МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики
и компьютерных наук им. И. И. Воровича

Кафедра информатики и вычислительного эксперимента

Пахомов Артём Алексеевич

РЕАЛИЗАЦИЯ ЯДРА JUPYTER NOTEBOOK ДЛЯ PASCALABC.NET С ВОЗМОЖНОСТЬЮ ИНТЕРАКТИВНОГО ВВОДА ДАННЫХ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки
02.03.02 – Фундаментальная информатика и информационные технологии

Научный руководитель –
доцент, к. ф.-м. н. Михалкович Станислав Станиславович

Допущено к защите:
заведующий кафедрой _____ Михалкович С.С.

Ростов-на-Дону – 2022

Оглавление

Постановка задачи

Разработать программу-ядро онлайн среды разработки Jupyter Notebook для PascalABC.NET. Для этого необходимо:

1. Разработать интерактивное ядро с возможностью выполнения программ, а также интерактивного ввода и вывода
2. Разработать утилиту CompilerRunner с возможностью компиляции, выполнения и перехвата ввода и вывода
3. Подготовить сценарий развертывания ядра на сервере Jupyter Hub с использованием docker-контейнера

Введение

Jupyter Notebook – это интерактивная среда разработки, представляющая собой веб-страницу с возможностью написания и выполнения кода, визуализацию результатов и использования языков разметки markdown и html. Преимуществами таких ноутбуков является то, что на них одновременно может находиться наглядное описание решаемой задачи в текстовом виде, хорошо воспринимаемом человеком, и программный код с результатами его выполнения. Ноутбуки могут найти достаточно широкое применение в учебном процессе, позволяя проводить интерактивные уроки по различным дисциплинам и не только программированию.

Любой ноутбук состоит из двух частей – непосредственно визуальной части ноутбука (далее фронтенд) и программно-аппаратной части или так называемого ядра. Нашей задачей было реализовать ядро Jupyter Notebook для PascalABC.Net, с возможностью использования основных необходимых функций, таких как выполнение программ на языке PascalABC.Net с возможностью интерактивного ввода и вывода, а также вывода графических изображений.

Ядро – это программа, которая устанавливает соединение с фронтендом ноутбука и отвечает на его запросы, такие так запрос на выполнение кода, запрос на прерывание выполнения и т. п (другие типы сообщений будут рассмотрены в основной части работы). Существует два основных способа реализации ядер:

- использование оболочки IPython, представляющей собой шаблон реализации ядра
- написание отдельной программы на любом языке программирования с самостоятельной реализацией всех необходимых функций

Изучив предметную область, а именно пользовательскую документацию по реализации ядер для Jupyter Notebook, а также реализации ядер для

различных языков программирования, находящихся в открытом доступе, было принято решение разработать собственное ядро с использованием языка программирования C#. Это решение обусловлено желанием более гибкой настройки ядра под наши нужды с возможностью разобраться в тонкостях его реализации. Также на такое решение повлияло то, что во многих других реализациях ядер, находящихся в открытом доступе, наблюдаются многочисленные проблемы, такие как отсутствие инструментов ввода, возможности прерывания выполнения программы и т. п.

1. Протокол взаимодействия компонент приложения

На сайте с документацией для разработчиков ядер [1] находится подробное описание процессов, осуществляемых на фронтенде ноутбука и необходимых для корректной работы ядра функций.

1.1. Основные понятия ZeroMQ

Взаимодействие между фронтедами и ядром осуществляется на прикладном уровне с помощью специальной сетевой библиотеки ZeroMQ [2]. Для дальнейшего описания работы ядра необходимо подробно изучить данную библиотеку.

Она представляет собой набор функций для взаимодействия между процессами, другими словами, даёт возможность работать с сокетами. Сокет – это программный интерфейс, который используется для обеспечения обмена данными между процессами. При этом процессы могут выполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой локальной или глобальной сетью. ZeroMQ позволяет абстрагироваться от возможных проблем, которые возникают при работе с сокетами на низком уровне, таких как заботы по буферизации данных, установление и подержание соединения, работа с очередями и даёт возможности для комфортной работы по обеспечению общения между процессами. Забегая вперёд, можно упомянуть, что и при разработки отдельных приложений в рамках данной работы были использованы сокеты ZeroMQ, об этом будет рассказано позже.

ZeroMQ поддерживает большинство стандартных паттернов общения, такие как pub/sub, request/reply, router/dealer и другие. Два из таких паттернов используются в сокетах Jupyter при взаимодействии фронтедов и ядер, поэтому стоит остановиться на них подробнее (само предназначение конкретных сокетов будет описано позже):

- pub/sub (издатель/подписчик) – паттерн проектирования, основанный на том, что сообщения, создаваемые «издателем» не предназначены для конкретных получателей, а содержат в себе так называемую тему (topic). При этом каждый «подписчик» может получать сообщения по одной или более темам не зависимо от конкретного «издателя».
- router/dealer – паттерн проектирования, представляющий собой усложненный вариант сокетов request/reply. Его суть заключается в том, что сокет router отслеживает каждое соединение, которое у него есть, и сообщает об этом вызывающей стороне. При этом сокет dealer не содержит в себе никаких особенных функций, а просто получает и отправляет сообщения с сокета router. Такой способ взаимодействия используется в Jupyter для того, чтобы одно ядро могло одновременно взаимодействовать с несколькими фронтендами

Отличительной чертой ZeroMQ, как было сказано выше, является простота использования. Так выглядит простейшая программа на языке C#, позволяющая наладить общение между двумя процессами:

Листинг 1. ZeroMQ программа-сервер

```
using (var responder = new ResponseSocket())
{
     responder.Bind("tcp://*:5555");
     while (true)
     {
          string str = responder.ReceiveFrameString();
          Console.WriteLine("Received Hello");
          Thread.Sleep(1000);
          responder.SendFrame("World");
     }
}
```

Листинг 2. ZeroMQ программа-клиент

```
using(var requester = new RequestSocket())
{
      requester.Connect("tcp://localhost:5555");
      int requestNumber;
      for (requestNumber = 0; requestNumber != 10; requestNum-
ber++)
      {
            Console.WriteLine("Sending Hello {0}...", requestNum-
      ber);
            requester.SendFrame("Hello");
            string str = requester.ReceiveFrameString();
            Console.WriteLine("Received World {0}", requestNumber);
      }
}
```

1.2. Взаимодействие с ядром

Как было описано ранее Jupyter Notebook состоит из двух частей – фронтенда и ядра. Для взаимодействия этих двух сущностей разработчиками Jupyter предусмотрен протокол взаимодействия.

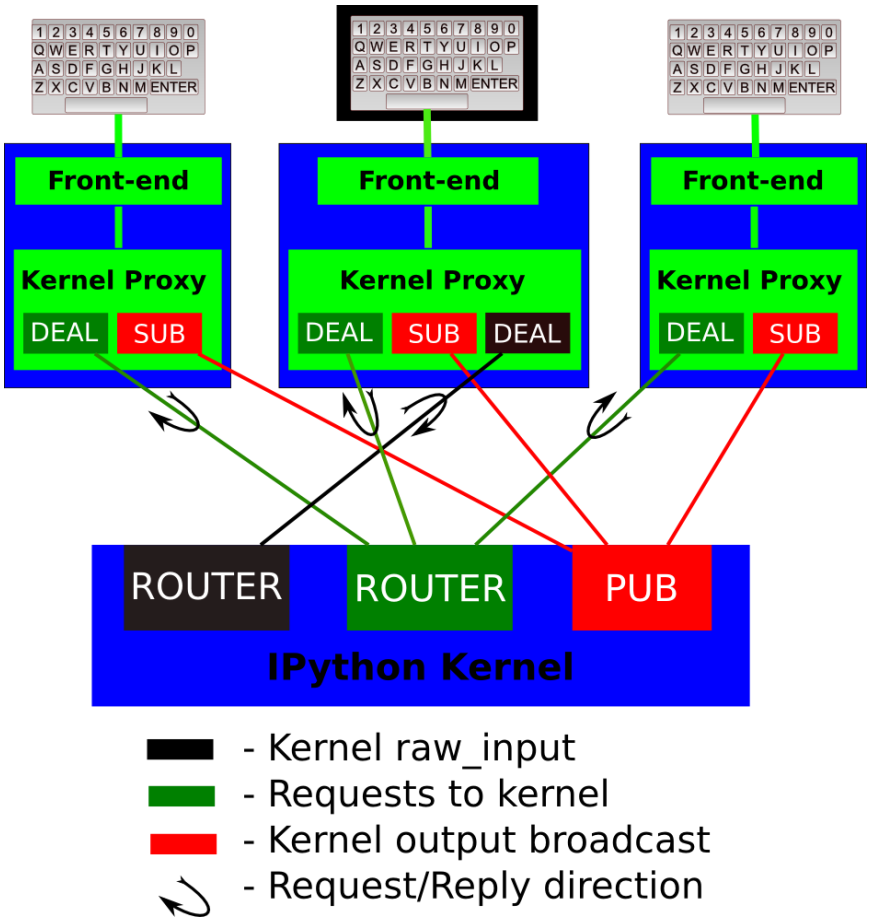

Рис. 1. Схема взаимодействия фронтендов и ядра

Все сообщения, отправляемые фронтедами и ядрами, передаются по 5 различным сокетам. Каждому сообщению типа “request” соответствует сообщение типа “reply”, представляющее ответ на данный запрос.

1. Shell-сокет – router/dealer сокет, по которому передаются основные типы сообщений:
   - kernel_info_request – первоначальный запрос на подключение фронтенда к ядру. Ответ на такой запрос – kernel_info_reply должен содержать основную информацию о ядре и реализованной версии языка
   - execute_request – запрос на выполнение программного кода. Ответ на такой запрос – execute_reply содержит в себе статус выполнения
2. Control-сокет – router/dealer сокет, по которому передаются вспомогательные сообщения:
   - shutdown_request – запрос, отправляемый фронтендом при завершении работы пользователем. Соответствующий ответ shutdown_reply содержит информацию о результате завершения работы фронтенда
   - interrupt_request – запрос, посылаемый фронтендом при желании пользователя прервать выполнение программы. Ответ interrupt_reply содержит статус прерывания.
3. IOPub-сокет – pub/sub сокет, отвечающий за вывод информации из стандартных потоков вывода и ошибок ядра на фронтенды:
   - execute_result – сообщение, отправляемое ядром для вывода результатов выполнения кода, содержащегося в запросе execute_request

- display_data и update_display_data – сообщения, предназначенные также для вывода различного вида информации на фронтенд. Сообщение без update выводит новую порцию данных, а сообщение с update обновляет часть данных

4. Stdin-сокет – router/dealer сокет, предназначенный для запросов ввода и ответов на них. Его существенное отличие от всех других сокетов, используемых в Jupyter состоит в том, что запросы тут отправляет ядро, а фронтент, то есть конечный пользователь должен на них отвечать. Существует единственный тип сообщений на этом сокете:
   - input_request – запрос, отправляемый ядром, и вызывающий на фронтенде поле, предназначенное для ввода данных (он возникает, когда в выполняемой программе есть запрос типа read()). После ввода данных в это поле фронтенд отправляет на ядро сообщение типа input_reply, содержащее в себе информацию о введённом значении.
5. Heartbeat сокет – сокет, предназначенный для того, чтобы фронтенд мог знать, работает ли связанное с ним ядро или нет. Сообщения на этом сокете представляют собой массив байтов, которые ядро должно просто отправить обратно.

1.3. Общий формат сообщений

Каждое сообщение, отправляемое фронтендом имеет определённый формат и, соответственно, ядро должно строго соблюдать данные соглашения. Каждое сообщение представляет собой последовательность 7 блоков байт. Первые три – вспомогательные блоки:

- ZeroMQ identities – префикс маршрутизации, содержащий ноль или более идентификаторов сокетов.
- <IDS|MSG> - разделитель, отделяющий идентификаторы от непосредственно сообщения

- HMAC подпись – механизм проверки подлинности сообщения. По умолчанию для вычисления этой криптографической подписи используется алгоритм хэширования sha256. Подпись состоит из конкатенации уникального ключа из конфигурационного файла с сериализованными словарями, описанными ниже

После подписи идёт собственно сообщение, представляющее собой набор четырёх ассоциативных словарей:

- Заголовок сообщения содержит в себе информацию о сообщении, такую как уникальные идентификаторы сообщения и сессии, имя текущего пользователя фронтенда, дату отправки, тип сообщения и версию протокола взаимодействия
- Родительский заголовок содержит в себе информацию тогда, когда текущее сообщение является «результатом» другого сообщения, например побочным эффектом или результатом выполнения. Тогда родительский заголовок – это копия заголовка родительского сообщения. Это необходимо учитывать при отправке сообщений-ответов из ядра. Если родителя нет, то данный словарь должен быть пустым
- Метаданные содержат в себе информацию о сообщении не являющейся основной частью содержимого. Это могут быть расширения файлом, MIME-типы и другая информация о запросе или контексте выполнения
- Основное содержимое – тело сообщения. Его структура определяется типом сообщения, указанным в заголовке. Для каждого типа сообщение его содержимое подробно описано в документации [1]

2. Установка ядра

Отдельного упоминания заслуживает установка ядра. Для работы Jupyter Notebook предварительно необходимо установить Python версии 3.8 и 3.9 (обязательно с пакетным менеджером pip). Далее с помощью командной строки необходимо инсталлировать Jupyter Lab командой pip install jupyter lab. Вместе с этим установится и Jupyter Notebook, в котором будет проходить наша основная работа. Далее необходимо скачать текущую сборку ядра PascalABC.NET из репозитория [3]. Перейдя в корневую папку локального репозитория необходимо выполнить команду

jupyter kernelspec install kernel-spec --name=PascalABC.NET

Тут стоит обратить внимание на файл спецификации ядра, определяющий корректность установки. Он представляет собой JSON файл следующей структуры:

```
{
    "argv":["C:/ProgramData/jupyter/kernels/PascalABC.NET/net5.0/ZM
QServer.exe","{connection_file}"],
    "display_name": "PascalABC.NET",
    "language": "pascal",
    "interrupt_mode": "message"
}
```

Поле argv содержит в себе путь к исполняемому файлу ядра. Этот файл будет запускаться фронтендом Jupyter при каждом новом открытии. При этом аргументом командной строки будет являться путь к файлу спецификации ядра (для этого в явном виде указана строка {connection_file}), о котором мы поговорим ниже. Поле display_name содержит в себе имя ядра, которое будет отображаться в системе. Поле language выглядит именно таким образом из-за необходимости указать MIME-тип языка программирования, для которого будет осуществляться подсветка синтаксиса. Ближайшим к PascalABC.NET типом как раз и является test/pascal. Поле interrupt_mode являет-

ся опциональным, но необходимо для корректной работы сокета control, о чем мы поговорим позже.

После выполнения всех вышеописанных действий ядро готово к использованию. Корректность его установки можно проверить, посмотрев список всех установленных ядер, с помощью команды jupyter kernelspec list. Запустить Jupyter можно с помощью команд jupyter lab или jupyter notebook.

3. Реализация ядра и вспомогательных программ

3.1. Установление соединения

В начале работы Jupyter передаёт ядру путь к файлу подключения. Он представляет собой JSON словарь следующего вида:


```
{
  "shell_port": 50091,
  "iopub_port": 50092,
  "stdin_port": 50093,
  "control_port": 50095,
  "hb_port": 50094,
  "ip": "127.0.0.1",
  "key": "1bde1de5-367dbba1925bcfe183778c3c",
  "transport": "tcp",
  "signature_scheme": "hmac-sha256",
  "kernel_name": "PascalABC.NET"
}
```


Рассмотрим подробнее каждое поле:

- пять полей с постфиксом _port, поле transport и поле ip определяют способ подключения ядра к соответствующим сокетам фронтенда с помощью ZeroMQ. Соответсвующая строка подключения тогда будет выглядеть следующим образом:

  {transport}://{ip}:{_port}

  При каждом новом запуске ядра порты выбираются случайным образом
- поля signature_scheme и key используются для создания криптографической подписи, как это было описано выше, для того чтобы другие пользователи не могли взаимодействовать с ядром

С использованием данной информации ядро устанавливает соединение с каждым из пяти портов. После этого фронтенд незамедлительно отправляет ядру сообщение типа kernel_info_request на shell-сокет. Получив данное сообщение ядро должно ответить по этому же сокету сообщением kernel_info_reply, содержащим в себе всю необходимую информацию. Если всё прошло успешно и соединение установлено, фронтенд переводится в режим ожидания действий пользователя, о чем символизирует индикатор в правом верхнем углу экрана.

3.2. Возможности работы с Jupyter

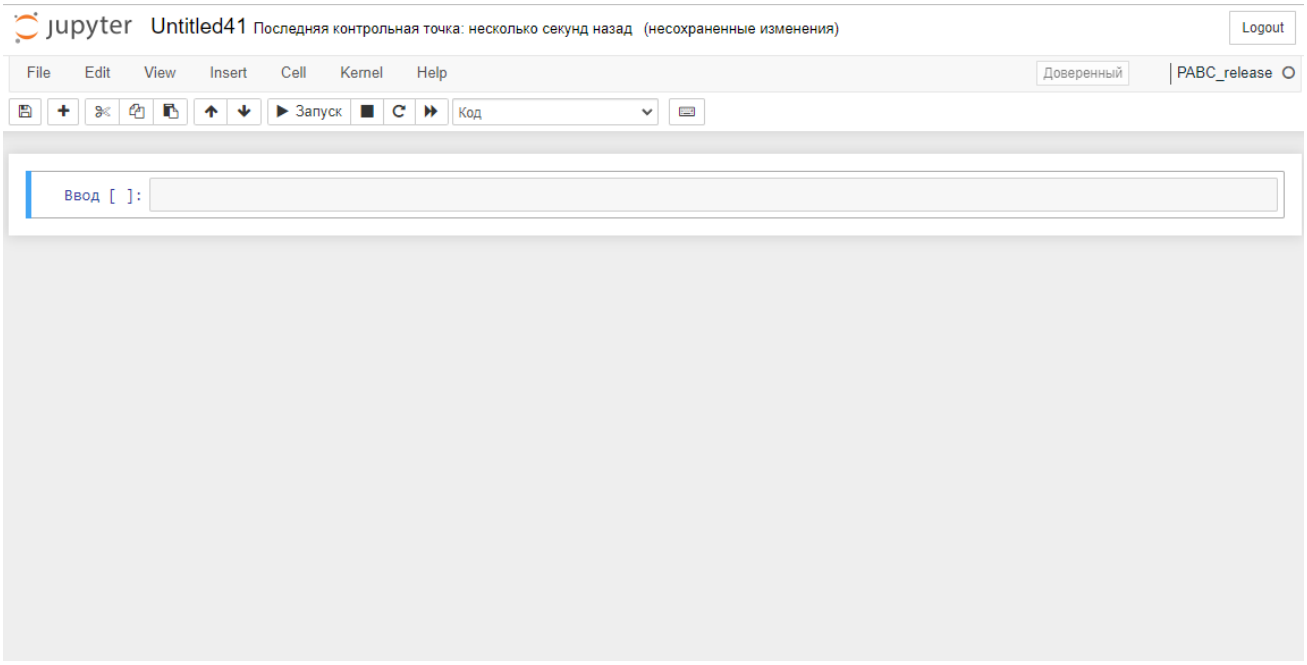

Рис. 2. Начальный вид пользовательского интерфейса

Пользовательский интерфейс предоставляет возможности для гибкой настройки внешнего вида. В основной части интерфейса расположено поле для ввода кода. Приведём пример простой программы на языке PascalABC.NET:

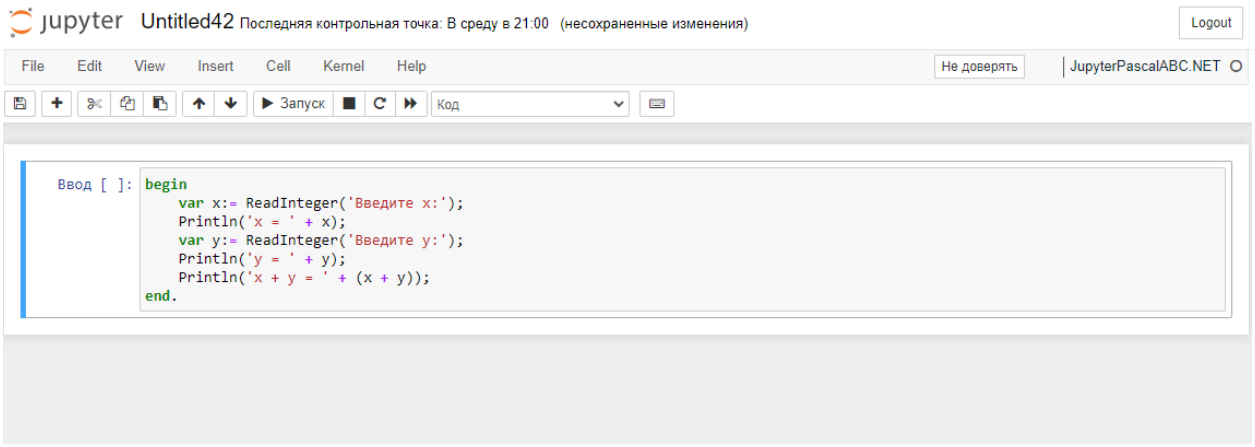

Рис. 3. Простая программа на PascalABC.NET

В следующем разделе будет описана инфраструктура реализации ядра и только после этого можно будет приступить к описанию действий, происходящих после запуска данной программы.

3.3. Утилита CompilerRunner

Для того, чтобы удобно компилировать и запускать код на языке PascalABC.NET в условиях данной работы, была разработана утилита CompilerRunner, предназначенная для компиляции и выполнения кода на языке PascalABC.NET, а также перехвата её потоков во время выполнения. Разработка велась так же на языке C#, но на платформе .NET Framework. Это приложение запускается при запуске ядра и взаимодействует с основной программой-ядром посредством всё тех же сокетов ZeroMQ. Утилита CompilerRunner получает на вход код, который мы хотим выполнить и выполняет его первоначальную подготовку к компиляции. Далее, с помощью библиотечных средств PascalABC.NET выполняется компиляция кода, с перехватом возможных ошибок компиляции. После этого, если компиляция прошла успешно, выполняется запуск созданного .exe файла. На этом моменте стоит остановиться подробнее.

Также важно упомянуть, что при использовании утилиты CompilerRunner, а не просто компиляции непосредственно в ядре, можно добиться значительного ускорения времени компиляции. При этом первая после запуска ядра компиляция будет немного дольше. Такое ускорение связано с тем, что,

запустившись один раз при запуске ядра, утилита CompilerRunner останется в памяти вместе со всеми вспомогательными модулями и, в отличии от простого запуска компилятора при необходимости, не будет необходимости каждый раз загружать эти вспомогательные модули в оперативную память.

Для наших нужд, а именно для того, чтобы успешно перехватывать сообщения из потока вывода и потока ошибок, а также вспомогательные сообщения, был доработан PascalABC.NET модуль RedirectIOMode. Часть его реализации будет представлена в приложении. Суть доработок заключается в том, что

1. Необходимо из выполняемого процесса перехватывать запросы на ввод данных в программу. Для этого были переопределены функции PascalABC.NET осуществляющие ввод. К ним был добавлен функционал печати специального сообщения
   [READLNSIGNAL] в поток ошибок. Это позволяет легко перехватить это сообщение из выполняемого процесса и послать ядру сообщение о том, что необходимо запросить ввод у пользователя ноутбука
2. Из-за особенностей реализации класса Process, предназначенного в C# для запуска процессов, перехват вывода из потоков вывода и ошибок осуществляется только тогда, когда сообщение заканчивается управляющим символом 0A или \n, то есть переводом строки. Из-за этого текст, напечатанный с помощью функции типа Print(), то есть таких, которые не переводят вывод на новую строку, не появляется на фронтенде в момент печати. А необходимость учитывать такие случаи возникла из-за возможности в PascalABC.NET писать конструкции типа
   var x := ReadInteger('Введите x: ')
   которые представляют собой приглашение ко вводу с дальнейшим запросом ввода. Для того, чтобы учитывать такой случай были переопределены стандартные методы, реализующие вывод

без перевода строки. При этом в них были добавлены специальные сообщения [NEWLINE] и [FAKELINE], отправленные с переводом строки и позволяющие корректно обработать их при перехвате.

Итак, мы имеем приложение, позволяющее компилировать и выполнять код, перехватывая при этом вывод и запросы на ввод. Рассмотрим действия этой программы при, собственно, перехвате вывода и запроса на ввод.

При перехвате вывода, необходимо сразу же, используя соответствующий выводной сокет передать перехваченное сообщение в ядро для того, чтобы напечатать это сообщение на фронтенде. Необходимо также не забыть корректно обработать присутствие в перехваченном сообщении специальных сообщений [NEWLINE] и [FAKELINE] и избавиться от них.

При перехвате запроса на ввод необходимо послать по отдельно выделенному под это сокету сообщение о том, что нам необходимо получить ввод от пользователя фронтенда. После того как на фронтенде будет введено какое-то сообщение ядро по ещё одному отдельному сокету отправляет результат. Для его получения в утилите CompilerRunner в отдельном потоке крутится бесконечный цикл, ожидающий введённое сообщение. При получении сообщение его необходимо направить во входной поток выполняемого процесса.

Также необходимо помнить об ошибках времени выполнения, которые будут попадать в поток ошибок выполняемого процесса. При получении сообщения о такой ошибке, необходимо отправить в ядро сообщение с текстом ошибки для того, чтобы информировать пользователя.

Также в утилите CompilerRunner предусмотрена функция завершения выполняемого процесса. Цель данной функции будет описана ниже.

Также для корректного завершения утилиты CompilerRunner при прекращении работы ядра предусмотрен отдельный heartbeat сокет, по которому приложение обменивается с ядром сообщениями каждую секунду. При от-

сутствии такого сообщения от ядра в течение более чем одной секунды утилита CompilerRunner завершает свою работу.

3.4. Процесс выполнения кода в инфраструктуре приложения

В этом разделе будет описан процесс обработки запроса фронтенда на выполнение кода. Напомним, что это запрос execute_request, приходящий на сокет shell и содержащий в теле запроса код, который необходимо выполнить. Кроме кода, content раздел запроса может содержать дополнительные поля, но они не представляют практического значения в рамках нашей задачи.

Здесь стоит упомянуть особенность реализации любого ядра для Jupyter ноутбуков. После получения любого запроса от фронтенда ядро должно немедленно отправить в ответ сообщение типа status на сокет IOPub, содержащее текущий статус ядра, а именно один из трёх: busy, idle или starting. В данном случае необходимо отправить статус busy, обозначающее, что ядро занято выполнением данного запроса. После выполнения любого запроса необходимо отправить сообщение со статусом idle, обозначающим, что ядро освободилось и готово принимать новые запросы.

Итак, после получения запроса на выполнение кода и отправки статуса «занят» ядро в первую очередь должно ответить на данный запрос соответствующим сообщением типа reply. Сообщение execute_reply содержит в себе единственное поле execution_count – глобальный счётчик, значения которого отображаются рядом с ячейками в ноутбуке.

Далее осуществляется подключение к полученному коду на языке PascalABC.NET модуля RedirectIOMode и отправка данного сообщения в утилиту CompilerRunner. Утилита CompilerRunner, получив данное сообщение, создаёт временный файл с расширением .pas и приступает к его компиляции. В зависимости от результатов компиляции обратно в ядро может быть отправлено либо сообщение об ошибке компиляции, либо специальное сообщение

[OK], обозначающее, что компиляция прошла успешно. Получив ответ от утилиты CompilerRunner, ядро может определить прошла ли компиляция, и, если нет, вывести на фронтенд сообщение об ошибке компиляции. В это время утилита CompilerRunner запускает полученный после компиляции .exe файл с возможностью перехвата вывода сообщений из стандартных потоков вывода и ошибок.

Перехватив сообщение из стандартного потока вывода, утилита CompilerRunner обрабатывает его (корректно избавляясь от вспомогательных сообщений [NEWLINE] и [FAKELINE]), добавляет данное сообщение в буфер и отправляет содержимое буфера в ядро. Получив это сообщение, ядро незамедлительно печатает его на фронтенд с помощью сообщения execute_result, отправляемого на сокет IOPub и содержащего в себе результат выполнения кода. Необходимо каждый раз после получения сообщений из потока вывода обновлять вывод на фронтенде, для этого сообщения и накапливаются в буфере внутри утилиты CompilerRunner. Сообщение из потока ошибок может быть двух типов: специальное сообщение [READLNSIGNAL] и сообщение, содержащее ошибку времени выполнения. Перехватив сообщение [READLNSIGNAL] утилита CompilerRunner просто переправляет его в ядро, которое после получения отправляет на фронтенд запрос input_request, предназначенный для сокета stdin. Получив в ответ сообщение input_reply с введённым в поле ввода текстом ядро немедленно отправляет этот текст в утилиту CompilerRunner, где он попадает в поток ввода исполняемого процесса и процесс продолжает выполнение.

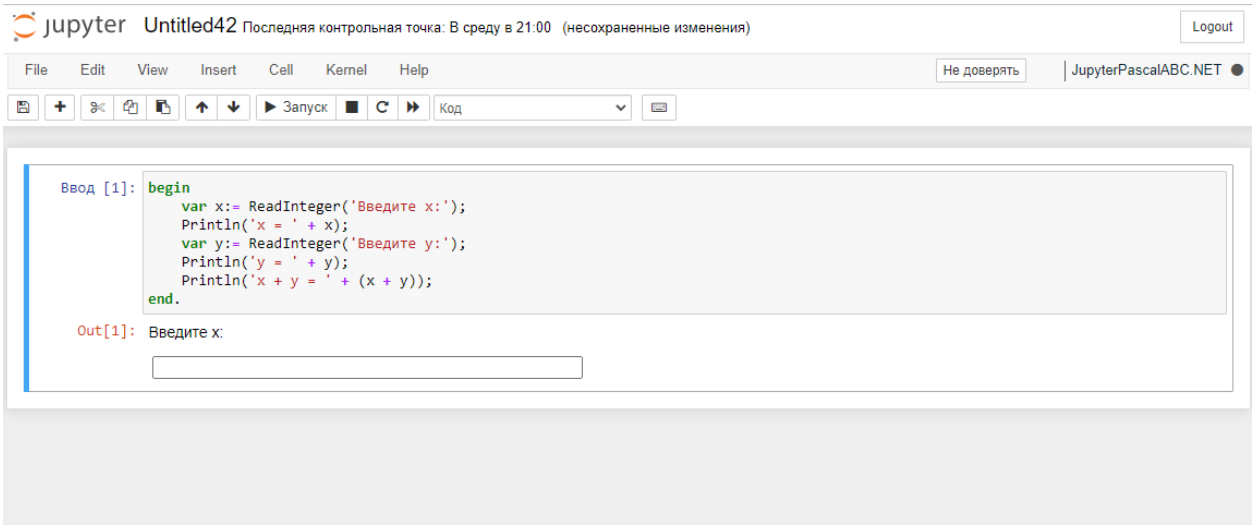

Рис. 4. Поле для ввода данных

Если в поток ошибок попадает сообщение об ошибке времени выполнения, то оно направляется в ядро и выводится на фронтенд. При этом исполняемый процесс завершает выполнение.

После того как процесс завершает выполнение, в ядро отправляется специальное сообщение [END], для ядра это означает, что необходимо отправить на фронтенд статус idle и очистить различные вспомогательные переменные.

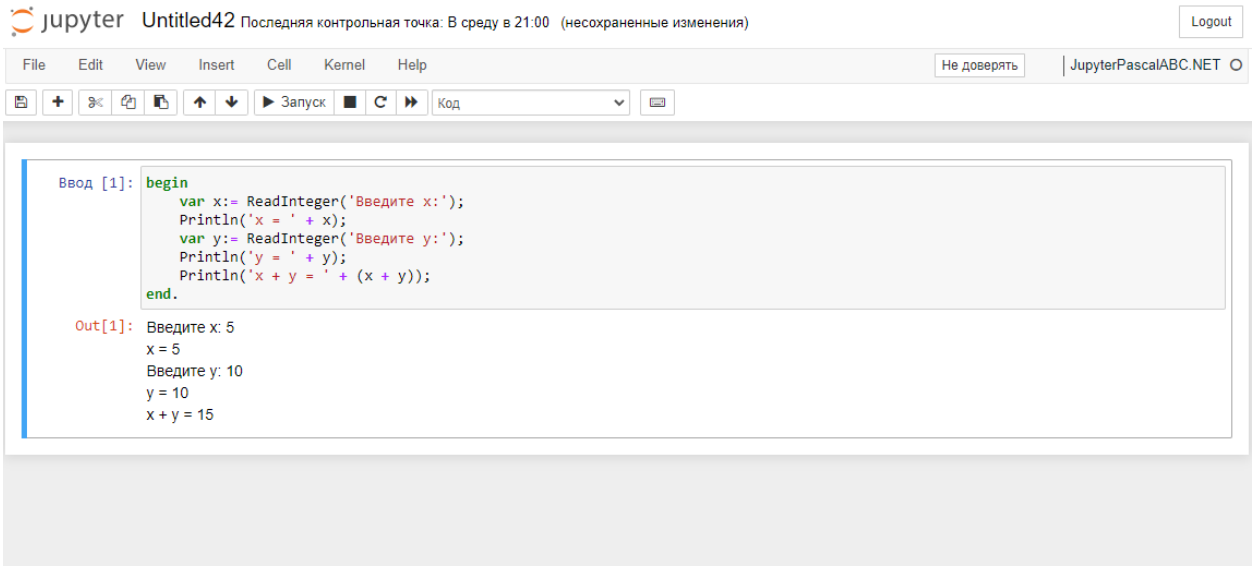

Рис. 5. Результат работы программы

3.5. Прерывание выполнения

Во время выполнения программы часто может возникнуть необходимость прервать её выполнение, к примеру, если программа зациклилась или обнаружилась ошибка в логике программы. Такая возможность предусмотрена в ядре. Для этого используется сообщение interrupt_request сокета control, которое отправляется в ядро при нажатии соответствующей кнопки на фронтенде.

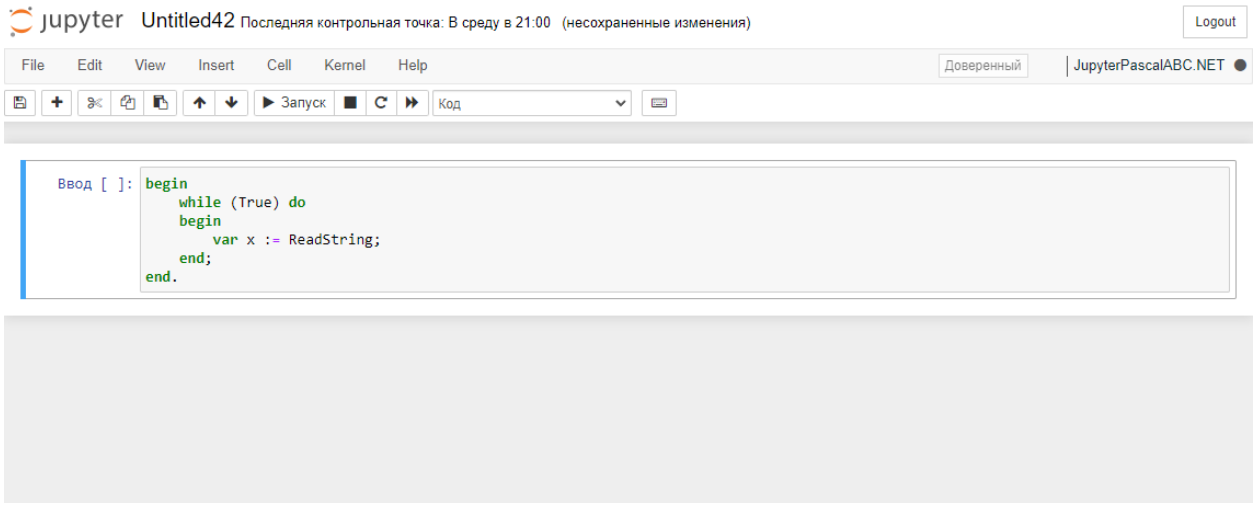

Рис 5. Простейший пример зацикливающейся программы

При получении запроса на прерывание необходимо ответить сообщением interrupt_reply, содержащим в теле поле со статусом прерывания. После этого в утилиту CompilerRunner отправляется специальное сообщение [BREAK] и устанавливается статус idle. Утилита CompilerRunner при получении сообщения [BREAK] завершает выполнение исполняемого процесса.

После всех этих действий ядро готово к получению нового запроса на выполнение.

4. Развертывание ядра в Jupyter Hub

Jupyter Hub – это многопользовательская версия ноутбука, предназначенная для групп людей. Также Jupyter Hub устанавливается на сервере, поэтому пользователи не обременяются задачами по установке ядер. Различные типы пользователей, в том числе студенты и преподаватели получают возможность выполнять требуемые задачи на общих ресурсах, эффективно управлять которыми могут системные администраторы.

Для установки уже готового ядра на сервер с операционной системой Linux, ядро было переведено на платформу .NET Framework. Цель перевода всего проекта на .NET Framework заключается в нежелании сильно увеличивать объём инсталлятора платформой .NET. Теперь, после перевода всего проекта на платформу .NET Framework для работы ядра необходима только она.

Одной из особенностей Jupyter Hub является то, что он самостоятельно заботится о том, как запускать процессы для нескольких пользователей, а именно: он выделяет под каждый созданный ноутбук отдельный процесс ядра. Это означает, что нет необходимости реализовывать эту логику самостоятельно.

Одной из самых больших проблем, возникших при попытках запуска ядра в операционной системе Linux, стала проблема, связанная с тем, что управляющие символы CR и LF в Windows и Linux обрабатываются поразному. Эта особенность операционных систем существует с момента их создания по различным историческим причинам. Суть её заключается в том, что в Unix/Linux маркером перехода на новую строку был выбран единственный символ LF, а в MS-DOS и, следовательно, Windows, как её наследника маркером перехода на новую строку стало сочетание CR+LF.

Чтобы развернуть ядро в Jupyter Hub собранное под .NET Framework ядро было перенесено на сервер, конфигурационный файл ядра был скорректирован с учётом запуска под Mono, а также были написаны специальные

инструкции для docker-контейнера для того, чтобы специально выделенный для ядра контейнер знал, какую последовательность действий ему нужно выполнять при установке ядра и при каждом запуске ядра.

```
WORKDIR /JupyterPABC_kernel
RUN wget -nv http://pascalabc.net/downloads/JPABC.zip && \
unzip JPABC.zip
ENV PATH="/pabc:${PATH}"

RUN DST="args = \['mono', r'\/pabc\/pabcnetcclear\.exe',
source_filename]"
RUN cd /JupyterPABC_kernel && sudo jupyter kernelspec install
kernel-spec —name=pabc_release
```

Листинг 5. Инструкции docker-контейнера

Заключение

В рамках дипломной работы были разработаны:

- утилита CompilerRunner с возможностью перехвата ввода и вывода
- интерактивное ядро с возможностью выполнения программ, а также интерактивного ввода и вывода
- сценарий для развертывания ядра на сервере Jupyter Hub

Литература

1. Сайт с документацией о Jupyter ядрах. – URL: https://jupyter-client.readthedocs.io/en/stable/ (дата обращения 05.06.2022).
2. Сайт с документацией о библиотеке ZeroMQ. – URL: https://zguide.zeromq.org/ (дата обращения 10.03.2022).
3. Репозиторий с последней сборкой ядра. – URL: https://github.com/Looken15/JupyterPascalABC.NET (дата обращения 05.06.2022).
4. Михалкович С.С., Баглий А.П., Кобзарь Д.В., Пахомов А.А Jupyter-ноутбуки для PASCALABC.NET и их использование в учебном процессе / XXIX Научная конференция «Современные информационные технологии: тенденции и перспективы развития». Материалы конференции. Ростов-на-Дону, 2022.

Приложение 1. Реализация класса RedirectIOMode

```
procedure __ReadSignalOISystem.writeln;
begin
  inherited writeln;
  Console.WriteLine('[NEWLINE]');
end;
procedure __ReadSignalOISystem.write(obj: object);
begin
  inherited write(obj);
  Console.WriteLine('[FAKELINE]');
end;
```

Листинг 3. Реализация перехвата вывода

```
procedure SendReadlnRequest;
begin
  ReadlnSignalSended := true;
  if not IsInputPipedOrRedirectedFromFile then
    WriteToProcessErrorStream(ReadlnSignalCommand);
end;
function __ReadSignalOISystem.peek: integer;
var i: integer;
begin
  if not ReadlnSignalSended then
    SendReadlnRequest;
  i := inherited peek;
  result := i;
end;
function __ReadSignalOISystem.read_symbol: char;
var c: char;
begin
  if not ReadlnSignalSended then
    SendReadlnRequest;
  c := inherited read_symbol;
  if (LastReadSymbol=N) and (c=R) then
    ReadlnSignalSended := false;
  result := c;
end;
function __ReadSignalOISystem.ReadLine: string;
begin
  if not ReadlnSignalSended then
    SendReadlnRequest;
  var s := inherited ReadLine;
  ReadlnSignalSended := false;
  Result := s;
end;
```

Листинг 4. Реализация перехвата запроса на ввод