Asakusa Framework

Asakusa Frameworkとは？

Hadoop上で大規模な基幹バッチ処理を行うためのフレームワーク。
独自ドメイン特化言語
(Domain Specific Language, DSL)を使って処理を記述する。DSLはJavaを元に作成されている。

設計思想

「システムの本質は設計である。設計のないシステムでは品質は担保されない。品質のないシステムは適切な価値を利用者に届けることはできない。」

※参考 http://www.asakusafw.com/service/

メリット

1.品質を向上させる
2.システム全体の再利用性を上げる
3.トータルでの開発効率を上げる

※参考 http://www.asakusafw.com/service/

デメリット

1.アドホックな開発をベースにする短期イテレーショナルなシステム構築は対象外。
2.品質よりも開発スピードを優先をするシステム開発は対象としない。
3.設計能力のない人材による力技の開発には向いていない。

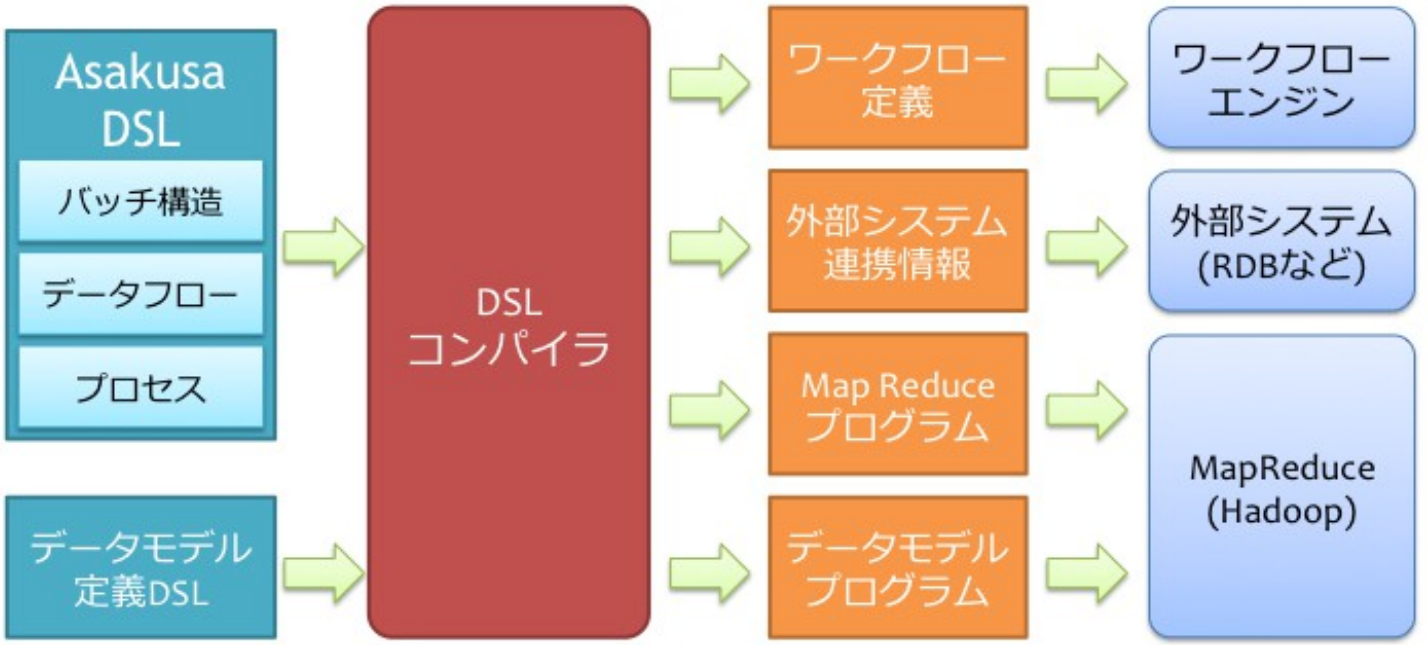
※参考 http://www.asakusafw.com/service/

つまり。。。

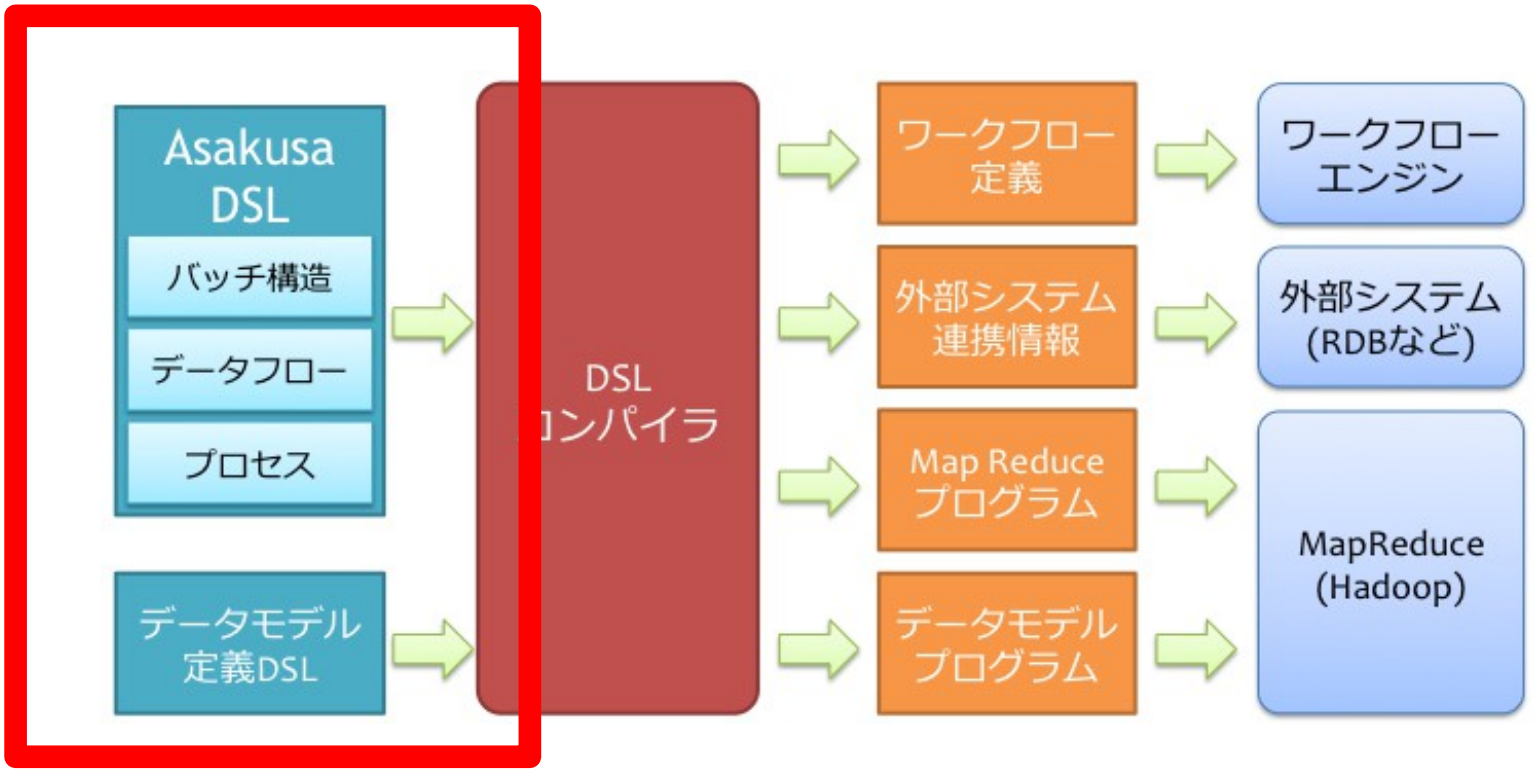
1.最近のアジャイル的な開発には向かない。
2.きちんと設計する時間を取って、きちんと設計できる人が使わないとメリットは得られない。
3.大規模開発等で大量にプログラマーを導入するような案件にも向かないと思われる。

アーキテクチャ

公式サイトから画像もってきたけどいいのだろうか。。。



開発者が気にするのは赤枠の部分



Asakusa Frameworkの
コンポーネント

1.データモデル定義DSL (DMDL)
2.Asakusa DSL
3.外部システム連携
4.バッチ実行ツール
5.自動テストサポート

1.データモデル定義DSL (DMDL)

Data Model Definition Language (DMDL)はAsakusa Frameworkで利用可能なデータモデルを定義するためのDSLです。 DMDLスクリプトというファイルにデータモデルの名前や構造を定義し、DMDLコンパイラを実行することで、定義したデータモデルに対応するJavaのプログラムを自動的に生成します。

※参考資料
http://asakusafw.s3.amazonaws.com/documents/latest/release/ja/html/dmdl/start-guide.html

ソース

```
model_a = {
    hoge : INT;
    fuga : TEXT;
    piyo : DOUBLE;
};
```

こんな感じでAsakusa Framework
上で扱うデータモデルを定義していく。

2.Asakusa DSL

Asakusa Frameworkでアプリケーションを作成するには、 Asakusa DSLで処理の流れや処理の本体を記述します。

※参考資料
http://asakusafw.s3.amazonaws.com/documents/latest/release/ja/html/dsl/index.html

種類

1.Operator DSL
2.Flow DSL
3.Batch DSL

以上の3種類。

Operator DSL

演算子と呼ばれるデータフロー処理の最小単位を記述する。

レコード → データフローに流れるデータ1つ分。Asakusa DSLでは「データモデルオブジェクト」として表現される。さっきのDMDL。

グループ → レコードを特定のキーでグループ化したもの。 Asakusa DSLでは、データモデルオブジェクトのリストや反復子などで表現される。

演算子の種類

Branch : レコードを内容に応じた出力に振り分ける
Update : レコードの内容を更新して出力する
Convert : レコードを別の種類のレコードに変換して出力する
MasterJoin : レコードにマスタデータを結合して出力する
MasterBranch : レコードとマスタデータの内容に応じた出力に振り分ける
MasterJoinUpdate : レコードの内容をマスタデータの情報を元に更新して出力する
Summarize : グループ化したレコードを集計して出力する
CoGroup : 複数種類のレコードをグループ化して任意の処理を行う

等。。。

ソース

```
/**
* レコードの値に100を設定する。
* @param hoge 更新するレコード
*/
@Update
public void edit(Hoge hoge) {
    hoge.setValue(100);
}
```

Flow DSL

演算子を組み合わせてデータフローの構造を記述するDSL。
以下の2種類

1.ジョブフロー
外部システムからデータを取り出して、外部システムにデータを書き出すデータフロー。

2.フロー部品
データフローそのものを演算子として定義する。
ほかのデータフローから演算子として利用できる。

ジョブフロー

```
package com.example.business.jobflow;
import com.asakusafw.vocabulary.flow.*;
@JobFlow(name = "piyo")
public class PiyoJob extends FlowDescription {
        In<Hoge> hoge;
        Out<Fuga> fuga;
        /**
        * コンストラクタ。
        */
        public StockJob(
        @Import(name = "hoge", description = HogeDb.class)
       In<Hoge> hoge,
       @Export(name = "fuga", description = FugaDb.class)
       Out<Fuga> fuga) {
                this.shipmentIn = shipmentIn;
                this.stockOut = stockOut;
        }
        @Override
        protected void describe() {
                CoreOperatorFactory core = new CoreOperatorFactory();
                OpFactory op = new OpFactory();
                 // チェックする
                CheckHoge check = op.checkHoge(hoge);
                ....
                // 結果を書き出す
                fugat.add(result.value);
        }
}
```

フロー部品

```
package com.example.business.flowpart;
import com.asakusafw.vocabulary.flow.*;
@FlowPart
public class PiyoPart extends FlowDescription {
    In<Hoget> hoge;
    Out<Fuga> fuga;
    /**
    * コンストラクタ。
    */
    public StockPart(
            In<Hoge> hoge,
            Out<Fuga> fuga) {
        this.hoge = hoge;
        this.fuga = fuga;
    }
}
```

Batch DSL

ジョブフローを組み合わせて、一連の処理を記述する。

ソース

```
package com.example.batch;
import com.asakusafw.vocabulary.batch.*;
@Batch(name = "piyo")
public class PiyoBatch extends BatchDescription {
    @Override
    protected void describe() {
        Work first = run(HogeFlow.class).soon();
        Work second = run(FugaFlow.class).after(first);
        ...
    }
}
```

3.外部システム連携

以下の三種類がある

1.WindGate
2.ThunderGate
3.Direct I/O

WindGate

・「ポータブルなThunderGate」の位置づけ。
・ローカルファイルシステム上のフラットファイル(CSV形式)に対するデータ入出力に対応。
・DBMS固有の機能に依存せず、標準SQL/JDBCインターフェースのみを使用した実装を提供。
・ThunderGateでは処理対象テーブルに対する管理カラムの追加が必要だが、WindGateでは不要。ただしThunderGateが提供していたロック機構などの仕組みが一部提供されない。

実装例
(データモデル)

```
@windgate.jdbc.table(name = "PIYO")
document = {
    "the name of this document"
    @windgate.jdbc.column(name = "HOGE")
    name : TEXT;

    "the content of this document"
    @windgate.jdbc.column(name = "FUGA")
    content : TEXT;
};
```

データモデルから、インポート、エクスポートを行うクラスの骨組みを自動生成してくれる。

インポート、エクスポートクラス

```
public class HogeDb extends JdbcImporterDescription {
	...
}

public class WordIntoDb extends JdbcExporterDescription {
	...
}
```

インポートクラスはJdbcImporterDescription
エクスポートクラスはJdbcExporterDescription
を継承する。

それぞれ各メソッドを実装。
String getProfileName()
Class<?> getModelType()
String getTableName()
List<String> getColumnNames()
Class<? extends DataModelJdbcSupport<?>> getJdbcSupport()

インポートクラスは
String getCondition()
も実装する。

ThunderGate

- バッチ処理中のデータに対する排他制御をサポート
- ジョブフロー内でのロングランニングトランザクションをサポート
- 変更差分のみをインポートするキャッシュ機能をサポート
- MySQL向けに最適化を実施
- テーブルメタデータからのデータモデル自動生成

ただし、以下の制約がある

- 現在はMySQLのみ対応
- データベースサーバー上での実行が必要
- ThunderGate用の管理テーブルや管理カラムが必要

インポート、エクスポートクラス

```
public class Hoge extends DbImporterDescription {
    ...
}

public class Fuga extends DbExporterDescription {
    ...
}
```

インポートクラスはDbImporterDescription
エクスポートクラスはDbExporterDescription
を継承する。

※ThunderGateはたくさん機能や設定があるので、
公式サイトを要参照
http://asakusafw.s3.amazonaws.com/documents/latest/release/ja/html/thundergate/index.html

Direct I/O

·Hadoopクラスターからバッチの入出力データを直接読み書きするための機構
·ThunderGateやWindGateと異なりデータ転送用の特別なツールは不要。
代わりにHadoopクラスターから直接参照できないリソースを利用できない。
·転送に時間のかかるデータを入出力する場合に適している。

4.バッチ実行ツール

YAESS

YAESS

・バッチを開発環境やテスト環境上で試験的に実行する
・ジョブ管理ツールからYAESSを経由してバッチを実行する
・他のツールにYAESSをライブラリとして組み込んで利用する

5.自動テストサポート

TestDriver

TestDriver

- Hadoopや外部入出力と自動的に連携したテストが可能
- 入出力と検査ルールを定義してバッチやデータフローを検証
- JUnitなどの様々なテストハーネスから利用可能

テスト

- 演算子のテスト
- データフローのテスト
- フロー部品のテスト
- ジョブフローのテスト
- バッチのテスト

例えば演算子のテスト

```
@Test
public void testCheckShipment_shipped() {
    StockOpImpl operator = new StockOpImpl();
    Shipment shipment = new Shipment();
    shipment.setShippedDate(new DateTime());
    shipment.setCost(100);

    ShipmentStatus actual = operator.checkShipment(shipment);

    assertThat("COSTが指定されていたらCOMPLETED",
            actual, is(ShipmentStatus.COMPLETED));
}
```

※参照元
http://asakusafw.s3.amazonaws.com/documents/latest/release/ja/html/testing/start-guide.html

実際に動かしてみる

Jinrikisha

Jinrikisha

Asakusa Framework の開発環境を手軽に構築するためのインストーラパッケージ。
Linux-32bit版
Linux-64bit版
MacOSX版 (Experimental)
の三つがある。

僕はMacOSX版 (Experimental)を使おうかと。。。

http://asakusafw.s3.amazonaws.com/documents/jinrikisha/ja/html/index.html

最後に

・書籍が無いので、資料は技術サイトを見ると良いです。
http://asakusafw.s3.amazonaws.com/documents/latest/release/ja/html/index.h

・サイトに導入事例のメニューがあるのですが、
まだ更新されていない模様。。。

他にも、開発者さんのブログを見ると何か情報があるかも？
http://d.hatena.ne.jp/okachimachiorz/