実践テクニックをご紹介

徒然PostgreSQL散策

第7回 PostgreSQLとネットワークプログラミング

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

さようならJPUG

1999年の創立以来，筆者はJPUG（日本PostgreSQLユーザ会）の理事長を務めさせていただいてきましたが，このたび理事長の職を退くことになりました．

お陰さまでJPUGは日本のコミュニティの中でも運営がうまくいっているところ，という評判をいただいており，これも理事の皆様，ボランティアの皆様の努力の結果であったと思っています．

どんな組織でもそうですが，長い間特定の人間に権力が集中すれば腐敗につながります．そうなる前に辞めたかったというのと，新しい方に道を譲って新しい発想でJPUGを盛り立てていただきたい，というのが退任の理由です．あ，それにもっとコードを書く時間が欲しかったというのもあります:-)

今後は一会員としてJPUGとPostgreSQLの発展に寄与していきたいと思っています．

pgpoolにレプリケーション機能追加！

以前この連載でpgpoolというPostgreSQL専用のコネクションプールサーバを取り上げたことがあるのを覚えていらっしゃるでしょうか？[注1] バックナンバーはすでに売り切れのようなので，そのときに書いたpgpoolの紹介部分を引用しておきます．

今回開発したのは「pgpool」というシンプルなコネクションプールサーバです．pgpoolはPostgreSQLのクライアントとPostgreSQLサーバの間に割り込む形で使用します（**図1**）．

pgpoolはクライアントから見るとPostgreSQLサーバに見え，同時にPostgreSQLサーバから見るとPostgreSQLのクライアントに見えるようになっており，いわばPostgreSQLのプロキシサーバとして動作します．PostgreSQLのアプリケーションからpgpoolを利用するためには，接続ポート番号を9999に変えるだけでよく，ポート番号の変更以外にはアプリケーションプログラムの変更は必要ありません．PostgreSQLサーバのほうは一切変更はいりません．また，この図1ではPHPを使っていますが，もちろんPerlでもJavaでも使えます．

pgpoolはPostgreSQLへのコネクションをキャッシュするので，その分効率が良くなります．また，PostgreSQLへの同時接続数を一定の数に制限し，PostgreSQLの性能を最大限に発揮できるメリットもあります．

（中略）

なお，pgpoolにはフェールオーバと言って，PostgreSQLサーバがダウンしたときに自動的にあらかじめ用意しておいた2台目のPostgreSQL（図1で「secondary」と表記）サーバに切り替える機能があります．フェールオーバにより，システムのダウンタイムを最小限に留めることができる，予備機を用意してのデータベースのメインテナンスが容易になるなどの利点があります．

ただしpgpoolには2台のデータベースの内容を同期させる機能はありませんので，必要ならばdbmirrorな

注1）本誌Vol.16，200ページ．

どのレプリケーションソフトを併用してください．

実は最後の「2台のデータベースの内容を同期させる機能はありません」というところがずっと気になっていまして，思い立ってこの3月にレプリケーション機能をpgpoolに追加してみました．大分安定してきたので，今回この連載で取り上げることにしました．

プログラミング事例としてのpgpool

といっても，この連載で取り上げる以上，機能や使い方といった普通の切り口ではつまりません．そこで今回はネットワークプログラミング，サーバプログラミングの事例としてpgpoolを取り上げることにしました．具体的には，ソケットの使い方やpre-forkテクニックの解説を行います．

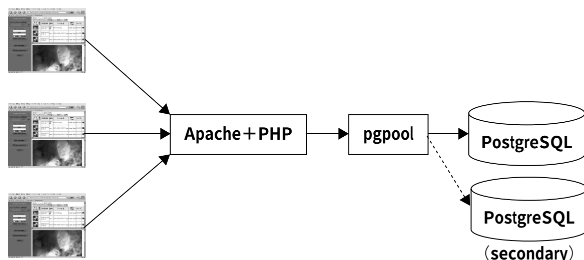
また，pgpoolはPostgreSQLのプロキシサーバであるとも言えます．つまりPostgreSQLの通信プロトコルを実装しているわけで，そういった面からもpgpoolを解説したいと思います．

なお，今回解説の対象とするのは本稿執筆時点の最新安定版であるpgpool 1.2.3です．ソースコードはhttp://www2b.biglobe.ne.jp/ caco/pgpool/から入手できます．

pgpoolによるレプリケーション

今回pgpoolに追加されたレプリケーション機能は，セカンダリホスト（図1参照）の機能を拡張して実装しています．すなわち，コネクションプールモードでは，フェールオーバのときだけセカンダリを使用していたのを，レプリケーションモードでは常にセカンダリにも同じ問い合わせを投げるようにし，結果としてマスタ側とセカンダリ側のデータベースの内容が一致するようにしています．

図1◎pgpoolを使用した場合のシステム構成



ネットワークアプリケーションの基礎

Linuxなどの近代的なOSでは，プログラムは独立した実行要素に分割されて管理されています．UNIXやLinuxでは，その単位は**プロセス**（process）と呼ばれています．プロセスはお互いに独立しており，それぞれ相手の動きに関係なく実行され，またメモリ空間もお互いに独立しています．これによって，あるプロセスがバグなどによって異常な状態になってもほかのプロセスには影響を与えないというメリットが生まれます．一方で，プロセス同士を連携させるためには何か特別な仕掛けが必要になります．

プロセス同士を連携させるための仕掛けの代表的なものとしては，以下のようなものがあります．

① シグナル
② セマフォ
③ 共有メモリ
④ パイプ
⑤ ネットワーク通信

このうち①から④までは同じホスト上のプロセス間の連携に用いられます．PostgreSQLやpgpoolは異なるホストの間での通信を行う必要があり，それが可能なのは⑤だけです．ネットワーク通信を行うアプリケーションは一般に「ネットワークアプリケーション」と呼ばれます．

ネットワークアプリケーションでは，**クライアント（client）／サーバ（server）モデル**がよく使われます．このモデルでは，通信を行う2つのプロセスは対等ではなく，サーバは一旦起動されたらクライアントからの通信要求を待ち続けます．クライアントは必要なときにサーバに接続し，処理を依頼します．サーバは処理を受け付けたら必要な仕事をしてその結果をクライアントに返します．クライアントは処理が終わったら「終わったよ」という応答をサーバに返し，通常この段階でクライアントとサーバの通信も終了します

図2◎ネットワークアプリケーションの例

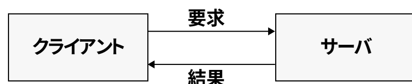
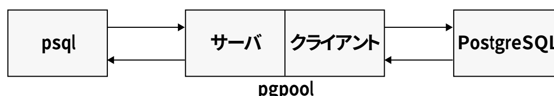


図3◎ネットワークアプリケーションのバリエーションとしてのpgpool



(図2).

クライアントサーバモデルにはさまざまなバリエーションが考えられます．たとえばpgpoolは，クライアント（psqlなど）に対してはサーバとして振る舞い，一方でPostgreSQLに対してはクライアントとして振る舞います（図3).

ソケットインタフェースとは

実際にクライアントとサーバが通信できるためには，そのためのハードウェアが必要です．すなわち，LANならばイーサネットや通信インタフェースカード，ADSLならばADSLモデムやルータ，ダイアルアップならばモデムと電話線などになります．当然のことながら，これらの異なるハードウェアを使うためのソフトウェアは異なってきます．しかし，こうした違いをアプリケーションで意識するのは極めて煩雑ですし，間違いも起きやすくなります．

そこで**ソケットインタフェース**（socket interface）が用いられます．ソケットインタフェースを使えば，どの通信媒体であっても，アプリケーションプログラムを変更することなく通信が可能です．ハードウェアの詳細をOSが隠蔽しているからです．しかもソケットインタフェースでは，通信データの読み書きをファイルディスクリプタを通じて行うので，普通のファイルに読み書きする要領でネットワーク通信が行えます．

ソケットインタフェースにはいくつかのAPI（Application Program Interface）があり，サーバなのかクライアントなのかで使い方が違いますし，またそれぞれのAPIを呼び出す順番も決まっています．さらに，通信プロトコルによってパラメータや呼び出し方の詳細も異なってきます．ここでは，誌面の都合もありますので，信頼性が高くもっとも広く利用されており，PostgreSQLやpgpoolでも使われているTCP/IPを中心に説明します．

サーバ側でのソケットの使い方

すでに説明したように，サーバ側はクライアント側からのコネクションを受動的に待ち受けるような作りになっています．ソケットインタフェースでは，概ね以下のような手順でAPIを呼び出し，このような動作を実現します．

① ソケットの作成

socket()を呼び出し，ソケットを作ります．このとき作られたソケットはサーバが終了するまでずっと使い続けられます．

② バインド

クライアントが接続先を特定できるように，bind()を呼び出して特定のポート番号にバインドします．

③ リスニング

listen()を呼び出し，クライアントからの接続準備を行います．これでクライアントはサーバに接続できるようになります．

④ クライアントからの接続受け入れ

accept()を呼び出し，クライアントからの接続を受け入れます．成功すればaccept()はファイルディスクリプタを返すので，それを使って普通のファイルに読み書きするようにしてサーバはクライアントと通信を行うことができます．

⑤ クライアントとの接続の切断

accept()が返したファイルディスクリプタをclose()で閉じることにより，クライアントとの接続を終了します．

クライアント側でのソケットの使い方

クライアント側ではソケットを作った後connect( )という関数を呼び出してサーバに接続します．このときサーバ側では③の状態になっていることが必要です．

❶ ソケットの作成

socket()を呼び出し，ソケットを作ります．socket()はファイルディスクリプタを返します．

❷ コネクションの確立

connect( )を呼び出し，サーバと接続します．成功すれば❶で作ったファイルディスクリプタを使ってサーバと通信ができるようになります．

❸ サーバとの接続の切断

socket( )が返したファイルディスクリプタをclose( )で閉じることにより，クライアントとの接続を終了します．

サーバ側ソケットの使い方の実例

では，pgpoolを例にとってサーバ側での実際のソケットの使い方を見てみましょう．pgpool.cでは，main.cの中に定義されているcreate_unix_domain_socket( )とcreate_inet_domain_socket( )の中でsocket( )を呼び出しています．create_unix_domain_socket( )は後回しにして，create_inet_domain_socket( )のほうを先に見てみましょう（**リスト1**）．

▼リスト1　create_inet_domain_socket( )

```
/*
* create inet domain socket
*/
static int create_inet_domain_socket(void)
{
    struct sockaddr_in addr;
    int fd;
    int status;
    int one = 1;
    int len;

    fd = socket(AF_INET, SOCK_STREAM, 0); ——①
    if (fd == -1)
    {
        pool_error("Failed to create INET domain socket. reason: %s", strerror(errno));
        myexit(1);
    }
    if ((setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, (char *) &one, sizeof(one))) == -1) ——②
    {
        pool_error("setsockopt() failed. reason: %s", strerror(errno));
        myexit(1);
    }
    memset((char *) &addr, 0, sizeof(addr));
    ((struct sockaddr *)&addr)->sa_family = AF_INET; ——③
    addr.sin_addr.s_addr = htonl(INADDR_ANY); ——④
    addr.sin_port = htons(pool_config.port); ——⑤
    len = sizeof(struct sockaddr_in);
    status = bind(fd, (struct sockaddr *)&addr, len); ——⑥
    if (status == -1)
    {
        pool_error("bind() failed. reason: %s", strerror(errno));
        myexit(1);
    }

    status = listen(fd, PGPOOLMAXLITSENQUEUELENGTH); ——⑦
    if (status < 0)
    {
        pool_error("listen() failed. reason: %s", strerror(errno));
        myexit(1);
    }
    return fd;
}
```

socket( )【リスト1-①】

socket( )の呼び出し（リスト1-①）を見ながら，socket( )の引数を追っていきます．

第1引数の「AF_INET」は，**アドレスファミリ**と呼ばれるもので，どのようなアドレス体系を使うかを指定します．ここではIPv4のインターネットプロトコルを扱うことを示しています．このほか，UNIXドメインソケット（後述）や，AppleTalkなどを指定することができます．詳細はsocket( )のマニュアルを見てください．

第2引数の「SOCK_STREAM」は，ストリーム接続であることを示します．すなわち，最初に接続を確立させてから送信するタイプのプロトコルです．SOCK_STREAM以外では，SOCK_DGRAMを使うこともあります．SOCK_DGRAMはTCP/IPよりも「低レベル」のプロトコルで，送信したデータが届く保証もなければ，届いたとしてもその順番も保証されません．にも関わらずSOCK_DGRAMが用意されているのは，少量のデータを送るようなときには効率が良いからです．TCP/IPは逆で，最初の接続処理などに時間がかかりますが，大量のデータを高い信頼性で送ることができます．

第3引数はほとんどの場合，0のままでOKです．

socket( )が成功すると，ファイルディスクリプタが返ってきますが，これはまだ通信には使えません．

setsockopt( )【リスト1-②】

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，SO_REUSEADDRというオプションを指定しています．これについては後で説明しますので，今はとりあえずサーバにはこれが必要だということだけ覚えておいてください．

bind( )【リスト1-⑥】

ソケットインタフェースでたぶん一番使い方が面倒なのがbind( )です．

第1引数にはsocket( )が返したファイルディスクリプタを渡します．

問題は第2引数です．/usr/include/netinet/in.hで定義されているsockaddr_inという構造体を設定しなければなりません．ただし，bind( )自体はANSI Cの規程により，sockaddr構造体を第2引数で受け付けるようになっています．そのためにリスト1-⑥ではstruct sockaddr *というキャストが必要になっています．ここで構造体に設定すべき情報は以下です．

●アドレスファミリ

アドレスファミリはsockaddr構造体で指定します（リスト1-③）．そのためキャストを指定しています．

●コネクションを受け付けるアドレス

IPアドレスを複数持つサーバで，特定のアドレスのみコネクションを受け付ける場合はこれを指定します．ここではサーバのすべてのIPアドレスを受け付けるので，特別なキーワード「INADDR_ANY」を使っています（リスト1-④）．

htonl( )は，4バイトのアドレスを**ネットワークバイトオーダ**に変換する関数です．ネットワークを通信するマシンのアーキテクチャは同じとは限りません．たとえばインテル製のCPUを積んだPCとPowerPCを積んだPCでは，整数の中のバイトの順序が異なります．そのため，ネットワークを流れる整数のバイト順の標準を規程したのがネットワークバイトオーダです．ネットワークバイトオーダを使うことによって，どのようなマシン同士でも通信ができるようになります．なお，htonl( )の逆を行うのがntohl( )です．

●ポート番号

バインドするポート番号をネットワークバイトオーダで指定します（リスト1-⑤）．

ポート番号は2バイト整数（short）なので，htons( )を使っています．

listen( )【リスト1-⑦】

これは簡単で，socket( )が返したファイルディスクリプタを渡します．第2引数はリスニングキューの長さです．これは複数のクライアントが同時に接続に来たときに，待たせておくための待ち行列です．最近のOSでは，適当に大きな数字を与えておけば，OSのほ

うで適当な値を設定してくれます[注2].

UNIXドメインソケット

UNIXやLinuxでは，同一ホストの中に通信が限定されたUNIXドメインソケットというものが使えます．もちろん同一ホストの中でもTCP/IPを使って通信することができますが，一般にUNIXドメインソケットのほうが高速なので，同一ホストの中の通信に使われることが多いようです[注3].

pgpoolでは，UNIXドメインソケットもサポートしています（リスト2）．ご覧のようにリスト1とほとんど同じですがbind( )の引数を作るのに使う構造体がsockaddr_inになっており，以下の点が異なります．

- sa_familyにはAF_UNIXを指定します（リスト2-①）
- ポート番号ではなく，UNIXドメインソケットへのパスを渡します（リスト2-②）

同時に複数クライアントを待ち受けるには？

この後は通常accept( )を発行し，accept( )の中でクライアントからの接続を待ち受けます．そして accept( )から戻ってくるとすでにクライアントと接続

▼リスト2　create_UNIX_domain_socket( )

```
/*
* create UNIX domain socket
*/
static int create_unix_domain_socket(void)
{
    struct sockaddr_un addr;
    int fd;
    int status;
    int len;

    fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (fd == -1)
    {
        pool_error("Failed to create UNIX domain socket. reason: %s", strerror(errno));
        myexit(1);
    }
    memset((char *) &addr, 0, sizeof(addr));
    ((struct sockaddr *)&addr)->sa_family = AF_UNIX; ――①
    snprintf(addr.sun_path, sizeof(addr.sun_path), un_addr.sun_path); ――②
    len = sizeof(struct sockaddr_un);
    status = bind(fd, (struct sockaddr *)&addr, len);
    if (status == -1)
    {
        pool_error("bind() failed. reason: %s", strerror(errno));
        myexit(1);
    }

    if (chmod(un_addr.sun_path, 0777) == -1)
    {
        pool_error("chmod() failed. reason: %s", strerror(errno));
        myexit(1);
    }

    status = listen(fd, PGPOOLMAXLITSENQUEUELENGTH);
    if (status < 0)
    {
        pool_error("listen() failed. reason: %s", strerror(errno));
        myexit(1);
    }
    return fd;
}
```

注2）ここまでの説明では複数のクライアントが接続に来たときのことを考慮していませんが，これについては後で説明します．

注3）同じUNIXでもSolarisではTCP/IPのほうが高速なようです．また，アプリケーションによってはそもそもUNIXドメインソケットがサポートされていないものもあります（たとえばJavaがそうです）．

された状態になっているので，accept()が返したファイルディスクリプタを使ってクライアントと通信を行います．

しかし，このような使い方では，あるクライアントの処理をしている間は，他のクライアントから新たな接続要求があってもそれを受け付けることができません．これでは困るので，普通はaccept()までしておいて，以降は別プロセスにまかせるようにします．すなわち，親プロセスでsocket( )，bind( )，listen( )，accept()まで行い，ここでfork()を使って子プロセスを生成します．

子プロセスは親プロセスのファイルディスクリプタを受け継いでいるので，accept()の返したファイルディスクリプタを使ってクライアントと通信を行うことができます．一方，親プロセスにはaccept()の返したファイルディスクリプタはもう不要なので，close( )し，またaccept( )の発行に戻ります（図4）．

この方式は処理が簡単なので多くのサーバプログラムに用いられています．PostgreSQLもこの方式です（親プロセスはpostmaster，子プロセスはpostgresという名前のプロセスです）．

pre-fork方式

この方式は簡単ですが，accept()するたびにプロセスを新たに作らなければならないのであまりパフォーマンスがよくないという欠点があります．この問題を解決するのがpre-fork方式です．pre-fork方式では，事前にある程度の数の子プロセスを作っておき，クライアントからのコネクション要求があるとそれらの子プロセスが一斉に要求を受け取りに行きます（図5）．

こう書くと子プロセス間の競合が心配になりますが，カーネルがうまく調整して要求を受け取るプロセスを1個だけにするので心配はありません．

WebサーバのApacheや，pgpoolはこの方式を採用しています．

すぐにaccept()から戻るしくみ

pre-fork方式では，多数の子プロセスが接続要求を取り合います．実際に要求を受け取ることができるのは1つのプロセスだけで，残りは空振りになります．そのため，空振りになった子プロセスはaccept()の中で永久に待たされます．これでは都合が悪いので，もし接続要求がなければすぐにaccept()から戻れるようにしておかなければなりません．このための仕掛けが非ブロックソケットとselect( )です．

●非ブロックソケットの利用

まず，非ブロックソケットを設定するにはfcntl()を使います．

```
var = fcntl(fd, F_GETFL, 0);
```

のようにして現在のソケットを取り出し，

```
fcntl(fd, F_SETFL, var | O_NONBLOCK);
```

図4◎fork( )を使った複数クライアントの待ち受け

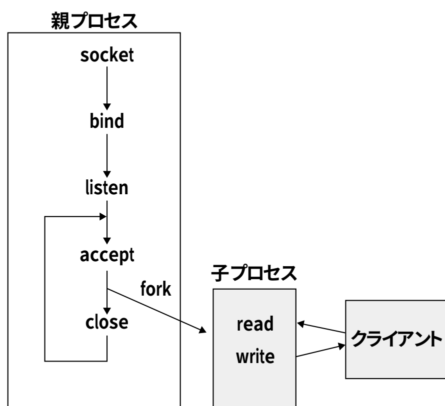
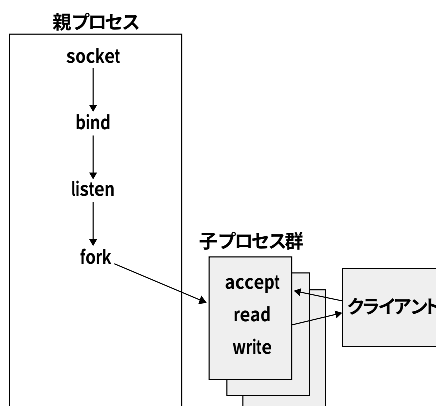


図5◎pre-fork方式



と，非ブロック属性O_NONBLOCKのビットを立てます．これで非ブロックソケットになります．再度ブロックソケットに戻すには，

```
fcntl(fd, F_SETFL, var & ~O_NONBLOCK);
```

としてビットを落とします．このあたりは，child.cのset_nonblock()（286行目あたり）とunset_nonblock()（308行目）を見てください（この部分は本稿には未掲載です）．

● select()の利用

さて，非ブロック属性を設定しただけではまだ接続要求が届いていない場合にすぐにaccept()から復帰してしまいます．かと言って，何も要求がないときにすぐにaccept()に戻るようなことをするとbusy loopになって負荷が高くなります．busy loopを避けるためには，途中でsleep()を入れることもできますが，これでは最悪sleep()する秒数分だけレスポンスが遅れます．

こうした問題を解決するのがselect()です．select()に監視したいファイルディスクリプタを指定しておくと，そこに何らかの入力があるまで待ち受けてくれます．

リスト3はchild.cのdo_accept()からの抜粋です．

select()の第1引数は監視するファイルディスクリプタ番号の最大＋1です（リスト3-①）．第2，第3，第4引数は，それぞれ読み込み，書き込み，例外事象発生監視対象のファイルディスクリプタを表わすビットマップです．最後の引数はタイムアウトで，NULLを指定するとタイムアウトしません．

▼リスト3　select()の利用

```
FD_ZERO(&readmask);
FD_SET(unix_fd, &readmask);
if (inet_fd)
    FD_SET(inet_fd, &readmask);

fds = select(Max(unix_fd, inet_fd)+1, &readmask, NULL, NULL, NULL);      ――①
if (fds == -1)
{
    if (errno == EAGAIN || errno == EINTR)
        return NULL;

    pool_error("select() failed. reason %s", strerror(errno));
    return NULL;
}

if (fds == 0)
    return NULL;

if (FD_ISSET(unix_fd, &readmask))
{
    fd = unix_fd;
}

if (FD_ISSET(inet_fd, &readmask))
{
    fd = inet_fd;
    inet++;
}

/*
 * Note that some SysV systems do not work here. For those
 * systems, we need some locking mechanism for the fd.
 */
addrlen = sizeof(addr);

afd = accept(fd, &addr, &addrlen);
if (afd < 0)
{
    pool_error("accept() failed. reason: %s", strerror(errno));
    return NULL;
}
```

ファイルディスクリプタ用のビットマップは，まずFD_ZERO( )で0クリア，FD_SET( )で該当ビットを立てることによって設定します．もしどれかのファイルディスクリプタに入力があると，select( )はそのファイルディスクリプタの数を返します．どのファイルディスクリプタが設定されたかはFD_ISSET( )でわかります．後はそのファイルディスクリプタを使ってaccept( )を呼び出せば，通信に使えるファイルディスクリプタが返ってきます．

pgpoolはTCP/IPソケットとUNIXドメインソケットの両方をサポートしています．select( )が監視するファイルディスクリプタとしてこの両方を監視するようにすれば，TCP/IPソケットとUNIXドメインソケットのどちらから接続要求があっても対応できます．

次回はPostgreSQLの通信プロトコルの解説

さて，クライアントとサーバの通信が確立するところまで説明したところで誌面が尽きました．次回はPostgreSQLの通信プロトコルについて解説します．

PostgreSQLは，PostgreSQL 7.3から7.4にバージョンアップした際に通信プロトコルが大きく変わりました．現在のpgpoolはまだ7.3の通信プロトコルにしか対応していません．PostgreSQL 7.4は7.3の通信プロトコルに対しても下位互換性があるので一応pgpoolも含めて7.3以前のクライアントは7.4と通信できますが，7.4になってから追加されたPREPARE，EXECUTEなどの機能には当然のことながら対応できません．

これでは残念なので，現在pgpoolも7.4にも対応すべく改良中です．次回までには対応ができているはずなので :-)，せっかくですから新しい7.4のプロトコルをベースに解説を行いたいと思っています．

参考文献：
『UNIXネットワークプログラミング（Vol.1/2）』，W.Richard Stevens 著，篠田 陽一 訳，ピアソンエデュケーション

初体験 はつたいけんでもやさしい
Java サーブレット

丸の内とら氏の初体験シリーズ第3弾です．Javaサーブレットは，Webサーバー側で実行されるJavaアプリケーションです．少し前まではCGIが多く使われてきましたが，最近は実行効率の良いJavaサーブレットに脚光が当たっています．

本書は，このJavaサーブレットをはじめて学習する人のための入門書です．覚えるよりも慣れることを目的としていますので，まず，動かしてみたいと思っている方にピッタリです．

好評発売中

CD-ROM付き

丸の内 とら 著
B5変形 判 ／ 312 ページ
本体価格 2180 円＋税
ISBN4-7741-1467-7

技術評論社