# Архитектура статической памяти и метаданных типов в среде выполнения.NET (CoreCLR): Глубокий анализ реализации версий 6, 7, 8 и 9

## 1. Введение: Парадигма статического хранения в управляемой среде

В экосистеме.NET концепция "статического" (static) выходит далеко за рамки синтаксического сахара, обозначающего принадлежность члена типу, а не экземпляру. С точки зрения архитектуры среды выполнения (CLR), статические данные представляют собой фундаментальный класс объектов, обладающих уникальными характеристиками жизненного цикла, аллокации и доступа. В отличие от традиционных нативных сред (например, C++), где статические переменные часто размещаются в сегменте данных исполняемого файла (.data или .bss), в управляемой среде.NET статические данные являются динамическими сущностями, создаваемыми, инициализируемыми и, в современных версиях, выгружаемыми по требованию среды выполнения.

Переход от классического.NET Framework к.NET Core, и далее к унифицированной платформе.NET 5+, ознаменовал радикальный сдвиг в управлении памятью. Отказ от доменов приложений (AppDomain) как основной границы изоляции в пользу контекстов загрузки сборок (AssemblyLoadContext) потребовал полной переработки механизмов хранения статических корней. В данном отчете мы детально рассмотрим физическую организацию памяти для статических данных и определений типов (MethodTable, EEClass), проанализируем эволюцию этих структур вплоть до.NET 9, и проведем критическое сравнение текущей реализации с моделями, описанными в фундаментальной работе Конрада Кокосы "Pro.NET Memory Management" (2018).

Исследование базируется на анализе исходного кода CoreCLR, данных отладчика SOS/LLDB и технической документации, охватывая как "горячие" (hot), так и "холодные" (cold) структуры данных, а также специализированные кучи загрузчика (Loader Heaps).

## 2. Анатомия типа: Физическое разделение метаданных

Для понимания того, где хранятся статические поля, необходимо сначала деконструировать механизмы хранения самих определений типов. В среде CLR тип не является монолитным блоком данных. В целях оптимизации производительности и локальности кэша процессора, информация о типе разделена на две ключевые структуры: MethodTable (MT) и EEClass.

### 2.1. MethodTable (MT): Оперативная проекция типа

MethodTable является основной структурой идентификации типа во время выполнения. Каждый объект в куче.NET (GC Heap) содержит заголовок, указывающий на соответствующую MethodTable. Эта структура спроектирована быть максимально компактной и содержать только те данные, которые критически необходимы для работы "горячих" путей исполнения: выделения памяти, динамического диспетчеризации методов и проверки типов (casting).1

#### 2.1.1. Локализация и жизненный цикл

MethodTable размещается в **HighFrequencyHeap** (Куча высокой частоты).2 Это стратегическое решение обусловлено тем, что доступ к MT происходит при каждом вызове виртуального метода и при каждой проверке типа оператором is или as. Размещение в "горячей" куче гарантирует, что структуры MT различных типов будут физически расположены рядом, минимизируя промахи кэша (cache misses) при интенсивной работе с объектами.

#### 2.1.2. Внутренняя структура и смещения

Анализ дампа памяти (через SOS команду !dumpmt) демонстрирует, что MethodTable содержит жестко оптимизированный набор полей 4:

* **Flags (4 байта):** Битовая маска, определяющая свойства типа (наличие финализатора, является ли тип значимым, содержит ли указатели).
* **Base Instance Size (4 байта):** Размер экземпляра объекта в байтах. Это значение используется аллокатором GC для быстрого выделения памяти в Generation 0 (bump pointer allocation).
* **EEClass Pointer (Pointer size):** Ссылка на "холодную" структуру EEClass.
* **Interface Map:** Ссылка на таблицу реализации интерфейсов.
* **VTable Slots:** Массив адресов точек входа (entry points) для виртуальных методов.

Важно отметить, что MethodTable *не содержит* имен методов, полных сигнатур или другой описательной метаинформации. Эти данные вынесены в EEClass, так как они не требуются для выполнения уже скомпилированного кода.

### 2.2. EEClass: Холодное хранилище метаданных

Структура EEClass (Execution Engine Class) содержит полную информацию о типе, необходимую для загрузчика типов (Type Loader), JIT-компилятора и подсистемы рефлексии. Поскольку эти операции происходят значительно реже, чем вызовы методов, EEClass вынесен в отдельную область памяти.

#### 2.2.1. Размещение в LowFrequencyHeap

EEClass хранится в **LowFrequencyHeap** (Куча низкой частоты).3 Эта куча оптимизирована под данные, доступ к которым происходит эпизодически. Разделение MT и EEClass позволяет CLR держать "горячие" таблицы методов в кэше процессора, не загрязняя его редко используемыми данными описания полей и иерархии наследования.

#### 2.2.2. Эволюция структуры

Начиная с ранних версий.NET Core, наблюдается тенденция к миграции данных из EEClass в MethodTable. Если в.NET Framework 4.x EEClass играл более активную роль, то в.NET 6/7/8 многие флаги и указатели были дублированы или перенесены в MT для устранения лишних уровней косвенности (indirection).1 Тем не менее, EEClass остается хранилищем для сложной логики компоновки полей (field layout) и иерархических связей.

### 2.3. Сравнение с "Pro.NET Memory Management"

В работе Конрада Кокосы (2018) описывается модель, актуальная для ранних версий.NET Core 2.x. С тех пор (в.NET 6-8) произошли следующие изменения:

1. **Уплотнение MethodTable:** Современная CLR использует более агрессивное сжатие флагов в заголовке MT.
2. **Generics Handling:** Механизм Canonical MethodTable для дженериков (обсуждается в разделе 5) стал более сложным, вводя дополнительные уровни абстракции, которые не были полностью описаны в издании 2018 года.
3. **Выгрузка (Unloading):** Связь между MT и LoaderAllocator стала критической в контексте AssemblyLoadContext (ALC), что добавило новые скрытые поля для отслеживания контекста загрузки.7

## 3. Кучи загрузчика (Loader Heaps): Физическая организация памяти

Статические данные не существуют в вакууме. Они аллоцируются внутри специализированных менеджеров памяти, называемых **Loader Heaps**. Понимание этих куч критично для диагностики утечек памяти, связанных со статикой, так как они, в отличие от GC Heap, управляются вручную средой CLR и не подвергаются компактной сборке мусора.

### 3.1. Классификация куч загрузчика

Согласно внутренним спецификациям CLR и данным отладки, существует несколько типов куч, каждая из которых имеет свое назначение 2:

| **Тип кучи (Heap Identifier)** | **Назначение и содержимое** | **Типичный размер блока** | **Примечание** |
| --- | --- | --- | --- |
| **HighFrequencyHeap** | Хранит критически важные структуры для выполнения кода: MethodTable, InterfaceMap, FieldDesc (описатели полей). Здесь же часто размещаются слоты для статических полей примитивных типов. | 64 КБ | Не подвергается уплотнению. Высокая плотность доступа. |
| **LowFrequencyHeap** | Хранит "холодные" данные: EEClass, таблицы поиска имен, метаданные загрузчика. | 64 КБ | Доступ происходит преимущественно при JIT-компиляции и Reflection. |
| **StubHeap** | Содержит переходники кода (thunks) для P/Invoke, COM-интеропа, вызовов виртуальных методов (VSD) и CAS (Code Access Security - исторически). | Переменный | Критичен для маршалинга. |
| **LoaderCodeHeap** | Содержит нативный код, сгенерированный JIT-компилятором (Executable Memory). | Переменный | Единственная куча с правами PAGE\_EXECUTE\_READ. |
| **IndirectionCellHeap** | Хранит ячейки косвенности для доступа к статическим полям через границы сборок и для дженериков. | Мелкие блоки | Важен для Shared Generics. |

### 3.2. LoaderAllocator: Контекст владения памятью

В.NET Core/5+ концепция глобальных куч загрузчика трансформировалась. Теперь каждый AssemblyLoadContext (ALC) имеет свой собственный экземпляр LoaderAllocator.

* **Изоляция:** MethodTable и статические поля типов, загруженных в конкретный ALC, аллоцируются в кучах, принадлежащих *этому* LoaderAllocator.
* **Жизненный цикл:** LoaderAllocator удерживается живым до тех пор, пока жив сам ALC. Это создает механизм выгрузки: когда ALC становится недостижимым, CLR освобождает весь набор куч (HighFrequency, Low и т.д.), связанных с этим аллокатором.9
* **Связь с GC:** LoaderAllocator регистрирует себя в GC через специальные дескрипторы (LoaderAllocatorScout), чтобы отслеживать момент, когда можно безопасно освободить нативную память.11

**Инсайт:** Именно здесь кроется основная причина утечек памяти в современных.NET приложениях. Если статический делегат или событие из Default ALC (глобального) подпишется на объект из Collectible ALC (выгружаемого), это создаст сильную ссылку на объект. Объект держит ссылку на свой Тип (MethodTable), Тип держит ссылку на LoaderAllocator, а LoaderAllocator удерживает все статические данные и скомпилированный код ALC. Таким образом, одна забытая подписка предотвращает выгрузку гигабайтов памяти.10

## 4. Статические поля: Механизмы хранения и доступ

Вопрос "где хранится статическое поле" имеет два разных ответа в зависимости от того, является ли поле ссылочным (Reference Type) или значимым (Value Type).

### 4.1. Статические поля значимых типов (Value Type Statics)

Для полей примитивных типов (int, double, bool) и структур (struct) CLR выделяет память непосредственно внутри структур, управляемых LoaderAllocator.

* **Размещение:** При загрузке типа CLR рассчитывает общий размер всех статических значимых полей. Этот блок памяти часто аллоцируется непосредственно в **HighFrequencyHeap** или в специальном массиве, на который указывает MethodTable.
* **Инициализация:** Гарантируется, что память обнулена перед доступом. Статический конструктор (.cctor) вызывается перед первым обращением к любому полю.
* **Boxing:** Статические структуры *не упаковываются* (not boxed). Они лежат в памяти "как есть". Это отличает их от структур, приведенных к интерфейсу или типу object.

### 4.2. Статические поля ссылочных типов (Reference Type Statics)

Здесь механизм сложнее, так как статические поля являются корнями для сборщика мусора (GC Roots).

* **Разделение хранения:**
  1. **Переменная (Slot):** Сама переменная, хранящая адрес объекта (ссылка размером 4 или 8 байт), находится в системном массиве, называемом **GC Handle Table** (или в специализированных массивах внутри LoaderAllocator для выгружаемых контекстов).12
  2. **Объект (Instance):** Сам объект (new MyClass(), new List<int>()), на который указывает статическое поле, всегда находится в **GC Heap** (SOH, LOH или POH). Статическое поле — это лишь "указатель", который говорит GC: "не удаляй этот объект".
* Indirection Cells (Ячейки косвенности):  
  В.NET Core доступ к статическим полям часто происходит не напрямую по жестко прописанному адресу, а через уровень косвенности. JIT-компилятор генерирует код, который читает адрес статики из специальной ячейки. Это необходимо для поддержки выгрузки сборок (адрес может измениться или стать невалидным) и для дженериков (см. раздел 5).14

### 4.3. Pinned Object Heap (POH) и статическая память

В.NET 5 был введен **Pinned Object Heap**.15 Это имеет прямое отношение к статическим полям типа static readonly byte, которые часто используются как буферы для ввода-вывода или интеропа.

* **Проблема:** Ранее, если статический массив закреплялся (pinning) для передачи в нативный код, он фрагментировал кучу (SOH/LOH), так как GC не мог его переместить.
* **Решение:** Теперь такие массивы можно аллоцировать в POH (GC.AllocateArray(..., pinned: true)). Поскольку статические поля живут долго, размещение их буферов в POH (или LOH) является архитектурно правильным решением.

### 4.4. Frozen Heap в.NET 8

В.NET 8 и 9 развита концепция **Frozen Heap** (или Non-GC Heap).17

* **Строковые литералы:** Статические строковые литералы теперь могут размещаться в специальном сегменте памяти, который вообще игнорируется сборщиком мусора.
* **Оптимизация статических полей:** Если JIT видит static readonly поле с неизменяемым контентом (например, массив байт, инициализированный константами), он может "запечь" эти данные прямо в сегмент данных или Frozen Heap, полностью избегая аллокации в куче GC. Это видно на примере оптимизации SearchValues<T> и преобразований Hex-строк в.NET 8/9.18

## 5. Дженерики (Generics) и сложность статического хранения

Статические поля в обобщенных типах (Generics) представляют собой наибольшую сложность для реализации CLR. Спецификация ECMA-335 требует, чтобы MyClass<int>.Field и MyClass<string>.Field были разными физическими полями.

### 5.1. Канонические типы (\_\_Canon)

Для ссылочных типов CLR использует оптимизацию **Code Sharing**. Вместо того чтобы генерировать отдельный код для List<string>, List<object>, List<MyType>, CLR генерирует единую версию кода для специального внутреннего типа **System.\_\_Canon**.19

* **Проблема:** Код один (\_\_Canon), а статические поля должны быть разными для каждого T.
* **Решение (Generic Dictionaries):** MethodTable каждого конкретного дженерика (List<string>) содержит указатель на структуру **Generic Dictionary**.

### 5.2. Реализация доступа к статике в Shared Code

Когда JIT компилирует метод для \_\_Canon, он не может знать адрес статического поля, так как он зависит от конкретного T.

1. **Hidden Argument:** В методы shared generics передается скрытый аргумент (часто в регистре или стеке), представляющий "контекст исполнения" (MethodTable или MethodDesc конкретного типа).
2. **Runtime Lookup:** Код содержит вызов хелпера (например, JIT\_GetSharedNonGCStaticBase), который использует переданный контекст и словарь дженерика для поиска адреса статического поля.20

**Аналитический инсайт:** Это объясняет, почему доступ к статическим полям в дженериках медленнее, чем в обычных типах. Требуется lookup во время выполнения. Однако, в **.NET 9** (как указано в 21) внедрена оптимизация, позволяющая инлайнить (встраивать) этот lookup, если T известен или предсказуем, значительно ускоряя доступ.

## 6. Статические делегаты и лямбда-выражения: Кэширование и Roslyn

Механизм работы статических лямбд и делегатов претерпел значительные изменения, диктуемые компилятором Roslyn.

### 6.1. Эволюция кэширования

* **До C# 5:** Лямбды компилировались в статические методы, но экземпляр делегата создавался при каждом вызове.
* **Современный подход (Roslyn):** Используется стратегия агрессивного кэширования. Компилятор создает вспомогательный класс (часто именуемый <>c), содержащий:
  + Статическое поле-синглтон для самого класса (public static readonly <>c <>9).
  + Статические поля для кэширования каждого делегата (public static Action <>9\_\_1\_0).  
    Это гарантирует, что аллокация делегата происходит только один раз за время жизни приложения (singleton per type).22

### 6.2. Static Lambdas (C# 9+)

Введение модификатора static для лямбд (static x => x \* x) является инструментом контроля. Оно не меняет фундаментально способ хранения (он и так был оптимизирован), но запрещает захват контекста this или локальных переменных, гарантируя, что компилятор сможет использовать наиболее эффективную стратегию кэширования в статической памяти ("<>c" класс), исключая создание замыканий (Closure Objects) в куче.23

### 6.3. Dynamic PGO в.NET 7/8

С внедрением **Dynamic PGO** (Profile-Guided Optimization) JIT получил возможность видеть, что статический делегат (например, в LINQ запросе) всегда указывает на один и тот же метод. Это позволяет JIT-компилятору **девиртуализировать** вызов делегата и заменить его прямым вызовом метода (или инлайнингом), полностью устраняя накладные расходы на обращение к памяти делегата.25

## 7. Сравнительный анализ с "Управление памятью в.NET" (Konrad Kokosa)

Книга Конрада Кокосы остается "золотым стандартом", но с момента ее выхода (2018) произошли архитектурные сдвиги, которые необходимо учитывать инженеру.

| **Аспект** | **Описание в книге (состояние ~2018)** | **Реальность.NET 6/7/8/9** | **Практическое значение** |
| --- | --- | --- | --- |
| **Изоляция** | Акцент на AppDomain как жесткой границе памяти и выгрузки. | AppDomain один (в Core). Изоляция через AssemblyLoadContext. | Статика теперь выгружаема, но требует дисциплины ссылок (LoaderAllocatorScout). |
| **Кучи** | Базовое описание Loader Heaps. | Детализация на POH, Frozen Heap. | Появление POH позволяет эффективно использовать static буферы без пиннинга в LOH. |
| **Generics** | Описание Generic Sharing. | Введение \_\_Canon lookup inlining (NET 9) и сложностей с UnmanagedCallersOnly.26 | Производительность дженерик-статики значительно выросла. |
| **Диагностика** | SOS !dumpheap, !do. | Новые команды SOS для ALC, улучшенная поддержка dotnet-dump. | Диагностика утечек LoaderAllocator стала сложнее, требуя анализа графа LoaderAllocatorScout.10 |
| **JIT** | Tiered Compilation как новинка. | Dynamic PGO, OSR (On-Stack Replacement). | Статические поля могут быть "constant-folded" (заменены на константы) агрессивнее благодаря PGO. |

## 8. Заключение

Современная реализация статической памяти в.NET CLR представляет собой высокооптимизированную, многоуровневую систему. Статические данные не просто "лежат в памяти"; они распределены между горячими (HighFrequencyHeap) и холодными (LowFrequencyHeap) регионами, управляются контекстно-зависимыми аллокаторами (LoaderAllocator) и подвергаются агрессивным оптимизациям со стороны JIT и компилятора (PGO, Frozen Heap).

Ключевые выводы для разработчика:

1. **Локальность:** Статические поля значимых типов обладают отличной локальностью данных (HighFrequencyHeap).
2. **Стоимость абстракций:** Доступ к статике в Generic-типах имеет накладные расходы на lookup, хотя они снижаются в новых версиях.NET.
3. **Опасность утечек:** В мире AssemblyLoadContext статическое поле — это потенциальная причина невозможности выгрузки целого плагина или модуля.
4. **Эволюция:** Механизмы, описанные в литературе 5-летней давности, требуют корректировки с учетом POH, Frozen Heap и Dynamic PGO.

Понимание этих внутренних структур (MethodTable, LoaderAllocator, \_\_Canon) позволяет писать код, который работает в гармонии с средой выполнения, избегая скрытых аллокаций и утечек памяти в высоконагруженных системах.

#### Источники

1. .NET Framework Internals: How the CLR Creates Runtime Objects | Microsoft Learn, дата последнего обращения: ноября 27, 2025, <https://learn.microsoft.com/en-us/archive/msdn-magazine/2005/may/net-framework-internals-how-the-clr-creates-runtime-objects>
2. High Frequency Heap - Stack Overflow, дата последнего обращения: ноября 27, 2025, <https://stackoverflow.com/questions/4405627/high-frequency-heap>
3. c# - Heap Memory Management .Net - Stack Overflow, дата последнего обращения: ноября 27, 2025, <https://stackoverflow.com/questions/8479529/heap-memory-management-net>
4. What does fields in method table mean in this example? - Stack Overflow, дата последнего обращения: ноября 27, 2025, <https://stackoverflow.com/questions/4104099/what-does-fields-in-method-table-mean-in-this-example>
5. Understanding the .NET GC - Pro .NET Memory Management, дата последнего обращения: ноября 27, 2025, <https://prodotnetmemory.com/slides/UnderstadingGC/>
6. SimpleReader Mobile: 'Performance is a Feature!', дата последнего обращения: ноября 27, 2025, [http://m.simplepie.org/?feed=mattwarren.org%2Fatom.xml](http://m.simplepie.org/?feed=mattwarren.org/atom.xml)
7. JIT: Continuation allocation logic can be improved · Issue #121088 · dotnet/runtime - GitHub, дата последнего обращения: ноября 27, 2025, <https://github.com/dotnet/runtime/issues/121088>
8. Large number of memfd:doublemapper (deleted) entries · Issue #89776 · dotnet/runtime, дата последнего обращения: ноября 27, 2025, <https://github.com/dotnet/runtime/issues/89776>
9. How to use and debug assembly unloadability in .NET - Microsoft Learn, дата последнего обращения: ноября 27, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/assembly/unloadability>
10. Add features to help diagnose unloading issues · Issue #11157 · dotnet/runtime - GitHub, дата последнего обращения: ноября 27, 2025, <https://github.com/dotnet/runtime/issues/11157>
11. Assemblies cannot be modified after loading AssemblyLoadContext is garbage collected · Issue #35342 · dotnet/runtime - GitHub, дата последнего обращения: ноября 27, 2025, <https://github.com/dotnet/runtime/issues/35342>
12. Static class memory allocation where it is stored C# - Stack Overflow, дата последнего обращения: ноября 27, 2025, <https://stackoverflow.com/questions/33562199/static-class-memory-allocation-where-it-is-stored-c-sharp>
13. .NET Type Internals from a Microsoft CLR Perspective — Part 1 | by Adityanand Pasumarthi | Medium, дата последнего обращения: ноября 27, 2025, <https://medium.com/@apasumarthi1999/net-type-internals-from-a-microsoft-clr-perspective-part-1-99a7c55a8342>
14. runtime/src/coreclr/vm/virtualcallstub.h at main - GitHub, дата последнего обращения: ноября 27, 2025, <https://github.com/dotnet/runtime/blob/main/src/coreclr/vm/virtualcallstub.h>
15. Optimize ASP.NET Core memory with DATAS - Thinktecture AG, дата последнего обращения: ноября 27, 2025, <https://www.thinktecture.com/en/net/optimize-asp-net-core-memory-with-datas/>
16. Pinned Object Heap in .NET 5 - TooSlowException, дата последнего обращения: ноября 27, 2025, <https://tooslowexception.com/pinned-object-heap-in-net-5/>
17. Performance Improvements in .NET 9 - Microsoft Developer Blogs, дата последнего обращения: ноября 27, 2025, <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-9/>
18. Performance Improvements in .NET 5 - Microsoft Developer Blogs, дата последнего обращения: ноября 27, 2025, <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/>
19. c# - Strange System.\_\_Canon exception - Stack Overflow, дата последнего обращения: ноября 27, 2025, <https://stackoverflow.com/questions/16854393/strange-system-canon-exception>
20. It doesn't support keyword Lock when debugging the clrcore by visual studio · Issue #5909 · dotnet/runtime - GitHub, дата последнего обращения: ноября 27, 2025, <https://github.com/dotnet/runtime/issues/5909>
21. DotNET 9 | PDF - Scribd, дата последнего обращения: ноября 27, 2025, <https://www.scribd.com/document/810482693/DotNET-9>
22. Why has a lambda with no capture changed from a static in C# 5 to an instance method in C# 6? - Stack Overflow, дата последнего обращения: ноября 27, 2025, <https://stackoverflow.com/questions/33948144/why-has-a-lambda-with-no-capture-changed-from-a-static-in-c-sharp-5-to-an-instan>
23. static modifier - C# reference - Microsoft Learn, дата последнего обращения: ноября 27, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static>
24. c# - How to create a static lambda for use with expression building? - Stack Overflow, дата последнего обращения: ноября 27, 2025, <https://stackoverflow.com/questions/43217853/how-to-create-a-static-lambda-for-use-with-expression-building>
25. Performance Improvements in .NET 7 - Microsoft Developer Blogs, дата последнего обращения: ноября 27, 2025, <https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_7/>
26. Allow [UnmanagedCallersOnly] on generic methods, when possible #55144 - GitHub, дата последнего обращения: ноября 27, 2025, <https://github.com/dotnet/runtime/issues/55144>