NTTソフトウェア

Background Workerでxxx

2014年12月5日

NTT Software Corporation

Tomonari　Katsumata

はじめに

- **本資料では、PostgreSQLのBackground Workerについて紹介する**
- **Background Workerの仕組みについて理解していただき、興味を持っていただくことを目的としている**

Agenda

1. Background Workerとは？
2. Background Workerでできること/できないこと
3. Background Workerの仕組み
4. Background Workerのサンプル(worker_spi)解説
5. Background Workerデモ

自己紹介

勝俣　智成（かつまた　ともなり）
NTTソフトウェア株式会社　主任エンジニア

- 2002年同社入社
- 数年間は全文検索に関する業務を担当
- PostgreSQLとの出会いは2004年
- PostgreSQLに全文検索機能やXML検索機能などを拡張する開発に従事
- 以降、開発・国内外のPostgreSQLカンファレンスへの参加、社内外でのPostgreSQL研修の講師などを行っている

→PostgreSQLの本出してます！
「内部構造から学ぶ PostgreSQL
　設計・運用計画の鉄則」
よろしく！

Background Workerとは？

Background Workerとは？

- Background Workerとは、PostgreSQL9.3から導入されたユーザが提供する機能を別々のプロセスとして実行する仕組み
- 「別々のプロセス」といっても独立したものではなく、PostgreSQLによって監視され、起動/停止するなど密接に関連している
- PostgreSQL文書では↓で解説されている

  https://www.postgresql.jp/document/9.3/html/bgworker.html

Background Workerで
できること/できないこと

BACKGROUND WORKERを利用することでどのようなことができるようになるのか？

Background Workerでできること

- できること
  - 定期的な裏方(Background)処理
    - PostgreSQLの共有メモリエリアへのアタッチ
    - バックエンドプロセスのように複数のトランザクションの実行
    - libpqを介してクライアントアプリケーションのようにデータベースに接続

通常のバックエンド/バックグラウンドプロセスと同じようなことは大抵できる。

Background Workerでできないこと

- できないこと
  - 表にでること
  - ・・・C言語で実装するのが一般的なので、たいていのことはできてしまう

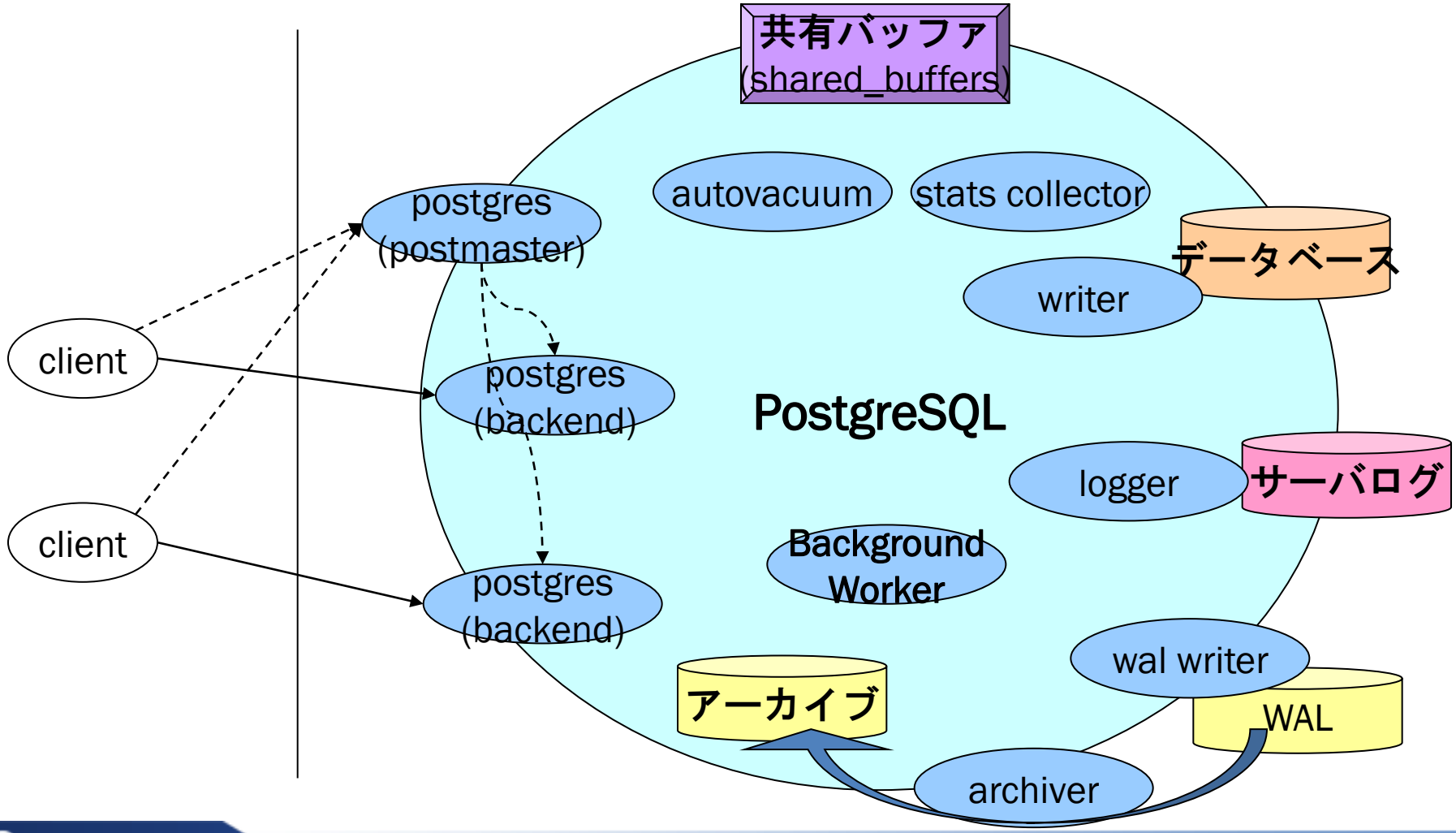
注意！
逆に、なんでもできてしまうのでセキュリティホールにならないようにすべし。
きちんとログ出力するようなものだけを使うなど、十分に注意！！

Background Workerの仕組み

BACKGROUND WORKERの仕組みとして、どのようなタイミングで起動/停止するのか確認する

バックグラウンドプロセス

- PostgreSQLのバックグラウンドプロセス



Background Workerの起動

- 起動のタイミング
  - Background Workerは、_PG_init関数で登録されて、PostgreSQLの起動に合わせて起動する
  - 「PostgreSQLの起動」がどこを指すかは、bgw_start_timeにオプション指定することで制御できる

bgw_start_timeに設定する値	意味
BgWorkerStart_PostmasterStart	初期化を終えたらすぐ起動
BgWorkerStart_ConsistentState	参照のみのクエリを受け付けられるようになったら起動
BgWorkerStart_RecoveryFinished	参照/更新クエリを受け付けられるようになったら起動

Background Workerの停止

- 停止のタイミング
  - Background Workerは、PostgreSQLの停止に合わせて停止する
  - 思いがけない停止(クラッシュ)時に、PostgreSQLから再起動を行うか否かをbgw_restart_timeで設定できる

bgw_restart_timeに設定する値	意味
正の数	再起動まで待つ間隔(秒単位)
BGW_NEVER_RESTART	再起動させない

Background Workerの動的起動

- PostgreSQL9.4からは動的にBackground Workerを起動できる
  - RegisterDynamicBackgroundWorker関数を呼びだすことで、登録する
  - WaitForBackgroundWorkerStartup関数を呼びだすことで、起動する
  - GetBackgroundWorkerPid関数を呼びだすことで状態を監視する
  - TerminateBackgroundWorker関数を呼びだすことで、停止する

Background Workerのサンプル worker_spi解説

CONTRIBに含まれるBACKGROUND WORKERのサンプルである
WORKER_SPIの挙動を解説する

worker_spi

- contribモジュールのひとつとしてworker_spiが提供されている
  - Background Workerのサンプルとして存在している
- 残念なことに全くもって文書化されていないので、ここで解説していく

worker_spiの機能

- worker_spiでは、DB接続＆テーブル作成をして、定期的にそのテーブルの要約をつくる
- 上記機能を補助する形で、以下の様々な処理を行っている
  - DBへの接続
  - トランザクション処理(with SPI)
  - GUCパラメータ処理(and reloading)
  - pg_stat_activityとのコラボ
  - Latch機構の利用
  - 動的起動

worker_spiーインクルードファイルー

```
23 #include "postgres.h"
24
25 /* These are always necessary for a bgworker */
26 #include "miscadmin.h"
27 #include "postmaster/bgworker.h"
28 #include "storage/ipc.h"
29 #include "storage/latch.h"
30 #include "storage/lwlock.h"
31 #include "storage/proc.h"
32 #include "storage/shmem.h"
33
34 /* these headers are used by this particular worker's code */
35 #include "access/xact.h"
36 #include "executor/spi.h"
37 #include "fmgr.h"
38 #include "lib/stringinfo.h"
39 #include "pgstat.h"
40 #include "utils/builtins.h"
41 #include "utils/snapmgr.h"
42 #include "tcop/utility.h"
```

Background Workerを作る際には必須になるヘッダファイル達

実行する処理に応じて必要なヘッダファイルは適宜インクルード

worker_spiー構造体初期化ー

```
345     /* set up common data for all our workers */
346     worker.bgw_flags = BGWORKER_SHMEM_ACCESS |
347             BGWORKER_BACKEND_DATABASE_CONNECTION;
348     worker.bgw_start_time = BgWorkerStart_RecoveryFinished;
349     worker.bgw_restart_time = BGW_NEVER_RESTART;
350     worker.bgw_main = worker_spi_main;
351     worker.bgw_notify_pid = 0;
352
353     /*
354      * Now fill in worker-specific data, and do the actual re
355      */
356     for (i = 1; i <= worker_spi_total_workers; i++)
357     {
358             snprintf(worker.bgw_name, BGW_MAXLEN, "worker %d", i);
359             worker.bgw_main_arg = Int32GetDatum(i);
360
361             RegisterBackgroundWorker(&worker);
362     }
```

BackgroundWoker型の構造体にどんなタイミングで何をするかを設定（詳細は次頁）

worker_spi－BackgroundWorker型－

```
typedef struct BackgroundWorker
{
        char bgw_name[BGW_MAXLEN];
        int bgw_flags;
        BgWorkerStartTime bgw_start_time;
        int bgw_restart_time; /* in seconds, or BGW
        bgworker_main_type bgw_main;
        char bgw_library_name[BGW_MAXLEN]; /* only if bgw_main is NULL */
        char bgw_function_name[BGW_MAXLEN]; /* only if bgw_main is NULL */
        Datum bgw_main_arg;
        int bgw_notify_pid;
} BackgroundWorke
```

Background Workerの名前。
psコマンドとかで見れる。

共有バッファ/DBへのアクセスがあるかどうかのフラグ
BGWORKER_SHMEM_ACCESS、
BGWORKER_BACKEND_DATABASE_CONNECTION

プロセスID。
登録後、起動を待たないなら0、
それ以外はMyProcPidを設定

メイン処理の関数ポインタと動的起動時用のライブラリ名、関数名。
引数はbgw_main_argに1つだけ与えることが可能

worker_spiーエッセンス(1)ー

- DB接続
  - BackgroundWorkerInitializeConnection関数を用いて、特定のDBへ接続する

```
183     /* Connect to our database */
184     BackgroundWorkerInitializeConnection("postgres", NULL);
```

- 第1引数＝DB名、第2引数=ユーザ名。
  - DB名をNULLにすると、共有カタログへのアクセスのみ実施できる
  - ユーザ名をNULLにするとinitdb実行ユーザでアクセス

worker_spiーエッセンス(2)ー

- トランザクション処理（with SPI）
  - 以下の流れがお決まり

```
 ・・・・・・・・
264         SetCurrentStatementStartTimestamp();
265         StartTransactionCommand();
266         SPI_connect();
267         PushActiveSnapshot(GetTransactionSnapshot());
 ・・・・・・・・
270         /* We can now execute queries via SPI */
271         ret = SPI_execute(buf.data, false, 0);
 ・・・・・・・・
294         SPI_finish();
295         PopActiveSnapshot();
296         CommitTransactionCommand();
```

トランザクション、クエリの発行時間を最新にする

MVCCの管理のため実行する

worker_spiーエッセンス(3)ー(1/3)

- GUCパラメータ処理
  - DefineCustomIntVariable関数で値を取得

```
55 /* GUC variables */
56 static int     worker_spi_naptime = 10;
57 static int     worker_spi_total_workers = 2;
・・・・・・・・・
315       /* get the configuration */
316       DefineCustomIntVariable("worker_spi.naptime",
・・・・・・・・・
327                                       NULL);
・・・・・・・・・
332       DefineCustomIntVariable("worker_spi.total_workers",
・・・・・・・・・
343                                       NULL);
・・・・・・・・・
```

worker_spiーエッセンス(3)ー(2/3)

- DefineCustomIntVariable関数に与える値

第N引数	意味	第N引数	意味
1	名前	7	最大値
2	短い説明	8	変更のタイミング
3	長い説明	9	フラグ
4	格納する変数のポインタ	１０	チェック処理(Hook)
5	デフォルト値	１１	アサイン処理(Hook)
6	最小値	１２	表示処理(Hook)

- 他にもbool型や文字列型のパラメータを扱う関数も用意されている
  - 詳細は、src/include/utils/guc.h に。

worker_spiーエッセンス(4)ー(1/2)

- リロード処理
  - シグナル(SIGHUP)を受け取って、設定を再読込

```
51 /* flags set by signal handlers */
52 static volatile sig_atomic_t got_sighup = false;


88 static void
89 worker_spi_sighup(SIGNAL_ARGS)
90 {
91      int             save_errno = errno;
92
93      got_sighup = true;
94      if (MyProc)
95          SetLatch(&MyProc->procLatch);
96
97      errno = save_errno;
98 }
```

フラグgot_sighupを定義

Signalハンドラを
独自に作成

worker_spiーエッセンス(4)ー(2/2)

- リロード処理
  - シグナル(SIGHUP)を受け取って、設定を再読込

```
176     /* Establish signal handlers before unblocking signals. */
177     pqsignal(SIGHUP, worker_spi_sighup);
178     pqsignal(SIGTERM, worker_spi_sigterm);
179
180     /* We're now ready to receive signals */
181     BackgroundWorkerUnblockSignals();



242         if (got_sighup)
243         {
244             got_sighup = false;
245             ProcessConfigFile(PGC_SIGHUP);
246         }
```

pqsignalで
singalハンドラを設定

BackgroundWorkerUnblockSignals関数で設定を有効にする

メインループの中でgot_sighupがtrueだったら、ProcessConfigFileで再読込

worker_spiーエッセンス(5)ー

- pg_stat_activityとのコラボ
  - pgstat_report_activity関数で値を設定

```
・・・・・・・・・
268         pgstat_report_activity(STATE_RUNNING, buf.data);
・・・・・・・・・
297         pgstat_report_activity(STATE_IDLE, NULL);
```

  - 第1引数＝BackendState
  - 第2引数＝クエリ文字列

```
typedef enum BackendState
{
        STATE_UNDEFINED,
        STATE_IDLE,
        STATE_RUNNING,
        STATE_IDLEINTRANSACTION,
        STATE_FASTPATH,
        STATE_IDLEINTRANSACTION_ABORTED,
        STATE_DISABLED
} BackendState;
```

worker_spiーエッセンス(6)ー(1/2)

- Latch機構の利用
  - Latch機構を使うことで、「定期的な」処理を行うことが可能
  - 主に利用するのは、下記3種の関数

関数名	処理内容
SetLatch	ラッチを設定する
ResetLatch	ラッチを解除して、設定可能状態にする
WaitLatch	ラッチが設定されるのをまつ

  - 典型的な利用方法は→の通り
    - Please read latch.h

```
* for (;;)
* {
*       ResetLatch();
*       if (work to do)
*             Do Stuff();
*       WaitLatch();
* }
```

worker_spiーエッセンス(6)ー(2/2)

- Latch機構の利用

```
・・・・・・・・・
93      got_sighup = true;
94      if (MyProc)
95          SetLatch(&MyProc->procLatch);
96
・・・・・・・・・
230         rc = WaitLatch(&MyProc->procLatch,
231                        WL_LATCH_SET | WL_TIMEOUT | WL_POSTMASTER_DEATH,
232                        worker_spi_naptime * 1000L);
233         ResetLatch(&MyProc->procLatch);
・・・・・・・・・
242         if (got_sighup)
243         {
244             got_sighup = false;
245             ProcessConfigFile(PGC_SIGHUP);
246         }
```

MyProcのprocLatchで特定する

ラッチが設定されるor
タイムアウトするor
Postmasterが逝く
のどれかで待ちをやめる

worker_spiーエッセンス(7)ー(1/3)

- 動的起動
  - 前述の通り、登録/起動/監視/停止する関数が用意されている
  - 構造体の初期化で異なるのは↓の太字部分

```
377     worker.bgw_flags = BGWORKER_SHMEM_ACCESS |
378         BGWORKER_BACKEND_DATABASE_CONNECTION;
379     worker.bgw_start_time = BgWorkerStart_RecoveryFinished;
380     worker.bgw_restart_time = BGW_NEVER_RESTART;
381     worker.bgw_main = NULL;       /* new worker might not have library loaded */
382     sprintf(worker.bgw_library_name, "worker_spi");
383     sprintf(worker.bgw_function_name, "worker_spi_main");
384     snprintf(worker.bgw_name, BGW_MAXLEN, "worker %d", i);
385     worker.bgw_main_arg = Int32GetDatum(i);
386     /* set bgw_notify_pid so that we can use WaitForBackgroundWorkerStartup */
387     worker.bgw_notify_pid = MyProcPid;
```

worker_spiーエッセンス(7)ー(2/3)

- 動的登録と起動

```
389     if (!RegisterDynamicBackgroundWorker(&worker, &handle))
390         PG_RETURN_NULL();
391
392     status = WaitForBackgroundWorkerStartup(handle, &pi
393
394     if (status == BGWH_STOPPED)
395         ereport(ERROR,
396                 _RESOURCES),
397                 errmsg("could not start background process"),
398                 errhint("More details may be available in the server log.")));
399     if (status == BGWH_POSTMASTER_DIED)
400         ereport(ERROR,
401                 (errcode(ERRCODE_INSUFFICIENT_RESOURCES),
402                 errmsg("cannot start background processes without postmaster"),
403                 errhint("Kill all remaining database processes and restart the
database.")));
404     Assert(status == BGWH_STARTED);
```

ハンドラとして BackgroundWorkerHandle を渡す

PIDが戻る

worker_spiーエッセンス(7)ー(3/3)

- **動的起動のきっかけ**
  - worker_spiでは、**動的起動のきっかけとしてユーザ定義関数を用いてる**

```
. . . . . .
368 Datum
369 worker_spi_launch(PG_FUNCTION_ARGS)
370 {
371     int32       i = PG_GETARG_INT32(0);
. . . . . .
```

  - **引数(整数値)は　workerの名前(bgw_name)に利用される**

Background Workerデモ

BACKGROUND WORKERを利用することでどのようなことができるようになるのか？

Background Workerデモ－1－

- とりあえず、worker_spi動かしてみる
  - shared_preload_librariesにworker_spiを追加
  - 起動
  - 確認(psコマンド&pg_stat_activityで)
  - countedテーブルに'total'データ、'delta'データ格納して¥watchしてみる
- ついでに動的起動も確認する
  - きっかけの関数を登録
  - ランチャ起動
    - SELECT worker_spi_launch(xx);

おわりに

- 本資料では下記についてまとめた
  - Background Workerの概要
  - Background Workerの仕組み
  - Background Workerサンプルのエッセンス
- また、Background Wokrerのデモを行い、動作イメージを持っていただいた

なんとなく理解できましたでしょうか？
タイトルにある「ｘｘｘ」は、なんでもできるということを意味してます。
少しでも興味を持ったら、Let's Try！！

ご静聴ありがとうございました！

Enjoy!