第十二章 网络编程

课程回顾（第十一章）

- 为什么需要泛型？什么是泛型？
- 泛型类、泛型方法、泛型接口
- 泛型的通配符
- 泛型的设计——模板方法模式
- 反射
- 设计安全的全局单例

泛型类（4）

- 多个泛型类使用

```
class Generic<T, E> {
    public T t;
    public E e;
    public void fun(T t, E e) {}
}

Generic<String, Integer> g3 = new Generic<>();
g3.t = "str";
g3.e = 100;
```

泛型方法（2）

```
非泛型类中定义泛型方法
class GenericFun{
    public <T,E> void fun1(E e){}
    public <T> T fun2(T t){
        return t;
    }
}
```

- 调用泛型方法时，在方法名前的尖括号中填入具体类型。

```
GenericFun g = new GenericFun();
g.<String>fun2("str");
```

- 多数情况下，方法调用可以省略类型参数。

```
g.fun2(“str”);//返回值为str
```

泛型方法（5）

- 类的静态泛型方法，不得使用泛型类中声明的泛型，可以独立声明。

```
class GenericStaticMethod<K>{
    private K k;
    public GenericStaticMethod(K k){
        this.k=k;
}
    public static GenericStaticMethod <K> fun1(K k){
        return new GenericStaticMethod <K>(k)}
}//报错：无法从静态上下文中引用非静态类型变量K
    public static <K> GenericStaticMethod<K> fun1(K k){
        return new GenericStaticMethod<K>(k)}
}//可以编译成功
            public static <T> GenericStaticMethod<T> fun1(T t){
        return new GenericStaticMethod<T>(t)}
}//写成另一种类型
```

泛型接口（1）

定义以下泛型接口

```
interface GenericInterface <T>{
    T fun1();
}
```

实现类为非泛型类，需具体指定接口的泛型

```
class GenericInterfaceImpl implements GenericInterface<String> {
    @Override
    public String fun1() {
        return null;
    }
}
```

泛型接口（2）

定义以下泛型接口

```
interface GenericInterface <T>{
    T fun1();
}
```

实现类为泛型类，实现类的泛型要与接口一致

```
class GenericInterfaceImpl<T> implements GenericInterface<T> {
    @Override
    public T fun1() {
        return null;
    }
}
```

泛型的通配符

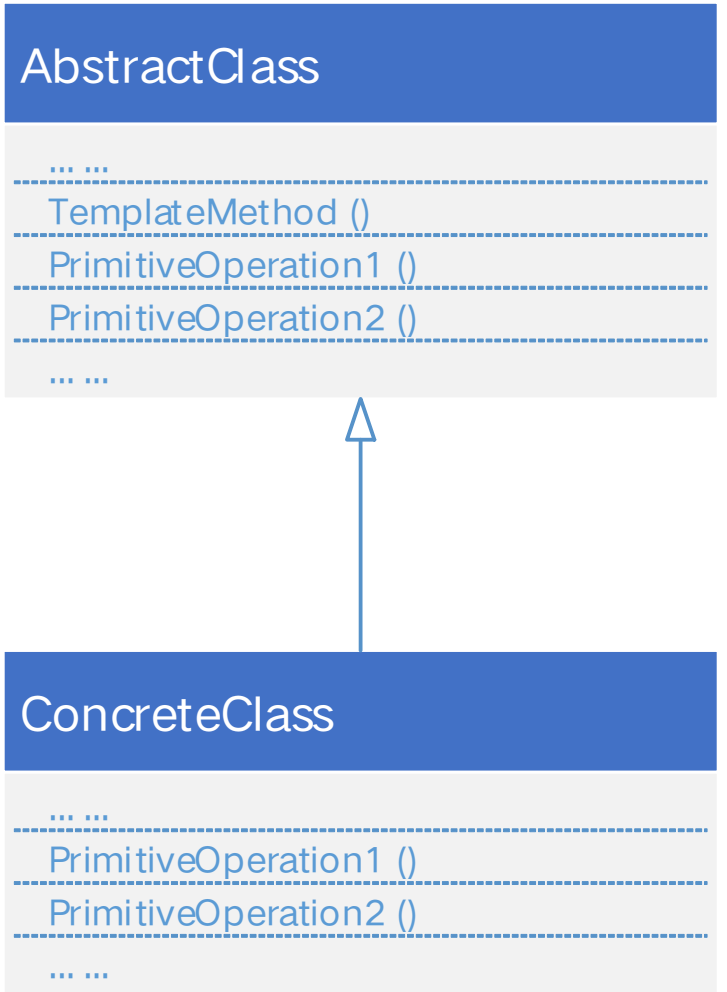
```
public static void printAllObject(List<?> list) {
    for (Object object : list) {
        System.out.println(object);
    }
}
public static void main(String[] args) {
    List<String> list1 = new Arraylist<>();
    List1.add(“java”);
    printAllObject(list1);
    List<Interger> list2 = new Arraylist<>();
    List2.add(7);
    printAllObject(list2);
}
```

泛型的通配符

- 通配符
  - 允许类型参数发生变化
  - <? extends *ClassName*>：类型参数是*ClassName*的子类
  - <? super *ClassName*>：类型参数是*ClassName*的超类
  - <?>：无限定通配符

模板方法模式

- AbstractClass抽象类
  - 抽象模板，定义并实现了一个模板方法
  - 给出顶层逻辑的骨架
- ConcreteClass具体类
  - 实现父类定义的一个或多个抽象方法
- 一个抽象类可以有任意多个具体子类
- 每一个抽象类都可以给出抽象方法的不同实现



反射

- 正常方式

```
Date date = new Date();
```

引入的包类名称 → new实例化 → 实例化对象

- 反射方式

实例化对象 → getClass()方法 → 完整的包类名称

```
System.out.println(date.getClass()); // "class java.util.Date
```

获取Class类对象（1）

- getClass() 方法
  - Object类中的getClass()方法会返回一个Class类型的实例

```
public class getClassTest {
    public static void main(String[] args) {
        Student Harry = new Student(“Harry Potter”,11); // Student类描述学生
        System.out.println(Harry.getClass()); // Class对象描述一个特定类的属性
        // 输出 Class Student
        System.out.println(Harry.getClass().getName());//Class.getName() 返回类的名字
        // 输出 Student
    }
}
```

通过反射构造类的实例

- 方法一：使用Class.newInstance
  - newInstance方法调用默认的构造函数(无参)初始化新创建的对象
  - 如果这个类没有默认的构造函数，就会抛出一个异常

类没有无参构造函数、
类构造器是private时怎么办？

```
Date date1 = new Date();
Class dateClass2 = date1.getClass();
Date date2 = dateClass2.newInstance();
```

通过反射构造类的实例

- 方法二：使用Constructor的newInstance
  - 通过反射先获取构造方法再调用
  - 先获取构造函数，再执行构造函数
- 区别
  - Constructor.newInstance是可以携带参数的
  - Class.newInstance是无参的

通过反射构造类的实例

- 获取构造函数
  - 获取所有"公有的"构造方法
    - public Constructor[] getConstructors() { }
  - 获取所有的构造方法（包括私有、受保护、默认、公有）
    - public Constructor[] getDeclaredConstructors() { }
  - 获取一个指定参数类型的"公有的"构造方法
    - public Constructor getConstructor(Class... parameterTypes) { }
  - 获取一个指定参数类型的"构造方法"，可以是私有的，或受保护、默认、公有
    - public Constructor getDeclaredConstructor(Class... parameterTypes) { }

设计安全的单例模式

- 单例的目的是保证某个类仅有一个实例
  - 在类被加载时就实例化一个对象

```
public class Singleton {
    private static Singleton singleton = new Singleton();
    private Singleton(){}

    public static Singleton getInstance(){
        return singleton;
    }
}
```

利用反射破坏单例

- 通过反射中的setAccessible(true)覆盖Java中的访问控制，进而调用私有的构造函数

```
public class SingletonTest {
    public static void main(String[] args) {
        Class objectClass = Singleton.class;
        Constructor constructor = objectClass.getDeclaredConstructor();
        constructor.setAccessible(true);
        Singleton instance = Singleton.getInstance();
        Singleton newInstance = (Singleton)constructor.newInstance();
        System.out.println(instance == newInstance);
    }
}// instance和newInstance是不同的实例 False
```

抵御反射破坏

```
public class Singleton {
    private static Singleton singleton = new Singleton();
    private Singleton(){
        if(singleton != null) {
            throw new RuntimeException("单例构造器禁止通过反射调用");
        }
    }
    public static Singleton getInstance(){
        return singleton;
    }
}
```

课程导航（第十二章）

- 网络通信的基本原理及IP地址
- Socket编程类库
- URL的使用
- 观察者模式

网络通信

网络编程的主要目的是：直接或者间接地通过网络协议与其他计算机进行通信。网络编程中**有两个主要问题：**

一、如何准确地定位网络上的一台或者多台主机。 ➡ **IP地址、端口号等的概念**

二、找到主机后，如何可靠高效地传输数据。 ➡ **网络传输协议**

本节首先回顾**IP地址**和**网络传输协议**。

IP地址

- 在互联网中，一个IP地址用于**唯一标识一个网络接口**（Network Interface）。一台联入互联网的计算机肯定有一个IP地址，但也可能有多个IP地址。所以如果一个机器有一张网卡，那么它将有两个地址，分别是本机地址、IP地址。

- IP地址分为IPv4和IPv6两种。
  - IPv4采用32位地址，IPv4的地址目前已耗尽，如101.202.99.12
  - IPv6则有128位地址，且地址是根本用不完的，如2001:0DA8:100A:0000:0000:1020:F2F3:1428
- 还有个特殊的IP地址，称之为**本机地址**，是**127.0.0.1**。

域名

IP地址用**数字表示不易记忆**，TCP/IP为了人们方便记忆，设计了一种字符表示地址的机制，称作**域名系统**（DNS）。

一个完整的域名一般为：

计算机主机名.本地名.组名.最高层域名

例如网站域名：

IP：112.80.248.74

域名：www.baidu.com

网络协议

- 1977年，国际标准化组织（ISO）成立了一个专门机构，提出了各种计算机能够在世界范围内互连成网络的标准框架，即著名的**开放系统互联基本参考模型**，简称OSI模型，这种模型是一种**理论模型**。

- Internet国际互联网上的计算机之间采用的是**TCP/IP协议进行通信**，这种协议组由4层组成：应用层，传输层，互联网层、网络接口层。每层又包括若干协议。**使用Java语言编写网络通信程序一般在应用层。**

网络协议

java.net包中提供了**两种常见的网络协议**的支持：

- **TCP**：TCP（英语：Transmission Control Protocol，传输控制协议）是一种**面向连接的**、**可靠的**、**基于字节流的传输层通信协议**，TCP 层是位于 IP 层之上，应用层之下的中间层。TCP 保障了两个应用程序之间的可靠通信。**通常用于互联网协议**，被称TCP/IP。
- **UDP**：UDP（英语：User Datagram Protocol，用户数据报协议），位于 OSI 模型的**传输层**。一个**无连接**的协议。提供了应用程序之间要发送数据的数据报。由于UDP**缺乏可靠性**且属于无连接协议，所以应用程序通常必须容许一些丢失、错误或重复的数据包。

课程导航

- 网络通信的基本原理及IP地址
- **Socket编程类库**
- URL的使用
- 观察者模式

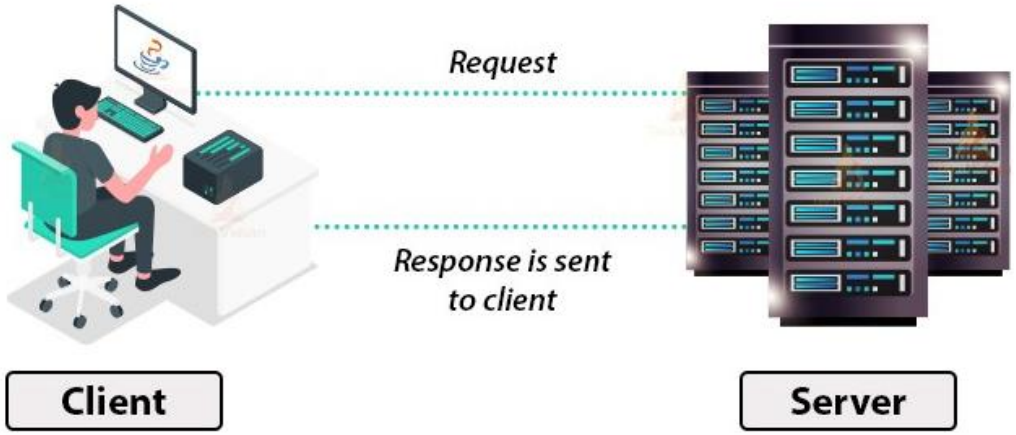
Socket

- Socket（套接字）是一个抽象概念，它是使用了**TCP协议**的通信机制。套接字使用TCP提供了两台计算机之间的通信机制，允许程序员把**网络连接当成一个流（Stream）**。**客户端**程序创建一个套接字，并尝试连接**服务器**的套接字。

- Socket由一个**端口号**和一个**IP地址**唯一确定。

  端口号 → 应用程序

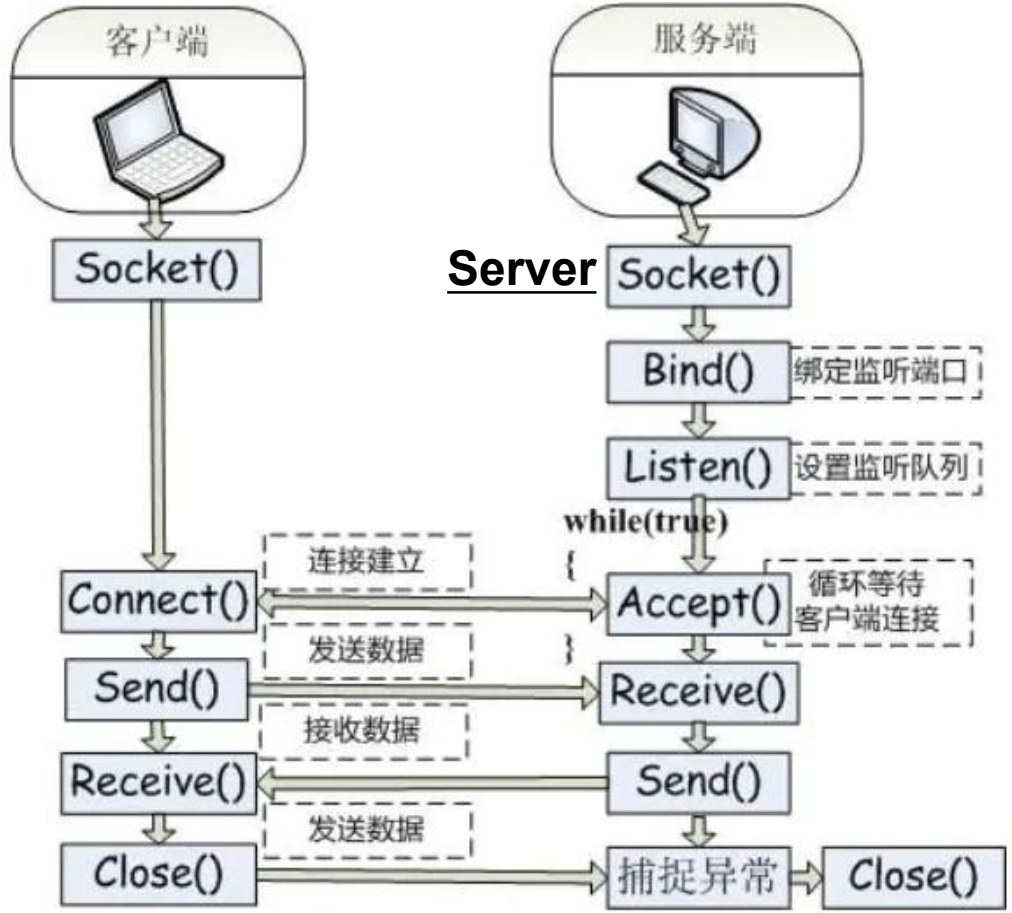
  IP地址 → 计算机



Socket

以下步骤在两台计算机之间使用Socket建立TCP连接时会出现：

- 服务器实例化一个 **ServerSocket** 对象
- 服务器调用 ServerSocket 类的 accept() 方法，**等待客户端的连接**。
- 客户端尝试实例化一个 **Socket** 对象，指定服务器名称和端口号来**请求连接**。



- 连接建立，服务器获得一个新的 **Socket** 对象与客户端使用 I/O 流在进行通信。

服务器端

服务器端创建 **ServerSocket** 的一个例子：

```
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket(6666); // 监听指定端口
        System.out.println("server is running...");

        for (;;) { // 无限循环，持续等待客户端连接
            Socket sock = ss.accept(); // 阻塞等待客户端连接
            System.out.println("connected from " + sock.getRemoteSocketAddress());

            Thread t = new Handler(sock); // 启动一个新线程处理连接
            t.start();
        }
    }
}
```

服务器端对服务器的所有IP地址，端口6666进行监听。通过accept()建立连接后，将获得的Socket送入Handler中进行后续操作。

客户端

客户端创建 **Socket**，与服务器连接，获取输入输出流，送入handle中，进行传输：

```
public class Client {
    public static void main(String[] args) throws IOException {
        Socket sock = new Socket("localhost", 6666); // 连接指定服务器和端口

        try (InputStream input = sock.getInputStream()) {
            try (OutputStream output = sock.getOutputStream()) {
                handle(input, output); // 处理通信逻辑（暂未定义）
                sock.close();
                System.out.println("disconnected.");
            }
        }
    }
}
```

注意：例子中的服务器是“localhost”，与本机6666端口进行传输。

注意：服务器与客户端建立连接后，也是用Socket，而非ServerSocket进行传输。

Socket流

当Socket连接创建成功后，无论是服务器端，还是客户端，我们都使用**Socket实例进行网络通信**。

因为TCP是一种基于流的协议，Java标准库使用 InputStream 和 OutputStream 来封装Socket的数据流，和普通IO流类似：

```
// 用于读取网络数据:
InputStream in = sock.getInputStream();
// 用于写入网络数据:
OutputStream out = sock.getOutputStream();
```

此后，服务器发送给服务器输出流的信息都会成为客户端的输入。

此后，来自客户端程序的所有输出都会包含在服务器的输入流中。

课程导航

- 网络通信的基本原理及IP地址
- Socket编程类库
- URL的使用
- 观察者模式

URL

什么是URL?

URL: Uniform Resource Locator, 统一资源定位符。用于从主机上读取资源（只能读取，不能向主机写）。

一个URL地址通常由4部分组成：

- 协议名：如http、ftp、file等
- 主机名：如baidu、220.181.112.143等
- 途径文件：如/java/index.jsp
- 端口号：如8080、8081等

URL

URL示例：

https://zh.wikipedia.org:443/w/index.php?title=统一资源定位符

- **https**，是协议；
- **zh.wikipedia.org**，是服务器；
- **443**，是服务器上的网络端口号；
- **/w/index.php**，是路径；
- **?title=统一资源定位符**，是询问

要使用URL进行通信，就要使用**URL类创建其对象**，通过引用URL类的方法完成网络通信。创建URL对象要引用java.net包中提供的java.net.URL类的构造方法。

创建URL类的对象

URL类提供用于创建URL对象的构造方法有4个：

- **1.public URL(String str)**

  它是使用URL的字符串来创建URL对象。如：

  ```
  URL myurl = new URL( “http://www.edu.cn” );
  ```

- **2.public URL(String protocol, String host, String file)**

  这个构造方法中指定了协议名“protocol”、主机名“host”、文件名“file”，端口使用缺省值。如：

  ```
  URL myurl = new URL(“http”, “www.edu.cn”, “index.html”);
  ```

创建URL类的对象

- **3.public URL(String protocol, String host, String port, String file)**

  这个构造方法与第二个构造方法比较，多了一个端口号“port”。如：

  ```
  URL myurl = new URL(“http”, “www.edu.cn”, “80”, “index.html”);
  ```

- **4.public URL(URL content, String str)**

  这个构造方法是给出了一个相对于content的路径偏移量。如：

  ```
  URL mynewurl = new URL(myurl, “setup/local.html”);
  ```

  其中URL对象myurl就是前面的URL对象，那么mynewurl所代表的URL地址为：

  http://www.edu.cn:80/setup/local.html

使用JAVA通过URL获取网页

```
import java.io.*;
import java.net.URL;

public class Main {
    public static void main(String[] args) throws Exception {
        URL url = new URL("https://www.baidu.com/"); // 创建URL对象
        BufferedReader reader = new BufferedReader(new InputStreamReader(url.openStream())); // 打开输入流
读取网页
        BufferedWriter writer = new BufferedWriter(new FileWriter("info.html")); // 创建输出流保存内容

        String line;
        while ((line = reader.readLine()) != null) { // 按行读取网页内容
            System.out.println(line);   // 输出到控制台
            writer.write(line);         // 写入文件
            writer.newLine();           // 换行
        }

        reader.close(); // 关闭输入流
        writer.close(); // 关闭输出流
    }
}
```

URL与Socket通信的区别

它们的区别在于：

- Socket通信方式是在**服务器端运行通信程序**，不停地监听客户端的连接请求，主动等待客户端的请求服务，当客户端提出请求时，马上连接并通信；而URL进行通信时，是**被动等待客户端请求**。
- Socket通信方式是服务器端**可以同时与多个客户端**进行相互通信，而URL通信方式是服务器只能与一个客户进行通信。

课程导航

- 网络通信的基本原理及IP地址
- Socket编程类库
- URL的使用
- 观察者模式

观察者模式

模式动机

- **软件系统：**一个对象的状态或行为的变化将导致其他对象的状态或行为也发生改变，它们之间将产生联动
- **观察者模式：**
  - 定义了对象之间一种一对多的依赖关系，让一个对象的改变能够影响其他对象
  - 发生改变的对象称为观察目标，被通知的对象称为观察者
  - 一个观察目标可以对应多个观察者

观察者模式

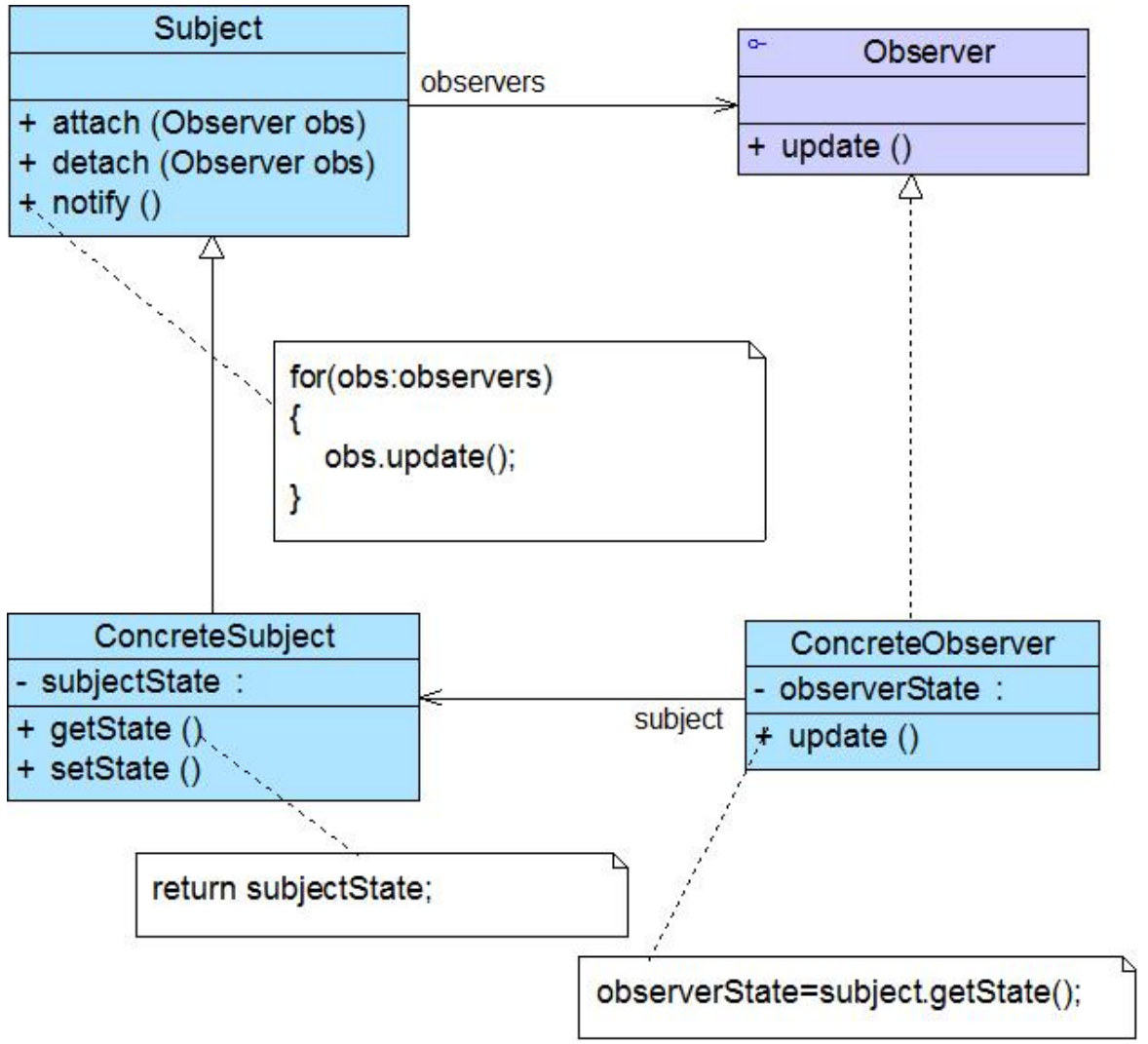
模式定义

- 观察者模式(Observer Pattern)：定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。
- 观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式
- 观察者模式是一种对象行为型模式

观察者模式

观察者模式包含如下角色：

- **抽象目标**：把所有对观察者对象的引用保存在一个集合中，每个抽象目标都可以有任意数量的观察者。抽象目标提供一个接口，可以增加和删除观察者角色。一般用一个抽象类和接口来实现。
- **抽象观察者**：为所有具体的观察者定义一个接口，在得到目标的通知时更新自己。
- **具体目标**：在具体主题内部状态改变时，给所有登记过的观察者发出通知。具体主题角色通常用一个子类实现。
- **具体观察者**：该角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。通常用一个子类实现。如果需要，具体观察者角色可以保存一个指向具体主题角色的引用。



观察者模式

模式分析

- 抽象目标类示例代码：

```
import java.util.*;
public abstract class Subject {
    //定义一个观察者集合用于存储所有观察者对象
    protected ArrayList observers<Observer> = new ArrayList();

    //注册方法，用于向观察者集合中增加一个观察者
    public void attach(Observer observer) {
        observers.add(observer);
    }

    //注销方法，用于在观察者集合中删除一个观察者
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    //声明抽象通知方法
    public abstract void notify();
}
```

观察者模式

模式分析

- 具体目标类示例代码：

```
public class ConcreteSubject extends Subject {
    //实现通知方法
    public void notify() {
        //遍历观察者集合，调用每一个观察者的响应方法
        for(Object obs:observers) {
            ((Observer)obs).update();
        }
    }
}
```

观察者模式

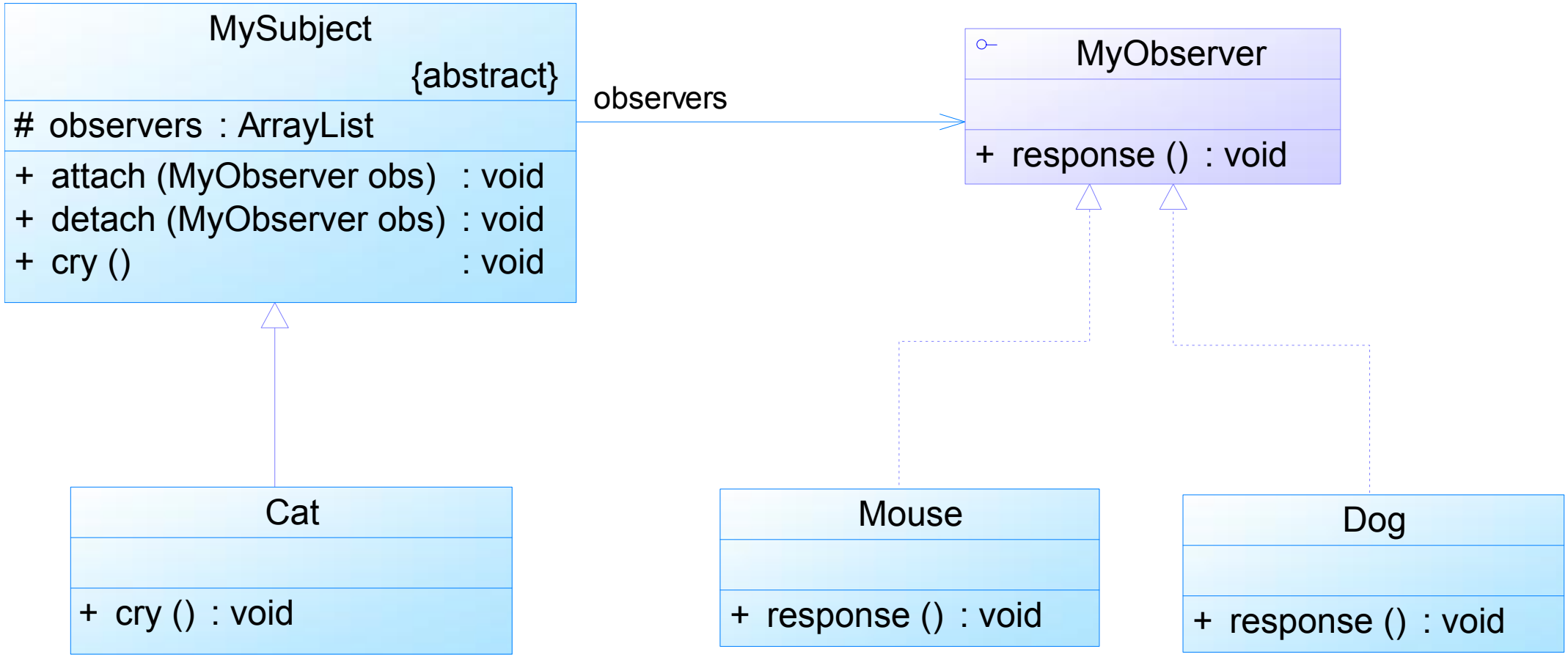
模式实例

- 猫、狗与老鼠：实例说明
  - 假设**猫**是老鼠和狗的观察目标，老鼠和狗是观察者，猫叫老鼠跑，狗也跟着叫，使用观察者模式描述该过程。

观察者模式

模式实例

- 猫、狗与老鼠：参考类图



观察者模式

参考代码：抽象目标

```
public abstract class MySubject
{
   protected ArrayList observers = new ArrayList();

   //注册方法
  public void attach(MyObserver observer)
   {
      observers.add(observer);
   }

   //注销方法
  public void detach(MyObserver observer)
   {
      observers.remove(observer);
   }

   public abstract void cry(); //抽象通知方法
}
```

观察者模式

参考代码：具体目标—猫

```
public class Cat extends MySubject
{
   public void cry()
   {
      System.out.println("猫叫！ ");
      System.out.println("-------");

      for(Object obs:observers)
      {
         ((MyObserver)obs).response();
      }

   }
}
```

观察者模式

参考代码：观察者接口

```
public interface MyObserver
{
   void response();  //抽象响应方法
}
```

观察者模式

参考代码：具体观察者-狗和老鼠

```
public class Dog implements MyObserver
{
  public void response()
  {
    System.out.println("狗跟着叫！ ");
  }
}
```

```
public class Mouse implements MyObserver
{
  public void response()
  {
    System.out.println("老鼠努力逃跑！ ");
  }
}
```

观察者模式

参考代码：测试程序

```
public class Client
{
   public static void main(String[] args)
   {
      MySubject subject=new Cat();

      MyObserver obs1,obs2,obs3;
      obs1=new Mouse();
      obs2=new Mouse();
      obs3=new Dog();

      subject.attach(obs1);
      subject.attach(obs2);
      subject.attach(obs3);

      subject.cry();
   }
}
```

观察者模式

观察者模式优点：

- 可以实现表示层和数据逻辑层的分离
- 在观察目标和观察者之间建立一个抽象的耦合
- 支持广播通信，简化了一对多系统设计的难度
- 符合开闭原则，增加新的具体观察者无须修改原有系统代码，在具体观察者与观察目标之间不存在关联关系的情况下，增加新的观察目标也很方便

观察者模式缺点：

- 将所有的观察者都通知到会花费很多时间
- 如果存在循环依赖时可能导致系统崩溃
- 没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而只是知道观察目标发生了变化

观察者模式

在以下情况下可以使用观察者模式：

- 一个抽象模型有两个方面，其中一个方面依赖于另一个方面，将这两个方面封装在独立的对象中使它们可以各自独立地改变和复用
- 一个对象的改变将导致一个或多个其他对象发生改变，且并不知道具体有多少对象将发生改变，也不知道这些对象是谁
- 需要在系统中创建一个触发链

课程导航

- 网络通信的基本原理及IP地址
- Socket编程类库
- URL的使用
- 观察者模式