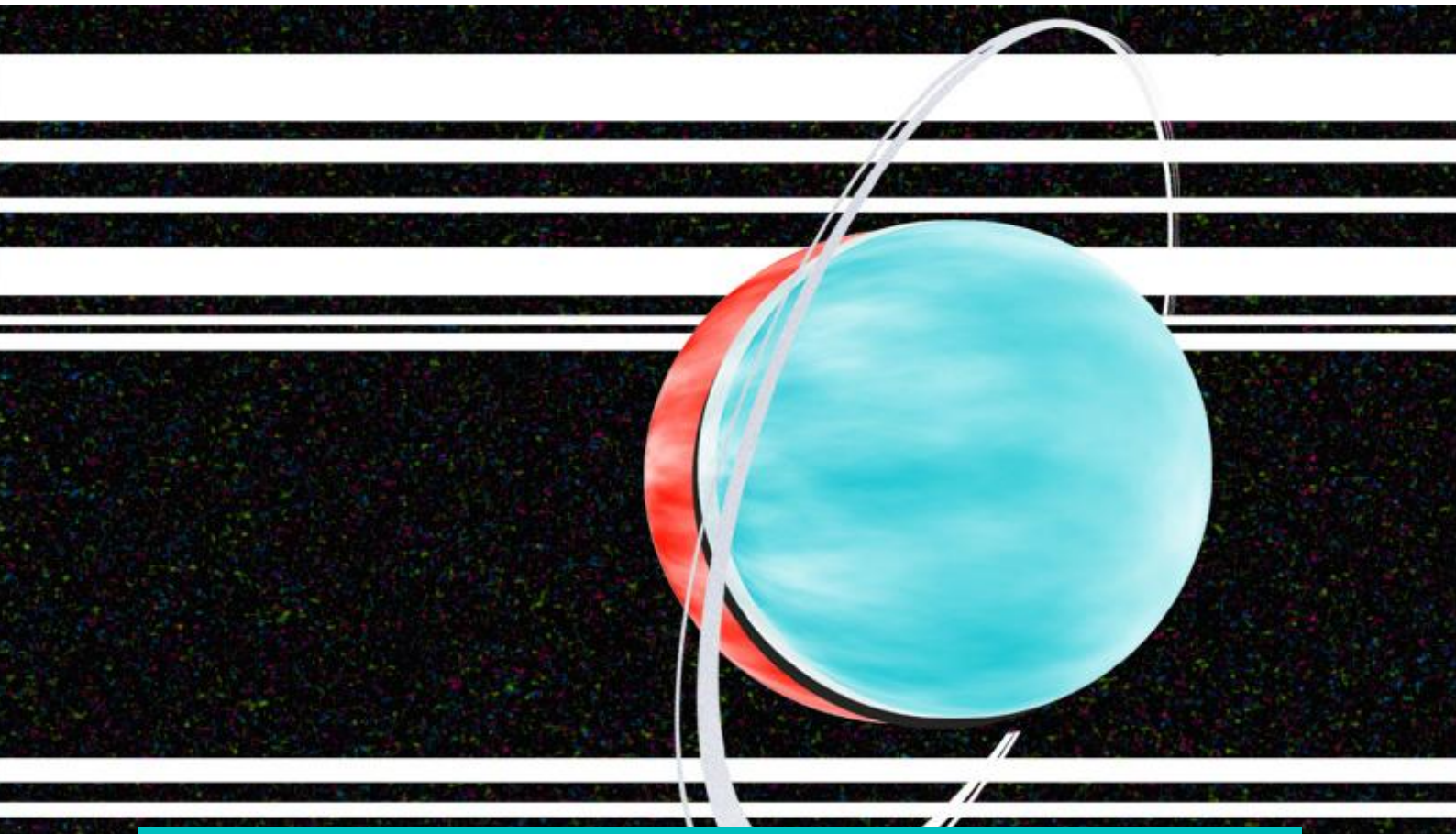
Recipes from the Uranoplums library



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、TerasoIna 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。

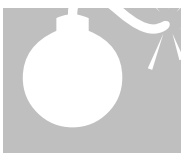
太字（ゴシック）：本文中で特に重要な箇所など、強調するときに使用します。

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 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1.2 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1.3 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

```
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 を利用します。

例 2-4 setIncludeFieldNamesPerttern の利用

```
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。

```
class CarDataObject extends UraDataObject {
    int id = 3400;
    String name = "ken";
    UraLog log = UraLoggerFactory.getUraLog();
}
…
```

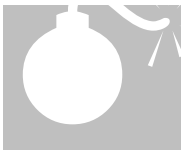
```
final CarDataObject car01 = new CarDataObject();
final CarDataObject car02 = new CarDataObject();
System.out.println("CompareTo[car01 car02]:" + car01.compareTo(car02));
```

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。

```
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

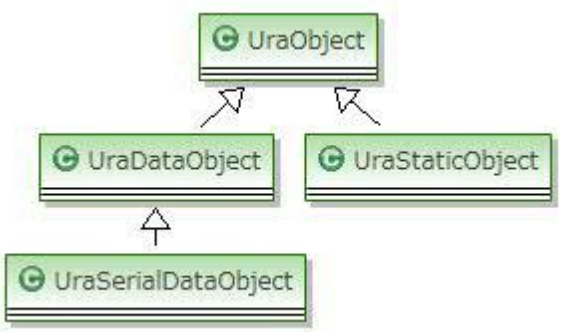
複数ある基盤オブジェクトの使い分けをしたい。

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。

```
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

```
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
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

```
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

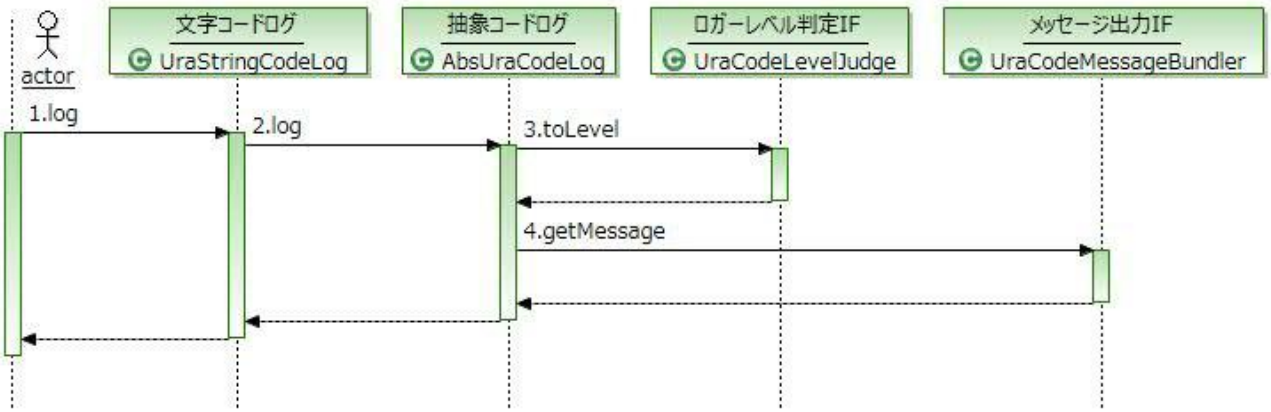
ログメッセージは、コードで指定し、内部でメッセージに展開し出力することが多いですが、展開するためのロジックは様々です。

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 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 3-8 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 3-9 リソース message_test.properties の作成

```
test.001=Wellcome to Uranoplums world!
test.003=Details...
```

例 3-10 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3.5 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 3-11 リソース message_test_ja.properties の追加

```
test.004=${test.mark} キーは `{0}' です ${test.mark}
test.mark=●●
test.log.dir=${test.baseDir}¥¥log
test.log.fileBase=${test.log.dir}¥¥apps_
```

例 3-12 リソース message_test_ja.properties の追加

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

例 3-13 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.log.fileBase") + "ja.log");
```

出力結果

```
●● キーは `鍵' です ●●
●● key is `キー' ●●
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。

その他業務では、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3.6 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 3-14 message_test_ja.json の使用

```
{
    test.001: "文字列",
    test.002: {
        001: "-",
        002: "男",
```

```
        003: "女"
    },
    test.003: [
        10,
        20
    ]
}
```

例 3-15 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
```

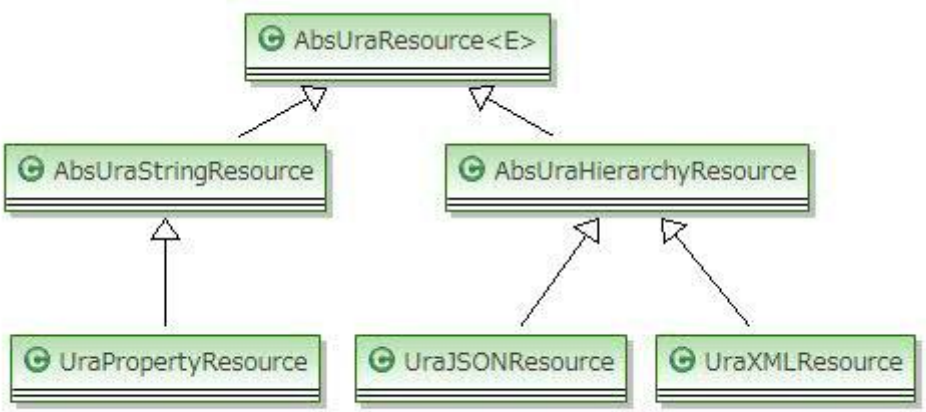
解説

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てきます。また、例にあるようにプロパティでは難しか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。

図 3-2 Uranoplums の持つリソース操作クラス階層



レシピ3.7 XML 形式のリソースも利用する

要望

執筆中・・・

解決

執筆中・・・

出力結果

執筆中・・・

解説

執筆中・・・

レシピ3.8 MDC を管理する

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

執筆中・・・

解決

執筆中・・・

出力結果

執筆中・・・

解説

執筆中・・・

レシピ4.2 フォーマットシンボルを拡張する

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4.3 日付フォーマットを拡張する

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4.4 日付・時刻のフォーマットで整合性チェックする

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

執筆中･･･

解決

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･･･