pgpool-II のオンラインリカバリ、および PITR の概要

鈴木啓修@InterDB.jp

本稿は、2008 年 12 月発売の、技術評論社 WEB+DB PRESS vol. 48 の「特集 3 PostgreSQL 大規模運用」の草稿を編集したものである。インストールや設定部分は契約上公開できないので了承願いたい[1]。

1. pgpool-II とは

pgpool-II は同期レプリケーション機能をはじめ、コネクションプール機能や負荷分散機能を提供する PostgreSQL 専用のミドルウエア。 pgpool-II は、先代に当たる pgpool の後継プロダクトとしてリリースされた。

表 1 pgpool の歴史

プロダクト	バージョン	リリース時期	主な追加機能
pgpool	ver1	2004.4	コネクションプール機能、(2 台による)同期レプリケーション機能
pgpool	ver2	2004.6	負荷分散機能
pgpool-II	ver1	2006.9	パラレルクエリー、2 台以上の同期レプリケーション機能
pgpool-II	ver2	2007.11	オンラインリカバリ

1.1. コネクションプール

pgpool は PostgreSQL のコネクションプールサーバとして誕生した。 PostgreSQL や Oracle などの RDBMS は、アプリケーションから接続を要求される毎にプロセスを起動して SQL 文を実行する(図 1)。

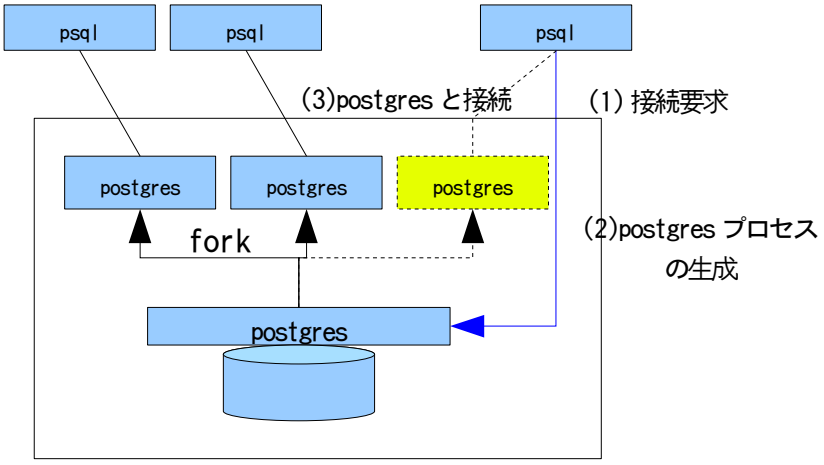

図 1 PostgreSQL との接続

接続の過程を説明すると:

(1)アプリケーションが postmaster(PostgreSQL のメインプロセス)に接続要求、

(2)その度に postgres というプロセスが生成、

(3)アプリケーションと接続完了、

1　興味がある場合には WEB+DB PRESS vol. 48 を入手するか、2009 年 5 月以降に http://www.interdb.jp/techinfo/pgpool-II/ にアクセスしていただきたい。

となる。

しかし、接続の度にプロセスを生成するのは負荷が高く処理性能が低下する。 これを避けるため予め複数のプロセスを起動しておく仕組みをコネクションプールという。

pgpool は PostgreSQL の前段で事前に複数の子プロセスを起動し、それぞれが PostgreSQL のプロセス postgres と接続されている(図 2)。 アプリケーションから pgpool に接続要求が届き SQL 文が送られると、それをそのまま PostgreSQL に転送するので、接続による性能低下が発生しない。

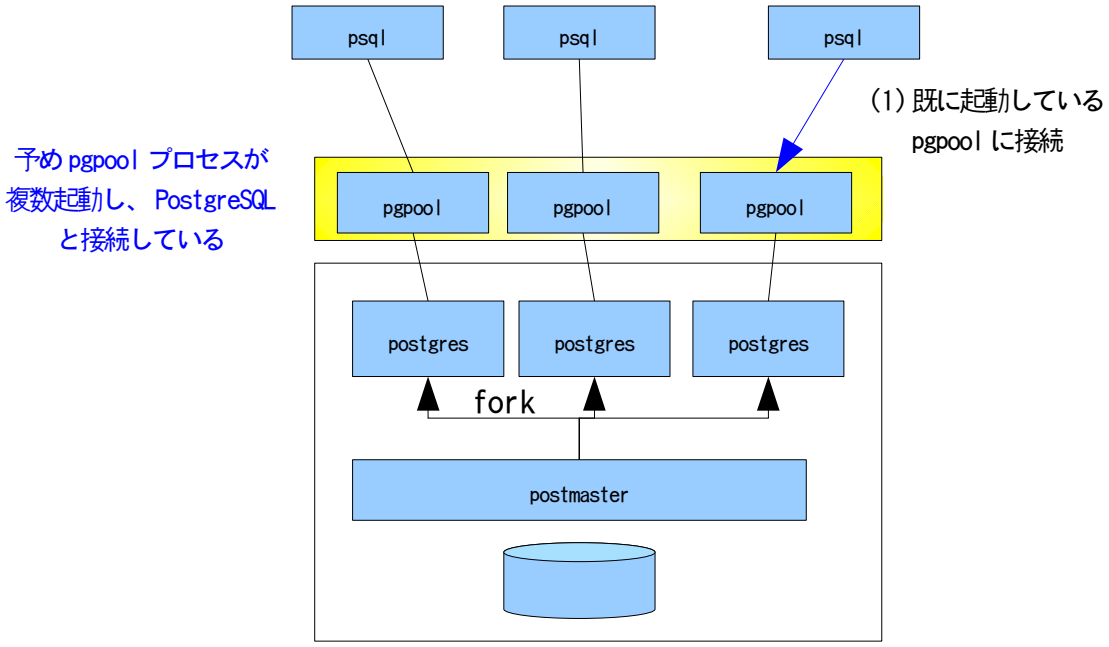

図 2 pgpool によるコネクションプールを利用した接続

1.2. 同期レプリケーション

PostgreSQL の前段で機能するという pgpool の特徴から、 非常に早い段階に同期レプリケーションも実装された(図 3)。 これは、各 pgpool プロセスが 2 台の PostgreSQL サーバに接続し、 (1)アプリケーションから受け取った更新系の SQL を、(2)2 台の PostgreSQL に転送する形式。

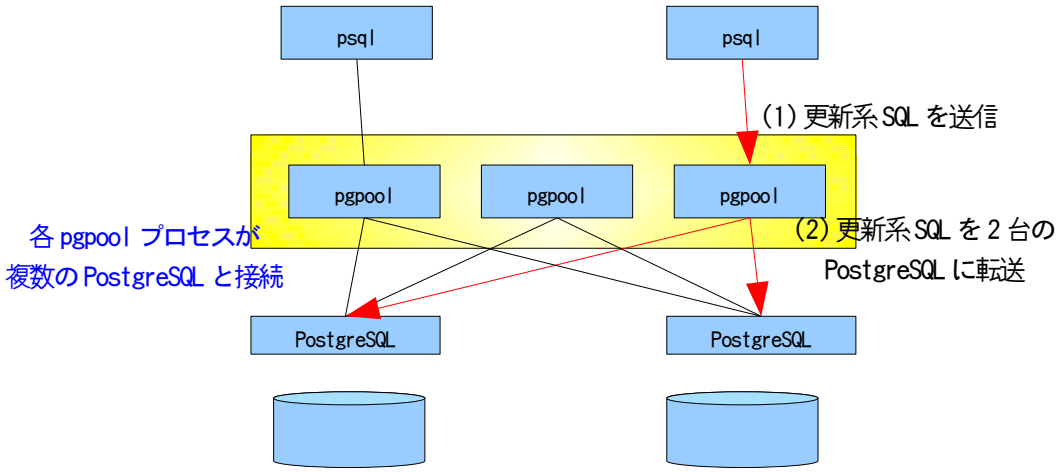

図 3 pgpool による同期レプリケーションのイメージ

1.3. 簡単な内部構造の説明

pgpool-II はマルチプロセスシステムで、共有メモリでデータ共有、セマフォで相互排他制御、シグナルによる内部通信を行う(図 4)。

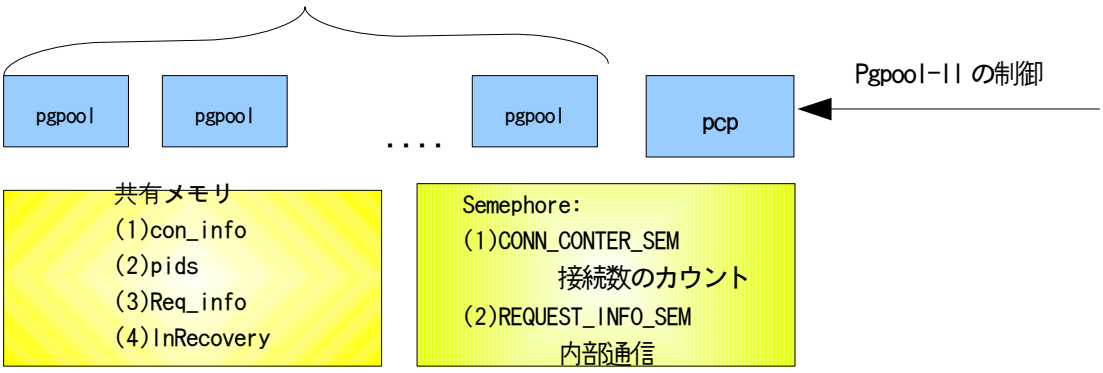

図 4. pgpool-II の内部構造

プロセス間で共有する情報は：

(1)接続情報	con_info
(2)プロセス pid	pids
(3)リクエスト情報	Req_info
(4)リカバリ中か否か	InRecovey

以下に(1)から(3)までの各種情報を定義する構造体を示す。

```
typedef struct {
        char            database[SM_DATABASE];    /* Database name */
        char            user[SM_USER];    /* User name */
        int                     major;    /* protocol major version */
        int                     minor;    /* protocol minor version */
        int                     counter; /* used counter */
        time_t          create_time; /* connection creation time */
        int load_balancing_node; /* load balancing node */
} ConnectionInfo;

typedef struct {
        pid_t pid; /* OS's process id */
        time_t start_time; /* fork() time */
        ConnectionInfo *connection_info; /* connection information */
} ProcessInfo;

typedef struct {
        POOL_REQUEST_KIND kind;    /* request kind */
        int node_id[MAX_NUM_BACKENDS];              /* request node id */
        int master_node_id;       /* the youngest node id which is not in down status */
        int conn_counter;
} POOL_REQUEST_INFO;
```

2.　オンラインリカバリ

2.1. オンラインリカバリの仕組み

pgpool-II のオンラインリカバリの仕組みを解説する(図 5)。 pgpool-II のオンラインリカバリは、2 つのフェーズに分かれている。

フェーズ 1：

リカバリ元のベースバックアップをリカバリするサーバに転送。

フェーズ 2：

フェース 1 の作業中に更新されたアーカイブログを転送し、サーバを再起動。 先に転送されたベースバックアップと合わせてデータベースのリカバリ後にサービス開始。

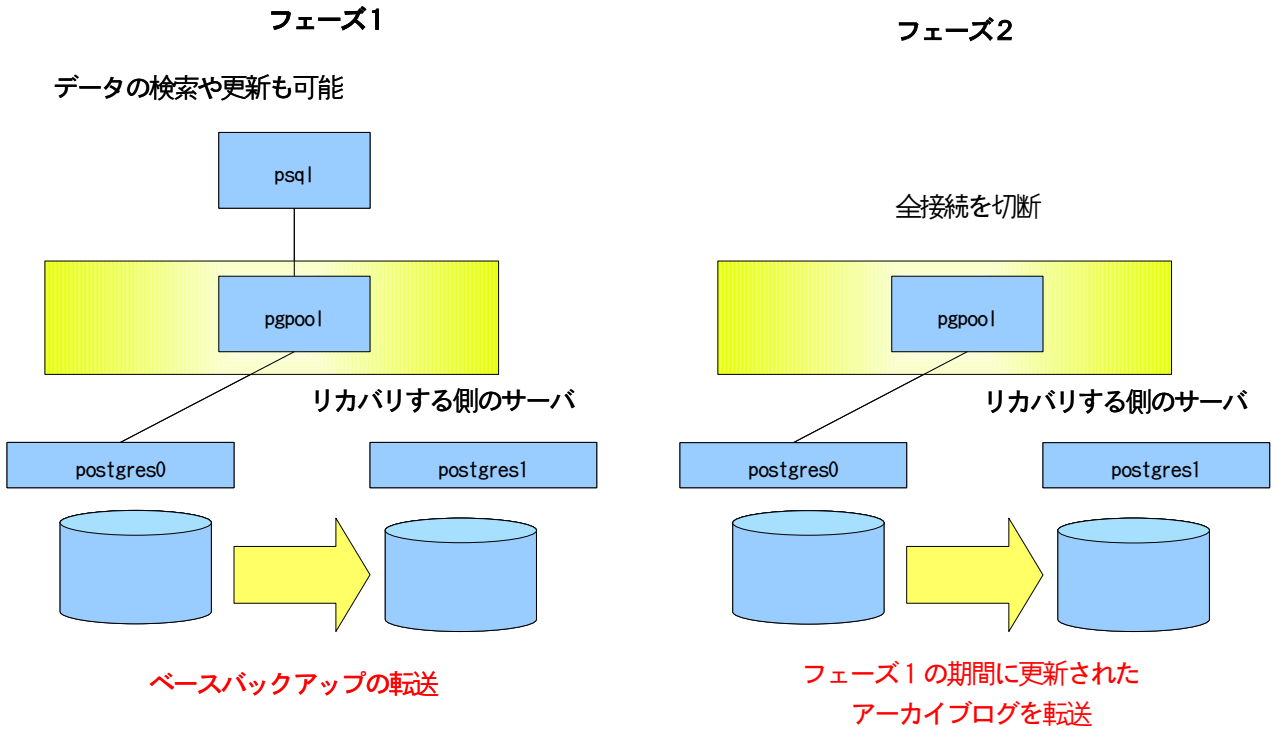

図 5 オンラインリカバリ

2.2. PostgreSQL にインストールする pgpool-II の関数

pgpool-II はオンラインリカバリを実行するため、PostgreSQL 側の関数 pgpool_recovery()と pgpool_remote_start()を実行する。

```
CREATE OR REPLACE FUNCTION pgpool_recovery(text, text, text)  RETURNS bool
AS '$libdir/pgpool-recovery', 'pgpool_recovery'  LANGUAGE C STRICT;

CREATE OR REPLACE FUNCTION pgpool_remote_start(text, text)    RETURNS bool
AS '$libdir/pgpool-recovery', 'pgpool_remote_start'  LANGUAGE C STRICT;
```

関数 pgpool_recovery()と pgpool_remote_start()の機能は、データベースクラスタ上に置かれたシェルスクリプトの実行である。

つまり：pgpool-II が関数 pg_recovery()などを介して、シェルスクリプトを実行させるわけである(図 6)。

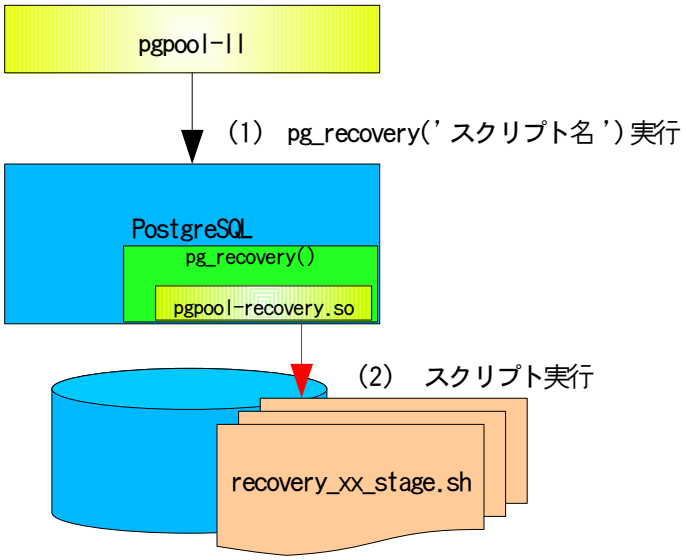

図 6.PostgreSQL 上のシェルスクリプト実行の仕組み

参考までに pgpool_recovery の実装を示す。本体は OS の system コマンドを呼び出すだけであることがわかる。

```
pgpool-recovey.c

Datum
pgpool_recovery(PG_FUNCTION_ARGS)
{
        int r;
        char *script = DatumGetCString(DirectFunctionCall1(textout,  PointerGetDatum(PG_GETARG_TEXT_P(0))));
        char *remote_host = DatumGetCString(DirectFunctionCall1(textout,
PointerGetDatum(PG_GETARG_TEXT_P(1))));
        char *remote_data_directory = DatumGetCString(DirectFunctionCall1(textout,
                 PointerGetDatum(PG_GETARG_TEXT_P(2))));

        snprintf(recovery_script, sizeof(recovery_script), "%s/%s %s %s %s",
                 DataDir, script, DataDir, remote_host,
                 remote_data_directory);
        elog(DEBUG1, "recovery_script: %s", recovery_script);

        r = system(recovery_script);

        if (r != 0){
                elog(ERROR, "pgpool_recovery failed");
        }
        PG_RETURN_BOOL(true);
}
```

2.3. オンラインリカバリのタイムシーケンス

以下に、pgpool-II による、オンラインリカバリ処理のタイムシーケンスを示す。ちなみにオンラインリカバリのトリガは、pcp_recovery_node コマンドである。

フェーズ1

pg_start_backup()関数の実行後にベースバックアップをリカバリ側のサーバに転送する。

フェーズ2：

WAL ログを切り替えて最新のアーカイブログをリカバリ側のサーバに転送、最後に PostgreSQL サーバを再起動する。

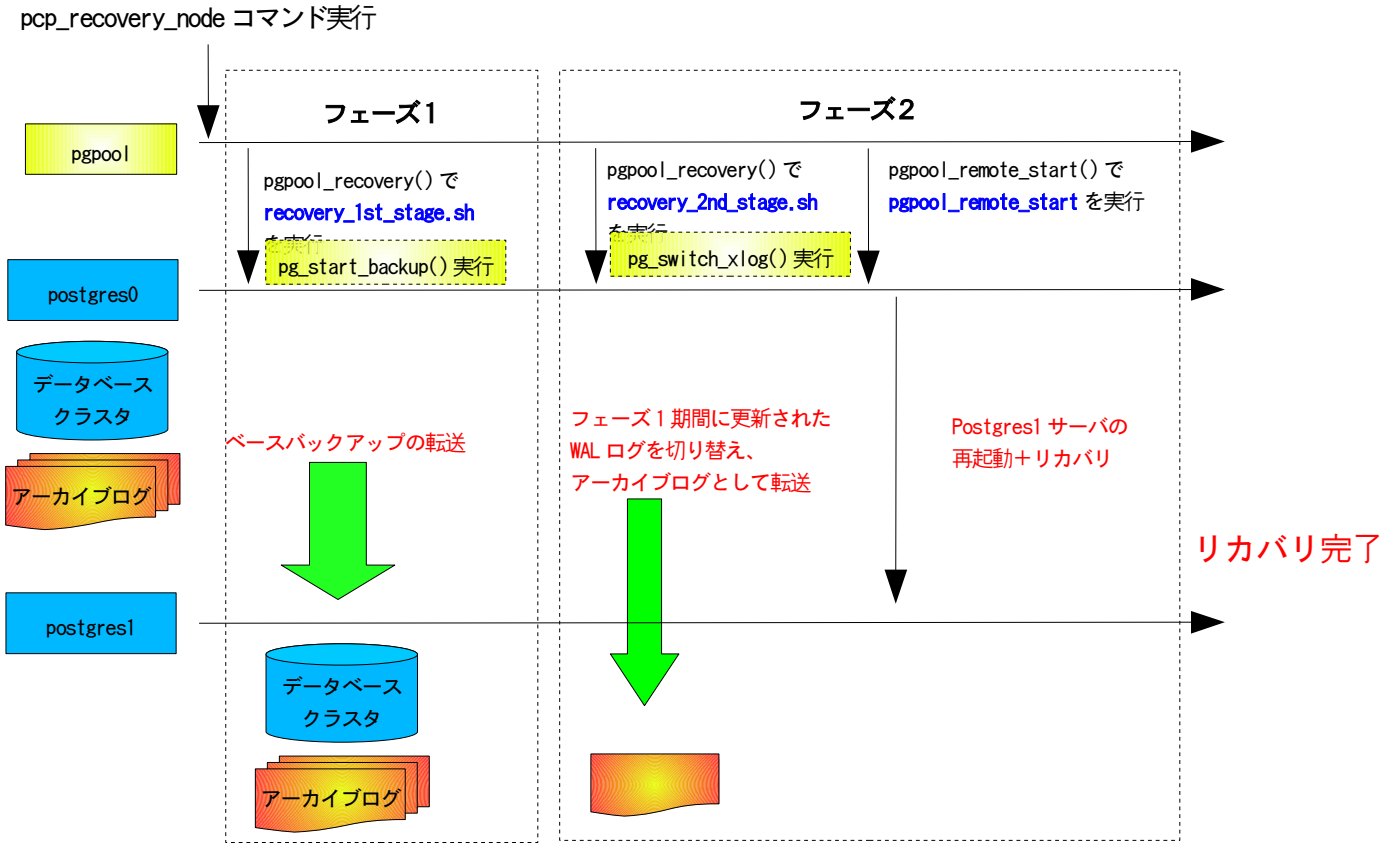

図7 オンラインリカバリのタイムシーケンス

参考までに、上記の2フェーズを実行する pgpool-II のソースコードを抜粋して示す。

recovery.c

```
int start_recovery(int recovery_node)
{
        /* 1st stage */
        if (exec_checkpoint(conn) != 0);
        if (exec_recovery(conn, recovery_backend, FIRST_STAGE) != 0);

        /* 2nd stage */
        if (wait_connection_closed() != 0);
        if (exec_checkpoint(conn) != 0);
        if (exec_recovery(conn, recovery_backend, SECOND_STAGE) != 0);
        if (exec_remote_start(conn, recovery_backend) != 0);
        if (check_postmaster_started(recovery_backend))

        send_failback_request(recovery_node);

        /* wait for failback */
        while (!pcp_wakeup_request){
                struct timeval t = {1, 0};
                /* polling SIGUSR2 signal per 1 sec */
                select(0, NULL, NULL, NULL, &t);
        }
        pcp_wakeup_request = 0;
        return 0;
}

static int exec_recovery(PGconn *conn, BackendInfo *backend, char stage)
{
        PGresult *result;
        char *hostname;
        char *script;
        int r;

        hostname = backend->backend_hostname;
        script = (stage == FIRST_STAGE) ?
              pool_config->recovery_1st_stage_command : pool_config->recovery_2nd_stage_command;

        snprintf(recovery_command,
                         sizeof(recovery_command),
                         "SELECT pgpool_recovery('%s', '%s', '%s')",
                         script, hostname, backend->backend_data_directory);

        result = PQexec(conn, recovery_command);
        r = (PQresultStatus(result) !=  PGRES_TUPLES_OK);
        PQclear(result);
        return r;
}
```

Appendix

A-1. recovery_1st_stage.sh

```
#! /bin/bash

PSQL=/usr/local/pgsql/bin/psql
MASTER_BASEDIR=$1
RECOVERY_HOST=$2
RECOVERY_BASEDIR=$3

# ベースバックアップの開始
$PSQL -c "SELECT pg_start_backup('pgpool-recovery')" postgres

# リカバリ先用のrecovry.confファイル生成
echo "restore_command = 'cp $RECOVERY_BASEDIR/archive_log/%f %p'" > $MASTER_BASEDIR/recovery.conf

# リカバリ先のデータベースクラスタを念のためにバックアップ
ssh -T $RECOVERY_HOST rm -rf $RECOVERY_BASEDIR.bk
ssh -T $RECOVERY_HOST mv -f  $RECOVERY_BASEDIR{,.bk}

# データベースクラスタ=ベースバックアップをリカバリ先に転送
rsync -az -e ssh $MASTER_BASEDIR/ $RECOVERY_HOST:$RECOVERY_BASEDIR/
ssh -T $RECOVERY_HOST cp -f  $RECOVERY_BASEDIR.bk/postgresql.conf $RECOVERY_BASEDIR
ssh -T $RECOVERY_HOST rm -f  $RECOVERY_BASEDIR/postmaster.pid

# リカバリ先に転送したので、不要になったrecovery.confを削除
rm -f $MASTER_BASEDIR/recovery.conf

# ベースバックアップの終了
$PSQL -c "SELECT pg_stop_backup()" postgres
```

A-2. recovery_2nd_stage.sh

```
#! /bin/bash

PSQL=/usr/local/pgsql/bin/psql
MASTER_BASEDIR=$1
RECOVERY_HOST=$2
RECOVERY_BASEDIR=$3

# 最新のアーカイブログを保存
$PSQL -c 'SELECT pg_switch_xlog()' postgres

# 最新のアーカイブログをリカバリ先に転送
rsync -az -e ssh $MASTER_BASEDIR/archive_log/ $RECOVERY_HOST:$RECOVERY_BASEDIR/archive_log/
```

A-3.　pgpool_remote_start

```
#! /bin/sh

PGCTL=/usr/local/pgsql/bin/pg_ctl

RECOVERY_HOST=$1
RECOVERY_BASEDIR=$2

# リカバリ先のPostgreSQLを起動
ssh -T $RECOVERY_HOST $PGCTL -w -D $RECOVERY_BASEDIR start 2>/dev/null 1> /dev/null < /dev/null &
```