

The Book of the Runtime

日本語版

.NET ランタイムの内部構造をわかりやすく日本語で解説

日本語で読める

.NET ランタイムの内部ドキュメントを日本語に翻訳して提供します。

初心者にもわかりやすい注釈

原文に加えて、プログラミング初級～中級者向けのわかりやすい注釈を追加しています。

オープンソース

誰でも翻訳や注釈の改善に貢献できます。GitHub で PR をお待ちしています。

The Book of the Runtime (BOTR) とは

The Book of the Runtime (BOTR) は、.NET ランタイムの内部構造について解説したドキュメント集です。ランタイムのコードを変更する開発者や、ランタイムの深い理解を求める方を対象としています。

このサイトでは、BOTR の内容を日本語に翻訳し、さらにプログラミング初級～中級者にもわかりやすいような注釈をつけて解説しています。

目次

- [BOTR FAQ](#) - よくある質問
- [CLR 入門](#) - 共通言語ランタイム (CLR) の概要
- [ガベージコレクション](#) - GC の設計と仕組み
- [スレッディング](#) - スレッド管理の仕組み
- [RyuJIT 概要](#) - JIT コンパイラの概要
- [RyuJIT の他プラットフォームへの移植](#) - RyuJIT の移植
- [型システム](#) - 型システムの設計
- [型ローダー](#) - 型の読み込み処理
- [メソッドディスクリプタ](#) - メソッドの記述子
- [仮想スタブディスパッチ](#) - 仮想メソッドのディスパッチ
- [スタックウォーキング](#) - スタックの走査
- [System.Private.CoreLib](#) - CoreLib とランタイムの呼び出し
- [DAC ノート](#) - データアクセスコンポーネント
- [プロファイリング](#) - プロファイリングの仕組み
- [プロファイラビリティの実装](#) - プロファイラビリティの実装
- [例外処理](#) - ランタイムの例外処理
- [ReadyToRun 概要](#) - AOT コンパイルの概要
- [CLR ABI](#) - CLR の ABI 仕様
- [クロスプラットフォームミニダンプ](#) - ミニダンプ生成
- [混合モードアセンブリ](#) - 混合モードアセンブリ
- [移植ガイド](#) - ランタイムの移植ガイド
- [ベクトルと組み込み関数](#) - SIMD サポート
- [ILC コンパイラアーキテクチャ](#) - ILC コンパイラの設計
- [マネージド型システムの概要](#) - マネージド型システム
- [ReadyToRun PerfMap フォーマット](#) - PerfMap フォーマット
- [ReadyToRun ファイルフォーマット](#) - R2R ファイル形式
- [ReadyToRun ネイティブエンベロープ](#) - ネイティブエンベロープ
- [共有ジェネリクスの設計](#) - 共有ジェネリクス
- [開発者向けランタイムロギング](#) - ランタイムのロギング

翻訳への貢献

翻訳の改善や新しい章の翻訳は、[GitHub リポジトリ](#) で受け付けています。

BOTR FAQ

原文

この章の原文は [Book of the Runtime FAQ](#) です。

The Book of the Runtime (BOTR) とは何ですか？

The Book of the Runtime (BOTR) は、.NET ランタイムの内部設計について解説した文書集です。ランタイムの非自明な内部構造について、開発者やランタイムの深い理解を求める人向けに書かれています。

 初心者向け補足

「ランタイム」とは、プログラムを実行するためのソフトウェアのことです。.NET ランタイムは、C# や F# などの.NET 言語で書かれたプログラムを実行する環境を提供します。

誰が BOTR を書いていますか？

BOTR は、.NET ランタイムの開発に携わるエンジニアたちによって書かれています。各章は、その分野に詳しい開発者が執筆しています。

BOTR はどのような人を対象としていますか？

主に以下のような人を対象としています：

- .NET ランタイムのソースコードを変更する開発者
- ランタイムの内部動作を深く理解したい開発者
- コンパイラやガベージコレクタなどの低レベルな仕組みに興味がある方

 初心者向け補足

この日本語版では、原文よりも幅広い読者を対象として、初級～中級のプログラマーにもわかりやすい注釈を追加しています。

共通言語ランタイム (CLR) 入門

原文

この章の原文は [Introduction to the Common Language Runtime \(CLR\)](#) です。

著者: Vance Morrison ([@vancem](#)) - 2007

共通言語ランタイム (Common Language Runtime, CLR) とは何でしょうか？簡潔に言えば：

共通言語ランタイム (CLR) とは、幅広いプログラミング言語とそれらの相互運用をサポートするために設計された、完全で高水準な仮想マシンです。

すいぶんと大げさな表現です。しかし、それだけでは大して意味が分かりません。しかしこの表現は 有用です。なぜなら、[CLR](#) という大規模で複雑なソフトウェアの機能を理解しやすい形にまとめるための第一歩だからです。これにより、ランタイムの大まかな目標と目的を理解するための「上空 10,000 フィート」からの視点を得ることができます。この高い視点で CLR を理解した後であれば、サブコンポーネントをより深く見ても、詳細の中で迷子になる可能性が低くなります。

初心者向け補足

「仮想マシン (Virtual Machine)」とは、ソフトウェアによって実現された仮想的なコンピュータのことです。CLR は Java の JVM (Java Virtual Machine) に相当するもので、C# や VB.NET などで書かれたプログラムの実行環境を提供します。プログラムのコードを実行するだけでなく、メモリ管理、セキュリティ、例外処理など、プログラムの実行に必要なあらゆる機能を提供する「裏方のソフトウェア」です。

CLR：(非常にまれな) 完全なプログラミングプラットフォーム

すべてのプログラムは、その実行環境に対して驚くほど多くの依存関係を持っています。最も明らかなのは、プログラムが特定のプログラミング言語で書かれていることですが、それはプログラマがプログラムに織り込む多くの前提の最初の一つに過ぎません。すべての意味のあるプログラムは、マシンの他のリソース（ユーザー入力、ディスクファイル、ネットワーク通信など）とやり取りするための何らかの ランタイムライブラリを必要とします。プログラムはまた、ネイティブハードウェアが直接実行できる形式に何らかの方法で変換される（インタプリタまたはコンパイルによって）必要があります。プログラムのこれらの依存関係は非常に多く、相互に依存し、多様であるため、プログラミング言語の実装者はほとんど常に他の標準に仕様の策定を委ねます。たとえば、C++ 言語は C++ 実行ファイルのフォーマットを規定していません。代わりに、各 C++ コンパイラは特定のハードウェアアーキテクチャ（例：X86）とオペレーティングシステム環境（例：Windows、Linux、Mac OS）に結びつけられており、それらが実行ファイル形式とロード方法を規定します。したがって、プログラマは「C++ 実行ファイル」ではなく、「Windows X86 実行ファイル」や「Power PC Mac OS 実行ファイル」を作成することになります。

既存のハードウェアやオペレーティングシステムの標準を活用することは通常良いことですが、仕様を既存の標準の抽象化レベルに縛りつけてしまうという欠点があります。たとえば、今日の一般的なオペレーティングシステムにはガベージコレクションヒープ (garbage-collected heap) という概念はありません。したがって、ガベージコレクションを活用するインターフェースを、既存の標準を使って記述する方法がないのです（たとえば、誰が文字列を削除する責任があるかを気にせずに文字列をやり取りするなど）。同様に、一般的な実行ファイル形式はプログラムを実行するための十分な情報は提供しますが、コンパイラが他のバイナリをその実行ファイルにバインドするための十分な情報は提供しません。たとえば、C++ プログラムは一般的に標準ライブラリ（Windows では msrvct.dll と呼ばれる）を使用し、これには一般的な機能のほとんど（例：printf）が含まれますが、そのライブラリが存在するだけでは十分ではありません。それに対応するヘッダファイル（例：stdio.h）がなければ、プログラマはそのライブラリを使えません。したがって、既存の実行ファイル形式の

標準は、実行可能なファイル形式の記述とプログラムを完全にするために必要な他の情報やバイナリの記述の両方に使うことはできません。

CLR は、非常に完全な仕様 (ECMA で標準化) を定義することでこれらの問題を解決します。この仕様には、構築とバインドからデプロイと実行まで、プログラムの完全なライフサイクルに必要な詳細が含まれています。したがって、CLR は特に以下を規定しています：

- プログラムが実行するプリミティブな操作を指定するための、独自の命令セット (共通中間言語 (Common Intermediate Language, CIL) と呼ばれる) を持つ GC 対応の仮想マシン。これにより、CLR は特定の種類の CPU に依存しません。
- プログラム宣言 (型、フィールド、メソッドなど) のためのリッチなメタデータ (metadata) 表現。これにより、他の実行ファイルを生成するコンパイラは「外部」の機能を呼び出すために必要な情報を得ることができます。
- ファイルにビットをどのように配置するかを正確に指定するファイルフォーマット。これにより、特定のオペレーティングシステムやコンピュータハードウェアに縛られない CLR EXE を適切に語ることができます。
- ロードされたプログラムのライフタイムセマンティクス (lifetime semantics)、ある CLR EXE ファイルが別の CLR EXE を参照するメカニズム、および実行時にランタイムが参照先のファイルを見つける方法に関するルール。
- CLR が提供する機能 (ガベージコレクション、例外、ジェネリック型など) を活用して、基本的な機能 (整数、文字列、配列、リスト、辞書など) とオペレーティングシステムサービス (ファイル、ネットワーク、ユーザーインターフェースなど) の両方にアクセスを提供するクラスライブラリ。

初心者向け補足

CIL (Common Intermediate Language) は、C# や VB.NET などのソースコードがコンパイルされた結果生成される中間言語です。Java のバイトコードに相当します。C# のコードは最初に CIL にコンパイルされ、実行時に JIT (Just-In-Time) コンパイラによってネイティブコード (CPU が直接実行するマシン語) に変換されます。この 2 段階のコンパイルにより、同じ CIL を異なる CPU アーキテクチャ (x86、ARM など) で実行できるという「プラットフォーム非依存性」が実現されます。

多言語サポート

これらの詳細をすべて定義し、仕様化し、実装することは非常に大きな取り組みです。だからこそ、CLR のような完全な抽象化は非常にまれなのです。実際、このような合理的に完全な抽象化の大部分は、単一の言語のために構築されました。たとえば、Java ランタイム、Perl インタプリタ、または初期バージョンの Visual Basic ランタイムは、同様に完全な抽象化境界を提供しています。これらの先行する取り組みと CLR を区別するのは、そのマルチランゲージ (多言語) の性質です。Visual Basic (COM オブジェクトモデルを活用しているため) の唯一の例外の可能性を除けば、言語内での体験はしばしば非常に良いものですが、他の言語で書かれたプログラムとの相互運用は非常に困難です。相互運用が困難なのは、これらの言語はオペレーティングシステムが提供するプリミティブを使ってのみ「外国语」と通信できるからです。OS の抽象化レベルが非常に低い (たとえば、オペレーティングシステムにはガベージコレクションヒープの概念がない) ため、不必要に複雑なテクニックが必要になります。共通言語ランタイムを提供することで、CLR は言語間で高水準のコントラクト (例：GC で管理される構造体) を使って通信できるようにし、相互運用の負担を劇的に軽減しています。

ランタイムが多くの言語間で共有されるということは、それを十分にサポートするためにより多くのリソースを投入できることを意味します。言語のための優れたデバッガやプロファイルを構築するのは多大な作業であり、そのため、それらはフル機能の形で最も重要なプログラミング言語に対してのみ存在します。しかし、CLR 上に実装された言語はこのインフラストラクチャを再利用できるため、個々の言語にかかる負担は大幅に軽減されます。おそらくさらに重要なのは、CLR 上に構築された言語は、CLR 上に構築されたすべてのクラスライブラリに即座にアクセスできることです。この大規模な (そして成長し続ける)、デバッグ済みでサポートされた機能群が、CLR が大成功を収めた大きな理由です。

要するに、ランタイムはプログラムを作成し実行するためにファイルに配置する必要がある正確なビットの完全な仕様です。これらのファイルを実行する仮想マシンは、幅広いプログラミング言語を実装するのに適した高水準なレベルにあります。この仮想マシンと、その仮想マシン上で実行されるクラスライブラリの絶えず増え続ける集合体のことを、私たちは共通言語ランタイム (CLR) と呼びます。

CLR の主要な目標

CLR がどのようなものか基本的な理解ができたところで、少し戻って、ランタイムが解決しようとしていた問題を理解するのが有益です。非常に高いレベルでは、ランタイムにはたった一つの目標しかありません：

CLR の目標は、プログラミングを簡単にすることです。

この主張が有用なのは 2 つの理由からです。第一に、ランタイムが進化する際の 非常に有用な指針となります。たとえば、基本的に単純なものだけが簡単になり得るため、ランタイムにユーザーに見える 複雑さを追加することは常に疑いの目で見るべきです。機能のコスト/ベネフィット比よりも重要なのは、その 追加される露出した複雑さ/すべてのシナリオにわたる重み付けされた利点の比率です。理想的には、この比率は負（つまり、新しい機能が制限を取り除いたり、既存の特殊ケースを一般化したりすることで複雑さを減少させる）です。しかし、より一般的には、露出する複雑さを最小化し、機能が価値を追加するシナリオの数を最大化することで、この比率を低く保ちます。

この目標がとても重要である第二の理由は、使いやすさこそが CLR の成功の根本的な理由 だということです。CLR がネイティブコードを書くよりも速いまたは小さいから成功したのではありません（実際、よく書かれたネイティブコードの方がしばしば優れています）。CLR がガーベージコレクション、プラットフォーム非依存性、オブジェクト指向プログラミング、バージョニングサポートなどの特定の機能をサポートしているから成功したのでもありません。CLR が成功したのは、それらすべての機能と他の多くの機能が組み合わあって、プログラミングをそうでなかった場合よりも大幅に簡単にしているからです。重要でありながらしばしば見過ごされる使いやすさの機能には以下が含まれます：

1. 簡素化された言語（例：C# や Visual Basic は C++ よりもはるかにシンプルです）
2. クラスライブラリにおけるシンプルさへの献身（例：文字列型は 1 つだけで、それはイミュータブル (immutable) です。これにより、文字列を使用するすべての API が大幅に簡素化されます）
3. クラスライブラリにおける命名の強い一貫性（例：API には完全な単語と一貫した命名規則を使用することが求められます）
4. アプリケーションの作成に必要なツールチェーンの優れたサポート（例：Visual Studio は CLR アプリケーションの構築を非常に簡単にし、Intellisense はアプリケーション作成に適切な型やメソッドを見つけるのを非常に容易にします）

使いやすさへのこの献身（これはユーザーモデルのシンプルさと表裏一体です）こそが、CLR の成功の理由として際立っています。奇妙なことに、最も重要な使いやすさの機能のいくつかは、最も「退屈」なものもあります。たとえば、どのプログラミング環境でも一貫した命名規則を適用できますが、大規模なクラスライブラリ全体にわたって実際にそれを行うのはかなりの作業です。多くの場合、そのような取り組みは他の目標（既存インターフェースとの互換性の維持など）と衝突したり、重大な物流上の懸念（非常に大きなコードベース全体でメソッドの名前を変更するコストなど）に直面したりします。このようなときこそ、ランタイムの最も重要な包括的目標を思い出し、その目標に到達するために優先順位を正しくする必要があります。

CLR の基本機能

ランタイムには多くの機能があるため、以下のようにカテゴリ分けすると有益です：

1. 基本機能 – 他の機能の設計に広範な影響を与える機能。以下が含まれます：
 1. ガベージコレクション (Garbage Collection)
 2. メモリ安全性 (Memory Safety) と型安全性 (Type Safety)
 3. プログラミング言語のための高水準サポート
2. 二次的機能 – 基本機能によって実現される機能で、多くの有用なプログラムでは必要とされない可能性がある機能：
 1. AppDomain によるプログラムの分離

2. プログラムのセキュリティとサンドボックス化
3. その他の機能 – すべてのランタイム環境が必要とするが、CLR の基本機能を活用しない機能。これらは完全なプログラミング環境を作成したいという要望の結果です。含まれるものとしては:
 1. バージョニング (Versioning)
 2. デバッグ/プロファイリング (Debugging/Profiling)
 3. 相互運用 (Interoperation)

CLR ガベージコレクタ (GC)

CLR が提供するすべての機能の中で、ガベージコレクタは特別に注目に値します。ガベージコレクション (GC) とは、自動メモリ回収の一一般的な呼称です。ガベージコレクションされるシステムでは、ユーザープログラムはメモリを削除するために特別な演算子を呼び出す必要がなくなります。代わりに、ランタイムがガベージコレクションヒープ内のメモリへのすべての参照を自動的に追跡し、時々それらの参照をたどって、どのメモリがまだプログラムから到達可能かを調べます。それ以外のすべてのメモリは ガベージであり、新しいアロケーション (allocation) に再利用できます。

💡 初心者向け補足

ガベージコレクション (GC) は、Java や Python などの言語でも採用されているメモリ管理方式です。C/C++ ではプログラマが `malloc / free` や `new / delete` を使って手動でメモリを管理しますが、GC がある環境ではランタイムが「もう使われていないメモリ」を自動的に検出して回収してくれます。これにより、メモリリーク（使い終わったメモリを解放し忘れる）やダングリングポインタ（すでに解放されたメモリを参照してしまう）といったバグを大幅に削減できます。

ガベージコレクションはプログラミングを簡素化するため、素晴らしいユーザー機能です。最も明らかな簡素化は、ほとんどの明示的な `delete` 操作が不要になることです。`delete` 操作をなくすことは重要ですが、プログラマにとっての本当の価値はもう少し繊細です：

1. ガベージコレクションはインターフェースの設計を簡素化します。なぜなら、インターフェースを越えて渡されるオブジェクトの削除にどちら側が責任を持つかを注意深く指定する必要がなくなるからです。たとえば、CLR のインターフェースは単に文字列を返します。文字列バッファとその長さを受け取ることはしません。つまり、バッファが小さすぎる場合にどうなるかという複雑さに対処する必要がないのです。したがって、ガベージコレクションにより、ランタイムのすべてのインターフェースが、そうでなかつた場合よりもシンプルになります。
2. ガベージコレクションは、一般的なユーチューブのクラス全体を排除します。特定のオブジェクトのライフタイムに関するミスを犯すのは恐ろしいほど簡単で、早すぎる削除（メモリ破損につながる）や遅すぎる削除（到達不可能なメモリリーク）が起こり得ます。一般的なプログラムは文字通り何百万ものオブジェクトを使用するため、エラーの確率は非常に高くなります。さらに、特にオブジェクトが多くの他のオブジェクトから参照されている場合、ライフタイムのバグを追跡することは非常に困難です。このクラスのミスを不可能にすることで、多くの苦労を避けることができます。

しかし、ここで特別に注目するに値するのは、ガベージコレクションの有用性ではありません。もっと重要なのは、それがランタイム自体に課す単純な要件です：

ガベージコレクションは、GC ヒープへのすべての参照を追跡する必要があります。

これは非常にシンプルな要件ですが、実際にはランタイムに対して深遠な影響を及ぼします。想像できるように、プログラム実行のあらゆる瞬間にオブジェクトへのすべてのポインタがどこにあるかを知ることは、非常に困難です。一つの軽減要因があります。厳密に言えば、この要件は GC が実際に発生する必要があるときにのみ適用されます（したがって、理論的には常にすべての GC 参照がどこにあるか知る必要はなく、GC 時にのみ知ればよい）。しかし実際には、CLR の別の機能のため、この軽減は完全には当てはまりません：

CLR は、単一のプロセス内で複数の同時実行スレッドをサポートしています。

いつでも、他の実行スレッドがガベージコレクションを必要とするアロケーションを実行する可能性があります。同時実行スレッド間の正確な操作のシーケンスは非決定的です。あるスレッドが GC をトリガーするアロケーションを要求したとき、別のスレッドが正確に何をしているかを知ることはできません。したがって、GC は実質的にいつでも発生し得ます。CLR は別のスレッドの GC 要求に 即座に応答する必要はないため、多少の「余裕」はありますが、実行の すべてのポイントで GC 参照を追跡する必要はないものの、別のスレッドでのアロケーションによって生じる GC の必要性に「タイムリーに」対応できることを保証するのに十分な場所では追跡する 必要があります。

これが意味するのは、CLR は GC ヒープへの すべての参照を ほぼ常に追跡する必要があるということです。GC 参照はマシンレジスタ、ローカル変数、静的フィールド (static)、その他のフィールドに存在する可能性があるため、追跡すべきものは非常に多くあります。これらの場所の中で最も問題なのは、マシンレジスタとローカル変数です。なぜなら、それらはユーザーコードの実際の実行と非常に密接に関連しているからです。実質的に、これが意味するのは、GC 参照を操作する マシンコード には追加の要件があるということです：使用するすべての GC 参照を追跡しなければなりません。これは、参照を追跡するための命令を出力するために、コンパイラに追加の作業が必要であることを意味します。

詳しくは、[ガベージコレクタ設計ドキュメント](#)をご覧ください。

「マネージドコード」の概念

自らのライブな GC 参照のすべてを「ほぼ常時」報告できるように追加のブックキーピング（記帳処理）を行うコードは、マネージドコード（CLR によって「管理されている」ため）と呼ばれます。これを行わないコードは アンマネージドコード と呼ばれます。したがって、CLR 以前のすべてのコードはアンマネージドコードであり、特にすべてのオペレーティングシステムのコードはアンマネージドコードです。

💡 初心者向け補足

マネージドコード (managed code) とアンマネージドコード (unmanaged code) の区別は .NET の重要な概念です。Java では基本的にすべてのコードが JVM によって管理されますが、.NET ではネイティブの C/C++ コードと共存できます。マネージドコードは CLR が GC 参照の追跡やメモリ安全性の保証を行えるコード、アンマネージドコードはそのような管理の外にあるコード (OS の API や C/C++ で書かれたライブラリなど) です。C# で `unsafe` キーワードを使うときは、マネージドコードの安全性の一部を手放してアンマネージドコードに近い操作を行っていることになります。

スタックアンワインドの問題

マネージドコードはオペレーティングシステムのサービスを必要とするため、マネージドコードがアンマネージドコードを呼び出す場合が明らかにあります。同様に、オペレーティングシステムが最初にマネージドコードを開始したため、アンマネージドコードがマネージドコードを呼び出す場合もあります。したがって、一般的に、マネージドプログラムを任意の場所で停止すると、コールスタックにはマネージドコードとアンマネージドコードによって作成されたフレームが混在しています。

アンマネージドコードのスタックフレームには、プログラムの実行以上の要件はありません。特に、実行時にアンワインド（巻き戻し）して呼び出し元を見つけることができるという要件はありません。これが意味するのは、プログラムを任意の場所で停止し、それがアンマネージドメソッド内にあった場合、一般的に^[1]呼び出し元が誰であったかを見つける方法がないということです。シンボル情報 (PDB ファイル) に格納された追加情報があるため、デバッガでのみこれを行えます。この情報は利用可能であることが保証されていません（デバッガで適切なスタックトレースが得られない場合がある理由です）。これはマネージドコードにとって非常に問題です。なぜなら、アンワインドできないスタックには、実際に報告が必要な GC 参照を含むマネージドコードフレームが含まれている可能性があるからです。

マネージドコードには追加の要件があります：実行中に使用するすべての GC 参照を追跡するだけでなく、呼び出し元にアンワインドできる必要があります。さらに、マネージドコードからアンマネージドコードへの遷移（またはその逆）がある場合、マネージドコードはアン

マネージドコードがスタックフレームをアンワインドする方法を知らないことを補うための追加のブックキーピングも行わなければなりません。実質的に、マネージドコードはマネージドフレームを含むスタックの部分をリンクします。したがって、アンマネージドスタックフレームは追加情報なしにはアンワインドできない可能性がありますが、マネージドコードに対応するスタックの塊を見つけ、それらの塊内のマネージドフレームを列挙することは常に可能です。

[1] 最近のプラットフォーム ABI（アプリケーションバイナリインターフェース（Application Binary Interface））はこの情報をエンコードするための規約を定義していますが、通常、すべてのコードがそれに従うという厳密な要件はありません。

マネージドコードの「世界」

結果として、マネージドコードへの遷移とマネージドコードからの遷移のたびに、特別なブックキーピングが必要です。マネージドコードは実質的に、CLR が知らない限り実行が入りきれない独自の「世界」に住んでいます。この 2 つの世界は非常に現実的な意味で互いに異なっています（どの時点でも、コードは マネージドの世界か アンマネージドの世界にいます）。さらに、マネージドコードの実行は CLR フォーマット（[共通中間言語 \(CLI\)](#)）で仕様化されており、CLR がネイティブハードウェアで実行するためにそれを変換するため、CLR はその実行が行うことに対してはるかに多くの制御を持っています。たとえば、CLR はオブジェクトからフィールドを取得することや関数を呼び出すことの意味を変更できます。実際、CLR は MarshalByReference オブジェクトの作成をサポートするために、まさにこれを行っています。これらは通常のローカルオブジェクトのように見えますが、実際には別のマシンに存在している可能性があります。要するに、CLR のマネージドの世界には、後のセクションでより詳しく説明される強力な機能をサポートするために使用できる多数の 実行フックがあるのです。

さらに、マネージドコードのもう一つの重要な影響があり、それほど明白ではないかもしれません。アンマネージドの世界では、GC ポイントは許可されておらず（追跡できないため）、マネージドコードからアンマネージドコードへの遷移にはブックキーピングコストが伴います。これが意味するのは、マネージドコードから任意のアンマネージド関数を呼び出すことは 可能ですが、多くの場合それは快適ではないということです。アンマネージドメソッドは引数や戻り値に GC オブジェクトを使用しないため、それらのアンマネージド関数が作成して使用する「オブジェクト」や「オブジェクトハンドル」は明示的にデアロケート（解放）する必要があります。これは非常に残念なことです。これらの API は例外や継承などの CLR 機能を活用できないため、マネージドコードでインターフェースが設計されていたならどうなるかと比較して、「不一致な」ユーザー体験を持つ傾向があります。

この結果、アンマネージドインターフェースは、マネージドコード開発者に公開される前にほぼ常に ラップされます。たとえば、ファイルにアクセスする際、オペレーティングシステムが提供する Win32 の CreateFile 関数を使うのではなく、この機能をラップしたマネージドの System.IO.File クラスを使います。実際、アンマネージドの機能がユーザーに直接公開されることは極めてまれです。

このラッピングは一見「悪い」ことのように思えるかもしれません（あまり多くのことをしないように見えるコードが増える）が、実際には非常に多くの価値を追加するため、良いことです。アンマネージドインターフェースを直接公開することは常に 可能でした。機能をラップすることを選んだのです。なぜでしょうか？ ランタイムの包括的な目標はプログラミングを簡単にすることであり、通常アンマネージド関数は十分に簡単ではないからです。多くの場合、アンマネージドインターフェースは使いやすさを念頭に置いて設計 されていませんが、むしろ完全性のためにチューニングされています。CreateFile や CreateProcess の引数を見て「簡単」だと言える人はまずいないでしょう。幸いなことに、機能はマネージドの世界に入るときに「フェイスリフト（美容整形）」を受けます。そしてこの変身はしばしば非常に「ローテク」な（名前変更、簡素化、機能の整理以上の複雑なものを必要としない）ものですが、それでも非常に有用です。CLR のために作成された非常に重要なドキュメントの一つが、[Framework Design Guidelines](#) です。この 800 ページ以上のドキュメントは、新しいマネージドクラスライブラリを作成するためのベストプラクティスを詳述しています。

💡 初心者向け補足

.NET の世界では、OS のネイティブ API を直接呼び出す代わりに、[System.IO.File](#) や [System.Net.Http.HttpClient](#) のような使いやすいマネージドクラスを通じて機能にアクセスするのが一般的です。Java でも [java.io.File](#) や [java.net.HttpURLConnection](#) のように、OS の API を直接呼ばずにラッパークラスを使うのと同じ考え方です。ラッピングにより、一貫した命名規則、例外によるエラー処理、GC によるメモリ管理といった恩恵を受けることができます。

したがって、マネージドコード（CLR と密接に関わるコード）がアンマネージドコードと異なる重要な点は 2 つあることが分かりました：

- ハイテク：コードは独自の世界に住んでおり、CLR が非常に微細なレベルで（潜在的に個々の命令に至るまで）プログラム実行のほとんどの側面を制御し、実行がマネージドコードに入りするときに CLR がそれを検出します。これにより、多種多様な有用な機能が実現されます。
- ローテク：マネージドコードからアンマネージドコードへの遷移コストがあること、およびアンマネージドコードが GC オブジェクトを使用できないことが、ほとんどのアンマネージドコードをマネージドのファーサード（外観）でラップする慣行を促進しています。これにより、インターフェースは「フェイスリフト」を受けて簡素化され、一貫した命名規則と設計ガイドラインに準拠することで、アンマネージドの世界にも存在し得たが実際には存在しなかった一貫性と発見しやすさのレベルを生み出します。

両方の特性が、マネージドコードの成功にとって非常に重要です。

メモリ安全性と型安全性

ガベージコレクタが実現する、目立たないが非常に広範囲に影響する機能の一つがメモリ安全性 (memory safety) です。メモリ安全性の不变条件は非常にシンプルです：プログラムがアロケートされた（かつ解放されていない）メモリにのみアクセスする場合、そのプログラムはメモリ安全です。これは単に、ランダムな場所（より正確には、早期に解放されたメモリ）を指す「ワイルドな」（ダングリング）ポインタを持たないことを意味します。明らかに、メモリ安全性はすべてのプログラムが持つべき性質です。ダングリングポインタは常にバグであり、それらを追跡するのはしばしば非常に困難です。

メモリ安全性の保証を提供するには、GC は必要です。

ガベージコレクタがメモリ安全性の保証にどのように役立つかはすぐに分かります。ユーザーがメモリを早期に解放する可能性を除去するからです（したがって、適切にアロケートされていないメモリへのアクセスを防ぎます）。それほど明白でないかもしれないのは、メモリ安全性を保証したい（プログラマがメモリ安全でないプログラムを作成することを不可能にしたい）場合、実質的にガベージコレクタを避けられないということです。その理由は、非自明なプログラムはヒープスタイルの（動的な）メモリアロケーションを必要とし、オブジェクトのライフタイムは本質的に任意のプログラム制御の下にあるためです（高度に制約されたアロケーションプロトコルを持つスタッカ割り当てメモリや静的割り当てメモリとは異なります）。このような制約のない環境では、特定の明示的な delete 文が正しいかどうかを判定する問題は、プログラム分析では予測不可能になります。実質的に、delete が正しいかどうかを判定する唯一の方法は、実行時にそれをチェックすることです。これはまさに GC が行うこと（メモリがまだ生きているかどうかをチェックすること）です。したがって、ヒープスタイルのメモリアロケーションを必要とするプログラムについて、メモリ安全性を保証したいなら、GC が必要なのです。

💡 初心者向け補足

メモリ安全性 (memory safety) と型安全性 (type safety) は、プログラムの信頼性を大幅に向上させる重要な概念です。C/C++ では「ダングリングポインタ」（解放済みメモリへの参照）や「バッファオーバーラン」（配列の範囲外アクセス）といった深刻なバグが発生しやすいですが、.NET の CLR ではこれらを仕組みとして防止しています。Java も同様にメモリ安全なプラットフォームですが、CLR はさらに型安全性（変数が宣言された型と互換性のある操作のみ許可する仕組み）も厳密に保証しています。

GC はメモリ安全性を保証するために必要ですが、十分ではありません。GC はプログラムが配列の末端を超えてインデックスしたり、オブジェクトの末端を超えてフィールドにアクセスしたり（ベースとオフセットの計算を使ってフィールドのアドレスを計算する場合に可能）することを防ぎません。しかし、これらのケースを防ぐことができれば、プログラムがメモリ安全でないプログラムを作成することを本当に不可能にできます。

[共通中間言語 \(CIL\)](#) は任意のメモリを取得・設定できる演算子を持っています（したがってメモリ安全性を侵害し得ます）が、以下のメモリ安全な演算子も持っています、CLR はほとんどのプログラミングでこれらの使用を強く推奨しています：

- フィールド取得演算子 (LDFLD, STFLD, LDFLDA)：名前でフィールドを取得（読み取り）、設定、およびアドレスを取得します。
- 配列取得演算子 (LDELEM, STELEM, LDELETEMA)：インデックスで配列要素を取得、設定、およびアドレスを取得します。すべての配列にはその長さを指定するタグが含まれています。これにより、各アクセスの前に自動的な境界チェックが容易になります。

ユーザーコードでより低レベルの（かつ安全でない）メモリ取得演算子の代わりにこれらの演算子を使用し、他の安全でない [CIL](#) 演算子も避ける（たとえば、任意の場所にジャンプできる演算子、つまり不正な場所にジャンプしかねないもの）ことで、メモリ安全ではあるがそれだけのシステムを構築することができます。しかし、CLR はそうしません。代わりに、CLR はより強い不变条件を強制します：型安全性です。

型安全性については、概念的に各メモリアロケーションは型に関連付けられます。メモリ位置に作用するすべての演算子も、概念的にそれらが有効な型でタグ付けされます。型安全性は、特定の型でタグ付けされたメモリが、その型に許可された操作のみを受けることを要求します。これにより、メモリ安全性（ダングリングポインタなし）が保証されるだけでなく、個々の型に対する追加の保証も可能になります。

これらの型固有の保証の中で最も重要なものの一つは、型に関連付けられたアクセシビリティ（可視性）属性（特にフィールドの可視性）が強制されることです。したがって、フィールドが `private`（その型のメソッドのみがアクセス可能）と宣言されている場合、そのプライバシーはすべての型安全なコードによって確実に尊重されます。たとえば、特定の型がテーブル内の項目数を表す `count` フィールドを宣言しているとします。`count` とテーブルのフィールドが `private` であり、それらを更新するコードだけが両方と一緒に更新すると仮定すれば、`count` とテーブル内の項目数が実際に同期しているという強い保証が（すべての型安全なコードにわたって）存在します。プログラムについて推論する際、プログラマはそれを知っているかどうかにかかわらず、常に型安全性の概念を使用しています。CLR は型安全性を單なるプログラミング言語/コンパイラの慣習から、実行時に厳密に強制できるものへと昇格させます。

検証可能コード - メモリ安全性と型安全性の強制

概念的に、型安全性を強制するには、プログラムが実行するすべての操作が、操作と互換性のある方法で型付けされたメモリ上で動作していることを確認する必要があります。システムはこれをすべて実行時に行うこともできますが、非常に遅くなるでしょう。代わりに、CLR には [CIL](#) 検証 (verification) の概念があり、[CIL](#) に対して（コードが実行される前に）静的解析が行われ、ほとんどの操作が確かに型安全であることを確認します。この静的解析が完全にできない場合にのみ、実行時チェックが必要になります。実際には、必要な実行時チェックの数は非常に少なく、以下の操作が含まれます：

1. 基底型へのポインタを派生型へのポインタにキャストすること（逆方向は静的にチェックできます）
2. 配列の境界チェック（メモリ安全性で見たのと同様）
3. ポインタの配列の要素に新しい（ポインタ）値を代入すること。この特定のチェックは、CLR の配列が自由なキャストルールを持っているため（後述）にのみ必要です。

これらのチェックを行う必要性は、ランタイムに対して要件を課すことに注意してください。特に：

1. GC ヒープ内のすべてのメモリはその型でタグ付けされなければなりません（キャスト演算子を実装できるように）。この型情報は実行時に利用可能でなければならず、キャストが有効かどうかを判定するのに十分なリッチさが必要です（たとえば、ランタイムは継承階層を知っている必要があります）。実際、GC ヒープ上のすべてのオブジェクトの最初のフィールドは、その型を表すランタイムデータ構造を指しています。
2. すべての配列はそのサイズも持っている必要があります（境界チェックのため）。
3. 配列は要素型に関する完全な型情報を持っている必要があります。

幸いなことに、最もコストのかかる要件（各ヒープ項目のタグ付け）は、ガベージコレクションをサポートするためにすでに必要だったもの（GC はすべてのオブジェクト内のどのフィールドにスキャンが必要な参照が含まれているかを知る必要がある）なので、型安全性を提供するための追加コストは低いものです。

したがって、コードの [CIL](#) を検証し、いくつかの実行時チェックを行うことで、CLR は型安全性（およびメモリ安全性）を保証できます。しかしながら、この追加の安全性はプログラミングの柔軟性に代価を課します。CLR には汎用的なメモリ取得演算子がありますが、コードが検証可能であるためには非常に制約された方法でのみ使用できます。特に、すべてのポインタ演算は現在、検証に失敗します。したがって、多くの古典的な C や C++ の慣習は、検証可能コードでは使用できません。代わりに配列を使わなければなりません。これはプログラミングを少し制約しますが、実際にはそれほど悪くなく（配列は非常に強力です）、利点（「厄介な」バグが大幅に減少する）は非常に現実的です。

CLR は検証可能で型安全なコードの使用を強く推奨しています。しかし、(主にアンマネージドコードを扱う際に) 検証不可能なプログラミングが必要な時もあります。CLR はこれを許可しますが、ベストプラクティスは、この安全でないコードをできるだけ限定することです。一般的なプログラムは、安全でないコードが必要な部分はごくわずかで、残りは型安全にできます。

高水準機能

ガベージコレクションをサポートすることは、ランタイムに深遠な影響を与えました。なぜなら、すべてのコードが追加のブックキーピングをサポートする必要があるからです。型安全性への欲求も深遠な影響を与え、プログラムの記述 ([CIL](#)) がフィールドやメソッドに詳細な型情報を持つ高水準なものであることを要求しました。型安全性への欲求はまた、[CIL](#) が型安全である他の高水準プログラミング構造をサポートすることを強制します。これらの構造を型安全な方法で表現するにはランタイムのサポートも必要です。これらの高水準機能の中で最も重要な 2 つは、オブジェクト指向プログラミングの 2 つの本質的な要素をサポートするために使用されます：継承 (inheritance) と仮想呼び出しディスパッチ (virtual call dispatch) です。

オブジェクト指向プログラミング

継承 (inheritance) は機械的な意味では比較的シンプルです。基本的な考え方は、型 `derived` のフィールドが型 `base` のフィールドのスーパーセットであり、`derived` がそのフィールドを `base` のフィールドが最初に来るよう配置する場合、`base` のインスタンスへのポインタを期待するコードには `derived` のインスタンスへのポインタを与えることができ、コードは「そのまま動く」ということです。したがって、型 `derived` は `base` から継承していると言われ、`base` が使えるところならどこでも使用できます。同じコードが多く異なる型に対して使用できるため、コードは ポリモーフィック（多態的）になります。ランタイムはどのような型変換が可能かを知る必要があるため、型安全性を検証できるように継承の指定方法を形式化しなければなりません。

仮想呼び出しディスパッチ (virtual call dispatch) は、継承のポリモーフィズムを一般化します。基底型が派生型によって オーバーライドされるメソッドを宣言できるようにします。型 `base` の変数を使用するコードは、仮想メソッドへの呼び出しが、実行時のオブジェクトの実際の型に基づいて正しいオーバーライドされたメソッドにディスパッチされることを期待できます。このような 実行時ディスパッチロジックは、ランタイムの直接サポートなしにプリミティブな [CIL](#) 命令を使って実装することもできましたが、2 つの重要な欠点がありました：

1. 型安全ではない（ディスパッチテーブルのミスは壊滅的なエラー）
2. 各オブジェクト指向言語がおそらく少しずつ異なる方法で仮想ディスパッチロジックを実装する。結果として、言語間の相互運用性が低下する（ある言語が別の言語で実装された基底型を継承できない）

この理由から、CLR はオブジェクト指向の基本機能に対する直接的なサポートを持っています。可能な限り、CLR はその継承モデルを「言語中立」にしようとしました。つまり、異なる言語が同じ継承階層を共有できるようにしたのです。しかし残念ながら、それが常に可能だったわけではありません。特に、多重継承 (multiple inheritance) は多くの異なる方法で実装できます。CLR はフィールドを持つ型の多重継承をサポートしないを選択しましたが、フィールドを持たないように制約された特殊な型（インターフェース (interface) と呼ばれる）からの多重継承はサポートしています。

💡 初心者向け補足

Java に馴染みのある方にとって、この設計は馴染み深いものでしょう。Java でもクラスの多重継承は許可されず、インターフェースの多重実装のみが許されます。CLR (C#) でも同じアプローチを採用しています。`class MyClass : BaseClass, IInterface1, IInterface2` のように、クラスは 1 つだけ継承し、インターフェースは複数実装できます。これにより、「ダイヤモンド継承問題」（2 つの親クラスから同じメソッドを継承したときの曖昧さ）を回避しつつ、多態的なプログラミングが可能です。

ランタイムがこれらのオブジェクト指向の概念をサポートしている一方で、その使用を要求しているのではないことを念頭に置くことが重要です。継承の概念を持たない言語（たとえば関数型言語）は、これらの機能を単に使用しないだけです。

値型（とボクシング）

オブジェクト指向プログラミングの深遠でありながら繊細な側面は、オブジェクト同一性 (object identity) の概念です：(別々のアロケーション呼び出しによって) アロケートされたオブジェクトが、すべてのフィールド値が同一であっても区別できるという概念です。オブジェクト同一性は、オブジェクトが値ではなく参照（ポインタ）によってアクセスされるという事実と強く関連しています。2つの変数が同じオブジェクトを保持している場合（ポインタが同じメモリを指している）、一方の変数への更新が他方の変数に影響します。

残念ながら、オブジェクト同一性の概念はすべての型に対して良いセマンティックな一致を示すわけではありません。特に、プログラマは一般的に整数をオブジェクトとは考えません。数字の「1」が2つの異なる場所でアロケートされた場合、プログラマは一般的にそれら2つの項目を等しいと見なしたいと思い、一方のインスタンスへの更新が他方に影響することを確実に望みません。実際、「関数型言語」と呼ばれる広範なクラスのプログラミング言語は、オブジェクト同一性と参照セマンティクスを完全に回避しています。

すべてが（整数を含めて）オブジェクトである「純粋な」オブジェクト指向システムを持つことは可能ですが（Smalltalk-80 がこれを行っています）、効率的な実装を得るためにこの均一性を元に戻すための一定量の実装上の「体操」が必要です。他の言語（Perl、Java、JavaScript）は実用的な見方を取り、一部の型（整数など）を値として、他の型を参照として扱います。CLR も混合モデルを選択しましたが、他とは異なり、ユーザー定義の値型 (value type) を許可しました。

値型の主要な特性は以下の通りです：

1. 値型の各ローカル変数、フィールド、または配列要素は、値のデータの個別のコピーを持ちます。
2. ある変数、フィールド、または配列要素が別のものに代入されると、値がコピーされます。
3. 等価性は常にその変数のデータのみで定義されます（その場所ではなく）。
4. 各値型には、1つの暗黙的な無名フィールドのみを持つ対応する参照型もあります。これはボックス化 (boxed) された値と呼ばれまます。ボックス化された値型は継承に参加でき、オブジェクト同一性を持ちます（ただし、ボックス化された値型のオブジェクト同一性の使用は強く推奨されません）。

💡 初心者向け補足

値型 (value type) と参照型 (reference type) の違いは .NET プログラミングの基礎です。`int`、`double`、`bool`、`struct` は値型で、変数に直接データが格納されます。`class`、`string`、配列は参照型で、変数にはヒープ上のオブジェクトへの参照（ポインタ）が格納されます。ボクシング (boxing) とは、値型を参照型に変換する操作です。たとえば `object obj = 42;` と書くと、整数 42 がヒープ上にオブジェクトとしてラップ（ボックス化）されます。Java でも `int` と `Integer` の間で同様のオートボクシング/アンボクシングが行われます。

値型は C (および C++) の `struct` (または C++ の `class`) の概念と非常によく似ています。C と同様に値型へのポインタを持つことができますが、ポインタは `struct` の型とは異なる型です。

例外

CLR が直接サポートするもう一つの高水準プログラミング構造が例外 (exception) です。例外はプログラミング言語の機能で、失敗が発生したポイントでプログラマが任意のオブジェクトをスローできるようにします。オブジェクトがスローされると、ランタイムはコールスタックを検索して、例外をキャッチできると宣言しているメソッドを探します。そのような `catch` 宣言が見つかった場合、そのポイントから実行が続行されます。例外の有用性は、呼び出したメソッドが失敗したかどうかをチェックしないという非常に一般的なミスを回避することです。例外がプログラミングミスの回避に役立つ（つまりプログラミングをより簡単にする）ことを考えると、CLR がそれらをサポートするのは驚くべきことではありません。

余談ですが、例外は一般的なエラー（失敗のチェック漏れ）を回避しますが、別のエラー（失敗時にデータ構造を一貫した状態に戻すこと）を防ぐことはできません。これは、例外がキャッチされた後、実行を続行しても追加のエラー（最初の失敗によって引き起こされたもの）が発生しないかどうかを一般的に知ることが困難であることを意味します。これは CLR が将来価値を追加する可能性がある分野です。しかし、現在の実装においても、例外は大きな前進です（さらに進む必要があるだけです）。

パラメータ化された型（ジェネリクス）

CLR のバージョン 2.0 以前は、唯一のパラメータ化された型は配列でした。その他すべてのコンテナ（ハッシュテーブル、リスト、キューなど）は、汎用的な Object 型で操作していました。`List<ElemT>` や `Dictionary<KeyT, ValueT>` を作成できることは、値型がコレクションに入る際にボックス化される必要があり、要素の取得時には明示的なキャストが必要であるため、確かにパフォーマンスに悪影響を及ぼしました。しかし、パラメータ化された型を CLR に追加する最も重要な理由はそれではありません。主な理由は、パラメータ化された型がプログラミングをより簡単にすることです。

💡 初心者向け補足

ジェネリクス (Generics) は、型をパラメータとして受け取るクラスやメソッドを定義する機能です。Java のジェネリクス (`List<String>`, `Map<String, Integer>`) と同じ概念ですが、実装方法が異なります。Java のジェネリクスは「型消去 (type erasure)」という方式で、コンパイル後に型パラメータ情報が消えますが、CLR のジェネリクスは「具象化 (reification)」という方式で、実行時にも型パラメータ情報が保持されます。そのため、CLR ではジェネリクスの実行時パフォーマンスが向上し、`List<int>` のようにボクシングなしで値型を直接格納できるという利点があります。

その理由は繊細です。その効果を理解する最も簡単な方法は、すべての型が汎用的な Object 型に置き換えた場合にクラスライブラリがどのように見えるかを想像することです。この効果は、JavaScript のような動的型付け言語で起こることと似ています。そのような世界では、プログラマが不正な（しかし型安全な）プログラムを作成する方法がはるかに多くなります。そのメソッドのパラメータはリストであるべきですか？文字列ですか？整数ですか？上記のいずれかですか？メソッドのシグネチャを見るだけではもはや明白ではありません。さらに悪いことに、メソッドが Object を返す場合、他のどのメソッドがそれをパラメータとして受け入れるでしょうか？一般的なフレームワークには数百のメソッドがあります。すべてが Object 型のパラメータを取る場合、メソッドが実行する操作に対してどの Object インスタンスが有効かを判定することが非常に困難になります。要するに、強い型付けはプログラマがその意図をより明確に表現するのを助け、ツール（例：コンパイラ）がその意図を強制できるようにします。これにより生産性が大幅に向上します。

これらの利点は、型が List や Dictionary に入れられたからといって消えないため、明らかにパラメータ化された型には価値があります。唯一の本当の問題は、パラメータ化された型が、CIL が生成される時点で「コンパイルアウト」される言語固有の機能として考えるのが最適か、この機能がランタイムでファーストクラスのサポートを持つべきかということです。どちらの実装も確かに可能です。CLR チームはファーストクラスのサポートを選択しました。なぜなら、それなしでは、パラメータ化された型は異なる言語によって異なる方法で実装されることになるからです。これは、相互運用性がせいぜい面倒になることを意味します。さらに、パラメータ化された型についてプログラマの意図を表現することは、クラスライブラリの インターフェースで最も価値があります。CLR が公式にパラメータ化された型をサポートしていないければ、クラスライブラリはそれらを使えず、重要なユーザビリティ機能が失われてしまいます。

データとしてのプログラム（リフレクション API）

CLR の基礎はガベージコレクション、型安全性、高水準言語機能です。これらの基本的な特性により、プログラムの仕様 (CIL) はかなり高水準であることを余儀なくされました。このデータが実行時に存在するようになると (C や C++ のプログラムでは真ではないこと)、このリッチなデータをエンドプログラマにも公開することに価値があることが明らかになりました。このアイデアの結果、System.Reflection インターフェース群が作成されました（プログラムが自分自身を見る（反映する、リフレクトする）ことを可能にするため、そのように名付けられました）。このインターフェースにより、プログラムのほぼすべての側面を探索できます（どのような型があるか、継承関係、どのようなメソッドやフィールドが存在するかなど）。実際、失われる情報は非常に少ないため、マネージドコードに対する非常に優れた「デコンパイラ」が可能です（例：[NET Reflector](#)）。知的財産権の保護を懸念する人々はこの機能に愕然としますが（これは 難読化 と呼ばれる操作で意図的に情報を破壊することで対処できます）、これが可能であるという事実は、マネージドコードで実行時に利用可能な情報の豊富さの証です。

実行時にプログラムを単に検査するだけでなく、プログラムに対して操作を実行することも可能です（例：メソッドの呼び出し、フィールドの設定など）。そして、おそらく最も強力なのは、実行時にスクラッチからコードを生成すること（System.Reflection.Emit）です。実際、ランタイムライブラリはこの機能を、文字列照合のための特殊なコードの作成（System.Text.RegularExpressions）や、オブジェクトをファイルに保存したりネットワーク経由で送信するための「シリアル化」コードの生成に使用しています。このような機能は以前は単に実現不可能でした（コンパイラを書かなければならなかったでしょう！）が、ランタイムのおかげで、はるかに多くのプログラミング問題の手の届く範囲にあります。

リフレクション機能は確かに強力ですが、その力は注意して使う必要があります。リフレクションは通常、静的にコンパイルされた対応するものよりも大幅に遅くなります。より重要なことに、自己参照システムは本質的に理解が困難です。これは、Reflection や Reflection.Emit のような強力な機能は、その価値が明確で実質的な場合にのみ使用すべきであることを意味します。

その他の機能

ランタイム機能の最後のグループは、CLR の基本的なアーキテクチャ（GC、型安全性、高水準仕様）とは関係ないが、完全なランタイムシステムの重要なニーズを満たす機能です。

アンマネージドコードとの相互運用

マネージドコードは、アンマネージドコードで実装された機能を使えなければなりません。相互運用には 2 つの主要な「種類」があります。1 つ目は、単にアンマネージド関数を呼び出す機能です（これは Platform Invoke または P/Invoke と呼ばれます）。アンマネージドコードにも COM（コンポーネントオブジェクトモデル（Component Object Model））と呼ばれるオブジェクト指向の相互運用モデルがあり、アドホックなメソッド呼び出しそりも構造化されています。COM と CLR の両方がオブジェクトやその他の規約（エラーの処理方法、オブジェクトのライフタイムなど）のモデルを持っているため、特別なサポートがある方が、CLR は COM コードとの相互運用をよりうまく行えます。

事前コンパイル (Ahead of Time Compilation)

CLR モデルでは、マネージドコードはネイティブコードではなく CIL として配布されます。ネイティブコードへの変換は実行時に行われます。最適化として、CIL から生成されたネイティブコードは、crossgen と呼ばれるツール（.NET Framework の NGEN ツールに似ています）を使用してファイルに保存できます。これにより、実行時の大量のコンパイル時間が回避され、クラスライブラリが非常に大きいため、非常に重要です。

スレッディング

CLR は、マネージドコードでマルチスレッドプログラムをサポートする必要性を完全に予見していました。当初から、CLR ライブラリには System.Threading.Thread クラスが含まれており、これはオペレーティングシステムの実行スレッドの概念の 1 対 1 のラッパーです。しかし、OS スレッドのラッパーに過ぎないため、System.Threading.Thread の作成は比較的コストが高く（開始に数ミリ秒かかります）。これは多くの操作には問題ありませんが、あるスタイルのプログラミングでは非常に小さい作業項目（数十ミリ秒しかかかるない）を作成します。これはサーバーコード（たとえば、各タスクが 1 つの Web ページを提供するだけ）や、マルチプロセッサを活用しよ

うとするコード（たとえば、マルチコアソートアルゴリズム）で非常に一般的です。これをサポートするため、CLR にはスレッドプール（ThreadPool）という概念があり、ワークアイテムをキューに入れることができます。このスキームでは、CLR が作業を行うために必要なスレッドを作成する責任を負います。CLR は ThreadPool を System.Threading.Threadpool クラスとして直接公開していますが、推奨されるメカニズムは [Task Parallel Library](#) の使用です。これは、非常に一般的な形式の並行性制御に対する追加のサポートを提供します。

実装の観点から、ThreadPool の重要なイノベーションは、作業をディスパッチするために最適な数のスレッドが使用されることを保証する責任を負っていることです。CLR は、スループットレートとスレッド数を監視し、スループットを最大化するためにスレッド数を調整するフィードバックシステムを使ってこれを行います。これは非常に優れています。プログラマは主に「並列性の公開」（つまり、ワークアイテムの作成）の観点で考えることができ、（ワーカーロードとプログラムが実行されるハードウェアに依存する）適切な並列度を決定するというより繊細な問題を考える必要がなくなるからです。

まとめとリソース

ランタイムは実に多くのことを行います！内部の詳細について話し始めることもなく、ランタイムの機能の一部を説明するだけで多くのページを費やしました。しかし、この入門がその内部の詳細をより深く理解するための有用なフレームワークを提供することを願っています。このフレームワークの基本的な概要は以下の通りです：

- ランタイムはプログラミング言語をサポートするための完全なフレームワークです。
 - ランタイムの目標はプログラミングを簡単にすることです。
 - ランタイムの基本的な機能は以下の通りです：
 - ガベージコレクション (Garbage Collection)
 - メモリ安全性と型安全性 (Memory and Type Safety)
 - 高水準言語機能のサポート (Support for High-Level Language Features)
-

便利なリンク

- [CLR の MSDN エントリ](#)
- [CLR の Wikipedia エントリ](#)
- [共通言語基盤 \(CLI\) の ECMA 標準](#)
- [.NET Framework 設計ガイドライン](#)
- [CoreCLR リポジトリのドキュメント](#)

ガベージコレクション

原文

この章の原文は [Garbage Collection Design](#) です。

著者: Maoni Stephens ([@maoni0](#)) - 2015

注: ガベージコレクション全般については *The Garbage Collection Handbook* を、CLR GC に特化した知識については *Pro .NET Memory Management* を参照してください。いずれも本ドキュメント末尾のリソースセクションで紹介しています。

💡 初心者向け補足

ガベージコレクション (GC) とは、プログラムが使用しなくなったメモリを自動的に回収する仕組みです。C/C++ ではプログラマが手動でメモリを解放しますが、.NET では GC がこれを自動的に行うため、メモリリークやダンギングポインタなどの問題を大幅に減らすことができます。

コンポーネントアーキテクチャ

GC に属する 2 つのコンポーネントは、アロケータ (allocator) とコレクタ (collector) です。アロケータはメモリの取得を担当し、適切なタイミングでコレクタをトリガーします。コレクタはガベージ、すなわちプログラムで使用されなくなったオブジェクトのメモリを回収します。

コレクタが呼び出される方法は他にもあります。たとえば、`GC.Collect` を手動で呼び出す方法や、ファイナライザスレッドがメモリ不足の非同期通知を受け取った場合（これによりコレクタがトリガーされます）などです。

アロケータの設計

アロケータは、実行エンジン (Execution Engine, EE) のアロケーションヘルパーから以下の情報とともに呼び出されます：

- 要求されたサイズ
- スレッドのアロケーションコンテキスト
- ファイナライズ可能なオブジェクトかどうかなどを示すフラグ

GC はオブジェクトの種類によって特別な扱いをしません。オブジェクトのサイズを取得するために EE に問い合わせます。

サイズに基づいて、GC はオブジェクトを 2 つのカテゴリに分類します：小さなオブジェクト (85,000 バイト未満) と大きなオブジェクト (85,000 バイト以上) です。原則として、小さなオブジェクトと大きなオブジェクトは同じように扱うことができますが、大きなオブジェクトのコンパクションはコストが高いため、GC はこの区別を行います。

💡 初心者向け補足

85,000 バイト (約 85KB) という閾値は、.NET GC がオブジェクトを「小さなオブジェクトヒープ (SOH: Small Object Heap)」と「大きなオブジェクトヒープ (LOH: Large Object Heap)」のどちらに配置するかを決める境界です。大きなオブジェクトはメモリ内で移動 (コンパクション) するコストが非常に高いため、別のヒープで管理されます。

GC がアロケータにメモリを渡す際、アロケーションコンテキスト (**allocation context**) の単位で行います。アロケーションコンテキストのサイズはアロケーションクォンタム (**allocation quantum**) によって定義されます。

- アロケーションコンテキストは、特定のヒープセグメントの中の小さな領域で、それぞれ特定のスレッド専用として割り当てられます。シングルプロセッサ (論理プロセッサが 1 つ) のマシンでは、単一のコンテキストが使用され、これが第 0 世代 (gen0) のアロケーションコンテキストとなります。
- アロケーションクォンタムは、アロケータがアロケーションコンテキスト内でオブジェクトの割り当てを行うためにメモリを追加で必要とするたびに確保するメモリのサイズです。通常は 8K バイトで、マネージドオブジェクトの平均サイズは約 35 バイトであるため、1 つのアロケーションクォンタムで多数のオブジェクトの割り当てが可能です。

大きなオブジェクトはアロケーションコンテキストやクォンタムを使用しません。1 つの大きなオブジェクト自体が、これらの小さなメモリ領域よりも大きくなります。また、これらの領域の利点 (後述) は小さなオブジェクトに特有のものです。大きなオブジェクトはヒープセグメントに直接割り当てられます。

アロケータは以下の目標を達成するように設計されています：

- 適切なタイミングでの **GC** トリガー: アロケータは、アロケーション予算 (コレクタが設定した閾値) を超えた場合、または特定のセグメントにそれ以上割り当てられなくなった場合に GC をトリガーします。アロケーション予算とマネージドセグメントについては後述します。
- オブジェクトの局所性の維持: 同じヒープセグメント上で一緒に割り当てられたオブジェクトは、互いに近い仮想アドレスに格納されます。
- 効率的なキャッシュ利用: アロケータはオブジェクト単位ではなく、アロケーションクォンタム 単位でメモリを割り当てます。そのメモリはゼロクリアされ、CPU キャッシュをウォームアップします。これは、そのメモリにすぐにオブジェクトが割り当てられるためです。アロケーションクォンタムは通常 8K バイトです。
- 効率的なロック: アロケーションコンテキストとクォンタムのスレッドアフィニティ (スレッド親和性) により、特定のアロケーションクォンタムに書き込むスレッドは常に 1 つだけであることが保証されます。その結果、現在のアロケーションコンテキストが使い果たされている限り、オブジェクトの割り当てにロックは不要です。
- メモリの整合性: GC は新しく割り当てられたオブジェクトのメモリを常にゼロクリアし、オブジェクト参照がランダムなメモリを指すことを防ぎます。
- ヒープのクローラビリティの維持: アロケータは各アロケーションクォンタムの残りのメモリをフリーオブジェクトにします。たとえば、アロケーションクォンタムに 30 バイト残っていて、次のオブジェクトが 40 バイトの場合、アロケータは 30 バイトをフリーオブジェクトにし、新しいアロケーションクォンタムを取得します。

💡 初心者向け補足

「スレッドアフィニティ」とは、各スレッドが専用のメモリ領域 (アロケーションコンテキスト) を持つ仕組みです。これにより、複数のスレッドが同時にオブジェクトを割り当てても、ロック (排他制御) なしで安全に動作できます。マルチスレッド環境での高いパフォーマンスの鍵となっています。

アロケーション API

```
Object* GCHeap::Alloc(size_t size, DWORD flags);
```

```
Object* GCHeap::Alloc(alloc_context* acontext, size_t size, DWORD flags);
```

上記の関数は小さなオブジェクトと大きなオブジェクトの両方の割り当てに使用できます。LOH (Large Object Heap) に直接割り当てる関数もあります：

```
Object* GCHeap::AllocLHeap(size_t size, DWORD flags);
```

コレクタの設計

GC の目標

GC は、メモリを極めて効率的に管理し、「マネージドコード」を書く人がほとんど意識する必要がないことを目指しています。効率的とは以下を意味します：

- GC は十分な頻度で発生し、マネージドヒープに大量の未使用のまま割り当てられたオブジェクト (ガベージ) が含まれることを避け、不必要的メモリの使用を防ぐべきです。
- GC は可能な限り低い頻度で発生し、有用な CPU 時間の消費を避けるべきです。ただし、頻繁な GC はメモリ使用量の削減につながります。
- GC は生産的であるべきです。GC が少量のメモリしか回収しない場合、GC (およびそれに伴う CPU サイクル) は無駄になります。
- 個々の GC は高速であるべきです。多くのワークロードには低レイテンシの要件があります。
- マネージドコードの開発者は、GC について多くを知らなくても良好なメモリ利用率 (ワークロードに対して相対的に) を達成できるべきです。
- GC は、異なるメモリ使用パターンを満たすように自己調整するべきです。

💡 初心者向け補足

GC の設計はトレードオフの連続です。GC を頻繁に実行すればメモリの無駄は減りますが、CPU 時間を消費します。逆に GC を減らせば CPU は節約できますが、メモリの無駄が増えます。CLR の GC はこのバランスを自動的に調整し、アプリケーションの特性に応じた最適な動作を目指しています。

マネージドヒープの論理的な表現

CLR GC は世代別コレクタ (**generational collector**) です。これは、オブジェクトが論理的に世代 (generation) に分類されることを意味します。世代 N が収集されると、生き残ったオブジェクトは世代 $N+1$ に属するものとしてマークされます。このプロセスはプロモーション (**promotion**) と呼ばれます。降格 (**demote**) する場合やプロモーションしない場合の例外もあります。

小さなオブジェクトの場合、ヒープは 3 つの世代に分けられます：**gen0**、**gen1**、**gen2** です。大きなオブジェクトの場合は 1 つの世代 (**gen3**) があります。**gen0** と **gen1** はエフェメラル世代 (**ephemeral generation**) と呼ばれます (短期間しか存在しないオブジェクト用)。

初心者向け補足

世代別 GC の基本的な考え方は「ほとんどのオブジェクトは短命である」という経験則(世代仮説)に基づいています。新しいオブジェクトを gen0 に配置し、GC を生き延びるたびに上の世代にプロモーションします。若い世代を頻繁に収集し、古い世代はまれにしか収集しないことで、効率的な GC を実現しています。

小さなオブジェクトヒープでは、世代番号は年齢を表します。gen0 が最も若い世代です。ただし、gen0 のすべてのオブジェクトが gen1 や gen2 のオブジェクトよりも若いとは限りません。例外があり、以下で説明します。ある世代を収集するとは、その世代とそのすべての若い世代のオブジェクトを収集することを意味します。

原則として大きなオブジェクトも小さなオブジェクトと同じように扱えますが、大きなオブジェクトのコンパクションは非常にコストが高いため、異なる扱いを受けます。大きなオブジェクトの世代は 1 つだけで、パフォーマンス上の理由から常に gen2 の収集と一緒に収集されます。gen2 と gen3 はどちらも大きくなりえるため、エフェメラル世代(gen0 と gen1)の収集は限られたコストで行う必要があります。

割り当ては常に最も若い世代に対して行われます。小さなオブジェクトの場合は常に gen0 に、大きなオブジェクトの場合は唯一の世代である gen3 に割り当てられます。

マネージドヒープの物理的な表現

マネージドヒープはマネージドヒープセグメントの集合です。ヒープセグメントとは、GC が OS から取得する連続したメモリブロックです。ヒープセグメントには、含まれる内容に応じて、小さなオブジェクト用、大きなオブジェクト用、ピン留めオブジェクト用のセグメントがあります。各ヒープ上でヒープセグメントはチェーンでつながっています。少なくとも 1 つの小さなオブジェクト用セグメントと 1 つの大きなオブジェクト用セグメントがあり、CLR のロード時に確保されます。また、読み取り専用(ro)セグメントを含む NonGC ヒープもあります。

各小さなオブジェクトヒープには常に 1 つだけエフェメラルセグメントがあり、gen0 と gen1 はここに存在します。このセグメントには gen2 のオブジェクトが含まれる場合も、含まれない場合もあります。エフェメラルセグメントに加えて、0 個以上の追加セグメントがあり得ますが、これらは gen2 オブジェクトのみを含むため gen2 セグメントとなります。

大きなオブジェクトヒープには 1 つ以上のセグメントがあります。

ヒープセグメントはアドレスの低い方から高い方へ消費されます。つまり、セグメント上のアドレスが低いオブジェクトは、アドレスが高いオブジェクトよりも古いくことになります。ここにも後述する例外があります。

初心者向け補足

ヒープセグメントは、GC が OS から取得する実際のメモリの塊です。論理的な「世代」の概念とは異なり、物理的なメモリレイアウトを表しています。エフェメラルセグメントは特に重要で、新しいオブジェクトの割り当てが行われる場所です。

ヒープセグメントは必要に応じて取得されます。ライブラリオブジェクトが含まれなくなると削除されますが、ヒープ上の初期セグメントは常に存在し続けます。各ヒープでは一度に 1 つのセグメントが取得されます。小さなオブジェクトの場合は GC 中に、大きなオブジェクトの場合は割り当て時に行われます。この設計により、大きなオブジェクトは gen2 の収集(比較的コストが高い)でのみ収集されるため、パフォーマンスが向上します。

ヒープセグメントは取得された順にチェーンでつながれます。チェーンの最後のセグメントが常にエフェメラルセグメントです。収集済みのセグメント(ライブラリオブジェクトなし)は、削除される代わりに再利用でき、新しいエフェメラルセグメントになります。セグメントの再利用は小さなオブジェクトヒープでのみ実装されています。大きなオブジェクトが割り当てられるたびに、大きなオブジェクトヒープ全体が考慮されます。小さなオブジェクトの割り当てではエフェメラルセグメントのみが考慮されます。

アロケーション予算

アロケーション予算 (allocation budget) は、各世代に関連付けられた論理的な概念です。超過すると、その世代の GC がトリガーされるサイズの上限です。

予算は、主にその世代の生存率 (survival rate) に基づいて設定されます。生存率が高い場合、次の GC でデッドオブジェクトとライブオブジェクトの比率がより良くなることを期待して、予算は大きく設定されます。

どの世代を収集するかの決定

GC がトリガーされると、GC はまずどの世代を収集するかを決定する必要があります。アロケーション予算以外にも、以下の要因を考慮する必要があります：

- 世代のフラグメンテーション: 世代のフラグメンテーション (断片化) が高い場合、その世代を収集すると生産的になる可能性が高い。
- マシンのメモリ負荷: メモリ負荷が高すぎる場合、GC は空きスペースが得られる可能性があればより積極的に収集します。これは (マシン全体での) 不必要なページングを防ぐために重要です。
- エフェメラルセグメントの空き容量: エフェメラルセグメントの空きが少なくなっている場合、GC は新しいヒープセグメントの取得を避けるために、より積極的なエフェメラル収集 (gen1 をより多く実行) を行うことがあります。

GC のフロー

マークフェーズ (Mark Phase)

マークフェーズの目標は、すべてのライブオブジェクト (生きているオブジェクト) を見つけることです。

世代別コレクタの利点は、すべてのオブジェクトを常に調べるのではなく、ヒープの一部だけを収集できることです。エフェメラル世代を収集する際、GC はこれらの世代のどのオブジェクトがライブであるかを特定する必要があります。この情報は EE によって報告されます。EE によってライブに保たれるオブジェクトに加えて、古い世代のオブジェクトも、若い世代のオブジェクトへの参照を保持することで、それらをライブに保つことができます。

GC は古い世代のマーキングにカード (card) を使用します。カードは代入操作時に JIT ヘルパーによって設定されます。JIT ヘルパーは、エフェメラル範囲内のオブジェクトを検出すると、ソースの場所を表すカードを含むバイトを設定します。エフェメラル収集時、GC はヒープの残りの部分の設定されたカードを確認し、それらのカードに対応するオブジェクトのみを調べることができます。

💡 初心者向け補足

「カード」とは、古い世代から若い世代への参照を効率的に追跡するための仕組みです。すべてのオブジェクトを毎回スキャンする代わりに、参照が変更された領域だけをチェックすることで、GC のパフォーマンスを向上させています。これは「ライトバリア (write barrier)」とも関連する技術です。

プランフェーズ (Plan Phase)

プランフェーズは、コンパクションをシミュレーションして効果的な結果を判断します。コンパクションが生産的であれば GC は実際のコンパクションを開始し、そうでなければスイープを行います。

💡 初心者向け補足

「コンパクション (compaction)」はオブジェクトをメモリ内で移動させて隙間を埋める操作、「スイープ (sweep)」は空き領域をフリーリストに登録するだけの操作です。コンパクションはフラグメンテーションを解消しますが、オブジェクトの移動にコストがかかります。プランフェーズではどちらが効率的かを判断しています。

リロケートフェーズ (Relocate Phase)

GC がコンパクションを行うと決定した場合、オブジェクトが移動するため、それらのオブジェクトへの参照を更新する必要があります。リロケートフェーズは、収集対象の世代にあるオブジェクトを指すすべての参照を見つける必要があります。これはマークフェーズとは対照的で、マークフェーズはライボオブジェクトのみを対象とするため、弱い参照 (weak reference) を考慮する必要がありません。

コンパクトフェーズ (Compact Phase)

このフェーズは非常に直接的です。プランフェーズがオブジェクトの移動先の新しいアドレスをすでに計算しているため、コンパクトフェーズはオブジェクトをそこにコピーします。

スイープフェーズ (Sweep Phase)

スイープフェーズは、ライボオブジェクト間のデッドスペース (空き領域) を探します。これらのデッドスペースにフリーオブジェクトを作成します。隣接するデッドオブジェクトは1つのフリーオブジェクトにまとめられます。これらのフリーオブジェクトはすべてフリーリスト (freelist) に配置されます。

コードフロー

用語：

- **WKS GC:** ワークステーション GC (Workstation GC)
- **SVR GC:** サーバー GC (Server GC)

機能的な動作

WKS GC (コンカレント GC 無効)

1. ユーザースレッドがアロケーション予算を使い果たし、GC をトリガーします。
2. GC は `SuspendEE` を呼び出してマネージドスレッドを一時停止します。
3. GC はどの世代を対象とするか (condemn) を決定します。
4. マークフェーズを実行します。
5. プランフェーズを実行し、コンパクション GC を行うかどうかを決定します。
6. コンパクションを行う場合はリロケートフェーズとコンパクトフェーズを実行し、行わない場合はスイープフェーズを実行します。

7. GC は `RestartEE` を呼び出してマネージドスレッドを再開します。

8. ユーザースレッドが実行を再開します。

💡 初心者向け補足

`SuspendEE` はすべてのマネージドスレッドを一時停止させる処理で、`RestartEE` はそれらを再開させる処理です。GC 中にオブジェクトの移動が行われるため、アプリケーションのスレッドが同時にそのオブジェクトにアクセスすると問題が起きます。このため、GC 中は「Stop-the-World」(世界を止める) と呼ばれるスレッドの一時停止が行われます。

WKS GC (コンカレント GC 有効)

以下は、バックグラウンド GC がどのように実行されるかを示しています。

1. ユーザースレッドがアロケーション予算を使い果たし、GC をトリガーします。
2. GC は `SuspendEE` を呼び出してマネージドスレッドを一時停止します。
3. GC はバックグラウンド GC を実行するかどうかを決定します。
4. バックグラウンド GC を実行する場合、バックグラウンド GC スレッドが起動されてバックグラウンド GC を実行します。バックグラウンド GC スレッドは `RestartEE` を呼び出してマネージドスレッドを再開します。
5. マネージドスレッドはバックグラウンド GC が作業を行っている間も割り当てを続けます。
6. ユーザースレッドがアロケーション予算を使い果たし、エフェメラル GC (フォアグラウンド GC と呼ばれます) をトリガーすることができます。これは「WKS GC (コンカレント GC 無効)」と同じ方法で実行されます。
7. バックグラウンド GC は `SuspendEE` を再度呼び出してマーキングを完了し、その後 `RestartEE` を呼び出してユーザースレッドの実行中にコンカレントスイープフェーズを開始します。
8. バックグラウンド GC が完了します。

💡 初心者向け補足

コンカレント GC (バックグラウンド GC) は、アプリケーションの応答性を向上させるための仕組みです。gen2 の収集のような時間のかかる GC を、専用のバックグラウンドスレッドで実行し、アプリケーションのスレッドはほとんど止まることなく動作を続けることができます。ただし、その間もエフェメラル世代の GC (フォアグラウンド GC) は必要に応じて実行されます。

SVR GC (コンカレント GC 無効)

1. ユーザースレッドがアロケーション予算を使い果たし、GC をトリガーします。
2. サーバー GC スレッドが起動され、`SuspendEE` を呼び出してマネージドスレッドを一時停止します。
3. サーバー GC スレッドが GC 作業を実行します (コンカレント GC 無効のワークステーション GC と同じフェーズ)。
4. サーバー GC スレッドが `RestartEE` を呼び出してマネージドスレッドを再開します。
5. ユーザースレッドが実行を再開します。

💡 初心者向け補足

サーバー GC (SVR GC) は、サーバーアプリケーション向けに最適化された GC モードです。CPU の各論理プロセッサに専用の GC スレッドとヒープが割り当てられ、GC 作業が並列に実行されます。これにより、スループットの高いサーバーワークロードで

高いパフォーマンスが得られます。一方、ワークステーション GC (WKS GC) はデスクトップアプリケーション向けで、単一スレッドで GC を実行します。

SVR GC (コンカレント GC 有効)

このシナリオは WKS GC (コンカレント GC 有効) と同じですが、バックグラウンドでない GC がサーバー GC スレッド上で実行される点が異なります。

物理アーキテクチャ

このセクションは、コードのフローを追うための参考です。

ユーザースレッドはアロケーションコンテキストのスペースを使い果たし、[try_allocate_more_space](#) を介して新しいコンテキストを取得します。

[try_allocate_more_space](#) は、GC をトリガーする必要があるときに [GarbageCollectGeneration](#) を呼び出します。

WKS GC (コンカレント GC 無効) のコードフロー

WKS GC (コンカレント GC 無効) の場合、[GarbageCollectGeneration](#) は GC をトリガーしたユーザースレッド上ですべて実行されます。コードフローは以下のとおりです：

```
GarbageCollectGeneration()
{
    SuspendEE();
    garbage_collect();
    RestartEE();
}

garbage_collect()
{
    generation_to_condemn();
    gc1();
}

gc1()
{
    mark_phase();
    plan_phase();
}

plan_phase()
{
    // 実際のプランフェーズの処理で
    // コンパクションするかどうかを決定
    if (compact)
    {
        relocate_phase();
        compact_phase();
    }
}
```

```

    }
else
    make_free_lists();
}

```

WKS GC (コンカレント GC 有効) のコードフロー

WKS GC (コンカレント GC 有効、デフォルト) の場合、バックグラウンド GC のコードフローは以下のとおりです：

cpp

```

GarbageCollectGeneration()
{
    SuspendEE();
    garbage_collect();
    RestartEE();
}

garbage_collect()
{
    generation_to_condemn();
    // バックグラウンド GC を実行することを決定
    // バックグラウンド GC スレッドを起動して作業を実行
    do_background_gc();
}

do_background_gc()
{
    init_background_gc();
    start_c_gc();

    // BGC によって再開されるまで待機
    wait_to_proceed();
}

bgc_thread_function()
{
    while (1)
    {
        // イベントを待機
        // 起動
        gc1();
    }
}

gc1()
{
    background_mark_phase();
    background_sweep();
}

```

リソース

- [.NET CLR GC 実装 \(ソースコード\)](#)
- [The Garbage Collection Handbook: The Art of Automatic Memory Management](#)
- [ガベージコレクション \(Wikipedia\)](#)
- [Pro .NET Memory Management](#)
- [.NET GC Internals 動画シリーズ](#)

スレッディング

原文

この章の原文は [Threading](#) です。

マネージドスレッドとネイティブスレッド

マネージドコードは「マネージドスレッド」上で実行されます。マネージドスレッドは、オペレーティングシステムが提供するネイティブスレッドとは異なるものです。ネイティブスレッドは物理マシン上でネイティブコードを実行するスレッドですが、マネージドスレッドはCLRの仮想マシン上で実行される仮想的なスレッドです。

JITコンパイラが「仮想的な」IL命令を物理マシン上で実行されるネイティブ命令にマッピングするのと同様に、CLRのスレッディング基盤は「仮想的な」マネージドスレッドをオペレーティングシステムが提供するネイティブスレッドにマッピングします。

初心者向け補足

マネージドスレッドとネイティブスレッドの関係は、ILコードとネイティブコードの関係に似ています。CLRはOSのスレッドを直接公開せず、独自の抽象化されたスレッドモデルを提供します。これにより、異なるOS間での移植性やスレッド管理の柔軟性が向上します。

任意の時点において、マネージドスレッドは実行のためにネイティブスレッドに割り当てられている場合もあれば、割り当てられていない場合もあります。たとえば、`new System.Threading.Thread` で作成されたがまだ `System.Threading.Thread.Start` で開始されていないマネージドスレッドは、ネイティブスレッドに割り当てられていないマネージドスレッドです。同様に、原理的にはマネージドスレッドは実行の過程で複数のネイティブスレッド間を移動することも可能ですが、現在のCLRの実装ではこれはサポートされていません。

マネージドコードに公開される Thread のパブリックインターフェースは、意図的に基盤となるネイティブスレッドの詳細を隠蔽しています。その理由は以下の通りです：

- マネージドスレッドは必ずしも単一のネイティブスレッドにマッピングされるわけではない（そもそもネイティブスレッドにマッピングされていない場合もある）
- オペレーティングシステムによってネイティブスレッドの抽象化が異なる
- 原則として、マネージドスレッドは「仮想化」されている

CLRはマネージドスレッドに対して同等の抽象化を提供し、CLR自身がこれを実装しています。たとえば、OSのスレッドローカルストレージ(TLS)メカニズムは公開せず、代わりにマネージドの「スレッド静的(thread-static)」変数を提供しています。同様に、ネイティブスレッドの「スレッドID」は公開せず、OSとは独立に生成される「マネージドスレッドID」を提供しています。ただし、診断目的で `System.Diagnostics` 名前空間のタイプを通じて基盤となるネイティブスレッドの一部の詳細を取得することは可能です。

マネージドスレッドには、ネイティブスレッドでは通常不要な追加機能が必要です。第一に、マネージドスレッドはスタック上にGC参照を保持するため、GCが発生するたびにCLRはこれらの参照を列挙（場合によっては変更）できる必要があります。そのため、CLRは各マネージドスレッドを「サスペンド」（すべてのGC参照が特定可能な地点でスレッドを停止）する必要があります。第二に、AppDomainがアンロードされる際、CLRはそのAppDomain内でコードを実行しているスレッドがないことを保証する必要があります。これには、スレッドを強制的にそのAppDomainから巻き戻す機能が必要であり、CLRはそのようなスレッドに `ThreadAbortException` を注入することでこれを実現します。

初心者向け補足

GC 参照の列挙とは、スレッドのスタックやレジスタ上にあるマネージドオブジェクトへの参照をすべて見つけ出すことです。GC はこの情報を使って、どのオブジェクトがまだ使用中でどのオブジェクトが回収可能かを判断します。スレッドが動いたままだと参照が変化してしまうため、安全な地点でスレッドを一時停止する必要があるのです。

データ構造

すべてのマネージドスレッドには、[threads.h](#) で定義された関連する Thread オブジェクトがあります。このオブジェクトは、VM がマネージドスレッドについて知る必要があるすべての情報を追跡します。これには、スレッドの現在の GC モードや Frame チェーンなど_必須_のものだけでなく、パフォーマンス上の理由からスレッドごとに割り当てられている多くのもの（一部の高速なアリーナスタイルのアロケータなど）も含まれます。

すべての Thread オブジェクトは ThreadStore（これも [threads.h](#) で定義）に格納されます。ThreadStore は、既知のすべての Thread オブジェクトの単純なリストです。すべてのマネージドスレッドを列挙するには、まず ThreadStoreLock を取得し、次に [ThreadStore:: GetAllThreadList](#) を使用してすべての Thread オブジェクトを列挙します。このリストには、現在ネイティブスレッドに割り当てられていないマネージドスレッド（たとえば、まだ開始されていないもの、またはネイティブスレッドが既に終了したものなど）が含まれている場合があります。

現在ネイティブスレッドに割り当てられている各マネージドスレッドは、そのネイティブスレッドのネイティブスレッドローカルストレージ (TLS) スロットを通じてアクセス可能です。これにより、そのネイティブスレッド上で実行中のコードが、[GetThread\(\)](#) を通じて対応する Thread オブジェクトを取得できます。

さらに、多くのマネージドスレッドは _マネージド_ の Thread オブジェクト（[System.Threading.Thread](#)）を持っています。これはネイティブの Thread オブジェクトとは別のものです。マネージド Thread オブジェクトは、マネージドコードがスレッドと対話するためのメソッドを提供し、ネイティブ Thread オブジェクトが提供する機能のラッパーとして機能します。現在のマネージド Thread オブジェクトは、マネージドコードから [Thread.CurrentThread](#) を通じてアクセスできます。

デバッガでは、SOS 拡張コマンドの [!Threads](#) を使用して ThreadStore 内のすべての Thread オブジェクトを列挙できます。

スレッドのライフタイム

マネージドスレッドは以下の状況で作成されます：

1. マネージドコードが [System.Threading.Thread](#) を通じて明示的に CLR に新しいスレッドの作成を依頼する
2. CLR が直接マネージドスレッドを作成する（下記の [「特殊なスレッド」](#) を参照）
3. ネイティブコードが、まだマネージドスレッドに関連付けられていないネイティブスレッド上でマネージドコードを呼び出す（「リバース P/Invoke」または COM 相互運用を通じて）
4. マネージドプロセスが開始される（プロセスの Main スレッド上で Main メソッドを呼び出す）

ケース #1 と #2 では、CLR がマネージドスレッドを裏付けるネイティブスレッドを作成する責任を持ちます。ただし、これはスレッドが実際に _開始_ されるまで行われません。これらのケースでは、ネイティブスレッドは CLR に「所有」されており、CLR がネイティブスレッドのライフタイムに責任を持ちます。CLR は、自身がスレッドを作成したという事実によって、スレッドの存在を認識しています。

ケース #3 と #4 では、マネージドスレッドの作成前にネイティブスレッドがすでに存在しており、CLR の外部のコードに所有されています。CLR はネイティブスレッドのライフタイムに責任を持ちません。CLR がこれらのスレッドを認識するのは、初めてマネージドコードの呼び出しが試みられたときです。

ネイティブスレッドが終了すると、CLR は `DllMain` 関数を通じて通知を受けます。これは OS の「ローダーロック」内で発生するため、この通知の処理中に（安全に）実行できることはほとんどありません。そのため、マネージドスレッドに関連するデータ構造を破棄する代わりに、スレッドは単に「デッド」としてマークされ、ファイナライザスレッドに実行シグナルを送ります。ファイナライザスレッドは ThreadStore 内のスレッドを走査し、デッドでありかつ_マネージドコードから到達不能なスレッドを破棄します。

💡 初心者向け補足

「ローダーロック」とは、Windows が DLL のロード・アンロードを安全に管理するために使用する内部ロックです。このロック内で複雑な操作を行うとデッドロックの原因になるため、CLR はスレッドの破棄を後回しにして、ファイナライザスレッドに委任しています。

サスペンション

CLR は GC を実行するために、マネージドオブジェクトへのすべての参照を見つける必要があります。マネージドコードは常に GC ヒープにアクセスし、スタックやレジスタに格納された参照を操作しています。CLR は、すべてのマネージドオブジェクトを安全かつ確実に見つけるために、すべてのマネージドスレッドを停止（ヒープの変更を防止）する必要があります。停止はレジスタやスタック上のライブ参照を検査できる_セーフポイント_でのみ行われます。

別の言い方をすれば、GC ヒープと各スレッドのスタック・レジスタの状態は、複数のスレッドからアクセスされる「共有状態」です。ほとんどの共有状態と同様に、これを保護するために何らかの「ロック」が必要です。マネージドコードはヒープにアクセスする際にこのロックを保持し、セーフポイントでのみロックを解放できます。

CLR はこの「ロック」をスレッドの **GC モード** と呼びます。コオペラティブモード (**cooperative mode**) にあるスレッドはロックを保持しています。GC が進行するためには、スレッドがロックを解放して GC に「協力 (cooperate)」する必要があります。プリエンプティブモード (**preemptive mode**) にあるスレッドはロックを保持していません。そのスレッドは GC ヒープにアクセスしていないことがわかっているため、GC は「先制的 (preemptively)」に進行できます。

💡 初心者向け補足

GC モードは、スレッドが GC ヒープを安全に操作できるかどうかを管理する仕組みです。「コオペラティブモード」はスレッドがヒープを使用中であることを示し、「プリエンプティブモード」はスレッドがヒープを使用していないことを示します。GC はすべてのスレッドがプリエンプティブモードになるまで待機してから実行されます。

GC は、すべてのマネージドスレッドがプリエンプティブモード（ロックを保持していない状態）にある場合にのみ進行できます。すべてのマネージドスレッドをプリエンプティブモードに移行させるプロセスは、**GC サスペンション**または**実行エンジン (EE)** のサスPENDとして知られています。

このロックの素朴な実装としては、各マネージドスレッドが GC ヒープへのアクセスのたびに実際のロックを取得・解放する方法が考えられます。すると GC は各スレッドのロックの取得を試みるだけで済み、すべてのスレッドのロックを取得できれば GC を安全に実行できます。

しかし、この素朴なアプローチには 2 つの理由で問題があります。第一に、マネージドコードがロックの取得と解放（または GC がロックの取得を試みているかどうかのチェック、いわゆる「GC ポーリング」）に多くの時間を費やすことになります。第二に、JIT がすべての

ポイントにおけるスタックとレジスタのレイアウトを記述する「GC 情報」を出力する必要があり、この情報は大量のメモリを消費します。

この素朴なアプローチを改良するため、JIT されたマネージドコードは部分的に割り込み可能 (**partially interruptible**) なコードと完全に割り込み可能 (**fully interruptible**) なコードに分離されます。

- 部分的に割り込み可能なコードでは、セーフポイントは他のメソッドへの呼び出し箇所と、JIT が GC の保留をチェックするコードを出力する明示的な「GC ポーリング」地点のみです。GC 情報はこれらの地点に対してのみ出力すればよいです。
- 完全に割り込み可能なコードでは、すべての命令がセーフポイントであり、JIT はすべての命令に対して GC 情報を出力しますが、GC ポーリングは出力しません。代わりに、完全に割り込み可能なコードは、スレッドのハイジャック（後述）によって「割り込み」を受けることができます。

JIT は、コード品質、GC 情報のサイズ、GC サスペンションのレイテンシの間で最適なトレードオフを見つけるためのヒューリスティクスに基づいて、完全に割り込み可能なコードと部分的に割り込み可能なコードのどちらを出力するかを選択します。

以上を踏まえると、定義すべき 3 つの基本操作があります：コオペラティブモードへの移行、コオペラティブモードからの離脱、EE のサスペンドです。

コオペラティブモードへの移行

スレッドは `Thread::DisablePreemptiveGC` を呼び出してコオペラティブモードに入ります。これにより、現在のスレッドの「ロック」が以下のように取得されます：

1. GC が進行中（GC がロックを保持している）であれば、GC が完了するまでブロックする
2. スレッドをコオペラティブモードとしてマークする。スレッドがプリエンプティブモードに再移行するまで、GC は進行できない

これら 2 つのステップはアトミックであるかのように進行します。

プリエンプティブモードへの移行

スレッドは `Thread::EnablePreemptiveGC` を呼び出してプリエンプティブモード（ロックの解放）に入ります。これは単にスレッドをコオペラティブモードではなくたとマークし、GC スレッドに進行可能になった可能性を通知します。

EE のサスペンド

GC が必要になると、最初のステップは EE のサスペンドです。これは `GCHeap::SuspendEE` によって以下のように行われます：

1. GC が進行中であることを示すグローバルフラグ（`g_fTrapReturningThreads`）を設定する。コオペラティブモードに入ろうとするスレッドは GC が完了するまでブロックされる
2. 現在コオペラティブモードで実行中のすべてのスレッドを見つける。それぞれのスレッドについて、スレッドをハイジャックしてコオペラティブモードから強制的に離脱させる
3. コオペラティブモードで動作中のスレッドがなくなるまで繰り返す

ハイジャック

GC サスペンションのためのハイジャックは `Thread::SysSuspendForGC` によって行われます。このメソッドは、現在コオペラティブモードで実行中のマネージドスレッドを「セーフポイント」でコオペラティブモードから強制的に離脱させることを試みます。これは

ThreadStore を走査してすべてのマネージドスレッドを列挙し、コオペラティブモードで実行中の各マネージドスレッドについて以下を行います：

1. ネイティブスレッドのサスPEND。Win32 の `SuspendThread` API を使用して行われます。この API はスレッドの実行を強制的に停止しますが、停止する地点は実行中のランダムな位置であり、必ずしもセーフポイントではありません。
2. スレッドの現在の `CONTEXT` を取得。`GetThreadContext` を使用して取得します。これは OS の概念であり、CONTEXT はスレッドの現在のレジスタ状態を表します。これにより命令ポインタを検査でき、スレッドが現在実行しているコードの種類を判定できます。
3. スレッドがまだコオペラティブモードかどうかを再確認。サスPENDされる前にすでにコオペラティブモードを離脱している可能性があります。その場合、スレッドは危険な状態にあります（任意のネイティブコードを実行している可能性がある）ため、デッドロックを回避するために直ちにスレッドを再開する必要があります。
4. スレッドがマネージドコードを実行中かどうかを確認。コオペラティブモードでネイティブ VM コードを実行している可能性があり（下記の「同期」を参照）、その場合はスレッドを直ちに再開する必要があります。
5. マネージドコード内でスレッドがサスPENDされた場合。そのコードが完全に割り込み可能か部分的に割り込み可能かに応じて、以下のいずれかが実行されます：
 - 完全に割り込み可能な場合：定義上、スレッドはセーフポイントにあるため、任意の地点で GC を実行できます。この地点でスレッドをサスPENDしたままにすることは安全ですが、歴史的な OS のバグにより、先ほど取得した CONTEXT が破損している可能性があるため、これは機能しません。代わりに、スレッドの命令ポインタが上書きされ、より完全な CONTEXT をキャプチャし、コオペラティブモードを離脱し、GC の完了を待ち、コオペラティブモードに再移行し、スレッドの以前の状態を復元するスタブにリダイレクトされます。
 - 部分的に割り込み可能な場合：定義上、スレッドはセーフポイントにはありません。しかし、呼び出し元はセーフポイント（メソッドの遷移地点）にあります。この知識を利用して、CLR は最上位のスタックフレームの戻りアドレスを「ハイジャック」し（スタック上のその場所を物理的に上書きし）、完全に割り込み可能なコードで使用されるものと同様のスタブに置き換えます。メソッドが戻る際、実際の呼び出し元ではなくスタブに戻ります（メソッドはその前に JIT が挿入した GC ポーリングを実行し、コオペラティブモードを離脱してハイジャックを元に戻す場合もあります）。

💡 初心者向け補足

「ハイジャック」とは、スレッドの実行フローを強制的に変更するテクニックです。スレッドの戻りアドレスや命令ポインタを書き換えて、GC が安全に実行できる状態に持っていきます。この仕組みは高度な低レベル操作であり、OS の API を活用しています。

ThreadAbort / AppDomain-Unload

AppDomain をアンロードするためには、CLR はその AppDomain 内でスレッドが実行されていないことを保証する必要があります。これを実現するために、すべてのマネージドスレッドが列挙され、アンロード対象の AppDomain に属するスタックフレームを持つスレッドが「中断 (abort)」されます。実行中のスレッドに `ThreadAbortException` が「注入」され、スレッドがバックアウトコードを実行しながら巻き戻され、AppDomain 内で実行されなくなった時点では `ThreadAbortException` は `AppDomainUnloadedException` に変換されます。

`ThreadAbortException` は特殊な種類の例外です。ユーザーコードでキャッチすることはできますが、CLR はユーザーの例外ハンドラの実行後に例外が再スローされることを保証します。そのため `ThreadAbortException` は「キャッチ不可能」と呼ばれることがあります、厳密にはこれは正確ではありません。

`ThreadAbortException` は通常、マネージドスレッドに「中断中 (aborting)」というビットを設定するだけでスローされます。このビットは CLR の各所（特に、すべての P/Invoke からの戻り時）で確認されるため、多くの場合、このビットを設定するだけでスレッドを

タイムリーに中断できます。

しかし、スレッドが長時間実行されるマネージドループを実行している場合など、このビットがチェックされないことがあります。このようなスレッドをより速く中断するために、スレッドは「ハイジャック」され、`ThreadAbortException` を発生させるよう強制されます。このハイジャックは GC サスペンションと同じ方法で行われますが、スレッドがリダイレクトされるスタブは、GC の完了を待つのではなく `ThreadAbortException` を発生させます。

このハイジャックは、`ThreadAbortException` がマネージドコード内の基本的にどの任意の地点でも発生しうることを意味します。このため、マネージドコードが `ThreadAbortException` を正常に処理することは極めて困難です。したがって、AppDomain-Unload 以外の目的でこのメカニズムを使用するのは賢明ではありません。AppDomain-Unload では `ThreadAbort` によって破損した状態が AppDomain とともにクリーンアップされることが保証されます。

💡 初心者向け補足

`ThreadAbortException` は、通常の例外と異なり、`catch` ブロックで捕捉しても自動的に再スローされる特殊な例外です。これにより、スレッドを確実に終了させることができます。ただし、.NET Core / .NET 5 以降では AppDomain のアンロードはサポートされておらず、`Thread.Abort()` も `PlatformNotSupportedException` をスローします。

同期: マネージド

マネージドコードは `System.Threading` 名前空間に集められた多くの同期プリミティブにアクセスできます。これには、Mutex、Event、Semaphore オブジェクトなどのネイティブ OS プリミティブのラッパーに加えて、Barrier や SpinLock などの抽象化が含まれます。しかし、ほとんどのマネージドコードが使用する主要な同期メカニズムは `System.Threading.Monitor` です。これは_任意のマネージドオブジェクト_に対する高性能なロック機能を提供し、さらにロックによって保護された状態の変化を通知するための「条件変数」セマンティクスも提供します。

Monitor はハイブリッドロックとして実装されています。スピinnロックとカーネルベースのロック (Mutex など) の両方の特性を兼ね備えています。その考え方は、ほとんどのロックは短時間しか保持されないため、カーネルを呼び出してスレッドをブロックするよりも、ロックが解放されるのをスピinnウェイト (ビギンウェイト) する方が速いということです。ただし、スピinnで CPU サイクルを無駄にしないよう、短期間のスピinnの後にロックが取得できなかった場合、実装はカーネルでのブロッキングにフォールバックします。

💡 初心者向け補足

ハイブリッドロックは、最初に短時間のスピinn (CPU をビギーに回す) を試み、それでもロックを取得できない場合にカーネルモードでの待機に切り替えます。これにより、短時間のロックでは高速に動作し、長時間のロックでは CPU リソースを無駄にしません。C# で `lock` ステートメントを使うと、内部的にこの `Monitor` が使用されます。

任意のオブジェクトがロック/条件変数として使用される可能性があるため、すべてのオブジェクトにロック情報を格納する場所が必要です。これはオブジェクトヘッダーと同期ブロック (`sync block`) によって行われます。

オブジェクトヘッダーは、すべてのマネージドオブジェクトの前に配置されるマシンワードサイズのフィールドです。オブジェクトのハッシュコードの格納など、多くの目的に使用されます。その目的の 1 つがオブジェクトのロック状態の保持です。オブジェクトヘッダーに収まらないほどのオブジェクト単位のデータが必要な場合、「同期ブロック」を作成してオブジェクトを「膨張 (inflate)」させます。

同期ブロックは同期ブロックテーブル (`Sync Block Table`) に格納され、同期ブロックインデックスでアドレスされます。関連する同期ブロックを持つ各オブジェクトは、そのインデックスをオブジェクトのオブジェクトヘッダーに保持しています。

オブジェクトヘッダーと同期ブロックの詳細は [syncblk.h / syncblk.cpp](#) で定義されています。

オブジェクトヘッダーに空きがある場合、Monitor はオブジェクトのロックを現在保持しているスレッドのマネージドスレッド ID を格納します（ロックを保持しているスレッドがない場合はゼロ (0)）。この場合、ロックの取得はオブジェクトヘッダーのスレッド ID がゼロになるまでスピンウェイトし、その後アトミックに現在のスレッドのマネージドスレッド ID を設定するだけの簡単な操作です。

一定回数のスピンの後にロックを取得できない場合、またはオブジェクトヘッダーがすでに他の目的で使用されている場合は、同期ブロックを作成する必要があります。同期ブロックには、現在のスレッドをブロックするために使用できるイベントなどの追加データが含まれており、スピンを停止してロックの解放を効率的に待機できます。

条件変数として使用されるオブジェクト（`Monitor.Wait` と `Monitor.Pulse` を通じて）は、必要な状態を保持するためのオブジェクトヘッダーの空きが十分にないため、常に膨張させる必要があります。

同期: ネイティブ

CLR のネイティブ部分もスレッディングを意識する必要があります。複数のスレッドからマネージドコードによって呼び出されるためです。これにはロック、イベントなどのネイティブ同期メカニズムが必要です。

ITaskHost API を使用すると、ホストはスレッドの作成、破棄、同期など、マネージドスレッディングの多くの側面をオーバーライドできます。ホストがネイティブ同期をオーバーライドできるということは、VM コードは一般的にネイティブ同期プリミティブ（クリティカルセクション、Mutex、Event など）を直接使用できず、これらに対する VM のラッパーを使用する必要があるということです。

さらに、前述のように、GC サスペンションは CLR のほぼすべての側面に影響する特殊な「ロック」です。VM 内のネイティブコードは、GC ヒープオブジェクトを操作する必要がある場合にコオペラティブモードに入ることができ、そのため「GC サスペンションロック」はネイティブ VM コードとマネージドコードの両方で最も重要な同期メカニズムの 1 つとなっています。

ネイティブ VM コードで使用される主要な同期メカニズムは、GC モードと Crst です。

GC モード

前述のように、すべてのマネージドコードは GC ヒープを操作する可能性があるため、コオペラティブモードで実行されます。一般的に、ネイティブコードはマネージドオブジェクトに触れないため、プリエンプティブモードで実行されます。しかし、VM 内の一部のネイティブコードは GC ヒープにアクセスする必要があり、コオペラティブモードで実行されなければなりません。

ネイティブコードは一般的に GC モードを直接操作せず、2 つのマクロ `GCX_COOP` と `GCX_PREEMP` を使用します。これらは目的のモードに入り、スコープを抜けるときにスレッドを以前のモードに戻すための「ホルダー」を設置します。

`GCX_COOP` は実質的に GC ヒープのロックを取得することを理解することが重要です。スレッドがコオペラティブモードにある間、GC は進行できません。また、ネイティブコードはマネージドコードのようにハイジャックできないため、スレッドは明示的にプリエンプティブモードに戻るまでコオペラティブモードのままでです。

したがって、ネイティブコードでコオペラティブモードに入ることは推奨されません。コオペラティブモードに入る必要がある場合は、可能な限り短い時間に抑えるべきです。このモードではスレッドをブロックしてはならず、特にロックを安全に取得することは一般的にできません。

同様に、`GCX_PREEMP` は、スレッドが保持していたロックを _解放_ する可能性があります。プリエンプティブモードに入る前に、すべての GC 参照が適切に保護されていることを確認するために細心の注意が必要です。

Crst

Monitor がマネージドコードに適したロックメカニズムであるのと同様に、Crst は VM コードに適したメカニズムです。Monitor と同様に、Crst はホストと GC モードを意識するハイブリッドロックです。Crst はまた、「ロックレベリング」によるデッドロック回避も実装しています。

一般的に、コオペラティブモード中に Crst を取得することは不正ですが、絶対に必要な場合には例外が認められています。

特殊なスレッド

マネージドコードによって作成されるスレッドの管理に加えて、CLR は独自の用途のためにいくつかの「特殊な」スレッドを作成します。

ファイナライザスレッド

このスレッドはマネージドコードを実行するすべてのプロセスで作成されます。GC がファイナライズ可能なオブジェクトが到達不能になったと判断すると、そのオブジェクトをファイナライゼーションキューに配置します。GC の終了時に、ファイナライザスレッドにシグナルが送られ、キュー内のすべてのファイナライザを処理します。各オブジェクトは 1 つずつデキューされ、そのファイナライザが実行されます。

このスレッドは、CLR 内部のハウスキーピングタスクの実行や、外部イベント（たとえば、メモリ不足の通知。これにより GC はより積極的にコレクションを行います）の通知待機にも使用されます。詳細は [GCHeap::FinalizerThreadStart](#) を参照してください。

GC スレッド

「コンカレント」または「サーバー」モードで実行している場合、GC はガベージコレクションの各段階を並行して実行するために、1 つ以上のバックグラウンドスレッドを作成します。これらのスレッドは完全に GC によって所有・管理され、マネージドコードを実行することはありません。

デバッガスレッド

CLR は各マネージドプロセスで単一のネイティブスレッドを維持しています。このスレッドは、アタッチされたマネージドデバッガに代わって各種タスクを実行します。

AppDomain-Unload スレッド

このスレッドは AppDomain のアンロードを担当します。これは、アンロードを要求したスレッドではなく、CLR 内部の別スレッドで行われます。その理由は、a) アンロードロジックのためのスタック空間を保証するため、b) 必要に応じて、アンロードを要求したスレッドを AppDomain から巻き戻せるようにするためです。

ThreadPool スレッド

CLR の ThreadPool は、ユーザーの「ワークアイテム」を実行するためのマネージドスレッドのコレクションを維持します。これらのマネージドスレッドは ThreadPool が所有するネイティブスレッドにバインドされています。ThreadPool はまた、「スレッドインジェクシ

ョン」、タイマー、「登録済み待機」などの機能を処理するための少数のネイティブスレッドも維持しています。

 初心者向け補足

ThreadPool は、スレッドの作成と破棄のオーバーヘッドを避けるために、あらかじめスレッドのプールを用意しておく仕組みです。C# では `Task.Run()` や `ThreadPool.QueueUserWorkItem()` を使って、ThreadPool のスレッド上で作業を実行できます。

RyuJIT 概要

原文

この章の原文は [RyuJIT Overview](#) です。

はじめに

RyuJIT は .NET ランタイムの JIT (Just-In-Time) コンパイラのコードネームです。.NET Framework の x86 用 JIT (jit32) から進化し、.NET がサポートするすべてのアーキテクチャとプラットフォームに対応しています。

RyuJIT の主な設計目標は以下の通りです：

- 以前の JIT との高い互換性を維持する
- コード最適化、レジスタ割り当て、コード生成を通じて良好なランタイムパフォーマンスを実現する
- 線形次数の最適化・変換を中心とし、良好なスループットを確保する
- さまざまなターゲットとシナリオをサポートするアーキテクチャを設計する

💡 初心者向け補足

JIT コンパイラとは、プログラムの実行時に中間言語 (IL) を CPU が直接実行できるネイティブコード（機械語）に変換するコンパイラです。「Just-In-Time (ちょうどその時に)」という名前の通り、コードが実際に必要になったタイミングでコンパイルを行います。

実行環境と外部インターフェース

RyuJIT は .NET ランタイムに対して JIT コンパイルと AOT (Ahead-of-Time) コンパイルの両方のサービスを提供します。ランタイム自体は EE (Execution Engine)、VM (Virtual Machine)、または CLR (Common Language Runtime) と呼ばれます。

RyuJIT は JIT/EE インターフェースの JIT 側を実装しています：

- `ICorJitCompiler` – JIT コンパイラが実装するインターフェースです。主要なメソッドは以下の通りです：
 - `compileMethod` – JIT のメインエントリポイント。EE から `ICorJitInfo` オブジェクト、IL、メソッドヘッダーなどの情報を受け取り、コードへのポインタ、サイズ、GC/EH/デバッグ情報を返します。
 - `getVersionIdentifier` – JIT/EE インターフェースのバージョン管理メカニズム。
- `ICorJitInfo` – EE が実装するインターフェースです。メタデータトークンの検索、型シグネチャの走査、フィールドや vtable のオフセット計算、メソッドエントリポイントの検索など、多くのメソッドが定義されています。

💡 初心者向け補足

JIT コンパイラは単独で動作するのではなく、ランタイム (EE) と密接に連携しています。EE は型情報やメソッドの情報を提供し、JIT はその情報を使ってネイティブコードを生成します。このインターフェースを通じた役割分担により、それぞれの責任範囲が明確になっています。

内部表現 (IR)

Compiler オブジェクト

`Compiler` オブジェクトは JIT の主要なデータ構造です。`ICorJitCompiler::compileMethod()` はメソッドごとに呼び出され、新しい `Compiler` オブジェクトを作成します。そのため、JIT は `Compiler` の状態にアクセスする際にスレッド同期を気にする必要がありません。

IR の概要

RyuJIT は関数を `BasicBlock` の双方向リンクリストとして表現します。各 `BasicBlock` には後続ブロックへの明示的なエッジがあり、非例外制御フローを定義します。

コンパイルの初期段階では、各 `BasicBlock` には HIR (High-level Intermediate Representation) と呼ばれる高レベルな文・木構造の形式でノードが含まれます。バックエンドの最初のフェーズ（合理化フェーズ）で、HIR は LIR (Low-level Intermediate Representation) と呼ばれる線形順序のノード指向形式に変換されます。

- HIR - 文 (`Statement`) の双方向リンクリストで構成。各文は式ツリーを参照。ツリー順序で実行される。
- LIR - `GenTree` ノードの双方向リンクリストで構成。リストの順序で実行される。

💡 初心者向け補足

コンパイラが内部的にプログラムを表現するためのデータ構造を「中間表現 (IR)」と呼びます。RyuJIT では、最初は人間にわかりやすい木構造 (HIR) でプログラムを表現し、最適化を経て、最終的に機械に近い線形表現 (LIR) に変換します。これは「高い抽象度から低い抽象度へ段階的に変換する」という、多くのコンパイラに共通するアプローチです。

GenTree ノード

各操作は `GenTree` ノードとして表現され、オペコード (`GT_xxx`)、ゼロ個以上の子・オペランドノード、およびそのノードのセマンティクスを表現するための追加フィールドを持ちます。すべてのノードには型、値番号、アサーション、レジスタ割り当てなどの情報が含まれます。

ローカル変数記述子

`LclVarDsc` は複数回定義・使用される可能性のある一時変数の情報を表します。ユーザーのローカル変数、引数、JIT が作成する一時変数を表現するために使用されます。ローカル変数は「追跡対象 (`lvTracked`)」にすることができ、その場合はデータフロー解析に参加し、レジスタ割り当ての候補となります。

RyuJIT のフェーズ

RyuJIT のコンパイルは以下の主要なフェーズで構成されています：

フェーズ	説明
プロファイル取り込み	PGO 情報を基本ブロックに取り込む
インポート	IL から <code>GenTree</code> ノードを作成し、 <code>Statement</code> と <code>BasicBlock</code> にリンク。インライン候補を特定
インライン化	インラインメソッドの IR をフローグラフに取り込む
オブジェクトスタック割り当て	エスケープ解析を行い、可能な場合はオブジェクトをスタックに割り当てる
構造体プロモーション	一部の構造体のフィールドに対して個別の <code>LclVarDsc</code> を作成
ローカル変形	ローカルアクセスの簡略化、アドレス公開ローカルの検出
グローバル変形	局所的な変換と単純な最適化を実行
QMARG 除去	<code>GT_QMARG</code> ノードを制御フローに変換
ループ検出	自然ループを検出し正規化
ループ反転	自然ループを "do...while" 形式に変換
ループクローン	一部のループを追加バージョンとしてクローン
ループ展開	自然ループを展開
SSA と値番号付け	生存解析、SSA 構築、値番号計算を実行
ループ不变コード移動 (LICM)	ループ不变な式をループ外に移動
コピー伝播	値番号に基づくコピー伝播
冗長分岐最適化	分岐の最適化
共通部分式除去 (CSE)	値番号に基づく冗長な部分式の除去
アサーション伝播	非 null 性などのプロパティに基づく変換
範囲解析	値番号とアサーションに基づく配列インデックス範囲チェックの除去
帰納変数最適化	スカラー進化解析に基づくループ内の帰納変数の最適化
合理化	HIR から LIR への変換。 <code>GT_COMMA</code> ノードの除去

フェーズ	説明
ローフリング	レジスタ割り当て用にノードを変換。ターゲット固有の最適化
レジスタ割り当て	レジスタの割り当て、スピル一時変数の計算
コード生成	フレームレイアウトの決定。各 <code>BasicBlock</code> のコード生成。プロローグ・エピローグの生成。EH、GC、デバッグ情報の出力

💡 初心者向け補足

JIT コンパイラの仕事は「IL を受け取って機械語を出力する」ことですが、その間に多くの最適化フェーズがあります。例えば：

- インライン化: 小さなメソッドの呼び出しをメソッド本体のコードで置き換え、関数呼び出しのオーバーヘッドを除去します
- CSE (共通部分式除去): 同じ計算が複数回行われている場合、1回だけ計算して結果を再利用します
- レジスタ割り当て: 変数をメモリではなく CPU レジスタに配置し、高速なアクセスを実現します
- ループ最適化: ループの実行を効率化する様々な手法（不变コードの移動、ループ展開など）を適用します

インポートフェーズ

インポートは IL 命令を1つずつ読み取り、メソッドの IR を作成するフェーズです。このプロセスでは、ネストされた複数の式を持つ IR を生成する必要がある場合があります。`GT_COMMA` ノード（実行順序の保証）や `GT_QMARK` / `GT_COLON` ノード（条件式）が使用されます。

インライン化

`fgInline` フェーズでは、各呼び出しサイトがインライン化の候補かどうかを判定します。候補メソッドの IL に対して状態マシンを実行し、ネイティブコードサイズを推定します。ヒューリスティクスに基づいてインライン化が有益と判断された場合、別の `Compiler` オブジェクトが作成され、候補メソッドのツリーがインポートされます。

SSA と値番号付け

SSA (Static Single Assignment) 形式は従来の方法で構築されます。SSA 名はローカル変数参照に記録されます。値番号付けはローカル変数の SSA を利用しますが、式ツリーの値番号付けも行います。型安全性を活用して、同じフィールドへの書き込みでない限り、ヒープ書き込みでフィールド参照の値番号を無効化しません。

合理化 (Rationalization)

合理化フェーズでは、フローグラフの順序がツリー順序から線形順序に変更されます。すべての `GT_COMMA` ノードが除去されます。この時点では IR は LIR に変換されます。

ローフリング (Lowering)

ローフーリングフェーズでは、IR ノードがレジスタ割り当てに適した形に変換されます。ターゲット固有の最適化もこのフェーズで行われます。例えば、アドレッシングモードの形成や、特定の命令パターンへの変換が行われます。

レジスタ割り当て

RyuJIT は LSRA (Linear Scan Register Allocation) を使用しています。各ノードにレジスタが割り当てられ、レジスタが不足する場合はスpill（一時的にメモリに退避）が行われます。

💡 初心者向け補足

CPU には少数の「レジスタ」と呼ばれる超高速のメモリがあります。変数をレジスタに置くとメモリアクセスよりはるかに高速ですが、レジスタの数は限られています。レジスタ割り当てとは、「どの変数をいつレジスタに置くか」を決定する重要な最適化です。レジスタに収まりきらない変数は「スpill」としてスタックに退避されます。

コード生成

コード生成フェーズでは、最終的なネイティブコードが生成されます。各 `BasicBlock` を走査し、LIR のノードに対応するマシン命令を出力します。このフェーズではフレームレイアウトの決定、プロローグ・エピローグコードの生成、EH (例外処理)、GC、デバッグ情報の出力も行います。

RyuJIT を異なるプラットフォームへ移植する

原文

この章の原文は [Porting RyuJIT to other platforms](#) です。

まず、[RyuJIT の概要](#) を読んで、JIT アーキテクチャを理解してください。

プラットフォーム (Platform) とは何か

- ターゲット命令セット (target instruction set)
- ターゲットポインタサイズ (target pointer size)
- ターゲットオペレーティングシステム (target operating system)
- ターゲット呼び出し規約 (calling convention) と ABI (アプリケーションバイナリインターフェース; Application Binary Interface)
- ランタイムデータ構造 (runtime data structures) (ここではあまり触れません)
- GC エンコーディング (GC encoding)
 - Windows x86 が JIT32_GCENCODER を使用する点を除き、すべてのターゲットは同じ GC エンコーディング方式と API を使用します。
 - デバッグ情報 (debug information) (ほとんどのターゲットで共通)
 - 例外処理 (EH; exception handling) の情報 (ここではあまり触れません)

CLR の利点の一つは、VM が (ABI 以外の) OS の違いを (ほぼ) 隠蔽してくれることです。

💡 初心者向け補足

ABI (Application Binary Interface) とは、コンパイル済みのバイナリ同士がどうやりとりするかを定めた低レベルの規約です。関数呼び出し時に引数をどのレジスタに置くか、戻り値をどう返すか、スタックフレームのレイアウトなどが含まれます。Java で言えば JNI のようなネイティブインターフェースの裏側で意識される部分です。

概観 (The Very High Level View)

新しいプラットフォームに対応するために、以下のコンポーネントを更新するか、ターゲット固有のバージョンを作成する必要があります。

- 基本部分
 - target.h
- 命令セットアーキテクチャ (Instruction Set Architecture):
 - registerXXX.h - アーキテクチャで使用するすべてのレジスタとそのエイリアスを定義

- emitXXX.h - 公開用の命令エミッション (instruction emission) メソッドのシグネチャ (例: 「整数の引数を1つ取る命令をエミットする」) およびプライベートなアーキテクチャ固有のヘルパーを定義
- emitXXX.cpp - emitXXX.h の実装
- emitfmtXXX.h - 命令のフォーマット方法に関する妥当性ルールをオプションで定義 (例: RISC-V ではルールが定義されていません)
- instrsXXX.h - アーキテクチャごとのアセンブリ命令を定義
- targetXXX.h - その他の箇所で使用されるアーキテクチャ上の制約を定義。例えば「呼び出し先保存 (callee-saved) の整数レジスタのビットマスク」や「浮動小数点レジスタのバイトサイズ」など
- targetXXX.cpp - このアーキテクチャの ABI クラシファイア (ABI classifier) を実装
- lowerXXX.cpp - このアーキテクチャのローフーリング ([Lowering](#)) を実装
- lsraXXX.cpp - [GenTree ノード](#) に基づくレジスタ要求の設定を実装
- codegenXXX.cpp - このアーキテクチャのメインのコード生成 (codegen) を実装 (つまり、[GenTree ノード](#) に基づいてアーキテクチャ固有の命令を生成する)
- hw intrinsic*XXX.* および simdshw intrinsic*XXX.h - ハードウェア組み込み関数 (hardware intrinsic) の機能を定義・実装 (例: ベクター命令)
- unwindXXX.cpp - 公開用のアンワインド (unwinding) API およびデバッグ用のアンワインド情報ダンプを実装
- 呼び出し規約と ABI: コードベース全体に散在しています
- 32 ビット vs. 64 ビット
 - これもコードベース全体に散在しています。ポインタサイズ固有のデータの一部は target.h に集約されていますが、おそらく 100% ではありません。

 初心者向け補足

XXX の部分には、ターゲットアーキテクチャの名前が入ります。例えば ARM64 向けの場合、`registerArm64.h`、`emitArm64.h`、`codegenArm64.cpp` のようなファイル名になります。RyuJIT は各アーキテクチャごとに専用のファイルセットを持ち、共通コードとアーキテクチャ固有のコードを分離しています。

移植の段階とステップ

JIT を移植するには、いくつかのステップを踏む必要があります (一部は並行して進められます)。以下に説明します。

初期立ち上げ (Initial bring-up)

- 新しいプラットフォーム固有のファイルを作成する
- プラットフォーム固有のビルド命令を作成する (CMakeLists.txt 内)。これにはおそらく、ソースツリーのルートレベルと JIT レベルの両方で、新しいプラットフォーム固有のビルド命令が必要になります。
- MinOpts に集中する。最適化フェーズを無効にするか、常に `DOTNET_JITMinOpts=1` でテストしてください。
- オプション機能を無効にする。例えば:

- `FEATURE_EH` -- 0 になると、すべての例外処理ブロックが削除されます。もちろん、例外のスローとキャッチに依存する例外処理テストは正しく動作しません。
- `FEATURE_STRUCTPROMOTE`
- `FEATURE_FASTTAILCALL`
- `FEATURE_TAILCALL_OPT`
- `FEATURE SIMD`
- 新しい JIT を altjit としてビルドする。このモードでは、「ベース」JIT が呼び出されてすべての関数をコンパイルしますが、`DOTNET_AltJit` 変数で指定された関数だけは例外です。例えば、`DOTNET_AltJit=Add` を設定してテストを実行すると、「ベース」JIT (例: Windows x64 ターゲットの JIT) がすべての関数をコンパイルしますが、`Add` だけは新しい altjit で最初にコンパイルされ、失敗した場合は「ベース」JIT にフォールバックします。こうすることで、ごく限られた JIT 機能だけが動作すればよく、「ベース」JIT がほとんどの関数を処理します。
- 基本的な命令エンコーディングを実装する。`CodeGen::genArm64EmitterUnitTests()` のようなメソッドを使用してテストしてください。
- 加算のような非常に単純な操作に対して、コンパイラがビルドしてコードを生成できる最低限の実装を行う。
- `CodeGenBringUpTests (src\tests\JIT\CodeGenBringUpTests)` に集中する。簡単なものから始めてください。
 - これらは、テスト `XXX.cs` に対して、コンパイル対象となる `XXX` という名前の単一の関数があるように設計されています (つまり、ソースファイルの名前と対象関数の名前が同じです。これはテストを実行するスクリプトを非常にシンプルにするためです)。`DOTNET_AltJit=XXX` を設定して、新しい JIT がその1つの関数だけをコンパイルするようにしてください。
 - マージされたテストグループは、各テストのエントリポイントを削除し、すべてのテストを单一のプロセスで呼び出す单一のラッパーを作成することで、これらのテストのシンプルさを損ないます。元の動作に戻すには、環境変数 `BuildAsStandalone` を `true` に設定してテストをビルドしてください。
- `DOTNET_JitDisasm` を使用して、コードが実行されなくても、関数に対して生成されたコードを確認できます。

 初心者向け補足

`altjit` とは「代替 JIT (alternative JIT)」の略で、新しいプラットフォーム向けの JIT を開発する際に非常に便利な仕組みです。既存の安定した JIT (ベース JIT) がほとんどの関数のコンパイルを担当し、開発中の新しい JIT は指定された関数だけをコンパイルします。これにより、一度にすべての機能を実装しなくても、一つずつ関数をテストしながら段階的に開発を進められます。Java の世界で例えると、C1 コンパイラをフォールバックとして使いながら新しいコンパイラの開発を進めるようなイメージです。

テストカバレッジの拡大 (Expand test coverage)

- ますます多くのテストが正常に実行されるようにしてください:
 - `JIT` ディレクトリのテストをさらに実行
 - すべての Pri-0 「innerloop」 テストを実行
 - すべての Pri-1 「outerloop」 テストを実行
- テストベース全体で JIT が生成するアサート (assert) のデータを収集し、頻度順にアサートを修正していくと効率的です。つまり、最も頻繁に発生するアサートを最初に修正してください。
- アサートの数、およびアサートの有無ごとのテスト数を追跡して、進捗状況を判断してください。

最適化フェーズの有効化 (Bring the optimizer phases on-line)

- `DOTNET_JITMinOpts=1` あり・なしの両方でテストを実行してください。
- かなり後の段階まで `DOTNET_TieredCompilation=0` を設定する（またはそのプラットフォームで完全に無効にする）のが合理的です。

品質の向上 (Improve quality)

- 基本モードでテストが通るようになったら、`JitStress` と `JitStressRegs` のストレスモードで実行を開始してください。
- `GCStress` を有効にしてください。これには VM 側の作業も必要です。
- `DOTNET_GCStress=4` の品質を向上させてください。crossgen/ngen が有効になったら、`DOTNET_GCStress=8` および `DOTNET_GCStress=C` でもテストしてください。

パフォーマンスの改善 (Work on performance)

- スループット（コンパイル時間）と生成コード品質 (CQ; Code Quality) の両方について、パフォーマンスを測定・改善するための戦略を策定してください。

プラットフォーム間の機能均一化 (Work on platform parity)

- 意図的に無効にした機能や、実装を延期していた機能を実装してください。
- SIMD (`Vector<T>`) およびハードウェア組み込み関数 (hardware intrinsics) のサポートを実装してください。

フロントエンドの変更 (Front-end changes)

- 呼び出し規約 (Calling Convention)
 - 構造体 (struct) の引数と戻り値が、最も複雑な差異となります
 - インポーター (Importer) とモーフ (Morph) はこれらを強く意識しています
 - 例: `fgMorphArgs()`、`fgFixupStructReturn()`、`fgMorphCall()`、`fgPromoteStructs()`、およびさまざまな構造体代入のモーフメソッド
 - ARM の HFA (Homogeneous Floating-point Aggregate; 同種浮動小数点集約体)
- テールコール (tail call) はターゲット依存ですが、おそらくもっと少なくすべきです
- 組み込み関数 (intrinsics): 各プラットフォームは異なるメソッドを組み込み関数として認識します（例: `Sin` は x86 のみ、`Round` は amd64 を除くすべて）
- mul, mod、div に対するターゲット固有のモーフ変換

バックエンドの変更 (Backend Changes)

- ローワリング (Lowering): 制御フロー (control flow) とレジスタ要求を完全に公開する

- コード生成 (Code Generation): レイアウト順にブロックを走査し、ノード上のレジスタ割り当てに基づいてコード (`InstrDesc`) を生成する
 - その後、プロローグ (prolog) とエピローグ (epilog)、GC テーブル、例外処理 (EH) テーブル、スコープテーブルを生成する
- ABI の変更:
 - 呼び出し規約のレジスタ要求
 - 呼び出し (call) と戻り (return) のローワリング
 - プロローグとエピローグのコードシケンス
 - フレーム (frame) の割り当てとレイアウト

ターゲット ISA の「構成」 (Target ISA "Configuration")

- 条件付きコンパイル (conditional compilation) (`jit.h` で設定、受け取った `define` に基づく。例: `#ifdef X86`)

```
C++  
_TARGET_64_BIT_ (32 ビットターゲットは単に !_TARGET_64BIT_)  
_TARGET_XARCH_, _TARGET_ARMARCH_  
_TARGET_AMD64_, _TARGET_X86_, _TARGET_ARM64_, _TARGET_ARM_
```

- `Target.h`
- `IntrsXXX.h`

💡 初心者向け補足

ISA (Instruction Set Architecture) とは命令セットアーキテクチャのことで、CPU がどのような命令をサポートするかを定義したものです。例えば x86/x64 (Intel/AMD) と ARM では全く異なる命令セットを持ちます。RyuJIT では `_TARGET_XARCH_` (x86/x64 系) と `_TARGET_ARMARCH_` (ARM 系) で大きく分岐しており、さらに `_TARGET_AMD64_` や `_TARGET_ARM64_` で細かい差異を扱います。C/C++ のプリプロセッサ (`#ifdef`) を使ってコンパイル時にターゲットを切り替えていきます。

命令エンコーディング (Instruction Encoding)

- `insGroup` と `instrDesc` のデータ構造がエンコーディングに使用されます
 - `instrDesc` はオペコード (opcode) ビットで初期化され、即値 (immediate) とレジスタ番号のフィールドを持ちます。
 - `instrDesc` は `insGroup` グループにまとめられます
 - ラベル (label) はグループの先頭にのみ存在できます
- エミッター (emitter) は以下のために呼び出されます:
 - コード生成 (CodeGen) の間に新しい命令 (`instrDesc`) を作成する
 - コード生成の完了後に `instrDesc` のビットをエミットする

- GC 情報（生存中の GC 変数とセーフポイント）を更新する
-

エンコーディングの追加 (Adding Encodings)

- 命令エンコーディングは instrsXXX.h に記述されます。各命令のオペコードビットを表しています
- 各命令セットのエンコーディングの構造はターゲット依存です
- 「命令 (instruction)」は単にオペコードの表現です
- `instrDesc` のインスタンスが、エミットされる命令を表します
- 各「タイプ」の命令に対して、エミットメソッドを実装する必要があります。これらはパターンに従いますが、ターゲットによっては固有のものがある場合があります。例:

C++

```
emitter::emitInsMov(instruction ins, emitAttr attr, GenTree* node)
emitter::emitIns_R_I(instruction ins, emitAttr attr, regNumber reg, ssize_t val)
emitter::emitInsTernary(instruction ins, emitAttr attr, GenTree* dst, GenTree* src1, GenTree* src2)
(現在 Arm64 のみ)
```

ローワリング (Lowering)

- ローワリングは、レジスタアロケータ (register allocator) に対してすべてのレジスタ要求を公開します
 - 使用カウント (use count)、定義カウント (def count)、「内部」レジスタカウント (internal reg count)、および特殊なレジスタ要求
 - すべての計算が明示的になるため、コード生成の半分の作業を担います
 - ただし、ローワリングされたツリーノードとターゲット命令が必ずしも 1:1 で対応するわけではありません
 - 最初のパスでツリーウォークを行い、命令を変換します。一部はターゲット非依存です。主な例外:
 - 呼び出し (call) と引数 (argument)
 - switch のローワリング
 - LEA 変換
 - 2番目のパスでは実行順にノードをウォークします
 - レジスタ要求を設定
 - すでに走査済みの子ノードのレジスタ要求を変更することもあります
 - LSRA のためにブロック順序とノードの位置を設定
 - `LinearScan::startBlockSequence()` と `LinearScan::moveToNextBlock()`

💡 初心者向け補足

ローワリング (Lowering) とは、JIT コンパイラの中間表現 (IR) を、ターゲットのハードウェアにより近い形に変換するフェーズです。例えば、高レベルの「加算」演算を、特定の CPU レジスタを使った具体的な命令に近づけます。LSRA (Linear Scan

Register Allocation) は線形走査レジスタ割り当てと呼ばれるアルゴリズムで、プログラム中の変数を CPU レジスタに効率よく割り当てます。ローフリングが「どのレジスタが必要か」を明示し、LSRA が「実際にどのレジスタを使うか」を決定するという分業になっています。

レジスタ割り当て (Register Allocation)

- レジスタ割り当ては大部分がターゲット非依存です
 - ローフリングの第2フェーズが、ほぼすべてのターゲット依存の作業を行います
- レジスタ候補はフロントエンドで決定されます
 - ローカル変数やテンポラリ (temp)、またはローカル変数やテンポラリのフィールド
 - アドレス取得 (address-taken) されていないこと、およびいくつかの追加制約
 - `lvaSortByRefCount()` でソートされ、`lvIsRegCandidate()` で決定されます

アドレッシングモード (Addressing Modes)

- アドレッシングモードを見つけてキャプチャするコードは、特に抽象化が不十分です
- CodeGenCommon.cpp 内の `genCreateAddrMode()` タリーを走査してアドレッシングモードを探し、その構成要素（ベース (base)、インデックス (index)、スケール (scale)、オフセット (offset)）を「出力パラメータ (out parameters)」としてキャプチャします
- コードを生成することなく、`gtSetEvalOrder` およびローフリングでのみ使用されます

コード生成 (Code Generation)

- ほとんどの場合、コード生成のメソッド構造はすべてのアーキテクチャで同じです
 - ほとんどのコード生成メソッドは「gen」で始まります
- 理論的には、CodeGenCommon.cpp はすべてのターゲットに「ほぼ」共通のコードを含みます（この分離は不完全ですが）
 - メソッドのプロローグ、エピローグなど
- `genCodeForBBLList()`
 - 実行順にツリーをウォークし、`genCodeForTreeNode()` を呼び出します。`genCodeForTreeNode()` は「含有 (contained)」されていないすべてのノードを処理する必要があります
 - ブロックの制御フローコード（分岐、例外処理）を生成します

型システム

原文

この章の原文は [Type System](#) です。

著者: David Wrighton ([@davidwrighton](#)) - 2010

はじめに

CLR の型システムは、ECMA 仕様に記述された型システムとその拡張をランタイム内部で表現するものです。

概要

型システムは、一連のデータ構造と、それらを操作・生成するアルゴリズムから構成されています。これはリフレクションを通じて公開される型システムではありません。ただし、リフレクションの型システムはこのシステムに依存しています。

💡 初心者向け補足

「型システム」とは、`int`、`string`、`List<T>`などの型に関する情報を管理する仕組みです。C# のコードで `typeof(string)` や `is` 演算子を使うと、裏ではこの型システムが動いています。リフレクション（`Type.GetType()` など）は、この内部型システムの上に構築された別のレイヤーです。

型システムが管理する主要なデータ構造は以下のとおりです：

- `MethodTable`
- `EEClass`
- `MethodDesc`
- `FieldDesc`
- `TypeDesc`
- `ClassLoader`

型システムに含まれる主要なアルゴリズムは以下のとおりです：

- **型ローダー (Type Loader):** 型を読み込み、型システムの主要なデータ構造の大部分を生成するために使用されます。
- **CanCastTo** および類似機能: 型同士を比較する機能です。
- **LoadTypeHandle:** 主に型の検索に使用されます。
- **シグネチャ解析 (Signature Parsing):** メソッドやフィールドの情報を比較・収集するために使用されます。
- **GetMethod/FieldDesc:** メソッド/フィールドを検索・読み込みするために使用されます。
- **Virtual Stub Dispatch:** インターフェースへの仮想呼び出しの宛先を見つけるために使用されます。

これら以外にも、CLR の他の部分にさまざまな情報を提供する補助的なデータ構造やアルゴリズムが多数存在しますが、システム全体の理解においてはそれほど重要ではありません。

コンポーネントアーキテクチャ

型システムのデータ構造は、一般にさまざまなアルゴリズムによって共通的に使用されます。本ドキュメントでは型システムのアルゴリズム自体は詳しく説明しません（他の Book of the Runtime のドキュメントで説明されている、またはされるべきものです）が、以下に主要なデータ構造を説明します。

依存関係

型システムは一般に CLR の多くの部分にサービスを提供しており、ほとんどのコアコンポーネントは型システムの動作に何らかの形で依存しています。

型システムの依存先

型システムの主な依存先は以下のとおりです：

- ローダー: 作業に必要な正しいメタデータを取得するために必要です。
- メタデータシステム: 情報を収集するためのメタデータ API を提供します。
- セキュリティシステム: 特定の型システム構造（例：継承）が許可されるかどうかを型システムに通知します。
- AppDomain: 型システムのデータ構造のメモリ割り当て動作を処理するための LoaderAllocator を提供します。

型システムに依存するコンポーネント

型システムに依存する主要なコンポーネントは 3 つあります：

- JIT インターフェース および JIT ヘルパーは、主に型・メソッド・フィールドの検索機能に依存しています。型システムオブジェクトが見つかると、JIT が必要とする情報を提供できるように調整されたデータ構造が返されます。
 - リフレクション は、型システムを利用して、CLR の型システムデータ構造に取り込まれた ECMA 標準化された概念への比較的シンプルなアクセスを提供します。
 - 一般的なマネージドコードの実行では、型比較ロジックや Virtual Stub Dispatch のために型システムを使用する必要があります。
-

型システムの設計

コアとなる型システムのデータ構造は、実際に読み込まれた型を表すデータ構造（TypeHandle、MethodTable、MethodDesc、TypeDesc、EEClass など）と、読み込まれた型を検索可能にするデータ構造（ClassLoader、Assembly、Module、RidMap など）に分類されます。

型を読み込むためのデータ構造とアルゴリズムについては、Book of the Runtime の [型ローダー](#) の章および [メソッドディスクリプタ](#) の章で説明されています。

これらのデータ構造を結びつけるのが、JIT / リフレクション / 型ローダー / スタックウォーカーが既存の型やメソッドを検索できるようにする一連の機能です。基本的な考え方は、これらの検索が ECMA CLI 仕様で規定されたメタデータトークンやシグネチャによって容易に駆動できることです。

そして最終的に、適切な型システムのデータ構造が見つかると、型から情報を収集したり、2つの型を比較したりするアルゴリズムがあります。この種のアルゴリズムの特に複雑な例は、Book of the Runtime の [仮想スタブディスパッチ](#) の章で見ることができます。

設計目標と非目標

目標

- 実行中の（リフレクション以外の）コードが実行時に必要な情報に高速にアクセスできること。
- コンパイル時にコード生成に必要な情報への簡便なアクセスを提供すること。
- ガベージコレクタ / スタックウォーカーが、ロックを取得したりメモリを割り当てたりせずに必要な情報にアクセスできること。
- 最小限の型だけが一度に読み込まれること。
- 型の読み込み時に、その型の最小限の部分だけが読み込まれること。
- 型システムのデータ構造を NGEN イメージに保存できること。

💡 初心者向け補足

NGEN (Native Image Generator) は、.NET アセンブリを事前にネイティブコードにコンパイルしておく仕組みです（現在は ReadyToRun が後継）。型システムのデータをこのイメージに保存できるようにすることで、アプリケーションの起動時間を短縮できます。

非目標

- メタデータ内のすべての情報が CLR のデータ構造に直接反映されること。
- リフレクションのすべての使用が高速であること。

実行時アルゴリズムの典型的な設計 — キャストアルゴリズム

キャスト（型変換）アルゴリズムは、マネージドコードの実行中に頻繁に使用される型システムのアルゴリズムの典型です。

このアルゴリズムには少なくとも 4 つの異なるエントリポイントがあります。各エントリポイントは、可能な限り最高のパフォーマンスを達成するために、異なるファストパスを提供するように選ばれています。

- オブジェクトを特定の、型等価でなく配列でない型にキャストできるか？
- オブジェクトをジェネリック変性を実装しないインターフェース型にキャストできるか？
- オブジェクトを配列型にキャストできるか？

4. ある型のオブジェクトを任意の他のマネージ型にキャストできるか？

最後のもの以外の各実装は、完全に汎用的でないことを代償に、より高速に動作するように最適化されています。

💡 初心者向け補足

C# で `(string)obj` や `obj is string` のようなキャストを行うと、内部的にはこのキャストアルゴリズムが呼び出されます。よくあるケースを高速に処理するために、専用の最適化された経路が複数用意されています。

例えば、「親の型にキャストできるか」の確認は、「特定の型等価でない非配列型へのキャスト」の変形であり、単方向リンクリストを辿る单一のループで実装されています。これは可能なキャスト操作のサブセットしか検索できませんが、キャストが強制しようとしている型を調べることで、それが適切なセットかどうかを判定できます。このアルゴリズムは JIT ヘルパー `JIT_ChrkCastClass_Portable` に実装されています。

前提:

- 専用目的のアルゴリズム実装は、一般的にパフォーマンスの向上をもたらします。
- アルゴリズムの追加バージョンは、克服できないメンテナンス上の問題を引き起こしません。

型システムにおける典型的な検索アルゴリズムの設計

型システムには、共通のパターンに従うアルゴリズムが多数あります。

型システムは型を見つけるために一般的に使用されます。これは JIT、リフレクション、シリアル化、リモーティングなど、さまざまな入力によってトリガーされます。

これらのケースにおける型システムへの基本的な入力は以下のとおりです：

- 検索を開始するコンテキスト（Module またはアセンブリのポインタ）
- 初期コンテキストにおいて目的の型を記述する識別子。通常はトークン、またはアセンブリが検索コンテキストの場合は文字列

アルゴリズムはまず識別子をデコードする必要があります。

型検索のシナリオでは、トークンは TypeDef トークン、TypeRef トークン、TypeSpec トークン、または文字列のいずれかです。これらの異なる識別子は、それぞれ異なる形式の検索を引き起こします。

- TypeDef トークンは、Module の RidMap での検索を引き起こします。これは単純な配列インデックスです。
- TypeRef トークンは、この TypeRef トークンが参照するアセンブリを見つけるための検索を引き起こし、その後、見つかったアセンブリポインタと TypeRef テーブルから取得した文字列を使用して型検索アルゴリズムが再度開始されます。
- TypeSpec トークンは、型を見つけるためにシグネチャを解析する必要があることを示します。シグネチャを解析して型の読み込みに必要な情報を取得します。これにより、さらなる型検索が再帰的にトリガーされます。
- 名前（文字列）は、アセンブリ間のバインディングに使用されます。TypeDef / ExportedTypes テーブルで一致するものが検索されます。注：この検索はマニフェストモジュールオブジェクトのハッシュテーブルによって最適化されています。

💡 初心者向け補足

メタデータトークンとは、.NET アセンブリ (DLL/EXE) のメタデータ内で型・メソッド・フィールドなどを一意に識別する 32 ビットの値です。TypeDef は同じモジュール内の型定義、TypeRef は別モジュールの型への参照、TypeSpec はジェネリック型のインスタンス化など複雑な型を表します。

この設計から、型システムの検索アルゴリズムの共通的な特徴がいくつか明らかになります：

- 検索はメタデータと密接に結合した入力を使用します。特に、メタデータトークンや文字列名が一般的に受け渡されます。また、これらの検索は Module に紐づけられており、Module は .dll ファイルや .exe ファイルに直接対応します。
- パフォーマンス向上させるためにキャッシュされた情報を使用します。RidMap やハッシュテーブルは、これらの検索を改善するために最適化されたデータ構造です。
- アルゴリズムは通常、入力に基づいて 3~4 つの異なるパスを持ちます。

この一般的な設計に加えて、いくつかの追加要件が重ねられています：

- 前提: 既に読み込まれている型の検索は、GC で停止中に実行しても安全です。
- 不变条件: 既に読み込まれた型は、検索すれば必ず見つかります。
- 課題: 検索ルーチンはメタデータの読み取りに依存しています。これは一部のシナリオでは不十分なパフォーマンスをもたらす可能性があります。

この検索アルゴリズムは、JIT 中に使用されるルーチンの典型です。以下のような共通の特徴があります：

- メタデータを使用します。
- 多くの場所でデータを探す必要があります。
- データ構造におけるデータの重複は比較的少ないです。
- 通常、深い再帰やループはありません。

これにより、IL ベースの JIT で作業するために必要なパフォーマンス要件と特性を満たすことができます。

ガベージコレクタが型システムに求める要件

ガベージコレクタは、GC ヒープに割り当てられた型のインスタンスに関する情報を必要とします。これは、すべてのマネージドオブジェクトの先頭にある型システムデータ構造 (MethodTable) へのポインタを通じて行われます。MethodTable には、型のインスタンスの GC レイアウトを記述するデータ構造が付属しています。このレイアウトには 2 つの形式があります（通常の型およびオブジェクト配列用と、値型の配列用）。

💡 初心者向け補足

GC ヒープ上のすべてのオブジェクトには、先頭に MethodTable へのポインタが格納されています。GC はこのポインタを使って、そのオブジェクトがどの型のインスタンスであるかを識別し、オブジェクト内のどのフィールドが他のオブジェクトへの参照を持つかを判断します。

- 前提: 型システムのデータ構造は、その型システムデータ構造で記述された型のマネージドオブジェクトよりも長い寿命を持ちます。
- 要件: ガベージコレクタには、ランタイムがサスPENDされた状態でスタックウォーカーを実行する要件があります。これについては次のセクションで説明します。

StackWalker が型システムに求める要件

StackWalker / GC StackWalker は、以下の 2 つのケースで型システムの入力を必要とします：

- Stack 上の値型のサイズを調べるため。
- Stack 上の値型内の報告すべき GC ルートを見つけるため。

型の遅延読み込みの要求や、(関連する GC 情報のみが異なる) コードの複数バージョン生成の回避に関するさまざまな理由から、CLR は現在、Stack 上にあるメソッドのシグチャのウォーキングを要求しています。この必要性はめったに発生しませんが (StackWalker が非常に特定のタイミングで実行される必要があるため)、信頼性の目標を達成するために、シグチャウォーカーは StackWalker 中に機能できなければなりません。

StackWalker はおよそ 3 つのモードで実行されます：

1. セキュリティまたは例外処理の理由で、現在のスレッドの Stack をウォークする場合。
2. GC のために全スレッドの Stack をウォークする場合 (全スレッドが EE によってサスペンドされている)。
3. プロファイラーのために特定のスレッドの Stack をウォークする場合 (そのスレッドがサスペンドされている)。

GC StackWalker の場合およびプロファイラーによる StackWalker の場合は、スレッドのサスペンションにより、メモリを割り当てたりほとんどのロックを取得したりすることは安全ではありません。

初心者向け補足

StackWalker とは、スレッドの呼び出し Stack を順番に辿っていく仕組みです。GC がオブジェクトの参照を正しく追跡するためには、Stack 上にある値型の中にもオブジェクトへの参照 (GC ルート) がないかを調べる必要があります。しかし GC の実行中は他のスレッドが停止しているため、ロックの取得やメモリの割り当てが許されない厳しい制約の下で動作しなければなりません。

のことから、型システムには上記の要件に従うことが保証できる経路が開発されています。

型システムがこの目標を達成するために必要なルールは以下のとおりです：

- メソッドが呼び出された場合、そのメソッドのすべての値型パラメータは、プロセス内のいずれかの AppDomain に読み込まれている必要があります。
- シグチャを持つアセンブリから型を実装するアセンブリへのアセンブリ参照は、StackWalker の一部としてシグチャのウォーキングが必要になる前に解決されている必要があります。

これは、型ローダー、NGEN イメージ生成プロセス、および JIT 内の広範かつ複雑な一連の強制によって実現されています。

- 課題: 型システムに対する StackWalker の要件は非常に脆弱です。
- 課題: 型システムにおける StackWalker 要件の実装は、読み込み済みの型を検索中に触れる可能性のある型システムのすべての関数に、一連のコントラクト違反を必要とします。
- 課題: 実行されるシグチャのウォークは、通常のシグチャウォーキングコードで行われます。このコードはシグチャをウォークしながら型を読み込むように設計されていますが、このケースでは実際には型の読み込みがトリガーされないという前提で型の読み込み機能が使用されます。
- 課題: StackWalker の要件は、型システムだけでなくアセンブリローダーからのサポートも必要とします。ローダーは、型システムのこのニーズを満たすにあたって多くの問題を抱えています。

静的変数

CoreCLR における静的変数は、「静的ベース」の取得と、そこからオフセットを加算して実際の値へのポインタを得るという組み合わせによって処理されます。静的ベースは、各フィールドに対して非 GC または GC のいずれかとして定義されます。現在、非 GC 静的変数はプリミティブ型（byte、sbyte、char、int、uint、long、ulong、float、double、各種ポインタ）および列挙型で表される静的変数です。GC 静的変数はクラスまたは非プリミティブ値型で表される静的変数です。GC 静的変数である値型の静的変数の場合、静的変数は実際にはその値型のボックス化されたインスタンスへのポインタです。

型ごとの静的変数情報

.NET 9 以降、静的変数のベースはすべてそれぞれの型に関連付けられるようになりました。静的変数のデータは、[MethodTable](#) を起始として、[DynamicStaticsInfo](#) を取得して静的変数ポインタを得るか、[ThreadStaticsInfo](#) を取得して TLSIndex を得て、そこからスレッド静的変数システムを使用して実際のスレッド静的変数ベースを取得することでアクセスできます。

上記の構造では、非 GC と GC の静的変数、およびスレッド静的変数と通常の静的変数に対して別々のフィールドがあります。通常の静的変数については、ポインタサイズのフィールドを 1 つ使用し、クラスコンストラクタが実行されたかどうかをエンコードしています。これは、静的フィールドのアドレス取得とクラスコンストラクタのトリガーが必要かどうかの判定の両方を、ロックフリーなアトミックアクセスで行えるようにするためにです。TLS 静的変数については、クラスコンストラクタの実行済み検出はスレッド静的変数インフラストラクチャの一部として説明される、より複雑なプロセスです。[DynamicStaticsInfo](#) および [ThreadStaticsInfo](#) 構造体はロックなしでアクセスされるため、メモリ順序のティアリング問題を避けるために、これらの構造体のフィールドへのアクセスが単一のメモリアクセスで行えることが重要です。

また、ジェネリック型の場合、各フィールドには型インスタンスごとに割り当てられる [FieldDesc](#) があり、複数の正規インスタンス間で共有されません。

コレクティブルアセンブリの静的変数のライフタイム管理

CoreCLR ランタイムにはコレクティブルアセンブリ（収集可能なアセンブリ）の概念があるため、静的変数のライフタイム管理が必要です。選択されたアプローチは、ランタイムのデータ構造が GC ヒープ上のマネージドオブジェクトの内部へのポインタを持つように、特殊な GC ハンドル型を構築することです。

ここで動作要件は、静的変数が自身のコレクティブルアセンブリを存続させてはならないということです。そのため、コレクティブル静的変数は、コレクティブルアセンブリが最終的に収集される前に存在してファイナライズされ得るという特殊な性質を持っています。復活シナリオがある場合、これは非常に予想外の動作につながる可能性があります。

スレッド静的変数

スレッド静的変数は、静的変数を含む型のライフタイムと、静的変数がアクセスされるスレッドのライフタイムのうち、短い方として定義されるライフタイムを持つ静的変数です。型の静的変数に [\[System.Runtime.CompilerServices.ThreadStaticAttribute\]](#) を付与することで作成されます。一般的な仕組みとしては、すべてのスレッドで同一の「インデックス」を型に割り当て、各スレッドではこのインデックスによって効率的にアクセスできるデータ構造を保持するというものです。

 初心者向け補足

スレッド静的変数は `[ThreadStatic]` 属性が付いた静的フィールドで、スレッドごとに独立した値を持ちます。例えば、各スレッドが独自のキャッシュやコンテキスト情報を保持する場合に使用されます。通常の静的変数はプロセス全体で共有されますが、スレッド静的変数はスレッドごとに別々のストレージが確保されます。

このアプローチにはいくつかの特殊な点があります：

- コレクティブルと非コレクティブルのスレッド静的変数を分離しています（`TLSIndexType::NonCollectible` と `TLSIndexType::Collectible`）。
- ネイティブ CoreCLR コードとマネージドコードの間で非 GC スレッド静的変数を共有する機能を提供しています（`TLSIndexType::DirectOnThreadLocalData` のサブセット）。
- 少数の非 GC スレッド静的変数に対して、非常に効率的なアクセス手段を提供しています（`TLSIndexType::DirectOnThreadLocalData` の残りの用途）。

スレッド静的変数のアクセスパターン

`DirectOnThreadLocalData` でないスレッド静的変数に対して JIT が使用するパターンは以下のとおりです：

- 何らかの方法で TLS インデックスを取得する。
- 現在のスレッドの OS 管理 TLS ブロックへの TLS ポインタを取得する。すなわち `pThreadLocalData = &t_ThreadStatics`。
- 整数値を 1 つ読み取る：`pThreadLocalData->cCollectibleTlsData` または `pThreadLocalData->cNonCollectibleTlsData`。
- `cTlsData` を検索しようとしているインデックスと比較する：`if (cTlsData < index.GetIndexOffset())`。
- インデックスが範囲外の場合、ステップ 11 にジャンプする。
- TLS ブロックからポインタ値を 1 つ読み取る：`pThreadLocalData->pCollectibleTlsArrayData` または `pThreadLocalData->pNonCollectibleTlsArrayData`。
- TLS 配列内からポインタを 1 つ読み取る：`pTLSBaseAddress = *((intptr_t*)(((uint8_t*)pTlsArrayData) + index.GetIndexOffset()))`。
- ポインタが NULL の場合、ステップ 11 にジャンプする。
- TLS インデックスが Collectible インデックスでない場合、`pTLSBaseAddress` を返す。
- `ObjectFromHandle((OBJECTHANDLE)pTLSBaseAddress)` が NULL の場合、ステップ 11 にジャンプする。
- `ObjectFromHandle((OBJECTHANDLE)pTLSBaseAddress)` を返す。
- ヘルパーをテールコールする：`return GetThreadLocalStaticBase(index)`。

`DirectOnThreadLocalData` 上のスレッド静的変数に対して JIT が使用するパターンは以下のとおりです：

- 何らかの方法で TLS インデックスを取得する。
- 現在のスレッドの OS 管理 TLS ブロックへの TLS ポインタを取得する。すなわち `pThreadLocalData = &t_ThreadStatics`。
- ThreadLocalData 構造体の先頭にインデックスオフセットを加算する：`pTLSBaseAddress = ((uint8_t*)pThreadLocalData) + index.GetIndexOffset()`。

スレッド静的変数のライフタイム管理

効率性のために、コレクティブルと非コレクティブルのスレッド静的変数を区別しています。

非コレクティブルなスレッド静的変数は、ランタイムによって収集できない型に定義されたスレッド静的変数です。これは実際に観測されるほとんどのスレッド静的変数に該当します。`DirectOnThreadLocalData` 静的変数は、特別に最適化された形式を持ち GC 報告を必要としない、このカテゴリのサブセットです。非コレクティブルスレッド静的変数の場合、`ThreadLocalData` 内のポインタ (`pNonCollectibleTlsArrayData`) は、`object[]`、`byte[]`、または `double[]` 配列を指すマネージド `object[]` へのポインタです。GC スキヤン時には、最初の `object[]` へのポインタだけを GC に報告すればよいです。

コレクティブルなスレッド静的変数は、ランタイムによって収集可能な型に定義されたスレッド静的変数です。`ThreadLocalData` 内のポインタ (`pCollectibleTlsArrayData`) は `malloc` で割り当てられたメモリチャンクへのポインタであり、`object[]`、`byte[]`、または `double[]` 配列へのポインタを保持します。GC スキヤン時には、型とスレッドの両方がまだ生きている場合のみ、各マネージドオブジェクトを個別に存続させる必要があります。これにはいくつかの状況を適切に処理する必要があります：

1. コレクティブルアセンブリが参照されなくなっても、それに関連するスレッド静的変数にファイナライザがある場合、オブジェクトはファイナライゼーションキューに移動する必要があります。
2. コレクティブルアセンブリに関連するスレッド静的変数が一連のオブジェクト参照を通じてコレクティブルアセンブリの `LoaderAllocator` を参照している場合、コレクティブルアセンブリが参照されていると見なされる理由を提供してはなりません。
3. コレクティブルアセンブリが収集された場合、関連する静的変数はもはや存在せず、そのコレクティブルアセンブリに関連する TLSIndex 値は再利用可能になります。
4. スレッドが実行を終了した場合、そのスレッドに関連するすべてのスレッド静的変数は存続させません。

選択されたアプローチは、2 つの異なるハンドル型を使用することです。効率的なアクセスのために、動的に調整される配列に格納されるハンドル型は `WeakTrackResurrection GCHandle` です。このハンドルインスタンスは正確なインスタンス化ではなく TLS データ内のスロットに関連付けられているため、関連するコレクティブルアセンブリが収集された後にスロットが再利用される場合に再利用できます。さらに、使用中の各スロットにはオブジェクトを `LoaderAllocator` が解放されるまで存続させる `LOADERHANDLE` があります。この `LOADERHANDLE` は `LoaderAllocator` が収集された場合に放棄されますが、`LOADERHANDLE` は `LoaderAllocator` が収集されない場合にのみクリーンアップが必要なため問題ありません。スレッドの破棄時には、TLS 配列内の各コレクティブルスロットについて、正しい `LoaderAllocator` 上の `LOADERHANDLE` を明示的に解放します。

主要なデータ構造

MethodTable

`MethodTable` は、型システムにおける中心的な実行時データ構造です。すべてのマネージドオブジェクトの先頭にはこのデータ構造へのポインタが格納されており、ランタイムが型に関する情報に素早くアクセスするために必要なすべての情報を含んでいます。

💡 初心者向け補足

`MethodTable` は型ごとに 1 つだけ存在し、その型のすべてのインスタンス（オブジェクト）から共有されます。例えば `string` 型のオブジェクトが 1000 個あっても、`MethodTable` は 1 つだけです。各オブジェクトの先頭にある `MethodTable` ポインタが、GC やキャストなどの操作で型を識別するために使用されます。

`MethodTable` には以下のようないい情報が含まれます：

- 親の型（基底クラス）への参照
- 実装するインターフェースのリスト
- 仮想メソッドテーブル（v-table）

- GC レイアウト情報
- 型のサイズやフラグなどの基本情報

EEClass

EEClass は、型の読み込み、JIT コンパイル、リフレクションに必要だが、実行時の高速パスでは不要な「コールド」データを格納するデータ構造です。

💡 初心者向け補足

「コールド」データとは、アクセス頻度の低いデータのことです。MethodTable にすべての情報を詰め込むとメモリ効率が悪くなるため、アクセス頻度の低い情報は EEClass に分離されています。同じジェネリック型の異なるインスタンス化（例：`List<int>` と `List<string>`）は、EEClass を共有する場合があります。

MethodDesc

MethodDesc は個々のメソッドを記述するデータ構造です。メソッドのメタデータトークン、メソッドの属性、ネイティブコードへのエントリーポイントなどの情報を含みます。

FieldDesc

FieldDesc は個々のフィールドを記述するデータ構造です。フィールドのオフセット、型、属性などの情報を含みます。

TypeDesc

TypeDesc は MethodTable だけでは表現できない型を記述するために使用されるデータ構造です。具体的には以下の型が含まれます：

- **ByRef** 型: 参照渡しパラメータ (`ref int` など)
- ポインタ型: アンマネージドポインタ (`int*` など)
- 関数ポインタ型: 関数ポインタ
- ジェネリック型変数: `T` や `U` などのジェネリックパラメータ

ClassLoader

ClassLoader は型を読み込むために使用されるコンポーネントです。メタデータから型情報を読み取り、上記のデータ構造を生成します。詳細は [型ローダー](#) の章を参照してください。

物理アーキテクチャ

型システムの主要部分は以下のソースファイルに含まれています：

- `Class.cpp/inl/h` – EEClass の関数、および Build Method Table
- `MethodTable.cpp/inl/h` – Method Table の操作関数
- `TypeDesc.cpp/inl/h` – TypeDesc の検査関数
- `MetaSig.cpp / SigParser` – シグネチャ関連のコード
- `FieldDesc / MethodDesc` – これらのデータ構造の検査関数
- `Generics` – ジェネリクス固有のロジック
- `Array` – 配列処理に必要な特殊ケースの処理コード
- `VirtualStubDispatch.cpp/h/inl` – Virtual Stub Dispatch のコード
- `VirtualCallStubCpu.hpp` – Virtual Stub Dispatch のプロセッサ固有コード
- `threadstatics.cpp/h` – スレッド静的変数の処理

主要なエントリポイントは `BuildMethodTable` 、 `LoadTypeHandleThrowing` 、 `CanCastTo*` 、
`GetMethodDescFromMemberDefOrRefOrSpecThrowing` 、 `GetFieldDescFromMemberRefThrowing` 、 `CompareSigs` 、 および
`VirtualCallStubManager::ResolveWorkerStatic` です。

関連資料

- [ECMA CLI 仕様](#)
- [型ローダー](#) — Book of the Runtime の章
- [仮想スタブディスパッチ](#) — Book of the Runtime の章
- [メソッドディスクリプタ](#) — Book of the Runtime の章

型ローダーの設計

原文

この章の原文は [Type Loader Design](#) です。

著者: Ladi Prosek - 2007

はじめに

クラスベースのオブジェクト指向システムでは、型 (type) は個々のインスタンスが含むデータと提供する機能を記述するテンプレートです。型を定義せずにオブジェクトを作成することはできません¹。2つのオブジェクトが同じ型であるとは、同じ型のインスタンスである場合に限ります。まったく同じメンバーのセットを定義していても、同じ型のインスタンスでなければ、それらは何の関連性も持ちません。

前述の内容は、典型的な C++ のシステムにも当てはまります。CLR に不可欠な追加機能は、完全なランタイム型情報の利用可能性です。マネージドコードを「管理」し型安全な環境を提供するために、ランタイムは任意の時点で任意のオブジェクトの型を知る必要があります。型の同一性クエリは頻繁に行われることが予想されるため（例：すべての型キャストはキャストが安全かつ実行可能かを検証するため型の同一性を問い合わせます）、このような型情報は大規模な計算なしに即座に利用可能でなければなりません。

初心者向け補足

「型ローダー」は、プログラムで使われる型（クラスや構造体など）の情報をメモリ上に構築するコンポーネントです。C# で `new MyClass()` と書くと、裏側では型ローダーが `MyClass` の情報（メソッド、フィールド、継承関係など）をメモリ上に準備しています。Java の「クラスローダー」に似た役割ですが、CLR ではクラスだけでなく値型も含むすべての型を扱います。

このパフォーマンス要件により、辞書検索のようなアプローチは除外され、以下の高レベルアーキテクチャが導かれます。

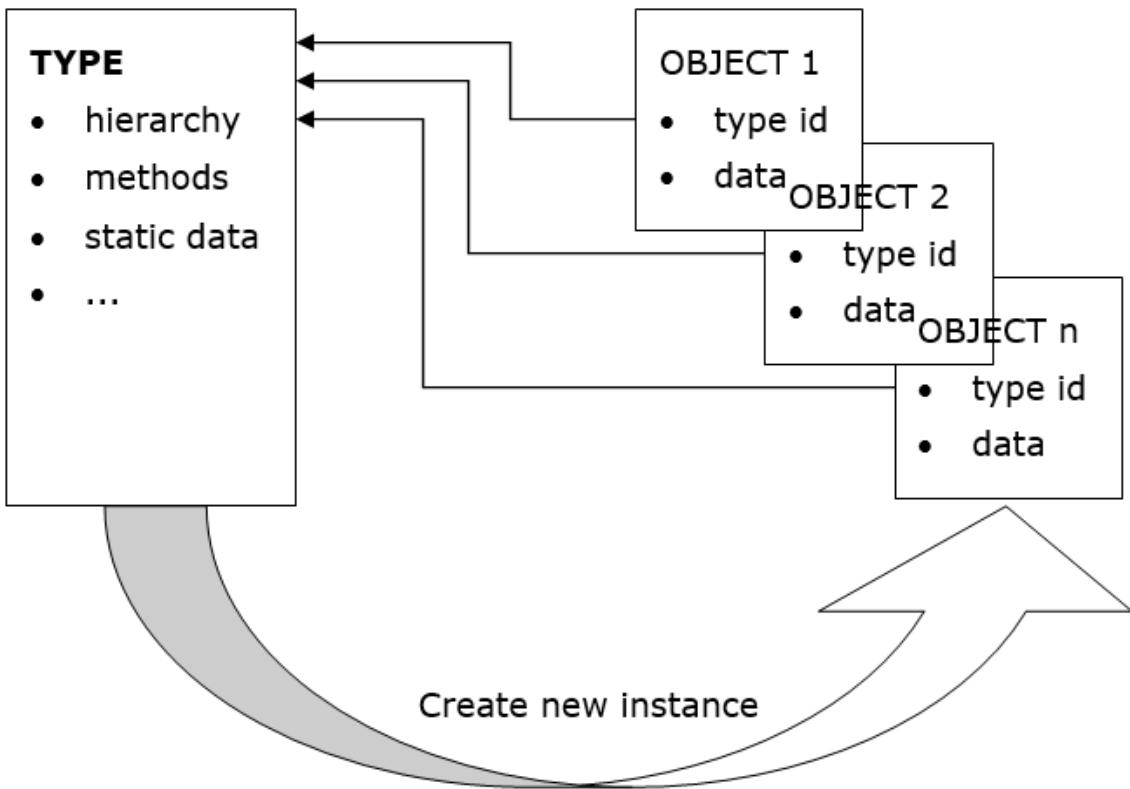


図1 抽象的な高レベルオブジェクト設計

実際のインスタンスデータの他に、各オブジェクトは型を表す構造体への単純なポインタである型 ID を含みます。この概念は C++ の v-table ポインタに似ていますが、ここで TYPE と呼び後ほどより正確に定義する構造体は、v-table 以上のものを含みます。例えば、「isa」包含関係の質問に答えるために階層に関する情報を含む必要があります。

¹ C# 3.0 の「匿名型 (anonymous types)」機能では、型への明示的な参照なしに、フィールドを直接列挙するだけでオブジェクトを定義できます。しかしこれに惑わされないでください。実際にはコンパイラが裏側で型を作成しています。

1.1 関連資料

[1] Martin Abadi, Luca Cardelli, *A Theory of Objects*, ISBN 978-0387947754

[2] Andrew Kennedy ([@andrewjkennedy](#)), Don Syme ([@dsyme](#)), [Design and Implementation of Generics for the .NET Common Language Runtime](#)

[3] [ECMA Standard for the Common Language Infrastructure \(CLI\)](#)

1.2 設計目標

型ローダー（クラスローダーと呼ばれることもありますが、厳密には正しくありません。クラスは型のサブセット、すなわち参照型のみを指し、ローダーは値型もロードするためです）の究極の目的は、ロードを要求された型を表すデータ構造を構築することです。型ローダーが持つべき特性は以下の通りです：

- 高速な型検索: [モジュール, トークン] → ハンドル、および [アセンブリ, 名前] → ハンドルの検索が高速であること。
- 最適化されたメモリレイアウト: ワーキングセットサイズ、キャッシュヒット率、JIT コンパイル済みコードの性能が良好であること。
- 型安全性: 不正な形式の型はロードされず、`TypeLoadException` がスローされること。
- 並行性 (concurrency): マルチスレッド環境で良好にスケールすること。

2型ローダーのアーキテクチャ

型ローダーへのエントリポイントは比較的少数です。各エントリポイントのシグネチャはわずかに異なりますが、いずれも同様のセマンティクスを持ちます。メタデータトークンまたは名前文字列の形式で型/メンバーの指定、トークンのスコープ（モジュールまたはアセンブリ）、およびフラグなどの追加情報を受け取ります。ロードされたエンティティはハンドルの形式で返されます。

💡 初心者向け補足

メタデータトークン (metadata token) とは、.NET のアセンブリ (DLL/EXE) 内で型やメソッドを識別するための番号のことです。JIT コンパイラがコードをコンパイルする際、IL コード中のトークンを型ローダーに渡して、実際の型情報を取得します。Java でいえば、クラスファイル内の「コンスタントプール」のインデックスに似た概念です。

JIT コンパイル中には通常、型ローダーへの多くの呼び出しが発生します。以下の例を考えてみましょう：

```
object CreateClass()
{
    return new MyClass();
}
```

csharp

IL 内で `MyClass` はメタデータトークンを使って参照されます。実際のインスタンス化を処理する `JIT_New` ヘルパーへの呼び出しを生成するために、JIT は型ローダーに型のロードとハンドルの返却を要求します。このハンドルは JIT コンパイルされたコードに即値 (immediate value) として直接埋め込まれます。型やメンバーが通常、実行時ではなく JIT 時に解決・ロードされるという事実は、以下のようなコードで遭遇しやすい、時に混乱を招く動作を説明しています：

```
object CreateClass()
{
    try {
        return new MyClass();
    } catch (TypeLoadException) {
        return null;
    }
}
```

csharp

もし `MyClass` がロードに失敗した場合（例えば、別のアセンブリで定義されるはずが最新のビルドで誤って削除された場合）、このコードは依然として `TypeLoadException` をスローします。catch ブロックがキャッチしなかった理由は、そもそも実行されなかったからです！例外は JIT コンパイル中に発生し、`CreateClass` を呼び出して JIT コンパイルを引き起こしたメソッドでのみキャッチ可能です。さらに、インライン化により JIT コンパイルがどの時点でトリガーされるかは必ずしも明らかではないため、ユーザーは決定論的な動作を期待したり依存したりすべきではありません。

💡 初心者向け補足

上記の `try-catch` の例は重要なポイントです。JIT コンパイルはメソッドが初めて呼び出されるタイミングで行われます。つまり、`new MyClass()` の型ロードエラーはメソッドの実行時ではなく JIT コンパイル時に発生するため、メソッド内の `catch` ブロックでは捕捉できません。Java のクラスローダーと異なり、.NET の型ロードは JIT コンパイルの一部として事前に行われることに注意してください。

主要データ構造

CLR における最も普遍的な型の指定は `TypeHandle` です。`TypeHandle` は、`MethodTable`（`System.Object` や `List<string>` のような「通常の」型を表す）または `TypeDesc`（`byref`、`ポインタ`、`関数ポインタ`、`ジェネリック変数`を表す）のいずれかへのポインタをカプセル化する抽象的なエンティティです。`TypeHandle` は型の同一性 (identity) を構成し、2つのハンドルが等しいのはそれらが同じ型を表す場合に限ります。領域を節約するため、`TypeHandle` が `TypeDesc` を含むという事実は、追加のフラグを使う代わりにポインタの最下位から2番目のビットを1に設定する（すなわち `(ptr | 2)`）ことで示されます²。`TypeDesc` は「抽象的」であり、以下の継承階層を持ちます。

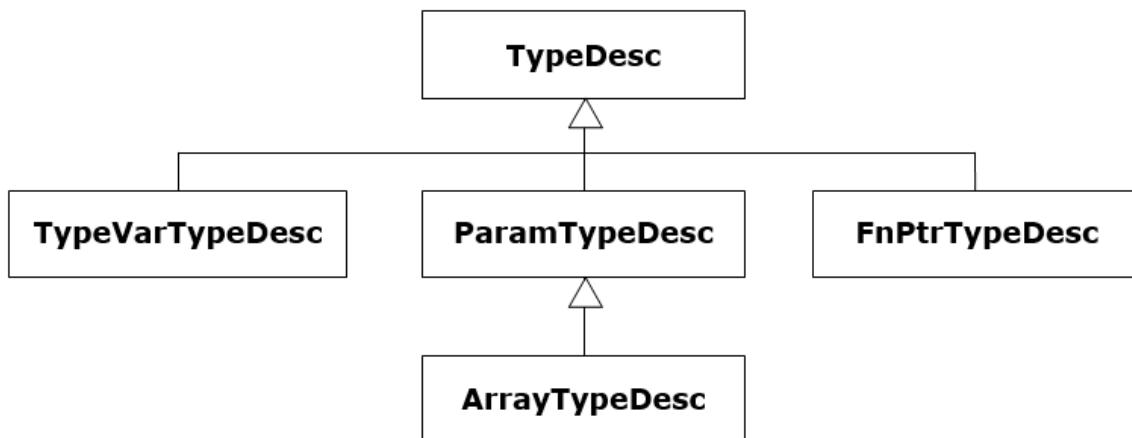


図2 TypeDesc の階層

TypeDesc

抽象的な型記述子 (type descriptor)。具象的な記述子の種類はフラグによって決定されます。

TypeVarTypeDesc

型変数を表します。すなわち `List<T>` の `T` や `Array.Sort<T>` の `T` です（後述のジェネリクスの項を参照）。型変数は複数の型やメソッド間で共有されることなく、各変数にはただ1つの所有者があります。

FnPtrTypeDesc

関数ポインタ (function pointer) を表します。本質的には戻り値の型とパラメータを参照する型ハンドルの可変長リストです。元々はマネージド C++ でのみ使用されていましたが、C# 9 以降では C# でもサポートされています。

ParamTypeDesc

`byref` 型とポインタ型を表す記述子です。`byref` は C# のメソッドパラメータに適用される `ref` および `out` キーワードの結果であり³、ポインタ型はアンセーフ C# およびマネージド C++ で使用されるデータへのアンマネージドポインタです。

MethodTable

これはランタイムの中心的なデータ構造です。上記のカテゴリのいずれにも該当しない型を表します（プリミティブ型、および「オープン」と「クローズド」の両方のジェネリック型を含みます）。親型、実装インターフェース、v-table など、迅速な検索が必要な型に関するすべての情報を含みます。

EEClass

`MethodTable` のデータは、ワーキングセットとキャッシュの利用効率を向上させるために「ホット」と「コールド」の構造体に分割されています。`MethodTable` 自体はプログラムの定常状態で必要な「ホット」データのみを格納することを意図しています。`EEClass` は型ロード、JIT コンパイル、またはリフレクションでのみ一般的に必要とされる「コールド」データを格納します。各 `MethodTable` は1つの `EEClass` を指します。

さらに、`EEClass` はジェネリック型間で共有されます。複数のジェネリック型の `MethodTable` が1つの `EEClass` を指すことが可能です。この共有により、`EEClass` に格納できるデータに追加の制約が生じます。

💡 初心者向け補足

`MethodTable` と `EEClass` の分離は、パフォーマンスのための設計パターンです。頻繁にアクセスされる「ホット」なデータ（v-table、インターフェースマップなど）と、たまにしかアクセスされない「コールド」なデータ（リフレクション情報など）を分けることで、CPU キャッシュの効率を向上させています。この「ホット/コールド分離」は、データベースのインデックス設計や Web アプリケーションのキャッシュ戦略と同じ発想です。

MethodDesc

この構造体がメソッドを記述することは驚くに値しません。実際にはいくつかのバリエーションがあり、対応する `MethodDesc` のサブタイプが存在しますが、そのほとんどはこのドキュメントの範囲外です。ジェネリクスにおいて重要な役割を果たす `InstantiatedMethodDesc` というサブタイプが1つあることだけを述べておきます。詳細については [メソッド記述子の設計](#) を参照してください。

FieldDesc

`MethodDesc` と同様に、この構造体はフィールドを記述します。特定の COM 相互運用 (COM interop) シナリオを除いて、EE はプロパティやイベントをまったく気にしません。なぜならそれは結局のところメソッドとフィールドに帰着し、コンパイラとリフレクションが構文糖 (syntactic sugar) のような体験を提供するためにそれらを生成・解釈するだけだからです。

² これはデバッグに有用です。`TypeHandle` の値が 2、6、A、または E で終わる場合、それは `MethodTable` ではなく、`TypeDesc` を正常に検査するためには余分なビットをクリアする必要があります。

³ `ref` と `out` の違いはパラメータ属性にあるだけです。型システムに関する限り、それらは両方とも同じ型です。

2.1 ロードレベル

型ローダーが指定された型のロードを要求されたとき（例えば `typedef/typeref/typespec` トークンと モジュール によって識別される型）、すべての作業を一度にアトミックに行うわけではありません。ロードは段階的に行われます。その理由は、型は通常他の型に依存し

ており、他の型から参照される前に完全にロードされることを要求すると、無限再帰やデッドロックが発生するためです。以下の例を考えてみましょう：

```
csharp
class A<T> : C<B<T>>
{ }

class B<T> : C<A<T>>
{ }

class C<T>
{ }
```

これらは有効な型であり、明らかに `A` は `B` に依存し、`B` は `A` に依存しています。

ローダーは最初に型を表す構造体を作成し、他の型をロードせずに取得できるデータで初期化します。この「依存関係なし」の作業が完了すると、構造体は他の場所から参照可能になります（通常は他の構造体にポインタを挿入することによって）。その後、ローダーはインクリメンタルなステップで進行し、構造体にますます多くの情報を追加して、最終的に完全にロードされた型に到達します。上記の例では、`A` と `B` の基底型は最初に他の型を含まない近似で置き換えられ、後で本物に置換されます。

正確な半ロード状態は、いわゆるロードレベル (load level) によって記述されます。`CLASS_LOAD_BEGIN` から始まり、`CLASS_LOADED` で終わり、間にいくつかの中間レベルがあります。個々のロードレベルに関する豊富で有用なコメントが [classloadlevel.h](#) ソースファイルにあります。

ロードレベルのより詳細な説明については、[Design and Implementation of Generics for the .NET Common Language Runtime](#) を参照してください。

💡 初心者向け補足

ロードレベルは、型の「準備状態」を段階的に表す仕組みです。型同士が相互に依存する場合（例：`A` が `B` を継承し、`B` も `A` を参照する）、すべてを一度にロードすると無限ループに陥ります。そのため、CLR は型を「大まかな情報だけ準備」→「詳細な情報を追加」→「完全にロード完了」と段階的に構築します。これは、ビルシステムにおける循環依存の解決と同じ考え方です。

2.1.1 型ローダー内のロードレベルの使用

型ローダー内で、型ローダーのさまざまな部分で動作している間、どのロードレベルを使用できるかについてさまざまな異なるルールが適用されます。

2.1.1.1 `ClassLoader::CreateTypeHandleForTypeDefThrowing` および `MethodTableBuilder::BuildMethodTableThrowing` 内のコード

`ClassLoader::CreateTypeHandleForTypeDefThrowing` 内のコードを `MethodTableBuilder::BuildMethodTableThrowing` の呼び出し前に実行している間、ロード中の型の `MethodTable` に依存するロジックは使用できません。これは、これらのルーチンが `MethodTable` を構築するルーチンであるという詳細に起因します。

これにはさまざまな影響がありますが、最も明白なのは、ロード中の型の基底型と関連するインターフェースまたはフィールド型を `CLASS_LOAD_APPROXPARENTS` を超えてロードすると、`TypeLoadException` をトリガーするリスクが生じるということです。例えば、基底型を `CLASS_LOAD_EXACTPARENTS` までロードした場合、`B<A>` から派生する型 `A` をロードできなくなります。このルール

の例外は存在し、型ロードプロセスを実際に実装するために必要ですが、一般的には避けるべきです。なぜならそれらは ECMA 仕様と一致しない動作を引き起こすためです。

2.1.1.2 `ClassLoader::DoIncrementalLoad` 内のコード

`DoIncrementalLoad` 中に実行されるコードは、一般的に、インクリメンタルにロードしようとしているレベルか、ロード中の型が既に到達しているレベルのいずれかへの型ロードを要求することが許可されています。ここでの区別は、型間の関係が循環的 (circular) か非循環的 (non-circular) かという点です。型パラメータとの関係のような循環的関係は、目標のロードレベルより低いレベルにのみロードできます。非循環的関係は、インクリメンタル操作が最終的に到達するロードレベルまでロードすることを要求できます。

例えば、型からその基底型への関係は非循環的です。型が推移的に自身の正確な基底型になることはできないためです。しかし、型からその基底型のインスタンス化引数への関係は循環的になります。

上記のルールの例として、`class A : B<A> {}` という型を考えてみましょう。クラス `A` を `CLASS_LOAD_EXACTPARENTS` までロードするとき、基底型 `B<A>` を `CLASS_LOAD_EXACTPARENTS` までロードすることを要求できます（非循環的関係であるため）。しかし、`B<A>` を `CLASS_LOAD_EXACTPARENTS` までロードするとき、型 `A` を `CLASS_LOAD_EXACTPARENTS` までロードすることを要求することはできません（循環性の問題が生じるため）。したがって、`B<A>` を `CLASS_LOAD_EXACTPARENTS` までロードする際には、`A` は `CLASS_LOAD_APPROXPARENTS` までしかロードを強制できません。

`ClassLoader::DoIncrementalLoad` で実行されるコードは、比較的単純なパターンに従います。コードは型が特定のロードレベルまでロードされていることに依存でき、インクリメンタルロードプロセスが特定のレベルで完了すると、ロード中の型のロードレベルがインクリメントされます。

2.1.1.3 `PushFinalLevels` 内のコード

型ロードの最後の2つのレベルは、異なるルールセットに従う `PushFinalLevels` によって処理されます。`PushFinalLevels` はレベルを上げるために、他の型が目標レベルよりも低いレベルまでロードされていることにのみ依存できるコードを実行します。しかし、型がより高いレベルに到達したとマークする前に、`PushFinalLevels` は他の型にも新しいレベルへの `PushFinalLevels` アルゴリズムの完了を要求できます。すべての型が新しいレベルに到達したことが確認された場合にのみ、型のセット全体が新しいレベルに到達したとマークできます。

2.1.2 型ローダー外でのロードレベルの使用

一般的な場合、型ローダーの一部ではないコードを操作する際にはロードレベルを単に無視し、完全にロードされた型を要求することが望ましいです。これがデフォルトであり、常に機能的に正しい選択です。ただし、パフォーマンス上の理由から、部分的にロードされた型のみを要求することも可能です。その場合、型の使用者は自分のコードが完全にロードされた状態に依存していないことを確認する必要があります。

2.2 ジェネリクス

ジェネリクスのない世界では、すべてがシンプルで問題はありません。なぜなら、すべての通常の（`TypeDesc` で表されない）型は1つの `MethodTable` を持ち、それが関連する `EEClass` を指し、`EEClass` は `MethodTable` を指し返すからです。型のすべてのインスタンスは、最初のフィールドとしてオフセット0に（すなわち参照値として見えるアドレスに）`MethodTable` へのポインタを含みます。領域を節約するため、型が宣言したメソッドを表す `MethodDesc` は、`EEClass` が指すチャンクのリンクリストに組織化されます⁴。

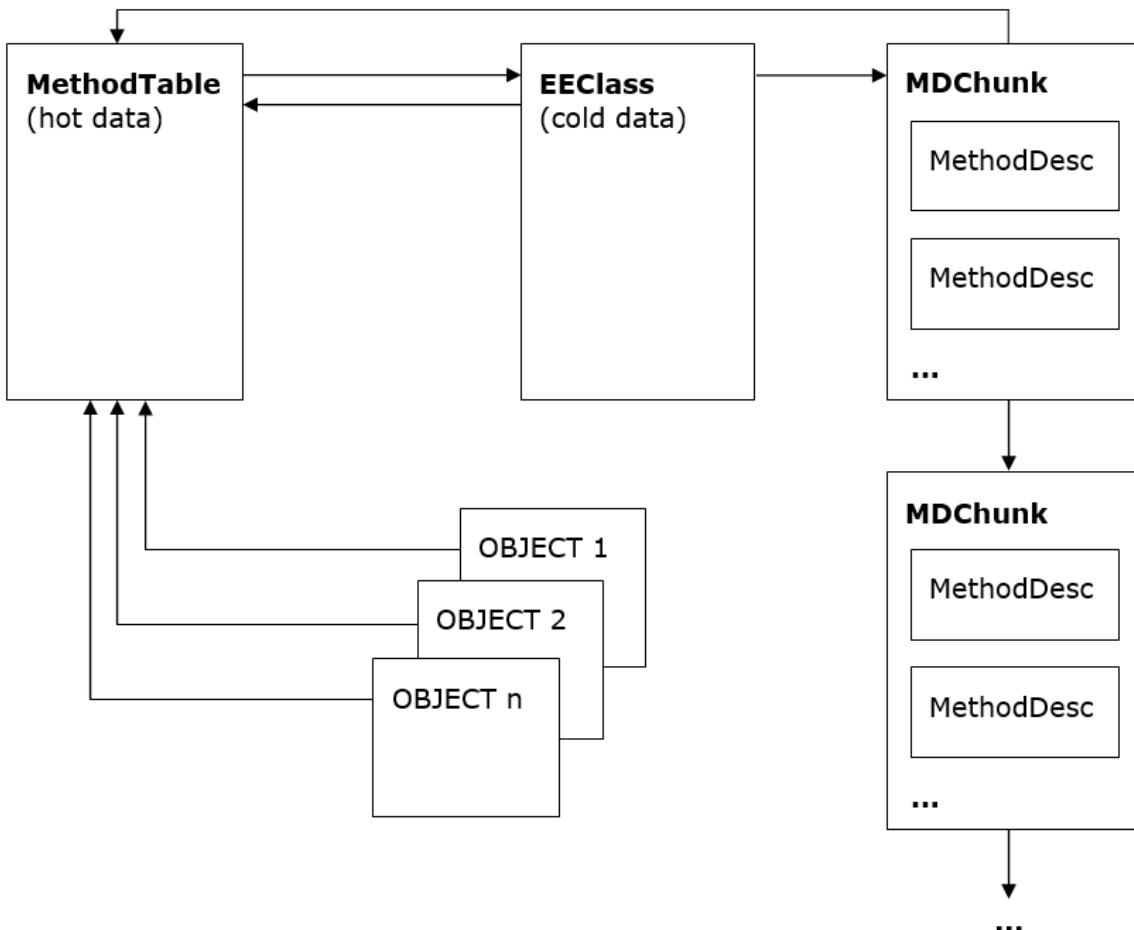


図3 非ジェネリック型と非ジェネリックメソッド

⁴もちろん、マネージドコードが実行される際、チャンク内のメソッドを検索して呼び出すわけではありません。メソッドの呼び出しは非常に「ホット」な操作であり、通常は `MethodTable` 内の情報にのみアクセスする必要があります。

2.2.1用語

ジェネリックパラメータ (Generic Parameter)

他の型で置換されるプレースホルダー。`List<T>` の宣言における `T` です。仮型パラメータ (formal type parameter) とも呼ばれます。ジェネリックパラメータには名前とオプションのジェネリック制約 (generic constraints) があります。

ジェネリック引数 (Generic Argument)

ジェネリックパラメータに置換される型。`List<int>` の `int` です。ジェネリックパラメータは引数としても使用できることに注意してください。以下を考えてみましょう：

```
List<T> GetList<T>()
{
    return new List<T>();
}
```

csharp

このメソッドには1つのジェネリックパラメータ `T` があり、これがジェネリックリストクラスのジェネリック引数として使用されています。

ジェネリック制約 (Generic Constraint)

ジェネリックパラメータがその潜在的なジェネリック引数に課すオプションの要件です。必要な特性を持たない型はジェネリックパラメータに置換できず、型ローダーによって強制されます。ジェネリック制約には3種類あります：

1. 特殊制約 (special constraints)

- 参照型制約 (reference type constraint) — ジェネリック引数は参照型（値型ではない）でなければなりません。C# ではこの制約を表現するために `class` キーワードが使用されます。

```
public class A<T> where T : class
```

csharp

- 値型制約 (value type constraint) — ジェネリック引数は `System.Nullable<T>` とは異なる値型でなければなりません。C# では `struct` キーワードが使用されます。

```
public class A<T> where T : struct
```

csharp

- デフォルトコンストラクタ制約 (default constructor constraint) — ジェネリック引数はパブリックな引数なしコンストラクタを持たなければなりません。C# では `new()` で表現されます。

```
public class A<T> where T : new()
```

csharp

2. 基底型制約 (base type constraints) — ジェネリック引数は指定された非インターフェース型から派生している（または直接その型である）必要があります。基底型制約として使用する参照型は0個または1個のみが理にかなっています。

```
public class A<T> where T : EventArgs
```

csharp

3. 実装インターフェース制約 (implemented interface constraints) — ジェネリック引数は指定されたインターフェース型を実装している（または直接その型である）必要があります。0個以上のインターフェースを指定できます。

```
public class A<T> where T : ICloneable, IComparable<T>
```

csharp

上記の制約は暗黙的な AND で結合されます。つまり、ジェネリックパラメータは指定された型から派生し、複数のインターフェースを実装し、デフォルトコンストラクタを持つように制約できます。宣言する型のすべてのジェネリックパラメータを制約の表現に使用でき、パラメータ間の相互依存関係を導入します。例えば：

```
public class A<S, T, U>
    where S : T
    where T : IList<U> {
```

csharp

```
    void f<V>(V v) where V : S {}
}
```

インスタンス化 (Instantiation)

ジェネリック型またはメソッドのジェネリックパラメータに置換されたジェネリック引数のリストです。ロードされた各ジェネリック型およびメソッドにはそのインスタンス化があります。

典型的インスタンス化 (Typical Instantiation)

型またはメソッド自身の型パラメータのみで構成され、パラメータが宣言された順序と同じ順序のインスタンス化です。各ジェネリック型およびメソッドには正確に1つの典型的インスタンス化が存在します。通常、オープンジェネリック型について話すとき、典型的インスタンス化を念頭に置いています。例：

```
public class A<S, T, U> {}
```

csharp

C# の `typeof(A<, ,>)` は `ldtoken A`3` にコンパイルされ、ランタイムに `A`3` を `S`、`T`、`U` でインスタンス化してロードさせます。

正規インスタンス化 (Canonical Instantiation)

すべてのジェネリック引数が `System.__Canon` であるインスタンス化です。`System.__Canon` は `corlib` で定義された内部型であり、その役割はジェネリック引数として使用される可能性のある他のどの型とも異なる、よく知られた型であることです。正規インスタンス化を持つ型/メソッドは、すべてのインスタンス化の代表として使用され、すべてのインスタンス化で共有される情報を保持します。

`System.__Canon` はそれぞれのジェネリックパラメータが持つ制約を明らかに満たせないため、制約チェックは `System.__Canon` に関する特別扱いされ、これらの違反を無視します。

💡 初心者向け補足

`System.__Canon` は CLR 内部で使われる特殊な型で、「すべての参照型の代表」として機能します。例えば `List<string>` と `List<object>` は異なる型ですが、内部的には `List<System.__Canon>` という共通のテンプレートを共有しています。これにより、参照型引数を使うジェネリック型ごとにメソッドコードを再生成する必要がなくなり、メモリ使用量が削減されます。Java のジェネリクスが型消去 (type erasure) で実装されているのに対し、.NET は具象化 (reification) されたジェネリクスをこのようないくつかの仕組みで効率的に実現しています。

2.2.2 共有

ジェネリクスの登場により、ランタイムがロードする型の数は多くなる傾向があります。異なるインスタンス化を持つジェネリック型（例：`List<string>` と `List<object>`）はそれぞれ固有の `MethodTable` を持つ異なる型ですが、共有できる情報がかなりの量あることが判明しています。この共有はメモリフットプリントにプラスの影響を与え、結果としてパフォーマンスも向上します。

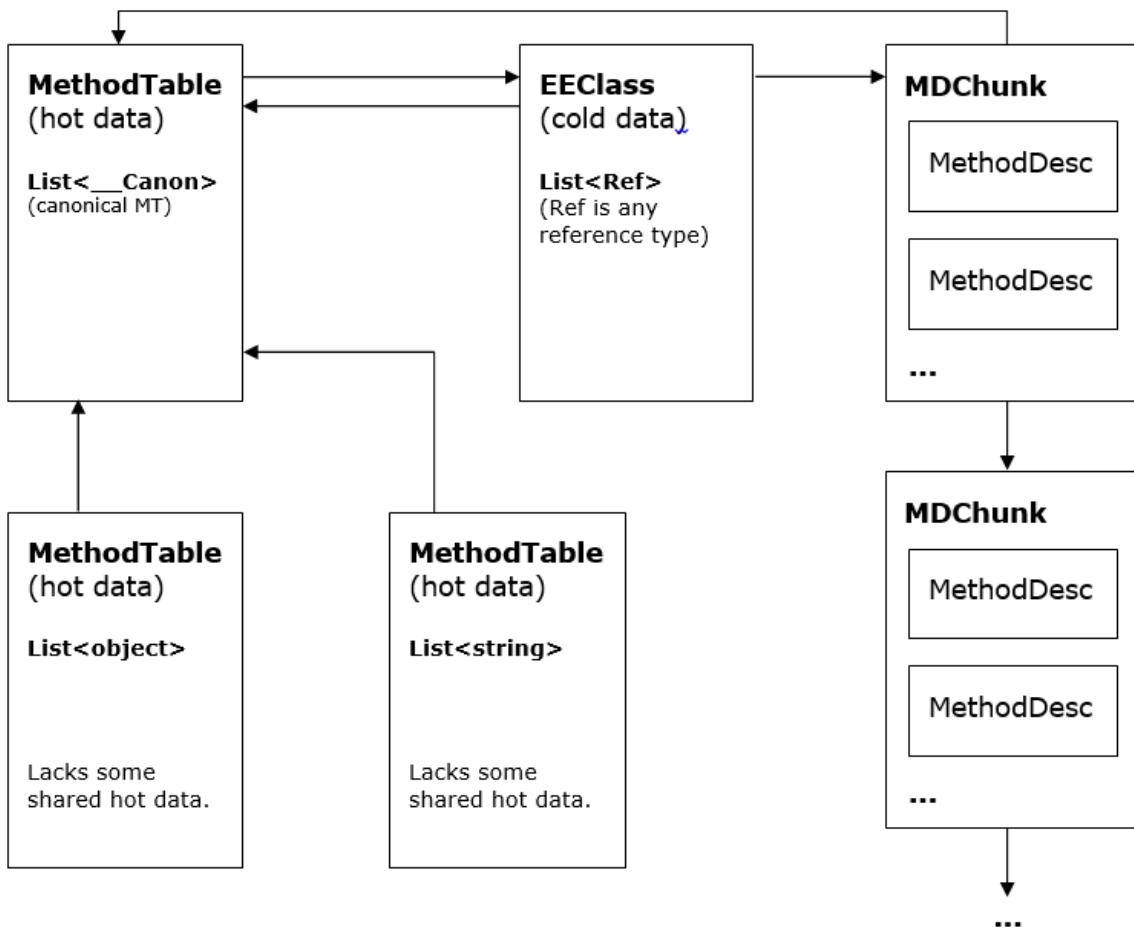


図4 非ジェネリックメソッドを持つジェネリック型 — 共有された `EEClass`

現在、参照型を含むすべてのインスタンス化は同じ `EEClass` とその `MethodDesc` を共有しています。すべての参照は同じサイズ（4 バイトまたは 8 バイト）であるため、これらすべての型のレイアウトは同じになります。これが共有を実現可能にしています。上の図は `List<object>` と `List<string>` についてこれを示しています。正規の `MethodTable` は最初の参照型インスタンス化がロードされる前に自動的に作成され、ホットだがインスタンス化固有ではないデータ（非仮想スロットなど）を含みます。値型のみを含むインスタンス化は共有されず、そのようなインスタンス化された型はそれぞれ独自の非共有 `EEClass` を持ります。

💡 初心者向け補足

`List<string>` と `List<object>` は異なる型ですが、内部的には多くの情報を共有しています。これは、参照型（クラス）のポインタサイズがすべて同じ（32ビット環境で4バイト、64ビット環境で8バイト）であるためです。一方、`List<int>` と `List<double>` は値型のサイズが異なるため共有できず、それぞれ独自のデータ構造を持ちます。この「共有」の仕組みにより、ジェネリクスを多用してもメモリの消費を抑えることができます。

これまでにロードされたジェネリック型を表す `MethodTable` は、そのローダーモジュールが所有するハッシュテーブルにキャッシュされます⁵。このハッシュテーブルは新しいインスタンス化が構築される前に参照され、同じ型を表す2つ以上の `MethodTable` インスタンスが存在しないことを保証します。

ジェネリック共有の詳細については、[Design and Implementation of Generics for the .NET Common Language Runtime](#) を参照してください。

⁵ NGEN イメージからロードされた型については、状況はもう少し複雑になります。

メソッドディスクリプタ (Method Descriptor)

原文

この章の原文は [Method Descriptor](#) です。

著者: Jan Kotas ([@jkotas](#)) - 2006

はじめに

MethodDesc (メソッドディスクリプタ) は、マネージドメソッドのランタイム内部表現です。以下のような複数の目的を果たします：

- ランタイム全体で使用可能な一意のメソッドハンドルを提供します。通常のメソッドにおいて、MethodDesc は <モジュール, メタデータトークン, インスタンス化> の三つ組に対する一意のハンドルです。
- メタデータから計算するとコストが高い、頻繁に使用される情報をキャッシュします（例：メソッドが静的かどうか）。
- メソッドのランタイム状態を保持します（例：メソッドのコードが既に生成されたかどうか）。
- メソッドのエントリポインツを所有します。

💡 初心者向け補足

MethodDesc は、C# で定義したメソッド（例えば `public void MyMethod()`）に対して、.NET ランタイムが内部的に作成するデータ構造です。Java の JVM における「メソッドエリア」に相当する概念で、メソッドに関するあらゆる情報（名前、引数の型、JIT コンパイル済みのネイティブコードへのポインタなど）を一箇所にまとめて管理します。プログラム中のすべてのメソッドに対して 1 つずつ存在します。

設計目標と非目標

目標

パフォーマンス: MethodDesc の設計は、すべてのメソッドに 1 つずつ存在するため、サイズの最適化が重点的に行われています。たとえば、通常の非ジェネリックメソッドの MethodDesc は、現在の設計では 8 バイトです。

非目標

情報の豊富さ: MethodDesc はメソッドに関するすべての情報をキャッシュするわけではありません。使用頻度の低い情報（例：メソッドシグネチャ）については、基礎となるメタデータにアクセスする必要があることが前提とされています。

MethodDesc の設計

MethodDesc の種類

MethodDesc には複数の種類があります：

IL

通常の IL メソッドに使用されます。

Instantiated (インスタンス化)

ジェネリックインスタンス化を持つ IL メソッドや、メソッドテーブルに事前割り当てされたスロットを持たない IL メソッドに使用されます。

FCall

アンマネージドコードで実装された内部メソッドです。これは [MethodImplAttribute\(MethodImplOptions.InternalCall\)](#) 属性が付与されたメソッド、デリゲートコンストラクタ、および tlbimp コンストラクタです。

PInvoke

P/Invoke メソッドです。DllImport 属性が付与されたメソッドがこれに該当します。

EImpl

ランタイムによって実装が提供されるデリゲートメソッド (Invoke、BeginInvoke、EndInvoke) です。[ECMA 335 Partition II - Delegates](#) を参照してください。

Array (配列)

ランタイムによって実装が提供される配列メソッド (Get、Set、Address) です。[ECMA Partition II – Arrays](#) を参照してください。

ComInterop

COM インターフェースメソッドです。非ジェネリックインターフェースはデフォルトで COM 相互運用に使用できるため、この種類は通常すべてのインターフェースメソッドに使用されます。

Dynamic (動的)

基礎となるメタデータを持たない、動的に作成されたメソッドです。Stub-as-IL や LKG (軽量コード生成、Light-weight Code Generation) によって生成されます。

💡 初心者向け補足

これらの種類は、メソッドがどのように定義・実装されているかによって分類されます。最も一般的なのは IL で、C# などで書いた通常のメソッドはすべてこれに該当します。PInvoke は Windows API のようなネイティブ DLL の関数を呼び出す際に使われ、

FCall は `string.Length` のようなランタイム自体がネイティブコードで実装している高速な内部メソッドに使われます。Java でいえば、**FCall** は JNI ネイティブメソッドに近い概念です。

代替実装

C++ では、仮想メソッドと継承を使って様々な種類の `MethodDesc` を実装するのが自然な方法です。しかし、仮想メソッドは各 `MethodDesc` に vtable ポインタを追加してしまい、貴重な領域を大量に浪費します。vtable ポインタは x86 では 4 バイトを占有します。代わりに、仮想化は `MethodDesc` の種類に基づくスイッチ分岐で実装されており、種類は 3 ビットに収まります。たとえば：

```
C++  
DWORD MethodDesc::GetAttrs()  
{  
    if (IsArray())  
        return ((ArrayMethodDesc*)this)->GetAttrs();  
  
    if (IsDynamic())  
        return ((DynamicMethodDesc*)this)->GetAttrs();  
  
    return GetMDImport()->GetMethodDefProps(GetMemberDef());  
}
```

💡 初心者向け補足

通常の C++ 設計では、`MethodDesc` を基底クラスとし、`ArrayMethodDesc` や `DynamicMethodDesc` などを派生クラスとして仮想関数（`virtual`）で多態性を実現します。しかし、仮想関数を使うと各オブジェクトに vtable ポインタ（x86 で 4 バイト、x64 で 8 バイト）が追加されます。`MethodDesc` はすべてのメソッドに 1 つずつ存在するため、この数バイトの追加が全体で大きなメモリ消費になります。そこで、種類を 3 ビットのフラグとして持ち、`if` 文で分岐する方式を採用してメモリを節約しています。

メソッドスロット (Method Slots)

各 `MethodDesc` はスロットを持ち、メソッドの現在のエントリポインが格納されています。スロットは、抽象メソッドのように一度も実行されないメソッドも含め、すべてのメソッドに存在しなければなりません。ランタイム内の複数の箇所が、エントリポインと `MethodDesc` の間のマッピングに依存しています。

各 `MethodDesc` は論理的にはエントリポインを持ちますが、`MethodDesc` の作成時にこれらを積極的に割り当てることはしません。不变条件として、メソッドが実行すべきメソッドとして識別されるか、仮想オーバーライドで使用される場合にのみ、エントリポインを割り当てます。

スロットは `MethodTable` 内または `MethodDesc` 自体のいずれかに格納されます。スロットの格納場所は、`MethodDesc` の `mdcHasNonVtableSlot` ビットによって決定されます。

仮想メソッドやジェネリック型のメソッドなど、スロットインデックスによる効率的な検索が必要なメソッドの場合、スロットは `MethodTable` に格納されます。この場合、`MethodDesc` にはスロットインデックスが含まれており、エントリポインの高速な検索が

可能です。

それ以外の場合、スロットは MethodDesc 自体の一部として格納されます。この方式はデータの局所性を向上させ、ワーキングセットを節約します。また、Edit & Continue で追加されたメソッド、ジェネリックメソッドのインスタンス化、[動的メソッド](#)など、動的に作成される MethodDesc に対しては、MethodTable にスロットを事前に割り当てることがそもそも不可能な場合もあります。

MethodDesc チャンク (MethodDesc Chunks)

MethodDesc は領域を節約するためにチャンク単位で割り当てられます。複数の MethodDesc は同一の MethodTable とメタデータトークンの上位ビットを共有する傾向があります。MethodDescChunk は、共通情報を先頭にまとめ、その後ろに複数の MethodDesc の配列を配置することで構成されます。各 MethodDesc は配列内での自身のインデックスのみを保持します。

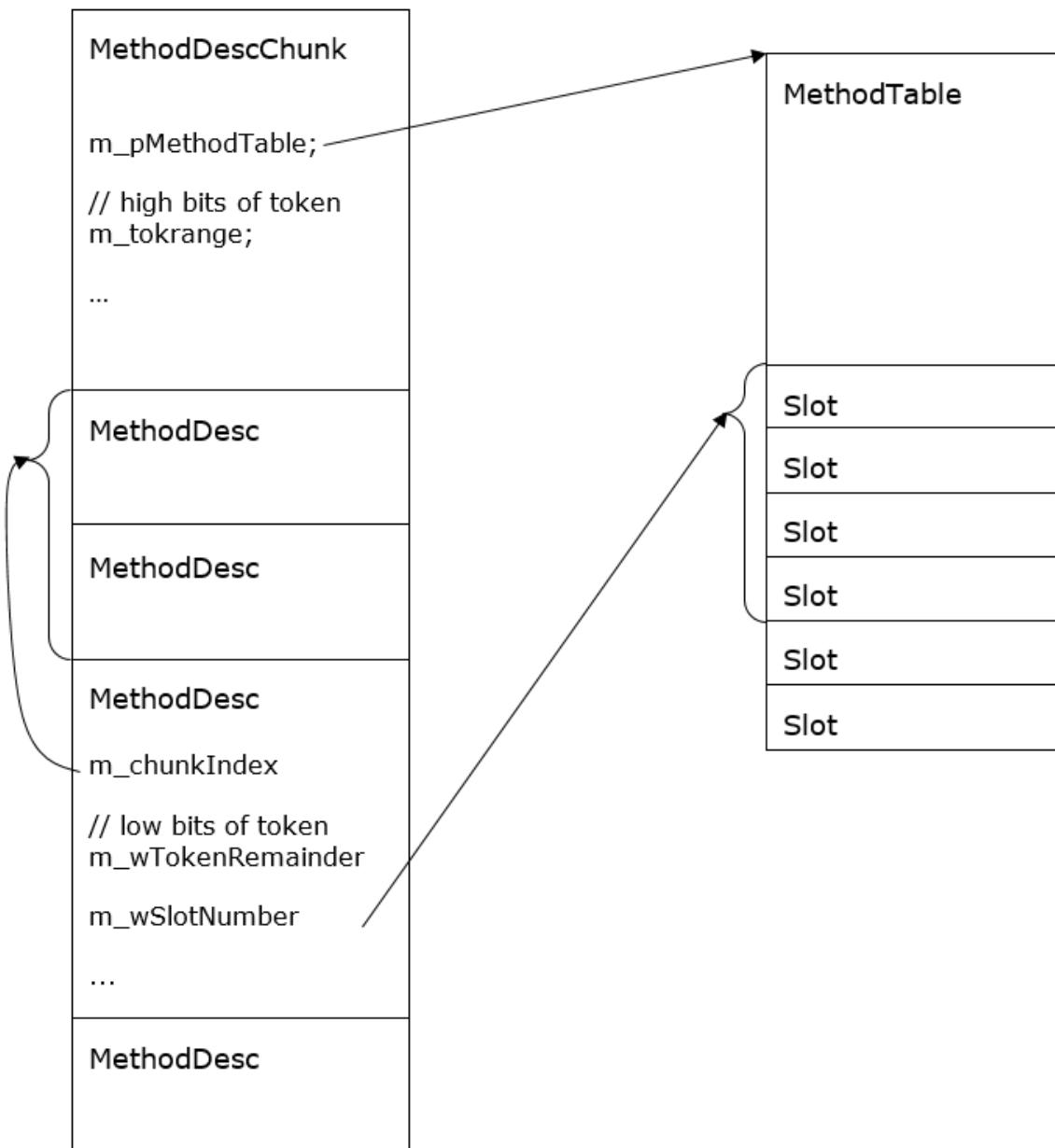


図 1 MethodDescChunk と MethodTable

初心者向け補足

チャンク（まとまり）による割り当ては、メモリ効率を高めるための最適化手法です。たとえば、あるクラスに 10 個のメソッドがある場合、10 個の MethodDesc がそれぞれ独立してクラス情報を持つのではなく、クラス情報を 1 箇所（MethodDescChunk の先頭）にまとめ、各 MethodDesc はそのチャンク内でのインデックス番号だけを持ちます。これは、同じマンション内の各部屋がそれぞれ住所を完全に保持する代わりに、部屋番号だけを持つようなものです。

デバッグ

以下の SOS コマンドが MethodDesc のデバッグに役立ちます：

- **DumpMD** – MethodDesc の内容をダンプします：

```
!DumpMD 00912fd8
Method Name: My.Main()
Class: 009111ec
MethodTable: 00912fe8md
Token: 06000001
Module: 00912c14
IsJitted: yes
CodeAddr: 00ca0070
```

- **IP2MD** – 指定されたコードアドレスから MethodDesc を検索します：

```
!ip2md 00ca007c
MethodDesc: 00912fd8
Method Name: My.Main()
Class: 009111ec
MethodTable: 00912fe8md
Token: 06000001
Module: 00912c14
IsJitted: yes
CodeAddr: 00ca0070
```

- **Name2EE** – 指定されたメソッド名から MethodDesc を検索します：

```
!name2ee hello.exe My.Main
Module: 00912c14 (hello.exe)
Token: 0x06000001
MethodDesc: 00912fd8
Name: My.Main()
JITTED Code Address: 00ca0070
```

- **Token2EE** – 指定されたトークンから MethodDesc を検索します（特殊な名前のメソッドの MethodDesc を見つけるのに便利です）：

```
!token2ee hello.exe 0x06000001
Module: 00912c14 (hello.exe)
Token: 0x06000001
```

```
MethodDesc: 00912fd8
Name: My.Main()
JITTED Code Address: 00ca0070
```

- **DumpMT -MD** – 指定された MethodTable 内のすべての MethodDesc をダンプします：

```
!DumpMT -MD 0x00912fe8
...
MethodDesc Table
Entry MethodDesc      JIT Name
79354bec  7913bd48  PreJIT System.Object.ToString()
793539c0  7913bd50  PreJIT System.Object.Equals(System.Object)
793539b0  7913bd68  PreJIT System.Object.GetHashCode()
7934a4c0  7913bd70  PreJIT System.Object.Finalize()
00ca0070  00912fd8  JIT My.Main()
0091303c  00912fe0  NONE My..ctor()
```

デバッグビルドでは、MethodDesc にメソッドの名前とシグネチャのフィールドが含まれます。これは、ランタイムの状態がひどく破損して SOS 拡張が機能しない場合のデバッグに役立ちます。

プリコード (Precode)

プリコード (Precode) は、一時的なエントリポイントの実装と、スタブの効率的なラッパーとして使用される小さなコード断片です。プリコードは、これら 2 つのケースにおいて可能な限り効率的なコードを生成するニッチなコードジェネレータです。理想的な世界では、ランタイムが動的に生成するすべてのネイティブコードは JIT によって生成されるべきです。しかし、これら 2 つのシナリオの特殊な要件を考えると、それは実現可能ではありません。x86 における基本的なプリコードは以下のようになります：

```
mov eax, pMethodDesc // MethodDesc をスクラッチレジスタにロード
jmp target           // ターゲットにジャンプ
```

効率的なスタプラッパー: 特定のメソッド（例：P/Invoke、デリゲート呼び出し、多次元配列のセッターやゲッター）の実装はランタイムによって提供され、通常は手書きのアセンブリリストアとして実装されます。プリコードは、スタブを複数の呼び出し元で多重化するための、領域効率の良いラッパーを提供します。

スタブのワーカーコードは、MethodDesc にマッピング可能で、スタブのワーカーコードにジャンプするプリコード断片によってラップされます。これにより、スタブのワーカーコードを複数のメソッド間で共有できます。これは P/Invoke マーシャリングスタブの実装に使用される重要な最適化です。また、MethodDesc とエントリポイントの間に 1 対 1 のマッピングを作成し、シンプルで効率的な低レベルシステムを確立します。

一時的なエントリポイント (**Temporary Entry Points**): メソッドは JIT コンパイルされる前にエントリポイントを提供する必要があります。これにより、JIT コンパイル済みのコードがそれらを呼び出すためのアドレスを持てます。これらの一時的なエントリポイントはプリコードによって提供されます。これはスタプラッパーの一形態です。

この手法は JIT コンパイルへの遅延的なアプローチであり、空間と時間の両方においてパフォーマンスの最適化を提供します。そうでなければ、メソッドの推移的閉包（あるメソッドが呼び出すすべてのメソッド、さらにそれらが呼び出すメソッドの全体）を、実行前に JIT コンパイルする必要があるでしょう。これは無駄です。なぜなら、JIT コンパイルが必要なのは、実際に実行されるコード分岐（例：if 文）の依存先だからです。

各一時的なエントリポイントは、典型的なメソッド本体よりもはるかに小さくなっています。数が多いため、パフォーマンスを犠牲にしてでも小さくする必要があります。一時的なエントリポイントは、メソッドの実際のコードが生成される前に一度だけ実行されます。

一時的エントリポイントのターゲットは PreStub であり、これはメソッドの JIT コンパイルをトリガーする特殊な種類のスタブです。PreStub は一時的エントリポイントを安定エントリポイント (stable entry point) にアトミックに置き換えます。安定エントリポイントはメソッドの存続期間を通じて一定でなければなりません。この不变条件は、メソッドスロットが常にロックなしでアクセスされるため、スレッドセーフティを保証するために必要です。

安定エントリポイント (stable entry point) は、ネイティブコードまたはプリコードのいずれかです。ネイティブコード (native code) は、JIT コンパイルされたコードまたは NGen イメージに保存されたコードです。実際にはネイティブコードを意味しているのに、JIT コンパイルされたコードと言及することがよくあります。

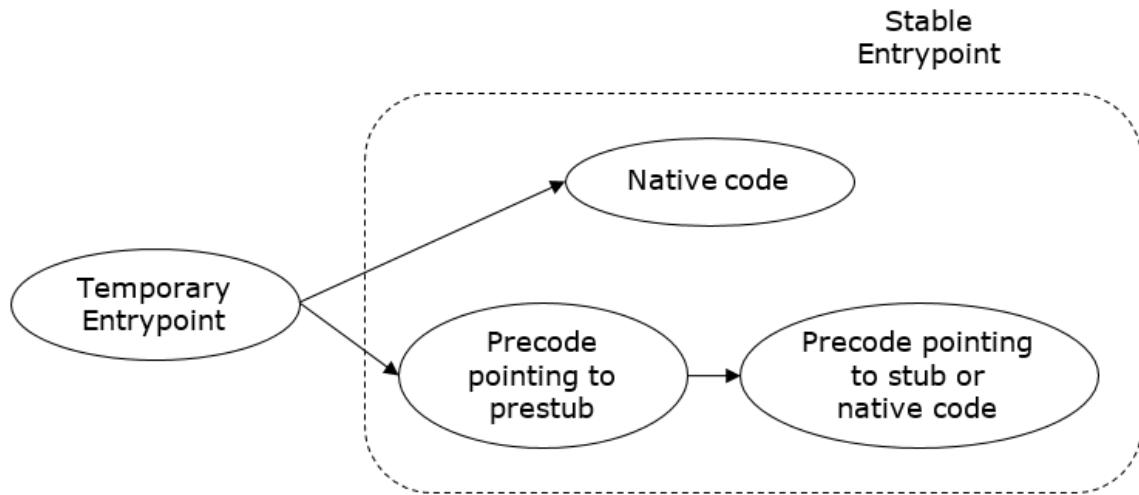


図 2 エントリポイントの状態遷移図

メソッドは、実際のメソッド本体の実行前に作業を行う必要がある場合、ネイティブコードとプリコードの両方を持つことがあります。この状況は通常、NGen イメージのフィックスアップで発生します。この場合、ネイティブコードはオプションの MethodDesc スロットになります。これは、メソッドのネイティブコードを安価で統一的な方法で検索するために必要です。

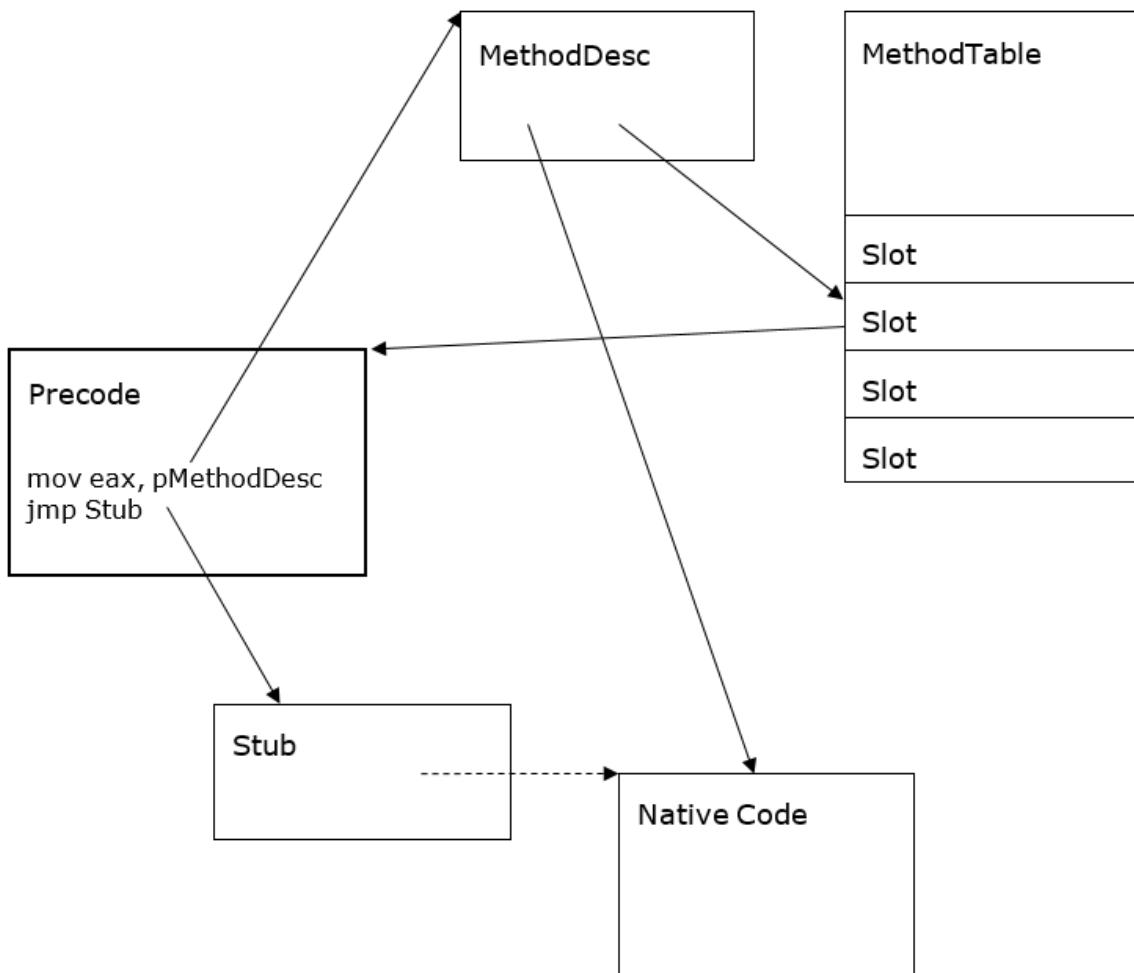


図3 プリコード、スタブ、ネイティブコードの最も複雑なケース

💡 初心者向け補足

プリコードの仕組みを日常的な例で説明すると、「電話の転送」に似ています。最初、メソッドを呼び出すと一時的エントリポインスト（転送先）に接続され、そこから PreStub（受付係）に繋がります。PreStub は JIT コンパイラにメソッドのネイティブコードを生成させ、以降の呼び出しはそのネイティブコードに直接繋がるようになります。これにより、実際に呼び出されるメソッドだけが JIT コンパイルされる「遅延コンパイル」が実現され、起動時間とメモリ使用量の両方が最適化されます。

シングルコーラブル vs マルチコーラブルエントリポインスト

メソッドを呼び出すためにはエントリポインストが必要です。MethodDesc は、与えられた状況に応じて最も効率的なエントリポインストを取得するロジックをカプセル化したメソッドを公開しています。重要な違いは、エントリポインストがメソッドの呼び出しに1回だけ使用されるか、複数回使用されるかです。

たとえば、一時的エントリポインストを使ってメソッドを複数回呼び出すのは良くない考えです。毎回 PreStub を経由してしまうからです。一方、一時的エントリポインストを使ってメソッドを1回だけ呼び出す場合は問題ありません。

MethodDesc から呼び出し可能なエントリポインストを取得するメソッドは以下の通りです：

- [MethodDesc::GetSingleCallableAddrOfCode](#)
 - [MethodDesc::GetMultiCallableAddrOfCode](#)
 - [MethodDesc::TryGetMultiCallableAddrOfCode](#)
 - [MethodDesc::GetSingleCallableAddrOfVirtualizedCode](#)
 - [MethodDesc::GetMultiCallableAddrOfVirtualizedCode](#)
-

プリコードの種類

プリコードには複数の特殊な種類があります。

プリコードの種類は命令シーケンスから安価に計算できる必要があります。x86 および x64 では、プリコードの種類は一定のオフセット位置にあるバイトを読み取ることで判別されます。当然ながら、これは様々なプリコードの種類を実装するために使用される命令シーケンスに制約を課します。

StubPrecode

StubPrecode は基本的なプリコードの種類です。MethodDesc をスクラッチレジスタ²にロードし、ジャンプします。プリコードが機能するためには、これを実装する必要があります。他の特殊なプリコードの種類が利用できない場合のフォールバックとして使用されます。

他のすべてのプリコードの種類は、プラットフォーム固有のファイルが HAS_XXX_PRECODE 定義によって有効にするオプションの最適化です。

StubPrecode は x86 では以下のようになります：

```
mov eax, pMethodDesc  
mov ebp, ebp // プリコードの種類を示すダミー命令  
jmp target
```

"target" は最初は PreStub を指しています。最終ターゲットを指すようにパッチされます。最終ターゲット（スタブまたはネイティブコード）は、eax 内の MethodDesc を使用する場合としない場合があります。スタブはよくそれを使用しますが、ネイティブコードは使用しません。

FixupPrecode

FixupPrecode は、最終ターゲットがスクラッチレジスタ²内の MethodDesc を必要としない場合に使用されます。FixupPrecode は MethodDesc のスクラッチレジスタへのロードを省略することで、数サイクルを節約します。

使用されるスタブのほとんどはより効率的な形式であり、現在、特殊な形式の Precode が不要な場合は相互運用メソッド以外のすべてにこの形式を使用できます。

FixupPrecode の初期状態 (x86) :

```
call PrecodeFixupThunk // この呼び出しは戻りません。リターンアドレスをポップし、  
// それを使って下の pMethodDesc を取得し、  
// JIT コンパイルが必要なメソッドを見つけます
```

```
pop esi // プリコードの種類を示すダミー命令
```

```
dword pMethodDesc
```

最終ターゲットにパッチされた後：

```
jmp target
```

```
pop edi
```

```
dword pMethodDesc
```

² MethodDesc をスクラッチレジスタに渡すことは、**MethodDesc 呼び出し規約 (MethodDesc Calling Convention)** と呼ばれます。

ThisPtrRetBufPrecode

ThisPtrRetBufPrecode は、値型を返すオープンインスタンスデリゲートにおいて、リターンバッファと this ポインタを入れ替えるため使用されます。MyValueType Bar(Foo x) の呼び出し規約を MyValueType Foo::Bar() の呼び出し規約に変換するために使用されます。

このプリコードは常に実際のメソッドエントリポイントのラッパーとしてオンデマンドで割り当てられ、テーブル (FuncPtrStubs) に格納されます。

ThisPtrRetBufPrecode は以下のようになります：

```
mov eax,ecx
```

```
mov ecx,edx
```

```
mov edx,eax
```

```
nop
```

```
jmp entrypoint
```

```
dw pMethodDesc
```

PInvokeImportPrecode

PInvokeImportPrecode は、アンマネージド P/Invoke ターゲットの遅延バインディング (lazy binding) に使用されます。このプリコードは利便性のため、およびプラットフォーム固有の配管コードを削減するために使用されます。

各 PInvokeMethodDesc は通常のプリコードに加えて PInvokeImportPrecode を持ります。

PInvokeImportPrecode は x86 では以下のようになります：

```
mov eax,pMethodDesc
```

```
mov eax,eax // プリコードの種類を示すダミー命令
```

```
jmp PInvokeImportThunk // pMethodDesc の P/Invoke ターゲットを遅延ロード
```

仮想スタブディスパッチ (Virtual Stub Dispatch)

原文

この章の原文は [Virtual Stub Dispatch](#) です。

著者: Simon Hall ([@snwbrdwndsrf](#)) - 2006

はじめに

仮想スタブディスパッチ (VSD: Virtual Stub Dispatch) は、従来の仮想メソッドテーブル (vtable) の代わりにスタブ (stub) を使用して仮想メソッド呼び出しを行う技術です。以前のインターフェースディスパッチでは、インターフェースにプロセス全体で一意な識別子が必要であり、読み込まれたすべてのインターフェースをグローバルなインターフェース仮想テーブルマップに追加する必要がありました。この要件により、すべてのインターフェースおよびインターフェースを実装するすべてのクラスは、NGEN シナリオにおいてランタイム時にリストアされなければならず、起動時のワーキングセットの大幅な増加を引き起こしていました。スタブディスパッチの動機は、関連するワーキングセットの多くを排除し、残りの処理をプロセスのライフトайム全体に分散させることでした。

VSD は仮想インスタンスマソッド呼び出しとインターフェースマソッド呼び出しの両方をディスパッチすることが可能ですが、現在はインターフェースディスパッチにのみ使用されています。

💡 初心者向け補足

仮想メソッドテーブル (vtable) とは、オブジェクト指向言語において仮想メソッド（オーバーライド可能なメソッド）の呼び出し先を解決するための従来の仕組みです。Java の vtable に相当します。各クラスがメソッドへのポインタの配列を持ち、メソッド呼び出し時にはこのテーブルを参照して実際の呼び出し先を決定します。

VSD はこの vtable を「スタブ」と呼ばれる小さなコード片に置き換えることで、メモリ使用量（ワーキングセット）を削減しようとする最適化技術です。

依存関係

コンポーネントの依存関係

スタブディスパッチのコードは、ランタイムの他の部分からは比較的独立して存在しています。依存コンポーネントが使用できる API を提供しており、以下に挙げる依存関係は比較的小さな接触面を構成しています。

コードマネージャー (Code Manager)

VSD は事実上、コードマネージャーに依存してメソッドの状態に関する情報を取得します。特に、特定のメソッドが最終状態に遷移したかどうかの情報であり、VSD がスタブの生成やターゲットのキャッシングなどの詳細を決定するために必要です。

型とメソッド (Types and Methods)

MethodTable は、任意の VSD コールサイトのターゲットコードアドレスを決定するために使用されるディスパッチマップ (dispatch map) へのポインタを保持しています。

特殊な型 (Special Types)

COM インターオプ型への呼び出しは、特殊なターゲット解決を持つため、カスタムディスパッチされなければなりません。

このコンポーネントに依存するコンポーネント

コードマネージャー (Code Manager)

コードマネージャーは、JIT コンパイラにインターフェース呼び出しのコールサイトターゲットを提供するために VSD に依存しています。

クラスビルダー (Class Builder)

クラスビルダーは、ディスパッチマッピングコードが公開する API を使用して、型の構築時にディスパッチマップを作成します。このマップはディスパッチ時に VSD コードによって使用されます。

設計の目標と非目標

目標

ワーキングセットの削減

インターフェースディスパッチは以前、プロセス全体のインターフェース識別子を扱う大きくやや疎な vtable ルックアップマップを使用して実装されていました。目標は、ディスパッチスタブを必要に応じて生成することでコールドワーキングセットの量を削減し、理論的には関連するコールサイトとそのディスパッチスタブを互いに近くに配置してワーキングセット密度を高めることでした。

💡 初心者向け補足

ワーキングセットとは、プロセスが実際にメモリ上に保持しているページの集合をいいます。コールドワーキングセットはめったにアクセスされないが確保されたままのメモリ領域を指します。ワーキングセットが大きいと、アプリケーションの起動が遅くなったり、メモリ使用量が増えたりします。VSD は「必要になったときだけスタブを生成する」ことで、このメモリ使用量を削減しています。

VSD のコールサイトあたりの初期ワーキングセットは、システムの実行中に作成・収集されるさまざまなスタブを追跡するために必要なデータ構造のせいで高くなることに注意が必要です。しかし、アプリケーションが定常状態に達すると、これらのデータ構造は単純なディスパッチには必要なくなるため、ページアウトされます。残念ながら、クライアントアプリケーションではこれは起動時間の遅延に等しく、仮想メソッドに対する VSD を無効にする要因の一つとなりました。

スループットの同等性

インターフェースおよび仮想メソッドのディスパッチを、以前の vtable ディスパッチメカニズムと償却的に同等のスループットに保つことが重要でした。

インターフェースディスパッチではこれが達成可能であることはすぐに明らかでしたが、仮想メソッドディスパッチではやや遅いことが判明し、これが仮想メソッドに対する VSD を無効にする要因の一つとなりました。

トークン表現とディスパッチマップの設計

ディスパッチトークン (dispatch token) はランタイムで割り当てられるネイティブワードサイズの値であり、内部的にはインターフェースとスロットを表すタプルで構成されています。

この設計では、割り当てられた型識別子の値とスロット番号の組み合わせを使用します。ディスパッチトークンはこれら2つの値の組み合いで構成されます。ランタイムとの統合を容易にするため、この実装はクラシックな vtable レイアウトと同じ方法でスロット番号を割り当てます。これは、ランタイムが MethodTable、MethodDesc、スロット番号をまったく同じ方法で扱えることを意味しますが、この抽象化を処理するために vtable に直接アクセスする代わりにヘルパーメソッドを介してアクセスする必要があります。

スロット (slot) という用語は、常にクラシックな vtable レイアウトにおけるスロットインデックス値のコンテキストで使用され、マッピングメカニズムによって作成・解釈されるものです。これは、ランタイムで以前実装されていた、仮想メソッドスロットの後に非仮想メソッドスロットが続くクラシックなメソッドテーブルレイアウトを想像した場合のスロット番号です。ランタイムコード内ではスロットがクラシックな vtable 構造へのインデックスとしての意味と、vtable 内のポインタのアドレスとしての意味の両方で使われるため、この区別を理解することが重要です。変更点は、スロットがインデックス値のみとなり、コードポインタアドレスは実装テーブル (implementation table) (後述) に格納されるようになったことです。

動的に割り当てられる型識別子の値については後述します。

メソッドテーブル (Method Table)

実装テーブル (Implementation Table)

これは、型によって導入された各メソッドボディに対して、そのメソッドのエントリポイントへのポインタを持つ配列です。メンバーは以下の順序で配置されます。

- 導入された (newslot) 仮想メソッド
- 導入された非仮想（インスタンスおよびスタティック）メソッド
- オーバーライドする仮想メソッド

この形式の理由は、クラシックな vtable レイアウトの自然な拡張を提供するためです。その結果、スロットマップ (slot map) (後述) の多くのエントリは、この順序やクラスの仮想・非仮想の総数などの詳細から推論できます。

仮想インスタンスマソッドのスタブディスパッチが無効になっている場合（現在無効になっています）、実装テーブルは存在せず、真の vtable に置き換えられます。すべてのマッピング結果は、実装テーブルではなく vtable のスロットとして表現されます。このドキュメント全体で実装テーブルに言及する場合は、この点を念頭に置いてください。

スロットマップ (Slot Map)

スロットマップは、0個以上の `<type, [<slot, scope, (index | slot)>]>` エントリのテーブルです。`type` は上述の動的に割り当てられた識別番号であり、現在のクラスを示すセンチネル値（仮想インスタンスマソッドの呼び出し）か、現在のクラスが実装するインターフェース（またはその親の1つによって暗黙的に実装されるもの）の識別子のいずれかです。サブマップ（角括弧内）には1つ以上のエントリがあります。各エントリ内で、最初の要素は常に `type` 内のスロットを示します。2番目の要素 `scope` は、3番目の要素が実装の `index` であるかスロット番号であるかを指定します。`scope` は、次の番号が仮想スロット番号として解釈されるべきことを示す既知のセンチネル値であり、`this.slot` として仮想的に解決されるべきものです。`scope` は、現在のクラスの継承階層における特定のクラスを識別することもでき、その場合、3番目の引数は `scope` で示されるクラスの実装テーブルへの `index` であり、`type.slot` の最終的なメソッド実装です。

初心者向け補足

スロットマップは、「どのインターフェースのどのメソッドが、どのクラスのどの実装に対応するか」を記録するテーブルです。Java における「インターフェースメソッドから具体的なクラスの実装メソッドへの対応表」に相当します。

例えば、インターフェース `IFoo` のメソッド `Bar()` がクラス `MyClass` のどのメソッドに対応するかを、このスロットマップが管理しています。

例

以下は、小さなクラス構造（C# でモデル化）と、各クラスの実装テーブルおよびスロットマップがどうなるかを示しています。

<pre>interface I { void Foo(); }</pre> <p>Classic Virtual Table I.Foo prestub</p>	<p>Implementation Table I.Foo prestub</p> <p>Slot Map</p> <table border="1" data-bbox="669 1097 1254 1186"> <thead> <tr> <th>ID for I</th><th>0</th><th>scope I</th><th>0 *</th></tr> </thead> </table>	ID for I	0	scope I	0 *							
ID for I	0	scope I	0 *									
<pre>class A : I { virtual void Foo() {} virtual void Bar() {} }</pre> <p>Classic Virtual Table A.Foo code ptr A.Bar code ptr</p>	<p>Implementation Table A.Foo code ptr A.Bar code ptr</p> <p>Slot Map</p> <table border="1" data-bbox="669 1389 1254 1614"> <tbody> <tr> <td rowspan="2">ID for A</td><td>4</td><td>scope A</td><td>0 *</td></tr> <tr><td>5</td><td>scope A</td><td>1 *</td></tr> <tr> <td>ID for I</td><td>0</td><td>virtual</td><td>4</td></tr> </tbody> </table>	ID for A	4	scope A	0 *	5	scope A	1 *	ID for I	0	virtual	4
ID for A	4		scope A	0 *								
	5	scope A	1 *									
ID for I	0	virtual	4									
<pre>class B : A { virtual override void Foo() {} virtual void Baz() {} }</pre> <p>Classic Virtual Table B.Foo code ptr A.Bar code ptr B.Baz code ptr</p>	<p>Implementation Table B.Baz code ptr B.Foo code ptr</p> <p>Slot Map</p> <table border="1" data-bbox="669 1816 1254 1926"> <tbody> <tr> <td rowspan="2">ID for B</td><td>6</td><td>scope B</td><td>0 *</td></tr> <tr><td>4</td><td>scope B</td><td>1</td></tr> </tbody> </table>	ID for B	6	scope B	0 *	4	scope B	1				
ID for B	6		scope B	0 *								
	4	scope B	1									

このマップを見ると、スロットマップのサブマップの最初の列が、クラシックな仮想テーブルビューのスロット番号に対応していることがわかります（`System.Object` は独自に4つの仮想メソッドを提供しますが、明確さのために省略されています）。メソッド実装の検索は常にボトムアップで行われます。したがって、型 `B` のオブジェクトがあり `I.Foo` を呼び出したい場合、`B` のスロットマップから `I.Foo` のマッ

ピングを探し始めます。そこで見つからなければ、*A* のスロットマップを探し、そこで見つかります。それは *I* の仮想スロット 0 (*I.Foo* に対応) が仮想スロット 4 によって実装されていることを示しています。次に *B* のスロットマップに戻って仮想スロット 4 の実装を検索し、それが *B* 自身の実装テーブルのスロット 1 によって実装されていることを見つけます。

その他の用途

このマッピング技術は、仮想スロットの `methodimpl` 再マッピング（つまり、インターフェーススロットが仮想スロットにマッピングされるのと同様に、現在のクラスのマップにおける仮想スロットマッピング）を実装するために使用できることに注意が必要です。マップのスコーリング機能により、非仮想メソッドも参照できます。これは、ランタイムが非仮想メソッドによるインターフェースの実装をサポートしたい場合に有用です。

最適化

スロットマップはビットエンコードされており、デルタ値を使用して典型的なインターフェース実装パターンを活用することで、マップサイズを大幅に削減しています。さらに、新しいスロット（仮想・非仮想の両方）は、実装テーブル内の順序から暗示できます。テーブルに新しい仮想スロット、次に新しいインスタンススロット、その後にオーバーライドが含まれている場合、適切なスロットマップエントリは、実装テーブル内のインデックスと親クラスから継承された仮想の数を組み合わせることで暗示できます。このような暗示されたマップエントリはすべて (*) で示されています。現在のデータ構造のレイアウトは以下のパターンを使用しており、`DispatchMap` は実装テーブルの順序から完全に暗示できない場合にのみ存在します。

```
MethodTable -> [DispatchMap ->] ImplementationTable
```

型 ID マップ (Type ID Map)

型を ID にマッピングします。ID は、以前にマッピングされていない型が検出されるたびに単調増加する値として割り当てられます。現在、そのような型はすべてインターフェースです。

現在、これは `HashMap` を使用して実装されており、両方向のルックアップのエントリを含んでいます。

ディスパッチトークン (Dispatch Tokens)

ディスパッチトークンは `<TypeID, slot>` タプルです。インターフェースの場合、型はそのインターフェースに割り当てられたインターフェース ID になります。仮想メソッドの場合、スロットがディスパッチ対象の型内で仮想的に解決されるべきことを示す定数値 (`this` に対する仮想メソッド呼び出し) になります。この値のペアは、ほとんどの場合、プラットフォームのネイティブワードサイズに収まります。x86 では、各値の下位 16 ビットを連結したものになる可能性が高いです。これは、ランタイムの `TypeHandle` が `MethodTable` ポインタまたは `<TypeHandle, TypeHandle>` ペアのいずれかになり得ると同様に、センチネルビットで 2つのケースを区別することでオーバーフロー問題を処理するように一般化できます。これが必要かどうかはまだ決定されていません。

仮想スタブディスパッチの設計

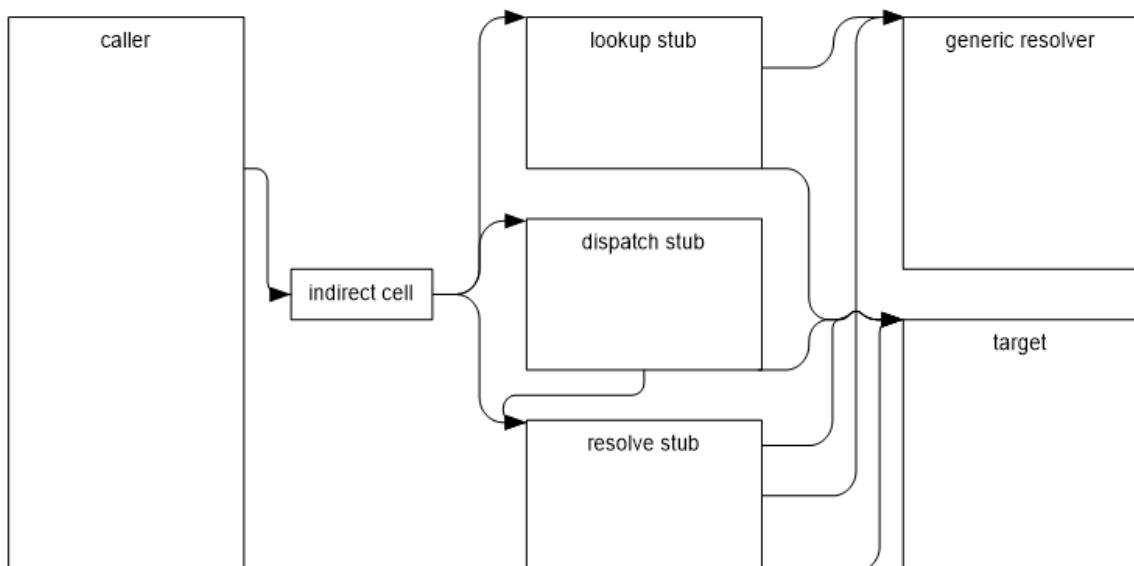
ディスパッチトークンから実装への解決

トークンと型が与えられると、トークンを型の実装テーブルインデックスにマッピングすることで実装が見つかります。実装テーブルは型の MethodTable から到達可能です。このマップは BuildMethodTable で作成されます。MethodTable を構築している型が実装するすべてのインターフェースを列举し、その型が実装またはオーバーライドするすべてのインターフェースメソッドを決定します。この情報を追跡することで、インターフェースディスパッチ時にトークンとターゲットオブジェクト（MethodTable とトークンマッピングを取得できる）からターゲットコードを決定することが可能になります。

スタブ (Stubs)

インターフェースディスパッチの呼び出しはスタブを経由します。これらのスタブはすべてオンデマンドで生成され、すべてトークンとオブジェクトを実装にマッチさせ、その実装に呼び出しを転送することを最終的な目的としています。

現在、3種類のスタブがあります。以下の図は、これらのスタブ間の一般的な制御フローを示しており、以下で説明します。



ジェネリックリゾルバ (Generic Resolver)

これは実際にはすべてのスタブの最終的なフェイルパスとして機能する C 関数です。 $\langle token, type \rangle$ タプルを受け取り、ターゲットを返します。ジェネリックリゾルバは、必要に応じてディスパッチスタブやリゾルバスタブの作成、より良いスタブが利用可能になったときの間接セル (indirection cell) のパッチ、結果のキャッシング、およびすべてのブックキーピングも担当します。

ルックアップスタブ (Lookup Stubs)

これらのスタブは、インターフェースディスパッチのコールサイトに最初に割り当たるものであり、JIT がインターフェースコールサイトをコンパイルするときに作成されます。JIT は最初の呼び出しが行われるまでトークンを満たすために使用される型の知識を持っていないため、このスタブはトークンと型を引数としてジェネリックリゾルバに渡します。必要であれば、ジェネリックリゾルバはディスパッチスタブとリゾルバスタブも作成し、コールサイトをディスパッチスタブにバックパッチして、ルックアップスタブがもう使用されないようにします。

ルックアップスタブは、一意のトークンごとに1つ作成されます（つまり、同じインターフェーススロットへのコールサイトは同じルックアップスタブを使用します）。

💡 初心者向け補足

バックパッチ (back patch) とは、コールサイト（メソッド呼び出し箇所）の呼び出し先を、より効率的なスタブに書き換えることです。最初はルックアップスタブが使われますが、一度呼び出しが行われて型が判明すると、より高速なディスパッチスタブに切り替わります。これは JIT コンパイラの自己最適化の一環です。

ディスパッチスタブ (Dispatch Stubs)

これらのスタブは、コールサイトが単態的 (monomorphic) な振る舞いをすると考えられる場合に使用されます。これは、特定のコールサイトで使用されるオブジェクトが通常同じ型である（つまり、呼び出されるオブジェクトがほとんどの場合、同じサイトで前回呼び出されたオブジェクトと同じ型である）ことを意味します。ディスパッチスタブは呼び出されるオブジェクトの型 (Method Table) を取得し、キャッシュされた型と比較して、成功すればキャッシュされたターゲットにジャンプします。x86 では、これは通常「比較、条件付き失敗ジャンプ、ターゲットへのジャンプ」というシーケンスになり、すべてのスタブの中で最高のパフォーマンスを提供します。スタブの型比較が失敗した場合、対応するリゾルブスタブ（後述）にジャンプします。

ディスパッチスタブは、一意の `<token, type>` タプルごとに1つ作成されますが、コールサイトのルックアップスタブが呼び出されたときに遅延的にのみ作成されます。

リゾルブスタブ (Resolve Stubs)

多態的 (polymorphic) なコールサイトはリゾルブスタブによって処理されます。これらのスタブは、キーペア `<token, type>` を使用してグローバルキャッシュ内のターゲットを解決します。ここで `token` は JIT 時に既知であり、`type` は呼び出し時に決定されます。グローバルキャッシュに一致するエントリがない場合、リゾルブスタブの最終ステップはジェネリックリゾルバを呼び出し、返されたターゲットにジャンプすることです。ジェネリックリゾルバが `<token, type, target>` タプルをキャッシュに挿入するため、同じ `<token, type>` タブルでの次の呼び出しはキャッシュ内でターゲットを正常に見つけます。

ディスパッチスタブが十分な頻度で失敗すると、コールサイトは多態的とみなされ、リゾルブスタブはコールサイトをリゾルブスタブに直接ポイントするようにバックパッチして、一貫して失敗するディスパッチスタブのオーバーヘッドを回避します。同期ポイント（現在は GC の終了時）では、多態的なサイトが単態的なコールサイトにランダムに昇格されます。これは、コールサイトの多態的な属性は通常一時的であるという仮定に基づいています。この仮定が特定のコールサイトに対して正しくない場合、すぐにバックパッチがトリガーされ再び多態的に降格されます。

リゾルブスタブはトークンごとに1つ作成されますが、すべてグローバルキャッシュを使用します。トークンごとに1つのスタブとすることで、`<token, type>` タプルの変化しないコンポーネントから導出された事前計算ハッシュを使用した高速で効果的なハッシュアルゴリズムが可能になります。

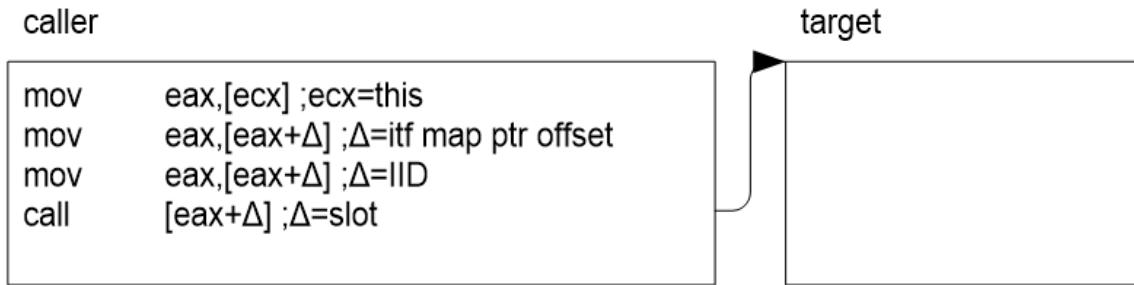
💡 初心者向け補足

単態的 (monomorphic) とは、ある呼び出し箇所で常に同じ型のオブジェクトが使われるることを意味します。多態的 (polymorphic) とは、複数の異なる型のオブジェクトが使われるることを意味します。

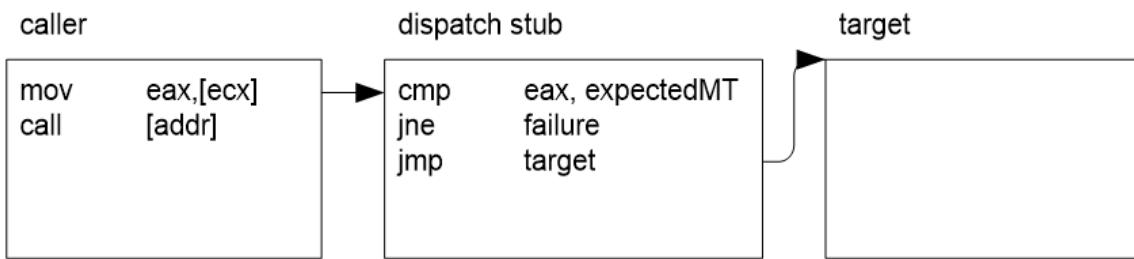
例えば、`IAnimal animal = new Dog();` と常に `Dog` 型だけが来るなら単態的、`Dog` や `Cat` が混在するなら多態的です。VSD は単態的なケースを高速化するためにディスパッチスタブを使い、多態的なケースにはリゾルブスタブ+グローバルキャッシュで対応します。

コードシーケンス

以前のインターフェース仮想テーブルディスパッチメカニズムは、以下のようなコードシーケンスを生成していました。



典型的なスタブディスパッチシーケンスは以下の通りです。



ここで `expectedMT`、`failure`、`target` はスタブにエンコードされた定数です。

典型的なスタブシーケンスは、以前のインターフェースディスパッチメカニズムと同じ数の命令を持ち、メモリの間接参照が少ないため、より小さなワーキングセット寄与で高速に実行できる可能性があります。また、作業の大部分がコールサイトではなくスタブ内にあるため、JITされるコードも小さくなります。これはコールサイトがめったに呼び出されない場合にのみ有利です。失敗分岐は、x86 の分岐予測が成功ケースを追うように配置されていることに注意してください。

現在の状態

現在、VSD はインターフェースメソッド呼び出しに対してのみ有効であり、仮想インスタンスマソッド呼び出しには有効ではありません。これにはいくつかの理由があります。

- 起動: 大量の初期スタブを生成する必要があるため、起動時のワーキングセットと速度が阻害されました。
- スループット: インターフェースディスパッチは VSD で一般的に高速になりますが、仮想インスタンスマソッド呼び出しは許容できない速度低下を受けていました。

仮想インスタンスマソッド呼び出しに対する VSD を無効にした結果、すべての型は仮想インスタンスマソッドのための vtable を持ち、上述の実装テーブルは無効化されています。インターフェースメソッドのディスパッチを可能にするため、ディスパッチマップは引き続き存在しています。

物理アーキテクチャ

ディスパッチトークンとマップの実装の詳細については、[clr/src/vm/contractImpl.h](#) および [clr/src/vm/contractImpl.cpp](#) を参照してください。

仮想スタブディスパッチの実装の詳細については、[clr/src/vm/virtualcallStub.h](#) および [clr/src/vm/virtualcallStub.cpp](#) を参照してください。

CLRにおけるスタックウォーキング

原文

この章の原文は [Stackwalking in the CLR](#) です。

著者: Rudi Martin ([@Rudi-Martin](#)) - 2008

CLR はスタックウォーキング (stack walking)、またはスタッククローリング (stack crawling) と呼ばれる技術を多用しています。これは、特定のスレッドのコールフレーム (call frame) のシーケンスを、最新のもの（そのスレッドが現在実行中の関数）からスタックのベース（底）まで順にたどる処理です。

💡 初心者向け補足

スタックウォーキングとは、プログラムの「呼び出し履歴」を順番にたどっていく処理のことです。たとえば Java の `Thread.getStackTrace()` でスタックトレースを取得するとき、内部的にはこれと同様の処理が行われています。CLR ではガベージコレクション (GC)、例外処理、デバッグなど、さまざまな場面でこの技術が使われています。

ランタイムはスタックウォークをさまざまな目的で使用しています：

- ガベージコレクション (GC) の実行中に、すべてのスレッドのスタックをウォークして、マネージドルート (managed root)（マネージドメソッドのフレーム内にあるオブジェクト参照を保持するローカル変数で、オブジェクトを生存状態に保つため、また GC がヒープをコンパクションする場合にオブジェクトの移動を追跡するために、GC に報告する必要があるもの）を探します。
- 一部のプラットフォームでは、例外処理の際にスタックウォーカーが使用されます（第1パスではハンドラーの検索、第2パスではスタックのアンワインド (unwind) を行います）。
- デバッガーがマネージドスタックトレース (managed stack trace) を生成する際に、この機能を使用します。
- さまざまな雑多なメソッド、通常はパブリックなマネージド API に近いメソッドが、呼び出し元に関する情報（呼び出し元のメソッド、クラス、アセンブリなど）を取得するためにスタックウォークを実行します。

スタックモデル

ここでは、いくつかの一般的な用語を定義し、スレッドのスタックの典型的なレイアウトについて説明します。

論理的には、スタックはいくつかの フレーム (frame) に分割されます。各フレームは、現在実行中であるか、別の関数を呼び出してその戻りを待っている、何らかの関数（マネージドまたはアンマネージド）を表します。フレームには、関連付けられた関数の特定の呼び出しに必要な状態が含まれます。通常、これにはローカル変数のためのスペース、別の関数への呼び出しのために PUSHED された引数、保存された呼び出し元のレジスターなどが含まれます。

💡 初心者向け補足

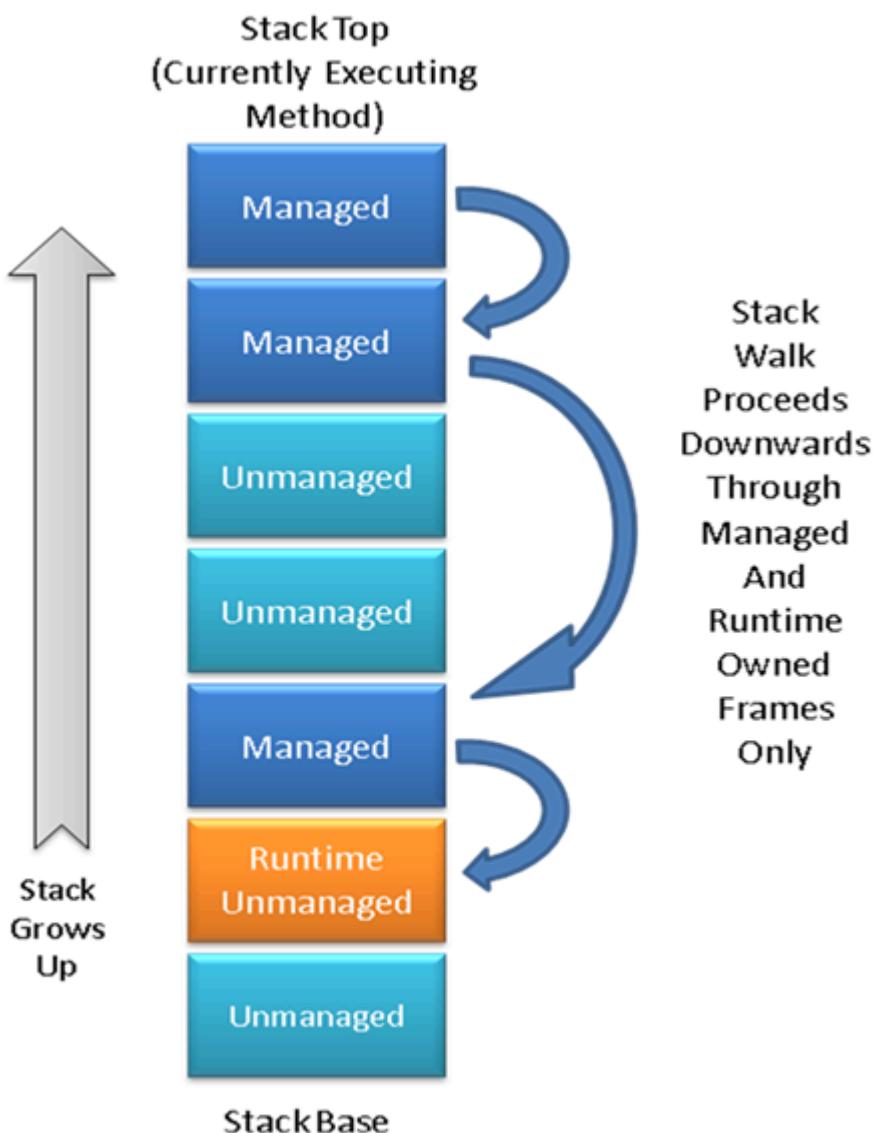
「フレーム」は、関数が1回呼び出されるたびにスタック上に作成される領域です。Java でも同様に、メソッド呼び出しごとに「スタックフレーム」が作られ、ローカル変数や引数が格納されます。たとえば `methodA()` が `methodB()` を呼ぶと、スタック上には `methodA` のフレームの上に `methodB` のフレームが積まれます。`methodB` が戻ると、そのフレームは破棄されます。

フレームの正確な定義はプラットフォームによって異なり、多くのプラットフォームでは、すべての関数が従う厳格なフレーム形式の定義がありません (x86 はその一例です)。代わりに、コンパイラはフレームの正確な形式を自由に最適化できます。このようなシステムでは、スタックウォークが100%正確で完全な結果を返すことを保証することはできません (デバッグ目的では、pdbなどのデバッグシンボルを使用してギャップを埋め、デバッガーがより正確なスタックトレースを生成できるようにしています)。

しかし、CLR にとってはこれは問題になりません。完全に汎用的なスタックウォークは必要ないからです。代わりに、CLR が関心を持つのは、マネージドフレーム (つまりマネージドメソッドを表すもの)、または、ある程度はランタイム自体の一部を実装するために使用されるアンマネージドコードからのフレームだけです。特に、サードパーティのアンマネージドフレームの忠実性について保証はなく、そのようなフレームがランタイム自体への遷移またはランタイムからの遷移を行う箇所を記録することのみが保証されます (つまり、私たちが関心を持つフレームの種類の1つです)。

私たちが関心を持つフレームの形式を制御しているため (詳細は後述します)、それらのフレームが100%の忠実性でクロール可能であることを保証できます。追加で必要な仕組みは、ランタイムフレームの分断されたグループを互いにリンクし、間にあるアンマネージド (でなければクロール不可能な) フレームをスキップできるようにするメカニズムだけです。

以下の図は、すべてのフレームタイプを含むスタックを示しています (このドキュメントでは、スタックがページの上方向に成長する慣例を使用しています) :



フレームをクロール可能にする

マネージドフレーム

ランタイムが JIT (ジャストインタイムコンパイラ) を所有・制御しているため、マネージドメソッドが常にクロール可能なフレームを残すように調整できます。ここでの1つの解決策は、すべてのメソッドに固定的なフレーム形式（たとえば x86 の EBP フレーム形式）を利用することです。しかし実際には、特に小さなリーフメソッド (leaf method)（典型的なプロパティアクセサなど）に対しては、これは非効率的になる可能性があります。

メソッドは通常、そのフレームがクロールされる回数よりもはるかに多くの回数呼び出されます（スタッククロールは、メソッドが通常呼び出される頻度に比べると、ランタイム内では比較的まれです）。そのため、メソッド呼び出しのパフォーマンスを犠牲にして、クロール時の追加処理に充てることは合理的です。その結果、JIT はコンパイルする各メソッドに対して追加のメタデータ (metadata) を生成し、そのメソッドに属するスタックフレームをスタッククローラーがデコードするのに十分な情報を含めます。

💡 初心者向け補足

JIT コンパイラは、メソッドを機械語にコンパイルする際に、「このメソッドのスタックフレームをどう読み解くか」という情報も一緒に記録します。これは「アンワインド情報 (unwind info)」と呼ばれる一種のメタデータです。Java の JVM でも同様に、各メソッドのスタックフレームに関するメタデータ（スタックマップなど）が生成されます。このメタデータがあるからこそ、GC が「このフレームのどこにオブジェクト参照があるか」を正確に特定できるのです。

このメタデータは、メソッド内のどこかの命令ポインタ (instruction pointer) をキーとしたハッシュテーブルの検索によって見つけることができます。JIT はこの追加のメソッドごとのメタデータの影響を最小限にするために、圧縮技術を利用しています。

いくつかの重要なレジスタの初期値（たとえば x86 ベースのシステムでは EIP、ESP、EBP）が与えられると、スタッククローラーはマネージドメソッドとそれに関連する JIT メタデータを見つけ、この情報を使用してレジスタの値をそのメソッドの呼び出し元の時点の値に巻き戻すことができます。このようにして、マネージドメソッドフレームのシーケンスを最新のものから最も古い呼び出し元の順にたどることができます。この操作は *仮想アンワインド (virtual unwind)* と呼ばれることがあります（ESP などの実際の値は更新せず、スタックをそのまま保持するため「仮想」と呼ばれます）。

ランタイムアンマネージドフレーム

ランタイムは部分的にアンマネージドコード（たとえば coreclr.dll）で実装されています。このコードの大部分は特殊であり、*手動マネージド (manually managed)* コードとして動作します。つまり、マネージドコードの多くのルールやプロトコルに従いますが、明示的に制御された方法で行います。たとえば、このようなコードは GC プリエンプティブモード (pre-emptive mode) を明示的に有効化・無効化でき、それに応じてオブジェクト参照の使用を管理する必要があります。

マネージドコードとの注意深い相互作用が関係するもう1つの領域が、スタックウォーク中です。ランタイムのアンマネージドコードの大部分は C++ で記述されているため、マネージドコードのようにメソッドフレーム形式を制御することはできません。同時に、ランタイムのアンマネージドフレームにスタックウォーク中に重要な情報が含まれるケースは多くあります。これには、アンマネージド関数がローカル変数にオブジェクト参照を保持している場合（ガベージコレクション中に報告する必要があります）や、例外処理の場合が含まれます。

各アンマネージドフレームをクロール可能にするのではなく、スタッククロールに報告すべき興味深いデータを持つアンマネージド関数は、情報を Frame と呼ばれるデータ構造にまとめます。この名前の選択は、スタック関連の議論で曖昧さを招く可能性があるため、残念なものです。このドキュメントでは、データ構造としての Frame は常に大文字で記述します。

💡 初心者向け補足

ここで言う「Frame」(大文字) は、C++ のクラス階層として実装されたデータ構造であり、一般的な「フレーム」(小文字) とは異なります。一般的なフレームは関数呼び出しごとにスタック上に自動的に作られる領域ですが、Frame (大文字) はランタイムのアンマネージドコードがスタックウォーカーに情報を伝えるために明示的に作成・管理するオブジェクトです。Java で例えると、JNI (Java Native Interface) 経由でネイティブコードを呼び出す際に JVM が管理するフレーム情報に近い概念です。

Frame は実際には Frame 型の階層全体の抽象基底クラス (abstract base class) です。Frame はサブタイプ化され、スタックウォーカーにとって興味深いさまざまな種類の情報を表現します。

では、スタックウォーカーはこれらの Frame をどのように見つけ、マネージドメソッドのフレームとどのように関連付けるのでしょうか？

各 Frame は単方向リンクリスト (singly linked list) の一部であり、このスレッドのスタック上の次に古い Frame への next ポインタを持ちます (その Frame が最も古い場合は null)。CLR の Thread 構造体は最新の Frame へのポインタを保持しています。アンマネージドランタイムコードは、Thread 構造体と Frame リストを操作することで、必要に応じて Frame をプッシュまたはポップできます。

この方式により、スタックウォーカーはアンマネージド Frame を最新から最古の順にイテレートできます (マネージドフレームがイテレートされるのと同じ順序です)。しかし、マネージドメソッドとアンマネージドメソッドはインターリーブ (交互に混在) できるため、すべてのマネージドフレームを処理してからアンマネージド Frame を処理する、またはその逆を行うのは、実際の呼び出しシーケンスを正確に表現しないため、誤りとなります。

この問題を解決するために、Frame にはさらに制約が課されています。Frame は、Frame リストにプッシュするメソッドのフレーム内のスタック上に割り当てなければなりません。スタックウォーカーは各マネージドフレームのスタック境界を知っているため、単純なポインタ比較を行うことで、特定の Frame が特定のマネージドフレームよりも新しいか古いかを判断できます。

本質的に、スタックウォーカーは現在のフレームをデコードした後、次の (より古い) フレームについて常に2つの選択肢を持ちます。レジスタセットの仮想アンワインドによって決定される次のマネージドフレームか、Thread の Frame リスト上の次に古い Frame のいずれかです。どちらが適切かは、どちらがスタックトップにより近いスタック空間を占めているかを判断することで決定できます。実際の計算はプラットフォームに依存しますが、通常は1つまたは2つのポインタ比較に帰着します。

マネージドコードがアンマネージドランタイムを呼び出す際、いくつかの形式のトランジション Frame (transition Frame) の1つが、アンマネージドのターゲットメソッドによってプッシュされることがよくあります。これは、呼び出し元のマネージドメソッドのレジスタ状態を記録するため (スタックウォーカーがアンマネージド Frame の列挙を終えた後、マネージドフレームの仮想アンワインドを再開できるようにするため) と、多くの場合、マネージドオブジェクト参照がアンマネージドメソッドに引数として渡され、ガベージコレクションが発生した場合に GC に報告する必要があるために必要です。

利用可能な Frame の種類とその用途の完全な説明は、このドキュメントの範囲外です。詳細については [frames.h](#) ヘッダーファイルを参照してください。

スタックウォーカーインターフェイス

完全なスタックウォーカーインターフェイスは、ランタイムのアンマネージドコードにのみ公開されています (マネージドコードには `System.Diagnostics.StackTrace` クラスを通じて簡略化されたサブセットが利用可能です)。一般的なエントリポイントは、ランタイムの Thread クラスの `StackWalkFramesEx()` メソッドです。

このメソッドの呼び出し元は、3つの主要な入力を提供します：

1. ウォークの開始点を示すコンテキスト。これは初期レジスタセット（たとえば、ターゲットスレッドをサスPENDして [GetThreadContext\(\)](#) を呼び出せる場合）、または初期 Frame（対象のコードがランタイムのアンマネージドコード内にあることがわかっている場合）のいずれかです。ほとんどのスタックウォークはスタックのトップから行われますが、正しい開始コンテキストを決定できる場合は、より下の位置から開始することも可能です。
2. 関数ポインタと関連するコンテキスト。提供された関数は、スタックウォーカーが各興味深いフレームに対して（最新から最古の順に）呼び出します。提供されたコンテキスト値は、コールバックの各呼び出しに渡され、ウォーク中に状態を記録したり蓄積したりできるようにします。
3. どの種類のフレームがコールバックをトリガーすべきかを示すフラグ。これにより、呼び出し元はたとえば純粋なマネージドメソッドフレームのみを報告するよう指定できます。完全なリストについては [threads.h](#) ([StackWalkFramesEx\(\)](#) の宣言のすぐ上) を参照してください。

[StackWalkFramesEx\(\)](#) はウォークが正常に終了したか（スタックのベースに到達し、報告すべきメソッドがなくなった）、コールバックの1つによって中断されたか（コールバックはこれを制御するために、スタックウォークに同じ型の enum を返します）、その他の雑多なエラーが発生したかを示す enum 値を返します。

[StackWalkFramesEx\(\)](#) に渡されるコンテキスト値の他に、スタックコールバック関数にはもう1つのコンテキストが渡されます。それは CrawlFrame です。このクラスは [stackwalk.h](#) で定義されており、スタックウォークの進行に伴って収集されるさまざまなコンテキストを含みます。

たとえば、CrawlFrame はマネージドフレームの [MethodDesc*](#) やアンマネージド Frame の [Frame*](#) を示します。また、その時点までフレームを仮想アンワインドすることで推定された現在のレジスタセットも提供します。

スタックウォークの実装の詳細

スタックウォーク実装のさらに低レベルの詳細は、現在このドキュメントの範囲外です。これらに関する知見をお持ちの方は、ぜひこのドキュメントを更新してその知識を共有してください。

System.Private.CoreLib とランタイムへの呼び出し

原文

この章の原文は [System.Private.CoreLib and calling into the runtime](#) です。

はじめに

`System.Private.CoreLib.dll` は、型システム (type system) のコア部分と、.NET Framework の基本クラスライブラリ (Base Class Library) の大部分を定義するためのアセンブリ (assembly) です。もともと .NET Core では `mscorlib` という名前でしたが、コードやドキュメントの多くの場所では依然として `mscorlib` と呼ばれています。この文書では `System.Private.CoreLib` または `CoreLib` を使用するよう努めます。基本データ型 (base data type) はこのアセンブリに存在し、CLR と密結合 (tight coupling) しています。ここでは、`CoreLib` がなぜ特別なのか、そして `QCall` メソッドと `FCall` メソッドを使ってマネージドコード (managed code) から CLR を呼び出す方法の基本について学びます。また、CLR 内部からマネージドコードを呼び出す方法についても説明します。

 初心者向け補足

`System.Private.CoreLib.dll` は、.NET ランタイムで最も基本的なアセンブリです。Java で例えると `rt.jar` (Java Runtime Library) に相当し、`Object`、`String`、`Int32` といったすべてのプログラムが依存する基本的な型がここに定義されています。このアセンブリはランタイム (CLR) と非常に密接に連携しており、通常のライブラリとは異なる特別な扱いを受けます。

依存関係

`CoreLib` は `Object`、`Int32`、`String` といった基本データ型を定義しているため、他のマネージドアセンブリ (managed assembly) に依存することができません。しかし、`CoreLib` と CLR の間には強い依存関係があります。`CoreLib` の多くの型はネイティブコード (native code) からアクセスする必要があるため、多くのマネージド型のレイアウト (layout) はマネージドコードと CLR 内部のネイティブコードの両方で定義されています。また、一部のフィールドは Debug ビルド、Checked ビルド、または Release ビルドでのみ定義される場合があるため、通常 `CoreLib` はビルドの種類ごとに個別にコンパイルする必要があります。

`System.Private.CoreLib.dll` は 64 ビットと 32 ビットで別々にビルドされ、公開するいくつかのパブリック定数はビット数によつて異なります。`IntPtr.Size` などのこれらの定数を使用することで、`CoreLib` より上位のほとんどのライブラリは 32 ビットと 64 ビットで別々にビルドする必要がなくなります。

System.Private.CoreLib が特別である理由

`CoreLib` には多くのユニークな特性があり、その多くは CLR との密結合に起因しています。

- CoreLib は、基本データ型（`Object`、`Int32`、`String` など）のような、CLR の仮想オブジェクトシステム (Virtual Object System) を実装するために必要なコア型を定義します。
- CLR は起動時に特定のシステム型をロードするために CoreLib をロードしなければなりません。
- レイアウトの問題により、プロセス内で一度にロードできる CoreLib は 1 つだけです。複数の CoreLib をロードするには、CLR と CoreLib の間の動作契約 (contract of behavior)、FCall メソッド、データ型レイアウトを形式化し、その契約をバージョン間で比較的安定に保つ必要があります。
- CoreLib の型はネイティブ相互運用 (native interop) で頻繁に使用され、マネージド例外 (managed exception) はネイティブのエラーコードやフォーマットに正しくマッピングされる必要があります。
- CLR の複数の JIT コンパイラ (JIT compiler) は、パフォーマンス上の理由から CoreLib 内の特定のメソッドの小さなグループを特別扱いすることができます。メソッドの最適化による除去（`Math.Cos(double)` など）や、特殊な呼び出し方法（`Array.Length` や、現在のスレッドを取得するための `StringBuilder` の一部の実装詳細など）の両方が含まれます。
- CoreLib は、主に基盤となるオペレーティングシステムや、時にはプラットフォーム適応レイヤー (platform adaptation layer) に対して、必要に応じて P/Invoke を通じてネイティブコードを呼び出す必要があります。
- CoreLib は、ガベージコレクション (garbage collection) のトリガー、クラスのロード、型システムとの複雑なやり取りなど、CLR 固有の機能を公開するために CLR を呼び出す必要があります。これには、マネージドコードと CLR 内の「手動管理された」ネイティブコードとの間のブリッジ (bridge) が必要です。
- CLR は、マネージドメソッドを呼び出したり、マネージドコードでのみ実装されている特定の機能にアクセスするために、マネージドコードを呼び出す必要があります。

初心者向け補足

「密結合 (tight coupling)」とは、2 つのコンポーネントが互いに強く依存している状態を指します。通常のライブラリは実行時に動的にロードされ、ランタイムとは独立していますが、CoreLib はランタイム自体の一部のように機能します。たとえば、`Object` クラスの内部レイアウト（フィールドの並び順やサイズ）は CLR の C++ コード側でも定義されていて、両者が完全に一致していかなければなりません。これは Java の `rt.jar` と JVM の関係に似ています。

マネージドコードと CLR コードの間のインターフェース

改めて述べると、CoreLib のマネージドコードには以下のニーズがあります：

- マネージドコードと CLR 内の「手動管理された」コードの両方で、一部のマネージドデータ構造のフィールドにアクセスする機能。
- マネージドコードから CLR を呼び出せること。
- CLR からマネージドコードを呼び出せること。

これらを実装するには、CLR がネイティブコード内でマネージドオブジェクトのレイアウトを指定し、オプションで検証する方法、ネイティブコードを呼び出すためのマネージド側の仕組み、そしてマネージドコードを呼び出すためのネイティブ側の仕組みが必要です。

ネイティブコードを呼び出すためのマネージド側の仕組みは、`String` のコンストラクタが使用する特殊なマネージド呼び出し規約 (calling convention) もサポートする必要があります。この規約では、コンストラクタがオブジェクトに使用するメモリを自分でアロケートします（GC がメモリをアロケートした後にコンストラクタが呼ばれるという一般的な規約とは異なります）。

CLR は内部的に [mscorlib バインダー \(binder\)](#) を提供しており、アンマネージド型 (unmanaged type) とフィールドからマネージド型とフィールドへのマッピングを提供します。バインダーはクラスの検索とロードを行い、マネージドメソッドの呼び出しを可能にします。また、マネージドコードとネイティブコードの両方で指定されたレイアウト情報の正当性を確認するための簡単な検証も行います。バ

インダーは、ロードしようとしているマネージドクラスが mscorelib に存在すること、ロードされていること、フィールドオフセットが正しいことを確認します。異なるシグネチャを持つメソッドのオーバーロード (overload) を区別する機能も必要です。

マネージドコードからネイティブコードへの呼び出し

マネージドコードから CLR を呼び出すための技術は 2 つあります。FCall は CLR のコードを直接呼び出すことができ、オブジェクトの操作に関して多くの柔軟性を提供しますが、オブジェクト参照を正しく追跡しないと GC ホール (GC hole) を引き起こしやすくなります。QCall も P/Invoke を通じて CLR を呼び出すことができますが、誤った使い方をしてしまう可能性はずっと低くなります。FCall はマネージドコード内で `MethodImplOptions.InternalCall` ビットが設定された extern メソッドとして識別されます。QCall は通常の P/Invoke と同様に `static extern` メソッドとしてマークされますが、`"QCall"` というライブラリに向けられます。

💡 初心者向け補足

FCall と QCall は、.NET ランタイムの「内部 API」を呼び出すための仕組みです。通常のプログラマが直接使うことはありませんが、`String.Length` や `Math.Cos()` といった基本的なメソッドの裏側ではこれらが使われています。Java で例えると、JNI (Java Native Interface) に似た概念ですが、ランタイム内部に特化した仕組みです。P/Invoke は外部の DLL (Windows API など) を呼び出すための公開された仕組みで、JNA に相当します。

FCall、QCall、P/Invoke、マネージドコードでの実装の選択

まず、できる限りマネージドコードで書くべきだということを忘れないでください。潜在的な GC ホールの問題を回避でき、より良いデバッグ体験が得られ、コードもしばしばシンプルになります。

過去に FCall を書く理由は、一般的に 3 つの陣営に分かれています：言語機能の不足、より良いパフォーマンス、またはランタイムとの独自のインタラクションの実装です。C# は今では unsafe コードやスタックアロケートバッファ (stack-allocated buffer) を含め、C++ から得られるほぼすべての有用な言語機能を持っており、これにより FCall の最初の 2 つの理由はなくなりました。過去に FCall に大きく依存していた CLR の一部（リフレクション (Reflection)、一部のエンコーディング (Encoding)、String 操作など）をマネージドコードに移植しており、この流れを続ける予定です。

FCall メソッドを定義する唯一の理由がネイティブメソッドを呼び出すことであるなら、P/Invoke を使用してメソッドを直接呼び出すべきです。`P/Invoke` はパブリックなネイティブメソッドインターフェースであり、正しい方法で必要なすべてのことを行えるはずです。

それでもランタイム内部に機能を実装する必要がある場合は、ネイティブコードへの遷移の頻度を減らす方法がないか検討してください。一般的なケースをマネージドで書き、まれなコーナーケースでのみネイティブに呼び出すことはできませんか？通常、できるだけ多くをマネージドコードに留めておくのが最善です。

QCall は今後の推奨される仕組みです。FCall を使用するのは「仕方がない」場合のみにすべきです。これは、最適化が重要な一般的な「短いパス (short path)」がある場合に発生します。この短いパスは数百命令以下であり、GC メモリのアロケーション、ロックの取得、例外のスローができません（`GC_NOTRIGGER`、`NOTHROWS`）。その他のすべての状況では、QCall を使用すべきです。

FCall は、最適化が必要な短いコードパスのために特別に設計されました。フレーム (frame) を構築するタイミングを明示的に制御できました。しかし、エラーが発生しやすく、多くの API にとってはその複雑さに見合いません。QCall は本質的に P/Invoke です。FCall のパフォーマンスが必要な場合は、QCall を作成して `SuppressGCTransitionAttribute` でマークすることを検討してください。

その結果、QCall は `SafeHandle` に対して有利なマーシャリング (marshaling) を自動的に提供します。ネイティブメソッドは単に `HANDLE` 型を受け取るだけで、そのメソッド本体の実行中に誰かがハンドルを解放するかどうかを心配せずに使用できます。同等の

FCall メソッドでは `SafeHandleHolder` を使用する必要があり、`SafeHandle` を保護する必要があるかもしれません。P/Invoke マーシャラー (marshaler) を活用することで、この追加のプラミングコード (plumbing code) を回避できます。

QCall の機能的動作

QCall は CoreLib から CLR への通常の P/Invoke と非常に似ています。FCall とは異なり、QCall は通常の P/Invoke と同様にすべての引数をアンマネージド型としてマーシャリングします。QCall は通常の P/Invoke と同様にプリエンプティブ GC モード (preemptive GC mode) に切り替えます。これら 2 つの特徴により、QCall は FCall と比較してより信頼性の高いコードを書きやすくなっています。QCall は、FCall でよく見られる GC ホールや GC スタバーション (starvation) バグの影響を受けにくくなっています。

QCall の引数に推奨される型は、P/Invoke マーシャラーによって効率的に処理されるプリミティブ型 (`INT32`、`LPCWSTR`、`BOOL`) です。`BOOL` が QCall の引数に適した真偽値型であることに注意してください。一方、`CLR_BOOL` は FCall の引数に適した真偽値型です。

一般的なアンマネージド EE 構造体へのポインターは、ハンドル型 (handle type) でラップする必要があります。これは、マネージド実装を型安全にし、unsafe C# をいたるところで使うことを避けるためです。例として、[vm\qcall.h](#) の AssemblyHandle を参照してください。

QCall でオブジェクト参照を受け渡しするには、ローカル変数へのポインターをハンドルでラップします。これは意図的に煩雑であり、合理的に可能であれば避けるべきです。以下の例の `StringHandleOnStack` を参照してください。QCall からオブジェクト、特に文字列を返すことは、生のオブジェクトを渡すことが広く許容される唯一の一般的なパターンです。(この制限セットが QCall を GC ホールに対してより安全にする理由については、下記の [「GC ホール、FCall、QCall」](#) セクションをお読みください。)

QCall は C スタイルのメソッドシグネチャで実装する必要があります。これにより、将来の AOT ツーリングがマネージド側の QCall をネイティブ側の実装に接続しやすくなります。

QCall の例 - マネージド側

コメントを実際の QCall 実装にそのまま複製しないでください。これは説明目的です。

CSharp

```
class Foo
{
    // すべての QCall は以下の DllImport 属性を持つべきです
    [DllImport(RuntimeHelpers.QCall, EntryPoint = "Foo_BarInternal", CharSet = CharSet.Unicode)]

    // QCall は常に static extern であるべきです。
    private static extern bool BarInternal(int flags, string inString, StringHandleOnStack retString);

    // 多くの QCall は、遷移前にできるだけ多くの作業を行うための
    // 薄いマネージドラッパーを持っています。例として、
    // ネイティブコードよりもマネージドコードの方が簡単な引数バリデーションがあります。
    public string Bar(int flags)
    {
        if (flags != 0)
            throw new ArgumentException("Invalid flags");

        string retString = null;
        // 文字列は、StringHandleOnStack を使用して
        // ローカル変数のアドレスを取得することで QCall から返されます
        if (!BarInternal(flags, this.Id, new StringHandleOnStack(ref retString)))
            return null;

        return retString;
    }
}
```

```

        FatalError();

        return retString;
    }
}

```

QCall の例 - アンマネージド側

コメントを実際の QCall 実装にそのまま複製しないでください。

QCall のエントリポイント (entry point) は、`DllImportEntry` マクロを使用して `vm\qcallentrypoints.cpp` のテーブルに登録する必要があります。下記の [「QCall または FCall メソッドの登録」](#) を参照してください。

C++

```

// すべての QCall はフリー関数であり、QCALLTYPE と extern "C" でタグ付けされるべきです
extern "C" BOOL QCALLTYPE Foo_BarInternal(int flags, LPCWSTR wszString, QCall::StringHandleOnStack retString)
{
    // すべての QCall は QCALL_CONTRACT を持つべきです。
    // これは THROWS; GC_TRIGGERED; MODE_PREEMPTIVE のエイリアスです。
    QCALL_CONTRACT;

    // オプションとして、前提条件を指定したい場合は
    // QCALL_CHECK と契約の展開形式を使用します：
    // CONTRACTL {
    //     QCALL_CHECK;
    //     PRECONDITION(wszString != NULL);
    // } CONTRACTL_END;

    // QCALL_CONTRACT と BEGIN_QCALL の間には、
    // 戻り値の宣言（ある場合）のみが置かれるべきです。
    BOOL retVal = FALSE;

    // 本体は BEGIN_QCALL/END_QCALL マクロで囲む必要があります。
    // これは例外処理に必要です。
    BEGIN_QCALL;

    // 引数のバリデーションは理想的にはマネージド側で行うべきですが、
    // 場合によってはネイティブ側で行う必要があります。引数のバリデーションが
    // マネージド側で行われている場合、ネイティブ側でのアサートは妥当です。
    _ASSERTE(flags != 0);

    // QCall に渡された文字列の GC による移動を心配する必要はありません。
    // マーシャリングが文字列をピン留めしてくれます。
    printf("%S\n", wszString);

    // これは文字列をマネージドコードに返す最も効率的な方法です。
    // StringBuilder を使用する必要はありません。
    retString.Set(L"Hello");

    // BEGIN_QCALL/END_QCALL の内部から return することはできません。
    // 戻り値はヘルパー変数を通じて渡す必要があります。
    retVal = TRUE;

    END_QCALL;
}

```

```
    return retVal;  
}
```

FCall の機能的動作

FCall はオブジェクト参照の受け渡しに関してより柔軟性がありますが、コードの複雑さが増し、ミスの機会も多くなります。さらに、無視できない長さの FCall に対しては、ガベージコレクションを実行する必要があるかどうかを明示的にポーリングしてください。これを怠ると、マネージドコードがタイトループ (tight loop) で FCall メソッドを繰り返し呼び出す場合にスタバーション (starvation) の問題につながります。なぜなら、FCall はスレッドが GC の実行を協調方式 (cooperative manner) でのみ許可している間に実行されるからです。

FCall には大量のボイラープレートコード (boilerplate code) が必要であり、ここで説明するには多すぎます。詳細は [fcall.h](#) を参照してください。

GC ホール、FCall、QCall

GC ホールに関するより完全な議論は [CLR Code Guide](#) にあります。「[Is your code GC-safe?](#)」を探してください。このカスタマイズされた議論は、FCall と QCall が奇妙な規約を持つ理由のいくつかを動機付けています。

FCall メソッドにパラメーターとして渡されたオブジェクト参照は GC 保護されません。つまり、GC が発生した場合、それらの参照はオブジェクトの新しい場所ではなくメモリ内の古い場所を指すことになります。このため、FCall は通常、パラメーター型として `StringObject*` のようなものを受け取り、GC をトリガーする可能性のある操作を行う前に明示的にそれを `STRINGREF` に変換するという規律に従います。オブジェクト参照を後で使用することが予想される場合は、GC をトリガーする前にオブジェクト参照を GC 保護する必要があります。

`OBJECTREF` を適切に報告しなかったり、内部ポインター (interior pointer) を更新しなかったりすることは、一般的に「GC ホール」と呼ばれます。`OBJECTREF` クラスは Debug ビルドと Checked ビルドで逆参照 (dereference) するたびに、有効なオブジェクトを指しているかどうかの検証を行うからです。無効なオブジェクトを指している `OBJECTREF` が逆参照されると、「Detected an invalid object reference. Possible GC hole?」のようなメッセージでアサート (assert) がトリガーされます。このアサートは「手動管理された」コードを書く際に残念ながら容易にヒットします。

QCall のプログラミングモデルは、スタック上のオブジェクト参照のアドレスを渡すことを強制しているため、GC ホールを回避するようになっています。これにより、オブジェクト参照が JIT のレポートロジックによって GC 保護されること、そして実際のオブジェクト参照は GC ヒープにアロケートされていないため移動しないことが保証されます。QCall は、GC ホールを書きにくくするために、まさに推奨されるアプローチです。

💡 初心者向け補足

「GC ホール」とは、ガベージコレクション (GC) が発生したときにオブジェクトへの参照が無効になってしまうバグのことです。GC はメモリ上のオブジェクトを移動 (コンパクション) することがありますが、その際にすべての参照が新しいアドレスに更新される必要があります。FCall ではネイティブコード内で生のポインターを扱うため、GC が参照を自動的に更新できず、古いアドレスを指したままになってしまい危険があります。これが「穴 (hole)」と呼ばれる由来です。QCall はこの問題を構造的に回避するよう設計されているため、より安全です。

x86 用の FCall エピローグウォーカー (epilog walker)

マネージドスタックウォーカー (stack walker) は FCall からの復帰方法を見つけられる必要があります。ABI の一部としてスタック巻き戻し (stack unwinding) の規約を定義している新しいプラットフォームでは比較的簡単です。x86 では ABI によるスタック巻き戻しの規約が定義されていません。ランタイムはエピローグウォーカーを実装することでこれを回避しています。エピローグウォーカーは FCall の実行をシミュレートすることで、FCall のリターンアドレスとカリーセーブレジスター (callee save register) を計算します。これにより、FCall 実装で許可されるコンストラクト (construct) に制限が課されます。

デストラクタを持つスタックアロケートされたオブジェクトや、FCall 実装内の例外処理のような複雑なコンストラクトは、エピローグウォーカーを混乱させる可能性があります。これにより、GC ホールやスタックウォーキング (stack walking) 中のクラッシュが発生する可能性があります。このクラスのバグを防ぐために避けるべきコンストラクトの包括的なリストはありません。ある日問題なく動作する FCall 実装が、次の C++ コンパイラの更新で壊れる可能性があります。この領域のバグを見つけるために、ストレステスト (stress run) とコードカバレッジ (code coverage) に依存しています。

FCall の例 – マネージド側

`String` クラスからの実際の例を示します：

```
public partial sealed class String
{
    [MethodImpl(MethodImplOptions.InternalCall)]
    private extern string? IsInterned();

    public static string? IsInterned(string str)
    {
        if (str == null)
        {
            throw new ArgumentNullException(nameof(str));
        }

        return str.IsInterned();
    }
}
```

CSharp

FCall の例 – アンマネージド側

FCall のエントリポイン트は、`FCFuncEntry` マクロを使用して `vm\ecalllist.h` のテーブルに登録する必要があります。[「QCall または FCall メソッドの登録」](#) を参照してください。

この例は、マネージドオブジェクト（`Object*`）を生のポインターとして受け取る FCall メソッドを示しています。これらの生の入力は「unsafe」と見なされ、GC の影響を受ける文脈で使用する場合はバリデーションまたは変換する必要があります。

```
FCIMPL1(FC_BOOL_RET, ExceptionNative::IsImmutableAgileException, Object* pExceptionUNSAFE)
{
    FCALL_CONTRACT;

    ASSERT(pExceptionUNSAFE != NULL);

    OBJECTREF pException = (OBJECTREF) pExceptionUNSAFE;

    FC_RETURN_BOOL(CLRException::IsPreallocatedExceptionObject(pException));
```

C++

```
}
```

```
FCIMPLEND
```

QCall または FCall メソッドの登録

CLR は、マネージドクラスとメソッド名の観点から、などのネイティブメソッドを呼び出すかという観点から、QCall と FCall メソッドの名前を知っている必要があります。FCall の場合、登録は [ecalllist.h](#) で、2 つの配列を使って行われます。最初の配列は名前空間 (namespace) とクラス名を関数要素の配列にマッピングします。その関数要素の配列は、個々のメソッド名とシグネチャを関数ポインターにマッピングします。

上記の例で `String.IsInterned()` の FCall メソッドを定義したとします。まず、String クラスの関数要素の配列があることを確認する必要があります。

```
// これらは name:namespace ペアでソートされたままである必要があることに注意：  
...  
FCClassElement("String", "System", gStringFuncs)  
...
```

C++

次に、`gStringFuncs` に `IsInterned` の適切なエントリが含まれていることを確認する必要があります。メソッド名に複数のオーバーロードがある場合は、シグネチャを指定できることに注意してください：

```
FCFuncStart(gStringFuncs)  
...  
FCFuncElement("IsInterned", AppDomainNative::IsStringInterned)  
...  
FCFuncEnd()
```

C++

QCall は [qcallentrypoints.cpp](#) の `s_QCall` 配列に、`DllImportEntry` マクロを使用して以下のように登録されます：

```
static const Entry s_QCall[] =  
{  
    ...  
    DllImportEntry(MyQCall),  
    ...  
};
```

C++

命名規約

FCall と QCall はパブリックに公開すべきではありません。代わりに、実際の FCall または QCall をラップし、API 承認された名前を提供してください。

内部の FCall または QCall は、FCall/QCall の名前をパブリックエントリポイントと区別するために「Internal」サフィックスを使用する必要があります（例：パブリックエントリポイントがエラーチェックを行い、まったく同じシグネチャの共有ワーカー関数を呼び出す場合）。これは、BCL の純粋なマネージドコードでこのような状況に対処する方法と何ら変わりません。

マネージド/アンマネージドの二重性を持つ型

特定のマネージド型は、マネージドコードとネイティブコードの両方で表現が利用可能でなければなりません。型の正規の定義がマネージドコードにあるのか CLR 内のネイティブコードにあるのかと問うことはできますが、答えは重要ではありません。重要なのは、両方が同一でなければならぬということです。これにより、CLR のネイティブコードはマネージドオブジェクト内のフィールドに高速かつ効率的にアクセスできるようになります。[MethodTable](#) や [FieldDesc](#) に対して本質的に CLR のリフレクション相当のものを使用してフィールド値を取得する、より複雑な方法もありますが、これは望ましいパフォーマンスが得られず、使い勝手も良くありません。よく使用される型については、ネイティブコードでデータ構造を宣言し、両者を同期させておくことが理にかなっています。

CLR はこの目的のためにバインダー (binder) を提供しています。マネージドクラスとネイティブクラスを定義した後、フィールドオフセットが同じままであることを確認し、誰かが誤って一方の型定義にのみフィールドを追加した場合にはばやく検出できるよう、バインダーにいくつかの手がかりを提供する必要があります。

[corelib.h](#) では、「_U」で終わるマクロを使用して、型、マネージドコードのフィールド名、および対応するネイティブデータ構造のフィールド名を記述します。さらに、メソッドのリストを指定し、後で呼び出しを試みるときに名前で参照することができます。

```
 DEFINE_CLASS_U(SAFE_HANDLE,           Interop,          SafeHandle,        SafeHandle)
 DEFINE_FIELD(SAFE_HANDLE,             HANDLE,           handle)
 DEFINE_FIELD_U(SAFE_HANDLE,           STATE,            _state,           SafeHandle,        m_st)
 DEFINE_FIELD_U(SAFE_HANDLE,           OWNS_HANDLE,      _ownsHandle,      SafeHandle,        m_ow)
 DEFINE_FIELD_U(SAFE_HANDLE,           INITIALIZED,      _fullyInitialized, SafeHandle,        m_fu)
 DEFINE_METHOD(SAFE_HANDLE,           GET_IS_INVALID,   get_IsInvalid,   IM_RetBool)
 DEFINE_METHOD(SAFE_HANDLE,           RELEASE_HANDLE,   ReleaseHandle,  IM_RetBool)
 DEFINE_METHOD(SAFE_HANDLE,           DISPOSE,          Dispose,          IM_RetVoid)
 DEFINE_METHOD(SAFE_HANDLE,           DISPOSE_BOOL,     Dispose,          IM_Boolean_RetVoid)
```

これで、[REF<T>](#) テンプレート (template) を使用して [SAFEHANDLEREF](#) のような型名を作成できます。[OBJECTREF](#) のすべてのエラーチェックは [REF<T>](#) テンプレートに組み込まれており、この [SAFEHANDLEREF](#) を自由に逆参照してネイティブコードでそのフィールドを使用できます。ただし、これらの参照は引き続き GC 保護する必要があります。

アンマネージドコードからマネージドコードへの呼び出し

CLR がネイティブからマネージドコードを呼び出す必要がある場所は明らかに存在します。この目的のために、多くのプログラミングを処理してくれる [MethodDescCallSite](#) クラスが追加されています。概念的には、呼び出したいメソッドの [MethodDesc*](#) を見つけ、「this」ポインターのマネージドオブジェクト（インスタンスマソッドを呼び出す場合）を見つけ、引数の配列を渡し、戻り値を処理するだけです。内部的には、GC がプリエンプティブモード (preemptive mode) で実行できるようにスレッドの状態を切り替える必要があるかもしれません。

以下は簡略化された例です。このインスタンスが、前のセクションで説明したバインダーを使用して [SafeHandle](#) の仮想 [ReleaseHandle](#) メソッドを呼び出していることに注目してください。

C++

```

void SafeHandle::RunReleaseMethod(SafeHandle* psh)
{
    CONTRACTL {
        THROWS;
        GC_TRIGGERES;
        MODE_COOPERATIVE;
    } CONTRACTL_END;

    SAFEHANDLEREF sh(psh);

    GCPROTECT_BEGIN(sh);

    MethodDescCallSite releaseHandle(s_pReleaseHandleMethod, METHOD__SAFE_HANDLE__RELEASE_HANDLE, (OBJECTREF*)&sh,
        ARG_SLOT releaseArgs[] = { ObjToArgSlot(sh) };
        if (!(BOOL)releaseHandle.Call_Boolean(releaseArgs)) {
            MDA_TRIGGER_ASSISTANT(ReleaseHandleFailed, ReportViolation)(sh->GetTypeHandle(), sh->m_handle);
        }

        GCPROTECT_END();
    }
}

```

初心者向け補足

上記のコードでは、ネイティブ（C++）側からマネージド（C#）側のメソッドを呼び出しています。 `GCPROTECT_BEGIN` / `GCPROTECT_END` マクロは、呼び出し中に GC が発生してもオブジェクト参照が正しく追跡されるようにするための仕組みです。
`CONTRACTL` ブロックはメソッドの「契約」を定義しており、このメソッドが例外をスローする可能性があること（`THROWS`）、GC をトリガーすること（`GC_TRIGGERES`）、協調モード（`MODE_COOPERATIVE`）で実行されることを宣言しています。

他のサブシステムとの相互作用

デバッガー

現在の FCall の制限の 1 つは、マネージドコードと FCall の両方を Visual Studio のインターチェンジ（またはミックスモード）で簡単にデバッグできないことです。現在、FCall にブレークポイントを設定してインターチェンジでデバッグすることは、うまく機能しません。これはおそらく修正されないでしょう。

物理アーキテクチャ

CLR が起動するとき、CoreLib は `SystemDomain::LoadBaseSystemClasses()` というメソッドによってロードされます。ここで、基本データ型や類似のクラス（`Exception` など）がロードされ、CoreLib の型を参照するための適切なグローバルポインターが設定されます。

FCall については、インフラストラクチャは `fcall.h` を参照し、FCall メソッドをランタイムに適切に通知するには `ecalllist.h` を参照してください。

QCall については、関連するインフラストラクチャは `qcall.h` を参照し、QCall メソッドをランタイムに適切に通知するには `qcallentrypoints.cpp` を参照してください。

より一般的なインフラストラクチャとネイティブ型定義は `object.h` にあります。バインダーはマネージドクラスとネイティブクラスを関連付けるために `mscorlib.h` を使用します。

データアクセスコンポーネント (DAC) ノート

原文

この章の原文は [Data Access Component \(DAC\) Notes](#) です。

日付: 2007年

マネージドコードのデバッグには、マネージドオブジェクト (managed objects) やマネージド構造体 (managed constructs) に関する特別な知識が必要です。たとえば、オブジェクトにはデータそのものに加えて、さまざまな種類のヘッダー情報があります。ガベージコレクタ (GC) が動作するにつれて、オブジェクトはメモリ内を移動することがあります。型情報の取得にはローダー (loader) の助けが必要な場合があります。エディットアンドcontiヌ (edit-and-continue) を経た関数の正しいバージョンの取得や、リフレクション (reflection) を通じて生成された関数の情報取得には、デバッガが EnC のバージョン番号やメタデータ (metadata) を認識している必要があります。デバッガはアプリケーションドメイン (AppDomain) やアセンブリ (assembly) を区別できなければなりません。VM ディレクトリのコードは、これらのマネージド構造体に必要な知識を具現化しています。これは本質的に、マネージドコードやデータに関する情報を取得するための API が、実行エンジン (execution engine) 自体が実行するものと同じアルゴリズムの一部を実行しなければならないことを意味します。

💡 初心者向け補足

DAC は「Data Access Component (データアクセスコンポーネント)」の略で、.NET ランタイムのデバッグ機能を支える重要な仕組みです。通常、デバッガがマネージドコードの内部状態を調べるには、ランタイムの内部構造を理解する必要があります。DAC は、デバッガがプロセス外 (out-of-process) からでもランタイムの内部データにアクセスできるようにするためのコンポーネントです。Java で例えるなら、JVM の内部状態をデバッガツール (jmap や jstack など) が読み取る仕組みに相当します。

デバッガは インプロセス (in-process) または アウトオブプロセス (out-of-process) のいずれかで動作できます。インプロセスで実行されるデバッガは、ライブなデータターゲット (デバッグ対象プロセス、デバッギー) を必要とします。この場合、ランタイムがロード済みで、ターゲットは実行中です。デバッギー内のヘルパースレッドが実行エンジンのコードを実行して、デバッガに必要な情報を計算します。ヘルパースレッドはターゲットプロセス内で実行されるため、ターゲットのアドレス空間とランタイムコードに直接アクセスできます。すべての計算はターゲットプロセス内で行われます。これは、デバッガがマネージド構造体を意味のある方法で表現するために必要な情報を取得するシンプルな方法です。それにもかかわらず、インプロセスデバッガには一定の制限があります。たとえば、デバッギーが現在実行されていない場合 (デバッギーがダンプファイルの場合など)、ランタイムはロードされておらず (マシン上で利用できない場合もあります)、デバッガには必要な情報を取得するためのランタイムコードを実行する手段がありません。

歴史的に、CLR デバッガはインプロセスで動作していました。デバッガ拡張である SOS (Son of Strike) や Strike (初期の CLR 時代) を使用して、マネージドコードを検査できます。.NET Framework 4 以降、デバッガはアウトオブプロセスで実行されます。CLR デバッガ API は、SOS の機能の多くと、SOS が提供しないその他の機能を提供します。SOS と CLR デバッグ API の両方が、アウトオブプロセスデバッグを実装するためにデータアクセスコンポーネント (DAC) を使用します。DAC は概念的には、ランタイムの実行エンジンコードのサブセットであり、アウトオブプロセスで実行されます。これは、ランタイムがインストールされていないマシン上であっても、ダンプファイルに対して操作できることを意味します。その実装は主に、マクロとテンプレートのセット、および実行エンジンのコードの条件付きコンパイルで構成されています。ランタイムがビルドされると、clr.dll と mscore.dll の両方が生成されます。CoreCLR ビルドでは、バイナリは若干異なり、coreclr.dll と msdaccore.dll になります。OS X などの他のオペレーティングシステム向けにビルドした場合も、ファイル名は異なります。ターゲットを検査するために、DAC はそのメモリを読み取って mscore.dll の VM コードへの入力を取得できます。その後、ホスト内で適切な関数を実行してマネージド構造体に関する必要な情報を計算し、最終的に結果をデバッガに返します。

DAC は ターゲットプロセスのメモリを読み取ることに注意してください。デバッガとデバッギーは、別々のアドレス空間を持つ別々のプロセスであることを認識することが重要です。したがって、ターゲットメモリとホストメモリを明確に区別することが重要です。ホストプロセスで実行されるコード内でターゲットアドレスを使用すると、完全に予測不能で一般的に不正な結果になります。DAC を使用してターゲットからメモリを取得する際は、正しいアドレス空間からのアドレスを使用するよう非常に注意する必要があります。さらに、ターゲット

ゲットアドレスが純粋にデータとして使用される場合もあります。この場合、ホストアドレスを使用することは同様に不正です。たとえば、マネージド関数に関する情報を表示するために、その開始アドレスとサイズをリストしたい場合があります。ここでは、ターゲットアドレスを提供することが重要です。DAC が実行する VM 内のコードを書く際は、ホストアドレスとターゲットアドレスのどちらを使用するかを正しく選択する必要があります。

💡 初心者向け補足

ホストとターゲットの違いは DAC を理解する上で最も重要な概念です。「ターゲット (target)」はデバッグ対象のプロセス（またはダンプファイル）であり、「ホスト (host)」はデバッガが動作しているプロセスです。これらは異なるアドレス空間を持つ別々のプロセスです。ターゲットのアドレス `0x12345678` に格納されている値は、ホストのアドレス `0x12345678` にある値とはまったく無関係です。DAC はこの2つの間でデータを安全に橋渡しする役割を果たします。

DAC インフラストラクチャ（ホストまたはターゲットのメモリアクセス方法を制御するマクロとテンプレート）は、どのポインタがホストアドレスでどのポインタがターゲットアドレスであるかを区別する特定の規約を提供します。関数が DAC 化 (DACized) されると（つまり、DAC インフラストラクチャを使用して関数をアウトオブプロセスで動作させると）、型 `T` のホストポインタは `T *` 型として宣言されます。ターゲットポインタは `PTR_T` 型です。ただし、ホスト対ターゲットの概念は DAC にのみ意味があることを覚えておいてください。非 DAC ビルドでは、アドレス空間は1つだけです。ホストとターゲットは同じ、つまり CLR です。VM 関数内で `T *` 型または `PTR_T` 型のいずれかの型でローカル変数を宣言した場合、それは「ホストポインタ」になります。clr.dll (coreclr.dll) 内でコードを実行している場合、`T *` 型のローカル変数と `PTR_T` 型のローカル変数の間にはまったく違いがありません。同じソースから msordacwks.dll (msdaccore.dll) にコンパイルされた関数を実行する場合、`T *` 型で宣言された変数は、デバッガをホストとする真のホストポインタになります。考えてみれば、これは明白です。しかし、これらのポインタを他の VM 関数に渡し始めるとき、混乱しやすくなります。関数を DAC 化する（つまり、適宜 `T *` を `PTR_T` に変更する）際は、ホスト型とターゲット型のどちらにすべきかを判断するために、ポインタの出所を追跡する必要がある場合があります。

DAC を理解していないと、DAC インフラストラクチャの使用を煩わしく感じやすいものです。`TADDR` や `PTR_this`、`dac_cast` など、コードを散らかして理解しにくくしているように見えます。しかし、少し学べば、これらが本当は難しくないことがわかるでしょう。ホストとターゲットのアドレスを明示的に異なるものにすることは、実際には厳密な型付け (strong typing) の一形態です。より注意深く行えば行うほど、コードの正しさを確保しやすくなります。

DAC はダンプに対して操作する可能性があるため、msordacwks.dll (msdaccore.dll) にビルドする VM ソースの部分は非侵襲的 (non-invasive) でなければなりません。具体的には、ターゲットのアドレス空間への書き込みを引き起こすようなことは通常行いたくありませんし、即座のガベージコレクションを引き起こす可能性のあるコードも実行できません。（GC を遅延できる場合は、アロケーションが可能な場合があります。）ホストの状態は常に変更されます（テンポラリ、スタック値、ローカルヒープ値）。問題となるのは ターゲット 空間の変更のみです。これを強制するために、コードのファクタリング (code factoring) と条件付きコンパイル (conditional compilation) の2つのことを行います。理想的な世界では、VM コードをファクタリングして、侵襲的な操作を非侵襲的な関数とは別の関数に厳密に分離します。

残念ながら、大規模なコードベースがあり、その大部分は DAC のことをまったく考えずに書かれたものです。「検索または作成 (find or create)」のセマンティクスを持つ関数が多数あり、検査のみを行う部分とターゲットに書き込む部分を持つ関数も多数あります。これは関数に渡されるフラグで制御される場合もあります。これはローダーコードなどでよく見られます。DAC を使用する前にすべての VM コードのリファクタリングという膨大な作業を完了させなくとも済むように、アウトオブプロセスからの侵襲的コードの実行を防ぐ第二の方法があります。定義済みプリプロセッサ定数 `DACCESS_COMPILE` を使用して、DAC にコンパイルするコードの部分を制御します。

`DACCESS_COMPILE` 定数はできるだけ使用を少なくしたいので、新しいコードパスを DAC 化する場合は、可能な限りリファクタリングを優先します。したがって、「検索または作成」のセマンティクスを持つ関数は、2つの関数に分割すべきです。1つは情報を検索しようとする関数、もう1つはそれを呼び出して検索が失敗した場合に作成するラッパーです。そうすれば、DAC コードパスは検索関数を直接呼び出して作成を回避できます。

DAC はどのように動作するか

前述のとおり、DAC は必要なデータをマーシャリング (marshaling) し、mscordacwks.dll (msdaccore.dll) モジュール内のコードを実行することで動作します。ターゲットアドレス空間から読み取ってターゲットの値を取得し、mscordacwks の関数が操作できるようにホストアドレス空間に格納することでデータをマーシャリングします。これはオンデマンドでのみ行われるため、mscordacwks の関数がターゲットの値を必要としなければ、DAC はそれをマーシャリングしません。

初心者向け補足

マーシャリング (marshaling) とは、あるプロセスのメモリ空間にあるデータを別のプロセスで使えるようにコピー・変換する処理のことです。Java の RMI (Remote Method Invocation) におけるオブジェクトのシリアル化に似た概念です。DAC の場合、ターゲット（デバッグ対象）のメモリからデータを読み取り、ホスト（デバッガ）のメモリ空間にキャッシュすることで、デバッガが安全にそのデータを参照できるようにします。

マーシャリングの原則

DAC は読み取ったデータのキャッシュ (cache) を維持します。これにより、同じ値を繰り返し読み取るオーバーヘッドを回避します。もちろん、ターゲットがライブの場合、値は変更される可能性があります。キャッシュされた値が有効であると仮定できるのは、デバッガーが停止したままである間だけです。ターゲットの実行を継続させたら、DAC のキャッシュをフラッシュする必要があります。デバッガがさらなる検査のためにターゲットを停止させたとき、DAC は再び値を取得します。DAC キャッシュのエントリは `DAC_INSTANCE` 型です。これには（他のデータとともに）ターゲットアドレス、データのサイズ、およびマーシャリングされたデータ自体のスペースが含まれます。DAC がデータをマーシャリングすると、このエントリのマーシャリングされたデータ部分のアドレスをホストアドレスとして返します。

DAC がターゲットから値を読み取ると、その値を（型によって決定される）所定のサイズのバイトチャunkとしてマーシャリングします。ターゲットアドレスをキャッシュエントリのフィールドとして保持することで、ターゲットアドレスとホストアドレス（キャッシュ内のアドレス）の間のマッピングを維持します。デバッガセッションの停止と再開の間で、同じ型を使用した後続のアクセスである限り、DAC は要求された各値を一度だけマーシャリングします。（ターゲットアドレスを2つの異なる型で参照する場合、サイズが異なる可能性があるため、DAC は新しい型のための新しいキャッシュエントリを作成します。）値がすでにキャッシュにある場合、DAC はそのターゲットアドレスで検索できます。つまり、同じ型を使用して両方のポインタにアクセスしている限り、2つのホストポインタの等値（不等値）比較を正しく行えます。ただし、このポインタの同一性は型変換をまといでは保持されません。さらに、別々にマーシャリングされた値がキャッシュ内でターゲットと同じ空間的関係を維持するという保証はないため、2つのホストポインタの大小関係の比較は不正です。オブジェクトのレイアウトはホストとターゲットで同一でなければならぬため、ターゲットで使用するのと同じオフセットを使用してキャッシュ内のオブジェクトのフィールドにアクセスできます。マーシャリングされたオブジェクトのポインタフィールドはターゲットアドレスになることを覚えておいてください（通常、`PTR` 型のデータメンバとして宣言されます）。それらのアドレスの値が必要な場合、DAC は参照解除 (dereference) する前にそれらをホストにマーシャリングする必要があります。

この DLL は mscorewks.dll (coreclr.dll) のビルドに使用するのと同じソースからビルドするため、デバッガが使用する mscordacwks.dll (msdaccore.dll) のビルドは mscorewks のビルドと正確に一致する必要があります。ビルド間で使用する型のフィールドを追加または削除することを考えると、これが明白に正しいことがわかります。mscorewks 内のオブジェクトのサイズは mscordacwks のサイズと異なり、DAC はオブジェクトを正しくマーシャリングできなくなります。これは考えてみれば明白ですが、見落としやすい影響があります。DAC ビルドにのみ存在する、または非 DAC ビルドにのみ存在するオブジェクト内のフィールドを持つことはできません。したがって、以下のような宣言は不正な動作につながります。

```
class Foo
{
    ...
    int nCount;

    // これは絶対にやってはいけない!! DAC ビルドではオブジェクトレイアウトが一致しなければならない
```

```

#ifndef DACCESS_COMPILE

    DWORD dwFlags;

#endif

PTR_Bar pBar;
...

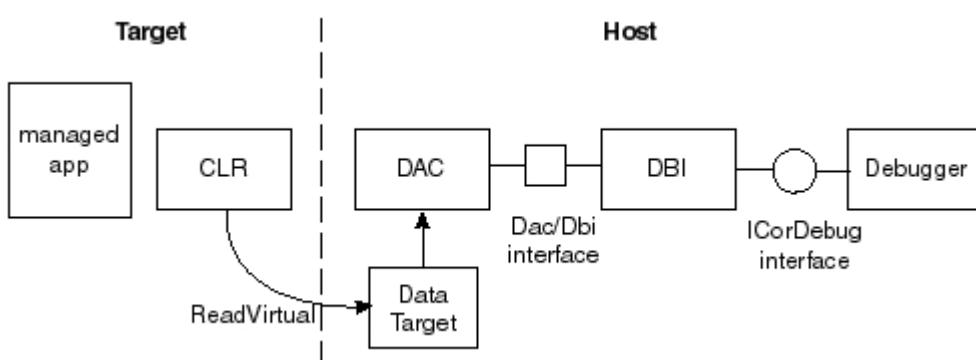
};


```

マーシャリングの詳細

DAC のマーシャリングは、DAC ビルドと非 DAC ビルドで一般的に異なる意味を持つ `typedef`、マクロ、テンプレート型のコレクションを通じて機能します。これらの宣言は [src\inc\daccess.h](#) にあります。このファイルの冒頭には、DAC を使用するコードを書くために必要な詳細を説明する長いコメントもあります。

マーシャリングの仕組みを理解するために、例が役立つかかもしれません。一般的なデバッグシナリオは、以下のブロック図に示されています。



この図のデバッガは、Visual Studio、MDbg、WinDbg などです。デバッガは、必要な情報を取得するために CLR デバッガインターフェース (DBI) API とやり取りします。ターゲットから取得しなければならない情報は DAC を経由します。デバッガはデータターゲットを実装し、ターゲット内のメモリを読み取るための `ReadVirtual` 関数の実装を担当します。図の点線はプロセス境界を表しています。

デバッガが、マネージドスタックから取得したマネージドアプリケーション内の ngen されたメソッドの開始アドレスを表示する必要があるとします。デバッガは既に DBI から `ICorDebugFunction` のインスタンスを取得していると仮定します。まず DBI API `ICorDebugFunction::GetNativeCode` を呼び出します。これは DAC/DBI インターフェース関数 `GetNativeCodeInfo` を通じて DAC を呼び出し、関数のドメインファイル (domain file) とメタデータトークン (metadata token) を渡します。以下のコードフラグメントは実際の関数の簡略版ですが、余分な詳細を導入せずにマーシャリングを示しています。

```

void DacDbiInterfaceImpl::GetNativeCodeInfo(TADDR taddrDomainFile,
                                            mdToken functionToken,
                                            NativeCodeFunctionData * pCodeInfo)
{
    ...
}

```

```

DomainFile * pDomainFile = dac_cast<PTR_DomainFile>(taddrDomainFile);

Module * pModule = pDomainFile->GetCurrentModule();

MethodDesc* pMethodDesc = pModule->LookupMethodDef (functionToken);

pCodeInfo->pNativeCodeMethodDescToken = pMethodDesc;

// モジュールのロード中で、以前に設定されたブレークポイントをバインドしようとしている場合、

// まだメソッド記述子がない可能性があるため、その状況をチェックする

if(pMethodDesc != NULL)

{

    pCodeInfo->startAddress = pMethodDesc->GetNativeCode();

    ...

}

}

```

最初のステップは、マネージド関数が存在するモジュール (module) を取得することです。渡す `taddrDomainFile` パラメータはターゲットアドレスを表しますが、ここでそれを参照解除 (dereference) できる必要があります。つまり、DAC に値をマーシャリングさせる必要があります。`dac_cast` 演算子は、`domainFileTaddr` の値に等しいターゲットアドレスを持つ `PTR_DomainFile` の新しいインスタンスを構築します。これを `pDomainFile` に代入すると、ホストポインタ型への暗黙の変換が行われます。この変換演算子は `PTR` 型のメンバであり、ここでマーシャリングが発生します。DAC はまずそのキャッシュでターゲットアドレスを検索します。見つからない場合、マーシャリングされた `DomainFile` インスタンスのためにターゲットからデータを読み取り、キャッシュにコピーします。最後に、マーシャリングされた値のホストアドレスを返します。

これで、この `DomainFile` のホストインスタンスに対して `GetCurrentModule` を呼び出せます。この関数は `DomainFile::m_pModule` を返すシンプルなアクセサです。`Module *` を返すことについて注意してください。これはホストアドレスになります。`m_pModule` の値はターゲットアドレスです (DAC は `DomainFile` インスタンスを生のバイトとしてコピーしたため)。しかし、フィールドの型は `PTR_Module` であるため、関数がそれを返す際、DAC は `Module *` への変換の一部として自動的にマーシャリングします。つまり、戻り値はホストアドレスです。これで正しいモジュールとメソッドトークンが得られたので、`MethodDesc` を取得するために必要なすべての情報があります。

```

Module * DomainFile::GetCurrentModule()

{
    LEAF_CONTRACT;

    SUPPORTS_DAC;

    return m_pModule;
}

```

このコードの簡略版では、メソッドトークンがメソッド定義であると仮定しています。次のステップは、`Module` インスタンスに対して `LookupMethodDef` 関数を呼び出すことです。

```

inline MethodDesc *Module::LookupMethodDef(mdMethodDef token)

{
    WRAPPER_CONTRACT;
}

```

```

SUPPORTS_DAC;

...
return dac_cast<PTR_MethodDesc>(GetFromRidMap(&m_MethodInfoToDescMap, RidFromToken(token)));
}

```

これは `RidMap` を使用して `MethodInfo` を検索します。この関数の定義を見ると、`TADDR` を返すことがわかります。

```

TADDR GetFromRidMap(LookupMap *pMap, DWORD rid)
{
    ...
    TADDR result = pMap->pTable[rid];
    ...
    return result;
}

```

これはターゲットアドレスを表しますが、実際にはポインタではなく、単なる数値です（アドレスを表しているとはいえ）。問題は、`LookupMethodDef` が参照解除できる `MethodInfo` のアドレスを返す必要があることです。これを実現するために、関数は `dac_cast` を使用して `TADDR` を `PTR_MethodInfo` に変換します。これは、ターゲットアドレス空間における `void *` から `MethodInfo *` へのキャストの形式を考えることができます。実際、`GetFromRidMap` が `TADDR`（整数セマンティクス）の代わりに `PTR_VOID`（ポインタセマンティクス）を返せば、このコードはわずかにクリーンになるでしょう。ここでも、`return` 文に暗黙の型変換があるので、DAC がオブジェクトを（必要に応じて）マーシャリングし、DAC キャッシュ内の `MethodInfo` のホストアドレスを返すことが保証されます。

`GetFromRidMap` の代入文は、配列にインデックスを付けて特定の値を取得します。`pMap` パラメータは `MethodInfo` からの構造体フィールドのアドレスです。そのため、DAC は `MethodInfo` インスタンスをマーシャリングした際にフィールド全体をキャッシュにコピーしています。したがって、この構造体のアドレスである `pMap` はホストポインタです。この参照解除には DAC はまったく関与しません。しかし、`pTable` フィールドは `PTR_TADDR` です。これが示すのは、`pTable` がターゲットアドレスの配列であるが、その型はマーシャリングされた型であることを示しているということです。つまり、`pTable` 自体もターゲットアドレスになります。`PTR` 型のオーバーロードされたインデックス演算子で参照解除します。これにより、配列のターゲットアドレスが取得され、目的の要素のターゲットアドレスが計算されます。インデックスの最後のステップで、配列要素が DAC キャッシュ内のホストインスタンスにマーシャリングされ、その値が返されます。要素（`TADDR`）をローカル変数 `result` に代入して返します。

最後に、コードアドレスを取得するために、DAC/DBI インターフェース関数は `MethodInfo::GetNativeCode` を呼び出します。この関数は `PCODE` 型の値を返します。この型はターゲットアドレスですが、参照解除できないもの（`TADDR` の単なるエイリアス）であり、特にコードアドレスを指定するために使用します。この値を `ICorDebugFunction` インスタンスに格納し、デバッガに返します。

PTR型

DAC はターゲットアドレス空間からホストアドレス空間に値をマーシャリングするため、DAC がターゲットポインタをどのように扱うかを理解することは基本的に重要です。これらのマーシャリングに使用される基本型を総称して「PTR型」と呼びます。`daccess.h` で、いくつかの派生型を持つ `_TPtrBase` と `_GlobalPtr` の2つのクラスが定義されていることがわかります。これらの型を直接使用することはなく、多くのマクロを通じて間接的にのみ使用します。これらのそれぞれには、値のターゲットアドレスを与える単一のデータメンバが含まれています。`_TPtrBase` の場合、これはフルアドレスです。`_GlobalPtr` の場合、DAC グローバルベースロケーションから参照される相対アドレスです。`_TPtrBase` の「T」は「target（ターゲット）」の略です。ご推察のとおり、データメンバまたはローカルであるポインタには `_TPtrBase` から派生した型を使用し、グローバルやスタティックには `_GlobalPtr` を使用します。

実際には、これらの型をマクロを通じてのみ使用します。[daccess.h](#) の冒頭コメントには、これらのすべての使用例があります。これらのマクロについて興味深いのは、DAC ビルドではこれらのマーシャリングテンプレートからインスタンス化された型を宣言するように展開されますが、非 DAC ビルドでは何もしない (no-op) ことです。たとえば、以下の定義はメソッドテーブルポインタを表す型として [PTR_MethodTable](#) を宣言します（規約として、これらの型は [PTR_](#) プレフィックスで命名されることに注意してください）。

```
typedef DPTR(class MethodTable) PTR_MethodTable;
```

DAC ビルドでは、[DPTR](#) マクロは [PTR_MethodTable](#) という名前の [__DPtr<MethodTable>](#) 型を宣言するように展開されます。非 DAC ビルドでは、マクロは単に [PTR_MethodTable](#) を [MethodTable *](#) として宣言します。これは、DAC の機能が非 DAC ビルドで動作の変更やパフォーマンスの低下を引き起こさないことを意味します。

さらに良いことに、DAC ビルドでは、前のセクションの例で見たように、[PTR_MethodTable](#) 型として宣言された変数、データメンバ、または戻り値を DAC が自動的にマーシャリングします。マーシャリングは完全に透過的です。[__DPtr](#) 型にはポインタの参照解除と配列のインデックスを再定義するためのオーバーロードされた演算子関数と、ホストポインタ型にキャストする変換演算子があります。これらの操作は、要求された値がすでにキャッシュにあるかどうかを判断し、キャッシュにある場合はすぐに返し、ターゲットから読み取ってキャッシュにロードしてから返す必要があるかどうかを判断します。詳細を理解することに興味がある場合、これらのキャッシュ操作を担当する関数は [DacInstantiateTypeByAddressHelper](#) です。

[DPTP](#) で定義された [PTR](#) 型はランタイムで最も一般的ですが、グローバルおよびスタティックポインタ、制限付き使用の配列、可変サイズオブジェクトへのポインタ、mscordacwks.dll (msdaccore.dll) から呼び出す必要があるかもしれない仮想関数を持つクラスへのポインタ用の [PTR](#) 型もあります。これらのほとんどはまれであり、必要に応じて [daccess.h](#) を参照して詳しく学ぶことができます。

[GPTP](#) マクロと [VPTP](#) マクロは、ここで特記するに値するほど一般的です。これらの使い方と外部動作は [DPTP](#) と非常に似ています。ここでも、マーシャリングは自動的かつ透過的です。[VPTP](#) マクロは、仮想関数を持つクラスのマーシャリングされたポインタ型を宣言します。この特別なマクロが必要なのは、仮想関数テーブル (vtable) が本質的に暗黙の余分なフィールドだからです。DAC はこれを別途マーシャリングする必要があります。関数アドレスはすべてターゲットアドレスであり、DAC がホストアドレスに変換する必要があります。これらのクラスをこのように扱うことで、DAC は正しい実装クラスを自動的にインスタンス化し、基底型と派生型の間のキャストを不要にします。[VPTP](#) 型を宣言する場合、[vptr_list.h](#) にもリストする必要があります。[__GlobalPtr](#) 型は、[GPTP](#)、[GVAL](#)、[SPTP](#)、[SVAL](#) マクロを通じて、グローバル変数とスタティックデータメンバの両方をマーシャリングする基本機能を提供します。グローバル変数の実装はスタティックフィールドの実装とほぼ同一であり（両方とも [__GlobalPtr](#) クラスを使用）、[dacvars.h](#) にエントリを追加する必要があります。DAC で使用されるグローバル関数は実装サイトでマクロを必要としませんが、アドレスが DAC で自動的に利用可能になるように [gfunc_list.h](#) ヘッダで宣言する必要があります。[daccess.h](#) および [dacvars.h](#) のコメントには、これらの型の宣言に関する詳細が記載されています。

グローバルおよびスタティックの値とポインタは、ターゲットアドレス空間へのエントリーポイントを形成するため興味深いものです（DAC の他のすべての使用では、ターゲットアドレスをすでに持っている必要があります）。ランタイム内のグローバルの多くはすでに DAC 化されています。以前は DAC 化されていなかった（または新しく導入された）グローバルを DAC で利用可能にする必要が時折生じます。適切なマクロと [dacvars.h](#) エントリを使用することで、DAC テーブルの仕組み ([dactable.cpp](#) にあります) がグローバルのアドレスを coreclr.dll からエクスポートされるテーブルに保存できるようにします。DAC はこのテーブルを実行時に使用して、コードがグローバルにアクセスする際にターゲットアドレス空間のどこを見ればよいかを判断します。

VAL 型

ポインタ型に加えて、DAC はスタティックおよびグローバルの値（スタティックまたはグローバルポインタによって参照される値とは対照的に）もマーシャリングする必要があります。このために [?VAL_*](#) マクロのコレクションがあります。グローバル値には [GVAL_*](#)、スタティック値には [SVAL_*](#) を使用します。[daccess.h](#) ファイルのコメントには、これらのさまざまな形式の使用方法を示す表と、DAC 化されたコードで使用するグローバルおよびスタティックの値（およびグローバルおよびスタティックのポインタ）の宣言方法の説明が含まれています。

純粋アドレス (Pure Addresses)

DAC の操作の例で紹介した `TADDR` 型と `PCODE` 型は、純粋なターゲットアドレスです。これらは実際にはポインタではなく、整数型です。これにより、ホスト内のコードが誤ってそれらを参照解除することを防ぎます。DAC もこれらをポインタとして扱いません。具体的には、型やサイズの情報がないため、参照解除やマーシャリングを行うことができません。これらを主に2つの状況で使用します。ターゲットアドレスを純粋なデータとして扱う場合と、ターゲットアドレスでポインタ演算を行う必要がある場合です（`PTR` 型でもポインタ演算は可能です）。もちろん、`TADDR` が指定するターゲットロケーションに型情報がないため、アドレス演算を行う場合はサイズを明示的に考慮する必要があります。

マーシャリングを伴わない特別なクラスの `PTR` もあります。`PTR_VOID` と `PTR_CVOID` です。これらはそれぞれ `void *` と `const void *` のターゲット等価物です。`TADDR` は単なる数値であるため、ポインタセマンティクスを持ちません。つまり、`void *` を `TADDR` に変換して DAC 化する場合（過去にはよく行われていました）、DAC 用にコンパイルされないコードであっても、追加のキャストやその他の変更が必要になることがあります。`PTR_VOID` を使用すると、`void *` に期待されるセマンティクスを保持したまま、`void *` を使用的コードをより簡単かつクリーンに DAC 化できます。`PTR_VOID` や `PTR_CVOID` を使用的関数を DAC 化する場合、どれだけのデータを読む必要があるかわからないため、これらのアドレスからデータを直接マーシャリングすることはできません。つまり、参照解除はできません（ポインタ演算もできません）が、これは `void *` のセマンティクスと同一です。`void *` の場合と同様に、使用する必要がある場合は通常、より具体的な `PTR` 型にキャストします。`PTR_BYTE` 型もあり、これは標準的なマーシャリングされたターゲットポインタです（ポインタ演算などをサポートします）。一般的に、コードを DAC 化する際は、期待どおり `void *` は `PTR_VOID` に、`BYTE *` は `PTR_BYTE` になります。`daccess.h` には、`PTR_VOID` 型の使用とセマンティクスに関する詳細なコメントがあります。

レガシーコードが `void *` などのホストポインタ型にターゲットアドレスを格納していることがあります。これは常にバグであり、コードの推論を非常に困難にします。クロスプラットフォーム対応（ポインタ型のサイズが異なる場合）でも壊れます。DAC ビルドでは、`void *` 型はターゲットアドレスを格納すべきではないホストポインタです。代わりに `PTR_VOID` を使用すると、void ポインタ型がターゲットアドレスであることを示すことができます。このような使用をすべて排除しようとしていますが、一部はコード内に広く浸透しており、完全に排除するにはしばらくかかります。

変換 (Conversions)

以前の CLR バージョンでは、型間のキャストに C スタイルの型キャスト、マクロ、およびコンストラクタを使用していました。たとえば、`MethodIterator::Next` では以下のようにになっていました。

```
if (methodCold)
{
    PTR_CORCOMPILE_METHOD_COLD_HEADER methodColdHeader
    = PTR_CORCOMPILE_METHOD_COLD_HEADER((TADDR)methodCold);

    if (((TADDR)methodCode) == PTR_TO_TADDR(methodColdHeader->hotHeader))
    {
        // コールドコードに一致
        m_pCMH = PTR_CORCOMPILE_METHOD_COLD_HEADER((TADDR)methodCold);
        ...
    }
}
```

`methodCold` と `methodCode` はどちらも `BYTE *` として宣言されていますが、実際にはターゲットアドレスを保持しています。4行目で、`methodCold` は `TADDR` にキャストされ、`PTR_CORCOMPILE_METHOD_COLD_HEADER` のコンストラクタへの引数として使用されます。この時点で、`methodColdHeader` は明示的にターゲットアドレスです。6行目では、`methodCode` の別の C スタイルのキャストがあります。`methodColdHeader` の `hotHeader` フィールドは `PTR_CORCOMPILE_METHOD_HEADER` 型です。`PTR_TO_TADDR` マクロは、この `PTR` 型から生のターゲットアドレスを抽出して `methodCode` に代入します。最後に9行目で、

`PTR_CORCOMPILE_METHOD_COLD_HEADER` 型の別のインスタンスが構築されます。ここでも、`methodCold` はこのコンストラクタに渡すために `TADDR` にキャストされています。

このコードが過度に複雑で混乱するように見えるなら、それは良いことです。実際にそうなのです。さらに悪いことに、ホストとターゲットのアドレスの分離を保護する機能を提供しません。`methodCold` と `methodCode` の宣言からは、これらをターゲットアドレスとして解釈する特別な理由はありません。これらのポインタが DAC ビルドで本当にホストポインタであるかのように参照解除された場合、プロセスはおそらくアクセス違反 (AV) を起こすでしょう。このスニペットは、任意のポインタ型 (`PTR` 型とは対照的に) が `TADDR` にキャストできることを示しています。これら2つの変数が常にターゲットアドレスを保持すると考えると、`BYTE *` ではなく `PTR_BYTE` 型であるべきです。

異なる `PTR` 型間でキャストするための規律ある手段もあります。`dac_cast` です。`dac_cast` 演算子は、C++ の `static_cast` 演算子の DAC 対応バージョンです (CLR コーディング規約では、ポインタ型のキャスト時に C スタイルのキャストの代わりに `static_cast` を使用することが規定されています)。`dac_cast` 演算子は、以下のいずれかを行います。

1. `TADDR` から `PTR` 型を作成する
2. ある `PTR` 型を別の `PTR` 型に変換する
3. 以前に DAC キャッシュにマーシャリングされたホストインスタンスから `PTR` を作成する
4. `PTR` 型から `TADDR` を抽出する
5. 以前に DAC キャッシュにマーシャリングされたホストインスタンスから `TADDR` を取得する

💡 初心者向け補足

`dac_cast` は C++ の `static_cast` のような型変換演算子ですが、DAC のホスト／ターゲットアドレスの区別を意識した安全なキャストです。たとえば、`dac_cast<PTR_MethodDesc>(someAddress)` と書くと、ターゲットアドレス `someAddress` にある `MethodDesc` を安全にマーシャリングして使えるようにしてくれます。C スタイルのキャスト `(MethodDesc*)ptr` とは違い、`dac_cast` はアドレス空間の混同を検出してコンパイルエラー・ランタイムエラーを出してくれるため、バグの早期発見に役立ちます。

さて、`methodCold` と `methodCode` の両方が `PTR_BYTE` 型として宣言されていると仮定すると、上記のコードは以下のように書き直すことができます。

```
if (methodCold)
{
    PTR_CORCOMPILE_METHOD_COLD_HEADER methodColdHeader
        = dac_cast<PTR_CORCOMPILE_METHOD_COLD_HEADER>(methodCold);

    if (methodCode == methodColdHeader->hotHeader)
    {
        // コールドコードに一致
        m_pCMH = methodColdHeader;
```

このコードはまだ複雑で混乱するように見えるかもしれません、少なくともキャストとコンストラクタの数を大幅に削減しました。また、ホストポインタとターゲットポインタの分離を維持する構造を使用しているため、コードをより安全にしました。特に、`dac_cast` は間違ったことをしようとすると、コンパイラエラーまたはランタイムエラーを生成することがよくあります。一般的に、変換には `dac_cast` を使用するべきです。

DAC 化 (DACizing)

いつ DAC 化が必要か？

新しい機能を追加する場合はいつでも、そのデバッグ可能性のニーズを考慮し、機能をサポートするためにコードを DAC 化する必要があります。また、バグ修正やコードのクリーンアップなどの他の変更が、必要に応じて DAC のルールに準拠していることを確認する必要があります。そうでないと、変更がデバッガや SOS を壊すことになります。(新しい機能を実装するのではなく) 既存のコードを単に変更する場合、変更する関数に `SUPPORTS_DAC` コントラクト (contract) が含まれていれば、DAC について心配する必要があると一般的に判断できます。このコントラクトには `SUPPORTS_DAC_WRAPPER` や `LEAF_DAC_CONTRACT` などのいくつかのバリエーションがあります。`contract.h` にはこれらの違いを説明するコメントがあります。関数内に多くの DAC 固有の型が見つかる場合は、そのコードが DAC ビルドで実行されると仮定すべきです。

DAC 化は、エンジン内のコードが DAC と正しく動作することを保証します。ターゲットからホストに値を正しくマーシャリングするために DAC を正しく使用することが重要です。ホストから不正に使用されたターゲットアドレス（またはその逆）は、マップされていないアドレスを参照する可能性があります。アドレスがマップされている場合、値は期待される値とはまったく無関係になります。その結果、DAC 化は主に、DAC がマーシャリングする必要があるすべての値に `PTR` 型を使用することを保証することになります。もう1つの主要なタスクは、DAC ビルドで侵襲的なコードの実行を許可しないことを保証することです。実際には、コードのリファクタリングや `DACCESS_COMPILE` プリプロセッサディレクティブの追加が必要になることがあります。また、適切な `SUPPORTS_DAC` コントラクトを追加することも重要です。このコントラクトの使用は、関数が DAC と連携して動作することを開発者に示します。これが重要な理由は2つあります。

1. 後で別の `SUPPORTS_DAC` 関数からこの関数を呼び出す場合、それが DAC セーフであることがわかっているため、DAC について心配する必要がありません。
2. 関数に変更を加える場合、それらが DAC セーフであることを確認する必要があります。この関数から別の関数への呼び出しを追加する場合、それが DAC セーフであること、または非 DAC ビルドでのみ呼び出されることを確認する必要があります。

プロファイリング

原文

この章の原文は [Profiling](#) です。

はじめに

プロファイリングとは、共通言語ランタイム (CLR) 上で実行されるプログラムの実行を監視することを意味します。この章では、ランタイムが提供するプロファイリング情報にアクセスするためのインターフェースについて詳しく説明します。

「プロファイリング API」と呼ばれていますが、この機能は従来のプロファイリングツールだけでなく、コードカバレッジユーティリティや高度なデバッグ支援ツールなど、より広範な診断ツールクラスに適しています。従来のプロファイリングツールは、プログラムの実行を計測すること、つまり各関数に費やされた時間やプログラムの経時的なメモリ使用量に焦点を当てています。しかし、プロファイリング API は実際にはより広範な診断ツールを対象としています。

これらすべての用途に共通しているのは、すべてが診断的な性質を持っているということです。つまり、ツールはプログラムの実行を監視するために書かれます。プロファイリング API はプログラム自身によって使用されるべきではなく、プログラムの実行の正確性はプロファイラーがアクティブであることに依存したり、影響を受けたりしてはなりません。

初心者向け補足

プロファイリングとは、プログラムの実行中にパフォーマンスのボトルネック（どの関数に時間がかかっているか、メモリがどのように使われているか）を特定するための技術です。Visual Studio の「診断ツール」や dotnet-trace などのツールが、この API を利用しています。Java では JFR (Java Flight Recorder) や VisualVM が類似の役割を果たします。

CLR プログラムのプロファイリングには、従来のコンパイル済みマシンコードのプロファイリングよりも多くのサポートが必要です。これは、CLR にはアプリケーションドメイン (Application Domain)、ガベージコレクション (Garbage Collection)、マネージド例外処理 (Managed Exception Handling)、JIT コンパイル (中間言語 (Intermediate Language) からネイティブマシンコードへの変換) といった概念があり、既存の従来のプロファイリングメカニズムではこれらを識別して有用な情報を提供できないためです。プロファイリング API は、CLR およびプロファイル対象プログラムのパフォーマンスへの影響を最小限に抑える効率的な方法で、この欠落している情報を提供します。

実行時にルーチンを JIT コンパイルすることは良い機会を提供します。API を使用すると、プロファイラーはルーチンのメモリ内 IL コードストリームを変更し、新たに JIT コンパイルし直すことを要求できます。このようにして、プロファイラーはより深い調査が必要な特定のルーチンに対して、動的にインストルメンテーション (Instrumentation) コードを追加できます。このアプローチは従来のシナリオでも可能ですが、CLR ではこれがはるかに容易です。

プロファイリング API の目標

- CLR 上で実行されるプログラムのパフォーマンスを判断・分析するために、既存のプロファイラーが必要とする情報を公開する。具体的には：

- 共通言語ランタイムの起動・シャットダウンイベント
- アプリケーションドメインの作成・シャットダウンイベント
- アセンブリのロード・アンロードイベント
- モジュールのロード・アンロードイベント
- COM VTable の作成・破棄イベント
- JIT コンパイル・コードピッ칭 (code pitching) イベント
- クラスのロード・アンロードイベント
- スレッドの生成・消滅・同期
- 関数の開始 (Entry)・終了 (Exit) イベント
- 例外
- マネージド実行とアンマネージド実行間の遷移
- 異なるランタイム コンテキスト間の遷移
- ランタイムサスペンション (suspension) についての情報
- ランタイムメモリヒープおよびガベージコレクション活動についての情報
- 任意の（非マネージド）COM 互換言語から呼び出し可能であること
- CPU とメモリ消費の面で効率的であること — プロファイリングの行為がプロファイル対象プログラムに大きな変化を与え、結果が誤解を招くようあってはならない
- サンプリング (*sampling*) プロファイラーと 非サンプリング (*non-sampling*) プロファイラーの両方に有用であること。（サンプリング プロファイラーは、一定のクロックティック — 例えば 5 ミリ秒間隔 — でプロファイル対象を検査する。非サンプリング プロファイラーは、イベントを引き起こしたスレッドと同期的にイベントの通知を受ける。）

初心者向け補足

サンプリングプロファイラーは一定間隔（例：5ミリ秒ごと）でプログラムの状態を調べます。オーバーヘッドが小さい反面、短時間の処理を見逃す可能性があります。非サンプリングプロファイラーはイベント発生時に同期的に通知を受けます。正確ですがオーバーヘッドが大きくなります。

プロファイリング API の非目標

- プロファイリング API はアンマネージドコードのプロファイリングをサポートしません。アンマネージドコードのプロファイリングには既存のメカニズムを使用する必要があります。CLR プロファイリング API はマネージドコードに対してのみ動作します。ただし、プロファイラーはマネージド/アンマネージド遷移イベントを提供して、マネージドコードとアンマネージドコード間の境界を判断できるようにします。
- プロファイリング API は、アスペクト指向プログラミング (Aspect-Oriented Programming) などの目的で自身のコードを変更するアプリケーションの作成をサポートしません。
- プロファイリング API は、境界チェック (bounds checking) に必要な情報を提供しません。CLR はすべてのマネージドコードに対する境界チェックの組み込みサポートを提供しています。

CLR コードプロファイラーインターフェースは、以下の理由からリモートプロファイリングをサポートしません：

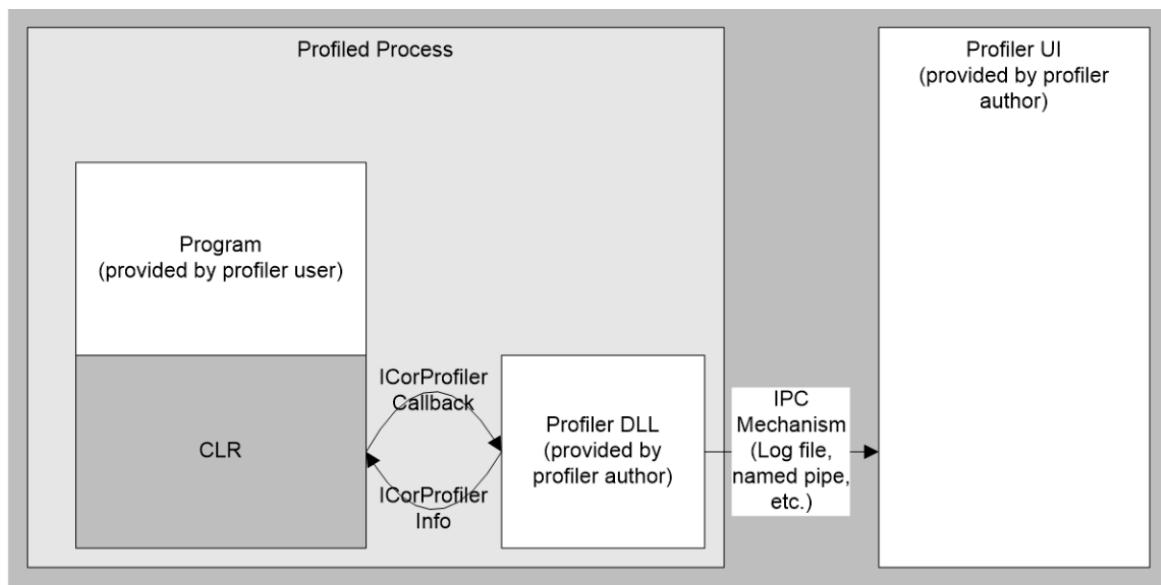
- プロファイリング結果が過度に影響を受けないように、これらのインターフェースを使用する実行時間を最小化する必要があります。これは特に実行パフォーマンスを監視する場合に重要です。ただし、メモリ使用量の監視やスタックフレーム・オブジェクトなどに関するランタイム情報を取得する目的でインターフェースが使用される場合は制約となりません。
- コードプロファイラーは、プロファイル対象アプリケーションが実行されるローカルマシン上のランタイムに、1つ以上のコールバック(callback) インターフェースを登録する必要があります。これにより、リモートコードプロファイラーの作成能力が制限されます。

プロファイリング API-概要

CLR 内のプロファイリング API を使用すると、実行中のアプリケーションの実行状況とメモリ使用量を監視できます。通常、この API はコードプロファイラーパッケージの作成に使用されます。以降のセクションでは、プロファイラーを あらゆる マネージドアプリケーションの実行を監視するために構築されたパッケージとして説明します。

プロファイリング API は、プロファイル対象プログラムと同じプロセスにロードされるプロファイラー DLL によって使用されます。プロファイラー DLL はコールバックインターフェース ([ICorProfilerCallback2](#)) を実装します。ランタイムはそのインターフェースのメソッドを呼び出して、プロファイル対象プロセス内のイベントをプロファイラーに通知します。プロファイラーは [ICorProfilerInfo](#) のメソッドを使用してランタイムにコールバックし、プロファイル対象アプリケーションの状態に関する情報を取得できます。

プロファイラーソリューションのデータ収集部分のみがプロファイル対象アプリケーションとインプロセスで実行されるべきであることに注意してください。UI とデータ分析は別のプロセスで行うべきです。



[ICorProfilerCallback](#) および [ICorProfilerCallback2](#) インターフェースは、[ClassLoadStarted](#)、[ClassLoadFinished](#)、[JITCompilationStarted](#) のような名前のメソッドで構成されています。CLR がクラスのロード/アンロード、関数のコンパイルなどをを行うたびに、プロファイラーの [ICorProfilerCallback](#) / [ICorProfilerCallback2](#) インターフェースの対応するメソッドを呼び出します（他のすべての通知についても同様です。詳細は後述します）。

例えば、プロファイラーは [FunctionEnter](#) と [FunctionLeave](#) の 2 つの通知を通じてコードパフォーマンスを測定できます。各通知にタイムスタンプを記録し、結果を蓄積してから、アプリケーションの実行中に最も CPU 時間やウォールクロック時間 (wall-clock time) を消費した関数のリストを出力するだけです。

[ICorProfilerCallback](#) / [ICorProfilerCallback2](#) インターフェースは「通知 (notifications) API」と考えることができます。

プロファイリングに関するもう一つのインターフェースは `ICorProfilerInfo` です。プロファイラーは必要に応じてこれを呼び出し、分析を支援するための追加情報を取得します。例えば、CLR が `FunctionEnter` を呼び出すたびに `FunctionId` の値を提供します。プロファイラーは `ICorProfilerInfo::GetFunctionInfo` を呼び出すことで、その `FunctionId` についての詳細情報（関数の親クラス、名前など）を見つけることができます。

💡 初心者向け補足

`ICorProfilerCallback` と `ICorProfilerInfo` の関係は、オブザーバーパターン (Observer Pattern) に似ています。

`ICorProfilerCallback` はランタイムからプロファイラーへの「プッシュ型」通知で、`ICorProfilerInfo` はプロファイラーからランタイムへの「プル型」問い合わせです。Java の JVMTI (JVM Tool Interface) にも類似の設計があります。

ここまで説明は、アプリケーションとプロファイラーが実行されているときに何が起こるかを記述しています。しかし、アプリケーションの開始時にこの 2 つはどのように接続されるのでしょうか？CLR は各プロセスの初期化時に接続を行います。プロファイラーに接続するかどうか、およびどのプロファイラーに接続するかは、以下の 2 つの環境変数の値に基づいて順番にチェックされます：

- `CORECLR_ENABLE_PROFILING` – この環境変数が存在し、非ゼロ値に設定されている場合にのみプロファイラーに接続する。
- `CORECLR_PROFILER` – この CLSID または ProgID のプロファイラーに接続する（事前にレジストリに登録されている必要があります）。`CORECLR_PROFILER` 環境変数は文字列として定義される：
 - `set CORECLR_PROFILER={32E2F4DA-1BEA-47ea-88F9-C5DAF691C94A}` または
 - `set CORECLR_PROFILER="MyProfiler"`
- プロファイラークラスは `ICorProfilerCallback` / `ICorProfilerCallback2` を実装するクラスです。プロファイラーは `ICorProfilerCallback2` を実装することが必須であり、実装していない場合はロードされません。

上記の両方のチェックが通過すると、CLR は `CoCreateInstance` と同様の方法でプロファイラーのインスタンスを作成します。プロファイラーは `CoCreateInstance` の直接呼び出しを通じてロードされるのではなく、スレッディングモデルの設定を必要とする `CoInitialize` の呼び出しを回避するためです。その後、プロファイラーの `ICorProfilerCallback::Initialize` メソッドを呼び出します。このメソッドのシグネチャは以下のとおりです：

```
HRESULT Initialize(IUnknown *pICorProfilerInfoUnk)
```

cpp

プロファイラーは `pICorProfilerInfoUnk` に対して `QueryInterface` を行い、`ICorProfilerInfo` インターフェースポインタを取得して保存し、後でプロファイリング中に追加情報を呼び出せるようにする必要があります。その後、`ICorProfilerInfo::SetEventMask` を呼び出して、関心のある通知カテゴリを指定します。例えば：

```
ICorProfilerInfo* pInfo;  
  
pICorProfilerInfoUnk->QueryInterface(IID_ICorProfilerInfo, (void**)&pInfo);  
  
pInfo->SetEventMask(COR_PRF_MONITOR_ENTERLEAVE | COR_PRF_MONITOR_GC)
```

cpp

このマスクは、関数の Enter/Leave 通知とガベージコレクション通知にのみ関心のあるプロファイラーで使用されます。プロファイラーは単に戻り値を返すだけで、実行が開始されます！

このように通知マスクを設定することで、プロファイラーは受信する通知を制限できます。これは明らかに、よりシンプルな、または特殊な目的のプロファイラーの構築を容易にします。また、プロファイラーが単に「捨てる」だけの通知を送信するための無駄な CPU 時間を削減します（詳細は後述）。

1つの環境内では、一度に1つのプロファイラーのみがプロセスをプロファイリングできることに注意してください。異なる環境では、各環境に2つの異なるプロファイラーを登録し、それぞれが別のプロセスをプロファイリングすることができます。

特定のプロファイラーイベントは不变 (**IMMUTABLE**) です。つまり、`ICorProfilerCallback::Initialize` コールバックで設定された後は、`ICorProfilerInfo::SetEventMask()` を使用してオフにすることはできません。不变イベントを変更しようとすると、`SetEventMask` は失敗した HRESULT を返します。

プロファイラーはインプロセス COM サーバー (inproc COM server)、つまりプロファイル対象プロセスと同じアドレス空間にマッピングされる DLL として実装する必要があります。他の種類の COM サーバーはサポートされていません。プロファイラーが例えばリモートコンピューターからアプリケーションを監視したい場合は、各マシンに「コレクタエージェント (collector agent)」を実装し、結果をバッチ処理して中央データ収集マシンに送信する必要があります。

プロファイリング API – 繰り返し登場する概念

このセクションでは、各メソッドの説明で繰り返し述べる代わりに、プロファイリング API 全体を通じて適用されるいくつかの概念を簡潔に説明します。

ID

ランタイム通知は、報告されたクラス、スレッド、AppDomain などの ID を提供します。これらの ID はランタイムに対して追加の情報を問い合わせるために使用できます。これらの ID は単にその項目を記述するメモリ内のブロックのアドレスですが、どのプロファイラーでも不透明なハンドル (opaque handle) として扱うべきです。無効な ID がプロファイリング API 関数への呼び出しで使用された場合、結果は未定義です。最も可能性が高いのは、アクセス違反 (access violation) になることです。使用される ID が完全に有効であることを保証する責任はユーザーにあります。プロファイリング API はいかなる種類の検証も実行しません。それはオーバーヘッドを生じさせ、実行速度を著しく低下させるためです。

一意性 (Uniqueness)

`ProcessID` はプロセスの存続期間中、システム全体で一意です。他のすべての ID はその ID の存続期間中、プロセス全体で一意です。

階層と包含関係 (Hierarchy & Containment)

ID はプロセス内の階層を反映した階層構造に配置されています。プロセスは AppDomain を含み、AppDomain はアセンブリを含み、アセンブリはモジュールを含み、モジュールはクラスを含み、クラスは関数を含みます。スレッドはプロセス内に含まれ、AppDomain 間を移動できます。オブジェクトはほとんどが AppDomain 内に含まれます（ごく少数のオブジェクトは同時に複数の AppDomain のメンバーである場合があります）。コンテキスト (Context) はプロセス内に含まれます。

存続期間と安定性 (Lifetime & Stability)

ある ID が消滅すると、その中に含まれるすべての ID も消滅します。

- `ProcessID` – `Initialize` の呼び出しから `Shutdown` の戻り値まで有効かつ安定。
- `AppDomainID` – `AppDomainCreationFinished` の呼び出しから `AppDomainShutdownStarted` の戻り値まで有効かつ安定。
- `AssemblyID`, `ModuleID`, `ClassID` – その ID に対する `LoadFinished` の呼び出しから `UnloadStarted` の戻り値まで有効かつ安定。
- `FunctionID` – `JITCompilationFinished` または `JITCachedFunctionSearchFinished` の呼び出しから、含まれる `ClassID` の消滅まで有効かつ安定。

- **ThreadId** – ThreadCreated の呼び出しから ThreadDestroyed の戻り値まで有効かつ安定。
- **ObjectID** – ObjectAllocated の呼び出し以降有効。各ガベージコレクションで変化または消滅する可能性がある。
- **GCHandleID** – HandleCreated の呼び出しから HandleDestroyed の戻り値まで有効。

加えて、プロファイリング API 関数から返された任意の ID は、返された時点で有効です。

AppDomain アフィニティ (App-Domain Affinity)

プロセス内の各ユーザー作成 AppDomain に対して AppDomainID があり、さらに「デフォルト」ドメインと、ドメインニュートラル (domain-neutral) なアセンブリを保持するための特別な擬似ドメインがあります。

Assembly、Module、Class、Function、および GCHandleID は AppDomain アフィニティを持ちます。つまり、アセンブリが複数の AppDomain にロードされた場合、そのアセンブリ（およびその中に含まれるすべてのモジュール、クラス、関数）はそれぞれ異なる ID を持ち、各 ID に対する操作は関連付けられた AppDomain 内でのみ有効です。ドメインニュートラルなアセンブリは、上記で述べた特別な擬似ドメインに表示されます。

特記事項 (Special Notes)

ObjectID を除くすべての ID は不透明な値として扱うべきです。ほとんどの ID はかなり自明ですが、いくつかはより詳しく説明する価値があります。

ClassID はクラスを表します。ジェネリッククラスの場合、完全にインスタンス化された型を表します。`List<int>`、`List<char>`、`List<object>`、`List<string>` はそれぞれ固有の ClassID を持ちます。`List<T>` はインスタンス化されていない型であり、ClassID を持ちません。`Dictionary<string, V>` は部分的にインスタンス化された型であり、ClassID を持ちません。

FunctionID は関数のネイティブコードを表します。ジェネリック関数（またはジェネリッククラスの関数）の場合、特定の関数に対して複数のネイティブコードのインスタンス化 (instantiation) が存在する可能性があり、したがって複数の FunctionID が存在する可能性があります。ネイティブコードのインスタンス化は異なる型間で共有される場合があります。例えば、`List<string>` と `List<object>` はすべてのコードを共有するため、FunctionID は複数の ClassID に「属する」場合があります。

ObjectID はガベージコレクション対象オブジェクトを表します。ObjectID は、プロファイラーが ObjectID を受信した時点でのオブジェクトの現在のアドレスであり、各ガベージコレクションで変化する可能性があります。したがって、ObjectID の値は受信した時点から次のガベージコレクションが開始されるまでの間のみ有効です。CLR はプロファイラーがオブジェクトを追跡する内部マップを更新できる通知も提供するため、プロファイラーはガベージコレクション間で有効な ObjectID を維持できます。

GCHandleID は GC のハンドルテーブル (handle table) 内のエントリを表します。GCHandleID は ObjectID とは異なり、不透明な値です。GC ハンドルは一部の状況ではランタイム自体によって作成されるか、ユーザーコードで `System.Runtime.InteropServices.GCHandle` 構造体を使用して作成できます (GCHandle 構造体はハンドルを表すだけであり、ハンドルは GCHandle 構造体内に「存在」するわけではないことに注意してください)。

ThreadId はマネージドスレッドを表します。ホストがファイバーモード (fiber mode) での実行をサポートしている場合、マネージドスレッドは検査する時点に応じて異なる OS スレッド上に存在する可能性があります（注：ファイバーモードアプリケーションのプロファイリングはサポートされていません）。

コールバックの戻り値 (Callback Return Values)

プロファイラーは、CLR がトリガーする各通知に対して HRESULT でステータスを返します。そのステータスは `S_OK` または `E_FAIL` の値を持つことができます。現在、ランタイムは `ObjectReferences` を除くすべてのコールバックでこのステータス値を無視します。

呼び出し元確保バッファ (Caller-Allocated Buffers)

呼び出し元確保バッファを取る `ICorProfilerInfo` 関数は、通常以下のシグネチャに準拠します：

```
HRESULT GetBuffer( [in] /* Some query information */,
    [in] ULONG32 cBuffer,
    [out] ULONG32 *pcBuffer,
    [out, size_is(cBuffer), length_is(*pcMap)] /* TYPE */ buffer[] );
```

これらの関数は常に以下のように動作します：

- `cBuffer` はバッファに割り当てられた要素数です。
- `*pcBuffer` は利用可能な要素の総数に設定されます。
- `buffer` は可能な限り多くの要素で埋められます。

要素が返された場合、戻り値は `S_OK` になります。バッファが十分な大きさであったかどうかを確認するのは呼び出し元の責任です。

`buffer` が NULL の場合、`cBuffer` は 0 でなければなりません。関数は `S_OK` を返し、`*pcBuffer` を利用可能な要素の総数に設定します。

オプションの出力パラメータ (Optional Out Parameters)

関数が 1 つの `[out]` パラメータしか持たない場合を除き、API のすべての `[out]` パラメータはオプションです。プロファイラーは関心のない `[out]` パラメータに対して単に NULL を渡します。プロファイラーは関連する `[in]` パラメータにも一貫した値を渡す必要があります。例えば、NULL の `[out]` パラメータがデータで埋められるバッファの場合、そのサイズを指定する `[in]` パラメータは 0 でなければなりません。

通知スレッド (Notification Thread)

ほとんどの場合、通知はイベントを生成したスレッドと同じスレッドで実行されます。そのような通知（例えば `FunctionEnter` や `FunctionLeave`）は明示的な ThreadID を提供する必要がありません。また、プロファイラーは、影響を受けるスレッドの ThreadID に基づくグローバルストレージへのインデックス付けと比較して、スレッドローカルストレージ (Thread-Local Storage, TLS) を使用して分析ブロックの保存と更新を行うことを選択できます。

各通知は、どのスレッドが呼び出しを行ったかを文書化しています。イベントを生成したスレッドか、ランタイム内の何らかのユーティリティスレッド（例：ガベージコレクタ）のいずれかです。異なるスレッドによって呼び出される可能性があるコールバックについては、ユーザーは `ICorProfilerInfo::GetCurrentThreadId` を呼び出して、イベントを生成したスレッドを特定できます。

これらのコールバックはシリアル化されないことに注意してください。プロファイラー開発者は、スレッドセーフなデータ構造を作成し、複数のスレッドからの並行アクセスを防ぐために必要に応じてプロファイラーコードをロックすることで、防御的なコードを書く必要があります。したがって、特定のケースでは異常なコールバックシーケンスを受信する可能性があります。例えば、マネージドアプリケーションが同一のコードを実行する 2 つのスレッドを生成しているとします。この場合、あるスレッドからある関数の

`JITCompilationStarted` イベントを受信し、対応する `JITCompilationFinished` コールバックを受信する前に、もう一方のスレッドがすでに `FunctionEnter` コールバックを送信している可能性があります。したがって、まだ完全に JIT コンパイルされていないよう見える関数の `FunctionEnter` コールバックを受信することになります！

 初心者向け補足

プロファイラーを開発する際は、複数のスレッドから同時にコールバックが呼ばれる可能性があることに注意が必要です。例えば、あるスレッドで JIT コンパイルが完了する前に、別のスレッドから関数実行の通知が届くことがあります。これはバグではなく、マルチスレッド環境での正常な動作です。

GC セーフコールアウト (GC-Safe Callouts)

CLR が `ICorProfilerCallback` の特定の関数を呼び出すとき、プロファイラーがその呼び出しから制御を返すまで、ランタイムはガベージコレクションを実行できません。これは、プロファイリングサービスがガベージコレクションに安全な状態にスタックを構築できない場合があるためです。代わりに、そのコールバックの周囲でガベージコレクションが無効化されます。これらのケースでは、プロファイラーはできるだけ早く制御を返すように注意する必要があります。これが適用されるコールバックは以下のとおりです：

- `FunctionEnter`, `FunctionLeave`, `FunctionTailCall`
- `ExceptionOSHandlerEnter`, `ExceptionOSHandlerLeave`
- `ExceptionUnwindFunctionEnter`, `ExceptionUnwindFunctionLeave`
- `ExceptionUnwindFinallyEnter`, `ExceptionUnwindFinallyLeave`
- `ExceptionCatcherEnter`, `ExceptionCatcherLeave`
- `ExceptionCLRCatcherFound`, `ExceptionCLRCatcherExecute`
- `COMClassicVTableCreated`, `COMClassicVTableDestroyed`

さらに、以下のコールバックはプロファイラーのブロックを許可する場合とそうでない場合があります。これは `fIsSafeToBlock` 引数を通じて呼び出しごとに示されます。このセットには以下が含まれます：

- `JITCompilationStarted`, `JITCompilationFinished`

プロファイラーがブロックする場合、ガベージコレクションが遅延されることに注意してください。これは、プロファイラーコード自体がマネージドヒープにスペースを割り当てようとしない限り無害です。マネージドヒープへの割り当てはデッドロック (deadlock) を引き起こす可能性があります。

💡 初心者向け補足

GC セーフコールアウトは、プロファイラーがランタイムのガベージコレクションとデッドロックしないための設計上の制約です。プロファイラーのコールバック内でマネージドヒープにメモリを割り当てようとすると、GC が必要 → しかし GC はコールバック完了まで待機 → デッドロック、という状況に陥る可能性があります。

COM の使用 (Using COM)

プロファイリング API インターフェースは COM インターフェースとして定義されていますが、ランタイムは実際にはそれらを使用するために COM を初期化しません。これは、マネージドアプリケーションが望ましいスレッディングモデルを指定する機会を得る前に、`CoInitialize` を介してスレッディングモデルを設定する必要を避けるためです。同様に、プロファイラー自体も `CoInitialize` を呼び出すべきではありません。プロファイル対象アプリケーションと互換性のないスレッディングモデルを選択し、アプリケーションを壊す可能性があるためです。

コールバックとスタック深度 (Callbacks and Stack Depth)

プロファイラーコールバックは、スタックが極めて制約された状況で発行される場合があり、プロファイラーコールバック内のスタックオーバーフロー (stack overflow) は即座のプロセス終了につながります。プロファイラーはコールバックへのレスポンスで可能な限り少ないスタックを使用するよう注意すべきです。プロファイラーがスタックオーバーフローに対して堅牢なプロセスに対して使用されることを想定している場合、プロファイラー自体もスタックオーバーフローのトリガーを避けるべきです。

NT サービスのプロファイリング方法 (How to profile a NT Service)

プロファイリングは環境変数を通じて有効化され、NT サービスはオペレーティングシステムのブート時に起動されるため、これらの環境変数はその時点で存在し、必要な値に設定されている必要があります。したがって、NT サービスのプロファイリングを行うには、適切な環境変数を事前にシステム全体で設定する必要があります：

マイコンピュータ → プロパティ → 詳細設定 → 環境変数 → システム環境変数

`CORECLR_ENABLE_PROFILING` と `CORECLR_PROFILER` の両方を設定する必要があります、プロファイラー DLL が登録されていることをユーザーが確認する必要があります。その後、NT サービスがこれらの変更を取得するために、ターゲットマシンを再起動する必要があります。これによりシステム全体でのプロファイリングが有効になることに注意してください。そのため、以降に実行されるすべてのマネージドアプリケーションがプロファイルされることを防ぐために、再起動後にそれらのシステム環境変数を削除する必要があります。

プロファイリング API- 高レベルの説明

ローダーコールバック (Loader Callbacks)

ローダーコールバックは、AppDomain、アセンブリ、モジュール、およびクラスのロードに対して発行されるコールバックです。

CLR がアセンブリのロードを通知し、その後にそのアセンブリに対する 1 つ以上のモジュールのロードが続くと期待するかもしれません。しかし、実際に何が起こるかはローダーの実装内のいくつかの要因に依存します。プロファイラーが依存できるのは以下の点です：

- 同じ ID に対して、Started コールバックは Finished コールバックよりも先に配信される。
- Started と Finished コールバックは同じスレッド上で配信される。

ローダーコールバックは Started/Finished ペアで構成されていますが、ローダー内部の操作に正確に時間を帰属させるために使用することはできません。

コールスタック (Call stacks)

プロファイリング API は、コールスタック (call stack) を取得するための 2 つの方法を提供します。コールスタックをまばらに収集するのに適したスナップショットメソッド (snapshot method) と、あらゆる瞬間にコールスタックを追跡するのに適したシャドウスタックメソッド (shadow-stack method) です。

スタックスナップショット (Stack Snapshot)

スタックスナップショットは、ある瞬間ににおけるスレッドのスタックのトレースです。プロファイリング API はスタック上のマネージド関数のトレースをサポートしていますが、アンマネージド関数のトレースはプロファイラー自身のスタックウォーカー (stack walker) に委ねています。

シャドウスタック (Shadow Stack)

上記のスナップショットメソッドを頻繁に使用すると、すぐにパフォーマンスの問題になる可能性があります。スタックトレースを頻繁に取得する必要がある場合、プロファイラーは代わりに `FunctionEnter`、`FunctionLeave`、`FunctionTailCall`、および `Exception*` コールバックを使用して「シャドウスタック (shadow stack)」を構築すべきです。シャドウスタックは常に最新であり、スタックスナップショットが必要な時にすぐにストレージにコピーできます。

シャドウスタックは、関数の引数、戻り値、およびジェネリックインスタンス化 (generic instantiation) に関する情報を取得できます。この情報はシャドウスタックを通じてのみ利用可能です。なぜなら、関数のエントリ時には容易に利用できますが、関数の実行の後半では最適化によって除去されている可能性があるためです。

💡 初心者向け補足

シャドウスタックは、プロファイラーが独自に管理する仮想的なコールスタックです。関数の Enter/Leave イベントごとにスタックを自前で管理することで、いつでも現在のコールスタックを高速に参照できます。毎回スレッドのスタックを走査する（スナップショットメソッド）よりもはるかに効率的です。

ガベージコレクション (Garbage Collection)

プロファイラーが `COR_PRF_MONITOR_GC` フラグを指定すると、`ICorProfilerCallback::ObjectAllocated` イベントを除くすべての GC イベントがプロファイラーでトリガーされます。ObjectAllocated イベントは、パフォーマンス上の理由から別のフラグによって明示的に制御されます。`COR_PRF_MONITOR_GC` が有効になると、コンカレント (Concurrent) ガベージコレクションがオフになることに注意してください。

プロファイラーは `GarbageCollectionStarted` / `Finished` コールバックを使用して、GC が実行中であること、およびどの世代 (generation) が対象であるかを識別できます。

移動されたオブジェクトの追跡 (Tracking Moved Objects)

ガベージコレクションは「デッド」オブジェクトが占有するメモリを回収し、その解放されたスペースをコンパクション (compaction) します。その結果、生存オブジェクトはヒープ内で移動されます。この影響により、以前の通知で渡された `ObjectID` の値が変更されます（オブジェクト自身の内部状態は変更されません（他のオブジェクトへの参照は除く）。変更されるのはメモリ上の位置、したがって `ObjectID` だけです）。`MovedReferences` 通知により、プロファイラーは `ObjectID` で情報を追跡する内部テーブルを更新できます。この名前はやや誤解を招きます。移動されなかったオブジェクトに対しても発行されるためです。

ヒープ内のオブジェクト数は数千から数百万に上る可能性があります。このような大量のオブジェクトに対して、各オブジェクトの前後の ID を提供することで移動を通知するのは現実的ではありません。しかし、ガベージコレクタは連続する生存オブジェクトの塊を「ひとまとめ」に移動する傾向があります。つまり、それらはヒープ内の新しい位置に移動しますが、依然として連続しています。この通知は、これらの連続するオブジェクトの塊の「前 (before)」と「後 (after)」の `ObjectID` を報告します（以下の例を参照）。

言い換えると、`ObjectID` の値が以下の範囲内にある場合：

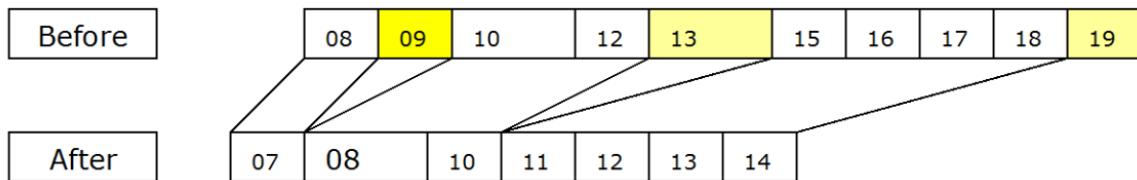
```
oldObjectIDRangeStart[i] <= ObjectID < oldObjectIDRangeStart[i] + cObjectIDRangeLength[i]
```

ここで $0 \leq i < cMovedObjectIDRanges$ のとき、`ObjectID` の値は以下に変更されています：

```
ObjectID - oldObjectIDRangeStart[i] + newObjectIDRangeStart[i]
```

これらのコールバックはすべてランタイムがサスPEND (suspend) されている間に行われるため、ランタイムが再開して別の GC が発生するまで、`ObjectID` の値は変更されません。

例: 以下の図は、ガベージコレクション前の 10 個のオブジェクトを示しています。これらは開始アドレス (ObjectID に相当) 08、09、10、12、13、15、16、17、18、19 に位置しています。ObjectID 09、13、19 はデッド (シェーディングで表示) であり、ガベージコレクション中にそのスペースが回収されます。



「後」の図は、デッドオブジェクトが占めていたスペースがどのように回収されて生存オブジェクトが保持されるかを示しています。生存オブジェクトはヒープ内の新しい位置に移動されました。その結果、それらの ObjectID はすべて変更されます。これらの変更を記述する最も単純な方法は、前後の ObjectID のテーブルです：

	oldObjectIDRangeStart[]	newObjectIDRangeStart[]
0	08	07
1	09	
2	10	08
3	12	10
3	13	
4	15	11
5	16	12
6	17	13
7	18	14
8	19	

これは機能しますが、明らかに、連続する塊の開始位置とサイズを指定することで情報をコンパクトにできます：

	oldObjectIDRangeStart[]	newObjectIDRangeStart[]	cObjectIDRangeLength[]
0	08	07	1
1	10	08	3
2	15	11	4

これはまさに `MovedReferences` が情報を報告する方法に対応しています。`MovedReferencesCallback` は、オブジェクトが実際にヒープ内で再配置される前に、オブジェクトの新しいレイアウトを報告していることに注意してください。したがって、古い ObjectID は `ICorProfilerInfo` インターフェースへの呼び出しにおいてまだ有効です（新しい ObjectID は有効ではありません）。

すべての削除されたオブジェクトの検出 (Detecting All Deleted Objects)

`MovedReferences` は、オブジェクトが移動したかどうかに関わらず、コンパクション GC を生き残ったすべてのオブジェクトを報告します。報告されなかったものは生き残っていません。ただし、すべての GC がコンパクションを行うわけではありません。

プロファイラーは `ICorProfilerInfo2::GetGenerationBounds` を呼び出して、GC ヒープセグメントの境界を取得できます。得られた `COR_PRF_GC_GENERATION_RANGE` 構造体の `rangeLength` フィールドを使用して、コンパクションされた世代における生存オブジェクトの範囲を特定できます。

`GarbageCollectionStarted` コールバックは、現在の GC でどの世代が収集されているかを示します。収集されていない世代にあるすべてのオブジェクトは GC を生き残ります。

非コンパクション GC（オブジェクトがまったく移動しない GC）の場合、どのオブジェクトが GC を生き残ったかを示すために `SurvivingReferences` コールバックが配信されます。

単一の GC が、ある世代に対してはコンパクションを行い、別の世代に対しては非コンパクションである場合があることに注意してください。特定の世代は、特定の GC に対して `SurvivingReferences` コールバックか `MovedReferences` コールバックのいずれかを受信しますが、両方を受信することはありません。

備考 (Remarks)

アプリケーションは、ランタイムがヒープに関する情報をコードプロファイラーに渡し終わるまで、ガベージコレクションの後に一時停止されます。メソッド `ICorProfilerInfo::GetClassFromObject` を使用して、オブジェクトがインスタンスであるクラスの `ClassID` を取得できます。メソッド `ICorProfilerInfo::GetTokenFromClass` を使用して、クラスに関するメタデータ情報を取得できます。

`RootReferences2` により、プロファイラーは特殊なハンドルを介して保持されているオブジェクトを識別できます。

`GetGenerationBounds` によって提供される世代境界情報と `GarbageCollectionStarted` によって提供される収集対象世代情報を組み合わせることで、プロファイラーは収集されなかった世代に存在するオブジェクトを識別できます。

オブジェクトインスペクション (Object Inspection)

`FunctionEnter2` / `Leave2` コールバックは、メモリの領域として関数の引数と戻り値に関する情報を提供します。引数は指定されたメモリ領域内に左から右に格納されます。プロファイラーは関数のメタデータシグネチャを使用して引数を解釈できます。以下のように解釈されます：

ELEMENT_TYPE	表現方法
プリミティブ (ELEMENT_TYPE <= R8, I, U)	プリミティブ値
値型 (VALUETYPE)	型に依存
参照型 (CLASS, STRING, OBJECT, ARRAY, GENERICINST, SZARRAY)	ObjectID (GC ヒープへのポインタ)
BYREF	マネージドポインタ (ObjectID ではないが、スタックまたは GC ヒープを指している場合がある)
PTR	アンマネージドポインタ (GC による移動不可)
FNPTR	ポインタサイズの不透明な値
TYPEDBYREF	マネージドポインタの後にポインタサイズの不透明な値

ObjectID とマネージドポインタ (managed pointer) の違いは以下のとおりです：

- ObjectId は GC ヒープまたはフローズンオブジェクトヒープ (frozen object heap) のみを指す。マネージドポインタはスタックも指す可能性がある。
- ObjectId は常にオブジェクトの先頭を指す。マネージドポインタはオブジェクトのフィールドの 1 つを指す場合がある。
- マネージドポインタは ObjectId を期待する関数に渡すことができない。

複合型のインスペクション (Inspecting Complex Types)

参照型または非プリミティブ値型のインスペクションには、いくつかの高度なテクニックが必要です。

値型および文字列・配列以外の参照型の場合、[GetClassLayout](#) が各フィールドのオフセットを提供します。プロファイラーはメタデータを使用してフィールドの型を判断し、再帰的に評価できます（[GetClassLayout](#) はクラス自身で定義されたフィールドのみを返すことに注意してください。親クラスで定義されたフィールドは含まれません）。

ボックス化 (boxing) された値型の場合、[GetBoxClassLayout](#) がボックス内の値型のオフセットを提供します。値型自体のレイアウトは変更されないため、プロファイラーがボックス内の値型を見つけたら、[GetClassLayout](#) を使用してそのレイアウトを理解できます。

文字列の場合、[GetStringClassLayout](#) が文字列オブジェクト内の重要なデータのオフセットを提供します。

配列はやや特殊であり、型ごとではなくすべての配列オブジェクトに対して関数を呼び出す必要があります（これは、オフセットで記述するには配列のフォーマットが多すぎるためです）。[GetArrayObjectInfo](#) がこの解釈を行うために提供されています。

静的フィールドのインスペクション (Inspecting Static Fields)

[GetThreadStaticAddress](#)、[GetAppDomainStaticAddress](#)、[GetContextStaticAddress](#)、および [GetRVAStaticAddress](#) は、静的フィールドの場所に関する情報を提供します。その場所のメモリを見て、以下のように解釈します：

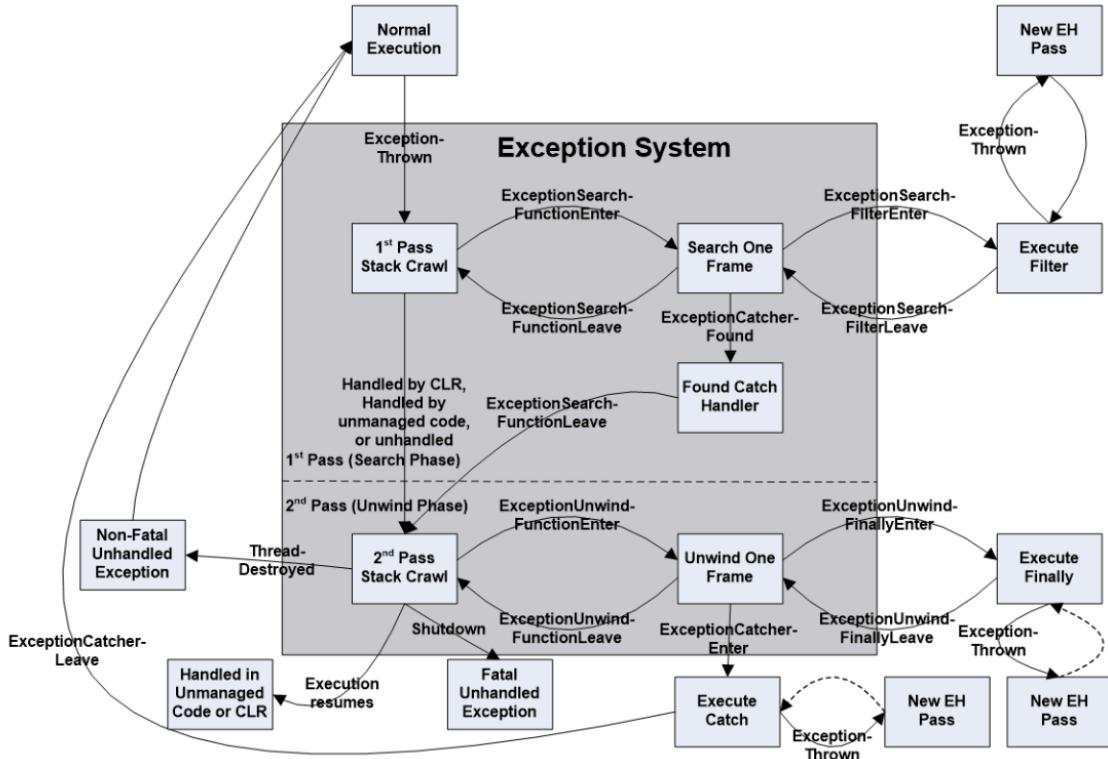
- 参照型：ObjectId
- 値型：実際の値を含むボックスの ObjectId
- プリミティブ型：プリミティブ値

静的フィールドには 4 つの種類があります。以下のテーブルでは、それらの内容と識別方法を説明します。

静的型	定義	メタデータでの識別方法
AppDomain	基本的な静的フィールド — 各 AppDomain で異なる値を持つ。	カスタム属性が付与されていない静的フィールド
Thread	マネージド TLS — 各スレッドおよび各 AppDomain で一意の値を持つ静的フィールド。	System.ThreadStaticAttribute を持つ静的フィールド
RVA	モジュールのデータセクションにホームを持つプロセススコープの静的フィールド。	hasRVA フラグを持つ静的フィールド
Context	各 COM+ コンテキスト (Context) で異なる値を持つ静的フィールド。	System.ContextStaticAttribute を持つ静的フィールド

例外 (Exceptions)

例外の通知は、すべての通知の中で記述および理解が最も困難です。これは、例外処理(exception handling)に固有の複雑性のためです。以下で説明される例外通知のセットは、洗練されたプロファイラーが必要とするすべての情報を提供するように設計されています。これにより、プロファイル対象プロセスのすべてのスレッドについて、あらゆる瞬間に、どのパス(第1パスまたは第2パス)、どのフレーム、どのフィルター(filter)、どのfinallyブロックが実行されているかを常に追跡できます。例外通知は ThreadID を提供しませんが、プロファイラーは常に `ICorProfilerInfo::GetCurrentThreadID` を呼び出して、どのマネージドスレッドが例外をスローしているかを特定できます。



上の図は、例外イベントを監視する際にコードプロファイラーがさまざまなコールバックをどのように受信するかを示しています。各スレッドは「通常の実行(Normal Execution)」から始まります。スレッドが大きな灰色のボックス内の状態にあるとき、例外システム(exception system)がスレッドを制御しています。スレッドがこれらの状態のいずれかにある間に発生する例外に関連しないコールバック(例：`ObjectAllocated`)は、例外システム自体に帰属させることができます。スレッドが大きな灰色のボックスの外側の状態にあるときは、任意のマネージドコードを実行しています。

ネストされた例外 (Nested Exceptions)

例外処理の途中でマネージドコードに遷移したスレッドは、別の例外をスローする可能性があります。これにより、完全に新しい例外処理パスが開始されます(上図の「New EH Pass」ボックス)。そのような「ネストされた」例外が元の例外のフィルター/finally/catchをエスケープすると、元の例外に影響を与えることがあります：

- ネストされた例外がフィルター内で発生し、フィルターをエスケープした場合、フィルターは「false」を返したとみなされ、第1パスが継続します。
- ネストされた例外が finally 内で発生し、finally をエスケープした場合、元の例外の処理は再開されません。
- ネストされた例外が catch 内で発生し、catch をエスケープした場合、元の例外の処理は再開されません。

アンマネージドハンドラー (Unmanaged Handlers)

例外がアンマネージドコードで処理される場合があります。この場合、プロファイラーはアンワインドフェーズ(unwind phase)を見ますが、catch ハンドラーの通知は受信しません。実行はアンマネージドコード内で正常に再開されます。アンマネージドを認識するプロフ

アイラーはこれを検出できますが、マネージドのみのプロファイラーは以下を含む（これらに限定されない）さまざまなことを検出する可能性があります：

- アンマネージドコードがマネージドコードを呼び出すか、マネージドコードに戻るときの [UnmanagedToManagedTransition](#) コールバック。
- スレッドの終了（アンマネージドコードがスレッドのルートにあった場合）。
- アプリケーションの終了（アンマネージドコードがアプリを終了させた場合）。

CLR ハンドラー (CLR Handlers)

CLR 自体が例外を処理する場合があります。この場合、プロファイラーはアンワインドフェーズを見ますが、catch ハンドラーの通知は受信しません。マネージドコードまたはアンマネージドコード内で正常に実行が再開される場合があります。

未処理の例外 (Unhandled Exceptions)

デフォルトでは、未処理の例外 (unhandled exception) はプロセスの終了につながります。アプリケーションがレガシー例外ポリシー (legacy exception policy) にロックバックしている場合、特定の種類のスレッドでの未処理の例外はスレッドの終了のみにつながる場合があります。

コード生成 (Code Generation)

IL からネイティブコードへ (Getting from IL to Native Code)

.NET アセンブリの IL は、2 つの方法のいずれかでネイティブコードにコンパイルされます。実行時に JIT コンパイルされるか、NGEN.exe (CoreCLR では CrossGen.exe) というツールによって「ネイティブイメージ (native image)」にコンパイルされます。JIT コンパイラと NGEN の両方に、コード生成を制御する多くのフラグがあります。

アセンブリがロードされる時点で、CLR はまずそのアセンブリのネイティブイメージを探します。適切なコード生成フラグのセットを持つネイティブイメージが見つからない場合、CLR は実行中に必要に応じてアセンブリ内の関数を JIT コンパイルします。ネイティブイメージが見つかってロードされた場合でも、CLR はアセンブリ内的一部の関数を JIT コンパイルすることができます。

💡 初心者向け補足

ネイティブイメージ (Native Image) は、事前にコンパイルされたネイティブコードを含む DLL です。JIT コンパイルのオーバーヘッドを削減し、アプリケーションの起動時間を短縮します。Java の AOT (Ahead-of-Time) コンパイルの概念に似ています。.NET ではかつて NGEN が使用されていましたが、現在は ReadyToRun (R2R) 形式が主流です。

コード生成に対するプロファイラーの制御 (Profiler Control over Code-Generation)

プロファイラーは、以下のようにコード生成を制御できます：

フラグ	効果
COR_PRF_USE_PROFILE_IMAGES	ネイティブイメージ検索をプロファイラー拡張イメージ (ngen /profile) に向ける。JIT コードには効果なし。

フラグ	効果
COR_PRF_DISABLE_INLINING	ネイティブイメージ検索には効果なし。JIT 時にインライン化 (inlining) を無効にする。他のすべての最適化は有効なままで。
COR_PRF_DISABLE_OPTIMIZATIONS	ネイティブイメージ検索には効果なし。JIT 時にインライン化を含むすべての最適化を無効にする。
COR_PRF_MONITOR_ENTERLEAVE	ネイティブイメージ検索をプロファイラー拡張イメージ (ngen /profile) に向ける。JIT 時に生成コードに Enter/Leave フックを挿入する。
COR_PRF_MONITOR_CODE_TRANSITIONS	ネイティブイメージ検索をプロファイラー拡張イメージ (ngen /profile) に向ける。JIT 時にマネージド/アンマネージド遷移ポイントにフックを挿入する。

プロファイラーとネイティブイメージ (Profilers and Native Images)

NGEN.exe がネイティブイメージを作成するとき、CLR が実行時に行うはずだった作業の多く（例えば、クラスのロードやメソッドのコンパイル）を行います。その結果、NGEN 時に作業が行われた場合、実行時に特定のプロファイラーコールバックが受信されないことがあります：

- JITCompilation*
- ClassLoad*, ClassUnload*

この状況に対処するために、プロファイラー拡張ネイティブイメージを要求してプロセスに干渉することを望まないプロファイラーは、FunctionID や ClassID が出現するたびに、必要なデータを遅延的 (lazily) に収集する準備をしておくべきです。

プロファイラー拡張ネイティブイメージ (Profiler-Enhanced Native Images)

NGEN /profile でネイティブイメージを作成すると、イメージのプロファイリングを容易にする一連のコード生成フラグが有効になります：

- Enter/Leave フックがコードに挿入される。
- マネージド/アンマネージド遷移フックがコードに挿入される。
- ネイティブイメージ内の各関数が初めて呼び出された時に `JITCachedFunctionSearch` 通知が提供される。
- ネイティブイメージ内の各クラスが初めて使用された時に `ClassLoad` 通知が提供される。

プロファイラー拡張ネイティブイメージは通常のネイティブイメージとは大きく異なるため、プロファイラーは追加の干渉 (perturbation) が許容できる場合にのみ使用すべきです。

プロファイリングにおけるセキュリティの問題 (Security Issues in Profiling)

プロファイラー DLL は、実質的に CLR の実行エンジン (execution engine) 自体の一部として動作するアンマネージド DLL です。その結果、プロファイラー DLL のコードはマネージドコードアクセスセキュリティ (Code Access Security, CAS) の制約を受けず、プロファイル対象アプリケーションを実行しているユーザーに対して OS が課す制限のみが適用されます。

マネージドコードとアンマネージドコードのプロファイラー内の組み合わせ (Combining Managed and Unmanaged Code in a Code Profiler)

CLR プロファイリング API を詳しく検討すると、マネージドコンポーネントとアンマネージドコンポーネントを持ち、COM Interop や PInvoke 呼び出しを通じて相互に呼び出すプロファイラーを書けるという印象を受けるかもしれません。

これは設計上は可能ですが、CLR プロファイリング API はこれをサポートしていません。CLR プロファイラーは純粋にアンマネージドであることが想定されています。CLR プロファイラーからマネージドコードとアンマネージドコードを組み合わせようすると、クラッシュ、ハング (hang)、デッドロック (deadlock) を引き起こす可能性があります。マネージド部分のプロファイラーがアンマネージドコンポーネントにイベントを「発火 (fire)」し、それが続いてプロファイラーのマネージド部分に呼び出されることになるため、危険は明白です。

CLR プロファイラーがマネージドコードを安全に呼び出せる唯一の方法は、メソッドの MSIL 本体を置き換えることです。プロファイラーは関数の JIT コンパイルが完了する前に、メソッドの MSIL 本体にマネージド呼び出しを挿入し、JIT にコンパイルさせます。この手法は、マネージドコードの選択的なインストルメンテーション (selective instrumentation) に使用でき、JIT に関する統計やタイミングの収集にも使用できます。

あるいは、コードプロファイラーは、すべてのマネージド関数の MSIL 本体にネイティブ「フック (hook)」を挿入し、アンマネージドコードに呼び出すこともできます。この手法はインストルメンテーションとカバレッジ (coverage) に使用できます。例えば、コードプロファイラーは、ブロックが実行されたことを保証するために、すべての MSIL ブロックの後にインストルメンテーションフックを挿入できます。メソッドの MSIL 本体の変更は非常に繊細な操作であり、考慮すべき多くの要因があります。

💡 初心者向け補足

プロファイラーを純粋にアンマネージドコード (C/C++ など) で書くべき理由は、マネージドコードを含めると循環的な依存関係が発生するためです。例えば、プロファイラーのマネージドコードが GC イベントを発生させ、そのイベントがプロファイラーのコールバックを呼び出し...というループが発生してデッドロックに至ります。MSIL の書き換えによるインストルメンテーションは、Java のバイトコードインストルメンテーション (`java.lang.instrument`) と概念的に似ています。

アンマネージドコードのプロファイリング (Profiling Unmanaged Code)

ランタイムプロファイリングインターフェースには、アンマネージドコードのプロファイリングのための最小限のサポートがあります。以下の機能が提供されています：

- スタックチェーン (stack chain) の列挙。これにより、コードプロファイラーはマネージドコードとアンマネージドコードの境界を判断できます。
- スタックチェーンがマネージドコードに対応するかネイティブコードに対応するかの判定。

これらのメソッドは、CLR デバッグ API のインプロセスサブセット (in-process subset) を通じて利用できます。これらは CorDebug.IDL で定義され、DebugRef.doc で説明されています。詳細については両方を参照してください。

サンプリングプロファイラー (Sampling Profilers)

ハイジャック (Hijacking)

一部のサンプリングプロファイラーは、サンプル時にスレッドをハイジャック (hijack) して、サンプルの作業を強制的に行わせることで動作します。これは非常にトリッキーな実践であり、推奨しません。このセクションの残りの部分は、主にこの方法を取ることを思いとどまらせるためのものです。

ハイジャックのタイミング (Timing of Hijacks)

ハイジャックを行うプロファイラーは、ランタイムのサスペンションイベント (`COR_PRF_MONITOR_SUSPENDS`) を追跡する必要があります。プロファイラーは、`RuntimeThreadSuspended` コールバックから戻ると、ランタイムがそのスレッドをハイジャックすると仮定すべきです。プロファイラーは、自身のハイジャックがランタイムのハイジャックと競合しないようにする必要があります。そのために、プロファイラーは以下を保証する必要があります：

1. プロファイラーは `RuntimeThreadSuspended` と `RuntimeThreadResumed` の間でスレッドのハイジャックを試みない。
2. プロファイラーが `RuntimeThreadSuspended` コールバックが発行される前にハイジャックを開始していた場合、ハイジャックが完了するまでコールバックから戻らない。

これは簡単な同期 (synchronization) で実現できます。

ランタイムの初期化 (Initializing the Runtime)

プロファイラーが `ICorProfilerInfo` 関数を呼び出す独自のスレッドを持っている場合、スレッドのサスペンションを行う前に、そのような関数を1回呼び出しておく必要があります。これは、ランタイムにはスレッドごとの状態 (per-thread state) があり、デッドロックの可能性を避けるために、他のすべてのスレッドが実行中の状態で初期化する必要があるためです。

プロファイラビリティの実装

原文

この章の原文は [Implementing Profilability](#) です。

本ドキュメントでは、CLR の機能にプロファイラビリティ (profilability) を追加するための技術的な詳細を説明します。これは、自身の機能をプロファイル可能にするためにプロファイリング API を変更する開発者を対象としています。

💡 初心者向け補足

「プロファイラビリティ (profilability)」とは、プログラムの実行状況（パフォーマンス、メモリ使用量、関数の呼び出し回数など）を外部ツール（プロファイラー）から観測・分析できるようにする仕組みのことです。CLR に新しい機能を追加する際には、その機能もプロファイラーから正しく監視できるようにする必要があります。Java における JVMTI (JVM Tool Interface) に相当する概念です。

設計哲学

コントラクト

プロファイリング API でどのコントラクト (contract) を使用すべきかの詳細に入る前に、全体的な哲学を理解しておくことが有用です。

CLR 全体（プロファイリング API の外側）におけるデフォルトコントラクト運動の背景にある哲学は、CLR の大部分がスロー (throw) やトリガー (trigger) といった「攻撃的な動作 (aggressive behavior)」に対処できるようにすることです。後述しますが、これはコールバック (ICorProfilerCallback) のコントラクトに関する推奨事項と表裏一体であり、コールバックのコントラクトは一般的に、より許容的（「攻撃的」）なコントラクトの選択を好みます。これにより、プロファイラーはコールバック中に最大限の柔軟性を持つことができます (ICorProfilerInfo を通じてどの CLR 呼び出しが可能かという点で)。

しかし、Info 関数 (ICorProfilerInfo) は正反対です。許容的ではなく、制限的であることが好まれます。なぜでしょうか？ プロファイラーがこれらの関数を可能な限り多くの場所から安全に呼び出せるようにしたいからです。たとえ、本来望ましいよりも制限的なコールバック（何らかの理由で GC_NOTRIGGER でなければならないコールバックなど）からであってもです。

また、ICorProfilerInfo でより制限的なコントラクトを好むことは、CLR 全体のデフォルトコントラクト哲学と矛盾しません。なぜなら、制限的である必要がある CLR 関数は少数派であると想定されているからです。ICorProfilerInfo は、このカテゴリに該当する呼び出しパスのルートです。プロファイラーはデリケートなタイミングで CLR に呼び出しを行う可能性があるため、これらの呼び出しあるは可能な限り控えめであることが求められます。これらは CLR のメインストリームの関数ではなく、慎重さが求められる特殊な呼び出しパスのごく少数のものです。

したがって、一般的なガイドラインとしては、可能な限り CLR 全体でデフォルトコントラクトを使用することです。しかし、プロファイラーから発生する呼び出しのパスを切り開く必要がある場合（すなわち ICorProfilerInfo から）、そのパスはコントラクトを明示的に指定し、デフォルトよりも制限的にする必要があります。

初心者向け補足

「コントラクト (contract)」とは、CLR の内部コードで使用される一種の「約束事」や「宣言」です。関数がどのような動作をするか（例外をスローするかどうか、GC をトリガーするかどうか、ロックを取得するかどうかなど）を明示的に記述します。Java のアノテーションや、C++ の `noexcept` 指定子に似た概念ですが、CLR ではより広範囲の動作を宣言します。これにより、開発者やツールが関数の安全性を事前にチェックでき、デバッグ時の問題特定が容易になります。

パフォーマンスか使いやすさか？

どちらも実現できれば理想的です。しかし、トレードオフが必要な場合はパフォーマンスを優先してください。プロファイリング API は、CLR とプロファイリング DLL の間の軽量で薄いインプロセス (in-process) レイヤーとなることを意図しています。プロファイラーの開発者はごく少数であり、そのほとんどが高度な技術を持つ開発者です。CLR による入力の簡単なバリデーション (validation) は期待されています。しかし、それにも限度があります。たとえば、すべてのプロファイラー ID を考えてみてください。それらは、直接呼び出される C++ EE オブジェクトインスタンスのキャストされたポインタに過ぎません (`AppDomain*`、`MethodTable*` など)。プロファイラーが不正な ID を渡すとどうなるでしょうか？CLR がアクセス違反 (AV) を起こします！これは想定された動作です。CLR はルックアップを検証するために ID をハッシュしたりしません。プロファイラーは自分が何をしているか理解していることが前提とされています。

とはいっても、繰り返しますが、CLR による入力の簡単なバリデーションは期待されています。NULL ポインタのチェック、検査対象のクラスが初期化済みであることの確認、「並行パラメータ (parallel parameters)」の一貫性の検証（例：配列ポインタパラメータは、そのサイズパラメータが非ゼロの場合、非 `null` でなければならない）などです。

ICorProfilerCallback

このインターフェースは、CLR からプロファイラーへ興味深いイベントを通知するためのコールバック (callback) で構成されます。各コールバックは、EE 内の薄いメソッドでラップされており、そのメソッドがプロファイラーの `ICorProfilerCallback(2)` の実装を見つけ、対応するメソッドを呼び出す処理を行います。

プロファイラーは、`ICorProfilerInfo::SetEventMask()` / `ICorProfilerInfo::SetEventMask2()` の呼び出しで対応するフラグ (flag) を指定することによりイベントをサブスクライブ (subscribe) します。プロファイリング API はこれらの選択を保存し、フラグに対応するビットに対してマスクする特殊なインライン関数 (`CORProfiler*`) を通じて CLR に公開します。すると、CLR のあちこちに、イベントの発生時にプロファイラーに通知するため `ICorProfilerCallback` ラッパーを呼び出すコードがありますが、この呼び出しはフラグが設定されているかどうか（特殊なインライン関数の呼び出しによって判定）を条件としています：

```
{  
    // プロファイラーがフラグを設定したかチェック  
  
    BEGIN_PROFILER_CALLBACK(CORProfilerTrackModuleLoads());  
  
    // ProfControlBlock ラッパーを通じてプロファイラーのコールバック実装を呼び出す  
    // DoOneProfilerIteration 内で EvacuationCounterHolder を通じてプロファイラーを固定 (pin) する  
    (&g_profControlBlock)->ModuleLoadStarted((ModuleID) this);  
  
    // コールバック完了後にプロファイラーの固定を解除
```

```

    END_PROFILER_CALLBACK();
}

}

```

明確にしておくと、上記のコードはコードベース全体に散りばめられているものです。呼び出される関数（この場合は `ModuleLoadStarted()`）は、プロファイラーのコールバック実装（この場合は `ICorProfilerCallback::ModuleLoadStarted()`）に対する私たちのラッパーです。すべてのラッパーは単一のファイル (`vm\EEToProfInterfaceImpl.cpp`) にあり、以下のセクションで提供されるガイドンスはそれらのラッパーに関するものであり、ラッパーを呼び出す上記のサンプルコードに関するものではありません。

`BEGIN_PROFILER_CALLBACK` マクロは、引数として渡された式を評価します。式が `TRUE` の場合、`BEGIN_PROFILER_CALLBACK` と `END_PROFILER_CALLBACK` マクロの間のコードが実行され、プロファイラーは `ProfControlBlock` ラッパーを通じてメモリに固定 (pin) されます（つまり、プロファイラーはプロセスからデタッチできなくなります）。式が `FALSE` の場合、`BEGIN_PROFILER_CALLBACK` と `END_PROFILER_CALLBACK` マクロの間のすべてのコードはスキップされます。`BEGIN_PROFILER_CALLBACK` と `END_PROFILER_CALLBACK` マクロの詳細については、コードベース内でその定義を見つけ、そこにあるコメントを読んでください。

コントラクト

各コールバックラッパーには、先頭にいくつかの共通の定型コードが必要です。以下は例です：

```

CONTRACTL
{
    // Yay!
    NOTHROW;

    // Yay!
    GC_TRIGGER;

    // Yay!
    MODE_PREEMPTIVE;

    // Yay!
    CAN_TAKE_LOCK;
}

CONTRACTL_END;
CLR_TO_PROFILER_ENTRYPOINT((LF_CORPROF,
    LL_INFO10,
    "***PROF: useful logging text here.\n"));

```

重要なポイント：

- throws、triggers、mode、take_lock、および ASSERT_NO_EE_LOCKS_HELD()（後者はコールバックでのみ必須）の値を明示的に指定する必要があります。これにより、プロファイラー開発者向けのドキュメントを正確に保つことができます。
- 各コントラクトにはそれぞれのコメントが必要です（コントラクトの具体的な詳細は以下を参照）

各コントラクトの種類には「推奨値 (preferred value)」があります。可能であればその値を使用し、「Yay!」とコメントしてください。これにより、コピー＆ペーストする他の開発者にとって何がベストかが分かります。推奨値を使用できない場合は、その理由をコメントしてください。

以下はコールバックの推奨値です。

推奨値	理由	詳細
NOTHROW	コールバックをあらゆる CLR コンテキストから発行可能にする。Info 関数も NOTHROW であるべきなので、プロファイラーにとって負担にならない。	プロファイラーが THROWS の Info 関数をここから呼び出すると、プロファイラーが try/catch で囲んでいても throws 違反が発生する（コントラクトシステムはプロファイラーの try/catch を認識できないため）。そのため、プロファイラーへの呼び出しの直前にスコープされた CONTRACT_VIOLATION(ThrowsViolation) を挿入する必要がある。
GC_TRIGGERED	プロファイラーが呼び出せる Info 関数の柔軟性を最大化する。	コールバックがデリケートなタイミングで行われ、すべてのオブジェクト参照を保護することがエラーを起こしやすいか、パフォーマンスを大幅に低下させる場合は、GC_NOTRIGGER を使用する（もちろんコメント付きで！）。
MODE_PREEMPTIVE (可能な場合)、それ以外は MODE_COOPERATIVE	MODE_PREEMPTIVE はプロファイラーが呼び出せる Info 関数の柔軟性を最大化する (ObjectId のために協調モードが必要な場合を除く)。また、 MODE_PREEMPTIVE は EE 全体で推奨される「デフォルト」コントラクトであり、コールバックをプリエンプティブモードにすることで、EE の他の場所でもプリエンプティブの使用を促進する。	ObjectId パラメータをプロファイラーに渡す場合は MODE_COOPERATIVE が妥当。それ以外は MODE_PREEMPTIVE を指定する。コールバックの呼び出し元はすでにプリエンプティブモードにあるはず。そうでない場合は、なぜそうでないかを再検討し、呼び出し元をプリエンプティブに変更することを検討する。それが無理な場合は、コールバック呼び出し前に GCX_PREEMP() マクロを使用する必要がある。
CAN_TAKE_LOCK	プロファイラーが呼び出せる Info 関数の柔軟性を最大化する。	特に追加事項なし。
ASSERT_NO_EE_LOCKS_HELD()	プロファイラーが呼び出せる Info 関数にさらなる柔軟性を与える。Info がロックの再取得や順序外のロック取得を試みないことを保証する（再取得すべきロックや順序	これは実際にはコントラクトではないが、コントラクトブロックは忘れないための便利な場所である。コントラクトと同様に、これを指定できない場合はその理由をコメントする。

推奨値	理由	詳細
	を壊すロックが取得されていないため)。	

注意：EE_THREAD_NOT_REQUIRED / EE_THREAD_REQUIRED はコールバックでは指定する必要はありません。GC コールバックはそもそも「REQUIRED」を指定できません（EE スレッドが存在しない可能性がある）し、これらは Info 関数（プロファイラー → CLR）でのみ考慮が必要です。

エントリポイントマクロ

上記の例のように、コントラクトの後にはエントリポイントマクロ (entrypoint macro) を配置する必要があります。このマクロは、ログの記録、EE スレッドオブジェクトへのコールバック中であるというマーキング、スタックガードの除去、およびいくつかのアサートの処理を行います。使用できるマクロにはいくつかのバリエントがあります：

`CLR_TO_PROFILER_ENTRYPOINT`

これが推奨されるマクロであり、通常使用されるマクロです。

他のマクロを使用することもできますが、上記の（推奨される）マクロを使用できない理由をコメントしなければなりません。

`*_FOR_THREAD_*`

これらのマクロは、`ThreadID` パラメータの値が必ずしも現在の `ThreadID` と一致しない `ICorProfilerCallback` メソッドに使用されます。`ThreadID` をこれらのマクロの最初のパラメータとして指定する必要があります。マクロは `GetThread()` の代わりに指定された `ThreadID` を使用して、そのコールバックがその `ThreadID` に対して現在許可されているかどうか（すなわち、その `ThreadID` に対してまだ `ThreadDestroyed()` が発行されていないこと）をアサートします。

ICorProfilerInfo

このインターフェースは、プロファイラーが CLR に呼び出すために使用するエントリポイント (entrypoint) で構成されます。

同期 / 非同期

各 Info 呼び出しは、同期 (synchronous) または非同期 (asynchronous) のいずれかに分類されます。同期関数はコールバック内からのみ呼び出す必要がありますが、非同期関数はいつでも安全に呼び出すことができます。

💡 初心者向け補足

ここでの「同期 (synchronous)」と「非同期 (asynchronous)」は、一般的なプログラミングにおける `async/await` とは異なる意味で使われています。プロファイリング API における「同期」とは、「CLR がプロファイラーにコールバック通知を送っている最

中にのみ呼び出せる関数」を意味します。つまり、CLR → プロファイラーのコールバックのスタック上にいるときだけ安全に呼び出せるということです。「同期」は、プロファイラーがいつでも（コールバック中でなくとも）安全に呼び出せる関数を指します。

同期

Info 呼び出しの大多数は同期です。プロファイラーがコールバック内で実行している間にのみ呼び出すことができます。言い換えると、同期 Info 関数を呼び出すことが合法であるためには、スタック上に ICorProfilerCallback が存在している必要があります。これは EE スレッドオブジェクトのビットで追跡されます。コールバックが行われるとビットを設定し、コールバックが返されるとビットをリセットします。同期 Info 関数が呼び出されると、このビットをテストし、設定されていなければ呼び出しを拒否します。

EE スレッドのないスレッド

上記のビットは EE スレッドオブジェクトを使用して追跡されるため、EE スレッドオブジェクトを持つスレッド上で行われた Info 呼び出しのみが「同期性」を強制されます。EE スレッドではないスレッド上で行われた Info 呼び出しは、即座に合法とみなされます。これは一般的に問題ありません。なぜなら、再入時に問題となる複雑なコンテキストを構築するのは主に EE スレッドだからです。また、正確性を保証するのは最終的にプロファイラーの責任です。前述のように、パフォーマンス上の理由から、プロファイリング API は歴史的に正確性チェックを最小限にとどめ、負荷を増加させないようにしています。通常、プロファイラーが EE スレッドではないスレッド上で行う Info 呼び出しは、以下のカテゴリに分類されます：

- サーバー GC を行っているスレッドでの GC コールバック中に行われる Info 呼び出し。
- プロファイラーが作成したスレッド上で行われる Info 呼び出し（例：サンプリングスレッド。したがってスタック上に CLR コードがない）。

Enter / Leave フック

プロファイラーが Enter / Leave フック (hook) を要求し、ファストパス (fast path)（すなわち、介在するプロファイリング API コードなしに、JIT コンパイルされたコードからプロファイラーへの直接関数呼び出し）を使用する場合、Enter / Leave フック内からの Info 関数の呼び出しはすべて非同期とみなされます。これもまた実用的な理由によるものです。プロファイリング API コードが実行される機会がなければ（パフォーマンスのため）、コールバック内で実行中であることを示す EE スレッドビットを設定する機会がありません。これは、プロファイラーが Enter / Leave フック内からは非同期安全な Info 関数のみを呼び出すことに制限されることを意味します。Enter / Leave に直接関数呼び出しを必要とするほどパフォーマンスを重視するプロファイラーは、おそらく Enter / Leave フック内から Info 関数を呼び出すことはないため、これは通常は許容されます。

もう一つの方法は、プロファイラーが引数または戻り値の情報を要求するフラグを設定することです。これにより、プロファイラーの Enter / Leave フックに情報を準備するために介在するプロファイリング API の C 関数が呼び出されることが強制されます。このようなフラグが設定されると、プロファイリング API は、プロファイラーの引数 / 戻り値情報を準備するこの C 関数の内部から EE スレッドビットを設定します。これにより、プロファイラーは Enter / Leave フック内から同期 Info 関数を呼び出すことが可能になります。

非同期

非同期 Info 関数は、いつでも（コールバック中であるかどうかに関わらず）安全に呼び出せる関数です。非同期 Info 関数は比較的少数です。これらは、ハイジャッキング型のサンプリングプロファイラー (hijacking sampling profiler)（例：Visual Studio プロファイラー）がサンプル内から呼び出したいものです。非同期ヒラベル付けされた Info 関数が、あらゆる可能なコールスタックから実行できることは極めて重要です。スレッドは、任意の数のロック（スピンドルロック (spin lock)、スレッドストアロック (thread store lock)、OS ヒープロック (OS heap lock) など）を保持している最中に割り込まれ、プロファイラーによって非同期 Info 関数を介してランタイムに再入せられる可能性があります。これは容易にデッドロック (deadlock) やデータ破壊 (data corruption) を引き起こし得ます。非同期 Info 関数が自身の安全性を確保する方法は 2 つあります：

- 非常に非常にシンプルにする。ロックを取得しない、GC をトリガーしない、不整合な可能性のあるデータにアクセスしない、など。または
- それよりも複雑にする必要がある場合、先頭に十分なチェックを設けて、ロックやデータ構造などが安全な状態にあることを確認してから処理を続行する。
 - 多くの場合、これには現在のスレッドが現在 Forbid Suspend Thread リージョン内にあるかどうかを確認し、そうであればエラーで中止することが含まれるが、すべてのケースで十分なチェックではない。
 - DoStackSnapshot は複雑な非同期関数の例である。チェックの組み合わせ（現在のスレッドが現在 Forbid Suspend Thread リージョン内にあるかどうかの確認を含む）を使用して、処理を続行するか中止するかを判定する。

初心者向け補足

「ハイジャッキング (hijacking)」とは、プロファイラーがターゲットスレッドを一時停止し、そのスレッドの命令ポインタを変更して自分のコードを実行させることで、実行の制御を「乗っ取る」手法のことです。サンプリングプロファイラー（Visual Studio のパフォーマンス分析ツールなど）がスレッドの実行状態を定期的にスナップショットするために使用します。この手法では、ターゲットスレッドがあらゆる状態（ロック保持中、GC 中など）にある可能性があるため、ハイジャック中に呼び出される関数は非常に慎重に設計される必要があります。

コントラクト

各 Info 関数には、先頭にいくつかの共通の定型コードが必要です。以下は例です：

```
CONTRACTL
{
    // Yay!
    NOTHROW;

    // Yay!
    GC_NOTRIGGER;

    // Yay!
    MODE_ANY;

    // Yay!
    EE_THREAD_NOT_REQUIRED;

    // Yay!
    CANNOT_TAKE_LOCK;
}
```

CONTRACTL_END;

```

PROFILER_TO_CLR_ENTRYPOINT_SYNC((LF_CORPROF,
                                LL_INFO1000,
                                "***PROF: EnumModuleFrozenObjects 0x%p.\n",
                                moduleID));

```

以下は、各コントラクトの種類に対する「推奨値」です。これらはコールバックの推奨値とはほとんど異なることに注意してください！混乱した場合は、上記の設計哲学のセクションを読み直してください。

推奨値	理由	詳細
NOTHROW	プロファイラーにとって呼び出しが容易になる。プロファイラー自身の try / catch が不要。	呼び出し先が NOTHROW であれば NOTHROW を使用する。そうでない場合は、自前の try / catch を設定するよりも自身を THROWS とマークするほうが実際には良い。プロファイラーは、複数の Info 呼び出しを共有の try ブロックで囲むことにより、おそらくより効率的に処理できる。
GC_NOTRIGGER	より多くの状況からプロファイラーが安全に呼び出せる。	トリガーしないように最大限努力する。Info 関数がトリガーする可能性がある場合（例：まだロードされていない型をロードする場合）、可能であればプロファイラーがトリガーパスを取らないように指定できる手段を確保する（例：FALSE に設定できる fAllowLoad パラメータ）。そして、コントラクトを条件付きで記述する。
MODE_ANY	より多くの状況からプロファイラーが安全に呼び出せる。	パラメータまたは戻り値が ObjectId の場合は MODE_COOPERATIVE が妥当。それ以外では MODE_ANY が強く推奨される。
CANNOT_TAKE_LOCK	より多くの状況からプロファイラーが安全に呼び出せる。	呼び出し先がロックを取得しないことを確認する。取得する必要がある場合は、正確にどのロックが取得されるかコメントする。
オプション： EE_THREAD_NOT_REQUIRED	プロファイラーが GC コールバックやプロファイラーが起動したスレッド（例：サンプリングスレッド）からこの Info 関数を使用できるようにする。	これらのコントラクトはまだ強制されていないため、空欄のままにしても問題ない。Info 関数が現在の EE スレッドを必要としない（またはそれを必要とする関数を呼び出さない）と確信できる場合は、スレッドコントラクトが強制されるようになったときのヒントとして EE_THREAD_NOT_REQUIRED を指定できる。

以下は、上記の例ほど「Yay!」ではない関数のコメント付きコントラクトの例です：

CONTRACTL

{

```
// ModuleILHeap::CreateNew ガスローする
```

```
THROWS;
```

```
// AppDomainIterator::Next が AppDomain::Release を呼び出し、AppDomain を破棄する可能性があり、
```

```

// ~AppDomain はコントラクトによるとトリガーする。
GC_TRIGGER;

// 協調モードが必要。そうでないと objectId が無効になる可能性がある
if (GetThreadNULLOk() != NULL) { MODE_COOPERATIVE; }

// Yay!
EE_THREAD_NOT_REQUIRED;

// Generics::GetExactInstantiationsFromCallInformation が最終的に
// メタデータを読み取り、リーダーロックを取得する。

CAN_TAKE_LOCK;
}

CONTRACTL_END;

```

エントリポイントマクロ

コントラクトの後には、エントリポイントマクロを配置する必要があります。このマクロは、ログの記録と、同期関数の場合はコールバック状態フラグを参照して本当に同期的に呼び出されているかを強制する処理を担います。Info 関数が同期か、非同期か、または Initialize コールバック内からのみ呼び出し可能かに応じて、以下のいずれかを使用してください：

- PROFILER*TO_CLR_ENTRYPOINT_SYNC * (典型的な選択) _
- PROFILER_TO_CLR_ENTRYPOINT_ASYNC
- PROFILER_TO_CLR_ENTRYPOINT_CALLABLE_ON_INIT_ONLY

前述のように、非同期 Info メソッドはまれであり、より高い負担が伴います。上記の推奨コントラクトは非同期メソッドの場合「さらに推奨」であり、以下の 2 つは完全に必須です：GC_NOTRIGGER と MODE_ANY。CANNOT_TAKE_LOCK は、非同期関数では同期関数よりもさらに推奨されますが、常に可能なわけではありません。その場合の対処法については、上記の「非同期」セクションを参照してください。

変更対象のファイル

メソッドの追加や変更のためにどこに行けばよいかは非常に明快であり、コードを見れば分かるでしょう。以下が訪れるべき場所です。

すべてのプロファイリング API インターフェースと型は [src\inc\corprof.idl](#) で定義されています。まずここで型とメソッドを定義してください。

EEToProfInterfaceImpl.*

プロファイルーの ICorProfilerCallback 実装に対するラッパーは [src\vm\EEToProfInterfaceImpl.*](#) にあります。

ProfToEEInterfaceImpl.*

ICorProfilerInfo の実装は [src\vm\ProfToEEInterfaceImpl.*](#) にあります。

すべての開発者がランタイムの例外について知るべきこと

原文

この章の原文は [What Every Dev needs to Know About Exceptions in the Runtime](#) です。

日付: 2005年

CLR における「例外 (Exception)」について話す際、重要な区別を念頭に置く必要があります。マネージド例外 (**managed exceptions**) は、C# の try/catch/finally などのメカニズムを通じてアプリケーションに公開されるもので、それらを実装するためのすべてのランタイム機構を伴います。一方、CLR 内部例外 (**CLR's internal exceptions**) は、ランタイム自体のエラー処理に使用されるものです。ほとんどのランタイム開発者は、マネージド例外モデルの構築と公開方法について考える必要はほとんどありませんが、すべてのランタイム開発者は、ランタイムの実装における例外の使用方法を理解する必要があります。区別を明確にする必要がある場合、この文書ではマネージドアプリケーションがスローまたはキャッチできる「マネージド例外」と、ランタイムが独自のエラー処理のために使用する「CLR 内部例外」を区別して呼びます。ただし、この文書の大部分は CLR 内部例外についてのものです。

💡 初心者向け補足

C# で `try { ... } catch (Exception e) { ... }` と書くとき、これは「マネージド例外」を扱っています。一方、CLR ランタイムの内部 (C++ で書かれた部分) でも独自の例外処理の仕組みがあり、これが「CLR 内部例外」です。Java に例えると、マネージド例外は Java アプリケーションの例外処理に、CLR 内部例外は JVM 実装内部のエラー処理に相当します。この章では主に後者について説明しています。

例外が重要な場面

例外はほぼすべての場所で重要です。例外をスローまたはキャッチする関数では特に重要です。なぜなら、そのコードは例外をスローするか、例外をキャッチして適切に処理するために明示的に記述されなければならないからです。たとえ特定の関数自体が例外をスローしなくても、例外をスローする関数を呼び出す可能性が十分にあるため、例外がスローされた場合に正しく動作するように記述する必要があります。ホルダー (**holders**) を適切に使用することで、このようなコードの正確な記述が大幅に容易になります。

CLR 内部例外が異なる理由

CLR の内部例外は C++ 例外によく似ていますが、完全に同じではありません。CoreCLR は Mac OSX、Linux、BSD、Windows 向けにビルドできます。OS とコンパイラの違いにより、標準的な C++ の try/catch をそのまま使用することはできません。さらに、CLR 内部例外はマネージドの "finally" や "fault" に似た機能を提供します。

いくつかのマクロの助けを借りれば、標準的な C++ とほぼ同じように簡単に記述・読解できる例外処理コードを書くことが可能です。

例外のキャッチ

EX_TRY

基本的なマクロは、もちろん EX_TRY / EX_CATCH / EX_END_CATCH であり、使用時には次のようにになります：

```
EX_TRY  
    // 関数を呼び出す。例外がスローされるかもしれない。  
    Bar();  
EX_CATCH  
    // ここに来たら、何かが失敗した。  
    m_finalDisposition = terminallyHopeless;  
    RethrowTransientExceptions();  
EX_END_CATCH
```

cpp

EX_TRY マクロは、単に try ブロックを導入するもので、C++ の "try" とほぼ同じですが、開き波括弧 "{" も含んでいる点が異なります。

💡 初心者向け補足

CLR のランタイムは C++ で書かれていますが、標準的な C++ 例外をそのまま使えません。代わりに `EX_TRY` / `EX_CATCH` というマクロを使います。これは C# の `try` / `catch` に相当しますが、すべての例外をキャッチするという点が大きく異なります。特定の例外だけを処理したい場合は、キャッチ後に例外を調べて、関係ないものは再スローする必要があります。

EX_CATCH

EX_CATCH マクロは、閉じ波括弧 "}" を含めて try ブロックを終了し、catch ブロックを開始します。EX_TRY と同様に、catch ブロックも開き波括弧で始まります。

そしてここが C++ 例外との大きな違いです：CLR 開発者は何をキャッチするかを指定できません。実際のところ、このマクロのセットは、アクセス違反 (AV) やマネージド例外を含む非 C++ 例外も含め、すべてをキャッチします。特定の例外やサブセットだけをキャッチしたい場合は、キャッチ後に例外を調べ、関連のないものは何でも再スローする必要があります。

繰り返しますが、EX_CATCH マクロはすべてをキャッチします。この振る舞いは、関数が必要とするものではないことが多いです。次の 2 つのセクションでは、キャッチすべきでない例外への対処方法について詳しく説明します。

GET_EXCEPTION() と GET_THROWABLE()

では、CLR 開発者はキャッチされたものが何であるかをどのように発見し、どう対処すべきかをどのように判断するのでしょうか？要件に応じていくつかのオプションがあります。

まず、キャッチされた（C++）例外が何であれ、それはグローバルな `Exception` クラスから派生したクラスのインスタンスとして配信されます。これらの派生クラスの中には、`OutOfMemoryException` のように明らかなものもあります。`EETypeLoadException` のようにやや特定のドメイン向けのものもあります。そして、`CLRException`（任意のマネージド例外を参照する `OBJECTHANDLE` を持つ）や `HRException`（`HRESULT` をラップする）のように、他のシステムの例外のラッパークラスに過ぎないものもあります。元の例外が `Exception` から派生していない場合、マクロはそれを `Exception` の派生クラスにラップします。（これらの例外はすべてシステム提供であり、よく知られたものです。新しい例外クラスは、コア実行エンジンチームの関与なしに追加すべきではありません！）

💡 初心者向け補足

ここでの「Exception クラス」は C# の `System.Exception` ではなく、CLR の C++ 実装内部で定義されている C++ クラスです。CLR 内部では、すべての例外がこの基底クラスから派生する形で統一的に扱われます。たとえば、COM から来た HRESULT エラーも `HRException` としてラップされ、マネージド例外は `CLRException` としてラップされます。

次に、CLR 内部例外には常に HRESULT が関連付けられています。HRException のように値が COM ソースから来る場合もありますが、内部エラーや Win32 API の失敗にも HRESULT があります。

最後に、CLR 内部のほぼすべての例外がマネージドコードに返される可能性があるため、内部例外から対応するマネージド例外へのマッピングが存在します。マネージド例外は必ずしも作成されるわけではありませんが、常にそれを取得する可能性があります。

これらの特徴を踏まえて、CLR 開発者はどのように例外を分類するのでしょうか？

多くの場合、例外を分類するために必要なのは、例外に対応する HRESULT だけであり、これは非常に簡単に取得できます：

```
HRESULT hr = GET_EXCEPTION()->GetHR();
```

cpp

より詳細な情報は、マネージド例外オブジェクト (managed exception object) を通じて最も便利に入手できることが多いです。そして、例外がマネージドコードに返される場合（すぐに返される場合でも、後でキャッシュされる場合でも）、マネージドオブジェクトが当然必要です。例外オブジェクトの取得も同様に簡単です。もちろん、それはマネージド objectref なので、通常のルールがすべて適用されます：

```
OBJECTREF throwable = NULL;
GCProtect_Begin(throwable);
// . .
EX_TRY
    // . . . 例外をスローする可能性のある処理
EX_CATCH
    throwable = GET_THROWABLE();
    RethrowTransientExceptions();
EX_END_CATCH
// . . . throwable を使って何かする
GCProtect_End()
```

cpp

C++ 例外オブジェクトが直接必要になることもあります、これは主に例外実装の内部で使用されます。C++ 例外の正確な型が重要な場合、例外を分類する軽量の RTTI (実行時型情報、Runtime Type Information) 的な関数のセットがあります。例えば：

```
Exception *pEx = GET_EXCEPTION();
if (pEx->IsType(CLRException::GetType())) {/* . . . */}
```

cpp

このコードは、例外が CLRException であるか（または CLRException から派生しているか）を判定します。

RethrowTransientExceptions

上記の例では、"RethrowTransientExceptions" は `EX_CATCH` ブロック内のマクロであり、「例外の処理方針 (exception disposition)」と考えられる 3 つの事前定義マクロの 1 つです。以下がそのマクロとその意味です：

- `RethrowTerminalExceptions` — より適切な名前は "RethrowThreadAbort" であり、まさにそれを行うマクロです。

- **RethrowTransientExceptions** — 「一時的 (transient)」な例外の最もよい定義は、再試行すれば（おそらく異なるコンテキストで）発生しない可能性のある例外です。以下が一時的な例外の一覧です：

- COR_E_THREADABORTED
- COR_E_THREADINTERRUPTED
- COR_E_THREADSTOP
- COR_E_APPDOMAINUNLOADED
- E_OUTOFMEMORY
- HRESULT_FROM_WIN32(ERROR_COMMITMENT_LIMIT)
- HRESULT_FROM_WIN32(ERROR_NOT_ENOUGH_MEMORY)
- (HRESULT)STATUS_NO_MEMORY
- COR_E_STACKOVERFLOW
- MSEEE_ASSEMBLYLOADINPROGRESS

どのマクロを使うか迷った CLR 開発者は、おそらく **RethrowTransientExceptions** を選ぶべきです。

いずれの場合でも、EX_CATCH ブロックを記述する開発者は、どの例外をキャッチすべきかを慎重に考え、それらの例外のみをキャッチする必要があります。そして、マクロはとにかくすべてをキャッチするため、例外をキャッチしない唯一の方法は再スローすることです。

💡 初心者向け補足

「一時的 (transient) な例外」とは、一時的な状況によって引き起こされる例外のことです。たとえば、メモリ不足 (OOM) はその瞬間だけのことかもしれませんし、スレッド中断は特定のタイミングでのみ起こります。これらは再試行すれば成功する可能性があるため、通常は飲み込みます（swallow せず）に再スローするのが適切です。C# の世界でも、`OutOfMemoryException` や `StackOverflowException` を安易にキャッチしてはいけないのと同じ考え方です。

EX_CATCH_HRESULT

例外に対応する HRESULT のみが必要な場合、特にコードが COM からのインターフェースにある場合、EX_CATCH_HRESULT は EX_CATCH ブロック全体を記述するよりも簡潔です。典型的なケースは次のようにになります：

```
HRESULT hr;
EX_TRY
    // コード
EX_CATCH_HRESULT (hr)

return hr;
```

しかし、非常に魅力的ではありますが、常に正しいわけではありません。EX_CATCH_HRESULT はすべての例外をキャッチし、HRESULT を保存し、例外を飲み込みます。したがって、その例外の飲み込みが関数に本当に必要なものでない限り、EX_CATCH_HRESULT は適切ではありません。

EX_RETHROW

上記のとおり、例外マクロはすべての例外をキャッチします。特定の例外だけをキャッチする唯一の方法は、すべてをキャッチし、対象外のものをすべて再スローすることです。したがって、例外がキャッチされ、調べられ、ログに記録されるなどした後に、やはりキャッチすべきでないと判断された場合は、再スローすることができます。EX_RETHROW は同じ例外を再度発生させます。

例外をキャッチしない場合

コードの一部が例外をキャッチする必要はないが、何らかのクリーンアップや補償的なアクション（compensating action）を実行する必要がある場合がよくあります。ホルダー（holders）はこのシナリオにまさに適していることが多いですが、常にそうとは限りません。ホルダーでは不十分な場合のために、CLR には "finally" ブロックの 2 つのバリエーションがあります。

EX_TRY_FOR_FINALLY

コードの終了時に何らかの補償アクションが必要な場合、finally が適切かもしれません。CLR で try/finally を実装するためのマクロのセットがあります：

```
EX_TRY_FOR_FINALLY  
  // コード  
EX_FINALLY  
  // 終了処理またはバックアウトコード  
EX_END_FINALLY
```

cpp

重要：EX_TRY_FOR_FINALLY マクロは C++ EH ではなく SEH（構造化例外処理、Structured Exception Handling）で構築されており、C++ コンパイラは SEH と C++ EH を同じ関数内で混在させることを許可しません。自動デストラクタ（auto-destructor）を持つローカル変数はデストラクタの実行に C++ EH を必要とします。したがって、EX_TRY_FOR_FINALLY を持つ関数では EX_TRY を使用できず、自動デストラクタを持つローカル変数を持つこともできません。

💡 初心者向け補足

SEH（構造化例外処理）は Windows 固有の例外処理メカニズムで、C++ の例外処理（C++ EH）とは異なる仕組みです。C++ コンパイラの制約により、1 つの関数内で SEH と C++ EH を混在させることはできません。そのため、EX_TRY_FOR_FINALLY（SEH ベース）と EX_TRY（C++ EH ベース）を同じ関数に書くことはできません。Java でいえば、try-finally と try-catch で異なる実装メカニズムが使われているようなものです。

EX_HOOK

例外がスローされた場合にのみ補償コードが必要になることがあります。このような場合、EX_HOOK は EX_FINALLY に似ていますが、"hook" 句は例外がある場合にのみ実行されます。例外は "hook" 句の終了時に自動的に再スローされます。

```
EX_TRY  
  // コード  
EX_HOOK  
  // "code" ブロックから例外がエスケープしたときに実行されるコード  
EX_END_HOOK
```

cpp

この構造は、単に EX_CATCH と EX_RETHROW を組み合わせるよりもいくらか優れています。なぜなら、スタックオーバーフロー以外の例外は再スローしますが、スタックオーバーフロー例外をキャッチして（スタックをアンワインドして）から、新しいスタックオーバーフロー例外をスローするからです。

例外のスロー

CLR で例外をスローすることは、一般的に以下を呼び出すことで行います：

```
COMPlusThrow ( < args > )
```

cpp

複数のオーバーロードがありますが、基本的な考え方は例外の「種類 (kind)」を COMPlusThrow に渡すことです。「種類」のリストは [_excep.h](#) に対するマクロのセットによって生成され、さまざまな「種類」には kAmbiguousMatchException、kApplicationException などがあります。追加の引数（オーバーロード用）はリソースと置換テキストを指定します。一般的に、正しい「種類」は、類似のエラーを報告する他のコードを探すことで選択されます。

いくつかの事前定義された便利なバリエーションがあります：

💡 初心者向け補足

`COMPlusThrow` や `COMPlusThrowOOM` などは、C# で言えば `throw new OutOfMemoryException()` に相当する操作です。特に OOM（メモリ不足）や StackOverflow（スタック溢れ）は特別な扱いが必要です。メモリ不足時に新しい例外オブジェクトを割り当てる事はできないため、ランタイムは事前に例外オブジェクトを割り当てておき、それを使い回します。

COMPlusThrowOOM()

`ThrowOutOfMemory()` に委譲し、C++ OOM 例外をスローします。これは事前割り当てされた例外をスローし、メモリ不足の状況でメモリ不足例外をスローしようとする問題を回避します。

この例外のマネージド例外オブジェクトを取得する際、ランタイムはまず新しいマネージドオブジェクトの割り当てを試みます^[1]。それが失敗した場合、事前割り当てされた共有グローバルのメモリ不足例外オブジェクトを返します。

[1] 結局のところ、2GB の配列のリクエストが失敗した場合、単純なオブジェクトなら問題ないかもしれません。

COMPlusThrowHR(HRESULT theBadHR)

`IErrorInfo`などを持っている場合のために、いくつかのオーバーロードがあります。特定の HRESULT に対応する例外の種類を判定するための、驚くほど複雑なコードがあります。

COMPlusThrowWin32() / COMPlusThrowWin32(hr)

基本的に `HRESULT_FROM_WIN32(GetLastError())` をスローします。

COMPlusThrowSO()

スタックオーバーフロー (SO) 例外をスローします。これはハード SO ではなく、処理を続行するとハード SO になる可能性がある場合にスローする例外です。

OOM と同様に、事前割り当てされた C++ SO 例外オブジェクトをスローします。OOM と異なり、マネージドオブジェクトを取得する際には、ランタイムは常に事前割り当てされた共有グローバルのスタックオーバーフロー例外オブジェクトを返します。

COMPlusThrowArgumentNullException()

「引数 foo は null であってはならない」例外をスローするヘルパーです。

COMPlusThrowArgumentOutOfRangeException()

名前の通りです。

COMPlusThrowArgumentException()

無効な引数例外のさらに別のバリエーションです。

COMPlusThrowInvalidCastException(thFrom, thTo)

試みられたキャストの from および to の型の型ハンドル (TypeHandle) が与えられると、ヘルパーはきれいにフォーマットされた例外メッセージを作成します。

EX_THROW

これは低レベルのスロー構造であり、通常のコードでは一般的に必要ありません。多くの COMPlusThrowXXX 関数は内部的に EX_THROW を使用しており、他の特殊な ThrowXXX 関数も同様です。例外メカニズムの複雑な詳細をできるだけカプセル化するためには、EX_THROW の直接使用は最小限に抑えるのが最善です。しかし、上位レベルの Throw 関数がどれも使えない場合は、EX_THROW を使用することは問題ありません。

このマクロは 2 つの引数を取ります：スローする例外の型 (C++ Exception クラスの何らかのサブタイプ) と、例外型のコンストラクタへの引数の括弧付きリストです。

SEH の直接使用

SEH を直接使用することが適切ないいくつかの状況があります。特に、SEH はスタックがアンワインドされる前の第 1 パス (first pass) で何らかの処理が必要な場合に唯一の選択肢です。SEH の __try/__except におけるフィルターコードは、例外を処理するかどうかを決定するだけでなく、何でも行うことができます。デバッガー通知 (debugger notifications) は、第 1 パスの処理が必要になることがある領域です。

フィルターコードは非常に慎重に記述する必要があります。一般的に、フィルターコードは、あらゆるランダムで、おそらく矛盾した状態に対して準備ができていなければなりません。フィルターは第1パスで実行され、デストラクタは第2パスで実行されるため、ホルダーはまだ実行されておらず、状態を復元していません。

PAL_TRY / PAL_EXCEPT, PAL_EXCEPT_FILTER, PAL_FINALLY / PAL_ENDTRY

フィルターが必要な場合、PAL_TRY ファミリーは CLR でフィルターを記述するポータブルな方法です。フィルターは SEH を直接使用するため、同じ関数内の C++ EH とは互換性がなく、関数内にホルダーを置くこともできません。

繰り返しますが、これらの使用はまれであるべきです。

_try / _except, _finally

CLR でこれらを直接使用する正当な理由はありません。

例外と GC モード

`COMPlusThrowXXX()` で例外をスローしても GC モードには影響せず、どのモードでも安全に使用できます。例外が EX_CATCH にアンワインドされて戻る際、スタック上にあったすべてのホルダーがアンワインドされ、リソースが解放され状態がリセットされます。EX_CATCH で実行が再開されるまでに、ホルダーで保護された状態は EX_TRY 時点の状態に復元されています。

遷移

マネージドコード、CLR、COM サーバー、その他のネイティブコードを考慮すると、呼び出し規約、メモリ管理、そしてもちろん例外処理メカニズム間の多くの遷移が可能です。例外に関しては、CLR 開発者にとって幸いなことに、これらの遷移のほとんどはランタイムの完全に外側にあるか、自動的に処理されます。CLR 開発者にとって日常的に関係する遷移は 3 つです。それ以外は上級トピックであり、それらについて知る必要がある人々は、自分がそれを知る必要があることを十分に認識しています！

マネージドコードからランタイムへ

これは "fcall"、"JIT ヘルパー" などです。ランタイムがマネージドコードにエラーを報告する典型的な方法は、マネージド例外を通じてです。したがって、fcall 関数が直接的または間接的にマネージド例外を発生させた場合、それはまったく問題ありません。通常の CLR マネージド例外実装が「正しいこと」を行い、適切なマネージドハンドラーを探します。

一方で、fcall 関数が CLR 内部例外（C++ 例外の 1 つ）をスローする可能性のある処理を行う場合、その例外がマネージドコードに漏れ出すことを許してはなりません。このケースを処理するために、CLR には `UnwindAndContinueHandler` (UACH) があります。これは C++ EH 例外をキャッチし、マネージド例外として再発生させるためのコードセットです。

マネージドコードから呼び出され、C++ EH 例外をスローする可能性のあるすべてのランタイム関数は、スローするコードを `INSTALL_UNWIND_AND_CONTINUE_HANDLER` / `UNINSTALL_UNWIND_AND_CONTINUE_HANDLER` でラップする必要があります。UACH のインストールには無視できないオーバーヘッドがあるため、すべての場所で使用すべきではありません。パフォーマンスが重要なコードで使用されるテクニックの 1 つは、UACH なしで実行し、例外をスローする直前にのみインストールすることです。

C++例外がスローされ、UACHが欠落している場合、典型的な障害はCPFH_RealFirstPassHandlerでの「GC_TRIGGERがGC_NOTRIGGER領域で呼び出された」というコントラクト違反(Contract Violation)になります。これを修正するには、マネージドからランタイムへの遷移を探し、[INSTALL_UNWIND_AND_CONTINUE_HANDLER](#)を確認してください。

💡 初心者向け補足

「遷移(transition)」とは、マネージドコード(C#)とネイティブコード(C++)の間の境界を越えることです。JavaでいえばJNI(Java Native Interface)を介したネイティブコード呼び出しに似ています。CLRでは、C++の内部例外がマネージドコード側に漏れ出さないように、UACH(UnwindAndContinueHandler)という仕組みで境界を守っています。UACHが欠落していると、例外処理が正しく動作せず、デバッグが困難なクラッシュの原因になります。

ランタイムコードからマネージドコードへ

ランタイムからマネージドコードへの遷移には、高度にプラットフォーム依存な要件があります。32ビットWindowsプラットフォームでは、CLRのマネージド例外コードはマネージドコードに入る直前に"COMPlusFrameHandler"がインストールされていることを必要とします。これらの遷移は高度に特殊化されたヘルパー関数によって処理され、適切な例外ハンドラーが設定されます。通常の新しいマネージドコードへの呼び出しが他の方法を使用する可能性は非常に低いです。COMPlusFrameHandlerが欠落していた場合、最も可能性の高い影響は、ターゲットのマネージドコード内の例外処理コードが単に実行されないこと—finallyブロックもcatchブロックも実行されないことです。

ランタイムコードから外部ネイティブコードへ

ランタイムから他のネイティブコード(OS、CRT、その他のDLL)への呼び出しには、特別な注意が必要な場合があります。重要なケースは、外部コードが例外を発生させる可能性がある場合です。これが問題となる理由は、EX_TRYマクロの実装、特に非ExceptionをExceptionに変換またはラップする方法にあります。C++EHでは、あらゆる例外をキャッチすることが可能ですが("catch(...)"を使用して)、キャッチされたものに関する情報をすべて失うことになります。Exception*をキャッチする場合、マクロは例外オブジェクト調べることができますが、それ以外のものをキャッチする場合、調べるものは何もなく、マクロは実際の例外が何であるかを推測しなければなりません。そして、例外がランタイムの外部から来る場合、マクロは常に誤った推測をします。

現在のソリューションは、外部コードへの呼び出しを「コールアウトフィルター(callout filter)」でラップすることです。フィルターは外部例外をキャッチし、ランタイムの内部例外の1つであるSEHExceptionに変換します。このフィルターは事前定義されており、使い方は簡単です。ただし、フィルターを使用するということはSEHを使用するということであり、もちろん同じ関数内でC++EHを使用することはできません。C++EHを使用する関数にコールアウトフィルターを追加するには、関数を2つに分割する必要があります。

コールアウトフィルターを使用するには、次のように書く代わりに：

```
length = SysStringLen(pBSTR);
```

次のように書きます：

```
BOOL OneShot = TRUE;
struct Param {
    BSTR* pBSTR;
    int length;
};
struct Param param;
param.pBSTR = pBSTR;
```

```
PAL_TRY(Param*, pParam, &param)
{
    pParam->length = SysStringLen(pParam->pBSTR);
}
PAL_EXCEPT_FILTER(CallOutFilter, &OneShot)
{
    _ASSERTE(!"CallOutFilter returned EXECUTE_HANDLER.");
}
PAL_ENDTRY;
```

💡 初心者向け補足

外部のネイティブコード（Windows API など）を呼び出す際、そのコードが例外を発生させると、CLR の例外マクロは何がスローされたのかを正しく判別できません。これは `catch(...)` すべてをキャッチしても、例外の型情報が失われるためです。そこで「コールアウトフィルター」というラッパーを使用して、外部例外を CLR が理解できる形（SEHException）に変換します。コードが冗長になりますが、正しい例外情報を保持するために必要です。

例外を発生させる呼び出しにコールアウトフィルターが欠落していると、常にランタイムで誤った例外が報告される結果になります。誤って報告される型は決定的でさえあります。すでにマネージド例外が「飛行中（in flight）」の場合、そのマネージド例外が報告されます。現在の例外がない場合は、OOM が報告されます。チェックビルトでは、コールアウトフィルターの欠落に対して通常アサートが発火します。これらのアサートメッセージには「The runtime may have lost track of the type of an exception（ランタイムが例外の型を見失った可能性があります）」というテキストが含まれます。

その他

EX_TRY には実際に多くのマクロが関わっています。そのほとんどは、マクロ実装の外部では決して使用すべきではありません。

ReadyToRun 概要

原文

この章の原文は [ReadyToRun Overview](#) です。

動機

.NET ランタイムが登場してから、マネージドコードコンポーネントの配布とデプロイに使用できるファイル形式は CLI ファイル形式の1つだけでした。この形式ではすべての実行がマシン非依存の中間言語 (IL) として表現され、コードの実行前にネイティブコードにコンパイルされる必要があります。

💡 初心者向け補足

通常、C# のコードは IL (中間言語) にコンパイルされ、実行時に JIT コンパイラがネイティブコードに変換します。これには起動時間がかかります。ReadyToRun は、あらかじめネイティブコードを含んだファイル形式で、起動を高速化する技術です。

`dotnet publish` で `-p:PublishReadyToRun=true` オプションを指定すると利用できます。

効率的で直接実行可能なファイル形式がないことは、アンマネージドコードと比較して大きな問題となっていました：

- ネイティブコード生成には比較的長い時間がかかり、電力を消費する
- セキュリティ/改ざん防止のために、実行されるネイティブコードを検証する強い要望がある
- 既存のネイティブコード生成戦略は脆弱なコードを生成し、ランタイムやフレームワークが更新されるとすべてのネイティブコードが無効化される

問題の制約

.NET ランタイムには以前からネイティブコードストーリー (NGEN) がありましたが、ここで提案されているものは NGEN とはアーキテクチャ的に異なります。NGEN は基本的にキャッシュ (オプションであり、アプリのパフォーマンスにのみ影響) であるため、イメージの脆弱性は問題になりませんでした。

ReadyToRun のネイティブファイル形式は、ランタイムやフレームワークの更新にもかかわらずファイルが動作し続けるという強い保証を提供します。

💡 初心者向け補足

NGEN (以前の AOT 技術) では、ランタイムが更新されるとコンパイル済みのネイティブイメージが無効になり、再コンパイルが必要でした。ReadyToRun はバージョン耐性を持つよう設計されており、ランタイムが更新されても以前のコンパイル済みコードが引き続き動作します。

ソリューションの概要

CLI ファイル形式を出発点として、以下の方法で拡張します：

- 既存の CLI 形式にネイティブヘッダーを追加し、直接実行に必要な追加情報を含める
- すべての機能を即座にサポート（完全な CLI を含むファイルの場合）
- 最も重要なデータのみを追加し、それ以外は CLI 形式の処理パスを使用

追加情報の最も重要な部分：

- ネイティブコード：メソッドのネイティブコードと外部参照方法
- GC 情報：各メソッドの GC ポインタ情報
- 例外処理テーブル：例外ハンドラーの検索用
- IP マップ：命令ポインタから GC/EH 情報への対応テーブル
- メタデータリンクテーブル：メタデータとネイティブ構造を結びつけるテーブル

バージョン互換性の定義

CIL と同じ互換性ルールを持つことが理想ですが、すべてのケースで効率的に行うことは困難です。最も難しい問題は値型（構造体）とジェネリックメソッドに関するものです。

値型の互換性ルール：公開値型のフィールド（プライベートを含む）の数や型を変更することは破壊的変更です。ただし、非公開（internal）の構造体で、公開値型のフィールドネストから到達できない場合はこの制限は適用されません。

以下の変更は許可されます：

- 参照クラスへのインスタンスフィールドと静的フィールドの追加
- 値クラスへの静的フィールドの追加
- 仮想、インスタンス、静的メソッドの追加
- 既存メソッドの変更（セマンティクスが互換である場合）
- 新しいクラスの追加

バージョンバブル

バージョンバブルとは、セットとして更新されることを前提とした DLL のセットです。バージョニングの観点からは、この DLL のセットは単一のモジュールとして扱われます。バージョンバブル内ではオンライン化やその他のクロスマジュール最適化が許可されます。

 初心者向け補足

例えば、アプリケーションの DLL A と DLL B が常にセットで配布される場合、それらを「バージョンバブル」として定義できます。バブル内では、DLL A のメソッドを DLL B にインライン化するなどの最適化が可能です。一方、バブルの境界を越える場合は、バージョン耐性のためにこのような最適化は制限されます。

重要な原則: バージョンバブルをまたがないメソッドや型のコードは、パフォーマンスペナルティを受けません。

バージョン耐性のあるネイティブコード生成

インスタンスフィールドアクセス

バージョンバブル内の場合、フィールドオフセットはコンパイル時に既知の定数としてコードに埋め込みます：

```
MOV RAX, [RCX + iField_Offset]
```

バージョンバブルをまたぐ場合、基底クラスのサイズが変更される可能性があるため、動的な値を使用します：

```
MOV TMP, [SIZE_OF_BASECLASS]
MOV EAX, [RCX + TMP + subfield_OffsetInSubClass]
```

JIT の選択的使用

バージョン耐性に関するコード品質のペナルティを回避するために、JIT コンパイラの選択的使用も可能です。例えば、バージョンバブルをまたぐインライン化候補がある場合、該当メソッドをランタイムコンパイル対象として属性で指定することで、JIT が自由に最適化を適用できます。

💡 初心者向け補足

ReadyToRun は「事前コンパイル」と「JIT コンパイル」のハイブリッドアプローチを取ることができます。ほとんどのコードは事前コンパイルされますが、パフォーマンスが重要な一部のメソッドは JIT に任せることで、バージョン耐性と最高のパフォーマンスの両方を実現できます。

📖 この章はまだ翻訳途中です。[翻訳に貢献する](#)

CLR ABI

原文

この章の原文は [CLR ABI](#) です。

本ドキュメントでは、.NET 共通言語ランタイム (Common Language Runtime, CLR) のソフトウェア規約 (ABI、「Application Binary Interface」) について説明します。x64 (AMD64)、ARM (ARM32 または Thumb-2)、および ARM64 プロセッサーアーキテクチャの ABI に焦点を当てています。x86 の ABI に関するドキュメントはやや少ないですが、呼び出し規約 (calling convention) の基本に関する情報は本ドキュメントの末尾に含まれています。

初心者向け補足

ABI (Application Binary Interface) とは、コンパイルされたプログラム同士が連携するための「契約」のようなものです。関数の呼び出し方、引数の渡し方、戻り値の受け取り方、レジスタの使い方、スタックメモリの管理方法などを定めたルールです。たとえば Java では JVM がこうした詳細を隠蔽しますが、CLR のような低レベルのランタイムでは、JIT コンパイラが生成するネイティブコードとランタイム (VM) の間でこれらの約束事を正確に守る必要があります。ABI が異なると、同じ CPU 上であっても関数を正しく呼び出すことができません。

本ドキュメントでは、JIT (Just-In-Time) コンパイラが VM に課す要件、およびその逆について説明します。

JIT コードベースに関する補足: JIT32 は、もともと x86 コードを生成し、その後 ARM コード生成に移植されたオリジナルの JIT コードベースを指します。JIT64 は、AMD64 をサポートするレガシーな .NET Framework のコードベースを指します。RyuJIT コンパイラは JIT32 から発展し、現在はすべてのプラットフォームとアーキテクチャをサポートしています。RyuJIT の歴史の詳細については [こちらの記事](#) を参照してください。

NativeAOT は、事前コンパイル (AOT: Ahead-Of-Time compilation) に最適化されたランタイムを指します。NativeAOT の ABI は、プラットフォーム間での簡潔性と一貫性のために、いくつかの詳細が異なります。

はじめに

Windows および非 Windows の ABI ドキュメントに記載されている内容をすべて読んでください。CLR はそれらの基本規約に従います。本ドキュメントでは、CLR 固有の事項、またはそれらのドキュメントからの例外のみを説明します。

Windows の ABI ドキュメント

AMD64: [x64 Software Conventions](#) を参照してください。

ARM: [Overview of ARM32 ABI Conventions](#) を参照してください。

ARM64: [Overview of ARM64 ABI conventions](#) を参照してください。

非 Windows の ABI ドキュメント

Arm 社の ABI ドキュメント (ARM32 および ARM64 向け) は[こちら](#)と[こちら](#)にあります。Apple の ARM64 呼び出し規約の差異については[こちら](#)を参照してください。

Linux System V x86_64 ABI は[System V Application Binary Interface / AMD64 Architecture Processor Supplement](#) に文書化されており、ドキュメントのソース資料は[こちら](#)にあります。

LoongArch64 の ABI ドキュメントは[こちら](#)にあります。

RISC-V ABI 仕様: [最新リリース](#)、[最新ドラフト](#)、[ドキュメントソースリポジトリ](#)。

Web Assembly Basic C ABI: [Basic C ABI](#)

一般的なアンワインド/フレームレイアウト

💡 初心者向け補足

「アンワインド (unwind)」とは、スタックフレーム (stack frame) を巻き戻す操作のことです。関数を呼び出すたびにスタック上に「フレーム」と呼ばれる領域が積み重なります。例外が発生したり、ガベージコレクション (GC) が実行されたりする際、ランタイムは現在のスタック状態を理解するためにフレームを一つずつ辿って戻る必要があります。この巻き戻し操作をアンワインドと呼びます。Java での例外発生時にスタックトレースが表示されるのと同様に、.NET ランタイムもスタックを正確にたどれる必要があります。

x86 以外のすべてのプラットフォームでは、ガベージコレクタ (GC) がアンワインドできるように、すべてのメソッドがアンワインド情報を持つ必要があります (ネイティブコードでは、リーフメソッドのアンワインド情報は省略可能です)。

ARM および ARM64: マネージドメソッドは、リターンアドレスハイジャック (return address hijacking) によってメソッドを適切にハイジャックできるように、常に LR をスタックにプッシュし、最小限のフレームを作成する必要があります。

フレームポインタチェーン (Frame pointer chains)

フレームポインタチェーン (frame pointer chain) とは、フレームポインタレジスター (frame pointer register) がスタック上の位置を指し、その位置に保存された前のフレームポインタ値 (現在の関数の呼び出し元のもの) のアドレスが格納されている状態を指します。このチェーンは、以下のような特定のシナリオで必要です：

1. Linux 上の gdb デバッガによるスタックウォーキング。
2. ETW イベントトレースによるスタックウォーキング。

考慮すべき点が 2 つあります：

1. フレームポインタレジスターをスタックウォーキング用に予約し、汎用的なコード生成など他の目的に使用しないこと。
2. フレームチェーンを作成すること。

関数がフレームチェーンに追加されていなくても、その関数がフレームポインタを変更しない限り、既存のフレームチェーンは引き続き有効です。ただし、チェーンをたどる際にその関数は表示されません。JIT は、フレームチェーンが作成されていなくても、例外処理ファンクレット (funclet) 内からメインの関数ローカル変数にアクセスするためなどの理由で、フレームポインタレジスタを作成・使用する場合があります。

フレームポインタレジスタは、各アーキテクチャで以下のとおりです: ARM: r11、ARM64: x29、x86: EBP、x64: RBP。

JIT は、Windows x64 以外のすべてのプラットフォームでほとんどの場合フレームチェーンを作成します。非常に単純な関数は、フレームセットアップコストを削減してパフォーマンスを向上させる目的で、フレームチェーンに追加されない場合があります (この選択のヒューリスティクスは `Compiler::rpMustCreateEBPFrame()` にあります)。Windows x64 では、アンワインドは常に生成されたアンワインドコードを使用して行われ、単純なフレームチェーンのトラバーサルは使用されません。

追加リンク:

- ARM64 のフレーム設計に関するドキュメントは [ARM64 JIT frame layout](#) を参照してください。
- Unix x64 で常に RBP チェーンを作成するようにした CoreCLR の変更是 [こちら](#) です (議論を含む Issue は [こちら](#))。

特殊パラメータ/追加パラメータ

初心者向け補足

呼び出し規約 (calling convention) とは、関数がどのように引数を受け取り、戻り値を返すかを定義するルールのことです。たとえば、「最初の引数はレジスタ RCX に入る」「戻り値は RAX に入る」といったルールです。CLR はネイティブ (OS やハードウェアが定める) ABI の規約に加えて、ガベージコレクション (GC) やジェネリクスなどをサポートするために、いくつかの 特殊なパラメータを追加しています。この章では、そうした CLR 独自の追加パラメータについて説明します。

`this` ポインタ

マネージド `this` ポインタは、ネイティブ ABI ではカバーされない新しい種類の引数として扱われます。そのため、常に最初の引数として (AMD64) `RCX` または (ARM, ARM64) `R0` で渡すことになります。

AMD64 のみ: .NET Framework 4.5 までは、マネージド `this` ポインタはネイティブの `this` ポインタと同じように扱われていました (つまり、呼び出しがリターンバッファ (return buffer) を使用する場合は第 2 引数となり、RCX ではなく RDX で渡されました)。.NET Framework 4.5 以降は、常に最初の引数として渡されます。

可変長引数 (Varargs)

可変長引数 (Varargs) とは、関数呼び出しにおいて可変個の引数を渡したり受け取ったりすることを指します。

初心者向け補足

可変長引数 (varargs) とは、関数に渡す引数の数が固定されていない仕組みのことです。C 言語の `printf` 関数のように、呼び出すたびに異なる数の引数を渡せます。C# では `params` キーワードを使った可変長引数がよく使われますが、これは内部的には

固定長の引数と同じ扱いです。一方、ここで説明するマネージド可変長引数 (`_arglist` など) は、よりローレベルな仕組みで、C++ の可変長引数に近い動作をします。

C# の可変長引数は `params` キーワードを使用しますが、IL レベルでは固定数のパラメータを持つ通常の呼び出しにすぎません。

マネージド可変長引数 (C# の疑似ドキュメント化された `...`、`_arglist` などを使用) は、C++ の可変長引数とほぼ同じように実装されています。最大の違いは、JIT がオプションのリターンバッファとオプションの `this` ポインタの後、かつ他のユーザー引数の前に「vararg クッキー (vararg cookie)」を追加することです。呼び出し先 (callee) は、このクッキーとそれ以降のすべての引数をホームロケーション (home location) にスpill (spill) しなければなりません。これらはクッキーをベースとしたポインタ演算でアドレスされる可能性があるためです。このクッキーは、ランタイムがパースできるシグネチャへのポインタであり、(1) 引数の可変部分に含まれる GC ポインタを報告する、または (2) ArgIterator を通じて取り出された引数の型チェック (および適切なウォークオーバー) を行うために使用されます。これは `IMAGE_CEE_CS_CALLCONV_VARARG` で示されますが、`IMAGE_CEE_CS_CALLCONV_NATIVEVARARG` と混同しないでください。後者はまさにネイティブの可変長引数そのもの (クッキーなし) であり、PInvoke IL スタブでのみ使用されるべきもので、ピニング留め (pinning) やその他の GC マジック (GC magic) を適切に処理します。

AMD64 では、ネイティブと同様に、浮動小数点レジスタで渡される浮動小数点引数 (固定引数を含む) は、整数レジスタにシャドウ (つまり複製) されます。

ARM および ARM64 では、ネイティブと同様に、浮動小数点レジスタには何も配置されません。

ただし、ネイティブの可変長引数とは異なり、すべての浮動小数点引数が `double` (`R8`) に昇格されるわけではなく、元の型 (`R4` または `R8`) が維持されます (ただし、managed C++ のような IL ジェネレータが呼び出し元でアップキャストを明示的に挿入し、呼び出し元のシグネチャを適切に調整することを妨げるものではありません)。これにより、ネイティブ C++ を C# に移植したり、managed C++ のさまざまなバリエントで管理したりする場合に、予期しない動作が生じることがあります。

マネージド可変長引数は Windows でのみサポートされています。

マネージド/ネイティブ可変長引数は Windows でのみサポートされています。非 Windows プラットフォームでのマネージド/ネイティブ可変長引数のサポートは [この Issue](#) で追跡されています。

ジェネリクス (Generics)

共有ジェネリクス (Shared generics)。コードアドレスがメソッドのジェネリックインスタンス化 (generic instantiation) を一意に識別できない場合、「ジェネリックインスタンス化パラメータ (generic instantiation parameter)」が必要になります。多くの場合、`this` ポインタがインスタンス化パラメータとしても兼用できます。`this` ポインタがジェネリックパラメータでない場合、ジェネリックパラメータは追加の引数として渡されます。ARM と AMD64 では、オプションのリターンバッファとオプションの `this` ポインタの後、かつユーザー引数の前に渡されます。ARM64 と RISC-V では、ジェネリックパラメータはオプションの `this` ポインタの後、かつユーザー引数の前に渡されます。x86 では、`this` ポインタを含むすべての関数の引数が引数レジスタ (ECX と EDX) に収まり、かつまだ使用可能な引数レジスタがある場合、隠し引数は次に使用可能な引数レジスタに格納されます。それ以外の場合は、最後のスタック引数として渡されます。ジェネリックメソッド (型ではなくメソッド自体に型パラメータがある場合) では、ジェネリックパラメータは現在 `MethodDesc` ポインタ (おそらく `InstantiatedMethodDesc`) です。静的メソッド (`this` ポインタがない場合) では、ジェネリックパラメータは `MethodTable` ポインタ/`TypeHandle` です。

💡 初心者向け補足

共有ジェネリクスとは、`List<int>` と `List<string>` のように異なる型引数で使われるジェネリック型やメソッドが、可能な場合には同じコンパイル済みコードを共有する最適化のことです。しかし、共有コードは実行時にどの型引数で呼ばれているかを知る必要があるため、「ジェネリックインスタンス化パラメータ」という隠しパラメータを追加で渡します。たとえば、インスタンス

メソッドの場合は `this` ポインタから型情報を取得できますが、静的メソッドでは `this` がないため、MethodTableへのポイントを別途渡す必要があります。

VMがJITにジェネリクスパラメータの報告と存続を要求する場合があります。この場合、パラメータはスタック上のどこかに保存し、プロローグ(prolog)とエピローグ(epilog)を除くメソッド全体にわたって通常のGC報告で存続させなければなりません(MethodDescやMethodTableではなく `this` ポインタだった場合)。また、パラメータをホームに配置するコードは、GC情報でプロローグとして報告されるコード範囲内になければなりません(これはアンワインド情報(unwind info)でプロローグとして報告される範囲とは異なる可能性があります)。

ジェネリックパラメータと可変長引数クッキーの間に定義された/強制された/宣言された順序はありません。ランタイムがその組み合わせをサポートしていないためです。VMとJITの中にはその組み合わせをサポートしているように見えるコードがいくつかありますが、別の場所ではアサートして禁止しているため、何もテストされておらず、バグや相違(たとえば、あるJITが別のJITやVMとは異なる順序を使用している、など)があるものと推測されます。

例

```
call(["this" pointer] [return buffer pointer] [generics context|varargs cookie] [userargs]*)
```

非同期(Async)

非同期呼び出し規約(Async calling convention)は、サポートされている場合に他の呼び出し規約に対して追加的に適用されます。対象となるシナリオは通常の静的/仮想呼び出しに限定されており、たとえばPInvokeや可変長引数はサポートしません。最低限、通常の静的呼び出し、`this`パラメータ付きの呼び出し、ジェネリック隠しパラメータ付きの呼び出しがサポートされます。

非同期呼び出し規約は、追加の `Continuation` パラメータと追加の戻り値を加えます。この戻り値は `null` でない場合に意味的に優先されます。戻り時に `Continuation` が非 `null` であることは、計算がまだ完了しておらず正式な結果が準備できていないことを示します。引数として非 `null` が渡されること、関数が再開中であり、`Continuation` から状態を復元して(他のすべての引数を無視して)実行を継続すべきであることを意味します。

`Continuation` はマネージドオブジェクトであり、それに応じて追跡する必要があります。GC情報には、非同期呼び出しサイトにおいて継続結果がライブとして含まれます。

`Continuation` の返却

`Continuation` を返すには、実際の結果を返すために使用できない volatile/callee-trash レジスタを使用します。

アーキテクチャ	REG_ASYNC_CONTINUATION_RET
x86	ecx
x64	rcx
arm	r2

アーキテクチャ	REG_ASYNC_CONTINUATION_RET
arm64	x2
risc-v	a2

Continuation 引数の受け渡し

Continuation パラメータは、ジェネリックインスタンス化パラメータと同じ位置、または両方が存在する場合はその直後に渡されます。x86 では引数の順序が逆になります。

```
call(["this" pointer] [return buffer pointer] [generics context] [continuation] [userargs]) // x86 以外
call(["this" pointer] [return buffer pointer] [userargs] [continuation] [generics context]) // x86
```

AMD64 のみ：値渡しの値型

ネイティブと同様に、AMD64 には暗黙的な参照渡し (implicit-byrefs) があります。サイズが 1、2、4、8 バイトでない（つまり 3、5、6、7、または 9 バイト以上の）構造体 (IL 用語では値型 (value type)) が値渡しで宣言されている場合、代わりに参照渡しされます。JIT が生成するコードの場合は、ネイティブ ABI に従い、渡される参照はスタック上のコンパイラが生成した一時ローカルへのポインタです。ただし、リモーティングやリフレクション内で stackalloc が困難すぎるケースがあり、GC ヒープ内のポインタを渡す場合があります。そのため、JIT されたコードは、呼び出し先がこれらのリフレクションパスのいずれかである場合に備えて、暗黙的な byref パラメータをインテリアポインタ (interior pointer) (JIT 用語では BYREF) として報告しなければなりません。同様に、すべての書き込みにはチェック付きライトバリア (checked write barrier) を使用する必要があります。

AMD64 のネイティブ呼び出し規約 (Windows 64 および System V) では、リターンバッファのアドレスを呼び出し先 (callee) が RAX で返すことを要求しています。JIT もこのルールに従います。

RISC-V のみ：ハードウェア浮動小数点呼び出し規約に従った構造体の受け渡し/返却

ネイティブと同様のハードウェア浮動小数点呼び出し規約に従った構造体の受け渡し/返却は、現在 [16 バイトまでのサポート](#) されています。それより大きい構造体は標準 ABI と異なり、整数呼び出し規約（暗黙的な参照渡し）に従って受け渡し/返却されます。

リターンバッファ (Return buffers)

.NET 10 以降、リターンバッファは常に呼び出し元 (caller) がスタック上に確保する必要があります。呼び出しの後、呼び出し元は必要に応じてライトバリア (write barrier) を使用してリターンバッファを最終的な送り先にコピーする責任があります。JIT はリターンバッファが常にスタック上にあると想定でき、GC ポインタをリターンバッファに書き込む際のライトバリアの省略など、それに応じた最適化を行

えます。さらに、バッファの内容はエイリアスされたりクロススレッドから可視であったりしてはならないため、メソッド内での一時ストレージとしての使用が許可されています。

ARM64 のみ：メソッドが 16 バイトより大きい構造体を返す場合、呼び出し元は結果を保持するのに十分なサイズとアラインメントを持つリターンバッファを確保します。バッファのアドレスは `R8` (JIT では `REG_ARG_RET_BUFF` として定義) でメソッドの引数として渡されます。呼び出し先は `R8` に格納された値を保持する必要はありません。

隠しパラメータ (Hidden parameters)

スタブディスパッチ (Stub dispatch) - 仮想呼び出しが VSD スタブを使用する場合、呼び出しコードをバックパッチ（または逆アセンブル）する代わりに、JIT は呼び出しターゲットをロードするために使用されるスタブのアドレス、すなわち「スタブ間接セル (stub indirection cell)」を (x86) `EAX` / (AMD64) `R11` / (ARM) `R4` / (ARM NativeAOT ABI) `R12` / (ARM64) `R11` に配置しなければなりません。JIT では、これは `VirtualStubParamInfo` クラスにカプセル化されています。

`Call PInvoke` - VM は `PInvoke` のアドレスを (AMD64) `R10` / (ARM) `R12` / (ARM64) `R14` (JIT では `REG_PINVOKE_TARGET_PARAM`) に、シグネチャ (`PInvoke` クッキー) を (AMD64) `R11` / (ARM) `R4` / (ARM64) `R15` (JIT では `REG_PINVOKE_COOKIE_PARAM`) に必要とします。

通常の `PInvoke` - VM はシグネチャに基づいて IL スタブを共有しますが、コールスタックと例外に正しいメソッドを表示させたいため、正確な `PInvoke` の `MethodDesc` が (x86) `EAX` / (AMD64) `R10` / (ARM, ARM64) `R12` (JIT では `REG_SECRET_STUB_PARAM`) で渡されます。その後、IL スタブ内で JIT が `CORJIT_FLG_PUBLISH_SECRET_PARAM` を受け取ると、レジスタをコンパイラの一時変数に移動しなければなりません。この値はイントリニシック (intrinsic) `NI_System_StubHelpers_GetStubContext` で返されます。

小さなプリミティブ型の戻り値

32 ビット未満のプリミティブ型は 32 ビットに拡張されます。符号付き小型は符号拡張 (sign extend) され、符号なし小型はゼロ拡張 (zero extend) されます。これは、リターンレジスタの未使用ビットの状態を未定義のままにする標準呼び出し規約とは異なる場合があります。

小さなプリミティブ型の引数

小さなプリミティブ型の引数は、上位ビットが未定義です。これは、正規化 (normalization) を要求する標準呼び出し規約 (例 : ARM32 や Apple ARM64) とは異なる場合があります。

RISC-V では、小さなプリミティブ型の引数は標準呼び出し規約に従って拡張されます。

PInvoke

💡 初心者向け補足

PInvoke (Platform Invoke) とは、マネージド .NET コードからネイティブ（アンマネージド）コード（C/C++ の DLL など）を呼び出すための仕組みです。Java における JNI (Java Native Interface) に相当するものです。PInvoke を使うと、OS の API やサードパーティのネイティブライブラリをマネージドコードから直接呼び出すことができます。

規約として、InlinedCallFrame を持つメソッド（IL スタブまたはインライン PInvoke を含む通常のメソッド）は、プロローグ/エピローグでそれぞれすべての不揮発性 (non-volatile) 整数レジスタの保存/復元を行います。これにより、InlinedCallFrame にはリターンアドレス、スタックポインタ、フレームポインタだけを格納すればよくなります。そしてこれら3つの値だけを使って、通常の RtlVirtualUnwind によるフルスタックウォークを開始できます。

JIT は PInvoke に遭遇すると、GC 遷移 (GC transition) を抑制すべきかどうかを VM に問い合わせます。GC 遷移の抑制は、PInvoke 定義に属性を追加することで示されます。VM が GC 遷移を抑制すべきと指示した場合、IL スタブまたはインラインのどちらのシナリオでも PInvoke フレームは省略され、アンマネージド呼び出しサイトの近くに GC ポール (GC Poll) が挿入されます。囲んでいる関数に複数のインライン PInvoke が含まれていても、すべてが GC 遷移の抑制を要求しているわけではない場合、他のインライン PInvoke に対しては引き続き PInvoke フレームが構築されます。

AMD64 では、InlinedCallFrame を持つメソッドはフレームレジスタとして RBP を使用しなければなりません。

ARM および ARM64 では、常にフレームポインタ (R11) を使用します。これは部分的にはフレームチェーンの要件によるものですが、VM も InlinedCallFrame を持つ PInvoke に対してこれを要求しています。

ARM では、VM は `REG_SAVED_LOCALLOC_SP` にも依存しています。

これらの依存関係はすべて `InlinedCallFrame::UpdateRegDisplay` の実装に現れています。

JIT32 は、現在のメソッドに PInvoke/InlinedCallFrame がある場合、エピローグを1つだけ生成し（すべての return をそこへ分岐させ）ます。

フレームごとの PInvoke 初期化

InlinedCallFrame は、IL スタブの先頭で1回、およびインライン PInvoke を行う各パスで1回初期化されます。

JIT64 では、実際に呼び出しを含むブロックで初期化が行われますが、ランディングパッドを持つループの外に押し出し、その後ドミネーターブロックを探します。IL スタブおよび EH を持つメソッドの場合は、諦めて最初のブロックに初期化を配置します。

RyuJIT/JIT32 (ARM) では、すべてのメソッドが JIT64 の IL スタブと同様に扱われます（つまり、フレームごとの初期化はプロローグの直後に1回だけ行われます）。

JIT は、InlinedCallFrame のアドレスと、NULL または IL スタブ用のシークレットパラメータを渡して `CORINFO_HELP_INIT_PINVOKE_FRAME` を呼び出すコードを生成します。`JIT_InitPInvokeFrame` は InlinedCallFrame を初期化し、現在の Frame チェーンの先頭を指すように設定します。そして、現在のスレッドのネイティブ Thread オブジェクトを返します。

AMD64 では、JIT は RSP と RBP を InlinedCallFrame に保存するコードを生成します。

IL スタブの場合のみ、フレームごとの初期化には `Thread->m_pFrame` を InlinedCallFrame に設定すること（事実上 Frame を「プッシュ」すること）が含まれます。

呼び出しサイトごとの PInvoke 作業

以下は、GC 遷移が抑制されていない場合に実行されます。

1. 直接呼び出しの場合、JIT 生成コードは `InlinedCallFrame->m_pDatum` を呼び出し先の MethodDesc に設定します。
 - JIT64 では、IL スタブ内の間接呼び出しはシークレットパラメータに設定します（これは冗長に見えますが、フレームごとの初期化以降に変更された可能性があります）。
 - JIT32 (ARM) の間接呼び出しでは、コメントによるとプッシュされた引数のサイズをこのメンバーに設定します。ただし、実装では常に 0 が渡されていました。
2. JIT64/AMD64 のみ: 次に非 IL スタブの場合、`Thread->m_pFrame` を InlinedCallFrame を指すように設定することで InlinedCallFrame を「プッシュ」します（フレームごとの初期化で既に `InlinedCallFrame->m_pNext` が前の先頭を指すように設定されていることを思い出してください）。IL スタブの場合、この手順はフレームごとの初期化で完了しています。
3. `InlinedCallFrame->m_pCallerReturnAddress` を設定することで Frame をアクティブにします。
4. 次に、`Thread->m_fPreemptiveGCDisabled = 0` を設定することで GC モードを切り替えます。
5. ここから先、GC ポインタがレジスタ内で生存してはなりません。RyuJIT の LSRA は、アンマネージド呼び出しおよび特殊ヘルパーの前に特殊な refPosition `RefTypeKillGCRefs` を追加することでこの要件を満たします。
6. ここで実際の呼び出し/PInvoke が行われます。
7. `Thread->m_fPreemptiveGCDisabled = 1` を設定することで GC モードを元に戻します。
8. 次に、`g_TrapReturningThreads` が設定されている（非ゼロ）かどうかを確認します。設定されている場合、`CORINFO_HELP_STOP_FOR_GC` を呼び出します。
 - ARM では、このヘルパー呼び出しはリターンレジスタ `R0`、`R1`、`S0`、`D0` を保持します。
 - AMD64 では、生成されたコードは PInvoke の戻り値を不揮発性レジスタまたはスタック上の場所に移動して手動で保持する必要があります。
9. ここから先、GC ポインタは再びレジスタ内で生存することが許されます。
10. `InlinedCallFrame->m_pCallerReturnAddress` を 0 にクリアします。
11. JIT64/AMD64 のみ: 非 IL スタブの場合、`Thread->m_pFrame` を `InlinedCallFrame.m_pNext` に戻すことで Frame チェーンを「ポップ」します。

すべての不揮発性レジスタの保存/復元は、現在のフレームで未使用的レジスタが親フレームからの生存中の GC ポインタ値を誤って保持してしまうことを防ぐのに役立ちます。引数レジスタとリターンレジスタは「安全」です。なぜなら、それらは GC 参照にはなり得ないからです。参照は他の場所でピン留めされ、代わりにネイティブポインタとして渡されるべきです。

IL スタブの場合、Frame チェーンは呼び出しサイトでポップされないため、代わりにエピローグの直前および jmp 呼び出しの直前にポップする必要があります。PInvoke IL スタブからのテールコールはサポートされていないようです。

例外処理

このセクションでは、マネージド例外処理 (EH) を実装するコードを生成する際に JIT が従うべき規約について説明します。正しい実装のために、JIT と VM はこれらの規約について合意していなければなりません。

ファンクレット (funclet)

💡 初心者向け補足

ファンクレット (funclet) とは、例外ハンドラ (catch、finally、fault、filter など) から抽出された独立したコード片です。OS からは独立した関数として扱われますが、CLR 上では親関数の一部として管理されます。たとえば、`try { ... } catch { ... }` `} finally { ... }` のような構文がある場合、catch ブロックと finally ブロックの中身がそれぞれ別のファンクレットとして抽出されます。

すべてのプラットフォームにおいて、マネージド EH ハンドラ (finally、fault、filter、filter-handler、catch) は独自の「ファンクレット」に抽出されます。OS にとっては、これらは第一級の関数として扱われます（個別の PDATA と XDATA (`RUNTIME_FUNCTION` エントリ) など）。CLR は現在、多くの点でこれらを親関数の一部として扱っています。メイン関数とすべてのファンクレットは、単一のコード割り当て内に配置されなければなりません（ホット/コールド分割を参照）。これらは GC 情報を「共有」します。ホットパッチできるのはメイン関数のプロローグのみです。

ハンドラのファンクレットに入る唯一の方法は呼び出しを通じてです。例外の場合、呼び出しが例外ディスパッチ/アンワインドの一環として VM の EH サブシステムから行われます。例外でない場合、これはローカルアンワインド (local unwind) または非ローカル終了 (non-local exit) と呼ばれます。C# では、try 本体からフォールスルーする/出る、または明示的な goto によって実現されます。IL では、try 本体内の LEAVE オペコードで、try 本体外の IL オフセットをターゲットにすることで常に実現されます。このような場合、呼び出しが親関数の JIT 生成コードから行われます。

クローン化された finally

RyuJIT は、「通常の」制御フロー（C# のフォールスルーで非例外的に try 本体を離れる場合）に沿って呼び出される finally を「インライン化」することで、通常の制御フローを高速化しようとしています。この最適化はすべてのアーキテクチャでサポートされています。

finally の呼び出し / 非ローカル終了

適切な前進保証 (forward progress) と `Thread.Abort` のセマンティクスのために、call-to-finally を配置できる場所と、呼び出しサイトの見た目に制約があります。リターンアドレスは、対応する try 本体内にあってはなりません（さもないと VM は finally が自身を保護していると考えてしまいます）。リターンアドレスは、外側の保護領域内になければなりません（finally 本体からの例外が正しく処理されるように）。

RyuJIT はジャンプアイランドに似た仕組みを作成します。try 本体の外にあるコードブロックが finally を呼び出し、その後 leave/非ローカル終了の最終ターゲットに分岐します。このジャンプアイランドは、EH テーブルではクローン化された finally であるかのようにマークされます。クローン化された finally 句により、ハンドラに入る前に Thread.Abort が発火することを防ぎます。リターンアドレスを try 本体の外に配置することで、もう1つの制約も満たします。

ThreadAbortException の考慮事項

初心者向け補足

ThreadAbortException は、あるスレッドから別のスレッドを中断 (abort) できるようにする .NET 固有のメカニズムです。デスクトップ .NET Framework では利用可能で、ASP.NET などで多用されていましたが、.NET Core / 現在の .NET ではサポートされていません。それにもかかわらず、JIT は互換性のためにすべてのプラットフォームで ThreadAbort 対応コードを生成します。

スレッドアボートには3種類あります: (1) ルードスレッドアボート (rude thread abort) — 停止できず、(すべての?) ハンドラを実行しないもの、(2) `Thread.Abort()` API の呼び出し、(3) 別のスレッドから注入される非同期スレッドアボートです。

ThreadAbortException はデスクトップフレームワークでは完全に利用可能であり、たとえば ASP.NET で頻繁に使用されていました。しかし、.NET Core、CoreCLR、Windows 8 の「モダンアプリプロファイル」ではサポートされません。それにもかかわらず、JIT はすべてのプラットフォームで ThreadAbort 互換のコードを生成します。

非ルードスレッドアボートの場合、VM はスタックをウォークし、ThreadAbortException をキャッチする catch ハンドラ（または `System.Exception` や `System.Object` などの親クラス）を実行し、finally を実行します。ThreadAbortException には非常に特殊な特性が1つあります。catch ハンドラが `ThreadAbortException` をキャッチし、`Thread.ResetAbort()` を呼び出さずに例外処理から戻った場合、VM は `ThreadAbortException` を自動的に再発生させます。そのために、catch ハンドラが返したレジュームアドレスを、再発生が発生したと見なされる実効アドレスとして使用します。これは catch ハンドラ内の LEAVE オペコードで指定されたラベルのアドレスです。JIT は合成的な「ステップブロック」を挿入して、このラベルが適切な外側の "try" リージョン内に収まるようにし、再発生が外側の catch ハンドラによってキャッチされるようにしなければならない場合があります。

例:

```
try { // try 1
    try { // try 2
        System.Threading.Thread.CurrentThread.Abort();
    } catch (System.Threading.ThreadAbortException) { // catch 2
        ...
        LEAVE L;
    }
} catch (System.Exception) { // catch 1
    ...
}
```

この場合、catch 2 で返されたラベル L に対応するアドレスが try 1 の外にあると、VM によって再発生された `ThreadAbortException` は catch 1 にキャッチされません（期待通り）。JIT は効果的に以下のようなコード生成になるようにブロックを挿入する必要があります:

```
try { // try 1
    try { // try 2
        System.Threading.Thread.CurrentThread.Abort();
    } catch (System.Threading.ThreadAbortException) { // catch 2
        ...
        LEAVE L';
    }
    L': LEAVE L;
} catch (System.Exception) { // catch 1
    ...
}
```

同様に、ThreadAbortException の自動再発生アドレスは finally ハンドラ内にあってはなりません。さもないと VM は再発生を中止し、例外を飲み込んでしまいます。これは、上述のように「クローン化された finally」としてマークされた call-to-finally サンクにより発生する可能性があります。例（これは擬似アセンブリコードであり、C# ではありません）：

```
try { // try 1
    try { // try 2
        System.Threading.Thread.CurrentThread.Abort();
    } catch (System.Threading.ThreadAbortException) { // catch 2
        ...
        LEAVE L;
    }
} finally { // finally 1
    ...
}
L:
```

これは以下のようなコードを生成します：

```
// beginning of 'try 1'
// beginning of 'try 2'
System.Threading.Thread.CurrentThread.Abort();
// end of 'try 2'
// beginning of call-to-finally 'cloned finally' region
L1: call finally1
nop
// end of call-to-finally 'cloned finally' region
// end of 'try 1'
// function epilog
ret

Catch2:
// do something
lea rax, &L1; // load up resume address
ret

Finally1:
// do something
ret
```

JIT は finally が呼び出されるように「ステップ」ブロックを挿入する必要がありますが、これだけでは ThreadAbortException の処理をサポートするには不十分です。なぜなら "L1" が「クローン化された finally」としてマークされているからです。この場合、JIT は "try 1" 内かつクローン化された finally ブロックの外にある別のステップブロックを挿入して、正しい再発生セマンティクスを可能にする必要があります。例：

```
// beginning of 'try 1'
// beginning of 'try 2'
System.Threading.Thread.CurrentThread.Abort();
// end of 'try 2'
L1':    nop
// beginning of call-to-finally 'cloned finally' region
L1: call finally1
nop
```

```

// end of call-to-finally 'cloned finally' region
// end of 'try 1'
// function epilog
ret

Catch2:
// do something
lea rax, &L1'; // load up resume address
ret

Finally1:
// do something
ret

```

JIT64 はこれを正しく実装していないことに注意してください。C# コンパイラはかつて常に必要なすべての「ステップ」ブロックを挿入していました。Roslyn C# コンパイラは一時期これを行わなくなりましたが、その後再び挿入するように変更されました。

ファンクレットのパラメータ

catch、filter、filter-handler は、引数として例外オブジェクト (GC 参照) を受け取ります ([REG_EXCEPTION_OBJECT](#))。AMD64 では RCX (Windows ABI) または RSI (Unix ABI) で渡されます。ARM および ARM64 では、これは最初の引数であり R0 で渡されます。

ファンクレットの戻り値

filter ファンクレットは、通常のリターンレジスタ (x86: [EAX](#)、AMD64: [RAX](#)、ARM/ARM64: [R0](#)) に単純な布尔値を返します。非ゼロは、対応する filter-handler が例外を処理する (つまり第2パスを開始する) ことを VM/EH サブシステムに示します。ゼロは、例外が処理されないことを VM/EH サブシステムに示し、別の filter または catch の検索を続行すべきことを意味します。

catch および filter-handler のファンクレットは、通常のリターンレジスタにコードアドレスを返します。これは、スタックのアンワンドと例外のクリーンアップ後に VM が実行を再開すべき場所を示します。このアドレスは、親ファンクレット内のどこか (catch や filter-handler が他のファンクレット内にネストされていない場合はメイン関数内) にあるべきです。IL の 'leave' オペコードは任意のネストのファンクレットと try 本体から抜け出すことができるため、JIT はしばしばステップブロックの挿入を求められます。これらは中間的な分岐ターゲットであり、ネイティブ ABI の制約により実際のターゲットに直接到達できるようになるまで、次の最も外側のターゲットに分岐します。これらのステップブロックは finally を呼び出すこともできます (finally の呼び出し / 非ローカル終了を参照)。

finally および fault のファンクレットには戻り値はありません。

レジスタの値と例外処理

例外処理は、例外処理を含む関数におけるレジスタの使用に一定の制約を課します。

CoreCLR と「デスクトップ」CLR は同じ動作をします。Windows と非 Windows の CLR 実装は共にこれらの規則に従います。

いくつかの定義:

不揮発性 (Non-volatile) (別名 *callee-saved* または *preserved*) レジスタとは、ABI により関数呼び出し後も保持されるレジスタです。不揮発性レジスタには、フレームポインタとスタックポインタなどが含まれます。

揮発性 (Volatile) (別名 *caller-saved* または *trashed*) レジスタとは、ABI により関数呼び出し後に保持されないレジスタであり、関数が戻った時点で異なる値になっている可能性があります。

ファンクレット入場時のレジスタ

例外が発生すると、VM が何らかの処理を行うために呼び出されます。例外が "try" リージョン内にある場合、最終的に対応するハンドラを呼び出します (filter の呼び出しも含まれます)。関数内の例外発生場所は、"throw" 命令が実行される場所、ヌルポインタ参照やゼロ除算などのプロセッサ例外の発生点、または呼び出し先が例外をスローしたがキャッチしなかった呼び出しの発生点のいずれかです。

VM はフレームレジスタを親関数と同じ値に設定します。これにより、ファンクレットはフレーム相対アドレスを使用してローカル変数にアクセスできます。

filter ファンクレットの場合、対応する "try" リージョンの例外発生点に存在していたその他すべてのレジスタ値は、ファンクレットへの入場時に破壊されます。つまり、既知の値を持つレジスタはファンクレットのパラメータとフレームレジスタのみです。

その他のファンクレットの場合、すべての不揮発性レジスタは例外発生点の値に復元されます。ただし、JIT のコード生成は[現在これを活用していません](#)。

ファンクレットからの戻り時のレジスタ

ファンクレットの実行が終了し、VM が (EH 句のネストがある場合はその外側のファンクレットまたは) 関数に実行を戻す際、不揮発性レジスタは例外発生点で保持していた値に復元されます。揮発性レジスタは破壊されていることに注意してください。

ファンクレット内で行われたレジスタの値の変更は失われます。ファンクレットがメイン関数（または "try" リージョンを含むファンクレット）に変数の変更を通知したい場合、その変数の変更は共有されるメイン関数のスタックフレームに対して行う必要があります。これは根本的な制限ではありません。必要であれば、ランタイムを更新してファンクレット内で行われた不揮発性レジスタの変更を保持することができます。

ファンクレットは不揮発性レジスタを保持する必要はありません。

EH 情報、GC 情報、およびホット & コールドスプリッティング

💡 初心者向け補足

ホット & コールドスプリッティング (Hot & Cold Splitting) とは、JIT が頻繁に実行されるコード（ホットコード）とめったに実行されないコード（コールドコード、例えば例外ハンドラ）を分離する最適化手法です。ホットコードをまとめて配置することで、CPU キャッシュの局所性が向上し、パフォーマンスが改善されます。

すべての GC 情報オフセットおよび EH 情報オフセットは、関数とファンクレット (funclet) をあたかも1つの大きなメソッド本体であるかのように扱います。したがって、すべてのオフセットはメインメソッドの先頭からの相対値です。ファンクレットは常にメイン関数コードのすべての後（末尾）にあると仮定されます。したがって、メイン関数にコールドコードがある場合、すべてのファンクレットもコールドでなければなりません。逆に言えば、ホットなファンクレットコードがある場合、メインメソッド全体がホットでなければなりません。

EH 句の順序

EH 句は、try 開始/try 終了ペアの IL オフセットに基づいて、内側から外側へ、先頭から末尾への順序でソートされなければなりません。唯一の例外はクローンされた finally であり、これは常に末尾に配置されます。

EH が GC 情報/報告に与える影響

メイン関数本体は、そのファンクレットの1つがスタック上にあるとき、常にスタック上に存在します。そのため、GC 情報は二重報告しないよう注意しなければなりません。JIT64 は、すべての名前付きローカル変数を親メソッドフレームに配置し、関数とファンクレット間で共有されるものはスタックにホーミングし、親関数のみがスタックローカル変数を報告する（ファンクレットはローカルレジスタを報告する場合がある）ことでこれを達成しました。JIT32 と RyuJIT (AMD64、ARM、ARM64 向け) は逆のアプローチを取ります。最も末端のファンクレットが、ファンクレットから生存している可能性のあるすべてのものを報告する責任を持ちます（フィルタの場合、元のメソッド本体に戻って再開する可能性があります）。これは GC ヘッダフラグ WantsReportOnlyLeaf (JIT32 と RyuJIT が設定し、JIT64 は設定しない) と、VM が特定のフレームに対してすでにファンクレットを検出したかどうかを追跡することで実現されます。JIT64 が完全に引退すれば、このフラグを GC 情報から削除できるはずです。

WantsReportOnlyLeaf モデルの VM 実装には、JIT が生成できるコードに影響を与える「コーナーケース」が1つあります。ネストされた例外処理を持つ以下の関数を考えてみましょう：

```
public void runtest() {
    try {
        try {
            throw new UserException3(ThreadId); // 1
        }
        catch (UserException3 e){
            Console.WriteLine("Exception3 was caught");
            throw new UserException4(ThreadId);
        }
    }
    catch (UserException4 e) { // 2
        Console.WriteLine("Exception4 was caught");
    }
}
```

内側の "throw new UserException4" が実行されると、例外処理の第1パスで外側の catch ハンドラがこの例外を処理することが判明します。例外処理の第2パスでスタックフレームを "runtest" フレームまでアンワインドし、catch ハンドラを実行します。元の catch ハンドラ ("catch (UserException3 e)") がスタック上になくなっているから、新しい catch ハンドラが実行されるまでの間に、時間的な空白が生じます。この間に GC が発生する可能性があります。この場合、VM は "runtest" 関数の GC ルートを適切に報告する必要があります。内側の catch はアンワインドされているため、そこを報告することはできません。スタック上にまだ残っている "// 1" で報告することも望ましくありません。なぜなら、それは実質的に実行を「遡る」ことになり、どのオブジェクト参照が生存しているかを正しく表していないためです。次に実行が行われる場所で生存しているオブジェクト参照を報告する必要があります。それが "// 2" の場所です。しかし、catch ファンクレットの最初の場所は非割り込み可能であるため、そこを報告することはできません。代わりに VM はそのハンドラ内の最初の割り込み可能ポイントを先読みし、JIT がその場所で報告する生存参照を報告します。この場所はハンドラプロローグの直後の最初の場所になります。この実装は JIT に対していくつかの影響を与えます。以下が要求されます：

1. EH 句を持つメソッドは完全に割り込み可能 (fully interruptible) でなければなりません。
2. すべての catch ファンクレットはプロローグの直後に割り込み可能ポイントを持たなければなりません。

3. catch ファンクレットの最初の割り込み可能ポイントは、スタック上の以下の生存オブジェクトを報告しなければなりません：

- 親メソッドと共有されるオブジェクトのみ。つまり、catch ファンクレットでのみ生存し、親メソッドでは生存していない追加のスタックオブジェクトは含めません。
 - catch ファンクレットおよび後続の制御フローで参照されるすべての共有オブジェクトが生存として報告されなければなりません。
-

フィルタの GC セマンティクス

フィルタ (filter) は EH 处理の第1パスで呼び出されるため、フォルトアドレス、フィルタハンドラ、またはその他の場所で実行が再開される可能性があります。VM はフィルタの呼び出し中および呼び出し後に GC の発生を許可しなければなりませんが、EH サブシステムがどこで再開するかがまだわかっていない段階であるため、フォルトアドレスおよびフィルタ内の両方ですべてを生存状態に保つ必要があります。これは3つの手段によって達成されます：(1) VM のスタックウォーカーと GCInfoDecoder がフィルタフレームとそれに対応する親フレームの両方を生存として報告する、(2) JIT がフィルタ内で生存しているすべてのスタックスロットをピン留め (pinned) としてエンコードする、(3) JIT がフィルタから生存アウト (live-out) するすべてのものを生存として報告する（場合によってはゼロ初期化する）。(1) のため、フィルタ内と try 本体の両方で生存しているスタック変数は二重報告される可能性が高いです。GC のマークフェーズでは二重報告は問題になりません。問題が生じるのはオブジェクトが再配置 (relocate) される場合のみです：同じ場所が2回報告されると、GC はその場所に格納されているアドレスを2回再配置しようとします。そのため、オブジェクトをピン留めして再配置を防止します。これが(2) を行う必要がある理由です。(3) は、フィルタが返った後、フィルタハンドラまたは同じフレーム内のいずれかの外側ハンドラを実行する前に、安全に GC を実行できるようにするために行われます。同じ理由から、制御はフィルタ領域の最終ブロックを介して終了しなければなりません（つまり、フィルタ領域はフィルタ領域を離れる命令で終了しなければならず、プログラムは他のパスからフィルタ領域を終了してはなりません）。

同じ try 領域をカバーする句

連続するいくつかの句が同じ `try` ブロックをカバーする場合があります。前の句と同じ領域をカバーする句は

`COR_IEXCEPTION_CLAUSE_SAMETRY` フラグでマークされます。例外 ex1 が別の例外 ex2 のハンドラの実行中にスローされ、例外 ex2 が ex1 のハンドラフレームをエスケープした場合、このフラグによりランタイムは ex1 を処理した句と同じ `try` ブロックをカバーする句をスキップできます。このフラグは NativeAOT と CoreCLR の新しい例外処理メカニズムで使用されます。NativeAOT はエンコードされた句データにこのフラグを格納せず、代わりに同じ `try` ブロックを持つ句の間にダミー句を挿入します。CoreCLR は句データのランタイム表現の一部としてこのフラグを保持します。現在の CoreCLR の例外処理はそれを使用しませんが、開発中の[新しい例外処理メカニズム](#)がそれを活用しています。

GC 割り込み可能性と EH

VM は、スレッドが停止されるたびに、GC セーフポイントにあるか、現在のフレームが再開不可能（つまり、同じフレーム内でキャッチされることのない throw）であると仮定します。したがって、実質的に EH を持つすべてのメソッドは完全に割り込み可能でなければなりません（最低限、すべての try 本体が割り込み可能でなければなりません）。現在、GC 情報は同じメソッド内で部分的に割り込み可能な領域と完全に割り込み可能な領域を混在させることをサポートしているように見えますが、どの JIT もこれを使用していないため、自己責任でご使用ください。

デバッガは常に GC セーフポイントで停止したいため、デバッグ可能なコードはデバッガが安全に停止できる場所を最大化するために完全に割り込み可能であるべきです。JIT が完全に割り込み可能なコード内に非割り込み可能な領域を作成する場合、各シーケンスポイントが

割り込み可能な命令で始まるようにすべきです。

AMD64/JIT64 のみ：JIT は必要に応じて割り込み可能な NOP を追加します。

セキュリティオブジェクト

セキュリティオブジェクトは GC ポインタであり、そのように報告され、メソッドの存続期間中生存状態に保たれなければなりません。

GS Cookie

GS Cookie は GC オブジェクトではありませんが、報告される必要があります。GC 情報でのエンコード/報告方法のため、ライフタイムは1つしか持てません。GS Cookie はスタックをポップすると無効になるため、エピローグはライブ範囲の一部にはなれません。ライブ範囲が1つしか取れないため、GS Cookie を持つメソッドではエピローグの後にコード（ファンクレットを除く）を配置することはできません。

NOP とその他のパディング

AMD64 パディング情報

アンワインドコールバックは、現在のフレームがリーフフレームなのかリターンアドレスなのかを判別できません。したがって、JIT は呼び出しのリターンアドレスが呼び出しと同じ領域内にあることを保証しなければなりません。具体的には、JIT は、呼び出しがそのまま try 本体の開始、try 本体の終了、またはメソッドの終了の直前に来る場合、その呼び出しの後に NOP（またはその他の命令）を追加しなければなりません。

OS はアンワインダに最適化があり、アンワインドの結果 PC がエピローグ内（またはエピローグの開始位置）にある場合、そのフレームは重要でないと仮定して再びアンワインドします。CLR はすべてのフレームを重要とみなすため、この二重アンワインド動作を望まず、JIT に呼び出しとエピローグの間に NOP（またはその他の命令）を配置することを要求します。

ARM および ARM64 パディング情報

OS アンワインダは `RUNTIME_FUNCTION` の範囲を使用して、どの関数またはファンクレットからアンワインドするかを決定します。結果として、`IL_Throw` への呼び出し（bl オペコード）を最後の命令にすることはできません。したがって AMD64 と同様に、`bl IL_Throw` が関数またはファンクレットの最後のオペコード、ホットセクションの終了前の最後のオペコード、または（これは ARM に漏れ出した x86 の慣習かもしれません）「特殊 throw ブロック」の直前の最後のオペコードになる場合、JIT はオペコード（この場合はブレークポイント）を挿入しなければなりません。

プロファイラーフック

初心者向け補足

プロファイラーフック (Profiler Hooks) は、プロファイリングツールが実行時にメソッドの入口・出口・テールコールを監視できるようにするコードバックの仕組みです。Java の JVMTI エージェントに似た概念で、パフォーマンス分析やコードカバレッジ計測などに使用されます。

JIT に `CORJIT_FLG_PROF_ENTERLEAVE` が渡された場合、JIT はネイティブの入口/出口/テールコールプローブを挿入する必要があるかもしれません。確実に判断するために、JIT は `GetProfilingHandle` を呼び出す必要があります。この API は `out` パラメータとして、JIT が実際にプローブを挿入すべきかどうかを示す動的な布尔値と、コードバックに渡すパラメータ (`void*` 型) を返します。オプションで間接参照 (NGEN で使用) もあります。このパラメータは常にすべてのコールアウトの第1引数です（したがって、通常の第1引数レジスタ `RCX` (AMD64) または `R0` (ARM、ARM64) に配置されます）。

プロローグの外側 (GC 割り込み可能な場所) で、JIT は `CORINFO_HELP_PROF_FCN_ENTER` への呼び出しを挿入します。AMD64 の場合、Windows ではすべての引数レジスタが呼び出し元が割り当てたスタック位置にホーミングされ (`varargs` と同様)、Unix ではすべての引数レジスタが内部構造体に格納されます。ARM および ARM64 の場合、すべての引数はプリスピル (`prespill`) されます (これも `varargs` と同様)。

戻り値を計算して正しいレジスタに格納した後、エピローグコードの前 (GS Cookie チェックの前を含む) に、JIT は `CORINFO_HELP_PROF_FCN_LEAVE` への呼び出しを挿入します。AMD64 の場合、この呼び出しがリターンレジスタを保存しなければなりません：Windows では `RAX` または `XMM0`、Unix では `RAX` と `RDX` または `XMM0` と `XMM1` です。ARM の場合、戻り値は `R0` から `R2` に移動され (`R0` にあった場合)、`R1`、`R2`、および `S0/D0` は呼び出し先によって保存されなければなりません (`long` は `R2`、`R1` の順 — レジスタの並び順が通常と異なることに注意、浮動小数点は `S0`、倍精度は `D0`、より小さい整数型は `R2`)。

TODO : ARM64 のプロファイルリーブ規約を記述する。

テールコールまたはジャンプコールの引数セットアップの前 (ただし引数の副作用の後) に、JIT は `CORINFO_HELP_PROF_FCN_TAILCALL` への呼び出しを挿入します。自己再帰的なテールコールがループに変換された場合は呼び出されないことに注意してください。

ARM のテールコールの場合、JIT は実際に先に送信引数をロードし、プロファイラーコールアウトの直前に `R0` の引数を別の非揮発性レジスタにスピルし、呼び出しを行い (コードバックパラメータを `R0` で渡し)、その後 `R0` を復元します。

AMD64 の場合、すべてのプローブはデフォルトの引数規則に従って `RDX` で渡される第2パラメータを受け取ります。これは引数のホーム位置の開始アドレス (呼び出し元のスタックポインタの値に相当) です。

TODO : ARM64 のテールコール規約を記述する。

Linux/x86 では、プロファイリングフックは `__cdecl` 属性で宣言されます。cdecl (C declaration の略) では、サブルーチンの引数はスタック上で渡されます。整数値とメモリアドレスは EAX レジスタで返され、浮動小数点値は ST0 x87 レジスタで返されます。レジスタ EAX、ECX、EDX は呼び出し元保存 (caller-saved) であり、残りは呼び出し先保存 (callee-saved) です。x87 浮動小数点レジスタ ST0 から ST7 は、新しい関数を呼び出す際に空 (ポップまたは解放) でなければなりません。また、関数の終了時には ST1 から ST7 は空でなければなりません。ST0 も戻り値の返却に使用しない場合は空でなければなりません。マネージドコードの戻り値は `leave/tailcall` プロファイリングフックの前に形成されるため、これらのフックで保存し、フックからの復帰時に復元する必要があります。プロファイリングフックのアセンブラー実装における命令 `ret` はパラメータなしでなければなりません。

JIT32 はプロファイラーフックがある場合、エピローグを1つだけ生成し (すべての `return` をそこにブランチさせ) ます。

同期メソッド

JIT32/RyuJIT はメソッドが同期化されている場合、エピローグを1つだけ生成し（すべての return をそこにブランチさせ）ます。

`Compiler::fgAddSyncMethodEnterExit()` を参照してください。ユーザーコードは try/finally でラップされます。try 本体の外側/前で、コードはブール値を false に初期化します。`CORINFO_HELP_MON_ENTER` または `CORINFO_HELP_MON_ENTER_STATIC` が呼び出され、ロックオブジェクト（インスタンスマソッドの場合には `this` ポインタ、静的メソッドの場合には Type オブジェクト）とブール値のアドレスが渡されます。ロックが取得されると、ブール値が true に設定されます（Thread.Abort/EH/GC などがブール値とロックの取得状態が一致しないときにスレッドを中断できないという意味で「アトミック」な操作です）。JIT32/RyuJIT は finally 内の `CORINFO_HELP_MON_EXIT` / `CORINFO_HELP_MON_EXIT_STATIC` の呼び出し配置についてまったく同じロジックと引数に従います。

Rejit

AMD64 でプロファイルアタッチシナリオをサポートするために、JIT は生成されるすべてのメソッドがホットパッチ可能であることを保証するよう要求される場合があります（`CORJIT_FLG_PROF_REJIT_NOPS` を参照）。これを実現する方法は、コードの最初の5バイトが非割り込み可能であり、それらのバイト内にブランチターゲットがないこと（呼び出し/リターンを含む）を保証することです。これにより、VM は（GC のときと同様に）すべてのスレッドを停止し、安全にその5バイトをメソッドの新バージョン（おそらくプロファイルによってインストルメントされたもの）へのブランチに置き換えることができます。JIT はこれらの2つの要件を達成するために NOP を追加するか、GC 情報で報告されるプロローグのサイズを増やします。

例外処理を持つ関数では、メイン関数のみが影響を受けます。ファンクレットのプロローグはホットパッチ可能にはされません。

エディットアンドコンティニュー

💡 初心者向け補足

エディットアンドコンティニュー (Edit and Continue, EnC) は、デバッグ中にアプリケーションを再起動することなくコードを変更できるデバッガ機能です。Java の HotSwap や、.NET のホットリロード (Hot Reload) に類似した概念です。開発者はブレークポイントで停止中にコードを編集し、そのまま実行を継続できます。

エディットアンドコンティニュー (EnC) は最適化されていないコードの特殊な形態です。デバッガは、メソッドの状態（命令ポインタとローカル変数）を元のメソッドコードから編集後のメソッドコードに確実に再マッピングできなければなりません。これは JIT が行うメソッドのスタックレイアウトに制約を課します。主要な制約は、既存のローカル変数のアドレスが編集後も同じでなければならないことです。この制約が必要なのは、ローカル変数のアドレスがメソッドの状態に格納されている可能性があるためです。

現在の設計では、JIT はメソッドの以前のバージョンにアクセスできないため、最悪のケースを想定する必要があります。EnC は生成されるコードのパフォーマンスではなく、シンプルさのために設計されています。

EnC は現在 x86、x64、ARM64 でのみ有効ですが、他のプラットフォームで有効にされた場合も同じ原則が適用されます。

以下のセクションでは、従わなければならないさまざまな EnC コード規約について説明します。

GCInfo の EnC フラグ

JIT は EnC コードの規約に従ったことを GC 情報に記録します。x64/ARM64 では、このフラグは EnC 編集間で保持されるスタックフレーム領域のサイズを記録すること（`GcInfoEncoder::SetSizeOfEditAndContinuePreservedArea`）によって暗示されます。x64 ではこの領域のサイズが `RSI` と `RDI` を含むように拡張され、ブロック初期化やブロックムーブに `rep stos` を使用できるようになります。ARM64 は EnC が有効な場合、FP と LR レジスタのみを保存し、他のカリー保存レジスタは使用しません。

EnC 遷移を正常に実行するために、ランタイムは遷移元と遷移先のスタックフレームのサイズを知る必要があります。x64 コードの場合、サイズはアンワインドコードから抽出できます。arm64 コードではフレームの設定方法によりアンワインドコードからこの値を取得できないため、ARM64 では GC 情報に EnC 目的で使用する固定スタックフレームのサイズも含まれます。

ローカル変数の逆方向への割り当て

EnC 編集で新しいローカル変数が追加された場合に既存のローカル変数のアドレスを保持するために必要です。つまり、最初のローカル変数は最も高いスタックアドレスに割り当てられなければなりません。アライメントの処理には特別な注意が必要です。メソッドフレームの合計サイズは、編集後に増加（ローカル変数の追加）することも減少（テンポラリの減少）することもあります。VM は新しく追加されたローカル変数をゼロクリアします。

カリー保存レジスタの固定セット

これにより VM で異なるレジスタセットに対応する必要がなくなり、ローカル変数のアドレスの保持が容易になります。揮発性レジスタは十分にあるため、非揮発性レジスタの不足は最適化されていないコードの品質に大きな影響を与えません。x64 では現在 RBP、RSI、RDI を保存し、ARM64 では FP と LR のみを保存します。

EnC は EH を持つメソッドでサポートされます

ただし、EnC の再マッピングはファンクレット内ではサポートされません。ファンクレットのスタックレイアウトは EnC には関係ありません。

Localloc

EnC コードで Localloc は許可されますが、メソッドが localloc 命令を実行した後は再マッピングが禁止されます。VM は上記の不变条件（x64 では `RSP == RBP`、ARM64 では `FP + 16 == SP + stack size`）を使用して、メソッドが localloc を実行したかどうかを検出します。

セキュリティオブジェクト

x64/arm64 では JIT による特別な処理は不要です（x86 とは異なります）。セキュリティオブジェクトは必要に応じて再マッピング中に VM によってコピーされます。セキュリティオブジェクトの場所は GC 情報から見つけられます。

同期メソッド

JIT が同期メソッドのために作成する追加の状態（ロック取得フラグ）は再マッピング中に保持されなければなりません。JIT はこの状態を保持領域に格納し、GC 情報で報告される保持領域のサイズをそれに応じて増加させます。

ジェネリクス

EnC はジェネリックメソッドの追加と編集、ジェネリック型のメソッド、および非ジェネリック型のジェネリックメソッドでサポートされています。

非同期メソッド

JIT はランタイム非同期メソッドで現在の `ExecutionContext` と `SynchronizationContext` を保存し、これらは再マッピング中に保持されなければなりません。新しい GC エンコーダはこの状態を EnC フレームヘッダサイズに含めます。JIT32 の場合、EE は `getMethodInfo` から `CORINFO_ASYNC_SAVE_CONTEXTS` が報告されたときにこの状態が存在することを期待します。

ポータブルエントリポイント

動的コード生成を許可するプラットフォームでは、ランタイムは動的にロードされたメソッドの実行戦略を `Precode` を割り当てることで抽象化します。`Precode` は実際のメソッドコードが取得されるまでの一時的なメソッドエントリポイントとして使用される小さなコードフラグメントです。`Precode` は通常の JIT コンパイルまたは AOT コンパイルされたコードを持たないメソッド（例えばスタブやインタープリタメソッド）の実行の一部としても使用されます。`Precode` により、ターゲットメソッドが使用する実行戦略に関係なく、ネイティブコードは同じネイティブコード呼び出し規約を使用できます。

動的コード生成を許可しないプラットフォーム（Wasm）では、ランタイムは動的にロードされたメソッドにポータブルエントリポイント（portable entrypoint）を割り当てることで実行戦略を抽象化します。`PortableEntryPoint` はターゲットメソッドの目的の実行戦略への効率的な遷移を可能にするデータ構造です。ランタイムがポータブルエントリポイントを使用するように構成されている場合、マネージド呼び出し規約は以下のように変更されます：

- 呼び出すネイティブコードはエントリポイントを逆参照することで取得されます
- エントリポイントアドレスは追加の最後の隠し引数として渡されます。この追加の隠し引数はすべてのメソッドのシグネチャに存在しなければなりません。JIT コンパイルまたは AOT コンパイルされたメソッドのコードでは使用されません。

ポータブルエントリポイントを使用した呼び出しの擬似コード：

```
(*void**)pfn)(arg0, arg1, ..., argN, pfn)
```

ポータブルエントリポイントは現在 Wasm のインターパリタでのみ使用されています。Native AOT は動的ロードをサポートしないため、Wasm の Native AOT ではポータブルエントリポイントは不要です。

System V x86_64 サポート

初心者向け補足

System V は Linux や macOS で使用される標準的な ABI (Application Binary Interface) です。Windows は独自の ABI を使用しており、レジスタの使い方や呼び出し規約が異なります。たとえば、Windows AMD64 では最初の整数引数を RCX で渡しますが、System V では RDI を使います。この違いを理解することは、クロスプラットフォームの .NET ランタイム開発において重要です。

このセクションは主に System V システム (Ubuntu Linux や Mac OS X など) における呼び出し規約に関するものです。System V x86_64 ABI ドキュメントに記載されている一般的なルールに従いますが、以下に示すいくつかの例外があります。

1. 値渡しの構造体に対する隠し引数は、常に `this` パラメータの後に配置されます (存在する場合)。これは System V ABI との違いであり、内部 JIT 呼び出し規約にのみ影響します。PInvoke 呼び出しでは、この場合 `this` パラメータが存在しないため、隠し引数は常に最初のパラメータになります (`CallConvMemberFunction` の場合を除く)。
2. フィールドを持たないマネージド構造体は、常にスタック上で値渡しされます。
3. JIT は、ネイティブ OS ツールによるスタックアンワインドなどを支援するために、フレームレジスタフレーム (`RBP` をフレームレジスタとして使用) を積極的に生成します。
4. PInvoke、EH、およびジェネリクスサポートに関する他のすべての内部 VM 規約はそのまま維持されます。詳細については上記の関連セクションを参照してください。ただし、異なる呼び出し規約のため、System V では使用されるレジスタが異なることに注意してください。たとえば、整数引数レジスタは順に RDI、RSI、RDX、RCX、R8、R9 です。したがって、Windows AMD64 では最初の引数 (通常は `this` ポインタ) が RCX に入りますが、System V では RDI に入ります。
5. 明示的レイアウト (explicit layout) を持つ構造体は、常にスタック上で値渡しされます。
6. 以下の表は System V x86_64 ABI に基づくレジスタ使用法を説明しています。

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of SSE registers used; 1st return argument	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4st integer argument to functions	No
%rdx	used to pass 3rd argument to functions	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12-%r15	callee-saved registers	Yes
%xmm0-%xmm1	used to pass and return floating point arguments	No

%xmm2-%xmm7	used to pass floating point arguments	No	
%xmm8-%xmm31	temporary registers	No	

x86 の呼び出し規約の詳細

RyuJIT がサポートする他のアーキテクチャとは異なり、マネージド x86 呼び出し規約はデフォルトのネイティブ呼び出し規約とは異なります。これは Windows と Unix の x86 の両方に当てはまります。

標準的なマネージド呼び出し規約は、Windows x86 の fastcall 規約のバリエーションです。主に引数がスタックにプッシュされる順序が異なります。

レジスタで渡すことができる値は、マネージドおよびアンマネージドポインタ、オブジェクト参照、および組み込み整数型 (int8、unsigned int8、int16、unsigned int16、int32、unsigned int32、native int、native unsigned int)、および 4 バイト整数プリミティブ型フィールドを 1 つだけ持つ列挙型と値型のみです。列挙型はその基底型として渡されます。すべての浮動小数点値と 8 バイト整数值はスタック上で渡されます。戻り値の型がレジスタで渡すことができない値型の場合、呼び出し元は結果を保持するバッファを作成し、このバッファのアドレスを隠しパラメータとして渡す必要があります。

引数は左から右の順序で渡されます。`this` ポインタ (インスタンスマソッドおよび仮想マソッドの場合) から始まり、必要に応じてリターンバッファポインタが続き、その後にユーザー指定の引数値が続きます。レジスタに配置できる最初の引数は ECX に、次の引数は EDX に入り、残りはすべてスタック上で渡されます。これは、引数を右から左の順序でスタックにプッシュする x86 ネイティブ呼び出し規約とは対照的です。

戻り値は以下のように処理されます。

1. 浮動小数点値はハードウェア FP スタックの先頭に返されます。
2. 32 ビット以下の整数は EAX に返されます。
3. 64 ビット整数は、EAX に下位 32 ビット、EDX に上位 32 ビットを格納して渡されます。
4. その他のすべてのケースではリターンバッファ (return buffer) を使用する必要があり、それを通じて値が返されます。[リターンバッファ](#) を参照してください。

Windows における制御フローガード (CFG) サポート

💡 初心者向け補足

制御フローガード (Control Flow Guard, CFG) は、攻撃者がプログラムの実行フローを乗っ取ることを防止するセキュリティ機能です。具体的には、間接呼び出し (関数ポインタ経由の呼び出しなど) の宛先が正当なものであるかを検証します。バッファオーバーフローなどの脆弱性を悪用して関数ポインタを書き換える攻撃に対する防御策として機能します。

制御フローガード (CFG) は Windows で利用可能なセキュリティ緩和策です。CFG が有効な場合、オペレーティングシステムは、あるアドレスが有効な間接呼び出しターゲットとみなされるかどうかを検証するために使用できるデータ構造を維持します。このメカニズムは、それぞれ異なる特性を持つ 2 つの異なるヘルパー関数を通じて公開されます。

最初のメカニズムはバリデータ (validator) であり、ターゲットアドレスを引数として受け取り、そのアドレスが期待される間接呼び出しターゲットでない場合にフェイルファストします。そうでなければ、何もせずに返ります。2 番目のメカニズムはディスパッチャ (dispatcher) であり、非標準レジスタでターゲットアドレスを受け取ります。アドレスの検証に成功すると、ターゲット関数に直接ジャンプします。Windows はディスパッチャを ARM64 と x64 でのみ利用可能にしていますが、バリデータはすべてのプラットフォームで利用可能です。ただし、JIT は ARM64 と x64 でのみ CFG をサポートしており、これらのプラットフォームでは CFG はデフォルトで無効になっています。CFG 機能の想定される用途は、CFG が必要とされる制約のある環境で実行される NativeAOT シナリオです。

ヘルパーは標準的な JIT ヘルパー `CORINFO_HELP_VALIDATE_INDIRECT_CALL` および `CORINFO_HELP_DISPATCH_INDIRECT_CALL` として JIT に公開されます。

バリデータを使用するには、JIT は間接呼び出しをバリデータへの呼び出しとそれに続く検証済みアドレスへの呼び出しに展開します。ディスパッチャの場合、JIT はターゲットを渡すように呼び出しを変換しますが、それ以外は通常通り呼び出しを設定します。

ここで「間接呼び出し」とは、命令ストリーム内の即値アドレスへの呼び出しではないすべての呼び出しを指すことに注意してください。たとえば、直接呼び出しであっても、ティアリング (tiering) やまだコンパイルされていない場合などの理由で、JIT コード生成時に間接呼び出し命令を発行することがあります。これらも CFG メカニズムで展開されます。

以下のセクションでは、JIT がこれらのヘルパーに期待する呼び出し規約について説明します。

ARM64 の CFG の詳細

ARM64 では、`CORINFO_HELP_VALIDATE_INDIRECT_CALL` は呼び出しアドレスを `x15` で受け取ります。通常のレジスタに加えて、すべての浮動小数点レジスタ、`x0` - `x8`、および `x15` を保持します。

`CORINFO_HELP_DISPATCH_INDIRECT_CALL` は呼び出しアドレスを `x9` で受け取ります。JIT は分岐予測器の性能が低下するため、デフォルトではディスパッチャヘルパーを使用しません。そのため、すべての間接呼び出しをバリデータヘルパーと手動呼び出しで展開します。

x64 の CFG の詳細

x64 では、`CORINFO_HELP_VALIDATE_INDIRECT_CALL` は呼び出しアドレスを `rcx` で受け取ります。通常のレジスタに加えて、すべての浮動小数点レジスタ、`rcx`、および `r10` も保持します。さらに、シャドウスタック空間を割り当てる必要はありません。

`CORINFO_HELP_DISPATCH_INDIRECT_CALL` は呼び出しアドレスを `rax` で受け取り、`r10` と `r11` を使用および破壊する権利を持ちます。JIT はコードサイズの利点がより正確でない分岐予測を上回ると期待されるため、可能な限り x64 でディスパッチャヘルパーを使用します。ただし、ディスパッチャでの `r11` の使用は VSD 呼び出しと互換性がないため、JIT はバリデータと手動呼び出しにフォールバックする必要があることに注意してください。

Memset/Memcpy に関する注意事項

一般的に、`memset` と `memcpy` はアトミック性 (atomicity) の保証を一切提供しません。つまり、`memset` / `memcpy` で変更されるメモリが他のスレッド (GC を含む) から観測可能でない場合、または[メモリモデル](#)に基づくアトミック性の要件がない場合にのみ使用すべきです。特にマネージドポインタを含むヒープを変更する場合は重要です。マネージドポインタはアトミックに更新する必要があります。たとえば、ポインタサイズの `mov` 命令を使用します (マネージドポインタは常にアライメントされています)。詳細は[アトミックメモリアクセス](#)を参照してください。「更新」とは「ゼロに設定する」ことを暗示しており、それ以外の場合はライトバリア (write barrier) が必要であることに注意してください。

例:

```
struct MyStruct  
{
```

CS

```

    long a;
    string b;
}

void Test1(ref MyStruct m)
{
    // ここでは memset を使用してはいけない
    m = default;
}

MyStruct Test2()
{
    // ここでは memset を使用できる
    return default;
}

```

インタプリタ ABI の詳細

初心者向け補足

.NET ランタイムには JIT コンパイラの代替としてインタプリタ (interpreter) が含まれています。インタプリタは、WebAssembly のような JIT コンパイルが利用できない環境や、起動時間を短縮したい場合に使用されます。Java のバイトコードインタプリタと同様に、IL コードを直接解釈実行しますが、JIT コンパイルされたコードよりも実行速度は遅くなります。

インタプリタのデータstackは通常の「スレッド」stackとは別に割り当てられ、上方向 (UP) に成長します。インタプリタの実行制御stackは「スレッド」stack上に割り当てられ、スレッドの Frame チェーンに配置される `InterpreterFrame` に対して単方向リンクリストで連結された一連の `InterpMethodContextFrame` 値として構成されます。`InterpMethodContextFrame` 構造体は常に降順に割り当てられるため、呼び出し先メソッドに関連する `InterpMethodContextFrame` は、その呼び出し元や含まれている `InterpreterFrame` よりも常にメモリ上の低いアドレスに配置されます。

メソッド内のベースstackポインタは変化しませんが、インタプリタ内で関数が呼び出されると、渡された引数のセットに関連付けられたstackポインタを持ちます。実質的に、引数渡しは呼び出し元関数の一時引数空間の一部を呼び出し先に与えることで行われます。

すべての命令と GC はstackポインタからの相対アドレスで参照され、stackポインタは移動しません。このため、`localalloc` 命令の実装は實際にはヒープ上にメモリを割り当てる必要があり、`localalloc` で割り当てられたメモリはデータstackとは一切関連付けられません。

すべてのインタプリタ関数のstackポインタは常に `INTERP_STACK_ALIGNMENT` 境界でアライメントされています。現在、これは 16 バイトのアライメント要件です。

stack要素は常に少なくとも `INTERP_STACK_SLOT_SIZE` でアライメントされ、`INTERP_STACK_ALIGNMENT` を超えることはありません。現在の実装では `INTERP_STACK_SLOT_SIZE` を 8、`INTERP_STACK_ALIGNMENT` を 16 に設定しているため、stack上のすべてのデータは 8 バイトまたは 16 バイトのアライメントになります。

4 バイト未満のプリミティブ型は、stack上では常にゼロ拡張または符号拡張されて 4 バイトになります。

関数が非同期 (async) の場合、継続リターン (continuation return) を持ります。このリターンはデータstackを使用せず、`InterpreterFrame` の Continuation フィールドを設定することで行われます。サンク (thunk) は、JIT でコンパイルされたコードに入る/出る際にこの値を設定/リセットする役割を担います。

Web Assembly ABI (R2R および JIT)

Web Assembly にコンパイルされたマネージドメソッド (以下「マネージドコード」) では、CLR は一般的に [Wasm Basic C ABI](#) に従います。

マネージドコードは C コードと同じリニアスタック (linear stack) を使用します。スタックは下方向に成長します。

受信引数 ABI

リニアスタックポインタ `$sp` はすべてのメソッドの最初の引数です。ネイティブからマネージドへの遷移時には、`$__stack_pointer` グローバルの値になります。このグローバルはマネージドコード内で現在の `$sp` に更新される場合があり、マネージドからネイティブへの境界では現在の `$sp` と同期している必要があります。メソッド内では、スタックポインタは常にスタックの底 (最低アドレス) を指します。一般的にこれはエントリ時のスタックポインタの値からの固定オフセットですが、動的割り当てが可能なメソッドでは例外です。

フレームポインタが使用される場合、スタックの「固定」部分の底を指し、正のオフセットのみを許可する Wasm アドレッシングモードの使用を容易にします。

構造体は一般的に参照渡し (by-reference) されます。ただし、単一のプリミティブフィールドを正確に含む場合 (またはそのような構造体を正確に含む構造体の場合) は例外です。リニアスタックが参照渡し構造体のバッキングストレージを提供します。

構造体は一般的に隠しバッファ (hidden buffer) を通じて返されます。そのアドレスは呼び出し元によって提供され、マネージド `this` の直後に渡されるか、`this` が存在しない場合は `$sp` 引数の後に渡されます。この場合、メソッドの戻り値は戻り値のアドレスです。ただし、構造体が Wasm スタック上で渡すことができる場合は、Wasm スタック上で返されます。

(未定: ベクトル型の ABI)

プロローグ

プロローグはスタックポインタをインクリメントし、リニアスタックに格納された引数をホームし、リニアスタックのスロットを適切にゼロ初期化します。必要に応じてフレームポインタを確立します。

また、GC と EH で使用するためにフレームディスクリプタをスタックに保存します。EH または GC セーフポイントを持つメソッドでは、EH および GC 情報にインデックスを提供してメソッド内の情報を提供し、外部コードがマネージドスタックフレームをウォークできるようにする「仮想 IP」用のスロットがリニアスタックに予約されます。

エピローグ

一般的にエピローグは空になります。Wasm にはカリーセーブレジスタ (callee-save register) の概念がなく、更新すべき他のグローバル状態もありません。

送信呼び出し ABI

直接マネージド呼び出しでは、Wasm はポータブルエントリポイント (Portable Entry Point) 機能を使用して、インターフェースコードとのスムーズな相互運用を実現します。これはすべてのマネージド呼び出しが間接的に行われ、ポータブルエントリポイントも最後の引数として渡されることを意味します。

呼び出しシーケンスは以下のようになります。

```
local.get sp
push arg 0
...
push arg N-1
load PortableEntryPointPtr    ;; ポータブルエントリポイントのアドレスをプッシュ (&pe)
dup
load CellIndex (from &pe)
call_indirect <tableIndex> <sigIndex>  (sig is: int32 (sp) arg0... argN-1 int32 (&pe))
```

最初はセルにターゲットメソッドが R2R コードを持つかインタプリタで実行する必要があるかを判定するコードが含まれます。メソッドに R2R コードがある場合、検証および必要に応じて修正されます。ターゲットが解決されると、セルは R2R コードがあればそれを直接参照するか、インタプリタを呼び出すためのサンクを参照するように更新できます。

仮想マネージド呼び出しのシーケンスは類似していますが、ポータブルエントリポイントは resolve ヘルパーを呼び出すことで取得されます。

```
local.get sp
push arg 0
...
push arg N-1
... push args for resolution ...
call resolve                ;; ポータブルエントリポイントのアドレスをプッシュ (&pe)
dup
load CellIndex (from &pe)      ;; 呼び出すコードの Wasm 関数テーブルインデックスをプッシュ

call_indirect <tableIndex> <sigIndex>  (sig is: int32 (sp) arg0... argN-1 int32 (&pe))
```

`&pe` 引数はポータブルエントリポイントに渡す必要があるため、すべてのメソッドシグネチャは追加の最終引数を反映する必要があります(使用されなくても)。したがって、たとえば `int F(int x)` のようなマネージドメソッドは Wasm シグネチャ `(func (param int32 int32 int32) (result int32))` を持ちます。

代替として、`&pe` を Wasm グローバルを通じて渡すことも選択できます。

最適化として、vtable ベースの仮想マネージド呼び出しでは、resolve ヘルパーを呼び出す代わりに、適切な vtable スロットからポータブルエントリポイントを取得できます。

最適化として、呼び出し先も R2R でコンパイルされていることがわかっている場合、呼び出し元は呼び出し先を直接呼び出すことができます。R2R メソッドボディは実行時に無効化される可能性があるため、呼び出し先の R2R の検証は呼び出し元の R2R の検証時に行う必要があります。

ネイティブコードで実装された FCall は同じマネージド呼び出し規約に従います。FCall 実装マクロ (`FCIMPL`) は `$__stack_pointer` を再確立する小さなインラインアセンブリラッパーを生成するように変更されます。

呼び出しサイトにおける GC 参照

Wasm は Wasm スタックへの外部アクセスを許可しません。そのため、GC をトリガーする可能性のある呼び出しサイトの前に、呼び出し後も生存するすべての GC 参照 (および追跡されないすべての GC 参照、これらは実質的に常に生存) をリニアスタックに保存する必要があります。これらの GC 参照は GC にピン留め (pinned) として報告されるため、通常 Wasm ローカルに存在する場合でも、呼び出し後にそれらのローカルを更新する必要はありません。リニアスタック上の生存 GC スロットは、仮想 IP (これもリニアスタックに格納) と GC 情報 (フレームディスクリプタからアクセス可能、これもリニアスタック上) によって識別されます。

たとえば、`x(a, y(b)); ... a; ... b;` のようなコードがあり、`a` と `b` が最初に Wasm ローカルにある GC 参照であるとするとき、このフラグメントは以下のようにコンパイルされます。

```
;; x への呼び出し用の sp
local.get sp

;; a をリニアメモリにスpill
local.get sp
local.get a
i32.store offset=(a's offset in gc area of stack)

;; x への呼び出し用の引数 a
local.get a

;; y への呼び出し用の sp
local.get sp

;; b をリニアメモリにスpill
local.get sp
local.get b
i32.store offset=(b's offset in gc area of stack)

;; y への呼び出し用の引数 b
local.get b

;; 生存 GC 参照を持つ y への呼び出し用に仮想 IP を更新
local.get sp
i32.const virtual-ip-for-call-to-y (gc info : a and b slots live)
i32.store offset=(virtual-ip offset)

;; y の &pe とセルインデックスを取得し、y を呼び出す
load PortableEntryPointPtr for y
dup
load CellIndex (from &pe)
call_indirect <tableIndex> <sigIndex> (sig is: int32 (sp) int32 int32 (&pe) : returns int32)

;; 生存 GC 参照を持つ x への呼び出し用に仮想 IP を更新 [最適化で省略可能]
local.get sp
i32.const virtual-ip-for-call-to-x (gc info : a and b slots live)
i32.store offset=(virtual-ip offset)

;; x の &pe とセルインデックスを取得し、x を呼び出す
load PortableEntryPointPtr for x
dup
load CellIndex (from &pe)
call_indirect <tableIndex> <sigIndex> (sig is: int32 (sp) int32 int32 (&pe) : returns int32)
```

注意事項:

- 最適化として、GC/EH 情報が前回の更新から変更されていない場合、仮想 IP の更新を省略できます。

- ネストされた呼び出し結果を親呼び出しに伝えるために、Wasm スタックの代わりに Wasm ローカルを使用して、呼び出しのネストを解除することも考えられます。
 - 最適化として、GC 参照のリニアスタックへの格納を最小限にします（たとえば、前回の更新から値が変更されていない場合）。
 - 最適化として、一部の GC 参照を主にリニアスタック上で保持し、Wasm ローカルには保持しないようにすることも検討できます。
-

末尾呼び出し (Tail Call)

末尾呼び出しとの唯一の違いは、呼び出しで `return_call_indirect` を使用し、元の `sp` 値を呼び出し先に渡すことです。

```
local.get sp
i32.const <frameSize>
i32.add

push arg 0
...
push arg N-1
load PortableEntryPointPtr
dup
load CellIndex (from &pe)
return_call_indirect <tableIndex> <sigIndex>  (sig is: int32 (sp) arg0... argN-1 int32 (&pe))
```

間接マネージド呼び出しでも同様です。

PInvoke

PInvoke はターゲットを呼び出す前に `$__stack_pointer` を再確立します。

リバース PInvoke (Reverse PInvoke)

リバース PInvoke のプロローグはグローバル `$__stack_pointer` をロードし、マネージド `sp` として使用します。

リターン時に、グローバル `$__stack_pointer` はスタブエントリ時の値にリセットされます。

非同期 (Async)

未定

インタプリタスタブ

マネージドコードからインタプリタへの呼び出しとインタプリタからマネージドコードへの呼び出しの両方にスタブが関与します。R2Rでは、これらはシグチャごとに crossgen2 によって生成されます。

インタプリタからマネージドへ

インタプリタからマネージドへのスタブはグローバル `$__stack_pointer` をロードし、次にインタプリタスタックからメソッド引数をロードし、最後にマネージドコードで無視される最終 `&pe` 引数のために `int32.const 0` をロードし (Wasm グローバルを通じて渡す場合はこの最後の部分は省略可能)、そしてマネージドメソッドを呼び出します。

リターン時に、グローバル `$__stack_pointer` はスタブエントリ時の値にリセットされます。

マネージドからインタプリタへ

このスタブには現在のマネージド `sp` が渡され、それをグローバル `$__stack_pointer` に格納する必要があります。インタプリタスタック (上記参照) は新しい `InterpMethodContextFrame` フレームで拡張され、引数は Wasm ローカルからフレームに移動されます。`&pe` 引数は適切な IL メソッドボディでインタプリタを呼び出すために使用されます。

クロスプラットフォームミニダンプ

原文

この章の原文は [Cross-platform Minidumps](#) です。

はじめに

Windows、Linux、およびその他の非 Windows プラットフォームにおけるダンプ (dump) 生成にはいくつかの課題があります。ダンプは非常に大きくなる可能性があり、ダンプのデフォルトの名前や保存場所はサポートされているすべてのプラットフォームで一貫していません。フルコアダンプ (full core dump) のサイズは「coredumpfilter」ファイル/フラグによってある程度制御できますが、最小の設定でも依然として大きすぎる場合があり、デバッグに必要なすべてのマネージド状態 (managed state) が含まれていない可能性があります。デフォルトでは、一部のプラットフォームは _core という名前を使用し、プログラムが起動されたカレントディレクトリにコアダンプを配置します。他のプラットフォームでは名前に pid を追加します。コア名と保存場所の構成にはスーパーユーザー権限が必要です。一貫性のためにスーパーユーザー権限を要求するのは満足のいく選択肢ではありません。

💡 初心者向け補足

ダンプ (dump) とは、プログラムがクラッシュした際にメモリの内容をファイルに保存したものです。Java でいうところのヒープダンプやスレッドダンプに近い概念です。このファイルを後からデバッガで開くことで、クラッシュ時のプログラムの状態を調査できます。「ミニダンプ (minidump)」は、すべてのメモリではなく、デバッグに必要な最小限の情報だけを含むダンプです。

私たちの目標は、サポートされている任意の Linux プラットフォーム上で、WER (Windows エラー報告) のクラッシュダンプと同等のコアダンプを生成することです。少なくとも以下を実現したいと考えています：

- 最小サイズのミニダンプの自動生成。ダンプに含まれる情報の品質と量は、従来の Windows ミニダンプに含まれる情報と同等であること。
- ユーザーによる簡単な構成 (su ではなく！)。

現時点での解決策は、ランタイムの PAL レイヤーで未処理の例外 (unhandled exception) をインター셉トし、coreclr 自身が「ミニ」コアダンプの生成をトリガーすることです。

設計

Breakpad やその派生 (例：SQL チームの内部 MS バージョンである msbreakpad など) のような既存の技術を検討しました。Breakpad は Windows ミニダンプを生成しますが、Windbg などの既存のツールとは互換性がありません。Msbreakpad はさらに互換性がありません。ミニダンプから Linux コアへの変換ユーティリティがありますが、余分な手順に思えます。Breakpad はシグナルハンドラ (signal handler) 内でプロセス内 (in-process) にミニダンプを生成することができます。「非同期」シグナルハンドラ (SIGSEGV など) で許可される API に制限し、同様に制約された C++ ランタイムの小さなサブセットを持っています。さらに、「マネージド」状態のメモリ領域のセットを追加する必要があり、これには DAC の (*) メモリ列挙インターフェイスのロードと使用が必要です。非同期シグナルハンドラではモジュールのロードは許可されていませんが、fork/execve は許可されているため、DAC をロードし、メモリ領域のリ

ストを列挙し、ダンプを書き込むユーティリティを起動するのが唯一の合理的な選択肢です。これにより、ダンプをサーバーにアップロードすることも可能になります。

💡 初心者向け補足

DAC (Data Access Component) は、coreclr ランタイムの一部を特別にビルドしたもので、マネージド状態（スタック、変数、GC 状態のヒープなど）をプロセス外から検査できるようにするものです。Java でいえば、JVM の内部状態を外部ツールから読み取るための仕組みに相当します。シグナルハンドラ (SIGSEGV などのクラッシュシグナルを受け取る関数) 内では安全に呼べる API が限られているため、fork で子プロセスを生成し、そこで DAC をロードしてダンプを書き出す設計になっています。

* DAC は coreclr ランタイムの一部を特別にビルドしたもので、ランタイムのマネージド状態（スタック、変数、GC 状態のヒープ）をコンテキスト外から検査できるようにします。提供される多くのインターフェイスの一つが [ICLRRDataEnumMemoryRegions](#) で、ミニダンプが実りあるデバッグ体験を実現するために必要なすべてのマネージド状態を列挙します。

Breakpad は生成ユーティリティ内でコンテキスト外で使用することもできましたが、ほとんどのシナリオでは lldb のようなプラットフォームツールの使用が必要になるため、ネイティブの Linux コア形式に変換しなければならない Windows 風のミニダンプ形式に価値はない判断しました。また、Google の Breakpad や SQL の msbreakpad ソースリポジトリへの coreclr ビルド依存関係が追加されます。唯一の利点は、Breakpad のミニダンプはメモリ領域がバイト粒度であるのに対し、Linux コアのメモリ領域はページ粒度である必要があるため、若干小さくなる可能性があることです。

実装の詳細

Linux

コアダンプ生成は、未処理のマネージド例外 (unhandled managed exception) や SIGSEGV、SIGILL、SIGFPE などの非同期シグナルにより、coreclr がプロセスをアボート ([PROCAbort\(\)](#) 経由) しようとするたびにトリガーされます。createdump ユーティリティは libcoreclr.so と同じディレクトリに配置されており、fork/execve で起動されます。子プロセスの createdump には ptrace の権限と、クラッシュしたプロセスの各種特殊 /proc ファイルへのアクセス権が与えられ、クラッシュしたプロセスは createdump が完了するまで待機します。

createdump ユーティリティは、ptrace を使用してターゲットプロセスのすべてのスレッドを列挙し、サスペンドすることから始まります。プロセスとスレッドの情報（ステータス、レジスタなど）が収集されます。auxv エントリと DSO 情報が列挙されます。DSO はターゲットによってロードされた共有モジュール (shared module) を記述するメモリ内データ構造です。このメモリは、ロードされた共有モジュールを列挙しそのシンボルにアクセスするために、gdb や lldb がダンプ内で必要とします。モジュールのメモリマッピングは /proc/\$pid/maps から収集されます。プログラムや共有モジュールのメモリ領域はダンプのメモリ領域に明示的には追加されません。DAC がロードされ、Windows と同様にメモリ領域列挙インターフェイスを使用してメモリ領域リストが構築されます。スレッドスタックと IP 周辺の 1 ページ分のコードが追加されます。バイトサイズの領域はページ単位に切り上げられ、連続した領域に結合されます。

/proc/\$pid/maps からのすべてのメモリマッピングは、実際にはメモリがダンプに含まれていなくても PT_LOAD セクションに含まれます。これらのファイルオフセット/サイズは 0 です。

すべてのプロセスクラッシュ情報が収集された後、ELF コアダンプが書き込まれます。メインの ELF ヘッダーが作成・書き込みされます。PT_LOAD ノートセクションがダンプ内の各メモリ領域ごとに 1 エントリずつ書き込まれます。プロセス情報、auxv データ、NTFILE エントリがコアに書き込まれます。NT_FILE エントリは /proc/\$pid/maps からのモジュールメモリマッピングから構築されます。次にスレッドの状態とレジスタが書き込まれます。最後に、_DAC などによって上記で収集されたすべてのメモリ領域がターゲットプロセスから読み取られ、コアダンプに書き込まれます。ターゲットプロセスのすべてのスレッドが再開され、createdump は終了します。

初心者向け補足

ELF (Executable and Linkable Format) は Linux で使われる実行ファイルやコアダンプの標準的なファイル形式です。コアダンプには PTLOAD セクション（メモリ領域の内容）やNOTEセクション（スレッド情報やレジスタの値など）が含まれます。`_createdump` は ptrace というシステムコールを使って、対象プロセスのメモリやレジスタを外部から読み取ります。これは Java の jmap や jstack が JVM の内部状態を取得するのと似た仕組みです。

深刻なメモリ破壊

シグナル/アポートハンドラに制御が到達でき、ユーティリティの fork/execve が成功しさえすれば、DAC のメモリ列挙インターフェイスはある程度の破壊を処理できます。ただし、結果として得られるダンプには有用なマネージド状態が十分に含まれていない可能性があります。このケースを検出してフルコアダンプを書き込むことを検討することもできます。

スタックオーバーフロー例外

深刻なメモリ破壊のケースと同様に、シグナルハンドラ（`SIGSEGV`）が制御を取得できれば、ほとんどのスタックオーバーフローのケースを検出でき、コアダンプをトリガーします。ただし、これが発生せず OS がプロセスを単純に終了してしまうケースも多くあります。ランタイムの初期バージョン（2.1.x 以前）には、スタックオーバーフロー時に `createdump` が起動されないバグがあります。

FreeBSD/OpenBSD/NetBSD

クラッシュ情報の収集にはいくつかの違いがありますが、これらのプラットフォームは依然として ELF 形式のコアダンプを使用するため、ユーティリティのその部分はあまり異ならないはずです。Linux で `createdump` に ptrace の使用権限と /proc へのアクセス権限を付与するために使用されるメカニズムは、これらのプラットフォームには存在しません。

macOS

.NET 5.0 では、`createdump` は macOS でのダンプ生成をサポートしていましたが、MachO ダンプ形式の代わりに ELF コアダンプを生成していました。これは、生成側の MachO ダンブライターと診断ツール側（dotnet-dump および CLRMD）の MachO リーダーの開発の時間的制約によるものでした。これは、5.0 ランタイムで動作するアプリから取得したダンプでは gdb や llDbg のようなネイティブデバッガが動作しないことを意味しますが、dotnet-dump ツールを使用すればマネージド状態を分析できます。この動作のため、以下の「構成/ポリシー」セクションの環境変数に加えて、追加の環境変数（`COMPlus_DbgEnableElfDumpOnMacOS=1`）を設定する必要がありました。

.NET 6.0 からは、ネイティブの Mach-O コアファイルが生成されるようになり、変数 `COMPlus_DbgEnableElfDumpOnMacOS` は非推奨となりました。

Windows

.NET 5.0 以降、`createdump` と以下の構成環境変数は Windows でもサポートされています。Windows の MiniDumpWriteDump API を使用して実装されています。これにより、すべてのプラットフォームで一貫したクラッシュ/未処理例外のダンプが可能になります。

構成/ポリシー

注意: Docker コンテナ内でのコアダンプ生成には ptrace ケーパビリティが必要です (--cap-add=SYS_PTRACE または --privileged の run/exec オプション)。

すべての構成やポリシーは環境変数で設定され、*createdump* ユーティリティにオプションとして渡されます。

サポートされている環境変数：

- `DOTNET_DbEnableMiniDump` : 「1」に設定すると、このコアダンプ生成が有効になります。デフォルトではダンプは生成されません。
- `DOTNET_DbMiniDumpType` : 以下を参照。デフォルトは「2」(MiniDumpWithPrivateReadWriteMemory)。
- `DOTNET_DbMiniDumpName` : 設定された場合、ダンプのパスとファイル名を作成するテンプレートとして使用されます。ダンプ名のフォーマット方法については「ダンプ名のフォーマット」を参照してください。デフォルトは /tmp/coredump.%p です。
- `DOTNET_DbCreateDumpToolPath` : (NativeAOT のみ) 設定された場合、*createdump* ツールが配置されているディレクトリパスを指定します。ランタイムはこのディレクトリ内の *createdump* バイナリを探します。これは、*createdump* がランタイムに同梱されておらず、独自のダンプ生成ツールを「持ち込む」必要があるシナリオで有用です。この環境変数は NativeAOT アプリケーションでのみサポートされ、それ以外では無視されます。
- `DOTNET_CreateDumpDiagnostics` : 「1」に設定すると、*createdump* ユーティリティの診断メッセージ (TRACE マクロ) が有効になります。
- `DOTNET_CreateDumpVerboseDiagnostics` : 「1」に設定すると、*createdump* ユーティリティの詳細な診断メッセージ (TRACE_VERBOSE マクロ) が有効になります。
- `DOTNET_CreateDumpLogFile` : 設定された場合、*createdump* の診断メッセージを書き込むファイルのパスです。
- `DOTNET_EnableCrashReport` : .NET 6.0 以降で「1」に設定すると、*createdump* はクラッシュしたアプリケーションのスレッドとスタックフレームに関する情報を含む JSON 形式のクラッシュレポートも生成します。クラッシュレポート名は、ダンプのパス/名前に .crashreport.json を付加したものです。
- `DOTNET_EnableCrashReportOnly` : .NET 7.0 以降で、`DOTNET_EnableCrashReport` と同じですが、コアダンプは生成されません。

`DOTNET_DbMiniDumpType` の値：

値	ミニダンプ列挙値	説明
1	MiniDumpNormal	プロセス内のすべての既存スレッドのスタックトレースをキャプチャするために必要な情報のみを含みます。GC ヒープメモリと情報は限定的です。
2	MiniDumpWithPrivateReadWriteMemory (デフォルト)	GC ヒープと、プロセス内のすべての既存スレッドのスタックトレースをキャプチャするために必要な情報を含みます。
3	MiniDumpFilterTriage	プロセス内のすべての既存スレッドのスタックトレースをキャプチャするために必要な情報のみを含みます。GC ヒープメモリと情報は限定的です。
4	MiniDumpWithFullMemory	プロセス内のすべてのアクセス可能なメモリを含みます。生のメモリデータは末尾に含まれるため、初期構造は生のメモリ情報なしに直接マッピングできます。このオプションは非常に大きなファイルを生成する可能性があります。

(上記のミニダンプ列挙値の意味については、MSDN の[ミニダンプ列挙値](#)を参照してください)

コマンドラインの使用法

createdump ユーティリティは、任意の .NET Core プロセスに対してコマンドラインから実行することもできます。ダンプの種類は以下のコマンドスイッチで制御できます。デフォルトは、必要なメモリとマネージド状態の大部分を含む「ミニダンプ」です。ptrace (CAP_SYS_PTRACE) の管理者権限がない限り、sudo または su で実行する必要があります。これは lldb やその他のネイティブデバッガでアタッチする場合と同じです。

```
createdump [options] pid
-f, --name - ダンプのパスとファイル名。デフォルトは '/tmp/coredump.%p'。以下の指定子は対応する値に置換されます：
  %p  ダンプされるプロセスの PID
  %e  プロセスの実行ファイル名
  %h  gethostname() が返すホスト名
  %t  ダンプの時刻。エポック (1970-01-01 00:00:00 +0000 (UTC)) からの秒数で表現
-n, --normal - ミニダンプを作成
-h, --withheap - ヒープ付きミニダンプを作成（デフォルト）
-t, --triage - トリアージミニダンプを作成
-u, --full - フルコアダンプを作成
-d, --diag - 診断メッセージを有効化
-v, --verbose - 詳細な診断メッセージを有効化
-l, --logfile - 診断メッセージを記録するファイルのパスと名前
--crashreport - クラッシュレポートファイルを書き込み（ダンプファイルパス + .crashreport.json）
--crashreportonly - クラッシュレポートファイルのみを書き込み（ダンプなし）
--crashthread <id> - クラッシュしたスレッドのスレッド ID
--signal <code> - クラッシュのシグナルコード
--singlefile - シングルファイルアプリのチェックを有効化
```

ダンプ名のフォーマット

.NET 5.0 以降、コアパターン（[core](#) を参照）のダンプ名フォーマットの以下のサブセットがサポートされています：

```
%% 単一の % 文字
%d ダンプされるプロセスの PID (createdump の後方互換性のため)
%p ダンプされるプロセスの PID
%e プロセスの実行ファイル名
%h gethostname() が返すホスト名
%t ダンプの時刻。エポック (1970-01-01 00:00:00 +0000 (UTC)) からの秒数で表現
```

カスタム createdump ツールの使用（NativeAOT のみ）

NativeAOT ランタイムが createdump ツールを同梱していないシナリオでは、[DOTNET_DbgCreateDumpToolPath](#) 環境変数を使用してカスタムディレクトリパスを指定できます：

```
bash
export DOTNET_DbgEnableMiniDump=1
export DOTNET_DbgCreateDumpToolPath=/path/to/directory
./myapp
```

ランタイムは指定されたディレクトリ内の `createdump` バイナリを探します。これにより、独自のダンプ生成ツールを「持ち込む」ことができます。この環境変数は NativeAOT アプリケーションでのみサポートされ、それ以外では無視されることに注意してください。

テスト

テスト計画は、（まだ非公開の）`debugger-tests` リポジトリの SOS テストを変更して、生成されたコアミニダンプをトリガーし使用することです。Linux 上でのマネージドコアダンプのデバッグは、ELF コアダンプリーダーが実装されるまで `mdbg` ではサポートされないため、（Linux 上で `lldb` を使用する）SOS テストのみが変更されます。

混合モードアセンブリ (Mixed Mode Assemblies)

原文

この章の原文は [Mixed Mode Assemblies](#) です。

はじめに

マネージドコード (managed code) とネイティブコード (native code) の間の相互運用 (interoperability) のほとんどは、P/Invoke、COM、または WinRT を使用します。P/Invoke は実行時にネイティブコードにバインドされるため、名前の誤りからスタック破壊を引き起こすシグネチャの微妙なミスに至るまで、さまざまなミスが発生しやすくなっています。COM はネイティブからマネージドコードを呼び出す場合にも使用できますが、多くの場合レジストレーション (registration) が必要であり、パフォーマンスのオーバーヘッドが追加されることがあります。WinRT はこれらの問題を回避しますが、すべてのケースで利用できるわけではありません。

💡 初心者向け補足

P/Invoke とは、C#などのマネージドコードからネイティブの C/C++ ライブラリ (DLL) の関数を呼び出すための仕組みです。Java でいう JNI (Java Native Interface) に相当します。たとえば、Windows API を C# から呼び出したい場合に `[DllImport("user32.dll")]` のように宣言して使います。ただし、関数名やシグネチャを手動で記述するため、間違いが起こりやすいという欠点があります。

C++/CLI は、混合モードアセンブリ (mixed-mode assemblies) と呼ばれる、コンパイラによって検証された異なる相互運用アプローチを提供します (It-Just-Works または IJW と呼ばれることがあります)。開発者が P/Invoke のような特別な宣言を行う必要はなく、C++ コンパイラがマネージドコードとネイティブコードの間の遷移に必要なすべてを自動的に生成します。さらに、C++ コンパイラは各 C++ メソッドがマネージドかネイティブかを判断するため、同一のアセンブリ内であっても、開発者の介入なしに遷移が頻繁に行われます。

💡 初心者向け補足

C++/CLI とは、Microsoft が提供する C++ の拡張言語で、ネイティブの C++ コードと .NET のマネージドコード同じプロジェクト内で混在させることができます。「混合モード (mixed mode)」とは、1つのアセンブリ (DLL や EXE) の中にネイティブコードとマネージドコードの両方が含まれている状態を指します。通常の C# や VB.NET ではすべてのコードが IL (中間言語) にコンパイルされますが、C++/CLI ではネイティブの機械語と IL が同居します。

ネイティブコードの呼び出し

C++/CLI のコードは、同じアセンブリ内のネイティブコードまたは別のライブラリのネイティブコードを呼び出すことができます。別のライブラリへの呼び出しへは、C# で手書きされるものと同様の P/Invoke が生成されます (ただし、C++ コンパイラがそのライブラリのヘッダーを読み取るため、P/Invoke は開発者のミスの影響を受けません)。しかし、同じアセンブリ内への呼び出しへは異なる動作をします。別のライブラリへの P/Invoke はライブラリの名前と呼び出すエクスポートの名前を指定しますが、同じライブラリへの P/Invoke はエン

トリポイントが null で、RVA (Relative Virtual Address、相対仮想アドレス) — ライブラリ内の呼び出し先アドレス — が設定されます。メタデータでは、次のようになります：

```
MethodName: delete (060000EE)
Flags      : [Assem] [Static] [ReuseSlot] [PinvokeImpl] [HasSecurity] (00006013)
RVA       : 0x0001332a
Pinvoke Map Data:
Entry point:
```

これらの P/Invoke の呼び出しは、名前付きエントリポイントを使用する P/Invoke と同じように機能しますが、エクスポートを検索する代わりに、モジュールアドレスと RVA に基づいてアドレスを手動で計算する点が異なります。

マネージドコードの呼び出し

ネイティブ → ネイティブの呼び出しやマネージド → ネイティブの呼び出しはネイティブ関数のアドレスに基づいて行うことができますが、ネイティブ → マネージドの呼び出しはそのようにできません。マネージドコードは実行不可能な IL であるためです。この問題を解決するために、コンパイラはルックアップテーブル (lookup table) を生成し、CIL メタデータヘッダーに `.vtfixup` テーブルとして表示されます。ディスク上のライブラリ内の `vtfixup` は、RVA からマネージドメソッドトークン (managed method token) へのマッピングを行います。アセンブリがロードされると、CLR は `.vtfixup` テーブル内の各メソッドに対して、対応するマネージドメソッドを呼び出すネイティブ呼び出し可能なマーシャリングスタブ (marshaling stub) を生成します。そして、トークンをスタブメソッドのアドレスに置き換えます。ネイティブコードがマネージドメソッドを呼び出す際には、`.vtfixup` テーブル内の新しいアドレスを経由して間接的に呼び出します。

💡 初心者向け補足

`vtfixup` テーブルは、ネイティブコードからマネージドコードを呼び出すための「橋渡し表」のようなものです。マネージドコードは IL (中間言語) のままでは CPU が直接実行できないため、CLR がロード時に実行可能なスタブ (小さなコード片) を生成し、`vtfixup` テーブルのエントリをそのスタブのアドレスに書き換えます。これにより、ネイティブコードはあたかも通常の関数呼び出しをしているかのようにマネージドメソッドを呼び出すことができます。Java の JNI における関数テーブルの仕組みに似た概念です。

たとえば、`IjwLib.dll` 内のネイティブメソッドが、トークン `06000002` を持つマネージドメソッド `Bar` を呼び出したい場合、次のように発行します：

```
call    IjwLib!Bar (1000112b)
```

そのアドレスには、ジャンプ間接参照が配置されます：

```
jmp     dword ptr [IjwLib!_mep?Bar$$FYAXXZ (10010008)]
```

ここで `10010008` は、次のような `.vtfixup` エントリに一致します：

```
.vtfixedup [1] int32 retainappdomain at D_00010008 // 06000002 (Bar のトークン)
```

ECMA 335 によると、`vtfixedup` は複数のエントリを含むことができます。しかし、Microsoft Visual C++ コンパイラ (MSVC) はそのようなエントリを生成しないようです。vtfixedup には、呼び出しが現在のスレッドのアプリケーションドメイン (AppDomain) に送られるべきかどうか、および呼び出し元がアンマネージドコードかどうかを示すフラグも含まれます。MSVC はこれらのフラグを常に設定するようです。

ランタイムの起動

混合モードアセンブリは、すでに実行中の CLR にロードされる場合もありますが、常にそうとは限りません。混合モードの実行可能ファイルがプロセスを開始したり、実行中のネイティブプロセスが混合モードライブラリをロードして呼び出したりすることもあります。.NET Framework（現在この機能を持つ唯一の実装）では、ネイティブコードの `Main` または `DllMain` が mscoree.dll の `_CorDllMain` 関数を呼び出します（既知の場所から解決されます）。その際、`_CorDllMain` はランタイムの起動と、上記で説明した vtfixedup の書き込みの両方を担当します。

.NET を新しいプロセッサーアーキテクチャへ移植するためのガイド

原文

この章の原文は [Guide For Porting](#) です。

このドキュメントは大きく2つのセクションに分かれています。

1. .NET ランタイムにおける各段階
2. 新しいアーキテクチャへの移植で影響を受ける主要コンポーネントに関する技術的な議論

初心者向け補足

「移植 (porting)」とは、あるプラットフォームやプロセッサーアーキテクチャ向けに書かれたソフトウェアを、別のプラットフォームやアーキテクチャで動作するように適応させる作業のことです。たとえば、x86 向けに作られたランタイムを ARM64 で動くようにする、といった作業が該当します。

移植の段階とステップ

.NET ランタイムを新しいアーキテクチャに移植する作業は、一般的に以下のような流れで進みます。

エンジニアリングが開発パスに沿って進む中で、ロジックはできるだけ早くランタイムのメインブランチ (main branch) に配置するのが最善です。これには主に2つの効果があります。

1. 個々のコミットがレビューしやすくなります。
2. 問題の修正アプローチが必ずしも受け入れられるとは限りません。変更がアップストリームの Git リポジトリに受け入れられない可能性もあり、そのような問題を早期に発見することで、大量の埋没費用を回避できます。
3. 他のプラットフォームを壊すような変更が行われた場合、破壊箇所を比較的簡単に特定できます。すべての変更を保留にして、製品が完全に機能するまで待ってから反映しようとすると、この作業ははるかに困難になりやすいです。

ステージ1：初期のブリングアップ (Initial Bring Up)

.NET を新しいプラットフォームに移植するには、まず CoreCLR を新しいアーキテクチャに移植することから始めます。

このプロセスは以下の戦略に従います。

- 新しいターゲットアーキテクチャをビルド環境に追加し、ビルドできるようにします。
- インタープリタ (interpreter) をブリングアップする十分なインセンティブがあるか、それとも単に JIT に新しいアーキテクチャを対応させる方が安価かを判断します。CLR のインタープリタは現在ブリングアップシナリオにのみ使用されており、一般的に動作する状態としてはメンテナンスされていません。CoreCLR のコードベースに精通したエンジニアがインタープリタを有効にするには1~2ヶ月かかると想定されます。機能するインタープリタがあれば、移植チームは JIT に専念するエンジニアと VM 部分に専念するエンジニアに分かれて作業できます。

初心者向け補足

「bring up」とは、新しいハードウェアやプラットフォーム上でソフトウェアを初めて動作させるプロセスのことです。Java でいえば、新しいアーキテクチャ上で JVM を初めて起動させるような作業に相当します。JIT (Just-In-Time コンパイラ) は実行時にコードをコンパイルし、インタープリタは命令を1つずつ解釈して実行します。

- CoreCLR のテストを実行するスクリプトのセットを構築します。CoreCLR テストを実行する通常の手段は XUnit ですが、これはフレームワークがおおむね機能するようになってから初めて適切に使えるようになります。これらのスクリプトは開発の進行に伴い、ますます増える開発ニーズに対応するよう進化していきます。このスクリプトセットには以下のタスクが期待されます。
 - テストのサブセットを実行する。テストはディレクトリ構造でカテゴリ別に整理されているため、このサブセット化の仕組みはディレクトリ構造ベースのシステムだけで十分です。
 - 一部のテストをテスト単位で除外する必要があります。製品の出荷準備が整った時点で、無効化されたテストの大部分を再有効化する必要がありますが、製品の品質が十分に高まるまで数ヶ月～数年間無効化されたままとなるテストもあります。
 - クラッシュダンプまたはコアダンプを生成する。このフェーズでは多くのテストの失敗モードがクラッシュになります。コアダンプをキャプチャするテスト実行ツールがあれば、こうした問題の診断が容易になります。
 - 失敗をバケット化したリストを生成する。一般的なアプローチは、アサーションでグループ化し、クラッシュの場合はクラッシュのコールスタックでグループ化することです。
- 最初に注力すべきテストカテゴリは JIT カテゴリです。これは .NET コードを実行する基本的な能力をbring upするためです。これらのテストのほとんどは非常にシンプルですが、何らかのコードを動作させることは、より複雑なシナリオを扱うための前提条件です。初期bring upの際は、GC の Gen0 バジェットを大きな数値に設定して、ほとんどのテスト中に GC が実行を試みないようになると非常に便利です。（`DOTNET_GCgen0size=99999999` を設定）

初心者向け補足

Gen0 バジェットとは、ガベージコレクタ (GC) が第0世代のヒープにどれだけのメモリを割り当ててからコレクションを開始するかを制御するパラメータです。この値を非常に大きくすると、GC がほぼ起動しなくなるため、GC が未完成の段階でも JIT の基本動作のテストに集中できます。

- 基本的なコードが実行できるようになったら、次は GC を動作させることに注力します。この初期フェーズでは、`FEATURE_CONSERVATIVE_GC` マクロを使って保守的 GC トラッキング (conservative GC tracking) を有効にするのが正しい選択です。この機能によりガベージコレクションはおおむね正しく機能しますが、.NET の本番利用には適しておらず、特定の状況下で際限のないメモリ使用を引き起こす可能性があります。
- 基本的な GC が動作し、基本的な JIT 機能が揃ったら、ランタイムのさまざまな機能に広げて作業できます。ランタイムを移植するエンジニアにとって特に関心が高いのは、EH (例外処理)、スタックウォーキング (stackwalking)、および相互運用 (interop) のテストサイトです。
- このフェーズでは、<https://github.com/dotnet/diagnostics> から SOS プラグインを移植することが非常に有用です。このツールで利用できる dumpmt、dumpdomain などのさまざまなコマンドは、ランタイムの移植を試みる開発者にとって日常的に役立ちます。

ステージ2：シナリオカバレッジの拡大 (Expand Scenario Coverage)

- CoreCLR のテストがおおむねパスするようになったら、次のステップは XUnit を有効にすることです。この時点で CLR はおそらく XUnit テストを実行できるだけの能力をおおむね備えており、ライブラリテストを使ったテストの追加には XUnit が正しく動作する必要があります。
- XUnit が機能するようになったら、ライブラリのテストセットをブリングアップします。CoreCLR コードベースの相当部分は、ライブラリのテストスイートによってのみテストされています。
- エンジニアはこの時点で、ASP.NET Core アプリケーションなどの実際のシナリオテストも開始すべきです。ライブラリのテストスイートが動作すれば、ASP.NET Core も動作するはずです。

ステージ3：パフォーマンスへの注力 (Focus on Performance)

- この時点でスループットパフォーマンスはおそらくあまり良くないでしょう。このステージではパフォーマンスを改善するための3つの主要な機会があります。
 - 保守的 GC を正確な GC (precise GC) に置き換える。
 - アセンブリスタブ (assembly stubs) をプラットフォーム上で高性能になるようチューニングし、手書きのアセンブリが同等の C++ コードよりも高速になるオプションのアセンブリスタブを実装する。
 - JIT が生成するコードを改善する。
- この時点まで、エンジニアはおそらく Ready To Run コンパイラ (crossgen/crossgen2) をプラットフォームで使用する代わりに、すべてのコードに JIT を使用してきたでしょう。AOT (Ahead-Of-Time) コンパイラの実装は、起動パフォーマンスを向上させるためにこの時点から有用になり始めます。

💡 初心者向け補足

「保守的 GC (conservative GC)」は、メモリ上の値が GC 参照かどうかを厳密に判別せず、「参照かもしれない」値はすべて参照として扱います。これにより実装は簡単になりますが、本来回収できるメモリを回収できない場合があります。「正確な GC (precise GC)」は、各値が参照かどうかを正確に把握するため効率的ですが、JIT や VM からの正確なメタデータが必要です。

ステージ4：ストレスへの注力 (Focus on Stress)

- システムが本当に動作するという信頼性を確保するためには、ストレステスト (stress testing) が必要です。
- CI で行われるさまざまなテストパスを参照してください。特に重要なのは GCStress テストです。 `DOTNET_GCStress` 環境変数の使用に関するドキュメントを参照してください。

ステージ5：製品化 (Productization)

- 製品化とは、ランタイムをプラットフォーム上で効果的に出荷・実行できるようにすることです。

- このドキュメントでは、ここでの作業を列挙することは試みません。使用するプラットフォームや多数のステークホルダーの意見に大きく依存するためです。

設計上の課題

以下の大きなアーキテクチャ固有の設計課題は、JIT と VM の両方に大きな影響を与えます。

1. 呼び出し規約 (calling convention) のルール – 呼び出し元スタック解放 (Caller pop) vs 呼び出し先スタック解放 (Callee pop)、HFA 引数、構造体の引数渡しルールなど。CoreCLR は OS の API と広く類似した ABI を利用するように設計されています。マネージド間呼び出しには通常、VM の効率のために ABI に対する小さな調整や拡張のセットがありますが、マネージドコードの ABI とネイティブコードの ABI は一般的に非常に類似することが意図されています。(これは厳格な要件ではなく、Windows X86 ではランタイムはマネージド間 ABI に加えて、相互運用のための3つの別々のネイティブ ABI をサポートしていますが、このスキームは一般的に推奨されません。) 既存アーキテクチャの動作については [CLR ABI ドキュメント](#) を参照してください。新しいプラットフォームのすべての必要な詳細と特殊ケースで CLR ABI ドキュメントが更新されていることを確認してください。CoreCLR の新しいプロセッサアーキテクチャ ABI の動作を定義する際には、以下を維持する必要があります。

- `this` ポインタは、他のパラメータに関係なく、常に同じレジスタで渡されること。
- さまざまなスタブタイプが追加の「秘密の (secret)」パラメータを必要とすること。パフォーマンスの詳細が、これらがどこに配置されるかを通常決定します。
- マネージドコードの実行時に、リターンアドレスをハイジャック (hijack) することが可能でなければならないこと。現在の実装では、リターンアドレスが常にスタック上にある必要がありますが、これは ARM64 などの RISC プラットフォームにおける既知のパフォーマンス上の欠陥です。

💡 初心者向け補足

ABI (Application Binary Interface) とは、コンパイルされたコード同士がどのように相互作用するかを定義する規約です。Java の JNI (Java Native Interface) に似た概念で、引数の渡し方、レジスタの使い方、スタックの管理方法などを規定します。「リターンアドレスのハイジャック」とは、ランタイムがスレッドを安全なポイントで停止させるために、関数の戻りアドレスを書き換えてランタイムのコードに制御を移すテクニックです。

- アーキテクチャ固有のリロケーション情報 (relocation information) (ロード、ストア、jmp、call 命令で使用されるリロケーションの生成を表現するため)。定義が必要な詳細の類例については <https://learn.microsoft.com/windows/win32/debug/pe-format#coff-relocations-object-only> を参照してください。
- プロセス内からのプロセッサのシングルステップ機能 (single step features) の動作とアクセシビリティ。Unix では CLR デバッガはプロセス内スレッドを使用して関数をシングルステップ実行します。
- アンワインド情報 (unwind information)。CoreCLR は Unix プラットフォーム上でも内部的に Windows スタイルのアンワインドデータを使用します。Windows スタイルのアンワインド構造体を定義する必要があります。さらに、GDB JIT <https://sourceware.org/gdb/onlinedocs/gdb/JIT-Interface.html> を通じて公開するための DWARF データの生成を有効にすることも可能です。このサポートは `#ifdef` による条件付きですが、過去に新しいプラットフォームのブリングアップをサポートするために使用されてきました。
- EH ファンクレット (EH Funclets)。.NET は例外フィルタ (exception filters) を適切にサポートするために、2パスの例外モデル (2 pass exception model) を必要とします。これは、ほとんどの Linux アーキテクチャで使用される典型的な Itanium ABI モデルとは大きく異なります。
- シグナル (Signals) に関する OS の動作。特に、報告される命令ポインタ (instruction pointer) が正確にどこに位置するか。

7. リトルエンディアン (little endian) vs ビッグエンディアン (big endian)。過去に .NET ランタイムがビッグエンディアンに移植された例はあります (Mono がさまざまなゲームコンソールや POWER をサポートしていた例、Xbox360 での XNA サポートなどが顕著な例です) が、CoreCLR のビッグエンディアンプラットフォームへの現行の移植はありません。

新しいアーキテクチャへの移植で影響を受けるランタイムのコンポーネント

これは、.NET ランタイムの注目すべきアーキテクチャ固有コンポーネントのリストです。このリストは完全ではありませんが、作業が必要となるほとんどの領域をカバーしています。

注目すべきコンポーネント

1. JIT。JIT はスタック内でアーキテクチャ固有のロジックの最大の集中を維持しています。これは驚くことではありません。ガイダンスについては [RyuJIT の移植](#) を参照してください。
2. CLR PAL。Windows 以外の OS に移植する場合、PAL が最初に移植が必要なコンポーネントとなります。
3. CLR VM。VM は完全にアーキテクチャに依存しないロジックと、非常にマシン固有のパスが混在しています。
4. アンワインダ (unwinder)。アンワインダは Windows 以外のプラットフォームでスタックをアンワインドするために使用されます。
<https://github.com/dotnet/runtime/tree/main/src/coreclr/unwinder> にあります。
5. System.Private.CoreLib/System.Reflection。プリングアップに必要なアーキテクチャ固有の作業はほとんど、またはまったくありません。あると望ましい作業としては、System.Reflection.ImageFileMachine enum および ProcessorArchitecture enum のアーキテクチャサポートの追加と、それらを操作するロジックがあります。
6. 新しいアーキテクチャを追加するための PE ファイルフォーマットの変更。また、C# コンパイラも新しいアーキテクチャ向けのマシン固有コードを生成するための新しいスイッチがおそらく必要です。
7. Crossgen/Crossgen2 - 汎用の MSIL からマシン固有のロジックを生成する AOT コンパイラとして、起動パフォーマンスを向上させるために必要になります。
8. R2RDump - 事前コンパイルされたコードの問題を診断できます。
9. coredistools - GCStress (命令境界の判定が自明でない場合)、および JIT 開発用の SuperPMI asm diff に必要です。
10. デバッグおよび診断コンポーネント - マネージドデバッガとプロファイラーはこのドキュメントの範囲外です。

CLR PAL

PAL は、CLR のコードベースがもともと Windows プラットフォーム上で動作するように設計されていたため、Win32 API に類似した API を提供します。PAL は主に OS の独立性に関わるものですが、アーキテクチャ固有のコンポーネントもあります。

1. pal.h - `CONTEXT` / `_KNONVOLATILE_CONTEXT_POINTERS` / `_RUNTIME_FUNCTION` などのアンワインドシナリオを処理するためのアーキテクチャ固有の詳細を含みます。
2. `seh-unwind.cpp` でのアンワインドサポート
3. context.cpp - レジスタコンテキストの操作とキャプチャを行います。

4. `jitsupport.cpp` - CPU の機能がどのように公開されるかに応じて、CPU の機能に関する情報を収集するために OS の API を呼び出すコードが必要になる場合があります。
 5. PAL arch ディレクトリ - <https://github.com/dotnet/runtime/tree/main/src/coreclr/pal/src/arch> このディレクトリには主に、シグナルと例外のアーキテクチャ固有の処理のためのアセンブリスタブが含まれています。
PAL のソースコードに加えて、<https://github.com/dotnet/runtime/tree/main/src/coreclr/pal/tests> に包括的な PAL テストのセットがあります。
-

CLR VM

VM のアーキテクチャ固有のロジックのサポートは、さまざまな方法でエンコードされています。

1. 完全にアーキテクチャ固有のコンポーネント。これらはアーキテクチャ固有のフォルダに保持されます。
2. 特定のアーキテクチャでのみ有効な機能。例：`FEATURE_HFA`。
3. 特定のアーキテクチャに使用されるアドホックな `#if` ブロック。必要に応じて追加されます。一般的な目標はこれらを最小限に抑えますが、ここでの難しさは主にプロセッサアーキテクチャがどのような特殊な動作を必要とするかによって決まります。

VM で CPU アーキテクチャを実装する方法の最新モデルとして、Arm64 がどのように実装されているかを見ることをお勧めします。

アーキテクチャ固有コンポーネント

すべてのアーキテクチャが実装しなければならない、さまざまなアーキテクチャ固有のコンポーネントがあります。

1. アセンブリスタブ (Assembly Stubs)
2. `cgencpu.h` (CPU 固有のヘッダ。スタブやその他の CPU 固有の詳細を定義)
3. VSD コールスタブ生成 (`virtualcallstubcpu.hpp` および関連ロジック)
4. プリコード/プレスタブ/ジャンプスタブの生成 (Precode/Prestub/Jumpstub generation)
5. `callingconventions.h` / `argdestination.h` — VM コンポーネントが使用する ABI の実装を提供します。実装は長い一連の C プリプロセッサマクロを介してアーキテクチャ固有にされています。
6. `gcinfodecoder.h` — GC 情報のフォーマットはアーキテクチャ固有です。これは、どの特定のレジスタが GC データを保持しているかについての情報を保持するためです。実装は一般的にレジスタ番号の観点で定義されるように簡略化されていますが、既存のアーキテクチャよりも多くのレジスタが使用可能なアーキテクチャの場合、フォーマットの拡張が必要になります。

アセンブリスタブ (Assembly Stubs)

ランタイムがさまざまなアセンブリスタブを必要とする理由は多くあります。以下は、Unix 上の Arm64 向けに実装されたスタブの注釈付きリストです。

1. パフォーマンスのみ。一部のスタブには C++ コードでの代替実装があり、アセンブリスタブがない場合はそちらが使用されます。コンパイラが改善されるにつれて、C++ バージョンをそのまま使用することがより合理的になってきました。多くの場合、最大のパフォー

マンスコスト/メリットは、スタックフレームを設定する必要がないファストパス (fast path) が書かれていることによるものです。キャスティングヘルパーのほとんどがこのカテゴリに該当します。

1. `JIT_Stelem_Ref` – `JIT_Stelem_Ref_Portable` のわずかに高速なバージョン。
2. 汎用的な正確性。一部のヘルパーは呼び出し先の ABI を興味深い方法で調整したり、「秘密の」引数を操作/解析したり、標準化された C の概念にコンパイルできない他の処理を行います。
 1. `CallDescrWorkerInternal` – VM からマネージド関数への呼び出しをサポートするために必要です。main メソッドはこの方法で呼び出されるため、すべてのアプリケーションに必要です。
 2. `PInvokeImportThunk` – P/Invoke への引数のセットを保存して、ランタイムが実際のターゲットを見つけられるようにするために必要です。秘密の引数の1つも使用します（すべての P/Invoke メソッドで使用）。
 3. `PrecodeFixupThunk` – 秘密の引数を FixupPrecode* から MethodDesc* に変換するために必要です。この関数は FixupPrecode のコードサイズを削減するために存在します（多くのマネージドメソッドで使用）。
 4. `ThePreStub` – ランタイムが正しいターゲットメソッドを見つけるか JIT コンパイルできるように、引数のセットをスタックに保存するために必要です。（JIT コンパイルされるメソッドの実行に必要。すべてのマネージドメソッドで使用）
 5. `ThePreStubPatch` – マネージドデバッガがブレークポイントを設置するための信頼性の高いスポットを提供するために存在します。
6. GC ライトバリア (Write Barriers) – どのメモリが更新されているかについての情報を GC に提供するために使用されます。これらの既存の実装はすべて複雑で、ランタイムがバリアの動作をさまざまな方法で調整するための多くのコントロールがあります。これらの調整の一部は、定数を注入するためにコードを変更したり、さまざまな部分をまるごと置き換えたりすることを含みます。高いパフォーマンスを達成するためにはこれらすべての機能が動作する必要がありますが、シンプルな GC をサポートするブリングアップを達成するには、シングルヒープのワークステーション GC のケースに焦点を当ててください。さらに、`FEATURE_MANUALLY_MANAGED_CARD_BUNDLES` と `FEATURE_USE_SOFTWARE_WRITE_WATCH_FOR_GC_HEAP` はパフォーマンスの必要性に応じて実装できます。
7. `ComCallPreStub` / `COMToCLRDispatchHelper` / `GenericComCallStub` – 現時点では Windows 以外のプラットフォームには不要です。
8. `TheUMEntryPrestub` / `UMThunkStub` – Marshal.GetFunctionPointerForDelegate API から生成されたエントリポイン트を通じて、非マネージドコードからランタイムに入るために使用されます。
9. `OnHijackTripThread` – GC やその他の停止を必要とするイベントをサポートするためのスレッド停止 (thread suspension) に必要があります。これは製品の非常に初期段階のブリングアップには通常必要ありませんが、ある程度のサイズのアプリケーションには必要です。
10. `CallEHFunclet` – catch、finally、fault のファンクレット (funclet) を呼び出すために使用されます。動作はファンクレットがどのように実装されているかに固有です。
11. `CallEHFilterFunclet` – フィルタファンクレット (filter funclet) を呼び出すために使用されます。動作はファンクレットがどのように実装されているかに固有です。
12. `ResolveWorkerChainLookupAsmStub` / `ResolveWorkerAsmStub` – 仮想スタブディスパッチ (virtual stub dispatch)（インターフェイスおよび一部の仮想メソッドの仮想呼び出しサポート）に使用されます。これらは virtualcallstubcpu.h のロジックと連動して、仮想スタブディスパッチ で説明されているロジックを実装します。
13. `ProfileEnter` / `ProfileLeave` / `ProfileTailcall` – ICorProfiler インターフェイスを通じて取得された関数のエントリ/イグジットプロファイル関数を呼び出すために使用されます。非常にまれな状況で使用されます。これらの実装は製品化の最終段階まで待つののが合理的です。ほとんどのプロファイルラーはこの機能を使用しません。

14. `JIT_PInvokeBegin` / `JIT_PInvokeEnd` – マネージドランタイム状態からの離脱/エントリ。ReadyToRun で事前コンパイルされた P/Invoke 呼び出しが GC スタベーション (starvation) を引き起こさないようにするために必要です。
15. `VarargPInvokeStub` / `GenericPInvokeCalliHelper` — calli P/Invoke をサポートするために使用されます。C# 8.0 でこの機能の使用が増加することが期待されます。現在、Unix でのこの機能の使用には手書きの IL が必要です。Windows ではこの機能は C++/CLI で一般的に使用されます。

初心者向け補足

「スタブ (stub)」とは、ある処理を別の処理に橋渡しするための小さなコード片のことです。Java でいうところのプロキシやアダプタに近い役割を果たします。たとえば、マネージドコード (C#) からネイティブコード (C/C++) を呼び出す際、引数の変換やレジスタの保存などを行うスタブが挿入されます。「ファンクレット (funclet)」は、例外処理のために分離された小さなコード片で、try-catch-finally の各ブロックに対応する独立した関数のようなものです。

`cgencpu.h`

このヘッダは VM ディレクトリのさまざまなコードからインクルードされます。アーキテクチャ固有の大きな機能セットを提供しており、以下を含みますが、これに限定されません。

1. VM が作成すべきさまざまなデータ構造のサイズなどを指定する、アーキテクチャ固有の定義
2. さまざまな JIT ヘルパーのうち、ポータブルな C++ 実装の代わりにアセンブリ関数に置き換えるべきものを指定する定義
3. プラットフォームの呼び出し規約を記述するために必要な `CalleeSavedRegisters`、`ArgumentRegisters`、および `FloatArgumentRegisters`
4. `ClrFlushInstructionCache` 関数。アーキテクチャが実際に手動で命令キャッシュ (icache) をフラッシュする必要がない場合、この関数は空です。
5. ジャンプ命令のデコードと操作のためのさまざまな関数。これらはコードがどこに向かうかを予測したり、シンプルなジャンプスタブを生成するために、さまざまなスタブルーチンで使用されます。
6. アーキテクチャの `StubLinkerCpu` クラス。各アーキテクチャは独自の `StubLinkerCpu API` サーフェスを定義し、それを使用して VM が生成するコードを生成します。汎用 VM コード (`EmitComputedInstantiatingMethodStub`、`EmitShuffleThunkshared`) から複数のアーキテクチャにわたって呼び出される小さな API のセットがあり、さらにアーキテクチャ固有の個々のアセンブリ命令エミッション関数があります。`StubLinker` は、スタブごとにエミットされるアセンブリ命令のセットが異なる複雑なスタブを生成するために使用されます。
7. さまざまなスタブデータ構造。非常にシンプルなスタブの多くは、バイトストリームのエミッションでは生成されず、代わりに非常に規則的であり、実質的に各スタブで同じ命令で、わずかに異なるデータメンバーを持つだけです。`StubLinker` メカニズムを使用する代わりに、VM はスタブ全体と関連データを表す構造体を持ち、通常のコンストラクタ呼び出しでマジックナンバーを設定してアセンブリ命令とデータフィールドを埋めます。実行可能であることに加えて、これらのスタブは、ある関数が何であるか、何をしているか、制御フローがどこに向かうかなどを正確に判別するためにペースされることもあります。

`virtualcallstubcpu.h`

このヘッダは、仮想スタブディスパッチ (virtual stub dispatch) で使用されるさまざまなスタブの実装を提供するために使用されます。これらのスタブは、[仮想スタブディスパッチ](#) で説明されているルックアップスタブ (lookup stub)、リゾルバスタブ (resolver stub)、およびディスパッチスタブ (dispatch stub) です。これらは歴史的な理由と、サイズの理由（ここにはかなり多くのロジックがあります）により、`cgencpu.h` の残りの部分とは別のファイルで管理されています。

System.Private.CoreLib

初期プリングアップ

System.Private.CoreLib では、初期プリングアップに必要な作業はありません。

完全なサポート

完全なサポートには、製品の公開 API サーフェスの変更が含まれます。これは GitHub 上のパブリックイシューと API レビュー ボードとの議論を通じて処理されるプロセスです。

- System.Reflection.ImageFileMachine enum および System.Reflection.ProcessorArchitecture enum、ならびに関連ロジックへのアーキテクチャサポートの追加
- SIMD 命令やその他の非標準 API サーフェスなど、アーキテクチャ固有のイントリニシック (intrinsics) のサポートの追加

ベクトルとハードウェアイントリンシック

原文

この章の原文は [Vectors and Intrinsics](#) です。

はじめに

CoreCLR ランタイムは、複数の種類のハードウェアイントリンシック (hardware intrinsics) をサポートしており、それらを使用するコードをコンパイルするためのさまざまな方法を提供しています。このサポートはターゲットプロセッサによって異なり、生成されるコードは JIT コンパイラの呼び出し方法に依存します。このドキュメントでは、ランタイムにおけるイントリンシックのさまざまな動作を説明し、ランタイムおよびライブラリの開発者への影響をまとめます。

💡 初心者向け補足

ハードウェアイントリンシック (hardware intrinsics) とは、CPU が持つ特定の命令 (SIMD 命令など) を、C# のコードから直接利用するための仕組みです。通常のコードよりも高速に処理できる場合があり、画像処理や数値計算などのパフォーマンスが重要な場面で活用されます。Java でいえば、JVM が内部的に行う最適化に近いのですが、.NET では開発者が明示的にこれらの命令を利用できます。

略語と定義

略語	定義
AOT	事前コンパイル (Ahead of Time)。このドキュメントでは、プロセスの起動前にコードをコンパイルし、ファイルに保存して後で使用することを指します。

イントリンシック API

ハードウェアイントリンシックのサポートの大部分は、さまざまなベクトル (Vector) API の使用に結びついています。ランタイムがサポートする主要な API サーフェスは 4 つあります。

- 固定長浮動小数点ベクトル (**fixed length float vectors**)。[Vector2](#)、[Vector3](#)、[Vector4](#) です。これらのベクトル型は、さまざまな長さの float の構造体を表します。型レイアウト、ABI、および相互運用 (interop) の目的では、適切な数の float を持つ構造体とまったく同じ方法で表現されます。これらのベクトル型の操作はすべてのアーキテクチャとプラットフォームでサポートされていますが、一部のアーキテクチャではさまざまな操作が最適化される場合があります。

- 可変長 `Vector<T>`。これは、ランタイムで決定される長さのベクトルデータ (vector data) を表します。任意のプロセス内では、`Vector<T>` の長さはすべてのメソッドで同一ですが、マシンやプロセス起動時に読み取られる環境変数の設定によって異なる場合があります。型パラメータ `T` には、`System.Byte`、`System.SByte`、`System.Int16`、`System.UInt16`、`System.Int32`、`System.UInt32`、`System.Int64`、`System.UInt64`、`System.Single`、`System.Double` の型を指定でき、ベクトル内で整数データや倍精度浮動小数点データを使用できます。`Vector<T>` の長さとアライメント (alignment) はコンパイル時には開発者にとって未知です (ただし、`Vector<T>.Count` API を使用してランタイムに取得可能です)。また、`Vector<T>` は相互運用シグネチャ (interop signature) で使用できません。これらのベクトル型の操作はすべてのアーキテクチャとプラットフォームでサポートされていますが、`Vector<T>.IsHardwareAccelerated` API が `true` を返す場合には一部のアーキテクチャでさまざまな操作が最適化されることがあります。
- `Vector64<T>`、`Vector128<T>`、`Vector256<T>`、`Vector512<T>` は、C++ で利用可能な固定サイズベクトルに近い固定サイズベクトルを表します。これらの構造体は実行されるあらゆるコードで使用できますが、作成以外にこれらの型で直接サポートされる機能はほとんどありません。主にプロセッサ固有のハードウェアイントリニシック API で使用されます。
- プロセッサ固有のハードウェアイントリニシック API (`System.Runtime.Intrinsics.X86.Ssse3` など)。これらの API は、特定のハードウェア命令に固有の個別の命令または短い命令シーケンスに直接マッピングされます。これらの API は、特定の命令をサポートするハードウェアでのみ使用可能です。設計については <https://github.com/dotnet/designs/blob/master/accepted/2018/platform-intrinsics.md> を参照してください。

💡 初心者向け補足

ベクトル型は大きく分けて 2 種類あります。`Vector2` / `Vector3` / `Vector4` や `Vector<T>` は、プラットフォームに依存せず安全に使用できる「ポータブル」なベクトルです。一方、`Vector128<T>` や `System.Runtime.Intrinsics.X86.Avx2` などは特定の CPU アーキテクチャに依存する「プラットフォーム固有」のベクトルおよび API です。後者を使用する場合は、そのハードウェアが利用可能かどうかを `IsSupported` プロパティで必ず確認する必要があります。

イントリニシック API の使用方法

イントリニシック API の使用モデルは 3 つあります。

- `Vector2`、`Vector3`、`Vector4`、`Vector<T>` の使用。これらについては、常に安全にそのまま型を使用できます。JIT はロジックに対して可能な限り最適なコードを無条件に生成します。
- `Vector64<T>`、`Vector128<T>`、`Vector256<T>`、`Vector512<T>` の使用。これらの型は無条件に使用できますが、プラットフォーム固有のハードウェアイントリニシック API と併用する場合にのみ真に有用です。
- プラットフォームイントリニシック API の使用。これらの API の使用はすべて、適切な種類の `IsSupported` チェックでラップする必要があります。そして、`IsSupported` チェック内でプラットフォーム固有の API を使用できます。複数の命令セットを使用する場合、アプリケーション開発者はそれぞれの命令セットに対してチェックを行う必要があります。

ハードウェアイントリニシックの使用がコード生成に与える影響

ハードウェアイントリニシックはコード生成 (codegen) に大きな影響を与え、これらのハードウェアイントリニシックのコード生成は、コードがコンパイルされるときにターゲットマシンで利用可能な ISA (命令セットアーキテクチャ) に依存します。

コードが JIT によってジャストインタイム (just-in-time) 方式でランタイムにコンパイルされる場合、JIT は現在のプロセッサの ISA に基づいて可能な限り最適なコードを生成します。このハードウェアイントリンシックの使用は、JIT コンパイルティア (compilation tier) に依存しません。`MethodImplOptions.AggressiveOptimization` を使用して、ティア 0 コードのコンパイルをバイパスし、メソッドに対して常にティア 1 コードを生成するようにすることができます。さらに、ランタイムの現在のポリシーでは、`MethodImplOptions.AggressiveOptimization` を使用して R2R コードとしてのコンパイルもバイパスできますが、これは将来変更される可能性があります。

💡 初心者向け補足

ティアードコンパイル (tiered compilation) とは、.NET ランタイムが採用する段階的コンパイル方式です。最初に高速だが最適化の少ないティア 0 コードを生成し、頻繁に呼び出されるメソッドについては後からより最適化されたティア 1 コードを再コンパイルします。`MethodImplOptions.AggressiveOptimization` を指定すると、最初から最適化されたコードを生成するようになります。R2R (ReadyToRun) は AOT コンパイルの一形態で、事前にコンパイルされたネイティブコードをアセンブリに埋め込み、アプリケーションの起動時間を短縮します。

AOT コンパイルの場合は、状況ははるかに複雑になります。これは、AOT コンパイルモデルの以下の原則によるものです。

1. AOT コンパイルは、いかなる状況においても、パフォーマンスの変化を除いてコードのセマンティックな動作を変更してはなりません。
2. AOT コードが生成された場合、使用を避ける正当な理由がない限り、そのコードを使用すべきです。
3. AOT コンパイルツールを誤用して原則 1 に違反することは、極めて困難でなければなりません。

Crossgen2 のハードウェアイントリンシック使用モデル

コンパイラに認識される命令セットは 2 つのセットがあります。

- ベースライン命令セット (**baseline instruction set**): デフォルトでは x86-64-v2 (SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT) ですが、コンパイラオプションで調整できます。
- 楽観的命令セット (**optimistic instruction set**): デフォルトでは (AES, GFNI, SHA, WAITPKG, X86SERIALIZE) です。

コードは楽観的命令セットを使用してコンパイルが駆動されますが、ベースライン命令セットを超える命令セットの使用は記録され、また楽観的セットを超える命令セットの使用がセマンティックな影響を持つ場合は、その試みも記録されます。ベースライン命令セットに `Avx2` が含まれている場合、`Vector<T>` のサイズと特性が既知になります。ABI に関する他の決定もエンコードされる可能性があります。たとえば、`Vector256<T>` と `Vector512<T>` の ABI は `Avx` サポートの有無に基づいて変わる可能性があります。

- `Vector<T>` を使用するコードは、`Vector<T>` のサイズが既知でない限り AOT コンパイルされません。
- Linux または Mac マシンで `Vector256<T>` または `Vector512<T>` をパラメータとして渡すコードは、`Avx` 命令セットのサポートが既知でない限り AOT コンパイルされません。
- 楽観的にサポートされるハードウェア能力を超えるハードウェアサポートを必要とする非プラットフォームイントリンシックは、その能力を活用しません。`MethodImplOptions.AggressiveOptimization` を使用して、この準最適なコードのコンパイルを無効にすることができます。
- 楽観的セットの命令セットを活用するコードは、ベースライン命令セットのみをサポートするマシンでは使用されません。
- 楽観的セット外の命令セットを使用しようとするコードは、その命令セットをサポートするマシン上で使用されないコードを生成します。

ルールから生じる特性

- 楽観的命令セット内のプラットフォームイントリンシックを使用するコードは、良好なコードを生成します。
- ベースラインまたは楽観的セットに含まれないプラットフォームイントリンシックに依存するコードは、その命令セットをサポートするハードウェアで使用された場合、ランタイム JIT と起動時間の問題を引き起こします。
- `Vector<T>` コードは、ベースラインを `Avx2` を含むように引き上げない限り、ランタイム JIT と起動時間の問題があります。

プラットフォームイントリンシック使用のコードレビュールール

- コードベースでのプラットフォームイントリンシックの使用は、関連する `IsSupported` プロパティへの呼び出しでラップすべきです (SHOULD)。このラッピングは、ハードウェアイントリンシックを使用する同じ関数内で行うことができますが、プログラマがハードウェアイントリンシックを使用する関数へのすべてのエントリポイントを制御できる限り、必須ではありません。
- アプリケーション開発者が起動パフォーマンスを非常に気にする場合、開発者は SSE4.2 を超えるイントリンシックの使用を避けるか、更新されたベースライン命令セットサポートで Crossgen を使用すべきです。

System.Private.CoreLib.dll のための Crossgen2 ルールの調整

System.Private.CoreLib.dll は、以下に記述するコードレビュールールに基づいてコードレビューされることが判明しているため、「楽観的セット外の命令セットを使用しようとするコードは、その命令セットをサポートするマシン上で使用されないコードを生成する」というルールを緩和することができます。これにより、これらの状況では非最適なコードが生成されますが、コードレビューとアナライザの効果により、生成されたロジックは正しく動作します。

System.Private.CoreLib.dll で記述されるコードのコードレビューおよびアナライザールール

- コードベースでのプラットフォームイントリンシックの使用は、関連する `IsSupported` プロパティへの呼び出しでラップしなければなりません (MUST)。このラッピングは、ハードウェアイントリンシックを使用する同じ関数内で行わなければなりません (MUST)。あるいは、プラットフォームイントリンシックを使用する関数は、`CompExactlyDependsOn` 属性を使用して、この関数がある型のプラットフォームイントリンシックを無条件に呼び出すことを示す必要があります。
- プラットフォームイントリンシックを使用する単一の関数内では、`CompExactlyDependsOn` 属性でマークされていない限り、`IsSupported` が true を返すか false を返すかにかかわらず、同一の動作をしなければなりません。これにより、R2R コンパイラがより低いイントリンシックサポートセットでコンパイルしても、ティアードコンパイルーション (tiered compilation) 下でその関数の動作が変わらないことを期待できます。
- イントリンシックの過度な使用は、追加の JIT コンパイルによる起動パフォーマンスの問題、または準最適なコード生成によるパフォーマンス特性の未達成を引き起こす可能性があります。これを修正するために、将来的には `CompExactlyDependsOn` でマークされたメソッドを適切なプラットフォームイントリンシックを有効にしてコンパイルするようにコンパイルルールを変更する可能性があります。

`IsSupported` プロパティと `CompExactlyDependsOn` 属性の正しい使用は、`System.Private.CoreLib` のビルド中にアナライザによって検査されます。このアナライザーは、`IsSupported` プロパティのすべての使用が、いくつかの特定のパターンに準拠することを要求します。これらのパターンは、if 文または三項演算子 (ternary operator) を介してサポートされます。

サポートされる条件チェックは以下のとおりです。

1. `IsSupported` フラグを使用した単純な if 文による囲み

```
csharp
if (PlatformIntrinsicType.IsSupported)
{
    PlatformIntrinsicType.IntrinsicMethod();
}
```

2. 使用されるイントリニシックがサポートされることを暗示するプラットフォームイントリニシック型をチェックする if 文

```
csharp
if (Avx2.X64.IsEnabled)
{
    Avx2.IntrinsicMethod();
}
```

3. 外側の条件が相互に排他的な条件の `IsSupported` チェックを OR で結合したシリーズであり、内側のチェックが一部のチェックの適用を除外する `else` 節であるネストされた if 文

```
csharp
if (Avx2.IsEnabled || ArmBase.IsEnabled)
{
    if (Avx2.IsEnabled)
    {
        // 何かを行う
    }
    else
    {
        ArmBase.IntrinsicMethod();
    }
}
```

4. より低いレベルの属性に対して `[CompExactlyDependsOn]` でマークされたメソッド内で、より高度な CPU 機能に対して明示的な `IsSupported` チェックを使用する場合があります。その場合、関数全体の動作は、CPU 機能が有効かどうかにかかわらず同一でなければなりません。アライザーはこの使用を警告として検出し、ヘルパーメソッド内の `IsSupported` の使用が、まったく同等の動作を維持するルールに従っていることを確認します。

```
csharp
[CompExactlyDependsOn(typeof(Sse41))]
int DoSomethingHelper()
{
#pragma warning disable IntrinsicsInSystemPrivateCoreLibAttributeNotSpecificEnough // else 節はセマンティック的に同等
    if (Avx2.IsEnabled)
#pragma warning disable IntrinsicsInSystemPrivateCoreLibAttributeNotSpecificEnough
    {
        Avx2.IntrinsicThatDoesTheSameThingAsSse41IntrinsicAndSse41.Intrinsic2();
    }
    else
    {
        Sse41.Intrinsic();
        Sse41.Intrinsic2();
    }
}
```

- 注: ヘルパーが異なる命令セットが有効な場合に異なる動作をする必要がある場合、正しいロジックでは `CompExactlyDependsOn` 属性をすべての呼び出し元に広げ、いかなる呼び出し元も間違った動作を期待してコンパイルされないようにする必要があります。
`Vector128.ShuffleUnsafe` メソッドおよびさまざまな使用例を参照してください。

`CompExactlyDependsOn` の動作は、特定のメソッドに1つ以上の属性を適用できることです。属性で指定された型のいずれかが、関連する `IsSupported` プロパティに対してランタイム時に不变の結果を持たない場合、そのメソッドはR2Rコンパイル中に別の関数にコンパイルまたはインライン化されません。そのように記述された型のいずれも `IsSupported` メソッドに対して true の結果を持たない場合、そのメソッドはR2Rコンパイル中に別の関数にコンパイルまたはインライン化されません。

5. `IsSupported` プロパティを直接使用してイントリニシックのサポートを有効/無効にすることに加えて、コードの重複を減らすために、以下のスタイルで記述されたシンプルな静的プロパティ (static property) を使用できます。

```
csharp
static bool IsVectorizationSupported => Avx2.IsSupported || PackedSimd.IsSupported

public void SomePublicApi()
{
    if (IsVectorizationSupported)
        SomeVectorizationHelper();
    else
    {
        // ベクトル化されない実装
    }
}

[CompExactlyDependsOn(typeof(Avx2))]
[CompExactlyDependsOn(typeof(PackedSimd))]
private void SomeVectorizationHelper()
{}
```

💡 初心者向け補足

`CompExactlyDependsOn` 属性は、`System.Private.CoreLib` 内のコードで使用される特殊な属性です。この属性は「このメソッドは指定された命令セットに正確に依存する」ことを宣言します。R2R (ReadyToRun) コンパイル時にこの情報が使用され、適切な命令セットが利用可能な場合にのみコードがコンパイル・インライン化されるようになります。一般的なアプリケーション開発者がこの属性を直接使用することはほとんどありません。

System.Private.CoreLib における非決定的イントリニシック

`System.Private.CoreLib` で公開されている一部の API は、意図的にハードウェア間で非決定的 (non-deterministic) であり、単一プロセスのスコープ内でのみ決定性を保証します。このような API のサポートを容易にするために、JIT は `Compiler::BlockNonDeterministicIntrinsics(bool mustExpand)` を定義しており、`ReadyToRun` などのシナリオでこのような API の展開をブロックするために使用されるべきです。さらに、このような API は自身を再帰的に呼び出すべきであり、間接的な呼び出し (デリゲート、関数ポインタ、リフレクションなど経由) でも同じ結果が計算されるようにします。

このような非決定的 API の例として、`System.Single` と `System.Double` で公開されている `ConvertToInt32Native` API があります。これらの API は、基盤となるハードウェアで利用可能な最速のメカニズムを使用して、ソース値をターゲット整数型に変換します。これらの API が存在する理由は、IEEE 754 仕様が入力を出力に収められない場合の変換を未定義としており（たとえば `float.MaxValue` を `int` に変換する場合）、その結果、異なるハードウェアがこれらのエッジケースで歴史的に異なる動作を提供してきたためです。これらの API により、エッジケースの処理を気にする必要がないが、デフォルトのキャスト演算子の結果を正規化するパフォーマンスオーバーヘッドが大きすぎる開発者に対して代替手段を提供します。

もう1つの例は、`float.ReciprocalSqrtEstimate` などのさまざまな `*Estimate` API です。これらの API により、ユーザーは多少の不正確さのコストでより高速な結果を選択でき、発生する正確な不正確さは入力と命令が実行される基盤ハードウェアに依存します。

JIT でさまざまな命令セットサポートに対して正しいコードを生成するためのメカニズム

JIT は使用可能な命令セット (instruction set) を指示するフラグを受け取り、新しい JIT インターフェース API

`notifyInstructionSetUsage(isa, bool supportBehaviorRequired)` にアクセスできます。

`notifyInstructionSetUsage` API は、コードがブールパラメータが示すとおりのランタイム環境でのみ実行できることを AOT コンパイラインフラストラクチャに通知するために使用されます。たとえば、`notifyInstructionSetUsage(Avx, false)` が使用された場合、生成されたコードは `Avx` 命令セットが使用可能な場合に使用されることはなりません。同様に、`notifyInstructionSetUsage(Avx, true)` は、コードが `Avx` 命令セットが利用可能な場合にのみ使用できることを示します。

上記の API は存在しますが、JIT 内の汎用コードがこれを使用することは想定されていません。一般的に、JIT コンパイルされるコードは、利用可能なハードウェア命令サポートを理解するためにいくつかの異なる API を使用することが期待されています。

API	使用方法の説明	厳密な動作
<code>compExactlyDependsOn(isa)</code>	命令セットを使用するかしないかの決定が、生成されるコードのセマンティクスに影響を与える場合に使用します。アサーションでは使用しないでください。	命令セットがサポートされているかどうかを返します。その計算結果とともに <code>notifyInstructionSetUsage</code> を呼び出します。
<code>compOpportunisticallyDependsOn(isa)</code>	命令セットを使用するかしないかの機会主義的な決定を行う場合に使用します。命令セットの使用が「あれば嬉しい最適化の機会」である場合に使い、 <code>false</code> の結果がプログラムのセマンティクスを変更する可能性がある場合には使用しないでください。アサーションでは使用しないでください。	命令セットがサポートされているかどうかを返します。命令セットがサポートされている場合に <code>notifyInstructionSetUsage</code> を呼び出します。
<code>compIsaSupportedDebugOnly(isa)</code>	命令セットがサポートされているかどうかをアサートするために使用します。	命令セットがサポートされているかどうかを返します。何も報告しません。デバッグビルドでのみ使用可能です。
<code>getVectorTByteLength()</code>	<code>Vector<T></code> 値のサイズを取得するために使用します。	<code>Vector<T></code> 型のサイズを決定します。アーキテクチャ上でサイズが変動する可能性がある場合は、コンパイル時とランタイム間でサイズ

API	使用方法の説明	厳密な動作
		が一貫するように <code>compExactlyDependsOn</code> を使用してクエリを実行します。
<code>getMaxVectorByteLength()</code>	このコンパイルで SIMD 型に使用される可能性がある最大バイト数を取得します。	サポートされる命令セットのセットをクエリし、サポートされる最大の SIMD 型を決定します。必要な最大サイズのみが記録されるように <code>compOpportunisticallyDependsOn</code> を使用してクエリを実行します。

ILC コンパイラーアーキテクチャ

原文

この章の原文は [ILC Compiler Architecture](#) です。

著者: Michal Strehovsky ([@MichalStrehovsky](#)) - 2018

ILC (IL Compiler) は、CIL (Common Intermediate Language、共通中間言語) で書かれたプログラムを、簡素化された CoreCLR ランタイム上で実行するためのターゲット言語または命令セットに変換する事前コンパイラー (Ahead-of-Time Compiler) です。ILC への入力は、C#、VB.NET、F#などの一般的なマネージド言語コンパイラーが生成する共通の命令形式です。ILC の出力は、ターゲットプラットフォーム向けのネイティブコード (native code) と、ターゲットランタイム上でそのコードを実行するために必要なデータ構造 (data structures) です。少し大げさに言えば、ILC は C# のための事前ネイティブコンパイラーと言えるでしょう。

💡 初心者向け補足

従来の .NET では、プログラムは「中間言語 (IL)」という形でコンパイルされ、実行時に JIT (Just-In-Time) コンパイラーがネイティブコードに変換していました。ILC はこれとは異なり、実行前にネイティブコードへ変換します。Java で例えると、GraalVM の Native Image に相当する仕組みです。これにより、アプリケーションの起動時間やメモリ使用量が大幅に改善されます。

従来、CIL は「ジャストインタイム」(JIT) でコンパイルされてきました。これは、CIL からターゲットランタイム環境で実行可能な命令セットへの変換が、ネイティブコードが実行の継続に必要になったとき（例えば CIL メソッドの初回呼び出し時）に、必要に応じて行われることを意味します。事前コンパイラー (ahead-of-time compiler) は、コードとデータ構造をプログラムの実行開始前にあらかじめ準備しようとします。コードの実行に必要なネイティブコードとデータ構造が事前に利用可能であることの主な利点は、プログラムの起動時間とワーキングセット (working set) の大幅な改善です。

完全に事前コンパイルされた環境では、コンパイラーは実行時に必要となるすべてのコードとデータ構造を生成する責任を負います。つまり、コンパイル後は元の CIL 命令やプログラムメタデータ（メソッド名やそのシグネチャなど）の存在はもはや不要です。留意すべき重要な点として、事前コンパイルは JIT コンパイルを排除するものではありません。アプリケーションの一部は事前コンパイルされ、他の部分は JIT コンパイルまたはインタープリタで実行される、混合実行モードを想像することもできます。ILC はこのような動作モードをサポートする必要があります。なぜなら、どちらにも長所と短所があるからです。私たちは過去にこのような混合実行モードのプロトタイプを作成したことがあります。

目標

- CIL をコンパイルし、ターゲットプラットフォーム向けのネイティブコードを生成する
- ランタイムがマネージドネイティブコードを実行するために必要な基本的なデータ構造を生成する（例外処理やメソッドの GC 情報、型を記述するデータ構造、GC レイアウトと vtable、インターフェースディスパッチャなど）
- 基本クラスライブラリ (base class libraries) がユーザーコードにリッチなマネージド API を提供するために必要なオプションのデータ構造を生成する（リフレクション (reflection)、相互運用 (interop)、テキストのスタックトレース情報、実行時の型ロードなどをサポートするデータ構造）
- プログラム全体の解析ステップからのオプション入力をサポートし、コンパイルに影響を与える
- 実行可能ファイルおよび静的/動的ライブラリの生成をサポートする（フラットな C スタイルのパブリック API サーフェスを持つ）
- 複数のコンパイルモードをサポートする:

- 単一ファイル出力 (Single-file output):
 - すべての入力アセンブリが ILC によって生成される単一のオブジェクトファイルにマージされる（マネージドアセンブリのマージは ILC 内で行われる）。このモードでは最大限の最適化が可能。
 - 複数のオブジェクトファイルを生成し、プラットフォームリンクによって単一の実行可能ファイルにマージする（マージは ILC の実行後にネイティブリンクで行われる）。このモードではインクリメンタルコンパイルが可能だが、ILC 内の最適化が制限される。
- マルチファイル出力 (Multi-file output) (1つ以上の入力アセンブリが 1つ以上の動的ライブラリを生成し、それらが互いに動的にリンクする)。このモードでは複数の実行可能ファイルや動的ライブラリ間でコード/データの共有が可能だが、多くの最適化が制限される。
- マルチスレッドコンパイル (Multi-threaded compilation)
- ターゲットプラットフォーム向けのネイティブデバッガ情報を生成し、ネイティブデバッガでのデバッging を可能にする
- プラットフォームのネイティブオブジェクトファイル形式 (.obj ファイルおよび .o ファイル) で出力を生成する
- 入力が不完全な場合 (例: アセンブリの欠落やアセンブリのバージョン不一致) に定義された動作を持つ

ILC の構成

ILC はおおまかに 3 つの部分で構成されています: コンパイルドライバ (compilation driver)、コンパイラー (compiler)、そしてコード生成バックエンド (code generation backends) です。

コンパイルドライバ

コンパイルドライバの役割は、コマンドライン引数を解析し、コンパイラーをセットアップし、コンパイルを実行することです。コンパイラーのセットアップ作業には [CompilationBuilder](#) の設定が含まれます。コンパイルビルダは、コマンドライン引数の指示に従ってドライバがコンパイラーの各種コンポーネントを設定・構成できるメソッドを公開しています。これらのコンポーネントは、何がコンパイルされるか、どのようにコンパイルが行われるかに影響を与えます。最終的に、ドライバは [Compilation](#) オブジェクトを構築し、コンパイルの実行、コンパイル結果の検査、出力のディスクへの書き込みなどのメソッドを提供します。

関連クラス: [CompilationBuilder](#), [ICompilation](#)

コンパイラー

コンパイラーは、本ドキュメントの残りの部分で説明する中核コンポーネントです。コンパイルプロセスの実行と、ターゲットランタイムおよびターゲット基本クラスライブラリ向けのデータ構造の生成を担当します。

コンパイラーは、何をコンパイルするか、どのデータ構造を生成するか、どのようにコンパイルを行うかについて、ポリシーフリー (policy-free) を維持しようとします。具体的なポリシーは、コンパイルの設定の一部としてコンパイルドライバによって供給されます。

コード生成バックエンド

ILC は、同一のランタイムをターゲットとする複数のコード生成バックエンド (code generation backends) をサポートするように設計されています。これは、コンパイラー内に共通の部分（何をコンパイルする必要があるかの決定、基盤ランタイム向けのデータ構造の生成）と、ターゲット環境固有の部分があるモデルを意味します。共通の部分（データ構造のレイアウト）はターゲット固有ではありません。タ

ターゲット固有の違いは、「ターゲットは相対ポインタの表現をサポートするか？」といった一般的な質問に限定され、データ構造の基本的な形状はターゲットプラットフォームに関係なく同じです。

ILC は現在、以下のコード生成バックエンドをサポートしています（完成度にはらつきあり）：

- **RyuJIT**: CoreCLR の JIT コンパイラとしても使用されるネイティブコードジェネレータ。このバックエンドは Windows、Linux、macOS、BSD 上の x64、arm64、arm32 をサポートします。
- **LLVM**: LLVM バックエンドは現在、Emscripten と連携して WebAssembly コードを生成するために使用されています。[NativeAOT-LLVM ブランチ](#) にあります。

本ドキュメントでは、すべてのコード生成バックエンドに共通するコンパイラの部分について説明します。

過去に ILC がサポートしていたバックエンドは以下の通りです：

- **CppCodegen**: CIL を C++ コードに変換するポータブルコードジェネレータ。新しい CPU アーキテクチャ用のコードジェネレータを構築する代わりに、プラットフォームがすでに持っている可能性の高い C++ コンパイラに依存することで、プラットフォームへの迅速な対応をサポートします。移植性にはある程度のコストが伴います。このコード生成バックエンドは、現在アーカイブされている CoreRT リポジトリからは[移植されませんでした](#)。

関連プロジェクトファイル: ILCompiler.LLVM.csproj, ILCompiler.RyuJit.csproj

依存関係分析

ILC でのコンパイルを駆動する中核概念は、依存関係分析 (dependency analysis) です。依存関係分析は、出力オブジェクトファイルに生成する必要のあるランタイムアーティファクト（メソッドのコード本体やさまざまなデータ構造）のセットを決定するプロセスです。依存関係分析は、各頂点が以下のいずれかであるグラフを構築します：

- 出力ファイルの一部となるアーティファクトを表す（「コンパイル済みメソッド本体」や「実行時に型を記述するデータ構造」など）— これは「オブジェクトノード (object node)」です。
- コンパイルされたプログラムの特定の抽象的な特性を捉える（「プログラムに `Object.GetHashCode` メソッドへの仮想呼び出しが含まれる」など）— これは一般的な「依存関係ノード (dependency node)」です。一般的な依存関係ノードは出力にバイトとして物理的に表れませんが、通常は（推移的に）出力の一部を形成するオブジェクトノードにつながるエッジを持っています。

グラフのエッジは「必要とする (requires)」関係を表します。コンパイルプロセスは、このグラフを構築し、どのノードがグラフの一部であるかを決定することに相当します。

💡 初心者向け補足

依存関係分析は、Java のプロファイルガイド最適化 (PGO) や、JavaScript のツリーシェイキング (tree shaking) に似た概念です。プログラムのどの部分が実際に必要かをグラフとして分析し、不要なコードやデータを出力から除外することで、最終的な実行ファイルのサイズを最小化します。

関連クラス: `DependencyNodeCore<>`, `ObjectNode`

関連プロジェクトファイル: ILCompiler.DependencyAnalysisFramework.csproj

依存関係の展開プロセス

コンパイルは、依存関係グラフ内のコンパイルルート (compilation roots) と呼ばれるノードのセットから開始されます。ルートはコンパイルドライバによって指定され、通常は `Main()` メソッドを含みますが、正確なルートのセットはコンパイルモードに依存します。例えば、ライブラリをビルドする場合、マルチファイルコンパイルを行う場合、単一ファイルアプリケーションをビルドする場合では、ルートのセットが異なります。

このプロセスは、ルートノードのリストを調べ、それらの依存関係（依存ノード）を確定することから始まります。依存関係がわかると、コンパイルはその依存関係の依存関係の検査に進み、すべての依存関係が判明して依存関係グラフ内でマークされるまで繰り返します。それが完了すると、コンパイルは終了です。

グラフの展開は、コンパイルグループ (compilation group) の範囲内に留まる必要があります。コンパイルグループは、依存関係グラフがどのように展開されるかを制御するコンポーネントです。その役割は、マルチファイルコンパイルと単一ファイルコンパイルを対比することで最もよく説明できます。単一ファイルコンパイルでは、ルートから静的到達可能なすべてのメソッドと型が、それらを定義する入力アセンブリに関係なく依存関係グラフの一部になります。マルチファイルコンパイルでは、コンパイルの一部が異なる作業単位として行われます。現在の作業単位に含まれないメソッドや型は、その依存関係を調査すべきではなく、依存関係グラフの一部であってはなりません。

2つの抽象化（コンパイルルートと、依存関係グラフの展開方法を制御するクラス）を持つ利点は、コアのコンパイルプロセスが特定のコンパイルモード（例：ライブラリをビルドしているか、マルチファイルコンパイルを行っているか）を全く意識しなくてよいことです。詳細は2つの抽象化の背後に完全にラップされており、すべてのロジックを一箇所に保ちながら、新しいコンパイルモードの定義や実験に大きな表現力を与えてくれます。例えば、1つのメソッドのみをコンパイルする单一メソッドコンパイルモードをサポートしています。このモードはコード生成のトラブルシューティングに便利です。コンパイルドライバは、コンパイラ自体を変更することなく、追加のコンパイルモード（例：単一の型とそれに関連するすべてのメソッドをコンパイルするモード）を定義できます。

関連クラス: `ICompilationRootProvider`, `CompilationModuleGroup`

依存関係の種類

依存関係グラフ分析は、ノード間のいくつかの種類の依存関係を扱うことができます:

- 静的依存関係 (**Static dependencies**): 最も一般的な依存関係です。ノード A が依存関係グラフの一部であり、ノード B を必要とする宣言した場合、ノード B も依存関係グラフの一部になります。
- 条件付き依存関係 (**Conditional dependencies**): ノード A がノード B に依存することを宣言しますが、それはノード C がグラフの一部である場合に限ります。この場合、ノード B は A と C の両方がグラフに含まれている場合にのみグラフの一部になります。
- 動的依存関係 (**Dynamic dependencies**): これらはシステム内で持つことのコストが高いため、まれにしか使用しません。ノードがグラフ内の他のノードを検査し、それらの存在に基づいてノードを注入できます。ほぼジェネリック仮想メソッド (generic virtual methods) の分析にのみ使用されます。

💡 初心者向け補足

3種類の依存関係を身近な例で説明すると:

- 静的依存関係: 「A を使うなら B も必要」 — npm の通常の依存関係と同じ
- 条件付き依存関係: 「A を使い、かつ C も使うなら B が必要」 — npm の `peerDependencies` に近い
- 動的依存関係: 「A がグラフ全体を見て、必要に応じて追加のノードを注入する」 — 実行時にしか決まらない柔軟な依存関係

実際の依存関係グラフがどのように見えるかを示すために、コンパイラ内の仮想メソッド使用追跡に関する（オプションの）最適化の例を見てみましょう:

csharp

```

abstract class Foo
{
    public abstract void VirtualMethod();
    public virtual void UnusedVirtualMethod() { }
}

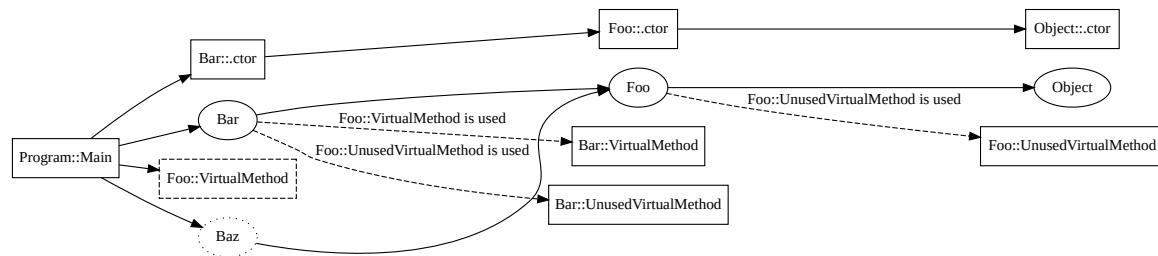
class Bar : Foo
{
    public override void VirtualMethod() { }
    public override void UnusedVirtualMethod() { }
}

class Baz : Foo
{
    public override void VirtualMethod() { }
}

class Program
{
    static int Main()
    {
        Foo f = new Bar();
        f.VirtualMethod();
        return f is Baz ? 0 : 100;
    }
}

```

上記プログラムの依存関係グラフは、おおよそ次のようにになります:



四角いノードはメソッド本体を、楕円のノードは型を、破線の四角は仮想メソッドの使用を、点線の楕円は未構築型 (unconstructed type) を表します。破線のエッジは条件付き依存関係で、条件がラベルに記されています。

- `Program::Main` は `Bar` の新しいインスタンスを作成します。そのために、GC ヒープ上にオブジェクトを割り当て、初期化のためにコンストラクタを呼び出します。したがって、`Bar` 型を表すデータ構造と `Bar` のデフォルトコンストラクタが必要です。続いてメソッドは `VirtualMethod` を呼び出します。この単純な例からは、最終的にどの具体的なメソッド本体が呼び出されるかわかります（頭の中で呼び出しを脱仮想化できます）が、一般的には知ることができないため、`Program::Main` は「`Foo::VirtualMethod` の仮想メソッド使用」にも依存すると言います。プログラムの最後の行は型チェックを行います。型チェックを行うために、生成されたコードは `Baz` 型を表すデータ構造を参照する必要があります。型チェックの興味深い点は、型を完全に記述するデータ構造を生成する必要がなく、キャストが成功するかどうかを判定するのに十分な情報だけあればよいということです。そこで、`Program::Main` は `Baz` の「未構築型データ構造 (unconstructed type data structure)」にも依存すると言います。
- `Bar` 型を表すデータ構造には、2 つの重要な種類の依存関係があります。基底型 (`Foo`) への依存（キャストを機能させるためにそのポインタが必要）と、vtable の内容です。vtable のエントリは条件付きです — 仮想メソッドが一度も呼び出されなければ、vtable に配置する必要はありません。グラフ内の状況の結果として、`Bar::VirtualMethod` のメソッド本体はグラフの一部になりますが、`Bar::UnusedVirtualMethod` はグラフに存在しないノードに条件付けられているため、含まれません。

- `Baz` を表すデータ構造は `Bar` とは少し異なります。これを「未構築型 (unconstructed type)」構造と呼びます。未構築型構造は vtable を含まないため、`Baz` には `Foo::VirtualMethod` の使用に条件付けられた `Baz::VirtualMethod` への仮想メソッド使用依存関係がありません。

条件付き依存関係を使用することで、仮想メソッドが一度も使用されていないため `Foo::UnusedVirtualMethod` と `Bar::UnusedVirtualMethod` のメソッド本体のコンパイルを回避できることに注目してください。また、`Baz` はプログラム内で一度もインスタンス化されていないため、`Baz::VirtualMethod` の生成も回避しました。`Baz` を表すデータ構造は生成しましたが、そのデータ構造はキャストの目的のみに生成されたため、依存関係グラフに `Baz::VirtualMethod` を引き込むような vtable がありません。

なお、「構築済み (constructed)」と「未構築 (unconstructed)」の型ノードは依存関係グラフ内では別々にモデル化されますが、オブジェクトの書き込み時には 1 つに統合されます。グラフが未構築と構築済みの両方の形式で型を持つ場合、構築済みの形式のみが実行可能ファイルに出力され、未構築の形式を参照している箇所は型の同一性を維持するために構築済みの形式にリダイレクトされます。

関連コンパイラスイッチ: `--dgmllog` は依存関係グラフを XML ファイルにシリアル化します。XML ファイルはグラフ内のすべてのノードをキャプチャしますが、ノードにつながる最初のエッジのみをキャプチャします（最初のエッジを知ることがほとんどの目的に十分です）。`--fulllog` はさらに大きな XML ファイルを生成し、すべてのエッジをキャプチャします。

関連ツール: [依存関係分析ビューア](#) は、マシン上のすべての ILC コンパイラプロセスが生成する ETW イベントをリッスンし、グラフをインタラクティブに探索できるツールです。

オブジェクトの書き込み

コンパイルの最終フェーズは、出力の書き込みです。コンパイルの出力はターゲット環境に依存しますが、通常は何らかのオブジェクトファイル (object file) になります。オブジェクトファイルは通常、コードまたはデータのブロブ (blob) とそれらの間のリンク（またはリロケーション (relocations)）、およびシンボル (symbols)（ブロブ内の名前付きの位置）で構成されます。リロケーションはシンボルを指し、そのシンボルは同じオブジェクトファイル内で定義されているか、別のモジュールにあります。

オブジェクトファイル形式はターゲットに非常に固有ですが、コンパイラはオブジェクトデータが関連付けられた依存関係ノードをターゲットに関係なく同じ方法で表現します — `ObjectNode` クラスを使用します。`ObjectNode` クラスは、子クラスがデータの配置先セクション（コード、読み取り専用データ、未初期化データなど）と、最も重要なこととして、データ自体 (`GetObjectData` メソッドから返される `ObjectData` クラスで表現) を指定できるようにします。

高レベルでは、オブジェクトライタの役割は、グラフ内でマークされたすべての `ObjectNode` を巡回し、それらのデータ、定義済みシンボル、他のシンボルへのリロケーションを取得して、オブジェクトファイルに格納することです。

NativeAOT コンパイラには複数のオブジェクトライタが含まれています:

- LLVM ベースのネイティブオブジェクトライター — Windows PE、Linux ELF、macOS Mach-O ファイル形式を生成可能
- WebAssembly 向けの LLVM ベースのネイティブオブジェクトライター
- CoreCLR 向けの Ready to Run 形式で、CIL とネイティブの混合実行可能ファイルを生成する Ready to Run オブジェクトライタ

関連コマンドライン引数: `--map` はオブジェクトファイルに出力されたすべてのオブジェクトノードのマップを生成します。

最適化のプラグイン可能性

完全に事前コンパイルされた環境の利点は、コンパイラがコンパイル対象のコードについてクローズドワールド仮定 (closed world assumptions) を行えることです。例えば、実行時に任意の CIL をロードする機能がない場合（`Assembly.Load` や `Reflection.Emit` を通じて）、コンパイラはインターフェースを実装する型が 1 つだけであることを確認すると、プログラム内のすべてのインターフェース呼び出しを直接呼び出しに置き換え、それによって可能になるインライン化 (inlining) などの追加の最適化を適用できます。ターゲット環境が動的コードを許可している場合、このような最適化は無効です。

初心者向け補足

クローズドワールド仮定 (closed world assumption) とは、「コンパイル時にプログラムのすべてのコードが分かっている」という前提です。この前提があると、例えばインターフェースの実装が 1 つしかなければ、仮想呼び出し (virtual call) を直接呼び出しに変換して高速化できます。Java の HotSpot JVM でも実行時に類似の最適化（モノモーフィックインライン化など）が行われますが、ILC ではこれをコンパイル時に確定できるのが強みです。

コンパイラはこのような最適化を可能にする構造を持っていますが、最適化をいつ適用すべきかについてはポリシーを維持します。これにより、完全な AOT コンパイルと混合 (JIT/インタープリタ) コード実行戦略の両方をサポートできます。ポリシーは常に抽象クラスまたはインターフェースにキャプチャされ、その実装はコンパイルドライバによって選択され、コンパイルビルダに渡されます。これにより高度な柔軟性が得られ、最適化が適用可能な条件をコンパイラにハードコーディングすることなく、コンパイルドライバからコンパイルに大きな影響力を持たせることができます。

このようなポリシーの例として、仮想メソッドテーブル (vtable) 生成ポリシーがあります。コンパイラは vtable を 2 つの方法で構築できます：遅延的 (lazily) に、または型のメタデータを読み取って型のメソッドリストに存在するすべての新しい仮想メソッドの vtable スロットを生成する方法です。前述の依存関係分析のサンプルグラフは、プログラムが動作するために生成が必要な vtable スロットと仮想メソッド本体を追跡するために条件付き依存関係をどのように使用できるかを説明していました。これはクローズドワールド仮定を必要とする最適化の例です。ポリシーは `VTableSliceProvider` クラスにキャプチャされ、ドライバが型ごとに vtable 生成ポリシーを選択できるようにします。これにより、コンパイルドライバは最適化をいつ許可するかを細かく制御できます（例：JIT が存在する場合でも、AOT コンパイルされていないプログラム部分やリフレクションからアクセス不可/不可視な型に対しては、この最適化を許可できます）。

ドライバで設定可能なポリシーは広範な領域にわたります：リフレクションメタデータの生成、脱仮想化 (devirtualization)、vtable の生成、`Exception.ToString` のスタックトレースメタデータの生成、デバッグ情報の生成、メソッド本体の IL ソースなど。

IL スキャン

ILC のもう 1 つのコンポーネントは IL スキャナ (IL scanner) です。IL スキャンは、コンパイルの前に実行できるオプションのステップです。多くの点で、IL スキャンは null/ダミーのコード生成バックエンドを持つ別のコンパイルとして動作します。IL スキャナは、ルートから依存関係グラフの一部となるすべてのメソッド本体の IL をスキャンし、それらの依存関係を展開します。IL スキャナはコード生成バックエンドが構築するのと同じ依存関係グラフを構築しますが、メソッド本体を表すグラフ内のノードにはマシンコード命令が関連付けられていません。コード生成が含まれないため、このプロセスは比較的高速ですが、結果のグラフにはコンパイルされたプログラムについて多くの貴重な洞察が含まれています。IL スキャナによって構築される依存関係グラフは、実際のコンパイルによって構築されるグラフの厳密なスーパーセット (superset) です。これは、IL スキャナがインライン化 (inlining) や脱仮想化 (devirtualization) などの最適化をモデル化しないためです。

初心者向け補足

IL スキャンは、本番コンパイルの前に行われる「偵察 (reconnaissance)」のようなものです。実際のネイティブコード生成は行わず、IL を高速にスキャンして「どの型やメソッドが必要か」「vtable のスロットはいくつ必要か」といった情報を収集します。この情報を本番コンパイルにフィードバックすることで、vtable のルックアップをインライン化するなど、より高品質なコードを生成できます。ただし、追加のステップのためコンパイル時間は長くなるというトレードオフがあります。

IL スキャナの結果は、後続のコンパイルプロセスに入力されます。例えば、IL スキャナは遅延的な vtable 生成ポリシーを使用して必要なスロットのみで vtable を構築し、スキャン終了時に vtable 内の各スロットにスロット番号を割り当てることができます。スキャン中に遅延的に計算された vtable レイアウトは、その後の実際のコンパイルプロセスで呼び出しサイトにおける vtable ルックアップのインライン化に使用できます。遅延的な vtable 生成ポリシーでは、遅延 vtable の正確なスロット割り当てがコンパイル完了まで安定しないため、呼び出しサイトでの vtable ルックアップのインライン化は不可能です。

IL スキャンプロセスはオプションであり、コンパイルループトがランタイムのコード品質よりも重要な場合、コンパイルドライバはこれをスキップできます。

関連クラス: [ILScanner](#)

基本クラスライブラリとの結合

コンパイラは、基盤となる基本クラスライブラリ（リポジトリ内の `System.Private.*` ライブラリ）と一定レベルの結合 (coupling) を持っています。結合は二重です：

- 生成されるデータ構造のバイナリ形式
- コアライブラリ内の特定のメソッドの存在に関する期待

コンパイラによって生成され、基本クラスライブラリによって使用されるバイナリ形式の例としては、実行時の型を表すデータ構造（`MethodTable`）の形式や、型に関する非必須情報（型名やメソッドのリストなど）を記述するバイト列の形式があります。これらのデータ構造はコントラクト (contract) を形成し、基本クラスライブラリ内のマネージドコードが実行時にライブラリ API（リフレクション API など）を通じてユーザーコードにリッチなサービスを提供できるようにします。これらのデータ構造の生成はオプションのものもありますが、マネージドコードの実行に必須のものもあります。

コンパイラは、生成されたコードをサポートするために、基本クラスライブラリ内の特定のよく知られたエントリーポイント (entrypoints) を呼び出す必要があります。基本クラスライブラリはこれらのメソッドを定義する必要があります。このようなエントリーポイントの例としては、数学演算中に `OverflowException` をスローするヘルパー、配列アクセス中に `IndexOutOfRangeException` をスローするヘルパー、P/Invoke マーシャリングコードの生成を支援するさまざまなヘルパー（例：ネイティブメソッドの呼び出し前後に UTF-16 文字列を ANSI に変換する）があります。

興味深い点として、コンパイラと基本クラスライブラリの結合は比較的疎であることが挙げられます（必須の部分はわずかです）。これにより、ILC で異なる基本クラスライブラリを使用できます。このような基本クラスライブラリは、通常の .NET 開発者が慣れているものとはかなり異なる外観になる可能性があります（例：`ToString` メソッドを持たない `System.Object`）が、通常の .NET が「重すぎる」と見なされる環境で型安全なコードを使用できるようにします。このような軽量コードによるさまざまな実験が過去に行われており、その一部は Windows オペレーティングシステムの一部として出荷されました。

このような代替基本クラスライブラリの例として `Test.CoreLib` があります。`Test.CoreLib` ライブラリは非常に最小限の API サーフェスを提供します。これは、ほとんど初期化を必要としないという事実と相まって、NativeAOT を新しいプラットフォームに移植する際の優れた助けになります。

ジェネリクスのコンパイル

共有ジェネリクスと正規化

プログラムが同じジェネリックメソッドまたは型を複数の参照型引数で使用する場合（例: `List<string>` と `List<object>`）、ILC はすべての互換性のあるインスタンス化 (instantiations) で共有される単一の 正規 (canonical) 形式のコードをコンパイルします。すべての参照型は `_Canon` として知られる正規表現を共有します。例えば、`List<string>` と `List<object>` は両方とも `List<_Canon>` 用にコンパイルされたコードを使用します。

💡 初心者向け補足

Java のジェネリクスでは「型消去 (type erasure)」により、`List<String>` も `List<Object>` もコンパイル後は同じ `List` になります。.NET のジェネリクスは通常「具象化 (reification)」—つまり型情報が実行時にも保持されます。しかし NativeAOT ではコードサイズ削減のため、参照型については `_Canon` という共通の正規形式を使ってコードを共有します。`List<int>` のような値型の場合はサイズが異なるため、個別にコードが生成されます。

この共有は参照型のインスタンス化にのみ適用されます。値型のインスタンス化（例: `List<int>`）は、サイズの違いがコード生成に影響するため、個別のネイティブコード本体が必要です。型引数自体が値型と参照型の混合コンポーネントを持つジェネリック型の場合、正規形式はそれを反映します。例えば、`List<KeyValuePair<int, string>>` は `List<KeyValuePair<int, _Canon>>` に正規化されます。

依存関係グラフに `List<string>.Add` のようなメソッドが含まれる場合、ILC は正規メソッド本体 `List<_Canon>.Add` への依存関係を追加し、`List<string>` インスタンス化用の [ジェネリックディクショナリ \(generic dictionary\)](#) を生成します。ILC が RyuJIT を呼び出してメソッドをコンパイルする際、正規形式（例: `List<_Canon>.Add`）を渡します。

ランタイム決定型

共有ジェネリックコードの依存関係分析には、型の正規形式もインスタンス化されていない形式も単独では不十分です。正規形式はパラメータの同一性を失います（`T` と `U` の両方が `_Canon` になる）。一方、シグネチャ変数（`!10`、`!11`）を持つインスタンス化されていない形式は、`sizeof(T)` のような操作に必要な具体的な型情報が不足しています。

これを解決するために、ILC は ランタイム決定型 (*runtime-determined types*) (`RuntimeDeterminedType`) を使用して、実行時にのみ解決される型を表現します。`RuntimeDeterminedType` は正規型とそれが由来するジェネリックパラメータの両方をキャプチャします: 内部的には `DefType` (例: `_Canon`) と `GenericParameterDesc` (例: `Foo<T>` の `T`) の両方への参照を保持します。

例えば、`List<T>` を参照する `Foo<T>.Method()` を分析する際、ILC はこれを `List<T_System._Canon>` として表現します。ここで型引数は `_Canon` と `T` を組み合わせた `RuntimeDeterminedType` です。

ジェネリック仮想メソッド

ジェネリック仮想メソッド (GVM: Generic Virtual Methods) は、ターゲットの実装がオブジェクトの実行時型に依存するため、実行時にメソッドポインタとジェネリックディクショナリ (generic dictionary) の両方を解決する必要があります。次の例を考えてみましょう:

```
csharp
abstract class Base
{
    public abstract void Method<T>();
}

class Derived : Base
{
    public override void Method<T>() { }
}
```

`baseRef.Method<string>()` のような呼び出しサイトでは、ランタイムはどの実装を呼び出すかを決定するために、`baseRef` の実行時型と型引数の情報の両方が必要です。

ILC は依存関係グラフで *動的依存関係 (dynamic dependencies)* を使用して GVM を処理します。コンパイラが GVM 呼び出しを検出すると、呼び出されたメソッドの正規形式に対する `GVMDependenciesNode` を作成します。このノードは動的依存関係を持ち、プログラム内でどの型が構築されているかを観察し、GVM をオーバーライドする可能性のある各型について、適切な実装が確実にコンパイルされるようにします。

例えば、`Base.Method<__Canon>` の `GVMDependenciesNode` が `Derived` が構築されていることを確認すると、`Derived.Method<__Canon>` への依存関係を追加し、ベーススロットから派生実装へのマッピングが記録されることを保証します。実行時には、GVM テーブルにより、オブジェクトの型とメソッドのインスタンス化に基づいて正しい実装をルックアップできます。

これは ILC で動的依存関係が使用される数少ない場所の 1 つです。なぜなら、可能な GVM 実装のセットは、単一のノードを個別に調べるだけでは決定できないからです。

シャドウメソッドノード

「シャドウ (Shadow)」メソッドノードは、依存関係追跡のみを目的として、部分的または完全にインスタンス化されたジェネリックメソッドを表します。これらのノード自体はコードを生成しません。

コード生成は常にメソッドの正規形式（`List<__Canon>.Add` など）に対して行われます。ただし、`List<string>.Add` や `List<object>.Add` への呼び出しがある場合、コンパイラはインスタンス化固有の依存関係を生成する必要がある場合があります。シャドウメソッドノードは、個別のコードは寄与しないものの、依存関係追跡の目的でこれらの特定のメソッドインスタンス化を表します。

`ShadowConcreteMethodNode` と `ShadowNonConcreteMethodNode` はどちらも `ShadowMethodNode` を拡張しています。`ShadowMethodNode` は正規メソッドの依存関係を調べることで動作します。`INodeWithRuntimeDeterminedDependencies` を実装する依存関係は、シャドウノードの型/メソッド引数で インスタンス化 (*instantiated*) され、抽象的な依存関係（例: 「`List<T>` の MethodTable」）を具体的なもの（例: 「`List<string>` の MethodTable」）に変換します。これにより、シャドウノードはジェネリックディクショナリに必要な依存関係をオブジェクトファイルに寄与できます。ジェネリックディクショナリノードは依存関係を直接報告できません。なぜなら、コンパイラは同じ辞書を使用するすべてのメソッドを発見するまで、すべての依存関係がわからないからです。シャドウメソッドは、コンパイルされた各メソッドに応じてグラフをインクリメンタルに拡張することを可能にします。

`ShadowConcreteMethodNode` は、依存関係分析のために `List<string>.Add` のような完全にインスタンス化されたメソッドを表します。

`ShadowNonConcreteMethodNode` は、まだ共有されている部分的にインスタンス化されたメソッドを表します。例えば、`Foo<string>.Bar<__Canon>()` は `Foo<__Canon>.Bar<__Canon>()` に裏付けられていますが、`T` が `string` であることを知っています。これにより、メソッドの型パラメータがまだ開いている場合でも、型の型パラメータにのみ関係する依存関係（例: `typeof(T)`、`new T[]`）を解決できます。

コンパイラ生成メソッド本体

入力アセンブリの形でユーザーが提供するコードのコンパイルに加えて、コンパイラはコンパイラ内で生成されるさまざまなヘルパーもコンパイルする必要があります。ヘルパーは、上位レベルの .NET の構成要素を、基盤となるコード生成バックエンドが理解できる概念に下位変換 (*lowering*) するために使用されます。これらのヘルパーは、ディスク上のアセンブリの IL に物理的に裏付けられることなく、IL コードとしてその場で出力されます。上位レベルの概念を通常の IL として表現することで、各コード生成バックエンドに上位レベルの概念を実装する必要がなくなります (IL はすべてのバックエンドが理解するものなので、一度だけ実装すればよい)。

ヘルパーは以下のようなさまざまな目的に使用されます:

- デリゲート (delegate) の呼び出しをサポートするヘルパー
- P/Invoke のパラメータと戻り値のマーシャリング (marshalling) をサポートするヘルパー
- `ValueType.GetHashCode` と `ValueType.Equals` をサポートするヘルパー
- リフレクション (reflection) をサポートするヘルパー: 例えば `Assembly.GetExecutingAssembly`

関連クラス: `ILEMitter` , `ILStubMethod`

関連 ILC コマンドラインスイッチ: `--ildump` は、生成されたすべての IL をファイルにダンプし、それにデバッグ情報をマッピングします (実行時に生成された IL をステップ実行してソースデバッグが可能になります)。

マネージド型システムの概要

原文

この章の原文は [Managed Type System Overview](#) です。

著者: Michal Strehovsky ([@MichalStrehovsky](#)) - 2016

はじめに

マネージド型システム (Managed Type System) は、AOT および IL 検証のための新世代の .NET ツールの主要コンポーネントです。プログラム内のモジュール、型、メソッド、およびフィールドを表現し、型システムの利用者がさまざまな興味深い質問に対する回答を得られるよう、より高レベルなサービスを提供します。

マネージド型システムは、[CoreCLR の型システム](#) を C# で書き直したものに相当します。私たちはランタイム機能を C# で実装したいとかねてから考えていました。マネージド型システムは、それを可能にするインフラストラクチャです。

💡 初心者向け補足

型システム (Type System) とは、プログラム中の「型」に関する情報を管理する仕組みです。Java でいえば、クラスローダーやリフレクション API が提供する型情報に相当します。.NET のマネージド型システムは、型のメタデータ（フィールド構成、継承関係、インターフェース実装など）を読み取り、ランタイムやコンパイラが必要とする高レベルな情報を計算します。

型システムが提供する高レベルなサービスには、以下のようなものがあります:

- メタデータからの新しい型の読み込み
- 特定の型が実装するインターフェース (interface) の集合の計算
- 静的フィールドおよびインスタンスフィールドのレイアウト (layout) の計算 (個々のフィールドへのオフセット割り当て)
- 型の静的およびインスタンス GC レイアウトの計算 (オブジェクト/クラステータ内での GC ポインタの識別)
- VTable レイアウトの計算 (仮想メソッドへのスロット割り当て) および仮想メソッドのスロットへの解決
- ある型を別の型の場所に格納できるかどうかの判定

型システムの設計を駆動する3つの主要なテーマがあります:

1. 低オーバーヘッドと高パフォーマンス
2. 並行性 (Concurrency)
3. 拡張性と再利用性

低オーバーヘッドは遅延読み込み (lazy loading) によって達成されます。型にフィールド、各種属性、名前などを先行的に (eagerly) 設定するのではなく、これらは基盤となるデータソース (メタデータ) からオンデマンドで読み取られます。キャッシングは控えめに使用されます。

必要に応じて、ポリモーフィズムやオブジェクト階層の代わりに、パーシャルクラス (partial class)、拡張メソッド (extension method)、およびプラガブルアルゴリズム (pluggable algorithm) が目標 3 を達成するために使用されます。型システムの再利用性はソ

ースレベルで行われます（異なるファイルセットをインクルードして異なる機能を得る）。これにより、目標1から離れることなく拡張性を実現できます。

型システムは、その最も純粋な形態（すなわち、パーシャルクラスの拡張なし）では、[ECMA-335 仕様](#)で定義されていない概念の導入を避けようとしています。この仕様は、本ドキュメントを読む前に読んでおくことが推奨される前提知識であり、本ドキュメントで使用されるさまざまな用語の定義を提供しています。

メタデータとの関係

メタデータ (metadata) (ECMA-335 仕様で記述されているファイルフォーマットなど) は型システムと密接な関係がありますが、この2つには明確な違いがあります。メタデータは型の物理的な形状を記述します（例：型の基底クラスは何か、どのようなフィールドを持つか）が、型システムはその形状の上により高レベルな概念を構築します（例：型のインスタンスをランタイムで格納するために何バイト必要か、継承されたものを含めて型がどのインターフェースを実装しているか）。

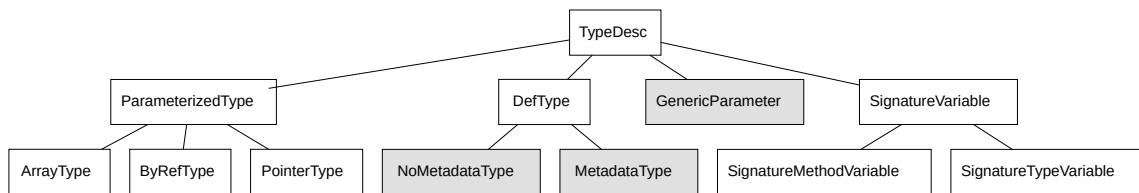
💡 初心者向け補足

メタデータと型システムの関係は、設計図と建物の関係に似ています。メタデータ（設計図）は「このクラスにはどんなフィールドがあるか」「基底クラスは何か」といった静的な定義情報を提供します。一方、型システム（建物の施工計画）は、その情報をもとに「実際にメモリ上でどのように配置するか」「どのインターフェースを実装しているか」といった実行時に必要な高レベルの情報を計算します。

型システムは基盤となるメタデータの大部分へのアクセスを提供しますが、その取得方法は抽象化されています。これにより、他のフォーマットのメタデータに裏付けられた型やメンバー、あるいは物理的なフォーマットをまったく持たないもの（配列型のメソッドなど）を、同じ型システムコンテキスト内で表現できるようになっています。

型システムのクラス階層

型システム内で型を表すクラスは以下のとおりです：



この階層のほとんどのクラスは、型システムの利用者が派生させることを想定しておらず、多くはそれを防ぐために `sealed` になっています。

拡張可能なクラス（実際には抽象クラス）は、上の図で暗い背景で表示されています。具象クラスは、ECMA-335 モジュールファイルからメタデータを読み取るなどのロジックに基づいて、抽象メソッドおよび仮想メソッドの実装を提供する必要があります（型システムは、例えば `EcmaType` において `MetadataType` のそのような実装をすでに提供しています）。理想的には、型システムの利用者は抽象クラスを操作し、具象クラスは新しいインスタンスを作成するときにのみ使用すべきです。`EcmaType` のような具象実装型へのキャストは推奨されません。

型システムのクラス

以下のセクションでは、型システム内で型を表すクラスについて簡単に説明します。

TypeDesc

`TypeDesc` は型システム内のすべての型の基底クラスです。すべてのクラスがサポートしなければならない操作のリストを定義します。すべての操作が `TypeDesc` のすべての子クラスに対して意味を持つわけではありません（例えば、ポインタ型に対してメソッドのリストを要求することは意味がありません）が、各子クラスに対して意味のある実装が提供されるよう配慮されています（すなわち、ポインタ型のメソッドリストは空になります）。

ParameterizedType (ArrayType, ByRefType, PointerType)

これらは単一のパラメータを持つ構築型 (constructed type) です：

- 配列 (array) (多次元配列、またはベクター (vector) — 暗黙のゼロ下限を持つ1次元配列)
- マネージド参照 (managed reference)
- アンマネージドポインタ型 (unmanaged pointer type)

ランク 1 の多次元配列とベクターの区別は非常に重要であり、型システムの利用者にとって潜在的なバグの原因となります。型システムの利用者は特に注意を払う必要があります。

💡 初心者向け補足

「ランク 1 の多次元配列」と「ベクター」の区別は、.NET 特有の重要な概念です。C# で `int[]` と書くとベクター（1次元でインデックスが 0 から始まる配列）が作られます、`Array.CreateInstance(typeof(int), new int[]{10})` で作った配列はランク 1 の多次元配列であり、型としては異なります。Java には対応する区別はありませんが、.NET の型システムではこの2つを厳密に区別する必要があります。

DefType (NoMetadataType, MetadataType)

`DefType` は値型 (value type)、インターフェース (interface)、またはクラス (class) を表します。`DefType` のインスタンスのほとんどは `MetadataType` (型を完全に記述する何らかの具体的なメタデータに基づく型) の子クラスのものになりますが、完全なメタデータが利用できなくなるシナリオもあります。その場合、限定的な情報 (GC ヒープ上の型のインスタンスが占めるバイト数や、その型が値型かどうかなど) のみが利用可能です。型システムがそのような型に対して操作できることは重要です。例えば、限定的なメタデータを持つ型が完全なメタデータを持つ型の基底型であることが可能であり、フィールドレイアウトアルゴリズムがそのような型のフィールドレイアウトを計算できる必要があります。

GenericParameter

ジェネリックパラメータ (generic parameter) を、その制約 (constraints) とともに表します。ジェネリック定義 (generic definition) は、ジェネリックパラメータに対するインスタンス化 (instantiation) として表現されます。

.NET リフレクション型システムに詳しい読者への注意: .NET リフレクション型システムはジェネリック定義（例: `List<T>`）とジェネリック型のオープンインスタンス化 (open instantiation)（例: `List<!0>`）を区別しませんが、マネージド型システムはこの2つを区別します。この区別は IL メソッド本体内からのメンバー参照を表現する際に重要です。例えば、LDTOKEN 命令を使用した `List<T>.Add` への IL 参照は常に未インスタンス化の定義を参照すべきですが、`List<!0>.Add` への参照はシグネチャ変数を置換した後の具象メソッドを参照します。

SignatureVariable (SignatureTypeVariable, SignatureMethodVariable)

シグネチャ変数 (signature variable) は、システム内の他の型によって置換可能な変数を表します。ジェネリックパラメータとは異なります（例えば、制約や変性 (variance) を持たないため）。これらは単に、インスタンス化 (instantiation) と呼ばれるプロセスの一部として他の型に置き換えられるプレースホルダです。シグネチャ変数は、インスタンス化コンテキスト内の位置を参照するインデックスを持ちます。

その他の型システムクラス

型システムの各使用は、型システムコンテキスト (type system context) の作成から始まります。型システムコンテキストは、すべての型が参照同一性 (reference identity) を共有する型ユニバースを表します（2つの `TypeDesc` オブジェクトが同一の型を表すのは、それらが同じオブジェクトインスタンスである場合に限ります）。型システムコンテキストは、ユニバース内のすべてのモジュールおよび構築型を解決するために使用されます。型システムコンテキストの外で構築型の新しいインスタンスを作成することは不正です。

💡 初心者向け補足

型システムコンテキストは、Java でいえば `ClassLoader` の役割に近いものです。あるコンテキスト内では、同じ型は必ず同じオブジェクトインスタンスで表されます（参照同一性）。これは Java の `ClassLoader` が同じクラスに対して同じ `Class` オブジェクトを返すのと似ています。この仕組みにより、型の比較を `==`（参照比較）で高速に行うことができます。

型システム内の他の重要なクラスとして、`MethodDesc`（型システム内のメソッドを表す）と `FieldDesc`（型システム内のフィールドを表す）があります。`ModuleDesc` は単一のモジュールを記述し、そのモジュールがアセンブリである場合はオプションで `IAssemblyDesc` インターフェースを実装できます。`ModuleDesc` は通常、モジュール内の型/メソッド/フィールド定義のオーナーです。それらの参照同一性を維持するのは `ModuleDesc` の責任です。

プラガブルアルゴリズム

型システムが提供するほとんどのアルゴリズム（例: フィールドレイアウトアルゴリズム）はプラガブル (pluggable) です。型システムコンテキストは、異なる実装を提供することでアルゴリズムの選択に影響を与えることができます。

アルゴリズムは、パーシャルクラスやソースインクルードでは十分でない場所で拡張メカニズムとして使用されます。特定のアルゴリズムの選択は複数の要因に依存する可能性があり、型システムの利用者はランタイムで決定される特定の条件セットに応じて複数のアルゴリズムを使い分けたい場合があります（例: 通常の `DefType` のランタイムインターフェースのリストの計算 vs. 配列型のランタイムインターフェースの計算）。

型システム内のハッシュコード

型システムの興味深い特性として、システム内で表現される任意の型やメソッドに対して、コンパイル時とランタイムの両方で確実に計算できるハッシュコードを算出する能力があります。コンパイル時とランタイムの両方で同じハッシュコードが利用できることを活用して、AOT コンパイルされたコードで高パフォーマンスなルックアップテーブルを構築しています。ハッシュコードは型名から計算され、ランタイムデータ構造の一部として保存されるため、コンパイラによって型名が最適化により除去された状況でも利用可能です。

型システムからの例外のスロー

型システム内からの例外のスローは、単純な `throw` 文よりもやや複雑です。これは、型システムがさまざまな場所で使用可能なように設計されており、それぞれが例外のスロー方法について異なる要件を持つ可能性があるためです。例えば、型システムがランタイムからインクルードされている場合、型の読み込みに失敗すると `System.TypeLoadException` がスローされるべきです。一方、コンパイラや IL 検証器で型の読み込みエラーが発生した場合、`System.TypeLoadException` はコンパイラを構成するマネージドアセンブリの実際の問題と区別がつかなくなります。したがって、異なる例外がスローされるべきです。

型システム内の例外スローは `ThrowHelper` クラスにラップされています。型システムの利用者がこのクラスとそのメソッドの定義を提供します。メソッドは、どの例外型がスローされるかを制御します。

型システムは、`TypeSystemException` 例外基底クラスから派生した例外をスローする `ThrowHelper` クラスのデフォルト実装を提供しています。このデフォルト実装は、非ランタイムシナリオでの使用に適しています。

例外メッセージには文字列 ID が割り当てられ、スローヘルパーによっても消費されます。この間接化は、コンパイラシナリオをサポートするために必要です。AOT コンパイル中に型の読み込み例外が発生した場合、AOT コンパイラには2つのタスクがあります—これが発生したことをユーザーに警告するための警告を出すことと、問題のある型がアクセスされたときにランタイムでこの例外をスローするメソッド本体を生成する可能性があることです。コンパイラのローカリゼーション (localization) は、コンパイラ出力がリンクするクラスライブラリのローカリゼーションと一致しない場合があります。実際の例外メッセージを文字列 ID を通じて間接化することで、これをラップできます。型システムの利用者は、この機能が必要な型システム外の場所でもスローヘルパーを再利用できます。

物理アーキテクチャ

型システムの実装は以下の場所にあります:

- `src/coreclr/tools/Common/TypeSystem/Common` : 共通型システムの大部分がここにあります
- `src/coreclr/tools/Common/TypeSystem/Ecma` : ECMA-335 モジュールファイルからメタデータを読み取る `MetadataType` 、`MethodDesc` 、`FieldDesc` などの具象実装がここにあります
- `src/coreclr/tools/aot/ILCompiler.TypeSystem.ReadyToRun.Tests` : 型システムの動作と機能について理解を深めることができます。ユニットテストがここにあります。コードについて学ぶための良い出発点です。

CoreCLR 型システムとの注目すべき違い

- マネージド型システムでは、`MethodDesc` は可能な場合に正確なジェネリックインスタンス化 (exact generic instantiation) を持つります。マネージド型システムにおけるコード共有ポリシー (code sharing policy) はプラガブルアルゴリズムの1つであり、`MethodDesc` の同一性 (identity) に影響を与えません。CoreCLR 型システムでは、コード共有ポリシーが `MethodDesc` の同一性と結合しています。この違いがどのように現れるかの例については、[こちらの PR](#) を参照してください。

Ready to Run PerfMap フォーマット

原文

この章の原文は [Ready to run PerfMap format](#) です。

従来、.NET ではシンボル (symbol) は PDB を使って記述されてきました。PDB は、JIT がコンパイルするコードについて、IL をソースの行にマッピングするために使用されます。JIT は通常、IL からネイティブアドレス (native address) へのマッピングデータを出力し、これによりシンボル解決 (symbolication) が可能になります。

💡 初心者向け補足

PDB (Program Database) は、デバッグ情報を格納するファイル形式です。Java の `.class` ファイルにデバッグ情報が埋め込まれるのと似た役割ですが、.NET では別ファイルとして管理されます。IL (Intermediate Language) は .NET のバイトコードで、Java のバイトコードに相当します。JIT コンパイラ (JIT compiler) が IL をネイティブコード (native code) に変換して実行します。

しかし、Ready to Run (R2R) では、実行時にこの IL からネイティブコードへの変換が行われません。そのため、R2R イメージ (image) を出力するツールは、ソースとネイティブアドレスの間のマッピングを容易にするための補助的な成果物 (artifact) を出力する必要があることがあります。Ready to Run PerfMap フォーマットは、そのようなマップの一つを記述するものです。ここでは、ソースコード内の各メソッド (method) が R2R イメージ内の領域 (region) に関連付けられます。これにより、そのイメージから実行される任意の領域を、ソースレベルのメソッドに紐づけることができます。これはパフォーマンス調査のためのスタックシンボル解決 (stack symbolication) のようなタスクを容易にしますが、ソース行レベルでのデバッグのようなタスクには適していません。

💡 初心者向け補足

Ready to Run (R2R) は、.NET アプリケーションの起動を高速化するための AOT (Ahead-of-Time) コンパイル技術です。通常の .NET では、実行時に JIT が IL をネイティブコードに変換しますが、R2R ではあらかじめネイティブコードを生成しておき、起動時間を短縮します。しかし、事前にコンパイルされているため、「どのネイティブコードがどのメソッドに対応するか」という情報を別途保持する必要があります。それが PerfMap の役割です。

バージョン 1

バージョン 1 の R2R PerfMap は、通常 `.ni.r2rmap` という拡張子のファイルとして存在します。プレーンテキストの UTF-8 形式で、各エントリ (entry) は個別の行に記述されます。各エントリは、イメージの先頭からの相対オフセット (offset)、長さ (length)、名前 (name) の3つ組 (triplet) で構成されます。ファイルは以下のように配置されます。

ヘッダー

ヘッダー (header) はファイルの先頭に位置し、特別なエントリで構成されます。各エントリは、RVA の代わりにエントリ内の情報の種類を示す 4 バイト整数の token、常に 0 の長さ、およびエントリデータを含みます。エントリは以下の順序で出力されます。

トークン	説明
0xFFFFFFFFFF	PerfMap を R2R イメージと関連付けるための署名 (signature) を表す 16 バイトのシーケンス。
0xFFFFFFFFFE	出力される PerfMap のバージョンを表す符号なし 4 バイト整数。
0xFFFFFFFFFD	イメージが対象とする OS を表す符号なし 4 バイト整数。 列挙値のセクション を参照。
0xFFFFFFFFFC	イメージが対象とするアーキテクチャ (architecture) を表す符号なし 4 バイト整数。 列挙値のセクション を参照。
0xFFFFFFFFFB	イメージの ABI を表す符号なし 4 バイト整数。 列挙値のセクション を参照。

これらのエントリには、ツールにとって有用なコンパイルに関する情報と、"[Ready to Run フォーマット - デバッグディレクトリエンタリ](#)"に記述されているように PerfMap をイメージと関連付けるために使用できる識別子 (identifier) が含まれています。

💡 初心者向け補足

RVA (Relative Virtual Address) は、実行可能ファイルがメモリにロードされたときの、ベースアドレスからの相対的なアドレスです。ABI (Application Binary Interface) は、コンパイルされたコードがオペレーティングシステムやハードウェアとやり取りするための低レベルなインターフェース仕様です。呼び出し規約 (calling convention) やデータのメモリ上の配置方法などを定義しています。

コンテンツ

各エントリは3つ組 (triplet) です。イメージの先頭からのメソッドの相対アドレス (relative address) を表す符号なし 4 バイト整数、ネイティブコードが使用するバイト数を表す符号なし 2 バイト整数、そしてメソッドの名前です。ヘッダーの後に 1 行に 1 エントリが記述され、コールド/ホットパス分割 (cold/hot path splitting) が行われた場合、同一メソッドが複数回出現することがあります。

ヘッダーで使用される列挙値

```
PerfMapArchitectureToken
Unknown      = 0,
ARM         = 1,
ARM64       = 2,
X64         = 3,
X86         = 4,
```

```
PerfMapOSToken
Unknown      = 0,
Windows     = 1,
Linux       = 2,
OSX         = 3,
FreeBSD     = 4,
```

```
NetBSD      = 5,  
SunOS       = 6,
```

```
PerfMapAbiToken  
Unknown = 0,  
Default = 1,  
Armel = 2,
```

ReadyToRun ファイルフォーマット

原文

この章の原文は [ReadyToRun File Format](#) です。

改訂履歴:

- 1.1 - [Jan Kotas](#) - 2015
- 3.1 - [Tomas Rylek](#) - 2019
- 4.1 - [Tomas Rylek](#) - 2020
- 5.3 - [Tomas Rylek](#) - 2021
- 5.4 - [David Wrighton](#) - 2021
- 6.3 - [David Wrighton](#) - 2022

はじめに

本ドキュメントでは、2019年6月時点での CoreCLR に実装されている ReadyToRun フォーマット (R2R) 3.1、およびコンポジット (Composite) R2R ファイルフォーマットのサポートのためにまだ実装されていない拡張提案 4.1 について説明します。コンポジット R2R ファイルフォーマットは、以前のリビジョンで定義された従来の R2R ファイルフォーマットと基本的に同じ構造を持ちますが、出力ファイルがより多くの入力 MSIL アセンブリを論理的な単位としてまとめてコンパイルしたものを表す点が異なります。

💡 初心者向け補足

ReadyToRun (R2R) は、.NET の事前コンパイル (AOT: Ahead-Of-Time) 技術の一つです。通常 .NET アプリケーションは実行時に JIT (Just-In-Time) コンパイラによって中間言語 (IL) からネイティブコードに変換されますが、R2R ではビルド時にあらかじめネイティブコードを生成しておくことで、起動時間を短縮できます。Java でいう AOT コンパイル (GraalVM の Native Image など) に似た概念です。

PE ヘッダーと CLI ヘッダー

单一ファイル ReadyToRun イメージは、ECMA-335 に記述された CLI ファイルフォーマットに準拠しますが、以下のカスタマイズが加えられています:

- PE ファイルは常にプラットフォーム固有です
- CLI ヘッダーの Flags フィールドに `COMIMAGE_FLAGS_IL_LIBRARY` (0x00000004) ビットが設定されています
- CLI ヘッダーの `ManagedNativeHeader` が READYTORUN_HEADER を指します

COFF ヘッダーの COM ディスクリプタ (descriptor) データディレクトリ項目が指す COR ヘッダーと ECMA 335 メタデータは、生成元の入力 IL および MSIL メタデータの完全なコピーを表します。

コンポジット R2R ファイルは現在、ネイティブエンベロープ (envelope) として Windows PE 実行可能ファイルフォーマットに準拠しています。今後は、ネイティブエンベロープとして [プラットフォームネイティブの実行可能フォーマットのサポートを段階的に追加する予定](#) です (Linux では ELF、macOS では MachO)。ファイル内にグローバルな CLI / COR ヘッダーが存在しますが、それは PDB 生成を

容易にするためだけのものであり、CoreCLR ランタイムによる使用には関与しません。ReadyToRun ヘッダー構造体は、よく知られたエクスポートシンボル `RTR_HEADER` によって指され、`READYTORUN_FLAG_COMPOSITE` フラグが設定されています。

入力 MSIL メタデータと IL ストリームは、コンポジット R2R ファイルに埋め込むことも、ディスク上の個別ファイルとして残すこともあります。MSIL が埋め込まれている場合、個々のコンポーネントアセンブリの「実際の」メタデータは、R2R セクション `ComponentAssemblies` を通じてアクセスされます。

スタンドアロン MSIL ファイルは、MSIL 埋め込みなしのコンポジット R2R 実行可能ファイルの IL とメタデータのソースとして使用されます。これらはコンポジット R2R 実行可能ファイルの隣の出力フォルダにコピーされ、コンパイラによって書き換えられ、所有者であるコンポジット R2R 実行可能ファイルへの転送情報 (セクション `OwnerCompositeExecutable`) を含む正式な ReadyToRun ヘッダーが付加されます。

初心者向け補足

PE (Portable Executable) は Windows で使われる実行可能ファイルのフォーマットで、.exe や .dll ファイルの構造を定義します。CLI (Common Language Infrastructure) は .NET の実行基盤であり、ECMA-335 規格で定義されています。R2R イメージはこの PE/CLI フォーマットの上に、事前コンパイルされたネイティブコードの情報を追加する形で構成されています。

デバッグディレクトリへの追加

現在出荷されている PE エンベロープ (单一封装およびコンポジットの両方) には、デバッグディレクトリに追加のデバッグ情報のコードを含めることができます。R2R イメージに固有のエントリの一つとして、R2R PerfMap 用のものがあります。補助ファイルのフォーマットは [R2R perfmap フォーマット](#) に記述されており、対応するデバッグディレクトリエントリは [PE COFF](#) に記述されています。

将来の改善点

現在のフォーマットには以下の制限があります:

- IL メタデータからの型ロード: 現在、すべての型は実行時に IL メタデータから構築されます。これはサイズを肥大化させ (イメージからの完全なメタデータの除去を妨げ)、脆弱性があります (固定のフィールドレイアウトアルゴリズムを前提としています)。ランタイムの型ロードに最適化されたコンパクトな型レイアウト記述を持つ新しいセクションが必要です (CTL と類似の概念)。
- デバッグ情報のサイズ: デバッグ情報がイメージを不必要に肥大化させています。このソリューションは、現在のデスクトップ/CoreCLR デバッグパイプラインとの互換性のために選択されました。理想的には、デバッグ情報は別ファイルに保存されるべきです。

構造体

構造体および付随する定数は、[readytorun.h](#) ヘッダーファイルで定義されています。基本的に、R2R 実行可能イメージ全体は、ネイティブ実行可能エンベロープのエクスポートセクションにあるよく知られたエクスポート `RTR_HEADER` が指す `READYTORUN_HEADER` シングルトン (singleton) を通じてアドレスされます。

单一封装 R2R 実行可能ファイルの場合、すべてのイメージセクションを表す 1 つのヘッダーのみが存在します。コンポジットおよびシングル exe の場合、グローバルな `READYTORUN_HEADER` には、コンポジット R2R イメージを構成するコンポーネントアセンブリを表す `ComponentAssemblies` 型のセクションが含まれます。このテーブルは `READYTORUN_MANIFEST_METADATA` テーブルと並列 (同じ

インデックスを使用) です。各 [READYTORUN_SECTION_ASSEMBLIES_ENTRY](#) レコードは、特定のアセンブリに固有のセクションを表す [READYTORUN_CORE_HEADER](#) 可変長構造体を指します。

READYTORUN_HEADER

C++

```
struct READYTORUN_HEADER
{
    DWORD             Signature;      // READYTORUN_SIGNATURE
    USHORT            MajorVersion;   // READYTORUN_VERSION_XXX
    USHORT            MinorVersion;

    READYTORUN_CORE_HEADER CoreHeader;
}
```

READYTORUN_HEADER::Signature

常に 0x00525452 に設定されます (RTR の ASCII エンコーディング)。このシグネチャは、ReadyToRun イメージを ManagedNativeHeader を持つ他の CLI イメージ (例: NGen イメージ) と区別するために使用できます。

READYTORUN_HEADER::MajorVersion/MinorVersion

現在のフォーマットバージョンは 3.1 です。MajorVersion の増加はファイルフォーマットの破壊的変更を意味します。MinorVersion の増加は互換性のあるファイルフォーマット変更を意味します。

例: ランタイムがサポートする最高バージョンが 2.3 であると仮定します。ランタイムはバージョン 2.9 のイメージからネイティブコードを正常に実行できるべきです。ランタイムはバージョン 3.0 のイメージからネイティブコードを実行することを拒否すべきです。

💡 初心者向け補足

バージョン管理の考え方は一般的なセマンティックバージョニング (Semantic Versioning) に似ています。メジャーバージョンが同じであれば後方互換性が保たれ、マイナーバージョンが高いイメージでも実行できます。しかし、メジャーバージョンが異なると互換性がなくなります。

READYTORUN_CORE_HEADER

C++

```
struct READYTORUN_CORE_HEADER
{
    DWORD             Flags;          // READYTORUN_FLAG_XXX
    DWORD            NumberOfSections;
```

```

// Array of sections follows. The array entries are sorted by Type
// READYTORUN_SECTION Sections[];
};


```

READYTORUN_CORE_HEADER::Flags

フラグ	値	説明
READYTORUN_FLAG_PLATFORM_NEUTRAL_SOURCE	0x00000001	元の IL イメージがプラットフォーム中立であった場合に設定されます。プラットフォーム中立性はアセンブリ名の一部です。このフラグは、元の完全なアセンブリ名を再構築するために使用できます。
READYTORUN_FLAG_COMPOSITE	0x00000002	イメージが、多数の入力 MSIL アセンブリの結合コンパイルの結果であるコンポジット R2R ファイルを表します。
READYTORUN_FLAG_PARTIAL	0x00000004	
READYTORUN_FLAG_NONSHARED_PINVOKE_STUBS	0x00000008	イメージにコンパイルされた PInvoke スタブは共有不可です（シークレット パラメータなし）。
READYTORUN_FLAG_EMBEDDED_MSIL	0x00000010	入力 MSIL が R2R イメージに埋め込まれています。
READYTORUN_FLAG_COMPONENT	0x00000020	これはコンポジット R2R イメージのコンポーネントアセンブリです。
READYTORUN_FLAG_MULTIMODULE_VERSION_BUBBLE	0x00000040	この R2R モジュールには、バージョン バブル (version bubble) 内に複数のモジュールがあります（バージョン 6.3 より前では、すべてのモジュールがこの特性を持つ可能性があると仮定されます）。
READYTORUN_FLAG_UNRELATED_R2R_CODE	0x00000080	この R2R モジュールには、このモジュールに自然にエンコードされないコードが含まれています。
READYTORUN_FLAG_PLATFORM_NATIVE_IMAGE	0x00000100	所有するコンポジット実行可能ファイルがプラットフォームネイティブフォーマットです。

READYTORUN_SECTION

C++

```
struct READYTORUN_SECTION
{
    DWORD             Type;           // READYTORUN_SECTION_XXX
    IMAGE_DATA_DIRECTORY Section;
};
```

READYTORUN_CORE_HEADER 構造体の直後に **READYTORUN_SECTION** レコードの配列が続き、個々の R2R セクションを表します。配列の要素数は **READYTORUN_HEADER::NumberOfSections** です。各レコードはセクションタイプとバイナリ内のその位置を含みます。配列はバイナリサーチ (binary search) を可能にするためにセクションタイプでソートされています。

このセットアップにより、ファイルフォーマットの破壊的変更なしに、新しいまたはオプションのセクションタイプを追加したり、既存のセクションタイプを廃止したりできます。ランタイムは、ReadyToRun ファイルをロードして実行するために、すべてのセクションタイプを理解する必要はありません。

以下のセクションタイプが定義されており、本ドキュメントの後半で説明されます:

ReadyToRunSectionType	値	スコープ (コンポーネントアセンブリ / イメージ全体)
CompilerIdentifier	100	イメージ
ImportSections	101	イメージ
RuntimeFunctions	102	イメージ
MethodDefEntryPoints	103	アセンブリ
ExceptionInfo	104	アセンブリ
DebugInfo	105	アセンブリ
DelayLoadMethodCallThunks	106	アセンブリ
AvailableTypes	107	(廃止 - 古いフォーマットで使用)
AvailableTypes	108	アセンブリ
InstanceMethodEntryPoints	109	イメージ
InliningInfo	110	アセンブリ (V2.1 で追加)
ProfileDataInfo	111	イメージ (V2.2 で追加)
ManifestMetadata	112	イメージ (V2.3 で追加)
AttributePresence	113	アセンブリ (V3.1 で追加)
InliningInfo2	114	イメージ (V4.1 で追加)

ReadyToRunSectionType	値	スコープ (コンポーネントアセンブリ / イメージ全体)
ComponentAssemblies	115	イメージ (V4.1 で追加)
OwnerCompositeExecutable	116	イメージ (V4.1 で追加)
PgoInstrumentationData	117	イメージ (V5.2 で追加)
ManifestAssemblyMvids	118	イメージ (V5.3 で追加)
CrossModuleInlineInfo	119	イメージ (V6.3 で追加)
HotColdMap	120	イメージ (V8.0 で追加)
MethodIsGenericMap	121	アセンブリ (V9.0 で追加)
EnclosingTypeMap	122	アセンブリ (V9.0 で追加)
TypeGenericInfoMap	123	アセンブリ (V9.0 で追加)

ReadyToRunSectionType.CompilerIdentifier

このセクションには、イメージの生成に使用されたコンパイラを識別するゼロ終端 ASCII 文字列が含まれます。

例: CoreCLR 4.6.22727.0 PROJECTK

ReadyToRunSectionType.ImportSections

このセクションには READYTORUN_IMPORT_SECTION 構造体の配列が含まれます。各エントリは、モジュール外部からの値で埋める必要があったスロットの範囲を記述します (通常は遅延処理)。各範囲のスロットの初期値は、ゼロまたは遅延初期化ヘルパーへのポインタです。

```
struct READYTORUN_IMPORT_SECTION
{
    IMAGE_DATA_DIRECTORY    Section;           // Section containing values to be fixed up
    USHORT                  Flags;              // One or more of ReadyToRunImportSectionFlags
    BYTE                   Type;               // One of ReadyToRunImportSectionType
    BYTE                   EntrySize;
    DWORD                  Signatures;         // RVA of optional signature descriptors
    DWORD                  AuxiliaryData;      // RVA of optional auxiliary data (typically GC info)
};

C++
```

READYTORUN_IMPORT_SECTIONS::Flags

ReadyToRunImportSectionFlags	値	説明
ReadyToRunImportSectionFlags::None	0x0000	なし
ReadyToRunImportSectionFlags::Eager	0x0001	セクション内のスロットがイメージロード時に初期化される必要がある場合に設定されます。遅延初期化ができない場合、または望ましくない信頼性やパフォーマンスへの影響（予期しない障害や GC トリガーポイント、遅延初期化のオーバーヘッド）がある場合に、遅延初期化を回避するために使用されます。
ReadyToRunImportSectionFlags::PCode	0x0004	セクションにコードへのポインタが含まれます

READYTORUN_IMPORT_SECTIONS::Type

ReadyToRunImportSectionType	値	説明
ReadyToRunImportSectionType::Unknown	0	このセクション内のスロットのタイプは未指定です。
ReadyToRunImportSectionType::StubDispatch	2	このセクション内のスロットはディスパッチにスタブを利用します。
ReadyToRunImportSectionType::StringHandle	3	このセクション内のスロットは文字列を保持します。
ReadyToRunImportSectionType::ILBodyFixups	7	このセクション内のスロットはクロスモジュール IL ボディを表します。

将来: セクションタイプは、同じタイプのスロットをグループ化するために使用できます。たとえば、すべての仮想スタブディスパッチ (virtual stub dispatch) スロットをグループ化して、仮想スタブディスパッチセルを初期状態にリセットすることを簡素化できます。

READYTORUN_IMPORT_SECTIONS::Signatures

このフィールドは、スロットの配列と並列な RVA の配列を指します。各 RVA は、対応するスロットを埋めるために必要な情報を含むフィックスアップシグネチャ (fixup signature) を指します。シグネチャのエンコーディングは、ECMA-335 のシグネチャに使用されるエンコーディングに基づいています。シグネチャの最初の要素はフィックスアップの種類を記述し、シグネチャの残りはフィックスアップの種類に基づいて異なります。

💡 初心者向け補足

フィックスアップ (fixup) とは、コンパイル時に確定できないアドレスや参照を、実行時（ロード時）に解決して埋め込む処理のことです。たとえば、あるメソッドの実際のメモリアドレスはロード時まで分からなかったため、R2R イメージにはフィックスアップ情報が埋め込まれ、ランタイムがそれを解決します。これは、C/C++ のリンクにおけるリロケーション (relocation) に似た概念です。

ReadyToRunFixupKind	値	説明
READYTORUN_FIXUP_ThisObjDictionaryLookup	0x07	<code>this</code> を使用したジェネリックルックアップ。型シグネチャとメソッドシグネチャが続きます。
READYTORUN_FIXUP_TypeDictionaryLookup	0x08	インスタンス化された型のメソッドに対する型ベースのジェネリックルックアップ。 typespec シグネチャが続きます。
READYTORUN_FIXUP_MethodDictionaryLookup	0x09	ジェネリックメソッドルックアップ。 method spec シグネチャが続きます。
READYTORUN_FIXUP_TypeHandle	0x10	ランタイムに対して型を一意に識別するポインタ。 typespec シグネチャが続きます (ECMA-335 参照)。
READYTORUN_FIXUP_MethodHandle	0x11	ランタイムに対してメソッドを一意に識別するポインタ。 メソッドシグネチャが続きます (以下参照)。
READYTORUN_FIXUP_FieldHandle	0x12	ランタイムに対してフィールドを一意に識別するポインタ。 フィールドシグネチャが続きます (以下参照)。
READYTORUN_FIXUP_MethodEntry	0x13	メソッドエントリポイントまたは呼び出し。 メソッドシグネチャが続きます。
READYTORUN_FIXUP_MethodEntry_DefToken	0x14	メソッドエントリポイントまたは呼び出し。 methoddef トークンが続きます (ショートカット)。
READYTORUN_FIXUP_MethodEntry_RefToken	0x15	メソッドエントリポイントまたは呼び出し。 methodref トークンが続きます (ショートカット)。
READYTORUN_FIXUP_VirtualEntry	0x16	仮想メソッドエントリポイントまたは呼び出し。 メソッドシグネチャが続きます。
READYTORUN_FIXUP_VirtualEntry_DefToken	0x17	仮想メソッドエントリポイントまたは呼び出し。 methoddef トークンが続きます (ショートカット)。
READYTORUN_FIXUP_VirtualEntry_RefToken	0x18	仮想メソッドエントリポイントまたは呼び出し。 methodref トークンが続きます (ショートカット)。
READYTORUN_FIXUP_VirtualEntry_Slot	0x19	仮想メソッドエントリポイントまたは呼び出し。 typespec シグネチャとスロットが続きます。
READYTORUN_FIXUP_Helper	0x1A	ヘルパー呼び出し。 ヘルパー呼び出し ID が続きます (第4章「ヘルパー呼び出し」参照)。

ReadyToRunFixupKind	値	説明
READYTORUN_FIXUP_StringHandle	0x1B	文字列ハンドル。メタデータ文字列トークンが続きます。
READYTORUN_FIXUP_NewObject	0x1C	新規オブジェクトヘルパー。typespec シグネチャが続きます。
READYTORUN_FIXUP_NewArray	0x1D	新規配列ヘルパー。typespec シグネチャが続きます。
READYTORUN_FIXUP_IsInstanceOf	0x1E	isinst ヘルパー。typespec シグネチャが続きます。
READYTORUN_FIXUP_ChkCast	0x1F	chkcast ヘルパー。typespec シグネチャが続きます。
READYTORUN_FIXUP_FieldAddress	0x20	フィールドアドレス。フィールドシグネチャが続きます。
READYTORUN_FIXUP_CctorTrigger	0x21	静的コンストラクタトリガー。typespec シグネチャが続きます。
READYTORUN_FIXUP_StaticBaseNonGC	0x22	非 GC 静的ベース。typespec シグネチャが続きます。
READYTORUN_FIXUP_StaticBaseGC	0x23	GC 静的ベース。typespec シグネチャが続きます。
READYTORUN_FIXUP_ThreadStaticBaseNonGC	0x24	非 GC スレッドローカル静的ベース。typespec シグネチャが続きます。
READYTORUN_FIXUP_ThreadStaticBaseGC	0x25	GC スレッドローカル静的ベース。typespec シグネチャが続きます。
READYTORUN_FIXUP_FieldBaseOffset	0x26	指定された型のフィールドの開始オフセット。typespec シグネチャが続きます。基底クラスの脆弱性 (fragility) に対処するために使用されます。
READYTORUN_FIXUP_FieldOffset	0x27	フィールドオフセット。フィールドシグネチャが続きます。
READYTORUN_FIXUP_TypeDictionary	0x28	ジェネリックコード用の隠しディクショナリ引数。typespec シグネチャが続きます。
READYTORUN_FIXUP_MethodDictionary	0x29	ジェネリックコード用の隠しディクショナリ引数。メソッドシグネチャが続きます。
READYTORUN_FIXUP_Check_TypeLayout	0x2A	型レイアウトの検証。typespec と期待される型レイアウトディスクリプタ (descriptor) が続きます。

ReadyToRunFixupKind	値	説明
READYTORUN_FIXUP_Check_FieldOffset	0x2B	フィールドオフセットの検証。フィールドシグネチャと期待されるフィールドレイアウトディスクリプタが続きます。
READYTORUN_FIXUP_DelegateCtor	0x2C	デリゲートコンストラクタ。メソッドシグネチャが続きます。
READYTORUN_FIXUP_DeclaringTypeHandle	0x2D	メソッド宣言型のディクショナリルックアップ。型シグネチャが続きます。
READYTORUN_FIXUP_IndirectPInvokeTarget	0x2E	インラインされた PInvoke のターゲット（間接）。メソッドシグネチャが続きます。
READYTORUN_FIXUP_PInvokeTarget	0x2F	インラインされた PInvoke のターゲット。メソッドシグネチャが続きます。
READYTORUN_FIXUP_Check_InstructionSetSupport	0x30	フィックスアップに関連付けられた R2R コードを使用するためにサポートされている必要がある/サポートされていない必要がある命令セットを指定します。
READYTORUN_FIXUP_Verify_FieldOffset	0x31	コンパイル時と実行時のフィールドオフセットが一致することを確認するランタイムチェックを生成します。CheckFieldOffset とは異なり、失敗時にメソッドを静かにドロップするのではなく、ランタイム例外を生成します。
READYTORUN_FIXUP_Verify_TypeLayout	0x32	コンパイル時と実行時のフィールドオフセットが一致することを確認するランタイムチェックを生成します。CheckFieldOffset とは異なり、失敗時にメソッドを静かにドロップするのではなく、ランタイム例外を生成します。
READYTORUN_FIXUP_Check_VirtualFunctionOverride	0x33	仮想関数解決がコンパイル時と実行時で同等の動作をすることを確認するランタイムチェックを生成します。同等でない場合、コードは使用されません。使用されるシグネチャの詳細については 仮想オーバーライドシグネチャ を参照してください。
READYTORUN_FIXUP_Verify_VirtualFunctionOverride	0x34	仮想関数解決がコンパイル時と実行時で同等の動作をすることを確認するランタイムチェックを生成します。同等でない場合、ランタイム障害を生成します。使用されるシグネチャの詳細については 仮想オーバーライドシグネチャ を参照してください。
READYTORUN_FIXUP_Check_IL_Body	0x35	IL メソッドが実行時にコンパイル時と同じように定義されているかを確認します。一致しない場合、コードは使用されません。詳細については IL Body シグネチャ を参照してください。

ReadyToRunFixupKind	値	説明
READYTORUN_FIXUP_Verify_IL_Body	0x36	IL ボディがコンパイル時と実行時で同じように定義されていることを検証します。一致しない場合、ハードなランタイム障害を引き起こします。詳細については IL Body シグネチャ を参照してください。
READYTORUN_FIXUP_ModuleOverride	0x80	フィックスアップ ID と OR されると、シグネチャ内のフィックスアップバイトの後に、シグネチャのマスターコンテキストモジュールの MSIL メタデータ内、またはマニフェストメタデータ R2R ヘッダーテーブル内の assemblyref インデックスを持つエンコードされた uint が続きます（INLINE 化によって入力 MSIL に見られないアセンブリへの参照が持ち込まれる場合に使用されます）。

メソッドシグネチャ

ECMA-335 で定義されている MethodSpec シグネチャは、ネイティブコードが参照するメソッドフレーバー (flavor) を記述するのに十分な豊富さはありません。メソッドシグネチャの最初の要素はフラグです。その後にメソッドトークンと、フラグによって決定される追加データが続きます。

ReadyToRunMethodSigFlags	値	説明
READYTORUN_METHOD_SIG_UnboxingStub	0x01	メソッドのアンボクシング (unboxing) エントリポイント。
READYTORUN_METHOD_SIG_InstantiatingStub	0x02	メソッドのインスタンス化エントリポイントで、隠しディクショナリジェネリック引数を取りません。
READYTORUN_METHOD_SIG_MethodInstantiation	0x04	メソッドインスタンス化。インスタンス化引数の数と、それぞれの typespec が追加データとして付加されます。
READYTORUN_METHOD_SIG_SlotInsteadOfToken	0x08	設定されている場合、トークンはスロット番号です。メタデータトークンを持たない多次元配列メソッド、および安定したインターフェースメソッドの最適化として使用されます。 MemberRefToken と組み合わせることはできません。
READYTORUN_METHOD_SIG_MemberRefToken	0x10	設定されている場合、トークンは memberref トークンです。設定されていない場合、トークンは methoddef トークンです。
READYTORUN_METHOD_SIG_Constrained	0x20	メソッド解決のための制約型。typespec が追加データとして付加されます。
READYTORUN_METHOD_SIG_OwnerType	0x40	メソッド型。typespec が追加データとして付加されます。

ReadyToRunMethodSigFlags	値	説明
READYTORUN_METHOD_SIG_UpdateContext	0x80	設定されている場合、トークン処理を実行する前にトークンの解析に使用されるモジュールを更新します。フラグの直後にモジュールテーブルへの uint インデックスが続きます。

フィールドシグネチャ

ECMA-335 は、ネイティブコードが参照するメソッドフレーバーを記述するのに十分な豊富さを持つフィールドシグネチャを定義していません。フィールドシグネチャの最初の要素はフラグです。その後にフィールドトークンと、フラグによって決定される追加データが続きます。

ReadyToRunFieldSigFlags	値	説明
READYTORUN_FIELD_SIG_IndexInsteadOfToken	0x08	安定したフィールドの最適化として使用されます。 MemberRefToken と組み合わせることはできません。
READYTORUN_FIELD_SIG_MemberRefToken	0x10	設定されている場合、トークンは memberref トークンです。設定されていない場合、トークンは fielddef トークンです。
READYTORUN_FIELD_SIG_OwnerType	0x40	フィールド型。typespec が追加データとして付加されます。

仮想オーバーライドシグネチャ

ECMA 335 には、オーバーライドされたメソッドを記述するための自然なエンコーディングがありません。これらのシグネチャは、ReadyToRunVirtualFunctionOverrideFlags バイトとしてエンコードされ、その後に宣言メソッドを表すメソッドシグネチャ、脱仮想化 (devirtualize) される型を表す型シグネチャ、および（オプションとして）実装メソッドを示すメソッドシグネチャが続きます。

ReadyToRunVirtualFunctionOverrideFlags	値	説明
READYTORUN_VIRTUAL_OVERRIDE_None	0x00	フラグは設定されていません。
READYTORUN_VIRTUAL_OVERRIDE_VirtualFunctionOverridden	0x01	設定されている場合、仮想関数には実装があり、オプションのメソッド実装シグネチャにエンコードされています。

IL Body シグネチャ

ECMA 335 は、メソッドの正確な実装をそれ自体で表現できるフォーマットを定義していません。このシグネチャは、メソッドのすべての IL、EH テーブル、ローカル変数テーブルを保持し、それらのテーブル内の各トークン（型参照を除く）は、ローカルシグネチャストリームへのインデックスに置き換えられます。これらのシグネチャは、MemberRef、TypeSpec、MethodSpec、StandaloneSignature、および文字列を記述するために必要なメタデータのそのままのコピーです。これらすべてが大きなバイト配列にバンドルされます。さらに、型参照を解決するための一連の TypeSignature と、インスタンス化されていないメソッドへのメソッド参照が続きます。これらすべてが実行時に存在するデータと一致する場合、フィックスアップは満たされたと見なされます。フォーマットの正確な詳細については ReadyToRunStandaloneMetadata.cs を参照してください。

READYTORUN_IMPORT_SECTIONS::AuxiliaryData

`READYTORUN_HELPER_DelayLoad_MethodCall` ヘルパーを通じて遅延解決されるスロットの場合、補助データ (auxiliary data) は、ヘルパーの実行中に正確な GC スタックスキャンを可能にする圧縮された引数マップです。CoreCLR ランタイムクラス `GCRefMapDecoder` がこの情報の解析に使用されます。このデータは、保守的なスタックスキャンを許可するランタイムでは必要ありません。

補助データテーブルには、インポートセクション内のメソッドエントリと正確に同じ数の GC 参照マップレコードが含まれます。GC 参照マップの検索を高速化するために、補助データセクションはランタイム関数テーブル内の 1024 番目ごとのメソッドの線形化された GC 参照マップ内のオフセットを保持するルックアップテーブルで始まります。

補助データ内のオフセット	サイズ	内容
0	4	メソッド #0 の GC 参照マップ情報へのこのバイトからの相対オフセット (すなわち $4 * (\text{MethodCount} / 1024 + 1)$)
4	4	メソッド #1024 の GC 参照マップ情報へのオフセット
8	4	メソッド #2048 の GC 参照マップ情報へのオフセット
...		
$4 * (\text{MethodCount} / 1024 + 1)$...	シリализされた GC 参照マップ情報

GC 参照マップは、呼び出しサイトの引数の GC 型をエンコードするために使用されます。論理的には、`<pos, token>` のシーケンスであり、`pos` はスタックフレーム内の参照の位置、`token` は GC 参照の型 (`GCREFMAP_XXX` 値のいずれか) です:

CORCOMPILE_GCREFMAP_TOKENS	値	スタックフレームエントリの解釈
GCREFMAP_SKIP	0	GC に関連しないエントリ
GCREFMAP_REF	1	GC 参照
GCREFMAP_INTERIOR	2	GC 参照へのポインタ
GCREFMAP_METHOD_PARAM	3	ジェネリックメソッドへの隠しメソッドインスタンス化引数
GCREFMAP_TYPE_PARAM	4	ジェネリックメソッドへの隠し型インスタンス化引数
GCREFMAP_VASIG_COOKIE	5	VARARG シグチャクッキー

位置の値は、`size_t` (`IntPtr`) 単位 (32 ビットアーキテクチャでは 4 バイト、64 ビットアーキテクチャでは 8 バイト) で、GC 参照を含む可能性のあるトランジションフレームの最初の位置から計算されます。`arm64` を除くすべてのアーキテクチャでは、これはスピンされた引数レジスタの配列の先頭です。`arm64` では、呼び出されたメソッドによって戻り値を格納する場所を渡すために使用される `x8` レジスタのオフセットです。

- エンコーディングは常にバイト境界から開始します。各バイトの最上位ビットは、エンコーディングストリームの終了を示すために使用されます。最後のバイトの最上位ビットはゼロです。つまり、各バイトには 7 つの有効ビットがあります。

- "pos" は常に前の pos からのデルタとしてエンコードされます。
 - 基本エンコーディング単位は 2 ビットです。値 0、1、2 は一般的な構成（單一スロットのスキップ、GC 参照、内部ポインタ）です。値 3 は拡張エンコーディングが続くことを意味します。
 - 拡張情報は、1 つ以上の 4 ビットブロックで整数エンコードされます。4 ビットブロックの最上位ビットは終了を示すために使用されます。
 - x86 の場合、エンコーディングは呼び出し先がポップするスタックのサイズから始まります。サイズは上記と同じメカニズム（2 ビットの基本エンコーディングと、大きな値のための拡張エンコーディング）を使用してエンコードされます。
-

ReadyToRunSectionType.RuntimeFunctions

このセクションには、イメージ内のすべてのコードブロックをアンワインド情報 (unwind info) へのポインタとともに記述する **RUNTIME_FUNCTION** エントリのソート済み配列が含まれます。名前にもかかわらず、これらのコードブロックはメソッドボディを表す場合もあれば、独自のアンワインドデータを必要とするその一部（例：ファンクレット (funclet)）のみを表す場合もあります。標準の Windows xdata/pdata フォーマットが使用されます。x86 アンワインド情報の標準がないことを補うために、x86 では ARM フォーマットが使用されます。アンワインド情報blob の直後に GC 情報blob が続きます。amd64 ではエンコーディングがわずかに異なり、アンワインド情報blob の終了 RVA を表す追加の 4 バイトがエンコードされます。

RUNTIME_FUNCTION (x86, arm, arm64, サイズ = 8 バイト)

オフセット	サイズ	値
0	4	アンワインド情報開始 RVA
4	4	GC 情報開始 RVA

RUNTIME_FUNCTION (amd64, サイズ = 12 バイト)

オフセット	サイズ	値
0	4	アンワインド情報開始 RVA
4	4	アンワインド情報終了 RVA (最後のバイトの RVA + 1)
8	4	GC 情報開始 RVA

ReadyToRunSectionType.MethodDefEntryPoints

このセクションには、methoddef 行をメソッドエントリポイントにマッピングするネイティブフォーマットスパース配列 (sparse array)（第4章「ネイティブフォーマット」参照）が含まれます。methoddef が配列のインデックスとして使用されます。配列の要素は

`RuntimeFunctions` 内のメソッドのインデックスであり、メソッドが実行を開始する前に埋める必要があるスロットのリストが続きます。

メソッドのインデックスは左に 1 ビットシフトされ、下位ビットがフィックスアップすべきスロットのリストが続くかどうかを示します。スロットのリストは以下のようにエンコードされます (NGen で使用されるのと同じエンコーディング) :

```
READYTORUN_IMPORT_SECTIONS absolute index
    absolute slot index
    slot index delta
    -
    slot index delta
    0
READYTORUN_IMPORT_SECTIONS index delta
    absolute slot index
    slot index delta
    -
    slot delta
    0
READYTORUN_IMPORT_SECTIONS index delta
    absolute slot index
    slot index delta
    -
    slot delta
    0
    0
```

フィックスアップリストは、ニブル (nibble) (1 ニブル = 4 ビット) としてエンコードされた整数のストリームです。ニブルの 3 ビットは値の 3 ビットを格納するために使用され、最上位ビットは次のニブルに値の残りが含まれるかどうかを示します。ニブルの最上位ビットが設定されている場合、値は次のニブルに続きます。

セクションインデックスとスロットインデックスは、初期の絶対インデックスからのデルタエンコーディングされたオフセットです。デルタエンコーディングとは、*i* 番目の値が値 [1..i] の合計であることを意味します。

リストは 0 で終端されます (0 は有効なデルタとしては意味がありません)。

注: これはアセンブリごとのセクションです。單一ファイル R2R ファイルでは、メインの R2R ヘッダーから直接指されます。コンポジット R2R ファイルでは、各コンポーネントモジュールが `READYTORUN_SECTION_ASSEMBLIES_ENTRY` コアヘッダー構造体によって指される独自のエントリポイントセクションを持ちます。

ReadyToRunSectionType.ExceptionInfo

例外処理情報。このセクションには、`MethodStart` RVA でソートされた `READYTORUN_EXCEPTION_LOOKUP_TABLE_ENTRY` の配列が含まれます。`ExceptionInfo` は、指定されたメソッドの例外処理情報を記述する `READYTORUN_EXCEPTION_CLAUSE` 配列の RVA です。

```
struct READYTORUN_EXCEPTION_LOOKUP_TABLE_ENTRY
{
    DWORD MethodStart;
    DWORD ExceptionInfo;
```

C++

```

};

struct READYTORUN_EXCEPTION_CLAUSE
{
    CorExceptionFlag    Flags;
    DWORD               TryStartPC;
    DWORD               TryEndPC;
    DWORD               HandlerStartPC;
    DWORD               HandlerEndPC;

    union {
        mdToken          ClassToken;
        DWORD             FilterOffset;
    };
};

```

NGen と同じエンコーディングが使用されます。

ReadyToRunSectionType.DebugInfo

このセクションには、デバッグをサポートするための情報（ネイティブオフセットとローカル変数のマップ）が含まれます。

TODO: デバッグ情報のエンコーディングをドキュメント化する。 NGen で使用されるのと同じエンコーディングです。 デバッガが別途保存されたデバッグ情報を処理できるようになった場合には不要です。

ReadyToRunSectionType.DelayLoadMethodCallThunks

このセクションは、 `READYTORUN_HELPER_DelayLoad_MethodCall` ヘルパーのサンク (thunk) を含む領域をマークします。これは、遅延解決される呼び出しへのステップインのためにデバッガが使用します。 デバッガが別途保存されたデバッグ情報を処理できるようになった場合には不要です。

ReadyToRunSectionType.AvailableTypes

このセクションには、コンパイルモジュール内のすべての定義型およびエクスポート型のネイティブハッシュテーブル (hashtable) が含まれます。 キーは完全な型名で、値はエクスポート型または定義型のトークン行 ID を左に 1 ビットシフトし、ビット 0 でトークンタイプを定義する値と OR したものです:

ビット値	トークンタイプ
0	定義型
1	エクスポート型

型名のハッシュに使用されるバージョン耐性ハッシュアルゴリズム (version-resilient hashing algorithm) は、[vm/versionresilienthashcode.cpp](#) に実装されています。

注: これはアセンブリごとのセクションです。単一ファイル R2R ファイルでは、メインの R2R ヘッダーから直接指されます。コンポジット R2R ファイルでは、各コンポーネントモジュールが `READYTORUN_SECTION_ASSEMBLIES_ENTRY` コアヘッダー構造体によって指される独自の利用可能型セクションを持ちます。

ReadyToRunSectionType.InstanceMethodEntryPoints

このセクションには、R2R 実行可能ファイルにコンパイルされたすべてのジェネリックメソッドインスタンス化のネイティブハッシュテーブルが含まれます。キーはメソッドインスタンスシグネチャです。適切なバージョン耐性ハッシュコードの計算は [vm/versionresilienthashcode.cpp](#) に実装されています。値は `EntryPointWithBlobVertex` クラスによって表され、ランタイム関数テーブル内のメソッドインデックス、フィックスアッププロップ、およびメソッドシグネチャをエンコードするプロップを格納します。

注: 非ジェネリックメソッドのエントリポイントとは対照的に、このセクションはコンポジット R2R イメージの場合にイメージ全体にわたります。コンポジット実行可能ファイル内のすべてのアセンブリが必要とするすべてのジェネリクスを表します。本ドキュメントの他の箇所で述べたように、CoreCLR ランタイムは、コンポジット R2R ケースでこのセクションに格納されたメソッドを適切に検索するための変更が必要です。

注: ジェネリックメソッドおよびジェネリック型の非ジェネリックメソッドはこのテーブルにエンコードされ、ランタイムは潜在的に複数のモジュールでこのテーブルを検索することが期待されます。まず、ランタイムはメソッドを定義するモジュールのこのテーブルを検索し、次に「代替」ジェネリクスの場所を使用することが期待されます。この代替の場所は、メソッドのジェネリック引数の一つの定義モジュールである、定義モジュールではないモジュールとして定義されます。この代替ルックアップは現在、深くネストされたアルゴリズムではありません。そのルックアップが失敗した場合、`READYTORUN_FLAG_UNRELATED_R2R_CODE` をフラグとして指定したすべてのモジュールに対してルックアップが行われます。

ReadyToRunSectionType.InliningInfo (v2.1+)

TODO: インライン情報のエンコーディングをドキュメント化する

ReadyToRunSectionType.ProfileDataInfo (v2.2+)

TODO: プロファイルデータのエンコーディングをドキュメント化する

ReadyToRunSectionType.ManifestMetadata (v2.3+、v6.3+ で変更あり)

マニフェストメタデータ (manifest metadata) は、入力 MSIL に格納されたアセンブリ参照に加えて、インライン化によってバージョンバブル (version bubble) 内に導入された追加の参照アセンブリを含む [ECMA-335] メタデータプロップです。R2R バージョン 3.1 時点では、メタデータは AssemblyRef テーブルにのみ使用されます。これは、シグネチャ内のモジュールオーバーライドインデックスを実際の

参照モジュールに変換するために使用されます（シグネチャフィックスアップバイトの `READYTORUN_FIXUP_ModuleOverride` ビットフラグまたは `ELEMENT_TYPE_MODULE_ZAPSIG` COR 要素型のいずれかを使用）。

💡 初心者向け補足

バージョンバブル (version bubble) とは、一緒にコンパイルされ、互いのコード/データ構造を直接参照できるアセンブリのグループのことです。バブル内のアセンブリは互いに「信頼」し合い、内部の詳細に依存できますが、バブル外のアセンブリに対しては安定した公開 API のみを使用する必要があります。

注: バージョンバブル外部のアセンブリへの参照を `READYTORUN_FIXUP_ModuleOverride` または `ELEMENT_TYPE_MODULE_ZAPSIG` の概念を通じてマニフェストメタデータで使用することは意味がありません。メタデータトークン値が一定であるという保証がないため、それらに対してシグネチャを相対的にエンコードすることはできません。ただし、R2R バージョン 6.3 以降、ネイティブマニフェストメタデータには、実際の実装アセンブリにさらに解決されるトークンが含まれる場合があります。

モジュールオーバーライドインデックスの変換アルゴリズムは以下の通りです (`ILAR` = 入力 MSIL の `AssemblyRef` 行数) :

R2R バージョン 6.2 以下の場合

モジュールオーバーライドインデックス (i)	参照アセンブリ
$i = 0$	グローバルコンテキスト - シグネチャを含むアセンブリ
$1 \leq i \leq \text{ILAR}$	i は MSIL <code>AssemblyRef</code> テーブルへのインデックス
$i > \text{ILAR}$	$i - \text{ILAR} - 1$ はマニフェストメタデータの <code>AssemblyRef</code> テーブルへのゼロベースインデックス

注: これは、 $i = \text{ILAR} + 1$ に対応するエントリが実際には未定義であることを意味します。マニフェストメタデータ `AssemblyRef` テーブルの `NULL` エントリ (ROWID #0) に対応するためです。マニフェストメタデータへの最初の意味のあるインデックスは $i = \text{ILAR} + 2$ で、ROWID #1 に対応し、歴史的に Crossgen によって入力アセンブリ情報で埋められていますが、これに依存すべきではありません。実際、入力アセンブリはマニフェストメタデータでは不要であり、特別なインデックス 0 を使用してモジュールオーバーライドをエンコードできます。

R2R バージョン 6.3 以上の場合

モジュールオーバーライドインデックス (i)	参照アセンブリ
$i = 0$	グローバルコンテキスト - シグネチャを含むアセンブリ
$1 \leq i \leq \text{ILAR}$	i は MSIL <code>AssemblyRef</code> テーブルへのインデックス
$i = \text{ILAR} + 1$	i はマニフェストメタデータ自体を参照するインデックス
$i > \text{ILAR} + 1$	$i - \text{ILAR} - 2$ はマニフェストメタデータの <code>AssemblyRef</code> テーブルへのゼロベースインデックス

さらに、`System.Private.CoreLib` を参照するモジュール内の `ModuleRef` は、マニフェストメタデータ内の `TypeRef` の `ResolutionContext` として機能できます。これは常に `System.Object` 型を含むモジュールを参照します。

ReadyToRunSectionType.AttributePresence (v3.1+)

TODO: 属性プレゼンスのエンコーディングをドキュメント化する

注: これはアセンブリごとのセクションです。単一ファイル R2R ファイルでは、メインの R2R ヘッダーから直接指されます。コンポジット R2R ファイルでは、各コンポーネントモジュールが `READYTORUN_SECTION_ASSEMBLIES_ENTRY` コアヘッダー構造体によって指される独自の属性プレゼンスセクションを持ちます。

ReadyToRunSectionType.InliningInfo2 (v4.1+)

インライン化情報セクションは、どのメソッドが他のメソッドにインライン化されたかを記録します。単一の ネイティブフォーマットハッシュテーブル（後述）で構成されます。

ハッシュテーブル内のエントリは、各インライナー (inlinee) に対するインライナー (inliner) のリストです。ハッシュテーブル内の 1 エントリは 1 つのインライナーに対応します。ハッシュテーブルは、モジュール名のハッシュコードとインライナーの RID を XOR した値でハッシュされます。

ハッシュテーブルのエントリは、圧縮された符号なし整数のカウント付きシーケンスです:

- インライナーの RID を左に 1 ビットシフトしたもの。最下位ビットが設定されている場合、これは外部モジュールからのインライナーです。その場合、モジュールオーバーライドインデックス（上記で定義）が別の圧縮された符号なし整数として続けます。
- インライナーの RID が続きます。インライナーのエンコード方法と同様にエンコードされます（左シフトし、最下位ビットが外部 RID を示します）。RID を直接エンコードする代わりに、RID デルタ（前の RID と現在の RID の差）がエンコードされます。これにより、より良い整数圧縮が可能になります。

外部 RID は、コンパイル時に脆弱なインライン化が許可された場合にのみ存在します。

TODO: `DelayLoadMethodCallThunks` や `InliningInfo` もコンポジット R2R ファイルフォーマットに固有の変更が必要かどうかはまだ検討中です。

ReadyToRunSectionType.ComponentAssemblies (v4.1+)

このイメージ全体のセクションは、コンポジット R2R ファイルのメイン R2R ヘッダーにのみ存在します。これは、マニフェストメタデータの `AssemblyRef` テーブルのインデックスと並列な `READYTORUN_SECTION_ASSEMBLIES_ENTRY` エントリの配列です。行インデックスが同等の `AssemblyRef` インデックスに対応する線形テーブルです。ECMA 335 の `AssemblyRef` テーブルと同様に、インデックスは 1 ベースです（テーブルの最初のエントリはインデックス 1 に対応します）。

```
struct READYTORUN_SECTION_ASSEMBLIES_ENTRY
{
    IMAGE_DATA_DIRECTORY CorHeader;           // Input MSIL metadata COR header (for composite R2R images with embedded)
    IMAGE_DATA_DIRECTORY ReadyToRunHeader; // READYTORUN_CORE_HEADER of the assembly in question
};
```

C++

ReadyToRunSectionType.OwnerCompositeExecutable (v4.1+)

スタンダードアロン MSIL を持つコンポジット R2R 実行可能ファイルの場合、MSIL ファイルはコンパイル中に書き換えられ、適切なシグチャとメジャー/マイナーバージョンペアを持つ正式な ReadyToRun ヘッダーが付与されます。Flags には

`READYTORUN_FLAG_COMPONENT` ビットが設定され、そのセクションリストには、この MSIL が属するコンポジット R2R 実行可能ファイルのファイル名を拡張子付き（パスなし）で UTF-8 文字列としてエンコードする `OwnerCompositeExecutable` セクションのみが含まれます。ランタイムは、MSIL をロードする際にコンパイル済みネイティブコードを含むコンポジット R2R 実行可能ファイルを特定するためにこの情報を使用します。

ReadyToRunSectionType.PgoInstrumentationData (v5.2+)

TODO: PGO インストルメンテーションデータをドキュメント化する

ReadyToRunSectionType.ManifestAssemblyMvids (v5.3+)

このセクションは、マニフェストメタデータ内の各アセンブリに対する 16 バイト MVID レコードのバイナリ配列です。マニフェストメタデータに格納されたアセンブリ数は、配列内の MVID レコード数と等しくなります。MVID レコードは実行時に、ロードされたアセンブリがバージョニングバブルを表すマニフェストメタデータによって参照されたものと一致することを検証するために使用されます。

ReadyToRunSectionType.CrossModuleInlineInfo (v6.3+)

インライン化情報セクションは、どのメソッドが他のメソッドにインライン化されたかを記録します。単一の ネイティブフォーマットハッシュテーブル（後述）で構成されます。

ハッシュテーブル内のエントリは、各インライナーに対するインライナーのリストです。ハッシュテーブル内の 1 エントリは 1 つのインライナーに対応します。ハッシュテーブルは、インスタンス化されていない methoddef インライナーのバージョン耐性ハッシュコードでハッシュされます。

ハッシュテーブルのエントリは、`InlineeIndex` で始まる圧縮された符号なし整数のカウント付きシーケンスです。`InlineeIndex` は 30 ビットのインデックスと 2 ビットのフラグを組み合わせたもので、インライナーのシーケンスの解析方法と、インライナーを見つけるためにインデックスされるテーブルを定義します。

- `InlineeIndex`
 - インライナーを定義するための最下位 2 ビットのフラグフィールドを持つインデックス
 - (`flags & 1`) == 0 の場合、インデックスは MethodDef RID であり、モジュールがコンポジットイメージの場合、メソッドのモジュールインデックスが続きます
 - (`flags & 1`) == 1 の場合、インデックスは ILBody インポートセクションへのインデックスです
 - (`flags & 2`) == 0 の場合、インライナーリストは:

- インライナー RID デルタ - 下記の定義を参照
 - (flags & 2) == 2 の場合、続くのは:
 - ILBody インポートセクションへのデルタエンコードされたインデックスのカウント
 - READYTORUN_IMPORT_SECTION_TYPE_ILBODYFIXUPS タイプを持つ最初のインポートセクションへのデルタエンコードされたインデックスのシーケンス
 - インライナー RID デルタ - 下記の定義を参照
- インライナー RID デルタ (READYTORUN_FLAG_MULTIMODULE_VERSION_BUBBLE フラグが設定されたモジュールで指定されるマルチモジュールバージョンバブルイメージの場合)
- 最下位ビットにフラグを持つインライナー RID デルタのシーケンス
 - フラグが設定されている場合、インライナー RID の後にモジュール ID が続きます
 - そうでない場合、モジュールはインライニーメソッドと同じモジュールです
- インライナー RID デルタ (シングルモジュールバージョンバブルイメージの場合)
- インライナー RID デルタのシーケンス

このセクションは InliningInfo2 セクションに加えて含まれる場合があります。

ReadyToRunSectionType.HotColdMap (v8.0+)

ReadyToRun 8.0+ では、メソッドをホットパート (hot part) とコールドパート (cold part) に分割して、それらが隣接しないようにするフォーマットがサポートされています。このホットコールドマップセクションは、メソッドがどのように分割されたかの情報を記録し、ランタイムがさまざまなサービスのためにそれらを特定できるようにします。

💡 初心者向け補足

ホット/コールド分割とは、頻繁に実行されるコード（ホット）とほとんど実行されないコード（コールド、例外処理パスなど）を物理的に分離する最適化手法です。ホットなコードをメモリ上で近くにまとめることで、CPU のキャッシュ効率を向上させ、パフォーマンスを改善します。

分割されたすべてのメソッドに対して、セクション内に 1 つのエントリがあります。各エントリには 2 つの符号なし 32 ビット整数があります。最初の整数はコールドパートのランタイム関数インデックスで、2 番目の整数はホットパートのランタイム関数インデックスです。

このテーブル内のメソッドは、ホットパートのランタイム関数インデックスでソートされています。コールドパートは常にホットパートと同じ順序で出力されるため、コールドパートのランタイム関数インデックスでもソートされています。あるいはランタイム関数テーブル 자체が RVA でソートされているため、RVA でソートされているとも言えます。

--hot-cold-splitting フラグがコンパイル時に指定されていない場合、またはコンパイラがメソッドを分割すべきではないと判断した場合、メソッドは分割されず、このセクションは存在しない場合があります。

ReadyToRunSectionType.MethodIsGenericMap (v9.0+)

このオプションセクションは、アセンブリ内の MethodDef がジェネリックパラメータを持つかどうかを示すビットベクトル (bit vector) を保持します。これにより、GenericParameter テーブルやメソッドのシグネチャを調べる代わりに、ビットベクトルへの問い合わせ（高速かつ効率的）によってメソッドがジェネリックかどうかを判定できます。

セクションは、ビットベクトル内のビット数を示す 32 ビット整数 1つで始まります。その整数の後に、すべてのデータの実際のビットベクトルが続きます。データは 8 ビットバイトにグループ化され、バイトの最下位ビットが最も低い MethodDef を表すビットです。

たとえば、ビットベクトルの最初のバイトは MethodDef 06000001 から 06000008 を表し、その最初のバイトの最下位ビットは MethodDef 06000001 の IsGeneric ビットを表すビットです。

ReadyToRunSectionType.EnclosingTypeMap (v9.0+)

このオプションセクションは、囲まれた型から囲む型への効率的な O(1) ルックアップを可能にします。ECMA 335 で定義された NestedClass テーブル（同じ情報をエンコードする）を使用する場合に必要なバイナリサーチが不要になります。このセクションは、アセンブリ内に 0xFFFF 未満の型が定義されている場合にのみ含めることができます。

このセクションの構造は: マップ内のエントリ数を示す 16 ビット符号なし整数 1つ。このカウントの後に、アセンブリ内で定義された各 TypeDef に対する 16 ビット符号なし整数が続きます。この typedef は囲む型の RID であり、typedef が他の型に囲まれていない場合は 0 です。

ReadyToRunSectionType.TypeGenericInfoMap (v9.0+)

このオプションセクションは、型に関するジェネリックの詳細の凝縮されたビューを表します。これにより、型のロードをより効率的にできます。

このセクションの構造は: マップ内のエントリ数を表す 32 ビット整数 1つの後に、型ごとに 1 つの 4 ビットエントリのシリーズが続きます。これらの 4 ビットエントリはバイトにグループ化され、各バイトは 2 エントリを保持し、バイトの最上位 4 ビットのエントリがより低い TypeDef RID を表すエントリです。

TypeGenericInfoMap エントリは、3 つの異なる情報セットを表す 4 ビットを持ちます。

1. ジェネリックパラメータの数はいくつか (0, 1, 2, MoreThanTwo) (TypeGenericInfoMap エントリの最下位 2 ビットで表されます)
2. ジェネリックパラメータに制約はあるか? (エントリの 3 番目のビット)
3. ジェネリックパラメータのいずれかに共変性 (covariance) または反変性 (contravariance) があるか? (エントリの 4 番目のビット)

ネイティブフォーマット

ネイティブフォーマット (Native Format) は、型システムデータを、実行時アクセスに効率的なバイナリフォーマットで永続化するためのエンコーディングパターンのセットです。ワーキングセットと CPU サイクルの両方で効率的です。（元々は .NET Native 向けに設計され、広く使用されています。）

 初心者向け補足

ネイティブフォーマットは、メタデータやルックアップテーブルなどのデータを、ランタイムが高速にアクセスできるよう特別にエンコードしたバイナリ形式です。整数の可変長エンコーディング、スペース配列、ハッシュテーブルなど、コンパクトさとアクセス速度の両方を追求したデータ構造が使われています。

整数エンコーディング

ネイティブフォーマットは、符号付きおよび符号なし数値に可変長エンコーディング方式を使用します。エンコーディングの最初のバイトの下位ビットが、後続バイト数を以下のように指定します:

- `xxxxxx00` (すなわち最下位ビットが 0) : 後続バイトなし。バイトを右に 1 ビットシフトし、符号付きおよび符号なし数値に対してそれぞれ符号拡張またはゼロ拡張します。
- `xxxxxx01` : 後続 1 バイト。読み取った 2 バイトからリトルエンディアン (little-endian) 順で 16 ビット数を構築し、右に 2 ビットシフトした後、符号拡張またはゼロ拡張します。
- `xxxxx011` : 後続 2 バイト。読み取った 3 バイトからリトルエンディアン順で 24 ビット数を構築し、右に 3 ビットシフトした後、符号拡張またはゼロ拡張します。
- `xxxx0111` : 後続 3 バイト。読み取った 4 バイトから 32 ビット数を構築した後、符号拡張またはゼロ拡張します。
- `xxxx1111` : 後続 4 バイト。最初のバイトを破棄し、続く 4 バイトから符号付きまたは符号なし数値を構築します (同様にリトルエンディアン順)。

例:

- 符号なし数値 12 (`0x0000000c`) は、单一バイト `0x18` として表現されます。
- 符号なし数値 1000 (`0x0000003e8`) は、2 バイト `0xa1, 0x0f` として表現されます。

スペース配列

NativeArray は、ヌル要素圧縮 (空ブロックの共有ストレージ) と可変サイズオフセットエンコーディング (データサイズに適応) を通じてコンパクトなストレージを維持しつつ、O(1) のインデックスアクセスを提供します。

配列は、ヘッダー、ブロックインデックス、ブロックの 3 つの部分で構成されます。

ヘッダーは可変エンコードされた値で:

- ビット 0-1: エントリインデックスサイズ
 - 0 = uint8 オフセット
 - 1 = uint16 オフセット
 - 2 = uint32 オフセット
- ビット 2-31: 配列内の要素数

ブロックインデックスはメモリ上でヘッダーの直後に続き、ブロックごとに 1 つのオフセットエントリ (ヘッダーにエンコードされた動的サイズ) で構成されます。各エントリは、ブロックインデックスセクションの先頭からの相対位置でデータブロックの位置を指します。配

例は最大ブロックサイズ 16 要素を使用し、ブロックインデックスは実質的に 16 個ずつの連続した配列インデックスの各グループを対応するデータブロックにマッピングします。

ブロックインデックスの後に実際のデータブロックが続きます。これらはツリーノード (tree node) とデータノード (data node) の 2 種類のノードで構成されます。

ツリーノードは可変長エンコードされた uint で構成されます:

- ビット 0: 設定されている場合、ノードにはより低いインデックスの子があります
- ビット 1: 設定されている場合、ノードにはより高いインデックスの子があります
- ビット 2-31: より高いインデックスの子のシフトされた相対オフセット

データノードはユーザー定義データを含みます。

各ブロックは最大 16 要素を持つため、深さは 4 です。

ルックアップアルゴリズムの手順

ステップ 1: ヘッダーの読み取り

- 配列から可変長エンコードされたヘッダー値をデコードします
- ビット 0-1 からエントリインデックスサイズを抽出します (0=uint8, 1=uint16, 2=uint32 オフセット)
- ヘッダー値を右に 2 ビットシフトして、ビット 2-31 から要素の総数を抽出します
- この情報を使用して、ブロックインデックスエントリの解釈方法を決定し、配列境界を検証します

ステップ 2: ブロックオフセットの計算

- ターゲット要素を含むブロックインデックス `blockIndex` を、インデックスをブロックサイズ (16) で割ることによって決定します。
- ブロックオフセットを含むメモリ位置 `pBlockOffset = baseOffset + entrySize * blockIndex` を計算します。ここで `baseOffset` はヘッダー直後のアドレスであり、`entrySize` はヘッダーの下位ビットによって決定されます。
- 計算された `pBlockOffset` とヘッダーによって決定されたエントリサイズを使用して、ブロックインデックステーブルからブロックオフセット `blockOffset` を読み取ります。
- 相対 `blockOffset` を絶対位置に変換するために `baseOffset` を加算します。

ステップ 3: ツリーナビゲーションの初期化

- 上記で計算された `blockOffset` を使用して、ブロックのバイナリツリー構造のルートから探索を開始します

ステップ 4: バイナリツリーのナビゲーションツリーの各レベルに対して (ビット位置 8, 4, 2, 1 を反復) :

ステップ 4a: ノードディスクリプタの読み取り

- 現在のノードの制御値をデコードします。ナビゲーションフラグと子オフセット情報が含まれます
- 左右の子ノードの存在を示すフラグを抽出します
- 右の子ノードへの相対オフセットを抽出します (存在する場合)

ステップ 4b: ナビゲーション方向の決定

- 現在のビット位置をターゲットインデックスに対してテストします
- ターゲットインデックスでビットが設定されている場合、右の子にナビゲートを試みます
- ターゲットインデックスでビットがクリアされている場合、左の子にナビゲートを試みます

ステップ 4c: ナビゲーションパスの追跡

- 目的の子が存在する場合（適切なフラグで示される）、現在の位置を更新します
- 右の子ナビゲーションの場合、エンコードされたオフセットを現在の位置に加算します
- 左の子ナビゲーションの場合、現在のノードの直後の位置に移動します
- ナビゲーションが成功した場合、次のビットレベルに進みます

ステップ 5: 要素位置の返却

- 探索が成功した場合、格納されたデータを指す最終オフセット位置を返します。
- 探索が成功しない場合（子ノードが存在しない）、要素は配列内に見つからず、失敗ステータスを返します。

ハッシュテーブル

概念的に、ネイティブハッシュテーブルは、テーブルの次元を記述するヘッダー、キーのハッシュ値をバケット (bucket) にマッピングするテーブル、および値を格納するバケットのリストで構成されます。これら 3 つのものはフォーマット内で連続して格納されます。

ルックアップを高速にするために、バケット数は常に 2 のべき乗です。テーブルは単純に $(1 + \text{バケット数})$ セルのシーケンスです。最初の (バケット数) セルについては、ネイティブハッシュテーブル全体の先頭からのバケットリストのオフセットを格納します。最後のセルはバケットの終端へのオフセットを格納します。エントリは、 $2^x = (\text{バケット数})$ として、最下位バイトにないハッシュの x 個の最下位ビットを使用してバケットにマッピングされます。たとえば、 $x=2$ の場合、32 ビットハッシュの以下の x でマークされたビットが使用されます: `b00000000_00000000_000000XX_00000000`。

物理的には、ヘッダーは單一バイトです。最上位 6 ビットは、バケット数の 2 を底とする対数を格納するために使用されます。残りの 2 ビットは、以下で説明するエントリサイズの格納に使用されます:

バケットリストへのオフセットは多くの場合小さい数値であるため、テーブルセルのサイズは可変です。1 バイト、2 バイト、または 4 バイトのいずれかです。3 つのケースは 2 ビットで記述されます。`00` は 1 バイト、`01` は 2 バイト、`10` は 4 バイトを意味します。

残りのデータはエントリです。エントリには、ハッシュコードの最下位バイトのみと、ハッシュテーブルに格納された実際のオブジェクトへのオフセットが含まれます。エントリはハッシュコードでソートされています。

ルックアップを実行するには、まずヘッダーを読み、ハッシュコードを計算し、バケット数を使用してハッシュコードからマスクするビット数を決定し、適切なポインタサイズを使用してテーブル内で検索し、バケットリストを見つけ、次のバケットリスト（またはテーブルの終端）を見つけて停止位置を知り、そのリスト内のエントリを検索します。ヒットした場合はオブジェクトが見つかり、そうでなければミスです。

すべての値を列挙するには、最初のエントリからハッシュテーブルの終端まで単純にウォークします。

これを実際に確認するために、以下のオブジェクトをネイティブハッシュテーブルに配置した例を見てみましょう。

オブジェクト	ハッシュコード
P	0x1231
Q	0x1232
R	0x1234
S	0x1338

バケット数を 2 に決定した場合、9 番目のビットのみがテーブルのインデックスに使用され、ハッシュテーブル全体は以下のようになります:

パート	オフセット	内容	意味
ヘッダー	0	0x04	これはヘッダーで、最下位ビットが <code>00</code> のため、テーブルセルは 1 バイトです。最上位 6 ビットは 1 を表し、バケット数は $2^1 = 2$ を意味します。
テーブル	1	0x04	これは符号なし整数 4 の表現で、ハッシュコード <code>0</code> に対応するバケットのオフセットに対応します。
テーブル	2	0x0A	これは符号なし整数 10 の表現で、ハッシュコード <code>1</code> に対応するバケットのオフセットに対応します。
テーブル	3	0x0C	これは符号なし整数 12 の表現で、ハッシュテーブル全体の終端のオフセットに対応します。
バケット1	4	0x31	これは P のハッシュコードの最下位バイトです
バケット1	5	P	これはオブジェクト P へのオフセットです
バケット1	6	0x32	これは Q のハッシュコードの最下位バイトです
バケット1	7	Q	これはオブジェクト Q へのオフセットです
バケット1	8	0x34	これは R のハッシュコードの最下位バイトです
バケット1	9	R	これはオブジェクト R へのオフセットです
バケット2	10	0x38	これは S のハッシュコードの最下位バイトです
バケット2	11	S	これはオブジェクト S へのオフセットです

ヘルパー呼び出し

READYTORUN_FIXUP_Helper がサポートするヘルパー呼び出しの一覧:

C++

```
enum ReadyToRunHelper
{
    READYTORUN_HELPER_Invalid          = 0x00,
    // Not a real helper - handle to current module passed to delay load helpers.
    READYTORUN_HELPER_Module          = 0x01,
    READYTORUN_HELPER_GScookie        = 0x02,
    //
    // Delay load helpers
    //
    // All delay load helpers use custom calling convention:
    // - scratch register - address of indirection cell. 0 = address is inferred from callsite.
    // - stack - section index, module handle
    READYTORUN_HELPER_DelayLoad_MethodCall      = 0x08,
    READYTORUN_HELPER_DelayLoad_Helper           = 0x10,
    READYTORUN_HELPER_DelayLoad_Helper_Obj       = 0x11,
    READYTORUN_HELPER_DelayLoad_Helper_ObjObj   = 0x12,
    //
    // JIT helpers
    //
    // Exception handling helpers
    READYTORUN_HELPER_Throw                 = 0x20,
    READYTORUN_HELPER_Rethrow              = 0x21,
    READYTORUN_HELPER_Overflow              = 0x22,
    READYTORUN_HELPER_RngChkFail          = 0x23,
    READYTORUN_HELPER_FailFast             = 0x24,
    READYTORUN_HELPER_ThrowNullRef         = 0x25,
    READYTORUN_HELPER_ThrowDivZero         = 0x26,
    READYTORUN_HELPER_ThrowExact           = 0x27,
    //
    // Write barriers
    READYTORUN_HELPER_WriteBarrier        = 0x30,
    READYTORUN_HELPER_CheckedWriteBarrier  = 0x31,
    READYTORUN_HELPER_ByRefWriteBarrier   = 0x32,
    //
    // Array helpers
    READYTORUN_HELPER_Stelem_Ref          = 0x38,
    READYTORUN_HELPER_Ldelema_Ref         = 0x39,
    //
    READYTORUN_HELPER_MemSet               = 0x40,
    READYTORUN_HELPER_MemCpy               = 0x41,
    //
    // Get string handle lazily
    READYTORUN_HELPER_GetString           = 0x50, // Unused since READYTORUN_MAJOR_VERSION 17.0
    //
    // Used by /Tuning for Profile optimizations
    READYTORUN_HELPER_LogMethodEnter     = 0x51, // Unused since READYTORUN_MAJOR_VERSION 10.0
    //
    // Reflection helpers
    READYTORUN_HELPER_GetRuntimeTypeHandle = 0x54,
```

```

READYTORUN_HELPER_GetRuntimeMethodHandle = 0x55,
READYTORUN_HELPER_GetRuntimeFieldHandle = 0x56,

READYTORUN_HELPER_Box = 0x58,
READYTORUN_HELPER_Box_Nullable = 0x59,
READYTORUN_HELPER_Unbox = 0x5A,
READYTORUN_HELPER_Unbox_Nullable = 0x5B,
READYTORUN_HELPER_NewMultiDimArr = 0x5C,

// Helpers used with generic handle lookup cases
READYTORUN_HELPER_NewObject = 0x60,
READYTORUN_HELPER_NewArray = 0x61,
READYTORUN_HELPER_CheckCastAny = 0x62,
READYTORUN_HELPER_CheckInstanceAny = 0x63,
READYTORUN_HELPER_GenericGcStaticBase = 0x64,
READYTORUN_HELPER_GenericNonGcStaticBase = 0x65,
READYTORUN_HELPER_GenericGcTlsBase = 0x66,
READYTORUN_HELPER_GenericNonGcTlsBase = 0x67,
READYTORUN_HELPER_VirtualFuncPtr = 0x68,
READYTORUN_HELPER_IsInstanceOfException = 0x69,
READYTORUN_HELPER_NewMaybeFrozenArray = 0x6A,
READYTORUN_HELPER_NewMaybeFrozenObject = 0x6B,

// Long mul/div/shift ops
READYTORUN_HELPER_LMul = 0xC0,
READYTORUN_HELPER_LMulOfv = 0xC1,
READYTORUN_HELPER_ULMulOfv = 0xC2,
READYTORUN_HELPER_LDiv = 0xC3,
READYTORUN_HELPER_LMod = 0xC4,
READYTORUN_HELPER_ULDiv = 0xC5,
READYTORUN_HELPER_ULMod = 0xC6,
READYTORUN_HELPER_LLsh = 0xC7,
READYTORUN_HELPER_LRsh = 0xC8,
READYTORUN_HELPER_LRsz = 0xC9,
READYTORUN_HELPER_Lng2Dbl = 0xCA,
READYTORUN_HELPER_ULng2Dbl = 0xCB,

// 32-bit division helpers
READYTORUN_HELPER_Div = 0xCC,
READYTORUN_HELPER_Mod = 0xCD,
READYTORUN_HELPER_Udiv = 0xCE,
READYTORUN_HELPER_UMod = 0xCF,

// Floating point conversions
READYTORUN_HELPER_Dbl2Int = 0xD0, // Unused since READYTORUN_MAJOR_VERSION 15.0
READYTORUN_HELPER_Dbl2IntOvf = 0xD1,
READYTORUN_HELPER_Dbl2Lng = 0xD2,
READYTORUN_HELPER_Dbl2LngOvf = 0xD3,
READYTORUN_HELPER_Dbl2UInt = 0xD4, // Unused since READYTORUN_MAJOR_VERSION 15.0
READYTORUN_HELPER_Dbl2UIntOvf = 0xD5,
READYTORUN_HELPER_Dbl2ULng = 0xD6,
READYTORUN_HELPER_Dbl2ULngOvf = 0xD7,
READYTORUN_HELPER_Lng2Flt = 0xD8,
READYTORUN_HELPER_ULng2Flt = 0xD9,

// Floating point ops
READYTORUN_HELPER_DblRem = 0xE0,
READYTORUN_HELPER_FltRem = 0xE1,
READYTORUN_HELPER_DblRound = 0xE2, // Unused since READYTORUN_MAJOR_VERSION 10.0

```

```
READYTORUN_HELPER_FltRound           = 0xE3, // Unused since READYTORUN_MAJOR_VERSION 10.0

#ifndef _TARGET_X86_
// Personality routines
READYTORUN_HELPER_PersonalityRoutine      = 0xF0,
READYTORUN_HELPER_PersonalityRoutineFilterFunclet = 0xF1,
#endif

//
// Deprecated/legacy
//

// JIT32 x86-specific write barriers
READYTORUN_HELPER_WriteBarrier_EAX        = 0x100,
READYTORUN_HELPER_WriteBarrier_EBX        = 0x101,
READYTORUN_HELPER_WriteBarrier_ECX        = 0x102,
READYTORUN_HELPER_WriteBarrier_ESI        = 0x103,
READYTORUN_HELPER_WriteBarrier_EDI        = 0x104,
READYTORUN_HELPER_WriteBarrier_EBP        = 0x105,
READYTORUN_HELPER_CheckedWriteBarrier_EAX = 0x106,
READYTORUN_HELPER_CheckedWriteBarrier_EBX = 0x107,
READYTORUN_HELPER_CheckedWriteBarrier_ECX = 0x108,
READYTORUN_HELPER_CheckedWriteBarrier_ESI = 0x109,
READYTORUN_HELPER_CheckedWriteBarrier_EDI = 0x10A,
READYTORUN_HELPER_CheckedWriteBarrier_EBP = 0x10B,

// JIT32 x86-specific exception handling
READYTORUN_HELPER_EndCatch               = 0x110,
};
```

参考文献

[ECMA-335](#)

ReadyToRun プラットフォームネイティブエンベロープ

原文

この章の原文は [ReadyToRun Platform Native Envelope](#) です。

.NET 10 まで、ReadyToRun (R2R) はすべてのプラットフォームでネイティブエンベロープ (envelope) として Windows PE 形式を使用しています。そのため、非 Windows プラットフォームでは、.NET ローダーが必要なフィックスアップ (fixup) とコードの有効化を行いながら PE ファイルを読み込みます。

初心者向け補足

PE (Portable Executable) 形式は Windows 独自の実行可能ファイル形式です。.NET ではこれまで、macOS や Linux 上でも PE 形式のファイルをラップして R2R イメージを配布していました。「エンベロープ (envelope)」とは、ネイティブコードを包む外側のファイル形式のこと指します。この章では、macOS 向けに Mach-O 形式という macOS ネイティブの形式をサポートする計画について説明します。

.NET 11 では、PE 形式を超えたサポートの追加を開始する予定です。対象とするサポートは以下のとおりです：

- コンポジット (composite) R2R のみ
- `crossgen2` が出力する Mach-O オブジェクトファイル
- ランタイムが Mach-O 共有ライブラリであるコンポジット R2R イメージを使用すること
 - オブジェクトファイルを共有ライブラリにリンクする処理は SDK が担当することを想定しており、このドキュメントでは扱いません。

以下に暫定的なハイレベル設計の概要を示します。このサポートの実装に伴い、このドキュメントはより詳細に更新されるべきであり、[ReadyToRun 概要](#) および [ReadyToRun フォーマット](#) も変更を反映して更新されるべきです。

crossgen2: Mach-O オブジェクトファイルの生成

Mach-O サポートは、ターゲット OS が macOS の場合に限り、コンポジット ReadyToRun でのみサポートされます。新しい `crossgen2` フラグによりオプトインで有効化します：

- `--obj-format macho`

`crossgen2` は以下を行います：

- `READYTORUN_HEADER` の `RTR_HEADER` エクスポートを含むコンポジット R2R イメージとして Mach-O オブジェクトファイルを生成する。
- 各入力 IL アセンブリをコンポーネント R2R アセンブリとしてマークする： `READYTORUN_FLAG_COMPONENT`。
- 各入力 IL アセンブリに、関連するコンポジットイメージがプラットフォームネイティブ形式であることを示す新しいフラグを設定する： `READYTORUN_FLAG_PLATFORM_NATIVE_IMAGE`

`crossgen2` は最終的な共有ライブラリを生成しません。別途 SDK / ビルドのリンクステップで、最終的な `dylib` に `RTR_HEADER` エクスポートを保持する必要があります。

💡 初心者向け補足

コンポジット R2R とは、複数の .NET アセンブリのネイティブコードを1つの R2R イメージにまとめたものです。`crossgen2` は .NET の AOT（事前コンパイル）ツールで、IL コードからネイティブコードを含む R2R イメージを生成します。ここでは `crossgen2` がまず Mach-O オブジェクトファイル（`.o`）を出力し、それを Apple のリンク（`ld` など）で共有ライブラリ（`.dylib`）にリンクするという2段階の流れになります。

Mach-O エミッタの設計判断

R2R フォーマットには Mach-O フォーマットではネイティブに表現できないケースがいくつかあり、エミュレーションが必要です。このセクションでは、Mach-O R2R フォーマットに関する設計判断について説明します。

セクション

`__TEXT, __text` から移動されるデータ：

- プリコンパイル済みマネージドコードは `__TEXT, __managedcode` に移動されます。`__TEXT, __text` はリンクから特別な扱いを受けるため、`__TEXT, __managedcode` を使用します。これは NativeAOT と一致します。
- ジャンプテーブル、CLR メタデータ、Win32 リソース、マネージドアンワインド情報 (unwind info)、GC 情報、R2R ヘッダーなどの読み取り専用データは `__TEXT, __const` に移動されます。

PE エンベロープと対応する場所に留まるデータ：

- フィックスアップテーブルなどの読み書き可能データ：`__DATA, __data`
- インポートサンク (import thunk)：`__TEXT, __text`

リロケーション

シンボル範囲 (symbol range) は、Mach-O では他のプラットフォームとは異なる方法で表現されます。Apple のリンクは、同じ場所に複数のシンボルが定義されている場合に問題が発生します。さらに、Mach フォーマットは2つのシンボル間の距離を表現する「サブトラクタ (subtractor)」リロケーションをネイティブにサポートしています。その結果、シンボル範囲の開始を範囲の開始シンボルとして表現できます。範囲のサイズは「終了シンボルの位置 - 開始シンボルの位置 + 終了シンボルのサイズ」として表現できます。

ベースシンボルと RVA

R2R フォーマットは PE フォーマットと同様に、イメージのベースシンボルに加算できる RVA (Relative Virtual Address、相対仮想アドレス) をイメージに出力することに大きく依存しています。COFF オブジェクトファイル形式はこの概念をネイティブにサポートしており、PE フォーマットも PE ヘッダーでこの概念を使用しています。しかし、他のフォーマットはこの概念をネイティブにサポートしていません。

💡 初心者向け補足

RVA（相対仮想アドレス）とは、イメージがメモリに読み込まれたベースアドレスからの相対的なオフセットです。例えば、ベースアドレスが `0x10000` でメソッドが `0x10500` にある場合、RVA は `0x500` になります。PE 形式ではこの仕組みが組み込まれていますが、Mach-O 形式では同等の機能をエミュレーションする必要があります。

Apple のリンクは Mach フォーマット用のベースシンボルを提供していますが、そのベースシンボルは出力タイプに依存し、一般的に `_mh_<output>_header` の形式になります。dylib の場合、シンボルは `_mh_dylib_header` です。このシンボルはベースアドレスとして `dlinfo` や `dladdr` が返すアドレスに位置しています。また、Mach ヘッダーを指しており、R2R データの読み取り範囲を制限するためにイメージのサイズを確認するのにも使用できます。

その結果、Mach フォーマットでこのサポートを容易にエミュレーションできます：

1. オブジェクトライター (object writer) で使用するベースシンボルは `_mh_dylib_header` とする。
2. ベースシンボルからの距離を出力するために、サブトラクタリロケーション (subtractor relocation) を使用して「シンボルの位置 - `_mh_dylib_header` の位置」を表現する。

ランタイム: プラットフォームネイティブ R2R イメージの使用

ランタイムは、アセンブリの読み込み時にプラットフォームネイティブ R2R イメージを処理するように更新されます。

1. IL アセンブリを読み込み、R2R アセンブリかどうかを判定する。
2. コンポーネント R2R アセンブリでない場合、既存の R2R ロードロジックで処理を続ける。
 - このシナリオではプラットフォームネイティブサポートは提供されない。
3. 新しい `READYTORUN_FLAG_PLATFORM_NATIVE_IMAGE` フラグが設定されたコンポーネント R2R アセンブリの場合：
 - a. `OwnerCompositeExecutable` の値を読み取る。
 - b. コンポーネントアセンブリのパスとオーナーコンポジット名を使用してホストコールバックを呼び出す。
 - c. 成功した場合、コンポジットの `READYTORUN_HEADER`へのポインタを取得し、ネイティブメソッドのルックアップ / フィックスアップに使用する。
 - d. 失敗した場合、IL/JIT パスにフォールバックする。
4. プラットフォームネイティブフラグが設定されていない場合、既存の R2R ロードロジック (PE アセンブリの検索と読み込み) で処理を続ける。

ホストコールバック

`host_runtime_contract` に、ネイティブコード情報を取得するための新しいコールバックが追加されます。

```
struct native_code_context
{
    size_t size; // この構造体のサイズ
    const char* assembly_path; // コンポーネントアセンブリのパス
    const char* owner_composite_name; // コンポーネント R2R ヘッダーからの名前
};

struct native_code_data
{
    size_t size; // この構造体のサイズ
    void* r2r_header_ptr; // ReadyToRun ヘッダー
    size_t image_size; // イメージのサイズ
```

```
    void* image_base;      // イメージが読み込まれたベースアドレス
};

bool get_native_code_data(
    const struct native_code_context* context,
    /*out*/ struct native_code_data* data
);
```

プラットフォームネイティブイメージの実際の読み込み（たとえば共有ライブラリの `dlopen`、ホスト自体に静的リンクされたものの使用など）はホストに委ねられます。また、必要なキャッシュ処理もホストが担当します。

共有ジェネリクスの設計

原文

この章の原文は [Shared Generics Design](#) です。

著者: Fadi Hanna - 2019

はじめに

共有ジェネリクス (Shared Generics) は、さまざまなインスタンス化 (instantiation) を持つジェネリックメソッド (generic method) に対してランタイムが生成するコード量を削減することを目的とした、ランタイムと JIT の連携機能です（ジェネリック型のメソッドとジェネリックメソッドの両方をサポートします）。この機能の基本的な考え方は、特定のインスタンス化においては、生成されるコードがごくわずかな命令を除いてほぼ同一になるため、メモリフィットプリントの削減とジェネリックメソッドの JIT コンパイルにかかる時間を短縮するために、ランタイムが単一の特別な正規バージョン (canonical version) のコードを生成し、そのメソッドの互換性のあるすべてのインスタンス化で共用できるようにする、というものです。

💡 初心者向け補足

ジェネリクスとは、`List<T>` のように型パラメータを使ってさまざまな型に対応できる仕組みです（Java のジェネリクスに似ています）。たとえば `List<string>` と `List<object>` はどちらも `List<T>` のインスタンス化ですが、参照型同士であればメモリレイアウトが同じなので、同じ機械語コードを使い回すことができます。これが「共有ジェネリクス」の核心です。

正規コード生成とジェネリックディクショナリ

以下の C# コードサンプルを考えてみましょう：

```
string Method<T>()
{
    return typeof(List<T>).ToString();
}
```

C#

共有ジェネリクスがない場合、`Method<object>` や `Method<string>` のようなインスタンス化のコードは、1つの命令を除いて同一になります。その命令とは、`List<T>` 型の正しいタイプハンドル (TypeHandle) をロードするものです：

```
mov rcx, type handle of List<string> or List<object>
call ToString()
ret
```

asm

共有ジェネリクスでは、正規コード (canonical code) に `List<T>` のタイプハンドルのハードコードされたバージョンは含まれず、代わりにランタイムヘルパー API の呼び出しから、実行中の `Method<T>` のインスタンス化のジェネリックディクショナリ (*generic dictionary*) からロードすることで、正確なタイプハンドルを検索します。コードは以下のようになります：

```
asm
mov rcx, generic context
mov rcx, [rcx + offset of InstantiatedMethodDesc::m_pPerInstInfo]
mov rcx, [rcx + dictionary slot containing type handle of List<T>]
call ToString()
ret
```

// Method<string> または Method<object>
// これがジェネリックディクショナリ

この例におけるジェネリックコンテキスト (generic context) は、 Method<object> または Method<string> の InstantiatedMethodDesc です。ジェネリックディクショナリ (generic dictionary) とは、共有ジェネリックコードがインスタンス化固有の情報を取得するために使用するデータ構造です。基本的には配列であり、そのエントリにはインスタンス化固有のタイプハンドル (type handle)、メソッドハンドル (method handle)、フィールドハンドル (field handle)、メソッドエントリポイン (method entry point) などが格納されています。MethodTable および InstantiatedMethodDesc 構造体の「PerInstInfo」フィールドは、それぞれジェネリック型とジェネリックメソッドのジェネリックディクショナリ構造体を指しています。

この例では、 Method<object> のジェネリックディクショナリには List<object> 型のタイプハンドルを持つスロットが含まれ、 Method<string> のジェネリックディクショナリには List<string> 型のタイプハンドルを持つスロットが含まれます。

この機能は現在、参照型 (reference type) のインスタンス化でのみサポートされています。参照型はすべて同じサイズ・プロパティ・レイアウトなどを持つためです。プリミティブ型 (primitive type) や値型 (value type) のインスタンス化については、ランタイムがインスタンス化ごとに個別のコード本体を生成します。

💡 初心者向け補足

参照型 (string 、 object 、ユーザー定義のクラスなど) は、すべてポインタとして表現されるため、メモリ上のサイズが同じです。そのため Method<string> と Method<object> は同じ機械語コードを共有できます。一方、値型 (int 、 double 、ユーザー定義の構造体など) はそれぞれサイズが異なるため、個別にコードを生成する必要があります。Java ではジェネリクスが型消去 (type erasure) によって実装されますが、.NET ではこのように実行時の型情報を保持しつつ、参照型についてはコード共有で効率化しています。

レイアウトとアルゴリズム

型とメソッドにおけるディクショナリポインタ

特定のジェネリックメソッドが使用するディクショナリは、そのメソッドの InstantiatedMethodDesc 構造体の m_pPerInstInfo フィールドによってポイントされています。これはジェネリックディクショナリデータの内容への直接ポインタです。

ジェネリック型では、もう 1 段階の間接参照 (indirection) があります。 MethodTable 構造体の m_pPerInstInfo フィールドは、ディクショナリのテーブルへのポインタであり、そのテーブルの各エントリが実際のジェネリックディクショナリデータへのポインタです。これは、型には継承があり、派生ジェネリック型 (derived generic type) は基底型 (base type) のディクショナリを継承するためです。

以下に例を示します:

```
C#
class BaseClass<T> { }

class DerivedClass<U> : BaseClass<U> { }

class AnotherDerivedClass : DerivedClass<string> { }
```

これらの各型の MethodTable は以下のようになります:

BaseClass[T] の MethodTable
...
<code>m_PerInstInfo</code> : 以下のディクショナリテーブルを指す
...
<code>dictionaryTable[0]</code> : 以下のディクショナリデータを指す
<code>BaseClass</code> のディクショナリデータ

DerivedClass[U] の MethodTable
...
<code>m_PerInstInfo</code> : 以下のディクショナリテーブルを指す
...
<code>dictionaryTable[0]</code> : <code>BaseClass</code> のディクショナリデータを指す
<code>dictionaryTable[1]</code> : 以下のディクショナリデータを指す
<code>DerivedClass</code> のディクショナリデータ

AnotherDerivedClass の MethodTable
...
<code>m_PerInstInfo</code> : 以下のディクショナリテーブルを指す
...
<code>dictionaryTable[0]</code> : <code>BaseClass</code> のディクショナリデータを指す
<code>dictionaryTable[1]</code> : <code>DerivedClass</code> のディクショナリデータを指す

`AnotherDerivedClass` はジェネリック型ではないため、独自のディクショナリを持たず、基底型のディクショナリポインタを継承していることに注意してください。

ディクショナリスロット

前述のとおり、ジェネリックディクショナリはインスタンス化固有の情報を含む複数のスロット (slot) の配列です。あるジェネリック型またはメソッドに対してディクショナリが最初にアロケート (allocate) されると、すべてのスロットは `NULL` で初期化され、コードの実行に応じて遅延的にオンデマンドで設定されます（`Dictionary::PopulateEntry(...)` を参照）。

インスタンス化の最初の N 個のスロット (N は引数の数) には、常にインスタンス化の型引数のタイプハンドルが格納されます（これは一種の最適化もあります）。それに続くスロットには、インスタンス化に基づく情報が格納されます。

たとえば、先ほどの `Method<string>` の例におけるジェネリックディクショナリの内容は以下のとおりです：

Method<string> のディクショナリ
slot[0]: TypeHandle(string)
slot[1]: ディクショナリの合計サイズ
slot[2]: TypeHandle(List<string>)
slot[3]: NULL (未使用)
slot[4]: NULL (未使用)

注 サイズスロットはジェネリックコードによって使用されることではなく、動的ディクショナリ拡張 (*dynamic dictionary expansion*) 機能の一部です。詳しくは後述します。

このディクショナリが最初にアロケートされた時点では、`slot[0]` のみが初期化されています。これはインスタンス化の型引数を含んでいるためです（もちろん、動的ディクショナリ拡張機能によりサイズスロットも初期化されます）。残りのスロット（たとえば `slot[2]`）は `NULL` であり、それらを使用するコードパスに到達した場合に遅延的に値が設定されます。

まだ `NULL` であるスロットから情報をロードする際、ジェネリックコードはディクショナリスロットに値を設定するために、以下のランタイムヘルパー関数のいずれかを呼び出します：

- `JIT_GenericHandleClass` : ジェネリック型ディクショナリの値を検索するために使用されます。このヘルパーは、ジェネリック型のすべてのインスタンスマソッドで使用されます。
- `JIT_GenericHandleMethod` : ジェネリックメソッドディクショナリの値を検索するために使用されます。このヘルパーは、すべてのジェネリックメソッド、またはジェネリック型の非ジェネリックな静的メソッドで使用されます。

共有ジェネリックコードを生成する際、JIT は `DictionaryLayout` の実装 (`genericdict.cpp`) を利用して、各種検索に使用するスロットと各スロットに含まれる情報の種類を把握します。

💡 初心者向け補足

ジェネリックディクショナリは、いわば「型情報の引き出し」のようなものです。`Method<string>` が実行されるとき、JIT が生成した共有コードは「今自分がどの型で動いているか」をこのディクショナリから調べます。たとえば `List<T>` の `T` が何であるかを知りたい場合、ハードコードする代わりにディクショナリのスロットを参照します。必要になるまでスロットを埋めない「遅延設定」方式により、使わない情報のための無駄な初期化を避けています。

ディクショナリレイアウト

`DictionaryLayout` 構造体は、ディクショナリ検索を実行する際にどのスロットを使用するかを JIT に指示するものです。この `DictionaryLayout` 構造体には、いくつかの重要な特性があります：

- これは、ある型またはメソッドの互換性のあるすべてのインスタンス化で共有されます。言い換えると、ディクショナリレイアウトは型またはメソッドの正規インスタンス化 (*canonical instantiation*) に関連付けられています。たとえば上記の例では、

`Method<object>` と `Method<string>` は互換性のあるインスタンス化であり、それそれが個別のディクショナリを持っていますが、すべてが正規インスタンス化 `Method<__Canon>` に関連付けられた同じディクショナリレイアウトを共有しています。

- ジェネリック型またはメソッドのディクショナリは、そのディクショナリレイアウトと同じ数のスロットを持ちます。注: 歴史的には、動的ディクショナリ拡張機能の導入以前は、ジェネリックディクショナリがレイアウトよりも小さい場合があり、特定の検索ではランタイムヘルパー API を呼び出す必要がありました（スローパス）。

ジェネリック型またはメソッドが最初に作成されたとき、そのディクショナリレイアウトには「未割り当て」のスロットが含まれています。割り当ては、JIT がディクショナリ検索シーケンスを発行する必要があるときに、コード生成の一部として行われます。この割り当ては `DictionaryLayout::FindToken(...)` API の呼び出し中に行われます。スロットが一度割り当てられると、それは特定のシグネチャ (signature) に関連付けられ、そのシグネチャはそのスロットインデックスにおいてすべてのインスタンス化されたディクショナリに格納される値の種類を記述します。

入力シグネチャが与えられた場合、スロットの割り当ては以下のアルゴリズムで実行されます:

```
slot = 0 から開始
ディクショナリレイアウトの各エントリに対して
If entry.signature != NULL
    If entry.signature == inputSignature
        return slot
    EndIf
Else
    entry.signature = inputSignature
    return slot
EndIf
slot++
EndForEach
```

では、上記のアルゴリズムが実行されたが、同じシグネチャを持つ既存のスロットが見つからず、しかも「未割り当て」のスロットが尽きた場合はどうなるでしょうか？ここで動的ディクショナリ拡張 (dynamic dictionary expansion) が登場し、レイアウトにスロットを追加してリサイズし、このレイアウトに関連付けられたすべてのディクショナリもリサイズします。

動的ディクショナリ拡張

歴史的背景

動的ディクショナリ拡張機能の導入以前は、ディクショナリレイアウトはバケット (bucket)（固定サイズの `DictionaryLayout` 構造体の連結リスト）で構成されていました。初期レイアウトバケットのサイズは、ジェネリック型についてはヒューリスティクスに基づいて計算された数値に常に固定され、ジェネリックメソッドについては常に 4 スロットに固定されていました。ジェネリック型とメソッドには、検索に使用できる固定サイズのジェネリックディクショナリもありました（「高速検索スロット (fast lookup slots)」とも呼ばれます）。

バケットがエントリで埋まると、新しい `DictionaryLayout` バケットをアロケートしてリストに追加するだけでした。しかし問題は、型やメソッドのジェネリックディクショナリはすでに固定サイズでアロケートされているためリサイズできず、JIT はディクショナリの連結リストに間接参照できる命令の生成をサポートしていなかったことです。この制約により、固定数の値（最初の `DictionaryLayout` バケットのエントリに関連付けられたもの）についてのみジェネリックディクショナリを検索でき、それ以外の検索についてはより低速なランタイムヘルパーを経由する必要がありました。

これは、[ReadyToRun](#) と階層コンパイル (Tiered Compilation) 技術が導入されるまでは許容できるものでした。ReadyToRun コードによってスロットが急速に割り当てられ、ランタイムがパフォーマンス向上のために特定のメソッドを再 JIT コンパイルしたとき、残りの「高速検索スロット」が見つからない場合があり、より低速なランタイムヘルパーを経由するコードを生成せざるを得ませんでした。これ

により一部のシナリオでパフォーマンスが低下し、ReadyToRun コードでは高速検索スロットを使用せず、再 JIT コンパイルされたコード用に予約しておくという決定がなされました。しかし、この決定は ReadyToRun のパフォーマンスを低下させるものでしたが、R2R のスループットよりも再 JIT コンパイルされたコードのスループットをより重視していたため、必要な妥協でした。

このような理由から、動的ディクショナリ拡張機能が導入されました。

💡 初心者向け補足

ReadyToRun (R2R) は、アプリケーションの起動時間を短縮するための事前コンパイル (AOT) 技術です。階層コンパイル (Tiered Compilation) は、最初は高速に起動できるコードを生成し、頻繁に実行されるメソッドについてのみ最適化された再コンパイルを行う仕組みです。これらの技術では多くのジェネリックインスタンス化が使用されるため、ディクショナリのスロットが不足しやすく、動的に拡張できる仕組みが必要になりました。

説明とアルゴリズム

この機能のコンセプトはシンプルです：ディクショナリレイアウトをバケットの連結リストから、動的に拡張可能な配列に変更する、というものです。シンプルに聞こえますが、実装には細心の注意が必要でした。その理由は以下のとおりです：

- `DictionaryLayout` 構造体だけをリサイズすることはできません。レイアウトのサイズが実際のジェネリックディクショナリのサイズより大きい場合、JIT がディクショナリデータのサイズと一致しない間接参照命令を生成し、アクセス違反 (access violation) を引き起こします。
- 型やメソッドのジェネリックディクショナリだけをリサイズすることもできません：
 - 型の場合、ジェネリックディクショナリは `MethodTable` 構造体の一部であり、マネージドコードで既に使用中のため再アロケートできません。
 - メソッドの場合、ジェネリックディクショナリは `MethodDesc` 構造体の一部ではありませんが、ジェネリックコードで使用中の可能性があります。
 - いずれにしても、同じ型やメソッドに対して複数の `MethodTable` や `MethodDesc` を持つことはできないため、再アロケートは選択肢にありません。
- 単一のインスタンス化のジェネリックディクショナリだけをリサイズすることもできません。たとえば上記の例で、`Method<string>` のディクショナリを拡張したいとします。レイアウトのリサイズは、JIT が `Method<__Canon>` に対して生成する共有正規コード (shared canonical code) に影響を与えます。`Method<string>` のディクショナリだけをリサイズした場合、共有ジェネリックコードはそのインスタンス化でのみ動作しますが、`Method<object>` のような他のインスタンス化で使用しようとすると、JIT コンパイルされた命令がディクショナリ構造のサイズと一致しなくなり、アクセス違反を引き起こします。
- ランタイムはマルチスレッド (multi-threaded) であり、これが複雑さをさらに増します。

現在の実装では、シンプルさを保つために、ディクショナリレイアウトと実際のディクショナリの拡張を別々に行ってています：

- ディクショナリレイアウトは、空のスロットがなくなったときに拡張されます。[genericdict.cpp](#) の `DictionaryLayout::FindToken()` の実装を参照してください。
- インスタンス化された型とメソッドのディクショナリは、その型またはメソッドのディクショナリのサイズを超えるスロットの値をコードが読み取ろうとしたときに、遅延的にオンデマンドで拡張されます。これは、前述のヘルパー関数（`JIT_GenericHandleClass` および `JIT_GenericHandleMethod`）の呼び出しを通じて行われます。

ディクショナリアクセスのコード生成は、以下のコードと同等です (JIT コンパイルされたコードと ReadyToRun コードの両方で同じ) :

```
void* pMethodDesc = <some value>; // インスタンス化されたジェネリックメソッドの入力 MethodDesc
int requiredOffset = <some value>; // アクセスする必要のあるオフセット
```

C++

```
void* pDictionary = pMethodDesc->m_pPerInstInfo;

// 'requiredOffset' で間接参照する前に、まずディクショナリサイズを確認していることに注目
if (pDictionary[sizeOffset] <= requiredOffset || pDictionary[requiredOffset] == NULL)
    pResult = JIT_GenericHandleMethod(pMethodDesc, <signature>);
else
    pResult = pDictionary[requiredOffset];
```

このサイズチェックは、ディクショナリから値を読み取るたびに無条件に実行されるわけではありません。もしそうすると、顕著なパフォーマンス低下を引き起こすことになります。ディクショナリレイアウトが最初にアロケートされたとき、初期に割り当てられたスロット数を記録しておく、その初期スロット数を超えるスロットの値を読み取ろうとする場合にのみサイズチェックを実行します。

型とメソッドのディクショナリは、[genericdict.cpp](#) の `Dictionary::GetTypeDictionaryWithSizeCheck()` および `Dictionary::GetMethodDictionaryWithSizeCheck()` ヘルパー関数によって拡張されます。

型に関して 1 つ注意すべき点は、型は基底型からディクショナリポインタを継承できるということです。これは、あるジェネリック型のジェネリックディクショナリをリサイズした場合、その派生型すべてに新しいディクショナリポインタを伝播させる必要があることを意味します。この伝播も、コードが派生型の MethodTable ポインタを使って `JIT_GenericHandleWorker` ヘルパー関数を呼び出す際に、遅延的に行われます。このヘルパーの中で、基底型のディクショナリポインタが更新されていることがわかった場合、それを派生型にコピーします。

💡 初心者向け補足

動的ディクショナリ拡張の設計上、最も困難な点は「すべてのインスタンス化のディクショナリを同期的にリサイズしなければならない」ことです。たとえば `Method<string>` のディクショナリだけを大きくしても、同じ共有コードを使う `Method<object>` のディクショナリが古いサイズのままだとクラッシュしてしまいます。この問題を解決するため、ランタイムはレイアウトの拡張とディクショナリの拡張を分離し、各ディクショナリは実際にアクセスされたタイミングで遅延的にリサイズされます。古いディクショナリは安全のために解放されず、新しいディクショナリが公開された時点で以降のアクセスはすべて新しいほうを使用します。

古いディクショナリはリサイズ後に解放されませんが、新しいディクショナリが MethodTable または MethodDesc に公開されると、ジェネリックコードによるその後のディクショナリ検索はすべて、新しくアロケートされたディクショナリを使用するようになります。古いディクショナリの解放は、特にマルチスレッド環境では極めて複雑になり、有用なメリットもありません。

ランタイム開発者のためのロギング

原文

この章の原文は [Runtime logging for developers](#) です。

日付: 2024年2月

.NET ランタイムのコードベースは非常に巨大で、複数の言語にまたがる数千ものファイルに広がっています。そのコードベース全体には数千ものログメッセージが散在しており、そのほとんどはデフォルトで無効になっています。では、あらゆる不可解なテスト失敗に悩まされるランタイム開発者であるあなたは、どうすればそれらのメッセージをランタイムから取り出してコンソールやログファイルに出力できるのでしょうか？そして、新たに書いているコードに適切にログメッセージを追加するにはどうすればよいのでしょうか？

注意: この文書では多くの環境変数を繰り返し参照します。ランタイムがサポートする環境変数の詳細なリストとその役割については、[_clrconfigvalues.h](#)、[_gcconfig.h](#)、および[_jitconfigvalues.h](#) を参照してください。

ロギングの種類

EventPipe

EventPipe は、.NET ランタイムで広く使用されている、ETW に類似したクロスプラットフォームのトレーシング (tracing) システムです。EventPipe の詳細な概要については、[.NET Learn の EventPipe ドキュメントページ](#) を参照してください。EventPipe は優れたパフォーマンスと高い柔軟性を備えているため、アクティブなランタイムから情報を取得する主要な方法の一つとして推奨されています。デバッグビルドでサポートされているほとんどのイベントは、リリースビルドでも公開されています。

💡 初心者向け補足

EventPipe は、.NET アプリケーションの実行中に内部で何が起きているかを観察するための仕組みです。Java でいえば JFR (Java Flight Recorder) に近い概念です。アプリケーションのパフォーマンス問題やバグの原因を調査する際に、ランタイムが発行するイベント（ガベージコレクションの発生、JIT コンパイルの実行など）をキャプチャして分析できます。ETW (Event Tracing for Windows) は Windows 固有の仕組みですが、EventPipe はクロスプラットフォームで同様の機能を提供します。

基本的なシナリオでは、[dotnet-counters](#) ツールを使用して、ランタイムが EventPipe を通じて報告するイベントやパフォーマンスカウンター (performance counters) を監視できます。たとえば、プロジェクトの実行中にデフォルトのカウンターを収集するには、`dotnet-counters collect -- dotnet exec myapp.dll` を使用します。-- の後の部分はトレース対象のコマンドを指定します。

[dotnet-trace](#) ツールを使用して、実行中のランタイムから EventPipe イベントをリアルタイムでキャプチャすることもできます。このツールには、重大度レベルやキーワードに基づいてイベントをフィルタリングするコマンドラインオプションが用意されています。たとえば、プロジェクトの実行中に情報レベルの GC イベントをキャプチャするには、`dotnet-trace collect --clreventlevel informational --clrevents gc -- dotnet exec myapp.dll` を使用します。-- の後の部分はトレース対象のコマンドを指定します。`collect` コマンドは、`--profile` スイッチを介してアクセスできる便利なデフォルトプロファイル (profile) のセットもサポートしています。例: `--profile gc-collect` :

cpu-sampling	- CPU 使用率と一般的な .NET ランタイム情報の追跡に役立ちます。プロファイルやプロバイダーが指定されていない場合
gc-verbose	- GC コレクションを追跡し、オブジェクトアロケーションをサンプリングします。
gc-collect	- 非常に低いオーバーヘッドで GC コレクションのみを追跡します。
database	- ADO.NET と Entity Framework のデータベースコマンドをキャプチャします。

`dotnet-trace collect` コマンドは `.nettrace` ファイルを生成します。これは `dotnet-trace report topN` (CPU サンプリング情報の場合) や `dotnet-trace convert` で分析できます。また、この `.nettrace` ファイルを Visual Studio で直接開いて調査することもできます。たとえば、ソリューションエクスプローラーでダブルクリックするか、Visual Studio のウィンドウにドラッグします。

アプリケーションが `dotnet-counters` または `dotnet-trace` の監視下で実行できない場合は、`--show-child-io` ツール引数を渡して子プロセスの出力を可視化し、エラーメッセージを確認してみてください。正しい作業ディレクトリから、必要な環境変数を設定した状態で実行していることを確認してください。

上記の各ツールの使い方の詳細については、リンク先のドキュメントページを参照してください。これらのツールには、時間制限、既存のプロセスへのアタッチ、設定可能な出力フォーマットなど、豊富なオプションと便利な機能があります。

ランタイムの C++ 部分から独自の EventPipe イベントを発行するには、`FireEtwXXX` API、またはその便利なラッパーである `ETW::` C++ 名前空間内のものを使用できます。既存のイベントを使用するのではなく、新しい種類のイベントを作成することが多いでしょう。イベントは `genEventing.py` スクリプトによって、`ClrEtwAll.man` の定義に基づいて生成されます。

C# から EventPipe イベントを発行するには、`System.Diagnostics.Tracing.EventSource` クラスを使用できます。このクラスから派生した独自のイベント型を定義します。サンプルと詳細についてはそのドキュメントを参照してください。

代替イベントコンシューマー

高度なシナリオでは、EventSource と C++ イベントラッパーの両方が、それぞれのプラットフォーム上の標準的なイベントトレーシングシステム (Windows では ETW、Linux では LTTNG) をサポートしています。EventPipe 自体はこれらのシステムと統合されませんが、EventPipe の上に構築されたランタイムのレイヤーは、それらが有効になっている場合にそれらも使用します。

💡 初心者向け補足

ETW (Event Tracing for Windows) は Windows に組み込まれた高性能なトレーシング機構で、LTTNG (Linux Trace Toolkit Next Generation) は Linux 向けの同等のツールです。EventPipe はこれらとは独立したクロスプラットフォームの仕組みですが、ランタイムはこれらのネイティブなトレーシングシステムとも連携できるように設計されています。たとえば、Windows で ETW を使い慣れているチームは、既存のワークフローをそのまま .NET の診断にも活用できます。

Windows では、標準的な `ETW` ツール、たとえば `WPR` の「.NET Activity」シナリオを使用できます。Windows で ETW トレースを収集・分析するための便利な汎用ツールとして `PerfView` があります。PerfView を使用するには、その[詳細なドキュメント](#)を参照するかチュートリアルを実施するのがよいでしょう。基本的な出発点としては、Collect→Run メニューオプションからアプリケーションを起動し、生成された記録を左側のエクスプローラーペインでダブルクリックして開きます。

Linux では `LTTNG` を使用でき、`perfcollect` スクリプトは ETW に近い操作感を提供する便利なツールです。

StressLog

StressLog は、ランタイムプロセス内部の循環バッファー (circular buffer) であり、通常はプロセス外部に出力されません。ほとんどの StressLog メッセージはリテールビルド (retail build) でも利用可能であるため、本番環境での GC やその他のサブシステムに関する問題のトラブルシューティングに役立ちます。StressLog を有効にするには、環境変数 `DOTNET_StressLog` を `1` に設定し、以下に示すように環境変数を使用して設定できます：

💡 初心者向け補足

StressLog は、ランタイム内部のデバッグに特化した軽量なログ機構です。通常のログとは異なり、メモリ上の循環バッファーに書き込まれるため、ファイル I/O のオーバーヘッドがなく、パフォーマンスへの影響が最小限に抑えられます。「循環バッファー」とは、古いメッセージが新しいメッセージで上書きされる固定サイズのバッファーのことです。本番環境でも使用できるため、デバッガーをアタッチできない状況やダンプファイルを取得できない場合に特に有用です。

```
bash
echo StressLog を有効にする
set DOTNET_StressLog=1

echo JIT のログメッセージを表示する
set DOTNET_LogFacility=0x00000008

echo 警告またはエラーのみ表示する
set DOTNET_LogLevel=3

echo バッファーが急速に埋まる場合、大きなバッファーサイズを設定すると効果的です。
echo ただし、スレッド数が多い場合は、メモリ枯渇を避けるために低い制限を設定するとよいでしょう。
echo StressLog のスレッドごとのサイズ制限を 20MB に設定する。設定値はデフォルトで16進数です。
set DOTNET_StressLogSize=1400000

echo すべての StressLog (合計) のプロセス全体のサイズ制限を 400MB に設定する。設定値はデフォルトで16進数です。
set DOTNET_TotalStressLogSize=18000000

echo デバッガーをアタッチできない場合やダンプファイルを取得できない場合のために、StressLog をメモリではなくファイルに書き込む。
set DOTNET_StressLogFilename=mystresslog.log
```

C++ から StressLog に独自のメッセージを書き込むには、`STRESS_LOGN(facility, level, msg, ...)` マクロを使用できます。例：
`STRESS_LOG1(LF_GC, LL_ERROR, "A significant but non-fatal error occurred in the garbage collector! Here's my favorite number: %d\n", 42)`。最初の引数は 1 つ以上のロギングファシリティ (logging facility) (`|` を使って組み合わせ可能、例: `LF_GC | LF_GCROOTS`)、2 番目の引数は重大度レベル (severity level)、3 番目の引数はログメッセージのフォーマット文字列です。詳細については、後述の[ログファシリティとレベル](#)を参照してください。

C# から StressLog に独自のメッセージを書き込むことはできません。テスト中にどうしても必要な場合は、カスタム QCall を通じて公開することが考えられます。

従来の .NET ランタイムロギング

「従来の」ログメッセージは、ランタイムのデバッグビルド (debug build) またはチェック済みビルド (checked build) でのみ利用可能です。これを有効にするには、環境変数 `DOTNET_LogEnable` を `1` に設定し、環境変数を使用して設定します。従来のロギングにはさまざまな設定変数があり、以下に示します：

```
bash
echo 従来のロギングを有効にする
set DOTNET_LogEnable=1

echo JIT のログメッセージを表示する
set DOTNET_LogFacility=0x00000008

echo 警告またはエラーのみ表示する
set DOTNET_LogLevel=3

echo デバッガーにログメッセージを出力する
set DOTNET_LogToDebugger=0

echo コンソールにログメッセージを出力する
set DOTNET_LogToConsole=1

echo 特定のファイルにログメッセージを出力する
set DOTNET_LogToFile=0
set DOTNET_LogFile=mylog.log

echo 起動時にファイルを消去する代わりに、ログメッセージをファイルに追記する
set DOTNET_LogFileAppend=1

echo クラッシュ後にメッセージが失われないように、書き込みごとにログファイルをフラッシュする
set DOTNET_LogFlushFile=1

echo マルチプロセスシナリオのために、すべてのログメッセージにプロセス ID を付加する
set DOTNET_LogWithPid=1
```

この古典的なロギングシステムはあまり頻繁に使用されていないため、遭遇する個々のログメッセージは部分的または完全に機能しない場合があります（たとえば、古いログ文における 64 ビット専用の問題など）。ただし、基本的な機能は常に動作するはずです。

C++ から独自の従来のログメッセージを送信するには、`LOG((facility, level, msg, ...))` マクロを使用できます。例：
`LOG((LF_GC, LL_INFO01000, "An insignificant thing happened in the garbage collector.\n"))`。引数は上記の StressLog で説明したものとほぼ同等です。ファシリティとレベルの詳細については、後述の[ログファシリティとレベル](#)を参照してください。

`LOG` マクロと `STRESS_LOG` マクロは非常に似ていることに注意してください。そのため、デバッグ目的で一時的に `LOG((...))` 文を `STRESS_LOG(...)` 文に変換して、StressLog の機能を活用して問題を診断することができます。

C# から従来のログにメッセージを送信する必要がある場合は、StressLog と同様に、一時的にカスタム QCall を通じて公開することが考えられます。

ログファシリティとレベル

StressLog と従来のロギングはいずれも、環境変数 `DOTNET_LogFacility`、`DOTNET_LogFacility2`、および `DOTNET_LogLevel` に依存して、冗長度の制御とログに記録される情報のフィルタリングを行います。

`DOTNET_LogLevel` は、カテゴリに関係なく、重要度の低いログメッセージをフィルタリングできます。この変数は名前付き定数ではなく、整数で指定します。この文書の執筆時点でのレベルは以下のとおりで、`log.h` から取得されています:

```
LL_EVERYTHING 10
LL_INFO1000000 9 // 小規模だが自明でない実行で 1,000,000 件のログが生成される見込み
LL_INFO100000 8 // 小規模だが自明でない実行で 100,000 件のログが生成される見込み
LL_INFO10000 7 // 小規模だが自明でない実行で 10,000 件のログが生成される見込み
LL_INFO1000 6 // 小規模だが自明でない実行で 1,000 件のログが生成される見込み
LL_INFO100 5 // 小規模だが自明でない実行で 100 件のログが生成される見込み
LL_INFO10 4 // 小規模だが自明でない実行で 10 件のログが生成される見込み
LL_WARNING 3
LL_ERROR 2
LL_FATALERROR 1
LL_ALWAYS 0 // オフにすることは不可能（ログレベルは負にならない）
```

💡 初心者向け補足

ログレベル (log level) とログファシリティ (log facility) は、ログ出力を制御するための 2 つの軸です。ログレベルはメッセージの重要度（エラー、警告、情報など）によるフィルタリングで、Java の `java.util.logging.Level` や SLF4J のログレベルに相当します。ログファシリティはメッセージのカテゴリ（GC、JIT、ローダーなど）によるフィルタリングで、ランタイムのどのサブシステムからのメッセージを見たいかを選択できます。両方を組み合わせることで、必要な情報だけを効率的に絞り込めます。

`DOTNET_LogFacility` と `DOTNET_LogFacility2` は、特定のカテゴリにメッセージをフィルタリングできます。これらの変数は名前付き定数ではなく、整数で指定します。たとえば、`LF_GC` ファシリティの値は `0x00000001`、つまり `1` です。この文書の執筆時点での `DOTNET_LogFacility` に利用可能なオプションは以下のとおりで、`loglf.h` から取得されています:

```
LF_GC          0x00000001
LF_GCINFO      0x00000002
LF_STUBS       0x00000004
LF_JIT          0x00000008
LF_LOADER      0x00000010
LF_METADATA    0x00000020
LF_SYNC         0x00000040
LF_EEMEM        0x00000080
LF_GCALLOC     0x00000100
LF_CORDB        0x00000200
LF_CLASSLOADER 0x00000400
LF_CORPROF      0x00000800
LF_DIAGNOSTICS_PORT 0x00001000
LF_DBGALLOC    0x00002000
LF_EH           0x00004000
LF_ENC          0x00008000
LF_ASSERT       0x00010000
LF_VERIFIER     0x00020000
LF_THREADPOOL   0x00040000
LF_GCRoots     0x00080000
LF_INTEROP      0x00100000
LF_MARSHALER   0x00200000
LF_TIEREDCOMPILATION 0x00400000 // 以前は IJW でしたが、現在は階層型コンパイル (tiered compilation) 用に転用されています
LF_ZAP          0x00800000
LF_STARTUP      0x01000000 // 起動とシャットダウンの失敗をログに記録
LF_APPDOMAIN   0x02000000
LF_CODESHARING  0x04000000
```

LF_STORE	0x08000000
LF_SECURITY	0x10000000
LF_LOCKS	0x20000000
LF_BCL	0x40000000

`DOTNET_LogFacility2` はより新しく、現時点では `LF2_MULTICOREJIT` (値は `0x00000001`) というファシリティが 1 つだけ利用可能です。

ログ内の特定のメッセージをキャプチャしようとしているのに表示されない場合は、そのログレベル/ファシリティを確認し、環境変数を適切に設定しているか確認してください！

Mono ロギング

Mono ランタイムを使用した .NET のビルドには、ランタイムが output する診断情報を制御する Mono 固有のログ設定環境変数があります。

`MONO_LOG_LEVEL` を使用して、`"error"`、`"critical"`、`"warning"`、`"message"`、`"info"`、`"debug"` のいずれかに設定することで、全体的な冗長度を設定できます。 `MONO_LOG_MASK` を使用して、`gc` や `aot` などの特定のカテゴリにログメッセージをフィルタリングできます。マスクオプションの完全なリストについては、[mono-logger.c](#) の `mono_trace_set_mask_string` 関数を参照してください。

Mono のインタプリター (interpreter) を扱う場合、環境変数 `MONO_VERBOSE_METHOD` は特定の名前を持つメソッドの詳細ログを有効にします。これは、メソッドが正しくコンパイルまたは最適化されていない状況を調査している場合や、どのバージョンのメソッドが実行されているか確信が持てない場合に非常に役立ちます。

WebAssembly ロギング

WebAssembly ビルドのランタイムで Mono ロギングを設定するには、csproj 内の設定項目に JSON プロブの形式で環境変数を指定する必要があります。以下のようにします:

```
<ItemGroup>
  <WasmExtraConfig Include="environmentVariables" Value='
  {
    "MONO_LOG_LEVEL": "warning",
    "MONO_LOG_MASK": "all"
  }' />
</ItemGroup>
```

xml

ビルド時やウェブブラウザーの起動時に外部で設定された環境変数は、自動的に WASM ランタイムに引き継がれません。

WebAssembly ビルドのランタイムには、TypeScript で書かれた追加のインターフェイス (interop layer) があり、独自のロギング機能を持っています。これは [logging.ts](#) で定義されています。

TypeScript レイヤーからのデバッグレベルのログメッセージは、`diagnosticTracing` フラグが設定されていない限り、デフォルトで抑制されます。設定するには、起動時に `dotnet` オブジェクトで `.withDiagnosticTracing(true)` を呼び出すか、`csproj` に以下のような追加の設定項目を記述します：

```
<ItemGroup>
  <WasmExtraConfig Include="diagnosticTracing" Value="true" />
</ItemGroup>
```

xml

その他のすべてのログ重大度はデフォルトで有効であり、ウェブブラウザーで実行している場合は開発者ツールコンソールに書き込まれます。コマンドラインから自動テストを実行している場合や、`node.js` や `v8` シェルなどの環境でアプリケーションを実行している場合は、標準出力および/または標準エラーに書き込まれます。

TypeScript 内からメッセージを送信するには、適切な API を使用してください。重大なエラーには `mono_log_error`、重要な情報には `mono_log_info`、一般ユーザーが見る必要のないものには `mono_log_debug` を使用します。

何らかの理由でこのロギング機能に C/C++ から直接アクセスする必要がある場合は、`mono_wasm_trace_logger` 関数を使用できます。この関数を通じて送信された致命的エラーは、書き込み後に即座にプロセスの終了をトリガーすることに注意してください。

デフォルトでは `jiterpreter` はエラーメッセージのみをコンソールに出力しますが、`MONO_VERBOSE_METHOD` が使用されている場合は、`verbose` メソッド内のトレースに関する詳細情報もログに記録します。`jiterpreter` でのより高度なロギングには、`jiterpreter.ts` の設定変数を編集し、ランタイムをソースからコンパイルする必要があります。