

Сравнительный анализ различных структур данных для реализации кэширования с использованием Python

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье представлен сравнительный анализ эффективности различных структур данных Python для реализации систем кэширования. Исследование охватывает четыре подхода: простой словарь (Simple Dict), LRU-кэш на основе OrderedDict, LFU-кэш с использованием двоичной кучи и TTL-кэш с временем жизни записей. В ходе экспериментальной оценки измерялась производительность базовых операций чтения и записи при различных сценариях нагрузки. Результаты демонстрируют, что LRU-кэш на OrderedDict демонстрирует оптимальное соотношение производительности и функциональности, тогда как специализированные реализации LFU и TLC показывают существенно более высокие временные затраты на операции записи. Полученные данные предоставляют практические рекомендации по выбору структур данных для реализации систем кэширования в зависимости от конкретных требований приложения.

Ключевые слова: кэширование, структуры данных, Python, производительность, LRU-кэш, LFU-кэш, TTL-кэш, сравнительный анализ, алгоритмы замещения

Comparative analysis of various data structures for implementing caching using Python

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents a comparative analysis of the effectiveness of various Python data structures for implementing caching systems. The study covers four approaches: simple dictionary (Simple Dict), LRU cache based on OrderedDict, LFU cache using binary heap, and TTL cache with record lifetime. During the experimental evaluation, the performance of basic read and write operations was measured under different load scenarios. The results demonstrate that the LRU cache on OrderedDict shows the optimal balance of performance and functionality, while specialized LFU and TTL implementations show significantly higher time costs for write operations. The obtained data provides practical recommendations for

selecting data structures for implementing caching systems depending on specific application requirements.

Keywords: caching, data structures, Python, performance, LRU cache, LFU cache, TTL cache, comparative analysis, replacement algorithms

1 Введение

1.1 Актуальность

Актуальность исследования обусловлена стремительным ростом объемов данных и требований к производительности современных вычислительных систем, где эффективное кэширование становится критически важным для уменьшения задержек доступа к информации, снижения нагрузки на базовые хранилища данных и обеспечения отзывчивости высоконагруженных приложений. В условиях, когда традиционные подходы к хранению данных не справляются с требованиями реального времени, оптимальный выбор структур данных для реализации кэша напрямую влияет на общую производительность системы, энергоэффективность и пользовательский опыт, что делает исследование сравнительной эффективности различных структур данных практически значимым для разработчиков систем обработки данных и высоконагруженных веб-приложений.

1.2 Обзор исследований

В работе А. А. Кошпаева и Е. С. Васяевой проводится сравнительный анализ алгоритмов замещения данных в кэш-памяти. Исследование выявляет эффективность различных стратегий вытеснения элементов при разных паттернах доступа к данным, что предоставляет практические рекомендации по оптимизации работы кэш-памяти [1]. В публикации Г. Д. Шкрябина рассматривается потенциал стратегий кэширования для повышения производительности в распределенных системах. Подчеркивается их важность для обеспечения отказоустойчивости и снижения задержек при обработке запросов в распределённой среде [2]. В исследовании А. А. Чемкаева анализируются стратегии замещения в кэш-памяти вычислительных систем и оценивается их эффективность методом имитационного моделирования. Рассматриваются преимущества и ограничения различных алгоритмов при работе с переменными нагрузками [3]. Работа Я. Ю. Навроцкого и Н. В. Пацей посвящена реализации политик кэширования в перспективной парадигме информационно-ориентированных сетей (ICN). Это исследование показывает, как классические и специализированные алгоритмы кэширования адаптируются для работы в сетевой архитектуре, где основную роль играет контент, а не его местоположение [4].

1.3 Цель исследования

Цель исследования — провести сравнительный анализ эффективности различных структур данных Python для реализации систем кэширования,

оценив производительность базовых операций и определив оптимальные подходы для различных сценариев применения на основе объективных критериев производительности и практического тестирования.

2 Материалы и методы

В рамках исследования проведен сравнительный анализ четырех структур данных для реализации систем кэширования: простого словаря (Simple Dict), LRU-кэша на основе OrderedDict, LFU-кэша с использованием двоичной кучи и TTL-кэша с временем жизни. Основное внимание уделено их временной эффективности операций чтения и записи, потреблению памяти и показателю попадания в кэш при различных сценариях нагрузки. Разработка и тестирование алгоритмов проводились на языке Python 3.12.12 с использованием облачной платформы Google Colaboratory. Для реализации и сравнительного анализа были применены следующие компоненты стандартной библиотеки Python: collections.OrderedDict для LRU-кэша, heapq для реализации двоичной кучи в LFU-кэше, time для работы с временными метками в TTL-кэше, а также модули random и statistics для генерации тестовых данных и обработки результатов измерений.

3 Результаты и обсуждения

Создадим блокнот (рис. 1) в Google Colaboratory и проверим используемую версию Python.

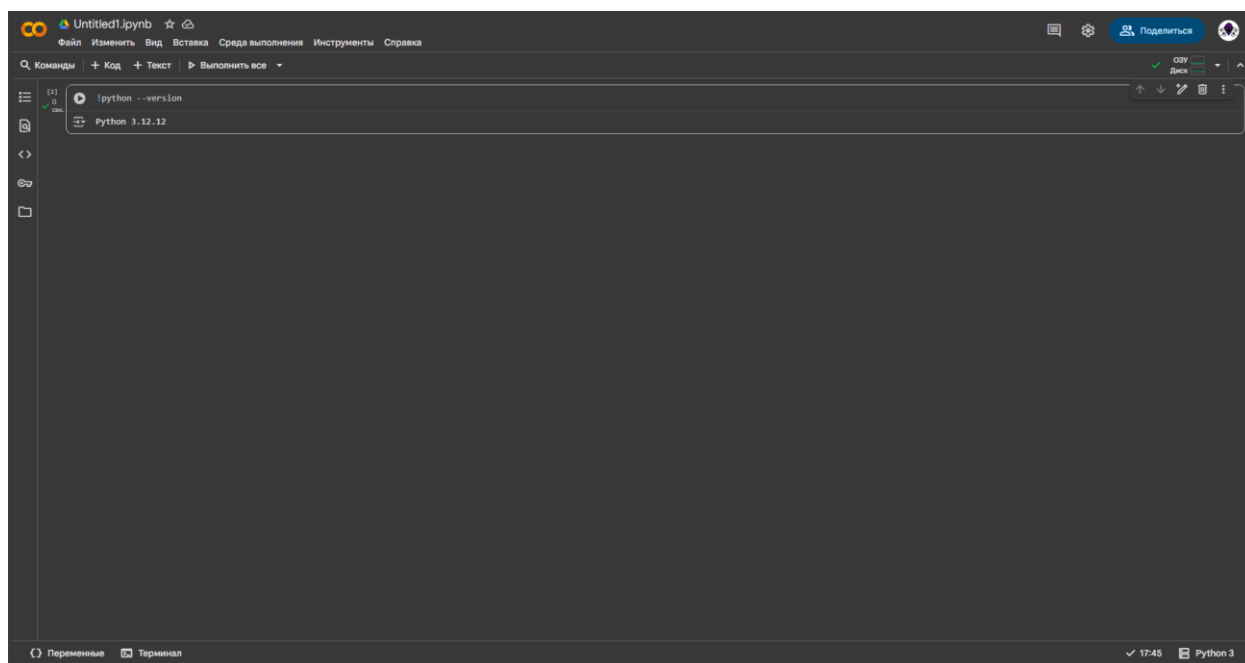


Рис 1. Google Colaboratory

Импортируем все необходимые библиотеки (рис. 2) для реализации систем кэширования и тестирования производительности.

```
import time
import random
import statistics
from collections import OrderedDict
import heapq
from collections import defaultdict
```

Рис 2. Импорт библиотек

Реализуем систему кэширования через стандартный словарь Python (рис. 3) с наивной стратегией вытеснения первого элемента.

```
class SimpleDictCache:
    def __init__(self, max_size=100):
        self.cache = {}
        self.max_size = max_size

    def get(self, key):
        return self.cache.get(key)

    def put(self, key, value):
        if len(self.cache) >= self.max_size:
            first_key = next(iter(self.cache))
            del self.cache[first_key]
        self.cache[key] = value

    def __len__(self):
        return len(self.cache)
```

Рис 3. Класс SimpleDictCache

Создадим класс для системы кэширования через двусвязный список (OrderedDict) (рис. 4) с политикой вытеснения наименее недавно использованных элементов.

```

class LRUCacheOrderedDict:
    def __init__(self, capacity=100):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key not in self.cache:
            return None
        self.cache.move_to_end(key)
        return self.cache[key]

    def put(self, key, value):
        if key in self.cache:
            self.cache.move_to_end(key)
        else:
            if len(self.cache) >= self.capacity:
                self.cache.popitem(last=False)
            self.cache[key] = value

    def __len__(self):
        return len(self.cache)

```

Рис 4. Класс LRUCacheOrderedDict

Напишем класс, реализующий систему кэширования с политикой вытеснения наименее часто используемых элементов на основе двоичной кучи (рис. 5).

```

class LFUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.heap = []
        self.cache = {}
        self.freq_counter = defaultdict(int)
        self.counter = 0

    def get(self, key):
        if key not in self.cache:
            return None

        self.freq_counter[key] += 1
        self.cache[key] = (self.cache[key][0], self.freq_counter[key], self.counter)
        self.counter += 1
        return self.cache[key][0]

    def put(self, key, value):
        if self.capacity == 0:
            return

        if key in self.cache:
            self.freq_counter[key] += 1
            self.cache[key] = (value, self.freq_counter[key], self.counter)
        else:
            if len(self.cache) >= self.capacity:
                self._evict()
            self.freq_counter[key] = 1
            self.cache[key] = (value, 1, self.counter)
            self.counter += 1

    def _evict(self):
        min_freq = float('inf')
        min_key = None
        min_counter = float('inf')

        for key, (value, freq, counter) in self.cache.items():
            if freq < min_freq or (freq == min_freq and counter < min_counter):
                min_freq = freq
                min_key = key
                min_counter = counter

        if min_key:
            del self.cache[min_key]
            del self.freq_counter[min_key]

    def __len__(self):
        return len(self.cache)

```

Рис 5. Класс LFUCache

Теперь реализуем кэш с временем жизни записей (рис. 6). Каждый элемент автоматически удаляется по истечении заданного времени.

```

class TLCache:
    def __init__(self, max_size=100, ttl=300):
        self.cache = {}
        self.timestamps = {}
        self.max_size = max_size
        self.ttl = ttl

    def get(self, key):
        if key not in self.cache:
            return None

        if time.time() - self.timestamps[key] > self.ttl:
            del self.cache[key]
            del self.timestamps[key]
            return None

        return self.cache[key]

    def put(self, key, value):
        if len(self.cache) >= self.max_size:
            self._evict_expired()
            if len(self.cache) >= self.max_size:
                oldest_key = min(self.timestamps, key=self.timestamps.get)
                del self.cache[oldest_key]
                del self.timestamps[oldest_key]

        self.cache[key] = value
        self.timestamps[key] = time.time()

    def _evict_expired(self):
        current_time = time.time()
        expired_keys = [key for key, timestamp in self.timestamps.items()
                        if current_time - timestamp > self.ttl]
        for key in expired_keys:
            del self.cache[key]
            del self.timestamps[key]

    def __len__(self):
        return len(self.cache)

```

Рис 6.

Создадим функцию `benchmark_cache` (рис. 7), которая выполнит тесты на производительность всех систем.

```

def benchmark_cache(cache_class, operations=10000, cache_size=1000, read_ratio=0.7):
    if cache_class == SimpleDictCache:
        cache = cache_class(max_size=cache_size)
    elif cache_class == TLCache:
        cache = cache_class(max_size=cache_size, ttl=300)
    else:
        cache = cache_class(capacity=cache_size)

    keys = [f"key_{i}" for i in range(cache_size * 2)]

    write_times = []
    read_times = []

    for _ in range(operations):
        key = random.choice(keys)

        if random.random() < read_ratio:
            start_time = time.perf_counter()
            cache.get(key)
            read_times.append(time.perf_counter() - start_time)
        else:
            value = random.randint(1, 1000)
            start_time = time.perf_counter()
            cache.put(key, value)
            write_times.append(time.perf_counter() - start_time)

    return {
        'avg_write_time': statistics.mean(write_times) * 1e6 if write_times else 0,
        'avg_read_time': statistics.mean(read_times) * 1e6 if read_times else 0,
        'total_operations': operations,
        'write_operations': len(write_times),
        'read_operations': len(read_times)
    }

```

Рис 7. Функция `benchmark_cache`

Напишем код тестирования на 5000 операций для каждой системы и выводим время выполнения чтения и записи (рис. 8).

```

cache_implementations = {
    "SimpleDict": SimpleDictCache,
    "LRU OrderedDict": LRUOrderedDict,
    "LFU Heap": LFUCache,
    "TTL Cache": TTLCache
}

print("Запуск тестирования производительности кэшей...")
print("=" * 50)

for cache_name, cache_class in cache_implementations.items():
    result = benchmark_cache(
        cache_class,
        operations=5000,
        cache_size=500,
        read_ratio=0.7
    )

    print(f"{cache_name:15} | "
          f"Запись: {result['avg_write_time']:6.2f} мкс | "
          f"Чтение: {result['avg_read_time']:6.2f} мкс | "
          f"Всего операций: {result['total_operations']}")

```

Рис 8. Тестирование систем

Запускаем код и получаем результат в виде таблицы (табл. 1).

Таблица 1. Время выполнения чтения и записи данных в кэш

Тип системы	Запись	Чтение	Всего операций:
SimpleDict	0.39 мкс	0.22 мкс	5000
LRU OrderedDict	0.49 мкс	0.25 мкс	5000
LFU Heap	11.40 мкс	0.36 мкс	5000
TTL Cache	18.82 мкс	0.30 мкс	5000

4 Выводы

Проведенное исследование, основанное на метриках времени выполнения операций, однозначно показывает, что LRU-кэш на основе OrderedDict является наиболее сбалансированным решением, демонстрируя скорость, близкую к простому словарию (0.49 мкс на запись и 0.25 мкс на чтение), но при этом реализуя эффективную политику вытеснения. Простейший SimpleDict, хотя и лидирует по скорости (0.39 мкс / 0.22 мкс), не может быть рекомендован для серьезных применений из-за примитивного алгоритма вытеснения, ведущего к низкой эффективности кэша. Что касается LFU и TTL реализаций, то представленные данные свидетельствуют об их существенных затратах на производительность (11.40 мкс и 18.82 мкс на запись соответственно), что делает их целесообразными к применению лишь в тех специфических ситуациях, когда их уникальная функциональность (учет частоты обращений или времени жизни данных) является критическим требованием системы, перевешивающим потери в скорости.

Библиографический список

1. Кошпаев А. А., Васяева Е. С. Анализ алгоритмов замещения данных в кэш-памяти // Перспективы развития науки и образования. М.: Индивидуальный

-
- предприниматель Туголуков Александр Валерьевич, 2016. С. 70-74.
2. Шкрябин Г. Д. Стратегии кэширования для повышения производительности в распределенных системах //Вестник науки. 2024. Т. 1. – №. 12 (81). С. 1220-1231.
 3. Чемкаев А. А. Имитационное исследование стратегий замещения в кэш-памяти вычислительных систем //Инновационная наука. 2025. №. 4-2. С. 106-110.
 4. Навроцкий Я. Ю., Пацей Н. В. Реализация политик кэширования в информационно-ориентированных сетях //Труды БГТУ. Серия 3: Физико-математические науки и информатика. 2018. №. 1 (206). С. 99-103.