JavaSE

概述

Java是面向对象、跨平台的一门编程语言。

特点

1. 跨平台
2. 面向对象
3. 提供一整套javase、web等开发组件
4. 强大的第三方组件支持

组成

1. jdk：java开发工具
2. jre：java运行环境
    - jvm：java虚拟机
    - 核心类库

面向对象四大特性

1. 抽象

   在面向对象的概念中，所有的对象都是通过类来描绘的。

2. 继承

   继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

3. 多态

   多态是同一个行为具有多个不同表现形式或形态的能力。

4. 封装

   封装（英语：Encapsulation）是指一种将抽象性函式接口的实现细节部份包装、隐藏起来的方法。

java核心机制

1. Java虚拟机（Java Virtual Machine）

   JVM是一个虚拟的计算机，具有指令集并使用不同的存储区域。负责执行指令，管理数据、内存、寄存器。

   Java虚拟机机制屏蔽了底层运行平台的差别，实现了“一次编译，到处运行”。

2. 垃圾收集机制（Garbage Collection）

   垃圾回收在 Java 程序运行过程中自动进行，程序员无法精确控制和干预。

内存分配

1. 栈

   一般存储基本数据类型、局部变量等数据

2. 堆

一般存储成员变量、静态的变量、对象等数据

关键字

被Java语言赋予了特殊含义，用做专门用途的字符串（单词）。

特点：

1. 关键字中所有字母都为小写
2. 包名、类名、方法名、变量名等不能使用关键字

标识符

Java 对各种变量、方法和类等要素命名时使用的字符序列称为标识符，凡是自己可以起名字的地方都叫标识符。

定义合法标识符规则：

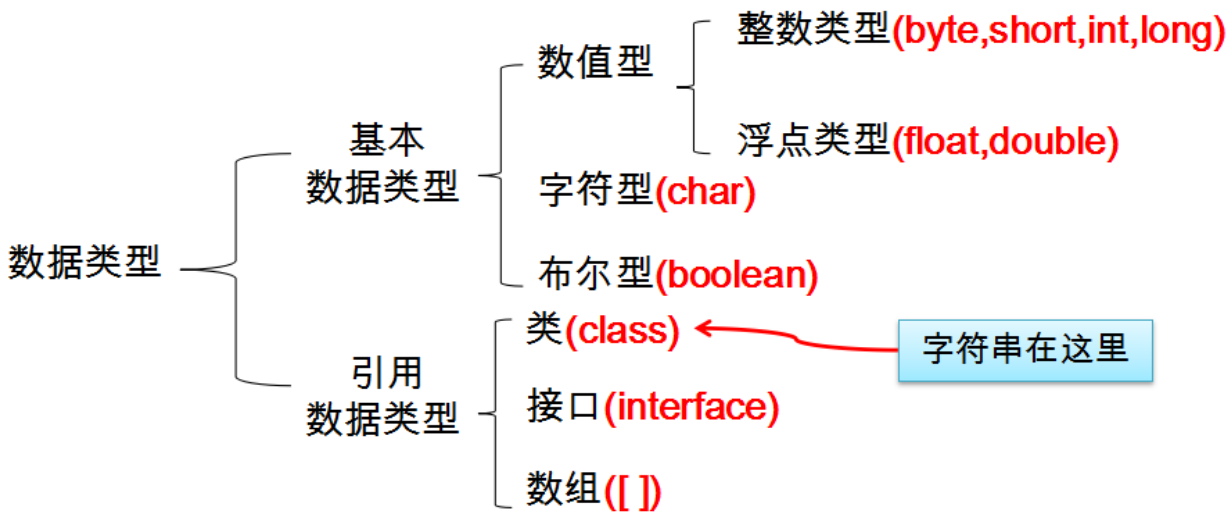
1. 由26个英文字母大小写，数字：0-9 ，_或 $ 组成
2. 数字不可以开头。
3. 不可以使用关键字和保留字，但能包含关键字和保留字。
4. Java中严格区分大小写，长度无限制。标识符不能包含空格。

变量

按声明位置

1. 成员变量：方法外部、类的内部定义的变量
2. 局部变量：方法或语句块内部定义的变量

按所属的数据类型



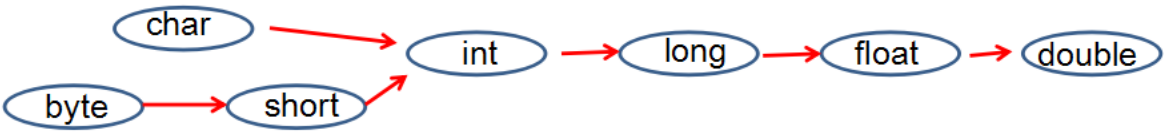
1. 基本数据类型
    - byte：$-2^7 \sim 2^7-1$，-128~127
    - short(2个字节)
    - int(4个字节)

- long(8个字节)
- float(4个字节)
- double(8个字节)
- boolean(1个字节)
- char(2个字节)：char可以存储一个汉字，字符常量是用单引号(' ')括起来的单个字符，比如 char c = '9';

2. 引用数据类型
   - 类（包括String）
   - 接口
   - 数组

基本数据类型转换

1. 自动转换



   - 有多种类型的数据混合运算时，系统首先自动将所有数据转换成容量最大的那种数据类型，然后再进行计算。
   - byte,short,char之间不会相互转换，他们三者在计算时首先转换为int类型。
   - 当把任何基本类型的值和字符串值进行连接运算时(+)，基本类型的值将自动转化为字符串类型。

2. 强制转换
   - 自动类型转换的逆过程，将容量大的数据类型转换为容量小的数据类型。使用时要加上强制转换符（()），但可能造成精度降低或溢出,格外要注意。如： long l = 2222222222L; int i = (int) l;
   - 通常，字符串不能直接转换为基本类型，但通过基本类型对应的包装类则可以实现把字符串转换成基本类型。如： String a = “43”; int i = Integer.parseInt(a);
   - boolean类型不可以转换为其它的数据类型。

注意：java默认的整数类型是int，小数数据类型是double

比如下面的定义是错误的：

byte b = 256; x

float f = 12112.22; x

进制

1. 所有数字在计算机底层都以二进制形式存在。
2. 计算机以补码的形式保存所有的整数。
3. 正数的补码与其原码相同；负数的补码是在其反码的末位加1。
4. 原码：直接将一个数值换成二进制数。
5. 反码：是对原码按位取反，只是最高位（符号位）保持不变。

运算符

1. 算术运算符

   i++/i--：先取值后运算

   ++i/--i：先运算后取值

   例一：

```
int x = 1;

int y = 2;

// 除了单目运算符、赋值运算符和条件运算符，其他的运算符都是从左到右结合的
y = x++ + ++x - --x;

//x++ + ++x：  x = 3, result = 4;

//4 - --x: x = 2, result = 2

//结果：x = 2, y = 2
```

   例二：

```
int x = 1;
if(x++ == 2){
    System.out.println("I am ok!");
}
if(++x == 3){
   System.out.println("Are you ok?");
}

//结果：Are you ok?
```

2. 逻辑运算符

   - “&”和“&&”的区别：

     单&时，左边无论真假，右边都进行运算；

     双&时，如果左边为真，右边参与运算，如果左边为假，那么右边不参与运算。

   - |和“||”的区别同理，双或时，左边为真，右边不参与运算。
   - 异或(^)与或(|)的不同之处是：对于 ^ 而言，当左右都为true时，结果为false。

     1 ^ 1 = false

     1^ 0 = true

     0 ^ 0 = false

3. 位运算符

   <<：被移除的高位丢弃，空缺位补0。左移n位值变大2^n次方，比如：1<<2 = 4

   >>：被移位的二进制最高位是0，右移后，空缺位补0；最高位是1，空缺位补1。右移n位值变小2^n次方，比如：8>>2 = 2

>>>：被移位二进制最高位无论是0或者是1，空缺位都用0补。

流程控制

1. switch

   switch(表达式)中表达式的返回值必须是下述几种类型之一：byte, char, short, int, 枚举, 字符串；

   当某个case分支满足条件执行但没有break语句，程序会继续执行直到遇到break语句，不管在它之后case分支是否成立。

2. break
   - break语句用于终止某个语句块的执行，如果没指定标签默认跳出离它最近的循环。

     { ..... break; ..... }

     break语句出现在多层嵌套的语句块中时，可以通过标签指明要终止的是哪一层语句块

     label1: { .....

     label2: { .....

     label3: { .....

     break label2;

     .....

     }

     }

     }

3. continue
   - continue语句用于跳过某个循环语句块的一次执行
   - continue语句出现在多层嵌套的循环语句体中时，可以通过标签指明要跳过的是哪一层循环

数组

1. 数组是多个**相同类型数据**的组合，实现对这些数据的统一管理
2. 数组属引用类型，数组型数据是对象(Object)，数组中的每个元素相当于该对象的成员变量
3. 数组中的元素可以是任何数据类型，包括基本类型和引用类型
4. **数组一旦定义长度，后面就不能再改变其长度。**

一维数组

1. int[] nums = {};
2. int[] nums = new int[5];
3. int[] nums = new int[]{};

二维数组

1. int[][] nums = {{1,2,3},{4,5,6}};
2. int[][] nums = new int[2][];.
3. int[][] nums = new int[2][2];

4. int[][] numbers = new int[][]{};

类和对象

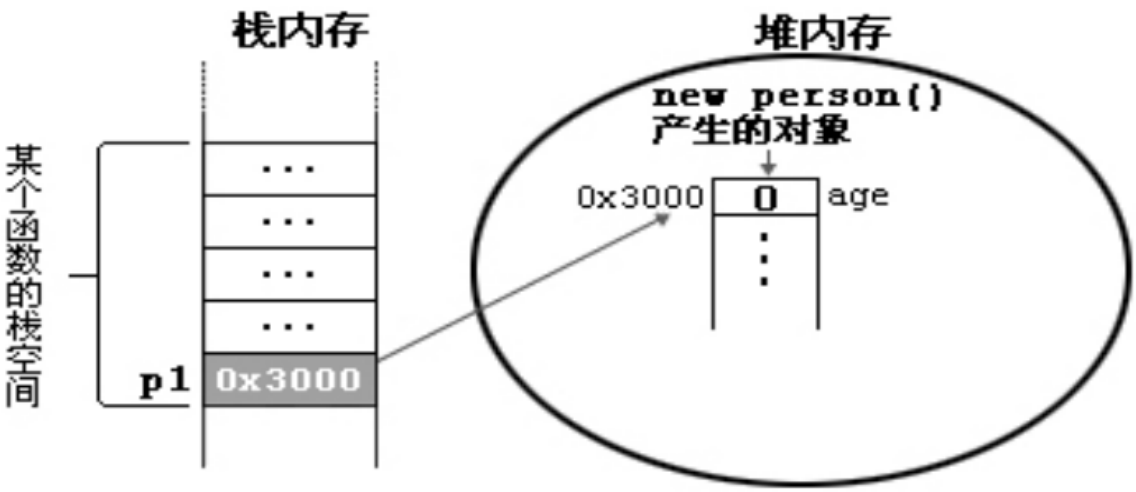
类(class)和对象(object)是面向对象方法的核心概念。

1. **类是对一类事物的描述，是抽象的、概念上的定义；对象是实际存在的该类事物的每个个体，因而也称实例(instance)。**
2. 面向对象程序设计的重点是类的设计，而不是对象的设计。

对象的产生

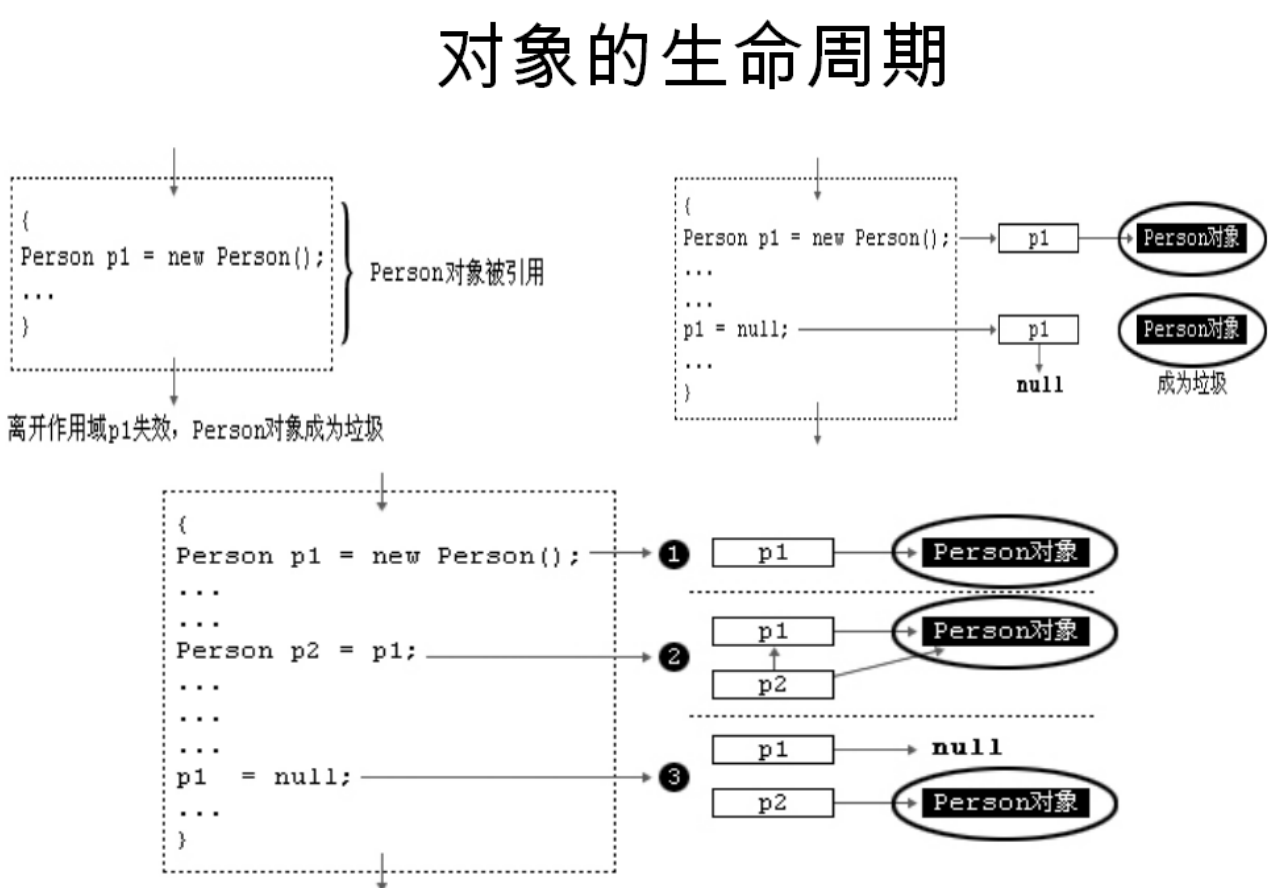
```
class Person
{
    int age;
    void shout()
    {
        System.out.println(“oh,my god! I am “ + age);
    }

    public static void main(String[] args){
        Person p1 = new Person();
        p1.shout();
    }
}
```



成员变量类型	初始值
byte	0
short	0
int	0
long	0L
float double	0.0F 0.0D
char boolean	'\u0000'（表示为空） False
All reference type	Null

对象的生命周期



构造方法

特点

1. 它具有与类相同的名称；
2. 它不含返回值；

作用

当一个类的**实例对象刚产生时**，这个类的构造方法就会被自动调用，我们可以在这个方法中加入要完成**初始化工作**的代码。

默认构造方法

如果类的定义者没有显式的定义任何构造方法，系统将自动提供一个默认的无参、没有方法体的构造方法。

一旦类的定义者显式定义了一个或多个构造方法，系统将不再提供默认的构造方法；

方法

形参

在方法被调用时用于接收外部传入的数据的变量。

实参

调用函数时实际传给函数形式参数的数据。

方法重载

方法的重载就是在**同一个类中**允许同时存在一个以上的**同名方法**，只要它们的**参数个数或类型不同**即可。

方法重写

方法重写就是子类对从父类继承过来的方法进行覆盖。

方法的重写和重载区别

1. 方法重载是指同一类中方法名相同，参数个数或者类型不同的方法，跟权限修饰符、返回值无关
2. 方法重写是指存在继承关系的两个类中，子类将父类的方法进行重写，重写的方法必须和被重写的方法具有相同的方法名称、参数列表和返回值类型，另外重写的方法不能使用比被重写的方法更严格的访问权限，比如父类的方法是protected，子类只能使用protected、public修饰

方法参数传递

1. 值传递

   一般是基本数据类型参数，相当于值的复制

2. 引用传递

   一般是对象参数，相当于对象的赋值

继承

多个类中存在相同的属性和行为，将这些内容抽取到一个单独的类中，那么多个类无需再定义这些属性和行为，只要继承这个类即可，这种思想叫做继承。

特点

1. 子类只能继承父类非私有的属性和方法（不包括构造方法）
2. Java只支持单继承，不允许多重继承

继承的优点

1. 提高代码的重用性
2. 让类与类之间产生关系，是多态的前提

继承的缺点

1. 破坏的类的封装
2. 耦合性高

this和super

1. this指当前对象的引用变量
    - 访问当前对象的属性

      this.name = "123";
    - 访问当前对象的构造方法

      this();
    - 访问当前对象的方法

      this.show();
2. super指当前所在类的父类对象的引用变量
    - 访问父类对象的属性

      super.name = "123";
    - 访问父类对象的构造方法

      super();
    - 访问父类对象的方法

      super.show();

访问控制

Java有四种访问权限，其中三种有访问权限修饰符，分别为private，public和protected，还有一种不带任何修饰符：

1. **private**: Java语言中对访问权限限制的最窄的修饰符，一般称之为“私有的”。被其修饰的类、属性以及方法只能被该类的对象访问，其子类不能访问，更不能允许跨包访问。

2. **default**：即不加任何访问修饰符，通常称为“默认访问模式“。该模式下，只允许在同一个包中进行访问。
3. **protect**: 介于public 和 private 之间的一种访问修饰符，一般称之为“保护形”。被其修饰的类、属性以及方法只能被类本身的方法及子类访问，即使子类在不同的包中也可以访问。
4. **public**： Java语言中访问限制最宽的修饰符，一般称之为“公共的”。被其修饰的类、属性以及方法不仅可以跨类访问，而且允许跨包（package）访问。下面用表格的形式来展示四种访问权限之间的异同点，这样会更加形象。注意其中protected和default的区别，表格如下所示：

修饰符	同一个类	同一个包	不同包的子类	不同包的非子类
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

多态

对象在不同时刻表现出来不同的形态叫多态。

前提条件

1. 要有继承或者实现关系
2. 要有方法的重写
3. 要有父类引用指向子类对象

多态优点

多态的存在提高了程序的扩展性和后期维护性。

多态缺点

父类调用的时候只能调用父类的方法，不能调用子类的特有的方法，因为父类不清楚将来会有什么样的子类继承它。

对象之间的转换

对Java对象的强制类型转换称为造型。

1. 从子类到父类的类型转换可以自动进行
2. 从父类到子类的类型转换必须通过造型(强制类型转换)实现
3. 无继承关系的引用类型间的转换是非法的
4. 在造型前可以使用**instanceof**操作符测试一个对象的类型

Object

Object类是所有Java类的根父类。

方法

1. getClass();
2. hashCode();
3. equals();
4. toString();
5. notify()/notifyAll();
6. wait();
7. clone();
8. finalize();

final、finalize、finally区别

1. final是修饰符，被它修饰的类、属性、方法不能重写
2. finalize是Object的方法，它会在被垃圾回收器回收时调用，可以利用它做一些释放资源、收尾等操作
3. finally是异常处理的代码块，被它包含的代码不管有没有发生异常都会执行，一般用来释放资源、收尾等操作

equals和==区别

==：

1. 如果比较数据是基本数据类型：byte、short、int、long、float、double、char、boolean，==比较的是两者的值是否相等
2. 如果比较数据是引用数据类型:对象、数组、接口、字符串等，==比较的是两者的内存地址是否相同

equals：

1. 如果比较数据是基本数据类型：byte、short、int、long、float、double、char、boolean，编译报错，equals不能用于基本数据类型比较
2. 如果比较数据是引用数据类型:对象、数组、接口、字符串等，并且没有重写Object的hashCode、equals方法，equals和==作用一样，所以说要用equals比较引用数据类型必须重写Object的hashCode和equals方法

static

在Java类中声明变量、方法和内部类时，可使用关键字static做为修饰符。

static标记的变量或方法由整个类(所有实例)共享，如访问控制权限允许，可不必创建该类对象而直接用类名加'.'调用。

一个类只有一个class对象，可以有多个实例对象，所有的实例对象共享一个class对象。

静态代码块

一个类中可以使用不包含在任何方法体中的静态代码块(static block)，当类被载入时，静态代码块被执行，且只被执行一次，静态块经常用来进行类属性的初始化。

```
static {
            ...
    }
```

抽象类

用abstract关键字来修饰一个类时，这个类叫做抽象类。

特点

1. 抽象类不能被实例化
2. 抽象方法和抽象类必须被abstract修饰
3. 含有抽象方法的类一定是抽象类
4. 抽象类可以有抽象方法，也可以有非抽象方法
5. 抽象类的方法调用，必须由子类对继承它，重写所有的抽象方法才能调用，如果只是部分重写抽象方法，子类也是抽象类
6. 抽象类可以定义成员变量、构造方法。

优点

1. 抽出共有的特性，提高代码重用性
2. 实现类个性化、多样化

接口

接口是抽象方法和常量的集合，实质上，接口其实是特殊的抽象类。

特点

1. 接口不能实例化
2. 接口只允许有抽象方法和常量，就算定义了变量实际上jvm在前面使用public static final修饰
3. 一个类如果实现了接口，要么是抽象类，要么全部实现接口的所有抽象方法

优点

1. 接口可以多实现
2. 接口实现模块化开发，提高开发效率
3. 接口可以降低程序间的耦合性

继承和实现的区别

1. 一个类可以实现多个接口，但只能继承一个类
2. 接口可以继承接口
3. 类与类之间关系是继承关系
4. 类与接口之间关系是实现关系
5. 接口与接口之间是继承关系

抽象类和接口的区别

1. 成员变量
    - 抽象类可以有变量，也可以有常量
    - 接口只能有常量
2. 成员方法
    - 抽象类可以有抽象方法，也可以有非抽象方法
    - 接口只有抽象方法
3. 构造方法
    - 抽象类有构造方法
    - 接口没有构造方法
4. 类与抽象类和接口的关系
    - 类和抽象类的关系是继承关系，使用extends来继承
    - 类和接口的关系是实现关系，使用implements来实现
5. 应用场景
    - 有共同的属性和方法，允许个性化的扩展，这时候使用抽象类合适

```
public abstract class AnimalAbstract {
    String cellType = "没有细胞壁";

    public void eat();
}

// 狗
public class Dog extends AnimalAbstract {
    public void eat() {
        System.out.println("吃骨头");
    }
}

// 猫
public class Cat extends AnimalAbstract {
    public void eat() {
        System.out.println("吃鱼");
    }
}
```

    - 只是定义行为规范，允许个性化实现，实现的类不属于同一类事物，这时候使用接口合适

```
public interface Fly {
```

```
    public void fly();
}

// 鸟
public class Bird implements Fly {

    public void fly(){
        System.out.println("使劲飞");
    }
}

// 飞机
public class Plane implements Fly {

    public void fly(){
        System.out.println("加油飞");
    }
}
```

异常

异常：在Java语言中，将程序执行中发生的不正常情况称为“异常”。

Throwable种类

1. error

   JVM系统内部错误、资源耗尽等严重情况。

2. exception

   其它因编程错误或偶然的外在因素导致的一般性问题。

Exception种类

1. 编译期异常（非运行期异常）

   在编译的时候发生的异常，这种异常需要手动处理。

   - IOException
   - FileNotFoundException

2. 运行期异常

   在程序运行过程中发生的异常，这种不需要手动处理。

   - NullPointerException
   - ArrayIndexOutBoundsException
   - ArithmeticException

异常处理方式

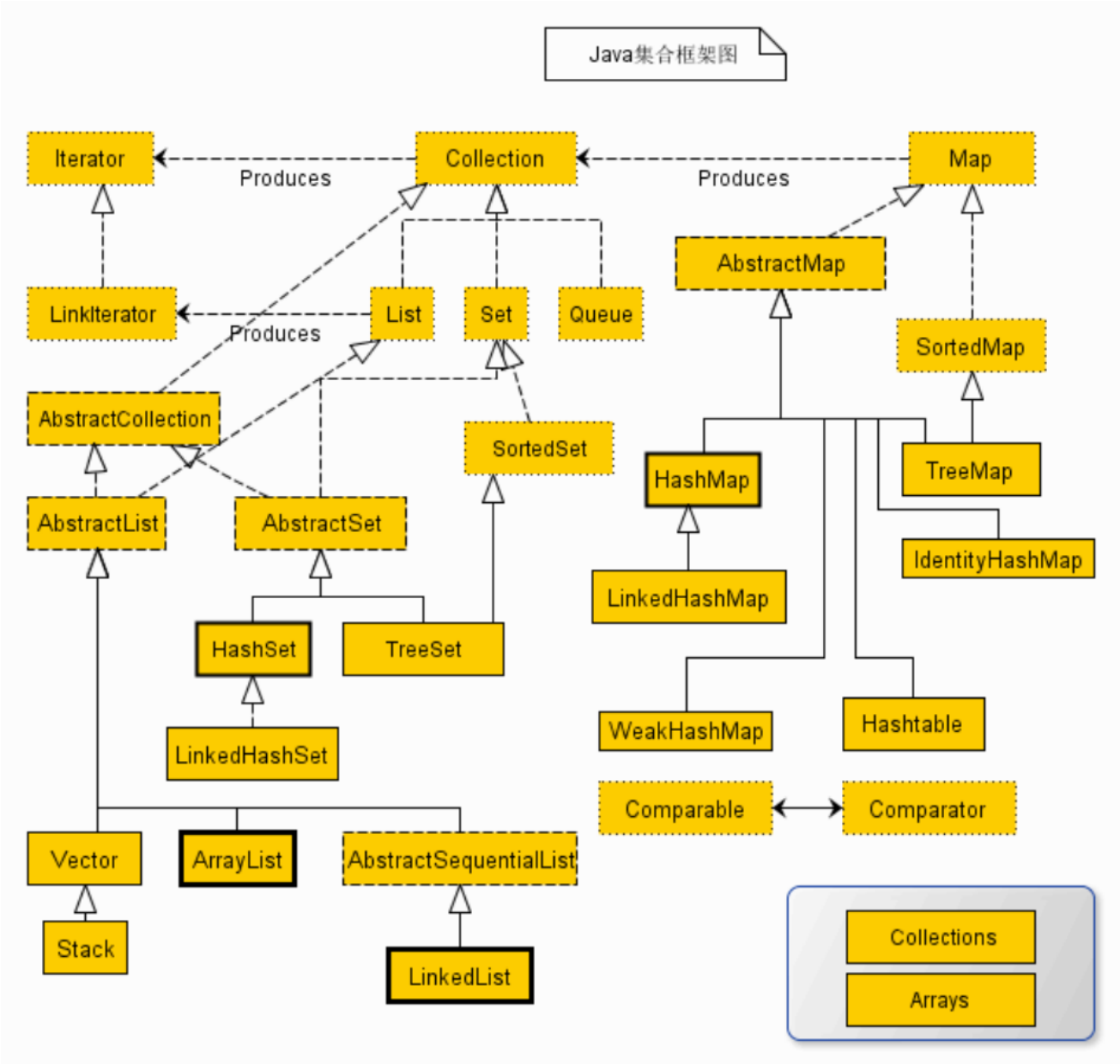
1. 捕获，通过try...catch...finally

```
try {
    ...
} catch (Exception e){
    ...
} finally {
    ...
}
```

2. 抛出，通过throws（在方法上）和throw（在方法内）抛出

```
public void method() throws Exception {
    ...
    throw new Exception("密码错误！");
}
```

集合



集合框架

1. Collection子接口：
    - List实现类
        - ArrayList(线程不安全)
        - LinkedList(线程不安全)
        - Vector(线程安全)
    - Set实现类
        - HashSet(无序)
        - TreeSet(有序)
2. Map接口实现类
    - HashMap(无序)
    - TreeMap(有序)

List特点

1. 元素允许重复
2. 元素有序

Set特点

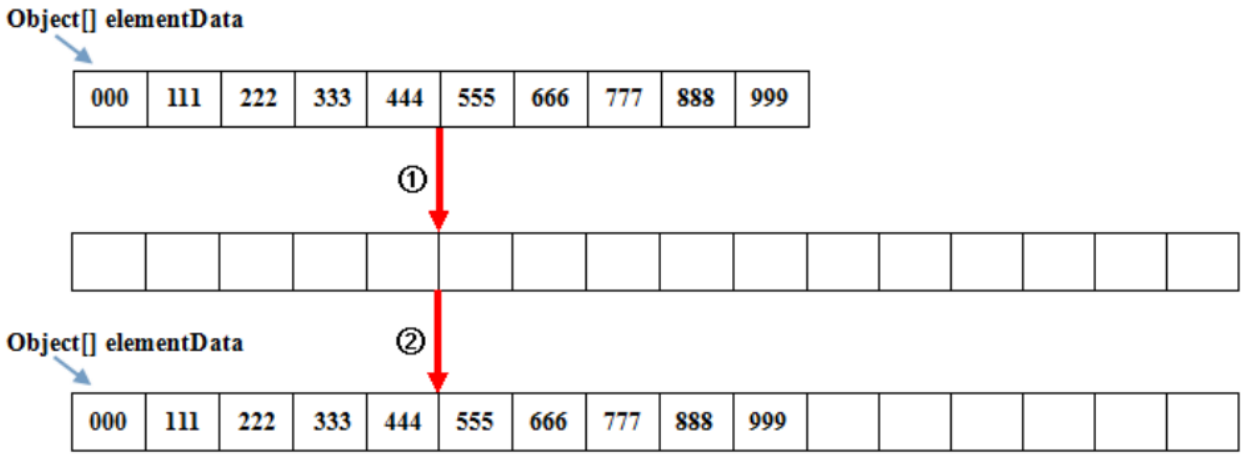
1. 元素不允许重复，重复元素会覆盖
2. 元素无序(HashSet)

Map特点

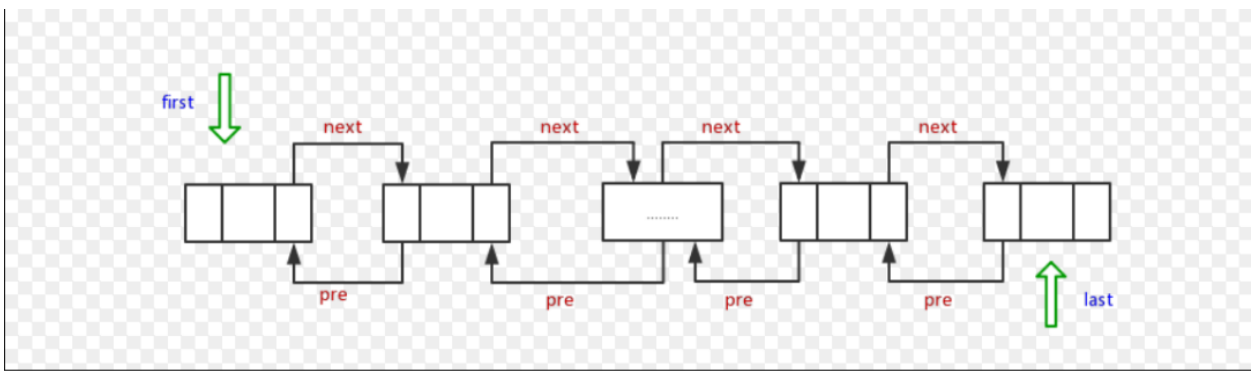
1. 元素是key-value键值对形式
2. 元素无序（HashMap）

ArrayList和LinkedList区别

1. ArrayList底层是**数组**实现，它默认创建一个长度为10的数组，当新添加的元素已经没有位置存放的时候，ArrayList就会自动进行扩容，扩容的长度为原来长度的1.5倍+1。它的线程是不安全的。



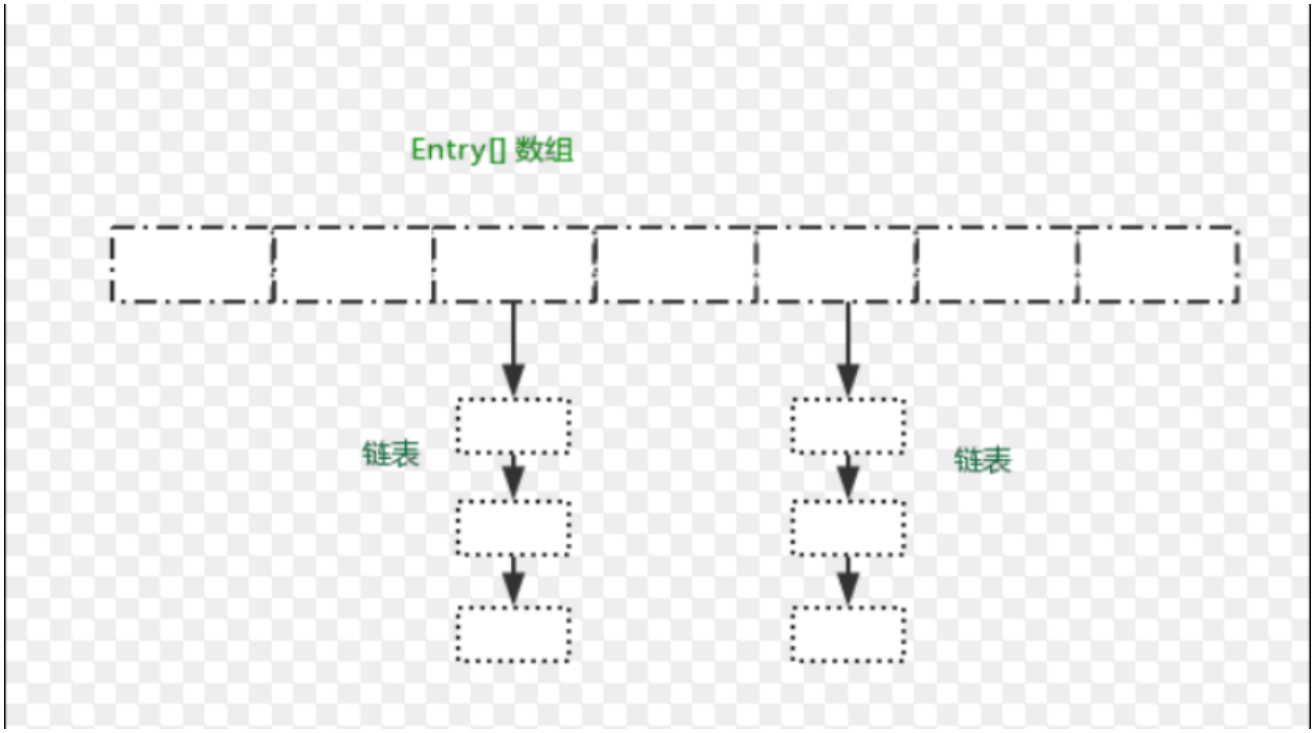
2. LinkedList底层是**双向链表**实现，既然是链表实现那么它的随机访问效率比ArrayList要低，顺序访问的效率要比较的高。每个节点都有一个前驱（之前前面节点的指针）和一个后继（指向后面节点的指针）。



3. ArrayList适合在查询多，修改少的场景；LinkedList适合在查询少，修改多的场景。

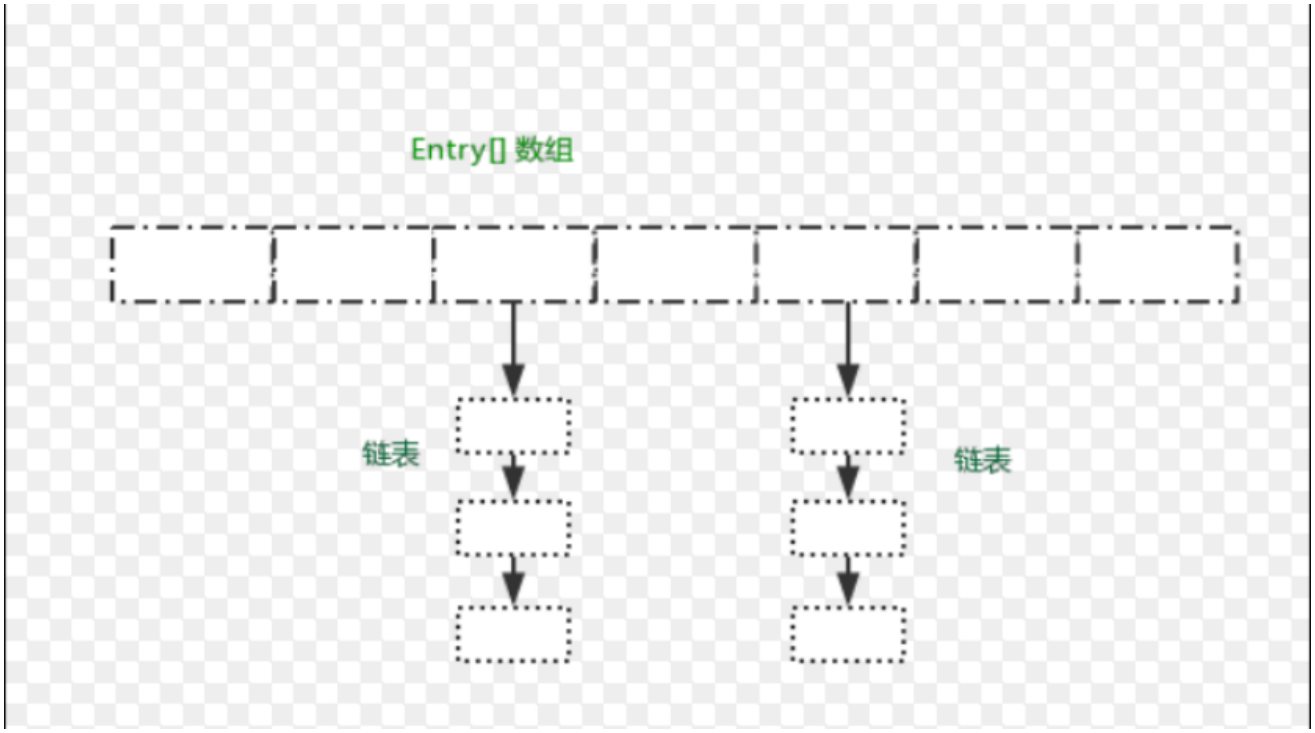
HashSet

HashSet的底层通过HashMap实现的，而HashMap在1.7之前使用的是**数组+链表**实现，在1.8+使用的数组+链表+红黑树实现。其实也可以这样理解，HashSet的底层实现和HashMap使用的是相同的方式，因为Map是无序的，因此HashSet也无法保证顺序。HashSet的方法也是借助HashMap的方法来实现的。



HashMap

HashMap采用Entry数组来存储key-value对，每一个键值对组成了一个Entry实体，Entry类实际上是一个单向的链表结构，它具有Next指针，可以连接下一个Entry实体，依次来解决Hash冲突的问题，因为HashMap是按照Key的hash值来计算Entry在HashMap中存储的位置的，如果hash值相同，而key内容不相等，那么就用链表来解决这种hash冲突。



泛型

泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

```
List list = new ArrayList();
list.add(123);
list.add("asd");
list.add(new Person());

Iterator it = list.iterator();
while(it.hasNext()){
    //报错
    int i = (int) it.next();
    //报错
    Person person = (Person) it.next();

    //正确
    Object object = it.next();
    if(object instanceof Person){
        Person person = (Object) object;
    }
}
```

应用泛型

```
//使用泛型，类型检查，安全
List<String> list = new ArrayList<>();
```

```
//报错
list.add(123);
//正确
list.add("asd");
//报错
list.add(new Person());

Iterator<String> it = list.iterator();
while(it.hasNext()){
    //正确
    String str = it.next();
}
```

泛型优点

1. 提供了编译时类型安全检测机制
2. 增加可读性和稳定性
3. 避免类型强转出错，不需要类型判断再强转

通配符

Collection<?>其中？是通配符，它的元素类型可以匹配任何类型。比如：

```
//具体实例化确定类型
Collection<?> collection = new ArrayList<String>();

//传入参数时确定类型
public void collection(Collection<?> collection){
    ...
}
```

有限制的通配符

允许指定它的父类、子类类型，使用extends、super

```
//传入参数时确定类型
public static void collection(Collection<? extends Person> collection){
    ...
}

public static void main(String[] args){
    List<String> list = new ArrayList<>();
    list.add("asd");

    //list元素不是person，报错
    collection(list);
}
```

泛型方法

```
//泛型类
public class Page<T> {

    public int getData(T t){

    }
}

//泛型方法
public class Page {

    public <T> int getData(T t){

    }
}
```

枚举和注解

枚举类和普通类的区别：

1. 使用 enum 定义的枚举类默认继承了 java.lang.Enum 类
2. 枚举类的构造器只能使用 private 访问控制符
3. 枚举类的所有实例必须在枚举类中显式列出(, 分隔 ; 结尾). 列出的实例系统会自动添加 public static final 修饰
4. 属性使用 private final 修饰所有的
5. 枚举类都提供了一个 values 方法, 该方法可以很方便地遍历所有的枚举值

```
enum SeasonEnum{

    SPRING("春天", "春风又绿江南岸"),
    SUMMER("夏天", "映日荷花别样红"),
    AUTUMN("秋天", "秋水共长天一色"),
    WINTER("冬天", "窗含西岭千秋雪");

    private final String seasonName;
    private final String seasonDesc;

    private SeasonEnum(String seasonName, String seasonDesc){
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }

    public String getSeasonName() {
        return seasonName;
    }

    public String getSeasonDesc() {
        return seasonDesc;
    }
}
```

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。

jdk自带的注解

1. @Override: 限定重写父类方法, 该注释只能用于方法
2. @Deprecated: 用于表示某个程序元素(类, 方法等)已过时
3. @SuppressWarnings: 抑制编译器警告.

自定义注解

1. 定义新的 Annotation 类型使用 @interface 关键字
2. 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.
3. 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
4. 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

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Controller{

    String name() default "";
}
```

注解的作用时机

1. SOURCE
2. CLASS
3. **RUNTIME**

IO

分类

1. 流向划分
    - 输入流
    - 输出流
2. 流处理的单位划分
    - 字节流
    - 字符流
3. 流的角色划分
    - 节点流
    - 处理流

IO体系

分类	字节输入流	字节输出流	字符输入流	字符输出流
抽象基类	InputStream	OutputStream	Reader	Writer
访问文件	FileInputStream	FileOutputStream	FileReader	FileWriter
访问数组	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
访问管道	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
访问字符串			StringReader	StringWriter
缓冲流	BufferedInputStream	BufferedOuputStream	BufferedReader	BufferedWriter
转换流			InputStreamReader	OutputStreamWriter
对象流	ObjectInputStream	ObjectOutputStream		
抽象基类	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
打印流		PrintStream		PrintWriter
推回输入流	PushbackInputStream		PushbackReader	
特殊流	DataInputStream	DataOutputStream		

序列化

1. 序列化的类必须实现Serializable接口
2. 调用 ObjectOutputStream 对象的 writeObject() 方法输出

反序列化

1. 调用 ObjectInputStream对象的 readObject() 方法读取

RandomAccessFile

在大文件读取时，可以随机读写文件的任意位置的内容。

创建对象的方式

1. new
2. 反射
3. clone
    - 浅克隆：只克隆对象的内容，但不能克隆其他引用的对象
    - 深度克隆：克隆所有的对象内容，包括引用的对象
4. 反序列化

常用类

String 、StringBuffer、StringBuilder 、Date 、DateFormat 、Random、Math 等

String、StringBuffer和StringBuilder区别

String：内容不可变

StringBuffer：内容可变，线程安全

StringBuilder：内容可变，线程不安全

反射

反射指从正在运行的程序中访问任意类的内部信息，并能直接操作任意对象的内部属性及方法。

反射和反编译区别

1. 反射指从正在运行的程序中访问类的内部信息。
2. 反编译指从.class文件得到编译前的源代码

Class对象

Class 对象只能由系统建立对象一个类。在 JVM 中只会有一个Class实例。每个类的实例都会记得自己是由哪个 Class 实例所生成。

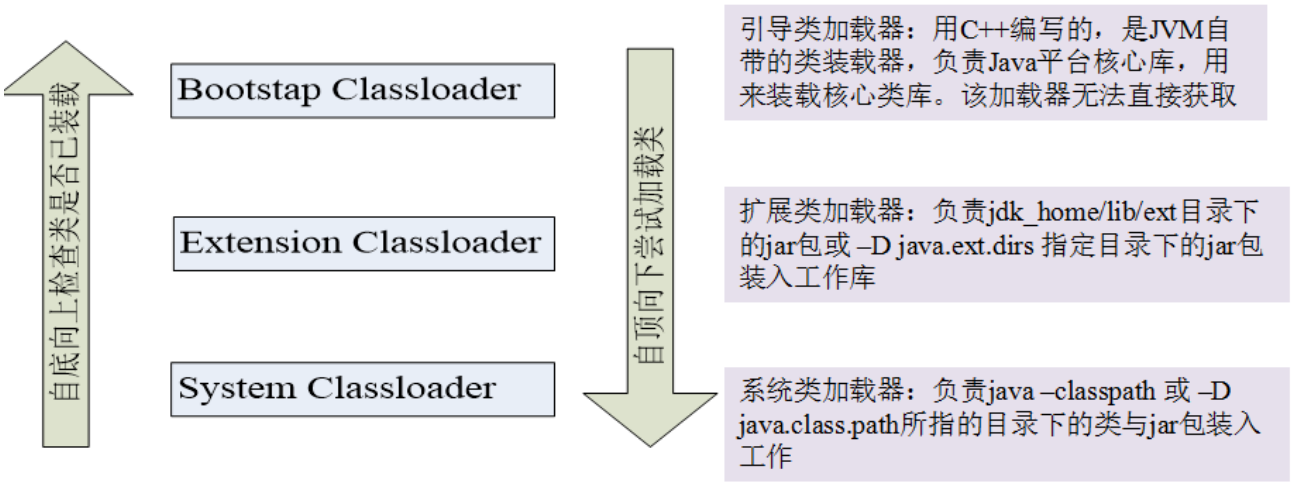
获取Class对象方式

1. 实例对象.getClass();
2. 类.class
3. Class.forName(类路径);
4. ClassLoader.getSystemClassLoader().loadClass(类路径)

ClassLoader

ClassLoader

类装载器是用来把类(class)装载进 JVM 的。JVM 规范定义了两种类型的类装载器：启动类装载器(bootstrap)和用户自定义装载器(user-defined class loader)。 JVM在运行时会产生3个类加载器组成的初始化加载器层次结构 ，如下图所示：



反射优点

1. 可以动态访问和修改程序
2. 可以实现IOC、动态代理等高级功能，比如spring、mybatis

反射缺点

1. 不安全，私有的属性、方法通过反射可以被访问和修改

```
Method[] methods = clazz.getMethods();
for (Method method : methods) {
    //设置访问权限，true为可以访问
    method.setAccessible(true);
}
```

2. 反射需要耗费cpu等资源，大量使用会拖慢系统速度