

java编译

- 使用 javac 将 .java 源文件编译成 .class 文件
- 运行 JVM 解释 .class 文件并执行其中的指令
- 编译选项

JVM

java虚拟机

- 虚拟机JIT（Just-In-Time）编译技术
- 虚拟机JVM（Java Virtual Machine）

虚拟机

- 虚拟机运行环境
- 虚拟机类加载器
- 虚拟机垃圾回收器
- 虚拟机安全策略

虚拟机

虚拟机



虚拟机

- 虚拟机运行环境JDK（Java Development Kit）
- 虚拟机类加载器
- 虚拟机垃圾回收器
  - 静态 final 常量池 ConstantValue
- 虚拟机安全策略
  - 代码 Code 块
- 虚拟机安全策略
  - 虚拟机安全策略

```
// 虚拟机运行环境

// 虚拟机Person类Person类Person类
public class Person implements java.io.Serializable{
    // 静态
    private String name;

    private static int count;

    // 静态常量池
    private static final int id = 0;

    // 静态
    // 虚拟机安全策略Code块
    public Person(String name) {
```

```

        this.name = name;
        count++;
    }

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

    public static int getCount() {
        return count;
    }
}

/**
 * 测试类
 * 测试SourceFile类
 */

```

测试

- 测试类
  - 测试类
  - Java测试类 CAFFBABE
- 测试类
  - 测试类\$测试类=测试类-44\$

测试

测试类

```

public class Test{
    private static final String name1 = "测试";
    private static final String name2 = "测试";
}

```

测试

测试类 static final 测试类

- 测试类
- 测试类
- 测试类
- 测试类

测试

测试类

- 测试类
- 测试类
  - main 测试 args 测试

测试

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

编译后

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

- `iconst_0` 常量 0 入栈
- `istore_1` 将栈顶元素存储到局部变量 1 中
- `iload_1` 将局部变量 1 中的值加载到栈顶
- `iadd` 将栈顶元素与 1 相加
- `iinc 1 by 1` 将局部变量 1 的值增加 1

编译后

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

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

编译后

1. 编译 `i = i++` 时，`load` 指令会先将 `i` 的值加载到栈顶，然后再将 `i` 的值增加 1，最后将栈顶的值存储回 `i`。

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

2. 编译 `j = ++j` 时，`load` 指令会先将 `j` 的值加载到栈顶，然后将 `j` 的值增加 1，最后将栈顶的值存储回 `j`。

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


编译后

클래스

- 클래스
- 클래스의 상속
- 클래스
- 클래스
- 클래스

클래스

1. 클래스의 상속
2. 클래스의 상속 InstanceClass 클래스의 상속

 image-20250827121529235


3. 클래스의 상속 java.lang.Class 클래스의 상속 JVM 클래스의 상속
- 클래스의 상속
  - 클래스의 상속 Class 클래스의 상속
  - 클래스의 상속
  - 클래스의 상속 new 클래스의 상속 java.lang.Class 클래스의 상속
    - 클래스의 상속 c++ 클래스의 상속
    - 클래스의 상속

클래스

- 클래스의 상속 Java 클래스의 상속
  - 클래스의 상속
  - 클래스의 상속
  - 클래스의 상속
  - 클래스의 상속
- 클래스의 상속
  - int 0
  - double 0.0
  - 클래스의 상속 null
  - 클래스의 상속 final 클래스의 상속
- 클래스의 상속

클래스

- 클래스의 상속
- 클래스의 상속 clinit 클래스의 상속
  - 클래스의 상속

 image-20250827165053890

클래스

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

```

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

    }
}

```

编译

```

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

```

- `putstatic #2 <init/Demo1.value : I>` 将常量池中的 2 放入 Demo1 的 value 中

运行

- 初始化
  - 加载 `final` 常量
  - 初始化静态变量
- `new` 操作
- `main` 方法
- `Class.forName()`
  - 加载类

编译

- 编译选项
- `clinit` 方法

编译

编译选项

编译选项

**JDK8**

编译

- `ClassLoader`
  - `Extension ClassLoader`
  - `Application ClassLoader` 加载 jar 包
- `Bootstrap ClassLoader`

编译选项

- Bootstrap 类加载器
- 负责加载 `java.lang.ClassLoader`

#### Bootstrap ClassLoader

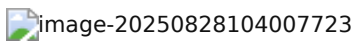
- 加载 `/jre/lib` 目录下的 `.jar`
- 加载 `rt.jar`
- 加载 `ext.jar`

#### Extension ClassLoader

- 由 `sun.misc.Launcher` 初始化 `URLClassLoader`
- 加载 `/jre/lib/ext` 目录下的 `.jar`

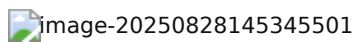
#### Application ClassLoader

- 由 `sun.misc.Launcher` 初始化 `URLClassLoader`
- 加载 `classpath` 下的 `.jar`



### JDK8

- JDK9 引入 `module` 系统
- 1. 由 `BootClassLoader` 加载 `java` 包下的 `jdk.internal.loader.ClassLoader`
- 2. `BootClassLoader` 加载 `BuiltinClassLoader`



3. 由 `BuiltinClassLoader` (Platform Class Loader) 加载 `BuiltinClassLoader`
4. 由 `BuiltinClassLoader` 加载 `BuiltinClassLoader`

类加载器

类加载器

类加载器

类加载器

- 类加载器
- 类加载器
- 类加载器
- 类加载器



类加载器

类加载器

类加载器

类加载器

Tomcat

- webTomcat
- 

image-20250828115521211

ClassLoader

- loadClass findClass
  - resolve
- findClass defineClass
- defineClass
- resolveClass

image-20250828120952341

- java.
- ApplacationClassLoader ClassLoader()

```
// 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){...}

    // 
    @Override
    protected Class<?> loadClass(String name) throws ClassNotFoundException{
        if(name.stratwith("java.")){
            return super.loadClass(name);
        }
        // 
        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(); // 
    }
}
```

SPI


- image-20250828142008540

[illegible]

# OSGI



- 

11111111

-  image-20250829110329036


5/5

**PC**

- 
- 

111

- 國際標準化組織
  - 國際標準化組織技術委員會
  - 國際標準化組織技術委員會

 image-20250828152526384

- Java CPU

1



- **Java** → Java
- `cpp` → native

- □□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□

□□□□□□□□

[illegible][illegible][illegible]

- StackOverflowError


- 虚拟机VM内存管理
- 内存 -Xss 内存大小
  - 内存1024KB

内存

内存Java内存

内存

- 内存管理
- 内存管理
- 内存管理
- 内存管理


 image-20250828181256837

- 内存管理 used 内存 total 内存 max
- used 内存 total 内存 max 内存管理
  - max 内存 total 内存管理
  - 内存管理 max 内存 total 内存管理

 image-20250828182043377

内存


- JDK7内存管理
  - 内存管理
- JDK8内存管理
  - 内存管理
  - 内存管理Class内存
- 内存管理

 image-20250828190129331

内存管理

内存管理

- 内存 InstanceClass 内存管理
- 内存 InstanceClass 内存管理
- 内存管理

 image-20250828184709777

内存管理

- 字符串常量池
- 字符串池

字符串池

- JDK7字符串池
- 字符串池
- new字符串池

jdk字符串池

.intern() 字符串池

jdk6

- jdk6 .intern() 字符串池
- jdk6字符串池
- 字符串池new字符串池

```
String s1 = new StringBuilder().append("think").append("123").toString(); // 字符串池
s1="think123"
System.out.println(s1.intern() == s1); // jdk6字符串池.intern()字符串池
false
```

jdk7

- jdk7字符串池.intern() 字符串池new字符串池
- jdk7字符串池new字符串池

```
String s1 = new StringBuilder().append("think").append("123").toString(); // 字符串池
s1="think123"
System.out.println(s1.intern() == s1); // jdk7字符串池.intern()字符串池
true
```

字符串

jdk4字符串NIO字符串

1. Java字符串池
2. 字符串IO字符串池

字符串

字符串池

- Garbage Collection字符串GC字符串池
- 字符串池C#pythonGo字符串池

字符串池

字符串池

1. 字符串池
2. 字符串池
3. 字符串 java.lang.Class 字符串池

## System.gc()

- 垃圾回收器
- 垃圾回收器是虚拟机VM的重要组成部分

垃圾回收器

- 垃圾回收器是虚拟机VM的重要组成部分

垃圾回收器

- Java垃圾回收器是虚拟机VM的重要组成部分
- 垃圾回收器是虚拟机VM的重要组成部分


垃圾回收器

- 垃圾回收器是虚拟机VM的重要组成部分
- 垃圾回收器是虚拟机VM的重要组成部分

垃圾回收器


垃圾回收器GC root

垃圾回收器


image-20250829130545212

垃圾回收器

- 垃圾回收器GC Root
- Thread

image-20250829140619318

- java.lang.Class
- 

image-20250829141616665

- synchronized
- synchronized
- 
- 

垃圾回收器

垃圾回收器

- 
- 

垃圾回收器

- 
- 
-

- `SoftReference` `ObjectReference`

```
byte[] bytes = new byte(1024 * 1024 * 1000);  
// [][][][]  
SoftReference<byte[]> softReference = new SoftReference<byte[]>(bytes);  
// [][][][][][][][][][][][][][]bytes  
bytes = null;
```

- **SoftReference**
  1. **WeakReference**
  2. **SoftReference**
  3. **StrongReference**

111

- 00000000000000000000000000000000
- 00 WebReference 0000000000 ThreadLocal 000
- 00000000000000000000

□□□□□□□□

- `Object`
- `Object`/`Object`
  - `Object`
  - `PhantomReference`
  - `Object`
    - `Object`
- `Finalizer` `FinalizerThread` `finalize`
- `finalize`

Page 10 of 10

□□□□□

- □□□□□□□□
- □□□□□□□□

**G**C Stop The World STW STW

□□□□□□□□□□□□

- CPU 0 CPU 1
- 00000000000000000000000000000000
- 00000000000000000000000000000000

#####

5/5

1. 在GC Root处添加断点
2. 设置断点



111

1.
2.

#####

□□□□□

1. 000000 From 000 To 00000000000000000000 From 00
2. 00000000 From 00000000 To 00
3. 000000 From 0 To 0000



111

- □□□□□□

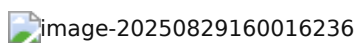
111

- 
- 

#####

5/5

1. 00000000000000000000000000000000GC Root0000000000000000
2. 00000000000000000000000000000000



111

- □□□□□□

111

- 
- 

#####

□□□□□□□□□□□□□□□□

- `java.lang.Object`
  - `java.lang.Object`Eden
  - `java.lang.Object`

- S0
- S1

- 0000000000000000



00000

- 000000000000Eden
  - 00Eden0000000000000000000000**GC**Minor GCYoung GC
  - Minor GC000000000000 Eden 0 From 00000000000000000000 To 00
  - 00 From 0 To 0000
- 00Minor GC0000000000000000000000000000Minor GC0001
- 00
  - 00000000000000000000
  - 000000000000000000000000000000000000
- 000000000000000000000000Minor GC000000000000**Full GC**000000000000

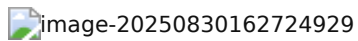
0000000000000000

- 000000000000000000000000000000000000
- 000000000000000000
- 00000000000000000000-000000000000
- 00000000000000

00000

000000000000000000000000

0000



**Serial**00000

- 000000000000000000
- 0000000000000000000000-00000



000-**ParNew**00000

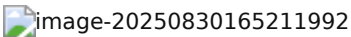
- 000000000000
- 00000**Serial**00**CPU**0000
- 000CMS000000000000
- 000000



000-**CMS**00000

- 000000000000

-  image-20250830164027933

[illegible]

- 800000000
- 80000000
- 1 Parallel Scavenge 80000000
- JDK8 80000000



- 記憶體空間
- 記憶體**CPU**埠
- 記憶體存取時間

- `Region`
- `Eden` `Survivor` `Old`



- `Young GC`



- Eden Survivor
- **STW**
- Eden Survivor
- Mixed GC
  - 
  - 
  - - GC Roots
    - GC Roots
    - 
    - Region

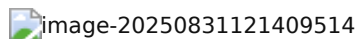


1. Eden G1 Young GC
2. Eden Survivor
3. Eden Survivor Survivor 1
- 4.
5. Region Humongous Region
6. Old

#####

- ParNew + CMS
- Parallel Scavenge + Parallel Old
- G1

- Java GC Root
- 



top

- 
- RES
- SHR

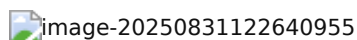


 image-20250831123228657

[illegible]

☐

[illegible]

- new [ ] remove [ ]
- [ ] remove [ ]

#### 4. String 与 intern

- JDK6之前，字符串常量存储在永久代，JDK7及以后，字符串常量存储在堆内存，通过 intern 方法可以获取字符串常量池中的字符串。

#### 5. 字符串常量池

- 字符串常量池中的字符串是不可变的。
- 字符串常量池中的字符串是唯一的，如果两个字符串的内容相同，那么它们指向的是同一个内存地址，即 null。

字符串常量池

- 字符串常量池中的字符串是不可变的。
- 字符串常量池中的字符串是唯一的，如果两个字符串的内容相同，那么它们指向的是同一个内存地址。
- 使用 Jmeter 测试字符串常量池。

字符串

字符串常量池中的字符串是不可变的，Heap Profile 字符串

- 字符串常量池中的字符串是不可变的，MAT 中的 hprof 字符串常量池。

MAT字符串常量池


字符串

- 字符串常量池中的字符串是不可变的。
- 字符串常量池中的字符串是唯一的，如果两个字符串的内容相同，那么它们指向的是同一个内存地址，即 B字符串常量池A字符串常量池B。

image-20250901101347721

字符串

- 字符串常量池中的字符串是不可变的。
- 字符串常量池中的字符串是唯一的，如果两个字符串的内容相同，那么它们指向的是同一个内存地址。
- 字符串常量池中的字符串是不可变的。
- 字符串常量池中的字符串是唯一的，如果两个字符串的内容相同，那么它们指向的是同一个内存地址。

image-20250901101838489

## GC

字符串常量池中的字符串

字符串常量池中的字符串

字符串

- 字符串常量池中的字符串是不可变的。
- 字符串常量池中的字符串是唯一的，如果两个字符串的内容相同，那么它们指向的是同一个内存地址。
- 字符串常量池中的字符串是不可变的，Full GC 字符串常量池。

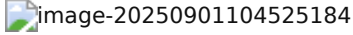
字符串

1. 字符串常量池中的字符串是不可变的。
2. 字符串常量池中的字符串是唯一的，如果两个字符串的内容相同，那么它们指向的是同一个内存地址。
3. 字符串常量池中的字符串是不可变的，Java字符串常量池。

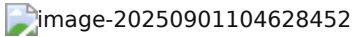
```
0000 jstat 00000000
```

□□□GC□□

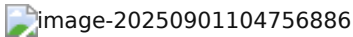
1. GC



2.



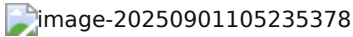
3. □□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□



4. ☐ Full GC ☐ Full GC ☐ CPU ☐



5. ☐ Full GC ☐ Full GC



# GraalVM

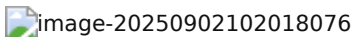
**JDK**

□□□□□

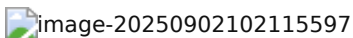
- JIT
  - 
  - **Graal**
- AOT
  - exe
  - **Native Image**

□□□□GC

□□□□□□□□



□ □ □ □ □ □ □ □ □ □ □ □



## Shenandoah

- □□□□□□□□□□□□

**ZGC**

- □□□□□□□

--	--	--	--	--	--	--

□□□□□□

[illegible]

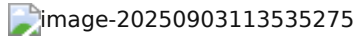
□□□□□□□□□□□□□□

- 64□□□□64□□**8**□□□

```
long double 8 (slot)
```

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

- `long` 64 bits, `int` 32 bits, `short` 16 bits, `byte` 8 bits



## Java8



111

- JVM slot

## Boolean

3264

JVM Boolean Int

- 0  $\Rightarrow$  false


[illegible]

2. `byte` `short` `char` `boolean`

## 内存管理

内存管理是 JVM 的重要组成部分

- 垃圾回收
  - 垃圾回收器
    - 垃圾回收 Mark Word 垃圾回收器垃圾回收器324648
    - 垃圾回收器垃圾回收器 InstanceClass 垃圾回收器
  - 垃圾回收
    - 垃圾回收器垃圾回收器
    - 垃圾回收器垃圾回收器
- 垃圾回收
  - 垃圾回收器垃圾回收器
    - 垃圾回收 Mark Word 垃圾回收器垃圾回收器
    - 垃圾回收器垃圾回收器 InstanceClass 垃圾回收器
    - 垃圾回收器

 image-20250903120801670

## 垃圾回收

垃圾回收是 JVM 的重要组成部分

- 垃圾回收
  - Hashcode 垃圾回收器
  - 垃圾回收器垃圾回收器15
- 垃圾回收
- 垃圾回收
- 垃圾回收
- 垃圾回收器

 image-20250903120952183

## 垃圾回收


垃圾回收器 InstanceClass 垃圾回收器垃圾回收器

- 32垃圾回收器垃圾回收器4垃圾回收器
- 64垃圾回收器垃圾回收器8垃圾回收器

垃圾回收器垃圾回收器JVM垃圾回收器垃圾回收器8垃圾回收器垃圾回收器4垃圾回收器垃圾回收器

垃圾回收

垃圾回收器垃圾回收器垃圾回收器1垃圾回收器垃圾回收器8垃圾回收器

 image-20250903151239017

垃圾回收

1. 内存地址从 0 开始，每 8 个字节为一个字。
2. 内存地址从 0 开始，每 32GB 为一个字。4 个字节为一个字，1 个字节为一个字，8 个字节为一个字。 $2^{32} = 2^{35}$  个字节。

 image-20250903122649561

内存

内存


内存地址从 0 开始，每 8 个字节为一个字。

内存地址从 0 开始，每 32GB 为一个字。

- 64 个 VM 的 CPU 地址从 0 开始，每 8 个字节为一个字。
- 内存地址从 0 开始，每 4 个字节为一个字。
  - A 地址从 0 开始，每 4 个字节为一个字。A 地址从 0 开始，每 4 个字节为一个字。B 地址从 0 开始，每 4 个字节为一个字。
  - B 地址从 0 开始，每 4 个字节为一个字。B 地址从 0 开始，每 4 个字节为一个字。
- 内存地址从 0 开始，每 4 个字节为一个字。

内存

内存地址从 0 开始，每 8 个字节为一个字。Offset 地址从 0 开始，每 8 个字节为一个字。Student 地址从 0 开始，每 8 个字节为一个字。

 image-20250903152602772

内存地址从 0 开始，每 8 个字节为一个字。Offset 地址从 0 开始，每 8 个字节为一个字。VM 地址从 0 开始，每 8 个字节为一个字。

内存

内存地址从 0 开始，每 8 个字节为一个字。

- `invoke` 方法
  - `invokestatic` 方法
  - `invokespecial` 方法 `private` 方法
  - `invokevirtual` 方法 `private` 方法
  - `invokeinterface` 方法
- `invoke` 方法 `instanceClass` 方法

内存

内存地址从 0 开始，每 8 个字节为一个字。final 方法

1. `invoke` 方法
2. `invoke` 方法

内存

- `invokevirtual` 方法 `( vtable )` `invokeinterface` 方法 `( itable )`
- `invokevirtual` 方法 `( vtable )` `invokeinterface` 方法 `( itable )`
  - `invokevirtual` 方法 `( vtable )` `invokeinterface` 方法 `( itable )`



1. invokevirtual `java.lang.Object` `InstanceClass`
2. `java.lang.Object`

11

111

111

- PC
- PC
- PC



fi

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

**דון**

111

Hot

- C1
- C2
- Graal



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

111



C1[



111

1.  $C_1$   $C_2$   $f$







-  image-20250906232349688

 image-20250906232407451



image-20250906232601037

 image-20250906233007584

 image-20250906233509833

 image-20250906234122459

1. Root
- 2.
3. GC Root GC Root

4.
5.
6. JN1


□□□□

Young GC

555

□□□□□□□□□□

- [illegible]

 image-20250906235256136

`bitmap`

- [illegible]

 image-20250906235613653

**SATB**[illegible]

- 数据库事务的ACID特性
  - 数据库事务的ACID特性: Atomicity, Consistency, Isolation, Durability. B.c = null 数据库事务

 image-20250907000221166

- `A.c = c` `A.C` JVM `C`

 image-20250907000236931

SATB

1. 000
2. 00000000000000000000 B.c = null 00000000c000SATB00000000
3. 000000000000SATB0000000000000000SATB000000000000SATB000  
o 00SATB0000000000000000

[illegible]

11

1. `00000000000000000000000000000000`
2. `000 GC Root 0000000000000000`

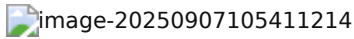




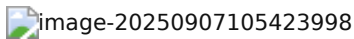
# ShenandoaGC

□□□□□□□□□□□□□□□□

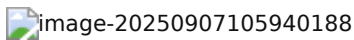
- □□□□□□□8□□□□□□□□



- □□□□□□□□□□□□□□□□



**2.0**



1111

[illegible]

1111

GC GC





- ☐ ☐ ☐ ☐ ☐ ☐ GC ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ CAS ☐ ☐

□□□□

□ □ □ □ □ □ □ □ □ □

byte

□□□□□□

- 
- 
- 
- 

□□□□□□□□

```

java

```

0000000000

- 1. 00 loadClass 00
- 2. 00SPI000000000000

Java00000000

Java00000000

JDK70800000000000