動的共有オブジェクト（DSO）サポート

Apache HTTP サーバはモジュール化されたプログラムで、 管理者がモジュールを選択することでサーバに組み込む機能を選ぶことができます。 モジュールはサーバがビルドされるときに httpd バイナリに 静的に組み込むことができます。もしくは、httpd バイナリとは別に存在する動的共有オブジェクト（訳注: Dynamic Shared Object）(DSO) としてコンパイルすることも できます。DSO モジュールはサーバがビルドされるときにコンパイルしたり、 Apache 拡張ツール（apxs [1]）を 使って後でコンパイルして追加したりできます。

この文書は DSO モジュールの使い方と、仕組みについて 説明します。

トピック

実装

関連モジュール	関連ディレクティブ
mod_so	LoadModule

個々の Apache モジュールをロードするための DSO サポートは mod_so.c というモジュールの機能に基づいています。 このモジュール は Apache のコアに静的に組み込まれている必要があります。 それは core.c 以外では DSO にできない唯一の モジュールです。事実上、他のすべての Apache のモジュールは、 インストールの文書 [2]で説明されているように、configure の --enable-module=shared オプションでそれぞれを DSO ビルドにすることにより、DSO モジュールにすることができます。 mod_foo.so のような DSO にモジュールがコンパイルされれば、 httpd.conf ファイル中で mod_so の LoadModule ディレクティブを使うことでサーバの起動や再起動時にこのモジュールを ロードするようにできます。

Apache モジュール用の（特にサードパーティモジュールの）DSO ファイルの 作成を簡単にするために、apxs [1] (APache eXtenSion) という新しいサポートプログラムがあります。Apache のソースツリーの外で DSO モジュールをビルドするために 使うことができます。発想は単純です: Apache のインストール時の configure、make install のときに Apache の C ヘッダをインストールし、DSO ビルド用のプラットフォーム依存の コンパイラとリンカのフラグを apxs プログラムに追加します。 これにより、ユーザが Apache の配布ソースツリーなしで、さらに DSO サポートのためのプラットフォーム依存のコンパイラやリンカの フラグをいじることなく Apache のモジュールのソースをコンパイル できるようになります。

使用法の概要

Apache 2.0 の DSO 機能の概略を知ることができるための、 短く簡潔な概要です:

1. 配布されている Apache モジュール、仮に mod_foo.c として、それを DSO mod_foo.so にビルド、インストール:

```
$ ./configure --prefix=/path/to/install --enable-foo=shared
```

```
$ make install
```

2. サードパーティ Apache モジュール、仮に mod_foo.c として、それを DSO mod_foo.so にビルド、インストール:

```
$ ./configure --add-module=module_type:/path/to/3rdparty/mod_foo.c
--enable-foo=shared
$ make install
```

3. 共有モジュールの 後々のインストール のために Apache を設定:

```
$ ./configure --enable-so
$ make install
```

4. サードパーティ Apache モジュール、仮に mod_foo.c として、それを apxs [1] を使って Apache ソースツリーの外で DSO にビルド、インストール:

```
$ cd /path/to/3rdparty
$ apxs -c mod_foo.c
$ apxs -i -a -n foo mod_foo.la
```

どの場合においても、共有モジュールをコンパイルした後で、 httpd.conf で LoadModule ディレクティブを使って Apache がモジュールを使用するように しなければなりません。

背景

最近の Unix 系の OS には 動的共有オブジェクト (DSO) の動的リンク/ロードという気のきいた機構が 存在します。これは、実行時にプログラムのアドレス空間に ロードできるような特別な形式でプログラムをビルドすることを 可能にします。

このロードは二つの方法で行なうことができます: 実行プログラムが 起動されたときに lod.so というシステムプログラム により自動的に行なわれる方法と、実行プログラム中から、システムコール dlopen()/dlsym() による Unix ローダへの プログラムシステムのインタフェースを使って手動で行なう方法とが あります。

最初の方法では DSO は普通は共有ライブラリや DSO ライブラリ と呼ばれていて、DSO の名前は libfoo.so や libfoo.so.1.2 のようになっています。 これらはシステムディレクトリ (通常 /usr/lib) に存在し、 実行プログラムへのリンクはビルド時に -lfoo をリンカに指定することで確立されます。これによりライブラリへの参照が実行プログラムの ファイルに書き込まれて、起動時に Unix のローダが /usr/lib や、 リンカの -R のようなオプションによりハードコードされたパス、 環境変数 LD_LIBRARY_PATH により設定されたパス、の中から libfoo.so の場所を見つけることができます。それから、 実行プログラム中の (まだ未解決の) シンボルを DSO にあるシンボルで 解決します。

普通は実行プログラム中のシンボルは DSO からは参照されません (DSO は一般的なコードによる再利用可能なライブラリですので)。 ですから、さらなるシンボルの解決は必要ありません。 シンボルは Unix ローダにより完全な解決が行なわれますので、実行ファイル自身は 何もする必要がありません。(実際のところ、静的でない方法でリンクされている すべての実行プログラムに組み込まれている開始用のコードの一部に ld.so を起動するコードが含まれています)。よく使われる ライブラリの動的ロードの利点は明らかです。ライブラリのコードは システムライブラリに libc.so のようにして一度保存するだけでよく、 プ

ログラムのために必要なディスクの領域を節約することができます。

二つめの方法では DSO は普通は共有オブジェクトや DSO ファイルと呼ばれていて、任意の拡張子を付けることができます (ただし、標準的な名前は foo.so です)。 これらのファイルは通常はプログラム専用のディレクトリに置かれ、 これらを使う実行プログラムへのリンクは自動的にはされません。 ですので、実行プログラムは dlopen() を使って 実行時に手動で DSO をプログラムのアドレス空間にロードする必要があります。 この時点では実行プログラムに対して DSO のシンボルの解決は行なわれません。 しかし、その代わりに Unix のローダが DSO の (まだ未解決の) シンボルを 実行プログラムによりエクスポートされたシンボルと既にロードされた DSO ライブラリによりエクスポートされたシンボル (特に、どこにでもある libc.so のすべてのシンボル) で自動的に解決します。 こうすることで、DSO は最初から静的にリンクされていたかのように、 実行プログラムのシンボルを知ることができます。

最後に、DSO の API を利点を生かすために、プログラムは 後でディスパッチテーブルなどでシンボルを使うことができるように、 dlsym() を使っていくつかのシンボルを解決します。 すなわち: 実行プログラムは必要なすべてのシンボルを手動で解決しなければ なりません。この機構の利点はプログラムのオプショナルな部分は 必要になるまでロードする必要がない (だからメモリも消費しない) ことです。必要ならば、基本プログラムの機能を拡張するために これらの部分を動的にロードすることができます。

この DSO 機構は簡単なように見えますが、少なくとも一つ難しい点が あります: プログラムを拡張するために DSO を使っているときに、 DSO が実行プログラムからシンボルを解決する点です (二番目の方法)。 これはなぜでしょうか。それは、DSO のシンボルを実行プログラムの シンボルから「逆解決」するというのはライブラリの設計 (ライブラリはそれを使用するプログラムのことは何も 知らない) に反していて、この機能はすべてのプラットフォームに あるわけではなく、標準化もされていないからです。 実際には実行プログラムのグローバルなシンボルは再エクスポートされることは あまりなく、DSO から使うことができません。リンカにグローバルシンボルすべてを エクスポートするようにさせる方法を見つけることが、実行時にプログラムを 拡張するために DSO を使うときの一番の問題です。

共有ライブラリのアプローチが普通の方法です。DSO 機構はそのために 設計されたものですから。したがって、その方法はオペレーティングシステムが 提供するほとんどすべての種類のライブラリで使われています。 一方、プログラムの拡張のために共有オブジェクトを使用する、という方は あまり使われていません。

1998 年の時点で、実行時に実際に機能拡張のために DSO 機構を使っている ソフトウェアパッケージは少しだけでした: Perl 5 (XS 機構と DnaLoader モジュール によるもの)、Netscape サーバなどです。Apache はすでに モジュールの概念を使って機能拡張をしていて、内部的にディスパッチリストに 基づいた外部モジュールの Apache コア機能へのリンクを行なっていましたので、 バージョン 1.3 から、Apache も DSO 機構を使う仲間になりました。 Apache は実行時に DSO を使ってモジュールをロードするようにすでに 運命付けられていたのです。

利点と欠点

上記の DSO に基づいた機能は以下の利点があります:

- 実際のサーバプロセスを組み立てるために、 ビルド時に configure のオプションを使

う代わりに 実行時に httpd.conf の設定用コマンド LoadModule を使うことができますので、サーバパッケージの柔軟性が高まりました。 たとえば、一つの Apache のインストールから 違う構成のサーバ (標準版と SSL 版、最小構成と拡張版 [mod_perl, PHP3] など) を実行することができます。

- インストールの後であっても、サーバのパッケージをサードパーティ モジュールで簡単に拡張できるようになりました。これは、Apache コア パッケージと、PHP3, mod_perl, mod_fastcgi など の追加の パッケージを作成できるので、少なくともベンダのパッケージ管理者にとって 大きな利点があります。
- Apache モジュールの開発が簡単になります。 これは DSO/apxs の組み合わせにより、Apache ソースツリーの 外で作業でき、開発中のモジュールの新しいバージョンを 実行中の Apache サーバに組み込むために apxs -i と apachectl restart を行なうだけで良くなるからです。

DSO には以下の欠点があります:

- すべてのオペレーティングシステムがプログラムのアドレス空間に コードを動的ロードすることをサポートしているわではないので、 プラットフォームによっては DSO 機構は使えません。
- Unix のローダがシンボルの解決をする必要ができたので、 そのオーバヘッドによりサーバの起動時間が約 20% 遅くなっています。
- 位置非依存コード (PIC) (訳注 position independent code) は 相対アドレスのために複雑なアセンブラのトリックが必要なことがあり、 それは必ずしも絶対アドレスと同じくらいの速度がでるわけではありませんので、 プラットフォームによってはサーバの実行速度が約 5% 遅くなります。
- DSO モジュールはすべてのプラットフォームで他の DSO に基づいた ライブラリに対してリンクできる (ld -lfoo) というわけではありませんので (たとえば、a.out のプラットフォームでは この機能はありませんが、ELF のプラットフォームにはあります)、すべての種類のモジュールに DSO 機構を使えるわけではありません。 言い換えると、DSO ファイルとしてコンパイルされたモジュールの 使えるシンボルは、 Apache のコアのシンボル、C ライブラリ (libc) と Apache コアが使っている他のすべての静的なライブラリと動的ライブラリの シンボル、PIC による静的なライブラリ (libfoo.a) の シンボルのみに制限されます。その他のコードを使う方法は、 Apache コア自身がすでにそのコードへの参照があるようにするか、 dlopen () を使ってコードを自分自身でロードするかの どちらかしかありません。

URI References

[1] http://httpd.apache.org/docs-2.1/programs/apxs.html
[2] http://httpd.apache.org/docs-2.1/install.html