Perl beágyazása MySQL rendszerbe

Készítsünk a MySQL-hez saját szolgáltatásokat a MyPerl segítségével, amelyek a hatékony és könnyen változtatható Perl-értelmező erejét hozzák el a relációs adatbázisok szívébe.

A MySQL igen gazdag függvénykészlettel rendelkezik, mégis könnyen előfordulhat, hogy egyedi igényeink támadnak, esetleg fejlettebb szabályoskifejezés-kezelő motorra van szükségünk. Az ilyen gondok megoldására találták ki MySQL alatt a felhasználói függvényeket (user-defined functions, UDF). Az UDF csatolófelületen keresztül dinamikusan új függvényeket tölthetünk be az adatbázisba. Annak ellenére, hogy igen hatékony módszerről van szó, most nem kell C- vagy C++-kódok hibáinak a javításával bajlódnunk. Bármennyire is kedvelem a C nyelvet, néha egyszerűen nincs időm az alkalmazások elkészítésére és a hibáik kijavítgatására, máskor pedig egyszerűen csak gyorsabb fejlesztési ciklust szeretnék. És itt lép be képbe a Perl, a nyelvek mindenese – ezer köszönet a Comprehensive Perl Archive Network (CPAN) hálózatának! A Perl MySQL környezetbe illesztése elképesztő rugalmasságot nyújtott adatbázisom gyors bővítéséhez. A fentieknek köszönhetően jött létre a MySQL Perl-értelmezője, a MyPerl.

A Perl beállítása

Az első lépés, amit az adatbázisba illesztés előtt meg kell tennünk, a Perl helyes beállítása lesz. A Perl ugyanis alapértelmezés szerint a szálakat (threads) nem kezeli, a MySQL viszont minden felhasználói kapcsolathoz külön szálat használ. Így ha az adatbázisunk belsejében Perlt szeretnénk üzemeltetni, kénytelenek leszünk Perl-száltámogatással lefordítani:

```
./Configure -Dusethreads -Duseithreads
```

Amint ezzel elkészülünk, máris egy száltámogatással rendelkező Perl-rendszert birtoklunk. Ez nem azt jelenti, hogy a kódunk vagy az abban felhasznált Perl-modulok szálbiztosak lesznek, mindössze annyit tudhatunk, hogy maga a Perl az lesz. A szálbiztos Perl fordítása elengedhetetlen, mivel egyetlen terjesztőt sem ismerek, aki szálbiztos Perl-változatot alkalmazna. Ne tévesszen meg bennünket, hogy a MyPerl nem szálbiztos Perl alatt is hajlandó lefordulni; lefordul ugyan, de a későbbiek során egyszer csak kifagyasztja az adatbázist. Gyanítom, hogy az Apache 2.0 és a `mod_perl2` eljövetelével néhány terjesztő már számításba fogja venni a szálak engedélyezésével fordított Perl-binárisok forgalmazását. Mostanában, miközben a cikk befejezéséhez készülődtem, Red Hat 9-re frissítettem, és láttam, hogy ezt a változatot már szálkezelést támogató Perllel szállították.

Beágyazott Perl-értelmező készítése

Az UDF-eknek három állapota van: `init` (inicializálás), `request` (kérelem) és `deinit` (deinicializálás). Az `init` lépcsőfok a lekérdezés elején, egyszer hívódik meg; a `request` állapot minden egyes soron végrehajtódik, végül a `deinit` állapot az adat ügyfélhez küldése után hívódik meg. Az `init` és `deinit` állapotokat átugorhatjuk, igaz, a legegyszerűbb UDF-ektől eltekintve szinte mindig szükségünk lesz memóriafoglalásra és- felszabadításra, hiszen így tudunk az ügyfélnek adatokat átadni.
A MyPerl a következő `init` függvénnyel kezdődik:

```
my_bool
myperl_init(UDF_INIT *initid, UDF_ARGS *args,
            ➥char *message)
{
     myperl_passable *pass = NULL;
     int exitstatus = 0;
     char *embedding[] = { "perl",
                           ➥"-MMyPerl::Request",
                           ➥"-e", "1" };
     PerlInterpreter *my_perl;
     uint i =0;
     initid->max_length = 256;
     initid->maybe_null=1;
```

Az `init` függvény három értéket kap meg, és siker vagy kudarc értékkel tér vissza. Az `UDF_INIT` szerkezet az UDF viselkedésével kapcsolatos értékeket tárolja, egyúttal ez az egyetlen szerkezet, amely a három állapot közt átadódik. Először is közöljük a MySQL-lel, hogy az UDF a `VARCHAR` méretnél nagyobb adatot fog visszaadni. A kiszolgáló figyelmét felhívjuk, hogy `VARCHAR` méretnél nagyobbra kell számítania, így azt feltételezi, hogy `blog` típust kap vissza. Bár a MyPerl nem tudja, hogy valóban ez fog-e történni, ezen a ponton semmilyen módon nem tudhatjuk meg, hogy mennyi adatot akar majd visszaadni, így a legbiztosabb, ha a kiszolgálónak `blog` típust jelentünk. Ezután a `maybe_null` értéket 1-re állítjuk, hiszen mindig fennáll a lehetőség, hogy

NULL értékeket kell visszaadnunk. A MyPerl egyaránt NULL értéket ad vissza, ha az eredmény üres, illetve, ha az eval() függvénnyel futtatott kód fordítási hibát jelez.
A következő feladat az átadandó sorok vizsgálata:

```
if (args->arg_count == 0 || i
     args->arg_type[0] != STRING_RESULT) {
     strncpy(message,USAGE, MYSQL_ERRMSG_SIZE);
     return 1;
}
for (i=0 ; i < args->arg_count; i++)
          args->arg_type[i]=STRING_RESULT;
```

A MyPerl elsőként a futtatandó kódot várja. Ezért aztán, ha egyetlen sort sem adunk át vagy ha az első sor nem string típusú, hibajelzést kapunk. A hibaüzenetek mérete legfeljebb MYSQL_ERRMSG_SIZE méretű lehet, amelyeket a message string változóba kell másolnunk. Időmegtakarításképpen a MyPerl végignézi az átadásra kerülő értékeket, és a MySQL kéri, hogy alakítsa őket karaktersorozattá.
A Perl-értelmező indítása előtt létre kell hoznunk egy szerkezetet, amelyben az értelmezőt tárolni fogjuk, illetve nyomon kell követnünk azt a memóriablokkot, ahol az egyes kérelmek visszatérő adatai kerülnek:

```
pass = (myperl_passable *)
     malloc(sizeof(myperl_passable));;
if (!pass) {
     strncpy(message, "Could not allocate memory",
             ➥MYSQL_ERRMSG_SIZE);
     return 1;
}
```

A szerkezet megadása:

```
typedef struct {
     char *returnable;
     size_t  size;
     PerlInterpreter *myperl;
     size_t  messagesize;
} myperl_passable;
```

A returnable karaktermutatót használjuk a memóriablokk címének tárolására, a size és messagesize változókat pedig a visszatérítendő adat jelenlegi és teljes méretének nyilvántartására. Minthogy a memória foglalása és felszabadítása igen erőforrás-igényes folyamat, nagyon fontos, hogy a lehető legkevesebbszer éljünk vele. A Perl-értelmezőt szintén ebben a szerkezetben tároljuk.
A lekérdezéshez használható Perl-értelmező beállításával lényegében el is készültünk. Jelenleg a MyPerl minden egyes lekérdezéshez külön Perl-értelmezőt készít, hogy elkerülje a memóriaszivárgást és ne veszélyeztesse a lekérdezések között az adatok biztonságát. Nagy valószínűséggel a jövőben inkább Perl-értelmezők egy csoportját fogjuk alkalmazni:

```
if((my_perl = perl_alloc()) == NULL) {
     strncpy(message, "Could not allocate perl",
             ➥MYSQL_ERRMSG_SIZE);
     return 1;
}
perl_construct(my_perl);
exitstatus = perl_parse(my_perl, xs_init, 4,
                        ➥embedding, environ);
PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
if (exitstatus) {
     strncpy(message, "Error in creating perl
             ➥parser",
              MYSQL_ERRMSG_SIZE);
     goto error;
}
exitstatus = perl_run(my_perl);
if (exitstatus) {
     strncpy(message, "Error in parsing your perl",
             ➥MYSQL_ERRMSG_SIZE);
     goto error;
}
```

Az első függvény, a perl_alloc(), új Perl-értelmezőt foglal le, amelyet aztán a perl_construct() hoz létre. Innentől már mindössze annyi dolgunk maradt, hogy lefuttassuk a Perlt. A beágyazott változót értékként használhatjuk a Perl-értelmezőhöz – ezek pontosan ugyanazok az értékek lesznek, amelyeket a parancssorban is alkalmaznánk. Valahányszor a Perl-értelmezőhöz nyúlunk, ellenőriznünk kell a hibákat. A jelenlegi megoldásban hiba esetén a MyPerl olyan függvényekre ugrik, amelyek felszabadítják a lefoglalt memóriát.
Sikerült előállítanunk egy használható Perl-értelmezőt, ideje, hogy az átadási szerkezetben is beállítsunk néhány alapértelmezett értéket. Az értelmezőt be kell állítanunk, a szerkezet

címét pedig az `initid->ptr` mutatóban kell tárolnunk, hogy aztán a teljes lekérdezés ideje alatt használható legyen:

```
pass->returnable = NULL;
pass->size = 0;
pass->messagesize = 0;
pass->myperl = my_perl;
initid->ptr = (char*)pass;
return 0;
```

A fenti beállítások után a MyPerl már készen áll a lekérdezések fogadására:

```
char *
myperl(UDF_INIT *initid, UDF_ARGS *args,
       ➥char *result, unsigned long *length,
       ➥char *is_null, char *error)
{
    myperl_passable *pass =
        (myperl_passable *)initid->ptr ;
    char *returnable = NULL;
    unsigned long x = 0;
    size_t size = 0;
    char *newspot = NULL;
    char *string = NULL;
    myperl_passable *pass =
         ➥(myperl_passable *)initid->ptr ;
    STRLEN n_a;
    PerlInterpreter *my_perl = pass->myperl;
```

Nagyon fontos, hogy az értelmező nevét a `my_perl` nevű változóba másoltuk. A Perl belső részek nagy többsége olyan makrókon alapul, amelyek elvárják, hogy változókat bizonyos néven nevezzünk. Az `STRLEN` változótípust a karaktersorozatok méretének tárolására használja. Meghívjuk az értelmezőt:

```
dSP;
ENTER;
SAVETMPS;
PUSHMARK(SP);
//a maradék értéket átadjuk az ARGV-nek
for(x = 0; x < args->arg_count  ; x++) {
    XPUSHs(sv_2mortal(newSVpvn(args->args[x],
          ➥args->lengths[x])));
}
PUTBACK;
call_pv("MyPerl::Request::handler", G_SCALAR);
SPAGAIN;
string = POPpx;
size = (size_t)n_a;
```

Az `XPUSHs` feladata az összes sort karaktersorozatok tömbjévé alakítani, amelyet aztán a `MyPerl::Request()` könyvtár `handler()` Perl függvénye kap meg. Ez a Perl-modul nagyon hasonló az `Apache::Request` moduljához, attól eltekintve, hogy ahol az Apache a fájlnevet használja a végrehajtott kód nyomon követéséhez, ott a MyPerl magát a kódot.
Minthogy a változó névleges mérete most már a visszatérítendő adatméretet is tartalmazza, le kell foglalnunk hozzá a helyet:

```
if (size) {
    if(pass->size < size) {
        newspot = (char *)realloc
                  ➥(pass->returnable, size);
        if(!newspot) {
            error[0] = ´1´;
            returnable =  NULL;
            goto error;
        }
        pass->size = size;
        pass->returnable = newspot;
    }
    // a jelenlegi méretet mindig tudjuk,
    // ez kevesebb is lehet, mint a teljes méret
    pass->messagesize = size;
    memcpy(pass->returnable, string, size);
} else {
    is_null[0] = ´1´;
}
error:
PUTBACK;
FREETMPS;
LEAVE;
*length = pass->messagesize;

return pass->returnable;
}
```

Itt tároljuk a kiszolgálónak küldendő adatokat. A `realloc()` memóriahívást használjuk, ha további memóriára van szükségünk. Ha a Perl-értelmezőtől nem kapunk további adatot, az `is_null` értékét 1-re állítjuk. Ezáltal a MySQL tudni fogja, hogy `NULL` értéket kell visszaadnia az ügyfélnek. A MyPerl azt is ellenőrzi, hogy a `call_pv()*` függvényben esetleg használt memóriát felszabadítottuk-e.
Ezután az összes sorra meghívjuk a `myperl()` függvényt. Miután a MySQL az ügyfélnek visszaadta az adatokat, meghívja a `deinit` függvényt, felszabadítva az értelmezőt és a további lefoglalt memóriát:

```
void myperl_deinit(UDF_INIT *initid)
{
    myperl_passable *pass =
          ➥(myperl_passable *)initid->ptr ;
    perl_destruct(pass->myperl);
    perl_free(pass->myperl);
    free(pass->returnable);
    free(initid->ptr);
}
```

Példák a MyPerl használatára

Most már birtokunkban van egy Perl UDF, így végre tudjuk hajtani a következő egyszerű trükköt:

```
mysql> select myperl(´return $ARGV[0]´,User)
       from mysql.user;
```

Mint láthatjuk, minden sor az `@ARGV` változó értékeinek

felel meg. A CPAN-modulokkal kiegészített MyPerlt használhatjuk közvetlen adatbevitelre is. A következő példa URL-ek listáját tölti le, majd a tartalmat adatbázisba helyezi:

```
mysql> insert into html select
       myperl("use LWP::Simple;
       my $content = get($ARGV[0]);
       return $content", url) from urls;
```

Az `XML::Simple` és `XML::XPath` modul segítségével bármilyen olyan XML formátumot lekérdezhetünk, amit esetleg az adatbázisunkban tároltunk. Én a MyPerl segítségével ellenőriztem például az adatbázisomban tárolt, sorosított (serialized) Perl-objektumokat.

Mi a helyzet a GROUP BY utasítással?

Fentebb ugyan bemutattuk, hogyan használjuk a kódot sorok lekérdezése esetén, mindez nemigen fog működni azoknál a lekérdezéseknél, amelyek `GROUP BY` segítségével kezeli az adatkészletet. Ezért létezik egy másik UDF-fajta, amelyet aggregátnak nevezünk. Az aggregátok abban különböznek szelídebb unokatestvéreiktől, hogy két további állapotuk is létezik: a `reset` és az `add`. Az aggregát UDF-ek esetében az add függvény kezeli az egyes sorokat, és a `request` lépcső válogatja ki az eredményeket és küldi el az adatokat az ügyfélnek. A `reset` állapot minden adatkészletnek az elején hívódik meg, ezért mindenképpen lefut, legalább egyszer. A MyPerl már most is rendelkezik aggregát UDF-fel, ez azonban még tervezés alatt áll.

Összegzés

Azáltal, hogy a Perlt beillesztjük a MySQL rendszerébe, az adatbázissal elvégezhető műveletek kínálata jócskán kiszélesedik. Igaz ugyan, hogy sok esetben éppen az a legjobb megoldás, ha a lehető legegyszerűbbre formáljuk az adatbázisunkat, vannak esetek, amikor ez mégsem tűnik célravezetőnek. Képzeljük el, hogy gigabájt nagyságrendű szövegeket kell az adatbázisból kivennünk és további feldolgozásra az ügyfélnek továbbküldenünk. Az adatküldésre fordított idő lesz a mérvadó; ha a Perl segítségével képesek vagyunk közvetlenül az adatbázisban feldolgozni az adatokat, bizony, nem kevés időt takaríthatunk meg. Ha használhatjuk a Perl fejlett szabályos kifejezéseit, könnyen készíthetünk ügyfeleket olyan nyelveken is, amelyek nem rendelkeznek kifinomult kifejezéstámogatással. Biztos vagyok benne, hogy a saját környezetünkben is jó alkalmazási lehetőségeket találunk. A ➲ http://software.tangent.org oldalon találjuk meg a MyPerlt, valamint további UDF-eket, amelyeket felhasználhatunk a saját változatunk elkészítéséhez.

Linux Journal 2003. december, 116. szám

Brian Aker (brian@tangent.org)
Ideje nagy részében MySQL- és Apache-modulokkal foglalkozik, keze nyomát viseli többek közt a mod_layout és a mod_mp3 Apache adatfolyam-szolgáltatásmodul is.