java□译流程

- □ `javac` 把 `.java` 文件编译成 `.class` 文件
- 再□JVM把 `.class` 文件编译成二进制文件给操作系统执行
- 因此它十分适合跨操作系统开发

JVM

就是**java**虚拟机

- 即时□译（**JIT**）：会监控代码执行频率，找出热点代码，把热点代码编译成机器码后存储在内存
- 内存管理：利用JVM的垃圾回收机制自动回收不用的对象

组成部分：

- 类加载器：加载字节码文件到内存
- 运行时数据区域：存储类和接口
- 执行引擎：将字节码转换为机器码
- 本地接口：调用本地已经□译的用c/cpp编写的方法

字节码文件

以二进制方式存储

image-20250827103136127

组成部分：

- 基础信息：字节码文件对应的JDK版本号，访问标识，类、父类和接口（指针/地址）
- 常量池：保存了各种字面量和符号引用，如字符串常量，类或接口名、字段名（实际的字符串名称）
- 字段：当前类或接口声明的字段信息
  - 如果是 `static final` 修饰，字节码文件中会有 `ConstantValue` 属性
- 方法：当前类或接口声明的方法信息
  - 方法如果没有 `Code` 属性，其就是抽象方法
- 属性：存储类的属性，比如源码文件名，内部类列表
  - 为类、字段或方法提供额外的元数据（描述信息），存储在类、字段或方法信息当中

```java
// □量池存储代码中所有的字面名称

// 基础信息：存Person、Person的父类以及Person实现的接口的索引
public class Person implements java.io.Serializable{
    // 字段信息
    private String name;

    private static int count;

    // 字段信息里面也会存储属性
    private static final int id = 0;

    // 方法信息
    // 属性：每个方法都包含一个Code属性
    public Person(String name) {
```

```
        this.name = name;
        count++;
    }

    public String getName() {
        return this.name;
    }

    public static int getCount() {
        return count;
    }

}
/**
*属性里面会存储一个类级别的属性表
*整个类会有一个SourceFile属性
**/
```

基础信息

- 魔数：字节码文件中，将文件头称作魔数
    - 软件是使用文件的头几个字节去校验文件类型的
    - Java字节码的头几个字节是 `CAFFBABE`
- 主副版本号：判断当前字节码的版本和运行时的JDK是否兼容
    - 计算方式：$大版本号=主版本号-44$

常量池

作用：避免相同的内容重复定义，节省空间

```
public class Test{
    private static final String name1 = "我爱你";
    private static final String name2 = "我爱你";
}
```

存储方式：

前提是使用了 `static final` 声明，否则不会有常量值索引

- 每个字段都会有一个常量值索引，常量值索引指向常量池的某一个□号（字符串）
- 常量池的编号（字符串）并不直接存储字符串字面量，它会指向常量池中的另一个□号，里面存储真正的字符串字面量
- 如果字段名和字面量相同，那么字段名存储的索引会直接指向真正的字符串字面量，节省内存
- 注意：常量池的字符串会直接加载到字符串常量池中，而字符串字面量也需要进行存储，因为字段名可能用到

方法

临时的数据结构，方法开始时建立，结束时销毁

- 操作数栈：临时存放一些数据
- 局部变量表：方法声明的局部变量，底层是数组实现的
    - `main` 方法的 `args` 会占据数组的一个下标空间

源代码

```java
int i = 0;
int j = i + 1;
```

字节码指令

```
iconst_0
istore_1
iload_1
iconst_1
iadd
istore_2
return
```

- `iconst_0`：将整数 `0` 压入操作数栈
- `istore_1`：从操作数栈弹出数据并放入局部变量表中的 `1` 号位置
- `iload_1`：从局部变量表中的 `1` 号数据复制一份放入操作数栈
- `iadd`：把操作数栈中栈顶部的两个元素进行相加，然后存放到栈中
- `iinc 1 by 1`：对局部变量表中的1号位置的数据加1

看一个例子

```java
int i = 0;
i = i++;

int j = 0;
j = ++j;
```

分析：

1. 第一个例子：先store到局部变量表，再**load**到操作数栈，然后自增，再从操作数栈加载到局部变量表（直接覆盖）

   ```
   iconst_0
   istore_1
   iload_1
   iinc 1 by 1
   istore_1
   return
   ```

2. 第二个例子：先store到局部变量表，再自增，然后**load**到操作数栈，再从操作数栈加载到局部变量表（先自增再加载到操作数栈，因此 `j` 是1）

   ```
   iconst_0
   istore_1
   iinc 1 by 1
   iload_1
   istore_1
   return
   ```

类的生命周期

五个阶段:

- 加载
- 连接：可以细分为验证、准备、解析
- 初始化
- 使用
- 卸载

加载

1. 类加载器根据类的全限定名通过不同的渠道以二进制流的方式获取字节码信息
2. 将字节码信息保存到方法区中，会生成一个 `InstanceKlass` 对象，保存类的所有信息（之前说的字节码文件里面的信息）

image-20250827121529235

3. 会在堆中生成一份与方法区中数据类似的 `java.lang.Class` 对象（字节码被加载到 **JVM** 后解析成的运行时结构）

- 静态数据是存放在堆中的
- 有助于我们利用反射去获取类的信息，反射获取到的 `Class` 是堆中的数据
- 同时，堆中的数据和方法区中的数据会建立一个关联
- 开发者new对象的时候是操作堆中的用 `java.lang.Class` 包装的类
    - 方法区中的对象是c++编写的，开发者无法直接操作
    - 方法区中的数据开发者不一定都要用到（虚方法表等）

连接

- 验证：验证内容是否满足Java虚拟机规范
    - 校验文件格式以及主次版本号
    - 元信息验证，例如类必须有父类
    - 验证字节码指令的正确性
    - 符号引用验证，例如是否访问了其它类的私有属性
- 准备：给静态变量赋值（默认值）
    - `int` : 0
    - `double` : 0.0
    - 引用数据类型：null
    - 如果使用 `final` 修饰了，在准备阶段会直接赋值（不是默认值）
- 解析：将常量池中的符号引用替换成实际内存地址的直接引用

初始化

- 执行静态代码块中的代码，为静态变量赋值
- 类的初始化会执行字节码文件中的 `clinit` 部分的字节码指令
    - 执行顺序和代码里面的顺序有关

image-20250827165053890

源代码

```
public class Demo1{
    public static int value = 1;
    static {
```

```
        value = 2;
    }
    public static void main(String []args){

    }
}
```

字节码指令

```
iconst_1
putstatic #2 <init/Demo1.value : I>
iconst_2
putstatic #2 <init/Demo1.value : I>
return
```

- `putstatic #2 <init/Demo1.value : I>`：从操作数栈弹出一个值赋值给常量池编号为 `2` 的变量，也就是 `Demo1` 里面的 `value` 变量

以下几种方式会导致类的初始化：

- 访问一个类的静态变量或者静态方法
  - 变量是 `final` 修饰的且等号右边是常量不会触发初始化！！！
  - 声明了静态变量但是没赋值不会触发初始化！！！
- new一个该类的对象时
  - 实例变量和实例代码块先执行
  - 再执行构造方法
- 执行main方法当前的类
- 调用 `Class.forName()` 时
  - 可以指定参数不让类初始化

继承情况下的类的初始化：

- 直接访问父类的静态变量，不会触发子类初始化
- 子类的初始化 `clinit` 调用之前，会先调用父类的 `clinit` 初始化方法

类加载器

任务：负责获取二进制的字节码信息，在方法区和堆上创建对象是调取虚拟机的接口实现的

类加载器的分类

JDK8之前

分类：

- 一类是Java实现的，继承自抽象类 `ClassLoader`
  - `Extension CLassLoader`：扩展类加载器，加载通用的类
  - `Application ClassLoader`：应用程序类加载器，加载自己编写的或者第三方 `jar` 包的
- 一类是虚拟机底层用cpp实现的，加载程序运行时的基础类
  - `Bootstrap ClassLoader`：启动类加载器，加载核心类

从虚拟机角度来看，只存在两种类加载器：

- `Bootstrap` 启动类加载器
- 其它类加载器，全部继承自 `java.lang.ClassLoader`

`Bootstrap ClassLoader`

- 默认加载 `/jre/lib` 目录下的类文件，后缀都是 `.jar`
- 不允许我们在代码中获取启动类加载器
- 可以利用虚拟机参数让我们的类被启动类加载器加载

`Extension CLassLoader`

- 源码位于 `sun.misc.Launcher` 中，静态内部类，继承自 `URLClassLoader` ，多级继承
- 默认加载 `/jre/lib/ext` 目录下的类文件

`Application ClassLoader`

- 源码位于 `sun.misc.Launcher` 中，静态内部类，继承自 `URLClassLoader` ，多级继承
- 默认加载 `classpath` 下的文件，也就是自己写的和第三方依赖的类文件

image-20250828104007723

JDK8之后

- JDK9之后引入了module的概念，类加载器在设计上发生了很多变化

  1. 启动类加载器 `BootClassLoader` 使用 `java` 编写，位于 `jdk.internal.loader.ClassLoader` 中
  2. `BootClassLoader` 继承自 `BuiltinClassLoader` 实现从模块中找到要加载的字节码文件

  image-20250828145345501

  3. 扩展类加载器被替换成平台类加载器(`Platform Class Loader`)，平台类加载器遵循模块化方式加载字节码文件，因此也继承了 `BuiltinClassLoader`
  4. 应用程序类加载器也继承 `BuiltinClassLoader`

双亲委派机制

核心：解决一个类到底由谁加载的问题

作用：保证类加载安全性，避免重复加载

工作流程：

- 一个类加载器接受到加载类的人物的时候会自底向上查找是否加载过
- 加载过的的话，直接加载类
- 如果三个类加载器都没加载过，向下尝试加载
- 类加载器会查看要加载的类是否在自己负责的路径下，如果是，则直接加载，反之则向下委派

image-20250828114310579

他们之间是上下级关系而不是继承关系

打破双亲委派机制

自定义类加载器

注意：相同的类加载器加载相同的类限定名才会冲突，才会被认为是同一个类

举个例子：Tomcat服务器

- 里面可以运行多个web应用，如果出现了相同限定名的类，Tomcat要保证这两个类都能被加载
- 因此，要为每个应用创建一个隔离的类加载器

image-20250828115521211

自定义类加载器

`ClassLoader` 有四个核心方法：

- `loadClass`：类加载的入口，提供了双亲委派机制，内部调用 `findClass`
  - 有个 `resolve` 参数，是用来决定是否执行连接
- `findClass`：获取二进制数据之后调用 `defineClass`
- `defineClass`：做类名的校验，调用虚拟机底层的方法将字节码信息加载到方法区和堆
- `resolveClass`：执行类生命周期的连接阶段

image-20250828120952341

案例：自定义类加载器

- 会对以 `java.` 开头的类会有保护机制，虚拟机认为这种类只能□启动类加载
- 自定义类加载器默认的父加载器是 `ApplacationClassLoader`，要修改父加载器的话，重写 **`ClassLoader()`** 构造方法即可

```
// 继承ClassLoader
public class BreakClassLoader extends ClassLoader{
    private String basePath;
    private static final FINAL_TEXT = ".class";

    public void setBasePath(String basePath){this.basePath = basePath;}

    private byte[] loadClassData(String name){...}

    // 打破双亲委派机制
    @Override
    protected Class<?> loadClass(String name) throws ClassNotFoundException{
        if(name.stratwith("java.")){
            return super.loadClass(name);
        }
        // 获取二进制流
        byte[] data = loadClassData(name);
        return defineClass(name,data,0,data.length);
    }

    public static void main(String []args){
        BreakClassLoader classLoader = new BreakClassLoader();
        classLoader.setBasePath("D:\\lib\\");
         Class<?> clazz = classLoader.loadClass("com.yourcompany.YourClass");
    Object instance = clazz.newInstance(); // 创建实例
    }
}
```

线程上下文类加载器

SPI机制

- JDK内置的一种服务提供发现机制
- 工作原理：（JDBC案例）
    - 驱动需要暴露给JDBC的 `DriverManager` （管理驱动的类）使用，□它来引入不同的数据库驱动， `DriverManager` 由启动类加载器加载
    - 驱动需要存在固定文件夹下 `META-INF/services` ，以接口的全限定名命名文件名，对应的文件里面应该写该接口的实现类
    - 使用 `ServiceLoader` 加载实现类
        - 里面的 `.load()` 方法需要传递接口的字节码文件，然后这个方法返回一个 `ServiceLoader` 实例
        - 用这个实例去扫描 `META-INF/services` 目录，得到类的全限定名列表
        - 根据这个类的类名，进行类的加载，并且创建对象返回给用户
- `.load()` 方法使用了线程上下文中保存的类加载器进行类的加载，这个类加载器一般是应用程序类加载器
- 线程的类加载器默认都是应用程序类加载器

image-20250828142008540

JDBC案例网上说法不一，有的认为打破了双亲委派机制，有的认为没有打破

OSGI框架类加载器

- 早期java没有模块化的思想，代码都放在 `rt.jar` 下进行管理
- 后来创建了OSGI框架进行模块化管理，它运训同级的类加载器进行委托加载，并且实现了热部署的功能

运行时数据区域

把整个区域划分成两大类：

- 线程不共享：程序计数器， `Java` 虚拟机栈，本地方法栈
- 线程共享：方法区，堆（有线程安全问题）

image-20250829110329036

程序计数器

程序计数器也叫做**PC**寄存器，存储当前要执行的字节码指令的地址

- 不会发生内存溢出
- 程序员无需对程序计数器做任何处理

作用：

- 控制程序指令的执行
    - 加载阶段，虚拟机将字节码文件中的指令读取到内存后，会把源文件的偏移量替换为内存地址
    - 代码执行过程中，程序计数器会记录下一行字节码指令的地址，执行完当前指令后，会根据程序计数器执行下一行指令

image-20250828152526384

- 多线程执行情况下，Java虚拟机可以通过程序计数器记录CPU切换前字节码文件执行到哪一行指令

栈

分成两部分：

- **Java**虚拟机栈：保存在Java中实现的方法
- 本地方法栈：保存在cpp实现的方法吗，用native声明的

Java虚拟机栈

- 采用栈的数据结构来管理方法调用中的基本数据
- 每一个方法的调用使用一个栈帧来保存方法的基本信息

image-20250828155651944

- 随着线程的创建而创建，回收则会在线程的销毁时执行

栈帧的组成部分：

局部变量表：方法执行过程中存放的所有的局部变量

- 局部变量表中的槽是可以复用的
  - 保存的内容：实例方法的**this**对象，方法的参数，方法体中声明的局部变量
    - 字节码文件中的局部变量表
      - 编号：根据声明顺序确定
      - 起始PC：从哪一行字节码指令开始可以访问这个局部变量
      - 长度：局部变量的生效范围
      - 序号：槽的起始□号
    - 栈帧中的局部变量表
      - 是一个数组，每一个位置称之为槽
      - long和double类型占用两个槽，其他类型占用一个槽
  - 实例方法中的序号为0的位置存放的是this，指的是当前调用方法的对象，运行时会在内存中存放实例对象的地址

image-20250828160443690

操作数栈：存放临时数据的一块区域，□译的时候可以确定其最大深度

帧数据：动态链接、方法出口、异常表

- 动态链接：将符号引用转变为直接引用，就是把字节码指令中的符号引用（指向常量池）转变为直接指向内存地址的直接引用（指向运行时常量池）

image-20250828163247278

- 方法出口：方法结束时，栈帧会被弹出，程序计数器应该指向上一个栈帧中下一条指令的地址，在当前栈帧中，要存放上一个栈帧下一条指令的地址
- 异常表：存放代码中异常的处理信息
  - 起始PC：异常捕获生效的起始字节码指令的行数
  - 结束PC：异常捕获生效的结束字节码指令的行数
  - 跳转PC：出现异常之后要跳转到的字节码指令的行数

栈的内存溢出

- 栈帧太多，占用内存过大，超过栈内存，会导致内存溢出，出现 `StackOverflowError` 错误

- 不指定栈的大小，JVM将创建一个具有默认大小的栈
- 可以使用 `-Xss` 设置栈的内存
    - 必须是1024的倍数

本地方法栈

处理方式和**Java**虚拟机栈类似

堆

- 堆内存是空间最大的一块区域，创建出来且能够被开发者使用的对象都存在于堆上
- 栈上的局部变量表可以存放堆上对象的引用
- 静态变量也可以存放堆上对象的引用，从而通过静态变量实现对象在线程间的共享
- 堆内存有内存溢出的风险

image-20250828181256837

- 堆空间有三个需要关注的值，`used` 、`total` 、`max`
- `used` 指的是当前已使用的堆内存，`total` 是虚拟机已经分配的可用堆内存，`max` 是虚拟机可分配的最大堆内存
    - `max` 默认是系统内存的四分之一，`total` 默认是系统内存的六十四分之一
    - 服务端开发的时候，把 `max` 和 `total` 设置成相同的值，后续无需向虚拟机再次申请，减少申请并分配内存的时间开销

image-20250828182043377

方法区

- JDK7以及之前的版本将方法区存放在堆区域中的永久代空间
    - 存类的元信息、运行时常量池、字符串常量池、类的静态变量
- JDK8及之后的版本将方法区存放在直接内存中的元空间中，元空间位于操作系统维护的直接内存中
    - 元空间存类的元数据、运行时常量池
    - 字符串常量池和类的静态变量被转移至堆空间，类的静态变量是存在堆空间的**Class**对象中的
- 方法区也存在溢出问题

image-20250828190129331

存放基础信息的位置，线程共享，包含三部分内容：

类的元信息：保存了所有类的基本信息

- 存的是 `InstanceKlass` 对象，将字节码文件的所有信息都存在这个对象当中，还会存入虚方法表（实现多态）
- 常量池和方法会新开辟一块内存存储，`InstanceKlass` 对象只是存储了他们的引用
- 在类的加载阶段完成

image-20250828184709777

运行时常量池：保存了字节码文件中的常量池内容

- 字节码文件中通过编号查表的方式找到常量，这种常量池称为静态常量池

- 当常量池加载到内存中之后，每一个常量的数据都可以通过地址去访问，叫做运行时常量池

字符串常量池：保存了字符串常量

- JDK7之前是在方法区，之后被转移至堆内存
- 字符串常量池存的是在代码中定义的常量字符串，相同的字符串只会存一份
- new出来的字符串会放在堆中，局部变量表会存放这个对象的引用

不同**jdk**版本的区别

`.intern()`：将字符串对象添加到字符串常量池中

jdk6及之前

- jdk6的 `.intern()` 是把第一次遇到的字符串实例复制到永久代的字符串常量池中
- jdk6及之前，字符串常量池存储的是字符串实例
- 正常赋值，字符串实例会存储在字符串常量池；new的话，会在堆中开辟空间存储

```
String s1 = new StringBuilder().append("think").append("123").toString(); // 在堆中创建对象存储"think123"
System.out.println(s1.intern() == s1); // jdk6之前，调用.intern()方法是把字符串实例存到字符串□量池，因此此处返回false
```

jdk7及之后

- jdk7及之后，字符串常量池在堆上，`.intern()` 会把第一次遇到的字符串引用（引用指向堆中的空间）放入到字符串常量池，无论是new还是正常赋值
- jdk7及之后，无论是new还是正常赋值，字符串常量池存储的都是堆中对象的引用

```
String s1 = new StringBuilder().append("think").append("123").toString(); // 在堆中创建对象存储"think123"
System.out.println(s1.intern() == s1); // jdk7之后，调用.intern()方法是把堆中对象的引用存到字符串□量池，因此此处返回true
```

直接内存

jdk4之后引入了**NIO**机制，使用了直接内存，主要解决以下两个问题

1. Java堆中的对象不再使用会回收，但会影响到对象的创建和使用
2. 可以提升IO操作的效率：直接放入直接内存即可，同时在堆上维护直接内存的引用

执行引擎

自动垃圾回收

- Garbage Collection，简称GC机制。通过垃圾回收器来对不再使用的对象完成自动的回收
- 主要负责将堆上的内存进行回收，**C#**，**python**，**Go**都有自己的垃圾回收器

方法区的回收

判定类被卸载，需要满足下面三个条件：

1. 此类的所有实例对象已经被回收，堆中不存在任何该类的实例对象以及子类对象
2. 加载该类的类加载器已经被回收
3. 该类对应的 `java.lang.Class` 对象没有在任何地方被引用

`System.gc()`

- 可以手动触发垃圾回收
- 注意：执行后并不一定立即回收，只是向JVM发送一个请求，是否要执行我们没法干预

常用于热部署的应用场景中

- 每个jsp文件对应唯一的类加载器，当一个jsp文件被修改了，立刻卸载这个jsp文件的类加载器，然后重新创建类加载器，重新加载jsp文件

堆的回收

- Java中的对象能否被回收，是根据对象是否被引用来决定的，如果对象被引用，则不允许被回收
- 如果是堆里面的对象循环引用，而栈里面没有变量存储这个对象引用，那么堆里面的对象也可以被回收

引用计数法

- 为每个对象维护一个引用计数器，对象被引用时加1，取消引用时减1
- 会存在循环引用问题，当A引用B且B引用A的时候会出现对象无法被回收的问题

可达性分析算法

如果对象无法被**GC root**链查找到，则可以对对象进行回收

我们称这种引用为强引用

image-20250829130545212

将对象分为两类：

- 垃圾回收的根对象（GC Root）：不会被GC回收
  - 线程Thread对象：引用线程栈帧中的方法参数，局部变量等

  image-20250829140619318

  - 系统类加载器加载的 `java.lang.Class` 对象，引用类中的静态变量
    - 这个对象持有对该类所有静态变量的引用，因为这个对象是系统类加载器加载的，所有它不会被回收，也就意味着它引用的静态变量不会被回收

  image-20250829141616665

  - 监视器对象，用来保存同步锁 `synchronized` 关键字持有的对象
    - 被 `synchronized` 修饰的对象被监视器对象引用了，不能被回收
  - 本地方法调用时使用的全局对象
- 普通对象

五种对象引用

强引用

- 可达性分析算法就是强引用
- 对象被强引用，无法被回收

软引用

- 相较强引用是一种较弱的引用关系
- 当程序内存不足时，就会将软引用中的数据进行回收
- 常用于缓存中

- 可以通过 `SoftReference` 类来实现软引用，并且它的对象需要被**GC root**关联到，否则它也会被回收
  - 可以把这个 `SoftReference` 类的对象看成一个盒子，里面存的就是一个对象

```java
byte[] bytes = new byte(1024 * 1024 * 1000);
// 创建软引用
SoftReference<byte[]> softReference = new SoftReference<byte[]>(bytes);
// 取消强引用，后续内存不足的话会释放掉刚刚的bytes
bytes = null;
```

  - `SoftReference` 提供了一套队列机制：可以通过这个队列获取到不包含对象的软引用对象，实现软引用对象的回收
    1. 软引用创建时，通过构造器传入引用队列
    2. 在软引用中包含的对象被回收时，该软引用对象会被放入引用队列
    3. 通过代码遍历引用队列，把 `SoftReference` 的强引用删除

弱引用

- 和软引用基本一致，相比软引用更弱，不管内存够不够都会直接回收
- 使用 `WebReference` 类来实现弱引用，主要在 `ThreadLocal` 中使用
- 弱引用对象也可以使用引用队列进行回收

虚引用和终结器引用

- 常规开发不会使用
- 虚引用也叫幽灵引用/幻影引用，不能通过虚引用对象获取到包含的对象
  - 唯一作用是当对象被垃圾回收器回收时可以接受到对应的通知
  - 使用 `PhantomReference` 实现虚引用
  - 解决了直接内存的内存释放问题
    - 创建对象的时候需要向直接内存申请空间，当堆中的对象被释放掉后，直接内存的那部分空间也需要释放，此时可以使用虚引用监控堆中的对象，当堆中的对象被释放掉，接收通知，告诉直接内存释放内存
- 终结器引用：当对象需要被回收时，终结器引用会关联对象并放置在 `Finalizer` 类的引用队列中，然后□一条 `FinalizerThread` 线程从队列中获取对象，执行对象的 `finalize` 方法，对象第二次被回收时，才真正被回收
  - 可以用一个强引用在 `finalize` 方法中指向对象，从而实现对象的自救

垃圾回收算法

核心思想：

- 找到内存中存活的对象
- 释放不再存活对象的内存

注意：垃圾回收过程会通过单独的GC线程来完成，但是有部分阶段需要停止所有的用户线程，这个过程被称之为 `Stop The World`，即 `STW`，`STW` 时间过长会影响用户使用

三种评价标准：不可兼得！！！

- 吞吐量：**CPU** 用于处理用户业务的时间与**CPU** 总时间的比值
- 最大暂停时间：垃圾回收过程中，导致应用程序停顿的最长时间
- 堆使用效率：垃圾回收器管理堆内存的内存利用率和内存碎片化的程度

####### 标记清除算法

工作流程：

1. 标记阶段：使用可达性分析算法把所有存活的对象进行标记，用GC Root通过引用链遍历出所有存活的对象
2. 清除阶段：从内存删除没有被标记的对象

image-20250829160259912

缺点：

1. 碎片化问题：对象被删除后会出现很多细小的内存单元，如果需要大的存储空间是无法进行分配的
2. 分配速度慢：底层会用一个空闲链表去维护空闲的内存空间

####### 复制算法

核心思想：

1. 准备两块空间 `From` 空间和 `To` 空间，每次在对象分配阶段，只能使用 `From` 空间
2. 垃圾回收阶段，将 `From` 中存活对象复制到 `To` 空间
3. 将两块空间的 `From` 和 `To` 名字互换

image-20250829160328473

缺点：

- 内存利用效率低

优点：

- 吞吐量高

- 不会发生碎片化

####### 标记整理算法

工作阶段：

1. 标记阶段：使用可达性分析算法把所有存活的对象进行标记，用GC Root通过引用链遍历出所有存活的对象
2. 整理阶段：将存活对象移动到堆的一端，清理掉非存活对象的内存空间

image-20250829160016236

缺点：

- 整理阶段效率低

优点：

- 内存使用效率高

- 不会发生碎片化

####### 分代垃圾回收算法

将整个内存区域划分为年轻代和老年代：

- 年轻代存放存活时间比较短的对象
  - 伊甸园区（Eden）：对象刚创建的时候存储的区域
  - 幸存者区：实现复制算法

    - `S0`
    - `S1`
- 老年代存放存活时间比较长的对象

image-20250829171806375

工作流程:

- 创建出来的对象首先被放入Eden区
  - 如果Eden区满了，新创建的对象无法放入，就会触发年轻代的**GC**（Minor GC或者Young GC）
  - Minor GC会根据可达性分析算法判断 `Eden` 和 `From` 中哪些对象需要被回收，不需要回收的对象放进 `To` 区域
  - 然后 `From` 和 `To` 互换名字
- 每次Minor GC中都会为对象记录他的年龄，初始值为0，触发一次Minor GC年龄加1
- 晋升
  - 当年龄到达阈值，对象会被晋升到老年代
  - 当年轻代空间不足的时候，即使年龄没到达阈值，也会被晋升老年代中
- 当老年代中空间不足，无法放入新的对象时，先尝试Minor GC，如果还是不足，就会触发**Full GC**，对整个堆进行垃圾回收

为什么要把堆分成年轻代和老年代?

- 系统中的大部分对象都是在创建出来之后很快就不再使用，可以放入新生代
- 老年代存储的是需要长期存放的对象
- 新生代使用复制算法，老年代使用标记-清除或者标记整理算法
- 分代设计允许只回收新生代

垃圾回收器

可以用虚拟机参数指定要使用的垃圾回收器

组合关系

image-20250830162724929

Serial垃圾回收器

- 单线程串行回收年轻代的垃圾回收器
- 新生代的使用复制算法，老年代的使用标记**-**整理算法

image-20250830163035897

年轻代**-ParNew**垃圾回收器

- 使用多线程进行垃圾回收
- 本质上是对**Serial**在多**CPU**下的优化
- 可以与CMS老年代垃圾回收器搭配使用
- 使用复制算法

image-20250830163831847

老年代**-CMS**垃圾回收器

- 关注的是系统的暂停时间

- 允许用户线程和垃圾回收线程在某些步骤中同时执行
- 使用标记清除算法
- 工作步骤：
  1. 初始标记，标记GC Roots能直接关联到的对象
  2. 进行并发标记，标记所有的对象，找出哪些对象需要回收，哪些不需要，用户线程不需要暂停
  3. 重新标记，停下用户线程，并发标记阶段有些对象会发生变化，产生错标漏标等
  4. 并发清理，清理死亡的对象，用户线程不需要暂停
- 缺点：
  - CMS使用了标记清除算法，进行垃圾回收会出现大量内存碎片，需要进行整理，这需要让用户等待
  - 老年代内存不足会被迫进行 `Full GC` ，CMS会退化成**Serial Old**单线程回收老年代

image-20250830164027933

年轻代**-Parallel Scavenge**垃圾回收器

- 多线程并行回收
- 关注的是系统吞吐量
- 能够自动调整堆内存大小：堆内存小了，最大暂停时间就少了
- 使用复制算法
- JDK8默认使用

image-20250830165211992

老年代**-Parallel Old**垃圾回收器

- 使用标记整理算法
- 多线程并发
- 与 `Parallel Scavenge` 配合使用
- JDK8默认使用

image-20250830165359547

####### G1垃圾回收器！！！

需要预留堆空间！！！，否则堆空间不足，进行**Full GC**的时候会使用单线程执行标记整理算法

将 `CMS` 垃圾回收器和 `Parallel Scavenge` 的优点融合：

- 支持巨大的堆空间回收，并拥有较高的吞吐量
- 支持多**CPU**并行
- 允许用户设定最大暂停时间

内存结构：

- 将整个堆划分成多个大小相等的区域，称之为区**Region**，区域不要求连续
- 分为 `Eden` 、 `Survivor` 、 `Old` 区

image-20250830171153722

垃圾回收的两种方式：

- 年轻代回收（Young GC）

    - 回收 `Eden`区 和 `Survivor` 区中不用的对象
    - 会导致整个用户线程的停止，即**STW**
    - 会记录每次垃圾回收时每个Eden和Survivor区的平均耗时，作为下次回收的参考依据

- 混合回收（Mixed GC）

    - 回收所有年轻代和部分老年代的对象以及大对象区

    - 采用复制算法

    - 分为：初始标记、并发标记、最终标记、并发清理

        - 初始标记：标记 `GC Roots` 引用的对象为存活，多线程并行，停止用户线程
        - 并发标记：和用户线程并行执行，将第一步中标记的对象引用的对象标记为存活，把整个 `GC Roots` 引用链的对象标记为存活
        - 最终标记：标记一些引用改变漏标的对象（不管新创建、不再关联的对象）
        - 并发清理：和用户线程并行，使用的是复制算法，且根据存活率判断要清理哪部分 `Region`

    image-20250830173009222

执行流程：

1. 新创建的对象放置在Eden区，当G1判断出年轻代区空间不足（有一个阈值），无法分配对象，会执行**Young GC**
2. 标记出 `Eden` 和 `Survivor` 区域存活的对象
3. 将 `Eden` 和 `Survivor` 区域存活的对象复制到一个新的**Survivor**区（年龄+**1**），然后清空这些区域，使用了复制算法
4. 当某个存活对象的年龄达到阈值，放入老年代
5. 部分对象如果大小超过**Region**的一半，会直接放入老年代，这类老年代称为 `Humongous` 区（大对象区）。如果对象过大就会横跨多个 `Region`
6. 多次回收之后会出现很多 `Old` 老年代区，当堆占有率达到阈值就会触发混合回收

####### 垃圾回收器组合

- `ParNew` + `CMS` （关注暂停时间）
- `Parallel Scavenge` + `Parallel Old` （关注吞吐量）
- `G1` （需要较大堆且关注暂停时间）

内存调优

- 内存泄露：在Java中如果不再使用一个对象，但是该对象依然在**GC Root**的引用链上，这个对象就不会被垃圾回收器回收，这种情况称之为内存泄漏
- 内存泄漏绝大多数清空是□堆内存泄露引起的

image-20250831121409514

可以使用 `top` 命令去查看系统的资源

- 可以查看系统的负载
- `RES` ：常驻内存，进程实际占用物理内存的大小
- `SHR` ：共享内存，进程占用的可被其他进程共享的内存

堆内存状况对比：

image-20250831122640955

内存泄露的原因

代码中的内存泄漏

image-20250831123228657

1. 未重写 **`equals()`** 和 **`hashCode()`** 导致内存泄漏
    - 即使是相同的对象，由于未重写上述两方法，会占用哈希表的大量空间，导致内存泄漏
2. 内部类引用外部类
    - 非静态的内部类默认会持有外部类，只要使用了这个内部类，外部类就无法被回收

```
public class Outer{
    private byte[] bytes = new byte[1024*1024];
    private String name = "测试";
    class Inner{
        private String name;
        public Inner(){
            this.name = Outer.this.name;
        }
    }
}
```

    - 匿名内部类对象如果在非静态方法中被创建，会持有无法被回收的调用者对象，需要改成静态方法

```
public class Outer{
    private byte[] bytes = new byte[1024*1024];
    public List<String> newList(){
        // 创建匿名内部类
        List <String> list = new ArrayList<String>(){{
            add("1");
            add("2");
        }}
        return list;
    }

    public static void main(String []args){
        int count = 0;
        ArrayList<Object> objects = new ArrayList<>();
        while(true){
            System.out.println(++count);
            // Outer对象不能被回收
            objects.add(new Outer().newList());
        }
    }
}
```

3. **`ThreadLocal`** 的使用
    - new出来的线程对象可以不用调用 `remove` 方法移除线程
    - 使用了线程池需要自己调用 `remove` 方法

4. **`String`** □ **`intern`** □□

   - JDK6□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `intern` □□□□□□□□□□□□□□□□□□□

5. □□□□□□□□□□

   - □□□□□□□□□□□□□□□□□□□□□□□□□□
   - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ **`null`**

□□□□□□

- □□□□□□□□□□□□□□
- □□□
- □□□ `Jmeter` □□□□□□□

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `Heap Profile` □□□□□

- □□□□□□□□□ `hprof` □□□□□□□□□□□ `MAT` □□ `hprof` □□□□□□□□□□□□□

MAT□□□□□□□□

□□□

- □□□□□□□□□□□□□□
- □□□□□□□□□□□□□□B□□□□□□□□A□□□□A□□□B

image-20250901101347721

□□□□□

- □□□□□□□□□□□□□□□□□
- □□
  - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□
- □□□

image-20250901101838489

GC□□

□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□

□□□

- □□□JVM□□□□
- □□□□□□□□JVM□□□□
- □□□□□□ `Full GC` □□□□□□□□□

□□□□□

1. □□□□□□□□□□□□□□□□□
2. □□□□□□□□□□□□□□□□□□□□□□□□
3. □□□□□□Java□□□□□□□□□□□□□

可以使用 `jstat` 工具进行内存监控

常见的GC模式

1. 正常情况：呈现锯齿状，创建对象后内存上升，GC后下降到底部

image-20250901104525184

2. 缓存对象过多：也是呈现锯齿状，处于比较高的位置

image-20250901104628452

3. 内存泄露：呈现锯齿状，每次垃圾回收之后下降到的内存位置越来越高

image-20250901104756886

4. 持续的 `Full GC`：在某个时间点产生多次Full GC，CPU使用率飙升，用户请求基本无法处理

image-20250901105053041

5. 元空间不足导致的 `Full GC`：堆内存太小，会导致持续的 `Full GC`

image-20250901105235378

GraalVM

一款高性能**JDK**，可以获得更低的内存使用率，更快的启动速度

两种模式：

- JIT模式：即时编译模式
  - 一次编写，到处运行
  - 通过内置的**Graal**即时□译器优化热点代码，生成更高性能的机器码
- AOT模式：提前编译模式
  - 通过源代码为特定平台创建可执行文件，即exe文件，不具备跨平台特性
  - 这种模式生成的文件称之为**Native Image**本地镜像

新一代的GC

垃圾回收器的技术演进

image-20250902102018076

不同垃圾回收器设计的目标

image-20250902102115597

Shenandoah

- 着眼于减少停顿时间，让用户线程和垃圾回收线程并行处理
- 处理大对象的时候性能非常非常差

ZGC

- 着眼于减少停顿时间，让用户线程和垃圾回收线程并行处理
- 吞吐量不佳
- 堆大小对STW时间无影响
- 最新版本支持分代

栈上的数据存储

八大数据类型

image-20250903112910186

这里的内存占用指的是在堆上或者数组中内存分配的空间大小，在栈上的实现更加负载

每个局部变量表数组元素空间大小

- 32位虚拟机为32位，**4**个字节
- 64位虚拟机为64位，**8**个字节

不管是几位虚拟机，long和double这种需要8个字节存储的类型都会占用两个数组元素的位置(slot槽)

且操作数栈里的存储大小和局部变量表相同：

- 比如long在64位虚拟机是占用16个字节，那么它在操作数栈里也占用16个字节

image-20250903113535275

Java8大数据类型在虚拟机中的实现：

image-20250903113953843

注意：

- JVM采用的是空间换时间的方案，在栈上不存储具体的类型，只根据slot槽进行数据的处理，避免了不同数据类型不同处理方式带来的时间开销

Boolean在栈上的存储方式

在32位或者64位虚拟机里面都是占用一个槽

JVM在处理 `Boolean` 类型的时候会把其当成 `Int` 类型处理

- 1代表true
- 0代表false

栈中的数据要保存到堆上或者从堆中加载到栈上时要怎么处理？

1. 堆中的数据加载到栈上，栈上的空间大于等于堆上的空间，可以直接处理
    - `byte`、`short` 为有符号，低位复制，高位非负则补0，是负数则补1
2. 栈中的数据保存到堆上，`byte`、`short`、`char` □于堆上的空间较小，需要将高位去掉。`boolean` 只取低位的最后一位保存

对象在堆上的存储方式

对象在堆中的内存布局，指的是对象在堆中存放时的各个组成部分

- 普通对象
  - 对象头：存放的是基本信息
    - 标记字段（`Mark Word`）：保存锁、垃圾回收器等特定功能信息，32位4字节，64位8字节
    - 元数据的指针：指向方法区的`InstanceKlass`对象
  - 对象数据：
    - 会存储当前类每一个字段对应的数据
    - 内存对齐填充：把当前对象的长度做调整
- 数组对象
  - 对象头：存放的是基本信息
    - 标记字段（`Mark Word`）：保存锁、垃圾回收器等特定功能信息
    - 元数据的指针：指向方法区的`InstanceKlass`对象
    - 数组长度

image-20250903120801670

标记字段

在不同的对象状态下存放的内容是不同的

- 正常状态
  - `Hashcode`是每个对象的哈希值
  - 分代年龄记录对象的年龄，不超过15
- 偏向锁
- 轻量级锁
- 重量级锁
- 垃圾回收标记

image-20250903120952183

元数据指针

指向方法区中保存的`InstanceKlass`对象，保存了这个对象的地址

- 在32位虚拟机中，这个地址是4个字节
- 在64位虚拟机中，这个地址是8个字节

但是8个字节有点浪费，因此JVM使用了指针压缩，将堆中**8**个字节的指针压缩成**4**个字节，减少内存开销

指针压缩

思想：将寻址的单位放大，比如原来按1字节寻址，现在可以按8字节寻址

image-20250903151239017

问题：

1. 需要进行内存对齐，指的是将对象的内存占用填充至8字节的倍数
2. 寻址大小仅能支持2^{35}个字节，32GB。使用了指针压缩之后，变成了4字节存储，1个字节可以指向1个8字节的存储空间，即$2^3*2^{32}=2^{35}$，如下图右侧

image-20250903122649561

对象数据

内存对齐

会对当前内存进行填充，JVM必须保证每个对象的字节数是8的倍数

主要目的：解决并发情况下CPU缓存失效的问题

- 64位JVM的CPU的每一个缓存行都是8个字节
- 如果假设有A、B两个4字节的数据存储到一个缓存行
  - 对A的数据进行修改，此时缓存行要进行数据更新，也就是把A擦除，重新从内存读取新的数据A，但是擦除的最小单位是缓存行，也就是会把没被修改的B也删除
  - 此时假设有一个线程B去读取B数据，就会引起阻塞
- 因此把对象进行内存对齐后，这个对象是不可能和其它对象共用一个缓存行的

字段重排列

要求每个属性的偏移量Offset（字段地址-起始地址）必须是字段长度的N倍。比如下图，Student类中的id属性类型为long，那么偏移量就是8的倍数

image-20250903152602772

如果无法通过字段重排列满足偏移量Offset（字段地址-起始地址）必须是字段长度的N倍这个条件，JVM就会尝试内存对齐：在字段之间进行内存填充

方法调用的原理

本质上是通过字节码指令的执行，在栈上创建栈帧

- 以 `invoke` 开头的字节码指令的作用是执行方法的调用
  - `invokestatic`：调用静态方法
  - `invokespecial`：调用对象的 `private` 方法、构造方法等
  - `invokevirtual`：调用对象的非 `private` 方法
  - `invokeinterface`：调用接口对象的方法
- `invoke` 指令执行时，需要找到方法区中的 `instanceKlass` 中保存的方法相关的字节码信息

静态绑定

只适用于处理静态方法、私有方法或者使用final修饰的方法

1. 编译期间，`invoke` 指令会携带一个参数符号引用，引用到常量池中的方法定义（包含了类名、方法名、返回值、参数）
2. 方法第一次调用时，这些符号引用会被替换成内存地址的直接引用

动态绑定（可以实现多态）

- 对于可能被重写的方法，就需要使用动态绑定
- 是基于方法表来实现的，`invokevirtual` 使用了虚方法表(`vtable`)，`invokeinterface` 使用了接口方法表(`itable`)
- 每个类中都有一个虚方法表，本质上他是一个数组，记录了方法的地址
  - 子类方法表中包含父类方法表的所有方法，子类如果重写了父类方法，则使用自己类中方法的地址进行替换

image-20250903155034850

1. □□ `invokevirtual` □□□□□□□□□□□□□□□□□ `InstanceKlass` □□□□□□
2. □□□□□□□□□□□□□□□□□□

□□□□□

□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□

- □□PC□□□□□□□□□□□□□
- □□PC□□□□□□□□□□□□□
- □□PC□□□□□□□□□□□□□□□

image-20250903160648014

`finally` □□

1. `finally` □□□□□□□□ `try` □ `catch` □□□□□□ `try` □□ `catch` □□□□□□finally□□
2. □□□□□□ `catch` □□□□□□□□□□ `finally` □□□□□□□□□□□□□□□□ `try` □ `catch` □□□□□□
   □□□□□□ `finally` □□□□□□□□□□□□□□□

JIT□□□□

□□□□JVM□□□

HotSpot□□□□□

- C1
- C2
- Graal

image-20250904110706975

- C1□□□□C2□□□□□□C2
- C1□□□□□□□□□□□C2□□□□□□□□□□□□□

□□□

image-20250904111010015

C1□□□□C2□□□□□□□□□□□□□□□□□□□□□□□□□

image-20250904111838983

□□□□□

1. □□C1□□□□□□□□□□□□□□□□□□□□□□□C2□□f□□□□□□□□

image-20250904111724970

2. 方法字节码执行数目过少，先收集信息，□JVM判断出来C1和C2优化性能差不多，转为不收集信息，□C1进行优化

image-20250904111740070

3. C1线程都在忙碌，□C2进行优化

image-20250904111646895

4. C2线程忙碌，先□2层C1收集一些信息，再交□3层C1处理，最后C2线程不忙碌了再交□C2进行处理

image-20250904111702177

方法内联

方法体中的字节码指令直接复制到调用方的字节码指令中，节省了创建栈帧的开销

条件限制：

1. 方法编译后的字节码指令<35字节
2. 方法编译后的字节码指令<325字节，并且是热方法（热点代码）
3. 方法编译后的字节码指令不能大于1000字节
4. 一个接口的实现必须小于3个

逃逸分析

如果JIT发现在方法内创建的对象不会被外部引用，就可以采用锁消除、标量替换等方式进行优化

锁消除

如果对象不会被逃逸出去，那么对象就不存在并发访问问题，可以把`synchronized`去掉

标量替换

- 对性能优化较大的方式
- 如果方法中的对象不会逃逸，其中的标量（对象中的基本数据类型）就会直接在栈上进行分配
    - 不在堆上创建对象，把标量当成局部变量放在栈帧里进行处理

G1垃圾回收器

年轻代回收只扫描年轻代对象（`Eden` + `Survivor`），沿着GC Root引用链可以很轻松查找哪些对象不能被回收

image-20250904114208683

年轻代回收

整个过程是**STW**的

是否回收通过`GC Root`引用链判断是有局限性的，如果年轻代的对象被老年代引用，年轻对象如何回收？

image-20250906232331562

方案一：从`GC Root`对象开始扫描所有对象，在其引用链上的对象就标记为存活

- 缺点：如果引用链长，会增加大量的对象扫描，增加扫描时间，导致执行效率低

image-20250906232349688

方案二：维护一个详细的表，记录哪个对象被老年代引用

- 缺点：如果对象太多的话会占用内存空间，会存在错标情况

image-20250906232407451

优化后

- 将引用详情表转变为记忆集，最后进行引用链查找的时候把每个区域的每个块（卡页）里面存放的对象加入成 `GC Root` 对象，将他们引用链上的对象标记为存活
- 记忆集里面存储的是区域和卡页□号的映射
    - 不会记录新生代到新生代的引用

image-20250906232601037

卡表（**Card Table**）

- 每一个区域都有自己的卡表，其实是一个字节数组
- 卡表会将整个堆内存均分成512字节的卡页
    - 如果这个卡页出现跨代引用，这个卡页对应的卡表上会将字节内容进行修改
    - 字节内容为0的卡页称为脏卡
- 生成记忆集可以通过遍历卡表来实现

image-20250906233007584

写屏障（**Write Barrier**）

JVM使用写屏障技术，在执行引用关系确立，如 `a.f = f` 时，可以在代码前和代码后加入一段指令，从而维护卡表

- 声明老年代引用年轻代对象后，就更新卡表
- 写前屏障会记录旧的引用值，如 `a.f` 里面的对象
- 写后屏障会记录新的引用值，如 `f`

image-20250906233509833

记忆集的生成流程

1. 通过写屏障获得引用变更的信息
2. 将引用关系记录到卡表中，并记录到一个脏卡队列中
3. JVW中会□ `Refinement` 线程定期从脏卡队列中获取数据，生成记忆集。不直接写入记忆集的原因是避免过多线程并发访问记忆集，导致线程安全问题

image-20250906234122459

年轻代回收流程

1. Root扫描，将所有的静态变量、局部变量扫描出来
2. 处理脏卡队列中没有处理完的信息，更新记忆集的数据
3. 标记存活对象，记忆集中的对象会加入到 `GC Root` 对象集合中，在 `GC Root` 引用链上的对象会被标记为存活

4. 根据设定的最大停顿时间，选择本次收集的区域
5. 使用复制算法进行垃圾回收
6. 处理软、弱、虚、终结器引用以及JNI中的弱引用

混合回收

在`Young GC`后或者大对象分配之后会检查当前内存阈值是否达到上限

三色标记法

采用三色标记法进行初始的标记

- 黑色：当前对象在`GC Root`引用链上，同时它引用的其他对象已经标记完成，为1
- 白色：不在`GC Root`引用链上，为0
- 灰色：当前对象在`GC Root`引用链上，它引用的其它对象还未标记完成，会被存储到一个队列进行处理

image-20250906235256136

黑色与白色是使用位图（**bitmap**）来实现的

- 每8个字节使用1个bit来标识内容
  - 对象在堆中存储的字节会被对齐成8的倍数
- 对象超过8个字节仅仅使用第1个bit处理

image-20250906235613653

SATB

进行初始标记后进入并发标记，此阶段和用户线程并行，处理方式同上，对颜色进行进一步标记，但是会有个严重问题

- 此阶段和用户线程并行，如果用户线程修改对象的引用关系，就会出现错标
  - 这个案例中正常情况下，B和C都会被标记成黑色，但是在BC标记之前，用户线程执行了`B.c = null`，将B到C的引用去除了

    image-20250907000221166

  - 同时执行了`A.c= = c`，添加了A到C的引用，导致JVM不会处理C的引用

    image-20250907000236931

为了解决这个问题，使用了SATB技术

1. 标记开始时创建一个快照，记录当前所有对象，标记过程中新生成的对象直接标记为黑色
2. 采用前置写屏障技术，在引用赋值前比如`B.c = null`前，将之前引用的**c**对象放入**SATB**待处理队列中
3. 然后进入最终标记，用来处理SATB相关的对象标记，会将所有线程的**SATB**队列中剩余的数据汇总到总的**SATB**队列中
   - 总的SATB队列中的对象按照存活来进行处理

SATB的缺点是本轮清理可能会将不存活的对象标记为存活对象，只能等到下一轮来清理掉

转移

1. 根据最终标记结果和停顿时间，选择转移效率最高的几个区域
2. 先转移`GC Root`直接引用的对象，再转移其他对象

3. 回收老的区域，如果外部区域对象有引用转移对象，需要把引用关系重新确立

混合回收的步骤

1. 初始标记，**STW**，采用三色标记法标记从 `GC Root` 可直达的对象
2. 并发标记，GC线程和用户线程并发执行，对存活对象进行标记
3. 最终标记，**STW**，处理 `SATB` 相关的对象标记
4. 清理，**STW**，如果区域中无任何对象就直接清理
5. 转移，将存活对象复制到别的区域

ZGC

转移阶段是并发的，g1的转移阶段是不让用户线程执行的

读屏障（**Load Barrier**）

在转移阶段实现转移后对象的获取

- 如果用户线程尝试获取一个对象引用，并且这个对象已经进行，会采用读后屏障指令，把对象引用指向转移后的对象
- 例子：

```
F f = obj.f;
f.count = 2;
```

image-20250907102050286

着色指针

使用着色指针存储对象的状态信息

- 使用对象地址的高位去存储状态信息
    - 最低的44位用来表示对象地址，可以表示16TB的内存空间
    - 中间4位是颜色位，只能放0或1
        - 终结位：标记对象是否注册了终结器，标记了的话这个对象只能通过终结器访问
        - 重映射位（Remap）：标记对象是否已完成引用更新
        - Marked0和Marked1：标记对象是否存活（**1**为存活），但这两个标志位是交替标记的，比如Marked0是当前轮，Marked1是上一轮

    image-20250907102229127

内存划分

堆内存会被划分成很多个区域，这些区域被称为Zpage

- 小区域：只能保存256KB内的对象
- 中区域：32M，保存256KB-4M的对象
- 大区域：只保存一个大于4M的对象

ZGC工作流程

- 初始标记阶段：标记 `GC Root` 直接引用的对象
- 并发标记阶段：用户线程通过读屏障去判断指针是不是红色，也可以帮忙标。会遍历所有对象，

- 并发处理阶段：选择需要转移的Zpage，创建转移表用于记录转移前对象和转移后对象的地址
- 转移：先将 `GC Root` 直接关联的对象转移到新的Zpage中，再把剩余对象转移到新的Zpage中，将两个对象的地址记入转移映射表
    - 将不转移对象的Remapped指针的值设置成1，说明这个对象已经完成转移的处理
- 这一轮的垃圾回收结束，但其实并没有完成所有指针的重映射工作，会放到下一阶段来进行
    - 重映射工作通过转移映射表来实现，旧的映射会指向旧对象，然后根据转移映射表指向新对象
    - 如图中的2，5'

image-20250907104350757

ShenandoaGC

通过修改对象头设计来完成并发转移

- 会在对象头开启一个8字节空间作为前向指针

image-20250907105411214

- 指向转移之后的对象，如果没有就指向自己

image-20250907105423998

2.0版本优化了前向指针的位置，仅在转移阶段将其放入**Mark Word**中

image-20250907105940188

读前屏障

使用读前屏障在用户线程进行读取的时候判断前向指针有没有指向别的对象，如果指向了转移后的对象，就去操作转移后的对象

写入屏障

写入数据会使用写前屏障，判断对象头的GC状态，根据GC状态的值来判断当前处于垃圾回收的哪个阶段

- 如果用户线程和GC线程都尝试把复制对象写入前向指针，会使用CAS实现

常见问题

类加载器的作用是什么

类加载器负责把类的字节码文件用二进制方式读取转换成byte，然后调用虚拟机底层的方法把这部分信息存入堆和方法区

有几种类加载器

- 启动类加载器
- 扩展类加载器
- 应用程序类加载器
- 自定义类加载器

什么是双亲委派机制

每个java对象的类加载器保存了一个父类加载器，自底向上是否查找是否加载过，再□顶向下加载，确保类只加载一次并且核心类能够被加载

如何打破双亲委派机制

1. 重写 `loadClass` 方法
2. 使用SPI机制和线程上下文类加载器

Java内存分为哪几部分

Java内存中哪些部分会溢出

JDK7和8中的内存结构上的区别是什么