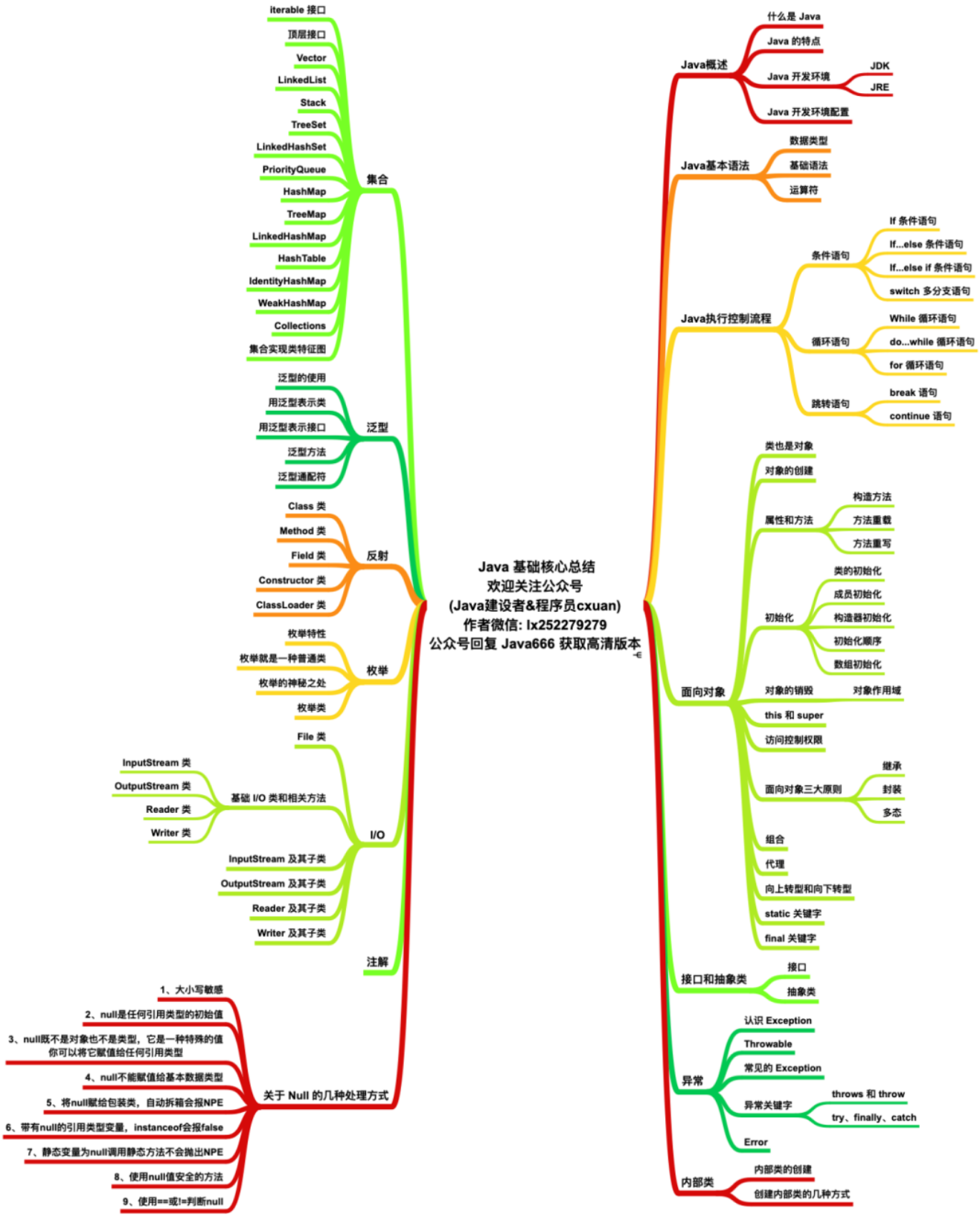

下面开始我们的文章。

Java 概述

什么是 Java?

Java 是 Sun Microsystems 于1995 年首次发布的一种 编程语言 和计算平台。编程语言还比较好理解，那么什么是 计算平台 呢?

> 计算平台是在电脑中运行应用程序（软件）的环境，包括 硬件环境 和 软件环境 。一般系统平台包括一台电脑的硬件体系结构、操作系统、运行时库。

Java 是快速，安全和可靠的。从笔记本电脑到数据中心，从游戏机到科学超级计算机，从手机到互联网，Java 无处不在！Java 主要分为三个版本

- JavaSE(J2SE)(Java2 Platform Standard Edition，java平台标准版)
- JavaEE(J2EE)(Java 2 Platform,Enterprise Edition，java平台企业版)
- JavaME(J2ME)(Java 2 Platform Micro Edition，java平台微型版)。

Java 的特点

- Java 是一门 面向对象 的编程语言

什么是面向对象？ 面向对象(Object Oriented) 是一种软件开发思想。它是对现实世界的一种抽象，面向对象会把相关的数据和方法组织为一个整体来看待。

相对的另外一种开发思想就是面向过程的开发思想，什么面向过程？ 面向过程(Procedure Oriented) 是一种以过程为中心的编程思想。举个例子：比如你是个学生，你每天去上学需要做几件事情?

起床、穿衣服、洗脸刷牙，吃饭，去学校。一般是顺序性的完成一系列动作。

```
class student {
  void student_wakeUp(){...}
  void student_cloth(){...}
  void student_wash(){...}
  void student_eating(){...}
  void student_gotoSchool(){...}
}
```

而面向对象可以把学生进行抽象，所以这个例子就会变为

```
class student(){
  void wakeUp(){...}
```

```
    void cloth(){...}
    void wash(){...}
    void eating(){...}
    void gotoSchool(){...}
}
```

可以不用严格按照顺序来执行每个动作。这是特点一。

- Java 摒弃了 C++ 中难以理解的多继承、指针、内存管理等概念；不用手动管理对象的生命周期，这是特征二。
- Java 语言具有功能强大和简单易用两个特征，现在企业级开发，快速敏捷开发，尤其是各种框架的出现，使 Java 成为越来越火的一门语言。这是特点三。
- Java 是一门静态语言，静态语言指的就是在编译期间就能够知道数据类型的语言，在运行前就能够检查类型的正确性，一旦类型确定后就不能再更改，比如下面这个例子。

```
public void foo() {
    int x = 5;
    boolean b = x;
}
```

静态语言主要有 **Pascal, Perl, C/C++, JAVA, C#, Scala** 等。

相对应的，动态语言没有任何特定的情况需要指定变量的类型，在运行时确定的数据类型。比如有**Lisp, Perl, Python、Ruby、JavaScript **等。

从设计的角度上来说，所有的语言都是设计用来把人类可读的代码转换为机器指令。动态语言是为了能够让程序员提高编码效率，因此你可以使用更少的代码来实现功能。静态语言设计是用来让硬件执行的更高效，因此需要程序员编写准确无误的代码，以此来让你的代码尽快的执行。从这个角度来说，静态语言的执行效率要比动态语言高，速度更快。这是特点四。

- Java 具有平台独立性和可移植性

Java 有一句非常著名的口号： Write once, run anywhere ，也就是一次编写、到处运行。为什么 Java 能够吹出这种牛批的口号来？核心就是 JVM 。我们知道，计算机应用程序和硬件之间会屏蔽很多细节，它们之间依靠操作系统完成调度和协调，大致的体系结构如下

那么加上 Java 应用、JVM 的体系结构会变为如下

Java 是跨平台的，已编译的 Java 程序可以在任何带有 JVM 的平台上运行。你可以在 Windows 平台下编写代码，然后拿到 Linux 平台下运行，该如何实现呢?

首先你需要在应用中编写 Java 代码;

用 Eclipse 或者 javac 把 Java 代码编译为 .class 文件;

然后把你的 .class 文件打成 .jar 文件；

然后你的 .jar 文件就能够在 Windows 、Mac OS X、Linux 系统下运行了。不同的操作系统有不同的 JVM 实现，切换平台时，不需要再次编译你的 Java 代码了。这是特点五。

- Java 能够容易实现多线程

Java 是一门高级语言，高级语言会对用户屏蔽很多底层实现细节。比如 Java 是如何实现多线程的。从操作系统的角度来说，实现多线程的方式主要有下面这几种

在用户空间中实现多线程

在内核空间中实现多线程

在用户和内核空间中混合实现线程

而我认为 Java 应该是在 用户空间 实现的多线程，内核是感知不到 Java 存在多线程机制的。这是特点六。

- Java 具有高性能

我们编写的代码，经过 javac 编译器编译称为 字节码(bytecode)，经过 JVM 内嵌的解释器将字节码转换为机器代码，这是解释执行，这种转换过程效率较低。但是部分 JVM 的实现比如 Hotspot JVM 都提供了 JIT(Just-In-Time) 编译器，也就是通常所说的动态编译□器，JIT 能够在运行时将□热点代码编译机器码，这种方式运行效率比较高，这是编译执行。所以 Java 不仅仅只是一种解释执行的语言。这是特点七。

- Java 语言具有健壮性

Java 的强类型机制、异常处理、垃圾的自动收集等是 Java 程序健壮性的重要保证。这也是 Java 与 C 语言的重要区别。这是特点八。

- Java 很容易开发分布式项目

Java 语言支持 Internet 应用的开发，Java 中有 net api，它提供了用于网络应用编程的类库，包括URL、URLConnection、Socket、ServerSocket等。Java的 RMI（远程方法激活） 机制也是开发分布式应用的重要手段。这是特点九。

Java 开发环境

JDK

JDK（Java Development Kit） 称为 Java 开发包或 Java 开发工具，是一个编写 Java 的 Applet 小程序和应用程序的程序开发环境。JDK是整个Java的核心，包括了 Java运行环境（Java Runtime Environment），一些 Java 工具 和 Java 的核心类库（Java API） 。

我们可以认真研究一下这张图，它几乎包括了 Java 中所有的概念，我使用的是 jdk1.8 ，可以点进去 Description of Java Conceptual Diagram ， 可以发现这里面包括了所有关于 Java 的描述

Oracle 提供了两种 Java 平台的实现，一种是我们上面说的 JDK，Java 开发标准工具包，一种是 JRE，叫做Java Runtime Environment，Java 运行时环境。JDK 的功能要比 JRE 全很多。

JRE

JRE 是个运行环境，JDK 是个开发环境。因此写 Java 程序的时候需要 JDK，而运行 Java 程序的时候就需要JRE。而 JDK 里面已经包含了JRE，因此只要安装了JDK，就可以编辑 Java 程序，也可以正常运行 Java 程序。但由于 JDK 包含了许多与运行无关的内容，占用的空间较大，因此运行普通的 Java 程序无须安装 JDK，而只需要安装 JRE 即可。

Java 开发环境配置

这个地方不再多说了，网上有很多教程配置的资料可供参考。

Java 基本语法

在配置完 Java 开发环境，并下载 Java 开发工具（Eclipse、IDEA 等）后，就可以写 Java 代码了，因为本篇文章是从头梳理 Java 体系，所以有必要从基础的概念开始谈起。

数据类型

在 Java 中，数据类型只有 四类八种

- 整数型：byte、short、int、long

byte 也就是字节，1 byte = 8 bits，byte 的默认值是 0；

short 占用两个字节，也就是 16 位，1 short = 16 bits，它的默认值也是 0；

int 占用四个字节，也就是 32 位，1 int = 32 bits，默认值是 0；

long 占用八个字节，也就是 64 位，1 long = 64 bits，默认值是 0L；

所以整数型的占用字节大小空间为 long > int > short > byte

- 浮点型

浮点型有两种数据类型：float 和 double

float 是单精度浮点型，占用 4 位，1 float = 32 bits，默认值是 0.0f；

double 是双精度浮点型，占用 8 位，1 double = 64 bits，默认值是 0.0d；

- 字符型

字符型就是 char，char 类型是一个单一的 16 位 Unicode 字符，最小值是 \u0000 (也就是 0)，最大值是 \uffff (即为 65535)，char 数据类型可以存储任何字符，例如 char a = 'A'。

- 布尔型

布尔型指的就是 boolean，boolean 只有两种值，true 或者是 false，只表示 1 位，默认值是 false。

以上 x 位 都指的是在内存中的占用。

基础语法

- 大小写敏感：Java 是对大小写敏感的语言，例如 Hello 与 hello 是不同的，这其实就是 Java 的字符串表示方式
- 类名：对于所有的类来说，首字母应该大写，例如 MyFirstClass
- 包名：包名应该尽量保证小写，例如 my.first.package
- 方法名：方法名首字母需要小写，后面每个单词字母都需要大写，例如 myFirstMethod()

运算符

运算符不只 Java 中有，其他语言也有运算符，运算符是一些特殊的符号，主要用于数学函数、一些类型的赋值语句和逻辑比较方面，我们就以 Java 为例，来看一下运算符。

- 赋值运算符

赋值运算符使用操作符 = 来表示，它的意思是把 = 号右边的值复制给左边，右边的值可以是任何常数、变量或者表达式，但左边的值必须是一个明确的，已经定义的变量。比如 int a = 4 。

但是对于对象来说，复制的不是对象的值，而是对象的引用，所以如果说将一个对象复制给另一个对象，实际上是将**一个对象的引用赋值给另一个对象**。

- 算数运算符

算数运算符就和数学中的数值计算差不多，主要有

算数运算符需要注意的就是 优先级问题 ，当一个表达式中存在多个操作符时，操作符的优先级顺序就决定了计算顺序，最简单的规则就是先乘除后加减， () 的优先级最高，没必要记住所有的优先级顺序，不确定的直接用 () 就可以了。

- 自增、自减运算符

这个就不文字解释了，解释不如直接看例子明白

```
int a = 5;
b = ++a;
c = a++;
```

- 比较运算符

比较运算符用于程序中的变量之间，变量和自变量之间以及其他类型的信息之间的比较。

比较运算符的运算结果是 boolean 型。当运算符对应的关系成立时，运算的结果为 true，否则为 false。比较运算符共有 6 个，通常作为判断的依据用于条件语句中。

- 逻辑运算符

逻辑运算符主要有三种，与、或、非

下面是逻辑运算符对应的 true/false 符号表

- 按位运算符

按位运算符用来操作整数基本类型中的每个 比特 位，也就是二进制位。按位操作符会对两个参数中对应的位执行布尔代数运算，并最终生成一个结果。

如果进行比较的双方是数字的话，那么进行比较就会变为按位运算。

按位与：按位进行与运算（AND），两个操作数中位都为1，结果才为1，否则结果为0。需要首先把比较双方转换成二进制再按每个位进行比较

按位或：按位进行或运算（OR），两个位只要有一个为1，那么结果就是1，否则就为0。

按位非：按位进行异或运算（XOR），如果位为0，结果是1，如果位为1，结果是0。

按位异或：按位进行取反运算（NOT），两个操作数的位中，相同则结果为0，不同则结果为1。

- 移位运算符

移位运算符用来将操作数向某个方向（向左或者右）移动指定的二进制位数。

- 三元运算符

三元运算符是类似 if...else... 这种的操作符，语法为：**条件表达式? 表达式 1：表达式 2。**问号前面的位置是判断的条件，判断结果为布尔型，为 true 时调用表达式 1，为 false 时调用表达式 2。

Java 执行控制流程

Java 中的控制流程其实和 C 一样，在 Java 中，流程控制会涉及到包括 **if-else、while、do-while、for、return、break** 以及选择语句 switch 。下面以此进行分析

条件语句

条件语句可根据不同的条件执行不同的语句。包括 if 条件语句与 switch 多分支语句。

if 条件语句

if 语句可以单独判断表达式的结果，表示表达的执行结果，例如

```
int a = 10;
if(a > 10){
  return true;
}
return false;
```

if...else 条件语句

if 语句还可以与 else 连用，通常表现为 **如果满足某种条件，就进行某种处理，否则就进行另一种处理。**

```
int a = 10;
int b = 11;
if(a >= b){
  System.out.println("a >= b");
}else{
  System.out.println("a < b");
}
```

if 后的 () 内的表达式必须是 boolean 型的。如果为 true，则执行 if 后的复合语句；如果为 false，则执行 else 后的复合语句。

if...else if 多分支语句

上面中的 if...else 是单分支和两个分支的判断，如果有多个判断条件，就需要使用 **if...else if**

```
int x = 40;
if(x > 60) {
  System.out.println("x的值大于60");
} else if (x > 30) {
  System.out.println("x的值大于30但小于60");
} else if (x > 0) {
  System.out.println("x的值大于0但小于30");
} else {
  System.out.println("x的值小于等于0");
}
```

switch 多分支语句

一种比 **if...else if ** 语句更优雅的方式是使用 `switch` 多分支语句，它的示例如下

```
switch (week) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
```

```
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("No Else");
        break;
}
```

循环语句

循环语句就是在满足一定的条件下反复执行某一表达式的操作，直到满足循环语句的要求。使用的循环语句主要有 **for、do...while() 、 while **，

while 循环语句

while 循环语句的循环方式为利用一个条件来控制是否要继续反复执行这个语句。while 循环语句的格式如下

```
while(布尔值){
  表达式
}
```

它的含义是，当 (布尔值) 为 true 的时候，执行下面的表达式，布尔值为 false 的时候，结束循环，布尔值其实也是一个表达式，比如

```
int a = 10;
while(a > 5){
  a--;
}
```

do...while 循环

while 与 do...while 循环的唯一区别是 do...while 语句至少执行一次，即使第一次的表达式为 false。而在 while 循环中，如果第一次条件为 false，那么其中的语句根本不会执行。在实际应用中，while 要比 do...while 应用的更广。它的一般形式如下

```
int b = 10;
// do…while循环语句
do {
  System.out.println("b == " + b);
  b--;
} while(b == 1);
```

for 循环语句

for 循环是我们经常使用的循环方式，这种形式会在第一次迭代前进行初始化。它的形式如下

```
for(初始化; 布尔表达式; 步进){}
```

每次迭代前会测试布尔表达式。如果获得的结果是 false，就会执行 for 语句后面的代码；每次循环结束，会按照步进的值执行下一次循环。

逗号操作符

这里不可忽略的一个就是逗号操作符，Java 里唯一用到逗号操作符的就是 for 循环控制语句。在表达式的初始化部分，可以使用一系列的逗号分隔的语句；通过逗号操作符，可以在 for 语句内定义多个变量，但它们必须具有相同的类型

```
for(int i = 1;j = i + 10;i < 5;i++, j = j * 2){}
```

for-each 语句

在 Java JDK 1.5 中还引入了一种更加简洁的、方便对数组和集合进行遍历的方法，即 for-each 语句，例子如下

```
int array[] = {7, 8, 9};

for (int arr : array) {
    System.out.println(arr);
}
```

跳转语句

Java 语言中，有三种跳转语句: **break、continue 和 return**

break 语句

break 语句我们在 switch 中已经见到了，它是用于终止循环的操作，实际上 break 语句在for、while、do…while循环语句中，用于强行退出当前循环，例如

```
for(int i = 0;i < 10;i++){
 if(i == 5){
   break;
  }
}
```

continue 语句

continue 也可以放在循环语句中，它与 break 语句具有相反的效果，它的作用是用于执行下一次循环，而不是退出当前循环，还以上面的例子为主

```
for(int i = 0;i < 10;i++){

  System.out.printl(" i = " + i );
 if(i == 5){
   System.out.printl("continue ... ");
   continue;
  }
}
```

return 语句

return 语句可以从一个方法返回，并把控制权交给调用它的语句。

```
public void getName() {
   return name;
}
```

面向对象

下面我们来探讨面向对象的思想，面向对象的思想已经逐步取代了过程化的思想 --- 面向过程，Java 是面向对象的高级编程语言，面向对象语言具有如下特征

- 面向对象是一种常见的思想，比较符合人们的思考习惯；
- 面向对象可以将复杂的业务逻辑简单化，增强代码复用性；
- 面向对象具有抽象、封装、继承、多态等特性。

面向对象的编程语言主要有：C++、Java、C#等。

所以必须熟悉面向对象的思想才能编写出 Java 程序。

类也是一种对象

现在我们来认识一个面向对象的新的概念 --- 类，什么是类，它就相当于是一系列对象的抽象，就比如书籍一样，类相当于是书的封面，大多数面向对象的语言都使用 `class` 来定义类，它告诉你它里面定义的对象都是什么样的，我们一般使用下面来定义类

```
class ClassName {
 // body;
}
```

代码段中涉及一个新的概念 `//` ，这个我们后面会说。上面，你声明了一个 class 类，现在，你就可以使用 new 来创建这个对象

```
ClassName classname = new ClassName();
```

一般，类的命名遵循 驼峰原则 ，它的定义如下

> 骆驼式命名法（Camel-Case）又称驼峰式命名法，是电脑程式编写时的一套命名规则（惯例）。正如它的名称 CamelCase 所表示的那样，是指混合使用大小写字母来构成变量和函数的名字。程序员们为了自己的代码能更容易的在同行之间交流，所以多采取统一的可读性比较好的命名方式。

对象的创建

在 Java 中，**万事万物都是对象**。这句话相信你一定不陌生，尽管一切都看作是对象，但是你操纵的却是一个对象的 引用(reference) 。在这里有一个很形象的比喻：你可以把车钥匙和车看作是一组**对象引用和对象**的组合。当你想要开车的时候，你首先需要拿出车钥匙点击开锁的选项，停车时，你需要点击加锁来锁车。车钥匙相当于就是引用，车就是对象，由车钥匙来驱动车的加锁和开锁。并且，即使没有车的存在，车钥匙也是一个独立存在的实体，也就是说，**你有一个对象引用，但你不一定需要一个对象与之关联**，也就是

```
Car carKey;
```

这里创建的只是引用，而并非对象，但是如果你想要使用 s 这个引用时，会返回一个异常，告诉你需要一个对象来和这个引用进行关联。一种安全的做法是，在创建对象引用时同时把一个对象赋给它。

```
Car carKey = new Car();
```

在 Java 中，一旦创建了一个引用，就希望它能与一个新的对象进行关联，通常使用 new 操作符来实现这一目的。new 的意思是，给我一个新 对象 ，如果你不想相亲，自己 new 一个对象就好了。祝你下辈子幸福。

属性和方法

类一个最基本的要素就是有属性和方法。

属性也被称为字段，它是类的重要组成部分，属性可以是任意类型的对象，也可以是基本数据类型。例如下

```
class A{
```

```
    int a;
    Apple apple;
}
```

类中还应该包括方法，方法表示的是 **做某些事情的方式**。方法其实就是函数，只不过 Java 习惯把函数称为方法。这种叫法也体现了面向对象的概念。

方法的基本组成包括 **方法名称、参数、返回值和方法体**， 下面是它的示例

```
public int getResult(){
    // ...
    return 1;
}
```

其中， getResult 就是方法名称、 () 里面表示方法接收的参数、 return 表示方法的返回值，注意：方法的返回值必须和方法的 参数 类型保持一致。有一种特殊的参数类型 --- void 表示方法无返回值。 {} 包含的代码段被称为方法体。

构造方法

在 Java 中，有一种特殊的方法被称为 构造方法 ，也被称为构造函数、构造器等。在 Java 中，通过提供这个构造器，来确保每个对象都被初始化。构造方法只能在对象的创建时期调用一次，保证了对象初始化的进行。构造方法比较特殊，它没有参数类型和返回值，它的名称要和类名保持一致，并且构造方法可以有多个，下面是一个构造方法的示例

```
class Apple {

    int sum;
    String color;

    public Apple(){}
    public Apple(int sum){}
    public Apple(String color){}
    public Apple(int sum,String color){}

}
```

上面定义了一个 Apple 类，你会发现这个 Apple 类没有参数类型和返回值，并且有多个以 Apple 同名的方法，而且各个 Apple 的参数列表都不一样，这其实是一种多态的体现，我们后面会说。在定义完成构造方法后，我们就能够创建 Apple 对象了。

```
class createApple {

    public static void main(String[] args) {
        Apple apple1 = new Apple();
        Apple apple2 = new Apple(1);
        Apple apple3 = new Apple("red");
        Apple apple4 = new Apple(2,"color");

    }
}
```

如上面所示，我们定义了四个 Apple 对象，并调用了 Apple 的四种不同的构造方法，其中，不加任何参数的构造方法被称为默认的构造方法，也就是

```
Apple apple1 = new Apple();
```

如果类中没有定义任何构造方法，那么 JVM 会为你自动生成一个构造方法，如下

```
class Apple {

  int sum;
  String color;

}

class createApple {

    public static void main(String[] args) {
        Apple apple1 = new Apple();

    }
}
```

上面代码不会发生编译错误，因为 Apple 对象包含了一个默认的构造方法。

默认的构造方法也被称为默认构造器或者无参构造器。

这里需要注意一点的是，即使 JVM 会为你默认添加一个无参的构造器，但是如果你手动定义了任何一个构造方法，**JVM 就不再为你提供默认的构造器，你必须手动指定，否则会出现编译错误。**

显示的错误是，必须提供 Apple 带有 int 参数的构造函数，而默认的无参构造函数没有被允许使用。

方法重载

在 Java 中一个很重要的概念是方法的重载，它是类名的不同表现形式。我们上面说到了构造函数，其实构造函数也是重载的一种。另外一种就是方法的重载

```
public class Apple {

    int sum;
    String color;

    public Apple(){}
    public Apple(int sum){}

    public int getApple(int num){
        return 1;
    }

    public String getApple(String color){
        return "color";
    }

}
```

如上面所示，就有两种重载的方式，一种是 Apple 构造函数的重载，一种是 getApple 方法的重载。

但是这样就涉及到一个问题，要是有几个相同的名字，Java 如何知道你调用的是哪个方法呢？这里记住一点即可，**每个重载的方法都有独一无二的参数列表**。其中包括参数的类型、顺序、参数数量等，满足一种一个因素就构成了重载的必要条件。

请记住下面重载的条件

- 方法名称必须相同。
- 参数列表必须不同（个数不同、或类型不同、参数类型排列顺序不同等）。
- 方法的返回类型可以相同也可以不相同。
- 仅仅返回类型不同不足以成为方法的重载。
- 重载是发生在编译时的，因为编译器可以根据参数的类型来选择使用哪个方法。

方法的重写

方法的重写与重载虽然名字很相似，但却完全是不同的东西。方法重写的描述是对 子类和父类之间的。而重载指的是同一类中的。例如如下代码

```
class Fruit {

 public void eat(){
   System.out.printl('eat fruit');
 }
}

class Apple extends Fruit{

 @Override
 public void eat(){
   System.out.printl('eat apple');
 }
}
```

上面这段代码描述的就是重写的代码，你可以看到，子类 Apple 中的方法和父类 Fruit 中的方法同名，所以，我们能够推断出重写的原则

- 重写的方法必须要和父类保持一致，包括**返回值类型,方法名,参数列表** 也都一样。

- 重写的方法可以使用 `@Override` 注解来标识
- 子类中重写方法的访问权限不能低于父类中方法的访问权限。

初始化

类的初始化

上面我们创建出来了一个 Car 这个对象，其实在使用 new 关键字创建一个对象的时候，其实是调用了这个对象无参数的构造方法进行的初始化，也就是如下这段代码

```
class Car{
  public Car(){}
}
```

这个无参数的构造函数可以隐藏，由 JVM 自动添加。也就是说，构造函数能够确保类的初始化。

成员初始化

Java 会尽量保证每个变量在使用前都会获得初始化，初始化涉及两种初始化。

- 一种是编译器默认指定的字段初始化，基本数据类型的初始化

- 一种是其他对象类型的初始化，String 也是一种对象，对象的初始值都为 `null` ，其中也包括基本类型的包装类。

- 一种是指定数值的初始化，例如

```
int a = 11
```

也就是说， 指定 a 的初始化值不是 0 ， 而是 11。其他基本类型和对象类型也是一样的。

构造器初始化

可以利用构造器来对某些方法和某些动作进行初始化，确定初始值，例如

```
public class Counter{
  int i;
  public Counter(){
    i = 11;
  }
}
```

利用构造函数，能够把 i 的值初始化为 11。

初始化顺序

首先先来看一下有哪些需要探讨的初始化顺序

- 静态属性：static 开头定义的属性
- 静态方法块：static {} 包起来的代码块
- 普通属性：非 static 定义的属性
- 普通方法块：{} 包起来的代码块
- 构造函数：类名相同的方法
- 方法：普通方法

```
public class LifeCycle {
    // 静态属性
    private static String staticField = getStaticField();
    // 静态方法块
    static {
```

```
        System.out.println(staticField);
        System.out.println("静态方法块初始化");
    }
    // 普通属性
    private String field = getField();
    // 普通方法块
    {
        System.out.println(field);
    }
    // 构造函数
    public LifeCycle() {
        System.out.println("构造函数初始化");
    }

    public static String getStaticField() {
        String statiFiled = "Static Field Initial";
        return statiFiled;
    }

    public static String getField() {
        String filed = "Field Initial";
        return filed;
    }
    // 主函数
    public static void main(String[] argc) {
        new LifeCycle();
    }
}
```

这段代码的执行结果就反应了它的初始化顺序

静态属性初始化 静态方法块初始化 普通属性初始化 普通方法块初始化 构造函数初始化

数组初始化

数组是相同类型的、用一个标识符名称封装到一起的一个对象序列或基本类型数据序列。数组是通过方括号下标操作符 `[]` 来定义使用。

一般数组是这么定义的

```
int[] a1;

//或者
```

```
int a1[];
```

两种格式的含义是一样的。

- 直接给每个元素赋值 : int array[4] = {1,2,3,4};
- 给一部分赋值，后面的都为 0 ： int array[4] = {1,2};
- 由赋值参数个数决定数组的个数 ： int array[] = {1,2};

可变参数列表

Java 中一种数组冷门的用法就是 可变参数 ，可变参数的定义如下

```
public int add(int... numbers){
  int sum = 0;
  for(int num : numbers){
    sum += num;
  }
  return sum;
}
```

然后，你可以使用下面这几种方式进行可变参数的调用

```
add();  // 不传参数
add(1);  // 传递一个参数
add(2,1);  // 传递多个参数
add(new Integer[] {1, 3, 2});  // 传递数组
```

对象的销毁

虽然 Java 语言是基于 C++ 的，但是它和 C/C++ 一个重要的特征就是不需要手动管理对象的销毁工作。在著名的一书 《深入理解 Java 虚拟机》中提到一个观点

在 Java 中，我们不再需要手动管理对象的销毁，它是由 Java 虚拟机 进行管理和销毁的。虽然我们不需要手动管理对象，但是你需要知道 对象作用域 这个概念。

对象作用域

J多数语言都有 作用域(scope) 这个概念。作用域决定了其内部定义的变量名的可见性和生命周期。在 C、C++ 和 Java 中，作用域通常由 {} 的位置来决定，例如

```
{
  int a = 11;
  {
    int b = 12;
  }
}
```

a 变量会在两个 {} 作用域内有效，而 b 变量的值只能在它自己的 {} 内有效。

虽然存在作用域，但是不允许这样写

```
{
  int x = 11;
  {
    int x = 12;
  }
}
```

这种写法在 C/C++ 中是可以的，但是在 Java 中不允许这样写，因为 Java 设计者认为这样写会导致程序混乱。

###this 和 super

this 和 super 都是 Java 中的关键字

this 表示的当前对象，this 可以调用方法、调用属性和指向对象本身。this 在 Java 中的使用一般有三种：指向当前对象

```
public class Apple {

    int i = 0;

    Apple eatApple(){
        i++;
        return this;
    }

    public static void main(String[] args) {
```

```
        Apple apple = new Apple();
        apple.eatApple().eatApple();
    }
}
```

这段代码比较精妙，精妙在哪呢，我一个 eatApple() 方法竟然可以调用多次，你在后面还可以继续调用，这就很神奇了，为啥呢？其实就是 this 在作祟了，我在 `eatApple` 方法中加了一个 `return this` 的返回值，也就是说哪个对象调用 eatApple 方法都能返回对象的自身。

this 还可以修饰属性，最常见的就是在构造方法中使用 this ，如下所示

```
public class Apple {

    private int num;

    public Apple(int num){
        this.num = num;
    }

    public static void main(String[] args) {
        new Apple(10);
    }
}
```

main 方法中传递了一个 int 值为 10 的参数，它表示的就是苹果的数量，并把这个数量赋给了 num 全局变量。所以 num 的值现在就是 10。

this 还可以和构造函数一起使用，充当一个全局关键字的效果

```
public class Apple {

    private int num;
    private String color;

    public Apple(int num){
        this(num,"红色");
    }

    public Apple(String color){
        this(1,color);
    }

    public Apple(int num, String color) {
```

```
        this.num = num;
        this.color = color;
    }

}
```

你会发现上面这段代码使用的不是 this, 而是 this(参数) 。它相当于调用了其他构造方法，然后传递参数进去。这里注意一点：this() 必须放在构造方法的第一行，否则编译不通过

如果你把 this 理解为指向自身的一个引用，那么 super 就是指向父类的一个引用。super 关键字和 this 一样，你可以使用 super.对象 来引用父类的成员，如下

```
public class Fruit {

    int num;
    String color;

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

```
public class Apple extends Fruit{

    @Override
    public void eat() {
        super.num = 10;
        System.out.println("eat " + num + " Apple");
    }

}
```

你也可以使用 super(参数) 来调用父类的构造函数，这里不再举例子了。

下面为你汇总了 this 关键字和 super 关键字的比较。

访问控制权限

访问控制权限又称为 封装 ，它是面向对象三大特性中的一种，我之前在学习过程中经常会忽略封装，心想这不就是一个访问修饰符么，怎么就是三大特性的必要条件了？后来我才知道，**如果你信任的下属对你隐瞒 bug，你是根本不知道的。**

访问控制权限其实最核心就是一点：只对需要的类可见。

Java中成员的访问权限共有四种，分别是 **public、protected、default、private**，它们的可见性如下

继承

继承是所有 OOP(Object Oriented Programming) 语言和 Java 语言都不可或缺的一部分。只要我们创建了一个类，就隐式的继承自 Object 父类，只不过没有指定。如果你显示指定了父类，那么你继承于父类，而你的父类继承于 Object 类。

继承的关键字是 extends ，如上图所示，如果使用了 extends 显示指定了继承，那么我们可以说 Father 是父类，而 Son 是子类，用代码表示如下

```
class Father{}

class Son extends Father{}
```

继承双方拥有某种共性的特征

```
class Father{

  public void feature(){
    System.out.println("父亲的特征");
  }
}

class Son extends Father {
}
```

如果 Son 没有实现自己的方法的话，那么默认就是用的是父类的 `feature` 方法。如果子类实现了自己的 feature 方法，那么就相当于是重写了父类的 feature 方法，这也是我们上面提到的重写了。

多态

多态指的是同一个行为具有多个不同表现形式。是指一个类实例（对象）的相同方法在不同情形下具有不同表现形式。封装和继承是多态的基础，也就是说，多态只是一种表现形式而已。

如何实现多态？多态的实现具有三种充要条件

- 继承
- 重写父类方法
- 父类引用指向子类对象

比如下面这段代码

```
public class Fruit {

    int num;

    public void eat(){
        System.out.println("eat Fruit");
    }
}

public class Apple extends Fruit{

    @Override
    public void eat() {
        super.num = 10;
        System.out.println("eat " + num + " Apple");
    }

    public static void main(String[] args) {
        Fruit fruit = new Apple();
        fruit.eat();
    }
}
```

你可以发现 `main` 方法中有一个很神奇的地方， `Fruit fruit = new Apple()` ，Fruit 类型的对象竟然指向了 Apple 对象的引用，这其实就是多态 -> 父类引用指向子类对象，因为 Apple 继承于

Fruit，并且重写了 eat 方法，所以能够表现出来多种状态的形式。

组合

组合其实不难理解，就是将对象引用置于新类中即可。组合也是一种提高类的复用性的一种方式。如果你想让类具有更多的扩展功能，你需要记住一句话**多用组合，少用继承**。

```
public class SoccerPlayer {

    private String name;
    private Soccer soccer;

}

public class Soccer {

    private String soccerName;
}
```

代码中 SoccerPlayer 引用了 Soccer 类，通过引用 Soccer 类，来达到调用 soccer 中的属性和方法。

组合和继承是有区别的，它们的主要区别如下。

关于继承和组合孰优孰劣的争论没有结果，只要发挥各自的长处和优点即可，一般情况下，组合和继承也是一对可以连用的好兄弟。

代理

除了继承和组合外，另外一种值得探讨的关系模型称为 代理 。代理的大致描述是，A 想要调用 B 类的方法，A 不直接调用，A 会在自己的类中创建一个 B 对象的代理，再由代理调用 B 的方法。例如如下代码

```
public class Destination {

    public void todo(){
        System.out.println("control...");
    }
}

public class Device {

    private String name;
    private Destination destination;
    private DeviceController deviceController;

    public void control(Destination destination){
        destination.todo();
    }

}

public class DeviceController {

    private Device name;
    private Destination destination;

    public void control(Destination destination){
        destination.todo();
    }
}
```

向上转型

向上转型代表了父类与子类之间的关系，其实父类和子类之间不仅仅有向上转型，还有向下转型，它们的转型后的范围不一样

- 向上转型 ： 通过子类对象(小范围)转化为父类对象(大范围)，这种转换是自动完成的，不用强制。
- 向下转型 ：通过父类对象(大范围)实例化子类对象(小范围)，这种转换不是自动完成的，需要强制指定。

static

static 是 Java 中的关键字，它的意思是 静态的 ，static 可以用来修饰成员变量和方法，static 用在没有创建对象的情况下调用 方法/变量。

- 用 static 声明的成员变量为静态成员变量，也成为类变量。类变量的生命周期和类相同，在整个应用程序执行期间都有效。

```
static String name = "cxuan";
```

- 使用 static 修饰的方法称为静态方法，静态方法能够直接使用**类名.方法名** 进行调用。由于静态方法不依赖于任何对象就可以直接访问，因此对于静态方法来说，是没有 this 关键字的，实例变量都会有 this 关键字。在静态方法中不能访问类的非静态成员变量和非静态方法，

```
static void printMessage(){
  System.out.println("cxuan is writing the article");
}
```

static 除了修饰属性和方法外，还有 静态代码块 的功能，可用于类的初始化操作。进而提升程序的性能。

```
public class StaicBlock {
    static{
        System.out.println("I'm A static code block");
    }
}
```

由于静态代码块随着类的加载而执行，因此，很多时候会将只需要进行一次的初始化操作放在 static 代码块中进行。

final

final 的意思是最后的、最终的，它可以修饰类、属性和方法。

- final 修饰类时，表明这个类不能被继承。final 类中的成员变量可以根据需要设为 final，但是要注意 final 类中的所有成员方法都会被隐式地指定为 final 方法。
- final 修饰方法时，表明这个方法不能被任何子类重写，因此，如果只有在想明确禁止该方法在子类中被覆盖的情况下才将方法设置为 final。

- final 修饰变量分为两种情况，一种是修饰基本数据类型，表示数据类型的值不能被修改；一种是修饰引用类型，表示对其初始化之后便不能再让其指向另一个对象。

接口和抽象类

接口

接口相当于就是对外的一种约定和标准，这里拿操作系统举例子，为什么会有操作系统？就会为了屏蔽软件的复杂性和硬件的简单性之间的差异，为软件提供统一的标准。

在 Java 语言中，接口是由 `interface` 关键字来表示的，比如我们可以向下面这样定义一个接口

```
public interface CxuanGoodJob {}
```

比如我们定义了一个 CxuanGoodJob 的接口，然后你就可以在其内部定义 cxuan 做的好的那些事情，比如 cxuan 写的文章不错。

```
public interface CxuanGoodJob {

    void writeWell();
}
```

这里隐含了一些接口的特征：

- `interface` 接口是一个完全抽象的类，他不会提供任何方法的实现，只是会进行方法的定义。
- 接口中只能使用两种访问修饰符，一种是 `public` ，它对整个项目可见；一种是 `default` 缺省值，它只具有包访问权限。
- 接口只提供方法的定义，接口没有实现，但是接口可以被其他类实现。也就是说，实现接口的类需要提供方法的实现，实现接口使用 `implements` 关键字来表示，一个接口可以有多个实现。

```
class CXuanWriteWell implements CxuanGoodJob{

    @Override
    public void writeWell() {
        System.out.println("Cxuan write Java is vary well");
```

```
    }
}
```

- 接口不能被实例化，所以接口中不能有任何构造方法，你定义构造方法编译会出错。
- 接口的实现比如实现接口的全部方法，否则必须定义为 抽象类 ，这就是我们下面要说的内容

抽象类

抽象类是一种抽象能力弱于接口的类，在 Java 中，抽象类使用 abstract 关键字来表示。如果把接口形容为狗这个物种，那么抽象类可以说是毛发是白色、小体的品种，而实现类可以是具体的类，比如说是博美、泰迪等。你可以像下面这样定义抽象类

```
public interface Dog {

    void FurColor();

}

abstract class WhiteDog implements Dog{

    public void FurColor(){
        System.out.println("Fur is white");
    }

    abstract void SmallBody();
}
```

在抽象类中，具有如下特征

- 如果一个类中有抽象方法，那么这个类一定是抽象类，也就是说，使用关键字 abstract 修饰的方法一定是抽象方法，具有抽象方法的类一定是抽象类。实现类方法中只有方法具体的实现。
- 抽象类中不一定只有抽象方法，抽象类中也可以有具体的方法，你可以自己去选择是否实现这些方法。
- 抽象类中的约束不像接口那么严格，你可以在抽象类中定义 **构造方法、抽象方法、普通属性、方法、静态属性和静态方法**
- 抽象类和接口一样不能被实例化，实例化只能实例化 具体的类

异常

异常是程序经常会出现的，发现错误的最佳时机是在编译阶段，也就是你试图在运行程序之前。但是，在编译期间并不能找到所有的错误，有一些 NullPointerException 和 ClassNotFoundException 异常在编译期找不到，这些异常是 RuntimeException 运行时异常，这些异常往往在运行时才能被发现。

我们写 Java 程序经常会出现两种问题，一种是 java.lang.Exception ，一种是 java.lang.Error，都用来表示出现了异常情况，下面就针对这两种概念进行理解。

认识 Exception

Exception 位于 java.lang 包下，它是一种顶级接口，继承于 Throwable 类，Exception 类及其子类都是 Throwable 的组成条件，是程序出现的合理情况。

在认识 Exception 之前，有必要先了解一下什么是 Throwable 。

什么是 Throwable

Throwable 类是 Java 语言中所有 错误(errors) 和 异常(exceptions) 的父类。只有继承于 Throwable 的类或者其子类才能够被抛出，还有一种方式是带有 Java 中的 @throw 注解的类也可以抛出。

在Java规范中，对非受查异常和受查异常的定义是这样的：

> The *unchecked exception classes* are the run-time exception classes and the error classes.

> The *checked exception classes* are all exception classes other than the unchecked exception classes. That is, the checked exception classes are Throwable and all its subclasses other than RuntimeException and its subclasses and Error and its subclasses.

也就是说，除了 RuntimeException 和其子类，以及 error 和其子类，其它的所有异常都是 checkedException 。

那么，按照这种逻辑关系，我们可以对 Throwable 及其子类进行归类分析

可以看到，Throwable 位于异常和错误的最顶层，我们查看 Throwable 类中发现它的方法和属性有很多，我们只讨论其中几个比较常用的

```
// 返回抛出异常的详细信息
public string getMessage();
public string getLocalizedMessage();

//返回异常发生时的简要描述
public public String toString();

// 打印异常信息到标准输出流上
public void printStackTrace();
public void printStackTrace(PrintStream s);
public void printStackTrace(PrintWriter s)

// 记录栈帧的的当前状态
public synchronized Throwable fillInStackTrace();
```

此外，因为 Throwable 的父类也是 Object ，所以常用的方法还有继承其父类的 getClass() 和 getName() 方法。

常见的 Exception

下面我们回到 Exception 的探讨上来，现在你知道了 Exception 的父类是 Throwable，并且 Exception 有两种异常，一种是 RuntimeException ；一种是 CheckedException ，这两种异常都应该去 捕获 。

下面列出了一些 Java 中常见的异常及其分类，这块面试官也可能让你举出几个常见的异常情况并将其分类

RuntimeException

UncheckedException

与 Exception 有关的 Java 关键字

那么 Java 中是如何处理这些异常的呢？在 Java 中有这几个关键字 **throws、throw、try、finally、catch** 下面我们分别来探讨一下

throws 和 throw

在 Java 中，异常也就是一个对象，它能够被程序员自定义抛出或者应用程序抛出，必须借助于 throws 和 throw 语句来定义抛出异常。

throws 和 throw 通常是成对出现的，例如

```
static void cacheException() throws Exception{

  throw new Exception();

}
```

throw 语句用在方法体内，表示抛出异常，由方法体内的语句处理。throws 语句用在方法声明后面，表示再抛出异常，由该方法的调用者来处理。

throws 主要是声明这个方法会抛出这种类型的异常，使它的调用者知道要捕获这个异常。throw 是具体向外抛异常的动作，所以它是抛出一个异常实例。

try 、finally 、catch

这三个关键字主要有下面几种组合方式 **try...catch 、try...finally、try...catch...finally**。

try...catch 表示对某一段代码可能抛出异常进行的捕获，如下

```
static void cacheException() throws Exception{

  try {
    System.out.println("1");
  }catch (Exception e){
    e.printStackTrace();
  }

}
```

try...finally 表示对一段代码不管执行情况如何，都会走 finally 中的代码

```
static void cacheException() throws Exception{
  for (int i = 0; i < 5; i++) {
    System.out.println("enter: i=" + i);
    try {
      System.out.println("execute: i=" + i);
      continue;
    } finally {
      System.out.println("leave: i=" + i);
    }
  }
}
```

try...catch...finally 也是一样的，表示对异常捕获后，再走 finally 中的代码逻辑。

什么是 Error

Error 是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。这些错误是不可检查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况，比如 OutOfMemoryError 和 StackOverflowError 异常的出现会有几种情况，这里需要先介绍一下 Java 内存模型 JDK1.7。

其中包括两部分，**由所有线程共享的数据区和线程隔离的数据区**组成，在上面的 Java 内存模型中，**只有程序计数器**是不会发生 OutOfMemoryError 情况的区域，程序计数器控制着计算机指令的分支、循环、跳转、异常处理和线程恢复，并且程序计数器是每个线程私有的。

> 什么是线程私有：表示的就是各条线程之间互不影响，独立存储的内存区域。

如果应用程序执行的是 Java 方法，那么这个计数器记录的就是 虚拟机字节码 指令的地址；如果正在执行的是 Native 方法，这个计数器值则为 空(Undefined) 。

除了程序计数器外，其他区域： 方法区(Method Area) 、 虚拟机栈(VM Stack) 、 本地方法栈(Native Method Stack) 和 堆(Heap) 都是可能发生 OutOfMemoryError 的区域。

- 虚拟机栈：如果线程请求的栈深度大于虚拟机栈所允许的深度，将会出现 StackOverflowError 异常；如果虚拟机动态扩展无法申请到足够的内存，将出现 OutOfMemoryError 。
- 本地方法栈和虚拟机栈一样
- 堆：Java 堆可以处于物理上不连续，逻辑上连续，就像我们的磁盘空间一样，如果堆中没有内存完成实例分配，并且堆无法扩展时，将会抛出 OutOfMemoryError。
- 方法区：方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。

在 Java 中，你可以把异常理解为是一种能够提高你程序健壮性的机制，它能够让你在编写代码中注意这些问题，也可以说，如果你写代码不会注意这些异常情况，你是无法成为一位硬核程序员的。

内部类

距今为止，我们了解的都是普通类的定义，那就是直接在 IDEA 中直接新建一个 class 。

新建完成后，你就会拥有一个 class 文件的定义，这种操作太简单了，时间长了就会枯燥，我们年轻人多需要更新潮和骚气的写法，好吧，既然你提到了那就使用 内部类 吧，这是一种有用而且骚气的定义类的方式，内部类的定义非常简单：**可以将一个类的定义放在另一个类的内部，这就是内部类。**

内部类是一种非常有用的特性，定义在类内部的类，持有外部类的引用，但却对其他外部类不可见，看起来就像是一种隐藏代码的机制，就和 弗兰奇将军 似的，弗兰奇可以和弗兰奇将军进行通讯，但是外面的敌人却无法直接攻击到弗兰奇本体。

下面我们就来聊一聊创建内部类的方式。

创建内部类

定义内部类非常简单，就是直接将一个类定义在外围类的里面，如下代码所示

```
public class OuterClass {
    private String name ;
    private int age;

    class InnerClass{
        public InnerClass(){
            name = "cxuan";
```

```
            age = 25;
        }
    }
}
```

在这段代码中，InnerClass 就是 OuterClass 的一个内部类。也就是说，每个内部类都能独立地继承一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响。这也是隐藏了内部实现细节。**内部类拥有外部类的访问权。**

内部类不仅仅能够定义在类的内部，还可以定义在方法和作用域内部，这种被称为 局部内部类 ，除此之外，还有匿名内部类、内部类可以实现 Java 中的 多重继承 。下面是定义内部类的方式

- 一个在方法中定义的类(局部内部类)
- 一个定义在作用域内的类，这个作用域在方法的内部(成员内部类)
- 一个实现了接口的匿名类(匿名内部类)
- 一个匿名类，它扩展了非默认构造器的类
- 一个匿名类，执行字段初始化操作
- 一个匿名类，它通过实例初始化实现构造

由于每个类都会产生一个 .class 文件，其中包含了如何创建该类型的对象的全部信息，那么，如何表示内部类的信息呢？可以使用 $ 来表示，比如 **OuterClass$InnerClass.class。**

集合

集合在我们的日常开发中所使用的次数简直太多了，你已经把它们都用的熟透了，但是作为一名合格的程序员，你不仅要了解它的基本用法，你还要了解它的源码；存在即合理，你还要了解它是如何设计和实现的，你还要了解它的衍生过程。

这篇博客就来详细介绍一下 Collection 这个庞大集合框架的家族体系和成员，让你了解它的设计与实现。

是时候祭出这张神图了

首先来介绍的就是列表爷爷辈儿的接口- **Iterator**

Iterable 接口

实现此接口允许对象成为 for-each 循环的目标，也就是增强 for 循环，它是 Java 中的一种 语法糖 。

```
List<Object> list = new ArrayList();
for (Object obj: list){}
```

除了实现此接口的对象外，数组也可以用 for-each 循环遍历，如下：

```
Object[] list = new Object[10];
for (Object obj: list){}
```

其他遍历方式

jdk 1.8之前 Iterator 只有 iterator 一个方法，就是

```
Iterator<T> iterator();
```

实现次接口的方法能够创建一个轻量级的迭代器，用于安全的遍历元素，移除元素，添加元素。这里面涉及到一个 fail-fast 机制。

总之一点就是能创建迭代器进行元素的添加和删除的话，就尽量使用迭代器进行添加和删除。

也可以使用迭代器的方式进行遍历

```
for(Iterator it = coll.iterator(); it.hasNext(); ){
    System.out.println(it.next());
}
```

顶层接口

Collection 是一个顶层接口，它主要用来定义集合的约定

List 接口也是一个顶层接口，它继承了 Collection 接口 ，同时也是 ArrayList、LinkedList 等集合元素的父类

Set 接口位于与 List 接口同级的层次上，它同时也继承了 Collection 接口。Set 接口提供了额外的规定。它对add、equals、hashCode 方法提供了额外的标准。

Queue 是和 List、Set 接口并列的 Collection 的三大接口之一。Queue 的设计用来在处理之前保持元素的访问次序。除了 Collection 基础的操作之外，队列提供了额外的插入，读取，检查操作。

SortedSet 接口直接继承于 Set 接口，使用 Comparable 对元素进行自然排序或者使用 Comparator 在创建时对元素提供定制的排序规则。set 的迭代器将按升序元素顺序遍历集合。

Map 是一个支持 key-value 存储的对象，Map 不能包含重复的 key，每个键最多映射一个值。这个接口代替了Dictionary 类，Dictionary 是一个抽象类而不是接口。

ArrayList

ArrayList 是实现了 List 接口的 可扩容数组(动态数组) ，它的内部是基于数组实现的。它的具体定义如下：

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, java.io.Seriali
```

- ArrayList 可以实现所有可选择的列表操作，允许所有的元素，包括空值。ArrayList 还提供了内部存储 list 的方法，它能够完全替代 Vector，只有一点例外，ArrayList 不是线程安全的容器。
- ArrayList 有一个容量的概念，这个数组的容量就是 List 用来存储元素的容量。
- ArrayList 不是线程安全的容器，如果多个线程中至少有两个线程修改了 ArrayList 的结构的话就会导致线程安全问题，作为替代条件可以使用线程安全的 List，应使用 Collections.synchronizedList 。

```
List list = Collections.synchronizedList(new ArrayList(...))
```

- ArrayList 具有 fail-fast 快速失败机制，能够对 ArrayList 作出失败检测。当在迭代集合的过程中该集合在结构上发生改变的时候，就有可能会发生 fail-fast，即抛出 ConcurrentModificationException 异常。

Vector

Vector 同 ArrayList 一样，都是基于数组实现的，只不过 Vector 是一个线程安全的容器，它对内部的每个方法都简单粗暴的上锁，避免多线程引起的安全性问题，但是通常这种同步方式需要的开销比较大，因此，访问元素的效率要远远低于 ArrayList。

还有一点在于扩容上，ArrayList 扩容后的数组长度会增加 50%，而 Vector 的扩容长度后数组会增加一倍。

LinkedList 类

LinkedList 是一个双向链表，允许存储任何元素(包括 null)。它的主要特性如下：

- LinkedList 所有的操作都可以表现为双向性的，索引到链表的操作将遍历从头到尾，视哪个距离近为遍历顺序。
- 注意这个实现也不是线程安全的，如果多个线程并发访问链表，并且至少其中的一个线程修改了链表的结构，那么这个链表必须进行外部加锁。或者使用

```
List list = Collections.synchronizedList(new LinkedList(...))
```

Stack

堆栈是我们常说的 后入先出(吃了吐) 的容器 。它继承了 Vector 类，提供了通常用的 push 和 pop 操作，以及在栈顶的 peek 方法，测试 stack 是否为空的 empty 方法，和一个寻找与栈顶距离的 search 方法。

第一次创建栈，不包含任何元素。一个更完善，可靠性更强的 LIFO 栈操作由 Deque 接口和他的实现提供，应该优先使用这个类

```
Deque<Integer> stack = new ArrayDeque<Integer>()
```

HashSet

HashSet 是 Set 接口的实现类，由哈希表支持(实际上 HashSet 是 HashMap 的一个实例)。它不能保证集合的迭代顺序。这个类允许 null 元素。

- 注意这个实现不是线程安全的。如果多线程并发访问 HashSet，并且至少一个线程修改了set，必须进行外部加锁。或者使用 Collections.synchronizedSet() 方法重写。
- 这个实现支持 fail-fast 机制。

TreeSet

TreeSet 是一个基于 TreeMap 的 NavigableSet 实现。这些元素使用他们的自然排序或者在创建时提供的Comparator 进行排序，具体取决于使用的构造函数。

- 此实现为基本操作 add,remove 和 contains 提供了 log(n) 的时间成本。
- 注意这个实现不是线程安全的。如果多线程并发访问 TreeSet，并且至少一个线程修改了 set，必须进行外部加锁。或者使用

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...))
```

- 这个实现持有 fail-fast 机制。

LinkedHashSet 类

LinkedHashSet 继承于 Set，先来看一下 LinkedHashSet 的继承体系：

LinkedHashSet 是 Set 接口的 Hash 表和 LinkedList 的实现。这个实现不同于 HashSet 的是它维护着一个贯穿所有条目的双向链表。此链表定义了元素插入集合的顺序。注意：如果元素重新插入，则插入顺序不会受到影响。

- LinkedHashSet 有两个影响其构成的参数：初始容量和加载因子。它们的定义与 HashSet 完全相同。但请注意：对于 LinkedHashSet，选择过高的初始容量值的开销要比 HashSet 小，因为 LinkedHashSet 的迭代次数不受容量影响。
- 注意 LinkedHashSet 也不是线程安全的，如果多线程同时访问 LinkedHashSet，必须加锁，或者通过使用

Collections.synchronizedSet

- 该类也支持fail-fast机制

PriorityQueue

PriorityQueue 是 AbstractQueue 的实现类，优先级队列的元素根据自然排序或者通过在构造函数时期提供Comparator 来排序，具体根据构造器判断。PriorityQueue 不允许 null 元素。

- 队列的头在某种意义上是指定顺序的最后一个元素。队列查找操作 poll,remove,peek 和 element 访问队列头部元素。
- 优先级队列是无限制的，但具有内部 capacity，用于控制用于在队列中存储元素的数组大小。
- 该类以及迭代器实现了 Collection、Iterator 接口的所有可选方法。这个迭代器提供了 `iterator()` 方法不能保证以任何特定顺序遍历优先级队列的元素。如果你需要有序遍历，考虑使用 `Arrays.sort(pq.toArray())` 。
- 注意这个实现不是线程安全的，多线程不应该并发访问 PriorityQueue 实例 如果有某个线程修改了队列的话，使用线程安全的类 `PriorityBlockingQueue` 。

HashMap

HashMap 是一个利用哈希表原理来存储元素的集合，并且允许空的 key-value 键值对。HashMap 是非线程安全的，也就是说在多线程的环境下，可能会存在问题，而 Hashtable 是线程安全的容器。HashMap 也支持 fail-fast 机制。HashMap 的实例有两个参数影响其性能：初始容量 和加载因子。可以使用 `Collections.synchronizedMap(new HashMap(...))` 来构造一个线程安全的 HashMap。

TreeMap 类

一个基于 NavigableMap 实现的红黑树。这个 map 根据 key 自然排序存储，或者通过 Comparator 进行定制排序。

- TreeMap 为 containsKey,get,put 和remove方法提供了 log(n) 的时间开销。
- 注意这个实现不是线程安全的。如果多线程并发访问 TreeMap，并且至少一个线程修改了 map，必须进行外部加锁。这通常通过在自然封装集合的某个对象上进行同步来实现，或者使用 `SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...))` 。
- 这个实现持有fail-fast机制。

LinkedHashMap 类

LinkedHashMap 是 Map 接口的哈希表和链表的实现。这个实现与 HashMap 不同之处在于它维护了一个贯穿其所有条目的双向链表。这个链表定义了遍历顺序，通常是插入 map 中的顺序。

- 它提供一个特殊的 LinkedHashMap(int,float,boolean) 构造器来创建 LinkedHashMap，其遍历顺序是其最后一次访问的顺序。
- 可以重写 removeEldestEntry(Map.Entry) 方法，以便在将新映射添加到 map 时强制删除过期映射的策略。
- 这个类提供了所有可选择的 map 操作，并且允许 null 元素。由于维护链表的额外开销，性能可能会低于HashMap，有一条除外：遍历 LinkedHashMap 中的 collection-views 需要与 map.size 成正比，无论其容量如何。HashMap 的迭代看起来开销更大，因为还要求时间与其容量成正比。
- LinkedHashMap 有两个因素影响了它的构成：初始容量和加载因子。
- 注意这个实现不是线程安全的。如果多线程并发访问LinkedHashMap，并且至少一个线程修改了map，必须进行外部加锁。这通常通过在自然封装集合的某个对象上进行同步来实现 `Map m = Collections.synchronizedMap(new LinkedHashMap(...))` 。
- 这个实现持有fail-fast机制。

Hashtable 类

Hashtable 类实现了一个哈希表，能够将键映射到值。任何非空对象都可以用作键或值。

- 此实现类支持 fail-fast 机制
- 与新的集合实现不同，Hashtable 是线程安全的。如果不需要线程安全的容器，推荐使用 HashMap，如果需要多线程高并发，推荐使用 `ConcurrentHashMap` 。

IdentityHashMap 类

IdentityHashMap 是比较小众的 Map 实现了。

- 这个类不是一个通用的 Map 实现！虽然这个类实现了 Map 接口，但它故意违反了 Map 的约定，该约定要求在比较对象时使用 equals 方法，此类仅适

用于需要引用相等语义的极少数情况。

- 同 HashMap，IdentityHashMap 也是无序的，并且该类不是线程安全的，如果要使之线程安全，可以调用 Collections.synchronizedMap(new IdentityHashMap(...)) 方法来实现。
- 支持fail-fast机制

WeakHashMap 类

WeakHashMap 类基于哈希表的 Map 基础实现，带有弱键。WeakHashMap 中的 entry 当不再使用时还会自动移除。更准确的说，给定key的映射的存在将不会阻止 key 被垃圾收集器丢弃。

- 基于 map 接口，是一种弱键相连，WeakHashMap 里面的键会自动回收
- 支持 null 值和 null 键。
- fast-fail 机制
- 不允许重复
- WeakHashMap 经常用作缓存

Collections 类

Collections 不属于 Java 框架继承树上的内容，它属于单独的分支，Collections 是一个包装类，它的作用就是为集合框架提供某些功能实现，此类只包括静态方法操作或者返回 collections。

同步包装

同步包装器将自动同步（线程安全性）添加到任意集合。六个核心集合接口（Collection，Set，List，Map，SortedSet 和 SortedMap）中的每一个都有一个静态工厂方法。

```
public static Collection synchronizedCollection(Collection c);
public static Set synchronizedSet(Set s);
public static List synchronizedList(List list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static SortedSet synchronizedSortedSet(SortedSet s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

不可修改的包装

不可修改的包装器通过拦截修改集合的操作并抛出 UnsupportedOperationException ，主要用在下面两个情景：

- 构建集合后使其不可变。在这种情况下，最好不要去获取返回 collection 的引用，这样有利于保证不变性
- 允许某些客户端以只读方式访问你的数据结构。你保留对返回的 collection 的引用，但分发对包装器的引用。通过这种方式，客户可以查看但不能修改，同时保持完全访问权限。

这些方法是：

```
public static Collection unmodifiableCollection(Collection<? extends T> c);
public static Set unmodifiableSet(Set<? extends T> s);
public static List unmodifiableList(List<? extends T> list);
public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);
public static SortedSet unmodifiableSortedSet(SortedSet<? extends T> s);
public static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> m);
```

线程安全的Collections

Java1.5 并发包 (java.util.concurrent) 提供了线程安全的 collections 允许遍历的时候进行修改，通过设计iterator 为 fail-fast 并抛出 ConcurrentModificationException。一些实现类是 CopyOnWriteArrayList ， ConcurrentHashMap ， CopyOnWriteArraySet

Collections 算法

此类包含用于集合框架算法的方法，例如二进制搜索，排序，重排，反向等。

集合实现类特征图

下图汇总了部分集合框架的主要实现类的特征图，让你能有清晰明了看出每个实现类之间的差异性

还有一种类型是关于强引用、弱引用、虚引用的文章，请参考

https://mp.weixin.qq.com/s/ZflBpn2TBzTNv_-G-zZxNg

泛形

在 Jdk1.5 中，提出了一种新的概念，那就是泛型，那么什么是泛型呢?

泛型其实就是一种参数化的集合，它限制了你添加进集合的类型。泛型的本质就是一种参数化类型。多态也可以看作是泛型的机制。一个类继承了父类，那么就能通过它的父类找到对应的子类，但是不能通过其他类来找到具体要找的这个类。泛型的设计之处就是希望对象或方法具有最广泛的表达能力。

下面来看一个例子说明没有泛型的用法

```
List arrayList = new ArrayList();
arrayList.add("cxuan");
arrayList.add(100);

for(int i = 0; i< arrayList.size();i++){
    String item = (String)arrayList.get(i);
  System.out.println("test === ", item);
}
```

这段程序不能正常运行，原因是 Integer 类型不能直接强制转换为 String 类型

```
java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
```

如果我们用泛型进行改写后，示例代码如下

```
List<String> arrayList = new ArrayList<String>();

arrayList.add(100);
```

这段代码在编译期间就会报错，编译器会在编译阶段就能够帮我们发现类似这样的问题。

泛型的使用

泛型的使用有多种方式，下面我们就来一起探讨一下。

用泛型表示类

泛型可以加到类上面，来表示这个类的类型

```
//此处 T 可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型
public class GenericDemo<T>{
    //value 这个成员变量的类型为T,T的类型由外部指定
    private T value;

    public GenericDemo(T value) {
        this.value = value;
    }

    public T getValue(){ //泛型方法getKey的返回值类型为T，T的类型由外部指定
        return value;
    }

    public void setValue(T value){
        this.value = value
    }
}
```

用泛型表示接口

泛型接口与泛型类的定义及使用基本相同。

```
//定义一个泛型接口
public interface Generator<T> {
    public T next();
}
```

一般泛型接口常用于 生成器(generator) 中，生成器相当于对象工厂，是一种专门用来创建对象的类。

泛型方法

可以使用泛型来表示方法

```
public class GenericMethods {
  public <T> void f(T x){
    System.out.println(x.getClass().getName());
  }
}
```

泛型通配符

List 是泛型类，为了 表示各种泛型 List 的父类，可以使用类型通配符，类型通配符使用 问号(?) 表示，它的元素类型可以匹配任何类型。例如

```
public static void main(String[] args) {
    List<String> name = new ArrayList<String>();
    List<Integer> age = new ArrayList<Integer>();
    List<Number> number = new ArrayList<Number>();
    name.add("cxuan");
    age.add(18);
    number.add(314);
    generic(name);
    generic(age);
    generic(number);
}

public static void generic(List<?> data) {
    System.out.println("Test cxuan :" + data.get(0));
}
```

上界通配符：<? extends ClassType> 该通配符为 ClassType 的所有子类型。它表示的是任何类型都是 ClassType 类型的子类。

下界通配符：<? super ClassType> 该通配符为 ClassType 的所有超类型。它表示的是任何类型的父类都是 ClassType。

反射

反射是 Java 中一个非常重要同时也是一个高级特性，基本上 Spring 等一系列框架都是基于反射的思想写成的。我们首先来认识一下什么反射。

Java 反射机制是在程序的运行过程中，对于任何一个类，都能够知道它的所有属性和方法；对于任意一个对象，都能够知道调用它的任意属性和方法，这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。（来源于百度百科）

Java 反射机制主要提供了以下这几个功能

- 在运行时判断任意一个对象所属的类
- 在运行时构造任意一个类的对象
- 在运行时判断任意一个类所有的成员变量和方法
- 在运行时调用任意一个对象的方法

这么一看，反射就像是一个掌控全局的角色，不管你程序怎么运行，我都能够知道你这个类有哪些属性和方法，你这个对象是由谁调用的，嗯，很屌。

在 Java 中，使用 Java.lang.reflect 包实现了反射机制。Java.lang.reflect 所设计的类如下

下面是一个简单的反射类

```
public class Person {
    public String name;// 姓名
    public int age;// 年龄

    public Person() {
        super();
    }

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

```
    public String showInfo() {
        return "name=" + name + ", age=" + age;
    }
}

public class Student extends Person implements Study {
    public String className;// 班级
    private String address;// 住址

    public Student() {
        super();
    }

    public Student(String name, int age, String className, String address) {
        super(name, age);
        this.className = className;
        this.address = address;
    }

    public Student(String className) {
        this.className = className;
    }

    public String toString() {
        return "姓名：" + name + ",年龄：" + age + ",班级：" + className + ",住址："
                + address;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}

public class TestRelect {

    public static void main(String[] args) {
        Class student = null;
        try {
            student = Class.forName("com.cxuan.reflection.Student");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
```

```
}

// 获取对象的所有公有属性。
Field[] fields = student.getFields();
for (Field f : fields) {
    System.out.println(f);
}
System.out.println("----------------------");
// 获取对象所有属性，但不包含继承的。
Field[] declaredFields = student.getDeclaredFields();
for (Field df : declaredFields) {
    System.out.println(df);
}

// 获取对象的所有公共方法
Method[] methods = student.getMethods();
for (Method m : methods) {
    System.out.println(m);
}
System.out.println("----------------------");
// 获取对象所有方法，但不包含继承的
Method[] declaredMethods = student.getDeclaredMethods();
for (Method dm : declaredMethods) {
    System.out.println(dm);
}

// 获取对象所有的公共构造方法
Constructor[] constructors = student.getConstructors();
for (Constructor c : constructors) {
    System.out.println(c);
}
System.out.println("----------------------");
// 获取对象所有的构造方法
Constructor[] declaredConstructors = student.getDeclaredConstructors();
for (Constructor dc : declaredConstructors) {
    System.out.println(dc);
}

Class c = Class.forName("com.cxuan.reflection.Student");
Student stu1 = (Student) c.newInstance();
// 第一种方法，实例化默认构造方法，调用set赋值
stu1.setAddress("河北石家庄");
System.out.println(stu1);

// 第二种方法 取得全部的构造函数 使用构造函数赋值
Constructor<Student> constructor = c.getConstructor(String.class,
                                    int.class, String.class, String.class);
```

```
        Student student2 = (Student) constructor.newInstance("cxuan", 24, "六班", "石家庄");
        System.out.println(student2);

        /**
        * 獲取方法并执行方法
        */
        Method show = c.getMethod("showInfo");//获取showInfo()方法
        Object object = show.invoke(stu2);//调用showInfo()方法


    }
}
```

有一些是比较常用的，有一些是我至今都没见过怎么用的，下面进行一个归类。

与 Java 反射有关的类主要有

Class 类

在 Java 中，你每定义一个 java class 实体都会产生一个 Class 对象。也就是说，当我们编写一个类，编译完成后，在生成的 `.class` 文件中，就会产生一个 Class 对象，这个 Class 对象用于表示这个类的类型信息。Class 中没有公共的构造器，也就是说 Class 对象不能被实例化。下面来简单看一下 Class 类都包括了哪些方法

toString()

```
public String toString() {
 return (isInterface() ? "interface " : (isPrimitive() ? "" : "class "))
   + getName();
}
```

toString() 方法能够将对象转换为字符串，toString() 首先会判断 Class 类型是否是接口类型，也就是说，普通类和接口都能够用 Class 对象来表示，然后再判断是否是基本数据类型，这里判断的都是基本数据类型和包装类，还有 `void` 类型。

所有的类型如下

- java.lang.Boolean : 代表 boolean 数据类型的包装类
- java.lang.Character: 代表 char 数据类型的包装类
- java.lang.Byte: 代表 byte 数据类型的包装类
- java.lang.Short: 代表 short 数据类型的包装类

- java.lang.Integer: 代表 int 数据类型的包装类
- java.lang.Long: 代表 long 数据类型的包装类
- java.lang.Float: 代表 float 数据类型的包装类
- java.lang.Double: 代表 double 数据类型的包装类
- java.lang.Void: 代表 void 数据类型的包装类

然后是 `getName()` 方法，这个方法返回类的全限定名称。

- 如果是引用类型，比如 String.class.getName() -> `java.lang.String`
- 如果是基本数据类型，byte.class.getName() -> `byte`
- 如果是数组类型，new Object[3]).getClass().getName() -> `[Ljava.lang.Object`

toGenericString()

这个方法会返回类的全限定名称，而且包括类的修饰符和类型参数信息。

forName()

根据类名获得一个 Class 对象的引用，这个方法会使类对象进行初始化。

例如 `Class t = Class.forName("java.lang.Thread")` 就能够初始化一个 Thread 线程对象

在 Java 中，一共有三种获取类实例的方式

- Class.forName(java.lang.Thread)
- Thread.class
- thread.getClass()

newInstance()

创建一个类的实例，代表着这个类的对象。上面 forName() 方法对类进行初始化，newInstance 方法对类进行实例化。

getClassLoader()

获取类加载器对象。

getTypeParameters()

按照声明的顺序获取对象的参数类型信息。

getPackage()

返回类的包

getInterfaces()

获得当前类实现的类或是接口，可能是有多个，所以返回的是 Class 数组。

Cast

把对象转换成代表类或是接口的对象

asSubclass(Class clazz)

把传递的类的对象转换成代表其子类的对象

getClasses()

返回一个数组，数组中包含该类中所有公共类和接口类的对象

getDeclaredClasses()

返回一个数组，数组中包含该类中所有类和接口类的对象

getSimpleName()

获得类的名字

getFields()

获得所有公有的属性对象

getField(String name)

获得某个公有的属性对象

getDeclaredField(String name)

获得某个属性对象

getDeclaredFields()

获得所有属性对象

getAnnotation(Class annotationClass)

返回该类中与参数类型匹配的公有注解对象

getAnnotations()

返回该类所有的公有注解对象

getDeclaredAnnotation(Class annotationClass)

返回该类中与参数类型匹配的所有注解对象

getDeclaredAnnotations()

返回该类所有的注解对象

getConstructor(Class...<?> parameterTypes)

获得该类中与参数类型匹配的公有构造方法

getConstructors()

获得该类的所有公有构造方法

getDeclaredConstructor(Class...<?> parameterTypes)

获得该类中与参数类型匹配的构造方法

getDeclaredConstructors()

获得该类所有构造方法

getMethod(String name, Class...<?> parameterTypes)

获得该类某个公有的方法

getMethods()

获得该类所有公有的方法

getDeclaredMethod(String name, Class...<?> parameterTypes)

获得该类某个方法

getDeclaredMethods()

获得该类所有方法

Field 类

Field 类提供类或接口中单独字段的信息，以及对单独字段的动态访问。

这里就不再对具体的方法进行介绍了，读者有兴趣可以参考官方 API

这里只介绍几个常用的方法

equals(Object obj)

属性与obj相等则返回true

get(Object obj)

获得obj中对应的属性值

set(Object obj, Object value)

设置obj中对应属性值

Method 类

invoke(Object obj, Object... args)

传递object对象及参数调用该对象对应的方法

ClassLoader 类

反射中，还有一个非常重要的类就是 ClassLoader 类，类装载器是用来把 类(class) 装载进 JVM 的。ClassLoader 使用的是双亲委托模型来搜索加载类的，这个模型也就是双亲委派模型。ClassLoader 的类继承图如下

枚举

枚举可能是我们使用次数比较少的特性，在 Java 中，枚举使用 enum 关键字来表示，枚举其实是一项非常有用的特性，你可以把它理解为具有特定性质的类。enum 不仅仅 Java 有，C 和 C++ 也有枚举的概念。下面是一个枚举的例子。

```
public enum Family {

    FATHER,
    MOTHER,
    SON,
    Daughter;

}
```

上面我们创建了一个 Family 的枚举类，它具有 4 个值，由于枚举类型都是常量，所以都用大写字母来表示。那么 enum 创建出来了，该如何引用呢?

```
public class EnumUse {
```

```
    public static void main(String[] args) {
        Family s = Family.FATHER;
    }
}
```

枚举特性

enum 枚举这个类比较有意思，当你创建完 enum 后，编译器会自动为你的 enum 添加 `toString()` 方法，能够让你方便的显示 enum 实例的具体名字是什么。除了 toString() 方法外，编译器还会添加 `ordinal()` 方法，这个方法用来表示 enum 常量的声明顺序，以及 `values()` 方法显示顺序的值。

```
public static void main(String[] args) {

  for(Family family : Family.values()){
    System.out.println(family + ", ordinal" + family.ordinal());
  }
}
```

enum 可以进行静态导入包，静态导入包可以做到不用输入 `枚举类名.常量` ，可以直接使用常量，神奇吗? 使用 ennum 和 `static` 关键字可以做到静态导入包

上面代码导入的是 Family 中所有的常量，也可以单独指定常量。

枚举和普通类一样

枚举就和普通类一样，除了枚举中能够方便快捷的定义 常量 ，我们日常开发使用的 public static final xxx 其实都可以用枚举来定义。在枚举中也能够定义属性和方法，千万不要把它看作是异类，它和万千的类一样。

```
public enum OrdinalEnum {

    WEST("live in west"),
    EAST("live in east"),
    SOUTH("live in south"),
    NORTH("live in north");

    String description;

    OrdinalEnum(String description){
        this.description = description;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public static void main(String[] args) {
        for(OrdinalEnum ordinalEnum : OrdinalEnum.values()){
            System.out.println(ordinalEnum.getDescription());
        }
    }
}
```

一般 switch 可以和 enum 一起连用，来构造一个小型的状态转换机。

```
enum Signal {
  GREEN, YELLOW, RED
}

public class TrafficLight {
    Signal color = Signal.RED;

    public void change() {
```

```
        switch (color) {
        case RED:
            color = Signal.GREEN;
            break;
        case YELLOW:
            color = Signal.RED;
            break;
        case GREEN:
            color = Signal.YELLOW;
            break;
        }
    }
}
```

是不是代码顿时觉得优雅整洁了些许呢?

枚举神秘之处

在 Java 中，万事万物都是对象，enum 虽然是个关键字，但是它却隐式的继承于 `Enum` 类。我们来看一下 Enum 类，此类位于 `java.lang` 包下，可以自动引用。

此类的属性和方法都比较少。你会发现这个类中没有我们的 values 方法。前面刚说到， values() 方法是你使用枚举时被编译器添加进来的 static 方法。**可以使用反射来验证一下。**

除此之外，enum 还和 Class 类有交集，在 Class 类中有三个关于 Enum 的方法

前面两个方法用于获取 enum 常量， isEnum 用于判断是否是枚举类型的。

枚举类

除了 Enum 外，还需要知道两个关于枚举的工具类，一个是 EnumSet ，一个是 EnumMap

EnumSet 和 EnumMap

EnumSet 是 JDK1.5 引入的，EnumSet 的设计充分考虑到了速度因素，使用 EnumSet 可以作为 Enum 的替代者，因为它的效率比较高。

EnumMap 是一种特殊的 Map，它要求其中的 key 键值是来自一个 enum。因为 EnumMap 速度也很快，我们可以使用 EnumMap 作为 key 的快速查找。

总的来说，枚举的使用不是很复杂，它也是 Java 中很小的一块功能，但有时却能够因为这一个小技巧，能够让你的代码变得优雅和整洁。

I/O

创建一个良好的 I/O 程序是非常复杂的。JDK 开发人员编写了大量的类只为了能够创建一个良好的工具包，想必编写 I/O 工具包很费劲吧。

IO 类设计出来，肯定是为了解决 IO 相关操作的，最常见的 I/O 读写就是网络、磁盘等。在 Java 中，对文件的操作是一个典型的 I/O 操作。下面我们就对 I/O 进行一个分类。

> 公号回复 IO 获取思维导图

I/O 还可以根据操作对象来进行区分：主要分为

除此之外，I/O 中还有其他比较重要的类

File 类

File 类是对文件系统中文件以及文件夹进行操作的类，可以通过面向对象的思想操作文件和文件夹，是不是很神奇？

文件创建操作如下，主要涉及 **文件创建、删除文件、获取文件描述符等**

```
class FileDemo{
  public static void main(String[] args) {
    File file = new File("D:\\file.txt");
    try{
     f.createNewFile(); // 创建一个文件

     // File类的两个常量
     //路径分隔符(与系统有关的）<windows里面是 ; linux里面是：  >
     System.out.println(File.pathSeparator);  //   ;
     //与系统有关的路径名称分隔符<windows里面是 \ linux里面是/ >
     System.out.println(File.separator);      //  \

     // 删除文件
     /*
     File file = new File(fileName);
     if(f.exists()){
        f.delete();
     }else{
```

```
            System.out.println("文件不存在");
        }
        */


      }catch (Exception e) {
          e.printStackTrace();
      }
    }
}
```

也可以对文件夹进行操作

```
class FileDemo{
  public static void main(String[] args) {
    String fileName = "D:"+ File.separator + "filepackage";
    File file = new File(fileName);
    f.mkdir();

  // 列出所有文件
    /*
    String[] str = file.list();
    for (int i = 0; i < str.length; i++) {
      System.out.println(str[i]);
    }
    */

    // 使用 file.listFiles(); 列出所有文件，包括隐藏文件

    // 使用 file.isDirectory() 判断指定路径是否是目录
  }
}
```

上面只是举出来了两个简单的示例，实际上，还有一些其他对文件的操作没有使用。比如创建文件，就可以使用三种方式来创建

```
File(String directoryPath);
File(String directoryPath, String filename);
File(File dirObj, String filename);
```

directoryPath 是文件的路径名，filename 是文件名，dirObj 是一个 File 对象。例如

```
File file = new File("D:\\java\\file1.txt");  //双\\是转义
System.out.println(file);
File file2 = new File("D:\\java","file2.txt");//父路径、子路径--可以适用于多个文件的!
System.out.println(file2);
File parent = new File("D:\\java");
File file3 = new File(parent,"file3.txt");//File类的父路径、子路径
System.out.println(file3);
```

现在对 File 类进行总结

基础 IO 类和相关方法

虽然. IO 类有很多，但是最基本的是四个抽象类，**InputStream、OutputStream、Reader、Writer**。最基本的方法也就是 read() 和 write() 方法，其他流都是上面这四类流的子类，方法也是通过这两类方法衍生而成的。而且大部分的 IO 源码都是 native 标志的，也就是说源码都是 C/C++ 写的。这里我们先来认识一下这些流类及其方法

InputStream

InputStream 是一个定义了 Java 流式字节输入模式的抽象类。该类的所有方法在出错条件下引发一个IOException 异常。它的主要方法定义如下

OutputStream

OutputStream 是定义了流式字节输出模式的抽象类。该类的所有方法返回一个void 值并且在出错情况下引发一个IOException异常。它的主要方法定义如下

Reader 类

Reader 是 Java 定义的流式字符输入模式的抽象类。类中的方法在出错时引发 IOException 异常。

Writer 类

Writer 是定义流式字符输出的抽象类。所有该类的方法都返回一个 void 值并在出错条件下引发 IOException 异常

InputStream 及其子类

FileInputStream 文件输入流：FileInputStream 类创建一个能从文件读取字节的 InputStream 类

ByteArrayInputStream 字节数组输入流： 把内存中的一个缓冲区作为 InputStream 使用

PipedInputStream 管道输入流：实现了pipe 管道的概念，主要在线程中使用

SequenceInputStream 顺序输入流：把多个 InputStream 合并为一个 InputStream

FilterOutputStream 过滤输入流：其他输入流的包装。

ObjectInputStream 反序列化输入流 ： 将之前使用 ObjectOutputStream 序列化的原始数据恢复为对象，以流的方式读取对象

**DataInputStream ** : 数据输入流允许应用程序以与机器无关方式从底层输入流中读取基本 Java 数据类型。

PushbackInputStream 推回输入流：缓冲的一个新颖的用法是实现 推回 （pushback） 。Pushback 用于输入流允许字节被读取然后返回到流。

OutputStream 及其子类

FileOutputStream 文件输出流：该类实现了一个输出流，其数据写入文件。

ByteArrayOutputStream 字节数组输出流 ： 该类实现了一个输出流，其数据被写入由 byte 数组充当的缓冲区，缓冲区会随着数据的不断写入而自动增长。

PipedOutputStream 管道输出流 ： 管道的输出流，是管道的发送端。

ObjectOutputStream 基本类型输出流 ： 该类将实现了序列化的对象序列化后写入指定地方。

FilterOutputStream 过滤输出流：其他输出流的包装。

PrintStream 打印流 通过 PrintStream 可以将文字打印到文件或者网络中去。

DataOutputStream : 数据输出流允许应用程序以与机器无关方式向底层输出流中写入基本 Java 数据类型。

Reader 及其子类

FileReader 文件字符输入流 ： 把文件转换为字符流读入

CharArrayReader 字符数组输入流 ： 是一个把字符数组作为源的输入流的实现

BufferedReader 缓冲区输入流 ： BufferedReader 类从字符输入流中读取文本并缓冲字符，以便有效地读取字符，数组和行

PushbackReader: PushbackReader 类允许一个或多个字符被送回输入流。

PipedReader 管道输入流：主要用途也是在线程间通讯，不过这个可以用来传输字符

Writer 及其子类

FileWriter 字符输出流： FileWriter 创建一个可以写文件的 Writer 类。

CharArrayWriter 字符数组输出流：CharArrayWriter 实现了以数组作为目标的输出流。

BufferedWriter 缓冲区输出流： BufferedWriter是一个增加了 flush( ) 方法的Writer。flush( )方法可以用来确保数据缓冲器确实被写到实际的输出流。

PrintWriter： PrintWriter 本质上是 PrintStream 的字符形式的版本。

PipedWriter 管道输出流：主要用途也是在线程间通讯，不过这个可以用来传输字符

Java 的输入输出的流式接口为复杂而繁重的任务提供了一个简洁的抽象。过滤流类的组合允许你动态建立客户端流式接口来配合数据传输要求。继承高级流类 InputStream、InputStreamReader、 Reader 和 Writer 类的 Java 程序在将来 （即使创建了新的和改进的具体类）也能得到合理运用。

注解

Java 注解（Annotation） 又称为 元数据 ，它为我们在代码中添加信息提供了一种形式化的方法。它是 JDK1.5 引入的，Java 定义了一套注解，共有 7 个，3 个在 java.lang 中，剩下 4 个在 java.lang.annotation 中。

作用在代码中的注解有三个，它们分别是

- @Override ： 重写标记，一般用在子类继承父类后，标注在重写过后的子类方法上。如果发现其父类，或者是引用的接口中并没有该方法时，会报编译错误。
- @Deprecated ： 用此注解注释的代码已经过时，不再推荐使用
- @SuppressWarnings ： 这个注解起到忽略编译器的警告作用

元注解有四个，元注解就是用来标志注解的注解。它们分别是

- @Retention：标识如何存储，是只在代码中，还是编入class文件中，或者是在运行时可以通过反射访问。

RetentionPolicy.SOURCE：注解只保留在源文件，当 Java 文件编译成class文件的时候，注解被遗弃；

RetentionPolicy.CLASS：注解被保留到 class 文件，但 jvm 加载 class 文件时候被遗弃，这是 默认的 生命周期；

RetentionPolicy.RUNTIME：注解不仅被保存到 class 文件中，jvm 加载 class 文件之后，仍然存在；

- @Documented：标记这些注解是否包含在 JavaDoc 中。
- @Target ： 标记这个注解说明了 Annotation 所修饰的对象范围，Annotation 可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。取值如下

```
public enum ElementType {
    TYPE,
    FIELD,
    METHOD,
    PARAMETER,
    CONSTRUCTOR,
    LOCAL_VARIABLE,
    ANNOTATION_TYPE,
    PACKAGE,
    TYPE_PARAMETER,
    TYPE_USE
```

- @Inherited ： 标记这个注解是继承于哪个注解类的。

从 JDK1.7 开始，又添加了三个额外的注解，它们分别是

- @SafeVarargs ： 在声明可变参数的构造函数或方法时，Java 编译器会报 unchecked 警告。使用 @SafeVarargs 可以忽略这些警告
- @FunctionalInterface：表明这个方法是一个函数式接口
- @Repeatable ： 标识某注解可以在同一个声明上使用多次。

> 注意：注解是不支持继承的。

关于 null 的几种处理方式

对于 Java 程序员来说，空指针一直是恼人的问题，我们在开发中经常会受到 NullPointerException 的蹂躏和壁咚。Java 的发明者也承认这是一个巨大的设计错误。

那么关于 null ， 你应该知道下面这几件事情来有效的了解 null ， 从而避免很多由 null 引起的错误。

null 是大小写敏感

首先，null 是 Java 中的 关键字 ， 像是 public、static、final。它是大小写敏感的，你不能将 null 写成 Null 或 NULL，编辑器将不能识别它们然后报错。

这个问题已经几乎不会出现，因为 eclipse 和 Idea 编译器已经给出了编译器提示，所以你不用考虑这个问题。

null 是任何引用类型的初始值

null 是所有引用类型的默认值，Java 中的任何引用变量都将null作为默认值，也就是说所有 Object 类下的引用类型默认值都是 null。这对所有的引用变量都适用。就像是基本类型的默认值一样，例如 int 的默认值是 0，boolean 的默认值是 false。

下面是基本数据类型的初始值

null 只是一种特殊的值

null 既不是对象也不是一种类型，它仅是一种特殊的值，你可以将它赋予任何类型，你可以将 null 转换为任何类型

```
public static void main(String[] args) {
```

```
    String str = null;
    Integer itr = null;
    Double dou = null;

    Integer integer = (Integer) null;
    String string = (String)null;

    System.out.println("integer = " + integer);
    System.out.println("string = " + string);
}
```

你可以看到在编译期和运行期内，将 null 转换成任何的引用类型都是可行的，并且不会抛出空指针异常。

null 只能赋值给引用变量，不能赋值给基本类型变量。

持有 null 的包装类在进行自动拆箱的时候，不能完成转换，会抛出空指针异常，并且 null 也不能和基本数据类型进行对比

```
public static void main(String[] args) {
  int i = 0;
  Integer itr = null;
  System.out.println(itr == i);
}
```

使用了带有 null 值的引用类型变量， `instanceof` 操作会返回 false

```
public static void main(String[] args) {
  Integer isNull = null;
  // instanceof = isInstance 方法
  if(isNull instanceof Integer){
    System.out.println("isNull is instanceof Integer");
  }else{
    System.out.println("isNull is not instanceof Integer");
  }
}
```

这是 instanceof 操作符一个很重要的特性，使得对类型强制转换检查很有用

静态变量为 null 调用静态方法不会抛出 NullPointerException。因为静态方法使用了静态绑定。

使用 Null-Safe 方法

你应该使用 null-safe 安全的方法，java 类库中有很多工具类都提供了静态方法，例如基本数据类型的包装类，Integer , Double 等。例如

```
public class NullSafeMethod {

    private static String number;

    public static void main(String[] args) {
        String s = String.valueOf(number);
        String string = number.toString();
        System.out.println("s = " + s);
        System.out.println("string = " + string);
    }
}
```

number 没有赋值，所以默认为null，使用 String.value(number) 静态方法没有抛出空指针异常，但是使用 toString() 却抛出了空指针异常。所以尽量使用对象的静态方法。

null 判断

你可以使用 == 或者 != 操作来比较 null 值，但是不能使用其他算法或者逻辑操作，例如小于或者大于。跟SQL不一样，在Java中 null == null 将返回 true，如下所示：

```
public class CompareNull {

    private static String str1;
    private static String str2;

    public static void main(String[] args) {
        System.out.println("str1 == str2 ? " + str1 == str2);
        System.out.println(null == null);
    }
}
```

关于思维导图

我把一些常用的 Java 工具包的思维导图做了汇总，方便读者查阅。

Java.IO

Java.lang

Java.math

Java.net

干货分享

> 最近将个人学习笔记整理成册，使用PDF分享。关注我，回复如下代码，即可获得百度盘地址，无套路领取！
>
> - 001：《Java并发与高并发解决方案》学习笔记；
> - 002：《深入JVM内核——原理、诊断与优化》学习笔记；
> - 003：《Java面试宝典》
> - 004：《Docker开源书》
> - 005：《Kubernetes开源书》
> - 006：《DDD速成（领域驱动设计速成）》
> - 007：全部
> - 008：加技术讨论群

近期热文

- 彻底解决 GitHub 拉取代码网速慢的问题
- 基于 SpringBoot2 和 Netty 实现一个简易的RPC通信框架
- 一本彻底搞懂MySQL索引优化EXPLAIN百科全书
- 盘点 10 个代码重构的小技巧
- 性能测试如何定位瓶颈？偶发超时？看高手如何快速排查问题
- 震精！Spring Boot内存泄露，排查竟这么难！

想知道更多？长按/扫码关注我吧↓↓↓

>>>技术讨论群<<<

喜欢就点个"在看"呗^_^