**·生成者与消费者程序的基本实现**

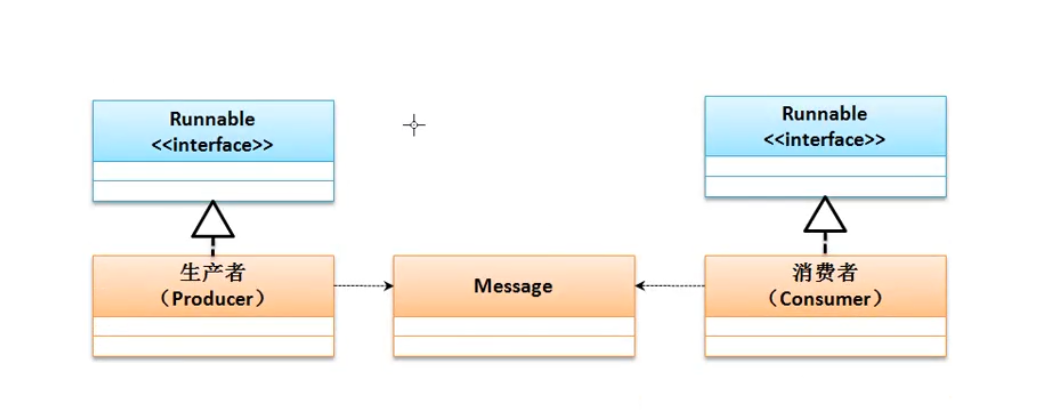
1. 可以将生产者与消费者定义为两个独立的线程类对象，但是对于现在生产的数据，可以使用如下的组成

·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集中的保存点，那么可以单独定义一个Message类实现

（2）程序基本模型



范例：程序基本结构

|  |
| --- |
| package com.company;  public class Main {   public static void main(String[] args) {  Message msg = new Message() ;  new Thread(new Producer(msg)).start() ; //启动生产者线程 => Start the producer thread  new Thread(new Consumer(msg)).start() ; //启动消费者线程 => Start the Consumer thread     } }  /\* 生产者继承多线程，它的主要任务是负责生产消息 The producer inherits multiple threads, and its main task is to produce messages \*/  class Producer implements Runnable {  private Message msg ;  public Producer (Message msg){  this.msg = msg ;  }  @Override  public void run() {  for (int i = 0; i < 100; i++) {  if (i % 2 == 0){  this.msg.setTitle("王建");  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，保持常态！");  }  }  } }   /\* \* 生产者继承多线程，他的主要任务是，显示数据 \* The producer inherits multithreading, and his main task is to display the message \* \*/  class Consumer implements Runnable{  private Message msg ;  public Consumer(Message msg){  this.msg = msg ;  }  @Override  public void run() {  for (int i = 0; i < 100; i++) {  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 String getTitle() {  return title;  }   public void setTitle(String title) {  this.title = title;  }   public String getContent() {  return content;  }   public void setContent(String content) {  this.content = content;  } } |

代码结果:

|  |
| --- |
| 王建--null  王建--null  王建--null  王建--null  王建--null  王建--null  王建--null  王建--null  王建--null  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  小高--宇宙大帅哥  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  王建--猥琐第一人，保持常态！  小高--宇宙大帅哥  Process finished with exit code 0 |

通过上述分析我们能发现两个主要问题:

·问题一：数据不同步;

·问题二：我们设计的初衷是生产一个取走一个，但是我们发现重复生产和重复取出的问题；

·**解决数据同步问题**

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定义我们的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

|  |
| --- |
| package com.company;  public class Main {   public static void main(String[] args) {  Message msg = new Message() ;  new Thread(new Producer(msg)).start() ; //启动生产者线程 => Start the producer thread  new Thread(new Consumer(msg)).start() ; //启动消费者线程 => Start the Consumer thread  } } /\* 生产者继承多线程，它的主要任务是负责生产消息 The producer inherits multiple threads, and its main task is to produce messages \*/ class Producer implements Runnable {  private Message msg ;  public Producer (Message msg){  this.msg = msg ;  }  @Override  public void run() {  for (int i = 0; i < 100; i++) {  if (i % 2 == 0){  this.msg.set("王建","宇宙大帅哥");  }else {  this.msg.set("小高","猥琐第一人，保持常态！");  }  }  } }  /\* \* 生产者继承多线程，他的主要任务是，显示数据 \* The producer inherits multithreading, and his main task is to display the message \* \*/ class Consumer implements Runnable{  private Message msg ;  public Consumer(Message msg){  this.msg = msg ;  }  @Override  public void run() {  for (int i = 0; i < 100; i++) {  try {  Thread.*sleep*(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(this.msg.get());  }  } } class Message {  private String title ;  private String content ;   public synchronized void set(String title , String content){  this.content = content ;  try {  Thread.*sleep*(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.title = title ;  }   public synchronized String get(){  try {  Thread.*sleep*(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  return this.title + "--" + this.content ;  } } |

此时我们发现，我们将同步问题交给message类进行处理，同时代码中的同步问题也通过synchronized 关键字解决了；

但是重复操作的问题还是没有解决；

而解决重复操作的问题通常用等待与唤醒的机制处理；

**·线程等待与唤醒**

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

等待机制:

不设置时间为死等，而wait中也可以设置等待的时间的参数：public final void wait() throws InterruptedException ;

唤醒机制:

public final void notify() 唤醒第一个等待的线程；

public final void notifyAll() 唤醒所有线程 ；

为了解决生产者与消费者的重复操作问题，我们修改message

范例：修改message

|  |
| --- |
| class Message {  private String title ;  private String content ;  private boolean flag = true ; //表示生产与消费的锁  //flag == true 允许生产不允许消费 为false 不能生产允许消费  public synchronized void set(String title , String content){  if (!flag){  try {  super.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  this.content = content ;  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.title = title ;  flag = false ;  super.notify();  }  public synchronized String get(){  if (flag){  try {  super.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  return this.title + "--" + this.content ;  } finally { //不管如何都要执行的  this.flag = true ;  super.notify();  }  }  } |