

复旦大学计算机科学技术学院



# 编程方法与技术

D.1. 上节课复习

周扬帆

2021-2022第一学期

# JNI: Java Native Interface

## □ 设计目的

### ■ 进行系统调用(glibc.so), 访问系统资源/功能

- 输入、输出
- 文件系统
- 系统环境
- 网络 ...

### ■ 调用c实现的库, 目的?

- 性能
- Legacy代码
- 已写好的代码
- 干Java干不了的事
- 让人看不懂 😊

# JNI 调用

```
1 package com.cudroid.jni;
2 public class HelloJNI {
3     public native void sayHelloWorld();
4     public static void main(String[] args){
5         System.loadLibrary("libHelloWorld");
6         HelloJNI hello = new HelloJNI();
7         hello.sayHelloWorld();
8     }
9 }
```

- ❑ 方法声明加上**native**关键字
  - 不需要在java写上实现代码
- ❑ 用System.loadLibrary加载实现了这个方法/函数的动态链接库
- ❑ 像调用普通java方法一样调用native方法

# JNI 基础数据

## □ 实现动态链接库

- javah生成头文件
- C实现头文件定义签名的JNI函数
- Gcc等编译器编译为.so

## □ 实现JNI函数

### ■ 基本类型

boolean	jboolean	unsigned char	无符号, 8 位
byte	jbyte	signed char	有符号, 8 位
char	jchar	unsigned short	无符号, 16 位
short	jshort	short	有符号, 16 位
int	jint	long	有符号, 32 位
long	jlong	__int64	有符号, 64 位
float	jfloat	float	32 位
double	jdouble	double	64 位
void	void	N/A	N/A

# JNI String

## □ 实现JNI函数

### ■ String

`public native` String echo(String str);

```
JNIEXPORT jstring HelloJNI_echo
(JNIEnv * env, jobject obj, jstring str)
{
    int i;
    char returnbuf[128] = "Hello, World!";
    const jbyte *buffer;
    buffer = (*env)->GetStringUTFChars(env, str, NULL);
    ...
    (*env)->ReleaseStringUTFChars(env, str, buffer);
    ...
    return (*env)->NewStringUTF(env, returnbuf);
}
```

通过这个参数来访问运行环境: JVM

把String转成数组

release内存

构建String对象返回

为什么要通过JVM处理?

# JNI数组

## □ 实现JNI函数

### ■ 数组

`public native` int sum(int [ ]arr);

```
JNIEXPORT jint HelloJNI_sum
(JNIEnv *env, jobject obj, jintArray arr)
{
    jint *carr;
    jint i, sum = 0;
    carr = (*env)->GetIntArrayElements(env, arr, NULL); 数组转换
    if (carr == NULL) {
        return 0; /* exception occurred */
    }
    for (i=0; i<10; i++) {
        sum += carr[i];
    }
    (*env)->ReleaseIntArrayElements(env, arr, carr, 0); 释放内存
    return sum;
}
```

# JNI访问对象

## □ java对象: jobject

```
public void callback(String fromNative){ ...}
```

```
JNIEXPORT jint HelloJNI_test (JNIEnv *env, jobject obj)
{
    jclass cls = env->GetObjectClass(obj);           获得Java层该对象实例的类引用
    jmethodID callbackID = env->GetMethodID(cls, "callback", "(Ljava/lang/String;)V");
    if(callbackID == NULL)                             获得方法句柄
    {
        ...
    }
    jstring native_desc = env->NewStringUTF(" I am Native");
    env->CallVoidMethod(obj, callbackID, native_desc); 调用
}
```

复旦大学计算机科学技术学院



# 编程方法与技术

D.2. Java注解

周扬帆

2021-2022第一学期



# Student例子回顾

```
class Student {
```

```
    char name[] = new char[10];  
    double gpa;  
    int studentID;
```

定义数据

```
    void setName(char [] studentName) {  
        name = new char [10];  
        for(int j = 0; j < name.length && j < studentName.length; j ++) {  
            //只取前10个  
            name [j] = studentName [j];  
        }  
    }
```

定义针对数据的操作方法

```
    boolean compareSID(Student s2) {  
        return studentID > s2.studentID;  
    }  
    boolean compareGPA(Student s2) {  
        return gpa > s2.gpa;  
    }  
    void printRecord() {  
        System.out.println("Student: "+ String.valueOf(name)  
            + ", ID = " studentID + ", GPA = " + gpa);  
    }  
}
```

# Student例子回顾

```
class Student {  
    char name[] = new char[10];  
    double gpa;  
    int studentID;  
    void setName(char [] studentName) {
```

## 输出结果示例

```
Student: student0, ID = 5, GPA = 3.8441517056853165  
Student: student1, ID = 4, GPA = 3.581222057848837  
Student: student2, ID = 3, GPA = 2.3226028811143973  
Student: student3, ID = 2, GPA = 2.895426842379352  
Student: student4, ID = 1, GPA = 3.740106136451817
```

```
    boolean compareGPA(Student s2) {  
        return gpa > s2.gpa;
```

```
    }  
    void printRecord() {  
        System.out.println("Student: " + String.valueOf(name)  
            + ", ID = " + studentID + ", GPA = " + gpa);
```

# 新增数据的处理

- 新需求：数据记录需要增加一个记录

- String address;

- 实现相应的方法

- setAddress

```
class Student2 extends Student
{
    String address;
    void setAddress(String studentAddress)
    {
        address = studentAddress;
    }
};
```

# 新增数据的处理

- ❑ 新需求：数据记录需要增加一个记录
  - String address;
- ❑ 实现相应的方法
  - setAddress
  - Student有个printRecord方法: 打印Student信息
  - Student2的信息扩展了Student
    - printRecord

```
void printRecord() {  
    super.printRecord();  
    System.out.println("Address: " + address);  
}
```

**写错函数签名怎么办？**

# Java注解

**@Override**

```
void printRecord() {  
    super.printRecord();  
    System.out.println("Address: " + address);  
}
```

Java注解

# Java注解

**@Override**

```
void printRecord() {  
    super.printRecord();  
    System.out.println("Address: " + address);  
}
```

- ❑ 加入**源代码**的特殊语法**元数据**
- ❑ 用于**标注**类、方法、变量、参数和包
- ❑ 编译器可以获取并做相应处理
- ❑ 标注可以被嵌入到字节码中
- ❑ 运行时可以获取到标注内容
  - 可通过**反射**获取标注内容

# Java内置注解

## □ @Override

- 检查注解的方法是否是重载方法
- 如果发现父类/接口中并没有该方法时，报编译错误

## □ @Deprecated

- 注解过时方法
- 使用了带这个注解的方法，报编译警告
- 为什么不直接删了这个方法

## □ @SuppressWarnings

- 指示编译器忽略注解声明的警告

# 注解的定义

- ❑ **@Override**
- ❑ **@Deprecated**
- ❑ **@SuppressWarnings**

**怎么做到的？**



# 元注解

## □ @Retention

- 标识这个注解的作用期
- 代码到编译/编译后到加载/加载到运行

## □ @Documented

- 标记这些注解是否包含在用户文档中 (javadoc)

## □ @Target

- 标记注解的应该是哪种东西 (方法、属性、类...)

## □ @Inherited

- 默认作用于子类
- 如果父类有这个注解，子类如果没有任何注解的话，会自动应用父类这个注解

# 元注解

## □ @Retention

- 标识这个注解的作用期
- 代码到编译/编译后到加载/加载到运行

## □ @Documented

- 标记这些注解是否包含在用户文档中

## □ @Target

- 标记注解的应该是哪种东西（方法、属性、类...）

## □ @Inherited

- 默认作用于子类
- 如果父类有这个注解，子类如果没有任何注解的话，会自动应用父类这个注解

# 元注解

## □ @Target

- **标记注解的应该是哪种东西（方法、属性、类...）**
- ElementType.PACKAGE
  - 给一个包进行注解
- ElementType.TYPE
  - 给一个类型进行注解，比如类、接口
- ElementType.METHOD
  - 给方法进行注解
- ElementType.CONSTRUCTOR
  - 给构造方法进行注解
- ElementType.FIELD
  - 给属性进行注解
- ...

# 注解的定义

## □ @Override

### ■ java.lang.Override

```
package java.lang;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.RetentionPolicy.SOURCE;
import static java.lang.annotation.ElementType.METHOD;

/**
 * This annotation is used as a marker to indicate that the annotated
 * method declaration is intended to override another method in the
 * class hierarchy. If this is not the case, the compiler will emit a
 * warning.
 *
 * @since 1.5
 */
@Override
public @interface Override
{
}
```

作用时间：编译器检查

注解是作用于方法上的

注解定义

# 注解的定义

## □ @Override

### ■ java.lang.Override

```
package java.lang;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.RetentionPolicy.SOURCE;
import static java.lang.annotation.ElementType.METHOD;

/**
 * This annotation is used as a marker to indicate that the annotated
 * method declaration is intended to override another method in the
 * class hierarchy. If this is not the case, the compiler will emit a
 * warning.
 *
 * @since 1.5
 */
@Retention(SOURCE) @Target(METHOD)
public @interface Override
{
}
```

它什么都不做!!!

# 注解的定义

## ❑ @Override

- java.lang.Override
- 什么行为都没有定义

## ❑ 注解是元数据，不显式地定义业务逻辑

## ❑ 如何检查在父类中有一个同签名的函数？

- @Retention(RetentionPolicy.SOURCE)

- **RetentionPolicy.SOURCE**

→ 编译后丢弃

- **RetentionPolicy.CLASS**

→ 类加载后丢弃，默认值！

- **RetentionPolicy.RUNTIME**

→ 始终在

# 注解的定义

## □ @Override

- java.lang.Override
- 什么行为都没有定义

## □ 注解是元数据，不显示定义业务逻辑

## □ 如何检查在父类中有一个同名的函数

- @Retention(RetentionPolicy.SOURCE)
- 行为在编译器中定义

# 自定义注解

## □ 和Override类似的定义注解

```
public @interface YZ {  
}
```

```
@YZ  
public class A {  
    //...  
}
```

- 有什么用?
  - 没什么用
- 除非我改IDE, 改编译器, 或者**实现我的代码**来处理这个注解



# 自定义注解

```
@Retention(RetentionPolicy.RUNTIME)
public @interface YZ {
}

@YZ
public class A {
    //...
}
```

## □ 实现代码来处理注解

- 反射!

```
boolean hasAnnotation = A.class.isAnnotationPresent(YZ.class);
if ( hasAnnotation ) {
    System.out.println("YZ Annotation Found");
}
```

# 自定义注解

## □ 定义注解的属性

- 注解可以带有成员变量
- `@Retention(RetentionPolicy.SOURCE)`
- 用于传入一些数据，协助获取、使用注解的代码判断如何处理注解

## □ 定义注解属性的方法

```
@interface YZ {  
    int id();  
    String msg();  
}
```

定义了注解有id和msg两个属性

## □ 使用方法

- 使用时赋值

```
@YZ (id=1, msg="hello")  
public class A {  
    //...  
}
```

# 自定义注解

```
@Retention(RetentionPolicy.RUNTIME)
public @interface YZ {
    int id();
    String msg();
}
@YZ(id=1, msg="hello")
public class A {
    //...
}
```

## □ 实现代码来处理注解

### ■ 反射!

```
boolean hasAnnotation = A.class.isAnnotationPresent(YZ.class);
if ( hasAnnotation ) {
    YZ anno = A.class.getAnnotation(YZ.class);
    System.out.println("YZ Annotation Found: id = " + anno.id()
        + ", msg = " + anno.msg());
}
```

# 自定义注解

## □ 定义注解属性的方法：设置默认值

```
@Retention(RetentionPolicy.RUNTIME)
@interface YZ {
    int id() default -1;
    String msg() default "blank";
}

@YZ
public class A {
    //...
}

boolean hasAnnotation = A.class.isAnnotationPresent(YZ.class);
if ( hasAnnotation ) {
    YZ anno = A.class.getAnnotation(YZ.class);
    System.out.println("YZ Annotation Found: id = " + anno.id()
        + ", msg = " + anno.msg());
}
```

YZ Annotation Found: id = -1, msg = blank

# 自定义注解

## ❑ 实现代码来处理注解

### ■ 反射!

## ❑ 类的注解

```
@Retention(RetentionPolicy.RUNTIME)
public @interface YZ {
    int id();
    String msg();
}
@YZ(id=1, msg="hello")
public class A {
    //...
}
```

调用Class的公有方法

```
boolean hasAnnotation = A.class.isAnnotationPresent(YZ.class);
if ( hasAnnotation ) {
    YZ anno = A.class.getAnnotation(YZ.class);
    // anno.id()
    // anno.msg();
}
```

# 自定义注解

## □ 实现代码来处理注解

### ■ 反射!

## □ 属性的注解

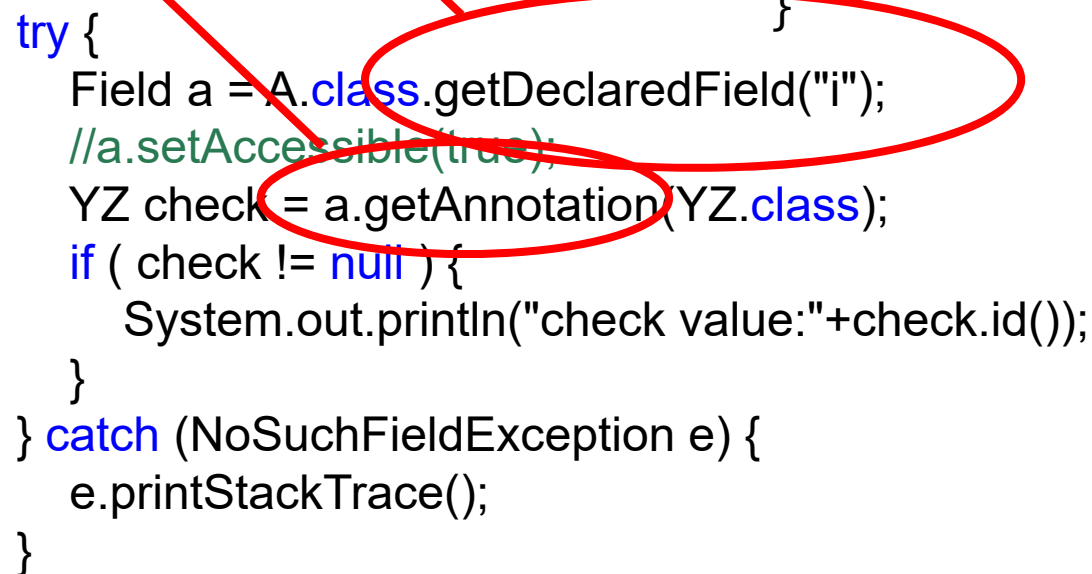
```
@Retention(RetentionPolicy.RUNTIME)
public @interface YZ {
    int id();
    String msg();
}

public class A {
    @YZ(id=1, msg="hello")
    int i;
    //...
```

调用公有方法

反射

```
try {
    Field a = A.class.getDeclaredField("i");
    //a.setAccessible(true);
    YZ check = a.getAnnotation(YZ.class);
    if ( check != null ) {
        System.out.println("check value:"+check.id());
    }
} catch (NoSuchFieldException e) {
    e.printStackTrace();
}
```



# 自定义注解

## □ 实现代码来处理注解

### ■ 反射!

## □ 方法的注解

```
@Retention(RetentionPolicy.RUNTIME)
public @interface YZ {
    int id();
    String msg();
}

public class A {
    @YZ(id=1, msg="hello")
    void test();
    //...
```

调用公有方法

反射

```
try {
    Method testMethod = A.class.getDeclaredMethod("test");
    if (testMethod != null) {
        YZ anno = testMethod.getAnnotation(YZ.class);
        System.out.println("check value:"+anno.id());
    }
} catch (NoSuchMethodException e) {
    e.printStackTrace();
}
```

# 自定义注解的作用

## ❑ 实现代码分析，控制代码的执行（反射）

- 标上注解的部分，可以用反射做相应的处理

## ❑ Junit

- 测试框架

```
@Test
public void methodToBeTested () {
    //...
}
```

## ❑ ButterKnife

- Android前端框架

```
@BindView(R.id.text)
TextView text;
```



# 自定义注解的作用

- ❑ IDE
- ❑ 编译器
- ❑ 代码重构
- ❑ 源代码分析
- ❑ 运行时
  - 框架：标记代码，分析、处理、执行

复旦大学计算机科学技术学院



# 编程方法与技术

D.3. Java注解

周扬帆

2021-2022第一学期

# Java的类与对象

- ❑ 类是数据结构（及相应方法）的描述
- ❑ 对象是具体的该数据结构的数据实例
- ❑ 关键字：数据
- ❑ 数据怎么传输、存储？
  - 数据 → 某种统一格式
  - 序列化： serialization

# 序列化

In computer science, in the context of data storage, serialization is the process of **translating** data structures or object state into a format that can be **stored** (for example, in a file or memory buffer, or transmitted across a network connection link) and **reconstructed** later in the **same or another** computer environment. -Wikipedia

- ❑ **序列化是指将程序内存中的某种数据结构的  
数据转化为数据流或字节数组的操作**
- ❑ **什么用**
  - **存储、传输**

# Java序列化

## □ 序列化Serialization

- 一个对象可以被表示为一个字节序列
- 该字节序列包括
  - 对象的类型信息
  - 对象中数据的类型
  - 对象的数据

## □ 反序列化Deserialization: 逆过程

- 将字节序列转换成Java对象

# Java序列化

- ❑ 可以序列化的类，实现Serializable接口
- ❑ Serializable接口
  - 只有接口声明，接口内内容
  - **空接口**，标记接口(Marker Interface)
    - 没有内容的空接口
  - 任何被开发者认为可被序列化的类都需继承Serializable接口

# Java序列化

```
public class Student implements java.io.Serializable
{
    public String address;
    public transient int SSN;
    public int studentID;
    public void printInfo() {
        // ...
    }
}
```

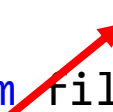


序列化接口

## □ 空接口的目的?

# write/readObject读写数据

- ❑ 例：保存为文件
- ❑ 将FileOutputStream封装为ObjectOutputStream
- ❑ 调用其writeObject方法将可序列化对象写入到输出流

```
public static void main(String[] args) {  
    //...  
    try {  
         对象输出流  
        FileOutputStream fileOut = new FileOutputStream("student.ser");  
        ObjectOutputStream outputStream = new ObjectOutputStream(fileOut);  
        outputStream.writeObject(obj);  
        outputStream.close();  
        fileOut.close();  
    } catch (Exception e) {  
        // ...  
    }  
}
```



# write/readObject读写数据

- ❑ 例：从文件中导入
- ❑ 将FileInputStream封装为ObjectInputStream
- ❑ 调用其readObject方法将输入流写入到可序列化对象

```
public static void main(String[] args) {  
    //...  
    try {  
        FileInputStream fileIn = new FileInputStream("student.ser");  
        ObjectInputStream in = new ObjectInputStream(fileIn);  
        obj = (Student) in.readObject();  
        in.close();  
        fileIn.close();  
    } catch (Exception e) {  
        // ...  
    }  
}
```

对象输入流

# 可被序列化的条件

- JAVA中可被序列化的对象
  - 对象必须实现Serializable接口
  - 所有**会被序列化**的成员必须满足以下任意一条
    - 成员类型为基本类型，例如：int, long, float等
    - 成员类型为数组，且数组中的所有元素均可被序列化
    - 成员类型实现了Serializable接口
      - 否则抛异常

# 会被序列化的成员

- ❑ 方法**不会**被序列化
- ❑ static成员**不会**被序列化，反序列化后值不会改变
- ❑ transient成员**不会**被序列化，反序列化后变成默认值（等会讲）
- ❑ final成员**会**被序列化
- ❑ 是否会被序列化与访问权限**无关**
  - 默认权限、public、protected、private的成员均可被序列化

# 会被序列化的成员

## □ transient关键字

```
public class Employee implements java.io.Serializable
{
    public String address;
    public transient int SSN;
    public int studentID;
    public void printInfo() {
        // ...
    }
}
```

# 会被序列化的成员

## □ transient关键字

- 修饰的属性强制不被序列化
- 反序列化后变成默认值

```
public class Employee implements java.io.Serializable
{
    public String address;
    public transient int SSN;
    public int studentID;
    public void printInfo() {
        // ...
    }
}
```

## □ 什么用?

- 节省空间（可以算出来的）
- 没用的（如和运行环境相关的）

# 自定义Java序列化

```
private void writeObject(ObjectOutputStream out)
    throws IOException;
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

- 如果一个类实现了上述签名的方法，JVM会优先使用该方法序列化
- 不在Serializable接口中，而是JVM的规定，怎么调用？
  - 会被JVM反射调用

# 自定义Java序列化

```
private void writeObject(ObjectOutputStream out)
    throws IOException;
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

## □ 自定义序列化目的

### ■ 节省存储空间

### ■ 处理无法序列化的成员

- 将其设为transient
- 在上述两个方法中自定义该成员的序列化和反序列化策略

# 自定义Java序列化

```
private void writeObject(ObjectOutputStream out)
    throws IOException;
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

## □ 自定义序列化

- 可在writeObject中调用ObjectOutputStream.defaultWriteObject()执行默认的序列化
- 可在readObject中调用ObjectInputStream.defaultReadObject()执行默认的序列化
- 用以处理有无法进行默认序列化成员的场景
  - Customize其序列化过程



# Java序列化+

- **父类可序列化, 所有的子类将可序列化**
  - 里式替换法则
  - 默认继承了Serializable接口
  - 注意要实现为可被序列化
  - 否则?

# Java序列化+

- **一个可序列化的子类，如果父类不可序列化**
  - 反序列化过程中，继承于父类的实例变量，将调用默认构造函数来初始化
  - 有可能造成bug
- **一个类如何避免子类实现为可序列化的？**
  - 实现自定义writeObject() 和readObject() 方法
  - 在方法里抛出NotSerializableException 异常

# Java序列化+

- 一个需被序列化的对象如果包含了其他对象
  - 该对象也必须可序列化（实现接口）
    - 自动隐式序列化
  - 如果该对象本来就没有实现序列化接口
    - 继承一个新的，实现接口（丑）
    - 对该对象添加transient 关键字
    - 编写自定义序列化，处理该对象

# Java序列化实例

```
public class ClassA implements Serializable {  
    .....  
  
    public transient ClassC unserializableObject = new ClassC();  
  
    public ClassA() {  
        unserializableObject.c = 6;  
    }  
  
    private void writeObject(ObjectOutputStream outputStream) throws IOException {  
        outputStream.defaultWriteObject();  
        outputStream.writeInt(unserializableObject.c);  
    }  
  
    private void readObject(ObjectInputStream inputStream)  
        throws IOException, ClassNotFoundException {  
        inputStream.defaultReadObject();  
        unserializableObject = new ClassC();  
        unserializableObject.c = inputStream.readInt();  
    }  
}
```

# 更多的序列化方案

- ❑ Java序列化方案虽强大方便
- ❑ 不同场景下需求不同，有可优化之处

协议	特点	适用场景
JSON	易读，纯文本	Restful Api
Protocol Buffers	跨语言，节省存储	高性能RPC
Parcelable	节省存储	Android

复旦大学计算机科学技术学院



# 编程方法与技术

D.4. Java注解

周扬帆

2021-2022第一学期

# 架构模式

## □ 架构模式

- 大软件系统的构造方式
- 前人总结的软件大的设计思路
  - 区别于**设计模式**这种小功能的设计思路

## □ 软件大了，架构就关键了

- 实现模块化分离
- 模块化的好处？
- 怎么**分离**是软件设计的**根本**问题

## □ 现实中，大家已经有很多架构设计思想在

- 可能最熟悉的：客户端-服务器模式

# 客户端-服务器模式

## □ 用于服务提供者/服务使用者分离

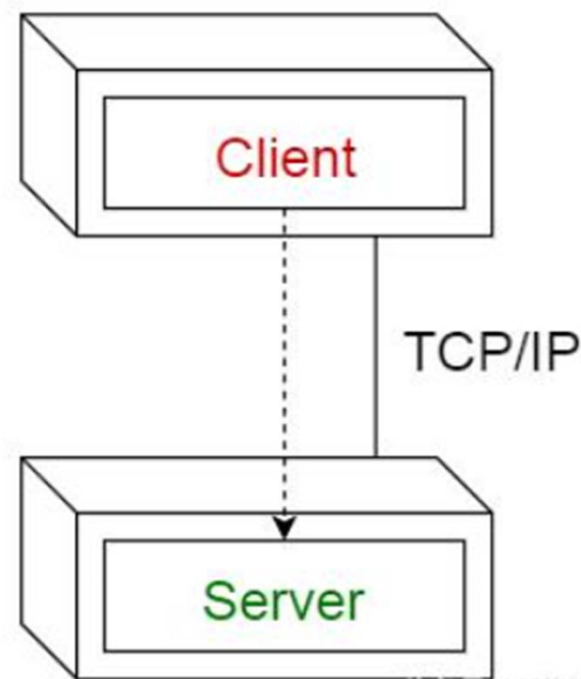
- 类似于函数调用
- 只是**远程**的函数调用
- 把不同功能实现在不同机器/进程上

## □ 好处在哪里？

## □ 场景？

- 功能足够独立
- 功能足够复杂

## □ 设计考虑？





# 主从模式

## □ 用于提供多个同样功能的多个服务

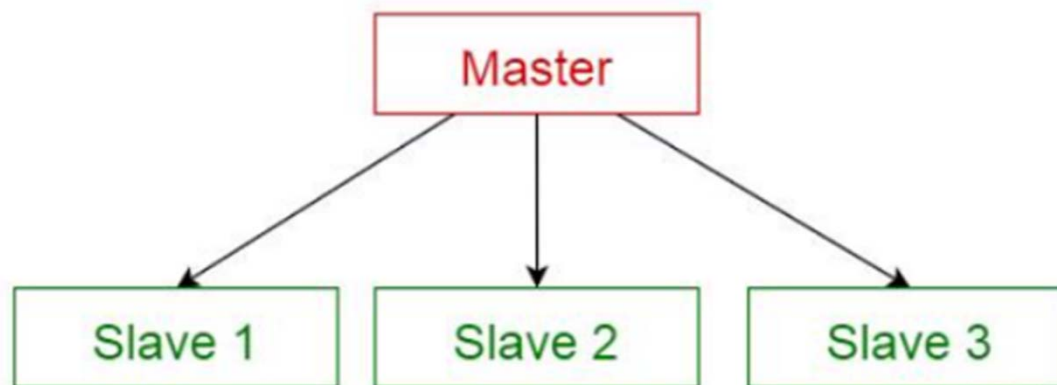
- 简单客户端-单服务器模式的问题?
- 把同样功能实现在不同机器/进程上

## □ 场景？好处在哪里？

- 负载大
- 用户分散

## □ 设计考虑？

- Master要简单不能成为瓶颈
- 负载均衡
- 负载监控

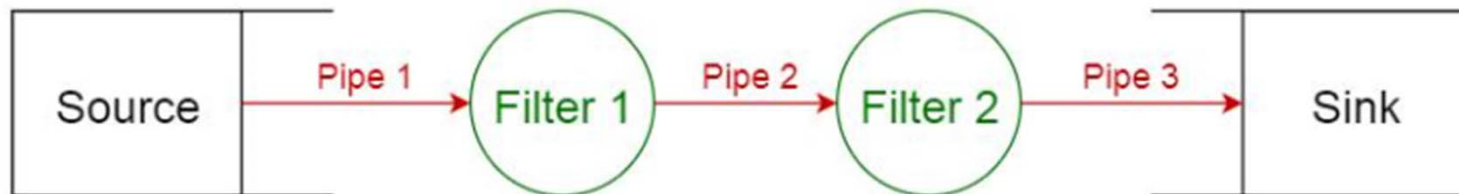


# 主备模式

- 用于提供多个同样功能的多个服务
  - 简单客户端-单服务器模式的问题?
  - 把同样功能实现在不同机器/进程上
- 好处在哪里?
- 场景?
  - 功能足够重要
- 设计考虑? 坏了怎么办?
  - 用户现场恢复
  - 数据状态一致性
    - 回滚Rollback
    - 检查点Checkpoint
    - MessageLog

# 管道-过滤器模式

- 用于串联一系列服务
  - 保证功能按顺序完成
- 好处在哪里？
  - 分离过程 —— 易于调试，易于部署
- 场景？
  - 数据流处理，文档分析例子
- 设计考虑？
  - 避免饥饿



# 代理模式

## □ 用于分发不同的服务请求

- 类似于switch – case，统一入口、区分服务

## □ 场景？

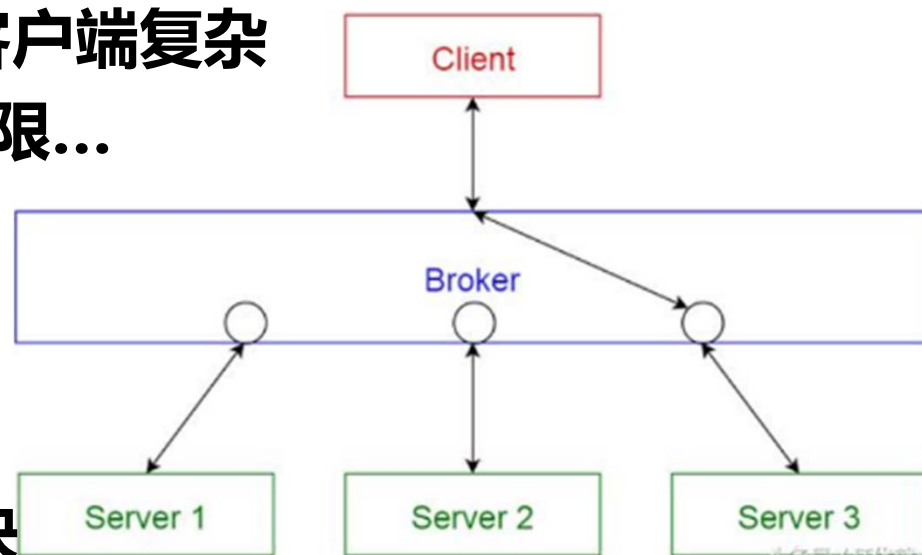
- 客户端可能需要多个服务的时候

## □ 好处在哪里？

- 集中分发管理，避免客户端复杂
- 可以集中log，判断权限...

## □ 设计考虑？

- 代理瓶颈
- 服务不一定是物理节点，可以是进程、模块



# 事件总线模式

- 用于区分消息关注

- 场景?

  - 系统信息接收

- 好处在哪里?

  - 可灵活插拔

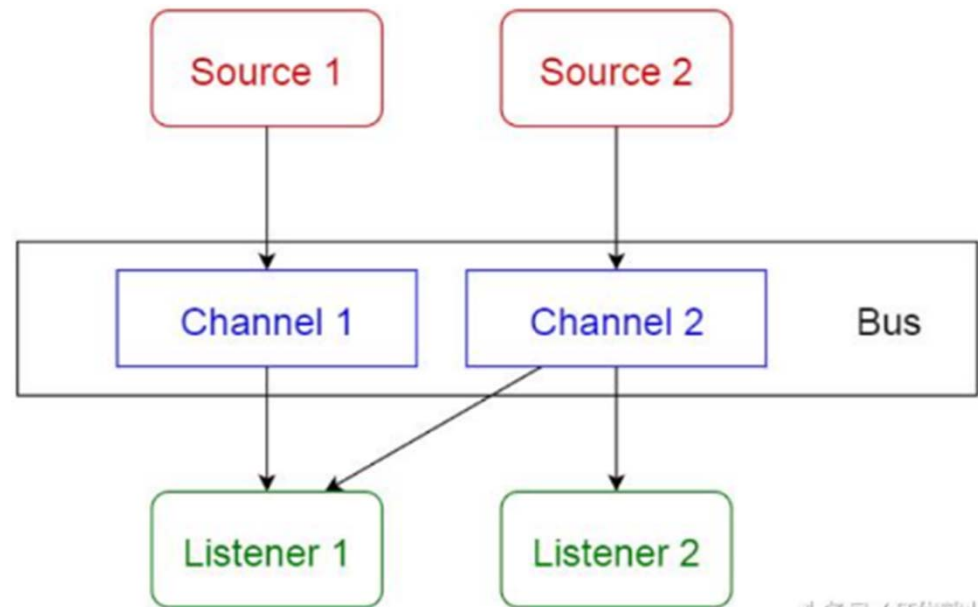
- 设计考虑?

  - 权限

  - 可扩展性

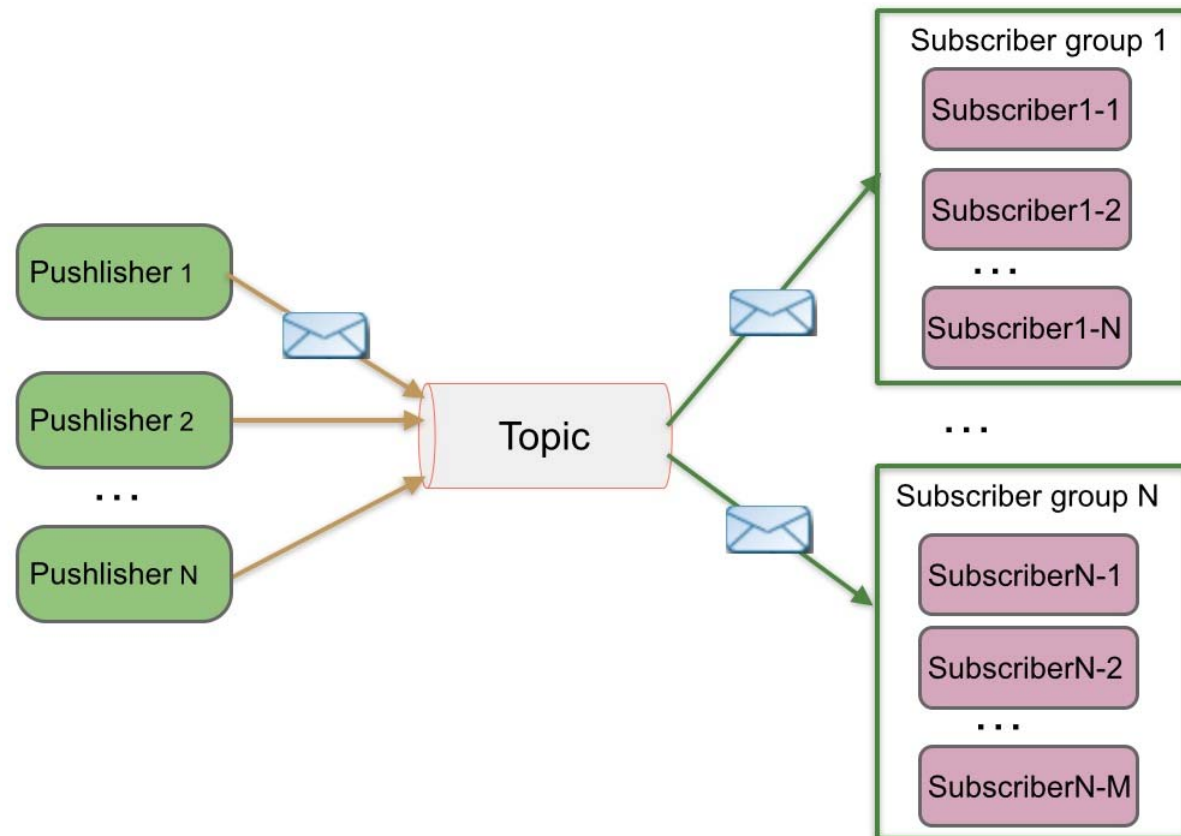
- Android例子

  - BroadcastReceiver



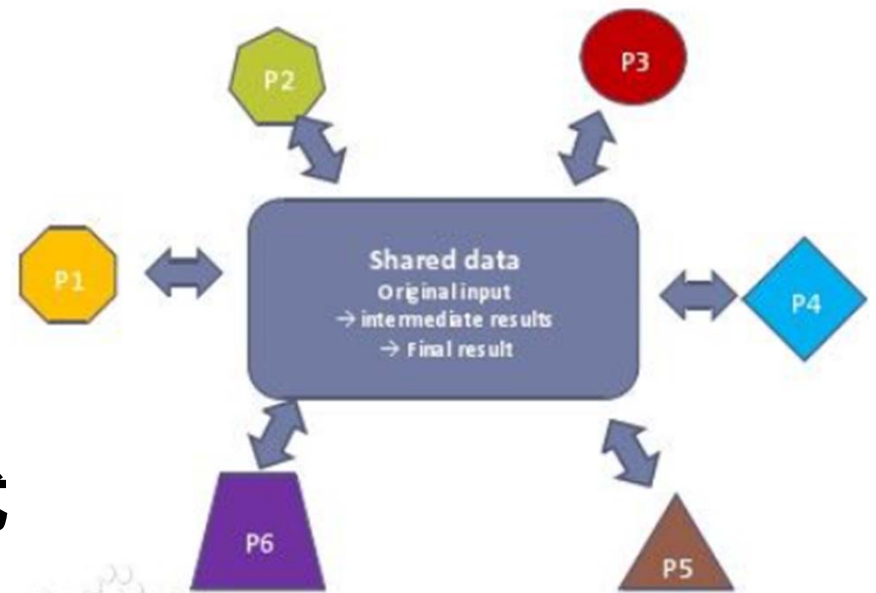
# 消息队列模式

- 用于模块调度、模块间通信等
  - 可以结合事件总线 and 代理模式



# 黑板模式

- 用于非确定解法的问题处理
- 数据由一系列处理程序共享
- 处理程序读取数据，再写回结果
- 设计考虑
  - 黑板是什么？
    - 数据库、共享内存、文件
  - 数据可能需要锁
  - 计算的并行支持
  - 约定程序处理规则
  - 例：机器学习应用黑板模式



# 分层模式

- ❑ 功能分层级，每个层都为下一个提供更高层次服务
- ❑ 场景？
  - TCP/IP
  - 计算机系统
- ❑ 好处
  - 便于理解，便于灵活替换
- ❑ 设计考虑
  - 层间耦合约定
  - 效率、复杂度的trade-off



# 解释器模式

## □ 解释专用语言

- 实施相应的功能

## □ 场景?

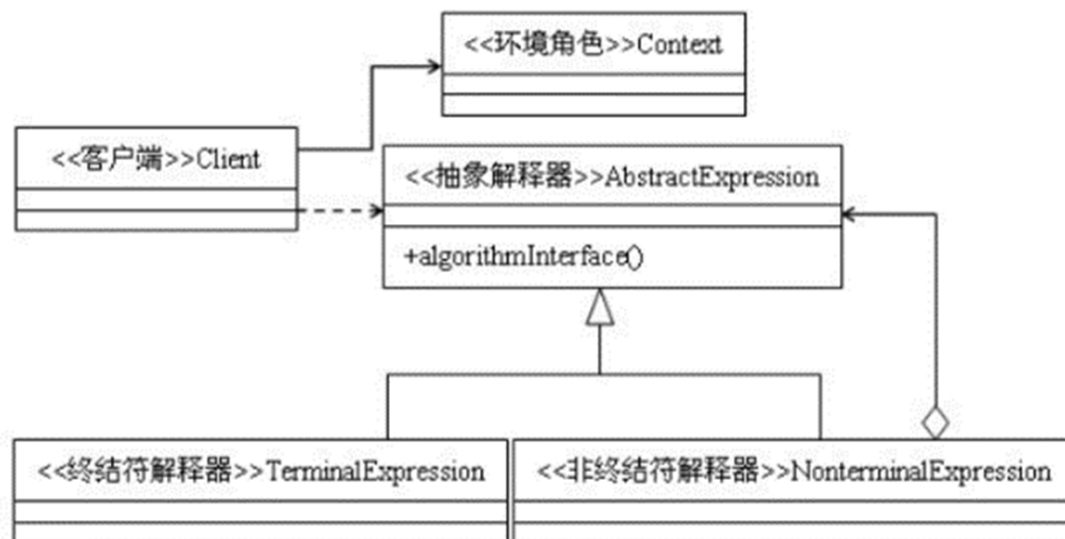
- SQL
- 通讯协议

## □ 设计考虑

- 语言的构造

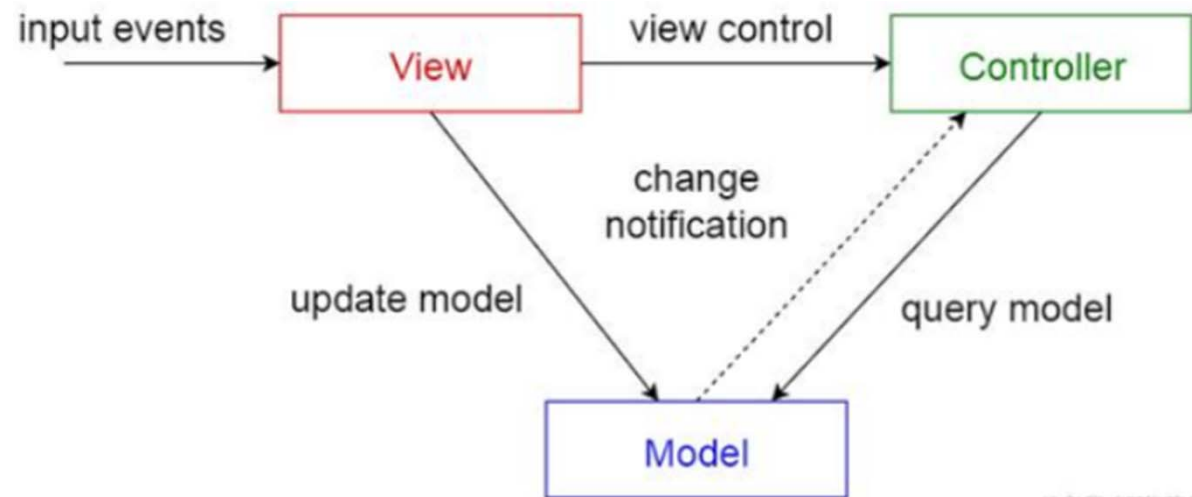
形式化方法

RegEx/更复杂的语法分析



# MVC模式

- 用于设计交互系统
- 前端/后端, 客户端/服务器
- 各种复杂的不同的画法
  - 核心：分离
  - 数据
  - 显示（界面）
  - 业务逻辑



# 总结

- ❑ 客户端-服务器
- ❑ 主备、主从、代理
- ❑ 分层
- ❑ MVC
- ❑ 事件总线、消息队列
- ❑ 黑板、管道-过滤器
- ❑ 解释器

复旦大学计算机科学技术学院



# 编程方法与技术

D.5. 练习

周扬帆

2021-2022第一学期

# 课堂练习

- ❑ 自定义注解, @entry
- ❑ 创建一个Library类
  - 里面包含若干方法
  - 其中一个方法带着@entry注解
- ❑ 创建一个Test类
  - 实例化Library类, 查找@entry注解的方法
    - ❑ 如果这个方法是无参数的, void XXX(), 调用它
  - 试着把@entry注解放到Library的另一个方法, 看看Test是不是自动调用另一个方法。