1. Springboot
   1. ApplicationContext
      1. getBean

applicationContext.getBean(name | classtype);根据名字获取bean，也可以根据class的类型获取

* 1. @SpringBootApplication

等价于@Configuration, @EnableAutoConfiguration and @ComponentScan

@EnableAutoConfiguration 自动将pom.xml中的各种starter加入到spring容器中。

* 1. 统一异常处理

@RestControllerAdvice定义全局处理异常的地方，具体异常在函数前加上注解

@ExceptionHandler(value = EntityNotFoundException.class)

public ResponseEntity<ApiError> entityNotFoundException(EntityNotFoundException e) {…}

这个函数就是处理EntityNotFoundException异常

当然在业务代码中使用@ExceptionHandler比全局的优先级要高.

* 1. 时间格式Timestamp

后台给前台时间，在字段上注解

@JsonFormat(pattern="yyyy-MM-dd HH:mm:ss",timezone = "GMT+8")

private Timestamp createTime;

依赖jar包

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-annotations</artifactId>

<version>2.8.8</version>

前台给后台格式

字段上加上注解

@DateTimeFormat(pattern = "yyyy-MM-dd")

依赖

<dependency>

<groupId>joda-time</groupId>

<artifactId>joda-time</artifactId>

<version>2.3</version>

</dependency>

=====================================================================================

1. shiro
   1. 字符串通配符权限

规则：“资源标识符：操作：对象实例ID” 即对哪个资源的哪个实例可以进行什么操作。其默认支持通配符权限字符串，“:”表示资源/操作/实例的分割；“,”表示操作的分割；“\*”表示任意资源/操作/实例。

缺失的部分默认为\*，表示所有，

，分割操作 主要避免重复写同一个资源标识符.

"user:update,delete:1"

对资源user的1实例拥有update、delete权限。

* 1. setFilterChainDefinitionMap

拦截器规则，拦截规则使用Map<String, String>将路径与拦截器的映射关系进行匹配.



* 1. 认证

shiro不会为我们提供现成的认证规则，需要我们自己编写认证规则。

编写认证规则时我们需要继承AuthorizingRealm类,并实现doGetAuthenticationInfo抽象方法，方法里面可以将认证信息和数据库比较，匹配是否相等.

认证通过时需要返回一个认@证信息接口AuthenticationInfo的实现类SimpleAuthenticationInfo

* 1. 权限

shiro本身并不提供权限管理的机制，权限的管理需要我们自己进行维护。

赋予权限时我们要继承AuthorizingRealm类，并实现doGetAuthorizationInfo抽象方法。方法中根据用户从数据库获取权限后，返回AuthorizationInfo对象，

使用代码、注解@RequiresPermissions、拦截器等方式匹配权限时，就是根据输入的资源和该用户AuthorizationInfo对象匹配，对象中发现输入的资源权限字符串，则匹配通过.

1. Java 背诵
   1. 数字转字符串。

Int ns = Integer.parseInt(string)

String s =String.valueOf(12)

或者

Integer ins = 24;

String s = ins.toString()

浮点数Double，转换函数parseDouble

* 1. Map接口

增加 V put(key,V value);

删除 remove(key)

修改 replace(key,value)

取值 V get(key);

遍历 for(Entry<Integer, String> e:m.entrySet()){

System.out.println(e.getValue());

}F

* 1. List接口

增加 add(obj),add(index,obj)

删除 remove(index), remove(obj)

修改 set(index,obj)

查 get(index), indexof(obj);

遍历 for(type e : list){}

函数返回一个空list，collections.EMPTY\_LIST;

1. JAVA EE
   1. AOP切面
      1. 基本概念

描述AOP常用的一些术语有通知(Adivce)、切点（Pointcut）、连接点（Join point）、切面（Aspect）

通知分为五中类型：

Before

在方法被调用之前调用

After

在方法完成后调用通知，无论方法是否执行成功

After-returning

在方法成功执行之后调用通知

After-throwing

在方法抛出异常后调用通知

Around

通知了好、包含了被通知的方法，在被通知的方法调用之前后调用之后执行自定义的行为

连接点（Join point） 在springaop中就是被插入前后处理的业务代码函数

切点（Pointcut） 对连接点规则的定义，如匹配哪些函数名，哪些注解进行切入.

切面（Aspect）

切面是切点和通知的结合。通知和切点共同定义了关于切面的全部内容，它是什么时候，在何时和何处完成功能

* + 1. 注解定义

@Aspect:作用是把当前类标识为一个切面供容器读取

@Pointcut：Pointcut是植入Advice的触发条件。每个Pointcut的定义包括2部分，一是表达式，二是方法签名。方法签名必须是 public及void型。可以将Pointcut中的方法看作是一个被Advice引用的助记符，因为表达式不直观，因此我们可以通过方法签名的方式为 此表达式命名。因此Pointcut中的方法只需要方法签名，而不需要在方法体内编写实际代码。

@Pointcut("@annotation(me.zhengjie.annotation.Limit)")

public void mypointcut() {

}

@annotation(me.zhengjie.annotation.Limit)是切点规则表达式（表示使用Limit注解的都是切点），但是不直观，所以函数名mypointcut作为别名，下面的注解直接使用该别名，不用把晦涩的表达式在每个注解前面都写上.

@Around：环绕增强，相当于MethodInterceptor

@AfterReturning：后置增强，相当于AfterReturningAdvice，方法正常退出时执行

@Before：标识一个前置增强方法，相当于BeforeAdvice的功能，相似功能的还有

@AfterThrowing：异常抛出增强，相当于ThrowsAdvice

@After: final增强，不管是抛出异常或者正常退出都会执行

* 1. 日志

两个公共接口库：common-logging和self4j 都是接口，common-logging运行时寻找具体日志实现，self4j编译时确定日志实现库。

具体实现：log4j，LogBack，后者据说某些方面性能更好。

Ref：java日志组件介绍（common-logging，log4j，slf4j，logback）。

* 1. Maven

Maven的生命周期包括clean，default, site。Site就是发布各种报告。

Default周期又叫build周期，包括

prepare-resources 资源拷贝 本阶段可以自定义需要拷贝的资源

compile 编译 本阶段完成源代码编译

package 打包 本阶段根据 pom.xml 中描述的打包配置创建 JAR / WAR 包

install 安装 本阶段在本地 / 远程仓库中安装工程包

pom文件里面的build包含的plugin对应的就是各周期的插件，如果不写，则有默认插件，在install命令时在日志中可以看到默认插件版本。如果jdk版本低等，则需要手动在pom填入插件，定义jdk版本。

生命周期：<http://www.jianshu.com/p/fd43b3d0fdb0>

maven 的用户名是 admin admin123

* + 1. 下载doc和source

在jar包右键，点击maven就有下载doc和source的按钮，自动从仓库下载了。

* 1. Jersey
     1. 注解

@Path。@Path("users/{username: [a-zA-Z][a-zA-Z\_0-9]\*}") 冒号后面为正则表达式过滤，路径中/可以写，可以不写。

@Produces()。方法前的该注解会覆盖类前的该注解。

* 1. Spring事务管理

使用三个变量类型定义事务，

1、TransactionDefinition。定义事务的隔离级别、传播性、超时时间、是否只读。2、TransactionStatus。标示一个事务对象，提供事务状态查询、保存点维护。3、PlatformTransactionManager，创建事务，返回TransactionStatus，提供提交、回滚操作应用于TransactionStatus。

具体的PlatformTransactionManager由各个ORM平台提供，如hibernate的为org.springframework.orm.hibernate4.HibernateTransactionManager。

事务传播性

在一个service中调用另外一个service时，新service的事务环境如何定义，REQUIRES\_NEW创建一个新事务，原service事务挂起；REQUIRED，加入原事务，如果没有原事务，就创建一个，较为常用。

事务配置

使用tx命名空间进行配置，

<tx:method name="save\*" propagation="REQUIRES\_NEW " />属性有：

Name：与事务关联的方法名，支持匹配符；

Propagation：传播特性；

Isolation：事务隔离机制；

Timeout：超时时间；

Read-only：只读事务；

Roll-back：回滚设置，因为哪些异常进回滚；

注解设置

@Transaction注解，最好定义在类这一层，这样子类也会继承事务设置。

高级特性

Spring的bean是单实例的，并且无状态，从而是线程安全的，但是数据库连接等有状态特性，非线程安全，spring通过ThreadLocal对象，不同线程提供不同的ThreadLocal对象，实现了与数据库相关的bean也是线程安全的。不同线程下的服务不会共享同一个事物。

ThreadLoacl的原理，是一个类中，使用map结构，key为thread的id，value为该线程对应的数据库连接，从而实现线程独有的数据库连接。

如果需要代码显式获取数据库连接，使用datasourceutils类，它会感知事物上下文环境，从当前事物获取连接，或者使用TransactionAwareDataSourceProxy对数据源进行代理，原理类似。

@transaction注解

Aop代理实现。todo

* 1. Java的Bean

条件：所有属性为private，提供不带参数默认构造函数，提供getter和setter方法，实现serializble接口。

Pojo：如果只包含属性和getset方法，则称为简单javabean，即为pojo；

与spring的bean比较：javabean仅仅为传值需要，作为作为值对象存在，从而jsp文件中可以直接使用该对象进行存储；spring的bean则由spring管理其生命周期，无其他要求。

* 1. Reflect 反射
     1. Class类表示类型的组织结构，从而反射可根据类的组织结构信息创建类对象。获取一个类型的class有三种方式：

Class c1 = Class.forName("java.lang.Integer");//全局获取

Integer i = new Integer(1);

Class c2 = i.getClass();//运行期获取

Class c3 = Integer.class;//编译器

函数参数为Object类型，输入参数对象转为object后，获取对象自身真实类型

Object object = A；

Class c1 = object.getClass();

Class c = Class.forName(c1.getName());

* + 1. 获取成员字段Field

Field[] publicFields = c1.getFields();//只有public字段

Field[] allFields = c1.getDeclaredFields();//所有字段，不包括父类继承的

获取字段的修饰符、类型、名称、值

for (Field field : allFields) {

field.setAccessible(true);//私有字段获取前要设置为access。结束记得设置回去。

try {

System.out.println(Integer.toString(field.getModifiers()) +" "+ field.getType() + " "+field.getName() +" "+ field.get(ip) + field.getGenericType().getTypeName() );

} catch (IllegalArgumentException | IllegalAccessException e) {

e.printStackTrace();

}

}

getGenericType如果类型是参数化类型，即泛型，则返回全部类型，包括泛型参数，如List里面具体包含的类型。

* + 1. 获得成员函数Method

getMethods(): 获得类的public类型的方法

getDeclaredMethods(): 获取类中所有的方法(public、protected、default、private)

获取某个方法，参数为数组，为参数的class数据。

Method m = c1.getMethod("methodName", new Class[]{List.class});

执行时第一个参数为对象，如果static函数，第一个参数为null

m.invoke(ip1, list);

invoke的返回值就是函数返回值，如果返回的是数组，如下处理：

Object array = method.invoke(obj, new Object[]{});

int length = java.lang.reflect.Array.getLength(array);

List<Object> list = new ArrayList<Object>();

for (int i=0; i<length; i++) {

list.add(java.lang.reflect.Array.get(array, i));

}

还可以获取构造函数

Constructor[] cs = c.getDeclaredConstructors();

获取内部类构造函数，需要传入外部类Class信息,内部类包含外部类this指针。

Constructor innerConstructor = voClass.getDeclaredConstructor(c);

vo = innerConstructor.newInstance(object);通过构造函数创建对象。

* + 1. 反射创建对象

newInstance()只能创建无参构造函数，如果其他构造函数，见上面几行。

* + 1. Class<T> 和Class<?>的区别

T表示模板参数，？是通配符，代表任意类型。

T用在定义一个数据类型时，即类的定义代码中

？用在声明时表示一个通用类型。即实例化后使用的代码中。

* + 1. 例子，初始化成员变量为null字段

Field[] fields = object.getClass().getDeclaredFields();

for (Field field : fields) {

field.setAccessible(true);

try {

Object value = field.get(object);

if (value == null) {

String type = field.getGenericType().toString();

if (type.endsWith("String")) {

field.set(object, "");

}else if (type.endsWith("Integer")) {

field.set(object, 0);

}

}

} catch (IllegalArgumentException | IllegalAccessException e) {

e.printStackTrace();

}

field.setAccessible(false);

}

* + 1. 泛型参数获取

Java泛型有这么一种规律：

位于声明一侧的，源码里写了什么，编译到class文件中还存在；

位于使用一侧的，源码里写什么，编译到class文件中类型均转为object；

举例：

public class myClass<T> {

T a ;

List<Integer> list;

}

myClass<String> obj = new myClass<String>();

编译后：

public class myClass<T> {

T a ;

List<Integer> list; //声明区，泛型保留了

}

myClass<Object> obj = new myClass<Object>();//定义区，泛型从integer变为object。

在说到泛型类型参数时，要先说一下Type接口，Type接口只有一个实现类Class，但是有四个子接口，这四个Type子接口描述了Java泛型的四种形式。常用的为ParameterizedType 该接口表示参数化类型，如 Collection<String>。所以例子中成员变量list的泛型参数可以获取，函数泛型参数也可以获取。

1. 成员变量类型的泛型参数。Type type = field. getGenericType();
2. 成员方法返回值的泛型参数。Type type = method. getGenericReturnType();
3. 成员方法参数类型的泛型参数，包括构造方法。

Type[] typeArr = method. getGenericParameterTypes();

函数获取泛型参数例子：

public Map<Integer,String> test02(){} 可以获取到泛型参数integer string

Method m2 =Generic.class.getMethod("test02", null);

Type returnType = m2.getGenericReturnType();//获取返回类型的泛型

if(returnType instanceof ParameterizedType){

Type [] genericTypes2 =((ParameterizedType)returnType).getActualTypeArguments();

for(Type genericType2:genericTypes2){

System.out.println("返回值，泛型类型"+genericType2);

}

}

敲黑板：原理分析。

Java的泛型和c++区别是：

Template<T>{};

C++使用时，Template<int>{} Template<string>{}，这是两个不同的类型

Java认为模板形参只是一个参数，所以上述两个是同一个类型。也就是运行时用Object代替string、int类型擦出避免了代码膨胀。

但是如果继承Template。

TemplateA<T> extends Template<T>

TemplateB<T> extends Template<T>

使用时，TemplateA<int> 和TemplateB<string>的父类为Template<int>{} Template<string>{}，A和B的父类不一致，必须加以区分，所以用Class具有getGenericSuperclass函数，如果父类是泛型类，则继承ParameterizedType接口，通过getActualTypeArguments()获取模板参数。

实际使用： package javax.ws.rs.core;中的GenericType 类型，

Ps:new GenericType<List<VO\_STATION>>(){} 加{}自动创建匿名子类，

从而 GenericType为父类内部就可以获取到模板参数List<VO\_STATION>，就可以获取全部类型信息。

* 1. Classloader

引导类加载器（bootstrap class loader）：它用来加载 Java 的核心库，即$JAVA\_HOME/jre/lib下面的，是用原生代码来实现的，并不继承自 java.lang.ClassLoader。

扩展类加载器（extensions class loader）：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在lib/ext文件夹下的目录里面查找并加载 Java 类。

系统类加载器（system class loader或者AppClassLoader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

以上三者是父子关系。

Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。

真正完成类的加载工作是通过调用 defineClass来实现的；而启动类的加载过程是通过调用 loadClass来实现的。在 Java 虚拟机判断两个类是否相同的时候，使用的是类的定义加载器。

对于一个类加载器实例来说，相同全名的类只加载一次，即 loadClass方法不会被重复调用。

每个Class对象都会有指向定义它的ClassLoader的引用，通过Class#getClassLoader()可以获得

class ClassLoader {

Class loadClass(String name)

Class defineClass(byte[] b)

URL getResource(String name)

Enumeration getResources(String name)

ClassLoader getParent()

loadClass方法接收一个类的全名称并返回一个Class类的对象。

Class param = systemLoader.loadClass("V1.detect.ParamPeriod");

System.out.println(param.toString());

defineClass方法用来创建JVM所需要的类描述，类型为Class。参数byte数组是从文件或其他地方加载的class字节码

getResource和getResources用来查找资源。给定类的名称或者资源的路径，返回所需的资源。

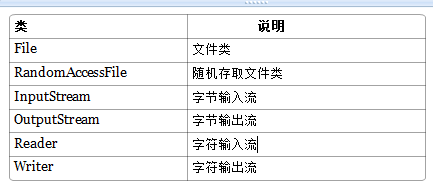
* 1. 匿名内部类和lambda表达式

Lambda是并发处理的。

* 1. Java的IO流

按照操作对象分。分为字符流和字节流。字节流，以字节为单位读取，基类为inputstream和outputstrenm，就是什么stream。字符流，以字符为单位读取，根据字符编码对字节流进行封装，基类为inputReader和outputwriter。转换核心为InputStreamReader和OutputStreamReader

命名规则，以一下字段结尾的类分别表示何种意思：



按照作用分，可分为节点流和处理流类型。

1、节点流标示IO对象，io对象可以是如下：

文　件 ：FileInputStream 、 FileOutputStrean 、FileReader 、FileWriter 文件进行处理的节点流

数　组 ：ByteArrayInputStream、 ByteArrayOutputStream、 CharArrayReader 、CharArrayWriter 对数组进行处理的节点流（对应的不再是文件，而是内存中的一个数组）

字符串 ：StringReader、 StringWriter 对字符串进行处理的节点流

管　道 ：PipedInputStream 、PipedOutputStream 、PipedReader 、PipedWriter 对管道进行处理的节点流

2、处理流包在节点流外面，提高读取效率。如：

缓冲流：BufferedInputStrean 、BufferedOutputStream、 BufferedReader、 BufferedWriter 增加缓冲功能，避免频繁读写硬盘。

转换流：InputStreamReader 、OutputStreamReader实现字节流和字符流之间的转换。

数据流：DataInputStream 、DataOutputStream 等-提供将基础数据类型写入到文件中，或者读取出来

3、处理流和节点流的区分方法，节点流在新建时需要一个数据源（文件、网络）作为参数，而处理流需要一个节点流作为参数。

4、InputStreamReader是字节转换为字符的桥梁，其构造函数传入字节流InputStream作为参数，可以设置字符编码类型。

public InputStreamReader(InputStream in, String charsetName)

5、缓冲流

减少频繁写入的次数，BufferedOutputStream构造函数使用数组长度8192为缓冲。

public BufferedOutputStream(OutputStream out) {

this(out, 8192);

}

读文件：

File file = new File(filePath);

reader = new BufferedReader(new FileReader(file));

while ((tempString = reader.readLine()) != null) {}

写文件：

write = new FileWriter(file)

write.write(prettyJsonString)；

其他需要研究的

java nio库效率更高。

Apache Commons IO

java.util.Scanner类扫描 读取大文件，不需要全部进内存。

Ref：java:io流学习小结

* 1. NIO
     1. 操作系统同步与异步理解

但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。所以同步的就是调用线程还需要进行数据读取，异步的系统自动填充调用线程预先设定的内存区（效率上：操作系统在用户态和内核态之间共享内存，所以异步效率高）

Ref: select、poll、epoll之间的区别总结[整理] ， java nio及操作系统底层原理

Java 的Selector利用epoll实现，channel绑定到Selector上，轮询epoll。

* + 1. Channel与selector

Selector实现一个线程对应多个socket，channel是配合selector使用，绑定到selector上，channel和bio里面的socket对应，ServerSocketChannel、SocketChannel、DatagramChannel，当自己代码查询到selector的四个事件read、write、connect、accept四个事件有消息，则进行相应处理。

Ref: Java 网络IO编程总结（BIO、NIO、AIO均含完整实例代码）

* + 1. AIO函数接口

异步io接口以Asynchronous单词开头，accept、connect、read、write函数不阻塞，当有结果时通过CompletionHandler接口回调相应的completed函数，因为是回调的方式，所以上述四个函数虽然是线程安全的，但是不支持同时操作，当前一个accept未完成并回调completed，继续调用accept函数时会抛出异常。

* + 1. Bytebuffer

使用缓冲区有这么两个好处：

1、减少实际的物理读写次数

2、缓冲区在创建时就被分配内存，这块内存区域一直被重用，可以减少动态分配和回收内存的次数.

见： ByteBuffer常用方法详解。

注意flip()函数，buffer一般是先缓存数据，然后flip，转换为读数据。读完后再重新存数据。

* + 1. TCP端口

标示链接的唯一性是：源端ip和port 目的端ip和port，四个都一样才是重复，所以tcp服务端listen到连接后分配的socket还是使用80监听端口和客户端通信，服务端对上百万客户端只是客户端的地址发生变化。

* 1. Netty

Netty使用多个Channel Handler来达到对事件处理的分离，因为可以很容的添加、更新、删除业务逻辑处理handler。Handler很简单，它的每个方法都可以被重写，它的所有的方法中只有channelRead方法是必须要重写的。

EventLoop，Netty中的EventLoopGroup包含一个或多个EventLoop，而EventLoop就是一个Channel执行实际工作的线程。EventLoop总是绑定一个单一的线程，在其生命周期内不会改变。

很多Channel会共享同一个EventLoop。这意味着在一个Channel保持EventLoop繁忙会禁止其他

Channel绑定到相同的EventLoop。EventLoopGroup可以包含很多个EventLoop

这些ChannelHandler会在程序的“引导”阶段被添加ChannelPipeline中，并且被添加的顺序将决定处理数据的顺序

必须不阻塞IO线程意味着在ChannelHandler中使用阻塞操作会有问题。幸运的是Netty提供了解决方案，我们可以在添加ChannelHandler到ChannelPipeline中时指定一个EventExecutorGroup，EventExecutorGroup会获得一个EventExecutor，EventExecutor将执行ChannelHandler的所有方法。EventExecutor将使用不同的线程来执行和释放EventLoop。ChannelPipeline类的 addLast(EventExecutorGroup group, ChannelHandler... handlers);

Channel是线程安全(thread-safe)的，它可以被多个不同的线程安全的操作，在多线程环境下，所有的方法都是安全的。

Netty中的NIO传输是“zero-file-copy”,也就是零文件复制，这种机制可以让程序速度更快，更高效的从文件系统中传输内容，零复制就是我们的应用程序不会将发送的数据先复制到JVM堆栈在进行处理，而是直接从内核空间操作。

Netty中的传输方式有如下几种：

NIO，io.netty.channel.socket.nio，基于java.nio.channels的工具包，使用选择器作为基础的方法。

OIO，io.netty.channel.socket.oio，基于java.net的工具包，使用阻塞流。

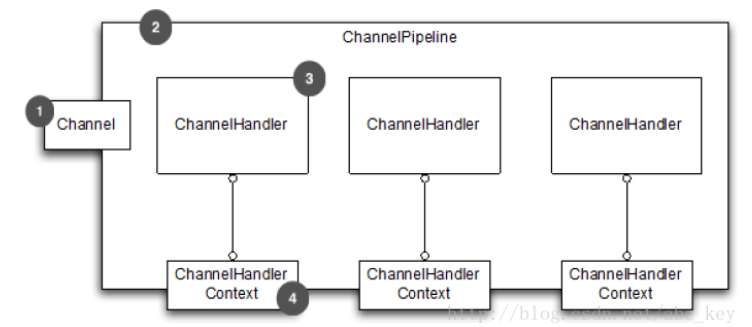
Local，io.netty.channel.local，用来在虚拟机之间本地通信。

Embedded，io.netty.channel.embedded，嵌入传输，它允许在没有真正网络的运输中使用ChannelHandler，可以非常有用的来测试ChannelHandler的实现。

一旦连接，Channel和ChannelPipeline之间的耦合是永久性的。Channel不能附加其他的ChannelPipeline或从ChannelPipeline分离。Channel是数据操作实体，绑定一个ChannelPipeline，里面包括很多channelhander是数据处理动作动作。

使用ChannelInboundHandler、ChannelInboundHandlerAdapter、SimpleChannelInboundhandler这三个中的一个来处理接收消息，使用哪一个取决于需求；大多数时候使用SimpleChannelInboundHandler处理消息，使用ChannelInboundHandlerAdapter处理其他的“入站”事件或状态改变。

关系。channelHandler可以一个实例供多个channelPipeline使用。使用@shareable注解



DefaultChannelPipeline中

tail = new TailContext(this);

head = new HeadContext(this);

组成链表，节点为AbstractChannelHandlerContext，pipeline加入channelhander时，channelhander作为AbstractChannelHandlerContext构造函数的参数，所以hander属于context的成员。

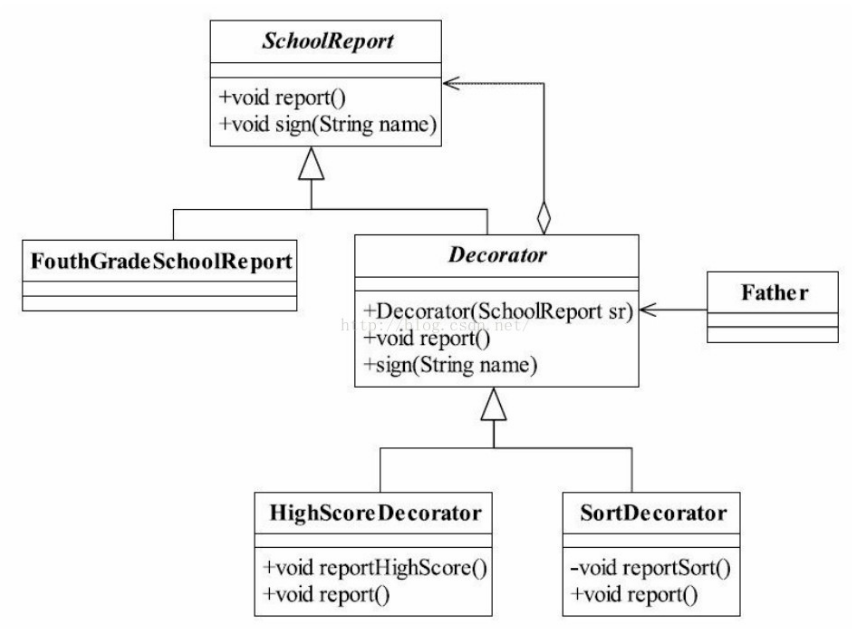
* 1. 代码优化
     1. 尽量使用类、方法的final修饰符，则不会继承，可提高50%效率。
  2. 设计模式
     1. 装饰者与io类

Decorator中有schoolreport类型的成员变量，highscoredecorator等concurrentDecorator实现接口。应用场景：

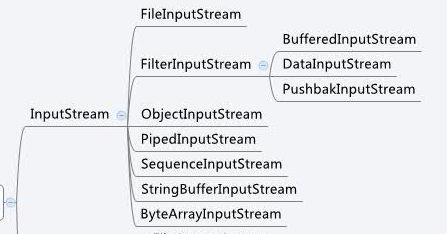
A、需要扩展一个类的功能，或给一个类增加附加功能。

B、需要动态地给一个对象增加功能，这些功能可以再动态地撤销。

C、需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式。



简化：1、就一个FouthGradeSchoolReport,则用其替代SchoolReport；2、就一个HighScoreDecorator，则不需要Decorator，替代即可。



●抽象构件(Component)角色：由InputStream扮演。这是一个抽象类，为各种子类型提供统一的接口。

●具体构件(ConcreteComponent)角色：由ByteArrayInputStream、FileInputStream、PipedInputStream、StringBufferInputStream等类扮演。它们实现了抽象构件角色所规定的接口。

●抽象装饰(Decorator)角色：由FilterInputStream扮演。它实现了InputStream所规定的接口。

●具体装饰(ConcreteDecorator)角色：由几个类扮演，分别是BufferedInputStream、DataInputStream以及两个不常用到的类LineNumberInputStream、PushbackInputStream。

* 1. Java语法
     1. 可变长度参数

public static boolean isBlank(Object...objects){

for (Object object : objects) {

}

}

1、只能作为函数最后一个参数。2、传入的是一个数组，要依次遍历。

* + 1. Default

接口中使用default修饰，给继承类提供一个默认实现，从而给接口新增成员函数时，不需要继承类必须实现该接口。

* + 1. 数组使用

定义： type [] 变量名= {初始化值}

操作：java.util.Arrays，提供许多静态方法。

* + 1. Static

只用在三种地方：成员变量、成员函数、静态代码块。成员函数用法：

Public static int functionName();

另外还有静态内部类：静态内部类无法访问外部类的非静态成员。

非静态内部类：生成对象时依附于外部类实例存在，内部类有外部类的this指针，所以可以访问外部类任意成员，自身不能创建静态成员。对于访问外部类函数里面的局部变量，必须保证局部变量为final类型，局部变量或者局部引用释放后，gc不会回收这个内存，而且保证外部和内部类使用的变量要一致。（没有完全理解）。Todo，gc回收机制。构造内部类对象时，innerClass f = outerInstance.new innerClass ();

创建内部类对象，需要依赖外部内对象VO\_FREQUENCY f = resource.new VO\_FREQUENCY();

* + 1. Final

可以用来修饰类、成员函数、方法；

Final类：不能被继承，所有方法均为final；

Final成员函数：防止被子类重载，子类可继承。提高效率，final函数会转为内嵌机制提高效率。

Final成员变量：final成员变量可以不在定义时给初始值，但是必须在构造函数中赋予一个值，从而不同实例有不同的该final成员变量值。

Final函数入参。在编译期不一定能初始化。函数内实现的runnable接口，可以访问final入参，即Final变量可以跨线程使用。

Final修饰引用，表示指向的引用地址不能修改，如果指向的是list，修改list里面的数据是可以的。

* + 1. Serializable 序列化

Serializable是一个接口，实现该接口后，对象在内存中的二进制存储数据可以被持久化，即持久化的是内存中的二进制数据。对象序列化如下：

byte[] ret = null;

ByteArrayOutputStream byteStream = new ByteArrayOutputStream();

ObjectOutputStream objectStream;

try {

objectStream = new ObjectOutputStream(byteStream);

objectStream.writeObject(this);

objectStream.flush();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

ret = byteStream.toByteArray();

* + 1. null关键字

是java语言的一个关键字，表示引用类型的默认值，表示引用的对象未初始化，所以instanceof该对象引用返回为false，该特征可用于在Integer类型转为int基本类型时进行判断，如果Integer对象值为null，转为int基本类型会报错。

* + 1. 数据类型

基本数据类型：数值型（整数byte,short,int,long，浮点float，double）、字符型:char、布尔型：boolean

引用数据类型：class 、interface、数组。

详细：float为4字节，double为8字节。Float定义，float a = 0;01f; f不可少。

Int为4字节，long为8字节。

基本数据类型都有对应的类对象：int – Integer，long-Long,float-Float,boolean – Boolean，char – String

这些数值类如Integer，有valueof(int)自动将int类型转为类对象，称为自动装箱。如Integer i=5;

同样类对象有intValue()自动转为int，称为自动拆箱。

0x81 这种16进制java默认为int类型，字节码为1000 0001 ，解释为int时为129，但是超出了byte的范围-128到127，则需要强制转换为byte类型。0x00到0x7f在byte范围内，则不需要强转。

代码中明确写的数字默认是int类型，如short t = 20;如果数字超过short范围，需要显示类型转换。函数参数类型如果是short，status.setLength((short) 2);编译器认为2是int，使用时都需要类型转换。

* + 1. Clone是浅拷贝

实现深复制，可以利用序列化，先将对象序列化，再反序列化

//将对象写到流里

ByteArrayOutoutStream bo=new ByteArrayOutputStream();

ObjectOutputStream oo=new ObjectOutputStream(bo);

oo.writeObject(this);

//从流里读出来

ByteArrayInputStream bi=new ByteArrayInputStream(bo.toByteArray());

ObjectInputStream oi=new ObjectInputStream(bi);

return(oi.readObject());

或者利用povoconverter。进行对象拷贝。

* + 1. Equals的写法

public class study {

int num;

Data data;

@Override

public boolean equals(Object obj){

if (this == obj) {

return true;

}

if (obj == null || this.getClass() != obj.getClass()) {

return false;

}

study s = (study)obj;

if (num == s.num &&

data == s.data) {

return true;

}else

return false;

}

}

重写了equals函数，必然需要重写hashcode函数，该函数在对象作为hash数据结构主键时使用。

Hashcode为32位即4字节的int，各数据类型的成员转为hashcode的方法如下：

Byte，char，short，int ---- （int）obj；

Long --- (int)(obj^obj>>>32)

Float --- Float:floatToBits(obj)

Double --- long bits = Double.doubleToLongBits(obj) ，再将Long转int；

* + 1. 常见数据结构

Collection接口，表示一组object对象。具体实现需要继承List和Set两个接口。

List接口实现如下。

ArrayList：线性表，效率高，线程不安全；

Vector：如果需要线程安全的ArrayList，用Vector；

LinkedList：链表，频繁的插入删除用这个；

Map接口，未继承Collection接口，具体实现如下：

HashMap，非线程安全，key可以为null，

HashTable，线程安全，key不为null，但是使用一个锁锁住整个Map，多线程下效率低，不采用，concurrentHashmap保证线程安全条件下，给map分段加锁，效率高。

遍历map时删除一个元素，需要使用迭代器，且处理一个对象时，只能对map进行一次修改，不能在一次处理中删除一个元素后又增加新元素，否则会使迭代器失效。

Iterator<Map.Entry<ScheduledFuture<?>, Runnable>> itor = futures.entrySet().iterator();

List<Runnable> temp = new ArrayList<Runnable>();

while (itor.hasNext()) {

Map.Entry<ScheduledFuture<?>, java.lang.Runnable> entry = itor.next();

ScheduledFuture<?> f = entry.getKey();

Runnable thread = entry.getValue();

System.out.println(new Long(f.getDelay(TimeUnit.SECONDS)).toString() + " delayed ");

if (f.getDelay(TimeUnit.SECONDS)< -5 ) {

f.cancel(true);

itor.remove();删除元素

temp.add(thread);新增的元素在迭代完之后再增加到map。

}

}

* + 1. String

String对象是对字符数组的封装，所以不可修改内容；对String对象的任何改变都不影响到原对象，相关的任何change操作都会生成新的对象。

String s = “abc” ；abc存储在常量池中；

String s = new String(“abc”) 存储在堆区中；

举例：

String str1 = "hello world";

String str2 = new String("hello world");

String str3 = "hello world";

String str4 = new String("hello world");

System.out.println(str1==str2); //false

System.out.println(str1==str3); // true

System.out.println(str2==str4); //false

因此在上述代码中，String str1 = “hello world”;和String str3 = “hello world”; 都在编译期间生成了 字面常量和符号引用，运行期间字面常量”hello world”被存储在运行时常量池（当然只保存了一份）。通过这种方式来将String对象跟引用绑定的话，JVM执行引擎会先在运行时常量池查找是否存在相同的字面常量，当然存在，因为编译器已经存入常量池，所以直接将引用指向已经存在的字面常量；

总所周知，通过new关键字来生成对象是在堆区进行的，执行到new指令，才在堆区创建"hello world"字符串，不检测是否有重复字符串（运行时扫描堆数据代价太大），所以引用地址不一致。

S = s + “def” 其实重写生成了新的字符串对象，效率低。所以进行字符串拼接使用stringbuilder的append和insert函数。Stringbuilder的线程安全版本为stringbuffer。

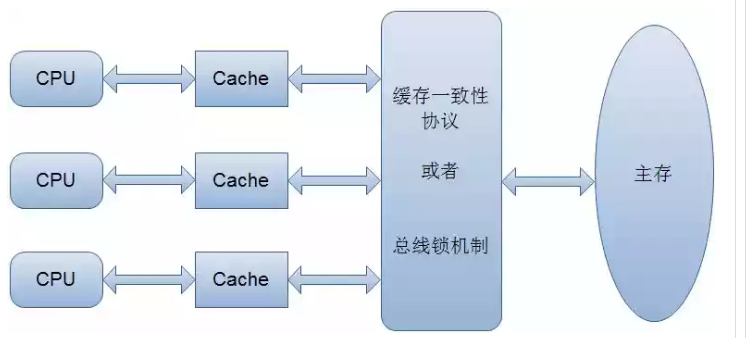
* + 1. Instanceof

判断是否是某一个类型的实例，派生类引用也是基类的实例，

* + 1. Volatile

该关键字要求各线程在读取变量值时从主内存中该变量的值（见解释），对变量修改还是在cpu缓存中，修改后立刻同步该变量值到主内存。

解释：其实不是每次都从内存读，毕竟内存速度低，而是主内存中该变量值发生修改后，缓存一致性协议会告诉该线程的cache，说该变量值失效，需要从主内存重新加载，如果缓存一致性没认为该变量失效，则不需要加载。以上操作等价于每次都从主内存读。



基础知识，三大特性。

原子性：一条语句是原子的即中间不会打断，不会有其他操作影响该语句操作。

Int a=1；int是32位的，赋值在一个cpu周期完成。原子操作

Object b = c; c引用对象赋值给b引用，引用占32位，在一个cpu周期完成。原子操作

long d = 100.9; long占用64位，分高4字节和低4字节分别赋值，需要两个周期，不是原子操作。

e++；等价于读e的值，tmp = e+1; e=tmp;三步操作，不是原子操作。

可见性：其他线程修改某个变量之后，其他线程读取该变量值为最新的值。Volatile保证可见性

//线程1执行的代码

int i = 0;

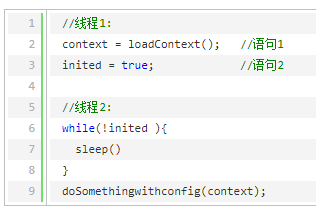
i = 10;

//线程2执行的代码

j = i;

线程1赋值为10，在缓存中，线程2从主内存读到的仍然为0，不可见i最新值，所以不满足可见性。

有序性：即程序执行的顺序按照代码的先后顺序执行。一个语句块内部，只要保证结果不变，即语句之间无依赖，语句顺序编译器可以任意调节。但是多线程下会有问题。如下图。语句1和语句2在线程1内部看无依赖关系，但是在线程2内部，context依赖inited状态，线程1并不知道，语句2执行了，线程2运行doSomethingwithconfig时，线程1未loadcontext，context为空指针。



要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。Volatile能保证三大性么？

1、保证可见性

2、不保证原子性。Volatile只是保证读的时候读到主内存最新值，即使修饰e,进行e++操作，等价于读e的值，tmp = e+1; e=tmp;三步操作，第一步读完后主内存修改e的值，但是tmp =e+1不需要读e值，使用的仍然是旧e值，所以不能保证。

3、保证volatile变量自身的有序性，即该变量之前的语句肯定在前面执行，该变量之后的语句在后面执行，所以上图语句1在语句2之前执行，就肯定没有问题。

解释：

假如某个时刻变量inc的值为10，线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；　然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的缓存行无效，所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。　然后线程1接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。　　那么两个线程分别进行了一次自增操作后，inc只增加了1。

　　解释到这里，可能有朋友会有疑问，不对啊，前面不是保证一个变量在修改volatile变量时，会让缓存行无效吗？然后其他线程去读就会读到新的值，对，这个没错。这个就是上面的happens-before规则中的volatile变量规则，但是要注意，线程1对变量进行读取操作之后，被阻塞了的话，并没有对inc值进行修改。然后虽然volatile能保证线程2对变量inc的值读取是从内存中读取的，但是线程1没有进行修改，所以线程2根本就不会看到修改的值。

对变量a的赋值要满足原子性的条件是：　1）对变量的写操作不依赖于当前值a；　　2）该变量没有包含在具有其他变量的不变式中，即a=b，如果b也是多线程修改的，不能保证b的值不会在b读入cpu缓存后，其他线程不修改。

如果变量a赋值依赖a自身状态，使用atomic变量，

Volatile的使用场景：适合直接赋值的场景，通常用来做标志位，如某个操作完成、发生中断或者状态的标志。

补充：

1. 在java中，非volatile的double和long类型的写操作被分成了两个32位的写操作，所以对于64位的double和long类型的写操作就不是原子操作。

2. 对于volatile类型的long和double类型的写操作是原子操作

3. 对于引用类型Long，Double的读写操作都是原子操作，无论引用类型的实际类型是32位的值还是64位的值。

4.int以及其他类型的读写操作都是原子操作，但是不用volatile修饰，会读到旧的int值，不能读到最新的。

* + 1. 同步synchronized

是java关键字，

1、仅一个对象的代码同步

修饰方法 public synchronized void fun(){}

修饰语句块。不同线程操作同一个对象所有被synchronized修饰的函数需要同步，但是不同函数逻辑无需同步时，可以如下，同时只能一个线程能操作该对象的mem1 成员。mem1可以为this，表示括号内对象本身应该同步。

public class PrivateLock {

private final Object myLock = new Object();//内部变量Object保证操作widget的线程同步。

Widget widget;

void someMethod() {

synchronized (myLock) {

// Access or modify the state of widget

}

}

}

2、所有对象的代码需要同步

public void method2() {

synchronized(ClassName.class) { 括号内为类自身

// todo

}

}

该关键字不具有继承性，子类需要同步，仍然需要加上该关键字。

* + 1. 基本类型对象同步Atomic包

原理解析，以AtomicInteger类为例：包括一个volatile变量value，一个native的cas方法（compare and set）。

Private volatile int value;// 保证读取volatile变量值永远是最新的

public final int getAndSet(int newValue) {

for (;;) {

int current =value; //1

if (compareAndSet(current, newValue)) //2

return current;

}

}

在1处，读取volatile变量value是原子的，保证每次读到最新的，但是读取并修改从java语言层面做不到原子性，所以使用native的compareAndSet。2处，如果current的值和value一致，则将value设置为newValue，否则返回false，使用for循环直到成功。compareAndSet操作是原子的，利用cpu硬件均支持的cas功能（compare and set）。

所有以concurrent开头的数据结构都是利用cas功能实现的。

所有的类成员函数都是线程安全的，get();if(equal) set();两个语句一起就不是原子的了 ,所以出现getAndSet函数，将两个语句形成一个原子的操作，否则只能自己给这两个语句加锁了。

AtomicInteger类

int getAndSet(int newValue)；等价于原子的 a = newValue;

int addAndGet(int delta) ; 等价于原子的a = a+delta;

AtomicReference类

getAndSet(V newValue) 同步操作对象，设置新值。如果新值不依赖旧值，否则使用下面的函数。

compareAndSet(V expect, V update); 当update的值依赖expect即旧状态时，比如update = expect +2,但是执行完之后，expect有可能被其他线程改变，必须判断expect没变才可以，可以这么写：

boolean flag = false;

while (!flag) {

Element storedElement = reference.get();

Element newElement = new Element(storedElement.x + 1, storedElement.y + 1);

flag = reference.compareAndSet(storedElement, newElement);

}

LongAdder类

多个线程统计数字，可以同步的累加数字，因为内部分散到多个变量，性能高一点。

add(long x) 增加值

sumThenReset() 取计算结果

线程安全的集合类

ConcurrentHashMap 如putIfAbsent函数

ArrayBlockingQueue 多线程安全的队列，队列空，取数据线程会阻塞

CopyOnWriteArrayList 并行的Arraylist 使用迭代器遍历，后面队列再修改不产生影响。

Ref: java8并发编程：原子变量和concurrentMap ，gitbook 。

Java 多线程编程中常用的同步方法与工具。

* + 1. 队列

BlockingQueue是统一的接口；对数据的加入和获取提供多种接口，在队列满或者队列无数据时，可以阻塞、抛出异常、返回false、定时返回false。

实现使用LinkedBlockingQueue和ArrayedBlockingQueue，

当然还有优先级队列，双向队列等数据结构，满足常用使用需求。

* + 1. 线程池

ExecutorService exector = Executors.newSingleThreadExecutor();

// ExecutorService exector = Executors.newFixedThreadPool(5);固定大小线程池

exector.submit(() -> System.out.println(Thread.currentThread().getName()));

循环执行某个线程

ScheduledExecutorService exector = Executors.newScheduledThreadPool(2);

exector.scheduleWithFixedDelay(

() -> System.out.println(Thread.currentThread().getName())

, 1, 3, TimeUnit.SECONDS);

取线程结果需要继承callable接口

Callable<Integer> task = ()->{

TimeUnit.SECONDS.sleep(1);

return 123;

};

Future<Integer> result = exector.submit(task);

System.out.println(result.isDone());

System.out.println(result.get());

关闭方式

try {

exector.shutdown();

exector.awaitTermination(5, TimeUnit.SECONDS);

} catch (InterruptedException e) {

// TODO: handle exception

System.out.println("tasks interrupted");

}finally{

exector.shutdownNow();

}

总结：线程池工厂Executors

线程池对象类型 ExecutorService ScheduledExecutorService（循环执行）

线程执行代码runnable接口、 callable接口（有返回值）

关闭时，不管try有没有异常，finally总是执行，如果shutdown成功，则shutdownNow什么也不做，否则强制关闭。

* + 1. 异常

Java异常exception包括Runtime异常（运行时异常）和其他异常如ioexception、反射调用找不到函数等等，Runctime异常不处理可以编译通过，其他异常必须显示在代码中进行处理。方式一，编写catch块，方式二，函数定义中throw出去，给外部作用域处理。

异常如果一直未catch，往出栈，直到线程崩溃关闭，其他线程不受影响。

Finally是永远执行的，在函数return前执行，用于释放资源文件、网络资源等。Final不能代替catch，异常总是要被捕捉，否则线程崩溃。

根据throwable对象打印堆栈信息，返回值可以通过日志输出.

public static String getStackTrace(Throwable throwable){

StringWriter sw = new StringWriter();

PrintWriter pw = new PrintWriter(sw);

try {

throwable.printStackTrace(pw);

return sw.toString();

} finally {

pw.close();

}

}

* 1. Java编码规范

编程习惯：

1. 中括号是数组类型的一部分,数组定义如下:String[] args;
2. POJO 类必须写 toString 方法。使用 IDE 的中工具:source> generate toString

时,如果继承了另一个 POJO 类,注意在前面加一下 super.toString。

说明:在方法执行抛出异常时,可以直接调用 POJO 的 toString()方法打印其属性值,便于排

查问题。

* 1. Java运行机制
     1. 内存布局

堆区：存放对象实例。

栈：存放基本数据类型和对象引用（不是对象本身，指向堆区的引用）

方法区：1、数据区：常量、静态对象、函数方法。2、执行上下文。3、指令区。

* + 1. 加载顺序

如下所示，1为静态编译阶段，2-4为类加载阶段，只是把类的结构信息和静态成员加载到jvm，5为类对象实例化阶段。

1. 源码编译为class文件，包括常量池和方法字节码两部分。
2. 加载阶段。Class文件中常量、方法字节码载入方法区的运行时数据结构，java堆中生成描述类结构信息的元对象，通过该对象访问方法区数据。
3. 准备阶段。在方法区给类中静态成员分配内存，赋给0值。
4. 初始化阶段。将类定义中静态成员的值覆盖该成员，在初始化阶段编译器会收集类中的**类静态变量**（区别实例变量）的赋值动作和静态语句块中的语句，而不是准备阶段的0值。

触发初始化的时机：创建类实例时，调用类的静态方法时，访问类成员变量或者为某个类变量赋值时，子类初始化触发父类初始化。总之，需要知道类的结构，需要内存分配时都会触发初始化。

反例：访问用final修饰的常量时，第2步就已经分配内存，所以不会触发初始化阶段。

1. 实例化阶段。在堆区给类对象分配内存。先初始化块进行初始化，继而构造函数进行初始化。实例化时父子类，先实例化父类包括父类的成员变量赋值及构造函数，再派生类，

public class Base

{

Base() {

preProcess();

}

void preProcess() {}

}

public class Derived extends Base

{

public String whenAmISet = "set when declared";

@Override void preProcess()

{

whenAmISet = "set in preProcess()";

}

}

public static void main(String[] args)

{

Derived d = new Derived();

System.out.println( d.whenAmISet );

}

初始化流程：

进入Derived 构造函数。

Derived 成员变量的内存被分配。

Base 构造函数被隐含调用。

Base 构造函数调用preProcess()。

Derived 的preProcess 设置whenAmISet 值为 “set in preProcess()”。

Derived 的成员变量初始化被调用。

执行Derived 构造函数体。

在4阶段初始化静态变量，5阶段初始化普通成员变量，都是先赋值动作和静态语句块（按照代码顺序），再执行构造函数，子类的所有初始化在父类后面。

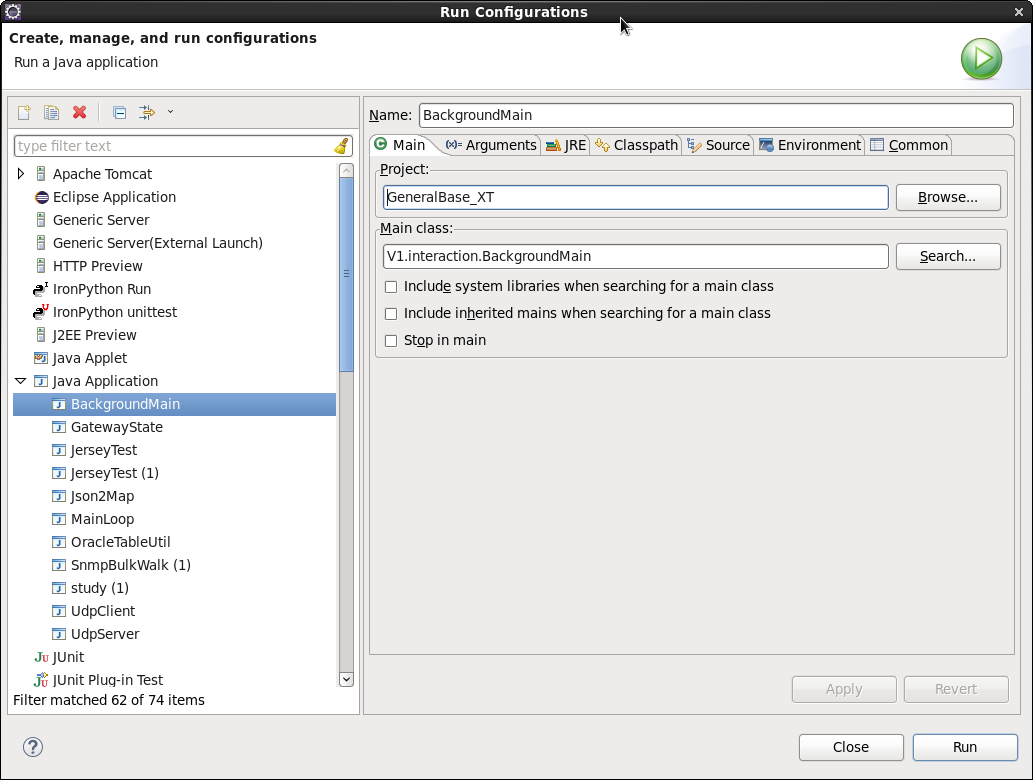
语句块的优势就是把各种构造函数里面重复的赋值放到一起去。

Ref:java类加载机制详解、JAVA构造时成员初始化的陷阱

* 1. Class运行机制

Main class 中的路径包括了包名V1.interaction 所以在classpath中需要加上V1的路径，组成完整的Main Class路径。Classpath中有advanced中可以添加v1文件夹路径。

运行时需要的依赖包也需要在classpath中加上。



获取classpath路径。

String path = System.*getProperty*("java.class.path");

String[] pathsList = path.split(":");

**for** (**int** i = 0; i < pathsList.length; i++) {

System.***out***.println(pathsList[i]);

}

获取当前工作目录，两种方法：

方法一：

System.***out***.println(System.*getProperty*("user.dir"));

方法二：

File directory = **new** File("");//设定为当前文件夹

**try**{

System.***out***.println(directory.getCanonicalPath());//获取标准的路径

System.***out***.println(directory.getAbsolutePath());//获取绝对路径

}**catch**(Exception e){}

Web项目中使用如下获取路径

this.getClass().getClassLoader().getResource("/").getPath()

* 1. Controler的参数
     1. @PathVariable

@RequestMapping(“/id/{user}”)

Function(@PathVariable(“user”) String user ){}自动将路径中的user映射到user参数中。

* + 1. @RequestParam

Function(@RequestParam (“user”) String user ){}自动将表单中的或者url查询参数中的user映射到user参数中。

直接填充

Function(User user ){}，对于一个pojo类型参数，将表单参数值自动与User的成员名称进行对应。

* 1. 注解
     1. @Autowird

在bean注入时使用，根据类型自动注入bean对象。可以放在成员变量前，也可以放在函数前，放在函数前，自动注入与函数参数类型匹配的实例。@Autowired注入时，有多个实现该接口的候选项候选项

1、@Primary注解制定候选优先级

2、@Autowired后面加@Qualifier（“implB”）指定注入 Bean 的名称。@Autowired 和@Qualifier 结合使用时，自动注入的策略就从 byType 转变成 byName 了

@Resource，默认根据name注入，与@Autowired不同，也有type参数按 byType 自动注入。

* + 1. @Component

标示该类为一个bean，在view层含义相同的为@Controller，服务层@service，存取层@Repository，还有@Configuration，在spring配置文件中可以设置只扫描某些类的bean

<!-- 使用annotation 自动注册bean, 并保证@Required、@Autowired的属性被注入 -->

<!-- 此处只加载service相关注解，不再加载controller相关，用于事务管理 -->

<context:component-scan base-package=*"com.cetc28"*>

<context:exclude-filter type=*"annotation"* expression=*"org.springframework.stereotype.Controller"*/>

<context:exclude-filter type=*"annotation"* expression=*"org.springframework.web.bind.annotation.ControllerAdvice"*/>

</context:component-scan>

@Bean也生成一个Bean，但是放在方法前，表示返回值是一个bean，如：

@Bean(name="consumerConfig")

自动注入时，使用名称consumerConfig进行注入。如果不加name属性，以函数名作为bean的名称

常配合@Configuration注解使用，其表示一组bean，即同一个类的多个实例。@Component只能表示一个类型。

* + 1. @SuppressWarnings

抑制编译器的警告信息，

* 1. 路径

web应用部署在webapps下面，下面是各工程文件夹，即工程的根目录，下面有WEB-INF 文件夹，tomcat隐藏该文件防止被外部直接访问，类路径WEB-INF/Class。

request.getContextPath() 获得工程的根路径,即工程文件夹路径，浏览器路径前面会加上服务器容器的路径。jquery语法为${ pageContext.request.contextPath }

相对路径：相对于本文件位置；

绝对路径：相对与工程路径，包括工程名称。浏览器前面的路径+绝对路径就算用户访问的资源。*/CommSystem/**media/assets/global/plugins/jquery.min.js。 其中CommSystem是工程名。*

<c:url value=“以根路径为参考点，”> 就算不许要 *CommSystem，直接 media*

* 1. web.xml解析

web.xml的加载顺序

tomcat会加载web.xml文件，<context-param>-> <listener> -> <filter> -> <servlet>

<listener>

主要用于监听Web应用事件，其中有两个比较重要的WEB应用事件：应用的启动和停止（starting up or shutting down）和Session的创建和失效（created or destroyed）。应用启动事件发生在应用第一次被Servlet容器装载和启动的时候；停止事件发生在Web应用停止的时候。Session创建事件发生在每次一个新的session创建的时候，类似地Session失效事件发生在每次一个Session失效的时候。为了使用这些Web应用事件做些有用的事情，我们必须创建和使用一些特殊的“监听类”。它们是实现了以下两个接口中任何一个接口的简单java类：javax.servlet.ServletContextListener或javax.servlet.http.HttpSessionListener，如果想让你的类监听应用的启动和停止事件，你就得实现ServletContextListener接口；想让你的类去监听Session的创建和失效事件，那你就得实现HttpSessionListener接口。

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

Spring对ServletContextListener接口的实现为ContextLoaderListener， 在tomcat启动时监听到后，spring以参数contextConfigLocation的值applicationContext.xml为配置文件，实现启动。classpath是指 WEB-INF文件夹下的classes目录。 Classes\*不仅会到classes路径，还包括jar文件中(classes路径)进行查找。

Filter

Filter可认为是Servlet的一种“加强版”，主要用于对用户请求request进行预处理，也可以对Response进行后处理，是个典型的处理链。使用Filter的完整流程是：Filter对用户请求进行预处理，接着将请求HttpServletRequest交给Servlet进行处理并生成响应，最后Filter再对服务器响应HttpServletResponse进行后处理。

Filter必须实现javax.servlet.Filter接口，在该接口中定义了三个方法：

void init(FilterConfig config)：用于完成Filter的初始化。FilteConfig用于访问Filter的配置信息。

void destroy()：用于Filter销毁前，完成某些资源的回收。

void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)：实现过滤功能的核心方法，该方法就是对每个请求及响应增加额外的处理。

Servlet

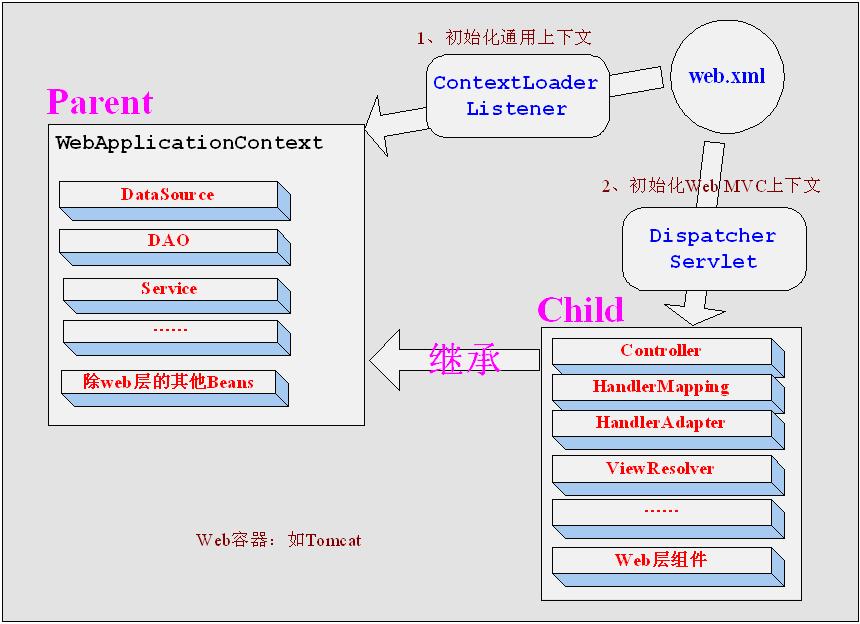
Servlet是个特殊的java类，继承于HttpServlet。

创建Servlet实例有两个时机：1、客户端第一次请求某个Servlet时，系统创建该Servlet的实例，大部分Servlet都是这种Servlet。2、Web应用启动时立即创建Servlet实例，即load-on-start Servlet。

每个Servlet的运行都遵循如下生命周期：1、创建Servlet实例。Web容器调用Servlet的init()方法，对Servlet进行初始化。2、Servlet初始化后，将一直存在于容器中，用于响应客户端请求，如果客户端发送GET请求，容器调用Servlet的doGet()方法处理并响应请求；如果客户端发送POST请求，容器调用Servlet的doPost()方法处理并响应请求。或者统一使用service()方法处理来响应用户请求。3、Web容器决定销毁Servlet时，先调用Servlet的destory()方法，通常在关闭Web应用时销毁Servlet实例。

默认查找MVC配置文件的地址是：/WEB-INF/${servletName}-servlet.xml

常用做法。ContextLoaderListener初始化的上下文加载的Bean是对于整个应用程序共享的，一般如：DAO层、Service层Bean；DispatcherServlet初始化的上下文加载的Bean是只对Spring MVC有效的Bean，如：Controller、HandlerMapping、HandlerAdapter等，该初始化上下文只加载Web相关组件。如下图所示。



* + 1. <mvc:annotation-driven />

提供了HttpMessageConverter，使用@ResponseBody注解时，根据请求accept的数据类型属性，如json，则HttpMessageConverter自动将java的map对象转换为json对象进行传输。

同理，使用@RequestBody注解时，根据Request的content-type部分类型，选择合适的转换器进行转换。

一下对默认转换器进行一些设置。

<mvc:annotation-driven>

<mvc:message-converters register-defaults="false">

<bean class="org.springframework.http.converter.FormHttpMessageConverter"/>

<bean class="org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter"/>

<bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">

<property name="objectMapper" ref="objectMapper"/>

</bean>

<bean class="org.springframework.http.converter.StringHttpMessageConverter">

<constructor-arg name="defaultCharset" value="UTF-8"/>//默认编码

</bean>

</mvc:message-converters>

</mvc:annotation-driven>

Ref: 使用spring MVC构建RESTful Web Services(二)：消息转换

* 1. Http协议



Accept: 表示需要返回什么类型的数据，

text/html : HTML 格式

text/plain :纯文本格式

text/xml :XML 格式

image/gif :gif 图片格式 image/jpeg :jpg 图片格式 image/png:png 图片格式

application/x-www-form-urlencoded : <form encType=””>中默认的 encType,form 表单数据被编码为key/value 格式发 送到服务器(表单默认的提交数据的格式)。

multipart/form-data : 当你需要在表单中进行文件上传时,就需要使用该格式;

application/xhtml+xml :XHTML 格式

  application/atom+xml :Atom XML 聚合格式

application/pdf :pdf 格式

application/octet-stream : 二进制流数据(如常见的文件下载)。

application/xml: XML 数据格式

application/json : JSON 数据格式

Content-Type：表示请求数据的格式

* 1. Tomcat配置

Open对tomcat进行配置

Add and Remove 将工程项目添加到发布目录

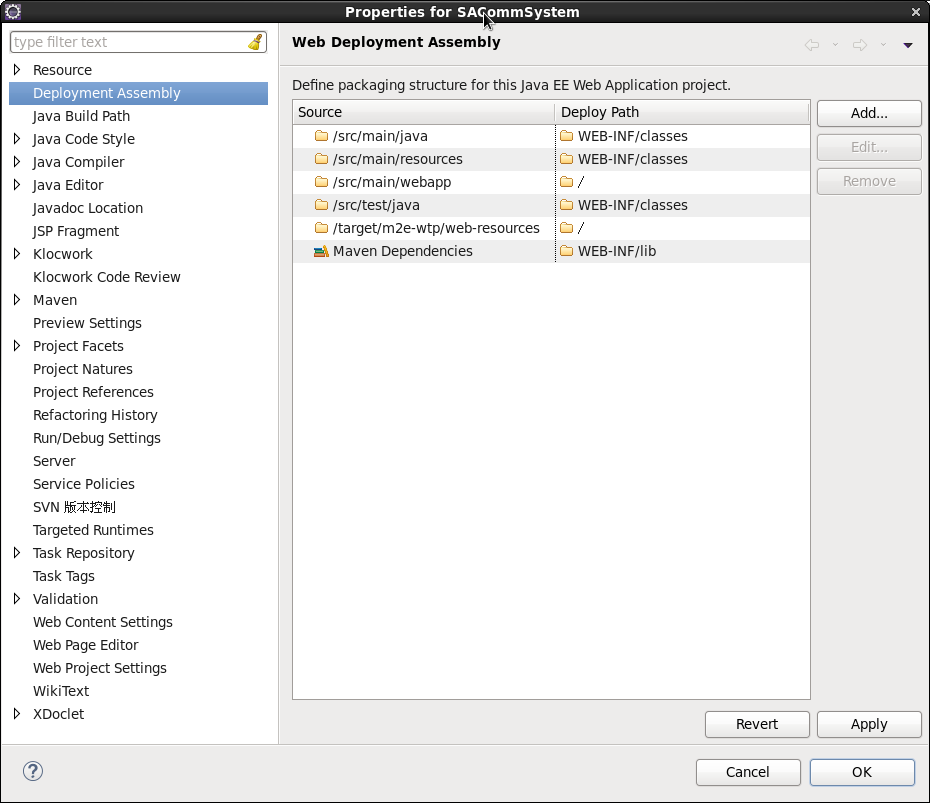
命令行启动tomcat：

在tomcat/bin的文件夹下面，

1. 直接启动，./startup 关闭 shut
2. 作为服务启动nohup ./startup.sh &
3. ./catalina.sh run

Ref:linux 下tomcat服务的相关命令

项目部署：



以上是各文件编译后的目录，/代表根目录。Java文件编译为class文件，放到web-inf/classes文件中。

Tomcat发布时，如果打成war包，则/代表根目标。如果server location配置为use workspace metadata，则路径为工程下的隐藏目录。

/home/workspace/.metadata/.plugins/org.eclipse.wst.server.core/tmp1/wtpwebapps。

配置页面

Configuration path中的路径中的配置文件即servers中的配置文件，在tomcat启动时会自动同步到tomcat配置文件中，配置的所有结果均在这里。

Modules中可以配置工程路径，工程路径和工程名不一致的原因就在此，配置结果在servers.xml中<Context docBase="GeneralBase\_XT" path="/GeneralBase" reloadable="true" source="org.eclipse.jst.j2ee.server:GeneralBase\_XT"/></Host>

* 1. Po bo vo dto dao

Po：persistent object 持久层对象；

Bo：business object 业务层对象，业务简单被po替代；

Vo：view object 视图层；

Dto：data transfer object 数据传输对象；常被vo替代，用于前后台数据交互。

Dao：data access object，获取po的数据库操作对象；

他们都是pojo，即简单的java bean对象，用来进行数据描述的。

1. Hibernate
   1. 概念

SessionFactory，线程安全的会话工厂类，管理数据库连接、缓存等资源。

Session，一次数据库操作，线程不安全，SessionFactory的getCurrentSession（）方法通过使用threadLocal方法将session和thread一一绑定，保证一个线程就一个session，此时就安全了。

Transaction，事务，hibernate利用依赖的驱动、数据库事物等进行自身事物的具体实现。

Hibernate中可持久化对象状态：

Transient，临时状态，刚刚new出来，未保存到session和数据库中的对象；

Persistent，持久化对象， session一级缓存中对象，可能在数据库中（新new的对象，刚save后，只在session中），可以对其数据库操作；

Detached，分离的对象，游离的状态，曾经在session一级缓存中对象，但是session一级缓存因为清空或者被一级缓存清理出来了，则对该对象进行属性修改，session不做任何操作，则不会反映到数据库对应数据上。

* 1. JPA注解

@Transient 实体类某些成员变量不需要存在数据库中，或者成员函数不是get，set函数

* 1. 扩大session范围

在spring中配置OpenSessionInViewFilter，可以让一个session为整个web服务工作。在web.xml中配置

* 1. HQL

大小写不敏感，但是类名、包名要和实际一致，从而通过反射可以查找到。

* 1. 数据访问的层次

JDBC是java规定的一套数据库访问接口规范，各数据库均提供数据库访问驱动，遵从这套规范。Mysql的驱动为mysql-connector-java-version.jar；oracle也有。

往上层是数据库连接池，封装了jdbc的接口，提供很多数据库连接，及时拿及时用。如org.apache.commons.dbcp2。

往上是hibernate等各类orm框架，hibernate的session中的连接就是从数据库连接池获取的。

备注：如果java程序，直接通过jdbc访问数据库，则只需要各类数据库连接驱动即可。

* 1. 缓存机制

一级缓存，在每个线程的session中，只有flush()或者close()或者commit()时才会写到数据库中。Session关闭后，一级缓存也消失，新的session需要重新建立一级缓存。

二级缓存，在sessionFactory中，是全局缓存，默认关闭。一般设置为read-only，即在不需要修改的实体类上设置二级缓存，提高效率。

查询缓存，一二级缓存都是缓存实体对象，查询缓存将检索的HQL语句和结果以键值对的形式缓存起来，默认关闭。也存在N+1问题，见iterator的描述，所以开启查询缓存必须开启二级缓存。

加载实体对象的过程。

1. 在一级缓冲，通过实体类型和主键进行查找，如果查找不到，在nonexist记录（之前执行过相同实体类型和主键查找，且未找到，都会在其中记录）
2. 在二级缓存查找；
3. 在查询缓存查找；
4. 在数据库检索，如果未找到更新一级缓存的nonexist记录；

Ref:hibernate缓存机制详细分析。

* 1. Session的接口

Query = session.createQuery();

List方法不读取缓存，直接从数据库查询，同时将结果写入到session的一级缓存。

Iterator方法读取缓存，（iterator方法存在N+1问题，1是要先检索所有的id字段，多了一次检索，接着根据N个id产生N条检索语句，效率低。但是如果list方法将对象检索到一级缓存，则iterator方法可以直接从缓存中检索，仅仅需要一条检索id的语句）

Session接口

Flush，会将缓存对象更新到数据库中；

Refresh，与flush相反，以数据库数据更新缓存数据；

Save，transient态变为persistent，除非指定数据库自增方式设置主键，否则不触发insert语句；

Persist，

Get， 立即加载对象，首先检索缓存，再数据库。不确定是否有该条数据时使用get，查询不到返回null。结果写入缓存

Load，延迟加载对象，在你确定该数据库存在该对象数据时使用，所以查询不到抛出异常，返回的对象在第一次被使用时才加载对象。结果写入缓存

Update，一般用于将detached态变为persist态，以便提交数据库时能修改对应数据。

Merge，类似update，如果是新对象，则等同于save，插入到insert数据。

Delete，删除缓存对象，不立即更新数据库。

1. Linux
   1. 常见命令
      1. 以Root用户打开文件管理器

sudo nautilus

* + 1. 显示隐藏文件夹

Ctrl + H

* + 1. 杀死某些进程

ps -aux | grep apt

sudo kill <pid>

* + 1. Linux安装软件

sudo gdebi sogoupinyin.deb

rpm –ivh \*\*.rpm

* + 1. 帮助

Type 命令 查看命令类型

内部命令：#help COMMAND

外部命令：

<1> # COMMAND –help # COMMAND -h

<2> 使用手册(manual) # man COMMAND

<3> 信息页 # info COMMAND

<4> 程序自身的帮助文档等 README INSTALL ChangeLog

* + 1. 解压

tar –zxvf

* + 1. 新建自己的命令

在 /etc/bash.bashrc文件中加上

alias 自定义命令=’具体语句’

再source 该文件生效

* + 1. U盘

dmesg | grep usb

dmesg -c 清除。

删除 /var/log文件夹下面的message\*\*.log的文件

* + 1. Hostname

主要提供ip地址的一个别名，通过别名就可以访问。在/etc/hosts文件中定义。

Echo $HOSTNAME 查看当前主机名，

Hostname xxx可以临时修改主机名；

Hostname –i可以查看当前主机名对应的ip地址；

重启后hostname仍然生效，需要修改/proc/sysconfig/network文件，修改hostname字段，系统启动会读取该字段作为主机名。

* + 1. 关闭防火墙iptables

Service iptables stop

* + 1. 搜索

Locate 和whereis 命令在已经建立的检索中搜索，更新检索使用updatedb命令。

全盘搜索使用find命令，可以根据文件的上次使用时间等条件进行搜索。

* + 1. 查看硬盘使用率

iotop –oP -d 查询间隔

1. Eclipse

取消validator

右键工程，属性，builder，

preference，validation

无菜单栏

通过sh eclipse.sh脚本启动

快捷键

Ctrl+M 最大、最小化当前编辑窗口，编码状态与调试状态切换；

Ctrl + T 搜索接口的实现；

Ctrl+K 向下搜索，+shift 向上

ctrl+j 增量查找

Ctrl+shift+R 快速根据文件名搜索文件，包括jsp和js文件都可以；

Ctrl+shift+T 根据java名称快速搜索；

Ctrl+E 在当前打开的编辑器中进行查找；

Alt+shift+w 在项目管理文件中定位当前文件；

CTRL+f6 切换编辑器 ctrl+f7 切换视图 ctr+f8 切换perspective

Ctrl + shift +p 定位到对应符号，如括号匹配；

1. Arcgis server

安装时，hostname要统一改成arcgisserver 要不就都不要改，下次启动时，hostname不一致可能启动失败。

问题：Unknown Linux distribution and/or version detected, skipping RPM package prerequisite check.

解决：需要修改操作系统版本号gedit /etc/redhat-release 输入 redhatenterpriseserver release 6.5(final)

生成.sd文件，在arcgis server 浏览器中加载服务。

制作数据

1. 使用global mapper将数据转为shapefile格式，文件-输出-矢量格式-shapefile即可。
2. 使用arcmap添加该shp文件，打开图层文件属性，label为注记、symbology设置绘制的颜色、display中scale symbol when a …. 设置绘制的粗细不随比例尺变化。
3. Python
   1. 调试

执行python -m pdb test.py 代码开头加入import pdb

pdb的常用命令说明：

　　l #查看运行到哪行代码

　　n #单步运行，跳过函数

　　s #单步运行，可进入函数

　　p 变量 #查看变量值

　　b 行号 #断点设置到第几行

　　b #显示所有断点列表

　　cl 断点号 #删除某个断点

　　cl #删除所有断点

　　c #跳到下一个断点

　　r #return当前函数

　　exit #退出

使用l命令查看代码行数，b命令临时增加断点，p命令查看变量值

* 1. Python的帮助

Pydoc –g 调出搜索浏览器，输入关键词，在浏览器中查看

1. Protobuf
   1. 格式定义

package tutorial;命名空间

message Person

{

required string name = 1;必填字段

required int32 id = 2;

optional string email = 3;选填字段

enum PhoneType

{

MOBILE = 0;

HOME = 1;

WORK = 2;

}

message PhoneNumber 每一个message可以使用已有message作为成员

{

required string number = 1;

optional PhoneType type = 2 [default = HOME];

}

repeated PhoneNumber phone = 4;重复字段，表示数组

}

message AddressBook {

repeated Person person = 1;

}

每个字段都有唯一的一个数字标识符。这些标识符是用来在消息的二进制格式中识别各个字段的，一旦开始使用就不能够再改变。注：[1,15]之内的标识号在编码的时候会占用一个字节。[16,2047]之内的标识号则占用2个字节。

* 1. 操作函数

序列化为二进制，以string类型来存储

string output;

person.SerializeToString(&output);

反向序列化，从string中解析

person\_input.ParseFromString(output);

* 1. 其他

String中以UTF-8的编码进行存储，中文开发环境下的GBK编码需要改转为UTF-8编码进行序列化。

1. Snmp 协议
   1. 扫描命令

snmpwalk -v 2c -c Public 192.168.245.241 .1 > /home/AR2200.txt

翻译命令

snmptranslate -On IP-MIB::ipvshi6ScopeZoneIndexA.5

MIB文件、my文件都是文本文件，需要使用snmp compiler编译为.smidb格式，compiler里面右侧工具栏有个编译按钮。生成的保存到MIB Browser\MIB\SMIDB文件夹下面，在browser里面导入即可。

模拟软件vxsnmpsimulator

C:\Program Files (x86)\vxsnmpsimulator 用户名admin 密码 pass