第一部分 分布式计算程序设计基础 (JAVA语言)

java分布式计算编程基础

- 基本技术
  - SOCKETS网络套接字
  - 多线程
  - 数据序列化
  - Java I/O流
  - 集合容器
  - 范型
  - 内部类、匿名类、Lambda
  - 项目构建管理工具

- 高级技术
  - 注解
  - 反射
  - Stream api
  - 动态代理设计模式
  - 控制反转与依赖注入

- 应用案例
  - RPC调用
  - RMI
  - 分布式企业级开发常用框架
  - Spring 生态圈
  - 通信模型

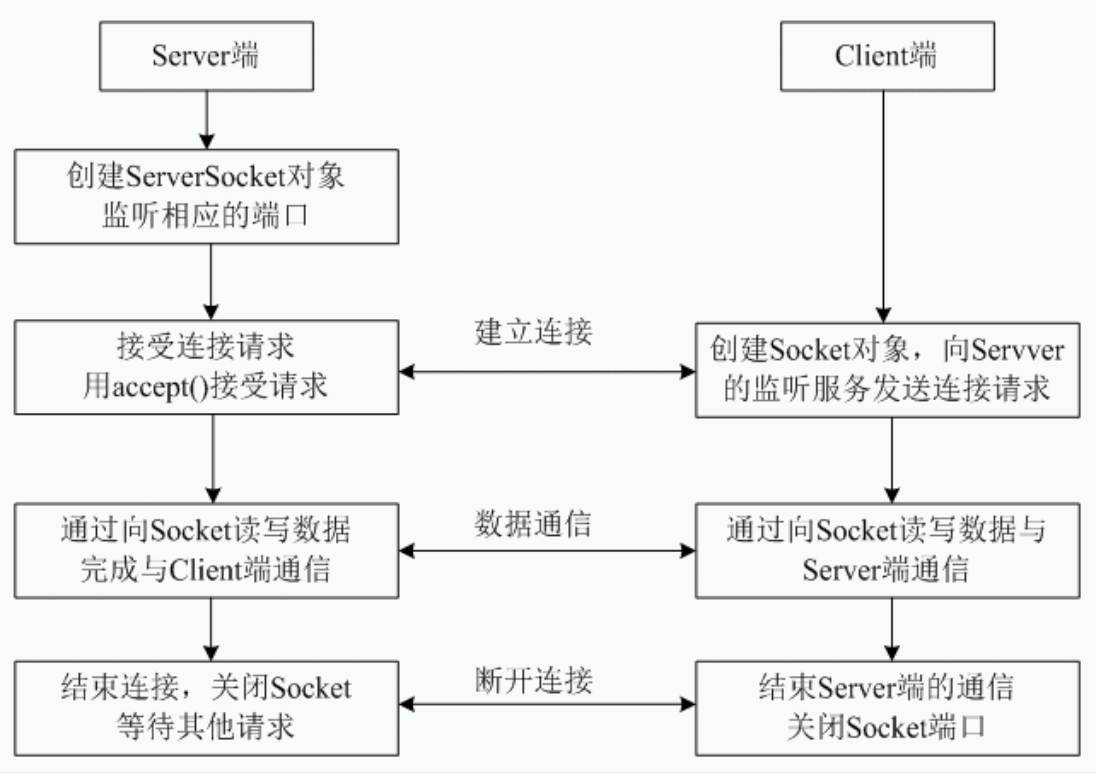
JAVA编程基本技术

回顾。。。。。。

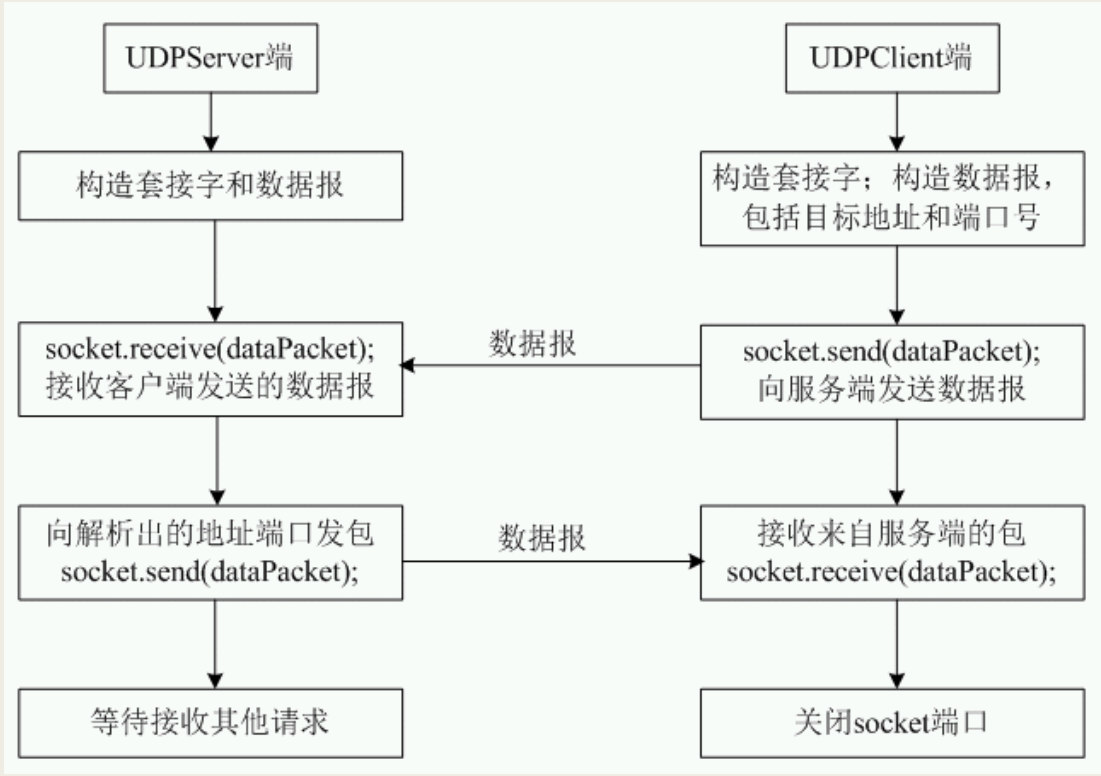
java多线程

- 多线程的基本概念和实现方法
  - 继承Thread类
  - 实现Runnable接口
  - 多线程的同步和通讯
- 多线程的扩展内容
  - Callable和Future
  - 高并发之线程池

SOCKETS网络套接字



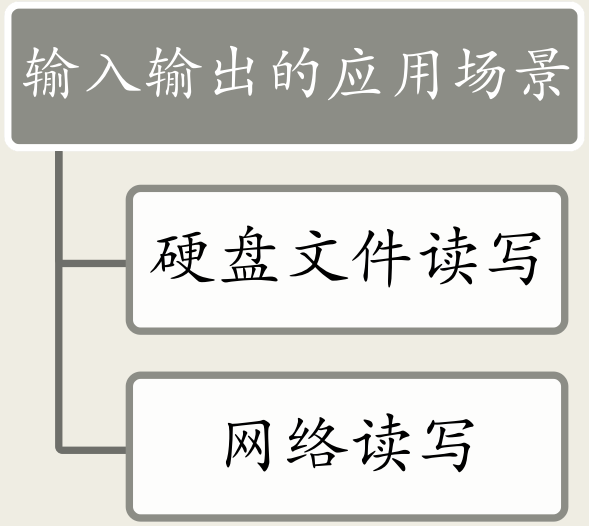
Tcp面向连接



Udp无连接

Java I/O 流

- 流是java进行数据I/O的基本形式
- 熟练掌握基本的字符和二进制流
- 可以在不同的应用场景下选择不同的流实现



序列化

- 序列化是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。
- 我们可以对流化后的对象进行读写操作和网络传输，也可将流化后的对象在网络之间进行传输(注：要想将对象传输于网络必须进行流化)

序列化机制

- 序列化分为两大部分：序列化和反序列化。

- 对象的序列化主要有两种用途：
  - *（1）把对象的字节序列永久地保存到硬盘上，通常存放在一个文件中*
  - *（2）在网络上传送对象的字节序列*

- 最终通过java I/O 流实现文件读写保存和网络传输

定制序列化过程注意事项

- 尽量添加 serialVersionUID 成员变量
- 序列化通常可以自动完成，但有过程进行控制。如果希望某个域不被序列化，可以在它前面加上transient关键字,节省空间，提高时可能要对这个性能
- 如果类中的某个域是静态的，它也不会被序列化

集合容器

- Set
- List
- Map
- Iterator & Enumeration

范型

- 自定义范型类
- 范型类的应用
  - *集合*
  - *反射*

➢ 范型通配符?

- <? super E >固定下界限定
- <? extends E>固定**上界限定**
- <?> 无限定

范型？通配符的Pecs原则

- Producer-Extends 生产者，只读
- Consumer –Super 消费者，只写

内部类、匿名类和Lambda

- 内部类
- 匿名类（内部类的简化）
- Lambda表达式（匿名类的简化）-一种函数式接口的实现对象
  - *函数式接口-仅含一个方法*
  - *本质上是一个接口的匿名实现类的方法函数*
  - *::方法引用*

Lambda表达式

- 基本语法：
  - *(参数列表) -> expression*
  - *(参数列表) ->{ statements; }*
- 举例
  - *( ) -> 5　无参*
  - *x -> 2 * x 有一个参数*
  - *(x, y) -> x – y 有两个参数*

好处

- 简化java程序的“繁重”
- 性能提升

方法引用（method reference）

- 静态方法引用（static method）语法：classname::methodname 例如： Person::getAge
- 对象的实例方法引用语法：instancename::methodname 例如： System.out::println
- 对象的超类方法引用语法： super::methodname
- 类构造器引用语法： classname::new 例如： ArrayList::new
- 数组构造器引用语法： typename[]::new 例如： String[]:new

需要熟练掌握的常用的函数式接口

- Comparator（比较-外部）
- Comparable（比较-内部）
- Supplier（生成）
- Comsumer（处理）
- Predicate（判断）
- Function（转换）

Stream API

语法简洁高效的批量数据操作

----------java中的sql

Stream基本操作

- 创建
- 处理-流水线作业处理，可以反复多次处理
  - *查找，映射，排序*
- 结果-结束生成结果
  - *匹配，归约，收集*

java分布式计算编程

基本技术

- SOCKETS网络套接字
- 多线程
- 数据序列化
- Java I/O流
- 集合容器
- 范型
- 内部类、匿名类、Lambda
- Stream api
- 项目构建管理工具

➢ 高级技术

- 注解
- 反射
- 控制反转与依赖注入
- 动态代理设计模式

➢ 应用案例

- Spring boot
- RPC调用
- RMI
- 分布式企业级开发常用框架
- 通信模型

Notation注解

- Java代码的一种辅助修饰元素
- 提高程序代码的动态配置能力
- 被大量第三方框架、工具、类库采用

注解的应用

- 自定义注解类
- 使用现有的注解类

Java的反射

- “程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言”。从这个观点看，Perl，Python，Ruby是动态语言，C++，Java，C#不是动态语言。
- 尽管在这样的定义与分类下Java不是动态语言，它却有着一个非常突出的动态相关机制：Reflection。

什么是反射

- 反射的概念是由Smith在1982年首次提出的，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力。
- JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

核心竞争力
灵活扩展

反射的应用

- 各种java平台的框架
- 编译器Api
- 其他需要动态调用的场景。。。

Java 反射相关的API

- java.lang包下
  - *Class<T>：表示一个正在运行的Java 应用程序中的类和接口，是Reflection 的起源*
- java.lang.reflect包下
  - *Field 类：代表类的成员变量（也称类的属性）*
  - *Method类：代表类的方法*
  - *Constructor 类：代表类的构造方法*
  - *Array类：提供了动态创建数组，以及访问数组的元素的静态方法*

Class<T>

- Java中任何东西都是对象，类型也是一个对象
- 类是程序的一部分，每个类都有一个Class对象。换言之，每当编写并且编译了一个新类，就会产生一个Class对象
- Class 没有公共构造方法。Class 对象是在加载类时由 Java 虚拟机自动构造的，因此不能显式地声明一个Class对象
- Class是Reflection起源。要想操纵类中的属性和方法，都必须从获取Class object开始

获取Class Object

根据具体情形和个人爱好，可以选择下面任何一种方式获得Class对象

获取方式	说明	示例
object.getClass() 每个对象都有此方法	获取指定实例对象的Class	List list = new ArrayList(); Class listClass = list.getClass();
Class. getSuperclass()	获取当前Class的继承类Class	List list = new ArrayList(); Class listClass = list.getClass(); Class superClass = listClass. getSuperclass();
Object.class	.class直接获取	Class listClass = ArrayList.class;

Class.forName(类名)	用Class的静态方法，传入类的全称即可	try {
		Class c =
		Class.*forName("java.util.ArrayList");*
		} catch (ClassNotFoundException e) {
		e.printStackTrace();
		}

Primitive.TYPE	基本数据类型的封装类获取	Class longClass = Long.TYPE; Class integerClass = Integer.TYPE;

通过反射实例化对象

- 平常情况我们通过new Object来生成一个类的实例，但有时候我们没法直接new，只能通过反射动态生成。
- 实例化无参构造函数的对象，两种方式：
  - *①Class. newInstance();// 不建议*
  - *②Class. getConstructor (new Class[]{}).newInstance(new Object[]{})*
- 实例化带参构造函数的对象：
  - *Class. getConstructor(Class<?>... parameterTypes) . newInstance(Object... initargs)*

通过反射调用Method(方法)

- 获得当前类以及超类的public Method：
  - *Method[] arrMethods = classType. getMethods();*
- 获得当前类申明的所有Method：
  - *Method[] arrMethods = classType. getDeclaredMethods();*
- 获得当前类以及超类指定的public Method：
  - *Method method = classType. getMethod(String name, Class<?>... parameterTypes);*
- 获得当前类申明的指定的Method：
  - *Method method = classType. getDeclaredMethod(String name, Class<?>... parameterTypes)*
- 通过反射动态运行指定Method：
  - *Object obj = method. invoke(Object obj, Object... args)*

案例：动态操纵Method

```java
public class User extends BaseUser{
    private int id;
    public String name;

    public User(){}
    public User(String name){
        this.name = name;
    }

    private int getId() {
        return id;
    }
    private void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```java
    public static void main(String[] args) throws Exception{
        User user = new User();
        Class<?> userClass = user.getClass();

        Method[] publicMethods = userClass.getMethods();
        for(Method method : publicMethods){
            System.out.println("获得当前类以及超类的所有public Method: "+method);
        }

        Method[] currentMethods = userClass.getDeclaredMethods();
        for(Method method : currentMethods){
            System.out.println("获得当前类自己申明的所有Method: "+method);
        }

        Method setBaseIdMethod = userClass.getMethod("setBaseId", new Class[]{int.class});
        System.out.println("获得当前类或超类的public Method setBaseId: "+setBaseIdMethod);

        Method setIdMethod = userClass.getDeclaredMethod("setId", new Class[]{int.class});
        System.out.println("获得当前类的Method setId: "+setIdMethod);

        /**
         * 下面首先动态运行setId(int id)方法，给id赋值为110
         * 然后通过动态运行getId()方法，获得id的值
         */
        setIdMethod.setAccessible(true);
        setIdMethod.invoke(user, new Object[]{110});

        Method getIdMethod = userClass.getDeclaredMethod("getId", new Class[]{});
        getIdMethod.setAccessible(true);
        Integer getId = (Integer) getIdMethod.invoke(user, new Object[]{});
        System.out.println("调用getId方法获得的值: "+getId);
    }
```

将此对象的 accessible 标志设置为指示的布尔值。值为 true 则指示反射的对象在使用时应该取消 Java 语言访问检查。值为 false 则指示反射的对象应该实施 Java 语言访问检查。

通过反射调用Field(变量)

- 获得当前类以及超类的public Field：
  - *Field[] arrFields = classType. getFields();*
- 获得当前类申明的所有Field：
  - *Field[] arrFields = classType. getDeclaredFields();*
- 获得当前类以及超类指定的public Field：
  - *Field field = classType. getField(String name);*
- 获得当前类申明的指定的Field：
  - *Field field = classType. getDeclaredField(String name);*
- 通过反射动态设定Field的值：
  - *fieldType.set(Object obj, Object value);*
- 通过反射动态获取Field的值：
  - *Object obj = fieldType. get(Object obj) ;*

案例：动态操纵Field

```
11    public static void main(String[] args) throws Exception {
12        User user = new User();
13        Class<?> userClass = user.getClass();
14
15        Field[] publicField = userClass.getFields();
16        for(Field field : publicField){
17            System.out.println("获得该类即超类所有public Field: "+field);
18        }
19
20        Field[] currentField = userClass.getDeclaredFields();
21        for(Field field : currentField){
22            System.out.println("获得该类自己申明的所有Field: "+field);
23        }
24
25        Field baseIdField = userClass.getField("baseId");
26        System.out.println("获得该类或超类名为baseId的public Field: "+baseIdField);
27
28        Field idField = userClass.getDeclaredField("id");
29        System.out.println("获得该类自己申明的名为id的Field: "+idField);
30
31        /**
32         * 下面我们给User的属性id赋值，并取值。
33         * 注意:这里仍然要使用setAccessible(true)，因为id是private的!
34         */
35        idField.setAccessible(true);
36        idField.set(user, 110);
37        Integer id = (Integer) idField.get(user);
38        System.out.println("id的值为: "+id);
39    }
```

Java反射总结

- 只要用到反射，先获得Class Object
- 没有方法能获得当前类的超类的private方法和属性，必须通过getSuperclass()找到超类以后再去尝试获得
- 通常情况即使是当前类，private属性或方法也是不需要访问的。

引申学习-ClassLoader

- 类加载器
- 实现动态加载和修改类
- 广泛应用于各类开源框架实现资源隔离、热部署、代码加密保护等场景