16 注解

16.1 编写四个 JUnit 测试用例，分别使用带或不带某个参数的@Test 注解。用 JUnit 执行这些测试

```
import org.junit.Test



class ScalaTest {



  @Test

  def test1(){

    print("test1")

  }



  @Test(timeout = 1L)

  def test2(){

    print("test2")

  }

}
```

16.2 创建一个类的示例，展示注解可以出现的所有位置。用@deprecated 作为你的示例注解。

```
@deprecated

class Test{
```

```
  @deprecated

  val t = _;



  @deprecated(message = "unuse")

  def hello(){

    println("hello")

  }

}



@deprecated

object Test extends App{

  val t = new Test()

  t.hello()

  t.t

}
```

16.3 Scala 类库中的哪些注解用到了元注解@param,@field,@getter,@setter,@beanGetter 或@beanSetter?

看 Scala 注解的源码就 OK 了

16.4 编写一个 Scala 方法 sum,带有可变长度的整型参数，返回所有参数之和。从 Java 调用该方法。

```
import annotation.varargs
```

```
class Test{


  @varargs

  def sum(n : Int*)={

    n.sum

  }

}

public class Hello {



    public static void main(String[] args){

        Test t = new Test();

        System.out.println(t.sum(1,2,3));

    }

}
```

16.5 编写一个返回包含某文件所有行的字符串的方法。从 Java 调用该方法。

```
import io.Source



class Test{
```

```
  def read()={

    Source.fromFile("test.txt").mkString

  }

}

public class Hello {

    public static void main(String[] args){

        Test t = new Test();

        System.out.println(t.read());

    }

}
```

16.6 编写一个 Scala 对象，该对象带有一个易失(volatile)的 Boolean 字段。让某一个线程睡眠一段时间，之后将该字段设为 true，打印消息，然后退出。而另一个线程不停的检查该字段是否为 true。如果是，它将打印一个消息并退出。如果不是，则它将短暂睡眠，然后重试。如果变量不是易失的，会发生什么？

这里只有一个线程修改 Boolean 字段，所以字段是否为 volatile 应该是没有区别的

```
import scala.actors.Actor



class T1(obj:Obj) extends Actor{

  def act() {

    println("T1 is waiting")

    Thread.sleep(5000)

    obj.flag = true
```

```
    println("T1 set flag = true")

  }

}



class T2(obj:Obj) extends Actor{

  def act() {

    var f = true

    while (f){

      if(obj.flag){

        println("T2 is end")

        f = false

      }else{

        println("T2 is waiting")

        Thread.sleep(1000)

      }

    }

  }

}
```

```
class Obj{

// @volatile

  var flag : Boolean = false

}



object Test{

  def main(args: Array[String]) {

    val obj = new Obj()

    val t1 = new T1(obj)

    val t2 = new T2(obj)

    t1.start()

    t2.start()

  }

}
```

16.7 给出一个示例，展示如果方法可被重写，则尾递归优化为非法

```
import annotation.tailrec



class Test{

  @tailrec

  def sum2(xs : Seq[Int],partial : BigInt) : BigInt = {
```

```
    if (xs.isEmpty) partial else sum2(xs.tail,xs.head + partial)

  }

}
```

编译报错,修改如下

```
import annotation.tailrec



object Test extends App{

  @tailrec

  def sum2(xs : Seq[Int],partial : BigInt) : BigInt = {

    if (xs.isEmpty) partial else sum2(xs.tail,xs.head + partial)

  }



  println(sum2(1 to 1000000,0))

}
```

16.8 将 allDifferent 方法添加到对象，编译并检查字节码。@specialized 注解产生了哪些方法?

```
object Test{

  def allDifferent[@specialized T](x:T,y:T,z:T) = x != y && x!= z && y != z

}
```

javap Test$得到

```
public final class Test$ extends java.lang.Object{
```

```
    public static final Test$ MODULE$;

    public static {};

    public boolean allDifferent(java.lang.Object, java.lang.Object, java.lang.Ob
ject);

    public boolean allDifferent$mZc$sp(boolean, boolean, boolean);

    public boolean allDifferent$mBc$sp(byte, byte, byte);

    public boolean allDifferent$mCc$sp(char, char, char);

    public boolean allDifferent$mDc$sp(double, double, double);

    public boolean allDifferent$mFc$sp(float, float, float);

    public boolean allDifferent$mIc$sp(int, int, int);

    public boolean allDifferent$mJc$sp(long, long, long);

    public boolean allDifferent$mSc$sp(short, short, short);

    public boolean allDifferent$mVc$sp(scala.runtime.BoxedUnit, scala.runtime.Bo
xedUnit, scala.runtime.BoxedUnit);

}
```

16.9 Range.foreach 方法被注解为@specialized(Unit)。为什么？通过以下命令检查字节码:

```
javap -classpath /path/to/scala/lib/scala-library.jar scala.collection.immutable.Range
```

并考虑 Function1 上的@specialized 注解。点击 Scaladoc 中的 Function1.scala 链接进行查看 首先来看 Function1 的源码

```
......
```

```
trait Function1[@specialized(scala.Int, scala.Long, scala.Float, scala.Double/*, scala.AnyRef*
/) -T1, @specialized(scala.Unit, scala.Boolean, scala.Int, scala.Float, scala.Long, scala.Doubl
e/*, scala.AnyRef*/) +R] extends AnyRef { self =>

  /** Apply the body of this function to the argument.

   * @return   the result of function application.

   */

  def apply(v1: T1): R

......
```

可以看到 Function1 参数可以是 scala.Int,scala.Long,scala.Float,scala.Double，返回值可以是 scala.Unit,scala.Boolean,scala.Int,scala.Float,scala.Long,scala.Double 再来看 Range.foreach 的源码

```
......

@inline final override def foreach[@specialized(Unit) U](f: Int => U) {

    if (validateRangeBoundaries(f)) {

      var i = start

      val terminal = terminalElement

      val step = this.step

      while (i != terminal) {

        f(i)

        i += step

      }

    }
```

```
  }

......
```

首先此方法是没有返回值的，也就是 Unit。而 Function1 的返回值可以是 scala.Unit,scala.Boolean,scala.Int,scala.Float,scala.Long,scala.Double 如果不限定@specialized(Unit), 则 Function1 可能返回其他类型，但是此方法体根本就不返回，即使设置了也无法获得返回值

16.10 添加 assert(n >= 0)到 factorial 方法。在启用断言的情况下编译并校验 factorial(-1)会抛异常。在禁用断言的情况下编译。会发生什么？用 javap 检查该断言调用

```
object Test {

  def factorial(n: Int): Int = {

    assert(n > 0)

    n

  }



  def main(args: Array[String]) {

    factorial(-1)

  }

}
```

编译报错

```
Exception in thread "main" java.lang.AssertionError: assertion failed

        at scala.Predef$.assert(Predef.scala:165)

        at Test$.factorial(Test.scala:6)

        at Test$.main(Test.scala:11)
```

```
        at Test.main(Test.scala)

        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)

        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:2
5)

        at java.lang.reflect.Method.invoke(Method.java:597)

        at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
```

禁用 assert

```
-Xelide-below 2011
```

反编译此类 javap -c Test$ 得到

```
......

public int factorial(int);

  Code:

    0:   getstatic       #19; //Field scala/Predef$.MODULE$:Lscala/Predef$;

    3:   iload_1

    4:   iconst_0

    5:   if_icmple       12

    8:   iconst_1

    9:   goto    13

    12:  iconst_0

    13:  invokevirtual   #23; //Method scala/Predef$.assert:(Z)V
```

```
  16:  iload_1

  17:  ireturn

......
```