# Углубленный анализ архитектуры кэширования и стратегий вытеснения данных в среде.NET Enterprise

## 1. Введение: Смена парадигмы — от игрового цикла к жизненному циклу запроса

Переход от разработки на Unity к созданию высоконагруженных бэкенд-систем на платформе.NET Core требует не просто изучения новых API, но и фундаментальной перестройки ментальной модели управления ресурсами. В среде GameDev, где вы работали ранее, доминирующим паттерном является игровой цикл (Game Loop), жестко ограниченный временным бюджетом кадра — обычно 16.6 миллисекунд для достижения стабильных 60 кадров в секунду. В этом контексте стратегия оптимизации памяти сводится к принципу "Zero Allocation in Update": разработчик обязан избегать аллокаций в куче (Heap) во время выполнения кадра, чтобы предотвратить срабатывание Garbage Collector (GC), которое может вызвать визуальные "фризы" или рывки изображения.1

В архитектуре Enterprise-бэкенда (ASP.NET Core) парадигма кардинально меняется. Здесь мы оперируем не кадрами, а пропускной способностью (Throughput), измеряемой в тысячах запросов в секунду (RPS), и латентностью (Latency) обработки каждого отдельного запроса. Хотя одна аллокация памяти может казаться незначительной, кумулятивное давление тысяч конкурентных запросов создает нагрузку на GC, способную перевести его в режим полной остановки мира (Stop-the-World) при сборке второго поколения (Gen 2). Это явление аналогично "зависанию" игры, но в контексте сервера оно парализует работу для всех подключенных пользователей одновременно, что недопустимо в критических бизнес-системах.1

Четвертый день четвертой недели нашего учебного плана посвящен **алгоритмам кэширования и управлению вытеснением данных**. Кэширование является наиболее эффективным механизмом повышения пропускной способности на чтение, однако оно привносит сложные проблемы, связанные с согласованностью данных, управлением памятью и потокобезопасностью. В отличие от однопоточной природы скриптов Unity (или строго контролируемой Job System), кэши бэкенда функционируют в агрессивной многопоточной среде, где распространены состояния гонки (Race Conditions) и эффекты "тромбообразования" (Thundering Herd). В данном документе мы проведем глубокий теоретический анализ алгоритмов LRU (Least Recently Used), разберем внутреннее устройство стандартного IMemoryCache в.NET, изучим влияние архитектуры процессора на производительность коллекций и рассмотрим вероятностные структуры данных, такие как фильтры Блума, необходимые для защиты от атак на кэш.

## 2. Теоретический фундамент: Иерархия памяти и аппаратные ограничения

Прежде чем переходить к обсуждению программных алгоритмов кэширования, необходимо понять физическую реальность работы современного оборудования, которая делает кэширование неизбежным. Разрыв в скорости доступа между регистрами процессора и сетевым вводом-выводом является логарифмическим и определяет всю архитектуру высокопроизводительных систем.

### 2.1 Латентность доступа к данным: Числа, которые обязан знать архитектор

Понимание стоимости операций ввода-вывода является базовым навыком для Performance Engineer. Рассмотрим иерархию задержек в тактах процессора и наносекундах:

* **L1 Cache Reference:** ~0.5 наносекунд. Доступ к данным, находящимся непосредственно в кэше первого уровня ядра.
* **L2 Cache Reference:** ~7 наносекунд.
* **Main Memory (RAM) Reference:** ~100 наносекунд. Обращение к оперативной памяти уже в 200 раз медленнее, чем к L1.
* **SSD Random Read:** ~150,000 наносекунд.
* **Network Round Trip (Datacenter):** ~500,000 наносекунд.

Когда бэкенд-сервис запрашивает данные из базы данных (сетевой запрос + дисковый ввод-вывод), эта операция примерно в **5,000–10,000 раз медленнее**, чем чтение тех же данных из оперативной памяти.2 Кэширование — это мост, соединяющий эти несопоставимые величины. Однако, внедряя кэш, мы сталкиваемся с одной из двух самых сложных проблем в компьютерных науках (помимо именования переменных) — инвалидацией кэша.

### 2.2 Пространственная и Временная локальность (Spatial vs. Temporal Locality)

В разработке на Unity вы привыкли оптимизировать код под **Пространственную локальность (Spatial Locality)**. Это означает размещение данных (например, компонентов сущностей в ECS или частиц) в памяти непрерывными блоками. Когда процессор загружает одну переменную, он автоматически подгружает в кэш-линию (обычно 64 байта) и соседние данные. Именно поэтому массивы структур (struct) в Unity предпочтительнее массивов классов: они обеспечивают последовательный доступ к памяти и минимизируют промахи кэша (Cache Misses).1

В бэкенд-кэшировании мы, в первую очередь, работаем с **Временной локальностью (Temporal Locality)** — принципом, гласящим, что если к элементу данных было обращение, то с высокой вероятностью к нему обратятся снова в ближайшее время. Алгоритмы вытеснения, такие как LRU (Least Recently Used) и LFU (Least Frequently Used), являются прямыми реализациями эвристик временной локальности.4

Тем не менее, *реализация* этих алгоритмов должна учитывать и пространственную локальность, чтобы оставаться эффективной под высокой нагрузкой. Наивная реализация LRU с использованием стандартного связного списка LinkedList<T> в.NET может страдать от серьезной фрагментации памяти ("pointer chasing"), вызывая постоянные промахи кэша CPU, что может нивелировать выигрыш от самого кэширования.5

## 3. Глубокое погружение: Алгоритм Least Recently Used (LRU)

Алгоритм Least Recently Used (LRU) является "золотым стандартом" стратегий вытеснения кэша в сценариях, где доступ к данным не является абсолютно случайным. Он работает на предпосылке, что данные, к которым обращались последними, наиболее важны для сохранения.

### 3.1 Анатомия классического LRU

Корректная реализация LRU-кэша должна поддерживать две операции с асимптотической сложностью $O(1)$:

1. **Get(key):** Получить значение и пометить его как "наиболее недавно использованное" (переместить в начало очереди).
2. **Put(key, value):** Вставить новое значение. Если кэш переполнен, удалить "наименее недавно использованный" элемент (из конца очереди).

Для достижения сложности $O(1)$ как для поиска, так и для упорядочивания, стандартная реализация комбинирует две структуры данных 6:

* **Двусвязный список (Doubly Linked List):** Поддерживает порядок использования. "Голова" списка (Head) представляет самый свежий элемент, а "Хвост" (Tail) — кандидат на удаление.
* **Хеш-таблица (Dictionary):** Обеспечивает мгновенный доступ к **узлам** связного списка по ключу. Важно отметить, что словарь хранит не просто значение TValue, а ссылку на LinkedListNode<K, V>.

#### Поток выполнения операции Get

Когда вызывается Get(key):

1. Словарь выполняет поиск ключа.
2. Если ключ найден, возвращается ссылка на объект узла LinkedListNode.
3. Алгоритм "отцепляет" узел от его текущей позиции в списке (изменяя ссылки node.Prev.Next и node.Next.Prev).
4. Алгоритм прикрепляет узел к "Голове" списка (Head).
5. Возвращается полезная нагрузка (Value).

#### Поток выполнения операции Put

Когда вызывается Put(key, value):

1. Если ключ уже существует, обновляется значение, и узел перемещается в "Голову".
2. Если ключ новый:
   * Создается новый объект узла.
   * Узел добавляется в "Голову".
   * Узел добавляется в словарь.
   * **Проверка переполнения:** Если Count > Capacity:
     + Берется узел из "Хвоста" (Tail).
     + Ключ этого узла удаляется из словаря.
     + Сам узел удаляется из списка.

### 3.2 Цена абстракций: Проблемы производительности LinkedList в.NET

Для разработчика Unity, привыкшего к паттернам "Zero Allocation", стандартный класс System.Collections.Generic.LinkedList<T> в.NET представляет собой проблему в высоконагруженных сценариях.

Каждый раз, когда вы добавляете элемент в LinkedList<T>, среда CLR аллоцирует новый объект LinkedListNode<T> в куче (Heap).

* **Накладные расходы памяти:** Объект LinkedListNode<T> содержит само значение T, ссылку на Next (8 байт на 64-битной системе), ссылку на Previous (8 байт), ссылку на владеющий список List (8 байт) и стандартный заголовок объекта (16 байт). Для кэширования небольшого числа (например, int или Guid), накладные расходы могут превышать размер полезных данных в 4-5 раз.1
* **Фрагментация памяти:** Эти узлы аллоцируются в куче индивидуально. Со временем они оказываются разбросаны по разным страницам памяти. Обход такого списка (или переход к узлу из словаря) заставляет процессор обращаться к хаотичным адресам памяти, что приводит к **CPU Cache Misses**. Процессор вынужден простаивать сотни тактов, ожидая подгрузки данных из RAM.5

### 3.3 Оптимизация: "Zero-Allocation" LRU на базе массивов

Чтобы спроектировать "High-Performance" LRU-кэш, подходящий для жестких требований по памяти (аналогично ограничениям игровых консолей), мы можем заменить стандартный LinkedList на **структурный связный список поверх массива (Array-Backed Linked List)**. Эта техника симулирует указатели, используя целочисленные индексы в заранее выделенном массиве.1

#### Архитектурный дизайн

Вместо создания миллионов объектов LinkedListNode<T>, мы аллоцируем один большой массив структур Entry:

C#

struct CacheEntry<TKey, TValue>  
{  
 public TKey Key;  
 public TValue Value;  
 public int NextIndex; // "Указатель" на следующий элемент  
 public int PrevIndex; // "Указатель" на предыдущий элемент  
 public int HashCode; // Кэшированный хеш для быстрого сравнения  
}

Мы используем ArrayPool<T> или фиксированный массив для хранения этих записей.1

* **Пространственная локальность:** Все узлы расположены в памяти непрерывно. Процессор может эффективно использовать механизм предвыборки (prefetching) кэш-линий.9
* **Отсутствие аллокаций (Zero Allocation):** Добавление элемента подразумевает запись данных в существующую ячейку массива по индексу, а не создание нового экземпляра класса. GC не испытывает нагрузки, так как для него существует только один объект — сам массив.
* **Управление индексами:** Мы должны реализовать собственный менеджер свободных слотов (Free List) внутри этого же массива, чтобы переиспользовать индексы удаленных элементов.

Этот подход идеально согласуется с философией "Data-Oriented Design" (DOD), доминирующей в Unity, но он также крайне эффективен в высокочастотном трейдинге и высоконагруженных бэкенд-системах, где паузы GC недопустимы.10

### 3.4 Стратегии конкурентности для LRU (Concurrency Strategies)

В отличие от локального состояния игры, кэш бэкенда доступен сотням потоков одновременно. Синхронизация LRU сложна тем, что каждая операция Read (Get) де-факто является операцией Write для внутреннего состояния (перемещение узла в начало списка).7

#### Стратегия A: Крупномасштабная блокировка (Coarse-Grained Locking)

Самый простой подход — обернуть каждый метод в lock (монитор):

C#

lock (\_syncRoot) {  
 // поиск в словаре, перелинковка узлов, возврат значения  
}

* **Преимущества:** Простота реализации, легкость верификации корректности.
* **Недостатки:** Высокая конкуренция (Contention). Под нагрузкой потоки проводят больше времени в ожидании блокировки, чем выполняя полезную работу. Это сериализует доступ к кэшу, сводя на нет преимущества многоядерных процессоров.11

#### Стратегия B: ReaderWriterLockSlim

Этот примитив синхронизации позволяет множеству потоков читать одновременно, но требует эксклюзивного доступа для записи.

* **Ловушка:** В классическом LRU **каждое чтение меняет состояние**. Следовательно, операция Get требует эксклюзивной блокировки на запись (EnterWriteLock), что превращает ReaderWriterLockSlim в обычный тяжелый эксклюзивный лок. Попытка использовать EnterUpgradeableReadLock и повышать блокировку до записи только при необходимости ("Lock Promotion") является дорогостоящей операцией и часто приводит к дедлокам, если не реализована идеально.12
* **Инсайт:** ReaderWriterLockSlim обычно *медленнее* простого lock для очень коротких критических секций из-за сложности внутреннего устройства и накладных расходов на ведение учета читателей.11 Для LRU, где операции сводятся к перестановке нескольких целочисленных индексов (наносекунды), простой lock или SpinLock часто выигрывают в производительности.

#### Стратегия C: Шардинг (Sharding/Partitioning)

По аналогии с устройством ConcurrentDictionary, мы можем создать $N$ независимых экземпляров LRU (шардов) и маршрутизировать ключи в нужный шард, используя GetHashCode() % N.

* **Преимущество:** Конкуренция снижается в $N$ раз. Если у нас 16 ядер и 32 шарда, вероятность того, что два потока столкнутся на одном замке, значительно снижается. Это предпочтительный метод для масштабирования.15

#### Стратегия D: Аппроксимированный LRU (Lazy Promotion)

Чтобы избежать захвата блокировки на запись при каждом чтении, можно ослабить строгость LRU.

* **Алгоритм:** При Get, перемещать узел в начало списка только в том случае, если он находится "достаточно далеко" от начала (например, во второй половине списка) или если с момента последнего перемещения прошло $X$ секунд.
* **Преимущество:** Большинство операций Get становятся действительно операциями только для чтения (Read-Only), что позволяет эффективно использовать ReaderWriterLockSlim в режиме EnterReadLock для основной массы трафика.7

## 4. Архитектурный анализ.NET IMemoryCache

Интерфейс Microsoft.Extensions.Caching.Memory.IMemoryCache является стандартом де-факто для внутрипроцессного кэширования в ASP.NET Core. Критически важно понимать, что его стандартная реализация **не является** строгим LRU. Она использует сложный набор эвристик для управления вытеснением.2

### 4.1 Внутренняя структура данных

Внутри MemoryCache использует ConcurrentDictionary<object, CacheEntry> для хранения данных. Это обеспечивает потокобезопасность и высокую скорость доступа без глобальной блокировки. Однако ConcurrentDictionary не поддерживает порядок элементов. Для реализации вытеснения MemoryCache полагается на вспомогательные структуры и фоновые процессы.

### 4.2 Механизм экспирации: "Сканирование" против "Таймеров"

Распространенным заблуждением является мнение, что каждый элемент в MemoryCache имеет свой собственный System.Threading.Timer, который срабатывает при истечении срока жизни. Это было бы крайне расточительно (миллионы таймеров убили бы планировщик потоков).

Вместо этого MemoryCache использует Пассивно-Активную стратегию экспирации:

1. **Пассивная проверка:** Когда вы пытаетесь получить элемент через Get, кэш проверяет, не истек ли его срок. Если истек — элемент удаляется, и возвращается null.6
2. **Активное сканирование (Expiration Scan):** Фоновый процесс периодически сканирует кэш для удаления просроченных элементов, к которым давно не было обращений.
   * Частота этого сканирования определяется опцией ExpirationScanFrequency (по умолчанию: 1 минута).17
   * **Следствие:** Просроченный элемент может оставаться в памяти ("зомби-объект") до 1 минуты (или дольше, если поток сканирования занят), занимая ценные ресурсы. Это поведение кардинально отличается от детерминированного освобождения ресурсов в Unity.

### 4.3 Политики вытеснения и Компактификация (Compaction)

Когда кэш достигает предела размера (SizeLimit), он должен освободить место. Этот процесс называется Компактификацией (Compaction).

Процесс компактификации в MemoryCache работает следующим образом:

1. **Триггер:** Срабатывает при превышении SizeLimit.
2. **Алгоритм:**
   * Кэш не удаляет элементы по одному. Он стремится удалить сразу пачку элементов (по умолчанию 5%), чтобы не запускать дорогостоящую компактификацию на каждой вставке.19
   * Сначала удаляются все **просроченные** (Expired) элементы.
   * Если места все еще недостаточно, удаляются элементы с наименьшим **Приоритетом** (CacheItemPriority).
   * Внутри одного приоритета используется эвристика **LRU** (Last Accessed Time).16

**Критический инсайт для High-Load:** Процесс компактификации является вычислительно дорогим. Поскольку ConcurrentDictionary не упорядочен, для нахождения кандидатов на удаление (LRU) необходимо просканировать коллекцию или поддерживать вторичный индекс. MemoryCache не поддерживает идеальный связный список (для экономии памяти и избежания блокировок), поэтому вытеснение является аппроксимацией. Под экстремальной нагрузкой процесс компактификации может вызывать скачки потребления CPU и задержки в обработке запросов.21

### 4.4 Дилемма "Размера" (Size Limit)

IMemoryCache не умеет магическим образом определять размер ваших объектов в байтах. Свойство Size в MemoryCacheEntryOptions — это безразмерное целое число, задаваемое разработчиком.6

* **Ответственность разработчика:** Вы должны вручную вычислять или оценивать размер (например, Size = 1 для лимита по количеству элементов, или Size = string.Length \* 2 для кэширования текста).
* **Риск:** Если свойство Size не установлено при добавлении элемента, общий SizeLimit кэша **игнорируется** для этого элемента. Кэш может расти бесконтрольно, пока не исчерпает всю оперативную память контейнера, что приведет к OutOfMemoryException или принудительному завершению процесса оркестратором (OOM Kill в Kubernetes).22 Это поведение отличается от "Soft References" в Java, где GC может автоматически очистить кэш под давлением памяти. В.NET Core MemoryCache удерживает **сильные ссылки (Strong References)** по умолчанию.

## 5. Вероятностные структуры данных: Фильтры Блума

В сценариях, где кэш используется для защиты медленной базы данных (паттерн Cache-Aside), мы сталкиваемся с риском **Cache Penetration** (Пробитие кэша). Это происходит, когда злоумышленник (или ошибка в коде) массово запрашивает ключи, которых **не существует** в базе данных.

1. Кэш проверяется -> Промах (Key not found).
2. База данных опрашивается -> Пустой результат.
3. Пустой результат (null) часто не кэшируется (или кэшируется на очень короткое время).
4. Повторение процесса 10,000 раз в секунду. База данных падает под нагрузкой от запросов несуществующих ключей.

**Фильтр Блума (Bloom Filter)** — это стандартный архитектурный щит для защиты от подобных сценариев.23

### 5.1 Алгоритмическая теория

Фильтр Блума — это компактная вероятностная структура данных, которая отвечает на вопрос: "Принадлежит ли этот элемент множеству?"

Ответ может быть:

* **"Нет"** (Точно нет). Если фильтр говорит "нет", мы можем смело возвращать 404, не обращаясь к базе данных.
* **"Возможно"** (Вероятно да). Существует небольшая вероятность ложноположительного срабатывания (False Positive), но ложноотрицательные (False Negative) невозможны.

### 5.2 Внутренняя механика

Структура состоит из битового массива размером $m$ бит и $k$ независимых хеш-функций.

1. **Add(item):** Вычислить $k$ хешей для элемента. Установить биты по полученным индексам в 1.
2. **Check(item):** Вычислить те же $k$ хешей. Проверить, равны ли все биты по этим индексам 1.
   * Если хотя бы один бит равен 0 -> Элемент точно отсутствует.
   * Если все биты равны 1 -> Элемент, вероятно, присутствует (биты могли быть установлены другими элементами при коллизии).

### 5.3 Математическая оптимизация

Для проектирования фильтра Блума в.NET бэкенде мы должны выбрать размер массива $m$ и количество хешей $k$, исходя из ожидаемого количества элементов $n$ и допустимой вероятности ошибки $p$.

Оптимальное количество хеш-функций $k$ определяется формулой 25:

$$k = \frac{m}{n} \ln 2$$

Вероятность ложноположительного срабатывания $p$ приблизительно равна:

$$p \approx \left( 1 - e^{-kn/m} \right)^k$$

Практический пример:

Чтобы хранить информацию о наличии 1 миллиона пользователей ($n=10^6$) с вероятностью ошибки 1% ($p=0.01$):

* Битов на элемент ($m/n$): ~9.6 бит.
* Общий объем памяти: ~1.2 МБ.
* Количество хеш-функций ($k$): 7.

Это фундаментально эффективнее, чем хранение HashSet<string> с миллионом ключей, который занял бы десятки мегабайт и нагружал бы GC проверками ссылок.

### 5.4 Реализация в.NET: BitArray против Span

Хотя в.NET существует класс System.Collections.BitArray, высокопроизводительные реализации часто используют массив long и битовые операции (|, &, <<) вручную.

* **Эффективность кэша CPU:** long располагается в памяти непрерывно. Проверка $k$ битов включает случайный доступ внутри этого массива. Если массив помещается в L3 кэш процессора (например, 1-2 МБ), проверка выполняется молниеносно.27
* Хеширование: Вместо вычисления $k$ различных дорогих криптографических хешей, мы можем использовать технику Двойного хеширования (Double Hashing):  
    
  $$Hash\_i(x) = (Hash\_A(x) + i \times Hash\_B(x)) \pmod m$$  
    
  Это позволяет симулировать множество независимых хеш-функций, используя всего два базовых хеша (например, 64-битный результат MurmurHash3 или XxHash, разделенный на две 32-битные части).23

## 6. Примитивы синхронизации и потокобезопасность в кэшировании

Выбор примитива синхронизации определяет потолок пропускной способности вашего кэша.

### 6.1 lock (Monitor)

* **Механизм:** Гибридный. Сначала выполняет спин-блокировку (SpinWait) в пользовательском режиме (быстро), затем, если ресурс все еще занят, переходит в ожидание ядра ОС (медленно).
* **Сценарий использования:** Короткие критические секции. В простом LRU проверка словаря и перестановка указателей занимает наносекунды. Стандартный lock в этом случае часто работает быстрее, чем более сложные примитивы, благодаря низким накладным расходам на вход/выход.11

### 6.2 ReaderWriterLockSlim (RWLS)

* **Механизм:** Позволяет множеству потоков читать одновременно, блокируя писателей.
* **Накладные расходы:** Поддержание состояния читателей (проверка рекурсии, флаги обновления) стоит процессорных тактов. Вход в EnterReadLock значительно дороже, чем вход в Monitor.Enter.
* **Вердикт:** RWLS превосходит lock только если операция чтения занимает **существенное время** (например, вычисление хеша огромного объекта или перебор коллекции) или если блокировка удерживается долго. Для мгновенных операций поиска в словаре (TryGetValue) накладные расходы RWLS часто превышают выигрыш от параллелизма, особенно если соотношение чтение/запись не экстремально перекошено в сторону чтения.11

### 6.3 Lock-Free и Wait-Free структуры

* **Interlocked:** Использует атомарные инструкции процессора (LOCK CMPXCHG). Это самый быстрый способ обновления одиночных переменных (например, счетчиков статистики кэша).
* **Copy-On-Write:** Для сценариев с очень редкой записью (например, кэш конфигурации) можно использовать подмену указателя на всю структуру данных через Interlocked.Exchange. Это позволяет читателям работать вообще без блокировок, но запись становится дорогой (копирование всей структуры). Этот паттерн неприменим для LRU, где состояние меняется постоянно.28

## 7. Сравнительный анализ: Unity vs. Enterprise Backend

В завершение теоретического модуля, сведем ключевые различия подходов в таблицу для наглядности.

| **Характеристика** | **Паттерн Unity / GameDev** | **Паттерн Enterprise Backend** |
| --- | --- | --- |
| **Управление памятью** | Ручное, Пулинг (Pooling), Zero Alloc. Избегание new в цикле Update. | Полагается на GC, но оптимизируется под Gen 0. Использование IMemoryCache для управления жизненным циклом. |
| **Локальность данных** | Пространственная (Spatial). Массивы структур для кэша CPU/GPU. | Временная (Temporal - LRU). Пространственная локальность достигается внутренней оптимизацией коллекций. |
| **Конкурентность** | Однопоточная (Main Thread) или Job System (Data parallel). | Многопоточная (Request parallel). Блокировки и синхронизация неизбежны. |
| **Коллекции** | List<T>, Arrays. Избегание Dictionary в горячем коде из-за хеширования. | Dictionary<K,V>, ConcurrentDictionary. Скорость поиска O(1) является приоритетом. |
| **Худший сценарий** | Падение FPS (Frame Drop), статтеры. | Таймаут запроса / 503 Service Unavailable / OOM Crash. |

## 8. Заключение

Четвертый день обучения знаменует переход от использования коллекций как простых контейнеров к архитектуре систем хранения данных. Наивное использование Dictionary приводит к утечкам памяти. Наивное использование LinkedList приводит к неэффективности кэша процессора.

Как Senior Architect, вы должны выбирать правильный инструмент:

* Используйте **IMemoryCache** для кэширования общего назначения, где важна простота и встроенное управление экспирацией, но помните о его "ленивой" очистке и отсутствии гарантий размера байтов.
* Реализуйте **Кастомный LRU** (возможно, на базе массивов), когда вам нужны строгий детерминизм, гарантии Zero-Allocation или когда накладные расходы на компактификацию в IMemoryCache становятся узким местом.
* Применяйте **Фильтры Блума** как щит для вашего слоя данных, чтобы обеспечить стабильность системы под злонамеренной нагрузкой.

Освоив эти паттерны, вы перестаете просто "писать код" и начинаете "инженерить системы", способные к масштабированию.

## 9. Приложение: Детали реализации для практического задания

### 9.1 Рекомендованная архитектура "Thread-Safe LRU"

Для выполнения практической задачи откажитесь от использования стандартного LinkedList<T>. Поставьте себе челлендж реализовать паттерн **Double-Linked-List-on-Array**, обсужденный в разделе 3.3.

**Предлагаемый интерфейс:**

C#

public interface ILRUCache<K, V>  
{  
 bool TryGet(K key, out V value);  
 void Put(K key, V value);  
 int Capacity { get; }  
 int Count { get; }  
}

Обертка потокобезопасности (Decorator):

Идеально реализовать основную логику LRU как непотокобезопасный класс (Core), а затем обернуть его в декоратор ThreadSafeLRU, который управляет блокировками. Это соответствует Принципу единственной ответственности (SRP) — разделение логики алгоритма и логики синхронизации.

### 9.2 Бенчмаркинг

Используйте BenchmarkDotNet для сравнения следующих реализаций:

1. System.Runtime.Caching.MemoryCache (Legacy, для сравнения).
2. Microsoft.Extensions.Caching.Memory.IMemoryCache.
3. Ваш кастомный LRU на базе LinkedList<T>.
4. Ваш кастомный LRU на базе массивов (Array-Backed).

Измеряйте не только **Mean Time** (среднее время выполнения), но и **Allocated Memory** (Gen 0/1/2), чтобы продемонстрировать влияние ваших архитектурных решений на подсистему управления памятью. Это ключевая метрика для обоснования выбора технологии в Enterprise-системах.

#### Источники

1. План обучения: Архитектура коллекций и кэширование
2. Caching in .NET - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/caching>
3. Doubly Linked List Tutorial - GeeksforGeeks, дата последнего обращения: декабря 4, 2025, <https://www.geeksforgeeks.org/dsa/doubly-linked-list/>
4. 7 Cache Eviction Strategies. Caching is a technique to make… | by Premchandu | Medium, дата последнего обращения: декабря 4, 2025, <https://medium.com/@premchandu.in/7-cache-eviction-strategies-you-should-know-0bc4f08fd414>
5. Implementing Data Structures of How to Implement Bloom Filter in C# - Compile7, дата последнего обращения: декабря 4, 2025, <https://compile7.org/implement-data-structures/how-to-implement-bloom-filter-in-c/>
6. Cache in-memory in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-10.0>
7. High Concurrency LRU Caching, дата последнего обращения: декабря 4, 2025, <https://www.openmymind.net/High-Concurrency-LRU-Caching/>
8. Can we use doubly linked list in C without dynamic memory allocation? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/12991054/can-we-use-doubly-linked-list-in-c-without-dynamic-memory-allocation>
9. IMemoryCache Interface (Microsoft.Extensions.Caching.Memory), дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.caching.memory.imemorycache?view=net-10.0-pp>
10. Achieving zero garbage collection in Go? : r/golang - Reddit, дата последнего обращения: декабря 4, 2025, <https://www.reddit.com/r/golang/comments/1fko1al/achieving_zero_garbage_collection_in_go/>
11. When is ReaderWriterLockSlim better than a simple lock? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/4217398/when-is-readerwriterlockslim-better-than-a-simple-lock>
12. System.Threading.ReaderWriterLockSlim class - .NET - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-threading-readerwriterlockslim>
13. Creating High-Performance Locks and Lock-free Code (for .NET) - Adam Milazzo's Website, дата последнего обращения: декабря 4, 2025, <https://www.adammil.net/blog/v111_Creating_High-Performance_Locks_and_Lock-free_Code_for_NET_.html>
14. Why lock is 240% faster than ReaderWriterLockSlim? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/72462347/why-lock-is-240-faster-than-readerwriterlockslim>
15. Cache Eviction Policies | System Design - AlgoMaster.io, дата последнего обращения: декабря 4, 2025, <https://algomaster.io/learn/system-design/cache-eviction-policies>
16. runtime/src/libraries/Microsoft.Extensions.Caching.Memory/src/MemoryCache.cs at main · dotnet/runtime · GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.Caching.Memory/src/MemoryCache.cs>
17. NET: Couple words about using MemoryCache - Habr, дата последнего обращения: декабря 4, 2025, <https://habr.com/en/articles/697574/>
18. Microsoft.Extensions.Caching.Memory.MemoryCache post-eviction callbacks cause thread pool queueing · Issue #118552 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/118552>
19. ASP.NET Core In-Memory Caching Exploration - C# Corner, дата последнего обращения: декабря 4, 2025, <https://www.c-sharpcorner.com/article/asp-net-core-in-memory-caching-exploration/>
20. [Microsoft.Extensions.Caching.Memory.MemoryCache] Latest inserted entry is evicted first when cache is full · Issue #119503 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/119503>
21. MemoryCache Compact/Removal Efficiency · Issue #76643 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/76643>
22. IMemoryCache, refresh cache before eviction - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/58406476/imemorycache-refresh-cache-before-eviction>
23. Bloom filter - Wikipedia, дата последнего обращения: декабря 4, 2025, <https://en.wikipedia.org/wiki/Bloom_filter>
24. Bloom Filters | System Design - AlgoMaster.io, дата последнего обращения: декабря 4, 2025, <https://algomaster.io/learn/system-design/bloom-filters>
25. How many hash functions does my bloom filter need? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/658439/how-many-hash-functions-does-my-bloom-filter-need>
26. Bloom Filter Mathematical Proof - Medium, дата последнего обращения: декабря 4, 2025, <https://medium.com/@humberto521336/bloom-filters-mathematical-proof-8aa2e5d7b06b>
27. python bit array (performant) - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/20845686/python-bit-array-performant>
28. Does ReaderWriterLockSlim provide thread safety and speed efficiency (compared to a traditional lock) when using a List? - Software Engineering Stack Exchange, дата последнего обращения: декабря 4, 2025, <https://softwareengineering.stackexchange.com/questions/294514/does-readerwriterlockslim-provide-thread-safety-and-speed-efficiency-compared-t>