|  |  |  |
| --- | --- | --- |
| **语言** | **C++** | **Java** |
| **oops** | 类是对象的抽象，而对象是类的具体实例 (instance) | |
| **类的声明;**  **对象的定义;** | 1:声明一个类类型和声明一个结构体类型是相似的  声明了一个名为 Student的结构体类型  struct Student {  int num;  char name[20];  char sex;  };  Student stud1,stud2; //定义了两个结构体变量 stud1和stud2  2:定义类一般形式  class 类名  {private:  私有数据及方法;  protected:  公有数据及方法  public:  公有数据及方法  };  3:现在声明一个类  class Student  {  int num;  char name[20];  char sex;  void display(){  cout<<”num:”<<num<<endl;  }  }; | [public/final] class 类名  {  属性(Field,字段)  [修饰符] 类型 属性名 [=默认值];  修饰符：可省略,public/protected/private/static/final,  public/protected/private只出现其一  方法  [修饰符] 返回值类型 方法名(参数列表)  {  语句;  }  修饰符：可省略，public/protected/private/static/final/abstract,  public/protected/private只出现其一  final/abstract只出现其一  }  现在声明一个类  public class Student  {  int num;  char name[20];  char sex;  void display(){  cout<<”num:”<<num<<endl;  }  }; |
| **构造函数**  **及**  **实例对象** | 构造器：特殊的成员函数，用于实例成员变量的初始化，建立对象时自动执行，必须与类同名，系统会默认定义一个函数体为空，无参数的构造器，用户可以自己定义无参的构造器此时系统的就不存在。  Student stud1;//调用不带参构造器  可以定义带参数的构造函数，构造函数一般格式：  构造函数名（类型1 形参1，类型2形参2，…）{…};  Student stud1(100);//调用带参构造器 | 构造器  [修饰符] 构造器名(类名)(参数列表)  {  语句;  }  修饰符：可省略，public/protected/private其一  构造器名必须为类名，无返回值类型  默认构造器无参数  如程序员定义个一个以上构造器，则系统默认构造器不存在  通过new来调用构造器，返回该类实例的引用  Person P = new Person();  现在可以通过P这个实例访问其包含的方法和属性  P.属性 P.方法  static修饰的属性和方法也可以通过:  类.属性 类.方法  因为构造器主要用于被其他方法调用，所以通常设置为public访问权限  如果构造器A中的内容只是构造器B中的一部分，则可以在构造器B中第一条语句用this调用构造器A |
| **对象的动态建立** | class Box  {public:  int height;  int volume();  };  动态建立一个对象：**new Box**;//返回的是一个指向新建对象的指针  常用形式：  Box \*pt = new Box;  pt->height=1000;  pt->volume();  如果有带参的构造器，也可以如下用法：  Box \*pt = **new Box(12,15,18)**;//带三个int参数 | 类 是一种引用数据类型  数组是一种引用数据类型  实际上**java中的引用类型就是c中的指针**  Person P = new Person();**//C++中无括号**  Person P1 = P;//P P1指向同一个对象  当对象没有引用变量指向时，会自动释放  所以如果想释放某个对象的内存，将指向其的引用变了置NULL就可 |
| **对象的动态释放** | 释放由new建立的对象时，用delete运算符  Box \*pt = new Box;  delete pt;  备注：delete对象前会先调用对象的析构函数 | delete P; |
| **对象成员的引用;** | 三种方法：  通过对象名和成员运算符访问对象中的成员  stud1.num=1001;  stud1.display( );  通过指向对象的指针访问对象中的成员  Student \*p;  Student stud1;  p=&stud1;  p->num=1001;  p->display();  通过对象的引用变量来访问对象中的成员  Student stud1;  Student &stud2 = stud1;//stud2和stud1占用同一空间  stud2.num = 1001; | Student P是定义了一个对象引用也就是一个指针，并没有定义对象，这与C++定义对象有区别  Student P = new Person();  现在可以通过P这个实例访问其包含的方法和属性  P.属性 P.方法  static修饰的属性和方法也可以通过:  类.属性 类.方法 |
| **static修饰** | static可修饰属性和方法，叫类属性和类方法  表明修饰的是属于类共有的，不属于单个实例的  static修饰的成员不能访问没有static修饰的成员 | |
| **属性/方法修饰符** | public/protected/private叫做访问修饰符，其实还有一个默认的default  访问控制严格从小到大  **public：**  修饰的属性/方法可以被全局类访问  **protected:**  修饰的属性/方法可以被同一个包内的类访问+可被全局子类访问  **default：**  修饰的属性/方法可以被同一个包内的类访问  **private**  修饰的属性/方法可以被类的内部访问，外部不能访问，这适合对外隐藏属性 | |
| **类外定义方法;** | 首先在类内声明方法，其次在类外用作用域限定符：：，如  **Student::**display(){  cout<<”num:”<<num<<endl;  } | java中的方法不能独立存在，只能在类中定义 |
| **是否支持全局函数;** | 支持全局函数，不属于任何一个类 | 不支持 |
| **几个名词;** | 类的成员函数称为方法  类的成员变量成为属性 | |
| **函数重载;** | C++,java允许同一个类中定义多个同名方法,只要形参列表不同就可，这叫做方法的重载  **方法重载overload**只要求1：方法名相同 2：形参列表不同，至于方法返回值类型，修饰符等与重载无关 | |
| **参数初始化表初始化数据成员;** | 带参构造函数也可以不在函数体而直接在函数首部完成成员变量的初始化，定义构造器类似如下：  class Box  {  public:  Box(int h,int w,int len):height(h),width(w),length(len){};  private:  int height;  int width;  int length;  }; |  |
| **this成员变量;** | **this引用**  每个实例中都有一个this引用，类似：  Person this;  且this引用指向对象本身，相当于在构造器结束前将对象引用给了this  其最主要的作用是让类中的方法可以访问本类中的另外一个方法或属性，  因为调用一个方法或属性(非类方法/属性)必须要有实例，所以相当于方法的参数列表中有个隐含参数[类名 this],属性实际为this.属性  注意,static修饰的方法中无这个隐含的参数,static修饰的变量无this.前缀，因为没有实例时可以调用static方法/属性，如果有this参数或前缀会报错，所以static修饰的方法/属性访问不了非static的属性和方法 | |
| **const的使用** | 常对象：  Time const t1;// t1是常对象，其值在任何情况下都不能改变  常成员函数：  void Time::fun( ) const //fun是Time类中的常成员函数，不能修改本来中的数据成员 | **final修饰符**  可用于修饰类，变量，方法，表明修饰的类，变量，方法不可改变  final 变量：表明变量获得初始值后不可改变  变量为成员变量  final修饰的成员变量可在如下几个地方获取初始值  如果修饰的成员变量是类属性：静态初始化块，定义时  如果修饰的成员变量是实例属性：非静态初始化块，定义时，构造器  变量为局部变量  final修饰的局部变量可在如下几个地方获取初始值  定义时，定义未赋值后面只能赋一次值  注意：当final修引用类型的变量时，因为引用变量是地址，只能保证地址不会改变，对于地址的内容也就是对象是可以改变的，比如属性可以重新赋值  final 方法：表明方法不可被子类重写Override  可重载overload  final类：表明修饰的类不可有子类  其是最终类 |
| **对象的赋值** | 同一个类的对象可以互相赋值，所谓赋值只是针对数据成员的值  赋值使用=，一般形式：对象名1 = 对象名2，注意要属于同一个类  Student stud1,stud2;  stud2 = stud1; |  |
| **对象的复制** | 对象的复制：使用一个已有的对象快速复制出多个完全相同的对象  一般形式：  类名 对象2(对象1);  Box box2(box1);//使用box1对象克隆box2对象  这与实例对象的方法类似，其实这也是调用了特殊的构造器-复制构造器，这是系统默认做好的  还有另外一种复制形式  Box box2=box1;//这也是调用复制构造器，注意与赋值的区别，赋值是先实例了stud2,调用了基本的构造器了 |  |
| **友元** | 友元可以访问与其有好友关系的类中的私有成员，友元包括友元函数和友元类  友元函数：  在类定义中有如下：  class Box  {  public:  friend void display();//一个普通函数，不属于任何类  friend Time::display2();//Time类的成员函数  };  友元类：  class Box  {public:  friend B;//B类是友元类，B中所有函数都是Box的友元函数，可以访问Box中的任何成员  } |  |
| **类模板** | 1：类模板又叫参数化的类，一般形式如下：  template<typename T>  class Box  {public:  T \* display();  };  2：实例化对象一般形式如下：  Box<int> box1;  Box<int> box2(12,13);  3：类模板外定义成员函数一般形式如下：  template<typename T>  void Box<T>:: Box(){  …  } | **泛型**  由于java集合中一旦把一个对象丢进集合，集合就会忘记对象的类型，取出对象后腰强制类型转换，比较麻烦，所以java在JDK1.5增加泛型支持  类似于C++中的模版  public class Apple**<T>**  {  private **T** info;  public Apple(**T** info){ this.info = info;}  public **T**  getInfo(){…}  }  Apple**<String>** a1 = new Apple**<String>**(“apple”);  Apple**<Double>** a2 = new Apple**<Double>**(5.5);  上述定义了一个带泛型声明的Apple<T>类  创建带泛型声明的接口，父类后，可以为该接口创建实现类，或者派生出子类，此时需将**类型参数指定具体类型**  public class A extends Apple<T>{…}//错误  public class A extends Apple<String>{…}//正确 |
| **多个类型参数** | 类模板的参数类型可以多个，一般形式如下：  template<typename T，typename T2 >  class Box  {  };  类模板外定义成员函数一般形式如下：  template<typename T>  void Box<T, T2>:: Box(){  …  } |  |
| **类型参数默认值** | template<typename T，typename T2 **= int**>  class Box  {  }; |  |
| **类模版的深入--非类型参数** | 1：类模版除了使用类型参数还可以使用非类型参数(或叫表达式参数)  template<typename T,int n>  class Box  {};  非类型参数或表达式参数的限制：  1：模版代码不能修改非类型参数的值，也不能使用其地址  2：实例化时，表达式参数必须是常量表达式 |  |
| **类模版的深入—多功能性** | template<typename T>  class Array{  …  };  1：模版类可用作基类  template<typename Type>  class GrowArray:public Array<Type>  {…  };  2：模版类可用作组件类  template<typename Tp>  class Stack  {  Array<Tp> ar;  };  3：模版类可用作其他模版的类型参数  Stack< Array<int> > as;**//注意C++98要求至少一个空格将两个>分开** |  |
| **内部类为模版类** | template<typename T>  class beta  {  private:  **template<typename V>**  **class hold**  **{…};**  hold<T> q;  hold<int> n;  } |  |
|  |  |  |
| **运算符重载** | 运算符重载一般格式如下：  函数类型 operator运算符名称（形参列表）{  …  };  class Complex  {public:  Complex **operater+**(Complex &c2);  };  Complex Complex::operater+( Complex &c2)  {…  }  备注：  只能重载部分已有的运算符 |  |
| **继承与派生** | 假设声明了一个基类Student,在此基础上通过单继承建立一个Student1派生类  class Student1 : public Student  {…};  一般格式如下：  class 派生类：[继承方式] 基类  {  派生类新增成员  }；  继承方式有public/private/protected,不写则默认位private  派生类的成员包括基类的成员和自己新增的成员，但是构造器和析构器不能从基类继承，派生类的构造器不仅要考虑对新增成员的初始化也要考虑对基类数据成员的初始化，**实例派生类对象时，系统执行派生类构造器时，先调用基类构造函数**，如Student1构造器如下：  class Student  {  public:  Student(int n,string nam,char s):num(n),name(nam),sex(s){};  private:  int num;  string name;  char sex;  }  class Student1:public Student  {public:  Student1(int n,string name,char s,int a,string ad):Student(n,name,s),age(a),addr(ad){};**// 使用Student中带参数构造器**  Student1(int a,string ad): age(a),addr(ad){}; **//使用Student中默认的无参数构造器**  private：  int age;  string addr;  } | java继承具有单继承特点，只有最多一个直接父类  用extends关键字实现继承，格式如下  [修饰符] class subclass extends SuperClass  {…}  子类获取父类全部的属性和方法，除父类构造器  如果一个类没有显式的直接父类，则默认继承java.lang.Object类，因此java.lang.Object类是所有类的父类，要么是直接父类，要么是间接父类  子类继承父类后，一方面增加新的属性和方法，一方面重写继承的方法，重写并不影响父类的方法，只是子类增加了与父类同名的方法，这叫做**方法重写**或**方法覆盖（Override）**,注意，方法覆盖需要遵循  1：方法名，参数列表，是否是static,相同  2：子类方法返回值比父类返回值类型要更小或相等  3：子类方法的访问权限要比父类方法更大或相等  子类覆盖父类方法/属性后，无法直接访问父类方法/属性，可以使用**super.方法**/**super****.属性**访问父类方法/属性  如果父类的方法是private访问权限的，子类定义一个相同名称，参数，是否static的方法，也不是覆盖，只是子类新增的方法而已  **super引用**  java在创建类时，系统会隐式创建其直接父类，直接父类又会创建其父类，直到创建java.lang.Object对象  **每个类中都隐含定义了一个直接父类的引用****super**,子类创建实例时，首先创建直接父类的实例，并将直接父类对象引用给了super  有时需要在子类构造器中执行父类构造器的初始化代码(并不是创建一个父类对象)，这是可在子类构造器第一行代码中使用super.构造器来实现 |
| **多重继承** | class D:public A,private B,protected C  {  ….  };  多重继承会导致二义性  1:    则 C c1;  c1.a=3;//编译报错，可以使用c1.A::a=3;  c1.display();//编译报错,可以使用c1.A::display();  2:  当类A,B都有一个Z基类，则这个基类被C继承了2次，这样派生类就有Z基类的成员变量和方法各两次，为此C++提供虚基类，使得继承间接共同基类只保留共同基类的一份成员  class 派生类名::virtual 继承方式 基类名  经过这样的声明后，当基类通过多条派生路径被一个派生类继承时，该派生类只继承该基类一次 |  |
| **基类与派生类的转换** | 1：派生类对象可以向基类对象赋值  A base;//基类  B sub;  base = sub;//丢弃派生类自己的成员  2：不可用基类对象给派生类对象赋值  3：派生类对象可以替代基类对象向基类对象的引用进行赋值或初始化  4：如果函数的参数是基类对象或基类对象的引用，相应的实参可以用子类对象 |  |
| **虚函数** | 类的继承层次中，不同层次可以出现名字相同，参数个数类型顺序相同的函数，这个叫做函数的重写，此时编译系统更加对象来调用  class Base{  public:  void display(){};  };  class sub:public Base  {  public:  void display(){};  };  sub B;  B.display();//调用sub中的display();  Base A;  A.display();//调用Base中的display();  或者使用基类指针  Base \*pt = new Base;  pt->display();//调用Base中的display();  Base \*pt = new sub;  pt->display();//调用Base中的display();  这样总是达不到用同一种形式一会调用父类中的display()一会调用子类中的display()  1：于是C++中规定可以将基类中的display声明为虚函数  virtual void display();  2：C++规定当一个成员函数被声明为虚函数后，其派生类中同名函数且参数个数顺序类型相同自动为虚函数  3：定义一个指向基类对象的指针变量，如果基类对象赋给这个指针，则调用的display时基类的，如果子类对象赋给这个指针，则调用的display是子类的 | **无** |
| **纯虚函数** | 纯虚函数无函数体，一般是向让子类去重写的,一般形式  **virtual 函数类型 函数名(参数列表)=0;**  virtual float area() =0; | **无** |
| **抽象方法** | **无** | 为了实现多态，也就是父类的引用调用其子类的方法，有时为简洁引入抽象方法  也就是父类中只包含方法签名，无方法实现，也就是没有方法体，这**相当于C++中的纯虚函数**  定义抽象方法只需要在普通方法上增加abstract修饰符并去掉方法体  如public abstract void test(); |
| **抽象类** | 包含纯虚函数的类都是抽象类，由于纯虚函数不能调用，所以包含纯虚函数的类无法建立对象，也就是抽象类不能建立对象 | 抽象类不能实例对象，只能被继承  抽象方法和抽象类必须使用abstract修饰符修饰，抽象方法的类只能定义成抽象类，抽象类可以没有抽象方法  **抽象类不能被创建实例**  定义抽象类只需要在类上增加abstract修饰符即可，抽象类不能实例对象  类BassClass,其子类SubClass  定义一个BassClass的引用变量 BassClass p;  这p的编译类型为BassClass引用  因为SubClass为BassClass的子类，所以可以将SubClass的对象引用给p  p = new SubClass();  则p的运行时类型为SubClass  p只能调用编译时类型的方法和属性，比如BassClass中有方法A ，SubClass中新增了方法B  则p.A正确，p.B报错  如果方法A在SubClass中被覆盖override ,则p.A调用的是SubClass中的A方法，这便是多态  简言之也就是用父类引用调用其子类的方法  属性不具有多态性，直接使用编译时类型，如BassClass中有bass=1,SubClass中有bass=2,则p.bass是1 |
| **接口** | **无** | **接口：更彻底的抽象**  接口中只有抽象方法，可有常量定义  **[修饰符] interface 接口名 extends 父接口1，父接口2**  **{…}**  接口可以有多个直接父接口，但是不能继承类  接口中不包含构造器，初始化块等，只可以有  **常量属性**(系统自动加上static和final修饰符)  **抽象方法**(系统自动加上public abstract修饰符)  **内部类(**系统自动加上public abstract修饰符)  **枚举定义(**系统自动加上public abstract修饰符)  使用接口  一个类可以实现一个或多个接口，实现使用implements关键字  **[修饰符] class 类名 extends 父类 implements 接口1，接口2**  **{…}**  implements 必须放于extends 后  接口不能显示继承任何类，单接口类型的引用变了可以直接赋给Object类型的引用变量 |
| **命名空间** | 当一个文件包含include的另外几个文件中有同名的全局变量或同名的类等，则编译发生报错，为了解决这个问题引入了命名空间的概念，命名空间是一个内存区域，可以把一些全局的类等放入不同的命名空间，同时包含时就不会冲突，指定命名空间形式如下：  namespace ns1//指定命名空间ns1  {  int a;  double b;  }  程序中使用a,b要用命名空间限定  **ns1::a**  **ns1::b**  这样显得比较累赘，于是可以在最上面加上  **using st1::a;**  **using st1::b;**  则后面用的a,b就是st1命名空间中的  或者  **using namespace ns1;**  则后面用的a,b就是st1命名空间中的 | 实际就是作用域，两个包中可以包含同名的类，同一个包内不能有重名类  实际上会将包名加到类名前面去  指定类所处于的包中，只需要在源文件最前面加上，这个源文件所有类都属于这个包  package packageName;  位于包中的每个类的完整类名为包名+类名的组合,如文件Hello.java  package lee.sub;  public class Hello{  }  javac –d . Hello.java  会在当前目录生成 ./lee/sub/Hello.class  进入lee，可执行java lee.sub.Hello //注意执行java Hello是错误的  另外一个包内的类想创建一个Hallo类实例，需要加上包名  lee.sub.Hello p = new lee.sub.Hello();  鉴于此，我们应当将java源文件放在与包名一致的目录结构中  一个源文件中只能指定一个包，也就是一条package语句  由于使用不同包内的类需要加上包名比较繁琐，所以引入import关键字，import关键字相当于c中的include，将包中定义的类部分或全部包含到当前源文件中来，import出现在package之后，这些包含进来的类也会成为当前包中的类  java默认在所有源文件中导入java.lang包下的所有类，都是常用类  导入指定类的单个静态属性如下  import static 包名.类名.属性名  导入指定类的全部静态属性如下  import static 包名.类名.\* |
| **内部类** |  | **内部类**  可以将类放于另外一个类的内部定义，这叫做内部类，包含内部类的类叫外部类  没有static修饰的叫非静态内部类  非静态内部类成员可以直接访问外部类的私有数据，因为非静态内部类是外部类的成员，但外部类不能访问非静态内部类的属性和方法,因为外部类创建对象时，非静态内部类对象根本不存在  如果外部类想访问非静态内部类的成员，则要显示创建非静态内部类对象  匿名内部类使用与只需要一次使用的类  static修饰符叫静态内部类  这个内部类是外部类的，不是外部类实例的  静态内部类可以包含静态和非静态成员，静态内部类相当于外部类的一个静态成员，不能访问外部类的非静态成员  使用内部类  内部类的访问修饰符决定其访问权限  default则内部类可被与外部类同一包内的类访问  protected则内部类可被与外部类同一包内的类访问+子类访问  public则内部类可被全局访问  注意在外部类外使用内部类时要将类名加全  外部类.内部类  因为非静态内部类的对象必须寄存在外部类对象中，所以创建非静态内部类对象前先创建外部类对象，并用new调用内部类  外部类对象引用.new 内部类构造器  在外部类中使用静态内部类，不需要创建外部类对象  new 外部类名.内部类名 |
| **匿名内部类** |  | **匿名内部类**  创建那种只使用一次的类，创建匿名内部类时立即创建一个该类实例，创建格式如下  new **父构造器(实参列表)** | **接口** ()  {  实现接口或重写父类方法/属性  }  创建匿名内部类时，必须实现接口或抽象父类中所有的抽象方法，因为要创建实例了  如果匿名内部类要访问外部类的局部变量，需要使用final修饰符修饰外部类局部变量，否则报错  匿名内部类必须继承一个父类或实现一个接口,且最多只能继承一个父类或实现一个接口  匿名内部类没有类名，所以无构造器，但是可以有初始化块来实现初始化  interface Product  {…}  创建匿名内部类 test( **new Product(){**  **实现接口或重写父类方法/属性**  **}** ); //test(Product p) |
| **线程** |  | **线程**  java使用Thread类代表线程，所有线程对象都必须是Thread类或其子类的实例  每个线程的任务是完成一定的任务，实际就是一段程序，java中用run方法封装这段程序流  **通过继承Thread类创建线程类**  创建并启动一个线程步骤如下：  1：定义一个Thread类的子类，并重写run方法  2：创建Thread子类的实例，也就是线程对象  3：用线程对象的start方法启动该线程对象  public class FirstThread extends Thread  {  public void run(){…}  }  new FirstThread().start();//启动第一条线程  new FirstThread().start();//启动第二条线程  线程对象有两个方法  Thread.currentThread();//静态方法，返回当前正在执行的线程对象  getName();返回调用该方法的线程名字  可以通过setName(String name)设置线程的名字，一般主线程名字为main,用户启动的线程名依次为Thread-0 Thread-1 …  **实现Runnable接口创建线程类**  创建并启动一个线程步骤如下：  1：定义Runnable接口的实现类，重写run方法  2：创建Runnable实现类的实例，并以此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象  3：调用线程对象的start方法来启动线程  SecondThread st = new SecondThread();//创建Runnable实现类的对象  new Thread(st).start();  Runable对象只是作为Thread对象的target，主要是Runnable实现类中的run方法，实际线程仍然为Thread实例  两种方式的对比  采用Runnable方式下，多个线程共享同一个target对象，适合多线程处理一份资源的情况  **线程的生命周期**  线程生命周期有新建，就绪，运行，阻塞，死亡五种状态  当调用start()方法后，该线程处于就绪状态，至于何时运行，看系统调度  当run方法执行完成，线程正常结束，或直接调用线程的stop()方法  **控制线程**  Thread提供让一个线程等待另外一个线程完成的方法：joib()  当某个程序执行流中调用了线程A.join方法，则调用处阻塞，直到线程A完成为止  **改变线程的优先级**  每个线程都有执行时都有一定的优先级，优先级高的执行的机会较多  每个线程的默认优先级与创建它的父线程相同，main具有普通优先级6  Thread提供了setPriority(int newPriority)和getPriority()方法来设置和返回指定线程的优先级，newPriority的值范围是1-10之间，java提供三个宏表示三个优先级  MAX\_PRIORITY 值为10  MIN\_PRIORITY 值为1  NORM\_PRIORITY 值为5  **线程同步**  两个线程可能修改同一个临界资源，此时需要线程同步  java多线程引入了同步监视器来解决这个问题，使用同步监视器的通用方法就是同步代码块，语法如下  synchronized(obj) //**obj就是同步监视器**  {  //同步代码块，也就是临界区  }  任何时刻只有一个线程可以获得同步监视器的锁定，同步代码块执行结束后，该 线程释放对该同步监视器的锁得  虽然java允许任何对象作为同步监视器，但建议用临界资源充当同步监视器  JDK1.5之后，java提供了**同步锁Lock对象**来实现线程间的同步 |
| **线程相关类** |  | **线程相关的类之ThreadLocal**  java为线程安全提供了一些工具类，如ThreadLocal类，它代表一个**线程的局部变量**，通过把数据放在ThreadLocal中就可以让每条线程创建一个该变量的副本，从而 避免并发访问的线程安全问题  ThreadLocal类  通过ThreadLocal类可以简化 多线程编程的并发访问，JDK1.5后为该类增加了泛型支持ThreadLocal<T>  ThreadLocal是将临界的资源复制出多份，每个线程都有一个，每个线程修改资源并不会影响到其他线程的资源，这这主要用到临界的资源不需要全局一致的情况下，比如对于同一个账户，两个线程同时执行取现操作，用ThreadLocal后可以得到对于每个线程账户对象的名字属性可以不同，但对于账户余额这种全局一致的属性则不能用ThreadLocal  class Account  {  prviate ThreadLocal<String> name = **new ThreadLocal<String>();**  public Account(String str){**name.set(str);**}  public void setName(String str){**name.set(srt);**}  }  class Test extends Thread  {  private Account account;  public Test(Account account,String name){ super(name); account = account; }  public void run(){ **account.setName(getName());** }//account对象中德name值设置为线程的名字  }  Account at = new Account(“初识账户名”);**//账户建立时创建这个线程局部变量，有几个线程用到Account这个临界资源，则这个局部变量对象name就复制几分，每个线程独有自己的name，每个线程都可以去改变Account中的name值，获取name值用name.get(),设置name值用name.set(xxx);** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |