没有分配内存的都是生成对象（抽象类），分配内存的都是实例化对象。一个Java对象的创建过程往往包括 类初始化 和 类实例化 两个阶段。

Java对象创建时机

一个对象在可以被使用之前必须要被正确地实例化。

1). 使用new关键字创建对象

调用一个类的构造函数显式地创建对象，这种方式在Java规范中被称为 ： 由执行类实例创建表达式而引起的对象创建。除了使用new关键字创建对象的方式外，其他方式全部都是通过转变为invokevirtual指令直接创建对象的。

2). 使用Class类的newInstance方法(反射机制)

事实上，newInstance方法通过调用无参的构造器创建对象，内部调用的是Constructor类的newInstance方法。eg：

```
Student student2 = (Student)Class.forName("Student类全限定名").newInstance();
或者  Student stu = Student.class.newInstance();
```

3). 使用Constructor类的newInstance方法(反射机制)

Constructor类的newInstance方法比Class类中的newInstance方法更强大，可通过这个newInstance方法调用有参数的和私有的构造函数。eg.

```
Constructor<Student> constructor = Student.class.getConstructor(Integer.class);
Student stu3 = constructor.newInstance(123);
```

4). 使用Clone方法创建对象

无论何时我们调用一个对象的clone方法（Java String类.note），JVM都会帮我们创建一个新的、一样的对象，特别的是，用clone方法创建对象的过程中并不会调用任何构造函数。

5). 使用(反)序列化机制创建对象

当我们反序列化一个对象时（实现Serializable接口），JVM会给我们创建一个单独的对象，在此过程中，JVM并不会调用任何构造函数。eg.

```
public class Student implements Cloneable, Serializable {

    private int id;

    public Student(Integer id) {
        this.id = id;
    }

    @Override
    public String toString() {
        return "Student [id=" + id + "]";
    }

    public static void main(String[] args) throws Exception {

        Constructor<Student> constructor = Student.class
                .getConstructor(Integer.class);
        Student stu3 = constructor.newInstance(123);

        // 写对象
        ObjectOutputStream output = new ObjectOutputStream(
                new FileOutputStream("student.bin"));
        output.writeObject(stu3);
        output.close();
```

```
        // 读对象
        ObjectInputStream input = new ObjectInputStream(new
FileInputStream(
                "student.bin"));
        Student stu5 = (Student) input.readObject();
        System.out.println(stu5);
    }
}
```

Java 对象的创建过程

实例化一个类的对象的过程是一个典型的递归过程。在准备实例化一个类的对象前，首先准备实例化该类的父类，一直递归到object类。实例化每个类时，都遵循如下顺序：先依次执行实例变量初始化和实例代码块初始化，再执行构造函数初始化。也就是说，编译器会将实例变量初始化和实例代码块初始化相关代码放到类的构造函数中去，并且这些代码会被放在对超类构造函数的调用语句之后，构造函数本身的代码之前。

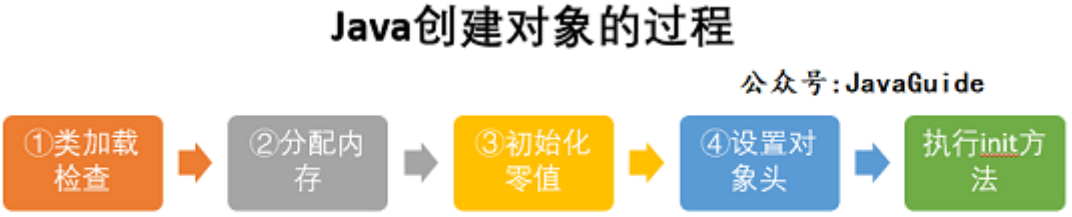
1、实例变量初始化与实例代码块初始化

定义的同时也可进行赋值，如**直接对实例变量或通过实例代码块进行赋值，**编译器会将其代码放进构造函数中。Java强制要求Object对象(Object是Java的顶层对象，没有超类)之外的所有对象构造函数的第一条语句必须是超类构造函数的调用语句或者是类中定义的其他的构造函数。直到递归到顶层父类Object 。ps.**不允许顺序靠前的实例代码块初始化在其后面定义的实例变量**

2、构造函数初始化

在编译生成的字节码中，这些构造函数会被命名成<init>()方法，参数列表与Java语言书写的构造函数的参数列表相同。如果我们显式调

用超类的构造函数，那么该调用必须放在构造函数所有代码的最前面。如果我们在一个构造函数中调用另外一个构造函数，则只允许第二个构造函数调用超类中的构造函数。



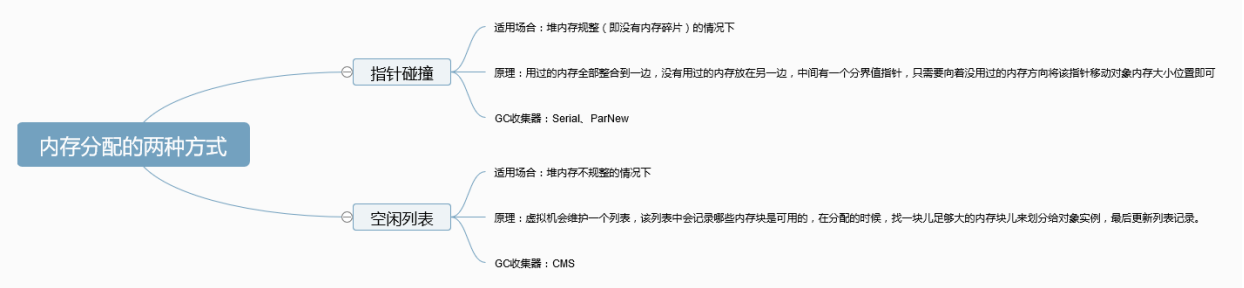
Step1:类加载检查

虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。（xf.好像并没有严格的执行类加载过程）

Step2:分配内存

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有 “指针碰撞” 和 “空闲列表” 两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

看GC 收集器的算法是"标记-清除"，还是"标记-整理"（也称作"标记-压缩"），值得注意的是，复制算法内存也是规整的



内存分配并发问题

在创建对象的时候有一个很重要的问题，就是保证线程安全：

- **CAS+失败重试：CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。**
- **TLAB： 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配**

Step3:初始化零值

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

Step4:设置对象头

初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。 这些信息存放在对象头中。 另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

Step5:执行 init 方法

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚开始，<init> 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 <init> 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。