BEA Tuxedo

BEA Tuxedo C リファレンス

BEA Tuxedo リリース 8.0J
8.0 版
2001 年 10 月

Copyright

Restricted Rights Legend

Trademarks or Service Marks

BEA Tuxedo C リファレンス

Document Edition	Date	Software Version
8.0J	2001 年 10 月	BEA Tuxedo リリース 8.0J

目次

このマニュアルについて

このマニュアルでは、BEA Tuxedo ATMI 環境で使用される C 言語関数について説明します。リファレンス・ページは関数名のアルファベット順になっています。

対象読者

このマニュアルは、次のような読者を対象としています。

- BEA Tuxedo 環境でアプリケーションを作成および管理する管理者
- BEA Tuxedo 環境でアプリケーションをプログラミングするアプリケーション開発者

このマニュアルは、BEA Tuxedo プラットフォームおよび C 言語のプログラミングについて理解していることを前提としています。

e-docs Web サイト

BEA 製品のマニュアルは BEA 社の Web サイト上で参照することができます。BEA ホーム・ページの [製品のドキュメント] をクリックするか、または http://edocs.beasys.co.jp/e-docs/index.html に直接アクセスしてください。

マニュアルの印刷方法

このマニュアルは、ご使用の Web ブラウザで一度に 1 ファイルずつ印刷できます。Web ブラウザの [ファイル] メニューにある [印刷] オプションを使用してください。

このマニュアルの PDF 版は、Web サイト上にあります。また、マニュアルの CD-ROM にも収められています。この PDF を Adobe Acrobat Reader で開くと、マニュアル全体または一部をブック形式で印刷できます。PDF 形式を利用するには、BEA Tuxedo Documents ページの [PDF 版] ボタンをクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader をお持ちではない場合は、Adobe Web サイト (http://www.adobe.co.jp/) から無償で入手できます。

関連情報

関連情報は各リファレンス・ページの「関連項目」に一覧表示されています。

サポート情報

皆様の BEA Tuxedo マニュアルに対するフィードバックをお待ちしています。ご意見やご質問がありましたら、電子メールで **docsupport-jp@bea.com** までお送りください。お寄せいただきましたご意見は、BEA Tuxedo マニュアルの作成および改訂を担当する BEA 社のスタッフが直接検討いたします。

電子メール メッセージには、BEA Tuxedo 8.0 リリースのマニュアルを使用していることを明記してください。

BEA Tuxedo に関するご質問、または BEA Tuxedo のインストールや使用に際して問題が発生した場合は、**www.bea.com** の BEA WebSUPPORT を通して BEA カスタマ・サポートにお問い合わせください。カスタマ・サポートへの問い合わせ方法は、製品パッケージに同梱されているカスタマ・サポート・カードにも記載されています。

カスタマ・サポートへお問い合わせの際には、以下の情報をご用意ください。

- お客様のお名前、電子メール・アドレス、電話番号、Fax 番号
- お客様の会社名と会社の住所
- ご使用のマシンの機種と認証コード
- ご使用の製品名とバージョン
- 問題の説明と関連するエラー・メッセージの内容

表記上の規則

このマニュアルでは、以下の表記規則が使用されています。

規則	項目
太字	用語集に定義されている用語を示します。
Ctrl + Tab	2 つ以上のキーを同時に押す操作を示します。
イタリック体	強調またはマニュアルのタイトルを示します。

等幅テキスト	コード・サンプル、コマンドとオプション、データ構造とメンバ、データ型、ディレクトリ、およびファイル名と拡張子を示します。また、キーボードから入力する文字も示します。
	例 :
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

等幅太字	コード内の重要な単語を示します。 例 : `void **commit** ( )`
等幅イタリック体	コード内の変数を示します。 例 : `String *expr*`
大文字	デバイス名、環境変数、および論理演算子を示します。 例 : LPT1 SIGNON OR
{ }	構文の行で選択肢を示します。かっこは入力しません。

規則	項目

[ ]	構文の行で省略可能な項目を示します。かっこは入力しません。
	例 :
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

|	構文の行で、相互に排他的な選択肢を分離します。記号は入力しません。

...	コマンド行で次のいずれかを意味します。
	■ コマンド行で同じ引数を繰り返し指定できること
	■ 省略可能な引数が文で省略されていること
	■ 追加のパラメータ、値、その他の情報を入力できること
	省略符号は入力しません。
	例 :
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	コード例または構文の行で、項目が省略されていることを示します。 省略符号は入力しません。

セクション 3c - C 関数

表 1 BEA Tuxedo C **関数**

名前	機能説明
C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて	C 言語 ATMI についての説明
AEMsetblockinghook(3c)	アプリケーション固有のブロッキング・フック関数を確立
AEOaddtypesw(3c)	実行時に、ユーザ定義のバッファ・タイプをインストールまたはリプレース
AEPisblocked(3c)	進行中のブロッキング呼び出しが存在するかどうかを確認
AEWsetunsol(3c)	BEA Tuxedo 任意通知型イベントに対する Windows メッセージを通知
buffer(3c)	tmtype_sw_t における要素の意味
catgets(3c)	プログラム・メッセージの読み取り
catopen、catclose(3c)	メッセージ・カタログをオープンまたはクローズ
decimal(3c)	十進数変換および算術ルーチン
gp_mktime(3c)	tm 構造体をカレンダー時間に変換
nl_langinfo(3c)	言語情報
rpc_sm_allocate、rpc_ss_allocate(3c)	RPC スタブ内でメモリを割り当て
rpc_sm_client_free、rpc_ss_client_free(3c)	クライアント・スタブから返されたメモリを解放
rpc_sm_disable_allocate、rpc_ss_disable_allocate(3c)	スタブ・メモリ管理スキーマで割り当てられたリソースとメモリを解放

表 1 BEA Tuxedo C 関数 (続き)

名前	機能説明
rpc_sm_enable_allocate、rpc_ss_enable_allocate(3c)	スタブ・メモリ管理環境の有効化
rpc_sm_free、rpc_ss_free(3c)	rpc_sm_allocate() ルーチンによって割り当てられたメモリを解放
rpc_sm_set_client_alloc_free、rpc_ss_set_client_alloc_free(3c)	クライアントのスタブが使用するメモリ管理および開放の機構を設定
rpc_sm_swap_client_alloc_free、rpc_ss_swap_client_alloc_free(3c)	クライアントのスタブが使用するメモリ管理および開放の機構をクライアントが提供する機構に交換
setlocale(3c)	プログラムのロケールを修正および照会
strerror(3c)	エラー・メッセージ文字列を取得
strftime(3c)	日付と時刻を文字列に変換
tpabort(3c)	現在のトランザクションをアボートするルーチン
tpacall(3c)	サービス要求の送信を行うルーチン
tpadmcall(3c)	起動されないアプリケーションの管理
tpadvertise(3c)	サービス名の宣言を行うルーチン
tpalloc(3c)	型付きバッファの割り当てを行うルーチン
tpbegin(3c)	トランザクションを開始するためのルーチン
tpbroadcast(3c)	名前によって通知をブロードキャストするルーチン
tpcall(3c)	サービス要求を送信し、その応答を待つルーチン
tpcancel(3c)	未終了の応答に対する呼び出し記述子を無効にするためのルーチン
tpchkauth(3c)	アプリケーションへの結合に認証が必要であるか検査するルーチン
tpchkunsol(3c)	任意通知型メッセージを検査するルーチン
tpclose(3c)	リソース・マネージャをクローズするルーチン

表 1 BEA Tuxedo C **関数 (続き)**

名前	機能説明
tpcommit(3c)	現在のトランザクションをコミットするルーチン
tpconnect(3c)	会話型サービスの接続を確立するルーチン
tpconvert(3c)	構造体を文字列表現に、または文字列表現を構造体に変換
tpcryptpw(3c)	管理要求内のアプリケーション・パスワードを暗号化
tpdequeue(3c)	キューからメッセージを取り出すルーチン
tpdiscon(3c)	会話型サービスの接続を切断するルーチン
tpenqueue(3c)	メッセージをキューに登録するルーチン
tpenvelope(3c)	型付きメッセージ・バッファに関連付けられているデジタル署名と暗号化情報にアクセス
tperrordetail(3c)	最後の BEA Tuxedo 呼び出しから生じるエラーに関する詳細を取得
tpexport(3c)	型付きメッセージ・バッファを、デジタル署名と暗号化シールを含む、マシンに依存しないエクスポート可能な文字列表現に変換
tpforward(3c)	サービス要求を他のサービス・ルーチンに転送するルーチン
tpfree(3c)	型付きバッファの解放を行うルーチン
tpgetadmkey(3c)	管理用認証キーを取得
tpgetctxt(3c)	現在のアプリケーション関連のコンテキスト識別子を取り出す
tpgetlev(3c)	トランザクションが進行中かどうかをチェックするルーチン
tpgetrply(3c)	以前の要求に対する応答を受信するためのルーチン
tpgprio(3c)	サービス要求の優先順位を受け取るルーチン

表 1 BEA Tuxedo C 関数 (続き)

名前	機能説明
tpimport(3c)	エクスポートされた表現を型付きメッセージ・バッファに再変換
tpinit(3c)	アプリケーションに参加
tpkey_close(3c)	前にオープンされたキー・ハンドルをクローズ
tpkey_getinfo(3c)	キー・ハンドルに関連する情報を取得
tpkey_open(3c)	デジタル署名生成、メッセージの暗号化、またはメッセージの暗号解読のためのキー・ハンドルをオープン
tpkey_setinfo(3c)	キー・ハンドルに関連付けられているオプションの属性パラメータを設定
tpnotify(3c)	クライアント識別子により通知を送信するルーチン
tpopen(3c)	リソース・マネージャをオープンするルーチン
tppost(3c)	イベントを通知
tprealloc(3c)	型付きバッファのサイズを変更するルーチン
tprecv(3c)	会話型接続でメッセージを受信するルーチン
tpresume(3c)	グロ―バル・トランザクションを再開
tpreturn(3c)	サービス・ルーチンから復帰するためのルーチン
tpscmt(3c)	tpcommit() がいつ戻るかを設定するルーチン
tpseal(3c)	暗号化する型付きメッセージ・バッファをマーク
tpsend(3c)	会話型接続でメッセージを送信するルーチン
tpservice(3c)	サービス・ルーチンのテンプレート
tpsetctxt(3c)	現在のアプリケーション関連にコンテキスト識別子を設定
tpsetunsol(3c)	任意通知型メッセージの処理方式を設定

表 1 BEA Tuxedo C **関数 (続き)**

名前	機能説明
tpsign(3c)	デジタル署名のための型付きメッセージ・バッファをマーク
tpsprio(3c)	サービス要求の優先順位を設定するルーチン
tpstrerror(3c)	BEA Tuxedo システムのエラーに関するエラー・メッセージ文字列を取得
tpstrerrordetail(3c)	BEA Tuxedo システムに関する詳細なエラー・メッセージ文字列を取得
tpsubscribe(3c)	イベントにサブスクライブ
tpsuspend(3c)	グローバル・トランザクションを一時停止
tpsvrdone(3c)	BEA Tuxedo システム・サーバを終了
tpsvrinit(3c)	BEA Tuxedo システム・サーバを初期化
tpsvrthrdone(3c)	BEA Tuxedo サーバ・スレッドを終了
tpsvrthrinit(3c)	BEA Tuxedo サーバ・スレッドを初期化
tpterm(3c)	アプリケーションを分離
tptypes(3c)	型付きバッファ情報を判別するルーチン
tpunadvertise(3c)	サービス名の宣言解除を行うルーチン
tpunsubscribe(3c)	イベントへのサブスクライブを取り消す
TRY(3c)	例外戻りインターフェイス
tuxgetenv(3c)	環境名に対して値を返す
tuxputenv(3c)	環境の値を変更または値を環境に追加
tuxreadenv(3c)	ファイルから環境へ変数を追加
tx_begin(3c)	グローバル・トランザクションを開始
tx_close(3c)	リソース・マネージャ・セットをクローズ
tx_commit(3c)	グローバル・トランザクションをコミット

表 1 BEA Tuxedo C 関数 (続き)

名前	機能説明
tx_info(3c)	グロ―バル・トランザクション情報を返す
tx_open(3c)	リソース・マネージャ・セットをオープン
tx_rollback(3c)	グローバル・トランザクションをロールバック
tx_set_commit_return(3c)	*commit_return* 特性を設定
tx_set_transaction_control(3c)	*transaction_control* 特性を設定
tx_set_transaction_timeout(3c)	*transaction_timeout* 特性を設定
userlog(3c)	BEA Tuxedo システムの中央イベント・ログにメッセージを書き込む
Usignal(3c)	BEA Tuxedo システム環境におけるシグナル処理
Uunix_err(3c)	UNIX システム・コール・エラーを出力

C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて

機能説明

アプリケーション・トランザクション・モニタ・インターフェイス (ATMI:Application-to-Transaction Monitor Interface) は、アプリケーションとトランザクション処理システムとの間に介在し、ATMI インターフェイスと呼ばれています。このインターフェイスは、リソースのオープンとクローズ、トランザクションの管理、型付きバッファの管理、要求 / 応答型サービス呼び出しや会話型サービス呼び出しの起動などを行う関数呼び出しを提供します。

コミュニケーション・パラダイム

ATMI リファレンス・ページに記述されている関数呼び出しには、特定のコミュニケーション・モデルがあります。このモデルは、クライアント・プロセスとサーバ・プロセスが要求および応答の各メッセージを使用して如何にコミュニケートできるかという観点から表現されています。

コミュニケーションの基本的パラダイムとして、要求 / 応答型と会話型の 2 つがあります。要求 / 応答型サービスは、サービス要求とそれに関わるデータによって呼び出されます。要求 / 応答型サービスは、要求を 1 つだけ受け取ることができ (該当サービス・ルーチンに入った時点で)、かつ応答も 1 つだけ送信することができます (該当サービス・ルーチンから戻った時点で)。一方、会話型サービスは接続要求によって呼び出されます。このとき、オープンされた接続を参照する手段が必要です (すなわち、以後の接続ルーチンを呼び出す際に使用される記述子)。接続が確立され、サービス・ルーチンが呼び出されると、以後、接続プログラムあるいは会話型サービスは、その接続が解除されるまで、アプリケーションによって定義されたようにデータの送受信を行えるようになります。

なお、プロセスは要求 / 応答型と会話型によるコミュニケーションのいずれをも行うことができますが、両方のサービス要求を受け付けることはできません。以下の節では、これら 2 種類のコミュニケーション・パラダイムについてその概要を説明します。

注記 BEA Tuxedo のマニュアルでは、さまざまな場所でスレッドについて説明しています。マルチスレッドのアプリケーションの説明で「スレッド」と記載されている場合は、文字どおりのスレッドを意味します。ただし、シングルスレッドとマルチスレッドの両方のアプリケーションに関するトピックで「スレッド」という言葉が使用される場合があります。このような場合にシングルスレッド・アプリケーションを実行していれば、「スレッド」はプロセス全体を指しているものと考えてください。

BEA Tuxedo ATMI システムのクライアント / サーバ用要求 / 応答型パラダイム

要求 / 応答型のコミュニケーションの場合、クライアントは要求を送り、応答を受け取ることができる 1 つのプロセスと定義されています。定義によれば、クライアントは要求を受け取ったり、応答を送ったりすることはできません。その代わり、クライアントはいくつでも要求を送ることができ、またそれと同時に応答を待ったり、あるいは適宜ある一定数の応答を受け取ったりすることができます。場合によっては、クライアントは応答を必要としない要求を送ることもできます。`tpinit()` と `tpterm()` を使用すれば、クライアントは BEA Tuxedo ATMI システム・アプリケーションと結合および分離することができます。

一方、要求 / 応答型サーバは一度に 1 つのサービス要求を受け取り、その要求に対して 1 つの応答を返すことができるプロセスと定義されています。ただし、サーバがマルチスレッドであれば、一度に複数の要求を受け取り、一度に複数の応答を返すことができます。サーバは特定の要求を処理しながら、一方で要求 / 応答型要求あるいは会話型要求を出し、それらの応答を受け取ることにより、クライアントのように働くこともできます。サーバはこうした能力の故に、リクエスタと呼ばれることもあります。ただし、クライアント・プロセスとサーバ・プロセスはどちらもリクエスタになることができます (実際、クライアントはリクエスタ以外の何物でもありません)。

要求 / 応答型サーバは別の要求 / 応答型サーバに要求を送る (転送する) ことができます。この場合、最初のサーバは受け取った要求を別のサーバに渡すだけで、応答を受け取ることは期待しません。この連鎖の中の最後サーバがその要求に対する応答をもとのリクエスタに送ります。この転送ルーチンの利用によって、もとのリクエスタは最終的にその応答を受け取ることができるのです。

サーバとサービス・ルーチンの利用から、BEA Tuxedo ATMI システム・アプリケーションの作成に構造化手法をとることが可能になります。サーバ側では、アプリケーション作成者は、要求の受信や応答の送信といったコミュニケーションの詳細ではなく、サービスによって実行する事柄に関する作業に専念すればよいのです。コミュニケーション上の詳細の多くは BEA Tuxedo ATMI システムの `main` が処理するため、サービス・ルーチンを作成するときにはある一定の規則に従う必要があります。サーバは、そのサービス・ルーチンを終了する時点で、`tpreturn()` を使用して応答を送ったり、あるいは `tpforward()` を使用して要求を転送したりできます。サービスはその他の作業を行ったり、この時点以後別のプロセスとコミニュニケートすることは許されません。そのため、サーバが実行するサービスは、要求が受け取られたときに開始され、応答が送信あるいは要求が転送された時点で終了します。

要求メッセージと応答メッセージとの間には、本質的に異なる点があります。すなわち、要求にはそれが送信される以前には関連するコンテキストはありませんが、応答にはあるという点です。たとえば、ある要求を送る際に、呼び出し元はアドレッシング情報を与えなければなりませんが、応答は常にその要求を出したプロセスに返されます。つまり、応答の場合には、要求が出されるときに与えられたアドレッシング情報が維持されていて、応答の送信側はそのあて先になんら手を加えることはできません。この両者の相違点については、`TPCALL(3cbl)` のルーチンのパラメータと説明で明らかにされています。

要求メッセージはその送信時に特定の優先順位が付与されます。この優先順位にしたがって、要求はキューから取り出されます。つまり、サーバはキューの中で最も優先順位の高い要求から順に取り出すのです。ただし、要求がいつまでもキューの中に残されてしまうのを防ぐために、優先順位に関係なく、最も長くキューに入っている要求が一定間隔で取り出されます。デフォルト設定では、要求の優先順位はその要求が送られるサービス名に対応させて付けられます。サービス名にはシステムのコンフィギュレーション時に優先順位を与えることができます (UBBCONFIG(5) 参照)。特に定義されていない場合には、デフォルトの優先順位が使用されます。この優先順位は、tpcall(3c) に説明のあるルーチン (tpsprio()) を使用して実行時に設定することができます。呼び出し元はこの方法により、メッセージ送信時にコンフィギュレーションまたはデフォルトの優先順位を変更できます。

BEA Tuxedo ATMI システムのクライアント / サーバ用会話型パラダイム

会話型のコミュニケーションの場合、クライアントは、会話接続を行うことはできるが、接続要求を受け付けることはできないプログラムと定義されています。

一方、会話型サーバは、接続要求を受け取ることができるプログラムです。接続が確立され、サービス・ルーチンが呼び出されると、以後、接続プログラムあるいは会話型サービスは、その接続が解除されるまで、アプリケーションによって定義されたようにデータの送受信を行えるようになります。会話は半二重方式で行われます。つまり、接続の一方の側が制御権をもってデータを送信し、他方の側は制御権を渡されるまではデータを送信できません。シングルスレッド・サーバでは、接続中のサーバは「予約状態」になり、他のプログラムがそのサーバに接続することはできません。しかしマルチスレッド・サーバに接続しても、そのサーバが 1 プロセスの専用サーバとして予約状態になることはなく、複数のクライアントのスレッドから要求を受け取ることができます。

要求 / 応答型サーバの場合と同様、会話型サーバは他の要求を出したり、他のサーバとの接続を行うことによりリクエスタとして機能できます。一方、要求 / 応答型サーバと異なり、会話サーバは要求を別のサーバに転送することはできません。このため、会話サーバによって実行される会話サービスは、要求を受け取った時点で開始され、tpreturn() を介して最終的な応答が送信された時点で終了します。

接続が確立されると、その接続記述子から、参加プロセス (パーティシパント) に関するアドレッシング情報を得るために必要なコンテキストが分かります。メッセージは、アプリケーション側の規定に従って送受信することができます。要求メッセージ応答メッセージとの間には本質的な相違はなく、またメッセージの優先順位に関する規定もありません。

メッセージの配信

会話モードの場合でも、要求 / 応答モードの場合でも、メッセージの送受信とは、アプリケーションの 2 つのユニット間のコミュニケーションを意味します。ほとんどの場合、メッセージによって応答または少なくとも承認が送られることになります。ですから、メッセージが確実に受信されたことが確認できます。しかし、メッセージ (システムが生成したメッセージやアプリケーションが生成したメッセージ) の中には、応答や承認を得ることができないものもあります。たとえば、システムは tpnotify() を TPACK() フラグなしで使用して任意通知型メッセージを送信でき、アプリケーションは tpacall() を TPNOREPLY() フラグ付きで使用して、メッセージを送信を行うことができます。受信プログラムのメッセージ・キューがいっぱいになった場合は、メッセージは失われます。

送信側と受信側が別のマシンにある場合、コミュニケーションは、ネットワークでメッセージを送受信する BRIDGE プロセス間で行われます。回線異常により、配信失敗の可能性が生じます。このような状況によって、イベントの発生や ULOG メッセージの書き込みを行っても、このイベントや ULOG メッセージと失われたメッセージを結びつけることは容易ではありません。

BEA Tuxedo ATMI システムの目的は広域ネットワークで大量のメッセージを処理することなので、前述のような小さな配信異常を検知して調整できるようにはプログラムされていません。このような理由から、すべてのメッセージを確実に配信できるという保証はありません。

メッセージのシーケンス (順序付け)

tpsend() および tprecv() を使用してメッセージを交換する会話型モデルの場合、メッセージ・ヘッダにシーケンス番号を付加し、送信された順番に受信します。サーバまたはクライアントが順序の誤ったメッセージを受け取ると会話が停止して、進行中のすべてのトランザクションはロールバックし、メッセージ LIBTUX 1572「間違った会話型シーケンス番号」がログに記録されます。

要求 / 応答モードでは、システムがメッセージを順序付けることはありません。アプリケーション・ロジックが順番を付けた場合、アプリケーションがこのシーケンスを監視し制御する責任があります。BRIDGE プロセス用の複数ネットワーク・アドレスをサポートすることで実現する並列メッセージ送信を行うことにより、メッセージが送信順に受信されない可能性が高まります。関連するアプリケーションによって、各 BRIDGE プロセスで使用する単一ネットワークアドレスを指定することができ、またメッセージにシーケンス番号を付加したり、定期的な承認を要求したりすることもできます。

キュー・メッセージ・モデル

BEA Tuxedo ATMI システムのキュー・メッセージ・モデルは、要求メッセージの完了を待たず、そのメッセージが後で処理されるようにキューに登録し、またオプションとしてキューに入れられた応答メッセージを介して応答が得られるようにします。メッセージをキューに登録したり応答をキューから取り出すための ATMI 関数は、`tpenqueue()` と `tpdequeue()` です。これらの関数は、BEA Tuxedo ATMI システムのアプリケーション・プロセスの全ての型：クライアント、サーバ、会話型のいずれのプロセスからも呼び出せます。キューに登録するアプリケーションもキューから取り出すアプリケーションも、サーバまたはクライアントとして指定されていない場合、関数 `tpenqueue()` と `tpdequeue()` をピア・ツー・ピア通信に使用することができます。

キュー・メッセージ機能は、XA 準拠のリソース・マネージャで提供されます。永続的なメッセージはトランザクション内でキューへの登録および取り出しが行われ、一度に処理されます。

ATMI トランザクション

BEA Tuxedo ATMI システムは、トランザクションの定義および管理について、相互に排他的な 2 つの関数をサポートしています。BEA Tuxedo システムの ATMI トランザクション境界関数 (名前の先頭が `tp`) と、X/Open の TX インターフェイス関数 (名前の先頭が `tx_`) です。X/Open では TX インターフェイスのベースとして ATMI のトランザクション境界関数が使用されるので、TX インターフェイスの構文およびセマンティクスは、ATMI とほとんど同じです。この項では、ATMI のトランザクション概念について概要を述べます。次の項では、TX インターフェイスについて補足説明します。

BEA Tuxedo ATMI システムにおけるトランザクションは、全体としてある結果を導く、あるいは何も結果を示さない 1 つの論理的な作業単位を定義するときに使用します。トランザクションにより、多くのプロセスによって (そして、おそらく様々な場所で) なされる作業を 1 つの作業単位として扱うことができます。トランザクションの開始者は `TPBEGIN(3cbl)` および `TPCOMMIT(3cbl)` または `TPABORT(3cbl)` を使用してトランザクション内での操作内容を記述します。

イニシエータはまた、`tpsuspend()` を呼び出して現在のトランザクションでの作業を中断することもできます。他のプロセスが `tpresume()` を呼び出して、中断されているトランザクションのイニシエータの役割を引き継ぐこともできます。トランザクションのイニシエータとして、プロセスは `tpsuspend()`、`tpcommit()` または `tpabort()` のいずれかを呼び出す必要があります。従って、あるプロセスがトランザクションを終了し、別のプロセスがトランザクションを開始することができます。

サービスを呼び出すプロセスがトランザクション・モードにあると、呼び出されたサービス・ルーチンも同じトランザクションのためにトランザクション・モードに入ります。このプロセスがトランザクション・モードでない場合、サービスがトランザクション・モードで呼び出されるかどうかは、コンフィギュレーション・ファイルにおいて該当サービスにどのようなオプションが指定されているかによって決まります。トランザクション・モードで呼び出されないサービスは、それが呼び出された時点から終了時点までの間に複数のトランザクションを定義できます。一方、トランザクション・モードで呼び出されたサービス・ルーチンは、1 つのトランザクションにのみ関与し、終了するまでそのトランザクションでの作業を続けます。なお、接続をトランザクション・モードにアップグレードすることはできません。会話接続中に tpbegin() が呼び出されると、会話状態はそのトランザクションの外側で維持されます (tpconnect() が TPNOTRAN() フラグ付きで呼び出された場合と同様)。

別のプロセスによって起動されたトランザクションに加わるサービスを、パーティシパントと呼びます。トランザクションは常に、1 つのイニシエータをもち、かついくつかのパーティシパントをもつことができます。同じトランザクションでの作業を行うためにサービスを 2 回以上呼び出すことができます。tpcommit() あるいは tpabort() を呼び出すことができるのは、トランザクションのイニシエータだけです (つまり、tpbegin() または tpresume() のいずれかを呼び出すプロセス)。パーティシパントは、tpreturn() あるいは tpforward() を使用することによりトランザクションの結果に影響を与えます。これらの 2 つの呼び出しはそれぞれ、サービス・ルーチンの終わり、およびそのルーチンがトランザクションの中でそれが担当する部分を終了したことを示すものです。

TX トランザクション

TX インターフェイスによって定義されるトランザクションは、ATMI 関数によって定義されるトランザクションと実質的に同じです。アプリケーション開発者は、クライアントとサービスのルーチンを作成する場合、どちらの関数も使用できますが、同一プロセス内に異なる関数を混在させることはできません。つまり、1 つのプロセスで tpbegin() を呼び出した後に tx_commit() を呼び出すことはできません。

TX インターフェイスには、移植性の高い方法でリソース・マネージャのオープンとクローズを行う 2 つの呼び出し tx_open() および tx_close() があります。トランザクションは、tx_begin() で開始され、tx_commit() または tx_rollback() のいずれかで完了します。tx_info() は、トランザクション情報を取り出す際に使用されます。また、トランザクションにオプションを設定する 3 つの呼び出し、tx_set_commit_return()、tx_set_transaction_control()、および tx_set_transaction_timeout() があります。TX インターフェイスには、ATMI の tpsuspend() と tpresume() に相当するものはありません。

TX インターフェイスには、ATMI トランザクションについて定義されているセマンティクスおよび規則の他にも、ここで説明しておくべきセマンティクスがいくつかあります。TX インターフェイスを使用するサービス・ルーチン開発者は、tpsvrinit() を呼び出す独自の tx_open() ルーチンを使用する必要があります。BEA Tuxedo ATMI システムが提供するデフォルトの tpserverinit() は、tpopen() を呼び出します。tpsvrdone() についても同じことがあてはまります。TX インターフェイスを使用している場合は、サービス・ルーチン開発者は、tx_close() を呼び出す独自の tpsvrdone() を使用しなければなりません。

次に、TX インターフェイスには、ATMI にはない別のセマンティクスが 2 つあります。それは、連鎖および非連鎖トランザクションと、トランザクション特性です。

連鎖および非連鎖トランザクション

TX インターフェイスは、トランザクション実行の連鎖モードおよび非連鎖モードをサポートしています。デフォルトでは、クライアント・ルーチンおよびサービス・ルーチンは、非連鎖モードで実行されます。この場合、アクティブなトランザクションが完了した際、新しいトランザクションは tx_begin() が呼び出されるまで開始されません。

連鎖モードでは、新しいトランザクションは、現在のトランザクションが完了すると、暗黙に開始されます。つまり、tx_commit() または tx_rollback() が呼び出されると、BEA Tuxedo ATMI システムは、現在のトランザクションの完了を調整し、制御を呼び出し元に返す前に新しいトランザクションを開始します。(異常終了の条件によっては、新しいトランザクションを開始できない場合もあります)。

クライアント・ルーチンおよびサービス・ルーチンは、tx_set_transaction_control() を呼び出すことによって連鎖モードのオンとオフを切り替えます。連鎖モードと非連鎖モードの間の遷移により、その次の tx_commit() 呼び出しまたは tx_rollback() 呼び出しの動作が変わります。tx_set_transaction_control() の呼び出しでは、呼び出し元のトランザクション・モードのオンとオフの切り替えは行いません。

tx_close() は、呼び出し元がトランザクション・モードにあるときには呼び出すことができないため、連鎖モードで実行中の呼び出し元が tx_close() を呼び出すには、非連鎖モードに切り替えて、現在のトランザクションを完了してから呼び出さなければなりません。

トランザクション特性

クライアント・ルーチンまたはサービス・ルーチンは、tx_info() を呼び出すことによって、そのルーチンのトランザクション特性の現在の値を取得したり、そのルーチンがトランザクション・モードで実行中であるかどうかを判別したりすることができます。

アプリケーション・プログラムの状態には、いくつかのトランザクション特性があります。呼び出し元は、tx_set_*() 関数の呼び出しによってこれらの特性を指定します。クライアント・ルーチンまたはサービス・ルーチンが特性の値を設定している場合は、異なる値を呼び出し元が指定するまでは、その値が有効のままです。呼び出し元が tx_info() を使用して特性の値を取得しても、これによって値が変更されることはありません。

エラー処理

多くの場合、ATMI 機能には 1 つまたは複数のエラー・リターンがあります。エラーの条件は、エラーの他には考えられない戻り値で示されます。この値は通常、エラー時で -1、間違ったフィールド識別子 (BADFLDID) またはアドレスの場合 0 となります。エラー・タイプは外部整数 tperrno() でも参照することができます。正常な呼び出しによって tperrno() がリセットされることはないので、エラーを検出した後にしか呼び出しのテストを行ってはいけません。

tpstrerror() 関数は標準エラー出力へのメッセージを生成します。この関数では 1 つの引数、つまり整数 (tperrno() にセットされている) を必要とし、LIBTUX_CAT のエラー・メッセージ・テキストへのポインタを返します。このポインタは userlog() の引数として使用できます。

現行スレッドで最後の BEA Tuxedo ATMI システム呼び出し時にエラーが発生した場合、エラーの詳細をさらに調べるには手順が 3 つあり、その第一段階として tperrordetail() を使用することができます。tperrordetail() は整数を返しますが、この整数は、エラーメッセージが含まれる文字列へのポインタを取り出す、tpstrerrordetail() の引数として使用します。ポインタは userlog または fprintf() の引数として使用できます。

エラーコードのうち、ATMI 関数で生成できるものについては、ATMI のマニュアル・ページで説明しています。F_error() と F_error32() 関数は、FML エラーの標準エラー出力にメッセージを出力します。この関数は、パラメータを 1 つ (文字列) 取り、コロンと空白を付加してその引数文字列を出力します。次に、エラー・メッセージとその後に続く復帰改行文字を出力します。表示されたエラー・メッセージは Ferror() または Ferror32() で定義したエラー番号に対応しています。これらはエラーが発生した時点で設定されます。

エラー・メッセージのテキストをメッセージ・カタログから取り出す時、Fstrerror() およびこれに相当する Fstrerror32() を使用することができます。これらによって userlog の引数として使用できるポインタを返します。

エラーコードのうち、FML 機能で生成できるものについては、マニュアルの FML の項目で説明しています。

タイムアウト

BEA Tuxedo ATMI システムには 3 種類のタイムアウトがあります。1 つはトランザクションの開始から終了までの期間に関連するもの、2 つめはブロッキング・コールで呼び出し元が制御権を再度入手するまでブロック状態を維持する最大時間に関連するものです。3 つめはサービスのタイムアウトです。これは呼び出しの秒数がコンフィギュレーション・ファイルの SERVICES セクションにおける SVCTIMEOUT パラメータで指定された秒数を越えた時に発生します。

最初のタイムアウトは、tpbegin() を使用してトランザクションを開始するときに指定します (詳細については、tpbegin(3c) を参照)。2 つ目のタイムアウトは、tpcall(3c) に定義されている BEA Tuxedo ATMI システムのコミュニケーション・ルーチンを使用する際に発生することがあります。これらのルーチンの呼び出し元は一般に、まだ届いていない応答を待っている間はブロック状態になります。これらの呼び出し元はデータの送信を行うこともブロックされることがあります (たとえば、要求キューがいっぱいの場合など)。呼び出し元がブロック状態になる最大時間は、BEA Tuxedo ATMI システムのコンフィギュレーション・ファイルに記述されているパラメータによって決まります (詳細については、UBBCONFIG(5) の BLOCKTIME パラメータの項を参照してください)。

ブロッキング・タイムアウトは呼び出し者がトランザクション・モードにないときにデフォルトの設定によって実行されます。クライアントあるいはサーバがトランザクション・モードにあると、そのトランザクションが開始したときに指定されたタイムアウト値が働き、UBBCONFIG ファイルに指定されているブロッキング・タイムアウト値の影響は受けません。

トランザクション・タイムアウトが発生すると、トランザクション・モードで行われた非同期の要求に対する応答は失効状態になることがあります。つまり、あるプロセスが、トランザクション・モードで送信された要求に対する特定の非同期応答の到着を待っているときに、トランザクション・タイムアウトが発生すると、その応答の記述子が無効になります。同様に、トランザクション・タイムアウトが発生すると、そのトランザクションに関連付けられている記述子に対してイベントが生成され、その記述子は無効になります。一方、ブロッキング・タイムアウトが発生した場合、該当する記述子は無効にならず、応答を待機しているプロセスはその応答を待機するための呼び出しを再度出すことができます。

サービス・タイムアウト機構によって、未知の、または予期しないシステム・エラーが原因でフリーズする可能性のあるプロセスについて、システムが強制終了を行うことができます。要求 / 応答サービス時にサービス・タイムアウトが発生すると、BEA Tuxedo ATMI システムによって、フリーズしたサービスを実行中のサーバ・プロセスが強制終了され、エラーコード TPESVCERR が戻ります。サービス・タイムアウトが会話型サービスで発生した場合は、TP_EVSVCERR イベントが返ります。

トランザクションがタイムアウトになった場合、そのトランザクションがアボートされる前の接続のうち、TPNOREPLY、TPNOTRAN、および TPNOBLOCK 付きの tpacall() への呼び出しのみが有効です。

リリース 6.4 から、TPESVCERR エラー・コードに関する詳細が追加されました。タイムアウトのしきい値を超えたためにサービスが失敗した場合、イベント .SysServiceTimeout がポストされます。

動的サービス宣言

デフォルトの設定では、サーバのサービスは、それがブートされるときに宣言され、シャットダウンするときに宣言が解除されます。サーバは、それが提供するサービス・セットに対する制御を実行時に必要とする場合、tpadvertise() および tpunadvertise() を使用します。これらのルーチンは、該当サーバが複数サーバ、単一キュー (MSSQ) セットに属していないかぎり、呼び出し元サーバが提供するサービスだけに影響します。MSSQ セットのサーバはすべて同じサービス・セットを提供しなければならないため、これらのルーチンもまた呼び出し元の MSSQ セットを共用するすべてのサーバの宣言に影響します。

バッファ管理

プロセスはその生成当初、バッファをもちません。メッセージの送信前に tpalloc() を使用してバッファを割り当てなければなりません。送信元のデータはこの後、割り当てられたバッファに一旦入れた後、送信することができます。このバッファは特有の構造をもっています。この構造は、tpalloc() 関数に *type* 引数を付けて指定します。ある種の構造にはさらに分類が必要になるので、サブタイプも与えることができるようになっています (たとえば、特定タイプの C 構造体など)。

メッセージを受け取る際には、アプリケーション・データを受け取ることができるバッファが必要です。このバッファは tpalloc() によって作成されたものでなければなりません。なお、BEA Tuxedo ATMI システムのサーバはその main においてバッファを割り当てますが、このバッファのアドレスはサービス呼び出し時に要求 / 応答型サービスまたは会話型サービスに渡されます (このバッファの扱い方の詳細については、tpservice(3c) を参照)。

メッセージの受信に使用するバッファは、送信に使用するバッファとは扱い方が若干異なります。バッファの大きさとアドレスはメッセージの受信にともなって変わります。これは、システムが内部で受信コールに渡されたバッファを、バッファを処理するのに使用する内部バッファと交換するからです。バッファ・サイズは、受信にともない増加することも減少することもあります。増加するか減少するかは、すべて送信側から送られたデータの量と、受信側が送信側からデータを受け取るために必要な内部データのフローによって決まります。バッファ・サイズに影響を与える要素は数多くあり、圧縮、異なるマシン・タイプからのメッセージの受信、使用されるバッファ・タイプの postrecv() 関数のアクションなどが挙げられます (buffer(3c) を参照)。ワークステーション・クライアントのバッファ・サイズは、通常ネイティブ・クライアントのバッファ・サイズとは異なります。

受信バッファは、メッセージを受信する実際の容器というよりプレイス・ホルダであると考えたほうがよいでしょう。システムは、渡したバッファ・サイズをヒントとして使用することもあるので、バッファを予想される応答を入れられるだけの大きさにすることには意味があります。

送信側では、バッファ・タイプは割り当てられた容量まで満たされないこともあります (例えば、FML や STRING バッファは送信に使われただけの大きさです)。ひとつの整数フィールドを含む 100K の FML32 バッファは、その整数だけを含むより小さいバッファとして送られます。

これは、受信者が送信者の割り当てたバッファ・サイズより小さく、送信されたデータのサイズより大きいバッファを受け取るということです。たとえば、10K バイトの STRING バッファが割り当てられ、文字列 "HELLO" がその中にコピーされた場合は、6 バイトのみが送られますが、受信者は 1K から 4K バイトのバッファを受け取ることになります (他の要素によりこれより大きかったり小さかったりします)。BEA Tuxedo ATMI システムは、受信メッセージがすべての送信されたデータを含んでいることは保証しますが、すべてのフリー・スペースまで保証するわけではありません。

応答を受信するプロセスは、バッファ・サイズの変更を知らせ (`tptypes()` を使用します)、必要な場合は割り当てをやり直します。受信者のバッファ・サイズを変えるすべての BEA Tuxedo ATMI 関数は、バッファ内のデータの量を返すので、応答を受信するたびにバッファ・サイズを確認するのが標準になるでしょう。

メッセージの送受信には同じデータ・バッファを使用することができます。また、それぞれのメッセージに対して別々のデータ・バッファを割り当てることも可能です。通常は、呼び出しプロセスが `tpfree()` を使用してそのバッファを解放します。ただし、ごく限られたケースでは、BEA Tuxedo ATMI システムが呼び出し元のバッファを解放します。バッファの使い方の詳細については、`tpfree()` などの通信関数の説明を参照してください。

バッファ・タイプ・スイッチ

`tmtype_sw_t` 構造体は、新しいバッファ・タイプをプロセスのバッファ・タイプ・スイッチ `tm_typesw()` に追加するために必要とされる記述です。スイッチ・エレメントは `typesw(5)` に定義されています。このエントリに使用される関数名は、BEA Tuxedo ATMI システムまたは独自のバッファ・タイプを作成するアプリケーションによって定義される実際の関数名のテンプレートとなります。これらの関数名は、簡単にスイッチ・エレメントにマップできます。テンプレート名を作成するには、関数ポインタのエレメント名の最初に `_tm` を追加します。たとえば、エレメント `initbuf` のテンプレート名は、`_tminitbuf` になります。

エレメント `type` は NULL 以外とし、最大 8 文字とします。この要素がスイッチ内で一意でない場合、`subtype()` は NULL 以外でなければなりません。

エレメント `subtype()` には NULL、最大 16 文字の文字列、または * (ワイルドカード文字) のいずれかを使用できます。`type()` と `subtype()` の組み合わせは、スイッチ内で一意に要素を指定するものでなければなりません。

あるタイプに複数のサブタイプがあってもかまいません。すべてのサブタイプをあるタイプに対して同じように扱う場合、ワイルドカード文字 "*" を使用できます。なお、サブタイプを区別する必要がある場合には、関数 tptypes() を使用して、バッファのタイプとサブタイプを弁別することができます。ある特定のタイプ内で一定のサブタイプのサブセットを個別に扱う必要があり、残りを同様に扱う場合には、特定の値でまとめるサブタイプは、ワイルドカードで指定する前にスイッチ内に指定しておく必要があります。このため、まずスイッチ内のタイプとサブタイプの検索が上から下の方向に行われ、ワイルドカードによるサブタイプのエントリは、残りの一致するタイプを受け付けることになります。

要素 dfltsize() は、バッファの割り当てまたは再割り当てを行うときに使用します。tpalloc() と tprealloc() の実行では、dfltsize() の値か、または tpalloc() および tprealloc() 関数のサイズ・パラメータ値の、どちらか大きい方の値を使用して、バッファの作成または再割り当てが行われます。固定サイズの C 構造体などの場合、バッファ・サイズはその構造体と同じにするべきです。dfltsize() をこの値に設定すると、呼び出し元はバッファが渡されるルーチンに対してそのバッファの長を指定する必要はなくなります。dfltsize() は 0 あるいはそれ以下にすることができます。ただし、tpalloc() や tprealloc() を呼び出して、その *size* パラメータも 0 もしくはそれ以下であると、このルーチンは異常終了します。dfltsize() は 0 よりも大きい値に設定することをお勧めします。

BEA Tuxedo ATMI システムには、5 つの基本バッファ・タイプがあります。

- CARRAY - 送信時に符号化も復号化も行われないヌル文字を含む文字配列
- STRING - ヌルで終了する文字配列
- FML - フィールド化バッファ (FML または FML32)
- XML - XML 文書またはデータグラム・バッファ
- VIEW - 単純な C 構造体 (VIEW または VIEW32)。すべての VIEW は同じルーチン・セットで処理されます。各 VIEW の名前は、そのサブタイプの名前です。

上記のバッファ・タイプの中の 2 つには、同等のタイプがあります。X_OCTET は CARRAY と同じであり、X_C_TYPE および X_COMMON は VIEW と同じです。X_C_TYPE は、VIEW がサポートするすべての要素をサポートします。これに対し、X_COMMON は、long、short、character のみをサポートします。X_COMMON は、C と COBOL の両方のプログラムがやりとりする際に、使用してください。

アプリケーションで独自のバッファ・タイプを使用する場合には、tm_types 配列にそのインスタンスを追加します。バッファ・タイプを追加したり削除したりする場合は、配列の終わりにヌル・エントリをそのまま残しておくようにしてください。ただし、NULL 名をもつバッファ・タイプを使用することはできません。

buildserver() または buildclient() コマンド行に、-f オプションを用いてソースまたはオブジェクト・ファイルを明示的に指定することにより、アプリケーション・クライアントまたはサーバが、新しいバッファ・タイプ・スイッチにリンクされます。

任意通知

上記のように定義されたクライアント / サーバ間でのやりとりの境界外からメッセージをアプリケーション・クライアントに送る方法は 2 通りあります。第 1 の方法は、tpbroadcast() によって実現されるブロードキャスト機構です。この関数により、アプリケーション・クライアント、サーバおよび管理者は割り当てられた名前に基づいて選択されるクライアントに型付きバッファ・メッセージをブロードキャストすることができます。クライアントに割り当てられる名前は、一部はアプリケーションにより TPINIT 型付きバッファに tpinit() 実行時に渡される情報ごとに、また一部はクライアントがアプリケーションのアクセスに使用するプロセッサに基づいてシステムにより決められます。

もう 1 つの方法は、以前のあるいは現在のサービス要求から識別される特定クライアントによる通知方法です。各サービス要求には、そのサービス要求を出したクライアントを特定する一意のクライアント識別子が含まれています。サービス・ルーチン内で tpcall() や tpforward() が呼び出されても、そのサービス要求の連鎖に対応する元のクライアントは変更されません。クライアント識別子は保存しておき、アプリケーション・サービス間で受け渡すことができます。この方法で特定されたクライアントに対する通知は、関数 tpnotify() を使用して行います。

プロセスごとのシングルコンテキストとマルチ・コンテキスト

BEA Tuxedo ATMI システムでは、クライアント・プログラムはプロセスごとに 1 つまたは複数のアプリケーションとの関連を作成することができます。TPINIT 構造体のフラグ・フィールドに TPMULTICONTEXTS パラメータを指定して tpinit() が呼び出された場合は、複数のクライアント・コンテキストを使用できます。tpinit() が暗黙的に呼び出された場合、ヌル・パラメータによって呼び出された場合、またはフラグ・フィールドに TPMULTICONTEXTS を指定せずに呼び出された場合は、1 つのアプリケーションとの関連しか作成できません。

シングルコンテキスト・モードで tpinit() が 2 回以上呼び出された場合 (つまり、クライアントがアプリケーションに参加した後で呼び出された場合) は、アクションは何も実行されず、成功を示す戻り値が返されます。

マルチコンテキスト・モードの場合、tpinit() の呼び出しのたびに新しいアプリケーション関連が作成されます。アプリケーションは、tpgetctxt() を呼び出すことによって、このアプリケーション関連を表すハンドルを取得することができます。同じプロセス内のどのスレッドも、tpsetctxt() を呼び出してスレッドのコンテキストを設定することができます。

いったんアプリケーションでシングルコンテキスト・モードが選択された場合は、すべてのアプリケーション関連が終了するまで、すべての tpinit() 呼び出しでシングルコンテキスト・モードを指定する必要があります。同様に、アプリケーションでマルチコンテキスト・モードが選択された場合は、すべてのアプリケーション関連が終了するまで、すべての tpinit() 呼び出しでマルチコンテキスト・モードを指定する必要があります。

サーバ・プログラムは 1 つのアプリケーションとしか関連付けられないため、クライアントとして機能することはありません。ただし、各サーバ・プログラム内に、複数のサーバ・ディスパッチ・コンテキストがある場合もあります。各サーバ・ディスパッチ・コンテキストは、それぞれのスレッド内で機能します。

表 2 は、クライアント・プロセス内で発生する、非初期化状態、シングルコンテキスト・モードで初期化された状態、およびマルチコンテキスト・モードで初期化された状態の遷移を示しています。

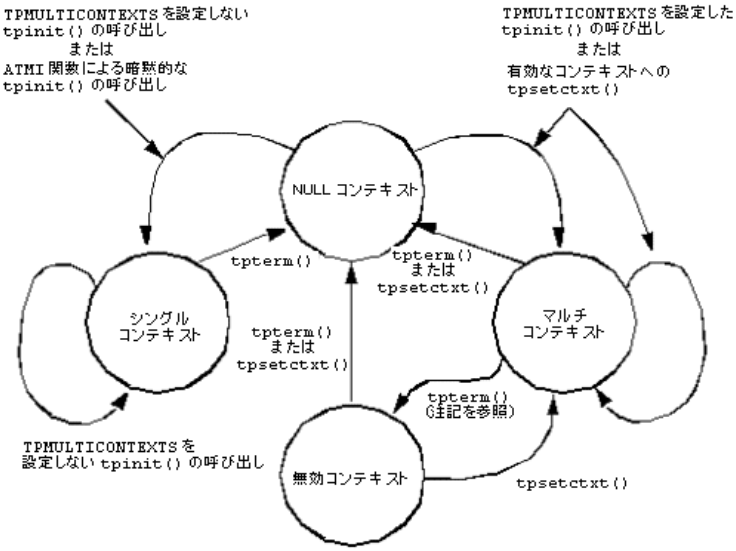
表 2 プロセスごとのコンテキスト・モード

	状態		
関数	非初期化 S_0	初期化されたシングルコンテキスト・モード S_1	初期化されたマルチコンテキスト・モード S_2
TPMULTICONTEXTS を指定しない tpinit	S_1	S_1	S_2(*error*)
TPMULTICONTEXTS を指定した tpinit	S_2	S_1 (*error*)	S_2
暗黙的な tpinit	S_1	S_1	S_2 (*error*)
tpterm – (最後の関連以外)			S_2
tpterm – (最後の関連)		S_0	S_0
tpterm – 関連なし	S_0		

クライアント・スレッドのコンテキスト状態の変化

マルチコンテキストのアプリケーションでは、いろいろな関数を呼び出すと、呼び出し元スレッドおよび呼び出し元プロセスと同じコンテキストでアクティブのその他のスレッドのコンテキスト状態が変化します。次の図は、tpinit()、tpsetctxt()、および tpterm() 関数を呼び出した場合のコンテキスト状態の変化を示しています。tpgetctxt() 関数を呼び出しても、コンテキスト状態は変化しません。

マルチコンテキスト状態の遷移



注記 tpterm() がマルチコンテキスト状態 (TPMULTICONTEXTS) で実行しているスレッドによって呼び出されると、呼び出し元スレッドはヌル・コンテキスト状態 (TPNULLCONTEXT) になります。終了するコンテキストに関連するその他すべてのスレッドは、無効コンテキスト状態 (TPINVALIDCONTEXT) に切り替わります。

表 3 は、tpinit()、tpsetctxt()、および tpterm() を呼び出した場合のコンテキスト状態の変化を示します。いずれの状態もスレッド固有のものであり、マルチコンテキストのアプリケーションの一部を構成するときは、スレッドごとに状態が異なります。一方、前掲の表 (「プロセスごとのコンテキスト・モード」) に示されるコンテキスト状態は、それぞれプロセス全体に適用されます。

表 3 クライアント・スレッドのコンテキスト状態の変化

	実行後のスレッドのコンテキスト状態			
実行する関数	**ヌル・コンテキスト**	**シングルコンテキスト**	**マルチコンテキスト**	**無効コンテキスト**
TPMULTICONTEXTS なしの tpinit	シングルコンテキスト	シングルコンテキスト	エラー	エラー
TPMULTICONTEXTS を指定した tpinit	マルチコンテキスト	エラー	マルチコンテキスト	エラー
TPNULLCONTEXT への tpsetctxt	NULL	エラー	NULL	NULL
コンテキスト 0 への tpsetctxt	エラー	シングルコンテキスト	エラー	エラー
コンテキスト > 0 への tpsetctxt	マルチコンテキスト	エラー	マルチコンテキスト	マルチコンテキスト
暗黙の tpinit	シングルコンテキスト	N/A	N/A	エラー
このスレッドでの tpterm	NULL	NULL	NULL	NULL
このコンテキストの別のスレッドでの tpterm	N/A	NULL	無効	N/A

スレッド・プログラミングのサポート

BEA Tuxedo ATMI システムは、いくつかの方法でプログラミングされたマルチスレッド・プログラミングをサポートしています。プロセスでシングルコンテキスト・モードが使用されている場合に、アプリケーションで新しいスレッドが作成されると、これらのスレッドではそのプロセスの BEA Tuxedo ATMI コンテキストを共有します。クライアントでは、あるスレッドがシングルコンテキスト・モードで tpinit() 呼び出しを発行すると、他のスレッドは ATMI 呼び出しを発行します。たとえば、1 つのスレッドが tpacall() を発行すると、同じプロセス内の別のプロセスは tpgetrply() を発行します。

マルチコンテキスト・モードの場合、最初のスレッドは BEA Tuxedo ATMI アプリケーションに関連付けられません。スレッドは、tpsetctxt() を呼び出して既存のアプリケーション関連に参加するか、TPMULTICONTEXTS フラグを設定した tpinit() を呼び出して新しい関連を作成します。

シングルコンテキスト・モードかマルチコンテキスト・モードかに関わりなく、ATMI 操作が適切なタイミングで実行されるようにスレッドを調整するのは、アプリケーションの役目です。

アプリケーションは、OS スレッド関数を使って、アプリケーション・サーバ内に追加スレッドを生成できます。これらの新しいスレッドは、BEA Tuxedo ATMI システムから独立して動作できます。また、いずれかのサーバ・ディスパッチ・スレッドと同じコンテキストで動作することもできます。最初、アプリケーション生成サーバ・スレッドは、どのサーバ・ディスパッチ・コンテキストにも関連していません。アプリケーションに作成されたサーバ・スレッドは、tpsetctxt() を呼び出し、サーバ・ディスパッチ・スレッドとの関連を確立します。アプリケーションに作成されたサーバ・スレッドは、ディスパッチされたスレッドが tpreturn() または tpforward() を呼び出す前に、すべての ATMI 呼び出しを終了していなければなりません。BEA Tuxedo ATMI システムによってディスパッチされたサーバ・スレッドは、tpsetctxt() を呼び出すことはできません。また、アプリケーション生成スレッドは、コンテキストと関連付けられていない場合、暗黙的に tpinit() を発生させる ATMI を呼び出すことができません。それに対して、ディスパッチャによって作成されたスレッドは、常にコンテキストと関連付けられているため、ATMI 呼び出しに失敗することはありません。すべてのサーバ・スレッドで、tpinit() の呼び出しは禁止されています。

マルチスレッドのアプリケーションでは、TPINVALIDCONTEXT 状態で動作するスレッドが、多数の ATMI 関数を呼び出すことは禁止されています。次のリストは、このような環境で呼び出し可能な関数と呼び出し不可能な関数を示しています。

BEA Tuxedo ATMI システムでは、TPINVALIDCONTEXT 状態で動作するスレッドは以下の関数を呼び出すことができます。

- catgets(3c)
- catopen、catclose(3c)
- decimal(3c)
- gp_mktime(3c)
- nl_langinfo(3c)
- rpc_sm_allocate、rpc_ss_allocate(3c)
- rpc_sm_client_free、rpc_ss_client_free(3c)
- rpc_sm_disable_allocate、rpc_ss_disable_allocate(3c)
- rpc_sm_enable_allocate、rpc_ss_enable_allocate(3c)

- rpc_sm_free、rpc_ss_free(3c)
- rpc_sm_set_client_alloc_free、rpc_ss_set_client_alloc_free(3c)
- rpc_sm_swap_client_alloc_free、
  rpc_ss_swap_client_alloc_free(3c)
- setlocale(3c)
- strerror(3c)
- strftime(3c)
- tpalloc(3c)
- tpconvert(3c)
- tpcryptpw(3c)
- tperrordetail(3c)
- tpfree(3c)
- tpgetctxt(3c)
- tprealloc(3c)
- tpsetctxt(3c)
- tpstrerror(3c)
- tpstrerrordetail(3c)
- tpterm(3c)
- tptypes(3c)
- TRY(3c)
- tuxgetenv(3c)
- tuxputenv(3c)
- tuxreadenv(3c)
- userlog(3c)
- Usignal(3c)
- Uunix_err(3c)

BEA Tuxedo ATMI システムでは、TPINVALIDCONTEXT 状態で動作するスレッドは以下の関数を呼び出すことができません。

- AEWsetunsol(3c)
- tpabort(3c)
- tpacall(3c)
- tpadmcall(3c)
- tpbegin(3c)
- tpbroadcast(3c)
- tpcall(3c)
- tpcancel(3c)

- tpchkauth(3c)
- tpchkunsol(3c)
- tpclose(3c)
- tpcommit(3c)
- tpconnect(3c)
- tpdequeue(3c)
- tpenqueue(3c)
- tpgetadmkey(3c)
- tpgetlev(3c)
- tpgetrply(3c)
- tpgprio(3c)
- tpinit(3c)
- tpnotify(3c)
- tpopen(3c)
- tppost(3c)
- tprecv(3c)
- tpresume(3c)
- tpscmt(3c)
- tpsend(3c)
- tpsetunsol(3c)
- tpsprio(3c)
- tpsubscribe(3c)
- tpsuspend(3c)
- tpunsubscribe(3c)
- tx_begin(3c)
- tx_close(3c)
- tx_commit(3c)
- tx_info(3c)
- tx_open(3c)
- tx_rollback(3c)
- tx_set_commit_return(3c)
- tx_set_transaction_control(3c)
- tx_set_transaction_timeout(3c)

C 言語の ATMI の戻り値とその他の定義

ルーチンは、次に挙げる戻り値とフラグの定義を使用します。異なるトランザクション・モニタで変更や再コンパイルなしにアプリケーションを使用するためには、各システムで次に示すようにそのフラグと戻り値を定義しておかなければなりません。

```
/*
 * 次の定義は atmi.h に含まれなければなりません
 */


/* サービス・ルーチンへのフラグ */

 #define TPNOBLOCK     0x00000001 /* 非ブロック送信 / 受信 */
 #define TPSIGRSTRT    0x00000002 /* 割り込み時受信再開 */
 #define TPNOREPLY     0x00000004 /* 応答なしを期待 */
 #define TPNOTRAN      0x00000008 /* トランザクション・モードでは送信しない */
 #define TPTRAN        0x00000010 /* トランザクション・モードでの送信 */
 #define TPNOTIME      0x00000020 /* タイムアウトなし */
 #define TPABSOLUTE    0x00000040 /* 絶対的な優先順位の指定 */
 #define TPGETANY      0x00000080 /* 有効応答の取り込み */
 #define TPNOCHANGE    0x00000100 /* 受信バッファのマッチング */
 #define RESERVED_BIT1 0x00000200 /* 将来の使用のために予約 */
 #define TPCONV        0x00000400 /* 会話型サービス */
 #define TPSENDONLY    0x00000800 /* 送信専用モード */
 #define TPRECVONLY    0x00001000 /* 受信専用モード */
 #define TPACK         0x00002000 /* */

/* tpreturn() へのフラグ - xa.h にも定義されている */
 #define TPFAIL        0x20000000 /* tpreturn に対するサービスの FAILURE */
 #define TPEXIT        0x08000000 /* サービスの終了によるサービスの FAILURE */
 #define TPSUCCESS     0x04000000 /* tpreturn に対するサービスの SUCCESS */

/* tpscmt() へのフラグ - 有効な TP_COMMIT_CONTROL
 * 特性値
 */
 #define     TP_CMT_LOGGED 0x01
                                  /* コミット決定を記録後、リターン */
 #define     TP_CMT_COMPLETE 0x02
                                  /* コミット完了後、リターン */



/* クライアント識別子構造 */
 struct clientid_t {
 long clientdata[4];              /* 内部での使用のため予約 */
 }
```

```
  typedef struct clientid_t CLIENTID;
  /* クライアント識別子構造 */
  typedef long TPCONTEXT_T;
  /* サービス・ルーチンへのインターフェイス */
  struct tpsvcinfo {
  name[32];
  long flags;                       /* サービス属性の説明 */
  char *data;                       /* データを指すポインタ */
  long len;                         /* 要求データ長 */
  int cd;                           /* (flags TPCONV) が真のとき接続記述子 */

  long appkey;                      /* アプリケーション認証用のクライアント・
  * キー */
  CLIENTID cltid;                   /* 発行元クライアント用の
   * クライアント識別子 */
  };

 typedef struct tpsvcinfo TPSVCINFO;

 /* tpinit(3) インターフェイス構造 */
  #define MAXTIDENT                 30

struct tpinfo_t {
  char usrname[MAXTIDENT+2];        /* クライアント・ユーザ名 */
  char cltname[MAXTIDENT+2];        /* アプリケーション・クライアント名 */
  char passwd[MAXTIDENT+2];         /* アプリケーション・パスワード */
  long flags;                       /* 初期化フラグ */
  long datalen;                     /* アプリケーション固有のデータの長さ */
  long data;                        /* アプリケーション・データのプレースホルダ */
  };
  typedef struct tpinfo_t TPINIT;

 /* tpsuspend(3) と tpresume(3) に渡されるトランザクション ID 構造体 */
  struct tp_tranid_t {
  long info[6];                     /* 内部的に定義 */
  };

 typedef struct tp_tranid_t TPTRANID;

 /* TPINIT のフラグ */
  #define TPU_MASK                  0x00000007  /* 任意
                                                 * 通知型 */
  #define TPU_SIG                   0x00000001  /* シグナル・ベース
                                                 * の通知 */
  #define TPU_DIP                   0x00000002  /* ディップ・イン・ベース
                                                 * の通知 */
```

```
 #define TPU_IGN                    0x00000004  /* 任意通知型
                                                 * メッセージを無視 */
 #define TPU_THREAD                 0x00000040  /* THREAD 通知 */
 #define TPSA_FASTPATH              0x00000008  /* システム・アクセス ==
                                                 * fastpath */
 #define TPSA_PROTECTED             0x00000010  /* システム・アクセス ==
                                                 * protected */
 #define TPMULTICONTEXTS            0x00000020  /* 各プロセスの
                                                 * マルチ・コンテキスト関連 */
 /*  /Q tpqctl_t データ構造体                    */
 #define TMQNAMELEN                 15
 #define TMMSGIDLEN                 32
 #define TMCORRIDLEN                32

struct tpqctl_t {                   /* キュー・プリミティブの制御パラメータ */
 long flags;                        /* どの値が設定されているかの指示 */
 long deq_time;                     /* キューから取り出すときの絶対時間 / 相対時間 */
 long priority;                     /* 登録優先順位 */
 long diagnostic;                   /* 異常終了の原因 */
 char msgid[TMMSGIDLEN];            /* 既存メッセージの ID ( そのメッセージの前に登録するため ) */
 char corrid[TMCORRIDLEN];          /* メッセージを識別するときに使用される相関識別子 */
 char replyqueue[TMQNAMELEN+1];     /* 応答メッセージ用キューの名前 */
 char failurequeue[TMQNAMELEN+1]; /* 異常終了メッセージ用キューの名前 */
 CLIENTID cltid;                    /* 発信元クライアントの */
                                    /* クライアント識別子 */
 long urcode;                       /* アプリケーション・ユーザ戻り値 */
 long appkey;                       /* アプリケーション認証用のクライアント・キー */
 long delivery_qos;                 /* 配信サービスの品質 */
 long reply_qos;                    /* 応答メッセージのサービス品質 */
 long exp_time                      /* 有効期限 */
 };
 typedef struct tpqctl_t TPQCTL;

/* 有効な /Q 構造体要素 - フラグに設定 */
 #ifndef TPNOFLAGS
 #define TPNOFLAGS                  0x00000    /* フラグの設定なし―獲得できません */
 #endif
 #define TPQCORRID                  0x00001    /* 相関 id の設定 / 獲得 */
 #define TPQFAILUREQ                0x00002    /* 障害キューの設定 / 獲得 */
 #define TPQBEFOREMSGID             0x00004    /* メッセージ id の前にキューに登録 */
 #define TPQGETBYMSGIDOLD           0x00008    /* 使用回避 */
 #define TPQMSGID                   0x00010    /* enq/deq メッセージの msgid の獲得 */
 #define TPQPRIORITY                0x00020    /* メッセージ優先順位の設定 / 獲得 */
 #define TPQTOP                     0x00040    /* キューの先頭に登録 */
 #define TPQWAIT                    0x00080    /* キューからの解除を待機 */
```

```
 #define TPQREPLYQ               0x00100    /* 応答キューの設定 / 獲得 */
 #define TPQTIME_ABS             0x00200    /* 絶対時間の設定 */
 #define TPQTIME_REL             0x00400    /* 相対時間の設定 */
 #define TPQGETBYCORRIDOLD       0x00800    /* 使用回避 */
 #define TPQPEEK                 0x01000    /* 非破壊的なキューからの取り出し */
 #define TPQDELIVERYQOS          0x02000    /* 配信サービス品質 */
 #define TPQREPLYQOS             0x04000    /* 応答メッセージのサービス品質 */
 #define TPQEXPTIME_ABS          0x08000    /* 絶対有効期限 */
 #define TPQEXPTIME_REL          0x10000    /* 相対有効期限 */
 #define TPQEXPTIME_NONE         0x20000    /* 有効期限なし */
 #define TPQGETBYMSGID           0x40008    /* msgid によるキューからの取り出し */
 #define TPQGETBYCORRID          0x80800    /* corrid によるキューからの取り出し */

/* TPQCTL 構造のサービス品質フィールドのための有効フラグ */
 #define TPQQOSDEFAULTPERSIST    0x00001    /* キューのデフォルトの持続性 */
                                            /* 方針 */
 #define TPQQOSPERSISTENT        0x00002    /* ディスク・メッセージ */
 #define TPQQOSNONPERSISTENT     0x00004    /* メモリ・メッセージ */

/* エラー・リターン・コード */
 extern int tperrno;
 extern long tpurcode;

/* tperrno 値 - エラー・コード */
 * マニュアル・ページで、下記のエラー・コードが返される可能性のある
 * コンテキストに関する説明があります。
 */

 #define TPMINVAL                0          /* 最小のエラー・メッセージ */
 #define TPEABORT                1
 #define TPEBADDESC              2
 #define TPEBLOCK                3
 #define TPEINVAL                4
 #define TPELIMIT                5
 #define TPENOENT                6
 #define TPEOS                   7
 #define TPEPERM                 8
 #define TPEPROTO                9
 #define TPESVCERR               10
 #define TPESVCFAIL              11
 #define TPESYSTEM               12
 #define TPETIME                 13
 #define TPETRAN                 14
 #define TPGOTSIG                15
 #define TPERMERR                16
 #define TPEITYPE                17
 #define TPEOTYPE                18
```

```
 #define TPERELEASE              19
 #define TPEHAZARD               20
 #define TPEHEURISTIC            21
 #define TPEEVENT                22
 #define TPEMATCH                23
 #define TPEDIAGNOSTIC           24
 #define TPEMIB                  25
 #define TPMAXVAL                26          /* 最大のエラー・メッセージ */

/* 会話 - イベント */
 #define TPEV_DISCONIMM          0x0001
 #define TPEV_SVCERR             0x0002
 #define TPEV_SVCFAIL            0x0004
 #define TPEV_SVCSUCC            0x0008
 #define TPEV_SENDONLY           0x0020

/* /Q 診断コード */
 #define QMEINVAL                -1
 #define QMEBADRMID              -2
 #define QMENOTOPEN              -3
 #define QMETRAN                 -4
 #define QMEBADMSGID             -5
 #define QMESYSTEM               -6
 #define QMEOS                   -7
 #define QMEABORTED              -8
 #define QMENOTA                 QMEABORTED
 #define QMEPROTO                -9
 #define QMEBADQUEUE             -10
 #define QMENOMSG                -11
 #define QMEINUSE                -12
 #define QMENOSPACE              -13
 #define QMERELEASE              -14
 #define QMEINVHANDLE            -15
 #define QMESHARE                -16

/* イベント・ブローカ・メッセージ */
 #define TPEVSERVICE             0x00000001
 #define TPEVQUEUE               0x00000002
 #define TPEVTRAN                0x00000004
 #define TPEVPERSIST             0x00000008

/* サブスクリプション制御構造 */
 struct tpevctl_t {
     long flags;
     char name1[XATMI_SERVICE_NAME_LENGTH];
     char name2[XATMI_SERVICE_NAME_LENGTH];
     TPQCTL qctl;
```

```
  };
  typedef struct tpevctl_t TPEVCTL;
```

C 言語の TX の戻り値とその他の定義

TX ルーチンは、次に挙げる戻り値とフラグの定義を使用します。異なるトランザクション・モニタで変更や再コンパイルなしにアプリケーションを使用するためには、各システムで次に示すようにそのフラグと戻り値を定義しておかなければなりません。

```
#define TX_H_VERSION           0       /* このヘッダ・ファイルの
                                        * 現バージョン */

 /*
  * トランザクション識別子
  */
  #define XIDDATASIZE          128     /* サイズ ( バイト数 ) */
  struct xid_t {
         long formatID;                /* 形式識別子 */
         long gtrid_length;            /* 64 以下の値 */
         long bqual_length;            /* 64 以下の値 */
         char data[XIDDATASIZE];
  };
  typedef struct xid_t XID;
  /*
  * 形式識別子が -1 の場合は、XID がヌルであることを意味する
  */

 /*
  * tx_ ルーチンの定義
  */
  /* commit の戻り値 */
  typedef long COMMIT_RETURN;
  #define TX_COMMIT_COMPLETED 0
  #define TX_COMMIT_DECISION_LOGGED 1

 /* トランザクション制御の値 */
  typedef long TRANSACTION_CONTROL;
  #define TX_UNCHAINED 0
  #define TX_CHAINED 1

 /* トランザクション・タイムアウトのタイプ */
  typedef long TRANSACTION_TIMEOUT;

 /* トランザクションの状態値 */
  typedef long TRANSACTION_STATE;
  #define TX_ACTIVE 0
  #define TX_TIMEOUT_ROLLBACK_ONLY 1
  #define TX_ROLLBACK_ONLY 2
```

```
/* tx_info() で格納される構造体 */
 struct tx_info_t {
        XID xid;
        COMMIT_RETURN when_return;
        TRANSACTION_CONTROL transaction_control;
        TRANSACTION_TIMEOUT transaction_timeout;
        TRANSACTION_STATE   transaction_state;
 };
 typedef struct tx_info_t TXINFO;


/*
 * tx_() の戻り値
 * ( トランザクション・マネージャがアプリケーションに報告する )
 */
 #define TX_NOT_SUPPORTED            1 /* サポートされていないオプション */
 #define TX_OK                       0 /* 正常実行 */
 #define TX_OUTSIDE                 -1 /* アプリケーションは、リソース・マネージャの
                                        * ローカル・トランザクションにある */
 #define TX_ROLLBACK                -2 /* トランザクションが
                                        * ロールバックされた */
 #define TX_MIXED                   -3 /* トランザクションが
                                        * 部分的にコミットされ、
                                        * 部分的にロールバックされた */
 #define TX_HAZARD                  -4 /* トランザクションが
                                        * 部分的にコミットされ、
                                        * 部分的にロールバックされた可能性がある */
 #define TX_PROTOCOL_ERROR          -5 /* ルーチンが不適切な
                                        * コンテキストで呼び出された */
 #define TX_ERROR                   -6 /* 一時的なエラー  */
 #define TX_FAIL                    -7 /* 致命的なエラー */
 #define TX_EINVAL                  -8 /* 無効な引数が指定された */
 #define TX_COMMITTED               -9 /* トランザクションが
                                        * ヒューリスティックにコミットされた */

 #define TX_NO_BEGIN                -100 /* トランザクションがコミットされたことに加え、
                                          * 新しいトランザクションが
                                          * 開始できなかった */
 #define TX_ROLLBACK_NO_BEGIN       (TX_ROLLBACK+TX_NO_BEGIN)
                                       /* トランザクションがロールバックされたことに加え、
                                          * 新しいトランザクションが
                                          * 開始できなかった */
 #define TX_MIXED_NO_BEGIN          (TX_MIXED+TX_NO_BEGIN)
                                       /* 混合条件が発生したことに加え、
                                          * 新しいトランザクションが開始できなかった */
```

```
  #define TX_HAZARD_NO_BEGIN          (TX_HAZARD+TX_NO_BEGIN)
                                        /* ハザードがあることに加え、
                                         * 新しいトランザクションが開始できなかった */
  #define TX_COMMITTED_NO_BEGIN       (TX_COMMITTED+TX_NO_BEGIN)
                                       /* トランザクションがヒューリスティックにコミットされ
たことに加え、
                                         * 新しいトランザクションが
                                         * 開始できなかった */
```

ATMI の状態遷移

BEA Tuxedo ATMI システムは、各プロセスの状態を記録し、各種の関数呼び出しやオプションごとに正当な状態遷移が行われているかどうかを検証します。この状態情報には、プロセスのタイプ (要求 / 応答型サーバ、会話型サーバ、またはクライアント)、初期化状態 (初期化済み、非初期化)、リソースの管理状態 (クローズまたはオープン)、プロセスのトランザクション状態およびすべての非同期要求および接続記述子の状態などがあります。不正な状態遷移が行われようとすると、呼び出された関数は異常終了し、tperrno() が TPEPROTO に設定されます。この情報に関する正規の状態と遷移について、次の表に示します。

表 4 は、要求 / 応答型サーバ、会話サーバおよびクライアントがどの関数を呼び出すことができるかを示しています。ただし、tpsvrinit()、tpsvrdone()、tpsvrthrinit()、および tpsvrthrdone() はこの表には示してありません。これらの関数はアプリケーションが提供する関数ですが、アプリケーションからは呼び出されず、BEA Tuxedo ATMI システムによって呼び出されます。

表 4 使用できる関数

	プロセス・タイプ		
関数	**要求 / 応答型サーバ**	**会話型サーバ**	**クライアント**
tpabort	Y	Y	Y
tpacall	Y	Y	Y
tpadvertise	Y	Y	N
tpalloc	Y	Y	Y
tpbegin	Y	Y	Y
tpbroadcast	Y	Y	Y
tpcall	Y	Y	Y
tpcancel	Y	Y	Y

表 4 使用できる関数 (続き)

関数	プロセス・タイプ		
	要求 / 応答型サーバ	会話型サーバ	クライアント
tpchkauth	Y	Y	Y
tpchkunsol	N	N	Y
tpclose	Y	Y	Y
tpcommit	Y	Y	Y
tpconnect	Y	Y	Y
tpdequeue	Y	Y	Y
tpdiscon	Y	Y	Y
tpenqueue	Y	Y	Y
tpforward	Y	N	N
tpfree	Y	Y	Y
tpgetctxt	Y	Y	Y
tpgetlev	Y	Y	Y
tpgetrply	Y	Y	Y
tpgprio	Y	Y	Y
tpinit	N	N	Y
tpnotify	Y	Y	Y
tpopen	Y	Y	Y
tppost	Y	Y	Y
tprealloc	Y	Y	Y
tprecv	Y	Y	Y
tpresume	Y	Y	Y
tpreturn	Y	Y	N

表 4 使用できる関数 (続き)

関数	プロセス・タイプ		
	要求 / 応答型サーバ	会話型サーバ	クライアント
tpscmt	Y	Y	Y
tpsend	Y	Y	Y
tpservice	Y	Y	N
tpsetctxt	Y (アプリケーション生成スレッドの場合)	Y (アプリケーション生成スレッドの場合)	Y
tpsetunsol	N	N	Y
tpsprio	Y	Y	Y
tpsubscribe	Y	Y	Y
tpsuspend	Y	Y	Y
tpterm	N	N	Y
tptypes	Y	Y	Y
tpunadvertise	Y	Y	N
tpunsubscribe	Y	Y	Y

以下に示す表は、特に明記されていないかぎり、クライアントとサーバ両方に適用されます。なお、ある種の関数はクライアントとサーバの両方が呼び出せるとは限らないので（例 : tpinit()、以下の状態遷移の中には両方のプロセス・タイプには適用できないものがあります）上記の表を参照して、目的のプロセスから特定の関数を呼び出すことができるかどうかを判断するようにしてください。

次の表は、クライアント・プロセスがトランザクション・マネージャで初期化され登録されているかどうかを示しています。この表では、シングルコンテキスト・モードのオプションとして tpinit() を使用するものとします。したがって、シングルコンテキストのクライアントは、多数の ATMI 関数の中のどれか (たとえば、tpconnect() または tpcall()) を発行することによって、暗黙のうちにアプリケーションを結合することができます。次のいずれかにあてはまる場合は、クライアントは tpinit() を使用しなければなりません。

- アプリケーション認証が必須の場合。詳細については、tpinit(3c) および UBBCONFIG(5) の SECURITY キーワードの説明を参照。

- クライアントが、XA 準拠のリソース・マネージャに直接アクセスする場合。詳細については tpinit(3c) を参照。
- クライアントが複数のアプリケーション関連を作成する場合。

サーバは tpsvrinit() 関数が呼び出される前に BEA Tuxedo ATMI システムによって初期化状態になり、tpsvrdone() 関数が返された後、BEA Tuxedo ATMI システムの main() によって非初期化状態になります。なお、下記のすべての表において、関数がエラーを起こした場合、特に明記されていないかぎり、プログラムの状態は変わりません。

表 5 スレッド初期化状態

関数	状態	
	非初期化 I_0	初期化 I_1
tpalloc	I_0	I_1
tpchkauth	I_0	I_1
tpfree	I_0	I_1
tpgetctxt	I_0	I_1
tpinit	I_1	I_1
tprealloc	I_0	I_1
tpsetctxt (ヌル以外のコンテキストに設定)	I_1	I_1
tpsetctxt (TPNULLCONTEXT コンテキスト設定)	I_0	I_0
tpsetunsol	I_0	I_1
tpterm (このスレッドで)	I_0	I_0
tpterm (このコンテキストの別のスレッドで)	I_0	I_0

表 5 スレッド初期化状態 (続き)

関数	状態	
	非初期化 I_0	初期化 I_1
tptypes	I_0	I_1
他のすべての ATMI 関数	I_1	I_1

以降の表は、前提条件として状態が I_1 であると想定しています (tpinit()、tpsetctxt()、または BEA Tuxedo ATMI システムの main() を介してこの状態でプロセスが到着したかどうかに関わりなく)。

表 6 は、クライアントまたはサーバのプロセスに対応するリソース・マネージャが初期化されているかいないかに応じて、そのプロセスの状態を示しています。

表 6 リソース・マネージャの状態

関数	状態	
	クローズ R_0	オープン R_1
tpopen	R_1	R_1
tpclose	R_0	R_0
tpbegin		R_1
tpcommit		R_1
tpabort		R_1
tpsuspend		R_1
tpresume		R_1
フラグ TPTRAN オンの tpservice		R_1
他のすべての ATMI 関数	R_0	R_1

表 7 は、プロセスがトランザクションに対応しているかどうかに関してそのプロセスの状態を示したものです。サーバの場合、状態 T_1 と T_2 への遷位は、事前条件として状態 R_1 を想定しています (たとえば、tpopen() はそれ以降 tpclose() または tpterm() への呼び出しがないものとして呼び出されています)。

表 7 アプリケーション関連のトランザクション状態

	状態		
関数	トランザクション内ではない T_0	イニシエータ T_1	構成要素 T_2
tpbegin			
tpabort		T_0	
tpcommit		T_0	
tpsuspend		T_0	
tpresume	T_1	T_0	
フラグ TPTRAN オンの tpservice	T_2		
tpservice (トランザクション・モードでない)	T_0		
tpreturn	T_0		T_0
tpforward	T_0		T_0
tpclose	R_0		
tpterm	I_0	T_0	
他のすべての ATMI 関数	T_0	T_1	T_2

表 8 は、tpacall() が返す 1 つの要求記述子の状態を示すものです。

表 8 非同期要求記述子の状態

関数	状態	
	記述子なし A_0	有効な記述子 A_1
tpacall	A_1	
tpgetrply		A_0
tpcancel		A_0 [a]
tpabort	A_0	A_0 [b]
tpcommit	A_0	A_0 [b]
tpsuspend	A_0	A_1 [c]
tpreturn	A_0	A_0
tpforward	A_0	A_0
tpterm	I_0	I_0
他のすべての ATMI 関数	A_0	A_1

注記 [a] この状態遷移は、記述子が呼び出し元のトランザクションに対応しない場合にのみ起こります。

[b] この状態遷移は、記述子が呼び出し元のトランザクションに対応する場合にのみ起こります。

[c] 記述子が呼び出し元のトランザクションに対応する場合、tpsuspend() はプロトコル・エラーを返します。

表 9 は、tpconnect() が返す、あるいは TPSVCINFO 構造でサービス呼び出しを行うことによって得られる接続記述子の状態を示したものです。接続記述子をとらないプリミティブの場合、特に明記されていないかぎり、状態の変化はすべての接続記述子に適用されます。

状態には次のものがあります。

- C_0 — 記述子なし
- C_1 — tpconnect() 記述子送信専用

- C_2 — tpconnect() 記述子受信専用
- C_3 — TPSVCINFO 記述子送信専用
- C_4 — TPSVCINFO 記述子受信専用

表 9 接続要求記述子の状態

関数 / イベント	状態				
	C_0	C_1	C_2	C_3	C_4
TPSENDONLY 指定の tpconnect	C_1 [a]				
TPRECVONLY 指定の tpconnect	C_2 [a]				
フラグ TPSENDONLY 指定の tpservice	C_3 [b]				
フラグ TPRECVONLY 指定の tpservice	C_4 [b]				
tprecv/ イベントなし			C_2		C_4
tprecv/TPEV_SENDONLY			C_1		C_3
tprecv/TPEV_DISCONIMM			C_0		C_0
tprecv/TPEV_SVCERR			C_0		
tprecv/TPEV_SVCFAIL			C_0		
tprecv/TPEV_SVCSUCC			C_0		
tpsend/ イベントなし		C_1		C_3	
TPRECVONLY 指定の tpsend		C_2		C_4	
tpsend/TPEV_DISCONIMM		C_0		C_0	
tpsend/TPEV_SVCERR		C_0			
tpsend/TPEV_SVCFAIL		C_0			
tpterm (クライアントのみ)	C_0	C_0			

表 9 接続要求記述子の状態 (続き)

関数 / イベント	状態				
	C_0	C_1	C_2	C_3	C_4
tpcommit (オリジネータのみ)	C_0	C_0 [c]	C_0 [c]		
tpsuspend (オリジネータのみ)	C_0	C_1 [d]	C_2 [d]		
tpabort (オリジネータのみ)	C_0	C_0 [c]	C_0 [c]		
tpdiscon		C_0	C_0		
tpreturn (CONV サーバ)		C_0	C_0	C_0	C_0
tpforward (CONV サーバ)		C_0	C_0	C_0	C_0
他のすべての ATMI 関数	C_0	C_1	C_2	C_3	C_4

注記 [a] プログラムがトランザクション・モードにあり、かつ TPNOTRAN の指定がない場合は、接続はトランザクション・モードになります。

[b] TPTRAN が設定されていると、接続はトランザクション・モードになります。

[c] 接続がトランザクション・モードにないと、状態は変化しません。

[d] 接続がトランザクション・モードの場合、tpsuspend() はプロトコル・エラーを返します。

TX の状態遷移

BEA Tuxedo ATMI システムでは、プロセスが必ず TX 関数を正しい順序で呼び出すことが確認されます。不正の状態遷移が試行されると (つまり、ブランクの遷移エントリの状態からの呼び出し)、呼び出された関数は、TX_PROTOCOL_ERROR を返します。TX 関数の正当な状態と遷移を、表 10 に示します。異常終了を返す呼び出しの場合、この表で特に明記されていないかぎり、状態遷移は行われません。BEA Tuxedo ATMI システムのクライアントまたはサーバはすべて、TX 関数を使用できます。

状態は、次のように定義されています。

- S_0: どの RM もオープンまたは初期化が行われていません。アプリケーション関連は、tx_open を正常に呼び出すまで、グローバル・トランザクションを開始できません。
- S_1: アプリケーション関連は RM をオープンしましたが、トランザクションには入っていません。transaction_control 特性は、TX_UNCHAINED です。

- S_2: アプリケーション関連は RM をオープンしましたが、トランザクションには入っていません。transaction_control 特性は、TX_CHAINED
- S_3: アプリケーション関連は RM をオープンし、トランザクションに入っています。transaction_control 特性は、TX_UNCHAINED
- S_4: アプリケーション関連は RM をオープンし、トランザクションに入っています。transaction_control 特性は、TX_CHAINED

表 10 TX 関数の状態遷移

関数	状態				
	S_0	S_1	S_2	S_3	S_4
tx_begin		S_3	S_4		
tx_close	S_0	S_0	S_0		
tx_commit -> TX_SET1				S_1	S_4
tx_commit -> TX_SET2					S_2
tx_info		S_1	S_2	S_3	S_4
tx_open	S_1	S_1	S_2	S_3	S_4
tx_rollback -> TX_SET1				S_1	S_4
tx_rollback -> TX_SET2					S_2
tx_set_commit_return		S_1	S_2	S_3	S_4
tx_set_transaction_control control = TX_CHAINED		S_2	S_2	S_4	S_4
tx_set_transaction_control control = TX_UNCHAINED		S_1	S_1	S_3	S_3
tx_set_transaction_timeout		S_1	S_2	S_3	S_4

- TX_SET1 は、TX_OK、TX_ROLLBACK、TX_MIXED、TX_HAZARD、または TX_COMMITTED のいずれかを表します。TX_ROLLBACK は、tx_rollback() からは返されず、TX_COMMITTED は、tx_commit() からは返されません。

- TX_SET2 は、TX_NO_BEGIN、TX_ROLLBACK_NO_BEGIN、TX_MIXED_NO_BEGIN、TX_HAZARD_NO_BEGIN、または TX_COMMITTED_NO_BEGIN のいずれかを表します。TX_ROLLBACK_NO_BEGIN は、tx_rollback() からは返されず、TX_COMMITTED_NO_BEGIN は、tx_commit() からは返されません。
- TX_FAIL は、いずれかの呼び出しから返された場合には、アプリケーション・プログラムの状態は、上記の表では未定義の状態になります。
- tx_info() が、トランザクション状態情報に TX_ROLLBACK_ONLY または TX_TIMEOUT_ROLLBACK_ONLY を返した場合には、そのトランザクションは、「アボートのみ (ロールバックのみ)」とマークされ、アプリケーション・プログラムが tx_commit() を呼び出したか tx_rollback() を呼び出したかに関係なく、ロールバック (アボート) されます。

関連項目

buffer(3c)、tpadvertise(3c)、tpalloc(3c)、tpbegin(3c)、tpcall(3c)、tpconnect(3c)、tpgetctxt(3c)、tpinit(3c)、tpopen(3c)、tpservice(3c)、tpsetctxt(3c)、tuxtypes(5)、typesw(5)

AEMsetblockinghook(3c)

名前

AEMsetblockinghook()— アプリケーション固有のブロッキング・フック関数を確立します。

形式

```
#include <atmi.h>
int AEMsetblockinghook(_TM_FARPROC)
```

機能説明

AEMsetblockinghook() は、Mac のタスクで ATMI ネットワーキング・ソフトウェアがブロッキング ATMI 呼び出しを実装するための新しい関数をインストールできる「Mac 対応 ATMI エクステンション」です。この関数には、インストールするブロッキング関数の関数アドレスへのポインタが必要です。

ATMI ブロッキング呼び出しを処理する、デフォルトの関数はすでに用意されています。AEMsetblockinghook() 関数によって、アプリケーションは、「ブロッキング」時にデフォルト関数の代わりに独自の関数を実行することができます。NULL ポインタを指定してこの関数を呼び出すと、ブロッキング・フック関数はデフォルトの関数にリセットされます。

アプリケーションが ATMI ブロッキング操作を呼び出すと、ブロッキング操作が起動し、次の疑似コードで示すようなループが開始します。

```
for(;;) {
        execute operation in non-blocking mode
        if error
               break;
        if operation complete
               break;
        while(BlockingHook())
                 ;
}
```

戻り値

AEMsetblockinghook() は、以前にインストールされたブロッキング関数のプロシージャ・インスタンスへのポインタを返します。AEMsetblockinghook() 関数を呼び出すアプリケーションまたはライブラリは、必要に応じて復元できるように、この戻り値を保存する必要があります。「入れ子」が重要でない場合は、アプリケーションは AEMsetblockinghook() によって返された値を廃棄し、最終的に AEMsetblockinghook (ヌル) を使ってデフォルト・メカニズムを復元することができます。AEMsetblockinghook() はエラー時にはヌルを返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、AEMsetblockinghook() は tperrno() を次の値に設定します。

[TPEPROTO]
AEMsetblockinghook() が、ブロッキング操作の実行中に呼び出されました。

移植性

このインターフェイスは、Mac クライアントでのみサポートされています。

注意事項

アプリケーションが tpterm(3c) を呼び出すと、ブロッキング関数はリセットされます。

AEOaddtypesw(3c)

名前

AEOaddtypesw()— 実行時にユーザ定義のバッファ・タイプをインストールまたはリプレースします。

形式

```
#include <atmi.h>
#include <tmtypes.h>

int FAR PASCAL AEOaddtypesw(TMTYPESW *newtype)
```

機能説明

AEOaddtypesw() は、OS/2 クライアントが実行時に、新しいユーザ定義のバッファ・タイプをインストールするか、既存のユーザ定義のバッファ・タイプをリプレースするための、「OS/2 の ATMI 拡張機能」です。この関数の引数は、インストールするバッファ・タイプに関する情報を含む、TMTYPESW 構造体を指すポインタです。

type() と subtype() がすでにインストールされているバッファ・タイプと一致する場合は、すべての情報が新しいバッファ・タイプにリプレースされます。情報が type() フィールドおよび subtype() フィールドと一致しない場合は、BEA Tuxedo ATMI システムによって登録された既存のバッファ・タイプに、新しいバッファ・タイプが追加されます。新しいバッファ・タイプの場合は、呼び出し処理に含まれる WSH および他の BEA Tuxedo ATMI システムのプロセスが新しいバッファ・タイプで作成されているようにします。

TMTYPESW 配列内の関数ポインタは、EXPORTS セクションにあるアプリケーションのモジュール定義ファイルに表示されていなければなりません。

アプリケーションでは、BEA Tuxedo ATMI システム定義のバッファ・タイプのルーチンも使用できます。また、アプリケーションおよび BEA Tuxedo ATMI システム・バッファ・ルーチンを、あるユーザ定義のバッファ・タイプ内で混在させることができます。

戻り値

AEOaddtypesw() は、正常終了時には、システム内のユーザ・バッファ・タイプの数を返します。異常終了時には -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、AEOaddtypesw() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
AEOaddtypesw() が呼び出され、type パラメータは NULL でした。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

移植性　このインターフェイスは、Windows クライアントでのみサポートされます。タイプ・スイッチをインストールする場合は、BEA Tuxedo ATMI システム・タイプスイッチ DLL に追加することをお勧めします。詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

注意事項　FAR PASCAL は、16 ビット OS/2 環境でのみ使用します。

使用例

```
#include <os2.h>
#include <atmi.h>
#include <tmtypes.h>

int FAR PASCAL Nfinit(char FAR *, long);
int (FAR PASCAL * lpFinit)(char FAR *, long);
int FAR PASCAL Nfreinit(char FAR *, long);
int (FAR PASCAL * lpFreinit)(char FAR *, long);
int FAR PASCAL Nfuninit(char FAR *, long);
int (FAR PASCAL * lpFuninit)(char FAR *, long);

TMTYPESW        newtype =
{
"MYFML",          ",             1024,           NULL,           NULL,
NULL,          _fpresend,     _fpostsend,     _fpostrecv,     _fencdec,
_froute
};

   newtype.initbuf = Nfinit;
   newtype.reinitbuf = Nfreinit;
   newtype.uninitbuf = Nfuninit;

   if(AEOaddtypesw(newtype) == -1) {
           userlog("AEOaddtypesw failed %s", tpstrerror(tperrno));
   }
int
FAR PASCAL
Nfinit(char FAR *ptr, long len)
{
      ......
      return(1);
}

int
FAR PASCAL
Nfreinit(char FAR *ptr, long len)
{
      ......
      return(1);
}
```

```
int
FAR PASCAL
Nfuninit(char FAR *ptr, long mdlen)
{
       ......
       return(1);
}
```

アプリケーション・モジュール定義ファイル:

```
; EXAMPLE.DEF file

NAME         EXAMPLE

DESCRIPTION  'EXAMPLE for OS/2'

EXETYPE      OS/2


EXPORTS
        Nfinit
        Nfreinit
        Nfuninit
         ....
```

関連項目 buildwsh(1)、buffer(3c)、typesw(5)

AEPisblocked(3c)

名前　　AEPisblocked()— 進行中のブロッキング呼び出しが存在するかどうかの確認

形式

```
#include <atmi.h>
int far pascal AEPisblocked(void)
```

機能説明　　AEPisblocked() は、OS/2 プレゼンテーション・マネージャ用拡張 ATMI 関数の 1 つです。この関数を使用することにより、OS/2 プレゼンテーション・マネージャのタスクは、タスクの実行が前のブロッキング呼び出しの完了を待っている最中にあるかどうかを判断できます。

戻り値　　AEPisblocked() は、完了待ちのブロッキング関数が存在する場合は 1 を、それ以外の場合は 0 を返します。

エラー　　エラーは返されません。

移植性　　このインターフェイスは、OS/2 プレゼンテーション・マネージャのクライアントにおいてのみサポートされます。

備考　　ATMI ブロッキング呼び出しは、アプリケーションから見ると「ブロック」しているように見えますが、OS/2 PM ATMI DLL は、プロセッサの制御権を放棄して他のアプリケーションが実行できるようにしなければなりません。このことは、ブロッキング呼び出しを発行したアプリケーションが、受信メッセージによって再入する可能性があることを意味します。このような場合は、AEPisblocked() 関数を使用すると、タスクが再入したのが未終了のブロッキング呼び出しの完了を待っている最中だったかどうかを確認できます。ATMI では、未終了の呼び出しが単一スレッド内に 2 つ以上存在することは禁止されているので、注意してください。

関連項目　　AEMsetblockinghook(3c)

AEWsetunsol(3c)

名前

AEWsetunsol()—BEA Tuxedo ATMI 任意イベントに対する Windows メッセージの掲示

形式

```
#include <windows.h>
#include <atmi.h>
int far pascal AEWsetunsol(HWND hWnd, WORD wMsg)
```

機能説明

Microsoft Windows のプログラミング環境によっては、BEA Tuxedo ATMI システムの任意通知型メッセージを Windows イベント・メッセージ・キューに送った方がよい場合があります。

AEWsetunsol() は、どのウィンドウに通知を行うか (*hWnd*)、またどの Windows メッセージ・タイプを掲示するか (*wMsg*) を制御します。BEA Tuxedo ATMI の任意通知型メッセージが到達すると、Windows のメッセージが掲示されます。lParam() は BEA Tuxedo ATMI システムのバッファ・ポインタに設定されます。メッセージがなければ、ゼロに設定されます。lParam() がゼロでなければ、アプリケーションは tpfree() を呼び出し、バッファを解放する必要があります。

wMsg がゼロであれば、その後の任意通知型メッセージはログに入れられ、無視されます。

マルチスレッドのアプリケーションでは、TPINVALIDCONTEXT 状態のスレッドは、AEWsetunsol() 呼び出しを発行することはできません。

戻り値

AEWsetunsol() は、エラー発生時には -1 を返し、エラー条件を示す tperrno() を設定します。

エラー

異常終了時には、AEWsetunsol() は tperrno() を次のいずれかの値に設定します。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

移植性

このインターフェイスは、Microsoft Windows クライアントでしか利用できません。

注意事項

Windows メッセージを掲示する AEWsetunsol() は、tpsetunsol() コールバック・ルーチンと同時に起動することはできません。最後の tpsetunsol() 要求あるいは AEWsetunsol() 要求が、任意通知型メッセージを処理する方法を決定します。

関連項目

tpsetunsol(3c)

buffer(3c)

名前　　buffer()—tmtype_sw_t における要素の意味

形式

```
int    /* 新しいデータ・バッファの初期化 */
_tminitbuf(char *ptr, long len)
int    /* 再割り当てされたデータ・バッファ */
_tmreinitbuf(char *ptr, long len)
int    /* 解放するデータ・バッファの初期化解除 */
_tmuninitbuf(char *ptr, long len)
long   /* 送信前のバッファの処理 */
_tmpresend(char *ptr, long dlen, long mdlen)
void   /* 送信後のバッファの処理 */
_tmpostsend(char *ptr, long dlen, long mdlen)
long   /* 受信後のバッファの処理 */
_tmpostrecv(char *ptr, long dlen, long mdlen)
long   /* 伝送形式とバッファの間の符号化 / 復号化 */
_tmencdec(int op, char *encobj, long elen, char *obj, long olen)
int    /* データに基づきルーティングのサーバ・グループを決定 */  _tmroute(char
*routing_name, char *service, char *data, long \  len, char *group)
int    /* バッファのデータのブール式を評価 */  _tmfilter(char *ptr,
long dlen, char *expr, long exprlen)
int    /* 形式文字列に基づくバッファのデータの抽出 */  _tmformat(char *ptr,
long dlen, char *fmt, char *result, long \  maxresult)
long   /* 送信前、おそらくコピー生成前のバッファの処理 */
_tmpresend2(char *iptr, long ilen, long mdlen, char *optr, long olen, int *flags )
```

機能説明

このマニュアル・ページでは、tmtype_sw_t 構造体に定義されている要素とルーチンの意味について説明します。ここでの説明は、プロセスのバッファ・タイプ・スイッチ tm_typesw に新しいバッファ・タイプを追加するときに必要になります。スイッチ・エレメントは typesw(5) に定義されています。また、このエントリで使用されている関数名は、BEA Tuxedo ATMI システムによって定義されている実際の関数名、および独自のバッファ・タイプを追加するアプリケーションで定義されている関数名のテンプレートです。これらの名前は、非常に簡単にスイッチ・エレメントに対応付けられます。テンプレート名は、各関数ポインタの要素名とその前に付く _tm で構成されます (たとえば、要素 initbuf の関数名は _tminitbuf() になります)。

要素 type は NULL 以外とし、最大 8 文字でなければなりません。また、要素 subtype には NULL、最大 16 文字の文字列、またはワイルドカード文字 "*" を指定できます。type がスイッチ内で一意でない場合、subtype を使用しなければなりません。type と subtype の組み合わせは、スイッチ内で一意に要素を指定するものでなければなりません。

1 つのタイプに対して、複数のサブタイプが存在してもかまいません。すべてのサブタイプを特定のタイプについて同じように扱う場合には、ワイルドカード文字 "*" を使用することができます。ただし、サブタイプを区別する必要がある場合には、関数 tptypes() を使用することでバッファのタイプとサブタイプを弁別できます。ある特定のタイプ内で一定のサブタイプのサブセットを個別に扱う必要があり、残りを同様に扱う場合には、特定の値でまとめるサブタイプはそのサブタイプをワイルドカードで指定する前にスイッチ内に指定しておく必要があります。このため、まずスイッチ内のタイプとサブタイプの検索が上から下の方向に行われ、ワイルドカードによるサブタイプのエントリは残りの一致するタイプを受け付けることになります。

要素 dfltsize() は、バッファの割り当てまたは再割り当てを行うときに使用します。tpalloc() と tprealloc() は、dfltsize() およびこれらのルーチンの *size* パラメータのうち大きい方を使用してバッファの作成または再割り当てを行うことを意味します。固定サイズの C 構造体などの場合、バッファ・サイズはその構造体と同じにするべきです。dfltsize() をこの値に設定すると、呼び出し元はバッファが渡されるルーチンに対してそのバッファの長を指定する必要はなくなります。dfltsize() は 0 あるいはそれ以下にすることができます。ただし、tpalloc() や tprealloc() を呼び出して、その *size* パラメータも 0 もしくはそれ以下であると、このルーチンは異常終了します。このため、dfltsize() を 0 以下の値に設定することはお勧めできません。

ルーチンの詳細

以下に指定する関数の名前は、BEA Tuxedo ATMI システム内で使用されるテンプレート名です。新しいルーチンをバッファ・タイプ・スイッチに追加するアプリケーションは、そのアプリケーションあるいはライブラリ・ルーチンが提供する実際の関数に対応する名前を使用しなければなりません。NULL 関数ポインタがバッファ・タイプ・スイッチ・エントリに格納されている場合、BEA Tuxedo ATMI システムは正しい数と引数タイプをとり、デフォルトの値を返すデフォルト関数を呼び出します。

_tminitbuf()

_tminitbuf() は、バッファの割り当て後、tpalloc() の中から呼び出されます。_tminitbuf() は新しいバッファを指すポインタ *ptr()* とそのサイズを渡されるので、適切にバッファを初期化することができます。*len()* は tpalloc() に渡されるより大きな長さであり、かつそのタイプのスイッチ・エントリに dfltsize() で指定されるデフォルト値です。なお、*ptr()* は、tpalloc() と tprealloc() の意味合いから NULL には決してなりません。正常終了すると、そのデータ・ポインタが tpalloc() の呼び出し元に返されます。

複数のサブタイプの処理に 1 つのスイッチ・エントリを使用する場合、_tminitbuf() の作成者は tptypes() を使用してそのサブタイプを特定することができます。

バッファを初期化しなおす必要がない場合には、NULL 関数ポインタを使用します。

正常終了の場合、_tminitbuf() は 1 を返します。異常終了の場合には、-1 を返し、これによって tpalloc() も異常終了を示す tperrno() を TPESYSTEM に返します。

_tmreinitbuf

_tmreinitbuf() はほとんど _tminitbuf() と同様に働きます。ただし、この関数は再割り当てされたバッファを初期化しなおすときに使用します。このルーチンは、バッファの再割り当ての後、tprealloc() の中から呼び出されます。

バッファを初期化しなおす必要がない場合には、NULL 関数ポインタを使用します。

正常終了の場合、_tmreinitbuf() は 1 を返します。異常終了の場合には、-1 を返し、これにより tprealloc() も異常終了を示す tperrno() を TPESYSTEM に返します。

_tmuninitbuf

_tmuninitbuf() は、データ・バッファの解放前に tpfree() から呼び出されます。_tmuninitbuf() には、データ・バッファのアプリケーション部分を指すポインタとそのサイズが渡されるので、これを使用してそのバッファに関連する構造体や状態情報を消去することができます。*ptr()* は、tpfree() のもつ意味合いから決して NULL にはなりません。ただし、_tmuninitbuf() でバッファ自体を解放しないようにしてください。tpfree() 関数は、データ・バッファのすべての FLD_PTR フィールドで自動的に呼び出されます。

バッファを解放する前に何も処理が必要とされない場合には、NULL 関数ポインタを使用してください。

正常終了の場合、_tmuninitbuf() は 1 を返します。異常終了の場合には、-1 が返され、tpfree() はログ・メッセージを出力します。

_tmpresend

_tmpresend() は、tpcall()、tpacall()、tpconnect()、tpsend()、tpbroadcast()、tpnotify()、tpreturn()、あるいは tpforward() でバッファが送られる前に呼び出されます。また、_tmroute() の後、ただし _tmencdec() の前にも呼び出されます。*ptr()* がヌルでない場合、このルーチンにより、バッファの送信前にバッファに対する前処理を行うことができます。_tmpresend() の最初の引数 *ptr()* は、送信呼び出しに渡されるアプリケーション・データ・バッファです。2 番目の引数 *dlen()* は、送信呼び出しに渡されるデータの長さです。また、3 番目の引数 *mdlen()* は、データがおかれるバッファの実際の長さです。

この関数に関する重要な条件の 1 つは、関数が返るときに、*ptr()* が指すデータが「そのまま」送られるようにすることです。つまり、_tmencdec() が呼び出されるのはバッファが異なるマシンに送られる場合だけであるので、_tmpresend() は戻る際に、*ptr()* が指すバッファのどの要素も、そのバッファに隣接しないデータを指すポインタとならないようにしなければならないのです。

データに対して前処理が必要なく、呼び出し元が指定したデータ量が送信すべきデータ量と同じ場合、NULL 関数ポインタを使用します。デフォルトのルーチンは *dlen()* を返し、バッファに対して何もしません。

_tmpresend2() がヌル以外の場合、_tmpresend() は呼び出されず、代わりに _tmpresend2() が呼び出されます。

正常終了の場合、_tmpresend() は送信するデータ量を返します。異常終了の場合には、-1 を返し、_tmpresend() の呼び出し元も異常終了を示す tperrno() を TPESYSTEM に返します。

_tmpostsend()

_tmpostsend() は、tpcall()、tpbroadcast()、tpnotify()、tpacall()、tpconnect()、または tpsend() でバッファが送信された後呼び出されます。このルーチンにより、バッファの送信後、関数が戻る前にバッファに対して後処理を行うことができます。送信呼び出しに渡されるバッファは戻り時に異なっていてはならないので、_tmpostsend() を呼び出して、_tmpresend() によってなされたバッファへの変更を修復します。この関数の最初の引数 *ptr()* は、_tmpresend() の実行結果として送信されたデータを指すポインタです。2 番目の引数 *dlen()* は、_tmpresend() から返されるデータの長さです。3 番目の引数 *mdlen()* は、データがおかれるバッファの実際のサイズです。このルーチンは、*ptr()* が NULL 以外の場合にのみ呼び出されます。

後処理が必要ない場合には、NULL 関数ポインタを指定します。

_tmpostrecv

_tmpostrecv() は、バッファを受信した後、おそらく tpgetrply()、tpcall()、tprecv()、あるいは BEA Tuxedo ATMI システムのサーバ用の関数で復号化された後、アプリケーションに返される前に呼び出されます。*ptr()* が NULL でない場合は、_tmpostrecv() により、バッファが受信された後、アプリケーションに渡される前にそのバッファに対して後処理を行うことができます。最初の引数 *ptr()* は、受信したバッファのデータ部分を指すポインタです。2 番目の引数 *dlen()* は、_tmpostrecv() に入るデータのサイズを指定します。また、3 番目の引数 *mdlen()* は、データがおかれるバッファの実際のサイズです。

_tmpostrecv() が後処理でデータ長を変更した場合は、新しいデータ長を返す必要があります。返される長さは、使用された呼び出しに基づく方法でアプリケーションに渡されます (たとえば、tpcall() は呼び出し元が戻り時にチェックする引数の 1 つにデータ長を設定します)。

後処理が成功するには、バッファの大きさが十分でない可能性があります。容量がさらに必要な場合、_tmpostrecv() は望ましいバッファ・サイズの負の絶対値を返します。次いで呼び出しルーチンはバッファの大きさを変更し、_tmpostrecv() を再度呼び出します。

データに対して後処理が不要で、受け取ったデータ量がアプリケーションに返されるデータ量と同じであれば、NULL 関数ポインタを使用します。デフォルトのルーチンは *dlen()* を返し、バッファに対して何もしません。

正常終了の場合、_tmpostrecv() は、そのバッファが対応する受信呼び出しから渡されるときにアプリケーションが知っているべきデータの長さを返します。異常終了の場合、-1 を返し、_tmpostrecv() の呼び出し元も異常終了を示す tperrno() を TPESYSTEM に返します。

_tmencdec

_tmencdec() は、異なるデータ表現を使用するマシン間でネットワークを介してバッファを送受信するときに符号化 / 復号化を行うために使用します。BEA Tuxedo ATMI システムでは、XDR の使用が推奨されていますが、このルーチンの意味合いに則っていればどのような符号化 / 復号化方式をとってもかまいません。

この関数は tpcall()、tpacall()、tpbroadcast()、tpnotify()、tpconnect()、tpsend()、tpreturn()、および tpforward() によって呼び出され、呼び出し元のバッファを符号化します。ただし、この関数は異なるマシンにバッファを送るときにのみ呼び出します。これらの呼び出しの中で、_tmencdec() は _tmroute() と _tmpresend() の後で呼び出されます。ここで、_tmencdec() に渡されるバッファには、そのバッファに隣接しないデータを指すポインタが含まれない、という _tmpresend() の説明を思い出してください。

受信側では、tprecv()、tpgetrply()、tpcall() の受信側、およびサーバ用の関数はすべて _tmencdec() を呼び出して、異なるマシンからバッファを受け取った後、_tmpostrecv() を呼び出す前にこのバッファを復号化します。

_tmencdec() の最初の引数 *op()* は、この関数がデータの符号化または復号化のいずれを行うかを指定します。*op()* には TMENCODE または TMDECODE のいずれかを指定します。

op() に TMENCODE を指定すると、*encobj()* は BEA Tuxedo システムによって割り当てられたバッファを指します (このバッファにデータの符号化版が複写されます)。非符号化データは *obj()* におかれます。つまり、*op()* が TMENCODE であると、_tmencdec() は *obj()* をその符号化形式に変換し、結果を *encobj()* に入れます。*encobj()* が指すバッファのサイズは *elen()* によって指定され、*obj()* によって示されるバッファの長さ (*olen()*) の少なくとも 4 倍です。*olen()* は、_tmpresend によって返される長さです。_tmencdec() は、符号化されたデータの長さを *encobj()* で返します (つまり、実際に送信するデータ量)。_tmencdec() は、この関数に渡されるバッファのいずれも解放しないものとします。

op() に TMDECODE を指定すると、*encobj()* は BEA Tuxedo ATMI システムによって割り当てられたバッファを指します (このバッファにデータの符号化版がおかれます)。バッファの長さは *elen()* です。*obj()* は、*encobj()* が指すバッファ (ここに復号化されたデータが複写されます) の長さと少なくとも同じです。*obj()* の長さは、*olen()* です。*obj()* はアプリケーションによって最終的に返されるバッファであるため、BEA Tuxedo ATMI システムは _tmencdec() を呼び出す前に、復号化データを収めるのに十分な大きさになるようこのバッファのサイズを大きくすることができます。_tmencdec() は、復号化データのサイズを *obj()* に返します。_tmencdec() が戻ると、_tmpostrecv() がその最初の引数として *obj()*、2 番目の引数として _tmencdec() の戻り値、そして 3 番目の引数として *olen()* をとって呼び出されます。_tmencdec() は、この関数に渡されるバッファのいずれも解放しないものとします。

_tmencdec() は、NULL 以外のデータを符号化あるいは復号化する必要がある場合にのみ呼び出されます。

異なるマシンがネットワーク上に存在する場合でもデータに符号化あるいは復号化を行う必要がない場合には、NULL 関数ポインタを使用します。このルーチンは *olen()* (*op()* は TMENCODE と同じ) あるいは *elen()* (*op()* は TMDECODE と同じ) のいずれかを返します。

正常終了の場合、_tmencdec() は、上述のように負でないバッファ長を返します。異常終了の場合は、-1 を返し、これにより _tmencdec() 呼び出し元も異常終了を示す tperrno() を TPESYSTEM に返します。

_tmroute

メッセージは、デフォルトの設定では、要求されたサービスを提供する任意のサーバ・グループにルーティングされます。UBBCONFIG ファイルに記述する各サービス・エントリでは、ROUTING パラメータを使用して該当サービスのルーティング基準の論理名を指定することができます。複数のサービスが同じルーティング基準を共有することができます。あるサービスがルーティング基準名を指定されている場合、_tmroute() を使用して、メッセージ中のデータに基づいてそのメッセージが送信されるサーバ・グループを判別します。データとサーバ・グループのこの対応付け方法を、「データ依存型ルーティング」と呼んでいます。_tmroute() の呼び出しは、バッファの送信前 (そして、_tmpresend() と _tmencdec() が呼び出される前) に tpcall()、tpacall()、tpconnect()、および tpforward() で行います。

routing_name は、ルーティング基準の論理名 (UBBCONFIG ファイルに指定されている) であり、データ依存ルーティングを必要とするサービスと関連つけられています。*service* は、要求の対象となるサービスの名前です。パラメータ *data* は、要求で送出されるデータを指し、*len* はそのデータの長さです。ここで説明している他のルーチンと異なり、_tmroute() は *ptr()* が NULL の場合でも呼び出されます。*group* パラメータは、要求の送り先になるグループの名前を返すときに使用します。このグループ名は、UBBCONFIG ファイルに記述されているグループ名の 1 つ (および、そのグループが選択されたときにアクティブであったグループ名) と一致していなければなりません。要求を指定サービスを提供する任意のサーバに送ることができる場合、*group* を NULL 文字列に設定し、この関数が 1 を返すようにしてください。

データ依存ルーティングが該当バッファ・タイプに必要とされない場合には、NULL 関数ポインタを使用してください。このルーチンは *group* を NULL 文字列に設定し、1 を返します。

正常終了の場合、_tmroute() は 1 を返します。異常終了の場合、-1 を返し、_tmroute() の呼び出し元も異常終了を返します。その結果として、tperrno() が TPESYSTEM にセットされます。サーバあるいはサービスの利用不可能によって _tmroute() が異常終了した場合は、tperrno() は TPENOENT にセットされます。

group が無効なサーバ・グループの名前に設定されると、_tmroute() を呼び出す関数はエラーを返し、tperrno() を TPESYSTEM に設定します。

_tmfilter()

_tmfilter() は、tppost() によってポストされたバッファの内容を分析するためにイベント・ブローカ・サーバによって呼び出されます。サブスクライバ (tpsubscribe()) が提供した式がバッファの内容を基に評価されます。式が真の場合、_tmfilter() は 1 を返し、イベント・ブローカはサブスクライバへの通知処理を実行します。_tmfilter() が 0 を返した場合は、イベント・ブローカはこのポストをサブスクリプションの「一致」とみなしません。

exprlen() が -1 の場合、*expr()* は NULL で終わる文字列とみなされます。それ以外の場合、*expr()* は *exprlen* バイトのバイナリ・データとみなされます。*exprlen* が 0 の場合は、式がないことを示します。

フィルタリングがこのバッファ・タイプに適用しない場合は、NULL 関数ポインタを指定します。デフォルトのルーチンは、式がないか、*expr()* が空の NULL で終わる文字列の場合は 1 を返します。それ以外の場合、デフォルトのルーチンは 0 を返します。

_tmformat()

_tmformat() は、*fmt* という形式指定に従って、バッファのデータを表示可能な文字列に変換するためにイベント・ブローカ・サーバによって呼び出されます。イベント・ブローカは、*userlog()* または *system()* 通知処理の入力のためにポストされたバッファを文字列に変換します。

出力は、*result()* が指すメモリの位置に文字列として格納されます。*result()* には、終端NULL 文字を含めて最大 *maxresult()* バイト書き込まれます。*result()* の大きさが十分でない場合は、_tmformat() は出力を切り捨てます。出力文字列は、必ずNULL で終わります。

正常終了の場合、_tmformat() は負でない整数を返します。1 は正常終了、2 は出力文字列が切り捨てられたことを示します。異常終了の場合、-1 を返し、*result()* に空の文字列を格納します。

形式設定がこのバッファ・タイプに適用しない場合は、NULL 関数ポインタを指定します。デフォルトのルーチンが後を引き継ぎ、*result()* に空の文字列を返します。

_tmpresend2

_tmpresend2() は、tpcall()、tpacall()、tpconnect()、tpsend()、tpbroadcast()、tpnotify()、tpreturn()、および tpforward() でバッファが送られる前に呼び出されます。また、_tmroute() の後、ただし _tmencdec() の前にも呼び出されます。*iptr* がヌル以外の場合、このルーチンにより、バッファの送信前に、バッファに対する前処理を行うことができます。

_tmpresend2() の最初の引数 *iptr* は、送信呼び出しに渡されるアプリケーション・データ・バッファです。2 番目の引数 *ilen* は、送信呼び出しに渡されるデータの長さです。3 番目の引数 *mdlen* は、データがおかれるバッファの実際の長さです。

_tmpresend() とは異なり、_tmpresend2() は必要な処理がすべて終了した後で、ポインタ *optr* を受信し、これを使って *iptr* のデータを置くバッファを指すポインタを渡します。このポインタを使用するのは、入力バッファを修正する代わりに、_tmpresend2() が修正したデータに新しいバッファを使用する場合です。5 番目の引き数 *olen* は、*optr* バッファのサイズです。6 番目の引き数 *flags* は、処理されるバッファが親バッファ (送信先バッファ) かどうかを _tmpresend2() に通知します。_tmpresend2() は *flags* 引き数を返して、処理結果を示します。

optr バッファの大きさが不十分で、後処理ができない場合がります。容量がさらに必要な場合、_tmpresend2() は必要なバッファ・サイズの負の絶対値を返します。*optr* バッファの *olen* バイトはすべて維持されます。次に呼び出しルーチンがバッファの大きさを変更し、_tmpresend2() を再度呼び出します。

データに対して後処理が不要で、受け取ったデータ量がアプリケーションに返されるデータ量と同じであれば、NULL 関数ポインタを使用します。デフォルトのルーチンは *ilen* を返し、バッファに対して何もしません。

以下は、_tmpresend2() に入力可能なフラグです。

[TMPARENT]
　　これは、親バッファ (送信先バッファ) です。

flags で返されるフラグは、_tmpresend2() の結果を示します。可能な値は次のとおりです。

[TMUSEIPTR]
: _tmpresend2() が成功しました。処理されたデータは *iptr* が参照するバッファ内にあり、戻り値には送信されたデータの長さが含まれます。

[TMUSEOPTR]
: _tmpresend2() が成功しました。処理されたデータは *optr* が参照するバッファ内にあり、戻り値には送信されたデータの長さが含まれます。

TMUSEOPTR が返された場合、メッセージ伝送後の処理が _tmpresend() の処理とは異なります。つまり、*iptr* バッファは変更されず _tmpostsend() は呼び出されません。TMUSEIPTR が返された場合、_tmpostsend() で呼び出されるように _tmpresend() が呼び出されます。*optr* バッファの割り当てと、解放またはキャッシュは、呼び出し元が行います。

型付きバッファにこの方法を用いるのは、次のような理由によります。

- 伝送処理で作成されるバッファは、入力バッファの最大長よりも長いこと。
- バッファの伝送準備を元に戻すのは非常に複雑な作業で、データを別のバッファにコピーする方が簡単であること。

_tmpresend2() 関数は、関数が返るときに、バッファ内の送信データをそのまま送信できるようにします。_tmencdec() は類似しないマシンにバッファが送信される場合にだけ呼び出されるため、_tmpresend2() は、すべてのデータが送信されるバッファ内に隣接して保存されるようにします。

データに対する前処理が不要で、呼び出し元が指定したデータ量が送信すべきデータ量と同じ場合、バッファ・タイプ・スイッチの _tmpresend2() にヌルの関数ポインタを指定します。_tmpresend2() がヌルの場合、デフォルト設定によって _tmpresend() が呼び出されます。

正常終了時には、_tmpresend2() は送信するデータ量を返します。より大きなバッファが必要な場合は、必要なバッファ・サイズの負の絶対値を返します。異常終了時には -1 を返し、これによって _tmpresend2() の呼び出し元も異常終了を返して tperrno() を TPESYSTEM に設定します。

関連項目 tpacall(3c)、tpalloc(3c)、tpcall(3c)、tpconnect(3c)、tpdiscon(3c)、tpfree(3c)、tpgetrply(3c)、tpgprio(3c)、tprealloc(3c)、tprecv(3c)、tpsend(3c)、tpsprio(3c)、tptypes(3c)、tuxtypes(5)

catgets(3c)

名前

catgets()— プログラム・メッセージの読み取り

形式

```
#include <nl_types.h>
char *catgets (nl_catd catd, int set_num, int msg_num, char *s)
```

機能説明

catgets() は、セット *set_num* 内のメッセージ *msg_num* を、*catd* で指定されたメッセージ・カタログから読み取ります。*catd* は、先に呼び出された catopen() から返されたカタログ記述子です。*s* は、デフォルトのメッセージ文字列を指すポインタであり、指定されたメッセージ・カタログが現在利用できない場合に、catgets() から返されます。

マルチスレッドのアプリケーションのスレッドは、TPINVALIDCONTEXT を含むどんなコンテキスト状態でも、catgets() を呼び出すことができます。

診断

指定されたメッセージが正常に取り出されると、catgets() は NULL で終了するメッセージ文字列を収める内部バッファ領域を指すポインタを返します。*catd* によって指定されているメッセージ・カタログがその時点で利用できないために呼び出しが異常終了した場合には、*s* を指すポインタが返されます。

関連項目

catopen、catclose(3c)

catopen、catclose(3c)

名前

catopen()、catclose()— メッセージ・カタログのオープン / クローズ

形式

```
#include <nl_types.h>
nl_catd catopen (char *name, int oflag)
int catclose (nl_catd catd)
```

機能説明

catopen() はメッセージ・カタログをオープンし、カタログ記述子を返します。*name* はオープンするメッセージ・カタログの名前を指定します。*name* に "/" があると、次にくる *name* はメッセージ・カタログのパス名を示します。それ以外の場合、環境変数 NLSPATH が使用されます。NLSPATH が環境にない場合や、NLSPATH で指定されたパスでメッセージ・カタログをオープンできない場合には、デフォルトのパスが使用されます (nl_types(5) を参照)。

メッセージ・カタログの名前とそれらの格納場所は、システムによって異なる可能性があります。個々のアプリケーションは、それぞれのニーズに従ってメッセージ・カタログの名前を付けたり、格納場所を指定したりすることができます。したがって、カタログを格納する場所を指定するためのメカニズムが必要となります。

NLSPATH 変数では、メッセージ・カタログの格納場所を検索パスの形式で指定したり、メッセージ・カタログ・ファイルに対応するネーミング規則を使用することができます。

例：

```
NLSPATH=/nlslib/%L/%N.cat:/nlslib/%N/%L
```

メタキャラクタ % は、置換フィールドを示します。この例で、%L は LANG 環境変数の現在の設定と置き替わります (下記の項を参照)。また、%N は catopen() に渡される *name* パラメータの値と置き替わります。このため、上例で、catopen() はまず /nlslib/$LANG/*name*.cat、次に /nlslib/*name*/$LANG を検索して、目的のメッセージ・カタログを見つけようとします。

NLSPATH は通常、システム全体にわたって有効になるよう設定されるので (すなわち、/etc/profile)、メッセージ・カタログに関連する格納場所とネーミング規則をプログラムもユーザも意識する必要はありません。

次の表は、メタキャラクタ・セットのリストです。

メタキャラクタ	説明
%N	catopen に渡される名前パラメータの値
%L	LANG の値
%l	LANG の言語要素の値
%t	LANG の地域要素の値
%c	LANG のコードセット要素の値
%%	1 文字の %

LANG 環境変数では、ネイティブ言語、地域の習慣および文字セットに関してユーザの要求条件を ASCII 文字列の形式で指定することができます。
LANG=language[_territory[.codeset]]

オーストリアで使用されるドイツ語を話すユーザが、ISO 8859/1 コードセットを用いている端末を使用する場合、LANG 環境変数は次のように設定します。

```
LANG=De_A.88591
```

この設定により、ユーザは関連するカタログ (存在すれば) を見つけることができるはずです。

LANG 変数が設定されていない場合、setlocale(3c) によって返される LC_MESSAGES の値が使用されます。この値が NULL あると、nl_types(5) に定義されているデフォルトのパスが使用されます。

引数 *oflag()* は使用されません。この引数は将来の用途のために予約されており、0 (ゼロ) に設定されます。このフィールドの値を別の値に変更した場合の動作は不定です。

catclose() は *catd* によって指定されたメッセージ・カタログをクローズします。

マルチスレッドのアプリケーションのスレッドは、TPINVALIDCONTEXT を含むどんなコンテキスト状態でも、catopen() または catclose() 呼び出すことができます。

診断

正常終了の場合、catopen() は、以後の catgets() および catclose() の呼び出し時に使用するメッセージ・カタログ記述子を返します。異常終了であれば、catopen() は (nl_catd) -1 を返します。catclose() は、正常終了のとき 0、異常終了のとき -1 を返します。

関連項目

catgets(3c)、setlocale(3c)、nl_types(5)

decimal(3c)

名前　　decimal()— 十進数変換および算術ルーチン

形式

```
#include "decimal.h"

int
lddecimal(cp, len, np)        /* 十進数のロード */
char*cp;         /* 入力：圧縮した形式の位置 */

int
len;             /* 入力：圧縮した形式の長さ */
dec_t*np;        /* 出力：dec_t 形式の位置 */

void
stdecimal(np, cp, len)       /* 十進数の格納 */
dec_t*np;        /* 入力：dec_t 形式の位置 */
char*cp;         /* 出力：圧縮した形式の位置 */
int len;         /* 入力：圧縮した形式の長さ */

int
deccmp(n1, n2)   /* 2 つの十進数の比較 */
dec_t*n1;        /* 入力：比較する数 */
dec_t*n2;        /* 入力：比較する数 */

int
dectoasc(np, cp, len, right) /* dec_t をアスキーに変換 */
dec_t*np;        /* 入力：変換する数 */
char*cp;         /* 出力：変換後の数 */
int len;         /* 入力：出力文字列の長さ */
int right;       /* 入力：小数点以下の数 */

int
deccvasc(cp, len, np)        /* アスキーを dec_t へ変換 */
char*cp;         /* 入力：変換する数 */
int len;         /* 入力：変換する数の最大長 */
dec_t*np;        /* 出力：変換後の数 */

int
dectoint(np, ip)             /* int を dec_t に変換 */
dec_t*np;        /* 入力：変換する数 */
int *ip;         /* 出力：変換後の数 */

int
deccvint(in, np) /* dec_t を int に変換 */
```

```
int in;          /* 入力：変換する数 */
dec_t*np;        /* 出力：変換後の数 */

int
dectolong(np, lngp)         /* dec_t を long に変換 */
dec_t*np;        /* 入力：変換する数 */
long*lngp;       /* 出力：変換後の数 */

int
deccvlong(lng, np)        /* long を dec_t に変換 */
longlng;        /* 入力：変換する数 */
dec_t*np;       /* 出力：変換後の数 */

int
dectodbl(np, dblp)        /* dec_t を double に変換 */
dec_t*np;       /* 入力：変換する数 */
double *dblp;   /* 出力：変換後の数 */

int
deccvdbl(dbl, np)         /* double を dec_t に変換 */
double *dbl;    /* 入力：変換する数 */
dec_t*np;       /* 出力：変換後の数 */

int
dectoflt(np, fltp)        /* dec_t を float に変換 */
dec_t*np;       /* 入力：変換する数 */
float*fltp;     /* 出力：変換後の数 */

int
deccvflt(flt, np)         /* float を dec_t に変換 */
double *flt;    /* 入力：変換する数 */
dec_t*np;       /* 出力：変換後の数 */

int
decadd(*n1, *n2, *n3)     /* 2 つの十進数の加算を行う */
dec_t*n1;        /* 入力：加数 */
dec_t*n2;        /* 入力：加数 */
dec_t*n3;        /* 出力：合計 */

int
decsub(*n1, *n2, *n3)     /* 2 つの十進数の減算を行う */
dec_t*n1;        /* 入力：被減数 */
dec_t*n2;        /* 入力：減数 */
dec_t*n3;        /* 出力：差 */

int
decmul(*n1, *n2, *n3)     /* 2 つの十進数の乗算を行う */
dec_t*n1;        /* 入力：被乗数 */
```

```
dec_t*n2;         /* 入力：被乗数 */
dec_t*n3;         /* 出力：積 */

int
decdiv(*n1, *n2, *n3)     /* 2 つの十進数の除算を行う */
dec_t*n1;         /* 入力：被除数 */
dec_t*n2;         /* 入力：除数 */
dec_t*n3;         /* 出力：商 */
```

機能説明

これらの関数を使用すると BEA Tuxedo ATMI システムにおいてパック化十進データを格納、変換、および操作することができます。BEA Tuxedo ATMI システムにおいて表現される十進数のデータ型の形式は、CICS でのその表現とは異なることに注意してください。

マルチスレッドのアプリケーションのスレッドは、TPINVALIDCONTEXT を含むどんなコンテキスト状態でも、10 進数の変換関数を呼び出すことができます。

ネイティブの十進数表現

十進数は、BEA Tuxedo ATMI システム上では dec_t 構造体を使用して表現されます。この構造体の定義を次に示します。この構造体の定義を次に示します。

```
#define DECSIZE                 16
struct decimal {
       short dec_exp;           /* 指数底 100 */
       short dec_pos;           /* 符号：正 (1)、負 (0)、ヌル (-1) */
       short dec_ndgts;         /* 有効桁数 */
       char  dec_dgts[DECSIZE]; /* 実際の桁数底 100 */
};
typedef struct decimal dec_t;
```

プログラマは、dec_t 構造体に直接アクセスする必要はありません。基本的なデータ構造体を理解するためにこの構造体を提示します。莫大な十進数データを格納する必要がある場合、より圧縮した形式を得るために stdecimal()、lddecimal() 関数を使用します。dectoasc()、dectoint()、dectolong()、dectodbl() および dectoflt() を使用すると、十進数から他のデータ・タイプへ変換することができます。deccvasc()、deccvint()、deccvlong()、deccvdbl() および deccvflt() を使用すると、他のデータ・タイプから十進数のデータ・タイプに変換することができます。deccmp() は、2 つの十進数を比較する関数です。1 番目の十進数が 2 番目の十進数より小さい場合、-1 を返します。2 つの十進数が等しい場合、0 を返します。1 番目の十進数が 2 番目の十進数より大きい場合 1 を返します。どちらかの引数が無効である場合、-1 以外の負の値を返します。decadd()、decsub()、decmul() および decdiv() は、十進数の算術演算を行います。

戻り値

特に指定がない限り、これらの関数は、正常終了時には 0 を返し、エラー時には負の数を返します。

gp_mktime(3c)

名前

gp_mktime()—tm 構造体をカレンダー時間に変換します。

形式

```
#include <time.h>
time_t gp_mktime (struct tm *timeptr);
```

機能説明

gp_mktime() は、*timeptr* が指す tm 構造体で表現される時間をカレンダー時間 (1970 年 1 月 1 日から数えた秒数) に変換します。

tm 構造体の形式は次の通りです。

```
struct tm {
  int tm_sec;     /* seconds after the minute [0, 61] */
  int tm_min;     /* minutes after the hour [0, 59] */
  int tm_hour;    /* hour since midnight [0, 23] */
  int tm_mday;    /* day of the month [1, 31] */
  int tm_mon;     /* months since January [0, 11] */
  int tm_year;    /* years since 1900 */
  int tm_wday;    /* days since Sunday [0, 6] */
  int tm_yday;    /* days since January 1 [0, 365] */
  int tm_isdst;   /* flag for daylight savings time */
};
```

gp_mktime() は、カレンダー時間を計算する以外に、指定された tm 構造体を正規化します。構造体の *tm_wday* メンバと *tm_yday* メンバの元の値は無視され、他のメンバの元の値は、構造体の定義で示される範囲に制限されません。正常終了の場合、*tm_wday* および *tm_yday* の値は適切に設定されます。他のメンバは指定されたカレンダー時間を表すように設定されますが、それらの値は、正しい範囲内に収まるように強制されます。*tm_mday* の最終的な値は、*tm_mon* および *tm_year* が決まるまで設定されません。

構造体の各メンバの元の値は、決められた範囲よりも大きくても小さくてもかまいません。たとえば、*tm_hour* が -1 なら、深夜 12 時の 1 時間前、*tm_mday* が 0 ならその月の 1 日前、*tm_mon* が -2 なら、*tm_year* の 1 月の 2 ヵ月を意味します。

tm_isdst が正の場合、元の値は代替タイムゾーンに基づいていると見なされます。代替タイムゾーンが計算されたカレンダー時間に対して有効でないことが判明すると、各メンバはメイン・タイムゾーンに調整されます。同様に、*tm_isdst* がゼロの場合、元の値はメイン・タイムゾーンであると見なされ、メイン・タイムゾーンが有効でなければ代替タイムゾーンに変換されます。*tm_isdst* が負の場合、正しいタイムゾーンが判断され、各メンバは調整されません。

ローカル・タイムゾーンの情報は、あたかも gp_mktime() が tzset() を呼び出したかのように使われます。

gp_mktime() は、特定のカレンダー時間を返します。カレンダー時間を表現できない場合、この関数は値 (time_t)-1 を返します。

マルチスレッドのアプリケーションのスレッドは、TPINVALIDCONTEXT を含むどんなコンテキスト状態でも、gp_mktime() を呼び出すことができます。

使用例

2001 年 7 月 4 日は何曜日か？

```
#include <stdio.h>
#include <time.h>

static char *const wday[] = {
"Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday", "-unknown-"
};

struct tm time_str;
/*...*/
time_str.tm_year        = 2001 - 1900;
time_str.tm_mon         = 7 - 1;
time_str.tm_mday        = 4;
time_str.tm_hour        = 0;
time_str.tm_min         = 0;
time_str.tm_sec         = 1;
time_str.tm_isdst       = -1;
if (gp_mktime(time_str) == -1)
   time_str.tm_wday=7;
printf("%s\en", wday[time_str.tm_wday]);
```

注意事項

tm 構造体の *tm_year* は、1970 年以降でなければなりません。1970 年 1 月 1 日の 00:00:00 UTC より前、または 2038 年 1 月 19 日の 03:14:07 UTC より後のカレンダー時間を表現することはできません。

移植性

コンパイル・システムが ANSI C の mktime() 関数をすでに提供しているシステムでは、gp_mktime() は mktime() を呼んで、変換を行うだけです。それ以外の場合、変換は直接 gp_mktime() の内部で行われます。

後者の場合、TZ 環境変数が設定されていなければなりません。多くのインストレーションでは、TZ は、ユーザがログオンする際にデフォルトで正しい値に設定されることに注意してください。TZ のデフォルト値は GMT0 です。TZ の形式は次の通りです。

```
stdoffset[dst[offset],[start[time],end[time]]]
```

std および *dst*

標準 (*std*) および夏時間 (*dst*) のタイムゾーンを表す 3 バイト以上の文字です。*std* だけが必須です。*dst* が無い場合、このロケールには夏時間は適用されません。大文字と小文字をどちらでも使用できます。先頭のコロン (:)、数字、コンマ (,)、マイナス (-)、またはプラス (+) 以外のすべての文字が使用できます。

offset

協定世界時 (UTC) を得るためにローカル時間に加算しなければならない値を表します。*offset* の形式は次の通りです。*hh*[:*mm*[:*ss*]]. 分 (*mm*) と秒 (*ss*) は省略可能です。時間 (*hh*) は必須で、一つの数字でもかまいません。*std* の次の *offset* は必須です。*dst* の次に *offset* が無ければ、夏時間は標準時間の 1 時間先であると見なされます。一つ以上の数字を使用でき、値は常に十進数として解釈されます。時間は 0 から 24 の間でなければならず、分 (および秒) は、もしあれば、0 から 59 の間でなければなりません。範囲外の値は、予期できない動作を引き起こす場合があります。先頭に "-" がつくと、タイムゾーンはグリニッジ子午線の東にあります。それ以外の場合、タイムゾーンは西にあります (省略可能な "+" 符号で示してもかまいません)。

start/time、*end/time*

いつ夏時間に変更し、いつ戻すかを示します。ここで、*start*/time は、いつ標準時間から夏時間への変更が発生しするかを示し、*end*/*time* は、逆の変更がいつ起こるかを示します。それぞれの *time* フィールドは、ローカル時間で、いつ変更が行われるかを示します。

start と *end* の形式は次のうちのどれかです。

J*n*

ユリウス日 *n* (1 <*n* < 365)。うるう日は含まれません。つまり、すべての年で 2 月 28 日が 59 日目で、3 月 1 日が 60 日目です。2 月 29 日があってもそれを指定することはできません。

N

ゼロ・ベースのユリウス日 (0 < *n* < 365)。うるう日が含まれるので、2 月 29 日を指定することができます。

M*m.n.d*

その年の *m* 月の *n* 週 (1 < 5、1 < *m* < 12) の *d* 番目の日 (0 < *d* <6) です。ただし、週 5 は、4 週目または 5 週目に発生する「*m* 月の最後の *d* 日」を意味します。週 1 は、*d* 番目の日が発生する最初の週です。日 0 (ゼロ) は、日曜日です。

start および *end* には、これらの省略可能なフィールドが与えられなかった場合、実装依存のデフォルトが使用されます。

time は、*offset* と同じ形式で、違いは先頭の符号 ("-" または "+") が許されることです。*time* が与えられなかった場合のデフォルトは 02:00:00 です。

関連項目　UNIX システムのリファレンス・マニュアルの ctime(3c) と getenv(3c)、timezone(4)

nl_langinfo(3c)

名前　　nl_langinfo()— 言語情報

形式

```
#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo (nl_item item);
```

機能説明　　nl_langinfo() は、ある特定言語あるいはプログラム・ロケールで定義されている文化圏に関連する情報を収めた NULL で終わる文字列を指すポインタを返します。*item* に入る定数名と値は langinfo.h に定義されています。

例：

```
nl_langinfo (ABDAY_1);
```

指定された言語がフランス語で、フランス語のロケールが正しくインストールされていれば、文字列 "Dim" を指すポインタを返します。また、指定された言語が英語であれば、"Sun" が返されます。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、nl_langinfo() の呼び出しを発行できます。

診断　　setlocale() が正常に呼び出されなかった場合、あるいはサポートされている言語の langinf0() データが存在しないか、*item* が該当箇所に定義されていない場合には、nl_langinfo() は C ロケールの対応する文字列を指すポインタを返します。どのロケールの場合も、*item* に無効な文字列が指定されると、nl_langinfo() は空の文字列を指すポインタを返します。

注意事項　　戻り値が指す配列は、プログラムで変更しないようにしてください。nl_langinfo() の次の呼び出しで、この配列の内容は変わってしまいます。

関連項目　　setlocale(3c)、strftime(3c)、langinfo(5)、nl_types(5)

rpc_sm_allocate、rpc_ss_allocate(3c)

名前

rpc_sm_allocate()、rpc_ss_allocate()—RPC スタブ・メモリ管理方式でのメモリの割り当て

形式

```
#include <rpc/rpc.h>
idl_void_p_t rpc_sm_allocate(unsigned32 size, unsigned32 *status)
idl_void_p_t rpc_ss_allocate(unsigned32 size)
```

機能説明

アプリケーションは rpc_sm_allocat3() を呼び出し、RPC スタブ・メモリ管理方式でメモリを割り当てます。入力パラメータ *size* は、割り当てるメモリの大きさをバイト単位で指定します。このルーチンを呼び出す前に、スタブ・メモリ管理環境が確立されていなければなりません。サーバ・スタブから呼び出されるサービス・コードでは通常、スタブ自身が必要な環境を確立します。スタブから呼び出されないコードで rpc_sm_allocate() を使用する場合には、アプリケーションが rpc_sm_enable_allocate() を呼び出し、必要なメモリ管理環境を確立しなければなりません。

サーバ・スタブのパラメータ中に参照によるパラメータの転送に使用される以外のポインタが含まれている、あるいは [enable_allocate] 属性が ACS ファイル内の操作に指定されている場合には、環境は自動的に設定されます。そうでなければ、環境は rpc_sm_enable_allocate() を呼び出すことで、アプリケーションにより設定されなければなりません。

スタブがメモリ管理環境を確立する場合には、スタブ自身が rpc_sm_allocate() により割り当てられたメモリを解放します。アプリケーションは rpc_sm_free() を呼び出し、呼び出し側スタブに戻る前にメモリを解放することができます。

アプリケーションがメモリ管理環境を確立した場合には、アプリケーションは rpc_sm_free() または rpc_sm_disable_allocate() を呼び出し、割り当てられたすべてのメモリを解放しなければなりません。

出力パラメータ *status* には、このルーチンからのステータス・コードが返されます。このステータス・コードは、ルーチンが成功して完了したか、または失敗した場合はその理由を示します。次は、ステータス・コードとその意味の一覧です。

rpc_s_ok
: 常にこのコードが返されます。戻り値により障害の原因を決定します。

rpc_ss_allocate() は、この関数の例外復帰バージョンであり、出力パラメータ *status* を持ちません。例外は発生しません。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、rpc_sm_allocate() または rpc_ss_allocate() の呼び出しを発行できます。

戻り値

正常終了すると、このルーチンは割り当てられたメモリを指すポインタを返します。idl_void_p_t は、ISO 標準 C 環境では void * に、他の環境では char * に定義されていることに注意してください。

メモリ不足の場合、ルーチンはヌル・ポインタを返します。

関連項目

rpc_sm_disable_allocate、rpc_ss_disable_allocate(3c)、rpc_sm_enable_allocate、rpc_ss_enable_allocate(3c)、rpc_sm_free、rpc_ss_free(3c)

『TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング』

rpc_sm_client_free、rpc_ss_client_free(3c)

名前

rpc_sm_client_free()、rpc_ss_client_free()— クライアント・スタブから返されたメモリの解放

形式

```
#include <rpc/rpc.h>
void rpc_sm_client_free (idl_void_p_t node_to_free, unsigned32 *status)
void rpc_ss_client_free (idl_void_p_t node_to_free)
```

機能説明

ルーチン rpc_sm_client_free() は、クライアント・スタブに割り当てられ、そのスタブから返されたメモリを解放します。入力パラメータ *node_to_free* は、クライアント・スタブから返されたメモリを指すポインタを指定します。ISO 標準の C 環境では、idl_void_p_t は void * と定義され、他の環境では char * と定義されます。

このルーチンにより、他のルーチンはそれが呼び出されたメモリ管理環境を意識せずに、RPC コールにより返された動的に割り当てられたメモリを解放することができます。

コードがサーバの一部として実行している場合であっても、このルーチンは常にクライアント・コードから呼び出されます。

出力パラメータ *status* には、このルーチンからのステータス・コードが返されます。このステータス・コードは、ルーチンが成功して完了したか、または失敗した場合はその理由を示します。このステータス・コードは、ルーチンが成功して完了したか、または失敗した場合はその理由を示します。次は、ステータス・コードとその意味の一覧です。

rpc_s_ok
: 正常終了

rpc_ss_client_free() は、この関数の例外復帰バージョンであり、出力パラメータ *status* を持ちません。例外は発生しません。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、rpc_sm_client_free() または rpc_ss_client_free() の呼び出しを発行できます。

戻り値

なし

関連項目

`rpc_sm_free`、`rpc_ss_free(3c)`、`rpc_sm_set_client_alloc_free`、`rpc_ss_set_client_alloc_free(3c)`、`rpc_sm_swap_client_alloc_free`、`rpc_ss_swap_client_alloc_free(3c)`

『TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング』

rpc_sm_disable_allocate、rpc_ss_disable_allocate(3c)

名前

rpc_sm_disable_allocate()、rpc_ss_disable_allocate()— 資源とスタブ・メモリ管理方式で割り当てられたメモリの解放

形式

```
#include <rpc/rpc.h>
void rpc_sm_disable_allocate(unsigned32 *status);
void rpc_ss_disable_allocate(void);
```

機能説明

rpc_sm_disable_allocate() ルーチンは、rpc_sm_enable_allocate() を呼び出して取得したすべての資源と、rpc_sm_allocate() の呼び出し後に rpc_sm_enable_allocate() を呼び出すことで割り当てられたすべてのメモリを解放します。

rpc_sm_enable_allocate() と rpc_sm_disable_allocate() は、対にして使用しなければなりません。rpc_sm_enable_allocate() を呼び出さずにこのルーチンを使用すると、予期できない結果が生じます。

出力パラメータ *status* には、このルーチンからのステータス・コードが返されます。このステータス・コードは、ルーチンが成功して完了したか、または失敗した場合はその理由を示します。次は、ステータス・コードとその意味の一覧です。

rpc_s_ok
　　正常終了

rpc_ss_disable_allocate() は、この関数の例外復帰バージョンであり、出力パラメータ *status* を持ちません。例外は発生しません。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、rpc_sm_disable_allocate() または rpc_ss_disable_allocate() の呼び出しを発行できます。

戻り値

なし

関連項目

rpc_sm_allocate、rpc_ss_allocate(3c)、rpc_sm_enable_allocate、rpc_ss_enable_allocate(3c)

『TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング』

rpc_sm_enable_allocate、rpc_ss_enable_allocate(3c)

名前

rpc_sm_enable_allocate()、rpc_ss_enable_allocate()— スタブ・メモリ管理環境を有効にする

形式

```
#include <rpc/rpc.h>
void rpc_sm_enable_allocate(unsigned32 *status)
void rpc_ss_enable_allocate(void)
```

機能説明

アプリケーションから rpc_sm_enable_allocate() を呼び出して、スタブ自身によってメモリ管理環境が設定されていない場合に、スタブ・メモリ環境を設定できます。すべての rpc_sm_allocate() の呼び出しの前に、スタブ・メモリ環境を設定しなければなりません。サーバのスタブから呼び出されるサービス・コードについては、通常、スタブ・メモリ環境はスタブ自身によって設定されます。他のコンテキストから呼び出すコード (たとえば、サービス・コードをスタブからでなく直接呼び出す場合) では、rpc_sm_allocate() を呼び出す前に rpc_sm_enable_allocate() を呼び出す必要があります。

出力パラメータ *status* には、このルーチンからのステータス・コードが返されます。このステータス・コードは、ルーチンが成功して完了したか、または失敗した場合はその理由を示します。次は、ステータス・コードとその意味の一覧です。

rpc_s_ok
: 正常終了

rpc_s_no_memory
: 必要なデータ構造体をセットアップするために十分なメモリを割り当てることができない。

rpc_ss_enable_allocate() は、この関数の例外復帰バージョンで、出力パラメータ *status* を持ちません。このルーチンは次の例外を発生します。

rpc_x_no_memory
: 必要なデータ構造体をセットアップするために十分なメモリを割り当てることができない。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、rpc_sm_enable_allocate() または rpc_ss_enable_allocate() の呼び出しを発行できます。

戻り値

なし

関連項目　　rpc_sm_allocate、rpc_ss_allocate(3c)、rpc_sm_disable_allocate、rpc_ss_disable_allocate(3c)

『TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング』

rpc_sm_free、rpc_ss_free(3c)

名前

rpc_sm_free,rpc_ss_free()—rpc_sm_allocate() ルーチンによって割り当てたメモリを解放する

形式

```
#include <rpc/rpc.h>
void rpc_sm_free(idl_void_p_t node_to_free, unsigned32 *status)
void rpc_ss_free(idl_void_p_t node_to_free)
```

機能説明

アプリケーションから　rpc_sm_free() を呼び出して、rpc_sm_allocate() を使用して割り当てたメモリを開放します。入力パラメータ *node_to_free* には、rpc_sm_allocate() を使用して割り当てたメモリへのポインタを指定します。ISO 標準の C 環境では、idl_void_p_t は void * と定義され、他の環境では char * と定義されます。

スタブ・メモリ管理環境でスタブがメモリを割り当てると、スタブから呼び出されるサービス・コードは、rpc_sm_free() を使用してスタブが割り当てたメモリを開放することができます。

rpc_sm_allocate() によって割り当てたのではないメモリへのポインタ、または rpc_sm_allocate() によって割り当てたメモリでも先頭以外の場所へのポインタで rpc_ss_free() を呼び出した場合は、予期できない結果が生じます。

出力パラメータ *status* には、このルーチンからのステータス・コードが返されます。このステータス・コードは、ルーチンが成功して完了したか、または失敗した場合はその理由を示します。次は、ステータス・コードとその意味の一覧です。

rpc_s_ok
　　正常終了

rpc_ss_free は、この関数の例外復帰バージョンであり、出力パラメータ *status* を持ちません。例外は発生しません。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、rpc_sm_free() または rpc_ss_free() の呼び出しを実行できます。

戻り値

なし

関連項目

rpc_sm_allocate、rpc_ss_allocate(3c)

『TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング』

rpc_sm_set_client_alloc_free、rpc_ss_set_client_alloc_free(3c)

名前

rpc_sm_set_client_alloc_free()、rpc_ss_set_client_alloc_free()— クライアントのスタブが使用するメモリ管理および開放の機構を設定する

形式

```
#include <rpc/rpc.h>
void rpc_sm_set_client_alloc_free(
   idl_void_p_t (*p_allocate)(unsigned long size),
   void (*p_free) (idl_void_p_t ptr), unsigned32 *status)

void rpc_ss_set_client_alloc_free(
   idl_void_p_t (*p_allocate)(unsigned long size),
   void (*p_free) (idl_void_p_t ptr))
```

機能説明

rpc_sm_set_client_alloc_free() は、クライアントのスタブがメモリ管理に使用するデフォルトのルーチンよりも優先されます。入力パラメータ *p_allocate* および *p_free* には、メモリの割り当ておよび開放のルーチンを指定します。サーバのコード中でリモート・コールが発生する場合 (この場合、メモリ管理ルーチンは rpc_ss_allocate(3) および rpc_ss_free(3) でなければなりません) を除いて、デフォルトのメモリ管理ルーチンは ISO C の malloc() および free() になります。

出力パラメータ *status* には、このルーチンからのステータス・コードが返されます。このステータス・コードは、ルーチンが成功して完了したか、または失敗した場合はその理由を示します。次は、ステータス・コードとその意味の一覧です。

rpc_s_ok
: 正常終了

rpc_s_no_memory
: 必要なデータ構造体をセットアップするために十分なメモリを割り当てることができない。

rpc_ss_set_client_alloc_free は、この関数の例外復帰バージョンで、出力パラメータ *status* を持ちません。このルーチンは次の例外を発生します。

rpc_x_no_memory
: 必要なデータ構造体をセットアップするために十分なメモリを割り当てることができない。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、rpc_sm_set_client_alloc_free() または rpc_ss_set_client_alloc_free() の呼び出しを発行できます。

戻り値　なし

関連項目　`rpc_sm_allocate`、`rpc_ss_allocate(3c)`、`rpc_sm_free`、`rpc_ss_free(3c)`
『TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング』

rpc_sm_swap_client_alloc_free、rpc_ss_swap_client_alloc_free(3c)

名前

rpc_sm_swap_client_alloc_free()、rpc_ss_swap_client_alloc_free()— クライアントのスタブが使用するメモリ管理および開放の機構を、クライアントが提供する機構に交換する

形式

```
#include <rpc/rpc.h>
void rpc_sm_swap_client_alloc_free(
  idl_void_p_t (*p_allocate)(unsigned long size),
  void (*p_free) (idl_void_p_t ptr),
  idl_void_p_t (**p_p_old_allocate)(unsigned long size),
  void (**p_p_old_free)( idl_void_p_t ptr),
  unsigned32 *status)

void rpc_ss_swap_client_alloc_free(
  idl_void_p_t (*p_allocate)(unsigned long size),
  void (*p_free) (idl_void_p_t ptr),
  idl_void_p_t (**p_p_old_allocate)(unsigned long size),
  void (**p_p_old_free)( idl_void_p_t ptr))
```

機能説明

rpc_sm_swap_client_alloc_free() ルーチンは、クライアントのスタブが使用する現在のメモリ管理および開放の機構を、呼び出し側が提供するルーチンに交換します。入力パラメータ *p_allocate* および *p_free* には、新しいメモリ割り当ておよび開放のルーチンを指定します。出力パラメータ *p_p_old_allocate* および *p_p_old_free* には、このルーチンを呼び出す前に使用されていたメモリの割り当ておよび開放のルーチンが返されます。

呼び出し可能なルーチンが RPC クライアントの場合は、その呼び出し元が選択した機構にかかわらず、どのメモリ割り当ておよび開放のルーチンが使用されたかを確認する必要があります。このルーチンを使用すれば、メモリ割り当ておよび開放の機構を意図的に交換して、使用されたルーチンを確認することができます。

出力パラメータ *status* には、このルーチンからのステータス・コードが返されます。このステータス・コードは、ルーチンが成功して完了したか、または失敗した場合はその理由を示します。次は、ステータス・コードとその意味の一覧です。

rpc_s_ok
: 正常終了

rpc_s_no_memory
: 必要なデータ構造体をセットアップするために十分なメモリを割り当てることができない。

`rpc_ss_swap_client_alloc_free` は、この関数の例外復帰バージョンで、出力パラメータ *`status`* を持ちません。このルーチンは次の例外を発生します。

`rpc_x_no_memory`
: 必要なデータ構造体をセットアップするために十分なメモリを割り当てることができない。

マルチスレッドのアプリケーション中のスレッドは、`TPINVALIDCONTEXT` を含め、どのコンテキスト状態で実行していても、`rpc_sm_swap_client_alloc_free()` または `rpc_ss_swap_client_alloc_free()` の呼び出しを発行できます。

戻り値

なし

関連項目

`rpc_sm_allocate`、`rpc_ss_allocate(3c)`、`rpc_sm_free`、`rpc_ss_free(3c)`、`rpc_sm_set_client_alloc_free`、`rpc_ss_set_client_alloc_free(3c)`

『TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング』

setlocale(3c)

名前

setlocale()— プログラムのロケールの修正と照会

形式

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

機能説明

setlocale() は、プログラムのロケールの該当部分を、*category* および *locale* 引数の指定に従って選択します。*category* 引数には、次のいずれかの値を指定することができます。

```
LC_CTYPE
LC_NUMERIC
LC_TIME
LC_COLLATE
LC_MONETARY
LC_MESSAGES
LC_ALL
```

これらの名前は、ヘッダ・ファイル locale.h に定義されています。BEA Tuxedo ATMI システムとの互換関数の場合、setlocale() によりすべてのカテゴリにつき１つだけ *locale* を使用できます。どのカテゴリを設定しても、LC_ALL (プログラムのロケール全体を表す) と同じに扱われます。

locale の値 "C" はデフォルトの環境を指定します。

また、値 "" は、そのロケールを環境変数から取り出すことを表します。環境変数 LANG からロケールが判別されます。

プログラムの起動時には、次のような関数が実行されます。

```
setlocale(LC_ALL, "C")
```

この関数は、各カテゴリを環境 "C" で記述されるロケールに初期化します。

ある文字列を指すポインタが *locale* に対して指定されると、setlocale() はすべてのカテゴリのロケールを *locale* に設定しようとします。*locale* は１つのロケールからなる単純なロケールでなければなりません。setlocale() がカテゴリに対するロケールの設定に失敗すると、NULL ポインタが返され、すべてのカテゴリに対するプログラムのロケールは変更されません。正常に設定されれば、設定されたロケールが返されます。

locale に NULL ポインタが設定されると、setlocale() は *category* に対応する現在のロケールを返します。プログラムのロケールは変更されません。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、setlocale() の呼び出しを発行できます。

ファイル

$TUXDIR/locale/C/LANGINFO—C ロケールの時刻と通貨のデータベース
$TUXDIR/locale/locale/*—各ロケールに関する固有の情報を含むファイル
$TUXDIR/locale/C/*_CAT—ロケールのテキスト・メッセージ

注意事項

複合ロケールはサポートされていません。複合ロケールは、"/" で始まり、各カテゴリのロケールを "/" で区切ってリストした文字列です。

関連項目

mklanginfo(1)

UNIX システムのリファレンス・マニュアルの ctime(3c)、ctype(3c)、getdate(3c)、localeconv(3c)、strftime(3c)、strtod(3c)、printf(3S)、environ(5)

strerror(3c)

名前　strerror()— エラー・メッセージ文字列の取り込み

形式

```
#include <string.h>
char *strerror (int errnum);
```

機能説明　strerror は *errnum* に指定されたエラー番号をエラー・メッセージ文字列にマッピングし、その文字列を指すポインタを返します。strerror は、perror と同じエラー・メッセージ・セットを使用します。返される文字列は上書きされないようにしてください。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、strerror() の呼び出しを発行できます。

関連項目　UNIX システムのリファレンス・マニュアルの perror(3)

strftime(3c)

名前　　strftime()— 日付と時刻を文字列に変換

形式

```
#include <time.h>

size_t *strftime (char *s, size_t maxsize, const char *format, const
struct tm *timeptr);
```

機能説明　　strftime() は、*format* が指す文字列の制御に従って、*s* が指す配列に文字を入れます。*format* 文字列は、ゼロ個以上のディレクティブと通常文字で構成します。すべての通常文字 (文字列の最後を表すヌル文字列を含む) は、そのまま配列に複写されます。strftime() の場合、*maxsize* 個以上の文字は配列には入りません。

format が (char *)0 である場合、ロケールのデフォルトの形式が使用されます。デフォルトの形式は "%c" と同じです。

各ディレクティブは、次のリストの記述に従って該当する文字と置き換えられます。該当文字はプログラムのロケールの LC_TIME カテゴリおよび *timeptr* が指す構造体に格納されている値によって判別されます。

文字	説明
%%	% と同じ
%a	該当ロケールにおける曜日の略称
%A	該当ロケールにおける曜日の正式名
%b	該当ロケールの月の略称
%B	該当ロケールの月の正式名
%c	該当ロケールの日付 / 時刻表現
%c	date(1) によって出力される該当ロケールの日付 / 時刻表現
%d	月の日 (01-31)
%D	%m/%d/%y 形式による日付
%e	月の日 (1-31: 1 桁の数字には先頭に空白が付く)
%h	該当ロケールの月の略称

%H	時刻 (00-23)
%I	時刻 (01-12)
%j	年の通算日 (001-366)
%m	月の番号 (01-12)
%M	分 (00-59)
%n	\ と同じ
%p	該当ロケールの午前または午後の表現
%r	%I:%M:%S [AM|PM] 形式の時刻表現
%R	%H:%M と同じ
%S	秒 (00-61)、うるう秒も可
%t	タブの挿入
%T	%H:%M:%S 形式の時刻表現
%U	年間の週番号 (00-53)、第 1 週の第 1 日目を日曜日とする
%w	曜日番号 (0-6)、日曜日 = 0
%W	年間の週番号 (00-53)、第 1 週の第 1 日目を月曜日とする
%x	該当ロケールの日付表現
%X	該当ロケールの時刻表現
%y	１世紀の年数 (00-99)
%Y	ccyy 形式の年表現 (例 : 1986)
%Z	時間帯の名前、時間帯が存在しなければなし

%U と %W の相違点は、日にちを週の初日から数えるかどうかという点です。週番号 01 は、%U の場合、日曜日から始まる１月の第１週、%W の場合は月曜日から始まる１月の第１週を表します。また、週番号 00 には、%U および %W でそれぞれ最初の日曜日または月曜日の前の日にちを表します。

終結 NULL 文字を含む結果として得られた文字の合計数が *maxsize* を超えない場合、strftime() は *s* が指す配列に置かれた文字数 (終結 NULL 文字を含まない) を返します。maxsize を超えた場合には、ゼロが返され、配列の内容は不定になります。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、strftime() の呼び出しを発行できます。

出力言語の選択

デフォルトの設定では、strftime() の出力は U.S.English で行われます。strftime() の出力は、setlocale() にカテゴリ LC_TIME の *locale* を設定することによって特定の言語に変更することができます。

時間帯

時間帯は環境変数 TZ から取り込まれます (TZ の詳細については、ctime(3C) を参照)。

使用例

strftime() の使用例を示します。この例は、*tmptr* が指す構造体に、ニュージャージー州における 1986 年 8 月 28 日木曜日の 12 時 44 分 36 秒に対応する値が入っている場合に、str の文字列がどのようになるかを示しています。

```
strftime (str, strsize, "%A %b %d %j", tmptr)
```

この結果、str には "Thursday Aug 28 240" の文字列が入ります。

ファイル

$TUXDIR/locale/*locale*/LANGINFO — コンパイルされたロケール固有の日付 / 時刻情報を収めているファイル

関連項目

mklanginfo(1)、setlocale(3c)

tpabort(3c)

名前

tpabort()— 現在のトランザクションをアボートするルーチン

形式

```
#include <atmi.h>
int tpabort(long flags)
```

機能説明

tpabort() は、トランザクションのアボートを指定します。この関数が終了すると、そのトランザクションでなされたリソースへの変更内容はすべて取り消されます。tpcommit() と同様、この関数は、トランザクションのイニシエータからしか呼び出せません。パーティシパント (つまり、サービス・ルーチン) は、トランザクションをアボートさせたい場合には、TPFAIL を付けて TPRETURN(3) を呼び出します。

未終了の応答に対する呼び出し記述子が存在するときに tpabort() を呼び出すと、この関数の終了時に、トランザクションはアボートし、呼び出し側のトランザクションに関連するこれらの記述子は以後無効になります。呼び出し側のトランザクションに関連がない呼び出し記述子の状態は有効のままです。

トランザクション・モードの会話サーバに対してオープン接続がある場合、tpabort() は TPEV_DISCONIMM イベントをサーバに送ります (そのサーバが接続の制御権を有するかどうかに関係なく)。tpbegin() の前に、あるいは TPNOTRAN フラグを付けて (つまり、トランザクション・モードでない状態で) オープンされた接続は、影響を受けません。

現時点では、tpabort() の唯一の引数 *flags* は将来の用途のために予約されており、0 を設定しておかなければなりません。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpabort() の呼び出しを発行できません。

戻り値

異常終了すると、tpabort() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了すると、tpabort() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
: *flags* が 0 ではありません。呼び出し元のトランザクションは影響を受けません

[TPEHEURISTIC]
: ヒューリスティックな判断、トランザクションに代わって行われた作業が一部はコミットされ、一部はアボートしました。

[TPEHAZARD]
ある種の障害のため、トランザクションの一部としてなされた作業がヒューリスティックに完了している可能性があります。

[TPEPROTO]
tpabort() が (パーティシパントに呼び出されるなど) 不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

注意事項

BEA Tuxedo ATMI システムのトランザクションを記述するために tpbegin()、tpcommit()、および tpabort() を使用する場合、XA インターフェイスに準拠した (呼び出し元に妥当にリンクされている) リソース・マネージャが行う作業のみがトランザクションの特性を備えていることを記憶しておくことが重要です。トランザクションにおいて実行される他のすべての操作は、tpcommit() あるいは tpabort() のいずれにも影響されません。

関連項目

tpbegin(3c)、tpcommit(3c)、tpgetlev(3c)

tpacall(3c)

名前

tpacall()— サービス要求の送信を行うルーチン

形式

```
#include <atmi.h>
int tpacall(char *svc, char *data, long len, long flags)
```

機能説明

tpacall() は、*svc* で指定されているサービスに要求メッセージを送ります。この要求は、以前になされた tpspri() の呼び出しで変更されていないかぎり、*svc* に定義されている優先順位で送信されます。*data* は、NULL でなければ、tpalloc() が以前に割り当てたバッファを指していなければなりません。*len* には送信するバッファに入るデータの量を指定します。ただし、*data* が長さの指定を必要としないバッファを指している場合 (FML フィールド化バッファなど)、*len* は無視されます (0 でかまいません)。*data* が NULL であると、*len* は無視され、要求はデータ部なしで送信されます。*data* のタイプとサブタイプは、*svc* が認識するタイプおよびサブタイプと一致しなければなりません。トランザクション・モードにあるときに送信される要求ごとに、最終的には対応する応答が受信されなければならないことに注意してください。

次に、有効な *flags* の一覧を示します。

TPNOTRAN
: 呼び出しプロセスがトランザクション・モードにあり、このフラグが設定されていると、*svc* が呼び出されたときに、このプロセスは呼び出し元のトランザクションの一部として実行されません。*svc* がトランザクションをサポートしないサーバに属している場合、呼び出し元がトランザクション・モードのときに、このフラグを設定しなければなりません (そうでないと、TPETRAN が返されます)。*svc* が依然としてトランザクション・モードで起動される場合がありますが、それは同じトランザクションでないことに注意してください。*svc* は、コンフィギュレーション属性で、自動的にトランザクション・モードで呼び出されるようにすることができます。このフラグを設定するトランザクション・モードの呼び出し元は、トランザクション・タイムアウトの影響を受けます。このフラグをセットして起動したサービスが異常終了した場合、呼び出し元のトランザクションは影響されません。

TPNOREPLY
: 応答を期待していないことを tpacall() に通知します。TPNOREPLY が設定されると、この関数は正常終了時には 0 を返します。0 は、無効な識別子です。呼び出しプロセスがトランザクション・モードにあるとき、TPNOTRAN が設定されない限りこの設定は使用できません。

TPNOBLOCK
: ブロッキング条件が存在する場合、要求は送られません (たとえば、メッセージを受け取るバッファがいっぱいのときなど)。TPNOBLOCK が指定されていないときにブロッキング条件が存在すると、呼び出し元は、その条件が解消されるか、またはタイムアウト (トランザクションまたはブロッキング) が発生するまではブロックされます。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。トランザクション・タイムアウトは依然として発生する可能性があります。

TPSIGRSTRT
: シグナルが関数内部のシステム・コールを中断すると、中断されたシステム・コールは出しなおされます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tpacall() の呼び出しを発行できません。

戻り値

正常終了の場合、tpacall() は、送信した要求の応答を受信するために使用できる記述子を返します。

異常終了すると、tpacall() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpacall() は tperrno() を次のいずれかの値に設定します (特に指定がない限り、障害は、呼び出し元のトランザクションに影響しません)

[TPEINVAL]
: 無効な引数が与えられました (たとえば、*svc* が NULL であったり、*data* が tpalloc() で割り当てられた領域を指していなかったり、あるいは *flags* が無効です)。

[TPENOENT]
: 存在しないか、会話サービスであるか、名前が "." で始まるため、*svc* 送信できません。

[TPEITYPE]
: *data* のタイプとサブタイプが、*svc* が受け付けるタイプとサブタイプのいずれでもありません。

[TPELIMIT]
: 未終了の非同期要求の数が、保持できる最大数に達したため、呼び出し元の要求が送信できませんでした。

[TPETRAN]
svc が、トランザクションをサポートしていないサーバに属しており、TPNOTRAN が設定されていませんでした。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、そのトランザクションは「アボートのみ」とマークされます。トランザクション・モードにない場合は、ブロッキング・タイムアウトが発生しており、TPNOBLOCK も TPNOTIME も指定されていませんでした。トランザクション・タイムアウトが発生すると、トランザクションがアボートされない限り、新しいリクエストの送信や未処理の応答の受信はできません (ただし、1 つの例外を除く)。これらの操作を行おうとすると、TPETIME が発生して失敗します。1 つの例外とは、ブロックされず、応答を期待せず、かつ呼び出し元のトランザクションのために送信されない (つまり、TPNOTRAN、TPNOBLOCK および TPNOREPLY が設定された状態で tpacall() が呼び出される場合) 要求です。

[TPEBLOCK]
ブロッキング状態のため、TPNOBLOCK が指定されました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tpacall() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。リモート・ロケーションにあるメッセージ・キューがいっぱいになった場合には、tpacall() が正常に復帰しても TPEOS が返されます。

関連項目　tpalloc(3c)、tpcall(3c)、tpcancel(3c)、tpgetrply(3c)、tpgprio(3c)、tpsprio(3c)

tpadmcall(3c)

名前　tpadmcall()— ブートされていないアプリケーションの管理

形式

```
#include <atmi.h>
#include <fml32.h>
#include <tpadm.h>

int tpadmcall(FBFR32 *inbuf, FBFR32 **outbuf, long flags)
```

機能説明

tpadmcall() は、ブートされていないアプリケーションの属性の検索と更新に使用します。また、アクティブなアプリケーションで、限られた属性の集合を直接検索することもできます。この場合、外部プロセスとの通信は必要ありません。この関数は、システム提供のインターフェイス・ルーチンを使用してシステムのコンフィギュレーションと管理が完全に行えるような十分な機能を提供します。

inbuf には、tpalloc() によって以前に割り当てた、希望する管理操作とそのパラメータが入っている FML32 バッファへのポインタを指定します。

outbuf には、結果を入れる FML32 バッファへのポインタのアドレスを指定します。*outbuf* は、元々 tpalloc() によって割り当てられた FML32 バッファを指していなければなりません。送信と受信に同じバッファを使用する場合は、*outbuf* には *inbuf* のアドレスを指定してください。

現在 tpadmcall() の最後の引数の *flags* は将来の使用のために予約されているため、0 に設定しなければなりません。

MIB(5) を調べて、管理要求の構築に関する一般的な情報を得る必要があります。また TM_MIB(5) および APPQ_MIB(5) を調べて、tpadmcall() を通してアクセスできるクラスについて情報を得る必要があります。

tpadmcall() は次の 4 つのモードで呼び出すことができます。

モード 1: ブートされていない、環境設定されていないアプリケーション
呼び出し元は、アプリケーションの管理者であると考えられます。許される操作は、NEW T_DOMAIN クラス・オブジェクトに対して SET を実行してアプリケーションの初期コンフィギュレーションを定義すること、そして APPQ_MIB() で定義されているクラスのオブジェクトに対して GET および SET を実行することだけです。

モード 2: ブートされていない、コンフィギュレーションされたアプリケーション
呼び出し元は、割り当てられた管理者であるか、ローカル・システムの管理者用のコンフィギュレーションで定義された権限と自分の uid/gid を比較した結果に基づく他の権限を持っています。呼び出し元は、TM_MIB() および APPQ_MIB() のあらゆるクラスのあらゆる属性に対して、これらに対して適切なパーミッションを持つ場合に、GET および SET を実行できます。クラスによっては、起動されていないアプリケーションからアクセスできない属性だけを持ち、これらのクラスへのアクセスが失敗する場合があることに注意してください。

モード 3: ブートされたアプリケーション、アタッチされていないプロセス
呼び出し元は、割り当てられた管理者であるか、ローカル・システムの管理者用のコンフィギュレーションで定義された権限と自分の uid/gid を比較した結果に基づく他の権限を持っています。呼び出し元は、TM_MIB() のあらゆるクラスのあらゆる属性に対して、これらに対して適切なパーミッションを持つ場合に、GET を実行できます。同様に呼び出し元は、クラス固有の制限にもよりますが、APPQ_MIB() のあらゆるクラスのあらゆる属性に対して GET および SET を実行できます。ACTIVE であるときにのみアクセスできる属性は返されません。

モード 4: ブートされたアプリケーション、アタッチされているプロセス
tpinit() の実行時に割り当てられた認証キーに従ってパーミッションが決められます。呼び出し元は、TM_MIB() のあらゆるクラスのあらゆる属性に対して、これらに対して適切なパーミッションを持つ場合に、GET を実行できます。さらに呼び出し元は、クラス固有の制限にもよりますが、APPQ_MIB() のあらゆるクラスのあらゆる属性に対して GET および SET を実行できます。

これらのインターフェイス・ルーチンを使用したバイナリの BEA Tuxedo ATMI システム・アプリケーション・コンフィギュレーション・ファイルに対するアクセスおよび更新は、ディレクトリ名やファイル名に関する UNIX システムのパーミッションによって制御されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpadmcall() の呼び出しを発行できません。

環境変数

このルーチンを呼び出す前に、次の環境変数を設定する必要があります。

TUXCONFIG
このアプリケーションに対する BEA Tuxedo システムのコンフィギュレーション・ファイルを保存するファイルまたはデバイスの名前を指定します。

注意事項

tpadmcall() を使用する場合、GET 要求における TA_OCCURS 属性の使用はサポートされていません。tpadmcall() を使用する場合、GETNEXT 要求はサポートされていません。

戻り値

tpadmcall() は成功すると 0 を、失敗すると -1 を返します。

エラー

異常終了時には、tpadmcall() は tperrno() を次のいずれかの値に設定します。

注記 TPEINVAL の場合を除いて、呼び出し元の出力バッファ *outbuf* は、TA_ERROR、TA_STATUS、そして場合によっては TA_BADFLD を含むように変更され、エラー条件についてさらに詳しい情報が得られます。このようにして返されるエラー・コードについて詳しくは、MIB(5)、TM_MIB(5)、および APPQ_MIB(5) を参照してください。

[TPEINVAL]
無効な引数が指定されました。*flags* の値が無効であるか、*inbuf* または *outbuf* は "FML32" タイプの型付きバッファへのポインタではありません。

[TPEMIB]
管理要求が失敗しました。*outbuf* が更新され、MIB(5) および TM_MIB(5) で説明するエラーの原因を示す FML32 のフィールドが設定され、呼び出し側に返されました。

[TPEPROTO]
tpadmcall() が不正に呼び出されました。

[TPERELEASE]
環境変数 TUXCONFIG に別のリリース・バージョンのコンフィギュレーション・ファイルが設定されて、tpadmcall() が呼び出されました。

[TPEOS]
オペレーティング・システムのエラーが発生しました。失敗したシステム・コールを示す数値が Uunixerr に入っています。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。このエラーの正確な内容は userlog() に書き込まれます。

相互運用性

このインターフェイスは、ローカルなコンフィギュレーション・ファイルおよび掲示板に対するアクセスおよび更新しかサポートしていません。したがって、相互運用性の問題はありません。

移植性

このインターフェイスは、BEA Tuxedo ATMI リリース 5.0 またはそれ以降が稼動する UNIX System サイトでしか利用できません。

ファイル

```
${TUXDIR}/lib/libtmib.a, ${TUXDIR}/lib/libqm.a,
${TUXDIR}/lib/libtmib.so.<rel>, ${TUXDIR}/lib/libqm.so.<rel>,
${TUXDIR}/lib/libtmib.lib, ${TUXDIR}/lib/libqm.lib
```

buildclient を使用する場合は、ライブラリを手動でリンクする必要があります。次のように指定します。

```
-L${TUXDIR}/lib -ltmid -lqm
```

関連項目

ACL_MIB(5)、APPQ_MIB(5)、EVENT_MIB(5)、MIB(5)、TM_MIB(5)、WS_MIB(5)
『BEA Tuxedo アプリケーションの設定』
『BEA Tuxedo アプリケーション実行時の管理』

tpadvertise(3c)

名前

tpadvertise()— サービス名の宣言を行うルーチン

形式

```
#include <atmi.h>
int tpadvertise(char *svcname, void (*func)(TPSVCINFO *))
```

機能説明

tpadvertise() 使用するとサーバは提供するサービスを宣言することができます。デフォルトの設定では、サーバのサービスは、サーバのブート時に宣言され、サーバのシャットダウン時にその宣言が解除されます。

複数サーバ単一キュー (MSSQ) セットに属するすべてのサーバは、同じサービス・セットを提供しなければなりません。これらのルーチンは、MSSQ セットを共有する全サーバを宣言することによってこの規則を適用します。

tpadvertise() は、サーバ (あるいは、呼び出し元の MSSQ セットを共有するサーバのセット) の *svcname* を宣言します。*svcname* に NULL あるいは NULL 文字列 ("") を指定することはできません。また、長さは 15 文字までとしてください (UBBCONFIG(5) の SERVICES セクションを参照)。*func* は BEA Tuxedo ATMI システムのサービス関数のアドレスです。この関数は、*svcname* に対する要求をサーバが受け取ったときに起動されます。*func* に NULL を指定することはできません。明示的に指定された関数名 (servopts(5) 参照) は、128 文字までの長さを指定できます。15 文字を超える名前は受け入れられますが、15 文字に短縮されます。短縮された名前が他のサービス名と一致しないように確認する必要があります。

svcname がすでにサーバに対して宣言されていて、*func* がその現在の関数と一致する (すでに宣言されている名前に一致する切り捨てられた名前も含まれます) 場合、tpadvertise() は正常に終了します。ただし、*svcname* がすでにサーバに対して宣言されていても、*func* が現在の関数と一致しない場合には (切り捨てられた名前がすでに宣言されている名前と同じ場合にも)、エラーが返されます。

'.' で始まるサービス名は、管理用サービスのために予約されています。アプリケーションがこれらのサービスの一つを宣言しようとするとエラーが返されます。

戻り値

異常終了すると、tpadvertise() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpadvertise() は tperrno() を次のいずれかの値にセットします

[TPEINVAL]
: *svcname* が NULL または NULL 文字列 ("") であるか、ピリオド (".") で開始するか、または *func* が NULL である場合。

[TPELIMIT]
領域に制限があるため、*svcname* を宣言できない場合 (UBBCONFIG(5) の RESOURCES セクションの MAXSERVICES を参照)。

[TPEMATCH]
svcname がサーバに対してすでに宣言されているが、*func* 以外の関数で宣言されている場合。この関数は異常終了しても、*svcname* はその現在の関数で宣言されたまま変わりません (すなわち、*func* は現在の関数と振り変わりません)。

[TPEPROTO]
tpadvertise() が不正なコンテキストで呼び出された場合 (たとえば、クライアントによって)。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目 tpservice(3c)、tpunadvertise(3c)

tpalloc(3c)

名前

tpalloc()— 型付きバッファの割り当てを行うルーチン

形式

```
#include <atmi.h>
char * tpalloc(char *type, char *subtype, long size)
```

機能説明

tpalloc() は、*type* タイプのバッファを指すポインタを返します。バッファのタイプによっては、*subtype* と *size* は両方とも省略することができます。BEA Tuxedo ATMI システムには、様々なタイプのバッファが提供されており、しかもアプリケーションは自由に独自のバッファ・タイプを追加することができます。詳細については、tuxtypes(5) を参照してください。

ある特定のバッファ・タイプの tmtype_sw で *subtype* が NULL でない場合、tpalloc() を呼び出す際に *subtype* を指定しなければなりません。割り当てるバッファは少なくとも *size* および dfltsize と同じ大きさにします。ここで、dfltsize は特定のバッファ・タイプの tmtype_sw に指定するデフォルトのバッファ・サイズです。バッファ・タイプが STRING の場合、最小は 512 バイトです。バッファ・タイプが FML あるいは VIEW の場合、最小は、1024 バイトです。

なお、*type* の最初の 8 バイトと *subtype* の最初の 16 バイトだけが有効です。

あるバッファ・タイプは使用する前に初期化が必要です。tpalloc() は、バッファが割り当てられて返される前に (BEA Tuxedo ATMI システム固有の方法で) バッファを初期化します。このため、呼び出し元から返されるバッファはすぐに使用できます。ただし、初期化ルーチンがバッファをクリアしないかぎり、そのバッファは tpalloc() によってゼロに初期化されません。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tpalloc() の呼び出しを発行できます。

戻り値

正常終了の場合、tpalloc() は long ワードの適切なタイプのバッファを指すポインタを返します。それ以外の場合は NULL を返し、tperrno() を設定して条件を示します。

エラー

異常終了時には、tpalloc() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
: 無効な引数が与えられた場合 (たとえば、*type* が NULL である場合)。

[TPENOENT]
: tmtype_sw のどのエントリも *type* と一致しない場合。また、subtype が NULL でなければ、*subtype* とも一致しない場合。

[TPEPROTO]
tpalloc() が不正なコンテキストで呼び出された場合。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

使用法

バッファの初期化が失敗した場合、割り当てられたバッファは解放され、tpalloc() は、異常終了して NULL を返します。

この関数を C のライブラリである malloc()、realloc()、あるいは free() と同時に使用してはいけません (たとえば、tpalloc() を使用して割り当てられたバッファを free() を使用して解放してはいけません)。

BEA Tuxedo ATMI システムの拡張に準拠したインプリメンテーションは、すべて 2 つのバッファ・タイプをサポートしています。詳細については、「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」を参照してください。

関連項目

tpfree(3c)、tprealloc(3c)、tptypes(3c)

tpbegin(3c)

名前

tpbegin()— トランザクションを開始するためのルーチン

形式

```
#include <atmi.h>
int tpbegin(unsigned long timeout, long flags)
```

機能説明

BEA Tuxedo ATMI システムにおけるトランザクションは、完全に成功するか、あるいは何も影響を残さない 1 つの論理的な作業単位を定義するときに使用します。トランザクションにより、多くのプロセスで (そして、多分様々なサイトで) 行われる作業を 1 つの分割できない単位として扱うことができます。トランザクションのイニシエータは tpbegin() とともに、tpcommit() または tpabort() を使用して、トランザクション内での処理を記述します。いったん tpbegin() が呼び出されると、他のプロセスとの通信は、後のプロセス (もちろん、サーバ) をトランザクション・モードにすることができます (つまり、サーバの作業はトランザクションの一部となります)。トランザクションに参加したプロセスをパーティシパントと呼びます。トランザクションには、必ず 1 つのイニシエータがあり、いくつかのパーティシパントをもつことができます。tpcommit() または tpabort() を呼び出せるのは、トランザクションの実行元だけです。パーティシパントは、tpreturn() を呼び出したときに使用する戻り値 (*rval*s) によって、トランザクションの結果に影響を与えることができます。いったんトランザクション・モードになると、サーバに出されたすべてのサービス要求は、トランザクションの一部として処理されます (明示的にリクエスタからのそれ以外の指定がなければ)。

また、会話サーバに対して確立されたオープン接続があるときに、プログラムがトランザクションを開始しても、これらの接続はトランザクション・モードには変わりません。これは、tpconnect() の呼び出し時に TPNOTRAN フラグを指定したことと同じです。

tpbegin() の最初の引数 *timeout* は、トランザクションのタイムアウトまでの時間を最低 *timeout* 秒にすることを指定します。トランザクションはタイムアウト時間を経過した後は、「アボートのみ」としてマーク付けされます。*timeout* の値が 0 であると、トランザクションにはシステムが許すかぎりの最大時間 (秒単位) のタイムアウト値が与えられます (つまり、このときのタイムアウト値は、システムが定義している符号なし long 型の最大値になります)。

現時点では、tpbegin() の第 2 引数 *flags* は将来の用途のために予約されており、0 に設定しておかなければなりません。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態 のスレッドは、tpbegin() の呼び出しを発行できません。

戻り値

異常終了すると、tpbegin() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpbegin() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
flags が 0 ではありません。

[TPETRAN]
呼び出し元は、トランザクション開始時にエラーが発生したため、トランザクション・モードになれません。

[TPEPROTO]
tpbegin() が不正なコンテキストで呼び出されました (たとえば、呼び出し元がすでにトランザクション・モードにある)。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

注意事項

BEA Tuxedo ATMI システムのトランザクションを記述するために tpbegin()、tpcommit()、および tpabort() を使用する場合、XA インターフェイスに準拠した (呼び出し元に妥当にリンクされている) リソース・マネージャが行う作業のみがトランザクションの特性を備えていることを記憶しておくことが重要です。トランザクションにおいて実行される他のすべての操作は、tpcommit() あるいは tpabort() のいずれにも影響されません。リソース・マネージャによって実行される操作が、BEA Tuxedo ATMI システムのトランザクションの一部となるように、XA インターフェイスを満たすリソース・マネージャをサーバにリンクします。詳しくは、buildserver() を参照してください。

関連項目

tpabort(3c)、tpcommit(3c)、tpgetlev(3c)、tpscmt(3c)

tpbroadcast(3c)

名前　　tpbroadcast()— 名前によって通知をブロードキャストするルーチン

形式

```
#include <atmi.h>

int tpbroadcast(char *lmid, char *usrname, char *cltname,
  char *data,  long len, long flags)
```

機能説明

tpbroadcast() は、クライアント・プロセスやサーバ・プロセスがシステム内に登録されているクライアントに任意通知型メッセージを送ることができるようにします。ターゲット・クライアント・セットは、tpbroadcast() に渡されるワイルドカード以外の識別子すべてと一致するクライアントで構成されます。ワイルドカードは、識別子を指定するのに使用します。

lmid、*usrname* および *cltname* は、ターゲット・クライアント・セットの選択に使用する論理識別子です。引数の NULL 値は、その引数のワイルドカードとなります。ワイルドカード引数は、そのフィールドの全クライアント識別子と一致します。長さ 0 の文字列の引数は、長さ 0 のクライアント識別子とのみ一致します。各識別子は、システムが有効とみなすよう定義されたサイズの制約事項を満たさなければなりません。つまり、各識別子の長さは 0 から MAXTIDENT 文字までででなければなりません。

要求のデータ部は、*data* によって示され、以前に tpalloc() によって割り当てられたバッファです。*len* には送信するデータの大きさを指定します。ただし、*data* が長さの指定を必要としないタイプのバッファを指す場合 (たとえば、FML フィールド化バッファ)、*len* は 0 でかまいません。また、*data* は NULL であってもかまいません。この場合、*len* は無視されます。このバッファは、他の送受信されるメッセージと同様、型付きバッファ・スイッチ・ルーチンで処理されます。たとえば、符号化 / 復号化は自動的に行われます。

次に、有効な *flags* の一覧を示します。

TPNOBLOCK
: ブロッキング条件が存在する場合、要求は送られません (たとえば、メッセージを受け取るバッファがいっぱいのときなど)。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。トランザクション・タイムアウトは依然として発生する可能性があります。

TPSIGRSTRT
: シグナルが関数内部のシステム・コールを中断すると、中断されたシステム・コールは出しなおされます。tpbroadcast() が正常終了した場合には、メッセージはシステムに渡され、選択されたクライアントに転送されます。tpbroadcast() は、選択された各クライアントにメッセージが送られるのを待機しません。

  マルチスレッドのアプリケーションでは、TPINVALIDCONTEXT 状態のスレッドは tpbroadcast() の呼び出しを発行できません。

戻り値

異常終了すると、tpbroadcast() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了には、tpbroadcast() はアプリケーション・クライアントにブロードキャスト・メッセージを送信せず、tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
: 無効な引数が与えられました (たとえば、識別子が長すぎる、あるいはフラグが無効など)。無効な LMID を使用すると、tpbroadcast() は正常に働かず、TPEINVAL が返されます。ただし、存在しないユーザやクライアント名の場合は、誰にもブロードキャストされないだけでこのルーチンは正常に終了します。

[TPETIME]
: ブロッキング・タイムアウトが生じましたが、TPNOBLOCK も TPNOTIME も指定されていませんでした。

[TPEBLOCK]
: 呼び出し時にブロッキング条件が検出されましたが、TPNOBLOCK が指定されていました。

[TPGOTSIG]
: シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
: tpbroadcast() が不正に呼び出されました。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

移植性

tpnotify(3c) で説明したインターフェイスはすべて、ネイティブ・サイトの UNIX システムベースのプロセッサ上で利用できます。さらに、ルーチン tpbroadcast() と tpchkunsol() は、関数 tpsetunsol() ともとに、UNIX システムおよび MS-DOS ベースのプロセッサ上で利用することができます。

使用法

シグナル・ベースの通知方法を選択したクライアントは、シグナルに関する制約から、システムによるシグナルを受け取ることはできません。このような状態で通知がなされた場合、システムは、選択されたクライアントに対する通知方法をディップインに切り替えることを示すログ・メッセージを生成し、以後、クライアントにはディップイン方式で通知が行われることになります (通知方法の詳細については、UBBCONFIG( ) の RESOURCES セクションの NOTIFY パラメータの説明を参照してください)。

クライアントのシグナル通知は、必ずシステムによって行われるので、通知呼び出しの起動元がどこであっても、通知の動作は一貫しています。したがって、シグナル・ベースの通知を使用するには以下の条件が必要です。

- ネイティブ・クライアントが、アプリケーション管理者として実行している必要があります。
- ワークステーション・クライアントは、アプリケーション管理者として実行している必要はありません。

ID は、アプリケーション管理者のアプリケーションのコンフィギュレーションの一部として識別されます。

あるクライアントに対してシグナル・ベースの通知方法を選択すると、いくつかの ATMI 呼び出しは異常終了します。TPSIGRSTRT の指定がなければ任意通知型メッセージを受け取るため、TPGOTSIG を返します。通知方法の選択方法については、UBBCONFIG(5) および tpinit(3c) を参照してください。

関連項目

tpalloc(3c)、tpinit(3c)、tpnotify(3c)、tpterm(3c)、UBBCONFIG(5)

tpcall(3c)

名前

tpcall()— サービス要求を送信し、その応答を待つルーチン

形式

```
int tpcall(char *svc, char *idata, long ilen, char **odata, long \
   *olen, long flags)
```

機能説明

tpcall() は、要求を送り、それと同期してその応答を待ちます。この関数への呼び出しは、tpacall() を呼び出した後、即座に tpgetrply() を呼び出すのと同じことです。tpcall() は、*svc* が指定するサービスに要求を送ります。この要求は、以前の tpspri() で変更されていないかぎり、*svc* に対して定義されている優先順位で送信されます。要求のデータ部分は、*idata* によって示されます。これは、あらかじめ tpalloc() によって割り当てられるバッファです。*ilen* は、送信する *idata* の大きさを指定します。なお、*idata* は、長さの指定を必要としないタイプのバッファを指している場合 (たとえば、FML のフィールド化バッファ)、*ilen* の値は無視されます (あるいは 0) にします。また、*idata* を NULL にすることもできますが、この場合には、*ilen* は無視されます。*idata* のタイプとサブタイプは、*svc* が認識するタイプおよびサブタイプのいずれかと一致しなければなりません。

odata は、応答が読み込まれるバッファを指すアドレス、*olen* は、その応答の長さを示します。**odata* は、もともと tpalloc() によって割り当てられたバッファを指していなければなりません。同じバッファを送信と受信の両方に使用する場合には、*odata* を *idata* のアドレスに設定してください。FML と FML32 バッファは、通常最小サイズ 4096 バイトを確保します。したがって、応答が 4096 バイトより大きい場合には、バッファ・サイズは返されるデータを入れるのに十分な大きさに拡大します。また、tpcall() が呼び出されたときに *idata* と **odata* が同じであり、**odata* が変更された場合、*idata* は有効なアドレスを指しません。古いアドレスの使用は、データの破損やプロセスの例外の原因になります。リリース 6.4 では、バッファに対するデフォルトの割り当ては 1024 バイトです。また、最近使用したバッファの履歴情報が保持され、最適サイズのバッファをリターン・バッファとして再利用できます。

容量まで満たされていないかもしれない送信者側のバッファ (たとえば、FML または FML32 バッファ) は、送信に使用されたサイズになります。システムは、受信データのサイズを任意の量で拡大します。これは、受信者が送信者の割り当てたバッファ・サイズより小さく、送信されたデータのサイズより大きいバッファを受け取ることを意味します。

受信バッファのサイズは、増加することも減少することもあります。また、アドレスもシステムがバッファを内部で交換するごとに常に変更されます。応答バッファのサイズが変わったどうか (また変わったとしたらどれくらい変わったのか) を決定するには、tpgetrply() が *len とともに発行される前に、合計サイズを比べてください。バッファ管理の詳細については、「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」を参照してください。

*olen が戻り時に 0 であると、応答にはデータ部がなく、*odata も、それが指すバッファも変更されていません。*odata または *olen が NULL であると、エラーになります。

次に、有効な flags の一覧を示します。

TPNOTRAN
: 呼び出しプロセスがトランザクション・モードにあり、このフラグが設定されていると、svc が呼び出されたときに、このプロセスは呼び出し元のトランザクションの一部として実行されません。svc が依然としてトランザクション・モードで起動される場合がありますが、それは同じトランザクションでないことに注意してください。svc は、コンフィギュレーション属性で、自動的にトランザクション・モードで呼び出されるようにすることができます。このフラグを設定するトランザクション・モードの呼び出し元は、トランザクション・タイムアウトの影響を受けます。このフラグをセットして起動したサービスが異常終了した場合、呼び出し元のトランザクションは影響されません。

TPNOCHANGE
: デフォルトの設定では、*odata で示されるバッファとタイプの異なるバッファを受け取った場合、受信プロセスが入ってくるバッファのタイプを認識できるかぎり、*odata のバッファ・タイプが受信バッファのタイプに変わります。このフラグを設定しておくと、*odata が指すバッファのタイプは変更されません。つまり、受信したバッファのタイプとサブタイプは、*odata が指すバッファのタイプとサブタイプと一致しなければなりません。

TPNOBLOCK
: ブロッキング条件が存在する場合、要求は送られません (たとえば、メッセージを受け取るバッファがいっぱいのときなど)。ただし、このフラグは tpcall() の送信部分にしか適用されません。この関数は応答を待ってブロックすることがあります。TPNOBLOCK が指定されていないときにブロッキング条件が存在すると、呼び出し元は、その条件が解消されるか、またはタイムアウト (トランザクションまたはブロッキング) が発生するまではブロックされます。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。ただし、呼び出し元がトランザクション・モードの場合、このフラグの効果はありません。この場合、呼び出し元はトランザクション・タイムアウトの制限に従います。トランザクション・タイムアウトは依然として発生する可能性があります。

TPSIGRSTRT
: シグナルが関数内部のシステム・コールを中断すると、中断されたシステム・コールは出しなおされます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpcall() の呼び出しを発行できません。

戻り値

tpcall() が正常に終了した場合、あるいは tperrno() が TPESVCFAIL に設定されて終了した場合、tpurcode() には、tpreturn() の一部として送信されたアプリケーションが定義した値が入ります。

異常終了すると、tpcall() は -1 を返し、tperrno() を設定してエラー条件を示します。呼び出しが特定の tperrno() 値で異常終了した場合、中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、生成されたエラーの詳細が提供されます。詳細については、tperrordetail(3c) リファレンス・ページを参照してください。

エラー

異常終了時には、tpcall() は tperrno() を次のいずれかの値に設定します (特に指定がない限り、障害は、呼び出し元のトランザクションに影響しません)。

[TPEINVAL]
: 無効な引数が指定されました (たとえば、*svc* が NULL である場合、あるいは *flags* が無効である場合など)。

[TPENOENT]
: プロセスが存在しないか、会話サービスであるか、または名前が "." で始まるため、*svc* に送信できません。

[TPEITYPE]
: *idata* のタイプとサブタイプが、*svc* が受け付けることのできるものでありません。

[TPEOTYPE]
応答のタイプおよびサブタイプは呼び出し元が認識できないものです。あるいは、TPNOCHANGE が *flags* に設定されているか、**odata* のタイプとサブタイプがそのサービスから送られた応答のタイプおよびサブタイプと一致しません。**odata*、その内容、そして **olen* はいずれも変更されません。サービス要求が呼び出し元の現在のトランザクションの一部として発行されると、応答が捨てられるためそのトランザクションは「アボートのみ」とマーク付けされます。

[TPETRAN]
svc が、トランザクションをサポートしていないサーバに属しており、TPNOTRAN が設定されていませんでした。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、そのトランザクションは「アボートのみ」とマークされます。トランザクション・モードにない場合は、ブロッキング・タイムアウトが発生しており、TPNOBLOCK も TPNOTIME も指定されていませんでした。いずれのケースでも、**odata*、その内容、**olen* はどれも変更されません。トランザクション・タイムアウトが発生すると、トランザクションがアボートされない限り、新しいリクエストの送信や未処理の応答の受信はできません (ただし、1 つの例外を除く)。これらの操作を行おうとすると、TPETIME が発生して失敗します。ブロックされない要求が応答を期待するものでなく、呼び出し元のトランザクションのために送信されないものである場合は例外です (つまり、TPNOBLOCK および TPNOREPLY を設定して tpacall() を呼び出した場合)。

[TPESVCFAIL]
呼び出し元の応答を送るサービス・ルーチンが、TPFAIL を設定した状態で tpreturn() を呼び出しました。これは、アプリケーション・レベルの障害です。サービスの応答の内容 (送信された場合) は、**odata* が示すバッファに入ります。呼び出し元のトランザクションの一部としてサービス要求が出された場合、トランザクションは「アボートのみ」とマーク付けされます。トランザクションがタイムアウトしたかどうかに関わりなく、トランザクションがアボートされる前の有効な通信だけが、TPNOREPLY、TPNOTRAN、および TPNOBLOCK を設定した tpacall() を呼び出します。

[TPESVCERR]
サービス・ルーチンが tpreturn(3c) または tpforward(3c) のいずれかでエラーを検出しました (たとえば、誤った引数が渡された場合など)。このエラーが生じた場合には、応答データは返されません (つまり、*odata、その内容、*olen はいずれも変更されません)。呼び出し元のトランザクションのためにサービス要求が出された場合 (つまり、TPNOTRAN が設定されていない場合)、このトランザクションには「アボートのみ」のマークが付けられます。トランザクションがタイムアウトしたかどうかに関わりなく、トランザクションがアボートされる前の有効な通信だけが、TPNOREPLY、TPNOTRAN、および TPNOBLOCK を設定した tpacall() を呼び出します。ubbconfig ファイル中の SVCTIMEOUT か、TM_MIB 中の TA_SVCTIMEOUT が 0 でない場合にサービスのタイムアウトが発生すると、TPESVCERR が返されます。

[TPEBLOCK]
送信呼び出しでブロッキング条件が検出されましたが、TPNOBLOCK が指定されていました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tpcall() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。リモート・ロケーションにあるメッセージ・キューがいっぱいになった場合には、tpcall() が正常に復帰しても TPEOS が返されることがあります。

関連項目　tpacall(3c)、tpalloc(3c)、tperrordetail(3c)、tpforward(3c)、tpfree(3c)、tpgprio(3c)、tprealloc(3c)、tpreturn(3c)、tpsprio(3c)、tpstrerrordetail(3c)、tptypes(3c)

tpcancel(3c)

名前

tpcancel()— 未終了の応答に対する呼び出し記述子を無効にするためのルーチン

形式

```
#include <atmi.h>
int tpcancel(int cd)
```

機能説明

tpcancel() は、tpacall() が返す呼び出し記述子 *cd* を取り消します。トランザクションに関連している呼び出し記述子を無効にしようとするとエラーになります。

正常終了の場合、*cd* は以後無効になり、*cd* のために受信する応答はすべて、何の警告もなく捨てられてしまいます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpcancel() の呼び出しを発行できません。

戻り値

異常終了すると、tpcancel() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpcancel() は tperrno() を次のいずれかの値に設定します。

[TPEBADDESC]
: *cd* は、無効な記述子です。

[TPETRAN]
: *cd* は、呼び出し元のトランザクションに関連しています。*cd* はそのまま有効で、呼び出し元の現在のトランザクションは影響を受けません。

[TPEPROTO]
: tpcancel() が不正に呼び出されました。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

関連項目

tpacall(3c)

tpchkauth(3c)

名前

tpchkauth()— アプリケーションへの参加に認証が必要であるか検査するルーチン

形式

```
#include <atmi.h>

int tpchkauth(void)
```

機能説明

tpchkauth() は、アプリケーションのコンフィギュレーションが認証を必要としているかどうかを調べます。これは一般に、tpinit() を呼び出す前にアプリケーション・クライアントが使用して、ユーザからのパスワードの入力を必要とするかどうかを判別します。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpchkauth() の呼び出しを発行できません。

戻り値

tpchkauth() は、正常終了時に次のような負でない値を返します。

TPNOAUTH
: 認証が必要とされないことを示します。

TPSYSAUTH
: システムの認証のみが必要とされることを示します。

TPAPPAUTH
: システムの認証およびアプリケーション固有の認証の両方が必要であることを示します。

異常終了すると、tpchkauth() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpchkauth() は tperrno() を次のいずれかの値に設定します。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

相互運用性

tpchkauth() は、リリース 4.2 以降を使用しているサイトでしか使用できません。

移植性

tpchkauth(3c) に記述されているインターフェイスは、UNIX、Windows および MS-DOS オペレーティング・システム上でサポートされています。

関連項目

tpinit(3c)

tpchkunsol(3c)

名前　tpchkunsol()— 任意通知型メッセージを検査するルーチン

形式

```
#include <atmi.h>
int tpchkunsol(void)
```

機能説明　tpchkunsol() は、クライアントが任意通知型メッセージの検査を行うときに使用します。クライアントでシグナル・ベースの通知方法を使用してこのルーチンを呼び出しても、何も行われず、ただちに終了します。この呼び出しには引数はありません。このルーチンを呼び出すと、アプリケーションで定義された任意通知型メッセージ処理ルーチンへの呼び出しが BEA Tuxedo ATMI システム・ライブラリによって行われることがあります。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpchkunsol() の呼び出しを発行できません。

戻り値　正常終了の場合、tpchkunsol() は、ディスパッチされた任意通知型メッセージの番号を返します。異常終了すると、tpchkunsol() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー　異常終了時には、tpchkunsol() は tperrno() を次のいずれかの値に設定します。

[TPEPROTO]
: tpchkunsol() が不正なコンテキストで呼び出されました (たとえば、サーバ内から)。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

移植性　tpnotify(3c) で説明したインターフェイスはすべて、ネイティブ・サイトの UNIX システムベースのプロセッサ上で利用できます。さらに、ルーチン tpbroadcast() と tpchkunsol() は、関数 tpsetunsol() ともとに、UNIX システムおよび MS-DOS ベースのプロセッサ上で利用することができます。

シグナル・ベースの通知方法を選択したクライアントは、シグナルに関する制約から、システムによるシグナルを受け取ることはできません。このような状態で通知がなされた場合、システムは、選択されたクライアントに対する通知方法をディップインに切り替えることを示すログ・メッセージを生成し、以後、クライアントにはディップイン方式で通知が行われることになります (通知方法の詳細については、UBBCONFIG(5) の RESOURCES セクションの NOTIFY パラメータの説明を参照してください)。

クライアントのシグナル通知は、必ずシステムによって行われるので、通知呼び出しの起動元がどこであっても、通知の動作は一貫しています。したがって、シグナル・ベースの通知を使うには以下の条件が必要です。

- ネイティブ・クライアントはアプリケーション管理者として実行している必要があります。
- ワークステーション・クライアントはアプリケーション管理者として実行している必要はありません。

ID は、アプリケーション管理者のアプリケーションのコンフィギュレーションの一部として識別されます。

あるクライアントに対してシグナル・ベースの通知方法を選択すると、いくつかの ATMI 呼び出しは異常終了します。TPSIGRSTRT の指定がなければ任意通知型メッセージを受け取るため、TPGOTSIG を返します。通知方法の選択の仕方については、UBBCONFIG(5) および tpinit(3c) を参照してください。

関連項目　tpbroadcast(3c)、tpinit(3c)、tpnotify(3c)、tpsetunsol(3c)

tpclose(3c)

名前

tpclose()— リソース・マネージャをクローズするルーチン

形式

```
#include <atmi.h>
int tpclose(void)
```

機能説明

tpclose() は、呼び出し元とそれにリンクされたリソース・マネージャとの関係を絶ちます。個々のリソース・マネージャのクローズ・セマンティクスは、それぞれ異なるため、指定されたリソース・マネージャをクローズするための特定情報は、コンフィギュレーション・ファイルに置かれます。

リソース・マネージャがすでにクローズしている場合 (すなわち、tpclose() が 1 回以上呼び出された)、何も処理は行われず、正常終了を示すコードが返されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpclose() の呼び出しを発行できません。

戻り値

異常終了すると、tpclose() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpclose() は tperrno() を次のいずれかの値に設定します。

[TPERMERR]
: リソース・マネージャが正しくクローズできませんでした。より詳しい理由については、リソース・マネージャを独自の方法で調べることによって得ることができます。ただし、エラーの正確な性質を判別するための呼び出しを使用すると、移植性が損なわれます。

[TPEPROTO]
: tpclose() が不正なコンテキストで呼び出されました (たとえば、呼び出し元がトランザクション・モードにあるとき)。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

関連項目

tpopen(3c)

tpcommit(3c)

名前

TPCOMMIT(3cbl)— 現在のトランザクションをコミットするルーチン

形式

```
#include <atmi.h>
int tpcommit(long flags)
```

機能説明

tpcommit() はトランザクションの終了 (コミット) を示します。tpcommit() は 2 フェーズ・コミット・プロトコルを使用して、パーティシパント間の調整をとります。tpcommit() は、トランザクションのイニシエータからのみ呼び出されます。いずれかのパーティシパントがトランザクションをコミットできない場合 (たとえば、それらが tpreturn() を呼び出したときに TPFAIL が返された場合など)、そのトランザクション全体がアボートし、tpcommit() は異常終了します。つまり、そのトランザクションに関連して各プロセスが行ったすべての作業は取り消されます。すべてのパーティシパントがトランザクションの中のそれぞれが担当する部分のコミットを決定した場合、この決定は安定記憶装置に記録された後、すべてのパーティシパントに対して作業のコミットが要求されます。

コミットの決定が記録された後、あるいは 2 フェーズ・コミット・プロトコルが完了した後、TP_COMMIT_CONTROL 特性の設定条件に従って (tpscmt(3c) を参照)、tpcommit() は正常に終了することができます。コミットの決定が記録された後、第 2 フェーズが完了する前 (TP_CMT_LOGGED) に tpcommit() が終了する場合、すべてのパーティシパントはトランザクションに代わって行った作業内容をコミットすることに同意していると見なし、第 2 フェーズでトランザクションをコミットする約束を果たすようにする必要があります。ただし、tpcommit() は第 2 フェーズが完了する前に終了してしまうので、パーティシパントの中には、この関数が正常終了した場合でもトランザクションの担当部分をヒューリスティックに (コミットの決定とは矛盾するような方法で) 完了するといった状況が発生してしまいます。

TP_COMMIT_CONTROL 特性が、2 フェーズ・コミット・プロトコルの完了 (TP_CMT_COMPLETE) 後に tpcommit() が終了するように設定されている場合、その戻り値には、トランザクションの正確な状態が反映されます (つまり、トランザクションがヒューリスティックに完了するかどうか)。

なお、トランザクションに 1 つのリソース・マネージャしか関与していない場合には、1 フェーズ・コミットが行われます (つまり、リソース・マネージャには、コミットできるかどうかの確認はされず、単にコミットの指示が出されます)。そして、この場合、TP_COMMIT_CONTROL 特性はコミットには関係せず、tpcommit() はヒューリスティックに得られた結果 (もしあれば) を返します。

未終了の応答に対する呼び出し記述子が存在するときに tpcommit() を呼び出すと、この関数の終了時に、トランザクションはアボートし、呼び出し元のトランザクションに関連しているこれらの記述子は以後無効になります。呼び出し側のトランザクションに関連がない呼び出し記述子の状態は有効のままです。

tpcommit() は、呼び出し元のトランザクションに関連しているすべての接続をクローズしたあと呼び出されなければなりません (そうでないと、TPEABORT が返され、トランザクションはアボートし、これらの接続は、TPEV_DISCONIMM イベントを使用して不規則に切断されます)。tpbegin() の前あるいは TPNOTRAN フラグの指定を付けてオープンされた接続 (つまり、トランザクション・モードでない状態での接続) は、tpcommit() または tpabort() の影響を受けません。

現時点では、tpcommit() の唯一の引数 *flags* は、将来の用途のために予約されており、0 を設定しておかなければなりません。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpcommit() の呼び出しを発行できません。

戻り値

異常終了すると、tpcommit() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpcommit() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
flags が 0 ではありません。呼び出し元のトランザクションは影響を受けません。

[TPETIME]
トランザクションがタイムアウトし、トランザクションの状態が不明です (つまり、そのトランザクションはコミットされているかもしれないし、アボートしているかもしれません)。ただし、トランザクションがタイムアウトし、その状態がアボートであることが分かっている場合には、TPEABORT が返されます。

[TPEABORT]
トランザクションの実行元あるいはそのパーティシパントが行った作業をコミットできなかったために、そのトランザクションをコミットできませんでした。また、このエラーは、tpcommit() が未終了の応答が残っているか、会話型接続をオープンしたまま呼び出された場合にも返されます。

[TPEHEURISTIC]
ヒューリスティックな判断、トランザクションに代わって行われた作業が一部はコミットされ、一部はアボートしました。

[TPEHAZARD]
ある種の障害のため、トランザクションの一部としてなされた作業がヒューリスティックに完了している可能性があります。

[TPEPROTO]
tpcommit() が不正なコンテキストで呼び出されました (たとえば、パーティシパントにより呼び出されるなど)。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

注意事項

BEA Tuxedo ATMI システムのトランザクションを記述するために tpbegin()、tpcommit()、および tpabort() を使用する場合、XA インターフェイスに準拠した (呼び出し元に妥当にリンクされている) リソース・マネージャが行う作業のみがトランザクションの特性を備えていることを記憶しておくことが重要です。トランザクションにおいて実行される他のすべての操作は、tpcommit() あるいは tpabort() のいずれにも影響されません。リソース・マネージャによって実行される操作が、BEA Tuxedo ATMI システムのトランザクションの一部となるように、XA インターフェイスを満たすリソース・マネージャをサーバにリンクします。詳細については buildserver(1) を参照してください。

関連項目

tpabort(3c)、tpbegin(3c)、tpconnect(3c)、tpgetlev(3c)、tpreturn(3c)、tpscmt(3c)

tpconnect(3c)

名前　　tpconnect()— 会話型サービス・ルーチンの接続を設定するルーチン

形式

```
#include <atmi.h>

int tpconnect(char *svc, char *data, long len, long flags)
```

機能説明

tpconnect() により、プログラムは会話型サービス *svc* との半二重接続をセットアップすることができます。この名前は、会話型サーバがポストした会話型サービス名の１つでなければなりません。

呼び出し元は、接続セットアップ処理の一部として、アプリケーション定義データをリスニング・プログラムに渡すことができます。呼び出し元がデータを渡す選択をした場合には、*data* は tpalloc() が以前に割り当てたバッファを指していなければなりません。*len* には送信バッファの大きさを指定します。ただし、*data* が長さの指定を必要としないバッファを指している場合 (FML フィールド化バッファなど)、*len* は無視されます (0 でかまいません)。また、*data* は NULL の場合もあります。この場合、*len* は無視されます (アプリケーション・データは会話型サービスに渡されません)。*data* のタイプとサブタイプは、*svc* が認識するタイプおよびサブタイプと一致しなければなりません。*data* と *len* は、該当サービスの呼び出し時に使用する TPSVCINFO 構造体を介して会話型サービスに渡されます。また、サービスはこのデータの獲得に tprecv() を呼び出す必要はありません。

次に、有効な *flags* の一覧を示します。

TPNOTRAN
: 呼び出しプロセスがトランザクション・モードにあり、このフラグが設定されていると、*svc* が呼び出されたときに、このプロセスは呼び出し元のトランザクションの一部として実行されません。*svc* が依然としてトランザクション・モードで起動される場合がありますが、それは同じトランザクションでないことに注意してください。*svc* は、コンフィギュレーション属性で、自動的にトランザクション・モードで呼び出されるようにすることができます。このフラグを設定するトランザクション・モードの呼び出し元は、トランザクション・タイムアウトの影響を受けます。このフラグをセットして起動したサービスが異常終了した場合、呼び出し元のトランザクションは影響されません。

TPSENDONLY
: 呼び出し元は、最初にそれがデータの送信のみを行え、呼び出されたサービスはデータの受信のみを行えるよう接続を設定しなければならないことがあります (つまり、呼び出し元が当初の接続の制御権を有するように)。TPSENDONLY または TPRECVONLY のいずれかを指定しなければなりません。

TPRECVONLY
: 呼び出し元は、それがデータの受信のみを行え、呼び出されたサービスがデータの送信のみを行えるように接続を設定しなければならないことがあります (つまり、呼び出されるサービスが当初、接続の制御権を有するように)。TPSENDONLY または TPRECVONLY のいずれかを指定しなければなりません。

TPNOBLOCK
: ブロッキング条件が存在する場合、接続が設定されておらず、データが送信されません (たとえば、メッセージが送信されるときに使用されるデータ・バッファがいっぱいである場合)。このフラグは、tpconnect() の送信部分にだけ適用されます。関数は、サーバからの承認を待つ間ブロックする場合があります。TPNOBLOCK が指定されておらず、ブロッキング条件が存在する場合、条件の解除、またはブロッキング・タイムアウトあるいはトランザクション・タイムアウトが発生するまで呼び出し元はブロックされます。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。ただし、トランザクション・タイムアウトはあいかわらず有効です。

TPSIGRSTRT
: ルーチン内部のシステム・コールがシグナルによって中断された場合、中断された呼び出しが再発行されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpconnect() の呼び出しを発行できません。

戻り値

tpconnect() が正常に終了した場合、以後の呼び出しでの接続を指すのに使用する記述子を返します。エラー時には、-1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpconnect() は tperrno() を次のいずれかの値に設定します (特に指定がない限り、障害は、呼び出し元のトランザクションに影響しません)

[TPEINVAL]
: 無効な引数が与えられました (たとえば、*svc* が NULL、*data* が NULL でなく tpalloc() で割り当てられたバッファを指していない、TPSENDONLY または TPRECVONLY が *flags* に指定されていない、あるいは *flags* が無効であるなど)。

[TPENOENT]
: *svc* が存在しないか、会話型サーバでないか、または名前が "." で始まるため、svc への接続を開始できません

[TPEITYPE]
data のタイプおよびサブ・タイプは、*svc* が受け付けるタイプあるいはサブ・タイプの１つでありません。

[TPELIMIT]
未終了の接続の最大数に達したため、呼び出し元の要求が送られませんでした。

[TPETRAN]
svc はトランザクションをサポートしないプログラムに属していますが、TPNOTRAN が設定されていませんでした。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、そのトランザクションは「アボートのみ」とマークされます。トランザクション・モードでなければ、ブロッキング・タイムアウトが発生し、TPNOBLOCK も TPNOTIME も指定されていませんでした。トランザクション・タイムアウトが発生すると、トランザクションがアボートされない限り、接続を使ったメッセージの送受信や新しい接続の開始はできません。これらの操作を行おうとすると、TPETIME が発生して失敗します。

[TPEBLOCK]
ブロッキング状態のため、TPNOBLOCK が指定されました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tpconnect() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目　tpalloc(3c)、tpdiscon(3c)、tprecv(3c)、tpsend(3c)、tpservice(3c)

tpconvert(3c)

名前

tpconvert()— 構造体の文字列表現とバイナリ表現の間の変換

形式

```
#include <atmi.h>
#include <xa.h>

int tpconvert(char *strrep, char *binrep, long flags)
```

機能説明

tpconvert() は、インターフェイス構造体の文字列表現 (*strrep*) とバイナリ表現 (*binrep*) の間で変換を行います。

変換の方向およびインターフェイス構造体のタイプは、どららも *flags* 引数で指定します。バイナリ表現から文字列表現へ変換する場合は、*flags* の TPTOSTRING ビットをセットし、文字列からバイナリへ変換する場合は、このビットをリセットします。変換する構造体のタイプを示すフラグは次のように定義されており、同時に 1 つのフラグのみを指定できます。

TPCONVCLTID
: CLIENTID を変換します (atmi.h を参照)。

TPCONVTRANID
: TPTRANID を変換します (atmi.h を参照)。

TPCONVXID
: XID を変換します (xa.h を参照)。

バイナリ表現から文字列表現に変換する場合、*strrep* は最低でも TPCONVMAXSTR 文字の長さがなければなりません。

TPTRANID と XID の値が異なる文字列バージョンをキー・フィールドとして許す TM_MIB(5) クラス (たとえば T_TRANSACTION および T_ULOG) にアクセスする場合は、システムはこれらの値を等しいものとして扱うことに注意してください。したがって、アプリケーション・プログラムでは、これらのデータ・タイプの文字列の値を作ったり操作したりすべきではありません。これらの値の 1 つがキー・フィールドとして使用された場合、TM_MIB(5) は文字列で識別されるグローバルトランザクションに一致するオブジェクトのみが返されることを保証します。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tpconvert() の呼び出しを発行できます。

戻り値

異常終了すると、tpconvert() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

次の条件が発生すると、tpconvert() は失敗し、tperrno() を次のように設定します。

[TPEINVAL]
無効な引き数が指定されました。*strrep* または *binrep* が NULL ポインタであるか、*flags* が構造体の 1 つのタイプを明確に示していません。

[TPEOS]
オペレーティング・システムのエラーが発生しました。失敗したシステム・コールを示す数値が Uunixerr に入っています。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容は userlog(3c) に書き込まれます。

移植性

このインターフェイスは、BEA Tuxedo ATMI リリース 5.0 またはそれ以降でしか利用できません。このインターフェイスは、ワークステーション・プラットフォームで利用できます。

関連項目

tpresume(3c)、tpservice(3c)、tpsuspend(3c)、tx_info(3c)、TM_MIB(5)

tpcryptpw(3c)

名前

tpcryptpw()— 管理要求内のアプリケーション・パスワードを暗号化

形式

```
#include <atmi.h>
#include <fml32.h>

int tpcryptpw(FBFR32 *buf)
```

機能説明

tpcryptpw() は、サービスにリクエストを送る前に管理要求バッファ内のアプリケーション・パスワードを暗号化するために使用します。アプリケーション・パスワードは、FML32 フィールド識別子 TA_PASSWORD を用いて文字列値として保存されます。この暗号化は、テキストレベルのパスワードが危険に晒されず、すべてのアクティブなアプリケーション・サイトに適当なパスワードの更新が伝播することを保証するために必要です。付加的なシステム・フィールドが呼び出し側のバッファに追加され、既存のフィールドが要求を満たすために変更されることがあります。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tpcryptpw() の呼び出しを発行できます。

戻り値

異常終了すると、tpcryptpw() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpcryptpw() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
: 無効な引数が指定されました。*buf* の値が NULL で、FML32 型付きバッファを指していないか、appdir が入力バッファもしくは環境から決定できませんでした。

[TPEPERM]
: 呼び出し側プロセスが、要求タスクの実行に必要な適当なパーミッションを持っていませんでした。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。失敗したシステム・コールを示す数値が Uunixerr に入っています。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容は userlog(3c) に書き込まれます。

移植性　このインターフェイスは、BEA Tuxedo ATMI リリース 5.0 またはそれ以降が稼動する UNIX system サイトでしか利用できません。このインターフェイスはワークステーション・クライアントでは利用できません。

ファイル　`${TUXDIR}/lib/libtmib.a, ${TUXDIR}/lib/libtmib.so.rel`

関連項目　`MIB(5)`、`TM_MIB(5)`

『BEA Tuxedo アプリケーションの設定』

『BEA Tuxedo アプリケーション実行時の管理』

tpdequeue(3c)

名前

tpdequeue() — キューからメッセージを取り出すルーチン

形式

```
#include <atmi.h>
int tpdequeue(char *qspace, char *qname, TPQCTL *ctl, char **data,
long *len, long flags)
```

機能説明

tpdequeue() は、キュー・スペース *qspace* 内の *qname* で指定されるキューから、処理するメッセージを取り出します。

省略時設定では、キューの先頭のメッセージが取り出されます。キュー上のメッセージのデフォルトの順序は、そのキューの作成時に定義されます。アプリケーションは、*ctl* パラメータでメッセージ識別子または相関識別子を指定することにより、特定のメッセージのキューからの取り出しを要求できます。メッセージが現在利用できなかった場合にはアプリケーションはメッセージを待つということを示すために、*ctl* フラグを使用することもできます。*ctl* パラメータを使って、メッセージをキューから削除したりキューでのメッセージの相対位置を変更せずにメッセージを見ることができます。この後のこのパラメータについての説明を参照してください。

data は、メッセージの読み込み先であるバッファを指すポインタのアドレスです。*len* は、そのメッセージの長さを指します。**data* は、事前に tpalloc() によって割り当てられたバッファを指さなければなりません。メッセージが tpdequeue に渡されるバッファよりも大きい場合、バッファはメッセージが入れる大きさまで拡大されます。メッセージ・バッファのサイズに変化があるかどうかを判別するには、tpdequeue() が発行される前の (合計の) サイズを **len* と比較します。**len* のほうが大きい場合、バッファは大きくなっています。そうでない場合は、バッファのサイズは変更されていません。なお、**data* は、バッファのサイズが大きくなったこと以外の理由でも変化する可能性があるので注意してください。終了時に **len* が 0 である場合は、取り出されたメッセージにはデータ部がなく、**data* も *data が指すバッファも変更されていません。**data* または *len* が NULL であるとエラーになります。

呼び出し元がトランザクション・モードにあり、TPNOTRAN フラグが設定されていない場合は、メッセージはトランザクション・モードでキューから取り出されます。この結果、tpdequeue() が正常終了して呼び出し元のトランザクションが正常にコミットされると、要求はキューから削除されます。呼び出し元のトランザクションが、明示的に、あるいはトランザクション・タイムアウトまたは何らかの通信エラーの結果としてロールバックされると、メッセージはキュー上に残されます (つまり、キューからのメッセージの削除もロールバックされます)。同じトランザクション内で、同じメッセージの登録と取り出しを行うことはできません。

呼び出し元がトランザクション・モードにないか、または TPNOTRAN フラグが設定されている場合は、メッセージはトランザクション・モードではキューから取り出されません。トランザクション・モードでないときに通信エラーまたはタイムアウトが発生した場合、アプリケーションにはメッセージが正しくキューから取り出されたかどうかがわからず、メッセージが失われることがあります。

次に、有効な *flags* の一覧を示します。

TPNOTRAN
: 呼び出し元がトランザクション・モードにあり、このフラグが設定されていると、メッセージは、呼び出し元と同じトランザクション内ではキューから取り出されません。このフラグを設定する、トランザクション・モードの呼び出し元は、メッセージをキューから取り出すときに、やはりトランザクション・タイムアウトの影響を受けます (それ以外はなし)。キューからのメッセージの取り出しに失敗した場合、呼び出し元のトランザクションは影響されません。

TPNOBLOCK
: ブロッキング条件が存在すると、メッセージはキューから取り出されません。このフラグが設定されていて、メッセージの転送先である内部バッファがいっぱいであるなどのブロッキング条件が存在する場合、呼び出しは異常終了し、tperrno() に TPEBLOCK が設定されます。このフラグが設定されていて、ターゲットのキューが別のアプリケーションによって排他的にオープンされているなどのブロッキング条件が存在する場合、呼び出しは異常終了して tperrno() に TPEDIAGNOSTIC が設定され、TPQCTL 構造体の診断フィールドは QMESHARE に設定されます。後者の場合、BEA Tuxedo ATMI システム以外の BEA 製品をベースとする別のアプリケーションが、キューイング・サービス API (QSAPI) を使って排他的な読み書きのためのキューをオープンします。

  TPNOBLOCK が指定されていないときにブロッキング条件が存在すると、呼び出し元は、その条件が解消されるか、またはタイムアウト (トランザクションまたはブロッキング) が発生するまではブロックされます。(TPQCTL 構造体の) *flags* に TPQWAIT オプションが指定されている場合は、このブロッキング条件には、キュー自体のブロッキングは含まれません。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。トランザクション・タイムアウトは依然として発生する可能性があります。

TPNOCHANGE
: このフラグが設定されていると、**data* が指すバッファのタイプは変更されません。デフォルトでは、**data* が指すバッファ型と異なるバッファ型を受信すると、**data* のバッファ型は受信したバッファ型に変更されます (受信プロセスがそのバッファ型を認識できる場合)。つまり、受信されたバッファのタイプおよびサブタイプは、**data* が指すバッファのタイプおよびサブタイプと一致しなければなりません。

TPSIGRSTRT
: このフラグが設定されていると、シグナルによって中断された基本となるシステム・コールが再発行されます。このフラグが設定されていないときにシグナルによってシステム・コールが中断されると、呼び出しは異常終了し、tperrno() は TPGOTSIG に設定されます。

tpdequeue() が正常終了すると、アプリケーションは、*ctl* データ構造体を使用してメッセージに関する追加情報を取得できます。この情報には、キューから取り出されたメッセージのメッセージ識別子、すべての応答または異常終了メッセージに付随して、発信元がメッセージと元の要求を結び付けることができるようにする相関識別子、メッセージが送られるサービスの品質、メッセージの応答が送られるサービスの品質、応答が要求された場合は応答キューの名前、およびメッセージをキューから取り出すときの失敗に関する情報をアプリケーションが登録できる異常終了キューの名前が含まれます。これらについては、次に説明します。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpdequeue() の呼び出しを発行できません。

制御パラメータ

TPQCTL 構造体は、アプリケーション・プログラムが、キューからのメッセージの取り出しに関連するパラメータを渡したり、取得したりするために使用します。TPQCTL の要素 *flags* は、この構造体の他のどの要素が有効であるかを示すために使用されます。

tpdequeue() への入力時には、次の要素を TPQCTL 構造体に設定できます。

```
long flags;             /* どの値が
                         * 設定されるかの指定 */
char msgid[32];         /* キューから取り出すメッセージの ID */
char corrid[32];        /* キューから取り出す
                         * メッセージの相関識別子 */
```

tpdequeue() の入力情報を制御する *flags* パラメータの有効なビットの一覧を次に示します。

TPNOFLAGS
: フラグは設定されません。制御構造体から情報は得られません。

TPQGETBYMSGID
: このフラグを設定すると、*ctl->msgid* によって指定されたメッセージ識別子を持つメッセージのキューからの取り出しを要求します。メッセージ識別子は、事前の tpenqueue(3c) の呼び出しによって取得できます。メッセージが別のキューに移動すると、メッセージ識別子が変わることに注意してください。また、メッセージ識別子の値は 32 バイトすべてが意味を持つため、*ctl->msgid* によって指定された値を完全に初期化する必要があります (たとえば、ヌル文字で埋めるなど)。

TPQGETBYCORRID
: このフラグを設定すると、*ctl->corrid* によって指定された相関識別子を持つメッセージのキューからの取り出しを要求します。相関識別子は、アプリケーションが tpenqueue() でキューにメッセージを登録したときに指定されます。また、相関識別子の値は 32 バイト全体が意味を持つため、*ctl->corrid* によって指定された値を完全に初期化する必要があります (たとえば、ヌル文字で埋めるなど)。

TPQWAIT
: このフラグを設定すると、キューが空の場合にはエラーは返されません。代わりに、メッセージを取り出せるようになるまで、プロセスは待機します。TPQGETBYMSGID または TPQGETBYCORRID とともに TPQWAIT が設定されている場合、指定されたメッセージ識別子または相関識別子を持つメッセージがキュー内に存在しない場合はエラーが返されません。代わりに、基準を満たすメッセージが利用可能になるまで、プロセスは待機しています。プロセスが呼び出し元のトランザクション・タイムアウトに従っている場合、またはトランザクション・モードでない場合、TMQUEUE プロセスに -t オプションで指定されたタイムアウトの影響を受けます。

  要求する基準を満たすメッセージがすぐには利用できず、設定されたアクションのリソースを使い尽くしてしまった場合には、tpdequeue は -1 を返し tperrno() を TPEDIAGNOSTIC に設定し、TPQCTL 構造体の診断フィールドを QMESYSTEM に設定します。

  TPQWAIT 制御パラメータを指定する tpdequeue() の各要求では、条件を満たすメッセージがすぐに利用できない場合、キュー・マネージャ (TMQUEUE) のアクション・オブジェクトを使用できる必要があります。アクション・オブジェクトが使用できない場合、tpdequeue() 要求は失敗します。使用可能なキュー・マネージャのアクションの数は、キュー・スペースが作成または修正されるときに指定されます。待機中のキューからの取り出し要求が終了すると、対応する関連アクション・オブジェクトが別の要求でも利用できるようになります。

TPQPEEK

このフラグが設定されていると、指定されたメッセージは読み取られますが、キューからは削除されません。このフラグは、tpdequeue() 操作に TPNOTRAN フラグが設定されていることを意味します。つまり、非破壊的なキューからの取り出しはトランザクションに含まれません。トランザクション終了前のトランザクション内で、キューに登録されたメッセージまたはキューから取り出されたメッセージを読み取ることはできません。

スレッドが TPQPEEK を使用してメッセージを非破壊的にキューから取り出している場合、システムが非破壊的なキューからの取り出し要求を処理している短い間、他のブロッキングされていない取り出し要求元からは、このメッセージが見えない場合があります。この中には、特定の選択基準 (メッセージ識別子や相関識別子など) を使って、現在非破壊的にキューから取り出されているメッセージを検出する取り出し要求元も含まれています。

tpdequeue() からの出力時には、次の要素が TPQCTL 構造体に設定されます。

```
long flags;             /* どの値が
                         * 設定されるかの指定 */

long priority;          /* 登録優先順位 */
char msgid[32];         /* キューから取り出されたメッセージの ID */
char corrid[32];        /* メッセージを識別するときに使用された相関識別子 */

long delivery_qos;      /* 配信サービスの品質 */
long reply_qos;         /* 応答メッセージのサービス品質 */
char replyqueue[16];    /* 応答のためのキュー名 */
char failurequeue[16]; /* 異常終了キューの名前 */
long diagnostic;        /* 異常終了の原因 */
long appkey;            /* アプリケーション認証用のクライアント
                         * キー */
long urcode;            /* ユーザ戻り値 */
CLIENTID cltid;         /* 発行元クライアント用のクライアント識別子 */
```

tpdequeue() からの出力情報を制御する *flags* パラメータの有効なビットの一覧を次に示します。これらのビットのいずれかについて、tpdequeue() の呼び出し時にフラグ・ビットをオンにしていると、その構造体の対応する要素には、メッセージがキューに登録されたときに指定された値が格納され、そのビットは設定されたままになります。値が使用できない (つまり、メッセージがキューに登録されたときに値が指定されなかった) 場合、または tpdequeue() を呼び出したときにビットが設定されていない場合は、tpdequeue() はフラグをオフにして終了します。

TPQPRIORITY

このフラグが設定されていて tpdequeue() の呼び出しが正常終了し、メッセージが明示的な優先順位でキューに登録された場合、この優先順位が *ctl->priority* に格納されます。優先順位は 1 以上 100 以内の範囲内で数値が高いほど優先順位も高くなります (高い数値のメッセージが低い数値のメッセージよりも先にキューから取り出される)。優先順位が決まっていないキューの場合、値は情報にすぎません。

メッセージがキューに登録されるときに優先順位が明示的に指定されずに tpdequeue() 呼び出しが正常に終了した場合、このメッセージの優先順位は 50 になります。

TPQMSGID

このフラグが設定されていて tpdequeue() の呼び出しが正常終了した場合、メッセージ識別子が *ctl->msgid* に格納されます。メッセージ識別子の値は 32 バイト全体が意味を持ちます。

TPQCORRID

このフラグが設定されていて tpdequeue() の呼び出しが正常終了し、メッセージが相関識別子によってキューに登録された場合、この相関識別子が *ctl->corrid* に格納されます。相関識別子の値は 32 バイト全体が意味を持ちます。BEA Tuxedo ATMI/Q が提供するメッセージの応答には、すべて元の要求メッセージの相関識別子が付いています。

TPQDELIVERYQOS

このフラグが設定されていて tpdequeue() の呼び出しが正常終了し、メッセージが配信サービスの品質を指定されてキューに登録された場合、TPQQOSDEFAULTPERSIST、TPQQOSPERSISTENT、TPQQOSNONPERSISTENT のいずれかのフラグが *ctl->delivery_qos* に格納されます。メッセージがキューに登録されたときに配信サービスの品質が明示的に指定されていない場合は、対象となるキューのデフォルトの配信方針によってメッセージ配信の品質が決まります。

TPQREPLYQOS

このフラグが設定されていて tpdequeue() の呼び出しが正常終了し、メッセージが応答サービス品質によってキューに登録された場合は、TPQQOSDEFAULTPERSIST、TPQQOSPERSISTENT、TPQQOSNONPERSISTENT のいずれかのフラグが *ctl->reply_qos* に格納されます。メッセージがキューに登録されたときに応答のサービス品質が明示的に指定されていない場合、*ctl-->replyqueue* キューのデフォルトの配信方針によって、すべての応答に対する配信サービスの品質が決まります。

デフォルトの配信方針は、メッセージの応答がキューに登録されるときに決定されます。つまり、デフォルトの応答キューの配信方針が、元のメッセージがキューに登録されてからメッセージの応答がキューに登録されるまでの

間に修正されると、応答が最終的にキューに登録されたときに有効だった方針が使用されます。

TPQREPLYQ
: このフラグが設定されていて tpdequeue() の呼び出しが正常終了し、メッセージが応答キューによってキューに登録された場合、応答キューの名前が *ctl->replyqueue* に格納されます。メッセージへの応答はすべて、要求メッセージと同じキュー・スペース内の、指定された応答キューに入る必要があります。

TPQFAILUREQ
: このフラグが設定されていて tpdequeue() の呼び出しが正常終了し、メッセージが異常終了キューによってキューに登録された場合、異常終了キューの名前が *ctl->failurequeue* に格納されます。すべての異常終了メッセージは、要求メッセージと同じキュー・スペース内の、指定された異常終了キューに入る必要があります。

flags パラメータの他のビット、TPQTOPTPQBEFOREMSGID、TPQTIME_ABS、TPQTIME_REL、TPQEXPTIME_ABS、TPQEXPTIME_REL、および TPQEXPTIME_NONE は、tpdequeue() が呼び出されるとクリア (ゼロに設定) されます。これらのビットは、tpenqueue() の入力情報を制御する *flags* パラメータの有効なビットです。

tpdequeue() の呼び出しが異常終了して tperrno() に TPEDIAGNOSTIC が設定された場合は、異常終了の原因を示す値が *ctl->diagnostic* に返されます。返される可能性のある値は、この後の「診断」の項で定義しています。

また出力時には、tpdequeue() 呼び出しが正常に終了すると、*ctl->appkey* にアプリケーション認証キーが設定され、*ctl->cltid* に要求の発信元であるクライアントの識別子が設定され、*ctl->urcode* にメッセージ登録時に設定されたユーザ戻り値が設定されます。

ctl パラメータが NULL である場合、入力フラグは TPNOFLAGS と見なされ、アプリケーション・プログラムは出力情報を利用できません。

戻り値

異常終了すると、tpdequeue() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpdequeue() は tperrno() を次のいずれかの値に設定します (特に指定がない限り、障害は、呼び出し元のトランザクションに影響しません)。

[TPEINVAL]
: 無効な引数が指定されました (たとえば、*qname* が NULL である場合、*data* が、tpalloc() によって割り当てられた空間を指していない場合、*flags* が無効である場合など)。

[TPENOENT]
qspace が利用できない (つまり、対応する TMQUEUE(5) サーバが利用できない) ため qspace にアクセスできないか、またはグローバル・トランザクション・テーブル (GTT) のエントリが不足しているためにグローバル・トランザクションが開始できません。

[TPEOTYPE]
応答のタイプおよびサブタイプのいずれかが、呼び出し元にとって未知のものです。または、TPNOCHANGE が *flags* に設定されているが、**data* のタイプおよびサブタイプが、サービスによって送信された応答のタイプおよびサブタイプと一致しません。いずれの場合も、**data*、その内容、および **len* はいずれも変更されません。トランザクション・モードで呼び出しが行われたときにこのエラーが発生すると、トランザクションは「アボートのみ」とマークされ、メッセージはキューに残ります。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、トランザクションはアボートされます。トランザクション・モードにない場合は、ブロッキング・タイムアウトが発生しており、TPNOBLOCK も TPNOTIME も指定されていませんでした。トランザクション・タイムアウトが発生すると、トランザクションがアボートされるまで、キューから新しいメッセージを取り出そうとしても常に TPETIME を示して異常終了します。

[TPEBLOCK]
ブロッキング状態のため、TPNOBLOCK が指定されました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tpdequeue() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。キューには影響ありません。

[TPEOS]
オペレーティング・システムのエラーが発生しました。キューには影響ありません。

[TPEDIAGNOSTIC]
指定されたキューからのメッセージの取り出しが異常終了しました。異常終了の原因は、*ctl* 構造体を介して返される診断値によって判別できます

診断

次の診断値は、キューからのメッセージの取り出し中に返されます。

[QMEINVAL]
無効なフラグ値が指定されました

[QMEBADRMID]
無効なリソース・マネージャ識別子が指定されました。

[QMENOTOPEN]
このリソース・マネージャは現在オープンしていません。

[QMETRAN]
トランザクション・モードで呼び出しが行われなかったため、または TPNOTRAN フラグを設定して呼び出したために、メッセージをキューから取り出すトランザクション開始のエラーが発生しました。この診断は、BEA Tuxedo リリース 7.1 以降のキュー・マネージャからは返されません。

[QMEBADMSGID]
キューからの取り出し用に、無効なメッセージ識別子が指定されました。

[QMESYSTEM]
システム・エラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[QMEOS]
オペレーティング・システムのエラーが発生しました。

[QMEABORTED]
操作は中断されました。グローバル・トランザクション内で実行されている場合、グローバル・トランザクションがロールバックのみとしてマークされています。それ以外の場合、キュ・ーマネージャは操作を中断しました。

[QMEPROTO]
トランザクション状態がアクティブでないときに、キューからの取り出しが行われました。

[QMEBADQUEUE]
無効または削除されたキューの名前が指定されました。

[QMENOMSG]
キューから取り出せるメッセージはありません。なお、メッセージがキュー上に存在し、別のアプリケーション・プロセスが、このメッセージをキューから読み取っていた可能性があることに注意してください。この場合は、その別のプロセスがトランザクションをロールバックしたときにメッセージはキューに戻されます。

[QMEINUSE]
: 相関識別子またはメッセージ識別子を使用してメッセージをキューから取り出す場合、ほかのトランザクションが指定したメッセージを使用しています。それ以外の場合、現在キューにあるメッセージはすべてほかのトランザクションによって使用中です。この診断は、BEA Tuxedo リリース 7.1 以降のキュー・マネージャからは返されません。

[QMESHARE]
: 指定されたキューからメッセージを取り出そうとしましたが、別のアプリケーションによってそのキューがオープンされています。別のアプリケーションとは、BEA Tuxedo システム以外の BEA 製品をベースとするアプリケーションであり、キューイング・サービス API (QSAPI) を使って排他的な読み書きのためのキューをオープンしています。

関連項目 `qmadmin(1)`、`tpalloc(3c)`、`tpenqueue(3c)`、`APPQ_MIB(5)`、`TMQUEUE(5)`

tpdiscon(3c)

名前

tpdiscon()— 会話型サービスの接続を切断するルーチン

形式

```
#include <atmi.h>
int tpdiscon(int cd)
```

機能説明

tpdiscon() は、*cd* で指定された接続をただちに切断し、接続の他方の側で TPEV_DISCONIMM イベントを生成します。

tpdiscon() は、会話の起動側からしか呼び出せません。tpdiscon() は、呼び出しに使用された記述子に対応する会話型サービス内からは呼び出せません。会話サービスは tpreturn() を使用して、会話の該当部分が完了したことを通知しなければなりません。同様に、会話型サービスとのやりとりを行うプログラムが tpdiscon() を呼び出す場合でも、そのサービスに tpreturn() で接続を切断するようにしてください。これによって、正しい結果が得られます。

tpdiscon() を使用すると、接続はただちに切断されます (すなわち、正常終了ではなく、アボート)。したがって、あて先に届いていないデータは失われます。tpdiscon() は、接続の他方の側のプログラムが呼び出し元のトランザクションに参加している場合でも発行されます。この場合、このトランザクションはアボートします。また、呼び出し元は、tpdiscon() が呼び出されるときにその接続の制御権をもっている必要はありません。

戻り値

異常終了すると、tpdiscon() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpdiscon() は tperrno() を次のいずれかの値に設定します。

[TPEBADDESC]
cd が無効であるか、会話型サービスに使用された記述子です。

[TPETIME]
タイムアウトが発生しました。記述子は無効になります。

[TPEPROTO]
tpdiscon() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。記述子は無効になります。

[TPEOS]
オペレーティング・システムのエラーが発生しました。記述子は無効になります。

関連項目　　tpabort(3c)、tpcommit(3c)、tpconnect(3c)、tprecv(3c)、tpreturn(3c)、tpsend(3c)

tpenqueue(3c)

名前　tpenqueue()— メッセージをキューに登録するルーチン

形式

```
#include <atmi.h>
int tpenqueue(char *qspace, char *qname, TPQCTL *ctl, char *data,
long len, long flags)
```

機能説明

tpenqueue() は、キュー・スペース *qspace* 内の *qname* で指定されるキューに、メッセージを格納します。キュー・スペースは、キューを集めたもので、そのうちの１つのキューが *qname* でなければなりません。

メッセージが BEA Tuxedo ATMI システムのサーバを対象としている場合、*qname* は、サーバによって提供されるサービスの名前に一致します。システムが提供するサーバである TMQFORWARD(5) は、メッセージをキューから取り出し、キューと同じ名前のサービスを提供するサーバに、そのメッセージを転送するデフォルトの機構となります。発信元が応答を期待していた場合は、転送されたサービス要求への応答は、特に指定されている場合を除き、発信元のキューに格納されます。発信元は、続いて応答メッセージをキューから取り出します。キューは、任意の２つの BEA Tuxedo ATMI システムのプロセス間 (クライアントやサーバ) における信頼性の高いメッセージ転送機構用としても使用できます。この場合、キューの名前は、サービス名ではなく、メッセージ転送について承認されている何らかの資格と一致します。

data が NULL 以外である場合、これは事前に tpalloc() によって割り当てられたバッファを指さなければならず、*len* は、キューに登録されるバッファ内のデータの大きさを指定しなければなりません。長さを指定する必要のないタイプのバッファ (FML フィールド化バッファなど) を *data* が指す場合、*len* は無視されます。*data* が NULL の場合、*len* は無視され、データ部分なしでメッセージはキューに登録されます

メッセージは、*qspace* について定義された優先順位が事前の tpsprio() の呼び出しによって無効化されていないかぎり、この優先順位でキューに登録されます。

呼び出し元がトランザクションにあり、TPNOTRAN フラグが設定されていない場合は、メッセージは、トランザクション・モードでキューに登録されます。この結果、tpenqueue() が正常終了して呼び出し元のトランザクションが正常にコミットされると、メッセージは、トランザクションの完了後に処理されることが保証されます。呼び出し元のトランザクションが、明示的に、あるいはトランザクション・タイムアウトまたは何らかの通信エラーの結果としてロールバックされると、メッセージは、キューから削除されます (つまり、キューへのメッセージの登録もロールバックされます)。同じトランザクション内で同じメッセージの登録と取り出しを行うことはできません。

呼び出し元がトランザクション・モードにないか、または TPNOTRAN フラグが設定されている場合は、メッセージはトランザクション・モードではキューに登録されません。tpenqueue() が正常終了すれば、出されたメッセージが処理されることが保証されます。トランザクション・モードでないときに通信エラーまたはタイムアウトが発生した場合、アプリケーションにはメッセージがキューに正しく格納されたかどうかがわかりません。

メッセージが処理される順序は、この後説明するように、*ctl* データ構造体を介してアプリケーションによって制御されます。デフォルトのキューの順序は、キューの作成時に設定されます。

次に、有効な *flags* の一覧を示します。

TPNOTRAN
: 呼び出し元がトランザクション・モードにあり、このフラグが設定されていると、メッセージは呼び出し元と同じトランザクション内ではキューに登録されません。このフラグを設定する、トランザクション・モードの呼び出し元は、メッセージをキューに登録するときに、やはりトランザクション・タイムアウトの影響を受けます (それ以外はなし)。キューへのメッセージの登録が失敗した場合、呼び出し元のトランザクションは影響されません。

TPNOBLOCK
: ブロッキング条件が存在すると、メッセージはキューに入りません。このフラグが設定されていて、メッセージの転送先である内部バッファがいっぱいであるなどのブロッキング条件が存在する場合には、呼び出しは異常終了し、tperrno() に TPEBLOCK が設定されます。このフラグが設定されていて、ターゲットのキューが別のアプリケーションによって排他的にオープンされているなどのブロッキング条件が存在する場合には、呼び出しは異常終了して tperrno() に TPEDIAGNOSTIC が設定され、TPQCTL 構造体の診断フィールドは QMESHARE に設定されます。後者の場合、BEA Tuxedo ATMI システム以外の BEA 製品をベースとする別のアプリケーションが、キューイング・サービス API (QSAPI) を使って排他的な読み書きのためのキューをオープンしています。

  TPNOBLOCK が指定されていないときにブロッキング条件が存在すると、呼び出し元は、その条件が解消されるか、またはタイムアウト (トランザクションまたはブロッキング) が発生するまではブロックされます。タイムアウトが発生すると、呼び出しは異常終了し、tperrno() は TPETIME に設定されます。

TPNOTIME
: このフラグを設定すると、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象になりません。トランザクション・タイムアウトは依然として発生する可能性があります。

TPSIGRSTRT
: このフラグが設定されていて、基本となるシステム・コールがシグナルによって中断された場合、中断されたシステム・コールは再発行されます。TPSIGRSTRT が指定されていなくて、システム・コールがシグナルによって中断された場合、tpenqueue() は異常終了し、tperrno() に TPGOTSIG が設定されます。

キューへのメッセージ登録に関する追加情報は、*ctl* データ構造体を介して指定できます。この情報には、デフォルトのキューの順序を無効化してキューの先頭または登録されているメッセージの前にメッセージを登録するための値、キューからメッセージを取り出すまでの絶対時間または相対時間、メッセージが期限切れになり、キューから削除される絶対時間または相対時間、メッセージ配信サービスの品質、メッセージが応答する際のサービス品質、メッセージとそのメッセージに関連付けられた応答または異常終了メッセージを結び付けるときに役立つ相関識別子、応答を登録するキューの名前、およびすべての異常終了メッセージを登録するキューの名前が含まれます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpenqueue() を呼び出すことはできません。

制御パラメータ

TPQCTL 構造体は、アプリケーション・プログラムが、キューへのメッセージの登録に関連するパラメータを渡したり、取得したりする際に使用されます。TPQCTL の要素 *flags* は、この構造体の他のどの要素が有効であるかを示すために使用されます。

tpenqueue() への入力時には、次の要素を TPQCTL 構造体に設定できます。

```
long flags;             /* どの値が設定されるかの指定 */

long deq_time;          /* キューから取り出すときの絶対時間 / 相対時間 */
long priority;          /* 登録優先順位 */
long exp_time           /* 有効期限 */
long delivery_qos       /* 配信サービスの品質 */
long reply_qos          /* 応答サービスの品質 */
long urcode;            /* ユーザ戻り値 */
char msgid[32];         /* 既存メッセージの ID ( 要求をそのメッセージの前に
                         * 登録するため ) */
char corrid[32];        /* msg を識別するときに使用される相関識別子 */

char replyqueue[16];    /* 応答メッセージ用キューの名前 */
char failurequeue[16]; /* 異常終了メッセージ用キューの名前 */
```

tpenqueue() の入力情報を制御する *flags* パラメータの有効なビットの一覧を次に示します。

TPNOFLAGS
: フラグおよび値は設定されません。制御構造体から情報は得られません。

TPQTOP

このフラグを設定すると、キューの順序は無効化され、メッセージはキューの先頭に位置付けられます。この要求は、キューが順序を無効化できるように構成されているかどうかに応じて、受け入れられない場合があります。TPQTOP および TPQBEFOREMSGID は、相互に排他的なフラグです。

TPQBEFOREMSGID

このフラグを設定すると、キューの順序はオーバーライドされ、メッセージは *ctl->msgid* によって識別されるメッセージの前に登録されます。この要求は、キューが順序を無効化できるように構成されているかどうかに応じて、受け入れられない場合があります。TPQTOP および TPQBEFOREMSGID は、相互に排他的なフラグです。また、メッセージ識別子の値は 32 バイト全体が意味を持つため、*ctl->msgid* によって指定された値を完全に初期化する必要があります (たとえば、ヌル文字で埋めるなど)。

TPQTIME_ABS

このフラグが設定されると、メッセージは *ctl->deq_time* によって指定された時間の後で使用可能になります。*deq_time* は、time(2)、mktime(3c)、または gp_mktime(3c) によって生成された絶対時間値です (UTC (協定世界時) 1970 年 1 月 1 日 00:00:00 から経過した秒数)。TPQTIME_ABS および TPQTIME_REL は、相互に排他的なフラグです。絶対時間は、キュー管理プロセスが存在するマシン上のクロックによって決まります。

TPQTIME_REL

このフラグを設定すると、メッセージは、キューへの登録操作が完了してからの相対時間経過後に使用可能になります。*ctl->deq_time* は、キューの登録が完了した後、出された要求が処理されるまでの遅延秒数を指定します。TPQTIME_ABS および TPQTIME_REL は、相互に排他的なフラグです。

TPQPRIORITY

このフラグを設定すると、メッセージがキュー・スペースに登録される際の優先順位が、*ctl->priority* に格納されます。優先順位は、1 以上 100 以下の範囲内でなければなりません。数値は高いほど優先順位も高くなります (高い数値のメッセージが低い数値のメッセージよりも先にキューに登録される)。優先順位が指定されていないキューについては、この値は情報にすぎません。

このフラグを設定しなかった場合、メッセージの優先順位はデフォルトの 50 になります。

TPQCORRID

このフラグが設定されると、*ctl->corrid* で指定された相関識別子の値は、要求が tpdequeue() によってキューから取り出されるときに使用できます。この識別子は、キューに登録されたすべての応答または異常終了メッセージに付随するので、アプリケーションは、応答を特定の要求と結び付けることができます。相関識別子の値は 32 バイト全体が意味を持つため、*ctl->corrid* によって指定された値は完全に初期化する必要があります (たとえば、ヌル文字で埋めるなど)。

TPQREPLYQ

このフラグが設定されると、*ctl->replyqueue* で指定された応答キューは、待機メッセージに関連付けられます。メッセージに対する応答はすべて、要求メッセージと同じキュー・スペース内の、指定されたキューに登録されます。この文字列は、NULL で終了しなければなりません (最大 15 文字)。

TPQFAILUREQ

このフラグが設定されると、*ctl->failurequeue* で指定された異常終了キューは、待機メッセージに関連付けられます。(1) キューに登録されたメッセージが TMQFORWARD() によって処理され、(2) TMQFORWARD が -d オプションで開始され、さらに (3) サービスが異常終了してヌル以外の応答を返す場合は、その応答と 関連する tpurcode によって構成される異常終了メッセージが、元の要求メッセージと同じキュー・スペース内で指定されたキューに登録されます。この文字列は、NULL で終了しなければなりません (最大 15 文字)。

TPQDELIVERYQOS, TPQREPLYQOS

TPQDELIVERYQOS フラグを設定した場合、*ctl->delivery_qos* によって指定されたフラグがメッセージの配信サービスの品質を制御します。この場合、3 つの相互に排他的なフラグ、TPQQOSDEFAULTPERSIST、TPQQOSPERSISTENT、または TPQQOSNONPERSISTENT のいずれかが *ctl->delivery_qos* に設定されている必要があります。TPQDELIVERYQOS が設定されていない場合、ターゲット・キューのデフォルトの配信方針によってメッセージの配信サービスの品質が決まります。

TPQREPLYQOS フラグが設定されている場合、*ctl->reply_qos* で指定されたフラグによってメッセージ応答のサービス品質が制御されます。この場合、3 つの相互に排他的なフラグ、TPQQOSDEFAULTPERSIST、TPQQOSPERSISTENT、または TPQQOSNONPERSISTENT のいずれかが *ctl->reply_qos* によって設定されている必要があります。TPQREPLYQOS フラグが使用されるのは、TMQFORWARD によって処理されたメッセージから応答が返る場合です。TMQFORWARD を使用してサービスを呼び出していないアプリケーションは、独自の応答機構のヒントとして TPQREPLYQOS フラグを使用することができます。

TPQREPLYQOS が設定されていない場合、*ctl->replyqueue* キューのデフォルトの配信方針によって、すべての応答の配信サービスの品質が決まります。デフォルトの配信方針は、メッセージに対する応答がキューに登録される際に決まります。つまり、デフォルトの応答キューの配信方針が、元のメッセージがキューに登録されてからメッセージの応答がキューに登録されるまでの間に修正されると、応答が最終的にキューに登録されたときに有効だった方針が使用されます。

次に *ctl->delivery_qos* と *ctl->reply_qos* の有効な flags の値を示します。

TPQQOSDEFAULTPERSIST
: このフラグは、ターゲット・キューで指定されるデフォルトの配信方針を使ってメッセージが渡されることを指定します。

TPQQOSPERSISTENT
: このフラグは、ディスク・ベースの配信方式によって永続的な方法でメッセージが渡されることを指示します。このフラグの設定は、ターゲット・キューに指定されているデフォルトの配信方針よりも優先されます。

TPQQOSNONPERSISTENT
: このフラグは、メモリ・ベースの配信方式によってメッセージが非永続的な方法で渡されることを指示します。つまり、メッセージは、キューから取り出されるまでメモリ内のキューに登録されます。このフラグの設定は、ターゲット・キューに指定されているデフォルトの配信方針よりも優先されます。呼び出し元がトランザクションに関与している場合、呼び出し元のトランザクション内では一時的なメッセージがキューに登録されますが、システムがシャットダウンしたりクラッシュした場合、またはキュー・スペースの IPC 共有メモリが削除された場合、一時的メッセージは失われます。

TPQEXPTIME_ABS
: このフラグが設定されると、メッセージに絶対有効期限が指定されます。つまりこの時間になると、メッセージはキューから削除されます。
絶対有効期限は、キュー管理プロセスが存在するマシンのクロックによって決まります。

  絶対有効期限は、*ctl->exp_time* に格納される値によって示されます。*ctl->exp_time* の値は、time(2)、mktime(3C)、または gp_mktime(3c) によって生成された絶対時間値に設定する必要があります (UTC (協定世界時) 1970 年 1 月 1 日 00:00:00 から経過した秒数)。

  キューに登録した時間よりも早い絶対時間が指定された場合、操作は成功しますが、メッセージはしきい値の計算にカウントされません。有効期限がメッセージが使用できる時間より前に設定された場合、有効期限前に利用可能時間がくるように利用可能時間または有効期限を変更しない限り、メッセージを

キューから取り出すことができません。さらに、これらのメッセージは絶対にキューから取り出されないにもかかわらず、有効期限になるとキューから削除されます。トランザクション内でメッセージの期限が切れた場合、それによってトランザクションが異常終了することはありません。トランザクション内でキューへの登録中、またはキューからの取り出し中に期限が切れたメッセージは、トランザクションが終了するとキューから削除されます。メッセージの期限切れは通知されません。

TPQEXPTIME_ABS、TPQEXPTIME_REL、および TPQEXPTIME_NONE は、相互に排他的なフラグです。どのフラグも設定されていない場合、ターゲット・キューに関連付けられているデフォルトの有効期限がメッセージに適用されます。

TPQEXPTIME_REL

このフラグが設定されると、メッセージに相対有効期限が指定されます。つまり、メッセージがキューに到着した後の時間 (秒数) で、この時間が経過するとメッセージはキューから削除されます。相対有効期限は、*ctl->exp_time* に格納されている値によって示されます。

この有効期限がメッセージが使用できるようになる時間より前に設定された場合、利用可能時間が有効期限より前にくるように利用可能時間または有効期限を変更しない限り、メッセージをキューから取り出すことはできません。さらに、これらのメッセージはキューから取り出せない場合でも、有効時間になるとキューから削除されます。トランザクション中にメッセージの期限が切れても、トランザクションが異常終了することはありません。トランザクション内で、キューへの登録、またはキューからの取り出し中に期限の切れたメッセージは、トランザクションが終了するとキューから削除されます。メッセージの期限切れの通知はありません。

TPQEXPTIME_ABS、TPQEXPTIME_REL、および TPQEXPTIME_NONE は、相互に排他的なフラグです。どのフラグも設定されていない場合、ターゲット・キューに関連付けられているデフォルトの有効期限がメッセージに適用されます。

TPQEXPTIME_NONE

このフラグが設定されると、メッセージの有効期限がないことを示します。このフラグは、ターゲット・キューに関連付けられたどのデフォルトの有効期限方針よりも優先されます。メッセージを削除する場合は、キューから取り出すか、管理インターフェイスを介して削除します。

TPQEXPTIME_ABS、TPQEXPTIME_REL、および TPQEXPTIME_NONE は、相互に排他的なフラグです。どのフラグも設定されていない場合、ターゲット・キューに関連付けられているデフォルトの有効期限がメッセージに適用されます。

また、TPQCTL の要素 *urcode* にユーザ戻り値を設定することができます。この値は、メッセージをキューから取り出すアプリケーションに返されます。

tpenqueue() からの出力時には、次の要素が TPQCTL 構造体に設定されます。

```
long flags;                     /* どの値が設定されるかの指定 */

char msgid[32];                 /* id of enqueued message */
long diagnostic;                /* 異常終了の原因 */
```

tpenqueue() からの出力情報を制御する *flags* パラメータの有効なビットの一覧を次に示します。tpenqueue() の呼び出し時にこのフラグ・ビットをオンにしていると、/Q サーバ TMQUEUE(5) によって構造体の関連要素にメッセージ識別子が設定されます。tpenqueue() の呼び出し時にこのフラグ・ビットをオフにしていると、TMQUEUE() によって構造体の関連要素にメッセージ識別子は設定されません。

TPQMSGID
: このフラグが設定され、tpenqueue() の呼び出しが正常終了した場合は、メッセージ識別子が *ctl->msgid* に格納されます。メッセージ識別子値の値は 32 バイト全体が意味を持つため、*ctl->msgid* に格納される値は完全に初期化されます (たとえば、ヌル文字で埋めるなど)。初期化に使用する実際の埋め字は、BEA Tuxedo ATMI/Q コンポーネントのリリースによって異なります。

制御構造体の残りのメンバーは、tpenqueue() への入力に使用されません。

tpenqueue() の呼び出しが異常終了し、tperrno() に TPEDIAGNOSTIC が設定された場合は、異常終了の原因を示す値が *ctl->diagnostic* に返されます。返される可能性のある値は、この後の「診断」の項で定義しています。

このパラメータが NULL である場合、入力フラグは、TPNOFLAGS と見なされ、アプリケーション・プログラムは出力情報を利用できません。

戻り値

異常終了すると、tpenqueue() は -1 を返し、tperrno() を設定してエラー条件を示します。エラーでないときは、メッセージは、tpenqueue() の終了時に正しくキューに登録されます。

エラー

異常終了時には、tpenqueue() は tperrno() を次のいずれかの値に設定します (特に指定がない限り、障害は、呼び出し元のトランザクションに影響しません)。

[TPEINVAL]
: 無効な引数が指定されました (たとえば、*qspace* が NULL である場合、*data* が、tpalloc() によって割り当てられた空間を指していない場合、*flags* が無効である場合など)。

[TPENOENT]
: 利用できない (対応する TMQUEUE(5) サーバが利用できない) か、またはグローバル・トランザクション・テーブル (GTT) にエントリがないためグローバル・トランザクションを開始できず、*qspace* にアクセスできません。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、トランザクションはアボートされます。トランザクション・モードにない場合は、ブロッキング・タイムアウトが発生しており、TPNOBLOCK も TPNOTIME も指定されていませんでした。トランザクション・タイムアウトが発生すると、トランザクションがアボートされるまで、キューに新しいメッセージを登録しようとしても常に TPETIME を示して異常終了します。

[TPEBLOCK]
ブロッキング状態のため、TPNOBLOCK が指定されました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tpenqueue() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

[TPEDIAGNOSTIC]
指定されたキューへのメッセージの登録が異常終了しました。異常終了の原因は、*ctl* を介して返される診断値によって判別できます。

診断

次の診断値は、キューへのメッセージの登録中に返されます。

[QMEINVAL]
無効なフラグ値が指定されました

[QMEBADRMID]
無効なリソース・マネージャ識別子が指定されました。

[QMENOTOPEN]
このリソース・マネージャは現在オープンしていません。

[QMETRAN]
呼び出しがトランザクション・モードで行われていないか、または TPNOTRAN フラグを設定して呼び出したときに、メッセージをキューに登録するトランザクションを開始してエラーが発生しました。この診断は、BEA Tuxedo リリース 7.1 以降のキュー・マネージャからは返されません。

[QMEBADMSGID]
無効なメッセージ識別子が指定されました。

[QMESYSTEM]
システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[QMEOS]
オペレーティング・システムのエラーが発生しました。

[QMEABORTED]
操作は中断されました。グローバル・トランザクション内で実行されている場合、グローバル・トランザクションがロールバックのみとしてマークされています。それ以外の場合、キュー・マネージャは操作を中断しました。

[QMEPROTO]
トランザクション状態がアクティブでないときに、キューへの登録が行われました。

[QMEBADQUEUE]
無効または削除されたキューの名前が指定されました。

[QMENOSPACE]
キューのスペースがないなど、リソースが不十分なために、必要なサービス品質のメッセージ (永続ストレージまたは非永続ストレージ) がキューに登録されませんでした。次のいずれかの設定リソースを超えた場合、QMENOSPACE が返されます。(1) キュー・スペースに割り当てられたディスク (永続的) 領域、(2) キュー・スペースに割り当てられたメモリ (非永続的) 領域、(3) キュー・スペースに許容されている同時にアクティブにできるトランザクションの最大数、(4) キュー・スペースが 1 度に保持できる最大メッセージ数、(5) キューイング・サービス・コンポーネントが同時に処理できる最大アクション数、または (6) キューイング・サービス・コンポーネントを同時に使用できる認証ユーザの最大数。

[QMERELEASE]
新しい機能をサポートしないバージョンの BEA Tuxedo システムのキュー・マネージャに、メッセージを登録しようとしました。

[QMESHARE]
指定されたキューにメッセージを登録しようとしましたが、別のアプリケーションによってそのキュー・が排他的にオープンされています。別のアプリケーションは、BEA Tuxedo システム以外の BEA 製品に基づくアプリケーションであり、キューイング・サービス API (QSAPI) を使って排他的に読み書きするためにキューをオープンしています。

関連項目　qmadmin(1)、gp_mktime(3c)、tpacall(3c)、tpalloc(3c)、tpdequeue(3c)、tpinit(3c)、tpsprio(3c)、APPQ_MIB(5)、TMQFORWARD(5)、TMQUEUE(5)

tpenvelope(3c)

名前

tpenvelope()— 型付きメッセージ・バッファに関連付けられているデジタル署名と暗号化情報へのアクセス

形式

```
#include <atmi.h>
int tpenvelope(char *data, long len, int occurrence, TPKEY
*outputkey, long *status, char *timestamp, long flags)
```

機能説明

tpenvelope() は、型付きメッセージ・バッファに関連付けられている以下のデジタル署名と暗号化情報へのアクセスを提供します。

- デジタル署名登録要求

  送信プロセスがメッセージ・バッファのデジタル署名要求を登録する場合は、tpsign() を呼び出して明示的に登録するか、TPKEY_AUTOSIGN フラグを指定した tpkey_open() を呼び出して暗黙的に登録します。

- デジタル署名

  メッセージ・バッファが送信される直前に、各暗号化登録要求に対して、公開鍵ソフトウェアがデジタル署名を生成し、メッセージ・バッファにアタッチします。デジタル署名によって、受信プロセスはメッセージの署名者 (発信元) を確認することができます。

- 暗号化登録要求

  送信プロセスがメッセージ・バッファの暗号化 (封印) 要求を登録する場合、tpseal() を呼び出して明示的に登録するか、TPKEY_AUTOENCRYPT フラグを指定した tpkey_open() を呼び出して暗黙的に登録します。

- 暗号化エンベロープ

  公開鍵ソフトウェアは、メッセージ・バッファが送信される直前に、各暗号化登録要求に対して、メッセージの内容を暗号化し、暗号化エンベロープをメッセージ・バッファに添付します。暗号化エンベロープにより、受信プロセスはメッセージを解読することができます。

署名および暗号化の情報は、送信プロセスと受信プロセスのどちらからも利用できます。送信プロセスでは、デジタル署名と暗号化の情報は普通は保留状態にあり、メッセージが送信されるまで待機しています。受信プロセスでは、デジタル署名が確認され、暗号化と解読も既に行われています。解読または署名の確認に失敗した場合、メッセージの配信が行われない可能性があります。この場合、受信プロセスはメッセージ・バッファを受け取ることができないため、メッセージ・バッファの情報が伝わりません。

data は、(1) 以前 tpalloc( ) を呼び出すプロセスによって割り当てられたメッセージ・バッファ、または (2) システムによって受信プロセスに渡されたメッセージ・バッファのうち、いずれかの有効な型付きメッセージ・バッファを指している必要があります。メッセージ・バッファが自己記述型の場合、*len* は無視されます (0 でかまいません)。それ以外の場合、*len* には *data* 内のデータの長さが格納されている必要があります。

デジタル署名登録要求、デジタル署名、暗号化登録要求、およびメッセージ・バッファに関連付けられている暗号化エンベロープの複数のオカレンスが同時に存在することがあります。これらのオカレンスは順番に格納され、最初の項目が 0 位置に、以降の項目は 0 に続く連続する位置に格納されます。*occurrence* 入力パラメータは、要求された項目を示します。*occurrence* の値が最後の項目の位置を過ぎると、tpenvelope( ) は TPENOENT エラー状態で異常終了します。TPENOENT が返されるまで、tpenvelope( ) を繰り返し呼び出してすべての項目を調べることができます。

デジタル署名登録要求、暗号化登録要求、または暗号化エンベロープに関連付けられたキーのハンドルは、*outputkey* を介して返されます。返されたキー・ハンドルは、tpkey_open( ) を呼び出してオープンされた元のキーの個別コピーです。PRINCIPAL 属性パラメータなどのキーのプロパティは、tpkey_getinfo( ) を呼び出して取得します。キー・ハンドル *outputkey* は、呼び出し元が tpkey_close( ) を呼び出して解放します。

注記 *outputkey* がヌルの場合、キー・ハンドルは返されません。

status 出力パラメータは、デジタル署名登録要求、デジタル署名、暗号化登録要求、または暗号化エンベロープの状態を報告します。ステータスの値がヌルでない場合、次のいずれかの状態に設定されます。

TPSIGN_PENDING
: 対応する秘密鍵に関連付けられている署名者プリンシパルの代わりにデジタル署名が要求され、このプロセスからメッセージ・バッファが伝送される際にデジタル署名が生成されます。

TPSIGN_OK
: デジタル署名が確認されました。

TPSIGN_TAMPERED_MESSAGE
: メッセージ・バッファの内容が変更されたため、デジタル署名が無効です。

TPSIGN_TAMPERED_CERT
: 署名者のデジタル証明書が変更されたため、デジタル署名が無効です。

TPSIGN_REVOKED_CERT
: 署名者のデジタル証明書が取り消されたため、デジタル署名が無効です。

TPSIGN_POSTDATED
: デジタル署名のタイムスタンプが極端に先であるため、デジタル署名が無効です。

TPSIGN_EXPIRED_CERT
: 署名者のデジタル証明書の期限が切れたため、デジタル署名が無効です。

TPSIGN_EXPIRED
: デジタル署名のタイムスタンプが古すぎるため、デジタル署名が無効です。

TPSIGN_UNKNOWN
: 署名者のデジタル証明書が未知の認証局 (CA) であるため、デジタル署名が無効です。

TPSEAL_PENDING
: 対応する公開鍵に関連付けられている受信者プリンシパルに対して暗号化 (封印) が要求されており、このプロセスからメッセージ・バッファが伝送されるときに暗号化されます。

TPSEAL_OK
: 暗号化エンベロープが有効です。

TPSEAL_TAMPERED_CERT
: 受信者のデジタル証明書が変更されているため、暗号化エンベロープが無効です。

TPSEAL_REVOKED_CERT
: 受信者のデジタル証明書が取り消されたため、暗号化エンベロープが無効です。

TPSEAL_EXPIRED_CERT
: 受信者のデジタル証明書の期限が切れたため、暗号化エンベロープが無効です。

TPSEAL_UNKNOWN
: 受信者のデジタル証明書を発行したのが未知の CA であるため、暗号化エンベロープが無効です。

timestamp 出力パラメータには、デジタル署名が生成されたマシンのローカル・クロックに従ったタイムスタンプが含まれています。この値の整合性は、関連付けられているデジタル署名によって保護されています。*timestamp* によって示されるメモリ位置は、YYYYMMDDHHMMSS 形式のヌルで終了する署名時間です。ここで YYYY は年、MM は月、DD は日、HH は時間、MM は分、SS は秒を表します。*timestamp* はヌルでもかまいませんが、この場合値は返されません。暗号化 (封印) にはタイムスタンプは含まれず、*timestamp* によって示されるメモリ位置は変わりません。

flags パラメータは、次のいずれかの値に設定できます。

- TPKEY_REMOVE—*occurrence* の位置にある項目が削除され、バッファとの関連がなくなります。この項目に関連する出力パラメータ *outputkey*、*status*、および *timestamp* は、項目が削除される前に取り込まれます。その後の項目は 1 つずつ繰り下がるため、*occurrence* の番号が不連続になることはありません。
- TPKEY_REMOVEALL—メッセージ・バッファに関連付けられているすべての項目が削除されます。出力パラメータ、*outputkey*、*status*、および *timestamp* は返されません。
- TPKEY_VERIFY—メッセージ・バッファに関連付けられているすべてのデジタル署名が再確認されます。再確認後に署名のステータスが変わる場合があります。たとえば、受信プロセスによってメッセージ・バッファが修正された場合、発信者の署名のステータスは TPSIGN_OK から TPSIGN_TAMPERED_MESSAGE に変わります。

戻り値

異常終了すると、この関数は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

[TPEINVAL]
無効な引数が指定されました。たとえば、*data* の値がヌルになっているか、または *flags* に割り当てられた値が認識されません。

[TPENOENT]
この *occurrence* は存在しません。

[TPESYSTEM]
エラーが発生しました。詳細については、システムのエラー・ログ・ファイルを参照してください。

関連項目

tpkey_close(3c)、tpkey_getinfo(3c)、tpkey_open(3c)、tpseal(3c)、tpsign(3c)

tperrordetail(3c)

名前

tperrordetail()— 最後の BEA Tuxedo ATMI システム呼び出しから生じるエラーに関する詳細の取得

形式

```
#include <atmi.h>
int tperrordetail(long flags)
```

機能説明

tperrordetail() は、カレント・スレッドで呼び出された最後の BEA Tuxedo ATMI ルーチンにより発生したエラーに関する追加の詳細情報を返します。tperrordetail() は、数値を返します。その数値は、シンボリック名でも表わされます。カレント・スレッドで呼び出された最後の BEA Tuxedo ATMI ルーチンによりエラーが発生していない場合、tperrordetail() は、ゼロを返します。従って、tperrordetail() はエラーが表示された後、つまり tperrno() が設定されたときに呼び出す必要があります。

flags は将来使用する予定であり、現在は必ずゼロを指定してください。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tperrordetail() の呼び出しを発行できます。

戻り値

異常終了すると、tperrordetail() は -1 を返し、tperrno() を設定してエラー条件を示します。

設定されるのは、tperrordetail() が返す各数値のシンボリック名および意味です。表示される順序は任意ではなく、優先順位を示すものではありません。

TPED_CLIENTDISCONNECTED
: Jolt クライアントは現在接続されていません。tpnotify() 呼び出しに TPACK フラグが使用され、tpnotify() のターゲットは現在 Jolt クライアントに接続していません。tpnotify() が異常終了したときは、中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、TPED_CLIENTDISCONNECTED が返されます。

TPED_DECRYPTION_FAILURE
: 暗号化されたメッセージを受信するプロセスが、メッセージを解読できません。このエラーが発生するのは、多くの場合、メッセージを解読するための秘密鍵にプロセスがアクセスできないためです。

  このエラーで呼び出しが異常終了した場合に、中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと TPED_DECRYPTION_FAILURE が返されます。

TPED_DOMAINUNREACHABLE

ドメインに到達できません。つまり、要求の作成時に、ローカル・ドメインではサービスできない要求を満たすように構成されたドメインに到達できませんでした。要求が異常終了した後で、中間 ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、TPED_DOMAINUNREACHABLE が返されます。

次の表は、ドメインに到達できないために tpcall()、tpgetrply()、または tprecv() 呼び出しが異常終了した場合に tperrno() が返す値を示します。引き続き tperrordetail() を呼び出すと、エラーの詳細として TPED_DOMAINUNREACHABLE が返されます。

ATMI **呼び出し**	tperrno	**エラーの詳細**
tpcall	TPESVCERR	TPED_DOMAINUNREACHABLE
tpgetrply	TPESVCERR	TPED_DOMAINUNREACHABLE
tprecv	TPEEVENT TPEV_SVCERR	TPED_DOMAINUNREACHABLE

TPED_DOMAINUNREACHABLE 機能は、BEA Tuxedo Domains にのみ適用されます。Connect OSI TP Domains や Connect SNA Domains など、他のドメインの製品には適用されません。

TPED_INVALID_CERTIFICATE

関連付けられたデジタル証明書が無効なため、デジタル署名付きのメッセージを受信するプロセスがデジタル署名を確認できません。このエラーが発生するのは、デジタル証明書の期限が切れている場合、デジタル証明書を発行しているのが未知の認証局 (CA) である場合、デジタル証明書が変更されている場合などです。

このエラーによって呼び出しが異常終了されたときに、中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、TPED_INVALID_CERTIFICATE が返されます。

TPED_INVALID_SIGNATURE

署名が無効なため、デジタル署名付きのメッセージを受信するプロセスがデジタル署名を確認できません。このエラーが発生するのは、メッセージが変更されている場合、デジタル署名のタイムスタンプが古すぎる場合、デジタル署名のタイムスタンプが極端に先のものである場合などです。

このエラーによって呼び出しが異常終了したときに、中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、TPED_INVALID_SIGNATURE が返されます。

TPED_INVALIDCONTEXT
別のスレッドがコンテキストを終了すると、スレッドが ATMI 呼び出し内でブロックされます。具体的には、別のスレッドがコンテキストを終了するときに ATMI 呼び出し内でブロックされたスレッドは、異常終了とともに ATMI 呼び出しから返され、tperrno() は TPESYSTEM に設定されます。中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、TPED_INVALIDCONTEXT が返されます。

TPED_INVALID_XA_TRANSACTION
トランザクションを開始しようとしましたが、このドメインでは、NO_XA フラグがオンになっています。

TPED_NOCLIENT
クライアントが存在しません。tpnotify() 呼び出しに TPACK フラグが指定されているのに、tpnotify() のターゲットがありません。tpnotify() が異常終了すると、tperrno() は TPENOENT に設定されます。中間の ATMI 呼び出しを省略して引き続き tperrordetail() 呼び出すと、TPED_NOCLIENT が返されます。

TPED_NOUNSOLHANDLER
クライアントに任意通知型ハンドラ・セットがありません。tpnotify() 呼び出しに TPACK フラグが指定され、tpnotify() のターゲットが BEA Tuxedo ATMI セッション内にあるのに、ターゲットに任意通知型の通知ハンドラが設定されていません。tpnotify() が異常終了すると、tperrno() は TPENOENT に設定されます。中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、TPED_NOUNSOLHANDLER が返されます。

TPED_SVCTIMEOUT
サーバは、サービスのタイムアウトにより終了しました。サービスのタイムアウトは、UBBCONFIG 中の SVCTIMEOUT 値によって制御されます。T_SERVER および T_SERVICE クラスの TA_SVCTIMEOUT は TM_MIB(5) に記載されています。このエラーよって呼び出しが異常終了したときは、中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、TPED_SVCTIMEOUT が返されます。

TPED_TERM
ワークステーション・クライアントが、アプリケーションに接続していません。このエラーによって呼び出しが異常終了したときは、中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、TPED_TERM が返されます。

エラー

異常終了時には、tperrordetail() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
flags はゼロに設定されていません。

関連項目

「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」、tpstrerrordetail(3c)、tperrno(5)

tpexport(3c)

名前

tpexport()— 型付きメッセージ・バッファを、デジタル署名と暗号化エンベロープを含むエクスポート可能なマシン独立型の文字列表現に変換

形式

```
#include <atmi.h>
int tpexport(char *ibuf, long ilen, char *ostr, long *olen,
long flags)
```

機能説明

tpexport() は、型付きメッセージ・バッファを外部化された表現に変換します。外部化された表現とは、通常は伝送直前にメッセージ・バッファに追加される BEA Tuxedo ATMI ヘッダ情報を含まない、メッセージ・バッファです。

外部化された表現は、任意の通信メカニズムを介して、プロセス、マシン、BEA Tuxedo ATMI アプリケーション間で伝送することができます。恒久的な記憶域にアーカイブでき、システムのシャットダウンおよび再起動後も有効です。

外部化される表現には、次のものがあります。

- *ibuf* に関連付けられているすべてのデジタル署名。これらの署名は、後で、バッファがインポートされたときに確認されます。
- *ibuf* に関連付けられているすべての暗号化エンベロープ。バッファの内容は、暗号化によって保護されます。解読のための有効な秘密鍵にアクセスできる指定された受信者だけが、後でバッファをインポートできます。

ibuf は、(1) 以前 tpalloc() を呼び出すプロセスによって割り当てられたメッセージ・バッファ、または (2) システムによって受信プロセスに渡されたメッセージ・バッファのうち、いずれかの有効な型付きメッセージ・バッファを指している必要があります。*ilen* は、エクスポートする *ibuf* の大きさを指定します。*ibuf* が長さの指定を必要としないタイプのバッファを指している場合 (たとえば、FML のフィールド化バッファ)、*ilen* の値は無視されます (0 でかまいません)。

ostr は、バッファの内容と関連プロパティの外部化された表現を保持する出力領域を指します。*flags* に TPEX_STRING が設定されている場合、外部化の形式は文字列型になります。それ以外の場合、出力する長さは **olen* によって指定され、ヌル・バイトを埋め込むことができます。

入力時には、**olen* は *ostr* で使用できる最大記憶サイズを指定します。出力時には、**olen* は *ostr* に書き込まれる実際のバイト数に設定されます。*flags* に TPEX_STRING が設定されている場合は、終端のヌル文字を含みます。

出力バッファに文字列形式 (base 64 エンコード) が要求される場合、*flags* 引き数は TPEX_STRING に設定します。それ以外の場合、出力はバイナリになります。

戻り値

異常終了すると、この関数は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

[TPEINVAL]
無効な引数が指定されました。たとえば、*ibuf* の値がヌル、または *flags* の値が正しく設定されていません。

[TPEPERM]
パーミッションに失敗しました。暗号サービスのプロバイダが、デジタル署名の生成に必要な秘密鍵にアクセスできませんでした。

[TPESYSTEM]
エラーが発生しました。詳細については、システム・エラーのログ・ファイルを参照してください。

[TPELIMIT]
十分な出力記憶域がありませんでした。**olen* を必要な容量に設定します。

関連項目

tpimport(3c)

tpforward(3c)

名前

tpforward()— サービス要求を他のサービス・ルーチンに転送するルーチン

形式

```
#include <atmi.h>
void tpforward(char *svc, char *data, long len, long flags)
```

機能説明

tpforward() を使用すると、サービス・ルーチンはクライアントの要求を別のサービス・ルーチンに転送して処理させることができます。tpforward() の働きは、サービス・ルーチンの中で最後に呼び出されるという点で tpreturn() と似ています。tpreturn() のように、tpforward() は、BEA Tuxedo ATMI システムのディスパッチャに制御を正常に返すことを保証するために、ディスパッチされるサービス・ルーチン内から呼び出されます。tpforward() は会話型サービス内から呼び出すことはできません。

この関数は、*data* が指すデータを使用して *svc* によって指定されるサービスに要求を転送します。サービス名の先頭に "." を付けてはいけません。要求を転送するサービス・ルーチンは応答を受け取りません。要求の転送が終わると、サービス・ルーチンは、通信マネージャ・ディスパッチャに戻り、他の作業を行える状態になります。なお、転送された要求からの応答は期待しないため、この要求は、その要求を転送するサービスと同じ実行形式内の任意のサービス・ルーチンに、エラーなして転送することができます。

サービス・ルーチンがトランザクション・モードであると、この関数はトランザクションの呼び出し元の部分を、そのトランザクションのイニシエータが tpcommit() または tpabort() のいずれかを出したときに完了できるような状態にします。トランザクションがサービス・ルーチンの実行中に tpbegin() により明示的に開始された場合、このトランザクションを tpcommit() または tpabort() で終了させてから、tpforward() を呼び出さなければなりません。このため、転送チェインに関与するすべてのサービスは、トランザクション・モードで起動するか、あるいはどれもトランザクション・モードでは起動しないようにします。

転送チェインの最後のサーバは、tpreturn() を使用して要求の発信元に応答を返します。要約して言えば、tpforward() は待機しているリクエスタに応答を返す役割を別のサーバに転送するわけです。

tpforward() は、サービス・ルーチンが出したサービス要求から期待されるすべての応答を受け取った後、呼び出すようにします。受信されていない未終了の応答は、受信後、通信マネージャ・ディスパッチャによって自動的に取り除かれます。さらに、以後、これらの応答の記述子は無効になり、要求は *svc* に転送されません。

data は送信される応答のデータ部を指すポインタです。*data* が NULL でない場合、tpalloc() を呼び出して事前に確保したバッファを指すポインタでなければなりません。このバッファが、サービス・ルーチン起動時にサービス・ルーチンに渡されたバッファと同じバッファである場合、そのバッファの配置は、BEA Tuxedo ATMI システム・ディスパッチャに一任されます。したがって、サービス・ルーチンをコーディングする人は、バッファが解放されているかどうかを気にする必要はありません。実際、ユーザがこのバッファを解放しようとしてもできません。しかし、tpforward() に渡されるバッファが、サービスが起動したものに等しいものでない場合は、tpforward() はそのバッファを解放します。*len* には送信するデータ・バッファの大きさを指定します。*data* が長さの指定を必要としないバッファを指すポインタである場合 (FML フィールド化バッファなど)、*len* は 0 でもかまいません。*data* が NULL の場合、*len* は無視され、長さ 0 のデータの要求が送信されます。

引数 *flags* は使用されません。この引数は将来の用途のために予約されており、0 (ゼロ) に設定されます。

戻り値

サービス・ルーチンは、呼び出し元である通信マネージャ・ディスパッチャには何も返しません。したがって、tpforward() は void で定義されています。詳細については tpreturn(3c) を参照してください。

エラー

関数に渡すパラメータの処理あるいはその関数のどちらかで何らかのエラーが発生した場合は、応答が送信されない限り、「失敗」メッセージが要求を起動した側に返されます。未終了の応答または従属する接続、あるいは「アボートのみ」とマーク付けされている呼び出し元のトランザクションがある場合、異常終了メッセージを生成する異常終了と見なされます。

ubbconfig ファイル中の SVCTIMEOUT か、TM_MIB 中の TA_SVCTIMEOUT が 0 でない場合にサービスのタイムアウトが発生すると、TPEV_SVCERR が返されます。この場合、tperrordetail(3) は TPED_SVCTIMEOUT を返します。

要求者は、エラーを示す TPESVCERR エラーによって、異常終了メッセージを検出します。このようなエラーが発生した場合、呼び出し元のデータは、送信されません。また、このエラーが発生すると、呼び出し元の現在のトランザクションは、「アボートのみ」とマーク付けされます。

サービス・ルーチンの処理中あるいは要求の転送中にトランザクション・タイムアウトになると、tpcall() または tpgetrply() で応答を待つ要求元は TPETIME エラーを受け取ります。また、待ち状態にある要求側は、どのようなデータも受信しません。サービス・ルーチンは、tpreturn() あるいは tpforward() のどちらかを使用して終了します。ただし、会話型サービス・ルーチンは tpreturn() を使用しなければならず、tpforward() は使用できません。

サービス・ルーチンが、tpreturn() または tpforward() のどちらも使用しないで戻る場合、(すなわち、サービス・ルーチンが C 言語の return 文を使用するかあるいは単に「関数の終わりで終了する」場合)、あるいは tpforward() が従来のサーバから呼び出される場合、サーバは、ログ・ファイルに警告メッセージを出力し、サービス・エラーを要求の起動側に返します。従属側へのすべてのオープン接続は、ただちに切断され、未終了の非同期応答は「処理済み」のマークが付けられます。障害が発生したときにサーバがトランザクション・モードの場合、そのトランザクションは「アボートのみ」とマークされます。tpreturn() または tpforward() がサービス・ルーチンの範囲外で (たとえば、クライアントで、あるいは、tpsvrinit() または tpsvrdone() で) 使用された場合は、これらのルーチンは、一切影響を及ぼすことなく単に終了します。

関連項目 tpalloc(3c)、tpconnect(3c)、tpreturn(3c)、tpservice(3c)、tpstrerrordetail(3c)

tpfree(3c)

名前

tpfree()— 型付きバッファの解放を行うルーチン

形式

```
#include <atmi.h>
void tpfree(char *ptr)
```

機能説明

tpfree() の引数は、以前に tpalloc() または tprealloc() によって得られたバッファを指すポインタです。*ptr* が NULL の場合は、動作は行われません。*ptr* が型付きバッファを指していない場合 (または、その前に tpfree() を使用して解放した領域を指している場合)、不定の結果が発生します。サービス・ルーチン内では、*ptr* がサービス・ルーチンに渡されたバッファを指している場合、tpfree() は単に終了し、そのバッファを解放しません。

ある種のバッファ・タイプは、バッファの解放処理の一環として、状態情報あるいは関連するデータを削除する必要があります。tpfree() は、バッファを解放する前にこうした関連情報を (通信マネージャ固有の方法に従って) 削除します。

tpfree() が終了した後は、*ptr* を BEA Tuxedo ATMI システムのルーチンの引数として使用したり、その他の方法で使用したりしないようにしてください。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tpfree() の呼び出しを発行できます。

tpfree() を使用して FML32 バッファを解放するとき、ルーチンは埋め込まれたバッファをすべて再帰的に解放して、メモリ・リークの発生を防ぎます。埋め込まれたバッファが解放されないようにするには、対応するポインタにヌルを指定してから tpfree() コマンドを発行します。上記で説明したように、*ptr* がヌルの場合アクションは発生しません。

戻り値

tpfree() は、呼び出し側に値を返しません。したがって、この関数は、void で宣言されています。

使用法

この関数を、C ライブラリの malloc()、realloc()、または free() と組み合わせて使用するのは避けてください (tpalloc() で割り当てたバッファを、free() を使用して解放しないでください)。

関連項目

「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」、tpalloc(3c)、tprealloc(3c)

tpgetadmkey(3c)

名前

tpgetadmkey()— 管理用認証キーを取得するルーチン

形式

```
#include <atmi.h>
long tpgetadmkey(TPINIT *tpinfo)
```

機能説明

tpgetadmkey() は、アプリケーションに特定の認証サーバーによって利用されます。このルーチンは、管理認証の目的のために指定ユーザーにふさわしいアプリケーション・セキュリティ・キーを返します。このルーチンは、tpsysadm() あるいは tpsysop() のクライアント名 (つまり *tpinfo->cltname*) を指定して呼び出す必要があります。そうでなければ、有効な管理キーは返されません。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpgetadmkey() の呼び出しを発行できません。

戻り値

tpgetadmkey() は、正常終了時には最上位ビットがセット (0x80000000) された 0 以外の値を返し、異常終了時には 0 を返します。*tpinfo* が NULL で、*tpinfo->cltname* が tpsysadm() もしくは tpsysop() でないか、またはユーザー ID が構成されたアプリケーション管理者でない時、0 が返されます。

エラー

0 が返されることで、有効な管理キーが割り当てられなかったことが分かります。

移植性

このインターフェイスは、BEA Tuxedo リリース 5.0 またはそれ以降が稼動する UNIX system サイトでしか利用できません。

関連項目

tpaddusr(1)、tpusradd(1)、tpinit(3c)、AUTHSVR(5)

『BEA Tuxedo アプリケーションの設定』

『BEA Tuxedo アプリケーション実行時の管理』

tpgetctxt(3c)

名前

tpgetctxt()— 現在のアプリケーション関連のコンテキスト識別子の取得

形式

```
#include <atmi.h>
int tpgetctxt(TPCONTEXT_T *context, long flags)
```

機能説明

tpgetctxt() は、現在のアプリケーション・コンテキストを表す識別子を取り出し、*context* に入れます。この関数はマルチスレッド環境ではスレッド単位で動作し、非スレッド環境ではプロセス単位で動作します。

一般にスレッドは、以下の動作を行います。

1. tpinit() を呼び出します。
2. tpgetctxt() を呼び出します。
3. *context* の値を次のように処理します。
   - マルチスレッドのアプリケーションの場合―同じプロセス内の別のスレッドに *context* の値を渡し、そのスレッドが tpsetctxt() を呼び出せるようにします。
   - シングルスレッドまたはマルチスレッドのアプリケーションの場合―指定されたコンテキストを後で復元できるように、このコンテキスト識別子を自身で保存します。

2 番目の引き数 *flags* は現在使用されていないので、0 に設定します。

tpgetctxt() は、マルチコンテキストのアプリケーションだけでなく、シングルコンテキストのアプリケーションでも呼び出すことができます。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tpgetctxt() の呼び出しを発行できます。

戻り値

正常終了時には、tpgetctxt() は負数でない値を返します。コンテキストは、以下のいずれかの形式で表される現在のコンテキスト ID に設定されます。

- 0 よりも大きいコンテキスト ID。マルチコンテキストのアプリケーション内のコンテキストを示します。
- TPSINGLECONTEXT。カレント・スレッドが TPMULTICONTEXTS フラグの指定されていない tpinit() を正常に実行したこと、または TPMULTICONTEXTS フラグが指定されていない tpinit() を正常に実行したプロセス内でカレント・スレッドが作成されたことを示します。TPSINGLECONTEXT の値は 0 です。

- TPNULLCONTEXT。カレント・スレッドがコンテキストに関連付けられていないことを示します。
- TPINVALIDCONTEXT。カレント・スレッドのコンテキスト状態が無効であることを示します。同じコンテキスト内で別のスレッドが作業している間に、マルチコンテキスト・クライアントのスレッドが tpterm() を呼び出すと、作業中のスレッドは TPINVALIDCONTEXT コンテキストになります。TPINVALIDCONTEXT の値は -1 です。

  TPINVALIDCONTEXT 状態のスレッドは、ほとんどの ATMI 関数を呼び出すことができません。呼び出せる関数、または呼び出せない関数のリストについては、「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」を参照してください。

  TPINVALIDCONTEXT コンテキスト状態の詳細については、tpterm(3c) を参照してください。

異常終了すると、tpgetctxt() は -1 を返し、tperrno を設定してエラー条件を示します。

エラー

異常終了時には、tpgetctxt() は tperrno を次のいずれかの値に設定します。

[TPEINVAL]
: 無効な引き数が指定されました。たとえば、コンテキストの値がヌルか、または *flags* の値が 0 以外です。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

関連項目

「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」、tpsetctxt(3c)、tpterm(3c)

tpgetlev(3c)

名前

tpgetlev()— トランザクション・モードかどうかチェックするルーチン

形式

```
#include <atmi.h>
int tpgetlev()
```

機能説明

tpgetlev() は現在のトランザクション・レベルを呼び出し元に返します。現時点では、定義されているレベルは 0 と 1 だけです。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tpgetlev() の呼び出しを発行できません。

戻り値

正常終了時 tpgetlev() は、トランザクション・モードではないということを示す 0 、またはトランザクション・モードだということを示す 1 のどちらかを返します。

異常終了すると、tpgetlev() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpgetlev() は tperrno() を次のいずれかの値に設定します。

[TPEPROTO]
: tpgetlev() が不正に呼び出されました。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

注意事項

BEA Tuxedo ATMI システムのトランザクションを記述するために tpbegin()、tpcommit()、および tpabort() を使用する場合、XA インターフェイスに準拠した (呼び出し元に妥当にリンクされている) リソース・マネージャが行う作業のみがトランザクションの特性を備えていることを記憶しておくことが重要です。トランザクションにおいて実行される他のすべての操作は、tpcommit() あるいは tpabort() のいずれにも影響されません。リソース・マネージャによって実行される操作が、BEA Tuxedo システムのトランザクションの一部となるように、XA インターフェイスを満たすリソース・マネージャをサーバにリンクします。詳細については buildserver(1) を参照してください。

関連項目

tpabort(3c)、tpbegin(3c)、tpcommit(3c)、tpscmt(3c)

tpgetrply(3c)

名前

tpgetrply()— 以前の要求に対する応答を受信するためのルーチン

形式

```
#include <atmi.h>
int tpgetrply(int *cd, char **data, long *len, long flags)
```

機能説明

tpgetrply() は、以前に送られた要求に対する応答を返します。この関数の最初の引数 *cd* は、tpacall() が返す呼び出し記述子を指します。デフォルトの設定では、この関数は、**cd* と一致する応答が届くか、タイムアウトが発生するまで処理を進めません。

data は、以前に tpalloc() で割り当てたバッファを指すポインタのアドレスでなければならず、*len* は tpgetrply() が正常に受信したデータ量を設定する long 型の値を指すようにしてください。正常終了時には、**data* その応答を収めたバッファを指し、**len* にはそのデータのサイズが入ります。FML と FML32 バッファは、通常最小サイズ 4096 バイトを確保します。応答が 4096 バイトより大きい場合には、バッファ・サイズは返されるデータを入れるのに十分な大きさに拡大します。リリース 6.4 では、バッファに対するデフォルトの割り当ては 1024 バイトです。また、最近使用したバッファの履歴情報が保持され、最適サイズのバッファをリターン・バッファとして再利用できます。

容量まで満たされていないかもしれない送信者側のバッファ (たとえば、FML または FML32 バッファ) は、送信に使用されたサイズになります。システムは、受信データのサイズを任意の量で拡大します。これは、受信者が送信者の割り当てたバッファ・サイズより小さく、送信されたデータのサイズより大きいバッファを受け取ることを意味します。

受信バッファのサイズは、増加することも減少することもあります。また、アドレスもシステムがバッファを内部で交換するごとに常に変更されます。応答バッファのサイズが変わったどうか (また変わったとしたらどれくらい変わったのか) を決定するには、tpgetrply() が **len* とともに発行される前に、合計サイズを比べてください。バッファ管理の詳細については、「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」を参照してください。

返されたときに **len* が 0 の場合は、応答にはデータ部分がなく、**data* やそれが指示するバッファは変更されていません。

**data* または *len* が NULL であるとエラーになります。

マルチスレッド・プログラムの各コンテキスト内では、次の条件があります。

- 特定のハンドルに対して tpgetrply(TPGETANY) 呼び出しと tpgetrply() の呼び出しを同時に発行することはできません。

- tpgetrply(TPGETANY) を同時に複数呼び出すこともできません。

これらの制限のどちらかに違反する tpgetrply() 呼び出しを発行した場合は、-1 が返され、tperrno は TPEPROTO に設定されます。

次のような呼び出しの発行は受け付けられます。

- 異なるハンドルに対する tpgetrply() は同時に呼び出すことができます。
- シングルコンテキスト内で tpgetrply(TPGETANY) を呼び出すときに、同時に別のコンテキストで tpgetrply() を呼び出すことは、TPGETANY の指定の有無にかかわらず可能です。

次に、有効な *flags* の一覧を示します。

TPGETANY
: このフラグは、tpgetrply() が、*cd* によって示される記述子を無視し、存在する応答があればそれらを返し、返された応答の呼び出し記述子を指すよう *cd* を設定します。応答が存在しなければ、tpgetrply() は１つの応答が届くまで待機します (デフォルトの設定の場合)。

TPNOCHANGE
: デフォルトでは、**data* が指すバッファ型と異なるバッファ型を受信すると、**data* のバッファ型は受信したバッファ型に変更されます (受信プロセスがそのバッファ型を認識できる場合)。このフラグが設定されていると、**data* が指すバッファのタイプは変更されません。すなわち、受信バッファのタイプおよびサブタイプは、**data* が指すバッファのものと同じでなければなりません。

TPNOBLOCK
: tpgetrply() は、応答が送られてくるまで待機しません。応答がキューから取り出せる状態であれば、tpgetrply() はその応答を取り込み、終了します。このフラグの指定がなく、応答もキューから取り出せる状態にない場合、呼び出し元は、応答が到着するまであるいはタイムアウト (トランザクション・タイムアウトまたはブロッキング・タイムアウト) が発生するまでブロックされます。

TPNOTIME
: このフラグは、呼び出し元をその応答に対して無期限にブロックでき、ブロッキング・タイムアウトの影響も受けないようにすることを指定します。トランザクション・タイムアウトは依然として発生する可能性があります。

TPSIGRSTRT
: シグナルが関数内部のシステム・コールを中断すると、中断されたシステム・コールは出しなおされます。

後に示す場合以外は、**cd* はその応答を受信した後は有効でなくなります。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpgetrply() の呼び出しを発行できません。

戻り値

tpgetrply() から正常に終了した場合、あるいは tperrno() が TPESVCFAIL に設定された状態で終了した場合、tpurcode() には、tpreturn() の一部として送信されたアプリケーション定義値が入ります。

異常終了すると、tpgetrply() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpgetrply() は tperrno() を次に示すように設定します。TPGETANY が設定されていない場合は、特に明記されていないかぎり、*cd は無効になります。TPGETANY が設定されている場合は、cd は異常が発生した応答の記述子を指します。応答が取り出せるようになる前にエラーが発生した場合には、cd は 0 を指します。また、特に明記しないかぎり、異常終了は呼び出し元のトランザクションが存在していても、それには影響しません。呼び出しが特定の tperrno() 値で異常終了した場合、中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、生成されたエラーの詳細が提供されます。詳細については、tperrordetail(3c) リファレンス・ページを参照してください。

[TPEINVAL]
無効な引数が与えられました (たとえば、cd、data、*data または len が NULL、あるいは flags が無効など)。cd が NULL でない場合、エラーが発生した後もこの値は有効で、応答は未終了のまま残されます。

[TPEOTYPE]
応答のタイプとサブタイプが呼び出し元が認識しているものでありません。あるいは、TPNOCHANGE が flags に設定されていて、*data のタイプとサブタイプがそのサービスから送られた応答のタイプおよびサブタイプと一致しません。*data の内容も *len も変更されていません。呼び出し元の現トランザクションに代わって応答が検索された場合、その応答は破棄されるため、トランザクションに「アボートのみ」のマークが付けられます。

[TPEBADDESC]
cd が無効な記述子を指しています。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、そのトランザクションは「アボートのみ」とマークされます。トランザクション・モードでなければ、ブロッキング・タイムアウトが発生し、TPNOBLOCK も TPNOTIME も指定されていませんでした。いずれの場合も、*data*、その内容、および **len* はいずれも変更されません。**cd* は、呼び出し元がトランザクション・モードでなければ (そして、TPGETANY が設定されていない場合) そのまま有効です。トランザクション・タイムアウトが発生すると、トランザクションがアボートされない限り、新しいリクエストの送信や未処理の応答の受信はできません (ただし、1 つの例外を除く)。これらの操作を行おうとすると、TPETIME が発生して失敗します。 1 つの例外とは、ブロックされず、応答を期待せず、かつ呼び出し元のトランザクションのために送信されない (つまり、TPNOTRAN、TPNOBLOCK および TPNOREPLY が設定された状態で tpacall() が呼び出される場合) 要求です。

[TPESVCFAIL]
呼び出し元の応答を送るサービス・ルーチンが、TPFAIL を設定した状態で tpreturn() を呼び出しました。これは、アプリケーション・レベルの障害です。サービスの応答の内容は (送信された場合) は、**data* が指すバッファに入ります。呼び出し元のトランザクションの代わりにサービス要求が発行された場合、トランザクションは「アボートのみ」とマークされます。トランザクションがタイムアウトしたかどうかに関わりなく、トランザクションが異常終了される前に有効な通信だけが、TPNOREPLY、TPNOTRAN、および TPNOBLOCK を設定した tpacall() を呼び出します。

[TPESVCERR]
サービス・ルーチンが tpreturn() あるいは tpforward() でエラーを検出しました。(たとえば、誤った引数が渡された場合など)。このエラーが発生すると、応答データは返されません (つまり、*data*、その内容、および **len* はいずれも変更されません)。呼び出し元のトランザクションの代わりにサービス要求が発行された場合、トランザクションは「アボートのみ」とマークされます。トランザクションがタイムアウトしたかどうかに関わりなく、トランザクションがアボートされる前の通信で有効であるのは、TPNOREPLY、TPNOTRAN、および TPNOBLOCK を設定した tpacall() の呼び出しだけです。UBBCONFIG ファイル中の SVCTIMEOUT か、TM_MIB 中の TA_SVCTIMEOUT が 0 でない場合にサービスのタイムアウトが発生すると、TPESVCERR が返されます。

[TPEBLOCK]
ブロッキング状態のため、TPNOBLOCK が指定されました。**cd* はそのまま有効です。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tpgetrply() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。リモート・ロケーションにあるメッセージ・キューがいっぱいの場合には、TPEOS が返される場合もあります。

関連項目 tpacall(3c)、tpalloc(3c)、tpcancel(3c)、tperrordetail(3c)、tprealloc(3c)、tpreturn(3c)、tpstrerrordetail(3c)、tptypes(3c)

tpgprio(3c)

名前

tpgprio()— サービス要求の優先順位を受け取るルーチン

形式

```
#include <atmi.h>
int tpgprio(void)
```

機能説明

tpgprio() は、カレント・スレッドがカレント・コンテキストで最後に送信または受信した要求の優先順位を返します。優先順位の範囲は 1 から 100 までです。最高優先順位は 100 です。tpgprio() は tpcall() または tpacall() (キュー管理機能がインストールされている場合には、tpenqueue() または tpdequeue()) の後に呼び出すことができ、返される優先順位は送信された要求のものです。また、tpgprio() はサービス・ルーチン内から呼び出して、呼び出されたサービスがどの優先順位で送られたかを明らかにします。tpgprio() は何回でも呼び出すことができ、次の要求が送られるまでは同じ値を返します。

マルチスレッドのアプリケーションでは、tpgprio() はスレッド単位で動作します。

会話プリミティブは優先順位とは関連付けられていないので、tpsend() や tprecv() を出しても、tpgprio() が返す優先順位には影響しません。また、会話サービス・ルーチンの場合も、tpcall() や tpacall() がそのサービス内から出されないかぎり、優先順位は関連付けられません。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpgprio() の呼び出しを発行できません。

戻り値

正常終了時には、tpgprio() は要求の優先順位を返します。

異常終了すると、tpgprio() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpgprio() は tperrno を次のいずれかの値に設定します。

[TPENOENT]
: tpgprio() が呼び出されましたが、(tpcall() または tpacall() を介して) 要求が送信されなかったか、要求が送られなかった会話型サービス内から呼び出されました。

[TPEPROTO]
: tpgprio() が不正に呼び出されました。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目 tpacall(3c)、tpcall(3c)、tpdequeue(3c)、tpenqueue(3c)、tpservice(3c)、tpsprio(3c)

tpimport(3c)

名前

tpimport()— メッセージ・バッファの外部化された表現を型付きメッセージ・バッファに変換

形式

```
#include <atmi.h>
int tpimport(char *istr, long ilen, char **obuf, long *olen,
long flags)
```

機能説明

tpimport() は、メッセージ・バッファの外部化された表現を、型付きメッセージ・バッファに変換します。外部化された表現とは、通常は伝送直前にメッセージ・バッファに追加される BEA Tuxedo ATMI ヘッダ情報を含まないメッセージ・バッファです。プロセスによって tpexport() 関数が呼び出され、型付きメッセージ・バッファが外部化された表現に変換されます。

istr に関連付けられているデジタル署名は、バッファがインポートされるときに確認されますが、インポート後も tpenvelope() を介して確認できます。

istr バッファ表現が暗号化されている場合、インポート・プロセスは解読に有効な秘密鍵にアクセス必要があります。解読はインポート・プロセス内で自動的に実行されます。

flags に TPEX_STRING が設定されていない場合、*istr* に含まれるバイナリ・データの長さは *ilen* に含まれます。*ilen* が 0 の場合は、*istr* はヌル終了文字列を指していると推定され、TPEX_STRING フラグが指定されているものと考えられます。

**obuf* は、(1) 以前 tpalloc() を呼び出すプロセスによって割り当てられたメッセージ・バッファ、または (2) システムによって受信プロセスに渡されたメッセージ・バッファのうち、いずれかの有効な型付きメッセージ・バッファを指している必要があります。バッファは結果に応じて再度割り当てられ、バッファのタイプまたはサブタイプは変更される場合があります。

**olen* は出力バッファに含まれる有効なデータ量に設定されます。*olen* が入力時にヌルの場合は、無視されます。

入力する外部化表現が文字列形式 (base 64 エンコード) の場合は、*flags* 引き数は TPEX_STRING に設定されています。それ以外の場合、入力は *ilen* の長さのバイナリ形式です。

戻り値

異常終了すると、この関数は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

[TPEINVAL]
無効な引数が指定されました。たとえば、*istr* の値がヌルか、または *flags* パラメータが正しく設定されていません。

[TPEPERM]
パーミッションに失敗しました。暗号化サービス・プロバイダが、解読に必要な秘密鍵にアクセスできませんでした。

[TPEPROTO]
プロトコルの異常終了が発生しました。異常終了には、*istr* のデータ形式が無効な場合や、デジタル署名が確認できなかった場合があります。

[TPESYSTEM]
エラーが発生しました。詳細については、システム・エラーのログ・ファイルを参照してください。

関連項目

tpenvelope(3c)、tpexport(3c)

tpinit(3c)

名前　　tpinit()— アプリケーションへの参加

形式

```
#include <atmi.h>
int tpinit(TPINIT *tpinfo)
```

機能説明

tpinit() は、クライアントが BEA Tuxedo ATMI システムのアプリケーションに参加するときに使用します。BEA Tuxedo ATMI システムのコミュニケーションあるいはトランザクション・ルーチンをクライアントが使用する前には、あらかじめクライアントは BEA Tuxedo ATMI システムのアプリケーションに参加しなければなりません。

tpinit() には、シングルコンテキスト・モードとマルチコンテキスト・モードの 2 つの操作モードがあります。次にこれについて説明します。シングルコンテキスト・モードでは tpinit() の呼び出しはオプションのため、シングルコンテキストのクライアントは、*tpinfo* をヌルに設定して tpinit() を透過的に呼び出す多くの ATMI ルーチン (たとえば、tpcall()) を呼び出すことによって、アプリケーションに参加することもできます。クライアントは、tpinit() を直接呼び出して、後に示すパラメータを設定する必要がある場合があります。また、マルチコンテキスト・モードが必要な場合、アプリケーションの認証が必要な場合 (UBBCONFIG(5) の SECURITY キーワードの説明を参照)、またはアプリケーションが独自のバッファ・タイプ・スイッチを提供する場合 (typesw(5) を参照) は、tpinit() を使用しなければなりません。tpinit() が正常に終了した場合は、クライアントはサービス要求を開始し、トランザクションを定義できます。

シングルコンテキスト・モードでは、tpinit() を 2 回以上呼び出す場合 (つまり、クライアントが既にアプリケーションに参加した後に呼び出す場合) は、何もアクションは実行されず、成功を示す戻り値が返されます。

マルチスレッド・クライアントの場合、TPINVALIDCONTEXT 状態のスレッドは tpinit() の呼び出しを発行できません。BEA Tuxedo ATMI アプリケーションに参加するには、マルチスレッドのワークステーション・クライアントは、たとえシングルコンテキスト・モードで動作している場合でも、必ず TPMULTICONTEXTS フラグを設定して tpinit() 関数を呼び出さなければなりません。

注記 TMNOTHREADS 環境変数を yes に設定しても、tpinit の TPMULTICONTEXTS モードは正しく動作します。この環境変数を yes に設定すると、スレッドを使用しないアプリケーションでマルチスレッド処理が無効になります。

TPINFO 構造体の説明

tpinit() の引数 *tpinfo* は、TPINIT タイプおよび NULL サブタイプの型付きバッファを指すポインタです。TPINIT はバッファ・タイプであり、ヘッダ・ファイル atmi.h に typedef で定義されています。このバッファは、tpinit() を呼び出す前に tpalloc() で割り当てなければなりません。このバッファの解放は、tpinit() の呼び出しの後、tpfree() を使用して行います。TPINIT 型付きバッファ構造体は次のようなメンバで構成されています。

```
char      usrname[MAXTIDENT+2];
char      cltname[MAXTIDENT+2];
char      passwd[MAXTIDENT+2];
char      grpname[MAXTIDENT+2];
long      flags;
long      datalen;
long      data;
```

usrname、cltname、grpname、および passwd はすべて NULL で終了する文字列です。usrname は呼び出し元を表す名前です。cltname は、その意味付けがアプリケーション側で定義されているクライアント名です。値 sysclient は、cltname フィールド用にシステムによって予約されています。usrname と cltname フィールドは、tpinit() 実行時にクライアントと関連付けられ、ブロードキャスト通知と管理統計情報の検索に使用されます。これらの名前は、MAXTIDENT の文字数を超えてはなりません。MAXTIDENT は 30 字に定義されています。passwd は、アプリケーション・パスワードとの認証に使用される非暗号化形式のアプリケーション・パスワードです。一方通行の暗号化方式に関する UNIX システムの制限から、passwd は 8 文字までしか意味をもちません。grpname は、クライアントをリソース・マネージャ・グループ名と関連付けるときに使用します。grpname が長さ 0 の文字列として設定されている場合は、クライアントはリソース・マネージャに関連付けられず、デフォルトのクライアント・グループに含まれます。ワークステーション・クライアントの場合、この値は NULL 文字列 (長さが 0 の文字列) でなければなりません。grpname は ACL グループとは関連がないことに注意してください。

シングルコンテキスト・モードとマルチコンテキスト・モード

tpinit() には、シングルコンテキスト・モードとマルチコンテキスト・モードの 2 つの操作モードがあります。シングルコンテキスト・モードでは、プロセスは 1 度に 1 つのアプリケーションに参加できます。このアプリケーションには、複数のアプリケーション・スレッドがアクセスできます。シングルコンテキスト・モードを指定するには、ヌルのパラメータを指定した tpinit() を呼び出すか、TPINIT 構造体の flags フィールドに TPMULTICONTEXTS フラグを指定せずに tpinit() を呼び出します。また、tpinit() が別の ATMI 関数によって暗黙的に呼び出されたときも、シングルコンテキスト・モードが指定されます。シングルコンテキスト・モードで動作するプロセスのコンテキスト状態は、TPSINGLECONTEXT です。

注記 TMNOTHREADS 環境変数を "yes" に設定しても、tpinit の TPMULTICONTEXTS モードは正しく機能します。

シングルコンテキスト・モードでは、tpinit() を 2 回以上呼び出す場合 (つまり、クライアントが既にアプリケーションに参加した後に呼び出す場合) は、何もアクションは実行されず、成功を示す戻り値が返されます。

TPINIT 構造体の flags フィールドに TPMULTICONTEXTS フラグを設定して tpinit() を呼び出すと、マルチコンテキスト・モードに移行します。マルチコンテキスト・モードでは、tpinit() を呼び出すたびに別のアプリケーション関連が作成されます。

アプリケーションの関連とは、プロセスと BEA Tuxedo ATMI アプリケーションを関連付けるコンテキストです。クライアントは、複数の BEA Tuxedo ATMI アプリケーションの関連を持ったり、同じアプリケーションに対して複数の関連を持つことができます。クライアントの関連は、すべて同じリリースの BEA Tuxedo ATMI システムを実行するアプリケーションに対する関連でなくてはなりません。また、すべての関連はネイティブ・クライアントかワークステーション・クライアントのいずれかでなければなりません。

ネイティブ・クライアントの場合、新しい関連を作成するときに TUXCONFIG 環境変数の値を使ってアプリケーションを識別します。ワークステーション・クライアントの場合、新しい関連を作成するときに WSNADDR または WSENVFILE 環境変数の値を使ってアプリケーションを識別します。カレント・スレッドのコンテキストは、新しい関連に設定されます。

マルチコンテキスト・モードでは、アプリケーションは tpgetctxt() を呼び出してカレント・コンテキストのハンドルを取得し、そのハンドルをパラメータとして tpsetctxt() に渡し、特定のスレッドまたはプロセスが動作するコンテキストを設定することができます。

シングルコンテキスト・モードとマルチコンテキスト・モードの両方を使用することはできません。アプリケーションがどちらかのモードを選択した場合、すべてのアプリエケーション関連で tpterm() が呼び出されるまで、他のモードで tpinit() を呼び出すことはできません。

TPINFO 構造体フィールドの説明

マルチコンテキスト・モードとシングルコンテキスト・モードを制御するほかに、flags の設定により、クライアント固有の通知メカニズムとシステム・アクセスのモードの両方を示すことができます。この 2 つの設定で、アプリケーションのデフォルト設定を上書きすることができます。これらの設定でアプリケーションのデフォルトを上書きできない場合、tpinit() はログ・ファイルに警告メッセージを記録し、その設定を無視して、tpinit() からの戻り時に flags フィールドの設定をアプリケーションのデフォルト設定に戻します。クライアントへの通知の場合、flags の値として次のものがあります。

TPU_SIG
: シグナルによる任意通知を選択します。このフラグは、シングルスレッド、シングルコンテキストのアプリケーション専用です。TPMULTICONTEXTS フラグが設定されている場合、このフラグは使用できません。

TPU_DIP
: ディップ・インによる任意通知を選択します。

TPU_THREAD
: BEA Tuxedo ATMI システムが管理する独立したスレッドで、THREAD 通知を選択します。このフラグは、マルチスレッドをサポートするプラットフォーム専用です。マルチスレッドをサポートしていないプラットフォームで TPU_THREAD が指定されると、無効な引き数と見なされ、エラーが返されて tperrno() が TPEINVAL に設定されます。

TPU_IGN
: 任意通知を無視します。

一度に上記の中から 1 つのフラグだけを使用できます。クライアントがフラグ・フィールドを使用して通知方法を選択しない場合、アプリケーションのデフォルトの方法が、tpinit() の終了時にフラグ・フィールドに設定されます。

システム・アクセス・モードの設定の場合、flags の値として次のものがあります。

TPSA_FASTPATH
: システム・アクセス・モードを fastpath に設定します。

TPSA_PROTECTED
: システム・アクセス・モードを protected に設定します。

一度に上記の中から 1 つのフラグだけを使用できます。クライアントが通知方法あるいはシステム・アクセス・モードをフラグ・フィールドで選択しない場合、tpinit() の終了時にアプリケーションのデフォルトの方法がフラグ・フィールドに設定されます。クライアントの通知方法とシステム・アクセス・モードの詳細については、UBBCONFIG(5) を参照してください。

アプリケーションがマルチスレッドまたはマルチコンテキストを使用する場合は、次のフラグを設定する必要があります。

TPMULTICONTEXTS
「シングルコンテキスト・モードとマルチコンテキスト・モード」の説明を参照してください。

datalen は、それに続くアプリケーション固有のデータの長さです。TPINIT 型付きバッファのバッファ・タイプ・スイッチ・エントリは、そのバッファに対して渡される合計サイズに基づいてこのフィールドを設定します (アプリケーション・データのサイズは、合計サイズから TPINIT 構造体自体のサイズを差し引き、構造体に定義されているプレース・ホルダのサイズを加えたものです)。data は、アプリケーションで定義された認証サービスに転送される可変長データ用のプレース・ホルダです。これは常に、この構造体の最後の要素となります。

マクロ TPINITNEED は、目的のアプリケーション固有のデータ長を収めるのに必要とされる TPINIT のバッファのサイズを判別するときに使用できます。たとえば、アプリケーション固有のデータを 8 バイトにしたい場合、TPINITNEED(8) は必要とされる TPINIT のバッファ・サイズを返します。

アプリケーションが BEA Tuxedo ATMI システムの認証機能を使用しない場合には、tpinfo に NULL 値を使用できます。NULL 引数を使用するクライアント・プロセスは、usrname、cltname、および passwd についてはデフォルトの設定である長さ 0 の文字列を獲得します。フラグは設定されず、アプリケーション・データも得られません。

戻り値

異常終了すると、tpinit() は呼び出しプロセスを元のコンテキストに維持したまま -1 を返し、tperrno を設定してエラー条件を示します。また tpurcode() は、AUTHSVR(5) サーバによって返される値に設定されます。

エラー

異常終了時には、tpinit() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
無効な引き数が指定されていました。*tpinfo* は NULL でありませんが、TPINIT 型付きバッファを指していません。

[TPENOENT]
領域制限のため、クライアントはアプリケーションに参加できません。

[TPEPERM]
クライアントは、パーミッションを持たないため、または正しいアプリケーション・パスワードが与えられていないために、アプリケーションに参加できません。許可が与えられない理由として、アプリケーション・パスワードが無効であった場合、アプリケーション固有の認証検査にパスできない場合、あるいは禁止されている名前を使用した場合などがあります。tpurcode() は、クライアントがアプリケーションに参加できない理由を説明するため、アプリケーション固有の認証サーバによって設定される場合があります。

[TPEPROTO]
tpinit() が不正に呼び出されました。たとえば、(a) 呼出し元がサーバである、(b) シングルコンテキスト・モードで TPMULTICONTEXTS フラグが指定されている、または (c) マルチコンテキスト・モードで TPMULTICONTEXTS フラグが指定されていない場合があります。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

相互運用性

tpchkauth() および、tpinit() の TPINIT タイプのバッファ引数に対する NULL 以外の値は、リリース 4.2 またはそれ以降を使用しているサイトでしか使用できません。

移植性

tpinit(3c) に記述されているインターフェイスは、UNIX システム、Windows および MS-DOS オペレーティング・システム上でサポートされています。ただし、シグナル・ベースの通知方法は、16 ビットの Windows および MS-DOS ではサポートされていません。tpinit() の実行時にこの通知方法が選択されると、userlog() メッセージが生成され、通知方式は自動的にディップインに設定されます。

環境変数

TUXCONFIG
ネイティブ・クライアントによって起動されたときに tpinit() 内で使用されます。クライアントが接続するアプリケーションを示します。なお、この環境変数は、tpinit() が呼び出されたときのみ参照されます。これ以降の呼び出しには、アプリケーション・コンテキストが使用されます。

WSENVFILE
ワークステーション・クライアントが呼び出すときに tpinit() 内で使用されます。この変数には、環境変数の設定条件を収めたファイルを指定しますが、この設定は呼び出し元の環境で行うようにします。ワークステーション・クライアントに必要とされる環境変数の設定に関する詳細については、compilation(5) を参照してください。なお、このファイルは、tpinit() が呼び出されるとき (その前ではなく) にのみ処理されます。

WSNADDR

ワークステーション・クライアントが呼び出すときに tpinit() 内から使用されます。これは、アプリケーションをアクセスするときに使用されるワークステーション・リスナ・プロセスのネットワーク・アドレスを示します。この変数はワークステーション・クライアントの場合は必須で、ネイティブ・クライアントの場合は無視されます。

TCP/IP アドレスは次の形式で指定します。

//host.name:port_number

//#.#.#.#:port_number

最初の形式では、ドメインはローカル・ネーム解決機能 （通常 DNS）を使って、*hostname* のアドレスを見つけます。*hostname* はローカル・マシンでなければならず、ローカル・ネーム解決機能は、ローカル・マシンのアドレスへ *hostname* を解決しなければなりません。

2 番目文字列 *#.#.#.#* はドットで区切った 10 進数の形式です。ドットで区切った 10 進数の形式では、各 # は 0 から 255 までの数でなければなりません。このドットで区切った 10 進数は、ローカル・マシンの IP アドレスを表現します。

上記両方の形式で、*port_number* はドメイン・プロセスが着信要求をリスンする TCP ポート番号です。*port_number* は 0 から 65535 の間の数字または名前です。*port_number* が名前の場合は、ローカル・マシンのネットワーク・サービス・データベースになければなりません。

アドレスは、先頭に 0x がついている場合は、16 進形式で指定することもできます。0x の後の各文字は、0 から 9 までの数字か、A から F までの英字 （大文字・小文字に関係なく）です。16 進数の形式は、IPX/SPX や TCP/IP のような任意のバイナリ・ネットワーク・アドレスに使うことができます。

アドレスはまた、任意の文字列として指定することもできます。値は、コンフィギュレーション・ファイルの中の NETWORK セクションの NLSADDR パラメータに指定された値と同じでなければなりません。

WSNADDR アドレス用のコンマで区切られたパス名のリストを指定すると、複数のアドレスを指定することができます。接続が確立するまで順番にアドレス指定が試みられます。アドレス・リストのメンバは、どれでもパイプで区切られたネットワーク・アドレスのかっこ付きのグループとして指定することができます。

例 :

```
WSNADDR=(//m1.acme.com:3050|//m2.acme.com:3050),//m3.acme.com:3050
```

Windows 下で実行するためには、アドレス文字列は以下のように表します。

```
set WSNADDR=(//m1.acme.com:3050^|//m2.acme.com:3050),//m3.acme.com:3050
```

パイプ記号 (|) は Windows では特殊文字と見なされるため、コマンド行で指定する場合は、Windows 環境のエスケープ文字、カラット (^) を前に付けます。ただし、envfile で WSNADDR が定義されている場合、BEA Tuxedo ATMI システムは tuxgetenv(3c) 関数を介して WSNADDR が定義する値を取得します。このコンテキストではパイプ記号 (|) は特殊記号と見なされないため、カラット (^) を前に付ける必要はありません。

BEA Tuxedo ATMI システムはかっこ付きアドレスを無作為に選択します。この方法は、一連のリスナ・プロセスに対してランダムに負荷分散します。接続が確立するまで順番にアドレス指定が試みられます。ワークステーション・リスナを呼び出すには、アプリケーションのコンフィギュレーション・ファイルの値を使用してください。この値が、"0x" で始まる文字列の場合は、16 進値文字列と解釈され、それ以外の場合は、ASCII 文字列と解釈されます。

WSFADDR

ワークステーション・クライアントが呼び出すときに tpinit() 内から使用されます。WSFADDR は、ワークステーション・クライアントがワークステーション・リスナまたはワークステーション・ハンドラに接続するときに使用するネットワーク・アドレスを指定します。この変数は、WSFRANGE 変数とともに、ワークステーション・クライアントが送信接続を確立する前にバインドする TCP/IP ポートの範囲を決めます。このアドレスは、TCP/IP アドレスでなければなりません。TCP/IP アドレスのポート部分は、ワークステーション・クライアントが TCP/IP ポート範囲でバインドできるベース・アドレスを表します。WSFRANGE 変数には、範囲のサイズを指定します。たとえば、このアドレスが //mymachine.bea.com:30000 で WSFRANGE が 200 の場合、この *LMID* から送信接続を確立しようとするネイティブ・プロセスは、すべて、mymachine.bea.com 3000 から 30200 の間のポートをバインドします。この変数が設定されていないと空の文字列にリセットされ、この場合はオペレーティング・システムによってローカル・ポートがランダムに選択されます。

WSFRANGE

ワークステーション・クライアントが呼び出すときに tpinit() 内で使用されます。この変数は、ワークステーション・クライアント・プロセスが、送信接続を確立する前にバインドを試みる TCP/IP ポートの範囲を指定します。WSFADDR パラメータは、範囲のベース・アドレスを指定します。たとえば、WSFADDR パラメータが //mymachine.bea.com:30000 で WSFRANGE が 200 に設定されている場合、この *LMID* から送信接続を確立するすべてのネイティブ・プロセスは、mymachine.bea.com 30000 から 30200 の間にあるポートをバインドします。有効範囲は 1 から 65535 で、デフォルト値は 1 です。

WSDEVICE
: ワークステーション・クライアントが呼び出すときに tpinit() 内で使用されます。これは、ネットワークのアクセス時に使用するデバイス名を示します。この変数はワークステーション・クライアントが使用し、ネイティブ・クライアントの場合は無視されます。なお、ソケットや NetBIOS などトランスポート・レベルのネットワーク・インターフェイスはデバイス名を必要としません。このようなインターフェイスによってサポートされているワークステーション・クライアントは、WSDEVICE を指定する必要はありません。

WSTYPE
: ワークステーション・クライアントが呼び出すときに tpinit() 内から使用され、ネイティブ・サイトとの間で符号化 / 復号化の責任範囲について調整を行います。この変数はワークステーション・クライアントの場合は省略可能で、ネイティブ・クライアントの場合は無視されます。

WSRPLYMAX
: tpinit() によって使用され、アプリケーションの応答をファイルに格納する前にバッファに入れるために使用するコア・メモリの最大サイズを設定します。このパラメータのデフォルトは、256,000 バイトです。詳細については、プログラミング・マニュアルを参照してください。

TMMINENCRYPTBITS
: BEA Tuxedo ATMI システムに接続するのに必要な暗号化の最小レベルを確立するときに使用されます。"0" は暗号化がないことを意味し、"56" と "128" は、暗号化キーのビット長を示します。後方互換のため、40 ビットのリンク・レベルの暗号化も使用できます。この最小レベルの暗号化が一致しない時は、リンクの確立は失敗します。デフォルトの値は "0" です。

TMMAXENCRYPTBITS
: BEA Tuxedo ATMI システムに接続するときに、このレベルまで暗号化を調整するのに使用されます。"0" は暗号化のないことを意味し、"56" と "128" は暗号キーのビット長を示します。また後方互換のため、40 ビットのリンク・レベルの暗号化も使用できます。デフォルトの値は "128" です。

警告

シグナル・ベースの通知は、マルチコンテキスト・モードでは使用できません。また、シグナルの制約によって、クライアントがシグナル・ベースの通知を選択しても、システムがそれを使用できない場合があります。このような場合、システムは、選択されたクライアントに対する通知をディップ・インに切り替えることを示すログ・メッセージを生成し、クライアントはそれ以降ディップ・イン通知によって通知されます通知方式の詳細については、UBBCONFIG(5) の RESOURCES セクションの NOTIFY パラメータの説明を参照してください。

クライアントのシグナル通知は、必ずシステムによって行われるので、通知呼び出しの起動元がどこであっても、通知の動作は一貫しています。したがって、シグナル・ベースの通知を使用するには次の条件が必要です。

- ネイティブ・クライアントは、アプリケーション管理者として実行している必要があります。
- ワークステーション・クライアントは、アプリケーション管理者として実行している必要はありません。

アプリケーション管理者の ID は、アプリケーション・コンフィギュレーションの一部として識別されます。

クライアントにシグナル・ベースの通知を選択すると、ある種の ATMI 呼び出しは正常に実行できないことがあります。このとき、TPSIGRSTRT の指定がない場合、任意通知型メッセージを受け取るため、TPGOTSIG が返されます。

関連項目　「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」、tpgetctxt(3c)、tpsetctxt(3c)、tpterm(3c)

tpkey_close(3c)

名前

tpkey_close()— 以前にオープンしたキー・ハンドルのクローズ

形式

```
#include <atmi.h>
int tpkey_close(TPKEY hKey, long flags)
```

機能説明

tpkey_close() は、以前にオープンしたキー・ハンドルと、それに関連するすべてのリソースを解放します。プリンシパルの秘密鍵など、すべてのセンシティブ情報がメモリから消去されます。

キー・ハンドルは、次のいずれかの方法でオープンできます。

- tpkey_open() の明示的呼び出し
- tpenvelope() からの出力

tpkey_close() を呼び出してキー・リソースを解放するのは、アプリケーションの役目です。あるプロセスがキーをクローズすると、同じプロセスがそのキー・ハンドルを使ってデジタル署名や暗号化のメッセージ・バッファを登録することはできなくなります。プロセスが TPKEY_AUTOSIGN または TPKEY_AUTOENCRYPT フラグを指定した tpkey_open() でキーをオープンした場合、キーをクローズした後の通信操作にはそのキー・ハンドルは適用されません。

ただし、キーをクローズした後でも、そのキー・ハンドルは、クローズ前に登録された関連署名要求や暗号化要求に対しては有効です。クローズしたキーに関連付けられている最後のバッファが解放されるか上書きされると、そのキーに属していたリソースは解放されます。

flags は将来使用する予定であり、現在は必ずゼロを指定してください。

戻り値

異常終了すると、この機能は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

[TPEINVAL]
　無効な引数が指定されました。たとえば、値 *hKey* が無効です。

[TPESYSTEM]
　エラーが発生しました。詳細については、システム・エラーのログ・ファイルを参照してください。

関連項目

tpenvelope(3c)、tpkey_getinfo(3c)、tpkey_open(3c)、tpkey_setinfo(3c)

tpkey_getinfo(3c)

名前

tpkey_getinfo()— キー・ハンドルに関連付けられた情報の取得

形式

```
#include <atmi.h>
int tpkey_getinfo(TPKEY hKey, char *attribute_name, void *value,
long *value_len, long flags)
```

機能説明

tpkey_getinfo() は、キー・ハンドルに関する情報を報告します。キー・ハンドルは、特定のプリンシパルのキーと、それに関連付けられている情報を表します。

調査対象のキーは、*hKey* 入力パラメータによって識別されます。情報が要求されている属性は、*attribute_name* 入力パラメータによって識別されます。暗号サービス・プロバイダ固有の属性もありますが、以下のような基本的属性はすべてのプロバイダがサポートしています。

属性	値
PRINCIPAL	キー (キー・ハンドル) に関連付けられているプリンシパルを識別する名前。ヌル終了文字列として表されます。
PKENCRYPT_ALG	公開鍵暗号化のためのキーによって使用される公開鍵アルゴリズムの ASN.1 Distinguished Encoding Rules (DER) オブジェクト識別子。 RSA のオブジェクト識別子については、次の表「アルゴリズム・オブジェクト識別子とアルゴリズムの対象表」を参照してください。
PKENCRYPT_BITS	公開鍵アルゴリズムのキーの長さ (RSA モジュロ・サイズ)。この値は 512 から 2048 の範囲でなければなりません。
SIGNATURE_ALG	デジタル署名のキーによって使用されるデジタル署名アルゴリズムの ASN.1 DER オブジェクト識別子。 RSA と DSA のオブジェクト識別子については、次の表「アルゴリズム・オブジェクト識別子とアルゴリズムの対象表」を参照してください。
SIGNATURE_BITS	デジタル署名アルゴリズムのキーの長さ (RSA モジュロ・サイズ)。この値は 512 から 2048 ビットの範囲でなければなりません。

属性	値
ENCRYPT_ALG	バルク・データ暗号化のキーによって使用される対称鍵アルゴリズムの ASN.1 DER オブジェクト識別子。 DES、3DES、RC2 のオブジェクト識別子については、次の表「アルゴリズム・オブジェクト識別子とアルゴリズムの対象表」を参照してください。
ENCRYPT_BITS	対称鍵アルゴリズムのキーの長さ。この値は 40 から 128 ビットの範囲でなければなりません。 キー長が固定されたアルゴリズムが ENCRYPT_ALG に設定されると、ENCRYPT_BITS の値は自動的にその固定キー長に設定されます。たとえば、ENCRYPT_ALG が DES に設定されると、ENCRYPT_BITS の値は自動的に 56 に設定されます。
DIGEST_ALG	デジタル署名のキーによって使用されるメッセージ・ダイジェスト・アルゴリズムの ASN.1 DER オブジェクト識別子。 MD5 と SHA-1 のオブジェクト識別子については、次の表「アルゴリズム・オブジェクト識別子とアルゴリズムの対象表」を参照してください。
PROVIDER	暗号サービス・プロバイダの名前。
VERSION	暗号サービス・プロバイダのソフトウェアのバージョン番号。

デフォルトの公開鍵インプリメンテーションがサポートする ASN.1 DER アルゴリズムのオブジェクト識別子を、表 11 に示します。

表 11 アルゴリズム・オブジェクト識別子とアルゴリズムの対象表

ASN.1 DER アルゴリズムのオブジェクト識別子	アルゴリズム
{ 0x06, 0x08, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x02, 0x05 }	MD5
{ 0x06, 0x05, 0x2b, 0x0e, 0x03, 0x02, 0x1a }	SHA1
{ 0x06, 0x09, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x01, 0x01, 0x01 }	RSA
{ 0x06, 0x05, 0x2b, 0x0e, 0x03, 0x02, 0x0c }	DSA
{ 0x06, 0x05, 0x2b, 0x0e, 0x03, 0x02, 0x07 }	DES
{ 0x06, 0x08, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x03, 0x07 }	3DES
{ 0x06, 0x08, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x03, 0x02 }	RC2

指定された *attribute_name* パラメータに関連付けられている情報は、値によって示されるメモリ位置に格納されます。この位置に格納できる最大データ量は、呼び出し元によって *value_len* に指定されます。

tpkey_getinfo() が完了すると、*value_len* は実際に返されたデータ・サイズに設定されます。この場合、string 値の終了ヌル値も含まれます。返されるバイト数が *value_len* より大きい場合、tpkey_getinfo() は異常終了し (TPELIMIT エラー・コードを返し)、*value_len* を必要な大きさに設定します。

flags は将来使用する予定であり、現在は必ずゼロを指定してください。

戻り値

異常終了すると、この関数は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

[TPEINVAL]
無効な引数が指定されました。たとえば、*hKey* が無効です。

[TPESYSTEM]
エラーが発生しました。詳細については、システム・エラーのログ・ファイルを参照してください。

[TPELIMIT]
要求された属性値を保持するための十分なスペースがありません。

[TPENOENT]
要求された属性は、このキーに関連付けられていません。

関連項目

tpkey_close(3c)、tpkey_open(3c)、tpkey_setinfo(3c)

tpkey_open(3c)

名前

tpkey_open()— デジタル署名の生成、メッセージの暗号化、またはメッセージの解読のためのキー・ハンドルのオープン

形式

```
#include <atmi.h>
int tpkey_open(TPKEY *hKey, char *principal_name, char *location,
char *identity_proof, long proof_len, long flags)
```

機能説明

tpkey_open() は、呼び出し側のプロセスからキー・ハンドルを使用できるようにします。キー・ハンドルは、特定のプリンシパルのキーとその関連情報を表します。

キーは、以下のうち 1 つまたは複数の目的に使用できます。

- デジタル署名の生成。これによって、型付きメッセージ・バッファの内容を保護し、指定されているプリンシパルがメッセージを発信したことを証明します。プリンシパルは個人またはプロセスです。このタイプのキーは秘密鍵であり、キーの所有者のみが使用できます。

  プリンシパル名と TPKEY_SIGNATURE または TPKEY_AUTOSIGN フラグを設定して tpkey_open() を呼び出すと、そのプリンシパルの秘密鍵とデジタル証明書に対するハンドルが返されます。

- デジタル署名の検証。これによって、型付きメッセージ・バッファの内容が変更されていないこと、また特定のプリンシパルによってメッセージが送信されたことが証明されます。

  署名の検証には tpkey_open() を呼び出す必要はありません。この検証プロセスでは、デジタル署名付きメッセージが添付されたデジタル証明書に指定されている公開鍵を使用して署名が検証されます。

- 特定のプリンシパルに宛てたメッセージ・バッファの暗号化。このタイプのキーは、プリンシパルの公開鍵とデジタル署名にアクセスするすべてのプロセスで使用できます。

  プリンシパル名と TPKEY_ENCRYPT または TPKEY_AUTOENCRYPT フラグを設定して tpkey_open() を呼び出すと、プリンシパルのデジタル証明書を介して、プリンシパルの公開鍵に対するハンドルが返されます。

- 特定のプリンシパルに送られたメッセージ・バッファの解読。このタイプのキーは秘密鍵であり、キーの所有者だけが利用できます。

  プリンシパルの名前と TPKEY_DECRYPT フラグを使って tpkey_open() を呼び出すと、プリンシパルの秘密鍵とデジタル証明書に対するハンドルが返されます。

tpkey_open() によって返されたキー・ハンドルは *hKey に格納されますが、値にヌルを使用することはできません。

principal_name 入力パラメータは、キーの所有者の ID を指定します。*principal_name* の値がヌル・ポインタまたは空の文字列の場合、デフォルト ID が使用されます。デフォルト ID は、現在のログイン・セッション、現在のオペレーティング・システムのアカウント、またはローカル・ハードウェア・デバイスなど別の属性に基づいています。

キーのファイル位置は、*location* パラメータに渡されます。基本となるキー管理プロバイダが位置を示すパラメータを必要としない場合、このパラメータの値はヌルを使用できます。

principal_name の ID を認証するには、パスワードやパス・フレーズなどの証明資料が必要になります。証明資料が必要な場合は、*identity_proof* によって資料を参照する必要があります。証明資料が不要な場合、このパラメータの値はヌルでかまいません。

証明資料の長さ (バイト) は、*proof_len* で指定します。*proof_len* が 0 の場合 *identity_proof* はヌルで終了する文字列と見なされます。

キーの操作モードに必要なキー・アクセスのタイプは、*flags* パラメータで指定します。

TPKEY_SIGNATURE:
: この秘密鍵は、デジタル署名の生成に使用されます。

TPKEY_AUTOSIGN:
: このプロセスがメッセージ・バッファを伝送すると、公開鍵ソフトウェアが署名者の秘密鍵を使用してデジタル署名を生成し、そのデジタル署名をバッファにアタッチします。TPKEY_SIGNATURE が暗黙的に指定されています。

TPKEY_ENCRYPT:
: この公開鍵は、暗号化されたメッセージの受信者を識別するのに使用されます。

TPKEY_AUTOENCRYPT:
: このプロセスによってメッセージ・バッファが伝送されると、公開鍵ソフトウェアがメッセージの内容を暗号化し、受信者の公開鍵を使って暗号化エンベロープを生成し、その暗号化エンベロープをバッファにアタッチします。TPKEY_ENCRYPT が暗黙的に指定されています。

TPKEY_DECRYPT:
: この秘密鍵は解読に使用します。

これらのフラグは 1 つまたは複数を組み合わせて使用することができます。キーを暗号化のみに使用する場合 (TPKEY_ENCRYPT)、*identity_proof* は必要ないのでヌルに設定できます。

戻り値

正常終了すると、**hKey* はこのキーを表す値に設定され、tpsign() や tpseal() などの関数で使用できるようになります。

異常終了すると、この関数は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

[TPEINVAL]
無効な引数が指定されました。たとえば、*hKey* の値がヌル、または *flags* パラメータが正しく設定されていません。

[TPEPERM]
パーミッションに失敗しました。暗号サービス・プロバイダが、指定された証明情報と現在の環境下では、このプリンシパルの秘密鍵にアクセスできません。

[TPESYSTEM]
エラーが発生しました。詳細については、システム・エラーのログ・ファイルを参照してください。

関連項目

tpkey_close(3c)、tpkey_getinfo(3c)、tpkey_setinfo(3c)

tpkey_setinfo(3c)

名前

tpkey_setinfo()— キー・ハンドルに関連するオプション属性パラメータの設定

形式

```
#include <atmi.h>
int tpkey_setinfo(TPKEY hKey, char *attribute_name, void *value,
long value_len, long flags)
```

機能説明

tpkey_setinfo() は、キー・ハンドルのオプション属性パラメータを設定します。キー・ハンドルは指定されたプリンシパルのキーと、それに関連する情報を表します。

情報が修正されるキーは、*hKey* 入力パラメータで識別されます。情報が修正される属性は、*attribute_name* 入力パラメータで識別されます。一部の暗号サービス・プロバイダに固有の属性もありますが、tpkey_getinfo(3c) リファレンス・ページに示す基本的属性は、すべてのプロバイダがサポートしています。

attribute_name パラメータに関連付けられた情報は、*value* によって示されるメモリ位置に格納されます。*value* のデータ内容が自己記述型の場合、*value_len* は無視されます (0 でかまいません)。それ以外の場合、*value_len* には *value* 内のデータの長さが格納されている必要があります。

引数 *flags* は使用されません。この引数は将来の用途のために予約されており、0 (ゼロ) に設定されます。

戻り値

異常終了すると、この関数は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

[TPEINVAL]
無効な引数が指定されました。たとえば、*hKey* が有効なキーでない場合や *attribute_name* が読み取り専用の値を参照している場合などです。

[TPELIMIT]
指定した *value* が大きすぎます。

[TPESYSTEM]
エラーが発生しました。詳細については、システム・エラー・ログ・ファイルを参照してください。

[TPENOENT]
要求された属性を、キーの暗号サービス・プロバイダが認識できません。

関連項目

tpkey_close(3c)、tpkey_getinfo(3c)、tpkey_open(3c)

tpnotify(3c)

関連項目　tpnotify()— クライアント識別子によって通知を送信するルーチン

形式

```
#include <atmi.h>
int tpnotify(CLIENTID *clientid, char *data, long len, long flags)
```

機能説明　tpnotify() は、各クライアントに任意通知型メッセージを送信できるようにします。

clientid は、以前のあるいは現在のサービス呼び出しの TPSVCINFO 構造体から保存された、または、他の何らかの通信機構によってクライアントに渡された (たとえば、管理インターフェイスを使って検索された) クライアント識別子を指すポインタです。

この要求のデータ部は *data* によって示され、以前に tpalloc() によって割り当てられたバッファです。*len* に送信するデータの大きさを指定します。ただし、*data* が長さの指定を必要としないタイプのバッファを指す場合 (たとえば、FML フィールド化バッファ)、*len* は 0 でかまいません。また、*data* は NULL であってもかまいません。この場合、*len* は無視されます。

tpnotify() が正常終了した場合、メッセージはシステムに渡され、指定されたクライアントに送信されます。TPACK フラグが設定されている場合は、正常終了は、クライアントがメッセージを受信したことを意味します。さらに、クライアントが任意通知型のメッセージ・ハンドラに登録している場合は、ハンドラが呼び出されます。

次に、有効な *flags* の一覧を示します。

TPACK
: 要求は送信され、呼び出し元は承認メッセージがターゲット・クライアントから受信されるまでブロックします。

TPNOBLOCK
: この要求は、通知の送信中にブロッキング条件が存在する場合 (たとえば、メッセージの送信先である内部バッファがいっぱいの場合など) には、送信されません。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。トランザクション・タイムアウトは依然として発生する可能性があります。

TPSIGRSTRT
: シグナルが関数内部のシステム・コールを中断すると、中断されたシステム・コールは出しなおされます。

  TPACK フラグを設定しない限り、tpnotify() は、メッセージがクライアントに送られるまで待機していません。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpnotify() の呼び出しを発行できません。

戻り値

異常終了すると、tpnotify() は -1 を返し、tperrno() を設定してエラー条件を示します。呼び出しが異常終了して tperrno() に特別の値が指定された場合、間に ATMI 呼び出しを入れず、引き続き tperrordetail() を呼び出すと、エラーに関する詳細な情報が提供されます。詳細については、tperrordetail(3c) リファレンス・ページを参照してください。

エラー

異常終了時には、tpnotify() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
: 無効な引数が指定されました (たとえば、無効なフラグなど)。

[TPENOENT]
: ターゲット・クライアントが存在せず、TPACK フラグが設定されました。

[TPETIME]
: ブロッキング・タイムアウトが発生し、TPNOBLOCK と TPNOTIME のいずれも指定されなかったか、TPACK は設定されたが承認が受信されず、TPNOTIME が指定されませんでした。

[TPEBLOCK]
: 呼び出し時にブロッキング条件が検出されましたが、TPNOBLOCK が指定されていました。

[TPGOTSIG]
: シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
: tpnotify() が不正に呼び出されました。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

[TPERELEASE]
TPACK が指定され、ターゲットは承認プロトコルをサポートしない BEA Tuxedo の前のリリースからのクライアントです。

関連項目

「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」、tpalloc(3c)、tpbroadcast(3c)、tpchkunsol(3c)、tperrordetail(3c)、tpinit(3c)、tpsetunsol(3c)、tpstrerrordetail(3c)、tpterm(3c)

tpopen(3c)

名前　tpopen()— リソース・マネージャをオープンするルーチン

形式

```
#include <atmi.h>
int tpopen(void)
```

機能説明　tpopen() は、呼び出し元がリンクされるリソース・マネージャをオープンします。呼び出し元には、多くとも 1 つのリソース・マネージャしかリンクできません。この関数はリソース・マネージャ固有の open() 呼び出しの代わりに使用するもので、これにより、移植性を損なう可能性のある呼び出しをサービス・ルーチンからなくすことができます。リソース・マネージャは初期化の内容がそれぞれで異なるため、個々のリソース・マネージャをオープンするために必要な情報をコンフィギュレーション・ファイルに記述します。

リソース・マネージャがすでにオープンされている場合 (すなわち、tpopen() を 2 回以上呼び出した場合)、何も処理は行われず、正常終了を示すコードが返されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT のスレッドは tpopen() の呼び出しを発行できません。

戻り値　異常終了すると、tpopen() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー　異常終了時には、tpopen() は tperrno() を次のいずれかの値に設定します。

[TPERMERR]
: リソース・マネージャを正しくオープンできませんでした。より詳しい理由は、そのリソース・マネージャを独自の方法で調査することで得ることができます。ただし、エラーの正確な性質を判別するための呼び出しを使用すると、移植性が損なわれます。

[TPEPROTO]
: tpopen() が不正なコンテキストで呼び出されました (たとえば、BEA Tuxedo システム・サーバ・グループに結合していないクライアントにより)。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

関連項目　tpclose(3c)

tppost(3c)

名前

tppost()— イベントをポストする

形式

```
#include <atmi.h>
int tppost(char *eventname, char *data, long len, long flags)
```

機能説明

呼び出し元は tppost() を使用して、イベントとそれに伴うすべてのデータをポストします。イベント名は *eventname* に指定し、*data* は、NULL 以外の場合はデータを指すようにします。ポストされたイベントとそのデータは、BEA Tuxedo ATMI イベント・ブローカによって、*eventname* に対して評価が成功するサブスクリプションを持ち、*data* に対して評価が成功するオプションのフィルタ・ルールを持つすべてのサブスクライバにディスパッチされます。

eventname には、最大で 31 文字の NULL で終了する文字列を指定します。*eventname* の最初の文字はドット (".") であってはいけません。なぜなら、この文字は BEA Tuxedo ATMI システム自身が定義するあらゆるイベントの最初の文字として予約されているからです。

data には、NULL 以外の場合は、tpalloc() によって以前に割り当てたバッファを指定し、*len* にはバッファ内のイベントと共にポストするバッファに入っているデータの長さを指定しなければなりません。長さを指定する必要のないタイプのバッファ (FML フィールド化バッファなど) を *data* が指す場合、*len* は無視されます。*data* が NULL の場合、*len* は無視され、イベントはデータなしでポストされます。

tppost() をトランザクション中で使用する場合は、トランザクションの境界を拡大して、これらのサーバ上、およびイベント・ブローカが通知する安定記憶域上のメッセージ・キューを含むようにすることができます。トランザクションによるポストが行われると、ポスト元のトランザクションに代わって、ポストされたイベントの受信側の一部 (たとえば、サーバおよびキューなど) には通知され、一部 (たとえば、クライアントなど) には通知されません。

ポスト元がトランザクション内にあり、TPNOTRAN がセットされている場合は、ポストされたイベントはイベント・ブローカに渡されます。これは、イベント・ブローカがイベントをポスト元のトランザクションの一部として処理できるようにするためです。イベント・ブローカはトランザクションによるイベント通知を、tpsubscribe() に渡される *ctl->flags* パラメータで TPEVTRAN ビットの設定を使用したサービス・ルーチンおよび安定記憶域上のキューのサブスクリプションだけにディスパッチします。イベント・ブローカは、クライアントへの通知、および tpsubscribe() に渡される *ctl->flags* パラメータで TPEVTRAN ビットの設定を使用しなかったサービス・ルーチンおよび安定記憶域上のキューにあるサブスクリプションのディパッチも行いますが、これはポスト元プロセスのトランザクションの一部としてではありません。

ポスト元がトランザクションの外部にある場合、イベントに関連するサービスが異常終了すると、tppost() は肯定応答のない一方向のポストになります。このような状況は、イベント用に TPEVTRAN が設定されている場合でも起こります (この設定には、tpsubscribe() に渡される *ctl->flags* パラメータを使用します)。ポスト元がトランザクション内にある場合は、イベントに関連するサービスが異常終了すると tppost() は TPESVCFAIL を返します。

次に、有効な *flags* の一覧を示します。

TPNOTRAN
: 呼び出し元がトランザクション・モードにあり、このフラグがセットされている場合は、イベントのポストは呼び出し元のトランザクションの代わりに実行されません。トランザクション・モードにあるこのフラグをセットする呼び出し元は、イベントのポストの際に、依然としてトランザクション・タイムアウトの対象となります。イベントのポストが失敗した場合、呼び出し元のトランザクションは影響されません。

TPNOREPLY
: tppost() が戻る前にイベント・ブローカが *eventname* に対するすべてのサブスクリプションを処理するのを待たないように、tppost に通知します。TPNOREPLY がセットされると、tppost() が成功して戻ったかどうかにかかわらず、tpurcode() はゼロに設定されます。呼び出しプロセスがトランザクション・モードにあるとき、TPNOTRAN が設定されない限りこの設定は使用できません。

TPNOBLOCK
: ブロッキング条件が存在する場合は、イベントはポストされません。このような条件が発生すると、呼び出しは失敗し、tperrno() には TPEBLOCK が設定されます。TPNOBLOCK が指定されていないときにブロッキング条件が存在すると、呼び出し元は、その条件が解消されるか、またはタイムアウト (トランザクションまたはブロッキング) が発生するまではブロックされます。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。トランザクション・タイムアウトは依然として発生する可能性があります。

TPSIGRSTRT
: シグナルが関数内部のシステム・コールを中断すると、中断されたシステム・コールは出しなおされます。TPSIGRSTRT が指定されていない場合にシグナルがシステム・コールを中断させると、tppost() は失敗し、tperrno() には TPGOTSIG が設定されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tppost() の呼び出しを発行できません。

戻り値

tppost() から成功して戻ると、tpurcode() には *eventname* の代わりにイベント・ブローカによってディスパッチされるイベント通知の数が設定されています (つまり、*eventname* に対するイベント表現の評価が成功し、*data* に対するフィルタ・ルールの評価が成功したサブスクリプションへのポストです)。tperrno() が TPESVCFAIL に設定されて戻った場合は、tpurcode() には、*eventname* の代わりにイベント・ブローカによってディスパッチされたトランザクション以外のイベント通知の数が設定されています。

異常終了すると、tppost() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tppost() は tperrno() を次のいずれかの値に設定します (特に指定がない限り、障害は、呼び出し元のトランザクションに影響しません)。

[TPEINVAL]
無効な引数 (たとえば、*eventname* に NULL) が指定されました。

[TPENOENT]
BEA Tuxedo User イベント・ブローカにアクセスできません。

[TPETRAN]
呼び出し元はトランザクション・モードにあり、TPNOTRAN は設定されておらず、tppost() はトランザクションの伝播をサポートしないイベント・ブローカにコンタクトしました (つまり、TMUSREVT(5) はトランザクションをサポートする BEA Tuxedo ATMI システムのグループで稼動していません)。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、そのトランザクションは終了します。トランザクション・モードでなければ、ブロッキング・タイムアウトが発生し、TPNOBLOCK も TPNOTIME も指定されていませんでした。トランザクション・タイムアウトが発生した場合、新しく処理を開始しようとしても、トランザクションがアボートするまで TPETIME になり、正常に行えません。

[TPESVCFAIL]
イベント・ブローカが呼び出し元のトランザクションに関してサービス・ルーチンまたは安定記憶キューにトランザクション・イベントをポストする際に、エラーが発生しました。呼び出し元の現在のトランザクションはアボートのみとマークされています。このエラーが返された場合、tpurcode() には、*eventname* の代わりにイベント・ブローカがディスパッチするトランザクション以外のイベント通知の数が設定されています。トランザクションのポスティングは、トランザクションの完了と共にその効果がなくなるため、カウントされません。トランザクションがタイムアウトしていないかぎり、通信はトランザクションがアボートするまで継続でき、また呼び出し元のトランザクションの一部として行った作業はすべてそのトランザクションの完了時にアボートされます (つまり、以後のやりとりで何らかの結果が得られる場合には、TPNOTRAN を設定しておくようにします)。

[TPEBLOCK]
ブロッキング状態のため、TPNOBLOCK が指定されました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tppost() が、不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目 tpsubscribe(3c)、tpunsubscribe(3c)、EVENTS(5)、TMSYSEVT(5)、TMUSREVT(5)

tprealloc(3c)

名前

tprealloc()— 型付きバッファのサイズを変更するルーチン

形式

```
#include <atmi.h>
char * tprealloc(char *ptr, long size)
```

機能説明

tprealloc() は、*ptr* が指すバッファのサイズを *size* バイトに変更し、新しい (おそらく移動した) バッファを指すポインタを返します。tpalloc() と同様、割り当てるバッファは少なくとも *size* および dfltsize と同じ大きさになります。ここで、dfltsize は特定のバッファ・タイプの tmtype_sw に指定されたデフォルトのバッファ・サイズです。この 2 つの値の大きい方のサイズが 0 またはそれ以下であると、このバッファは変更されず、NULL が返されます。バッファのタイプは再割り当て後も同じままです。この関数が正常に終了した場合、返されたポインタは、バッファを参照するために使用します。以後、バッファの参照に *ptr* を使用しないようにしてください。バッファの内容は新しいサイズと古いサイズで短い方の長さまでは変更されません。

ある種のバッファ・タイプは、使用する前に初期化を行っておく必要があります。tprealloc() は、バッファを再割り当てした後、(通信マネージャ固有の方法で) バッファを再度初期化します。このため、呼び出し元から返されるバッファはすぐに使用できます。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tprealloc() の呼び出しを発行できます。

戻り値

tprealloc() は正常終了すると、long ワードに境界を合わせた、適切なタイプのバッファへのポインタを返します。

異常終了すると、tprealloc() はヌルを返し、tperrno() を設定してエラー条件を示します。

エラー

再初期化関数が正常に実行できなかった場合、tprealloc() は異常終了して NULL を返し、*ptr* が指すバッファの内容は無効になってしまう可能性があります。異常終了時には、tprealloc() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
: 無効な引数が与えられた場合 (たとえば、*ptr* が、もともと tpalloc() によって割り当てられたバッファを指していない場合など)。

[TPEPROTO]
: tprealloc() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

使用法

バッファの再初期化が正常に実行できなかった場合、tprealloc() は異常終了して、NULL を返し、*ptr* が指すバッファの内容は無効になってしまう可能性があります。この関数は、C ライブラリの malloc()、realloc()、または free() とともに使用することはできません (たとえば、tprealloc() で割り当てたバッファは free() で解放することはできません)。

関連項目

tpalloc(3c)、tpfree(3c)、tptypes(3c)

tprecv(3c)

名前

tprecv()— 会話型接続でメッセージを受信するルーチン

形式

```
#include <atmi.h>
int tprecv(int cd, char **data, long *len, long flags, long \
   *revent)
```

機能説明

tprecv() は、別のプログラムからオープン接続を介してデータを受け取るときに使用します。tprecv() の最初の引数 *cd* は、データを受け取るオープン接続を指定します。*cd* には、tpconnect() から返される記述子、あるいは会話サービスに渡される TPSVCINFO パラメータに含まれる記述子のいずれかを指定します。2 番目の引数 *data* は、tpalloc() によって以前に割り当てられたバッファを指すポインタのアドレスです。

data は、前に tpalloc() が割り当てたバッファへのポインタのアドレスでなければなりません。また、*len* は long 型 （tprecv() が受信したデータのサイズに設定する）を指さなければなりません。**data* は応答を含んでいるバッファを指し、**len* は、バッファのサイズを含みます。FML と FML32 バッファは、通常最小サイズ 4096 バイトを確保します。したがって、応答が 4096 バイトより大きい場合は、バッファ・サイズは返されるデータを入れるのに十分な大きさに拡大します。

容量まで満たされていない送信側のバッファ（例えば、FML および STRING バッファ）は、送信に使用された大きさになります。システムは、受信データのサイズを任意の量で拡大します。これは、受信者が送信者の割り当てたバッファ・サイズより小さく、送信されたデータのサイズより大きいバッファを受け取ることを意味します。

受信バッファのサイズは、増加することも減少することもあります。また、アドレスもシステムがバッファを内部で交換するごとに常に変更されます。応答バッファのサイズが変わったどうか （また変わったとしたらどれくらい変わったのか）を決定するには、tprecv() が **len* とともに発行される前に、合計サイズを比べてください。バッファ管理の詳細については、「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」を参照してください。

len* が 0 の場合、データは受け取られず、data* も、それが指すバッファも、変更されていません。*data*、**data* または *len* が NULL であると、エラーになります。

tprecv() は、接続の制御をもたないプログラムしか出せません。

次に、有効な *flags* の一覧を示します。

TPNOCHANGE
: デフォルトでは、*data が指すバッファ型と異なるバッファ型を受信すると、*data のバッファ型は受信したバッファ型に変更されます (受信プロセスがそのバッファ型を認識できる場合)。このフラグが設定されていると、*data が指すバッファのタイプは変更されません。すなわち、受信したバッファのタイプとサブタイプは、*data が指すバッファのタイプとサブタイプに一致していなければなりません。

TPNOBLOCK
: tprecv() はデータが到着するまで待機しません。すでにデータが受信できる状態であると、tprecv() はデータを取り込んで終了します。このフラグが設定されておらず、かつ受信できるデータがない場合、呼び出し元はデータが到着するまでブロックされます。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。ただし、トランザクション・タイムアウトはあいかわらず有効です。

TPSIGRSTRT
: シグナルが関数内部の受信システム・コールを中断させると、呼び出しが再度出されます。

  記述子 *cd* に対してイベントが存在すると、tprecv() は終了し、tperrno() を TPEEVENT に設定します。イベントのタイプが *revent* で返されます。TPEV_SVCSUCC、TPEV_SVCFAIL および TPEV_SENDONLY イベントとともに、データを受け取ることができます。tprecv() の有効なイベントを次に示します。

TPEV_DISCONIMM
: 会話の従属側が受け取るこのイベントは、その会話の起動元が tpdiscon() により即時切断要求を出したこと、あるいは接続をオープンにしたままで tpreturn()、tpcommit()、または tpabort() を出したことを示します。このイベントは、通信エラー (サーバ、マシン、ネットワークの障害など) により接続が切断されたときにも起動元またはその従属側に返されます。これは即時切断通知 (つまり、正常ではなくアボート) であるため、処理途中のデータは失われます。2 つのプログラムが同じトランザクションに参加していた場合、そのトランザクションには「アボートのみ」のマークが付けられます。この接続に使用された記述子は無効になります。

TPEV_SENDONLY
: 接続の他方の側にあるプログラムは、接続の制御を放棄しました。このイベントの受信側はデータを送信することはできますが、制御を放棄するまではデータを受信することはできません。

TPEV_SVCERR
: このイベントは、会話の起動元が受け取るもので、会話の従属側が tpreturn() を出したことを示します。tpreturn() に、サービスが正しく応答を返すことができないようなエラーが発生しています。たとえば、不正な引数が tpreturn() に渡されていたり、tpreturn() が、そのサービスが別の従属側にオープン接続を持っている最中に呼び出されている可能性があります。このイベントの性質上、アプリケーションが定義したデータや戻りコードは返されません。この接続は切断され、記述子は無効になります。このイベントが受信側のトランザクションの一部として発生した場合は、トランザクションには「アボートのみ」のマークがつけられます。

TPEV_SVCFAIL
: このイベントは、接続の起動元が受け取るもので、会話の他方の側の従属サービスが、アプリケーションで定義されているように正常に終了しなかったことを示します (つまり、TPFAIL または TPEXIT とともに tpreturn() が呼び出されています)。従属サービスは、tpreturn() が呼び出されたときにこの接続の制御下にあった場合は、アプリケーション定義の戻りコードおよび型付きバッファを接続の要求元に返すことができます。サービス・ルーチンの終了処理の一部として、サーバはこの接続を切断しています。このため、*cd* は無効な記述子となります。このイベントが受信側のトランザクションの一部として発生した場合は、トランザクションには「アボートのみ」のマークが付けられます。

TPEV_SVCSUCC
: このイベントは、会話の起動元が受け取るもので、接続の他方の側の従属サービスがアプリケーションで定義されているように正常に終了したことを示します (すなわち、TPSUCCESS とともに tpreturn() が呼び出されています)。サービス・ルーチンの終了処理の一部として、サーバはこの接続を切断しています。このため、*cd* は無効な記述子となります。受信側がトランザクション・モードである場合、そのトランザクションをコミット (それが起動元でもある場合) するか、アボートして、サーバ (トランザクション・モードである場合) が行った作業内容をコミットあるいはアボートさせます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tprecv() の呼び出しを発行できません。

戻り値

revent が TPEV_SVCSUCC または TPEV_SVCFAIL のどちらかに設定されている場合に、tprecv() が終了した場合は、グローバル変数 tpurcode には、tpreturn() の一部として送信されるアプリケーション定義の値が含まれます。

異常終了すると、tprecv() は -1 を返し、tperrno() を設定してエラーの条件を示します。呼び出しが異常終了して tperrno に特定の値が設定されたときは、中間の ATMI 呼び出しを省略して引き続き tperrordetail() を呼び出すと、エラーに関する詳細な情報が提供されます。詳細については、tperrordetail(3c) リファレンス・ページを参照してください。

エラー

異常終了時には、tprecv() は tperrno を次のいずれかの値に設定します。

[TPEINVAL]
無効な引数が与えられました (たとえば、data が tpalloc() によって割り当てられたバッファを指すポインタのアドレスでない、あるいは *flags* が無効である場合など)。

[TPEOTYPE]
受け取るバッファのタイプおよびサブタイプのどちらも呼び出し元に通知されていません。または TPNOCHANGE が *flags* に設定されていて、**data* のタイプおよびサブタイプが受け取るバッファのタイプおよびサブタイプと合っていません。**data* の内容も **len* も変更されていません。会話が呼び出し元の現在のトランザクションの一部になっている場合は、受け取るバッファが放棄されるため、トランザクションには「アボートのみ」のマークが付けられます。

[TPEBADDESC]
cd が無効です。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、そのトランザクションは「アボートのみ」とマークされます。トランザクション・モードでなければ、ブロッキング・タイムアウトが発生し、TPNOBLOCK も TPNOTIME も指定されていませんでした。いずれのケースも、**data* とその内容はともに変更されません。トランザクション・タイムアウトが発生すると、トランザクションがアボートされない限り、接続を使ったメッセージの送受信や新しい接続の開始はできません。これらの操作を行おうとすると、TPETIME が発生して失敗します。

[TPEEVENT]
イベントが発生し、そのタイプが revent に記録されます。[TPETIME] と、[TPEEVENT] リターン・コードには関連があります。トランザクション・モードにおいては、会話の受信側が tprecv にブロックされていて、送信側が tpabort() を出した場合、受信側は [TPEVENT] リターン・コードを TPEV_DISCONIMM のイベントと共に取得します。ただし、受信側が tprecv() を出す前に、送信側が tpabort() を出した場合は、そのトランザクションはすでに GTT から削除されてしまっているので、tprecv() は異常終了し、[TPETIME] コードが返されます。

[TPEBLOCK]
ブロッキング条件が存在し、TPNOBLOCK が指定されていました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tprecv() が不正なコンテキストで呼び出されました (たとえば、呼び出しプログラムがデータの送信のみを行えるよう接続が確立されている場合など)。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

使用法

サーバは、tpreturn() を呼び出すとき、アプリケーション定義の戻りコードおよび型付きバッファを渡すことができます。戻りコードはグローバル変数 *tpurcode* で使用され、バッファは *data* で使用されます。

関連項目

tpalloc(3c)、tpconnect(3c)、tpdiscon(3c)、tperrordetail(3c)、tpsend(3c)、tpservice(3c)、tpstrerrordetail(3c)

tpresume(3c)

名前

tpresume()— グローバル・トランザクションの再開

形式

```
#include <atmi.h>
int tpresume(TPTRANID *tranid, long flags)
```

機能説明

tpresume() を使用して、中断されているトランザクションでの作業を再開します。呼び出し元がトランザクションの作業を再開した場合、その作業は tpsuspend() で再度停止させるか、あるいはあとで tpcommit() または tpabort() を利用して完了させる必要があります。

トランザクションの作業を再開する際には、呼び出し元はリンクされたリソース・マネージャが (tpopen() を利用して) オープンされていることを確認する必要があります。

tpresume() は、*tranid* でポイントされるグローバル・トランザクション識別子により呼び出し元をトランザクション・モードにします。*tranid* が NULL の場合はエラーです。

flags は将来使用するために予約されており、0 に設定します。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpresume() の呼び出しを発行できません。

戻り値

tpresume() は、エラーが発生すると -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

次の条件の場合、tpresume() は異常終了し、tperrno() を次の値に設定します。

[TPEINVAL]
tranid が NULL ポインタか、または存在しないトランザクション識別子 (前に完了しているトランザクションやタイムアウトしたトランザクションを含む) を指しているか、あるいは呼び出し元が再開することを許可されていないトランザクション識別子を指しています。トランザクションについての呼び出し元の状態は変化しません。

[TPEMATCH]
tranid が、他のプロセスが既に再開したトランザクション識別子を指しています。トランザクションについての呼び出し元の状態は変化しません。

[TPETRAN]
呼び出し元が、1 つまたは複数のリソース・マネージャでグローバル・トランザクション外の作業に関与しているため、BEA Tuxedo システムはグローバル・トランザクションを再開できません。そのような作業は全て、グローバル・トランザクションを再開する前に完了していなければなりません。トランザクションについての呼び出し元の状態は変化しません。

[TPEPROTO]
tpresume() が不正なコンテキストで呼び出されました (例えば、呼び出し元が既にトランザクション・モードになっている)。トランザクションについての呼び出し元の状態は変化しません。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

注意

XA 準拠のリソース・マネージャがグローバル・トランザクションに含まれるようにするには、そのリソース・マネージャが正常にオープンされている必要があります (詳細は tpopen(3c) を参照)。

中断したトランザクションを再開するプロセスは、トランザクションを中断したプロセスと同じ論理マシン (LMID) 上に存在しなければなりません。ワークステーション・クライアントでは、それが接続されるワークステーション・ハンドラ (WSH) が、トランザクションを中断したワークステーション・クライアントのハンドラと同じ論理マシン上に存在しなければなりません。

関連項目

tpabort(3c)、tpcommit(3c)、tpopen(3c)、tpsuspend(3c)

tpreturn(3c)

名前

tpreturn()—BEA Tuxedo ATMI システム・サービス・ルーチンからの復帰

形式

```
void tpreturn(int rval, long rcode、char *data、long len、long \
  flags)
```

機能説明

tpreturn() は、サービス・ルーチンが完了したことを示します。tpreturn() の働きは、C 言語における return 文と似ています (つまり、tpreturn() が呼び出されると、サービス・ルーチンは BEA Tuxedo ATMI システムのディスパッチャに制御を返します)。制御を BEA Tuxedo ATMI システムのディスパッチャに確実に正しく戻すために、ディスパッチされたサービス・ルーチン内から tpreturn() を呼ぶようにしてください。

tpreturn() はサービスの応答メッセージを送るときに使用します。応答を受け取るプログラムが tpcall()、tpgetrply() または tprecv() で待機している場合、tpreturn() の呼び出しが成功した時点で、受信側のバッファに応答が入ります。

会話型サービスの場合、tpreturn() は接続自体も切断します。したがって、サービス・ルーチンは、tpdiscon() を直接呼び出すことができません。正常な結果を保証するためには、会話型サービスに接続しているプログラムは tpdiscon() を呼び出してはなりません。そのようなプログラムは、会話型サービスが完了したという通知を待たなければなりません (たとえば、そのプログラムは、tpreturn() によって送信される TPEV_SVCSUCC または TPEV_SVCFAIL などのイベントのちの 1 つを待つ必要があります)。

また、サービス・ルーチンがトランザクション・モードにあった場合には、tpreturn() はトランザクションのサービス部分を、そのトランザクションの完了時点でコミットあるいはアボートできる状態にします。サービスは同じトランザクションの一部として複数回呼び出すことができるので、tpcommit() あるいは tpabort() がそのトランザクションのイニシエータによって呼び出されるまでは、完全にコミットあるいはアボートさせる必要は必ずしもありません。

tpreturn() は、該当サービス・ルーチンが出したサービス要求から期待されるすべての応答を受け取った後、呼び出すようにしてください。そうしない場合は、サービスの性質によって決まりますが、TPESVCERR 状態または TPEV_SVCERR イベントのいずれかが、サービス・ルーチンとの接続を開始したプログラムに返ります。受け取られなかった応答は、受信処理の後、通信マネージャによって自動的に取り除かれます。また、これらの応答に対応する記述子は無効になります。

tpreturn( ) は、サービスが出したすべての接続をクローズした後、呼び出される必要があります。そうしない場合はサービスの性質によって決まりますが、TPESVCERR または TPEV_SVCERR イベントのいずれかが、サービス・ルーチンとの通信を開始したプログラムに返されます。また、即時切断イベント (つまり、TPEV_DISCONIMM) が、オープンしているすべての接続を通して従属側に送信されます。

会話型サービスは、自分で開始しなかったオープン接続を 1 つだけ備えているため、通信マネージャは、送信すべき記述子データ (およびイベント) を認識しています。このため、記述子は、tpreturn( ) に渡されません。

次に、tpreturn( ) の引数について説明します。*rval* は次のいずれかに設定できます。

TPSUCCESS
: サービスは正常に終了しました。データが存在する場合、そのデータは送られます (戻り処理の失敗の場合を除き)。呼び出し元がトランザクション・モードにある場合には、tpreturn( ) は、そのトランザクションの呼び出し元の部分を、トランザクションが最終的にコミットすべき時点でコミットできるような状態にします。なお、tpreturn( ) を呼び出しても、必ずしもトランザクション全体が終了することにはつながりません。また、呼び出し元が正常終了を示したとしても、未終了の応答またはオープン接続がある場合、該当サービス内で行われた作業が原因でトランザクションが「ロールバックのみ」のマークを付けられた場合、異常終了メッセージが送られます (つまり、応答の受信側は TPESVCERR 指示あるいは TPEV_SVCERR イベントを受け取ります)。なお、サービス・ルーチンの処理中に何らかの理由でトランザクションに「ロールバックのみ」のマークが付けられると、*rval* が TPFAIL に設定されます。TPSUCCESS が会話型サービスに対して指定されると、TPEV_SVCSUCC イベントが生成されます。

TPFAIL
: アプリケーション側から見て、サービスが異常終了しました。応答を受け取ったプログラムにエラーが報告されます。つまり、応答を受け取る呼び出しは異常終了し、受信側は TPSVCFAIL 指示あるいは TPEV_SVCFAIL イベントを受け取ります。呼び出し元がトランザクション・モードであると、tpreturn( ) はそのトランザクションに「ロールバックのみ」のマークを付けます (ただし、そのトランザクションには、すでに「ロールバックのみ」のマークが付いている場合もあります)。戻り処理が失敗した場合を除き、呼び出し元のデータが送られます (もしあれば)。トランザクション・タイムアウトになって、呼び出し元のデータが送られない場合もあります。このケースでは、応答を待つプログラムは TPETIME エラーを受け取ることになります。会話型サービスに TPFAIL が指定された場合には、TPEV_SVCFAIL イベントが生成されます。

TPEXIT
: この値は、サービスの完了という点では TPFAIL と同じように動作しますが、TPEXIT が返されると、サーバは、トランザクションがロールバックして応答が要求元に返された後に終了します。
マルチスレッド・プロセスにこの値が指定された場合、TPEXIT は (そのプロセス内の単一のスレッドではなく) プロセス全体が強制終了されることを示します。
サーバが再開可能な場合は、自動的に再開します。

rval がこれら 3 つの値のいずれかに設定されていない場合、デフォルトの値として TPFAIL が使用されます。

アプリケーションが定義した戻りコード *rcode* をサービスの応答を受け取るプログラムに送ることもできます。このコードは、応答を正常に送ることができさえすれば (つまり、受信呼び出しが正常に行われる、あるいは TPESVCFAIL が返されれば)、*rval* の設定には関係なく送ることができます。さらに、会話型サービスでは、このコードは、サービス・ルーチンが tpreturn() を発行したときに接続の制御権を持っている場合のみ送信されます。*rcode* の値は、受信側の変数 tpurcode() に入ります。

data は送信される応答のデータ部を指すポインタです。*data* が NULL でなければ、以前に tpalloc() の呼び出しによって得られたバッファを指していなければなりません。このバッファが、サービス・ルーチン起動時にサービス・ルーチンに渡されたバッファと同じバッファである場合、そのバッファの配置は、BEA Tuxedo ATMI システムのディスパッチャに一任されます。したがって、サービス・ルーチンをコーディングする人は、バッファが開放されているかどうかを気にする必要はありません。実際、ユーザがこのバッファを解放しようとしてもできません。ただし、tpreturn() に渡されたバッファが、そのサービスが呼び出されたときのものと異なる場合には、tpreturn() でそのバッファを解放することができます。メイン・バッファが解放されても、そのバッファ内に埋め込まれたフィールドが参照するバッファは解放されません。*len* は送信するデータ・バッファの大きさを指定します。*data* が長さの指定を必要としないバッファを指すポインタである場合 (FML フィールド化バッファなど)、*len* は 0 でもかまいません。

data が NULL の場合、*len* は無視されます。この場合、サービスを起動したプログラムが応答を待っている状態にあると、データなしの応答が返されます。応答を待たない状態にあると、tpreturn() は、必要に応じて *data* を解放し、応答を送信しないで復帰します。

現在、*flags* は将来の用途のために予約されており、0 に設定します (0 以外の値に設定すると、応答の受信者は TPESVCERR または TPEV_SVCERR イベントを受信します)。

会話型サービスの場合、アプリケーションの戻りコードとデータ部が送られないケースがいくつかあります。

- 呼び出しが行われたときに接続がすでに切断されていた場合 (呼び出し元がその接続上で TPEV_DISCONIMM を受け取っていた場合)、この呼び出しは単にサービス・ルーチンを終了させ、現在のトランザクション (もしあれば) をアボートさせます。
- 呼び出し元が接続の制御権をもたない場合、上記のように、TPEV_SVCFAIL または TPEV_SVCERR が接続の要求元に送られます。発信元が受信したイベントの種類にかかわらず、データは送信されません。ただし、発信元が TPEV_SVCFAIL イベントを受信した場合は、発信元の tpurcode() 変数にリターン・コードが設定されます。

戻り値

サービス・ルーチンは呼び出し元である BEA Tuxedo ATMI システムのディスパッチャに値を返しません。このため、このルーチンは void として宣言されます。しかし、サービス・ルーチンは tpreturn() または tpforward() を使用して終了させるようになっています。会話型サービス・ルーチンの場合、tpreturn() を使用しなければならず、tpforward() を使用することはできません。サービス・ルーチンを tpreturn() または tpforward() のいずれをも使用せずに終了させる場合 (すなわち、このルーチンが C 言語の return 文を使用するか、ごく単純に関数の実行に失敗した場合)、あるいは tpforward() が会話型サーバから呼び出された場合、そのサーバはログに警告メッセージを出し、サービス・エラーをサービスの要求元に返します。従属側へのすべてのオープン接続は、ただちに切断され、未終了の非同期応答はドロップされます。障害時にサーバがトランザクション・モードであった場合、このトランザクションには「ロールバックのみ」のマークが付けられます。tpreturn() や tpforward() がサービス・ルーチンの外部から使用される場合 (たとえば、クライアントや tpsvrinit() あるいは tpsvrdone() で)、これらのルーチンは単に終了するだけです。

エラー

tpreturn() はサービス・ルーチンを終了させるので、引数処理またはサービス処理の間に発生したエラーについて、関数の呼び出し元に示すことはできません。このようなエラーが起こると、tpcall() または tpgetrply() を使用してサービスの結果を受信するプログラムのために tperrno() が [TPESVCERR] に設定されます。またイベント TPEV_SVCERR が、tpsend() または tprecv() を使用するプログラムに、会話を通して送信されます。

UBBCONFIG ファイル中の SVCTIMEOUT か、TM_MIB 中の TA_SVCTIMEOUT が 0 でない場合にサービスのタイムアウトが発生すると、TPEV_SVCERR が返されます。

tperrordetail() と tpstrerrordetail() を使用すると、現在のスレッドの中で呼び出された最新の BEA Tuxedo ATMI システムのルーチンが出したエラーに関して、追加的な情報を取得できます。エラーが起きると、tperrordetail() は数値を返しますが、この数値を trstrerrordetail() への引数として利用することで、エラーの詳細に関するテキストを受け取ることができます。

関連項目　　tpalloc(3c)、tpcall(3c)、tpconnect(3c)、tpforward(3c)、tprecv(3c)、tpsend(3c)、tpservice(3c)

tpscmt(3c)

名前

tpscmt()—tpcommit() がいつ戻るかを設定するルーチン

形式

```
#include <atmi.h>
int tpscmt(long flags)
```

機能説明

tpscmt() は、*flags* で指定した値を TP_COMMIT_CONTROL 特性に設定します。TP_COMMIT_CONTROL 特性は、tpcommit() の呼び出し元に制御を戻すことに関して、tpcommit() の動作に影響を与えます。プログラムがトランザクション・モードにあるかどうかに関係なく、プログラムから tpscmt() を呼び出すことができます。他のプログラムがコミットしなければならないトランザクションに呼び出し元が参加している場合は、tpscmt() を呼び出してもそのトランザクションに影響を与えないことに注意してください。むしろ、呼び出し元がコミットするその後のトランザクションに影響を与えます。

ほとんどの場合、BEA Tuxedo ATMI システムのスレッドの制御が tpcommit() を呼び出す場合にのみ、トランザクションがコミットされます。ただし、例外が 1 つあります。UBBCONFIG ファイルの *SERVICES セクションの AUTOTRAN 変数が有効になっているためサービスがトランザクション・モードでディスパッチされる場合、トランザクションは tpreturn() を呼び出して戻ります。tpforward() が呼び出されると、最終的にサーバが tpreturn() を呼び出すことでトランザクションが完了します。このように、tpreturn() を呼び出すサービスの TP_COMMIT_CONTROL 属性の設定によって、サーバ中で tpcommit() からいつ制御が戻るかが決まります。tpcommit() がヒューリスティックなエラー・コードを返した場合、サーバはメッセージをログ・ファイルに書き込みます。

クランアントが BEA Tuxedo ATMI システムのアプリケーションに参加する場合は、この特性の初期設定はコンフィギュレーション・ファイルから取られます (UBBCONFIG(5) の *RESOURCES セクションの CMTRET 変数の項を参照)。

flags に設定できる有効な値を次に示します。

TP_CMT_LOGGED
: このフラグは、2 フェーズ・コミット・プロトコルの第 1 フェーズによって第 2 フェーズの前にコミット決定が記録された後、tpcommit() から返ることを指定します。この設定は、tpcommit() の呼び出し元に対するより速い反応を見込んでいます。だだし、第 2 フェーズの完了を待つ時間的な遅延のため、トランザクションのパーティシパントが処理をヒューリスティックに完了する (すなわち、異常終了を示します) と決めるかもしれないという危険が存在します。この場合は、tpcommit() はすでに戻っているので、これを呼び出し元に伝える方法はありません (ただし、リソース・マネージャがヒューリスティックな設定を行うと、BEA Tuxedo システムはメッセージをログ・ファイルに書き込みます)。正常な状態では、第 1 フェーズの間にコミットすることを約束しているパーティシパントは、第 2 フェーズでコミットします。通常、ネットワークまたはサイトの障害による問題は、第 2 フェーズの間にヒューリスティックな決定が行われる原因になります。

TP_CMT_COMPLETE
: このフラグは、2 フェーズ・コミット・プロトコルが完全に終了してから tpcommit(3c) が終了することを指定します。この設定により、tpcommit() は第 2 フェーズのコミット中にヒューリスティックな判断がなされたことを示すことができます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tpscmt() の呼び出しを発行できません。

戻り値

成功の場合、tpscmt() は TP_COMMIT_CONTROL 特性の以前の値を返します。

異常終了すると、tpscmt() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpscmt() は tperrno を次のいずれかの値に設定します

[TPEINVAL]
: *flags* は、TP_CMT_LOGGED または TP_CMT_COMPLETE のいずれかではありません。

[TPEPROTO]
: tpscmt() が不正に呼び出されました。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

注意事項

BEA Tuxedo ATMI システムのトランザクションを記述するために tpbegin()、tpcommit()、および tpabort() を使用する場合、XA インターフェイスに準拠した (呼び出し元に妥当にリンクされている) リソース・マネージャが行う作業のみがトランザクションの特性を備えていることを記憶しておくことが重要です。トランザクションにおいて実行される他のすべての操作は、tpcommit() あるいは tpabort() のいずれにも影響されません。詳細については buildserver(1) を参照してください。

関連項目

tpabort(3c)、tpbegin(3c)、tpcommit(3c)、tpgetlev(3c)

tpseal(3c)

名前

tpseal()— 暗号化する型付きメッセージ・バッファのマーク

形式

```
#include <atmi.h>
int tpseal(char *data, TPKEY hKey, long flags)
```

機能説明

tpseal() は、暗号化するメッセージ・バッファをマーク (登録) します。*hKey* を所有するプリンシパルは、このバッファを解読し、その内容にアクセスすることができます。tpseal() を何度か呼び出すことによって、複数の受信者のプリンシパルに 1 つのバッファを指定できます。

data は、(1) 以前 tpalloc() を呼び出すプロセスによって割り当てられたメッセージ・バッファ、または (2) システムによって受信プロセスに渡されたメッセージ・バッファのうち、いずれかの有効な型付きメッセージ・バッファを指している必要があります。バッファの内容は、tpseal() を呼び出した後で修正することができます。

data が指すメッセージ・バッファがプロセスから伝送されると、公開鍵ソフトウェアがメッセージ内容を暗号化し、各暗号化登録要求のメッセージ・バッファに暗号化エンベロープをアタッチします。暗号化エンベロープによって、受信プロセスはメッセージを解読することができます。

引数 *flags* は使用されません。この引数は将来の用途のために予約されており、0 (ゼロ) に設定します。

戻り値

異常終了すると、この関数は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

[TPEINVAL]
: 無効な引数が指定されました。たとえば、*hKey* が暗号化に有効なキーでないか、または *data* がヌルです。

[TPESYSTEM]
: エラーが発生しました。詳細については、システム・エラー・ログ・ファイルを参照してください。

関連項目

tpkey_close(3c)、tpkey_open(3c)

tpsend(3c)

名前

tpsend()— 会話型接続でメッセージを送信するルーチン

形式

```
#include <atmi.h>
int tpsend(int cd、char *data、long len、long flags、long *revent)
```

機能説明

tpsend() は、別のプログラムにオープン接続を介してデータを送信するときに使用します。このとき、呼び出し元がこの接続を制御できなければなりません。tpsend() の最初の引数 *cd* は、データを送信するオープン接続を指定するものです。*cd* には、tpconnect() から返される記述子、あるいは会話サービスに渡される TPSVCINFO パラメータに含まれる記述子のいずれかを指定します。

2 番目の引数 *data* は、tpalloc() によって以前に割り当てられたバッファを指していなければなりません。*len* には送信バッファの大きさを指定します。ただし、*data* が長さの指定を必要としないバッファを指している場合 (FML フィールド化バッファなど)、*len* は無視されます (0 でかまいません)。また、*data* は NULL でもかまいません。この場合、*len* は無視されます (アプリケーション・データは送信されません。これはデータを送信せず、たとえば接続の制御だけを与えるときに使用されます)。*data* のタイプとサブタイプは、接続の他方の側が認識するタイプおよびサブタイプと一致していなければなりません。

次に、有効な *flags* の一覧を示します。

TPRECVONLY
: このフラグは、呼び出し元のデータが送信された後、呼び出し元が接続の制御を放棄することを指定します (つまり、呼び出し元はそれ以降、tpsend() 呼び出しを出すことはできなくなります)。接続の他方の側のプログラムが tpsend() から送られたデータを受け取る場合、接続の制御権を得たことを示すイベント (TPEV_SENDONLY) も受け取ります (そして、それ以上、tprecv() を呼び出すことができなくなります)。

TPNOBLOCK
: ブロッキング条件が存在する場合、データもどのようなイベントも送信されません (たとえば、メッセージの送信に使用される内部バッファがいっぱいのときなど)。TPNOBLOCK が指定されていないときにブロッキング条件が存在すると、呼び出し元は、その条件が解消されるか、またはタイムアウト (トランザクションまたはブロッキング) が発生するまではブロックされます。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。トランザクション・タイムアウトは依然として発生する可能性があります。

TPSIGRSTRT
: シグナルが関数内部のシステム・コールを中断すると、中断されたシステム・コールは出しなおされます。

記述子 *cd* にイベントが発生すると、tpsend() は正常終了できず、呼び出し元のデータは送信されません。イベントのタイプが *revent* で返されます。tpsend() の有効なイベントを次に示します。

TPEV_DISCONIMM
: 会話の従属側が受け取るこのイベントは、その会話の起動元が tpdiscon() により即時切断要求を出したことを、または接続をオープンにしたままで tpreturn() か、tpcommit() か、もしくは tpabort() を出したことを示します。このイベントは、通信エラー (サーバ、マシン、ネットワークの障害など) により接続が切断されたときにも起動元またはその従属側に返されます。

TPEV_SVCERR
: 会話の起動元が受け取るこのイベントは、その接続の従属側が会話の制御権をもたずに tpreturn() を出したことを示します。さらに、tpreturn() は、TPEV_SVCFAIL について後述している方法とは異なる方法で出されました。このイベントは ACL パーミッション違反によっても発生します。つまり、呼び出し元が受け取り先プロセスに接続するためのパーミッションをもっていないことを示します。このイベントは、tpconnect() が出されるのと同時にではなく、最初の tpsend() と共に (フラグ TPSENDONLY をもった tpconnect() に続いて) 返されるか、または最初の tprecv() と共に (フラグ TPRECVONLY をもった tpconnect() に続いて) 返されます。また、システム・イベントとログ・メッセージも生成されます。

TPEV_SVCFAIL
: 会話の起動元が受け取るこのイベントは、その接続の従属側が会話の制御権をもたずに tpreturn() を出したことを示します。さらに、tpreturn() は、*rval* として TPFAIL または TPEXIT を設定し、*data* として NULL を設定した状態で出されました。

これらのイベントはそれぞれ、即時切断通知 (すなわち、正常ではなくアボート) を示すので、処理途中のデータは失われます。この接続に使用された記述子は無効になります。2 つのプログラムが同じトランザクションに参加していた場合には、そのトランザクションに「アボートのみ」のマークが付けられます。

UBBCONFIG ファイル中の SVCTIMEOUT か、TM_MIB 中の TA_SVCTIMEOUT のどちらか一方が 0 でない場合にサービスのタイムアウトが発生すると、TPESVCERR が返されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tpsend() の呼び出しを発行できません。

戻り値

TPEV_SVCSUCC または TPEV_SVCFAIL のどららかが *revent* に設定されて tpsend() が戻った場合、tpurcode() によってポイントされているグローバル変数には、tpreturn() の一部として送信された、アプリケーションで定義した値が入っています。関数 tpsend() はエラー時には -1 を返し、tperrno() を設定してエラーの条件を示します。またイベントが存在し、かつエラーが発生しない場合、tpsend() は -1 を返し、tperrno() に [TPEEVENT] を設定します。

エラー

異常終了時には、tpsend() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
無効な引数が与えられました (たとえば、*data* が tpalloc() によって割り当てられたバッファを指していないか、*flags* が無効)。

[TPEBADDESC]
cd が無効です。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、そのトランザクションは「アボートのみ」とマークされます。トランザクション・モードにない場合は、ブロッキング・タイムアウトが発生しており、TPNOBLOCK も TPNOTIME も指定されていませんでした。いずれのケースでも、**data*、その内容、**len* はどれも変更されません。トランザクション・タイムアウトが発生すると、トランザクションがアボートされない限り、接続を使ったメッセージの送受信や新しい接続の開始はできません。これらの操作を行おうとすると、TPETIME が発生して失敗します。

[TPEEVENT]
イベントが発生しました。このエラーが発生すると、*data* は送られません。イベントのタイプが *revent* で返されます。

[TPEBLOCK]
ブロッキング条件が存在し、TPNOBLOCK が指定されていました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tpsend() が不正なコンテキストで呼び出されました (たとえば、呼び出し元がデータの受信しかできないように、接続が確立された場合など)。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目　　tpalloc(3c)、tpconnect(3c)、tpdiscon(3c)、tprecv(3c)、tpservice(3c)

tpservice(3c)

名前

tpservice()— サービス・ルーチンのテンプレート

形式

```
#include <atmi.h>                    /* C インターフェイス */
void tpservice(TPSVCINFO *svcinfo)  /* C++ インターフェイス -
                                     * C リンケージが必要 */
extern "C" void tpservice(TPSVCINFO *svcinfo)
```

機能説明

tpservice() は、サービス・ルーチン作成時のテンプレートです。このテンプレートは、ルーチン tpcall()、tpacall()、または tpforward() を介して要求を受け取るサービス、およびルーチン tpconnect()、tpsend()、または tprecv() を介して通信を行うサービスに使用できます。

tpcall() または tpacall() を介して行われる要求を処理するサービス・ルーチンは、1 つだけ着信メッセージを受け取り (*svcinfo* の *data* 要素内に)、1 つだけ応答を送る (tpreturn() を使用してサービス・ルーチンを終了するとき) ことができます。

一方、会話型サービスは、1 つの受信メッセージとオープン接続記述子を持つ接続要求により呼び出されます。会話型サービス・ルーチンが呼び出されると、接続元プログラムあるいは会話型サービスはアプリケーション側で定義されたようにデータの送信および受信を行うことができます。この接続は半二重方式で確立されます。つまり、一方は他方から明示的に制御権を渡されるまで、会話権を得られません (データを送信できません)。

トランザクションとの関連で言えば、サービス・ルーチンはトランザクション・モードで呼び出されると、1 つのトランザクションにしか参加できません。サービス・ルーチン作成者側から見るかぎり、トランザクションはサービス・ルーチンから返った時点で終了します。サービス・ルーチンは、トランザクション・モードで呼び出されなかった場合、tpbegin()、tpcommit() および tpabort() を使用して必要な回数だけトランザクションを起動できます。ただし、トランザクションの終了には tpreturn() を使用しません。したがって、サービス・ルーチン内から起動された未終了のトランザクションについて、tpreturn() を呼び出すと、エラーになります。

サービス・ルーチンは、サービス情報を収めた構造体を指すポインタ *svcinfo* を引数として呼び出されます。この構造体には、次のメンバが含まれます。

```
char        name[32];
char        *data;
long        len;
long        flags;
int         cd;
long        appkey;
CLIENTID    cltid;
```

name には、要求元がサービスの呼び出しに使用したサービス名を指定します。

サービス・ルーチンに入った時点で設定される *flags* は、サービス・ルーチンが注目する必要のある属性を示します。以下に、*flags* に指定できる値を示します。

TPCONV
: 会話型の接続要求が受け入れられたときに、その接続記述子が *cd* に設定されることを示します。このフラグが設定されていなければ、これは要求 / 応答型サービスであり、*cd* は無効です。

TPTRAN
: サービス・ルーチンはトランザクション・モードにあります。

TPNOREPLY
: 呼び出し元は応答を期待していません。このオプションは、TPCONV が設定されている場合には設定されません。

TPSENDONLY
: サービスは、接続を介してデータの送信のみ可能で、接続の他方の側のプロセスはデータの受信しかできないよう呼び出されます。このフラグは TPRECVONLY と相互に排他的で、TPCONV が設定されているときにしか設定できません。

TPRECVONLY
: サービスは、接続を介してデータの受信のみ可能で、接続の他方の側のプロセスはデータの送信しかできないよう呼び出されます。このフラグは TPSENDONLY と相互に排他的で、TPCONV が設定されているときにしか設定できません。

data は要求メッセージのデータ部を指し、*len* はデータの長さです。*data*data によって指されたバッファは、通信マネージャで tpalloc() により割り当てられたものです。このバッファのサイズは、ユーザが tprealloc() を使用して大きくすることができます。ただし、これをユーザが解放することはできません。このバッファは、サービス終了時に tpreturn() または tpforward() に渡すようにしてください。異なるバッファをこれらのルーチンに渡すと、そのバッファはそれらによって解放されてしまいます。なお、*data* によって指されるバッファは、このバッファが tpreturn() または tpforward() に渡されない場合でも、次に出されたサービス要求によって変更されてしまいます。*data* は、要求とともにデータが渡されなかった場合には NULL になります。この場合、*len* は 0 になります。

TPCONV を *flags* に設定する場合、*cd* に接続記述子を指定しますが、これはこの会話を開始したプログラムとのコミュニケーションを行うために tpsend() および tprecv() とともに使用します。

appkey は、アプリケーション側で定義した認証サービスが要求クライアントに割り当てるアプリケーション・キーに設定します。このキー値は、このサービス・ルーチンのこの呼び出し中になされたあらゆるサービス要求とともに渡されます。アプリケーション認証サービスを通らないクライアントを起動する場合には、*appkey* の値は -1 になります。

cltid は、このサービス要求に対応する元のクライアントを示す一意のクライアント識別子です。この構造体は atmi.h にのみ、アプリケーションが利用できるよう定義されているので、必要によりアプリケーション・サーバ間でクライアント識別子をやりとりすることができます。このため、以下に定義されているフィールドの意味は明記されていません。アプリケーション側では CLIENTID 構造体の内容を操作しないようにしてください。内容を操作してしまうと、この構造体自体が無効になってしまいます。CLIENTID 構造体には、次のようなメンバが含まれます。

```
long        clientdata[4];
```

C++ では、サービス関数に C リンケージが必要なことに注意してください。リンケージは、関数を 'extern "C"' と宣言することで行えます。

戻り値

サービス・ルーチンは呼び出し元である通信マネージャ・ディスパッチャに値を返しません。このため、このルーチンは void として宣言します。しかし、サービス・ルーチンは tpreturn() または tpforward() を使用して終了させるようになっています。会話サービス・ルーチンの場合、tpreturn() を使用しなければならず、tpforward() を使用することはできません。サービス・ルーチンを tpreturn() または tpforward() のいずれをも使用せずに終了させる場合 (すなわち、このルーチンが C 言語の return 文を使用するか、ごく単純に関数の実行に失敗した場合)、あるいは tpforward() が会話型サーバから呼び出された場合、そのサーバはログ・ファイルに警告メッセージを出し、サービス・エラーを要求元あるいはリクエスタに返します。従属側へのすべてのオープン接続は、ただちに切断され、未終了の非同期応答は「処理済み」のマークが付けられます。障害が発生したときにサーバがトランザクション・モードの場合、そのトランザクションは「アボートのみ」とマークされます。tpreturn() または tpforward() がサービス・ルーチンの範囲外で (たとえば、クライアントで、あるいは、tpsvrinit() または tpsvrdone() で) 使用された場合は、これらのルーチンは、一切影響を及ぼすことなく単に終了します。

エラー

tpreturn() はサービス・ルーチンを終了させるので、引数処理またはサービス処理の間に発生したエラーについて、関数の呼び出し元に示すことはできません。このようなエラーが発生した場合は、tpcall() または tpgetrply() を介してサービスから情報を受信するプログラムでは、tperrno() が TPESVCERR に設定され、tpsend() または tprecv() を使用するプログラムには、会話を通してイベント TPEV_SVCERR が送信されます。

関連項目　　tpalloc(3c)、tpbegin(3c)、tpcall(3c)、tpconnect(3c)、tpforward(3c)、tpreturn(3c)、servopts(5)

tpsetctxt(3c)

名前

tpsetctxt()— 現在のアプリケーション関連に対するコンテキスト識別子の設定

形式

```
#include <atmi.h>
int tpsetctxt(TPCONTEXT_T context, long flags)
```

機能説明

tpsetctxt() は、現在のスレッドが動作するコンテキストを定義します。この関数は、マルチスレッド環境ではスレッド単位で、非スレッド環境ではプロセス単位で動作します。

次にこのスレッドで BEA Tuxedo ATMI が呼び出されると、コンテキストが示すアプリケーションが参照されます。コンテキストは、同一プロセス内のスレッドで tpgetctxt() の呼び出しを発行することによって提供されます。*context* の値が TPNULLCONTEXT の場合、現在のスレッドはどの BEA Tuxedo ATMI コンテキストにも関連付けられません。

マルチコンテキスト・モードで動作しているプロセスの各スレッドでは、次の呼び出しを発行するによって、TPNULLCONTEXT 状態にすることができます。

```
tpsetctxt(TPNULLCONTEXT, 0)
```

TPINVALIDCONTEXT は、*context* に入力できる有効な値ではありません。

TPINVALIDCONTEXT 状態のスレッドは、ほとんどの ATMI 関数に対する呼び出しを発行できません (呼び出せる関数と呼び出せない関数のリストについては、「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」を参照してください)。このため、必要に応じてスレッドの TPINVALIDCONTEXT 状態を終了させる必要があります。それには、コンテキストを TPNULLCONTEXT か別の有効なコンテキストに設定して tpsetctxt() を呼び出します。また、tpterm() 関数を呼び出して TPINVALIDCONTEXT 状態を終了させることもできます。

2 番目の引数 *flags* は現在使用されていないので、0 に設定します。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tpsetctxt() の呼び出しを発行できます。

戻り値

tpsetctxt() は、正常終了時には 0 以外の値を返します。

異常終了すると、tpsetctxt() は呼び出しプロセスを元のコンテキストに維持したまま -1 を返し、tperrno を設定してエラー条件を示します。

エラー

異常終了時には、tpsetctxt() は tperrno を次のいずれかの値に設定します。

[TPEINVAL]
無効な引数が指定されました。たとえば、*flags* が 0 以外の値に設定された、またはコンテキストが TPINVALIDCONTEXT です。

[TPENOENT]
context の値が有効なコンテキストではありません。

[TPEPROTO]
tpsetctxt() が正しくないコンテキストで呼び出されました。たとえば、(a) サーバがディスパッチしたスレッドで呼び出された、(b) tpinit() を呼び出していないプロセスで呼び出された、(c) TPMULTICONTEXTS フラグを指定しないで tpinit() を呼び出したプロセスで呼び出された、または (d) TMNOTHREADS 環境変数がオンになっているプロセスで複数のスレッドから呼び出された、などです。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目

「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」、tpgetctxt(3c)

tpsetunsol(3c)

名前

tpsetunsol()— 任意通知型メッセージの処理方式の設定

形式

```
#include <atmi.h>
void (*tpsetunsol (void (_TMDLLENTRY *)(*disp) (char *data, long len,
long flags)))
(char *data, long len, long flags)
```

機能説明

tpsetunsol() は、任意通知型メッセージが BEA Tuxedo ATMI システムのライブラリによって受け取られる際に呼び出すルーチンをクライアントが指定できるようにします。tpsetunsol() の最初の呼び出しの前に、BEA Tuxedo ATMI システムのライブラリがクライアントのために受け取った任意通知型メッセージは記録されますが、無視されます。NULL 関数ポインタを使用する tpsetunsol() を呼び出した場合も、同じ結果になります。システムが通知や検出のために使用する方法は、アプリケーションのデフォルトの設定によって決まります (RESOURCES セクションの NOTIFY パラメータ)。このデフォルトの設定は、クライアントごとに変更できます (tpinit(3c) を参照)。

tpsetunsol() の呼び出し時に渡される関数ポインタは、所定のパラメータ定義に準拠していなければなりません。*data* は受け取った型付きバッファを指し、*len* はそのデータの長さを指定します。*flags* は現時点では使用されていません。*data* は、通知と一緒にデータが渡されない場合には NULL になります。*data* は、クライアントが認識しないタイプ / サブタイプのバッファであることがありますが、その場合、メッセージ・データは不明瞭になります。

data はアプリケーション・コードで解放することはできません。ただし、システムはこれを解放し、終了後、データ領域を無効にします。

アプリケーションの任意通知型メッセージ処理ルーチン内での処理は、次の BEA Tuxedo ATMI 関数に限定されています。tpalloc() tpfree() tpgetctxt() tpgetlev() tprealloc() tptypes()

マルチスレッド・プログラミング環境では、任意通知型メッセージ処理ルーチンが tpgetctxt() を呼び出して、別のスレッドを作成し、そのスレッドに適切なコンテキストの tpsetctxt() を呼び出し、新しいスレッドに、クライアントが使用できる ATM 関数をすべて使用させることができます。

tpsetunsol() がコンテキストに関連していないスレッドから呼び出されると、新しく生成されるすべての tpinit() コンテキストに対して、プロセスごとのデフォルトの任意通知型メッセージ・ハンドラが作成されます。これは、既にシステムに関連付けられているコンテキストには影響しません。特定のコンテキストは、コンテキストがアクティブのときに tpsetunsol() を再度呼び出して、そのコンテキストの任意通知型メッセージ・ハンドラを変更することができます。プロセスごとのデフォルトの任意通知型メッセージ・ハンドラは、コンテキストに現在関連していないスレッドで tpsetunsol() を再度呼び出して変更することができます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tpsetunsol() の呼び出しを発行できません。

戻り値

tpsetunsol() は、正常終了時には、任意通知型メッセージ処理ルーチンの以前の設定条件を返します (NULL も正常な戻り値の 1 つであり、メッセージ処理関数を事前に設定していなかったことを示します)。

異常終了すると、この関数は TPUNSOLERR を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpsetunsol() は tperrno() を次のいずれかの値に設定します。

[TPEPROTO]
tpsetunsol() が正しくないコンテキストで呼び出されました。たとえば、サーバ内部から呼び出されています。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

移植性

tpnotify(3c) で説明したインターフェイスはすべて、ネイティブ・サイトの UNIX システム・ベースおよび Windows のプロセッサ上で利用できます。さらに、ルーチン tpbroadcast() と tpchkunsol() は、関数 tpsetunsol() ともとに、UNIX システムおよび MS-DOS ベースのプロセッサ上で利用することができます。

関連項目

tpinit(3c)、tpterm(3c)

tpsign(3c)

名前

tpsign()— デジタル署名のための型付きメッセージ・バッファのマーク

形式

```
#include <atmi.h>
int tpsign(char *data, TPKEY hKey, long flags)
```

機能説明

tpsign() は、*hKey* に関連するプリンシパルに代わって、デジタル署名のためのメッセージ・バッファをマーク (登録) します。

data は、(1) 以前 tpalloc() を呼び出すプロセスによって割り当てられたメッセージ・バッファ、または (2) システムによって受信プロセスに渡されたメッセージ・バッファのうち、いずれかの有効な型付きメッセージ・バッファを指している必要があります。バッファの内容は、tpsign() を起動してから修正することができます。

data が指すバッファがプロセスから伝送されると、デジタル署名登録要求に対して、公開鍵ソフトウェアがデジタル署名を生成し、メッセージ・バッファにアタッチします。デジタル署名によって、受信プロセスはメッセージの署名者 (発信者) を検証することができます。

引数 *flags* は使用されません。この引数は将来の用途のために予約されており、0 (ゼロ) に設定します。

戻り値

異常終了すると、この関数は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

[TPEINVAL]
無効な引数が指定されました。たとえば、*hKey* が署名に有効なキーでない、または *data* の値がヌルです。

[TPESYSTEM]
エラーが発生しました。詳細については、システム・エラー・ログ・ファイルを参照してください。

関連項目

tpkey_close(3c)、tpkey_open(3c)

tpsprio(3c)

名前

tpsprio()— サービス要求の優先順位の設定

形式

```
#include <atmi.h>
int tpsprio(prio, flags)
```

機能説明

tpsprio() は、カレント・コンテキストのカレント・スレッドが、次に送信または転送する要求の優先順位を設定します。設定された優先順位は、次に送信される要求に対してのみ有効です。メッセージのキュー登録機能がインストールされている場合、優先順位を tpenqueue() や tpdequeue() によってキューへ登録、または削除されたメッセージに対しても設定することができます。デフォルトの設定では、*prio* の設定条件が正か負かにより、サービスのデフォルトの優先順位が最大 100、あるいは最小 1 に上下します。100 が最も高い優先順位です。要求のデフォルトの優先順位は、その要求の送信先となるサービスによって決まります。このデフォルトの設定は、構成時に指定してもよいですし (UBBCONFIG(5) を参照)、システムのデフォルト値 50 を使用してもかまいません。tpsprio() は、tpconnect() あるいは tpsend() を介して送られたメッセージには影響しません。

メッセージは、10 回に 1 回は FIFO 方式に基づいて取り出されるため、優先順位の低いメッセージがキューにいつまでも残されることはありません。優先度の低いインターフェイスやサービスでは、応答時間を問題にすべきではありません。

マルチスレッドのアプリケーションでは、tpsprio() はスレッド単位で動作します。

次に、有効なフラグの一覧を示します。

TPABSOLUTE
: 次の要求の優先順位は、*prio* の絶対値で送信されます。この絶対値は 1 から 100 までの範囲内の数値とします (最も高い優先順位は 100 です)。この範囲外の値を指定すると、デフォルトの値が使用されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tpsprio() の呼び出しを発行できません。

戻り値

異常終了すると、tpsprio() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpsprio() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
: *flags* が無効です。

[TPEPROTO]
: tpsprio() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目 tpacall(3c)、tpcall(3c)、tpdequeue(3c)、tpenqueue(3c)、tpgprio(3c)

tpstrerror(3c)

名前

tpstrerror()—BEA Tuxedo ATMI システムのエラー・メッセージ文字列の取得

形式

```
#include <atmi.h>
char *
tpstrerror(int err)
```

機能説明

tpstrerror() は LIBTUX_CAT からエラー・メッセージのテキストを取得するために使用します。*err* は、BEA Tuxedo ATMI システムの関数呼び出しが -1 またはその他の異常終了値を返した場合に tperrno() に設定されるエラー・コードです。

ユーザは、tpstrerror() から返されるポインタを、userlog() または fprintf() への引数として使用できます。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても tpstrerror() の呼び出しを発行できます。

戻り値

正常終了すると、tpstrerror() はエラー・メッセージ・テキストを含む文字列を指すポインタを返します。

err が無効なエラー・コードであった場合は、tpstrerror() は、NULL を返します。

エラー

異常終了すると、tpstrerror() はヌルを返しますが tperrno() は設定しません。

使用例

```
#include <atmi.h>
.
.
.
  char *p;
  if (tpbegin(10,0) == -1) {
    p = tpstrerror(tperrno);
    userlog("%s", p);
    (void)tpabort(0);
    (void)tpterm();
    exit(1);
  }
```

関連項目

userlog(3c)、Fstrerror、Fstrerror32(3fml)

tpstrerrordetail(3c)

名前

tpstrerrordetail()—BEA Tuxedo ATMI のエラーに関する詳細なメッセージ文字列の取得

形式

```
#include <atmi.h>
char * tpstrerrordetail(int err, long flags)
```

機能説明

tpstrerrordetail() は、Tuxedo ATMI エラーの詳細情報のテキストを取り出すのに使用します。*err* は、tperrordetail() が返す値です。

ユーザは、tpstrerrordetail() が返したポインタを userlog() または fprintf() に対する引数として使用できます。

flags は将来使用する予定であり、現在は必ずゼロを指定してください。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tpstrerrordetail() の呼び出しを発行できます。

戻り値

正常終了した場合は、この関数は、エラー・メッセージのテキストを持つ文字列を指すポインタを返します。

異常終了時 (すなわち *err* が無効なエラー・コードの場合)、tpstrerrordetail() はヌルを返します。

エラー

異常終了すると、tpstrerrordetail() はヌルを返し、tperrno() は設定しません。

使用例

```
#include <atmi.h> .  .  .
int ret;
char *p;

if (tpbegin(10,0) == -1) {
       ret = tperrordetail(0);
       if (ret == -1) {
              (void) fprintf(stderr, "tperrordetail() failed!\n");
              (void) fprintf(stderr, "tperrno = %d, %s\n",
                     tperrno, tpstrerror(tperrno));
       }

       else if (ret != 0) {
              (void) fprintf(stderr, "errordetail:%s\n",
                            tpstrerrordetail(ret, 0));
       }
       .
```

```
        .
        .
}
```

関連項目　「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」、tperrordetail(3c)、tpstrerror(3c)、userlog(3c)、tperrno(5)

tpsubscribe(3c)

名前　tpsubscribe()— イベントをサブスクライブする

形式

```
#include <atmi.h>
long tpsubscribe(char *eventexpr, char *filter, TPEVCTL *ctl, long
flags)
```

機能説明

呼び出し元は tpsubscribe() を使用して、*eventexpr* で示されるイベントまたはイベントの集合をサブスクライブします。サブスクリプションは、BEA Tuxedo ATMI イベント・ブローカ、TMUSREVT(5) によって保持され、イベントがポストされたときにサブスクライバに通知するために、tppost() によって使用されます。それぞれのサブスクリプションには、通知メソッドを指定します。通知メソッドは、クライアント通知、サービス呼び出し、または安定記憶域内のキューへのメッセージ登録の 3 つの内いずれかの形式をとります。通知メソッドはサブスクライバのプロセスのタイプと tpsubscribe() に渡された引数によって決まります。

サブスクライブするイベントまたはイベントの集合は、*eventexpr* で指定します。*eventexpr* には、最大で 255 文字の正規表現が入った NULL で終了する文字列を指定します。たとえば、*eventexpr* が "\e\e..*" であれば、呼び出し元はシステムで生成されたすべてのイベントをサブスクライブします。*eventexpr* が "\e\e.SysServer.*" であれば、呼び出し元はシステムで生成されたサーバに関連するすべてのイベントをサブスクライブします。*eventexpr* が "[A-Z].*" であれば、呼び出し元は先頭に A-Z を持つすべてのユーザ・イベント文字列をサブスクライブします。*eventexpr* が ".*(ERR|err).*" である場合、呼び出し側はサブストリング "ERR" または "err" のいずれかを含むすべてのユーザ・イベントを登録しています。たとえば、account_error および ERROR_STATE と呼ばれるイベントは、どちらも登録の対象となります。正規表現の詳細については、第 章の 244 ページ「正規表現」を参照してください。

filter を指定する場合は、ブール値のフィルタ・ルールを含む文字列を指定します。このルールは、イベント・ブローカがイベントをポストする前に正しく評価しなければなりません。ポストすべきイベントを受け取ったイベント・ブローカは、ポストするイベントのデータにフィルタ・ルール (存在する場合) を適用します。データがフィルタ・ルールのチェックにパスした場合、イベント・ブローカは通知方式を呼び出します。データがフィルタ・ルールを通過しない場合は、イベント・ブローカは対応する通知メソッドを呼び出しません。呼び出し元は別のフィルタ・ルールを使用して、同じイベントを何回もサブスクライブすることができます。

フィルタ・ルールは、それが適用される型付きバッファに固有なものです。FML バッファおよび VIEW バッファの場合は、フィルタ・ルールはそれぞれのブール式コンパイラ (それぞれ Fboolco(3fml) および Fvboolco(3fml) を参照) に渡すことができて、ポストされたバッファ (Fboolev(3fml) および Fboolev(3fml) を参照) に対して評価することができる文字列です。STRING バッファの場合は、フィルタ・ルールは正規表現です。他のすべてのタイプのバッファの場合、カスタマイズしたフィルタ評価機構が必要です (カスタマイズしたフィルタ評価機構を追加する方法について詳しくは、buffer(3c) および typesw(5) を参照してください)。*filter* には、最大 255 文字の NULL で終了する文字列を指定します。

サブスクライバが BEA Tuxedo ATMI のクライアント・プロセスで、*ctl* が NULL の場合は、サブスクライブしているイベントがポストされたときに、イベント・ブローカはサブスクライバに任意通知型メッセージを送ります。この場合、*eventexpr* に対して評価が成功するイベント名がポストされると、イベント・ブローカは *eventexpr* に対応したフィルタ・ルールに対してポストされたデータをテストします。データがフィルタ・ルールを通過した場合、またはイベントに対するフィルタ・ルールが存在しない場合は、サブスクライバはイベントと共にポストされたすべてのデータと共に任意通知型の通知を受け取ります。任意通知型メッセージを受け取るためには、クライアントは任意通知処理ルーチンを (tpsetunsol() を利用して) 登録しておく必要があります。BEA Tuxedo システムのサーバ・プロセスが、*ctl* パラメータを NULL にして tpsubscribe() を呼び出した場合は、tpsubscribe() は異常終了して tperrno() を TPEPROTO に設定します。

任意通知型メッセージによってイベント通知を受け取るクライアントは、終了する前にイベント・ブローカのアクティブなサブスクリプションのリストから、そのサブスクリプションを削除するべきです (詳しくは tpunsubscribe(3c) を参照してください)。クライアントで tpunsubscribe() のワイルド・カード・ハンドル -1 を使用すれば、任意通知型の通知メソッドに対応したサブスクリプションを含む、そのクライアントのすべての「非持続型」サブスクリプションを削除することが簡単に行えます (プロセスの終了後も持続するサブスクリプションおよびこれらに関連した通知メソッドについては、次の TPEVPERSIST に関する説明を参照してください)。クライアントが非持続型のサブスクリプションを削除せずに終了した場合は、イベント・ブローカはそのクライアントにアクセス不可能になっていることを検知した時点でそれらのサブスクリプションを削除します。

サブスクライバが (プロセス・タイプにかかわらず) イベント通知をサービス・ルーチンまたは安定記憶域内のキューに送りたい場合は、*ctl* パラメータは有効な TPEVCTL 構造体を指さなければいけません。この構造体には次のエレメントが含まれます。

```
long    flags;
char    name1[32];
char    name2[32];
TPQCTL  qctl;
```

次に、*ctl->flags* 要素に指定する、イベントをサブスクライブするための制御オプションの有効なビットの一覧を示します。

TPEVSERVICE

このフラグは、サブスクライバがイベント通知を *ctl->name1* という名前の BEA Tuxedo ATMI システムのサービス・ルーチンに送りたいことを示します。この場合、*eventexpr* に対して評価が成功するイベント名がポストされると、イベント・ブローカは *eventexpr* に対応したフィルタ・ルールに対してポストされたデータをテストします。データがフィルタ・ルールを通過する場合、またはイベントに対するフィルタ・ルールが存在しない場合は、サービス要求はイベントと共にポストされたデータと合わせて *ctl->name1* に送られます。*ctl->name1* のサービス名には、BEA Tuxedo ATMI システムの有効な任意のサービス名を指定することができ、このサービスはサブスクライブされたときにアクティブである場合もあれば、アクティブでない場合もあります。イベント・ブローカによって呼び出されたサービス・ルーチンは、応答データとともに戻ることはできません。つまり、引数に NULL データを指定して tpreturn() を呼び出すはずです。tpreturn() に渡されるデータはドロップされます。TPEVSERVICE と TPEVQUEUE を同時に指定することはできません。

TPEVTRAN も同時に *ctl->flags* に設定し、また tppost() を呼び出すプロセスがトランザクション・モードにある場合は、イベント・ブローカはサービス・ルーチンがポスト元のトランザクションの一部となるようにサブスクライブされたサービス・ルーチンを呼び出します。イベント・ブローカ (TMUSREVT(5)) とサブスクライブされたサービス・ルーチンの両方が、トランザクションをサポートするサーバ・グループに属していなければなりません (詳しくは UBBCONFIG(5) を参照してください)。*ctl->flags* に TPEVTRAN を設定していない場合は、イベント・ブローカは、サービス・ルーチンがポスト元のトランザクションの一部とならないように、サブスクライブされたサービス・ルーチンを呼び出します。

TPEVQUEUE

このフラグを設定することは、サブスクライバがイベント通知を *ctl->name1* という名前のキュー・スペース、および *ctl->name2* という名前のキューへ登録することを希望していることを示します。この場合、*eventexpr* に対して評価が成功するイベント名がポストされると、イベント・ブローカは *eventexpr* に対応したフィルタ・ルールに対してポストされたデータをテストします。データがフィルタ・ルールを通過した場合、またはイベントに対応したフィルタ・ルールが存在しない場合は、イベント・ブローカはメッセージをイベントと共にポストされたデータと合わせて、*ctl->name1* という名前のキュー・スペース、および *ctl->name2* という名前のキューに登録します。キュー・スペースおよびキューの名前は、BEA Tuxedo ATMI システムの有効な任意のキュー・スペースおよびキューの名前で、サブスクリプションの実行時に存在している場合と存在していない場合があります。

ctl->qctl には、ポストされたイベントをイベント・ブローカがキューに登録することに関するオプションをさらに指定することができます。オプションを何も指定しない場合は、*ctl->qctl.flags* には TPNOFLAGS を設定してください。設定する場合は、tpenqueue(3c) のマニュアル・ページの「制御パラメータ」サブセクションで説明しているようにオプションを設定できます (特に、tpenqueue(3c) への入力情報を制御するフラグの有効なリストを説明しているセクションを参照してください)。TPEVSERVICE と TPEVQUEUE を同時に指定することはできません。

ctl->flags に TPEVTRAN も同時に指定し、tppost() を呼び出すプロセスがトランザクション・モードにある場合は、イベント・ブローカは、ポストされたイベントとそのデータがポスト元のトランザクションの一部となるように、それらをキューに登録します。イベント・ブローカ (TMUSREVT(5)) はトランザクションをサポートするサーバ・グループに属していなければなりません (詳しくは、UBBCONFIG(5) を参照してください)。*ctl->flags* に TPEVTRAN を設定しない場合は、イベント・ブローカは、ポストされたイベントとそのデータがポスト元のトランザクションの一部とならないように、それらをキューに登録します。

TPEVTRAN

このフラグを設定することは、このサブスクリプションのイベント通知が存在する場合に、サブスクライバがこれをポスト元のトランザクションに含めることを希望していることを示します。ポスト元がトランザクション以外の場合、このイベントを通知するためにトランザクションが開始されます。このフラグを設定しない場合は、このサブスクリプションに対してポストされたいかなるイベントも、ポスト元が参加しているどのトランザクションの代わりに実行させることはできません。このフラグは、TPEVSERVICE または TPEVQUEUE のどちらかと同時に指定できます。

TPEVPERSIST

デフォルトで、BEA Tuxedo のイベント・ブローカは、ポストしようとしている資源が利用できない場合 (たとえば、イベント・ブローカがサービス・ルーチンまたはイベントのサブスクリプションに対応したキュー・スペースやキューの名前、もしくはそれらの両方にアクセスできない場合)、サブスクリプションを削除します。このフラグを設定することは、そのようなエラーが発生してもサブスクリプションが持続するように (多くの場合、資源は後で利用できるようになるため) サブスクライバが求めることを示します。このフラグを設定しない場合、このサブスクリプションで指定されたサービス名またはキュー・スペース名 / キュー名のいずれかへのアクセス時にエラーが発生すると、イベント・ブローカはこのサブスクリプションを削除します。

このフラグを TPEVTRAN と同時に指定し、イベントの通知時に資源が利用できない場合は、イベント・ブローカはポスト元に戻り、トランザクションが中止しなければならないようにします。つまりこれは、サブスクリプションがそのままの状態で残ったとしても、リソースを利用できないことがポスト元のトランザクションの異常終了の原因になるということです。

イベント・ブローカのアクティブなサブスクリプションのリストに、tpsubscribe() が要求するサブスクリプションと一致するものがある場合は、この関数は異常終了して tperrno() に TPEMATCH を設定します。サブスクリプションが既存のサブスクリプションと一致するためには、*eventexpr* と *filter* の両方が、イベント・ブローカのアクティブなサブスクリプションのリストにすでに存在するサブスクリプションの *eventexpr* と *filter* に一致しなければなりません。さらに、通知メソッドによって異なりますが、一致を調べるために他の基準も使用されます。

サブスクライバが BEA Tuxedo ATMI システムのクライアント・プロセスで、(イベントがポストされたときに、呼び出し元が任意通知を受け取るように) *ctl* に NULL を設定した場合は、システム定義によるそのクライアント識別子 (CLIENTID と呼ばれています) も一致を調べるために使用されます。つまり tpsubscribe() は、*eventexpr*、*filter*、および呼び出し元の CLIENTID が、イベント・ブローカにすでに知られているサブスクリプションが持つそれらの値と一致する場合に異常終了します。

呼び出し元が *ctl->flags* に TPEVSERVICE を設定した場合は、*eventexpr*、*filter*、および *ctl->name1* に設定されたサービス名が、イベント・ブローカにすでに知られているサブスクリプションが持つそれらの値と一致する場合に tpsubscribe() は異常終了します。

安定記憶域内のキュー、キュー・スペース、およびキューの名前へのサブスクリプションの場合は、一致を調べる際に *eventexpr* および *filter* に加えて相関識別子が使用されます。相関識別子は、同じイベント表現とフィルタ・ルールを持ち、同じキューに向けられている複数のサブスクリプションを区別するために使用できます。したがって、呼び出し元が *ctl->flags* に TPEVQUEUE を設定し、*ctl->qctl.flags* に TPQCOORID が設定されなかった場合、*eventexpr*、*filter*、*ctl->name1* に設定されたキュー・スペース名、および *ctl->name2* に設定されたキュー名が、イベント・ブローカにすでに知られている (相関識別子が指定された) サブスクリプションが持つそれらの値と一致すると tpsubscribe() は異常終了します。さらに、*ctl->qctl.flags* に TPQCOORID が設定されている場合は、*eventexpr*、*filter*、*ctl->name1*、*ctl->name2*、および *ctl->qctl.corrid* がイベント・ブローカに既に知られている (同じ相関識別子が指定された) サブスクリプションのデータと一致すると、tpsubscribe() は異常終了します。

次に tpsubscribe() に指定できる有効な *flags* の一覧を示します。

TPNOBLOCK
: ブロッキング条件が存在する場合は、サブスクリプションは行われません。このような条件が発生すると、呼び出しは異常終了し、tperrno() に TPEBLOCK が設定されます。TPNOBLOCK が指定されていないときにブロッキング条件が存在すると、呼び出し元は、その条件が解消されるか、またはタイムアウト (トランザクションまたはブロッキング) が発生するまではブロックされます。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。ただし、トランザクションのタイムアウトは発生します。

TPSIGRSTRT
: シグナルが関数内部のシステム・コールを中断すると、中断されたシステム・コールは出しなおされます。TPSIGRSTRT が指定されていない場合にシグナルがシステム・コールを中断させると、tpsubscribe() は異常終了し、tperrno() には TPGOTSIG が設定されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tpsubscribe() の呼び出しを発行できません。

正規表現

表 12 で説明する正規表現は、UNIX システム・エディタ、ed(1) で使用されるパターンに似ています。一般的な正規表現のほか、代替演算子 (|) も使用できます。ただし、全体的にはほとんど変わりません。

正規表現 (RE: Regular Expression) は、次に示すいずれかの規則を 1 回以上適用して作成します。

表 12 正規表現

規則	一致対象のテキスト
任意の文字	任意の文字 (以下に示す特殊文字を除く任意の ASCII 文字)。
\ 任意の文字	以下に示す以外の任意の文字。 ■ \\— 改行 ■ \\t— タブ ■ \\b— バックスペース ■ \\r— キャリッジ・リターン ■ \\f— フォームフィード
\ 特殊文字	非特殊文字。特殊文字には、ピリオド (.)、*、+、?、|、(、)、[、{、\\、があります。 .— 行末文字 (通常は改行文字または NULL 文字) 以外の任意の文字 ^— 行頭 $— 行末文字
[class]	一連の文字または範囲、あるいはその両方で表すクラス内の任意の文字。範囲は、「character-character」という形式で指定されます。たとえば、文字クラス [a-zA-Z0-9_] は、アルファベット文字またはアンダーライン "_" と一致します。ハイフンをクラスに含めるには、"\\" の後のハイフンをエスケープするか、クラス内の先頭または終わりに指定する必要があります。リテラル "]" の場合は、エスケープするか、またはクラスの先頭に指定します。リテラル "^" がクラスの先頭にある場合は、エスケープする必要があります。
[^ class]	行末文字を除く、クラスの補集合における ASCII 文字セットに関する文字。
RE RE	正規表現のシーケンス (連結)。
RE | RE	左側の RE または右側の RE (左右の二者択一)。
RE *	RE が 0 回以上発生。
RE +	RE が 1 回以上発生。
RE ?	RE が 0 回または 1 回発生。
RE { n }	RE が *n* 回発生。*n* の範囲は 0 ～ 255 です。

表 12 正規表現 (続き)

規則	一致対象のテキスト
RE { m, n }	RE が m ～ n の範囲の回数発生。*m* が指定されない場合は 0 になります。*n* が指定されない場合は、RE が *m* 回以上発生することを示します。
(RE)	優先順位またはグループ化を明示的に示します。
(RE) $ n	テキスト・マッチングの RE は、*n* 番目のユーザ・バッファにコピーされます。*n* の範囲は、0 ～ 9 です。ユーザ・バッファは、マッチング処理の開始までにクリアされ、パターン全体がマッチした場合にのみロードされます。

優先順位のレベルは 3 つあります。結合の強さの順に並べると、次のようになります。

- 連結閉鎖 (*、+、?、{...})
- 連結
- 二者択一 (|)

上記のとおり、かっこは、優先順位が高いことを明示的に示すために使用します。

戻り値

tpsubscribe() は正常終了すると、イベント・ブローカのアクティブなサブスクリプションのリストからこのサブスクリプションを削除するために使用できるハンドルを返します。サブスクライバやその他のプロセスは、いずれも返されたハンドルを利用してこのサブスクリプションを削除することができます。

異常終了すると、tpsubscribe() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpsubscribe() は tperrno() を次のいずれかの値に設定します (特に指定がない限り、障害は、呼び出し元のトランザクションに影響しません)。

[TPEINVAL]
無効な引数 (たとえば、*eventexpr* に NULL) が指定されました。

[TPENOENT]
BEA Tuxedo イベント・ブローカにアクセスできません。

[TPELIMIT]
イベント・ブローカが最大サブスクリプション数に達したため、サブスクライブできません。

[TPEMATCH]
すでにイベント・ブローカのリストに存在するサブスクリプションと同じであるため、サブスクライブできません。

[TPEPERM]
クライアントは tpsysadm としてアタッチされず、サブスクリプション・アクションは、サービスの呼び出しかメッセージのキューへの登録になります。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、そのトランザクションは終了します。トランザクション・モードでなければ、ブロッキング・タイムアウトが発生し、TPNOBLOCK も TPNOTIME も指定されていませんでした。トランザクション・タイムアウトが発生した場合、新しく処理を開始しようとしても、トランザクションがアボートするまで TPETIME になり、正常に行えません。

[TPEBLOCK]
ブロッキング状態のため、TPNOBLOCK が指定されました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tpsubscribe() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目

buffer(3c)、tpenqueue(3c)、tppost(3c)、tpsetunsol(3c)、tpunsubscribe(3c)、Fboolco、Fboolco32、Fvboolco、Fvboolco32(3fml)、Fboolev、Fboolev32、Fvboolev、Fvboolev32(3fml)、EVENTS(5)、EVENT_MIB(5)、TMSYSEVT(5)、TMUSREVT(5)、tuxtypes(5)、typesw(5)、UBBCONFIG(5)

tpsuspend(3c)

名前

tpsuspend()— グローバル・トランザクションの中断

形式

```
#include <atmi.h>
int tpsuspend(TPTRANID *tranid, long flags)
```

機能説明

tpsuspend() を使用して、呼び出し元のプロセス内でアクティブなトランザクションを中断します。tpbegin() により開始したトランザクションは、tpsuspend() で中断できます。中断を行ったプロセスまたは他のプロセスのいずれかが tpresume() を使用して、中断されたトランザクション上の作業を再開できます。tpsuspend() が復帰すると、呼び出し元はトランザクション・モードではなくなります。ただしトランザクションが中断されている間、トランザクションに関連する全てのリソース (データベース・ロック等) は、アクティブのままです。アクティブなトランザクションと同様に、中断されたトランザクションは、それが開始された時に割り当てられたトランザクション・タイムアウトの値に影響されます。

トランザクションを別のプロセスで再開するには、tpsuspend() の呼び出し元が明示的に tpbegin() を呼び出すことによってトランザクションを起動している必要があります。tpsuspend() は、トランザクションの開始元以外のプロセスから呼び出すこともできます (例えば、トランザクション・モードで要求を受信するサーバ)。後者の場合、tpsuspend() の呼び出し元のみ tpresume() を呼び出してトランザクションを再開することができます。これは、プロセスが一時的にトランザクションを中断して、そのトランザクションを完了する前に別のトランザクションを開始し、処理するのに使用できます (例えば、障害ログをとるトランザクションを実行してから元のトランザクションに戻る)。

tpsuspend() は、中断されているトランザクションの識別子を、*tranid* で指される領域に返します。呼び出し元は *tranid* が指す空間を割り当てなければなりません。*tranid* が NULL の場合はエラーです。

正常終了するためには、呼び出し元は tpsuspend() を実行する前に、サーバとの未終了のコミュニケーションを全て完了していなければなりません。すなわち、呼び出し元は、呼び出し元のトランザクションに関連のある tpacall() で送出した要求に対する応答を、すべて受け取っていなければなりません。また、呼び出し元は、呼び出し元のトランザクションに関連のある会話サービスとの接続を、全てクローズしていなければなりません (つまり、tprecv() は TPEV-SVCSUCC イベントを返していなければ なりません)。この規則のいずれかが守られない場合には、tpsuspend() は異常終了し、呼び出し元の現在のトランザクションは中断されず、トランザクション・コミュニケーションの記述子は有効なままです。呼び出し元のトランザクションに関連しないコミュニケーション記述子は、tpsuspend() の結果に関係なく有効なままです。

flags は将来使用するために予約されており、0 に設定されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tpsuspend() の呼び出しを発行できません。

戻り値

tpsuspend() はエラーの場合は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

次の条件の場合、tpsuspend() は異常終了し、tperrno() を次の値に設定します。

[TPEINVAL]
tranid が NULL ポインタか、または *flags* が 0 ではありません。トランザクションについての呼び出し元の状態は変化しません。

[TPEABORT]
呼び出し元のアクティブなトランザクションがアボートしました。トランザクションに関連する全てのコミュニケーション記述子は、もはや有効ではありません。

[TPEPROTO]
tpsuspend() が不正なコンテキストで呼ばれました (例えば、呼び出し元がトランザクション・モードではない)。トランザクションについての呼び出し元の状態は変化しません。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目

tpacall(3c)、tpbegin(3c)、tprecv(3c)、tpresume(3c)

tpsvrdone(3c)

名前

tpsvrdone()—BEA Tuxedo ATMI システム・サーバの終了

形式

```
#include <atmi.h>
void tpsvrdone(void)
```

機能説明

BEA Tuxedo ATMI システムのサーバ用ルーチンは、サービス要求の処理完了後、ルーチンを終了する前に tpsvrdone() を呼び出します。このルーチンを呼び出した時点では、サーバはまだシステムの一部のままですが、それ独自のサービスは宣言から外されています。このため、このルーチンで、BEA Tuxedo ATMI システムとのコミュニケーションとトランザクションの定義を行うことができます。ただし、接続がオープン状態にある場合、保留中の非同期応答がある場合、あるいはまだトランザクション・モードにある場合に tpsvrdone() が終了すると、BEA Tuxedo ATMI システムはその接続をクローズし、保留中の応答を無視し、サーバが終了する前にトランザクションをアボートさせます。

サーバが tmshutdown -y を呼び出してシャットダウンされた場合、サービスは中断され、tpsvrdone() で通信したりトランザクションを開始する機能は制限されます。

アプリケーションがこのルーチンをサーバで提供しない場合、BEA Tuxedo ATMI システムが提供するデフォルトのバージョンが代わりに呼び出されます。サーバがシングルスレッド・サーバとして定義されている場合、デフォルトの tpsvrdone() は tpsvrthrdone() を呼び出し、デフォルト・バージョンの tpsvrthrdone() は tx_close() を呼び出します。サーバがマルチスレッド・サーバとして定義されている場合、各サーバのディスパッチ・スレッドで tpsvrthrdone() が呼び出され、tpsvrdone() からは呼び出されません。サーバがマルチスレッドかどうかに関わらず、デフォルトの tpsvrdone() は userlog を呼び出し、サーバが終了することを示します。

使用法

tpsvrdone() で呼び出された場合、tpreturn() と tpforward() は何も行わずにただちに戻ります。

関連項目

tpsvrthrdone(3c)、tpsvrthrinit(3c)、servopts(5)

tpsvrinit(3c)

名前

tpsvrinit()—BEA Tuxedo システム・サーバの初期化

形式

```
#include <atmi.h>
int tpsvrinit(int argc, char **argv)
```

機能説明

BEA Tuxedo ATMI システムのサーバ用ルーチンは、その初期化処理中に tpsvrinit() を呼び出します。このルーチンは、サーバに制御が移った後、サービス要求を処理する前に呼び出されます。このため、BEA Tuxedo ATMI システムとのコミュニケーションとトランザクションの定義をこのルーチンで行うことができます。ただし、接続がオープン状態にあるとき、保留中の非同期応答があるとき、あるいはまだトランザクション・モードにあるときに tpsvrinit() が戻った場合、BEA Tuxedo ATMI システムはその接続をクローズし、保留中の応答を無視し、サーバが終了する前にトランザクションをアボートします。

アプリケーションがこのルーチンをサーバで提供しない場合、BEA Tuxedo ATMI システムが提供するデフォルトのバージョンが代わりに呼び出されます。

サーバがシングルスレッド・サーバとして定義されている場合、デフォルトの tpsvrinit() は tpsvrthrinit() を呼び出し、デフォルト・バージョンの tpsvrthrinit() は tx_open() を呼び出します。サーバがマルチスレッド・サーバとして定義されている場合、tpsvrthrinit() は各サーバのディスパッチ・スレッドで呼び出されますが、tpsvrinit() からは呼び出されません。サーバがシングルスレッドかマルチスレッドかに関わらず、デフォルト・バージョンの tpsvrinit() は userlog() を呼び出してサーバが正常に開始したことを示します。

アプリケーション固有のオプションをサーバに渡し、tpsvrinit() で処理させることができます (servopts(5) 参照)。このオプションは *argc* と *argv* を使用して渡します。getopt() が BEA Tuxedo ATMI システムのサーバ用ルーチンで使用されているため、optarg()、optind() および opterr() を使用してオプションの解析やエラー検出を tpsvrinit() で制御できます。

tpsvrinit() でエラーが生じた場合、アプリケーションから -1 を返して明示的にサーバを終了させることができます (サービス要求をとらずに)。アプリケーション自体では exit() を呼び出さないようにしてください。

戻り値

負の戻り値は、サーバを適切に終了させます。

使用法

tpreturn() や tpforward() がサービス・ルーチンの外部 (たとえば、クライアントや tpsvrinit()、あるいは tpsvrdone() で) 使用された場合、何も行わずにただちに戻ります。

関連項目　　tpopen(3c)、tpsvrdone(3c)、tpsvrthrinit(3c)、servopts(5)
C 言語リファレンス・マニュアルの getopt(3)

tpsvrthrdone(3c)

名前

tpsvrthrdone()—BEA Tuxedo ATMI サーバのスレッドの終了

形式

```
#include <atmi.h>
void tpsvrthrdone(void)
```

機能説明

BEA Tuxedo ATMI のサーバでは、ディスパッチされたサービス要求を処理するために開始された各スレッドを終了するときに、tpsvrthrdone() を呼び出します。つまり、スレッドが要求を処理する前に終了された場合でも、tpsvrdone() 関数が呼び出されます。このルーチンが呼び出されたとき、対象のスレッドはまだ BEA Tuxedo ATMI サーバの一部ですが、すべてのサービス要求の処理を終了しています。このため、BEA Tuxedo ATMI とのコミュニケーションとトランザクションの定義をこのルーチンで行うことができます。ただし、接続がオープン状態にある場合や保留中の非同期応答がある場合、あるいはまだトランザクション・モードにある場合に tpsvrthrdone() が終了すると、BEA Tuxedo ATMI システムはその接続をクローズし、保留中の応答を無視し、サーバが終了する前にトランザクションを異常終了させます。

アプリケーションがサーバ中にこのルーチンを記述していない場合、BEA Tuxedo ATMI システムによって提供されるデフォルト・バージョンの tpsvrthrdone() が代わりに呼び出されます。デフォルト・バージョンの tpsvrthrdone() では、tx_close() が呼び出されます。

tpsvrthrdone() は、シングルスレッド・サーバでも呼び出されます。シングルスレッド・サーバの tpsvrthrdone() は、デフォルト・バージョンの tpsvrdone() から呼び出されます。複数のディスパッチ・スレッドが予測されるサーバでは、tpsvrdone() は tpsvrthrdone() を呼び出しません。

使用法

tpsvrthrdone() から呼び出された場合、tpreturn() と tpforward() 関数は単純に返るだけで影響はありません。

関連項目

tpforward(3c)、tpreturn(3c)、tpsvrdone(3c)、tpsvrthrinit(3c)、tx_close(3c)、servopts(5)

tpsvrthrinit(3c)

名前

tpsvrthrinit()—BEA Tuxedo ATMI サーバのスレッドの初期化

形式

```
#include <atmi.h>
int tpsvrthrinit(int argc, char **argv)
```

機能説明

BEA Tuxedo ATMI のサーバでは、ディスパッチされたサービス要求を処理する各スレッドを初期化するときに、tpsvrthrinit() を呼び出します。このルーチンは、サーバに制御が移った後、サービス要求を処理する前に呼び出されます。このため、BEA Tuxedo ATMI とのコミュニケーションとトランザクションの定義をこのルーチンで行うことができます。ただし、接続がオープン状態にある場合や保留中の非同期応答がある場合、あるいはまだトランザクション・モードにある場合に tpsvrthrinit() が終了すると、BEA Tuxedo ATMI システムはその接続をクローズし、保留中の応答を無視し、サーバが終了する前にトランザクションを異常終了させます。

アプリケーションがサーバにこのルーチンを提供していない場合、BEA Tuxedo ATMI システムによって提供されたデフォルト・バージョンの tpsvrthrinit() が代わりに呼び出されます。デフォルト・バージョンの tpsvrthrinit() は tx_open() を呼び出します。

tpsvrthrinit() は、シングルスレッド・サーバでも呼び出されます。シングルスレッド・サーバでは、tpsvrthrinit() はデフォルト・バージョンの tpsvrinit() から呼び出されます。複数のディスパッチ・スレッドが予測されるサーバでは、tpsvrinit() は tpsvrthrinit() を呼び出しません。

アプリケーション固有のオプションをサーバに渡し、tpsvrthrinit() で処理させることができます。オプションの詳細については、servopts(5) を参照してください。このオプションは *argc* と *argv* を使用して渡します。getopt() が BEA Tuxedo ATMI サーバで使用されているため、optarg()、optind()、および opterr() を使用して tpsvrthrinit() のオプションの解析やエラー検出を制御できます。

tpsvrthrinit() でエラーが生じた場合、アプリケーションから -1 を返して正常に (サービス要求をとらずに) サーバのディスパッチ・スレッドを終了させることができます。アプリケーションが、exit() やその他オペレーティング・システムのスレッド終了関数を呼び出してはいけません。

戻り値

負の戻り値によって、サーバのディスパッチ・スレッドは正常に終了します。

使用法

サービス・ルーチンの外側で使用された場合 (たとえば、クライアントまたは tpsvrinit()、tpsvrdone()、tpsvrthrinit()、tpsvrthrdone() で使用された場合)、tpreturn() と tpforward() 関数は単純に返るだけで影響はありません。

関連項目

`tpforward(3c)`, `tpreturn(3c)`, `tpsvrthrdone(3c)`, `tpsvrthrinit(3c)`, `tx_open(3c)`, `servopts(5)`

C 言語リファレンス・マニュアルの `getopt`(3)

tpterm(3c)

名前　tpterm()— アプリケーションからの分離

形式

```
#include <atmi.h>
int tpterm(void)
```

機能説明　tpterm() は、BEA Tuxedo ATMI システム・アプリケーションからクライアントを削除します。クライアントがトランザクション・モードであると、トランザクションはロール・バックします。tpterm() が正常に終了すると、呼び出し元は BEA Tuxedo ATMI クライアント操作を実行できません。未終了の会話はただちに切断されます。

tpterm() を 2 回以上呼び出した場合 (すなわち、呼び出し元がすでにアプリケーションから離れた後で呼び出した場合)、何も処理は行われず、正常終了を示す値が返されます。

マルチスレッド化およびマルチコンテキスト化の問題

適切なプログラミングでは、1 つを残して他のすべてのスレッドがコンテキストを終了または切り替えると、最後のスレッドが tpterm() 呼び出しを発行します。このようにプログラミングされていないと、残りのスレッドは TPINVALIDCONTEXT コンテキスト状態になります。次に、このコンテキストのセマンティクスを説明します。

複数のスレッドが関連するコンテキストにおいて、1 つのスレッドで tpterm() が呼び出されると、この tpterm() は以下のように動作します。

- 1 つのプロセス内のすべてのコンテキストではなく、1 つのコンテキスト内のすべてのスレッド上で動作します。
- 同じプロセスの他のスレッドとそのコンテキストがまだ関連している場合でも、すぐに動作を実行します。

別のスレッドがコンテキストを終了したときに ATMI 呼び出し内でブロックされたスレッドがあると、そのスレッドは ATMI 呼び出しから異常終了によって返され、tperrno() は TPESYSTEM に設定されます。また、このような異常終了の後で tperrordetail() が呼び出された場合、tperrordetail() は TPED_INVALIDCONTEXT を返します。

シングルコンテキストのアプリケーションでは、単一のスレッドが tpterm() を呼び出すと、すべてのスレッドのコンテキスト状態が TPNULLCONTEXT に設定されます。

それに対して、マルチコンテキストのアプリケーションでは、tpterm() が 1 つのスレッドで呼び出されると、同じコンテキスト内の他のすべてのスレッドは、ほとんどの ATMI 関数を呼び出しても異常終了し、tperrno が TPEPROTO に設定されるような状態になります。このような無効なコンテキスト状態で使用できる関数と使用できない関数のリストについては、7 ページ「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」を参照してください。無効なコンテキスト状態 (TPINVALIDCONTEXT) のスレッドが tpgetctxt() 関数を呼び出した場合、tpgetctxt() によってコンテキストのパラメータが TPINVALIDCONTEXT に設定されます。

TPINVALIDCONTEXT 状態を終了させるには、次の関数のどちらかを呼び出します。

- TPNULLCONTEXT コンテキストまたは別の有効なコンテキストを設定した tpsetctxt()
- tpterm()

TPINVALIDCONTEXT のコンテキストを設定して tpsetctxt() を呼び出すことはできません。このような場合は、異常終了して tperrno が TPEPROTO に設定されます。呼び出し元とアプリケーションを関連させる必要がない tpsetunsol() 以外の ATMI 関数をスレッドが呼び出すと、ATMI 関数はヌル・コンテキストで呼び出されたときと同じように動作します。任意通知型のスレッド通知を使用するクライアント・アプリケーションは、tpterm() を明示的に呼び出して、任意通知型スレッドを終了する必要があります。

tpterm() 呼び出し後、スレッドは TPNULLCONTEXT コンテキスト状態になります。TPNULLCONTEXT コンテキストのスレッドで呼び出される ATMI 関数の多くは、暗黙的な tpinit() を実行します。tpinit() の呼び出しが成功するかどうかは、コンテキスト固有の問題やスレッド固有の問題ではなく、通常の要因によって決まります。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tpterm() の呼び出しを発行できます。

戻り値

シングルコンテキストのアプリケーションで正常終了すると、このアプリケーションの現在のコンテキスト内のすべてのスレッドは、TPNULLCONTEXT 状態になります。

マルチコンテキストのアプリケーションで正常終了すると、呼び出し側のスレッドは TPNULLCONTEXT 状態になり、呼び出し側のスレッドと同じコンテキスト内の他のスレッドはすべて TPINVALIDCONTEXT 状態になります。後者のスレッドのコンテキスト状態は、引き数 *context* を TPNULLCONTEXT か別の有効なコンテキストに設定して tpsetctxt() を実行すれば変更できます。

異常終了すると、tpterm() は呼び出し側のプロセスを元のコンテキスト状態のままで -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpterm() は tperrno() を次のいずれかの値に設定します。

[TPEPROTO]
tpterm() が不正なコンテキストで呼び出されました (たとえば、呼び出し元がサーバである場合)。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目

tpinit(3c)、tpgetctxt(3c)、tpsetctxt(3c)、tpsetunsol(3c)

tptypes(3c)

名前

tptypes()— 型付きバッファ情報を判別するルーチン

形式

```
#include <atmi.h>
long tptypes(char *ptr, char *type, char *subtype)
```

機能説明

tptypes() は、その第 1 引数として、データ・バッファを指すポインタをとり、2 番目と 3 番目の引数でそれぞれタイプとサブタイプを返します。*ptr* は、tpalloc() から得たバッファを指していなければなりません。*type* と *subtype* が NULL でない場合、この関数は、そのバッファのタイプとサブタイプの名前をそれぞれ該当する文字配列に入れます。これらの名前が最大長であると (*type* の場合は 8、*subtype* の場合は 16)、この文字配列は NULL で終了しません。また、サブタイプが存在しない場合は、*subtype* が指す配列には NULL 文字列が入ります。

なお、*type* の場合は最初の 8 バイト、*subtype* の場合は最初の 16 バイトが格納されます。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tptypes() の呼び出しを発行できます。

戻り値

正常終了の場合、tptypes() はバッファのサイズを返します。

異常終了すると、この関数は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tptypes() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
: 無効な引数が与えられた (たとえば、*ptr* がもともと \% tpalloc() から得たバッファを指していない場合など)。

[TPEPROTO]
: tptypes() が不正に呼び出されました。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

関連項目

tpalloc(3c)、tpfree(3c)、tprealloc(3c)

tpunadvertise(3c)

名前

tpunadvertise()— サービス名の宣言解除を行うルーチン

形式

```
#include <atmi.h>
int tpunadvertise(char *svcname)
```

機能説明

tpunadvertise() を使用すると、サーバは、宣言したサービスの宣言解除を行うことができます。デフォルトの設定では、サーバのサービスは、サーバのブート時に宣言され、サーバのシャットダウン時にその宣言が解除されます。

複数サーバ単一キュー (MSSQ) セットに属するすべてのサーバは、同じサービス・セットを提供しなければなりません。これらのルーチンは、MSSQ セットを共有する全サーバを宣言することによってこの規則を適用します。

tpunadvertise() は、該当サーバ (または、呼び出し元の MSSQ セットを共有するサーバ・セット) に対して宣言されたサービスとして *svcname* を除去します。*svcname* に NULL または NULL 文字列 ("") を使用することはできません。また、*svcname* の長さは 15 文字までとしてください (UBBCONFIG(5) の SERVICES セクションを参照)。これ以上の長さの名前でも受け付けられますが、15 文字以降は切り捨てられてしまいます。このため、切り捨てられた名前が他のサービス名と同じにならないよう、注意が必要です。

戻り値

異常終了すると、tpunadvertise() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpunadvertise() は tperrno() を次のいずれかの値に設定します。

[TPEINVAL]
: *svcname* が NULL または NULL 文字列 ("") である場合。

[TPENOENT]
: *svcname* がサーバによって宣言されていない場合。

[TPEPROTO]
: tpunadvertise() が不正なコンテキストで呼び出された場合 (たとえば、クライアントによって)。

[TPESYSTEM]
: BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
: オペレーティング・システムのエラーが発生しました。

関連項目　　tpadvertise(3c)

tpunsubscribe(3c)

名前

tpunsubscribe()— イベントのサブスクリプションを削除する

形式

```
#include <atmi.h>
int tpunsubscribe(long subscription, long flags)
```

機能説明

呼び出し元は tpunsubscribe() を使用して、BEA Tuxedo イベント・ブローカのアクティブなサブスクリプションのリストからイベントのサブスクリプションまたはイベントのサブスクリプションの集合を削除します。*subscription* には、tpsubscribe() が返したイベントのサブスクリプションのハンドルを指定します。*subscription* をワイルドカード値の -1 に設定すると、tpunsubscribe() が、呼び出し元のプロセスが以前に行ったすべての非持続型のサブスクリプションを削除する指示になります。非持続型のサブスクリプションとは、tpsubscribe() の *ctl->flags* パラメータに TPEVPERSIST ビットを設定して行われたサブスクリプションのことです。持続タイプのサブスクリプションは、tpsubscribe() から返されたハンドルを使用することによってのみ削除できます。

ハンドル -1 を指定すると、呼び出し元のプロセスが行ったサブスクリプションのみを削除し、呼び出し元が以前に起動されたときに行ったサブスクリプションは削除しないことに注意してください (たとえば、異常終了した後で再起動したサーバでは、ワイルドカードを使用して以前のサーバが行ったサブスクリプションを削除することはできません)。

次に、有効な *flags* の一覧を示します。

TPNOBLOCK
: ブロッキング条件が存在する場合は、サブスクリプションは削除されません。このような条件が発生すると、呼び出しは異常終了し、tperrno() には TPEBLOCK が設定されます。TPNOBLOCK が指定されていないときにブロッキング条件が存在すると、呼び出し元は、その条件が解消されるか、またはタイムアウト (トランザクションまたはブロッキング) が発生するまではブロックされます。

TPNOTIME
: このフラグは、呼び出し元が無制限にブロックでき、ブロッキング・タイムアウトの対象にならないようにすることを指定します。トランザクション・タイムアウトは依然として発生する可能性があります。

TPSIGRSTRT
: シグナルが関数内部のシステム・コールを中断すると、中断されたシステム・コールは出しなおされます。TPSIGRSTRT が指定されていない場合にシグナルがシステム・コールを中断させると、tpunsubscribe() は異常終了し、tperrno() には TPGOTSIG が設定されます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tpunsubscribe() の呼び出しを発行できません。

戻り値

tpunsubscribe() が成功して戻ると、tpurcode() にはイベント・ブローカのアクティブなサブスクリプションのリストから削除されたサブスクリプションの数(0 または 1 以上) が指定されます。tpurcode() に 1 より大きな数が設定されるのは、ワイルドカードのハンドルを -1 に指定した場合のみです。また、tpunsubscribe() が失敗して終了した場合にも、tpurcode() に 1 より大きな数が設定されることがあります (つまり、ワイルドカードのハンドルを指定して、イベント・ブローカがいくつかのサブスクリプションの削除に成功した後で、他のサブスクリプションを削除する際にエラーが発生したような場合です)。

異常終了すると、tpunsubscribe() は -1 を返し、tperrno() を設定してエラー条件を示します。

エラー

異常終了時には、tpunsubscribe() は tperrno() を次のいずれかの値に設定します (特に指定がない限り、障害は、呼び出し元のトランザクションに影響しません)。

[TPEINVAL]
無効な引数 (たとえば、*subscription* に無効なサブスクリプションのハンドル) が指定されました。

[TPENOENT]
BEA Tuxedo のイベント・ブローカにアクセスできません。

[TPETIME]
タイムアウトが発生しました。呼び出し元がトランザクション・モードの場合は、トランザクション・タイムアウトが発生し、そのトランザクションは終了します。トランザクション・モードでなければ、ブロッキング・タイムアウトが発生し、TPNOBLOCK も TPNOTIME も指定されていませんでした。トランザクション・タイムアウトが発生した場合、新しく処理を開始しようとしても、トランザクションがアボートするまで TPETIME になり、正常に行えません。

[TPEBLOCK]
ブロッキング状態のため、TPNOBLOCK が指定されました。

[TPGOTSIG]
シグナルを受け取りましたが、TPSIGRSTRT が指定されていません。

[TPEPROTO]
tpunsubscribe() が不正に呼び出されました。

[TPESYSTEM]
BEA Tuxedo システムのエラーが発生しました。エラーの正確な内容はログ・ファイルに書き込まれます。

[TPEOS]
オペレーティング・システムのエラーが発生しました。

関連項目 tppost(3c)、tpsubscribe(3c)、EVENTS(5)、EVENT_MIB(5)、TMSYSEVT(5)、TMUSREVT(5)

TRY(3c)

名前

TRY( )— 例外復帰インターフェイス

形式

```
#include <texc.h>

TRY
try_block
[  CATCH(exception_name) handler_block] ...
[CATCH_ALL handler_block]
ENDTRY

TRY
try_block
FINALLY
finally_block
ENDTRY

RAISE(exception_name)
RERAISE

/* 例外の宣言 */
EXCEPTION exception_name;

/* アドレス ( アプリケーション ) 例外の初期化 */
EXCEPTION_INIT(EXCEPTION exception_name)

/* ステータス例外の初期化 ( ステータスを例外にマップ ) */
exc_set_status(EXCEPTION *exception_name, long status)

/* ステータス例外をステータスにマップ */
exc_get_status(EXCEPTION *exception_name, long *status)

/* 例外の比較 */
exc_matches(EXCEPTION *e1, EXCEPTION *e2)

/* stderr にエラーを出力 */
void exc_report(EXCEPTION *exception)
```

機能説明

TRY/CATCH インターフェイスは、ステータス変数 (例えば、*errno* や RPC オペレーションで返されるステータス変数) を使用せずに例外を処理する機能を提供します。これらのマクロは texc.h ヘッダ・ファイルに定義され、このヘッダ・ファイルは tidl(1) で作成されるすべてのヘッダ・ファイルに自動的にインクルードされます。

TRY の *try_block* は C または C++ の宣言文とステートメントのブロックであり、このブロック内で例外が発生します (例外の発生と関係のないコードは *try_block* の前、あるいは後に配置します)。TRY/ENDTRY の対により、例外のスコープ、つまり例外を検出するコード領域が構成されます (C 言語のスコーピングとは異なります)。これらのスコープはネストすることができます。例外が発生すると、例外を処理するためのアクション (CATCH または CATCH_ALL クローズ)、あるいはスコープを完結するためのアクション (FINALLY クローズ) に対応するアクティブなスコープを検索することにより、エラーがアプリケーションにレポートされます。スコープが例外処理を行えない場合には、そのスコープは終了し次の上位レベルで例外が生成されます (例外スコープのスタックをアンワインドします)。実行は、処理が行われた箇所の後から再開します。エラーが発生した箇所から実行を再開することはありません。いずれのスコープでも例外が処理されない場合には、プログラムは終了します (userlog(3c) と abort(3) を呼び出すことにより、メッセージがログに書き込まれます)。

CATCH (*exception_name*) *handler_block* は、存在しなくても複数回記述してもかまいません。各 *handler_block* は C または C++ の宣言文またはステートメントであり、対応する例外 (*exception_name*) の処理を行います (通常、障害からリカバリするためのアクションが指定されます)。例外が *try_block* 内のステートメントで発生した場合には、その例外に対応する最初の CATCH クローズが実行されます。

CATCH または CATCH_ALL *handler_block* 内では、現レベルの例外は EXCEPTION ポインタ THIS_CATCH により参照できます (たとえば、例外値に基づくロジックや、例外値の印刷など)。

例外がいずれの CATCH クローズでも処理できない場合には、CATCH_ALL クローズが実行されます。デフォルトでは CATCH または CATCH_ALL クローズで処理される例外に対しては、それ以上のアクションは取られません。CATCH_ALL クローズが存在しない場合には、*try_block* が他の *try_block* にネストされているものと想定し、次の上位の *try_block* で例外が発生します。ANSI C コンパイラを使用した場合には、ハンドラ・ブロック内で使用されるレジスタと自動変数に volatile 属性を付加して宣言しなければなりません (setjmp/longjmp を使用するブロックの場合も同様です)。また、例外を発生する関数からの出力パラメータと戻り値は、不明であることに注意してください。

CATCH または CATCH_ALL *handler_block* 内では、現レベルの例外は RERAISE ステートメントにより、次の上位のレベルに伝播されます (例外が再度発生します)。RERAISE ステートメントは、*handler_block* のスコープ内 (つまり *handler_block* により呼び出された関数内ではありません) になければなりません。検出はできますが、完全には処理できない例外はすべて出し直さなければなりません。ほとんどの場合、ハンドラはすべての例外を処理するようには書かれていないので、CATCH_ALL ハンドラは例外を再度発生させる必要があります。アプリケーションは、例外が該当するスコープで発生し、ハンドラ・ブロックが適切な処理を行って該当するステータスを変更するように書かれなければなりません (例えば、ファイルのオープン中に例外が発生した場合には、そのレベルの例外関数は、オープンされていないファイルをクローズしてはいけません)。

例外は、RAISE (*exception_name*) ステートメントを使用してどこででも発生させることができます。このステートメントにより、例外は現在の *try_block* から伝播を開始して、処理が行われる上位レベルに到達するまで出し直されます。

FINALLY クローズは、例外の発生に関係なく、*try_block* の後で実行されるコードの終局ブロックを指定するのに使用します。例外は *try_block* 内で発生した場合には、*finally_block* が実行された後で出し直されます。このクローズは、終局コードを重複して使用しないように、つまり CATCH_ALL クローズ内と ENDTRY の後で二度繰り返して使用しないようにするためのものです。このクローズは通常、クリーンアップ作業を行い、例外の発生に関係なく (つまりブロックでの正常終了と異常終了の両方で) スコープがアンワインドする際にインバリアント (例えば、共有データ、ロック) をリストアするのに使用します。FINALLY クローズは、同じ *try_block* に対しては CATCH や CATCH_ALL クローズと一緒に使用することはできません。*try_block* をネストして使用します。

TRY ブロックを終了するには、ENDTRY ステートメントを使用しなければなりません。このステートメントは、例外が処理されコンテキストがクリーンアップされたことを確かめるために実行するコードを含んでいます。*try_block* や *handler_block*、*finally_block* には、return、ローカルではない jump、あるいは ENDTRY に到達せずにブロックを出る手法 (goto()、break()、continue()、longjmp() など) を含めてはいけません。

このインターフェイスは、RPC オペレーションからの例外を処理するために提供されています。ただしこのインターフェイスは、すべてのアプリケーションで使用することのできる汎用のインターフェイスです。例外は、EXCEPTION の形式で宣言されます (これは複雑なデータ・タイプで、long integer のようには使用できません)。2 種類の例外があります。どちらも同じ方法で宣言されますが、初期化の方法は異なります。

1 番目の例外は、RPC 実行時プリミティブが出すオペレーティング・システムのシグナルと、例外に関連するステータス値を伝播するのに使用されます。ステータス値ごとに例外があらかじめ定義されています (例えば、例外 rpc_x_no_memory が、ステータス rpc_s_no_memory に対して定義されています)。これらの例外は trpcsts.h ヘッダ・ファイルに宣言されています。必須ではありませんが (ステータス例外はあらかじめ定義されています)、ステータス例外はアプリケーションで宣言し、exc_set_status() マクロで初期化できます。このマクロには、初期化する EXCEPTION へのポインタとステータス値が必要です。*status* 例外に関連するステータス値は、exc_get_status() マクロを使用して取得できます。EXCEPTION へのポインタとステータス値が返される変数へのポインタが必要です。*status* 例外であれば、マクロの値は 0 であり、そうでなければ -1 です。

2 番目の例外はアプリケーション例外を定義するのに使用されます。EXCEPTION_INIT() マクロを呼び出して初期化されます。例外のアドレスは *address* 例外内の値として格納されます。この値は単一のアドレス空間内でのみ有効であり、例外が自動変数の場合には変更されることに注意してください。このため *address* 例外は静的変数あるいは外部変数として宣言し、自動変数あるいはレジスタ変数として宣言すべきではありません。exc_get_status() マクロは、*address* 例外では -1 となります。この例外で exc_set_status() マクロを使用すると *status* 例外となります。

exc_matches マクロは、2 つの例外を比較するのに使用します。同一であるためには、どちらの例外も同じタイプであり、同じ値を持たなければなりません (つまり、*status* 例外に対しては同じステータス値を持つか、*address* 例外に対しては同じアドレスを持ちます)。この比較は、上記の CATCH クローズで使用されます。

ステータス例外が発生した場合の通常の処理は、ステータス値を表示するか、より好ましい方法としてはステータス値が何であるかを示す文字列を表示することです。文字列を標準エラー出力に表示するのであれば、文字列を 1 回の操作で表示できるように *status* 例外へのポインタを付加して関数 exc_report() を呼び出します。

```
CATCH_ALL
{
 exc_report(THIS_CATCH);
}
ENDTRY
```

文字列に対して他の処理を行う場合には (例えば、エラーをユーザ・ログに出力する)、exc_get_status() を THIS_CATCH で使用し、ステータス値を取得できます (THIS_CATCH は EXCEPTION に対するポインタであり、EXCEPTION 自体ではありません)。dce_error_inq_text() を使用して、ステータス値の文字列を取得することができます。

```
CATCH_ALL
{
```

```
  unsigned long status_to_convert;
  unsigned char error_text[200];
  int status;

  exc_get_status(THIS_CATCH,status_to_convert);
  dce_error_inq_text(status_to_convert, error_text, status);
  userlog("%s", (char *)error_text);
}
ENDTRY
```

注記 マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、TRY/CATCH インターフェイスを呼び出すことができます。

例外とステータス復帰をいつ使用するか

RPC オペレーションのステータスは、各オペレーションごとにステータス変数を定義することにより決定できます ([comm_status] と [fault_status] パラメータは、ACF (Attribute Configuration File) で定義されます)。ステータス復帰インターフェイスは、X/Open RPC 仕様で提供される唯一のインターフェイスです。fault_status 属性は、不適切なパラメータ、リソース不足、コーディング・エラー等によりサーバ上で発生したエラーが、補助的なステータス引数や戻り値でレポートされることを示しています。同様に comm_status 属性は、RPC コミュニケーション障害が補助的なステータス引数や戻り値でレポートされることを示します。ステータス値を利用する処理は、各コールごとに起こり得るエラーに対して指定されたリカバリを行う (コールごとの) きめ細かいエラー処理と、障害の時点での再試行が必要な場合にはうまく機能します。欠点としては、コールがローカルなものであってもリモートなものであっても、いずれの場合にも透過性がない点が挙げられます。リモート・コールには追加のステータス・パラメータ、あるいは void の代わりにステータスの戻り値が必要になります。従ってアプリケーションでは、ローカルと分散コード間で調整を行うプロシージャ宣言が必要です。

TRY/CATCH 例外復帰インターフェイスは、OSF/DCE 環境からのアプリケーション移植性のためにも提供されています。このインターフェイスは、すべての環境で利用できるものではありません。しかしプロシージャ宣言をローカルと分散コード間で調整する必要がないという利点があり、既存のインターフェイスを保持できます。各プロシージャ・コールは固有の障害チェックやリカバリ・コードを持つ必要がなく、エラー・チェッキングを簡素化できます。あるレベルでエラー処理が行えない場合には、プログラムは、「エラーが検出されたが修正可能である」などのシステム・エラー・メッセージを出力して終了します (メッセージを省略すると、エラー・チェッキングはより簡単になります)。例外は、大まかな例外処理を行う場合には有用です。

組み込み例外

表 13 に示す例外は、この例外インターフェイスで使用するために組み込まれている例外です。最初の TRY クローズはシグナル・ハンドラを設定し、以下にリストされたシグナル (無視されることがなく、補足可能なもの) を補足します。その他の例外は DCE 対応のプログラムでの移植性のためにのみ定義されたものです。

表 13 組み込み例外

Exception	機能説明
exc_e_SIGBUS	処理できない SIGBUS シグナルが発生しました。
exc_e_SIGEMT	処理できない SIGEMT シグナルが発生しました。
exc_e_SIGFPE	処理できない SIGFPE シグナルが発生しました。
exc_e_SIGILL	処理できない SIGILL シグナルが発生しました。
exc_e_SIGIOT	処理できない SIGIOT シグナルが発生しました。
exc_e_SIGPIPE	処理できない SIGPIPE シグナルが発生しました。
exc_e_SIGSEGV	処理できない SIGSEGV シグナルが発生しました。
exc_e_SIGSYS	処理できない SIGSYS シグナルが発生しました。
exc_e_SIGTRAP	処理できない SIGTRAP シグナルが発生しました。
exc_e_SIGXCPU	処理できない SIGXCPU シグナルが発生しました。
exc_e_SIGXFSZ	処理できない SIGXFSZ シグナルが発生しました。
pthread_e_badparam	
pthread_e_defer_q_full	
pthread_e_existence	
pthread_e_in_use	
pthread_e_nostackmem	
pthread_e_nostack	
pthread_e_signal_q_full	
pthread_e_stackovf	

表 13 組み込み例外 (続き)

Exception	機能説明
pthread_e_unimp	
pthread_e_use_error	
exc_e_decovf	
exc_e_exquota	
exc_e_fltdiv	
exc_e_fltovf	
exc_e_fltund	
exc_e_illaddr	
exc_e_insfmem	
exc_e_intdiv	
exc_e_intovf	
exc_e_nopriv	
exc_e_privinst	
exc_e_resaddr	
exc_e_resoper	
exc_e_subrng	
exc_e_uninitexc	

名前の後に "_e" が付加された同じ例外コードが定義されています (exc_e_SIGBUS は、exc_SIGBUS_e とも定義されています)。同等のステータス・コードが同様な名前で定義されていますが、"_e_" は "_s_" に変更されています (exc_e_SIGBUS は、exc_s_SIGBUS ステータス・コードと同等です)。

警告

OSF/DCE では、ヘッダ・ファイルは exc_handling.h と命名されますが、BEA Tuxedo ATMI システムのヘッダ・ファイルは texc.h です。同じソース・ファイルで DCE と BEA Tuxedo ATMI システムの例外処理を両方使用することはできません。また 1 つのプログラム内では、シグナル例外の処理は DCE か BEA Tuxedo ATMI システムのいずれか一方でしか行えません。1 つのプログラム内で BEA Tuxedo ATMI システム /TxRPC スタブと OSF/DCE スタブを統合する方法については、『TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング』を参照してください。

このインターフェイスを用いてプログラムをリンクする場合には、$TUXDIR/lib/libtrpc.a をインクルードしなければなりません。

使用例

例外を使用する C 言語のソース・ファイルを以下に示します。

```
#include <texc.h>

EXCEPTION badopen_e;                        /* 不正な open() に対する例外宣言 */

doit(char *filename)
{
  EXCEPTION_INIT(badopen_e);                /* 例外の初期化 */
  TRY  get_and_update_data(filename);       /* 処理の実行 */
  CATCH(badopen_e)                          /* 例外 - open() が異常終了しました */
     fprintf(stderr, "Cannot open %s\en", filename);
  CATCH_ALL                                 /* 他のエラーの処理 */
     /* 利用できない rpc サービスの処理 , ... */
     exc_report(THIS_CATCH)
   ENDTRY
}
/*
 * 出力ファイルのオープン
 * リモート・データ項目の取得
 * ファイルへの書き込み
 */
get_and_update_data(char *filename)
{
  FILE *fp;
  if ((fp == fopen(filename)) == NULL)      /* 出力ファイルのオープン */
    RAISE(badopen_e);                       /* 例外の生成 */
  TRY
    /* このブロックでは、ファイルは正常にオープンされました -
     * FINALLY を使用してファイルをクローズします
     */
      long data;
      /*
       * RPC コールの実行 -
       * 呼び出し側関数 doit() に対して例外を出します
```

```
       */
      data = remote_get_data();
      fprintf(fp, "%ld\en", data);
  FINALLY
    /* 例外が発生してもしなくても、ファイルをクローズします */
     fclose(fp);
  ENDTRY
}
```

関連項目　tidl(1)、userlog(3c)

UNIX システムのリファレンス・マニュアルの abort(2)

『TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング』

tuxgetenv(3c)

名前

tuxgetenv()— 環境名に対して値を返す

形式

```
#include <atmi.h>
char *tuxgetenv(char *name)
```

機能説明

tuxgetenv() は、環境リストから *name=value* という形式の文字列を探し、その文字列が見つかった場合に、現在の環境内の *value* を指すポインタを返します。文字列が見つからなかった場合は、NULL ポインタを返します。

この関数は、BEA Tuxedo ATMI システムがサポートされる、異なるプラットフォーム間で使用する環境変数に対して、移植可能なインターフェイスを提供します。プラットフォームには、通常は環境変数を持たないプラットフォームも含まれます。

tuxgetenv は大文字と小文字を区別することに注意してください。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、tuxgetenv() の呼び出しを発行できます。

戻り値

文字列のポインタが存在する場合、tuxgetenv() はそのポインタを返します。ポインタが存在しない場合、tuxgetenv() はヌル・ポインタを返します。

移植性

MS Windows では、この関数によって、アプリケーションとダイナミック・リンク・ライブラリとの間で環境変数を共有できるようになります。BEA Tuxedo ATMI/WS DLL は、付加された各アプリケーションの環境コピーを保持します。この関係付けられた環境およびコンテキスト情報は、tpterm() が Windows アプリケーションから呼び出されると無効になります。環境変数の値は、アプリケーション・プログラムが tpterm() を呼び出した後で変更できます。

DOS、Windows、OS/2、および NetWare 環境では、大文字の変数名を使用することをお勧めします (tuxreadenv() はすべての環境変数名を大文字に変換します)。

関連項目

tuxputenv(3c)、tuxreadenv(3c)

tuxputenv(3c)

名前

tuxputenv()— 環境の値を変更、または値を環境に追加する

形式

```
#include <atmi.h>
int tuxputenv(char *string)
```

機能説明

string は、"name=value" という形式の文字列を指すようにします。tuxputenv() は、環境変数名の値を、既存の変数を変更するか、新しい変数を作成することで値と等しくします。どちらの場合も、*string* によって指された文字列は環境変数の一部になります。

この関数は、BEA Tuxedo ATMI がサポートされる、異なるプラットフォーム間で使用する環境変数に対して、移植可能なインターフェイスを提供します。プラットフォームには、通常は環境変数を持たないプラットフォームも含まれます

tuxputenv() は大文字と小文字を区別することに注意してください。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても tuxputenv() の呼び出しを発行できます。

戻り値

tuxputenv() は、malloc() を介して拡張環境のための十分な領域を獲得できなかった場合にゼロでない整数を返します。領域を獲得できた場合はゼロを返します。

移植性

MS Windows では、この関数によって、アプリケーションとダイナミック・リンク・ライブラリとの間で環境変数を共有できるようになります。BEA Tuxedo ATMI/WS DLL は、付加された各アプリケーションの環境コピーを保持します。この関係付けられた環境およびコンテキスト情報は、tpterm() が Windows アプリケーションから呼び出されると無効になります。環境変数の値は、アプリケーション・プログラムが tpterm() を呼び出した後で変更できます。

DOS、Windows、および OS/2 環境では、大文字の変数名を使用することをお勧めします (tuxreadenv() はすべての環境変数名を大文字に変換します)。

関連項目

tuxgetenv(3c)、tuxreadenv(3c)

tuxreadenv(3c)

名前　tuxreadenv()— ファイルから環境へ変数を追加

形式

```
#include <atmi.h>
int tuxreadenv(char *file, char *label)
```

機能説明　tuxreadenv() は環境変数を含むファイルを読み込み、プラットフォームから独立して環境変数を環境に追加します。変数は tuxgetenv() を使用して利用でき、tuxputenv() を使用して再設定できます。

環境ファイルの形式は、次のとおりです。

- 各行の先頭のスペースまたはタブ文字は無視され、次の点では考慮されません。
- 環境に挿入する変数を含む行は次の形式です。
  variable=value
  または
  set *variable=value*
  ここで、*variable* は、先頭がアルファベット文字またはアンダーライン文字で、全体が英数字またはアンダーライン文字のみからなる文字列です。また、*value* には改行文字を除くすべての文字を使用できます。
- *value* 内で、${*env*} という形式の文字列は、環境内にすでにある変数を使用して展開されます。前方参照はサポートされておらず、値が設定されていない場合、変数は空の文字列に置き換えられます。バックスラッシュ (\) を使用して、ドル記号およびそれ自体をエスケープすることができます。その他すべてのシェルのクォートおよびエスケープのメカニズムは無視され、展開された *value* がそのまま環境に入れられます。
- スラッシュ (/)、シャープ (#)、またはエクスクラメーション・マーク (!) で始まる行は、コメントとして扱われ、無視されます。これらのコメント文字、左角かっこ、アルファベット文字またはアンダライン文字以外の文字で始まる行は、将来の用途のため予約されています。使用法は未定義です。
- ファイルは、ラベルとして動作する、左角かっこ ([) で始まる行によってセクションごとに区切られます。ラベルは、31 文字を超えると切り捨てられます。ラベルの形式は次のとおりです。
  [*label*]
  label は、上記の *variable* と同じ規則 (無効な *label* 値を持つ行は無視される) に従います。

- ファイルの先頭と最初のラベルの間の変数行は、すべてのラベルの環境 (つまりグローバル・セクション) に挿入されます。他の変数は、ラベルが、アプリケーション用に指定したラベルに一致する場合にのみ、その環境に挿入されます。ラベル [] はグローバル・セクションを示します。

file が NULL の場合、デフォルトのファイル名が使用されます。固定のファイル名は次のとおりです。

```
DOS、Windows、OS2、NT:C:¥TUXEDO¥TUXEDO.ENV
MAC: システム環境設定ディレクトリ内の TUXEDO.ENV
NETWARE:SYS:SYSTEM\TUXEDO.ENV
POSIX:/usr/tuxedo/TUXEDO.ENV  または  /var/opt/tuxedo/TUXEDO.ENV
```

label が NULL の場合、グローバル・セクション内の変数だけが環境に挿入されます。*label* が他の値の場合、グローバル・セクションの変数と、*label* に一致するセクション内の変数が環境に入れられます。

エラー・メッセージは、メモリ障害が発生した場合、NULL でないファイル名が存在しない場合、または NULL でないラベルがない場合は、`userlog()` に出力されます。

マルチスレッドのアプリケーション中のスレッドは、`TPINVALIDCONTEXT` を含め、どのコンテキスト状態で実行していても、`tuxreadenv()` の呼び出しを発行できます。

使用例

環境ファイルの例を次に示します。

```
TUXDIR=/usr/tuxedo
[application1]
; これはコメントです
/* これはコメントです */
# これはコメントです
// これはコメントです
FIELDTBLS=app1_flds
FLDTBLDIR=/usr/app1/udataobj
[application2]
FIELDTBLS=app2_flds
FLDTBLDIR=/usr/app2/udataobj
```

戻り値

`tuxreadenv()` は、`malloc()` を介して拡張環境のための十分な領域を獲得できなかった場合か、ヌルでない名前を持つファイルをオープンできず読み取ることができない場合に、ゼロ以外の整数を返します。それ以外の場合、`tuxreadenv()` はゼロを返します。

移植性

DOS、Windows、OS/2、および NetWare 環境では、`tuxreadenv()` がすべての環境変数名を大文字に変換します。

関連項目

`tuxgetenv(3c)`、`tuxputenv(3c)`

tx_begin(3c)

名前

tx_begin()— グローバル・トランザクションの開始

形式

```
#include <tx.h>
int tx_begin(void)
```

機能説明

tx_begin() は、呼び出し元の制御スレッドをトランザクション・モードにする際に使用します。呼び出し元のスレッドは、トランザクションを開始する前に、リンクされているリソース・マネージャがオープンしている (tx_open() を介して) ことを、まず第 1 に確実にしなければなりません。呼び出し元がすでにトランザクション・モードにある場合、または tx_open() が呼び出されていない場合には、tx_begin() は ([TX_PROTOCOL_ERROR] を返して) 異常終了します。

トランザクション・モードに入ると、呼び出し元のスレッドは、現在のトランザクションを完了させるために、tx_commit() または tx_rollback() を呼び出さなければなりません。トランザクションの連鎖に関連する条件によっては、トランザクションの開始に明示的に tx_begin() を呼び出す必要がないこともあります。詳細は、tx_commit() および tx_rollback() を参照してください。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tx_begin() の呼び出しを発行できません。

選択可能なセットアップ

tx_set_transaction_timeout()

戻り値

tx_begin() は、正常終了時には、負数ではない戻り値 TX_OK を返します。

エラー

次の条件の場合、tx_begin() は異常終了し、次のいずれかの負の値を返します。

[TX_OUTSIDE]
: 呼び出し元の制御スレッドが、1 つ以上のリソース・マネージャを利用して、グローバル・トランザクションの外部で現在作業中であるため、トランザクション・マネージャはグローバル・トランザクションを開始できません。このような作業がすべて完了してからでなければ、グローバル・トランザクションは開始できません。トランザクションについての呼び出し元の状態は変化しません。

[TX_PROTOCOL_ERROR]
: この関数が不正なコンテキストで呼び出されました (たとえば、呼び出し元がすでにトランザクション・モードにある場合)。トランザクション・モードについての呼び出し元の状態は、変更されません。

[TX_ERROR]
トランザクション・マネージャまたは 1 つ以上のリソース・マネージャが、新しいトランザクションの開始において一時的エラーを検出しました。このエラーが返された場合は、呼び出し元はトランザクション・モードにありません。エラーの正確な内容はログ・ファイルに書き込まれます。

[TX_FAIL]
トランザクション・マネージャまたは 1 つ以上のリソース・マネージャが、致命的エラーを検出しました。このエラーでは、トランザクション・マネージャまたは 1 つ以上のリソース・マネージャ、あるいはその両方は、アプリケーションのために作業を行うことができなくなります。このエラーが返された場合は、呼び出し元はトランザクション・モードにありません。エラーの正確な内容はログ・ファイルに書き込まれます。

関連項目

tx_commit(3c)、tx_open(3c)、tx_rollback(3c)、tx_set_transaction_timeout(3c)

警告

XA 準拠のリソース・マネージャがグローバル・トランザクションに含まれるようにするには、そのリソース・マネージャが正常にオープンされている必要があります (詳細については、tx_open(3c) を参照してください)。X/Open TX インターフェイスと X-Window システムは、いずれも型 XID を定義します。同一のファイルで X-Window コールと TX コールの両方を使用することはできません。

tx_close(3c)

名前

tx_close()— リソース・マネージャ・セットをクローズする

形式

```
#include <tx.h>
int tx_close(void)
```

機能説明

tx_close() は、移植性の高い方法でリソース・マネージャ・セットをクローズします。これにより、トランザクション・マネージャが呼び出されて、リソース・マネージャ固有の情報がトランザクション・マネージャ固有の方法で読み取られ、この情報は呼び出し元がリンクされているリソース・マネージャに渡されます。

tx_close() は、呼び出し元がリンクしているリソース・マネージャをすべてクローズします。この関数は、リソース・マネージャ固有の「クローズ」呼び出しの代わりに使用されるので、アプリケーション・プログラムは、移植性を損なう危険性のある呼び出しを使用することがなくなります。リソース・マネージャは終了の内容がそれぞれで異なるため、個々のリソース・マネージャを「クローズ」するために必要な情報をリソース・マネージャごとに通知しなければなりません。

tx_close() は、アプリケーションの制御スレッドがグローバル・トランザクションに関与する必要がなくなったときに呼び出してください。呼び出し元がトランザクション・モードにあると、tx_close() は ([TX_PROTOCOL_ERROR] を返して) 異常終了します。したがって、現在のトランザクションに関与しない制御スレッドがあっても、リソース・マネージャは一切クローズされません。

tx_close() が正常に終了すると (TX_OK)、呼び出し元のスレッドにリンクしているリソース・マネージャはすべてクローズされます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tx_close() の呼び出しを発行できません。

戻り値

tx_close() は、正常終了時には、負数でない戻り値 TX_OK を返します。

エラー

次の条件の場合、tx_close() は異常終了し、次のいずれかの負の値を返します。

[TX_PROTOCOL_ERROR]
: この関数が不正なコンテキストで呼び出されました (たとえば、呼び出し元がトランザクション・モードにある場合)。リソース・マネージャは一切クローズされません。

[TX_ERROR]
: トランザクション・マネージャまたは 1 つ以上のリソース・マネージャが、一時的エラーを検出しました。エラーの正確な内容はログ・ファイルに書き込まれます。クローズ可能なリソース・マネージャはすべてクローズされます。

[TX_FAIL]
トランザクション・マネージャまたは 1 つ以上のリソース・マネージャが、致命的エラーを検出しました。このエラーでは、トランザクション・マネージャまたは 1 つ以上のリソース・マネージャ、あるいはその両方は、アプリケーションのために作業を行うことができなくなります。エラーの正確な内容はログ・ファイルに書き込まれます。

関連項目 `tx_open(3c)`

警告 X/Open TX インターフェイスと X-Window システムは、いずれも型 XID を定義します。同一のファイルで X-Window コールと TX コールの両方を使用することはできません。

tx_commit(3c)

名前

tx_commit()— グローバル・トランザクションのコミット

形式

```
#include <tx.h>
int tx_commit(void)
```

機能説明

tx_commit() は、呼び出し元の制御スレッドでアクティブなトランザクションの作業をコミットするために使用します。

transaction_control 特性(tx_set_transaction_control(3c) を参照) が TX_UNCHAINED である場合は、tx_commit() が終了すると、呼び出し元はトランザクション・モードではなくなります。一方、*transaction_control* 特性が TX_CHAINED である場合は、tx_commit() が終了すると、呼び出し元は、新しいトランザクションのためにトランザクション・モードのままになります (このページの「戻り値」および「エラー」の項を参照してください)。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tx_commit() の呼び出しを発行できません。

選択可能なセットアップ

- tx_set_commit_return()
- tx_set_transaction_control()
- tx_set_transaction_timeout()

戻り値

tx_commit() は、正常終了時には、負数でない戻り値 TX_OK を返します。

エラー

次の条件の場合、tx_commit() は異常終了し、次のいずれかの負の値を返します。

[TX_NO_BEGIN]
: 現在のトランザクションは、正常にコミットされました。ただし、新しいトランザクションを開始できなかったので、呼び出し元はトランザクション・モードではなくなりました。この戻り値は *transaction_control* 特性が TX_CHAINED である場合のみ発生します。

[TX_ROLLBACK]
: 現在のトランザクションはコミットできず、ロール・バックされました。また、*transaction_control* 特性が TX_CHAINED の場合は、新しいトランザクションが開始します。

[TX_ROLLBACK_NO_BEGIN]
現在のトランザクションはコミットできず、ロール・バックされました。また、新しいトランザクションを開始できなかったので、呼び出し元はトランザクション・モードではなくなりました。この戻り値は *transaction_control* 特性が TX_CHAINED である場合のみ発生します。

[TX_MIXED]
トランザクションのために行われた作業は、部分的にコミットされ、部分的にロールバックされました。また、*transaction_control* 特性が TX_CHAINED の場合は、新しいトランザクションが開始します。

[TX_MIXED_NO_BEGIN]
トランザクションのために行われた作業は、部分的にコミットされ、部分的にロールバックされました。また、新しいトランザクションを開始できなかったので、呼び出し元はトランザクション・モードではなくなりました。この戻り値は *transaction_control* 特性が TX_CHAINED である場合のみ発生します。

[TX_HAZARD]
障害が原因で、トランザクションのために行われた作業は、部分的にコミットされ、部分的にロール・バックされた可能性があります。また、*transaction_control* 特性が TX_CHAINED の場合は、新しいトランザクションが開始します。

[TX_HAZARD_NO_BEGIN]
障害が原因で、トランザクションのために行われた作業は、部分的にコミットされ、部分的にロール・バックされた可能性があります。また、新しいトランザクションを開始できなかったので、呼び出し元はトランザクション・モードではなくなりました。この戻り値は *transaction_control* 特性が TX_CHAINED である場合のみ発生します。

[TX_PROTOCOL_ERROR]
この関数が不正なコンテキストで呼び出されました (たとえば、呼び出し元がトランザクション・モードにない場合)。トランザクション・モードについての呼び出し元の状態は、変更されません。

[TX_FAIL]
トランザクション・マネージャまたは 1 つ以上のリソース・マネージャが、致命的エラーを検出しました。このエラーでは、トランザクション・マネージャまたは 1 つ以上のリソース・マネージャ、あるいはその両方は、アプリケーションのために作業を行うことができなくなります。エラーの正確な内容はログ・ファイルに書き込まれます。トランザクションについての呼び出し元の状態は、不明です。

関連項目　tx_begin(3c)、tx_set_commit_return(3c)、tx_set_transaction_control(3c)、tx_set_transaction_timeout(3c)

警告

X/Open TX インターフェイスと X-Window システムは、いずれも型 XID を定義します。同一のファイルで X-Window コールと TX コールの両方を使用することはできません。

tx_info(3c)

名前

tx_info()— グローバル・トランザクション情報を返す

形式

```
#include <tx.h>
int tx_info(TXINFO *info)
```

機能説明

tx_info() は、グローバル・トランザクション情報を、*info* が指す構造体に返します。また、この関数は、呼び出し元が現在トランザクション・モードにあるかどうかを示す値も返します。*info* が NULL 以外の値であれば、tx_info() は、*info* が指す TXINFO 構造体にグローバル・トランザクション情報を入れます。TXINFO 構造体には次の要素があります。

```
XID                 xid;
COMMIT_RETURN       when_return;
TRANSACTION_CONTROL transaction_control;
TRANSACTION_TIMEOUT transaction_timeout;
TRANSACTION_STATE    transaction_state;
```

tx_info() がトランザクション・モードにおいて呼び出されると、*xid* に現在のトランザクションのブランチ識別子が、*transaction_state* に現在のトランザクションの状態が入ります。呼び出し元がトランザクション・モードにない場合は、*xid* に NULL の XID が入ります (詳細は tx.h ファイルを参照)。また、呼び出し元がトランザクション・モードにあるかどうかに関係なく、*when_return*、*transaction_control* および *transaction_timeout* に、*commit_return* および *transaction_control* 特性の現在の設定、および秒単位のトランザクション・タイムアウト値が入ります。

返されるトランザクション・タイムアウト値は、次のトランザクション開始時に使用される設定を反映しています。したがって、この値は、呼び出し元の現在のグローバル・トランザクションのタイムアウト値を反映しているわけではありません。現在のトランザクション開始後に行われた tx_set_transaction_timeout() の呼び出しによって、この値が変更されていることがあるからです。

info が NULL である場合は、TXINFO 構造体は返されません。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tx_info() の呼び出しを発行できません。

戻り値

呼び出し元がトランザクション・モードにある場合は、1 を返します。呼び出し元がトランザクション・モードにない場合は、0 を返します。

エラー

次の条件の場合、tx_info() は異常終了し、次のいずれかの負の値を返します。

[TX_PROTOCOL_ERROR]
この関数が不正なコンテキストで呼び出されました (たとえば、呼び出し元がまだ tx_open() を呼び出していない場合)。

[TX_FAIL]
トランザクション・マネージャが致命的エラーを検出しました。このエラーでは、トランザクション・マネージャは、アプリケーションのために作業を行うことができなくなります。エラーの正確な内容はログ・ファイルに書き込まれます。

関連項目

tx_open(3c)、tx_set_commit_return(3c)、
tx_set_transaction_control(3c)、tx_set_transaction_timeout(3c)

警告

同一のグローバル・トランザクション内では、後続の tx_info() 呼び出しは、XID に同一の *gtrid* 構成要素を示すことが保証されていますが、同一の *bqual* 構成要素を示すことは必ずしも保証されません。X/Open TX インターフェイスと X-Window システムは、いずれも型 XID を定義します。同一のファイルで X-Window コールと TX コールの両方を使用することはできません。

tx_open(3c)

名前

tx_open()— リソース・マネージャ・セットをオープンする

形式

```
#include <tx.h>
int tx_open(void)
```

機能説明

tx_open() は、移植性の高い方法でリソース・マネージャ・セットをオープンします。これにより、トランザクション・マネージャが呼び出されて、リソース・マネージャ固有の情報がトランザクション・マネージャ固有の方法で読み取られ、この情報は呼び出し元がリンクされているリソース・マネージャに渡されます。

tx_open() はアプリケーションにリンクされているすべてのリソース・マネージャのオープンを試行します。この関数は、リソース・マネージャ固有の「オープン」呼び出しの代わりに使用されるので、アプリケーション・プログラムは、移植性を損なう可能性のある呼び出しを使用することがなくなります。リソース・マネージャは開始の内容がそれぞれで異なるため、個々のリソース・マネージャを「オープン」するために必要な情報をリソース・マネージャごとに通知しなければなりません。

tx_open() が TX_ERROR を返した場合は、リソース・マネージャは一切オープンされません。tx_open() が TX_OK を返した場合は、いくつかまたはすべてのリソース・マネージャがオープンされています。オープンされなかったリソース・マネージャは、アプリケーションによってアクセスされるときに、リソース・マネージャ固有のエラーを返します。tx_open() は、制御スレッドがグローバル・トランザクションに関与する前に、正常に終了していなければなりません。

tx_open() が正常に終了した後で (tx_close() を呼び出す前に)、tx_open() を呼び出すことができます。このような後続の呼び出しは、正常終了しますが、トランザクション・マネージャは、リソース・マネージャの再オープンは一切行いません。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは、tx_open() の呼び出しを発行できません。

戻り値

tx_open() は、正常終了時には、負数でない戻り値 TX_OK を返します。

エラー

次の条件の場合、tx_open() は異常終了し、次のいずれかの負の値を返します。

[TX_ERROR]
トランザクション・マネージャまたは 1 つ以上のリソース・マネージャが、一時的エラーを検出しました。リソース・マネージャは一切オープンされません。エラーの正確な内容はログ・ファイルに書き込まれます。

[TX_FAIL]
トランザクション・マネージャまたは 1 つ以上のリソース・マネージャが、致命的エラーを検出しました。tpinit() を呼び出さずにセキュリティの掛かったアプリケーション (SECURITY APP_PW) の中で tx_open を呼び出すと、TX_FAIL が出されます。このエラーでは、トランザクション・マネージャまたは 1 つ以上のリソース・マネージャ、あるいはその両方は、アプリケーションのために作業を行うことができなくなります。エラーの正確な内容はログ・ファイルに書き込まれます。

関連項目 tx_close(3c)

警告 X/Open TX インターフェイスと X-Window システムは、いずれも型 XID を定義します。同一のファイルで X-Window コールと TX コールの両方を使用することはできません。

tx_rollback(3c)

名前

tx_rollback()— グローバル・トランザクションのロールバック

形式

```
#include <tx.h>
int tx_rollback(void)
```

機能説明

tx_rollback() は、呼び出し元の制御スレッドでアクティブなトランザクションをロールバックするのに使用します。

transaction_control 特性(tx_set_transaction_control(3c) を参照)が TX_UNCHAINED である場合は、tx_rollback() が終了すると、呼び出し元はトランザクション・モードではなくなります。一方、*transaction_control* 特性が TX_CHAINED である場合は、tx_rollback() が終了すると、呼び出し元は新しいトランザクションのためにトランザクション・モードのままになります(このページの「戻り値」および「エラー」の項を参照してください)。

マルチスレッド・アプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tx_rollback() の呼び出しを発行できません。

選択可能なセットアップ

- tx_set_transaction_control()
- tx_set_transaction_timeout()

戻り値

tx_rollback() は、正常終了時には、負数ではない戻り値 TX_OK を返します。

エラー

次の条件の場合、tx_rollback() は異常終了し、次のいずれかの負の値を返します。

[TX_NO_BEGIN]
: 現在のトランザクションは、ロールバックしました。ただし、新しいトランザクションを開始できなかったので、呼び出し元はトランザクション・モードではなくなりました。この戻り値は *transaction_control* 特性が TX_CHAINED である場合のみ発生します。

[TX_MIXED]
: トランザクションのために行われた作業は、部分的にコミットされ、部分的にロールバックされました。また、*transaction_control* 特性が TX_CHAINED の場合は、新しいトランザクションが開始します。

[TX_MIXED_NO_BEGIN]
トランザクションのために行われた作業は、部分的にコミットされ、部分的にロールバックされました。また、新しいトランザクションを開始できなかったので、呼び出し元はトランザクション・モードではなくなりました。この戻り値は *transaction_control* 特性が TX_CHAINED である場合のみ発生します。

[TX_HAZARD]
障害が原因で、トランザクションのために行われた作業は、部分的にコミットされ、部分的にロール・バックされた可能性があります。また、*transaction_control* 特性が TX_CHAINED の場合は、新しいトランザクションが開始します。

[TX_HAZARD_NO_BEGIN]
障害が原因で、トランザクションのために行われた作業は、部分的にコミットされ、部分的にロール・バックされた可能性があります。また、新しいトランザクションを開始できなかったので、呼び出し元はトランザクション・モードではなくなりました。この戻り値は *transaction_control* 特性が TX_CHAINED である場合のみ発生します。

[TX_COMMITTED]
トランザクションのために行われた作業は、ヒューリスティックにコミットされました。また、*transaction_control* 特性が TX_CHAINED の場合は、新しいトランザクションが開始します。

[TX_COMMITTED_NO_BEGIN]
トランザクションのために行われた作業は、ヒューリスティックにコミットされました。また、新しいトランザクションを開始できなかったので、呼び出し元はトランザクション・モードではなくなりました。この戻り値は *transaction_control* 特性が TX_CHAINED である場合のみ発生します。

[TX_PROTOCOL_ERROR]
この関数が不正なコンテキストで呼び出されました (たとえば、呼び出し元がトランザクション・モードにない場合)。

[TX_FAIL]
トランザクション・マネージャまたは 1 つ以上のリソース・マネージャが、致命的エラーを検出しました。このエラーでは、トランザクション・マネージャまたは 1 つ以上のリソース・マネージャ、あるいはその両方は、アプリケーションのために作業を行うことができなくなります。エラーの正確な内容はログ・ファイルに書き込まれます。トランザクションについての呼び出し元の状態は不明です。

関連項目　tx_begin(3c)、tx_set_transaction_control(3c)、tx_set_transaction_timeout(3c)

警告

X/Open TX インターフェイスと X-Window システムは、いずれも型 XID を定義します。同一のファイルで X-Window コールと TX コールの両方を使用することはできません。

tx_set_commit_return(3c)

名前

tx_set_commit_return()—*commit_return* 特性の設定

形式

```
#include <tx.h>
int tx_set_commit_return(COMMIT_RETURN when_return)
```

機能説明

tx_set_commit_return() は、*when_return* で指定されている値に *commit_return* 特性を設定します。この特性は、呼び出し側に制御を返すことに関する tx_commit() の動作に影響します。tx_set_commit_return() は、その呼び出し側がトランザクション・モードかどうかにかかわらず呼び出されます。この設定は、引き続き呼び出される tx_set_commit_return() で変更されるまで有効です。

この特性の初期設定は、TX_COMMIT_COMPLETED です。

when_return の有効な設定を次に示します。

TX_COMMIT_DECISION_LOGGED
: このフラグは、2 フェーズ・コミット・プロトコルの 1 番目のフェーズによってコミットの意志が記録された後、2 番目のフェーズが終了する前に tx_commit() が返らなければならないことを示しています。この設定をすることにより、tx_commit() を呼び出した側へ高速に応答することができます。しかし、トランザクションにヒューリスティックな結果が発生する危険があります。この場合、呼び出し側が、tx_commit() の戻り値からこの状況を知ることはできません。正常な状態では、第 1 フェーズの間にコミットすることを約束しているパーティシパントは、第 2 フェーズでコミットします。しかし、ある異常な環境 (たとえば、応答のないネットワークまたはノード障害) においては、2 番目のフェーズが終了しない可能性があり、ヒューリスティックな結果が発生する場合があります。

TX_COMMIT_COMPLETED
: このフラグは、2 フェーズのコミット・プロトコルが完全に終了した後、tx_commit() が返ることを示しています。この設定により、tx_commit() の呼び出し側は、トランザクションにヒューリスティックな結果が発生したか、または発生した可能性があるかを示す戻り値を調べることができます。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tx_set_commit_return() の呼び出しを発行できません。

戻り値

正常終了の場合、tx_set_commit_return() は、負でない戻り値 TX_OK を返します。

エラー

次の条件の場合、tx_set_commit_return() は、*commit_return* 特性の設定を変更することなく、次に示す 3 つの負の値のうちの 1 つを返します。

[TX_EINVAL]
when_return が、TX_COMMIT_DECISION_LOGGED または TX_COMMIT_COMPLETED のいずれでもありません。

[TX_PROTOCOL_ERROR]
この関数が不正なコンテキストで呼び出されました (たとえば、呼び出し側がまだ tx_open() を呼び出していません)。

[TX_FAIL]
トランザクション・マネージャが致命的エラーを検出しました。このエラーでは、トランザクション・マネージャは、アプリケーションのために作業を行うことができなくなります。エラーの正確な内容はログ・ファイルに書き込まれます。

関連項目

tx_commit(3c)、tx_info(3c)、tx_open(3c)

警告

X/Open TX インターフェイスと X-Window システムは、いずれも型 XID を定義します。同一のファイルで X-Window コールと TX コールの両方を使用することはできません。

tx_set_transaction_control(3c)

名前

tx_set_transaction_control()—*transaction_control* 特性を設定する

形式

```
#include <tx.h>
int tx_set_transaction_control(TRANSACTION_CONTROL control)
```

機能説明

tx_set_transaction_control() は、*control* で指定されている値に *transaction_control* 特性を設定します。この特性は、tx_commit() および tx_rollback() のどちらかが、これらの呼び出し側に返る前に、新しいトランザクションを開始するかどうかを決めます。tx_set_transaction_control() は、アプリケーション・プログラムがトランザクション・モードかどうかにかかわらず呼び出すことができます。この設定は、後に呼び出される tx_set_transaction_control() で変更されるまで有効です。

この特性の初期設定は、TX_UNCHAINED です。

control の有効な設定を次に示します。

TX_UNCHAINED
: このフラグは、tx_commit() および tx_rollback() が、これらの呼び出し側に返る前に新しいトランザクションを開始してはならないことを示しています。呼び出し側は新しいトランザクションを開始するために tx_begin() を出さなければなりません。

TX_CHAINED
: このフラグは、tx_commit() および tx_rollback() が、これらの呼び出し側に返る前に新しいトランザクションを開始することを示しています。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tx_set_transaction_control() の呼び出しを発行できません。

戻り値

正常終了の場合、tx_set_transaction_control() は、負でない戻り値 TX_OK を返します。

エラー

次の条件の場合、tx_set_transaction_control() は、*transaction_control* 特性の設定を変更することなく、次に示す 3 つの負の値のうちの 1 つを返します。

[TX_EINVAL]
: *control* が、TX_UNCHAINED または TX_CHAINED のいずれでもありません。

[TX_PROTOCOL_ERROR]
: この関数が不正なコンテキストで呼び出されました (たとえば、呼び出し側がまだ tx_open() を呼び出していません)。

[TX_FAIL]
トランザクション・マネージャが致命的エラーを検出しました。このエラーでは、トランザクション・マネージャは、アプリケーションのために作業を行うことができなくなります。エラーの正確な内容はログ・ファイルに書き込まれます。

関連項目

tx_begin(3c)、tx_commit(3c)、tx_info(3c)、tx_open(3c)、tx_rollback(3c)

警告

X/Open TX インターフェイスと X-Window システムは、いずれも型 XID を定義します。同一のファイルで X-Window コールと TX コールの両方を使用することはできません。

tx_set_transaction_timeout(3c)

名前

tx_set_transaction_timeout()—*transaction_timeout* 特性を設定する

形式

```
#include <tx.h>
int tx_set_transaction_timeout(TRANSACTION_TIMEOUT timeout)
```

機能説明

tx_set_transaction_timeout() は、*timeout* で指定されている値に *transaction_timeout* 特性を設定します。この値は、トランザクションが、トランザクション・タイムアウトになる前に終了しなければならない時間間隔、すなわちアプリケーションが、tx_begin() を呼び出してから tx_commit() または tx_rollback() を呼び出すまでの時間間隔を指定します。tx_set_transaction_timeout() は、呼び出し側がトランザクション・モードかどうかにかかわらず呼び出すことができます。tx_set_transaction_timeout() が、トランザクション・モードで呼び出される場合、新しいタイムアウト値は、次のトランザクションが開始されないと有効になりません。

transaction_timeout の初期値は、0 (タイムアウトなし) です。

timeout は、トランザクションが、トランザクション・タイムアウトを受けないでいられる秒数を指定します。この値は、最大値、すなわちシステムによって定義されている long までの任意の値に設定されます。*timeout* 値が 0 の場合、タイムアウト機能は働きません。

マルチスレッドのアプリケーションの場合、TPINVALIDCONTEXT 状態のスレッドは tx_set_transaction_timeout() の呼び出しを発行できません。

戻り値

正常終了の場合、tx_set_transaction_timeout() は、負でない戻り値 TX_OK を返します。

エラー

次の条件の場合、tx_set_transaction_timeout() は、*transaction_timeout* 特性の設定を変更することなく、次に示す 3 つの負の値のうちの 1 つを返します。

[TX_EINVAL]
: 指定されたタイムアウト値は無効です。

[TX_PROTOCOL_ERROR]
: この関数が不正に呼び出されましたたとえば、呼び出し元が tx_open() を呼び出す前に呼び出されました。

[TX_FAIL]
: トランザクション・マネージャが、致命的なエラーを見つけました。このエラーでは、トランザクション・マネージャは、アプリケーションのために作業を行うことができなくなります。エラーの正確な内容はログ・ファイルに書き込まれます。

関連項目 `tx_begin(3c)`、`tx_commit(3c)`、`tx_info(3c)`、`tx_open(3c)`、`tx_rollback(3c)`

警告 X/Open TX インターフェイスと X-Window システムは、いずれも型 XID を定義します。同一のファイルで X-Window コールと TX コールの両方を使用することはできません。

userlog(3c)

名前

userlog()—BEA Tuxedo ATMI システムの中央イベント・ログへのメッセージの書き込み

形式

```
#include "userlog.h"
extern char *proc_name;

int userlog (format [ ,arg] . . .)
char *format;
```

機能説明

userlog() は、printf(3S) スタイルのフォーマット指定を受け付け、固定出力ファイル (BEA Tuxedo ATMI システムの中央イベント・ログ) を使用します。

中央のイベント・ログは通常の UNIX システムのファイルで、そのパス名は次のように構成されています。シェル変数 ULOGPFX が設定されている場合、その値がファイル名の接頭辞として使用されます。ULOGPFX が設定されていない場合は、ULOG が使用されます。この接頭辞は最初に userlog() が呼び出されたときに判別されます。userlog() が呼び出されるたびに、日付が判別され、月、日、年が mmddyy の形式で接頭辞に連結されてファイルの名前が設定されます。プロセスが初めてユーザ・ログに書き込む場合、対応する BEA Tuxedo ATMI システムのバージョンを示す追加のメッセージを書き込みます。

このあと、メッセージがそのファイルに追加されます。この方法の場合、それ以降の日に userlog() をプロセスを呼び出すと、メッセージは別のファイルに書き込まれます。

メッセージは、呼び出しプロセスの時刻 (hhmmss)、システム名、プロセス名、呼び出しプロセスのプロセス ID、スレッド ID、コンテキスト ID からなるタグと一緒にログ・ファイルに書き込まれます。このタグの末尾にはコロン (:) が付けられます。プロセスの名前は、外部変数 proc_name のパス名から取られます。proc_name の値が NULL であると、得られる名前は ?proc に設定されます。

BEA Tuxedo ATMI システムのシステムがログ・ファイルに出すエラー・メッセージの先頭には、一意の識別用文字列が次の形式で付けられます。

```
<catalog>:number>:
```

この文字列は、メッセージ文字列を収めている国際化されたメッセージ・カタログの名前にメッセージ番号を加えたものです。規則によれば、BEA Tuxedo ATMI システムのエラー・メッセージは一箇所でのみ使用されるようになっているため、この文字列はソース・コード内の位置を一意に特定します。

format 指定の最後の文字が改行文字でない場合、userlog() は改行文字を 1 つ追加します。

シェル変数 ULOGDEBUG の先頭文字が 1 または y であると、userlog() に送られるメッセージは fprintf(3S) 関数により呼び出しプロセスの標準エラー出力にも書き出されます。

userlog() は、BEA Tuxedo ATMI システムが各種のイベントを記録するために使用します。

この userlog の機構は、BEA Tuxedo ATMI システムのトランザクション記録機構とは完全に独立しています。

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても userlog() の呼び出しを発行できます。

移植性

userlog() インターフェイスは、UNIX システムおよび MS-DOS オペレーティング・システムで利用できます。ログ・メッセージの一部として生成されるシステム名は、MS-DOS システムでは利用できません。このため、MS-DOS システムのシステム名としては、値 PC を使用します。

使用例

変数 ULOGPFX が /application/logs/log に設定されている場合、および userlog() の最初の呼び出しが 1990 年 9 月 7 日に行われた場合、作成されるログ・ファイルには /application/logs/log.090790 という名前が付けられます。たとえば、

```
userlog("UNKNOWN USER '%s' (UID=%d)", usrname, UID);
```

上記のような呼び出しが、プロセス ID が 23431 であるプログラム sec により UNIX システムが指定した m1 上で 4:22:14pm に出され、環境変数 usrname に文字列 "sxx" が含まれ、かつ変数 UID に整数 123 が指定されている場合、ログ・ファイルには次のような行が書き込まれます。

```
162214.m1!sec.23431:UNKNOWN USER 'sxx' (UID=123)
```

プロセスがトランザクション・モードのときにメッセージが中央イベント・ログに送られた場合、ユーザ・ログ・エントリのタグに追加の要素が加わります。これらの要素は、リテラル gtrid と、それに続く 3 桁の long 型の 16 進整数で構成されます。これらの整数はグローバル・トランザクションを一意に示し、グローバル・トランザクション識別子と呼ばれます。この識別子は種に管理上の目的で使用されますが、中央イベント・ログでメッセージの先頭に付けられるタグの中に付けられます。前述のメッセージがトランザクション・モードで中央イベント・ログに書き出される場合、結果として得られるログ・エントリは次のようになります。

```
162214.logsys!security.23431:gtrid x2 x24e1b803 x239:UNKNOWN USER
'sxx' (UID=123)
```

シェル変数 ULOGDEBUG の値が y であると、ログ・メッセージはプログラム security の標準エラーにも書き出されます。

エラー

`userlog()` は、送信されるメッセージが `stdio.h` で定義されている `BUFSIZ` より大きい場合にハングします。

診断

`userlog()` は、出力された文字数を返します。出力エラーがあった場合には、負の値を返します。出力エラーには、現在のログ・ファイルのオープンや書き込みができないといったエラーがあります。`ULOGDEBUG` が設定されている場合、標準エラーへの書き込みができない場合は、エラーとは見なされません。

注意事項

アプリケーションで `userlog()` メッセージを使用する場合には、アプリケーション・エラーをデバッグするのに有用なものだけを使用するようにします。ログが情報であふれてしまうと、本来のエラーを検出するのが難しくなります。

関連項目

UNIX システム・リファレンスマニュアルの `printf`(3S)

Usignal(3c)

名前

Usignal()—BEA Tuxedo ATMI システム環境でのシグナル処理

形式

```
#include "Usignal.h"

UDEFERSIGS()
UENSURESIGS()
UGDEFERLEVEL()
URESUMESIGS()
USDEFERLEVEL(level)

int (*Usignal(sig,func)()
int sig;
int (*func)();

void Usiginit()
```

機能説明

BEA Tuxedo ATMI システム・ソフトウェアが提供する多くの機能は、共有メモリ内のデータ構造を並行アクセスする必要があります。共有データ構造をアクセスするプログラムはユーザ・モードで走るため、シグナルを使用して中断することができます。共有データ構造の一貫性を維持するためには、それらの構造をアクセスする類の操作が、UNIX システムのシグナルを受け取っても中断されないことが重要です。ここで説明する関数は、ほとんどの一般的なシグナルに対する保護機構を提供するもので、BEA Tuxedo ATMI システムのコードの多くがその内部で使用します。また、これらの関数は、アプリケーション側で使用して、不用意にシグナルを受け取らないようにします。

この BEA Tuxedo ATMI システムのシグナル処理パッケージは、重要なコード部ではシグナルの発行を遅らせる、という発想から生まれたものです。このため、シグナルは、それが受信されてもすぐには処理されません。BEA Tuxedo ATMI システムのルーチンはまず最初に送信されたシグナルを捕捉します。そのシグナルを処理しても安全であれば、そのシグナルに指定された処理が実行されます。シグナルが安全でないということが判明した場合には、そのシグナルの到着が通知されるだけで、重要なコード部が終了したことをユーザが示すまでは、そのシグナルの処理は行われません。

マルチスレッド・プログラムでシグナルを使用することは可能ですが、使用しないことをお勧めします。ただしシグナルを使用した場合には、マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても Usignal() の呼び出しを発行できます。

シグナルの捕捉

rmopen() または tpinit() を使用するユーザ・コードは、Usignal() 関数を使用してシグナルを捕捉するようにします。Usignal() は UNIX の signal() システム・コールのように機能しますが、例外として Usignal() では最初に、内部ルーチンがシグナルを捕捉してから、ユーザ・ルーチンをディスパッチするように調整します。

シグナルの遅延とリストア

ここで説明する呼び出しは、アプリケーション・コードでシグナルを後回しにする必要がある場合にのみ使用します。一般に、これらのルーチンは、不適切な場面でシグナルが到着しないようにするために BEA Tuxedo ATMI システムのルーチンが自動的に呼び出します。

シグナルを遅延またはリストアする前に、メカニズムを初期化する必要があります。初期化は、BEA Tuxedo ATMI システム・クライアントが tpinit() を呼び出したときに、BEA Tuxedo ATMI システムのクライアントとサーバのために自動的に行われます。また、初期化は、アプリケーションがはじめに Usignal() 呼び出すときに行われます。初期化は、Usiginit() を呼び出すことで明示的に行うことができます。

UDEFERSIGS() マクロは、重要なコード部分に入るときに使用してください。UDEFERSIGS() が呼び出された後、シグナルは保留状態になります。また、URESUMESIGS() マクロは、その重要な部分が終わる時点で呼び出すようにします。シグナル遅延スタックに注意してください。このスタックは、最初にゼロに設定されるカウンタを通して実現されます。UDEFERSIGS() への呼び出しによってシグナルが遅延されると、カウンタの値が 1 大きくなります。URESUMESIGS() の呼び出しによりシグナルが再開されると、カウンタの値は 1 小さくなります。カウンタがゼロ以外の値のときにシグナルが到着すると、そのシグナルの処理は後回しになります。シグナルが到着したときにカウンタがゼロであれば、そのシグナルはただちに処理されます。シグナルの再開により、カウンタがゼロになると (すなわち、再開の前のカウンタ値が 1 のとき)、その据置期間に到着したシグナルが処理されます。一般に、UDEFERSIGS() の呼び出しは、URESUMESIGS() の呼び出しと対で使用するようにします。

UDEFERSIGS は遅延カウンタの値を増分しますが、返す値はその増分前の値です。マクロ UENSURESIGS() は、遅延カウンタを明示的にゼロに設定する (そして、据え置かれたシグナルを強制的に処理させる) ときに使用できます。この場合、ユーザは UDEFERSIGS() と URESUMESIGS() の不一致が生じないようにする必要があります。

関数 UGDEFERLEVEL() は遅延カウンタの現在の設定値を返します。マクロ USDEFERLEVEL(level) を使用すると、特定の遅延レベルを設定することができます。UGDEFERLEVEL() と USDEFERLEVEL() は、setjmp/longjmp の状況で適切にカウンタを設定するときに便利です。この場合、遅延 / 再開の組み合わせは迂回されます。つまり、setjmp() が呼び出されたときに、UGDEFERLEVEl() の呼び出しによってカウンタの値を保存し、longjmp() が実行されたときに USDEFERLEVEL() の呼び出しによりカウンタ値をリストアする、という考え方です。

注意事項

Usignal は、SIGHUP、SIGINT、SIGQUIT、SIGALRM、SIGTERM、SIGUSR1、および SIGUSR2 の各シグナルについて遅延処理を行います。その他すべてのシグナル番号に対する処理要求は、Usignal() により直接 signal() 関数に渡されます。シグナルは非常に長い期間その処理が据え置かれることがあります。このため、シグナル遅延の間、シグナルの到着がすべてカウントされます。何回も到着する可能性のあるシグナルの処理が安全な場合、そのシグナルの処理ルーチンが繰り返し呼び出され、シグナルが到着した各シグナルが処理されます。各呼び出しの前には、シグナルのデフォルトの処理がなされます。つまり、安全なコードで連続して処理が行われる場合と同様に、据え置かれたシグナルが処理されるようにする、という考え方です。

一般に、ユーザは signal() と Usignal() の呼び出しを同じシグナルに対して組み合わせて使用するべきではありません。できれば、Usignal() を使用する方法をとることが推奨されます。これによって、常にシグナルの状態を知ることができるからです。Usignal は、アプリケーションが BEA Tuxedo ATMI システム・サービスでアラームを使用したい場合などには必要です。こうすることにより、Usiginit() は、シグナルの遅延メカニズムを初期化するために呼び出されなければなりません。次に、signal() を呼び出し、意図するシグナル用に、メカニズムを変更します。シグナルの遅延メカニズムをリストアするために、Usignal() を SIG_IGN で呼び出してから、再び、意図するシグナル処理関数で呼び出す必要があります。

シェル変数 UIMMEDSIGS を使用すれば、シグナルの据置を変更することができます。この変数の値が次のように英字 y で始まる場合、

```
UIMMEDSIGS=y
```

シグナルは Usignal() コードでインタセプトされません (したがって、据え置かれません)。このような場合、Usignal() の呼び出しはただちに signal() に渡されます。

また、Usignal は DOS オペレーティング・システムの配下では役に立ちません。

ファイル

Usignal.h

関連項目

UNIX システムのリファレンス・マニュアルの signal(2)

Uunix_err(3c)

名前

Uunix_err()—UNIX システム・コール・エラーの出力

形式

```
#include Uunix.h

void Uunix_err(s)
char *s;
```

機能説明

BEA Tuxedo ATMI システムの関数が UNIX システム・コールを呼び出したときに、そのシステム・コールがエラーを検出すると、エラーが返されます。外部整数 Uunixerr() が、そのエラーを返したシステム・コールを示す値 (Uunix.h に定義されています) に設定されます。さらに、このシステム・コールは、errno() を失敗した理由を示す値 (errno.h に定義されています) に設定します。

Uunix_err() 関数は、BEA Tuxedo ATMI システムの関数の呼び出し中に最後に検出したシステム・コールを示すメッセージを標準エラー出力に書き出します。この関数は引数を 1 つ (文字列) をとります。この関数はこの引数文字列に続いて、コロンと空白、失敗したシステム・コールの名前、失敗の理由、そして改行文字を書き出します。様々な利用形態を考慮して、この引数文字列には、エラーを出したプログラムの名前も含めておくようにしてください。システム・コールのエラー番号は外部変数 Uunixerr() からとられ、そのエラーの理由は errno() から取り込まれます。どちらの変数も、エラーが発生した時点で設定されます。これらの変数はエラーのない呼び出しが行われたときにクリアされます。

メッセージの多彩な形式を単純化するため、次のようなメッセージ文字列の配列

```
extern char *Uunixmsg[];
```

が提供されます。Uunixerr() は、このテーブルへのインデックスとして使用し、失敗したシステム・コールの名前を (改行文字なしで) 取り込むことができます

マルチスレッドのアプリケーション中のスレッドは、TPINVALIDCONTEXT を含め、どのコンテキスト状態で実行していても、Uunix_err() の呼び出しを発行できます。

使用例

```
#include Uunix.h
extern int Uunixerr, errno;

  ..........
   if((fd=open("myfile", 3, 0660)) == -1)
       {
       Uunixerr = UOPEN;
       Uunix_err("myprog");
```

```
exit(1);
}
```