

Простое CORBA приложение — своими руками

Александр Гордин

Технология CORBA — это стандарт написания распределенных приложений, предложенный консорциумом OMG (Open Management Group). Создавая CORBA-объекты, мы можем, например, существенно уменьшить время решения задач, требующих выполнения большого объема вычислений. Это возможно благодаря размещению CORBA-объектов на разных машинах. Каждый удаленный объект решает определенную подзадачу, тем самым разгружает клиент от выполнения лишней работы.

1. Взаимодействие объектов в архитектуре CORBA

Рассмотрим взаимодействие объектов в архитектуре CORBA

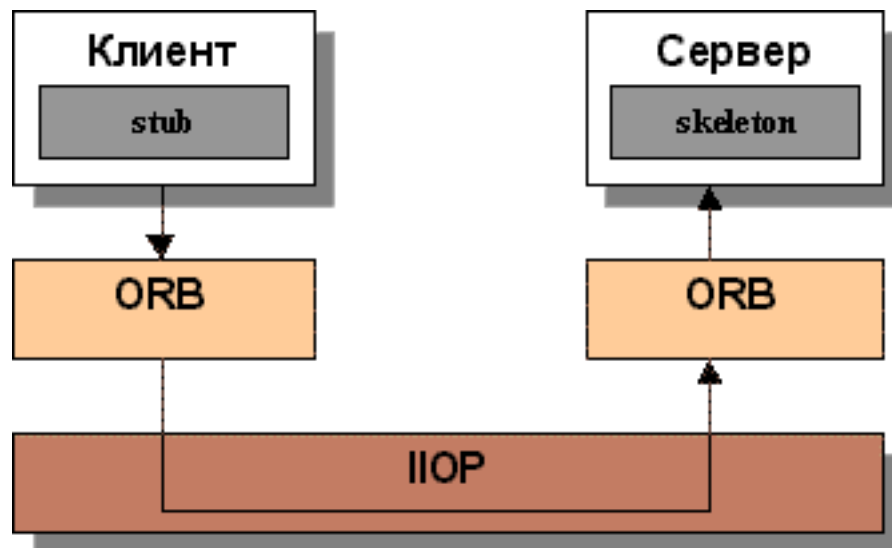


Рисунок 1. Взаимодействие объектов в архитектуре CORBA.

Основу CORBA составляет объектный брокер запросов (Object Request Broker). ORB управляет взаимодействием объектов в распределенной сетевой среде. ИИОР (Internet Inter-ORB Protocol) — это специальный протокол взаимодействия между ORB.

В адресном пространстве клиента функционирует специальный объект, называемый заглушкой (stub). Получив запрос от клиента, он упаковывает параметры запроса в специальный формат и передает его серверу, а точнее скелету.

Скелет (skeleton) — объект, работающий в адресном пространстве сервера. Получив запрос от клиента, он распаковывает его и передает серверу. Также скелет преобразует ответы сервера и передает их клиенту (заглушке).

Для того, чтобы написать любое приложение CORBA используя технологию Java, необходимо иметь две вещи — это установленный пакет JDK1.2 и компилятор `idltojava`. JDK предоставляет набор классов для работы с CORBA объектами, а `idltojava` производит отображение языка IDL в Java.

2. Создание CORBA приложения

2.1. Написание интерфейса

Создание CORBA приложения на Java начинается с написания интерфейса для удаленного объекта, используя язык описания интерфейсов (Interface Definition Language, IDL).

Создадим файл `test.idl`

```
module testApp {
    interface test
    {
        long count(in string msg);
    };
};
```

Данный интерфейс описывает лишь один метод **count**. Причем, нам не важно, что делает этот метод, главное мы определяем, что он есть, определяем какие у него входные и выходные параметры.

Воспользуемся компилятором `idltojava`.

```
idltojava Hello.idl
```

Замечание:

Данный компилятор по умолчанию использует препроцессор языка C++, поэтому что бы не мучится с сообщением об ошибке **имя команды или файла указано неправильно** (в переменной окружения CPP должен быть прописан путь к нему) отключим его использование, установив флаги.

```
idltojava -fno-cpp Hello.idl
```

Результат работы данной программы можно представить в виде схемы.

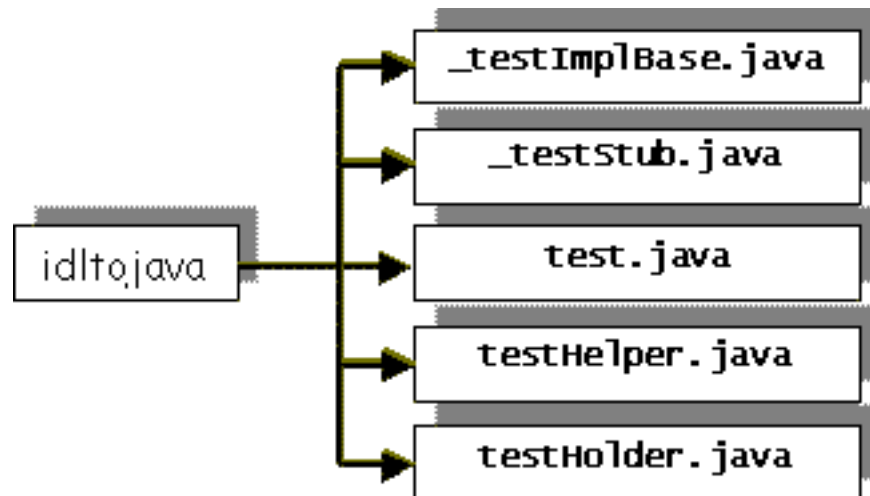


Рисунок 2. Работа `idltojava` компилятора.

В текущей директории появилась новая папка **testApp**, в которой содержатся пять `java`-файлов. Каждый из них имеет свое назначение.

_testImplBase.java — абстрактный класс, который представляет собой не что иное, как скелет сервера (skeleton) и обеспечивает функциональность сервера;

_testStub.java — класс, реализующий заглушку (stub) клиента. Обеспечивает функциональность клиента;

test.java — класс, содержащий описание интерфейса `test` на языке `Java`;

testHelper.java и **testHolder.java** — классы, предоставляющие вспомогательные

функции для CORBA объектов.

Теперь наша задача — написать класс, реализующий интерфейс `test`. Такие классы должны называться так, чтобы в их имени было слово “Servant”, в нашем случае это будет `testServant`.

```
class testServant extends _testImplBase
{
    public int count(String msg)
    {
        return msg.length();
    }
}
```

Обратите внимание, что этот класс унаследован от `_testImplBase`. Как видно, здесь реализован метод `count`, который в данном примере считает количество букв в принятом сообщении.

2.2. Написание серверной части

Теперь перейдем непосредственно к написанию серверной части приложения.

Первое что мы делаем, создаем ORB. Затем создаем экземпляр класса удаленного объекта (`testServant`) и регистрируем его в ORB. Далее вызываем специальную службу имен (`NameService`) и регистрируем в ней имя удаленного объекта, чтобы клиент смог его найти (есть и другой способ нахождения удаленного объекта, но о нем чуть позже).

Рассмотрим подробнее эти этапы.

1. Создание и инициализация ORB. Производится вызовом статического метода `init` класса ORB

```
ORB orb = ORB.init();
```

2. Создание экземпляра класса удаленного объекта и регистрация его в ORB

```
testServant testRef = new testServant();
orb.connect(testRef);
```

3. Получение контекста имен (NamingContext)

```
org.omg.CORBA.Object objRef =  
  
orb.resolve_initial_references("NameService");  
  
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

в первой строчке мы получаем объектную ссылку на службу имен (NameService). Но фактически это обыкновенный CORBA-объект и для того, чтобы использовать его как контекст имен (NamingContext), необходимо вызвать метод **narrow** класса **NamingContextHelper**, который как бы конкретизирует данный CORBA-объект.

4. Регистрация имени удаленного объекта (testServant)

Как было сказано раньше регистрация имени производится для того чтобы клиент смог найти удаленный объект. Этой цели служит функция **rebind(NameComponent[] nc, Object obj)** интерфейса **NamingContext**.

```
NameComponent nc = new NameComponent("test", "");  
    //первый параметр  
    //указывает имя объекта,  
    //второй нам использовать не обязательно  
NameComponent path[] = {nc};  
ncRef.rebind(path, testRef);
```

5. Ожидание запросов от клиента.

```
java.lang.Object sync = new java.lang.Object();  
    synchronized (sync) {  
        sync.wait();  
    }
```

После того как сервер обработает запрос от клиента и выполнит метод **count** он снова перейдет в состояние ожидания.

Теперь сервер готов к работе

Листинг 1. testServer.java

```
import testApp.*;  
import org.omg.CosNaming.*;  
import org.omg.CosNaming.NamingContextPackage.*;  
import org.omg.CORBA.*;
```

```
import java.lang.*;

class testServant extends _testImplBase
{
    public int count(String msg)
    {
return msg.length();
    }
}

public class testServer {

    public static void main(String args[])
    {
        try
        {
            ORB orb = ORB.init(args, null);

            testServant testRef = new testServant();
            orb.connect(testRef);

            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef =
                NamingContextHelper.narrow(objRef);

            NameComponent nc =
                new NameComponent("test", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, testRef);

            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait()
            }

        }
        catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

```
    }  
  }  
}
```

Обратите внимание на то, что все операции выполняемые над CORBA-объектами заключены в try-catch блок.

2.3. Написание клиентского кода

Перейдем к написанию кода для клиента.

Основные шаги написания клиентского приложения

1. Создание и инициализация ORB
2. Получение контекста службы имен (**NamingContext**)
3. Нахождение удаленного объекта
4. Вызов метода **count**.

Как видно, первые два пункта совпадают с этапами создания серверного приложения, поэтому рассматривать их не будем.

Третий пункт реализуется тоже достаточно просто. Создается объект **NameComponent**. Вызывается метод **resolve(NameComponent[] path)**, который отыскивает по имени удаленный объект (стандартный CORBA-объект). При помощи метода **narrow(org.omg.CORBA.Object obj)** класса **testHelper** (сгенерированного **idltojava** компилятором) получаем объектную ссылку на интерфейс **test**.

```
NameComponent nc = new NameComponent("test", "");  
NameComponent path[] = {nc};  
org.omg.CORBA.Object obj= ncRef.resolve(path);  
test testRef = testHelper.narrow(obj);
```

Теперь можно вызывать метод **count**

```
String msg = "try to count";  
int count = testRef.count(msg);
```

Листинг 2. testClient.java

```
import testApp.*;
```

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import java.lang.*;

public class testClient
{
    public static void main(String args[])
    {
        try
        {
            ORB orb = ORB.init(args, null);
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");

            NamingContext ncRef =
                NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("test", "");
            NameComponent path[] = {nc};
            test testRef =
                testHelper.narrow(ncRef.resolve(path));

            String msg = "try to count";
            int count = testRef.count(msg);
            System.out.println
                ("number of chars in message is:" + count);

        }
        catch (Exception e)
        {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }
}
```

3. Запуск приложения

1. Запуск сервера имен (входит в поставку с JDK1.2). Это делается, чтобы мы смогли получить ссылку на службу имен

```
tnameserv
```

по умолчанию сервер запускается по порту 900. Это значение можно изменить, указав параметр запуска `-ORBInitialPort`, например

```
tnameserv -ORBInitialPort 1024
```

2. Запуск сервера `testServer`

```
java testServer -ORBInitialPort 1024
// указывается тот порт
//по которому работает сервер имен
```

3. Запуск клиента `testClient`

```
java testClient
  -ORBInitialHost javas.stu.neva.ru
  -ORBInitialPort 1024
```

параметр `-ORBInitialHost` указывает хост на котором работает `testServer`

после выполнения этого приложения, на консоль будет выведено

```
number of chars in message is:12
```

4. Соединение с сервером без использования службы имен

Основная идея этого метода состоит в том, что сервер сохраняет экземпляр класса удаленного объекта (`testServant`) в виде текстового файла в любом месте, доступном клиенту. Клиент загружает из файла данные (в объект `String`) и преобразует специальными методами в объектную ссылку на удаленный объект.

Все это реализуется следующим образом

Удалим некоторую часть кода — это касается и клиента (`testServer.java` и `testClient.java`)

1. Исключим из `import` библиотеки

```
org.omg.CosNaming.*;
org.omg.CosNaming.NamingContextPackage.*;
```

2. Удалим код соответствующий п.3-п.4

Вместо удаленного кода вставим — для сервера:

```
//преобразуем объект в строку
String ior = orb.object_to_string(testRef);
String filename = System.getProperty("user.home") +
System.getProperty("file.separator")+"test_file";

//создаем файл test_file
FileOutputStream fos = new FileOutputStream(filename);
PrintStream ps = new PrintStream(fos);
//записываем в него данные
ps.print(ior);
ps.close();
```

для клиента:

```
String filename = System.getProperty("user.home") +
System.getProperty("file.separator")+"test_file";
//открываем файл
FileInputStream fis = new FileInputStream(filename);
DataInputStream dis = new DataInputStream(fis);
//читаем данные
String ior = dis.readLine();
//преобразуем в объект CORBA
org.omg.CORBA.Object obj = orb.string_to_object(ior);
//получаем ссылку на удаленный объект
test testRef = testHelper.narrow(obj);
```

Скомпилируем приложения, запустим сервер и клиент аналогично тому, как мы делали это раньше (сервер имен запускать в данном случае не надо).

5. Ресурсы

- <http://developer.javasoft.com/developer/earlyAccess/jdk12/idltojava.html> Компилятор idltojava

[ЦНИИ РТК, Санкт-Петербург.](#)