JDK新特性

1、本章内容

1. 静态导入
2. 可变长参数
3. 格式化输入输出
4. 枚举
5. 注解 单元测试（Junit）
6. Lambda表达式
7. JVM调优

2、静态导入

方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。

正常的import声明从包中导入类，因此可以在没有包引用的情况下使用它们。类似地，静态导入声明从类中导入静态成员，并允许它们在没有类引用的情况下使用。

静态导入声明还有两种形式：单静态导入和静态导入按需。单静态导入声明从类型中导入一个静态成员。static-import-on-demand声明导入一个类型的所有静态成员。静态导入声明的一般语法如下：

```
//Single-static-import declaration:
import static << package name>>.<<type name>>.<< static member name>>;
//Static-import-on-demand declaration:
import static << package name>>.<<type name>>.*;
```

静态导入示例

例如，您记得使用该`System.out.println()`方法在标准输出中打印消息。`System`是**java.lang**包中的一个类，它有一个名为的静态变量`out`。当你使用时`System.out`，你指的是`System`该类之外的那个静态变量。您可以使用静态导入声明`out`从`System`类中导入静态变量，如下所示：

```
import static java.lang.System.out;
```

您的代码现在可以在程序中使用名称**out**表示`System.out`。编译器将使用静态进口报关来解析名称`out`来`System.out`。

```
public class StaticImportTest {
    public static void main(String[] args) {
        out.println( "Hello static import!" );
    }
}
```

静态导入规则

以下是有关静态导入声明的一些重要规则。

1）如果导入两个具有相同简单名称的静态成员，一个使用单静态导入声明，另一个使用静态导入按需声明，则使用单静态导入声明导入的静态成员优先。

假设有两个类，`package1.Class1`和`package2.Class2`。这两个类都有一个名为的静态方法`methodA`。以下代码将使用`package1.Class1.methodA()`方法，因为它是使用单静态导入声明导入的：

```
import static package1.Class1.methodA;
// Imports Class1.methodA() method
import static package2.Class2.*;
// Imports Class2.methodA() method too
public class Test {
    public static void main(String[] args) {
        methodA();    // Class1.methodA() will be called
    }
}
```

2）不允许使用单静态导入声明导入两个具有相同简单名称的静态成员。以下静态导入声明会生成错误，因为它们都使用相同的简单名称导入静态成员`methodA`：

```
import static package1.Class1.methodA;
import static package1.Class2.methodA;   // An error
```

3）如果使用单静态导入声明导入静态成员，并且在同一个类中存在具有相同名称的静态成员，则使用该类中的静态成员。

```
// A.java  package package1;
public class A {
    public static void test() {
        System.out.println(  "package1.A.test()"  );
    }
} // Test.java  package package2;
import static package1.A.test;
public class Test {
    public static void main(String[] args) {
        test();   // Will use package2.Test.test() method, not package1.A.test()
method
    }
    public static void test() {
        System.out.println(  "package2.Test.test()"  );
    }
} //Output: package2.Test.test()
```

静态导入似乎可以帮助您使用静态成员的简单名称来简化程序的编写和读取。有时静态导入可能会在程序中引入细微的错误，这可能很难调试。建议您根本不使用静态导入，或仅在极少数情况下使用静态导入。

3、可变长参数

1、什么是可变长参数

可变长参数顾名思义是可以改变长度的参数，意为传入的参数个数可以不固定。

Java中什么数据类型可以改变长度？当然是数组。确实，在JDK5之前的确是使用数组来实现可变长参数的。

那么什么是可变长参数？如下代码：

```
public static void test(String[] args){
    for(String str:args){
        System.out.println(str);
    }
}
```

上面代码中的`String[] args`是不是可变长的参数？确实是可以实现的，但是可读性不高，所以有了下面的写法：

```
public static void test(String...args){
    for(String str:args){
        System.out.println(str);
    }
}
```

如上代码中的`String...args`就是我们要讲的可变长参数。

2、可变长参数的定义

可变长参数说到底还是一个参数，需要定义在方法的参数列表中，如：

```
public void test(int...nums){
    // 方法体
}
```

3、可变长参数的使用

参数分形参和实参，而可变长参数是定义在形参中，并且可以与其他类型的参数组合使用。需要注意的是，如果方法的形参有多个，那么**可变长参数只能定义在参数列表的最后**，并且方法的**参数中只能有一个可变长参数**，否则无法通过编译。

正确使用方式如下：

```
// 定义可变长参数的方法
public void test(Integer number,String...args){
}
123
// 调用如上方法
test(1,"可变长参数1","可变长参数2");
```

4、遍历可变长参数

要遍历这个可变长参数之前我们需要搞清楚它是一个什么数据类型，所以我们可以进行如下实验：

```
public void test(Integer number,String...args){
    System.out.println(number.getClass());
    System.out.println(args.getClass());
}
```

输出结果为：

```
class java.lang.Integer
class [Ljava.lang.String;
```

class的描述中，可变长参数以“[”开头，表示它是一个数组。既然是一个数组我们就可以以遍历数组的方式遍历这个可变长参数，如：

```
// 遍历可变长参数
public static void test(Integer number,String...args){
    System.out.println(number);
    for (String s : args) {
        System.out.println(s);
    }
}
```

4、格式化输入输出

格式化输出

Java提供了格式化输出的功能，如果要把数据显示成我们期望的格式，就需要使用格式化输出的功能。

格式化输出使用System.out.printf()，通过使用占位符 "%"，printf()可以把后面的参数格式化成指定格式。

```
double d = 12900000;
System.out.println(d);  // 输出结果：1.29E7
double b = 3.1415926;
System.out.printf("%.2f\n", b);  // 小数点后保留2位，输出结果：3.14
System.out.printf("%.4f\n", b);  // 小数点后保留4位，输出结果：3.1416
```

Java格式化功能占位符，可以把各种数据类型"格式化"成指定的字符串：

占位符	说明
%d	格式化输出整数
%x	格式化输出十六进制整数
%f	格式化输出浮点数
%e	格式化输出科学计数法表示的浮点数
%s	格式化字符串

注意：由于%表示占位符，因此，连续两个%%表示一个%字符本身。

格式化输入

从控制台读取一个字符串和一个整数的例子：

```
// 创建Scanner对象，System.in代表标准输入流
Scanner scanner = new Scanner(System.in);
```

```
// 打印提示，输入名字
System.out.print("Input your name: ");
// 读取一行输入并获取字符串
String name = scanner.nextLine();
// 打印提示，输入年龄
System.out.print("Inpur your age: ");
// 读取一行输入并获取整数
int age = scanner.nextInt();
// 格式化输出用户信息
System.out.printf("Hi, %s, you are %d\n", name, age);
// 输出结果：
Input your name: xiaokang
Inpur your age:  20
Hi, xiaokang, you are 20
```

注：

（1）System.out代表标准输出流，System.in代表标准输入流。

（2）读取用户输入的字符串，使用scanner.nextLine()

（3）读取用户输入的整数，使用scanner.nextInt()

参考实例

1.对齐输出：

右对齐

```
double x=5247.1213560977;
System.out.printf("%08.2f \n",x);//0补齐8位
```

```
05247.12
```

左对齐

```
double x=86.16689727392823;
System.out.printf("%-8.2f \n",x);
```

```
86.17
```

2.分组的分隔符

```
double x=7968.401526605161;
System.out.printf("%,.2f \n",x);
```

```
7,968.40
```

3.$表示参数重用

```
double x=7968.401526605161;
System.out.printf("%1$10.4f %1$9.2f %2$s \n",x,"abc");//1$第一个参数，2$第二个参数
```

```
 7968.4015   7968.40 abc
```

4.输出完整的日期和时间

```
System.out.printf("%tc \n",new Date());//c代表完整的日期和时间
```

```
周二 10月 12 11:37:22 CST 2021
```

5.输出月份

```
System.out.printf("%tb \n",new Date); //b代表月
```

```
10月
```

```
System.out.printf("%tB \n",new Date); //b代表月
```

```
十月
```

5、枚举

什么是枚举

Java中的枚举是一种类型，顾名思义：就是一个一个列举出来。所以它一般都是表示一个有限的集合类型，它是一种类型，在维基百科中给出的定义是：

> 在数学和计算机科学理论中，一个集的枚举是列出某些有穷序列集的所有成员的程序，或者是一种特定类型对象的计数。这两种类型经常（但不总是）重叠.。枚举是一个被命名的整型常数的集合，枚举在日常生活中很常见，例如表示星期的SUNDAY、MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY就是一个枚举。

出现的原因

在Java5之前，其实是没有enum的，所以先来看一下Java5之前对于枚举的使用场景该怎么解决？这里我看到了一片关于在Java 1.4之前的枚举的设计方案：

```
public class Season {
    public static final int SPRING = 1;
    public static final int SUMMER = 2;
    public static final int AUTUMN = 3;
    public static final int WINTER = 4;
}
```

这种方法称作int枚举模式。可这种模式有什么问题呢?通常我们写出来的代码都会考虑它的安全性、易用性和可读性。 首先我们来考虑一下它的类型安全性。当然这种模式不是类型安全的。比如说我们设计一个函数，要求传入春夏秋冬的某个值。但是使用int类型，我们无法保证传入的值为合法。代码如下所示：

```
private String getChineseSeason(int season){
        StringBuffer result = new StringBuffer();
        switch(season){
            case Season.SPRING :
                result.append("春天");
                break;
            case Season.SUMMER :
                result.append("夏天");
                break;
            case Season.AUTUMN :
```

```
                result.append("秋天");
                break;
            case Season.WINTER :
                result.append("冬天");
                break;
            default :
                result.append("地球没有的季节");
                break;
        }
        return result.toString();
    }
```

因为我们传值的时候，可能会传其他的类型，就可能导致走default，所以这个并不能在源头上解决类型安全问题。

接下来我们来考虑一下这种模式的可读性。使用枚举的大多数场合，我都需要方便得到枚举类型的字符串表达式。如果将int枚举常量打印出来，我们所见到的就是一组数字，这是没什么太大的用处。我们可能会想到使用String常量代替int常量。虽然它为这些常量提供了可打印的字符串，但是它会导致性能问题，因为它依赖于字符串的比较操作，所以这种模式也是我们不期望的。 从类型安全性和程序可读性两方面考虑，int和String枚举模式的缺点就显露出来了。幸运的是，从Java1.5发行版本开始，就提出了另一种可以替代的解决方案，可以避免int和String枚举模式的缺点，并提供了许多额外的好处。那就是枚举类型（enum type）。

枚举定义

枚举类型（enum type）是指由一组固定的常量组成合法的类型。Java中由关键字enum来定义一个枚举类型。下面就是java枚举类型的定义。

```
public enum Season {
    SPRING, SUMMER, AUTUMN, WINER;
}
```

Java定义枚举类型的语句很简约。它有以下特点：

- 使用关键字enum
- 类型名称，比如这里的Season
- 一串允许的值，比如上面定义的春夏秋冬四季
- 枚举可以单独定义在一个文件中，也可以嵌在其它Java类中
- 枚举可以实现一个或多个接口（Interface）
- 可以定义新的变量和方法

重写上面的枚举方式

下面是一个很规范的枚举类型

```
public enum Season {
    SPRING(1), SUMMER(2), AUTUMN(3), WINTER(4);
    private int code;
    private Season(int code){
        this.code = code;
    }
    public int getCode(){
        return code;
    }
}
```

```
public class UseSeason {
    /**
     * 将英文的季节转换成中文季节
     * @param season
     * @return
     */
    public String getChineseSeason(Season season){
        StringBuffer result = new StringBuffer();
        switch(season){
            case SPRING :
                result.append("[中文：春天，枚举常量:" + season.name() + "，数据:" +
season.getCode() + "]");
                break;
            case AUTUMN :
                result.append("[中文：秋天，枚举常量:" + season.name() + "，数据:" +
season.getCode() + "]");
                break;
            case SUMMER :
                result.append("[中文：夏天，枚举常量:" + season.name() + "，数据:" +
season.getCode() + "]");
                break;
            case WINTER :
                result.append("[中文：冬天，枚举常量:" + season.name() + "，数据:" +
season.getCode() + "]");
                break;
            default :
                result.append("地球没有的季节 " + season.name());
                break;
        }
        return result.toString();
    }
    public void doSomething(){
        for(Season s : Season.values()){
            System.out.println(getChineseSeason(s));//这是正常的场景
        }
        //System.out.println(getChineseSeason(5));
        //此处已经是编译不通过了，这就保证了类型安全
    }
    public static void main(String[] arg){
        UseSeason useSeason = new UseSeason();
        useSeason.doSomething();
    }
}
```

Enum类的常用方法

方法名称	描述
values()	以数组形式返回枚举类型的所有成员
valueOf()	将普通字符串转换为枚举实例
compareTo()	比较两个枚举成员在定义时的顺序
ordinal()	获取枚举成员的索引位置

values() 方法

通过调用枚举类型实例的 values() 方法可以将枚举的所有成员以数组形式返回，也可以通过该方法获取枚举类型的成员。

下面的示例创建一个包含 3 个成员的枚举类型 Signal，然后调用 values() 方法输出这些成员。

```
public enum Signal {
        //定义一个枚举类型
        GREEN,YELLOW,RED;
    public static void main(String[] args)
    {
        for(int i=0;i<Signal.values().length;i++)
        {
            System.out.println("枚举成员："+Signal.values()[i]);
        }
    }
}
```

结果

```
//枚举成员：GREEN
//枚举成员：YELLOW
//枚举成员：RED
```

valueOf方法

通过字符串获取单个枚举对象

```
public enum Signal {
        //定义一个枚举类型
        GREEN,YELLOW,RED;
        public static void main(String[] args)
        {
            Signal green = Signal.valueOf("GREEN");
            System.out.println(green);
        }
}
```

结果

```
//GREEN
```

ordinal() 方法

通过调用枚举类型实例的 ordinal() 方法可以获取一个成员在枚举中的索引位置。下面的示例创建一个包含 3 个成员的枚举类型 Signal，然后调用 ordinal() 方法输出成员及对应索引位置。

```
public class TestEnum1
{
    enum Signal
    {
        //定义一个枚举类型
        GREEN,YELLOW,RED;
    }
```

```
    public static void main(String[] args)
    {
        for(int i=0;i<Signal.values().length;i++)
        {
            System.out.println("索引"+Signal.values()[i].ordinal()+",
值："+Signal.values()[i]);
        }
    }
}
```

结果

```
//索引0，值：GREEN
//索引1，值：YELLOW
//索引2，值：RED
```

6、注解 单元测试

注解 (Annotation) 概述

从 JDK 5.0 开始，Java 增加了对元数据(MetaData) 的支持，也就是Annotation(注解).

Annotation 其实就是代码里的特殊标记，这些标记可以在编译，类加载，运行时被读取，并执行相应的处理。通过使用 Annotation，程序员可以在不改变原有逻辑的情况下，在源文件中嵌入一些补充信息。代码分析工具、开发工具和部署工具可以通过这些补充信息进行验证或者进行部署。

Annotation 可以像修饰符一样被使用，可用于修饰包,类，构造器，方法，成员变量，参数，局部变量的声明，这些信息被保存在Annotation 的“name=value”对中。

在JavaSE中，注解的使用目的比较简单，例如标记过时的功能，忽略警告等。在JavaEE/Android中注解占据了更重要的角色，例如：用来配置应用程序的任何切面，代替JavaEE旧版中所遗留的繁冗代码和XML配置等。

未来的开发模式都是基于注解的，JPA是基于注解的，Spring2.5以上都是基于注解的，Hibernate3.x以后也是基于注解的，现在的Struts2有一部分也是基于注解的了，注解是一种趋势，一定程度上可以说：框架 = 注解 + 反射 + 设计模式。

常见的Annotation示例

使用 Annotation 时要在其前面增加 @ 符号，并把该 Annotation 当成一个修饰符使用。用于修饰它支持的程序元素。

示例一：生成文档相关的注解

```
    @author 标明开发该类模块的作者，多个作者之间使用,分割;
    @version 标明该类模块的版本;
    @see 参考转向，也就是相关主题;
```

```
    @since 从哪个版本开始增加的;
    @param 对方法中某参数的说明，如果没有参数就不能写;
    @return 对方法返回值的说明，如果方法的返回值类型是void就不能写;
    @exception 对方法可能抛出的异常进行说明 ，如果方法没有用throws显式抛出的异常就不能写;

    其中
        @param @return 和 @exception 这三个标记都是只用于方法的;
        @param的格式要求：@param 形参名 形参类型 形参说明;
        @return 的格式要求：@return 返回值类型 返回值说明;
        @exception的格式要求：@exception 异常类型 异常说明;
        @param和@exception可以并列多个;



/**
 * @author jimbo
 * @version 1.0
 */
public class JavadocTest {
    /**
     * 程序的主方法，程序的入口
     *
     * @param args String[] 命令行参数
     */
    public static void main(String[] args) {
    }

    /**
     * 求圆面积的方法
     *
     * @param radius double 半径值
     * @return double 圆的面积
     */
    public static double getArea(double radius) {
        return Math.PI * radius * radius;
    }
}
```

示例二：在编译时进行格式检查(JDK内置的三个基本注解)

```
    @Override: 限定重写父类方法，该注解只能用于方法;

    @Deprecated: 用于表示所修饰的元素(类，方法等)已过时。通常是因为所修饰的结构危险或存在更好的选择;

    @SuppressWarnings: 抑制编译器警告;



public class AnnotationTest {
    public static void main(String[] args) {
        @SuppressWarnings("unused")
        int a = 10;
```

```
        @SuppressWarnings({ "unused", "rawtypes" })
        ArrayList list = new ArrayList();
    }

    @Deprecated
    public void print() {
        System.out.println("过时的方法");
    }

    @Override
    public String toString() {
        return "重写的toString方法()";
    }
}
```

示例三：跟踪代码依赖性，实现替代配置文件功能

```
    Servlet3.0提供了注解(annotation),使得不再需要在web.xml文件中进行Servlet的部署。



@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws
        ServletException, IOException { }

    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws
        ServletException, IOException {
        doGet(request, response);
    }
}



<servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>com.servlet.LoginServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/login</url-pattern>
</servlet-mapping>



     spring框架中关于“事务”的管理。



@Transactional(propagation=Propagation.REQUIRES_NEW,
               isolation=Isolation.READ_COMMITTED,readOnly=false,timeout=3)
```

```
public void buyBook(String username, String isbn) {
    //1.查询书的单价
    int price = bookShopDao.findBookPriceByIsbn(isbn);
    //2. 更新库存
    bookShopDao.updateBookStock(isbn);
    //3. 更新用户的余额
    bookShopDao.updateUserAccount(username, price);
}
```

```
<!-- 配置事务属性 -->
<tx:advice transaction-manager="dataSourceTransactionManager" id="txAdvice">
    <tx:attributes>
        <!-- 配置每个方法使用的事务属性 -->
        <tx:method name="buyBook" propagation="REQUIRES_NEW"
                   isolation="READ_COMMITTED" read-only="false" timeout="3" />
    </tx:attributes>
</tx:advice>
```

自定义 Annotation

定义新的 Annotation 类型使用 @interface 关键字；

自定义注解自动继承了java.lang.annotation.Annotation接口；

Annotation 的成员变量在 Annotation 定义中以无参数方法的形式来声明。其方法名和返回值定义了该成员的名字和类型。我们称为配置参数。类型只能是八种基本数据类型、String类型、Class类型、enum类型、Annotation类型、以上所有类型的数组。

可以在定义 Annotation 的成员变量时为其指定初始值，指定成员变量的初始值可使用 default 关键字；

如果只有一个参数成员，建议使用参数名为value；

如果定义的注解含有配置参数，那么使用时必须指定参数值，除非它有默认值。格式是“参数名 = 参数值”，如果只有一个参数成员，且名称为value，可以省略“value=”；

没有成员定义的 Annotation 称为标记；包含成员变量的 Annotation 称为元数据 Annotation；

注意：

自定义注解必须配上注解的信息处理流程(使用反射)才有意义。

自定义注解通常都会指明两个元注解：Retention、Target。

```
import java.lang.annotation.*;

import static java.lang.annotation.ElementType.*;

@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
public @interface MyAnnotation {
```

```
    String value() default "hello";
}

// 如果定义的注解含有配置参数，那么使用时必须指定参数值，除非它有默认值。
// @MyAnnotation(value = "hi")
@MyAnnotation
class Person {
    private String name;
    private int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void walk() {
        System.out.println("人走路");
    }

    public void eat() {
        System.out.println("人吃饭");
    }
}
```

JDK 中的元注解

```
    JDK 的元 Annotation 用于修饰其他 Annotation 定义;

    JDK5.0提供了4个标准的meta-annotation类型，分别是:
        Retention
        Target
        Documented
        Inherited
```

@Retention元注解

@Retention: 只能用于修饰一个 Annotation 定义，用于指定该 Annotation 的生命周期，@Rentention 包含一个RetentionPolicy 类型的成员变量，使用@Rentention 时必须为该 value 成员变量指定值：

RetentionPolicy.SOURCE:在源文件中有效（即源文件保留），编译器直接丢弃这种策略的注释；

RetentionPolicy.CLASS:在class文件中有效（即class保留），当运行 Java 程序时，JVM 不会保留注解。 这是默认值；

RetentionPolicy.RUNTIME:在运行时有效（即运行时保留），当运行 Java 程序时，JVM 会保留注释。程序可以通过反射获取该注释。


```
public enum RetentionPolicy {
    /**
     * Annotations are to be discarded by the compiler.
     * 编译器将丢弃注释。
     */
    SOURCE,

    /**
     * Annotations are to be recorded in the class file by the compiler
     * but need not be retained by the VM at run time.  This is the default
     * behavior.
     * 注释将由编译器记录在类文件中但不需要在运行时由VM保留。这是默认设置行为
     */
    CLASS,

    /**
     * Annotations are to be recorded in the class file by the compiler and
     * retained by the VM at run time, so they may be read reflectively.
     *
     * @see java.lang.reflect.AnnotatedElement
     * 注释将由编译器记录在类文件中，并且由VM在运行时保留，因此可以反射地读取它们。
     */
    RUNTIME
}

@Retention(RetentionPolicy.SOURCE)
@interface MyAnnotation1{ }

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation2{ }
```

@Target元注解

@Target: 用于修饰 Annotation 定义，用于指定被修饰的 Annotation 能用于修饰哪些程序元素。 @Target 也包含一个名为 value 的成员变量。

取值(ElementType)		取值(ElementType)	
CONSTRUCTOR	用于描述构造器	PACKAGE	用于描述包
FIELD	用于描述域	PARAMETER	用于描述参数
LOCAL_VARIABLE	用于描述局部变量	TYPE	用于描述类、接口(包括注解类型) 或enum声明
METHOD	用于描述方法		

```
public enum ElementType {
    /** Class, interface (including annotation type), or enum declaration */
    // 类、接口（包括注释类型）或枚举声明
    TYPE,

    /** Field declaration (includes enum constants) */
    // 字段声明（包括枚举常量）
    FIELD,

    /** Method declaration */
    // 方法声明
    METHOD,

    /** Formal parameter declaration */
    // 形式参数声明
    PARAMETER,

    /** Constructor declaration */
    // 构造函数声明
    CONSTRUCTOR,

    /** Local variable declaration */
    // 局部变量声明
    LOCAL_VARIABLE,

    /** Annotation type declaration */
    // 注释类型声明
    ANNOTATION_TYPE,

    /** Package declaration */
    // 包声明
    PACKAGE,

    /**
     * Type parameter declaration
     *
     * @since 1.8
     */
    TYPE_PARAMETER,
```

```
    /**
     * Use of a type
     *
     * @since 1.8
     */
    TYPE_USE
}

// 我们移除MyAnnotation @Target中的TYPE,如果再在类上定义该注解，则报错
@Target({FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, TYPE_PARAMETER,
TYPE_USE})
public @interface MyAnnotation {
    String value() default "hello";
}


// 报错信息：'@MyAnnotation' not applicable to type
```

@Documented元注解

```
    @Documented: 用于指定被该元 Annotation 修饰的 Annotation 类将被javadoc 工具提取成
文档。默认情况下，javadoc是不包括注解的。
        定义为Documented的注解必须设置Retention值为RUNTIME。
    注：详情见Date API：



@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER,
TYPE})
public @interface Deprecated {

}
```

@Inherited元注解

```
    @Inherited: 被它修饰的 Annotation 将具有继承性。如果某个类使用了被@Inherited 修饰的
Annotation, 则其子类将自动具有该注解。
        比如：如果把标有@Inherited注解的自定义的注解标注在类级别上，子类则可以继承父类类级别
的注解。
        实际应用中，使用较少。



import java.lang.annotation.*;

import static java.lang.annotation.ElementType.*;
```

```
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
public @interface MyAnnotation {
    String value() default "hello";
}



@MyAnnotation(value="hi")
class Person{
    private String name;
    private int age;

    public Person() {
    }
    @MyAnnotation
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @MyAnnotation
    public void walk(){
        System.out.println("人走路");
    }
    public void eat(){
        System.out.println("人吃饭");
    }
}

class Student extends Person{

}

class Test{
    public static void main(String[] args) {
        // 利用反射测试子类是否继承父类的注解
        Class clazz = Student.class;
        Annotation[] annotations = clazz.getAnnotations();
        for(int i = 0;i < annotations.length;i++){
            System.out.println(annotations[i]);
        }
    }
}
```

利用反射获取注解信息

JDK 5.0 在 java.lang.reflect 包下新增了 AnnotatedElement 接口，该接口代表程序中可以接受注解的程序元素。

当一个 Annotation 类型被定义为运行时 Annotation 后，该注解才是运行时可见，当 class 文件被载入时保存在 class 文件中的 Annotation 才会被虚拟机读取。

程序可以调用 AnnotatedElement对象的如下方法来访问 Annotation 信息。

方法摘要	
<T extends Annotation> T	getAnnotation(Class<T> annotationClass) 如果存在该元素的指定类型的注释，则返回这些注释，否则返回 null。
Annotation[]	getAnnotations() 返回此元素上存在的所有注释。
Annotation[]	getDeclaredAnnotations() 返回直接存在于此元素上的所有注释。
boolean	isAnnotationPresent(Class<? extends Annotation> annotationClass) 如果指定类型的注释存在于此元素上，则返回 true，否则返回 false。

7、Lambda表达式

Lambda 表达式简介

Lambda 表达式是 JDK8 的一个新特性，可以取代大部分的匿名内部类，写出更优雅的 Java 代码，尤其在集合的遍历和其他集合操作中，可以极大地优化代码结构。

JDK 也提供了大量的内置函数式接口供我们使用，使得 Lambda 表达式的运用更加方便、高效。

Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。

Lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中）。

使用 Lambda 表达式可以使代码变的更加简洁紧凑。

在JDK8之前，一个方法能接受的参数都是变量，例如： object.method(Object o);那么，如果需要传入一个动作呢？比如回调,那么你可能会想到匿名内部类。

例如：匿名内部类是需要依赖接口的，所以需要先定义个接口;

```
public interface PersonCallBack {

    void callBack(Person person);

}
```

Person类

```
public class Person {
    private int id;

    private String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    /**
     * 创建一个Person后进行回调
     *
     * @param id
     * @param name
     * @param personCallBack
     */
    public static void create(Integer id, String name, PersonCallBack
personCallBack) {
        Person person = new Person(id, name);
        personCallBack.callBack(person);
    }

}
```

调用方法：

```
public class Test {
    public static void main(String[] args) {
        // 调用方法，传入回调类，传统方式，使用匿名内部类
        Person.create(1, "zhangsan", new PersonCallBack() {
            @Override
            public void callBack(Person person) {
                System.out.println("callback -- " + person.getName());
            }
        });
    }
}
```

上面的PersonCallBack其实就是一种动作，但是我们真正关心的只有callback方法里的内容而已，我们用Lambda表示，可以将上面的代码就可以优化成：

```
public class Test {
    public static void main(String[] args) {
        //调用方法，传入回调类，传统方式，使用匿名内部类
        Person.create(1, "zhangsan", new PersonCallBack() {
            @Override
            public void callBack(Person person) {
                System.out.println("callback -- " + person.getName());
            }
        });

        //使用lambda表达式实现
        Person.create(2, "lisi", (person) -> {
            System.out.println("lambda callback -- " + person.getName());
        });

        //进一步简化：这归功于Java8的类型推导机制。因为现在接口里只有一个方法，那么现在这个Lambda表达式肯定是对应实现了这个方法，既然是唯一的对应关系，那么入参肯定是Person类，所以可以简写，
        //并且方法体只有唯一的一条语句，所以也可以简写，以达到表达式简洁的效果。
        Person.create(3, "wangwu", person ->
                System.out.println("lambda callback -- " + person.getName())
        );
    }
}
```

Lambda允许把函数作为一个方法的参数，一个lambda由用逗号分隔的参数列表、-> 符号、函数体三部分表示。

一个Lambda表达式实现了接口里的有且仅有的唯一一个抽象方法。那么对于这种接口就叫做函数式接口。

Lambda表达式其实完成了实现接口并且实现接口里的方法这一功能，也可以认为Lambda表达式代表一种动作，我们可以直接把这种特殊的动作进行传递。

为什么使用Lambda表达式

首先说一下为什么要使用Lambda表达式？

在回答这个问题之前我先举个例子：在这里有一个员工的集合，员工类也就是id、age、name、salary这几个属性，无参，有参的构造函数，然后实现他们setter和getter方法。

员工类:

```
public class Employee {
    private Integer id;
    private String name;
    private Integer age;
    private Double salary;

    public Employee() {
    }

    public Employee(Integer id, String name, Integer age, Double salary) {
        super();
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Double getSalary() {
        return salary;
    }

    public void setSalary(Double salary) {
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
```

```
                ", salary=" + salary +
                '}';
    }

}
```

存储员工的集合:

```
import java.util.Arrays;
import java.util.List;

public class EmployeeTest {
    public static void main(String[] args) {
        //存储员工的集合
        List<Employee> emps = Arrays.asList(
                new Employee(101, "张三", 18, 9999d),
                new Employee(102, "李四", 59, 6666d),
                new Employee(103, "王五", 18, 3333d),
                new Employee(104, "赵六", 8, 7777d),
                new Employee(105, "田七", 28, 5555d)
        );

    }
}
```

现在有这么一个需求，就是获取员工集合中年龄大于20岁的员工的集合，在之前我们大部分我想应该都是这样写的吧:

```
public static List<Employee> filterEmployeeAge(List<Employee> emps) {
    List<Employee> list = new ArrayList<>();
    for (Employee emp : emps) {
        if (emp.getAge() > 20) {
            list.add(emp);
        }
    }
    return list;
}
```

ok，这样是能很完美的解决问题，很符合现实的逻辑，但是现在我们对这段代码进行优化，在优化的过程中会慢慢演变到主题Lambda的使用。

优化方式一：策略模式

创建一个泛型接口，接口里面只有一个方法。

```
public interface MyPredicate<T> {
    boolean test(T t);
}
```

再写一个接口的实现类：

```
public class FilterEmployeeForAge implements MyPredicate<Employee> {
    @Override
    public boolean test(Employee t) {
        return t.getAge() > 20;
    }
}
```

接下来就是优化的关键代码：

```
public static List<Employee> filterEmployee(List<Employee> emps,
MyPredicate<Employee> mp) {
    List<Employee> list = new ArrayList<>();
    for (Employee employee : emps) {
        if (mp.test(employee)) {
            list.add(employee);
        }
    }
    return list;
}
```

测试代码：

```
List<Employee> employees = filterEmployee(emps, new FilterEmployeeForAge());

for (Employee employee : list) {
    System.out.println(employee);
}
```

上面的测试代码的关键方法是 `filterEmployee` ，通过传入的员工列表和你过滤的方式 ，而 `FilterEmployeeForAge` 是 `MyPredicate` 的实现类，在 `filterEmployee` 中对每个员工进行判断，然后决定是否要加入到集合中。

如果你下次判断的不是age而是salary，就可以另外写一个MyPredicate的实现类，实现你自己想要的过滤的方式就可以了。

其实上面的优化方案就是设计模式中的策略设计模式。

上面是优化方式一的实现的一个思路，现在我在说一下第二种优化方案：

优化方式二：匿名内部类

```
List<Employee> list = filterEmployee(emps, new MyPredicate<Employee>() {
    @Override
    public boolean test(Employee t) {
        return t.getAge() > 20;
    }
});

for (Employee employee : list) {
    System.out.println(employee);
}
```

通过匿名内部类就不需要在去写接口的实现类了，看到这里，小伙伴们是不是着急了，想看看我们表达式怎么用呢？好的，接下来我就说一下Lambda表达式怎么使用。

优化方式三：Lambda表达式

```
List<Employee> list = filterEmployee(emps, t -> t.getAge() > 20);

list.forEach(System.out::println);
```

这样代码是不是很简洁。

教你看懂System.out::println

在不经意间， 我们会看到这样的代码。

```
List<String> list = new ArrayList<>();
list.add("a");
list.add("b");
list.add("c");
list.forEach(System.out::println);
```

第一印象， 哇， 好高大上的写法， 那么这究竟是怎样的一种语法呢。

其中list.forEach可以改写成以下代码：

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}

//或者等于以下代码：
for (String s : list) {
    System.out.println(s);
}
```

我们来看forEach方法的源码：

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

其实它本质上是调用了for循环，它的参数的是一个Consumer对象，在for循环中调用了对象的accept方法，注意该方法是接口的default方法，是JDK8新增的语法。

至于为什么可以写成System.out::println，这种语法叫做方法引用。该功能特性也是JDK8以后引入的，你可以把它看做Lambda表达式的语法糖。如果我们不这样写的话，我们可以用Lambda表达式改写成以下代码：

```
list.forEach((t) -> System.out.println(t));
```

如果还不明白的话，也可以这样：

```
list.forEach((String t) -> System.out.println(t));
```

这样的效果跟System.out::println是一样的。

Lambda 表达式的基础语法

Lambda 表达式的基础语法：Java8中引入了一个新的操作符 "->" 该操作符称为箭头操作符 或 Lambda 操作符，箭头操作符将 Lambda 表达式拆分成两部分：

左侧：Lambda 表达式的参数列表；

右侧：Lambda 表达式中所需执行的功能，即 Lambda 体；

我们可以这样去进行比对，Lambda表达式是用来替换匿名内部类（基于接口，函数式接口- 只要一个抽象方法），Lambda 表达式的参数列表其实就是函数式接口抽象方法定义的参数个数；那么右侧Lambda表达式所需执行的功能就是以前在抽象方法里面写的代码。

对应的每种语法，这里都会给出一个例子，方便大家的理解。

语法格式一

无参数，无返回值。

语法：

```
() -> System.out.println("Hello Lambda!");
```

```
@Test
public void test1() {
    // jdk1.8之前,匿名内部类写法
    Runnable r = new Runnable() {
        @Override
        public void run() {
            System.out.println("Hello World!");
        }
    };

    r.run();

    System.out.println("--------------------------------");
    // jdk1.8之后,Lambda写法
    Runnable r1 = () -> System.out.println("Hello Lambda!");
    r1.run();
}
```

语法格式二

有一个参数，并且无返回值。

语法：

```
(x) -> System.out.println(x)
```

```
@Test
public void test2() {
    Consumer<String> con1 = new Consumer<String>() {
```

```
        @Override
        public void accept(String t) {
            System.out.println(t);
        }

    };
    con1.accept("欢迎使用匿名内部类方式! ");

    System.out.println("-------------------------------");

    Consumer<String> con = (x) -> System.out.println(x);
    con.accept("欢迎使用Lamdba表示式! ");
}
```

语法格式三

```
    若只有一个参数，小括号可以省略不写。

    语法：
        x -> System.out.println(x)



@Test
public void test2(){
    Consumer<String> con = x -> System.out.println(x);
    con.accept("欢迎使用Lamdba表示式! ");
}
```

语法格式四

```
    有两个以上的参数，有返回值，并且 Lambda 体中有多条语句。

    语法：
        (x, y) -> {
            System.out.println("多条语句 1");
            System.out.println("多条语句 2");
            return x + y;
        };



@Test
public void test3() {
    Comparator<Integer> com1 = new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            // TODO Auto-generated method stub
            return Integer.compare(o1, o2);
        }
```

```
    };
    int compare1 = com1.compare(10, 20);
    System.out.println("使用匿名内部类方式比较:" + compare1);

    System.out.println("------------------------------");

    Comparator<Integer> com = (x, y) -> {
        System.out.println("函数式接口");
        return Integer.compare(x, y);
    };
    int compare = com.compare(10, 20);
    System.out.println("使用Lamdba表达式方式比较:" + compare);

}
```

语法格式五

```
    若 Lambda 体中只有一条语句， return 和 大括号都可以省略不写。

    语法：
        (x, y) -> x > y ? x : y;



@Test
public void test4(){
    Comparator<Integer> com = (x, y) -> Integer.compare(x, y);
}
```

语法格式六

```
    Lambda 表达式的参数列表的数据类型可以省略不写，因为JVM编译器通过上下文推断出，数据类型，即“类型推断”。

    语法：
        (Integer x, Integer y) -> Integer.compare(x, y);
        等价于
        (x,y) -> Integer.compare(x,y)
```

ok，讲到这里语法就差不多了在这里了，如果掌握了这些我想应该能解决平时正常的需求，这里有一副对联：

上联：左右遇一括号省
下联：左侧推断类型省
横批：能省则省

这里讲一个大家需要注意的地方：`Lambda` 表达式需要“函数式接口”的支持。

8、JVM调优

- 为什么要JVM调优?

> 减少GC，特别是FullGC
>
> 1. 对象优先在堆的 Eden 区分配
> 2. 大对象直接进入老年代
> 3. 长期存活的对象将直接进入老年代. 当 Eden 区没有足够的空间进行分配时，虚拟机会执行一次 Minor GC.Minor Gc 通 常发生在新生代的 Eden 区，在这个区的对象生存期短，往往发生 Gc 的频率较高， 回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代 GC 的时候不会触发 Minor GC,但是通过配置，可以在 Full GC 之前进行一次 Minor GC 这样可以加快老年代的回收速度。
>
> - GC 的两种判定方法：
>
> 1. 引用计数法：指的是如果某个地方引用了这个对象就+1，如果失效了就-1，当为 0 就会回收但是 JVM 没有用这种方式，因为无法判定相互循环引用（A 引用 B,B 引用 A）的情况
> 2. 引用链法： 通过一种 GC ROOT 的对象（方法区中静态变量引用的对象等-static 变量）来判断，如果有一条链能够到达 GC ROOT 就说明，不能到达 GC ROOT 就说明 可以回收

JVM的垃圾回收器和内存分配

1. 串行垃圾回收器

> 是指使用单线程进行垃圾回收的回收器，每次回收只有一个工作线程(对并行能力比较弱的电脑，运行性能较好)
>
> 注意：串行回收器运行时，所有`应用程序的线程都停止工作，属于独占式的垃圾回收方式`；
>
> 线程进行等待的现象称为–>Stop-The-World,造成非常糟糕的用户体验;

1. 并行垃圾回收器

> 多个线程同时进行垃圾回收，适合并行能力强的计算机；
>
> 1. 新生代ParNew回收器，只是简单将串行回收器多线程化，也是独占式的回收器
> 2. 还有其他的回收器，都`关注吞吐量`，其中包括复制算法，标记压缩算法等回收算法
>
> 注意：还是会造成线程等待现象–>Stop-The-World（STW），但是减少垃圾回收的停顿时间就会同时减小系统的吞吐量

1. CMS回收器(jdk1.8以前)

> > CMS回收器主要关注`系统的停顿时间`，并发标记清除，是一个`基于标记清除算法的回收器`；
> >
> > CMS掉工作过程相对复杂，不是独占式的回收器，工作过程中，应用程序仍然工作;
> >
> > 不会等到堆内存饱和后进行回收，而是到达一定阈值才开始垃圾回收

参数	说明
-XX:CMSInitiatingOccupancyFraction	默认堆老年代使用达到68%,执行CMS回收,如果在执行过程中内存不足，就会启动串行回收器进行垃圾回收,应用程序将完全中断;
	根据此参数进行调优,增大阈值可以降低CMS的触发,减少老年代的回收次数；如果内存使用增长很快，应该降低阈值;

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的垃圾收集器。从名字可以看出，CMS 是基于标记-清除算法的。它的运作过程主要分为四个步骤：

1. 初始标记（CMS initial mark）：**STW**，标记GC Roots能直接关联到的对象，速度很快。单线程
2. 并发标记（CMS concurrent mark）：从GC Roots的直接关联对象开始遍历整个对象图的过程，耗时较长，不需要停顿用户线程
3. 重新标记（CMS remark）：**STW**，修正并发标记期间，因用户程序继续运作而导致标记发生变动的那一部分对象的标记记录（增量更新），时间稍长于初始标记，但远低于并发标记
4. 并发清除（CMS concurrent sweep）：清除已死亡对象，因为不需要移动对象，所以与用户线程是并发的关系

G1回收器(JDK1.7推出,1.9默认)

Garbages First(G1)垃圾回收器，作为CMS的长久替代方案，使用了全新的分区算法；

1. 并行性：G1回收期间，多个GC线程可以同时工作
2. 并发性：G1可以跟应用程序交替执行的能力，不会在回收期间完全阻塞应用程序
3. 分代GC：G1依然是一个分代回收器，与之前回收器不同，G1兼顾年轻代和老年代，如CMS工作在老年代
4. 空间整理：G1回收过程中，会适当进行对象移动，如CMS若干次GC后，CMS必须进行一次碎片整理，但是G1，每次回收都会有效复制对象，减少碎片空间；
5. 可预见性：由于分区原因，G1只对选取的部分区域进行回收。可以很好的控制全局停顿；

新生代GC(主要回收eden区和survivor区,复制算法)

eden区被占满，新生代GC启动，`ednn区会被全部回收`，至少存在一个survivor区，老年代区域增大；

G1的并发标记周期

1. 初始化标记：标记从根节点直接可达的对象，发生一次新生代GC，产生全局停顿
2. 根区域扫描：eden区已经清空，扫描标记由survivor区直接可达的老年代区域,无法与新生代GC同时执行
3. 并发标记：与CMS类似，全局标记堆中存活的对象
4. 重新标记：G1会使用(SATB)算法，为存活对象创建快照，有助于加速重新标记速度
5. 独占清理：计算各个区域存活对象和GC回收比例，识别可供混合回收的区域
6. 并发清理：会识别并清理空闲区域，并发清理不会引起停顿

混合回收

在并发标记后就知道哪个区域的垃圾较多，G1就会优先回收垃圾比例高的区域

FullGC

> 1. 堆内存不足时，就会触发FullGC,
> 2. 对于并行回收器的FullGC之前，都会触发一次新生代GC
> 3. 使用system.gc()方法，触发一次GC，在并行回收器中，FullGC之前会发生一个新生代GC，这样可以缩短停顿时间(STW)

对象进入老年代

1. 新对象在eden区
2. JVM提供一个参数来控制新生代对象的年龄（MaxTenuringThreshold），默认初始值为15，新生代对象最多经历15次GC就可以到老年代
3. 新生代无法容纳的大对象直接进入老年代，可以通过调节参数（PretenureSizeThreshold），设置对象晋升到老年代的阈值；

- 如使用的JVM测试参数(使用开发工具可以自定义)

```
-Xmx32m -Xms32m -XX:+UseSerialGC -XX:+PrintGCDetails
1
```

G1和CMS的区别

> **G1具备如下特点：**
>
> - **并行与并发**：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-the-world停顿的时间，部分其他收集器原来需要停顿Java线程执行的GC操作，G1收集器仍然可以通过**并发**的方式让Java程序继续运行。
> - 分代收集
> - 空间整合：与CMS的标记-清除算法不同，G1从整体来看是基于**标记-整理算法**实现的收集器，从局部（两个Region之间）上来看是基于“**复制**”算法实现的。但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。
> - 可预测的停顿：这是G1相对于CMS的一个优势，降低停顿时间是G1和CMS共同的关注点。

CMS 用于老年代的回收，而 G1 用于新生代和老年代的回收。

G1 使用了 Region 方式对堆内存进行了划分，且基于标记整理算法实现，整体减少了垃圾碎片的产生。CMS使用“标记-清理”算法会产生大量的空间碎片;

相关参数

3. 与 CMS 回收器相关的参数（JDK 9、JDK 10 已经开始废弃 CMS 回收器，建议使用 G1 回收器）

- -XX:+UseConcMarkSweepGC：新生代使用并行回收器，老年代使用 CMS+串行回收器。
- -XX:ParallelCMSThreads：设定 CMS 的线程数量。
- -XX:CMSInitiatingOccupancyFraction：设置 CMS 回收器在老年代空间被使用多少后触发，默认为 68%。
- -XX:+UseCMSCompactAtFullCollection：设置 CMS 回收器在完成垃圾回收后是否要进行一次内存碎片的整理。
- -XX:CMSFullGCsBeforeCompaction：设定进行多少次 CMS 垃圾回收后，进行一次内存压缩。
- -XX:+CMSClassUnloadingEnabled：允许对类元数据区进行回收。
- -XX:CMSInitiatingPermOccupancyFraction：当永久区占用率达到这一百分比时，启动 CMS 回收（前提是激活了-XX:+CMSClassUnloadingEnabled）。
- -XX:UseCMSInitiatingOccupancyOnly：表示只在到达阈值的时候才进行 CMS 回收。
- -XX:+CMSIncrementalMode：使用增量模式，比较适合单 CPU。增量模式在 JDK 8 中标记为废弃，并且将在 JDK 9 中彻底移除。

4. 与 G1 回收器相关的参数

- -XX:+UseG1GC：使用 G1 回收器。
- -XX:MaxGCPauseMillis：设置最大垃圾回收停顿时间。
- -XX:GCPauseIntervalMillis：设置停顿间隔时间。

5. TLAB 相关

- -XX:+UseTLAB：开启 TLAB 分配。

- -XX:+PrintTLAB（考虑到兼容性问题，JDK 9、JDK 10 不再支持此参数）：打印 TLAB 相关分配信息。
- -XX:TLABSize：设置 TLAB 区域大小。
- -XX:+ResizeTLAB：自动调整 TLAB 区域大小。

6. 其他参数

- -XX:+DisableExplicitGC：禁用显式 GC。
- -XX:+ExplicitGCInvokesConcurrent：使用并发方式处理显式 GC。

JVM调优很大程度上是对GC的调优，导出堆，对导出的dump文件进行分析，我们可以找到内存热点，可以找到哪个类型的对象数量最多，且占用的内存最多；哪个对象的体积大，还频繁被销毁创建；
但是JVM的调优本质实际上是通过JVM监控来分析JAVA代码的工作情况，找出不合理的设计和低质量的代码，进行改进；