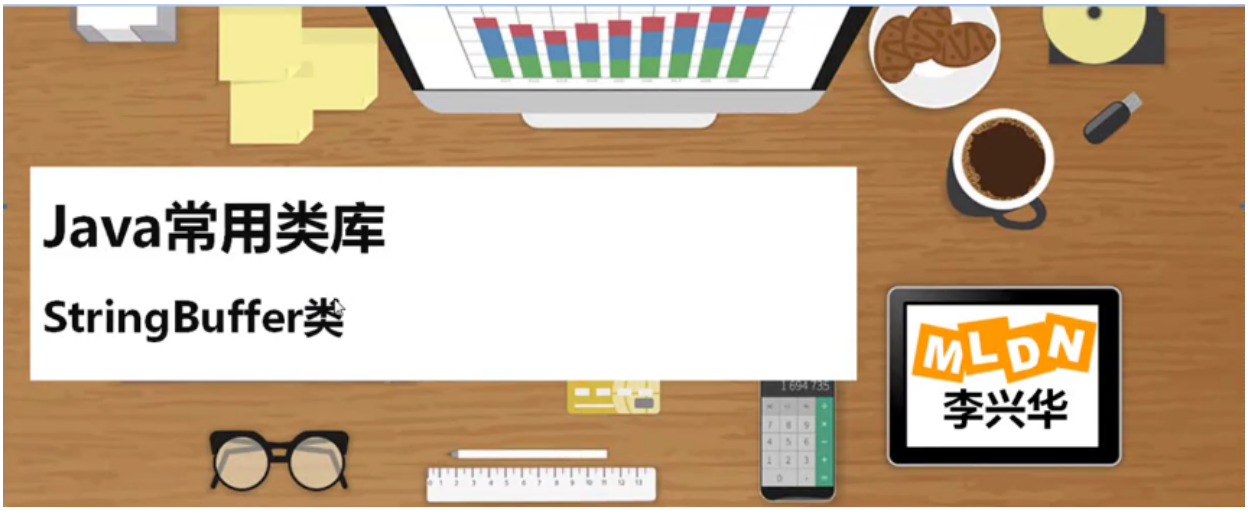


博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

String类是在所有项目开发之中一定会使用到的一个功能类，并且这个类拥有如下的特点：

·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，这样可以直接讲对象保存在对象池之中以方便下次重用；

虽然String类很好使用，但是如果认真去思考也会发现其最大的弊端：内容不允许修改，虽然大部分的情况下都不会涉及到字符串内容的频繁的修改，但是依然可能会存在有这样的情况，所以为了解决此问题，专门提供有一个StringBuffer类可以实现字符串内容的修改处理。

StringBuffer并不像String类那样拥有两种对象实例化方式，StringBuffer必须像普通类对象那样首先进行实例化，而后才可以调用方法执行处理，而这个时候可以考虑使用StringBuffer类中的如下方法：

·构造方法：public StringBuffer();

·构造方法：public StringBuffer(String str)，接收初始化字符串内容；

·数据追加：public StringBuffer append(数据类型 变量)，相当于字符串中的“+”操作；

范例：观察String与StringBuffer对比

String类对象引用传递：	**StringBuffer类对象引用传递：**

public class JavaAPIDemo {
public static void main(String[] args) {
String str = "Hello " ;
	public class JavaAPIDemo {
	public static void main(String[] args) {
	StringBuffer buf = new StringBuffer("Hello ") ;

```
        change(str) ;
        System.out.println(str);
    }
    public static void change(String temp) {
        temp += "World !" ;       // 内容并没有发生改变
    }
}
```

```
        change(buf) ;
        System.out.println(buf.toString());
    }
    public static void change(StringBuffer temp) {
        temp.append("World !") ;       // 内容并没有发生改变
    }
}
```

实际上大部分的情况下，很少会出现有字符串内容的改变，这种改变指的并不是针对于静态常量池的改变。

范例：分析一下已有问题

```
public class JavaAPIDemo {
    public static void main(String[] args) {
        String strA = "www.mldn.cn" ;
        String strB = "www." + "mldn" + ".cn" ;
        System.out.println(strA == strB);
    }
}
```

这个时候的strB对象的内容并不算是改变，或者更加严格的意义上来讲，对于现在的strB当程序编译之后会变为如下的形式：

```
String strB = "www." + "mldn" + ".cn" ;
```

```
StringBuffer buf = new StringBuffer() ;
buf.append(".cn").insert(0, "www.").insert(4, "mldn") ;
```

所有的“+”在编译之后都变为了StringBuffer中的append()方法，并且在程序之中StringBuffer与String类对象之间本来就可以直接互相转换：

·String类对象变为StringBuffer可以依靠StringBuffer类的构造方法或者使用append()方法；

·所有的类对象都可以通过toString()方法将其变为String类型；

在StringBuffer类里面除了可以支持有字符串内容的修改之外，实际上也提供有了一些String类所不具备的方法。

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

```
public class JavaAPIDemo {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer() ;
        buf.append(".cn").insert(0, "www.").insert(4, "mldn") ;
        System.out.println(buf);
    }
}
```

2、删除指定范围的数据：public StringBuffer delete(int start, int end)

```
public class JavaAPIDemo {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer() ;
        buf.append("Hello World !").delete(6, 12).insert(6, "MLDN") ;
```

```
            System.out.println(buf);
        }
}
```

3、字符串内容反转：public StringBuffer reverse()

```
public class JavaAPIDemo {
        public static void main(String[] args) {
                StringBuffer buf = new StringBuffer() ;
                buf.append("Hello World !") ;
                System.out.println(buf.reverse());
        }
}
```

实际上与StringBuffer类还有一个类似的功能类：StringBuilder类，这个类是在JDK1.5的时候提供的，该类中提供的方法与StringBuffer功能相同，最大的区别在于StringBuffer类中的方法属于线程安全的，全部使用了synchronized关键字进行标注，而StringBuilder类属于非线程安全的。

面试题：请解释String、StringBuffer、StringBuilder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StringBuilder类的内容允许修改；

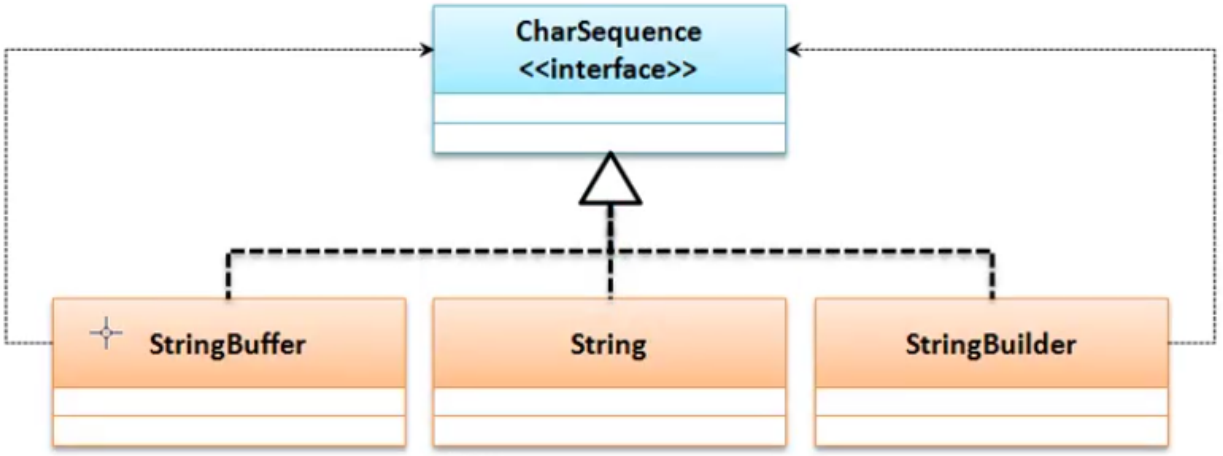
·StringBuffer是在JDK1.0的时候提供的，属于线程安全的操作，而StringBuilder是在JDK1.5之后提供的，属于非线程安全的操作。

2、具体内容

CharSequence是一个描述字符串结构的接口，在这个接口里面一般发现有三种常用子类：

String类	StringBuffer类	StringBuilder类
public final class String extends Object implements Serializable,	public final class StringBuffer extends Object	public final class StringBuilder extends Object implements

Comparable<String>, **CharSequence**	implements Serializable, **CharSequence**	Serializable, **CharSequence**



现在只要有字符串就可以为CharSequence接口实例化。

```
public class JavaAPIDemo {
	public static void main(String[] args) {
		CharSequence str = "www.mldn.cn" ;	// 子类实例向父接口转型
	}
}
```

Charsequence本身是一个接口，在该接口之中也有定义如下操作方法：

·获取指定索引字符：public char charAt(int index);

·获取字符串的长度：public int length();

·截取部分字符串：CharSequence subSequence(int start,int end);

范例：字符串截取

```
public class JavaAPIDemo {
	public static void main(String[] args) {
		CharSequence str = "www.mldn.cn" ;	// 子类实例向父接口转型
		CharSequence sub = str.subSequence(4, 8) ;
		System.out.println(sub);
	}
}
```

以后只要看见了Charsequence描述的就是一个字符串。

2、具体内容

AutoCloseable主要是用于日后进行资源开发的处理上，以实现资源的自动关闭（释放资源），例如：在以后进行文件、网络、数据库开发的过程之中由于服务器的资源有限，所以使用之后一定要关闭资源，这样才可以被更多的使用者所使用。

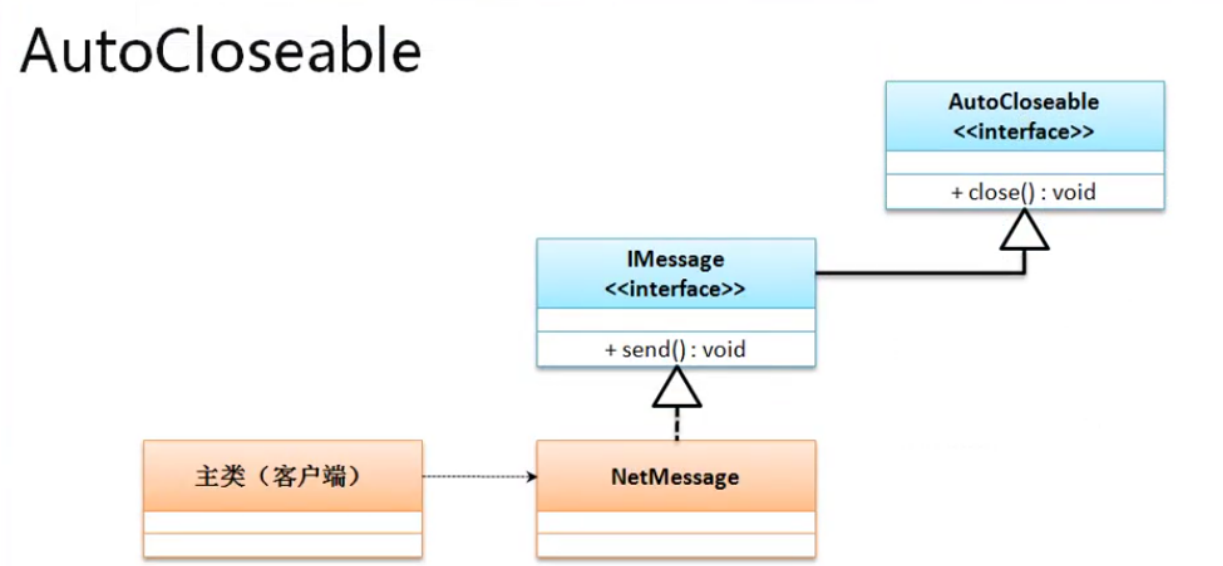
下面为了更好的说明资源的问题，将通过一个消息的发送处理来完成。

范例：手工实现资源处理

```
public class JavaAPIDemo {
	public static void main(String[] args) {
		NetMessage nm = new NetMessage("www.mldn.cn") ;	// 定义要发送的处理
		nm.send();		// 消息发送
		nm.close();	// 关闭连接
	}
}
interface IMessage {
	public void send() ; // 消息发送
}
class NetMessage implements IMessage {	// 实现消息的处理机制
	private String msg ;
	public NetMessage(String msg) {
		this.msg = msg ;
	}
	public boolean open() {  // 获取资源连接
		System.out.println("【OPEN】获取消息发送连接资源。");
		return true ;
	}
	@Override
	public void send() {
		if (this.open()) {
			System.out.println("【*** 发送消息 ***】" + this.msg);
		}
	}
	public void close() {
		System.out.println("【CLOSE】关闭消息发送通道。");
	}
}
```

此时有位设计师说了，既然所有的资源完全处理之后都必须进行关闭操作，那么能否实现一种自动关闭的功能呢？在这样的要求下，推出了AutoCloseable访问接口，这个接口是在JDK1.7的时候提供的，并且该接口只提供有一个方法：

·关闭方法：void close() throws Exception；



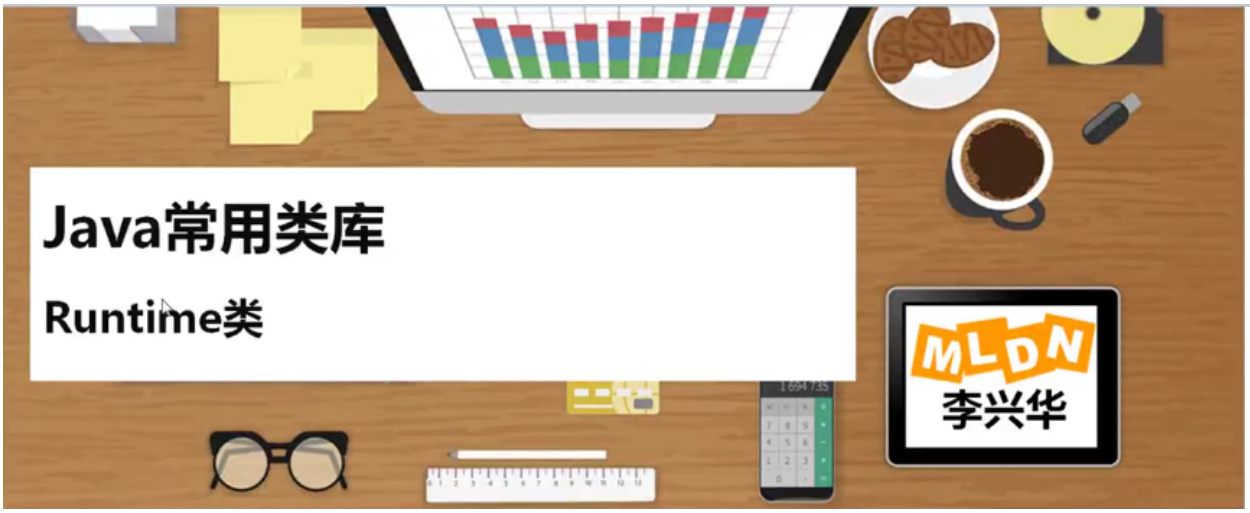
要想实现自动关闭处理，除了要使用AutoCloseable之外，还需要结合异常处理语句才可以正常调用。

范例：实现自动关闭处理

```
public class JavaAPIDemo {
    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 {
    public void send() ; // 消息发送
}
class NetMessage implements IMessage {    // 实现消息的处理机制
    private String msg ;
    public NetMessage(String msg) {
        this.msg = msg ;
    }
    public boolean open() {  // 获取资源连接
        System.out.println("【OPEN】获取消息发送连接资源。");
        return true ;
    }
    @Override
    public void send() {
        if (this.open()) {
            System.out.println("【*** 发送消息 ***】" + this.msg);
        }
    }
    public void close() throws Exception {
```

```
            System.out.println("【CLOSE】关闭消息发送通道。");
        }
}
```

在以后的章节之中胡接触到资源的关闭问题，往往都会见到AutoCloseable接口的使用。

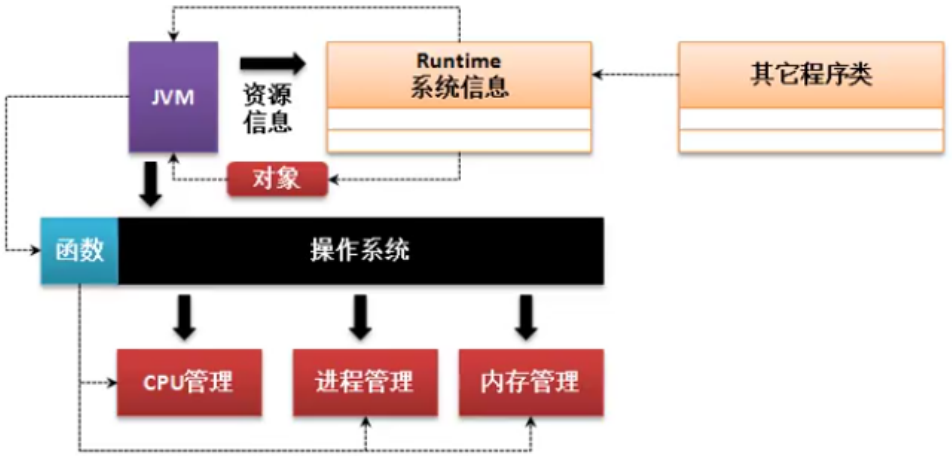


2、具体内容

Runtime描述的是运行时的状态，也就是说在整个的JVM之中，Runtime类时唯一 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进程里面只允许提供有一个Runtime类的对象，所以这个类的构造方法被默认私有化了，那么就证明该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一个static方法获取本类实例。

Runtime



由于Runtime类属于单例设计模式，如果要想获取实例化对象，那么就可以依靠类中的getRuntime()方法完成：

·**获取实例化对象**：public static Runtime getRuntime();

通过这个类中的availableProcessors()方法可以获取本机的CPU内核数；

范例：获取Runtime对象

```
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Runtime run = Runtime.getRuntime() ; // 获取实例化对象
		System.out.println(run.availableProcessors());
	}
}
```

但是除了以上的方法之外，在Runtime类里面还提供有以下四个重要的操作方法：

·获取最大可用内存空间：public long maxMemory()，默认的配置为本机系统内存的4分之1；

·获取可以内存空间：public long totalMemory()，默认的配置问本机系统内存的64分之1；

·获取空闲内存空间：public long freeMemory()；

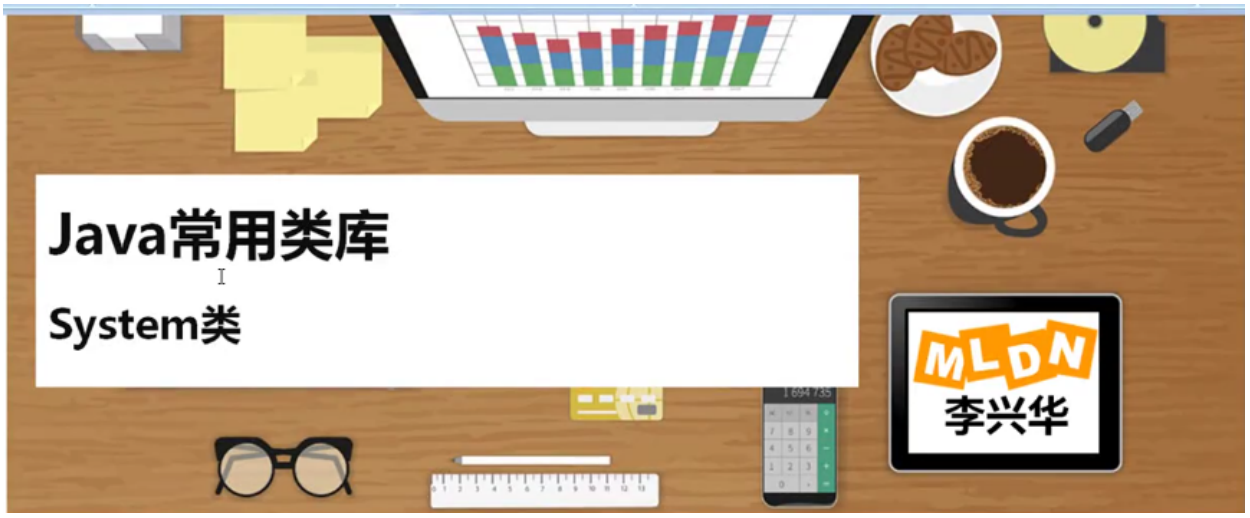
·手工进行GC处理：public void gc()；

范例：观察内存状态

```
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Runtime run = Runtime.getRuntime() ; // 获取实例化对象
		System.out.println("【1】MAX_MEMORY：" + run.maxMemory());
		System.out.println("【1】TOTAL_MEMORY：" + run.totalMemory());
		System.out.println("【1】FREE_MEMORY：" + run.freeMemory());
		String str = "" ;
		for (int x = 0 ; x < 30000; x ++) {
			str += x ; // 产生大量的垃圾空间
		}
		System.out.println("【2】MAX_MEMORY：" + run.maxMemory());
		System.out.println("【2】TOTAL_MEMORY：" + run.totalMemory());
		System.out.println("【2】FREE_MEMORY：" + run.freeMemory());
		run.gc();
		System.out.println("【3】MAX_MEMORY：" + run.maxMemory());
		System.out.println("【3】TOTAL_MEMORY：" + run.totalMemory());
		System.out.println("【3】FREE_MEMORY：" + run.freeMemory());
	}
}
```

面试题：请问什么是GC？如何处理？

·GC（Garbage Collector）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。



2、具体内容

System类是一直陪伴着我们学习的程序类，之前使用的系统输出采用的就是System类中的方法，而后在System类里面也定义有了一些其它的处理方法：

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

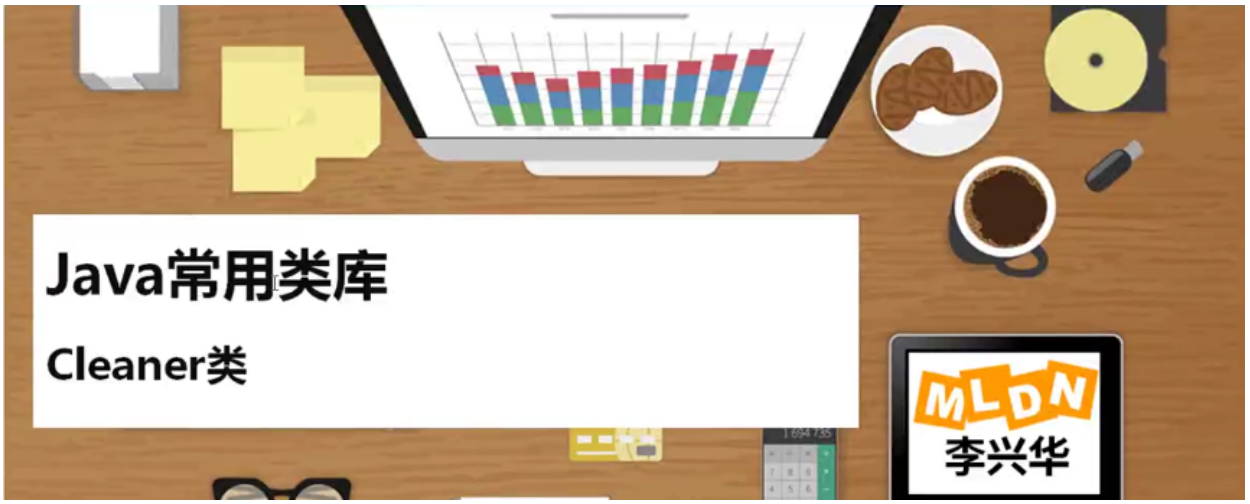
·获取当前的日期时间数值：public static **long** currentTimeMillis();

·进行垃圾回收：public static void gc();

范例：操作耗时的统计

```
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		long start = System.currentTimeMillis() ;
		Runtime run = Runtime.getRuntime() ; // 获取实例化对象
		String str = "" ;
		for (int x = 0 ; x < 30000; x ++) {
			str += x ; // 产生大量的垃圾空间
		}
		long end = System.currentTimeMillis() ;
		System.out.println("操作耗时：" + (end - start));
	}
}
```

在System类里面会发现也提供有一个gc()方法，但是这个gc()方法并不是重新定义的新方法，而是继续调用了Runtime的gc()操作（Runtime.getRuntime().gc();）。



2、具体内容

Cleaner是在JDK1.9之后提供的一个对象清理操作，其主要的功能是进行finalize()方法的替代。在C++语言里面有两种特殊的函数：构造函数、析构函数（对象手工回收），在Java里面所有的垃圾空间都是通过GC自动回收的，所以狠毒情况下是不需要使用这类析构函数的，也正是因为入池，所以Java并没有提供这方面支持。

但是Java本身依然提供了给用户收尾的操作，每一个实例化对象在回收之前至少给它一个喘息的机会，最初实现对象收尾处理的方法是Object类中所提供的finalize()方法，这个方法的定义如下：

```
@Deprecated(since="9")
protected void finalize() throws Throwable
```

该替换指的是不建议继续使用这个方法了，而是说子类可以继续使用这个方法名称。但是这个方法上最大的特点是抛出了一个Throwable异常类型，而这个异常类型分为两个子类型：Error、Exception，平常所处理的都是Exception。

范例：观察传统回收

```
class Member {
	public Member() {
		System.out.println("【构造】在一个雷电交加的日子里面，林强诞生了。");
	}
	@Override
	protected void finalize() throws Throwable {
		System.out.println("【回收】最终你一定要死的。");
		throw new Exception("我还要再活500年...") ;
	}
}
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Member mem = new Member() ;  // 诞生
		mem = null ; // 成为垃圾
		System.gc();
		System.out.println("太阳照常升起，地球照样转动。");
	}
```

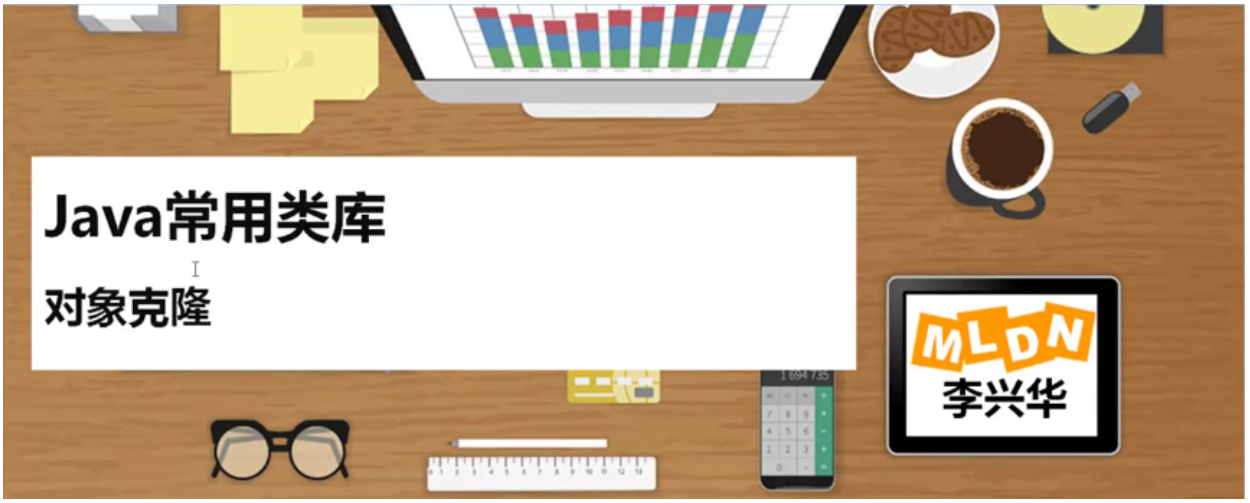
```
}
```

但是从JDK1.9开始，这一操作已经不建议使用了，而对于对象回收释放。从JDK1.9开始建议开发者使用AutoCloseable或者使用java.lang.ref.Cleaner类进行回收处理（Cleaner也只持有AutoCloseable处理）

```
import java.lang.ref.Cleaner;

class Member implements Runnable {
	public Member() {
		System.out.println("【构造】在一个雷电交加的日子里面，林强诞生了。");
	}
	@Override
	public void run() {   // 执行清除的时候执行的是此操作
		System.out.println("【回收】最终你一定要死的。");
	}
}
class MemberCleaning implements AutoCloseable { // 实现清除的处理
	private static final Cleaner cleaner = Cleaner.create() ; // 创建一个清除处理
	private Member member ;
    private Cleaner.Cleanable cleanable ;
    public MemberCleaning() {
	this.member = new Member() ; // 创建新对象
	this.cleanable = this.cleaner.register(this, this.member) ;      // 注册使用的对象
	}
	@Override
	public void close() throws Exception {
		this.cleanable.clean(); // 启动多线程
	}
}
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		try (MemberCleaning mc = new MemberCleaning()){
			// 中间可以执行一些相关的代码
		} catch (Exception e) {}
	}
}
```

在新一代的清除回收处理的过程之中，更多的情况下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。



2、具体内容

所谓的对象克隆指的就是对象的赋值，而且属于全新的复制。即：使用已有对象内容创建一个新的对象，如果要想进行对象克隆需要使用到Object类中提供的clone()方法：

protected Object clone() throws CloneNotSupportedException;

所有的类都会继承Object父类，所以所有的类都一定会有clone()方法，但是并不是所有的类都希望被克隆。所以如果要想实现对象克隆，那么对象所在的类需要实现一个Cloneable接口，此接口并没有任何的方法提供，是因为它描述的是一种能力。

范例：实现对象克隆

```
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Member memberA = new Member("林强",30) ;
		Member memberB = (Member) memberA.clone() ;
		System.out.println(memberA);
		System.out.println(memberB);
	}
}
class Member implements Cloneable {
	private String name ;
	private int age ;
	public Member(String name,int age) {
		this.name = name ;
		this.age = age ;
	}
	@Override
	public String toString() {
		return "【" + super.toString() + "】name = " + this.name + "、age = " + this.age ;
	}
	@Override
	protected Object clone() throws CloneNotSupportedException {
		return super.clone();	// 调用父类中提供的clone()方法
	}
}
```

如果在开发之中不是非常特别的需求下，很少会出现有对象克隆的需求。