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

2、具体内容

UUID是一种生成无重复字符串的一种程序类，这种程序类的主要功能是根据时间戳实现一个自动的无重复的字符串定义。

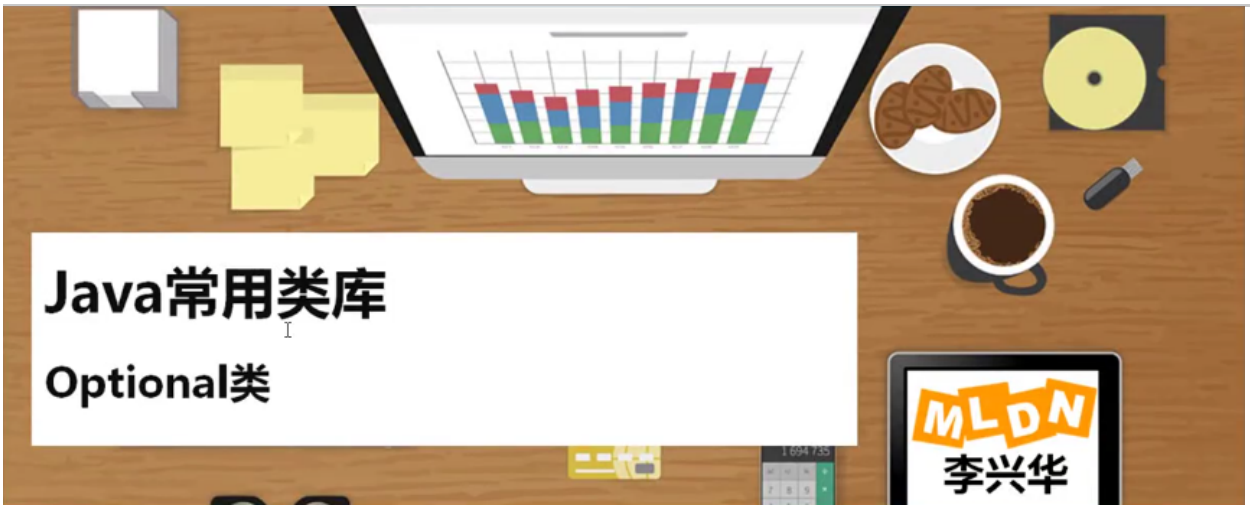
一般在获取UUID的时候往往都是随机生成的一个内容，所以可以通过如下方式获取：

·获取UUID对象：public static UUID randomUUID();

·根据字符串获取UUID内容：public static UUID fromString(String name);

```
import java.util.UUID;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		UUID uid = UUID.randomUUID() ;
		System.out.println(uid.toString());
	}

}
```

在对一些文件进行自动命名处理的情况下，UUID类型非常好用。



2、具体内容

Optional类的主要功能是进行null的相关处理，在以前进行程序开发的时候，如果为了防止程序之中出现空指向异常，往往可以追加有null的验证。

范例：传统的引用传递问题

```
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		MessageUtil.useMessage(MessageUtil.getMessage());
	}
}
class MessageUtil {
	private MessageUtil() {}
	public static IMessage getMessage() {
		return null ;
	}
	public static void useMessage(IMessage msg) {
		if (msg != null) {
			System.out.println(msg.getContent()); // 有可能因为出现null，而导致空指向
		}
	}
}
interface IMessage {
	public String getContent() ;
}
class MessageImpl implements IMessage {
	@Override
	public String getContent() {
		return "www.mldn.cn";
	}
}
```

在接收引用的一方往往都是被动的进行判断，所以为了解决这种被动的处理操作，在Java类中提供有一个Optional的类，这个类可以实现null的处理操作，这个类里面提供有如下的一些操作方法：

·返回空数据：public static <T> Optional<T> empty();

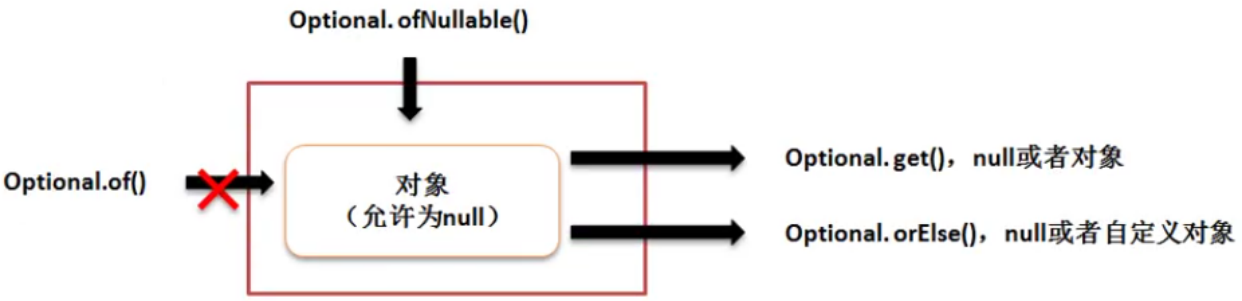
·获取数据：public T get();

·保存数据，但是不允许出现Null：public static <T> Optional<T> of(T value);

|- 如果在保存数据的时候存在有null，则会抛出NullPointerException异常；

·保存数据，允许为空：public static <T> Optional<T> ofNullable(T value);

·空的时候返回其它数据：public T orElse(T other);



范例：修改程序，按照正规的结构完成

```
import java.util.Optional;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
//		IMessage temp = MessageUtil.getMessage().get() ; // 获取数据
//		MessageUtil.useMessage(temp);
		Optional.of(null);
	}
}
class MessageUtil {
	private MessageUtil() {}
	public static Optional<IMessage> getMessage() {
		return Optional.of(new MessageImpl()) ;	// 有对象
	}
	public static void useMessage(IMessage msg) {
		if (msg != null) {
			System.out.println(msg.getContent()); // 有可能因为出现null，而导致空指向
		}
	}
}
interface IMessage {
	public String getContent() ;
}
class MessageImpl implements IMessage {
	@Override
	public String getContent() {
		return "www.mldn.cn";
	}
}
```

如果说现在数据保存的内容是Null，则就会在保存处出现异常：

```
public static Optional<IMessage> getMessage() {
	return Optional.of(null) ; // 有对象
}
```

```
Exception in thread "main" java.lang.NullPointerException
	at java.util.Objects.requireNonNull(Unknown Source)
	at java.util.Optional.<init>(Unknown Source)
	at java.util.Optional.of(Unknown Source)
	at com.mdln.www.MessageUtil.getMessage(JavaAPIDemo.java:14)
	at com.mdln.www.JavaAPIDemo.main(JavaAPIDemo.java:7)
```

由于Optional类中允许保存有null的内容，所以在数据获取的时候也可以进行null的处理。但是如果为Null，则在使用get()获取数据的时候就会出现“Exception in thread

"main" java.util.NoSuchElementException: No value present”异常信息，所以此时可以更换为orElse()方法。

范例：处理null

```
import java.util.Optional;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		IMessage temp = MessageUtil.getMessage().orElse(new MessageImpl()) ; // 获取数据
		MessageUtil.useMessage(temp);
	}
}
class MessageUtil {
	private MessageUtil() {}
	public static Optional<IMessage> getMessage() {
		return Optional.ofNullable(null) ;  // 有对象
	}
	public static void useMessage(IMessage msg) {
		if (msg != null) {
			System.out.println(msg.getContent()); // 有可能因为出现null，而导致空指向
		}
	}
}
interface IMessage {
	public String getContent() ;
}
class MessageImpl implements IMessage {
	@Override
	public String getContent() {
		return "www.mldn.cn";
	}
}
```

在所有引用数据类型的操作处理之中，null是一个重要的技术问题，所以JDK1.8后提供的这个新的类对于null的处理很有帮助，同时也是在日后进行项目开发之中使用次数很多的一个程序类。

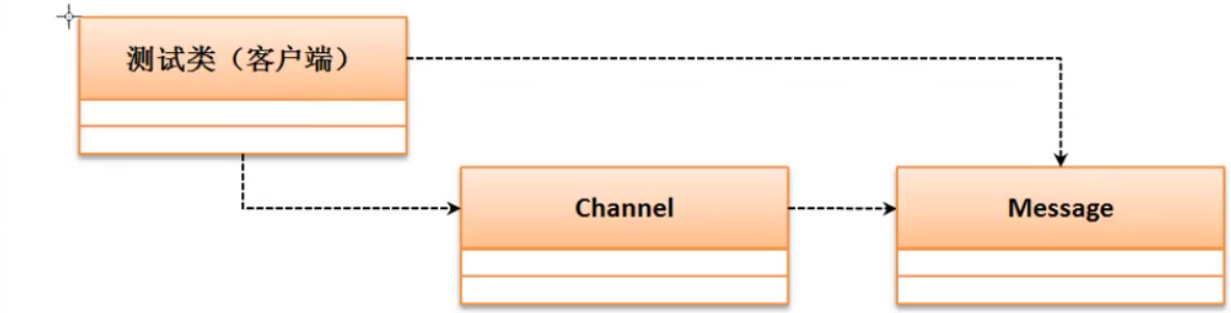
2、具体内容

在真正去了解ThreadLocal类作用的时候下面编写一个简单的程序做一个先期的分析。

范例：现在定义这样的一个结构

```
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Message msg = new Message() ;  // 实例化消息主体对象
		msg.setInfo("www.mldn.cn"); // 设置要发送的内容
		Channel.setMessage(msg); // 设置要发送的消息
		Channel.send(); // 发送消息
	}
}
class Channel {		// 消息的发送通道
	private static Message message ;
	private Channel() {}
	public static void setMessage(Message m) {
		message = m ;
	}
	public static void send() {		// 发送消息
		System.out.println("【消息发送】" + message.getInfo());
	}
}
class Message {		// 要发送的消息体
	private String info ;
	public void setInfo(String info) {
		this.info = info;
	}
	public String getInfo() {
		return info;
	}
}
```

ThreadLocal



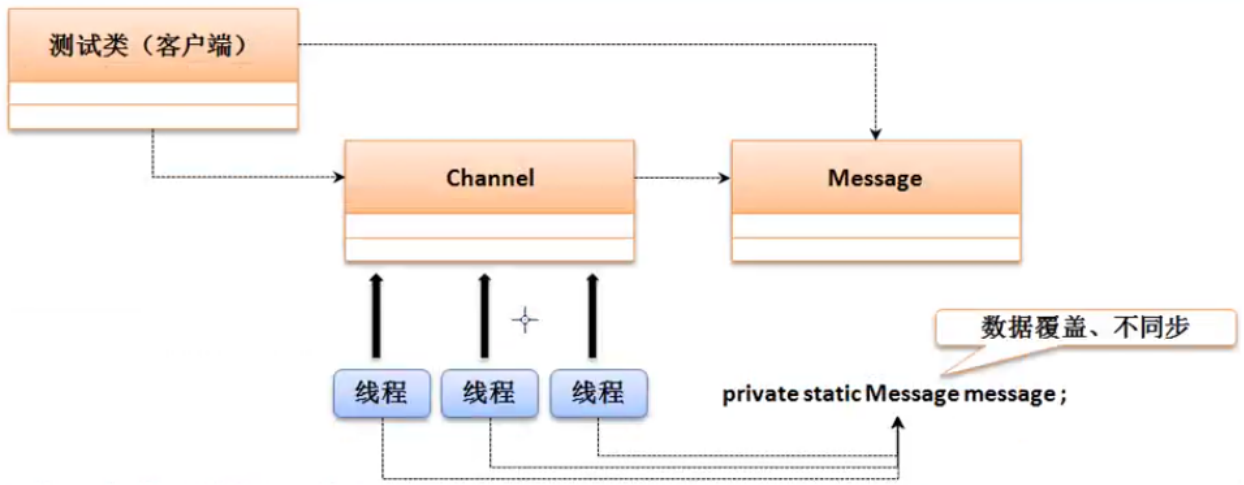
对于当前的程序实际上采用的是一种单线程的模式来进行处理的。那么如果在多线程的状态下能否实现完全一致的操作效果呢？为此将启动三个线程进行处理。

范例：多线程的影响

```
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
```

```
            new Thread(()->{
                Message msg = new Message() ;   // 实例化消息主体对象
                msg.setInfo("第一个线程的消息"); // 设置要发送的内容
                Channel.setMessage(msg); // 设置要发送的消息
                Channel.send(); // 发送消息
            },"消息发送者A") .start() ;
            new Thread(()->{
                Message msg = new Message() ;   // 实例化消息主体对象
                msg.setInfo("第二个线程的消息"); // 设置要发送的内容
                Channel.setMessage(msg); // 设置要发送的消息
                Channel.send(); // 发送消息
            },"消息发送者B") .start() ;
            new Thread(()->{
                Message msg = new Message() ;   // 实例化消息主体对象
                msg.setInfo("第三个线程的消息"); // 设置要发送的内容
                Channel.setMessage(msg); // 设置要发送的消息
                Channel.send(); // 发送消息
            },"消息发送者C") .start() ;
        }
}

class Channel {        // 消息的发送通道
        private static Message message ;
        private Channel() {}
        public static void setMessage(Message m) {
            message = m ;
        }
        public static void send() {       // 发送消息
            System.out.println("【" + Thread.currentThread().getName() + "、消息发送】" +
message.getInfo());
        }
}
class Message {       // 要发送的消息体
        private String info ;
        public void setInfo(String info) {
            this.info = info;
        }
        public String getInfo() {
            return info;
        }
}
```

这个时候消息的处理产生了影响。



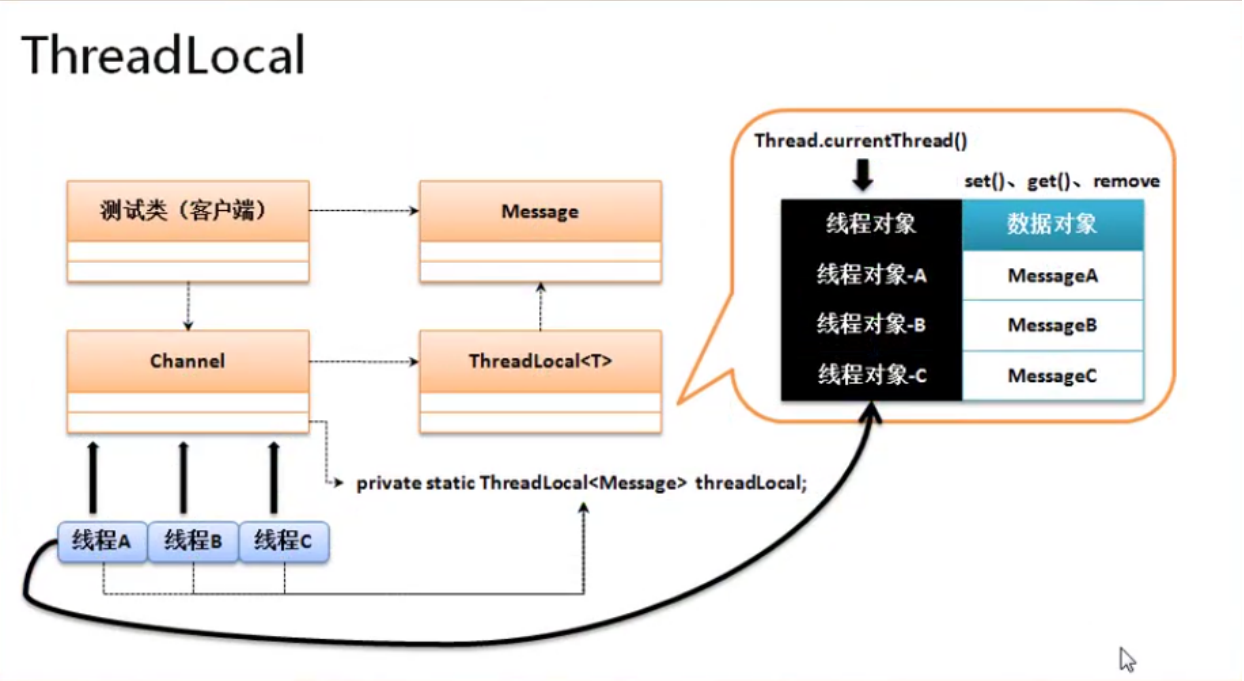
在保持Channel（所有发送的通道）核心结构不改变的情况下，需要考虑到每个线程的独立操作问题。那么在这样的情况下就发现对于channel类而言除了要保留有发送的消息之外，还应该多存放有一个每一个线程的标记（当前线程），那么这个时候就可以通过ThreadLocal类来存放数据。在ThreadLocal类里面提供有如下的操作方法：

·构造方法：public ThreadLocal();

·设置数据：public void set(T value);

·取出数据：public T get();

·删除数据：public void remove();

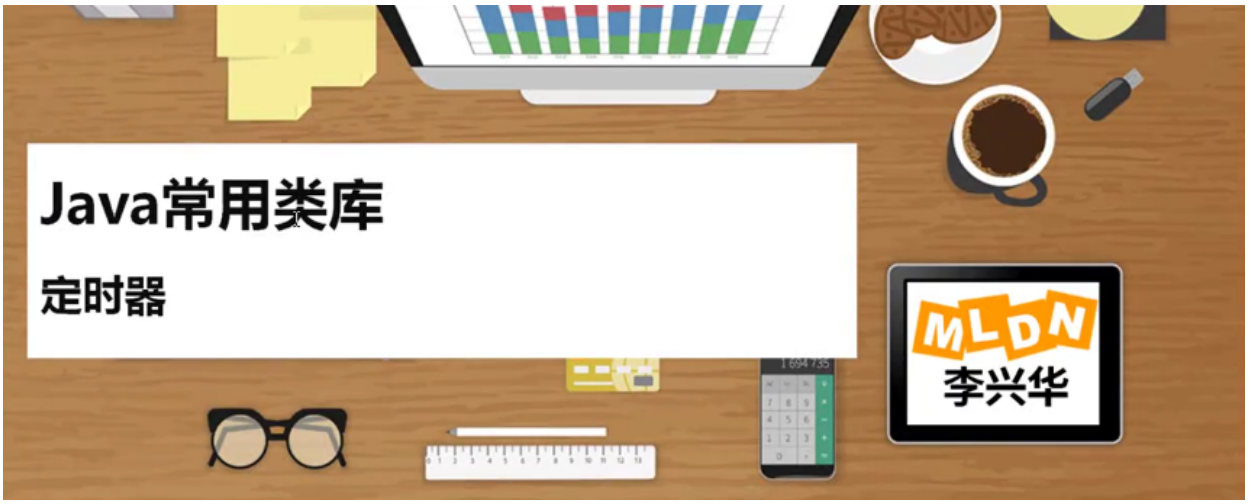


范例：解决线程同步问题

```
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        new Thread(()->{
            Message msg = new Message() ;  // 实例化消息主体对象
            msg.setInfo("第一个线程的消息"); // 设置要发送的内容
            Channel.setMessage(msg); // 设置要发送的消息
            Channel.send(); // 发送消息
```

```
            },"消息发送者A") .start() ;
            new Thread(()->{
                Message msg = new Message() ;  // 实例化消息主体对象
                msg.setInfo("第二个线程的消息"); // 设置要发送的内容
                Channel.setMessage(msg); // 设置要发送的消息
                Channel.send(); // 发送消息
            },"消息发送者B") .start() ;
            new Thread(()->{
                Message msg = new Message() ;  // 实例化消息主体对象
                msg.setInfo("第三个线程的消息"); // 设置要发送的内容
                Channel.setMessage(msg); // 设置要发送的消息
                Channel.send(); // 发送消息
            },"消息发送者C") .start() ;
        }
}
class Channel {       // 消息的发送通道
        private static final ThreadLocal<Message> THREADLOCAL = new
ThreadLocal<Message>() ;
        private Channel() {}
        public static void setMessage(Message m) {
            THREADLOCAL.set(m);  // 向ThreadLocal中保存数据
        }
        public static void send() {       // 发送消息
            System.out.println("【" + Thread.currentThread().getName() + "、消息发送】" +
THREADLOCAL.get().getInfo());
        }
}
class Message {       // 要发送的消息体
        private String info ;
        public void setInfo(String info) {
            this.info = info;
        }
        public String getInfo() {
            return info;
        }
}
```

每一个线程通过ThreadLocal只允许保存一个数据。



2、具体内容

定时器的主要操作是进行定时任务的处理，就好比你们每天早晨起来的铃声一样。在Java中提供有定时任务的支持，但是这种任务的处理只是实现了一种间隔触发的操作。

如果要想实现定时的处理操作主要需要有一个定时操作的主体类，以及一个定时任务的控制。可以使用两个类实现：

·java.util.TimerTask类：实现定时任务处理；

·java.util.Timer类：进行任务的启动，启动的方法；

|- 任务启动：public void schedule(TimerTask task,long delay)、延迟单位为毫秒；

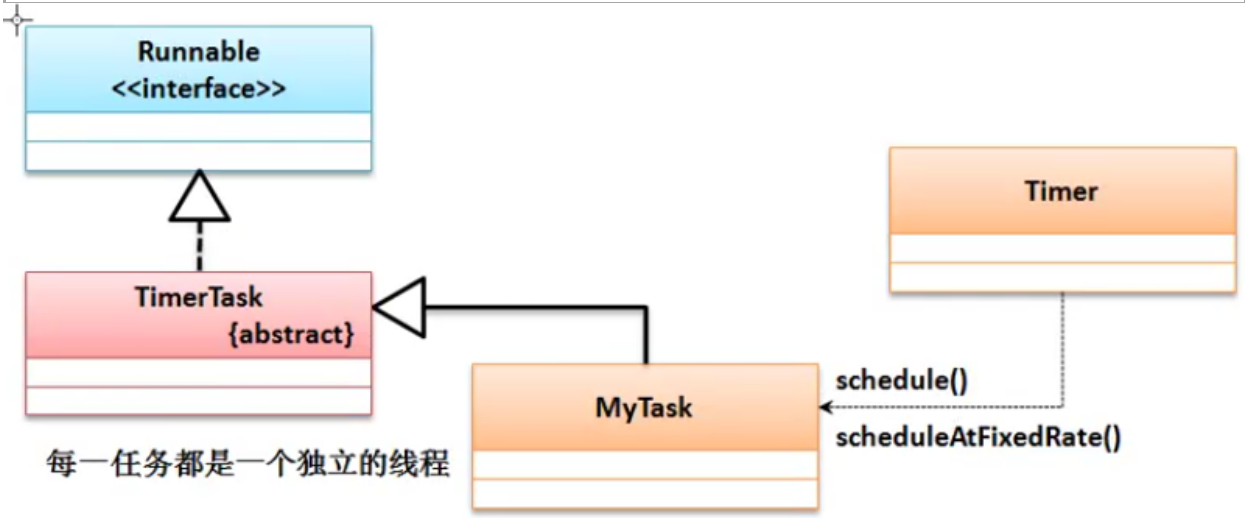
|- 间隔触发：public void scheduleAtFixedRate(TimerTask task,long delay,long period)；

范例：实现定时任务处理

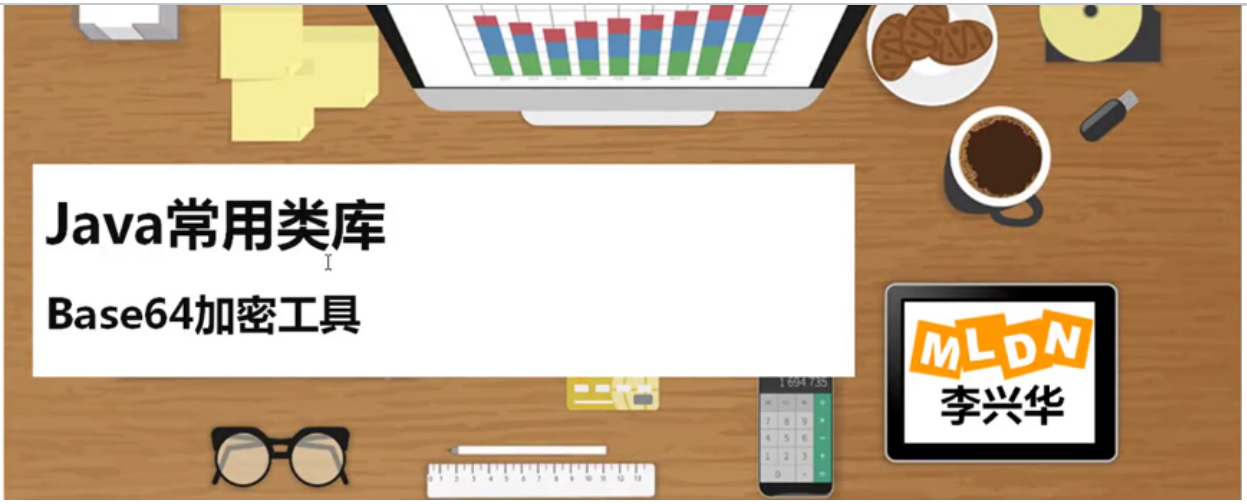
```
import java.util.Timer;
import java.util.TimerTask;

public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Timer timer = new Timer();	//定时任务
		timer.schedule(new MyTask(),1000);//延迟时间设置为0表示立即启动
	}
}

class MyTask extends TimerTask{	//任务主体

	@Override
	public void run() {	//多线程的处理方法
		System.out.println(Thread.currentThread().getName()+"、定时任务执行，当前时间"+System.currentTimeMillis());
	}
}
```

```
import java.util.Timer;
```

```
import java.util.TimerTask;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Timer timer = new Timer();	//定时任务
		//定时间隔任务，100毫秒后开始执行，每秒执行1次
		timer.schedule(new MyTask(),100,1000);
	}
}
class MyTask extends TimerTask{	//任务主体

	@Override
	public void run() {	//多线程的处理方法
		System.out.println(Thread.currentThread().getName()+"、定时任务执行，当前时间"
+System.currentTimeMillis());
	}
}
```



这种定时是由JDK最原始的方式提供的支持，但是实际上开发之中利用此类方式进行的定时处理实现的代码会非常的复杂。



2、具体内容

正常来讲加密基本上永远都要伴随着解密，所谓的加密或者是解密往往都需要有一些所谓的规则。在JDK1.8开始提供有一组新的加密处理操作类，Base64处理。

·Base64.Encoder：进行加密处理；

|- 加密处理：public byte[] encode(byte[] src)；

·Base64.Decoder：进行解密处理；

|- 解密处理：public byte[] decode(String src)；

范例：实现加密与解密操作

```
import java.util.Base64;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		String msg = "www.mldn.cn" ; // 要发送的信息内容
		String encMsg = new String(Base64.getEncoder().encode(msg.getBytes())) ; // 数据
加密
		System.out.println(encMsg);
		String oldMsg = new String(Base64.getDecoder().decode(encMsg));
		System.out.println(oldMsg);
	}
}
```

su虽然Base64可以实现加密与解密的处理，但是其由于是一个公版的算法，所以如果直接对数据进行加密往往并不安全，那么最好的做法是使用盐值操作。

```
import java.util.Base64;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		String salt = "mldnjava" ;
		String msg = "www.mldn.cn" + "{" + salt + "}"; // 要发送的信息内容
		String encMsg = new String(Base64.getEncoder().encode(msg.getBytes())) ; // 数据
加密
		System.out.println(encMsg);
		String oldMsg = new String(Base64.getDecoder().decode(encMsg));
		System.out.println(oldMsg);
	}
}
```

即便现在有盐值实际上发现加密的效果也不是很好，最好的做法是多次加密。

范例：复杂加密

```
import java.util.Base64;

class StringUtil {
	private static final String SALT = "mldnjava" ; // 公共的盐值
	private static final int REPEAT = 5 ; // 加密次数
	/**
	 * 加密处理
	 * @param str 要加密的字符串，需要与盐值整合
	 * @return 加密后的数据
	 */
	public static String encode(String str) {  // 加密处理
		String temp = str + "{" + SALT + "}" ; // 盐值对外不公布
```

```
            byte data [] = temp.getBytes() ; // 将字符串变为字节数组
            for (int x = 0 ; x < REPEAT ; x ++) {
                data = Base64.getEncoder().encode(data) ; // 重复加密
            }
            return new String(data) ;
        }
        /**
         * 进行解密处理
         * @param str 要解密的内容
         * @return 解密后的原始数据
         */
        public static String decode(String str) {
            byte data [] = str.getBytes() ;
            for (int x = 0 ; x < REPEAT ; x ++) {
                data = Base64.getDecoder().decode(data) ;
            }
            return new String(data).replaceAll("\\{\\w+\\}", "") ;
        }
}
public class JavaAPIDemo {
        public static void main(String[] args) throws Exception {
            String str = StringUtil.encode("www.mldn.cn") ;
            System.out.println(StringUtil.decode(str));
        }
}
```

最好的做法是使用2-3种加密程序，同时再找到一些完全不可解密的加密算法。