PHP Internals Book

PHP 7

Julien Pauli
Nikita Popov
Anthony Ferrara

目录

第一章 介绍 **1**

1.1 介绍 1

第二章 使用 php 构建系统 **3**

2.1 构建 php 3

2.1.1 为什么不使用包 (packages) ? 3

2.1.2 获取源代码 4

2.1.3 构建概述 5

2.1.4 ./buildconf 脚本 6

2.1.5 ./configure 脚本 6

2.1.6 make 和 make install 9

2.1.7 运行测试套件 12

2.1.8 修复编译问题并进行清理 13

2.2 构建 PHP 扩展 13

2.2.1 加载共享扩展 14

2.2.2 通过 PECL 安装扩展 15

2.2.3 向 PHP 源代码树添加扩展 15

2.2.4 使用 phpize 构建扩展 16

2.2.5 展示扩展信息 16

2.2.6 扩展 API 兼容性 18

第三章 内部类型 **22**

3.1 Zvals 22

3.1.1 基本结构 22

3.1.2 Zval 内存管理和垃圾收集 31

3.1.3 zvals 里的强制类型转换和操作 32

3.2 字符串管理 33

3.2.1 字符串管理:zend_string 33

3.2.2 smart_str API 41

3.2.3 PHP 自定义的 printf 函数 44

3.3 资源类型:zend_resource 49

3.3.1 什么是“资源”类型? 49

3.3.2 资源类型和资源销毁 49

3.3.3 使用资源 50

3.3.4 资源引用计数 52

3.3.5 持久化资源 52

3.4 HashTables: zend_array 53

3.5 Functions: zend_function 54

3.6 Objects and classes 55

第四章 扩展开发 **56**

4.1 学习 PHP 生命周期 56

第一章 附录 **57**

A.1 常用宏及函数 57

第一章　介绍

1.1　介绍

本书是几个 php 语言开发者之间合作的成就，为了更好的记录和描述 php 内部如何工作。

本书有三个主要目标：

- 记录并描述 PHP 内部的工作原理
- 记录并描述如何使用扩展（extensions）来扩展语言
- 记录并描述如何与社区互动来开发 PHP 语言

这本书的目标用户是有 C 语言编程经验的开发人员。

然而，只要有可能，我们将尝试提炼信息并对其进行总结，以便不熟悉 C 语言的开发人员仍然能够理解这些内容。

无论如何，让我们坚持。如果你不懂 C 语言，你将无法实现高效、稳定的（在任何平台下无崩溃）、高性能和有用的程序。

这里有一些相当不错的关于 C 语言本身、它的生态系统、构建工具以及操作系统 API 的在线资源：

- http://www.tenouk.com
- https://en.wikibooks.org/wiki/C_Programming
- http://c-faq.com
- https://www.gnu.org/software/libc
- http://www.faqs.org/docs/Linux-HOWTO/Program-Library-HOWTO.html
- http://www.iecc.com/linker/linker10.html

我们也极力推荐你一些书。您将了解如何有效地使用 C 语言，以及如何使其转化为高效的 CPU 指令，以便您可以设计强大/快速/可靠和安全的程序。

- 《C 程序设计语言》The C Programming Language (Ritchie & Kernighan)
- Advanced Topics in C Core Concepts in Data Structures
- 《笨办法学 C 语言》Learn C the Hard Way
- 《软件调试的艺术》The Art of Debugging with GDB DDD and Eclipse
- 《Linux/UNIX 系统编程手册》The Linux Programming Interface
- 《Linux 高级程序设计》Advanced Linux Programming
- 《高效程序的奥秘》Hackers Delight
- 《编程卓越之道》Write Great Code (2 Volumes)

这本书还是半成品，有些章节还没有写完。我们不关注特定的顺序，而是根据我们的感觉添加内容。

本书的仓库可以在GitHub上找到。请报告问题并提供问题跟踪反馈。

第二章　使用 php 构建系统

在本章中，我们将解释如何使用 PHP 构建系统来编译 PHP 本身及其扩展。本章还不会涉及编写您自己的 autoconf 构建指令，仅仅解释如何使用工具。

2.1　构建 php

本章将说明如何在某种程度上适用于扩展开发或者内核修改的方式编译 PHP。我们只包括基于类 Unix 系统的构建。

如果您希望在 Windows 上构建 PHP，您应该查看PHP Wiki中一步一步的构建说明 [1]。

本章还概述了 PHP 构建系统的工作原理以及它使用的工具，但详细说明超出了本书的范围。

[1] 免责声明: 对于在 Windows 上编译 PHP 所造成的任何不良健康影响，我们不承担任何责任。

2.1.1　为什么不使用包 (packages) ?

如果你当前正在使用 php，你可能会通过包管理安装，使用 *sudo apt-get install php* 命令。

在解释实际编译前，你应当首先理解为什么自己编译是必要的并且不能仅仅使用预编译包。

对此有几个原因:

首先，预编译包只包含生成的二进制文件，但是省略了编译扩展所必需的其他内容，例如头文件。通过安装开发包（通常称为 **php-dev**）可以很容易地解决上面这个问题。为了便于使用 **valgrind** 或 **gdb** 进行调试，可以另外安装调试符号，这通常是另一个称为 **php-dbg** 的包。

但是即使你安装了头文件和调试符号，你使用的仍然是 PHP 发布 (release) 版本。这意味着它将以高优化级别构建，这使得调试非常困难。此外，发布版本 (release) 不会生成有关内存泄漏或数据结构不一致的警告。此外，预编译包不支持线程安全，这在开发过程中非常有用。

另一个问题是，几乎所有发行版都会向 PHP 打其他补丁。在某些情况下，这些补丁仅包含与配置相关的微小更改，但某些发行版使用像 Suhosin 这样的高度侵入性补丁。已知其中一些补丁会引入与 opcache 等低级扩展的不兼容性。

PHP 仅提供对 php.net 上提供的软件的支持，而不对分发修改版本支持。如果您想报告错误，提交补丁或使用我们的帮助渠道进行扩展编写，您应该始终针对官方 PHP 版本。当我们在本书中讨论“PHP”时，我们总是指官方支持的版本。

2.1.2 获取源代码

在构建 PHP 之前，首先需要获取其源代码。有两种方法可以执行此操作：您可以从PHP 的下载页面下载存档文件，或者从git.php.net（或Github 上的镜像）克隆 git 仓库。

对于这两种情况构建过程略有不同: 通过 git 仓的库不捆绑 configure 脚本，因此您需要使用 buildconf 脚本生成它，该脚本使用 autoconf 工具生成。此外，git 仓库的不包含预生成的解析器，因此还需要安装 **bison**。

我们建议您从 git 中签出源代码，因为这将为您提供一种简单的方法来更新您的安装并尝试使用不同版本的代码。如果要提交补丁或拉取 PHP 请求还需要 git checkout。

要克隆仓库, 在 shell 中运行以下命令:

```
~> git clone http://git.php.net/repository/php-src.git
~> cd php-src
# by default you will be on the master branch, which is the current
# development version. You can check out a stable branch instead:
~/php-src> git checkout PHP-7.0
```

如果你有关于 git checkout 的问题，请查看 PHP wiki 上的Git FAQ。如果你想为 PHP 本身做贡献，Git FAQ 还解释了如何设置 git。此外，它包含有关为不同 PHP 版本设置多个工作目录的说明。如果您需要针对多个 PHP 版本和配置测试扩展或更改，这可能非常有用。

在继续之前，您还应该使用包管理器安装一些基本的构建依赖项（默认情况下，您可能已经安装了前三个）:

- gcc 或者其他编译器套件
- libc-dev，它提供了 C 标准库，包含头文件。
- make，PHP 使用的构建管理工具。
- autoconf, 用于生成配置脚本。
  - 2.59 或者更高 (PHP 7.0-7.1)
  - 2.64 或者更高 (PHP 7.2)
  - 2.68 或者更高 (PHP 7.3)

- libtool，帮助管理共享库。
- bison (2.4 或者更高)，用于生成 PHP 解析器.
- (可选) re2c, 用于生成 PHP 词法分析器。由于 git 仓库里已经包含生成的词法分析器，因此如果您希望对其进行更改，则只需要 re2c。

在 Debian/Ubuntu 服务器上，你可以通过以下命令安装这些软件：

```
~/php-src> sudo apt-get install build-essential autoconf libtool bison re2c
```

根据您在./configure 阶段启用的扩展，PHP 将需要许多额外的库。安装这些软件包时，请检查是否有以-dev 或-devel 结尾的软件包版本，并且安装它们。没有 dev 的软件包通常不包含必要的头文件。例如，默认的 PHP 构建将需要 libxml，您可以通过 libxml2-dev 软件包安装它。

如果你在使用 Debian 或者 Ubuntu，你可以使用 *sudo apt-get build-dep php7* 来一次性安装大量可选的构建依赖项。如果您的目标仅仅是一个默认的构建，那么许多依赖包都是不必要的。

2.1.3 构建概述

在仔细研究各个构建步骤的作用之前，以下是为“默认”PHP 构建执行所需的命令：

```
~/php-src> ./buildconf      # 只有在构建从 git 获取的代码时才需要
~/php-src> ./configure
~/php-src> make -jN
```

为了快速构建，请将 N 替换为您可用的 CPU 核数（*grep "cpu cores" /proc/cpuinfo*）。

默认情况下 PHP 将要构建 CLI 和 CGI SAPIS 二进制文件，他们分别位于 sapi/cli/php 和 sapi/cgi/php-cgi。验证以下是否编译成功，尝试运行 ‘sapi/cli/php -v‘命令。

此外你可以运行 *sudo make install* 命令来将 php 安装到**/usr/local** 目录。在配置阶段可以指定**–prefix** 参数来更改目标目录。

```
~/php-src> ./configure --prefix=$HOME/myphp
~/php-src> make -jN
~/php-src> make install
```

这里 $HOME/myphp 是在 *make install* 步骤中使用的安装位置。请注意，这不是必需的，但是如果您想在扩展开发之外使用 PHP 构建，那么安装 PHP 是很方便的。

现在让我们细细研究各个构建步骤!

2.1.4 ./buildconf 脚本

如果你使用 git 仓库的代码构建，首先要做的事情是运行**./buildconf** 脚本。这个脚本仅仅是调用 **build/build.mk** 这个 makfile 文件，而这个文件依次又调用了 **build/build2.mk** 文件。

这些 makefile 文件的主要工作是运行 **autoconf** 来生成**./configure** 脚本和运行 **autoheader** 来生成 **main/php_config.h.in** 模版。后面生成的文件将被 configure 用来生成最终的配置头文件 **main/php_config.h**。

两个实用程序都会从 **configure.in** 文件生成结果 (指定大多数 PHP 构建过程)，**acinclude.m4** 文件（它指定了大量特定于 PHP 的 M4 宏）和个别扩展的 **config.m4** 文件和 SAPIS（以及其他一些 m4 文件）

好消息是编写扩展甚至进行核心修改不需要与构建系统进行太多交互。稍后您将不得不编写小的 **config.m4** 文件，但这些文件通常只使用 **acinclude.m4** 提供的两个或三个高级宏。因此，我们在此不再详述。

./buildconf 脚本仅仅只有两个选项: –debug 在调用 **autoconf** 和 **autoheader** 时将会显示警报信息。除非你想在构建系统上工作，否则这个选项对你来说不太重要。

第二个选项是**–force**, 允许在发版包（例如：如果您下载了打包 (release) 的源代码并希望生成一个新的**./configure**）运行**./buildconf** 并清理配置缓存 **config.cache** 和 **autom4te.cache**

如果你使用 *git pull*（或者一些其他命令）更新你的 git 仓库并在 **make** 步骤时报出奇怪的错误，这通常意味着构建配置发生了更改，您需要运行*./buildconf —force*。

2.1.5 ./configure 脚本

一旦**./configure** 脚本生成，你就可以使用它来构建你自定义的 php。你可以使用**–help** 命令来列出其所支持的选项:

```
~/php-src> ./configure --help | less
```

帮助的第一部分将列出各种通用选项，所有基于 autoconf 的配置脚本都支持它们。其中一个是已经提及的**–prefix = DIR**，它改变了由 **make install** 使用的安装目录。另外一个有用的选项是**-C**, 它将各种测试的结果缓存在 **config.cache** 文件中并且加速后面**./configure** 的调用。只有在您已经具有可用的构建并希望在不同配置之间快速切换之后，使用此选项才有意义。

除了通用的 autoconf 选项之外，还有许多特定于 PHP 的设置。例如，您可以使用**–enable-NAME** 和**–disable-NAME** 开关来选择编译哪个扩展和 SAPIs。如果扩展或 SAPI 具有外部依赖关系，则需要使用**–with-NAME** 和**–without-NAME** 代替。如果通过 **NAME** 所需的库不在默认位置（例如: 因为您自己编译的），则可以使用**–with-NAME=DIR** 为其指定位置。

默认情况下 PHP 将构建 CLI 和 CGI SAPIs，以及一些扩展。您可以使用**-m** 选项查看 PHP 二进制文件中包含的扩展名。对于默认的 PHP7.0 构建，结果如下:

```
~/php-src> sapi/cli/php -m
```

```
[PHP Modules]
Core
ctype
date
dom
fileinfo
filter
hash
iconv
json
libxml
pcre
PDO
pdo_sqlite
Phar
posix
Reflection
session
SimpleXML
SPL
sqlite3
standard
tokenizer
xml
xmlreader
xmlwriter
```

如果您现在想要停止编译 CGI SAPI，以及 tokenizer 和 sqlite3 扩展，而是启用 opcache 和 gmp，相应的配置命令如下：

```
~/php-src> ./configure --disable-cgi --disable-tokenizer --without-sqlite3 \
           --enable-opcache --with-gmp
```

默认情况下，大多数扩展将进行静态编译，也就是它们将成为生成的二进制文件的一部分。默认情况下，只有 opcache 扩展是共享的，也就是说它会在 **modules/**目录下生成 **opcache.so** 共享对象。您可以通过编写**–enable-NAME=shared** 或**–with-NAME=shared**（但并非所有的扩展支持这点）将其他扩展编译为共享对象。我们将在下一节讨论如何使用共享扩展。

要了解您需要使用哪个开关以及扩展是否默认启用，请检查*./configure ——help*。如果开关是**–enable-NAME** 或**–with-NAME**，这就意味着此扩展默认情况下没有编译，并且需要显式启用。另一方面**–disable-NAME** 或者 **–without-NAME** 表示扩展默认编译，但可以显式禁用。

有些扩展总是被编译并且不能被禁用。要创建仅包含最少量扩展的构建，请使用**–disable-all** 选项：

```
~/php-src> ./configure --disable-all && make -jN
~/php-src> sapi/cli/php -m
[PHP Modules]
Core
date
pcre
Reflection
SPL
standard
```

如果您想要一个快速的构建并且不需要太多功能（例如在实现语言更改时），则**–disable-all** 选项非常有用。对于尽可能小的构建，您还可以额外指定**–disable-cgi**，这样就只生成 CLI 二进制文件。

还有两个开关，在开发扩展或使用 PHP 时应始终指定：

–enable-debug 启用调试模式，它有多种效果: 编译将使用**-g** 来生成调试符号并使用最低的优化级别**-O0**。这将使得 php 运行慢很多，但使用 gdb 等工具进行调试更具可预测性。此外，调试模式定义了 **ZEND_DEBUG** 宏，它将在引擎中启用各种调试助手。除其他外，内存泄漏以及某些数据结构的不正确使用也将被报告。

–enable-maintainer-zts 启用线程安全。此开关将定义 **ZTS** 宏，依次来启用 PHP 使用的整个 TSRM（线程安全资源管理器）机制。为 PHP 编写线程安全扩展非常简单，但前提是确保启用此开关。如果您需要更多有关 PHP 线程安全和全局内存管理的信息，您应该阅读全局管理章节

另一方面，如果要为代码执行性能基准测试，则不应使用这些选项中的任何一个，因为两者都会导致显着和不对称的减速。

注意**–enable-debug** 和**–enable-maintainer-zts** 改变 PHP 二进制文件的 ABI，例如：通过向许多函数添加额外的参数。因此，在调试模式下编译的共享扩展将与在发布模式下构建的 PHP 二进制文件不兼容。类似地，线程安全扩展（ZTS）与非线程安全的 PHP 构建（NTS）不兼容。

由于 ABI 不兼容，**make install**（和 PECL 安装）会根据以下选项将共享扩展存放在不同的目录中：

$PREFIX/lib/php/extensions/no-debug-non-zts-API_NO 非线程安全正式版本 $PREFIX/lib/php/extensions/debug-non-zts-API_NO 非线程安全调试版本 $PREFIX/lib/php/extensions/no-debug-zts-API_NO 线程安全正式版本 $PREFIX/lib/php/extensions/debug-zts-API_NO 非线程安全调试版本

上面的 **API_NO** 占位符指的是 **ZEND_MODULE_API_NO**，它只是一个像 20100525 这样的日期，用于内部 API 版本控制。

在大多数情况下，上面描述的配置开关应该足够了，但当然**./configure** 提供了更多选项，您可以在帮助中找到这些选项。除了传递要配置的选项之外，您还可以指定许多环境变量。在 configure help（*./configure –help | tail -25*）的末尾记录了一些更重要的内容。

例如您可以通过使用 **CC** 来使用不同的编译器和使用 **CFLAGS** 来更改使用的编译标志:

```
~/php-src> ./configure --disable-all CC=clang CFLAGS="-O3 -march=native"
```

在此配置中，构建将使用 clang(而不是 gcc) 并使用非常高的优化级别 (**-O3 -march=native**)。

您可以使用其他编译器警告标志来帮助您发现一些错误。对于 GCC，您可以在GCC 手册中阅读它们

2.1.6 make 和 make install

配置完所有内容后，您可以使用 make 来执行实际编译:

```
~/php-src> make -jN    # where N is the number of cores
```

此操作的主要结果是生成启用 SAPIs 的 PHP 二进制文件（默认为 **sapi/cli/php** 和 **sapi/cgi/php-cgi**), 以及 **modules/**目录中的共享扩展。

现在你可以运行 **make install** 来将安装 PHP 到**/usr/local**(默认) 或者你通过**–frefix** 配置指令指定的任何目录。

make install 只会将一些文件复制到新位置。除非你在配置时指定**–without-pear**，否则它通常还会下载并且安装 PEAR。下面是默认 PHP 构建的结果树:

```
> tree -L 3 -F ~/myphp

/home/myuser/myphp
|-- bin
|   |-- pear*
|   |-- peardev*
|   |-- pecl*
|   |-- phar -> /home/myuser/myphp/bin/phar.phar*
|   |-- phar.phar*
|   |-- php*
|   |-- php-cgi*
|   |-- php-config*
|    -- phpize*
|-- etc
|   -- pear.conf
|-- include
|   -- php
|       |-- ext/
|       |-- include/
|       |-- main/
|       |-- sapi/
```

```
|       |-- TSRM/
|        -- Zend/
|-- lib
|   -- php
|       |-- Archive/
|       |-- build/
|       |-- Console/
|       |-- data/
|       |-- doc/
|       |-- OS/
|       |-- PEAR/
|       |-- PEAR5.php
|       |-- pearcmd.php
|       |-- PEAR.php
|       |-- peclcmd.php
|       |-- Structures/
|       |-- System.php
|       |-- test/
|        -- XML/
-- php
    -- man
        -- man1/
```

目录结构的简述:

- **bin/**目录包含 SAPI 二进制文件 (php 和 php-cgi), 以及 **phpize** 和 **php-config** 脚本。它也是各种 PEAR/PECL 脚本的所在地。
- *etc/*包含配置文件。注意默认的 **php.ini** 目录并不在这里。
- *include/php* 包含头文件，这些文件是构建其他扩展或在自定义软件中嵌入 PHP 所必需的。
- *lib/php* 包含 PEAR 文件。**lib/php/build** 目录包含构建扩展所必须的文件, 例如包含 PHP 的 M4 宏的 **acinclude.m4** 文件。如果我们已经编译了任何共享扩展，那么他们将存于 **lib/php/extensions** 的子目录中。
- *php/man* 显然包含 php 命令的手册页。

如前所述，默认的 **php.ini** 并不在 **etc/**目录下。您可以使用 PHP 的–ini 选项来显示其位置:

```
~/myphp/bin> ./php --ini
Configuration File (php.ini) Path: /home/myuser/myphp/lib
Loaded Configuration File:         (none)
Scan for additional .ini files in: (none)
Additional .ini files parsed:      (none)
```

正如您所看到的默认 php.ini 目录是 **$PREFIX/lib**（libdir）而不是 **$PREFIX/etc**（sysconfdir）。您可以使用**–with-config-file-path=PATH** 选项调整默认的 **php.ini** 位置。

另外要注意 *make install* 不会创建 ini 文件。如果您想使用 **php.ini** 文件，您需要自己创建一个。例如您可以复制默认的开发配置:

```
~/myphp/bin> cp ~/php-src/php.ini-development ~/myphp/lib/php.ini
~/myphp/bin> ./php --ini
Configuration File (php.ini) Path: /home/myuser/myphp/lib
Loaded Configuration File:         /home/myuser/myphp/lib/php.ini
Scan for additional .ini files in: (none)
Additional .ini files parsed:      (none)
```

bin/目录除了 PHP 二进制文件之外，还包含两个重要的脚本：**phpize** 和 **php-config**。

phpize 相当于**./buildconf** 的扩展。它从 **lib/php/build** 复制各种文件并且调用 **autoconf/autoheader**。您将在下一节中了解有关此工具的更多信息。

php-config 提供构建 PHP 的配置信息。试一下:

```
~/myphp/bin> ./php-config
Usage: ./php-config [OPTION]
Options:
  --prefix            [/home/myuser/myphp]
  --includes          [-I/home/myuser/myphp/include/php -I/home/myuser/myphp/
   include/php/main -I/home/myuser/myphp/include/php/TSRM -I/home/myuser/myphp
   /include/php/Zend -I/home/myuser/myphp/include/php/ext -I/home/myuser/myphp
   /include/php/ext/date/lib]
  --ldflags           [ -L/usr/lib/i386-linux-gnu]
  --libs              [-lcrypt   -lresolv -lcrypt -lrt -lrt -lm -ldl -lnsl  -
   lxml2 -lxml2 -lxml2 -lcrypt -lxml2 -lxml2 -lxml2 -lcrypt ]
  --extension-dir     [/home/myuser/myphp/lib/php/extensions/debug-zts
   -20100525]
  --include-dir       [/home/myuser/myphp/include/php]
  --man-dir           [/home/myuser/myphp/php/man]
  --php-binary        [/home/myuser/myphp/bin/php]
  --php-sapis         [ cli cgi]
  --configure-options [--prefix=/home/myuser/myphp --enable-debug --enable-
   maintainer-zts]
  --version           [5.4.16-dev]
  --vernum            [50416]
```

这个脚本类似于 linux 发行版使用的 pkg-config 脚本。在扩展构建过程中通过它来获取编译器选项和路径信

息。您还可以使用它快速获取有关您构建的信息，例如您的配置选项或默认扩展目录。这些信息也可以通过‘./php -i‘（phpinfo）提供，但 php-config 提供了更简单的形式（自动化工具很容易使用))。

2.1.7 运行测试套件

如果 **make** 命令成功完成，它将打印一条消息鼓励您运行 **make test**:

```
Build complete.
Don't forget to run 'make test'
```

make test 将针对我们测试套件运行 PHP CLI 二进制文件，该测试文件在 PHP 源码树不同的 ‘tests/‘目录下。作为默认构建要针对大约 9000 个测试（对于最小的构建来说更少，如果启用其他扩展则更多），这可能要花费几分钟。*make test* 当前不支持并行，因此指定-jN 选项不会使其更快。

如果这是你第一次在你的平台上编译 PHP，我们鼓励你运行测试套件。根据你的系统和编译环境，你通过运行测试套件可能发现一些问题。如果有任何失败，运行脚本将会询问你是否是向我们的 QA 平台发送一份报告，这将允许贡献者来分析这些失败。请注意有少许测试失败这很正常并且只要你没有看到几十个失败，你的构建就可以正常运行。

make test 命令在内部使用 CLI 二进制文件调用 run-tests.php 文件。您可以运行 *sapi/cli/php run-tests.php –help* 来显示此脚本接受的选项列表。

如果手动运行 **run-tests.php**，则需要指定**-p** 或**-P** 选项（或丑陋的环境变量）:

```
~/php-src> sapi/cli/php run-tests.php -p pwd/sapi/cli/php
~/php-src> sapi/cli/php run-tests.php -P
```

-p 用于显式地指定要测试的二进制文件。注意，为了正确运行所有测试，这应该是一个绝对路径 (或者独立于调用它的目录)。**-P** 是一个快捷方式，它将使用 **run-tests.php** 调用的二进制文件。在上面的例子中，两种方法都是相同的。

你可以通过限定的某些目录作为参数传递给 **run-tests.php**，而不是运行整个测试套件。例如只测试 Zend 引擎、反射扩展和数组函数:

```
~/php-src> sapi/cli/php run-tests.php -P Zend/ ext/reflection/ ext/standard/
   tests/array/
```

这个非常有用，因为它允许你快速的运行只与你修改的那部分测试套件。

你不需要显式地使用 **run-tests.php** 来传递选项或者限制目录。相反您可以使用 **TESTS** 变量通过 **make test** 传递其他参数。例如相当于前面的命令:

```
~/php-src> make test TESTS="Zend/ ext/reflection/ ext/standard/tests/array/"
```

我们稍后将要更详细的看一下 **run-tests.php** 系统。特别是还要讨论如何编写自己的测试以及如何调试测试失败。请参阅专用测试章节。

2.1.8 修复编译问题并进行清理

正如您可能知道的，**make** 执行增量构建, 也就是说它不会重新编译所有文件，只编译自上次调用后更改的那些**.c** 文件。这是缩短构建时间的好方法，但它并不总是有效: 例如如果修改头文件中的结构，**make** 将不会自动重新编译使用该头的所有**.c** 文件，从而导致构建失败。

如果在运行 **make** 时出现奇怪错误或者生成的二进制文件被破坏（例如如果 'make test'在运行第一个测试之前崩溃了），则应该尝试运行 *make clean*。这将删除所有编译对象，从而强制执行下一次 **make** 调用以执行完整构建。

有时在你更改**./configure** 选项时通常需要运行 *make clean*。如果只启用额外的扩展，增量构建应该是安全的，但是更改其他选项可能需要完全重新构建。

一个更激进的清理目标通过 *make distclean* 是可使用的。这将执行一个常规清理，但也会回滚通过调用**./configure** 命令生成的任何文件。它将删除配置缓存，Makefiles，配置头和各种其他文件。顾名思义，这个目标是"为发行版进行清理"，因此它主要由发版管理者使用。

编译问题的另一个来源是修改 **config.m4** 文件或者作为 PHP 构建系统一部分的其他文件。如果修改了此类文件，则必须重新运行**./buildconf** 脚本。如果您自己进行修改，您可能会记得运行该命令，但如果它作为 *git pull*（或其他一些更新命令）的一部分发生，则问题可能不那么明显。

如果你遇到任何奇怪的编译问题通过运行 *make clean* 没有得到解决，通过运行*./buildconf –force* 很可能会解决你的问题。为避免以后输入之前的**./configure** 选项，您可以使用**./config.nice** 脚本（它包含了你上次的**./configure** 调用）:

```
~/php-src> make clean
~/php-src> ./buildconf --force
~/php-src> ./config.nice
~/php-src> make -jN
```

PHP 提供的最后一个清理脚本是**./vcsclean**, 这个仅适用你是通过 git 获取源码的情况下。它实际上就是对 *git clean -X -f -d* 的调用，它将删除 git 忽略的所有未跟踪的文件和目录。你应该小心使用这个。

2.2 构建 PHP 扩展

既然你已经知道了如何编译 PHP 自身，我们将要继续编译其他扩展。我们将要讨论构建过程是如何工作的以及有哪些不同的选项可用。

2.2.1 加载共享扩展

正如你在前一节中已经知道的，PHP 扩展可以静态地构建到 PHP 二进制文件中，也可以编译成共享对象 (‘.so‘)。绝大多数附带的扩展默认是静态链接, 然而可以显式的传**–enable-EXTNAME=shared** 或者 **–with-EXTNAME=shared** 到**./configure** 来创建共享对象。

虽然静态扩展总是可用，但是共享扩展则需要通过使用 **extension** 或者 **zend_extension** ini 选项来加载。两个选项都采用要么是相对于**.so** 文件的绝对路径或者是相对于 **extension_dir** 的相对路径。

例如：考虑使用此配置行来进行 PHP 的构建编译：

```
~/php-src> ./configure --prefix=$HOME/myphp \
                       --enable-debug --enable-maintainer-zts \
                       --enable-opcache --with-gmp=shared
```

在这种情况下，opcache 扩展和 GMP 扩展都被编译为位于 **modules/**目录中的共享对象。您可以通过更改 **extension_dir** 或通过传递绝对路径来加载它们：

```
~/php-src> sapi/cli/php -dzend_extension=pwd/modules/opcache.so \
                        -dextension=pwd/modules/gmp.so
# or
~/php-src> sapi/cli/php -dextension_dir=pwd/modules \
                        -dzend_extension=opcache.so -dextension=gmp.so
```

在 **make install** 这一步，两个**.so** 文件将要被移动到你 PHP 安装时的 extension 目录，你可以使用 **php-config –extension-dir** 命令看查询此目录。对于上面的构建选项，它将是**/home/myuser/myphp/lib/php/extensions/ no-debug-non-zts-MODULE_API**。这个值也是 ini **‘extension_dir** 选项的默认值，因此不必显式指定它，可以直接加载扩展：

```
~/myphp> bin/php -dzend_extension=opcache.so -dextension=gmp.so
```

这就给我们留下了一个问题：你应该使用那种机制？共享对象允许你拥有一个基础的 PHP 二进制文件并且通过 php.ini 加载其他扩展。发行版利用了这一点，提供了一个简单的 PHP 包，并将扩展作为单独的包分发。另一方面，如果你正在编译自己的 PHP 二进制文件，则可能不需要这样做，因为你已经知道你需要哪些扩展。

根据经验，您要对 PHP 本身附带的扩展使用静态链接并且对其他的扩展使用共享扩展。原因很简单，因为作为共享对象构建外部扩展更容易 (或者至少不那么麻烦)，稍后您将看到这一点。另一个好处是你可以在不重新构建 PHP 的情况下更新扩展。

如果你需要了解关于 extensions 和 Zend extensions 的区别内容，您可以查看专门的章节。

2.2.2 通过 PECL 安装扩展

[PECL](http://pecl.php.net/)【PHP Extension Community Library】是 PHP 扩展社区库，为 PHP 提供了大量的扩展。当扩展被从主 PHP 发行版中删除时，它们通常继续存在于 PECL 中。同样，现在 PHP 附带的许多扩展以前都是 PECL 扩展。

除非你在 PHP 构建的配置阶段指定**–without-pear**，否则 **make install** 将会下载并安装作为 PEAR 一部分的 PECL。在 **$PREFIX/bin** 目录下你将会发现 **pecl** 脚本。安装扩展现在就像运行 *pecl install EXTNAME* 一样简单，例如：

```
~/myphp> bin/pecl install apcu
```

这个命令将会下载、编译和安装APCu扩展。结果是在你的扩展目录下将会生成一个 **apcu.so** 文件，然后可以通过传递 ini 选项 **extension=apcu.so** 来加载该扩展。

虽然 **pecl install** 对终端用户非常方便，但扩展开发人员对此不感兴趣。在下文中我们将描述两种手动构建扩展的方法：通过将扩展导入到 PHP 源代码树中（这允许静态链接）或者通过执行外部构建 (仅共享)

2.2.3 向 PHP 源代码树添加扩展

第三方扩展和与 PHP 附带的扩展之间没有本质区别。因此只需将外部扩展复制到 PHP 源代码树中，然后使用常用的构建过程及可构建外部扩展。我们将以 APCu 为例进行演示。

首先，你必须将扩展的源代码放入 PHP 源代码树的 **ext/EXTNAME** 目录中。如果扩展可以通过 git 获得，这就像从 **ext/**目录下克隆仓库一样简单：

```
~/php-src/ext> git clone https://github.com/krakjoe/apcu.git
```

或者你也可以下载源码包并解压它：

```
/tmp> wget http://pecl.php.net/get/apcu-4.0.2.tgz
/tmp> tar xzf apcu-4.0.2.tgz
/tmp> mkdir ~/php-src/ext/apcu
/tmp> cp -r apcu-4.0.2/. ~/php-src/ext/apcu
```

该扩展将包含一个 **config.m4** 文件，该文件指定了 autoconf 使用的特定于扩展的构建指令。要将它们合并到**./configure** 脚本中，您必须再次运行**./buildconf**。为了确保配置文件确实是重新生成的，建议提前删除它：

```
~/php-src> rm configure && ./buildconf --force
```

您现在可以使用**./config.nice** 脚本将 APCu 添加到现有配置中，或者使用全新的配置行重新开始：

```
~/php-src> ./config.nice --enable-apcu
# or
~/php-src> ./configure --enable-apcu # --other-options
```

最后运行 **make -jN** 来执行实际构建。由于我们没有使用——**enable-apcu=shared**, 这个扩展被静态地链接到 PHP 二进制文件中。也就是说，使用它不需要额外的操作。显然，您还可以使用 **make install** 来安装生成的二进制文件。

2.2.4 使用 phpize 构建扩展

通过使用我们在构建 PHP 章节我们已经提到过的 **phpize** 脚本，也可以从 PHP 构建中分开构建扩展。

phpize 的作用类似于 PHP 构建中使用的**./buildconf** 脚本: 首先它要通过从 **$PREFIX/lib/php/build** 复制文件导入 PHP 构建系统到你的扩展中。这些文件包括 **acinclude.m4**（PHP 的 M4 宏），**phpize.m4**（在你的扩展中将重命名为 **configure.in** 并包含主要构建指令）和 **run-tests.php**。

然后 **phpize** 将要调用 **autoconf** 生成一个**./configure** 文件，该文件可用于自定义扩展构建。请注意，没有必要将**–enable-apcu** 传递给它，因为这是隐含的假设。相反，您应该使用**–with-php-config** 来指定 **php-config** 脚本的路径:

```
/tmp/apcu-4.0.2> ~/myphp/bin/phpize
Configuring for:
PHP Api Version:         20121113
Zend Module Api No:      20121113
Zend Extension Api No:   220121113

/tmp/apcu-4.0.2> ./configure --with-php-config=$HOME/myphp/bin/php-config
/tmp/apcu-4.0.2> make -jN && make install
```

在构建扩展时，你应始终指定**–with-php-config** 选项（除非你只有一个全局安装的 PHP），否则**./configure** 将无法正确确定要构建的 PHP 版本和标志。指定 **php-config** 脚本还可以确保 **make install** 将生成的**.so** 文件 (可以在 **modules/**目录中找到) 移动到正确的扩展目录。

由于 **run-tests.php** 文件在 **phpize** 阶段也被复制，你可以使用 **make test**（或直接调用 **run-tests**）来运行扩展测试。

如果在更改后的增量构建失败，以删除编译对象为目标的 **make clean** 也是可用的并允许你强制完全重新构建扩展。另外 **phpize** 也提供了清理选项 **phpize –clean**。这将删除 **phpize** 导入的所有文件，以及**/configure** 脚本生成的文件。

2.2.5 展示扩展信息

PHP CLI 提供了几个选项用来显示扩展的有关信息。你已经知道了**-m**，它将列出所有加载的扩展。你可以使用它来验证扩展是否已正确加载:

```
~/myphp/bin> ./php -dextension=apcu.so -m | grep apcu
```

```
apcu
```

还有几个以**-r** 开头的开关，它们引入了反射功能。例如，您可以使用——**ri** 来显示扩展的配置:

```
~/myphp/bin> ./php -dextension=apcu.so --ri apcu
apcu

APCu Support => disabled
Version => 4.0.2
APCu Debugging => Disabled
MMAP Support => Enabled
MMAP File Mask =>
Serialization Support => broken
Revision => $Revision: 328290 $
Build Date => Jan  1 2014 16:40:00

Directive => Local Value => Master Value
apc.enabled => On => On
apc.shm_segments => 1 => 1
apc.shm_size => 32M => 32M
apc.entries_hint => 4096 => 4096
apc.gc_ttl => 3600 => 3600
apc.ttl => 0 => 0
# ...
```

–re 开关列出了扩展添加的所有 ini 设置，常量，函数和类:

```
~/myphp/bin> ./php -dextension=apcu.so --re apcu
Extension [ <persistent> extension #27 apcu version 4.0.2 ] {
  - INI {
    Entry [ apc.enabled <SYSTEM> ]
      Current = '1'
    }
    Entry [ apc.shm_segments <SYSTEM> ]
      Current = '1'
    }
    # ...
  }

  - Constants [1] {
    Constant [ boolean APCU_APC_FULL_BC ] { 1 }
  }
```

```
  - Functions {
    Function [ <internal:apcu> function apcu_cache_info ] {

      - Parameters [2] {
        Parameter #0 [ <optional> $type ]
        Parameter #1 [ <optional> $limited ]
      }
    }
    # ...
  }
}
```

–re 开关仅适用于普通扩展，Zend 扩展使用**–rz** 代替。你可以在 opcache 上试试这个：

```
~/myphp/bin> ./php -dzend_extension=opcache.so --rz "Zend OPcache"
Zend Extension [ Zend OPcache 7.0.3-dev Copyright (c) 1999-2013 by Zend
   Technologies <http://www.zend.com/> ]
```

正如你所看到的，这不会显示任何有用的信息。原因是 opcache 注册了普通扩展和 Zend 扩展，前者包含所有 ini 设置、常量和函数。因此，在这种特殊情况下，您仍然需要使用**–re**。其他 Zend 扩展通过**–rz** 提供其信息。

2.2.6 扩展 API 兼容性

扩展对 5 个主要因素非常敏感。如果它们不合适，扩展将不会加载到 PHP 中并且将无用：

- PHP Api Version
- Zend Module Api No
- Zend Extension Api No
- Debug mode
- Thread safety

phpize 工具会让你回想起其中的一些信息。因此如果你已经构建的 PHP 带有调试模式，并且尝试加载并使用一个不带调试模式构建的扩展，这根本行不通。其他检查也是如此。

PHP Api Version 是内部 API 的版本号。Zend Module Api No 和 Zend Extension Api No 分别是 PHP 扩展和 Zend 扩展 API 的。

这些编号稍后作为 C 宏传递给正在构建的扩展，因此它本身可以检查这些参数并根据 C 预处理器 **#ifdef** 采用不同的代码路径。当这些编号作为宏传递给扩展代码时，它们被编写在扩展结构中，所以任何时候只要你尝

试在 PHP 二进制文件中加载此扩展，都将根据 PHP 二进制文件本身的编号进行检查。如果它们不匹配，则不会加载扩展，并显示一条错误消息。

如果我们看一下扩展 C 结构，它看起来像这样：

```
zend_module_entry foo_module_entry = {
    STANDARD_MODULE_HEADER,
    "foo",
    foo_functions,
    PHP_MINIT(foo),
    PHP_MSHUTDOWN(foo),
    NULL,
    NULL,
    PHP_MINFO(foo),
    PHP_FOO_VERSION,
    STANDARD_MODULE_PROPERTIES
};
```

到目前为止我们感兴趣的是 **STANDARD_MODULE_HEADER** 宏。如果我们展开它，我们可以看到：

```
#define STANDARD_MODULE_HEADER_EX sizeof(zend_module_entry), ZEND_MODULE_API_NO
    , ZEND_DEBUG, USING_ZTS
#define STANDARD_MODULE_HEADER STANDARD_MODULE_HEADER_EX, NULL, NULL
```

注意 **ZEND_MODULE_API_NO** 和 **ZEND_DEBUG、USING_ZTS** 是如何使用的。

如果你查看 PHP 扩展的默认目录，它应该看起来想 **no-debug-non-zts-20090626**。正如你所猜测的，此目录由不同的部分连接在一起组成的：调试模式，下一项是线程安全信息，然后是 Zend Module Api No。因此默认情况下，PHP 试图帮助你使用扩展进行导航。

通常当你成为内核开发人员或者扩展开发人员时，你就不得不使用调试参数，并且如果必须处理 Windows 平台，则线程也会出现。你可以根据这些参数的一些情况多次编译相同的扩展。

请记住 PHP 的每个新主/次版本都会更改参数，例如 PHP Api 版本，这就是为什么需要针对新的 PHP 版本重新编译扩展。

```
> /path/to/php70/bin/phpize -v
Configuring for:
PHP Api Version:         20151012
Zend Module Api No:      20151012
Zend Extension Api No:   320151012
```

```
> /path/to/php71/bin/phpize -v
Configuring for:
PHP Api Version:         20160303
Zend Module Api No:      20160303
Zend Extension Api No:   320160303

> /path/to/php56/bin/phpize -v
Configuring for:
PHP Api Version:         20131106
Zend Module Api No:      20131226
Zend Extension Api No:   220131226
```

Zend Module Api No 本身使用年. 月. 日的日期格式构建。这是 API 更改并打标签的日期。Zend Extension Api No 是 Zend 版本，随后是 Zend Module Api No.

编号太多了吗? 是的。一个 API 号绑定到一个 PHP 版本，对于任何人来说都足够了，并且可以简化对 PHP 版本控制的理解。不幸的是，除了 PHP 版本之外，我们还得到了 3 个不同的 API 号。你应该找哪一个? 答案是任何一个: 当 PHP 版本发展时，它们三个都在发展。由于历史原因，我们得到了 3 个不同的编号。

但是，你是 C 开发人员，不是吗? 为什么不根据这些编号建立一个“兼容性”的头? 我们作者，在我们的扩展中使用这样的东西:

```
#include "php.h"
#include "Zend/zend_extensions.h"

#define PHP_5_5_X_API_NO            220121212
#define PHP_5_6_X_API_NO            220131226

#define PHP_7_0_X_API_NO            320151012
#define PHP_7_1_X_API_NO            320160303
#define PHP_7_2_X_API_NO            320160731

#define IS_PHP_72          ZEND_EXTENSION_API_NO == PHP_7_2_X_API_NO
#define IS_AT_LEAST_PHP_72 ZEND_EXTENSION_API_NO >= PHP_7_2_X_API_NO
```

```
#define IS_PHP_71           ZEND_EXTENSION_API_NO == PHP_7_1_X_API_NO
#define IS_AT_LEAST_PHP_71 ZEND_EXTENSION_API_NO >= PHP_7_1_X_API_NO

#define IS_PHP_70           ZEND_EXTENSION_API_NO == PHP_7_0_X_API_NO
#define IS_AT_LEAST_PHP_70 ZEND_EXTENSION_API_NO >= PHP_7_0_X_API_NO

#define IS_PHP_56           ZEND_EXTENSION_API_NO == PHP_5_6_X_API_NO
#define IS_AT_LEAST_PHP_56 (ZEND_EXTENSION_API_NO >= PHP_5_6_X_API_NO &&
   ZEND_EXTENSION_API_NO < PHP_7_0_X_API_NO)

#define IS_PHP_55           ZEND_EXTENSION_API_NO == PHP_5_5_X_API_NO
#define IS_AT_LEAST_PHP_55 (ZEND_EXTENSION_API_NO >= PHP_5_5_X_API_NO &&
   ZEND_EXTENSION_API_NO < PHP_7_0_X_API_NO)

#if ZEND_EXTENSION_API_NO >= PHP_7_0_X_API_NO
#define IS_PHP_7 1
#define IS_PHP_5 0
#else
#define IS_PHP_7 0
#define IS_PHP_5 1
#endif
```

看到了吗?

或者，更简单 (更好) 的方法是使用你可能更熟悉的 PHP_VERSION_ID:

```
#if PHP_VERSION_ID >= 50600
```

第三章　内部类型

3.1　Zvals

在本章中，我们将详细介绍 PHP 内部使用的特殊类型。其中一些类型直接绑定到用户层 PHP，比如“zval”数据结构。其他结构/类型，如“zend_string”，从用户角度来看并不是真的可见，但是要知道你是否计划从内部编写 PHP 是一个细节。

3.1.1　基本结构

zval（Zend value 的缩写）表示任意的 PHP 值。因此，它可能是 PHP 所有结构中最重要的并且你将会经常使用它。本节介绍 zval 背后的基本概念及其用法。

类型和值

此外，每个 zval 都存储一些值以及该值的类型。这是有必要的，因为 PHP 是一种动态类型语言，而且变量的类型只能在运行时才会知道而不是在编译时。此外，值类型可以在 zval 的生命周期内更改，因此如果 zval 先前存储了一个整数，那么在以后的某个时间点它可能包含一个字符串。

用整数标签存储类型 (unsigned int)。它可以是几个值之一。一些值与 PHP 中可用的八种类型相对应，其他值仅用于内部引擎。使用 **IS_TYPE** 这种形式的常量引用这些值。例如,**IS_NULL** 对应于 null 类型和 **IS_STRING** 对应于字符串类型。

实际值存储在 union 中。union 的定义如下:

```
typedef union _zend_value {
        zend_long         lval;
        double            dval;
        zend_refcounted  *counted;
        zend_string      *str;
        zend_array       *arr;
        zend_object      *obj;
```

```
        zend_resource     *res;
        zend_reference    *ref;
        zend_ast_ref      *ast;
        zval              *zv;
        void              *ptr;
        zend_class_entry  *ce;
        zend_function     *func;
        struct {
            uint32_t w1;
            uint32_t w2;
        } ww;
    } zend_value;
```

写给那些不熟悉联合体 (union) 概念的人：联合体定义了多个不同类型的成员，但是每次只能使用其中的一个成员。例如如果赋值 **value.lval** 成员后，那么访问这个值也是通过 **value.lval** 成员而不是其他成员（这样会违背" 严格别名" 保证并导致未定义行为）。原因是联合体将所以的成员值存储在相同的内存位置，并根据你访问的成员对该位置的值进行不同的解释。联合体的大小就是其最大成员的大小。

在使用 zvals 时，类型标记用于查明当前联合体 (union) 的哪个成员正在使用。在查看一些相关 API 之前，让我们一起详细了解 PHP 支持的不同类型以及它们的存储方式: 最简单的类型是 **IS_NULL**：它实际上不需要存储任何值，因为它只是一个 **null** 值。

为了存储数字 PHP 提供了 **IS_LONG** 和 **IS_DOUBLE** 类型，它们分别使用了 *zend_long lval* 和 *double dval* 成员。前者用于存储整数，而后者存储浮点数。

关于 **zend_long** 类型需要注意一些事情: 首先这是有符号整型类型，即它可以存储正整数和负整数，但通常不太适合做位操作。其次，**zend_long** 相当于平台 long 类型的抽象，所以无论你使用什么平台，**zend_long** 在 32 位平台上的为 4 字节，在 64 位平台上的为 8 字节。

除此之外，你可以使用与 long 类型有关的宏，例如 **SIZEOF_ZEND_LONG** 或者 **ZEND_LONG_MAX**。有关更多信息，请参阅源代码中的 **Zend/zend long.h**。

用于存储浮点数的 **double** 类型 (通常) 是符合 IEEE-754 规范的 8 字节值。这里不会讨论这种格式的细节，但是你至少应该注意这样一个事实: 这种类型的精度有限，通常不能存储你想要的确切值。

布尔类型使用 **IS_TRUE** 或者 **IS_FALSE** 标记并且不需要存储任何其他信息。

存在一种称为"伪类型"标记的 **_IS_BOOL** 类型，但是你不应该将其用作 zval 类型，这是不正确的。这种伪类型用于一些罕见的内部情况 (例如提示类型)。

其余四种类型只会在这里简述，后续会在他们自己的章节中进行更详细的讨论:

字符串 **IS_STRING** 存储在 **zend_string** 结构中，即它们由 *char* * 字符串和 **size_t** 长度组成。您将在string一章中找到关于 **zend_string** 结构及其专用 API 的更多信息。

数组使用 **IS_ARRAY** 类型标记并存储在 **zend_array * arr** 成员中。**HashTable** 结构的工作方式将在Hashtables章节中讨论。

对象 (**IS_OBJECT**) 使用 **zend_object *obj** 成员。PHP 的类和对象系统将在objects一章中描述。

资源 (IS_RESOURCE) 是使用 *zend_resource *res* 成员的特殊类型。资料在Resources 章节中。

综上所述，这是一个包含所有可用类型标记及其值的相应存储位置的表:

Type tag	Storage location
IS_NULL	none
IS_TRUE or IS_FALSE	none
IS_LONG	zend_long lval
IS_DOUBLE	double dval
IS_STRING	zend_string *str
IS_ARRAY	zend_array *arr
IS_OBJECT	zend_object *obj
IS_RESOURCE	zend_resource *res

特殊类型

你可能会看到 zvals 中的其他类型，我们还没有讨论的。这些类型是 PHP 语言用户区中不存在的特殊类型，仅用于内部实例的引擎。zval 结构被认为是非常灵活的，并且在内部用于承载几乎任何类型的有关的数据，而不仅仅是我们上面刚刚讨论过的 PHP 特定类型。

特殊的 **IS_UNDEF** 类型有特殊的含义。这意味着“这个 zval 不包含相关的数据，不要从它访问任何数据字段”。这是用于内存管理的。如果您看到 **IS_UNDEF** zval，这意味着它没有特殊类型，也不包含有效信息。

*zend_refcounted * count* 字段非常难以理解。基本上，该字段用作任何其他引用可计数类型的头。这部分我们将在Zval 内存管理和垃圾收集章节详细介绍。

当你从编译器操作 AST 时将使用 *zend_ast_ref * ast*。Zend 编译器章节将详细介绍 PHP 编译。

*zval * zv* 仅在内部使用。你不应该操纵它。它与 **IS_INDIRECT** 一起使用，允许将 *zval** 嵌入到 **zval** 中。这个字段特殊的隐藏用法是被用做表示 *$GLOBALS[] PHP* 超全局变量。

*void * ptr* 字段非常有用。同样此字段在 PHP 用户空间不可使用，仅限内部。当您想要将“某些东西”存储到 zval 中时，您基本上会使用此字段。是的，这会是一个 *void** 类型，在 C 中表示“指向任何大小的某个内存区域的指针，包含（希望）任何东西”。然后在 zval 中使用 **IS_PTR** 标志类型。

当您阅读对象章节时，您将了解 **zend_class_entry** 类型。*zval 'zend_class_entry * ce* 字段用于将对 PHP 类的引用转换为 zval。同样，在 PHP 语言本身（userland）中没有直接使用这种情况，但在内部你需要它。

最后 *zend_function * func* 字段用于将 PHP 函数嵌入到 zval 中。函数章节详细介绍了 PHP 函数。

访问宏

现在让我们看看 zval 结构实际上是什么样子的:

```
struct _zval_struct {
        zend_value        value;                    /* value */
        union {
                struct {
                        ZEND_ENDIAN_LOHI_4(
                                zend_uchar    type,                  /* active
   type */
                                zend_uchar    type_flags,
                                zend_uchar    const_flags,
                                zend_uchar    reserved)          /* call info
   for EX(This) */
                } v;
                uint32_t type_info;
        } u1;
        union {
                uint32_t     next;                 /* hash collision chain */
                uint32_t     cache_slot;           /* literal cache slot */
                uint32_t     lineno;               /* line number (for ast
   nodes) */
                uint32_t     num_args;             /* arguments number for EX(
   This) */
                uint32_t     fe_pos;               /* foreach position */
                uint32_t     fe_iter_idx;          /* foreach iterator index */
                uint32_t     access_flags;         /* class constant access
   flags */
                uint32_t     property_guard;       /* single property guard */
                uint32_t     extra;                /* not further specified */
        } u2;
};
```

如上所述，zval 存储着 **value** 和 **type_info** 字段。value 存储在上面讨论的 union 的 **zvalue_value** 字段中，类型标记存储在 **u1** union 的 zend_uchar 字段中。此外，该结构还有 u2 字段。我们暂时忽略它们，稍后再讨论它们的功能。通过 **type_info** 访问 **u1**。**type_info** 分为 type、type_flags、const_flags 和 reservation 字段。记住在 **u1** 联合体中，**u1.v** 四个字段的值大小和存储在 **u1.type_info** 中的值相同。这里使用了一个聪明的内存对齐规则。**u1** 非常常用，因为它将关于存储在 zval 中的类型的信息嵌入其中。

u2 有完全不同的含义。我们现在不需要详细说明 **u2** 字段，你可以无视它，我们稍后再看。

了解 zval 结构后，你现在可以编写使用它的代码：

```
zval zv_ptr = /* ... get zval from somewhere */;

if (zv_ptr->type == IS_LONG) {
    php_printf("Zval is a long with value %ld\n", zv_ptr->value.lval);
} else /* ... handle other types */
```

此代码应该为 php5 的写法

虽然上面的代码有效，但这种写法不常用。它直接访问 zval 成员，而不是通过一组特殊的访问宏。

```
        zval *zv_ptr = /* ... */;

        if (Z_TYPE_P(zv_ptr) == IS_LONG) {
            php_printf("Zval is a long with value %ld\n", Z_LVAL_P(zv_ptr));
        } else /* ... */
```

上面的代码使用 **Z_TYPE_P()** 宏来检索类型标记，使用 **Z_LVAL_P()** 宏来获取 long(整型) 的值。所有访问宏都有 **_P** 后缀或根本没有后缀的变体。使用哪一个取决于您是使用 **zval** 还是 **zval***

```
        zval zv;
        zval *zv_ptr;
        zval **zv_ptr_ptr; /* very rare */

        Z_TYPE(zv);                 // = zv.type
        Z_TYPE_P(zv_ptr);           // = zv_ptr->type
```

此代码应该为 php5 的写法

P 表示"pointer" 指针的意思。这种只适用于 *zval**。没有特殊的宏来处理 *zval*** 或更多的的指针的指针，因为在实践中很少需要这样的宏 (您只需要在使用 * 操作符解引用该值)。

与 *Z_LVAL* 类似，还有一些宏用于获取所有其他类型的值。为了演示它们的用法，我们将创建一个简单的函数来转储 zval:

```
        PHP_FUNCTION(dump)
        {
```

```
        zval *zv_ptr;

        if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zv_ptr) == FAILURE
) {
            return;
        }

        switch (Z_TYPE_P(zv_ptr)) {
            case IS_NULL:
                php_printf("NULL: null\n");
                break;
            case IS_TRUE:
                php_printf("BOOL: true\n");
                break;
            case IS_FALSE:
                php_printf("BOOL: false\n");
                break;
            case IS_LONG:
                php_printf("LONG: %ld\n", Z_LVAL_P(zv_ptr));
                break;
            case IS_DOUBLE:
                php_printf("DOUBLE: %g\n", Z_DVAL_P(zv_ptr));
                break;
            case IS_STRING:
                php_printf("STRING: value=\"");
                PHPWRITE(Z_STRVAL_P(zv_ptr), Z_STRLEN_P(zv_ptr));
                php_printf("\", length=%zd\n", Z_STRLEN_P(zv_ptr));
                break;
            case IS_RESOURCE:
                php_printf("RESOURCE: id=%d\n", Z_RES_HANDLE_P(zv_ptr));
                break;
            case IS_ARRAY:
                php_printf("ARRAY: hashtable=%p\n", Z_ARRVAL_P(zv_ptr));
                break;
            case IS_OBJECT:
                php_printf("OBJECT: object=%p\n", Z_OBJ_P(zv_ptr));
                break;
        }
    }
```

```
const zend_function_entry funcs[] = {
    PHP_FE(dump, NULL)
    PHP_FE_END
};
```

让我们试一试:

```
dump(null);                 // NULL: null
dump(true);                 // BOOL: true
dump(false);                // BOOL: false
dump(42);                   // LONG: 42
dump(4.2);                  // DOUBLE: 4.2
dump("foo");                // STRING: value="foo", length=3
dump(fopen(__FILE__, "r")); // RESOURCE: id=???
dump(array(1, 2, 3));       // ARRAY: hashtable=0x???
dump(new stdClass);         // OBJECT: object=0x???
```

访问值的宏非常简单：**Z_LVAL** 对应 long 类型，**Z_DVAL** 对应双精度 double 类型。对于字符串，**Z_STR** 实际上返回 *zend_string* * 类型字符串，**ZSTR_VAL** 返回 *char* * 类型字符串，而 **Z_STRLEN** 可以获取其长度。可以使用 **Z_RES_HANDLE** 获取资源 ID，并使用 **Z_ARRVAL** 访问数组的 *zend_array* *。

当你想要访问 zval 内容时，你每次都应该通过这些宏，而不是直接访问其成员。这保持了一定程度的抽象，使意图更加清晰。使用宏可以在将来 php 版本中内部 zval 表示法更改时起保护作用。

赋值

上面介绍的大多数宏只是访问 zval 结构的某些成员，同样的你可以使用它们来读取和写入相应的值。作为一个例子，考虑以下函数，它只返回字符串“hello world!”:

```
PHP_FUNCTION(hello_world) {
        Z_TYPE_P(return_value) = IS_STRING;
        Z_STR_P(return_value) = zend_string_init("hello world!", strlen
("hello world!"), 0);
    };

    /* ... */
        PHP_FE(hello_world, NULL)
    /* ... */
```

运行 *php -r "echo hello_world();"* 将在终端输出 **hello world!**。

在上面的例子中，我们设置了 **return_value** 变量，它是由 **PHP_FUNCTION** 宏提供的 *zval* * 类型。下一章我

们将更详细地讨论这个变量，现在只要知道这个变量的值就是函数的返回值就足够了。默认情况下，初始化为 **IS_NULL** 类型。

使用访问宏设置 zval 值非常简单，但是有一些事情需要记住: 首先，你需要记住类型标记决定 zval 的类型。仅仅设置值是不够的 (通过 *Z_STR_P*)，还需要设置类型标记。

此外你需要知道在大多数情况下,zval“拥有”其值，并且 zval 的生命周期比你设置它的值的作用域更长。有时这在处理临时 zvals 时并不适用，但在大多数情况下是正确的。

使用上面的例子，这意味着在我们离开函数体之后，**return_value** 将继续存在（这很明显，否则没有人可以使用返回值），因此它不能使用函数的任何临时值。

因此我们需要使用 *zend_string_init()* 创建一个新的 *zend_string*。这将在堆上创建此字符串的单独副本。因为 zval“承载”它的值，所以当 zval 被销毁时，它将被确保释放该副本或者减少引用计数。这也适用于 zval 的任何其他“复杂”值。例如，如果你为一个数组设置 *zend_array*,zval 以后将会” 携带” 它，并在 zval 销毁时释放它。对于释放的意思是，要么减少引用计数，要么当引用计数变为 0 时释放次结构。当使用像整型或者双精度基本类型时, 显然不需要关心这些，因为它们总是被复制的。所有这些内存管理步骤，例如分配，释放或引用计数; 将在memory_and_gc章节详细介绍。

zval 赋值是一项非常常见的操作，PHP 为此提供了另一组宏。它们允许你同时设置类型标记和值。使用这样的宏重新编写前面的示例会得到如下结果:

```
PHP_FUNCTION(hello_world) {
    ZVAL_STRINGL(return_value, "hello world!", strlen("hello world!"));
}
```

此外，我们不需要手工计算 strlen，可以使用 *ZVAL_STRING* 宏 (末尾没有 L):

```
PHP_FUNCTION(hello_world) {
    ZVAL_STRING(return_value, "hello world!");
}
```

如果您知道字符串的长度 (因为它是以某种方式传递给你的)，那么你应该始终通过 *ZVAL_STRINGL* 宏来使用它，以保持二进制安全。如果不知道长度 (或者不知道字符串不包含 NUL 字节，就像通常的字面意思一样)，可以使用 *ZVAL_STRING*。

除了 *ZVAL_STRING(L)* 之外，还有一些用于设置值的宏，如下例所示:

```
ZVAL_NULL(return_value);

ZVAL_FALSE(return_value);
ZVAL_TRUE(return_value);

ZVAL_LONG(return_value, 42);
ZVAL_DOUBLE(return_value, 4.2);
```

```
ZVAL_RES(return_value, zend_resource *);

ZVAL_EMPTY_STRING(return_value);
/* a special way to manage the "" empty string */

ZVAL_STRING(return_value, "string");
/* = ZVAL_NEW_STR(z, zend_string_init("string", strlen("string"), 0)); */

ZVAL_STRINGL(return_value, "nul\0string", 10);
/* = ZVAL_NEW_STR(z, zend_string_init("nul\0string", 10, 0)); */
```

请注意，这些宏将赋值，但不会销毁 zval 之前可能保留的任何值。对于 *return_value* zval，这无关紧要，因为它已初始化为 *IS_NULL*（没有需要释放的值），但在其他情况下，您必须首先使用下一节中描述的函数销毁旧值。

3.1.2 Zval 内存管理和垃圾收集

3.1.3 zvals 里的强制类型转换和操作

3.2 字符串管理

任何程序都需要管理字符串。管理就像分配、搜索、连接、扩展、收缩等等。

字符串需要许多操作。尽管 C 标准库为这一目标提供了许多功能，但 C 经典字符串，即 *char **（或 *char []*）在 PHP 这样的强程序中按原样使用通常有点弱。

因此，PHP 在 C 字符串之上设计了一个层:**zend_strings**。另外还存在另一个 API，它实现了 C 经典字符串或 **zend_strings** 常见的字符串操作: **smart_str** API。

3.2.1 字符串管理:zend_string

它增加了内存管理功能，以便相同字符串可以在多个地方共享，而不需要复制它。另外一些字符串被“扣押”，即它们被“持久”分配并由内存管理器专门管理，因此它们不会在多个请求中被销毁。这些内存稍后将从Zend内存管理器获得永久分配。

结构和访问宏

下面是 **zend_string** 结构:

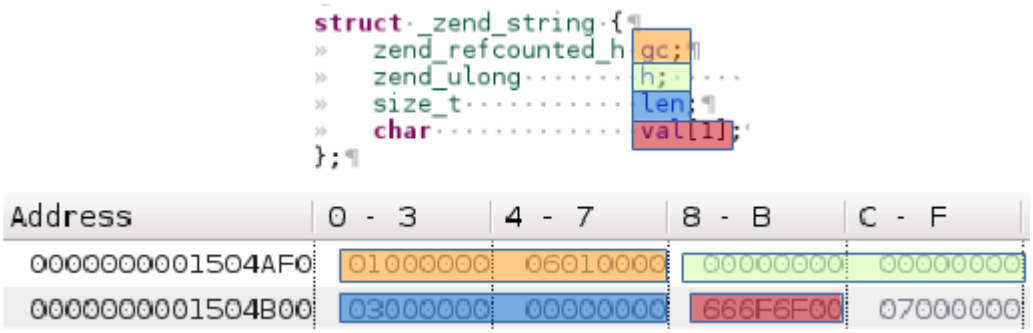
```
struct _zend_string {
        zend_refcounted_h gc;           /*gc信息*/
        zend_ulong        h;            /* hash value */
        size_t            len;          /*字符串长度*/
        char              val[1];       /*字符串起始地址*/
};
```

如您所见，该结构嵌入了 *zend_refcounted_h* 头。这样做是为了内存管理和引用。由于字符串很可能用作哈希表查询的键，所以它将其哈希值嵌入到 **h** 字段中。这是一个无符号 long *zend_ulong*。此数字仅在需要对 *zend_string* 进行哈希处理时使用，尤其是与HashTables zend_array一起使用时; 这很可能。

如您所知, 可以通过 **len** 字段知道字符串的长度，用来支持“二进制字符串”。二进制字符串是嵌入一个或多个 **NUL** 字符（\0）的字符串。当传递给 libc 函数时，这些字符串将被截断，否则它们的长度将无法正确计算。所以在 **zend_string** 中，字符串的长度总是已知的。请注意，计算 ASCII 字符 (字节) 数量的长度，不计算结尾的 **NUL**，但是计算可能在中间的 NULs。例如，字符串“foo”在 **zend_string** 中存储为“foo\0”，其长度为 3。此外，字符串“foo\0bar”将存储为“foo\0bar\0”，长度将为 7。

最后，字符存储在 *char [1]* 字段中。这不是一个 *char **，而是一个 *char[1]*。为什么？这是一种名为“C struct hack”的内存优化。(你可以搜索一下这些术语)。基本上，它允许引擎为 *zend_string* 结构和要存储的字符分配空间，作为一个单独的 C 指针。这优化了内存访问, 因为内存将是一个连续分配的块，而不是内存中稀疏的两个块（一个用于 *zend_string **，一个用于存储到其中的 *char **）。

struct hack 一定要记住，因为内存布局看起来像 C 字符位于 C *zend_string* 结构的末尾，在使用 C 调试器 (或在调试字符串时) 时可能会感觉到/看到。这个 hack 完全由你在操作 *zend_string* 结构时使用的 API 管理。



使用 *zend_string* API

简单实例

与 **Zvals** 一样，您不要手工操作 *zend_string* 内部字段，而是始终要使用宏。还有一些宏可以触发字符串上的操作。这些不是函数，而是宏，都存储在所需的Zend/zend_string.h头:

```
zend_string *str;

str = zend_string_init("foo", strlen("foo"), 0);
php_printf("This is my string: %s\n", ZSTR_VAL(str));
php_printf("It is %zd char long\n", ZSTR_LEN(str));

zend_string_release(str);
```

上面的简单示例展示了基本的字符串管理。

zend_string_init() 函数 (它实际上是宏，但是让我们传入这些细节) 应当传入完整 *char* * 型 C 字符串及其长度。最后一个参数 (类型为 int) 值应该是 0 或 1。如果传 0，则要求引擎使用 Zend 内存管理器使用请求绑定堆分配。这种分配将在当前请求结束时销毁。如果您不这样做，在调试构建时，引擎将会对您刚刚创建的发出内存泄漏的警告。如果传 1，您将请求我们所说的“持久”分配，即引擎将调用传统的 C *malloc()* 方法，并且不会以任何方式跟踪内存分配。

如果您想了解有关内存管理的更多信息，可以阅读专门章节。

然后，我们输出字符串。我们使用 *ZSTR_VAL()* 宏访问字符数组。*ZSTR_LEN()* 获取长度信息。*zend_string* 相关的宏都以 *ZSTR_**()* 开头，请注意这与 *Z_STR**()* 宏不同。

长度使用 *size_t* 类型存储。因此为了输出它 *printf()* 需要使用"%zd"。您应该始终使用正确的 *printf()* 格式。如果不这样做可能会导致应用程序崩溃或产生安全问题。有关 printf() 格式的精彩回忆，请访问此链接

最后，我们使用 *zend_string_release()* 释放字符串。这个" 释放" 是强制性的。这是关于内存管理的。"释放"是一个简单的操作：减少字符串的引用计数器，如果它变为 0，API 将为你释放字符串。如果忘记释放字符串，很可能会造成内存泄漏。

您必须始终考虑 C 语言中的内存管理。如果您分配内存-无论是直接使用 *malloc()*，还是使用 API 来完成，你必须在某个时刻进行 *free()*。如果不这样做会造成内存泄露并会变成没有人能够安全使用的设计糟糕的程序

使用 *hash*

如果需要访问 hash 值，请使用 *ZSTR_H()*。但是，创建 *zend_string* 时不会自动计算哈希值。但是当使用 HashTable API 操作字符串时，它将会完成计算。如果你想强制立即计算 hash 值，可以使用 *ZSTR_HASH()* 或者 *zend_string_hash_val()*。一旦计算出 hash 值，它将会被保存并且不会再次被计算。如果处于某种原因，您需要重新计算它 - 例如：因为你更改了字符串的值 - 使用 *zend_string_forget_hash_val()*:

```
zend_string *str;

str = zend_string_init("foo", strlen("foo"), 0);
php_printf("This is my string: %s\n", ZSTR_VAL(str));
php_printf("It is %zd char long\n", ZSTR_LEN(str));

zend_string_hash_val(str);
php_printf("The string hash is %lu\n", ZSTR_H(str));

zend_string_forget_hash_val(str);
php_printf("The string hash is now cleared back to 0!");

zend_string_release(str);
```

字符串复制及内存管理

zend_string API 的一个非常好的特性是它允许一部分通过简单地声明对它的兴趣来"拥有"一个字符串。然后引擎不会在内存中复制字符串，而只是递增其引用计数（作为其 *zend_refcounted_h* 的一部分）。这就在代码中实现了内存共享。

那样的话，当我们说"复制"（copying）*zend_string* 时，事实上，我们不会在内存中复制任何东西。如果需要 -

这仍然是一个可能的操作 - 我们然后讨论“复制”（duplicating）字符串。我们开始吧：

```
    zend_string *foo, *bar, *bar2, *baz;

    foo = zend_string_init("foo", strlen("foo"), 0); /* creates the "foo"
string in foo */
    bar = zend_string_init("bar", strlen("bar"), 0); /* creates the "bar"
string in bar */

    /* creates bar2 and shares the "bar" string from bar into bar2. Also
increments the refcount of the "bar" string to 2 */
    bar2 = zend_string_copy(bar);

    php_printf("We just copied two strings\n");
    php_printf("See : bar content : %s, bar2 content : %s\n", ZSTR_VAL(bar)
, ZSTR_VAL(bar2));

    /* Duplicate in memory the "bar" string, create the baz variable and
make it solo owner of the newly created "bar" string */
    baz = zend_string_dup(bar, 0);

    php_printf("We just duplicated 'bar' in 'baz'\n");
    php_printf("Now we are free to change 'baz' without fearing to change '
bar'\n");

    /* Change the last char of the second "bar" string turning it to "baz"
*/
    ZSTR_VAL(baz)[ZSTR_LEN(baz) - 1] = 'z';

    /* Forget the old hash (if computed) as now the string changed, thus
its hash must also change and get recomputed */
    zend_string_forget_hash_val(baz);

    php_printf("'baz' content is now %s\n", ZSTR_VAL(baz));

    zend_string_release(foo);  /* destroys (frees) the "foo" string */
    zend_string_release(bar);  /* decrements the refcount of the "bar"
string to one */
    zend_string_release(bar2); /* destroys (frees) the "bar" string both in
 bar and bar2 vars */
```

```
        zend_string_release(baz);  /* destroys (frees) the "baz" string */
```

我从分配”foo” 和”bar“开始。然后我们创建了作为 *bar* 副本的 *bar2* 字符串。在这里，我们要知道：*bar* 和 *bar2* 在内存中指向相同的 C 字符串，修改其中一个字符串就会改变第二个。这就是 *zend_string_copy()* 的行为: 它只增加所拥有的 C 字符串的 refcount 值。

如果我们想分开这两个字符串 - 即我们希望那个字符串在内存中有两个不同的副本 - 我们需要使用 *zend_string_dup()* 复制。然后我们将 bar2 变量的字符串复制到 baz 变量中。现在，baz 变量嵌入了自己的字符串副本，并且可以在不影响 bar2 的情况下更改它。这就是我们所做的: 我们将 bar 中的最后一个 r 用 z 替换为 baz。然后我们打印他们，并释放每个字符串的内存。

请注意我们忘记了 hash 值（我们如果之前计算过它，则无需考虑该细节）。这是一个值得记住的好习惯。如前所述，如果 *zend_string* 被用做 HashTables 的一部分，hash 则被使用了。这在开发中是非常常见的操作，更改字符串值也需要重新计算 hash 值。忘记这一步将导致可能花费一些时间来跟踪的 bug。

字符串操作

zend_string API 允许其他操作，例如扩展或缩小字符串，更改其大小写或比较。目前还没有可用的联合操作，但这很容易执行:

```
        zend_string *FOO, *bar, *foobar, *foo_lc;

        FOO = zend_string_init("FOO", strlen("FOO"), 0);
        bar = zend_string_init("bar", strlen("bar"), 0);

        /* Compares a zend_string against a C string literal */
        if (!zend_string_equals_literal(FOO, "foobar")) {
            foobar = zend_string_copy(FOO);

            /* reallocates the C string to a larger buffer */
            foobar = zend_string_extend(foobar, strlen("foobar"), 0);

            /* concatenates "bar" after the newly reallocated large enough "FOO
    " */
            memcpy(ZSTR_VAL(foobar) + ZSTR_LEN(FOO), ZSTR_VAL(bar), ZSTR_LEN(
    bar));
        }

        php_printf("This is my new string: %s\n", ZSTR_VAL(foobar));

        /* Compares two zend_string together */
        if (!zend_string_equals(FOO, foobar)) {
            /* duplicates a string and lowers it */
```

```
        foo_lc = zend_string_tolower(foo);
    }

    php_printf("This is FOO in lower-case: %s\n", ZSTR_VAL(foo_lc));

    /* frees memory */
    zend_string_release(FOO);
    zend_string_release(bar);
    zend_string_release(foobar);
    zend_string_release(foo_lc);
```

使用 *zvals* 访问 *zend_string*

既然您已经知道如何管理和操作 *zend_string*，那么让我们看看它们与 zval 容器之间的交互。

注意：你需要熟悉 zvals，如果不熟悉，请阅读:zvals 一章。这些宏允许您将 *zend_string* 存储到 zval 中，或者从 zval 读取 *zend_string*：

```
    zval myval;
    zend_string *hello, *world;

    zend_string_init(hello, "hello", strlen("hello"), 0);

    /* Stores the string into the zval */
    ZVAL_STR(&myval, hello);

    /* Reads the C string, from the zend_string from the zval */
    php_printf("The string is %s", Z_STRVAL(myval));

    zend_string_init(world, "world", strlen("world"), 0);

    /* Changes the zend_string into myval : replaces it by another one */
    Z_STR(myval) = world;

/* ... */
```

您必须记住的是，以 *ZSTR_***(s)* 开头的每个宏都将作用于 *zend_string*。

- ZSTR_VAL()
- ZSTR_LEN()
- ZSTR_HASH()

- ...

以 *Z_STR**(z)* 开头的每个宏都将作用于嵌入到 zval 中的 *zend_string*

- Z_STRVAL()
- Z_STRLEN()
- Z_STRHASH()
- ...

其他存在的一些你可能用不到。

***PHP* 的历史和经典的 *C* 字符串**

简单介绍一下经典的 C 字符串。在 C 语言中，字符串是字符数组 (char foo[]) 或者是指向字符的指针 (char *)。他们对自己的长度一无所知, 这就是以 NUL 终止的原因 (知道字符串的开始和结束，你就知道了它的长度)。

在 PHP 7 之前，*zend_string* 结构根本不存在。老版本使用了传统的 char * / int 对。你仍然可以在 PHP 源代码中找到很少的地方，还在使用 char */ int 对而不是 *zend_string*。你还可以找到 API 工具，在 *zend_string* 和 char * / int 对之间进行交互。

尽可能：使用 *zend_string*。一些罕见的地方没有使用 *zend_string*，因为在那个地方使用它们是无关紧要的，但是你会在 PHP 源代码中找很多对 *zend_string* 的引用。

Interned zend_string

在这里简单介绍一下 interned 字符串。在扩展开发中，您应该很少会需要这样的概念。Interned 字符串还与 OPCache 扩展有交互。

Interned 字符串是重复数据删除的字符串. 当使用 OPCache 时，它们从一个请求到另一个请求还会被重复使用。

假设您要创建字符串“foo”。你倾向于做的只是创建一个新的字符串“foo”:

```
zend_string *foo;
foo = zend_string_init("foo", strlen("foo"), 0);

/* ... */
```

但是问题来了：那部分字符串是不是在你使用之前就已经创建完成？当您需要一个字符串时，你的代码会在 PHP 生命中的某个时刻执行，这意味着在您之前发生的一些代码可能需要完全相同的字符串 (以“foo”为例)。

interned 字符串是关于要求引擎探测 interned 字符串储存区，如果它可以找到你的字符串，就可以重用已分配的指针。如果不存在：创建一个新的字符串并且“intern”它，这使得它可以用于 PHP 源代码的其他部分 (其他扩展、引擎本身等等)。

下面是一个例子:

```
zend_string *foo;
foo = zend_string_init("foo", strlen("foo"), 0);

foo = zend_new_interned_string(foo);

php_printf("This string is interned : %s", ZSTR_VAL(foo));

zend_string_release(foo);
```

在上面的代码中，我们创建了一个非常经典的 *zend_string*。然后，我们将创建的 *zend_string* 传递给 *zend_new_interned_string()*。这个函数在引擎内的 interned 字符串缓冲区中寻找相同的字符串 (这里是“foo”)。如果找到了它 (意味着有人已经创建了这个字符串)，那么它将释放您的字符串 (可能释放它)，并将用来自 interned 字符串缓冲区的字符串替换它。如果没有查到它：会将它添加到 interned 字符串缓冲区中，因此可供将来使用或 PHP 的其他部分使用。

你必须注意内存分配。interned 字符总是将 refcount 设置为 1，因为他们不需要被计数，因为它们将与 interned 字符串缓冲区共享，因此不能被销毁。

例:

```
zend_string *foo, *foo2;

foo  = zend_string_init("foo", strlen("foo"), 0);
foo2 = zend_string_copy(foo); /* increments refcount of foo */

/* refcount falls back to 1, even if the string is now
* used at three different places */
foo = zend_new_interned_string(foo);

/* This doesn't do anything, as foo is interned */
zend_string_release(foo);

/* This doesn't do anything, as foo2 is interned */
zend_string_release(foo2);
```

```
        /* At the end of the process, PHP will purge its interned
        string buffer, and thus free() our "foo" string itself */
```

这都是关于垃圾回收的。

当一个字符串是 interned 时，它的 GC 标志通过添加 *IS_STR_INTERNED* 标志来改变，无论他们使用什么内存分配类（永久的或基于请求的）。当你想复制或者释放一个字符串时会查询这个标志。如果这个字符串是 interned，引擎在复制字符串时不会增加它的 refcount。但是如果你释放（release）字符串时，它也不会减少或者释放（free）它。它什么也不会做。在进程生命周期结束时，它将销毁其 interned 字符串缓冲区，并释放 interned 字符串。

实际上，如果 OPCache 启动，这个过程会比这个要复杂一点。OPCache 扩展改变了 interned 字符串的使用方式。在没有 OPCache 的情况下，如果您在请求的过程中创建了一个 interned *zend_string*，那么该字符串将在当前请求的末尾被清除，并且不会被下一次请求重用。但是，如果您使用 OPCache，那么 interned 字符串将存储在一个共享内存段中，并在同一个池中的每个 PHP 进程之间共享。此外，interned 字符串会跨多个请求重用。

interned 字符串可以节省内存，因为相同的字符串在内存中存储的次数不会超过一次。当时它可能会耗一些 CPU 时间，因为它经常需要查找 interned 字符串存储，即使该进程已经过优化。作为扩展开发者，以下是全局规则：

- 如果使用 OPCache（应该是这样），并且如果您需要创建请求绑定的只读字符串：使用 interned 字符串
- 如果您需要一个字符串，您确信知道 PHP 已经 interned 的字符串（一个众所周知的 PHP 字符串，例如“php”或者“str_replace”），请使用一个 interned 字符串
- 如果字符串不是只读的，并且在创建字符串后可以/应该进行更改，则不要使用 interned 字符串
- 如果将来不太可能重用该字符串，则不要使用 interned 字符串

Interned 字符串的详细介绍在Zend/zend_string.c

3.2.2 smart_str API

这看起来可能很奇怪，C 语言几乎没有提供任何字符串处理功能 (构建、连接、收缩、展开、转换等等)。C 是一种低级通用语言，可用于构建 API 以处理更具体的任务，例如字符串构造。

显然，你们都知道我们讨论的是 ASCII 字符串，也就是字节。而不是 Unicode。

PHP 的 ‘smart_str‘ API 可以帮助你构建字符串特别是将字节块连接为字符串。这些 API 位于:PHP’s special printf() APIs 和zend_string 用来帮助字符串管理。

smart_str VS smart_string

下面是两种结构体:

```
typedef struct {
    char *c;
    size_t len;
    size_t a;
} smart_string;

typedef struct {
    zend_string *s;
    size_t a;
} smart_str;
```

如您所见，一个将使用传统的 C 字符串 (如 *char*/size_t*)，另一个将使用 PHP 的特定 *zend_string* 结构。

我们将详细说明后者：*smart_str*, 它与zend_strings一起使用。这两个 API 完全相同，只需注意一个（我们将在这里详述的那个）由 *smart_str_**()* 开头，另一个由 *smart_string_***()* 开头。不要混淆!

smart_str API 详细介绍在Zend/zend_smart_str.h（也是.c 文件）。

不要将 *smart_str* 与 *smart_string* 混淆。

基本 API 使用

到目前为止，这个 API 非常容易管理。

您在堆栈分配一个 *smart_str*，并将其指针传递给 *smart_str_***()*API 函数，它为您管理内含的 *zend_string*。你构建你的字符串，使用它，然后释放它。里面没什么强大的东西，对吗?

内含的 *zend_string* 将要被永久的或者请求绑定的分配，这取决于您将使用的最后一个扩展 API 参数:

```
    smart_str my_str = {0};

    smart_str_appends(&my_str, "Hello, you are using PHP version ");
    smart_str_appends(&my_str, PHP_VERSION);

    smart_str_appendc(&my_str, '\n');
```

```
        smart_str_appends(&my_str, "You are using ");
        smart_str_append_unsigned(&my_str, zend_hash_num_elements(CG(
    function_table)));
        smart_str_appends(&my_str, " PHP functions");

        smart_str_0(&my_str);

        /* Use my_str now */
        PHPWRITE(ZSTR_VAL(my_str.s), ZSTR_LEN(my_str.s));

        /* Don't forget to release/free it */
        smart_str_free(&my_str);
```

我们在这里使用了简单的 API，扩展以 *_ex()* 为结尾，允许你选择 **zend_string** 是持久或者是请求绑定时分配。例如：

```
        smart_str my_str = {0};

        smart_str_appends_ex(&my_str, "Hello world", 1); /* 1 means persistent
    allocation */
```

然后，根据你想要追加的内容，你需要调用正确的 API。如果你追加一个经典的 C 字符串，你可以使用 *smart_str_appends(smart_str *dst, const char *src)*。如果你追加二进制字符串并且知道它的长度，你可以使用 *smart_str_appendl(smart_str *dst, const char *src, size_t len)*。

*smart_str_append(smart_str *dest, const zend_string *src)* 只是讲 *zend_string* 追加到 *smart_str*。如果你想追加其他的 *smart_str*, 使用 *smart_str_append_smart_str(smart_str *dst, const smart_str *src)* 来合并它们。

smart_str 特定技巧

- 永远不要忘记通过调用 *smart_str_0()* 来完成你的字符串。此操作会在内含的字符串末尾追加一个 NUL 字符，会使其用 libc 字符串函数兼容
- 永远不要忘记在使用完成字符串后通过 *smart_str_free()* 来释放 (free)。
- *smart_str* 内含一个 *zend_string*, 允许你通过使用其 reference 计数在后面的其他地方共享它。请访问zend_string 专用章节了解更多信息。
- 你可以考虑一下 *smart_str* 分配。了解一下 *smart_str_alloc()* 系列。
- *smart_str* 被大量的用于 PHP 的核心代码。例如：PHP 的特定函数printf()在内部使用了 *smart_str* 缓冲区。
- *smart_str* 绝对是一个您需要掌握的简单结构。

3.2.3 PHP 自定义的 printf 函数

你们都知道 libc 的 *printf()* 及其相关函数。本章将详细介绍 PHP 声明和使用的许多克隆，它们的目标是什么，为什么使用它们以及何时使用它们。

关于 *printf()* 以及相关函数的 Libc 文档在这里

您知道这些函数很有用, 但有时没有提供足够的功能。此外你知道向 *printf()* 相关函数添加格式化字符串是不容易的，不可移植的，并且有安全风险。

PHP 添加了自己的类 printf 函数来替代 libc 的函数，并被内部开发人员使用。他们主要添加了新格式，并用 *zend_string* 代替 char * 等等，让我们一起来看一下。

您必须掌握 libc *printf()* 的默认格式。请阅读这个文档

添加这些函数是为了替换 libc 函数，这意味着例如如果你使用 *sprintf()*, 这不会调用 libc 的 *sprintf()*, 而会调用 php 的替代函数。除了常用的 *printf()* 之外, 其他的函数都被替换了。

传统用法

首先，您不应该使用 *sprintf()*，因为该函数不执行任何检查，并且容许许多缓冲区溢出错误的出现。请尽量避免使用它。

请尽量避免使用 *sprintf()*。

然后，你有一些选择。

已知缓存大小

如果您知道缓存的大小，可以使用 *snprintf()* 或 *slprintf()* 来完成这项工作。这些函数的返回值不同，但是功能是相同的。

他们都根据传入的格式进行打印，并且总是以 NUL 字符'
0' 为结尾。然而 *snprintf()* 返回传入的字符数，而 *slprintf()* 返回有效使用的字符数。这因此能够检测缓存区太小和字符串截断。这个是不包括最后的'
0'。

下面是一个例子，可以让你充分理解:

```
        char foo[8];                                /* 8-char large buffer */
        const char str[] = "Hello world";           /* 12 chars including \0 in
    count */
        int r;

        r = snprintf(foo, sizeof(foo), "%s", str);
        /* r = 11 here even if only 7 printable chars were written in foo */

        /* foo value is now 'H' 'e' 'l' 'l' 'o' ' ' 'w' '\0' */
```

对使用 *snprintf()* 来说，它不是一个很好的函数，因为它不允许检测最终的字符串截断。从上面的例子中你可以很明显的看出，"Hello world
0" 不能被容纳到 8 字节的缓存中。但是 *snprintf()* 仍然返回 11，也就是 *strlen("Hello world
0")* 的值。所以你无法检测到字符串被截断了。

下面是 *slprintf()*:

```
        char foo[8];                                /* 8-char large buffer */
        const char str[] = "Hello world";           /* 12 chars including \0 in
    count */
        int r;

        r = slprintf(foo, sizeof(foo), "%s", str);
        /* r = 7 here , because 7 printable chars were written in foo */

        /* foo value is now 'H' 'e' 'l' 'l' 'o' ' ' 'w' '\0' */
```

对于 *slprintf()*, **foo** 的结果缓存中包含完全相同的字符串，但是现在的返回值为 7。7 小于"Hello world" 字符串的 11 字符，因此你可以发现它被截断了:

```
        if (slprintf(foo, sizeof(foo), "%s", str) < strlen(str)) {
                /* A string truncation occurred */
        }
```

记住:

- 字符串无论是否截断，这两个函数总是以 NUL 终止。结果字符串是安全的 C 字符串。
- 只有 *slprintf()* 允许检测字符串截断。

这两个函数定义在main/snprintf.c

缓存大小未知

现在如果你不知道结果缓存的大小，那么你需要一个动态分配的缓存，然后你使用 *spprintf()*。记住，你必须自己释放缓存!

下面是个例子:

```
    #include <time.h>

    char *result;
    int r;

    time_t timestamp = time(NULL);

    r = spprintf(&result, 0, "Here is the date: %s", asctime(localtime(&
timestamp)));

    /* now use result that contains something like "Here is the date: Thu
Jun 15 19:12:51 2017\n" */

    efree(result);
```

spprintf() 返回已被打印到结果缓存里的字符数，不包括最后'
0'。因此你知道为你分配的字节数 (减掉 1)。

请注意，是使用 ZendMM 完成分配的（请求分配），因此应该作为请求的一部分使用并且使用 *efree()* 而不是 *free()* 来释放缓存。

关于Zend 内存管理器(ZendMM) 的一章详细介绍了如何通过 PHP 动态分配内存。

如果你想要限制缓存的大小，可以将此限制值作为函数的第二个参数传入，如果你传 0，则表示没有限制:

```
    #include <time.h>
```

```
    char *result;
    int r;

    time_t timestamp = time(NULL);

    /* Do not print more than 10 bytes || allocate more than 11 bytes */
    r = spprintf(&result, 10, "Here is the date: %s", asctime(localtime(&
timestamp)));

    /* r == 10 here, and 11 bytes were allocated into result */

    efree(result);
```

尽可能不要使用动态内存分配，这会影响性能。如果你可以选择，使用静态堆栈分配缓存。

spprintf() 定义在main/spprintf.c。

printf()

你需要使用 *printf()*, 也就是格式化输出到输出流，请使用 *php_printf()*。该函数内部使用 *spprintf()*，因此执行动态分配，在将其发送到 SAPI 输出流，也就是 CLI 模式下的 stdout 或者其他 SAPI 的输出缓存区 (例如：CGI 缓冲区) 后释放自己。

特殊的 PHP printf 格式

记住 PHP 使用了自己设计的函数替代了大多数 libc 的 *printf()* 函数。你可以看一下参数解析 API，通过阅读源码你可以很容易理解。

这意味着参数解析算法已经被完全重新，并且可能与你使用的 libc 习惯不一样。例如：在大多数情况下，libc 本地化处理的不好。你还可以使用一些特殊格式，例如："%I64"显式打印 int64, 或者"%I32"。还可以使用"%Z"打印 zval（根据 PHP 转换规则转为字符串），这是一个很好的补充。

格式化程序还将识别无限大的数字，并打印为"INF"，或非数字表示为"NAN"。

如果你犯了一个错误，并要求格式化输出一个 NULL 指针，libc 肯定会崩溃，PHP 将返回"(NULL)"字符串。

如果你在 printf 中看到一个神奇的“(null)”出现，这意味着你向 PHP printf 系列函数传递了一个空指针。

Printf()ing into zend_strings

由于zend_string是 PHP 源代码中非常常见的结构，因此您可能需要将 *printf()* 转换为 *zend_string* 而不是传统的 C *char* *。为此，请使用 *strpprintf()*。

API 是 *zend_string *strpprintf(size_t max_len, const char *format, ...)*, 其返回值是 *zend_string*, 而不是你所期望的打印字符的数量。你可以使用第一个参数 (传 0 表示无限的) 来限制这个数字; 你必须记住，*zend_string* 将使用 Zend 内存管理器分配，从而绑定到当前请求。

显然，format API 与上面看到的 API 是共通的。

这里有一个简单的例子:

```
zend_string *result;

result = strpprintf(0, "You are using PHP %s", PHP_VERSION);

/* Do something with result */

zend_string_release(result);
```

zend_ API 的说明

您可能会遇到 *zend_spprintf()* 或 *zend_strpprintf()* 函数。这些与上面看到函数的完全相同。

它们只是 Zend 引擎和 PHP 核心之间分离的一部分，这个细节对我们来说并不重要，在源代码中，所有内容都混合在一起。

3.3 资源类型:zend_resource

实际上尽管 PHP 已经可以丢弃“资源”类型，因为自定义对象存储允许构建任何抽象类型的数据，在当前 PHP 版本中仍然存在资源类型，你还是可能需要处理它。

如果你需要创建资源，我们希望您不要这样做，而是使用对象和自定义对象存储处理。对象（Objects）是 PHP 的一种类型，它可以嵌入任何类型的任何内容。然而因为历史原因，PHP 仍然要知道这种特殊类型的“资源”，并且在其核心代码或者一些扩展中仍然在使用它。我们一起了解一下这种类型。但是要注意，它确实很神秘并且历史悠久，所以不要对它的设计感到惊讶，特别是在阅读有关它的源码时。

3.3.1 什么是“资源”类型?

你很容易就知道了。我们正在讨论这个:

```
$fp = fopen('/proc/cpuinfo', 'r');
var_dump($fp); /* resource(2) of type (stream) */
```

在内部，资源被绑定到 *zend_resource* 结构类型:

```
struct _zend_resource {
        zend_refcounted_h gc;
        int               handle;
        int               type;
        void             *ptr;
};
```

我们发现了通用的 *zend_refcounted_h* 头，这意味这资源是可引用的。*handle* 是整数类型, 它在引擎内部被用来将资源定位到内部资源表中。它被用作此类表的键。

type 被用来将相同类型的资源重组在一起。这是关于资源被销毁的方式以及如何从它们的 handle 中取回。

最后，在 *zend_resource* 的 *ptr* 字段是你的抽象数据。请记住，资源是关于存储一个抽象数据的，它不能适合 PHP 原生的任何数据类型（但是对象可以，就像我们之前说的那样）。

3.3.2 资源类型和资源销毁

资源必须注册析构函数。当用户在 PHP 用户层使用资源时，当它们不再使用过这些资源时，它们也不会费心清理它们。例如：看见 *fopen()* 调用但是没有看到对应的 *fclose()* 调用是不稀罕的。使用 C 语言，这充其量是个坏主意，至多也就是一场灾难。但是使用像 PHP 这样的高级语言，事情就容易多了。

作为内部开发人员，你必须面对这样一个事实：用户将会创建许多你运行它们使用的资源，而且没有正确的清理它们并释放内存/系统资源。因此你必须一个析构函数，该析构函数将在引擎即将销毁该类型的资源时被

调用。

析构函数是按照类型分组的，资源也是这样。你不能请求“dtabase”类型资源的析构函数来来处理”file” 类型的资源。

还有两种资源，它们的生命周期是不同的。

- 经典资源，最常用的一种，不会跨多个请求存留，它们的析构函数会在请求关闭时被调用。
- 持久性资源将持续存在于多个请求中，并且只有在 PHP 进程终止时才会被销毁。

你可能会对PHP 生命周期这一章感兴趣，该章节展示了在 PHP 的流程生命周期内发生的不同步骤。此外，Zend Memory Manager章节可能有助于理解持久性和请求绑定内存分配的概念。

3.3.3 使用资源

与资源相关的 API 可以在zend/zend_list.c中查到。您可能会发现其中有一些不一致的地方，比如讨论“资源”的“列表”时。

创建资源

要创建资源，首先必须要为它注册析构函数，并使用 *zend_register_list_destructors_ex()* 将其关联到一个资源类型名称。此函数将返回一个表示你注册的资源类型的整数。该调用将返回一个整数，该整数表示您注册的资源类型。你必须记住该整数，因为你稍后需要从它那里获取资源。

然后，可以使用 *zend_register_resource()* 注册一个新的资源。它将会返回一个 *zend_resource*。让我们一起看一个简单的例子：

```
#include <stdio.h>

int res_num;
FILE *fp;
zend_resource *my_res;
zval my_val;

static void my_res_dtor(zend_resource *rsrc)
{
    fclose((FILE *)rsrc->ptr);
}
```

```
    /* module_number should be your PHP extension number here */
    res_num = zend_register_list_destructors_ex(my_res_dtor, NULL, "my_res"
, module_number);
    fp      = fopen("/proc/cpuinfo", "r");
    my_res  = zend_register_resource((void *)fp, res_num);

    ZVAL_RES(&my_val, my_res);
```

在上面的代码中，我们使用 libc 的 *fopen()* 打开了一个文件，并将返回的指针存储到一个资源中。在此之前，我们注册了一个析构函数，当他被调用时它会调用 libc's 的 *fclose()*。然后，我们将资源注册到引擎上，并将资源传递到一个 *zval* 容器中，它可以在用户层获取到。

Zvals 章节在这里

资源类型必须要记住。在这里我们注册了一个类型为“my_res“的资源。这是类型的名称。引擎实际上并不关注类型名称，而是类型标识符，即 *zend_register_list_destructors_ex()* 返回的整数。你应该在某处记下它，就像我们使用 *res_num* 变量那样。

获取资源

现在我们已经注册了一个资源并将其放入 *zval* 中作为实例，我们应该学习如何从用户层获取该资源。记住，该资源存储在 *zval* 中。资源结构中存储了资源类型编号（在 type 字段中）。因此，从用户层面获取资源，我们必须从 *zval* 中提取 *zend_resource*, 并调用 *zend_fetch_resource()* 来获取我们的 *FILE* * 指针：

```
    /* ... later on ... */

    zval *user_zval = /* fetch zval from userland, assume type IS_RESOURCE
*/

    ZEND_ASSERT(Z_TYPE_P(user_zval) == IS_RESOURCE); /* just a check to be
sure */

    fp = (FILE *)zend_fetch_resource(Z_RESVAL_P(user_zval), "my_res",
res_num);
```

如前所述: 从用户层面获取一个 *zval*(类型为 IS_RESOURCE)，并痛殴调用，并通过调用 *zend_fetch_resource()* 获取资源指针。

该函数将检查资源的类型是否与作为第三个参数传递的类型相同（这里是 *res_num*）。如果相同，它将返回你需要的 *void* * 资源指针，我们就完成了。如果不同，那么它会抛出一个类似于“supplied resource is not a valid type name resource”的警告。这种情况就会发生，例如，你期望得到一个“my_res” 类型的资源，但是你得到了一个资料类型为“gzip”的 *zval*，例如 PHP 函数 *gzopen()* 返回的那样，那么就会发生这种情况。

资源类型只是引擎将不同类型的资源 (类型为“file”、“gzip”甚至“mysql connection”) 混合到同一个资源表中的一种方法。资源类型有名称，因此可以在错误消息或调试语句中使用这些名称 (比如 *var_dump($my_resource)*)，并且它们还被表示为一个整数，用于在内部从资源类型中取回资源指针，并注册一个析构函数。

如您所见，如果我们使用对象，它们自身就表示类型，并且不需要从它的标识符获取资源来验证它的类型。对象是自描述类型。但是对于当前的 PHP 版本，资源仍然是有效的数据类型

3.3.4 资源引用计数

与许多其他类型一样，*zend_resource* 也是引用计数的。我们可以看到它的 *zend_refcounted_h* 头。如果你需要的话 (一般来说你不应该真的需要它)，可以使用引用计数的 API:

- *zend_list_delete(zend_resource *res)* 递减引用计数，如果为零可销毁资源
- *zend_list_free(zend_resource *res)* 检查 refcount 是否为零，如果是则销毁资源。
- *zend_list_close(zend_resource *res)* 在任何条件下调用资源析构函数

3.3.5 持久化资源

持久化资源在请求结束时不会被销毁。经典用例是持久化数据库连接。这些连接是在请求之间循环使用的。传统上，您不应该使用持久化资源，因为每个请求都是不同的。在这样做之前，重用相同的资源确实需要深思熟虑。

要注册持久化资源，请使用持久析构函数而不是经典析构函数。这是在 *zend_register_list_destructors_ex()* 调用时完成的，该 API 类似于:

```
      zend_register_list_destructors_ex(rsrc_dtor_func_t destructor,
rsrc_dtor_func_t persistent_destructor,
      const char *type_name, int module_number);
```

3.4 HashTables: zend_array

3.5 Functions: zend_function

3.6 Objects and classes

第四章　扩展开发

在本章你将学习如何开发 PHP 扩展。你将会了解 PHP 的生命周期、如何以及何时管理内存、你可以使用不同的钩子或者可以替换的不同函数指针来主动更改 PHP 的内部机制。你将通过开发扩展来实现 PHP 函数和类，并将使用 ini 设置。

4.1　学习 PHP 生命周期

PHP 是一个复杂的系统，任何要要参与它的人都应该真正掌握它的生命周期。主要内容如下:

PHP 启动。

第一章　附录

A.1　常用宏及函数

- *zend_string_init* 创建 zend_string
- *zend_string_copy* 字符串复制 (只增加引用计数)
- *zend_string_dup* 字符串拷贝 (发生内存分配)

```
//创建zend_string
zend_string *zend_string_init(const char *str, size_t len, int persistent);

//字符串复制(只增加引用计数)
zend_string *zend_string_copy(zend_string *s);

//字符串拷贝(发生内存分配)
zend_string *zend_string_dup(zend_string *s, int persistent);
```