# 什么是线程，什么是进程

线程是进程的一个执行单元

# 容器

## collection

Collection是一个根接口，有一些子接口如List，Set等
add(E e)：向集合中添加一个元素。
remove(Object o)：从集合中移除指定元素。
contains(Object o)：判断集合是否包含指定元素。
size()：返回集合中元素的数量。
isEmpty()：判断集合是否为空。
clear()：移除集合中的所有元素。

### List

ArrayList是List的一个实现类，基于数组

```
List<String> list = new ArrayList<>();

    // 添加元素
    list.add("apple");
    list.add("banana");
    list.add("cherry");

    // 获取元素
    System.out.println("获取第一个元素: " + list.get(0));

    // 修改元素
    list.set(1, "orange");

    // 移除元素
    list.remove("cherry");
```

### Set

HashSet，基于哈希表实现

```
 Set<Integer> set = new HashSet<>();

    // 添加元素
    set.add(1);
```

## Iterator

用于遍历集合中的元素 hasNext()（判断是否还有下一个元素）
next()（返回下一个元素）
remove()（从集合中移除当前元素）

```
List<String> list = new ArrayList<>();
    list.add("one");
    list.add("two");
    list.add("three");

    Iterator<String> iterator = list.iterator();
    while (iterator.hasNext()) {
        String element = iterator.next();
        System.out.println(element);
        // 移除元素
        if ("two".equals(element)) {
            iterator.remove();
        }
    }
```

# 泛型

作用：对传入的参数类型进行限制，在编译阶段限制错误的发生
eg

```
List list=new ArrayList();
list.add("aaa");
list.add(123);
list.add(new student("张三"));
```

在这个例子中，由于对list存储的内容没有限制，在使用的时候就有可能产生错误
所以一般我们会

```
List<String> list = new ArrayList<>();
```

来将存入的内容限制为String或者其他类型。 在当我们不知都到要传入什么类型时，就可以用来代替具体类型

## 泛型类

格式

```
class Box<T> {
private T data;

public Box(T data) {
```

```
    this.data = data;
}

public T getData() {
    return data;
}

public void setData(T data) {
    this.data = data;
}
}
```

在我们使用的时候，就可以指定具体的类型

```
Box<Integer> integerBox = new Box<>(10);
```

### 泛型方法

格式

```
 public static <T> void printArray(T[] array) {}
```

### 泛型接口

格式

```
interface Generator<T> {
```

### 通配符

<?>：对元素类型没有限制
<? super T>：添加元素是 T 或其父类。
<? extends T>： 添加元素是T 或其子类。

```
public class Main {
public static void main(String[] args) {
    ArrayList<ye>list1=new ArrayList<>();
    ArrayList<fu>list2=new ArrayList<>();
    ArrayList<zi>list3=new ArrayList<>();

    method(list1);
    method(list2);
    method(list3);
}
public static void method(ArrayList<?>list){

}
}
class ye{}
class fu extends ye{}
class zi extends fu{}
```

要想将list1，list2，list3都可以传入method方法，还可以这样写

```
public static void method(ArrayList<? extends zi>list){}

public static void method(ArrayList<? super ye>list){}
```

# 多线程

### 创建线程

#### 继承Thread类

使用方法

```
class MyThread extends Thread {
@Override
public void run() {
    System.out.println("线程正在运行");
}
}

public class ThreadCreationExample {
public static void main(String[] args) {
    MyThread t1 = new MyThread();
    t1.start();
}
}
```

#### 实现 Runnable 接口

使用方法

```
class MyRunnable implements Runnable {
@Override
public void run() {
    System.out.println("线程正在运行");
}
}

public class RunnableExample {
public static void main(String[] args) {
    MyRunnable mr = new MyRunnable();
```

```
    Thread t = new Thread(myRunnable);
    t.start();
}
}
```

### 休眠

```
Thread.sleep(2000); // 休眠 2 秒
```

### 唤醒

wait() notify()

### 资源上锁

#### synchronized

同步方法

```
public synchronized void increment() {}
```

同步代码块

```
static Object lock = new Object();

synchronized (锁对象) {
        count++;
        System.out.println(Thread.currentThread().getName() + " 执行后，count 的值为: " + count);
    }
```

要记得创建锁对象
synchronized不便于控制开始和结束，所以还有一种上锁方法Lock

#### Lock

```
static Lock lock=new ReentrantLock();

lock.lock();
        count++;
        System.out.println(Thread.currentThread().getName() + " 执行后，count 的值为: " + count);
 lock.unlock();
```

# 1. 生产者与消费者模型

```
public class Main {
public static void main(String[]args){
    ArrayBlockingQueue <String>queue=new ArrayBlockingQueue<>(5);
    int total=0;
    int residue=0;

    Cook c=new Cook(queue);
    Foodie f1=new Foodie(queue);
    Foodie f2=new Foodie(queue);

    c.setName("厨师");
    f1.setName("顾客1");
    f2.setName("顾客2");

    c.start();
    f1.start();
    f2.start();
}

}

public class Cook extends Thread{
private ArrayBlockingQueue<String>queue;
public Cook(ArrayBlockingQueue<String>queue){
    this.queue=queue;
}

@Override
public void run(){
    while(true){
        try {
            Thread.sleep(500);
            queue.put("面条");
            System.out.println("厨师做了一碗面条");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

    }
}
}
public class Foodie extends Thread{
ArrayBlockingQueue<String> queue;
public Foodie(ArrayBlockingQueue<String>queue){
    this.queue=queue;
}

@Override
public void run(){
    while(true) {
```

```
        try {
            Thread.sleep(500);
            String food=queue.take();
            System.out.println(getName()+"吃了"+food);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
}
```

## 2. 攻击程序

```
package yic1;

class Player {
private int hp = 100;

public synchronized void beAttacked(String attackerName) {
    if (hp > 0) {
        System.out.println(attackerName + " attack...");
        hp -= 20;
        if (hp < 0) {
            hp = 0;
        }
        System.out.println(attackerName + ": 当前player的hp值= " + hp);
        if (hp == 0) {
            System.out.println(attackerName + ": player is dead.");
        }
    }
}
}

class Creep implements Runnable {
private String name;
private Player player;

public Creep(String name, Player player) {
    this.name = name;
    this.player = player;
}

@Override
public void run() {
    for (int i = 0; i < 3; i++) {
        player.beAttacked(name);
    }
    System.out.println(name + " end.");
}
}

public class Main {
public static void main(String[] args) {
    Player player = new Player();
    Thread creepA = new Thread(new Creep("Creep-A", player));
    Thread creepB = new Thread(new Creep("Creep-B", player));

    creepA.start();
    try {
        creepA.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    creepB.start();
  }
```

## 3. Java实现泛型链表

```
class Node<T> {
T data;
Node<T> next;

public Node(T data) {
    this.data = data;
    this.next = null;
}
}

// 定义泛型链表类
class GenericLinkedList<T> {
private Node<T> head;
private int size;

public GenericLinkedList() {
    this.head = null;
    this.size = 0;
}

// 添加结点
public boolean addList(Node<T> node) {
    if (head == null) {
        head = node;
    } else {
        Node<T> current = head;
        while (current.next != null) {
            current = current.next;
        }
```

```
        current.next = node;
    }
    size++;
    return true;
}

// 删除尾结点
public void removeList0() {
    if (head == null) {
        return;
    }
    if (head.next == null) {
        head = null;
    } else {
        Node<T> current = head;
        while (current.next.next != null) {
            current = current.next;
        }
        current.next = null;
    }
    size--;
}

// 根据节点值删除节点
public void removelistByValue(T value) {
    if (head == null) {
        return;
    }
    if (head.data.equals(value)) {
        head = head.next;
        size--;
        return;
    }
    Node<T> current = head;
    while (current.next != null && !current.next.data.equals(value)) {
        current = current.next;
    }
    if (current.next != null) {
        current.next = current.next.next;
        size--;
    }
}

// 找到值为 value 的结点，返回这个结点下标（下标从 0 开始计算）
public int find(T value) {
    Node<T> current = head;
    int index = 0;
    while (current != null) {
        if (current.data.equals(value)) {
            return index;
        }
        current = current.next;
        index++;
    }
    return -1;
}

// 获取链表大小
public int getSize() {
    return size;
}
}
}

    // 测试类
public class Main {
public static void main(String[] args) {
    GenericLinkedList<Integer> list = new GenericLinkedList<>();
    Node<Integer> node1 = new Node<>(1);
    Node<Integer> node2 = new Node<>(2);
    Node<Integer> node3 = new Node<>(3);

    // 添加结点
    list.addList(node1);
    list.addList(node2);
    list.addList(node3);

    System.out.println("链表大小: " + list.getSize());

    // 查找结点
    int index = list.find(2);
    System.out.println("值为 2 的结点下标: " + index);

    // 删除尾结点
    list.removeList0();
    System.out.println("删除尾结点后链表大小: " + list.getSize());

    // 根据值删除结点
    list.removelistByValue(1);
    System.out.println("删除值为 1 的结点后链表大小: " + list.getSize());
}
}
```