```
NNAMED
        //运行时反射需要添加 --add-
opens=jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED
        Field maxLevel =
com.sun.tools.javac.code.Source.Feature.MODULES.getDeclaringClass().
getDeclaredField("maxLevel");
        maxLevel.setAccessible(true);
        removeFinal(maxLevel);
        maxLevel.set(Source.Feature.MODULES, Source.JDK1_2);
    } catch (Exception ignored) {
    }
}

```

其中removeFinal方法的Java17实现是：

```
private static final Field MODIFIERS_FIELD;

static {
    Field field = null;
    try {
        //运行时反射需要添加 --add-opens=java.base/java.lang=ALL-
UNNAMED --add-opens=java.base/java.lang.reflect=ALL-UNNAMED
        Method getDeclaredFields0 =
Class.class.getDeclaredMethod("getDeclaredFields0", boolean.class);
        getDeclaredFields0.setAccessible(true);
        Field[] fields = (Field[]) getDeclaredFields0.invoke(Field.class,
false);
        for (Field f1 : fields) {
            if (f1.getName().equals("modifiers")) {
                f1.setAccessible(true);
                field = f1;
                break;
            }
        }
    } catch (Throwable ignored) {
    }
    MODIFIERS_FIELD = field;
}

@SneakyThrows
```

```
public static void removeFinal(Field field) {
    if (MODIFIERS_FIELD == null) {
        throw new UnsupportedOperationException("Can't remove final modifier,please add " +
                                                "--add-opens=java.base/java.lang=ALL-UNNAMED " +
                                                "--add-opens=java.base/java.lang.reflect=ALL-UNNAMED " +
                                                "to your jvm options");
    }
    MODIFIERS_FIELD.setInt(field, field.getModifiers() & ~Modifier.FINAL);
}
```

ps: 如果上述代码在IDEA编译时出现 "java: 不允许在使用 --release 时从系统模块 jdk.compiler 导出程序包"，取消勾选 `构建、执行、部署〉编译器〉Java编译器〉使用'-release'选项进行交叉编译` 即可，如果是springboot 程序，则需要在 pom properties 中添加 `<maven.compiler.release></maven.compiler.release>`

解决问题3：关闭注解处理器

1. 首先添加 `-proc:none` 到参数：

```
Boolean result = compiler.getTask(null, memoryFileManager, diagnostics,
        List.of("-source", "17", "-target", "17", "-encoding", "UTF-8","-proc:none"), null
        , List.of(new MemoryInputJavaFileObject("LambdaContainer.java", source))).call();
```

2. 在 MemoryFileManager 中 覆盖如下方法：

```
    static class MemoryFileManager extends ForwardingJavaFileManager<JavaFileManager> {
    /**
     * 关闭注解处理器
     */
```

```
    @Override
    public boolean hasLocation(Location location) {
        if (location == ANNOTATION_PROCESSOR_MODULE_PATH) {
            //返回true 交给 getServiceLoader 处理
            return true;
        }
        return super.hasLocation(location);
    }

    /**
     * 关闭注解处理器，通过空加载器减少扫描
     */
    @Override
    @SuppressWarnings("unchecked")
    public <S> ServiceLoader<S> getServiceLoader(Location location,
Class<S> service) {
        // load EMPTY
        // 如果你一定需要注解处理器，那你就需要实现加载指定的注解处理器
，可以考虑在这里实现, 向 ServiceLoader 传递一个自定义 ClassLoader 覆盖
其中的 getResources 方法来加载明确已知的的注解处理器的 SPI 文件 从而优
化性能，但是依然无法完全避免每次都重复读取 SPI 文件
        return (ServiceLoader<S>) ServiceLoader.loadInstalled(new Object()
{}.getClass());
    }

//省略其他部分.....
    }

```

解决问题4：复用解析结果

其实 JDK 中就有关于复用 JavaCompliler API 的类
：`com.sun.tools.javac.api.JavacTaskPool` 它被用在 JShell 中

要使用它，我们要添加参数`--add-exports
jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED`

同时，我们发现`com.sun.tools.javac.code.Types#candidatesCache` 竟然是一个减慢速度的cache，并且内部实现是`WeakHashMap`说明也不是起到防止无限循环的作用，在内存层面也没有观察到减少内存占用，非常迷惑，由于解决方案更简单，甚至不用反射，这里就一并给出了

```
public static void main(String[] args) throws URISyntaxException {
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    JavacTaskPool javacTaskPool = new JavacTaskPool(1);
    while (true) {
        DiagnosticCollector<JavaFileObject> diagnostics = new
DiagnosticCollector<>();
        StandardJavaFileManager fileManager =
compiler.getStandardFileManager(diagnostics, null, null);
        MemoryFileManager memoryFileManager = new
MemoryFileManager(fileManager);

        List<MemoryOutputJavaFileObject> result =
javacTaskPool.getTask(null, memoryFileManager, diagnostics,
                List.of("-source", "17", "-target", "17", "-encoding", "UTF-
8", "-proc:none"), null
                , List.of(new
MemoryInputJavaFileObject("LambdaContainer.java", source)), t -> {
                        Types types = Types.instance(((JavacTaskImpl)
t).getContext());
                        // 这个 cache 会导致速度大量下降??，所以禁用
                        //noinspection rawtypes,unchecked
                        types.candidatesCache.cache = new HashMap() {
                            @Override
                            public Object put(Object key, Object value) {
                                return null;
                            }
                        };
                    if (Boolean.TRUE.equals(t.call())) {
                        return memoryFileManager.getOutputs();
                    }
                    return Collections.emptyList();
                });
        assert !result.isEmpty();
    }
```

然后你会惊奇的发现，JDK JavaCompliler API 比 Janino 还快！：

![6c7d6e2ab4063b64a600cc8817c944d.png](https://p6-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/c78f1c57ed68435aa2b21a9a2ede2b86~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1225&h=328&s=42250&e=png&b=1f2023)
(看到那个±10206的方差了吗？没错，是伏笔)

然后你会更惊奇的发现，它竟然内！存！泄！露！了！ 方差大的原因就是最后一次循环内存吃紧在不停GC

![1608442455089_1.gif](https://p9-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/bcb9865c5e86443991b36d684dcb3254~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=200&h=133&s=265743&e=gif&f=20&b=585048)

经过两天痛苦的排查，内存的泄露点竟然是JDK本K！~~这我还写个毛啊~~

好在，还是有解决办法的：

解决问题4+1：内存泄漏

1. 内存泄露最快的地方就是`com.sun.tools.javac.code.Types.MembersClosureCache#nilScope`，很明显，是由于`com.sun.tools.javac.code.Types#newRound` 只清理了`com.sun.tools.javac.code.Types.MembersClosureCache#_map`没有清理`com.sun.tools.javac.code.Types.MembersClosureCache#nilScope`（这个字段名单独写在了后面没有跟其他字段放在一起所以看漏了？），老样子，还是用反射解决：

```
private static final Field MEMBERS_CACHE;
private static final Field NIL_SCOPE;

static {
    try {
        // 运行时反射需要添加 --add-opens=jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED
        MEMBERS_CACHE = Types.class.getDeclaredField("membersCache");
        MEMBERS_CACHE.setAccessible(true);
        Class<?> aClass = Class.forName("com.sun.tools.javac.code.Types$MembersClosureCache");
        NIL_SCOPE = aClass.getDeclaredField("nilScope");
        NIL_SCOPE.setAccessible(true);
    } catch (Exception e) {
```

```
        throw new RuntimeException(e);
    }
}
/**
 * JDK bug, {@link Types#newRound} 在清除缓存时，没有清除{@link
Types.MembersClosureCache#nilScope}，会导致大量的内存泄露
 */
@SneakyThrows
@SuppressWarnings("JavadocReference")
public static void clear(Types types) {
    NIL_SCOPE.set(MEMBERS_CACHE.get(types), null);
}
```

在之前的`Types.instance`之后调用它:

```
Types types = Types.instance(((JavacTaskImpl) t).getContext());
clear(types);
```

2. 其次就是一些内存泄露缓慢，但是分布广泛的地方：

```
private static final Field CLASSES;
private static final Field SUB_SCOPES;
private static final Field LISTENERS;
private static final Field LIST_LISTENERS;


static {
// 运行时反射需要添加 --add-
opens=jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED
    try {
        LISTENERS = Scope.class.getDeclaredField("listeners");
        LISTENERS.setAccessible(true);
    } catch (NoSuchFieldException e) {
        throw new RuntimeException(e);
    }

    try {
        LIST_LISTENERS =
```

```
Scope.ScopeListenerList.class.getDeclaredField("listeners");
        LIST_LISTENERS.setAccessible(true);
    } catch (NoSuchFieldException e) {
        throw new RuntimeException(e);
    }

    try {
        CLASSES = Symtab.class.getDeclaredField("classes");
        CLASSES.setAccessible(true);
    } catch (NoSuchFieldException e) {
        throw new RuntimeException(e);
    }

    try {
        SUB_SCOPES =
Scope.CompoundScope.class.getDeclaredField("subScopes");
        SUB_SCOPES.setAccessible(true);
    } catch (NoSuchFieldException e) {
        throw new RuntimeException(e);
    }

}

@SneakyThrows
@SuppressWarnings({"unchecked", "rawtypes"})
public static void clear(Symtab symtab) {
    Map<Name, Map<Symbol.ModuleSymbol, Symbol.ClassSymbol>>
classes = (Map) CLASSES.get(symtab);
    if (classes == null) {
        return;
    }

    classes.values().parallelStream()

methodVisitor.visitFrame(Opcodes.F_APPEND, 1, new Object[]{"java/util/
Iterator"}, 0, null);
                                        methodVisitor.visitVarInsn(ALOAD, 4);
methodVisitor.visitMethodInsn(INVOKEINTERFACE, "java/util/Iterator", "
hasNext", "()Z", true);
                                        Label label2 = new Label();
                                        methodVisitor.visitJumpInsn(IFEQ, label2);
                                        methodVisitor.visitVarInsn(ALOAD, 4);
methodVisitor.visitMethodInsn(INVOKEINTERFACE, "java/util/Iterator", "
next", "()Ljava/lang/Object;", true);
                                        methodVisitor.visitTypeInsn(CHECKCAST,
"com/sun/tools/javac/code/Scope$ScopeListener");
                                        methodVisitor.visitVarInsn(ASTORE, 5);
                                        Label label3 = new Label();
```

```
                                        methodVisitor.visitLabel(label3);
                                        methodVisitor.visitLineNumber(217,
label3);
                                        methodVisitor.visitVarInsn(ILOAD, 3);
                                        Label label4 = new Label();
                                        methodVisitor.visitJumpInsn(IFEQ, label4);
                                        Label label5 = new Label();
                                        methodVisitor.visitLabel(label5);
                                        methodVisitor.visitLineNumber(218,
label5);
                                        methodVisitor.visitVarInsn(ALOAD, 5);
                                        methodVisitor.visitVarInsn(ALOAD, 1);
                                        methodVisitor.visitVarInsn(ALOAD, 2);
methodVisitor.visitMethodInsn(INVOKEINTERFACE, "com/sun/tools/java
c/code/Scope$ScopeListener", "symbolRemoved",
"(Lcom/sun/tools/javac/code/Symbol;Lcom/sun/tools/javac/code/Scop
e;)V", true);
                                        Label label6 = new Label();
                                        methodVisitor.visitJumpInsn(GOTO,
label6);
                                        methodVisitor.visitLabel(label4);
                                        methodVisitor.visitLineNumber(220,
label4);
methodVisitor.visitFrame(Opcodes.F_APPEND, 1, new Object[]{"com/sun
/tools/javac/code/Scope$ScopeListener"}, 0, null);
                                        methodVisitor.visitVarInsn(ALOAD, 5);
                                        methodVisitor.visitVarInsn(ALOAD, 1);
                                        methodVisitor.visitVarInsn(ALOAD, 2);
methodVisitor.visitMethodInsn(INVOKEINTERFACE, "com/sun/tools/java
c/code/Scope$ScopeListener", "symbolAdded",
"(Lcom/sun/tools/javac/code/Symbol;Lcom/sun/tools/javac/code/Scop
e;)V", true);
                                        methodVisitor.visitLabel(label6);
                                        methodVisitor.visitLineNumber(222,
label6);
methodVisitor.visitFrame(Opcodes.F_CHOP, 1, null, 0, null);
                        methodVisitor.visitJumpInsn(GOTO, label1);
                                        methodVisitor.visitLabel(label2);
                                        methodVisitor.visitLineNumber(223,
label2);
methodVisitor.visitFrame(Opcodes.F_CHOP, 1, null, 0, null);
                        methodVisitor.visitInsn(RETURN);
                                        methodVisitor.visitMaxs(3, 6);
                                        methodVisitor.visitEnd();
                                    }
                                    classWriter.visitEnd();
                                }
                            };
```

```
                        }
                    }))
                .installOn(inst);
    }
}

```

使用 agent 时带上`=debug,outputDir=E:\TEMP\agent` 即可观察修改后的class

清理编译后残余

如果重复编译一个文件，残余清不清理都是无所谓的，不过显然这在生产中并不可能，所以我们需要清理动态编译后存储在编译任务上下文中的对象：

```
public static void main(String[] args) throws URISyntaxException {
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    JavacTaskPool javacTaskPool = new JavacTaskPool(1);
    while (true) {
        DiagnosticCollector<JavaFileObject> diagnostics = new
DiagnosticCollector<>();
        StandardJavaFileManager fileManager =
compiler.getStandardFileManager(diagnostics, null, null);
        MemoryFileManager memoryFileManager = new
MemoryFileManager(fileManager);

        List<MemoryOutputJavaFileObject> result =
javacTaskPool.getTask(null, memoryFileManager, diagnostics,
                List.of("-source", "17", "-target", "17", "-encoding", "UTF-
8", "-proc:none"), null
                , List.of(new
MemoryInputJavaFileObject("LambdaContainer.java", source)), t -> {
                    Context ctx = ((JavacTaskImpl) t).getContext();
                    Types types = Types.instance(ctx);
                    clear(types); // 如果用 agent 或者JDK修改了实现 就不用调用
这个方法了
                    // 这个cache 会导致速度大量下降，所以禁用
                    //noinspection rawtypes,unchecked
                    types.candidatesCache.cache = new HashMap() {
                        @Override
                        public Object put(Object key, Object value) {
```

```
                    return null;
                }
            };
            try {
                if (Boolean.TRUE.equals(t.call())) {
                    return memoryFileManager.getOutputs();
                }
                return Collections.emptyList();
            } finally {
                //附加清理：清除已编译的软件包：
                Symtab symtab = Symtab.instance(ctx);
                Names names = Names.instance(ctx);
                Symbol.ModuleSymbol module = symtab.java_base ==
symtab.noModule ? symtab.noModule
                        : symtab.unnamedModule;
                Symbol.Completer completer =
ClassFinder.instance(ctx).getCompleter();
                List<MemoryOutputJavaFileObject> outputs =
memoryFileManager.getOutputs();
                for (MemoryOutputJavaFileObject output : outputs) {
                    String binaryName = output.getBinaryName();
                    Symbol.ClassSymbol aClass =
symtab.getClass(module, names.fromString(binaryName));
                    if (aClass != null) {
                        for (Symbol.ClassSymbol value : remove(symtab,
aClass.flatName()).values()) {
                            value.packge().members_field = null;
                            value.packge().completer = completer;
                        }
                    }
                }
                // 清理 Scope 中可能的未清理资源
                clear(symtab);  // 如果用 agent 或者JDK修改了实现 就不
用调用这个方法了
            }
        });
    assert !result.isEmpty();
  }
}

```


```
@NonNull
@SneakyThrows
@SuppressWarnings({"unchecked", "rawtypes"})
public static Map<Symbol.ModuleSymbol, Symbol.ClassSymbol>
```

```
remove(Symtab symtab, Name flatName) {
    Map<Name, Map<Symbol.ModuleSymbol, Symbol.ClassSymbol>>
classes = (Map) CLASSES.get(symtab);
    if (classes == null) {
        return Collections.emptyMap();
    }
    return Objects.requireNonNullElse(classes.remove(flatName),
Collections.emptyMap());
}
```

至此，优化 JavaCompiler API 就结束了（`JavaFileManager`读取嵌套jar的部分就不赘述，已经放够多代码了，有兴趣的可以去文末的代码仓库，或者查找Drools的实现以及其他大佬的文章）

优化结果：

![74933eb70b078871999c64ef8446b6e.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/e958578d1abf435f9d76333caf3918f0~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1345&h=404&s=58213&e=png&b=1f2023)
费这么一通功夫优化到了最后，JDK JavaCompiler API 终于又是世界最快的JAVA内存编译方法了

代码已发布在 [github: fast-compiler](http://cxyroad.com/”https://github.com/muyuanjin/fast-compiler”)

原文链接: https://juejin.cn/post/7364409181053321242