実践テクニックをご紹介

徒然PostgreSQL散策

第6回 トランザクションログ

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

はじめに

今回はPostgreSQLにおけるトランザクションログについて解説します．現在，PostgeSQLの開発コミュニティでは，PITR（Point In Time Recovery）という機能の実装が進行中です．PITRは現バージョンで実装されているトランザクションログの拡張とも言えるので，この機会にPostgreSQLのトランザクションログの実装について振り返っておきたいと思います．

トランザクションログの必要性

データベースシステムは，データの保全に関して非常に気を使っています．たとえばデータベースに対して行われた更新は最終的にはハードディスク上に記録されますが，途中さまざまな過程を経ていきます．

① 更新データはまずPostgreSQLが管理する共有バッファ（shared buffer）上に書き込まれます
② 共有バッファの空きがなくなると，PostgreSQLは①のバッファ上のデータをwrite( )システムコールを使ってOSに書き込み要求します
③ OSは更新要求のあったデータをOSが管理するメモリ上のバッファ領域に書き込みます
④ OSは都合のよいときにハードディスクにバッファ上のデータを書き込みます

最終的に④に至る過程のどこかでOSやハードウェアに障害が発生すると，当然データはハードディスクに書き込まれません．まったく書き込まれないのならまだ救われます．アプリケーションが再度データベースに対してデータの更新を依頼すればよいわけですから．しかし，④の処理の最中に障害が発生すると，あるデータはディスクに書き込まれたが，あるデータは書かれなかったということが起こり得ます．データの更新を全部やり直すことも考えられますが，更新ではなくてデータの追加を行っていた場合にはこの方法では対処できません．データの二重登録が発生するかもしれないからです．

PostgreSQLの7.0までは，このような問題に対処するために，トランザクションをコミットした際にすべてのデータを同期書き込みという方法でディスクに書き込むようにOSに依頼していました．つまり，④で，「都合のよいときに」ではなく，直ちにハードディスクにデータを書き込むようにしていたわけです．この方法では確かにデータは保全されますが，I/Oがその都度発生するために非常に性能が悪くなるという別の問題があります．

そこで7.1で採用されたのが**トランザクションログ**です．トランザクションログはディスク上のファイルで，ここにデータベースへの更新情報が逐一書かれます．トランザクションログにはすべての更新記録が記されていますから，万が一障害が発生してもトランザクションログからデータを復旧できるというわけです．

WALとは

バージョン7.1以降のPostgreSQLでは，更新の際にまずトランザクションログを確実に同期書き込みで記録し，それが完了してからデータファイルやインデ

ックス（以後「データファイル」と呼ぶことにします）を書き込みます（このときに同期書き込みは行いません）．この方式はトランザクションログを先に書き込むことからWAL（Write Ahead Logging：ログ先行書き込み）と呼ばれ，PostgreSQL以外のデータベースシステムでも広く採用されています．

このように説明すると，トランザクションログとデータファイルの両方にデータを重複して書いていくのでかえって性能が落ちるような気がするかもしれませんが，WALへのデータの書き込みは常に追記になるため，ハードディスクの特性上非常に効率よく書き込みが行われます．一方，データファイルへの書き込みは効率の悪いランダム書き込みにならざるを得ず，どうしても効率が落ちます．トランザクションログを使う場合はこれを最小限に避けることができるため，総合的に見ると格段に性能が向上します．実際，同じハードウェアを使った場合，WALを採用する前の7.0と採用後の7.1ではトランザクション性能に10倍以上の差がついた記憶が筆者にはあります．

トランザクションログを使ったリカバリ

データベースが異常終了すると，データファイルの内容は同期書き込みされていませんから，一貫性のない状態になっている可能性があります．そこで，トランザクションログに書かれた情報を使ってもう一度データファイルへの更新処理をやり直します．これをREDOと呼びます．

一方，アボートしたトランザクションや，ダウン直前まで実行中だったトランザクションによって行われた変更は，一般的には取り消す（UNDO）必要があります．ところがPostgreSQLでは追記型の記憶構造を採用しているため変更前のデータがすべて残っているので，UNDOは不要です．コミットされなかったトランザクションに対しては，単にアボートしたと見なす処理をするだけで済みます．

リソースマネージャ

更新系のSQL文が発行されると，トランザクションログが作られます．トランザクションログの各レコード（ログレコード）の内容はデータベース内の処理の種類によって異なります．これらの処理は「リソース」という概念で識別され，それぞれ「リソースマネージャ」によって管理されます．WALシステム自体はトランザクションログレコードの詳細には関知しておらず，実際にリカバリ処理を実行するのはリソースマネージャの仕事になります．リソースマネージャの実体はデータベースエンジン内の内部関数で，表1のように分類されます．

PostgreSQL 7.4の時点で登録されているリソースマネージャには表2のものがあります[注1]．また，各リソースマネージャの行う処理はさらに細かく分かれています．

ログレコードの生成

ログレコードの生成は，低レベルのデータベースアクセスメソッドなどの中で行われます．実際にログレコードに書かれる内容はリソースマネージャの種類とその処理詳細によって異なりますが，たとえばHEAP_INSERTの場合は少なくとも次の情報が含まれます[注2]．

- リソースマネージャの種類（この場合Heap）
- 処理詳細（この場合HEAP_INSERT）

表1◎リソースマネージャの実体

●エントリ	●説明
rm_redo	REDO処理
rm_undo	UNDO処理（現在のところ未使用）
rm_desc	デバッグ時に使用
rm_startup	リカバリ処理の初期処理
rm_cleanup	リカバリ処理の終了処理

注1）7.5ではさらに「Storage」というリソースマネージャが追加されていますが，これについては後述します．
注2）このほか大きなログデータを複数のログレコードに分割するためのデータもありますが，煩雑になるのでここでは省略します．

表2◎リソースマネージャと処理

●リソースマネージャの名称	●詳細処理名	●説明
XLOG（OIDなどの管理）	NEXTOID	OIDカウンタの管理
	CHECKPOINT_SHUTDOWN	終了処理中のチェックポイント
	CHECKPOINT_ONLINE	チェックポイント
Transaction（トランザクション管理）	XACT_COMMIT	コミット処理
	XACT_ABORT	アボート処理
Storage（ストレジマネージャ管理）	未実装	
CLOG（トランザクションコミットステータス管理）	CLOG_ZEROPAGE	pg_clogへのページ追加
Heap（ヒープ管理）	HEAP_INSERT	INSERT処理
	HEAP_DELETE	DELETE処理
	HEAP_UPDATE	UPDATE処理
	HEAP_MOVE	VACUUMによるタプルの移動
	HEAP_CLEAN	タプルの消去
Btree（Btree管理）	BTREE_INSERT_LEAF	リーフページの追加
	BTREE_INSERT_UPPER	上位ページの追加
	BTREE_INSERT_META	メタデータの追加
	BTREE_SPLIT_L	ページの分割（左）
	BTREE_SPLIT_R	ページの分割（右）
	BTREE_SPLIT_L_ROOT	rootページの分割（左）
	BTREE_SPLIT_R_ROOT	rootページの分割（右）
	BTREE_DELETE	削除
	BTREE_DELETE_PAGE	ページの削除
	BTREE_DELETE_PAGE_META	メタページの削除
	BTREE_NEWROOT	rootページの追加
	BTREE_NEWMETA	メタページの追加

- リレーションOID
- リレーション内のブロック（ページ）番号
- その中での行のバイトオフセット
- ページに対応した共有バッファ番号
- 行データの長さ
- 列の数

注意していただきたいのは，1つのSQL文が複数の（ときには多数の）ログレコードを生成することがあることです．たとえば，Btreeインデックスを持つテーブルに行を追加した際には，上記のログレコードに加えて，Btreeリソースマネージャに対応したログレコードが作られます．

WALバッファとは

ログレコードは，まず共有メモリ上のWALバッファと呼ばれるメモリ領域に蓄えられます．トランザクションがコミットしたり，WALバッファが満杯になったときにはじめてトランザクションログに書き込まれます．これは，コミットしていないデータに関してはリカバリ時に失われてもどのみち構わないからで，こうすることによって効率を高めることができます．

WALの構造

PostgreSQLでは，トランザクションログは単独のファイルではなく，データベースクラスタの中のpg_xlogというディレクトリの下にいくつかのファイルに分かれて書き込まれます．個々のファイルは**セグメントファイル**と呼ばれ，それぞれ16Mバイトの固定のサイズになっています．

各セグメントファイルの内部は8Kバイトのページに分かれており，各ページの先頭には以下の情報が書き込まれています[注3]．

- マジックナンバー
- フラグ
- スタートアップID
- このページのLSN

セグメントファイルにログレコードが書き込まれる際には，WALバッファのデータに加え，さらに以下の情報が追加されます[注4]．

- CRC[注5]
- 1つ前のログレコードのLSN
- このトランザクションの1つ前のログレコードのLSN
- トランザクションID
- リソースマネージャID
- 処理詳細

LSN

ここでLSN（Log Sequence Number）について説明しておきましょう．個々のログレコードには，一連番号であるLSNが割り当てられます．LSNには次の性質があります．

① 個々のログレコードは重複しないLSNを持ちます
② ログレコードは生成されるたびに単調増加するLSNが割り当てられます

以上から容易に推測されるように，LSNは非常に大きな桁の数字になりますし，32ビットの整数カウンタ程度では簡単に桁溢れが発生します[注6]．

このため，PostgreSQLでは以下のような巧妙な方法でLSNを割り振ります．

① トランザクションログはいくつかの仮想的な「論理ログファイル」構成されるものとします．そして論理ログファイルには「ログファイル番号」という0から始まる番号を付けます．ログファイル番号は2の32乗，すなわち約42億までの値を取ります
② 1つの論理ログファイルは4,278,190,080バイト，すなわち約4Gバイトの大きさを持つものとします．ログレコードが書かれる位置はこの中のバイトアドレスになります．これを「バイトオフセット」と呼ぶことにします
③ ログファイル番号とバイトオフセットを合わせてLSNとします

つまりLSNは概(おおむ)ね64ビット整数の範囲の値を取ることができます[注7]．

ただし，実際に1個のログファイルの大きさを4Gバイトとしてしまうと何かと使い難いので，実際には16Mバイトの大きさを持つセグメントファイルに分割しているわけです．セグメントファイルには以下のルールで名前が付けられています．

① 論理ログファイル番号をゼロサプレスしない8桁の16進数で表します
② バイトオフセットを16Mバイトで割った商をゼロサプレスしない8桁の16進数で表します
③ ①と②を結合してファイル名とします

したがって，論理ログファイル番号＝1でオフセッ

注3）XLogPageHeaderData構造体です．
注4）XLogRecord構造体です．
注5）Cyclic Redundancy Checkの略で，データが壊れていないかどうかチェックするために計算された数値．トランザクションログが正常に記録されているかどうかを判断できます．
注6）32ビット整数の扱える値の上限はせいぜい40億程度です．LSNはそのデータベースのライフサイクルに渡ってカウントアップされていきますので，これでは不足です．
注7）仮に1つのログレコードの大きさの平均が8192バイトで，1日に1億個のログレコードを365日休みなく生成し続けたとしても，LSNを使い尽くすのには6万年以上かかる計算になります．

ト＝4000万のLSNに対応するファイル名は，0000000100000002となります．このように，この方式ではLSNからセグメントファイル名が計算でき，ログレコードにすぐにアクセスできるので非常に都合がよいわけです．

チェックポイント

このように，LSNとWALセグメントの考え方によって，大量のログレコードを管理できるようになりましたが，このままでは無限にログレコードが増え続け，いつかディスクがパンクしてしまいます．そこで考え出されたのが**チェックポイント**（check point）という方法です．

チェックポイントでは，共有バッファ上のデータのうち，更新が行われてディスクへの書き戻しが必要なものを同期書き込みによって強制的にディスクに書き込みます．つまり，チェックポイントを実施することによってその時点でデータベースに投入されたデータがハードディスクに完全に記録されるわけですから，ハードディスクがクラッシュでもしない限りはもう安心です．こうしてチェックポイント以前のログは不要になり，ハードディスクから消去することができるので，ハードディスクが満杯になる心配はなくなります[注8]．

チェックポイントの実行タイミングはpostgresql.confのcheckpoint_segmentsとcheckpoint_timeoutによって制御されます．WALセグメントファイルの数がcheckpoint_segmentsで指定した数（デフォルトは3）に達するか，前回のチェックポイントからcheckpoint_timeout秒（デフォルトは300）経過するとチェックポイント処理が実行されます．このほか，手動でCHECKPOINTコマンドを発行することによってもチェックポイント処理が走ります．

▼図1　pg_controldataの出力

```
$ pg_controldata
pg_control version number:            72
Catalog version number:               200310211
Database cluster state:               in production
pg_control last modified:             2004年03月14日 10時13分57秒
Current log file ID:                  0
Next log file segment:                12
Latest checkpoint location:           0/B954730
Prior checkpoint location:            0/B798340
Latest checkpoint's REDO location:    0/B8C3250
Latest checkpoint's UNDO location:    0/0
Latest checkpoint's StartUpID:        67
Latest checkpoint's NextXID:          7180
Latest checkpoint's NextOID:          710622
Time of latest checkpoint:            2004年03月14日 10時13分53秒
Database block size:                  8192
Blocks per segment of large relation: 131072
Maximum length of identifiers:        64
Maximum number of function arguments: 32
Date/time type storage:               floating-point numbers
Maximum length of locale name:        128
LC_COLLATE:                           C
LC_CTYPE:                             C
```

pg_control

チェックポイントを行った地点は後でリカバリ処理を行うための基準になるものですから，非常に重要です．チェックポイントを実行すると，**チェックポイントレコード**という特殊なログレコードがWALに書かれます．そのLSNは，pg_controlというファイルに記録されます．pg_controlファイルはバイナリファイルなので，簡単には中身を見ることができませんが，そのかわりにpg_controldataというコマンドを使って内容を表示させることができます（図1）．

ここで，「Latest checkpoint location」がチェックポイントレコードのLSNを表しています．図1では，チェックポイントLSNの論理ファイル番号が0，オフセットが16進でB954730だとわかります[注9]．

チェックポイント処理の詳細

このように見てくると，チェックポイント処理は単に共有バッファの内容をディスクに同期書き込みするだけなので簡単そうですが，実際はそうではありません．

チェックポイント中もデータベースの更新要求は発生し，その結果共有バッファの内容が書き換わる可能性があります．これを防ぐためには，すべての共有バ

注8）実際には不要になったログセグメントを削除するのではなく，名前を変えて再利用しています．これは古いファイルを消して新しいファイルを作るよりも高速に処理できるからです．

注9）このほかに「Prior checkpoint location」という項目がありますが，これは前回のチェックポイントレコードの位置です．万が一チェックポイントレコードが読めなかった場合は，前回のチェックポイントレコードを読むことによって，リカバリができない最悪の状態になる可能性を少しでも減らそうとしているわけです．

ッファにロックをかけ，チェックポイント処理が終わるまで待たせておけばよいわけで，このような方式を**シャープチェックポイント**と呼びます．この方式ではチェックポイントレコードのLSNとREDOポイントが一致し，そこから回復処理を開始すればよいので処理が簡単ですが，チェックポイント中はデータベースが実質的に停止してしまうという問題点があります．

ファジーチェックポイント

PostgreSQLで使われているのはチェックポイント中もバッファ全体をロックしない**ファジーチェックポイント**と呼ばれる方式です．この方式では，チェックポイント処理開始時のLSNを**REDOポインタ**として覚えておきます．先ほどのpg_controldataの出力（図1）で言えば，「Latest checkpoint's REDO location」になります．そして，データファイルに対応した共有バッファの同期書き込みが完了した時点でチェックポイントレコードを書き込みます．これが先ほどの例では「Latest checkpoint location」になるわけです．REDOポインタとチェックポイントレコードのLSNが一致していないのは，チェックポイント処理中もトランザクションがコミットされてログレコードが追加されていたからです（図2）．

リカバリ処理

ではいよいよリカバリ処理がどのように行われるのかを見ていきましょう．リカバリ処理の本体はbackend/access/transam/xlog.cに定義されているStartupXLOGに記述されています．詳細に興味のある方はソースコードをご覧ください．

ステップ1：リカバリの必要性の判定

リカバリはデータベースシステムが正常に終了していないときにのみ行います．postmasterが立ち上がったときにpg_controlに書いてあるステータス「Database cluster state」が「in production」であればデータベース稼働中に突然データベースが落ちたと判断してリカバリ処理に移ります[注10]．

ステップ2：チェックポイントレコードの読み込み

前に説明したように，チェックポイントレコードのLSNはpg_controlに記録されています．したがって，それを使ってチェックポイントレコードを読み込むことができます．チェックポイントレコードには以下の情報が含まれます．

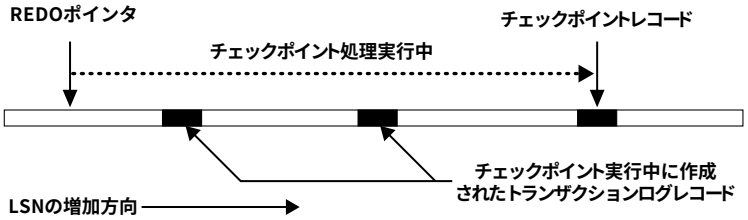
- REDOポインタ
- 利用可能な次のトランザクションID
- 利用可能な次のOID
- チェックポイントを実行した時刻

万が一チェックポイントレコードが読めなかった場合には，1つ前のチェックポイントレコードを読み込みます．もしもそれも読めなかった場合にはリカバリは失敗します．

ステップ3：REDO処理の準備

チェックポイントレコードの情報を使って，共有メモリ上のトランザクションIDとOIDのカウンタを初期化します．次に各リソースマネージャの初期化ルーチン（rm_startup）を実行します．

図2◎ファジーチェックポイント



注10）pg_ctlコマンドのimmediateモード（pg_ctl -m i）で終了させた場合にもリカバリが必要と判断されます．

ステップ4：REDO処理の実行

以上で準備が整ったので，いよいよREDO処理を実行します．

① トランザクションログからログレコードを読み込みます
② もしログレコード中のトランザクションIDが共有メモリ上のトランザクションIDカウンタよりも大きい場合は，トランザクションIDカウンタをログレコード中のトランザクションIDに＋1したものにします
③ もし「バックアップブロック」がログレコード中にある場合は，ディスク上の該当ブロックを置き換えます

バックアップブロックは，チェックポイント後（正確にはREDOポイント以降），テーブルやインデックス上のブロック（ページ）へ最初の更新が行われたときに作られます．

④ 処理の対象となるデータファイル上のページにはLSNが記録されています．ログレコードのLSNと比較し，ログレコードのLSNのほうが小さいか等しい場合には，すでにそのページにログレコードが適用済なので，REDO処理はスキップします[注11]．そうでなければREDOを行います
⑤ もしまだログレコードがあれば，①から繰り返します
⑥ もしもうログレコードがなければ，REDO処理は終わりです．リソースマネージャの終了処理（rm_cleanup）を呼び出し，新しいチェックポイントを作成して終了します

PITR

PostgreSQLでは，トランザクションログを使ったリカバリにより，データベースやOSがダウンした際にもコミットされたトランザクションは保証されます．

しかし，チェックポイント以前のトランザクションログは捨てられていますので，その部分に対応するページが失われるような事態が発生すると，リカバリができません．また，データファイルの作成や消去もログの対象になっていないので，ディスクがクラッシュしてデータファイルが失われる（メディア障害）ようなことになると，やはりリカバリできません．

逆に言うと，データファイルの生成，消滅をログし，かつチェックポイント以前のトランザクションログをどこかに保存するようにしておけば，最悪の事態からのリカバリも可能になるわけです．また，指定した時点までのログのみREDOするようにすれば，「4月1日の15:00の時点のデータベースに戻す」といったこともできるようになります．これは，たとえば誤って大事なデータを消してしまった，などというときに大変役に立ちます．

もちろん，今のバージョンのPostgreSQLでもpg_dumpを使ってバックアップを取っておけば，ディスククラッシュが起きてもバックアップした時点に復旧することはできますが，バックアップ以後に発生したトランザクションのデータは戻ってきません．

レプリケーションはPITRの代りにはならない

クラッシュが発生する直前のデータを保証するために，レプリケーションを使うという考え方もあります．PostgreSQLには標準的なレプリケーションの機能がまだないため，現状ではサードパーティのレプリケーションソフトを使うことになりますが，現存するレプリケーションソフトはすべてラージオブジェクトやOID，シーケンスなどの一部のデータが正確にレプリケーションできないという制約があります．

したがって，レプリケーションがあるからといって，PITRが不要というわけにはいかないのです．

また，レプリケーションでは操作ミスに対しては対処ができません．

PostgreSQL 7.5に向けてのPITR開発の現状

PITRを実現すべく，取り組みが始まっています．今のところ，データファイルの生成，消滅をログするところまでは実装ができています．PITRでは，バックアップにtarなどのOSレベルのバックアップツール

注11）こうして，たとえば同じ行を2回INSERTしてしまうような事態を避けることができます．また，REDO処理が2回以上行われる可能性は常に考慮しておく必要があります．リカバリ処理が途中で失敗した場合は，再度リカバリ処理を試行できなければならないからです．

を使うことになります．当然のことながら，データベースを運用中にtarでバックアップを取ると一貫性がないバックアップとなりますが，REDO処理によって一貫性がある状態に復元できるので問題ありません．むしろ問題は，バックアップ中にチェックポイント処理が走らないように管理しなければならないことです．その他，古くなったトランザクションログを別の場所に保存する機能など，管理ツールが必要になりますが，現在はこれらのツールを作るためにバックエンドの機能をどのように拡張するか，ツールのデザインをどうするか，などの議論が行われています．

新しいリソースマネージャ「Storage」を体験する

前述のように，今のところ，データファイルの生成，消滅をログするところまでは実装ができています．これは新しいリソースマネージャ「Storage」の追加によって実装されています．さっそく試してみましょう．

図3の結果は，3月14日時点のCVSスナップショットテストにコンパイルフラグWAL_DEBUGを追加，postgresql.confにwal_debug=trueを追加してWALログ処理のトレースを行ったものです．

このように，テーブルに対応するファイルを消去しても，きちんとリカバリ処理で復元されています．つまり，ディスク内容が失われるような障害にも対応するための本質的な部分は，実装がきちんとできているのです．PITRの実現はかなり前から言われていることですが，こうして着実に実現に近づいているところを目の当たりにすると感動してしまいます:-)．バージョン7.5の登場が本当に待ち遠しいですね．

▼図3 WALログ処理のトレース

```
test=# CREATE TABLE t1(i INTEGER); ← テーブルを作成
CREATE TABLE

LOG:  INSERT @ 0/B14A08: prev 0/B149C8; xprev 0/0; xid 782: XLOG - nextOid: 25460
LOG:  INSERT @ 0/B14A2C: prev 0/B14A08; xprev 0/0; xid 782: Storage - file create: 17259/17268 ← ファイル名17268でt1に対応するテーブルファイルが作成されたことがログされた
LOG:  INSERT @ 0/B14A54: prev 0/B14A2C; xprev 0/0; xid 782; bkpb 1: Heap - insert: node 17259/1259; tid 3/22
LOG:  INSERT @ 0/B16AAC: prev 0/B14A54; xprev 0/B14A54; xid 782; bkpb 1: Btree - insert: node 17259/16613; tid 1/185
(途中省略)
LOG:  INSERT @ 0/B2F884: prev 0/B2D82C; xprev 0/B2D82C; xid 782; bkpb 1: Btree - insert: node 17259/16625; tid 13/115
LOG:  INSERT @ 0/B318DC: prev 0/B2F884; xprev 0/B2F884; xid 782: Transaction - commit: 2004-03-14 21:24:14
LOG:  xlog flush request 0/B31900; write 0/B2E000; flush 0/B14A08

test=# INSERT INTO t1 VALUES(1); ← 行を追加
INSERT 17270 1

LOG:  INSERT @ 0/B31900: prev 0/B318DC; xprev 0/0; xid 783: Heap - insert: node 17259/17268; tid 0/1
LOG:  INSERT @ 0/B3193C: prev 0/B31900; xprev 0/B31900; xid 783: Transaction - commit: 2004-03-14 21:24:42
LOG:  xlog flush request 0/B31960; write 0/B31900; flush 0/B31900

$ pg_ctl -m i stop ← postmasterを強制終了
```

謝辞

本稿の査読にご協力いただいたNTTデータ先端技術株式会社の井久保寛明氏と，NTTサイバースペース研究所の坂田哲夫氏に深く感謝します．WD

参考文献：

- 『トランザクション処理（上，下）』／ジム・グレイ他著／日経BP
- 『データベースシステム概論』／デイト著／丸善

```
waiting for postmaster to shut down....done
postmaster successfully shut down

LOG:  received immediate shutdown request
WARNING:  terminating connection because of crash of another server process
DETAIL:  The postmaster has commanded this server process to roll back the current transaction and exit, because another server process exited abnormally
and possibly corrupted shared memory.
HINT:  In a moment you should be able to reconnect to the database and repeat your command.

$ ls -l 17268
-rw-------    1 t-ishii  t-ishii      8192 Mar 14 21:24 17268
$ rm 17268 ← t1に対応するファイルを消去
$ postmaster& ← postmasterを起動
[1] 14563
LOG:  database system was interrupted at 2004-03-14 21:23:37 JST ← リカバリ処理の開始
LOG:  checkpoint record is at 0/B149C8
LOG:  redo record is at 0/B149C8; undo record is at 0/0; shutdown TRUE
LOG:  next transaction ID: 779; next OID: 17268
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/B14A08
LOG:  REDO @ 0/B14A08; LSN 0/B14A2C: prev 0/B149C8; xprev 0/0; xid 782: XLOG - nextOid: 25460
LOG:  REDO @ 0/B14A2C; LSN 0/B14A54: prev 0/B14A08; xprev 0/0; xid 782: Storage - file create: 17259/17268 ← 消去したテーブルファイルを復元している
LOG:  REDO @ 0/B14A54; LSN 0/B16AAC: prev 0/B14A2C; xprev 0/0; xid 782; bkpb 1: Heap - insert: node 17259/1259; tid 3/22
[途中省略]
LOG:  REDO @ 0/B31900; LSN 0/B3193C: prev 0/B318DC; xprev 0/0; xid 783: Heap - insert: node 17259/17268; tid 0/1
LOG:  REDO @ 0/B3193C; LSN 0/B31960: prev 0/B31900; xprev 0/B31900; xid 783: Transaction - commit: 2004-03-14 21:24:42
LOG:  record with zero length at 0/B31960
LOG:  redo done at 0/B3193C
LOG:  INSERT @ 0/B31960: prev 0/B3193C; xprev 0/0; xid 0: XLOG - checkpoint: redo 0/B31960; undo 0/0; sui 16; xid 784; oid 25460; shutdown
LOG:  xlog flush request 0/B319A0; write 0/B31960; flush 0/B31960
LOG:  database system is ready

$ psql test
Welcome to psql 7.5devel, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

Pager usage is off.
test=# SELECT * FROM t1; ← 確かにt1が復元されている
 i
---
 1
(1 row)
```