開発の効率化と安全性のためのリソース管理

– マルチスレッドシステムを支える最重要要素 –

2014 年 2 月 24 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 2 月 24 日	板垣 衛	（初稿）

■ 目次

■ 概略

マルチスレッドに最適化したリソース管理システムを設計する。

ファイルマネージャと連携したリソースの自動リロード機能や、シーンマネージャと連携したリソースの一時削除と復元機能、安全かつ柔軟にリソースを扱うためのプログラミングモデルについても解説する。

■ 目的

本書は、マルチスレッドに最適化したリソース管理システムを構築することにより、多数のプログラマーが参加するプロジェクトであっても、安全かつ効率を損ねることなくマルチスレッドを活用したシステムを開発できるようにすることを目的とする。

リソースアクセスのプログラミング手法を標準化することで、自然とマルチスレッドに最適なプログラミングを行えるものとする。そして、マルチスレッドで同時アクセスを要するデータのほとんどを扱うものとすることで、問題の発生を抑える。

分かり易く統一的なプログラミング手法により、開発効率を向上させつつ、処理のパフォーマンスを高めることを大きな目的とする。

■ 要件定義

▼ 基本要件

本書が扱うシステムの基本要件は下記のとおり。

- 基本的に、「モデル」や「テクスチャ」などのリソースファイルを読み込んで、メモリ上で管理するための、汎用リソースマネージャとする。
- マルチスレッドに最適化し、リソース構築・アクセスの両面において、最良のパフォーマンスを実現しつつ、マルチスレッド特有の問題を極力抑えた安全性の高いシステムを構築する。
- ファイルの読み込みはリソースマネージャが行い、リソースを扱う（ゲーム側の）各処理系が読み込み処理を行う必要はないものとする。

・ 既に構築済みのリソースと同じリソースの構築要求があった場合など、リソースはできる限り自動的に共有し、無駄な構築処理を省き、処理効率とメモリ効率を最大限に高めるものとする。

・ 同様に、リソース削除のタイミングも、共有状況などに基づいて、リソースマネージャが適切に判断するものとする。

・ リソースマネージャは、リソースファイルに基づく静的リソースだけではなく、任意のデータを扱えるものとする。
  - スレッド間で共有の必要な情報のほとんどを、リソースマネージャが備えるスレッドセーフ機構の恩恵を受けて安全かつ効率的に扱えるものとする。

・ シーンマネージャと連携し、一時的にリソースを削除し、また復元する機能を備えるものとする。
  - 例えば、ムービー再生のために大きな連続領域が必要となった場合、リソースの管理情報は残したまま、特定のメモリブロック上のリソースを全て削除し、ムービー再生が終わったあと、自動的に再度ロードして構築し直すといった処理に対応する。
  - また、例えば、メインメニュー（ゲーム中いつでも開ける）の中の一部のコンテンツに、多くのリソースを要するリッチなものがあった場合、リソースの管理情報は残したまま、任意の（敵・NPC などの）リソースを削除し、そのコンテンツを抜けた時に再構築するといった要件に対応する。

・ 上記のムービーの例のようなメモリブロックを扱う上で困らないように、同じリソースでも必ずしも共有せずに、別のメモリブロックに構築するような柔軟な制御も行えるものとする。

・ コンテンツ制作をランタイムでトライ＆エラーするために、ファイルマネージャと連携し、リソースのリロード（一旦破棄してロード・構築し直し）に対応するものとする。

▼ 要求仕様／要件定義

以下、本書が扱うシステムの要件を定義する。なお、要件として不確定の要求仕様も併記する。

・ ゲーム上で扱うほぼすべての「リソース」は、「リソースマネージャ」によって管理されるものとする。
  - 基本的に、ファイル読み込みを伴うデータやマルチスレッドで共有するデータはすべて「リソースマネージャ」が管理するものとする。
  - 一時的なテストやデバッグ用途に限り、十分に安全性を考慮した上で、リソースマネ

ージャを通さずにファイルを読み込んだりデータを共有したりしても良いものする。

・「リソースファイルのロード」「リソースの構築」「リソースの破棄」さらに「リソースへのアクセス」に対しては、必ずリソースマネージャを通さなければならない。

● システム間の依存関係

リソースマネージャとファイルマネージャは密接に連携するが、その依存関係は、リソースマネージャ⇒ファイルマネージャの一方向である。

また、リソースマネージャとシーンマネージャは密接に連携するが、その依存関係は、シーンマネージャ⇒リソースマネージャの一方向である。

- リソースマネージャは起動時にファイルマネージャにオブザーバー（コールバック）を登録する。オブザーバーにより、リソースマネージャが「自動リロードファイル」を読み込むと、コールバックが発生する。
- シーンオブジェクト（シーンマネージャがかなり）がリソースのハンドルを保有してリソースを使用する。
- ファイルマネージャについては、別紙の「開発を効率化するためのファイルシステム」を参照。
- シーンマネージャについては、別紙の「ゲーム全体を円滑に制御するためのシーン管理」を参照。

● リソースハンドルに関する要件

リソースは構築を要求された瞬間にユニークな「リソースハンドル」が発行される。（以降単に「ハンドル」と表記）

- リソースを要求した各処理系（主には「シーンオブジェクト」）は、発行されたハンドルを保持し、そのハンドルを使用してリソースにアクセスする。
  - リソースの構築状態（「ロード中」など）もハンドルを使用して確認できる。
  - リソースが不要になった時点で明示的にハンドルを解放する必要がある。（シーンオブジェクトのデストラクタを利用するなどして、確実な解放を行う。）
- 他の処理系がリソースを参照する必要がある場合は、ハンドルの受け渡しを行う。
  - 例えば、描画スレッドにはモデルやテクスチャなどのリソースハンドルを受け渡す。
  - リソースのポインタを受け渡すことは厳禁。

- 各処理系は、リソースのポインタを長期保持してはならず、一連の処理ごとにハンドルを通してリソースを参照しなければならない。
  - リソースはメモリ再配置が行われる可能性がある。
  - リソースは削除されている可能性がある。
  - 数フレームに渡るような長いバックグラウンド処理（例えば経路探索など）を行う場合も、時折リソースを参照し直すことで、他の処理による更新や削除の機会を与える。

- ハンドルを通したリソースアクセス時は、自動的にリソースをロックするため、アクセスの途中で再配置や削除が行われないことを保証する。
  - ロックについては、マルチスレッドの最適化要件として後述する。

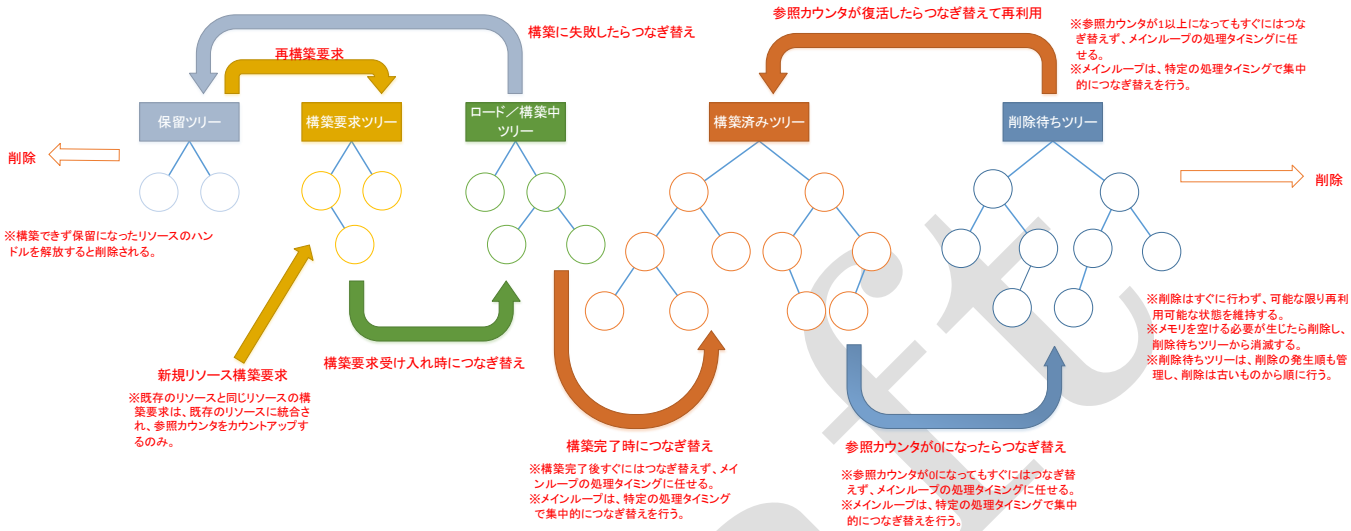
- ハンドルは 64bit の整数。

- ハンドルは、「任意の識別データ」（上位 24bit：主にシーン ID）＋「リソースファイルパスの CRC 値」（中位 32bit）＋「任意のインデックス」（下位 8bit）で構成する。
  - 純粋な静的リソース（ファイルから読み込んだリソース）は、上位 24bit が 0。
    - 下位 8bit も基本は 0 だが、別のメモリブロックに同じインスタンスを持つ場合があり、その際は 1 以上の値が扱われる。
    - ただし、複数のインスタンスがある場合であっても、同じ CRC の静的リソースは必ず同じ構造である。
  - 上位 24bit は、静的リソース以外の目的でリソースマネージャを利用する際に使用する。
  - 例えば、「3D モデルの姿勢データ」は、上位 24bit が「シーン ID」＋下位 8bit が「データ種別 ID」（シーンオブジェクトの中で「モデル」であることを示す）＋中位 32bit が「モデルファイルの CRC 値」といった構成となる。
  - また、例えば、「キャラクターの HP などの管理データ」は、上位 24bit が「シーン ID」＋下位 8bit が「データ種別 ID」（キャラクターの状態情報である事を示す）＋中位 32bit が 0 といった構成になる。

- 管理されるリソースの数だけリソースハンドルが発行され、それぞれリソース管理情報を持つ。
  - リソース管理情報は、リソースの構築状態や対象とするメモリなどの情報を扱う。
  - リソース管理情報は、検索効率のために、ハンドルをキーにした平衡木で管理する。（赤黒木を想定）
  - リソース管理情報のツリーは、幾つかの種類（先頭ノードが異なる）に分割して扱う。
    - ツリーの種類は、「構築要求ツリー」「ロード／構築中ツリー」「構築済みツリー」「削除待ちツリー（再利用のためのキャッシュ扱い）」「保留ツリー」。
    - ツリー内のノードの連結操作が発生すると、その間ツリーがロックされる。それにより、

他のスレッドからのリソース検索処理がブロックされてしまう。ツリーが分割管理されることにより、そのブロックの機会を減らす。

リソース管理ツリーのイメージ：



● リソース要求とリソースの共有に関する要件

各処理系（主に「シーンオブジェクト」）から「リソース要求」された際、リソースマネージャは、リソースの構築状態に応じた処理を行う。

- リソースがどのツリーにも存在しない場合、新たにリソース管理情報を作り、「構築要求ツリー」に連結する。
- リソースがすでに存在する場合、そのリソースを共有する。
  - リソースの参照カウンタをカウントアップする。
  - 構築済みのリソースの場合、リソースを要求した直後からリソースにアクセス（参照）可能。
- 要求されたリソースが「削除済みリソース」出会った場合、リソースを共有した上、「削除済みツリー」から「構築済みツリー」につなぎ替える。
- 「構築要求ツリー」のリソース（要求）は、処理可能な状態になり次第、「ロード／構築中ツリー」につなぎ替え、ファイルロード～構築処理を順次行う。
- 「リソースファイルが存在しない」などの理由で、リソース構築が不可能なものは、ロード／構築処理に失敗し、「保留ツリー」につなぎ替える。
  - 「保留ツリー」は、本来メモリ不足などの理由で構築できなかったリソースの構築をリトライするためのツリーだが、完全に構築不可能な場合も同様に扱い、構築の結果を、（構築を要求した処理が）知ることができるようにする。

- ハンドルが解放された時にこのツリーのリソース（要求）が破棄される。
- 「保留ツリー」上のリソース（要求）も、誤った要求が続発する限りは参照カウンタをカウントアップして共有する。

➢ 構築が終わったリソースは、「構築済みツリー」につなぎ替え、リソースにアクセス可能な状態になる。

➢ 他のスレッドからアクセス（参照）可能なリソースは基本的に「構築済みツリー」に連結されたもののみ。ここから見つからなければリソースアクセスは失敗となる。

➢ 各処理系がハンドルを開放し、参照がなくなったリソースは「削除待ちツリー」に連結し直す。

➢ 「構築済みリソース」のリソース本体のみが削除されて、「ロード／構築中ツリー」に戻されるケースもある。（後述する）

➢ 「構築済みツリー」への連結、および、「構築済みツリー」からの連結の削除が発生すると、そのツリー操作が行われている間、「構築済みツリー」へのアクセスがブロックされる。そのような操作が頻繁に発生すると全体のパフォーマンスを損ねるので、ツリーの連結操作は、ゲームループ中の特定のタイミングで集中的に行うようにする。

- 「ロード／構築中ツリー」のリソースは、構築が終わってもすぐには「構築済みツリー」に連結せず、集中処理のタイミングで組み替える。
- 「削除待ちツリー」にあるリソースが要求された際、すぐには「構築済みツリー」に連結せず、集中処理のタイミングで組み替える。その間、参照カウンタがカウントアップしているので、実際の削除対象にはならない。
- 参照を失ったリソースも同様に、集中処理のタイミングまでは「削除済みツリー」への移し替えを行わない。その間、参照カウンタが 0 のため、「構築済みツリー」に連結された状態でも、アクセス不可として扱う。
- アクセス中のリソースはロックによって保護されるため、アクセスの途中で削除されることはないが、ツリーの組み替えは行われる可能性がある。
  - ・ リソースアクセス中は、検索が済んでいるのでツリーが変わっても問題がない。
  - ・ リソースアクセス中は参照カウンタを更新せずにロックのみで保護するため、「参照カウンタに基づくツリーの組み替え」は行われる。

➢ このような連結処理のタイミング調整があるため、「削除済みリソース」が再利用される際は、構築済みリソースであるにも関わらず、すぐに利用できない。

- ツリーの連結し直しのために 1 フレーム待たされることになる。

➢ 「構築要求ツリー」から「ロード／構築中ツリー」への組み替えは、「リソース構築管

理スレッド」が「構築要求ツリー」の状態を監視して、随時行う。

- 「構築要求ツリー」のリソースの参照が失われた場合、その時点でノード（リソース管理情報）を破棄する。
  - 「削除待ちツリー」には連結されない。

- 「ロード／構築中ツリー」のリソースの参照が失われた場合、ファイル読み込みをキャンセルして、ノードを破棄する。
  - 「削除待ちツリー」には連結されない。
  - 構築が済んでいて「構築済みツリー」への連結待ちの状態だったものは、「削除待ちツリー」に連結される。

- リソース共有の効率を向上させるために、アーカイブファイルのリソースを構築する際は、アーカイブ内のファイル一つ一つを別々のリソースとして管理する。
  - 例えば、モデルとテクスチャをアーカイブしている場合、リソース管理上はそれぞれを別リソースに分けて管理する。
  - 読み込み効率をよくするために、同じリソースファイルが複数のアーカイブファイルに分散している場合があることと、整然としたメモリ管理の観点から、単純なリソース管理を徹底する。
  - ファイルマネージャからアーカイブファイル内のファイルリストがリソースマネージャに返されると、各ファイルのリソースが要求（構築）される。
  - アーカイブファイルと、アーカイブ内の各ファイルは、依存関係が成立する。
    - アーカイブが「親リソース」で、アーカイブ内の各ファイルが「子リソース」という関係が成立する。しかし、依存関係は親リソースが子リソースを参照する方向である。
    - 親リソースが子リソースを参照して参照カウンタをカウントアップする。
    - アーカイブファイルを要求した各処理系は、親リソース（アーカイブファイル）のハンドル一つを保持しておけば、子リソースが削除されることはなく、親リソースのハンドルを開放すれば、子リソースも解放できる。
  - 親リソースのハンドルから子リソースのハンドルを取得できる。
    - 例えば、描画スレッドに投入する情報としては、アーカイブのハンドルではなく、そこから取り出したテクスチャのハンドルを渡す必要がある。

● リソース構築に関する要件

タイトル固有のリソース構築処理を使用可能。

- リソースを構築するための処理クラス（「リソースビルダー」と呼ぶ）を用意し、「リソース種別」と関連付けてリソースマネージャに登録する。

- 「リソースビルダー」は任意の処理が可能。
  - ファイルを読み込んで構築するものに限定しない。

- 例えば、敵キャラの HP や STR を管理する「CEnemyStatus」というクラスのインスタンをリソースとして扱うことも可能。
  - こうしたデータ管理も、マルチスレッド処理が必要ならリソースマネージャを通してメモリの安全性を確保する。この例の場合、AI 処理をマルチスレッド化した場合に安全にアクセスできる。
  - リソースビルダーはデフォルトコンストラクタによる初期化しか行わないため、その後キャラ固有の情報をセットしたりするのは、「構築完了後」の処理となる。

- ファイルをそのままコピーして済むものは、リソースビルダーの処理を必要としない。
  - リソースビルダーの登録時に「ファイルコピーのみ」と指定すれば、処理を記述せずとも、リソースマネージャがコピー処理を行う。

- アーカイブファイルのリソースビルダーでは、アーカイブ内ファイルのリストがファイルバッファに格納されて渡されるので、それに基づいてそれぞれの構築要求を出す。

- リソース要求時は、「リソースファイル」（指定不要な場合もある）、「リソースビルダー」、「構築先のメモリ」（オプション）を指定する。
  - 「リソースビルダー」の属性設定で固定される構築要件もある。
    - 例えば、「コピーのみで完結するか？」や「メモリ再配置可能か？」といった要素はリソースビルダーで指定される。
    - 「構築先のメモリ」もデフォルトはリソースビルダーによって指定されるが、任意に変更することができる。

- リソースファイルのロードはファイルマネージャによってバックグラウンドで行われる。

- リソースの構築は「ジョブスケジューラ」を使用して、マルチスレッドで並行処理する。
  - 「ジョブスケジューラ」については、別紙の「「サービス」によるマルチスレッドの効率化」を参照。
  - 「リソースビルダー」は、スレッド（ジョブ）のローカル変数としてインスタンス化して使用する。（このプログラミング手法は後述する）

- リソースを破棄する際もリソースビルダーの処理が実行される。
  - ファイルをそのままコピーして済むものは、リソースビルダーを使用せずに処理する。
  - リソースビルダーにより、適切なデストラクタ呼び出しを行う。

- リソースの破棄もジョブとして実行される。

● メモリ管理に関する要件

リソースに必要なメモリは、リソースマネージャがリソース用に管理するメモリマネージャを使用して確保する。

- リソースマネージャは、複数のメモリブロックを管理する。
  - 巨大なヒープ一つでリメモリ管理するのではなく、複数のメモリブロックを用いることで、ゲームを制御し易くする。
- 例えば、「再配置可能なリソース」と「再配置不可能なリソース」はメモリブロックを分けておいた方が効率的である。
  - 混在は可能だが、それだと再配置が効率よく行われない。
- また、例えば、「グラフィックリソースブロック A（削除が許されない）」と「グラフィックリソースブロック B（削除してもよい）」に分けてメモリブロックを管理することで、「ムービー再生のための大きな連続領域が必要」といった要求に応じてリソースの一括削除を行う。
  - この時、管理情報は削除せず、「サスペンド状態」にする。
  - 「サスペンド状態」とは、管理情報も参照カウンタもある状態だが、リソースの実体が無い状態。「ロード中」状態ではあるが、実際にロードの実行が禁止されている。
  - リソース管理情報は「構築要求ツリー」から「ロード／構築中ツリー」へ組み替えられる。
  - ムービーが終了して「サスペンド状態」が解除されると、自動的に再ロードし、構築が行われる。
- リソース管理情報のためのメモリは、管理数の上限を決めて固定数の情報をあらかじめ確保しておく。
- リソースに必要なメモリの確保は、リソースビルダー内の処理で任意に行う。
  - ファイルをそのままコピーして済むものはリソースマネージャが行う。
  - メモリ確保の際に、再配置可能なメモリとして確保するか、不可能なメモリとして確保するか指定する。（メモリマネージャの機能）
- リソース構築開始時に、最終的なメモリサイズが不明なリソースは、メモリマネージャの機能により、一旦大きめにメモリを確保した後に、サイズを小さくすることができる。
  - 例えば、一旦ファイルサイズの 1.5 倍のメモリを確保して、最終的に 1.2 倍で済んだなら、メモリマネージャに通知してメモリサイズを変更する。
  - 最後に大きくすることはできない。どうしても必要なら自前でメモリを確保し直すなどする。

- リソース管理情報では、2 つまでメモリ情報を管理する。
  - メインメモリ＋GPU メモリのように、二つ一組で一つのリソースを構成するようなものを扱える。
  - リソースの共有はリソース単位で行うのみで、メモリ情報ごとの共有には対応しない。
  - それ以上のメモリ確保も可能だが、リソースマネージャが直接管理できないので、リソースビルダーを通した処理で、確実な破棄を行う必要がある。
  - リソースごとのリソース管理情報は、確保中のメモリサイズを保持するが、必ずしも正確である必要はない。優先度の高いリソースにメモリを譲る場合などに参考にされるサイズである。複雑なメモリ管理をしているリソースは正確な集計が難しい場合がある。

- リソースは「メモリ再配置」（コンパクション）の機能により、可能な限り連続領域を大きく確保する。
  - メモリ再配置は、メモリマネージャの機能で行う。
  - メモリ再配置は、リソースマネージャがメモリマネージャに対して明示的に実行する。
    - メモリマネージャが自動的に実行することはない。
    - メインループ中のある一点でのみ、リソース全体をブロックした上で（リソースツリーをライトロックした状態で）、集中的に処理する。
    - 再配置は完全に実行せず、そのフレームで許された時間の分だけ進める。
  - メモリマネージャが管理するメモリは、メモリノードごとに再配置の可／不可が設定され、あとから変更することもできる。再配置不可設定のリソースや、構築・破棄中のリソースは、確保したメモリに対して再配置不可設定を行い、構築完了後に再配置を許可するなどする。これはリソースマネージャが適切に処理しなければならない。

- リソース構築は「メモリがある限り」という条件で構築を試みる場合がある。つまり、「リソース要求」は常に「成功必須」ということではない。リソースマネージャは、このような要件に柔軟に対応する。
  - 例えば「メモリが許す限り NPC を出現させる、なければ出なくていい」といった処理要件に対応する。
  - 通常の処理では、リソース要求に対してメモリが足りない場合、メモリが確保できるまでリソース構築をリトライし続ける。リソースマネージャは、最適なリトライを行えるようにサポートする。
    - リソースマネージャは、この「メモリ不足」が発生した時に、「削除待ちリソースの削除」や「メモリ再配置」を積極的に行い、メモリを空けようとする。
    - まず、「削除待ちリソース」が残っている限りは、メモリ確保要求の中で要求されるメモリが得られるまで削除を行う。
    - その上でメモリ確保に失敗した場合は、一旦リソース構築処理を失敗終了させなければな

らない。

- ・ リソースビルダーのリソース構築処理では、「そこまでに確保したメモリを全て破棄した上で」構築処理を失敗終了させなければならない。
- ・ 途中までの確保を捨てずにリトライしたい場合は、リソースビルダーの処理の中で、独自のループ・リトライ処理を記述する必要がある。ただし、ジョブのライフサイクルが長くなるので、推奨されない方法。
- ・ 失敗終了の際に、リトライ要求するか諦めるかを指定する。
- ・ リトライの前に、どのメモリに対して、どれだけのメモリが必要だったかをリソースマネージャに通知しておく。
- ・ リトライ要求で失敗終了すると一旦ジョブ（スレッド）は終了し、次回の再配置／削除処理後に再度実行される。
- ・ 「メモリ再配置」はゲームループ中の特定のタイミングのみで行うので、再配置に頼る場合はそれまで待つことになる。
- ・ リトライ処理は、リソースが必要とするメモリの再配置・削除が発生しない限り、リトライが起こらない。
- ・ リトライ発生中は、同じメモリを使用するリソースで、「優先度」が低いものは、構築が待たされる。
- ・ リトライ発生時は、「優先度」が低く「一時削除が許可」されたリソースの中から、要求されたメモリサイズに近いものから順に削除し、「ロード／構築中ツリー」に戻す。
- ・ 一定時間リトライに失敗したリソースは、「保留ツリー」に回し、リトライをやめる。
- ・ このタイムアウト時間は、リソース構築時に任意の時間を指定できる。
- ・ リソース構築を要求した処理系は、リソースが「保留」になったことでタイムアウトを知ることができる。
- ・ 「保留」になったリソースは、再度構築を要求することができる。
- ・ 「保留」リソースのハンドルが解放された場合、「削除待ちツリー」にも回さず、リソース管理情報を破棄する。

➢ メモリ制限はメモリブロックだけではなく、「メモリカテゴリ」によってさらに細かく行うこともできる。

- リソース用のメモリブロックはできるだけ分けないほうが効率的にメモリを使用できるが、カテゴリごとにメモリの制限は行いたい、という時に使用する。
- 例えば、「メニューカテゴリ」「NPC カテゴリ」などの任意のカテゴリを扱い、リミットサイズを設定することができる。
- 空きメモリに余裕があっても、カテゴリのリミットを超える場合はメモリ確保に失敗する。
- このリミットの機能は、リソースマネージャではなく、メモリマネージャの機能として提供する。（リソースマネージャは正確に使用メモリサイズを管理しきれない）

- メモリマネージャの機能により、カテゴリごとのメモリ使用量が 90%を超えると警告を表示する。
  - ・ この警告のしきい値はカテゴリごとに設定可能。
  - ・ 警告は、しきい値を超えた瞬間にだけ表示する。
- このメモリ制限は、実行中にいつでも変更できる。
- メモリ制限を縮小した時、すでにそれ以上のメモリが確保されている可能性がある。その場合、強制的にメモリを破棄するようなことはなく、100%越の警告が一度表示される。

● 特殊なリソース構築に関する要件

リソース要求時に、同じリソースが存在していても、それを共有せずに、指定メモリへの構築を強制することができる。

- リソース構築時にメモリブロックを指定するが、メモリブロックが違うものでも可能な限り共有するのがデフォルトの挙動。
- 例えば、シーンを切り替える際、特定のメモリブロックを強制的に空けなければならない時、そのメモリブロックに含まれるメインキャラを、次のシーンのために別のメモリブロックに構築し直したいといった要件に対応する。
- 既存のリソースと同じリソースを別のメモリブロックに構築する場合は、可能な限りリソースからリソースへのコピーで済ませる。
  - ファイル読み込みは行わない。
  - リソースビルダーを必要としないリソースは、リソースマネージャがそのままコピーする。
  - リソースビルダーを必要とするリソースは、リソースビルダーのコピー処理を使用する。
  - リソースビルダーのコピー処理は、任意に作成する必要がある。
  - リソースビルダーのコピー処理が「コピー不可」を返した場合、通常の構築と同じ手順で、ファイル読み込みから構築を行う。

● マルチスレッドの最適化に関する要件

マルチスレッドに最適化するために、リソース一つ一つが「リード・ライトロック」を持つ。

- 読み込み操作は複数のスレッドが同時に処理可能だが、書き込み時は全てのスレッドがブロックされる。詳しくは別紙の「効率化と安全性のためのロック制御」を参照。

- マルチスレッドに最適化するために、リソース管理情報ツリーの組み替え操作、および、メモリ再配置は、ゲームループ中のある一点でのみ、集中的に行う。
  - ゲーム中に「構築済みツリー」の組み替えが発生すると、たびたびロックが発生してしまうので、その処理は集中的に行う。
  - 「構築済みツリー」が関係しないツリーの組み替えは、マルチスレッドで随時行う。
  - 「メモリ再配置」についても同様に、たびたびロックを発生させてしまうので、同じタイミングで集中的に行う。
  - 実行のタイミングは、フレームの処理の最後。そのフレームでのリソース構築・削除の要求が全て集まったあと、描画スレッドを起動して大量のリソースアクセスが発生するまでの間に行う。

- 「構築要求ツリー」を監視して構築処理を開始する処理は、マルチスレッドによるバックグラウンドで行う。
  - バックグラウンドでリソース操作の監視と実行を行う専用スレッド「リソース操作スレッド」が常に稼働している。

- リソース操作スレッドの優先度は、メインスレッドより低く、ファイルマネージャより高く設定する。

- リソース操作スレッドは、以下の操作を行う。
  - 「構築要求ツリー」の状態を監視し、リソース要求があったら「ロード／構築中ツリー」につなぎ替える。
  - 「ロード／構築中ツリー」の状態を監視し、ファイル読み込みが必要なものはファイルマネージャに読み込み要求を出す。
  - 「ロード／構築中ツリー」の状態を監視し、ファイル読み込みが完了したもの、もしくは、ファイル読み込みが不要なものは、リソース構築を行うために、「ジョブスケジューラ」に「リソース構築ジョブ」（スレッド）を投入する。（コピー構築の場合も含む）
  - ファイルの内容をそのままコピーして済むリソースの場合、ファイルマネージャから取得したファイルサイズ分のメモリを確保し、そのバッファをファイルマネージャに渡す。（コピーが完了するまで再配置が行われないように注意）
  - 「ロード／構築中ツリー」から「構築済みツリー」へのつなぎ替えはしない。これはメインスレッドの処理。
  - リソース削除の要求を受けて、「削除待ちツリー」のリソース削除を行うために、ジョブスケジューラに「リソース削除ジョブ」（スレッド）を投入する。
    - 要求されたメモリブロックとメモリサイズに基づいて、要求に見合ったリソースを選んで削除する。
    - 「削除待ちツリー」に連結されていても、新たな参照によって再利用されるリソースもあ

る。そのようなリソースは、参照カウンタが 0 より大きい値になっているので、削除の対象にはしない。

- 全般的に、リソース操作スレッドは瞬間的に終わるような処理しか行わず、何らかの処理要求に対してすぐに反応できる状態にしておく。
- ジョブ（スレッド）の完了はとくに監視しないため、ジョブ完了後のリソースの状態更新は、そのジョブ自身の処理で行う。
- 「ジョブスケジューラ」に関する説明は、別紙の「「サービス」によるマルチスレッドの効率化」を参照。

● 一時的なメモリ削除と復元

前述の「メモリ管理に関する要件」に示したように、構築済みのリソースが、リソースの存在を維持したまま、リソース本体を削除することがある。

- 一時的にメモリを空けて別シーンに切り替えたあと、また元のシーンに元の状態で復帰したい場合に使用する。
- リソースの一時削除が指定された場合、リソース管理情報は消さずにリソース本体だけ削除し、リソースは「構築要求ツリー」につなぎ替える。この時、リソースをさらに「サスペンド状態」に設定し、明示的に「再開」が呼び出されるまで構築を開始しない。

● リソースの自動リロード

ファイルマネージャと連携したリソースの自動リロードに対応し、ランタイムでリソースファイルを読み直してトライ＆エラーを行うことを可能とする。

- ファイルマネージャに「自動リロード用アーカイブ」が読み込まれると、ファイルマネージャに登録したオブザーバーのコールバックにより、アーカイブに含まれるファイルのリストを得ることができる。
- このコールバックを受けて、リソース管理情報と照合し、該当するリソースが存在したならリソース本体だけを破棄して「構築要求ツリーにつなぐ」。
- これにより、「自動リロード用アーカイブ」のファイルを使用して、リソースが再構築される。
- この処理はデバッグ用の処理なので、処理全体をブロックして時間がかかっても良い。
  - 処理をブロックして対象のリソースを強制的に「削除候補ツリー」に連結し、「強制削除」状

態にする。

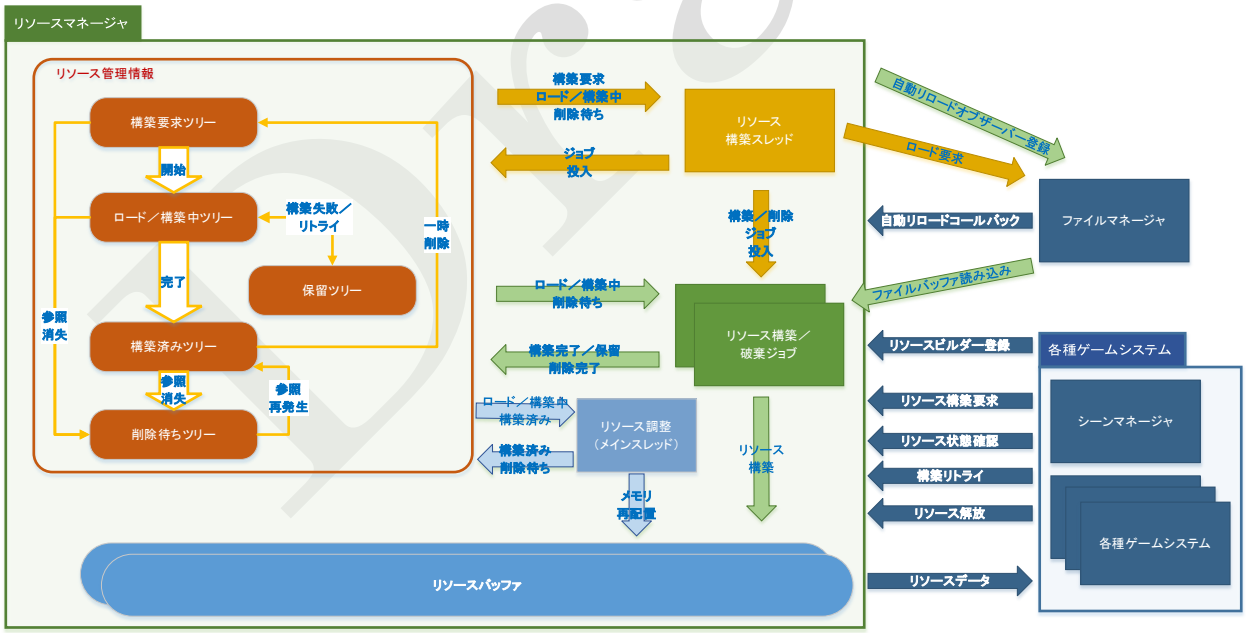
- 「リソース操作スレッド」は、「削除候補ツリー」から「強制削除」状態のリソースをピックアップして順次ジョブ投入する。
- 「強制削除」状態のリソースは、新たなリソース要求で参照カウンタが変動しても、「削除候補ツリー」に残る。
- 削除ジョブが完了した後は、「強制削除」状態を解除して、通常処理に任せておけばよい。参照カウンタが残っているので、「構築要求ツリー」につなぎ替えられて、構築処理が開始される。

■ 仕様概要

▼ システム構成図

要件に基づくシステム構成図を示す。

リソースマネージャのシステム構成図：



■ プログラミングモデル

リソース構築要求、リソースアクセスの処理仕様として、具体的なプログラミングモデ

ルを示す。

リソースマネージャ自体の処理は、多数の要件を挙げたとおり、複雑なものとなるが、反面、リソースを利用する側は簡単に扱えるものとする。

手軽にリソースを扱えるようにしながらも、安全性とパフォーマンスを損ねないプログラミングモデルを確立する。

▼ リソース構築処理：アブストラクトファクトリーパターンの実践

タイトル固有のリソース構築処理を扱うためのプログラミングモデルを確立する。

要は、汎用の共通処理であるリソースマネージャに、タイトル固有の処理を実行させる仕組みを作成する。

● プログラミングのイメージ

独自のリソース構築用クラスを作成し、リソースの構築要求時に、そのクラスをリソースファイルパスと共に受け渡すことが可能な構造にする。

まず、そのようなプログラミングのイメージを示す。

リソース構築処理のプログラミングイメージ：

【リソース構築処理】

```
//モデルリソースの構築要求情報を作成
CResBuildReq<CModelResourceBuilder> req("/data/chara/x0010.mdl");
//※CModelResourceBuilder クラスは、タイトル固有のリソース構築処理クラス（以下にイメージを示す）
//※CResBuildReq テンプレートクラスは、汎用のリソース構築要求クラス

//リソース構築要求
CSingletonUsing<CResManager> res_man("test");//リソースマネージャシングルトンを取得
HRES handle = res_man->request(req);//リソースマネージャにリソース構築要求を出す
```

【リソース構築用クラス】

```
//タイトル固有のリソース構築クラス：モデル用
class CModelResourceBuilder : public IResBuider
{
	//…（略）…
public:
	//リソース構築
	E_BUILD_RESULT create(const CResBuildReqInfo& req_info) override
	{
		//…（処理）…
		return BUILD_SUCCESS;
	}
	//リソース破棄
	E_BUILD_RESULT destroy(void* res) override
	{
		//…（処理）…
		return BUILD_SUCCESS;
	}
	//…（略）…
};
```

赤字で示したのが、リソース構築用に用意する独自のリソース構築用クラス。

● アブストラクトファクトリーパターン

リソース構築要求を出した後、実際のリソース構築はそのずっと後になる。

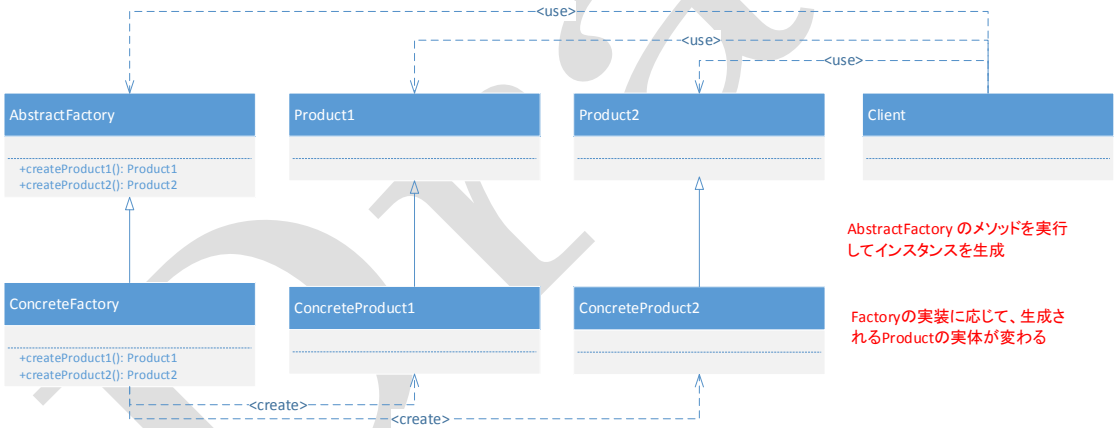
リソースマネージャは最初にファイルパスに基づいてファイル読み込みを行い、読み込みが完了した後に、構築処理をジョブスケジューラに投入する。実際にリソース構築処理が実行されるのは、このジョブ（スレッド）が実行された時である。

この対応として、デザインパターンの「アブストラクトファクトリーパターン」を活用する。

「アブストラクトファクトリーパターン」は、「オブジェクトを生成するためのオブジェクト」を扱うパターンである。（オブジェクトを生成するオブジェクトを「ファクトリー」と呼ぶ）

このパターンを活用し、リソース構築処理オブジェクトを生成するための「ファクトリー」を、リソース構築処理の識別 ID と共にリソースマネージャに登録する方法をとる。

一般的なアブストラクトファクトリーパターン：



● シングルトンパターン

リソース構築処理はマルチスレッドで並行実行するため、複数のインスタンスが同時に存在する可能性があるが、ファクトリーのインスタンスは一つだけで十分なので、static 変数で扱う。

プログラマーの手間を少しでも減らすように、テンプレートクラスを活用し、「リソース構築クラス」だけ用意すれば、「ファクトリー」クラスを自動的に作って static 変数で扱うようにする。

- この「static 変数で扱う」という処理は、「シングルトンパターン」の応用である。「シングルトンパターン」については、別紙の「効率化と安全性のためのロック制御」で詳

しく説明している。

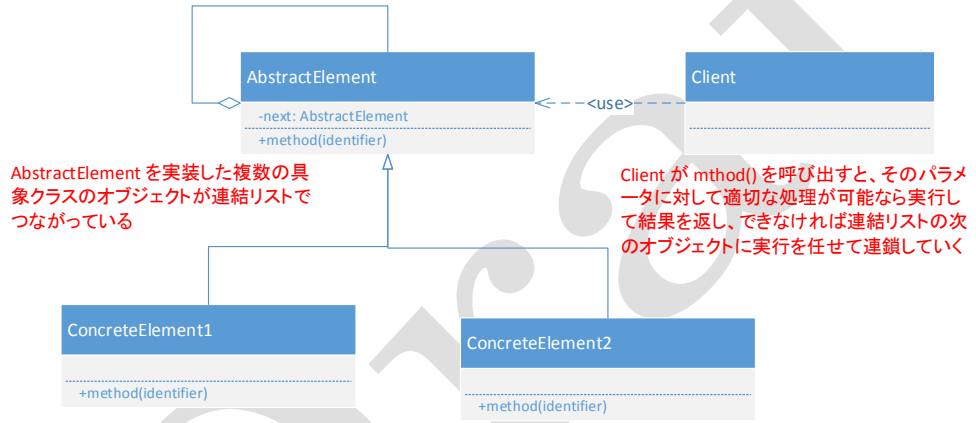
● チェインオブレスポンシビリティパターン

さらに、そのテンプレートクラスは、ファクトリークラスのオブジェクト（static 変数）を連結リストにする。

それにより、リソース構築が要求された際に、その連結リストを辿って適切なファクトリーを探し、見つかった時点でリソース構築処理オブジェクトを生成して返す。

- この「適切なファクトリーを探し、見つかった時点で…」という処理は、「チェインオブレスポンシビリティパターン」の応用である。

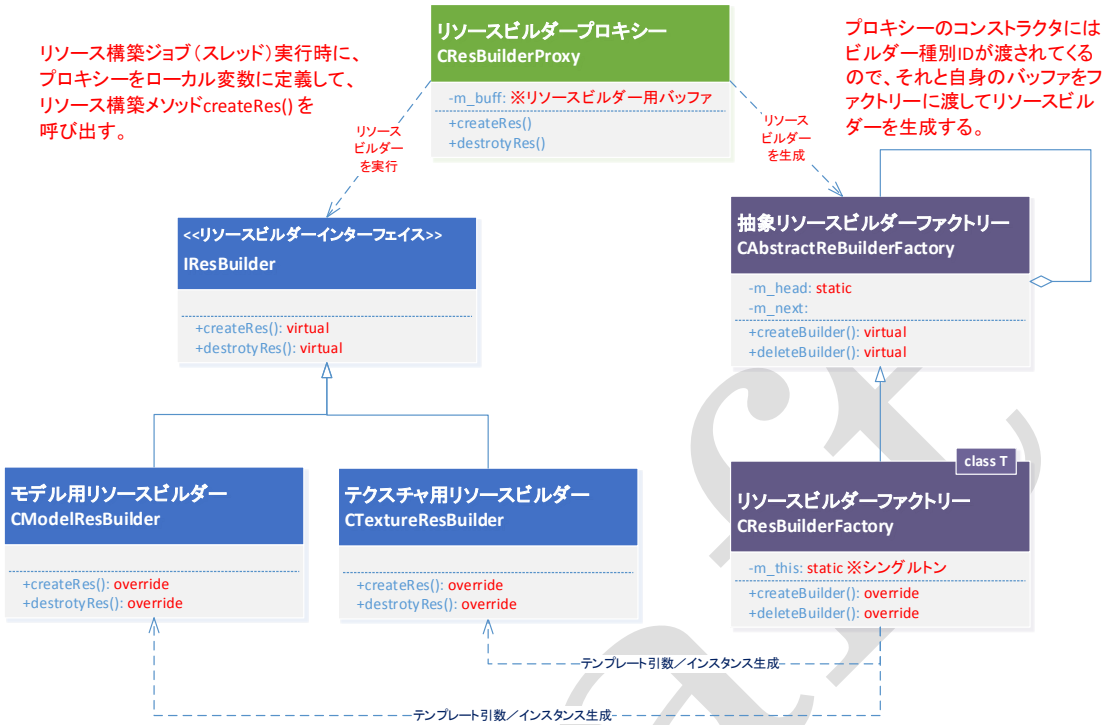
一般的なチェインオブレスポンシビリティパターン：



● リソースビルダーの設計

以上に基づいて、リソース構築処理のための、リソースビルダークラスを設計する。

リソースビルダークラス：



● リソースビルダーのサンプル

以上の設計に基づいて、リソースビルダーを通してリソー構築を行うサンプルプログラムを示す。上述のデザインパターンに関係したポイントとなる処理は赤字で示す。

リソースビルダーのサンプル：

【インクルード】

```
//-----------------------------------------------------------------------------
//リソースビルダー処理テスト
#include <stdio.h>
#include <stdlib.h>
#include <atomic>
```

【サンプル用ダミー：共通型宣言と共通処理】

```
//-----------------------------------------------------------------------------
//【サンプル用ダミー】共通型宣言と共通関数

//CRC32 型
typedef unsigned int CRC32;

//CRC 算出関数
CRC32 CRC(const char* str)
{
    return 123;//ダミー
}
```

【サンプル用ダミー：シングルトン】

```
//-----------------------------------------------------------------------------
//【サンプル用ダミー】シングルトンクラス

//シングルトンアクセスクラス（ダミー）
```

```
template<class T>
class CSingletonUsing
{
public:
	//オペレータ
	T* operator->(){ return &m_instance; }//アロー演算子
public:
	//コンストラクタ
	CSingletonUsing(const char* name)
	{}
private:
	//フィールド
	static T m_instance;//シングルトンインスタンス
};

//シングルトン static インスタンス生成用マクロ
#define SINGLETON_INSTANCE(T) ¥
	T CSingletonUsing<T>::m_instance;
```

【リソースマネージャ共通】

```
//-----------------------------------------------------------------------------
//リソースマネージャ共通

//---------------------------------------
//定数

//リソース属性
enum E_RES_ATTR
{
	RES_ATTR_MOVABLE = 0x00000000,//再配置可能
	RES_ATTR_FIXED = 0x00000001,//再配置不可
};

//---------------------------------------
//型

//リソースハンドル
struct HRES
{
	union
	{
		struct
		{
			unsigned int m_id;//任意の識別 ID
			CRC32 m_pathCRC;//ファイルパス CRC
		};
		unsigned long long m_key;//64bit キー
	};
	//コンストラクタ
	HRES(unsigned int id, CRC32 path_crc) :
		m_id(id),
		m_pathCRC(path_crc)
	{}
	HRES(unsigned long long key) :
		m_key(key)
	{}
	//デフォルトコンストラクタ
	HRES()
	{}
};
```

【リソースビルダー】

```
//-----------------------------------------------------------------------------
//リソースビルダー

//---------------------------------------
//定数
```

```
//リソースビルド属性
enum E_RES_BUILD_ATTR
{
	RES_TO_COPY_ONLY = 0x00000001,//コピーのみ
	RES_TO_BUILD = 0x00000000,//構築処理必要

	RES_REQUIRED_FILE = 0x00000002,//リソースにはファイルが必要
	RES_NOT_REQUIRED_FILE = 0x00000000,//リソースにはファイルが不要
};

//ビルド結果
enum E_BUILD_RESULT
{
	BUILD_IMPOSSIBLE = 0,//ビルド・コピー不可
	BUILD_SUCCESS = 1,//ビルド成功
	BUILD_RETRY = -1,//ビルド失敗：リトライ要求
	BUILD_GIVEUP = -2,//ビルド失敗：諦める
};

//---------------------------------------
//リソース構築要求情報クラス
//※リソースファイルを指定して構築を要求するためのクラス
//※リソースビルダー生成のためのビルダー種別 ID を保持する
//※構築先のメモリの指定も可能（通常はリソースビルダーによって規定される）
//※本来はリソースファイルを伴わないメモリ上のデータや、
//  リソースの依存関係、リソース優先度、構築リトライのタイムアウト、
//  メモリカテゴリといった情報も扱うが、このサンプルではそこまで対応しない
class CResBuildReqInfo
{
public:
	//定数
	static const int MEM_INFO_MAX = 2;//メモリ情報の最大数
public:
	//データ型
	//メモリ情報型
	struct T_MEM_INFO
	{
		int m_memId;//メモリ ID
		bool m_isDenyShare;//共有禁止（メモリ ID が違う同名リソースの禁止）
		bool m_isLocked;//ロック（破棄禁止）
		void* m_loadedBuff;//読み込み後のバッファ
		//コンストラクタ
		T_MEM_INFO(const int mem_id, const bool is_deny_share = true, bool is_locked = false) :
			m_memId(mem_id),
			m_isDenyShare(is_deny_share),
			m_isLocked(is_locked),
			m_loadedBuff(nullptr)
		{}
		//デフォルトコンストラクタ
		T_MEM_INFO(){}
	};
public:
	//アクセッサ
	int getResTypeId() const { return m_resTypeId; }//リソース種別 ID 取得
	int getBuilderTypeId() const { return m_builderTypeId; }//リソースビルダー種別 ID 取得
public:
	//メソッド
	//メモリ情報追加
	bool addMemInfo(const T_MEM_INFO&& info)
	{
		if (m_memInfoNum >= MEM_INFO_MAX)
			return false;
		m_memInfo[m_memInfoNum++] = info;
		return true;
```

```
	}
	//メモリ情報取得
	int getMemInfoNum() const { return m_memInfoNum; }
	T_MEM_INFO* getMemInfo(const int index)
	{
		if (index < 0 || index >= m_memInfoNum)
			return nullptr;
		return &m_memInfo[index];
	}
	//メモリ情報取得
	const T_MEM_INFO* getMemInfo(const int index) const
	{
		if (index < 0 || index >= m_memInfoNum)
			return nullptr;
		return &m_memInfo[index];
	}
	//メモリ情報更新
	T_MEM_INFO* setMemInfo(const int index, const T_MEM_INFO&& info)
	{
		if (index < 0 || index >= m_memInfoNum)
			return nullptr;
		m_memInfo[index] = info;
		return &m_memInfo[index];
	}
	//一般的なファイルリソースハンドル作成
	HRES makeNormalHandle() const
	{
		HRES handle(0, m_pathCRC);
		return handle;
	}
public:
	//コンストラクタ
	//※構築するリソースのリソース種別 ID、リソースビルダー種別 ID とリソースのパスを渡す
	CResBuildReqInfo(const int res_type_id, const int builder_type_id, const char* path) :
		m_resTypeId(res_type_id),
		m_builderTypeId(builder_type_id),
		m_pathCRC(CRC(path)),
		m_memInfoNum(0)
	{
	}
	//デストラクタ
	~CResBuildReqInfo()
	{
	}
private:
	//フィールド
	int m_resTypeId;//リソース種別 ID
	int m_builderTypeId;//リソースビルダー種別 ID
	CRC32 m_pathCRC;//リソースパス (CRC)
	T_MEM_INFO m_memInfo[MEM_INFO_MAX];//リソース構築用のメモリ情報
	int m_memInfoNum;
};

//-------------------------------------
//リソース構築要求クラス
template<class T>
class CResBuildReq : public CResBuildReqInfo
{
public:
	//メソッド
	//ハンドル発行
	HRES makeHandle() const
	{
		T builder;
		return builder.makeHandle(*this);
```

```
	}
public:
	//コンストラクタ
	//※構築するリソースのパスを渡す
	CResBuildReq(const char* path) :
		CResBuildReqInfo(T::RES_TYPE_ID, T::BUILDER_TYPE_ID, path)
	{
		//リソース構築要求情報のデフォルト設定
		T::setDefaultMemInfo(*this);

		//リソースビルダーファクトリーシングルトン生成
		CResBuilderFactory<T>::createSingleton();
	}
	//デストラクタ
	~CResBuildReq()
	{
	}
};

//---------------------------------------
//リソースビルダーインターフェースクラス
class IResBuider
{
public:
	//メソッド
	//ハンドル作成
	virtual HRES makeHandle(const CResBuildReqInfo& req_info) = 0;
	//リソース構築
	virtual E_BUILD_RESULT create(const CResBuildReqInfo& req_info) = 0;
	//リソースコピー
	virtual E_BUILD_RESULT copy(const CResBuildReqInfo& req_info, const void* src_res) = 0;
	//リソース破棄
	virtual E_BUILD_RESULT destroy(void* res) = 0;
public:
	//デストラクタ
	virtual ~IResBuider()
	{}
};

//-----------------------------------
//リソースビルダーファクトリー抽象クラス
class CAbstractResBuilderFactory
{
public:
	//アクセッサ
	virtual int getBuilderTypeID() const = 0;//リソースビルダー種別 ID 取得
	CAbstractResBuilderFactory* getNext() const { return m_next; }//次のリソースビルダーファクトリー取得
public:
	//メソッド
	//リソースビルダー構築
	virtual IResBuider* createBuilder(const int builder_type_id, char* buff, const std::size_t buff_size) = 0;
	//リソースビルダー破棄
	virtual bool deleteBuilder(const int builder_type_id, IResBuider* p) = 0;
	//リソースビルダーファクトリーの先頭ノードを取得
	static CAbstractResBuilderFactory* getFactoryHead() { return m_head; }
protected:
	//フィールド
	CAbstractResBuilderFactory* m_next;//次のリソースビルダーファクトリー
	static std::atomic<CAbstractResBuilderFactory*> m_head;//リソースビルダーファクトリーの先頭ノード
};

//リソースビルダーファクトリー抽象クラス：static 変数のインスタンス化
std::atomic<CAbstractResBuilderFactory*> CAbstractResBuilderFactory::m_head = nullptr;
                                                                    //リソースビルダーファクトリーの先頭ノード
```

```
//-------------------------------------
//リソースビルダーファクトリークラス
//※リソースビルダーを構築するためのクラス。
//  「アブストラクトファクトリーパターン」の実装。
//※ゲーム中で使用されるリソースビルダーファクトリーは
//  全てシングルトンとして保持し、連結リストでつながっている。
//※リソースビルダーを生成する際は、ファクトリーの連結リストをたどって
//  タイプ ID に該当するファクトリーでインスタンスを生成して返す。
//  「チェインオブレスポンシビリティパターン」の応用。
template<class T>
class CResBuilderFactory : public CAbstractResBuilderFactory
{
	friend class CResBuildReq<T>;
	friend void* operator new(std::size_t size, CAbstractResBuilderFactory&, char* buff, const std::size_t buff_size);
	friend void operator delete(void* p, CAbstractResBuilderFactory&, char*, const std::size_t);
public:
	//定数
	static const int RES_TYPE_ID = T::RES_TYPE_ID;//リソース種別 ID
	static const E_RES_ATTR RES_ATTR = T::RES_ATTR;//リソース属性
	static const int BUILDER_TYPE_ID = T::BUILDER_TYPE_ID;//リソースビルダー種別 ID
	static const E_RES_BUILD_ATTR TO_BUILD = T::TO_BUILD;//構築にはビルド処理が必要か？
	static const E_RES_BUILD_ATTR REQUIRED_FILE = T::REQUIRED_FILE;//構築にはファイルが必要か？
public:
	//アクセッサ
	int getBuilderTypeID() const override { return BUILDER_TYPE_ID; }//リソースビルダー種別 ID 取得
public:
	//メソッド
	//リソースビルダー構築
	//※オーバーライド
	IResBuider* createBuilder(const int builder_type_id, char* buff, const std::size_t buff_size) override
	{
		if (builder_type_id != BUILDER_TYPE_ID)
		{
			if (!m_next)
				return nullptr;
			//リソースビルダー種別 ID が異なる場合、次の連結リストにたくす
			//※チェインオブレスポンシビリティパターン
			return m_next->createBuilder(builder_type_id, buff, buff_size);
		}
		return new(this, buff, buff_size) T();
	}
	//リソースビルダー破棄
	//※オーバーライド
	bool deleteBuilder(const int builder_type_id, IResBuider* p) override
	{
		if (builder_type_id != BUILDER_TYPE_ID)
		{
			if (!m_next)
				return nullptr;
			//リソースビルダー種別 ID が異なる場合、次の連結リストにたくす
			//※チェインオブレスポンシビリティパターン
			return m_next->deleteBuilder(builder_type_id, p);
		}
		T* derived = dynamic_cast<T*>(p);
		if (!derived)
			return false;
		derived->~T();
		operator delete(derived, this, nullptr, 0);
		return true;
	}
private:
	//メソッド
	//自クラスのシングルトンインスタンス生成
	static void createSingleton()
	{
```

```
            if (m_this)
                  return;

            //シングルトンインスタンス生成
            m_this = new(m_this, m_buff, sizeof(m_buff))CResBuilderFactory<T>();

            //連結リストに連結
            //※スレッドセーフかつロックフリーな連結手法
            m_this->m_next = m_head.load();
            while (!m_head.compare_exchange_weak(m_this->m_next, m_this));

            //ID の重複チェック
            {
                  CAbstractResBuilderFactory* _this = m_this;
                  CAbstractResBuilderFactory* node = m_head.load();
                  while (node)
                  {
                        if (node != m_this && m_this->getBuilderTypeID() == node->getBuilderTypeID())
                        {
                              fprintf(stderr, "BUILDER_TYPE_ID(%d) is alread exist!¥n", m_this->BUILDER_TYPE_ID);
                        }
                        node = node->getNext();
                  }
            }
      }
private:
      //フィールド
      static CResBuilderFactory<T>* m_this;//自クラスシングルトンインスタンスのポインタ
      static char m_buff[];//自クラスシングルトンインスタンスのバッファ
};

//リソースビルダーファクトリーテンプレートクラスの static インスタンス生成用
#define RESOURCE_BUILDER_FACTORY_INSTANCE(T) ¥
      CResBuilderFactory<T>* CResBuilderFactory<T>::m_this = nullptr; ¥
      char CResBuilderFactory<T>::m_buff[sizeof(CResBuilderFactory<T>)];

//---------------------------------------
//リソースビルダーファクトリー処理用配置 new
void* operator new(std::size_t size, CAbstractResBuilderFactory*, char* buff, const std::size_t buff_size)
{
      if (size > buff_size)
            return nullptr;
      return buff;
}
//リソースビルダーファクトリー処理用配置 delete
void operator delete(void* p, CAbstractResBuilderFactory*, char*, const std::size_t)
{}

//---------------------------------------
//リソースビルダープロキシークラス
//※リソースビルダーのインスタンスを構築するためのバッファを持ち、
//  リソースビルダーとして振るまうクラス
//  （ローカル変数のバッファでリソースビルダーのインスタンスを生成するために使用する）
class CResBuiderProxy
{
public:
      //定数
      static const size_t BUILDER_SIZE_MAX = 32;//リソースビルダーの最大サイズ（バイト）
      //※リソースビルダーはスタックのメモリを使用する
      //※フィールドを一つも持たないリソースビルダーのサイズは 4 バイト
public:
      //メソッド
      //リソースビルド
      E_BUILD_RESULT create(const CResBuildReqInfo& req_info)
      {
```

```
            if (!m_builder)
                  return BUILD_IMPOSSIBLE;
            return m_builder->create(req_info);
      }
      //リソースコピービルド
      E_BUILD_RESULT copy(const CResBuildReqInfo& req_info, const void* src_res)
      {
            if (!m_builder || !src_res)
                  return BUILD_IMPOSSIBLE;
            return m_builder->copy(req_info, src_res);
      }
      //リソース破棄
      E_BUILD_RESULT destroy(void* res)
      {
            if (!m_builder)
                  return BUILD_IMPOSSIBLE;
            return m_builder->destroy(res);
      }
public:
      //コンストラクタ
      CResBuiderProxy(const int builder_type_id) :
            m_builderType(builder_type_id),
            m_builder(nullptr)
      {
            CAbstractResBuilderFactory* factory = CAbstractResBuilderFactory::getFactoryHead();
            if (factory)
                  m_builder = factory->createBuilder(m_builderType, m_builderBuff, BUILDER_SIZE_MAX);
      }
      //デストラクタ
      ~CResBuiderProxy()
      {
            CAbstractResBuilderFactory* factory = CAbstractResBuilderFactory::getFactoryHead();
            if (factory)
                  factory->deleteBuilder(m_builderType, m_builder);
      }
private:
      //フィールド
      const int m_builderType;//ビルダー種別
      IResBuider* m_builder;//リソースビルダー
      char m_builderBuff[BUILDER_SIZE_MAX];//リソースビルダー用バッファ
};
```

【サンプル用ダミー：リソースマネージャ】

```
//-----------------------------------------------------------------------------
//【サンプル用ダミー】リソースマネージャクラス

#include <vector>

//リソースマネージャクラス（ダミー）
class CResManager
{
public:
      //メソッド
      //ビルド要求
      //※リソースにアクセスするためのハンドルを返す
      template<class T>
      HRES request(T& req)
      {
            HRES handle = req.makeHandle();
            requestCore(req);
            return handle;
      }
      //全要求をビルド（ダミー）
      //※本来このようなメソッドではなく、一つ一つのビルド処理をジョブとして投入する
      void createAll()
      {
```

```
            for (CResBuildReqInfo& build_info : m_reqList)
            {
                  CResBuiderProxy proxy(build_info.getBuilderTypeId());
                  proxy.create(build_info);
            }
      }
      //全要求をコピーでビルド（ダミー）
      //※本来このようなメソッドではなく、一つ一つのビルド処理をジョブとして投入する
      void copyAll()
      {
            for (CResBuildReqInfo& build_info : m_reqList)
            {
                  CResBuiderProxy proxy(build_info.getBuilderTypeId());
                  void* src_res_dummy = "";
                  proxy.copy(build_info, src_res_dummy);
            }
      }
      //全リソースを破棄（ダミー）
      //※本来このようなメソッドではなく、一つ一つの破棄処理をジョブとして投入する
      void destroyAll()
      {
            for (CResBuildReqInfo& build_info : m_reqList)
            {
                  CResBuiderProxy proxy(build_info.getBuilderTypeId());
                  void* res_dummy = "";
                  proxy.destroy(res_dummy);
            }
      }
private:
      void requestCore(CResBuildReqInfo& req)
      {
            m_reqList.push_back(req);
      }
private:
      //フィールド
      std::vector<CResBuildReqInfo> m_reqList;//ビルド要求リスト
};

//リソースマネージャシングルトンインスタンス
SINGLETON_INSTANCE(CResManager);
```

【タイトル固有のリソースビルダークラス】

```
//-----------------------------------------------------------------------------
//【サンプル】タイトル固有のリソースビルダークラス

//-------------------------------------
//定数：リソース種別 ID
enum E_RES_TYPE_ID
{
      RES_UNKNOWN = 0,
      RES_MODEL,//モデル
      RES_TEXTURE,//テクスチャ
};

//-------------------------------------
//定数：リソースビルダー種別 ID
enum E_BUILDER_TYPE_ID
{
      BUILDER_UNKNOWN = 0,
      BUILDER_MODEL,//モデル構築
      BUILDER_TEXTURE,//テクスチャ構築
};

//-------------------------------------
//定数：メモリ ID
enum E_MEM_ID
```

```
{
	MEM_MAIN_MODEL,//メインメモリ：モデル情報用
	MEM_GPU_MODEL,//GPU メモリ：モデル構築用
	MEM_GPU_TEX,//GPU メモリ：テクスチャ構築用
};

//---------------------------------------
//独自リソースビルダークラス：モデル用
//※ビルド実行時に、ビルドスレッド処理がインスタンスを生成して
//  create() メソッドを実行する。
//※create () メソッドのパラメータ CResBuildReqInfo に、
//  読み込んだファイルのバッファや、構築先のメモリ情報などが格納されている
//※デフォルトコンストラクタしか使えない
//※フィールドは好きに追加して良いが、
//  CResBuiderProxy::BUILDER_SIZE_MAX を超えるサイズは不可。
//※リソースビルダーは実行の都度インスタンスを生成するが、
//  CResBuiderProxy によってローカル変数として生成される。
class CModelResourceBuilder : public IResBuider
{
public:
	//定数
	//※必須
	static const E_RES_TYPE_ID RES_TYPE_ID = RES_MODEL;//リソース種別 ID
	static const E_RES_ATTR RES_ATTR = RES_ATTR_MOVABLE;//リソース属性：再配置可能
	static const E_BUILDER_TYPE_ID BUILDER_TYPE_ID = BUILDER_MODEL;//リソースビルダー種別 ID
	static const E_RES_BUILD_ATTR TO_BUILD = RES_TO_BUILD;//構築にはビルド処理が必要か？
	static const E_RES_BUILD_ATTR REQUIRED_FILE = RES_REQUIRED_FILE;//構築にはファイルが必要か？
public:
	//メソッド
	//メモリ情報を作成（デフォルト設定）
	//※必須
	static void setDefaultMemInfo(CResBuildReqInfo& req_info)
	{
		req_info.addMemInfo(CResBuildReqInfo::T_MEM_INFO(MEM_MAIN_MODEL));
		req_info.addMemInfo(CResBuildReqInfo::T_MEM_INFO(MEM_GPU_MODEL));
	}
public:
	//メソッド
	//ハンドル発行
	//※オーバーライド
	HRES makeHandle(const CResBuildReqInfo& req_info) override
	{
		return req_info.makeNormalHandle();
	}
	//リソース構築
	//※オーバーライド
	E_BUILD_RESULT create(const CResBuildReqInfo& req_info) override
	{
		printf("CModelResourceBuilder::create(%d, %d)¥n", req_info.getResTypeId(), req_info.getBuilderTypeId());
		return BUILD_SUCCESS;
	}
	//リソースコピー
	//※オーバーライド
	E_BUILD_RESULT copy(const CResBuildReqInfo& req_info, const void* src_res) override
	{
		printf("CModelResourceBuilder::copy(%d, %d, %p)¥n", req_info.getResTypeId(), req_info.getBuilderTypeId(),
                                                                                                    src_res);
		return BUILD_SUCCESS;
	}
	//リソース破棄
	//※オーバーライド
	E_BUILD_RESULT destroy(void* res) override
	{
		printf("CModelResourceBuilder::destroy(%p)¥n", res);
		return BUILD_SUCCESS;
```

```
	}
public:
	//コンストラクタ
	CModelResourceBuilder()
	{
	}
	//デストラクタ
	~CModelResourceBuilder() override
	{
	}
};

//リソースビルダーファクトリーシングルトンの static インスタンス化
RESOURCE_BUILDER_FACTORY_INSTANCE(CModelResourceBuilder);

//----------------------------------------
//独自リソースビルダークラス：テクスチャ用
class CTextureResourceBuilder : public IResBuider
{
public:
	//定数
	//※必須
	static const E_RES_TYPE_ID RES_TYPE_ID = RES_TEXTURE;//リソース種別 ID
	static const E_RES_ATTR RES_ATTR = RES_ATTR_MOVABLE;//リソース属性：再配置可能
	static const E_BUILDER_TYPE_ID BUILDER_TYPE_ID = BUILDER_TEXTURE;//リソースビルダー種別 ID
	static const E_RES_BUILD_ATTR TO_BUILD = RES_TO_COPY_ONLY;//構築にはビルド処理が必要か？
	static const E_RES_BUILD_ATTR REQUIRED_FILE = RES_REQUIRED_FILE;//構築にはファイルが必要か？
public:
	//メソッド
	//メモリ情報を作成（デフォルト設定）
	//※必須
	static void setDefaultMemInfo(CResBuildReqInfo& req_info)
	{
		req_info.addMemInfo(CResBuildReqInfo::T_MEM_INFO(MEM_GPU_TEX));
	}
public:
	//メソッド
	//ハンドル発行
	//※オーバーライド
	HRES makeHandle(const CResBuildReqInfo& req_info) override
	{
		return req_info.makeNormalHandle();
	}
	//リソース構築
	//※オーバーライド
	E_BUILD_RESULT create(const CResBuildReqInfo& req_info) override
	{
		printf("CTextureResourceBuilder::create(%d, %d)¥n", req_info.getResTypeId(), req_info.getBuilderTypeId());
		return BUILD_SUCCESS;
	}
	//リソースコピー
	//※オーバーライド
	E_BUILD_RESULT copy(const CResBuildReqInfo& req_info, const void* src_res) override
	{
		printf("CTextureResourceBuilder::copy(%d, %d, %p)¥n", req_info.getResTypeId(), req_info.getBuilderTypeId(),
                                                                                                    src_res);
		return BUILD_SUCCESS;
	}
	//リソース破棄
	//※オーバーライド
	E_BUILD_RESULT destroy(void* res) override
	{
		printf("CTextureResourceBuilder::destroy(%p)¥n", res);
		return BUILD_SUCCESS;
	}
```

```
public:
	//コンストラクタ
	CTextureResourceBuilder()
	{
	}
	//デストラクタ
	~CTextureResourceBuilder() override
	{
	}
};

//リソースビルダーファクトリーシングルトンの static インスタンス化
RESOURCE_BUILDER_FACTORY_INSTANCE(CTextureResourceBuilder);
```

【ビルド要求処理】

```
//-----------------------------------------------------------------------------
//【サンプル】ビルド要求処理

//---------------------------------------
//ビルドリクエスト①
void func1()
{
	//モデルリソースのビルド要求情報を構築
	CResBuildReq<CModelResourceBuilder> req("/data/chara/x0010.mdl");

	//ビルド要求
	CSingletonUsing<CResManager> res_man("func1");//リソースマネージャシングルトンにアクセス
	HRES handle = res_man->request(req);
}

//---------------------------------------
//ビルドリクエスト②
void func2()
{
	//テクスチャリソースのビルド要求情報を構築
	CResBuildReq<CTextureResourceBuilder> req("/data/chara/x0010.tex");
	req.getMemInfo(0)->m_isLocked = true;//メモリ設定変更

	//ビルド要求
	CSingletonUsing<CResManager> res_man("func2");//リソースマネージャシングルトンにアクセス
	HRES handle = res_man->request(req);
}
```

【ダミー：ビルド処理】

```
//-----------------------------------------------------------------------------
//【サンプル】ビルド処理

//リソース構築処理（ダミー）
void createAll()
{
	//全要求をビルド
	CSingletonUsing<CResManager> res_man("createAll");//リソースマネージャシングルトンにアクセス
	res_man->createAll();
}

//リソースコピー処理（ダミー）
void copyAll()
{
	//全要求をビルド
	CSingletonUsing<CResManager> res_man("copyAll");//リソースマネージャシングルトンにアクセス
	res_man->copyAll();
}

//リソース破棄処理（ダミー）
void destroyAll()
{
	//全要求をビルド
```

```
    CSingletonUsing<CResManager> res_man("destroyAll");//リソースマネージャシングルトンにアクセス
    res_man->destroyAll();
}
```

【テストメイン】

```
//-----------------------------------------------------------------------------
//【サンプル】テストメイン
int main(const int argc, const char* argv[])
{
    //ビルド要求
    printf("----- ビルド要求 -----¥n");
    func1();
    func2();

    //ビルド実行
    printf("----- ビルド実行 -----¥n");
    createAll();

    //コピー実行
    printf("----- コピー実行 -----¥n");
    copyAll();

    //リソース破棄
    printf("----- リソース破棄 -----¥n");
    destroyAll();

    //終了
    return EXIT_SUCCESS;
}
```

↓（実行結果）

```
----- ビルド要求 -----
----- ビルド実行 -----
CModelResourceBuilder::create(1, 1)
CTextureResourceBuilder::create(2, 2)
----- コピー実行 -----
CModelResourceBuilder::copy(1, 1, 008D21BB)
CTextureResourceBuilder::copy(2, 2, 008D21BB)
----- リソース破棄 -----
CModelResourceBuilder::destroy(008D21BB)
CTextureResourceBuilder::destroy(008D21BB)
```

▼ リソースアクセス処理：リード・ライトロックの活用

前述の要件定義で示しているとおり、リソースアクセス時はリード・ライトロックを用いることで、効率化と安全性の両面を最適化する。

● リード・ライトロックによる最適化

リード・ライトロック機構により、「リソースに対する読み込みアクセスは複数の処理が同時に行うことができるが、書き込みが発生したら一切同時アクセスできない」ということを保証する。

また、プログラミング構造上、「リソースアクセス時は必然的にリソースをロックし、アクセス終了時に確実にロック解除する」という安全性も保証する。

● プログラミングのイメージ

リード・ライトロック機構については、別紙の「効率化と安全性のためのロック制御」で詳しく説明している。そこに記載しているサンプルプログラムとほぼ同じものだが、上記の要件を踏まえたプログラミングのイメージを以下に示す。

必然的なリード・ライトロックを伴うリソースアクセスのイメージ：

```
//リソースマネージャのインスタンス（シングルトン）取得
CSingltonUsing<CResMan> res_man;

//モデルデータのリード処理ブロック
{
	//リソースをリソース ID で検索して取得、同時にリードロック
	//※変数 res_id はリソース ID
	//※CModel がリソースのインスタンスの型
	CResR<CModel> model(res_man, res_id);
	//※ライトロック時は CResW<T> を使用。

	//ロック取得後は、リソースの存在をチェックしてからアクセスする
	//※ロック取得待機中にリソースが破棄された可能性もある
	if(model.isExist())
	{
		//ロックオブジェクト CResR<T> が、クラス T のプロキシーとして振る舞う
		model->member1(); //リードロック時は、const CModel* 型の変数として振る舞う（アロー演算子でメンバーアクセス）
		model->member2(); //ライトロック時は、CModel* 型の変数として振る舞う（アロー演算子でメンバーアクセス）
		…処理…
	}
}
//処理ブロック終了時に、CResR<T> のデストラクタにより、自動的にロック解放
//※CResR<T>::unlock() により明示的な解放と、CResR<T>::rLock() などによる明示的な再ロックも可能。
```

▼ リソースツリー管理：赤黒木（平衡木）

リソース管理情報のツリーは、検索効率のために、平衡木を使用する。

ハンドルをキーにした赤黒木を想定する。ハンドルには絶対に重複がない。

● マスク検索

リソース検索は「マスク検索」に対応する。

二分木は大小比較で検索を行うため、上位ビットでマスクして検索する分には、検索効率をほとんど損ねない。それを利用し、下記のような検索を可能とする。

- 【前提】ハンドルは、
  「任意の識別 ID（主にシーン ID）」（24bit）＋
  「リソースファイルパスの CRC」（32bit）＋
  「任意のインデックス」（8bit）
  の 64bit で構成。
- 【前提】静的リソースは、0（24bit）＋CRC（32bit）＋インデックス（8bit：通常は

0）で構成される。
別メモリブロックに同じリソースのインスタンスが作られると、インデックスが 1 以上の値になることがある。
どのインスタンスであっても、CRC が一致するものは必ず同じ内容。

・ 【検索要件】どのメモリブロックのインスタンスでもかまわないので、リソースを検索したい。
⇒上位 56bit でマスク検索。
・ 【検索要件】指定のシーン ID 固有のリソースを全て列挙したい。
⇒上位 24bit でマスク検索。

● ツリーの連結情報

このようなマスク検索の要件を踏まえて、検索結果を受け取った処理が「ノードの次のノードを取得」という操作を行えるようにするために、ツリーの連結情報は「小」「大」に加えて「親」も持つものとする。所用メモリが大きくなるが、アクセス速度を重視する。

● リード・ライトロック

「検索」などのツリーへのアクセス時は、リードロックを暗黙的に用いる。
また、「ノードの追加・削除」といったツリーの組み替え操作を行う際は、ライトロックを明示的に用いる。

■ データ仕様

▼ リソース管理情報

以上の要件を踏まえたリソース管理情報のデータ構造を示す。

リソース管理情報	サイズ (32bit)	サイズ (64bit)
連結情報		
親ノードのインデックス	2 bytes	2 bytes
小ノードのインデックス	2 bytes	2 bytes
大ノードのインデックス	2 bytes	2 bytes
依存関係情報	(32bit)	(64bit)
依存関係リスト①の先頭インデックス	2 bytes	2 bytes
依存関係リスト②の先頭インデックス	2 bytes	2 bytes
構築状態	2 bytes	2 bytes
参照カウンタ	2 bytes	2 bytes
(パディング)	2 bytes	2 bytes
リソースハンドル	8 bytes	8 bytes
リード・ライトロックオブジェクト	28 bytes	28 bytes
インスタンス情報①		
メモリブロック識別ID	4 bytes	4 bytes
サイズ	4 bytes	4 bytes
ポインタ	4 bytes	8 bytes
インスタンス情報②		
メモリブロック識別ID	4 bytes	4 bytes
サイズ	4 bytes	4 bytes
ポインタ	4 bytes	8 bytes
計:	76 bytes	84 bytes
最大ノード数:	16,384 個	16,384 個
合計サイズ:	1,245,184 bytes 1,216.000 KB 1.188 MB	1,376,256 bytes 1,344.000 KB 1.313 MB

依存関係リスト①:依存元→移動先方向		
連結情報		
次ノードのインデックス	2 bytes	2 bytes
依存関係情報		
依存先管理情報インデックス	2 bytes	2 bytes
計:	4 bytes	4 bytes
最大ノード数:	65,536 個	65,536 個
合計サイズ:	262,144 bytes 256.000 KB 0.250 MB	262,144 bytes 256.000 KB 0.250 MB

依存関係リスト②:依存先→移動元方向		
連結情報		
次ノードのインデックス	2 bytes	2 bytes
依存関係情報		
依存元管理上オフインデックス	2 bytes	2 bytes
計:	4 bytes	4 bytes
最大ノード数:	65,536 個	65,536 個
合計サイズ:	262,144 bytes 256.000 KB 0.250 MB	262,144 bytes 256.000 KB 0.250 MB
全体サイズ:	1,769,472 bytes 1,728.000 KB 1.688 MB	1,900,544 bytes 1,856.000 KB 1.813 MB

■■以上■■

■ 索引

索引項目が見つかりません。

開発の効率化と安全性のためのリソース管理

以　上