分布式系统与云计算

课程实验报告

更新：2020 年 12 月 31 日

1 实验一：数据包 Socket 应用

1.1 实验目的

1). 理解数据包 *socket* 应用
2). 实现数据包 *socket* 通信
3). 了解 *Java* 并行编程的基本方法

1.2 实验内容

1). 构建客户端程序
   a). 构建 *DatagramSocket* 对象实例
   b). 构建 *DatagramPacket* 对象实例, 并包含接收者主机地址、接收端口号等信息
   c). 调用 *DatagramSocket* 对象实例的`send`方法, 将 *DatagramPacket* 对象实例作为参数发送
2). 构建服务端程序
   a). 构建 *DatagramSocket* 对象实例并指定接收端口号

b). 构建 *DatagramPacket* 对象实例用于重组接受信息

c). 调用 *DatagramSocket* 对象实例的`receive`方法, 进行消息接收，并将 *DatagramPacket* 对象实例作为参数

1.3 详细设计

1.3.1 服务器端

创建一个 DatagramSocket 类的实例 socket，选定本机端口 40096，并与本机默认 IP 地址进行绑定。标志该 socket 所在程序为服务器端

```
DatagramSocket socket = new DatagramSocket(40000)
```

服务器端接收客户端数据。在 Socket 接收数据前，先创建一个 DatagramPacket 类的实例对象 inPacket 用于接收数据，可接收长度为 inbuff.length 的 packets。并调用 receive 方法等待数据报到来。并将缓冲区的接收数据其转化为字符串打印输出。

```
DatagramPacket inPacket = new DatagramPacket(inBuff, inBuff.length);
socket.receive(inPacket);
```

服务器端发送收到给客户端。创建一个 DatagramPacket 类的对象 outPacket，将一个标识服务器端成功接收信息的字符串放入其缓冲区，socket 调用 send 方法将 outPacket 对象发送给客户端，这里通过 DatagramPacket 的 getSocketAddress() 方法接受的客户端的数据报 inPacket 携带的 IP 地址及端口号，以此为目标将数据报发送给客户端。

```
String message = "Server received message!";
byte[] sendData = message.getBytes();
DatagramPacket outPacket = new DatagramPacket(sendData, sendData.
    length, inPacket.getSocketAddress());
socket.send(outPacket);
```

1.4 客户端

客户端的实现与服务器端类似，主要不同在于客户端的 socket 不指定端口，增加了从键盘接收输入的部分。

首先创建一个 DatagramSocket 类的实例 socket 作为客户端，并将其与任意一个有效的端口绑定。

```
DatagramSocket socket = new DatagramSocket();
```

接着创建一个 DatagramPacket 类的实例 outPacket 用于发送数据报, 这里设置 IP 地址为本地机器，端口号为先前服务器的端口号。

```
DatagramPacket outPacket = new DatagramPacket(new byte[0],0,
    InetAddress.getByName("127.0.0.1"),40000);
```

读写键盘数据。通过读入键盘输入并将其放入 outPacket 实例中，通过调用 DatagramSocket 类的 send 方法将内容发送到服务器端。

```
Scanner scanner = new Scanner(System.in);
      while (scanner.hasNextLine()){
          byte[] buff = scanner.nextLine().getBytes();
          outPacket.setData(buff);
          socket.send(outPacket);
      }
```

最后接收服务器端发送的收到信息，通过构建一个 DatagramPacket 类的实例 inPacket 用于读取 socket 实例中的内容，并打印输出

```
byte[] inBuff = new byte[8000];
DatagramPacket inPacket = new DatagramPacket(inBuff, inBuff.length);
socket.receive(inPacket);
System.out.println(new String(inBuff, 0, inPacket.getLength()));
```

1.5 实验小结

通过完成该实验容易发现在使用 DatagramSocket,DatagramPacket 类构建客户端和服务器端时，许多代码是类似的，双方都需要创建 DatagramSocket 实例用于发送接收数据报，并用 DatagramPacket 实例用于封装数据报。最大的区别仅在于服务器端的 DatagramSocket 实例指定了 IP 地址与端口号而客户端未指定。

在完成实验的过程中也可以体会到 java 构建的 DatagramSocket,DatagramPacket 类在设计上也充分反映了 UDP 协议的特点，需要调用 DatagramPacket 的 getSocketAddress 方法，获取发送者 IP 地址及端口号，才能决定将数据报发往何处体现出 UDP 面向非连接的特点。DatagramSocket，DatagramPacket 类一个用于传递数据报一个用于包装数据报也与 UDP 不区分客户端服务端的思想相吻合。

2 实验二：CORBA 系统编程

2.1 实验目的

1). 了解接口定义语言 *IDL* 的使用
2). 理解 *CORBA* 的工作原理
3). 掌握如何利用 *IDL* 构建 *CORBA* 分布式应用

2.2 实验内容

1). 定义远程接口
2). 使用 *idlj* 编译远程接口
3). 实现服务端
    a). 创建并初始化 *ORB* 实例
    b). 取得根 POA 的引用并启动 POAManager

c). 取得 CORBA 对象命名空间的引用

d). 取得根命名空间

e). 在命名空间 Hello 名下注册新 CORBA 对象

f). 等待客户端调用

4). 实现客户端

a). 创建并初始化 *ORB* 实例

b). 找到服务端

c). 在命名空间发布到 Hello 对象的引用

d). 调用服务器的方法

5). 启动服务

2.3 详细设计

2.3.1 远程接口的定义 & 编译

IDL 是一种接口描述语言，IDL 允许我们用任一定义了与 IDL 映射的语言构建客户端与服务器端。因此我们可以将 IDL 接口映射到不同语言，分别开发。比如将 IDL 用 idlj 映射到 Java 开发客户端，将同一 IDL 映射到 C 开发服务器端，它们之间可以像用同一种语言开发一样流畅互通。目前 IDL 可以映射的语言包括 *C*，*C++*，*Python*，*Java* 等。

这里我们首先使用远程接口定义语言 IDL 定义远程对象的接口，然后用 idlj 将 IDL 语言映射到 Java。

首先创建一个 idl 文件如下：

```
module HelloApp

interface Hello
{
  string sayHello();
```

```
  oneway void shutdown();
};
```

在该文件中，HelloApp 模块的声明会在编译时被转换为 Java 中包的声明，文件中定义的名为 Hello 的 CORBA 接口，编译后它会被转换为 Java 的接口。除此之外，接口内我们定义了两个 CORBA 操作，编译后它也会变成 Java 接口中相应的方法声明。IDL 文件编译后最终生成的 Java 命令如下：

```
package HelloApp;
public interface Hello extends HelloOperations,
org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity
{
}
public interface HelloOperations
{
  String sayHello ();
  void Shutdown ();
}
```

我们使用 idlj 工具读取 IDL 文件并创建相应的 Java 文件。通过-fall 操作在生成客户端绑定时同样生成服务端的骨架。控制台启动 idlj 并输入如下命令：

```
idlj -fall Hello.idl
```

编译后会生成一系列 Java 文件：

1. HelloPOA.java
2. _HelloStub.java
3. Hello.java
4. HelloHelper.java
5. HelloHolder.java
6. HelloOperations.java

这些 IDL 编译生成的 Java 文件用以支撑分布式应用开发，也为接下来客户端与服务端的开发做好了准备。

2.4 实现服务端

2.4.1 实现 HelloImpl.java

在实现服务端之前我们需要首先创建 HelloImpl 文件。它重新声明定义了 IDL 接口文件中的方法，每一个 Hello 实例都需要通过 HelloImpl 实例创建。HelloImpl 是编译生成文件 HelloPOA 文件的子类，它继承了基本 CORBA 函数性质。

```
class HelloImpl extends HelloPOA

重写IDL文件的sayHello(),shutdown()方法
```

声明创建 sayHello() 方法：

```
public String sayHello()
{
  return "\nHello world!!\n";
}
```

声明创建 shutdown 方法：

```
public void shutdown() {
  orb.shutdown(false);
}
```

这里 shutdown 方法调用了 org.omg.CORBA.ORB.shutdown() 方法，该方法接收布尔值，我们传入 FALSE，意味立即结束 ORB。

由于 shutdown() 方法中涉及了对 ORB 的结束，还需要定义一个变量用于接收 ORB 值。

```
private ORB orb;

public void setORB(ORB orb_val) {
  orb = orb_val;
}
```

2.4.2 实现 HelloServer 类

首先创建初始化 ORB 对象，

```
ORB orb = ORB.init(args, null);
```

然后取得取得根 POA 的引用并启动 POAManager，ORB 通过使用 resolve_initital_refe 方法取得引用。并通过 activate() 操作激活 POAManager，让关联的 POA 对象开始处理请求。

```
POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("
    RootPOA"));
    rootpoa.the_POAManager().activate();
```

同时我们实例化 HelloImpl 类去调用 ORB.shutdown() 方法。

```
HelloImpl.setORB(orb);
```

接下来我们需要取得 CORBA 对象命名空间的引用。这里使用了 narrow() 方法取得引用。

```
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
    Hello href = HelloHelper.narrow(ref);
```

由于服务器端通过取得命名服务的对象引用并将其发布的方式来使客户端可以调用 HelloImpl 对象的方法。我们还需要取得根命名空间：

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

在命名空间 Hello 名下注册新 CORBA 对象：

```
String name = "Hello";
NameComponent path[] = ncRef.to_name( name );
ncRef.rebind(path, href);
```

最后通过调用 ORB 对象的 run() 方法等待客户端调用：

```
orb.run();
```

2.5 实现客户端

客户端通过生成文件的 _HelloStub.java 启动自己的 ORB, 找到服务器端使用的命名服务并取得引用后调用它的方法。

首先创建并初始化 *ORB* 实例：

```
ORB orb = ORB.init(args, null);
```

客户端通过对 ORB 发出请求来定位服务端，这里客户端使用 COS 命名服务取得对象引用并通过 Narrow（方法将 CORBA 对象 objRef 转变为 NamingContextExt 对象：

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

接下来在命名空间发布到 Hello 对象的引用, 这里首先标识了 Hello 对象名然后通过命名服务的 resolve_str() 方法来取得服务器端的引用，并将其转换为 Hello 对象：

```
String name = "Hello";
helloImpl = HelloHelper.narrow(ncRef.resolve-str(name));
```

调用服务器方法,CORBA 允许客户端如同使用本地对象的方法一样使用服务器的方法，中间复杂的转化已经由 IDL 生成的文件来完成了。

```
System.out.println(helloImpl.sayHello());
helloImpl.shutdown();
```

2.6 启动服务

首先通过 javac 命令来编译客户端与服务端文件, 先打开 cmd 切换到文件所在目录后输入以下命令：

```
$ javac HelloServer.java
$ javac HelloClient.java
```

接下来通过 orbd 启动服务：

```
$ orbd -ORBInitialPort 1050 -ORBInitialHost localhost&
```

分别启动客户端与服务端：

```
$ java HelloServer -ORBInitialPort 1050 -ORBInitialHost localhost&
$ java HelloClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

最终服务器端启动后显示如下：

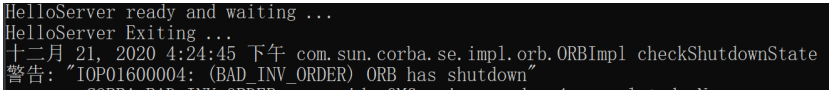

图 1: 服务器端

客户端启动后显示如下：

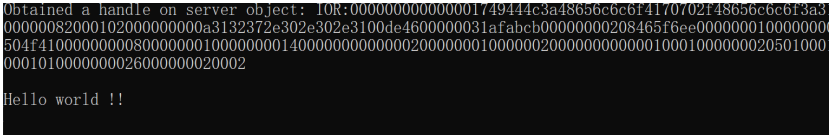

图 2: 客户端

2.7 小结

本次实验通过 IDL 到 Java 的映射实现了公共对象请求代理体系结构 (CORBA) 的构建。CORBA 的构建依赖于 ORB（对象请求代理）来支持分散的程序中的对象之间的交互。ORB 是一个类库，它实现了 CORBA 应用之间的底层通信。

实验具体实现一个经典的“Hello World”分布式应用，它包括客户端和服务端。服务端提供远程接口，客户端去调用这个远程接口。

在服务端中，ORB 使用 HelloImpl 处理远程调用命令并调用本地对象。该文件将 IDL 语言写成的调用和参数映射成 java 语言然后开始调用方法。当调用返回，HelloImpl 可以把结果重新映射为 IDL 并通过 ORB 送往客户端。

在客户端中，需要取得一个远程对象引用。这个对象引用能通过远程调用的存根 (_HelloStub.java) 方法与 ORB 相连接，因此调用它可以使 ORB 连接服务端，调用服务端的方法。

3 实验三：Google AppEngine 程序设计

3.1 实验目的

1). 理解云计算的基本理论知识
2). 掌握 *AppEngine* 软件包的基本使用
3). 运用 *AppEngine* 进行简单的留言本的开发

3.2 实验环境

Windows10 系统，*AppEngine* 软件开发包，*Python3.8*，*Flask* 框架。

3.3 实验内容

Google App Engine 提供一整套开发组件来让用户轻松地在本地构建和调试网络应用，之后能让用户在 Google 强大的基础设施上部署和运行网络应用程序，并自动根据应用所承受的负载来对应用进行扩展，并免去用户对应用和服务器等的维护工作。同时提供大量的免费额度和灵活的资费标准。在开发语言方面，现支持 Java 和 Python 这两种语言，并为这两种语言提供基本相同的功能和 API。

3.3.1 安装 Google Cloud SDK

下载并安装 Google Cloud SDK，由于国内 Google 经常出现各种问题，在 Windows 上尝试安装后失败，转移到一台位于国外的 VPS 服务器上进行实验。安装好 SDK 后，执行 gcloud init –console-only 命令。命令行给出地址，浏览器访问进行授权。复制授权码到命令行。

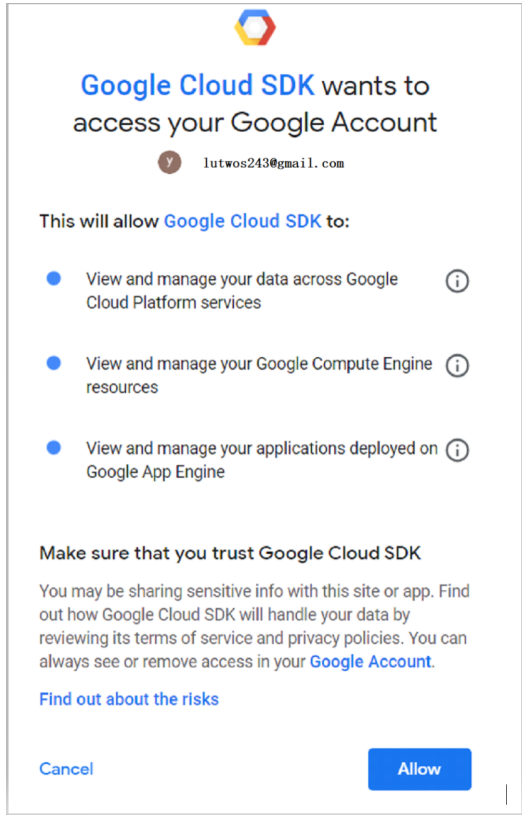

图 3: 登录 Google Cloud SDK

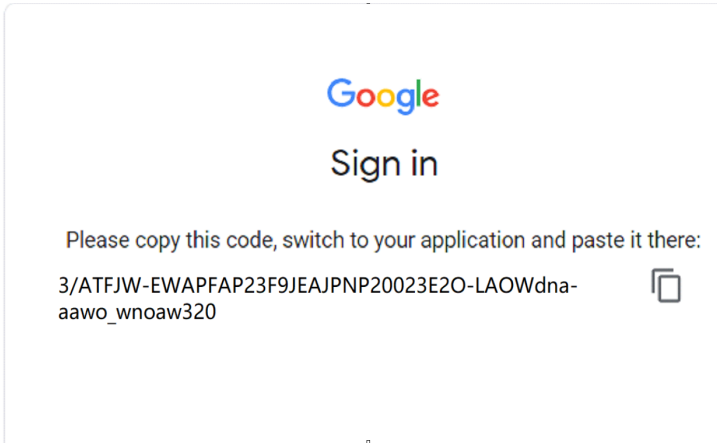

图 4: 复制授权码

3.3.2 创建项目

首先执行 gcloud app create 命令 显示已成功创建。

```
This gcloud configuration is called [default]. You can create additional configurations if you work with multiple
accounts and/or projects.
Run `gcloud topic configurations` to learn more.

Some things to try next:

* Run `gcloud --help` to see the Cloud Platform services you can interact with. And run `gcloud help COMMAND` to g
et help on any gcloud command.
* Run `gcloud topic --help` to learn about advanced features of the SDK like arg files and output formatting
```

Cloud SDK 会进行身份验证，这里使用 Google Cloud 的用户凭据，并使用 Datastore 启用本地测试：输入：gcloud auth application-default login，再次登陆验证后，成功建立项目，启用了用户凭据。

```
Credentials saved to file: [/root/.config/gcloud/application_default_credentials.json]

These credentials will be used by any library that requests Application Default Credentials (ADC).
/usr/bin/../lib/google-cloud-sdk/lib/third_party/google/auth/_default.py:69: UserWarning: Your application has aut
henticated using end user credentials from Google Cloud SDK without a quota project. You might receive a "quota ex
ceeded" or "API not enabled" error. We recommend you rerun `gcloud auth application-default login` and make sure a
 quota project is added. Or you can use service accounts instead. For more information about service accounts, see
 https://cloud.google.com/docs/authentication/
  warnings.warn(_CLOUD_SDK_CREDENTIALS_WARNING)

Quota project "distributed-homework" was added to ADC which can be used by Google client libraries for billing and
 quota. Note that some services may still bill the project owning the resource.
```

3.3.3 建立 Web 服务

首先创建 templates/index.html 文件

```
<!doctype html>
<html>
<head>
    <title>Datastore and Firebase Auth Example</title>
    <script src="{{ url_for('static', filename='script.js') }}"></
        script>
    <link type="text/css" rel="stylesheet" href="{{ url_for('static
        ', filename='style.css') }}">
</head>
<body>
<h1>Datastore and Firebase Auth Example</h1>
<h2>Last 10 visits</h2>
{% for time in times %}
<p>{{ time }}</p>
{% endfor %}
</body>
</html>
```

接下来使用 static/script.js 和 static/style.css 文件添加行为和样式：

```
'use strict';
window.addEventListener('load', function () {

  console.log("Hello World!");

;
body {
      font-family: "helvetica", sans-serif;
      text-align: center;
  }
```

在 main.py 文件中, 使用 Flask 呈现包含占位符数据的 HTML 模板：

```
import datetime
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/')
def root():
    dummy_times = [datetime.datetime(2018, 1, 1, 10, 0, 0),
                   datetime.datetime(2018, 1, 2, 10, 30, 0),
                   datetime.datetime(2018, 1, 3, 11, 0, 0),
                   ]
  return render_template('index.html', times=dummy_times)
if __name__ == '__main__':
    app.run(host='127.0.0.1', port=8088, debug=True)
```

最后我们还需要一个 app.yaml 文件来将 Web 服务部署到 App Engine。配置文件中定义了 App Engine 的 Web 服务设置。在这个简单的 Web 服务中，app.yaml 文件只需定义静态文件的运行时设置和处理程序。已配置、创建并测试 Web 服务，接下来即可将该版本的 Web 服务部署到 App Engine。

3.4 运行结果

通过命令行启动服务端：图见下页

```
descriptor:      [/root/Misc/gcloud/python-docs-samples-master/appengi
-an-app-1/app.yaml]
source:          [/root/Misc/gcloud/python-docs-samples-master/appengi
-an-app-1]
target project:  [distributed-homework]
target service:  [default]
target version:  [20210105t150716]
target url:      [https://distributed-homework.df.r.appspot.com]


Do you want to continue (Y/n)?  Y

Beginning deployment of service [default]...
╔═══════════════════════════════════════════════════════════╗
╠═ Uploading 8 files to Google Cloud Storage                 ═╣
╚═══════════════════════════════════════════════════════════╝
File upload done.
Updating service [default]...
```

图 5: 启动服务端

通过 Cloud SDK 的 gcloud 工具将 Web 服务部署到 App Engine 后从 app.yaml 文件所在项目的根目录中运行 gcloud app deploy 命令。并访问Web 应用。

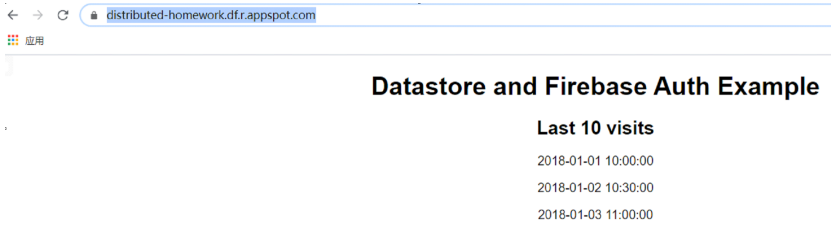

图 6: 部署成功

3.5 小结

通过本次实验，我了解了 Google AppEngine 的使用和部署方法，并成功上传了一个简单的 Web 应用。对国外云计算服务的使用有了一定经验，也让我体验了

云计算带来的种种便捷。