向对象开发总结

本节目标

> 1. 包的定义及使用、访问控制权限
> 2. 单例设计模式&多例设计模式
> 3. Java异常与捕获
> 4. JavaSE面向对象实战之Java链表

1.包的定义及使用

包的本质实际上就属于一个文件夹。在项目开发中很难避免类名称重复的问题。如果所有的java文件都放在一个文件夹中，就有可能存在覆盖问题。

1.1 包的定义

在.java 文件首行使用package 包名称; 即可

范例：定义包

```
package www.bit.java ;
public class Test {
    public static void main(String[] args) {
        System.out.println("Hello World") ;
    }
}
```

一旦程序出现包名称，那么*.class必须存在相应目录下。在JDK编译的时候使用配置参数。

打包编译命令： javac -d . 类.java

> 1. -d：表示生成目录，根据package的定义生成。
> 2. "." :表示在当前所在目录生成目录。

```
javac -d . Test.java
```

按照此种方式编译完成之后发现，自动会在当前目录下生成相应的文件夹以及相应的*.class文件。一旦程序类上出现了包名称，那么在执行的时候就需要带上包名称，即使用完整类名称"包.类"。

```
java www.bit.java.Test
```

在以后进行项目开发之中，一定要定义包

1.2 包的导入

开发中使用包的定义之后，相当于把一个大的项目分别按照一定要求保存在了不同的包之中，但是这些程序类一定会发生互相调用的情况，这个时候就需要包的导入。

范例：编写简单类，本类需要被其他程序类所使用。

```
package www.bit.java.util;

public class Message {
    public void print() {
        System.out.println("[Message] Hello Package");
    }
}
```

范例：导入包

```
package www.bit.java.test ;
import www.bit.java.util.Message ;  // 导入程序类
public class Test {
    public static void main(String[] args) {
        Message msg = new Message() ;
        msg.print() ;
    }
}
```

从正常的角度来讲，Test类引用了Message类，那么首先编译的应该是Message类，而后才是Test类。

```
package www.bit.java.util ;

public class Message {
    public static void main(String[] args) {
        System.out.println("[Message] Hello Java") ;
    }
}
```

最好的方法是让java自己去匹配编译的先后顺序，最常用的打包编译命令为:javac -d . ./*.java

注意：类使用class和public class的区别：

> 1. public class: 文件名称必须与类名称保持一致，如果希望一个类被其他包访问，则必须定义为public class .
> 2. class: 文件名称可以与类名称不一致，在一个*.java中可以定义多个class，但是这个类不允许被其他包所访问。

另外需要注意的是，以上导入的语句为"import 包.类"这样只会导入一个类，如果说现在导入一个包中的多个类，可以直接采用通配符"*"来完成。

```
import www.bit.java.util.* ;
```

这种"*"并不意味要将包中的所有类都进行导入，而是根据你的需求来导入。

问题：不同包但是相同类名的情况

```
package www.bit.java.test ;
public class Message {
    public void print() {
        System.out.println("[Message] Hello Java") ;
    }
}
```

假设现在Test需要同时导入这两个包

```
package www.bit.java.test ;
import www.bit.java.util.* ;  // 导入程序类
import www.bit.java.message.* ; // 导入程序类
public class Test {
        public static void main(String[] args) {
                Message msg = new Message() ;
                msg.print() ;
        }
}
```

这个时候如果还是直接使用Message类就会产生一个编译上的歧义。

这个时候我们一般在使用时使用全名称定义

```
www.bit.java.message.Message msg = new www.bit.java.message.Message() ;
```

1.3 系统常用包(了解)

系统常用包：

1. java.lang:系统常用基础类(String、Object),此包从JDK1.1后自动导入。
2. java.lang.reflect:java 反射编程包;
3. java.net:进行网络编程开发包。
4. java.sql:进行数据库开发的支持包。
5. java.util:是java提供的工具程序包。(集合类等)(**巨重要**)
6. java.io:I/O编程开发包。
7. java.awt(离不开windows平台)、java.swing:UI开发包，主要进行界面开发包，目前已经不用了。

1.4 访问控制权限

在之前所学习到的private、public就是访问控制权限。在java中提供有四种访问控制权限：`private<default<protected<public`，这四种访问控制权限的定义如下：

No	范围	private	default	protected	public
1	同一包中的同一类	✓	✓	✓	✓
2	同一包中的不同类		✓	✓	✓
3	不同包中的子类			✓	✓
4	不同包中的非子类				✓

对于public永远都可以访问，对于封装性而言主要是private、default、protected权限。

范例：观察protected访问权限。

```
package father;

public class Father {
        // 此时定义的是protected权限
        protected String msg = "www.bit.java" ;
}
```

范例：定义另外一个包进行该类继承

```
package child;

import father.Father; // 不同包

public class Child extends Father{
        public void print() {
                System.out.println(super.msg); // 父类中protected权限
        }
}
```

范例：定义测试类

```
package test;

import child.Child;

public class TestProtected {
        public static void main(String[] args) {
                Child child = new Child() ;
                child.print();
        }
}
```

在不同包中，只有子类能访问父类中的protected权限。

总结：关于权限选择

> 1. 对于封装的描述90%使用private,只有10%会使用protected,这两个都叫封装。
> 2. 属性都使用private,方法都使用public.

封装性就是指`private、default、protected`三种权限的使用。

1.5 jar命令

jar本质上也是一种压缩文件，里面保存的都是*.class文件。也就是说现在要实现某一个功能模块，可能有几百个类，最终交付给用户使用时，为了方便管理，就会将这些文件形成压缩包提供给用户。

在JDK中提供实现jar文件操作的命令，只需要输入一个jar即可。对于此命令，有如下几个常用参数：

> 1. "c":创建新档案
> 2. "f":指定档案文件名
> 3. "v":在标准输出中生成详细输出

范例：编译源文件并打包

```
package www.bit.java.util;

public class Message {
    public void print() {
        System.out.println("Hello I am Message");
    }
}
```

对上述源文件编译而后变为jar文件。

> 1. 打包进行程序编译: javac -d . Message.java
> 2. 将生成的程序类打包为jar文件:jar -cvf Message.jar Message.class

打开后发现有一个META-INF文件夹，里面包含版本号等信息。

此时的Message.jar就包含我们需要的程序类.

范例：编写程序调用Message.jar

```
package www.bit.java.testjar;

import www.bit.java.util.Message;

public class Testjar {
        public static void main(String[] args) {
                Message msg = new Message();
                msg.print() ;
        }
}
```

要想使用jar文件，并不是说将其放到程序的目录之中就可以，还需要配置CLASSPATH，设置jar文件的加载路径才会起效。

以后的开发会使用大量的jar文件，所有的jar文件必须配置在classpath中。

2.设计模式（重要）

2.1 单例设计模式

所谓的单例设计指的是一个类只允许产生一个实例化对象。

范例：一个简单程序

```
package test;

class Singleton{
        public void print() {
                System.out.println("Hello World");
        }
}
public class SingletonTest {
        public static void main(String[] args) {
                Singleton singleton = null ; // 声明对象
                singleton = new Singleton() ; // 实例化对象
                singleton.print();
        }
}
```

以上程序在进行对象实例化的时候调用了Singleton的无参构造。

范例：使用private声明构造方法

```
class Singleton{
        private Singleton() { // private声明构造
        }
        public void print() {
                System.out.println("Hello World");
        }
}
```

这个时候类中已经明确的提供了一个私有的构造方法，那么默认生成的无参构造不再产生，此时进行对象实例化的时候一定会有错误。

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The constructor Singleton() is not visible
```

一旦构造方法被私有化了，表示外部无法调用构造即外部不能够产生新的实例化对象。此时的类是一个相对而言封闭的状态。

如果此时还想继续调用Singleton类的print()方法，那么必须提供实例化对象。考虑到封装的特点，可以在类的内部产生一个实例化对象。

范例：在类的内部产生实例化对象

```
class Singleton{
        // 在类的内部可以访问私有结构，所以可以在类的内部产生实例化对象
        Singleton instance = new Singleton() ;
        private Singleton() { // private声明构造
        }
        public void print() {
                System.out.println("Hello World");
        }
}
```

现在Singleton内部的instance对象（属性）是一个普通属性，所有的普通属性必须在有实例化对象的时候才能进行内存空间的分配，而现在外部无法产生实例化对象，所以必须想一个办法，可以在Singleton没有实例化对象产生的时候，也可以将instance进行使用。此时，联想到使用static关键字。

范例：使用static产生实例化对象

```
package test;

class Singleton{
        // 在类的内部可以访问私有结构，所以可以在类的内部产生实例化对象
        static Singleton instance = new Singleton() ;
        private Singleton() { // private声明构造
        }
        public void print() {
                System.out.println("Hello World");
```

```
        }
}
public class SingletonTest {
        public static void main(String[] args) {
                Singleton singleton = null ; // 声明对象
                singleton = Singleton.instance ;
                singleton.print();
        }
}
```

以上虽然可以取得Singleton类的实例化对象，但是对于类中属性应该使用private进行封装，要想取得private属性，应该提供getter()方法。由于此时访问的是static属性，并且这个类无法在外部提供实例化对象，因此应该提供一个static的getter()方法，因为static方法也不受对象实例化控制。

范例:static的getter()方法

```
package test;

class Singleton{
        // 在类的内部可以访问私有结构，所以可以在类的内部产生实例化对象
        private static Singleton instance = new Singleton() ;
        private Singleton() { // private声明构造
        }
        public static Singleton getInstance() {
                return instance ;
        }
        public void print() {
                System.out.println("Hello World");
        }
}
public class SingletonTest {
        public static void main(String[] args) {
                Singleton singleton = null ; // 声明对象
                singleton = Singleton.getInstance() ;
                singleton.print();
        }
}
```

这样做到底要干哈？？？？？？？

只希望类中产生唯一的一个实例化对象

对于单例设计模式也有两类形式：懒汉式、饿汉式。

上面的代码实际上就是饿汉式的应用。**不管你是否使用Singleton类的对象，只要该类加载了，那么一定会自动创建好一个公共的instance对象。既然是饿汉式，就希望整体的操作之中只能够有一个实例化对象，所以一般还会在前面追加一个final关键字**

范例：饿汉式单例模式

```
package test;

class Singleton{
        // 在类的内部可以访问私有结构，所以可以在类的内部产生实例化对象
        private final static Singleton INSTANCE = new Singleton() ;
        private Singleton() { // private声明构造
        }
        public static Singleton getInstance() {
                return INSTANCE ;
        }
        public void print() {
                System.out.println("Hello World");
        }
}
public class SingletonTest {
        public static void main(String[] args) {
                Singleton singleton = null ; // 声明对象
                singleton = Singleton.getInstance() ;
                singleton.print();
        }
}
```

面试题：请编写一个单例程序，并说明程序的主要特点。

特点：构造方法私有化，外部无法产生新的实例化对象，只能通过static方法取得实例化对象

范例：懒汉式单例模式

特点：当第一次去使用Singleton对象的时候才会为其产生实例化对象的操作.

```
package test;

class Singleton{
        private static Singleton instance ;
        private Singleton() { // private声明构造
        }
        public static Singleton getInstance() {
                if (instance==null) { // 表示此时还没有实例化
                        instance = new Singleton() ;
                }
                return instance ;
        }
        public void print() {
                System.out.println("Hello World");
        }
}
public class SingletonTest {
        public static void main(String[] args) {
                Singleton singleton = null ; // 声明对象
```

```
                singleton = Singleton.getInstance() ;
                singleton.print();
        }
}
```

对于饿汉式和懒汉式我们暂时理解即可，各位需要把单例设计的核心组成记住，慢慢理解用法。

（懒汉式存在多线程安全问题，而饿汉式不会。）

单例模式是一个重点，重点，重点。虽然代码量不大，但是概念用到的很多。

面试：如何解决懒汉式的线程安全问题?(后面会讲)(双重加速单例模式)

2.2 多例设计模式（理解概念）

要求描述一周数的类，只能有七个对象；描述性别的类，只能有两个。这些都属于多例设计模式。

所谓的多例只是比单例追加了更多个内部实例化对象产生而已。

范例：定义一个表示性别的多例类

```
package test;

class Sex {
        private String title ;
        public static final int MALE_FLAG = 1 ;
        public static final int FEMALE_FLAG = 2 ;
        private static final Sex MALE = new Sex("男") ;
        private static final Sex FEMALE = new Sex("女") ;
        private Sex(String title) {
                this.title = title ;
        }
        public static Sex getInstance(int flag) {
                switch (flag) {
                case MALE_FLAG:
                        return MALE ;
                case FEMALE_FLAG:
                        return FEMALE ;
                default:
                        return null ;
                }
        }
        @Override
        public String toString() {
                return this.title ;
        }
}

public class MultitonTest {
        public static void main(String[] args) {
                Sex male = Sex.getInstance(Sex.MALE_FLAG) ;
```

```
                System.out.println(male);
        }
}
```

不管多例还是单例都有共同特点：

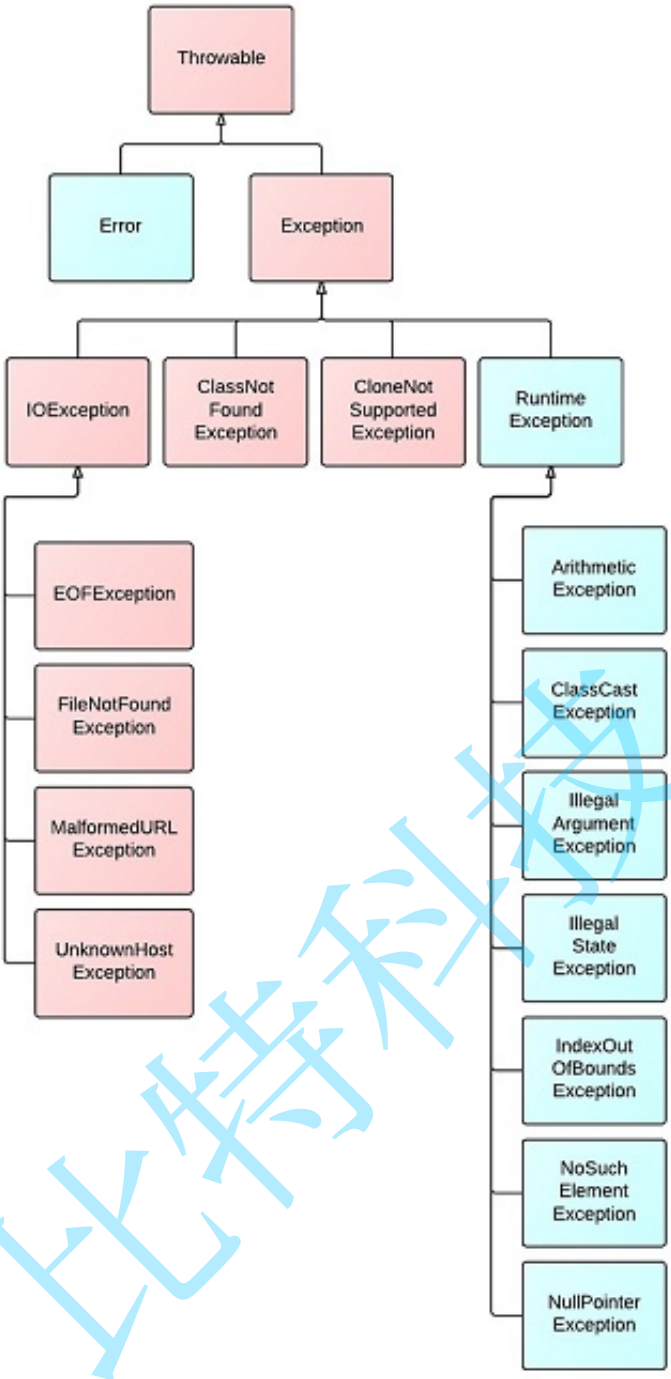
1. 构造方法私有化。
2. 类内部一定会提供一个static方法用于取得实例化对象。

总结： 单例和多例的代码我们编写的几率并不高，但是对于代码结构我们一定要清楚，尤其是单例设计，很多面试会问。多例设计模式我们先理解概念，该概念已经被枚举所取代。

3. 异常与捕获

几乎所有的代码里面都会出现异常，为了保证程序在出现异常之后可以正常执行完毕，就需要进行异常处理。

Java异常体系的类继承结构如下图:



所有的异常都是由Throwable继承而来,我们来看他下面的两个子类Error和Exception.

- Error类：描述了Java运行时内部错误和资源耗尽错误。应用程序不抛出此类异常，这种内部错误一旦出现，除了告知用户并使程序安全终止之外，再无能无力。这种情况很少出现。
- 我们需要关心的是Exception以及其子类。在Exception之下又分为两个分支，RuntimeException和IOException。

由于程序错误导致的异常属于RuntimeException；而如果程序本身没有问题，但由于像I/O错误这类问题导致的异常属于IOException。

Java语言规范将派生于Error类或RuntimeException类的所有异常称为非受查异常；所有的其他异常称为受查异常。

3.1 异常的影响

异常是导致程序中断执行的一种指令流。程序之中如果出现异常并且没有合理处理的话就会导致程序终止执行。

范例：观察正确程序流

```
package test;

public class Test {
        public static void main(String[] args) {
                System.out.println("1.数学计算开始前");
                System.out.println("2.进行数学计算："+10/2);
                System.out.println("3.数学计算结束后");
        }
}
```

此时没有任何异常产生，程序可以正常执行完毕。

范例：产生异常

```
package test;

public class Test {
        public static void main(String[] args) {
                System.out.println("1.数学计算开始前");
                System.out.println("2.进行数学计算："+10/0);
                System.out.println("3.数学计算结束后");
        }
}
```

1.数学计算开始前 Exception in thread"main"java.lang.ArithmeticException: / by zero at test.Test.main(Test.java:6)

现在程序之中产生了异常，但是在异常语句产生之前的语句可以正常执行完毕，而异常产生之后程序直接进行了结束。为了保证程序出现异常后还可以继续向下执行，就需要异常处理。

3.2 异常处理格式

异常处理的语法格式如下：

```
try{
        有可能出现异常的语句 ;
}[catch (异常类 对象) {
} ... ]
[finally {
        异常的出口
}]
```

对于以上三个关键字，可以出现的组合：`try..catch`、`try..finally`、`try..catch..finally`

范例：对异常进行处理

```
package test;

public class Test {
	public static void main(String[] args) {
		System.out.println("1.数学计算开始前");
		try {
			System.out.println("2.进行数学计算："+10/0);
		} catch (ArithmeticException e) {
			System.out.println("异常已经被处理了");
		}
		System.out.println("3.数学计算结束后");
	}
}
```

出现了异常之后，由于存在异常处理机制，依然可以正常执行完毕。

以上代码虽然进行了异常处理，但是存在一个问题：你现在根本不知道程序产生了什么样的异常。所以为了明确的取得异常信息，可以直接输出异常类对象，或者调用所有异常类中提供的printStackTrace()方法进行完整异常信息的输出。

范例：取得异常的完整信息

```
package test;

public class Test {
	public static void main(String[] args) {
		System.out.println("1.数学计算开始前");
		try {
			System.out.println("2.进行数学计算："+10/0);
		} catch (ArithmeticException e) {
			e.printStackTrace();
		}
		System.out.println("3.数学计算结束后");
	}
}
```

在进行异常处理的时候还可以使用`try..catch..finally`进行处理。

范例：使用`try..catch..finally`进行处理

```
package test;

public class Test {
	public static void main(String[] args) {
		System.out.println("[1].数学计算开始前");
		try {
```

```
                        System.out.println("[2].进行数学计算: "+10/0);
                } catch (ArithmeticException e) {
                        e.printStackTrace();
                }finally {
                        System.out.println("[Finally]不管是否产生异常，都执行此语
句");
                }
                System.out.println("[3].数学计算结束后");
        }
}
```

不管此时是否产生异常，最终都要执行finally程序代码，所以finally会作为程序统一出口。

以上程序是直接固定好了两个数字进行除法运算，现在通过初始化参数来进行除法运算

范例：初始化参数进行数学运算

```
package test;

public class Test {
        public static void main(String[] args) {
                System.out.println("[1].数学计算开始前");
                try {
                        int x = Integer.parseInt(args[0]) ;
                        int y = Integer.parseInt(args[1]) ;
                        System.out.println("[2].进行数学计算: "+x/y);
                } catch (ArithmeticException e) {
                        e.printStackTrace();
                }finally {
                        System.out.println("[Finally]不管是否产生异常，都执行此语
句");
                }
                System.out.println("[3].数学计算结束后");
        }
}
```


此时会存在如下问题：

> - 用户没有输入初始化参数：ArrayIndexOutOfBoundsException
> - 用户输入的不是数字: NumberFormatException
> - 被除数为0：ArithmeticException

以上代码我们发现，通过catch捕获异常的时候如果没有捕获指定异常，程序依然无法进行处理，现在最直白的解决方法就使用多个catch。

范例：多个catch块

```
package test;

public class Test {
```

```
        public static void main(String[] args) {
                System.out.println("[1].数学计算开始前");
                try {
                        int x = Integer.parseInt(args[0]) ;
                        int y = Integer.parseInt(args[1]) ;
                        System.out.println("[2].进行数学计算："+x/y);
                }catch (ArithmeticException e) {
                        e.printStackTrace();
                }
                catch (NumberFormatException e) {
                        e.printStackTrace();
                }
                catch (ArrayIndexOutOfBoundsException e) {
                        e.printStackTrace();
                }finally {
                        System.out.println("[Finally]不管是否产生异常，都执行此语
句");
                }
                System.out.println("[3].数学计算结束后");
        }
}
```

问题是真这么写，真这么测试，不使用异常处理也一样，因为完全可以使用if..else判断。如果要想更好的处理异常，必须清楚异常的处理流程。

3.3 throws关键字

在进行方法定义的时候，如果要告诉调用者本方法可能产生哪些异常，就可以使用throws方法进行声明。即，如果该方法出现问题后不希望进行处理，就使用throws抛出。

thorws用在方法上

范例：使用throws定义方法

```
package test;

public class Test {
        public static void main(String[] args) {
                try {
                        System.out.println(calculate(10, 0));
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
        public static int calculate(int x,int y) throws Exception {
                return x/y ;
        }
}
```

如果现在调用了throws声明的方法，那么在调用时必须明确的使用try..catch..进行捕获，因为该方法有可能产生异常，所以必须按照异常的方式来进行处理。

主方法本身也属于一个方法，所以主方法上也可以使用throws进行异常抛出，这个时候如果产生了异常就会交给JVM处理。

范例：主方法抛出异常

```
package test;

public class Test {
        public static void main(String[] args) throws Exception{
                        System.out.println(calculate(10, 0));
        }
        public static int calculate(int x,int y) throws Exception {
                return x/y ;
        }
}
```

以后编写代码里面，一定要斟酌好可能产生的异常。面对未知的程序类，如果要进行异常的处理，就必须知道有多少种异常。

3.4 throw关键字

throw是直接编写在语句之中，表示人为进行异常的抛出。如果现在异常类对象实例化不希望由JVM产生而由用户产生，就可以使用throw来完成

throw用在方法中

范例：使用throw产生异常类对象。

```
        public static void main(String[] args){
                try {
                        throw new Exception("抛个异常玩玩") ;
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
```

一般而言throw和break、continue、break都要和if结合使用。

面试题：请解释throw和throws的区别

> 1. throw用于方法内部，主要表示手工异常抛出。
> 2. throws主要在方法声明上使用，明确告诉用户本方法可能产生的异常，同时该方法可能不处理此异常。

3.5 异常处理标准格式

以上为止，异常中的所有核心概念都掌握了：`try、catch、finally、throws、throw`。

现在要求编写一个方法进行除法操作，但是对于此方法有如下要求：

> 1. 在进行除法计算操作之前打印一行语句"**"。
> 2. 如果在除法计算过程中出现错误，则应该将异常返回给调用处。
> 3. 不管最终是否有异常产生，都要求打印一行计算结果信息。

范例：以上程序实现

```
package test;

public class Test {
	public static void main(String[] args){
		try {
			System.out.println(calculate(10, 0));
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
	public static int calculate(int x,int y) throws Exception {
		int result = 0 ;
		System.out.println("1.[计算开始前]*****");
		try {
			result = x / y ;
		} catch (Exception e) {
			throw e ; // 抛出去
		} finally {
			System.out.println("2.[计算结束]#####");
		}
		return result ;
	}
}
```

对于以上格式还可以进一步简化，直接使用try..finally

```
	public static int calculate(int x,int y) throws Exception {
		int result = 0 ;
		System.out.println("1.[计算开始前]*****");
		try {
			result = x / y ;
		}finally {
			System.out.println("2.[计算结束]#####");
		}
		return result ;
	}
```

对于此时的格式一定要吸收，后面会对此结构进一步优化。（使用代理模式）

3.6 RuntimeException类（面试重点）

先来看一段贼简单的代码：

```
package test;

public class Test {
        public static void main(String[] args){
                String str = "100" ;
                int num = Integer.parseInt(str) ;
                System.out.println(num * 2);
        }
}
```

我们来看parseInt的源码定义：

```
public static int parseInt(String s) throws NumberFormatException
```

这个方法上已经明确抛出异常，但是在进行调用的时候发现，即使没有进行异常处理也可以正常执行。这个就属于RuntimeException的范畴。

很多的代码上都可能出现异常，例如"10/0"都可能产生异常，如果所有可能产生异常的地方都强制性异常处理，这个代码就太复杂了。所以在异常设计的时候，考虑到一些异常可能是简单问题，所以将这类异常称为RuntimeException，也就是使用RuntimeException定义的异常类可以不需要强制性进行异常处理。

面试题：请解释Exception与RuntimeException的区别，请列举几个常见的RuntimeException：

> 1. 使用Exception是RuntimeException的父类，使用Exception定义的异常都要求必须使用异常处理，而使用RuntimeException定义的异常可以由用户选择性的来进行异常处理。
> 2. 常见的RuntimeException:ClassCastException、NullPointerException等。

3.7 断言assert（了解）

断言是从JDK1.4开始引入的概念。断言指的是当程序执行到某些语句之后其数据的内容一定是约定的内容。

范例：观察断言

```
package test;

public class Test {
        public static void main(String[] args){
                int num = 10 ;
                assert num == 55 : "错误：num应当为55" ;
                System.out.println(num);
        }
}
```

如果要想让断言起作用，则必须使用`-ea`的参数，启用断言。

范例：启用断言(java -ea)

```
Exception in thread "main" java.lang.AssertionError: 错误：num应当为55
        at test.Test.main(Test.java:6)
```

实际上断言的意义并不是很大，Java之所以引入主要是为了与C++兼容，开发之中不提倡使用断言。

3.8 自定义异常类

在Java里，针对于可能出现的公共的程序问题都会提供有相应的异常信息，但是很多时候这些异常信息往往不够我们使用。例如，现在有需求：在进行加法运算时，如果发现两个数相加内容为50，那么就应当抛出一个AddException异常。这种异常Java不会提供，所以就必须定义一个属于自己的异常类。

自定义异常类可以继承两种父类：Exception、RuntimeException。

范例：实现自定义异常类

```
package test;

class AddException extends Exception {
        public AddException(String msg) {
                super(msg) ;
        }
}
public class Test {
        public static void main(String[] args) throws Exception{
                int num1 = 20 ;
                int num2 = 30 ;
                if (num1+num2==50) {
                        throw new AddException("错误的相加操作") ;
                }
        }
}
```

在项目的系统设计或者架构设计时，一定会涉及到与业务相关的异常问题，此时需要自定义异常类。

以后所有java中的问题都去https://stackoverflow.com/网站查询。

本节需要掌握的：

1. 异常处理流程
2. 异常处理格式
3. 异常处理模型

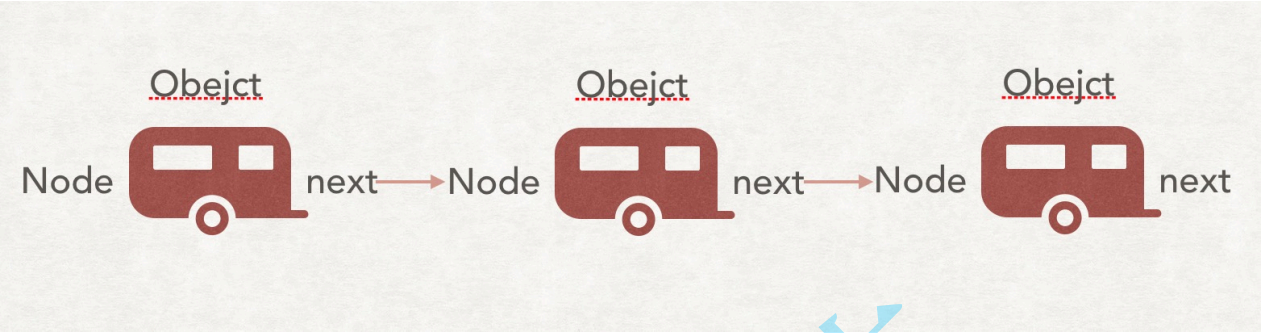
4. Java链表(面试、笔试)

4.1 链表的基本介绍

如果现在要想保存多个对象，那么首先可以想到的就是对象数组；如果要想保存多个任意对象，那么可以想到的一定是Object型的数组。

```
Object[] data = new Object[3] ;
```

但是在实际开发中，要面临的一个问题是：数组是一个定长的线性结构。一旦我们的内容不足或者内容过多，都有可能造成空间浪费。要想解决此类问题，最好的做法就是不定义一个固定长度的数组，有多少数据就保存多少数据。应该采用火车车厢的设计模式，动态进行车厢的挂载。



如果要想定义火车车厢，肯定不可能只保存一个数据，还需要另外一个指向，指向下一个节点。

范例：Java链表节点结构

```
package test;


class Node {
	private Object data;
	private Node next; // 指向下一个节点
	public Node(Object data) {
		this.data = data;
	}
	public void setData(Object data){
		this.data = data ;
	}
	public Object getData() {
		return this.data ;
	}
	public void setNext(Node next) {
		this.next = next ;
	}
	public Node getNext() {
		return this.next ;
	}
}
public class Test {
	public static void main(String[] args) throws Exception {
		// 1.封装节点
		Node root = new Node("火车头") ;
		Node n1 = new Node("车厢A") ;
```

```
                Node n2 = new Node("车厢B") ;
                Node n3 = new Node("车厢C") ;
                // 2.设置车厢关系，即挂载
                root.setNext(n1);
                n1.setNext(n2);
                n2.setNext(n3);
        }
}
```

设置好关系后，但是节点里存放的都是数据，如果要将数据依次取出，可以采用递归的形式。

范例：链表基本结构

```
package test;

class Node {
        private Object data;
        private Node next; // 指向下一个节点
        public Node(Object data) {
                this.data = data;
        }
        public void setData(Object data){
                this.data = data ;
        }
        public Object getData() {
                return this.data ;
        }
        public void setNext(Node next) {
                this.next = next ;
        }
        public Node getNext() {
                return this.next ;
        }
}
public class Test {
        public static void main(String[] args) throws Exception {
                // 1.封装节点
                Node root = new Node("火车头") ;
                Node n1 = new Node("车厢A") ;
                Node n2 = new Node("车厢B") ;
                Node n3 = new Node("车厢C") ;
                // 2.设置车厢关系，即挂载
                root.setNext(n1);
                n1.setNext(n2);
                n2.setNext(n3);
                // 3.依次取出节点
                getNodeData(root);
        }
        public static void getNodeData(Node node) {
```

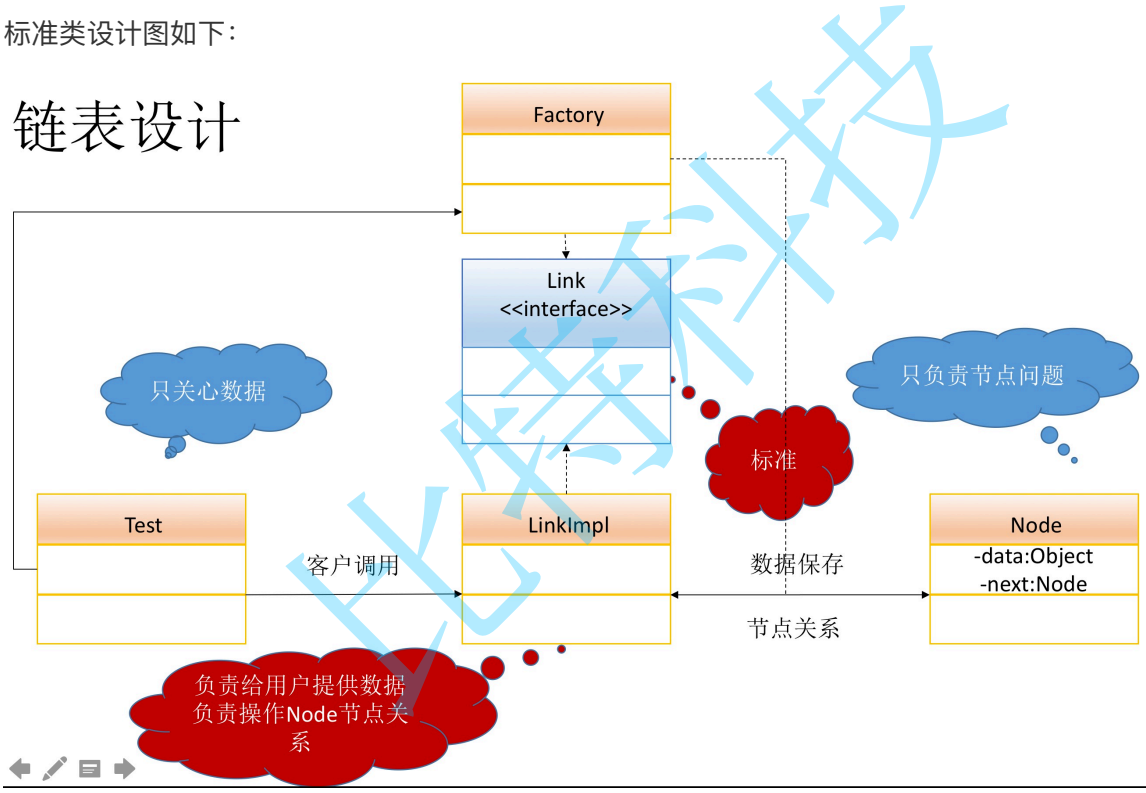
```
                if (node!=null) { // 当前存在节点
                        System.out.println(node.getData());
                        getNodeData(node.getNext());
                }
        }
}
```

在整个链表的实现过程中，Node类的核心作用在于保存数据和连接节点关系。但是发现，客户端需要自己来进行节点的创建以及关心的配置。所以链表还需要一个单独的类Link，通过Link来实现Node类的数据保存以及关系处理。(Node是链表的关键所在)

4.2 链表实现结构

通过上节讲解，为了更方便的管理和配置Node类的创建以及多个Node之间的关系，就需要引入新的类Link。

标准类设计图如下：



双向链表设计以及代码:

```
package www.bit.java;


import java.util.LinkedList;

interface ILink {
    /**
     * 链表增加节点操作
     * @param data 节点内容
     * @return
```

```
     */
    boolean add(Object data);

    /**
     * 判断指定内容节点在链表中是否存在
     * @param data 要判断的内容
     * @return 返回找到的节点索引
     */
    int contains(Object data);

    /**
     * 删除指定内容节点
     * @param data
     * @return
     */
    boolean remove(Object data);

    /**
     * 根据指定下标修改节点内容
     * @param index 索引下标
     * @param newData 替换后的内容
     * @return 替换之前的节点内容
     */
    Object set(int index,Object newData);

    /**
     * 根据指定下标返回节点内容
     * @param index
     * @return
     */
    Object get(int index);

    /**
     * 链表清空
     */
    void clear();

    /**
     * 将链表转为数组
     * @return 返回所有节点内容
     */
    Object[] toArray();

    /**
     * 链表长度
     * @return
     */
    int size();
```

```
    /**
     * 遍历链表
     */
    void printLink();
}

class LinkImpl implements ILink {
    private Node head;
    private Node last;
    private int size;

    // ----------------------------------------------------
    private class Node {
        private Node prev;
        private Object data;
        private Node next;

        public Node(Node prev, Object data, Node next) {
            this.prev = prev;
            this.data = data;
            this.next = next;
        }
    }
    // ----------------------------------------------------

    @Override
    public boolean add(Object data) {
        Node temp = this.last;
        Node newNode = new Node(temp,data,null);
        this.last = newNode;
        if (this.head == null) {
            this.head = newNode;
        }else {
            temp.next = newNode;
        }
        this.size++;
        return true;
    }

    @Override
    public int contains(Object data) {
        // null
        if (data == null) {
            int i = 0;
            for (Node temp = this.head;temp!=null;temp=temp.next) {
                if (temp.data == null){
                    return i;
                }
                i++;
```

```
            }
        }else {
            int i = 0;
            for (Node temp = this.head;temp!=null;temp=temp.next) {
                if (temp.data.equals(data)){
                    return i;
                }
                i++;
            }
        }
        return -1;
    }

    @Override
    public boolean remove(Object data) {
        if (data == null) {
            for (Node temp = this.head;temp!=null;temp=temp.next) {
                if (temp.data == null) {
                    unLink(temp);
                    return true;
                }
            }
        }else {
            for (Node temp = this.head;temp!=null;temp=temp.next) {
                if (data.equals(temp.data)) {
                    unLink(temp);
                    return true;
                }
            }
        }
        return false;
    }

    @Override
    public Object set(int index, Object newData) {
        if (!isLinkIndex(index)) {
            return null;
        }
        Node node = node(index);
        Object elementData = node.data;
        node.data = newData;
        return elementData;
    }

    @Override
    public Object get(int index) {
        if (!isLinkIndex(index)) {
            return null;
        }
```

```
        return node(index).data;
    }

    @Override
    public void clear() {
        for (Node temp = head;temp!=null;) {
            temp.data = null;
            Node node = temp.next;
            temp = temp.prev = temp.next = null;
            temp = node;
            this.size --;
        }
    }

    @Override
    public Object[] toArray() {
        Object[] result = new Object[size];
        int i = 0;
        for (Node temp = head;temp!=null;temp=temp.next){
            result[i++] = temp.data;
        }
        return result;
    }

    @Override
    public int size() {
        return this.size;
    }

    @Override
    public void printLink() {
        Object[] data = this.toArray();
        for (Object temp: data)  {
            System.out.println(temp);
        }
    }

    /**
     * 根据指定索引取得具体节点
     * @param index
     * @return
     */
    private Node node(int index) {
        if (index < (size>>1)) {
            Node temp = this.head;
            for (int i = 0;i<index;i++){
                temp = temp.next;
            }
            return temp;
```

```
        }
        Node temp = this.last;
        for (int i = size-1;i>index;i--) {
            temp = temp.prev;
        }
        return temp;
    }
    /**
     * 判断指定索引是否合法
     */
    private boolean isLinkIndex(int index) {
        return index>=0 && index<size;
    }
    private Object unLink(Node x) {
        Object elementData = x.data;
        Node prev = x.prev;
        Node next = x.next;
        if (prev == null) {
            this.head = next;
        }else {
            prev.next = next;
            x.prev = null;
        }
        if (next == null) {
            this.last = prev;
        }else {
            next.prev = prev;
            x.next = null;
        }
        x.data = null;
        this.size--;
        return elementData;
    }
}


public class Test {
    public static void main(String[] args) {
        ILink link = new LinkImpl();
        link.add("火车头");
        link.add("车厢1");
        link.add("车厢2");
        link.add(null);
        link.add("车厢尾");
        System.out.println(link.remove("车厢尾"));
        link.printLink();
    }
}
```