# 1.4 线程安全

在PHP初期，是作为单进程的CGI来运行的，所以并没有考虑线程安全问题。
我们可以随意的在全局作用域中设置变量并在程序中对他进行修改、访问，内核申请的资源如果没有正确的释放，也会在CGI进程结束后自动地被清理干净。

后来，php被作为apache多进程模式下的一个模块运行，但是这仍然把php局限在一个进程里，
我们设置的全局变量，只要在每个请求之前将其正确的初始化，并在每个请求之后正确的清理干净，
便不会带来什么麻烦。由于对于一个进程来说，同一个时间只能处理一个请求，
所以这是内核中加入了针对每个请求的内存管理功能，来防止服务器资源利用出现错误。

随着使用在多线程模式的软件系统越来越多，php内核中亟需一种新的资源管理方式，
并最终在php内核中形成了一个新的抽象层：TSRM(Thread Safe Resource Management)。

## 线程安全与非线程安全

在一个没有线程的程序中，我们往往倾向于把全局变量声明在源文件的顶部，
编辑器会自动的为它分配资源供我们在声明语句之下的程序逻辑中使用。

（即使通过**fork()**出一个子进程，它也会重新申请一段内存，父子进程中的变量从此没有了任何联系）

但是在一个多线程的程序中，如果我们需要每个线程都拥有自己独立的资源的话，
便需要为每个线程独立开辟出一个区域来存放它们各自的资源，
在使用资源的时候，每个线程便会只在自己的那一亩三分地里找，而不会拔了别人的庄稼。

## Thread-Safe Data Pools(线程安全的资源池？)

在扩展的Module Init里，扩展可以调用ts_allocate_id()来告诉TRSM自己需要多少资源。
TRSM接收后更新系统使用的资源，并得到一个指向刚分配的那份资源的id。

```
typedef struct {
    int sampleint;
    char *samplestring;
} php_sample_globals;
int sample_globals_id;

PHP_MINIT_FUNCTION(sample)
{
    ts_allocate_id(&sample_globals_id,
        sizeof(php_sample_globals),
        (ts_allocate_ctor) php_sample_globals_ctor,
        (ts_allocate_dtor) php_sample_globals_dtor);
    return SUCCESS;
}
```

当一个请求需要访问数据段的时候，扩展从TSRM层请求当前线程的资源池，
以ts_allocate_id()返回的资源ID来获取偏移量。
换句话说，在代码流中，你可能会在前面所说的MINIT语句中碰到SAMPLE_G(sampleint) = 5;
这样的语句。在线程安全的构建下，这个语句通过一些宏扩展如下：

```
(((php_sample_globals*)(*((void ***)tsrm_ls))[sample_globals_id-1])->sampleint = 5;
```

如果你看不懂上面的转换也不用沮丧，它已经很好的封装在PHPAPI中了，以至于许多开发者都不需要知道它怎样工作的。

# 当不在线程环境时

因为在PHP的线程安全构建中访问全局资源涉及到在线程数据池查找对应的偏移量，这是一些额外的负载，结果就是它比对应的非线程方式（直接从编译期已经计算好的真实的全局变量地址中取出数据）慢一些。
考虑上面的例子，这一次在非线程构建下：

```
typedef struct {
    int sampleint;
    char *samplestring;
} php_sample_globals;
php_sample_globals sample_globals;

PHP_MINIT_FUNCTION(sample)
{
    php_sample_globals_ctor(&sample_globals TSRMLS_CC);
    return SUCCESS;
}
```

首先注意到的是这里并没有定义一个int型的标识去引用全局的结构定义，
只是简单的在进程的全局空间定义了一个结构体。
也就是说SAMPLE_G(sampleint) = 5;展开后就是sample_globals.sampleint = 5; 简单，快速，高效。
非线程构建还有进程隔离的优势，这样给定的请求碰到完全出乎意料的情况时，它也不会影响其他进程，
即便是产生段错误也不会导致整个webserver瘫痪。
实际上，Apache的MaxRequestsPerChild指令就是设计用来提升这个特性的，
它经常性的有目的性的kill掉子进程并产生新的子进程，来避免某些可能由于进程长时间运行“累积”而来的问题（比如内存泄露）。

# 访问全局变量

在创建一个扩展时，你并不知道它最终的运行环境是否是线程安全的。幸运的是，你要使用的标准包含文件集合中已经包含了条件定义的ZTS预处理标记。当PHP因为SAPI需要或通过enable-maintainer-zts选项安装等原因以线程安全方式构建时，这个值会被自动的定义，并可以用一组#ifdef ZTS这样的指令集去测试它的值。
就像你前面看到的，只有在PHP以线程安全方式编译时，才会存在线程安全池，只有线程安全池存在时，才会真的在线程安全池中分配空间。这就是为什么前面的例子包裹在ZTS检查中的原因，非线程方式供非线程构建使用。
在本章前面PHP_MINIT_FUNCTION(myextension)的例子中，你可以看到#ifdef ZTS被用作条件调用正确的全局初始代码。对于ZTS模式它使用ts_allocate_id()弹出myextension_globals_id变量，而非ZTS模式只是直接调用myextension_globals的初始化方法。这两个变量已经在你的扩展源文件中使用Zend宏：DECLARE_MODULE_GLOBALS(myextension)声明，它将自动的处理对ZTS的测试并依赖构建的ZTS模式选择正确的方式声明。

在访问这些全局变量的时候，你需要使用前面给出的自定义宏SAMPLE_G()。在第12章，你将学习到怎样设计这个宏以使它可以依赖ZTS模式自动展开。

## 即便你不需要线程也要考虑线程

正常的PHP构建默认是关闭线程安全的，只有在被构建的sapi明确需要线程安全或线程安全在./configure阶段显式的打开时，才会以线程安全方式构建。
给出了全局查找的速度问题和进程隔离的缺点后，你可能会疑惑为什么明明不需要还有人故意打开它呢？这是因为，多数情况下，扩展和SAPI的开发者认为你是线程安全开关的操作者，这样做可以很大程度上确保新代码可以在所有环境中正常运行。
当线程安全启用时，一个名为tsrm_ls的特殊指针被增加到了很多的内部函数原型中。这个指针允许PHP区分不同线程的数据。回想一下本章前面ZTS模式下的SAMPLE_G()宏函数中就使用了它。没有它，正在执行的函数就不知道查找和设置哪个线程的符号表；不知道应该执行哪个脚本，引擎也完全无法跟踪它的内部寄存器。这个指针保留了线程处理的所有页面请求。
这个可选的指针参数通过下面一组定义包含到原型中。当ZTS禁用时，这些定义都被展开为空；当ZTS开启时，它们展开如下：

```
#define TSRMLS_D     void ***tsrm_ls
#define TSRMLS_DC     , void ***tsrm_ls
#define TSRMLS_C     tsrm_ls
#define TSRMLS_CC     , tsrm_ls
```

非ZTS构建对下面的代码看到的是两个参数：int, char *。在ZTS构建下，原型则包含三个参数：int, char *, void ***。当你的程序调用这个函数时，只有在ZTS启用时才需要传递第三个参数。下面代码的第二行展示了宏的展开：

```
int php_myext_action(int action_id, char *message TSRMLS_DC);
php_myext_action(42, "The meaning of life" TSRMLS_CC);
```

通过在函数调用中包含这个特殊的变量，php_myext_action就可以使用tsrm_ls的值和MYEXT_G()宏函数一起访问它的线程特有全局数据。在非ZTS构建上，tsrm_ls将不可用，但是这是ok的，因为此时MYEXT_G()宏函数以及其他类似的宏都不会使用它。
现在考虑，你在一个新的扩展上工作，并且有下面的函数，它可以在你本地使用CLI SAPI的构建上正常工作，并且即便使用apache 1的apxs SAPI编译也可以正常工作：

```
static int php_myext_isset(char *varname, int varname_len)
{
    zval **dummy;


    if (zend_hash_find(EG(active_symbol_table),
        varname, varname_len + 1,
        (void**)&dummy) == SUCCESS) {
        /* Variable exists */
        return 1;
    } else {
        /* Undefined variable */
        return 0;
    }

}
```

所有的一切看起来都工作正常，你打包这个扩展发送给他人构建并运行在生产服务器上。让你气馁的是，对方报告扩展编译失败。
事实上它们使用了Apache 2.0的线程模式，因此它们的php构建启用了ZTS。当编译期碰到你使用的EG()宏函数时，它尝试在本地空间查找tsrm_ls没有找到，因为你并没有定义它并且没有在你的函数中传递。
修复这个问题非常简单；只需要在php_myext_isset()的定义上增加TSRMLS_DC，并在每行调用它的地方增加TSRMLS_CC。不幸的是，现在对方已经有点不信任你的扩展质量了，这样就会推迟你的演示周期。这种问题越早解决越好。
现在有了enable-maintainer-zts指令。通过在./configure时增加该指令来构建php，你的构建将自动的包含ZTS，哪怕你当前的SAPI（比如CLI）不需要它。打开这个开关，你可以避免这些常见的不应该出现的错误。
注意：在PHP4中，enable-maintainer-zts标记等价的名字是enable-experimental-zts；请确认使用你的php版本对应的正确标记。

# 寻回丢失的tsrm_ls

有时，我们需要在一个函数中使用tsrm_ls指针，但却不能传递它。通常这是因为你的扩展作为某个使用回调的库的接口，它并没有提供返回抽象指针的地方。考虑下面的代码片段：

```
void php_myext_event_callback(int eventtype, char *message)
{
    zval *event;


    /* $event = array('event'=>$eventtype,
                      'message'=>$message) */
    MAKE_STD_ZVAL(event);
    array_init(event);
    add_assoc_long(event, "type", eventtype);
    add_assoc_string(event, "message", message, 1);


    /* $eventlog[] = $event; */
    add_next_index_zval(EXT_G(eventlog), event);
}
PHP_FUNCTION(myext_startloop)
{
    /* The eventlib_loopme() function,
     * exported by an external library,
     * waits for an event to happen,
     * then dispatches it to the
     * callback handler specified.
     */
    eventlib_loopme(php_myext_event_callback);
}
```

虽然你可能不完全理解这段代码，但你应该注意到了回调函数中使用了EXT_G()宏函数，我们知道在线程安全构建下它需要tsrm_ls指针。修改函数原型并不好也不应该这样做，因为外部的库并不知道php的线程安全模型。那这种情况下怎样让tsrm_ls可用呢？
解决方案是前面提到的名为TSRMLS_FETCH()的Zend宏函数。将它放到代码片段的顶部，这个宏将执行给予当前线程上下文的查找，并定义本地的tsrm_ls指针拷贝。

这个宏可以在任何地方使用并且不用通过函数调用传递tsrm_ls，尽管这看起来很诱人，但是，要注意到这一点：TSRMLS_FETCH调用需要一定的处理时间。这在单次迭代中并不明显，但是随着你的线程数增多，随着你调用TSRMLS_FETCH()的点的增多，你的扩展就会显现出这个瓶颈。因此，请谨慎的使用它。
注意：为了和c++编译器兼容，请确保将TSRMLS_FETCH()和所有变量定义放在给定块作用域的顶部（任何其他语句之前）。因为TSRMLS_FETCH()宏自身有多种不同的解析方式，因此最好将它作为变量定义的最后一行。

# links


- 目录
- 上一节: PHP的生命周期
    - 下一节: 小结