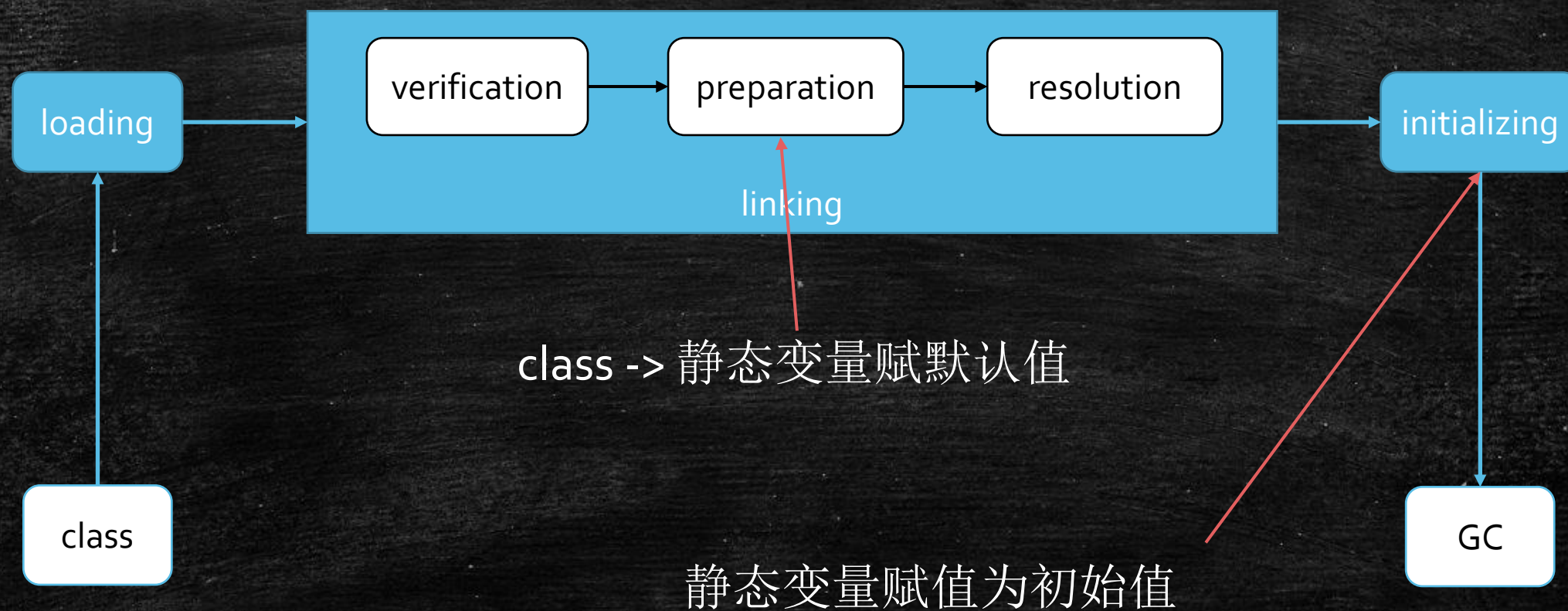


Class Loading Linking Initializing

马士兵

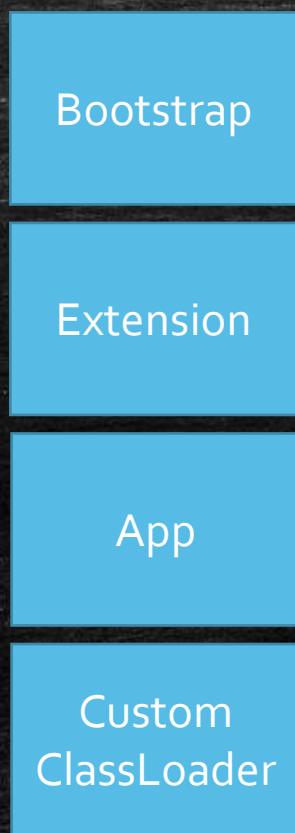
class cycle



类加载器

JVM是按需动态加载
采用双亲委派机制

自底向上检查
该类是否已经加载
parent方向



自顶向下
进行实际查找和加载
child方向



加载lib/rt.jar charset.jar等核心类, C++实现

加载扩展jar包
jre/lib/ext/*.jar,
或由-Djava.ext.dirs指定

加载classpath指定内容

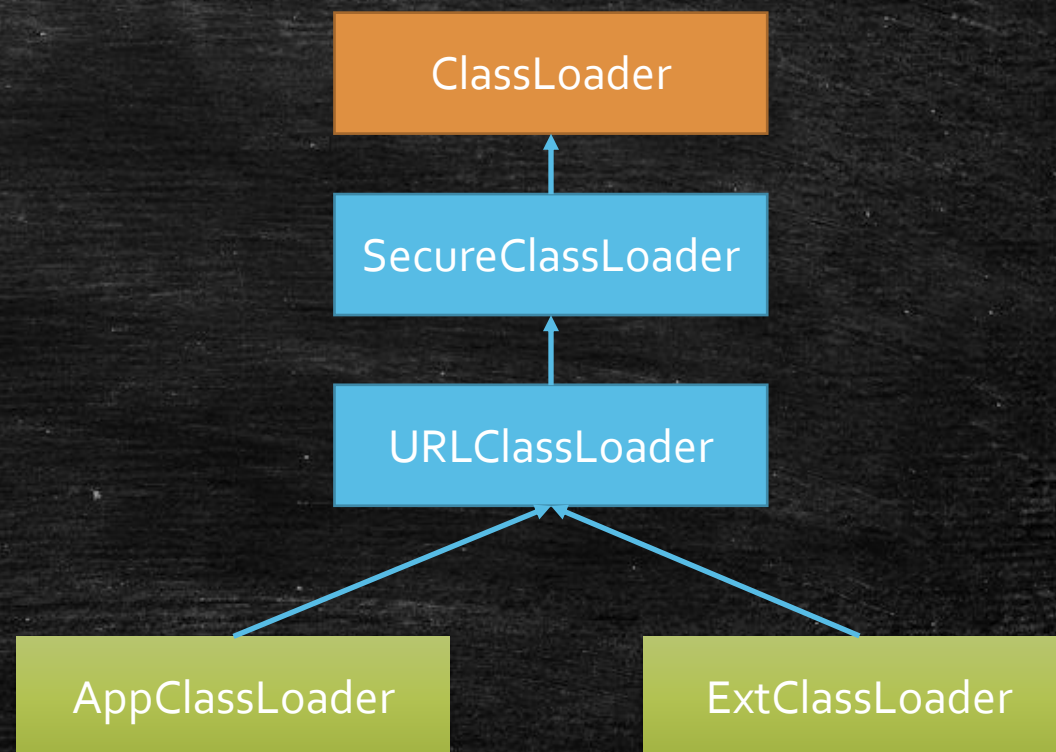
自定义ClassLoader

双亲委派

- 父加载器
 - 父加载器不是“类加载器的加载器”！！！！！！也不是“类加载器的父类加载器”
- 双亲委派是一个孩子向父亲方向，然后父亲向孩子方向的双亲委派过程
- 思考：为什么要搞双亲委派
 - java.lang.String类由自定义类加载器加载行不行？

com.mashibing.jvm.classloader.Too4_ParentAndChild

类加载器继承关系



类加载器范围

- (来自Launcher源码)
 - sun.boot.class.path
 - Bootstrap ClassLoader加载路径
 - java.ext.dirs
 - ExtensionClassLoader加载路径
 - java.class.path
 - AppClassLoader加载路径

`com.mashibing.jvm.classloader.T003_ClassLoaderScope`

自定义类加载器

- 继承ClassLoader
- 重写模板方法findClass
 - 调用defineClass
- 自定义类加载器加载自加密的class
 - 防止反编译
 - 防止篡改

`com.mashibing.jvm.classloader.T005_MSBCClassLoader`

`com.mashibing.jvm.classloader.T007_MSBCClassLoaderWithEncription`

ClassLoader源码解析

▪ loadClass

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, resolve: false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            }
        }
    }
}
```

```
if (c == null) {
    // If still not found, then invoke findClass in order
    // to find the class.
    long t1 = System.nanoTime();
    c = findClass(name);
}
```


ClassLoader源码解析

- findClass
- 如果是AppClassLoader首先会执行URLClassLoader的findClass方法
- 思考这是一个什么设计模式?

```
String path = name.replace( oldChar: '.', newChar: '/' ).concat(".class");
Resource res = ucp.getResource(path, false);
if (res != null) {
    try {
        return defineClass(name, res);
    } catch (IOException e) {
        throw new ClassNotFoundException(name, e);
    }
}
```

defineClass最终把二进制流转换为Class类对象

混合模式

- 解释器
 - bytecode interpreter
 - JIT
 - Just In-Time compiler
 - 混合模式
 - 混合使用解释器 + 热点代码编译
 - 起始阶段采用解释执行
 - 热点代码检测
 - 多次被调用的方法（方法计数器：监测方法执行频率）
 - 多次被调用的循环（循环计数器：检测循环执行频率）
 - 进行编译
- -Xmixed 默认为混合模式
开始解释执行，启动速度较快
对热点代码实行检测和编译
 - -Xint 使用解释模式，启动很快
执行稍慢
 - -Xcomp 使用纯编译模式，执行
很快，启动很慢

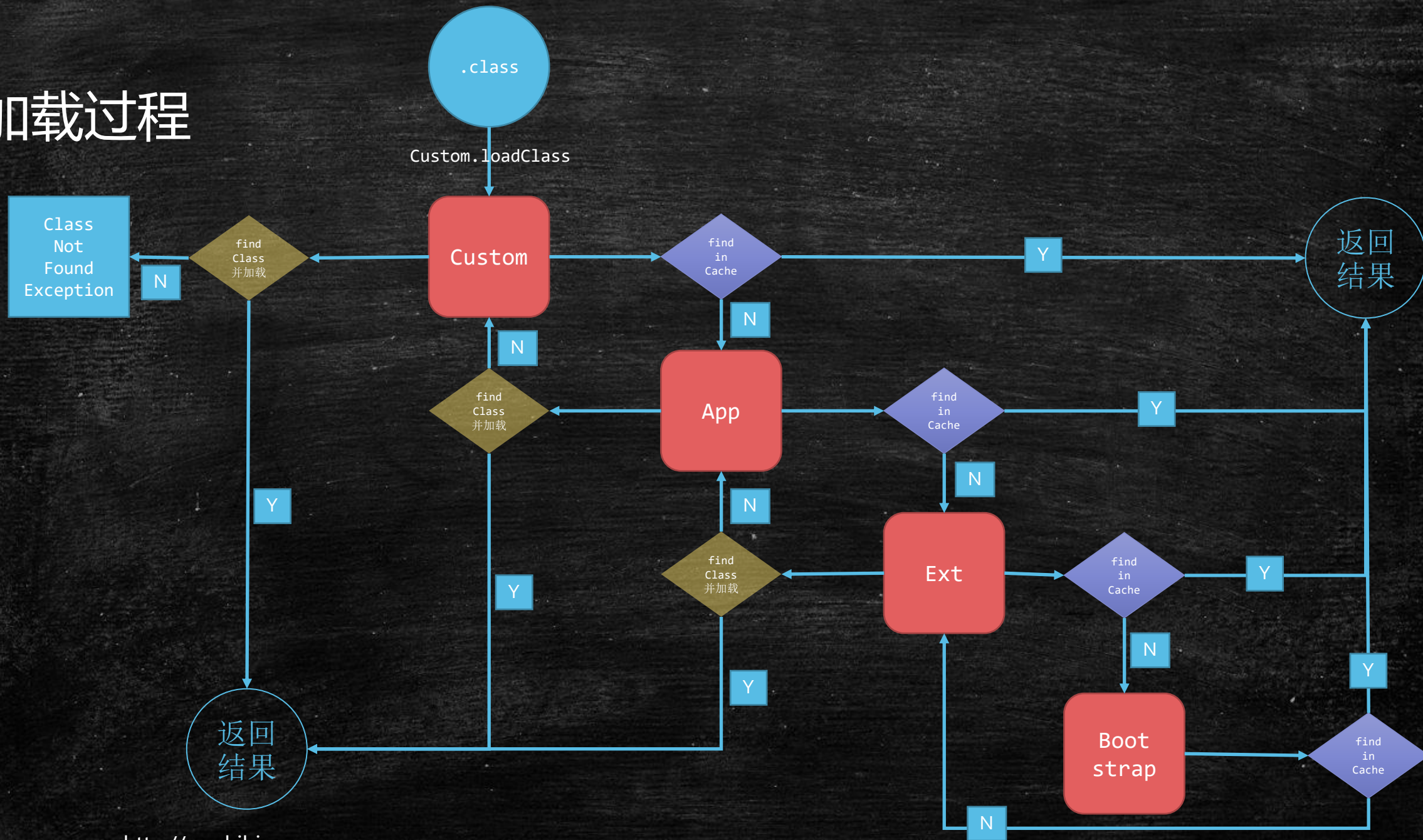
`com.mashibing.jvm.classloader.T009_WayToRun`

lazyloading

- 严格讲应该叫lazyInitializing
- JVM规范并没有规定何时加载
- 但是严格规定了什么时候必须初始化
 - new getstatic putstatic invokestatic指令，访问final变量除外
 - java.lang.reflect对类进行反射调用时
 - 初始化子类的时候，父类首先初始化
 - 虚拟机启动时，被执行的主类必须初始化
 - 动态语言支持java.lang.invoke.MethodHandle解析的结果为REF_getstatic REF_putstatic REF_invokestatic的方法句柄时，该类必须初始化

`com.mashibing.jvm.classloader.T008_LazyLoading`

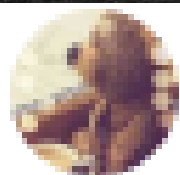
类加载过程



-
- 针对小白
 - java0基础后端课 SSM
 - 双料架构师
 - AI

volatile Object o = new Object()

```
1  0 new #2 <java/lang/Object>
2  3 dup
3  4 invokespecial #1 <java/lang/Object.<init>>
4  7 astore_1
5  8 return
```

點點點

已经有原子性了，重排也会执行完所有的代码，也并不会读到半初始化状态啊

一道面试题

```
package com.mashibing.jvm.classloader;

public class T001_ClassLoadingProcedure {
    public static void main(String[] args) {
        System.out.println(T.count);
    }
}

class T {
    public static T t = new T();
    public static int count = 2;

    private T() {
        count++;
    }
}
```

- A. 2
- B. 3
- C. 0
- D. 1

调整顺序

?

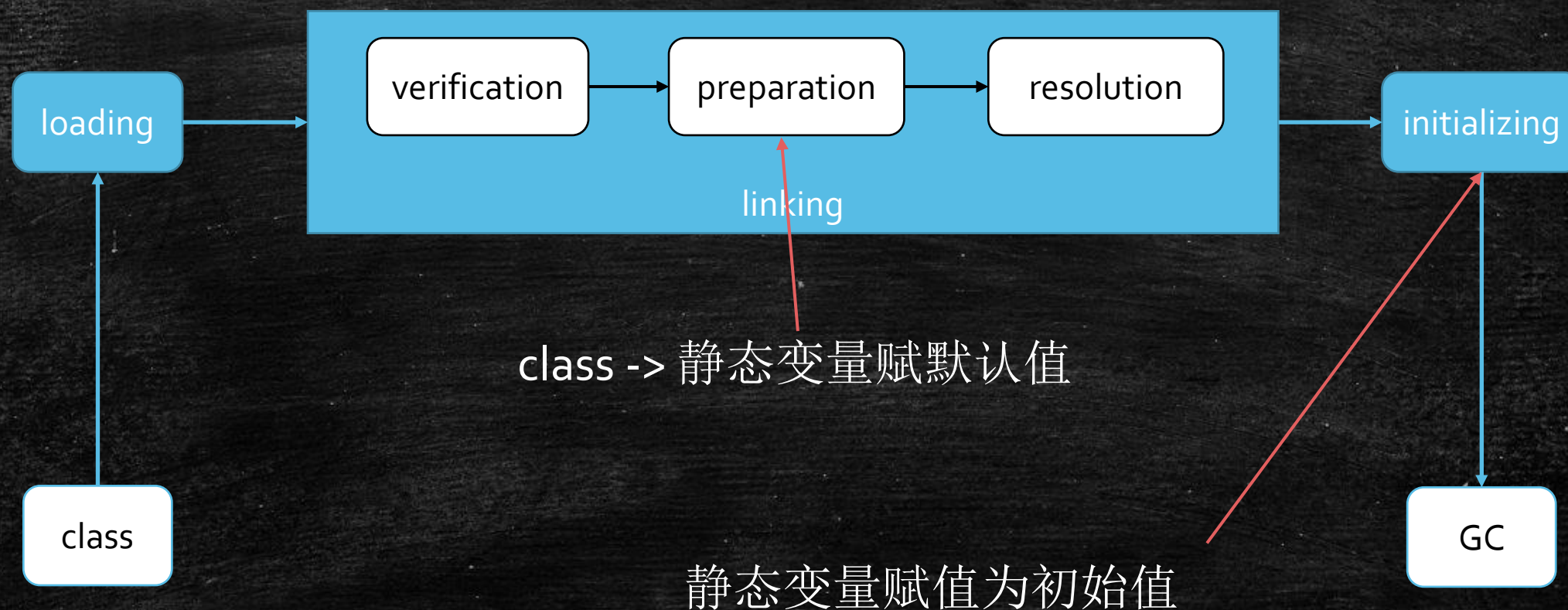

```
package com.mashibing.jvm.classloader;

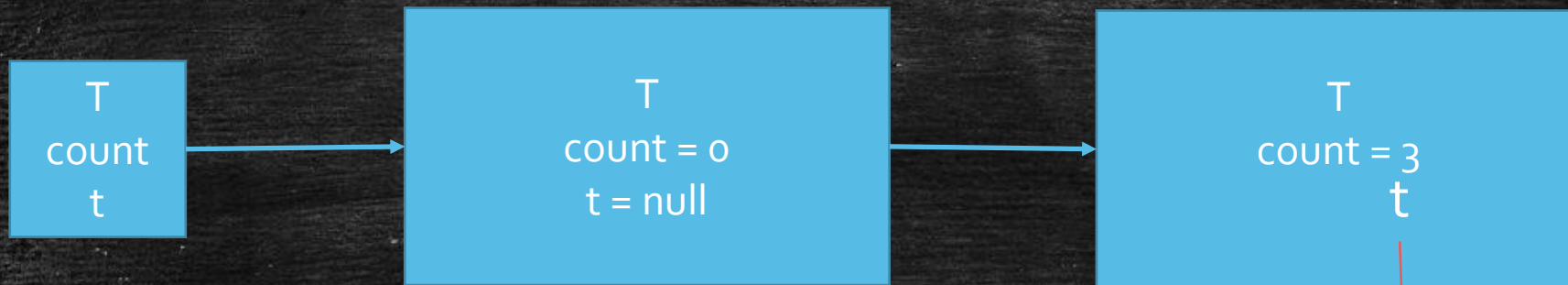
public class T001_ClassLoadingProcedure {
    public static void main(String[] args) {
        System.out.println(T.count);
    }
}

class T {
    public static int count = 2;
    public static T t = new T();

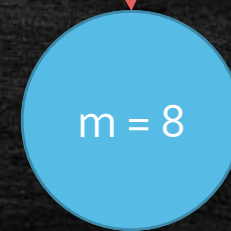
    private T() {
        count++;
        //System.out.println("--" + count);
    }
}
```


class cycle

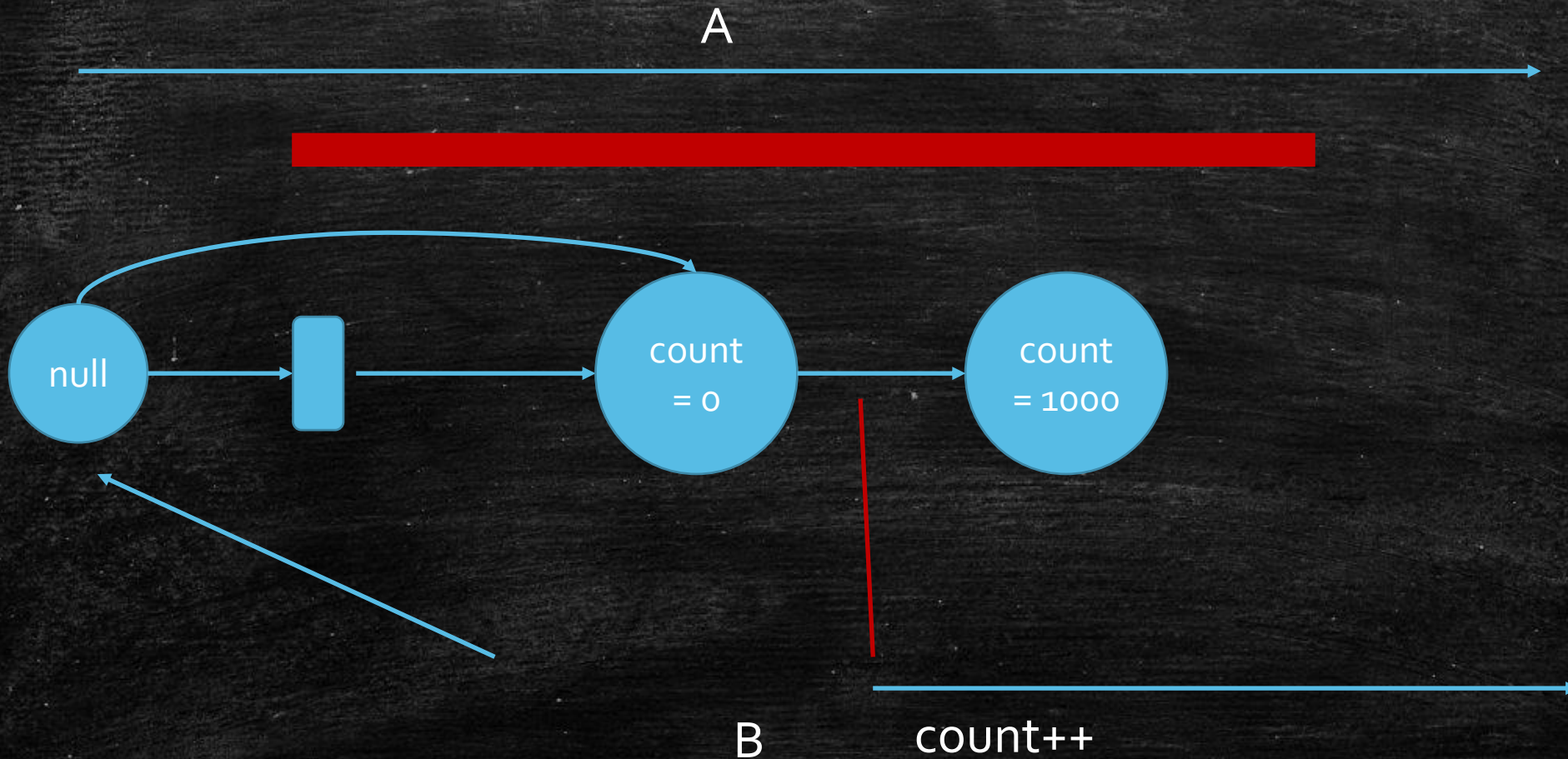




```
class T {  
    public static T t = new T(); // null  
    public static int count = 2; //0  
  
    //private int m = 8;  
  
    private T() {  
        count++;  
        System.out.println("--" + count);  
    }  
}
```



double check singleton



指令重排序

```
public class T {  
    public static void main(String[] args) {  
        T t = new T();  
    }  
}
```

```
1 0 new #2 <T>  
2 3 dup  
3 4 invokespecial #3 <T.<init>>  
4 7 astore_1  
5 8 return
```

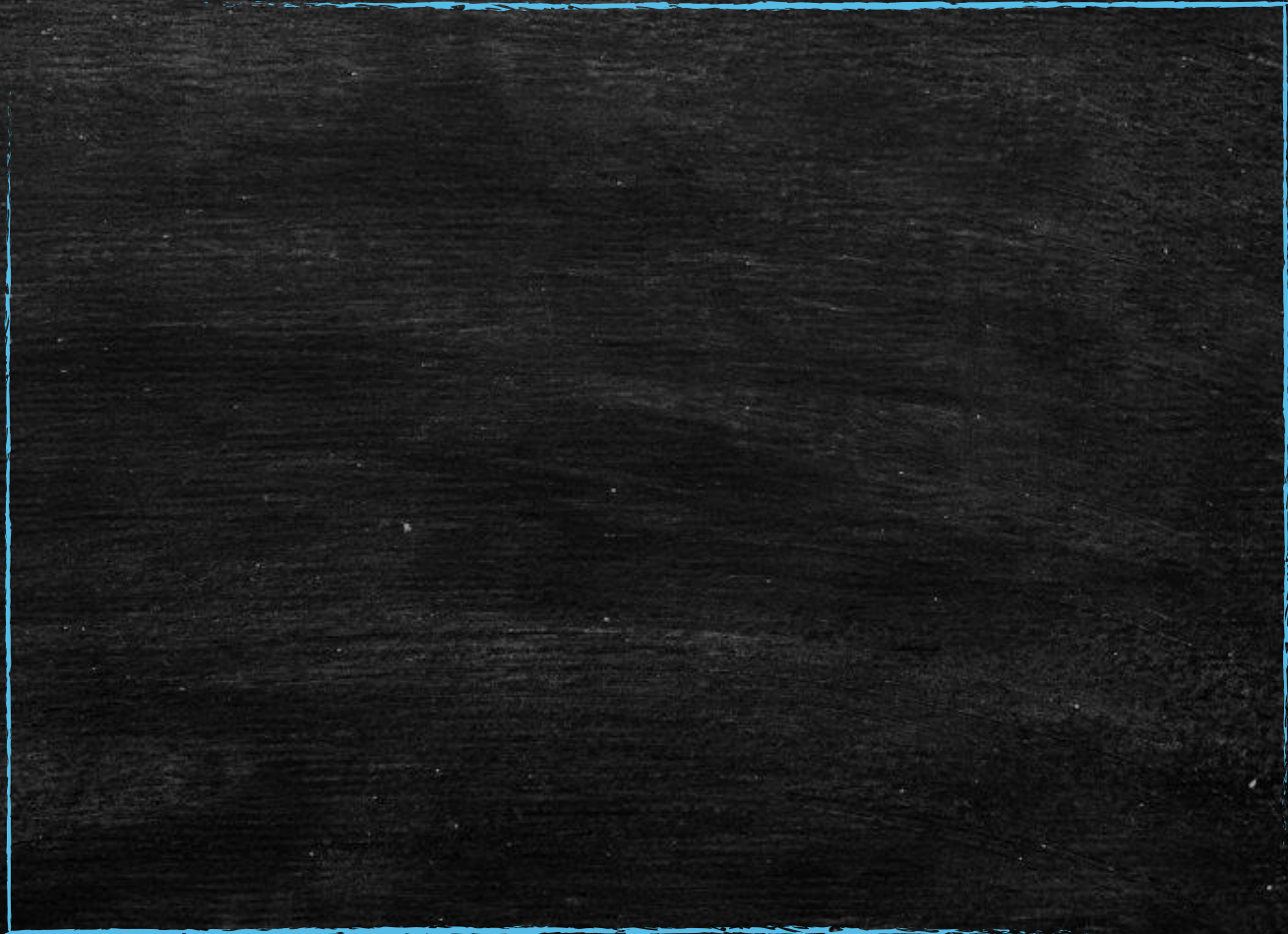

volatile

- 线程间可见性
- 防止指令重排序

preparation - initialization

- preparation
 - 静态变量设定为默认值
- initialization
 - <clinit>
 - 设定为人工指定值

添加幻灯片标题 - 4



添加幻灯片标题 - 5

