**BC/NW 2018 № 2 (33):3.1**

**РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ОЦЕНКИ ФУНКЦИОНИРОВАНИЯ DOCKER-КОНТЕЙНЕРА**

 Макаревич В.В.

**Постановка задачи**

В рамках данной работы необходимо разработать набор программ для проведения тестирования и анализа функционирования приложений, упакованных в *Docker*-контейнеры.

*Docker* [1] - программное обеспечение (ПО) для управления приложениями в среде виртуализации на уровне операционной системы (ОС). В данный момент Docker является очень популярным ПО для переноса и развертывания приложений. Принцип работы *Docker* показан на рис. 1. Главное отличие контейнеров *Docker* от виртуальных машин (ВМ) заключается в том, что контейнеры делят ресурсы операционной системы, как показано на рисунке, при этом каждый контейнер изолирован друг от друга. Важным принципом является то, что каждый контейнер запускается в отдельном изолированном процессе.



Рис. 1. Принцип работы Docker 4

Очевидно, что Docker необходим для решения некоторых проблем, связанных с недостатками виртуальных машин. Основной целью НИР является анализ средств виртуализации в рамках сравнения применения ВМ и контейнеризации и формирование вывода о том, для каких целей подходят контейнеры, а для каких - ВМ.

В данной работе рассматриваются именно контейнеры Docker и анализируется работа с ними для выполнения следующих задач:

1) Задачи, связанные с большим количеством расчетов

2) Задачи, связанные с большой нагрузкой на сервер (обработка большого количества запросов)

3) Задачи, связанные с передачей файлов больших размеров (пропускная способность контейнера)

Таким образом, в соответствии с поставленной в задании задачей необходимо разработать следующие тестируемые программы (разработка данных программ предусматривает упаковывание их в Docker-контейнер):

● Программа для рекурсивного расчета чисел Фибоначчи

● Программа для выполнения сжатия данных

● Программа, содержащая RESTful API для обработки запросов, а также способная принимать файлы, передающиеся по локальной сети

Для анализа функционирования данных контейнеров разработать 2 программы: 5

● Программа для оценки средней нагрузки на CPU (в %) за время ее активной работы по идентификатору процесса

● Программа для проведения нагрузочного тестирования RESTful API

**Разработка тестируемых программ**

Для разработки тестируемых программ была выбрана среда Node.js [2], основанная на языке высокого уровня JavaScript. Данная технология является популярным решением для написания серверных приложений, с ее помощью можно быстро создать сервер с RESTful API, а также она содержит API, созданный с использованием C++, что позволяет легко выполнить низкоуровневые команды, что впоследствии позволит получать данные, например, о нагрузке на CPU. Данные факторы повлияли на выбор Node.js, как основного языка для выполнения данной работы.

Описание разработки рассматриваемых приложений приведено ниже. Для каждой задачи предусмотрен подсчет времени выполнения.

*Разработка программы для расчета чисел Фибоначчи*

Последовательность чисел Фибоначчи можно представить следующим образом: 𝐹={𝑓𝑖},где 𝑓𝑖 = 𝑓𝑖−1 + 𝑓𝑖−2

Классическим способом нахождения числа из последовательности Фибоначчи является рекурсия. Для нахождения 𝑓𝑖,где 𝑖 −большое число,с помощью рекурсии потребуется очень много ресурсов, поскольку количество вызовов функций с увеличением i будет многократно возрастать, как показано на рис. 2. При подсчете получилось, что, например, для 𝑖=45количество вызовов 3672623805, что приводит к большой нагрузке на CPU. 7



Рис. 2. Схематичное изображение рекурсивных вызовов

*Разработка программы для сжатия данных*

Данная задача рассматривается на примере сжатия папки (600 МБ, 10 файлов) в формат *zip.* Вообще говоря, для сжатия файлов в формат в *zip* используется алгоритм, называемый *Deflate*, представляющий из себя сочетание алгоритмов кодирования *LZ77* и Хаффмана. Алгоритм представляет из себя различные битовые операции. Поэтому для большого количества данных сжатие представляет собой затратную операцию.

Для реализации сжатия папки на языке *Node.js* была использована библиотека *compressing*, которая выполняет кодирование файлов по описанному выше алгоритму. Данная библиотека содержит функцию *compressDir*, на вход которой подается адрес директории, которую необходимо сжать. Вторым параметром данной функции является адрес файла, являющегося результатом сжатия. Результатом функции является сообщение об успешной или же неуспешной попытке архивирования файла.

Листинг программы, выполняющей сжатие файла, представлен в *приложении Б*.

*. Разработка программы для обработки запросов*

Основным инструментом для создания данной программы является библиотека *express*, служащая для создания *HTTP*-сервера. С помощью данной библиотеки процесс, в котором запущен сервер, слушает порт и при получении запроса обрабатывает пришедшие данные и отправляет ответ клиенту. 8

Для создания данной программы были реализованы конечные точки и контроллеры обработки этих запросов. Их описание представлено в табл. 1.





Рис. 3 Схема алгоритма поиска чисел Фибоначи с помощью циклов

**3. Упаковывание программ в Docker**

На примере программы сжатия файлов, описанной в пункте 2.2, покажем, как создать Docker-контейнер с программой и необходимым окружением.

Перед запуском контейнера необходимо создать *Docker*-образ. Образ - это шаблон, по которому создается контейнер, т. е. он представляет собой структуру, которая содержит все необходимое окружение. В случае с рассматриваемой программой образ должен содержать *Node.js*, архивируемую папку и саму написанную программу. *Docker*-образ создается с помощью специального сценарного файла *Dockerfile*, который помещается в папку с приложением. *Docker* также удобен тем, что существует ресурс *DockerHub*, содержащий множество готовых образов, на основе которых можно создавать свои образы.

Написание *Dockerfile* для данной программы начинается с команды, позволяющей получить образ, содержащий все необходимое окружение для работы среды *Node.js* 8, на основе которого мы построим образ для данной программы:

FROM node:8

Далее необходимо обозначить директорию, в которой будет находиться приложение в файловой системе контейнера с помощью команды:

WORKDIR /usr/src/app

После этого необходимо написать набор команд для копирования файлов программы в образ, а также скрипт, с помощью которого скачаются все необходимые приложению модули:

COPY package\*.json ./

COPY . .

RUN npm install

Далее необходимо задать порт, который приложение будет использовать в *Docker*-контейнере и после этого написать скрипт, с помощью которого программа будет запущена:

EXPOSE 7878

CMD [ "npm", "run", "dev" ]

Следующим шагом является создание *Docker*-образа с тегом, идентифицирующим этот образ:

docker build -t image-name .

И последним шагом является запуск контейнера с помощью команды:

docker run -p 49160:7878 -d image-name

Запрос, направленный на порт 49160, будет переадресовано на порт 7878 в контейнере. После запуска контейнера сразу же запустится приложение внутри этого контейнера благодаря конфигурации *Dockerfile.*

Стоит отметить, что другие программы были упакованы в контейнеры по тому же принципу. Для получения логов работы приложения внутри контейнера служит команда: docker logs CONTAINER\_ID.

**4. Разработка программ для анализа функционирования контейнеров**

Как было указано выше, в работе рассматриваются 3 типа приложений. Ниже приведено описание инструементов, необходимых для анализа функционирования каждого из типов

**4.1. Анализ нагрузки на *CPU***

Для тестирования нагрузки на *CPU* была выбрана задача с интенсивными расчетами, в связи с этим для решения необходимо разработать программу, оценивающую среднюю нагрузку процессом, соответствующим тестируемой программе на *CPU* (в %) во время активной работы. Входным параметром для программы является идентификатор процесса.

Выбранные вычислительные мощности характеризуются наличием 2 ядер, поэтому максимальным значением нагрузки может быть 100×2 =200(%). В *Unix*-системах для вывода нагрузки на *CPU* служит команда в терминале:

ps up **PROCESS\_ID** | tail -n1 | tr -s ' ' | cut -f3 -d' '

*PROCESS\_ID* является идентификатором процесса. В связи с этим необходимо разработать программу, функционирующую по алгоритму, показанному на рис. 4, где *CCL* - результат выполнения указанного выше скрипта. Результат выполнения данной программы выглядит следующим образом: average cpu usage is 119.75 %



Рис.4 Схема алгоритма программы анализирующей загрузку на CPU

*Анализ приема и обработки запросов*

Для решения данной задачи необходимо разработать программу, с помощью которой можно будет посылать большое количество *HTTP*-запросов по локальной сети. Данная программа позволит оценить работу *RESTful API* сервера путем подсчета процентного соотношения запросов, ответа на которых не было.

Язык высокого уровня *Python* [3 хорошо подходит для выполнения данной задачи, поскольку поддерживает работу с потоками. Для решения поставленной задачи была выбрана библиотека *locust* [4], которая позволяет просимулировать большое количество запросов и создать нагрузочное тестирование. Запросы будут посылаться на адрес №1 из табл. 1. В теле запроса будет содержаться случайное число из диапазона [100; 200]. Листинг данной тестирующей программы представлен в *Приложении В*.

Для запуска тестирования необходимо запустить команду:

locust --host=http://localhost:7878

Данная библиотека содержит GUI, с помощью которого можно проанализировать результат выполнения тестирования, графики, сообщения об ошибках.

*Анализ пропускной способности*

Для решения данной задачи будут выполнены следующие действия:

1) На описанный выше *RESTful API* сервер будет посылаться файл большого размера (976 МБ).

2) Будут замерены следующие показатели: время ответа (Вот) на запрос и время обработки запроса на сервере (Воб). Первый параметр будет получен при помощи использования консоли разработчика браузера *Google Chrome*, а второй непосредственно на сервере с помощью получения логов.

3) Путем подсчета разности Вот и Воб будет получено время передачи (Вп) файла от клиента к серверу

4) Данные, полученные с использованием *Node*-процесса и *Docker*-контейнера будут сравнены, будет сформирован вывод о пропускной способности контейнера.

**Получение и обработка результатов**

Для получения наиболее полной оценки работы контейнеров необходимо сравнить функционирование данных программ, запущенных в *Docker*-контейнерах и *Node*-процессах. Для этого были произведены серии по 5 тестов для каждой программы с использованием соответствующих, описанных выше инструментов. Тестирование производилось на следующих вычислительных мощностях: процессор - 2.3 GHz Intel Core i5, 2 ядра; память - 8 ГБ 2133 MHz LPDDR3. Данные о тестировании приведены ниже. Все данные в таблицах усреднены.

 *Результаты анализа нагрузки на CPU*

Средняя нагрузка на CPU, а также среднее выполнение задач, описанных в п. 2.1 и 2.2 данной работы, представлены в табл. 2. Все данные округлены до целых значений.

Табл. 2 Результаты анализа нагрузки на *CPU*



**5.2. Результаты нагрузочного тестирования на *REST API* сервер**

Для данного тестирования были взяты следующие исходные данные: количество запросов в секунду, равное 500, количество запросов 60000. В табл. 3 приведены усредненные результаты о количестве запросов, на 17

которые не были получены ответы, в том числе и в %. Количество округлено до целых значений.

Табл. 3

Результаты нагрузочного тестирования на *REST API* сервер



*Результаты анализа пропускной способности*

Табл. 4 содержит данные о Вот, Воб и Вп файла размером 964 МБ по локальной сети. Впвзят, как разница между средними значениями

Табл. 4

Результаты анализа пропускной способности контейнера



**Заключение**

В процессе выполнения работы были получены навыки по работе с ПО Docker и написанию тестирования, проанализированы различные параметры, характеризующие Docker-контейнер. Были получены следующие результаты:

1) Нагрузка на *CPU* - приложение с интенсивными расчетами, упакованное в контейнер показало удовлетворительный результат и не сильно отличалось от расчетов произведенных непосредственно на хосте. Предварительно сформирован вывод о том, что контейнеризацию можно применять для решения сложных расчетных задач (например, обучение нейронных сетей), поскольку разница в нагрузке и времени расчета не велика.

2) Обработка большого количества запросов - результат был получен неудовлетворительный, количество ошибок с использованием Docker-контейнера является очень большим. Однако, есть вероятность, что это связано со слишком слабыми ресурсами оборудования

3) Пропускная способность контейнера - был получен также неудовлетворительный результат. Предварительный вывод состоит в том, что контейнеризацию не стоит применять для обработки больших данных

Стоит заметить, что все сделанные выводы являются предварительными. При выполнении НИР их необходимо подтвердить или опровергнуть после выполнения всех операций на более производительных ресурсах.

**Литература**

1. Официальный сайт загрузки Docker. Режим доступа: https://www.docker.com/

2. Официальный сайт загрузки Node.js. Режим доступа: http://nodejs.com/

3. Официальный сайт загрузки Python. Режим доступа: https://www.python.org/

4. Mouat A. Using Docker Developing and Deploying Software with Containers - O’Reilly Media, 2015 — 354 p.

Очевидно, что Docker необходим для решения некоторых проблем, связанных с недостатками виртуальных машин. Основной целью НИР является анализ средств виртуализации в рамках сравнения применения ВМ и контейнеризации и формирование вывода о том, для каких целей подходят контейнеры, а для каких - ВМ.

В данной работе рассматриваются именно контейнеры *Docker* и анализируется работа с ними для выполнения следующих задач:

1) Задачи, связанные с большим количеством расчетов

2) Задачи, связанные с большой нагрузкой на сервер (обработка большого количества запросов)

3) Задачи, связанные с передачей файлов больших размеров (пропускная способность контейнера)

Таким образом, в соответствии с поставленной в задании задачей необходимо разработать следующие тестируемые программы (разработка данных программ предусматривает упаковывание их в *Docker*-контейнер):

● Программа для рекурсивного расчета чисел Фибоначчи

● Программа для выполнения сжатия данных

● Программа, содержащая *RESTful API* для обработки запросов, а также способная принимать файлы, передающиеся по локальной сети

Для анализа функционирования данных контейнеров разработать 2 программы: 5

● Программа для оценки средней нагрузки на *CPU* (в %) за время ее активной работы по идентификатору процесса

● Программа для проведения нагрузочного тестирования *RESTful API*

Задание на курсовой проект расположено в *приложении А*. 6

2. Разработка тестируемых программ

Для разработки тестируемых программ была выбрана среда *Node.js* [2], основанная на языке высокого уровня *JavaScript*. Данная технология является популярным решением для написания серверных приложений, с ее помощью можно быстро создать сервер с *RESTful API*, а также она содержит *API*, созданный с использованием *C*++, что позволяет легко выполнить низкоуровневые команды, что впоследствии позволит получать данные, например, о нагрузке на *CPU*. Данные факторы повлияли на выбор *Node.js*, как основного языка для выполнения данной работы.

Описание разработки рассматриваемых приложений приведено ниже. Для каждой задачи предусмотрен подсчет времени выполнения.

2.1. Разработка программы для расчета чисел Фибоначчи

Последовательность чисел Фибоначчи можно представить следующим образом: 𝐹={𝑓𝑖},где 𝑓𝑖 = 𝑓𝑖−1 + 𝑓𝑖−2

Классическим способом нахождения числа из последовательности Фибоначчи является рекурсия. Для нахождения 𝑓𝑖,где 𝑖 −большое число,с помощью рекурсии потребуется очень много ресурсов, поскольку количество вызовов функций с увеличением *i* будет многократно возрастать, как показано на рис. 2. При подсчете получилось, что, например, для 𝑖=45количество вызовов 3672623805, что приводит к большой нагрузке на *CPU*. 7

Рис. 2. Схематичное изображение рекурсивных вызовов

Листинг программы, выполняющей рекурсивный подсчет числа из последовательности Фибоначчи представлен в *Приложении Б*.

2.2. Разработка программы для сжатия данных

Данная задача рассматривается на примере сжатия папки (600 МБ, 10 файлов) в формат *zip.* Вообще говоря, для сжатия файлов в формат в *zip* используется алгоритм, называемый *Deflate*, представляющий из себя сочетание алгоритмов кодирования *LZ77* и Хаффмана. Алгоритм представляет из себя различные битовые операции. Поэтому для большого количества данных сжатие представляет собой затратную операцию.

Для реализации сжатия папки на языке *Node.js* была использована библиотека *compressing*, которая выполняет кодирование файлов по описанному выше алгоритму. Данная библиотека содержит функцию *compressDir*, на вход которой подается адрес директории, которую необходимо сжать. Вторым параметром данной функции является адрес файла, являющегося результатом сжатия. Результатом функции является сообщение об успешной или же неуспешной попытке архивирования файла.

Листинг программы, выполняющей сжатие файла, представлен в *приложении Б*.

2.3. Разработка программы для обработки запросов

Основным инструментом для создания данной программы является библиотека *express*, служащая для создания *HTTP*-сервера. С помощью данной библиотеки процесс, в котором запущен сервер, слушает порт и при получении запроса обрабатывает пришедшие данные и отправляет ответ клиенту. 8

Для создания данной программы были реализованы конечные точки и контроллеры обработки этих запросов. Их описание представлено в табл. 1.

Табл. 1

|  |  |  |
| --- | --- | --- |
| Описание всех обработчиков Endpoint  | Контроллер  | Описание  |
| *POST(/)*  | *fibCounting*  | Входные данные: n Подсчет числа из последовательности Фибоначчи с индексом *n.* Для данного контроллера был реализован другой алгоритм для подсчета числа, отличный от рекурсивного, использующий вычисления в цикле для увеличения нагрузки. Схема данного алгоритма показана на рис. 3. С помощью данного запроса будет протестирована интенсивная нагрузка на сервер, связанная с большим количеством запросов.  |
| *POST(/upload)*  | *fileUploading*  | Входные данные: файл Прием файла и его сохранение. С помощью данного запроса будет протестирована пропускная способность контейнера.  |