⚡ Практическое занятие

Метрики производительности и CAP-теорема

ФИО студента:

Завьялова Софья Александровна

Группа:

426

Дата:

26.10.2025

📊 Основные метрики производительности

Percentiles (Процентили)

- **P50 (медиана):** 50% запросов быстрее этого времени
- **P95:** 95% запросов быстрее
- **P99:** 99% запросов быстрее
- **P99.9:** 99.9% запросов быстрее

⚖️ CAP-теорема

Правило: В распределенной системе можно гарантировать только 2 из 3 свойств:

Практические компромиссы:

- **CP системы:** MongoDB, Redis Cluster (жертвуют доступностью)
- **AP системы:** Cassandra, DynamoDB (жертвуют согласованностью)
- **CA системы:** PostgreSQL, MySQL (не устойчивы к разделению)

🔒 ACID свойства

D - Durability (Долговечность)

Зафиксированные изменения сохраняются навсегда

Пример: После подтверждения транзакции данные не теряются даже при сбое

📈 Availability (Доступность)

Формула: Availability = (Uptime / Total Time) × 100%

SLA уровни:

🧪 Практические задания

Требования к выполнению:

- выполняются на ОС **Linux** (любой дистрибутив)
- пакеты устанавливаются в **виртуальное окружение venv**
- для создания серверов можно использовать **LLM** (желательно Claude Sonnet подключить в IDE)

Задание 1: Установка и использование Apache Bench

Установка Apache Bench на Linux:

```
# Ubuntu/Debian
sudo apt-get install apache2-utils

# CentOS/RHEL/Fedora
sudo yum install httpd-tools
# или
sudo dnf install httpd-tools

# Проверка установки
ab -V
```

Тестирование производительности:

```
# Запуск 1000 запросов с 10 одновременными соединениями
ab -n 1000 -c 10 http://httpbin.org/get
```

Результаты тестирования:

Время выполнения теста:

69.095 секунд

Requests per second:

14.47 запросов/сек

Time per request (mean):

690.945 мс

Статистика соединений (мс):

	min	mean	median	max
Connect:	109	132	115	3057
Processing:	110	528	253	4671

Перцентили времени ответа:

P50: 380 мс P90: 1402 мс P95: 1829 мс P99: 3800 мс

Описание полученных результатов:

```
Негативные аспекты:
Низкая пропускная способность - всего 14.47 запросов в секунду
Высокое время отклика - среднее время запроса 691 мс (почти 0.7
секунды)
Большой разброс времени ответа - от 221 мс до 4789 мс (разница в
21 раз)
Есть ошибки - 7 failed requests (0.7% от всех запросов)
Долгая обработка - серверу требуется значительное время на
Processing (до 4.6 секунд)
Позитивные аспекты:
Стабильное соединение - время установки соединения относительно
```

📊 Подробный анализ результатов Apache Bench

Общая информация о тесте:

Утилита: ApacheBench 2.3
URL: http://httpbin.org/get
Количество запросов: 1000
Уровень параллелизма: 10 одновременных соединений

Параметры сервера:

```
Server Software: awselb/2.0 ← AWS Elastic Load Balancer
Server Hostname: httpbin.org
Server Port: 80 ← HTTP порт
Document Path: /get ← Тестируемый endpoint
Document Length: 162 bytes ← Размер ответа
```

Расшифровка статистики соединений:

- **Connect:** Время установки TCP-соединения
- **Processing:** Время обработки запроса сервером
- **Waiting:** Время от отправки запроса до получения первого байта
- **Total:** Общее время запроса

Анализ качества:

Failed requests: 7 (ошибок нет) Transfer rate: 6.82 Kbytes/sec

Выводы о производительности:

```
1. Стабильность: Высокое стандартное отклонение (652.6 мс) -
нестабильная работа сервера
2. Производительность: 14.47 запросов/секунду - низкая пропускная
способность
3. Задержки: Среднее время ответа 690.945 мс - высокие задержки
```

Рекомендации для улучшения:

- Оптимизировать можно за счет кеширования
- Рассмотреть CDN для уменьшения задержек
- Увеличить параллелизм для проверки пределов сервера

⏱️ Задание 2: Измерение задержек (latency)

Простой HTTP сервер с задержками (Flask):

```python
from flask import Flask
import time
import random

app = Flask(__name__)

@app.route('/fast')
def fast_endpoint():
    return {'response': 'fast', 'latency': '10ms'}

@app.route('/slow')
def slow_endpoint():
    time.sleep(random.uniform(0.1, 0.5))  # 100-500ms задержка
    return {'response': 'slow', 'latency': '100-500ms'}

if __name__ == '__main__':
    app.run(port=5000)
```

Измерение latency:

```bash
# Тестирование быстрого endpoint
ab -n 100 -c 5 http://localhost:5000/fast

# Тестирование медленного endpoint
ab -n 100 -c 5 http://localhost:5000/slow
```

Результаты /fast endpoint:

RPS:

417.64

Средняя задержка:

11.972 мс

Результаты /slow endpoint:

RPS:

15.15

Средняя задержка:

330.063 мс

Сравнение веб-фреймворков

FastAPI (порт 8000)

Код на Python:

```
from fastapi import FastAPI
import time
import random
import uvicorn

app = FastAPI()

@app.get("/fast")
async def fast_endpoint():
    return {
        "response": "fast",
        "framework": "FastAPI",
        "latency": "10ms"
    }

@app.get("/slow")
async def slow_endpoint():
    delay = random.uniform(0.1, 0.5)  "color: #95a5a6;"># 100-500ms задержка
    time.sleep(delay)
    return {
        "response": "slow",
        "framework": "FastAPI",
        "latency": "100-500ms"
    }

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

2816.90

Средняя задержка:

1.775 мс

Результаты /slow endpoint:

RPS:

3.17

Средняя задержка:

1578.319 мс

Краткие выводы:

FastAPI идеально подходит для асинхронных веб-приложений с большим количеством быстрых запросов, но требует осторожности с блокирующими операциями. Для достижения максимальной производительности необходимо использовать асинхронные библиотеки вместо синхронных sleep/blocking операций

Django (порт 8001)

Код на Python:

```
import os
import sys
import time
import random
from django.conf import settings
from django.core.handlers.wsgi import WSGIHandler
from django.urls import path
from django.http import JsonResponse

"color: #95a5a6;"># Настройка Django
settings.configure(
    DEBUG=True,
    SECRET_KEY='django-insecure-test-key-12345',
    ROOT_URLCONF=__name__,
    ALLOWED_HOSTS=['*'],
    INSTALLED_APPS=[
        'django.contrib.contenttypes',
        'django.contrib.auth',
    ],
    MIDDLEWARE=[
        'django.middleware.common.CommonMiddleware',
    ],
    USE_TZ=True,
)

"color: #95a5a6;"># View функции
def fast_endpoint(request):
    return JsonResponse({
        "response": "fast",
        "framework": "Django",
        "latency": "10ms"
    })

def slow_endpoint(request):
    delay = random.uniform(0.1, 0.5)
    time.sleep(delay)
    return JsonResponse({
        "response": "slow",
        "framework": "Django",
        "latency": "100-500ms"
    })

"color: #95a5a6;"># URL patterns
```

```
urlpatterns = [
    path('fast', fast_endpoint),
    path('slow', slow_endpoint),
]

"color: #95a5a6;"># Запуск сервера
if __name__ == '__main__':
    from django.core.management import execute_from_command_line

    "color: #95a5a6;"># Имитируем команду runserver
    execute_from_command_line(['manage.py', 'runserver', '8001'])
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

1492.36

Средняя задержка:

3.350 мс

Результаты /slow endpoint:

RPS:

15.56

Средняя задержка:

321.257 мс

Краткие выводы:

Django продемонстрировал сбалансированную производительность с показателями 1492.36 RPS и задержкой 3.35 мс на быстрых запросах, что в 3.6 раза превышает производительность Flask, но уступает FastAPI. На медленных запросах Django показал наивысшую эффективность с 15.56 RPS и задержкой 321 мс, что в 4.9 раза лучше FastAPI и характеризуется минимальным разбросом времени ответа от 105 до 494 мс. Синхронная архитектура Django обеспечила

Django REST Framework (порт 8002)

Код на Python:

```
import os
import time
import random

"color: #95a5a6;"># Сначала настраиваем Django
from django.conf import settings

settings.configure(
    DEBUG=True,
    SECRET_KEY='django-insecure-test-key-12345',
    ROOT_URLCONF=__name__,
    ALLOWED_HOSTS=['*'],
    INSTALLED_APPS=[
```

```
        'django.contrib.contenttypes',
        'django.contrib.auth',
        'rest_framework',
    ],
    MIDDLEWARE=[
        'django.middleware.common.CommonMiddleware',
    ],
    REST_FRAMEWORK={
        'DEFAULT_RENDERER_CLASSES': [
            'rest_framework.renderers.JSONRenderer',
        ],
        'DEFAULT_PARSER_CLASSES': [
            'rest_framework.parsers.JSONParser',
        ]
    },
    USE_TZ=True,
)

"color: #95a5a6;"># Теперь импортируем остальные модули DRF
from django.urls import path
from django.core.wsgi import get_wsgi_application
from rest_framework.response import Response
from rest_framework.views import APIView
from rest_framework.decorators import api_view

"color: #95a5a6;"># View через APIView класс
class FastEndpoint(APIView):
    def get(self, request):
        return Response({
            "response": "fast",
            "framework": "Django REST Framework",
            "latency": "10ms"
        })

class SlowEndpoint(APIView):
    def get(self, request):
        delay = random.uniform(0.1, 0.5)
        time.sleep(delay)
        return Response({
            "response": "slow",
            "framework": "Django REST Framework",
            "latency": "100-500ms"
        })

"color: #95a5a6;"># URL patterns
urlpatterns = [
    path('fast', FastEndpoint.as_view()),
    path('slow', SlowEndpoint.as_view()),
]

"color: #95a5a6;"># Запуск сервера
if __name__ == '__main__':
    import django
    django.setup()
```

```
from django.core.management import execute_from_command_line
execute_from_command_line(['manage.py', 'runserver', '8002'])
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

1113.46

Средняя задержка:

4.490 мс

Результаты /slow endpoint:

RPS:

16.60

Средняя задержка:

301.203 мс

Краткие выводы:

Результаты тестирования DRF показали производительность 1113.46 RPS с задержкой 4.490 мс на быстрых запросах и 16.60 RPS с задержкой 301.203 мс на медленных запросах. По сравнению с чистым Django, DRF демонстрирует снижение производительности на 25.4% на быстрых запросах (1113 vs 1492 RPS) при практически идентичных показателях на медленных операциях (16.60 vs 15.56 RPS). Дополнительная абстракция фреймворка увеличила задержку на 34% для легких запросов, но не оказала существенного влияния на обработку блокирующих операций. DRF обеспечивает баланс между функциональностью REST API и приемлемой производительностью, сохраняя надежность работы с нулевым количеством ошибок и

LiteStar (порт 8003)

Код на Python:

```
from litestar import Litestar, get
import time
import random

@get("/fast")
async def fast_endpoint() -> dict:
    return {
        "response": "fast",
        "framework": "LiteStar",
        "latency": "10ms"
    }

@get("/slow")
async def slow_endpoint() -> dict:
    "color: #95a5a6;"># Используем синхронный sleep в отдельном потоке
    delay = random.uniform(0.1, 0.5)
    time.sleep(delay)  "color: #95a5a6;"># Синхронный sleep блокирует поток
    return {
```

```
        "response": "slow",
        "framework": "LiteStar",
        "latency": "100-500ms"
    }

app = Litestar(route_handlers=[fast_endpoint, slow_endpoint])

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8003)
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

3009.96

Средняя задержка:

1.661 мс

Результаты /slow endpoint:

RPS:

3.02

Средняя задержка:

1657.893 мс

Краткие выводы:

LiteStar показал наивысшую производительность на быстрых запросах с 3009.96 RPS и задержкой 1.661 мс, превзойдя все другие фреймворки. На медленных запросах производительность составила 3.02 RPS с задержкой 1657.893 мс, что сопоставимо с FastAPI но хуже чем Django-подобные фреймворки. Асинхронная архитектура LiteStar обеспечивает максимальную эффективность на I/O-bound операциях, однако при наличии синхронных блокирующих операций демонстрирует значительное снижение производительности

Общие выводы о производительности веб-фреймворков:

Сравнительный анализ пяти фреймворков выявил четкую зависимость производительности от архитектурного подхода. Асинхронные фреймворки (LiteStar и FastAPI) демонстрируют максимальную производительность на легких запросах с 3009.96 и 2816.90 RPS соответственно, однако значительно уступают синхронным решениям при обработке блокирующих операций - их производительность падает до 3.02-3.17 RPS. Синхронные фреймворки показывают противоположную тенденцию: Django и Django REST Framework обеспечивают стабильно высокую производительность на медленных запросах (15.56-16.60 RPS) при умеренных показателях на быстрых операциях (1113-1492 RPS). Flask занимает последнее место по производительности на легких запросах с 417.64 RPS. Наибольший баланс производительности демонстрирует Django, который сочетает хорошие показатели на быстрых запросах с лидирующими результатами

📝 Домашнее задание

Задача: Доделать задания

Срок сдачи: К следующему занятию

Что нужно сделать:

- Протестировать 3 разных сайта с помощью Apache Bench
- Сравнить их производительность
- Построить график зависимости RPS от количества одновременных соединений
- Создать HTTP серверы на всех указанных фреймворках
- Провести сравнительное тестирование производительности

📊 Критерии оценки

📚 Литература

- Клеппман М. "Высоконагруженные приложения"
- Фаулер М. "Архитектура корпоративных программных приложений"
- Документация Apache Bench
- Официальная документация веб-фреймворков