THIS IS CLR

GC

GCの動作は単純明快！

- ルート（オブジェクトを参照している変数）の存在しないオブジェクトを回収し、メモリを解放する。
- それだけを知っていればよい。

GCは全く透過的だ！

- GCの動作の詳細を知らなくても殆どにおいて問題ない。GCはプログラム、或いはプログラマからは全く透過的だ。

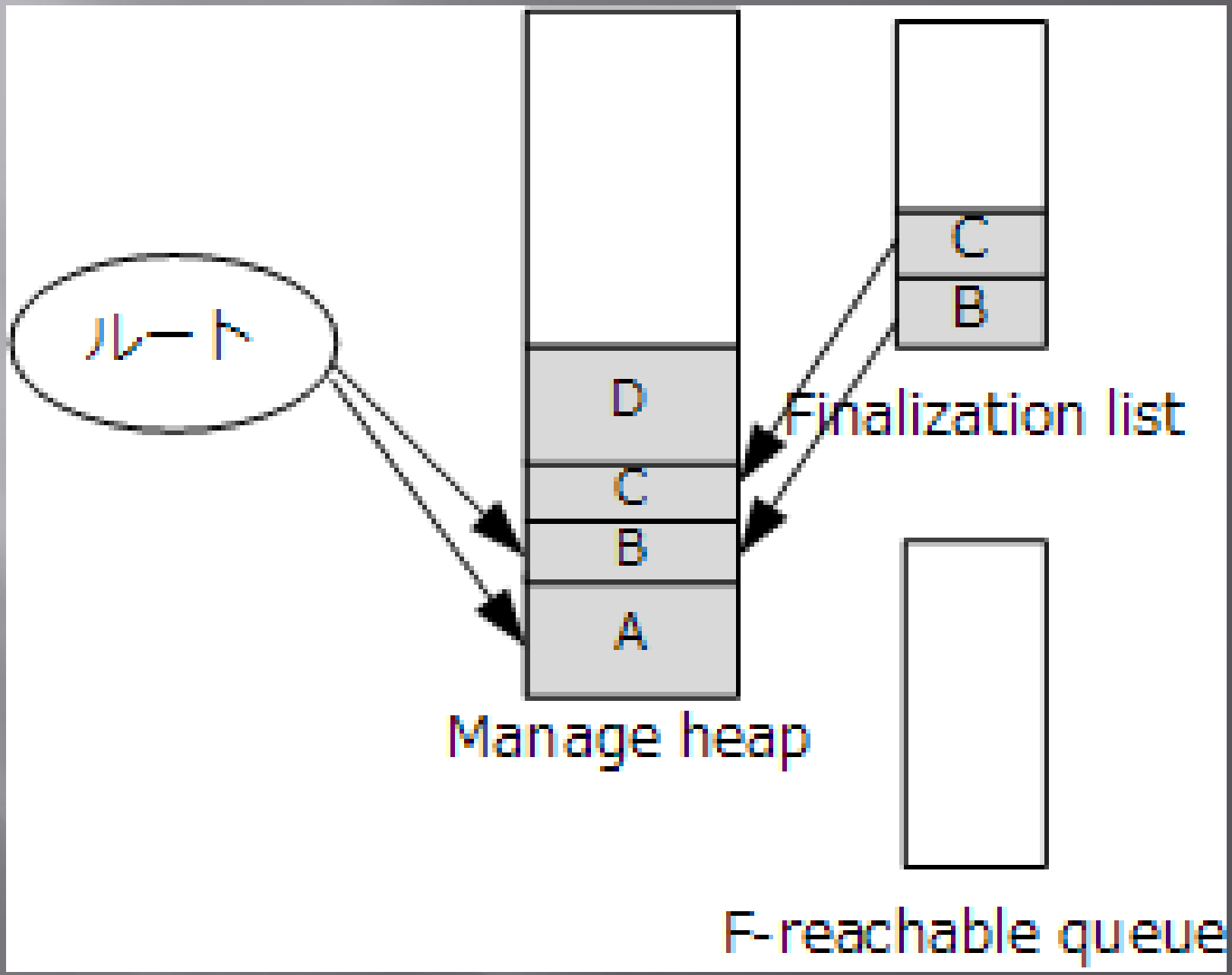
だが、知識は役に立つ

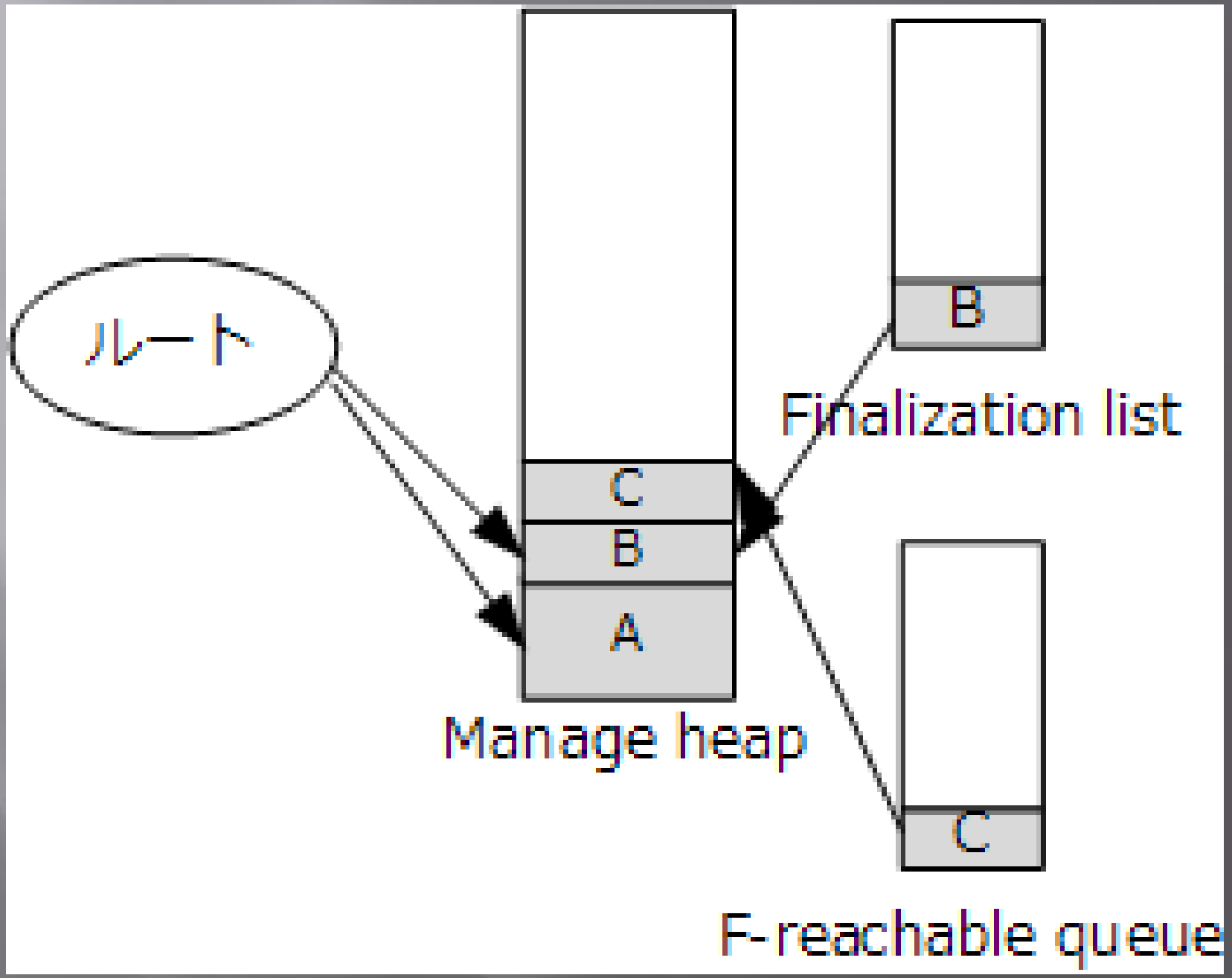
- パフォーマンス
- アンマネージリソースの確実な解放
- アンマネージコードとの連携
- ルートがないように見えるオブジェクトの維持
- etc…

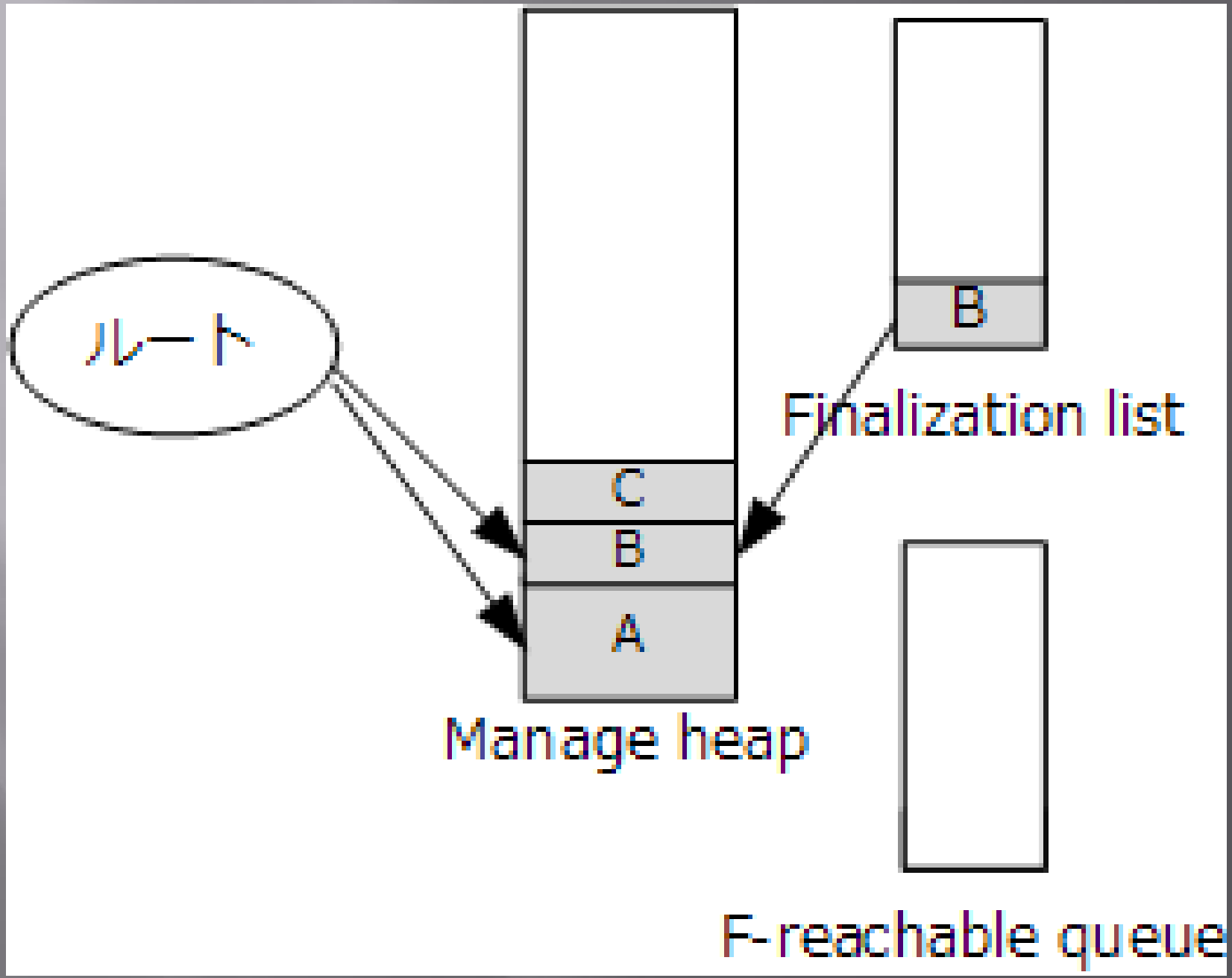
- 知識はトラブルシュートの役立つ

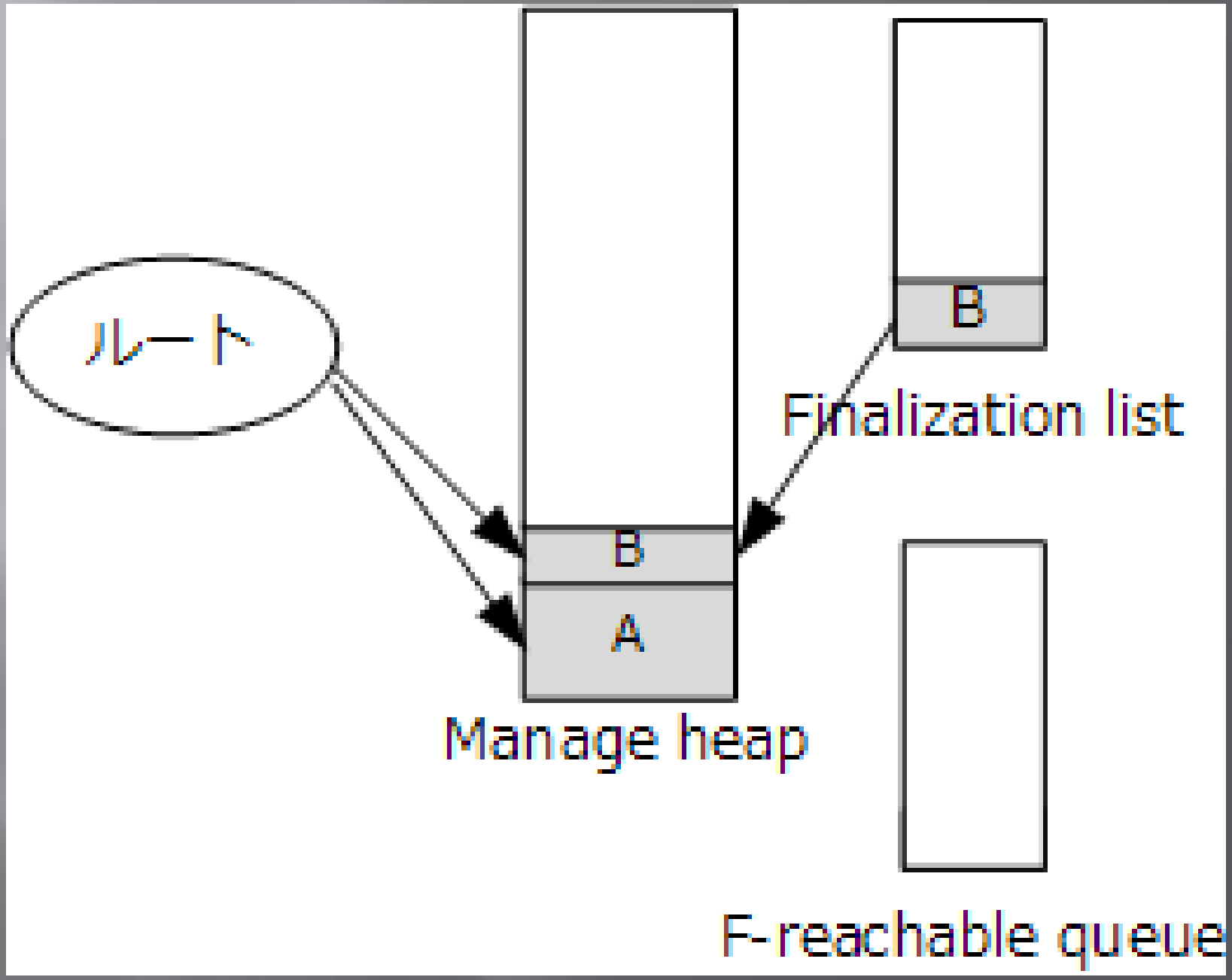
パフォーマンス

- Finalize を実装するかしないかで、そのクラスのオブジェクトのGCパフォーマンスは大きく変わる。
- Finalize を実装しているクラスのオブジェクトは一度のGCでは決して回収されない。


ルート
D
C
B
A
Manage heap
C
B
Finalization list
F-reachable queue


ルート
C
B
A
Manage heap
B
Finalization list
C
F-reachable queue


ルート
C
B
A
Manage heap
B
Finalization list
F-reachable queue


ルート
B
A
Manage heap
B
Finalization list
F-reachable queue

パフォーマンスが悪い！

- Finalize実装オブジェクトは最低でも3回以上（ジェネレーション昇格も発生）のGC実行後にしか回収されない。

アンマネージリソースの確実な解放

- アンマネージリソースを持つクラスは、アンマネージリソースを確実に解放するためにFinalizeを実装する。
- 加えて、任意のタイミングで解放できるようにDisposeパターンも使用する。

Finalizeを実装したら確実か？

- Finalizeを実装したからといって確実に呼ばれるとは限らない。
- Finalize呼び出し時に、FinalizeメソッドをJit compileできないぐらいメモリが逼迫していたらどうなる？

CriticalFinalizerObject

- インスタンス化と同時にFinalizeメソッドを JIT COMPILE してしまう。
- 通常のFinalize実装クラスの後にCriticalFinalizerObjectのFinalizeを呼ぶ。

アンマネージコードとの連携

- GCの仕事は、使われなくなったオブジェクトを解放するだけではないという事も覚えておこう。
- 穴空きになったMange heapをコンパクションし、使用中のメモリを一ヶ所に集める。

オブジェクトは動く

- マネージコード内のオブジェクトのアドレスは不定
- GCを備えていない実装系ではオブジェクトのアドレスが動くという想定はしていない。

オブジェクトを固定する

- マネージコートとアンマネージコードを連携するにはアドレスを固定、或いは代替アドレスを使用する必要がある。

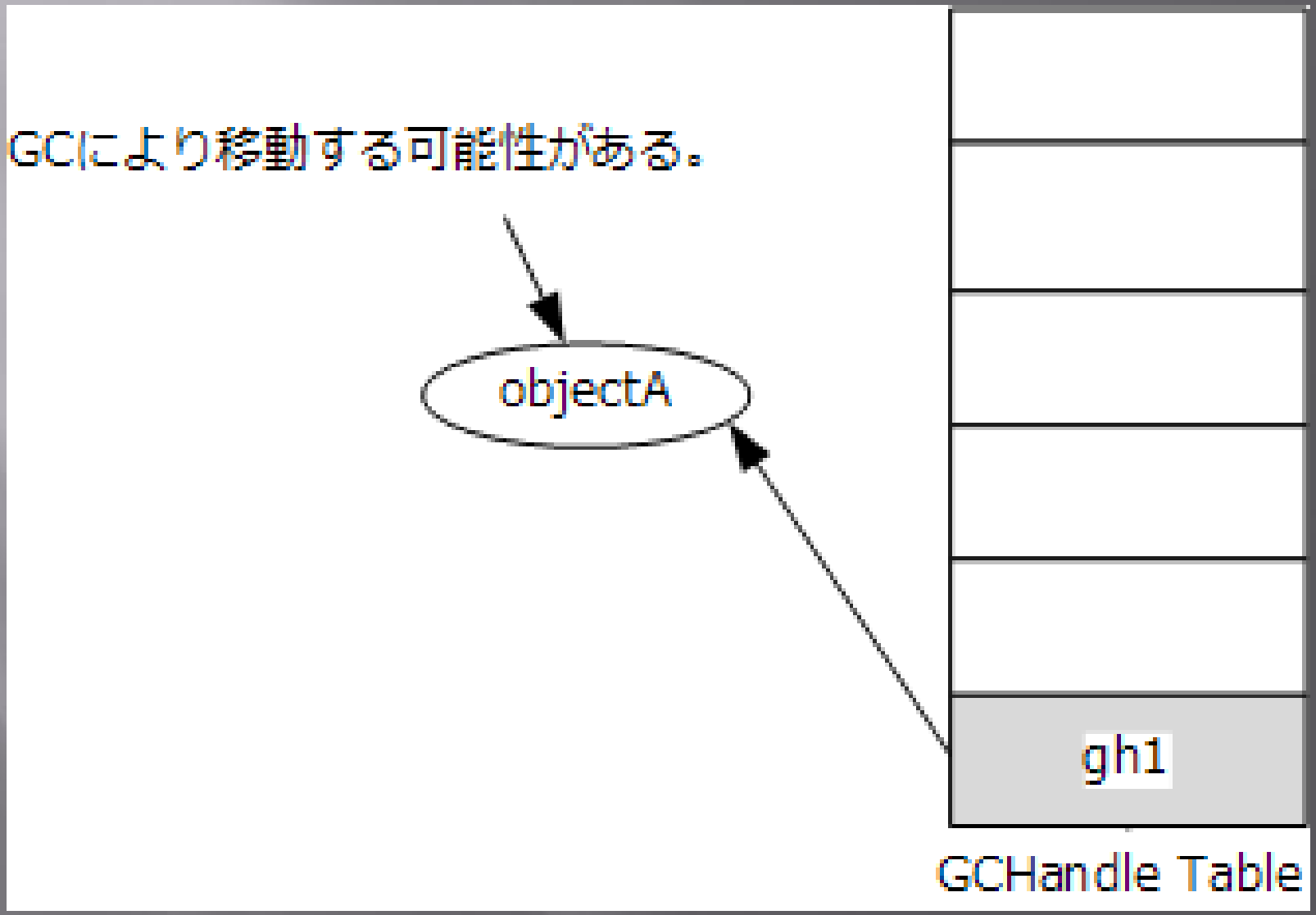
GCHandle

- AppDomain毎に一つ存在するGCHandle Tableを利用し、固定されたアドレスを得る。

GCHandleType.Normal

- // オブジェクトをGCHandle Tableに登録する。
- object objectA = new object();
- GCHandle gh1 = GCHandle.Alloc(o, GCHandleType.Normal);
- IntPtr p = (IntPtr)gh1
- // p をアンマネージコードに渡す。
- // p のアドレスは決して動かない。
- ・・・
- // アンマネージコードからpがコールバックされる。
- GCHandle g = (GCHandle)p;
- object o2 = g.Target;

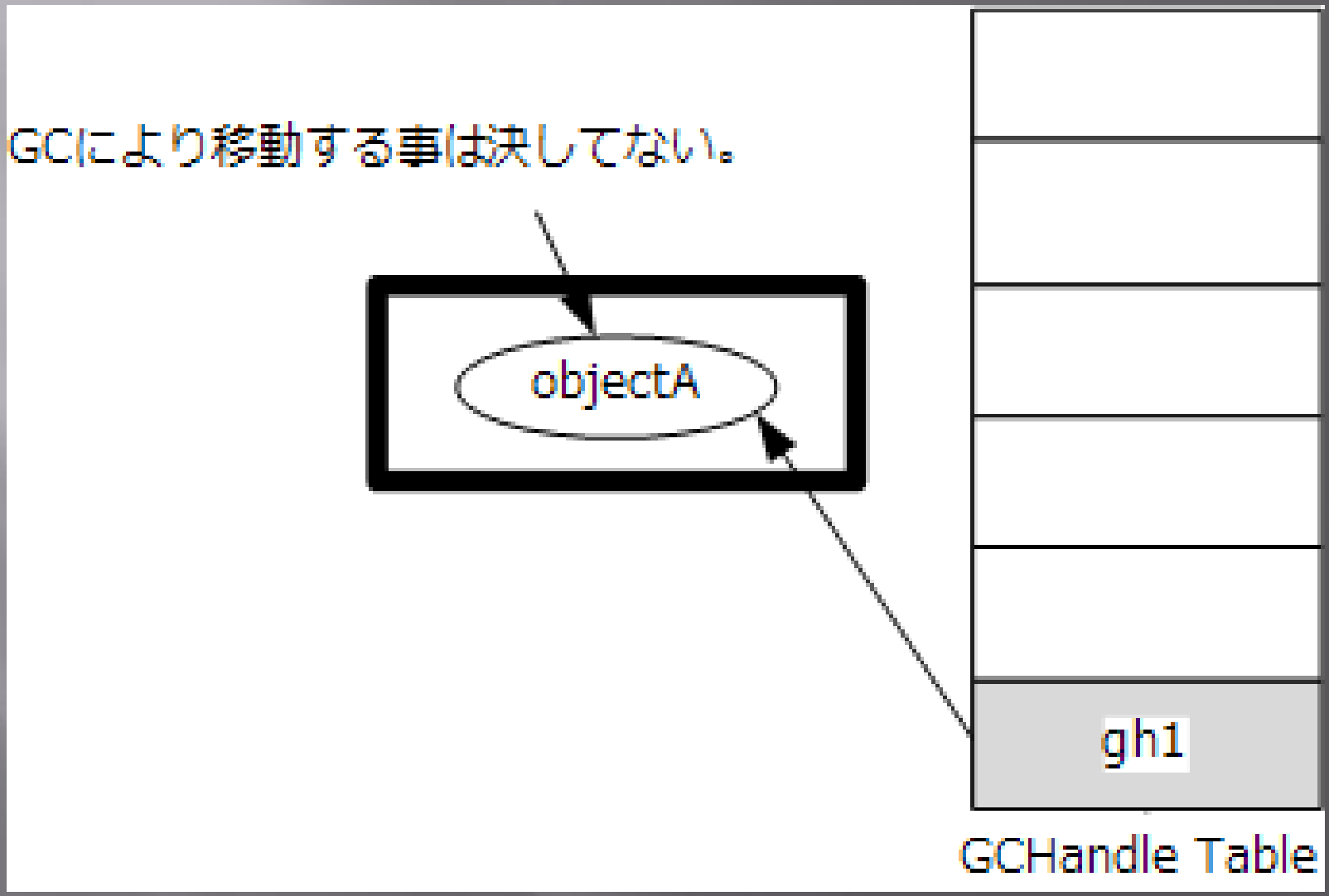
GCHandleType.Normal



GCHandleType.Pinned

- // オブジェクトをGCHandle Tableに登録する。
- object objectA = new object();
- GCHandle gh1 = GCHandle.Alloc(o, GCHandleType.Pinned);
- // objectAのアドレスををアンマネージコードに渡す。
- // objectA のアドレスは決して動かない。
- ・・・
- // アンマネージコードからobjectAを直接触る。

GCHandleType.Pinned



何故Formが消えない？

- void Func() {
- 　Form1 f = new Form1();
- 　f.Show();
- }

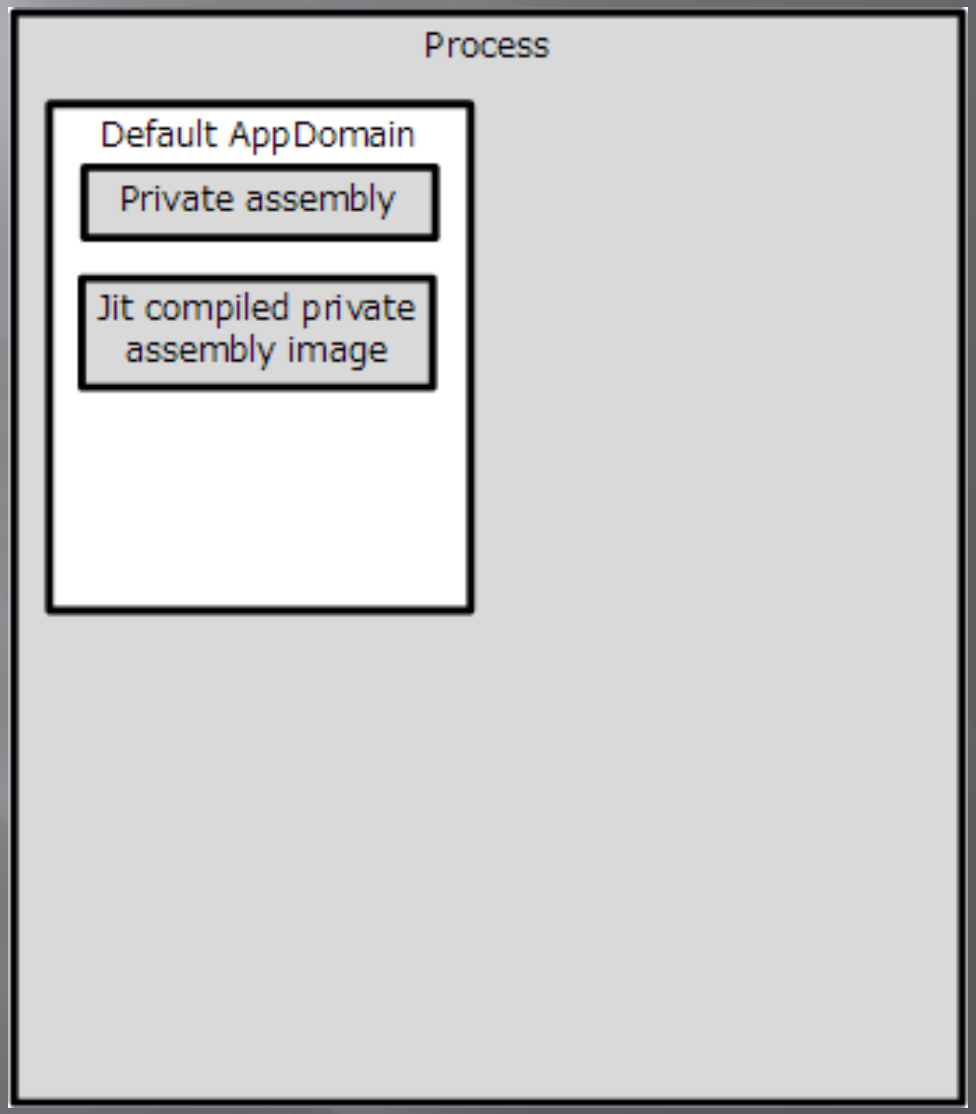
- GCHandleのインスタンスを内部に持ち、自身をTargetとしている。

AppDomain と assembly

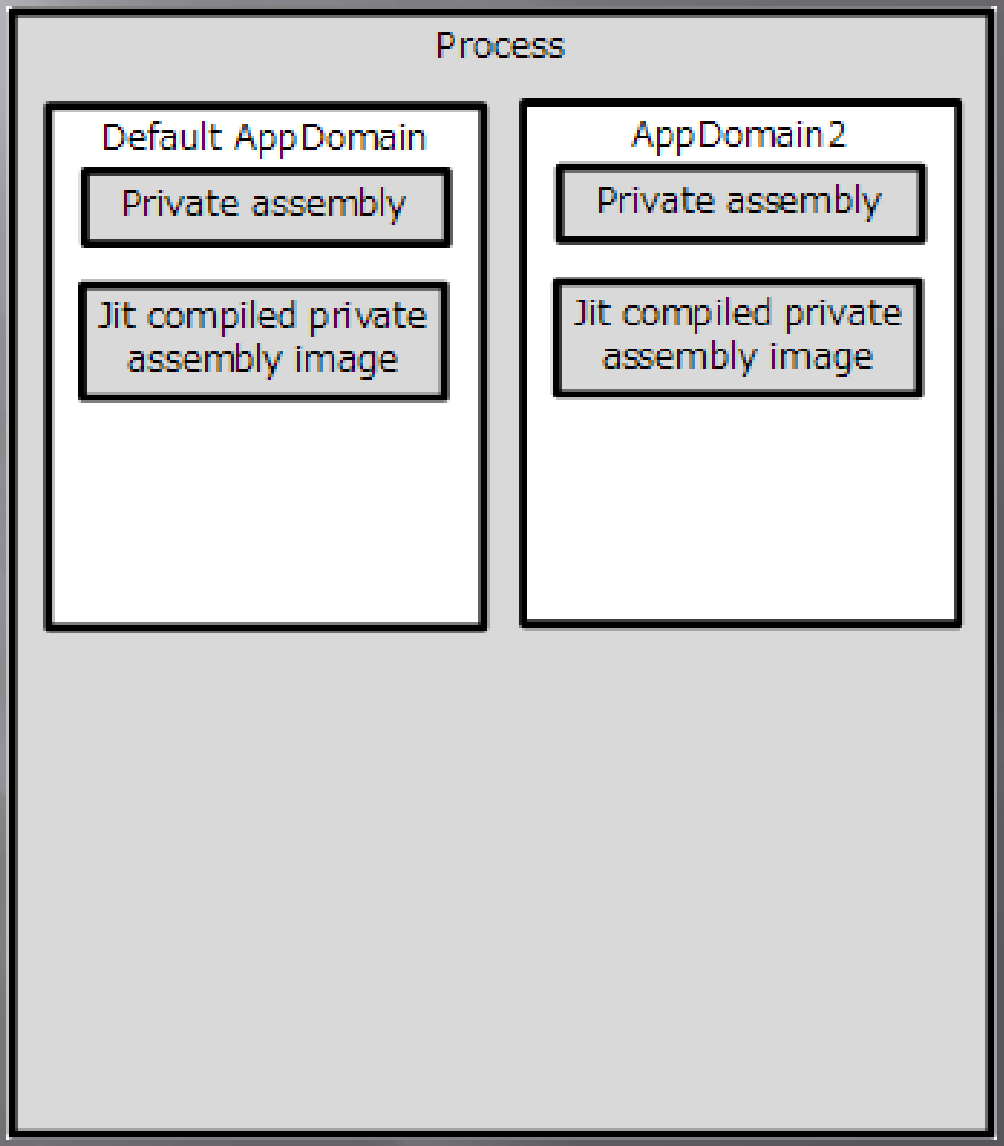
殆どのプロセスは シングルAppDomain

- AppDomainなど知らなくても問題ない。殆どのアプリケーションはたった一つのAppDomainで動作する（ように見える）。

Single AppDomain



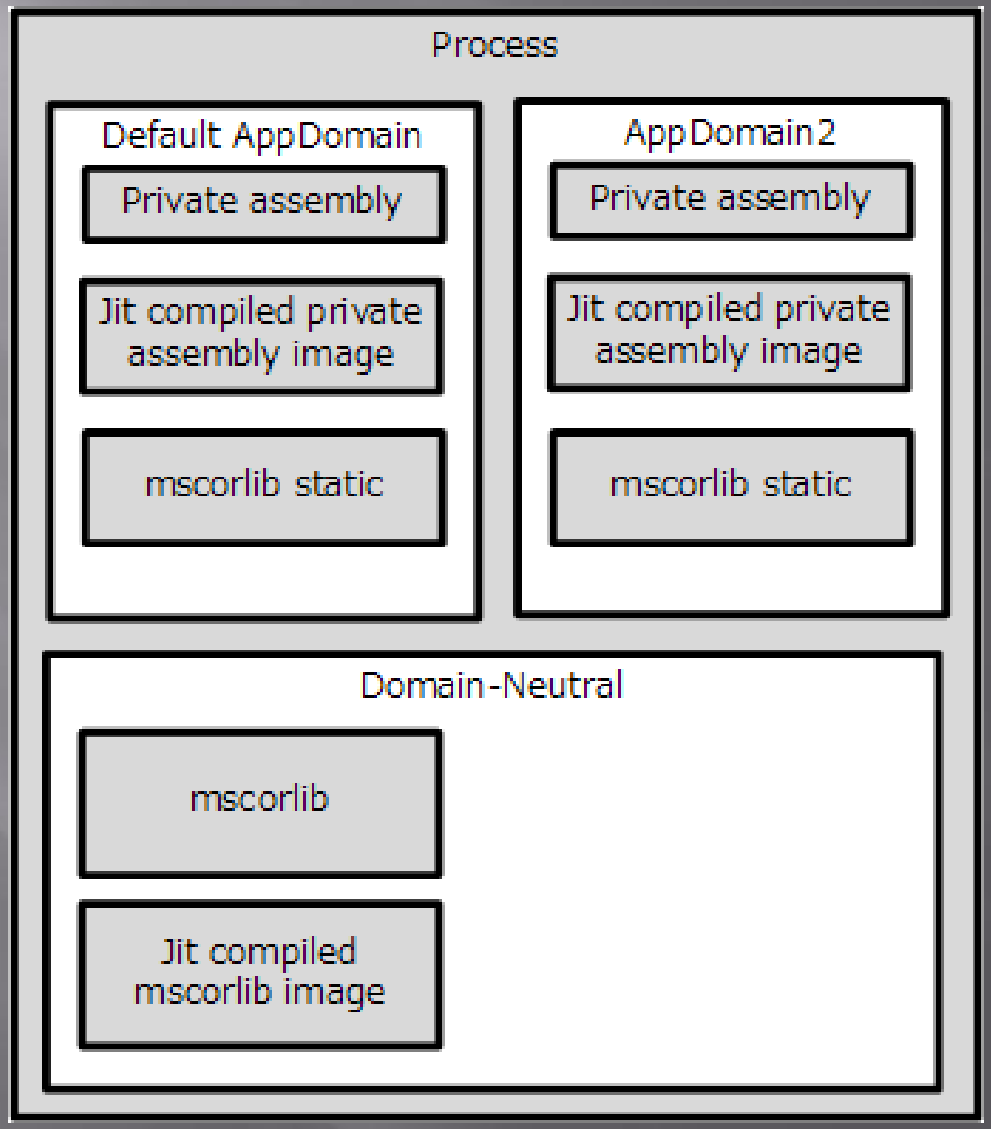
Multi AppDomain



Multi AppDomain

- assemblyはAppDomain単位で各々にロードされる。
- Jit コンパイルされたネイティブイメージも各々のAppDomainが持つ。

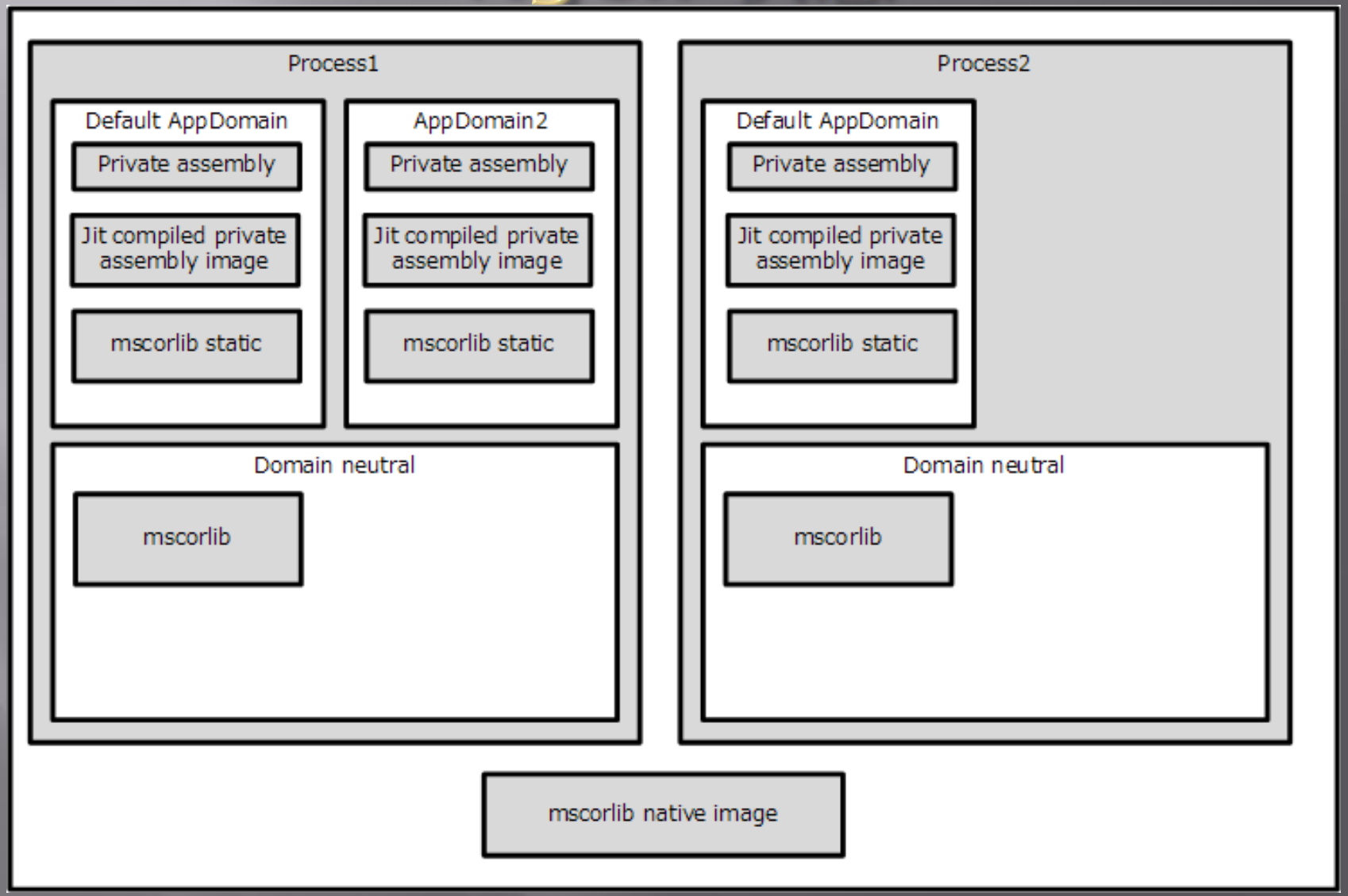
Domain Neutral



Domain Neutral

- ドメイン中立にあるassemblyはプロセス内の全てのassemblyで共有する。
- Jit コンパイルされたネイティブイメージも共有できる。
- static データは各々のassemblyでコピーが必要。
- しかし、各プロセスでそれぞれJit コンパイルする必要がある。

Ngen 状態

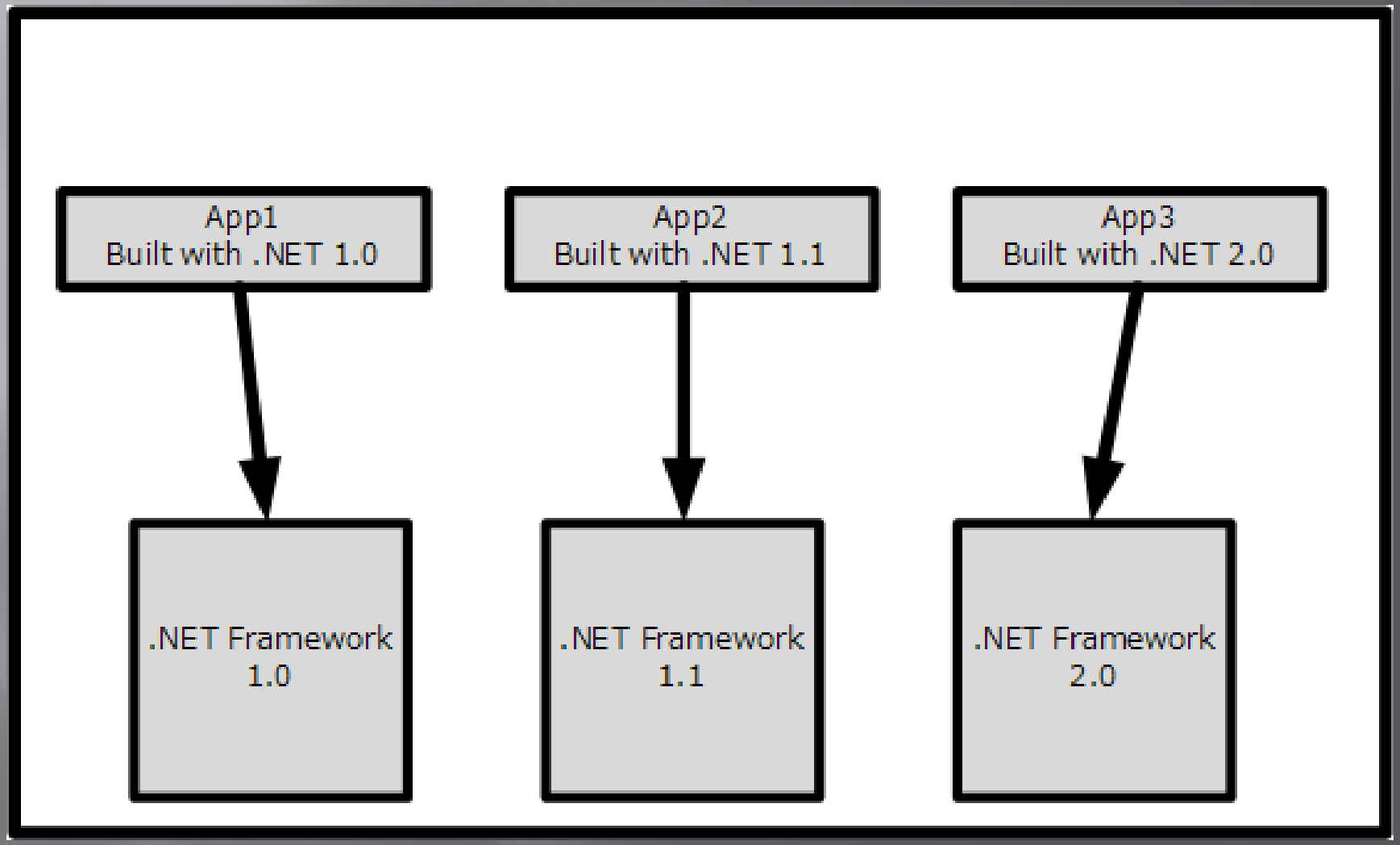

Process1
Default AppDomain
Private assembly
Jit compiled private assembly image
mscorlib static
AppDomain2
Private assembly
Jit compiled private assembly image
mscorlib static
Domain neutral
mscorlib
Process2
Default AppDomain
Private assembly
Jit compiled private assembly image
mscorlib static
Domain neutral
mscorlib
mscorlib native image

Ngen 状態

- ngen.exeで予めネイティブイメージを作っておくと全てのプロセスでコードページを共有できる。

side-by-side

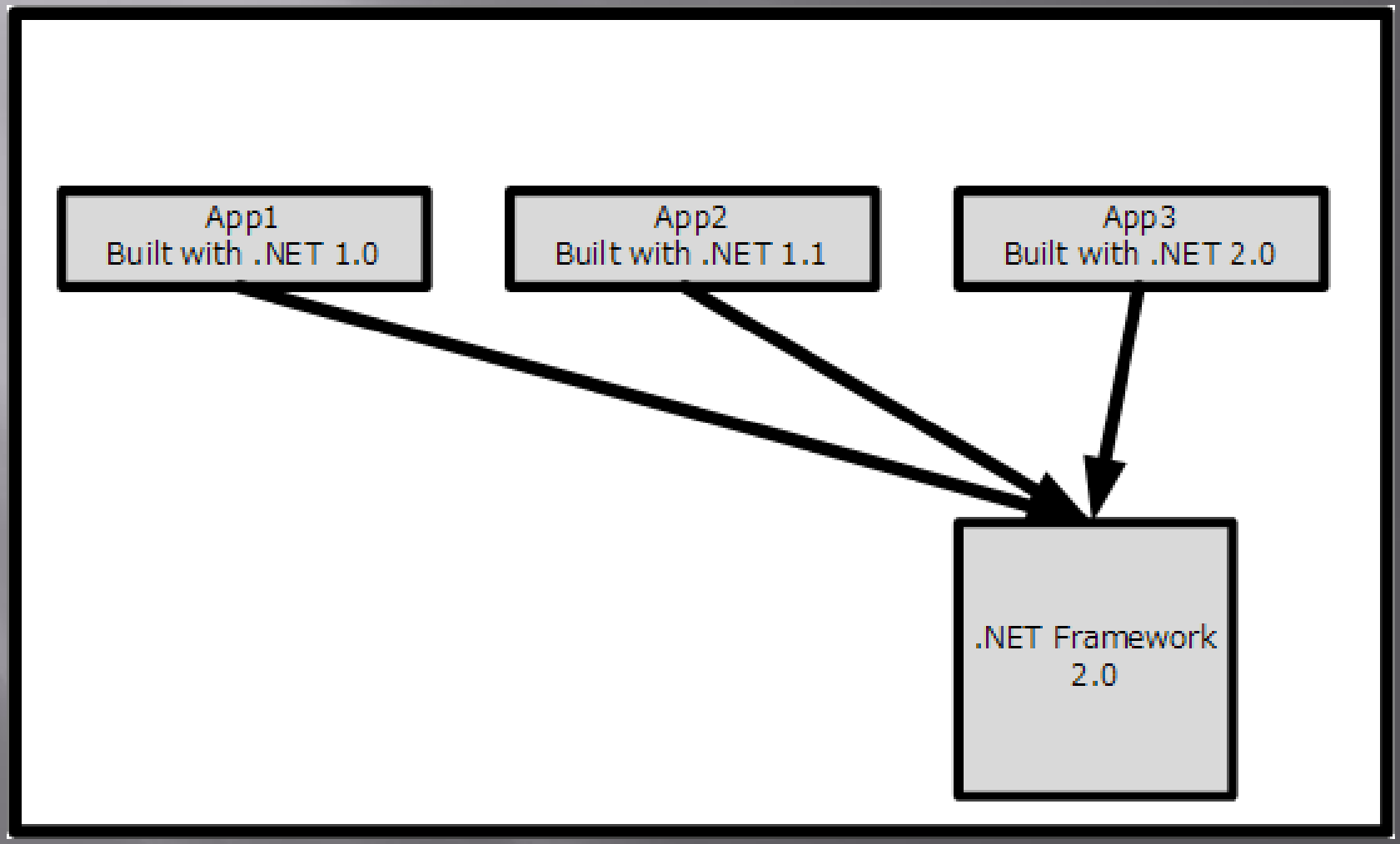
side-by-side


App1
Built with .NET 1.0
App2
Built with .NET 1.1
App3
Built with .NET 2.0
.NET Framework
1.0
.NET Framework
1.1
.NET Framework
2.0

side-by-side

- .NET アプリケーションは「コンパイルタイム」の環境がそのまま「ランタイム」の環境となる。

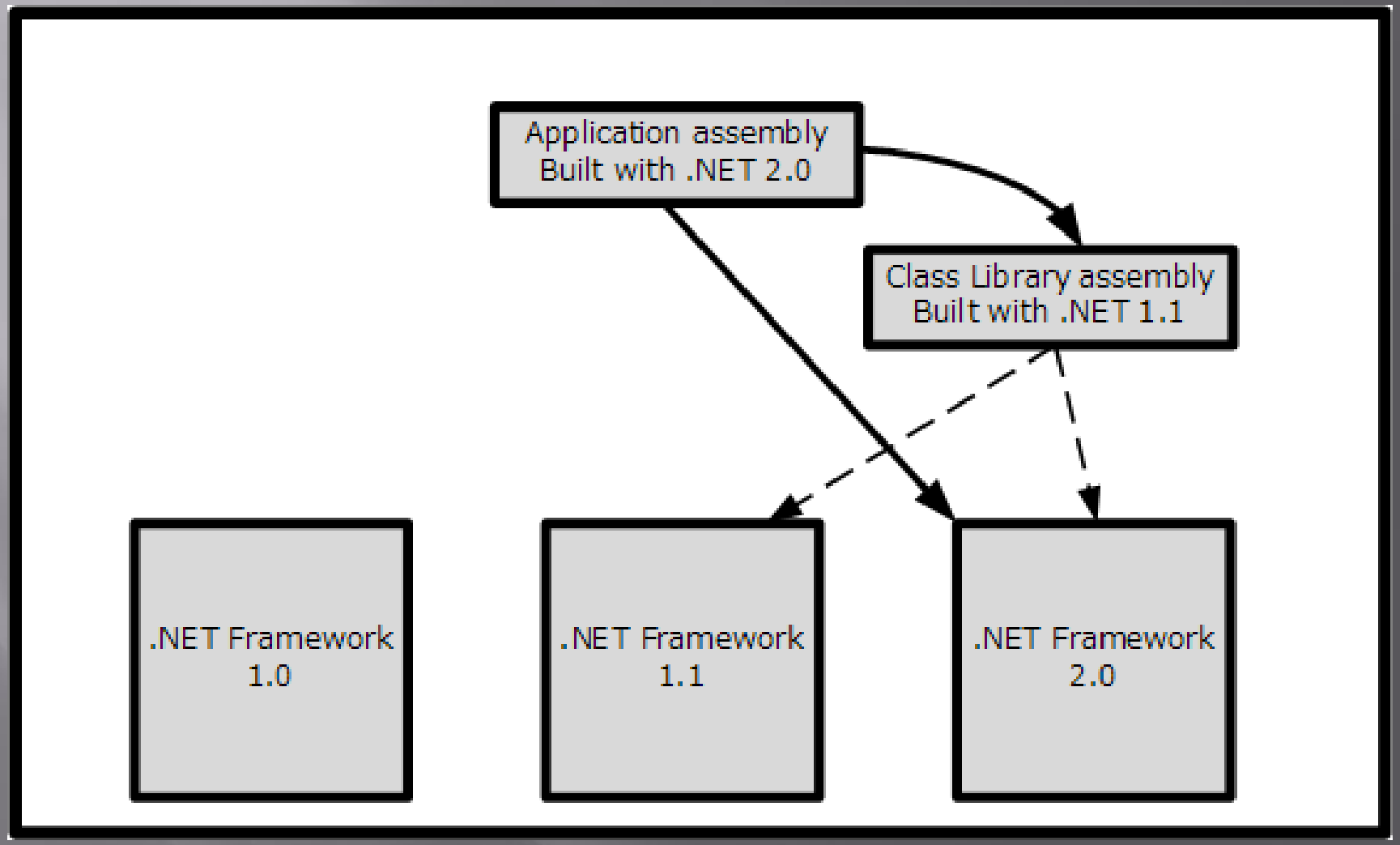
バージョンリダイレクト



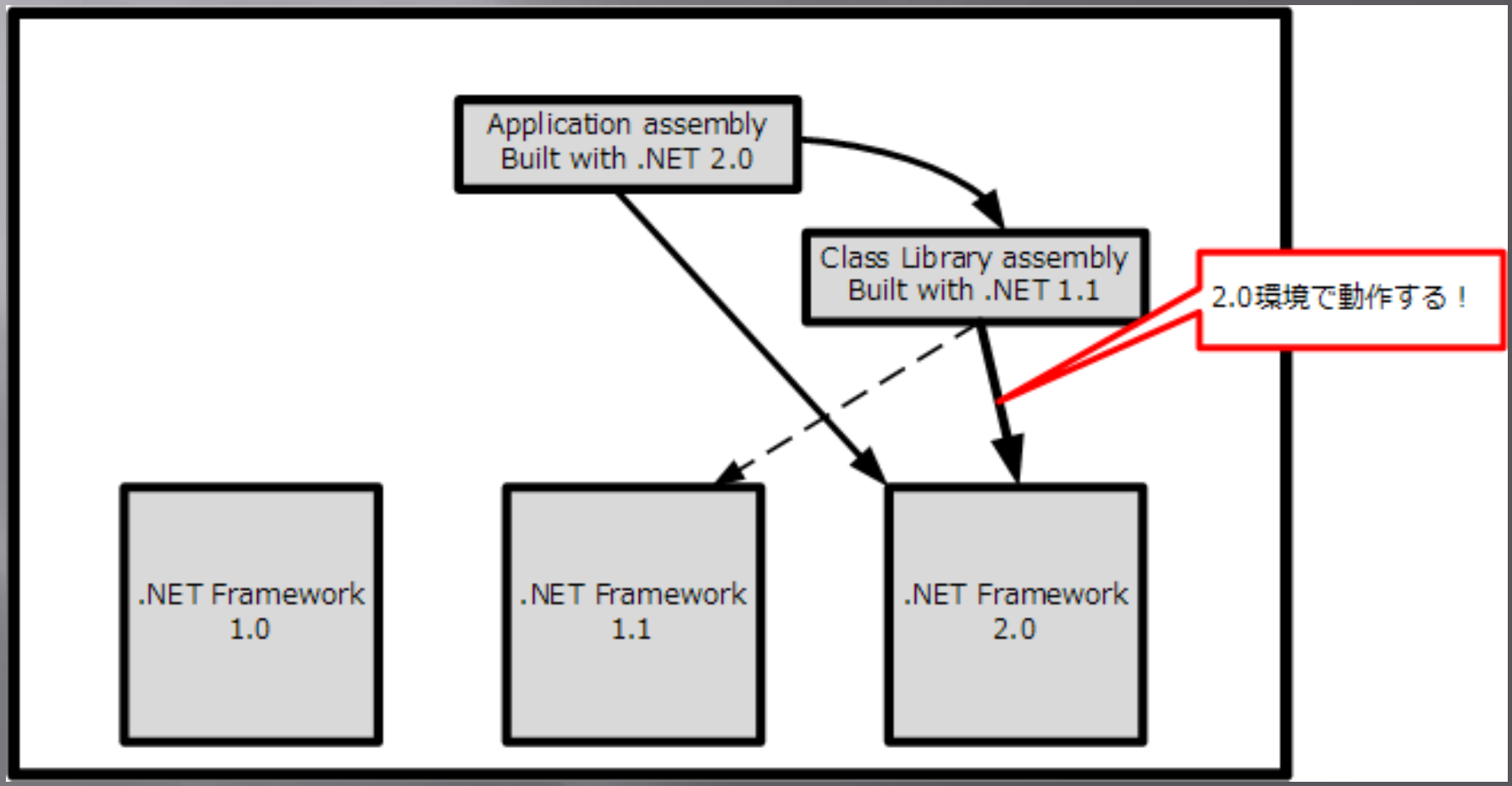
バージョンリダイレクト

- side-by-sideで動作させるのは、最終手段である。基本はバージョンアップ後のライブラリで動作するのが望ましい。

.NET Framework Unification



.NET Framework Unification



.NET Framework Unification

- 一つのプロセスには一つのCLRしかロードできない。
- CLRのバージョンとmscorlibのバージョンは必ず同一でなければならない。
- CLRのバージョンとmscorlib以外の.NET Framework Class Libarry のバージョンは必ずしも同一でなくてもよい。

.NET Framework Unification

DEMO

ご静聴
ありがとう
ございました。