**Java基础类库**

**一、StringBuffer类**

String 类是所有项目开发之中一定会使用到的一个功能类，并且这个类拥有以下一些功能：

·每一个字符串的常量都属于一个String类的匿名对象，并且不可更改；

·String 有两个常量池：静态常量池、运行时常量池；

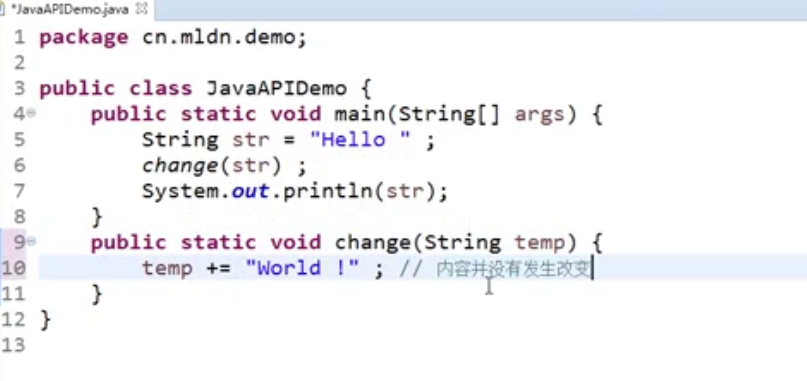
·String 类对象实例化建议使用直接赋值的形式完成，这样可以直接将对象保存到对象池之中以方便下次重用；

虽然String类很好使用，但是如果认真思考会发现String类有一个最大的弊端: 字符串内容无法修改！因此我们我们提供了StringBuffer类来进行字符串的修改。

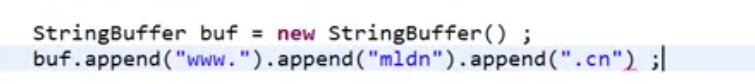
StringBuffer 并不像String 类那样拥有两种实例对象的方式，StringBuffer必须像普通类对象那样首先进行对象的实例化，然后使用才可以调用方法进行处理，而这个时候我们可以调用StringBuffer的如下方法来处理。

·

范例：观察String与StringBuffer类之间的区别



此时str内容并没有改变，是符合String类特性的，内容无法改变。假设str=”hello”我们通常的Str+= ”world”只是重新开辟了world字符串堆空间和 hello world 的字符串堆空间，再将str的引用指向 hello world 堆空间，实际上字符串的内容是无法改变的。



实际上 str += “world” 在编译后所有的 + 都会变成 StringBuffer 的append()

因此我们实际上关注的是 + 而不是append（）方法，因为我们在编译的时候会自动的转换为append（）。并且程序中String与StringBuffer之间可以互相转换

·String类对象变为StringBuffer类可以使用我们的StringBuffer构造方法或者使用append()方法。

·所有的类（包括StringBuffer类）都可以通过toString()方法转换为String类型

在StringBuffer类里面除了可以支持修改字符串的内容以外，还具备一些String类不具备的方法

1、插入数据：public StringBuffer insert(int offset , 数据类型 b)

|  |
| --- |
|  |

结果: [www.mldn.cn](http://www.mldn.cn)

1. 删除指定范围字符串:

public StringBuffer delete(int start , int end)

|  |
| --- |
|  |

1. 字符串反转：public StringBuffer reverse()

|  |
| --- |
|  |

实际上与StringBuffer类相似功能的还有一个StringBuilder类,这个类是在JDK1.5的时候提供的，该类中的方法与StringBuffer类功能相同，他们之间最大的区别在于StringBuffer类中的方法全部是线程安全的，全部使用了我们synchronized关键字进行标注，而我们StringBuilder类属于非线程安全的。

面试题：请解释String 、StringBuilder、StringBuffer的区别。

·String类是字符串的首选类型，其最大的特点是内容不允许修改。

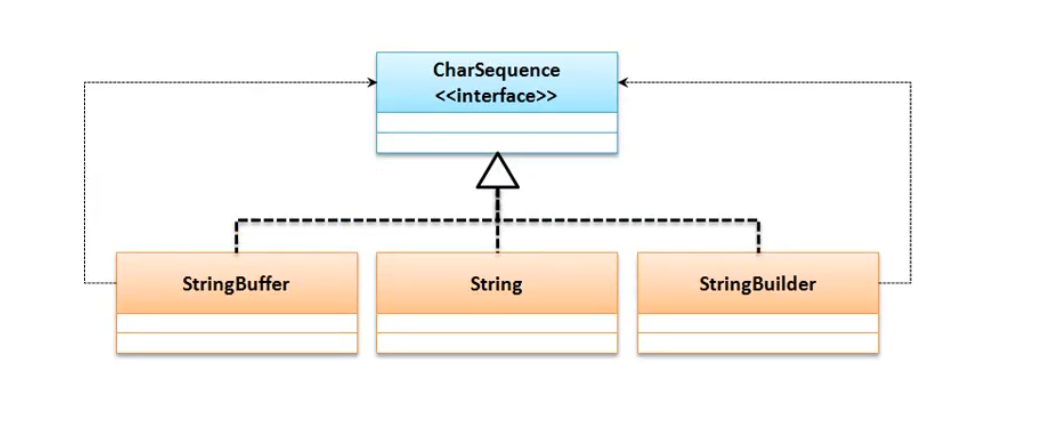
·StringBuffer与我们StringBuilder类的内容是运行修改的。

·StringBuffer 是在JDK1.0的时候提供的，属于线程安全的操作，而StringBuilder是JDK1.5之后提供的，是非线程安全的操作。

1. **CharSequence类**

CharSequence是一个描述字符串结构的接口，在这个接口里我们有三个常用的子类：

|  |  |  |
| --- | --- | --- |
| String类： | StringBuffer类： | StringBuilder类： |
| public final class String  extends Object  implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc | public final class StringBuffer  extends Object  implements Serializable, Comparable<StringBuffer>, CharSequence | public final class StringBuilder  extends Object  implements Serializable, Comparable<StringBuilder>, CharSequence |



现在我们发现只要有字符串就可以为CharSequence接口实例化。

范例：

|  |
| --- |
|  |

CharSequence本身是一个接口，在该接口之中定义有如下方法：

·public char charAt(int index) : 获取指定索引的字符。

·public int length() : 获取字符串的长度。

·public CharSequence subSequence(int start , int end ):截取指定范围字符串。

范例：截取指定范围的字符串

|  |
| --- |
|  |

以后看到一个CharSequence就是描述一个字符串。

1. **AutoCloseable接口**

AutoCloseable 主要用于日后进行资源开发的处理上，实现资源的自动关闭（释放资源），例如：以后进行文件、网络数据库开发的过程中由于服务器的资源有限 ，所以在使用之后一定要关闭资源，这样才可以被跟多的使用者使用。

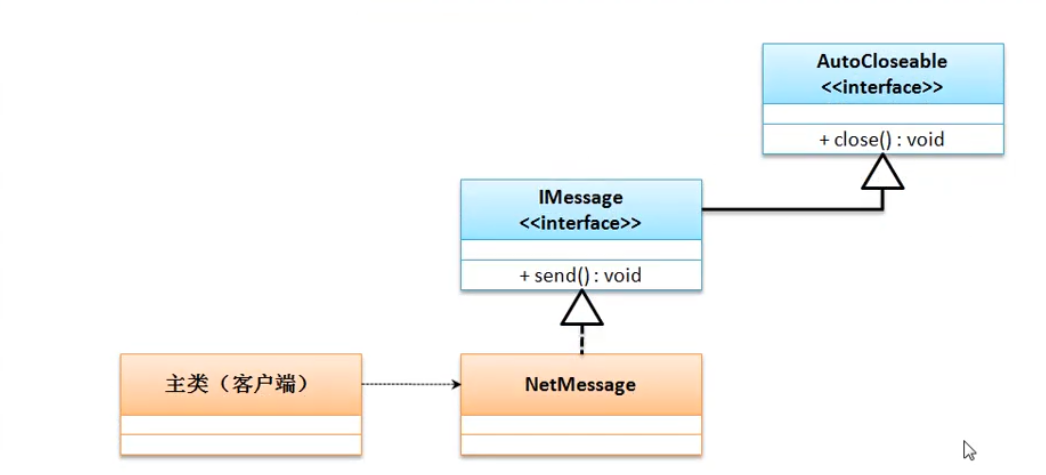
下面为了根号的说明资源的问题，我们将通过一个消息的发送处理来完成。

范例：手工实现资源处理

|  |
| --- |
|  |

那么此时有位设计师说，既然所有的资源都必须在处理完成之后进行关闭，那么是否能够实现一种自动关闭功能。 在这种要求下，推出了AutoCloseable 访问接口， 这个接口是在JDK1.7的时候提供的，并且该接口只提供一个方法：

·关闭方法: public void close() throws Exception ;



要想实现自动关闭处理，除了要使用AutoCloseable 以外还需要结合异常处理语句才可以正常执行。

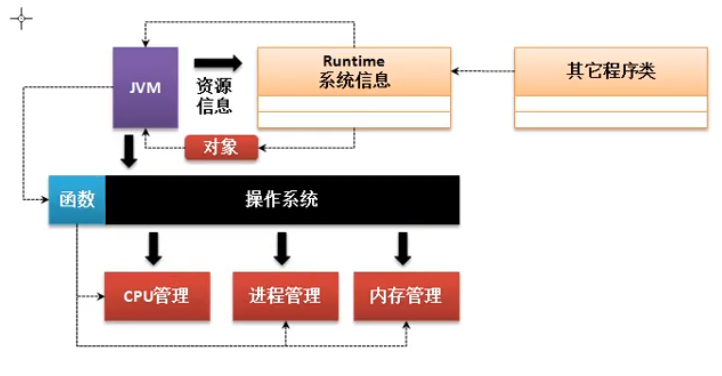
范例:实现自动关闭处理

|  |
| --- |
| package com.company.Java基础类库;  public class AutoClose {  public static void main(String[] args) throws Exception{  try(IMessage nm = new NetMessage("www.mldn.cn")) {  nm.send();  }catch (Exception e){}  }  }  interface IMessage extends AutoCloseable{  /\*\*  \* Function: Send message  \*/  public void send();  }  class NetMessage implements IMessage {  private String msg ;  public NetMessage (String msg){  this.msg = msg ;  }  public boolean open() {  System.out.println("【\*\*\*\*资源获取\*\*\*\*】获取连接资源");  return true;  }  @Override  public void close() throws Exception {  System.out.println("【\*\*\*\*CLOSE\*\*\*\*】关闭发送通道");  }  @Override  public void send() {  if (this.open()){  System.out.println("【\*\*\*\*发送消息\*\*\*\*】" + this.msg);  }  }  } |

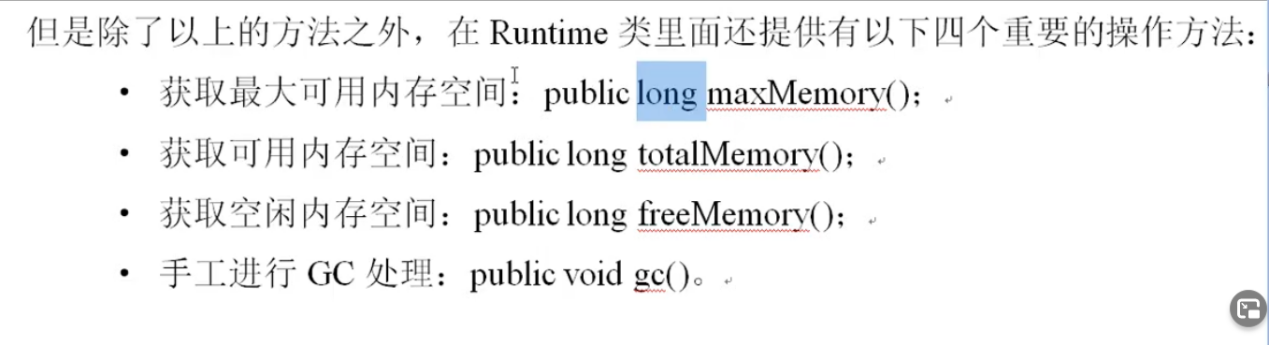
1. **Runtime类**

Runtime描述的是运行时的状态，也就是说在整个的JVM之中，Runtime类是唯一一个与JVM运行状态有关的类，并且都会默认提供有一个该类的实例化对象。

由于在每一个JVM进程里面只允许有一个Runtime类的对象，所以这个类的构造化方法被私有化了，所以这个类使用的是单例设计模式，并且单例设计模式一定会提供有提供一个static方法获取本类实例。



由于我们的Runtime类属于单例设计模式，如果想要获取实例化对象，那么可以依靠类中的getRuntime()方法完成。



Long是表示的字节，当我们返回的类型为字节的时候，就用long

范例: 获取Runtime 类对象

|  |
| --- |
| package com.company.Java基础类库;  public class Run {  public static void main(String[] args) {  Runtime run = Runtime.getRuntime();  System.out.println(run.maxMemory());  System.out.println(run.totalMemory());  System.out.println(run.freeMemory());  System.out.println("\*\*\*end\*\*\*");  }  } |

面试题:什么是GC？如何处理？

·GC是指（Garbage Collect）垃圾收集器， 是可以由系统自动调用的垃圾释放功能，或者使用Runtime类中的gc()手工调用。

1. **System类**
   * 1. **具体实现**

System类是一直陪伴在我们学习过程中的程序类，之前使用的系统输出就是System类，它还定义有一些其它的处理方法。

·数组拷贝: arraycopy​(Object src, int srcPos, Object dest, int destPos, int length) ;

·获取当前的日期时间数值：public static **long** currentTimeMil();

·进行垃圾回收:public static void gc() ;

范例:操作耗时统计（currentTimeMil()）

|  |
| --- |
|  |

在System中也定义了有一个gc()方法，但是这gc方法不是重新定义的方法，而是调用的Runtime类中的gc操作（Runtime.getRuntime().gc()）。

1. Cleaner类

Cleaner 是JDK1.9之后提供的一个对象清理操作，其主要功能是进行finialize()方法的替代。finialize()就是指资源回收操作，JDK1.9之后不建议使用finialize()，而是使用AutoCloseable 或者 Cleaner类进行回收处理。

范例:JDK1.9之后的手工回收处理

|  |
| --- |
| package com.company.Java基础类库;  import java.lang.ref.Cleaner;  import java.lang.reflect.Method;  public class Clea {  public static void main(String[] args) {  try(MemberCleaning mc = new MemberCleaning()) {  } catch (Exception e) {  e.printStackTrace();  }  }  }  class Member implements Runnable {  public Member (){  System.out.println("李斯诞生了！");  }  @Override  public void run() {  System.out.println("\*\*\*【回收李斯】\*\*\*");  }  }  class MemberCleaning implements AutoCloseable{  private static final Cleaner cleaner = Cleaner.create();  private final Member member ;  private final Cleaner.Cleanable cleanable;  public MemberCleaning(){  this.member = new Member() ;  this.cleanable = cleaner.register(this, this.member) ;  }  @Override  public void close() throws Exception {  this.cleanable.clean();  }  } |

在新一代的清楚回收处理过程之中，更多的考虑的是多线程的使用，即:为了防止有可能造成的延迟处理，所以许多对象回收前的处理都是单独通过一个线程完成的。这样能保证我们整体执行性能的提高。

1. **对象克隆**
2. 具体实现

所谓的对象克隆就是对象的复制，而且属于一种全新的复制。即使用已有对象内容创建一个新的对象，如果要进行对象的克隆的话需要使用到Object类之中提供的clone() 方法。

·protected Object clone()

throws CloneNotSupportedException

所有的类都会继承Object父类，所以所有的类都有clone()方法，但是不是所有的类都希望被clone。如果要想实现对象克隆，那么对象所在类必须实现一个接口:Cloneable,此接口没有任何的方法，是因为它描述的是一种能力。

范例:对象的克隆

|  |
| --- |
|  |