MONTSUQI 解説書
Ver 0.0

日本医師会綜合政策研究機構
生越昌己

2004 年 3 月 8 日

目次

第Ⅰ部

機能編

第 1 章

概要

1.1 特徴

MONTSUQI とは、UNIX 系 OS 上でオンラインシステムを実現する、オンライントランザクションモニターである。**MONTSUQI** は以下のことを狙って開発したものである。

1. **負荷の制御を容易にする**
2. **プログラミングを簡単にする**
3. **信頼性を向上させる**
4. **処理性能を向上させる**

以下に順を追って説明する。

1. 負荷状態の制御を容易にする

業務システムのオンラインでは確実な動作が要求される。すなわち、投入された処理は確実に処理をされて、その結果が返らなければならない。受け付けた処理は確実に実行されることが期待され、受け付けたからには結果を返さなければならない。しかし、通常のオープン系アプリケーション実行環境においては、処理要求が増えるとそのまま負荷が増え続け、アプリケーションの処理が重くなるだけではなく、ついには制御不能に陥る。

MONTSUQI では 1 つのアプリケーションプロセスが複数の端末をサービスすることにより、処理要求数の増加がそのままシステムの負荷にならないにした。また、キューによる実行制御を行うことにより、処理が大量に投入されても一定以上の負荷にはならないような制御を可能にした。これにより、処理要求数が増えても、反応時間が遅くなるだけで負荷はほとんど増えない。各種パラメータを調整してやれば計算機の能力に合わせて処理量を増やすことができ、負荷状態を制御することが可能になる。

2. プログラミングを簡単にする

セションやトランザクションの管理をシステムで自動的に行うことにより、アプリケーションプログラマをセションやトランザクションを管理するシステム制御処理のプログラミングから解放した。これによりアプリケーションプログラマはシステム制御のプログラムにわずらわされることなく、アプリケーションの開発に専念することができる。

また、アプリケーションが実行される時の環境であるデータベースやプレゼンテーションの仮想化を行い、特定の環境に依存しないようにした。これによりデータベースやプレゼンテーションが変わっても、アプリケーションが影響を受けなくなる。

MONTSUQI ではアプリケーション記述言語を限定していない。COBOL, C だけではなく他の言語でのアプリケーション記述が可能である。このため、目的とするアプリケーションを最も書きやすい言語の選択が可能である。

3. 信頼性を向上させる

MONTSUQI は、デッドロックやアプリケーションサーバの異常等といった何らかの理由によりトランザクション処理に失敗すると、ロールバック後に再試行を行う。この時アプリケーションサーバが停止していればトランザクションは保留され、アプリケーションサーバが再起動された後に再試行される。これらのことにより、確実なトランザクション処理が可能となる。

MONTSUQI はデータベースおよびアプリケーションサーバを必要に応じて多重化することが可能である。このことにより、万一データベースを収容しているハードウェアが故障したり、何らかの理由でアプリケーションサーバが停止した場合でも、データの保全および処理の続行が可能である。

4. 処理性能を向上させる

MONTSUQI のアプリケーションサーバは、必要に応じて並列処理をさせたり、別のホスト上に分散させることが可能である。このことはの信頼性の向上と共に、処理性能を向上させる。

またこの他にも**拡張性**も考えられており、通信プロトコル、アプリケーション記述言語、データベース等、モニタの設計の制約となるものとのインターフェイスは、モジュール化されている。このため、モニタ自身の拡張は容易である

第 2 章

MONTSUQI の理解に必要な概念

2.1 用語

2.1.1 概略

MONTSUQI には以下のような固有の概念がある。

- **イベント**
  クライアントからの処理要求の単位。トランザクション発生の契機となる
- **LD**
  業務から見たアプリケーションのことで、トランザクションデータの論理的な宛先 (Logical Dentination) となる
- **SPA**
  セションに関する全ての変数を保持する領域で、全ての業務に共通な領域であるリンケージ領域 (LINKAREA) と、同一 LD 内でのみ有効な SPAAREA とからなる。
- **多重化グループ**
  データの依存性のある LD を組にしたもの。トランザクションは、この組の中では直列に処理されるが、この組同士は並列に処理される
- **データベースグループ**
  同じデータベースに保存されるテーブルを組にしたもの。データベースのアクセスや保全はこの単位で行われる
- **BD**
  バッチプログラムの実行環境について記述した定義体
- **ハンドラ**
  「実際の処理」を行うためのクラスを実現するモジュール。現在のところ、アプリケーションの言語インターフェイスを司る「言語ハンドラ」と、データベースマネージャとのインターフェイスを司る「データベースハンドラ」が存在する
- **ポート**
  通信に使われる「口」のことである。UNIX 上で使われるプロセス間通信の方法は様々あるが、**MONTSUQI** 中ではそれらを統一的に表現するために、ポートという概念で統合している

2.1.2　イベント

利用者の操作は、アプリケーションへの処理の要求となる。イベントとは、アプリケーションが処理を行う契機となる事象とそれに付随するデータからなる。利用者はイベントを発生させて、アプリケーションに処理要求を出す。

イベントは以下の要素からなる。

1. **ウィンドウ名**
2. **ウィジェット名**
3. **イベント名**
4. **状態**
5. **PL データ**

1. ウィンドウ名

イベントの発生したウィンドウの名前である。ウィンドウという概念のない PL サービスでは、**画面**や**パネル**といったものに対応される。**MONTSUQI** 内部では、トランザクションパケットをアプリケーションに振り分ける時のに使われる。

2. ウィジェット名

イベントの発生したウィジェットの名前である。一般にウィジェットとは GUI 部品を指すが、GUI を持たない PL サービスの場合は可視化されている何らかの実体や**文脈**のようなものに対応される。**MONTSUQI** 内部では特別な意味は持たず、アプリケーションに渡されるだけの情報である。

3. イベント名

発生したイベントを識別するための名前である。これはアプリケーションがどのような処理をするべきか識別する最も重要な情報である。**MONTSUQI** 内部では特別な意味は持たず、アプリケーションに渡されるだけの情報である。

4. 状態

ここで状態とは、アプリケーションに制御が渡った時の状態で、アプリケーションに直接関係があるのは、

- **画面遷移の結果他のプログラムから制御が渡った**
- **画面出力後にイベントが発生した**

である。イベント自体は画面出力後にしか発生しない。

5.PL データ

クライアントから送られて来る、クライアントが保持しているデータである。処理効率の観点から、多くの PL サーバと PL クライアントの間は差分の通信がされるため、PL サーバが保持しているデータで補完されてからバックエンドに送信される。

2.1.3 LD

オンラインの業務画面は、全てが独立したものではなく、いくつかの画面が集まって業務が構成され、その業務の集合体が全体のシステムとなっている。

この業務を構成をする単位の中では、いくつかの画面 (を駆動するモジュール) 間では、かなり密にデータの共有が行われる。そこで、そのような画面のまとまりを単位としてシステム設計を行うと、設計の効率が良くなる。また、データが密に共有されているため、プログラムの実行単位としても効率がいい。このような単位を LD と呼ぶ。

本システムでは、この LD がアプリケーションサーバ (APplication Server, APS) の実行単位となる。つまり、論理的には LD とアプリケーションサーバは対応する。

同じ LD 内では次に述べるアプリケーション共通領域を共有する。

2.1.4 SPA

一般に 1 つの業務は 1 回のトランザクションでは終結せず、いくつかのトランザクションを組み合わせて行われるものである。このトランザクションの組み合わせをセションと呼ぶ。トランザクション間でのデータをひきつぎ、セションを維持するための情報を持っている領域が、この SPA(Scratch Pad Area) である。

アプリケーションは、この領域にデータを保存することにより、トランザクションにまたがってデータを保持することが出来る。逆の言い方をすれば、この領域に保存されないデータは、トランザクション終結後に消失する。

SPA 領域は、同一 LD 内でのみ有効な領域と、LD をまたがって保持される領域とがある。前者をアプリケーション共通領域と呼び、後者をセション共通領域または *linkarea* と呼ぶ。

アプリケーション共通領域の内容は、別の LD に制御を移した時点で消失する。セション共通領域については、セションが終結するまで保持される。

アプリケーション共通領域、セション共通領域共に、寿命の範囲でグローバルなデータ領域として参照される。そのため、アプリケーションモジュール間のデータ交換領域として使用可能である。このことは言語ハンドラの種類に依らない。

2.1.5 並列化

トランザクションは、発生する順序に従い順次処理が行われるのが基本である。しかし、データの相互依存性のないトランザクションについては、並列に実行しても問題は起きないし、そうすることによって実行効率は上がる。本システムでは、4 つのレベルを設けて、並列化の指示を行う。

1. **no**
   全く並列化を行わないものである。端末から投入されたトランザクションは、全て直列に処理される。すなわち、1 つの端末からのトランザクションが終結後、次のトランザクションが処理させる。この戦略は排他制御上の問題は一切発生しないので、デッドロック等資源管理上の問題が起きない。その代わり、処理が完全に直列なので、スループットは落ちる
2. **id**
   データの依存関係のある LD をまとめてグループ化し、同じグループのトランザクションは直列に処理し、異なるグループのトランザクションは並列に処理を行うようにする戦略である。この戦略は、デー

タ更新の安全性を保ちつつ、処理効率を上げることが可能になる。
このデータの依存関係のある LD をまとめたものを多重化グループと呼び、ワークフローコントローラが実行制御を行う際のヒントとして教えてやる

3. **ld**
全てのアプリケーションサーバを並列に稼働させる戦略である。この戦略は LD 間のデータ依存性がない時に設定可能である。排他制御はデータベースマネージャに全て依存するので、並列に実行されたトランザクション間でデッドロックが起きる危険性がある。その代わり、全てのアプリケーションサーバが完全に並列に動く。
4. **aps**
同一 LD を処理するアプリケーションサーバを、さらに多重に稼働させる戦略である。同じ LD でも操作する端末が異なれば、異なるデータベースレコードを操作する確率が高いため、同じ LD であっても並列に処理をすることは可能である。これを積極的に利用して、より並列性を高めることが可能である。この戦略は同じ業務 (LD) を操作する端末が多い時に効果的である。全ての戦略の中で最もスループットが高い

2.1.6 データベースグループ

業務で使うデータテーブルは、単一のデータベースに保存されるとは限らず、いくつかのデータベースに分散して置かれることも考えられる。また、ログデータの発生のしかたによってログファイルを異るファイルに出したいこともある。

このように、データテーブルは、その目的や運用の違いによって、いくつかのグループに分ける方が、運用の都合上好ましいことが少なくない。そこで、本システムでは、データテーブルをグループ化して運用が出来るように、データベースグループという概念を持っている。

同じデータベースグループのテーブルは、まとめて同じデータベースマネージャにリクエストが出される。また、同じデータベースグループのログデータは、同じ出口 (リダイレクタ) に出力される。

2.1.7 BD

バッチプログラムの実行環境について記述した定義体 (Batch Description) である。

本来バッチプログラムは、どのように書かれていても実行可能なはずである。しかし、オンラインシステムの実行メカニズムと矛盾しないように、また同じ API でデータベースが操作を行い、またデータ保全が行われるためには、オンラインプログラムの実行と同じ管理を行う必要がある。

また、バッチプログラムも、様々な言語によって記述出来るため、そのバッチプログラムが、どのような言語で書かれているかを、バッチ用データベースサーバに教えてやる必要がある。

このような理由から、バッチプログラムもオンラインプログラム同様、**MONTSUQI** の配下で動かせるようになっている。

2.1.8 ハンドラ

MONTSUQI のコアモジュールは、**MONTSUQI** 内でのデータの操作だけを行い、実際にアプリケーションを実行したりデータベースをアクセスしたりという処理は、全て「ハンドラ」と呼ばれるモジュールにまかせている。ハンドラモジュールは読み込まれるとクラスという形で利用準備が可能になる。

言語処理系毎に呼び出しや実行のためのシーケンスやインタフェイスは異なるが、それらは「言語ハンドラ」と呼ばれるモジュールが、実際の処理を行っている。本システムを新しい言語に対応させるには、この言語ハンドラを書いてやることによって実現される。

データベースマネージャとのインターフェイスも、データベースマネージャ毎に異るが、これは「データベースハンドラ」が実際の処理を行っている。新しいデータベースマネージャに対応することは、このデータベースハンドラを書いてやることによって実現される。

2.1.9 ポート

MONTSUQI では、プロセス間通信のための口を「ポート」と呼ぶ。これは以下の形式からなるデータ構造である。

1. **ホスト名:TCP ポート名 (番号)**
2. **ファイル名:パーミッション**

1 は TCP であり、2 は UNIX ドメインである。

1. ホスト名:ポート番号

TCP を使った通信を行う指定である。TCP は IP 上のプロトコルであるため、ホストを超えての通信が可能である。この通信には、

- ホストを超えて処理が可能で、配置の自由度が高い
- 通信内容に対して盗聴や不正侵入についての対策を別途考える必要がある
- 通信のオーバーヘッドについてを考える必要があることもある

と言った特性がある。**MONTSUQI** では古くから使われている通信方法であり、実績もある。

ホスト名を省略した場合は、暗黙のうちに 'localhost' が指定されたものとみなされる。

TCP ポート名 (番号) が省略された場合は、システムの持っているデフォルト値が使われ、この値はポートの用途による。

両方が省略された場合は、ポート自体の指定がないということになるため、多くの場合エラーになるか、システムの持っているデフォルト値が採用される。

2. ファイル名:パーミッション

UNIX ドメインを使った通信を行う指定である。この通信は同一ホスト上でのみ有効である。この通信には、

- 同一ホスト内での処理のも可能で配置の自由度が低い
- 外部からの不正侵入や盗聴がありえないので安全性が高い
- 通信自体が軽い

と言った特性がある。

ファイル名の省略はできない。このファイル名は UNIX ドメインの接続のために使われるファイル名なので、実在のファイルがあってはならない。

パーミッションを指定すると、そのポートに接続可能な利用者を制限することができる。この指定は UNIX のファイルの指定と同じ 8 進数による指定となる。省略した場合は 666 となり、全ての利用者に解放される。

TCP と UNIX ドメインの指定は形式が異なるため、解釈時に自動的に判定される。しかし、修飾を持たない現用ディレクトリ上のファイル名を指定し、かつパーミッションの指定をしない UNIX ドメインの場合、ドメイン名で修飾しないホスト名の指定と区別がつかないために、TCP ポートと間違える。これを避けるために、明示的にファイル名の前に‘#’を前置してやると、強制的に UNIX ドメインと解釈させることができる。

第 3 章

構造

3.1 プロセス構成の概要

図 3.1 に **MONTSUQI** のプロセス構成を示す。以下のプロセスによって構成される。

1. **PL クライアント**
   利用者側にあるプロセスで、利用者からのリクエストを受けつけ結果を出力する。通常は glclient という GUI のクライアントが使われている。glclient には X 版の他に Windows 版と Java 版がある。また、glclient の他に web インターフェイスを実現する mon がある。
2. **PL サーバ**
   PL クライアントと通信を行い、データのやりとりを行う。現在のところ端末とのセションの維持は PL サーバが行っている。PL クライアント毎に対応した PL サーバを必要とする。
3. **ワークフローコントローラ**
   セションとトランザクションの管理を行い、プレゼンテーションサーバとアプリケーションサーバ間のルーティングを行う。
4. **アプリケーションサーバ**
   アプリケーションの実行を行う
5. **データベースレイヤ**
   各種データベースマネージャとのインターフェイス、及びデータ保全機能の実現を行う

以下にこれらの概要について説明を行う。

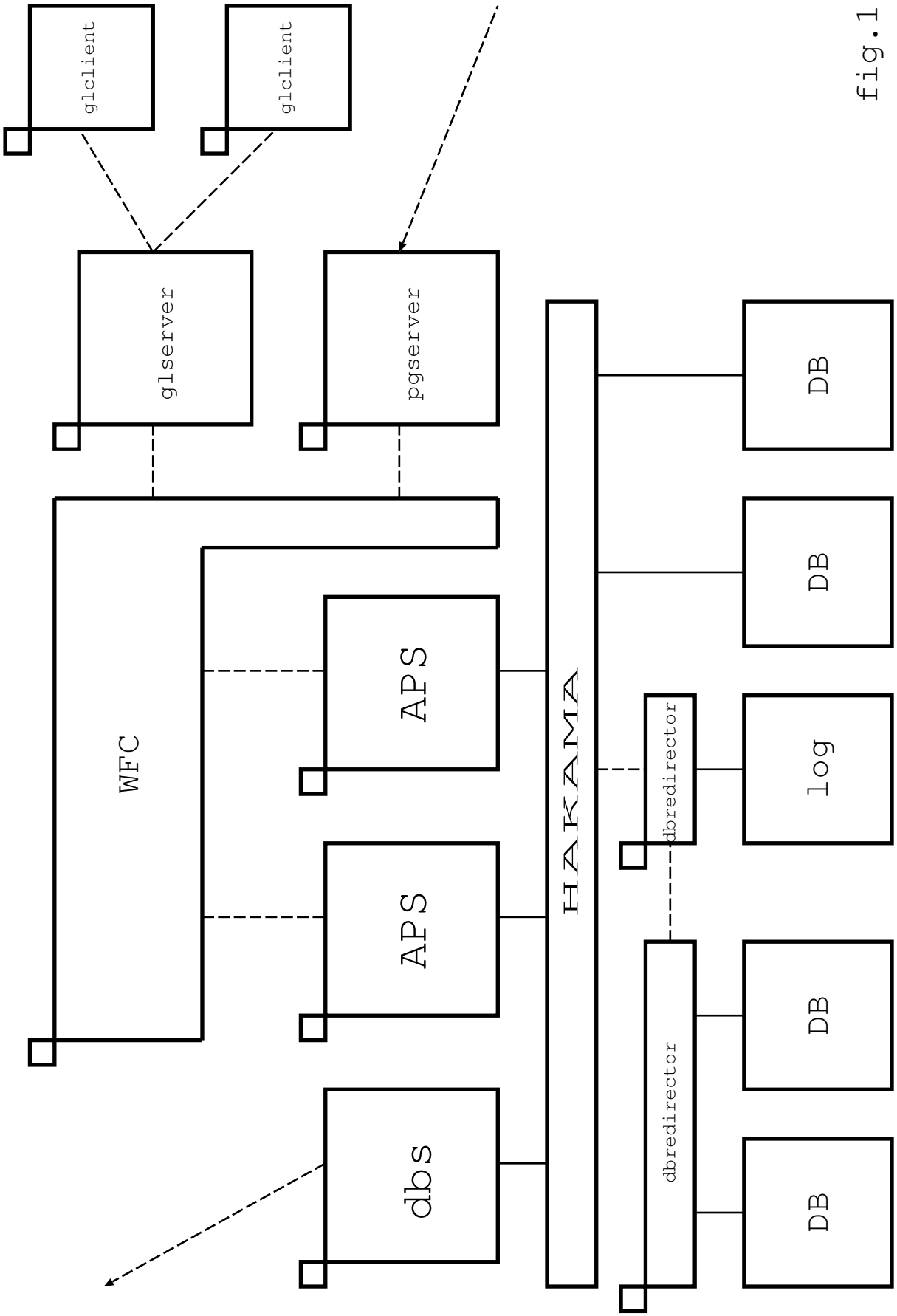

図 3.1　MONTSUQI のプロセス構成

3.2 PL サービス

PL サービスとは、Presentation Layer service という意味である。Presentation Layer とは日本語では「表現層」と呼ばれ、通信データを「表現」するための層である。**MONTSUQI** では、利用者がアプリケーションを操作するための、主に表示を行う部分を指す言葉として、Presentetion Layer を用いる。ただし、既に述べたように **MONTSUQI** ではアプリケーションは特定の表現方法に依存しないため、「表示」を持たないもの、たとえば他のプログラムや音声応答といったものも、このサービスに含まれる。利用者から見て PL サービスよりも「向こう」にあるプロセス群をバックエンド、PL サービスのことをフロントエンドという呼び方もある。

MONTSUQI の PL サービスは **MONTSUQI** のアーキテクチャ的にクライアントサーバ構成である。つまり、利用者側でクライアントプロセスが動き、それに対応したサーバプロセスがサーバ上で動く。クライアントプロセスのことを **PL クライアント**、サーバプロセスのことを **PL サーバ**と呼ぶ。サーバプロセスはさらにバックエンドと通信をしてアプリケーションを実行する。この間でやりとりされるデータを **PL データ**と呼ぶ。

MONTSUQI は様々なクライアントに対応するため、**MONTSUQI** のコアプロセスであるワークフローコントローラは、直接クライアントとの通信を行わない。全ての通信制御はプレゼンテーションサーバが行い、**MONTSUQI** 自身とは独立したものとして存在する。

表現方法によるデータや制御の違いは、全て PL サービスによって吸収され、バックエンドでは PL サービスが具体的にどのような動作をするかは、全く意識をしない。この独立性により、他の表現方法に対応するためには、別の PL サービスシステムを記述することにより実現出来る。正しく記述されたアプリケーションは、異る PL サービスの下であっても正しく動くはずである。

3.2.1 glserver/glclient

glclient は GUI を実現する端末ソフトウェアである。画面レイアウト情報と埋め込みデータを glserver との通信によって得て表示を行い、利用者の操作情報と画面データを glserver に送り返す。現在のところ、glclient 自身はこれ以上のインテリジェンスを持たない一種の**ダム端末**である。

現在のところ、glclient で使用する画面レイアウト情報は、glade によって生成される XML ファイルである。

3.2.2 htserver/mon

mon はウェブインターフェイスを実現するために CGI として起動されるプログラムである。画面レイアウト情報を読み込み、その情報を元に埋め込むデータを htserver との通信によって得て表示を行い、利用者の操作情報と画面データを htserver に送り返す。セションの維持はセション ID によって行われる。セション ID のやりとりには、hidden タグによる方法と cookie による方法とが利用できるため、cookie 機能を持たない i モード端末のようなものであっても、セションの保持ができる。

画面レイアウト情報 `HTC` という拡張子を持つテキストファイルで、3.2.1glclient で使う XML を元に `htcgen` というスクリプトによって生成することができる。これは任意に編集することが可能である。

htserver は mon とは独立に使うことができる。このため、eRuby や PHP からも利用が可能である。

3.2.3 pgserver

pgserver は **MONTSUQI** 外部のアプリケーションから **MONTSUQI** の機能を使うために起動されるプログラムである。外部のプログラムは pgserver と通信することにより、**MONTSUQI** の画面操作をエミュレートすることができる。

3.2.4 PL サービスアプリケーション

PL サービスには、**PL サービスアプリケーション (Presentation Layser service application)** というものが存在し、C による API でプログラムをすることができる。これは **MONTSUQI** のバックエンドに依存しないもので、バックエンドとは独立に動作させることが可能である。この API は既に表現方法の違いを吸収しているため、**MONTSUQI** バックエンドと独立に表現方法に依存しないアプリケーションを作成することが可能である。**MONTSUQI** バックエンドとの連携は、PL サービスアプリケーションとして実現されている。

PL サービスアプリケーションは、**MONTSUQI** バックエンド配下のアプリケーションとは以下の点で異なる。

- **接続される端末毎に独立した空間で動く**
- **トランザクション管理等のサービス等は提供されない**
- **表現層以外の API を持たない**

このため、高度な業務システムには向かないが、オンラインのユーティリティを抽象化表現層の下で作るといったことに利用可能である。

3.3 ワークフローコントローラ

MONTSUQI のコアプロセスである。主な処理は、

- **セションとトランザクションの管理**
- **プレゼンテーションサービスとの通信**
- **アプリケーションサーバとの通信**
- **トランザクションキューの管理**

である。直接アプリケーションの実行は行わないが、アプリケーションの実行を行うためのトランザクションの管理を行うためのプロセスである。

3.3.1 プロセス構造

ワークフローコントローラは、多くのスレッドからなるプロセスである。それぞれのスレッドは以下のようなグループに分類される。

1. **プレゼンテーションサービスとの通信 (termthread)**
2. **アプリケーションサーバとの通信 (mqthread)**
3. **全体のキューのルーティング (corethread)**
4. **制御プログラムとの通信 (controlthread)**

1. プレゼンテーションサービスとの通信 (termthread)

プレゼンテーションサーバに接続される端末毎に起動されるスレッドである。プレゼンテーションサーバとの通信電文とトランザクションパケットの変換を行うと共に、BLOB のスプーリングを行う。

2. アプリケーションサーバとの通信 (mqthread)

アプリケーションサーバが接続される毎に起動されるスレッドである。アプリケーションサーバとの通信電文とトランザクションパケットの変換を行う。

3. 全体のキューのルーティング (corethread)

キューのルーティングは現在のところ 1 つのスレッドで行われている。ここでは 1 で作られたトランザクションパケットを、LD 毎に振り分けている。このスレッドによりキューの整列も行われる。

4. 制御プログラムとの通信 (controlthread)

制御プログラムが接続される毎に起動されるスレッドであるが、通常は存在しないか、しても 1 つだけである。このスレッドはワークフローコントローラの動作を規定する変数やフラグの参照および更新を行う。

3.3.2 トランザクションパケット

トランザクションの内容はパケットとして各キューに接続される。このパケットは大略以下のような項目から構成されている。

1. **プレゼンテーション情報**

2. **イベント情報**
3. **セション変数**
4. **管理情報**

1. プレゼンテーション情報

プレゼンテーションサービスとやりとりする情報である。ワークフローコントローラでは、保持されるだけで内容については関知しない。

2. イベント情報

処理の契機となったイベントの情報としてプレゼンテーションサービスから通知される。ワークフローコントローラ内での関知はない。

3. セション変数

セション変数はセション中ずっと有効に保持される link 領域と、同一 LD 内で有効な spa 領域とがある。いずれも定義情報に従いワークフローコントローラ内で確保保持される。

4. 管理情報

MONTSUQI バックエンドが必要とする管理情報でセション中ずっと有効である。アプリケーションから直接操作参照することは、原則的にできない。限られた情報が mcp 領域としてアプリケーションから操作可能である。

3.3.3　スケジューリング

アプリケーションサーバはの動作は、ワークフローコントローラからの処理要求によって制御されている。そのため、**MONTSUQI** の動作はこのワークフローコントローラによって制御されている。

3.4 アプリケーションサーバ

実際の業務を行うプロセスである。ワークフローコントロールから 1 つづつメッセージを読み出し、1 つづつ結果を返す。

1 つのアプリケーションサーバは、複数のクライアントからの要求を処理する。これはセション変数をワークフローコントローラとの間で受け渡すことによって実現されている。つまり、実行に必要な状態変数[*1]はトランザクション毎に全て完全な形で渡される。アプリケーションはこの状態変数とデータベースのみを元に処理を行う。

正しく作られたアプリケーションは、プロセスが異なっていても、同じトランザクションを処理すれば同じ結果を返す。データベースの状態とセション変数が変化していなければ、同じトランザクションを何度実行しても、原理的に同じ結果が得られるはずである。この性質を利用してトランザクション異常終了時の再試行が実現されている。

データベースのトランザクションが異常終了した場合は、データベースはロールバックされるため、トランザクション開始前の状態に戻っている。ここであらためてトランザクションを試行すれば、同じ条件で処理が実行される。その時にトランザクションを異常にした原因 (デッドロックやプロセスの異常終了) がなくなっていればデータベースのトランザクションは成功する。データベースのトランザクションが成功した時に、初めて状態変数を更新する。

3.4.1 概要

MONTSUQI のアプケーションサーバは、以下の機能を持つ。

- **LD 電文の解釈**
- **プログラムの実行制御**
- **インターフェイスの翻訳**

アプリケーションサーバ上のアプリケーションは、アプリケーションサーバから呼び出されるサブルーチンとして作成される[*2]。

3.4.2 言語ハンドラ

アプリケーションサーバは、言語による制御やデータ形式の違いを吸収するために、言語毎に異るハンドラが存在している。この言語ハンドラを記述することにより、多くの言語に対応することが出来る。

言語ハンドラは**言語クラス**を元にして、データのシリアライズの方法や文字のエンコーディング等を指定して生成されるインスタンスである。このインスタンスをアプリケーションモジュールと `bind` することにより、アプリケーションモジュールの実行要求があった時に、指定した言語ハンドラの配下でアプリケーションモジュールが動作する。

現在提供されている言語クラスは、

1. **C**
2. **OpenCOBOL**

[*1] 状態変数の内容はトランザクションパケットに全て入っている

[*2] 実装としてサブルーチンそのものであるかどうかは、言語クラスモジュールの実装による

3. **dotCOBOL**
4. **Ruby**
5. **Exec**

のものがある。

3.4.3　アプリケーションの呼び出し

アプリケーションサーバはイベントの発生した**ウィンドウ**に対応したアプリケーションを検索し、対応したアプリケーションをサブルーチンとして呼び出す。その呼び出しパラメータは、

- *mcp*
  アプリケーションサーバの制御情報領域
- *spa*
  アプリケーション共通領域
- *link*
  セション共通領域
- *scr*
  プレゼンテーション情報

である。
アプリケーションから見た **MONTSUQI** は、

1. **イベントが発生する**
2. **トランザクションを開始する**
3. **アプリケーションを呼び出す**
4. **トランザクションをコミットする**
5. **ウィンドウを描画する**

といった動きをする。
アプリケーションは呼び出されてから復帰するまでに、必ず一度はプレゼンテーションデータ出力のための API を呼び出さなくてはならない。これは画面描画のためのサブルーチンであるが、実際の描画は復帰後に行われる。

3.4.4　システム API

アプリケーションサーバ上のアプリケーションが **MONTSUQI** の提供するサービスを利用する場合は、システムの用意した API を通じて要求を出す。システムの提供するサービスは、

- **オンライン機能に関するも**
- **データベース機能に関するも**
- **その他**

に大別される。

オンライン機能に関するも

現在システムが共通で提供しているオンラインに関する機能は、**ウィンドウの描画**のみである。

オンライン機能を利用するのに必要な情報は、**出力するウィンドウ名**と**出力のタイプ**である。出力のタイプと機能を表 3.1 に示す。

CURRENT	現在の画面のまま出力
NEW	指定した画面を新しく開く
CLOSE	指定した画面を閉じる
CHANGE	指定した画面を新しく開き、元の画面を閉じる
BACK	前の画面を新しく開き、現在の画面を閉じる
FORK	指定した画面を新しく開く。JOIN で元に戻れる
JOIN	FORK した画面から戻る
EXIT	セションを終了させる

表 3.1 出力タイプ

データベース機能に関するもの

3.5 でも述べているように、**MONTSUQI** のアプリケーションが操作して良い全ての資源は、データベースとして見えるようになっている。そのため、資源の操作は全てデータベースの操作になる。

データベース機能に関する操作はさらに、

- **データベースの操作**
- **テーブルの操作**

に大別される。

データベースの操作に必要な情報は、**データベースの名前**である。データベースの操作では、以下の機能は **MONTSUQI** 内部からも使っている関係から、データベースのクラスによらず定義済みである。

DBOPEN
データベースの利用を開始する
DBDISCONNECT
データベースの利用を終わる
DBSTART
トランザクションを開始する
DBCOMMIT
トランザクションをコミットする

テーブルの操作に必要な情報は、**データベースの名前**, **パスの名前**, **テーブルの領域**である。テーブルの操作に具体的にどのような機能があるかは、データベースのクラスの実装やデータベース定義体の定義内容に依る。

3.5 データベースレイヤ

3.5.1 MONTSUQI の資源管理

MONTSUQI の管理下にある資源は、全てデータベースとして見えるようになっている。このため、全ての資源操作はデータベースのへのアクセスと同じ操作になる。つまり、**MONTSUQI** の資源の管理はデータベースの管理だということである。

MONTSUQI のデータベース機能はこのような考え方から、データベース操作が抽象化され、個々の特性は極力隠蔽されている。このため同じような特性を持つデータベース (たとえばリレーショナルデータベースどうし) であれば、異なるエンジンに置き換えてもアプリケーションに手を入れることなく移行することができる。

MONTSUQI ではシステムの操作、たとえば外部のプログラムやシステムの稼働環境もデータベースとして抽象化されている。このことにより、データベースレイヤの機能として用意されているトランザクション管理やレプリケーション、またバックアップといったことが、システムの操作にも適用可能になる。

3.5.2 データベースプラグイン

前節で説明したように、**MONTSUQI** の資源は全てがデータベースである。データベースの操作は**データベース操作モジュール**が行うようになっているが、このモジュールはプラグインとして拡張可能になっている。現在のところ、**MONTSUQI** が提供しているデータベースの種類 (クラス) は、

- **PostgreSQL**
  PostgreSQL をアクセスする
- **Shell**
  shell をキックする
- **System**
  システム環境の状態を獲得する

といった 3 種類であるが、**MONTSUQI** 自体はこれらに依存したものではなく、必要に応じて他のデータベースエンジンに対応することは難しくない。

3.5.3 データベースオペレーション

現代のデータベースは、一般的には SQL で操作される。SQL は高水準のデータベース操作を提供する宣言型の言語であり、ANSI/ISO/JIS によって標準化された言語である。そのため、多くのデータベースの上で操作言語として採用されている。しかし残念なことに、SQL はデータベースエンジン毎に方言があり、またデータベースエンジンの構造によって効率の良い書き方が異なるため[*3]、必ずしも互換性の高い言語ではない。そのため、アプリケーションの中に SQL を書いた場合、データベースエンジンが変わるとそれに応じて SQL 部分にも手を入れる必要が出て来る。その結果、アプリケーションがどのようなデータベースエンジンの上で動かということに依存してしまう。

また、**MONTSUQI** では多くの資源管理がデータベースとして抽象化されている。これらは一般で言うところのデータベースとは異なるものをデータベースとして扱えるようにしているだけなので、元々SQL が使え

[*3] 本来高水準言語なので、書き方による効率の差はないはずであるが、現実には存在する

るようなものではないものも多い。

これらの問題を解決するために、**MONTSUQI** ではデータベース操作言語を直接アプリケーション中に記述するのではなく、データベースを定義する定義体の中に、データベースを操作するための手続きを記述してやり、アプリケーションからはそれを呼び出すような形でデータベースの操作を実現している。こうしてやることにより、アプリケーションからはより高い抽象化されたレベルでのデータベース操作を実現することが可能になり、データベース定義の方からは、よりそのデータベースに合ったような操作を記述することができるようになる。

3.6　モニタ

モニタは **MONTSUQI** 全体のプロセスを管理するもので、以下のことを行う。

- **プロセスの起動**
- **プロセスの再起動**
- **プロセスの停止**

MONTSUQI のプログラムは単独で起動可能であるが、確実に正しく起動するためには、モニタに起動を任せる方が安全である。

3.6.1　プロセスの起動

モニタはディレクトリの情報を元にプロセスの起動を行う。**MONTSUQI** はプロセス間通信を TCP/IP で行うのが基本であるため、プロセスは異なるホスト上で起動しても処理可能である。モニタはこのような記述があると、モニタが起動されたホスト名を元にそのホスト上で起動するかどうかの判断も行う。

3.6.2　プロセスの再起動

モニタはプロセスの稼働状態を監視していて、プロセスの再起動オプションが指定されていると、

- **アプリケーションサーバ**
- **データベースリダイレクタ**

についてはプロセスが異常終了した際に自動的に再起動が行われる。

3.7 ユーティリティ

3.7.1 copygen

MONTSUQI の各種定義体から COBOL プログラムに必要な各種 COPY 句を生成するプログラムである。生成する COPY 句は、

- **画面情報領域**
- **画面データ領域 (dotCOBOL のみ必要)**
- **DB データ領域 (dotCOBOL のみ必要)**
- **DB パス情報**
- **DB 情報領域**
- **DB 通信データ領域 (dotCOBOL のみ必要)**
- **SPA 領域**
- **LINK 領域**
- **MCP 領域**

である。これらの領域の項目名は、定義体の定義内容に由来するものが自動的に生成される。また、オプションにより接頭語接尾語等の変更をすることも可能である。

3.7.2 dbgen

DB の定義体を元に create 文を生成するプログラムである。

3.7.3 fdw/fdd

ネットワーク上のホストにファイル転送を行い、そのファイルを転送先のホスト上で処理をするプログラムである。`fdd` はホスト上で待機する daemon であり、`fdw` は `fdd` と通信を行いファイルと処理要求を送る。

このコマンドは、主にはホストに接続された外部装置に、転送したファイルを書き出すといった目的に使う。

第 II 部

内部構造編

第 4 章

言語ハンドラ作成の手引

4.1 言語ハンドラとは

言語ハンドラとは、MONTSUQI のアプリケーションサーバが、アプリケーションを実行する時、言語固有のインターフェイスを吸収する目的で存在する。つまり、MONTSUQI から見たアプリケーションのインターフェイスを実現するモジュールである。

このモジュールにより、言語固有のデータの扱いや呼び出し手順等を吸収することにより、アプリケーションサーバを多くの言語に対応させ、**処理に向いた言語の選択**を容易にする。

4.2 言語ハンドラ構造体

言語ハンドラの構造体は以下のようになっている。

```
typedef struct {
    char    *name;
    Bool    fUse;
    void    (*ExecuteProcess)(ProcessNode *);
    void    (*StartBatch)(char *name, char *param);
    /*  DC function */
    void    (*ReadyDC)(void);
    void    (*StopDC)(void);
    void    (*CleanUpDC)(void);
    /*  DB function */
    void    (*ReadyDB)(void);
    void    (*StopDB)(void);
    void    (*CleanUpDB)(void);
}   MessageHandler;
```

それぞれの値をセットして、アプリケーションサーバ起動時に登録する。メンバの意味は以下のようになっている。

- **name**
  ハンドラの扱う言語の名前を指定する。この名前を**言語ハンドラ名**として、LD 定義体や BD の bind 定義で参照する

- **fUse**
  アプリケーションサーバが起動する時に、その言語ハンドラが実際に使われているかどうかを調べた結果を入れるフラグ領域。起動時に一旦初期化されるので、値は何でも構わない
- **ExecuteProcess**
  オンラインプログラムのトランザクション毎にアプリケーションに処理を渡すために呼ばれる
- **StartBatch**
  バッチ処理を起動する時に呼び出される
- **ReadyDC**
  アプリケーションサーバが起動する時に、言語ハンドラ固有の初期化を行うために呼び出される。
- **StopDC**
  オンラインプログラムを停止する時に呼び出される
- **CleanUpDC**
  オンラインプログラム停止後の後始末に呼び出される
- **ReadyDB**
  アプリケーションが使うデータベースサブシステムを初期化するために呼び出される
- **StopDB**
  データベースサブシステムとアプリケーションを切り離すために呼び出される
- **CleanUpDB**
  アプリケーションサーバ停止後のデータベースサブシステムの後始末に呼び出される

4.2.1 ExecureProcess

オンラインプログラムがトランザクションを処理するために、実際のアプリケーションを呼び出す処理である。

引数は `ProcessNode *`である。この構造体はアプリケーションが実行されるに必要な全ての環境が格納されている。詳細は**??**を参照のこと。

この関数の中での実際の処理は、

1. mcprec 中の dc.module を読み出し、アプリケーションモジュール名を得る
2. アプリケーションモジュールをロードする
3. アプリケーションを呼び出す

ということになる。

アプリケーションの実行は、トランザクション毎にアイソレートされなくてはならない。つまり、同じ内容の引数でこの手続きが呼び出された場合、データベースの更新を除いては、常に同じ結果にならなくてはならない。これは、この関数には処理結果に影響を与えるような状態変数はあってはならないことを意味している。

4.2.2 StartBatch

バッチアプリケーションを起動する時に呼び出される。

引数はアプリケーションモジュール名と、それに渡すべき起動パラメータである。いずれも `char *`の ASCIZ 文字列である。

この関数の中での実際の処理は、

1. アプリケーションモジュールをロードする
2. アプリケーションを呼び出す

ということになる。また、バッチプログラムの場合、後に述べる ReadyDC, StopDC, CleanUpDC は呼び出されないため、アプリケーションが動くための環境の準備等も、この中で行わなければならない。

4.2.3 ReadyDC

アプリケーションサーバが起動する時に、言語ハンドラ固有の初期化を行うために呼び出される。
引数はない。
この関数はオプショナルなので、機能として必要のない場合は、何もしなくても良く、その時には NULL にする。

4.2.4 StopDC

アプリケーションサーバが停止する時に、言語ハンドラ固有の終了処理を行うために呼び出される。
引数はない。
この関数はオプショナルなので、機能として必要のない場合は、何もしなくても良く、その時には NULL にする。

4.2.5 CleanUpDC

アプリケーションサーバが停止する時に、言語ハンドラ固有の終了処理後の後処理を行うために呼び出される。
引数はない。
この関数はオプショナルなので、機能として必要のない場合は、何もしなくても良く、その時には NULL にする。
この関数は全ての言語ハンドラの StopDC を実行した後に呼び出される。

4.2.6 ReadyDB

アプリケーションサーバが起動する時に、言語ハンドラ固有のデータベースの初期化を行うために呼び出される。
引数はない。
この関数はオプショナルなので、機能として必要のない場合は、何もしなくても良く、その時には NULL にする。

4.2.7 StopDB

アプリケーションサーバが停止する時に、言語ハンドラ固有のデータベースの終了処理を行うために呼び出される。
引数はない。
この関数はオプショナルなので、機能として必要のない場合は、何もしなくても良く、その時には NULL にする。

4.2.8　CleanUpDB

アプリケーションサーバが停止する時に、言語ハンドラ固有のデータベース終了処理後の後処理を行うために呼び出される。

引数はない。

この関数はオプショナルなので、機能として必要のない場合は、何もしなくても良く、その時には NULL にする。

この関数は全ての言語ハンドラの `StopDC` を実行した後に呼び出される。

4.3　ProcessNode 構造体

未了

(まだ)

第 5 章

プロトコル

5.1 外部データベース公開インターフェイス

5.1.1 概要

本システムのデータベースを外部のプログラムに公開するサーバの通信手順について解説をする。

5.1.2 データの伝送

データの伝送には TCP を使う。現在のところ、通信に暗号化は行われない。

5.1.3 セションの開設

セションの開設は、クライアント側からポート番号 8013 に connect することにより行われる。
なお、ポート番号は dbs の起動オプションにより変更可能である。dbs についての詳細は、A.1 参照のこと。

5.1.4 データ形式

- サーバ $leftrightarrow$ クライアント間の通信は行単位で行われる
- 現在のところデータは文字列にエンコードされる通信方法のみが実装されている
- 全ての伝送される文字列は URL encode によりエンコードされる
- NULL 値は文字 0x01 によって表現される。この文字も URL encode の対象となる

5.1.5 セションの流れ

1. **セション開設要求**

   - セションの開設要求を行う
   - クライアント → サーバ
   - 以下のものを空白で区切り送信する
     - **バージョン番号**
     - **ユーザ名**
     - **パスワード**
     - **形式**

string または *stringe* を指定する。*stringe* を指定すると、データ受信の時に型情報が付加される

- 開設要求後には、開設応答に遷移する

2. **セション開設応答**

   - セションの開設の結果の応答を行う
   - セション開設が成功応答の後、コマンド受付可能状態になる
   - クライアント ← サーバ
   - 開設成功時には、以下のものを空白で区切り送信する
     - **固定文字列** *Connect:*
     - **固定文字列** *OK*
   - 開設失敗時には、以下のものを空白で区切り送信する
     - **固定文字列** *Error:*
     - **エラー内容**
   - 開設失敗時のエラー内容には、以下のものがある
     - **version**
       サーバ、クライアントのバージョン不整合
     - **authentication**
       認証エラー
   - セション開設成功通知後は**コマンド送信**に遷移
   - セション開設失敗通知後は切断

3. **コマンド送信**

   - コマンドの送信を行う
   - クライアント → サーバ
   - 以下のものを空白で区切り送信する
     - **処理区分**
     - **コマンド文字列**
   - 処理区分としては、以下のものがある
     - 固定文字列 *Schema:*
       DB スキーマの獲得
     - 固定文字列 *Exec:*
       コマンドの実行
     - 固定文字列 *End:*
       処理の終了。この時コマンド文字列を指定しても無視される
   - データベーススキーマの獲得
     データベーススキーマの獲得の時には、以下のものを:で区切り送信する
     - **テーブル名**
     - **パス名**

     コマンド送出後は**スキーマ送信**に遷移
   - コマンドの実行
     コマンド文字列として以下のものを:で区切り送信する

- **処理名**
  DB 定義体に指定されたマクロ名
- **テーブル名**
  処理対象のテーブル名。テーブルに関係のない処理 (*DBOPEN*, *DBCLOSE*, *DBSTART*, *DBEND* 等) については、指定不要
- **パス名**
  処理対象のパス名。パスの関係ない処理については、指定不要

コマンド送出後は**データ送信**に遷移
実行はデータ送信後に行われる

4. **スキーマ送信**

- データベースの項目定義テキストを送信する
- データベースの項目定義テキストのうち、余分な空白文字 (タブや改行も含む) を除いたものを、1 行にして送信する

5. **状態コード応答**

- コマンドの処理状態の通知を行う
- クライアント ← サーバ
- 以下のものを送信する
  - **固定文字列** *Exec:*
  - **状態通知コード**
- 状態通知コードには、以下のものがある

値	状態
0	OK
1	EOF
2	NONFATAL
-1	引数不正
-2	処理名不正
-3	SQL 不正
-4	その他不正

- 状態通知コード送信後には**データ受信**に遷移

6. **データ送信**

- クライアントからサーバにデータを送る
- クライアント → サーバ
- 以下のものを必要回送信する
  - **レコード名**
  - **固定文字**’.’
  - **項目名**
  - **固定文字列**’: ’
  - **値文字列**
- 必要回送信後は、空行を送信して終了を通知する

- データ送信後は、直前に送信されたコマンドを実行する
- 実行後は**状態コード応答**に遷移

7. **データ受信**

   - サーバからクライアントにデータを送る
   - クライアント ← サーバ
   - 以下のことを必要回行う
     - **必要項目名送信**
     - **項目値受信**
   - 必要回実行後は、空行を送信して終了を通知する
   - データ受信が必要がない場合は、空行のみ送る
   - 実行後は**コマンド送信**に遷移

8. **項目名送信**

   - データを受信したい項目名をサーバに通知する
   - クライアント → サーバ
   - 以下のものを送信する
     - **レコード名**
     - **固定文字**'.'
     - **項目名**
     - **名前送信指定**
       ここで固定文字列': ' を付加すると、項目名がサーバから送信されず、値だけ送信される。': ' を付加しなければ、項目名と共に値が送信される
   - 項目名指定のさいに下位項目を持つ項目を指定すると、下位項目全てのデータを送信する。この時には名前送信指定をしておくと、項目の名前と共に送られて来るので識別が可能になる。問い合わせの往復がなくなるので、多くの項目を一度に読む時には、この指定が良い
   - 複数の項目が一度に送られる指定を行った場合、全て送った後は空行が送られて来て終了を通知する

9. **項目値受信**

   - サーバの値をクライアントに送信する
   - クライアント ← サーバ
   - 以下のものを受信する
     - **項目名**
       項目名送信で名前送信指定が行われた場合付加される。項目名にはレコード名も含む
     - **型情報**
       セション開設時に *stringe* を指定すると、型情報が付加される。型情報の前には';' が前置される
     - **固定文字列**': '
       項目名送信で名前送信指定が行われた場合付加される
     - **値**
       URL encode された値を表現する文字列

5.1.6 標準処理名

以下に挙げる処理名は、システムで既定であり、特に定義する必要はない。定義された場合は、既定の動作を置換する。

DBOPEN　データベース処理の開設
DBDISCONNECT　データベース処理の終了
DBSTART　トランザクションの開始
DBCOMMIT　トランザクションの終了

これ以外の既定処理については、データベースハンドラ固有のものとなるので、各データベースハンドラのドキュメントを参照のこと。

5.1.7 型情報

型情報は以下の形式を持っている。

型情報　::= char **型** | varchar **型** | byte **型** | text **型** | number **型** | **整数型** | **レコード型** .
varchar **型**　::= "char" ["(" **文字数** ")"] .
char **型**　::= "char" ["(" **文字数** ")"] .
byte **型**　::= "byte" ["(" byte **数** ")"] .
number **型**　::= "number" ["(" **桁数** ["," **小数点以下桁数**] ")"] .
文字数　::= **整数** .
byte **数**　::= **整数** .
桁数　::= **整数** .
小数点以下桁数　::= **整数** .
text **型**　::= "text" .
整数型　::= "int" .

第 III 部

プログラミング編

第 6 章

共通事項

6.1 考え方

MONTSUQI のアプリケーションが処理をする対象は、ワークフローコントローラから送られて来るトランザクションパケットである。このトランザクションパケットが LD に送られ、aps が LD を読んで処理を行う。トランザクションパケットの内容は、3.3.2 で挙げている。この中のイベント情報には、

1. **イベントの発生したウィンドウ名**
2. **発生したイベントを識別する文字列**
3. **イベントの発生したウィジェット名**
4. **イベントが発生した時の PL サービスの状態**

がある。アプリケーションサーバはイベント発生したウィンドウ名を見て、実際に処理をするべきアプリケーションモジュールを呼び出す。

アプリケーションサーバは、イベント情報のうち、2, 3, 4 を使ってオペレータが何を行ったか判断を行い、それに応じた処理を行う。

6.2 定義体

MONTSUQI で実用的なアプリケーションを動かすためには、以下に示す定義体を用意する必要がある。

- **システム定義体**
- **LD 定義体**
- **PL レコード定義体**
- **SPA レコード定義体**
- **データベース定義体**

またこの他に、

- **画面定義体**
  画面のある表現層を使う場合
- **バッチ定義体**
  オンライン以外のプログラムがデータベースを使う場合
- **データベース公開定義体**
  MONTSUQI 以外のプログラムから **MONTSUQI** 配下のデータベースを使う場合

などが必要になる。

システム定義体

システム定義体は「ディレクトリ」とも呼ばれ、システム全体の動作を規定する。ここにはシステム全体の動作を決定するパラメータが記述される。

LD 定義体

LD 定義体は LD の動作を規定するファイルである。ここには LD が動作するのに必要な情報が記述される。この定義体は LD の数だけ存在する。

PL データ定義体

プレゼンテーションレイヤが扱うデータについて記述するものである。

SPA レコード定義体

2 種類の SPA データについて記述するものである。

データベース定義体

データベース定義体とは、データベースを使うにあたって、そのデータベースのスキーマやアクセス方法を記述するものである。

次節以降ではこれらの詳細について説明をする。

6.3 システム定義体

6.3.1 システム定義体で宣言するもの

システム定義体では、以下のものを宣言する。

1. **システムの名前**
2. **ディレクトリパス**
3. **画面スタックサイズ**
4. **多重化パラメータ**
5. **セション共通領域宣言**
6. **ワークフローコントローラのパラメータ**
7. **LD 宣言**
8. **バッチ定義体の名前**
9. **データベース公開定義体の名前**
10. **データベースグループの定義**

詳細の文法は、付録 C.1 に挙げる。
以下に詳細について説明する。

6.3.2 システムの名前

システムを識別するための名前を書く。

6.3.3 ディレクトリパス

定義体の格納されているディレクトリへのパスを記述する。パスには 2 種類あり、

- **LD 定義体、バッチ定義体、データベース公開定義体へのパス**
- **レコード定義体、データベース定義体へのパス**

がある。これらは便宜のために区別されている。ここで宣言したパスは、プログラム起動時に起動パラメータにより変更することも可能である

6.3.4 画面スタックサイズ

画面遷移を保持するスタックの大きさを指定する。最低値が 16 で、それよりも小さい指定は出来ない (無視される)。

画面スタックは、画面が遷移すると積まれ、「前に戻る」という動作のために使われる。前に戻ればその分降ろされるため、スタックの大きさは画面遷移の深さ分だけあれば良い。ただし、画面が遷移した結果、過去に遷移したのと同じ画面に遷移した場合には、その途中の画面遷移はなかったことになる。これは「前に戻る」動作をした時に、オペレータが混乱しないためである。一般に画面操作は局所性があり、一連の作業を行う場合は同じ画面を何度も遷移するため、実際に画面スタックが膨れ上がることはまずない。そのため、実用的にはデフォルト値のままで十分である。

画面スタックはアプリケーションから参照可能なデータ領域であるため、この定義を変更した場合は COBOL の場合は COPY 句の再生成と全アプリケーションの再コンパイルが必要になる。

6.3.5　多重化パラメータ

トランザクション処理の並列化をする戦略を指定する。値の意味については、2.1.5 を参照。安全性が必要な場合は、並列化のレベルを下げ、スループットを上げたい場合には並列化のレベルを上げる。

並列度が上げられるかどうかは、異なるトランザクションで更新されるデータベースの依存関係が問題となる。異なるトランザクション間で依存関係のあるレコードを操作する場合、排他制御上の問題 (多くはデッドロック) が発生する危険性がある。排他制御上の問題から回復するためには、排他を行ったているトランザクションのうちのどれかをアボートしなければならなくなり、頻発すると処理効率を下げる。このため、並列度を上げるためには、排他制御上の問題が発生しないことが重要である。

適切に設計されたデータベースを正しい手順で操作する場合には、排他制御上の問題は起きないはずである。排他制御上の問題が発生する危険がなければ、高い並列化レベルにしても問題はない。

ただし、**MONTSUQI** では排他制御上の問題が発生してトランザクションがアボートした場合でも、システムによって再試行されるため、頻発して処理効率を下げる程のことがなければ並列度が高くても大きな問題にはならない。

並列度を上げると、**MONTSUQI** は指定された並列度の範囲で投入されたトランザクションを並列に処理をしようとする。このことによりオンラインの処理効率は上がる。しかしその結果として、

- **アプリケーション実行のための資源 (メモリや CPU) を多く消費する**
- **データベースエンジンの負荷が高くなる**

といったことが起きる。このため、資源不足に陥ってかえってパフォーマンスが落ちることもありえる。多くの場合、アプリケーションのパフォーマンスはデータベースエンジンの処理速度に依存しているため、データベースエンジンへの負荷が高いアプリケーションの場合、この現象は顕著に起きる。また、パフォーマンスが低下すると、データベースが施錠されている時間が長くなるため、それだけ排他制御上の問題が発生する危険性も高くなるので注意が必要である。

また、並列度を上げることは、待ち行列を処理するものを増やすことであって、処理そのものを速くするものではない。つまり、端末数増加によるパフォーマンスの低下を抑えるための手段であって、元々遅い処理を速くする効果はないということも留意する必要がある。

6.3.6　セション共通領域宣言

2.1.4 で説明したように、SPA 領域には 2 つの種類がある。ここではセション共通領域を定義するデータ構造定義体を宣言する。

セション共通領域は、2.1.4 でも述べたように、セションが有効な間は内容が保持される。このため、この領域を大きく取ると、常にそれだけの空間を使うことになる。また、全てのアプリケーションから共有される領域であるため、あまり多くのデータ項目をセション共通領域に置くと、思わぬ虫の元になる。このようなことから、セション共通領域に置くデータは極力少量にするのが望ましい。

セション共通領域はアプリケーションから参照可能なデータ領域であるため、この定義を変更した場合は COBOL の場合は COPY 句の再生成と全アプリケーションの再コンパイルが必要になる。

6.3.7 ワークフローコントローラのパラメータ

ここでは、ワークフローコントローラが待機する 3 種類のポートを定義する。
ワークフローコントローラは、異なる 3 つの接続を受ける。それは、

- **PL サービスからの接続**
- **アプリケーションサーバからの接続**
- **制御のための接続**

である。ここではそれぞれについての定義を行う。これらは省略した場合には、表 6.1 のようになる。

多くの場合、特に設定しないでデフォルトのままで問題は起きないが、システムで複数の **MONTSUQI** を動かす場合には使うポートがぶつからないように設定してやらなくてはならない。

PL サービスからの接続	9000
アプリケーションサーバからの接続	9001
制御のための接続	UNIX

表 6.1 デフォルトポート番号

制御のための接続については、セキュリティのことを考えると TCP ではなくて UNIX ドメインを使うことが望ましい。

6.3.8 LD 宣言

システムで使う LD について宣言を行う。**MONTSUQI** では LD はアプリケーションサーバの実行単位でもあるので、この定義は同時にアプリケーションサーバの実行についての記述となる。

多重化パラメータを 'aps' にした場合、1 つの LD をいくつのアプリケーションサーバで処理をするかという**多重度**はここで定義する。多重度は高くしてもプロセス数が増えるだけなので、実際に使われなければプロセスのために仮想空間を多く消費するだけで特に害はない[*1]。しかし、LD につながるキューのエントリ数以上に増やしても、起動されるだけで使われないプロセスが存在することになり、仮想空間の浪費となる。

ある瞬間の **LD につながるキューのエントリ数**は、同一 LD に同時に処理要求を出した端末の数と等しい。一般に LD は業務毎に作られるものなので、同じ業務を処理する端末の最大数が、指定して意味のある最大の多重度ということになる。実際にはトランザクションは断続的に発生するので、それよりも小さい多重度で十分である。また、むやみに多重処理をしても、データベースエンジンがボトルネックとなるため、かえって遅くなることも考えられる。そのため、このパラメータは実際に運用しながら最適値を求めるべきである。

6.3.9 バッチ定義体の名前

システムで使うバッチ定義体の宣言を行う。バッチ定義体は LD 宣言同様に、複数個指定可能である。

バッチ定義体の具体的な内容は後述するが、バッチ定義体には定義されたバッチで使う資源が記述されている。プログラムの書きやすさから言えば、「1 つの定義体にたくさんのプログラムを宣言し、多くのリソースを使う」ようにするのが楽である。しかし、それでは思わぬプログラムの虫により、資源を破壊してしまったりセキュリティホールになったりする危険性がある。このようなことを避けるために、「**バッチのグループ毎に定義体を作り、必要最低限のリソースを使う**」ようにするべきである。

[*1] 動かないプロセスは仮想空間のみを消費し、実メモリは消費しない。無論、動かないプロセスは CPU も消費しない。

6.3.10　データベース公開定義体の名前

データベースを外部公開するための定義体の宣言を行う。この定義体も LD 宣言同様に、複数個指定可能である。

これもバッチ定義体の使い方と同じような問題を持っているので、極力「万能定義体」を作らないようにするべきである。

6.3.11　データベースグループの定義

システムで使うデータベースグループについて定義をする。データベースグループについては 2.1.6 を参照のこと。データベースグループ名は指定しないと、無名のデータベースグループを定義したことになる。

データベースグループには大別して、

- **直接データベースをアクセスするためのもの**
- **移送のためのもの**

の 2 種類があるが、これらは定義項目の違いだけなので、厳密な区別があるわけではない。

データベースグループを定義する時の定義項目は現在のところ、

1. **commit 優先度**
2. **データベースクラス名**
3. **データベース名、データベースユーザ名、パスワード、データベース接続ポート**
4. **移送先**
5. **移送ポート**
6. **ログファイル名**

である。

1.commit 優先度

データベースをコミットする時の相対優先度である。これはデータベースの種類によってコミットが後になった方が都合が良いもののために、主に優先度を下げる目的で指定する。具体的には、Shell ハンドラを使うアプリケーションで他のデータベースよりも後にコミットされることを仮定していることが多いため、このパタメータを指定する。

優先度は相対値であり、絶対値には意味がない。

デフォルト値は 50 である。

2. データベースクラス名

データベースハンドラの名前を指定する。デフォルト値はない。ログファイルを書き出すだけのデータベースグループであっても、そのログがどのようなデータベースクラス用のものかを明示する必要があるので、必ず指定しなければならない。

3. データベース名、データベースユーザ名、パスワード、データベース接続ポート

データベースエンジンに接続するために指定する。データベース名以外は、データベースエンジンによっては省略可能である。データベース名がない場合にはデータベースには接続されない。

4. 移送先

データベースアクセスの記録は、本来処理対象となるデータベースエンジンの他のところにも転送することができる。これを**データベースの移送 (リダイレクション)** と呼ぶ。移送はデータベースグループを単位として宣言する。つまり、あるデータベースグループのアクセスを別のデータベースグループへも行うというのが移送である。これを使って、

- **データベースの二重化 (レプリケーション)**
- **データベースの遠隔更新**
- **実行ログのファイルへの保存**
- **実行ログのリアルタイムバックアップ**

等が可能となる。

データベースグループの定義の中に移送先を宣言すると、その移送先のデータベースグループに対応した `dbredirector` に対してアクセスを移送する。`dbredirector` は対応したデータベースグループの記述に従って処理を行う。

5. 移送ポート

移送のための `dbredirector` が待機するポートを指定する。移送のデータは、このポートに送られる。

6. ログファイル名

データベースの処理の記録をファイルに落とす時のファイル名を指定する。

ログファイルの形式はデータベースのクラスによって異なるが、記録されている内容は、

- **発生時刻**
- **トランザクション連番**
- **処理内容**

である。また、トランザクション毎に何らかの区切りが入れられ、トランザクションの開始終了についての命令は除かれる。

データベースクラスが “PostgreSQL” の場合は、1 トランザクションが 1 行となり、発生時刻とトランザクション連番がコメントの形式で行の先頭にあり、処理内容は処理の SQL が書かれている。そのため、このまま `psql` に入力してやれば処理が行われる。

6.3.12 例

以下は定義の例である。

```
name    demo;

base    "~/MONTSUQI/panda/samples";

ddir    "=/sample1";
record  "=/sample1";

multiplex_level aps;

linkage     demolink;

control {
    port    "#/tmp/wfc";
};

wfc {
    port    ":9002";
};

ld  {
    demo1   2;
    demo2;
};

db_group    "log"   {
    priority    100;
    type    "PostgreSQL";
    port    "other";
    name    "ORCA2";
    redirect_port   "localhost:9300";
    file    "orca.log";
};
```

```
db_group    "ORCA"  {
    type    "PostgreSQL";
    port    "localhost";
    name    "ORCA";
    redirect    "log";
};

db_group    "system"    {
    name    "system";
    type    "System";
};

db_group    "shell" {
    type    "Shell";
    name    "shell";
    priority    1000;
};
```

このシステム定義体には、概略以下のようなことが記述されている。

- 'demo' という名前のシステムを定義している。
- ファイルを参照するための基本となるディレクトリは、"~/MONTSUQI/panda/samples"である。この中で~は起動した利用者のホームディレクトリとして解釈される。
- 各種定義体は"=/sample1"**に置かれる。この中で**\verb=—は base で指定したディレクトリとして解釈されるため、結果的には"~/MONTSUQI/panda/samples/sample1"と指定したのと同じである。
  レコード定義体も同様である。
- 多重度については"aps" が指定されているため、aps は多重化して起動される。
- セション情報を保持するレコードの名前は 'demolink' である
- 制御のためのポートは"/tmp/wfc"という UNIX ドメインである
- aps からの接続を待つポートは TCP の 9002 番ポートである。ホストは localhost が指定されているとみなされている
  PL サービスからのポートは指定されていないので、デフォルト値 (TCP で localhost でポート番号は 9000 番) である
- LD は 2 つあり、demo1 と demo2 という名前である。demo1 については aps が 2 つ起動される
- データベースグループには、ORCA,log,system と shell がある
- log は PostgreSQL ハンドラを使うものであり、other というホストに待機しているデータベースエンジンの ORCA2 という名前のデータベースに処理を行うと共に、orca.log というファイルにも出力をしている。これは dbredirector が TCP の localhost ポート番号 9300 で待機している
- ORCA は PostgreSQL ハンドラを使うものであり、localhost に待機しているデータベースエンジンの ORCA という名前のデータベースに処理を行う。また log に対して移送を行う
- system は System ハンドラを使うものである。System ハンドラはデータベースの名前という概念は存在しないが、ログ出力でないということを明示するために system という名前をダミーで指定している
- shell は Shell ハンドラを使うものである。Shell ハンドラにもデータベースの名前という概念は存在しないが、ログ出力でないということを明示するために system という名前をダミーで指定している。こ

のデータベースをコミットする優先度は 1000 であり、他のデータベースよりも遅いタイミングである

6.4 LD 定義体

6.4.1 LD 定義体で宣言するもの

LD 定義体では、以下のものを宣言する。

1. **LD の名前**
2. **並列実行グループについての記述**
3. **実行ハンドラの定義**
4. **bind 定義 (ウィンドウ名と処理モジュールの対応)**
5. **デフォルト定義 (可変長配列、可変長文字列のための宣言)**
6. **実行ディレクトリの宣言**
7. **処理データの定義**
8. **データベースの宣言**
9. **キャッシュの宣言**

詳細の文法は、付録**??**に挙げる。
以下に詳細について説明する。

6.4.2 LD の名前

LD を識別するための名前を書く。この名前は、6.3.8 で説明した宣言で使われる。

ただし、現在のところ **MONTSUQI** が LD の定義を検索する時には、定義体のファイル名を使うので、ファイル名の拡張子を除いた部分と LD の名前は一致していなければならない。LD 定義体の拡張子は`.ld` である。

6.4.3 並列実行グループについての記述

6.3.5 で説明した多重化パラメータが `id` の場合に使うグループ名を宣言する。多重化パラメータが `id` 以外の場合には意味を持たない。

この名前には特別に制約や文法はなく、グループの名前として識別されれば何であっても構わない。

この名前が同じものが、同じ `id` に所属するということになる。同じ `id` の LD は、並列に処理は**されない**。

6.4.4 実行ハンドラの定義

実行モジュールを動作させるためのハンドラの特性について定義する。
ハンドラには以下のパラメータがある。

1. **クラス**
2. **データ変換規則**
3. **起動パラメータ**
4. **文字コード**
5. **文字列変換規則**
6. **アプリケーションをロードするパス**

1. クラス

アプリケーションモジュールが記述されてる言語に対応したハンドラクラスを指定する。
現在のところ、

- **dotCOBOL**
- **OpenCOBOL**
- **C**
- **Exec**
- **Ruby**

が組み込み可能であるが、実際にどれが動作可能であるかは、コンパイル時に対応するクラスモジュールを組み込む指定にしたかどうかで決まる。

2. データ変換規則

MONTSUQI の標準データ形式から、アプリケーションが処理できるデータ形式に変換する時の規則を指定する。
多くのハンドラクラスではデータ変換規則は固定であり、その場合にはこの指定は意味を持たない。しかし、ハンドラクラスによっては、いくつかの規則に対応したものがあるため、その場合にはアプリケーションの処理しやすいデータ形式を指定してやる。
現在のところ指定可能なデータ変換規則は、

- **dotCOBOL**
- **OpenCOBOL**
- **CSV1, CSV2, CSV3, CSVE, CSV**
- **RFC822**
- **CGI**
- **XML**

であるが、有効であるかどうかは、ハンドラクラスの実装による。

3. 起動パラメータ

実行ハンドラを起動する時に与えるパラメータ文字列を指定する。このパラメータ文字列がどのような意味を持つかは、ハンドラクラスの実装に依存する。

4. 文字コード

MONTSUQI 標準の文字コードは `UTF-8` であり、国際化されている。しかし、アプリケーションによっては特定の文字コード系に依存したものが少なくない。そのために、データ変換を行う時にコード変換する文字コードを指定する。
MONTSUQI では文字列の変換には `iconv(3)` を使っているので、ここで指定可能な文字コードは、`iconv(3)` の実装による。
指定がない場合にはハンドラクラスの実装に依存する。

5. 文字列変換規則

テキスト系のデータ変換規則を使うハンドラの場合に、特殊な意味を持つ文字が含まれるデータの場合には何らかの変換処理が行われる。この時の変換規則を指定する。

現在のところ、

- **URL**
- **base64**

が指定可能である。

このパラメータが意味を持つのは、データ変換規則のうち、

- **CSV1, CSV2, CSV3, CSVE, CSV**
- **RFC822**
- **CGI**
- **XML**

である。

6. アプリケーションをロードするパス

アプリケーションをロードするディレクトリパスを指定する。複数のディレクトリからロードしたい場合には、‘:’で区切って指定する。

基本的にハンドラは基本的にプログラマが定義するものであるが、以下のものについてはシステムで暗黙に定義されている。

```
handler "OpenCOBOL" {
    class       "OpenCOBOL";
    serialize   "OpenCOBOL";
    coding      "euc-jp";
    start       "";
};
handler "C" {
    class       "C";
    start       "";
}
handler "Exec"  {
    class       "Exec";
    serialize   "CGI";
    coding      "euc-jp";
    start       "%m";
}
```

ハンドラ定義は同じ名前のものが複数定義されると、後に書かれた内容で上書きされる。そのため、既定のハンドラの動作を変えたい場合には、変更した内容で定義する。

6.4.5　bind 定義 (ウィンドウ名と処理モジュールの対応)

イベントの発生したウィンドウ (ウィンドウという概念を持たない PL デバイスは対応する他の概念のもの。以下同様) と実行ハンドラ処理とモジュールについての対応をつける。

複数のウィンドウを同じモジュールで処理することは可能である。また、実行ハンドラが異なれば同じモジュール名でも別のモジュールを指定したことになる。初画面要求[*2]があった場合の処理モジュールを指定には、ウィンドウ名に""を指定する。

6.4.6　デフォルト定義 (可変長配列、可変長文字列のための宣言)

配列や文字列が固定長でなければならない言語を使う場合、**MONTSUQI** の機能である可変長配列や可変長文字列がそのままでは処理ができない。そのため、固定長しか許さない言語のために、可変長配列や可変長文字列を十分な長さの固定長として扱うような機能がある。この長さは大き過ぎれば領域の無駄となり、小さ過ぎれば処理不能になる。そのため、この大きさは定義可能になっている。固定長しか許さない言語の場合は、ここで指定した大きさで、copyge やデータ変換が行われる。

配列	64
文字列	256

表 6.2　暗黙値

可変長配列や可変長文字列を使ったデータ定義がある場合に、このパラメータを変更した場合には、copygen のやり直しと再コンパイルが必要になる。

指定のない場合には、表 6.2 に示すようになる。

不定長配列や不定長文字列が直接扱える言語の場合は、特に意味を持たない

6.4.7　実行ディレクトリの宣言

アプリケーションサーバが実行されるディレクトリを指定する。

原則的にはアプリケーションサーバの動作は、実行されるディレクトリに依存してはならない。この宣言はなくても構わないはずのものである。

6.4.8　処理データの定義

アプリケーションが使うデータのうち、**MONTSUQI** が管理するデータについての定義を行う。

ここで定義するデータは、

- **アプリケーション共通領域**
- **プレゼンテーション情報**

であり、レゼンテーションデータの並び順は、ここで定義された順である。

ここに指定する名前は、データ定義体の名前であり、拡張子に".rec" を補ったものがファイル名となる。指定しても該当するファイルがない場合はそのデータは存在していないとみなされる。プレゼンテーションデータの定義とウィンドウとは直接の関係はない。そのため、データを持たないウィンドウがある場合には無理にそのデータについて記述する必要はない。

[*2] ウィンドウを出す前に LD に接続されること。アプリケーションの最初の画面の時にはそうなる

アプリケーション共通領域定義は、実際のアプリケーション共通領域の構造を記述する。ワークフローコントローラは、この領域については、単純に領域確保と内容保持のみを行い、内容には関知しないが、構造が記述されていればアプリケーションサーバの言語ラッパで言語依存のデータ構造について互換を取るようにする。

3.3.2 で述べたように、トランザクションパケットは、ここで定義したデータと、

- **イベント情報**
- **システム定義体で定義したセション共通領域**
- **システムが管理する管理情報**

によって構成される。

6.4.9 データベースの宣言

アプリケーションが使うデータベースの宣言をデータベースグループ毎に行う。ここに書かれる名前は、データベース定義体の名前であり、拡張子に".db"を補ったものがファイル名となる。

宣言していないデータベースは使えない。

DB グループ名を指定しないと、無名の DB グループを指定したことになる。

6.4.10 キャッシュの宣言

トランザクションパケットをキャッシュするセションの数を指定する。

通常、トランザクションパケットはワークフローコントローラによって管理される。このため、トランザクション毎にアプリケーションサーバはワークフローコントローラと通信して、トランザクションパケットを受け取る。

しかし、同じ端末から同じ LD のアプリケーションを使っている場合、トランザクションパケットの中で外的要因によっての変化があるものは、

- **イベント情報**
- **システムが管理する管理情報**
- **プレゼンテーション情報**

だけであって、SPA 領域についてはアプリケーションで加工したままである。そこで、ワークフローコントローラとアプリケーションサーバ間の通信量を減らしてやるために、SPA 領域についてキャッシュをする。キャッシュは現在のところメモリ上に取られるので、あまり大きくすることはできない。そのためキャッシュするセションの上限値を決めてやり、これを超えた場合はキャッシュを破棄するようになっている。

キャッシュが存在する場合であっても、別の LD に処理が移った場合は、SPA 領域の内容は変化している可能性が高いので、キャッシュの内容は破棄されて、ワークフローコントローラから取得する。

6.4.11 例

以下は定義の例である。

```
name    sample;

handler "RubyExec"  {
    class       "Exec";
    serialize   "CGI";
    start       "/usr/bin/ruby -Ke %p/%m.rb";
    coding      "euc-jp";
    encoding    "URL";
};

handler "Ruby"  {
    class       "Ruby";
    loadpath    "=/sample2";
    coding      "euc-jp";
};

bind    ""          "OpenCOBOL" "SAMPLE1";
bind    "list"      "OpenCOBOL" "SAMPLE1";
#bind   ""          "RubyExec"      "SAMPLE1";
#bind   "list"      "RubyExec"      "SAMPLE1";
#bind   ""          "Ruby"      "Sample1";
#bind   "list"      "Ruby"      "Sample1";

arraysize   20;
textsize    200;
cache       5;

home    "=/sample2";

data    {
    spa     samplespa;
    window  {
        list;
    };
};
db  "ORCA"  {
    adrs;
};
db  "shell" {
    shell;
};
db  "system"    {
    metadb;
};
```

この LD 定義体には、概略以下のようなことが記述されている。

- この LD の名前は ‘sample’ である
- RubyExec というハンドラが定義されている

  - ハンドラクラスは Exec である
  - データ変換規則は CGI である
  - このハンドラは起動パラメータとして"/usr/bin/ruby -Ke %p/%m.rb" を受け取る。Exec というハンドラでは、これはアプリケーションモジュールを exec(2) する時の引数として用いる
  - 文字コードは euc-jp である
  - データ変換規則 CGI はテキスト系の規則なので、特殊文字等は URL エンコーディングされる
- Ruby というハンドラが定義されている
  - ハンドラクラスは Ruby である
  - モジュールのロードパスは"=/sample2"である。=はシステム定義体と同じ意味である
  - 文字コードは euc-jp である
-
  - ウィンドウ名を持たない初画面では、OpenCOBOL というハンドラを使って、SAMPLE1 というモジュールを呼び出す。OpenCOBOL というハンドラについては、暗黙宣言のものを使っている
  - ウィンドウ名が list なら、OpenCOBOL というハンドラを使って、SAMPLE1 というモジュールを呼び出す。
  - それ以外の bind 文はコメントアウトされている。
- 可変長配列については、固定長配列しか許さない言語では要素数が 20 としてデータ変換する
- 可変長文字列については、固定長文字列しか許さない言語では文字列長が 200 としてデータ変換する
- キャシュするセションは 5 である
- アプリケーションサーバが実行されるディレクトリは、"=/sample2" である
- トランザクションパケットのうち、アプリケーション共通領域のデータ定義体は samplespa という名前である

  プレゼンテーションデータの定義体は list という名前である
- データベースは、
  - ORCA というデータベースグループの adrs
  - shell というデータベースグループの shell
  - system というデータベースグループの metadb

  を使う

6.5　PL データ定義体

6.5.1　PL データについて

PL データはイベント処理の基本となる、利用者とやりとをするためのデータである。多くの場合、PL クライアントで何らかの形で利用者に対して可視化され、利用者によって入力される。実際にどのような形で可視化されるかは、PL クライアントの特性による。

PL データはウィンドウに対応したレコードによって構成される。

6.5.2　glclient V1 プロトコルにおける PL データ

`glclient` の V1 プロトコルでは、PL データの構造とウィジェットの包含関係は一致していることが要求される。このため、ウィジェットの包含関係に合わせて PL データを定義してやる必要がある。

`glclient` の画面レイアウト情報は、`glade` によって生成される XML である。V1 プロトコルで使う PL データの定義体は、この XML を元に骨格を生成し、それを編集することによって作る。

図 6.1　glclient の画面例 (1)

図 6.1 のような画面を定義した画面定義データは、図 6.2 のようになる。

```
<?xml version="1.0"?>
<GTK-Interface>
<project>
  <name>Project2</name>
  <program_name>project2</program_name>
  <directory></directory>
  <source_directory>src</source_directory>
  <pixmaps_directory>pixmaps</pixmaps_directory>
  <language>C</language>
  <gnome_support>True</gnome_support>
  <gettext_support>True</gettext_support>
</project>
<widget>
  <class>GtkWindow</class>
  <name>list</name>
  <title>一覧</title>
  <type>GTK_WINDOW_TOPLEVEL</type>
  <position>GTK_WIN_POS_NONE</position>
  <modal>False</modal>
  <allow_shrink>False</allow_shrink>
  <allow_grow>True</allow_grow>
  <auto_shrink>False</auto_shrink>
  <widget>
    <class>GtkFixed</class>
    <name>fixed1</name>
    <widget>
      <class>GtkEntry</class>
      <name>key</name>
      <x>16</x>
      <y>16</y>
      <width>158</width>
      <height>22</height>
      <can_focus>True</can_focus>
      <signal>
        <name>changed</name>
        <handler>changed</handler>
      </signal>
      <editable>True</editable>
      <text_visible>True</text_visible>
      <text_max_length>32</text_max_length>
      <text></text>
    </widget>
    <widget>
      <class>GtkButton</class>
      <name>button1</name>
      <x>192</x>
      <y>16</y>
```

図 6.2　画面定義例 (1)

```
    <width>55</width>
    <height>22</height>
    <can_focus>True</can_focus>
    <signal>
      <name>clicked</name>
      <handler>send_event</handler>
      <data>Search</data>
    </signal>
    <label>検索</label>
  </widget>
  <widget>
    <class>GtkCList</class>
    <name>clist1</name>
    <x>16</x>
    <y>48</y>
    <width>400</width>
    <height>216</height>
    <can_focus>True</can_focus>
    <signal>
      <name>click_column</name>
      <handler>send_event</handler>
      <data>Clist</data>
    </signal>
    <columns>4</columns>
    <column_widths>72,86,100,80</column_widths>
    <selection_mode>GTK_SELECTION_SINGLE</selection_mode>
    <show_titles>True</show_titles>
    <shadow_type>GTK_SHADOW_IN</shadow_type>
    <widget>
      <class>GtkLabel</class>
      <child_name>CList:title</child_name>
      <name>label1</name>
      <label>名前</label>
      <justify>GTK_JUSTIFY_CENTER</justify>
      <wrap>False</wrap>
      <xalign>0.5</xalign>
      <yalign>0.5</yalign>
      <xpad>0</xpad>
      <ypad>0</ypad>
    </widget>
    <widget>
      <class>GtkLabel</class>
      <child_name>CList:title</child_name>
      <name>label2</name>
      <label>電話</label>
      <justify>GTK_JUSTIFY_CENTER</justify>
      <wrap>False</wrap>
```

図 6.3　画面定義例 (1 続き)

```
        <xalign>0.5</xalign>
        <yalign>0.5</yalign>
        <xpad>0</xpad>
        <ypad>0</ypad>
      </widget>
      <widget>
        <class>GtkLabel</class>
        <child_name>CList:title</child_name>
        <name>label3</name>
        <label>メールアドレス</label>
        <justify>GTK_JUSTIFY_CENTER</justify>
        <wrap>False</wrap>
        <xalign>0.5</xalign>
        <yalign>0.5</yalign>
        <xpad>0</xpad>
        <ypad>0</ypad>
      </widget>
      <widget>
        <class>GtkLabel</class>
        <child_name>CList:title</child_name>
        <name>label4</name>
        <label>住所</label>
        <justify>GTK_JUSTIFY_CENTER</justify>
        <wrap>False</wrap>
        <xalign>0.5</xalign>
        <yalign>0.5</yalign>
        <xpad>0</xpad>
        <ypad>0</ypad>
      </widget>
    </widget>
    <widget>
      <class>GtkButton</class>
      <name>button2</name>
      <x>360</x>
      <y>272</y>
      <width>55</width>
      <height>22</height>
      <can_focus>True</can_focus>
      <signal>
        <name>clicked</name>
        <handler>send_event</handler>
        <data>Quit</data>
      </signal>
      <label>終わり</label>
    </widget>
  </widget>
</widget>
</GTK-Interface>
```

図 6.4　画面定義例 (1 続き)

この例では、PL データに直接関係のあるウィジェットは、<class>と</class>に囲まれた名前 (ウィジェットクラス) が、GtkEntry, GtkLabel, GtkCList のものであり、階層構造を構成するコンテナは、GtkFixed と GtkWindow である。といったようなことを人手で間違いなく拾い出すのは、このような単純な例であっても困難である。そこで、recdefgen というユーティリティを使うと、

```
list    {
    fixed1  {
        key {
            value   varchar(32);
        };
        clist1  {
            count   int;
            item    {
                value0  varchar(9);
                value1  varchar(10);
                value2  varchar(12);
                value3  varchar(10);
            }   [??];
            select  bool[??];
        };
    };
};
```

図 6.5 PL データ定義体骨格 (1)

このようなデータを吐き出す。このうち、??となっているものは、画面定義ファイルからは拾い出すことのできない、GtkCList の中の要素数 (行数) である。このような項目を編集して適当な大きさを与えてやれば、この図**??**の画面を表現する PL データ定義体が得られる。

glclient はこのデータ構造とウィジェットの包含構造を照合させながら、ウィジェットにデータを埋め込む。この時、照合させるものは、データ項目の階層構造であって順序ではない。そのため、名前の一致が取れる限り、順序は変更しても構わない。しかし、PL データ定義体から COBOL の COPY 句を生成させた場合には、データの順序が違ってはならない。このため、プログラム作成の手順としては、recrefgen で PL データ定義体骨格を生成したら、

- ??となっている項目に数値をあてはめる
- データ項目の順序を整理する

といったことを行ってから COPY 句を生成する。

C や Ruby のようにデータ項目 (メンバ) の参照に長形式メンバ名を使う言語の場合は、データ項目の順序は気にする必要がない。

6.5.3 glclient V2 プロトコルにおける PL データ

glclient の V2 プロトコルでは、PL データの構造とウィジェットの包含関係に依らず、ウィジェットにつけられた名前によって PL データの構造が構成される。このため、ウィジェットの名前に合わせて PL データ

を定義してやる必要がある。

V2 プロトコルで使う PL データの定義体も、画面定義体の XML を元に骨格を生成し、それを編集することによって作る。

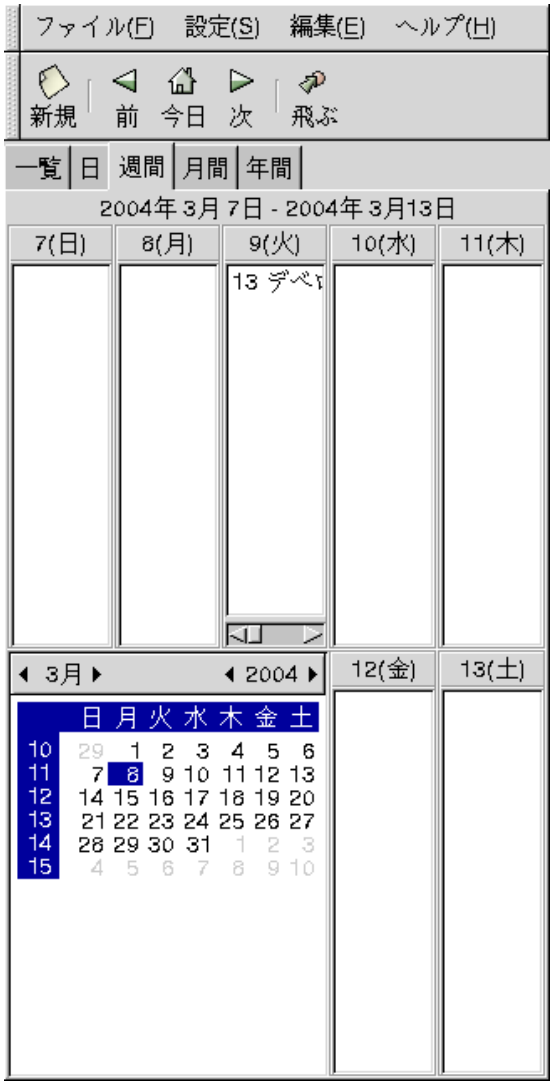

図 6.6 glclient の画面例 (2)

図 6.6 のような画面を定義した画面定義データは、図 6.7 のようになる。これは非常に大きいので、ここでは一部を挙げる。

```
<widget>
  <class>GtkVBox</class>
  <name>vbox16</name>
  <homogeneous>False</homogeneous>
  <spacing>0</spacing>
  <widget>
    <class>GtkLabel</class>
    <name>W.day[5]</name>
    <label>XXXXXX</label>
    <justify>GTK_JUSTIFY_CENTER</justify>
    <wrap>False</wrap>
    <xalign>0.5</xalign>
    <yalign>0.5</yalign>
    <xpad>0</xpad>
    <ypad>0</ypad>
    <child>
      <padding>0</padding>
      <expand>False</expand>
      <fill>False</fill>
    </child>
  </widget>
  <widget>
    <class>GtkHSeparator</class>
    <name>hseparator14</name>
    <child>
      <padding>0</padding>
      <expand>False</expand>
      <fill>False</fill>
    </child>
  </widget>
  <widget>
    <class>GtkScrolledWindow</class>
    <name>scrolledwindow11</name>
    <hscrollbar_policy>GTK_POLICY_AUTOMATIC</hscrollbar_policy>
    <vscrollbar_policy>GTK_POLICY_AUTOMATIC</vscrollbar_policy>
    <hupdate_policy>GTK_UPDATE_CONTINUOUS</hupdate_policy>
    <vupdate_policy>GTK_UPDATE_CONTINUOUS</vupdate_policy>
    <child>
      <padding>0</padding>
      <expand>True</expand>
      <fill>True</fill>
    </child>
    <widget>
      <class>GtkViewport</class>
      <name>viewport7</name>
      <shadow_type>GTK_SHADOW_IN</shadow_type>
```

図 6.7　画面定義例 (2)

```
            <widget>
              <class>GtkList</class>
              <name>W.apo[5]</name>
              <selection_mode>GTK_SELECTION_SINGLE</selection_mode>
            </widget>
          </widget>
        </widget>
      </widget>
    </widget>
```

図 6.8 画面定義例 (2 続き)

この例では、GtkLabel, GtkList の名前が、それぞれ W.day[5], W.apo[5] となっている。このようなものが、W.day[0]〜W.day[6], W.apo[0]〜W.apo[6] のようになっている。これらの項目は、図 6.9 のように定義されているとデータ参照がしやすい。ここで、recgen.rb を使ってやることにより、この部分は、図 6.10 のように生成される。ここで、apo の中は GtkList の中身であるので、このような構造になっている。この?? に適当な数値を入れてやれば、PL データ定義体は完成される。

V2 の定義はプログラマがウィジェットにつけた名前のみからデータの照合がされ、ウィジェットの包含関係には依存しない。このため、画面定義の自由度が高い。

```
W   {
    apo {
        count   int;
        item    varchar(64)[24];
        select  bool[24];
    }[7];
    day {
        value   varchar(6);
    }[7];
};
```

図 6.9 期待される PL データ定義体 (2) 抜粋

```
W   {
    apo {
        item    varchar(??)[??];
        count   int;
        select  bool[??];
    }[7];
    day {
        value   varchar(6);
    }[7];
};
```

図 6.10　生成された PL データ定義体骨格 (2) 抜粋

6.6 バッチ定義体

6.6.1 バッチ定義体で宣言するもの

バッチ定義体では、以下のものを定義する。

1. **定義体の名前**
2. **実行ハンドラ**
3. **プログラム名**
4. **デフォルト定義 (可変長配列、可変長文字列のための宣言)**
5. **実行ディレクトリの宣言**
6. **処理データの定義**
7. **データベースの宣言**

詳細の文法は、付録 C.3 に挙げる。

以下に詳細について説明するが、多くの点で LD 定義体と共通の部分がある。特に説明していない点については、6.4 の解説を参照のこと。

6.6.2 定義体の名前

バッチ定義体の名前を書く。この名前は、6.3.9 で説明した宣言で使われる。

ただし、現在のところ **MONTSUQI** がバッチ定義体を検索する時には、定義体のファイル名を使うので、ファイル名の拡張子を除いた部分とバッチ定義体の名前は一致していなければならない。バッチ定義体の拡張子は.bd である。

6.6.3 bind 定義 (処理モジュールの定義)

バッチで使用されるアプリケーションモジュールと実行ハンドラの関連付けを決定する。すなわち、どのアプリケーションモジュールがどの実行ハンドラで制御するかを定義する。

6.6.4 例

以下は定義の例である。

```
name    sample;
arraysize   20;
textsize    200;
bind    "SAMPLEB"       "OpenCOBOL";
bind    "say"           "OpenCOBOL";
db  "ORCA"  {
    metadb;
    adrs;
};
```

このバッチ定義体には、概略以下のようなことが記述されている。

- このバッチ定義体の名前は ‘sample’ である
-
  - SAMPLEB というプログラムは OpenCOBOL というハンドラを使う
  - say というプログラムは OpenCOBOL というハンドラを使う
- 可変長配列については、固定長配列しか許さない言語では要素数が 20 としてデータ変換する
- 可変長文字列については、固定長文字列しか許さない言語では文字列長が 200 としてデータ変換する
- データベースは、ORCA というデータベースグループにある metadb と adrs を使う

6.7 データベース公開定義体

6.7.1 データベース公開定義体で宣言するもの

データベース公開定義体では、以下のものを定義する。

1. **定義体の名前**
2. **デフォルト定義 (可変長配列、可変長文字列のための宣言)**
3. **実行ディレクトリの宣言**
4. **データベースの宣言**

詳細の文法は、付録 C.4 に挙げる。

データベース公開定義体は、LD 定義体やバッチ定義体から、データベース公開に関係のない宣言を除いたものである。そのため、個別の項目の説明は共通となっている。

6.7.2 例

以下は定義の例である。

```
name    sample;

arraysize   20;
textsize    200;

db  "ORCA"  {
    metadb;
    adrs;
};

db  "shell" {
    shell;
};
```

この定義体には、概略以下のようなことが記述されている。

- このデータベース公開定義体の名前は 'sample' である
- 可変長配列については、固定長配列しか許さない言語では要素数が 20 としてデータ変換する
- 可変長文字列については、固定長文字列しか許さない言語では文字列長が 200 としてデータ変換する
- データベースは、
  - ORCA というデータベースグループの metadb と adrs
  - shell というデータベースグループの shell

  を使う

6.8　データベース定義体

6.8.1　データベース定義体で定義するもの

データベース定義体では、以下のものを定義する。

- **レコード内容**
- **プライマリキー**
- **参照する他のデータベース定義体**
- **データベースのアクセス方法**

によって構成されている。このうち必須な項目は、レコード内容定義である。
詳細の文法は、付録 C.5 に挙げる。

6.8.2　レコード内容

データベースを参照する時のレコードの定義を行う。この文法はデータ定義体と同じなので、詳しくはを参照のこと。

データベースをアクセスするスクリプト (たとえば SQL) 中のデータ参照では、階層構造をもったデータの参照が出来ないことが多いが、**MONTSUQI** のデータベース定義体では、スクリプトを処理する部分で処理を行っているため、擬似的に階層構造も持つことが出来るようになっている。以下に例を示す。

```
adrs    {
    name    char(40);
    tel     char(13);
    mail    {
        home    char(64); ※ 1
        office  char(64);
    };
    toll    number(5,1);
    weight  number(5,1);
    address char(80)[3]; ※ 2
    tv      char[], virtual; ※ 3

};
```

この例で、

※ 1　構造体メンバである。これをデータベース定義の中で参照するには、‘.’ で区切って表現する。すなわち、‘mail.home’ のように表される

※ 2　配列である。これは個々の要素を見る時には、‘address[1]’ のように表現する。配列全体は ‘address’ のように表現される。配列の添字は 1 から始まる。

※ 3　属性の指定である。属性に”virtual”とつけることにより、データ処理や COPY 句の生成は行われるが、テーブルの項目は作成しない。これは、スクリプト用インターフェイス変数をテーブルと関係なしに作成することが出来る。

6.8.3 プライマリキー

タプルにプライマリキー制約を与えるための要素を指定する。この宣言は dbgen で create を生成するために使われる。つまり、現在の **MONTSUQI** の場合は

- SQL を使う
- データベースの実体がリレーショナルデータベース
- 実体のテーブルが存在しているデータベース定義

の場合に有効である。この条件にあてはまらないデータベース定義には、この宣言は必要ではなく、単に無視される。

以下に例を示す。

```
primary {
    name;
};
```

複数の要素を連結してプライマリキーにする場合は、

```
primary {
    name, mail.home;
};
```

のように、‘,’ で区切って表現される。この例のように、構造体の要素も使用可能である。

6.8.4 参照する他のデータベース定義体

他のデータベースを参照する場合に指定する。

MONTSUQI のデータベース機能は、SQL を直接使うような自由度の高さを捨て、その代わりアプリケーションがデータベースの実装に直接依存しないように設計されている。データベース定義体の中で、この差異を吸収するような規則を記述してやることにより、仮想的なデータベースを実際のデータベースシステム上に実現しているのである。

このデータベース定義の時、レコード内容の定義やデータベースのアクセス方法の中で他のデータベースを参照する必要がある場合、この方法で参照するデータベースの名前を宣言する。これはたとえば、*join* を使って仮想的なテーブルを定義する場合、その元となるテーブルを参照するために使う。

6.8.5 データベースのアクセス方法

MONTSUQI ではデータベースのアクセス方法を**パス**と呼ぶ。パスはいくつかの**アクセス方法** (**操作**とも呼ぶ) から成る。

スクリプト

データベースをアクセスする手段を定義する。アクセス方法はスクリプトで記述するが、そのスクリプトがどのように解釈されるかは、データベースのハンドラの実装に依存する。

スクリプトの引数

データベースをアクセスする時の必要となる情報は、6.8.2 で定義されたレコードを通じて行われる。しかし、これでは何を設定するべきかが、アプリケーションプログラマにとって明示的ではない。また、余計な項目に値を与える等の問題も起こしかねない。そのため、アクセスする時に実際に使うデータ項目に制限を与えることが可能である。これは、プログラムのデータの授受の時に引数を使うかグローバル変数を使うかというのと同様である。従来の **MONTSUQI** には「グローバル変数を使う」方法しか提供されていなかったが、現在は「引数を使う」方法も提供されている。

この引数は、パス毎に、また操作毎に持つことができ、参照は**近い方が見える**ようになっている。

ハンドラクラス既定のアクセス方法

データベースハンドラのクラスによっては、クラスモジュールの中でアクセス方法が既に定義されているものがある。

たとえば、データベースハンドラのクラスが PostgreSQL の場合は、

- **DBSTART**
- **DBCOMMIT**
- **DBFETCH**
- **DBUPDATE**
- **DBDELETE**
- **DBINSERT**

が定義済みである。どのようなアクセス方法が既定であるか、あるいはその処理内容がどういったものであるかについては、それぞれのハンドラクラスのドキュメントに記述されている。

同じ名前のアクセス方法が定義された場合、後に書かれたものが有効になる。そのため、クラス既定のアクセス方法を変更したい場合は、あらためて定義すればそちらが有効になる。

6.8.6 例

最も単純な例

図 6.11 は定義したレコードとデータベーステーブルの内容が一致している、単純な定義例である。

```
adrs    {
    name    varchar(40);
    tel     varchar(13);
    mail    {
        home    varchar(64);
        office  varchar(64);
    };
```

図 6.11　最も単純な例

```
    toll    number(5,1);
    weight  number(5,1);
    address varchar(80)[3];
    info    varchar(2000);
};
primary {
    name;
};

path    tel {
    DBSELECT    {
        DECLARE adrs_tel_csr CURSOR FOR
        SELECT  *
            WHERE
                tel = :tel
        ;
    };
};

path    mail    {
    DBSELECT    {
        DECLARE adrs_mail_csr CURSOR FOR
        SELECT
                name,
                tel,
                mail.home,
                mail.office,
                toll,
                weight,
                address
            FROM adrs
            WHERE
                mail.home like :mail.home
            ORDER BY
                name
        ;
    };
    DBFETCH {
        FETCH FROM adrs_mail_csr
            INTO
                :name,
                :tel,
                :mail.home,
                :mail.office,
                :toll,
                :weight,
                :address
        ;
    };
```

図 6.12 最も単純な例 (続き)

```
    DBCLOSECURSOR   {
        CLOSE adrs_mail_csr;
    };
    DBUPDATE    {
        UPDATE adrs
            SET
                tel = :tel,
                weight = :weight,
                address = :address
            WHERE
                adrs.name = :name;
    };
    DBINSERT    {
        INSERT  INTO    adrs
            (
                name,
                mail.home,
                mail.office,
                toll,
                weight,
                address
            )
            VALUES  (
                :name,
                :mail.home,
                :mail.office,
                :toll,
                :weight,
                :address
            )
        ;
    };
    DBDELETE    {
        DELETE  FROM    adrs
            WHERE
                adrs.name = :name
        ;
    };
};
```

図 6.13　最も単純な例 (続き)

このデータベース定義体には、概略以下のようなことが記述されている。

- プライマリキーは name である。
  dbgen を使ってテーブル定義のための SQL を生成することが可能である。
- レコードの名前は adrs である。
- 最初に定義されたパスは tel であり、操作として DBSELECT が定義されている。ここではデータベース

にありがちの他の操作が定義されていないので、おそらくハンドラクラス既定のものが使われる予定である

- 次に定義されたパスは mail であり、操作として、
  - DBSELECT
  - DBFETCH
  - DBCLOSECURSOR
  - DBUPDATE
  - DBINSERT
  - DBDELETE

  が定義されている。

他のデータベースを参照している例

図 6.14 は他のデータベースも参照している例である。

```
virtual person  {
    userid          varchar(64);
    address_type    int;
    tel_type        int;
    mail_type       int;

    name    = person_name.name;
    yomi    = person_name.yomi;
    zip     = person_address.zip;
    address = person_address.address;
    tel     = person_tel.no;
    mail    = person_mail.address;
};

use person_name;
use person_address;
use person_tel;
use person_mail;

path    simple  {
    operation   DBSELECT    {
        DECLARE person_simple_csr CURSOR FOR
        SELECT
            person_name.name,
            person_name.yomi,
            person_address.zip,
            person_address.address,
            person_tel.no,
            person_mail.address
```

図 6.14 他のデータベースを参照している例

```
        FROM
            person_name,
            person_address,
            person_tel,
            person_mail
        WHERE
            person_name.id = :userid AND
            person_address.type = :address_type AND
            person_address.id = :userid AND
            person_tel.type = :tel_type AND
            person_tel.id = :userid AND
            person_mail.type = :mail_type AND
            person_mail.id = :userid
        ;
    };
    operation   DBFETCH {
        FETCH FROM person_simple_csr
            INTO
                :name,
                :yomi,
                :zip,
                :address,
                :tel,
                :mail
        ;
    };
};
```

図 6.15　他のデータベースを参照している例 (続き)

このデータベース定義体には、以下のようなことが記述されている。

- このデータベースのレコード全体は仮想的なものであって、テーブル実体を持たない。
  名前は person である。
- 参照しているデータベース定義は、
  - person_name
  - person_address
  - person_tel
  - person_mail

  である。
- simple というパスが定義されている。操作として、
  - DBSELECT
  - DBFETCH

  が定義されている

この定義の場合、データベースレコードのうち、name は、参照している外部データベースである

person_name の name が実体である。yomi, zip, address, tel, mail についても、外部のデータベースで定義されているデータが実体であるということを定義している。

操作に引数のある例

以下の定義は、引数を持った操作がある場合の例である。

```
person_photo    {
    userid  varchar(64);
    photo   object;
};

primary {
    userid;
};

path    simple  {
    operation   DBSELECT    {
        DECLARE person_photo_simple_csr CURSOR FOR
        SELECT
            person_photo.userid,
            person_photo.photo
        FROM
            person_photo
        WHERE
            person_photo.userid = :userid
        ;
    };
    operation   DBINSERT(
            userid  varchar(64),
            fname   varchar(255))   {
        INSERT  INTO    person_photo
            (
                userid,
                photo
            )
```

図 6.16 操作に引数のある例

この例で特別なのは、パス名が simple の DBINSERT である。この例では、DBINSERT を実行する時には、冒頭に定義されているレコードの person_photo ではなく、DBINSERT の定義で定義されている、userid と fname が使われる。

```
            VALUES  (
                :userid,
                lo_import(:fname)
            )
        ;
    };
    operation   DBDELETE(
            userid  varchar(64))    {
        DELETE  FROM    person_photo
            WHERE
                userid = :userid
        ;
    };
};
```

図 6.17　操作に引数のある例 (続き)

6.9 レコード定義言語

6.9.1 概要

MONTSUQI で扱うデータは、全てここで規定するデータ構造定義言語によって定義されたデータである。この記述により、モニタ内でのデータの扱いを決定し、また言語間のデータの互換性を維持する。

このレコード定義言語を用いて、各種レコード定義体は定義される。

6.9.2 レコード定義言語で宣言するもの

レコード定義言語では、以下のものを宣言する。

1. **レコードの名前**
2. **レコードを構成するメンバの名前**
3. **レコードを構成するメンバの型**
4. **レコードを構成するメンバの属性**

詳細の文法は、付録 C.6 を参照のこと。

6.9.3 レコードの名前

レコードの名前を定義する。

MONTSUQI では定義体を探すのにファイル名を使うので、レコードの名前は定義ファイルの名前 (から拡張子を除いたもの) と一致させておくべきである。

6.9.4 レコードを構成するメンバの名前

メンバの名前を定義する。

レコード定義では、単純な名前だけで定義を行うが、参照する場合はその名前に到達するためのパスも指定する必要がある。このパスのついたメンバ名は**長形式メンバ名**と呼ぶ。

メンバの名前はそのレコード定義内でのみスコープを持つ。つまり、他のレコード定義に同じ名前が現れても、それらは別々のものとして扱われるデータベース定義では、外部から参照しなければならないため、レコード名で修飾した形式で長形式メンバ名を使う。これを**完全修飾メンバ名**と呼ぶ。

6.9.5 レコードを構成するメンバの型

データの型を指定する。

bool　論理値を表現する

byte　固定長領域を表現する。中身に関してはシステムは何もしない

binary　可変長領域を表現する。中身に関してはシステムは何もしない

char　固定長文字列を表現する。長さは”(” ”)”の中の数字で指定し、省略した場合は 1 が指定されたものとみなす

varchar　内部固定長外部可変長文字列を表現する。この型は内部的には固定長文字列であるが、外部的には出来る限り可変長として扱うデータ形式である。最大長は”(” ”)”の中の数字で指定し、省略した場合

は 1 が指定されたものとみなし、固定長文字列として扱う場合にはこの長さの文字列として扱われる。SQL の varchar に対応したものである

float　浮動小数点数を表現する。実行環境によって内部表現は変わるが、多くのプラットホーム上では 64bit IEEE 形式である

int　単形式符号付き整数を表現する。実行環境によって内部表現は変わるが、多くのプラットホーム上では 32bit 整数である

number　固定長数字列を表現する。数字のみによる文字列である他は、char 型と同じである。定義には全体の桁数と小数点以下の桁数を指定する

text　可変長文字列を表現する。長さの指定はない。内部的にも外部的にも可変長である。

object　BLOB(Binary Large OBject) を表現する。この型は object id だけを扱うので、実体を操作するには他の API で行う必要がある

dbcode　varchar 同様の内部固定長外部可変長文字列であるが、コード変換を行わないもの。データベースのスクリプト (SQL 等) を格納するのに使う

構造体　内部構造を持つもの

配列は型の後に大きさを指定することによって宣言する。大きさが省略された場合は可変長になる。次元については特に制限はないが、データ変換を行う時に変換先の形式によって制限を受けることがある。可搬性を考えれば、3 次元以下にしておくのが望ましい。

6.9.6　レコードを構成するメンバの属性

メンバの属性を指定する。

現在有効で指定可能な属性は virtual のみである。この属性は、データベース定義体の中で有効で、テーブル実体を持たないメンバであることを意味する。

6.9.7　例

以下は定義の例である。

```
adrs    {
    name    varchar(40);
    tel     varchar(13);
    mail    {
        home    varchar(64);
        office  varchar(64); ※ 1
    };
    toll    number(5,1);
    weight  number(5,1);
    address varchar(80)[3];  ※ 2
    info    varchar(2000);
};
```

この定義を copygen で COBOL 形式に変換すると、

```
01  ADRS.
    02  NAME    PIC X(40).
    02  TEL PIC X(13).
    02  MAIL.
      03  HOME    PIC X(64).
      03  OFFICE  PIC X(64).
    02  TOLL    PIC S9(4)V9(1).
    02  WEIGHT  PIC S9(4)V9(1).
    02  ADDRESS
            PIC X(80)   OCCURS  3 TIMES.
    02  INFO    PIC X(2000).
```

のようになる。ただし、**MONTSUQI** 内部のデータ形式では COBOL で言う集団項目としての切り口は持たない。

この例の※ 1 を C で参照する場合には、

```
value = GetItemLongName(adrs,"mail.office");
```

のようになる。また※ 2 の 3 番目の要素を参照する場合には、

```
value = GetItemLongName(adrs,"address[2]");
```

のようにする。C の場合はこのように、長形式メンバ名を指定して要素を参照する。この他にも API はあるが、詳細は C についての解説を参照のこと。

第 7 章

COBOL

7.1 基本事項

7.1.1 データ形式

MONTSUQI のデータ規約は COBOL では以下のように変換される。

7.2 COBOL クラス

7.2.1 アプリケーションの呼び出し

COBOL ハンドラクラスは、アプリケーションを以下の意味を持った引数のあるサブルーチンとして呼び出す。

- *mcp*
  アプリケーションサーバの制御情報領域
- *spa*
  アプリケーション共通領域

定義言語での表現	COBOL での表現	意味
bool	X(1)	真 'T' 偽 'F'
byte(n)	X(n)	そのまま格納される
binary	X(n)	そのまま格納される
char(n)	X(n)	コード変換を伴う
varchar(n)	X(n)	コード変換を伴う
float	X(8)	対応する変換がないため、そのまま格納される
int	S9(9) USAGE BINARY	endian の変換がある
number(m,n)	S9(m-m)V9(n)	
text	X(n)	n は`textsize`
object	X(8)	そのまま格納される
dbcode	X(n)	そのまま格納される
構造体	集団項目	
[n]	OCCURS n	

- *link*
  セション共通領域
- *scr*
  プレゼンテーション情報

これを受ける側 (つまりアプリケーション) では、図 7.1 のように定義する。ここで現れる COPY 句の生成方法については、7.3.2 参照のこと。

```
       LINKAGE                 SECTION.
       COPY    MCPAREA.
       COPY    SPAAREA.
       COPY    LINKAREA.
       COPY    SCRAREA.
      ************************************************************************
       PROCEDURE           DIVISION   USING
           MCPAREA
           SPAAREA
           LINKAREA
           SCRAREA.
```

図 7.1 アプリケーション呼び出しのインターフェイス

7.3 アプリケーションの作成

7.3.1 基本ロジック

アプリケーションサーバ配下のアプリケーションは、MCPAREA の項目である MCP-WIDGET, MCP-EVENT, MCP-STATUS の内容によって、自分への要求の内容を知る。

それぞれの項目の意味は、以下のようになる。

- *MCP-STATUS*
  アプリケーションの状態。他のプログラムから制御が来た場合は 'LINK'、自分の出力した画面の処理なら 'PUTG'
- *MCP-WIDGET*
  イベント発生契機となったウィジェット名
- *MCP-EVENT*
  イベント識別の文字列

アプリケーションプログラムの先頭で、これらの項目の値によって処理を分枝するようにプログラムを書く。

多くの場合、MCP-WIDGET の意味するところと、MCP-EVENT の意味するところは、意味的に等価であることが多いので、実際には MCP-EVENT と MCP-STATUS の値によって処理を分枝するようにする。

具体的に、図 7.2 のようなコードを書く。この例のように、分枝後の実際の処理は PERFORM で呼び出すようにするのが良い。

```
       LINKAGE                 SECTION.
       COPY    MCPAREA.
       COPY    SPAAREA.
       COPY    LINKAREA.
       COPY    SCRAREA.
      ******************************************************************************
       PROCEDURE           DIVISION   USING
           MCPAREA
           SPAAREA
           LINKAREA
           SCRAREA.
       000-MAIN                SECTION.
           EVALUATE    MCP-STATUS      ALSO    MCP-EVENT
             WHEN     'LINK'           ALSO    ANY
               PERFORM 010-INIT
             WHEN     'PUTG'           ALSO   'OpenCalendar'
               PERFORM 210-OPEN-CALENDAR
             WHEN     'PUTG'           ALSO   'CloseCalendar'
               PERFORM 220-CLOSE-CALENDAR
             WHEN     'PUTG'           ALSO   'PutData'
               PERFORM 230-PUT-DATA
             WHEN     'PUTG'           ALSO   'Left'
               PERFORM 240-CLICK-LEFT
             WHEN     'PUTG'           ALSO   'Right'
               PERFORM 250-CLICK-RIGHT
             WHEN      OTHER
               PERFORM 290-OTHER
           END-EVALUATE.
      *
           EXIT    PROGRAM.
```

図 7.2 COBOL の入口処理

7.3.2 COPY 句の生成

アプリケーションサーバとのインターフェイスに使う、MCPAREA と LINKAREA については、システム単位 (システム定義体の単位) に作成し、SPAAREA と SCRAREA については LD 単位に作成する必要がある。これらの COPY 句は LINKAREA については linkage 宣言と、SPAAREA については data 宣言中の spa 宣言と、また SCAAREA については data 宣言中の window 宣言と正しく対応関係がなければならない。そのため、これらを直接コーディングすることは行わず、自動生成で生成するか、それを元に編集して作ることが望ましい。

これは copygen によって行うが、copygen の使い方の詳細については、付録 B.1 を参照することにして、ここではその基本について説明をする。

MCPAREA の生成

MCPAREA はシステム定義体の内容によって決まる。制御情報領域に関する定義体は存在せず、**MONTSUQI**

の内部で定義されている。そのため、MCPAREA を生成する時には、単に、

```
$ copygen -mcp
```

のように実行する。

SPAAREA の生成

図 7.3 のような定義から、図 7.4 のような COPY 句を生成するには、

```
spa {
    current     {
        user    varchar(64);
        group   varchar(64);
        year    number(4);
        month   number(2);
        day     number(2);
        hour    number(2);
        min     number(2);
    };
    selected    {
        user    varchar(64);
        cdate   int;
    };
    new     bool;
    keyl    {
        cdate   int;
    }[100];
    keylcount   int;
    keyw    {
        wday    int;
        cdate   int;
    }[100];
    keywcount   int;
    grouplist   varchar(64)[100];
};
```

図 7.3 アプリケーション共通領域の定義例

```
$ copygen -name SPAAREA -prefix SPA- <SPA のレコード定義体名>
```

のように実行する。copygen は処理結果を標準出力に出すので、適当なファイルにリダイレクトしてやる。COPY 句に使うことのできるファイル名については、OpenCOBOL の規約に合致していれば何であっても構わない。しかし、SPAAREA は LD 毎に異なることから、

- **LD 名**

```
01  SPAAREA.
    02  SPA-CURRENT.
      03  SPA-USER    PIC X(64).
      03  SPA-GROUP   PIC X(64).
      03  SPA-YEAR    PIC S9(4).
      03  SPA-MONTH   PIC S9(2).
      03  SPA-DAY PIC S9(2).
      03  SPA-HOUR    PIC S9(2).
      03  SPA-MIN PIC S9(2).
    02  SPA-SELECTED.
      03  SPA-USER    PIC X(64).
      03  SPA-CDATE   PIC S9(9)   BINARY.
    02  SPA-NEW PIC X.
    02  SPA-KEYL        OCCURS  100 TIMES.
      03  SPA-CDATE   PIC S9(9)   BINARY.
    02  SPA-KEYLCOUNT   PIC S9(9)   BINARY.
    02  SPA-KEYW        OCCURS  100 TIMES.
      03  SPA-WDAY    PIC S9(9)   BINARY.
      03  SPA-CDATE   PIC S9(9)   BINARY.
    02  SPA-KEYWCOUNT   PIC S9(9)   BINARY.
    02  SPA-GROUPLIST
            PIC X(64)   OCCURS  100 TIMES.
```

図 7.4 生成した SPAAREA

- **SPAAREA であること**

がわかるような名前にするのが望ましい。具体的には、

$$LD\text{名} + \text{“} - SPA''$$

といった形式の名前にしておくと混乱が少ない。

LINKAREA の生成

図 7.5 のような定義から、図 7.6 のような COPY 句を生成するには、

```
link    {
    linktext    char(200);
};
```

図 7.5 セション共通領域の定義例

LINKAREA を生成するには、

```
$ copygen -linkage -ld <LD 名>
```

```
01  LINKAREA.
    02  LINKDATA-REDEFINE.
      03  FILLER      PIC X(217).
    02  SCHEDULE    REDEFINES   LINKDATA-REDEFINE.
        04  LINKTEXT    PIC X(200).
```

図 7.6 生成した LINKAREA

のようにする。

LINKAREA はシステムで 1 つの定義なので、本来は LD が異なっても変化するものではない。しかし、arraysize, textsize が LD 毎の定義となっているので、**どの LD で使われる LINKAREA か**ということを意識して生成する必要がある。すなわち名称の混乱がないようにする必要がある。

SCRAREA の生成

LD 定義体に、

```
data  spa samplespa; window  list; ; ;
```

という記述があり、list の定義が、図 7.7 のようになっている場合に、
図 7.8 のような COPY 句を生成するには、

```
list  fixed1  key  value varchar(30); ; clist1  count int; item  value1 char(12); value2 char(12); value3 char(12);
value4 char(12);  [20]; select bool[20]; ; ; ;
```

図 7.7 プレゼンテーションデータの定義例

```
$ copygen -screen -name SCRAREA -wprefix -ld <LD 名>
```

のようにする。SCRAREA も SPAAREA 同様に LD 毎に異なるので、

- **LD 名**
- **SCRAREA であること**

がわかるような名前にするのが望ましい。これも、SPAAREA 同様に、

$$LD\ 名 + \text{“} - SCR''$$

といった形式の名前にしておくと混乱が少ない。

```
01  SCRAREA.
    02  SCREENDATA.
      03  LIST.
        04  LIST-FIXED1.
          05  LIST-KEY.
            06  LIST-VALUE  PIC X(30).
          05  LIST-CLIST1.
            06  LIST-COUNT  PIC S9(9)   BINARY.
            06  LIST-ITEM       OCCURS  20 TIMES.
              07  LIST-VALUE1 PIC X(12).
              07  LIST-VALUE2 PIC X(12).
              07  LIST-VALUE3 PIC X(12).
              07  LIST-VALUE4 PIC X(12).
            06  LIST-SELECT
                    PIC X       OCCURS  20 TIMES.
```

図 7.8 生成した SCRAREA

SCRAREA はこのままでも使えないことがないが、glclient の V1 プロトコルを使っている場合は、項目の階層に画面レコード定義の情報を直接反映してしまうので、画面の構成によっては余分な集団項目を生成しがちである。また、項目名が画面レコード定義と同じものを使ってしまうため、名前の重複が発生しやすい。さらに、いくつかの画面を束ねて LD を作るため、複数の画面での項目名までもが重複する危険がある。COBOL の文法的には重複した項目は OF で修飾するようにすれば問題ないのではあるが、かなりソースが繁雑になってしまう。

そこで、この生成された SCRAREA を使いやすい形に修正を行う。具体的には、

- **項目名の変更**
- **不要な階層の除去**

を行う。

MONTSUQI の COBOL 用データ変換は、項目のオフセットのみを意識してデータを並べるので、項目名等は任意に変更して構わない。ただし、項目のオフセットは絶対に変更してはならない。そのため、

- **基本項目の削除**
- **OCCURS 数の変更**
- **項目の順序変更**

等は行ってはならない。

この方法は、一度に全部を一人で作成する時には都合が良いが、現実には同じ LD のアプリケーションを複数の開発者によって開発されることも少なくない。そのような時には、分割して生成を行う。分割して生成するには、まず

```
$ copygen -screen
```

として、レコード定義の最初の 2 行を出力し、それに

```
$ copygen -screen <画面レコード定義体>
```

として出力したものを連結して行く。もちろん結合前に項目名やレベル番号等の変更を行っておく。この時の留意点は一度に作成する場合と同じである。また、連結の順序は、**LD 定義体の window 定義と同じ**でなくてはならない。

PATH の生成

COBOL アプリケーションからのパス指定は、数値の組によって行う。この数値の組の具体的な内容は、LD 定義体の内容によって決まる。この具体的な値については、**パス情報**として COPY 句に作る。

このコピー句を生成するには、

```
$ copygen -dbpath -ld <LD 名>
```

のようにする。これも他の LD 毎に作る COPY 句と同じような扱いにしておくのが間違いがない。

7.4 COBOL 用 API

7.4.1 共通事項

OpenCOBOL のアプリケーションは、MCPSUB というサブルーチンを呼び出すことによって、**MONTSUQI** のサービスを利用することができる。処理の内容によっては、MCPAREA の直接操作によって処理を行うことができるものもあるが、互換性や安全性に問題があるので、直接操作をせずに MCPSUB を呼び出すことによって操作するのが望ましい。

7.4.2 データベース機能の利用

MCPSUB のデータベース機能を使うには、

1. **必要ならキー値をレコード上の項目に設定する**
2. **必要なら MCP-PATH にパス情報を設定する**
3. **MCP-FUNC に機能名を設定する**
4. **MCPSUB を呼び出す**
5. **必要なら MCP-RC を見る**

という手順で行う。MCPSUB の呼び出し方は、

```
          CALL    'MCPSUB'      USING
              MCPAREA
              <レコード項目名>
```

のように行う。COBOL の引数解釈の関係から、レコードを必要としない機能の場合でも、ダミーのものを指定しておく必要がある。

MCP-RC は S9(9) BINARY の変数であり、その値の意味は

0	正常終了
正	回復可能エラー
負	回復不可能エラー

表 7.1 MCP-RC の値

である。

7.4.3 MCPSUB のオンライン機能

MCPSUB のオンライン機能には、以下の機能がある。

PUTWINDOW
画面を表示する

この機能を使う場合は、

1. MCP-FUNC に機能名 (PUTWINDOW) を設定する
2. MCP-PUTYPE に出力タイプを設定する
3. MCP-WINDOW にウィンドウ名を設定する
4. MCPSUB を呼び出す
5. 必要なら MCP-RC を見る

という手順で行う。

■ 名前

PUTWINDOW - 画面出力を行う

■ パラメータ

- *MCP-PUTTYPE*
  出力タイプ
- *MCP-WINDOW*
  画面名

■ 説明

出力タイプは表 3.1 に示す。

第 8 章

データベースハンドラ

8.1 基本事項

8.1.1 概要

MONTSUQI のデータベースアクセスは、言語依存のデータベースアクセスルーチンからエントリし、データベースアクセスハンドラを介して行われる。このハンドラは、本システムのモジュールとして実装され、DB group の定義の時に指定する。

8.2 PostgreSQL ハンドラ

8.2.1 概要

本ハンドラは、PostgreSQL を、libpq を介して TCP/IP 接続によってアクセスを行う。現在のところ、接続するユーザは、DB group 定義に書かれたユーザか、無名ユーザである。

8.2.2 基本機能

テーブル操作については、以下の機能が標準で提供されている。

- **DBFETCH**
  レコード (タプル) の全カラム取得を行う。この時のカーソル名は、

  <テーブル名> + '_' + **<パス名>** + '_' + "csr"

  であることを仮定している。そのため、標準の DBFETCH を使う場合は、selection を行うときのカーソルはこの名前でなくてはならない。
- **DBUPDATE**
  タプルの全カラム更新である。位置指定はプライマリキーを用いる。
- **DBDELETE**
  当該タプルの削除である。位置指定はプライマリキーを用いる。
- **DBINSERT**
  タプルの全カラム指定の挿入である。

基本機能の DBFETCH を使う場合、あらかじめカーソルをセットしなくてはならない。カーソルをセットする機能の名前としては、DBSELECT が推奨されている。DBSELECT については、処理の具体的な内容を定義して

いないので、必ず定義しなくてはならない。

8.2.3　機能定義手続きのスクリプトの文法

既定の機能を使わない場合は、機能をスクリプトで記述しなくてはならない。PostgreSQL をアクセスする際のスクリプトは、PostgreSQL に規定された SQL 文である。

タプルを受け取る命令 (FETCH 等) の場合、INTO の後の変数には ':*' と指定することにより、タプルの全項目をレコード領域に取り込むことが出来る。

8.3 Shell ハンドラ

8.3.1 概要

本ハンドラは、shell を起動する。現在のところ、実行するユーザは、aps が起動されたユーザと同じである。

8.3.2 基本機能

本ハンドラには、既定の基本機能はない。

8.3.3 機能定義手続きのスクリプトの文法

Shel をアクセスする際のスクリプトは、`/bin/sh` に既定された shell の文である。
現在のところ、このハンドラではデータの受け取りは記述出来ない。

8.4　System ハンドラ

8.4.1　概要

本ハンドラは、システムの情報を取得する。

8.4.2　基本機能

テーブル操作については、以下の機能が標準で提供されている。

- **DBSELECT**
- **DBFETCH**

8.4.3　機能定義手続きのスクリプトの文法

本ハンドラではスクリプトを記述する必要はない。

8.4.4　補足

本ハンドラでは、現在のところデータを取得するには、取得用のレコードを定義してやるだけである。必要な情報の名前をメンバ名として定義したレコードを用意して DBSELECT して DBFETCH してやれば、取得できる。

現在、取得可能なシステムの情報は、PL クライアントの接続元ホスト名だけであり、情報の名前は host である。型や長さは任意であり、それに応じて適当なデータ変換が行われる。

付録 A

コアプログラムの説明

A.1 dbs

■ **名前**

dbs - 外部システムが MONTSUQI 配下のデータベースをアクセスするためのインターフェイス。

■ **書式**

dbs <OPTIONS> <DBD 名>

■ **説明**

dbs は外部のシステムが MONTSUQI 配下のデータベースにアクセスするためのインターフェイスプロセスである。デーモンとして動かす。

■ **オプション**

-port

待機用ポート。デフォルトは 8013。

-back

接続待ちキュー長。デフォルトは 5。

-base

環境定義体に定義された base を上書きするのに使う

-dir

環境定義体のファイル名を指定する。デフォルトは./directory

-record

レコード定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-ddir

DB 定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-dbhost

データベース稼働ホスト名デフォルトは-dir で指定したファイルの内容に従う

-dbport

データベース待機ポート番号デフォルトは-dir で指定したファイルの内容に従う

-db

データベース名デフォルトは-dir で指定したファイルの内容に従う

-dbuser

データベースユーザ名デフォルトは-dir で指定したファイルの内容に従う

-dbpass

データベースのパスワードデフォルトは-dir で指定したファイルの内容に従う

-auth

認証サーバの指定を URL 形式で行う。デフォルトは `glauth://localhost:8001`。これは glauth プロトコルで認証を行い、localhost のポート番号 8001 に要求を出すという意味である。

-nocheck

`dbredirector` の起動チェックを行わない

-noredirect

`dbredirector` が起動していても移送を行わない

■ **例**

```
dbs sample &
```

pgserver を起動する。定義体は、起動ディレクトリから取得する。

■ **バグ**

■ **参照**

dbslib.rb, DBIO

■ **注意**

dbs はクライアントが接続する毎にサブプロセスを生成する。

A.2 glserver

■ 名前

`glserver` - GUI プレゼンテーションサーバ

■ 書式

```
glserver <OPTIONS>
```

■ 説明

`glserver` は glclient を接続するためのプレゼンテーションサーバである。デーモンとして動かす。

■ オプション

-port
待機用ポート番号。デフォルトは 8000。

-back
接続待ちキュー長。デフォルトは 5。

-screen
画面格納ディレクトリ名。デフォルトは起動したディレクトリ。

-record
レコード定義体のあるディレクトリ。デフォルトは起動したディレクトリ。

-auth
認証サーバの指定を URL 形式で行う。デフォルトは `glauth://localhost:8001`。これは glauth プロトコルで認証を行い、localhost のポート番号 8001 に要求を出すという意味である。

-ssl
glclient との通信に SSL を使う。SSL を使用しない場合は、SSL 用のパラメータである`-key`, `-cert`, `-CApath`, `-CAfile`, `-verifypeer` の指定は意味を持たないデフォルトは SSL を使用しない

-key
SSL の秘密鍵ファイル名。鍵は pem 形式デフォルトは秘密鍵ファイルを指定しない (証明書と同じファイルになっている)

-cert
証明書ファイル名。証明書は pem 形式。デフォルトは証明書を指定しない。`-key` も`-cert` も指定されていない場合はエラーになる

-CApath
CA 証明書へのパス。デフォルトは CA 証明書パスを使用しない。

-CAfile
CA 証明書ファイルへのパス。デフォルトは CA 証明書ファイルを指定しない。-CApath も-CAfile も指定されていない場合は、CA 証明書による機能を使わない。

-verifypeer
クライアント証明書の検証を行う。デフォルトは検証を行わない。

-panda
wfc の待機ポート。デフォルトは `localhost:9000`。

■ 例

```
glserver &
```

glserver を起動する。画面およびレコードの定義体は、起動ディレクトリから取得する。

■ **バグ**

■ **参照**

■ **注意**

glserver は glclient が接続する毎にサブプロセスを生成する。

SSL を使う場合、正規の証明書が必須である (自己証明は不可)。

SSL を使った場合で、かつクライアント証明書の検証が行われる場合 (`-verifypeer` が指定された場合)、glauth を使った認証は無効になる。

A.3 glclient

■ 名前

`glclient` - GUI クライアント

■ 書式

`glclient <OPTIONS>` **アプリケーション指定**

■ 説明

`glclient` は glserver に接続する GUI 端末クライアントである。

■ オプション

-port

glserver の待機ポート。デフォルトは `localost:8000`。

-cache

画面キャッシュ格納ディレクトリ。デフォルトは `cache`。

-style

追加スタイルファイル格納ディレクトリ。デフォルトは空。すなわち追加スタイルファイルを使用しない。この場合は、`gltermrc` に記述したスタイルのみを使用する。

-user

glserver に login するユーザ名。デフォルトは環境変数 `USER` の値。

-pass

glserver に login するパスワード。デフォルトは空文字。

-v1

V1 プロトコルを使う。デフォルトは使用する。

-v2

V2 プロトコルを使う。デフォルトは使用しない。

-mlog

実行時ログの取得を行う。デフォルトは取得しない。

-ssl

glclient との通信に SSL を使う。SSL を使用しない場合は、SSL 用のパラメータである`-key`, `-cert`, `-CApath`, `-CAfile`, `-verifypeer` の指定は意味を持たないデフォルトは SSL を使用しない

-key

SSL の秘密鍵ファイル名。鍵は pem 形式デフォルトは秘密鍵ファイルを指定しない (証明書と同じファイルになっている)

-cert

証明書ファイル名。証明書は pem 形式。デフォルトは証明書を指定しない。`-key` も`-cert` も指定されていない場合はエラーになる

-CApath

CA 証明書へのパス。デフォルトは CA 証明書パスを使用しない。

-CAfile

CA 証明書ファイルへのパス。デフォルトは CA 証明書ファイルを指定しない。-CApath も-CAfile

も指定されていない場合は、CA 証明書による機能を使わない。

-verifypeer

クライアント証明書の検証を行う。デフォルトは検証を行わない。

アプリケーション指定は、glserver 上で動くアプリケーションモジュール名と引数:で区切って指定する。引数の解釈は、アプリケーションモジュール依存である。MONTSUQI を使用する場合は、`panda:<LD名>`という指定になる。

■ **例**

```
glclient -user ogochan -pass "ain@Lkad" panda:sample
```

glclient を起動する。ユーザは ‘ogochan’。アプリケーション ‘panda’ に引数 ‘sample’ を持って接続する。

■ **バグ**

■ **参照**

glserver

■ **注意**

A.4 htserver

■ 名前

`htserver` - web 用プレゼンテーションサーバ

■ 書式

```
htserver <OPTIONS>
```

■ 説明

`htserver` は `mon` と協調して web browser を接続するためのプレゼンテーションサーバである。デーモンとして動かす。

■ オプション

-port
待機用ポート。デフォルトは 8000。
-back
接続待ちキュー長。デフォルトは 5。
-record
レコード定義体のあるディレクトリ。デフォルトは起動したディレクトリ。
-expire
セションを無捜査時間の経過で無効にするまでの時間 (秒)。デフォルトは 300 秒。
-panda
wfc の待機ポート。デフォルトは `localhost:9000`。

■ 例

```
htserver &
```

htserver を起動する。レコードの定義体は、起動ディレクトリから取得する。

■ バグ

■ 参照

mon, wfc, glserver

■ 注意

htserver は web browserglclient がセションを開設する毎にサブプロセスを生成する。

A.5　mon

■ **名前**

`mon` - web 用プレゼンテーションサーバの cgi インターフェイス

■ **書式**

```
mon <OPTIONS>
```

■ **説明**

`mon` は `htserver` と協調して web browser を接続するためのプレゼンテーションサーバである。CGI として動かす。

■ **オプション**

-server
サーバのポート。デフォルトは `localhost:8010`。

-screen
画面格納ディレクトリデフォルトは CGI が起動されたディレクトリ。

-get
action を GET で処理する。デフォルトは POST で処理する

-dump
デバッグ用に変数のダンプを行う。デフォルトは OFF。

-cookie
セション変数の保持を cookie で行う。デフォルトは hidden で HTML に埋め込む

-command
セション開設時に htserver に渡すコマンドライン。デフォルトはなし。

■ **例**

```
./mon -command "panda:demo1" -server "tencha:8010"
```

mon を起動する。レコードの定義体は、起動ディレクトリから取得する。htserver は tencha というホストの 8010 ポートで待機している。
実際に mon を使う場合は、このコマンドラインを格納したシェルスクリプトを作り、それを使うようにする。

■ **バグ**

web browser で「戻る」や「再読込」等の処理を行うと、セション変数が失われてセションを忘れてしまう。

■ **参照**

htserver, wfc, glserver

■ **注意**

A.6 pgserver

■ **名前**

pgserver - アプリケーションと直接会話するためのプレゼンテーションサーバ

■ **書式**

```
pgserver <OPTIONS>
```

■ **説明**

pgserver はアプリケーションを直接接続するためのプレゼンテーションサーバである。デーモンとして動かす。

■ **オプション**

-port
待機用ポート。デフォルトは 8011。

-back
接続待ちキュー長。デフォルトは 5。

-record
レコード定義体のあるディレクトリ。デフォルトは起動したディレクトリ。

-auth
認証サーバの指定を URL 形式で行う。デフォルトは `glauth://localhost:8001`。これは glauth プロトコルで認証を行い、localhost のポート番号 8001 に要求を出すという意味である。

-panda
wfc の待機ポート。デフォルトは `localhost:9000`。

■ **例**

```
pgserver &
```

pgserver を起動する。レコードの定義体は、起動ディレクトリから取得する。

■ **バグ**

■ **参照**

monlib.rb, PGIO

■ **注意**

pgserver はクライアントが接続する毎にサブプロセスを生成する。

A.7　glauth

■ **名前**

glauth - glserver 用認証サーバ

■ **書式**

```
glauth <OPTIONS>
```

■ **説明**

glauth は glclient が glserver に接続する際の、認証用サーバである。デーモンとして動かす。

■ **オプション**

-port

待機用ポート。デフォルトは 8001。

-back

接続待ちキュー長。デフォルトは 5。

-password

パスワードファイル名。デフォルトは./passwd。

■ **例**

```
glauth &
```

■ **バグ**

■ **参照**

■ **注意**

パスワードファイルは MD5 で暗号化されている。

A.8 wfc

■ 名前

wfc - ワークフローコントローラ

■ 書式

```
wfc <OPTIONS>
```

■ 説明

wfc はワークフローコントローラの本体である。デーモンとして動かす。

■ オプション

-port
プレゼンテーションサーバからの接続待機用ポート。デフォルトは 9000。

-apsport
アプリケーションサーバからの接続待機用ポート。デフォルトは 9001。

-back
接続待ちキュー長。デフォルトは 5。

-dir
環境定義体のファイル名を指定する。デフォルトは./directory

-base
環境定義体に定義された base を上書きするのに使う

-record
レコード定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-ddir
LD 定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-retry
トランザクションを再試行する回数の上限。この回数を超えると、トランザクションの再試行を諦めてセションを切断する。デフォルトは無制限。

■ 例

```
wfc &
```

ワークフローコントローラを起動する。

■ バグ

■ 参照

aps

■ 注意

ワークフローコントローラはマルチスレッドのプログラムであり、Ver 2.6 以前の Linux 上で動かすとアプリケーションサーバやプレゼンテーションサーバが接続する毎に ps(1) で見た時のプロセス数は増える。

A.9　aps

■ **名前**

aps - アプリケーションサーバ

■ **書式**

aps <OPTIONS>　LD 名

■ **説明**

aps はアプリケーションサーバである。デーモンとして動かす。

■ **オプション**

-wfcport

ワークフローコントローラが接続待機しているポート。デフォルトは-dir で指定したファイルの内容に従う

-base

環境定義体に定義された base を上書きするのに使う

-record

レコード定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-lddir

LD 定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-dir

環境定義体のファイル名を指定する。デフォルトは./directory

-path

アプリケーション等のロードパス。デフォルトは OpenCOBOL の場合は環境変数 COB_LIBRARY_PATH に従う

-dbhost

データベース稼働ホスト名デフォルトは-dir で指定したファイルの内容に従う

-dbport

データベース待機ポート番号デフォルトは-dir で指定したファイルの内容に従う

-db

データベース名デフォルトは-dir で指定したファイルの内容に従う

-dbuser

データベースユーザ名デフォルトは-dir で指定したファイルの内容に従う

-dbpass

データベースのパスワードデフォルトは-dir で指定したファイルの内容に従う

-maxtran

起動したアプリケーションサーバで処理するトランザクション数。これを超えるとアプリケーションサーハ終了する。デフォルトは無制限に処理する。

-cache

キャッシュを行うセション数。デフォルトは-dir で指定したファイルの内容に従う

-nocheck

dbredirector の起動チェックを行わない

-noredirect

`dbredirector` が起動していても移送を行わない

■ **例**

```
aps sample &
```

sample という名前の LD に対応したアプリケーションサーバを起動する。

■ **バグ**

■ **参照**

wfc

■ **注意**

アプリケーションサーバがアプリケーションモジュールを呼び出すためには、様々なアプローチが取られる。どのようなアプローチでどのような制約事項があるかは、それぞれの言語ハンドラおよびデータベースハンドラの説明を見ること。

wfc が終了すると、aps は正常終了に終了する。これが現在のところ aps を正常終了させる唯一の方法である。

A.10 aps

■ **名前**

dbstub - バッチ用アプリケーションサーバ

■ **書式**

dbstub <OPTIONS> **<モジュール名>**

■ **説明**

dbstub はバッチ用アプリケーションサーバである。

■ **オプション**

-dir
環境定義体のファイル名を指定する。デフォルトは./directory

-base
環境定義体に定義された base を上書きするのに使う

-record
レコード定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-ddir
バッチ定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-parameter
アプリケーションに渡すパラメータ。デフォルトは空文字列。

-path
アプリケーション等のロードパス。デフォルトは OpenCOBOL の場合は環境変数 COB_LIBRARY_PATH に従う

-host
データベース稼働ホスト名デフォルトは-dir で指定したファイルの内容に従う

-port
データベース待機ポート番号デフォルトは-dir で指定したファイルの内容に従う

-bd
バッチ定義体名デフォルトはないので、必ず指定しなくてはならない

-db
データベース名デフォルトは-dir で指定したファイルの内容に従う

-user
データベースユーザ名デフォルトは-dir で指定したファイルの内容に従う

-pass
データベースのパスワードデフォルトは-dir で指定したファイルの内容に従う

-nocheck
dbredirector の起動チェックを行わない

-noredirect
dbredirector が起動していても移送を行わない

■ **例**

```
dbstub -bd sample SAMPLEB &
```

sample という名前の BD に従い、アプリケーションサーバを起動する。アプリケーションモジュールとして SAMPLEB を指定する

■ **バグ**

■ **参照**

aps

■ **注意**

アプリケーションサーバがアプリケーションモジュールを呼び出すためには、様々なアプローチが取られる。どのようなアプローチでどのような制約事項があるかは、それぞれの言語ハンドラおよびデータベースハンドラの説明を見ること。

A.11 dbredirector

■ **名前**

dbredirector - データベースへの要求をファイルまたは他のデータベースにリダイレクトする

■ **書式**

dbredirector <OPTIONS> <DB **グループ名**>

■ **説明**

dbredirector はデータベースへの要求をリダイレクトする。デーモンとして動かす。

■ **オプション**

-port
アプリケーションサーバからの接続待機用ポート番号。デフォルトは-dir で指定したファイルの内容に従う
-back
接続待ちキュー長。デフォルトは 5。
-dir
環境定義体のファイル名を指定する。デフォルトは./directory
-base
環境定義体に定義された base を上書きするのに使う
-record
レコード定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う
-ddir
定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う
-host
データベース稼働ホスト名デフォルトは-dir で指定したファイルの内容に従う
-port
データベース待機ポート番号デフォルトは-dir で指定したファイルの内容に従う
-db
データベース名デフォルトは-dir で指定したファイルの内容に従う
-user
データベースユーザ名デフォルトは-dir で指定したファイルの内容に従う
-pass
データベースのパスワードデフォルトは-dir で指定したファイルの内容に従う

■ **例**

```
dbredirector log &
```

log という名前の DB グループに対応した dbredirector を起動する。

■ **バグ**

■ **参照**

monitor

■ **注意**

付録 B

ユーティリティコマンドの説明

B.1 copygen

■ **名前**

copygen - 各種定義体から COBOL 用 COPY 句を生成する結果は標準出力に出る。

■ **書式**

```
copygen <OPTIONS>
```

■ **説明**

copygen はレコード定義および LD 定義を元に MONTSUQI の COBOL で書かれるアプリケーションプログラムの各所で必要となる COPY 句を生成するコマンドである。

■ **オプション**

-ldw

(dotCOBOL のみ) LD 定義から MCPMAIN 用出力ファイル定義用 COPY 句を生成するレコードの中身を制御フィールド以外は FILLER のみにして、領域のみを確保する。LD から生成する時に有効となる。デフォルトは OFF。

-ldr

(dotCOBOL のみ) LD 定義から MCPMAIN 用入力ファイル定義用 COPY 句を生成する。レコードの中身を制御フィールドと SPA 以外は FILLER のみにして、領域のみを確保する。LD 定義体から生成する時に有効となる。デフォルトは OFF。

-spa

LD 定義から SPA 領域定義用 COPY 句を生成する。デフォルトは OFF。

-linkage

実行環境定義から LINKAGE 領域定義用 COPY 句を生成する。デフォルトは OFF。

-screen

LD 定義から画面領域用 COPY 句を生成する。デフォルトは OFF。

-db

(dotCOBOL のみ) LD 定義またはバッチ定義体から MCPSUB を CALL する時の第 2 パラメータ (結果格納用領域) の MCPSUB 側 COPY 句を生成する。デフォルトは OFF。

-dbpath

LD 定義またはバッチ定義体から MCPSUB を CALL する時の MCP-PATH にセットする、DB のパス名テーブルの COPY 句を生成する。デフォルトは OFF。

-dbrec
LD 定義またはバッチ定義体から MCPSUB を CALL する時の第 2 パラメータ (結果格納用領域) の COPY 句を生成する。デフォルトは OFF。
-dbcomm
(dotCOBOL のみ) LD 定義またはバッチ定義体から MCPSUB と DB サーバとの通信領域の COPY 句を生成する。デフォルトは OFF。
-mcp
実行環境定義から MCPAREA を生成する。デフォルトは OFF。
-textsize
不定長文字列を固定長にする時の長さ。レコード定義から生成する時に有効となる (それ以外だと実行環境定義の値を使う)。デフォルトは 100
-arraysize
不定長配列を固定長にする時の要素数。レコード定義から生成する時に有効となる (それ以外だと実行環境定義の値を使う)。デフォルトは 10
-prefix
項目名の前に付加する文字列。デフォルトは何も付加しない
-wprefix
画面レコードの項目名の前にウィンドウ名を付加する。デフォルトは何も付加しない
-name
01 レベルの項目名 (レコードの名前) を指定する。デフォルトは定義体で定義した名前と同じとなる
-filler
レコードの中身を FILLER のみにし、領域のみを確保する。デフォルトは OFF
-noconv
項目名をレコード定義と同じにし、大文字に加工しない。デフォルトは OFF
-dir
環境定義体のファイル名を指定する。デフォルトは`./directory`
-record
レコード定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う
-lddir
LD 定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う
-bddir
バッチ定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う
-ld
LD 定義体の名前。デフォルトはないので、LD 定義体の必要なオプションを指定したら必ず指定する
-bd
バッチ定義体の名前。デフォルトはないので、バッチ定義体の必要なオプションを指定したら必ず指定する

■ **例**

```
copygen -ctrl -spa -screen -prefix LDR- -name LDR demo
```

demo.ld を元に生成する。レコード名は LDR、接頭語として LDR-を付加し、制御領域、SPA 領域、画面領域を出力する。

```
copygen -filler -name SPAAREAR demospa.rec
```

demspa.rec を元に生成する。レコード名は SPAAREA、中身のデータ項目を個々に出力せず、FILLER によって大きさのみ確保する。

■ **バグ**

■ **参照**

recdefgen.rb

■ **注意**

B.2　copygen

■ **名前**

dbgen - DB 定義体から create 文を生成する

■ **書式**

dbgen <OPTIONS> <DB **定義体名**>

■ **説明**

dbgen は DB 定義体から create table の SQL 文を生成するコマンドである。結果は標準出力に出る。

■ **オプション**

-create

DB 定義体から create table の SQL 文を生成する。デフォルトは OFF。

-textsize

不定長文字列を固定長にする時の長さ。レコード定義から生成する時に有効となる (それ以外だと実行環境定義の値を使う)。デフォルトは 100

-arraysize

不定長配列を固定長にする時の要素数。レコード定義から生成する時に有効となる (それ以外だと実行環境定義の値を使う)。デフォルトは 10

-record

レコード定義体のあるディレクトリ。デフォルトは起動したカレントディレクトリ

■ **例**

```
dbgen -create sample.db
```

sample.db を元に生成する。

■ **バグ**

■ **参照**

■ **注意**

B.3 htcgen

■ **名前**

htcgen - glade を元に htc を生成する

■ **書式**

ruby htcgen.rb glade **ファイル**

■ **説明**

htcgen は glade ファイルを元に htc ファイルを生成する。結果は標準出力に出る。
htc ファイルは mon が使う web インターフェイス用のダイナミック HTML の一種で、Gtk+ のウィジェットに対応したマクロが定義されている。

■ **オプション**

■ **例**

```
ruby htcgen.rb project1.glade > project1.htc
```

./project1.glade を元に project1.htc を作る

■ **バグ**

GtkCalendar の処理が完全ではない
その他、対応しきれていないウィジェットが多数ある

■ **参照**

■ **注意**

B.4 checkdir

■ **名前**

checkdir - 環境定義体から各種情報を出力する

■ **書式**

```
checkdir <OPTIONS>
```

■ **説明**

checkdir は環境定義体から各種情報を出力するコマンドである。結果は標準出力に出る。

■ **オプション**

-ld

LD 情報を出力する。デフォルトは OFF。

-bd

バッチ情報を出力する。デフォルトは OFF。

-dbg

DB グループ情報を出力する。デフォルトは OFF。

-dir

環境定義体のファイル名を指定する。デフォルトは./directory

-record

レコード定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-lddir

LD 定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-bddir

バッチ定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

■ **例**

```
checkdir -ld -bd -dbg
```

./directory を元に、LD, バッチ, DB グループの情報を出力する

■ **バグ**

■ **参照**

■ **注意**

B.5 user

■ **名前**

gluseradd - ユーザを登録する

gluserdel - ユーザを削除する

glusermod - ユーザ情報を変更する

■ **書式**

gluseradd <OPTIONS> **ユーザ名**

gluserdel <OPTIONS> **ユーザ名**

glusermod <OPTIONS> **ユーザ名**

■ **説明**

glserver が認証するユーザ情報を管理するコマンド群である。

■ **オプション**

-file

パスワードファイルを指定する。デフォルトは./passwd

-u

ユーザ ID となる数値を指定する。gluseradd, glusermod で有効。デフォルトは、gluseradd の際は現在パスワードファイルが保持している最大のユーザ

$$ID + 1$$

。glusermod の際は元のユーザ ID。

-g

グループ ID となる数値を指定する。gluseradd, glusermod で有効。デフォルトは、gluseradd の際は 0。glusermod の際は元のグループ ID。

-o

ユーザの付加情報を指定する。任意の文字列が指定可能。デフォルトは空文字列。

-p

パスワードを指定する。パスワードは平文で指定する。デフォルトは空文字列。

■ **例**

```
gluseradd ogochan -p "1wsadfghp"
```

ユーザ ogochan を登録する

```
gluserdel ogochan -file /usr/local/etc/glpasswd
```

/usr/local/etc/glpasswd に登録されているユーザ ogochan を削除する。

■ **バグ**

■ **参照**

glauth

■ **注意**

ユーザ情報変更後は、glauth に SIGHUP を送って、パスワードファイルを再読み込みさせなくてはならない。

B.6 monitor

■ 名前

`monitor` - wfc, aps, dbredirector の一括起動監視プログラム

■ 書式

```
monitor <OPTIONS>
```

■ 説明

`monitor` は wfc, aps, dbredirector を環境定義体等の定義に従って起動し、異常終了等の検出すると再起動を行う。

■ オプション

-ApsPath

aps のコマンドパスを指定する。デフォルトは`/usr/local/panda/bin/aps`。

-wfcPath

wfc のコマンドパスを指定する。デフォルトは`/usr/local/panda/bin/wfc`。

-RedPath

dbredirector のコマンドパスを指定する。デフォルトは`/usr/local/panda/bin/dbredirector`。

-myhost

プログラムが起動されるホスト名を指定する。これは分散環境でのプロセス起動を行う時に利用される。デフォルトは localhost。

-redirector

dbredirector を起動する。デフォルトでは起動しない。

-restart

プロセス監視を行い、プロセス異常終了時には出来る限りの再起動を行う。デフォルトでは監視しない。

-log

ログを取得する。現在のところプロセス起動のコマンドラインのログが取得される。デフォルトでは取得しない。

-dir

環境定義体のファイル名を指定する。デフォルトは`./directory`

-record

レコード定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-ddir

LD 定義体のあるディレクトリ。デフォルトは-dir で指定したファイルの内容に従う

-q

プロセスの起動をする時に`-?`を指定する。全てのプログラムは、`-?`が指定されると、有効になったコマンドパラメータの表示を行って終了する。すなわち、このパラメータを指定すると、実際に指定されたパラメータがどうなっているか知ることが出来る。デフォルトは OFF。

■ 例

```
monitor -ApsPath ~/panda/aps/aps -WfcPath ~/panda/wfc/wfc -dir ~/panda/samples/sample2/director
```

~/panda/samples/sample2/directory を参照して、必要なプロセスを起動する。

■ **バグ**

■ **参照**

wfc, aps, dbredirector

■ **注意**

このコマンドが有効に使われるためには、個々のプロセスが環境定義体を上書きするパラメータの指定なしで起動出来る程度に設定されていなければならない。

-restart が指定されていると、アプリケーションが異常終了してもリカバリするが、デバッグ時にこのオプションが指定されていると、虫の発見に支障をきたす危険があるので、このオプションを指定するのは、実際の運用になってからの方が望ましい。

付録 C

定義体の文法

C.1 システム定義体文法

C.1.1 構文

システム定義体 ::= **システム名定義** [**ディレクトリ宣言**] [multiplex_level **指定**]
[**画面スタック宣言**] [**システム共通領域宣言**]
[LD **宣言**] [**バッチ宣言**] [wfc **宣言**] { DB **グループ宣言** } .
システム名定義 ::= "name" **システム名** ";" .
システム名 ::= **名前** .
ディレクトリ宣言 ::= [**ベースディレクトリ宣言**] [**定義体ディレクトリ宣言**]
[**レコード定義ディレクトリ宣言**] .
ベースディレクトリ宣言 ::= "base" **ベースディレクトリ名** .
ベースディレクトリ名 ::= **文字列** .
定義体ディレクトリ宣言 ::= "ddir" **定義体ディレクトリ名** .
定義体ディレクトリ名 ::= **文字列** .
wfc **宣言** ::= "wfc" "{" { wfc **宣言行** } "}" ";" .
wfc **宣言行** ::= [aps **待機ポート定義**] [PL **サーバ待機ポート定義**] .
aps **待機ポート定義** ::= "port" **待機ポート** ";" .
PL **サーバ待機ポート定義** ::= "termport" **待機ポート** ";" .
待機ポート ::= **ポート** .
レコードディレクトリ宣言
::= "record" **レコードディレクトリ名** .
レコードディレクトリ名 ::= **文字列** .
multiplex_level **指定** ::= "multiplex_level" **多重化レベル** .
多重化レベル ::= "no" | "ld" | "id" | "aps" .
画面スタック宣言 ::= "stacksize" **大きさ** ";" .
大きさ ::= **整数** .
システム共通領域宣言 ::= "linkage" **システム共通領域名** .
システム共通領域名 ::= **名前** .
LD **宣言** ::= "ld" "{" { LD **宣言行** } "}" ";" .
LD **宣言行** ::= LD **名** aps **記述** ";" .

```
aps 記述                    ::= [ [ "*" ]  aps 数 ] .
aps 数                      ::= 整数 .
データベースグループ定義
                            ::= "db_group" [ データベースグループ名 ] "{"
                                共通項目 [ データベースクラス依存項目 ] "}" ";" .
データベースグループ名   ::= 文字列 .
共通項目                    ::= データベースクラス宣言 [ 移送先定義 ] [ 移送ポート定義 ]
                             [ commit 優先度定義 ] [ ログファイル宣言 ] [ データベース名宣言 ] .
データベースクラス宣言   ::= "type" データベースクラス名 ";" .
データベースクラス名     ::= 文字列 .
移送先定義                  ::= "redirect" 移送先データベースグループ名 ";" .
移送ポート宣言           ::= "redirect_port" 移送ポート ";" .
移送ポート                  ::= ポート .
commit 優先度定義          ::= "priority" 優先度パラメータ ";" .
優先度パラメータ       ::= 整数 .
ログファイル宣言          ::= "log" ファイル名 ";" .
ファイル名                  ::= 文字列 .
データベース名宣言       ::= "name" データベース名 ";" .
データベース名             ::= 文字列 .
```

C.1.2 その他事項

- **外部ファイル参照**
  本定義ファイルは、任意の位置で他のファイルを読み込むことが出来る。このための擬似命令が #include である。使用するには、#include の後に<>または""でくくったファイル名を指定すると、指定したファイルを組み入れる
- **コメント**
  本定義ファイルは、任意の位置にコメントが記述出来る。コメントとなるのは、#のから行末までである
- **ディレクトリ指定のメタ文字**
  ディレクトリ指定の際は、以下のメタ文字が有効となる
  - ~
    起動したユーザのホームディレクトリを意味する
  - =
    base で定義したディレクトリ、あるいは起動パラメータでの-base で指定したディレクトリを意味する

C.2 LD 定義体文法

??

C.2.1 構文

LD 定義体 ::= LD 名定義 [言語ハンドラ定義] [multiplex_group 指定]
[キャッシュ数指定] [デフォルト定義]
[ホームディレクトリ宣言] { bind 定義 }
DC データ定義 { DB 宣言 } .
LD 名定義 ::= "name" LD 名 ";" .
LD 名 ::= 名前 .
言語ハンドラ定義 ::= "handler" ハンドラ名 "{" ハンドラ定義 "}" ";" .
ハンドラ定義 ::= クラス宣言 [データ変換宣言] [起動パラメータ宣言]
[文字コード宣言] [文字列変換規則宣言] [ロードパス宣言] .
クラス宣言 ::= "class" クラス名 ";" .
クラス名 ::= (名前 | 文字列) .
データ変換宣言 ::= "serialize" 変換規則名 ";" .
変換規則名 ::= (名前 | 文字列) .
起動パラメータ宣言 ::= "start" 起動パラメータ ";" .
起動パラメータ ::= 文字列 .
文字コード宣言 ::= "coding" 文字コード名 ";" .
文字コード名 ::= (名前 | 文字列) .
文字列変換規則宣言 ::= "encoding" エンコード名 ";" .
エンコード名 ::= (名前 | 文字列) .
ロードパス宣言 ::= "loadpath" ロードパス ";" .
ロードパス ::= 文字列 .
multiplex_group 指定 ::= "multiplex_group" multiplex_group 名 ";" .
multiplex_group 名 ::= 名前 .
bind 定義 ::= "bind" 画面名 ハンドラ名 モジュール名 ";" .
画面名 ::= 文字列 .
ハンドラ名 ::= 文字列 .
モジュール名 ::= 文字列 .
ホームディレクトリ宣言
::= "home" 実行ディレクトリ名 .
実行ディレクトリ名 ::= 文字列 .
DC データ定義 ::= デフォルト定義 レコード定義 .
デフォルト定義 ::= 配列定義 文字列定義 .
配列定義 ::= "arraysize" 大きさ ";" .
文字列定義 ::= "textsize" 大きさ ";" .
大きさ ::= 整数 .
キャッシュ数指定 ::= "cache" 保持セション数 ";" .
保持セション数 ::= 整数 .
DC データ定義 ::= "data" "{" SPA 定義 | window 定義 "}" .
SPA 定義 ::= "spa" SPA レコード名 ";" .

SPA **レコード名** ::= **名前** .
ウィンドウ定義 ::= "window" "{" { **ウィンドウ名行** } "}" ";" .
ウィンドウ名行 ::= **ウィンドウ名** ";" .
ウィンドウ名 ::= **名前** .
DB **宣言** ::= "db" [DB **グループ名**] "{" { DB **名行** } "}" ";" .
DB **グループ名** ::= **文字列** .
DB **名行** ::= DB **名** ";" .
DB **名** ::= **名前** .

C.2.2 その他事項

- **外部ファイル参照**
  本定義ファイルは、任意の位置で他のファイルを読み込むことが出来る。このための擬似命令が #include である。使用するには、#include の後に<>または""でくくったファイル名を指定すると、指定したファイルを組み入れる
- **コメント**
  本定義ファイルは、任意の位置にコメントが記述出来る。コメントとなるのは、#のから行末までである
- **ディレクトリ指定のメタ文字**
  ディレクトリ指定の際は、以下のメタ文字が有効となる
  - **〜**
    起動したユーザのホームディレクトリを意味する
  - **=**
    base で定義したディレクトリ、あるいは起動パラメータでの-base で指定したディレクトリを意味する

C.3 バッチ定義体文法

C.3.1 構文

バッチ定義体 ::= **バッチ定義体名定義** [**言語ハンドラ定義**] [**デフォルト定義**]
[**ホームディレクトリ宣言**] { bind **定義** } { DB **宣言** } .
バッチ定義体名定義 ::= "name" **バッチ定義体名** ";" .
バッチ定義体名 ::= **名前** .
言語ハンドラ定義 ::= "handler" **ハンドラ名** "{" **ハンドラ定義** "}" ";" .
ハンドラ定義 ::= **クラス宣言** [**データ変換宣言**] [**起動パラメータ宣言**]
[**文字コード宣言**] [**文字列変換規則宣言**] [**ロードパス宣言**] .
クラス宣言 ::= "class" **クラス名** ";" .
クラス名 ::= (**名前** | **文字列**) .
データ変換宣言 ::= "serialize" **変換規則名** ";" .
変換規則名 ::= (**名前** | **文字列**) .
起動パラメータ宣言 ::= "start" **起動パラメータ** ";" .
起動パラメータ ::= **文字列** .
文字コード宣言 ::= "coding" **文字コード名** ";" .
文字コード名 ::= (**名前** | **文字列**) .
文字列変換規則宣言 ::= "encoding" **エンコード名** ";" .
エンコード名 ::= (**名前** | **文字列**) .
ロードパス宣言 ::= "loadpath" **ロードパス** ";" .
ロードパス ::= **文字列** .
デフォルト定義 ::= **配列定義** **文字列定義** .
配列定義 ::= "arraysize" **大きさ** ";" .
文字列定義 ::= "textsize" **大きさ** ";" .
大きさ ::= **整数** .
ホームディレクトリ宣言
::= "home" **実行ディレクトリ名** .
bind **定義** ::= "bind" **モジュール名** **言語ハンドラ名** ";" .
言語ハンドラ名 ::= **文字列** .
モジュール名 ::= **文字列** .
DB **宣言** ::= "db" [DB **グループ名**] "{" { DB **名行** } "}" ";" .
DB **グループ名** ::= **文字列** .
DB **名行** ::= DB **名** ";" .
DB **名** ::= **名前** .

C.3.2 その他事項

- **外部ファイル参照**
本定義言語では、任意の位置で他のファイルを読み込むことが出来る。このための擬似命令が#include

である。使用するには、#include の後に<>または""でくくったファイル名を指定すると、指定したファイルを組み入れる

- **コメント**
  本定義言語では、任意の位置にコメントが記述出来る。コメントとなるのは、#のから行末までである

C.4 データベース公開定義体文法

C.4.1 構文

データベース公開定義体
::= **データベース公開定義体名定義** [**デフォルト定義**]
[**ホームディレクトリ宣言**] { DB **宣言** } .

データベース公開定義体名定義
::= "name" **データベース公開定義体名** ";" .

データベース公開定義体名
::= **名前** .

デフォルト定義 ::= **配列定義** **文字列定義** .

配列定義 ::= "arraysize" **大きさ** ";" .

文字列定義 ::= "textsize" **大きさ** ";" .

大きさ ::= **整数** .

ホームディレクトリ宣言
::= "home" **実行ディレクトリ名** .

DB **宣言** ::= "db" [DB **グループ名**] "{" { DB **名行** } "}" ";" .

DB **グループ名** ::= **文字列** .

DB **名行** ::= DB **名** ";" .

DB **名** ::= **名前** .

C.4.2 その他事項

- **外部ファイル参照**
  本定義言語では、任意の位置で他のファイルを読み込むことが出来る。このための擬似命令が#include である。使用するには、#include の後に<>または""でくくったファイル名を指定すると、指定したファイルを組み入れる
- **コメント**
  本定義言語では、任意の位置にコメントが記述出来る。コメントとなるのは、#のから行末までである

C.5 DB 定義体文法

C.5.1 構文

```
DB 定義体          ::= レコード定義 [ プライマリキー定義 ] [ 参照定義 ]
                       [ パス定義 ] .
プライマリキー定義  ::= "primary" "{" キー要素並び行 "}" ";" .
キー要素並び行      ::= キー要素並び ";" .
キー要素並び        ::= キー要素名 | キー要素名 "," キー要素並び .
キー要素名          ::= 名前 .
参照定義            ::= "use" DB 定義名 .
DB 定義名           ::= 名前 .
パス定義            ::= "path" パス名 [ 引数定義 ] "{" 機能並び "}" ";" .
パス名              ::= 名前 .
機能並び            ::= { 機能 } .
機能                ::= [ "operation" ] 機能名 [ 引数定義 ]
                        "{" スクリプト並び "}" ";" .
スクリプト並び      ::= { スクリプト行 } .
スクリプト行        ::= スクリプト命令 ";" .
引数定義            ::=  "(" メンバ定義 ")" .
```

レコード定義、メンバ定義については、**??**を参照のこと。

C.5.2 パス定義におけるスクリプトの記述方法

スクリプトの実行についての実装は、それぞれのハンドラの実装のしかたに依る。多くの場合、任意の命令を任意の長さで記述出来るはずである (そう実装されていることが多い)。スクリプト内変数名はそのまま、ホスト変数名には:を前置する。このホスト変数名参照の時も構造の要素が使える。

文が区切られた場合、実際のデータベース処理は、この区切られた単位で行われる。

:*はその DB レコードの全ホスト変数を表現する。これは SQL の全カラム指定の*に対応したものである。この指定が実際に使えるかどうかは、ハンドラの実装に依る。

C.6 データ構造定義文法

C.6.1 構文

```
データ構造定義  ::= レコード定義 .
レコード定義    ::= "{" { メンバ定義 } "}" .
メンバ定義      ::= ( 実メンバ定義 | 仮想メンバ定義 ) .
実メンバ定義    ::= メンバ名 型 { 配列指定 } { "," 属性 } ";" .
仮想メンバ定義  ::= メンバ名 "=" 完全修飾メンバ名 ";" .
メンバ名        ::= 名前 .
完全修飾メンバ名    ::= レコード名 "." 長形式メンバ名 .
```

長形式メンバ名 ::= **名前** ["." **名前**] .
配列指定 ::= "[" (**配列サイズ** | "") "]" .
配列サイズ ::= **整数** .
型 ::= bool **型** | byte **型** | binary **型** | char **型** | varchar **型** | float **型**
| **整数型** | number **型** | text **型** | object **型** | dbcode **型** | **構造体型** .
bool **型** ::= "bool" .
byte **型** ::= "byte" ["(" byte **数** ")"] .
binary **型** ::= "binary" .
char **型** ::= "char" ["(" **文字数** ")"] .
varchar **型** ::= "varchar" ["(" **文字数** ")"] .
float **型** ::= "float" .
整数型 ::= "int" .
number **型** ::= "number" ["(" **桁数** ["," **小数点以下桁数**] ")"] .
text **型** ::= "text" .
object **型** ::= "object" .
dbcode **型** ::= "dbcode" ["(" **文字数** ")"] .
構造体型 ::= **レコード定義** .
文字数 ::= **整数** .
byte **数** ::= **整数** .
桁数 ::= **整数** .
小数点以下桁数 ::= **整数** .
属性 ::= ["virtual"] ;

C.6.2 その他事項

- **外部ファイル参照**
  本定義言語では、任意の位置で他のファイルを読み込むことが出来る。このための擬似命令が#include である。使用するには、#include の後に<>または""でくくったファイル名を指定すると、指定したファイルを組み入れる
- **コメント**
  本定義言語では、任意の位置にコメントが記述出来る。コメントとなるのは、#のから行末までである

C.7 HTC 定義体

HTC 定義体とは、web 上で panda を実行するにあたって、ホスト変数を参照してでデータの入出力や体装の調整を行うことを目的として、HTML を拡張したものである。
実際の内容は、HTML にいくつかのタグを追加したものである。

C.7.1 追加タグ一覧

以下に追加されたタグの一覧を示す。

- **ENTRY**
  1 行文字列の入出力
- **COMBO**
  リストからの選択文字列入力
- **FIXED**
  文字列の出力
- **TEXT**
  複数行文字列の入出力
- **BUTTON**
  処理起動用ボタン
- **TOGGLEBUTTON**
  クリックで ON/OFF の切り換わる状態ボタン
- **CHECKBUTTON**
  クリックでチェックの付く状態ボタン
- **RADIOBUTTON**
  クリックでチェックの付く選択状態ボタン
- **CALENDAR**
  クリックで日付入力の出来るカレンダ
- **COUNT**
  繰り返し用命令

C.7.2 ENTRY

1 行文字列の入出力を行う。
属性は以下の通り。

name
入出力するホスト変数の名前を指定する文字列
size
表示する領域の大きさ。単位は文字数
maxlength
入力する文字列の最大長

C.7.3 COMBO

選択リストを選択することにより、文字列の入力を行う。
属性は以下の通り。

name
入力するホスト変数の名前を指定する文字列
size
表示する領域の大きさ。単位は文字数
item

選択リストを与える文字列配列変数
count
選択リストの要素数

C.7.4 FIXED

文字列の出力をホスト変数から行う。
属性は以下の通り。

name
出力するホスト変数の名前を指定する文字列
size
表示する領域の大きさ。単位は文字数

C.7.5 TEXT

複数行文字列の入出力を行う。
属性は以下の通り。

name
入出力するホスト変数の名前を指定する文字列
rows
領域の行数
cols
領域の桁数

C.7.6 BUTTON

処理起動用ボタン。
属性は以下の通り。

event
ボタンの識別に使うイベント文字列
face
ボタン上に表示する文字列
size
大きさ

C.7.7 TOGGLEBUTTON

クリックで ON/OFF の切り換わる状態ボタン。
属性は以下の通り。

name
入出力するホスト変数の名前を指定する文字列
label
ラベルで表示するホスト変数の名前を指定する文字列

C.7.8　CHECKBUTTON

クリックでチェックの付く状態ボタン。
属性は以下の通り。

name
入出力するホスト変数の名前を指定する文字列
label
ラベルで表示する文字列

C.7.9　RADIOBUTTON

クリックでチェックの付く選択状態ボタン
属性は以下の通り。

name
入出力するホスト変数の名前を指定する文字列
label
ラベルで表示する文字列
group
組にして動かすボタンを識別するための文字列

C.7.10　CALENDAR

日付入力を行うカレンダー
属性は以下の通り。属性を省略すると、「今日」が指定されたものとして、カレンダー表示のみを行う。

year
「年」を入出力するホスト変数の名前を指定する文字列
month
「月」を入出力するホスト変数の名前を指定する文字列
day
「日」を入出力するホスト変数の名前を指定する文字列

C.7.11　COUNT

範囲を定義して、その範囲内を指定した回数だけ展開する。

var
繰り返しの数を数える変数 (制御変数) を指定する文字列。省略すると無名
from
var の初期値。省略すると 0
to
var の最終値。省略は出来ない
step
var を増やす時の増分。省略すると 1

<COUNT>**命令は、**\verbvar—で指定した制御変数が、from の値から to の値になるまで step 分だけ増やし、</COUNT>までの範囲を展開する。制御変数名が違えば、多重にすることも可能である。
実際に使う場合には、たとえば以下のようにする。

```
<TABLE BORDER>
<TR>
<TD align="center"><fixed name="project1.vbox1.swin1.clist1.label1.value"></TD>
<TD align="center"><fixed name="project1.vbox1.swin1.clist1.label2.value"></TD>
<TD align="center"><fixed name="project1.vbox1.swin1.clist1.label3.value"></TD>
    <COUNT var="i" from=0 to="project1.vbox1.swin1.clist1.count">
    <TR>
    <TD align="left"><fixed name="project1.vbox1.swin1.clist1.item[#i].value1"></TD>
    <TD align="left"><fixed name="project1.vbox1.swin1.clist1.item[#i].value2"></TD>
    <TD align="left"><fixed name="project1.vbox1.swin1.clist1.item[#i].value3"></TD>
    </TR>
    </COUNT>
</TABLE>
```

この例では、制御変数である i を使って、project1.vbox1.swin1.clist1.iterm[].value 順にアクセスしながら、それを埋め込むためのテーブルを生成している。変数を引用する場合、ホスト変数と区別するために、現在のところ#を前置しなくてはならない。

C.8 環境変数

本システムで使う環境変数は以下のものがある。

- *MONPS_LOAD_PATH*
  glserver のアプリケーションの場所。具体的には panda.so の場所。デフォルトは$libdir
- *APS_HANDLER_LOAD_PATH*
  aps(dbstab も含む) の言語ハンドラの場所。デフォルトは$libdir
- *MONDB_LOAD_PATH*
  HAKAMA のデータベースハンドラの場所。デフォルトは$libdir
- *MONDB_LOCALE*
  データベースバックエンドの locale。デフォルトは const.h の DB_LOCALE に従う

- *APS_LOAD_PATH*
  C のアプリケーションロードパス。デフォルトは`$libdir`
- *APS_EXEC_PATH*
  exec type アプリケーションロードパス。デフォルトは`$libdir`
- *COB_LIBRARY_PATH*
  OpenCOBOL のアプリケーションロードパス。デフォルト**は**`$libdir`
- *LD_LIBRARY_PATH*
  その他 dynamic load するライブラリのパス
- *LOG_FILE_NAME*
  メッセージを出力するファイル。デフォルトは標準出力。先頭に`@` があると、それ以降をホスト名とみなして、そのホストの *msgd* に出力する
- *LOG_DATA_FORMAT*
  ログデータの出力フォーマットを決定する文字列。デフォルトは、`%Y/%M/%D/%h:%m:%s %F:%f:%L:%B`。ただし、*msgd* を使う場合は、`%F:%i:%f:%L:%B` に固定である。
- *RUBYLIB*
  Ruby のライブラリロードパス

C.8.1 LOG_DATA_FORMAT の意味

フォーマット文字列はフラグ文字と一般の文字に分類される。一般の文字はそのまま出力され、フラグ文字は以下に挙げるルールにより置換されて出力される。

フラグ文字は `%` を前置することによって識別される。

文字	意味
Y	年 (4 桁)
M	月 (2 桁)
D	日 (2 桁)
h	時 (2 桁)
m	分 (2 桁)
s	秒 (2 桁)
F	発生ソースファイル名
f	メッセージフラグ
i	モジュール ID
L	ソース行番号
B	メッセージ本体

C.9 Unti-news(最新版との差異)

本ドキュメントは最新開発版の **MONTSUQI** について記述されている。しかし、現在流通しているものは、安定版であるため、機能的には本ドキュメントでの解説よりも若干不足している。本章では、最新開発版と安定版、およびそれ以前の版との機能の差について説明を行う。

C.9.1 開発版と安定版の差異

- **ポート指定の変更**
  ポートの指定は TCP のみである。
- **aps は wfc からの接続を受ける**
  あまり実用的でない接続なので廃止した
- **システム定義体**
  - ワークフローコントローラは制御接続を持たない
  - ddir の変わりに lddir, bddir, dbddir の 3 種類の宣言子がある。ただし、これは同じものを指す
- **wfccontrol**
  ワークフローコントローラは制御接続を持たないため、このコマンドを持たない。panda を正常終了させる唯一の方法は kill である

C.9.2 安定版と旧安定版の差異

- **dbredirector のための記述**
  - 移送元のデータベースグループで移送先が待機する**べき**ポート番号を指定する。
  - データベースグループの記述にファイル名を書くと、そのデータベースグループに対応する `dbredirector` の起動が必要になり、その `dbredirector` がファイルに記録をする。
- **copygen**
  起動オプションには`-ddir` の代わりに`-lddir`,`-bddir` があり、それぞれを指定する。
- **glclient には V2 手順がない**
  glclient とやりとりするデータ構造はウィジェットの構造に依存したものしかない。これ以外の手順は存在しないので、解釈を指定するオプションも存在しない。
- **データの順序**
  画面レコードのデータ並びは、bind で出現するウィンドウ名に対応した順に並ぶ
- **データベースからの復帰コード**
  SQL 文がいくつもあるデータベース操作の場合、最後に実行した SQL の状態が返却される。致命的なエラーの時は-1 が返却され、取得すべきデータがなかった場合は 1 が返却され、正常な時は 0 が返却される。
- **LD 定義体**
  アプリケーションサーバは SPA データ等のキャッシュを持たないので、キャッシュ数の指定はない
- **システム定義体**
  - **multiplex_level**
    1 つの LD に対して起動可能なアプリケーションサーバは 1 つだけである。”aps” という並列化パラメータは存在しない
  - **wfc 宣言子**
    ない
  - **ld**
    * aps の多重化がないので、aps の多重度を設定する宣言はない
    * wfc からの接続を待機するポートの指定がある

- **言語ハンドラは dotCOBOL, OpenCOBOL のみである**
- **Shell ハンドラの名前は”shell” である**
- **モジュールは組み込みでありプラグインのロードパスはない**
- **データ処理ライブラリは MONTSUQI に組み込みである**
- **国際化はされていない**
- **トラフィックの暗号化はない**