公共语言运行时

原文：*Boot of the Runtime*
翻译: @dontpanic
版本: 20230821 (b9790bb)
在线阅读：*https://tuesday.dontpanic.blog*

1 CLR 简介

这是一篇译文。作者：Vance Morrison - 2007
原文链接 https://github.com/dotnet/coreclr/blob/master/Documentation/botr/intro-to-clr.md

1.1 什么是 CLR？

什么是公共语言运行时（Common Language Runtime, CLR）？ 简单来说就是：

> 公共语言运行时（CLR）是一套完整的、高级的虚拟机，它被设计为用来支持不同的编程语言，并支持它们之间的互操作。

啊，有点绕口，同时也不太直观。不过这样的表述还是 **有用的**，它把 CLR 的特性用一种易于理解的方式分了类。由于 CLR 实在太过庞大和复杂，这是我们理解它的第一步——犹如从万米高空俯视它，我们可以了解到 CLR 的整体目标；而在这之后，我们就可以带着这种全局观念，更好地详细了解各个子模块。

1.2 CLR：一个（很少见的）完备的编程平台

每一个程序都有大量的运行时依赖。当然，一个程序需要由某种特定的编程语言编写而成，不过这只是程序员把想法变成现实的第一步。所有有意义的程序，都免不了需要与一些**运行⌧⌧**打交道，以便能够操作机器的其他资源（比如用户输入、磁盘文件、网络通讯，等等）。程序代码还需要某种变换（翻译或编译）才能够被硬件直接执行。这些依赖实在是太多了，不仅种类繁多还互相纠缠，因此编程语言的实现者通常都把这些问题交由其他标准来指定。例如，C++ 语言并没有制定一种 “C++可执行程序” 格式；相反，每个 C++ 编译器都会与特定的硬件架构（例如 x86）以及特定的操作系统（例如 Windows、Linux 或 macOS）绑定，它们会对可执行文件的格式进行描述，并规定要如何加载这些程序。因此，程序员们并不会搞出一个 “C++ 可执行文件”，而是 “Windows X86 可执行程序” 或 “Power PC Mac OS 可执行程序”。

通常来说，直接使用现有的硬件和操作系统标准是件好事，但它同样也会把语言规范与现有标准的抽象层次紧密捆绑起来。例如，常见的操作系统并没有支持垃圾回收的堆内存，因此我们就无法用现有的标准来描述一种能够利用垃圾回收优势的接口（例如，把一堆字符串传来传去而不用担心谁来删除它们）。同样，典型的可执行文件格式只提供了运行一个程序所需要的信息，但并没有提供足够的信息能让编译器把其他的二进制文件与这个可执行文件绑定。举例来说，C++ 程序通常都会使用标准库（在 Windows 上叫做 msvcrt.dll），它包含了大多数常用的功能（例如 `printf`），但只有这一个库文件是不行的。程序员如果想使用这个库，必须还要有与它相匹配的头文件（例如 `stdio.h`）才可以。由此可见，现有的可执行文件格式标准无法同时做到：1、满足运行程序的需求；2、提供使程序完整所必须的其他信息或二进制文件。

CLR 能够解决这些问题，因为它制定了一套非常完整的规范（已被 ECMA 标准化）。这套规范描述了一个程序的**完整**生命周期中所需要的所有细节，从构建、绑定一直到部署和执行。例如，CLR 制订了：

- 一个支持 GC 的虚拟机, 它拥有自己的指令集（叫做公共中间语言, Common Intermediate Langauge），用来描述程序所能执行的基本操作。这意味着 CLR 并不依赖于某种特定类型的 CPU。

- 一种丰富的元数据表示，用来描述一个程序的声明（例如类型、字段、方法等等）。因此编译器能够利用这些信息来生成其他程序，它们能够从“外面”调用这段程序提供的功能。
- 一种文件格式，它指定了文件中各个字节所表达的意含义。因此你可以说，一个“CLR EXE”并没有与某个特定的操作系统或计算机硬件相捆绑。
- 已加载程序的生命周期语义，即一种“CLR EXE 引用其他 CLR EXE”的机制。同时还制订了一些规则，指定了运行时要如何在执行阶段查找并引用其他文件。
- 一套类库，它们能够利用 CLR 所支持的功能（例如垃圾回收、异常以及泛型）来向程序提供一些基本功能（例如整型、字符串、数组、列表和字典），同时也提供了一些与操作系统有关的功能（例如文件、网络、用户交互）。

1.2.1 多语言支持

定义、规范以及实现所有这些细节是一项艰巨的任务，这也是为什么像 CLR 一样完备的抽象非常少见。事实上，大部分这些基本完备的抽象都是为某一个语言而生的。例如，Java 运行时、Perl 翻译器、或是早期的 Visual Basic 运行时，都提供了类似的完整的抽象界限。使得 CLR 在这些工作中脱颖而出的是它的多语言支持特性。很多语言，当单独使用时，体验很好；但当与其他语言交互时却非常麻烦（Visual Basic 大概是个例外，因为它使用了 COM 对象模型）。语言之间的交互难点在于，它们只能使用操作系统所提供的基本功能与其他语言进行交互。由于操作系统的抽象层次太低（例如操作系统并不知道支持垃圾回收的堆内存是什么），因此跨语言交互通常都很复杂。通过提供**公共语言运行时**，CLR 允许语言之间使用更高层次的结构进行交互（例如支持 GC 的结构），极大简化了交互的复杂性。

由于运行时在 **很多**语言之间共享，它意味着我们我们可以投入更多的资源在运行时上。为一个语言构建一个优秀的调试器和性能分析器通常需要大量的工作，因此通常来说只有那些最重要的编程语言才拥有完备的调试器和性能分析器。然而，由于在 CLR 上的语言可以复用这些基础设施，为某种语言实现调试器的负担就会减轻很多。更重要的时，任何建立在 CLR 之上的语言，都可以立刻获得访问 **所有**类库的能力。这些庞大（并且还在不断完善）的类库是 CLR 成功的另一个重要原因。

简而言之，运行时就是一套完整的规范，它规定了创建和运行一个程序所需要的方方面面。而负责运行这些程序的虚拟机，非常适合用来实现各种各样的编程语言。这个虚拟机、以及跑在这个虚拟机上的（不断完善的）类库，就是我们所说的公共语言运行时（CLR）。

1.3 CLR 的主要目标

现在我们对 CLR 是什么有了一个基本的认识，下面我们就来看看运行时究竟想要解决什么问题。从非常高的角度来说，运行时只有一个目标：

CLR 的目标是让编程变得简单。

这条表述可以从两方面来理解：

一方面，在运行时不断进化的过程中，这是一条 **非常**有用的指导准则。例如，从根本上来说，简洁的东西才会简单。如果某个改动会向运行时中添加**用户可见**的复杂性，我们就需要秉持怀疑的态度来审视。相比于计算某个功能的“成本收益比”，我们更看重“添加的用户可见复杂度”与“在所有场景上的加权收益”之比。理想情况下，这个比值应该是负的——即新功能通过减少限制或泛化特例，从而使的复杂性降低。在现实情况下，我们应当尽量最小化暴露给外部的复杂度，并最大化这个功能所适用的场景。

另一方面，这个目标的重要性在于：**易用性是 CLR 成功的基石**。CLR 并不是因为比原生代码更快更小而成功的（事实上，写的好的原生代码通常在这些方面都会胜出）；CLR 也并不是因为提供了某种特别的功能而成功的（例如垃圾回收、平台无关、面向对象编程或版本管理）。CLR 的成功在于：这些功能、以及其他不计其数的功能加在一起，使得编程变得简单得多。很多很重要但是经常被忽视的易用功能包括：

1. 简化的语言（例如，C# 和 Virual Basic 要比 C++ 简单太多）
2. 致力于简化类库（例如，我们只有一种字符串类型，它是不可变的；这极大地简化了适用字符串的 API）
3. 类库中名称之间很强的一致性（例如，要求 API 使用完整的单词，并使用一致的命名规范）
4. 对创建一个程序所需要的工具链提供了大力支持（例如，Visual Studio 使得构建 CLR 应用程序非常简单，Intellisense 使得查找正确的类型和方法变得非常容易）

正是这些在易用性上的努力（它们与用户模型的简单性密切相关），才是 CLR 能够成功的原因。奇怪的是，一些在易用性方面最重要的特性通常都是最“无聊”的。比如，其实任何编程环境都可以提供一致的命名规范，但在如此庞大的类库上保持一致性还是需要很多工作的（译注：黑人问号 PHP）。这样的做法通常会与其他目标冲突（例如与现有接口保持兼容性），或者与做起来比较复杂（例如在一个**非常**大的代码库中重命名一个方法）。正因如此，我们才需要时刻提醒自己，什么才是 CLR 最重要的目标。这样才能够更快地向目标迈进。

1.4 CLR 的主要功能

运行时有很多功能，我们可以概括为以下几类：

1. **基础功能**——那些对其他的特性有广泛影响的功能。包括：
    1. 垃圾回收
    2. 内存安全和类型安全
    3. 对编程语言的高级支持
2. **次要功能**——那些由基础功能发展而来的、但不是必须的功能：
    1. AppDomains 程序隔离
    2. 程序安全与沙盒
3. **其他功能**——那些运行时环境需要的、但并不依赖基础功能的特性。这些功能帮助我们建立了一个完整的编程环境。比如：
    1. 版本管理
    2. 调试、性能分析
    3. 互操作

1.4.1 CLR 垃圾回收器

在 CLR 所提供的所有功能中，垃圾回收器值得特别关注。垃圾回收（GC）的意思是“内存自动回收”。在一个支持垃圾回收的系统中，用户程序不再需要调用一个特殊的操作符来删除内存。相反，运行时会自动跟踪在 GC 堆内存上的所有内存引用，并且他会不时地遍历这些引用，判断这些内存是不是还会被程序所使用。所有不再被使用的内存就是**垃圾**，它们可以被用于新的内存申请。

垃圾回收是一个非常有用的功能，因为它简化了编程工作。最明显的简化就是，大多数显式的删除操作都可以省略了。当然，省略删除操作这一点很重要，但垃圾回收给程序员带来的

真正价值要更微妙一点：

1. 垃圾回收简化了接口设计。没有垃圾回收的话你就需要考虑，究竟接口的哪一侧需要负责删除在接口上传入传出的对象。例如，CLR 的接口就可以很简单地返回一个字符串，我们不需要担心字符串的缓冲区和长度。这也意味着我们也不需要担心“缓冲区是不是足够大”。因此，垃圾允许运行时中所有的接口都要比以前更简洁一些。
2. 垃圾回收消除了一些常见的用户错误。对于某一个特定的对象来说，我们非常容易搞错它的生命周期，要么是删除的太早（将会导致内存内容失效），或者删除的太晚（内存泄漏）。一个典型的程序会使用成千上万个对象，出现错误的概率确实很大。更进一步，生命周期这一类的 bug 很难调试，尤其是这个对象被很多其他对象所引用的时候。垃圾回收使得这类错误不会再发生，给我们带来了很大的便利。

垃圾回收非常有用，这一点我们就说到这了。我们还有更重要的事情需要讨论，那就是垃圾回收给运行时带来的这个最直接的需求：

> 垃圾回收要求运行时跟踪 GC 堆内存上**所有**的引用。

这个要求看起来非常简单，然而事实上它给运行时带来了深远的影响。就像你所想到的那样，在程序运行的每时每刻都要知道每一个指针指向了哪个对象，这太难了。不过，我们可以稍微降低一下需求。从技术上说，只有在真正进行垃圾回收的时候，我们才需要上面这个要求得到满足（因此，理论上说我们并不需要时刻知道所有的 GC 引用，只有在进行 GC 时才需要）。然而在实践中，这个小技巧并不能完全搞定这个问题，因为 CLR 还有另一个特性：

> CLR 支持在同一个进程中并发执行多个线程。

在任何时间，某个线程的执行都可能会导致内存的申请，进而可能需要进行一次垃圾回收。并发线程的执行顺序是无法确定的，我们没办法知道当一个线程出发了垃圾回收时另一个线程在干什么。因此，GC 有可能发生在某个线程执行当中的任何时间。CLR 并不需要 **立即**响应某个线程的 GC 请求，因此 CLR 确实还是有一定的“回旋余地”的。不过，CLR 还是需要保证它能在一定的时间内对 GC 请求做出响应。

这说明：CLR 需要 _几乎_随时跟踪 GC 堆上的 **所有** 引用。GC 引用可能会存放在机器寄存器中、在局部变量中、在静态域或其他域中等等，确实有不少地方需要我们关注。比较难办的是存放在机器寄存器及局部变量中的引用，因为它们与用户代码的执行紧密相关。事实上这就意味着，参与操作 GC 引用的 **机器代⌧** 必须能够跟踪 GC 引用——也就是说，编译器需要生成额外的代码来完成这些工作。

想要了解更多内容的话，请参看垃圾回收器设计文档。

1.4.2 什么是“托管代码”

这种能够做到“几乎随时”报告所有仍然生效的 GC 引用的代码，就叫做“托管代码”（因为它由 CLR 进行“托管”）。不满足这样的要求的代码就叫做非托管代码。因此所有在 CLR 启动之前执行的代码都是非托管代码，例如，所有的操作系统代码都是非托管的。

1.4.2.1 栈展开问题 很明显，由于托管代码需要使用操作系统提供的服务，有时托管代码就需要调用非托管代码。类似地，由于托管代码是由操作系统所启动的，因此有时非托管代码还会调用托管代码。因此，普遍来说，如果你在任意时刻暂停了某个托管程序，调用栈中将会混合着由托管代码和非托管代码创建的不同类型的栈帧。

非托管代码所创建的栈帧只要满足程序能够运行就可以了。举例来说，这些栈帧并不需要支持查看谁调用了它们。这就是说，如果我们暂停了一个程序，它恰好正在执行非托管代码，那么并没有一种通用的方法能够知道调用者是谁 [1]。虽然我们能够在调试器中看到调用者，但这是由于有额外的符号信息支持（PDB 文件），而这种信息并不保证一定存在（这就是为什么在调试器里我们也经常拿不到完美的 Stack Trace）。对于托管代码来说，这绝对是个问题，因为栈里面很可能包含有托管代码的栈帧（托管代码的栈帧中包含了需要报告的 GC 引用）。

对于托管代码来说，我们对它有一些附加要求：它不仅需要在执行时跟踪所有的 GC 引用，还必须能够回溯到它的调用者。除此之外，当我们从托管代码进入非托管代码的世界时（或者非托管代码调用托管代码也一样），托管代码必须进行额外的操作来绕过非托管代码无法进行栈展开的问题。在实践中，托管代码会把所有包含托管栈帧的内存块都互相连起来。因此，虽然我们还是没办法不借助调试信息来展开非托管栈帧，但是我们能够做到始终能够找到一块托管代码产生的栈内存，然后遍历所有的托管栈帧块。

[1] 大多数最新的平台 ABI（Application Binary Interfaces）都定义了包含这种信息的约定，但通常并不是强制性的。

1.4.2.2 托管代码的世界 当每次进入、退出托管代码的世界时，就必须执行这种额外的机制。不论进入还是退出，CLR 都一定会知道。这两个世界泾渭分明（在任何时刻，代码要么在托管世界，要么在非托管世界）。更进一步，因为托管代码的执行基于一种 CLR 熟知的格式（以及使用公共中间语言，CIL），并且是 CLR 将其转换为能够在硬件上直接执行的指令，因此 CLR 能够做出比只是 “执行” **多得多** 的操作。比如，CLR 能够改变 “从一个对象中读取成员” 或 “调用一个函数” 的意义。事实上，CLR 在创建 MarshalByReference 对象时就是这么做的。它们看起来像是普通的本地对象，但事实上它们可能保存于另一台机器上。简而言之，在 CLR 的托管世界中存在大量的**⊠行⊠子**，它们可以用来实现非常强大的功能。我们下文会详细介绍。

除此之外，托管代码还带来了另一个重要影响，虽然可能不那么明显。在非托管的世界，是没有 GC 指针的（因为它们无法被追踪），同时托管代码调用非托管代码还存在着额外的开销。这就意味着，虽然你**可以**调用任意的非托管代码，但这种体验不是很友好。非托管方法的参数和返回值并不包含 GC 对象，也就是说，它们所创建和使用的对象及对象句柄需要显示释放。同时，这些 API 还无法使用 CLR 所支持的功能（例如异常和继承），它们与托管代码在用户体验上并不统一。

结果就是，非托管的接口在总是 **包装**之后才提供给托管代码使用。例如，当访问文件的时候，你并不会直接使用操作系统提供的 Win32 CreateFile 函数，而是使用包装了文件操作的 System.IO.File 类。让用户直接使用非托管的功能确实非常少见。

尽管这种包装看起来没什么好处（增加了很多没干什么事情的代码），但其实它们的价值非常大。我们总是**可以** 直接使用非托管的接口，但我们 **⊠⊠了**包装它们。为什么？ 因为运行时的终极目标是**使编程变得简单**，通常来说非托管函数并不足够简单。常见的情况是，非托管的接口在设计时并没有时刻考虑易用性，而是优先满足完整性。如果你看过 CreateFile 或是 CreateProcess 的参数列表，你很难把他们归为 “简单” 那一类的接口。幸运的是，这些功能在托管世界中被 “整容” 了，尽管这种 “整容” 没什么技术难度（就是重命名、简化并重新组织相关功能），但仍然非常实用。CLR 最重要的文档之一就是 Framework 设计指南，这篇 800 多页的文档详细描述了创建新的托管类库的最佳实践。

因此，我们可以看到，托管代码（与 CLR 紧密相连）与非托管代码在两方面有着显著的不同：

1. 有技术含量的一面：托管代码有自己完全不同的世界，CLR 能够细粒度地控制程序执行的几乎每个方面（可能能够细到每一条指令），CLR 还能够检测到指令执行何时会进出托管世界。这使得很多有用的功能得以实现。
2. 没什么技术含量的一面：托管代码调用非托管代码时存在调用开销，非托管代码无法使用 GC 对象。因此，将非托管代码进行包装是一种推荐的方式。接口会被“整容”，从而变得简单，并能够统一命名和设计规范，提高一致性和可发现性。

这两点特性对于托管代码的成功都非常重要。

1.4.3 内存和类型安全

由垃圾收集器带来的一个不那么明显、但影响深远的特性是：内存安全。内存安全不变量（invariant）的要求非常简单：如果一个程序只访问已经申请（同时还未释放）的内存，那么它就是内存安全的。这意味着，不会有任何“野指针”（悬空指针）指向某个随机的内存地址（更准确地说，不会指向提前释放了的内存）。很显然，我们希望所有程序都能够做到内存安全。悬空指针就是 bug，调试这种 bug 通常有一些难度。

GC 是提供内存安全保证的必要条件。

显然，垃圾回收器消除了用户提前释放内存的可能性（从而不会访问到没有正确申请的内存）。不过，不那么明显的是：如果想要确保内存安全，从实践上讲我们必须要有一个垃圾收集器。原因在于，对于那些需要**堆**（动态）内存申请的复杂程序，对象的生命周期基本上处于随意管理的状态（不像栈内存、或静态申请的内存，它们需要遵守高度受限的申请协议）。在这样的不受限的环境下，程序分析器无法确定需要在哪里插入显式的释放语句。实际上，决定何时释放内存的唯一途径就是在运行时确定。这其实就是 GC 的任务（检查某块内存是否仍然有效）。因此，任何需要在堆上进行内存申请的程序，如果想保证内存安全性，我们就**需要** GC。

GC 是保证内存安全的必要条件，但不充分。GC 并不会禁止程序越界访问数组，或是越界访问一个对象的成员（如果你通过基地址和偏移来计算成员地址的话）。不过，如果我们有办法解决这些问题，我们就能够实现内存安全的程序。

公共中间语言（CIL）**确**⌧提供了一些操作符，它们可以用来在任意内存上读取和写入数据（因此破坏了内存安全性），不过他还提供了下面这些内存安全的操作符，CLR 也强烈建议在大多数的情况下使用它们：

1. 字段访问操作符（LDFLD、STFLD、LDFLDA），它们能够通过名字来读取、写入一个字段，以及获取一个字段的地址。
2. 数组访问操作符（LDELEM、STELEM、LDELEMA），它们能够通过数据索引来读取、设置数组元素，以及获取数组元素的地址。所有的数组都有一个标签，写明了数组的长度。在每次访问数组元素时，都会自动进行边界检查。

使用这些操作符来取代那些低级的（同时也是不安全的）内存访问操作符，同时避免使用其他的不安全的 CIL 操作符（例如有一些操作符支持跳转到任意地址），这样的话我们就可以创建一个内存安全的系统了。但是，仅此而已。CLR 没有选择这条路；相反，CLR 选择了确保一个更强的不变量：类型安全。

对于类型安全来说，从概念上讲，每一块申请的内存都将与一种类型相关联。所有在内存地址上的操作都将在概念上使用有效的类型进行标记。类型安全需要保证的是，某一块标记了

某一种特定类型的内存，只能够进行这种类型允许的操作。这不仅确保了内存安全（没有悬空指针），同时它还对不同的类型提供了额外的保证。

在这些与类型相关的保证当中，最重要的保证之一就是（与字段相关联的）可见性控制属性（Attribute）。如果一个字段声明为 private（仅能够由这个类型中的方法所访问），那么这种限制就会被所有其他的类型安全的代码所遵守。例如，某个类型可能会声明一个名为 count 的字段，它代表了一张表里面对象的个数。假设这个 count 和这张表都是 private 的，同时我们假定代码一定会同时把这两个成员一起更新，那么现在我们就有了一个强保证：在所有类型安全的代码中，count 和这张表中的对象个数是一致的。当我们编写程序时，不论程序员知道与否，他们无时无刻都在利用着类型安全的概念。CLR 将类型安全由编程语言/编译器之间的简单约定，提升到可以在运行时也严格执行的强制约定。

1.4.3.1 可验证代码——强制内存安全和类型安全 从概念上说，为了保证类型安全，我们需要对程序的每一个操作进行检查，以便确保目标内存的类型是否与这种操作兼容。尽管我们可以做到这一点，但可想而知肯定非常慢。在 CLR 中，我们有 “CIL 验证” 的概念，在代码运行之前，我们会对 CIL 进行一次静态分析，进而确保大多数操作都是类型安全的。只有当这种静态分析无法满足需求时，运行时检查才有必要。在实践当中，需要运行时检查的情况其实并不多见，其中包括：

1. 将一个指向基类的指针转换为指向子类的指针（相反的操作可以进行静态检查）
2. 数组边界检查
3. 将指针数组中的一个元素赋值为一个新的（指针）值。需要这种检查的原因是，CLR 的数组支持自由转换规则（后文会详细介绍）

需要注意的是，运行时需要额外的特性来满足这些检查的需要：

1. 所有在 GC 堆上的内存必须标记其类型（以便转换操作能够执行）。它的类型信息在运行时必须能够获得，而且必须包含足够的信息来确定类型转换是否合法（例如，运行时需要知道继承结构）。事实上，GC 堆中的每一个对象的第一个字段都指向一个表示其类型的数据结构。
2. 所有的数组必须包含它的大小（以便进行边界检查）
3. 数组必须包含它的元素类型的完整类型信息

幸运的是，大多数看起来昂贵的要求（例如为每一个堆对象标记类型）已经是为了实现垃圾回收所必须的条件了（例如 GC 需要知道每个对象中需要扫描的字段），因此类型安全所带来的额外开销并没有多少。

因此，验证了代码的 CIL，又做了一些运行时检查，CLR 能够确保类型安全（以及内存安全）。然而，这种额外的安全需要在编程的灵活性上做出一点牺牲。CLR 提供了通用的内存访问操作符，但为了让代码可以验证，这些操作符的使用需要收到一定的限制。具体来说，目前所有的指针算术计算都会让验证失败，因此很多经典的 C/C++ 约定无法在可验证代码中使用；我们必须使用数组来替代。不过虽然这限制了一点编程的灵活性，但它并不是件坏事（数组很强大），带来的好处也很明显（烦人的 Bug 少了很多）。

CLR 机器鼓励使用可验证的、类型安全的代码。尽管如此，还是有时候需要不可验证的程序（主要是与非托管代码打交道时）。CLR 是允许这样的情况的，但最好把这样的代码尽可能的加以限制。常见的程序只需要一小块不安全的代码，其余的代码都可以是类型安全的。

1.4.4 高级特性

支持垃圾回收给运行时带来了很大的影响，因为他要求所有的代码必须支持额外的跟踪记录。我们对类型安全的期望同样给运行时带来了很大的影响，不仅要求程序的描述（即 CIL）支持字段和方法附带详细的类型信息，还要求它对其他的类型安全的高级编程语言结构提供支持。以类型安全的方式来表达这些结构同样需要运行时的支持。这两点重要的高级特性用来支持面向对象编程中最基础的两个要素：继承和虚调用分发。

1.4.4.1 面向对象编程 从机械的角度来讲，继承相对简单一些。它的基本思想是：如果 `derived` 类型的字段是 `base` 类型的字段的超集，那么只要将 `derived` 字段中的 `base` 那一部分字段放在前面，那么所有接受 `base` 指针的代码都能够接受 `derived` 对象，这样代码依然能够工作。这样的话，我们就说 `derived` 是继承自 `base`，代表着它能够在任何需要 `base` 的地方使用。这样的话，代码就变得**多**，因为同样的代码作用于很多不同的类型。由于运行时需要知道什么样的类型转换是合法的，因此运行时必须形式化它所支持的“继承”，以便能够对类型安全进行验证。

虚调用分发泛化了继承多态。它允许基类型声明某个方法能够被子类所**重**。使用 `base` 类型的代码可以调用虚方法，这些调用会在运行时根据对象的真实类型分发至正确的重载方法。这种**运行的分**可以不需要运行时的直接支持，而是使用基本的 CIL 指令来实现，但这样做有两点很重要的弊端：

1. 这样做可能有违类型安全（分发表一旦出错，将会带来灾难性的后果）
2. 每个面向对象语言可能会使用一些稍微不同的方法来实现它的虚分发逻辑。结果就是，这些语言之间的互操作性受到了影响（在一门语言中无法继承由另一门语言实现的基类）

正因如此，CLR 对基本的面向对象特性提供了直接支持。CLR 在最大的程度上尽量保证它的继承模型“语言中立”，因为不同语言之间仍然有可能共享相同的继承结构。然而，这并不是一定能够实现的。具体来说，多重继承可以通过很多种方式来实现。CLR 的选择是：不支持那些带有字段的类型的多重继承，但支持一些不含有字段的特殊类型（即 interface）的多重继承。

值得注意的是，尽管运行时支持这些面向对象的概念，我们并不需要一定使用它们。没有继承概念的语言（例如函数式语言）只需要简单地抛弃这些特性就好了。

1.4.4.2 值类型和装箱（Boxing） 在面向对象编程中，一个深远又微妙的影响是对象标识：我们能够区分出通过不同的内存申请调用产生的不同对象，就算是两个对象中的所有字段全部相等也没关系，这是由于对象使用的是引用（指针）而不是通过值来进行访问的。如果两个变量持有同一个对象（他们的指针指向同样的内存），那么更新某一个变量就会影响到另一个变量。

然而，这种对象标识的概念并不是对所有的类型都合适。举例来说，大多数程序员不会把整数看作是对象。如果在两个不同的地方申请了数字“1”，程序员通常会希望这两个东西相等，并且不想要更新某一个时影响另一个。事实上，有一大类编程语言（函数式语言）就在极力避免“对象标识”与引用语义。

尽管我们额能够做出一个“纯”面向对象系统，其中所有的东西（包括整数）都是一个对象（就像 Smalltalk-80 一样），但在这一层统一性下面，我们还是由很多工作要做，才能够得到一种高效的实现。其他的语言（比如 Perl、Java、Javascript）采用了一种实用的方法，它们讲某些类型（例如整型）看作是值类型，其他的类型使用引用类型。CLR 同样选择了一种混合模型，但区别在于，它允许用户自定义值类型。

值类型的关键特点在于：

1. 每一个值类型的局部变量、字段和数组元素都包含了值的独有拷贝。
2. 当一个变量、字段或者数组元素进行赋值操作时，值会被拷贝。
3. 相等性永远使用变量中的数据进行定义（而不是它的地址）。
4. 每一个值类型都有一个对应的引用类型，这个引用类型只有一个隐式的、未命名的字段。这叫做这个值类型的装箱值（boxed value）。装箱值类型可以参与继承，并且拥有对象标志（不过非常不推荐使用一个装箱值类型的对象标志）。

值类型与 C（和 C++）中的结构体有些相似。像 C 一样，你可以使用指针指向值类型，但这个指针类型与结构体的类型是不同的。

1.4.4.3 异常 另一个 CLR 直接支持的高级语言结构是异常。异常允许程序员在发生错误时**抛出**一个任意的对象。当这个对象被抛出时，运行时就会搜索调用栈，去寻找是否有哪个方法声明了它可以**捕捉**这个异常。如果这样的捕捉声明存在，程序就继续从捕捉声明这里执行。异常的用途在于它规避了程序员忘记检查某个方法是否成功。异常有助于程序员规避错误（因而使得编程变得简单），因此 CLR 支持它们也就不奇怪了。

尽管异常能够避免这类常见错误，但它们无法解决另一类问题：在异常发生时，如何将数据恢复至一致的状态。这就是说，在异常被捕获之后，很难讲如果继续执行的话会不会发生（由第一次的错误而引起的）其他的错误。这一方面是 CLR 在未来值得拓展的地方。不过就目前来说，异常仍然是向前迈出的一大步（我们还需要走得更远）。

1.4.4.4 参数化类型（泛型） 在 CLR 2.0 版本之前（译注：非 CoreCLR），数组是唯一一个参数化的类型。所有的其他容器（比如哈希表、列表、队列等等）都操作于通用的 Object 类型之上。无法创建一个 `List<T>` 或者 `Dictionary<KeyT, ValueT>` 在性能上会有所劣势，因为这些类型都需要在容器的接口处进行装箱，并在取出元素时进行显式类型转换。然而，这些都不是 CLR 加入参数化类型的根本原因。最主要的原因是，**参数化类型能够使编程变得更简单**。

想知道原因的话，我们可以想象一下，一个只使用通用的 Object 类型的类库是什么样子的，这与那些动态类型语言（比如 Javascript）很像。在这种情况下，程序员有非常容易写出不正确（但是类型安全）的程序。这个方法的参数应该是一个列表吗？ 一个字符串？ 还是一个整数？ 从方法的签名就很难得到答案。更糟糕的是，当一个方法返回了一个 Object，哪些方法可以接受它作为参数？ 通常来说，一个框架可能有成百上千种方法；如果它们所有的参数都是 Object 类型，就很难判断哪些 Object 对象是这个方法所需要的。简而言之，强类型能够帮助程序员更清晰地表达出他的意图，同时还允许工具（例如编译器）来确保他的意图一定会实现。这样就极大地提升了生产力。

当我们谈到列表和字典等容器时，这些优点仍然成立，因此参数化类型是非常有价值的。下面需要考虑的问题是，我们是选择把参数化类型作为一门语言的一项特性，然后在编译时将这一层概念抹掉，还是说应该作为运行时的一等公民提供支持？ 其实哪一种实现都可以，CLR 团队选择了一等公民支持。原因在于，不这样的话，每一种语言都可能会有不同的参数化类型的实现方式。这就意味着互操作也会变得麻烦起来。最重要的是，使用参数化类型来表达程序员的意图尤其在类库的**接口上** 非常有用。如果 CLR 不正式支持参数化类型，那么类库就无法使用它们，就会丢掉这条特性的一个重要的使用场景。

1.4.4.5 程序即数据（反射 API） CLR 的基础功能是垃圾回收、类型安全、以及高级语言特性。这些基础的特性使得 CIL 需要在一个相对高级的层次制定规范。在运行时，我们能够得到非常丰富的信息（相反，C 或 C++ 程序就不存在），把它们暴露给程序员使用就非常有价值。这样的想法催生了 System.Reflection 接口（之所以叫做反射是因为这些接口允许程序（通过自己的反射）看到自己）。这一套接口允许我们探索一个程序的绝大部分方面（例如它都有哪些类型、继承关系、拥有哪些方法和字段）。事实上，由于只有很少的信息丢失，托管代码拥有一些非常好的“反编译器”（例如 NET Reflector）。尽管在知识产权保护方面可能会让人感到忧虑（但其实我们可以通过一种叫做**混淆**的方式有意擦除这些信息），但这也恰好证明了，托管代码在运行时仍然拥有很丰富的信息。

除了在运行时检视程序外，我们还能够对其进行一些操作（例如调用方法、设置字段等等），而最强大的功能可能是在运行时从零开始生成代码（System.Reflection.Emit）。事实上，运行时类库使用这种方式来创建匹配字符串的特化代码（System.Text.RegularExpressions），以及创建用来“序列化”对象（使得对象能够存于文件或在网络上传输）的代码。这在以前是办不到的（你需要写一个编译器！），但是 CLR 所提供的这类能力使得很多编程问题变得更加容易解决。

尽管反射功能确实非常强大，但在使用上需要小心。反射要比静态编译出的代码慢上很多，而更重要的是，自我引用的系统更加难以理解。也就是说，只有当应用价值非常大、需求非常明确时，才应该使用 Reflection 或 Reflection.Emit。

1.5 其他功能

最后要介绍的运行时功能与 CLR 的基础架构（GC、类型安全、高级规范）无关，但仍然时任何完备的运行时系统都需要拥有的特性。

1.5.1 与非托管代码的交互

托管代码需要使用非托管代码中的功能。CLR 提供两种不同“口味”的交互方法，一种是直接调用非托管函数（叫做 Platform Invoke，PINVOKE）；除此之外，非托管代码同样有一种面向对象的交互模型，名曰 COM（Component Object Model），与 Ad-Hoc 方法调用相比，他更加结构化一些。由于 COM 同样拥有对象模型和其他约定（例如错误是如何处理的、对象的声明周期等），在有特别支持的情况下，CLR 与 COM 之间的交互会更容易。

1.5.2 提前编译（Ahead of Time）

在 CLR 的模型中，托管代码以 CIL 的形式分发，而不是原生代码。CLR 在运行时将 CIL 翻译为原生代码。作为一种优化，可以使用 crossgen 工具（类似于 .NET Framework 中的 NGEN）将 CIL 转化为原生代码，并保存下来。这能够在运行时节省大量的编译时间。由于类库的规模十分庞大，这一功能非常重要。

1.5.3 多线程

CLR 非常重视托管代码对与多线程的需求。从一开始，CLR 类库就包含了 System.Threading.Thread 类，它是对操作系统线程的 1 对 1 的包装。然而，正因为它是对操作系统线程的包装，创建一个 System.Threading.Thread 相对昂贵（需要花费数毫秒来启动）。对于很多操

作来说这也许够了，但有时程序需要创建一些很小的工作任务（比如只需要花费数十毫秒）。这在服务器编程上很常见（例如，每一个任务都只服务于一个网页），在需要利用多处理器的算法种也很常见（例如多核排序算法）。为了支持这些场景，CLR 还提供了 ThreadPool 的概念，用来完成一个个工作任务，同时由 CLR 负责创建这些必要的线程。尽管 CLR 确实直接暴露了 ThreadPool（System.Threading.ThreadPool 类），但推荐使用的机制是 Task Parallel Library，它提供了对常见的并发控制的额外支持。

从实现的角度来说，ThreadPool 的创新之处在于是由 CLR 负责确定合理的工作线程数目。CLR 使用了一种反馈系统，它监视着吞吐率与线程的数量，并调整线程的数量以便最大化吞吐量。这能够让程序员直接关注于使用并发（即创建工作任务），而不是去先思考如何设置正确的并发量（这取决于工作负载和硬件）。

1.6 总结和资源

啊~！ 运行时实在是做了太多事了！ 我们花了很长的篇幅，只介绍了运行时的**部分**功能，也还没有深入内部细节。我希望，这篇介绍性文章能够帮助你对这些内部实现有一个更深的理解。这篇文章介绍过的内容有：

- CLR 运行时是用来支持编程语言的一整套框架
- 运行时的目的是让编程变得简单
- 运行时的首要功能为：
  - 垃圾回收
  - 内存安全和类型安全
  - 对高级语言功能的支持

1.6.1 值得一看的链接

- MSDN 上关于 CLR 的页面
- Wikipeida 上关于 CLR 的页面
- CLI 的 ECMA 标准
- .NET Framework 设计指南
- CoreCLR 仓库文档

2 CLR 垃圾回收器的设计

> 这是一篇译文。作者：Maoni Stephens (@maoni0) - 2015
> 原文链接 https://github.com/dotnet/coreclr/blob/master/Documentation/botr/garbage-collection.md

注：请参考 The Garbage Collection Handbook 来了解更多关于垃圾回收话题的通用知识；如果希望了解关于 CLR GC 的特定知识，请参考 Pro .NET Memory Management 一书。在本文最后列出了其他可供参考的资源。

2.1 整体结构

CLR GC 由两部分组成：分配器和回收器。分配器用来申请更多的内存，并在合适的时机触发垃圾回收。回收器收集内存垃圾，即程序不再使用的那些对象所占用的内存。

除了分配器，还有其他的途径能够触发回收器，例如手动调用 GC.Collect，或是 Finalizer 线程收到了 LowMemory 的异步通知。

2.2 分配器的设计

分配器由执行引擎（Execution Engine，EE）通过一些帮助函数进行调用，会传递给分配器以下信息：

- 申请的内存大小
- 线程分配上下文
- 标志位，例如标记这个对象是否 Finalizable

分配器并不关心对象类型的信息，它只会通过 EE 得到对象的大小。基于对象大小，GC 会将对象分为两类：小对象（小于 85000 字节）和大对象（大于等于 85000 字节）。理论上来说，大对象、小对象不需要区别对待，但由于对大对象进行整理（Compact）的代价较大，因此 GC 会做出这样的区分。

分配器会向 GC 申请内存。GC 会以分配上下文（Allocation Context）的形式向分配器提供内存，分配上下文的大小由分配量（Allocation Quantum）决定。

- 所谓分配上下文，是指一个堆内存段（Heap Segment）上的一小部分，将会由某个线程单独使用。在单一逻辑处理器的机器上，只会使用唯一一个上下文，作为第 0 代分配上下文。
- 所谓分配量，是指当分配器需要更多的内存来创建对象时，它每次获得的新内存空间的大小。分配量通常定义为 8k 字节，而托管对象的平均大小大约为 35 字节，这使得分配器能够在一个分配上下文中创建很多个对象。

大对象不使用分配上下文和分配量，它们本身就可能要比常规的分配量大得多；同时，分配上下文的优势仅能在小对象中体现出来。因此，大对象会直接在堆内存段上申请。

分配器的特点包括：

- 在合适的时机触发垃圾回收：当分配的内存大小超过一定分配额度（Allocation Budget）之后、或是分配器在一个堆内存段上没有内存可用的时候，它就会触发一次垃圾回收。关于分配额度和托管段（Managed Segment），后文还会详细介绍。
- 保持对象局部性：在同一个对内存段上的对象，它们的虚拟地址会依次相邻。
- 高效利用缓存：分配器每次会一次性申请分配量大小的内存，而不是每个对象都来申请一次内存。它会把这些缓存都清零，从而使得 CPU 能够预先缓存这块内存。
- 减少加锁需求：由于一个分配上下文和分配量只由一个线程使用，因此只要当前的分配上下文没有耗尽，就无需加锁。
- 内存完整性：GC 总是会为新对象清零内存，因此不会存在指向随机地址的对象引用。
- 保证堆内存可被爬取：分配器会保证每一个分配量中无法分配的剩余内存作为一个空闲对象（Free Object）管理。例如，如果一个分配量只剩下 30 字节的空间，但新对象需要 40 字节，那么分配器会创建一个 30 字节大小的空闲对象（译注：空闲对象会加入空

闲列表中进行管理，从而不会白白浪费），同时申请一个新的分配量大小的内存来分配新的对象。

2.2.1 内存分配 API

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

大对象和小对象在分配内存均可以使用上面的函数。除此之外，大对象的分配还会使用下面这个函数：

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

2.3 回收器的设计

2.3.1 GC 的目标

GC 的目标是极度高效地管理内存，使得编写 “托管代码” 的人几乎不需要付出额外的成本即可受益。“高效” 指的是：

- GC 的执行频率需要足够高。这可以避免托管堆中残留大量的无用对象（即垃圾），而导致占用过多不必要的内存。
- GC 的执行频率需要尽量低。这可以避免浪费宝贵的 CPU 时间。
- 一次 GC 需要很有效。如果一次 GC 后几乎没有回收到内存，那么这次 GC（以及它所占用的 CPU 时间）就被浪费了。
- 每次 GC 都需要很快。很多程序有低延迟的需求。
- 托管代码的开发人员不需要知道 GC 的细节就可以享受到不错的内存使用率。也就是说，GC 应该自行调节，以便使用不同的内存使用模式。

2.3.2 托管堆的逻辑表示

CLR GC 是一个分 “世代” 的垃圾回收器，即对象在逻辑上会被分为不同的世代。当回收器回收了第 N 代后，仍然存活的对象将被标记为第 N + 1 代，这个过程叫做升级（Promotion）。不过，这里也存在例外，比如我们可能会考虑让某个对象不升级或者降级（demote）。

对于小对象来说，托管堆被分为 3 个世代：gen0、gen1 和 gen2。对于大对象来说，我们只有一个世代—— gen3。gen0 和 gen1 统称为短暂世代（Ephemeral Generations），代表着这两个世代中的对象存活时间较短。

小对象堆中的世代编号就代表了辈分—— gen0 是最年轻的世代。不过，这并不意味着 gen0 中的所有对象都要比 gen1 或者 gen2 中的对象更年轻，下文会提到一些例外的情况。当我们对某一个世代进行垃圾回收时，也会同时回收所有比它年轻的世代。

从理论上讲，大对象的处理方式也可以像小对象一样。然而，由于整理（Compact）大对象的代价比较高，因此我们将大小对象区别对待。大对象只有一个世代 gen3，考虑到性能原因，它会与 gen2 一同进行垃圾回收，因为 gen2、gen3 世代相对庞大；而短暂世代（gen0、gen1）中的对象往往生存期短，对它们进行回收时则会尽量限制性能开销。

分配内存时，会从最年轻的世代开始——对于小对象来说就是 gen0，大对象来说就是 gen3。

2.3.3 托管堆的物理表示

托管堆由一组托管堆段组成。一个堆段是一块连续的内存，它是由 GC 向操作系统申请而得。堆段分为两种：小对象堆段和大对象堆段。在每一个堆上，堆段之间会以链式连接。一个程序的内存中至少会存在一个小对象堆段和一个大对象堆段，这两个堆段会在 CLR 启动时被默认建立。

在每一个小对象堆中，仅会有一个堆段用来存放 gen0 和 gen1，它叫做“短暂世代堆段”。这个堆段也可以同时用来存放 gen2。除了这个堆段之外，还可能会有一个或多个段用来存放 gen2。

大对象堆中也可能会有多个堆段。

一个堆段会以地址“从低到高”的方式进行使用，也就是说，低地址上的对象要比高地址上的对象更老。当然，下文会提到一些例外情况。

堆段可以按需申请。如果一个堆段不包含任何任何对象，那么它就会被删除。不过，一个堆中的初始堆段是个例外，它会始终存活。对于每个堆来说，小对象的垃圾回收、或者大对象的内存分配，都有可能会触发新的堆段申请，每次只会申请一个新堆段。这样的设计能够提供更好的性能，因为大对象会与 gen2 一同进行回收，比较耗时。

堆段会按照它们的申请时间依次相连。最后的堆段一定是短暂世代堆段。对于小对象堆来说，没有对象的堆段可以重复利用，作为新的短暂世代堆段。小对象在申请内存时，只会在短暂世代堆段上进行；而大对象在申请内存时，则会在整个大对象堆上进行。

2.3.4 分配额度（Allocation Budget）

分配额度是与世代相关的一个逻辑概念，它是触发在这一世代上进行垃圾回收的阈值。这个额度的设置与这一世代的存活率有关。如果存活率较高，那么分配额度也将设置的更高，这样当下次垃圾回收时就能够回收到更多的垃圾。

2.3.5 如何决定对哪一个世代进行回收

当触发了垃圾回收时，垃圾回收器要做的第一件事就是确定回收哪一个世代。除了分配额度，还有其他的因素需要考虑：

- 世代的碎片化程度——如果一个世代的碎片化程度很高，那么在这个世代上进行垃圾回收将会更富有成效。
- 当机器的内存负载很高时，如果回收某一世代有可能能够释放内存空间，那么垃圾回收器会更加激进地进行回收。这对避免比不要的（跨机器）分页很重要。
- 如果短暂对象堆段空间不足时，垃圾回收器会更加激进地对 gen1 进行回收，来避免申请一个新的堆段。

2.4 垃圾回收的流程

2.4.1 标记（Mark）阶段

标记阶段的目标是找到所有的存活对象。

分世代的垃圾回收器的优势之一在于，它可以只对堆中的一部分对象进行回收，而不是去一次性考虑所有的对象。当回收短暂世代时，垃圾回收器需要知道这些时代中哪些对象仍然有效。执行引擎能提供它持有的所有对象的信息；然而，更老世代中的对象也可能持有年轻世代中的对象引用。

垃圾回收器通过利用“卡片”（Card）标记，来更快地确定更老世代中的对象是否持有年轻世代中某个对象的引用。卡片由 JIT 帮助方法在赋值操作发生时进行设置。当 JIT 帮助方法发现一个了短暂世代中的对象被其他对象持有时，它将会设置短暂对象中相应的卡片字节以便标记引用源的大致位置。在回收短暂世代的过程中，垃圾回收器可以通过对象的卡片标记值来有选择地扫描内存，而不是把所有的内存都扫描一遍。

2.4.2 计划（Plan）阶段

计划阶段模拟了一次整理（Compact）过程，用来确定对这一世代是否需要整理。如果不需要的话，垃圾回收器则会执行清扫（Sweep）。

2.4.3 重定位（Relocate）阶段

如果垃圾回收器决定进行整理，那么被整理的对象的位置将会发生改变，因此必须要更新这些对象的引用。重定位阶段需要找到被回收世代中对象的所有引用。与之相比，标记阶段只需要找到那些会影响对象生命周期的引用，而不必考虑弱引用。

2.4.4 整理（Compact）阶段

这个阶段比较简单。由于在计划阶段已经计算好了那些需要移动的对象的新位置，整理阶段只需要将它们拷贝到目标地址即可。

2.4.5 清扫（Sweep）阶段

清扫阶段会找出那些夹在存活对象之间的垃圾空间，垃圾回收器会创建一个“自由对象”（Free Object）来占据这些空间，相邻的垃圾空间也会被合并进一个自由对象。这些自由对象会被放入自由对象列表 `freelist` 中。

2.5 代码流程

术语：

- WKS GC：Workstation GC
- SVR GC：Server GC

2.5.1 各种配置下的行为

2.5.1.1 WKS GC，关闭并发 GC

1. 用户线程的分配额度不足，触发了垃圾回收
2. 垃圾回收器调用 `SuspendEE` 来暂停托管线程
3. 垃圾回收器决定对哪个世代进行回收

4. 执行标记阶段
5. 执行计划阶段，并决定是否执行整理
6. 如果需要整理，那么即执行重定位阶段和整理阶段；如果不需要整理，就执行清扫阶段
7. 垃圾回收器调用 RestartEE 重新恢复托管线程的执行
8. 用户线程继续执行

2.5.1.2 WKS GC，开启并发 GC

这一节描述了后台 GC 是如何工作的。

1. 用户线程的分配额度不足，触发了垃圾回收
2. 垃圾回收器调用 SuspendEE 来暂停托管线程
3. 垃圾回收器决定是否需要启动后台 GC
4. 如果需要的话，那么一个后台 GC 线程将被唤醒。后台 GC 线程调用 RestartEE 来恢复托管线程
5. 托管线程继续分配内存，于此同时后台 GC 线程仍然在进行工作
6. 用户线程可能会由于分配额度不足，触发一次短暂 GC (即 “前台 GC”)。短暂 GC 与上文 “WKS GC，关闭并发 GC” 的流程一致
7. 后台 GC 再次调用 SuspendEE 以便完成标记阶段，随后它会调用 RestartEE 重新恢复用户线程，此时清扫阶段与用户线程并发执行
8. 后台 GC 结束

2.5.1.3 SVR GC，关闭并发 GC

1. 用户线程的分配额度不足，触发了垃圾回收
2. Server GC 线程被唤醒，它们会调用 SuspendEE 来暂停托管线程
3. Server GC 线程执行垃圾回收（与 WKS GC 关闭并发 GC 时的操作一致）
4. Server GC 线程调用 SuspendEE 恢复托管线程
5. 用户线程继续执行

2.5.1.4 SVR GC，开启并发 GC

这一场景与 WKS GC 开启并发 GC 的行为大致一致，只是后台 GC 任务是在 Server GC 线程上完成的。

2.6 物理架构

这一节将有助于理解垃圾回收的代码流程。

用户线程的分配量不足，它会通过 try_allocate_more_space 来申请新的分配量。当需要触发 GC 时，try_allocate_more_space 会调用 GarbageCollectGeneration。如果使用的是 WKS GC 并关闭了并发 GC，则 GarbageCollectGeneration 是在用户线程上执行的。代码流程为：

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

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

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

plan_phase()
{
    // actual plan phase work to decide to
    // compact or not
    if (compact)
    {
        relocate_phase();
        compact_phase();
    }
    else
        make_free_lists();
}
```

如果使用的是 WKS GC 并开启了并发 GC (这是默认情况)，后台 GC 的代码流程为:

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

garbage_collect()
{
    generation_to_condemn();
    // decide to do a background GC
    // wake up the background GC thread to do the work
    do_background_gc();
}

do_background_gc()
{
    init_background_gc();
```

```
    start_c_gc ();

    //wait until restarted by the BGC.
    wait_to_proceed();
}

bgc_thread_function()
{
    while (1)
    {
        // wait on an event
        // wake up
        gc1();
    }
}

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

2.7 其他资源

- .NET CLR GC Implementation
- The Garbage Collection Handbook: The Art of Automatic Memory Management
- Garbage collection (Wikipedia)
- Pro .NET Memory Management

3 CLR 类型加载器的设计

这是一篇译文。作者：Ladi Prosek - 2007
原文链接 https://github.com/dotnet/coreclr/blob/master/Documentation/botr/type-loader.md

3.1 引言

在一个基于类的面向对象系统中，类型就是一种模板，它描述了每个独立的实例所包含的数据、以及它们能够提供的功能。如果没有定义一个对象的类型，我们就不可能创建出这个对象来 1。如果我们说某两个对象的类型相同，那么它们一定是同一个类型的两个实例。事实上，这两个对象也定义了完全相同的成员，但这与类型判断完全无关。

上面这段话其实也很好的描述了一个典型的 C++ 系统。不过 CLR 还有一个非常重要的基本功能，那就是它能够提供完整的运行时类型信息。为了“管理”托管代码、并提供一个类型安

全的环境，运行时必须在任何时刻都能够知晓任何对象的类型。获取类型信息时，也不能引入复杂的计算，因为“类型比较”这一操作会非常频繁地发生（比如，任何类型转换都会涉及查询当前对象的类型信息，以便确定转换是安全的、可以操作的）。

这种对性能上的需求，就把字典查询方法完全排除在外了。留给我们的，就只剩下下面这样的架构：

图 1 抽象、宏观的对象设计

在每个对象中，除了包含真正的实例数据，还有一个“type id”指针，指向一个代表了它的类型的结构。这样的概念与 C++ 中的虚表指针很像，然而不同的是，这种结构（我们先把它叫做“TYPE”，后面我们会更清楚地定义它）比虚表所包含的信息要多很多。比如，它还包含了类型的继承信息，以便我们能够回答“is-a”这样的问题。

1 C# 3.0 中的“匿名类型”能够让你无需显式定义类型即可创建一个对象，只需要直接列出它的所有字段即可。不要被这个功能蒙蔽双眼，编译器其实背着你创建了一个类型。

3.1.1 相关阅读

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

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

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

3.1.2 设计目标

有时我们也把类型加载器（type loader）叫做类加载器（class laoder），但严格来说这个叫法并不正确，因为类只能算是类型的一个子集，即引用类型。这个加载器也能加载值类型。类型加载器的终极目标是：当有人需要加载一个类型时，它能够构建出表示这个类型的数据结构。下面的这些性质是加载器所应该具备的：

- 快速的类型查找（通过 Module 和 token 进行查找，或通过 Assembly 和类型名进行查找）。
- 对内存布局进行优化，以便占用较小的内存工作集、实现较高的缓存命中率、以及提高 JIT 编译后代码的效率。
- 类型安全——拒绝加载不正确的类型，并抛出 `TypeLoadException`.
- 并发——能够扩展至多线程的场景。

3.2 类型加载器的架构

加载器的公开方法很少。尽管这几个方法的函数签名不尽相同，它们都有着相似的语义：接收一个元数据 **token** 或是类型名 **name** 字符串参数，它代表了某个类型或成员；接收另一个参数，代表了这个 token 所在的范围（某个 **module** 或是 **assembly**）；再接收一些额外信息，比如一些标志。它会以 **handle** 的型时返回加载好的实体。

在 JIT 时，通常会调用多次类型加载器。考虑这样的代码：

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

在 IL 层面，MyClass 以元数据 token 的方式被引用。JIT_New 帮助方法是真正做实例化工作的函数，为了生成对 JIT_New 调用指令，JIT 会要求类型加载器加载这个类型，并返回一个 handle。这个 handle 会被直接以立即数的形式嵌入到 JIT 编译后的代码中。由于类型和成员是在 JIT 阶段被解析和加载的（而不是在运行时），因此下面这样的代码很具有迷惑性：

(译注：CLR 会先对函数进行 JIT 编译，而后才会执行它)

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

如果 MyClass 类型加载失败（比如它所在的 Assembly 刚好在文件系统中被删掉了），它们这段代码仍然会抛出 TypeLoadException。catch 块没有捕获到这个异常的原因是是：这段代码根本没有执行！ 异常是在 JIT 时被抛出的，只能被调用 CreateClass 的函数（进而触发了 JIT 编译）所捕捉。不仅如此，考虑到存在内联（inline）这一特性，触发 JIT 编译的时机有时并不是那么明显，因此用户不应该依赖于这种难以琢磨的行为。

3.2.1 关键数据结构

在 CLR 中，最通用的类型表示方法是 TypeHandle。它是对 MethodTable 以及 TypeDesc 的抽象，即它要么封装了一个指向 MethodTable 的指针（它代表了 “普通” 的类型，比如 System.Object 或是 List<string>），要么封装了一个指向了 TypeDesc 的指针（代表了 byref、指针、函数指针、数组以及泛型类型）。它就是类型的标志，当且仅当两个 handle 相同时，它们所表示的类型也相同。为了节省空间，TypeHandle 通过指针的第二低位（the second lowest bit）来标记它到底是一个 TypeDesc 还是 MethodTable 。如果第二低位是 1 (即 (ptr | 2))，则代表它是一个 2。TypeDesc 则是对下面几种类型的抽象：

图 2 TypeDesc 体系

TypeDesc

抽象的类型描述符。具体的描述符类型由标志位决定。

TypeVarTypeDesc

代表了一个类型变量，例如在 List<T> 或是 Array.Sort<T> 中的那个 T（详见下文关于泛型的部分）。类型变量不会由多个类型或方法共享，因此每个变量都只有一个 owner。

FnPtrTypeDesc

代表了一个函数指针，它是由一组变长的类型 handle 列表组成的，这组 handle 表示了返回值和参数的类型。这个描述符并不常见，因为 C# 不支持函数指针。不过，托管 C++ 会使用

这个描述符。

ParamTypeDesc

这个描述符代表了 byref 和指针类型。在 C# 的方法参数中的使用 ref 和 out 关键字即可得到这种类型 3；指针类型则代表了非托管的指向数据的指针，用于 unsafe C# 和托管 C++。

ArrayTypeDesc

代表了数组类型。它继承自 ParamTypeDesc，因为它同样只有一个类型参数（即元素的类型）。这与泛型实例化不同，泛型实例化所需要的参数是不定的。

MethodTable

目前，运行时中最核心的类型数据结构就是它了。它代表着所有没有落入到上述类别中的类型（包括基本类型、开放（open）或闭合（closed）泛型类型）。它包含了所有需要快速查找的信息，例如它的父类型、实现的接口，以及虚表。

EEClass

MethodTable 数据分为两类，“热（hot）”结构和“冷（cold）”结构，这将有利于降低内存占用以及更有利于缓存的使用。MethodTable 本身只用来存储常用（热）数据，即那些为了执行程序所必须的数据。EEClass 保存不常用的（冷）数据，这些数据通常只在类型加载、JIT 编译或是反射时才需要。每一个 MethodTable 都指向一个 EEClass。

此外，EEClass 在泛型类型之间是共享的。多个泛型 MethodTable 可能会指向同一个 EEClass。这就对能够存放在 EEClass 中的数据提出了额外的限制条件。

MethodDesc

顾名思义，这个结构描述了一个方法。其实这个结构通常都以它的一些子类型出现，不过大多数子类型都已经超出了这篇文章的范围。其中有一个子类型值得一提：InstantiatedMethodDesc，它在泛型类型中扮演了一个重要的角色。更多信息请参考 **Method Descriptor Design**。

FieldDesc

与 MethodDesc 类似，这个结构描述的是一个字段。除了某些特定的 COM 交互场景，执行引擎根本不在乎属性（property）和事件（events），因为它们最终还是会指向各种方法和字段。只有编译器、以及反射机制能够生成或理解属性和事件，这只是一种语法糖。

2 这在调试时非常有用。如果 TypeHandle 的值以 2、6、A、或 E 结尾，那么它就不是一个 MethodTable。如果想要访问到 TypeDesc，这个多余的位需要清零。

3 需要注意的是，ref 和 out 之间只在参数属性上有所不同。对于类型系统来说，它们其实是相同的类型。

3.2.2 加载级别（Load Levels）

我们可以使用诸如 typedef/typeref/typespec 之类的 **token**、并指定一个 **Module** 来加载类型。当类型加载器加载类型时，它并不会一次性做完所有的工作，而是分阶段完成的。这样做的原因是，有一些类型可能会依赖其他类型，因此如果要完成类型的加载，则必须先加载那些被依赖的类型，二者可能会导致无限递归和死锁。例如：

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

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

class C<T>
{ }
```

它们都是合法的类型，很明显 A 依赖于 B，同时 B 也依赖于 A。

加载类型时，加载器会先创建一些结构用来表示被加载的类型，此时并不需要加载其他被依赖的类型。这一步完成后，这些结构就可以被其它地方所引用，例如把指向它的指针塞进其他结构中。然后假踩起就会不断地一点一点把这些结构填满，一直到真正加载完这些类型。在上面的例子中，A 和 B 的基类会首先近似为一种不需要依赖其他类型的类型，然后才会被真正的类型所替代。

所谓的加载级别，就是为了定义这种未完全加载的状态。从 CLASS_LOAD_BEGIN 开始，到 CLASS_LOADED 结束，中间穿插了很多中间级别。在 classloadlevel.h 中，有很详细的注释对各个加载级别进行了说明。例如，类型可以以 NGEN 镜像的形式存储，但并不是简单地把它们映射到内存就能使用，而是需要“恢复”（restore）。加载等级中有一级叫做 CLASS_LOAD_UNRESTORED，它就描述了这种需要“恢复”的状态。

更多对于加载等级的详细解释，请参考 Design and Implementation of Generics for the .NET Common Language Runtime。

3.2.3 Generics

在没有泛型的世界里，一切都很友好、所有人都很开心——因为每个普通的类型（即不是由 TypeDesc 所表示的类型）都只有一个 MethodTable，其中包含了指向了它所关联的 EEClass 的指针，而 EEClass 又指回了这个 MethodTable。为了节省空间，代表了方法的 MethodDesc 几个组成一组，每组之间以链表的形式相连 4：

图 3 没有泛型方法的非泛型类型

4 在执行托管代码时，并不是需要去查询这些组才能进行方法调用。方法调用是一种非常常见的操作，通常只需要 MethodTable 中的信息就能完成。

3.2.3.1 术语 泛型形式参数（Generic Parameter）

是指一个占位符，能够被其他类型替代，例如声明 List<T> 中的 T。有时也称作形式类型参数（formal type parameter）。泛型形参具有名字，以及可选的泛型约束。

泛型实际参数 (Generic Argument)

是指用来替代形参的那个类型，例如 List<int> 中的 int。需要注意的是，泛型形参也可以被用作一个实参。例如：

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

这个方法有一个泛型形参 T，它被用作了泛型列表的泛型实参。

泛型约束

是可选的，它是指当泛型实参替代反省形参时所需满足的要求。不满足要求的类型不能替换反省实参，这是由类型加载器所强制要求的。泛型约束分为三类：

1. 特殊约束
   - 引用类型约束——泛型实参必须是引用类型（与之相对的是值类型）。C# 使用 `class` 关键字来表示这种约束。

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

   - 值类型约束——反省实参必须是除了 `System.Nulable<T>` 之外的值类型。C# 使用 `struct` 关键字来表示。

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

   - 默认构造函数约束——泛型实参必须要具有一个公开的无参构造函数。C# 使用 `new()` 来表示这种约束。

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

2. 基类型约束——泛型实参必须继承自（或者就是）给定的非接口类型。很显然，这种约束要么没有，要么只能有一个引用类型作为约束。

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

3. 接口实现约束——泛型实参必须实现（或者就是）给定的接口类型。多个接口可以同时作为约束：

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

上面这些约束之间的关系是“与”（AND），即一个泛型形参可以被约束为需要继承自一个给定的类型、实现几个接口、同时还需要有默认构造函数。类型声明中所有的泛型形参都可以用来表达约束，这可能会引入参数之间的互相依赖：

```
public class A<S, T, U>
    where S : T
    where T : IList<U> {
    void f<V>(V v) where V : S {}
}
```

实例（Instantiation）

是用来替换泛型类型或泛型方法中泛型形参的一组泛型实参。每个加载了的泛型类型和方法都有它自己的实例。

典型实例（Typical Instantiation）

是指按照泛型形参声明的顺序，仅包含泛型类型或方法自己的类型形参的一个实例。对于每一个泛型类型和方法，仅存在一个典型实例。通常来说，当我们提到开放泛型类型（Open generic type）时，就是指它的典型实例。例如：

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

C# 会把 typeof(A<,,>) 编译为一个 IdToken A'3，运行时就会加载使用 S、T、U 实例化的 A`3。

规范实例（Canonical Instantiation）

是指所有的泛型实参均为 System.__Canon 的实例。System.__Canon 是定义在 mscorlib 中的一个内部类型，它就只充当一种约定，与其他的泛型实参都不同。类型和方法的规范实例用来代表所有的其他实例，并且携带有会在所有实例之间共享的信息。显然，System.__Canon 无法满足泛型形参上可能携带的任何约束，因此对于 System.__Canon 约束检查是个特例，加载器会将不满足的约束忽略。

3.2.4 共享

随着泛型的加入，运行时需要加载的类型变得更多了。虽然泛型类型的不同实例（例如 List<string> 和 List<object>）是不同的类型，也各自有它们自己的 MethodTable，但还是有有一些信息是重复的，可以共享。这种共享会给内存占用和性能都带来积极影响。

图 4 不包含泛型方法的泛型类型——共享 EEClass

目前所有包含引用类型的实例都会共享同一个 EEClass 以及 MethodDesc。这种方式可行的原因在于，所有的引用所占用的内存大小都一样，例如 4 个字节或是 8 个字节，因此这些类型的布局都相同。上图示意了 List<object> 和 List<string> 的情形。那个规范的 MethodTable 会在第一个引用类型实例加载时自动创建，它里面包含了那些常用的、不局限于某个特定实例的数据，例如非虚的方法槽以及 RemotableMethodInfo。只包含值类型的实例不会共享信息，每一个实例化的类型都会有它自己独立的 EEClass。

已加载的泛型类型的 MethodTable 会被缓存在一个哈希表中，这个哈希表由加载它们的模块所持有。在一个新的实例构造之前，加载器会首先查询这个哈希表，这样就不会出现有同一个类型具有多个 MethodTable 实例的情况了。

更多关于泛型共享的细节，请参考 Design and Implementation of Generics for the .NET Common Language Runtime。