ゲーム制御のためのメモリ管理方針

– ゲームを円滑に制御するためのメモリ管理を考える –

2014 年 3 月 13 日　初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 3 月 13 日	板垣 衛	（初稿）

■ 目次

■ 概略

ゲームシステムのメモリ管理について考察し、最適なメモリ管理の方針を固め、メモリ管理システム設計の土台とする。

本書で固めた方針に基づいて、別紙の「柔軟性を追求したメモリ管理システム」を設計する。

■ 目的

本書は、メモリ管理システムの設計にあたり、考慮すべき事項を明確にし、設計方針を固めることを目的とする。

システム要件を確定するための検討事項のまとめである。

■ メモリ管理の課題と方針

考慮すべきメモリ管理の課題とその対応方針を検討する。

▼ メモリ制限の方針：制作者がコストの上限を意識するために

マップ、メインキャラ、敵キャラ、ボス、NPC、エフェクト、イベント、サウンド、メニューなど、それぞれメモリには制限が必要であり、コンテンツ制作者はそれを意識して制作できなければならない。

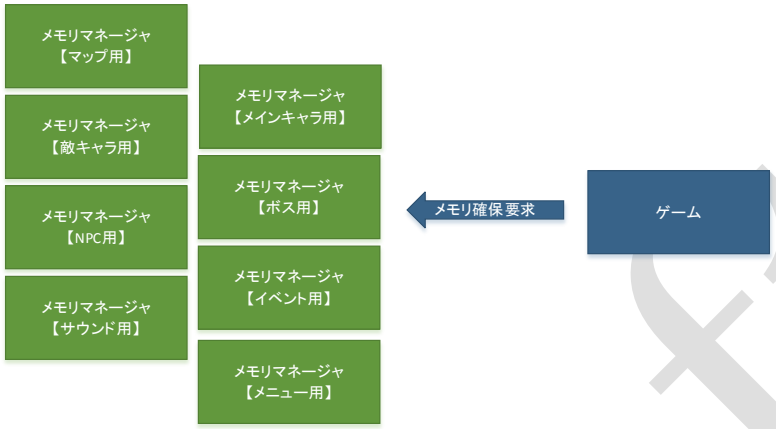
● 【不採用】方法①：メモリマネージャのインスタンスを多数用意

メモリの区分ごとにメモリマネージャのインスタンスを設け、それぞれのサイズのサイズをあらかじめ決めておく。

- 実装方法： メモリ確保の際、メモリ区分もしくはメモリマネージャのインスタンスを指定する。
- 長所： ・ 確実な制限が可能。
  ・ メモリ確保数が分散し、メモリマネージャの処理効率が良い。
  ・ フラグメンテーションの影響を分散できる。

- 短所：
  - 状況に応じて制限を変える事が難しい。
  - 区分が多くなるとそれぞれの空き領域の合計が大きくなり、メモリの使い方として無駄が多い。

イメージ：



メモリ区画ごとに多数のメモリマネージャのインスタンスを用意

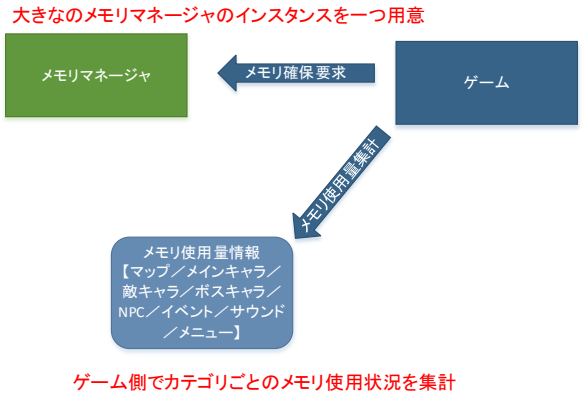
● 【不採用】方法②：メモリ使用状況をゲーム側で集計して画面表示

大きなメモリ空間を共有し、「カテゴリ」ごとにメモリ使用量を集計する。

デバッグ機能により、画面に区分ごとのメモリ使用状況を表示し、制限を超えたときに警告を出すなどする。

- 実装方法：メモリ確保の際、ゲーム側で用意した集計機能付きのアロケータを使用し、カテゴリを指定する。
- 長所：
  - メモリの無駄が少ない。
- 短所：
  - 不確実。
  - 一つのメモリマネージャで多数のメモリ確保を扱う為、処理効率が劣化する。
  - 実装が手間。
  - フラグメンテーションの影響が全体に及ぶ。

イメージ：

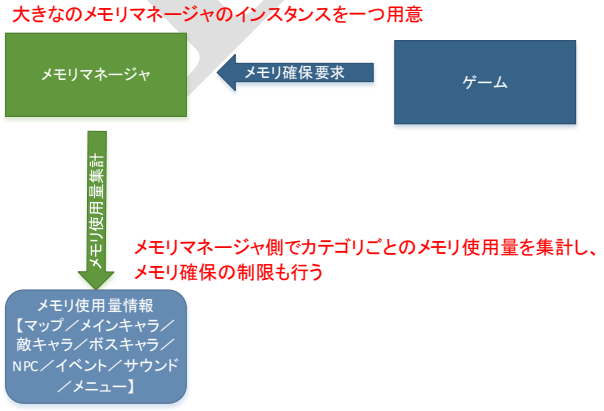


● 【不採用】方法③：メモリマネージャに「メモリ制限機能」を加える

メモリマネージャに対して、「カテゴリを指定してメモリを確保する機能」と、「カテゴリのメモリ制限を設定する機能」を実装し、制限を超えるメモリ確保をできなくする。

- 実装方法： メモリマネージャにカテゴリごとの集計と確保制限の機能を実装し、メモリ確保の際、メモリ区分を指定する。
- 長所：
  - 確実な制限が可能。
  - メモリの無駄が少ない。
- 短所：
  - 一つのメモリマネージャで多数のメモリ確保を扱う為、処理効率が劣化する。
  - フラグメンテーションの影響が全体に及ぶ。
  - 実装がやや手間だが、一度作ればよいので大きな問題ではない。

イメージ：



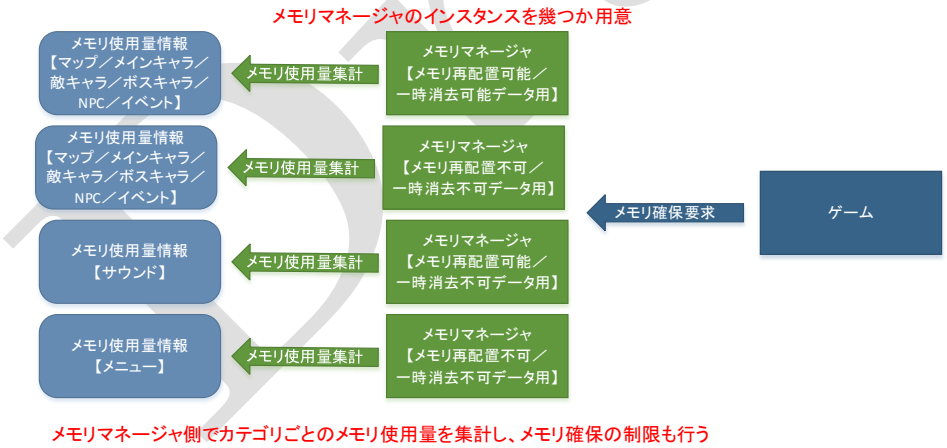
● 【不採用】方法④：「方法①」と「方法③」の組み合わせ

メモリマネージャのインスタンスを大きなまとまりに区分けし、その中で「カテゴリ」ごとの集計と制限を行う。

インスタンスの区分けは、「メモリ再配置可能なメモリ」、「（ムービー再生時などで）大きな連続領域の要求時に全消去可能なメモリ」といったことを基準にする。

- 実装方法：　メモリマネージャにカテゴリごとの集計と確保制限の機能を実装た上で、複数のメモリマネージャのインスタンスを扱う。一つ一つのメモリマネージャは、それぞれ複数のカテゴリに対応する。メモリ確保の際は、メモリ区分が指定され、メモリ区分に応じて適切なメモリマネージャ（のインスタンス）からメモリ確保を行う。
- 長所：
  - 確実な制限が可能。
  - メモリの無駄が少ない。
  - フラグメンテーションの影響を分散できる。
  - メモリ確保数がやや分散し、メモリマネージャの処理効率が少しだけよくなる。
- 短所：
  - 実装がやや手間。

イメージ：



● 【採用】方法⑤：「方法④」の改良版

基本的に「方法④」を踏襲するが、必要なインスタンスの数を減らす（完全に一つにはできない）。

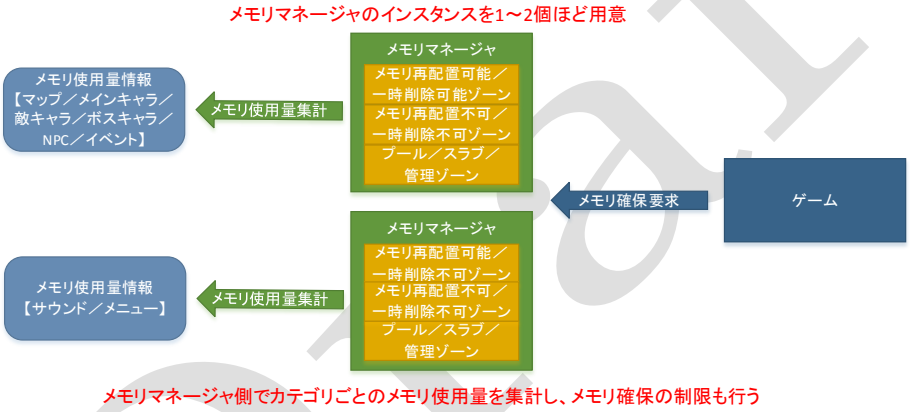
メモリマネージャ内に物理的な区画（ゾーン）を設けることで対応する。

- 実装方法：　メモリマネージャに物理的な区画（ゾーン）を設ける。「カテゴリ」のグ

ループのような位置づけで扱い、メモリ確保時はカテゴリだけの指定で良いようにする。

- 長所：
  - 確実な制限が可能。
  - メモリの無駄が少ない。後述する「プールアロケータ」や「スラブアロケータ」を、複数の区画が共通利用できるので、「方法④」よりも更に効率的になる。
  - フラグメンテーションの影響を分散できる。
- 短所：
  - 実装がやや手間。
  - 一つのメモリマネージャで多数のメモリ確保を扱う為、処理効率が劣化する。ただし、メモリノード（チャンク）の高速検索アルゴリズムを採用することでこの問題を解消する。

イメージ：



▼ ムービー対応の方針：大きな連続領域の確保のために

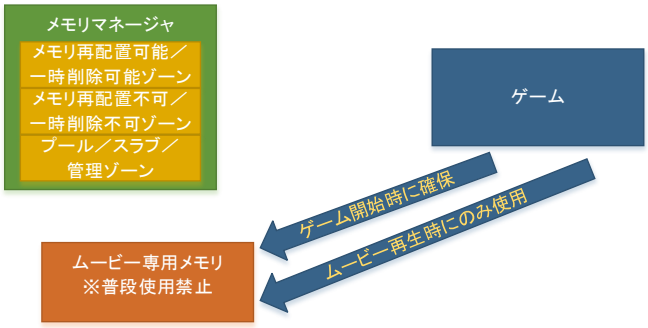
ムービー再生時は大きな連続領域を必要とするため、あらかじめムービー再生に備えた何らかの対処をしておかないとメモリを確保できない。

● 【不採用】方法①：常にムービー用のメモリを確保しておく

ムービー再生用のメモリは常に確保しておく。

- 実装方法：　ゲーム開始時にメモリを確保する。
- 長所：
  - 確実。
  - 実装が簡単。
  - ムービー再生中に、他のリソースに影響を与えない。
- 短所：
  - ムービー時以外のメモリがもったいない。（大きな問題）

イメージ：

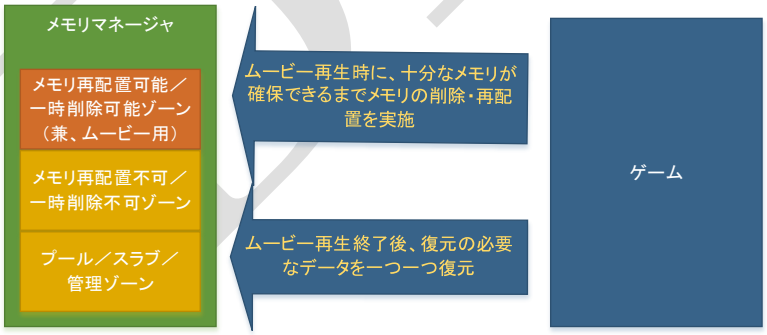


● 【不採用】方法②：ムービー再生時にメモリを空ける

ムービー再生時に、データ削除やコンパクションを利用し、何とかして大きな連続領域を空ける。

- 実装方法： ムービー再生時に、ムービーのためのメモリ確保が成功するまでデータの削除やコンパクションを行う。
- 長所： ・ムービー時以外にもメモリを無駄にしない。
- 短所： ・不確実。（大きな問題）
  - ・実装が手間。
  - ・処理に手間がかかる。
  - ・復元（前の状態に戻す）処理に手間がかかる。

イメージ：



● 【採用】方法③：ムービー時に全クリア可能なメモリを用意

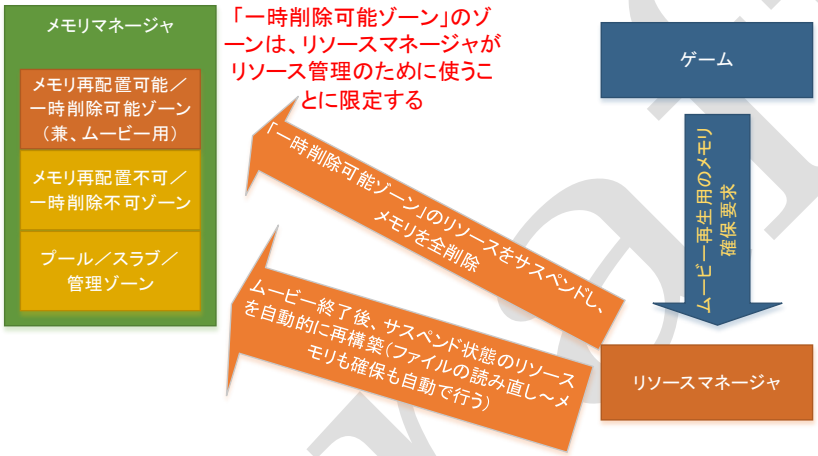
通常は他の用途で使用していながら、ムービー時には全クリアして良い、物理的なメモリ領域（もしくはメモリマネージャのインスタンス）を用意する。

- 実装方法： 「リソースマネージャ」の機能で「リソースの一時削除と自動再構築」が

できることを前提に、「一時削除が可能なグラフィックリソース専用のメモリ領域」を設ける。（別紙の「開発の効率化と安全性のためのリソース管理」でも説明）

- 長所：
  - （削除の手間はかかるが）確実。（大きな問題の解決）
  - ムービー時以外にもメモリを無駄にしない。（大きな問題の解決）
- 短所：
  - 実装が手間。（リソースマネージャとそれを使用したシーンマネージャなど、システムをしっかりと固める必要がある）
  - ムービー前にメモリを空ける処理と、ムービー後にリソースを復元する処理にやや手間（時間）がかかる。

イメージ：



▼ メモリ効率向上のための方針：少しでも無駄のないメモリ確保を行うために

Linux のメモリ管理システムなどを参考に、幾つかのメモリ管理方法に基づいて、ゲームに最適なメモリ管理方法を検討する。

● 【部分採用】検討事項：64bit 対応

メモリのアドレスやサイズを 64bit で扱うことで、4GB を超える大きなメモリ空間を扱うことができる。

しかし、単純な 64bit アドレスの利用は不採用とする。

現状、4GB のメモリ空間で十分なことが多いため、メモリマネージャ内でのアドレス情報（ポインタ）、サイズ情報を 4 バイトに抑え、少しでもメモリの消費を抑える。

4GB 以上のメモリを扱う場合、メモリマネージャのインスタンスを分けて扱う。

アドレスは、メモリマネージャが管理するメモリ領域の先頭からのオフセットで扱い、加算した値を実アドレスとする。メモリマネージャの外部とは実アドレスでやりとりする。

【不採用】検討：32bit で 16GB 拡張

メモリマネージャ内に物理的な区画（ゾーン）を設ける場合、区画ごとに先頭の実アドレスを管理すると、4GB を超える領域を効率的に扱る。

ただし、この案は採用しない。

メモリノード（チャンク）の検索効率を考え、全区画のノードを連結管理することを考えたいため、メモリマネージャ内では単純なアドレッシングとしたい。

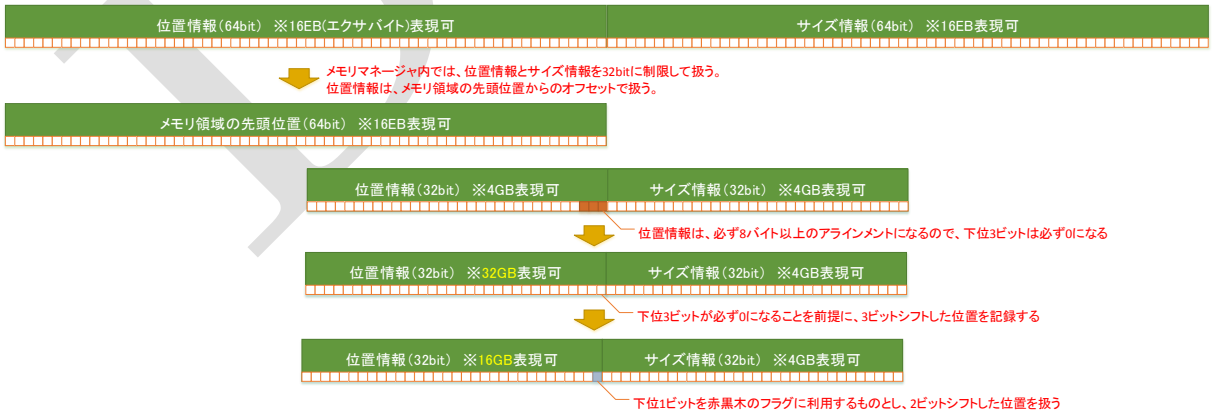
また、区画の境界を動的に変動させるような仕組みを導入したい場合も障害になる。

【採用】検討：32bit で 8GB 拡張

確保するメモリが実質 8 バイトでアラインメントされることを考えると、内部的なアドレス（とサイズ）は 3 ビットの右シフトが可能になり、32GB のアドレス空間を表現できるようになる。

ヒープメモリのノード連結に赤黒木（red-black tree）アルゴリズムを採用することを考慮すると、1 ビットはフラグに利用し、アドレスのシフトを 1 ビットに抑え、16GB のアドレス空間を扱うような仕組みも考えられる。

イメージ：



- **【部分採用】検討事項：仮想アドレス**

Linux や Windows などのマルチプロセス OS は、CPU の仮想アドレス機能を利用し、プロセスのメモリ空間を保護する。

仮想アドレスの仕組みにより、物理メモリと異なるアドレス空間を扱うことが可能となる。これにより、マルチプロセス環境では、それぞれのプロセスが同じアドレスのメモリにアクセスしても、実際の物理メモリは異なるため、競合することがない。

Linux のように、4GB 以上のメモリをサポートする 32bit OS は、この機能を使って、物理メモリを 32bit のアドレス空間に割り付けて扱う。

この仕組みを利用すると、物理メモリを大きく超えるメモリ空間を扱うこともできる。

ページ単位（4KB）で実際にメモリを使用する時にアドレスを割り当て、不要になったら返却する。これにより、ページ単位でのメモリの再アドレッシングが可能となり、断片化が発生しても、アドレスを再割り当てして大きな連続領域として確保し直すといったことも可能となる。

しかし、この仕組みの単純な利用は不採用とする。

仮想アドレスと物理メモリのマッピングを管理するための「メモリリージョン」の管理が必要となり、管理情報が増え、処理が複雑化する。

内部で複数のメモリ区画を扱うことを考えた場合、それぞれの区画に「どれだけ空きがあるのか？」を判定するのも難しくなる。

【可能なら採用】検討：拡張メモリ

全面的な仮想アドレスの採用とせず、「いざという時の拡張用」にだけ仮想アドレスを利用可能とする。

各メモリ区画内の空き領域が再アドレッシングされることはない。

全区画のどれにでも再割り当て可能なメモリとして予約しておき、範囲を超えそうなメモリ区画の領域を自動拡張する。

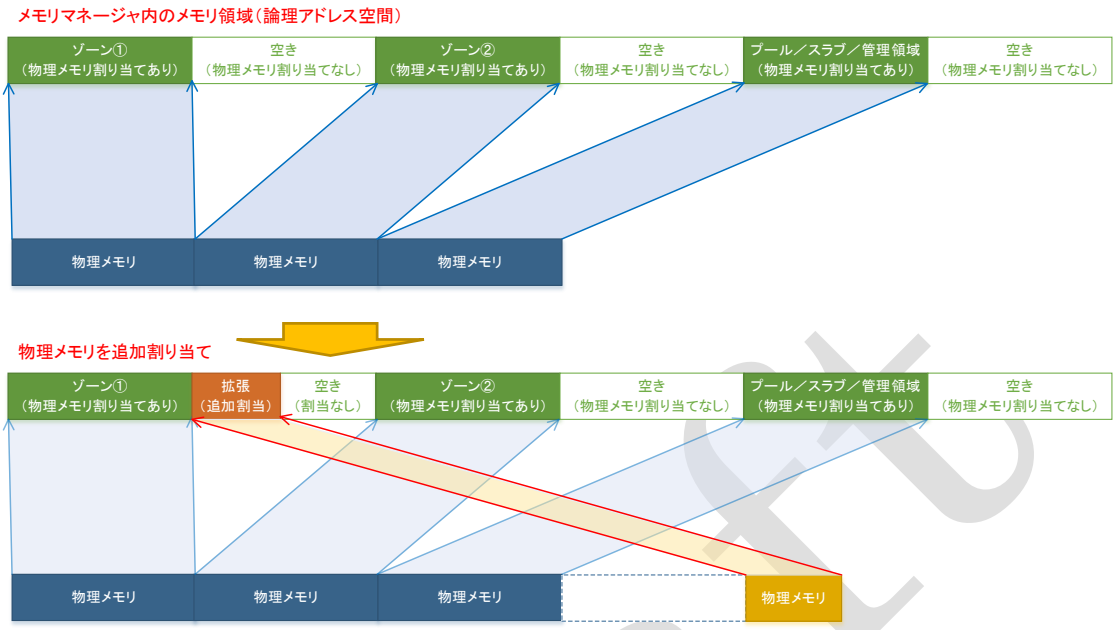
各区画の最初のメモリ割り当て時点で、十分な間隔を空けて仮想アドレスと初期物理メモリの割り当てを行っておき、必要に応じてリニアなアドレス空間にメモリを割り当てる。一度追加割り当てしたメモリは、その区画のメモリが全消去されない限り解放されない。

拡張可能な最大サイズや、一回に拡張するサイズもあらかじめ区画ごとに決めておく。

当然メモリリージョンの管理も必要になるが、わずかな管理となる。

このような形で、少しだけ「メモリの制限を緩くする」ことにより、メモリ不足の問題が発生した時にもハングせず、メモリ使用状況を把握することができるため、制作効率がよくなる。

拡張メモリのイメージ：



【補足】Linux の「メモリリージョン」の仕組みは、プロセスメモリ空間の管理に用いられるものである。そのため、本来メモリ管理の順序としては、以降で検討事項にあげる「ゾーン」や「ページ管理」の後にくるものである。
しかし、本書は「ゲーム」という一つのプロセス内で用いるメモリマネージャを検討するものであるため、「仮想アドレスを意図的に扱うか？」という決定は、最初に検討すべきこととなる。

【参考】仮想メモリの最少単位（ページサイズ）は CPU の機能によって決まるが、使用する CPU アーキテクチャを前提に、各 OS でサイズを規定している。Linux や Windows では 4KB である。

● 【変則的採用】検討事項：ゾーン（ゾーン・アロケータ）

Linux のメモリ管理では、x86(32bit)で三つ、x64(64bit)で二つの「ゾーン」を管理しする。メモリ使用の目的に合わせて物理的なメモリの「ゾーン」が大きく区切られている。

この考え方を真似て、「ゾーン」の仕組みを導入する。

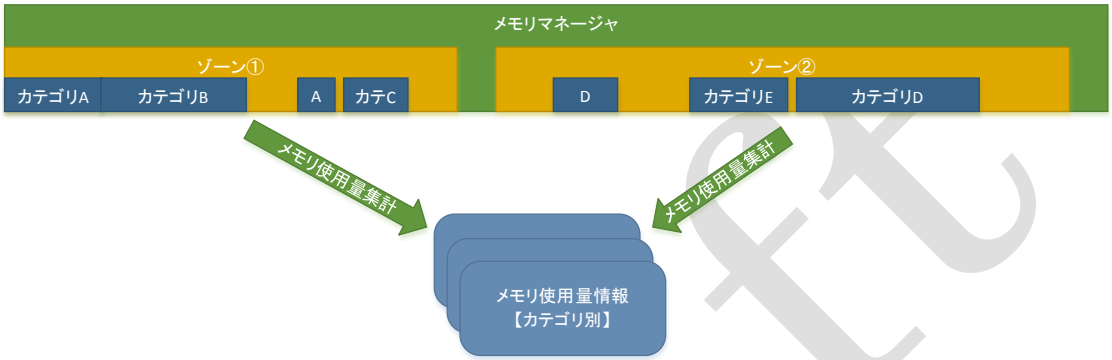
ただし、Linux のようにハードウェア要件に合わせた大きな区分けとしてのゾーンではなく、メモリ用途に応じた区分けとして扱う。

前述の「メモリ制限の方針」にも示したとおり、一つのメモリマネージャのインスタンス内で、メモリの物理的な区画を設ける。その区画を「ゾーン」と呼ぶ。

「カテゴリ」は、「メモリの論理的な制限」として扱う。

ゾーンとカテゴリは階層関係にあり、一つのゾーンには複数のカテゴリがある。カテゴリには物理的な区画がないため、一つのゾーンの中に多数のカテゴリのメモリが入り乱れるが、それぞれの合計サイズを管理し、制限を超えると、物理的な空きがあっても確保に失敗する。

ゾーンのイメージ：



【参考】x86 と x64 でゾーンの数が違うのは、x86 の三つめのゾーンが「High Memory」というゾーンで、32bit より大きな物理メモリを扱うためのものであるため。
そのゾーンでは、大きな物理メモリを 32bit アドレス空間にマッピングして扱うが、x64 ではそもそも 64bit アドレス空間を扱うので、必要が無い。
なお、残りの二つは、「DMA」ゾーン（0~16MB の空間）と「Normal Memory」ゾーン（x86 で 16~896MB の空間）。「DMA」ゾーンは、古い ISA デバイスが 16MB までしかアクセスできないので分けられている。

● 【部分採用】検討事項：ページ管理

Linux のメモリ管理では、x86 系の CPU のメモリ管理に合わせ、4KB のページサイズでメモリを分割して扱う。ゾーン・アロケータがゾーン内のメモリをページ単位で割り当てる。

しかし、単純なページ管理はゲーム用のメモリマネージャを無駄に圧迫するだけなので採用しない。

各「ゾーン」のメモリは、完全にヒープ領域としてのみ扱い、ページの管理は行わない。

【採用】検討：「ゾーン」以外のメモリのページ管理

「ゾーン」以外に、「プールアロケータ」、「スラブアロケータ」、「個別メモリ管理

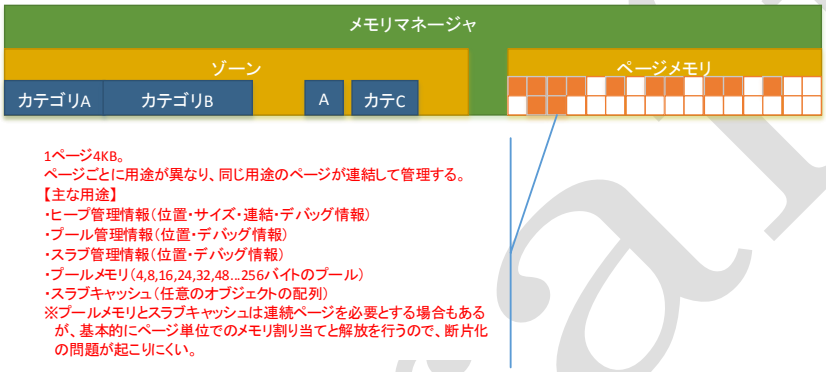
情報」（ゾーンに確保した個々のメモリノードの管理情報など）といったメモリの管理を併用することを考慮し、かつ、これらのメモリ配分をランタイムで流動的に扱うために、ページメモリを管理する。

自由なサイズのメモリ確保が可能なヒープメモリと異なり、ページごとに用途を限定し、それぞれの用途に合わせた固定メモリの配列を扱う。

このメモリ空間は最大でも 1GB ほどの空間を上限に扱うものとし、下記の「バディシステム」で管理する。

また、ゾーンは 1MB アラインメントなどの大きなアラインメントが必要になることも考慮し、このページメモリ区画は、メモリ管理領域全体の最後方に配置して扱う。

ページメモリのイメージ：



【可能なら採用】検討：「ゾーン」のメモリ境界の変動

仮想アドレスを利用したゾーンのメモリ拡張を「可能なら採用」とするが、それができない場合、ゾーンのメモリ境界を変動する仕組みを検討する。

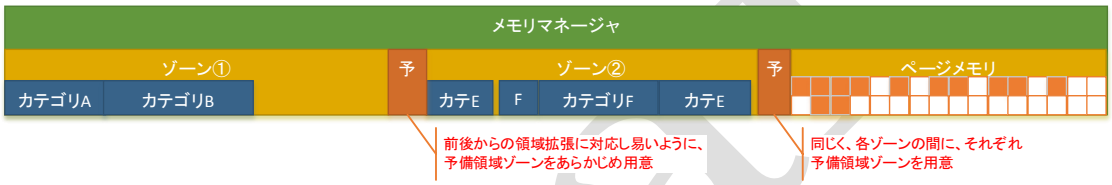
各ゾーンはリニアなメモリ空間に配置し、それぞれアドレスの先頭から順に使用する。ゾーンのメモリが足りなくなった場合に、一つ前に配置されているゾーンの最後方の空きを削り、メモリ境界を動かして区画のサイズを変動する。また、一つ後ろのメモリも前方に空きがあれば同様に境界を動かすことが可能。ページ管理領域も同様の変動が可能。なお、変動可能なメモリサイズにはあらかじめ制限を設け、変動するサイズはページサイズ（4KB）の倍数とする。

これにより、「メモリ不足の警告を出しつつもなるべくゲームを止めない」ということを実現する。

各ゾーンの間にあらかじめ「予備領域ゾーン」を挟むのも可。これはゲーム側でゾーンの設定を工夫することで対応できる。

ゾーンの領域拡張のイメージ：

あらかじめ領域拡張に備えたゾーン配置のイメージ：



【参考】仮に 4GB のメモリ空間を 4KB に区切ったページで管理すると、4GB÷4KB=約 100 万（1M）ページの管理が必要となる。仮に一ページにつき 16 バイトの管理情報を必要とした場合、それだけで 16MB のメモリが必要になる。1GB のメモリ空間なら 4MB。

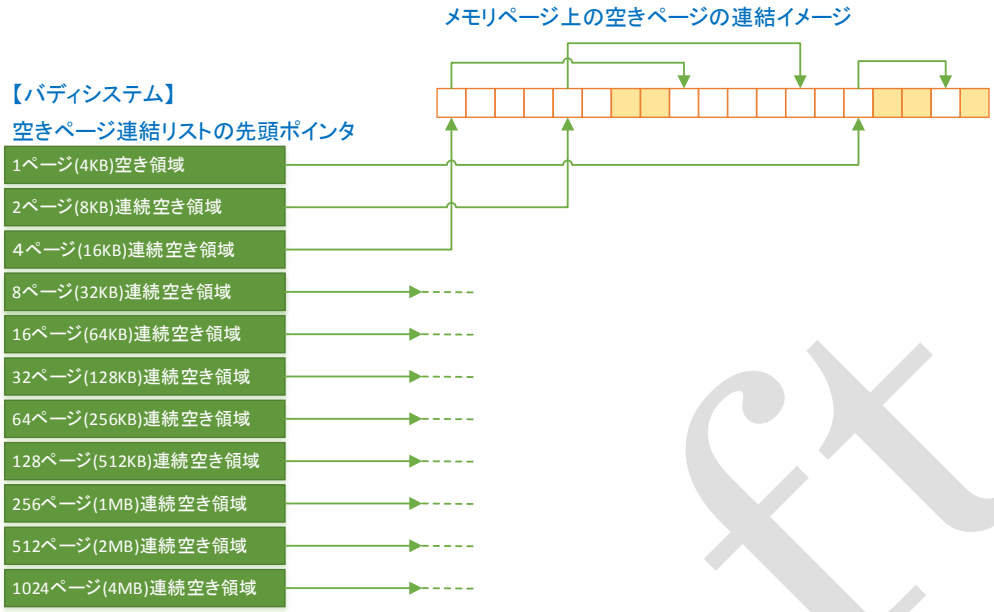
【参考】Linux のメモリ管理システムは、「ページ」と「ページフレーム」を扱う。どちらも 4KB のメモリ空間のことだが、後者は物理的なメモリを指し、ページの論理アドレスに割り当てて（マッピングして）扱う。本システムでは、ページフレームを意識することは特になく、すべて「ページ」と表記する。

- **【採用】検討事項：バディシステム**

Linux のメモリ管理では、「空きページ」の管理方法として、ページの断片化に強い「バディシステム」を採用している。

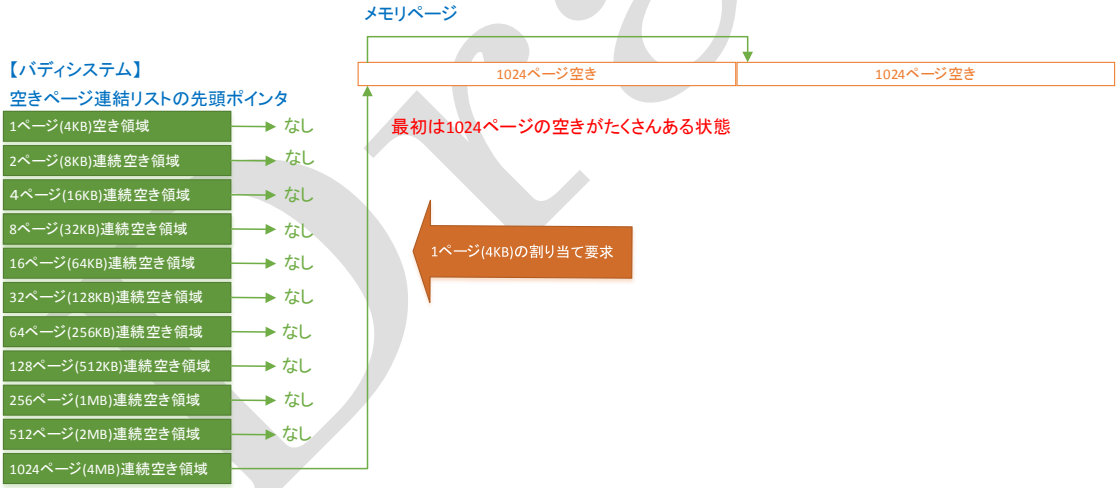
バディシステムは、効率的なページ管理ができるので、ほぼそのまま採用する。

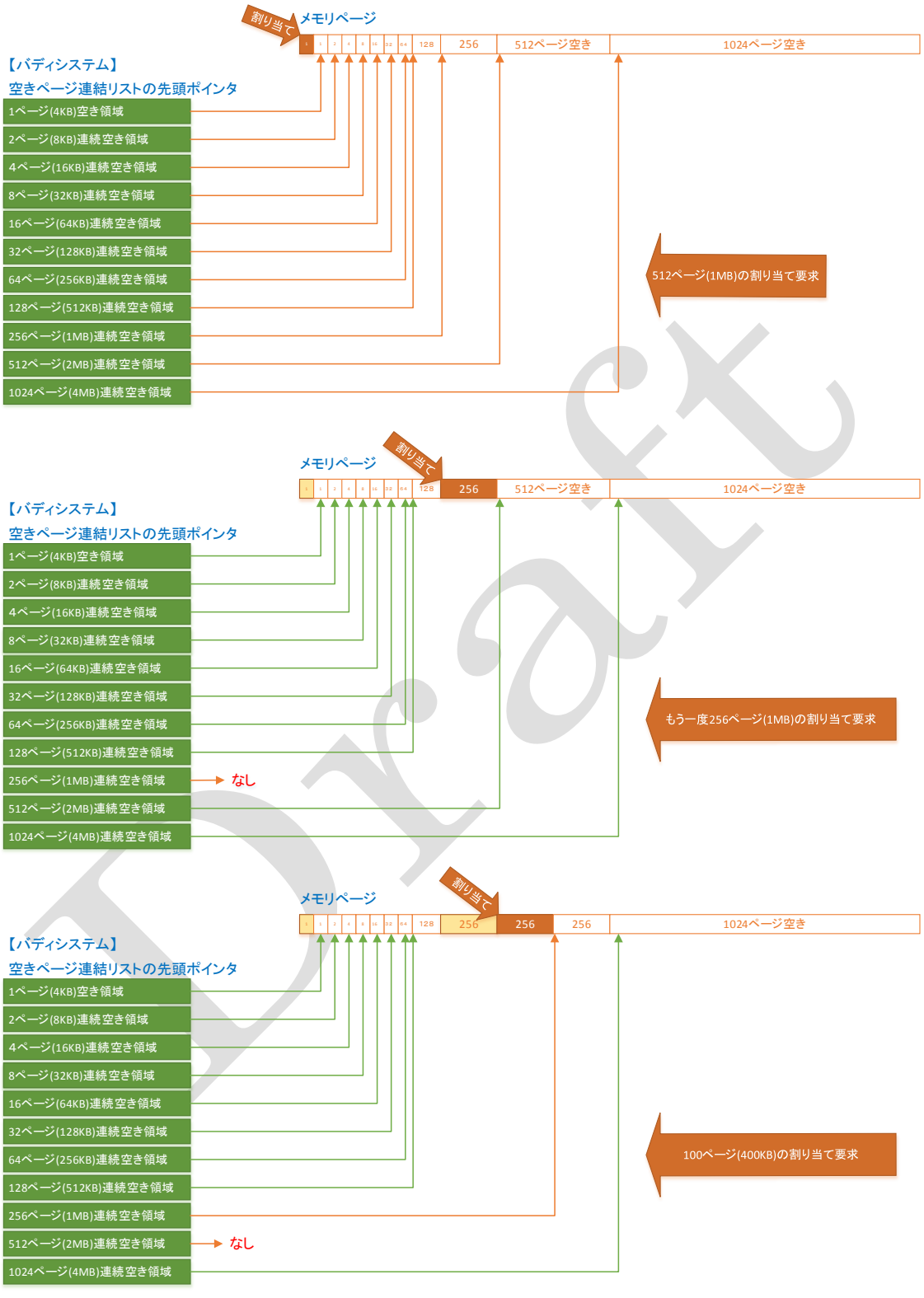
バディシステムのイメージ：

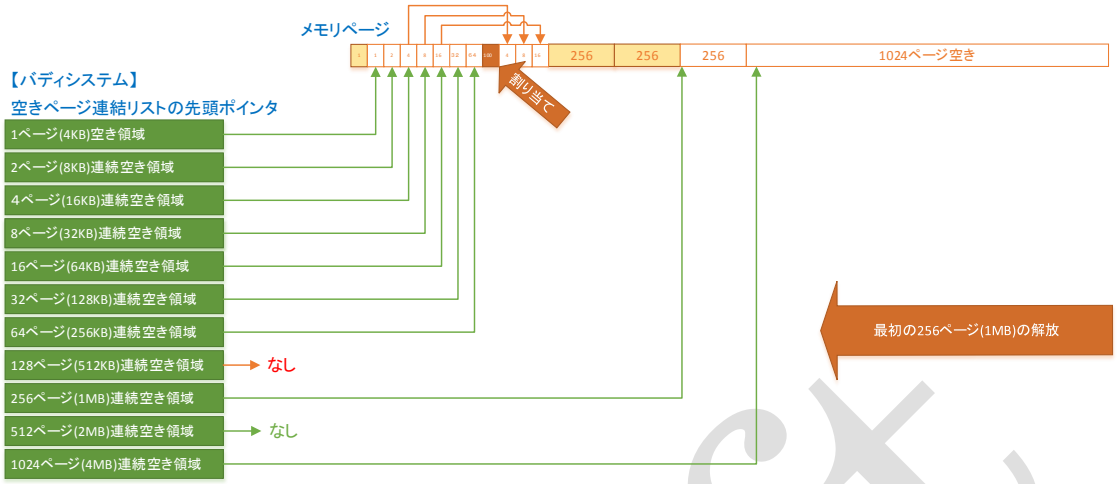


バディシステムは、11個の「連続空き領域（ページ）」連結リストを管理する

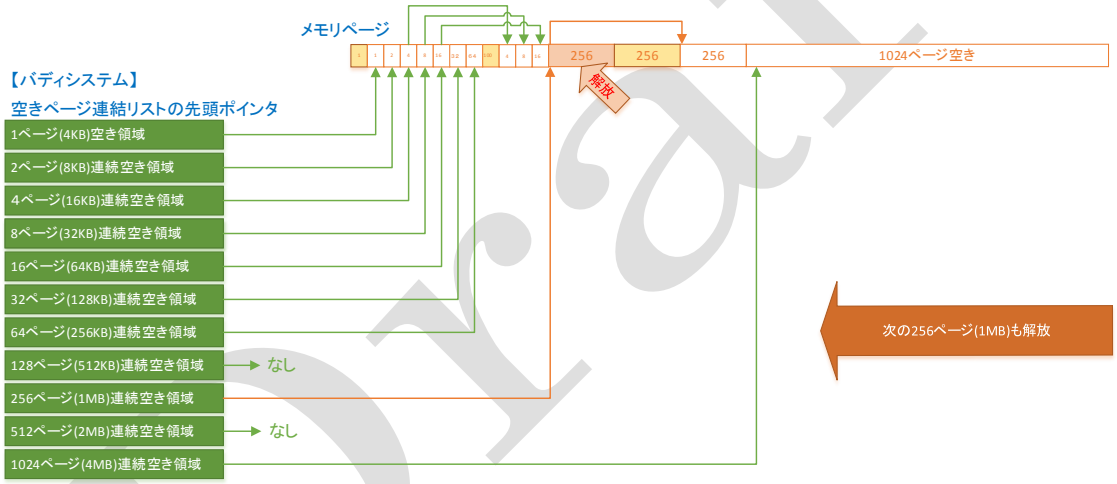
バディシステムのページ割り当てイメージ：

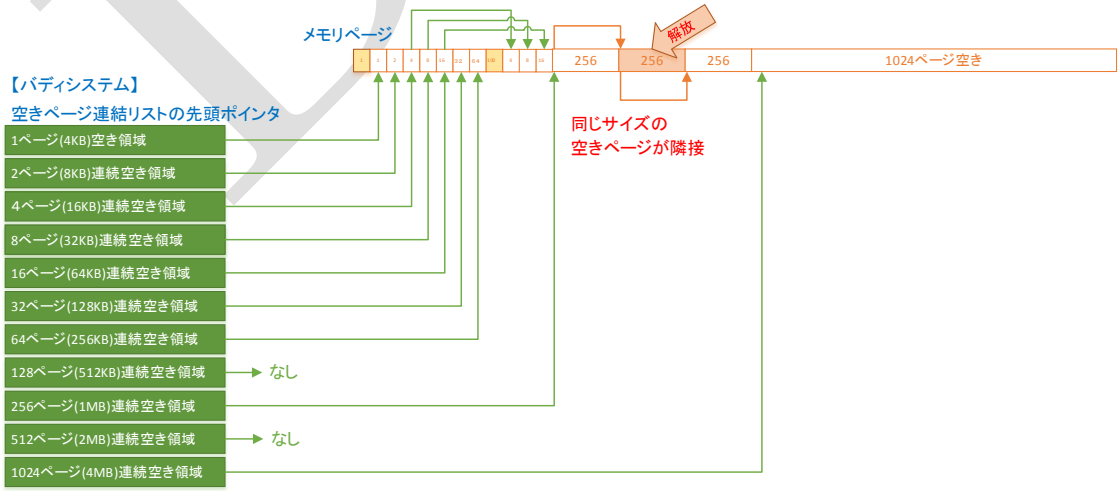


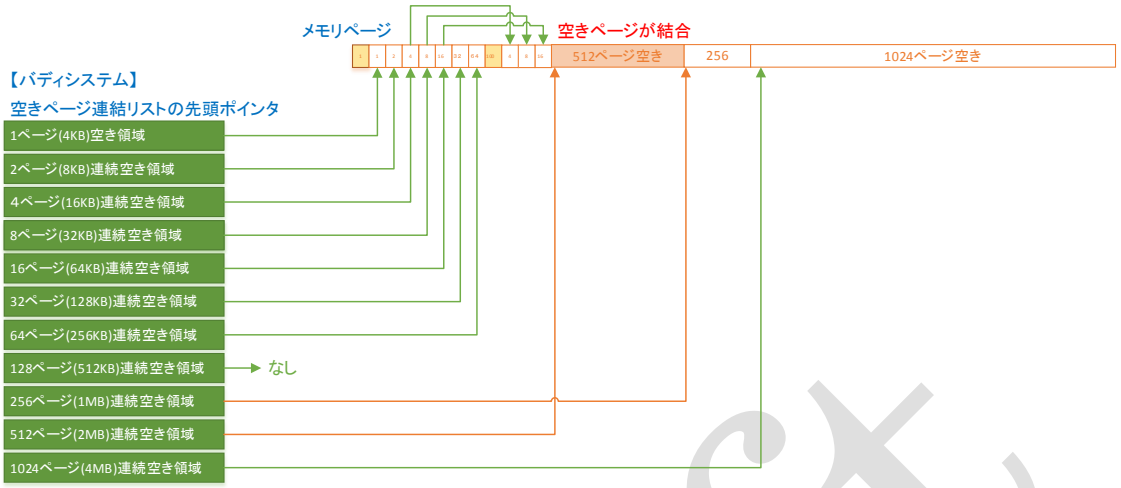

割り当て
メモリページ
128
256
512ページ空き
1024ページ空き
【バディシステム】
空きページ連結リストの先頭ポインタ
1ページ(4KB)空き領域
2ページ(8KB)連続空き領域
4ページ(16KB)連続空き領域
8ページ(32KB)連続空き領域
16ページ(64KB)連続空き領域
32ページ(128KB)連続空き領域
64ページ(256KB)連続空き領域
128ページ(512KB)連続空き領域
256ページ(1MB)連続空き領域
512ページ(2MB)連続空き領域
1024ページ(4MB)連続空き領域
512ページ(1MB)の割り当て要求
割り当て
メモリページ
128
256
512ページ空き
1024ページ空き
【バディシステム】
空きページ連結リストの先頭ポインタ
1ページ(4KB)空き領域
2ページ(8KB)連続空き領域
4ページ(16KB)連続空き領域
8ページ(32KB)連続空き領域
16ページ(64KB)連続空き領域
32ページ(128KB)連続空き領域
64ページ(256KB)連続空き領域
128ページ(512KB)連続空き領域
256ページ(1MB)連続空き領域
なし
512ページ(2MB)連続空き領域
1024ページ(4MB)連続空き領域
もう一度256ページ(1MB)の割り当て要求
割り当て
メモリページ
128
256
256
256
1024ページ空き
【バディシステム】
空きページ連結リストの先頭ポインタ
1ページ(4KB)空き領域
2ページ(8KB)連続空き領域
4ページ(16KB)連続空き領域
8ページ(32KB)連続空き領域
16ページ(64KB)連続空き領域
32ページ(128KB)連続空き領域
64ページ(256KB)連続空き領域
128ページ(512KB)連続空き領域
256ページ(1MB)連続空き領域
なし
512ページ(2MB)連続空き領域
1024ページ(4MB)連続空き領域
100ページ(400KB)の割り当て要求



バディシステムのページ解放イメージ：







【参考】Linux（x64）のバディシステム使用状況表示イメージ：

```
$ cat /proc/buddyinfo          ← /proc/buddyinfo に状況が常に記録されている
Node 0, zone      DMA      4      1      3      4      2      3      2      2      2      2      0
Node 0, zone    DMA32    685   1098    677    101     32     13     13     12     10      4      1
                  ↑x64 なのでゾーンが二つ
                        ↑2^0 ページから 2^10 ページまでの 11 種類の連続空き領域の個数が表示
                              （けっこう断片化していることがわかる）
```

Linux では、頻繁にページの割り当て（物理メモリのマッピング）と解放が繰り返されることがあるため、CPU のキャッシュ効率を上げる仕組みを実装しているが、そこまでは対応しない。

ページ管理の意図

本システムが「ページ管理」を採用するのは、用途ごとのメモリの配分を実行時に調整できるようにするためであり、頻繁なページの開放は意図しない。

例えば、後述するプールアロケータの利用に際して、「8 バイトプール」の割り当て個数と「16 バイトプール」の割り当て個数のバランスを、事前決定しなくていいようにする、といったことが目的である。

ページ単位で「8 バイトプールのページ」、「16 バイトプールのページ」のように扱い、要求が多いものは都度ページを追加して割り当てていく。

同様に、ヒープメモリの個々のメモリ管理（ノード）情報や、プールメモリを含む個々のメモリ割り当てのデバッグ情報なども、1 ページ単位で都度割りあてを行う。

バディシステムの有効活用

バディシステムは 2 ページ以上の連続空き領域を管理するためのものであるが、本システムにおいても有効活用できる。

本システムは、別途ヒープメモリを用意することもあり、2 ページ以上に渡る連続領域の確保が要求される機会がそこまで多くはないが、全くないわけではない。大きなサイズのメモリプールや、初期のメモリ配分、スラブアロケータの利用に際しては、やはりまとまったメモリが必要になる。

- **【変則的採用】検討事項：プールアロケータ**

Python のメモリ管理では、256 バイト以下のメモリを無駄なく高速に管理するために、「低レベルメモリアロケータ」を実装している。

256 バイト以下の小さなメモリが、とりわけアロケートされる機会が多いことに基づく実装である。

4KB 単位の「プール」に区分けされたメモリを、メモリ確保の要求に応じて「8 バイト用（×512 ブロック）」、「16 バイト用（×256 ブロック）」、「24 バイト用（×170 ブロック）」...「256 バイト用（×16 ブロック）」のように割り当てて扱う。小さいメモリは多数の確保が予測されるため、最初のメモリ確保時点で 4KB 分まとめて予約する仕組みである。

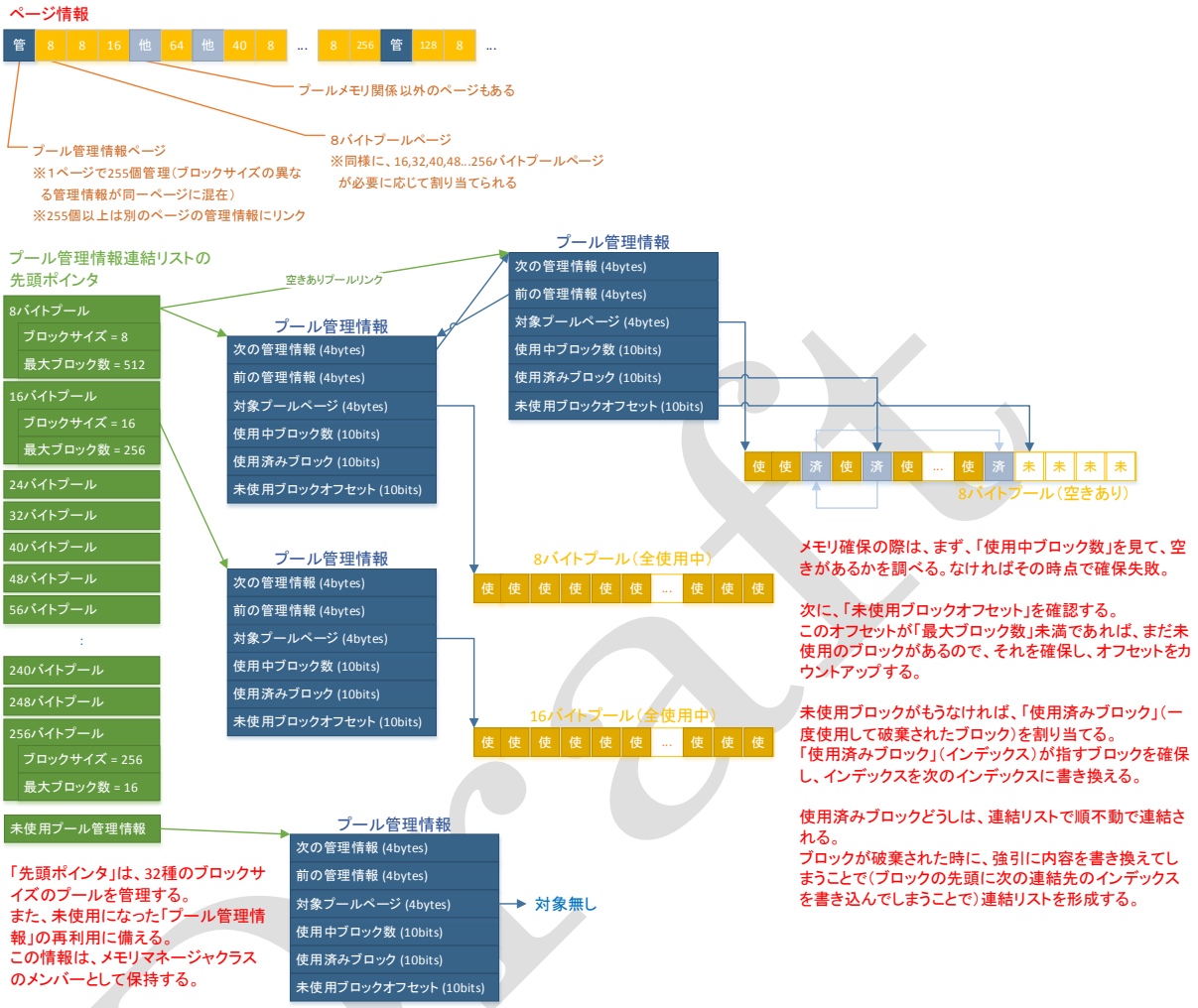
例えば、8 バイト（以下）のメモリ確保要求の際に、512 個のメモリブロックが一気に用意され、以後同様に 8 バイト（以下）のメモリ確保要求があったらそのプールが使用される。プールを使い尽くした後にまた要求があったら、新たにプールを割り当てる。

「低レベルメモリアロケータ」は、ゲームプログラミングにおいても有効であるため、ほぼそのまま採用する。

その呼称は、分かり易く「プールアロケータ」とする。

メモリ確保時に指定された「ゾーン」と無関係に、小さなメモリは全てこのプールアロケータから確保する。なお、「カテゴリ」別のメモリ使用量の集計と確保制限には対応する。

プールアロケータのイメージ：（本システム向けの構造として大きく改変した状態）



プールのページ管理

4KB の「プール」は、そのまま 4KB の「ページ」に対応するため、必要に応じてページをプールに割り当てる方法を取る。

アリーナの代替

Python では、「アリーナ」と呼ばれる 256KB の領域をまとめて確保し、それを 4KB ずつ区切ったものを「プール」としている。256KB を超えるプールが必要になったらまたアリーナを確保する仕組みである。

本システムにおいては、アリーナはとくに扱わず、要求に応じてページを割り当てていくものとする。

ページ領域の管理

ページ領域はプールアロケータ専用の領域ではなく、他の用途にも用いているため、制限を設ける。例えば「全ページ数の 80%」といった条件とする。それ以上はヒープからの割り当てを行う。

また、アリーナのようにまとめてプール用のメモリを確保せず、常に必要に応じて 1 ページずつプールに割り当てる。プール内のメモリがまるごと解放されたらページを解放し、他の用途に回せるようにする。

プール管理情報専用ページ

Python ではプールの状態をアリーナが管理しているが、本システムではアリーナを使用しないので、プールの状態を管理するための専用情報を設け、そのための専用ページを割り当てて扱う。

プール管理情報に 1 ページをまるごと割り当て、管理情報が 1 ページを超えるようならまた新たなページを管理情報に割り当てて連結リストでつなぐ。

なお、管理情報ページ内の管理情報がまるごと未使用状態になった場合、255 個の連結組み替えによって破棄することができるが、重くなるので実際に処理すべきかどうかは要検討。

プール管理情報の構造

プールの管理情報は、管理情報どうしを連結する双方向連結リストで構成し、対象プールのページ（アドレス）と使用中ブロック数、使用済みブロックのインデックス、未使用ブロックオフセットを扱う。1 情報あたり 16 バイトに収めると、4KB のページ内には 256 – 1 プールの管理情報を持つことができる。（ページの先頭にはページ自体の管理情報があるため -1 する）

仮に 1 ページまるごと 8 バイト用プールの管理情報だった場合、512 ブロック × 255 プール = 130,560 ブロックを管理できる。

ブロックの管理

メモリ確保要求に応じてブロックを割り当てた場合、後述するヒープメモリのメモリノード管理と同じく、個々の管理情報を持ち、一様にスマートポインタやカテゴリによるメモリ制限、デバッグ情報などを扱う。

● 【採用】検討事項：ヒープメモリ

Linux のメモリ管理では、プロセス固有のメモリ空間としてメモリリージョンを使用する。メモリリージョンは、前述の通り、物理メモリをプロセスに割り当てた上、プロセス固有の仮想アドレスに割り付けする仕組みである。これにより、プロセス間でのメモリの干渉を防ぎ、必要に応じてプロセス用のメモリを増減させることを可能にする。

メモリリージョンによって確保されたメモリ領域を、「ヒープメモリ」として、プロセス（プログラム）は自由に使用することができる。「malloc／free 関数」によって操作するメモリのことである。

前述のとおり、本システムでは、ヒープメモリ用に、あらかじめ「ゾーン」を割り当てて扱う。また、ある程度ゾーンの増減にも対応する。

【補足】

若干本題から逸れるが、重要なことなので記述する。

前述もしているが、本システムにおける「ゾーンの増減」の仕組みは、「フレキシブルなメモリ活用」を目的としたものではなく、「制作者に向けてメモリ限界を警告しつつ、ゲームを止めないこと」を目的としたものである。

とにかくゲーム制作は「メモリに対する意識」が重要であり、それをきちんと制作者に伝えるシステムが肝要である。それも「ハング」のような原因追求が必要な形ではなく、動きを止めずに的確な問題の通知ができるのが効率的である。

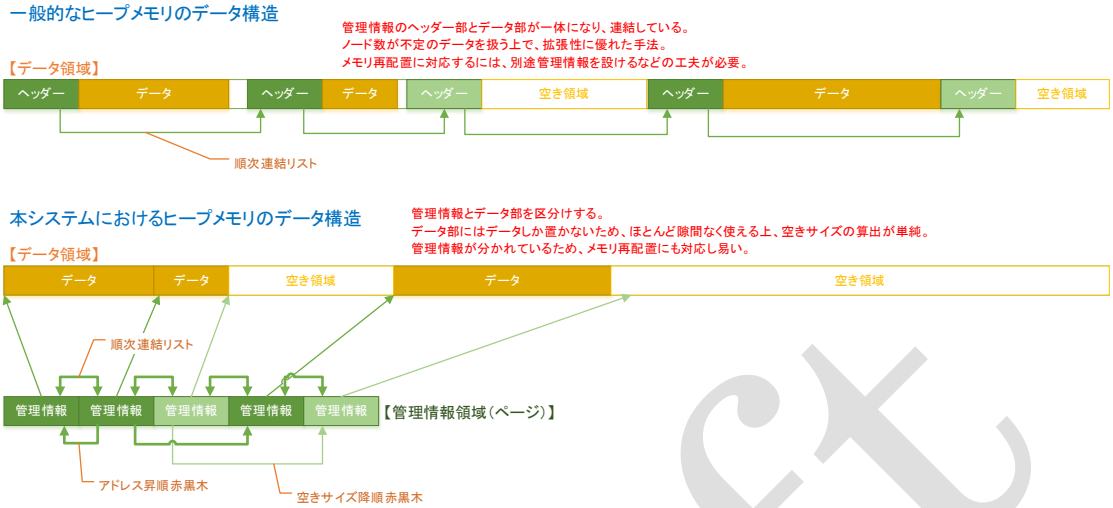
本システムでは、256 バイト未満のメモリ確保には「プールアロケータ」を用いるが、それより大きいメモリの確保にはヒープメモリを用いる。

ヒープメモリの管理方法は、よくある「ヘッダー部」＋「データ部」の構造を取らず、ヒープの管理情報を専用ページに割り当てる構造を取る。これにより、ゾーンの領域には純粋なデータしか配置されない。

この構造を取る目的は下記の通り。

- メモリ再配置（コンパクション）に対応
- 参照カウンタ、デバッグ情報などの管理情報を、プールアロケータと共通化
- デバッグ情報の有無によるメモリ状態への影響を極小化
- アラインメントの無駄を削減
  - 大きなアラインメントのデータを確保する際、連続確保でも一つ一つ隙間を作る
- 空き領域管理の単純化
  - 空きノード（チャンク）を結合する際にヘッダーサイズ以下の隙間を意識する必要があることや、空きノード再利用時にアラインメントに合わせたヘッダーの位置調整などが面倒

ヒープメモリのイメージ：



管理情報の分離は、多目的なページ管理の導入を前提に、じゅうぶん実現可能な構造である。

専用ページを持つことによる冗長性も生じるが、最大でも 4KB の無駄に収まる上、それが全てのゾーンの総計としての無駄であるため、メモリ効率は十分である。

具体的な管理情報の内容は、以降の検討課題によって確定する。

なお、管理情報の分離により、下記のような新たな課題も発生するが、許容可能なトレードオフとする。

- ページ管理が手間 ..... ページ割り当ては十分高速であり、そこまで頻繁ではない。
- 管理情報の追加が手間 ..... 利用可能な空き管理情報は、常にリンクリストとページ内の空き情報オフセットで管理し、すぐに利用できる（プールアロケータの空きブロック管理手法とほぼ同様）。ページ追加時に手間が発生。
- メモリが空いていても管理ページが空いていないとメモリ確保できない
  ..... 最大の課題。この問題が発生しないほどの十分なページ領域を用意する必要がある。最初に大きすぎるほどの領域を用意しておき、観測に基づいて次第に減らしていくのが良い。ページ管理は断片化の影響がほとんどないので、サイズ縮小時の懸念は大きくない。
- 実データのアドレスに対応した管理情報の検索が手間
  ..... 管理情報の連結に赤黒木アルゴリズムを採用し、高速化。（後述するが、この仕組みにより、「ゾーンを

意識しない高速検索」、「Best-Fit アロケーションの最適化」、「プールメモリにも管理情報を持たせる」といった利点も生む）

- **【採用】検討事項：Best-Fit アロケーション①：ヒープメモリ**

メモリアロケータが要求されたメモリを割り当てる際には、三つの戦略がある。

- First Fit ..... 空きノード（チャンク）を順番にチェックしていき、要求されたメモリサイズ以上の空きノードが見つかった時点で割り当てる。高速性とメモリ効率の折衷案的手法。
- Best Fit ..... 全ての空きノードの中で、要求されたメモリサイズに最も近い空きノードを割り当てる。もっともメモリ効率が良いが、検索に時間がかかる。
- Worst Fit ..... 常に最大の空きノードを割り当てる。もっとも高速だが断片化し易い。メモリ確保後の余りや、メモリ解放時のサイズなどから、最大サイズはすぐに計算でできる。

いずれもメモリ割り当て後、残った領域は（ある程度以上のサイズが残ったなら）空きノードとして分割管理する。

単純な連結リストによるヒープメモリの場合、「First Fit」が選ばれることが多い。

本システムにおいては「Best Fit」を採択する。

（連結情報が増えることになるが）メモリサイズをキーにした降順の赤黒木を用いることで、十分な検索速度を維持できる。重複したキーを扱う必要性も生じるが、「同値は大とみなす」と規定すれば、アラインメント要件に適合したノードを見つけるための検索も単純に行うことができる。（詳しくは後述）

また、「メモリ確保失敗」を瞬間的に判定できるように、Worst-Fit と同様に、最大空きノードのサイズも常に管理する。

いずれにしても、ゾーンごとのツリー管理と空きサイズ管理となる。

ヒープメモリのメモリ確保戦略のイメージ：

- **【採用】検討事項：合体（Coalescing）**

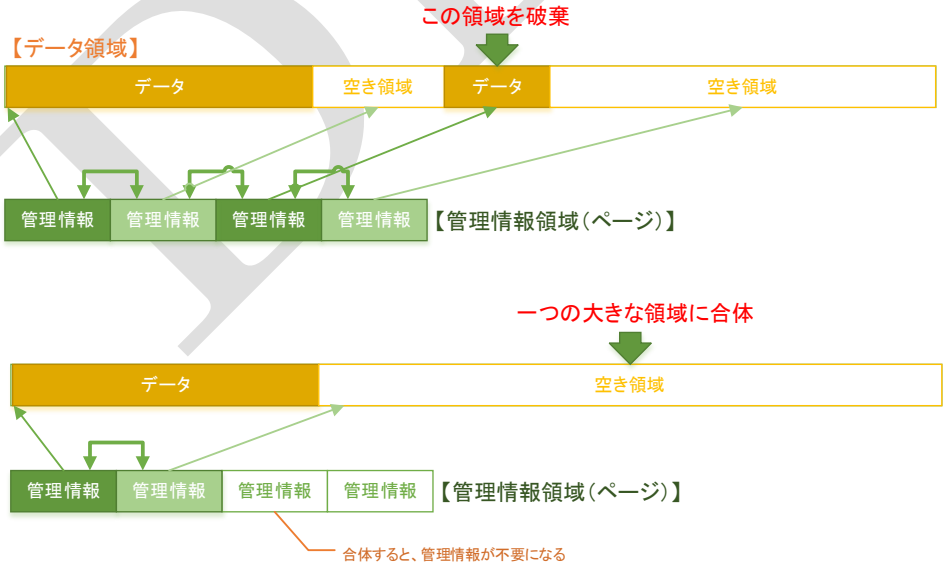
ヒープメモリの空き領域を効率的に再利用できるように、隣接する空きノード（チャンク）は合体（Coalescing = コアレッシング）を行う。

本システムもこの処理に対応する。

効率的にこの処理を行うために、隣接するメモリノードは単純な双方向リストで連結する。

メモリノードは、前述の赤黒木用の大小ノード連結と合わせて、二種類の連結リストを扱う。

ヒープメモリの合体イメージ：



● 【採用】検討事項：Best-Fit アロケーション②：プールメモリ

メモリを効率的に扱うためのアロケーション戦略はヒープメモリに限ったものではなく、プールメモリに対しても考慮する必要がある。

本システムでは、プールメモリに対して、下記のフローによる Best-Fit アロケーションを行う。

【手順①】要求サイズが 256 以下であるか判定
【手順②】必要サイズ算出：(size + 7) & 0x1f8;　※内輪で最も近い 8 の倍数
【手順③】必要サイズのプールが確保できなかったらページ追加
【手順④】ページ追加ができなかったらヒープから確保
【手順⑤】ヒープから確保できなかったら、一段大きなサイズのプールから確保
【手順⑥】それでも確保できなかったらさらに大きなサイズのプールから確保
【手順⑦】要求サイズ以上のすべてのプールで確保に失敗したら、メモリ確保失敗

「手順④」を最後のフェーズにする方法も考えられるが、これが最もメモリの無駄が少なくシンプルな手順である。「手順④」以降は例外的なフェーズと見なし、警告を出す。

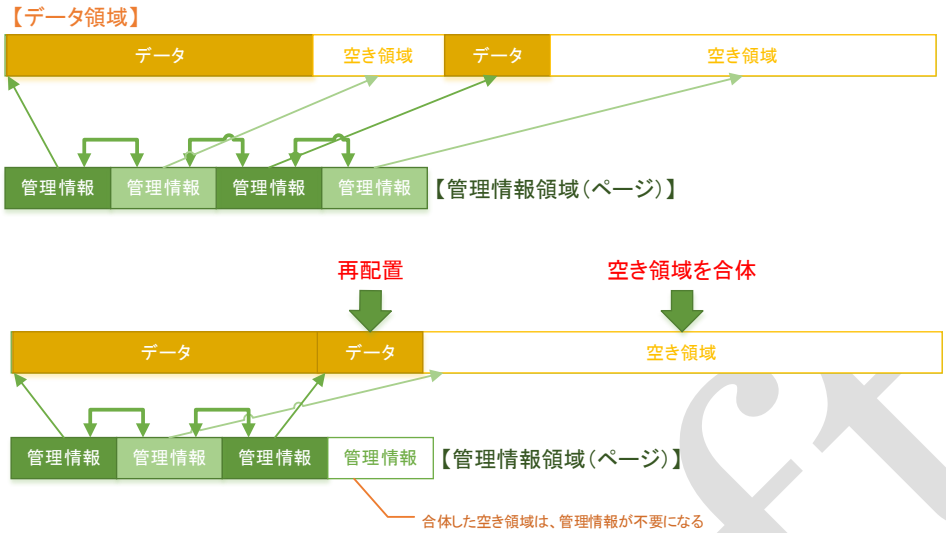
● 【採用】検討事項：メモリ再配置（Compaction）

ガベージコレクションに対応したメモリ管理では、フラグメンテーションを解消して大きな連続領域を空けるために、メモリ再配置（Compaction＝ コンパクション）に対応しているものが多い。

本システムは C/C++言語で扱うことを前提としたメモリマネージャであるが、メモリ再配置に対応する。

C/C++言語では、確保したメモリのポインタを直接操作するため、知らないうちにメモリ再配置が行われると問題となる。
ただし、本システムはメモリ管理情報を持ち、その情報が再配置されることはないため、常に管理情報を経由したメモリアクセスを行うのであれば、再配置が可能となる。

ヒープメモリの再配置のイメージ：



メモリ再配置の対象

メモリ管理情報自体は固定サイズの情報であり、フラグメンテーションが起きることもないため、再配置には対応しない。メモリ再配置に対応するのは、ゾーンのヒープメモリのみである。

同様の理由で、プールアロケータのメモリも再配置には対応しない。

ページ管理はフラグメンテーションの影響を受ける可能性があるが、2 ページ以上の連続空き領域を必要とする要件が常時発生しないこともあり、ページの再配置には対応しない。2 ページ以上の連続空き領域を必要とすることが全くないわけではないが、フラグメンテーションによりその確保ができない時は、素直に失敗とする。

メモリ再配置に求められる機能

メモリ再配置は全てのメモリノードに対して可能というものではない。メモリ再配置は、下記の通り制限を設けて使用可能とする。

- 再配置が許可されたゾーンのみ再配置可能
- 再配置が許可されたメモリノードのみ再配置可能

再配置が許可されないゾーンは、一切再配置を行わない。

再配置が許可されたゾーンの中でも、再配置が許可されたメモリノードしか再配置できない。

メモリノードの再配置許可状態は、任意に変更することを可能とし、「一時的に再配置を禁止する」といった操作が可能。

プログラミング方法の検討【ステップ①】：C++/CLI の gcnew を参考にする

「メモリ確保後に返されたポインタを、実際にどのようにして再配置可能なメモリとして扱うか」を検討する。

まず、参考までに、ガベージコレクションに対応した C++言語である「C++/CLI」の手法を確認する。

C++/CLI は、.Net Framework のための C++の拡張言語仕様である。「マネージコード」と呼ばれる、ガベージコレクション対応メモリを扱うための言語仕様が追加されている。

古くは「C++マネージ拡張」「マネージド C++」などと呼ばれていたが、現在では「C++/CLI」という呼称に定まっている。なお、「CLI」とは「Common Language Interface：共通言語基盤」のことであり、.Net Framework の基幹を構成する実行コードや実行環境などを策定した仕様である。

C++/CLI では、ガベージコレクション対応メモリを扱うために、「new」演算子の代わりに 「gcnew」演算子を使用し、かつ、「ポインタ」の代わりに「ハンドル」を使用する。「ポインタ型」は「*」を付けるが、ハンドル型は「^」を付ける。ハンドルを参照渡しする時は「^%」という構文になる（ポインタ型なら「*&」という構文に相当）。

gcnew の使用サンプル：（少し特徴的な内容を含めて記述する）

```
#include "stdafx.h"

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string>
#include <msclr/marshal.h>
#include <msclr/marshal_cppstd.h>

//-------------------------------------
//テスト用クラス
//※マネージクラスのため「ref」を付ける
//※マネージクラスは、「IDisposable」インターフェース実装を暗黙的に行い、デストラクタに対応付ける
public ref class CTest// : public IDisposable//IDisposable を明示的に宣言する必要なし
{
public:
	//メソッド
	//名前表示
	void showName()//const//マネージクラスのメソッドにこの修飾子は使えない
	{
		System::Console::WriteLine(L"名前=¥"{0}¥"", m_name);//.Net Framework 式の文字列標準出力
	}
public:
	//コンストラクタ
	CTest(System::String^ name):
		m_name(name)
	{
		msclr::interop::marshal_context context;//マネージ型の文字列をネイティブ型に変換（マーシャリング）
		printf("コンストラクタ(%s)¥n", context.marshal_as<const char*>(m_name));
```

```
                                                        //C 言語標準ライブラリ式の文字列標準出力
    }
    //デストラクタ
    //※IDisposable::Dispose()の実装として解釈される
    //※C#ではこの関数がファイナライザで、デストラクタは
    //  明示的な IDisposable::Dispose() 呼び出しの必要があるので注意
    ~CTest()
    {
        msclr::interop::marshal_context context;//マネージ型の文字列をネイティブ型に変換（マーシャリング）
        std::cout << "デストラクタ(" << context.marshal_as<std::string>(m_name) << ")" << std::endl;
                                                        //C++言語の標準ライブラリ式の文字列標準出力
    }
protected:
    //ファイナライザ
    //※ガベージコレクション時にオブジェクトを削除する際に実行
    !CTest()
    {
        System::Console::WriteLine(L"ファイナライザ({0})", m_name);//.Net Framework 式の文字列標準出力
    }
private:
    //フィールド
    System::String^ m_name;//名前
};

//--------------------------------------
//テストメイン関数
int main(array<System::String^>^ args)//.Net Framework スタイル
//int main(const int argc, const char* argv[])//C/C++言語スタイル（これも OK）
{
    {
        CTest^ obj1 = gcnew CTest(L"テスト 1");//gcnew でインスタンス生成
        obj1->showName();//アロー演算子でメンバーアクセス
        delete obj1;//delete はそのまま
    }
    {
        CTest obj2(L"テスト 2");//直接ローカル変数としてインスタンス化も問題なし
        obj2.showName();
    }
    {
        CTest^ obj3 = gcnew CTest(L"テスト 3");//gcnew でインスタンス生成
        obj3->showName();
        //delete obj3;//delete しない
    }
    //明示的なガベージコレクション呼び出し
    //※即時ゴミ回収するが、ゴミと認識されるまで時間がかかることがある
    System::Console::WriteLine(L"ToralMemory={0}", System::GC::GetTotalMemory(false));
    System::GC::Collect();
    System::Console::WriteLine(L"ToralMemory={0}", System::GC::GetTotalMemory(false));
    //終了
    System::Console::WriteLine(L"終了");
    return EXIT_SUCCESS;
}
```

↓（実行結果）

```
コンストラクタ(テスト 1)
名前="テスト 1"
デストラクタ(テスト 1)
コンストラクタ(テスト 2)
名前="テスト 2"
デストラクタ(テスト 2)
コンストラクタ(テスト 3)
名前="テスト 3"
ToralMemory=37772
ToralMemory=32908
終了
```

```
ファイナライザ(テスト 3)　　←このタイミングでファイナライザが実行
```

余談になるが、C++/CLI は普通にポインタ変数や C/C++の標準ライブラリも使用できる。

これにより、C/C++で作成したライブラリを、.Net Framework の優れた UI 開発環境に統合して使用するようなことが可能となる。複合プロジェクトのソリューションを構成して、「C++/CLI は C++コードの橋渡しにとどめ、UI 開発は C#で」といった生産性の高い構成も可能。

プログラミング方法の検討【ステップ②】: スマートポインタの活用

「メモリ確保後に返されたポインタを、実際にどのようにして再配置可能なメモリとして扱うか」の対応として、C++/CLI は、独自に言語仕様を拡張し、ハンドルを管理するスタイルとした。

このような「ポインタを隠すが、アロー演算子でポインタのように見せる」というプログラミング手法は、標準の C++言語でも可能である。

それは、デザインパターンの「プロキシーパターン」として有名な手法であり、「スマートポインタ」にも活用されている。

この手法を応用し、new 演算子で生成したオブジェクトのポインタを、スマートポインタのコンストラクタでハンドル（管理情報のポインタ）に変換すると、「メモリ再配置に対応したスマートポインタ」を実現できる。

スマートポインタの使用サンプル：　（メモリ再配置に対応したものではない）

```
#include <stdio.h>
#include <stdlib.h>

//-------------------------------------
//簡易版スマートポインタのテンプレートクラス
template<class T>
class CSmartPtr
{
	//…（略）…
};

//-------------------------------------
//テスト用クラス
class CTest
{
public:
	//メソッド
	//名前表示
	void showName() const
	{
		printf("名前=¥"%s¥"¥n", m_name);
	}
public:
	//コンストラクタ
	CTest(const char* name) :
		m_name(name)
```

```
	{
		printf("コンストラクタ(%s)¥n", m_name);
	}
	//デストラクタ
	~CTest()
	{
		printf("デストラクタ(%s)¥n", m_name);
	}
private:
	//フィールド
	const char* m_name;//名前
};

//---------------------------------------
//テストメイン関数
int main(const int argc, const char* argv[])
{
	{
		CSmartPtr<CTest> obj1 = new CTest("テスト1");//インスタンスを生成してスマートポインタで扱う
		obj1->showName();//アロー演算子でメンバーアクセス
		obj1 = nullptr;//deleteは使用せず、nullptrの代入で破棄
	}
	{
		CTest obj2("テスト2");//直接ローカル変数としてインスタンス化も問題なし
		obj2.showName();
	}
	{
		CSmartPtr<CTest> obj3 = new CTest("テスト3");//インスタンスを生成してスマートポインタで扱う
		obj3->showName();
		//obj1 = nullptr;//明示的に破棄しない
	}

	//終了
	return EXIT_SUCCESS;
}
```

↓（実行結果）

```
コンストラクタ(テスト1)
名前="テスト1"
デストラクタ(テスト1)
コンストラクタ(テスト2)
名前="テスト2"
デストラクタ(テスト2)
コンストラクタ(テスト3)
名前="テスト3"
デストラクタ(テスト3)
```

なお、プロキシーパターンとスマートポインタについては、別紙の「デザインパターンの活用」で詳しく説明している。

プログラミング方法の検討【ステップ③】：ポインタをハンドルに変換

「メモリ確保後に返されたポインタを、実際にどのようにして再配置可能なメモリとして扱うか」の対応として、スマートポインタを利用する方法を示した。

次には、スマートポインタのコンストラクタ内で、ポインタをハンドル（管理情報のポインタ）に変換する必要がある。

これには、下記の３種類の方法が考えられる。

- 方法①：メモリマネージャは、メモリ確保時のポインタと管理情報のポインタを記憶しておき、

「直前のメモリ確保情報」として参照することができる。

- 方法②：メモリマネージャにメモリのポインタを渡し、管理情報のポインタを検索して返すメソッドを用意する。
- 方法③：「方法①」と「方法②」を組み合わせ、渡されたポインタが直前に確保したメモリであれば、記憶している管理情報のポインタを即座に返し、違っていたら検索して返す。

【実装上の注意点】

「方法②」は、この要件に限らず、メモリマネージャの構造上必須の仕組みである。後述するが、赤黒木によって検索効率が最適化される。「方法①」は、TLS（スレッドローカルストレージ）を活用することで、マルチスレッドで安全かつ効率的な処理を行うことができる。

また、管理情報はヒープメモリでもプールメモリでも扱うので、スマートポインタの処理は、両者を区別することなく共通化できる。

なお、このスマートポインタは、完全に本システム（メモリマネージャ）専用の処理となる。Boost C++などの汎用ライブラリのものは使用できない。

プログラミング方法の検討【ステップ④】：ハンドルに変換したメモリの扱い

ハンドルへの変換要求があったメモリノードのみ、再配置を許可する。

デフォルトでは、メモリ確保を行っただけでは再配置は許可されない。

プログラミング上は、確保したメモリをスマートポインタで扱った時のみ、再配置が許可されるものとする。

プログラミング方法の検討【ステップ⑤】：再配置処理の実行方法

再配置処理は、自動では行わない。

特定のタイミングで任意に実行するものとする。

再配置処理は、メモリノードを動かし、一つずつ隙間を詰めて、管理情報のポインタを書き直すという処理を行う。結果として空きノードが合体されれば、管理情報の消滅とリンクの組み替えも発生する。

再配置処理は状況によっては非常に時間がかかるため、指定の時間を超えた時点で打ち切る処理とする。

再配置が完全終了する前に打ち切るようなことになっても、断片化が悪化するようなことはない。

● 【変則的採用】検討事項：スラブアロケータ（スラブキャッシュ）

Linux のメモリ管理では、大量に要するオブジェクトを効率的に扱う為に、「スラブアロケータ」を採用している。

スラブアロケータは「プールアロケータ」のように、あらかじめ大量のオブジェクトを用意してメモリ確保に備える仕組みである。要するに、Linux に実装されているプールアロケータの仕組みがこのスラブアロケータである。

スラブアロケータは、プールアロケータと管理構造がほとんど一緒だが、任意のサイズのプールを任意の数だけ用意して使用する。使用する用語は「ブロック」の代わりに「オブジェクト」、「プール」の代わりに「スラブ」と呼び、意味はほぼ一緒である。

スラブは、複数の連続ページにまたがるサイズになることもあるが、プールと同様に、ページをスラブ専用に割り付けて扱う。スラブで用意した数以上のオブジェクトが必要になれば、プールと同様にスラブを追加して連結して管理する。このスラブの集合を「キャッシュ」と呼ぶ。

また、プールアロケータと同様に、「汎用的に使える 32 バイトのメモリ」のような利用も可能だが、「クラス A 専用メモリ」のような割り当ても可能。前者を「汎用キャッシュ」、後者を「特定用途のキャッシュ」と呼ぶ。

プールアロケータとの最も大きな違いは、スラブアロケータは「キャッシュを明示的に指定してメモリを割り当てること」にある。

プールアロケータは、通常の「malloc／new」で要求されたサイズから自動的にプールを選択するのに対して、スラブアロケータでは常に使用するキャッシュを明示する。メモリ確保要求のインターフェースがそもそも異なる。

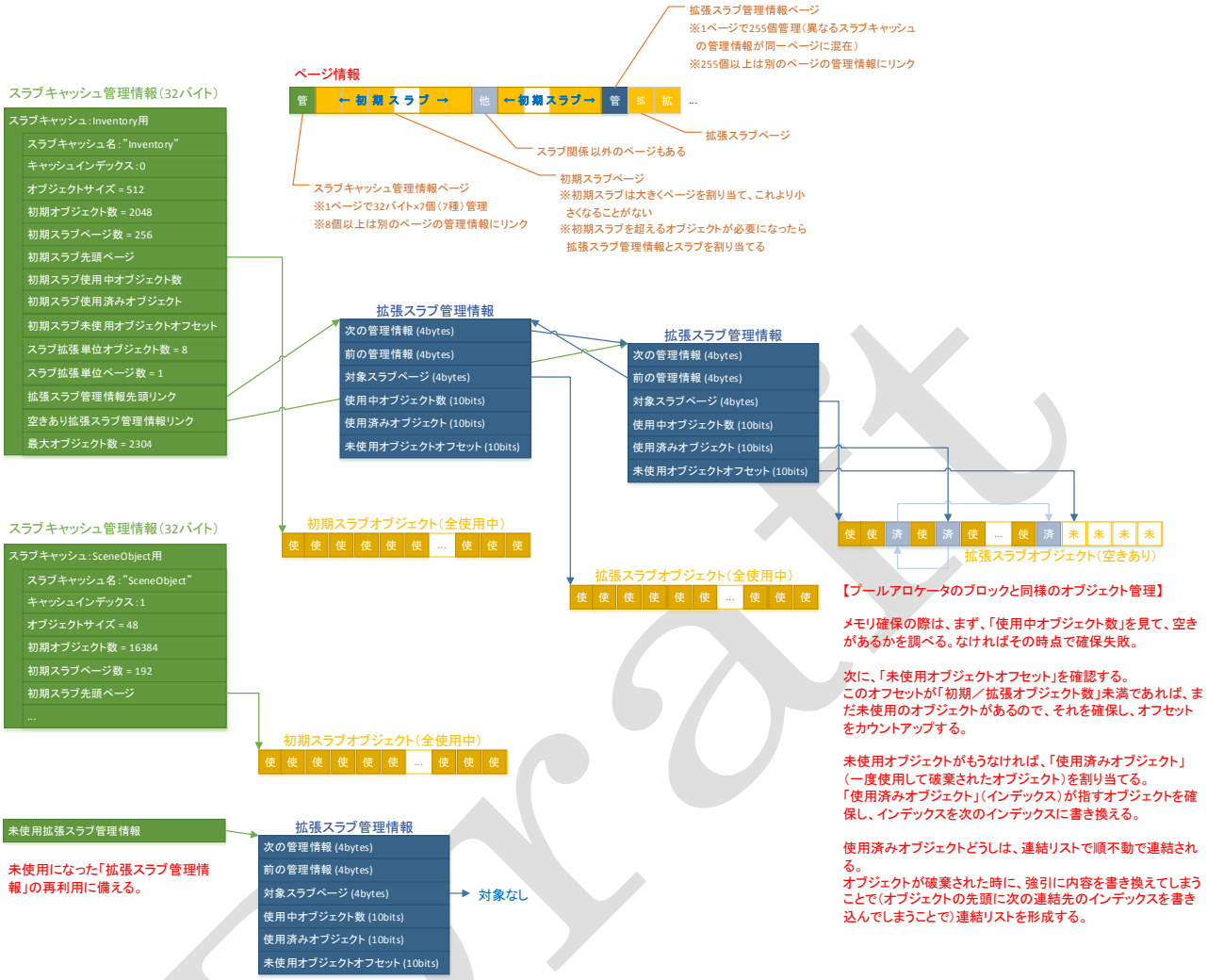
本システムはプールアロケータを実装するが、それとは別に、スラブアロケータも採用する。

スラブアロケータは、任意のオブジェクトを扱うのにとても効率的である。とくに、256 バイトを超えるクラス／構造体を大量に扱う場合にメモリ効率が良い。

プールアロケータとの棲み分けとして、基本的に「特定用途のキャッシュ」専用の仕組みとして位置づける。

なお、「ゲームのため」のメモリマネージャであることを考慮し、ゲーム開発で利用し易い仕組みの実装を検討する。

スラブアロケータのイメージ：（本システム向けの構造として大きく改変した状態）



【参考】Linux のスラブキャッシュ使用状況表示イメージ：

```
$ cat /proc/slabinfo          ← /proc/slabinfo に状況が常に記録されている
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount>
<sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
…(略)…
rpc_buffers            8      8   2048    2    1 : tunables   24   12    8 : slabdata      4      4      0
rpc_tasks              8     15    256   15    1 : tunables  120   60    8 : slabdata      1      1      0
rpc_inode_cache        8      8    832    4    1 : tunables   54   27    8 : slabdata      2      2      0
nf_conntrack_expect      0      0    240   16    1 : tunables  120   60    8 : slabdata      0      0      0
nf_conntrack_ffffffff81b16540     10     24    312   12    1 : tunables   54   27    8 : slabdata      2
2      0
fib6_nodes            42    118     64   59    1 : tunables  120   60    8 : slabdata      2      2      0
ip6_dst_cache         32     50    384   10    1 : tunables   54   27    8 : slabdata      5      5      0
ndisc_cache           22     30    256   15    1 : tunables  120   60    8 : slabdata      2      2      0
ip6_mrt_cache          0      0    128   30    1 : tunables  120   60    8 : slabdata      0      0      0
RAWv6                 67     68   1024    4    1 : tunables   54   27    8 : slabdata     17     17      0
UDPLITEv6              0      0   1024    4    1 : tunables   54   27    8 : slabdata      0      0      0
UDPv6                  6      8   1024    4    1 : tunables   54   27    8 : slabdata      2      2      0
…(略)…
size-4194304(DMA)      0      0 4194304    1 1024 : tunables    1    1    0 : slabdata      0      0      0
```

```
size-4194304          0      0 4194304    1 1024 : tunables    1    1    0 : slabdata      0      0      0
size-2097152(DMA)     0      0 2097152    1  512 : tunables    1    1    0 : slabdata      0      0      0
size-2097152          0      0 2097152    1  512 : tunables    1    1    0 : slabdata      0      0      0
size-1048576(DMA)     0      0 1048576    1  256 : tunables    1    1    0 : slabdata      0      0      0
size-1048576          0      0 1048576    1  256 : tunables    1    1    0 : slabdata      0      0      0
…(略)…
size-512(DMA)         0      0    512    8    1 : tunables   54   27    8 : slabdata      0      0      0
size-512           1000   1024    512    8    1 : tunables   54   27    8 : slabdata    128    128      0
size-256(DMA)         0      0    256   15    1 : tunables  120   60    8 : slabdata      0      0      0
size-256            889    945    256   15    1 : tunables  120   60    8 : slabdata     63     63      0
size-192(DMA)         0      0    192   20    1 : tunables  120   60    8 : slabdata      0      0      0
size-192           1955   2000    192   20    1 : tunables  120   60    8 : slabdata    100    100      0
size-128(DMA)         0      0    128   30    1 : tunables  120   60    8 : slabdata      0      0      0
size-64(DMA)          0      0     64   59    1 : tunables  120   60    8 : slabdata      0      0      0
size-64           25390  27553     64   59    1 : tunables  120   60    8 : slabdata    467    467      0
size-32(DMA)          0      0     32  112    1 : tunables  120   60    8 : slabdata      0      0      0
size-128           4941   4980    128   30    1 : tunables  120   60    8 : slabdata    166    166      0
size-32          353786 353920     32  112    1 : tunables  120   60    8 : slabdata   3160   3160      0
kmem_cache          187    187  32896    1   16 : tunables    8    4    0 : slabdata    187    187      0
…(略)…
計 187 件　前半が「特定用途のキャッシュ」で、後半が「汎用キャッシュ」
```

管理構造は Linux のものから大きく変え、前述のプールアロケータに近いものとする。Linux では CPU のキャッシュ効率やメモリ効率のために、複雑な処理を行っているが、プールアロケータと統合的に扱うことを考慮し、単純化する。

なお、上図で「〇〇管理情報」と表記しているものは、Linux のスラブアロケータでは「〇〇ディスクリプタ」と呼ぶ。オブジェクト一つ一つの状態を管理する「オブジェクトディスクリプタ」も扱っているが、本システムではそこまで対応しない。また、Linux ではスラブキャッシュディスクリプタにオブジェクトのコンストラクタとデストラクタの関数ポインタも持い、初期化／終了処理を自動的に行うが、本システムでは対応しない。

スラブキャッシュ用の領域の検討：ページ領域の利用

大きな領域をまとめて確保することが考えられるため、ヒープメモリを利用する方が扱い易いかもしれないが、下記の理由から、ページの領域を利用する。

- ヒープメモリの「ゾーン」を特定せずにスラブキャッシュを使用したい。
- ゾーンのメモリ再配置や全削除といった要件の邪魔にならないようにしたい。
- プールアロケータや本家（Linux）のスラブアロケータと同様に、自動拡張に対応したい。

スラブキャッシュの管理方法

スラブキャッシュの種類は、プールメモリのように 32 種類に固定されない。この対処として、スラブキャッシュの管理情報は、専用ページを設けて扱う。

また、プールアロケータと同様に、スラブの管理も専用のページを設ける。

これにより、スラブアロケータは下記の 3 種類のページを扱うものとなる。

- スラブキャッシュの管理ページ（１ページずつ割り当て、必要に応じて追加して連結する）
- スラブの管理ページ（１ページずつ割り当て、必要に応じて追加して連結する）
- スラブオブジェクトのページ（複数連続ページを割り当て、必要に応じて追加する）

スラブアロケータの使用上の注意

スラブキャッシュは自由度が高い反面、連続空きページを必要とするなど、ページ領域の断片化の影響を受けるものとなり、確実性が低い。

また、ページ領域の全体サイズは比較的小さいため、スラブキャッシュの増大が、重要な管理ページの圧迫などを招く恐れがある。

以上のことから、スラブアロケータを使用する際は、下記の注意点を設けるものとする。

- ページ領域に十分空きを残すように、スラブキャッシュにメモリを割り当てすぎない。
- 頻繁なスラブキャッシュの追加は行わず、ゲームの始めの方で大きく割り当てて使い続ける。
- １ページ（4KB）を超えるオブジェクトはなるべく扱わず、自動拡張は極力１ページずつにする。
- なるべく自動拡張が発生しないようにする。自動拡張はいざという時の救済措置的位置づけで考える。

より、問題を抑えるためにも、「スラブキャッシュはページ領域全体の 50%までしか使えない」といったシステム的な制限を設けることも要検討。

スラブアロケータを使用したプログラミングのイメージ

スラブアロケータを使用したプログラミングは、下記の手順となる。

- 手順①： スラブキャッシュの作成（キャッシュ名、オブジェクトサイズ、初期オブジェクト数、一回の拡張で追加するオブジェクト数、最大オブジェクト数を指定し、「ハンドル」（スラブキャッシュ管理情報の先頭ポインタ）を受け取る）
- 手順②： スラオブジェクトの生成（スラブアロケータ専用メソッドを使用し、「ハンドル」を指定してメモリを割り当て）
  ※専用の配置 new を用意し、コンストラクタ呼び出しに対応する。
- 手順③： スラオブジェクトの破棄（スラブアロケータ専用メソッドを使用し、「ハンドル」とオブジェクトのポインタを指定してメモリを解放）
  ※専用の配置 delete を用意し、デストラクタ呼び出しに対応する。
- 手順④： スラブキャッシュの破棄（「ハンドル」を指定してスラブキャッシュを解放）

スラブオブジェクトの管理

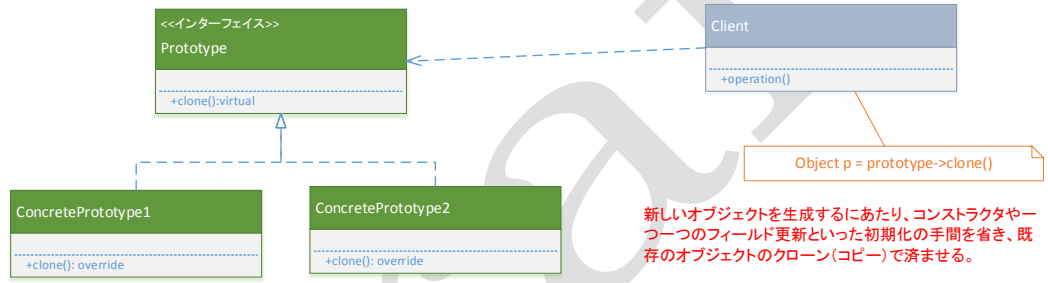
要求に応じてオブジェクトを割り当てた場合、ヒープメモリのメモリノード管理と同じく、個々の管理情報を持ち、一様にデバッグ情報などを扱うことができるようにする。ただし、スマートポインタの参照カウンタ管理や、カテゴリによるメモリ制限は、スラブオブジェクトには必要ない。

スラブアロケータの活用①：プロトタイプパターンの実践

スラブアロケータを、デザインパターンの「プロトタイプパターン」の実践に活用する。

まず、一般的なプロトタイプパターンのクラス図を示す。

一般的なプロトタイプパターン：



例として、インベントリデータへのプロトタイプパターンの適用を考える。

インベントリデータには「剣」「盾」「回復アイテム」など様々な種類があり、それぞれ「基底アイテムクラス」から派生した「剣クラス」「盾クラス」「回復アイテムクラス」で構成される。それぞれの基本パラメータは別途静的データを参照するとしても、「耐久力」や「残り使用可能回数」など、それぞれ特有の動的データも含む。その値は、「短剣」や「ロングソード」など、個々のアイテムに応じて異なる。

インベントリに新たにアイテムを追加する際は、最初にアイテムの種類を調べ、種類に合ったクラスのインスタンスを生成する（メモリ確保を伴う）。その上で、アイテム固有のパラメータを調べて初期値を設定する。

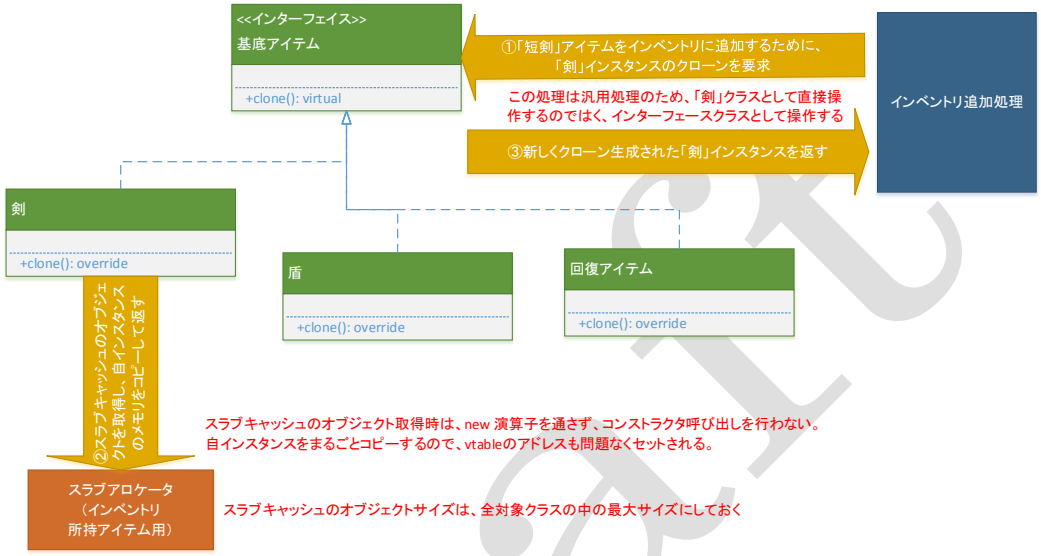
この初期化の処理を「プロトタイプパターン」を使用して簡略化する。

まず、プロトタイプとなるインスタンスをあらかじめ用意しておく。「短剣」、「ロングソード」のレベルではインスタンスが多くなりすぎるので、クラスの種類数分だけ用意する。

続いて、アイテムを入手した際は、アイテムの種類に応じたインスタンスを選んでクローンを生成し、そのクローンに対してアイテム固有の初期化を行う。

クローン処理の内部では、スラブアロケータからのメモリ確保を行う。この時、コンストラクタの呼び出しはなく、クローン元のインスタンスのメモリコピーで初期化する。スラブキャッシュのオブジェクトサイズは、最も大きな派生クラスのサイズに合わせておく。

処理イメージ：



このクローンメソッドはどのようなインスタンスに対しても使えるので、ゲーム中の状態に合わせた利用も便利に使える。

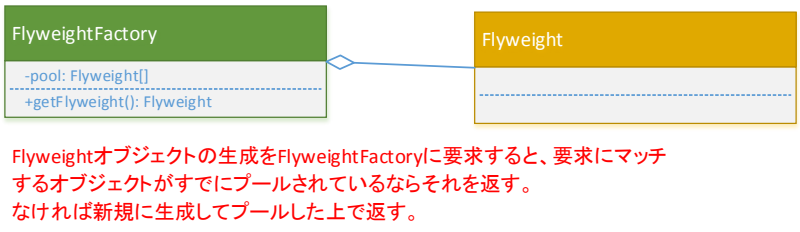
例えば、「回復アイテムの半分を倉庫に移す」という処理を実行したければ、現在のイベントリのインスタンスのクローンを生成して個数だけ調整すれば手っ取り早い。

スラブアロケータの活用②：フライウェイトパターンの実践

先の「プロトタイプパターン」の対応をさらに発展させ、デザインパターンの「フライウェイトパターン」と組み合わせて活用する。

まず、一般的なフライウェイトパターンのクラス図を示す。

一般的なフライウェイトパターン：



先の「プロトタイプパターン」の例では、クローン用に、「剣クラス」「盾クラス」「回復アイテムクラス」のインスタンスをあらかじめ用意しておくものとした。

フライウェイトパターンとの併用により、そのインスタンス生成は「必要になった時に初めて」行い、以後、「すでに存在するなら再利用」する。

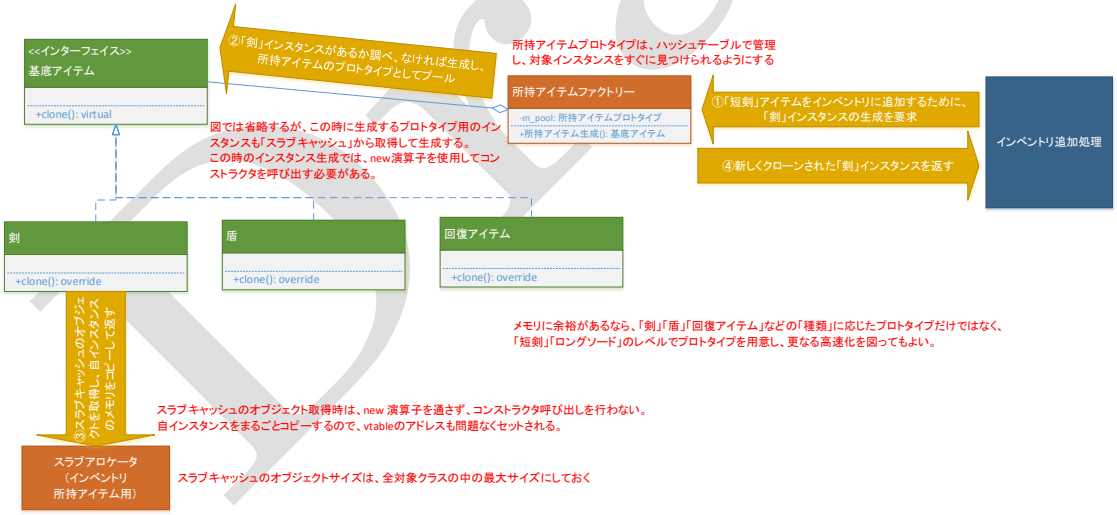
「所持アイテムファクトリークラス」を用意し、内部にはアイテム種別に応じたプロトタイプ（インスタンス）のハッシュテーブルを持つ。

アイテムを入手した際は、ファクトリークラスにアイテム種別を渡せば、（必要に応じて）プロトタイプを生成した上で、そのクローンを生成して返す。プログラミングインターフェースとしても、すっきりとまとまった構造になる。

なお、プロトタイプ自体もスラブアロケータから取得したインスタンスで扱う。プロトタイプ生成時は配置 new を利用してコンストラクタ呼び出しを伴う初期化を行う。

また、フライウェイトパターンを組み合わせた場合もクローンメソッドは隠ぺいせず、前述の「回復アイテムの半分を倉庫に移す」のような用途には使えるものとする。

処理イメージ：



余談だが、フライウェイトパターンは、この例での「プロトタイプ」のように、一度インスタンスを生成したら、後は内容が変わらないようなものを扱うのが通常である。このような「状態が変わらない（変えることができない）」オブジェクトを「イミュータブル（immutable）なオブジェクト」と呼ぶ。対義語は「ミュータブル」（mutable）。

ミュータブルなオブジェクトにフライウェイトパターンを適用するケースとしては、例えば、HTML に埋め込まれた画像をダウンロードするようなケースが考えら

れる。HTML 内では同じ画像が数か所に使用されることも多く、それらは同じインスタンスを使いまわして共有する。画像のダウンロードには少し時間がかかるが、一つダウンロードすれば共有できる。

このようなケースにフライウェイトパターンを導入して、画像データ（の管理）を共有する。画像データは「ダウンロード要求」「ダウンロード中」「ダウンロード完了」と、その状態を変えていくが、最初の「ダウンロード要求」の時点から共有して扱う。

別紙の「開発の効率化と安全性のためのリソース管理」で示すリソース共有の仕組みがこれと同様である。

▼ 可搬性のための方針：独立したグラフィックメモリの管理のために

昨今では、ユニファイドメモリが主流になりつつあるが、PS3 などのように、グラフィックメモリが独立しているものも多い。

本システムはヒープメモリの管理情報が独立しており、ヒープのデータ部には一切管理情報が置かれないため、論理的なメモリ空間の管理が可能である。

これを利用し、「グラフィックメモリ」のように、メモリマネージャが直接アクセスできないメモリ空間のメモリ配分のみを管理することも可能とする。

その際は、「プールアロケータ」と「スラブアロケータ」を無効化し、物理メモリ全域が「ヒープメモリ管理情報のためのページ」として構成される。

▼ 処理効率向上のための方針：高速なメモリ確保と解放のために

メモリ効率向上のための方針を固めたことに基づき、それを高速に処理するための方針を固める。

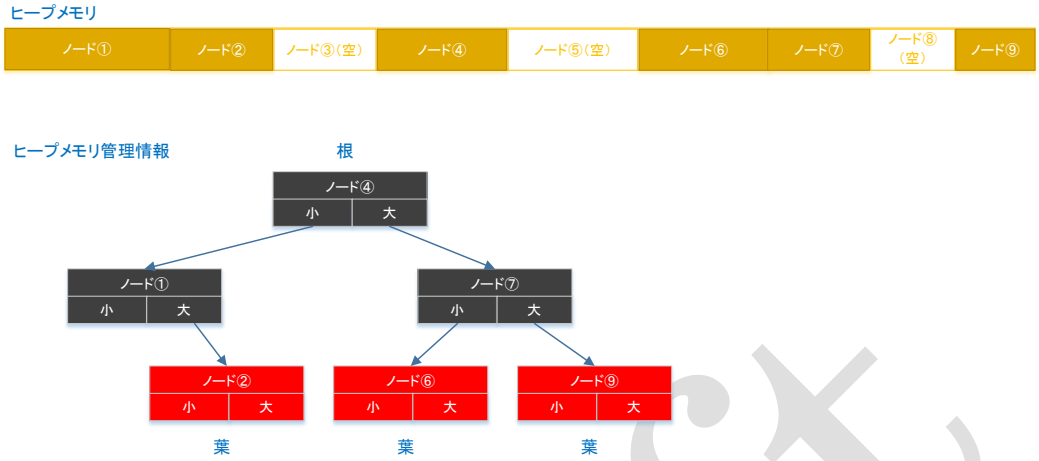
● 【採用】検討事項：メモリノード管理情報の赤黒木（ポインタ昇順）

本システムのヒープメモリは、データ部とメモリノード管理情報（ヘッダー部）が分離しているため、メモリを破棄する場合などに、データ部のポインタから管理情報を検索する必要がある。

この検索を高速化するために、データ部のポインタの昇順に基づく二分木で管理情報を構成する。

さらに、二分木の検索性能を維持するために、平衡木を用いる。平衡木には情報量の少ない赤黒木（red-black tree）を採用する。

赤黒木のイメージ：



赤黒木の採用例は多い。Linux のメモリリージョン管理や STL の map, set などに用いられている。

赤黒木のアルゴリズムについて詳しい説明は省くが、下記の特徴と条件のもとに平衡木を保つ仕組みであることだけ説明しておく。

【特徴】

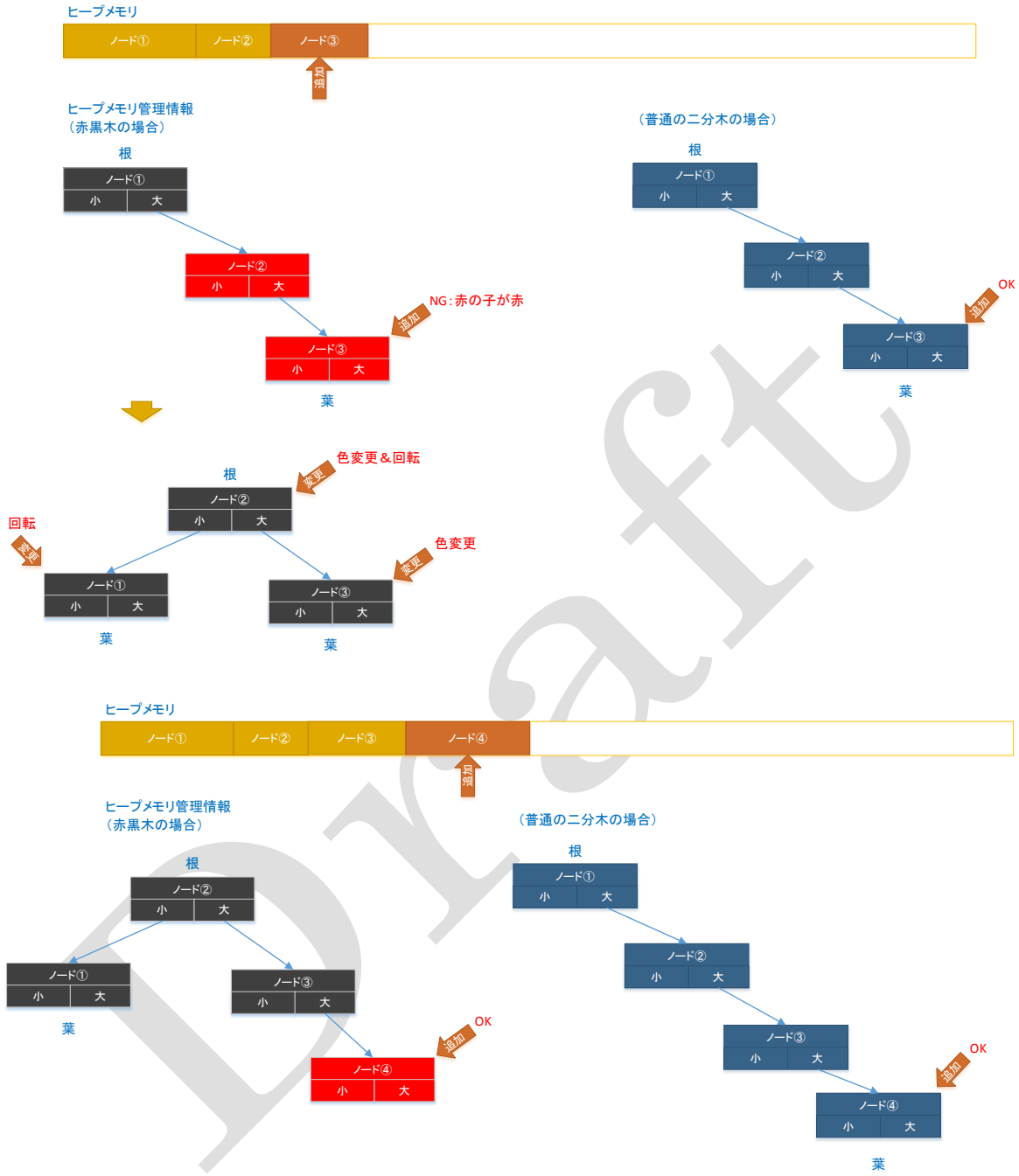
- 二分木である。
  - ノードの左側には、キーの値が自ノードより小さいノードを連結。
  - ノードの右側には、キーの値が自ノードより大きいノードを連結。
- 平衡木である。
  - 常に左右の木のバランスを保ち、探索時間の大きな劣化がない木構造。

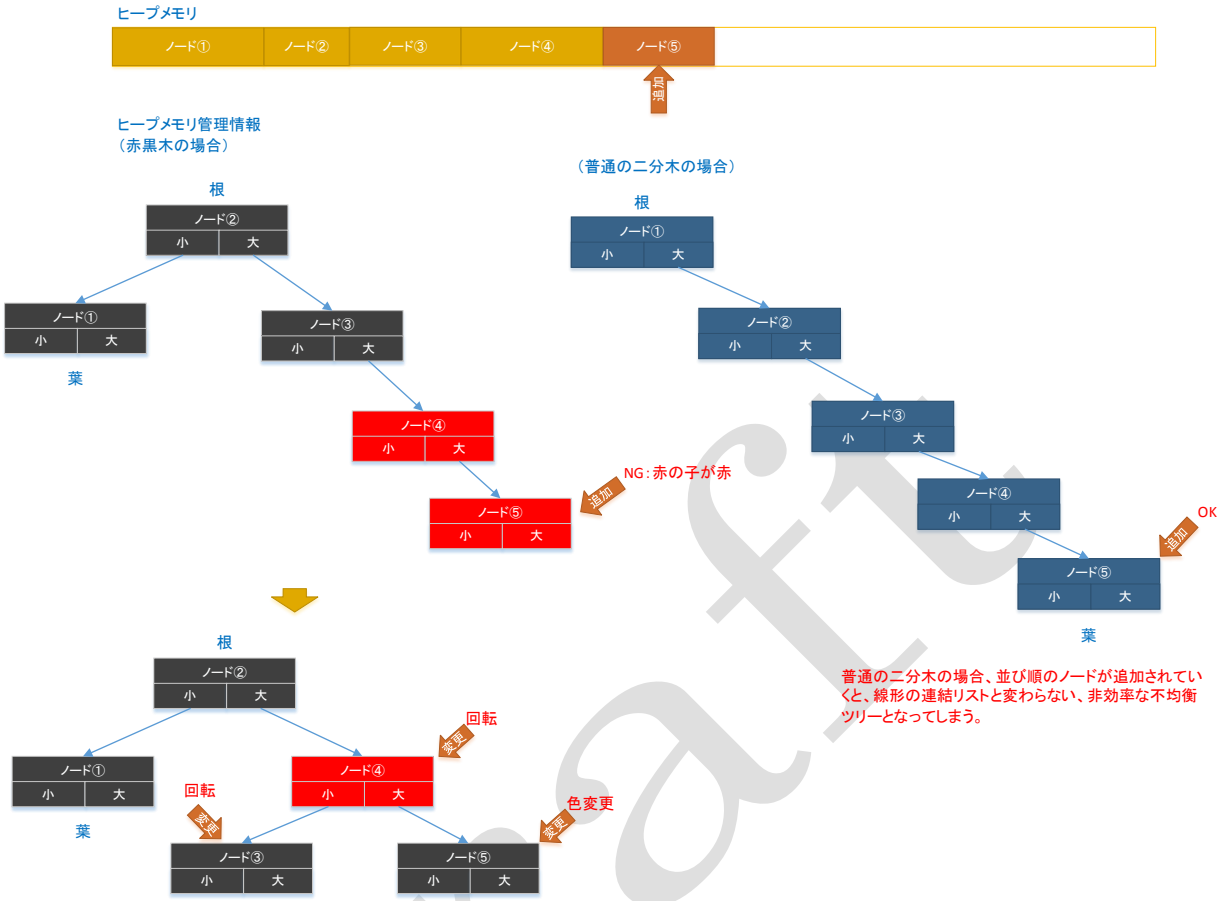
【条件】

- 各ノードは「赤」か「黒」の「色」を持つ。（1 ビット情報）
- 「根」（root）は必ず「黒」。
- 「赤」の子は必ず「黒」。
  - 待遇により、「赤」の親は必ず「黒」。
  - 「黒」の子は「赤」でも「黒」でも良い。
- すべての「根」から「葉」までの経路上にある「黒」の数は、あらゆる経路で一定。

ツリーの平衡性を保つために、ノードの追加・削除の際に、「回転」という操作を行う。

赤黒木の回転イメージ：（実際はけっこう複雑なアルゴリズム）


ヒープメモリ
ノード①
ノード②
ノード③
追加
ヒープメモリ管理情報
（赤黒木の場合）
根
ノード①
小
大
ノード②
小
大
NG：赤の子が赤
追加
ノード③
小
大
葉
（普通の二分木の場合）
根
ノード①
小
大
ノード②
小
大
OK
追加
ノード③
小
大
葉
色変更＆回転
変更
根
ノード②
小
大
回転
変更
色変更
変更
ノード①
小
大
葉
ノード③
小
大
葉
ヒープメモリ
ノード①
ノード②
ノード③
ノード④
追加
ヒープメモリ管理情報
（赤黒木の場合）
根
ノード②
小
大
ノード①
小
大
葉
ノード③
小
大
OK
追加
ノード④
小
大
（普通の二分木の場合）
根
ノード①
小
大
ノード②
小
大
ノード③
小
大
OK
追加
ノード④
小
大
葉



連結するメモリノード管理情報について

この赤黒木は、前述の通り「ポインタからメモリノード管理情報を検索するため」のものである。

また、メモリノード管理情報には「データの位置」（ポインタのオフセット）や、「カテゴリ」、「要求サイズ／アラインメント」、「参照カウンタ」、「デバッグ情報」を記録するため、ヒープメモリだけではなく、使用中のプールメモリブロックとスラブオブジェクトも扱う。

以上のことから、この赤黒木は、「ヒープ」「ゾーン」「プール」「スラブ」の垣根を越えて、全て一まとめに連結して扱うものとする。

これにより、メモリ解放処理が単純化される。メモリノード管理情報には「メモリの種類（ヒープ／ゾーン／プール／スラブ）」や、「プール／スラブの管理情報の位置」も記録し、すぐに処理できる状態にする。

なお、「ヒープの空きノード」は連結しない。詳しくは後述するが、ツリー用のリンク（メンバー変数）を再利用し、「空きノード検索専用」のツリーを構成する。

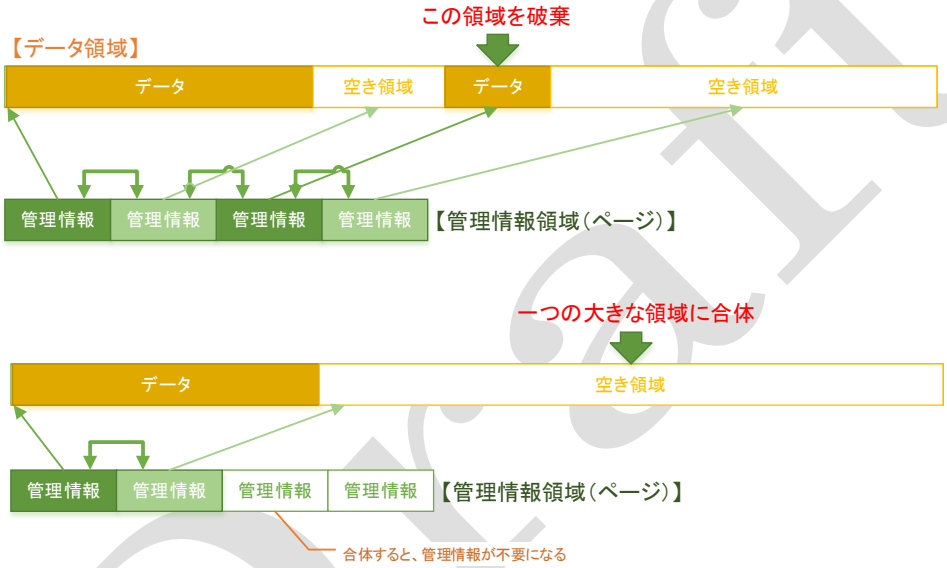
- **【採用】検討事項：ヒープメモリノード管理情報の隣接ノード連結**

ヒープのメモリノードを削除した際は、隣接する空きノードとの合体操作を行う。また、メモリ再配置の際は、前方に隣接するノードが空いているかをチェックする。

これらの操作を高速に行うために、ヒープメモリノード管理情報は、赤黒木による連結のほか、隣接ノードとの単純な双方向リンクも構成する。

なお、この連結リストは「ゾーン」ごとに管理する。

ヒープメモリの合体イメージ：（再掲載）



- **【採用】検討事項：ヒープメモリノード管理情報の空きサイズ降順赤黒木**

ヒープメモリの空きノードを再利用する際、「Best-Fit アロケーション」を行う。

Best-Fit アロケーションを高速に実現するために、ヒープメモリの空きノードをサイズ降順で連結する二分木を構成する。

二分木の検索性能を維持するために、ポインタ昇順ツリーと同様に赤黒木を用い、その連結情報領域（メンバー変数）を共用する。

なお、この空きノードツリーは、ポインタ昇順ツリーと異なり、「ゾーン」ごとに構成する。

【要検討】空きノードツリーの連結情報の場所

赤黒木の連結情報をポインタ昇順ツリーと共用するのは、メモリを効率化するためであり、データ部のポインタからヒープノード管理情報を割り出す必要性がないからである。

もし、空きノードであってもデータ部のポインタからヒープノード管理情報を割り出す必要がある場合は、空きノードツリーを構成するための連結情報を別途設けなければならない。

その場合、無駄に管理情報を肥大させないように、下記のいずれかの方法を検討する。

- 「参照カウンタ」や「デバッグ情報」など、「空きノード」には不要な場所を共用する。ただし、その場合、「リリースビルド時にデバッグ情報部を削ってサイズ縮小」といったことができなくなる点に注意。
- データ部を書き換えて、空きノードの連結情報を置く。最低 8 バイトアラインメントされることを前提とすれば、大小リンク情報を 8 バイトで扱うことができる。ただし、この場合、「独立したグラフィックメモリ」のように、メモリマネージャが直接操作できないメモリを管理できなくなる点に注意。
- やはりベストは「ポインタ昇順ツリーと共用」なので、ポインタから空きノードの管理情報を検索する処理が必要ないようにする。

【検討】赤黒木のキー重複について

二分木は、「キーの重複を許さない」のが普通である。

STL でも、赤黒木を用いる map, set は重複を許可していない。(参考までに、STL でキーの重複を許容するデータを扱うには、multimap, multiset を使用する)

この「空きノード降順ツリー」では、どうしてもキーの重複が発生する。このツリーにおける「キー」とは、「空きノードのサイズ」である。

この問題は、ツリーにノードを追加する際、「重複するキーは大とみなす」(もしくはその逆)と規定して処理する。

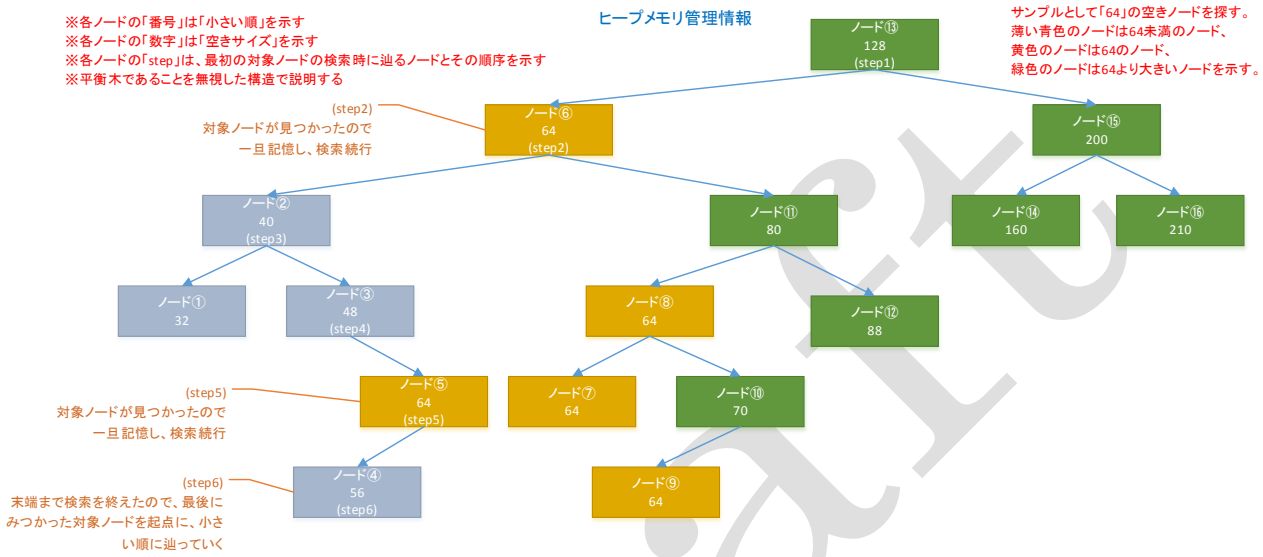
空きノードツリーの検索要件は、Best-Fit アロケーションなので、最初に検索するのは「要求サイズと同じノード」もしくは「要求サイズ以上で最小のノード」である。アラインメント指定に対応する必要もあるので、サイズが一致していればよいというものではなく、検索してみつかった最小ノードを基点に、一つずつ大きいノードをチェックして、要求サイズとアラインメントに適合する空きノードを探し出す。

最初のノードを探す際は、「同値」が存在する可能性があるため、一つノードが見

つかっても、「小」方向に同値が隠れている可能性があるため、葉（末端）まで検索を続けなければならない。（赤黒木はノードの回転があるので、「重複するキーは大とみなす」と規定しても、小方向に同値が潜む可能性がある）

最小ノードを見つけたあとは、一つずつ次に大きいノードをチェックする。

空きノードツリーのイメージ：



なお、「次のノード」の探索は少し面倒。「Ａ：大側の子の小側の葉（末端）」（例：⑥→⑦、⑧→⑨）もしくは「Ｂ：小側の子になっている先祖の親」（例：⑤→⑥、⑦→⑧、⑨→なし）を辿る（「Ａ」から先に探し、なければ「Ｂ」を探す）。

各ノードは親ノードへのリンクを持たないので、再帰処理を行う必要がある点に注意。（ちなみに、効率化のために再帰処理を避けて、sta::stack のようなスタック処理を用いるのも良い）

▼ ガベージコレクションの方針：安全なメモリ操作のために

本システムは、基本的にガベージコレクションには対応しない。

しかし、手動メモリ再配置（コンパクション）と、スマートポインタ支援をサポートする。

なお、メモリ再配置については前述のとおり。

● 【採用】検討事項： 参照カウンタ

メモリノード管理情報には 4 バイトの「参照カウンタ」を含め、スマートポインタをサポートする。

Boost C++ のスマートポインタは、実データのポインタと参照カウンタのための領域を 8 バイト確保するので（ポインタサイズが 4 バイトとした場合）、この情報が管理情報に含まれているとメモリ効率がよくなる。

ただし、全てのメモリがスマートポインタを必要としているわけではないので、全体的に見ればこの情報がメモリを圧迫することになりかねない。

1～数ビットに参照カウンタを節約する技法もあるが、スマートポインタ目的の参照カウンタとするため、採用できない。

コンパイル時の指定によって参照カウンタの有無を選択できるようにすることも考えられるが、その場合、スマートポインタ側の処理が大きく変わり（参照カウンタ用の 4 バイトをアロケートする必要がある）、コードの静的分岐が煩雑になる。

以上のことから、単純に参照カウンタは全てに持つものとして扱う。

なお、参照カウンタが有効に機能するのは、スマートポインタを使用した時のみである。参照カウンタが 0 になったことを確認してメモリを破棄するのもスマートポインタの処理であり、メモリマネージャがメモリの破棄を判断することはない。

なお、スマートポインタについては、前述の「メモリ再配置」の検討事項の中で詳しく説明している。

● 【変則的採用】検討事項：マークスイープ GC によるメモリリーク検出

本システムはガベージコレクションを採用しないが、メモリリーク検出のためのデバッグ機能として、マークスイープ GC（ガベージコレクション）の機能は用意するものとする。

（注：以下に示すのは、実現性が確認できていない処理要件を含むので、この実装は「要検討」である）

メモリノード管理情報に 1 ビットの「マーク」を設け、「メモリリーク」を探すための仕組みとして実装する。処理手順は下記の通り。

【処理手順①：準備フェーズ】

- メモリノード管理情報の「ポインタ昇順ツリー」を辿り、全ての（使用中メモリの）メモリノード管理情報の「マーク」を OFF にする。
  - 「空きノード」の管理情報は無視する。
  - 「処理手順④：スイープフェーズ」が同様の処理を行うため、本来このフェーズは不要。
  - ゲーム中の任意のタイミングでこの機能を有効化することを想定しているため、このフェーズを必要とする。

【処理手順②：通常処理】

- 新たに確保したメモリは、同時に「マーク」を ON にする。

【処理手順③：マークフェーズ】

- 「静的変数領域」(.data セクションと.bss セクション)、メモリマネージャが管理する「ヒープ領域」「プールメモリ領域」「スラブ領域」、および、「その他のメモリ管理領域」（異なるメモリマネージャで管理している領域）の全てに対して、下記の走査を行う。
  - 4 バイトずつメモリをチェック。（64bit ポインタの環境なら 8 バイトずつ）
  - その値が、メモリマネージャの管理領域の範囲内であるかチェック
  - その値が、8 の倍数であるかチェック
  - メモリマネージャの「メモリノード管理情報」のポインタに一致するなら、その管理情報の「マーク」を ON にする（そのアドレスが「ページ領域の範囲内か？」「管理情報のアラインメントに一致しているか？」「管理情報を構成するメンバーとみなせるか？」を判定する）
  - メモリマネージャの「データ領域」のポインタに一致するなら、その管理情報の「マーク」を ON にする（ポインタでツリーを検索し、対象管理情報が見つかるかどうかで判定する）
  - スタック領域はチェックの対象外とする。（本来はチェックする必要がある）

【処理手順④：スイープフェーズ】

- メモリノード管理情報の「ポインタ昇順ツリー」を辿り、全ての（使用中メモリの）メモリノード管理情報の「マーク」を確認し、OFF のものがあればメモリリークと判断し、ログなどに通知する。
  - 本来のガベージコレクションでは、このフェーズでメモリを解放するが、本システムでは「解放し忘れの報告のみ」とする。
  - この時、ON のマークは OFF にする。
  - 「空きノード」の管理情報は無視してよい。

メモリ走査の際に、対象ポインタと判定したものが、ポインタと無関係の数値である可能性もあるが、それによってメモリリークと誤認されることはないので不問とする。（メモリリークを見過ごす可能性はある）

スタック領域を走査の対象としないので、メモリリークの誤認も起こりえる。

非常に処理も重くなるので、デバッグメニューなどから機能を ON/OFF できるようにし、メモリリークを調べたいタイミングで実行するようにする。

- **【採用】検討事項：ソフトリセット時のメモリリーク報告**

別紙の「マルチスレッドによるゲームループ管理」に記述しているとおり、ゲームシステムが「完全なソフトリセット」に対応するなら、そのタイミングでメモリマネー

ジャも一旦破棄することができる。

その際、明示的に破棄されずに残ったメモリ（メモリノード管理情報）をダンプ出力することで、メモリリーク（の可能性があるもの）を検出することができる。

▼ 開発支援機能の方針：的確な問題追及のために

メモリ確保時に、なるべく多くの情報をメモリノード管理情報に記録することで、問題発生時の原因追及を少しでも容易なものにする。

- **【要検討】検討事項：メモリノード管理情報に記録するデバッグ情報**

デバッグ情報もメモリを圧迫するので、どれだけの情報を扱うかは要検討。

以下、記録されていると便利な情報を示す。

- ソースファイル名（__FILE__マクロ）
- ソースファイル行番号（__LINE__マクロ）
- ソースファイルのタイムスタンプ（__TIMESTAMP__マクロ）
- 関数名（__FUNCSIG__ , __FUNCTION__, __PRETTY_FUNCTION__ マクロ）
- コールポイント名
- ゲーム時間
- タイプ名
- マークスイープ GC 用のマーク

これらの情報は、アロケート処理（new 演算子など）を呼び出すマクロを用意することにより、暗黙的に受け渡すことが可能。

「ソースファイル名」「ソースファイル行番号」「ソースファイルのタイムスタンプ」は、マクロを並べて記述する事で、プリプロセッサ時に結合（トークン連結）して一つの文字列にしておくことが可能。（別紙の「本当にちょっとしたプログラミング Tips」にも詳しく説明している）

なお、GCC の関数名を返すマクロ（__FUNCTION__, __PRETTY_FUNCTION__）は、__FILE__や__TIMESTAMP__などと異なり、const char* を返すので、文字列のトークン連結が使えない点に注意。デバッグ情報として記録する場合、ファイル名と関数名は分けて扱う必要がある。

「コールポイント名」は、直接メモリ確保を行った処理ではなく、その処理を呼び出した処理を記録するためのもの。別紙の「効果的なデバッグログとアサーション」にて、コールポイントの仕組みを説明する。

「ゲーム時間」は、ゲームを開始してからのプレイ時間。ゲームループが管理する情報。浮動小数点情報で単位は秒。

「タイプ名」は、「typeid(T).name()」で取得できるデータ型の名称。仮想クラスの型名を取得するには、コンパイラオプションで RTTI が有効になっている必要がある。RTTI については、別紙の「プログラミング禁則事項」にて説明。

参考までに、デバッグ情報以外に、メモリノード管理情報に記録する情報を列挙する。

- 赤黒木ツリー（大・小ノード連結ポインタ）
- 赤黒木の色（1 ビットのフラグ）
- 隣接ノードとの双方向連結ポインタ（ヒープのみ）
- カテゴリ ID（ヒープとプールのみ）
- ゾーン ID（ヒープのみ）
- メモリ要求サイズ
- メモリ要求アラインメントサイズ
- 実データのポインタ
- 参照カウンタ（ヒープとプールのみ）
- メモリ種別（ヒープ／プール／スラブ）
- プール管理情報のポインタ（プールのみ）
- スラブ管理情報のポインタ（スラブのみ）
- メモリ再配置許可フラグ（ヒープのみ）
- メモリ再配置のためのスピンロック（ヒープのみ）

デバッグ情報と合わせて、これらの情報をできる限り凝縮し、1 ページ 4KB に隙間なく収まるようにする。なお、ページの先頭には管理ページどうしの連結情報を持つ。

▼ マルチスレッドに対する方針：スレッドセーフなメモリマネージャのために

本システムは、スレッドセーフなメモリマネージャとするため、ロック制御の方針を固めなければならない。以下で示すロックは、一部を除いて、単純なスピンロックで実装する。

- ページのロック ..... ページの割り当て／返却処理の際にロック。
- プールのロック① ..... 32 種類のプールごとに、プール管理情報の割り当て／返却、および、プールからのメモリブロック確保／削除の際に、それぞれロック。他の種類のプールにはロックの影響がない。
- プールのロック② ..... プール管理ページの追加／返却の際に、全プールをロッ

ク。

・スラブのロック① ..... スラブごとに、スラブ管理情報の割り当て／返却、および、スラブからのオブジェクト確保／削除の際に、それぞれロック。他のスラブにはロックの影響がない。

・スラブのロック② ..... スラブキャッシュの追加／削除の際に、全スラブをロック。

・スラブのロック③ ..... スラブ管理ページの追加／返却の際に、全スラブをロック。

・ヒープのロック ..... ゾーンごとに、要求されたメモリの確保／削除処理をロックする。

・メモリノード管理情報のロック
..... メモリノード管理情報のツリー追加／削除操作の際にロック。全てのメモリ操作時に発生するロック。単純に検索するだけの場合もあるので、リード・ライトロックを用いる。

・メモリ境界移動時のロック . メモリ境界移動時、および、物理メモリ割り当て増減時は、複数の処理同時実行されないようにロックする。

・メモリ再配置のロック ..... メモリ再配置時は、移動中のメモリノードをロックし、スマートポインタアクセスと排他制御する。

・スマートポインタアクセスのロック
..... （同上）

・メモリ再配置のロック ..... メモリ再配置時は、移動中のメモリノードをロックする。

・参照カウンタの更新 ..... ロックしない。ただし、アトミック操作が必要。

■■以上■■

■ 索引

は

ひ

ふ

へ

ま

め

ゲーム制御のためのメモリ管理方針

以　上