Лабораторная работа 2. Моделирование клиент-серверной системы удаленных вычислений (методом Map-Reduce)

Целью лабораторной работы является моделирование системы удаленных вычислений, т.е. системы, предоставляющей свое машинное время для решения трудоемких задач по клиентским запросам. Имеются сервера, считающиеся системами с высокой вычислительной мощностью, и клиент, решающий некоторую np-полную задачу путем разделения ее на подзадачи, раздачи частей по серверам (map) и дальнейшего сокращения результатов (reduce).

В качестве решаемой задачи рекомендуется брать np-полные задачи, легко разделяющиеся на более мелкие задачи, например, вычисление простых чисел, вычисление совершенных чисел и пр.

Используемые технологии: ввод - вывод, сетевое взаимодействие, многопоточность, сериализация, загрузчик классов, reflection

Требуется написать приложение типа client-server.

Сервер является приложением, принимающим соединения по порту, указанному в его конфигурационных параметрах. При получении запроса сервер устанавливает соединение с клиентом в отдельном потоке. Сервер может поддерживать соединение одновременно с произвольным количеством клиентов, но одновременно исполняет задачи только одного клиента.

Требования к серверу. Сервер умеет выполнять два действия = регистрация задачи клиента и выполнение задачи клиента. Регистрация задачи клиента заключается в получении по сети class-файла и загрузки его в java-машину. По условиям задачи считается, что сервер знает интерфейс class-файла.

Требования к клиенту. Клиент должен уметь регистрировать произвольное количество серверов. Клиент умеет передавать серверу установленное соединение класс-файл, представляющий задачу, после чего сервер регистрирует этот класс-файл и считается способным исполнять код клиента. По условию интерфейс класс-файла известен. Далее клиент опрашивает сервера на предмет – свободны ли они, если свободны то отправляет им части задачи и собирает ответы.

Доп. Задание. Реализовать возможность загрузки кода в виде текста на javascript.

Доп. Задание. Использовать ForkJoinPool

При демонстрации лабораторной работы необходимо продемонстрировать работоспособность системы при одновременно подключении клиента к десяти серверам. Клиент должен информировать пользователя о прогрессе работы. Вычисления должны производиться с неограниченной точностью.

Далее будет приведена информация о некоторых необходимых для реализации задачи технологиях.

Map-Reduce

Алгоритм Map-Reduce не следует путать с одноименным фреймворком распределенных вычислений, хотя в определенном смысле целью лабораторной работы является построение упрощенной модели этого фреймворка.

Идея алгоритма состоит в том, что клиентский процесс делит задачу на куски и рассылает серверам, которые генерируют ему ответы. Эти ответы клиент склеивает в конечный результат.

Например, перед клиентом стоит задача найти все простые числа от одного до ста. Допустим, у клиента имеется три подключенных сервера вычислений. Клиент делит задачу, например, на десять диапазонов по 10 чисел и рассылает серверам, собирая их ответы (шаг Map). Получившиеся ответы склеиваются в конечный результат (шаг Reduce)

Ввод и вывод

В java существует две основных технологии ввода и вывода, а именно, “старый” ввод вывод через потоки (для бинарных файлов) либо через объекты Reader и Writer (для текстовых файлов) и “новый” ввод-вывод NIO. NIO является новым весьма условно, так как впервые появился в Java 1.4.2; тем не менее в Java 7 NIO был существенно переработан. В процессе выполнения данной лабораторной работы у пользователя возникает задача чтения файла, которая может быть решена с помощью любого из этих подходов.

Тем не менее, рекомендуется использовать NIO по причине краткости кода. Следующий пример иллюстрирует применение NIO для чтения коротких файлов на примере файла c:/myfile.dat

```
byte[] data=Files.readAllBytes(Paths.get("c:/myfile.dat"));

System.out.println("Read "+data.length+" bytes");
```

Установление сетевого подключения.

Сетевое подключение между двумя процессами устанавливается через так называемый механизм сетевых сокетов. Сокет – это абстракция, состоящая из сетевого адреса и сетевого порта. Как внутри устроен сокет для решения задачи установления сетевого соединения знать абсолютно не нужно, этот вопрос полностью отдается на совесть операционной системы. Для нас важно знать, что сокет умеет устанавливать соединение с другим сокетом, адрес и порт которого мы знаем,

после чего возвращает нам два потока – поток ввода и поток вывода, которые могут использоваться для передачи информации от клиента серверу и назад.

В соединении один из сокетов всегда должен быть клиентским, а другой – серверным. Клиентский сокет имеет тип java.net.Socket. Типовой код для установления соединения выглядит так:

```
SocketAddress addr = new InetSocketAddress(адрес_сервера, номер_порта);
Socket client = new Socket();
try {
   client.connect(addr, таймаут_на_подключение);
   if (!client.isConnected()) {
      //сюда мы попадаем, если соединение не установилось
      System.out.println("Сервер не отвечает");
      return;
   }
   InputStream inStream = client.getInputStream();
   OutputStream outStream = client.getOutputStream();
  //Здесь идет основной код программы
} finally {
   try {
      client.close();
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Серверный сокет имеет тип java.net.ServerSocket. Логика работы этого сокета заключается в том, что он вызывает метод bind, связывающий его с сокетом, а затем в бесконечном цикле вызывает метод accept. Каждый раз, когда серверный сокет получает запрос на соединение, он создает новый **клиентский** сокет на одном из свободных портов операционной системы, и переадресует входящий запрос на него, после чего возвращает его в качестве результата вызова accept. Серверный код должен взять этот новый сокет и передать отдельному потоку – обработчику команд, после чего продолжить бесконечный цикл.

Типичный пример кода сервера:

```
try{
 java.net.ServerSocket serverSocket=new java.net.ServerSocket();
```

```
    SocketAddress address= new
InetSocketAddress(Inet4Address.getByName(адрес_Сервера),порт_Сервера);

    serverSocket.bind(address);

    for(;;){

        Socket s=serverSocket.accept();

        //Здесь идет код по перемещению обработки в другой поток

      }

    }catch (Exception e){

       e.printStackTrace();

    } finally {

        serverSocket.close();

    }
```

Classloader и Reflection

Каким образом мы можем взять произвольный поток байт, преобразовать его в класс, создать экземпляр этого класса и выполнить его методы? Эта задача решается с помощью технологий classloader и reflection

Classloader – загрузчик классов – отвечает за преобразование потока байт в определение класса. В случае, если класс хранится каким-либо нестандартным методом – например, лежит на сети, в виде xml, зашифрован, а также в любых других нестандартных вариантах, может потребоваться написание собственного загрузчика классов, способного к получению данных из нестандартного места. В нашем случае нам надо преобразовать только поток байт, пришедший по сети, и для этого мы можем использовать стандартный загрузчик классов ClassLoader.getSystemClassLoader().

У полученного загрузчика классов есть два интересующих нас метода. Сперва необходимо вызвать метод defineClass(), который преобразует поток байт в класс. Затем необходимо вызвать метод resolve(), который собственно, подготовит полученный класс к использованию.

С этого момента мы имеем класс, с которым можем работать через технологию Reflection

Сперва необходимо получить объект класса Class, представляющий данный класс. Это делается вызовом вида

```
Class c = Class.forName("имя класса");
```

Этот объект может ответить на вопрос, какие в классе есть методы и переменные, соответственно, с помощью методов getConstructors(), getMethods() и getFields(). Наша задача состоит в 1) конструировании экземпляра данного класса и 2) вызове метода execute().

Создать новый объект заданного класса можно двумя способами. Первый – вызвать c.newInstance(), что приведет к вызову конструктора по умолчанию. Второй – получить ссылку на конструктор с помощью вызова getConstructors() или аналогичного, и вызвать метод newInstance уже у этого конструктора.

Полученный объект будет объектом данного класса, но, так как на этапе компиляции программы мы про этот класс ничего не знаем, нам по прежнему придется работать через reflection. Для того, чтобы вызвать определенный метод, необходимо найти его с помощью метода getMethods(), после чего вызвать на полученном методе invoke().

Таким образом, две упомянутые технологии – ClassLoader и Reflection – позволяют преобразовать поток байт в исполняемый код и выполнить этот код.

Сериализация

Сериализация – запись и чтение объектов из потока. Для сериализации используются специальные классы, представляющие поток – ObjectInputStream и ObjectOutputStream.

Объект любого класса, который реализует интерфейс java.lang.Serializable, может быть сохранен в поток и восстановлен из потока с помощью методов writeObject и readObject. Некоторые классы, для которых напрямую не указываются предки, также реализуют этот интерфейс, например, массивы. Важно знать, что все классы коллекций (списки, словари, множества) реализуют этот интерфейс, таким образом, сохранение коллекции или массива возможно. Если все элементы этой коллекции или массива поддерживают интерфейс Serializable.

При сериализации в поток сохраняются и потом восстанавливаются из потока все поля, которые не объявлены статическими и не имеют модификатора transient. В случае, если сериализуемый класс наследуется от класса, не реализующего интерфейс Serializable, то последний должен иметь конструктор по умолчанию (т.е. без аргументов), который будет вызван для инициализации всех членов класса, унаследованных сериализуемым классом. Десериализованного класса конструктор и секции инициализации не вызываются. Члены класса, объявленные как transient, будут инициализированы значениями по умолчанию.

В некоторых ситуациях программист может посчитать, что его не устраивают стандартные средства сериализации и десериализации. В таком случае он может переопределить то, как они выполняются для объекта, с помощью переопределения следующих методов:

```
private void writeObject(java.io.ObjectOutputStream out)
     throws IOException
 private void readObject(java.io.ObjectInputStream in)
     throws IOException, ClassNotFoundException;
 private void readObjectNoData()
     throws ObjectStreamException;
```

В некоторых случаях может потребоваться сериализовать в поток один класс, а восстановить другой. Для этого программист может переопределить методы

```
Object writeReplace() throws ObjectStreamException;
Object readResolve() throws ObjectStreamException;
```

Первый должен быть переопределен у сериализуемого класса, а второй – у десериализуемого.

В некоторых ситуациях может оказаться, что класс, который был сериализован в поток, на другой стороне имеет другую версию (например, вследствие изменений при разработке клиента, в то время как сервер не менялся). При десериализации с целью проверки совместимости проверяется версия класса, которая генерируется автоматически. Соответственно, если код изменился, возможны ошибки, связанные с проверкой версии. Для того, чтобы указать, что это тот же самый класс, необходимо указать его версию сериализации напрямую, например

```
static final long serialVersionUID = 42L;
```