Ruby master - Bug #14858

Introduce 2nd GC heap named Transient heap

06/20/2018 06:13 PM - ko1 (Koichi Sasada)

Status:	Closed		
Priority:	Normal		
Assignee:	ko1 (Koichi Sasada)		
Target version:	2.6		
ruby -v:	2.6	**Backport:**	2.3: UNKNOWN, 2.4: UNKNOWN, 2.5: UNKNOWN

Description

Abstract

We propose to introduce "2nd" GC managed heap named "Transient heap" into MRI, instead of malloc management heap.
Employied GC algorithm is similar to "generational" "copying" GC algorithm.
This technique can reduce problems of malloc'ed heap.

Background

MRI and malloc

MRI manages memory by GC managed heap and malloc'ed heap.

Objects are allocated from GC managed heap. However, each object has 3 extra words to store data because each object slot only has 5 words (2 words for internal use). Extra data are allocated from malloc'ed heap and free the malloc'ed memory object when the object died.

For example, an array object consists of equal or less than 3 elements, it doesn't need malloc because we can use 3 slots to store them into an object slot. However, if an array object consists of greater than 3 elements, it require malloc'ed heap.

Using malloc'ed heap is easy way to implement, however there are several disadvantages.

- difficult to control malloc/free (pointed by https://bugs.ruby-lang.org/issues/14857#note-2, and discussion on glibc malloc and jemalloc)
- malloc/free has several overhead to call.

Introducing custom malloc library is one option, but it should be tough work to tune with system specific knowlege and to maintain it.

Generational Hypothesis

Most of objects die in young. This is common obserbation used by generational GC.
And we can see same behavior at least rdoc measurement https://bugs.ruby-lang.org/issues/14857.

Can we use this nature?

Copy GC algorithm

Copy GC algorithm is known as one of fastest GC because:

- Sweep time is proportional to living objects number.
- Copying memory objects can increase space locality.
- Copying becomes compaction so no fragmentation issue.

However, MRI can't employ copying algorithm because of C implementation limitation (implementation/compatibility issue).
Mostly copying (compaction) GC is suggested several times.

Copying GC can be combine with generational technique (collect only young objects).

Bump allocation

Fastest memory allocation algorithm is "bump allocation".

- (Init) allocate memory area and initialize index as 0
- (Alloc) increment index with required size and return original index.

Of course, this algorithm has many problem, especially fragmentation.

Design

Provide another generational copying GC heap for MRI and use it if we can use.
This proposal shows how to make a generational copying GC and use it from MRI.

The ideas are:

- (1) Provide copy GC area beside normal GC managed area.
- (2) Marking memory objects at normal GC marking.
- (3) Promote long-lived memory objects and copy them into malloc'ed heap.
- (4) Copy marked memory objects at *interrupt point*.

(3) and (4) are different from normal (generational) copying GCs.

Our proposal has several advantages:

- Fast allocation: We can employ bump allocation for memory allocation.
- Fast free: We don't need to touch free'ed memory objects.
- Early free: We don't need to wait lazy sweep to free spaces.
- Generational: Long lived memory objects are copied to malloc'ed heap.
- Controllable heap: We can manage memory buffers allocated by system (for short-lived memory objects).
- Easy maintenance: We can use system's malloc library for long living objects.
- Compatible: Our proposal is compatible with current MRI and existing C extensions.

We named this 2nd managed heap as "Transient heap" because this heap only manage short-living objects.

Algorithm

The following is outline of our proposed algorithm:

(1) Prepare memory area.
(2) rb_transient_heap_alloc() returns memory objects in memory area using bump allocation and mark object as transient object. Objects using transient heap are marked as "transient flag".
(2') If there are no space to return, then return NULL.
(3) Start GC marking. If marked object uses transient heap (marked by transient flag), tell object and allocated pointer to transient heap by rb_transient_heap_mark().
(4) After marking, wait until interrupt point. De-transient (allocate memory by malloc and copy from transient heap to malloc'ed memory) for all marked memory objects.
After de-transient all marked objects, then the memory area is free and reuse it.
(5) goto (2)

We need to wait interrupt point to de-transient because we can't move memory objects just after GC marking.

For example,

```
const VALUE *ptr = RARRAY_CONST_PTR(ary);
VALUE *tmp = ALLOC_N(size); // can cause GC
use(ptr);                   // ptr should point valid area
```

we can't move memory area between GC (marking).

However, we can move memory objects across interrupt point because any Ruby program can run at interrupt point and such Ruby program can invalidate raw pointers.

For example,

```
const VALUE *ptr = RARRAY_CONST_PTR(ary);
VALUE v = rb_funcall(...); // call Ruby method and can pass over interrupt point.
  // at interrupt point, if ary.replace(...) or something destructive operations can run here.
```

```
use(ptr);                   // ptr can be invalidated
```

This case, we need to care that ptr can be invalidated even if we are using current MRI.

Scenario

To show the algorithm, let's discuss with Array objects.

(1) Allocate an array

Array.new(4) allocates 4 words by malloc with current MRI.
Instead of using malloc', callrb_transient_heap_alloc(size)` to allocate a memory.
Returning pointer is allocated pointer from transient heap.
Set "Transient flag" to a created array object and set pointer.

(2-1) Free array object

If an array object is a short-lived object, then we don't do any more. We don't need to call free.

(2-2) Mark and deserialize

If an array object is long-lived object (survive one GC), we need to make this object survive.
So that GC marking phase, when an array is marked and the array is transient flag,
then call rb_transient_heap_mark() to tell transient heap that the object is living.

(3) Cleanup memory area for transient heap

Wait for an interrupt point (RUBY_VM_CHECK_INTS()) and de-transient for all marked (living) objects.
Transient heap has no living memory objects so that we can reuse this heap area.

Details

The above explanation is simplified description. See source code for details.

Points are:

- Using blocks instead of using one big memory area for transient heap.
- Separate "using blocks" and "marked blocks".
- Prepare promoted objects table to take care promoted objects.

Copying not just after marking was difficult than I expected...

Compatibility

Some code can rely on that raw pointer acquired from a frozen object can not be invalidate.
We need to survey more.

Result

Now only Array uses transient heap because of limitation of human resource.

micro-benchmark

```
require 'benchmark'
N = 10_000_000

def n_times str, args = ''
  eval <<-EOS
    proc{|max, #{args}|
      i = 0
      while i < max
        #{str}
        i+=1
      end
    }
  EOS
end

m = n_times 'ary = Array.new(size)', 'size'

Benchmark.bm(10){|x|
  0.step(to: 16){|i|
    size = i
    x.report(size){
      m.call(N, size)
    }
  }
}
```

This microbenchmark measure the time Array.new(n) (10M times) where 0 <= n <= 10.

Current MRI:

```
                 user     system      total        real
0            0.973336   0.000000   0.973336 (  0.969181)
1            1.005509   0.000000   1.005509 (  1.004734)
2            1.021980   0.000000   1.021980 (  1.025868)
3            1.153219   0.003657   1.156876 (  1.157852)
4            1.814474   0.004027   1.818501 (  1.822703)
5            1.882669   0.000000   1.882669 (  1.882087)
6            1.979505   0.003985   1.983490 (  1.984733)
7            1.960319   0.000000   1.960319 (  1.959602)
8            2.032639   0.000000   2.032639 (  2.033494)
9            2.160479   0.003994   2.164473 (  2.163686)
10           2.045469   0.008000   2.053469 (  2.054952)
11           1.901561   0.000000   1.901561 (  1.901080)
12           1.868254   0.004001   1.872255 (  1.872298)
13           1.864242   0.004001   1.868243 (  1.867719)
14           1.867359   0.004000   1.871359 (  1.871252)
15           1.849659   0.004004   1.853663 (  1.853597)
16           1.940981   0.003995   1.944976 (  1.946946)
```

Using proposed transient heap:

```
                 user     system      total        real
0            0.948643   0.000000   0.948643 (  0.951356)
1            0.945786   0.000000   0.945786 (  0.945294)
2            0.967074   0.000000   0.967074 (  0.967439)
3            0.964790   0.003212   0.968002 (  0.968342)
4            1.177959   0.000013   1.177972 (  1.178215)
5            1.200862   0.000006   1.200868 (  1.200600)
6            1.190707   0.000000   1.190707 (  1.190252)
7            1.297028   0.000000   1.297028 (  1.297079)
8            1.256999   0.000006   1.257005 (  1.256636)
9            1.364489   0.000002   1.364491 (  1.368413)
10           1.294030   0.000002   1.294032 (  1.293557)
11           1.392180   0.000000   1.392180 (  1.392618)
12           1.347170   0.000000   1.347170 (  1.347101)
13           1.426268   0.000000   1.426268 (  1.425432)
14           1.330723   0.000004   1.330727 (  1.333744)
15           1.458826   0.000000   1.458826 (  1.458509)
16           1.349267   0.000000   1.349267 (  1.348561)
```

Notable change is here:

```
# current MRI:
3            1.153219   0.003657   1.156876 (  1.157852)
4            1.814474   0.004027   1.818501 (  1.822703)

# transient heap:
3            0.964790   0.003212   0.968002 (  0.968342)
```

```
4            1.177959   0.000013   1.177972 (  1.178215)
```

We don't need to allocate extra heap for 3 elements.
With 4 elements the first element size requiring extra memory, the impact of transient heap is easy to see.

app benchmark

Run make install gcbench-rdoc.

Current MRI:

```
../../clean-trunk/benchmark/gc/rdoc.rb
      user     system      total        real
 15.599328  14.789680  30.389008 ( 35.894391)
GC total time (sec): 3.7352798950005948

VmHWM: 501916 kB

Summary of rdoc on 2.6.0dev     35.894391419016756      3.7352798950005948      167
         (real time in sec, GC time in sec, GC count)
```

Transient heap:

```
../../trunk/benchmark/gc/rdoc.rb
      user     system      total        real
 22.403497   6.122249  28.525746 ( 33.900175)
GC total time (sec): 3.0769403599987992

VmHWM: 408460 kB

Summary of rdoc on 2.6.0dev     33.90017474500928       3.0769403599987992      184
         (real time in sec, GC time in sec, GC count)
```

Surprisingly, VmHWM is reduced well (not surveyed yet).

No notable speedup.

Futurework

We need to implement more and mroe.

- Support String (most important)
- Support Hash (st), but it seems difficult
- Extend (shrink) policy for transient heap (now fixed size heap)

Conclusion

This ticket propose 2nd heap managed by generational copying GC named "Transient heap".
Transient heap only contains young memory objects and old memory objects (pointed from promoted objects) are escaped to malloc'ed memory.
Most of memory objects are died in young age (generatioal hypothesis) so that

I hope this proposal can solve malloc'ed heap problems.

Associated revisions

Revision 90ac549f - 10/30/2018 08:46 PM - ko1 (Koichi Sasada)

introduce TransientHeap. [Bug #14858]

- transient_heap.c, transient_heap.h: implement TransientHeap (theap). theap is designed for Ruby's object system. theap is like Eden heap on generational GC terminology. theap allocation is very fast because it only needs to bump up pointer and deallocation is also fast because we don't do anything. However we need to evacuate (Copy GC terminology) if theap memory is long-lived. Evacuation logic is needed for each type.

See [Bug #14858] for details.

- array.c: Now, theap for T_ARRAY is supported.

ary_heap_alloc() tries to allocate memory area from theap. If this trial
sccesses, this array has theap ptr and RARRAY_TRANSIENT_FLAG is turned on.
We don't need to free theap ptr.

- ruby.h: RARRAY_CONST_PTR() returns malloc'ed memory area. It menas that if ary is allocated at theap, force evacuation to malloc'ed memory. It makes programs slow, but very compatible with current code because theap memory can be evacuated (theap memory will be recycled).

If you want to get transient heap ptr, use RARRAY_CONST_PTR_TRANSIENT()
instead of RARRAY_CONST_PTR(). If you can't understand when evacuation
will occur, use RARRAY_CONST_PTR().

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65444 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 65444 - 10/30/2018 08:46 PM - ko1 (Koichi Sasada)

introduce TransientHeap. [Bug #14858]

- transient_heap.c, transient_heap.h: implement TransientHeap (theap). theap is designed for Ruby's object system. theap is like Eden heap on generational GC terminology. theap allocation is very fast because it only needs to bump up pointer and deallocation is also fast because we don't do anything. However we need to evacuate (Copy GC terminology) if theap memory is long-lived. Evacuation logic is needed for each type.

See [Bug #14858] for details.

- array.c: Now, theap for T_ARRAY is supported.

ary_heap_alloc() tries to allocate memory area from theap. If this trial
sccesses, this array has theap ptr and RARRAY_TRANSIENT_FLAG is turned on.
We don't need to free theap ptr.

- ruby.h: RARRAY_CONST_PTR() returns malloc'ed memory area. It menas that if ary is allocated at theap, force evacuation to malloc'ed memory. It makes programs slow, but very compatible with current code because theap memory can be evacuated (theap memory will be recycled).

If you want to get transient heap ptr, use RARRAY_CONST_PTR_TRANSIENT()
instead of RARRAY_CONST_PTR(). If you can't understand when evacuation
will occur, use RARRAY_CONST_PTR().

Revision 65444 - 10/30/2018 08:46 PM - ko1 (Koichi Sasada)

introduce TransientHeap. [Bug #14858]

- transient_heap.c, transient_heap.h: implement TransientHeap (theap). theap is designed for Ruby's object system. theap is like Eden heap on generational GC terminology. theap allocation is very fast because it only needs to bump up pointer and deallocation is also fast because we don't do anything. However we need to evacuate (Copy GC terminology) if theap memory is long-lived. Evacuation logic is needed for each type.

See [Bug #14858] for details.

- array.c: Now, theap for T_ARRAY is supported.

ary_heap_alloc() tries to allocate memory area from theap. If this trial
sccesses, this array has theap ptr and RARRAY_TRANSIENT_FLAG is turned on.
We don't need to free theap ptr.

- ruby.h: RARRAY_CONST_PTR() returns malloc'ed memory area. It menas that if ary is allocated at theap, force evacuation to malloc'ed memory. It makes programs slow, but very compatible with current code because theap memory can be evacuated (theap memory will be recycled).

If you want to get transient heap ptr, use RARRAY_CONST_PTR_TRANSIENT()
instead of RARRAY_CONST_PTR(). If you can't understand when evacuation
will occur, use RARRAY_CONST_PTR().

Revision 312b105d - 10/30/2018 09:53 PM - ko1 (Koichi Sasada)

introduce TransientHeap. [Bug #14858]

- transient_heap.c, transient_heap.h: implement TransientHeap (theap). theap is designed for Ruby's object system. theap is like Eden heap on generational GC terminology. theap allocation is very fast because it only needs to bump up pointer and deallocation is also fast because we don't do anything. However we need to evacuate (Copy GC terminology) if theap memory is long-lived. Evacuation logic is needed for each type.

See [Bug #14858] for details.

- array.c: Now, theap for T_ARRAY is supported.

ary_heap_alloc() tries to allocate memory area from theap. If this trial

sccesses, this array has theap ptr and RARRAY_TRANSIENT_FLAG is turned on.
We don't need to free theap ptr.

- ruby.h: RARRAY_CONST_PTR() returns malloc'ed memory area. It menas that if ary is allocated at theap, force evacuation to malloc'ed memory. It makes programs slow, but very compatible with current code because theap memory can be evacuated (theap memory will be recycled).

If you want to get transient heap ptr, use RARRAY_CONST_PTR_TRANSIENT()
instead of RARRAY_CONST_PTR(). If you can't understand when evacuation
will occur, use RARRAY_CONST_PTR().

(re-commit of r65444)

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65449 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 65449 - 10/30/2018 09:53 PM - ko1 (Koichi Sasada)

introduce TransientHeap. [Bug #14858]

- transient_heap.c, transient_heap.h: implement TransientHeap (theap). theap is designed for Ruby's object system. theap is like Eden heap on generational GC terminology. theap allocation is very fast because it only needs to bump up pointer and deallocation is also fast because we don't do anything. However we need to evacuate (Copy GC terminology) if theap memory is long-lived. Evacuation logic is needed for each type.

See [Bug #14858] for details.

- array.c: Now, theap for T_ARRAY is supported.

ary_heap_alloc() tries to allocate memory area from theap. If this trial
sccesses, this array has theap ptr and RARRAY_TRANSIENT_FLAG is turned on.
We don't need to free theap ptr.

- ruby.h: RARRAY_CONST_PTR() returns malloc'ed memory area. It menas that if ary is allocated at theap, force evacuation to malloc'ed memory. It makes programs slow, but very compatible with current code because theap memory can be evacuated (theap memory will be recycled).

If you want to get transient heap ptr, use RARRAY_CONST_PTR_TRANSIENT()
instead of RARRAY_CONST_PTR(). If you can't understand when evacuation
will occur, use RARRAY_CONST_PTR().

(re-commit of r65444)

Revision 65449 - 10/30/2018 09:53 PM - ko1 (Koichi Sasada)

introduce TransientHeap. [Bug #14858]

- transient_heap.c, transient_heap.h: implement TransientHeap (theap). theap is designed for Ruby's object system. theap is like Eden heap on generational GC terminology. theap allocation is very fast because it only needs to bump up pointer and deallocation is also fast because we don't do anything. However we need to evacuate (Copy GC terminology) if theap memory is long-lived. Evacuation logic is needed for each type.

See [Bug #14858] for details.

- array.c: Now, theap for T_ARRAY is supported.

ary_heap_alloc() tries to allocate memory area from theap. If this trial
sccesses, this array has theap ptr and RARRAY_TRANSIENT_FLAG is turned on.
We don't need to free theap ptr.

- ruby.h: RARRAY_CONST_PTR() returns malloc'ed memory area. It menas that if ary is allocated at theap, force evacuation to malloc'ed memory. It makes programs slow, but very compatible with current code because theap memory can be evacuated (theap memory will be recycled).

If you want to get transient heap ptr, use RARRAY_CONST_PTR_TRANSIENT()
instead of RARRAY_CONST_PTR(). If you can't understand when evacuation
will occur, use RARRAY_CONST_PTR().

(re-commit of r65444)

History

#1 - 06/20/2018 06:58 PM - jeremyevans0 (Jeremy Evans)

ko1 (Koichi Sasada) wrote:

> Some code can rely on that raw pointer acquired from a frozen object can not be invalidate.
> We need to survey more.

Is it possible to move the storage from the transient heap to the malloc heap eagerly if the pointer is requested via the C-API while the array is stored on the transient heap, and then return the new pointer to the malloc heap location at that point? Basically, never leak the pointer to the transient heap via the C-API? If not, I'm not sure this is safe enough.

Using your example:

```
const VALUE *ptr = RARRAY_CONST_PTR(ary); // move ary storage from
       // transient heap to malloc heap, return ptr to malloc heap
VALUE *tmp = ALLOC_N(size); // can cause GC
use(ptr);                   // ptr still points to valid area
```

Is it possible to have a compile time or run time flag to disable the use of transient heaps? By doing this custom memory management, you lose any security features in the malloc implementation, such as canaries to detect heap overflows. In some environments, going faster and using less memory but reducing security is not a good tradeoff.

#2 - 06/21/2018 01:52 AM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> transient_heap.patch (43 KB)

I wanted to try it, but patch seems misgenerated:

> c:/ko1/tools/mklog.rb:21:in ===': invalid byte sequence in UTF-8 (ArgumentError)
> from c:/ko1/tools/mklog.rb:21:inblock in '
> from c:/ko1/tools/mklog.rb:19:in each_line'
> from c:/ko1/tools/mklog.rb:19:in'

And incomplete, it ends at the top of rb_transient_heap_start_marking
(bottom of function and everything else is missing).

#3 - 06/21/2018 03:12 AM - ko1 (Koichi Sasada)

- File transient_heap.patch added

> I wanted to try it, but patch seems misgenerated:

Sorry it was my mistake.

#4 - 06/21/2018 03:12 AM - ko1 (Koichi Sasada)

- File deleted (transient_heap.patch)

#5 - 06/21/2018 09:52 AM - Eregon (Benoit Daloze)

Sounds nice, I was wondering when MRI would use a custom allocator for Ruby-level data since the jemalloc discussion started.
malloc() is generic but so much slower than a specialized allocator.

#6 - 06/21/2018 11:12 PM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> https://bugs.ruby-lang.org/issues/14858

Thanks, I can confirm a good result with this. However, 32k
is too small to be worth using mmap on.
And maybe mmap is unnecessary (and bad for portability).

Using malloc, I seem to get a tiny improvement in space and time:

```
--- a/transient_heap.c
+++ b/transient_heap.c
@@ -237,10 +237,8 @@ static struct transient_heap_block *
transient_heap_block_alloc(struct transient_heap* theap)
{
struct transient_heap_block *block;
-    block = mmap(NULL, TRANSIENT_HEAP_BLOCK_SIZE, PROT_READ | PROT_WRITE,
-                     MAP_PRIVATE | MAP_ANONYMOUS,
-                   -1, 0);
-     if (block == MAP_FAILED) rb_bug("transient_heap_block_alloc: err:%d\n", errno);
+     block = malloc(TRANSIENT_HEAP_BLOCK_SIZE);
+     if (!block) rb_bug("transient_heap_block_alloc: err:%d\n", errno);

 reset_block(block);

 @@ -501,12 +499,7 @@ transient_heap_reset(void)
 theap->total_objects -= block->info.objects;
  #if TRANSIENT_HEAP_INFINITE_BLOCK_MODE
 // debug mode
-         if (madvise(block, TRANSIENT_HEAP_BLOCK_SIZE, MADV_DONTNEED) != 0) {
-             rb_bug("madvise err:%d", errno);
-         }
-         if (mprotect(block, TRANSIENT_HEAP_BLOCK_SIZE, PROT_NONE) != 0) {
-             rb_bug("mprotect err:%d", errno);
-         }
+         free(block);
 theap->total_blocks--;
  #else
 reset_block(block);
```

#7 - 06/22/2018 06:49 AM - ko1 (Koichi Sasada)

> Thanks, I can confirm a good result with this. However, 32k
> is too small to be worth using mmap on.
> And maybe mmap is unnecessary (and bad for portability).

Thank you. It should be fixed.
(I'll use posix_memalign for blocks_alloc_header_to_block)

BTW, why small size mamp is harmful?
Making small page table entries?

#8 - 06/22/2018 08:12 AM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> Thank you. It should be fixed.
> (I'll use posix_memalign for blocks_alloc_header_to_block)

OK, but be careful with over-using memalign, it can cause more
fragmentation.

> BTW, why small size mamp is harmful?
> Making small page table entries?

Yes, and overall syscall overhead.

Also, I think TRANSIENT_HEAP_ALLOC_MAX should be dynamic or
even unlimited (SSIZE_MAX) to work well with strings.

#9 - 06/24/2018 01:04 AM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> - Support String (most important)

I expect this will be difficult, too.

Too many places release GVL or otherwise trigger GC while
RSTRING_PTR is being used.

rb_str_locktmp is not widely-used (and not documented in
doc/extension.rdoc). rb_str_tmp_frozen_acquire/release aren't
even exported, so GVL-releasing write()-like functions will
all be in trouble.

RSTRING_PTR will will have to be gradually replaced in similar way to RARRAY_PTR is being replaced for RGenGC.

We have no idea how long a read()/write() will take when we
release GVL.

#10 - 10/30/2018 08:46 PM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

Applied in changeset trunk|r65444.

---

introduce TransientHeap. [Bug #14858]

- transient_heap.c, transient_heap.h: implement TransientHeap (theap). theap is designed for Ruby's object system. theap is like Eden heap on generational GC terminology. theap allocation is very fast because it only needs to bump up pointer and deallocation is also fast because we don't do anything. However we need to evacuate (Copy GC terminology) if theap memory is long-lived. Evacuation logic is needed for each type.

See [Bug #14858] for details.

- array.c: Now, theap for T_ARRAY is supported.

ary_heap_alloc() tries to allocate memory area from theap. If this trial
sccesses, this array has theap ptr and RARRAY_TRANSIENT_FLAG is turned on.
We don't need to free theap ptr.

- ruby.h: RARRAY_CONST_PTR() returns malloc'ed memory area. It menas that if ary is allocated at theap, force evacuation to malloc'ed memory. It makes programs slow, but very compatible with current code because theap memory can be evacuated (theap memory will be recycled).

If you want to get transient heap ptr, use RARRAY_CONST_PTR_TRANSIENT()
instead of RARRAY_CONST_PTR(). If you can't understand when evacuation
will occur, use RARRAY_CONST_PTR().

Files

transient_heap.patch	45.5 KB	06/21/2018	ko1 (Koichi Sasada)