COBOL マイグレーション（opensource COBOL 他）

目次

【改変履歴】
2023/8/末 項番 7 以降に SQL による集計例と Java による実装例を追加

COBOL マイグレーション（opensource COBOL 他）

はじめに

汎用機（メインフレーム）のシステムは COBOL や PL/I で作られているもの多かったのですが、時代の趨勢でハードウェアをサーバで置換する際、ソフトウェアについても性能が高くなりそうな言語（例えば C）や将来性のありそうな言語（例えば Java）を選択して再開発する例が多くありました。

なかには計算を含む高速なリアルタイム処理が必要だからという理由で C 言語を選択し、十進数を扱えるライブラリが無いのに選んだ DBMS は使える数値が外部十進数と内部十進数だけだったという笑えない事例もあります。また、COBOL から Java へステップ バイ ステップの命令置換を行う木に竹を接ぐようなことも行われています。

サーバの COBOL 実行環境は当時も存在し今も開発が続けられています。作った当時の背景が良く分からないけど日常使っているような古い資産は作り替えるよりも新システムからデータを抽出して COBOL へのインタフェースを考えた方が費用対効果は良くなるかもしれません。また、将来改修が入りそうな機能でも COBOL の要員が確保できるなら当面そのまま運用することも可能です。どれにも該当しなければ、オブジェクト指向で再設計した方が効率も良く人員調達も容易になります。

1. COBOL 仕様とコンパイラ

2023 年 7 月現在まで COBOL の仕様は改定が続いています[1]。

機能としては COBOL 2002 でオブジェクト指向プログラミングのサポートが追加され、2014 標準で浮動小数点に関する機能強化や TRIM 組込関数[2]の追加が行われました。

コンパイラは幾つかのコンピュータ・メーカが自社 OS 用に、複数 OS 向けにマイクロフォーカス社（とその後継会社）が出荷しています。オープンソースでも OpenCOBOL[3]をベースに、OSS コンソーシアムが opensource COBOL[4]、Free Software Foundation, Inc.が GnuCOBOL を公開しています。GnuCOBOL は COBOL 標準 (2002、2014、2023) の大部分と他社コンパイラの拡張機能の多くも実装していると表明しています。

2. 移行のパターン…仕様、方式、機能、ソース

メインフレームからサーバに独自開発のシステムを移すとき、以下のパターンが考えられます。

① 現在、将来とも改修要件が発生する見込みはない：オンラインの Web 化やデータベースの再構成、他システムとのインターフェース部分を除き、既存の COBOL ソースをそのまま活かす。
（但し、データベースの再構成は広範に影響するので埋め込み SQL やアクセスルーチンでテーブルを個別にアクセスしている箇所が多いのであれば新たな方式に作り替えた方が製造・試験・保守の各工程が安上がりになる可能性が高い）

② 重要な基幹系システムで改修の即時性が求められる：要員を集め易い（扱い易い）言語や環境に機能をそのまま再構築する。更に、重要な機能はオブジェクト指向で再設計して保守性を維持したい。

---

[1] ISO/IEC 1989:2023 Programming language COBOL https://www.iso.org/standard/74527.html

[2] COBOL コンソーシアム http://www.cobol.gr.jp/knowledge/technical.html

[3] 平成１４年度 未踏ソフトウェア創造事業「未踏ユース」 オープンソースＣＯＢＯＬコンパイラ
https://www.ipa.go.jp/archive/files/000005635.pdf

[4] opensource COBOL https://github.com/opensourcecobol

COBOL マイグレーション（opensource COBOL 他）

③ ハードの性能・容量の制約で実現できなかった大規模な部品組立業の日程計画リアルタイム化や AI 利用、小売り決済のバッチからリアルタイム化、RDBMS から NoSQL への変更等の方式自体を変える場合は方式や業務に合った開発言語でシステムを作り替えるかパッケージの利用になります。それでも、一定の割合の既存プログラムは①と同様に流用できる可能性があります。

④ 企業間や消費者との決済条件や物品の受け渡し条件が変わる場合は仕様の追加や廃止、場合によっては他システムに必要な処理を移管して当該システムの廃止を検討した方が良い場合があります。

以降、①の候補になり得る環境や②で使える可能性があるツールを紹介します。

3. opensource COBOL とは

2023/7 現在、opensource COBOL は 2019 年の更新（v1.5.2J）が最後で近年は GnuCOBOL ほど活発に更新されていませんが、Docker のイメージで公開されているので簡単に使うことができます。

COBOL85 の仕様と日本語の拡張を謳っているので、実際に試行した結果を以下に記載します。

【使用環境】

・エディション　Windows 10 Pro
・バージョン　22H2
・OS ビルド　19045.3208
・エクスペリエンス　Windows Feature Experience Pack 1000.19041.1000.0

※Docker コンテナの実行は Rancher Desktop 1.9.1.0

【COBOL 開発手順】

opensource COBOL のコンテナには日本語の環境（locale）が入っていないのとインストールされているエディターが少し慣れが必要な vim だけなので、ソースの編集やデータの準備は Windws 側で行いコンパイルと実行をコンテナで行う手順を前提に説明します。

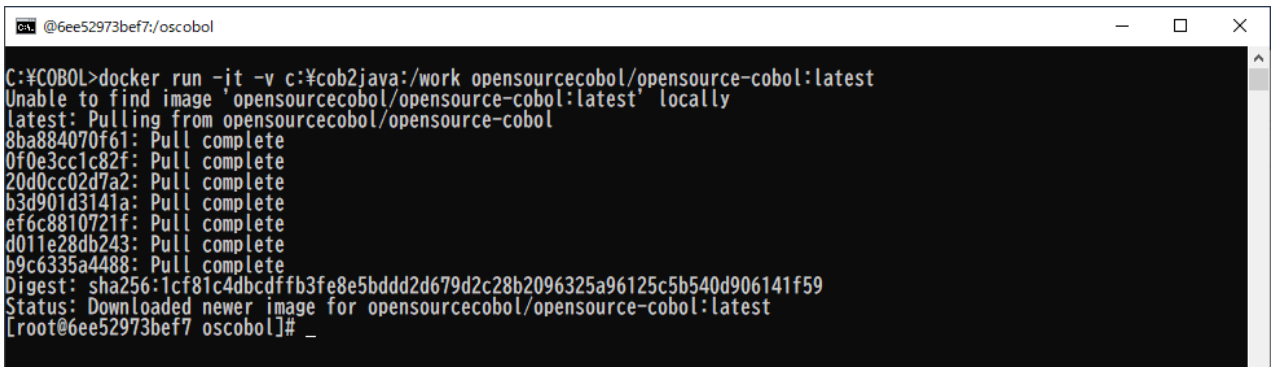
① コンテナのイメージ取得（初回）と起動

コマンドプロンプト等から以下のコマンドを実行します。

```
docker run -it -v c:¥cob2java:/work opensourcecobol/opensource-cobol:latest
```

c:¥cob2java は Windows 側で COBOL ソースの編集等の作業を行うディレクトリでコンテナ側からは/work ディレクトリとして参照する共用ディレクトリです。（ディレクトリ名は各 OS が許す範囲で自由に付けられます）

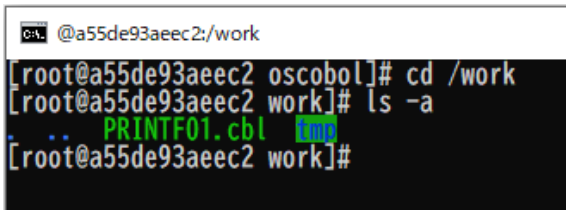
このコマンドを実行すると初回（イメージがローカルにない）だけ最新のイメージのダウンロードと解凍が行われ、起動したコンテナに接続した状態になります。



COBOL マイグレーション（opensource COBOL 他）

② 共用ディレクトリの資材

Windows（左）とコンテナ（右）の共用ディレクトリで同一の資材を参照します。

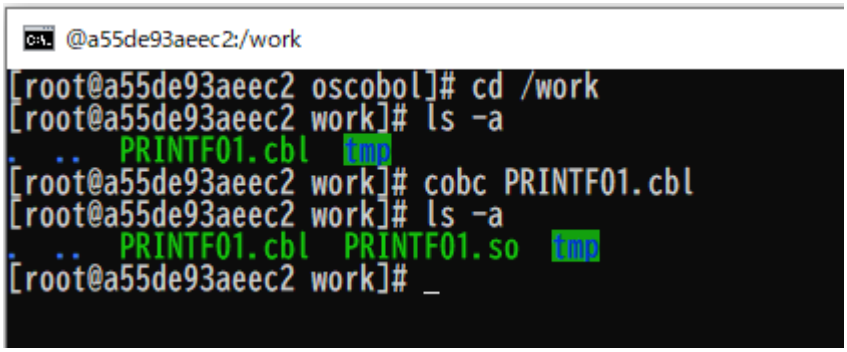


③ コンパイル

コンテナでソースが格納されている共用ディレクトリに移動して、以下のコマンドを実行します。

```
cobc PRINTF01.cbl
```

この例では、”PRINTF01.cbl”が COBOL のソースファイル名です。致命的なエラーがなければ、拡張子 so のファイル（Linux の共有ライブラリ）が作られます。

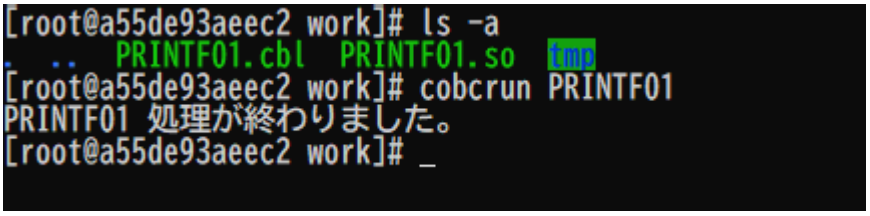


④ 実行

コンパイルが正常に終了し so ファイルが作られたら、以下のコマンドを実行します。

```
cobcrun PRINTF01
```

cobcrun が COBOL の実行環境を準備してパラメタで指定した共有ライブラリ（コンパイル結果）の呼出しを行います。



COBOL マイグレーション（opensource COBOL 他）

4. COBOL ソース

メインフレームの COBOL と opensource COBOL では実行環境の違いによりいくつか変更が必要で、以下が代表的な部分です。

4.1. 入出力レコード

メインフレームで扱うデータセットには物理的な IO の単位になるブロックサイズが書かれており、レコードは主に固定長かレコードの先頭に長さが書かれた可変長レコードが連続して格納されます。

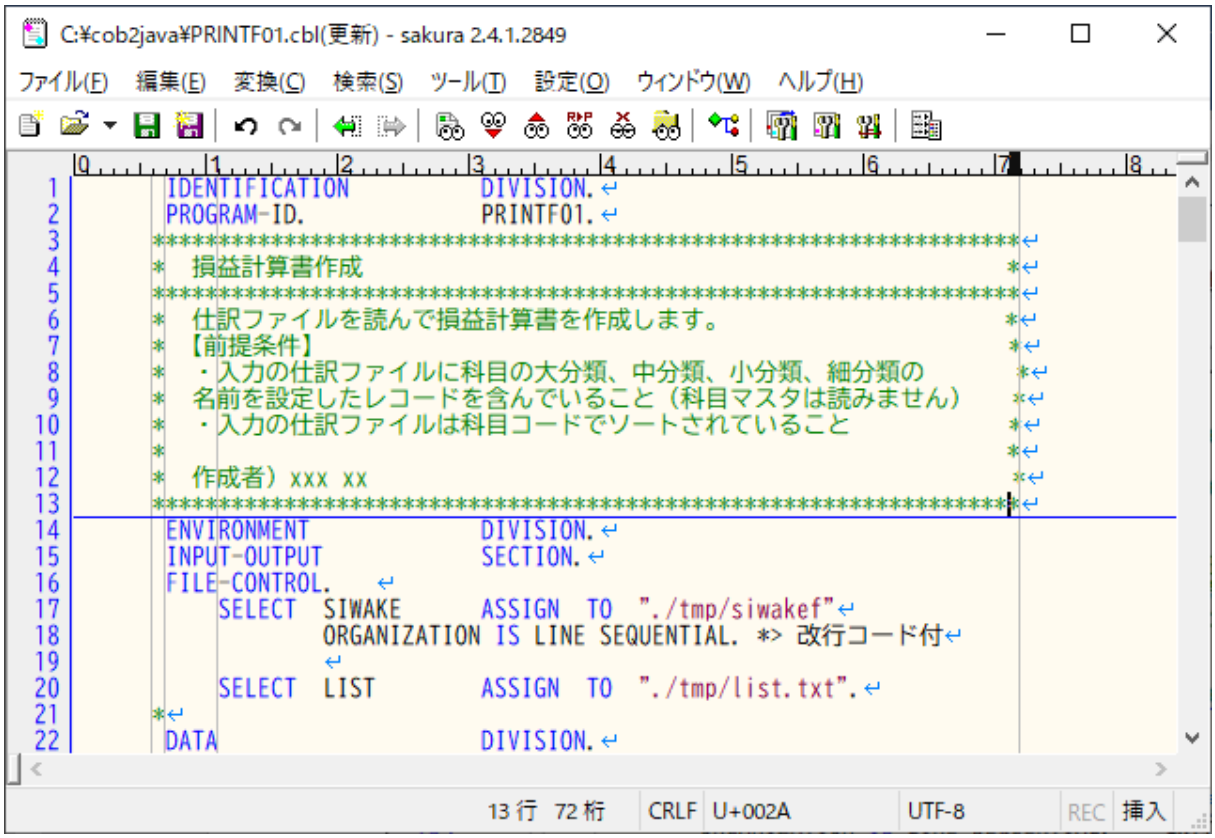
サーバの方は、制御記号「改行」でレコードを区切ります。固定長で扱うことも可能ですが、日本語を含む場合は 1 文字のバイト数が異なる（場合がある）ので実際に固定長で取り扱うのは困難です。

4.2. 文字コード

Windows では Windows-31J（MS932）という Shift_JIS を拡張した文字コードを使い、コンテナの Linux はデフォルトが UTF-8 です。Shift_JIS は半角 1 バイトと全角 2 バイトで全ての文字を表現し、UTF-8 は英数字の ASCII コードは 1 バイト、Shift_JIS と同じ範囲の文字は 3 バイト、JIS X 0213 の第三水準、第四水準にあたる文字は 4 バイトで符号化しています。

4.3. COBOL コード例

cobc の入力ソースはデフォルトでは 7 桁目がコメント標識欄、11 桁目まで A 領域、72 桁目までが B 領域と呼ばれる固定形式です。



COBOL マイグレーション（opensource COBOL 他）

＜サンプル処理内容＞

・以下のテーブルに登録されている科目毎に借方／貸方の残高を集計して損益計算書を作成します。

・科目コードは大分類 1 桁、中分類 1 桁、小分類 2 桁、細分類 2 桁（無い場合もある）の科目名が全て登録されている。

・最初に RDBMS の操作ツール psql でテーブルを読んで固定長レコードの仕訳ファイルを作り、プログラムの入力ファイルとします。

＜テーブルの仕様＞

テーブル仕様書	テーブル名	勘定科目マスタ	作成	2005/9/30	担当		項目数
	TABLE_NAME	m_kamoku	修正		担当		5

〔備考〕 勘定科目の情報を管理する

インデックス	キー名	一意
p	pk_m_kamoku	○
s1		
s2		
s3		

外部キー	制約名	参照先
1		
2		
3		
4		

No.	FIELD_NAME	日本語名	型	日本語	桁数	小数	必須	IDX P	IDX s1	IDX s2	IDX s3	外部キー	備考
1	kamokucd	科目ｺｰﾄﾞ	CHAR		6		○	1					
2	kamokunm	科目名	VARCHAR	○	30		○						
3	dennyukb	伝票入力可否区分	CHAR		1		○						1:伝票入力有り、0:無し
4	taisyakukb	貸借区分	CHAR		1		○						1:借方、2:貸方
5	chohyokb	対象帳票区分	CHAR		1		○						1:貸借、2:損益、3:利益処分
6													

＜実際に登録されているデータ＞

```
postgres=# select * from m_kamoku;
 kamokucd |                kamokunm                | dennyukb | taisyakukb | chohyokb
----------+---------------------------------------+----------+------------+----------
 000000   | 諸口                                  | 0        | 3          | 0
 100000   | 流動資産                              | 0        | 1          | 1
 110000   | 当座資産                              | 0        | 1          | 1
 110100   | 現金及び預金                          | 0        | 1          | 1
 110101   | 現金                                  | 1        | 1          | 1
 110102   | 預金                                  | 1        | 1          | 1
 110103   | 普通預金                              | 1        | 1          | 1
 110104   | 定期預金                              | 1        | 1          | 1
 110105   | 他預金                                | 1        | 1          | 1
 110200   | 受取手形                              | 0        | 1          | 1
 110201   | 受取手形（関係会社のものを除く）      | 1        | 1          | 1
 110207   | 子会社受取手形                        | 1        | 1          | 1
 110208   | その他の関係会社受取手形              | 1        | 1          | 1
 110209   | 貸倒引当金（受取手形）                | 1        | 2          | 1
 110300   | 売掛金                                | 0        | 1          | 1
 110301   | 売掛金（関係会社のものを除く）        | 1        | 1          | 1
 110307   | 子会社売掛金                          | 1        | 1          | 1
 110308   | その他の関係会社売掛金                | 1        | 1          | 1
 110309   | 貸倒引当金（売掛金）                  | 1        | 2          | 1
 110400   | 有価証券                              | 1        | 1          | 1
 110500   | 自己株式                              | 1        | 1          | 1
 110600   | 親会社株式                            | 1        | 1          | 1
 120000   | たな卸資産                            | 0        | 1          | 1
```

＜ソースの全体＞

バッチ処理の代表的なパターンのコントロールブレークになっています。但し、一般的なパターンでは入力ファイルには大分類や中分類の集計単位のレコードを含みません(134、163 行目が不要)。

```
1         IDENTIFICATION          DIVISION.
2         PROGRAM-ID.             PRINTF01.
3        *****************************************************************
4        *  損益計算書作成                                               *
5        *****************************************************************
6        *  仕訳ファイルを読んで損益計算書を作成します。                 *
7        *  【前提条件】                                                 *
```


COBOL マイグレーション（opensource COBOL 他）

```
*  ・入力の仕訳ファイルに科目の大分類、中分類、小分類、細分類の       *
*  名前を設定したレコードを含んでいること（科目マスタは読みません）   *
*  ・入力の仕訳ファイルは科目コードでソートされていること             *
*                                                                *
*  作成者）xxx xx                                                *
******************************************************************
 ENVIRONMENT             DIVISION.
 INPUT-OUTPUT            SECTION.
 FILE-CONTROL.
     SELECT  SIWAKE      ASSIGN  TO  "./tmp/siwakef"
             ORGANIZATION IS LINE SEQUENTIAL. *> 改行コード付

     SELECT  LIST        ASSIGN  TO  "./tmp/list.txt".
*
 DATA                    DIVISION.
 FILE                    SECTION.
*仕訳ファイル
 FD  SIWAKE.
 01  SIWAKE-REC          PIC X(200).

*損益計算書
 FD  LIST.
 01  LIST-REC            PIC X(103).
 01  LIST-CRLF           PIC X(02).

*
 WORKING-STORAGE         SECTION.
 01  SIWAKE-WK.
     03  SIWAKE-KAMOKUCD     PIC  X(06).
     03  KEY-BREAK           REDEFINES  SIWAKE-KAMOKUCD.
         05  KEY-L3
           07  KEY-L2.
             09  KEY-DAI     PIC X.
             09  KEY-CHU     PIC X.
           07  KEY-SYO       PIC XX.
         05  KEY-SAI         PIC XX.
     03  SIWAKE-PKINGAKU     PIC  9(13).
     03  SIWAKE-LKINGAKU     PIC  9(13).
     03  SIWAKE-DENNYUKB     PIC  X.
         88  KBN-DENNYU      VALUE "1".
         88  KBN-NODEN       VALUE "0".
     03  SIWAKE-TAISYAKUKB   PIC  X.
         88  KBN-KARI        VALUE "1".
         88  KBN-KASI        VALUE "2".
     03  SIWAKE-CHOHYOKB     PIC  X.
         88  NOTPL           VALUE "1","3".
         88  PL              VALUE "2".
     03  SIWAKE-KAMOKUNM     PIC  X(90).
*
 01  PR-MIDASHI.
     03  FILLER              PIC X(10) VALUE SPACE.
     03  FILLER              PIC X(20) VALUE "損益計算書".
*
 01  PR-MEISAI.
     03  PR-L1KAMOKUNM.
       05  FILLER            PIC X(03).
       05  PR-L2KAMOKUNM.
         07  FILLER          PIC X(03).
         07  PR-KAMOKUNM     PIC X(60).
     03  PR-GMIDASI          PIC X(06).
     03  PR-LKINGAKU         PIC ---,---,---,--9.
     03  PR-PKINGAKU         PIC ---,---,---,--9.
*合計行分離線
 01  PR-SEP
     03  FILLER    PIC X(20) VALUE "--------------------".
     03  FILLER    PIC X(20) VALUE "--------------------".
     03  FILLER    PIC X(20) VALUE "--------------------".
     03  FILLER    PIC X(20) VALUE "--------------------".
```



```
      *
      *科目コード
       01  SV-KAMOKU.
           03  SV-L3
             05  SV-L2.
               07  SV-DAI      PIC X.
               07  SV-CHU      PIC X.
             05  SV-SYO        PIC XX.
           03  SV-SAI          PIC XX.
      *
       01  SV-KAMOKUNM.
           03  SV-DAIKAMOKU    PIC X(60).
           03  SV-CHUKAMOKU    PIC X(60).
      *
       01  SUMA.
           03  SUM-L1          PIC S9(13) COMP-3.
           03  SUM-L2          PIC S9(13) COMP-3.
      *
       01  WORK.
           03  WK-MEIKINGAKU   PIC S9(13) COMP-3.
      *Windows 用改行コード
       01  CRLF                PIC X(02) VALUE X"0D0A".
      *
       PROCEDURE               DIVISION.
      ****************************************************************
      * 損益計算書作成メイン                                         *
      ****************************************************************
       000-SONEKIKEISAN        SECTION.
       000-RTN.
           PERFORM  100-INIT.
           PERFORM  200-MAIN   UNTIL KEY-BREAK = HIGH-VALUE.
           PERFORM  300-TERMINATE..
       000-END.
           STOP RUN.
      ****************************************************************
      * 前処理                                                       *
      ****************************************************************
       100-INIT                SECTION.
       100-RTN.
           OPEN  INPUT   SIWAKE.
           OPEN  OUTPUT  LIST.
           WRITE LIST-REC      FROM  PR-MIDASHI AFTER PAGE.
           WRITE LIST-CRLF     FROM    CRLF.

           PERFORM S00-READ.
           MOVE  KEY-BREAK     TO  SV-KAMOKU.
       100-END.
           EXIT.
      ****************************************************************
      * 主処理（L1:大分類）                                          *
      ****************************************************************
       200-MAIN                SECTION.
       200-RTN.
      *    大分類レベルの処理
           MOVE  SIWAKE-KAMOKUNM   TO  SV-DAIKAMOKU.
           MOVE  SPACE             TO  PR-MEISAI.
           MOVE  SIWAKE-KAMOKUNM   TO  PR-L1KAMOKUNM.
           WRITE LIST-REC      FROM    PR-MEISAI AFTER 2.
           PERFORM  S00-READ.                  *> 中分類レコード読込み

           MOVE  ZERO          TO  SUM-L1.
           MOVE  KEY-BREAK     TO  SV-KAMOKU.
           PERFORM  210-L2     UNTIL KEY-BREAK   = HIGH-VALUE
                               OR    KEY-DAI NOT = SV-DAI.
      *    合計出力
           MOVE  SPACE         TO  PR-MEISAI.
           MOVE  "合計"        TO  PR-GMIDASI.
           MOVE  SV-DAIKAMOKU  TO  PR-L1KAMOKUNM.
```

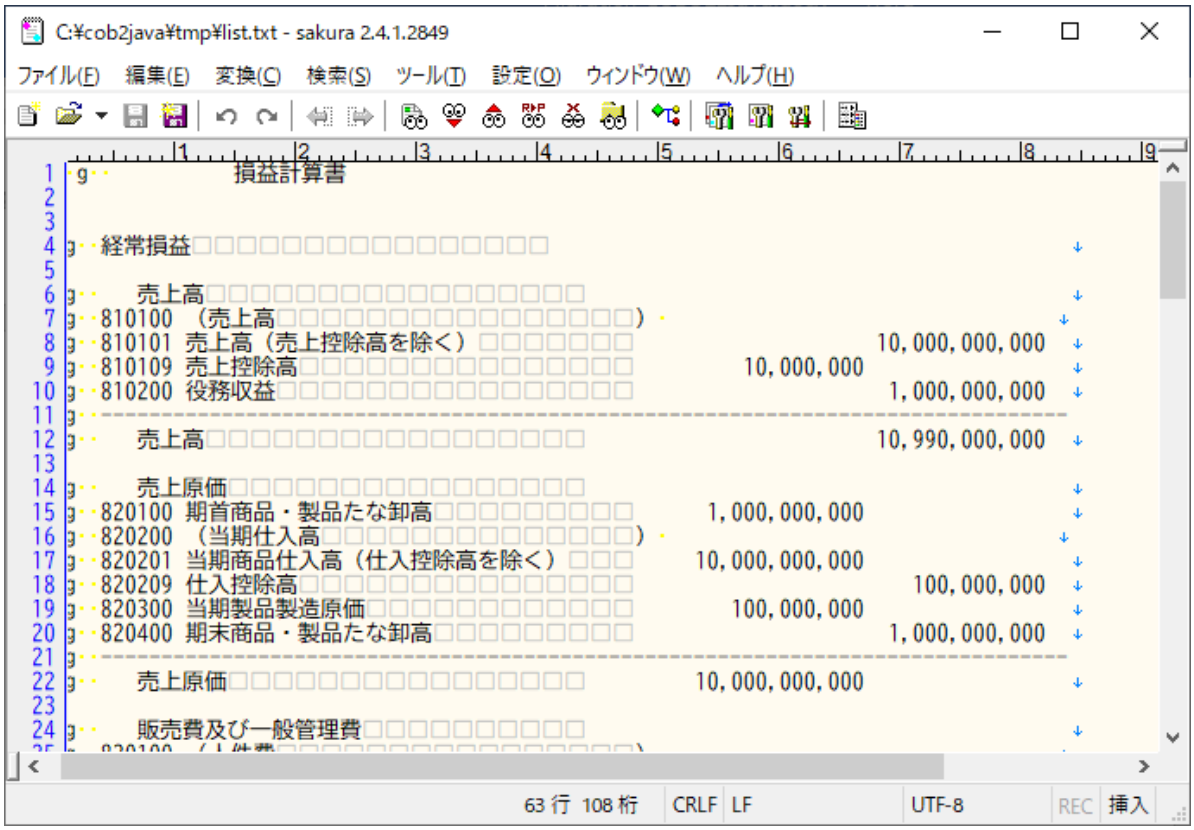


```
144             IF  SUM-L1          <   0
145                 COMPUTE PR-LKINGAKU =   SUM-L1 * -1
146             ELSE
147                 MOVE  SUM-L1        TO  PR-PKINGAKU
148             END-IF.
149             WRITE LIST-REC      FROM    PR-SEP AFTER 1.
150             WRITE LIST-REC      FROM    PR-MEISAI AFTER 1.
151          200-END.
152             EXIT.
153         ****************************************************************
154         * L2:中分類                                                    *
155         ****************************************************************
156          210-L2                 SECTION.
157          210-RTN.
158         *    中分類レベルの処理
159             MOVE  SIWAKE-KAMOKUNM   TO  SV-CHUKAMOKU.
160             MOVE  SPACE             TO  PR-MEISAI.
161             MOVE  SIWAKE-KAMOKUNM   TO  PR-L2KAMOKUNM.
162             WRITE LIST-REC      FROM    PR-MEISAI AFTER 2.
163             PERFORM  S00-READ.                  *> 小分類レコード読込み
164
165             MOVE  ZERO          TO  SUM-L2.
166             MOVE  KEY-BREAK     TO  SV-KAMOKU.
167             PERFORM  220-L3     UNTIL KEY-BREAK   = HIGH-VALUE
168                                 OR    KEY-L2  NOT = SV-L2.
169             ADD   SUM-L2        TO  SUM-l1.
170         *    合計出力
171             MOVE  SPACE         TO  PR-MEISAI.
172             MOVE  SV-CHUKAMOKU  TO  PR-L2KAMOKUNM.
173             IF  SUM-L2          <   0
174                 COMPUTE PR-LKINGAKU =   SUM-L2 * -1
175             ELSE
176                 MOVE  SUM-L2        TO  PR-PKINGAKU
177             END-IF.
178             WRITE LIST-REC      FROM    PR-SEP AFTER 1.
179             WRITE LIST-REC      FROM    PR-MEISAI AFTER 1.
180          210-END.
181             EXIT.
182         ****************************************************************
183         * L3:小／細分類                                                *
184         ****************************************************************
185          220-L3                 SECTION.
186          220-RTN.
187         *    小分類、細分類レベルの処理
188             MOVE  SIWAKE-KAMOKUCD   TO  PR-MEISAI.
189         *    伝票入力が有る科目だけ貸方／借方発生側に残高表示
190             IF  KBN-DENNYU
191                 MOVE  SIWAKE-KAMOKUNM       TO  PR-KAMOKUNM
192                 IF  KBN-KARI
193                     MOVE     SIWAKE-PKINGAKU TO    WK-MEIKINGAKU
194                     SUBTRACT SIWAKE-LKINGAKU FROM  WK-MEIKINGAKU
195                     MOVE     WK-MEIKINGAKU   TO    PR-LKINGAKU
196                     SUBTRACT WK-MEIKINGAKU   FROM  SUM-L2
197                 ELSE
198                     MOVE     SIWAKE-LKINGAKU TO    WK-MEIKINGAKU
199                     SUBTRACT SIWAKE-PKINGAKU FROM  WK-MEIKINGAKU
200                     MOVE     WK-MEIKINGAKU   TO    PR-PKINGAKU
201                     ADD      WK-MEIKINGAKU   TO    SUM-L2
202                 END-IF
203             ELSE
204                 STRING
205                     " ("            DELIMITED BY SIZE
206                     SIWAKE-KAMOKUNM DELIMITED BY SIZE
207                     INTO PR-KAMOKUNM
208                 END-STRING
209                 MOVE  ") "          TO  PR-GMIDASI
210             END-IF.
211
```



```
212             WRITE LIST-REC      FROM    PR-MEISAI AFTER 1.
213             PERFORM  S00-READ.                  *> 次レコード読込み
214          220-END.
215             EXIT.
216         ****************************************************************
217         * 後処理                                                       *
218         ****************************************************************
219          300-TERMINATE           SECTION.
220          300-RTN.
221             DISPLAY "PRINTF01 処理が終わりました。".
222             CLOSE  SIWAKE LIST.
223          300-END.
224             EXIT.
225         ****************************************************************
226         * READ 処理
227         ****************************************************************
228          S00-READ                SECTION.
229          S00-RTN.
230             READ  SIWAKE    INTO  SIWAKE-WK
231                 AT  END
232                     MOVE  HIGH-VALUE        TO  KEY-BREAK
233             END-READ
234
235          *   科目の対象帳票区分が「損益計算書」の仕訳データだけ選びます。
236             PERFORM UNTIL       KEY-BREAK =   HIGH-VALUE
237                     OR          PL
238                 READ  SIWAKE    INTO  SIWAKE-WK
239                     AT  END
240                         MOVE  HIGH-VALUE        TO  KEY-BREAK
241                 END-READ
242             END-PERFORM.
243          S00-END.
244             EXIT.
```

＜プログラムの出力＞



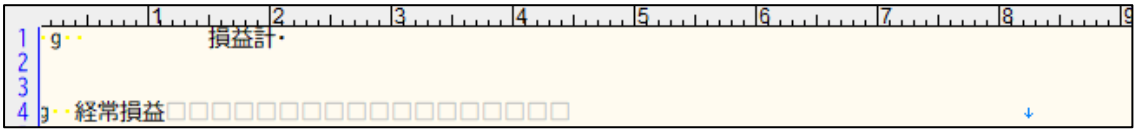
COBOL マイグレーション（opensource COBOL 他）

4.4. opensurceCOBOL 日本語処理の不具合

opensurce COBOL は日本語向けの拡張を謳っており、確かに PIC N が使えますが UTF-8 への対応は不完全のようです（2 バイト／1 文字で計算している？）。

59 行目を　03　FILLER　　PIC N(05) VALUE N”損益計算書”. とすると以下の出力になります。

（または　03　FILLER　　PIC N(05) VALUE　”損益計算書”. としても同様に）



5. RDB から固定長ファイルの作成方法

Windows で UTF-8 のデータ作ることと固定長のレコードを作るには一定の手順が必要になります。

以下が PostgreSQL を使って UTF-8 の固定長のレコードを出力する例です。

① psql の環境設定とデータ問合せ SQL（コメント含む）を UTF-8 で書いたファイルを作ります

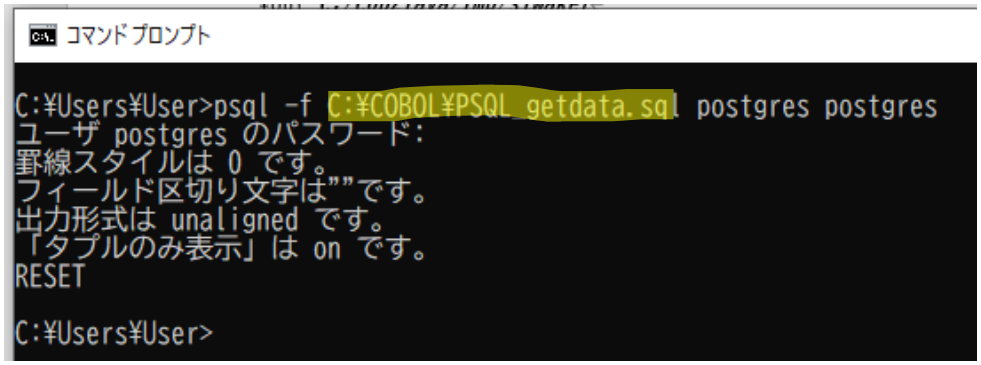
――コマンドプロンプトでは入力を MS932 に変換し、PostgreSQL はクライアントの文字コード（Windows-MS932）への変換処理を行うので、会話形式では UTF-8 のデータを作れません――

＜ファイルの内容＞

```
--罫線を非表示
¥pset border 0
--項目区切文字を空にする
¥pset fieldsep ''
--出力位置合わせをしない
¥pset format unaligned
--見出し、フッター出力を停止
¥pset tuples_only
--出力文字符号を UTF-8
¥encoding UTF-8
--出力先ファイルを指定
¥out C:/cob2java/tmp/siwakef
SELECT
      k.kamokucd
    , to_char(coalesce((SELECT sum(kingaku) FROM  t_siwake_meisai
                        WHERE kamokucd = k.kamokucd
                        AND   taisyakukb = '1'
      ),0),'FM0999999999999') 借計     --仕訳明細の登録が無ければ 0
    , to_char(coalesce((SELECT sum(kingaku) FROM  t_siwake_meisai
                        WHERE kamokucd = k.kamokucd
                        AND   taisyakukb = '2'
      ),0),'FM0999999999999') 貸計     --仕訳明細の登録が無ければ 0
    , k.dennyukb
    , k.taisyakukb
    , k.chohyokb
    , rpad(k.kamokunm, 30, '　') --30 文字まで UTF-8 の全角空白で埋める（'　'は UTF-8 で設定）
FROM m_kamoku k
ORDER BY k.kamokucd
;
--出力先をデフォルト（ターミナル）に戻す
¥out
--出力するデータの文字コードをデフォルトに変更
reset client_encoding;
```

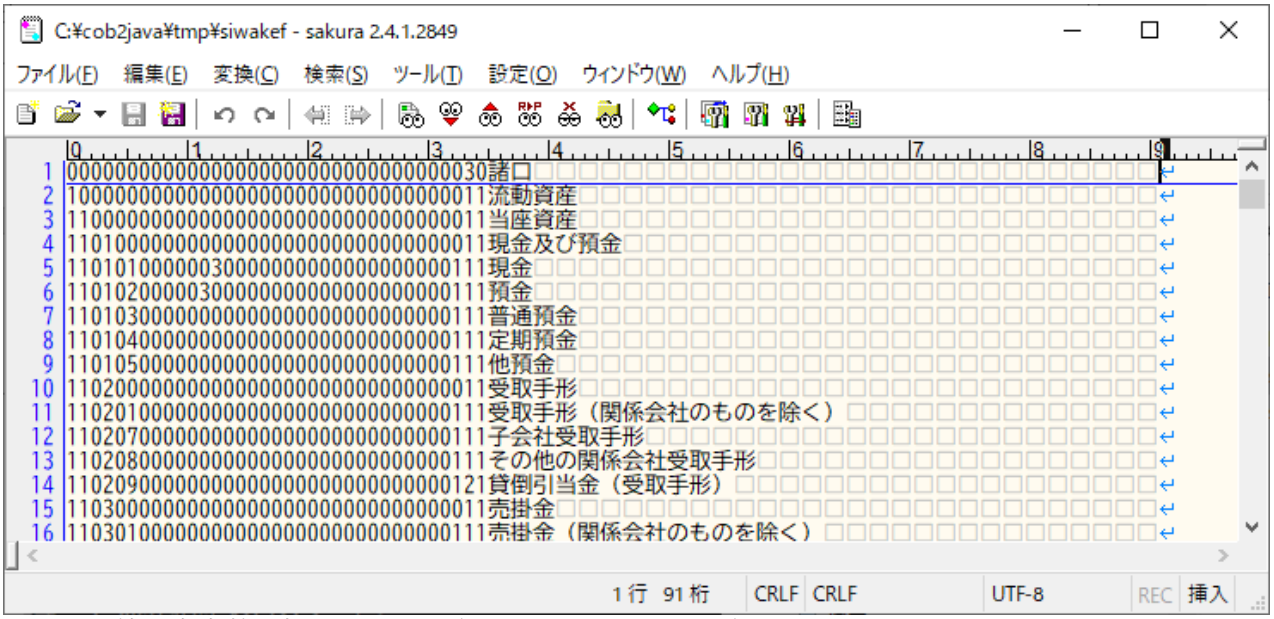

② psql の-f パラメータとして①のファイルを渡します。

psql -f *①ファイル名　データベース名　ユーザ名*



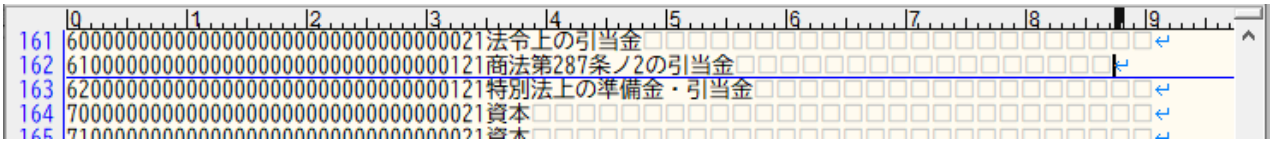
＜抽出結果＞

等幅フォントで表示しているので科目名の表示が 半角 2 桁×30 文字でエディター上は 1 行が 90 桁になっていますが、バイト数では 124 になり COBOL の入力レコードの定義でもその分が必要です。



ASCII 以外も文字数で扱うので、混在するとレコード長が短くなります。

COBOL は READ INTO を使うと不足分は ASCII の空白になり、不足領域への参照を避けられます。



COBOL マイグレーション（opensource COBOL 他）

6. COBOL から Java への変換ツール

opensource COBOL を基に opensource COBOL 4J が開発され（ライセンスは GPL-3、LGPL-3 混）ています。これも Docker イメージが公開されているので簡単に使うことができます。

① コンテナのイメージ取得（初回）と起動

コマンドプロンプト等から以下のコマンドを実行します。

```
docker run -it -v c:¥cob2java:/work opensourcecobol/opensourcecobol4j:latest
```

c:¥cob2java は Windows 側で COBOL ソースの編集等の作業を行うディレクトリでコンテナ側からは/work ディレクトリとして参照する共用ディレクトリ・・・等々は opensourceCOBOL と全く同様。

② 変換ツールの実行（Java ソースの生成）

opensource COBOL を基にしているので COBOL85 としてコンパイルエラーがでない資材が対象になります。以下のコマンドを実行します。

```
cobj ソースファイル名
```

前出の COBOL ソースに対して実行すると、以下のエラーが出ますが Java ソースは生成されます。

```
root@9c386518794a: /work

C:¥COBOL>docker run -it -v c:¥cob2java:/work opensourcecobol/opensourcecobol4j:latest
root@9c386518794a:~# cd /work
root@9c386518794a:/work# cobj PRINTF01.cbl
PRINTF01.cbl:60: Warning: Value size exceeds data size
./PRINTF01.java:994: error: unmappable character (0xEF) for encoding SJIS
  public static final byte[] str_literal_6 = CobolUtil.stringToBytes("???");

./PRINTF01.java:994: error: unmappable character (0x88) for encoding SJIS
  public static final byte[] str_literal_6 = CobolUtil.stringToBytes("???");

./PRINTF01.java:998: error: unmappable character (0xEF) for encoding SJIS
  public static final byte[] str_literal_2 = CobolUtil.stringToBytes("???   ");

./PRINTF01.java:998: error: unmappable character (0x89) for encoding SJIS
  public static final byte[] str_literal_2 = CobolUtil.stringToBytes("???   ");

4 errors
root@9c386518794a:/work# dir
PRINTF01.cbl  PRINTF01.java  PRINTF01.so  tmp
root@9c386518794a:/work#
```

＜生成された Java ソース＞

```
import java.io.UnsupportedEncodingException;
import jp.osscons.opensourcecobol.libcobj.*;
import jp.osscons.opensourcecobol.libcobj.common.*;
import jp.osscons.opensourcecobol.libcobj.data.*;
import jp.osscons.opensourcecobol.libcobj.exceptions.*;
import jp.osscons.opensourcecobol.libcobj.termio.*;
import jp.osscons.opensourcecobol.libcobj.call.*;
import jp.osscons.opensourcecobol.libcobj.file.*;
import jp.osscons.opensourcecobol.libcobj.ui.*;
import java.util.Optional;

public class PRINTF01 implements CobolRunnable {
```


COBOL マイグレーション（opensource COBOL 他）

```
  private boolean initialized = false;
  private CobolModule cobolCurrentModule;
  private CobolModule module;
  private CobolFrame frame;
  private static boolean cobolInitialized = false;
  private CobolCallParams cobolSaveCallParams = null;
  private CobolCallParams cobolCallParams = null;
  private boolean cobolErrorOnExitFlag;
  private int entry;

  private CobolRunnable cob_unifunc;


  @Override
  public int run(CobolDataStorage... argStorages) {
    return PRINTF01_(0, argStorages);
  }

  @Override
  public void cancel() {
    PRINTF01_(-1);
  }

  @Override
  public boolean isActive() {
    return false;
  }

  public CobolResultSet execute () {
    int returnCode = run_module(0);
    return new CobolResultSet(returnCode);
  }

  public int PRINTF01_ (int entry, CobolDataStorage ...argStorages) {
    this.entry = entry;
    return this.run_module(entry);
  }

～【中略】～

  public static void main(String[] args)
  {
    CobolUtil.cob_init(args, cobolInitialized);
    CobolDecimal.cobInitNumeric();
    new PRINTF01().PRINTF01_(0);
    CobolStopRunException.stopRun();
  }

  public PRINTF01()
  {
    init();
  }
～【後略】～
```


③ Java コンパイル（javac）

ツールの実行過程でエラーが出ているのは java ソースが生成された後に Java のコンパイルが行われ、ソース中の文字コード（UTF-8）が正しく認識できずにコンパイルが失敗しています。

以下のように使っている文字コードをパラメータで指定すると、Java のコンパイルが通ります。

```
javac -encoding UTF-8 PRINTF01.java
```

```
root@9c386518794a:/work# dir
PRINTF01.cbl  PRINTF01.java  PRINTF01.so  tmp
root@9c386518794a:/work# javac -encoding UTF-8 PRINTF01.java
root@9c386518794a:/work# dir
PRINTF01$1.class   PRINTF01$14.class  PRINTF01$19.class  PRINTF01$23.class  PRINTF01$6.class  PRINTF01.class
PRINTF01$10.class  PRINTF01$15.class  PRINTF01$2.class   PRINTF01$24.class  PRINTF01$7.class  PRINTF01.java
PRINTF01$11.class  PRINTF01$16.class  PRINTF01$20.class  PRINTF01$3.class   PRINTF01$8.class  PRINTF01.so
PRINTF01$12.class  PRINTF01$17.class  PRINTF01$21.class  PRINTF01$4.class   PRINTF01$9.class  tmp
PRINTF01$13.class  PRINTF01$18.class  PRINTF01$22.class  PRINTF01$5.class   PRINTF01.cbl
root@9c386518794a:/work#
```

この例では、PRINTF01.class の他に PRINTF01$*nn*.class（無名クラス）が 24 個作られています。

6.1. COBOL→Java 変換ツールの用途

opensource COBOL は 10 進数計算等の COBOL 言語の動作をシミュレーションするランタイム（共有ライブラリ）を呼出す C 言語を生成し C コンパイラで実行形式にします。

同様に Java 変換ツール（cobj）で生成した Java ソースは c 言語の代わりに組み込みクラスを呼出す Java の文や式を COBOL のソースに対してステップ バイ ステップで出力します。このため、生成した Java ソースには以下の問題があります。

・改修するためには COBOL シミュレーション用の組み込みクラスの機能を理解する必要がある
・COBOL の固定長で隙間なくつなげる入出力レコードは Java と相性がよくない

以上のことから、残念ながら変換ツールの用途や利用価値が見当たりません。

移行前の COBOL ソースがそのままま または軽微な改修で使えるなら COBOL のまま運用し、改修が必要なら設計から見直した方が費用対効果が上がりそうです。

7. オブジェクト指向による再実装

オブジェクト指向言語（例えば Java）とフレームワークを使い、エンティティを土台にして再設計を行えば、COBOL85 の構造化によるモジュール分割よりも強度を高め結合度を下げられる可能性があります。また要件によっては SQL クエリだけで実現できる場合もあります。

以降のサンプルは全て「5. RDB から固定長ファイルの作成方法」と同一の環境で実行しています。

7.1. SQL だけで行う多段階集計の例

COBOL のコード例であげたコントロールブレイクによる「大分類＞中分類＞…」の集計は SQL だけでも実現することができます。

＜SQL による集計例＞

COBOL マイグレーション（opensource COBOL 他）

```
-- ①科目コード毎に益（貸方）－損（借方）で集計した作業テーブルを作成する
WITH meisai AS(
    SELECT
        kamokucd
      , kamokunm
      ,(
          coalesce((SELECT sum(kingaku) * -1 FROM  t_siwake_meisai
                         WHERE kamokucd = k.kamokucd
                         AND   taisyakukb = '1'
          ),0)
        +  coalesce((SELECT sum(kingaku)      FROM  t_siwake_meisai
                         WHERE kamokucd = k.kamokucd
                         AND   taisyakukb = '2'
          ),0)
       ) AS kingaku
      , taisyakukb
    FROM   m_kamoku AS k
    WHERE  kamokucd NOT LIKE ('%0000') --大分類／中分類のコードを除外
      AND  chohyokb = '2'
)
-- ②生成した作業テーブルから科目コードの中分類に集計した L2 作業テーブルを作る
-- 作業テーブルの宣言は WITH 作業テーブル 1 AS（クエリ），作業テーブル 2 AS（クエリ），…
, L2 AS(
    SELECT kamokucd
         , kamokunm
         , (
              SELECT sum(kingaku) FROM meisai
              WHERE  SUBSTR(kamokucd, 1, 2) = SUBSTR(k.kamokucd, 1, 2)
           ) AS kingaku
         , taisyakukb
    FROM   m_kamoku AS k
    WHERE  kamokucd LIKE ('%0000')
      AND  kamokucd NOT LIKE ('%00000')  --中分類のコード以外を除外
      AND  chohyokb = '2'
)
-- ③中分類の L2 作業テーブルから大分類に集計する
SELECT kamokucd
     , kamokunm
     , (
          SELECT sum(kingaku) FROM L2
          WHERE  SUBSTR(kamokucd, 1, 1) = SUBSTR(k.kamokucd, 1, 1)
       ) AS kingaku
     , taisyakukb
FROM   m_kamoku AS k
WHERE  kamokucd LIKE ('%00000')
      AND  chohyokb = '2'

-- 明細と中分類の作業テーブルも出力して科目コード順に並べる
UNION SELECT * FROM L2
UNION SELECT * FROM meisai
ORDER BY kamokucd
```


＜出力＞

コマンド プロンプト - psql postgres postgres

```
postgres-# ORDER BY kamokucd
postgres-#
postgres-# ;
 kamokucd |              kamokunm              |   kingaku    | taisyakukb
----------+------------------------------------+--------------+------------
 800000   | 経常損益                           |    990000000 | 2
 810000   | 売上高                             |  10990000000 | 2
 810100   | 売上高                             |            0 | 2
 810101   | 売上高（売上控除高を除く）         |  10000000000 | 2
 810109   | 売上控除高                         |    -10000000 | 1
 810200   | 役務収益                           |   1000000000 | 2
 820000   | 売上原価                           | -10000000000 | 1
 820100   | 期首商品・製品たな卸高             |  -1000000000 | 1
 820200   | 当期仕入高                         |            0 | 1
 820201   | 当期商品仕入高（仕入控除高を除く） | -10000000000 | 1
 820209   | 仕入控除高                         |    100000000 | 2
 820300   | 当期製品製造原価                   |   -100000000 | 1
 820400   | 期末商品・製品たな卸高             |   1000000000 | 2
 830000   | 販売費及び一般管理費               |            0 | 1
 830100   | 人件費                             |            0 | 1
 830101   | 給料手当                           |            0 | 1
 830102   | 賞与                               |            0 | 1
 830103   | 退職金                             |            0 | 1
 830104   | 賞与引当金繰入                     |            0 | 1
 830105   | 退職給与                           |            0 | 1
 830106   | 法定福利費                         |            0 | 1
```

このデータを CSV 化して BI ツールや帳票編集ツールの入力にします。

7.2. Java で処理するための CSV ファイルの作成

（1）RDB から CSV データの抽出

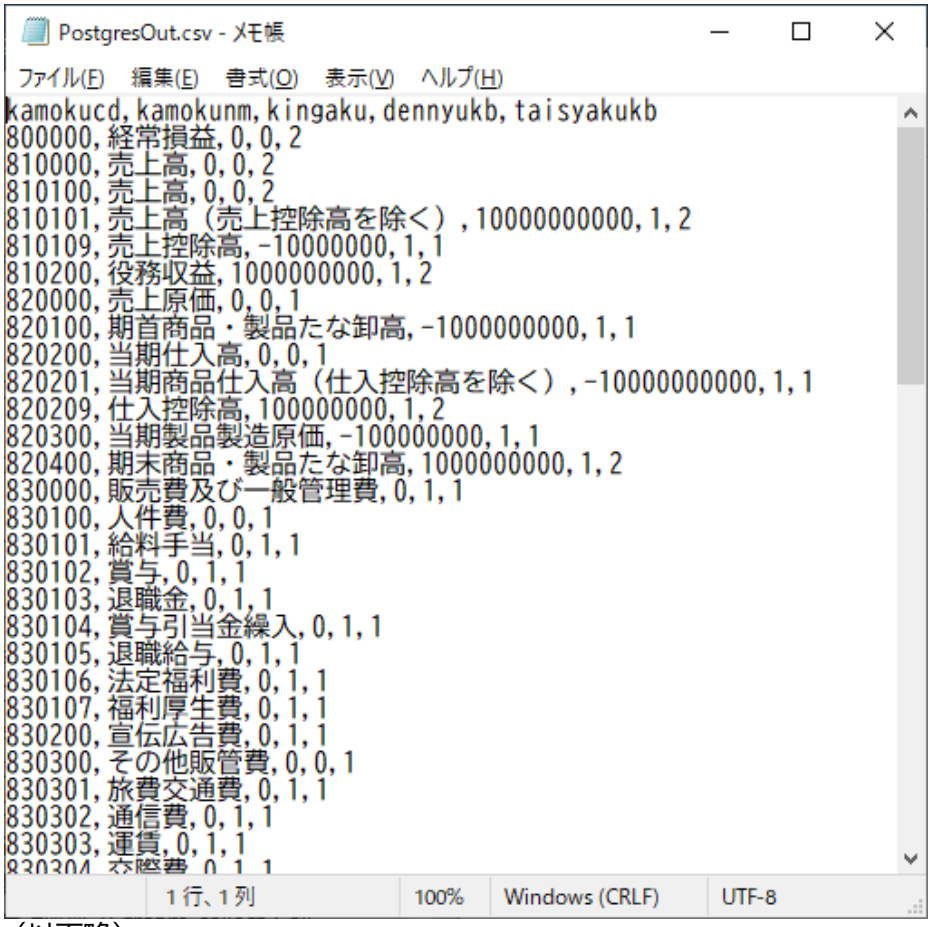
PostgreSQL で CSV 形式のデータを抽出するには¥pset format csv を行います。CSV の様式に関係する他のパラメータをしなかった場合、項目の値にカンマ「,」が含まれる場合だけ「”」で囲んで出力されます。psql で実行する全体の内容は以下になります。

```
--CSV 形式で出力する
¥pset format csv
--出力するデータの文字コードを UTF8 に変更
¥encoding UTF-8
--出力先ファイルを指定
¥out C:/COBOL/PostgresOut.csv

SELECT
    kamokucd
  , kamokunm
  ,(
      coalesce((SELECT sum(kingaku) * -1 FROM  t_siwake_meisai
                    WHERE kamokucd = k.kamokucd
                    AND   taisyakukb = '1'
      ),0)
    +  coalesce((SELECT sum(kingaku)      FROM  t_siwake_meisai
                    WHERE kamokucd = k.kamokucd
                    AND   taisyakukb = '2'
```



```
        ),0)
    ) AS kingaku
  , dennyukb
  , taisyakukb

FROM   m_kamoku AS k
WHERE  chohyokb = '2'
ORDER BY kamokucd
;

--出力先をデフォルト（ターミナル）に戻す
¥out

--出力するデータの文字コードをデフォルトに変更
reset client_encoding;
```

＜出力＞



（以下略）

（２）Java から CSV データの利用

CSV を扱うライブラリは幾つかありますが、ここでは Apache Commons CSV（オープンソース：Apache License 2.0）https://commons.apache.org/proper/commons-csv/ を使う例を挙げます。

このライブラリは CSVRecord 型に CSV レコードを読込み、ヘッダ行から取得したキー（項目名）を使い、Map（キー：バリュー）形式で項目にアクセスすることができます。複数の異なる形式の CSV に対応しており、PostgreSQL の出力にも対応しています。

COBOL マイグレーション（opensource COBOL 他）

7.3. Java による集計処理（Stream-collect-Collectors.groupingBy）

7.3.1. グルーピング

JDK 1.8 以降であれば List や Map 等のオブジェクト集合（Collection[5]）を Stream インタフェース[6]経由でデータをグルーピングすることができます。具体的には以下のようにします。

```
// 大分類の Map の中に中分類の Map を作り List で CSV データ(csvRecord)を格納する
Map<String, Map<String, List<CSVRecord>>> map = csvRecords.stream().collect(
    //1 桁目でグルーピング（コード順にするため TreeMap を使用）
    Collectors.groupingBy(l1-> l1.get("kamokucd").substring(0,1), TreeMap::new
       //1 桁目のグルーピング結果から、1-2 桁目でグルーピング
    , Collectors.groupingBy(l2 -> (l2.get("kamokucd").substring(0,2)), TreeMap::new, Collectors.toList())
    )
);
```

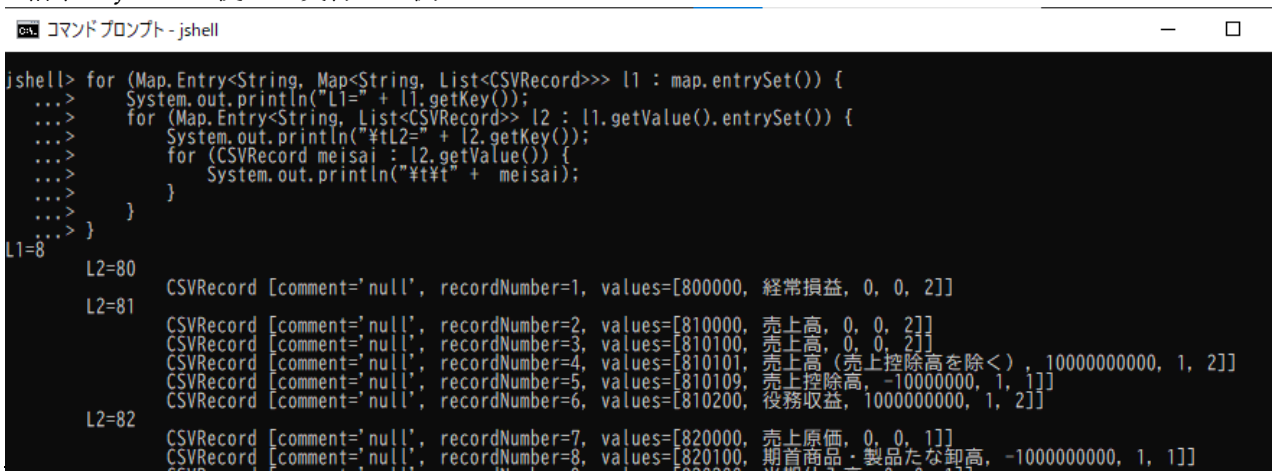
※ groupingBy は第 1 引数の条件で分類し、分類値をキーに第 2 引数で指定した型の Map を生成し、第 3 引数で指定したコレクターに渡します。第 3 引数のコレクターが goupingBy になっていることで Map の階層が作られ、最後の toList により Map のキー値で束ねた List が作られます。

7.3.2. グルーピング結果の展開

Map > Map > List の階層は Map に格納されているキーと値のセット Map.Entry をループさせて参照します。Map の階層を“map”という変数で参照する場合、具体的には以下のようにして参照します。

```
// Map（変数名 map）・Map・List の展開
for (Map.Entry<String, Map<String, List<CSVRecord>>> l1 : map.entrySet()) { //最初の階層の Map
    System.out.println("L1=" + l1.getKey());
    for (Map.Entry<String, List<CSVRecord>> l2 : l1.getValue().entrySet()) {   //2 階層目の Map
        System.out.println("¥tL2=" + l2.getKey());
        for (CSVRecord meisai : l2.getValue()) {                               //明細の List
            System.out.println("¥t¥t" +  meisai);
        }
    }
}
```

<結果> Jshell を使って実行した例



[5] Collections Framework の概要
https://docs.oracle.com/javase/jp/8/docs/technotes/guides/collections/overview.html
[6] インタフェース Stream<T>
https://docs.oracle.com/javase/jp/8/docs/api/java/util/stream/Stream.html

7.3.3. グループ毎の集計

グルーピングで使った Stream を使うと集計も簡潔な表現で行うことができます。

```
BigDecimal l2Sum = l2.getValue().stream()  //l2(Map)に格納されている List から stream を取得
        .map(r->r.get("kingaku"))          //List の要素（CSVRecord）から kingaku を取り出し
        .filter(Objects::nonNull)          //kingaku が null 以外の要素を選択
        .map(BigDecimal::new)              //選択後の kingaku から BigDecimal をインスタンス化
        .reduce(BigDecimal.ZERO, BigDecimal::add); //0 から開始して stream の BigDecimal を累算
```

※この例ではグルーピングが終わった後に集計を行っているので全入力レコードに対して 2 回走査が必要になります。制御用のコードが不要で保守性が上がりますが、大量のデータを扱う場合は方式を変更するか性能検証が必要になるかもしれません

7.3.4. コードの実行

ここまで説明したコードの全体は以下のようになり、Jshell を使うと簡単に試すことができます。

＜Jshell で実行する準備＞

① Jshell に CSV ライブラリのクラスパスを設定しておきます

/env でダウンロードした Apache Commons CSV の jar ファイルをクラスパスに指定します

```
#jshell> /env --class-path C:¥Tools¥AdoptOpenJDK¥jdk-11.0.10.9-hotspot¥commons-csv-1.10.0.jar
```

② コード中の以下のファイルパスを変更します

C:¥¥COBOL¥¥PostgresOut.csv ･･･項番 7.2 で作った CSV ファイル

C:¥¥COBOL¥¥PostgresOut.html･･･このコードが出力する損益計算書の html ファイル

※Windows 環境で実行する場合はパス区切記号の"¥"を"¥"でエスケープする必要があります

＜コードの全体＞

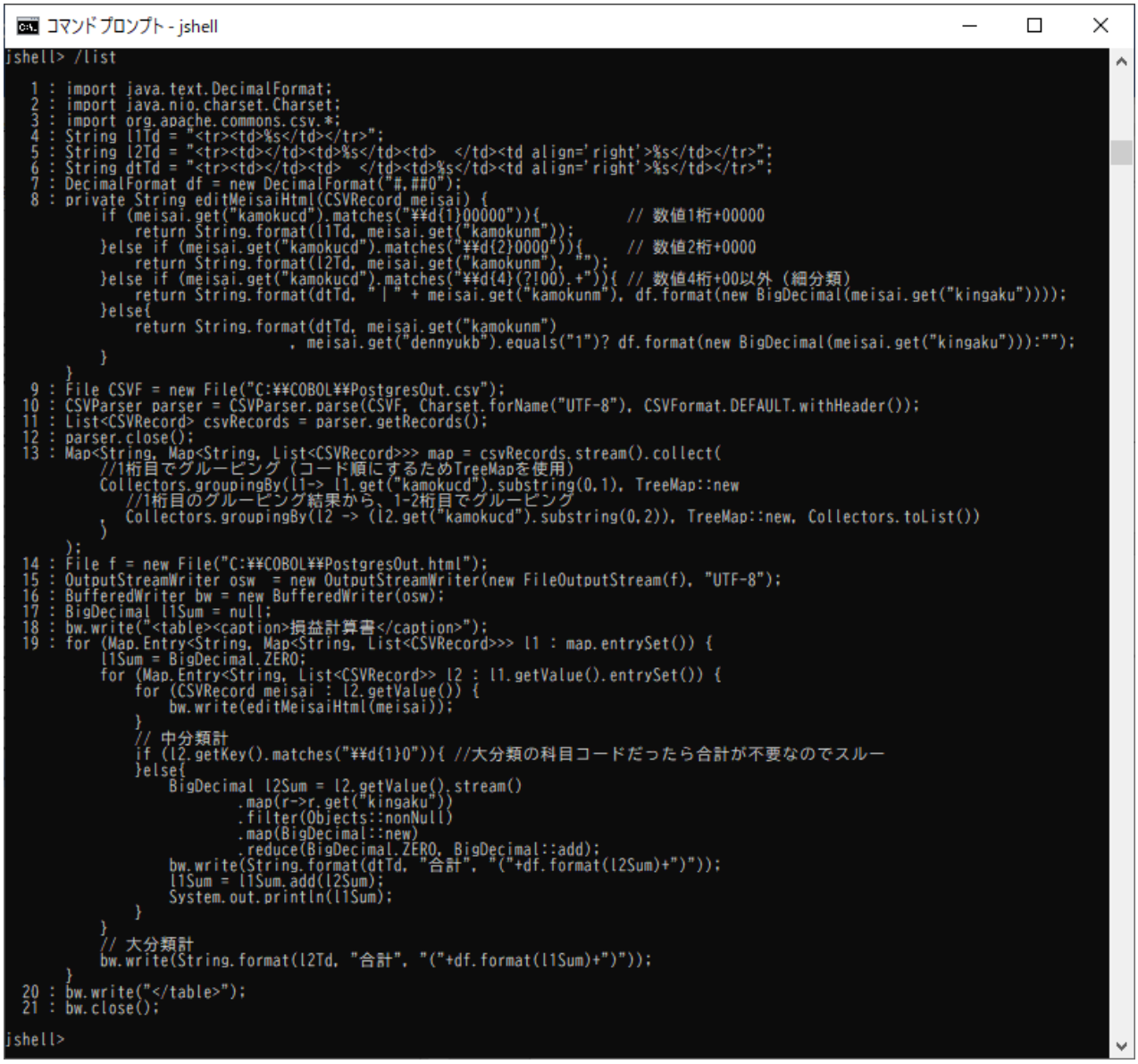
```
import java.text.DecimalFormat;
import java.nio.charset.Charset;
import org.apache.commons.csv.*;

String l1Td = "<tr><td>%s</td></tr>";
String l2Td = "<tr><td></td><td>%s</td><td>  </td><td align='right'>%s</td></tr>";
String dtTd = "<tr><td></td><td>  </td><td>%s</td><td align='right'>%s</td></tr>";
DecimalFormat df = new DecimalFormat("#,##0");

// 行単位の出力編集
private String editMeisaiHtml(CSVRecord meisai) {
    if (meisai.get("kamokucd").matches("¥¥d{1}00000")){          // 数値 1 桁+00000
        return String.format(l1Td, meisai.get("kamokunm"));
    }else if (meisai.get("kamokucd").matches("¥¥d{2}0000")){     // 数値 2 桁+0000
        return String.format(l2Td, meisai.get("kamokunm"), "");
    }else if (meisai.get("kamokucd").matches("¥¥d{4}(?!00).+")){ // 数値 4 桁+00 以外（細分類）
        return String.format(dtTd, "｜" + meisai.get("kamokunm")
                           , df.format(new BigDecimal(meisai.get("kingaku"))));
    }else{
        return String.format(dtTd, meisai.get("kamokunm")
       , meisai.get("dennyukb").equals("1")? df.format(new BigDecimal(meisai.get("kingaku"))):"");
    }
}
```


COBOL マイグレーション（opensource COBOL 他）

```
File CSVF = new File("C:¥¥COBOL¥¥PostgresOut.csv"); //PostgeSQL で出力した CSV ファイル

CSVParser parser = CSVParser.parse(CSVF, Charset.forName("UTF-8"),
CSVFormat.DEFAULT.withHeader());
List<CSVRecord> csvRecords = parser.getRecords();
parser.close();

// 大分類の Map の中に中分類の Map を作り List で格納する
Map<String, Map<String, List<CSVRecord>>> map = csvRecords.stream().collect(
    //1 桁目でグルーピング（コード順にするため TreeMap を使用）
    Collectors.groupingBy(l1-> l1.get("kamokucd").substring(0,1), TreeMap::new
       //1 桁目のグルーピング結果から、1-2 桁目でグルーピング
    ,  Collectors.groupingBy(l2 -> (l2.get("kamokucd").substring(0,2)), TreeMap::new,
Collectors.toList())
    )
);

File f = new File("C:¥¥COBOL¥¥PostgresOut.html"); //出力する損益計算書の html ファイル
OutputStreamWriter osw  = new OutputStreamWriter(new FileOutputStream(f), "UTF-8");
BufferedWriter bw = new BufferedWriter(osw);

BigDecimal l1Sum = null;

bw.write("<table><caption>損益計算書</caption>");
for (Map.Entry<String, Map<String, List<CSVRecord>>> l1 : map.entrySet()) {
    l1Sum = BigDecimal.ZERO;
    for (Map.Entry<String, List<CSVRecord>> l2 : l1.getValue().entrySet()) {
        for (CSVRecord meisai : l2.getValue()) {
            bw.write(editMeisaiHtml(meisai));
        }
        // 中分類計
        if (l2.getKey().matches("¥¥d{1}0")){ //大分類の科目コードだったら合計が不要なのでスルー
        }else{
            BigDecimal l2Sum = l2.getValue().stream()
                    .map(r->r.get("kingaku"))
                    .filter(Objects::nonNull)
                    .map(BigDecimal::new)
                    .reduce(BigDecimal.ZERO, BigDecimal::add);
            bw.write(String.format(dtTd, "合計", "("+df.format(l2Sum)+")"));
            l1Sum = l1Sum.add(l2Sum);
            System.out.println(l1Sum);
        }
    }
    // 大分類計
    bw.write(String.format(l2Td, "合計", "("+df.format(l1Sum)+")"));
}

bw.write("</table>");
bw.close();
```


COBOL マイグレーション（opensource COBOL 他）

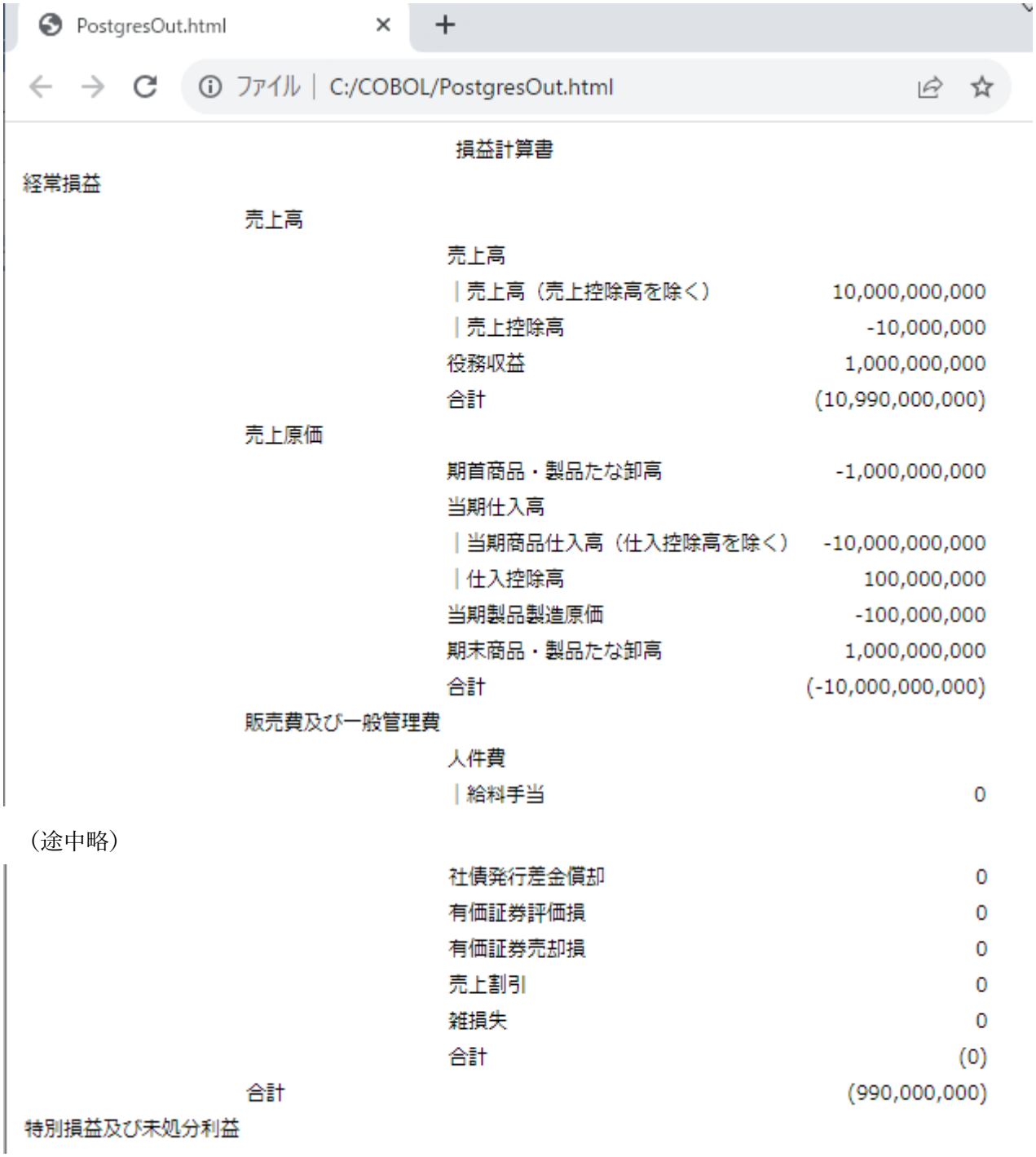
エラーがなければ /list コマンドで以下のように表示されます。



エラーがなければ次ページの内容が出力されます。

COBOL マイグレーション（opensource COBOL 他）

出力された html ファイルをブラウザで開いた内容



PostgresOut.html

ファイル | C:/COBOL/PostgresOut.html

損益計算書

経常損益			
	売上高		
		売上高	
		｜売上高（売上控除高を除く）	10,000,000,000
		｜売上控除高	-10,000,000
		役務収益	1,000,000,000
		合計	(10,990,000,000)
	売上原価		
		期首商品・製品たな卸高	-1,000,000,000
		当期仕入高	
		｜当期商品仕入高（仕入控除高を除く）	-10,000,000,000
		｜仕入控除高	100,000,000
		当期製品製造原価	-100,000,000
		期末商品・製品たな卸高	1,000,000,000
		合計	(-10,000,000,000)
	販売費及び一般管理費		
		人件費	
		｜給料手当	0

（途中略）

		社債発行差金償却	0
		有価証券評価損	0
		有価証券売却損	0
		売上割引	0
		雑損失	0
		合計	(0)
	合計		(990,000,000)
特別損益及び未処分利益			

（以下略）

COBOL マイグレーション（opensource COBOL 他）

7.3.5. マイグレーション後の強度と結合度

Java の場合は変数をなるべく内側、使う場所のブロック（メソッド、制御文の処理ブロック）で宣言して参照可能範囲を極小化します。また、共用の定数は static final を付けて宣言後の値の更新を抑止します。(COBOL85 は DATA DIVISION での記述が全て共用で更新可能)

この項で説明したソースを class 宣言の形式にすると、以下のようになります。

・メソッド間でのデータ共有はパラメータだけ

・メソッドの外で宣言されている変数（クラス変数とインスタンス変数）は final で更新が抑止されている（注意：final を付けると変数の参照先は変えられませんが、参照先の内容は更新可能です）

※ 他に Jshell が肩代わりしてくれていた機能の import 文と main メソッドの追加が必要です

```
import java.util.*;
import java.util.stream.*;
import java.text.DecimalFormat;
import java.math.BigDecimal;
import java.io.*;
import java.nio.charset.Charset;
import org.apache.commons.csv.*;

public class CSVCounting{
    static final String l1Td = "<tr><td>%s</td></tr>";
    static final String l2Td = "<tr><td></td><td>%s</td><td>  </td><td align='right'>%s</td></tr>";
    static final String dtTd = "<tr><td></td><td>  </td><td>%s</td><td align='right'>%s</td></tr>";
    final DecimalFormat df = new DecimalFormat("#,##0");

    public static void main(String[] args){ //実行の起点として main メソッド追加
        try {
            new CSVCounting().doCounting();
    } catch(IOException e) {
            e.printStackTrace();
        }
    }

    private String editMeisaiHtml(CSVRecord meisai) {

            (省略)

    }

    private void doCounting() throws IOException { //main メソッドから呼び出すためメソッド宣言追加
        File CSVF = new File("C:¥¥COBOL¥¥PostgresOut.csv");

        CSVParser parser = CSVParser.parse(CSVF, Charset.forName("UTF-8"), CSVFormat.DEFAULT.withHeader());
        List<CSVRecord> csvRecords = parser.getRecords();
        parser.close();

            (省略)

        bw.write("</table>");
        bw.close();
    }
}
```


COBOL マイグレーション（opensource COBOL 他）

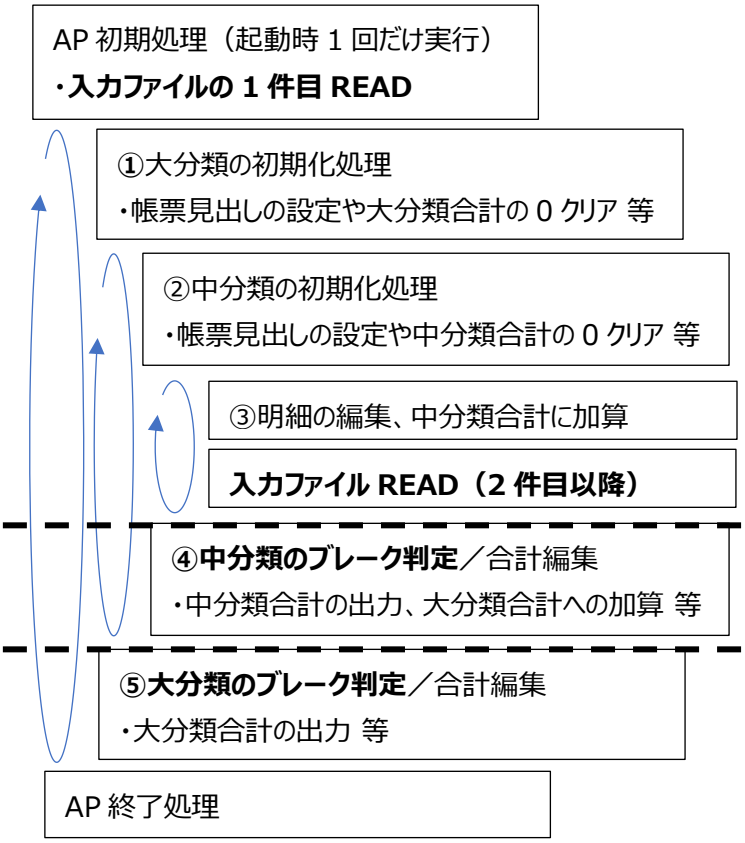
7.4. Java による集計処理（大量データを扱う場合）

Stream でグルーピングを行うには全データを JVM のヒープに格納することになります。実行時にヒープのサイズを指定しなかった場合はサーバー/クライアント、GC の方式、実メモリの搭載量等を基に決定されますが、Java 8 は最低 1M バイトのヒープを確保する[7]ので平均レコード長が 100 バイトなら 1 万件程度は格納することができます。（デフォルトで確保する最大ヒープサイズは 64 ビット、サーバーJVM、128 GB 以上の物理メモリがある場合で 32 GB です[8]）

ヒープを大量に使いたくない場合は、レコードを読みながら処理していくコントロールブレークの構造になります。

7.4.1. COBOL のコントロールブレーク

典型的な分類・集計を行うフローは下図のようになります（2 階層、ファイル open/close 等省略）。



《処理》
分類キー（ブレークキー）が変化するまで加算処理を繰り返すために、初期処理で 1 件目のレコードを先読みしておき、2 レコード目以降を読む都度キー項目を比較します

7.4.2. Java で実装する場合のフロー

他の言語と同様、Java も機能拡張の方向性として「計算操作を宣言的に記述する」に向かっています（例えば、Stream を使った filter、map、collect、reduce 他の操作）。ファイルの読み方についても Oracle 社の The Java™ Tutorials の Reading, Writing, and Creating Files[9]にはテキストファイルの

---

[7] java -Xmssize https://docs.oracle.com/javase/jp/8/docs/technotes/tools/windows/java.html
[8] 並列コレクタ … サーバー JVM のデフォルトの初期および最大ヒープ サイズ
https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html#default_heap_size
[9] Reading, Writing, and Creating Files https://docs.oracle.com/javase/tutorial/essential/io/file.html

COBOL マイグレーション（opensource COBOL 他）

読み方として、以下のように例示しています。

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

今回使った CSVParser のドキュメントでは for each ループを使った以下の例が書かれています。

```
CSVParser parser = CSVParser.parse(csvData, CSVFormat.EXCEL);
 for (CSVRecord csvRecord : parser) {
     ...
 }
```

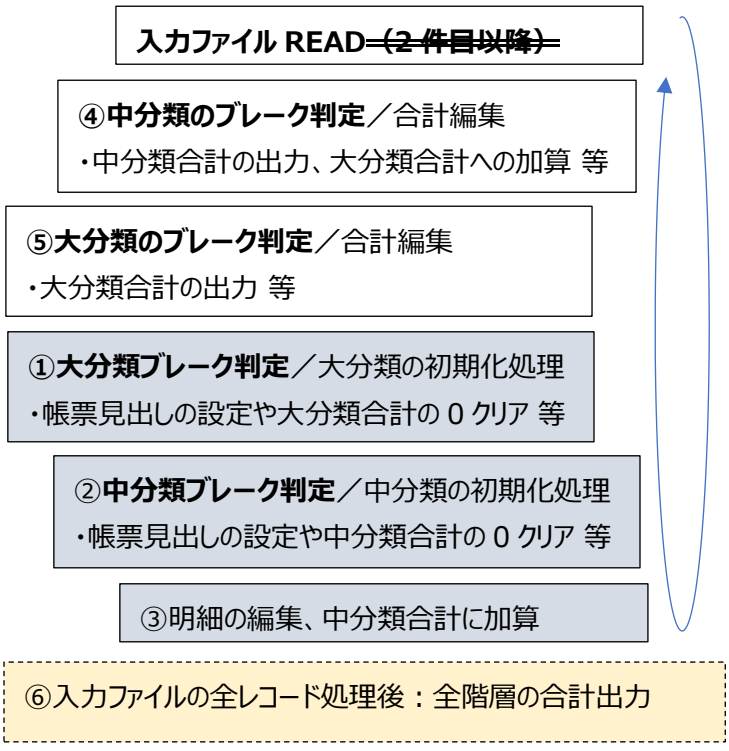
もう一例が 1 メソッドでメモリに全件読み込む以下の例です（項番 7.3 で使った方法 COBOL 仕様とコンパイラ）。

```
Reader in = new StringReader("a;b¥nc;d");
 CSVParser parser = new CSVParser(in, CSVFormat.EXCEL);
 List<CSVRecord> list = parser.getRecords();
```

1 件ずつ読込みむメソッド（CSVParser.iterator().next()）を使う例は書かれていません。

Oracle 社のドキュメントには「制御用のコードの量に比例してエラーが混入する確率も増える」（意訳）と書かれた箇所があるので、Java 言語仕様もそのように拡張されていくと考えらえます。
そういった点を踏まえて、for each ループを使ったコントロールブレークの実装例を説明します。

【Java（for each）の制御フロー】



《変更点》
・1 件目読込み（先読み）が無くなる
・各階層の初期化処理①②と明細編集／中分類合計加算③をループの最後に移す
・初期化処理①②にブレーク判定を追加
・全レコード処理後（ループ抜け後）に集計途中だった合計の出力処理を追加

7.4.3. オブジェクトの利用（階層のクラス化）

各階層の処理（合計編集、初期化処理）は同様のコードになります。COBOL85 の場合は階層毎に別の変数とそれぞれのコードが必要でしたが、Java は共通のクラスとして宣言しインスタンス化により階層毎の状態を管理させることができます。また、内部クラスとして宣言すればそれを包含している外部クラスの変数を参照することもできます。

7.4.4. for each と内部クラスを使用したコード例

Jshell で実行可能（入力の CSV ファイルが必要です）なコードの例を以下に示します。

```
import java.text.DecimalFormat;
import java.nio.charset.Charset;
import org.apache.commons.csv.*;

//html 出力編集用のフォーマット
String l1Td = "<tr><td>%s</td></tr>";
String l2Td = "<tr><td></td><td>%s</td><td>  </td><td align='right'>%s</td></tr>";
String dtTd = "<tr><td></td><td>  </td><td>%s</td><td align='right'>%s</td></tr>";
DecimalFormat df = new DecimalFormat("#,##0");

//入力レコードの種類による出力フォーマット割り当て
private String editMeisaiHtml(CSVRecord meisai) {
    if (meisai.get("kamokucd").matches("¥¥d{1}00000")){          // 数値 1 桁+00000
        return String.format(l1Td, meisai.get("kamokunm"));
    }else if (meisai.get("kamokucd").matches("¥¥d{2}0000")){     // 数値 2 桁+0000
        return String.format(l2Td, meisai.get("kamokunm"), "");
    }else if (meisai.get("kamokucd").matches("¥¥d{4}(?!00).+")){ // 数値 4 桁+00 以外（細分類）
        return String.format(dtTd, " | " + meisai.get("kamokunm")
            , df.format(new BigDecimal(meisai.get("kingaku"))));
    }else{
        return String.format(dtTd, meisai.get("kamokunm")
            , meisai.get("dennyukb").equals("1")? df.format(new BigDecimal(meisai.get("kingaku"))):"");
    }
}

//コントロールブレークキー毎の情報を管理する（内部）クラス
private class KamokuKei{
    KamokuKei upper; //一階層上のブレークキー管理オブジェクト
    String svCd;     //このオブジェクトが集計しているブレークキー
    int    len;      //ブレークキーの長さ
    String format;   //出力に使うフォーマット文字列
    BigDecimal kei = BigDecimal.ZERO;//累計値

    KamokuKei(KamokuKei upper, int len, String format){
        this.upper = upper;
        this.len = len;
        this.format = format;
        this.svCd = "000000".substring(0, len);
    }
    //最新の読込みレコードに対する処理-①、②
    void header(String kamokuCd)  throws java.io.IOException{
        if (!svCd.equals(kamokuCd.substring(0, len))) {
            if (upper != null) upper.header(kamokuCd);//上位計を先に処理
            svCd = kamokuCd.substring(0, len);
            kei = BigDecimal.ZERO;
        }
```



```
47          }
48          //ブレークしたコード（svCd に保存していたコード）に対する処理-④、⑤
49          void footer(String kamokuCd, String kingaku) throws java.io.IOException{
50              if (svCd.equals(kamokuCd.substring(0, len))) {
51                  kei = kei.add(new BigDecimal(kingaku)); //③
52              }else{ //キーブレーク！
53                  flush();
54                  if (upper != null) upper.footer(kamokuCd, kei.toString());//上位計に足し込み
55              }
56          }
57          //ブレーク時と EOF の時の出力（但し、初期値のままか上位の集計科目…x00000 だったら出力しない）
58          void flush() throws java.io.IOException{
59              if (!svCd.substring(len-1, len).matches("0+")){
60                  //bw と df は外部クラスに定義されているものを参照
61                  bw.write(String.format(format, "合計", "("+df.format(kei)+")"));
62              }
63          }
64      }
65
66      //ファイルの入力制御と主処理
67      File CSVF = new File("C:¥¥COBOL¥¥PostgresOut.csv");
68      CSVParser parser = CSVParser.parse(CSVF, Charset.forName("UTF-8"), CSVFormat.DEFAULT.withHeader());
69
70      File f = new File("C:¥¥COBOL¥¥PostgresOut.html");
71      OutputStreamWriter osw  = new OutputStreamWriter(new FileOutputStream(f), "UTF-8");
72      BufferedWriter bw = new BufferedWriter(osw);
73      bw.write("<table><caption>損益計算書</caption>");
74
75      // 大分類の状態管理オブジェクト
76      KamokuKei kkL1 = new KamokuKei(null, 1, l2Td);
77      // 中分類の状態管理オブジェクト
78      KamokuKei kkL2 = new KamokuKei(kkL1, 2, dtTd);
79      // CSV ファイルを 1 レコードずつ読み処理を行う。ブレーク判定とブレーク処理は KamokuKei クラスで行う
80      for (CSVRecord meisai : parser) { //入力レコードを 1 件ずつループ処理
81          kkL2.footer(meisai.get("kamokucd"), meisai.get("kingaku")); //④,⑤
82          kkL2.header(meisai.get("kamokucd"));                        //①,②
83          bw.write(editMeisaiHtml(meisai));                           //③
84      }
85      kkL2.footer("~~~~~~", "0"); //ascii 文字コード（utf-8 が前提）の一番大きな値の"~"で強制全ブレーク⑥
86
87      bw.write("</table>");
88      bw.close();
89      parser.close();
```

26～64 行：KamokuKei クラス-各階層のブレークキー毎の情報を管理する内部クラス

79～85 行：コントロールブレークの制御処理

＜内部クラス:KamokuKei＞

・一階層上のオブジェクトへの参照を持っていてブレークが発生したときに上位階層の同一メソッドを呼び出すことでブレークを連鎖させている（不要な判定処理を避けるため）

・header メソッドは先に上位階層の header メソッドを呼出して大分類＞中分類の順に処理を行い、footer メソッドは逆に中分類＞大分類の順に処理が行われるようにしている

COBOL マイグレーション（opensource COBOL 他）

【少し簡略化したコード】

この損益計算書を作成するコードは合計編集と次のサイクルの初期化処理が同一の実行条件で、挟まる処理はありません。このため初期化の処理内容を少し変えて 51、52 行目に移し header メソッドを省略できます（大、中分類の初期化順が変わりますがこのアプリケーションの処理結果に影響はありません）。

```
27      private class KamokuKei{
28          KamokuKei upper; //一階層上のブレークキー管理オブジェクト
29          String svCd;     //このオブジェクトが集計しているブレークキー
30          int    len;      //ブレークキーの長さ
31          String format;   //出力に使うフォーマット文字列
32          BigDecimal kei = BigDecimal.ZERO;//累計値
33
34          KamokuKei(KamokuKei upper, int len, String format){
35              this.upper = upper;
36              this.len = len;
37              this.format = format;
38              this.svCd = "000000".substring(0, len);
39          }
40          //最新の読込みレコードに対する処理
41          void header(String kamokuCd){
42              //このアプリでは、ここで実行すべき処理はありません
43          }
44          //ブレークしたコード（svCd に保存していたコード）に対する処理
45          void footer(String kamokuCd, String kingaku) throws java.io.IOException{
46              if (svCd.equals(kamokuCd.substring(0, len))) {
47                  kei = kei.add(new BigDecimal(kingaku)); //③
48              }else{ //キーブレーク！
49                  flush();  //④、⑤
50                  if (upper != null) upper.footer(kamokuCd, kei.toString());//上位計に足し込み //④、⑤
51                  svCd = kamokuCd.substring(0, len);  //①、②
52                  kei = new BigDecimal(kingaku);      //①、②
53              }
54          }
55          //ブレーク時と EOF の時の出力（但し、初期値のままか上位の集計科目…x00000 だったら出力しない）
56          void flush() throws java.io.IOException{
57              if (!svCd.substring(len-1, len).matches("0+")){
58                  //bw と df は外部クラスに定義されているものを参照
59                  bw.write(String.format(format, "合計", "("+df.format(kei)+")"));
60              }
61          }
62      }
```

コントロールブレークの制御処理は、以下のようになります（82～83 行目）。

```
75      // 大分類の状態管理オブジェクト
76      KamokuKei kkL1 = new KamokuKei(null, 1, l2Td);
77      // 中分類の状態管理オブジェクト
78      KamokuKei kkL2 = new KamokuKei(kkL1, 2, dtTd);
79      // CSV ファイルを 1 レコードずつ読み処理を行う。ブレーク判定とブレーク処理は KamokuKei クラスで行う
80      for (CSVRecord meisai : parser) { //入力レコードを 1 件ずつループ処理
81          kkL2.footer(meisai.get("kamokucd"), meisai.get("kingaku")); //④,⑤,①,②
82          bw.write(editMeisaiHtml(meisai));                           //③
83      }
84      kkL2.footer("~~~~~~", "0"); //ascii 文字コード（utf-8 が前提）の一番大きな値の"~"で強制全ブレーク⑥
```

以上

