1. Springboot
   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. 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::ipv6ScopeZoneIndexA.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

1. 有道词典
   * 1. 把日期改为2017年11月1日，即可激活会员词典功能
2. 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. 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。

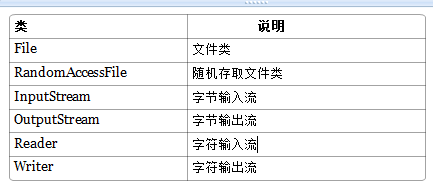
* 1. 匿名内部类和lambda表达式

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。

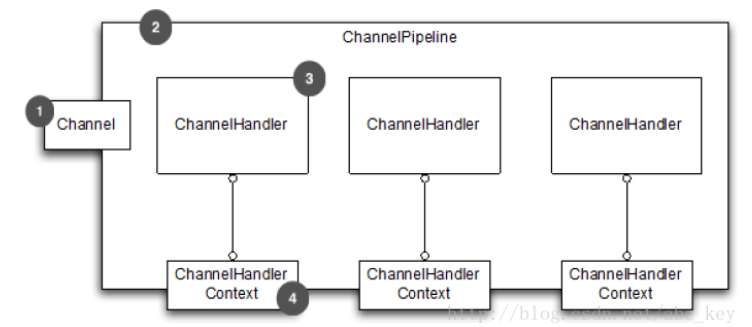
Local，io.netty.channel.local，用来在虚拟机之间本地通信。

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。

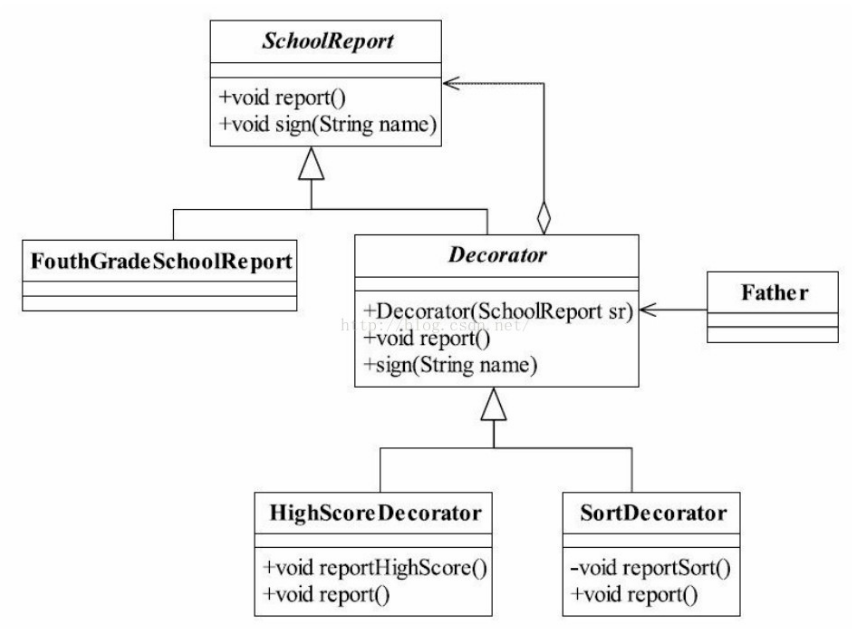
* 1. 代码优化
     1. 尽量使用类、方法的final修饰符，则不会继承，可提高50%效率。
  2. 设计模式
     1. 装饰者与io类

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

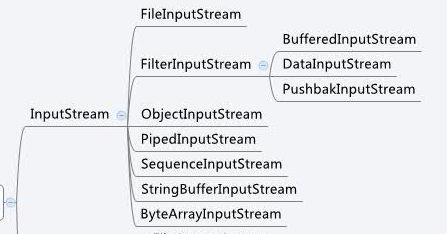
A、需要扩展一个类的功能，或给一个类增加附加功能。

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。

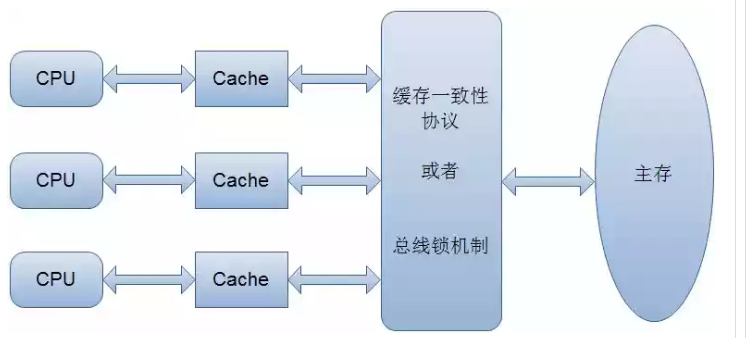
* + 1. Instanceof

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;

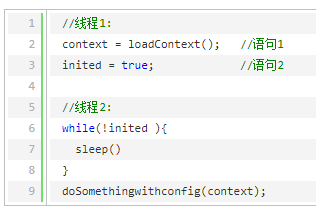
i = 10;

//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。

* 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。

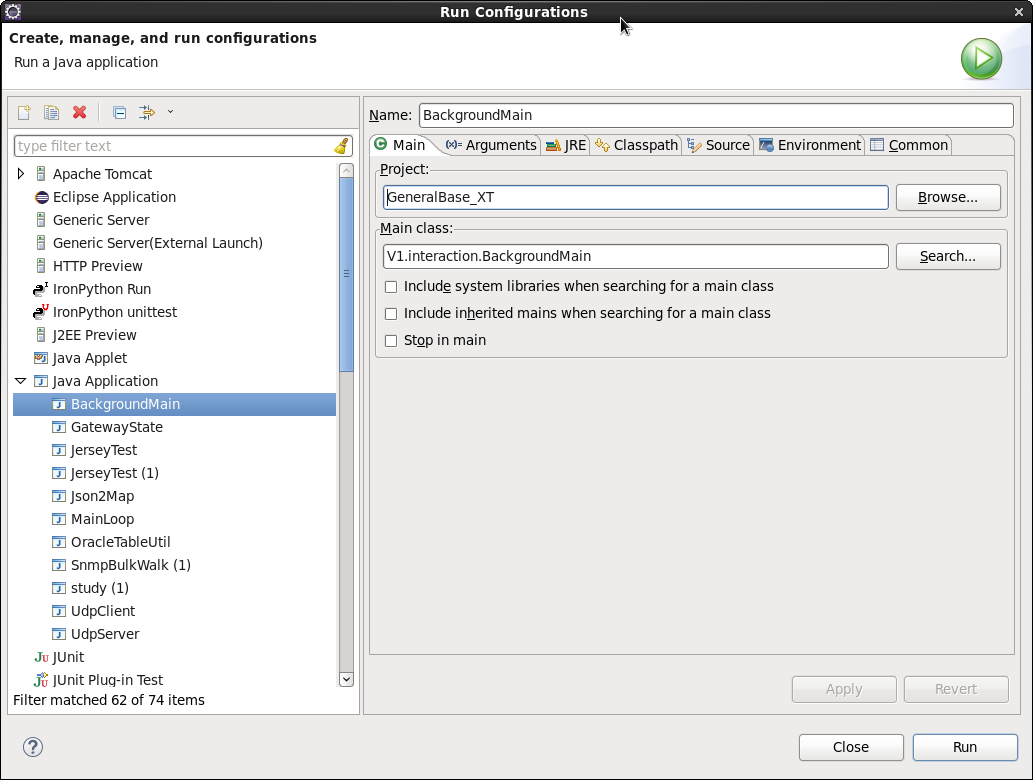
语句块的优势就是把各种构造函数里面重复的赋值放到一起去。

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

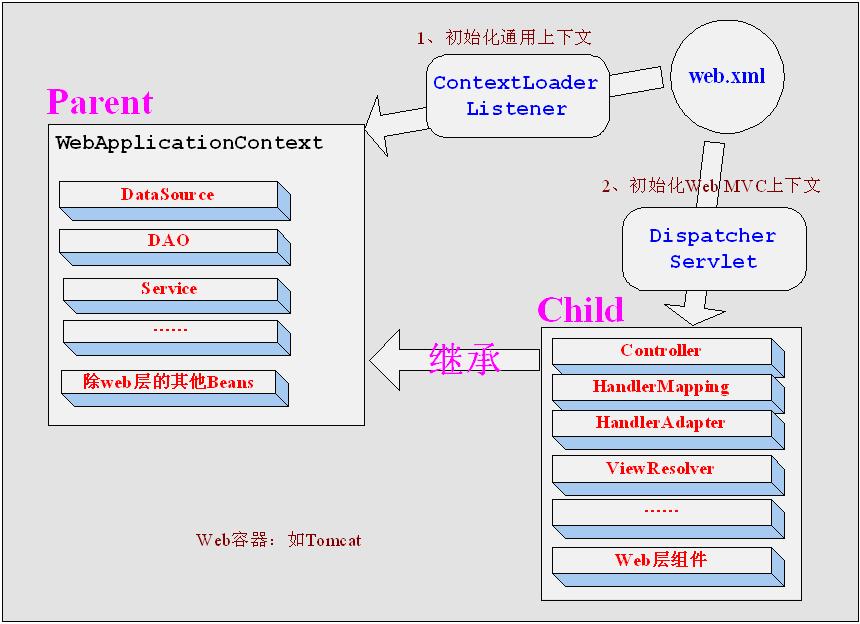
Servlet是个特殊的java类，继承于HttpServlet。

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

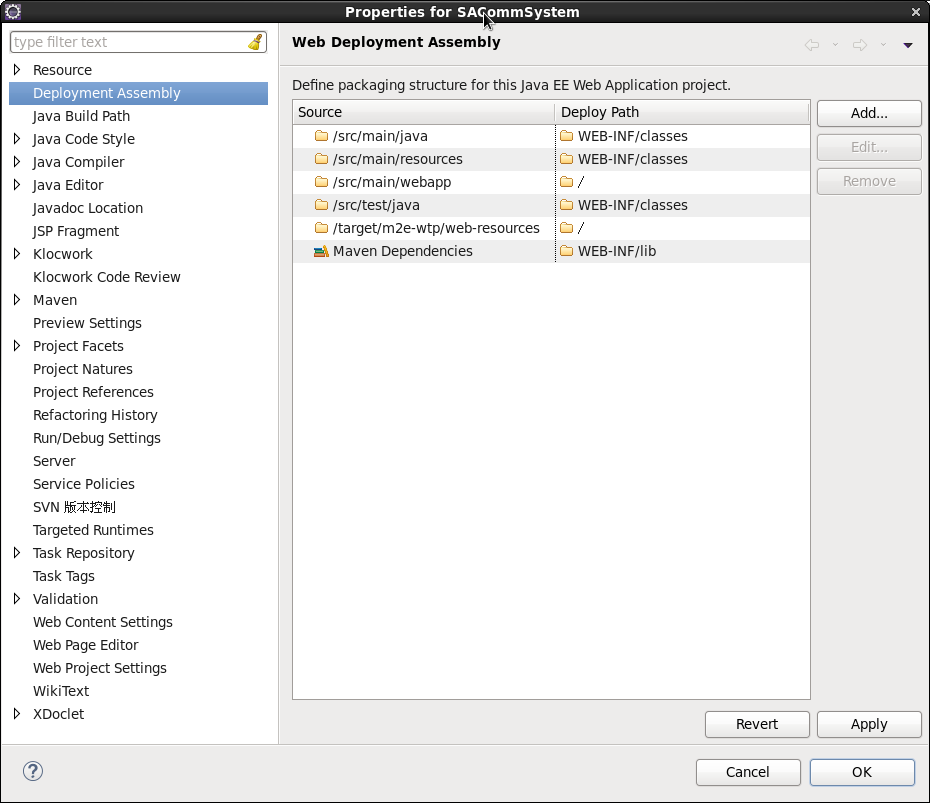
命令行启动tomcat：

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. Elasticsearch
   1. 笔记
      1. 客户端

节点客户端(node client):

节点客户端以无数据节点(none data node)身份加入集群,换言之,它自己不存储任何数据,但是它知道数据在集群中的具体位置,并且能够直接转发请求到对应的节点上。

传输客户端(Transport client):

这个更轻量的传输客户端能够发送请求到远程集群。它自己不加入集群,只是简单转发请求给集群中的节点。

* + 1. 分布式原理

一个节点(node)就是一个Elasticsearch实例,而一个集群(cluster)由一个或多个节点组成,它们具有相同的 cluster.name ,它们协同工作,分享数据和负载。

索引只是一个用来指向一个或多个分片(shards)的“逻辑命名空间(logical namespace)”.

一个分片(shard)是一个最小级别“工作单元(worker unit)”,它只是保存了索引中所有数据的一部分。一个文档使用一个分片存储，不会分为两个分片存储。

当索引创建完成的时候,主分片（primary shard）的数量就固定了,但是复制分片（replica shard）的数量可以随时调整。

* + 1. 数据

在Elasticsearch中,文档(document)这个术语有着特殊含义。它特指最顶层结构或者根对象(root object)序列化成的JSON数据(以唯一ID标识并存储于Elasticsearch中)。

一个文档不只有数据。它还包含了元数据(metadata)——关于文档的信息。三个必须的元数据节点是: 索引(index) 文档存储的地方、\_type 文档代表的对象的类、文档的唯一标识\_id，Elasticsearch中每个文档都有版本号,每当文档变化(包括删除)都会使 \_version 增加。元数据貌似都是以下划线开头。

\_source表示文档的原始数据，与输入的数据保持一致。

增加数据：

PUT /{index}/{type}/{id}

{

"field": "value",

...

}

自动生成id

POST /itest/ttest/

{

"name": "Leo Tong",

"school": "The 28th ",

"age": 51,

"time":null

}

检索数据

只检索部分字段

GET /itest/ttest/00011?\_source=age,school

只得到\_source字段而不要其他的元数据

GET /website/blog/123/\_source

更新数据

文档是不可变的——它们不能被更改,只能被替换，

多线程的安全性。同时修改一个数据，使用version关键字。在version为1时才修改成功，如果其他线程修改了该数据，version不为1，则需要重新读取version号，并进行修改。

PUT /website/blog/1?version=1

{

}

update命令内部用新的文档代替旧文档，利用上述原理保证更新操作是原子的。

批量操作

\_mget docs

有一个最佳的 bulk 请求大小，开始的数量可以在1000~5000个文档之间，一个好的批次最好保持在5-15MB大小间

* + 1. 搜索

语法：

逗号分隔在多个索引中检索，支持名称以\*作为通配符。

/g\*,u\*/\_search

结果分页，从第10个文档开始，显示5条文档，内部其实前10个文档也要检索并排序，所以检索靠后的文档该方式效率低。

GET /\_search?size=5&from=10

如果没指定查询index和type，则查询所有。

GET /\_search?q=2014

结果解释：

\_score字段,这是相关性得分(relevance score),它衡量了文档与查询的匹配程度。

took 告诉我们整个搜索请求花费的毫秒数

* + 1. 映射与分析

每个类型拥有自己的映射(mapping)或者模式定义(schema definition)。一个映射定义了字段类型,每个字段的数据类型,以及字段被Elasticsearch处理的方式。映射还用于设置关联到类型上的元数据。

查询指定字段，则使用确切值，不拆分查询字符串。

查询全文时，拆分查询字段，比如日期"2014-09-15"拆为"2014" 、"09"和 "15"。

查看分析器analyzer的分析结果，下面使用标准分析器。

GET /\_analyze?analyzer=standard&text=Text to analyze

index 参数控制字符串以何种方式被索引。它包含以下三个值当中的一个:

analyzed，首先分析这个字符串,然后索引。换言之,以全文形式索引此字段。

not\_analyzed ，索引这个字段,使之可以被搜索,但是索引内容和指定值一样。不分析此字段。

no ，不索引这个字段。这个字段不能为搜索到。

* + 1. 分布式文档存储

文档映射到哪一个shard。routing 值是一个任意字符串,它默认是\_id但也可以自定义。

shard = hash(routing) % number\_of\_primary\_shards。

这也解释了为什么主分片的数量只能在创建索引时定义且不能修改:如果主分片的数量在未来改变了,所有先前的路由值就失效了,文档也就永远找不到了

* + 1. 常见命令

集群健康状况

GET /\_cluster/health

索引情况

curl -XGET http://192.168.245.3:9200/\_cat/indices?v

高亮

"highlight": {

"fields": {

"school": {}

}

}

查询类型各字段的映射，即理解为整形还是字符串还是日期还是bool

GET /itest/\_mapping/ttest

PUT /megacorp/employee/1

{

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

"about" : "I love to go rock climbing",

"interests": [ "sports", "music" ]

}

1. MySql
   1. 导入导出

导入：mysql -uroot -psystem ODC\_Work < HQ-ODC\_Work.sql

服务器导出：mysqldump -udump\_user -pdump -h192.168.1.101 ODC\_Work > HQ-ODC\_Work.sql

本机导出：mysqldump -uroot -psystem ODC\_Work > Base-ODC\_Work.sql

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. HTML
   1. role属性

<form class="form form-horizontal" role="form">

用于增强语义，告诉浏览器这是form对象，即使标签为div。

* 1. tiles

web-inf下面的tiles.xml 定义了模板，

<definition name=*"template"* template=*"/pages/templates/template.jsp"*>模板名称

<put-attribute name=*"title"* value=*"CommSystem"*/>

定义模板的组成部分。

*template.jsp中*

*<*title><tiles:insertAttribute name=*"title"*/></title>

*定义了title属性对应的界面脚本，其他引用template.jsp的jsp文件中，*

*<*tiles:putAttribute name=*"title"*>Station Physical Topo</tiles:putAttribute>就可以将*template.jsp中title属性对应的部分引入进来。*

*模板相当于基类，其他引用template.jsp的jsp文件相当于继承类，修改基类，则继承类全变了。*

*另外一种情况，普通jsp通过jsp:include引入template.jsp文件相当与组合模式，如果想去除template.jsp，则需要修改所有普通jsp，相比，使用tiles方式，即继承的方式，把成员变量清空，即tiles.xml中的属性去除就去除来template.jsp的部分效果。*

。ref：分分钟知道tiles框架是干嘛用的

* 1. <a href=”#”>

链接本页面，无意义，因为a标签必须要有href属性

1. CSS
   1. 样式的优先级
2. 样式后面！important，覆盖所有样式；
3. 写在元素内的style属性；
4. Id选择器，权值1000；
5. Class选择器，权值100；
6. 标签选择器，权值10；
7. 通配符选择器，权值1；

So：同一个级别，后定义的覆盖前一个定义的；

组合选择器，根据权值进行计算总和。

* 1. 选择器

后代选择器：div li ，表示选择div 所有后代li元素。范围更广

子选择器：div>li ,表示div的一级子元素li。

:表示伪类，用于筛选被选择对象某些状态

* 1. position

position 属性规定元素的定位类型，absolute生成绝对定位的元素，相对于 static 定位以外的第一个父元素进行定位。元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。

* 1. 定位

div、h1 或 p 元素常常被称为块级元素。这意味着这些元素显示为*一块内容*，即“块框”，会独占一行。与之相反，span 和 strong 等元素称为“行内元素”，这是因为它们的内容显示在行中，即“行内框”。

CSS 有三种基本的定位机制：普通流、float和position，除非专门指定，否则所有框都在普通流中定位。

普通流：按照顺序，块状元素从上到下，行内元素从左到右依次排列。

* 1. 盒子模型

CSS 中，width 和 height 指的是内容区域的宽度和高度。

可以为元素的内边距（padding）设置百分数值。百分数值是相对于其父元素的 width 计算的,上下内边距与左右内边距一致；即上下内边距的百分数会相对于父元素宽度设置，而不是相对于高度。

如果您希望边框出现，就必须声明一个边框样式。

margin 的默认值是 0，margin设置为百分数的依据同内边距。

外边距合并指的是，当两个垂直外边距（上下边距）相遇时，它们将形成一个外边距。合并后的外边距的高度等于两个发生合并的外边距的高度中的较大者。

1. JS
   1. $.Ajax()

$.ajaxSetup({}) 可以设置ajax的默认初始值。

$.ajax({}) 选项

Datatype :返回的数据格式，如json、text等

error:function(xhr, message, error) message是jquery的状态码，值为error时表示http错误，具体从xhr的返回状态去判断。

Complete： ajax完成之后就，调用玩success或者error回调之后，会调用该函数。

Anysc：默认为true，即异步，想变为同步，等待返回再结束ajax，则改为true。

报bad request 400 错误

Js中的$.ajax的data的参数类型，与controler中的java参数不一致（data的变量为undefined也会造成类型不一致），或者变量名不一致

* 1. 复制一个对象

使用target = $.extend({},src); 只复制第一层；

target = $.extend(true,{},src);深复制；

jquery复制dom树。

Var element = $(“selector”).clone(); 选中的是所有匹配js变量，包括符合selector的dom对象及上下文，在必须接收dom对象作为参数的场合，需要element[0]来取对应的dom对象。如果一直使用jquery函数则直接对js变量进行操作，当然需要$(element);

* 1. 页面操作

刷新：history.go(0) location.reload() 页面前进history.go()

* 1. 对象属性的访问

成员函数访问自身属性

**var** test ={

name1:"1",

fun:**function**(){

**this**.name1 = "2";//需要加上this关键字

}

}

test.fun();

对象访问自身属性

对象属性可以通过get，set函数访问：

Obj.get(“属性名”)；

遍历所有属性

For(var I in obj){

Obj[i]；i是属性名称

}

判断属性是否在对象中

if (name in layerSet) {

return layerSet[name];

}

综上所述，对象可以当一个map使用

判断是否在数组中，用jquery的$.inArray(id,ary) 如果在，则返回索引，不在返回-1；

注意：id需要和ary的元素类型一致，如果id为Number类型，ary里面为string类型，则无法找到，处理方法：1、自己遍历ary，利用==号可以比较数字和字符串的特点，2、转换其中一个的数据类型。

* 1. $(function(){}

是 $(document).ready(function(){})的缩写，就是文档准备好之后再执行；openlayer3的地图创建必须在该函数中，否则文档未创建好，对文档操作会出现异常行为。

* 1. 函数参数

传值还是传参数：数字、字符串、布尔类型是传值；对象、数组、字符串对象是传地址。

字符串对象即通过：str = new string（“abc”）；创建的对象，typeof（str）是object类型。

* 1. 函数对象、普通对象 原型对象(prototype) 原型链

通过内置Function()函数创建的为函数对象，表现在两种创建方式：

fun(){}

var f = function(){}

两种，它们内部均调用Function函数创建。

其他方式为普通对象。

函数对象才具备原型对象(prototype)

函数对象有个成员：prototype和\_\_proto\_\_;

prototype主要用于继承使用，如：

fun.prototype.member1 = “1”;

任何对象都有\_\_proto\_\_,new返回的实例的\_\_proto\_\_指向函数对象的 prototype。函数对象的 prototype中有成员\_\_proto\_\_，指向更上一级的prototype，从而形成原型链。函数对象的 prototype中还有成员constructor指向函数对象本身

则fun的实例对象(obj = new fun();此处fun()也称为构造器函数)，此时obj.\_\_proto\_\_即为fun.prototype，则obj可以拥有member1成员。 prototype中的属性只是给实例对象（或者说是子类使用的一组属性），fun本身无法调用(fun.member1不存在)；

* 1. new关键字

例子：

function Animal(name){this.name = name；}

Animal.color = “black”;

Animal.prototype.say = function(){consol.log(this.name);}

cat = new Animal(“cat”);

其中new发生来什么？伪代码如下：

new Animal(“cat”) = {

var obj={};

obj.\_\_proto\_\_ = Animal.prototype;//如果return时返回obj，则cat成

//为Animal的实例对象（即子类）

var result = Animal.call(obj,”cat”);//obj拥有来name属性

return result === “object”? Result:obj;//正常构造器函数均返回obj

}

实例对象cat的原型链：cat.\_\_proto\_\_ = Animal. Prototype

Animal. Prototype.\_\_proto\_\_ = Object. Prototype

Object. Prototype.\_\_proto\_\_ =null

函数对象Animal的原型链：Animal.\_\_proto\_\_ = Function. Prototype

Function. Prototype.\_\_proto\_\_= Object. Prototype

普通对象的原型链：var tmp = {}; var tmp\_obj = new Object();

两者一样， tmp.\_\_proto\_\_ = Object. Prototype

Object. Prototype.\_\_proto\_\_ =null

ref：js核心系列：理解new的运行机制

* 1. 多重继承

多个函数对象可以通过原型链实现多重继承，继承成员函数通过goog.inherits(child,parent),继承父类构造函数中定义的成员通过goog.base()，

ref:google closure--继承模块一:goog.inherits();

自己的实现

var baseClass = function(id){

this.id = id;

}

baseClass.prototype.setid = function(id){this.id = id;}

var derivedClass = function(id,name){

baseClass.call(this,"derived");//让基类成员变为派生类成员

this.name = name;

}

derivedClass.prototype.setName = function(name){this.name= name;}

derivedClass.prototype.\_\_proto\_\_ = baseClass.prototype;

var baseObj = new baseClass("baseObj");

var derivedObj = new derivedClass("derivedObj","derived");

baseObj.setid("success");

derivedObj.setid("hh");

* 1. instanceof

判断规则，就算左值是不是右值的实例对象（子类）

如：cat instanceof Animal;

判断如下：

var L = cat.\_\_proto\_\_;

var R = Animal.prototype;

if(L === R)

return true;

用处：判断一个对象是不是数组

**var** a = [];

console.log(a **instanceof** Array); // true

* 1. this

函数中，this总是指向当前函数的所有者对象，只有在运行时才知道其具体指向。

fun(); //此处调用者是window对象。

fun.call(obj); //此处把对象obj放到fun函数上运行，所以所有者为obj。

* 1. 作用域

if while for 不能形成作用域，只有函数才有作用域，即if语句块内变量可以被if外面访问。

* 1. 遍历

$(selecter).each(func(index,element){}) 在遍历页面文档时使用较多。

$.each(data,func(index,element){}) 在遍历数据时使用较多，element参数可以省略，函数体中用this替代。

如果break，使用return false；

如果continue，使用return true；

* 1. 对象

对象由花括号分隔。在括号内部，对象的属性以名称和值对的形式 (name : value) 来定义。属性由逗号分隔。如：

var person={firstname:"Bill", lastname:"Gates", id:5566};

对象属性有两种寻址方式：

name=person.lastname;

name=person["lastname"];

* 1. 数据类型

对变量或值调用 typeof 运算符将返回下列值之一：

undefined - 如果变量是 Undefined 类型的 ，声明来对象，但是未赋值

boolean - 如果变量是 Boolean 类型的

number - 如果变量是 Number 类型的

string - 如果变量是 String 类型的

object - 如果变量是一种引用类型或 Null 类型的

类型转换

Boolean 、Number、String可以进行强制类型转换

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. 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. 数据库
   1. SQL优化

充分利用索引类

* + 1. 少用in

ORACLE试图将其转换成多个表的连接，如果转换不成功则先执行IN里面的子查询，再查询外层的表记录，如果转换成功则直接采用多个表的连接方式查询。推荐方案：在业务密集的SQL当中尽量不采用IN操作符，用EXISTS 方案代替。

* + 1. 不用IS NULL 或IS NOT NULL操作（判断字段是否为空）

只要这些列中有一列含有null，该列就会从索引中排除。也就是说如果某列存在空值，即使对该列建索引也不会提高性能。数据库中空值用0或者‘ ’填入，从而：a is not null 改为 a>0 或a>’’等

* + 1. 用>=代替>

执行A>2与A>=3的效果有区别，因为A>2时ORACLE会先找出为2的记录索引再进行比较，而A>=3时ORACLE则直接找到=3的记录索引。

* + 1. Like ‘abc%’ abc前面不要带匹配符%，因为第一个字符为匹配符，从而无法使用索引。
    2. UNION all 代替union

UNION在进行表链接后会筛选掉重复的记录，所以在表链接后会对所产生的结果集进行排序运算，删除重复的记录再返回结果。UNION ALL操作只是简单的将两个结果合并后就返回。

* + 1. 不用not，<> 会全表扫描，放弃索引

其他

* + 1. Where条件限制严格的往后放，会先被解析。可以过滤掉最大数量记录
    2. SELECT子句中避免使用 ‘ \* ‘

ORACLE在解析的过程中, 会将'\*' 依次转换成所有的列名, 这个工作是通过查询数据字典完成的, 这意味着将耗费更多的时间。

分析sql语句耗费情况，v$sqlarea视图

SELECT EXECUTIONS , DISK\_READS, BUFFER\_GETS,

ROUND((BUFFER\_GETS-DISK\_READS)/BUFFER\_GETS,2) Hit\_radio,

ROUND(DISK\_READS/EXECUTIONS,2) Reads\_per\_run,ELAPSED\_TIME,

SQL\_TEXT

FROM V$SQLAREA

WHERE EXECUTIONS>0

AND BUFFER\_GETS > 0

and Parsing\_Schema\_Name='COMM\_USER' and module = 'JDBC Thin Client'

ORDER BY 6 DESC;

Ref:数据库性能优化之SQL语句优化 - 全栈开发者.mht

* 1. Join

select count(\*) from a ,b where a.id = b.sid;

红色部分等价于select count(\*) from a cross join b，产生一个笛卡尔积，a的数据条数乘以b的数据条数。形成一个临时表，再通过where条件过滤，效率比inner join低很多。Join 的on条件是在生成临时表时就起作用，所以效率高。在大数据量时能体现区别。

* 1. 分区表

创建表空间

create tablespace partition1 datafile 'C:\app\Administrator\oradata\orcl\p1.dbf' size 20m

create tablespace partition2 datafile 'C:\app\Administrator\oradata\orcl\p2.dbf' size 20m

创建表

CREATE TABLE PROBLEM\_TICKETS

(

DATE\_ENTERED DATE NOT NULL PRIMARY KEY,

STATUS VARCHAR2(20)

)

PARTITION BY LIST (STATUS)

(

PARTITION p\_plane VALUES ('plane') TABLESPACE partition1,

PARTITION p\_ship VALUES ('ship') TABLESPACE partition2

)

创建本地索引

CREATE INDEX index\_problem\_ticket ON PROBLEM\_TICKETS(DATE\_ENTERED) LOCAL;

本地索引就是每个分区一个索引，索引存放在相应表空间中。全局索引就是一个统一索引，分区改变后索引会失效。

* 1. Oracle日志分类

Alert log file(警报日志)

Trace file(跟踪日志,用户和进程信息)

undo日志（事物执行过程中的日志,用于回滚）

redo日志（重做日志恢复数据库使用,用于数据库崩溃,已经commit数据在内存缓存中,还没有同步到磁盘上,所以redo日志缓存比数据修改缓存先写入到磁盘上 Ref:理解数据库中的undo日志、redo日志、检查点）

* + 1. redo日志分为

online redo log 在线重做日志（联机重做日志）,sql语句存入本地磁盘,用于数据内存的数据未同步到硬盘上,但是机器崩溃了,重启机器时,根据本地磁盘的redo log恢复到硬盘中。

Archive redo log 归档重做日志,简称归档日志,若数据库处于归档模式下（咱们用的数据库大部分是非归档模式）,在线重做日志满了自动写入到归档日志,以本地磁盘的形式,归档模式用于整个数据库恢复

Ref:Oracle-归档日志详解(运行模式、分类)

* 1. 获取tnsnames.ora位置

如果设置环境变量TNS\_ADMIN环境变量,则所有Oracle版本均以这个为准,适用于安装多个oracle客户端的机器,通过统一一个tnsnames.ora文件连接服务器。

注册表位置：搜索ORACLE\_HOME

* 1. 表空间
     1. 表空间已用容量

select tablespace\_name, count(\*) as extends, sum(bytes) / 1024 / 1024 as MB,sum(blocks) as blocks from dba\_data\_files group by tablespace\_name;

* + 1. 表空间剩余容量

select tablespace\_name ,count(\*) as extends,round(sum(bytes)/1024/2014,2) as MB,

sum(blocks) as blocks from dba\_free\_space group by tablespace\_name

* + 1. 表空间各种属性

select \* from dba\_data\_files

* + 1. 给表空间增加数据文件

Alter tablespace dblzhf\_ts add datafile 'D:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\DBLZHF\_1.DBF' size 100M autoextend on next 100M maxsize unlimited

* + 1. 创建表空间

create tablespace dblzhf\_ts logging datafile 'D:\oracle\product\10.2.0\oradata\orcl\dblzhf\_ts.dbf'

size 50m autoextend on next 50m extent management local;

* 1. 检索数据技巧
     1. 查询自己想要的数据库表

select \* from user\_tab\_comments where comments like '%情报%'

可以根据表的注释找，同时也可以找表名

* + 1. 检索海量数据中某一条

select \* from (select rownum rn, jssj from (select msgid, rownum, jssj from lzhf order by jssj desc) t) a where a.rn >= 100000 and a.rn < 110000

* + 1. 按照时间检索

select \* from lzhf where jssj between to\_date('2015-10-21 17:04:32', 'yyyy-mm-dd hh24:mi:ss')

and to\_date('2015-10-21 17:05:34', 'yyyy-mm-dd hh24:mi:ss')

* 1. 索引
     1. 创建索引

create index jssj\_index on lzhf(jssj)

* 1. 约束
     1. 获取所有的约束和主键约束

select owner ,constraint\_name,table\_name,column\_name from user\_cons\_columns where table\_name = 'LZHF'

* + 1. 删除主键约束

alter table lzhf drop constraint PK\_LZHF

* + 1. 删除表前先删约束

cascade constraints意识是如果其他表的约束中规定以来该表，则先删除该约束，再继续删表，保证表能被删除成功。

* 1. 启用、禁用约束

alter table 表名称 enable constraint 限制名称

启动后,出错,是因为检查约束是否可行,有的表不符合约束

alter table 表名称 disable constraint 限制名称

* 1. 给Oracle增加内存使用量以提高效率

alter system set sga\_target = 1100M scope=both;

alter system set db\_cache\_size = 500M scope=both;

* 1. Oracle sequence的作用

用来在多用户环境下产生唯一整数的数据库对象,该整数可以用来作为整数类型的主键值,在多用户并发情况下有较好的性能;

如果用程序代码生成该主键值,则需要保证互斥,效率比较低。一般在需要数据库优化的地方会使用sequence。

* 1. Win7下面oracle的odbc的位置

C:\Windows\SysWOW64\odbcad32.exe

* 1. 权限和角色

有各种各样的权限如连接数据库,建表等,将常用权限组合为角色。

* 1. 数据库自动处理脚本bat脚本解释

sqlplus /nolog @ScriptAdd.exec

登录到sqlplus,执行ScriptAdd.exec中内容

* + 1. SPOOL cat\_all.log

Spool是Oracle中输出日志使用

* + 1. @setvars.sql

设定运行的一些环境变量,这里灵活性比较强

* + 1. SET VERIFY OFF

是否显示替代变量被替代后的语句

* + 1. DEFINE root\_path=C:\oracle\product\10.2.0\oradata\orcl

定义数据文件存放目录

* + 1. DEFINE pwd\_of\_sys=orcl

-- 定义 SYS 用户启用 SYSDBA 角色时的口令

* + 1. conn dbsjgx/dbsjgx@orcl

连接某个用户

* + 1. @addScript.sql

执行sql脚本

* + 1. Disconnect 断开连接
    2. Exit直接退出
    3. Pause暂停,不退出命令行
  1. 数据库导入导出exp imp
     1. 导出

exp userid=system/manager@orcl file=c:\DBQKZH.dmp log=c:\DBQKZH.log owner=DBQKZH

如果出现exp-00003 未找到段的存储定义,则最好到数据库本机上导出

* + 1. 导入

imp userid=system/manager@orcl file=C:\DBQKZH.dmp grants=n ignore=y full=y log=C:\DBQKZH\_in.log

* + 1. PLsql导出sql文件

在export tables界面选择完表名后,右键 save object selection,记录下这次选择的表名,下次直接load这个文件就可以了。

* 1. 过程包解释

DELIMITER $$ 让$$为分隔符

DROP PROCEDURE IF EXISTS `PKG\_CSXT$SEZHDXWZLOG`$$ 如果PKG\_CSXT$SEZHDXWZLOG存在，则删除该过程包

DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @DUPLEX\_KEY=1; 如果主键冲突，设置DUPLEX\_KEY变量为1。

Sql语句调用过程包 CALL PKG\_CSXT$SEZHDXWZLOG ('tt','11','22','0000-00-00 00:00:00')

* 1. Blob字段

获取所有包含blob字段的表

select \* from user\_lobs

* 1. 其他杂项
     1. 无system用户密码创建用户和表空间

使用sys用户连接数据库,可在不知道system的密码情况下创建表空间和用户

先Sqlplus /nolog

Conn sys/oracle as sysdba

即可以管理员登录。

Sqlplus /nolog只是登录sqlplus,并没有连接数据库

* + 1. 大小写,字段名,关键词做字段名

Oracle的双引号之间是区分大小写的,列名自动为大写。

1. **C++编程**
   1. 动态加载动态库失败

动态加载Qkzh\_plg.dll,但是Qkzh\_plg.dll依赖A.dll,而A不存在,则总是显示Qkzh\_plg.dll加载失败,用dependence软件就发现原因。

Key:loadlibrary

* 1. 单例

Singleton.h文件

Class Singleton

{

Public:

Static Singleton \* getIns(){return m\_instance;}

Private:

Singleton(){}

Singleton \* m\_instance;

}

Singleton.cpp文件

Singleton \* Singleton::m\_instance = null;

注释：

getIns()函数是静态函数,static并不是修饰返回值对象的,而是表明这是静态函数,所以该函数不能访问非静态成员变量;

getIns()必须是静态的,静态函数可以在类对象未初始化时调用,所以第一个掉getIns后才能初始化该对象实例。

* 1. C语言运行库MTD MDD

程序依赖外部库同时有静态库和动态库，要保证这两个库编译时；C/C++-》代码生成-》代码生成 ，该选项选择C语言运行库版本，要保证外部库编译时，选用的是一样的MDD或者MTD；如下例子会错误：

程序依赖静态库A，动态库B；

A编译时C运行库为MTD，则是静态C语言运行库；

B编译时C运行库为MDD，则是动态C语言运行库；

程序编译时，会同时加载A和B，两个C语言运行库就会同时加载，则会出现

msvcprtd.lib(MSVCP100D.dll) : error LNK2005: "bool \_\_cdecl std::uncaught\_exception(void)" (?uncaught\_exception@std@@YA\_NXZ) 已经在 libcpmtd.lib(uncaught.obj) 中定义

即msvcprtd.lib和libcpmtd.lib这两个c语言运行库冲突。

混用可能产生跨动态库传参数失效。A库调用B库中函数

Fun(list<unsigned char> para)

,A库使用Debug版C运行库,B库使用Release版C运行库, para参数在传递过程中丢失了,一定保证参数一致。

* 1. vector map 防止迭代器失效的问题

Vector:删除后,内存数据是顺序存储,被删元素后面的元素均向前移动,则原始后面迭代器均失效,所以erase返回下一个元素迭代器,使用该新迭代器

Itor = Vecotr.erase(itor)

Map:红黑树存储,删除节点,节点内存不变,只是更改节点指向的数据,所以后面迭代器仍然可用,则利用后++,传递的参数仍然是旧迭代器,这样

map.erase(itor++)

* 1. map基本使用

插入

map<int ,int> m;

m[1] = 1;直接插入

检索

不能使用m[1],如果没1这个key,则会插入一条

应该是m.find(1),返回对象迭代器

删除参见2.4

* 1. 各种链接失败原因
     1. 因编码方式造成的

库文件 1、使用多字节或者Unicode编译未统一 ,即使在项目常规中选择字符集为“未设置”,也需要在预处理器中 去掉 UNICODE宏（QT-addin生成的项目总是在预处理器中加入UNICODE宏,需要去除）,2、c/c++中语言 – 将wchar\_t设为内置类型 这两点与应用程序不一致,会造成无法链接库文件中的函数,应该将两者统一

* + 1. 某些库必须定义宏才能将宏定义为\_\_declspec(dllimport)

#if defined(Q\_WS\_WIN)

# if !defined(QT\_QTWINMIGRATE\_EXPORT) && !defined(QT\_QTWINMIGRATE\_IMPORT)

# define QT\_QTWINMIGRATE\_EXPORT

# elif defined(QT\_QTWINMIGRATE\_IMPORT)

# if defined(QT\_QTWINMIGRATE\_EXPORT)

# undef QT\_QTWINMIGRATE\_EXPORT

# endif

# define QT\_QTWINMIGRATE\_EXPORT \_\_declspec(dllimport)

# elif defined(QT\_QTWINMIGRATE\_EXPORT)

# undef QT\_QTWINMIGRATE\_EXPORT

# define QT\_QTWINMIGRATE\_EXPORT \_\_declspec(dllexport)

# endif

#else

# define QT\_QTWINMIGRATE\_EXPORT

#endif

以上这一段在MFC调用Qt的Qt\_WinMigrate库中,必须加QT\_QTWINMIGRATE\_IMPORT宏其他项目才能使用该库

* 1. c++ 编程规范
     1. 参数 输出在前,输入在后. 保证风格统一

输出参数必填,输入参数有默认值,放到后面可以不填

* + 1. 头文件排序:

本cpp头文件

c++系统文件

qt系统文件

其他库头文件

本项目头文件

* + 1. 头文件按照项目源代码结构安排,不要使用../../../ 这类

使用src/comprehensive/openData.h这种风格.

1. **VS开发环境**
   1. 重构-重命名带来的缺陷

重构-rename时,不会改变头文件开始的

#ifndef TARGETSELECT\_H

#define TARGETSELECT\_H

则,下次使用重复使用该类名,因为宏TARGETSELECT\_H被定义,则新的文件用于不会被定义,则编译器总是提示 找不到新的文件中的类。

* 1. VS编辑操作技巧
     1. 行编辑
     2. Alt + shift +T 交换光标所在行和下一行
     3. 代码模板

一段常用代码,选中后放入工具箱,下次点击即可直接使用

* + 1. Ctrl + i渐进式搜索

输入ctrl+I,按照状态栏提示进行搜索,再次按ctrl + i搜索下一个,非常好用的文档内字符匹配搜索工具

* + 1. 不同标签页直接的切换

按住,Ctrl + alt + 向下键,列出当前所有打开的cpp和h文件标签,输入文件名自动匹配,实现不用鼠标在标签页直接切换

Ctrl +tab键,也可实现在最近使用的几个标签页直接切换

* + 1. Alt + shift +S检索所有符号

exec –cmd .ab cd. 检索包含exec的符号,排除cmd字符串 以ab开头 以cd结尾, 后面的控制可不加 ,还可以勾选仅显示类名,检索出的结果上面可以直接右键菜单操作

* + 1. Alt + shift +f查看所有引用

有一个Clone Result,可以在新Tab页保存当前结果,查另一个对象的引用时,不会丢失该结果

* + 1. Alt + 上下 跳到下一个函数,跳到上一个函数
    2. Change Signature更改函数的名字,参数顺序等,会将调用该函数的地方同步修改。
    3. Encapsulate Field 给变量自动提供Get和Set函数

输入函数变量,右键 Encapsulate Field,自动提供Get和Set函数

* + 1. Alt + shift +] alt +shift +[

智能选择光标前后区域, 多次按]表示扩大,[表示减小,Alt + ] 一般选择一个函数

* + 1. Tools中,可以将类声明中函数自动排序
    2. 让vs重新解析，清空下面的路径即可

C:\Users\Administrator\Local Settings\Application Data\VisualAssist

* + 1. 成员变量输入,自动加入\_

成员变量输入时,按m后按shift键自动加入\_,从而免去输入shift+-的麻烦

* + 1. 查看引用时,显示读取的

在 va find references results结果中,D显示读取变量的,W显示写入变量的。

* 1. 调试时，应用代码更改

设置：c/c++常规 调试信息格式 用于“编辑并继续”的程序数据库 (/ZI)

* 1. 多进程编译

多进程同时编译选项，在库文件的 属性 - c++ - 命令行 加入 /MP

四核可以保证四个进程同时编译，提高一倍以上的速度。

* 1. Va snippet好用的快捷缩写

$end$光标位置

$MONTH$/$DAY$/$YEAR$ 日期

Class 关键词

class $Class\_name$

{

public:

$Class\_name$();

~$Class\_name$();

protected:

$end$

private:

}; 用户自己定义Class\_name,弹出界面让用户输入

* 1. 文件中相对路径的问题

Cpp中写相对路径的问题. Vs的exe程序需要在 “调试””工作目录”设定$(OutDir),这样程序中,qss配置中的资源的相对路径均相对该地址,保证双击程序也可以执行.

不建议将vcproj路径设定为基准路径,因为发布后,没有vcproj,用户在其他文件夹双击exe执行,则变为相对exe,路径自然错误

 Cpp中路径最好使用appDataDirAbsPath类似函数,获取exe路径,再取相对位置

保证双击exe执行也正常.

Cpp文件中, QFile styleFile("../../../Data//ComprehensiveStatus//Config//myStyle.qss");

字符串中路径相对于工程配置vcproj文件,在编译期,编译器根据工程当前路径,填补完整该路径.

干扰项：有时候配置文件编码与程序不一致如 配置文件编码 UTF-8,程序为GBK,则读入配置文件的中文为乱码,误以为文件路径不对。

1. **QT知识**
   1. QSetting

读注册表,使用

new QSettings("HKEY\_LOCAL\_MACHINE\\SOFTWARE ",QSettings::NativeFormat);

构造函数中使用的双反斜杠,取值时使用的单正斜杠

setting->value("KEY\_OraDb11g\_home1/ORACLE\_HOME")

读ini文件使用

QSettings settings("myapp.ini",QSettings::IniFormat);

* 1. Qt的界面属性设置
     1. Qt::Tool

永远在父窗口的顶层显示。用于包括一些工具按钮,通常边框会小一点。

* 1. Qobject的特性

metaObject() 返回元对象，其中存储了类名、各属性、信号等，特殊特性都依赖于该元对象

connectNotify() 有信号连接到该对象时可进行通知。

objectName() 每个对象都有自己的名称,默认为空

findChildren() 根据名称(objectName)寻找该对象的子对象

event() 事件处理，可以通过installEventFilter过滤事件

事件对象为QEvent，有一个标志位表示接受还是忽略该消息。

继承自QObject的类无复制构造函数，每一个对象的特性是唯一的

* + 1. 智能指针

QSharedPointer,共享指针,类似boost中的share\_ptr,但是不能用于有父子关系的界面qt类,因为父类释放时会调用子类的释放函数, QSharedPointer再次释放子类就出错了。

QScopedPointer出了局部范围自动删除指针,防止程序在一对大括号之间运行时突然奔溃,指针未释放的情况。

* 1. 关闭窗口

QWidget::setVisible()设置窗口显隐，show，hide，sethidden都是马甲，方便函数。

QWidget::close() 发送QCloseEvent事件，隐藏窗口，如果设置Qt::WA\_DeleteOnClose标志位则释放该对象，destroyed()信号会发送，用来将对象指针设为NULL

Widget::closeEvent ( QCloseEvent)，点击关闭按钮或者调用close槽会发送QCloseEvent事件并调用该函数

非模态对话框关闭时释放对象并置指针为空的方法：

w->setAttribute(Qt::WA\_DeleteOnClose);关闭时会删除对象，发送destroyed信号

connect(w,SIGNAL(destroyed()),this,SLOT(SetPointerNull()));槽函数中将w指针置为NULL

QDialog::done() 和close类似，隐藏窗口，结束消息循环

Dialog关闭时，除非是确定按钮，exec函数返回QDialog::Accepted，其他情况均返回Dialog::Rejected

* 1. QString的使用技巧
     1. Right;取右侧的子串
  2. Model/view delegates
     1. QitemEditorFactory中有item常见类型的delegate,不够可以使用registerEditor注册。

QitemEditorFactory根据model的setdata(QVariant)中, Qvariant实际传入的类型来进行判断该使用哪种编辑控件,所以时间Qdatetime类型会自动使用QdateTimeEdit控件。同时,对于int类型,可以自己注册registerEditor类型,对于用int表示的经纬度使用特殊处理,其他使用默认处理。

* + 1. QmodelIndex

View和delegate通过QmodelIndex在model中定位信息,但是获取每一个信息是临时变量,因为Model中的信息随时在变化,需要保存永久的model中的位置信息使用QpersistentModelIndex。

Tree每一个节点下面带一个表格坐标系,使用index(row, column, parent)表示。Parent即为Tree当前节点的QmodelIndex对象。

Views和delegates通过model index访问数据项,Model.model()返回model指针。获取model index时需要考虑model的三个属性：行,列,父项,顶层项的父项为QModelIndex()。每一个数据项包含多个角色role,

自己实现model,1、非层次结构的只要实现rowCount()和data()函数,2、层次结构的还需要实现index()和parent()函数

* + 1. QSqlTableModel::setFilter(qstring);条件语句中不要有where,不然总是执行错误。
  1. Qt的预编译头文件

C++的编译只针对cpp文件，每次将include的头文件展开插入到cpp文件中，所以可以将不变的include头文件作为预编译文件\*.pch。

设置方法：

1. 项目-预编译头- 依次设定：使用预编译文件，预编译文件名，如：myHeader.h
2. 右键myHeader.cpp 预编译头-创建预编译头，（因为只编译cpp文件，所以myHeader.cpp 只需要一句 include “myHeader.h”即可）
3. 在所有cpp的最前面加入 include “myHeader.h”（头文件中不需要，因为编译时展开cpp文件，最前面是myHeader.h的内容，保证头文件中需要的均已经定义）
4. Qt的moc.exe生成的moc\_XXXX.cpp文件因为自动生成,没有包含myHeader.h,可以在工程-属性-C/C++-高级-强制包含myHeader.h,这样moc\_XXXX.cpp代码在第一句就隐式的加载了myHeader.h头文件。（或者在xxx.h的moc.exe的命令行中加入"-fStdAfx.h",新建cpp文件时,勾选使用预编译头,就会在moc.exe的命令行中加入"-fStdAfx.h"）
   1. 鼠标按键一直按着的问题

鼠标漫游地图,一直按住左键.消息如下:

keypressEvent

keyreleaseEvent

keypressEvent

keyreleaseEvent

…

如此循环,但是我们不需要处理keyreleaseEvent,利用isAutorepeat()函数.消息顺序如下:

1 keypressEvent isAutorepeat = false; //此时还没到keyreleaseEvent

2 keypressEvent isAutorepeat = true;

3 keyreleaseEvent isAutorepeat = true;

4 若未松开,回到3 , 若松开, keyreleaseEvent isAutorepeat = false;

所以,只要在keyreleaseEvent 事件处理函数开头写:

if (me->isAutoRepeat())

{

return ;

}

即可保证返回中途不处理释放按键事件.

* 1. 字体绘制

QFont font;

font.setPixelSize(50);

QRect fontBoundingRect = QFontMetrics(font).boundingRect(tr("Qt"));

text.addText(-QPointF(fontBoundingRect.center()), font, tr("Qt"));

addText（相对于字体左下角的偏移量，字体，字符串）;

* 1. styleSheet,qss
     1. 选择符

|  |  |  |  |
| --- | --- | --- | --- |
| 选择符类型 | 例子 | 解释 | 常用场景 |
| \* |  | 通配符，匹配所有情况 |  |
| 类型选择符 | QPushButton | 匹配QPushButton这个类型及子类 |  |
| 类选择 | .QPushButton | 仅匹配QPushButton这个类型 |  |
| 属性选择符 | QPushButton[flat="false"] | 匹配有对象有属性为flat；例子：界面中强调的地方使用黄色，则给按钮，输入框等组件添加同一个属性nameEdit->setProperty("mandatoryField", true);即可 |  |
| ID选择器 | QPushButton#okButton | okButton就是object name设定的名称 | 对单个对象进行定制化 |
| 后代选择器 | QDialog QPushButton | 选择QDialog中的QPushButton包括QDialog子控件和孙子控件的按钮 | 不常用 |
| 子选择器 | QDialog > QPushButton | 前一个特例，仅QDialog的子控件按钮， | 常用 |

::符号，某个控件本身带子控件，（该类控件一般为复合控件），如下拉框的下拉按钮，通过::设置，可以与选择符合用。例子,QComboBox::drop-down { image: url(dropdown.png) }

:符号，状态选择符hover，checked，pressed

* + 1. 属性及属性类型

|  |  |  |  |
| --- | --- | --- | --- |
| 描述 | 值 | 解释 |  |
| border-style | solid:实线  dotted:点线  dashed：虚线 | 边框线型 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

属性类型

|  |  |  |  |
| --- | --- | --- | --- |
| 描述 | 值 | 解释 |  |
| Color | rgb(r, g, b)：红绿蓝  rgba(r, g, b, a)最后是透明度, | 对于顶层窗口,如QDialog,透明效果无效,设为全透明后,会有一层黑色底色,顶层窗口使用setWindowOpacity 函数设置后全部都透明了,非顶层窗口QFrame则有效 |
| qproperty-各种属性 | 例如qproperty-text:面板;将text属性设为“面板” | 通过qproperty-设置所有属性,很重要的功能,扩展了Qss的作用范围见注1 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

注1：所有的property都可以使用,但是

Q\_PROPERTY(QPoint pos READ pos WRITE move DESIGNABLE false STORED false)

表示不存储该字段,该字段信息设置了可能没有用,该字段值依赖其他字段来取。

* + 1. Box model

盒子模型，从内到外，padding 内边距， border 边框，Margin 外边距。默认为0

Padding 允许我们在内容里面挖出一块地方填充其他对象,比如qlabel

Padding-right:20px 则右侧文字部分空出20像素宽,我们可以在这里放置其他窗口部件.

默认background-image 是从border内部开始的，经过测试不包含border ，qpushbutton默认设置包含一点点宽度border的

Min-height ,max-height同时设置为同一值(见手册stylesheet的reference的width描述),才指定widget的高度,高度是指box model的contents部分，但是函数中height()返回的包含盒子模型的margin在内的所有高度.

* + 1. Border-image 使用

border-image:url(../../../res/TSKJ/zoomIn.png) 2 2 2 2;

border: 2px 2px 2px 2px transparent;

九宫格的边缘和border的宽度一致。

* + 1. 关于 setStyleSheet函数与继承性.

子窗口会继承父窗口的stylesheet 样式**语句,**是否作用取决于选择符的写法.以继承的label类myLabel包含qpushbutton为例.

myLabel->setStyleSheet(“color:red”);

这句等价于:

\*{ color:red } 则子button继承该语句,也变成红色,建议这样写:

myLabel->setStyleSheet(“Qlabel{color:red}”); 这样,不会对子窗口产生影响,

确定选择符后,子窗口不使用父窗口的样式.

* 1. 布局
     1. qgridlayout

Margin在四个方向默认为11个像素，spacing默认7像素

* + 1. 新建布局对象,无需指定父指针

QGridLayout \* layout = new QGridLayout;

在调用setlayout 或者addlayout函数时,会自动添加layout的父指针.

因为一个窗口只能有一个主layout,你在第一个layout创建时指定父指针为当前窗口,则后面主layout就无法成为当前窗口的layout.

* + 1. Layout与qwidget的父子关系

处于Layout中qwidget的父窗口不是layout,而是layout所处的qwidget,qwidget的父窗口只能是qwidget.

* 1. Qlabel的使用

可以使用在qlabel的字符串中填入html风格的字体颜色配置项。

* 1. Xml解析总结

<tag>this is string</tag>

改变this is string的值时，定位到tag之后为node后，需要再使用第一个孩子即：

Node.firstChild().setNodeValue(new string),取孩子，才是this is string的值，然后修改为参数里面的值。

解析常用函数：

elementsByTagName（str） 返回该节点下所有相应名称子节点，好用

firstChildElement（str）寻找该节点下第一个名为str的节点

namedItem和上面类似

qdomnode基类

qdomelement代表一个节点如<tag>this is string</tag>，底下可以包含更多节点

qdomtext代表this is string字符串，这也是一个节点。

* + 1. 将所有子节点组织为map里面

QMap<QString,QString> para\_value\_s;

for (QDomElement node = root.firstChildElement();!node.isNull();node=node.nextSiblingElement())

{

para\_value\_s.insert(node.tagName(),node.text());

}

可以编写一个对象,对应一级xml,包含子对象可以继续展开,或者为根节点,这样可以快速从该对象访问到需要的节点数据。如string = root.member(“first level”). member(“second level”).data();

* 1. 设置插件路径

QStringList l = QCoreApplication::libraryPaths();

QStringList library\_path\_list;

library\_path\_list.append(QCoreApplication::applicationDirPath()+"/sqldrivers");

library\_path\_list.append(QCoreApplication::applicationDirPath());

QCoreApplication::setLibraryPaths(library\_path\_list);

从而动态加载动态库时,可以从自己的路径加载;

另外发布Qt程序,参考qt.conf文件,但是上次测试该文件无效。

1. **基础知识**
   1. 日志库Log4cplus
      1. 配置文件编写

#日志根对象,呈现父子结构,子对象集成父对象特性,定义格式 对象名=[日志级别] #appender名称root这个根对象有两个个appender,为ALL\_MSGS,FATAL\_MSGS,对应不同#级别的日志

log4cplus.rootLogger= TRACE,ALL\_MSGS,FATAL\_MSGS

# filelogger是自定义的子日志对象,给特定库或者用户使用的,

log4cplus.logger.filelogger=TRACE,R

#表示不集成父logger对象的特性

log4cplus.additivity.filelogger=FALSE

#appender决定logger的信息写到哪里去,一个appender对应一个日志文件或控制台

log4cplus.appender.ALL\_MSGS=log4cplus::RollingFileAppender

log4cplus.appender.ALL\_MSGS.File=all\_msgs.log

#日志文件大小50MB,可以有5个这样的

log4cplus.appender.ALL\_MSGS.MaxFileSize=50MB

log4cplus.appender.ALL\_MSGS.MaxBackupIndex=5

log4cplus.appender.ALL\_MSGS.layout=log4cplus::PatternLayout

log4cplus.appender.ALL\_MSGS.layout.ConversionPattern=%D{%H:%M:%S %y/%m/%d} - %m %n

log4cplus.appender.FATAL\_MSGS=log4cplus::RollingFileAppender

log4cplus.appender.FATAL\_MSGS.File=fatal\_msgs.log

log4cplus.appender.FATAL\_MSGS.layout=log4cplus::PatternLayout

log4cplus.appender.FATAL\_MSGS.layout.ConversionPattern=%D{%H:%M:%S %y/%m/%d} - %m %n

#控制只有FATAL这个级别的日志才写入到fatal\_msgs.log日志文件

log4cplus.appender.FATAL\_MSGS.filters.1=log4cplus::spi::LogLevelRangeFilter

log4cplus.appender.FATAL\_MSGS.filters.1.LogLevelMin=FATAL

log4cplus.appender.FATAL\_MSGS.filters.1.LogLevelMax=FATAL

log4cplus.appender.FATAL\_MSGS.filters.1.AcceptOnMatch=true

log4cplus.appender.FATAL\_MSGS.filters.2=log4cplus::spi::DenyAllFilter

#自定义日志对象的appender,输出到R.log中

log4cplus.appender.R=log4cplus::RollingFileAppender

log4cplus.appender.R.File=R.log

#log4cplus.appender.R.MaxFileSize=5MB

#log4cplus.appender.R.MaxFileSize=500KB

#log4cplus.appender.R.MaxBackupIndex=5

#这是日志输出格式布局控制

log4cplus.appender.R.layout=log4cplus::PatternLayout

log4cplus.appender.R.layout.ConversionPattern=%D{%H:%M:%S %y/%m/%d} - %m %n

* + 1. UNICODE版本与多字节版本

中文操作系统使用vs开发环境,编译到exe中的字符均为GBK编码,所以必须使用非UNICODE版本的,否则写入文本对象会死掉,见：

if(!out.good()) {

getErrorHandler()->error( LOG4CPLUS\_TEXT("file is not open: ")

+ filename);

return;

}

但是考虑以后切换英文操作系统,后果未知,建议日志用英文编写

* + 1. 阅读笔记
    2. 未进main函数之前就生成了root对象,

class \_static\_log4cplus\_initializer {

public:

\_static\_log4cplus\_initializer() {

log4cplus::initializeLog4cplus();//初始化函数

}

* + 1. } static initializer;利用创建静态对象时创建
  1. QGIS使用
  2. QGIS修改坐标系,切割部分地图数据

打开shp文件;图层名a

右键另存为，

1、坐标系选择WGS84（只要打开过正确的坐标系，这里有记录，

或者右键正确图层，点击“将图层坐标系设为项目的坐标参考系”，就有正确参考系可以候选）；

2、选择范围，点击图层范围或者地图视图范围，保证“东南西北”是逻辑合理的范围内。

切割部分地图数据

1、切换到编辑状态

2、鼠标状态为 选择，可以选择按多边形状态选择；

3、选择区域后，右键图层另存为，注意勾选"save only selected feature"

* 1. Svn

清除代码中不受版本管理的内容,使用clean up-勾选delete unversioned files and folders.

合并代码,Merge – Merge a range of revisions ,选择需要从其他版本中合并过来的版本号,

合并后即使未提交,clean up也无法取消合并状态。然后就可以了,Ref：TortoiseSVN 中Branching和Merging实践。

* 1. Visio2013的中uml的使用

Refer: uml各类图 博客园 wangkangluo1

* 1. sublime 使用手册
     1. 跳转类命令

Ctrl + d 选中光标当前单词,再次Ctrl+d继续选中同名,可以多个同名字段同时编辑

F3查找下一个,alt + f3选中其出现的所有位置

Ctrl + p 快速定位文件 支持文件名 输入#则只在本文件检索并跳转

Ctrl +tab 最近的标签页之间跳转

Ctrl + r 本文件内快速定位符号(这里指一个类对象,一个语句块的标识符等)并跳转

Ctrl + shift + 2 分为2屏, Ctrl + shift + 1 分为1屏,ctrl + 数字跳到相应的屏幕

Shift + f11 切换无干扰全屏

Alt + -退回到上一次光标所在处

Alt + left /right 以单词为单位跳转

* + 1. 编辑类命令

编辑-行列-重新缩进 进行自动排版

选中后,再按right,光标在选中区域右侧,再按left,光标在选中区域左侧

向下移动当前行Ctrl+shift+down

删除当前行Ctrl+shift+k

Ctrl+v直接复制光标所在行

Ctrl+shift+/ 注释语句块 ctrl+/注释行

Ctrl+enter 相当于光标到行尾,再enter,加 shift在上一行

* + 1. 选择类命令

Ctrl+alt+up down 列选择

Ctrl+D 选择单词 多次按,选择多个单词

Ctrl + L 行选择 多次按,选择多行

Ctrl + shift + j 选择当前光标同一缩进的范围内容,最多选择同一缩进

Ctrl+shift + m 选择当前光标同一括号内容,最多选择同一个括号

Ctrl+shift + space 选择当前一个作用域的,多次可继续展开

* + 1. 查找类命令

ctrl+f3 选择当前单词作为查找对象,再次按f3查找下一个,其实在屏幕范围内,会高亮相同的字符,除非整篇文章查找需要。

* 1. 查看端口被哪个程序占用

查占用该端口的进程pid

命令： netstat –aon | findstr “端口号”

查进程: tasklist | findstr “pid号码”

* 1. 正则表达式

元字符

. 除换行符外任意字符

\w 字母,数字,汉字

\s 匹配任意的空白符，包括空格，制表符(Tab)，换行符

\d 匹配数字

\b 匹配单词的开始和结束,占位符,

^ 字符串的开始

$ 字符串的结束

限定符,

\* 重复0次到任意次

+ 重复一次或更多

? 重复0次或一次

{n} 重复n次

[] 匹配括号中任意几个。

* 1. 防止崩溃总结
  2. Qstringlist lonLat = posText.split("Lat");

m\_GeoLatShow->setText(lonLat[1]).

这里取lonLat[1],但是有可能分割不成功,不存在lonLat[1],所以需要判断lonLat分割数量.

* 1. 不用的对象指针给注释掉

某窗口类成员指针p,表征一个子窗口，后来不用了，但是paintevent中还在使用该指针，所以系统启动后就崩溃，查了好久，所以不用指针需要注释掉。

* 1. 代码中有中文字符,如何做到跨平台编译、运行、编译

Windows平台下编译时

Vs2010（包括vs2008）对于cpp文件代码的中文注释,如果是UTF-8带签名,则以UTF-8解析,如果是UTF-8不带签名,则以控制面板中系统语言解析（这时想编译过,系统语言需要选择【英语美国】即UTF-8字符集,如果系统为【中国】,则以GBK解析,则会出现奇怪的编译错误）,

Linux平台下,操作系统只认UTF-8,所以编译带签名和无签名UTF-8的文件均没有问题,GBK编码的完全无法编译。

运行时,生成的exe执行程序中的字符,在vs2010以下版本时,是以操作系统的语言存储的,（即使代码是UTF-8的,中文操作系统下,编译成的执行文件中”xx”也是以GBK存储,根据setCodecForCStrings的原理,需要设定setCodecForCStrings（QTextCodec::codecForName("GBK")）;

总结：有中文,代码文件编码为带签名UTF-8, setCodecForCStrings为GBK

附录：UTF-8带签名无签名区别,就是文件开头是否表示 该文件以UTF-8编码,多3个字节而已。UTF-8是国际标准,UTF-8带签名是微软力推,也是国际标准,linux也支持。

Ref：Qstring与中文问题。

* 1. 字符编码
     1. UNICODE  和\_UNICODE宏

操作系统API有两套,本地字符集的（多字节-每个字符用char表示,如果中文则多个char）,UNICODE字符集（UNICODE,每个字符用wchar\_t表示,wchar\_t在windows统一为2字节）,定义了UNICODE宏就要用wchar\_t表示字符,鉴于我们代码中都是char,所以使用本地字符集,即多字节。

|  |  |  |  |
| --- | --- | --- | --- |
| 对比 | 多字节 | UNICODE |  |
| \* | Char | Wchar\_t |
|  | Ifstream | wifstream |
|  | ofstream | wofstream |
|  | string | wstring |
|  |  |  |
|  |  |  |
|  |  |  |

* + 1. 基础知识

如果cpp文件中字符格式为UTF-8的,则在控制面板-区域语言-管理-非Unicode程序语言。。

中需要选择英语美国,这样编译器才会将cpp文件中注释以UTF-8的格式去解析。

QTextCodec \*textCodec = QTextCodec::codecForName("GB18030");

分别设置本地，tr函数，qstring的编码方案

QTextCodec::setCodecForLocale(textCodec);

QTextCodec::setCodecForTr(textCodec);

QTextCodec::setCodecForCStrings(textCodec);

用处：Qstring a(“中文”)，qstring内部存储字符使用unicode编码，但是代码里面“中文”这样的字符串编码方式为 vs下文件 –》高级编码方案中显示的，变为qstring时，需要告诉qstring这个编码方案，即QTextCodec::setCodecForCStrings(textCodec);否则，qstring默认为拉丁编码，从而转换后得到乱码。Tr函数原理也类似。setCodecForLocale是进行本地文件读取时,告诉qt本地文件的编码方式。

Unicode、utf-8、GB2312的区别

Unicode和Gb2312是编码方式，前者国际通用，windows操作系统内部也使用该编码，gb2312是我国的汉字编码方案。

Utf-8 （unicode translation format 叫做unicode转换格式）是编码存储标准，规定了unicode的存储方式，英文1字节，希腊字符2字节，汉字3字节，如“汉”的Unicode编码为6c49,utf-8规定用1-3个字节表示编码，6c49属于0800 至FFFF范围内，使用3字节模板

1110XXXX 10XXXXXX 10XXXXXX ，用6c49的比特流代替X部分，得到E6 B1 89 就是”汉”的unicode编码的utf-8表示形式。

代码页的问题

Windows默认使用unicode表达文字，对于各自GB2312,big5等其他编码，提供代码页，从而也支持这些编码，在控制面板，区域语言 ，administrator中选择不用unicode时，使用哪种编码，若选择chinese，则不用unicode时使用gb2312编码。如果全部使用unicode则不涉及该选项。

* 1. Windows不要修改用户名,影响远程登录

修改用户名只是一个别名,用户文件夹名称等还是以前的,远程登录时依然需要使用以前的用户名。Ref：彻底修改Windows系统用户名。