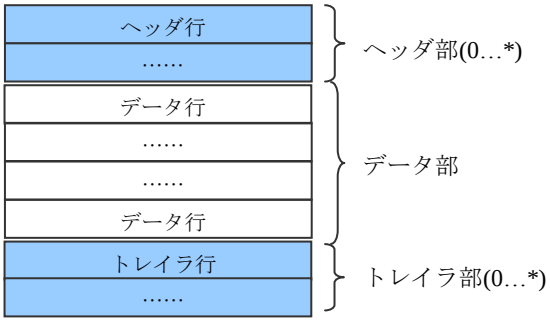
BL-07 ファイルアクセス機能

■ 概要

◆ 機能概要

- CSV 形式、固定長形式、可変長形式ファイルの入出力機能を提供する。
  - ファイル入力機能はシーケンシャルアクセス機能のみ提供し、ランダムアクセス機能は提供していない。

- ファイルアクセス機能では、入出力対象のファイルを下図のとおりヘッダ部/データ部/トレイラ部の 3 つに分けて扱う。
  - ヘッダ部とトレイラ部は文字列のリストとしてビジネスロジックから扱う。ヘッダ部とトレイラ部のカーディナリティは 0…*であり、ヘッダ行やトレイラ行のないファイルはヘッダ部やトレイラ部を 0 行として扱う。
  - データ部は 1 行あたり 1 つの DTO(今後、この DTO をファイル行オブジェクトと呼ぶ)としてビジネスロジックから扱う。

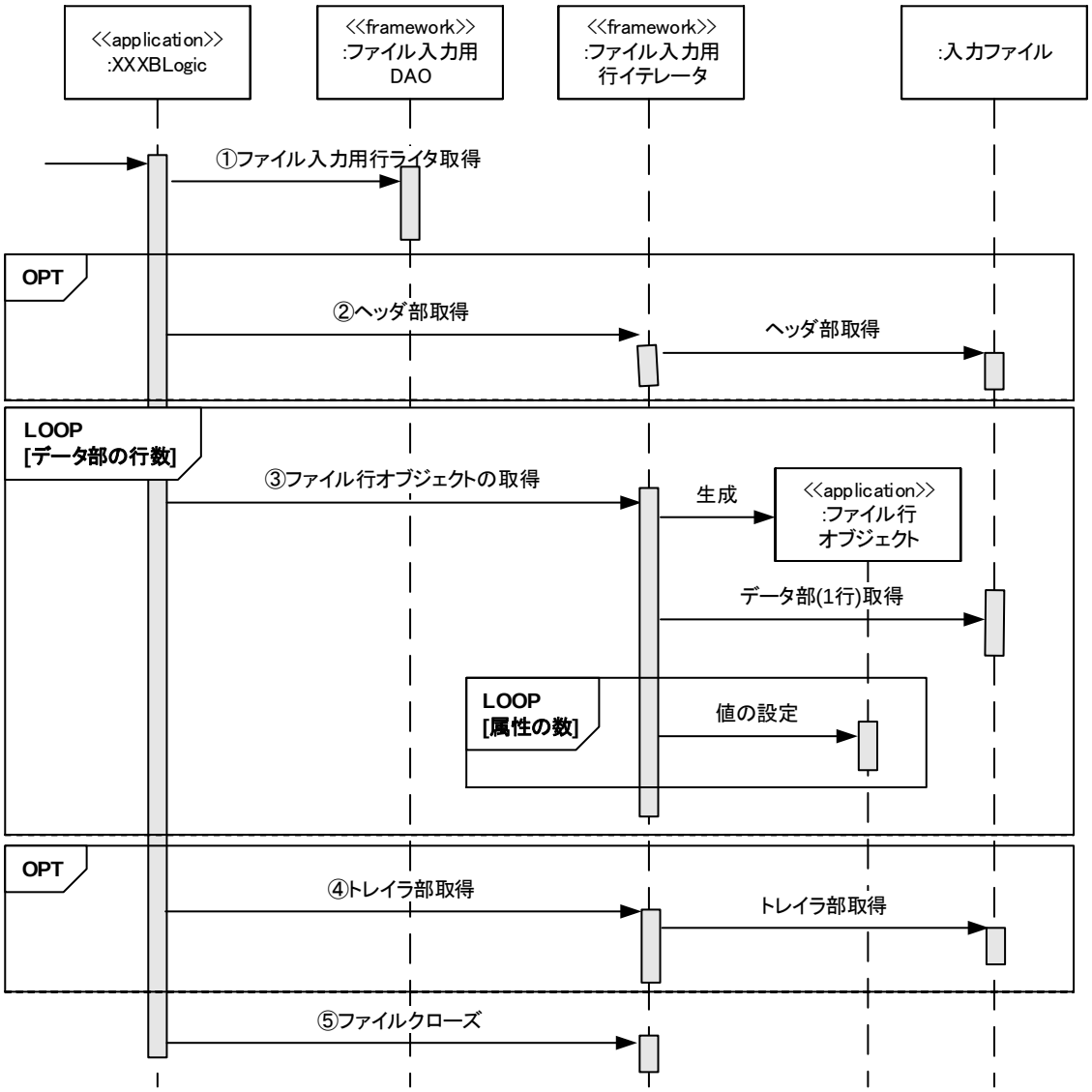


  - データ部の各項目では、ファイル行オブジェクトの定義により、項目に対するパディング(Padding)、トリム(Trim)、文字変換(StringConverter)等のフォーマット処理を行える。

- 本機能は TERASOLUNA Batch Framework for Java ver 2.x の『BC-01 ファイルアクセス機能』と同等である。

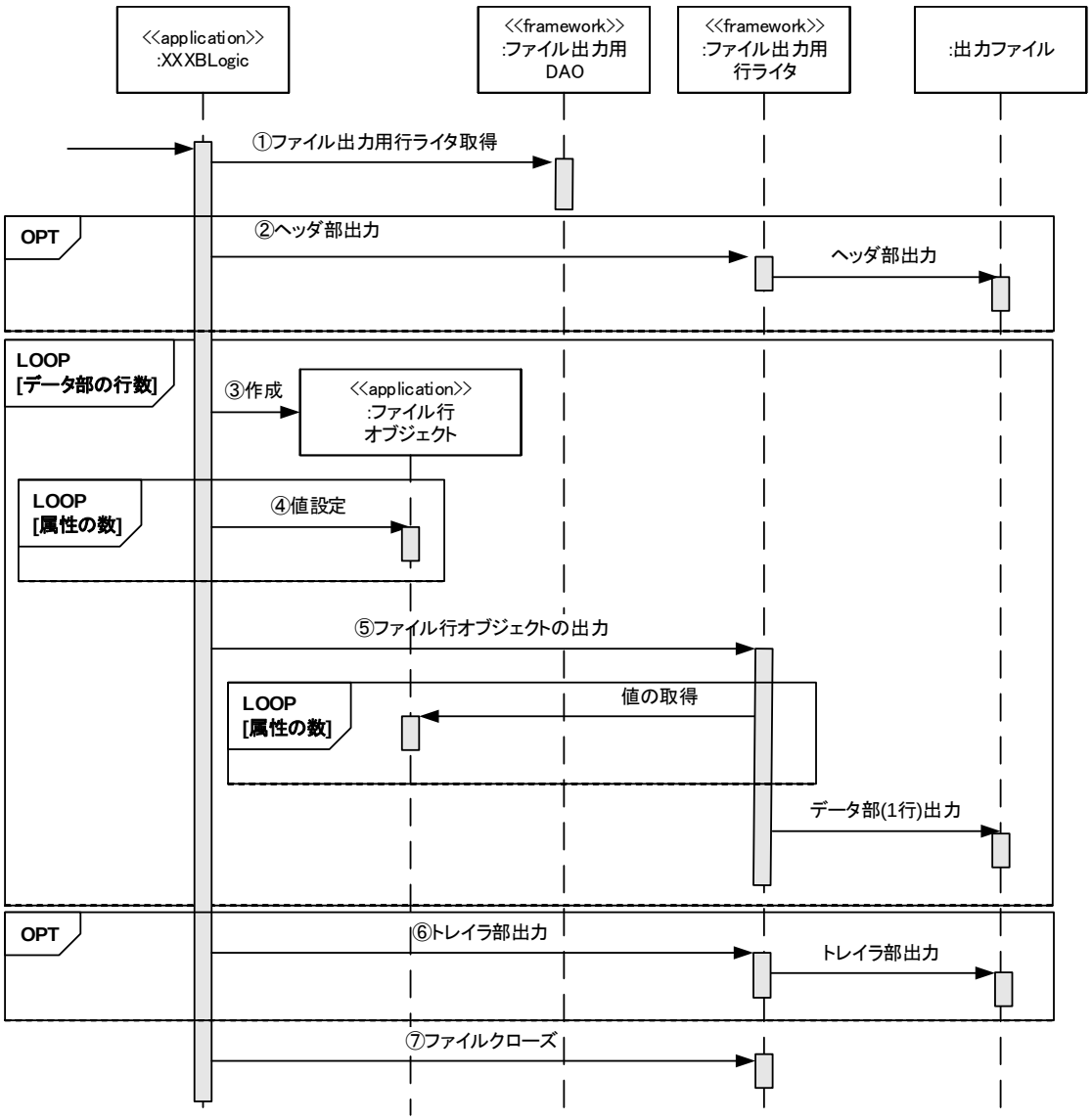
◆ 概念図

- ファイル入力処理



① ファイル入力用 DAO からファイル入力用行イテレータを取得する。
② ファイル入力用行イテレータを使用し、ヘッダ部を取得する。
③ ファイル入力用行イテレータを使用し、データ部を取得する。
ファイル入力用行イテレータはファイルからデータ部を 1 行読み取り、ファイル行オブジェクトに変換する。
④ ファイル入力用行イテレータを使用し、トレイラ部を取得する。
⑤ ファイルをクローズする。

- ファイル出力処理



① ファイル出力用 DAO からファイル出力用行ライタを取得する。
② ファイル出力用行ライタを使用し、ヘッダ部を出力する。
③ ファイル行オブジェクトを作成する。
④ ファイル行オブジェクトに値を設定する。
⑤ ファイル出力用行ライタを使用し、データ部を出力する。
ファイル出力用行ライタはビジネスロジックから受け取ったファイル行オブジェクトをファイル形式に沿った 1 行分のデータに変換する。
⑥ ファイル出力用行ライタを使用し、トレイラ部を出力する。
⑦ ファイルをクローズする。

◆ 解説

- ファイルアクセス機能で取り扱えるファイル形式

  CSV 形式、固定長形式、可変長形式、文字列データに対するファイル入出力機能を提供する。
  ファイル内の各行の項目数および項目の並び順は同一である必要がある。

  ✧ CSV 形式

  CSV 形式とは、データを「,(カンマ)」で区切ったものである。データを区切る際に使用している文字を特に"区切り文字"と呼ぶ。CSV 形式は可変長ファイルの区切り文字を「,(カンマ)」に固定したものになる。

  ✧ 固定長形式

  固定長形式とは、データを各項目で割り当てた長さ(バイト数)で区切ったものである。すべての行で項目の長さが同じである必要がある。

  ✧ 可変長形式

  可変長形式とは、データを任意の"区切り文字"を使って区切ったものである。

  ✧ 文字列データ

  文字列データとは、区切り文字やバイト数でデータを区切る必要がないものである。1 行分のデータを String 型として扱う。

- フレームワークが提供するファイル入力用 DAO
  フレームワークではファイル入力用 DAO インタフェース、およびファイル入力用行イテレータインタフェースを規定し、ファイル形式に対応したそれぞれのデフォルト実装を提供する。
  ファイル入力用 DAO の execute()メソッドを実行し、ファイル入力用行イテレータを取得する。ファイルの各行は、ファイル入力用行イテレータの next()メソッドで取得する。

  ✧ ファイル入力用インタフェース

項番	インタフェース名	概要
1	jp.terasoluna.fw.file.dao.FileQueryDAO	ファイル入力用 DAO インタフェース
2	jp.terasoluna.fw.file.dao.FileLineIterator	ファイル入力用行イテレータインタフェース

  ✧ ファイル入力用 DAO 実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileQueryDAO	CSV 形式のファイル入力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileQueryDAO	固定長形式のファイル入力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileQueryDAO	可変長形式のファイル入力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileQueryDAO	文字列データをファイルから入力する場合に利用する

  ・ ファイル入力用 DAO は、ファイル入力用行イテレータを生成する。

  ✧ ファイル入力用行イテレータ実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileLineIterator	CSV 形式のファイル入力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileLineIterator	固定長形式のファイル入力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileLineIterator	可変長形式のファイル入力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileLineIterator	文字列データをファイルから入力する場合に利用する

  ・ ファイル入力でのデータ部の入力は、データ部の 1 行分のデータを入出力オブジェクトに格納し、呼び出し元に返却する処理を提供する。
  ・ ヘッダ部、トレイラ部からの入力用メソッドを提供する。

- フレームワークが提供するファイル出力用 DAO
  フレームワークではファイル出力用 DAO、およびファイル出力用行ライタのインタフェースを規定し、ファイル形式に対応したそれぞれのデフォルト実装を提供する。
  ファイル出力用 DAO の execute()メソッドを実行し、ファイル出力用行ライタを取得する。ファイルの各行は、ファイル出力用行ライタの printDataLine()メソッドで出力する。

  ✧ ファイル出力用インタフェース

項番	インタフェース名	概要
1	jp.terasoluna.fw.file.dao.FileUpdateDAO	ファイル出力用 DAO インタフェース
2	jp.terasoluna.fw.file.dao.FileLineWriter	ファイル出力用行ライタインタフェース

  ✧ ファイル出力用 DAO 実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileUpdateDAO	CSV 形式のファイル出力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileUpdateDAO	固定長形式のファイル出力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileUpdateDAO	可変長形式のファイル出力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileUpdateDAO	文字列データをファイルへ出力する場合に利用する

  ・ ファイル出力用 DAO は、ファイル出力用イテレータを生成する。

  ✧ ファイル出力用行ライタ実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileLineWriter	CSV 形式のファイル出力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileLineWriter	固定長形式のファイル出力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileLineWriter	可変長形式のファイル出力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileLineWriter	文字列データをファイルへ出力する場合に利用する

  ・ ファイル出力でのデータ部の出力は、ファイル行オブジェクトに格納された 1 行分のデータをファイルに書込む処理を提供する。
  ・ ヘッダ部、トレイラ部への出力メソッドを提供する
  ・ ファイル出力先は存在するフォルダを指定する必要がある。フォルダが存在しない場合、例外が発生する。

- ファイルアクセス機能から発生する例外
  ファイルアクセス時に例外が発生した場合、ファイル入出力用 DAO やファイル入力用行イテレータ、ファイル出力用行ライタからエラーが発生したファイルの情報を格納された例外がスローされる。スローされる例外には以下の 2 つがある。

項番	例外クラス名	概要
1	jp.terasoluna.fw.file.dao. FileException	ファイル全体に関わるエラーに対応する例外クラス。 以下の情報を保持する。 ・ファイル名
2	jp.terasoluna.fw.file.dao. FileLineException (FileException のサブクラス)	ファイルの行に関わるエラーに対応する例外クラス。 以下の情報を保持する。 ・ファイル名 ・エラーが発生した箇所の行番号 ・エラーが発生したカラムのカラムインデックス (0 から開始) ・エラーが発生したカラムのカラム名(ファイル行オブジェクトのプロパティ名)

例外クラスが保持する情報を用いて、主にログ出力を行うことができる。例外処理はビジネスロジック上で行うほか、『BL-04 例外ハンドリング機能』を用いて処理することもできる。
例外ハンドリング機能の使用方法についての詳細は機能説明書の『BL-04 例外ハンドリング機能』を参照すること。

■ 使用方法

◆ コーディングポイント

- ファイル行オブジェクトの実装
  - ファイル全体に関わる定義
  - ファイル項目に関わる定義
  - ファイル行オブジェクトの実装例
- ビジネスロジックの実装
  - ファイル入力処理
  - ファイル出力処理

- ファイル行オブジェクトの実装
  入出力対象のファイルをヘッダ部/データ部/トレイラ部に分け、データ部の 1 行を表すファイル行オブジェクトを作成する。
  ファイル行オブジェクトには、データ部 1 行分の各項目に対応する属性と、属性へのアクセサメソッドを実装する。
  ファイル全体に関わる定義(改行文字等)やファイルの個々の項目の定義(項目のバイト長等)はファイル行オブジェクトに Java アノテーションを付与して設定する。

  ➢ ファイル全体に関わる定義
  ファイル行オブジェクトのクラスに対して FileFormat アノテーションを付与して設定する。FileFormat アノテーションは入力ファイル、および出力ファイルのどちらの場合にも同じアノテーションを設定する。

  ✧ アノテーション FileFormat 設定項目

項番	論理項目名 物理項目名	説明	デフォルト値	CSV		固定長		可変長		文字列データ	
				入	出	入	出	入	出	入	出
1	行区切り文字 lineFeedChar	行区切り文字(改行文字)を設定する。	システムデフォルト/なし(固定長)	○	○	○	○	○	○	○	○
2	区切り文字 delimiter	「,(カンマ)」等の区切り文字を設定する。	「,(カンマ)」	×	×	×	×	○	○		
3	囲み文字 encloseChar	「"(ダブルクォーテーション)」等のカラムの囲み文字を設定する。	なし	○	○	×	×	○	○		
4	ファイルエンコーディング fileEncoding	入出力を行うファイルのエンコーディングを設定する。	システムデフォルト	○	○	○	○	○	○	○	○
5	ヘッダ行数 headerLineCount	入力ファイルのヘッダ部に相当する行数を設定する。	0	○		○		○		○	
6	トレイラ行数 trailerLineCount	入力ファイルのトレイラ部に相当する行数を設定する。	0	○		○		○		○	
7	ファイル上書きフラグ overWriteFlg	出力ファイルと同じ名前のファイルが存在する場合に上書きするかどうかを設定する。 [true/false]（上書きする/上書きしない）	FALSE		○		○		○		○

※○の項目は必要に応じて設定可。×の項目は設定できないことを表している(デフォルト以外の値に変更した場合は、実行時エラーとなる)。無印は設定を無視することを表している。

- すべての形式に共通の補足
  - 「行区切り文字」、「区切り文字」の「システムデフォルト」とは、以下で取得できる実行環境に依存した値である。
    「行区切り文字」: System.getProperty("line.separator");
    「ファイルエンコーディング」: System.getProperty("file.encoding");
  - 「行区切り文字」、「区切り文字」にタブや改行文字を使用する場合、Java 言語仕様で定められているエスケープシーケンス(¥t、¥r 等)で記述すること。
  - 「行区切り文字」と「区切り文字」は同一の値を設定することができない。
  - 「区切り文字」と「囲み文字」は同一の値を設定することができない。
  - 複数のビジネスロジック、ファイル出力用行ライタから 1 つのファイルに同時に出力する場合、「ファイル上書きフラグ」を TRUE に設定するとデータが破損する可能性があり、ファイル上書きファイルフラグを FALSE にするとデータの出力順番がランダムになる可能性がある。

- 固定長形式の場合の補足
  - 「行区切り文字」のデフォルト値は「なし」（改行なし）である。
  - 「行区切り文字」を設定しなかった場合、ヘッダ行数とトレイラ行数を設定することはできない。

- 可変長形式の場合の補足
  - 「区切り文字」に「¥u0000」を設定することはできない。

- 文字列データ形式の場合の補足
  - ファイル入力用行イテレータやファイル出力用行ライタの取得用に、@FileFormat のみを記述したファイル行オブジェクトを作成する必要がある。

➢ ファイル項目に関わる定義

ファイル行オブジェクトの属性に対して InputFileColumn アノテーション(入力用の設定)、または、OutputFileColumn アノテーション(出力用の設定)を付与して設定する。1 つのファイル行オブジェクトを入力用、出力用の両方に使用する場合には、1 つの属性に対して InputFileColumn アノテーションと OutputFileColumn アノテーションの両方を設定する。

✧ InputFileColumn, OutputFileColumn の設定項目

項番	論理項目名 物理項目名	説明	デフォルト値	CSV		固定長		可変長	
				入	出	入	出	入	出
1	カラムインデックス columnIndex	データ部の 1 行のカラムの内、何番目のデータをファイル行オブジェクトの属性に格納するのかを設定する。インデックスは「0(ゼロ)」から始まる整数。	なし	◎	◎	◎	◎	◎	◎
2	フォーマット columnFormat	BigDecimal 型、Date 型に対するフォーマットを設定する。int 型、String 型に対しては設定を行っても有効にならない。	(補足参照)	○	○	○	○	○	○
3	バイト長 bytes	各カラムに対するバイト長を設定する。	-1	○	○	◎	◎	○	○
4	パディング種別 paddingType	パディングの種別を設定する。列挙型 PaddingType から値を選択する。[RIGHT(左寄せ)/LEFT(右寄せ)/NONE(パディングなし)]	NONE	○	○	○	○	○	○
5	パディング文字 paddingChar	パディングする文字を設定する(半角文字のみ設定可能)。	半角スペース	○	○	○	○	○	○
6	トリム種別 trimType	トリムの種別を設定する。列挙型 TrimType から値を選択する。[RIGHT(右トリム)/LEFT(左トリム)/BOTH(両側トリム)/NONE(トリムなし)]	NONE	○	○	○	○	○	○
7	トリム文字 trimChar	トリムする文字を設定する(半角文字のみ設定可能)。	半角スペース	○	○	○	○	○	○
8	文字変換種別 stringConverter	String 型のカラムについて、大文字変換等を設定する。StringConverter インタフェースの実装クラスを指定する。[StringConverterToUpperCase.class (大文字に変換) / StringConverterToLowerCase.class(小文字に変換) / NullStringConverter.class(変換しない)]	NullStringConverter.class	○	○	○	○	○	○
9	囲み文字 columnEncloseChar	カラム単位で「"(ダブルクォーテーション)」等のカラムの囲み文字を設定する。	なし	○	○	×	×	○	○

※◎の項目はアノテーションを設定する際の必須項目(必須項目を設定しなかった場合、コンパイルエラーまたは実行時エラーとなる)。○の項目は必要に応じて設定可。×の項目は設定できないことを表している(デフォルト以外の値に変更した場合は、実行時エラーとなる)。無印は設定を行っても有効にならないことを表している。

- すべての形式に共通の補足
  - ファイル行オブジェクトで使用できる属性の型は、java.lang.String、int、java.math.BigDecimal、java.util.Date の 4 種類とする。
    フレームワークから値を操作できるように、各属性には可視性が public の setter/getter を用意すること。
  - ファイルのカラム数とアノテーションを付与したファイル行オブジェクトの属性の数が等しくなるように設定すること。異なる場合、例外が発生する。
  - ファイルからの入力文字列を BigDecimal に変換したり、BigDecimal のデータをファイルに出力したりする際には、「フォーマット」で指定したフォーマットに従う。デフォルトのフォーマットは「yyyyMMdd」である。その他の有効なフォーマット形式の詳細については、"java.text.SimpleDateFormat"の Java SE API 仕様を参照のこと。フォーマットに変換できない場合、例外が発生する。
  - ファイルからの入力文字列を Date に変換したり、Date のデータをファイルに出力したりする際は、「フォーマット」で指定したフォーマットに従う。デフォルトのフォーマットは、BigDecimal#toPlainString の結果である。その他の有効なフォーマット形式の詳細については、"java.text.DecimalFormat"の Java SE API 仕様を参照のこと。フォーマットに変換できない場合、例外が発生する。
  - 「バイト長」に 1 以上の値を設定した場合、入出力時にバイト長チェックが行われる。
    - 入力時の「バイト長」には各種変換処理前のファイルから取得する時点の長さを設定する。
    - 出力時の「バイト長」には各種変換処理後のファイルへ出力する時点の長さを設定する。
  - 各種変換処理の順番は、入力時と出力時で異なることに留意すること。
    - 入力時はバイト数チェック、トリム処理、パディング処理、文字列変換処理の順番である。
    - 出力時はトリム処理、パディング処理、文字列変換処理、バイト数チェックの順番である。
  - パディング種別で NONE 以外を指定した場合は、設定した「バイト長」になるまでパディングを行うため、「バイト長」を必ず設定すること。
  - ファイル項目の「囲み文字」の設定は、ファイル全体の「囲み文字」の設定よりも優先される。
  - ファイル全体/ファイル項目に囲み文字を設定しており、出力データに囲み文字が含まれていた場合、フレームワークは同じ囲み文字を追加してエスケープ編集を行ってからファイルに文字列として出力する。

- 固定長形式の場合の補足
  - 「バイト長」は必ず設定すること。
  - 固定長形式の場合、バイト長チェックの単位がカラムではなく行となる。アノテーションで設定した「バイト長」の合計と、ファイルから読み取った 1 行のバイト数が等しくなるようにすること。異なる場合、例外が発生する。

- 文字列データ形式の場合の補足
  - 文字列データ型の場合、ファイル項目の定義は不要である。

- ファイル行オブジェクトの実装例

  - CSV 形式のデータをファイル行オブジェクトに格納する場合の記述例

```
@FileFormat(encloseChar = '"')
public class SampleFileLineObject {
      ……
    @InputFileColumn(
           columnIndex = 0,
           columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @InputFileColumn(
           columnIndex = 1,
           stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
           columnIndex = 2,
           columnFormat="###,###,###")
    private BigDecimal uriage = null;
      …(getter/setter は省略)…
}
```

アノテーションの FileFormat は必須

アノテーション InputFileColumn とパラメータの設定

  - 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下のとおりとなる。

```
"2006/07/01","shop01","1,000,000" ← CSV形式のデータ
```

  - ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

  - 以下のように、囲み文字で囲まれていないデータや、カラムが囲み文字で囲まれていないが、データ内にエスケープされていない囲み文字が含まれているような場合でも、フレームワークはデータを DTO に変換できる。

```
2006/07/01,shop01,"1,000,000"
"2006/07/01",A"shop01"B,"1,000,000"
```

囲み文字で囲まれていないデータ部があっても正しく shop01 が取得できる。

囲み文字で囲まれていないデータ部に対して囲み文字があった場合は、囲み文字がエスケープされているものとして取得する。(shopId に A"shop01"B という文字列が格納されるため注意)。

➢ ファイル全体に関わる定義情報を設定する場合の記述例

```
@FileFormat(encloseChar = '"', lineFeedChar="¥r¥n",
  headerLineCount = 1, trailerLineCount= 1)
public class SampleFileLineObject {
     ……
    @InputFileColumn(
           columnIndex = 0,
           columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @InputFileColumn(
           columnIndex = 1,
           stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
           columnIndex = 2,
           columnFormat="###,###,###")
    private BigDecimal uriage = null;
     …(getter/setter は省略)…
}
```

FileFormat で、ヘッダ部行数とトレイラ部行数を指定する。

✧ 上記のファイル行オブジェクトに下記のデータを格納すると、各属性の値は以下のとおりとなる。

```
支店名：千葉支店                                ← ヘッダ部
"2006/07/01","shop01","1,000,000"               ← データ部
合計金額：1,000,000                             ← トレイラ部
```

ヘッダ部とトレイラ部を含んだファイル

✧ ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

※ヘッダ部とトレイラ部はファイル行オブジェクトに格納されない。

➢ ファイル項目でトリム種別を設定し、デフォルトのトリム文字を使用した定義情報を設定する場合の記述例

```
@FileFormat(encloseChar = '"')
public class SampleFileLineObject {
     ……
    @InputFileColumn(
           columnIndex = 0,
           columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @InputFileColumn(
           columnIndex = 1,
           trimType = TrimType.RIGHT,
           stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
           columnIndex = 2,
           columnFormat="###,###,###")
    private BigDecimal uriage = null;
     …(getter/setter は省略)…
}
```

右側の空白をトリム(削除)するように設定する。

✧ 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下のとおりとなる。

```
"2006/07/01","shop01  ","1,000,000"        ← データ部
```

✧ ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01     ←右側にあった空白文字を削除している。
uriage = 1000000
```

- ファイル項目でトリム種別を設定し、個別のトリム文字を使用した定義情報を設定する場合の記述例

```
@FileFormat(encloseChar = '"')
public class SampleFileLineObject {
    ……
    @InputFileColumn(
            columnIndex = 0,
            columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @InputFileColumn(
            columnIndex = 1,
            trimType = TrimType.LEFT,
            trimChar = '0',
            stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
            columnIndex = 2,
            columnFormat="###,###,###")
    private BigDecimal uriage = null;
    …(getter/setter は省略)…
}
```

左側の'0'の文字ををトリム(削除)するように設定する。

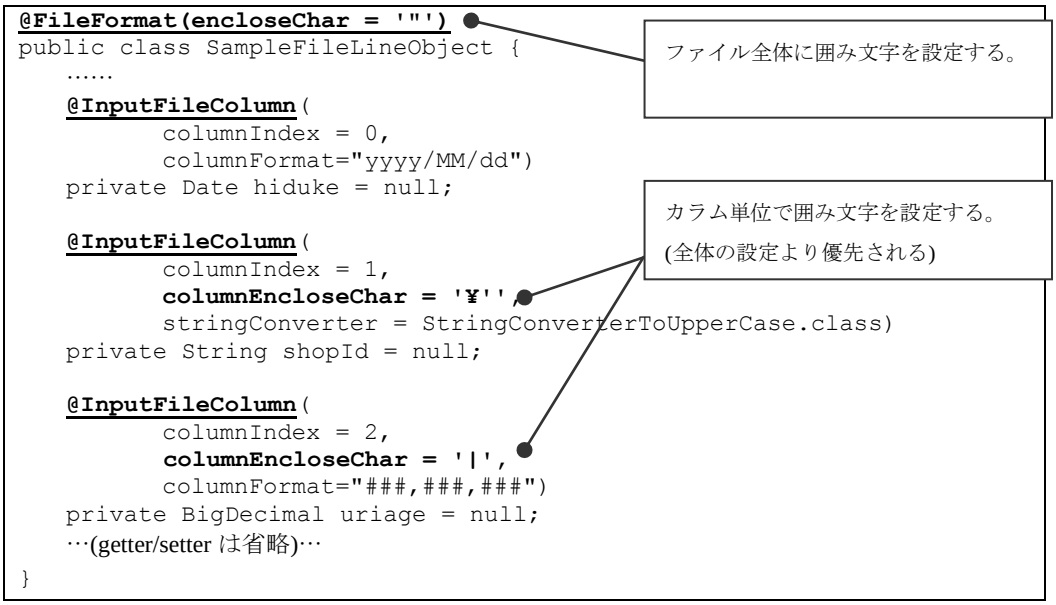
  - 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下のとおりとなる。

```
"2006/07/01","000shop01","1,000,000"        ← データ部
```

  - ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01 ←対象文字列の左側にある'0'が削除される。
uriage = 1000000
```

- ファイル全体に囲み文字を設定し、更にファイル項目で個別の囲み文字を使用した定義情報を設定する場合の記述例



- 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下のとおりとなる。

```
"2006/07/01",'shop01',|1,000,000|    ← データ部
```

- ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

- 固定長形式で行区切りなしのデータをファイル行オブジェクトに格納する場合の記述例

```
@FileFormat(lineFeedChar = "")
public class SampleFileLineObject {
    ……
    @InputFileColumn(
            columnIndex = 0,
            bytes = 10,
            columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @InputFileColumn(
            columnIndex = 1,
            bytes = 6,
            stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
            columnIndex = 2,
            bytes = 9,
            columnFormat="###,###,###")
    private BigDecimal uriage = null;
    …(getter/setter は省略)…
}
```

行区切り文字を無しに設定する。

各カラムにバイト数を設定する。

- 上記のファイル行オブジェクトに下記の固定長形式のデータを格納すると、各属性の値は以下のとおりとなる。

```
2006/07/01shop011,000,000  ← データ部
```

- ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

- ファイルの設定として、囲み文字と区切り文字を設定し、データの一部をデフォルトのパディング文字でパディング処理したデータをファイルに出力する場合の記述例

```
@FileFormat(delimiter = ',', encloseChar = '"' )
public class SampleFileLineObject {
    ……
    @OutputFileColumn(
            columnIndex = 0,
            columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @OutputFileColumn(
            columnIndex = 1,
            paddingType = PaddingType.LEFT,
            bytes = 10,
            stringConverter = StringConverterToLowerCase.class)
    private String shopId = null;

    @OutputFileColumn(
            columnIndex = 2,
            columnFormat="###,###,###")
    private BigDecimal uriage = null;
    …(getter/setter は省略)…
}
```

アノテーションの FileFormat は必須
区切り文字、囲み文字を設定

アノテーション OutFileColumn とパラメータの設定

パディング処理を行う場合は、バイト長の指定が必須となる

✧ 出力対象となるファイル行オブジェクトの値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

✧ 上記のファイル行オブジェクトを出力すると以下の値となる。

```
"2006/07/01","    shop01","1,000,000"
```

- データの一部を個別のパディング文字でパディング処理したデータをファイルに出力する場合の記述例

```
@FileFormat(delimiter = ',', encloseChar = '"' )
public class SampleFileLineObject {
    ……
    @OutputFileColumn(
            columnIndex = 0,
            columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @OutputFileColumn(
            columnIndex = 1,
            paddingType = PaddingType.RIGHT,
            paddingChar = '0',
            bytes = 10,
            stringConverter = StringConverterToLowerCase.class)
    private String shopId = null;

    @OutputFileColumn(
            columnIndex = 2,
            columnFormat="###,###,###")
    private BigDecimal uriage = null;
    …(getter/setter は省略)…
}
```

右側のパディング処理を行い、パディング文字として'0'を設定する。

- 出力対象となるファイル行オブジェクトの値

```
 hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

- 上記のファイル行オブジェクトを出力すると以下の値となる。

```
"2006/07/01","shop010000","1,000,000"
```

- ビジネスロジックの実装
  - ファイル入力処理

    - ビジネスロジックの実装

      ファイル入力処理を行うクラスでは、FileQueryDAO の execute()メソッドでファイル入力用行イテレータを取得する。ファイル入力用行イテレータ取得時に、ファイルオープンが行われる。ファイル入力用行イテレータの next メソッドで、ファイル行オブジェクトを取得する。

      - ビジネスロジックの実装例

```
@Component
public class B001001BLogic implements BLogic {

    @Inject
    @Named("csvFileQueryDAO")
    FileQueryDAO fileQueryDAO = null;

    public int execute(BLogicParam arg0) {

        ...
        // ファイル入力用行イテレータの取得
        FileLineIterator<SampleFileLineObject> fileLineIterator
                                = fileQueryDAO.execute(basePath +
                                        "/some_file_path/uriage.csv",FileColumnSample.class);

        try {

          // ヘッダ部の読み込み
          List<String> headerData = fileLineIterator.getHeader();
           ... // 読み込んだヘッダ部に対する処理

          while(fileLineIterator.hasNext()){
                // データ部の読み込み
                SampleFileLineObject sampleFileLine
                                        = fileLineIterator.next();
                 ... // 読み込んだ行に対する処理
          }
          // トレイラ部の読み込み
          List<String> trailerData = fileLineIterator.getTrailer();
          ... // 読み込んだトレイラ部に対する処理
        } finally {
          // ファイルのクローズ
          fileLineIterator.closeFile();
        }
... (以下略)
```

Batch 3.x では@Inject アノテーションを使用して、ビジネスロジックへの DAO の DI を行う。(FileDAO の場合は FileQueryDAO インタフェースの実装クラスが多数存在するので、@Named を使用して Bean 名を指定する必要がある。)

ファイルパスとファイル行オブジェクトクラスを引数にして、ファイル入力用行イテレータを取得する。

アノテーション FileFormt の headerLineCount で設定した行数分のヘッダ部を取得する。

ファイル形式に関わらず、next()メソッドを使用する

アノテーション FileFormt の trailerLineCount で設定した行数分のトレイラ部を取得する

closeFile()メソッドでファイルを閉じること

➢ ファイルの入力順序

トレイラ部の入力は、データ部の入力がすべて終わった後に行う必要がある点に留意すること。

➢ 入力チェック

読み込んだファイルの入力チェックを行うには、『AL-043 入力チェック機能』を使用する。使用方法についての詳細は機能説明書の『AL-043 入力チェック機能』を参照すること。

➢ スキップ処理

ファイル入力機能では入力を開始する行を指定することができる。スキップ処理は中断していたファイルの読み込みを再開するような場合に使用することができる。

✧ ビジネスロジック実装例

```
……
// スキップ処理
fileLineIterator.skip(1000);
……
```

fileLineIterator のカレント行から 1000 行分のデータ行を読み飛ばす処理を行う

- ファイル出力処理

  - ビジネスロジックの実装

    ファイル出力処理を行うクラスでは、FileUpdateDAO の execute メソッドでファイル出力用行ライタを取得する。ファイル出力用行ライタの取得時に、ファイルがオープンされる。

✧ ビジネスロジック実装例

```
@Component
public class B001001BLogic implements BLogic {

    @Inject
    @Named("csvFileUpdateDAO")
    FileUpdateDAO csvFileUpdateDAO = null;

    public int execute(BLogicParam arg0) {

        // ファイル出力用行ライタの取得
        FileLineWriter<SampleFileLineObject> fileLineWriter
                = fileUpdateDAO.execute(basePath + "/some_file_path/uriage.csv",
                                        SampleFileLineObject.class);

        try {
            // ヘッダ部の出力
            fileLineWriter.printHeaderLine(headerStringList);
            ...
        while ( ... ) {
            ...
            // データ部の出力(1行)
            fileLineWriter.printDataLine(sampleFileLineObject);
            ...
        }
        ...
        // トレイラ部の出力
        fileLineWriter.printTrailerLine(trailerStringList);
        ...
    } finally {
        // ファイルのクローズ
        fileLineWriter.closeFile();
    }
...(以下略)
```

ファイル入力の際と同様に @Inject アノテーションを使用して、ビジネスロジックへの DAO の DI を行う。

ファイル名とパラメータクラスを引数に、ファイル出力用行ライタを取得する。

ヘッダ部を出力する。
String 型のリストを引数とする。

ファイル形式に関わらず、printDataLine メソッドで出力する。
出力される項目には、項目定義用のアノテーションを付加しておく。

トレイラ部を出力する。
String 型のリストを引数とする。

出力が終了したら、ファイルをクローズする。

  - ファイルの出力順序

    ヘッダ部の出力は、データ部の出力の前に行う必要がある点に留意すること。
    同様にトレイラ部の出力は、データ部の出力がすべて終わった後に行う必要がある点に留意すること。

◆ 拡張ポイント

- なし

■ 関連機能

- 『AL-043 入力チェック機能』
- 『BL-04 例外ハンドリング機能』

■ 使用例

- 機能網羅サンプル(terasoluna-batch-functionsample)
- チュートリアル(terasoluna-batch-tutorial)

■ 備考

- ファイル入出力 DAO を使用しない方法(非推奨)

ファイル入出力処理を行うビジネスロジックの実装方法には、これまで説明した方法のほかに、ビジネスロジックに直接ファイル入力用行イテレータやファイル出力用行ライタをインジェクションしてファイル入出力処理を行う方法もある。
以下に、直接ファイル入力用行イテレータを設定する例を述べる。

✧ ジョブ Bean 定義ファイル設定例

```
<bean class="jp.terasoluna.fw.file.dao.standard.CSVFileLineIterator"
        destroy-method="closeFile">
    <constructor-arg index="0" value="some_file_path/uriage.csv" />
    <constructor-arg index="1"
          value="jp.terasoluna.batch.sample.b000001.UriageFileLineObject" />
    <constructor-arg index="2" ref="columnParserMap" />
</bean>
```

第 1 引数：ファイル名
第 2 引数：ファイル行オブジェクトのフルパス
第 3 引数："columnParserMap"固定

✧ ビジネスロジック実装例

```
@Component
public class B001001BLogic implements BLogic {

    @Inject
    @Named("iterator")
    FileLineIterator fileLineIterator = null;

    public int execute(BLogicParam arg0) {
        ...
        try {
            // ヘッダ部の読み込み
            List<String> headerData = fileLineIterator.getHeader();
            ... // 読み込んだヘッダ部に対する処理

            while(fileLineIterator.hasNext()){
                // データ部の読み込み
            }
            // トレイラ部の読み込み
        } finally {
            // ファイルのクローズ
            fileLineIterator.closeFile();
        }
... (以下略)
```

ジョブ Bean 定義ファイルに定義した FileLineIterator をビジネスロジックに直接 DI する。

DI した FileLineIterator を使用してファイル入力処理を実装する。

両者の違いは、『AL-041 入力データ取得機能』の使用可否と、ファイルのオープン/クローズのタイミング[1]にある。ファイル入力用行イテレータを直接使用する場合は、『AL-041 入力データ取得機能』が使用できないため、推奨しない。

[1] ファイル入力用行イテレータやファイル出力用行ライタの生成/クローズ時にファイルのオープン/クローズが生じる。そのため、ファイル入出力 DAO を使用する場合は、ビジネスロジックで生成/クローズしたタイミングとなる。一方、ファイル入力用行イテレータやファイル出力用行ライタを直接使用する場合は、アプリケーションコンテキストの生成/破棄のタイミングとなる。