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

2、具体内容

在多线程的开发过程之中最为著名的案例就是生产者与消费者操作，该操作的主要流程如下：

·生产者负责信息内容的产生；

·每当生产者生产完成一项完整的信息之后消费者要从这里取走信息；

·如果生产者没有生产完则消费者要等待它生产完成，如果消费者还没有对信息进行消费，则生产者应该等待消费处理完成后再继续生产。

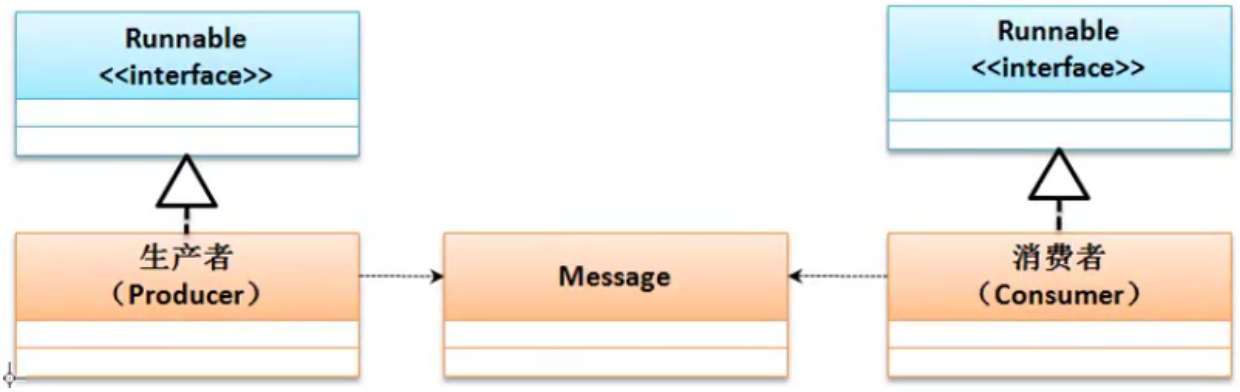
■程序的基本实现

可以将生产者与消费者定义为两个独立的线程类对象，但是对于现在生产的数据，可以使用如下的组成：

·数据一：title = 王建、content = 宇宙大帅哥；

·数据二：title = 小高、content = 猥琐第一人；

既然生产者与消费者是两个独立的线程，那么这两个独立的线程之间就需要有一个数据的保存集中点，那么可以单独定义一个Message类实现数据的保存。



范例：程序基本结构

```
public class ThreadDemo {
```

```
	public static void main(String[] args) throws Exception {
		Message msg = new Message() ;
		new Thread(new Producer(msg)).start();		// 启动生产者线程
		new Thread(new Consumer(msg)).start();		// 启动消费者线程
	}
}
class Producer implements Runnable {
	private Message msg ;
	public Producer(Message msg) {
		this.msg = msg ;
	}
	@Override
	public void run() {
		for (int x = 0 ; x < 100 ; x ++) {
			if (x % 2 == 0) {
				this.msg.setTitle("王健");
				try {
					Thread.sleep(100);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				this.msg.setContent("宇宙大帅哥");
			} else {
				this.msg.setTitle("小高");
				try {
					Thread.sleep(100);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				this.msg.setContent("猥琐第一人，常态保持。");
			}
		}
	}
}
class Consumer implements Runnable {
	private Message msg ;
	public Consumer(Message msg) {
		this.msg = msg ;
	}
	@Override
	public void run() {
		for (int x = 0 ; x < 100 ; x ++) {
			try {
				Thread.sleep(10);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println(this.msg.getTitle() + " - " + this.msg.getContent());
		}
	}

}
```

```
class Message {
	private String title ;
	private String content ;
	public void setContent(String content) {
		this.content = content;
	}
	public void setTitle(String title) {
		this.title = title;
	}
	public String getContent() {
		return content;
	}
	public String getTitle() {
		return title;
	}
}
```

通过整个代码的执行会发现此时有两个主要问题：

·问题一：数据不同步了；

·问题二：生产一个取走一个，但是发现有了重复生产和重复取出问题；

■解决数据同步

如果要解决问题，首先解决的就是数据同步的处理问题，如果要想解决数据同步最简单的做法是使用synchronized关键字定义同步代码块或同步方法，于是这个时候对于同步的处理就可以直接在Message类中完成。

范例：解决同步操作

```
class Message {
	private String title ;
	private String content ;
	public synchronized void set(String title,String content) {
		this.title = title ;
		try {
			Thread.sleep(100);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		this.content = content ;
	}
	public synchronized String get() {
		try {
			Thread.sleep(10);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		return this.title + " - " + this.content ;
	}
}
```

```
public class ThreadDemo {
	public static void main(String[] args) throws Exception {
		Message msg = new Message() ;
		new Thread(new Producer(msg)).start();		// 启动生产者线程
		new Thread(new Consumer(msg)).start();		// 启动消费者线程
	}
}
class Producer implements Runnable {
	private Message msg ;
	public Producer(Message msg) {
		this.msg = msg ;
	}
	@Override
	public void run() {
		for (int x = 0 ; x < 100 ; x ++) {
			if (x % 2 == 0) {
				this.msg.set("王健","宇宙大帅哥");
			} else {
				this.msg.set("小高","猥琐第一人，常态保持。");
			}
		}
	}
}
class Consumer implements Runnable {
	private Message msg ;
	public Consumer(Message msg) {
		this.msg = msg ;
	}
	@Override
	public void run() {
		for (int x = 0 ; x < 100 ; x ++) {
			System.out.println(this.msg.get());
		}
	}

}
```

在进行同步处理的时候肯定需要有一个同步的处理对象，那么此时肯定要将同步操作交由Message类处理是最合适的。这个时候发现数据以及可以正常的保持一致了，但是对于重复操作的问题依然存在。

■线程等待与唤醒

如果说现在要想解决生产者与消费者的问题，那么最好的解决方案就是使用等待与唤醒机制。而对于等待与唤醒操作机制主要依靠的是Object类中提供的方法处理的：

·等待机制

|- 死等：public final void wait() throws InterruptedException；

|- 设置等待时间：public final void wait(long timeout) throws InterruptedException；

|- 设置等待时间：public final void wait(long timeout,int nanos)throws InterruptedException；

·唤醒第一个等待线程：public final void notify()；

·唤醒全部等待线程：public final void notifyAll()；

如果此时有若干个等待线程的话，那么notify()表示的是唤醒第一个等待的，而其它的线程等待继续等待，而notifyAll()表示会唤醒所有等待的线程，那个线程的优先级高就有可能先执行。

对于当前的问题主要的解决应该通过Message类完成处理。

范例：修改Message类

```
class Message {
	private String title ;
	private String content ;
	private boolean flag = true ; // 表示生产或消费的形式
	// flag = true：允许生产，但是不允许消费
	// flag = false：允许消费，不允许生产
	public synchronized void set(String title,String content) {
		if (this.flag == false) {	// 无法进行生产，应该等待被消费
			try {
				super.wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		this.title = title ;
		try {
			Thread.sleep(100);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		this.content = content ;
		this.flag = false ; // 已经生产过了
		super.notify(); // 唤醒等待的线程
	}
	public synchronized String get() {
		if (this.flag == true) {	// 还未生产，需要等待
			try {
				super.wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		try {
			Thread.sleep(10);
		} catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
            try {
                return this.title + " - " + this.content ;
            } finally {  // 不管如何都要执行
                this.flag = true ; // 继续生产
                super.notify(); // 唤醒等待线程
            }
        }
}
public class ThreadDemo {
        public static void main(String[] args) throws Exception {
            Message msg = new Message() ;
            new Thread(new Producer(msg)).start();        // 启动生产者线程
            new Thread(new Consumer(msg)).start();       // 启动消费者线程
        }
}
class Producer implements Runnable {
        private Message msg ;
        public Producer(Message msg) {
            this.msg = msg ;
        }
        @Override
        public void run() {
            for (int x = 0 ; x < 100 ; x ++) {
                if (x % 2 == 0) {
                    this.msg.set("王健","宇宙大帅哥");
                } else {
                    this.msg.set("小高","猥琐第一人，常态保持。");
                }
            }
        }
}
class Consumer implements Runnable {
        private Message msg ;
        public Consumer(Message msg) {
            this.msg = msg ;
        }
        @Override
        public void run() {
            for (int x = 0 ; x < 100 ; x ++) {
                System.out.println(this.msg.get());
            }
        }

}
```

这种处理形式就是在进行多线程开发过程之中最原始的处理方案，整个的等待、同步、唤醒机制都由开发者自行通过原生代码实现控制。

同步问题解决