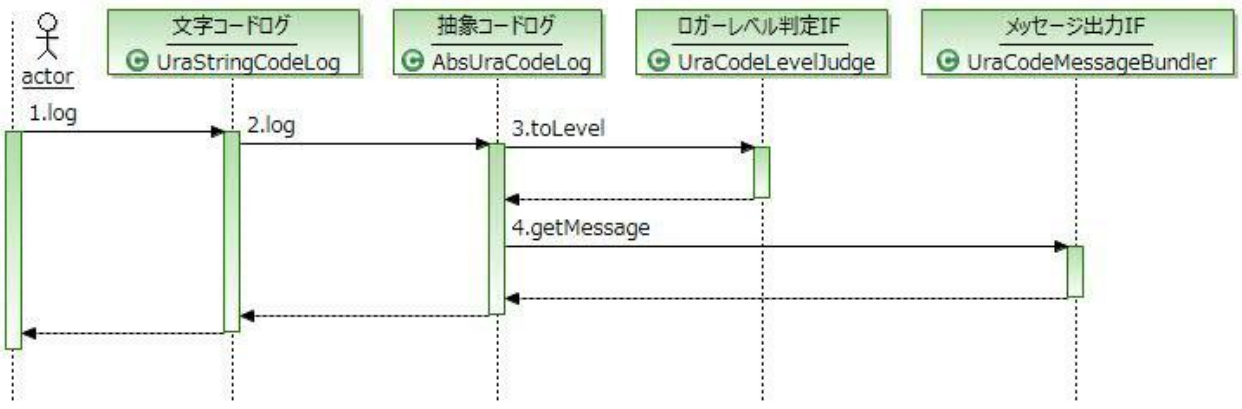
ログメッセージは、コードで指定し、内部でメッセージに展開し出力することが多いですが、展開するためのロジックは様々です。

AbsUraCodeLog<E>は予め展開用のメソッドgetMessageを用意し、logメソッドで入力したメッセージやコードを第一引数から取得し、ログ出力用の文字列へ変換し、返却させます。

上記の例では、予めリソース管理されたクラス(UraJSONResource)を使用し、message.jsonリソースを使用し展開しています。

レシピのまとめとして、シーケンスを図示します。

図 3-1 AbsUraCodeLog<E>クラスのシーケンス



レシピ3.4 MDCを管理する

要望

ログにシステムの固定値を指定する必要がある。

解決

logbackのMDC機能を利用します。

例 3-8 logback.xmlファイルの変更

```
<layout class="ch.qos.logback.classic.PatternLayout">
  <param name="Pattern"
    value="TEST %d{HH:mm:ss.SSS} T[%X{TID}] [%thread] %-5level %logger{36} - %msg%n"
    />
</layout>
```

例 3-9 MDC（TID）の実装

```
// スレッドID をTIDというキー名で MDC経由で追加する。
UraLoggingDiagnosticManager.getInstance().put("TID", String.format("%d",
    Thread.currentThread().getId()), "-- No Thread ID --");
// インスタンス化
UraLogger<String> logger = new CodeMessageLog(UraLoggerFactory.getLogger());
int id = 234;
// ログ出力例
logger.log("DBG001", id);
logger.log("ERR002", logger.toString());
// MDCで追加しているものを削除
UraLoggingDiagnosticManager.getInstance().removeAll();
```

出力結果

```
TEST 08:51:42.797 T[1] [Main Thread] DEBUG o.u.typical.log.AbsUraCodeLogTest -
    _/_/_/ ～～出力。[Object=234] _/_/_/
TEST 08:51:42.877 T[1] [Main Thread] ERROR o.u.typical.log.AbsUraCodeLogTest -
    _/_/_/ ～～に失敗しました。
    [ID=Logger[org.uranoplums.typical.log.AbsUraCodeLogTest]] _/_/_/
```

解説

ログにシステム固有の追加情報を設定することも良くありますが、uranoplums内のlogbackに関してもMDC機能を使用することができます。

MDCは、**UraLoggingDiagnosticManager**を使用することで、一元管理できます。例は、スレッドIDをTIDというキー名でputメソッドを使用して設定しています。

ligbacl.xmlでは、%Xを使用して出力位置を決めます。

最後にremoveAllで設定をクリアします。

これは、オリジナルセッションID等を設定する際にも便利です。

レシピ3.5 メッセージなどのリソースを国際化対応する

要望

リソースを簡単に国際化対応したい。

解決

UraPropertyResourceを利用します。

例 3-10 リソースmessage_test_ja.propertiesの作成

```
test.001=ようこそ！Uranoplumsの世界へ
test.002=簡易説明
```

例 3-11 リソースmessage_test.propertiesの作成

```
test.001=Wellcome to Uranoplums world!
```

```
test.003=Details...
```

例 3-12 UraPropertyResourceの利用

```
UraPropertyResource resource = new UraPropertyResource("message_test");
System.out.println(resource.getResourceValue(Locale.JAPANESE, "test.001"));
System.out.println(resource.getResourceValue(Locale.ENGLISH, "test.001"));
System.out.println(resource.getResourceValue(Locale.JAPANESE, "test.002"));
System.out.println(resource.getResourceValue(Locale.JAPANESE, "test.003"));
```

出力結果

```
ようこそ！Uranoplumsの世界へ
Wellcome to Uranoplums world!
簡易説明
Details...
```

解説

UraPropertyResourceは、propertiesファイルのキー値を取得するクラスです。一般的なResourceBundleクラスと同様に_（アンダーバー）国名（ロケール）＋.properties別に存在するキー・値を取得します。上記の例は、jaロケールのmessage_test.propertiesファイルと指定のないmessage_test.propertiesファイルを、それぞれクラスパスの設定されている位置に配置します。

UraPropertyResourceのコンストラクタ第一引数に、使用したいファイル名をロケールを含めずに、設定します。

あリソースの値はgetResourceValueで取得します。第一引数で取得対象のロケール、第二に検索キーを指定すると、対象ファイルを指定ロケール、デフォルトロケール、ロケール指定なしの順にキーを検索し表示します。

この例では、test.001～test.003 をJAPANESEロケールで取得し、一部ENGLISHロケールを使用しています。test.001 とtest.002 はmessage_test_ja.propertiesファイルに存在しますがtest.003 キーに相当する値は見つかりません。そう言った際は、先に記載したルールに基づき、message_test.propertiesにあるtest.003 キーから値を取得します。明示的にロケールを指定しても同様の動きです。test.001 はmessage_test_ja.propertiesにもmessage_test.propertiesにも存在するキー値ですから、指定したロケールで「ようこそ！Uranoplumsの世界へ」か「Wellcome to Uranoplums world!」の出力を制御できます。

表 3-2 UraPropertyResourceの主なメソッド

メソッド名称	内容
getResourceValue(Locale, String)	第 1 引数のロケールを優先的にリソースからキー値を取得します。第 2 引数はキー名です。
getResourceValue(Locale, String, Object[])	上記に第 3 引数が付いたメソッド。取得した値に、変換できるキーワードがあった場合、そのリストを指定します。
getResourceValue(Locale, String, String...)	上記のキーワードがすべてString型であった場合には、こちらを利用します。
getResourceValue(String)	ロケールを指定しない場合。Locale.defaultLocaleか、下記のsetDefaultLocaleで本クラス内限定で指定したデフォルトロケール優先となります。
getResourceValue(String, String...)	上記の取得値向け変換引数付き。
isExists(Locale, String)	指定したリソースとキーでリソースにキーが存在するかチェックします。あるばあいはtrueを返却。
setDefaultCharset(Charset)	デフォルトCharsetを設定します。
setDefaultLocale(Locale)	デフォルトLocaleを設定します。ロケール未指定で、getResourceValueを実行した場合、こちらのLocaleが優先されます。
setReturnNull(boolean)	第 1 引数をtrueで設定した場合、取得したキー値がnullであれば、nullを返却。falseである場合は、相当の値（文字列など）を返却します。

レシピ3.6 リソース内で、別に設定したキー/値を使えるリソースバンドルを使う

要望

リソースの値に他のキーを指定したい。

解決

引続き、UraPropertyResourceを使用します。

例 3-13 リソースmessage_test_ja.propertiesの追加

```
test.004=${test.mark} キーは `{0}' です ${test.mark}
test.006=${test.error} 文字は数字のみを入力してください。
test.007=${test.error} データベース接続に失敗しました。
test.error=エラーが発生しました。
test.mark=●●
test.log.dir=${test.baseDir}¥¥log
test.log.fileBase=${test.log.dir}¥¥apps_
```

例 3-14 リソースmessage_test.propertiesの追加

```
test.005=${test.mark} key is `{0}' ${test.mark}
test.mark=***
test.baseDir=c:¥¥project
```

例 3-15 UraPropertyResourceの利用

```
UraPropertyResource resource = new UraPropertyResource("message_test");
System.out.println(resource.getResourceValue("test.004", "鍵"));
System.out.println(resource.getResourceValue("test.005", "キー"));
System.out.println(resource.getResourceValue("test.006"));
System.out.println(resource.getResourceValue("test.007"));
System.out.println(resource.getResourceValue("test.log.fileBase") + "ja.log");
```

出力結果

```
●● キーは `鍵' です ●●
●● key is `キー' ●●
エラーが発生しました。 文字は数字のみを入力してください。
エラーが発生しました。 データベース接続に失敗しました。
c:¥project¥log¥apps_ja.log
```

解説

UraPropertyResourceは、国際化対応の他に、他のキーを別の値の内容に設定することができる機能を併せて内容しています。ちょうど、Jakart Antのbuild.propertiesのように設定します。

例ではtest.004の内容をキーtest.markを${}で囲み設定しています。結果、メッセージの前後にtest.markの値●●が表示されます。

また、共通の文言を、複数のメッセージに使用する場合にも便利です。test.006、test.007 はエラーメッセージでので、共通の接頭語を使ってエラーであることを説明しています。

業務では、例えばディレクトリを再利用したい場合などに使用できます。test.log.fileBaseキーは、プロジェクトのディレクトリと、ログファイ配置ディレクトリをベースに結合し、ファイルの接頭語のみ記載すれば良いように作成できます。このため、プロジェクトの配置ディレクトリを変更した際もtest.log.fileBaseの値は変更することなく反映されます。

レシピ3.7 JSON形式のリソースを利用する

要望

プロパティリソースのようにJSONを使いたい。更に国際化対応したJSON形式のリソースバンドルが必要である。

解決

UraJSONResourceを使用します。

例 3-16 message_test_ja.jsonの使用

```
{
    test.001: "文字列",
    test.002: {
        001: "-",
        002: "男",
        003: "女"
    },
    test.003: [
        10,
        20
    ]
}
```

例 3-17 UraJSONResourceの利用

```
UraJSONResource jsonResource = new UraJSONResource("message_test");
List<String> list = jsonResource.getResourceList("test.003");
Map<String, Object> map = jsonResource.getResourceMap("test.002");
String value = jsonResource.getResourceString("test.001");
System.out.println("test.003:" + list.toString());
System.out.println("test.002:" + map.toString());
System.out.println("test.001:" + value);
```

出力結果

```
test.003:[10.0, 20.0]
test.002:{001=-, 002=男, 003=女}
test.001:文字列
```

解説

プロパティファイル（.properties拡張子のファイル）は、２バイト以上の文字を扱うためには、native2asciiなどのツールを使用しエスケープする必要があり（バージョンによって解決方法もあります）管理や使用するためにはその"クセ"を覚えておかなければなりません。現在において、設定ファイルとして使用できるリソースは選択肢が増え、仕組みもxmlから、yaml、json、groovyなど簡易的で多機能な書式を持つことができるものが増えました。

uranoplumsはその中でもよく使用されているJSONについてをリソースの代わりに、また国際化できる仕組みを用意しました。それがAbsUraHierarchyResourceクラスであり、その派生クラスであるUraJSONResourceクラスです。

json拡張子を持つファイルは、コンストラクタの第一引数に拡張子なしで入力することで、UraPropertyResourceクラス同様にロケール別にキー値検索することができます。また、例にあるようにプロパティでは困難だった階層構造の取り扱いや、型の指定などを行うことが可能です。以下にUraJSONResourceの親クラスであるAbsUraHierarchyResourceのリソース取得メソッド３０種類の内、よく使う１５種について記載します。

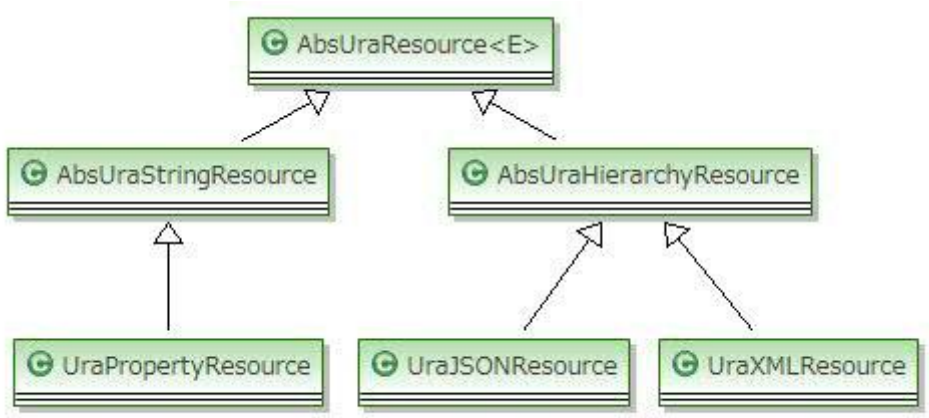
表 3-3AbsUraHierarchyResourceの主な値取得用メソッド

メソッド名称	内容
getResourceArray(Locale, String, T...)	指定したロケールとキーから対象値を配列形式で取得する。
getResourceArray(String, T...)	指定したキーから対象値を配列で取得する。
getResourceArray(T[], Locale, String, String...)	第一引数の配列に指定したロケールとキーで取得した値を追加し、返却する。
getResourceArray(T[], String, String...)	第一引数の配列に指定したロケールとキーで取得した値を追加し、返却する。
getResourceList(Locale, String, String...)	指定したロケールとキーから対象値をリスト形式で取得する。
getResourceList(String, String...)	指定したキーから対象値をリスト形式で取得する。
getResourceMap(Locale, String, String...)	指定したロケールとキーから対象値をマップ形式で取得する。
getResourceMap(String, String...)	指定したキーから対象値をマップ形式で取得する。
getResourceString(Locale, String, String...)	指定したロケールとキーから対象値を文字列形式で

	取得する。
getResourceString(String, String...)	指定したキーから対象値を文字列形式で取得する。
getResourceStringArray(Locale, String, String...)	指定したロケールとキーから対象値を文字列配列形式で取得する。
getResourceStringArray(String, String...)	指定したキーから対象値を文字列配列形式で取得する。
getResourceValue(Class<T>, Locale, String, String...)	指定したロケールとキーから対象値を第一引数の型形式で取得する。
getResourceValue(Class<T>, String, String...)	指定したキーから対象値を第一引数の型形式で取得する。
getResourceValue(Locale, String, T...)	指定したロケールとキーから対象値を返却先の型形式で取得する。

以下に、Uranoplumsの持つリソース操作管理クラスのクラス図を示します。

図 3-2 Uranoplumsの持つリソース操作クラス階層



レシピ3.8 XML形式のリソースも利用する

要望

プロパティリソースのようにxml形式のファイルも扱いたい。

解決

UraXMLResourceクラスを利用します。

例 3-18 message_test_ja.xmlの使用

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE RelativeLayout>
<resources detail="message&amp;codes">
  <test.001 value="文字列" />
  <test.002 typecode="%">
    <key>001</key><value>-</value>
    <key>002</key><value>男</value>
  </test.002>
  <!-- 別のマップ追加方法 -->
  <test.002 typecode="%" key="003" value="女" />
  <test.003 typecode="@">
    <value>10</value>
  </test.003>
  <!-- 別のリスト追加方法 -->
  <test.003 typecode="@" value="20" />
</resources>
```

例 3-19 UraXMLResourceの利用

```
UraXMLResource xmlResource = new UraXMLResource("message_test");
List<String> list = xmlResource.getResourceList("test.003");
Map<String, Object> map = xmlResource.getResourceMap("test.002");
String value = xmlResource.getResourceString("test.001");
System.out.println("test.003:" + list.toString());
System.out.println("test.002:" + map.toString());
System.out.println("test.001:" + value);
```

出力結果

```
test.003:[10, 20]
test.002:{001=-, 002=男, 003=女}
test.001:文字列
```

解説

レシピ 3.7 に加え、xml形式のファイルをリソースとして使用したい場合もあります。このUraXMLResonanceも階層構造を指定可能です。またxml形式のファイル解析に必要なスキーマファイルは無く、幾つかのパターンで記載することが可能です。以下にリソースファイル内のルールを示します。

表 3-4 UraXMLResonanceのxml内容解析ルール

要素	属性	必須	内容
resources	-	●	リソースのルート属性の宣言です。
(任意の要素)	-	-	typecode属性がない限り、直上のオブジェクトのキー値として扱います。
(任意の要素)	typecode	-	任意要素の持つタイプを記号で指定します。 ・%：Map型 ・@：List型 ・指定なし：任意の要素を親要素のキーとした文字列型
(任意の要素)	key	-	任意の要素が、typecodeに指定されている場合、追加（もしくは値を変更）するキーを指定します。任意の要素がMap型の要素でない場合、この属性は無視されます。
(任意の要素)	value	-	任意の要素にtypecodeの指定がない場合、任意の要素を親要素のキー名とし、本属性の内容が値として設定されます。 指定がある場合は次の通り ・@の場合、任意の要素の持つList値に追加する。 ・%の場合、任意の要素の持つMapにkey属性で指定したキー値に追加もしくは上書きします。
key	-	-	直上の親の任意の要素に、typecodeが指定されている場合、追加（もしくは値を変更）するキーを指定します。親の任意の要素がMap型の要素でない場合、この要素は無視されます。
value	-	-	直上の任意の要素にtypecodeの指定がない場合、任意の要素を親要素のキー名とし、本要素の内容が値として設定されます。 指定がある場合は次の通り ・@の場合、親の任意の要素の持つList値に追加する。 ・%の場合、親の任意の要素の持つMapに直前のkey要素で指定したキー値に追加もしくは上書きします。

参照

レシピ 3.7 も本項目もレシピ 3.6 同様にリソース内で定義したその他のキー値を利用することも出来ます。

例 3-20 message_test_ja.xmlの追加

```
…
<test.error value="エラーが発生しました。" />
<test.004 value="$(test.error) 英字のみ入力可能です。[{0}]" />
<test.003 typecode="@" value="$(test.error) リストに追加できます。" />
…
```

例 3-21 UraXMLResourceの実施

```
UraXMLResource xmlResource = new UraXMLResource("message_test");
List<String> list = xmlResource.getResourceList("test.003");
```

```
String value = xmlResource.getResourceString("test.004", "a-Z");
System.out.println("test.003:" + list.toString());
System.out.println("test.004:" + value);
```

出力結果

```
test.003:[10, 20, エラーが発生しました。リストに追加できます。]
test.004:エラーが発生しました。 英字のみ入力可能です。[a-Z]
```

UraPropertyResourceクラスとは異なり、階層構造を持つことができますが、引数の変換は第 1 階層の値を返却する時のみに限定しています。（キー値がString型の場合のみに限定）

もしも取得するキーの値がリストやマップであって且つ、内容を可変にしたい場合(test.003 のようにリストで持ち、その中身を可変にしたいなど)は、使用する直前にJDK標準にあるMessageFormatを利用して自身で変換かけるなどの対応をする必要があります。

第4章
日付・時刻

日付・時刻操作をもっと自由に

レシピ4.1 和暦と西暦を併用利用可能なカレンダークラスの実装

要望

西暦の他にその国独自の暦も 1 つのカレンダークラスで使用したい

解決

UraCalendarUtilsからインスタンス化されたUraCalendarを利用します。

例 4-1 UraCalendarの利用

```
// 基本の使い方
UraCalendar cal = UraCalendarUtils.newUraCalendar();
cal.addYear(-300);
String localeEra = cal.getLocaleEraDisplayName(Locale.JAPAN);
String localeYear = cal.getLocaleYearDisplayName(Locale.JAPAN);
System.out.println("今年の 300 年前は西暦" + cal.getYear() + "年で、和暦は" + localeEra + loca
leYear + "年です。");
System.out.println();
// 国別に出力したい場合
cal = UraCalendarUtils.newUraCalendar(2001, 5, 15);
// テスト的に日本、米国、イタリア、タイのロケールを設定
List<Locale> localeList = newArrayList();
localeList.add(UraLocale.japanese());
localeList.add(UraLocale.US);
localeList.add(UraLocale.italian("IT"));
localeList.add(UraLocale.thai());
// ロケール別出力
System.out.println("newUraCalendar(2001, 5, 15)");
for(Locale loc: localeList) {
    String locName = loc.getDisplayName();
    String month = cal.getMonthDisplayName(UraCalendar.SHORT, loc);
    String dayOfWeek = cal.getDayOfWeekDisplayName(UraCalendar.SHORT, loc);
    String localeEra2 = cal.getLocaleEraDisplayName(loc);
    String localeYear2 = cal.getLocaleYearDisplayName(loc);
    System.out.println(" [" + cal.getYear() + "-"+ month + "-" + cal.getDayOfMonth() + "("
+ dayOfWeek + ")]["+localeEra2+localeYear2+"] 国(" + locName + ")の場合");
 }
```

出力結果

```
今年の 300 年前は西暦 1715 年で、和暦は正徳 5 年です。

newUraCalendar(2001, 5, 15)
```

```
[2015-6-15(月)][平成 27] 国(日本語)の場合
[2015-Jun-15(Mon)][AD2015] 国(英語 (アメリカ合衆国))の場合
[2015-giu-15(lun)][平成 27] 国(イタリア語 (イタリア))の場合
[2015-มิ.ย.-15(จ.)][พ.ศ.2558] 国(タイ語)の場合
```

解説

Javaの日付編集クラスはCalendarクラスですが（1.4 位まではDateクラスでも編集できましたが非推奨となってます）メモリ消費を防ぐためのメソッド数の省略のためか、かなり使いにくいものとなっております。例えば年を変更する場合、setメソッドで設定しますが、Calendarクラスで設定されているフィールド値を指定する必要があります。UraCalendarでは、setYearでsetと同様に年を設定でき、且つ戻り値が自インスタンスであることから、いわゆるチェーンメソッドとして利用することが可能です。（set(2015, 2, 16)はsetYear(2015).setMonth(2).setDay(16)と同様です。）併せて、フィールドが２種類追加されてローカル暦（localeEra）、ローカル年（localeYear）が使用できるようになっています。

日本は、和暦の文化があるため、Locale.JAPANなど日本のロケールを選んだ際には従来ロケール別に変更される、月、曜日、午前/午後、暦の他に、和暦と対象暦の年を表示出来ます。

これらは後述もしますがDateFormatSymbolsを拡張可能としたUraDateFormatSymbols利用しており、実データは、jsonファイルで管理されてますので、プロジェクト毎に自由に編集が可能となっています。いままで、和暦といえばJapaneseImperialCalendarクラスを使用することを検討し、編集出来ないことが理由で使用することを控えプロジェクトでオリジナルの和暦変換クラスを用意しなければならなくなることも多かったと思いますが、それが解消されます。

れい

参照

JDKのCalendarクラスは、タイ等で使われる仏暦（仏滅紀元暦）BuddhistCalendarも提供していますが、UraCalendarも和暦外の暦の1つとして仏暦を提供しています。

例 4-2 UraCalendar(仏暦)の使用

```
UraCalendar cal = UraCalendarUtils.newUraCalendar();
cal.setNow();
Locale buddhistLocale = UraLocale.thai("TH");
Calendar bCal = Calendar.getInstance(buddhistLocale);
bCal.setTime(cal.getTime());
System.out.println("西暦:[" + cal.getYear() + "], 仏暦:[" + bCal.getDisplayName(Calendar.ERA, Calendar.SHORT, buddhistLocale) + bCal.get(Calendar.YEAR) + "]はローカル暦では[" + cal.getLocaleEraDisplayName(buddhistLocale) + cal.getLocaleYearDisplayName(buddhistLocale) + "]です");
```

出力結果

```
西暦:[2015], 仏暦:[พ.ศ.2558]はローカル暦では[พ.ศ.2558]です
```

レシピ4.2 拡張されたカレンダークラスを使いこなす

要望

UraCalendarクラスのより有効な使用方法を知る

解決

add～やsetStrict～など使用します。

例 4-3 UraCalendarの便利なメソッド

```
UraCalendar cal = UraCalendarUtils.newUraCalendar(0L);
UraDateFormat udf = UraDateFormat.getInstance("yyyy-MM-dd aKK:mm:ss.SSS(EE)");
System.out.println("変更前:" + udf.format(cal));
// 編集
cal.addMilliseconds(555);
cal.setHoursOfDay(15).setDay(18);
System.out.println("変更中:" + udf.format(cal));
cal.rollMinutes(62).rollHoursOfDay(14);
// cal.rollMinutes(62).rollHours(14); // <-- これは、AM/PM変わらない
```

```
cal.setStrictMilliseconds(10137).setStrictDayOfWeek(0);
System.out.println("変更中:" + udf.format(cal));
cal.trancateDay().addMonth(-13);
System.out.println("変更後:" + udf.format(cal));
```

出力結果

```
変更前:1970-01-01 午前 09:00:00.000(木曜日)
変更中:1970-01-18 午後 03:00:00.555(日曜日)
変更中:1970-01-24 午前 05:02:00.137(土曜日)
変更後:1968-12-24 午前 00:00:00.000(火曜日)
```

解説

前述したとおり、UraCalendarには、フィールドごとのadd、set、get、rollメソッドがあり、get以外は全て自インスタンスを返却するチェーンメソッドになっています。その他にも、対象よりも下位のフィールド値をリセットするtruncate～メソッドや対象フィールドの限界値以上に設定できないsetStrict～（rollのset版のようなイメージ）、比較可能なcompareTimeやcompareDate、簡易的にformat出力可能なgetDisplay～やformat、fastFormatもメンバですのでそのまま利用することが可能です。必要に応じて利用ください。

レシピ4.3 フォーマットシンボルを拡張する

要望

存在するロケールを使用し、業務で拡張する必要がある

解決

必要なdate-format-symbols.jsonファイルを作成し、相当するLocaleクラスを指定します。

例 4-4 date-format-symbols_ja_JP_Hira.jsonの作成

```
{
  Eras: ["紀元前", "西暦"],
...
  DayAbbreviations: ["", "にち", "げつ", "か", "すい", "もく", "きん", "ど"],
...
  LocaleEra: [ ..., "へいせい" ],
  LocaleYear:[ ..., 600188400000 ]
}
```

例 4-5 date-format-symbols_ja_KR.jsonの作成

```
{
  "Eras": [ "紀元前", "西暦" ],
...
  DayAbbreviations: ["", "일", "월", "화", "수", "목", "금", "토"],
  LocaleEra: [ ..., "평성" ],
  LocaleYear:[ ..., 600188400000 ]
}
```

例 4-6 作成したロケールを使用する

```
Locale loc = UraLocale.japanese("JP", "Hira");
cal.setLocale(loc);
System.out.println("Name["+loc.getDisplayName()+"],Cal["+cal.format("yyyy.MMM.dd E (gN)
")+"]");

loc = UraLocale.japanese("KR");
cal.setLocale(loc);
System.out.println("Name["+loc.getDisplayName()+"],Cal["+cal.format("yyyy.MMM.dd E (gN)
")+"]");
```

出力結果

```
Name[Japanese (Japan,Hira)],Cal[2015.10.12 げつ (へいせい 27)]
Name[Japanese (South Korea)],Cal[2015.10.12 월 (평성 27)]
```

解説

UraCalendarはUraDateFormatSymbolsクラスを使用してgetDisplay～メソッドを表現しています。UraDateFormatSymbolsクラスはロケール別の外部ファイルdate-format-symbols.jsonを使用し、ロケールごとのFormatSymbolを返却します。

uranoplumsは、Localeは改修してはおりませんが、DateFormatSymbolsの改修は行っており、それを使用するCalendarクラス、DateFormatクラスにおいても編集しやすいよう改修したDateFormatSymbolsを使用したクラスを作成しています。上記は、言語は日本語ですが、地域を「ひらがな」にしたja_JP_Hiraと、国を「韓国」にしたja_KRを作成した例です。

DayAbbreviationsは、曜日の省略形を指定した場合に利用されます。（Calendar.SHORTの使用）未使用、日、月、火、水、木、金、土の順で登録します。以下に外部ファイルのキーの設定例を示します。

例 4-7 date-format-symbol.jsonファイルの設定

シンボルキー名	説明	getDisplay～	フィールド	表示方法
Eras	"西暦"を表示します 紀元前、西暦順に設定します	Era	ERA	ー
MonthNames	"月名"の正式名を表示します １月～１２月、未使用順に設定します	Month	MONTH	LONG
MonthAbbreviations	”月名"の略式名を表示します 設定順序は、正式名と同様です。	Month	MONTH	SHORT
AmPmMarkers	"午前午後"を表示します 午前、午後の順に設定します	AmPm	AM_PM	ー
DayNames	"曜日"正式名を表示します 未使用、日～土曜日の順に設定します	DayOfWeek	DAY_OF_WEEK	LONG
DayAbbreviations	"曜日"略式名を表示します 設定順序は、正式名と同様です。	DayOfWeek	DAY_OF_WEEK	SHORT
LocaleEra	"ローカル暦"を表示します LocaleYearごとの名称を決定します	LocaleEra	LOCALE_ERA	ー
LocaleYear	"ローカル暦年" 対象ローカルの暦の区切りをミリ秒もしくは年-月-日の文字列で設定します	LocaleYear	LOCALE_YEAR	ー

レシピ4.4 日付フォーマットを拡張する

要望

表示する日付フォーマットの種類を追加する必要がある

解決

UraDateFormatを継承します。

例 4-8 UraDateFormatの継承

```
public class TestDateFormat extends UraDateFormat {
    private static final long serialVersionUID = 2524757024617282830L;

    public TestDateFormat(String string) {
        super(string);
    }
    @Override
    protected void initialize() {
        super.initialize();
        this.optionalFormatMap.put("b", new UraDateFormat.PrinterField() {
            @Override
```

```
                public String getFormatField(String pattern, UraCalendar cal, Locale locale) {
                    return String.valueOf(cal.getMonth());
                }
            });
            this.optionalParserMap.put("b", new UraDateFormat.NumberParserField(UraCalen
dar.MONTH, "b") {
                @Override
                protected int modify(int iValue) {
                    return iValue - 1;
                }
            });
        }
        @Override
        protected void visitOptionalPattern(List<String> optionalPatternList) {
            // フォーマット許容パターンに、新しいパターン"b"を追加する
            optionalPatternList.add("b");
        }
    }
```

例 4-9 継承したクラスの使用

```
TestDateFormat ldf = new TestDateFormat("b");
Date target = ldf.parse("12");
System.out.println("[" + target + "]");

UraCalendar cal = UraCalendarUtils.newUraCalendar();
cal.setMonth(6);
String targetFmt = ldf.format(cal);
System.out.println("現在[" + targetFmt + "]です。");
```

出力結果

```
 [Tue Dec 01 00:00:00 JST 1970]
現在[6]です。
```

解説

UraDateFormatはJava7で追加されたフォーマットを加えた拡張性を意識したフォーマットクラスです。通常のDateFormatのformatメソッドによるフォーマット機能、parseメソッドによるパース機能を提供しています。

上記はDateFormatに"b"というパターンを追加した例です。（追加パターン"b"は月の表示パターンMと同等の設定をしています。）

追加方法はUraDateFormatのinitializeメソッドを継承し、formatならば" optionalFormatMap"に、parserならば、" optionalParserMap"マップに規則を追加し、パターンの対象となるように" optionalPatternList"にパターン"b"を追加します。

使用する側もformat、parseメソッドのどちらもオーバロードした拡張メソッドもあり、引数、戻り値にDateの他Calendar、UraCarendar、long型を利用できます。

追加執筆予定

第5章
文字列

文字操作。基本的な機能＋α

レシピ5.1 文字セットを自動判別する

要望

対象のデータの文字セットを判別する必要がある

解決

UraCharsetのgetCharsetを使用します。
執筆中

出力結果

執筆中・・・

解説

執筆中・・・

レシピ5.2 ひらがなとカタカナを相互変換する

要望

執筆中・・・

解決

執筆中・・・

出力結果

執筆中・・・

解説

執筆中・・・

レシピ5.3 ひらがなをローマ字変換する

要望

執筆中・・・

解決

執筆中・・・

出力結果

執筆中・・・

解説

執筆中・・・

レシピ5.4 ユーティリティクラスを使用する

要望

執筆中・・・

解決

執筆中・・・

出力結果

執筆中・・・

解説

執筆中・・・

第6章
その他

その他の機能について

レシピ6.1 リスト、マップのnewを簡略化する

要望

執筆中･･･

解決

執筆中･･･

出力結果

執筆中･･･

解説

執筆中･･･

レシピ6.2 文字に対して１方向ハッシュ暗号変換する

要望

執筆中･･･

解決

執筆中･･･

出力結果

執筆中･･･

解説

執筆中･･･

レシピ6.3 その他XXXXXX

要望

執筆中･･･

解決

執筆中･･･

出力結果

執筆中･･･

解説

執筆中･･･