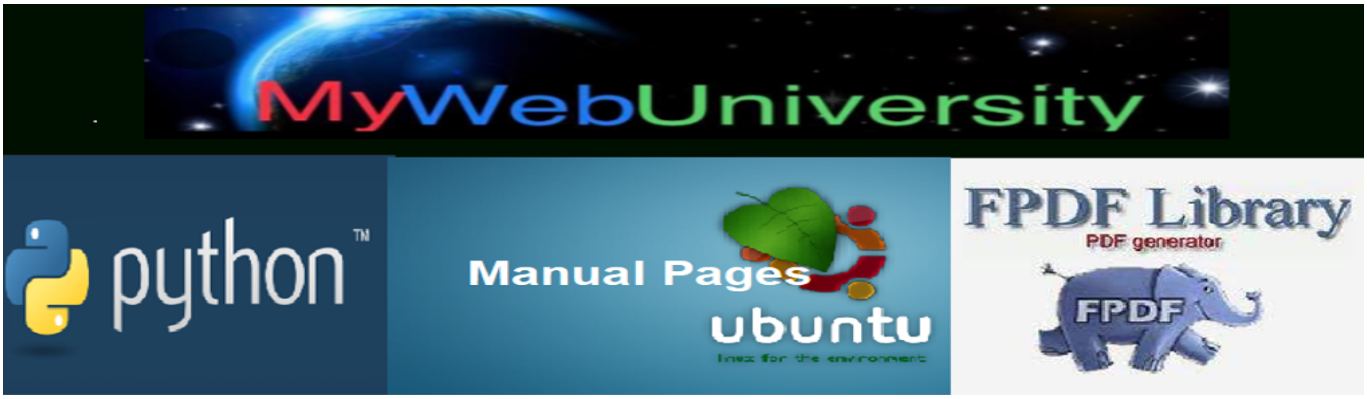




## *Rocky Enterprise Linux 9.2 Manual Pages on command 'APR::Pool.3pm'*

***$ man APR::Pool.3pm***

libapache2-mod-perl2-2.0.12::doUseraContriblibapache2-mod-perl2-2.0.12::docs::api::APR::Pool(3pm)

NAME

APR::Pool - Perl API for APR pools

Synopsis

```
use APR::Pool ();
my $sp = $r->pool->new;
my $sp2 = APR::Pool->new;
# $sp3 is a subpool of $sp,
# which in turn is a subpool of $r->pool
$sp3 = $sp->new;
print '$r->pool is an ancestor of $sp3'
    if $r->pool->is_ancestor($sp3);
# but sp2 is not a sub-pool of $r->pool
print '$r->pool is not an ancestor of $sp2'
    unless $r->pool->is_ancestor($sp2);
# $sp4 and $sp are the same pool (though you can't
# compare the handle as variables)
my $sp4 = $sp3->parent_get;
# register a dummy cleanup function
# that just prints the passed args
$sp->cleanup_register(sub { print @{ $_[0] || [] } }, [1..3]);
# tag the pool
$sp->tag("My very best pool");
```

```
# clear the pool
$sp->clear();
# destroy sub pool
$sp2->destroy;
```

## Description

"APR::Pool" provides an access to APR pools, which are used for an easy memory management. Different pools have different life scopes and therefore one doesn't need to free allocated memory explicitly, but instead it's done when the pool's life is getting to an end. For example a request pool is created at the beginning of a request and destroyed at the end of it, and all the memory allocated during the request processing using the request pool is freed at once at the end of the request.

Most of the time you will just pass various pool objects to the methods that require them. And you must understand the scoping of the pools, since if you pass a long lived server pool to a method that needs the memory only for a short scoped request, you are going to leak memory. A request pool should be used in such a case. And vice versa, if you need to allocate some memory for a scope longer than a single request, then a request pool is inappropriate, since when the request will be over, the memory will be freed and bad things may happen.

If you need to create a new pool, you can always do that via the "new()" method.

## API

"APR::Pool" provides the following functions and/or methods:

### "cleanup_register"

Register cleanup callback to run

```
$pool->cleanup_register($callback);
$pool->cleanup_register($callback, $arg);
```

obj: $pool ( "APR::Pool object" )

The pool object to register the cleanup callback for

arg1: $callback ( CODE ref or sub name )

a cleanup callback CODE reference or just a name of the subroutine (fully qualified unless defined in the current package).

opt arg2: $arg ( SCALAR )

If this optional argument is passed, the $callback function will receive it as the first and only argument when executed.

To pass more than one argument, use an ARRAY or a HASH reference

ret: no return value

excpt:

If a registered callback dies or throws an exception $@ is stringified and passed to "warn()". Usually, this results in printing it to the error_log. However, a $SIG{__WARN__} handler can be used to catch them.

```
$pool->cleanup_register(sub {die "message1\n"});
$pool->cleanup_register(sub {die "message2\n"});
my @warnings;
{
    local $SIG{__WARN__}=sub {push @warnings, @_};
    $pool->destroy;       # or simply undef $pool
}
```

Both of the cleanups above are executed at the time "$pool->destroy" is called. @warnings contains "message2\n" and "message1\n" afterwards. "$pool->destroy" itself does not throw an exception. Any value of $@ is preserved.

since: 2.0.00

If there is more than one callback registered (when "cleanup_register" is called more than once on the same pool object), the last registered callback will be executed first (LIFO).

Examples:

No arguments, using anon sub as a cleanup callback:

```
$r->pool->cleanup_register(sub { warn "running cleanup" });
```

One or more arguments using a cleanup code reference:

```
$r->pool->cleanup_register(\&cleanup, $r);
$r->pool->cleanup_register(\&cleanup, [$r, $foo]);
sub cleanup {
    my @args = (@_ && ref $_[0] eq ARRAY) ? @{ +shift } : shift;
    my $r = shift @args;
    warn "cleaning up";
}
```

No arguments, using a function name as a cleanup callback:

```
$r->pool->cleanup_register('foo');
```

"clear"

Clear all memory in the pool and run all the registered cleanups. This also destroys all sub-pools.

```
$pool->clear();
```

obj: $pool ( "APR::Pool object" )

The pool to clear

ret: no return value

since: 2.0.00

This method differs from "destroy()" in that it is not freeing the previously allocated, but allows the pool to re-use it for the future memory allocations.

"DESTROY"

"DESTROY" is an alias to "destroy". It's there so that custom "APR::Pool" objects will get properly cleaned up, when the pool object goes out of scope. If you ever want to destroy an "APR::Pool" object before it goes out of scope, use "destroy".

since: 2.0.00

"destroy"

Destroy the pool.

```
$pool->destroy();
```

obj: $pool ( "APR::Pool object" )

The pool to destroy

ret: no return value

since: 2.0.00

This method takes a similar action to "clear()" and then frees all the memory.

"is_ancestor"

Determine if pool a is an ancestor of pool b

```
$ret = $pool_a->is_ancestor($pool_b);
```

obj: $pool_a ( "APR::Pool object" )

The pool to search

arg1: $pool_b ( "APR::Pool object" )

The pool to search for

ret: $ret ( integer )

True if $pool_a is an ancestor of $pool_b.

since: 2.0.00

For example create a sub-pool of a given pool and check that the pool is an ancestor of

that sub-pool:

```
use APR::Pool ();
my $pp = $r->pool;
my $sp = $pp->new();
$pp->is_ancestor($sp) or die "Don't mess with genes!";
```

"new"

Create a new sub-pool

```
my $pool_child = $pool_parent->new;
my $pool_child = APR::Pool->new;
```

obj: $pool_parent ( "APR::Pool object" )

The parent pool.

If you don't have a parent pool to create the sub-pool from, you can use this object method as a class method, in which case the sub-pool will be created from the global pool:

```
my $pool_child = APR::Pool->new;
```

ret: $pool_child ( "APR::Pool object" )

The child sub-pool

since: 2.0.00

"parent_get"

Get the parent pool

```
$parent_pool = $child_pool->parent_get();
```

obj: $child_pool ( "APR::Pool object" )

the child pool

ret: $parent_pool ( "APR::Pool object" )

the parent pool. "undef" if there is no parent pool (which is the case for the top-most global pool).

since: 2.0.00

Example: Calculate how big is the pool's ancestry:

```
use APR::Pool ();
sub ancestry_count {
    my $child = shift;
    my $gen = 0;
    while (my $parent = $child->parent_get) {
```

```
            $gen++;
            $child = $parent;
        }
        return $gen;
    }
```

"tag"

Tag a pool (give it a name)

```
 $pool->tag($tag);
```

obj: $pool ( "APR::Pool object" )

The pool to tag

arg1: $tag ( string )

The tag (some unique string)

ret: no return value

since: 2.0.00

Each pool can be tagged with a unique label. This can prove useful when doing low level apr_pool C tracing (when apr is compiled with "-DAPR_POOL_DEBUG"). It allows you to grep(1) for the tag you have set, to single out the traces relevant to you.

Though there is no way to get read the tag value, since APR doesn't provide such an accessor method.

## Unsupported API

"APR::Pool" also provides auto-generated Perl interface for a few other methods which aren't tested at the moment and therefore their API is a subject to change. These methods will be finalized later as a need arises. If you want to rely on any of the following methods please contact the the mod_perl development mailing list so we can help each other take the steps necessary to shift the method to an officially supported API.

"cleanup_for_exec"

META: Autogenerated - needs to be reviewed/completed

Preparing for exec() --- close files, etc., but *don't* flush I/O buffers, *don't* wait for subprocesses, and *don't* free any memory.  Run all of the child_cleanups, so that any unnecessary files are closed because we are about to exec a new program

ret: no return value

since: subject to change

## See Also

mod_perl 2.0 documentation.

## Copyright



## Authors

The mod_perl development team and numerous contributors.