Расширение PHP для работы с YottaDB
Проект реализации

Данный документ является описанием предполагаемой реализации расширения языка PHP, предназначенного для работы с СУБД YottaDB. Реализация базируется на описанном ранее программном интерфейсе для данного расширения, который был приведён в виде отдельного документа.

1. Общая структура расширения

Предполагается, что код данного расширения будет выполняться в двух отдельных процессах: часть кода будет выполняться в процессе интерпретатора PHP, а часть — в дополнительном сопроцессе. Это нужно, чтобы избежать конфликта, возникающего из-за того, что PHP и YottaDB могут назначать разные обработчики одним и тем же сигналам. Также это позволяет избежать определения «чужих» переменных окружения в адресном пространстве интерпретатора PHP.

Код, выполняемый в интерпретаторе PHP, в большинстве случаев, является «заглушкой» — его задача подготовить аргументы, переданные пользователем, и сформировать на их основе запрос, который будет отправлен сопроцессу, затем, дождаться от сопроцесса ответа и вернуть пользователю полученый в этом ответе результат.

Сопроцесс выполняет всю работу с базой данных. Получив запрос, он определяет, какое именно действие необходимо выполнить, затем выполняет вызов процедуры MUMPS, соответствующей этому действию и возвращает ответ, поместив в него возвращаемое значение процедуры, если оно имеется.

Взаимодействие кода расширения, находяшегося в интерпретаторе PHP, и работающего с базой данных сопроцесса будет выполняться с помощью протокола *JSON-RPC* [1]. Данные между процессами будут передаваться через неименованые каналы в виде текста, без какого либо шифрования и сжатия.

2. Протокол передачи данных пежду процессами

Протокол JSON-RPC является текстовым протоколом, используемым для удалённого вызова процедур, т.е. вызова процедур в другом адресном пространстве. В данном случае выполняется вызов процедур (функций), находящихся в адресном пространстве сопроцесса, из интерпретатора PHP.

Удалённый вызов процедуры происходит следующим образом:

- Вызывающая сторона генерирует запрос, указывающий функцию, которую необходимо вызвать. Также этот запрос содержит аргументы для этой функции, если они имеются.
- Получив запрос, выполняющая сторона (удалённый процесс или компьютер) запускает указанную в запросе функцию и, если требуется, передаёт ей аргументы, содержащиеся в запросе.

- После завершения работы запущеной функции, выполняющая сторона генерирует ответ, содержащий результат выполнения функции или сообщение об ошибке, если она произошла.
- Получив ответ, вызывающая сторона извлекает из него результат выполнения функции или ошибку и продолжает свою работу.

Это можно изобразить, как показано на схеме (рис. 1), где *PHP* — интерпретатор PHP, *coprocess* — сопроцесс, *procedure* — вызываемая процедура, *request* и *response* — вызов удалённой процедуры и ответ на неё, а *arguments* и *return value* — аргументы и возвра-

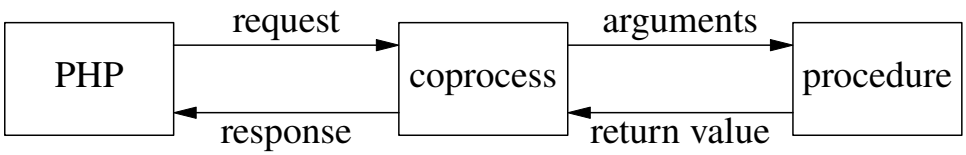

Рис. 1. Схема удалённого вызова процедуры.

щаемое значение процедуры.

В данном протоколе, как ясно из названия, используется формат JSON. Описание этого формата можно найти в [2]. Запрос в JSON-RPC будет выглядеть следующим образом:

```
{
        "jsonrpc": "2.0",
        "id": id,
        "method" : method,
        "params": [param1, param2, ...]
}
```

, где `id` — идентификатор запроса, `method` — строка, содержащая имя вызываемой процедуры, а `param1, param2, ...` — аргументы, передаваемые вызываемой процедуре, если они имеются. Несмотря на то, что спецификация протокола позволяет использовать в качестве аргументов как числа, так и строки, в данном расширении предполагается представлять аргументы исключительно строками. Помимо этого спецификация допускает, что в качестве элемента `params` может не массив, а объект, однако, в данном расширении элемент `params` всегда является массивом.

Ответ, если удалённая процедура была выполнена без ошибок, выглядит так:

```
{
        "jsonrpc": "2.0",
        "id": id,
        "result": result,
}
```

, где `id` — идентификатор запроса, результат вызова которого содержит этот ответ, а `result` — результат вызова. Как и в случае с аргументами, несмотря на то, что спецификация не ограничивает тип этого элемента, в данном расширении предполагается использовать только строки.

Ответ, если при вызове удалённой процедуры произошла ошибка, выглядит так:

```
{
        "jsonrpc": "2.0",
        "id": id,
        "error": {
                "code": errorcode,
                "message": errormessage
        }
}
```

, где `id` — идентификатор запроса, при вызове которого произошла ошибка, `errorcode` — числовой код ошибки, `errormessage` — строка, содержащая текстовое описание ошибки. Источником ошибки может быть либо сам запрос или ответ, либо база данных, либо операционная система. В случае ошибки в запросе спецификацией JSON-RPC предусмотрены следующие коды ошибок:

Код	Сообщение	Значение
-32700	Parse error	Ошибка при разборе JSON-запроса или JSON-ответа.
-32600	Invalid Request	Отправленное JSON-сообщение не является корректным запросом.
-32601	Method not found	Метод не найден или не доступен
-32602	Invalid params	Неверные параметры
-32603	Internal error	Внутренняя ошибка JSON-RPC
-32000 – -32099	Server error	Зарезервированы под определяемые реализацией ошибки сервера.

В случае, когда ошибка произошла при работе с базой данных, возвращается один из кодов ошибок, определенных в [3], в подразделе «ZMessage Codes» раздела «Error Message Quick Reference». Также код ошибки YottaDB можно отличить от других с помощью констант `YDB_MIN_YDBERR` и `YDB_MAX_YDBERR`, определённых в файле заголовков `libyottadb.h` и указывающих нижнюю и верхнюю границы диапазона кодов ошибок, используемых в YottaDB. Любые другие ошибки следует считать ошибками, исходящими от операционной системы.

Также стоит отметить, что спецификация протокола позволяет использовать в качестве идентификаторов запросов как целые числа, так и строки, однако в данном расширении предполагается только использование целых чисел.

3. Обработка запросов и ответов

Для чтения запросов и ответов предполагается использовать рекурсивный парсер. Благодаря простому синтаксису JSON, такой парсер, даже будучи написанным на C, получится достаточно компактным. Парсер будет разбирать запросы и ответы, помещая их в соответствующие им структуры C.

Чтобы упростить разделение получаемых сообщений, а также чтобы снизить количество выполняемых компьютером обращений к памяти и вычислений, предполагается объединить чтение и разбор JSON-сообщения в одну функцию. Текстовые данные будут читаться из неименованного канала «на ходу» разбираясь парсером. Как только будет получен полноценный JSON-объект, функция будет завершаться и возвращать результат.

Формирование и отправку JSON-сообщения, также предполагается объединить в одну функцию. Эта функция будет принимать структуру Си, описывающую запрос или ответ и формировать на её основе JSON-сообщение, сразу записывая это сообщение в неименованный канал.

Ошибки, которые могут происходить при отправке или получении запросов и ответов могут быть связаны либо с передачей данных, либо с протоколом передачи данных. В первом случае коды ошибок определяются стандартной библиотекой Си, а сообщения об ошибках аналогичны сообщениям, получаемым при вызове функции стандартной библиотеки Си `strerror`. Во втором случае коды ошибок определяются перечислением

```
enum YP_JRERROR {
        YP_ERR_JRPARSE = -32700
        YP_ERR_JRINVALREQ = -32600
        YP_ERR_JRINVALRES = -32001
        YP_ERR_JRREQSTRUCT = -32002
        YP_ERR_JRRESSTRUCT = -32003
}
```

и имеют следующие значения:

`YP_ERR_JRPARSE` — получен некорректный JSON-объект.
`YP_ERR_JRINVALREQ` — получен некорректный запрос.
`YP_ERR_JRINVALRES` — получен некорректный ответ.
`YP_ERR_JRREQSTRUCT` — структура, содержащая запрос, некорректна.
`YP_ERR_JRRESSTRUCT` — структура, содержащая ответ, некорректна.

Данные коды ошибок организованны таким образом, чтобы соответствовать кодам ошибок JSON-RPC, не пересекаясь диапазоном значений с кодами ошибок стандартной библиотеки Си и кодами ошибок YottaDB. Текстовое сообщение, соответствующее последней полученной при отправке или получении запросов и ответов ошибки можно получить с помощью функции

```
char *yp_jrstrerror();
```

, возвращающей указатель на строку, содержащую это описание. Эту строку нельзя модифицировать.

Для представления запросов в Си предполагается использовать следующую структуру:

```
struct yp_request {
        int id;
        enum METHOD method;
        int paramcount;
        char **param;
}
```

, где `id` — идентификатор запроса, `method` — значение из перечисления `YP_METHOD`, указывающее, какую функцию необходимо вызвать, `paramcount` — количество аргументов, передаваемых вызываемой функции, а `param` — набор строк, каждая из которых содержит значение одного из этих аргументов. Перечисление `YP_METHOD` предлагается определить следующим образом, при необходимости добавляя в него новые значения:

```
enum YP_METHOD {
        YP_INIT,
        YP_FINILIZE,
        YP_GET,
        YP_SET,
        YP_KILL,
        YP_DATA,
        YP_ORDER
}
```

В данную структуру будут помещаться все полученные и разобранные парсером ответы. Эта же структура будет заполнятся для формирования и отправки JSON-сообщений, содержащих ответы.

Ответы в Си предполагается представлять следующей структурой:

```
struct yp_response {
        int id;
        char *result;
        int errcode;
        char *errmessage;
}
```

, где `id` — идентификатор запроса, результат вызова которого содержит этот ответ, `result` — строка, содержащая результат вызова запроса, `errcode` — код ошибки, `errmessage` — текстовое сообщение об ошибке. Отличить ответ с результатом успешного выполнения функции от сообщения об ошибке можно по элементу `result` — если его значение равно `NULL`, значит ответ является сообщением об ошибке, в любом другом случае выполнение было успешным. Как и в случае с запросами эта структура будет использоваться как для получения ответов, так и для их отправки.

Функция для чтения запросов имеет следующий заголовок:

```
int yp_readrequest(struct yp_request *r);
```

, где `r` — указатель на структуру `yp_request`, куда следует поместить прочитанный запрос. Помимо ошибок стандартной библиотеки Си, данная функция может вернуть ошибки с кодами `YP_JRPARSE` и `YP_ERR_JRINVALREQ`.

Заголовок функции для чтения ответов выглядит так:

```
int yp_readresponse(struct yp_response *r);
```

, где `r` — указатель на структуру `yp_response`, куда будет помещен прочитанный ответ. Помимо ошибок стандартной библиотеки Си, данная функция может вернуть ошибки с кодами `YP_ERR_JRPARSE` и `YP_ERR_JRINVALRES`.

Для генерации и отправки запроса будет использоваться функция с заголовком

```
int yp_writerequest(struct yp_request *r);
```

, где `r` — структура `yp_request`, содержащая запрос, для которого нужно сгенерировать и отправить JSON-сообщение. Помимо ошибок стандартной библиотеки Си, данная функция может вернуть ошибку с кодом `JR_ERR_REQSTRUCT`.

Отправку ответов предполагается осуществлять функцией

```
int yp_writeresponse(struct yp_response *r);
```

, где `r` — структура `yp_response`, содержащая ответ, который нужно сгенерировать и отправить. Помимо ошибок стандартной библиотеки Си, данная функция может

вернуть ошибку с кодом JR_ERR_RESSTRUCT.

4. Работа с запросами в PHP расширении

Как уже было написано выше, большинство функций данного расширения, исполняющихся в интерпретаторе PHP, являются «заглушками», формирующими запрос на основе переданных им аргументов и возвращающие пользователю данные из полученного ответа.

Исключением являются функции *функции-ловушки* (hooks) RINIT и RSHUTDOWN, вызываемые пользователем PHP функции ydb_init и ydb_shutdown, а также функции ydb_error, ydb_strerror и ydb_szinfo.

Функция RINIT используется для инициализации глобальной переменной yp_initilized, указывающей, был ли создан сопроцесс, предназначенный для работы с базой данных. Функция RINIT вызывается без участия пользователя перед началом обработки очередного CGI-запроса (см. [5], подраздел «Learning the PHP lifecycle» раздела «Extensions design») и присваивает глобальной переменной yp_initilized значение 0.

Функция ydb_init в процессе интерпретатора отвечает за инициализацию сопроцесса. Выглядит это так:

1. Если глобальная переменная yp_initilized равна 1, вернуть True.
2. Создать два неименованных канала с помощью системного вызова pipe — один для передачи запросов, другой для передачи ответов. В случае ошибки:
   - Поместить значение глобальной переменной errno в глобальную переменную yp_errcode.
   - Скопировать строку, полученную в результате вызова фукнции strerror в глобальную переменную yp_errmessage.
   - Вернуть False.
3. Запустить сопроцесс с помощью системного вызова fork. В случае ошибки:
   - Поместить значение глобальной переменной errno в глобальную переменную yp_errcode.
   - Скопировать строку, полученную в результате вызова фукнции strerror в глобальную переменную yp_errmessage.
   - Вернуть False.
4. В сопроцессе:
   a. Закрыть конец для записи в неименованном канале, предназначенном для передачи запросов.
   b. Закрыть конец для чтения в неименованном канале, предназначенном для передачи ответов.
   c. Запустить основной цикл сопроцесса (будет описан в разделе 4).
5. В интерпретаторе PHP:
   a. Закрыть конец для чтения в неименованном канале, предназначенном для передачи запросов.
   b. Закрыть конец для записи в неименованном канале, предназначенном для передачи ответов.
6. Сформировать запрос для вызова в сопроцессе функции ydb_init (которая будет описана в разделе 5 данного документа) с аргументами, задающими переменные окружения, которые были переданы пользователем PHP в функцию ydb_init в интерпретаторе.

7. С помощью функции `yp_writerequest` отправить сформированный запрос сопроцессу. Если произошла ошибка:
   - Закрыть неименованные каналы для передачи данных.
   - Получить код завершения сопроцесса с помощью функции `waitpid`.
   - Поместить код возврата функции `yp_writerequest` в глобальную переменную `yp_errcode`.
   - Скопировать строку, полученную в результате вызова функции `yp_jrstrerror` в глобальную переменную `yp_errmessage`.
   - Вернуть `False`.
8. Получить ответ от сопроцесса с помощью функции `yp_readresponse`. Если данная функция завершилась с ошибкой:
   - Закрыть неименованные каналы для передачи данных.
   - Получить код завершения сопроцесса с помощью функции `waitpid`.
   - Поместить код возврата функции `yp_readresponse` в глобальную переменную `yp_errcode`.
   - Скопировать строку, полученную в результате вызова функции `yp_jrstrerror` в глобальную переменную `yp_errmessage`.
   - Вернуть `False`.
9. Если поле `result`, полученного ответа равно `NULL`:
   - Закрыть неименованные каналы для передачи данных.
   - Получить код завершения сопроцесса с помощью функции `waitpid`.
   - Поместить код возврата функции `yp_writerequest` в глобальную переменную `yp_errcode`.
   - Скопировать строку, полученную в результате вызова функции `yp_jrstrerror` в глобальную переменную `yp_errmessage`.
   - Вернуть `False`.
10 Присвоить переменной `yp_initilized` значение `1`.
11 Вернуть `True`.

Функция `RSHUTDOWN` в процессе интерпретатора выполняет действия, необходимые для завершения работы сопроцесса и вызывается после окончания работы с очередным CGI-запросом (см. [5], подраздел «Learning the PHP lifecycle» раздела «Extensions design»). Выполняемые ей действия можно записать так:

1. Если глобальная переменная `yp_initilized` равна `0`, вернуть `SUCCESS`.
2. Присвоить локальной переменной `iserror` значение `0`.
3. С помощью функции `yp_writerequest` отправить запрос для вызова в сопроцессе функции `ydb_finilize` (будет описана в разделе 4 данного документа). Если произошла ошибка:
   - Используя функцию `php_err_docref`, вывести предупреждение, содержащее строку, полученную в результате вызова функции `yp_jrstrerror`.
   - Присвоить локальной переменной `iserror` значение `1`.
   - Перейти к пункту 5.
4. Получить ответ от сопроцесса с помощью функции `yp_readresponse`. Если при получении ответа произошла ошибка:
   - Используя функцию `php_err_docref`, вывести предупреждение, содержащее строку, полученную в результате вызова функции `yp_jrstrerror`.
   - Присвоить локальной переменной `iserror` значение `1`.
   - Перейти к пункту 5.
5. Если поле `result` полученного ответа равно `NULL`:

- Вывести содержимое поля `errmessage` с помощью фукнции `php_error_docref`.
- Присвоить локальной переменной `iserror` значение `1`.

6. Закрыть неименованные каналы для передачи данных.
7. Получить код завершения сопроцесса с помощью функции `wait`. Если код не равен нулю или его не удалось получить:
   - Вывести предупреждение `"Coprocess ended abnormaby."` с помощью функции `php_error_docref`.
   - Присвоить локальной переменной `iserror` значение `1`.
8. Присвоить глобальной переменной `ydb_initilized` значение `0`.
9. Если значение переменной `iserror` равно `0`, вернуть `SUCCESS`, иначе вернуть `FAILURE`.

Функция `ydb_finilize` предназначена для завершения сопроцесса пользователем PHP «вручную» и выполняет действия аналогичные тем, что выполняются функцией `RSHUTDOWN`, от которой она отличается только обработкой ошибок. В случае ошибки данная функция возвращает `False`, при этом помещая код ошибки в глобальную переменную `yp_errcode`, а сообщение об ошибке — в глобальную переменную `yp_errmessage`. При успешном выполнении возвращается `True`.

Функции `ydb_set`, `ydb_get`, `ydb_gettyped`, `ydb_kill`, `ydb_data` и `ydb_order`, как уже упоминалось, формируют запрос, содержащий переданные им аргументы, отправляют его, дожидаются ответа и возвращают результат выполнения функции в сопроцессе, содержащийся в этом ответе.

Первые аргументы приведённых выше функций — `var` и произвольное число аргументов `keys` (см. описание программного интерфейса) обрабатываются одинаковым образом. Эти аргументы преобразуются в одну цельную строку, всегда помещаемую в запрос в качестве первого аргумента и указывающую переменную или элемент массива, над которым нужно выполнить действие. Данное преобразование выглядит так:

1. Инициализировать строку `s` значением переданного пользователем PHP аргумента `var`.
2. Если пользователем PHP был передан хотя бы один аргумент `keys`, добавить к строке `s` символ «`(`».
3. Для каждого переданного пользователем PHP аргумента `keys`, с первого такого аргумента по последний:
   a. Если этот аргумент является строкой, присоединить его к `s`, дополнив символом «`,`».
   b. Если этот аргумент является числом, преобразовать его в строку и присоединить к `s`, дополнив символом «`,`».
   c. Если этот аргумент является массивом PHP с целочисленными ключами, то для каждого его элемента массива, с первого по последний:
      •. Если данный элемент является строкой, присоединить его к `s`, дополнив символом «`,`».
      •. Если данный элемент является числом, приобразовать его в строку и присоединить к `s`, дополнив символом «`,`».
   d. Если аргумент имеет любой другой тип:
      - Присвоить глобальной переменной `ydb_errcode` значение `YDB_ERR_WRONGARGS`.
      - Присвоить глобальной переменной `ydb_errmessage` значение `"Argument [argnum] has wrong type."`, где `[argnum]` — где номер этого аргумента.

4. Если пользователем PHP был передан хотя бы один аргумент `keys`, заменить символ «`,`», находящийся в конце строки `s` на символ «`)`».

Остальные действия для данных функций, кроме `ydb_gettyped` практически идентичны:

1. Если глобальная переменная `yp_initilized` равна `0`:
   - Присвоить глобальной переменной `ydb_errcode` значение `YDB_ERR_NOTINITILIZED`.
   - Присвоить глобальной переменной `ydb_errmessage` значение `"Coprocess wasn't initilized. You should run ydb_init() first."`.
   - Вернуть `false`.
1. Объединить переданные пользователем арументы `var` и произвольное число аргументов `keys` в строку `s`, как было описано выше.
2. Сформировать запрос вызывающий в сопроцессе функцию с аналогичным именем и содержащий строку `s` в качестве первого аргумента. Для функций `ydb_set` и `ydb_order` запрос помимо этого в качестве второго аргумента, должен содержать переданные пользователем PHP аргументы `val` и `dir` соответственно.
3. С помощью функции `yp_writerequest` отправить сформированный запрос сопроцессу. Если произошла ошибка:
   - Поместить код возврата функции `yp_writerequest` в глобальную переменную `yp_errcode`.
   - Скопировать строку, полученную в результате вызова функции `yp_jrstrerror` в глобальную переменную `yp_errmessage`.
   - Вернуть `False`.
4. Получить ответ от сопроцесса с помощью функции `yp_readresponse`. Если при получении ответа произошла ошибка:
   - Поместить код возврата функции `yp_readresponse` в глобальную переменную `yp_errcode`.
   - Скопировать строку, полученную в результате вызова функции `yp_jrstrerror` в глобальную переменную `yp_errmessage`.
   - Вернуть `False`.
5. Для функций `ydb_get`, `ydb_data` и `ydb_order`, если поле `result` не равно `NULL`, вернуть содержащееся в нём значение пользователю PHP в виде строки. Для функции `ydb_kill` в этом случае вернуть `True`. Если же поле `result` равно `NULL`:
   - Поместить содержимое поля `errcode` в глобальную переменную `yp_errcode`.
   - Поместить содержимое поля `errmessage` в глобальную переменную `yp_errmessage`.
   - Вернуть `False`.
6. Вернуть `True`.

Функция `ydb_gettyped` отличается от остальных. Во-первых, вместо одноимённой себе функции она вызывает функцию `ydb_get`. Во-вторых, после получения ответа выполняется еще одно действие — определение типа содержащегося в поле `result` значения. Это делается с помощью следующей функции интерпретатора PHP:

```
zend_uchar is_numeric_string(const char *str, size_t length,
        zend_long *lval, double *dval, int allow_errors);
```

, где `str` — строка, содержащая значение, `length` — длина этой строки, `lval` —

значение, возвращаемое в случае, если строка `str` содержит целое число, `dval` — значение, возвращаемое в случае, если строка `str` содержит десятичную дробь, а `allow_error` — указывает, считать строку, где число содержится только в её начале численной или нет (1 — считать, 0 — не считать). Данная функция возвращает значение `IS_LONG`, если `str` содержит целое число или `IS_DOUBLE`, если `str` содержит десятичную дробь. Любое другое возвращаемое значение указывает, что `str` не содержит числа и может быть представлено в PHP только строкой.

Полностью алгоритм определения типа возвращаемого значения выглядит так:

1. Если элемент `result` ответа равен `"true"` (независимо от регистра), вернуть `True`.
2. Если элемент `result` ответа равен `"false"` (независимо от регистра), вернуть `False`.
3. Вызвать `is_numeric_string`.
4. Если возвращаемое значение функции `is_numeric_string` равно `IS_LONG`, вернуть пользователю PHP целое число, полученное через её третий аргумент.
5. Если возвращаемое значение функции `is_numeric_string` равно `IS_DOUBLE`, вернуть пользователю PHP число с плавающей точкой, полученное через её четвёртый аргумент аргумент.
6. Если возвращаемое значение функции `is_numeric_string` не равно `IS_DOUBLE` или `IS_LONG`, вернуть его как строку.

Функции `ydb_error`, `ydb_strerror` и `ydb_zsinfo`, возвращают полученную ранее информацию об ошибках. Работа функций `ydb_error` и `ydb_strerror` сводится к возврату пользователю PHP значений глобальных переменных `yp_errcode` и `yp_errmessage` соответственно. Переменная `yp_errcode` возвращается как целое число, а `yp_errmessage` — как строка.

Функция `ydb_szinfo`, предназначенная для получения информации об ошибке YottaDB, как и `ydb_strerror` работает с глобальной переменной `yp_errmessage`, однако она не просто возвращает её значение, а разбирает его, предполагая синтаксис, описанный в [4], в разделе «Intrinsic Special Variables», в части, посвященной описанию специальной внутренней переменной YottaDB `$ZSTATUS`, помещая искомые элементы сообщения в соответствующие им возвращаемые значения. В случае, если значение переменной `yp_errmessage` разобрать не удаётся, функция возвращает `False`, иначе возвращает `True`.

Описанные в данном разделе функции позволяют использовать данное расширение как с интерпретатором в режиме обычного CGI, так и с интерпретатором в режиме *FastCGI*. Однако в случае работы в режиме FastCGI имеется ограничение — интерпретотор должен обрабатывать CGI-запросы последовательно, не используя *нити* (threads). Это связано с тем с активным использованием глобальных переменных, которые в случае разделения процесса на нити, могут одновременно читаться и изменяться несколькими нитями, что рано или поздно вызовет ошибку.

В приложении 1 приведен пример обработки запроса при работе интерпретотора в режиме FastCGI. В случае, если интерпретатор работает в режиме обычно CGI, обработка запроса будет выглядить аналогично, за исключением того, что выполнение кода в функции `MSHUTDOWN` будет избыточным.

5. Сопроцесс для работы с базой данных

Выше, при описании работы функции ydb_init был упомянут основной цикл сопроцесса, запускаемый после настройки неименованных каналов для передачи запросов и ответов. По сути, в данный цикл и выполняет всю работу в сопроцессе. Записать его можно так:

Выполнять в бесконечном цикле:

1. С помощью функции yp_readrequest получить запрос из интерпретатора PHP. Если при получении запроса произошла ошибка:
   - Сгенерировать ответ, где поле errcode содержит код возврата функции yp_readrequest, а поле errmessage содержит строку, полученную в результате вызова yp_jrstrerror.
   - Попытаться отправить этот ответ.
   - Завершить сопроцесс с ненулевым кодом возврата помощью функции exit.
2. Запустить в данном сопроцессе функцию, соответствующую значению, содержащемуся в поле method структуры yp_request, передав ей в качестве параметров аргументы, содераящиеся в поле param этой структуры. Все запускаемые таким образом функции имеют один интерфейс:

   ```
   void func(const char *arg1, const char *arg2, ...,
          struct yp_result *r);
   ```

   , где func — имя запускаемой функции, arg1, arg2, ... — аргументы, содержащиеся в поле param структуры yp_request, а r — структура yp_result, куда помещается результат выполнения фукции.
3. С помощью функции yp_writeresponse отправить пользователю ответ, содержащийся в структуре result, которая была заполнена вызванной функцией. Если при отправке ответа произошла ошибка, завершить сопроцесс с ненулевым кодом возврата с помощью функции exit.
4. Если только что вызванная функция — это ydb_finilize, завершить сопроцесс с нулевым кодом возврата с помощью функции exit.

Все функции, за исключением ydb_init и ydb_finilize, упомянутые в пункте 2 приведённого выше цикла, являются такими же «точками входа», «заглушками», как и функции, вызывающие их из интерпретатора PHP. Однако в отличие от последних, они вызывают процедуры в адресном пространстве своего же процесса. Вызываемые процедуры являются являются процедурами MUMPS, выполняющими всю работу с базой данных.

Вызов процедур MUMPS выполняется с помощью специального интерфейса, предназначенного для вызовов YottaDB из Си и предоставляемого разделяемой библиотекой libyottadb.so (см. [4], раздел «Integrating External Routines). Данный интерфейс определяет несколько функций, из которых предполагается использовать только две — ydb_ci, для запуска процедур YottaDB, и ydb_zstatus, для получения сообщений об ошибках.

Функция ydb_ci использует для своей работы два файла — первый файл содержит вызываемые процедуры MUMPS, а второй файл представляет собой *таблицу вызовов* (*call-in table*), позволяющую связать процедуры MUMPS с языком Си. Файл с процедурами MUMPS, используемыми в данном расширении приведён в приложении 2 (ydbphp.m), а соответствующая этим процедурам таблица вызовов приведена в приложении 3 (ydbphp.ci).

Функция ydb_zstatus, как уже упоминалось, используется для получения сообщения об ошибке, если она произошла. Эта функция записывает сообщение, содержащееся в специальной внутренней переменной YottaDB $ZSTATUS ([4], раздел «Intrinsic Special Variables») в переданную ей в качестве аргумента строку. Данное сообщение в

дальнейшем помещается в поле `errmessage` ответа, отправляемого обратно интерпретатору PHP.

Как упоминалось выше, функции `ydb_init` и `ydb_finilize` — единственные вызываемые в сопроцессе функции, не являющиеся «точками входа» для процедур MUMPS. Задача этих функций — инициализация YottaDB в сопроцессе и завершение работы сопроцесса.

Так как в YottaDB все первоначальные настройки выполняются с помощью переменных окружения, то и работа `ydb_init` сводится к настройке окружения сопроцесса. Выглядит это так:

1. Задать переменную окружения `gtmroutines` таким образом, чтобы она содержала путь к директории, где лежит файл `ydbphp.m`.
2. Задать переменную окружения `GTMCI` таким образом, чтобы она содержала путь к файлу `ydbphp.ci`.
3. Для каждого переданного в запросе аргумента:
   a. Если данный аргумент задаёт переменную окружения `gtmroutines`, добавить пути, указанные в аргументе к тому пути, что уже задан в данной переменной окружения.
   b. Если данный аргумент задаёт переменную окружения `GTMCI`:
      - Поместить в поле `errcode` ответа значение `YP_ERR_ENV`
      - Поместить в поле `errmesssage` ответа значение `"Enviroment variable GTMCI cannot be set by user"`.
   c. Если данный аргумент задаёт любую другую переменную окружения, предназначенную для настройки YottaDB, задать переменной окружения указанное значение.
   d. Если данный аргумент задаёт переменную окружения, не имеющую отношения к настройке YottaDB:
      - Поместить в поле `errcode` ответа значение `YP_ERR_ENV`
      - Поместить в поле `errmesssage` ответа значение `"Enviroment variable [envname] is not used by YottaDB"`, где `[envname]` — имя задаваемой данным аргументом переменной окружения.

Функция `ydb_finilize` не выполняет никаких действий ни с базой данных, ни с сопроцессом. Вызов данной функции является сигналом для сопроцесса, указывающим ему, что необходимо завершить свою работу. Данный сигнал, как было показано выше, обрабатывается основным циклом сопроцесса, поэтому все, что делает функция `ydb_finilize` — это задает поле `result` ответа, помещая в него пустую строку.

Остальные функции выглядят одинаково и запускают соответствующую им процедуру MUMPS с помощью `ydb_ci`:

> С помощью `ydb_ci` запустить процедуру MUMPS `[funcname]`, передав ей имеющиеся в запросе аргументы, где `[funcname]` — имя данной функции. Если `ydb_ci` вернула отличный от нуля код:
> - Поместить код возврата `ydb_ci` в поле `errcode` ответа.
> - Поместить сообщение об ошибке, полученное с помощью вызова функции `ydb_zstatus` в поле `errmessage` ответа.

Приложение 1. Пример обработки FastCGI-запроса.

В этом приложении приведён пример обработки на FastCGI-запроса в следующем PHP-сценарии:

```
$env = array(
        "gtm_dist=/usr/local/lib/yottadb/r128",
        "gtmgbldir=/home/evgeniy/.yottadb/g/yottadb.gld",
        "ydb_dir=/home/qwerty/.yottadb",
        "ydb_rel=r1.28_x86_64");

if (!ydb_init($env)) {
        error_log("Error occured: " . ydb_strerror(), 0);
        return 1;
}

if (!ydb_set("^person", "Komi Republic", "Syktyvkar",
        1, "name", "Ivan")) {
        error_log("Error occured: " . ydb_strerror(), 0);
        return 1;
}

if ((s = ydb_get("^person", "Komi Republic", "Syktyvkar",
        1, "name")) === FALSE) {
        error_log("Error occured: " . ydb_strerror(), 0);
        return 1;
}

if ((s = ydb_get("^person", "Komi Republic", "Syktyvkar",
        1, "surname", "Ivan")) === FALSE) {
        error_log("Error occured: " . ydb_strerror(), 0);
        return 1;
}

return 0;
```

Сначала, при получении CGI-запроса, запускается функция `RINIT`, устанавливающая глобальную переменную `yp_initilized` в `0`.

После этого заполняется массив `env`, который далее передаётся функции `ydb_init` в качестве аргумента. Данная функция создаёт неименованные каналы и выполняет запуск сопроцесса, а затем формирует запрос для вызова функции `ydb_init` в сопроцессе. Для этого сначала заполняется структура `yp_request`:

id: 1
method: YDB_INIT
paramcount: 4
param[0]: "gtm_dist=/usr/local/lib/yottadb/r128"
param[1]: "gtmgbldir=/home/qwerty/.yottadb/g/yottadb.gld"
param[2]: "ydb_dir=/home/qwerty/.yottadb"
param[3]: "ydb_rel=r1.28_x86_64"

Далее на основе этой структуры формируется JSON-запрос

```
{
        "jsonrpc" : "2.0",
        "id" : 1,
        "method" : "YDB_INIT",
        "params" : [
                "gtm_dist=/usr/local/lib/yottadb/r128",
                "gtmgbldir=/home/qwerty/.yottadb/g/yottadb.gld",
                "ydb_dir=/home/qwerty/.yottadb",
                "ydb_rel=r1.28_x86_64"
        ]
}
```

, который отправляется сопроцессу. От сопроцесса приходит JSON-ответ

```
{
        "jsonrpc" : "2.0",
        "id" : 1,
        "result" : ""
}
```

, помещаемый в структуру

id: 1
result: ""
errcode: undefined
errmessage: undefined

После получения ответа функция `ydb_init` возвращает `True` и выполнение переходит к следующим командам.

При вызове `ydb_set` из переданных пользователем аргументов формируется структура `yp_request` следующего вида:

id: 2
method: YDB_SET
paramcount: 2
param[0]: "^person(\"Komi Republic\",\"Syktyvkar\",1,\"Name\")"
param[1]: "Ivan"

На основе этой структуры формируется JSON-запрос

```
{
        "jsonrpc" : "2.0",
        "id" : 2,
        "method" : "YDB_SET",
        "params" : [
                "^person("Komi Republic","Syktyvkar",1,"Name")",
                "Ivan"
        ]
}
```

, который отправляется сопроцессу. JSON-ответ, полученный от сопроцесса выглядит

следующим образом:

```
{
        "jsonrpc" : "2.0",
        "id" : 2,
        "result" : ""
}
```

Этот ответ помещается в структуру `yp_response`:

id: 2
result: ""
errcode: undefined
errmessage: undefined

Далее, так как поле `result` этой структуры содержит пустую строку, функция `ydb_set` возвращает `True`.

Следующей вызывается функция `ydb_get`, где сформированная из переданных пользователем аргументов структура `yp_request` выглядит следующим образом:

id: 3
method: YDB_GET
paramcount: 1
param[0]: "^person(\"Komi Republic\",\"Syktyvkar\",1,\"Name\")"

На её основе формируется JSON-запрос

```
{
        "jsonrpc" : "2.0",
        "id" : 3,
        "method" : "YDB_GET",
        "params" : [
                "^person("Komi Republic","Syktyvkar",1,"Name")"
        ]
}
```

, отправляемый сопроцессу. От сопроцесса приходит JSON-ответ

```
{
        "jsonrpc" : "2.0",
        "id" : 3,
        "result" : "Ivan"
}
```

, помещаемый в структуру `yp_response`:

id: 3
result: "Ivan"
errcode: undefined
errmessage: undefined

После этого, функция `ydb_get` завершается и возвращает пользователю значение поля `result` этой струкуры — строку `"Ivan"`.

Далее функция `ydb_get` вызывается второй раз, но уже с другими аргументами, что провоцирует ошибку, связанную с отсутсвием искомого элемента в массиве MUMPS. В результате этой ошибки, от сопроцесса приходит следующий JSON-ответ (переносы строки в элементе `"message"` добавлены для удобства чтения и в реальном ответе отсутствуют):

```
{
        "jsonrpc" : "2.0",
        "id" : 2,
        "error" : {
                "code" : 150372994,
                "message" : "150372994,ydbget+1^ydbphp,%YDB-E-GVUNDEF,
Global variable undefined:
^person(\"Komi Republic\",\"Syktyvkar\",1,\"surname\")"
        }
}
```

Этот ответ помещается в структуру `yp_response`:

id: 4
result: NULL
errcode: 150372994
errmessage: "150372994,ydbget+1^ydbphp,%YDB-E-GVUNDEF, Global variable undefined: ^person(\"Komi Republic\",\"Syktyvkar\",1,\"surname\")"

Из-за того, что была получена ошибка, значения полей `errcode` и `errmessage` помещаются в глобальные переменные `yp_errcode` и `yp_errmessage`, а затем функция `ydb_get` завершается, вернув значение `False`.

Следующей вызывается функция `ydb_strerror`. Эта функция возвращает значение переменной `yp_errmessage` в виде строки PHP. Полученная строка отправляется в журнал ошибок, определенный конфигурацией интерпретатора PHP, а затем обработка CGI-запроса завершается с кодом `1`.

После завершения работы с CGI-запросом, запускается функция `RSHUTDOWN`. Сначала данной функцией формируется структура `yp_request`:

id: 5
method: YDB_FINILIZE
paramcount: 0
param: NULL

, которая преобразуется в JSON-запрос

```
{
        "jsonrpc" : "2.0",
        "id" : 5,
        "method" : "YDB_FINILIZE",
}
```

, отправляемый сопроцессу. Ответ сопроцесса выглядит следующим образом:

```
{
        "jsonrpc" : "2.0",
        "id" : 5,
        "result" : ""
}
```

Этот ответ помещается в структуру `yp_response`:

id: 5
result: ""
errcode: undefined
errmessage: undefined

Далее закрываются неименованные каналы для передачи запросов и ответов, а затем, с помощью вызова функции `wait` забирается код возврата сопроцесса. Так как ошибок при завершении сопроцесса не произошло, функция `MSHUTDOWN` возвращает значение `SUCCESS` и интерпретатор PHP переходит к обработке следующего CGI-запроса.

Приложение 2. ydbphp.m

```
ydbget(var)
        quit @var
ydbset(var, val)
        set @var=val
        quit
ydbkill(var)
        kill @var
        quit
ydbdata(var)
        quit $data(@var)
ydborder(var)
        quit $order(@var)
```

Приложение 3. ydbphp.ci

```
ydb_get : gtm_char_t *ydbget^ydbphp(I:gtm_char_t *)
ydb_set : void ydbset^ydbphp(I:gtm_char_t *)
ydb_kill : void ydbkill^ydbphp(I:gtm_char_t *)
ydb_data : gtm_char_t *ydbdata^ydbphp(I:gtm_char_t *)
ydb_order : gtm_char_t *ydborder^ydbphp(I:gtm_char_t *)
```

Ссылки

1. `https://www.jsonrpc.org/specification` — JSON-RPC 2.0 Specification.
2. `https://www.json.org/json-ru.html` — JSON
3. `https://docs.yottadb.com/MessageRecovery` — Messages and Recovery Procedures documentation
4. `https://docs.yottadb.com/ProgrammersGuide` — Programmer's Guide documentation
5. `http://www.phpinternalsbook.com` — PHP Internals Book

Соображения

- Объединение трех разных диапазонов кодов ошибок в один, наверное не очень хорошая идея. Может быть, стоит сделать как-то по другому.
- Неименованные каналы не единственный вариант сообщения процессов. Возможно стоит рассмотреть другие варианты.
- Функция `ydb_gettyped` использует плохо документированную функцию PHP. Но другой пока я не нашел.
- Возможно стоит добавить для функции `ydb_gettyped` возможность выгружать из базы узел вместо с дочерними узлами, помещаяя все в ассоциативный массив. JSON как раз позволит это реализовать. Для значения самое узла можно выделить какое-нибудь специальное значение.
- Я не стал подробно описывать JSON-парсер. Возможно стоит это сделать в отдельном документе. У меня есть прототип на `AWK` от другой задачи, возможно это хорошая стартовая точка.
  Также нет описания разбора перменной `$ZSTATUS`. Однако, я думаю, тут можно обойтись без полноценного парсера.
- Я возлагаю почти всю обработку ошибок на саму базу данных. Надо будет убедится, что она выдаёт все необходимые ошибки. Пока мне кажется, что всё слишком красиво. Над кодом, выполняемым в MUMPS надо подумать.
- Скорее всего еще понадобится реализация блокировок. Но она, мне кажется, не будет отличаться от других функций.
- В данный момент при использовании FastCGI, сопроцесс создаётся при обработке каждого нового запроса. Возможно стоит рассмотреть вариант, когда сопроцесс создаётся единожды, при запуске интерпретора. Однако в этом случае придётся постоянно проверять и поддерживать целостность окружения базы данных в сопроцессе, но возможно, это будет расходовать еще больше ресурсов, чем удаление и создание нового процесса.
- Теоретически, может получиться процессс-зомби, но в случае проблем с передачей данных сопроцесс должен немедленно завершиться. Надо получше изучить этот момент.

Изменения

Раздел 3:

- В структуру `yp_request` добавлено поле `paramcount`.

Раздел 4:

- Добавление описание функции `RINIT`.
- Немного переделана функция `ydb_init` и функции для работы с базой данных.
- Код функции `ydb_finilize` теперь вызывается в функции `RSHUTDOWN`.

Добавлено приложение с примером обработки FastCGI-запроса.