実践テクニックをご紹介

徒然PostgreSQL散策

第8回 PostgreSQLとネットワークプログラミング（2）

日本PostgreSQLユーザ会　石井 達夫
ISHII Tatsuo●ishii@postgresql.jp

はじめに

前回はpgpoolのソースを使って，以下のようなネットワークプログラミングの基礎を解説しました．

- クライアント／サーバモデルとは
- ソケットインタフェースとは
- サーバ側でのソケットの使い方
- pre-fork方式とは
- 非ブロックソケットとは

ここまでで一応，サーバがネットワーク経由でクライアントからの接続を受け入れるところまで，解説が終わっています．

今回はその続きということで，クライアントがサーバに接続するところから始めたいと思いますが，その前に，解説の対象となるpgpoolのバージョンが2.0に上がったのでお知らせします．

pgpool 2.0について

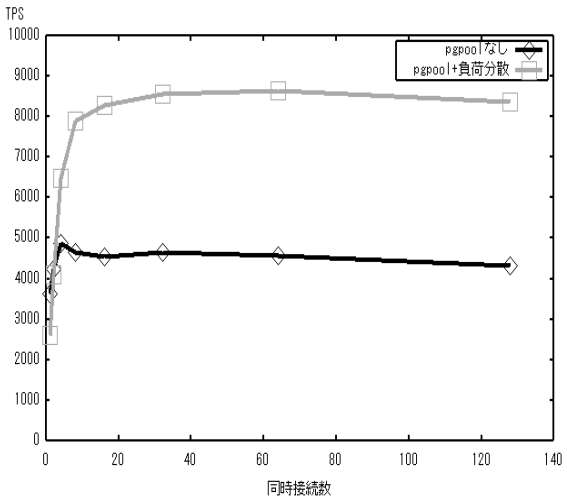
pgpoolはバージョン1でコネクションプールとレプリケーションをサポートしましたが，2.0ではついに負荷分散機能が使えるようになりました．負荷分散とは，一般に複数のサーバで処理を分担しあってトータルでの性能を向上させる手法を指します．たとえばWebサーバを複数並べ，クライアントからの要求をそれぞれに割り振ってより多くのリクエストをこなすようにするのは大規模なWebサイトではごく普通に行われています．同じように，pgpool 2.0（以下，単にpgpoolと呼称します）ではマスタとセカンダリにランダムに要求を割り振り，単位時間あたり，より多くのDBアクセスをこなすことができるようにしています．

もちろん，更新を伴うSQLを負荷分散させてはマスタとセカンダリのデータベースの内容が一致しなくなってしまいますので，pgpoolではSQL文が「SELECT」で始まるときだけ負荷分散するようにしています[注1]．

では早速その効果を見てみましょう．ベンチマークプログラムであるpgbenchに-Sオプションを渡し，検索処理のみを実行させた結果が図1です．ご覧のように，単体のPostgreSQLに比べほぼ2倍近い性能が得られています．

これ以外にも，V3プロトコル（PostgreSQL 7.4以

図1◎検索処理をpgpoolで負荷分散させたときの効果



注1）もちろん「select」でも大丈夫です．ただしpgpoolは，更新副作用を含むSELECT文かどうかの判定まではしてくれないので，そういう場合はpgpoolに負荷分散をしない旨の指示を与えなければなりません．

降で採用されている通信プロトコル）に対応し，PostgreSQL 7.4と一緒に使ったときに機能の制約がなくなったとか，効率がアップしたなどの改良が加えられています．

connect( )でサーバに接続

前置きが長くなりましたが，さっそく前回の続きから始めましょう．

クライアントがサーバに接続するには，connect( )というシステムコールを使います．本稿執筆時点の最新版であるpgpool 2.0.5では，pool_connection_pool.cの299行目辺りにあるconnect_inet_domain_socket( )という関数がconnect( )を呼び出しています（リスト1）．

クライアントからサーバに接続する際もまずsocket( )システムコールを使ってソケットを作ります（①）．

▼リスト1　pgpoolの中でconnect( )を呼び出しているところ

```
int connect_inet_domain_socket(int secondary_backend)
{
    int fd;
    int len;
    int on = 1;
    struct sockaddr_in addr;
    struct hostent *hp;
    int port;
    char *host;

    fd = socket(AF_INET, SOCK_STREAM, 0); ――①
    if (fd < 0)
    {
        pool_error("connect_inet_domain_socket: socket() failed: %s", strerror(errno));
        return -1;
    }

    /* set nodelay */
    if (setsockopt(fd, IPPROTO_TCP, TCP_NODELAY, ――②
                   (char *) &on,
                   sizeof(on)) < 0)
    {
        pool_error("connect_inet_domain_socket: setsockopt() failed: %s", strerror(errno));
        close(fd);
        return -1;
    }

    memset((char *) &addr, 0, sizeof(addr));
    ((struct sockaddr *)&addr)->sa_family = AF_INET; ――③

    host = secondary_backend?pool_config.secondary_backend_host_name:pool_config.current_backend_host_name;

    port = secondary_backend?pool_config.secondary_backend_port:pool_config.current_backend_port;
    addr.sin_port = htons(port); ――④
    len = sizeof(struct sockaddr_in);

                                                                                    ⑤
    hp = gethostbyname(host); ――――――――――――――――――――――――――――――――――――――――――――――――
    if ((hp == NULL) || (hp->h_addrtype != AF_INET))
    {
        pool_error("connect_inet_domain_socket: gethostbyname() failed: %s host: %s", strerror(errno), host);
        close(fd);
        return -1;
    } ――――――――――――――――――――――――――――――――――――――――――――――――――――――
    memmove((char *) &(addr.sin_addr),
            (char *) hp->h_addr,
            hp->h_length);

    if (connect(fd, (struct sockaddr *)&addr, len) < 0) ――⑥
    {
        pool_error("connect_inet_domain_socket: connect() failed: %s",strerror(errno));
        close(fd);
        return -1;
    }
    return fd;
}
```

socket( )については前回説明しました.

作成したソケットにはsetsockopt( )でオプションを与えることができます．ここでは，TCP_NODELAYというオプションを指定しています（②）．これを指定しないと，カーネルは小さなTCPのパケットをできるだけまとめて送信しようとするため，PostgreSQLの通信プロトコルに支障をきたすことがあります.

次にconnect( )を使って目的のサーバに接続します．connect( )の第1引数はsocket( )の返すファイルディスクリプタです．第2引数は接続するサーバを指定するための構造体sockaddrです．前回述べたように，TCP/IP接続の場合は実際にはsockaddr_inという構造体に値をセットし，connect( )への引数としてはsockaddr *にキャストしています．sockaddr_inには前回説明したとおり，アドレスファミリ（③）とアドレスを指定します．アドレスは前回はINADDR_ANYを指定していましたが，クライアントがサーバに接続する際には相手のサーバをポート番号とホスト名（またはIPアドレス）ではっきり指定しなければなりません.

ポート番号はpostmasterの待ち受けポート番号をhtons( )を使ってネットワークバイトオーダーに変換して設定します（④）.

文字列のホスト名はそのままconnect( )に渡すことはできないので，gethostbyname( )を使ってIPアドレスに変換します（⑤）.

こうして作成した引数をconnect( )に渡します（⑥）.成功すればファイルディスクリプタが返却されます．このファイルにディスクリプタにread( )やwrite( )を使ってデータの読み書きを行えばそのままサーバとデータの送受信ができます.

read( )/write( )の注意事項

こうしてread( )やwrite( )を使ってサーバと通信ができるようになったとはいえ，普通のファイルの読み書きとはまた違った考慮がネットワーク通信では必要です.

●必ずエラーチェックをする

ネットワーク上の通信では，回線が物理的に切断したり，あるいはサーバがダウンするなど，いつ何時通信路が閉鎖されるかわかりません．きちんとエラーチェックを行うことが必要です．なお，切断したソケットに書き込みを行うとSIGPIPEというシグナルが発生し，プロセスが強制終了させられますが，後始末をなどを考えるとあまり都合の良い動作ではありません．SIGPIPEを無視する設定をし，read( )やwrite( )でエラーを検知した際にきめ細かなエラー処理をするようにしたほうが良いでしょう.

●readで指定バイト数読み出せなくてもエラーではない

ソケットをread( )すると指定したバイト数が読み出せないことがありますが，これはエラーではありません．このような場合は，取得できなかったバイト数を再度読み出さなければなりません.

●read( )やwrite( )はできるだけ節約

通常のファイルへの書き込みと違って，ソケットへのread( )やwrite( )は非常に遅くなります．これはどちらかというと，データ量よりもシステムコールの発行回数が問題になります．また，read( )に関してはカーネルのバッファリングがあるので，せいぜいシステムコールを呼び出すオーバヘッドが問題になるくらいですが，write( )のオーバヘッドは深刻です．バッファリング機能を持つ標準入出力ライブラリを使う手もありますが，read( )の際に前もってデータが届いているかどうかをチェックできないのが困りものです．そこでpgpoolでは，自前のread( )やwrite( )のラッパー関数を書いています[注2].

pgpoolにおけるラッパー関数の実装

ソケットインタフェースのaccept（サーバの場合）やconnect（クライアントの場合）が返すファイルディスクリプタは全二重通信，すなわち1本のストリームで読み出しも書き込みもできます．前述の理由で読

注2）PostgreSQLでもそのような実装になっています.

み出しに関しては独自のバッファリングを実装しますが，書き込みに関しては標準入出力関数を使わない理由は特にないので，ソケットのファイルディスクリプタからfdopen( )を使って標準入出力関数が利用できる形式のファイルディスクリプタを作成します．図2にpgpoolでの入出力関数の構造を示します．

提供しているラッパー関数

pgpoolのラッパー関数はすべてpool_stream.cにまとめてあります．提供しているインタフェースは以下のものです．

●POOL_CONNECTION *pool_open(int fd)

ソケットインタフェースの返したファイルディスクリプタから書き込み用の標準入出力インタフェースのファイルディスクリプタを生成し，POOL_CONNECTION構造体にセットして返却します．

POOL_CONNECTION構造体はpool.hにリスト2のように定義されており，pgpoolのストリーム入出力関数はすべてこの構造体をインタフェースとして使用します．

●void pool_close(POOL_CONNECTION *cp)

pool_open( )で開いたストリームを閉じます．

図2◎pgpoolのストリーム入出力関数の構造

▼リスト2　POOL_CONNECTION構造体

```
typedef struct {
    int fd;                     /* fd for connection */
    FILE *write_fd;             /* stream write connection */

    char *hp;                   /* pending data buffer head address */
    int po;                     /* pending data offset */
    int bufsz;                  /* pending data buffer size */
    int len;                    /* pending data length */

    char *sbuf;                 /* buffer for pool_read_string */
    int sbufsz;                 /* its size in bytes */

    char *buf2;                 /* buffer for pool_read2 */
    int bufsz2;                 /* its size in bytes */

    int isbackend;              /* this connection is for backend if non 0 */
    int issecondary_backend;    /* this connection is for secondary backend if non 0 */

    char tstate;                /* transaction state (V3 only) */

    /*
     * following are used to remember when re-use the authenticated connection
     */
    int auth_kind;                    /* 3: clear text password, 4: crypt password, 5: md5 password */
    int pwd_size;                     /* password (sent back from frontend) size in host order */
    char password[MAX_PASSWORD_SIZE]; /* password (sent back from frontend) */
    char salt[4];                     /* password salt */

    /*
     * following are used to remember current session paramter status.
     * re-used connection will need them (V3 only)
     */
    ParamStatus params;

    int no_forward;  /* if non 0, do not write to frontend */

} POOL_CONNECTION;
```

●int pool_read(POOL_CONNECTION *cp, void *buf, int len)

指定したバイト数分ストリームから読み込み，bufに返します．POOL_CONNECTION構造体の，hp，po，bufsz，lenを使って内部的にバッファリングをしており，なるべく読み込み回数を減らすようにしています．このとき読み込みすぎたデータは「ペンディングデータ」として内部的に扱い，次回のpool_read( )の呼び出しで利用されます．

●char *pool_read2(POOL_CONNECTION *cp, int len)

指定したバイト数分ストリームから読み込み，内部的な読み込みバッファのアドレスを返します．pool_read( )との違いは，lenで指定したバイト数分だけしか読み込みを行わないことです．したがって，バイト数が少なく，しかも固定長の読み込みにはpool_read( )，そうでない場合にはpool_read2( )を使うようにします．

●int pool_write(POOL_CONNECTION *cp, void *buf, int len)

fwrite( )を使ってストリームに書き込みます．

●int pool_flush(POOL_CONNECTION *cp)

pool_write( )は標準入出力ライブラリのバッファに書き込むだけですが，pool_flush( )はfflush( )を呼び出し，実際にネットワークにデータが送出されることを保証します．

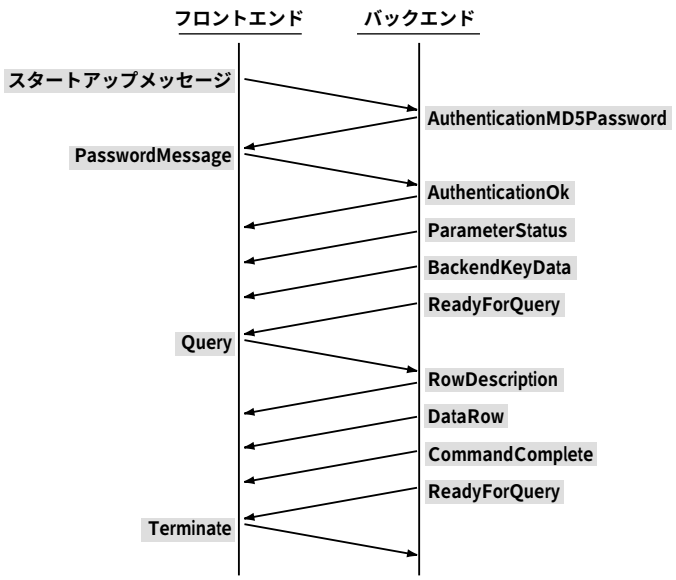
●int pool_write_and_flush(POOL_CONNECTION *cp, void *buf, int len)

pool_write( )とpool_flush( )のコンビネーションです．

●char *pool_read_string(POOL_CONNECTION *cp, int *len, int line)

改行またはNULLが来るまでストリームから文字列を読み込みます．V3プロトコルでは使用されません．

図3◎フロントエンドとバックエンドの通信のシーケンス図



V3プロトコルによる通信処理とは

では実際にPostgreSQLがどのようにしてフロントエンドとバックエンドの間で通信を行っているかを見てみましょう[注3]．

接続の開始から終了までのおおまかな流れは図3のようになります．このあたりの詳細な処理は，pgpoolではpool_process_query.cに書いてあるので，興味がある方はソースコードをご覧ください．

以下，処理概要を各ステップごとに解説します．

注3）もっと詳しいことを知りたい方は，PostgreSQL付属マニュアルの「Chapter 44. Frontend/Backend Protocol」，あるいは日本PostgreSQLユーザ会が配布する日本語版の「第44章フロントエンド／バックエンドプロトコル」をご覧ください．

スタートアップ

フロントエンドから図4のような構造を持つ「スタートアップメッセージ」が送られてきます.

最初の4バイトは,自身を含むメッセージ全体のバイト数です.次の4バイトはこの通信プロトコルのバージョンで,頭16ビットが「メジャーバージョン」,V3なら3です.後の16ビットは「マイナーバージョン」で,今のところ0です.この後項目名文字列とその値文字列のペアが続きます.図ではuserの次にdatabaseが来ていますが,この順番は保証されていません.databaseやoptionsは省略可能です.databaseが省略された場合はユーザ名と同じ名前のデータベースを使うものと見なされます.

pgpoolではスタートアップメッセージの処理はchild.cに定義されたread_startup_packet()という関数に記述されています.

バックエンド側では,データベース名やpg_hba.confを参照してフロントエンドからの接続要求を受け入れるかどうかを決定します.もし問題なければ次のステップに移ります.

認証処理

もしTRUST認証,すなわちパスワード設定がされていない場合は,図5の「AuthenticationOkメッセージ」がフロントエンドに送信されてきます.

頭1バイトは「R」という文字で,認証要求を表します.次の4バイトは自身を含むメッセージのバイト数です.このように,頭1バイトがメッセージの種類で,次の4バイトがメッセージの長さになっているのはほかのメッセージも共通です(ただし,スタートアップメッセージを除く).そして最後が4バイトの0です.

パスワードのやり取りがある場合

パスワードが必要なければ次のステップに進むことができますが,MD5認証などではパスワードをやり取りしなければなりません.この場合図6の「AuthenticationMD5Passwordメッセージ」がフロントエンドに送られてきます.

AuthenticationMD5Passwordメッセージが送られてきたら,返答として図7の「PasswordMessage」をバックエンドに返却します.

MD5認証の場合,パスワードを生のままで送るのではなく,AuthenticationMD5Passwordメッセージに含まれるソルトを使ってMD5メッセージ化したものを送るようにします.こうすれば,ネットワーク上の盗聴にもある程度対処できます.

パラメータステータス

もしパスワードがOKならば,AuthenticationOkメ

図4◎スタートアップメッセージの構造

スタートアップメッセージのバイト数	
プロトコルバージョン番号(4バイト)	
“user”	ユーザ名
“database”	データベース名
“options”	optionの値
メッセージの終端(0x00)	

図5◎AuthenticationOkメッセージ

'R'
メッセージのバイト長(8)
認証OK(0)

図6◎AuthenticationMD5Passwordメッセージ

'R'
メッセージのバイト長(12)
MD5パスワード要求(5)
パスワード暗号化ソルト

図7◎PasswordMessageメッセージ

'p'
メッセージのバイト長
パスワード文字列

ッセージがバックエンドから送られてきます．そして続いて，「パラメータステータス」(Parameter Status：図8)が送られてきます．

パラメータステータスとは，そのセッションの設定値を示す値です．SETコマンドで設定するとこのメッセージが送られてきますが，セッションの開始にもセッションのデフォルト値がまとめて送られてきます．今のところ，表1のようなパラメータステータスが送信されてきます．

秘密鍵

次は「BackendKeyDataメッセージ」です（図9）．

バックエンドから送られてくるこのメッセージには，バックエンドのプロセスIDと「秘密鍵」が含まれます．フロントエンドは，後で実行中の問い合わせを中断したくなったらこのプロセスIDと秘密鍵を送信しなければなりません．このようにして，誰もが問い合わせをキャンセルすることを防いでいます．

図8◎ParameterStatusメッセージ

'S'
メッセージのバイト長
パラメータの名前（文字列）
その値（文字列）

図9◎BackendKeyDataメッセージ

'B'
メッセージのバイト長（12）
バックエンドのプロセスID
秘密鍵

ReadyForQueryメッセージ

すべてがOKならば，最後の締めくくりにバックエンドから「ReadyForQueryメッセージ」が送られてきます（図10）．このメッセージを受け取るまでは，フロントエンドは問い合わせを送信してはいけません．ReadyForQueryはスタートアップ時だけでなく，1つの問い合わせ処理が終わるたびにバックエンドから送信されてきます．

ReadyForQueryメッセージの重要な役割は，現在のトランザクションの状態を報告することです．状態は1バイトの文字で報告されます（表2）．

問い合わせの種類

PostgreSQLが受け付ける問い合わせにはいろいろな種類があります（表3）．

ここでは「簡易問い合わせ」と呼ばれる問い合わせに対してただちに結果が返るタイプのものを説明します．例題として使う問い合わせは「SELECT 1」という極めて単純なものです．

図10◎ReadyForQueryメッセージ

'Z'
メッセージのバイト長（5）
バックエンドのトランザクション状態

表2◎ReadyForQueryメッセージの状態

●状態文字	●説明
I	アイドル状態
T	トランザクションブロック内
E	エラー中のトランザクション

表1◎パラメータステータス

●名前	●典型的な値	●説明
client_encoding	EUC_JP	クライアントのエンコーディング
DateStyle	ISO，MDY	日付データ形式
is_superuser	off	スーパユーザかどうか
server_version	7.4.3	PostgreSQLバックエンドのバージョン
session_authorization	t-ishii	セッションユーザ名

● SELECT文の発行

問い合わせは，「Queryメッセージ」をフロントエンドが発行するところから始まります（図11）.

● RowDescriptionメッセージ

首尾よく問い合わせが実行されると，まず結果のデータの構造を表す「RowDescriptionメッセージ」（図12）がバックエンドから送られてきます．このメッセージは少々複雑ですが，検索結果のデータに関する詳細な情報を含んでいます.

メッセージタイプT，メッセージのバイト長の後にまず結果の列数があります．後のフィールドはその数分だけの情報があります.

● 列名

列の名前です．この例ではテーブルからのデータの取得ではないため，「?column」となっています.

● テーブルOID

テーブルからのデータ取得の場合にはそのテーブルのOID，そうでなければ0です.

● このデータの型に対応するOID

この例では整数（int4）で，PostgreSQL 7.4では23となります.

● このデータのバイト数によるサイズ

この例では4です．可変長のデータでは1となります.

● 型修飾子

データ型によって異なりますが通常0です.

● 書式コード

テキスト形式で結果が帰る場合は0，バイナリで返る場合は1で，この例では0です.

● DataRowメッセージ

RowDescriptionの次は結果の行数分だけ「DataRowメッセージ」（図13）が返ります.

なお，データがNULLの場合は，データ長は-1，その後に続くはずの「データ」はありません.

表3◎問い合わせの種類

●形式	●説明
簡易問い合わせ	問い合わせに対してただちに結果が返る
拡張問い合わせ	Parse/Bind/Executeによって問い合わせを実行
COPY	COPYコマンドで使用
関数呼び出し	指定関数の実行

図11◎Queryメッセージ

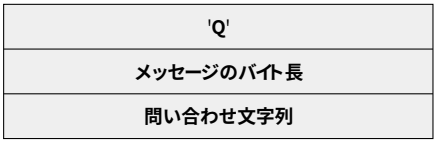


図12◎RowDescriptionメッセージ

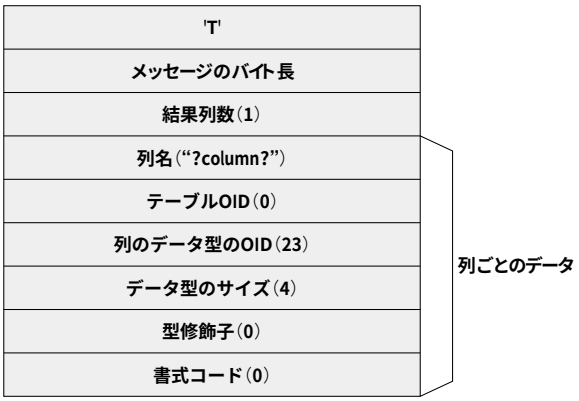


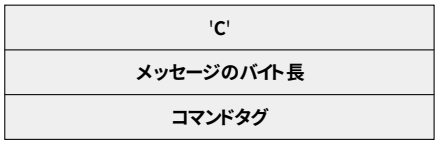
図13◎DataRowメッセージ

● CommandCompleteメッセージ

問い合わせ結果が無事に返ると，次に「CommandCompleteメッセージ」（図14）がバックエンドから送られてきます.

「コマンドタグ」はどのSQLコマンドが完了したかを

図14◎CommandCompleteメッセージ



表す文字列です．たとえばSELECTなら「SELECT」，VACUUMなら「VACUUM」となります．INSERTコマンドなどでは挿入された行数，行にアサインされたOIDなどの付加情報が付与されます．

●ErrorResponseメッセージ

ここでは間違いが起きようもないSELECT文ですが，万が一エラーが起きたらどうなるのでしょう？そういう場合は「ErrorResponseメッセージ」（図15）が返ります．

フィールド識別コードには表4のものがあります．

再びReadyForQueryメッセージ

こうして1つの問い合わせが処理されると，バックエンドは再びReadyForQueryメッセージを送信し，問い合わせを受け付けることができるようになります．

図16◎Terminateメッセージ

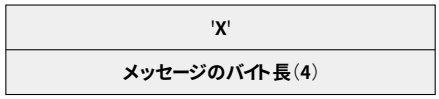
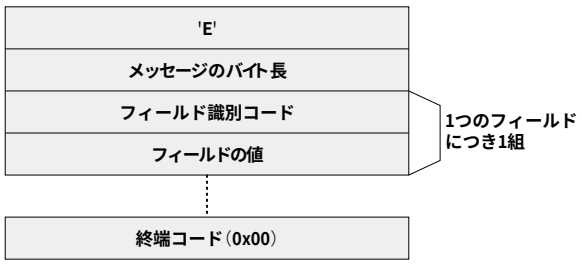


図15◎ErrorResponseメッセージ



接続の終了

接続を終了したい場合は，「Terminateメッセージ」（図16）をフロントエンドからバックエンドに送信します．

最後に

2回にわたってネットワークプログラミングについて解説しました．ネットワークプログラミングは慣れないとなかなかわかりにくい面もありますが，プログラミングのテクニックを磨く上では最高の題材だと思います．また，今の世の中ネットワークは避けて通れない存在です．

しかし，最初から難しく考える必要はありません．実はpgpoolも最初は300行ほどの小さなプログラムから出発し，今ではそれなりの機能を持つサーバソフトへと成長しました．皆さんもこれを機会に簡単なネットワークプログラムを作ってみてはどうでしょう．WD

表4◎フィールド識別コード

●フィールド識別コード	●意味	●フィールド値の例
S	エラーの深刻度	ERROR
C	エラーコード	22021
M	エラーメッセージ	could not create unique index
D	詳細メッセージ	Table contains duplicated values.
H	ヒント	Please REINDEX it.
P	エラーの位置	92
F	ソースファイル名	scan.c
L	行番号	110
R	関数名	foo