主要内容：

* JVM是如何构建一个实例的(java中new一个对象的过程)
* .class文件
* 类加载器
* Class类
* 反射API

内存：即JVM内存，栈、堆、方法区啥的都是JVM内存，只是人为划分

.class文件：就是所谓的字节码文件，这里称.class文件，直观些

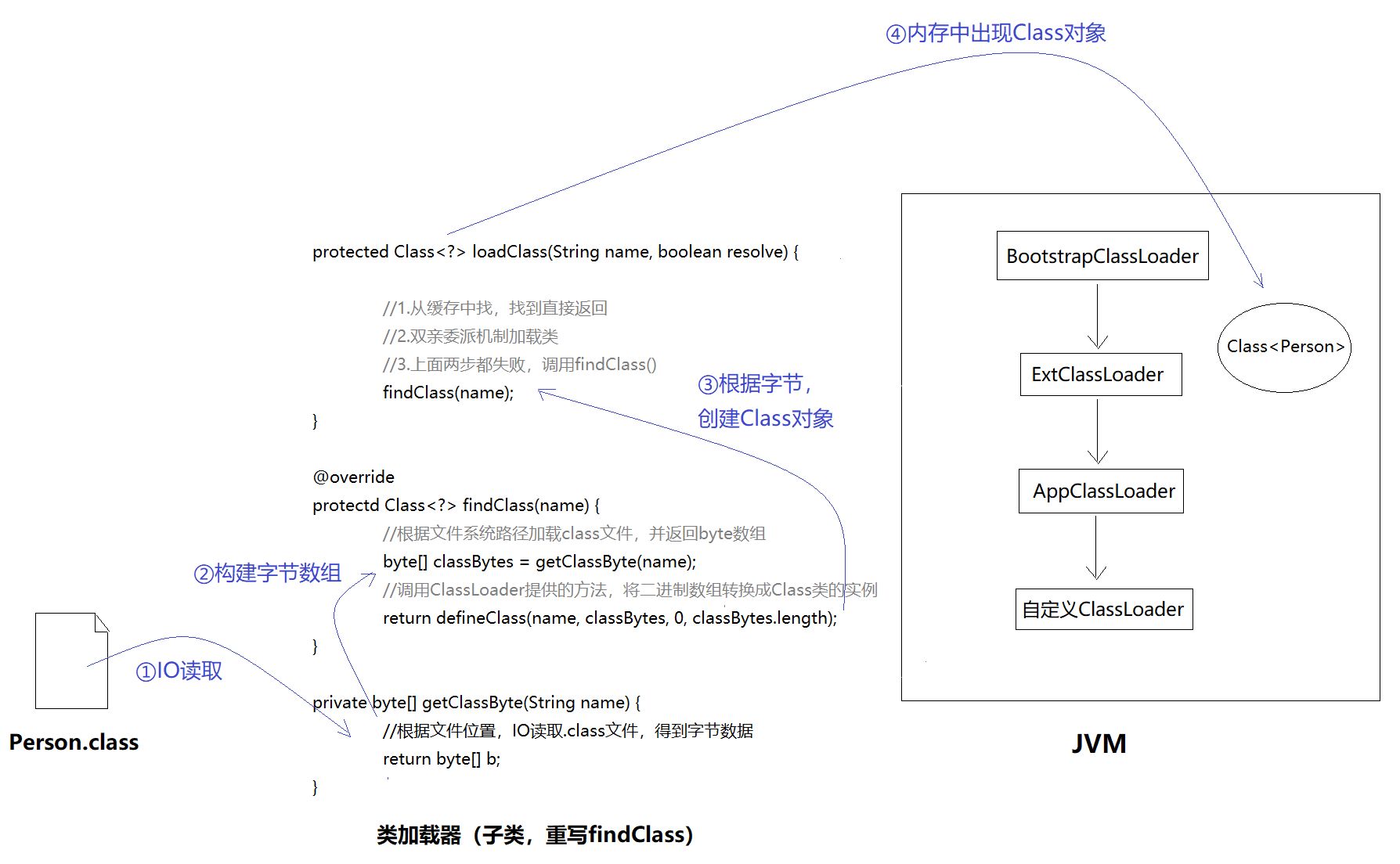
java在new一个对象的时候，会先查看对象所属的类有没有被加载到内存，如果没有的话，就会先通过类的全限定名来加载。加载并初始化类完成后，再进行对象的创建工作。

我们先假设是第一次使用该类，这样的话new一个对象就可以分为两个过程：加载并初始化类 和创建对象。

加载.class文件大致可以分为3个步骤：

1. 检查是否已经加载，有就直接返回，避免重复加载
2. 当前缓存中确实没有该类，那么遵循父优先加载机制，加载.class文件
3. 上面两步都失败了，调用findClass()方法加载

反正，目前我们关于类加载只需知道以下信息：



Java中new一个对象的过程：

java在new一个对象的时候，会先查看对象所属的类有没有被加载到内存，如果没有的话，就会先通过类的全限定名来加载。加载并初始化类完成后，再进行对象的创建工作。

我们先假设是第一次使用该类，这样的话new一个对象就可以分为两个过程：**加载并初始化类和创建对象。**

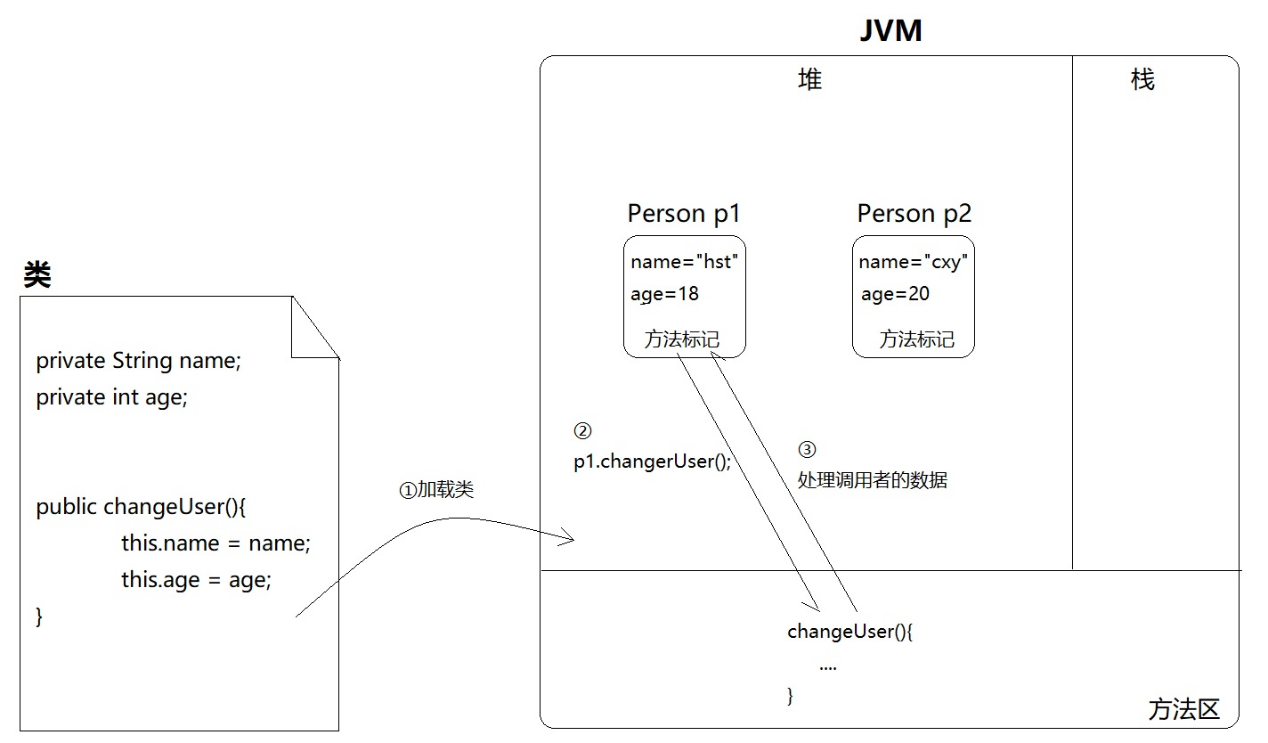
**类的加载过程：**

使用双亲委托机制的好处是：能够有效确保一个类的全局唯一性，当程序中出现多个限定名相同的类时，类加载器在执行加载时，始终只会加载其中的某一个类。（A a = new A(); 类的加载）

加载步骤：

**1、加载：**

由类加载器负责根据一个类的全限定名来读取此类的二进制字节流到JVM内部，并存储在运行时内存区的方法区，然后将其转换为一个与目标类型对应的java.lang.Class对象实例。



**类的编译与类的加载的区别：**

编译：是将XXX.java文件编译成XXX.class字节码文件

编译有3步：

1. 词法分析和输入到符号表
2. 注解处理
3. 语义分析和生成字节码文件

**加载：**是创建对象，例如 A a = new A();将A.class字节码文件加载进内存中

**2、验证**

格式验证：验证是否符合class文件规范

语义验证：检查一个被标记为final的类型是否包含子类；检查一个类中的final方法是否被子类进行重写；确保父类和子类之间没有不兼容的一些方法声明（比如方法签名相同，但方法的返回值不同）

操作验证：在操作数栈中的数据必须进行正确的操作，对常量池中的各种符号引用执行验证（通常在解析阶段执行，检查是否可以通过符号引用中描述的全限定名定位到指定类型上，以及类成员信息的访问修饰符是否允许访问等）

**3、准备**

为类中的所有的静态变量分配内存空间，并为其设置一个初始值，由于还未产生对象，所以实例变量不在此操作范围。

被final修饰的static变量（常量）会直接赋值。

**4、解析(暂时不理解，这具体是做什么的？)**

将常量池中的符号引用转为直接引用（得到类或者字段、方法在内存中的指针或者偏移量，以便直接调用该方法），这个可以在初始化之后再执行。

解析需要静态绑定的内容。 // 所有不会被重写的方法和域都会被静态绑定

以上2、3、4三个阶段又合称为链接阶段，链接阶段要做的是将加载到JVM中的二进制字节流的类数据信息合并到JVM的运行时状态中。

**5、初始化(先父后子)**

1、 为静态变量赋值

2、 执行static代码块

注意：static代码块只有jvm能够调用

如果是多线程需要同时初始化一个类，仅仅只能允许其中一个线程对其执行初始化操作，其余线程必须等待，只有在活动线程执行完对类的初始化操作之后，才会通知正在等待的其他线程。

因为子类存在对父类的依赖，所以类的加载顺序是先加载父类后加载子类，初始化也一样。不过，父类初始化时，子类静态变量的值也有有的，是默认值。

最终，方法区会存储当前类类信息，包括类的静态变量、类初始化代码（定义静态变量时的赋值语句 和 静态初始化代码块）、实例变量定义、实例初始化代码（定义实例变量时的赋值语句实例代码块和构造方法）和实例方法，还有父类的类信息引用。

**创建对象：**

1、在堆区分配对象需要的内存

分配的内存包括本类和父类的所有实例变量，但不包括任何静态变量。

2、对所有实例变量赋默认值

将方法区内对实例变量的定义拷贝一份到堆区，然后赋默认值。

3、执行实例初始化代码

初始化顺序是先初始化父类再初始化子类，初始化时先执行实例代码块然后是构造方法。

4、如果有类似于Child c = new Child()形式的c引用的话，在栈区定义Child类型引用变量c，然后将堆区对象的地址赋值给它

需要注意的是：每个子类对象持有父类对象的引用，可在内部通过super关键字来调用父类对象，但在外部不可访问

补充：

通过实例引用调用实例方法的时候，先从方法区中对象的实际类型信息找，找不到的话再去父类类型信息中找。

如果继承的层次比较深，要调用的方法位于比较上层的父类，则调用的效率是比较低的，因为每次调用都要经过很多次查找。这时候大多系统会采用一种称为虚方法表的方法来优化调用的效率。

所谓虚方法表，就是在类加载的时候，为每个类创建一个表，这个表包括该类的对象所有动态绑定的方法及其地址，包括父类的方法，但一个方法只有一条记录，子类重写了父类方法后只会保留子类的。当通过对象动态绑定方法的时候，只需要查找这个表就可以了，而不需要挨个查找每个父类。

反射API:

反射的基本使用：

User user = new User();  
 *//方式一* Class<? extends User> mClass01 = user.getClass();  
 *//方式二* Class<User> mClass02 = User.class;  
 *//方式三* Class<?> mClass03 = Class.forName("com.ztk.StaticAndThis.User");  
 try {  
*//方式一创建对象* User user1 = mClass01.newInstance();  
 *//获取所有方法，会将其父类的方法一并获取* Method[] methods1 = mClass01.getMethods();  
 *//获取该类的所有的方法* Method[] methods2 = mClass01.getDeclaredMethods();  
 *//获取指定的方法 eat, String.class:是该方法的参数的类型的字节码对象， 如果没有参数，则传null* Method methods3 = mClass01.getMethod("eat", String.class);  
 *//执行该eat方法，但是需要传参数， user1：表明是哪个对象， 上面已经newInstance()出来了  
 // aa: 是方法的传参* Object invoke1 = methods3.invoke(user1, "aa");  
 System.***out***.println(invoke1);  
 } catch (InstantiationException e) {  
 e.printStackTrace();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }