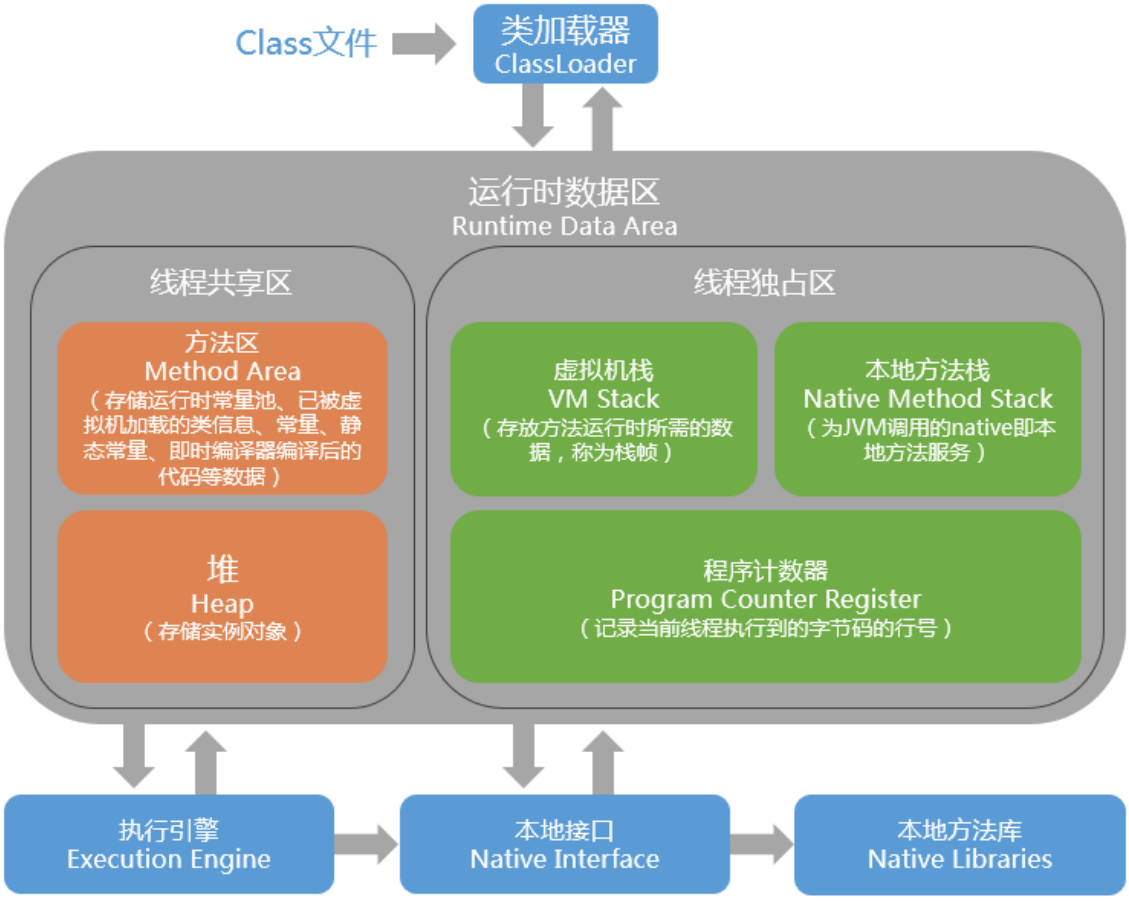
corejava-day06

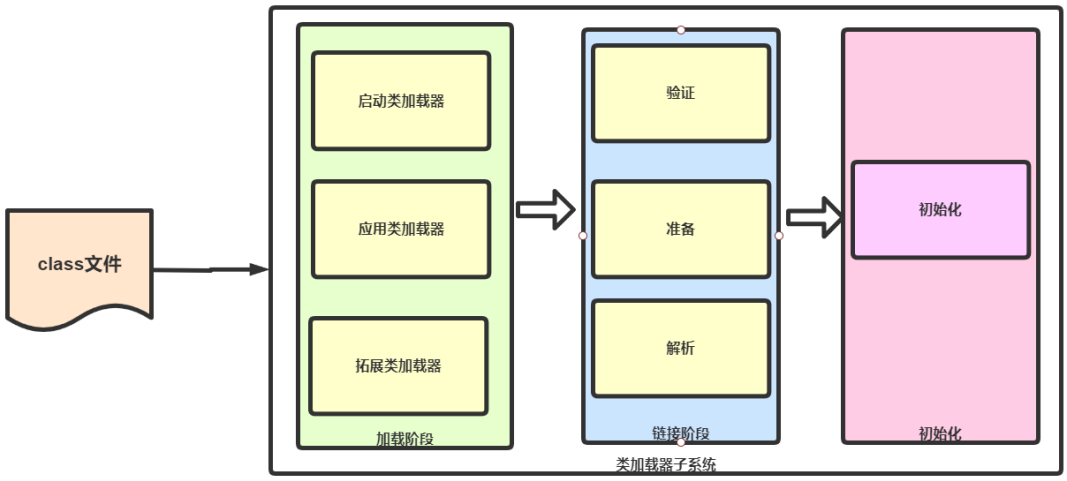
0.前置知识

> 彻底终结jvm内存模型

- jvm内存模型



- 类加载器子系统



- 类加载器只负载加载class文件,至于这个类是否被执行有执行引擎决定，每个class都有特定的标识，称为 魔数
- 加载类的信息存放一块存放于一块方法区的空间。

- 方法区

Method Area (方法区)

虚拟机已加载的类信息

Class1	Class2	Class3.....n
1、类型信息	1、类型信息	
2、类型的常量池	2、类型的常量池	
3、字段信息	3、字段信息	
4、方法信息	4、方法信息	
5、类变量	5、类变量	
6、指向类加载器的引用	6、指向类加载器的引用	
7、指向Class实例的引用	7、指向Class实例的引用	
8、方法表	8、方法表	

运行时常量池

> 类型信息
>
> - 类的完整名称（比如，java.long.String）
> - 类的直接父类的完整名称（java.lang.Object）
> - 类的直接实现接口的有序列表（因为一个类直接实现的接口可能不止一个，因此放到一个有序表中）

- 类的修饰符
  可以看做是，对一个类进行登记，这个类的名字叫啥，他爹是谁、有没有实现接口， 权限是啥；

类型的常量池 （运行时常量池）

- 每一个Class文件中，都维护着一个常量池（这个保存在类文件里面，不要与方法区的运行时常量池搞混），里面存放着编译时期生成的各种字面值和符号引用；这个常量池的内容，在类加载的时候，被复制到方法区的运行时常量池；
- 字面值：就是像string, 基本数据类型，以及它们的包装类的值，以及final修饰的变量，简单说就是在编译期间，就可以确定下来的值；

类变量(即static变量)

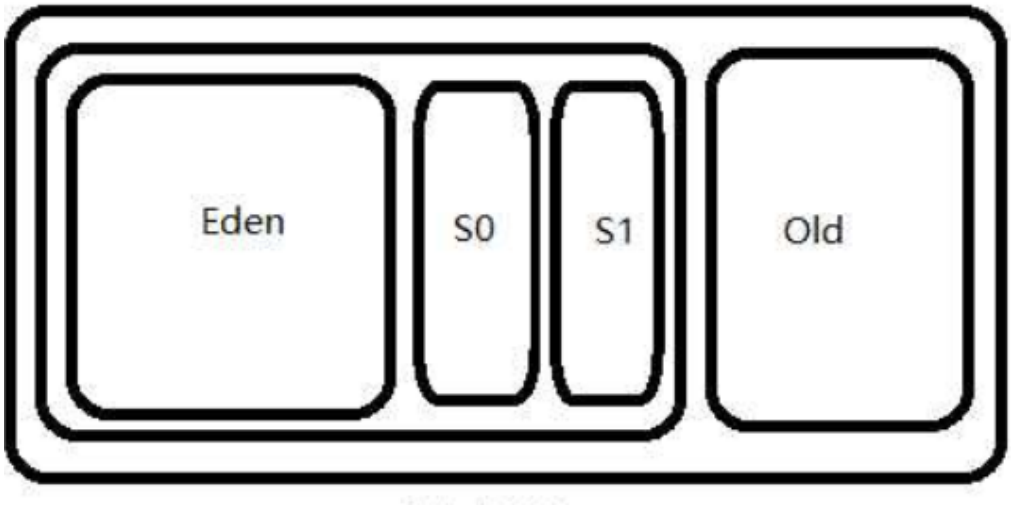
- 非final类变量
- 在java虚拟机使用一个类之前，它必须在方法区中为每个非final类变量分配空间。非final类变量存储在定义它的类中；

  final类变量（不存储在这里）

  - 由于final的不可改变性，因此，final类变量的值在编译期间，就被确定了，因此被保存在类的常量池里面，然后在加载类的时候，复制进方法区的运行时常量池里面； final类变量存储在运行时常量池里面，每一个使用它的类保存着一个对其的引用；

- 堆

  可以分为两个部分：年轻代和老年代(同时jdk1.8之后永久代也将被移除)



jdk8 以后移除了永久代

新生代

> 新生成的对象优先存放在新生代中，新生代对象朝生夕死，存活率很低，在新生代中，常规应用进行一次垃圾收集一般可以回收70% ~ 95% 的空间，回收效率很高。
>
> HotSpot将新生代划分为三块，一块较大的Eden空间和两块较小的Survivor空间，默认比例为8：1：1。划分的目的是因为HotSpot采用复制算法来回收新生代，设置这个比例是为了充分利用内存空间，减少浪费。新生成的对象在Eden区分配（大对象除外，大对象直接进入老年代），当Eden区没有足够的空间进行分配时，虚拟机将发起一次 `Minor GC`
>
> GC开始时，对象只会存在于Eden区和Survivor 0区，S1区是空的（作为保留区域）。GC进行时， `Eden` 区中所有存活的对象都会被复制到S1区，而在 `S0` 区中，仍存活的对象会根据它们的年龄值决定去向，年龄值达到年龄阀值（默认为15，新生代中的对象每熬过一轮垃圾回收，年龄值就加1，GC分代年龄存储在对象的 `header` 中）的对象会被移到老年代中。
>
> 没有达到阀值的对象会被复制到 `s1` 区。接着清空 `Eden` 区和 `s0` 区，新生代中存活的对象都在 `s1` 区。接着， `s0` 区和 `s1` 区会交换它们的角色，也就是新的 `s1` 区就是上次GC清空的 `s0` 区，新的 `s0` 区就是上次GC的 `s1` 区，总之，不管

> 怎样都会保 `s1` 区在一轮GC后是空的。GC时当 `s1` 区没有足够的空间存放上一次新生代收集下来的存活对象时，需要依赖老年代进行分配担保，将这些对象存放在老年代中。

老年代

> 在新生代中经历了多次（具体看虚拟机配置的阀值）GC后仍然存活下来的对象会进入老年代中。老年代中的对象生命周期较长，存活率比较高，在老年代中进行GC的频率相对而言较低，而且回收的速度也比较慢。

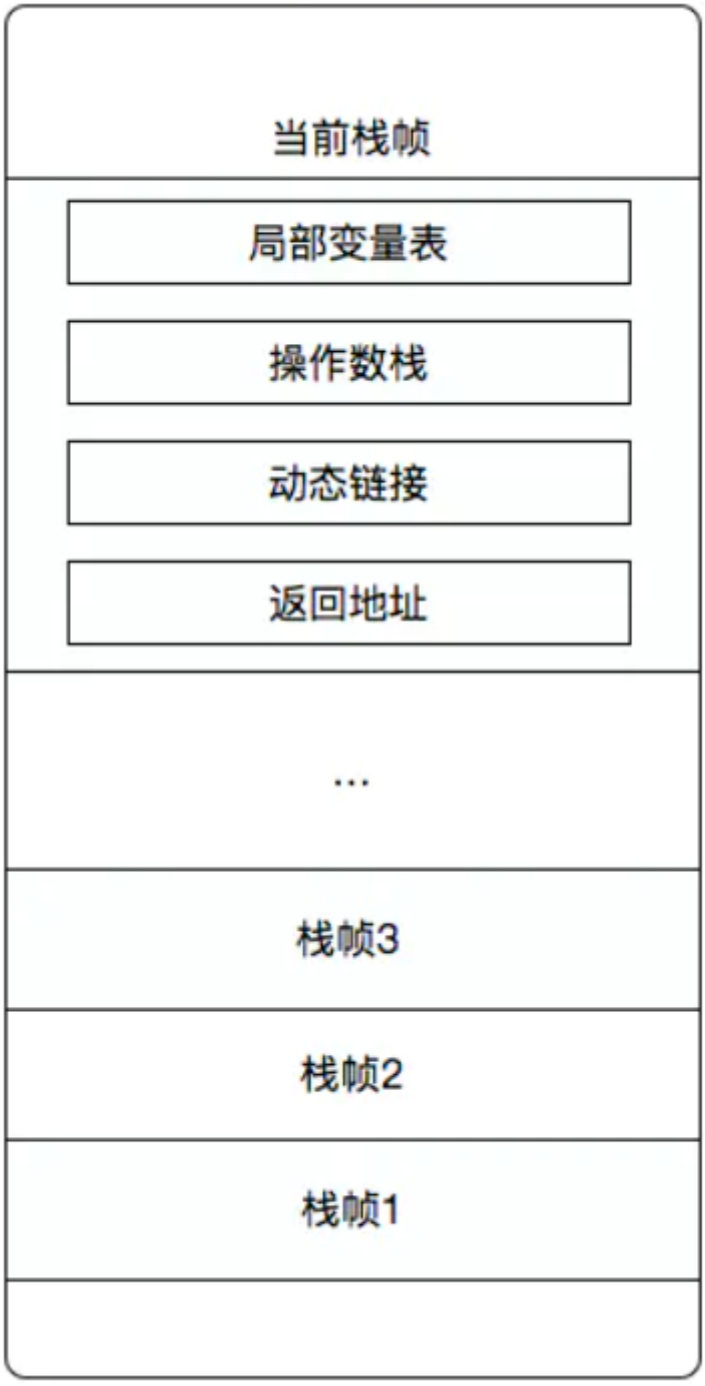
垃圾回收

> 新生代GC（Minor GC）：Minor GC指发生在新生代的GC，因为新生代的Java对象大多都是朝生夕死，所以Minor GC非常频繁，一般回收速度也比较快。当Eden空间不足以为对象分配内存时，会触发Minor GC
>
> 老年代GC（Full GC/Major GC）：Full GC指发生在老年代的GC，出现了Full GC一般会伴随着至少一次的Minor GC（老年代的对象大部分是Minor GC过程中从新生代进入老年代），比如：分配担保失败。Full GC的速度一般会比Minor GC慢10倍以上。当老年代内存不足或者显式调用System.gc()方法时，会触发Full GC。

- 虚拟机栈

> 描述的是java方法执行的内存模型
>
> 每个方法被执行的时候都会创建一个”栈帧”,用于存储局部变量表(包括参数)、操作栈、方法出口等信息。

> 每个方法被调用到执行完的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。栈帧由四部分组成：局部变量表、操作数栈、动态链接,方法返回。
>
> 局部变量表一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量

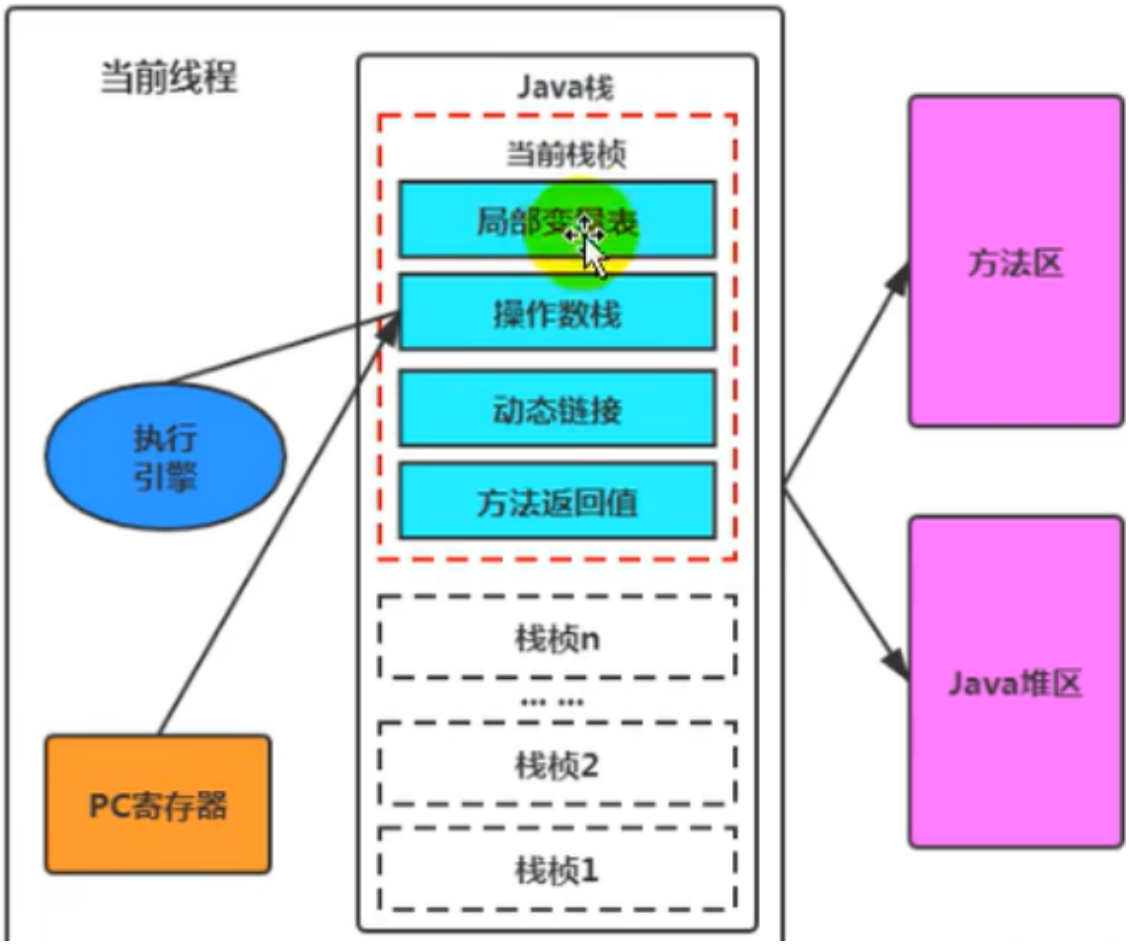


执行都是的当前的栈帧，谁在栈顶谁就是当前栈帧

- 本地方法栈

与虚拟机栈基本类似，区别在于虚拟机栈为虚拟机执行的java方法服务，而本地方法栈则是为Native方法服务。(栈的空间大小远远小于堆)

- 程序计数器（PC 寄存器）



是最小的一块内存区域，它的作用是当前线程所执行的字节码的行号指示器，在虚拟机的模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、异常处理、线程恢复等基础功能都需要依赖计数器完成。

存储下一个指令的地址，也就是下一个即将执行的代码。由执行引擎区

1.面向对象

OOP

> OOP，编写对象编程

OOP（object oriented programming），面向对象编程

- 把构成问题的各种事物，抽象成各个对象，这些对象具有解决问题的行为（方法），同时对象还可以具有解决很多类似问题的行为（方法），而不只是能解决一个问题。

  例如：小明开奥迪车去北京

  前面的一个同学帮他女朋友清空了购物

POP（procedure oriented Programming），面向过程编程

- 分析出，解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，然后依次调用就可以了。

  小明吃红烧肉

  买肉 --> 买配料(姜蒜葱桂皮八角冰糖) ---> 买厨具 --->洗厨具 --> 做菜 ---> 装盘 ---> 吃

FP（functional programming)，函数式编程

- 类似于面向过程的程序设计方式，具有很高的抽象程度。JDK8中的一些特性，可以支持使用函数式编程。

例如，完成做饭功能

面向过程：起锅-->倒油-->放葱姜蒜-->放菜-->翻炒3分钟-->放生抽-->放盐-->翻炒1分钟-->道-->出锅
起来
面向对象：打开全自动炒菜机-->放各种配料-->启动-->出锅

> 注意，关键是要把**炒菜机**这个类给定义好，然后创建这个类的对象，调用方法完成功能

面向对象思想，是一种程序设计思想，使用这种思想进行编程的语言，就是面向对象编程语言。

Java语言是众多面向对象编程语言中的其中一种，我们需要在面向对象思想的指引下，使用Java语言去设计、开发计算机程序。

面向对象中的对象，泛指现实中一切事物，每种事物都具备自己的属性和行为。面向对象思想就是在计算机程序设计过程中，参照现实中事物，将事物的属性特征、行为特征**抽象**出来，定义为程序中的一种数据类型。

面向对象的思想，主要强调的是通过调用对象的行为来实现功能，而不是自己一步一步的去操作实现。

思考，java中的类是怎么来的？对象又是怎么获得的？

java中的类直接或者间接继承了Object

> 封装（Encapsulation）、继承（inheritance）、多态（polymorphism），是面向对象编程中的基本特征，java语言也同样具有这三种特征。后面会对这些特征进行详细的了解和学习。

2 类

java中对数据类型的描述和定义，都是抽象的，每一种数据类型，都是对同一类数据的抽象描述，描述了这种数据的基本特点。

例如，基本数据类型中的`int`，就是对计算机中一个简单的32位数据的定义，描述了这种数据的基本特点和表达的含义。

例如，引用数据类型中的`String`，就是对程序中的字符串这种数据的定义，描述了作为一个字符串数据，应该具有哪些属性、行为和特点。

而`int`和`String`都不能直接当做具体的数据来使用或者参与运算，因为它们俩个都是对数据的抽象描述，并不能当做具体的数据使用，如果想使用的这些数据的话，可以使用`int`或者`String`类型进行变量的声明，再用变量接收数据，然后就可以使用这个变量来进行操作。

例如，

```
1  public class Student{
2      public String name;
3
4      public void sayHello(){
5
6      }
7  }
```

这里我们先定了一个`Student`类，这个`Student`类就是一个自定义的数据类型，它本身是对学生的抽象描述，规定了一个学生数据，应该具有一个`String`类型的name属性，以及有一个`sayHello`方法。

我们并不能直接使用`Student`类进行操作，因为`Student`类是对学生这种数据的抽象描述，但是我们可以使用`Student`类声明变量，并且使用这个变量接收一个学生数据（对象），然后再使用这个变量对具体的学生数据进行操作（访问属性、调用方法）。

例如，

```java
public static void main(String[] args){
    //使用Student类，声明变量
    Student stu;
    //使用这个变量，接收一个具体的学生数据（也就是对象）
    stu = new Student();

    //使用对象访问它的的属性和调用它的方法
    stu.name = "tom";
    stu.sayHello();

}
```

3 对象

> 放眼望去，我们身边的能看到的东西，都是对象，例如桌子，椅子，话筒，电脑，鼠标，水杯等等，面向对象的思想中，一切皆为对象。

思考，类和对象的关系是什么？

通过前面的学习可知，类是一组相关属性和行为的集合，它是对某一种具体事物的抽象描述。

也可以把类看作一个模板，我们使用的对象，就是按照这个模板中的定义，来进行创建的。

- 类是对一类事物的描述，是抽象的
- 对象是一类事物的实例，是具体的
- 类是对象的模板，对象是类的实体

例如，

对猫的定义：（这就相等于定义了一个类）

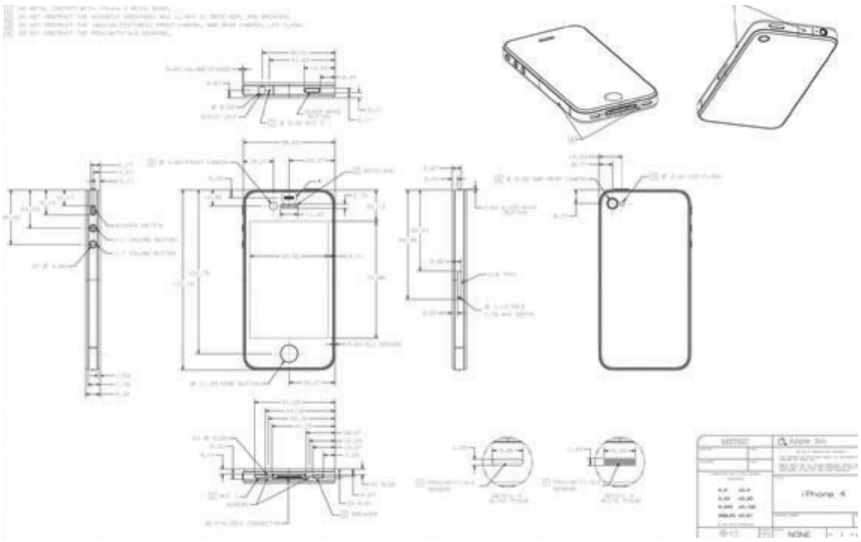
- 属性：名字、体重、年龄、颜色
- 行为：走、跑、叫

我家里的小猫皮皮：（这就是一个具体的对象）

- 名字：皮皮，体重：3KG，年龄：3岁，颜色：白色
- 皮皮可以走，也可以跑，也可以喵喵的叫

> 可以看出，皮皮这个小猫（对象），完全符合上面对猫的定义的一切属性和行为。

例如，**这是手机的图纸，定义了手机一些特征**（相当于定义了类）



这就是按着上面的图纸，制造出来的俩台具体的手机（相当于创建了手机类的俩个不同的对象）

> 如果有需要，可以根据手机类（图纸），创建出无数个具体的对象（手机）

> 一个类的对象，也就是这个类的具体实例。
>
> 所以说，类是一种抽象的数据描述，对象是类的一个具体的实例。

思考，有对象之后，可以使用对象做什么？

4 引用

> 引用类型的变量，简称引用。
>
> 引用是可以指向对象的，简称引用指向对象。

使用类创建对象之后，怎么给对象起一个名字，以便后面对这个对象进行操作

例如，

```
1
2  //使用new 加上 Student类中构造器，来创建Student类的对象
3  new Student();
4
5  //为了能方便的使用这个对象，就可以给这个对象起一个名字
6  //这个过程，其实就是之前学习过的 =号赋值操作
7  //把新创建的学生对象，赋值给了引用stu
8  Student stu = new Student();
9
10 //后面就可以使用引用stu来操作对象了，也就是访问对象中的属性和调用对象中的方法
11 stu.name = "tom";
12 stu.sayHello();
13
```

思考，如果没给对象起名字，这个对象能使用么？

能使用，但是只能在创建这个对象的同时就使用，并且只能使用一次，后面就不能使用了，因为它没有名字，无法在语法上面标示对这个对象进行使用

例如，

```
1  (new Student()).sayHello();
2
3  //下面就无法再使用上面创建出来的这个对象，因为他没有名字
4  ...
5
6  //如果再这样写一次，其实是创建了第二个对象，并进行使用，但是后面也无法使用这个对象，因为没有名字
7  (new Student()).sayHello();
```

引用、对象、类之间的关系：

例如，工厂根据电视机图纸（**类**），生产出了很多台电视机（**对象**），其中一台电视机卖给了张三，张三坐在沙发上，使用遥控器（**引用**），可以对这台具体的电视机（**对象**）进行很方便的操作。

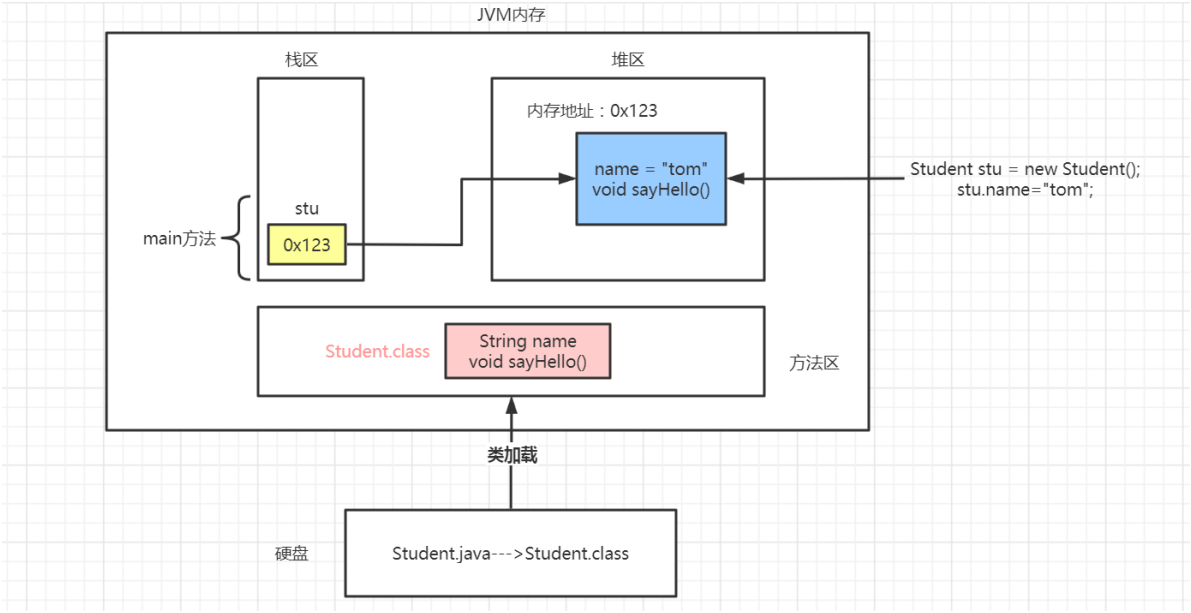
例如，工厂根据风筝图纸（**类**），生产出了很多只风筝（**对象**），其中一只风筝卖给了张三，张三站在草地上，使用线（**引用**），可以对这只具体的风筝（**对象**）进行很方便的操作。

5 内存

例如，用以下代码结合内存图进行说明

```
public class Student{
    public String name;
    public void sayHello(){

    }
}

public static void main(String[] args){

    Student stu = new Student();
    stu.name = "tom";
    stu.sayHello();

}
```

对应的内存图：



注意1，类加载过程，把Student.class文件内容加载到方法区中

注意2，main方法运行时，整个main方法的代码都被加载到栈区中

注意3，创建对象，给name属性赋值，这些操作的代码也在main方法中，但由于代码太长，图中标注到右边

注意4，根据类（相当于模板）创建出的对象，都是在堆区中

注意5，对象中的属性和方法，和类中定义的保持一致，因为对象就是根据类创建出来的

注意6，对象是具体的，我们可以给它的属性赋一个具体的值，例如tom

注意7，引用stu在栈去中，它保存了这个对象在堆区中的地址（0x123），形象的描述为，引用指向对象

注意8，=号赋值操作，其实就把对象的内存地址，赋值给了引用stu

注意9，如果有需要，可以根据类，继续在堆区中，创建出其他具体的学生对象

> 从这里可以看出，引用指向对象后，为什么就可以使用引用来操作对象，例如访问对象的属性和调用方法

6 方法

> 方法定义在类中，属于类的成员，所以也可以叫做成员方法。类似的，类中的属性，也可以称为成员变量

方法定义的格式：

```
修饰符 返回类型 方法名(参数列表)抛出异常的类型{

    //code

}
```

修饰符：

- public、static、abstract、final等这些都属于修饰符，可以用来修饰方法、属性、类
- 一个方法上，可以同时拥有多个不同的修饰符，例如程序入口main方法
  `public static void main(String[] args){...}`
  这里使用俩个修饰符public和static来修饰main方法
- 如果方法上有多个修饰符，这些修饰符是没顺序之分的
  例如，这俩个写法最终的效果是一样的
  `public static void main(String[] args){}`
  `static public void main(String[] args){}`

- 方法上也可以不写修饰符
  例如，

```
void hello(){
    //...
}
```

返回类型:

- 方法执行完，如果有要返回的数据，那么在方法上就一定要声明返回数据的类型是什么，如果没有要返回的数据，那么在方法上就必须使用void进行声明
  `public int getNum(){...}`
  `public void print(){...}`
- 只有一种特殊的方法没有返回类型，也不写void，那就是构造方法，也就是构造器

```
public class Student{
    //构造方法
    public Student(){}
}
```

- 声明有返回类型的方法，就要使用`return`关键字在方法中，把指定类型的数据返回

```
public int test(){
    return 1;
}
```

> 思考，如果一个方法的返回类型声明为void，那么在这个方法中还能不能使用return关键字?

方法名：

- 只要满足java中标识符的命名规则即可
- 推荐使用有意义的方法名

参数列表：

- 根据具体情况，可以定义为无参、1个参数、多个参数、可变参数。

```
public void test(){}

public void test(int a){}

public void test(int a,int b,int c){}

public void test(int... arr){}

public void test(int a,String str){}
```

抛出的异常类型：

- 在方法的参数列表后，可以使用`throws`关键字，表明该方法在j将来调用执行的过程中，【可能】会抛出什么类型的异常
- 可以声明多种类型的异常，因为在方法执行期间，可能会抛出的异常类型不止一种。

```
public void test()throws RuntimeException{}

public void test()throws
IOException,ClassNotFoundException{}
```

7 参数传递

java方法的参数，分为形参和实参。

形参：

- 形式上的参数

```
1  public void test(int a){}
```

其中，参数a就是test方法形式上的参数，它的作用就是**接收**外部传过来的实际参数的值

实参：

- 实际上的参数

```
1  public class Test{
2      public void test(int a){}
3  }
4
5  public static void main(String[] args){
6      Test t = new Test();
7      t.test(1);
8      int x = 10;
9      t.test(x);
10 }
11
```

其中，调用方法的时候，所传的参数1和x，都是test方法实际调用时候所传的参数，简称实参

值传递：

方法的参数是**基本类型**，调用方法并传参，这时候进行的是值传递。

例如，

```
public class Test{
    //该方法中，改变参数当前的值
    public static void changeNum(int a){
        a = 10;
    }

    public static void main(String[] args){
        int a = 1;
        System.out.println("before: a = "+a);//传参之前，变量a的值
        changeNum(a);
        System.out.println("after: a = "+a);//传参之后，变量a的值
    }

}
```

> 值传递，实参把自己存储的值（基本类型都是简单的数字）赋值给形参，之后形参如何操作，对形参一点影响没有。

引用传递：

方法的参数是**引用类型**，调用方法并传参，这时候进行的是引用传递。

这时候之所以称之为引用传递，是因为参数和形参都是引用类型变量，其中保存都是对象在堆区中的内存地址。

例如，

```
public class Test{
    //该方法中，改变引用s所指向对象的name属性值
    public static void changeName(Student s){
        s.name = "tom";
    }

    public static void main(String[] args){
        Student s = new Student();
        System.out.println("before: name =
"+s.name);//传参之前，引用s所指向对象的name属性值
        changeName(s);
        System.out.println("after: name =
"+s.name);//传参之后，引用s所指向对象的name属性值
    }

}
```

> 由于引用传递，是实参将自己存储的对象地址，赋值给了形参，这时候俩个引用（实参和形参）指向了同一个对象，那么任何一个引用（实参或形参）操作对象，例如属性赋值，那么另一个引用（形参或实参）都可以看到这个对象中属性的变量，因为俩个引用指向同一个对象。
>
> 这个时候就相当于，俩个遥控器，同时控制同一台电视机。

8 this

> 在类中的非静态方法中，可以使用this关键，来表示当前类将来的一个对象。

this关键字的使用场景：

1. 区别成员变量和局部变量
2. 调用类中的其他方法
3. 调用类中的其他构造器

区别成员变量和局部变量：

```
public class Student{
    public String name;
    public void setName(String name){
        //=号左边的this.name，表示类中的属性name
        //=号右边的name，表示当前方法的参数name（就近原则）
        this.name = name;
    }

}
```

调用类中的其他方法：

```
 public class Student{
     public String name;

     public void setName(String name){
         this.name = name;
     }
     public void print(){
         //表示调用当前类中的setName方法
         this.setName("tom");
     }
 }
```

> 默认情况下，setName("tom")和this.setName("tom")的效果是一样的，所以这里也可以省去 this.

调用类中的其他构造器：

```
public class Student{
    public String name;
    public Student(){
        //调用一个参数的构造器,参数的类型是String
        this("tom");
    }
    public Student(String name){
        this.name = name;
    }
}
```

> 注意，this的这种调用构造器的用法，只能在类中的一个构造器，调用另一个构造器。并且不能在普通的方法中调用类的构造器。
>
> 并且要求，this调用构造器的代码，是当前构造器中的第一句代码，否则编译报错。

this关键字的意义：

this代表，所在类的当前对象的引用（地址值），即对象自己的引用。

例如，

```java
public class Student{

    public void sayHello(){

    }

    public void show(){

    }

}

public static void main(String[] args){

    Student stu = new Student();
    stu.sayHello();

}

```

`Student`类中定义了俩个方法，sayHello和show，想要调用这俩个方法，就需要创建`Student`类的对象，然后使用对象调用这俩个方法。

这里就包含了一个固定的规则：**类中的非静态方法，一定要使用类的对象来进行调用，没有其他的方式！**

那么在这个情况下，思考一个问题：如果我们使用使用了stu调用了sayHello方法，例如`stu.sayHello()`，那么在sayHello中，怎么再调用到这个对象中的show方法？

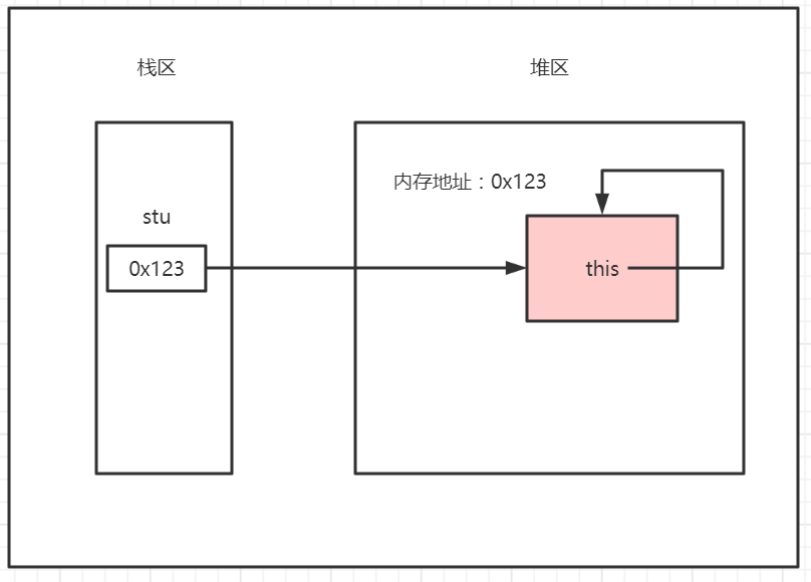
这个时候，如果在sayHello方法中，能有一个变量，可以表示stu这个对象本身，是不是就可以调用到了show方法？毕竟只有对象自己，才能调用到自己的方法，没有其他方式！

所以，这时候，就可以在sayHello方法中，使用this来表示stu对象本身，那么**this.show()**，就可以表示这个对象调用自己的show方法了

例如，

```
1  public class Student{
2
3      public void sayHello(){
4          this.show();//这个this就表示当前类的对象stu
5      }
6
7      public void show(){
8
9      }
10
11 }
12
13 public static void main(String[] args){
14
15     Student stu = new Student();
16     stu.sayHello();
17
18 }
```

对应的内存图：



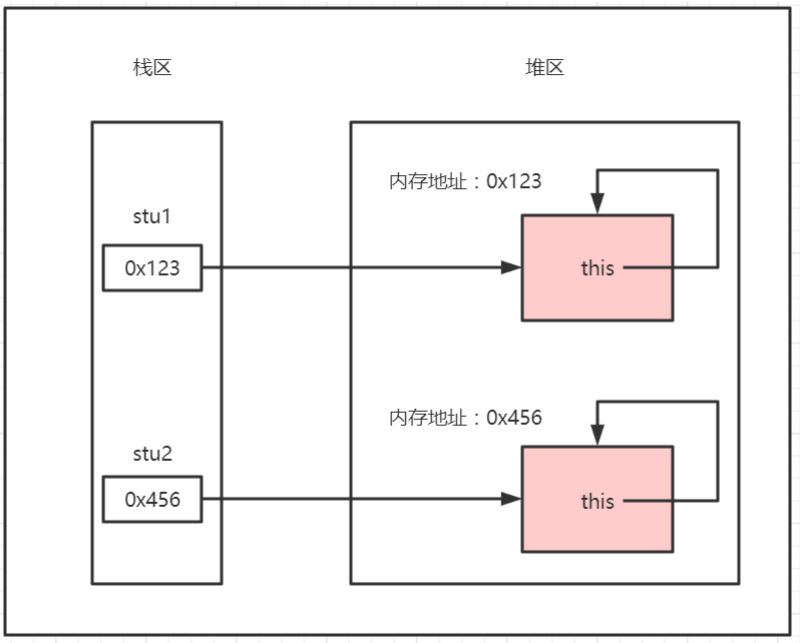
> 也就是说，stu里面存的是0x123，this里面存的也是0x123

思考：如果这时候创建俩个对象，那么this又代表哪一个对象呢？

例如，

```
1  public class Student{
2
3      public void sayHello(){
4          this.show();//问题：这个this代表的时候stu1还是
   stu2
5      }
6
7      public void show(){
8
9      }
10
11 }
12
13 public static void main(String[] args){
14
15     Student stu1 = new Student();
16     stu1.sayHello();
17
18     Student stu2 = new Student();
19     stu2.sayHello();
20
21 }
```

对应的内存图：



> 可以看出，其实每一个对象中，都有自己的this，和其他对象中的互不影响。
>
> 当前执行stu1.sayHello()代码的时候，this代表的就是stu1
>
> 当前执行stu2.sayHello()代码的时候，this代表的就是stu2

> 记住：方法被哪个对象调用，方法中的this就代表那个对象。即谁在调用，this就代表谁。

思考：在生活中，我们每一个人心中的this，指的是哪一个汉字？

思考：现在想想为什么，this.name表示访问自己属性name