libpgprobackup Documentation

libpgprobackup Documentation

libpgprobackup

libpgprobackup

libpgprobackup — library with API to back up data, restore from backups, as well as validate and merge backups

Description

The libpgprobackup library contains functions to back up data, restore from backups, as well as validate and merge backups. The API provided enables you to create your own application to back up and restore data instead of using the command-line utility pg_probackup.

The library stores backups in files of its own format.

The library takes over the interaction with the database server, data processing, coding, and decoding, as well as creation and storing the metadata for backups.

An application that uses the library must provide it with access to the storage, such as a file system, S3 storage, or tape. However, file system operations, such as reading and storing backups, as well as storing the metadata, are the responsibility of the application. The application interacts with libpgprobackup as follows:

1. The application prepares the database instance for backing up and calls the libpgprobackup backup function.
2. libpgprobackup transforms the data files and WAL files to the pg_probackup3 format and redirects them to the application.
3. The client application sends the data to a file on the target storage (file system, S3 storage, or tape) and saves the backup metadata. Note that no intermediate storage is used.
4. To restore from a backup by calling the libpgprobackup restore function, to perform the backup integrity check by calling the validate function, or to merge backups by calling the merge function, the application provides the library with functions to get the backup data and metadata.

libpgprobackup is implemented in C++, but it can be used in applications written in different programming languages. The integration with C/C++ applications must be least complicated. The library uses the `extern "C"` calling convention.

libpgprobackup structures and functions are declared in the in the `probackup_lib.h` file.

Functions

The libpgprobackup library contains functions described below. When library functions are called, they accept structures with command parameters and a structure with pointers to functions that work with files and metadata.

`bool backup ( ConnectOptionsLib *conn_opt, BackupOptionsLib *backup_opt, MetadataProviderLib *metadata );`

Connects to the local or remote server and performs a backup. Returns `true` in the case of success and `false` in case of error.

Arguments:

- *`conn_opt`* — pointer to the `ConnectOptionsLib` structure that defines the options to connect to the Postgres Pro server, such as `pgdatabase`, `pghost`, `pgport`, `pguser`, `password`.
- *`backup_opt`* — pointer to the `BackupOptionsLib` structure that contains the backup options, such as the backup mode or number of threads.
- *`metadata`* — pointer to the `MetadataProviderLib` structure that provides callback functions for processing of backup metadata and file operations.

`bool restore ( RestoreOptionsLib *restore_opt, MetadataProviderLib *metadata );`

Restores data from a backup using the parameters passed in the appropriate structures to the local file system. Returns `true` in the case of success and `false` in case of error.

Arguments:

- *`restore_opt`* — pointer to the `RestoreOptionsLib` structure that contains the options to restore from a backup.
- *`metadata`* — pointer to the `MetadataProviderLib` structure that provides callback functions for processing of backup metadata and file operations.

`bool validate ( RestoreOptionsLib *restore_opt, MetadataProviderLib *metadata, bool withParents );`

Verifies that all the files required to restore the cluster from a backup are present and are not corrupt. If *`withParents`* is true, all the backups in the chain of parents are also validated. Returns `true` in the case of success and `false` in case of error.

Arguments:

- *`restore_opt`* — pointer to the `RestoreOptionsLib` structure that contains the options for integrity check of a backup.
- *`metadata`* — pointer to the `MetadataProviderLib` structure that provides callback functions for processing of backup metadata and file operations.
- *`withParents`* — a boolean value that defines whether parent backups must also be validated.

`bool merge ( MergeOptionsLib *merge_opt, MetadataProviderLib *metadata, bool with_file_map );`

Merges a chain of incremental backups into one full backup. During the merge a new full backup created includes all the parent backups. If the parent backups do not have additional dependencies, they are removed after a successful merge. Returns `true` in the case of success and `false` in case of error.

Arguments:

- *`merge_opt`* — pointer to the `MergeOptionsLib` structure that contains the options for merging backups.
- *`metadata`* — pointer to the `MetadataProviderLib` structure that provides callback functions for processing of backup metadata and file operations.
- *`with_file_map`* — enables file map generation in metadata.

`void set_probackup_logger ( Logger info, Logger warning, Logger error, Logger debug );`

Defines three callback logger functions that libpgprobackup will use to output debug, information, warning, or error messages. Each of these functions must accept a string parameter with the message to be passed by the library.

Arguments:

- *`info`* — pointer to the function that will process information messages.
- *`warning`* — pointer to the function that will process warnings.
- *`error`* — pointer to the function that will process error messages.
- *`debug`* — pointer to the function that will process debug messages.

`void get_api_info`

Returns the `ProBackupApiInfo` structure that contains information about the API version, compiler and operation system.

`void set_cancel`

Signals the current command to initiate graceful termination. The command will complete its current operation, perform necessary cleanup procedures, set the backup status to `ERROR`, and terminate execution.

Structures to Work with Files

The library gets feedback from the application through the structure of the `MetadataProviderLib` type, which must contain pointers to functions that work with files and metadata.

Callbacks from the library to functions passed by the application are used. Pointers to functions are passed to the library through the `MetadataProviderLib` structure. File read or write operations are defined in the `CSource` and `CSink` structures, respectively.

For the feedback from the application, library structures contain the `void *thisPointer` pointer, where the application can store the pointer to its own function or class instance. This pointer is passed to callback functions when they are called from the library.

MetadataProviderLib

A pointer to this structure is passed to all the main librobackup functions, such as backup, restore, validate, and merge. This structure defines methods to work with backup data, to write and read backup metadata, and to get the list of backups.

```
typedef struct
{
    CSinkArray *(*get_sinks_for_backup)(const char *backup_id, size_t nsinks, void
 *thisPointer);
    CSourceArray *(*get_sources_for_backup)(const char *backup_id, void *thisPointer);

    CSink *(*get_sink_for_backup_map)(const char *backup_id, void *thisPointer);
    CSource *(*get_source_for_backup_map)(const char *backup_id,
                                          void       *thisPointer);

    void (*register_backup)(PgproBackup *backup, void *ptr);
    PgproBackup *(*get_backup_by_id)(const char *backup_id, void *thisPointer);
    void (*free_backup)(PgproBackup *backup, void *thisPointer);
    char **(*list_backup_ids)(void *thisPointer);

    bool (*write_backup_status)(const char *backup_id, BackupStatus status,
                                void *thisPointer);

    void *thisPointer;
} MetadataProviderLib;
```

Where:

get_sinks_for_backup

A pointer to the function that returns `CSinkArray` — an array of pointers to the `CSink` structure (see `CSource` and `CSink` for more details). Needed for writing information into the backup. *backup_id* contains the string with the backup ID. In *thisPointer*, the application gets the pointer that was previously passed to the library through the structure.

get_sources_for_backup

A pointer to the function that returns `CSourceArray` — an array of pointers to the `CSource` structure (see `CSource` and `CSink` for more details). Needed for reading information from the backup. *backup_id* contains the string with the backup ID. In *thisPointer*, the application gets the pointer that was previously passed to the library through the structure.

get_sink_for_backup_map

A pointer to the function that returns a pointer to `CSink` structure (see `CSource` and `CSink` for more details). Needed for writing information into the backup metadata about the backup map. *backup_id* contains the string with the backup ID. In *thisPointer*, the application gets the pointer that was previously passed to the library through the structure.

get_source_for_backup_map

A pointer to the function that returns a pointer to the `CSource` structure (see `CSource` and `CSink` for more details). Needed for reading information about the backup map from the backup metadata. *backup_id* contains the string with the backup ID. In *thisPointer*, the application gets the pointer that was previously passed to the library through the structure.

register_backup

A pointer to the function that stores the backup metadata. Accepts the `PgproBackup` structure.

get_backup_by_id

A pointer to the function that gets metadata of the backup defined by *backup_id*.

free_backup

Frees the memory for the `PgproBackup` structure returned by `get_backup_by_id`.

list_backup_ids

Returns the list of available backup IDs. The list contains pointers to strings with the zero character at the end. The last pointer in the list must also be zero. The memory for the list must be allocated by a C function such as `malloc` or `strdup` because the library frees the memory using the `free` function.

write_backup_status

A pointer to the function that saves the backup status. The backup status is updated separately from saving the rest of the backup metadata.

*void *thisPointer*

A pointer to be passed from the library to all callback functions.

`CSource` and `CSink`

These structures are needed for reading and writing data blocks to a backup file. Arrays of pointers to these structures must be returned by the get_sources_for_backup and get_sinks_for_backup functions, respectively. Each of these structures is actually a backup file that is open for reading or writing, respectively.

```
/* Support structure */
typedef struct
{
 unsigned char *ptr;
 size_t         len;
} c_buffer_t;

typedef struct
{
 c_buffer_t (*read_one)(void *thisPointer);
 ssize_t (*read)(char *buf, size_t size, void *thisPointer);
 void (*seek)(off_t offset, void *thisPointer);
 off_t (*get_offset)(void *thisPointer);
 void (*close)(void *thisPointer);
 void *thisPointer;
} CSource;
```

```
typedef struct
{
 /* Write one Pb structure. See @PbStructs. */
 size_t (*write_one)(uint8_t *buf, size_t size, void *thisPointer);
 ssize_t (*write)(char *buf, size_t size, void *thisPointer);
 /* Close this Sink. No more calls will be done. */
 void (*close)(void *thisPointer);
 void *thisPointer;
} CSink;

typedef struct
{
 size_t nsinks;
 CSink **array;
} CSinkArray;

typedef struct
{
 size_t nsources;
 CSource **array;
} CSourceArray;
```

In the `thisPointer` field, an application can pass to the structure a pointer to a structure or application class that, for instance, contains a descriptor of an open file.

Functions from the `CSource` structure are used for reading the backup file, while functions from the `CSink` structure are used for writing the backup file. Each of these structures currently has two functions for that. The `read_one` and `write_one` functions perform lower-level operations with a data block. `write_one` saves the block size, while `read_one` first reads the block size and then the data block of this size.

`read` and `write` functions are used for simpler implementation of reading/writing. Their arguments are similar to those of common functions to work with files. Processing of the block size is done inside the library. These functions must return the size of the read/written data or 1 in case of error.

The `seek` function is optional and currently only works together with file maps. If implementing `seek` would be too costly for your storage system, you can omit it.

The `get_offset` function returns the current read position in the specified output target (*sink*).

The `close` function closes the file.

PgproBackup

Backup metadata is passed in this structure.

```
typedef struct
{
    BackupStatus     backup_status;
    BackupMode       backup_mode;      /* Mode - one of BACKUP_MODE_xxx */
    char            *backup_id;        /* Identifier of the backup. */
    char            *parent_backup_id; /* Identifier of the parent backup. */
    BackupTimeLineID tli;       /* timeline of start and stop backup lsns */
    BackupXLogRecPtr start_lsn; /* backup's starting transaction log location */
    BackupXLogRecPtr stop_lsn; /* backup's finishing transaction log location */
    BackupTimestampTz   start_time; /* UTC time of backup creation */
    BackupTransactionId minxid;     /* min Xid for the moment of backup start */
    BackupMultiXactId
            minmulti; /* min multixact for the moment of backup start */
```

```
    bool stream; /* Was this backup taken in stream mode? I.e. does it include
                    all needed WAL files? */
    bool  from_replica;     /* Was this backup taken from replica */
    char *primary_conninfo; /* Connection parameters of the backup in the format
                               suitable for recovery.conf */
    char *note;

    /* For compatibility check */
    uint32_t block_size;
    uint32_t wal_block_size;
    char    *program_version;
    int      server_version;

    size_t uncompressed_bytes; ///< Size of data and non-data files before
                               ///< compression is applied
    size_t data_bytes; ///< Size of data and non-data files after compression is
                       ///< applied
    CompressAlg compress_alg;
    int         compress_level;

    BackupTimestampTz
            end_time; /* the moment when backup was finished, or the moment
                       * when we realized that backup is broken */
    BackupTimestampTz
            end_validation_time; /* UTC time when validation finished */

    BackupSource backup_source;/* direct, base or pro backup method*/

    size_t wal_bytes;// not used in pb3
    BackupTimestampTz   recovery_time;
} PgproBackup;
```

After execution of the backup function, the library calls a callback function register_backup, where the filled-in `PgproBackup` structure is passed. Software engineers can save the passed information at their discretion.

To get information on an existing backup, the library uses a callback function get_backup_by_id. From this function, the application must return a pointer to the `PgproBackup` structure with the metadata of the backup having the specified ID or a zero pointer in case of error.

To free the memory for the `PgproBackup` structure, the application must call the callback function free_backup.

ProBackupApiInfo

This structure contains the following libpgprobackup API information:

- libpgprobackup version:
  - Version components (major, minor, patch)
  - User-displayable version string
- Supported compression algorithms
- Compiler version
- Operating system version
- Build type

```
typedef struct tag_ProBackupApiInfo {
```

```
 const int pbk_ver_major;
 const int pbk_ver_minor;
 const int pbk_ver_patch;

 /* User-visible probackup lib version. */
 const char *version;
 /* Used for dynamic linking */
 const int dynamic_api_version;
 /* List of supported compressions. */
 const char **supported_compressions;
 /* Compiler info */
 const char *compiler_info;
 /* OS info */
 const char *os_info;
 /* Release/Debug */
 const char *build_type;
} ProBackupApiInfo;
```

`dynamic_api_version` helps to prevent incompatible versions of the library from being used by clients. `supported_compressions` is a null-terminated list of supported compression algorithms.

The structure is statically allocated — applications must not attempt to free its memory.

Use get_api_info to get a pointer to this structure.

Command Options

This section lists structures that are used to pass options to the commands. For more details, see the library header file `probackup_lib.h`.

connectOptionsLib

The following structure defines options to connect to the Postgres Pro server:

```
typedef struct connectOptionsLib
{
 /* The database name. */
 const char *pgdatabase;
 /* Name of host to connect to. */
 const char *pghost;
 /* Port number to connect to at the server host, or socket file name
  * extension for Unix-domain connections.*/
 const char *pgport;
 /* PostgreSQL user name to connect as. */
 const char *pguser;
 /* Password to be used if the server demands password authentication. */
 const char *password;
} ConnectOptionsLib;
```

backupOptionsLib

The structure below defines options to create a backup. Note that the `FULL` and `DELTA` backup modes are only supported so far.

```
typedef struct backupOptionsLib
{
 /* Number of threads, if backup mode supports multithreading */
 int num_threads;
 /* Backup mode PAGE, PTRACK, DELTA, AUTO, FULL*/
 BackupMode backup_mode;
```

```
 /* Backup source DIRECT, BASE or PRO */
 BackupSource backup_source;
 /* For DIRECT source is required to set up PGDATA. It is not required for
  * other sources */
 const char *pgdata;
 /* Backup Id if you wants to use custom id, otherwise it will be generated a
  * unique id using datetime of creation */
 const char *backup_id;
 /* Id of parent backup. It is not required for FULL backup mode */
 const char *parent_backup_id;
 /* Name of replication slot, if you use custom slot created by
  * pg_create_physical_replication_slot(). Otherwise will be used auto
  * generated slot */
 const char *replication_slot;
 bool        create_slot;
 const char *backup_note;
 /* If true, then WAL will be sent in stream mode, otherwise in archive mode
  */
 bool stream_wal;
 /* Progress flag. If true progress will be logged */
 bool        progress;
 const char *external_dir_str;
 /* Verify check sums. It is available only if checksums turn on on
  * PostgresPro server */
 bool verify_checksums;
 /* Compression algorithm, that is used for sending data between PostgresPro
  * server and libpgprobackup */
 CompressAlg compress_alg;
 /* Compression level, that is used for sending data between PostgresPro
  * server and libpgprobackup */
 int compress_level;
 /* Wait timeout for WAL segment archiving */
 uint32_t archive_timeout;
 /* Number of backup segment files. */
 uint32_t num_segments;
 /* Should we send raw 8k pages, or package them into larger groups. */
 TransferMode transfer_mode;
 /* Number of threads which write into segments. */
 uint32_t num_write_threads;
 /* Number of threads used for backup validation. */
 uint32_t num_validate_threads;
} BackupOptionsLib;
```

restoreOptionsLib

The following structure defines options to restore from a backup. They are also used for backup validation:

```
typedef struct restoreOptionsLib
{
 /* Number of threads */
 uint32_t    num_threads;
 /* Path to pgdata for restoration */
 const char *pgdata;
 /* Backup ID for restoration */
 char       *backup_id;
 /* Progress flag. If true progress will be logged */
 bool        progress;
 /* Skip external directories */
```

```
    bool        skip_external_dirs;
    const char *external_mapping;
    const char *tablespace_mapping;
    const char *db_str_oids;
    const char *db_str_names;
    PartialRestoreType restoreTyp
    /* Should we verify postgres page checksumse */
    bool        verify_checksums;

    /* Restore command to write in postgresql.auto.conf */
    /* https://postgrespro.ru/docs/enterprise/16/runtime-config-wal#GUC-RESTORE-COMMAND */
    const char *config_content;
    /* Need recovery signal */
    bool        need_recovery_signal;
    /* Need standby signal */
    bool need_standby_signal;
    /* No synchronization */
    bool no_sync;
    /* Number of threads to write restored data */
    uint32_t num_write_threads;
    /* Number of threads used for validation */
    uint32_t num_validate_threads;
   } RestoreOptionsLib;
```

mergeOptionsLib

The following structure defines options to merge backups:

```
   typedef struct mergeOptionsLib
   {
    /* Number of threads */
    int         num_threads;
    /* Path to pgdata to restore to */
    const char *pgdata;
    /* Incremental backup ID for the merge */
    const char *backup_id;
    /* Target backup ID for the merge */
    const char *target_backup_id;
    /* Progress flag. If true progress will be logged */
    bool        progress;
    /* ID of the last increment */
    const char *merge_from_id;
    /* Time interval within which to merge backups */
    int         interval;
    /* Number of threads to write merged data */
    uint32_t    num_write_threads;
    /* Number of threads used for validation */
    uint32_t    num_validate_threads;
   } MergeOptionsLib;
```

Constants

BackupMode

```
   /*
    Backup Mode is how backup is taken.
    All DIFF modes require parent backup id passed in the BackupOptions.
    Parent backup id is ignored in the FULL mode.
```

```
 DIFF_AUTO returns selected mode in the metadata. Tentatively it prefers
 DIFF_PAGE if WAL summarization is available, if not it tries to do
 DIFF_PTRACK if ptrack is enabled and finally it falls back to DIFF_DELTA.
 In case when even DIFF_DELTA is not possible (no parent full backup exists)
 FULL backup is taken.
*/
typedef enum BackupMode
{
    BACKUP_MODE_INVALID = 0,
    BACKUP_MODE_DIFF_PAGE,   /* incremental page backup */
    BACKUP_MODE_DIFF_PTRACK, /* incremental page backup with ptrack system */
    BACKUP_MODE_DIFF_DELTA,  /* incremental page backup with lsn comparison */
    BACKUP_MODE_DIFF_AUTO, /* library selects diff backup mode automatically */
    BACKUP_MODE_FULL       /* full backup */
} BackupMode;
```

CompressAlg

```
/*
* Compression mode which is used to transfer data between PG server and the client lib.
* Default is NONE_COMPRESS.
*/
typedef enum CompressAlg
{
    NONE_COMPRESS = 0,
    ZLIB_COMPRESS,
    LZ4_COMPRESS,
    ZSTD_COMPRESS,
} CompressAlg;
```

BackupSource

```
/**
Backup Source is a method used by the client to access PGDATA.
*/
typedef enum
{
    /*
     * Direct access. The client reads data files directly.
     * Opens normal connection to execute PG_BACKUP_START/STOP.
     * Local file access. Multithreaded. No special PostgresPro edition
     * required.
     */
    BACKUP_SOURCE_DIRECT,
    /*
     * Uses pg_basebackup protocol.
     * Opens replication connection.
     * Remote file access. No special PostgresPro edition required.
     */
    BACKUP_SOURCE_BASE_BACKUP,
    /*
     * Uses pg_probackup protocol.
     * Opens replication connection.
     * Remote file access. Multithreaded. Only works with PostgresPro builds.
     * Supported PostgresPro versions start with ent-15.
     */
    BACKUP_SOURCE_PRO_BACKUP
} BackupSource;
```

Index

L