プログラミング言語 5
ガベージコレクション
Programming Language 5
Garbage Collection

田浦

目次

1. C/C++のメモリ管理 / Memory Management in C/C++
2. ガベージコレクション (GC) / Garbage Collection (GC)
3. 基本原理と用語 / Basics and Terminologies
   - 2 大方式 (走査型と参照カウント) / Two basic methods (traversing GC and reference counting)
   - GC の良し悪しの基準 / Criteria of evaluating GCs
   - 2 つの走査型 GC (マーク&スイープとコピー) / Two traversing GCs (mark&sweep and copying)
   - 走査型 GC のメモリ割り当てコスト (mark-cons 比) / Memory allocation cost of traversing GCs (mark-cons ratio)
4. C/C++用の GC ライブラリ / A GC library for C/C++

動機づけ: C/C++言語のメモリ割り当て

1. 大域変数/配列
2. 局所変数/配列
3. ヒープ

```
1  int g; int ga[10];
2  int foo() {
3    int l; int la[10];
4    int * a = &g;
5    int * b = ga;
6    int * c = &l;
7    int * d = la;
8    int * e = malloc(sizeof(int));
9  }
```

4. 寿命 (lifetime)

	開始	終了
大域	プログラム開始時	プログラム終了時
局所	ブロック開始時	ブロック終了時
ヒープ	malloc, new	free, delete

5. 注: 以降の議論では「変数」も「配列」も合わせて変数と呼ぶ

Motivation: memory allocation in C/C++

1. Global variables/arrays
2. Local variables/arrays
3. Heap

```
1  int g; int ga[10];
2  int foo() {
3    int l; int la[10];
4    int * a = &g;
5    int * b = ga;
6    int * c = &l;
7    int * d = la;
8    int * e = malloc(sizeof(int));
9  }
```

- lifetime

	starts	ends
global	when the program starts	when program ends
local	when a block starts	when a block ends
heap	malloc, new	free, delete

- note: the following discussion calls both "variables" and "arrays" variables (the distinction is not important)

起きうる間違い

- 寿命を超えて変数をアクセスする
- ヒープから割り当てた変数を開放し忘れる (メモリリーク)

How they can go wrong

- access a variable beyond its lifetime
- forget to release/reclaim a variable (memory leak)

寿命を超えたアクセス

- 変数の「寿命」の意味: 「寿命の間」に限り，変数はまともに振る舞う = 代入した値を覚えておいてくれる.
- 寿命を超えたアクセス
  - 仕様: 「未定義」
  - 実際の症状: その変数が (寿命の間) 置かれていた領域が，開放される
    - ⇒ 他の変数のために再利用される
    - ⇒ 知らぬ間に壊される，他のデータを壊す
    - もちろん実行時の型安全性も保証されなくなる

Accessing a variable after its lifetime

- what is the "lifetime" of a variable: the period in which it behaves as expected (= remembers the assigned value)
- if you access a variable after its lifetime
  - specification: "undefined"
  - what happens in practice: the memory region that hosted the variable (during its lifetime) may have been released
    - ⇒ the region may have been reused for other variables
    - ⇒ the variable corrupts other variables and vice versa
    - type safety will be lost too

寿命を超えたアクセスの例

局所変数

```
1  int * foo() {
2    int a[100];
3    return a;
4  }
5
6  int main() {
7    int * p = foo();
8    p[0] = ...
9  }
```

ヒープ

```
1  typedef struct list {
2    int val;
3    struct list * next;
4  } list;
5
6  void destroy_list(list * n) {
7    for (list * p = n; p; p = p->next) {
8      free(p);
9    }
10 }
```

An example accessing a variable beyond its lifetime

local variable

```
1  int * foo() {
2    int a[100];
3    return a;
4  }
5
6  int main() {
7    int * p = foo();
8    p[0] = ...
9  }
```

heap (do you see what's wrong?)

```
1  typedef struct list {
2    int val;
3    struct list * next;
4  } list;
5
6  void destroy_list(list * n) {
7    for (list * p = n; p; p = p->next) {
8      free(p);
9    }
10 }
```

ガベージコレクション (Garbage Collection; GC)

- つまりは「寿命」と「アクセスする期間」が一致していないのが問題
  - 寿命後でもアクセスできてしまう
  - もうアクセスしないのに開放しない (生きっ放し)
- ⇒ ガベージコレクション (GC)
  - 今後アクセスされ得るものは残し，され得ないものは開放 (再利用) する
  - それを処理系が自動的に行う
  - ⇒ リークや，寿命後のアクセスによるメモリ破壊をなくす
  - C, C++, 古代の言語以外はほぼ搭載している
- 今後アクセスされ { 得る・得ない } ものなんてなぜわかるのでしょう?

Garbage Collection (GC)

- the fundamental problem is that "lifetime" does not match "the period accessed"
  - you may access a variable after its lifetime
  - you may not free a variable alive despite you no longer access it

Garbage Collection (GC)

- the fundamental problem is that "lifetime" does not match "the period accessed"
  - you may access a variable after its lifetime
  - you may not free a variable alive despite you no longer access it
- $\Rightarrow$ Garbage collection (GC)
  - keep variables alive if ever be accessed in future and release (recycle) otherwise
  - the system automatically does that
  - $\Rightarrow$ eliminate memory leak and corruption
  - most languages have it, except C, C++, and some ancient languages

Garbage Collection (GC)

- the fundamental problem is that "lifetime" does not match "the period accessed"
  - you may access a variable after its lifetime
  - you may not free a variable alive despite you no longer access it
- $\Rightarrow$ Garbage collection (GC)
  - keep variables alive if ever be accessed in future and release (recycle) otherwise
  - the system automatically does that
  - $\Rightarrow$ eliminate memory leak and corruption
  - most languages have it, except C, C++, and some ancient languages
- the question: how does the system know *which variables may be accessed in future*?

今後アクセスされ{得る・得ない}もの

- 正確な判定は決定不能
- (f(x) 開始時点で)「p が指している領域は今後アクセスされる」
  $\Longleftrightarrow$ 「f(x) が終了して 0 を返す」
  $\rightarrow$ 停止問題を解く必要が...
- $\rightarrow$「今後アクセスされるかも」を大きめに評価
  - NG: 実はアクセスされるものを回収
  - OK: 実はアクセスされないものを回収しない
- 上の例ならば p はアクセスされる「かも」($\rightarrow$ 回収しない) とわかれば十分

```
1 int main() {
2   if (f(x) == 0) {
3     printf("%d\n", p->f->x);
4   }
5 }
```

Variables that may {ever/never} be accessed

- the precise judgment is undecidable
- (at the start of `f(x)`) "the variable pointed to by `p` will ever be accessed" $\Longleftrightarrow$ "`f(x)` will terminate and return 0" $\rightarrow$ you need to be able to solve the halting problem...

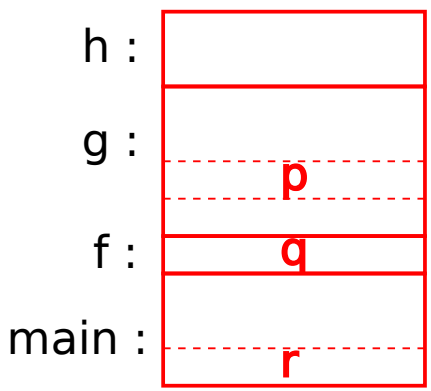
```
1 int main() {
2   if (f(x) == 0) {
3     printf("%d\n", p->f->x);
4   }
5 }
```

- $\rightarrow$ conservatively estimate variables that "may be accessed"
  - NG: reclaim those that are accessed
  - OK: not to reclaim those that are in fact never accessed
- in the above example, OK to judge the variable pointed to by `p` "may be" accessed ($\rightarrow$ so will be retained)

アクセスされる「かも」しれないデータ

- 大域変数
- 現在活性な (始まったが終了していない) 関数呼び出しの局所変数

```
1  int * s;
2  int * t;
3  void h() { ... }
4  void g() {
5    ...
6    h();
7    ... = p->x ...   }
8  void f() {
9    ...
10   g()
11   ... = q->y ... }
12 int main() {
13   ...
14   f()
15   ... = r->z ... }
```



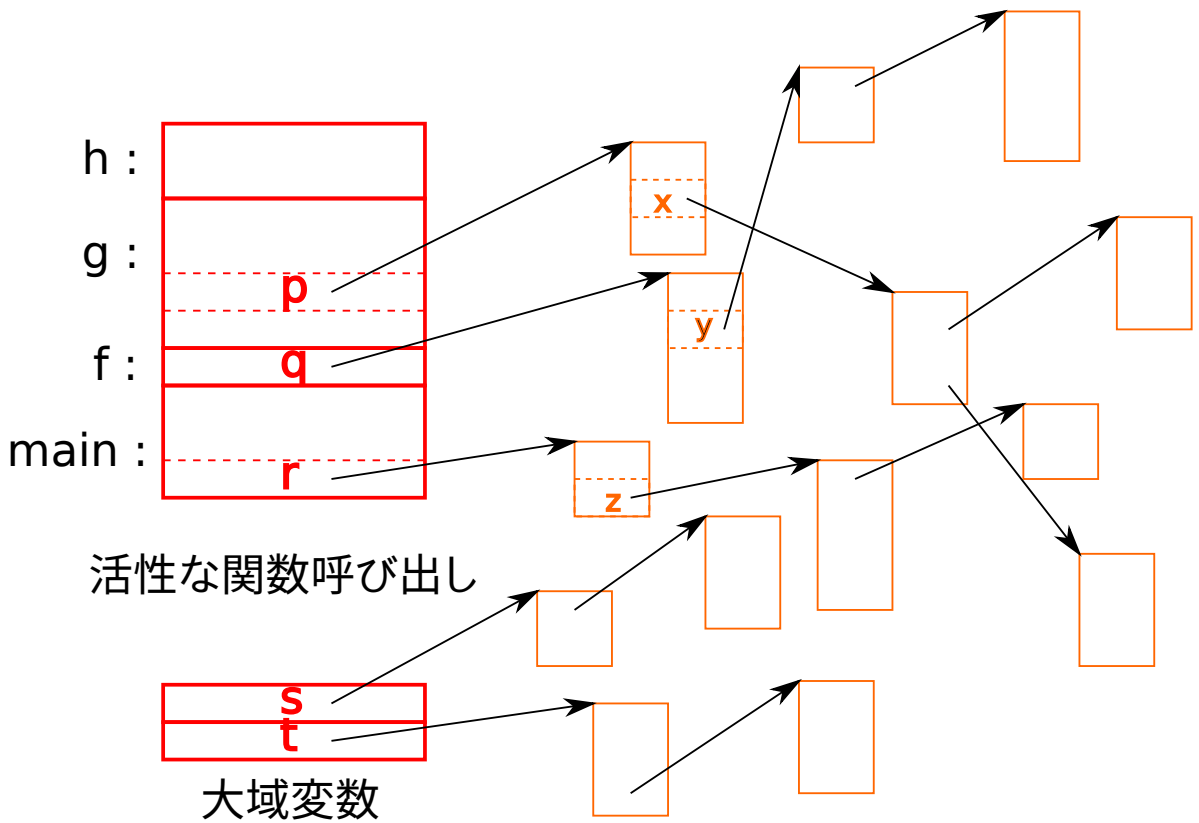
活性な関数呼び出し

大域変数

アクセスされる「かも」しれないデータ

- 大域変数
- 現在活性な (始まったが終了していない) 関数呼び出しの局所変数
- それらからポインタをたどって辿り着くデータ

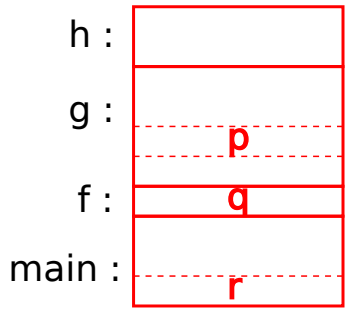
```
1  int * s;
2  int * t;
3  void h() { ... }
4  void g() {
5    ...
6    h();
7    ... = p->x ...   }
8  void f() {
9    ...
10   g()
11   ... = q->y ... }
12 int main() {
13   ...
14   f()
15   ... = r->z ... }
```



Variables that “may be” accessed

- global variables
- local variables of active function calls (calls that have started but have not finished)

```
1  int * s;
2  int * t;
3  void h() { ... }
4  void g() {
5    ...
6    h();
7    ... = p->x ...  }
8  void f() {
9    ...
10   g()
11   ... = q->y ... }
12 int main() {
13   ...
14   f()
15   ... = r->z ... }
```



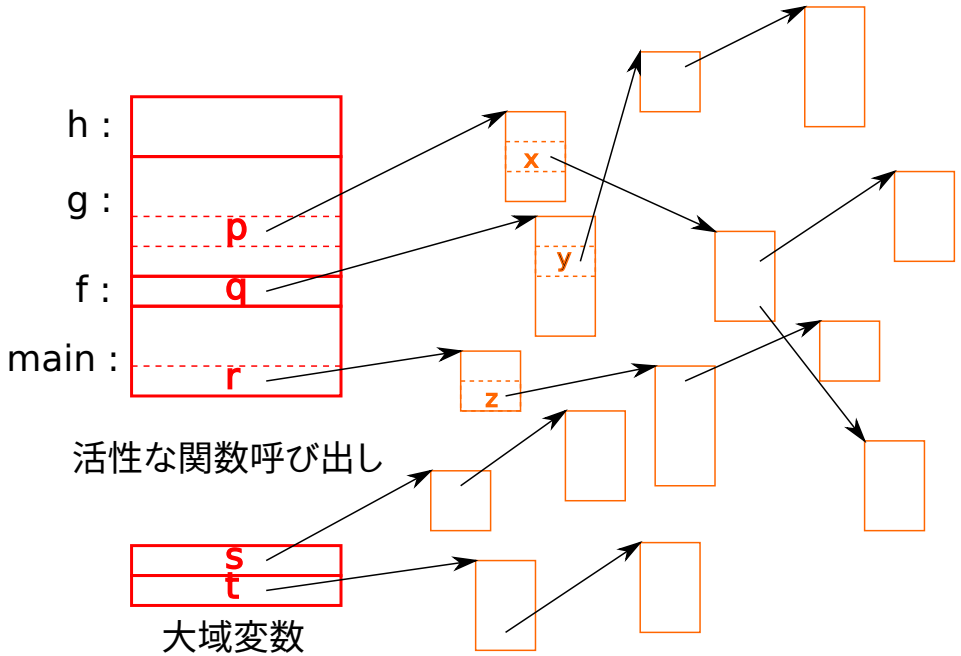
活性な関数呼び出し

大域変数

Variables that “may be” accessed

- global variables
- local variables of active function calls (calls that have started but have not finished)
- variables reachable from them by traversing pointers

```
1  int * s;
2  int * t;
3  void h() { ... }
4  void g() {
5    ...
6    h();
7    ... = p->x ...  }
8  void f() {
9    ...
10   g()
11   ... = q->y ... }
12 int main() {
13   ...
14   f()
15   ... = r->z ... }
```



GC の基本原理 (と用語)

- オブジェクト: メモリ割り当て・回収の単位 (C ならば malloc)
- ルート: 大域変数や現在活性中の関数の局所変数など，ポインタをひとつもたどらずにアクセスされうるオブジェクト
- 到達可能 (reachable): ポインタをたどってたどりつける
- 生きている (live)，死んでいる (dead): 今後アクセスされうる，され得ない
- ゴミ: 死んでいるオブジェクト
- collector: GC をするプログラム (やスレッド/プロセス)
- mutator: 要するにユーザプログラムのこと (vs. collector). 超 GC 目線な言葉. ユーザプログラムは「グラフを書き換える (mutate する) 人」

GC の基本原理:
ルートから到達不能なオブジェクトは死んでいる

The basic workings (and terminologies) of GC

- an object: the unit of automatic memory allocation/release (malloc in C; objects in Java; etc.)
- the root: objects accessible without traversing pointers, such as global variables and local variables of active function calls
- reachable objects: objects reachable from the root by traversing pointers
- live / dead objects: objects that {may be / never be} accessed in future
- garbage: dead objects
- collector: the program (or the thread/process) doing GC
- mutator: the user program (vs. collector). very GC-centric terminology, viewing the user program as someone simply "mutating" the graph of objects

the basic principle of GC:
objects unreachable from the root are dead

Contents

1. C/C++のメモリ管理 / Memory Management in C/C++
2. ガベージコレクション (GC) / Garbage Collection (GC)
3. 基本原理と用語 / Basics and Terminologies
   - 2 大方式 (走査型と参照カウント) / Two basic methods (traversing GC and reference counting)
   - GC の良し悪しの基準 / Criteria of evaluating GCs
   - 2 つの走査型 GC (マーク&スイープとコピー) / Two traversing GCs (mark&sweep and copying)
   - 走査型 GC のメモリ割り当てコスト (mark-cons 比) / Memory allocation cost of traversing GCs (mark-cons ratio)
4. C/C++用の GC ライブラリ / A GC library for C/C++

2 大 GC 方式

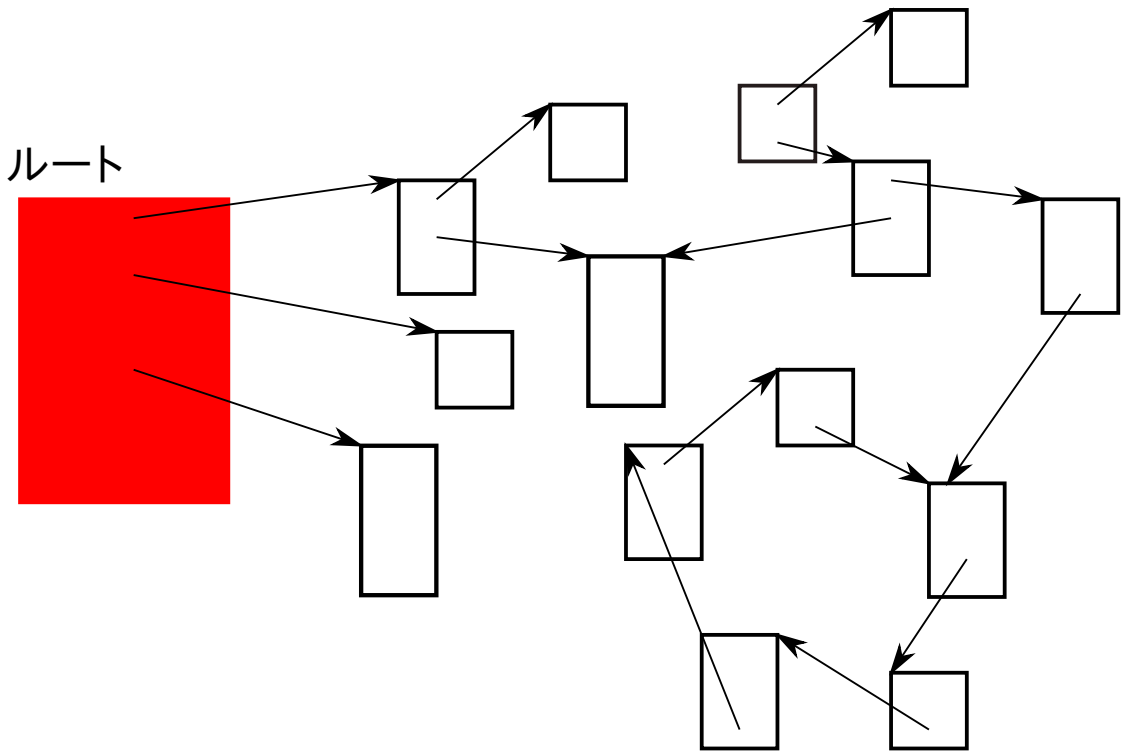
- 走査型 GC (traversing GC):
  - 素直にルートからポインタをたどり，ルートから到達可能なオブジェクトを発見
  - 発見されなかったものを回収
  - 2 タイプの走査型 GC
    - ★ マーク&スイープ GC (mark&sweep GC)
    - ★ コピー GC (copying GC)
- 参照カウント GC (reference counting GC):
  - あるオブジェクトを指すポインタの数 (参照数) を数えながら実行
  - 参照数が 0 になったものを回収
  - 注: 参照数が 0 → 到達不能
- 注: 巷では走査型 GC だけを GC と呼ぶこともあるよう

The two major GC methods

- traversing GC:
  - simply traverse pointers from the root, to find (or *visit*) objects reachable from the root
  - reclaim objects not visited
  - two basic traversing methods
    - ⋆ mark&sweep GC
    - ⋆ copying GC
- reference counting GC (or RC):
  - during execution, maintain the number of pointers (reference count) pointing to each object
  - reclaim an object when its reference count drops to zero
  - note: an object's reference count is zero → it's unreachable from the root
- remark: "GC" sometimes narrowly refers to traversing GC

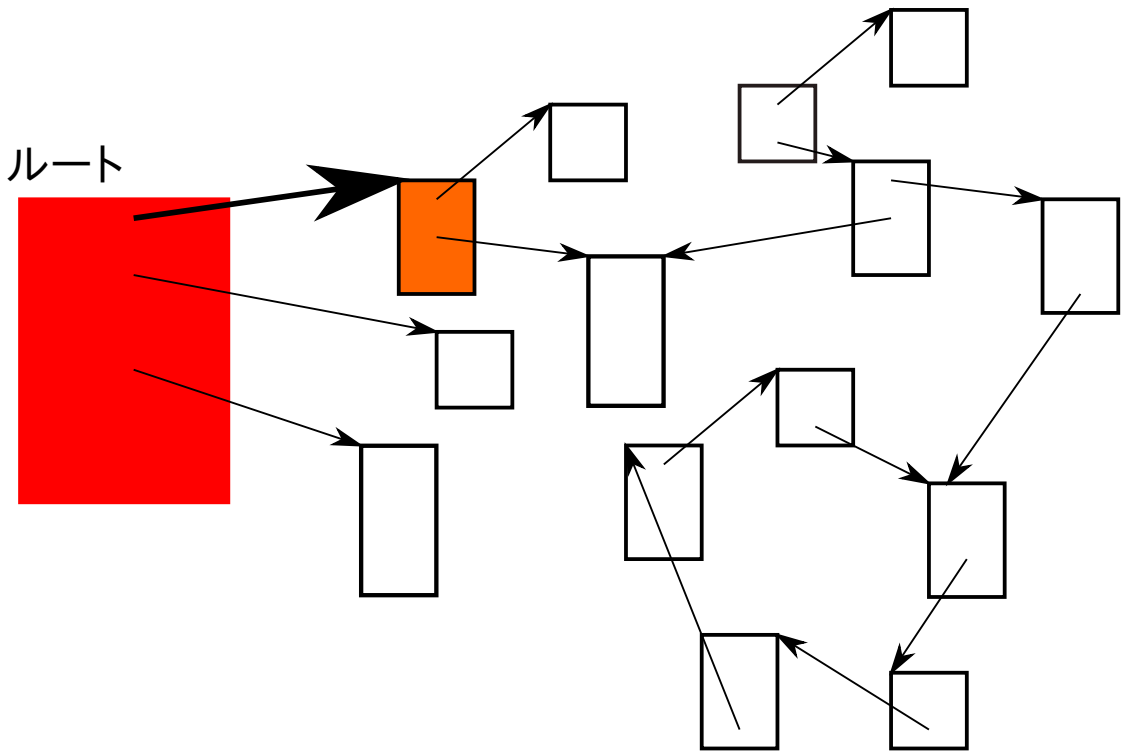
走査型GCの原理

- ルートからポインタをたどっていく
- これ以上たどるポインタがなくなったところで，訪問されていないオブジェクトがゴミ
- マーク&スイープとコピーの違いは後述



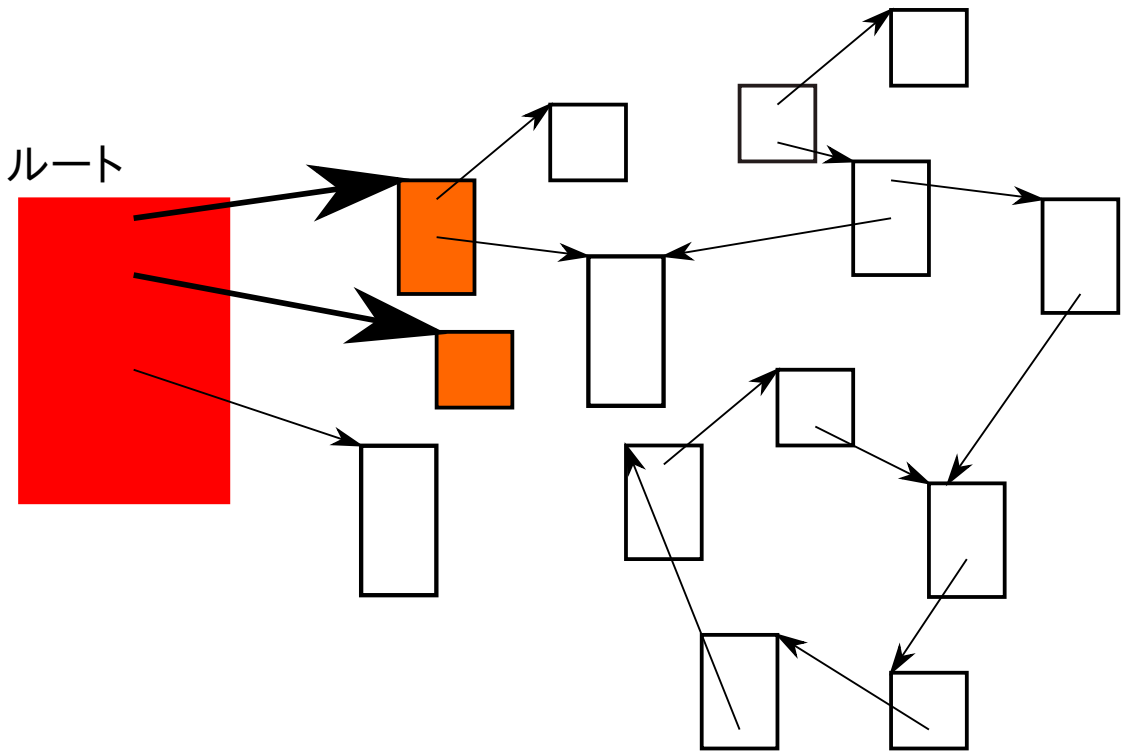
走査型GCの原理

- ルートからポインタをたどっていく
- これ以上たどるポインタがなくなったところで，訪問されていないオブジェクトがゴミ
- マーク&スイープとコピーの違いは後述



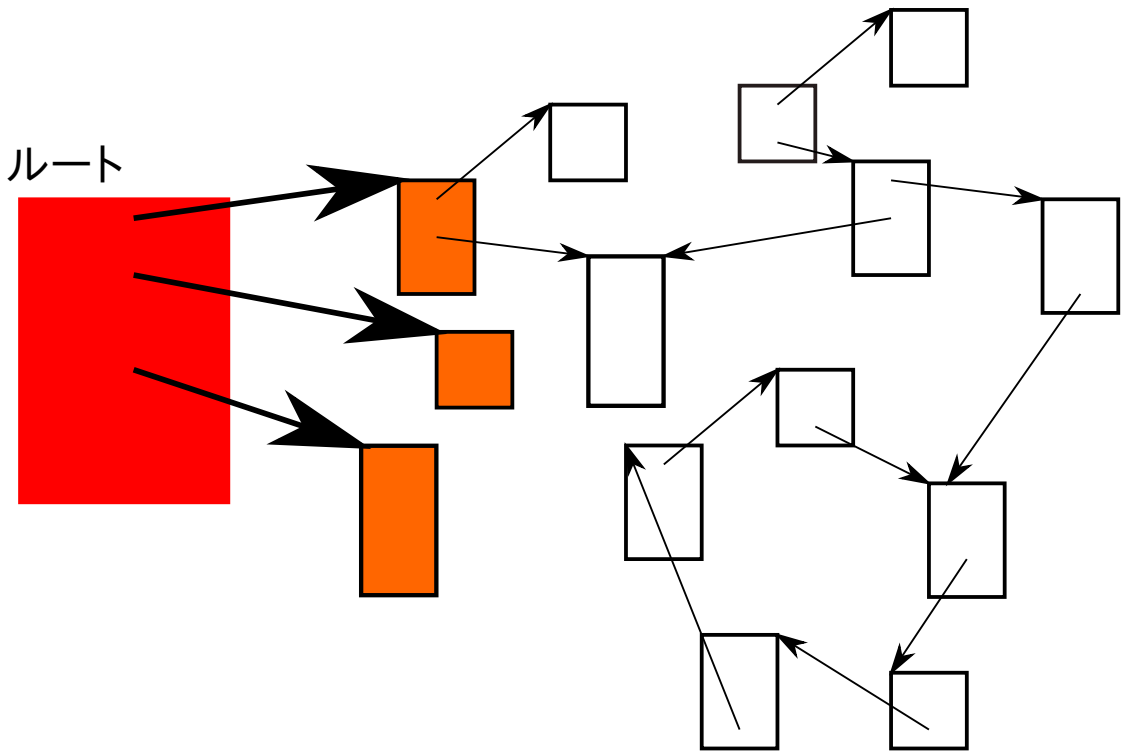
走査型GCの原理

- ルートからポインタをたどっていく
- これ以上たどるポインタがなくなったところで，訪問されていないオブジェクトがゴミ
- マーク&スイープとコピーの違いは後述



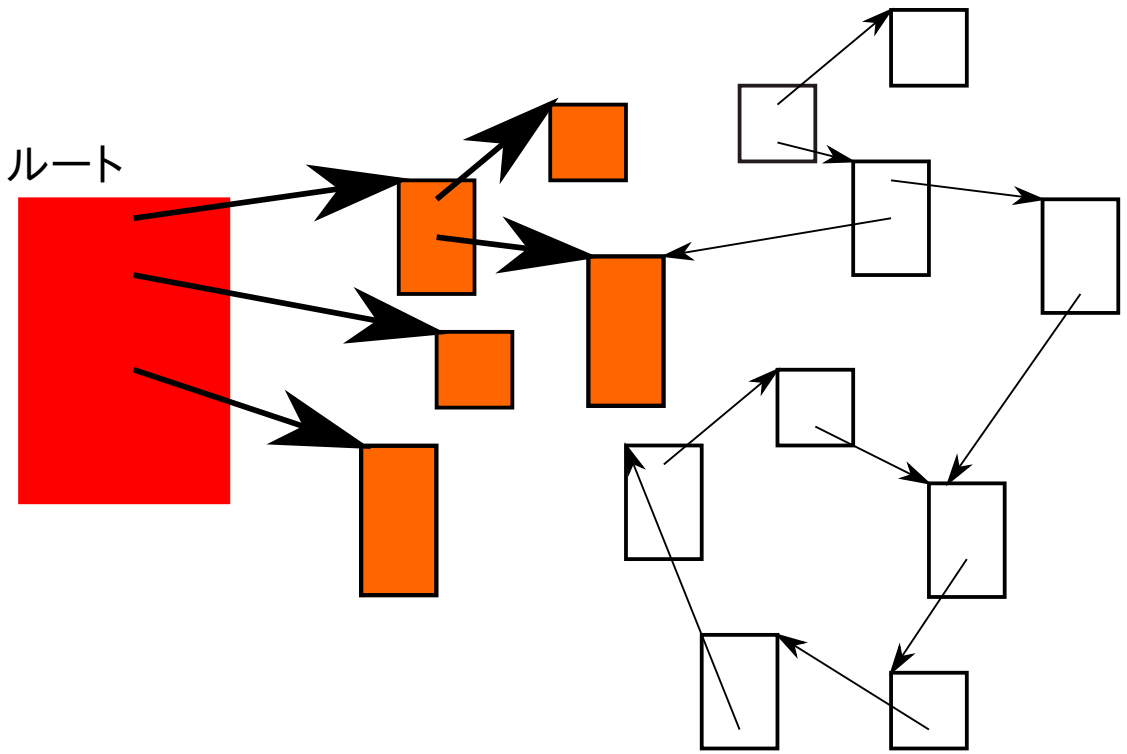
走査型GCの原理

- ルートからポインタをたどっていく
- これ以上たどるポインタがなくなったところで，訪問されていないオブジェクトがゴミ
- マーク&スイープとコピーの違いは後述



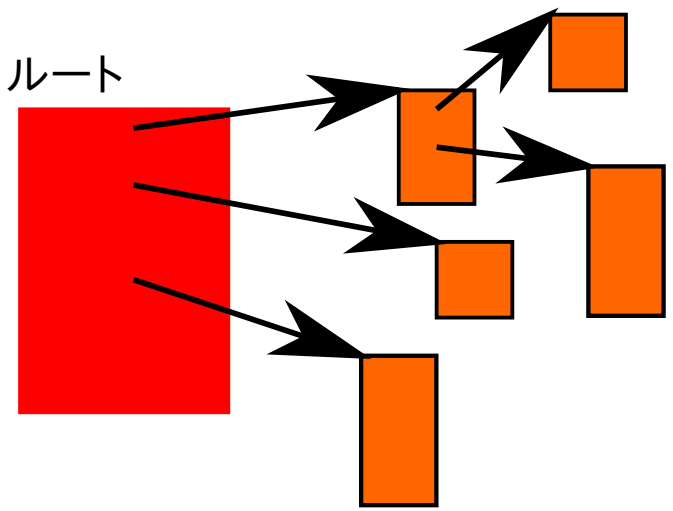
走査型GCの原理

- ルートからポインタをたどっていく
- これ以上たどるポインタがなくなったところで，訪問されていないオブジェクトがゴミ
- マーク&スイープとコピーの違いは後述



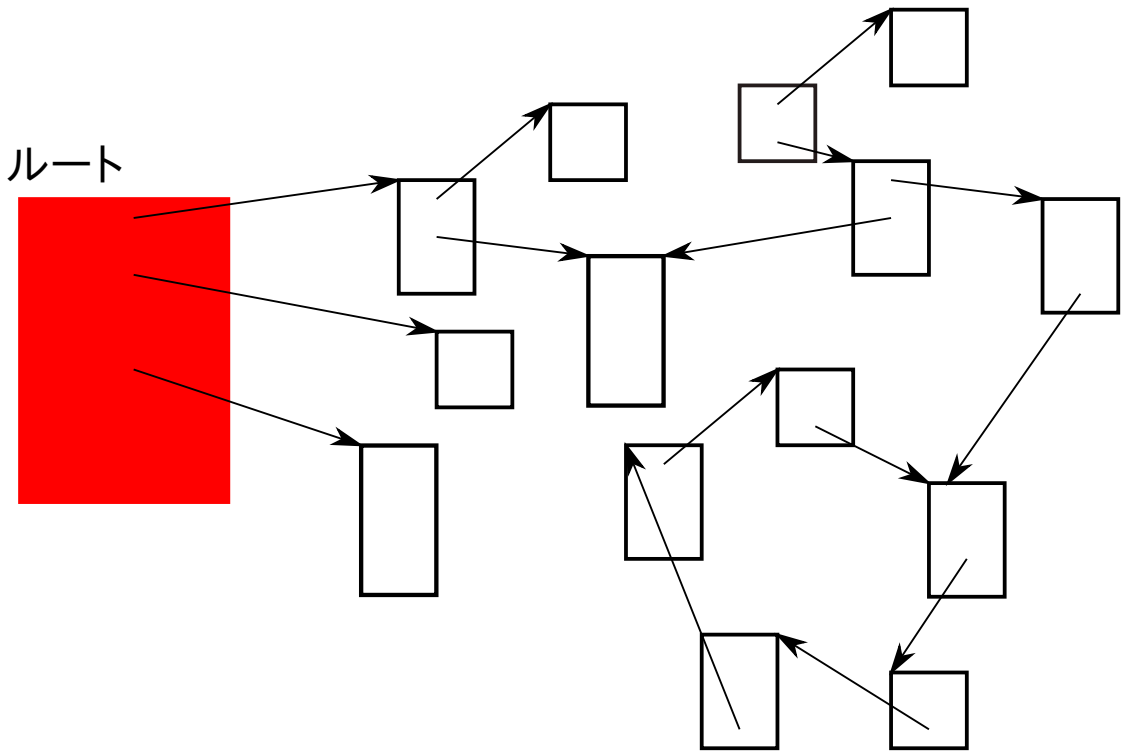
走査型GCの原理

- ルートからポインタをたどっていく
- これ以上たどるポインタがなくなったところで，訪問されていないオブジェクトがゴミ
- マーク&スイープとコピーの違いは後述



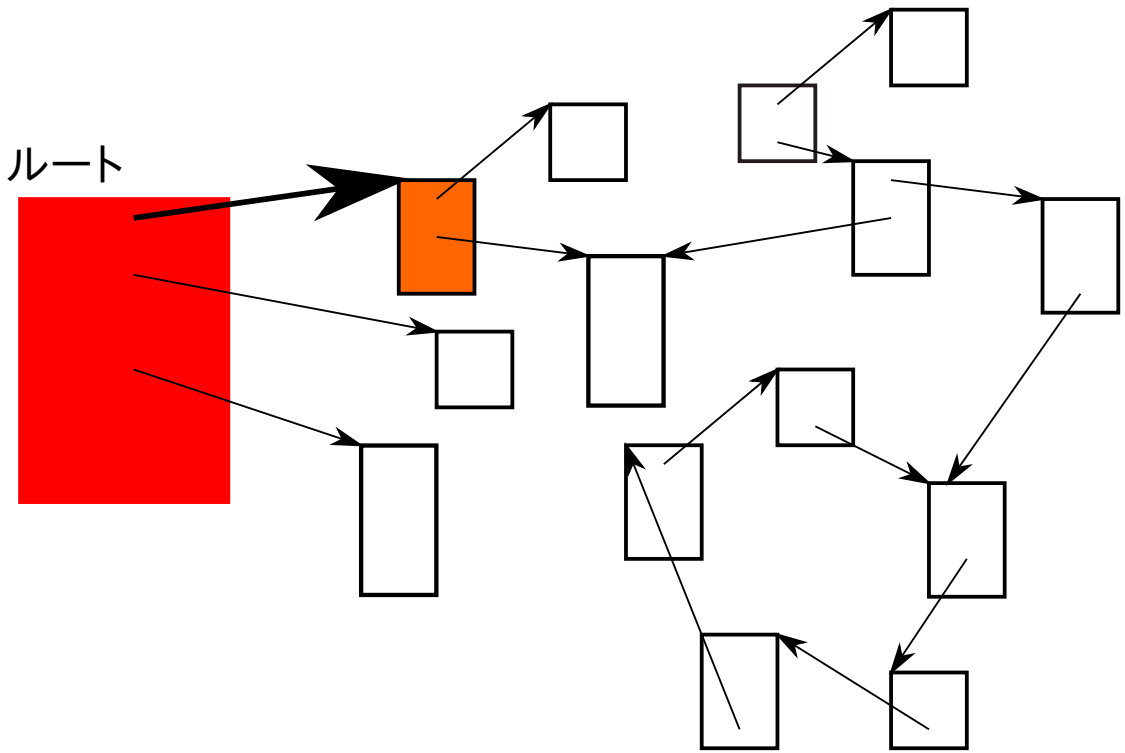
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



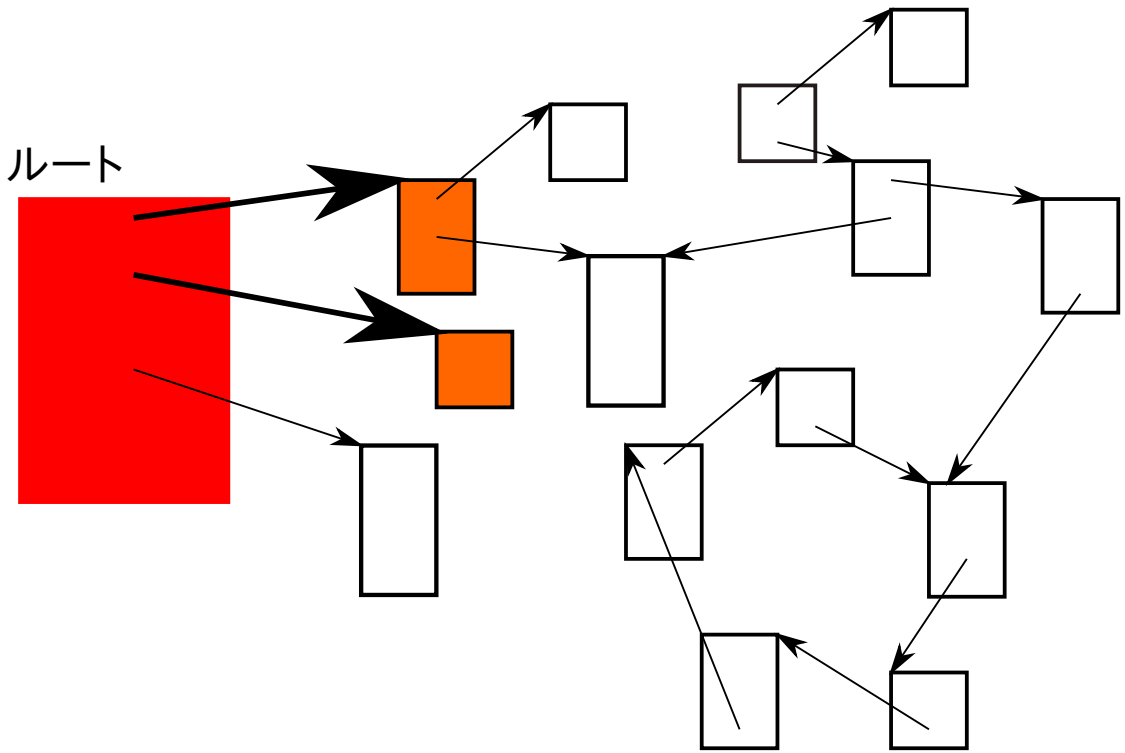
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



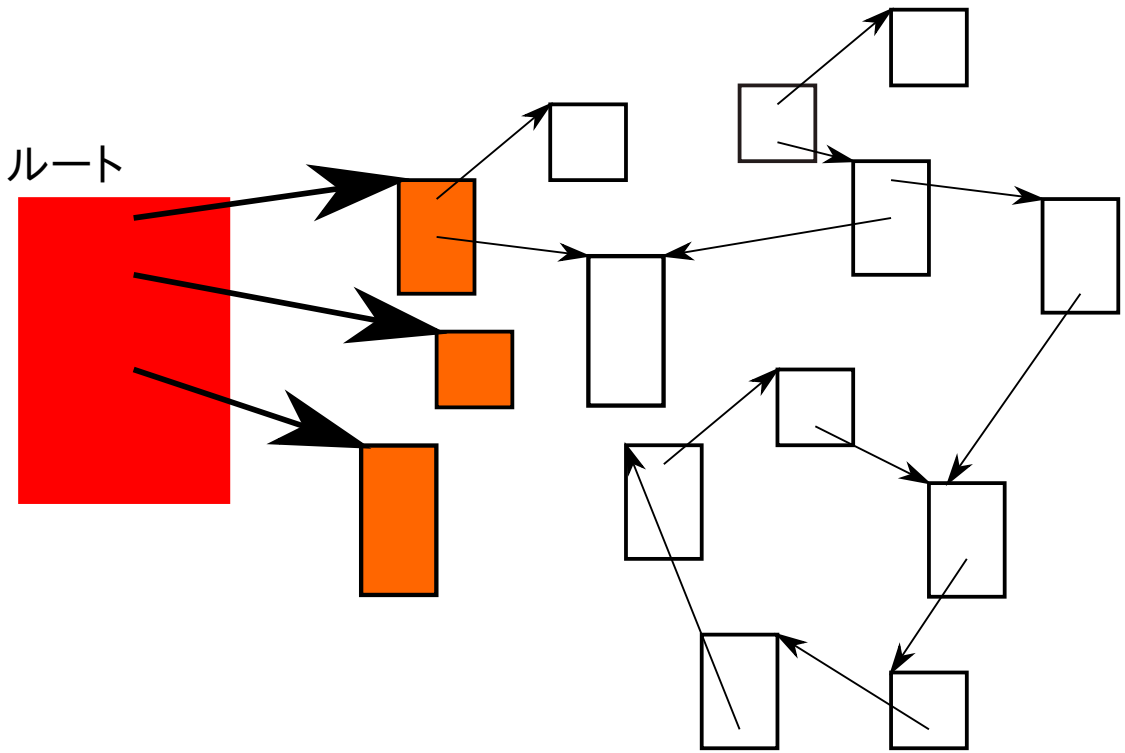
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



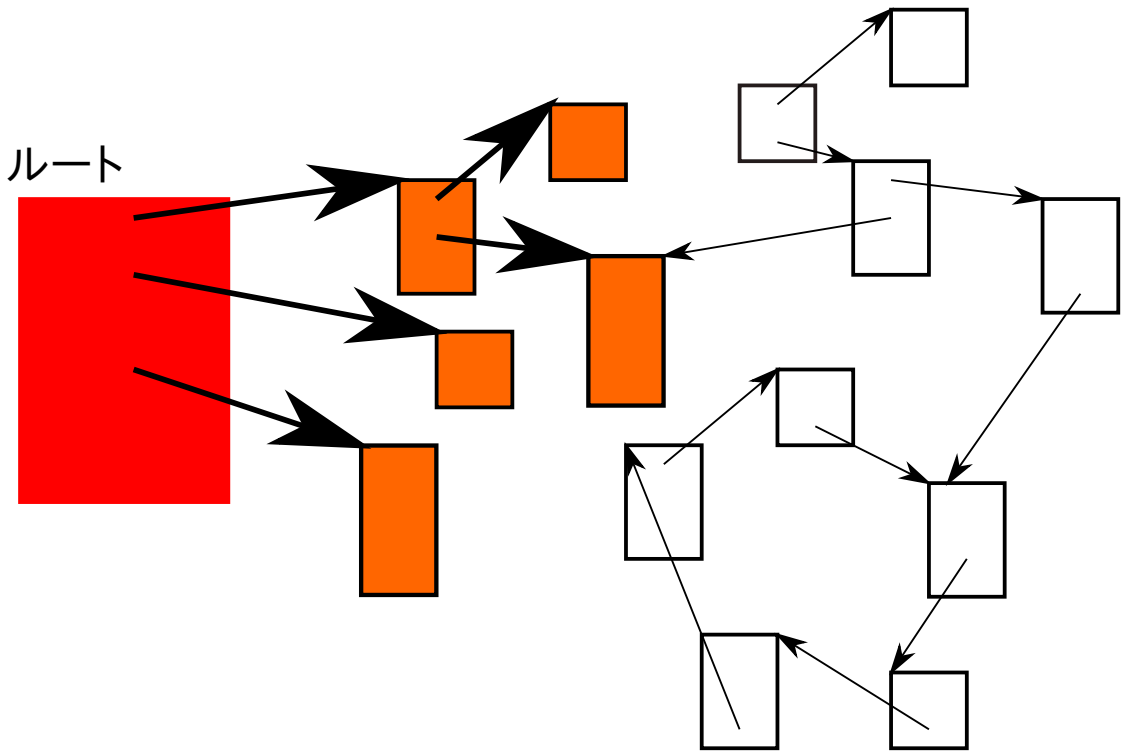
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



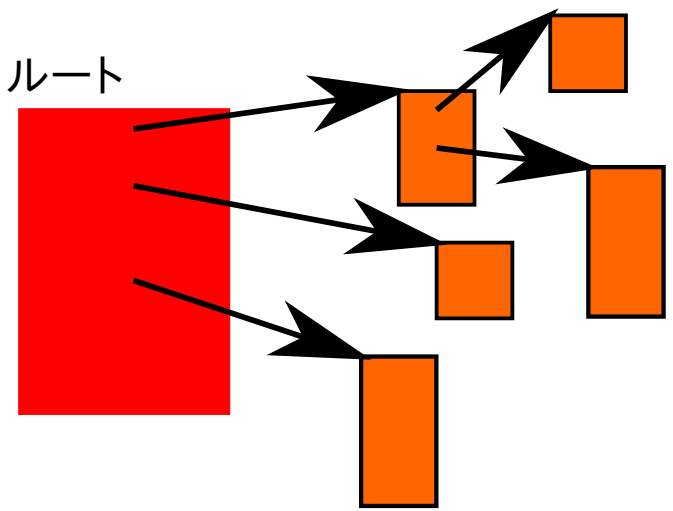
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



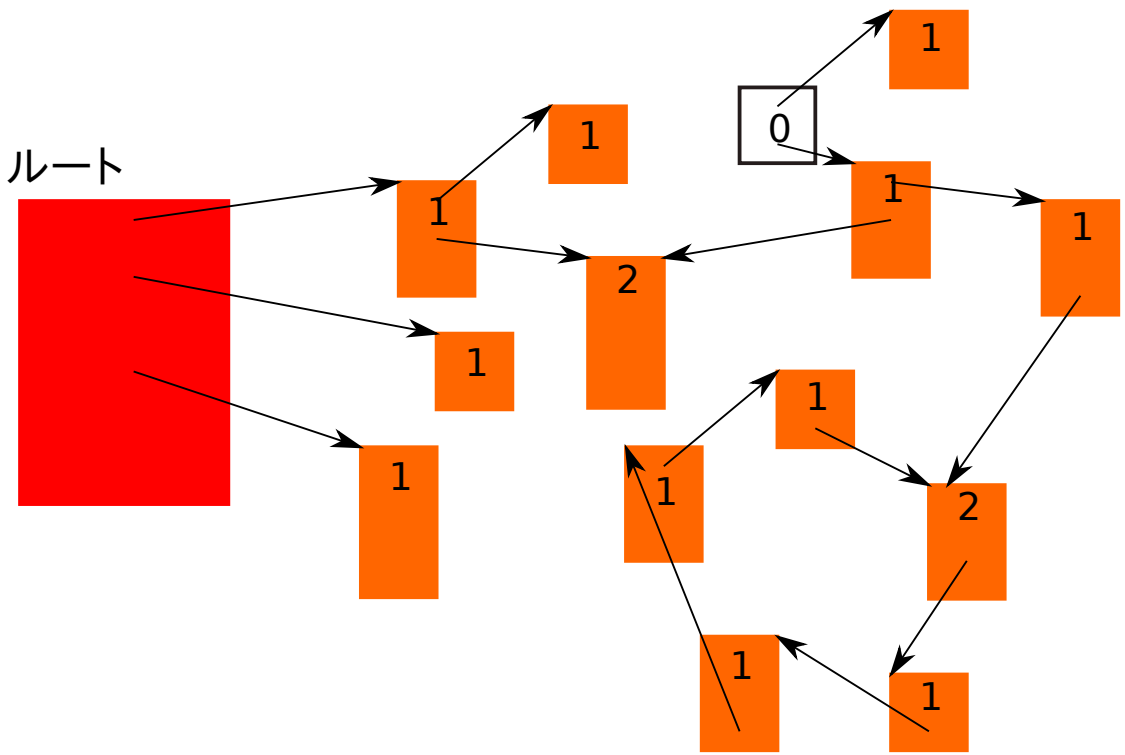
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



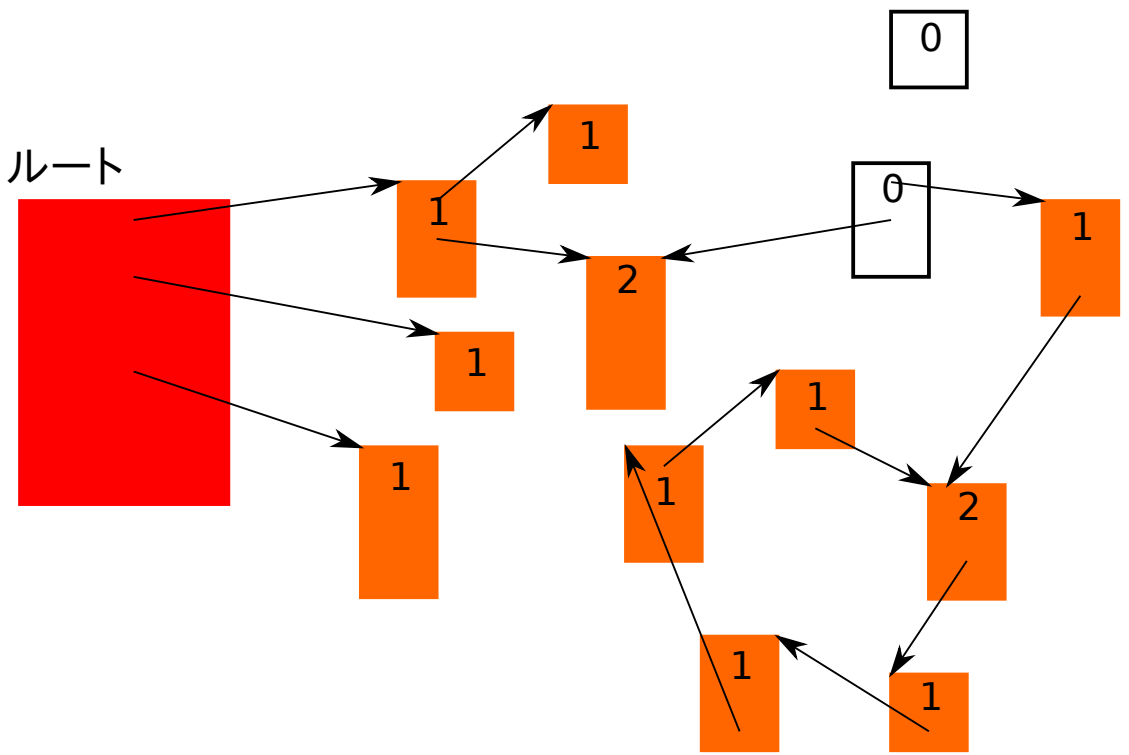
参照カウントGCの原理

- 各オブジェクトに参照数 (それを指すポインタの数) を付随
- ポインタの書き換え時に参照数更新; p = q; を実行 →
  - p に入っていたポインタが指すオブジェクトの参照数: 1 減る
  - q に入っているポインタが指すオブジェクトの参照数: 1 増える
- 参照数 0 になったものを回収 → 回収されたオブジェクト内のポインタが指していたオブジェクトの参照数が 1 減る



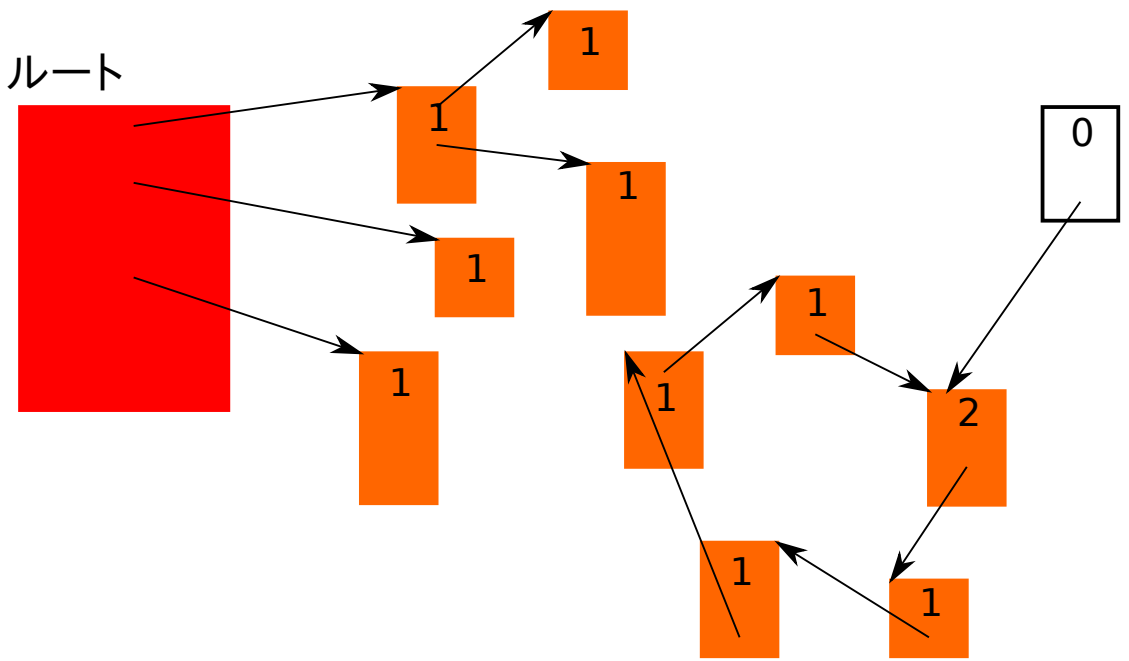
参照カウントGCの原理

- 各オブジェクトに参照数 (それを指すポインタの数) を付随
- ポインタの書き換え時に参照数更新; p = q; を実行 →
  - p に入っていたポインタが指すオブジェクトの参照数: 1 減る
  - q に入っているポインタが指すオブジェクトの参照数: 1 増える
- 参照数 0 になったものを回収 → 回収されたオブジェクト内のポインタが指していたオブジェクトの参照数が 1 減る



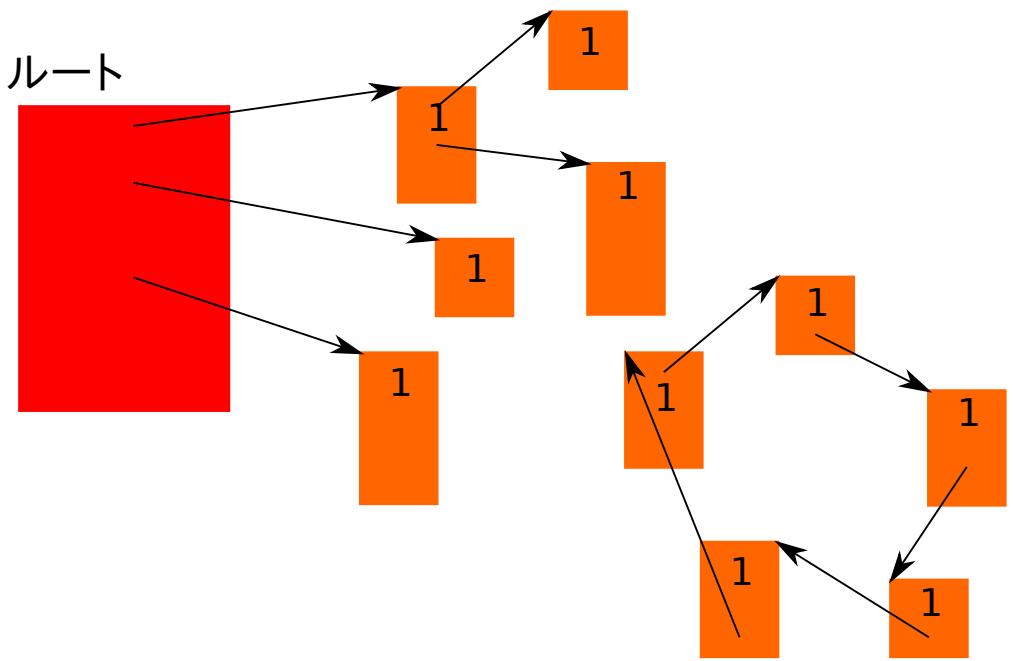
参照カウントGCの原理

- 各オブジェクトに参照数 (それを指すポインタの数) を付随
- ポインタの書き換え時に参照数更新; p = q; を実行 →
  - p に入っていたポインタが指すオブジェクトの参照数: 1 減る
  - q に入っているポインタが指すオブジェクトの参照数: 1 増える
- 参照数 0 になったものを回収 → 回収されたオブジェクト内のポインタが指していたオブジェクトの参照数が 1 減る



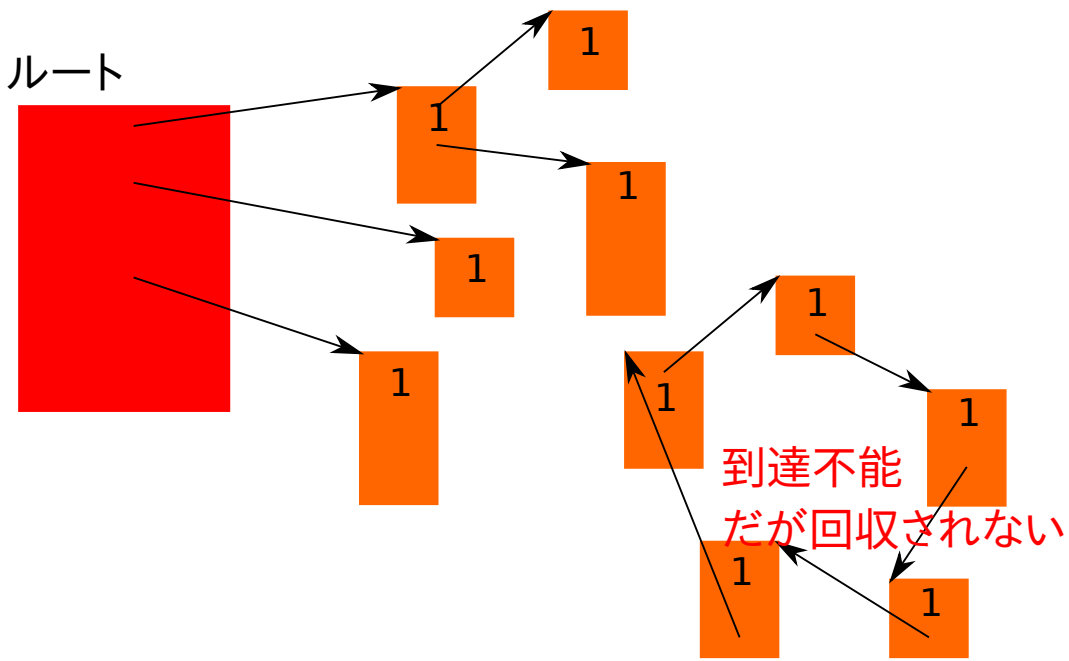
参照カウントGCの原理

- 各オブジェクトに参照数 (それを指すポインタの数) を付随
- ポインタの書き換え時に参照数更新; p = q; を実行 →
  - p に入っていたポインタが指すオブジェクトの参照数: 1 減る
  - q に入っているポインタが指すオブジェクトの参照数: 1 増える
- 参照数 0 になったものを回収 → 回収されたオブジェクト内のポインタが指していたオブジェクトの参照数が 1 減る



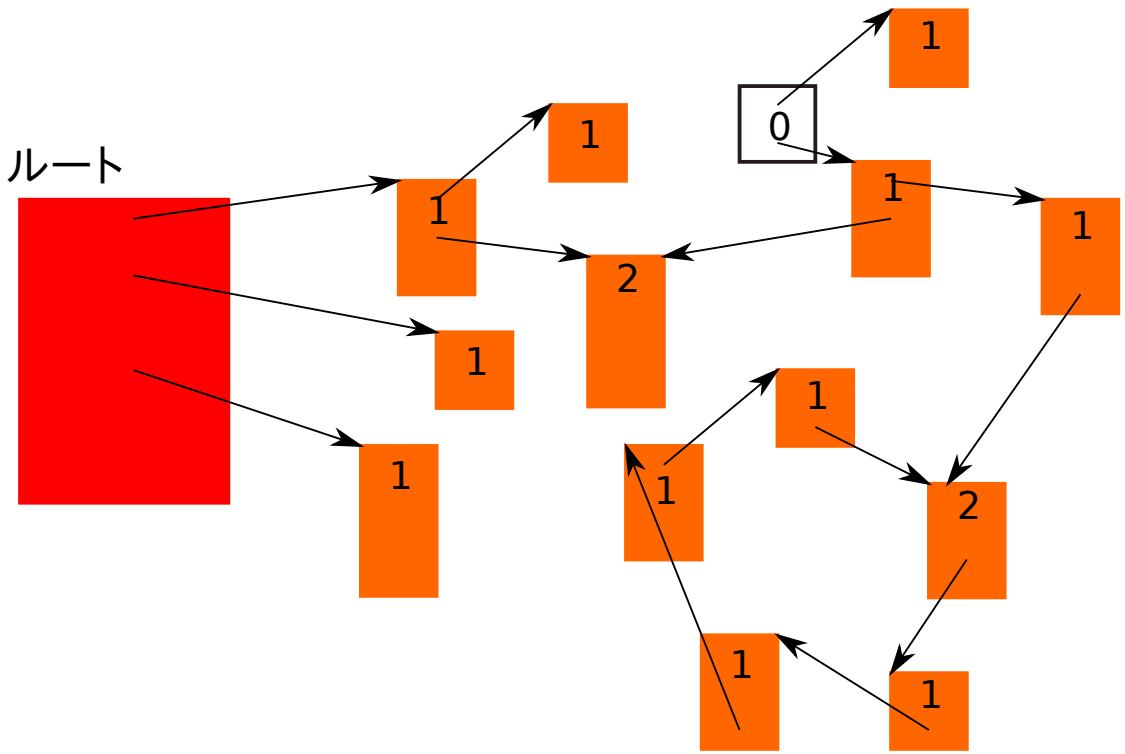
参照カウントGCの原理

- 各オブジェクトに参照数 (それを指すポインタの数) を付随
- ポインタの書き換え時に参照数更新; p = q; を実行 →
  - p に入っていたポインタが指すオブジェクトの参照数: 1 減る
  - q に入っているポインタが指すオブジェクトの参照数: 1 増える
- 参照数 0 になったものを回収 → 回収されたオブジェクト内のポインタが指していたオブジェクトの参照数が 1 減る



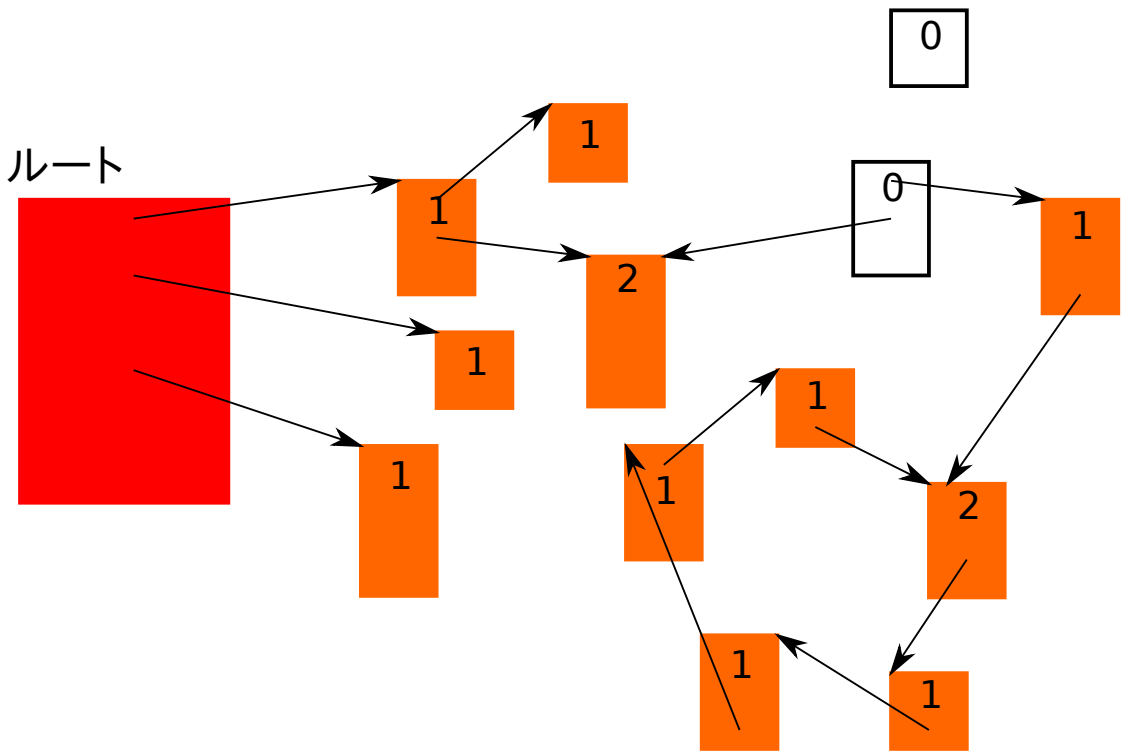
How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` →
  - the RC of the object `p` points to `-= 1`
  - the RC of the object `q` points to `+= 1`
- reclaim an object when its RC drops to zero → RCs of objects pointed to by the now reclaimed object decrease



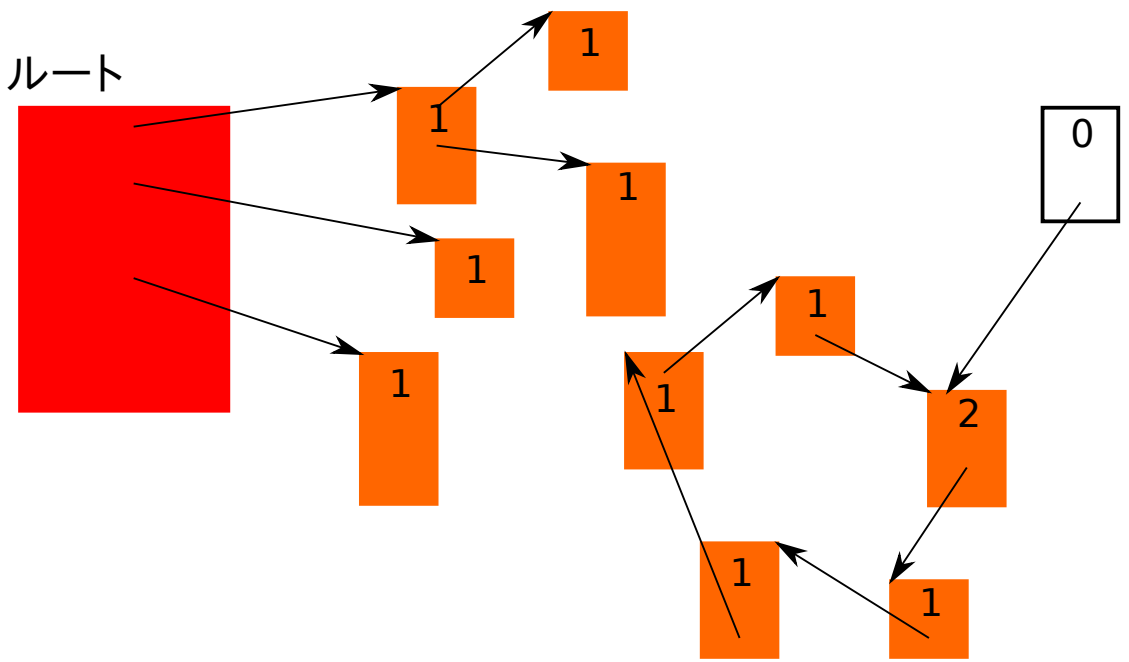
How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` $\rightarrow$
  - the RC of the object `p` points to `-= 1`
  - the RC of the object `q` points to `+= 1`
- reclaim an object when its RC drops to zero $\rightarrow$ RCs of objects pointed to by the now reclaimed object decrease



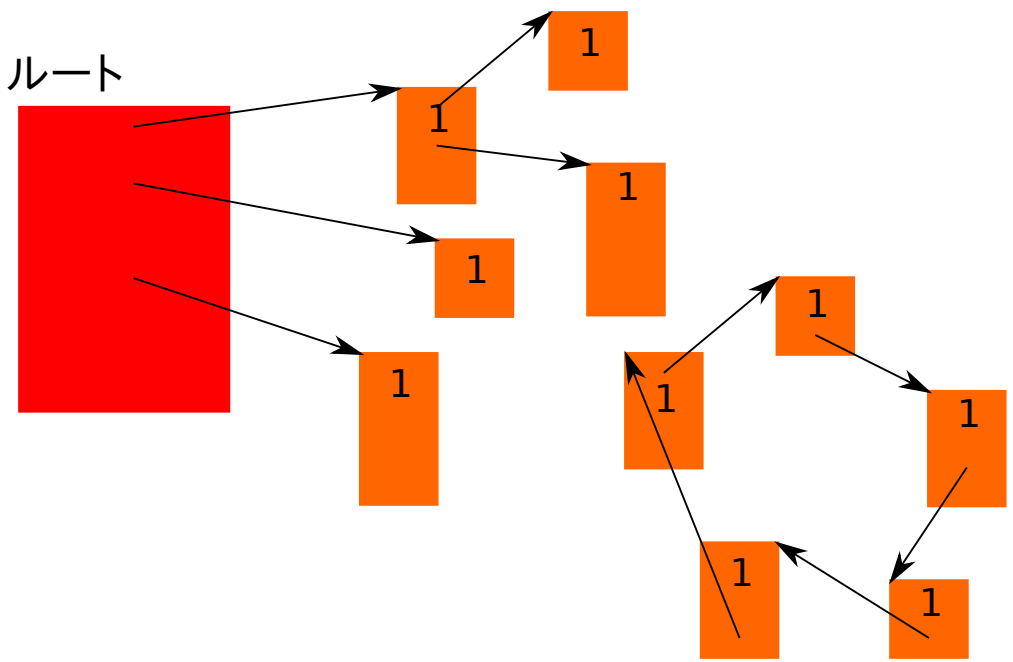
How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` →
  - the RC of the object `p` points to `-= 1`
  - the RC of the object `q` points to `+= 1`
- reclaim an object when its RC drops to zero → RCs of objects pointed to by the now reclaimed object decrease



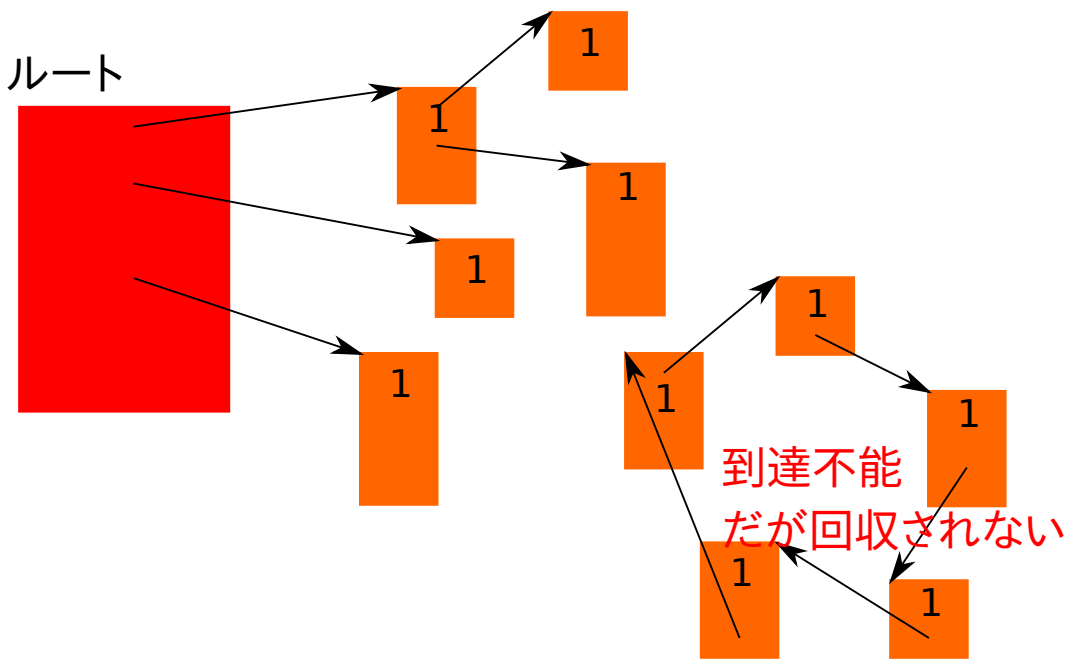
How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` →
  - the RC of the object `p` points to `-= 1`
  - the RC of the object `q` points to `+= 1`
- reclaim an object when its RC drops to zero → RCs of objects pointed to by the now reclaimed object decrease



How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` →
  - the RC of the object `p` points to `-= 1`
  - the RC of the object `q` points to `+= 1`
- reclaim an object when its RC drops to zero → RCs of objects pointed to by the now reclaimed object decrease



参照数が変化するのは

- pointer update `p = q; p->f = q;` etc.
- variables go out of scope

When an RC changes

- a pointer is updated `p = q;` `p->f = q;` etc.
- a function gets called

```
1  int main() {
2    object * q = ...;
3    f(q);
4  }
```

- a function returns or a variable goes out of scope

```
1  f(object * p) {
2    object * r = ...;
3
4    return ...; /* RC of p and r should decrease */
5  }
```

- etc. any point pointer variables get copied / become no longer used

Contents

1. C/C++のメモリ管理 / Memory Management in C/C++
2. ガベージコレクション (GC) / Garbage Collection (GC)
3. 基本原理と用語 / Basics and Terminologies
   - 2 大方式 (走査型と参照カウント) / Two basic methods (traversing GC and reference counting)
   - GC の良し悪しの基準 / Criteria of evaluating GCs
   - 2 つの走査型 GC (マーク&スイープとコピー) / Two traversing GCs (mark&sweep and copying)
   - 走査型 GC のメモリ割り当てコスト (mark-cons 比) / Memory allocation cost of traversing GCs (mark-cons ratio)
4. C/C++用の GC ライブラリ / A GC library for C/C++

GCの良し悪しの基準

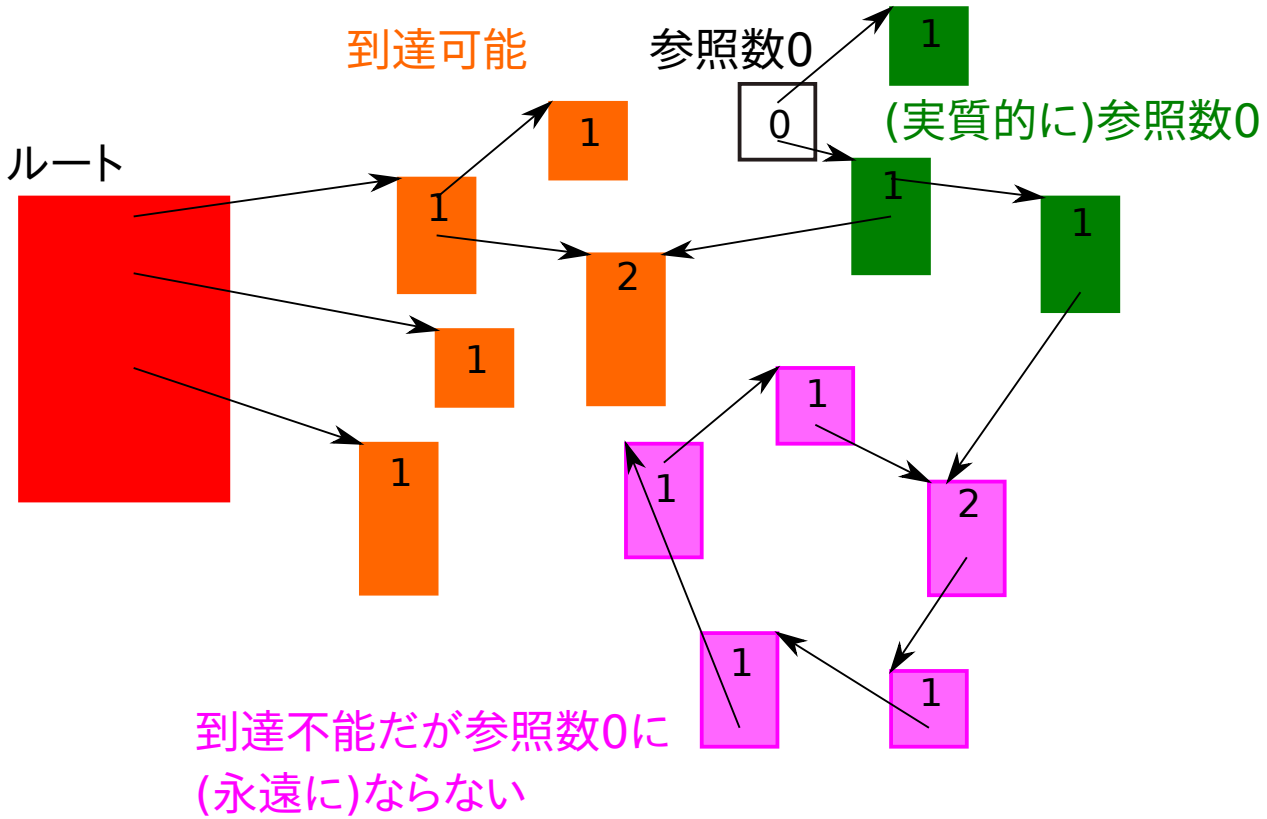
1. 正確さ:
   - 回収可能なゴミの範囲が広いか
2. メモリ割り当てコスト:
   - メモリ割当をするのに必要な (GC を含めた) 仕事
3. mutator オーバーヘッド:
   - GC が機能するために mutator に課されるオーバーヘッドが少ないか
4. 停止時間 (pause time):
   - GC が機能するために mutator が (一時的に) 停止しなくてはならない時間が短いか

Evaluating GCs

1. preciseness:
   - garbage that can be collected
2. memory allocation cost:
   - the work (including GC) required to allocate memory
3. mutator overhead:
   - the overhead imposed on the mutator for GC to function
4. pause time:
   - the (worst case) time the mutator has to (temporarily) suspend for GC to function

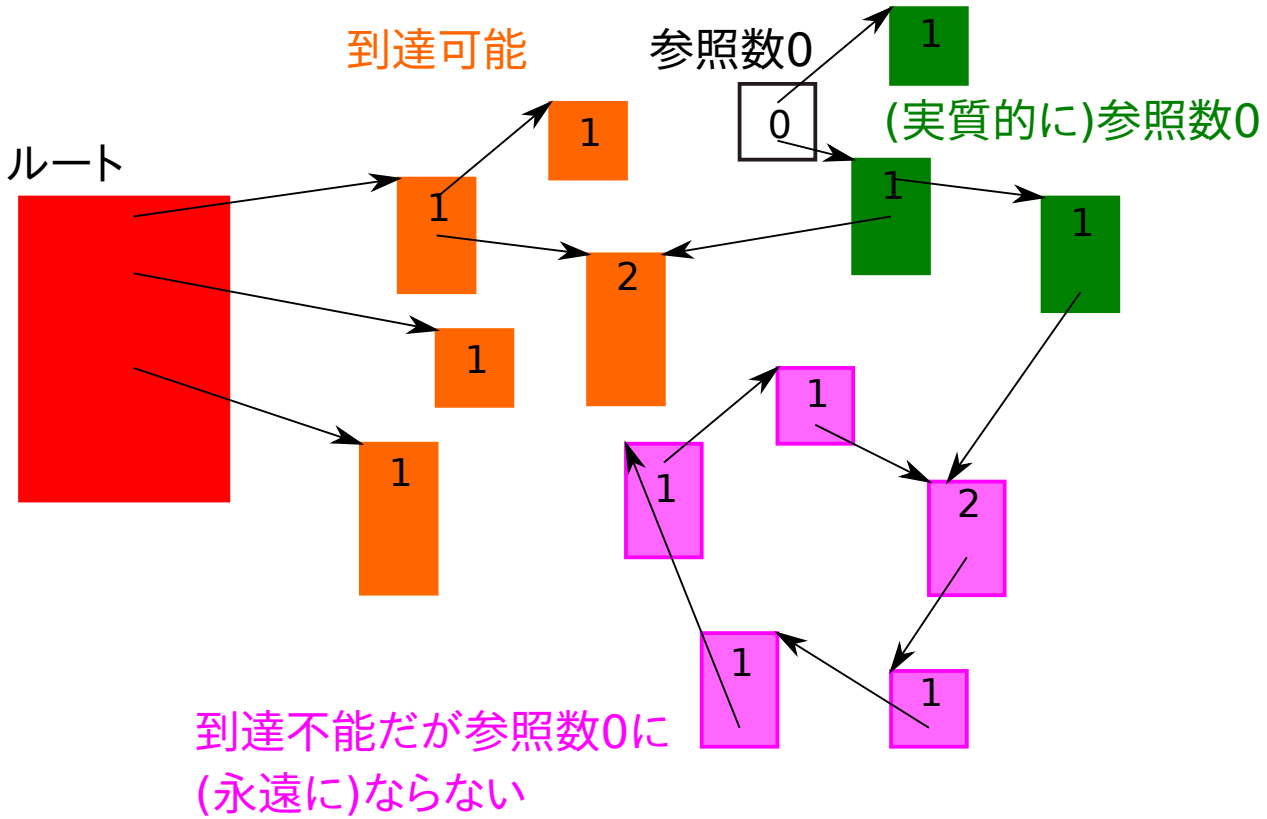
基準 1: 正確さ

- 参照カウントは循環ゴミを回収できない
- 参照カウント < 走査型 (走査型のほうが通常優れる)



Criteria #1: preciseness

- *reference counting cannot reclaim cyclic garbage*
- reference count < traversing GC (traversing GC is better)



基準2: メモリ割り当てコスト

- 一言で甲乙をつけるのは難 (詳しくは後述)

基準 2: メモリ割り当てコスト

- 一言で甲乙をつけるのは難 (詳しくは後述)
- 走査型:
  - コストは「到達可能だったオブジェクト」と「そうでなかった (回収できた) オブジェクト」の大きさの比で決まる (後述)
  - アプリと使用メモリ次第. 極小〜極大まで
  - 改善手段: 世代別 GC

基準 2: メモリ割り当てコスト

- 一言で甲乙をつけるのは難 (詳しくは後述)
- 走査型:
  - コストは「到達可能だったオブジェクト」と「そうでなかった (回収できた) オブジェクト」の大きさの比で決まる (後述)
  - アプリと使用メモリ次第. 極小〜極大まで
  - 改善手段: 世代別 GC
- 参照カウント:
  - 参照数 0 になったオブジェクトの回収コストは少ない&一定
  - メモリがかつかつでも一定 (優秀)

Criteria #2: memory allocation cost

- difficult to say in a few words (more details ahead)

Criteria #2: memory allocation cost

- difficult to say in a few words (more details ahead)
- traversing GC:
  - *the cost is determined by the ratio "reachable objects" / "unreachable (reclaimed) objects"* (later)
  - totally depending on apps and memory size, it can be anywhere from the minimum to infinity
  - an advanced technique: generational GC

Criteria #2: memory allocation cost

- difficult to say in a few words (more details ahead)
- traversing GC:
  - *the cost is determined by the ratio "reachable objects" / "unreachable (reclaimed) objects"* (later)
  - totally depending on apps and memory size, it can be anywhere from the minimum to infinity
  - an advanced technique: generational GC
- reference counting:
  - the cost of reclaiming an object once its RC drop to zero is small and constant
  - it is constant even if memory is scarce (good)

基準 3: 停止時間 (pause time)

- 参照カウント < 走査型 (参照カウントのほうが通常優れている)

基準 3: 停止時間 (pause time)

- 参照カウント < 走査型 (参照カウントのほうが通常優れている)
- 走査型:
  - 生きているオブジェクトを「全部一気に」たどり，たどられなかったオブジェクトを「全部一気に」回収
  - ドンと働いてドンと回収

基準 3: 停止時間 (pause time)

- 参照カウント < 走査型 (参照カウントのほうが通常優れている)
- 走査型:
  - 生きているオブジェクトを「全部一気に」たどり，たどられなかったオブジェクトを「全部一気に」回収
  - ドンと働いてドンと回収
  - たどってる最中に mutator に動かれると (= グラフを書き換えられると) 厄介
    - その厄介を何とかする方法: インクリメンタル GC
    - 世代別 GC にも似た効果あり

基準 3: 停止時間 (pause time)

- 参照カウント < 走査型 (参照カウントのほうが通常優れている)
- 走査型:
  - 生きているオブジェクトを「全部一気に」たどり，たどられなかったオブジェクトを「全部一気に」回収
  - ドンと働いてドンと回収
  - たどってる最中に mutator に動かれると (= グラフを書き換えられると) 厄介
    - その厄介を何とかする方法: インクリメンタル GC
    - 世代別 GC にも似た効果あり
- 参照カウント:
  - (mutator がポインタを書き換えた結果) 参照数 0 が発生したら，即回収可能
  - 発生したゴミをこまめに回収

Criteria #3: pause time

- reference counting < traversing GC (reference counting is better)

Criteria #3: pause time

- reference counting < traversing GC (reference counting is better)
- traversing GC:
  - traverse *all* live objects, *en masse*, and reclaim *all* unreached objects, *en masse*
  - do a whole bunch of work and get a whole bunch of memory

Criteria #3: pause time

- reference counting < traversing GC (reference counting is better)
- traversing GC:
  - traverse *all* live objects, *en masse*, and reclaim *all* unreached objects, *en masse*
  - do a whole bunch of work and get a whole bunch of memory
  - troubled if the mutator runs (= changes the graph of objects) during traversing
    - a solution: incremental GC
    - generational GCs mitigate it too

Criteria #3: pause time

- reference counting < traversing GC (reference counting is better)
- traversing GC:
  - traverse *all* live objects, *en masse*, and reclaim *all* unreached objects, *en masse*
  - do a whole bunch of work and get a whole bunch of memory
  - troubled if the mutator runs (= changes the graph of objects) during traversing
    - a solution: incremental GC
    - generational GCs mitigate it too
- reference counting:
  - when an object's RC drops to zero (as a result of mutator's action), it can be reclaimed immediately
  - reclaim garbage as they arise

基準 4: mutator オーバーヘッド

- 走査型 < 参照カウント (走査型のほうが優れている)
- 参照カウントは，ポインタの更新時のオーバーヘッド大

```
1 object * p, * q;
2 p = q;
```

は

```
1 if (p) p->rc--;
2 if (q) q->rc++;
3 p = q;
```

に．さらに，

  - マルチスレッドプログラムでは?
  - カウンタが溢れたらどうする? (そのためのチェックどうする?)

- 改善技術: 遅延参照カウント，sticky 参照カウント，1 bit 参照カウント
- 注: 走査型でも世代別 GC，インクリメンタル GC などではポインタ更新時のオーバーヘッドあり

Criteria #4: mutator overhead

- traversing < reference counting (traversing GC is better)
- reference counting has a large overhead for updating RCs

```
1 object * p, * q;
2 p = q;
```

will do:

```
1 if (p) p->rc--;
2 if (q) q->rc++;
3 p = q;
```

Moreover,

  - what about multithreaded programs?
  - what if the counter overflows (how to check it)?

- techniques: deferred reference counting, sticky reference counting, 1 bit reference counting
- remark: some traversing GCs (e.g., generational and incremental) add overhead to pointer updates too

Contents

1. C/C++のメモリ管理 / Memory Management in C/C++
2. ガベージコレクション (GC) / Garbage Collection (GC)
3. 基本原理と用語 / Basics and Terminologies
   - 2 大方式 (走査型と参照カウント) / Two basic methods (traversing GC and reference counting)
   - GC の良し悪しの基準 / Criteria of evaluating GCs
   - 2 つの走査型 GC (マーク&スイープとコピー) / Two traversing GCs (mark&sweep and copying)
   - 走査型 GC のメモリ割り当てコスト (mark-cons 比) / Memory allocation cost of traversing GCs (mark-cons ratio)
4. C/C++用の GC ライブラリ / A GC library for C/C++

マーク&スイープGCとコピーGC

到達可能なオブジェクトをどうするかの違い

- コピーGC: 別の (連続) 領域にコピーする

マーク&スイープGCとコピーGC

到達可能なオブジェクトをどうするかの違い

- コピーGC: 別の (連続) 領域にコピーする
- マーク&スイープGC: 「訪問済み」印をつけるだけ

mark&sweep GC vs. copying GC

they differ in what to do on reachable objects

- copying GC: copy them into a distinct (contiguous) region

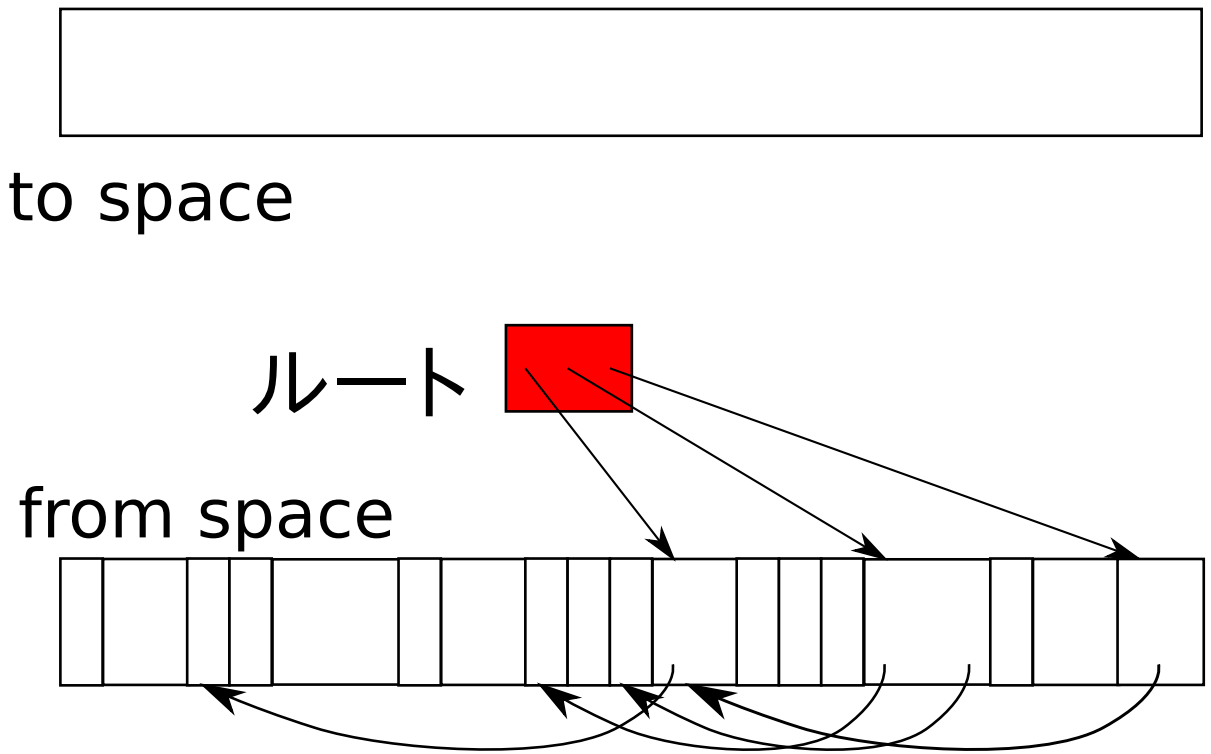
mark&sweep GC vs. copying GC

they differ in what to do on reachable objects

- copying GC: copy them into a distinct (contiguous) region
- mark&sweep GC: just mark them as "visited"

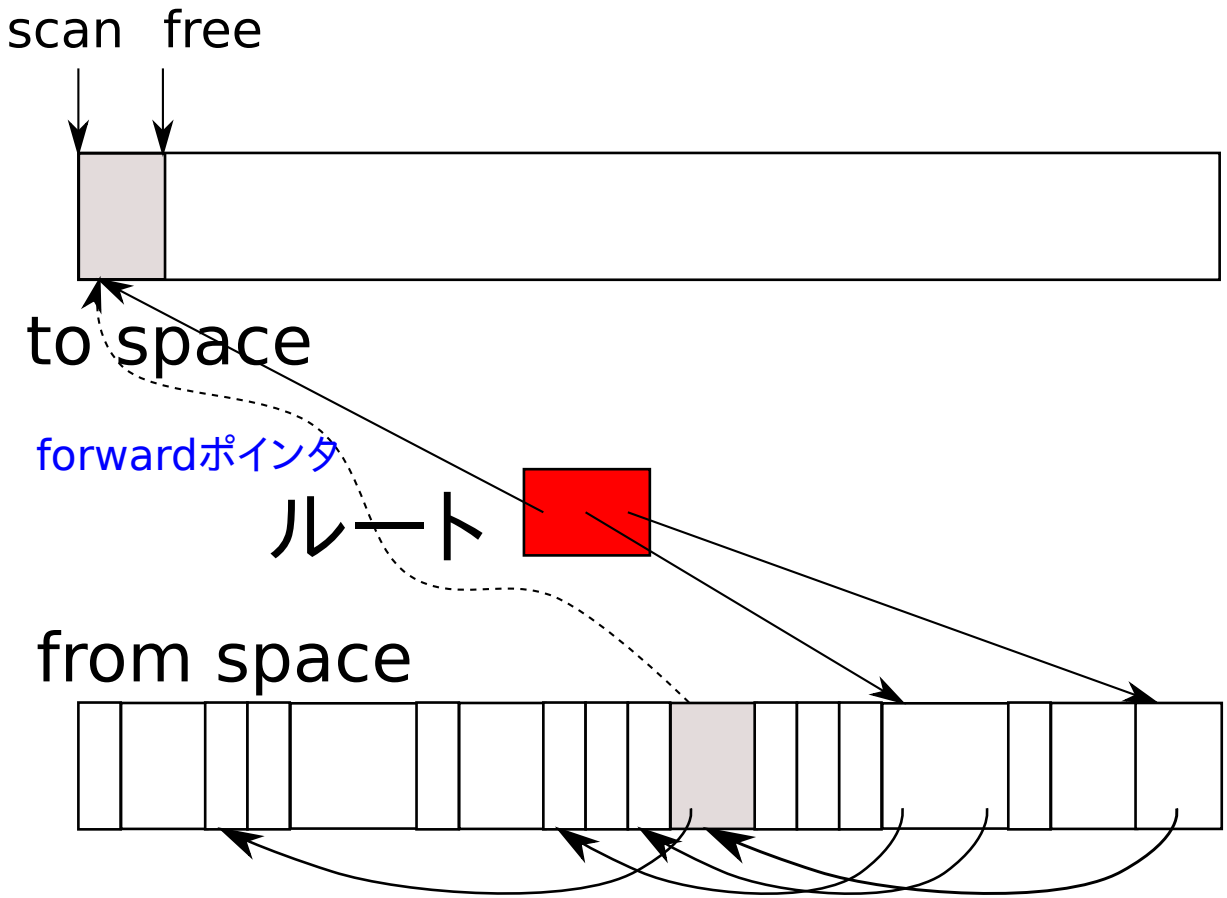
最も基本的なコピーGC図解

- 本質 ≈ グラフのコピー (≈ シリアライズ)
  - もともと同じポインタはコピー後も同じになるように
- semi-space GC (ルートから到達可能なオブジェクトをまるごと別領域へコピー)



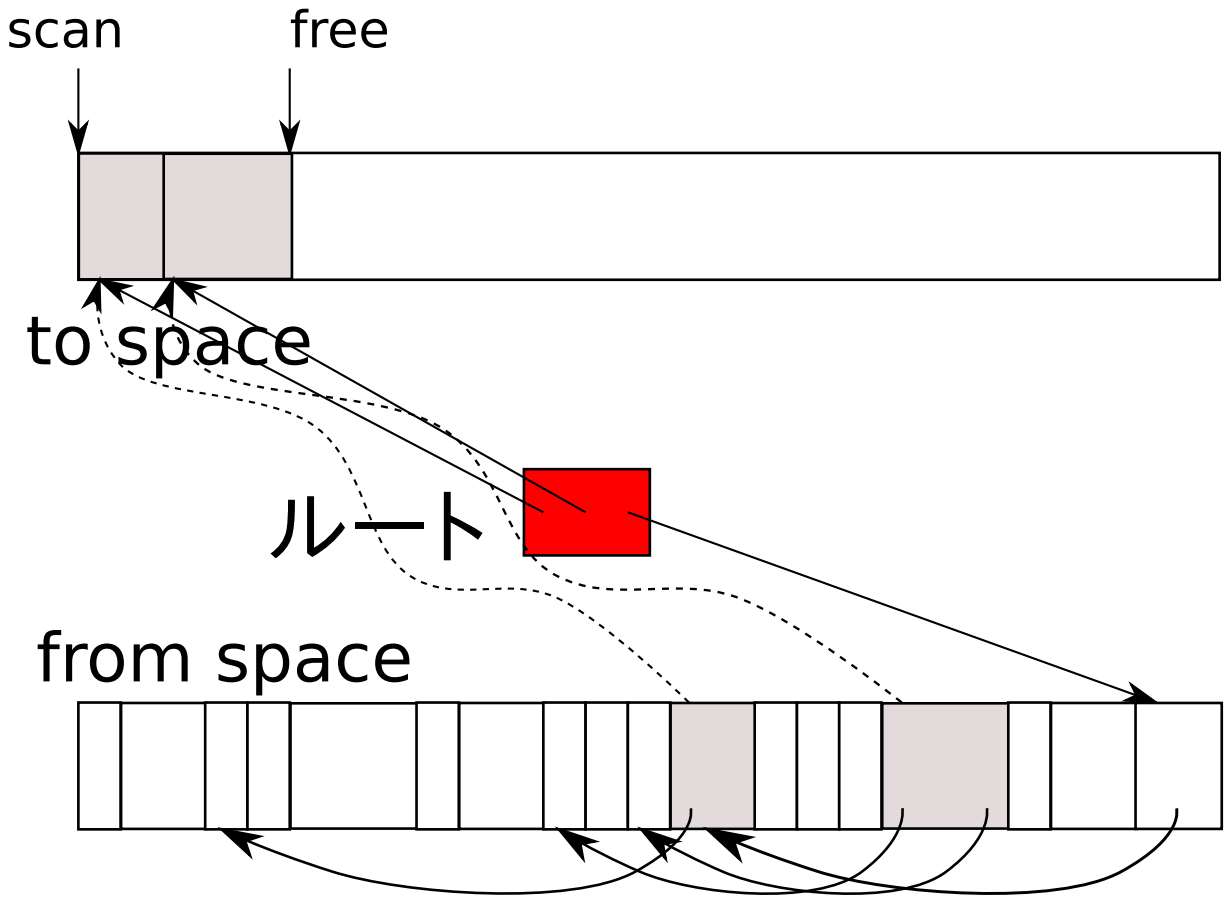
最も基本的なコピーGC図解

- 本質≈グラフのコピー(≈シリアライズ)
  - もともと同じポインタはコピー後も同じになるように
- semi-space GC (ルートから到達可能なオブジェクトをまるごと別領域へコピー)



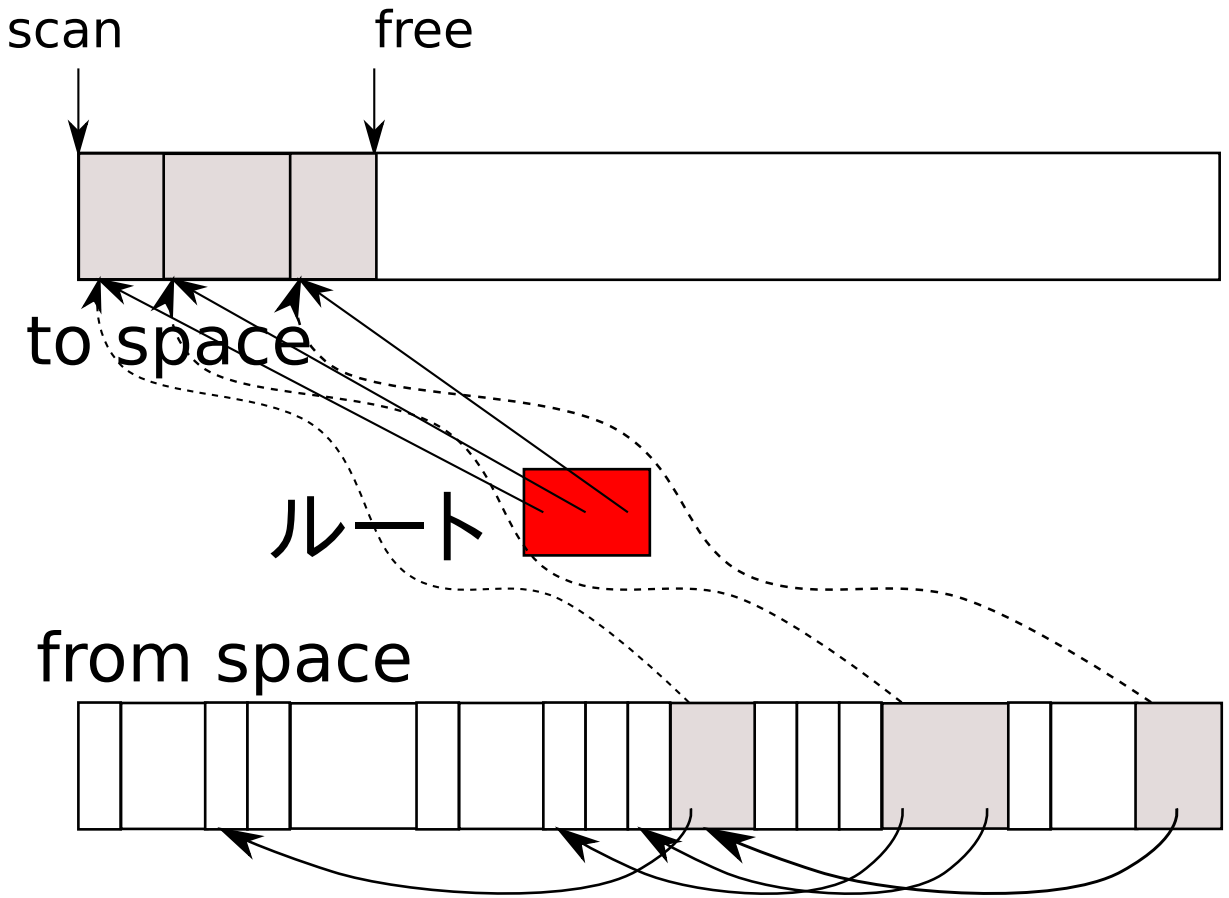
最も基本的なコピーGC図解

- 本質≈グラフのコピー(≈シリアライズ)
  - もともと同じポインタはコピー後も同じになるように
- semi-space GC (ルートから到達可能なオブジェクトをまるごと別領域へコピー)



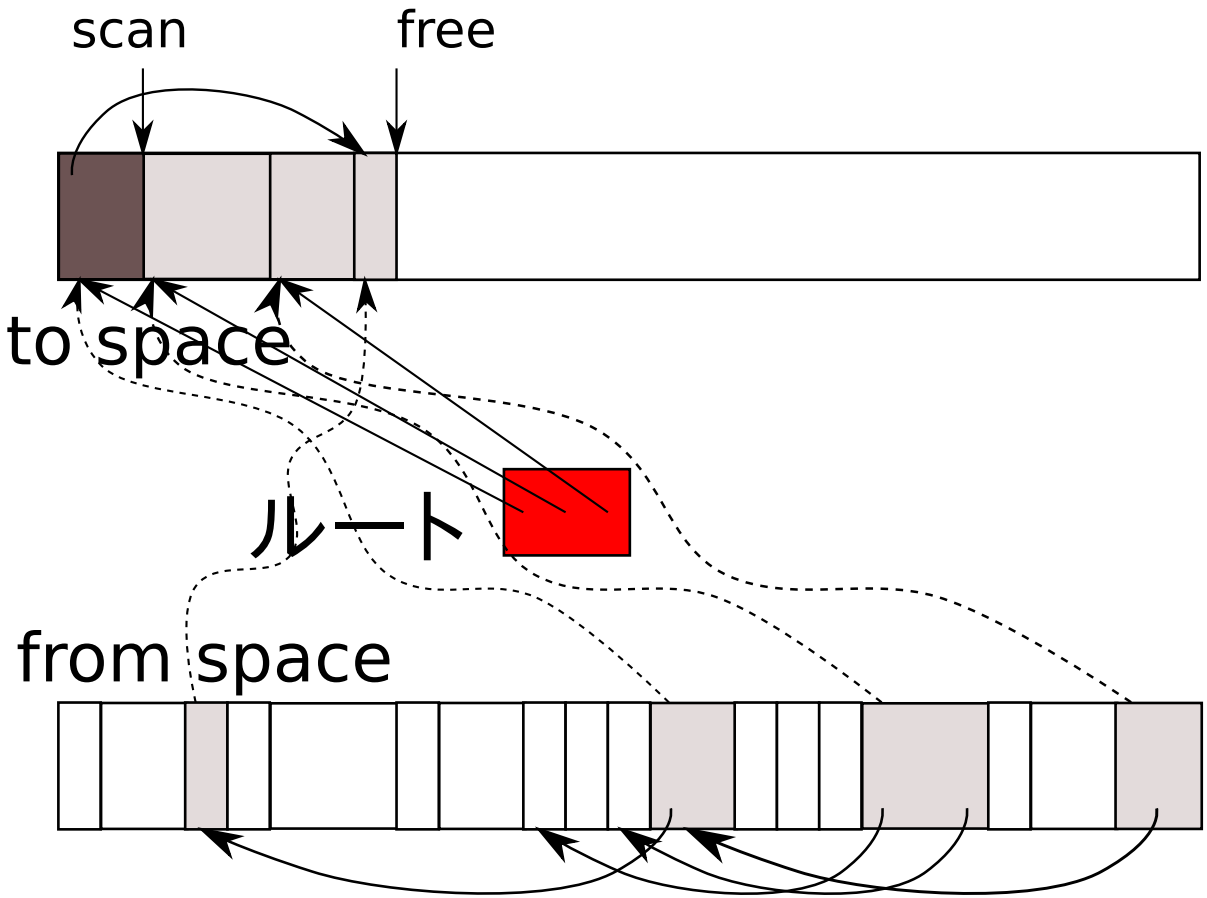
最も基本的なコピーGC図解

- 本質≈グラフのコピー(≈シリアライズ)
  - もともと同じポインタはコピー後も同じになるように
- semi-space GC (ルートから到達可能なオブジェクトをまるごと別領域へコピー)



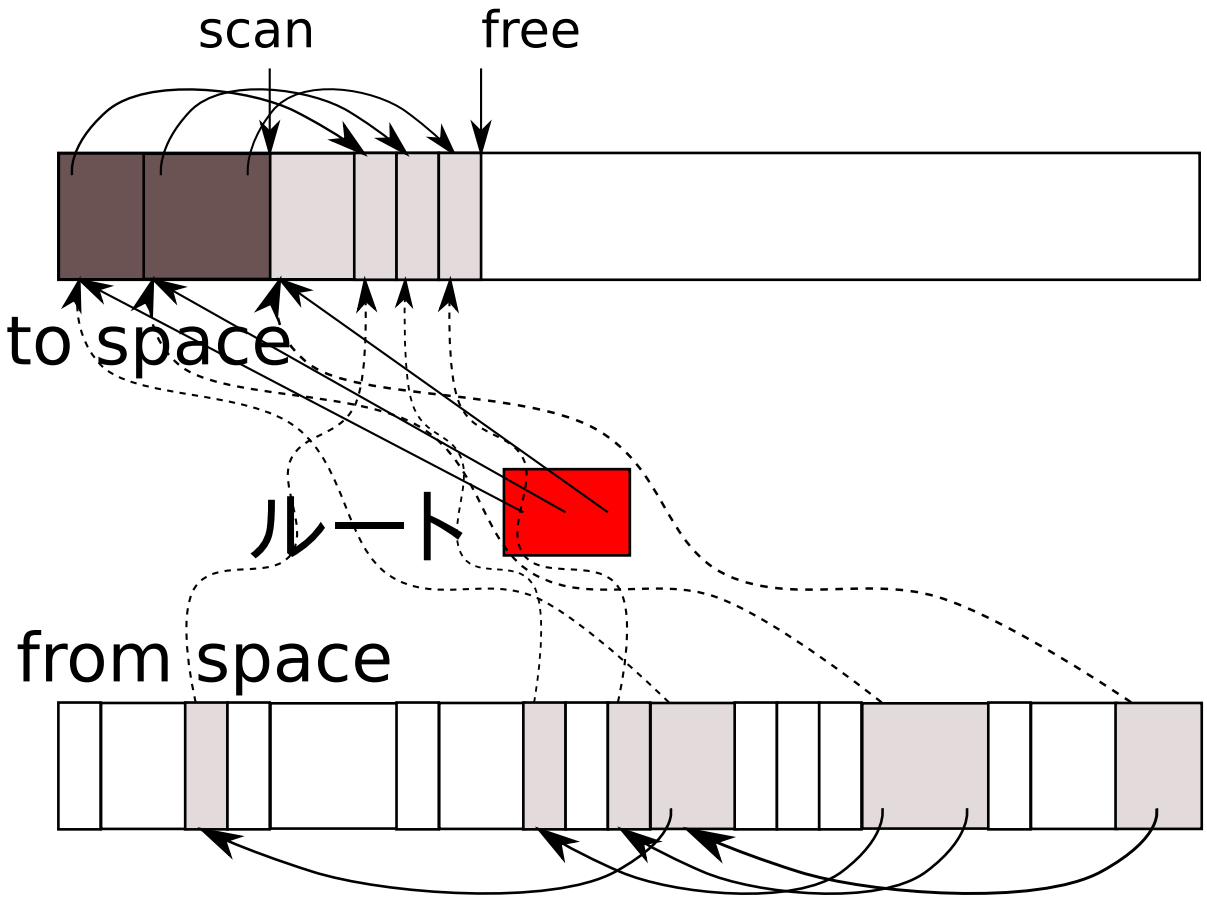
最も基本的なコピーGC図解

- 本質≈グラフのコピー(≈シリアライズ)
  - もともと同じポインタはコピー後も同じになるように
- semi-space GC (ルートから到達可能なオブジェクトをまるごと別領域へコピー)



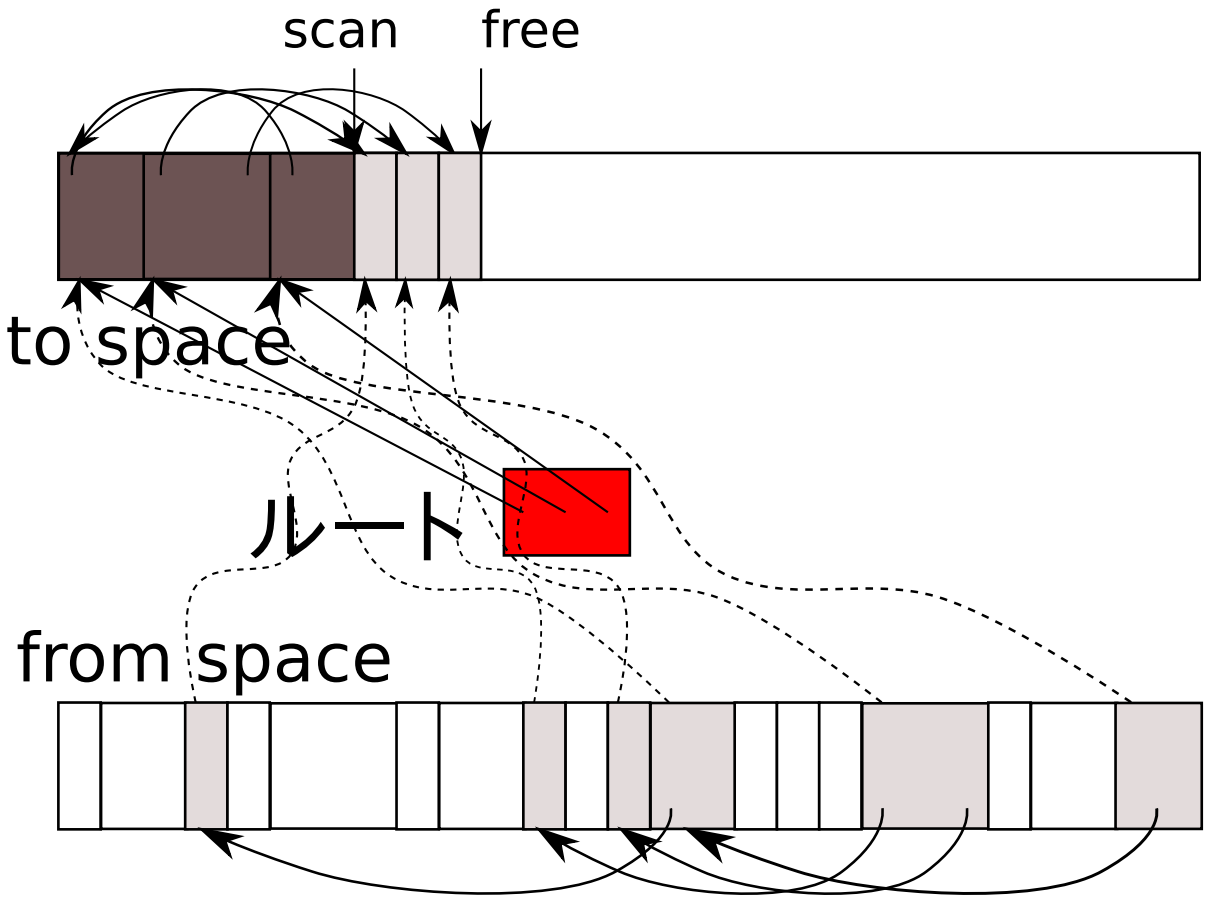
最も基本的なコピーGC図解

- 本質≈グラフのコピー(≈シリアライズ)
  - もともと同じポインタはコピー後も同じになるように
- semi-space GC (ルートから到達可能なオブジェクトをまるごと別領域へコピー)



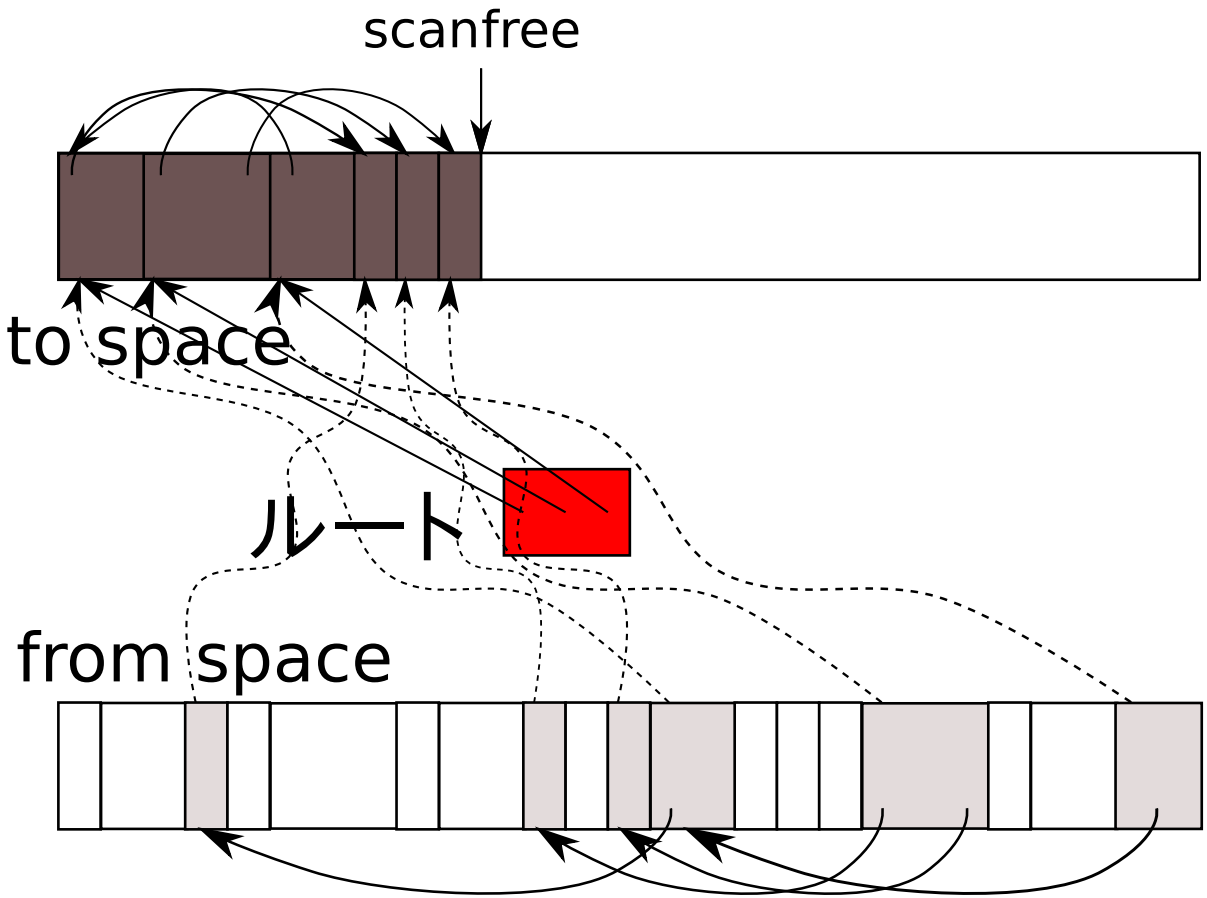
最も基本的なコピーGC図解

- 本質≈グラフのコピー(≈シリアライズ)
  - もともと同じポインタはコピー後も同じになるように
- semi-space GC (ルートから到達可能なオブジェクトをまるごと別領域へコピー)



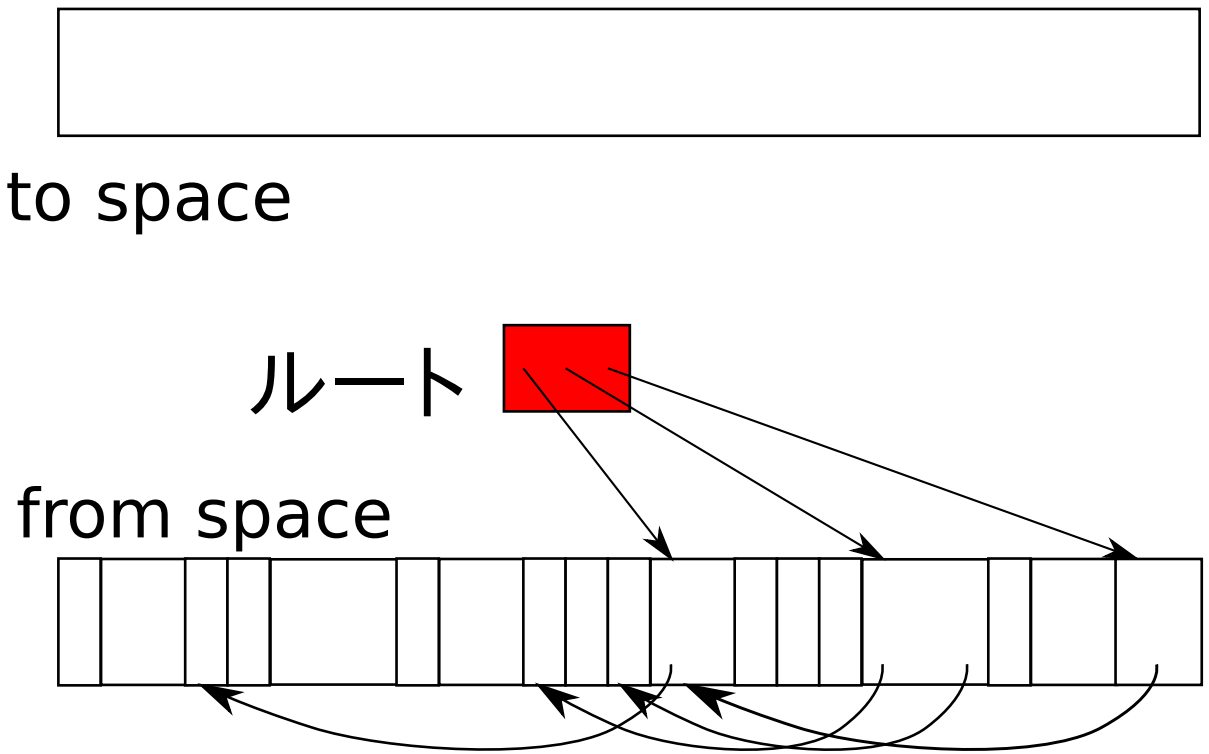
最も基本的なコピーGC図解

- 本質≈グラフのコピー(≈シリアライズ)
  - もともと同じポインタはコピー後も同じになるように
- semi-space GC (ルートから到達可能なオブジェクトをまるごと別領域へコピー)



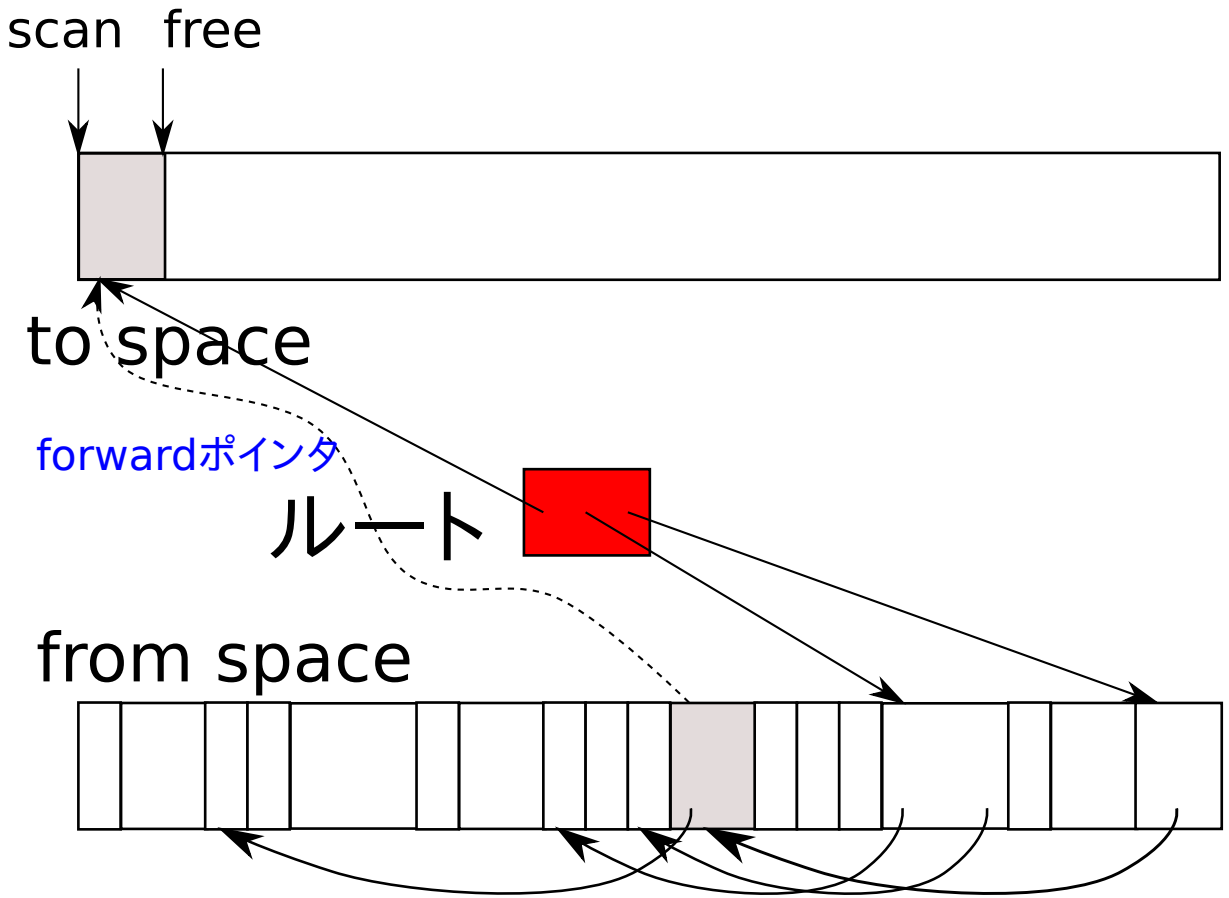
The most basic copying GC: illustration

- in essence, ≈ copying a graph (≈ serialization)
  - the same pointers must remain the same after the copy
- semi-space GC (copy all objects reachable from the root into another space)



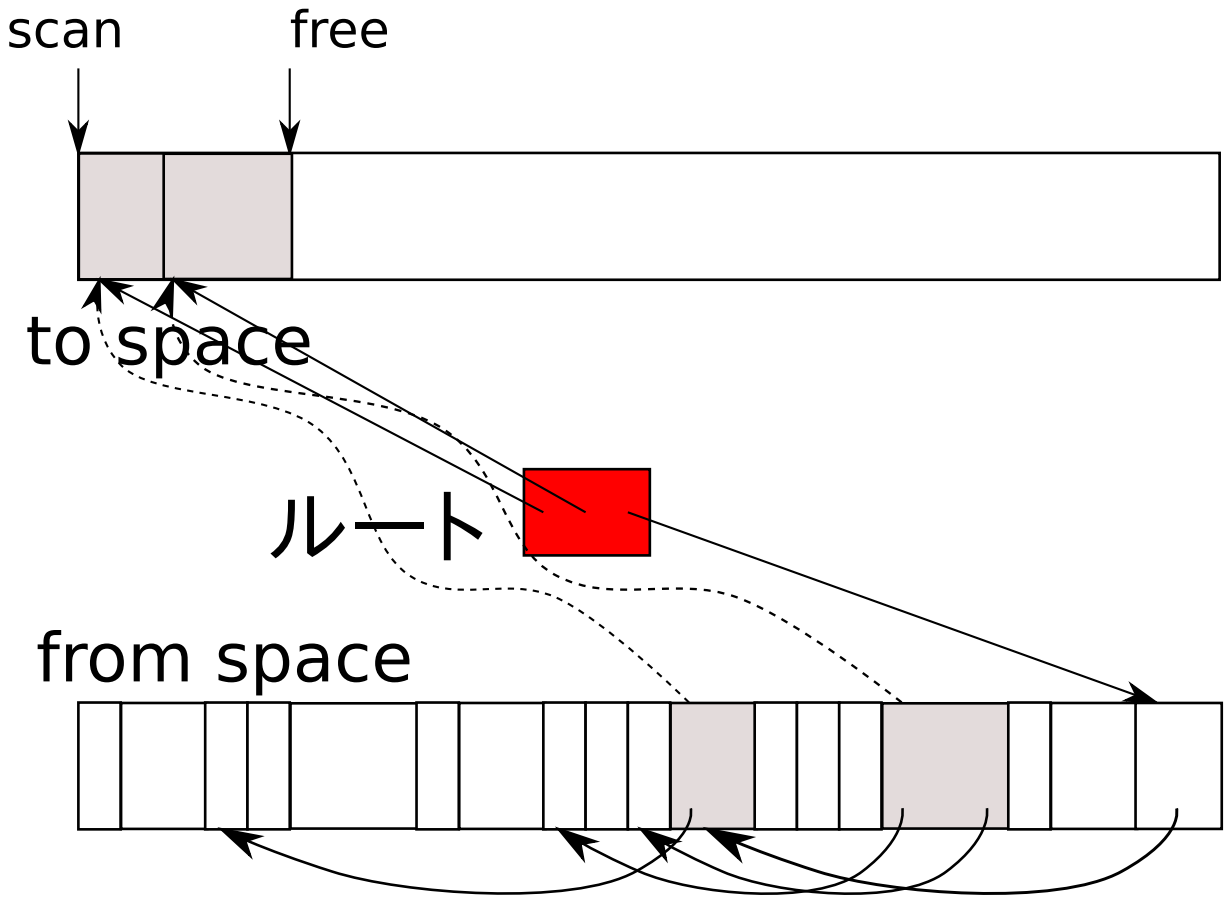
The most basic copying GC: illustration

- in essence, ≈ copying a graph (≈ serialization)
  - the same pointers must remain the same after the copy
- semi-space GC (copy all objects reachable from the root into another space)



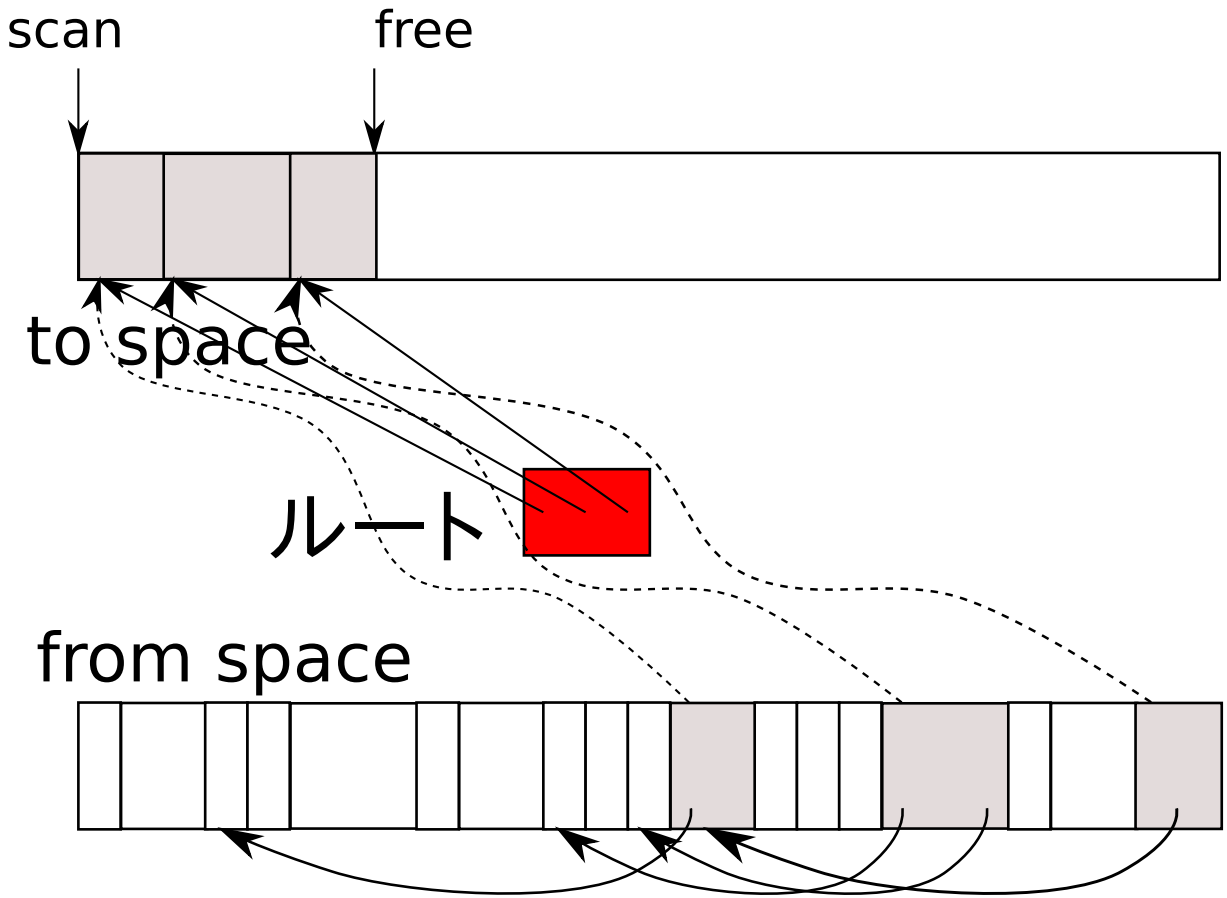
The most basic copying GC: illustration

- in essence, ≈ copying a graph (≈ serialization)
  - the same pointers must remain the same after the copy
- semi-space GC (copy all objects reachable from the root into another space)



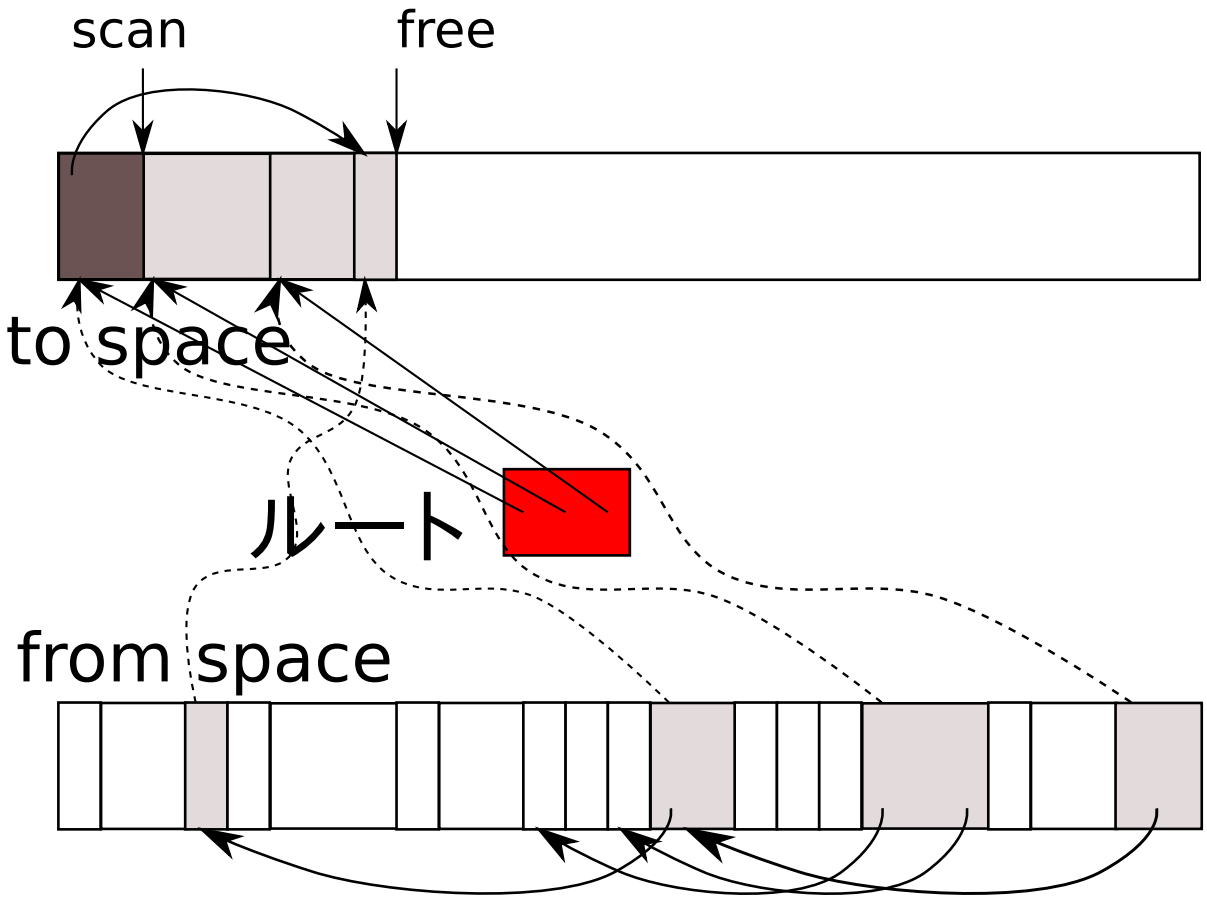
The most basic copying GC: illustration

- in essence, ≈ copying a graph (≈ serialization)
  - the same pointers must remain the same after the copy
- semi-space GC (copy all objects reachable from the root into another space)



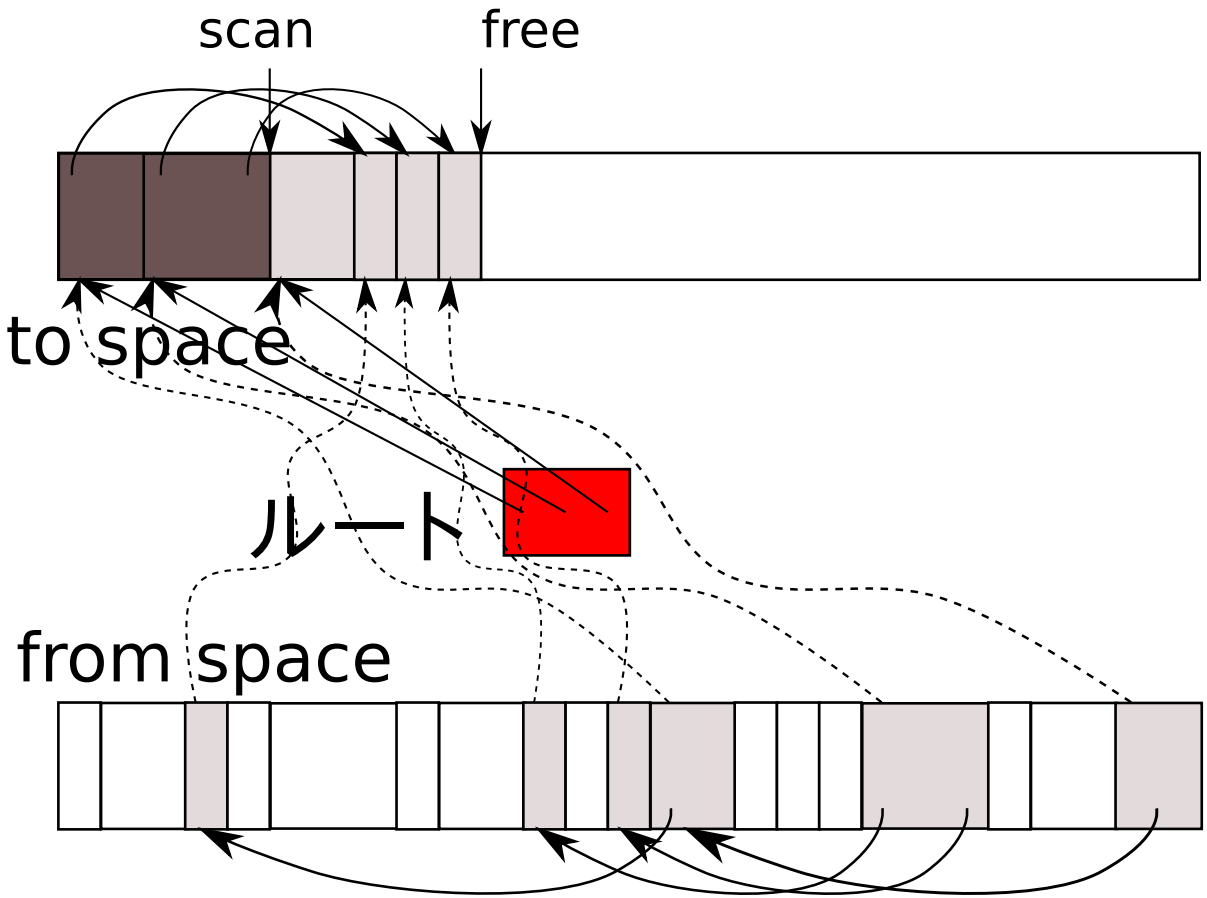
The most basic copying GC: illustration

- in essence, ≈ copying a graph (≈ serialization)
  - the same pointers must remain the same after the copy
- semi-space GC (copy all objects reachable from the root into another space)



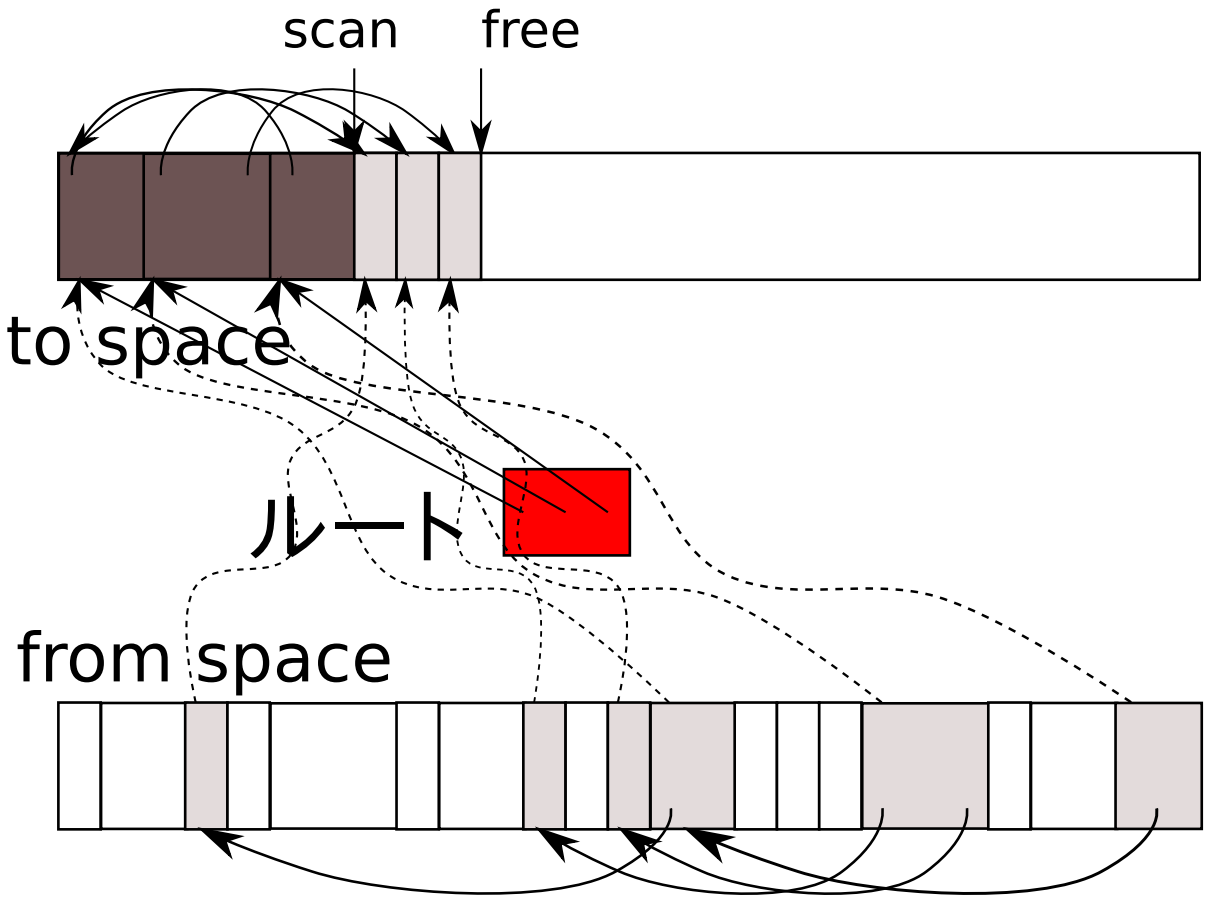
The most basic copying GC: illustration

- in essence, ≈ copying a graph (≈ serialization)
  - the same pointers must remain the same after the copy
- semi-space GC (copy all objects reachable from the root into another space)



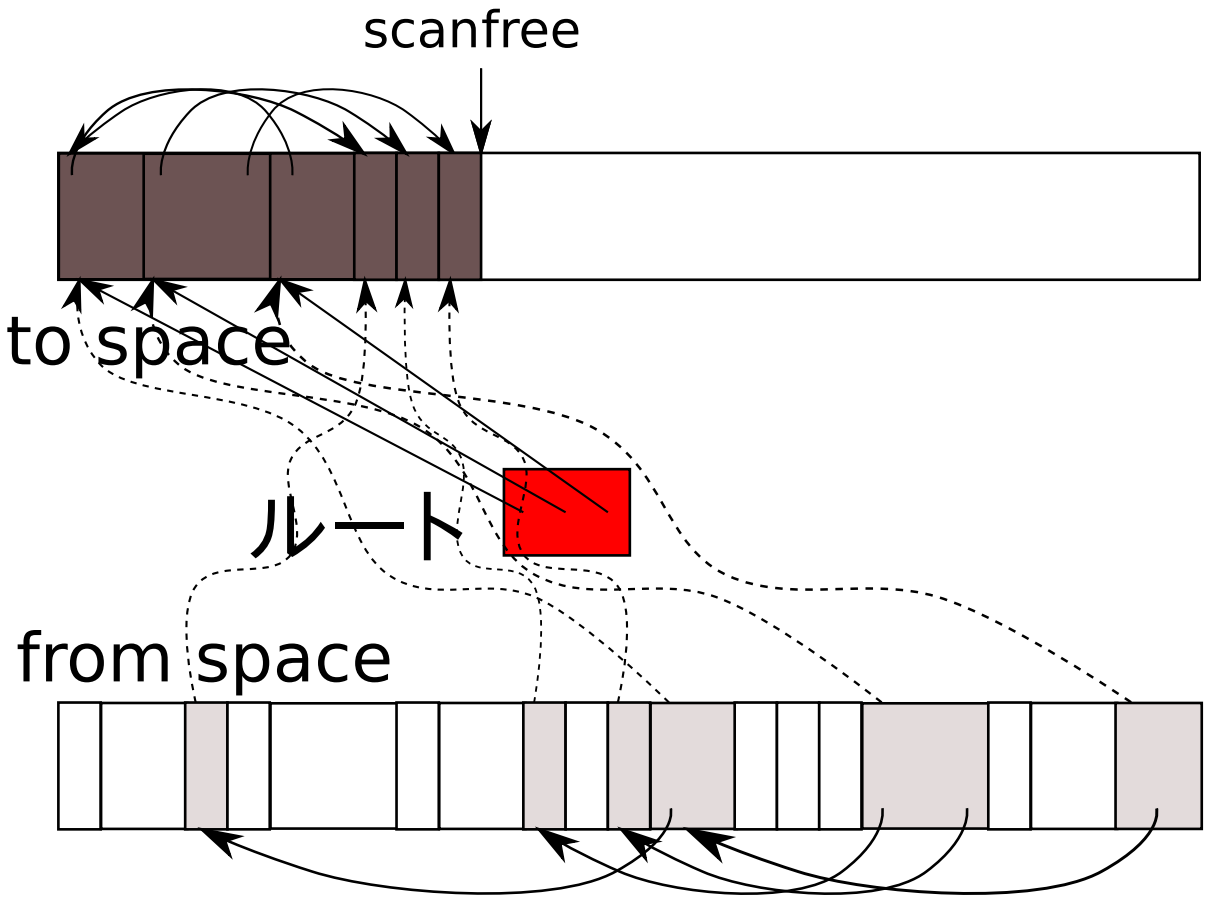
The most basic copying GC: illustration

- in essence, ≈ copying a graph (≈ serialization)
  - the same pointers must remain the same after the copy
- semi-space GC (copy all objects reachable from the root into another space)



The most basic copying GC: illustration

- in essence, ≈ copying a graph (≈ serialization)
  - the same pointers must remain the same after the copy
- semi-space GC (copy all objects reachable from the root into another space)

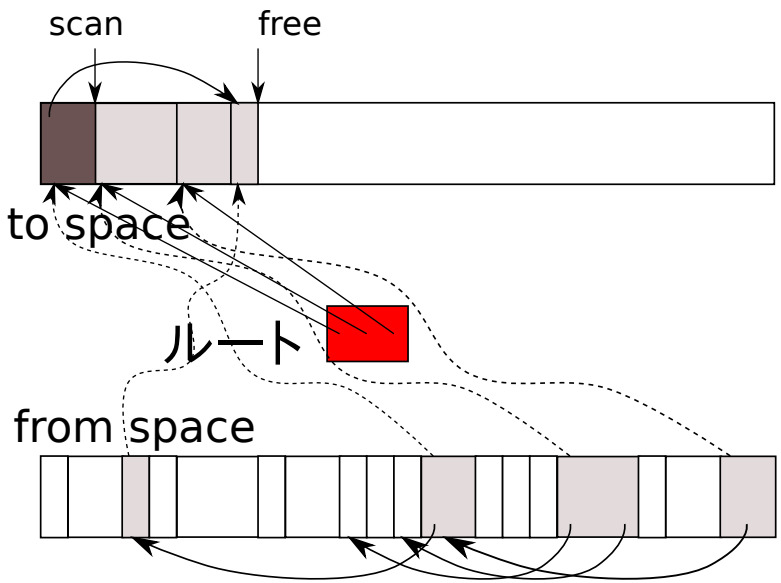


コピーGC: アルゴリズム

```
void *free, *scan;
copy_gc() {
  free = scan = to_space;
  redirect_ptrs(root);
  while (scan < free) {
    redirect_ptrs(scan);
    scan += scan が指すオブジェクトのサイズ;
  }
}
redirect_ptrs(void * o) {
  for (p ∈ o に含まれるポインタ) {
    if (p は copy 済み) {
      p = p の forward pointer;
    } else {
      p を free へコピー;
      p = free;
      p の forward pointer = free;
      free += p が指すオブジェクトのサイズ;
    }
  }
}
```

不変条件

- $p <$ `scan` $\Rightarrow$ p は redirect 済み (from space へのポインタは含まない)
- $p <$ `free` はコピー済み

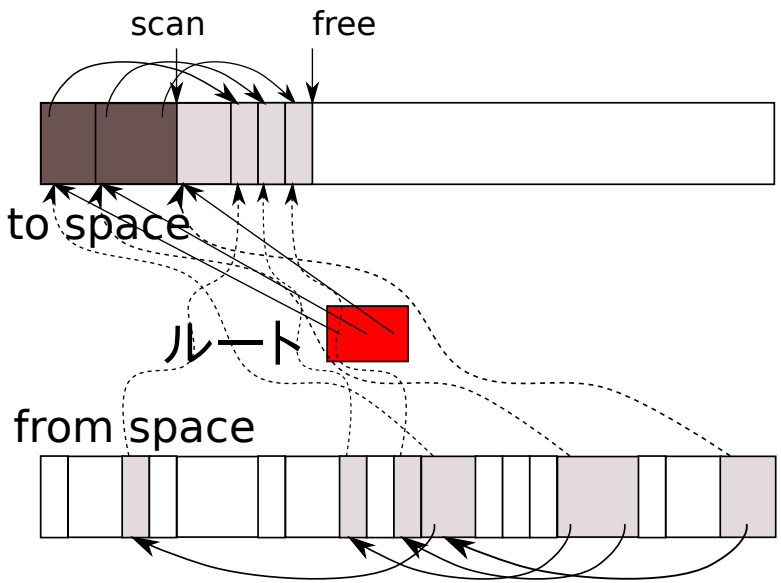


コピーGC: アルゴリズム

```
void *free, *scan;
copy_gc() {
  free = scan = to_space;
  redirect_ptrs(root);
  while (scan < free) {
    redirect_ptrs(scan);
    scan += scan が指すオブジェクトのサイズ;
  }
}
redirect_ptrs(void * o) {
  for (p ∈ o に含まれるポインタ) {
    if (p は copy 済み) {
      p = p の forward pointer;
    } else {
      p を free へコピー;
      p = free;
      p の forward pointer = free;
      free += p が指すオブジェクトのサイズ;
    }
  }
}
```

不変条件

- $p <$ `scan` $\Rightarrow$ p は redirect 済み (from space へのポインタは含まない)
- $p <$ `free` はコピー済み

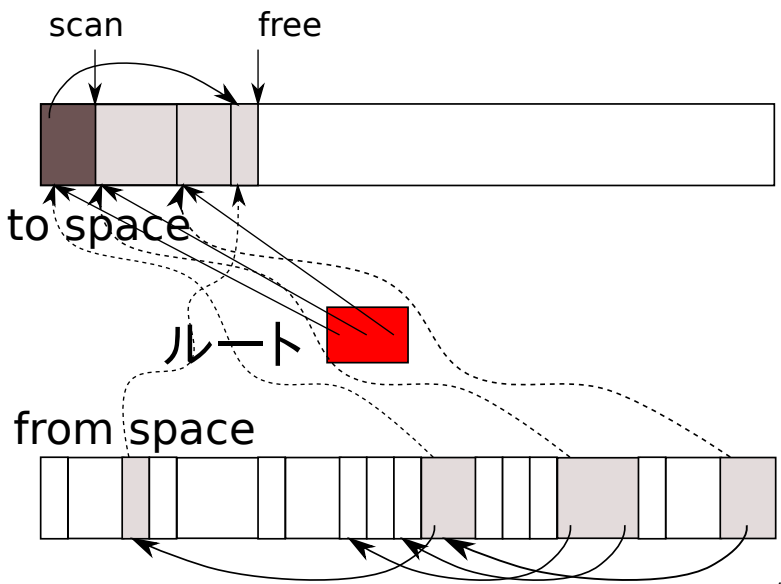


Copying GC: algorithm

```
void *free, *scan;
copy_gc() {
  free = scan = to_space;
  redirect_ptrs(root);
  while (scan < free) {
    redirect_ptrs(scan);
    scan += the size of object scan points to;
  }
}
redirect_ptrs(void * o) {
  for (p ∈ pointers in o) {
    if (p has been copied) {
      p = p's forward pointer;
    } else {
      copy p to free;
      p = free;
      p's forward pointer = free;
      free += the size of object p points to;
    }
  }
}
```

invariant

- $p <$ `scan` $\Rightarrow$ p has been *redirected* (never contains pointers to the from space)
- $p <$ `free` has been *copied* (may not have been redirected)

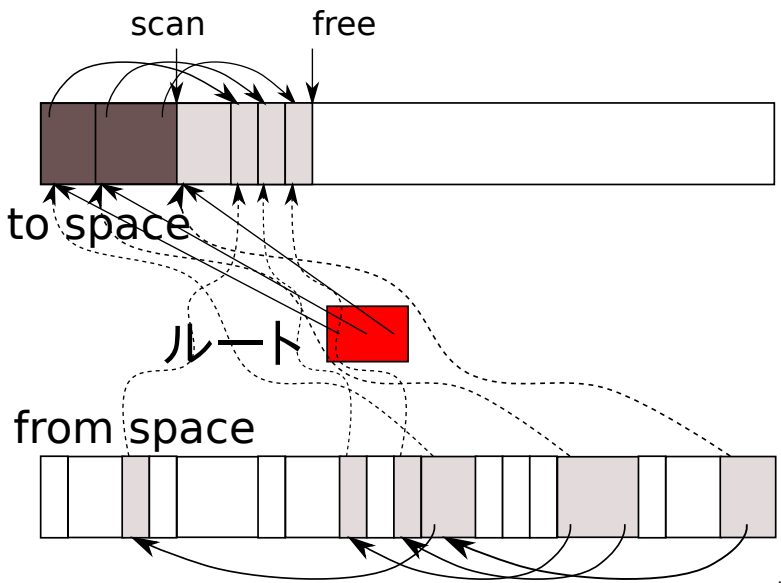


Copying GC: algorithm

```
void *free, *scan;
copy_gc() {
  free = scan = to_space;
  redirect_ptrs(root);
  while (scan < free) {
    redirect_ptrs(scan);
    scan += the size of object scan points to;
  }
}
redirect_ptrs(void * o) {
  for (p ∈ pointers in o) {
    if (p has been copied) {
      p = p's forward pointer;
    } else {
      copy p to free;
      p = free;
      p's forward pointer = free;
      free += the size of object p points to;
    }
  }
}
```

invariant

- $p <$ `scan` $\Rightarrow$ p has been *redirected* (never contains pointers to the from space)
- $p <$ `free` has been *copied* (may not have been redirected)



マーク&スイープGC

- 発見したオブジェクトに「印をつける (マークする)」だけ．コピーはしない
- ポインタの書き換えなども必要ない
- 空き領域は連続していないので，割り当てのサイズに応じて，適切な空き領域を見つけられるような管理方式が必要

Mark&sweep GC

- just “mark” an object when it is found, not copying it
- no need to update pointers
- free blocks in memory are not contiguous, so it must maintain a management data structure to find a free block good for the requested size

マーク&スイープGC vs コピーGC

- コピーGCの利点
  - GC後，生きているオブジェクトの領域が連続領域
  - → 空き領域も連続領域
  - → メモリ割り当てのオーバーヘッド少 (空き領域の探索が事実上不要)
- コピーGCの欠点
  - そもそもコピーは重い
  - コピーのための空き領域が必要 (メモリ利用効率が悪い)
    - 「コピーされ得るオブジェクト量 $\leq$ 空き領域」の保証が必要
    - → from space = to space
  - ポインタと非ポインタが正確に区別できないと動かない (曖昧なポインタが許されない)
    - ポインタだったらコピー先に張り換え
    - ポインタじゃないものを書き換えたら惨事

Mark&sweep vs. copying GC

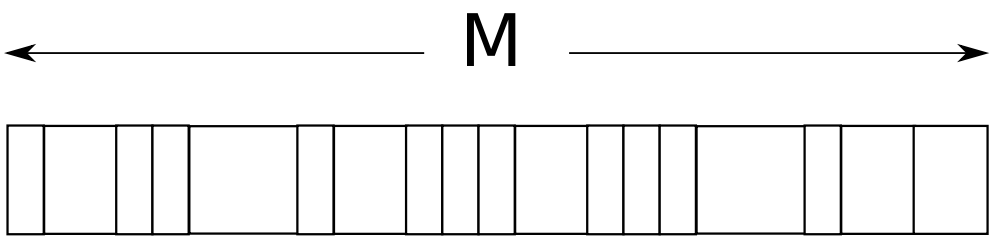
- copying GC pros:
  - live objects occupy a contiguous region after a GC
  - $\rightarrow$ the free region becomes contiguous too
  - $\rightarrow$ the overhead for memory allocation is small (no need to "search" the free region)
- copying GC cons:
  - copy is expensive, obviously
  - the free region must be reserved to accommodate objects copied (low memory utilization)
    - must ensure "size of objects that may be copied" $\leq$ "size of the region to copy them into"
    - $\rightarrow$ "from space" = "to space"
  - pointers must be "precisely" distinguished from non-pointers (ambiguous pointers are not allowed)
    - pointers are updated to the destinations of copies
    - a disaster occurs if you update non-pointers

Contents

1. C/C++のメモリ管理 / Memory Management in C/C++
2. ガベージコレクション (GC) / Garbage Collection (GC)
3. 基本原理と用語 / Basics and Terminologies
   - 2 大方式 (走査型と参照カウント) / Two basic methods (traversing GC and reference counting)
   - GC の良し悪しの基準 / Criteria of evaluating GCs
   - 2 つの走査型 GC (マーク&スイープとコピー) / Two traversing GCs (mark&sweep and copying)
   - 走査型 GC のメモリ割り当てコスト (mark-cons 比) / Memory allocation cost of traversing GCs (mark-cons ratio)
4. C/C++用の GC ライブラリ / A GC library for C/C++

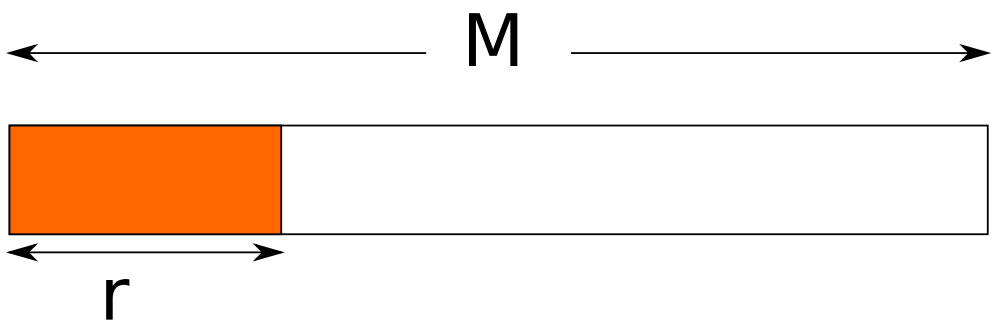
走査型GCのメモリ割り当てコスト

- 大雑把には,
  一回の GC 時間 $\propto$ 到達可能だったオブジェクトの量
- 前提:
  - ヒープの大きさ (copy GC ならば semi space の大きさ) $= M$
  - 到達可能なオブジェクトの量 $= r$
  - 常に r というのは非現実的だがさしあたりそう仮定する
- 挙動: 以下の繰り返し:
  1. GC 発生 $\rightarrow r$ バイト スキャン (またはコピー) して; $(M - r)$ だけの空き領域を作る
  2. $(M - r)$ バイト (GC せずに) 割り当て



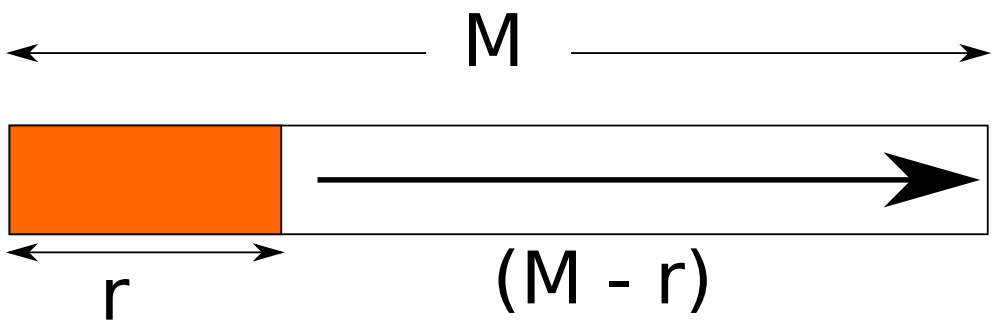
走査型GCのメモリ割り当てコスト

- 大雑把には,
  一回の GC 時間 $\propto$ 到達可能だったオブジェクトの量
- 前提:
  - ヒープの大きさ (copy GC ならば semi space の大きさ) $= M$
  - 到達可能なオブジェクトの量 $= r$
  - 常に r というのは非現実的だがさしあたりそう仮定する
- 挙動: 以下の繰り返し:
  1. GC 発生 $\rightarrow r$ バイト スキャン (またはコピー) して; $(M - r)$ だけの空き領域を作る
  2. $(M - r)$ バイト (GC せずに) 割り当て



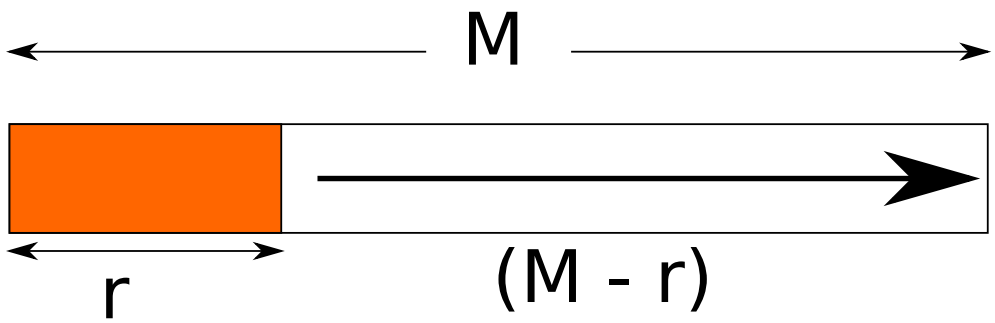
走査型GCのメモリ割り当てコスト

- 大雑把には,
  一回の GC 時間 $\propto$ 到達可能だったオブジェクトの量
- 前提:
  - ヒープの大きさ (copy GC ならば semi space の大きさ) $= M$
  - 到達可能なオブジェクトの量 $= r$
  - 常に r というのは非現実的だがさしあたりそう仮定する
- 挙動: 以下の繰り返し:
  1. GC 発生 $\rightarrow r$ バイト スキャン (またはコピー) して; $(M - r)$ だけの空き領域を作る
  2. $(M - r)$ バイト (GC せずに) 割り当て



走査型GCのメモリ割り当てコスト

- 大雑把には,
  一回の GC 時間 $\propto$ 到達可能だったオブジェクトの量
- 前提:
  - ヒープの大きさ (copy GC ならば semi space の大きさ) $= M$
  - 到達可能なオブジェクトの量 $= r$
  - 常に r というのは非現実的だがさしあたりそう仮定する
- 挙動: 以下の繰り返し:
  1. GC 発生 $\rightarrow r$ バイト スキャン (またはコピー) して; $(M - r)$ だけの空き領域を作る
  2. $(M - r)$ バイト (GC せずに) 割り当て



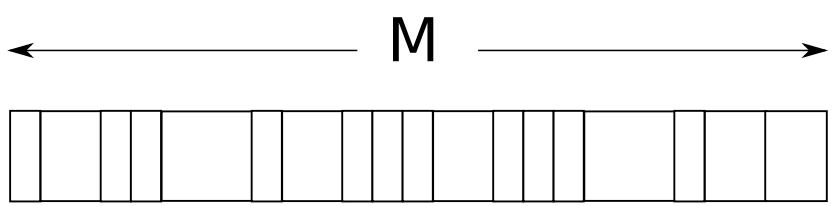
1 バイト割り当てあたりのコスト $= \frac{r}{M-r}$

Memory allocation cost of traversing GCs

- let's quantify the amount of GC's work per allocation
- roughly,

  the time (work) of a single GC $\propto$ *the amount of reached objects*

- assume:
  - heap size (size of a semi-space in case of copying GC) = M
  - reached objects = r
  - assume for the sake of argument it's *always* r
- behavior at equilibrium: repeat the following:
  1. a GC occurs $\rightarrow$ scan (or copy) r bytes, to make a free space of $(M - r)$ bytes
  2. allocate $(M - r)$ bytes (without triggering a GC)

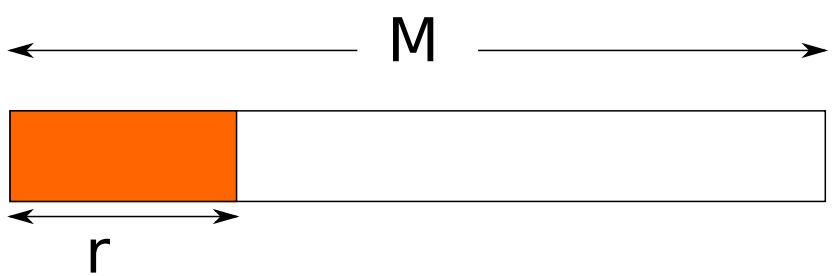


Memory allocation cost of traversing GCs

- let's quantify the amount of GC's work per allocation
- roughly,

  the time (work) of a single GC $\propto$ *the amount of reached objects*

- assume:
  - heap size (size of a semi-space in case of copying GC) = M
  - reached objects = r
  - assume for the sake of argument it's *always* r
- behavior at equilibrium: repeat the following:
  1. a GC occurs $\rightarrow$ scan (or copy) r bytes, to make a free space of $(M - r)$ bytes
  2. allocate $(M - r)$ bytes (without triggering a GC)

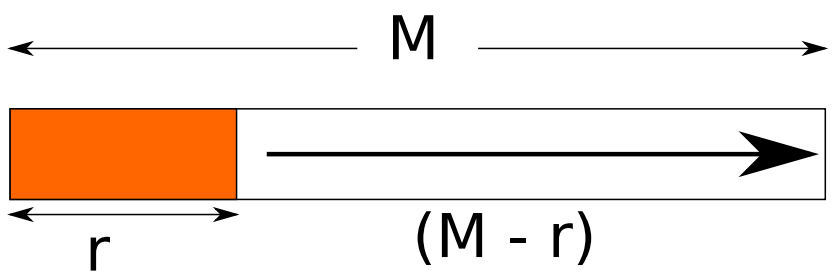


Memory allocation cost of traversing GCs

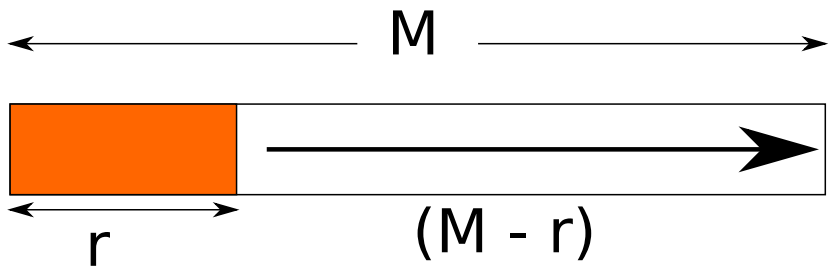
- let's quantify the amount of GC's work per allocation
- roughly,

  the time (work) of a single GC $\propto$ *the amount of reached objects*

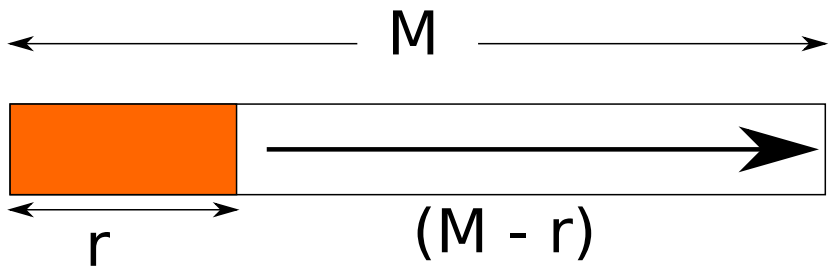
- assume:
  - heap size (size of a semi-space in case of copying GC) = M
  - reached objects = r
  - assume for the sake of argument it's *always* r
- behavior at equilibrium: repeat the following:
  1. a GC occurs $\rightarrow$ scan (or copy) r bytes, to make a free space of $(M - r)$ bytes
  2. allocate $(M - r)$ bytes (without triggering a GC)



Memory allocation cost of traversing GCs



Memory allocation cost of traversing GCs



$\therefore$ the cost of allocating a byte $\propto \frac{r}{M-r}$

走査型GCのメモリ割り当てコスト

- 重要な式:

$$1\text{ バイト割り当てあたりのコスト} \propto \frac{r}{M-r}$$

- 右辺をしばしば mark-cons 比 (mark-cons ratio) と呼ぶ.
  語源:
  - mark : 到達可能なオブジェクトに印をつける仕事量
  - cons : Lisp という言語で，リストセルの割り当て =(`cons x y`)

Memory allocation cost of traversing GCs

- important formula:

$$\text{cost per byte} \propto \frac{r}{M - r}$$

- right-hand side is often called *mark-cons ratio*. its origin:
  - mark : the amount of work to *mark* reachable objects
  - cons : the synonym of memory allocation in the ancient Lisp language =`(cons x y)`

走査型GCのメモリ割り当てコスト

$$1\,\text{バイト割り当てあたりのコスト} \propto \frac{r}{M-r}$$

- r はアプリ固有の量 (GC に左右されない)
  - 注: GC 起動の「タイミング」によって r が上下することはある
- M は調節可能なパラメータ
- M が大 $\rightarrow$ コスト小
- $\rightarrow$ M (使用メモリ) を大きくしてコストを削減可能
- 要は余剰メモリがあればGCをあまりしなくてよくなるというあたりまえのこと
- ただし，単にGCが少なくなる，という以上の定量的な把握は重要

Memory allocation cost of traversing GCs

$$\text{cost per byte} \propto \frac{r}{M - r}$$

- r (primarily) depends only on app (not dependent of GCs)
  - remark: r may fluctuate depending on "when" GCs occur
- M is an adjustable parameter (up to GC's choice)
- M is large $\rightarrow$ the cost is small
- $\rightarrow$ you can reduce the cost by making M (memory usage) larger
- may sound as obvious as "having more memory will reduce the frequency of GCs"
- remember, however, that what is important is to *quantify (and reduce) the cost per allocation (byte)*, not frequency of GCs

M (使用メモリ量) はいくらにするのか?

- M を大にすればコスト小!?
  - $\rightarrow$ 好きなだけ (搭載メモリ量まで) 大きくすれば?
- 普通「節度」を持って M を決める

$$M \propto r$$

例えば，$\alpha > 1$ なる定数を決めて，

$$M = \alpha r$$

- 実際には GC 時に，その時到達可能だったオブジェクトのサイズを計測し，r とする

How large do we make M (memory usage)?

- alright, the larger we make M, the smaller the cost becomes
  - $\rightarrow$ why don't we make it arbitrarily large (up to physical memory)?
- we normally set M "modestly", like:

$$M \propto r$$

e.g., choose a constant $\alpha > 1$ and set:

$$M = \alpha r$$

- a GC measures the amount of reachable objects after that and set r to it (and set M accordingly)

M (使用メモリ量) はいくらにするのか?

- このとき,
  - コスト:

$$\text{mark-cons 比} = \frac{r}{\alpha r - r} = \frac{1}{\alpha - 1}$$

  - 使用メモリ:

$$\propto \text{ある瞬間に到達可能だったオブジェクトのサイズ}$$

  どちらも「理にかなっている」
- ほとんどの GC は α を設定できる
- 通常 $\alpha = 1.5 \sim 2$ 程度だが，大胆に増やせばコストが減ることは知っておくと良い

How large do we make M (memory usage)?

- in this setting,
  - cost:

$$\text{mark-cons ratio} = \frac{r}{\alpha r - r} = \frac{1}{\alpha - 1}$$

  - memory usage

$$\propto \text{the size of reachable objects at a point during execution}$$

  both are "reasonable"
- most GCs allow you to set α (or M directly)
- normally, $\alpha = 1.5 \sim 2$, but it is worth knowing that you can reduce the cost by setting it large

C/C++用のGCライブラリ

- 保守的GC (conservative GC)
- `http://hboehm.info/gc/`
- 通称 Boehm GC (設計者: Hans Boehm, Alan Demers, Mark Weiser)
- C/C++の malloc/calloc/new の代わりに呼ぶだけで GC してくれる
  - malloc, calloc, new → `GC_MALLOC`
  - free → 呼ばない
  - C++用のインタフェース (new の置き換え) もある
- α (メモリ使用量) 調節 → `GC_set_free_space_divisor(`d`)`
- 他の関数: `gc.h` を読んで見よ

A GC library for C/C++

- conservative GC
- `http://hboehm.info/gc/`
- normally called Boehm GC (inventor: Hans Boehm, Alan Demers, Mark Weiser)
- replace malloc/calloc/new of C/C++ and you get GC!
  - `malloc, calloc` → `GC_MALLOC`
  - `free` → don't
  - C++ `new` T → `new (GC)` T (check manual)
- adjust α (memory usage) → `GC_set_free_space_divisor(`d`)` (consult manual for the meaning of d)
- other functions: read `gc.h`

演習の目的

- とにかくありがたく使って効果を見てみる
- 割り当てのコスト，GC の回数などを計測
- 使用メモリを増やせば割り当てのコスト減少を観測

The goal of the exercise

- just try and see it's working (you will appreciate it)
- measure cost per allocation, the occurrences of GCs, etc.
- observe the cost will reduce when you use more memory