Ruby - Feature #7457

GC.stat to return "allocated object count" and "freed object count"

11/29/2012 04:31 AM - ko1 (Koichi Sasada)

Status:	Closed
Priority:	Normal
Assignee:	authorNari (Narihiro Nakamura)
Target version:	2.0.0

Description

How about to return "allocated object count" and "freed object count"?

The following patch enable to show "total allocated object number"
and "total freed (deallocated) object number".

pp GC.stat #=>
{:count=>0,
:heap_used=>12,
:heap_length=>12,
:heap_increment=>0,
:heap_live_num=>7494,
:heap_free_num=>0,
:heap_final_num=>0,
:heap_allocated_num=>7585, # <= new one!
:heap_freed_num=>88} # <= new one!

Maybe performance has mostly no impact with this patch.

Exact live object number can be calculated by "heap_allocated_num - heap_freed_num".

These values will be overflow. So they are only hint of performance tuning.

Index: gc.c

--- gc.c (revision 37946)
+++ gc.c (working copy)
@@ -225,7 +225,8 @@ typedef struct rb_objspace {
struct heaps_free_bitmap *free_bitmap;
RVALUE *range[2];
struct heaps_header *freed;

- size_t live_num;

- size_t allocated_num;
- size_t freed_num;
  size_t free_num;
  size_t free_min;
  size_t final_num;
  @@ -352,8 +353,6 @@ static inline void gc_prof_mark_timer_st
  static inline void gc_prof_sweep_timer_start(rb_objspace_t *);
  static inline void gc_prof_sweep_timer_stop(rb_objspace_t *);
  static inline void gc_prof_set_malloc_info(rb_objspace_t *);
  -static inline void gc_prof_inc_live_num(rb_objspace_t *);
  -static inline void gc_prof_dec_live_num(rb_objspace_t *);

/*
@@ -531,7 +530,6 @@ assign_heap_slot(rb_objspace_t *objspace
objspace->heap.sorted[hi]->bits = (uintptr_t *)objspace->heap.free_bitmap;
objspace->heap.free_bitmap = objspace->heap.free_bitmap->next;
memset(heaps->bits, 0, HEAP_BITMAP_LIMIT * sizeof(uintptr_t));

- objspace->heap.free_num += objs;
  pend = p + objs;

if (lomem == 0 || lomem > p) lomem = p;
if (himem < pend) himem = pend;
@@ -660,7 +658,7 @@ newobj(VALUE klass, VALUE flags)
RANY(obj)->file = rb_sourcefile();
RANY(obj)->line = rb_sourceline();
#endif

- gc_prof_inc_live_num(objspace);

- objspace->heap.allocated_num++;

return obj;
}
@@ -1422,7 +1420,8 @@ finalize_list(rb_objspace_t *objspace, R
if (!FL_TEST(p, FL_SINGLETON)) { /* not freeing page */
add_slot_local_freelist(objspace, p);
if (!is_lazy_sweeping(objspace)) {

- [redacted]

- [redacted]

- [redacted]
  }

}
else {
@@ -1873,10 +1872,16 @@ gc_clear_slot_bits(struct heaps_slot *sl
memset(slot->bits, 0, HEAP_BITMAP_LIMIT * sizeof(uintptr_t));
}

+static size_t
+objspace_live_num(rb_objspace_t *objspace)
+{

- return objspace->heap.allocated_num - objspace->heap.freed_num;
  +}

static void
slot_sweep(rb_objspace_t *objspace, struct heaps_slot *sweep_slot)
{

- size_t free_num = 0, final_num = 0;

- size_t empty_num = 0, freed_num = 0, final_num = 0;
  RVALUE *p, *pend;
  RVALUE *final = deferred_final_list;
  int deferred;
  @@ -1903,17 +1908,17 @@ slot_sweep(rb_objspace_t *objspace, stru
  p->as.free.flags = 0;
  p->as.free.next = sweep_slot->freelist;
  sweep_slot->freelist = p;

- [redacted]

- [redacted]
  }
  }
  else {

- [redacted]

- [redacted]

```
        }
    }
    p++;
```

}
gc_clear_slot_bits(sweep_slot);

- if (final_num + free_num == sweep_slot->header->limit &&

- if (final_num + freed_num + empty_num == sweep_slot->header->limit &&
objspace->heap.free_num > objspace->heap.do_heap_free) {
RVALUE *pp;

@@ -1925,13 +1930,14 @@ slot_sweep(rb_objspace_t *objspace, stru
unlink_heap_slot(objspace, sweep_slot);
}
else {

- [redacted]

- [redacted]

```
        link_free_heap_slot(objspace, sweep_slot);
    }
    else {
        sweep_slot->free_next = NULL;
    }
```

- [redacted]

- objspace->heap.freed_num += freed_num;
- objspace->heap.free_num += freed_num + empty_num;
}
objspace->heap.final_num += final_num;

@@ -1990,7 +1996,8 @@ after_gc_sweep(rb_objspace_t *objspace)

```
inc = ATOMIC_SIZE_EXCHANGE(malloc_increase, 0);
if (inc > malloc_limit) {
```

- malloc_limit += (size_t)((inc - malloc_limit) * (double)objspace->heap.live_num / (heaps_used * HEAP_OBJ_LIMIT));

- malloc_limit +=

- [redacted]

if (malloc_limit < initial_malloc_limit) malloc_limit = initial_malloc_limit;
}

@@ -2063,7 +2070,7 @@ gc_prepare_free_objects(rb_objspace_t *o
gc_marks(objspace);

```
before_gc_sweep(objspace);
```

- if (objspace->heap.free_min > (heaps_used * HEAP_OBJ_LIMIT - objspace->heap.live_num)) {

- if (objspace->heap.free_min > (heaps_used * HEAP_OBJ_LIMIT - objspace_live_num(objspace))) {
set_heaps_increment(objspace);
}

@@ -2544,7 +2551,6 @@ gc_mark_ptr(rb_objspace_t *objspace, VAL
register uintptr_t *bits = GET_HEAP_BITMAP(ptr);

if (MARKED_IN_BITMAP(bits, ptr)) return 0;
MARK_IN_BITMAP(bits, ptr);

- objspace->heap.live_num++;
  return 1;
  }

@@ -2905,11 +2911,8 @@ gc_marks(rb_objspace_t *objspace)
objspace->mark_func_data = 0;

```
gc_prof_mark_timer_start(objspace);
```

- objspace->heap.live_num = 0;
  objspace->count++;

- SET_STACK_END;

  th->vm->self ? rb_gc_mark(th->vm->self) : rb_vm_mark(th->vm);
  @@ -2956,7 +2959,8 @@ rb_gc_force_recycle(VALUE p)
  add_slot_local_freelist(objspace, (RVALUE *)p);
  }
  else {

- [redacted]

- objspace->heap.freed_num++;
- objspace->heap.free_num++;
  slot = add_slot_local_freelist(objspace, (RVALUE *)p);
  if (slot->free_next == NULL) {
  link_free_heap_slot(objspace, slot);
  @@ -3172,9 +3176,11 @@ gc_stat(int argc, VALUE *argv, VALUE sel
  rb_hash_aset(hash, ID2SYM(rb_intern("heap_used")), SIZET2NUM(objspace->heap.used));
  rb_hash_aset(hash, ID2SYM(rb_intern("heap_length")), SIZET2NUM(objspace->heap.length));
  rb_hash_aset(hash, ID2SYM(rb_intern("heap_increment")), SIZET2NUM(objspace->heap.increment));

- rb_hash_aset(hash, ID2SYM(rb_intern("heap_live_num")), SIZET2NUM(objspace->heap.live_num));

- rb_hash_aset(hash, ID2SYM(rb_intern("heap_live_num")), SIZET2NUM(objspace_live_num(objspace)));
  rb_hash_aset(hash, ID2SYM(rb_intern("heap_free_num")), SIZET2NUM(objspace->heap.free_num));
  rb_hash_aset(hash, ID2SYM(rb_intern("heap_final_num")), SIZET2NUM(objspace->heap.final_num));
- rb_hash_aset(hash, ID2SYM(rb_intern("heap_allocated_num")), SIZET2NUM(objspace->heap.allocated_num));
- rb_hash_aset(hash, ID2SYM(rb_intern("heap_freed_num")), SIZET2NUM(objspace->heap.freed_num));
  return hash;
  }

@@ -3952,7 +3958,7 @@ gc_prof_set_malloc_info(rb_objspace_t *o
static inline void
gc_prof_set_heap_info(rb_objspace_t *objspace, gc_profile_record *record)
{

- size_t live = objspace->heap.live_num;

- size_t live = objspace_live_num(objspace);
  size_t total = heaps_used * HEAP_OBJ_LIMIT;

  record->heap_total_objects = total;
  @@ -3960,16 +3966,6 @@ gc_prof_set_heap_info(rb_objspace_t *obj
  record->heap_total_size = total * sizeof(RVALUE);
  }

-static inline void
-gc_prof_inc_live_num(rb_objspace_t *objspace)
-{

-}

-static inline void
-gc_prof_dec_live_num(rb_objspace_t *objspace)
-{
-}

#else

static inline void
@@ -4057,18 +4053,6 @@ gc_prof_set_heap_info(rb_objspace_t *obj
record->have_finalize = deferred_final_list ? Qtrue : Qfalse;
record->heap_use_size = live * sizeof(RVALUE);
record->heap_total_size = total * sizeof(RVALUE);
-}

-static inline void
-gc_prof_inc_live_num(rb_objspace_t *objspace)
-{

- objspace->heap.live_num++;
  -}

-static inline void
-gc_prof_dec_live_num(rb_objspace_t *objspace)
-{

- objspace->heap.live_num--;
  }

#endif /* !GC_PROFILE_MORE_DETAIL */

History

#1 - 11/29/2012 05:06 AM - bitsweat (Jeremy Daer)

Yes!! A million times yes. Tracking *total* allocations makes it possible to profile Ruby code by object creation instead of time. This is very useful, often more so than profiling process time, because reducing excessive object creation will massively speed up GC and reduce the max # of heaps the VM needs, so lower memory for the process, all due to less object churn.

The ruby-prof gem supports object allocation profiling for 1.8.x (REE patches). The REE patches provide rb_os_allocated_objects and ObjectSpace.allocated_objects. Ideally, we would have a simple reader method like this to avoid creating lots of GC.stat Hash objects.

Even better, the benchmark.rb stdlib could then support benchmarking by objects created and objects created per second in addition to process time!

#2 - 11/29/2012 08:53 AM - authorNari (Narihiro Nakamura)

Could you commit it?

Thanks!

#3 - 11/29/2012 01:59 PM - ko1 (Koichi Sasada)

(2012/11/29 5:06), bitsweat (Jeremy Kemper) wrote:

> The ruby-prof gem supports object allocation profiling for 1.8.x (REE patches). The REE patches provide rb_os_allocated_objects and ObjectSpace.allocated_objects. Ideally, we would have a simple reader method like this to avoid creating lots of GC.stat Hash objects.

You can avoid to create Hash object using 1st parameter.

```
h = {}
r = 10.times.map{
GC.stat(h); h[:heap_total_allocated_num]
}
```

p r #=> [2422, 2423, 2423, 2423, 2423, 2423, 2423, 2423, 2423, 2423]

--
// SASADA Koichi at atdot dot net

#4 - 11/29/2012 02:10 PM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

Finally, I introduce two keys.

```
  * total_allocated_object: total allocated object number.
  * total_freed_object: total freed object number.
```

I remove "heap_" prefix because this information is not about current "heap_".

Give us comments.

Thanks,
Koichi