20180321_抽象类和接口的区别

抽象类：

1- 主要的出发点是为了继承，可以提供多态

2- 抽象类还是有用重构器，可以将　公共方法沿着继承层次向上移动

接口：

1- 比抽象类更加抽象的“类”

2- 接口本身就不是类，从我们不能实例化一个接口就可以看出。如new Runnable();肯定是错误的，我们只能new它的实现类。

3- 建立类与类之间的协议，它所提供的只是一种形式，而没有具体的实现。同时实现该接口的实现类必须要实现该接口的所有方法，通过使用implements关键字[关键字是不同的]

4- 接口弥补了抽象类不能多重继承的缺陷

5- Interface的方所有法访问权限自动被声明为public。确切的说只能为public。(声明为其它的会报错)

6- 接口中的变量：接口中可以定义“成员变量”，或者说是不可变的常量，因为接口中的“成员变量”会自动变为为public static final。可以通过类命名直接访问：ImplementClass.name。

7- 接口中不存在实现的方法

8- 实现接口的非抽象类必须要实现该接口的所有方法。抽象类可以不用实现

9- 不能使用new操作符实例化一个接口，但可以声明一个接口变量，该变量必须引用（refer to)一个实现该接口的类的对象。可以使用 instanceof 检查一个对象是否实现了某个特定的接口。例如：if(anObject instanceof Comparable){}。　[接口不能实例化，但是可用 instanceof 进行检查]

10- 实现多接口的时候一定要避免方法名的重复。

11-

区别：

1- 抽象类可以拥有不同类型的数据。但是接口只能拥有 static final 类型的数据

2- 抽象层次不同。抽象类是对类抽象，而接口是对行为的抽象。

3- 跨域不同。抽象类所跨域的是具有相似特点的类，而接口却可以跨域不同的类。抽象类所体现的是一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"is-a" 关系，即父类和派生类在概念本质上应该是相同的。对于接口则不然，并不要求接口的实现者和接口定义在概念本质上是一致的，　仅仅是实现了接口定义的契约而已。

4- 设计层次不同。对于抽象类而言，它是自下而上来设计的，我们要先知道子类才能抽象出父类，而接口则不同，它根本就不需要知道子类的存在，只需要定义一个规则即可，至于什么子类、什么时候怎么实现它一概不知。

抽象类是自底向上抽象而来的，接口是自顶向下设计出来的。

5- 一个子类只能存在一个父类，但是可以存在多个接口

6- 在抽象类中可以拥有自己的成员变量和非抽象类方法，但是接口中只能存在静态的不可变的成员数据（不过一般都不在接口中定义成员数据），而且它的所有方法都是抽象的

7- 抽象类所代表的是“is-a”的关系，而接口所代表的是“like-a”的关系。

```
联系：
    1- 用继承确定本质，用接口扩展功能。


扩展：
   ISP（Interface Segregation Principle）：面向对象的一个核心原则。它表明使用多个专门的接口比使用单一的总接口要好。[尽量使接口分离]
```


1- 抽象类

以表征抽象概念的抽象类是不能实例化的。

同时，抽象类体现了数据抽象的思想，是***实现多态的一种机制***。它定义了一组抽象的方法，至于这组抽象方法的具体表现形式有派生类来实现。同时抽象类提供了继承的概念，它的**出发点就是为了继承，否则它没有存在的任何意义。所以说定义的抽象类一定是用来继承的，同时在一个以抽象类为节点的继承关系等级链中，叶子节点一定是具体的实现类。**（不知这样理解是否有错!!!高手指点....）

1、抽象类不能被实例化，实例化的工作应该交由它的子类来完成，它只需要有一个引用即可。

2、抽象方法必须由子类来进行重写。

3、只要包含一个抽象方法的抽象类，该方法必须要定义成抽象类，不管是否还包含有其他方法。

4、抽象类中可以包含具体的方法，当然也可以不包含抽象方法。

5、子类中的抽象方法不能与父类的抽象方法同名。

6、abstract不能与final并列修饰同一个类。

7、abstract 不能与private、static、final或native并列修饰同一个方法。、

实例：

定义一个抽象动物类Animal，提供抽象方法叫cry()，猫、狗都是动物类的子类，由于cry()为抽象方法，所以Cat、Dog必须要实现cry()方法。如下：

```
public abstract class Animal {
```

```
    public abstract void cry();
}

public class Cat extends Animal{

    @Override
    public void cry() {
        System.out.println("猫叫：喵喵...");
    }
}

public class Dog extends Animal{

    @Override
    public void cry() {
        System.out.println("狗叫:汪汪...");
    }

}

public class Test {

    public static void main(String[] args) {
        Animal a1 = new Cat();
        Animal a2 = new Dog();

        a1.cry();
        a2.cry();
    }
}

----------------------------------------------------------------------
Output:
猫叫：喵喵...
狗叫:汪汪...
```

创建抽象类和抽象方法非常有用,因为他们可以使类的抽象性明确起来,并告诉用户和编译器打算怎样使用他们.**抽象类还是有用的重构器,**因为它们使我们可以很容易地将公共方法沿着继承层次结构向上移动。

2- 接口

接口是一种**比抽象类更加抽象的“类”**。这里给“类”加引号是我找不到更好的词来表示，但是我们要明确一点就是，**接口本身就不是类**，从我们**不能实例化一个接口就可以看出。如new Runnable();肯定是错误的，我们只能new它的实现类。**

接口是用来**建立类与类之间的协议，它所提供的只是一种形式，而没有具体的实现。**同时实现该接口的实现类必须要实现该接口的所有方法，通过使用implements关键字，他表示该类在遵循某个或某组特定的接口，同时也表示着“interface只是它的外貌，但是现在需要声明它是如何工作的”。

接口是抽象类的延伸，java了保证数据安全是不能多重继承的，也就是说继承只能存在一个父类，但是接口不同，一个类可以同时实现多个接口，不管这些接口之间有没有关系，所**以接口弥补了抽象类不能多重继承的缺陷，**但是推荐继承和接口共同使用，因为这样既可以保证数据安全性又可以实现多重继承。

在使用接口过程中需要注意如下几个问题：

1、个Interface的方所有法访问权限自动被声明为public。确切的说只能为public，当然你可以显示的声明为protected、private，但是编译会出错！

2、接口中可以定义“成员变量”，或者说是不可变的常量，因为接口中的“成员变量”会自动变为为public static final。可以通过类命名直接访问：ImplementClass.name。

3、接口中不存在实现的方法。

4、实现接口的非抽象类必须要实现该接口的所有方法。抽象类可以不用实现。

5、不能使用new操作符实例化一个接口，但可以声明一个接口变量，该变量必须引用（refer to)一个实现该接口的类的对象。可以使用 instanceof 检查一个对象是否实现了某个特定的接口。例如：if(anObject instanceof Comparable){}。

6、在实现多接口的时候一定要避免方法名的重复。

3- 抽象类与接口的区别

尽管抽象类和接口之间存在较大的相同点，甚至有时候还可以互换，但这样并不能弥补他们之间的差异之处。下面将从语法层次和设计层次两个方面对抽象类和接口进行阐述。

3.1语法层次

在语法层次，java语言对于抽象类和接口分别给出了不同的定义。下面已Demo类来说明他们之间的不同之处。

使用抽象类来实现:

```
public abstract class Demo {
    abstract void method1();


    void method2(){
        //实现
    }
}
```

使用接口来实现

```
interface Demo {
    void method1();
    void method2();
}
```

抽象类方式中，**抽象类可以拥有任意范围的成员数据，同时也可以拥有自己的非抽象方法，但是接口方式中，它仅能够有静态、不能修改的成员数据**（但是我们一般是不会在接口中使用成员数据），同时它所有的方法都必须是抽象的。在某种程度上来说，**接口是抽象类的特殊化。**

对子类而言，它只能继承一个抽象类（这是java为了数据安全而考虑的），但是却可以实现多个接口。

3.2设计层次

上面只是从语法层次和编程角度来区分它们之间的关系，这些都是低层次的，要真正使用好抽象类和接口，我们就必须要从较高层次来区分了。只有从设计理念的角度才能看出它们的本质所在。一般来说他们存在如下三个不同点：

1、 **抽象层次不同。抽象类是对类抽象，而接口是对行为的抽象。**抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。

2、 ***跨域不同。抽象类所跨域的是具有相似特点的类，而接口却可以跨域不同的类。***我们知道抽象类是从子类中发现公共部分，然后泛化成抽象类，子类继承该父类即可，但是接口不同。实现它的子类可以不存在任何关系，共同之处。例如猫、狗可以抽象成一个动物类抽象类，具备叫的方法。鸟、飞机可以实现飞Fly接口，具备飞的行为，这里我们总不能将鸟、飞机共用一个父类吧！所以说抽***象类所体现的是一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"is-a" 关系，即父类和派生类在概念本质上应该是相同的。对于接口则不然，并不要求接口的实现者和接口定义在概念本质上是一致的，仅仅是实现了接口定义的契约而已。***

3、 ***设计层次不同。对于抽象类而言，它是自下而上来设计的，我们要先知道子类才能抽象出父类，而接口则不同，它根本就不需要知道子类的存在，只需要定义一个规则即可，至于什么子类、什么时候怎么实现它一概不知。比***如我们只有一个猫类在这里，如果你这是就抽象成一个动物类，是不是设计有点儿过度？我们起码要有两个动物类，猫、狗在这里，我们在抽象他们的共同点形成动物抽象类吧！所以说抽象类往往都是通过重构而来的！但是接口就不同，比如说飞，我们根本就不知道会有什么东西来实现这个飞接口，怎么实现也不得而知，我们要做的就是事前定义好飞的行为接口。所以说***抽象类是自底向上抽象而来的，接口是自顶向下设计出来的。***

（上面纯属个人见解，如有出入、错误之处，望各位指点！！！！）

为了更好的阐述他们之间的区别，下面将使用一个例子来说明。该例子引自：http://blog.csdn.net/ttgjz/article/details/2960451

我们有一个Door的抽象概念，它具备两个行为open()和close()，此时我们可以定义通过抽象类和接口来定义这个抽象概念：

抽象类：

```
abstract class Door{
    abstract void open();
    abstract void close();
}
```

接口

```
interface Door{
    void open();
    void close();
}
```

至于其他的具体类可以通过使用extends使用抽象类方式定义Door或者Implements使用接口方式定义Door，这里发现两者并没有什么很大的差异。

但是现在如果我们需要门具有报警的功能，那么该如何实现呢?

解决方案一：给Door增加一个报警方法:clarm();

```
abstract class Door{
    abstract void open();
    abstract void close();
    abstract void alarm();
}
```

或者

```
interface Door{
    void open();
    void close();
    void alarm();
}
```

这种方法违反了面向对象设计中的一个核心原则 ISP (Interface Segregation Principle)—见批注，在Door的定义中把**Door概念本身固有的行为方法和另外一个概念"报警器"的行为方 法混在了一起。这样引起的一个问题是那些仅仅依赖于Door这个概念的模块会因为"报警器"这个概念的改变而改变**，反之依然。

解决方案二

既然open()、close()和alarm()属于两个不同的概念，那么我们依据ISP原则将它们分开定义在***两个代表两个不同概念的抽象类里***面，定义的方式有三种：

1、两个都使用抽象类来定义。

2、两个都使用接口来定义。

3、一个使用抽象类定义，一个是用接口定义。

由于java不支持多继承所以第一种是不可行的。后面两种都是可行的，但是选择何种就反映了你对问题域本质的理解。

如果选择第二种都是接口来定义，那么就反映了两个问题：1、我们可能没有理解清楚问题域，AlarmDoor在概念本质上到底是门还报警器。2、如果我们对问题域的理解没有问题，比如我们在分析时确定了AlarmDoor在本质上概念是一致的，那么我们在设计时就没有正确的反映出我们的设计意图。因为**你使用了两个接口来进行定义，他们概念的定义并不能够反映上述含义。**

第三种，如果我们对问题域的理解是这样的：**AlarmDoor本质上Door，但同时它也拥有报警的行为功能，这个时候我们使用第三种方案恰好可以阐述我们的设计意图。AlarmDoor本质是们，所以对于这个概念我们使用抽象类来定义，同时AlarmDoor具备报警功能，说明它能够完成报警概念中定义的行为功能，所以alarm可以使用接口来进行定义**。如下：

```
abstract class Door{
    abstract void open();
    abstract void close();
}

interface Alarm{
    void alarm();
}

class AlarmDoor extends Door implements Alarm{
    void open(){}
    void close(){}
    void alarm(){}
}
```

用继承确定本质，用接口扩展功能。

这种实现方式基本上能够明确的反映出我们对于问题领域的理解，正确的揭示我们的设计意图。其实抽象类表示的是"is-a"关系，接口表示的是"like-a"关系，大家在选择时可以作为一个依据，当然这是建立在对问题领域的理解上的，比如：如果我们认为AlarmDoor在概念本质上是报警器，同时又具有Door的功能，那么上述的定义方式就要反过来了。

批注：

> ISP（Interface Segregation Principle）：面向对象的一个核心原则。它表明使用多个专门的接口比使用单一的总接口要好。
>
> 一个类对另外一个类的依赖性应当是建立在最小的接口上的。
>
> 一个接口代表一个角色，不应当将不同的角色都交给一个接口。没有关系的接口合并在一起，形成一个臃肿的大接口，这是对角色和接口的污染。

4-总结

1、 抽象类在java语言中所表示的是一种继承关系，***一个子类只能存在一个父类，但是可以存在多个接口***。

2、 在抽象类中可**以拥有自己的成员变量和非抽象类方法**，但是接口中**只能存在静态的不可变的成员数据（不过一般都不在接口中定义成员数据），而且它的所有方法都是抽象的。**

3、抽象类和接口所反映的设计理念是不同的，**抽象类所代表的是“is-a”的关系，而接口所代表的是“like-a”的关系。**

抽象类和接口是java语言中两种不同的抽象概念，他们的存**在对多态提供了非常好的支持，虽然他们之间存在很大的相似性。但是对于他们的选择往往反应了您对问题域的理解**。只有对问题域的本质有良好的理解，才能做出正确、合理的设计。