单元测试

Engine 类：

1.测试 findTargetNode 方法

对：public static Node findTargetNode(SubsetParams subsetParams, List<Node> allNodes)
进行单元测试
这个方法的作用是根据给定的目标函数名称，在节点列表中查找匹配的节点。如果找到多个或者没有找到，都会抛出异常，否则返回唯一的节点。
测试用例应包括：

- 通过 token 匹配到唯一节点。
- 通过 token_with_ownership 匹配到唯一节点。
- 通过 name 匹配到唯一节点。
- 没有节点匹配时抛出异常。

在 engineTest 测试类中用如下方法进行测试：

```
// 测试通过 token 匹配单个节点
@Test
void whenSingleMatchByToken_returnsNode() {
    Node targetNode = createNode("targetFunc", "TestGroup", "TestFile");
    SubsetParams params = new SubsetParams("targetFunc", 0, 0);
    List<Node> nodes = Collections.singletonList(targetNode);

    Node result = engine.findTargetNode(params, nodes);
    assertEquals(targetNode, result);
}

// 测试通过 token_with_ownership 匹配
@Test
void whenSingleMatchByTokenWithOwnership_returnsNode() {
    Node targetNode = createNodeWithOwnership("method", "TestClass", "TestFile");
    SubsetParams params = new SubsetParams("TestClass.method", 0, 0);

    Node result = engine.findTargetNode(params, Collections.singletonList(targetNode));
    assertEquals(targetNode, result);
}

// 测试无匹配节点时抛出异常
@Test
```

```
void whenNoMatchingNode_throwsException() {
    SubsetParams params = new SubsetParams("nonexistent", 0, 0);
    List<Node> nodes = Arrays.asList(
            createNode("func1", "Group1", "File1"),
            createNode("func2", "Group2", "File2")
    );

    Exception e = assertThrows(IllegalArgumentException.class, () ->
            engine.findTargetNode(params, nodes));
    assertTrue(e.getMessage().contains("Could not find node 'nonexistent'"));
}

// 测试通过 name 匹配
@Test
void whenSingleMatchByName_returnsNode() {
    Node targetNode = createNodeWithOwnership("method", "TestClass", "TestFile");
    String expectedName = "TestFile::TestClass.method";
    SubsetParams params = new SubsetParams(expectedName, 0, 0);

    Node result = engine.findTargetNode(params, Collections.singletonList(targetNode));
    assertEquals(targetNode, result);
}
```

所有测试方法均通过

2. 测试 findLinks 方法：

对 public static List<CallLinker.FindLinkResult> findLinks(Node nodeA, List<Node> allNodes) 进行单元测试

findLinks 方法接收一个 Node 对象 nodeA 和一个 Node 列表 allNodes。它遍历 nodeA 的所有调用，对每个调用使用 CallLinker.findLinkForCall 方法进行处理，然后过滤掉那些 matchedNode 和 unresolvedCall 都为 null 的结果，最后将结果收集到列表中返回。

测试需要覆盖以下几个关键点：

- 正常匹配：当调用能找到对应的节点时，返回包含 matchedNode 的结果。
- 未解析的调用：当调用无法解析时，返回包含 unresolvedCall 的结果。
- 过滤无效结果：确保结果为 null 的条目被过滤掉。
- 多调用处理：验证方法能正确处理多个调用的情况。

```
// 测试正常匹配场景
@Test
void whenCallsHaveMatchedNodes_returnsValidResults() {
    // 准备测试数据
```

```
        Node nodeA = createTestNode("nodeA");
        Node matchedNode = createTestNode("targetNode");

        List<Call> calls = new ArrayList<>();
        calls.add(createTestCall("validCall1", true, false));
        calls.add(createTestCall("validCall2", true, false));
        nodeA.calls = calls;

        List<Node> allNodes = List.of(nodeA, matchedNode);

        // 执行方法
        List<CallLinker.FindLinkResult> results = engine.findLinks(nodeA, allNodes);

        // 验证结果
        assertEquals(2, results.size());
        assertNotNull(results.get(0).matchedNode);
        assertNotNull(results.get(1).matchedNode);
    }

    // 测试未解析调用场景
    @Test
    void whenCallsUnresolved_returnsUnresolvedResults() {
        Node nodeA = createTestNode("nodeA");
        List<Call> calls = new ArrayList<>();
        calls.add(createTestCall("unresolved1", false, true));
        calls.add(createTestCall("unresolved2", false, true));
        nodeA.calls = calls;

        List<CallLinker.FindLinkResult> results = engine.findLinks(nodeA, new ArrayList<>());

        assertEquals(2, results.size());
        assertNotNull(results.get(0).unresolvedCall);
        assertNotNull(results.get(1).unresolvedCall);
    }

    // 测试混合匹配场景
    @Test
    void whenMixedCalls_returnsFilteredResults() {
        Node nodeA = createTestNode("nodeA");
        Node matchedNode = createTestNode("targetNode");

        List<Call> calls = new ArrayList<>();
        calls.add(createTestCall("validCall", true, false));       // 有效匹配
        calls.add(createTestCall("unresolved", false, true));     // 未解析
```

```
        calls.add(createTestCall("invalid", false, false));        // 应被过滤
        nodeA.calls = calls;

        List<Node> allNodes = List.of(matchedNode);

        List<CallLinker.FindLinkResult> results = engine.findLinks(nodeA, allNodes);

        assertEquals(2, results.size());
        assertTrue(results.stream().anyMatch(r -> r.matchedNode != null));
        assertTrue(results.stream().anyMatch(r -> r.unresolvedCall != null));
}

// 测试空调用列表
@Test
void whenNoCalls_returnsEmptyList() {
        Node nodeA = createTestNode("nodeA");
        nodeA.calls = new ArrayList<>();

        List<CallLinker.FindLinkResult> results = engine.findLinks(nodeA, new ArrayList<>());
        assertTrue(results.isEmpty());
}
```

所有测试方法均通过

3. filterNodesForSubset 方法

对 public static Set<Node> filterNodesForSubset(SubsetParams subsetParams, List<Node> allNodes, List<Edge> edges)进行单元测试
这个方法的目的是根据给定的子集参数、节点列表和边列表，过滤出符合条件的节点集合。
测试场景包括：

- 目标节点存在且深度为 0：应该只包含目标节点自己。
- 下游深度为 1：包含目标节点及其直接下游节点。
- 上游深度为 2：包含目标节点及其直接上游节点。
- 深度为无限（-1） ：包含所有可达节点。

```
// 测试空边集
@Test
void whenEmptyEdges_returnsOnlyTarget() {
        Node target = createNode("A");
        List<Node> nodes = Collections.singletonList(target);
        List<Edge> edges = Collections.emptyList();
        SubsetParams params = new SubsetParams("A", 0, 0);
```

```
        Set<Node> result = engine.filterNodesForSubset(params, nodes, edges);

        assertEquals(1, result.size());
        assertTrue(result.contains(target));
    }

    // 测试下游深度 1
    @Test
    void downstreamDepthOne_returnsDirectChildren() {
        Node a = createNode("A");
        Node b = createNode("B");
        Node c = createNode("C");
        List<Node> nodes = Arrays.asList(a, b, c);
        List<Edge> edges = Arrays.asList(
                createEdge(a, b),
                createEdge(b, c)
        );
        SubsetParams params = new SubsetParams("A", 0, 1);

        Set<Node> result = engine.filterNodesForSubset(params, nodes, edges);

        assertEquals(2, result.size());
        assertTrue(result.containsAll(Arrays.asList(a, b)));
    }

    // 测试上游深度 2
    @Test
    void upstreamDepthTwo_returnsMultiLevelParents() {
        Node a = createNode("A");
        Node b = createNode("B");
        Node c = createNode("C");
        Node d = createNode("D");
        List<Node> nodes = Arrays.asList(a, b, c, d);
        List<Edge> edges = Arrays.asList(
                createEdge(b, a),
                createEdge(c, b),
                createEdge(d, c)
        );
        SubsetParams params = new SubsetParams("A", 2, 0);

        Set<Node> result = engine.filterNodesForSubset(params, nodes, edges);

        assertEquals(3, result.size());
```

```
        assertTrue(result.containsAll(Arrays.asList(a, b, c)));
}

// 测试无限深度
@Test
void infiniteDepth_returnsAllConnectedNodes() {
        Node a = createNode("A");
        Node b = createNode("B");
        Node c = createNode("C");
        List<Node> nodes = Arrays.asList(a, b, c);
        List<Edge> edges = Arrays.asList(
                        createEdge(a, b),
                        createEdge(b, c)
        );
        SubsetParams params = new SubsetParams("A", -1, -1);

        Set<Node> result = engine.filterNodesForSubset(params, nodes, edges);

        assertEquals(3, result.size());
        assertTrue(result.containsAll(nodes));
}
```

所有测试方法均通过

4. filterEdgesForSubset 方法

对 public static List<Edge> filterEdgesForSubset(Set<Node> newNodes, List<Edge> edges)做单元测试
接收一个 Node 的集合 newNodes 和一个 Edge 的列表 edges，返回过滤后的 Edge 列表。过滤条件是只有那些两个节点都在 newNodes 中的边才会被保留下来。

测试用例包括：

- 正常情况：边两端的节点都在 newNodes 中，应该保留这条边。
- 一个节点不在 newNodes 中：边的 node0 或 node1 不在集合中，应该过滤掉这条边。
- 两个节点都不在 newNodes 中：当然这种情况边也会被过滤掉。
- 空边列表：输入的 edges 为空，应该返回空列表。

```
// 测试用例 1: 正常情况，边两端节点都在集合中
@Test
void keepEdgeWhenBothNodesPresent() {
        Node a = new Node("A");
        Node b = new Node("B");
```

```
        Set<Node> nodes = new HashSet<>(Arrays.asList(a, b));
        Edge edge = new Edge(a, b);

        List<Edge> result = MyClass.filterEdgesForSubset(nodes, Arrays.asList(edge));

        assertEquals(1, result.size());
        assertSame(edge, result.get(0));
}

// 测试用例 2: 一个节点不存在
@Test
void filterEdgeWhenOneNodeMissing() {
        Node a = new Node("A");
        Node b = new Node("B");
        Set<Node> nodes = new HashSet<>(Collections.singletonList(a));
        Edge edge = new Edge(a, b);

        List<Edge> result = MyClass.filterEdgesForSubset(nodes, Arrays.asList(edge));

        assertTrue(result.isEmpty());
}

// 测试用例 3: 两个节点都不存在
@Test
void filterEdgeWhenBothNodesMissing() {
        Node a = new Node("A");
        Node b = new Node("B");
        Set<Node> nodes = new HashSet<>(Arrays.asList(new Node("C"), new Node("D")));
        Edge edge = new Edge(a, b);

        List<Edge> result = MyClass.filterEdgesForSubset(nodes, Arrays.asList(edge));

        assertTrue(result.isEmpty());
}

// 测试用例 4: 空边列表输入
@Test
void handleEmptyEdgesList() {
        Set<Node> nodes = new HashSet<>(Arrays.asList(new Node("A")));
        List<Edge> result = MyClass.filterEdgesForSubset(nodes, Collections.emptyList());

        assertTrue(result.isEmpty());
}
```

以上测试方法均通过

5. filterGroupsForSubset 方法

对 public static List<Group> filterGroupsForSubset(Set<Node> newNodes, List<Group> fileGroups) 做单元测试
方法的作用是过滤文件组，移除那些不在 newNodes 集合中的节点，并清理掉所有空的子组。整个过程分为三步：第一步是移除不需要的节点，第二步是过滤掉空的文件组，第三步是深度清理空的子组。

测试场景：

- 移除未包含的节点：当一个组中的某些节点不在 newNodes 中时，这些节点应该被移除。
- 过滤空的文件组：如果某个文件组在移除节点后变成空，应该被过滤掉。
- 深度清理空子组：当子组为空时，应该被移除，包括多层嵌套的情况。

```
// 测试用例 1: 基本节点过滤
@Test
void shouldRemoveUnwantedNodes() {
    // 准备
    Set<Node> validNodes = Collections.singleton(keptNode);

    // 执行
    List<Group> result = engine.filterGroupsForSubset(validNodes, List.of(rootGroup));

    // 验证节点移除
    Group remainingSubgroup = result.get(0).subgroups.get(0);
    assertEquals(1, remainingSubgroup.nodes.size(), "应只保留 1 个节点");
    assertEquals(keptNode, remainingSubgroup.nodes.get(0), "保留的节点应为 keptNode");
}

// 测试用例 2: 空组过滤
@Test
void shouldFilterEmptyGroups() {
    // 准备
    Set<Node> validNodes = Collections.singleton(keptNode);

    // 执行
    List<Group> result = engine.filterGroupsForSubset(validNodes, List.of(rootGroup));

    // 验证组结构
    assertEquals(1, result.size(), "应保留根组");
    assertEquals(1, result.get(0).subgroups.size(), "应保留 subgroup1");
```

```
}

// 测试用例 3: 深度清理空子组
@Test
void shouldPruneNestedEmptyGroups() {
    // 执行
    Set<Node> validNodes = Collections.singleton(keptNode);
    List<Group> result = engine.filterGroupsForSubset(validNodes, List.of(rootGroup));

    // 验证子组结构
    Group remainingGroup = result.get(0);
    assertEquals(1, remainingGroup.subgroups.size(), "应保留 1 个子组");
    assertEquals(0, remainingGroup.subgroups.get(0).subgroups.size(), "子组不应包含嵌套空组");
}
```

以上测试方法均通过

6.isAnyParentIncluded 方法

对 private static boolean isAnyParentIncluded(Group subgroup, List<String> includeList)进行单元测试
方法的作用是检查给定的 subgroup 或其任何父组是否存在于 includeList 中。如果存在，返回 true，否则返回 false。另外，如果 includeList 为空，直接返回 true。

测试场景包括：

- includeList 为空的情况：应该返回 true。
- 当前 Group 的 token 在 includeList 中：直接返回 true。
- 父 Group 的 token 在 includeList 中：需要确保能正确遍历父链并找到。
- 没有任何 Group 的 token 在 includeList 中：返回 false。

```
// 测试用例 1: includeList 为空
@Test
void shouldReturnTrue_WhenIncludeListIsEmpty() {
    Group subgroup = new Group("test", null);
    assertTrue(engine.isAnyParentIncluded(subgroup, Collections.emptyList()));
```

```
}

// 测试用例 2: 当前组本身在 includeList 中
@Test
void shouldReturnTrue_WhenCurrentGroupIncluded() {
    Group subgroup = new Group("target", null);
    assertTrue(engine.isAnyParentIncluded(subgroup, List.of("target")));
}

// 测试用例 3: 直接父组在 includeList 中
@Test
void shouldReturnTrue_WhenDirectParentIncluded() {
    Group parent = new Group("parent", null);
    Group subgroup = new Group("child", parent);
    assertTrue(engine.isAnyParentIncluded(subgroup, List.of("parent")));
}

// 测试用例 4: 祖父组在 includeList 中
@Test
void shouldReturnTrue_WhenGrandparentIncluded() {
    Group hierarchy = createGroupHierarchy();
    assertTrue(engine.isAnyParentIncluded(hierarchy, List.of("grandparent")));
}
```

Group 类

1. Group 类的构造函数

测试场景：

正常情况下的参数初始化：当所有参数都有效时，检查成员变量是否正确赋值。

- import_tokens 为 null：此时应该初始化一个空的 ArrayList。
- inherits 为 null：同样，应该初始化为空的 ArrayList。
- group_type 无效：当传入的 group_type 不在 Model.GROUP_TYPE 的 values 中时，应 AssertionError。
- uid 的生成：验证 uid 是否以"cluster_"开头，并且符合预期的格式（比如长度是否正确）。
- 默认值的初始化：比如 nodes、subgroups 是否初始化为空列表，root_node 是否为 null。

```
// 测试 import_tokens 为 null 的情况
@Test
```

```
void shouldHandleNullImportTokens() {
    Group group = new Group("Test", GROUP_TYPE.get("CLASS"), "Class", null, null, null, null);

    assertNotNull(group.import_tokens);
    assertTrue(group.import_tokens.isEmpty());
}

// 测试 inherits 为 null 的情况
@Test
void shouldHandleNullInherits() {
    Group group = new Group("Test", GROUP_TYPE.get("NAMESPACE"), "Namespace", List.of(),
10, null, null);

    assertNotNull(group.inherits);
    assertTrue(((List<?>)group.inherits).isEmpty());
}

// 测试无效 group_type 的情况
@Test
void shouldThrowForInvalidGroupType() {
    String invalidType = "INVALID_TYPE";

    AssertionError error = assertThrows(AssertionError.class, () ->
            new Group("Test", invalidType, "Invalid", null, null, null, null)
    );

    assertEquals("group_type must be one of GROUP_TYPE values", error.getMessage());
}

// 参数化测试：验证不同 group_type 值
@ParameterizedTest
@MethodSource("validGroupTypes")
void shouldAcceptAllValidGroupTypes(String typeKey, String expectedType) {
    Group group = new Group("Test", expectedType, "Display", null, null, null, null);

    assertEquals(expectedType, group.group_type);
}

private static List<Arguments> validGroupTypes() {
    return Arrays.asList(
            Arguments.of("FILE", GROUP_TYPE.get("FILE")),
            Arguments.of("CLASS", GROUP_TYPE.get("CLASS")),
            Arguments.of("NAMESPACE", GROUP_TYPE.get("NAMESPACE"))
    );
```

```
}

// 测试 UID 生成唯一性（概率性测试）
@Test
void shouldGenerateUniqueUid() {
    Group group1 = new Group("A", GROUP_TYPE.get("FILE"), "File", null, 1, null, null);
    Group group2 = new Group("B", GROUP_TYPE.get("FILE"), "File", null, 2, null, null);

    assertNotEquals(group1.uid, group2.uid);
}

// 测试父组关联
@Test
void shouldSetParentRelationship() {
    Group parent = new Group(GROUP_TYPE.get("NAMESPACE"), "Parent", "Namespace", null,
0, null, null);
    Group child = new Group("Child", GROUP_TYPE.get("CLASS"), "Class", null, 5, parent, null);

    assertSame(parent, child.parent);
    assertTrue(parent.subgroups.contains(child));
}

// 测试行号处理
@Test
void shouldHandleNullLineNumber() {
    Group group = new Group("Test", GROUP_TYPE.get("FILE"), "File", null, null, null, null);

    assertNull(group.line_number);
}

// 测试显示类型默认值
@Test
void shouldUseSensibleDisplayType() {
    Group fileGroup = new Group("Test", GROUP_TYPE.get("FILE"), null, null, null, null, null);
    Group classGroup = new Group("Test", GROUP_TYPE.get("CLASS"), null, null, null, null, null);

    assertEquals("File", fileGroup.display_type);
    assertEquals("Class", classGroup.display_type);
}
```

构造函数通过单元测试

2. get_variables 方法

对 public List<Variable> get_variables(Integer line_number)进行单元测试
测试场景：

当 root_node 为 null 时：

- 此时应该返回空列表。这个场景比较简单，只需确保当没有根节点时方法返回空。

当 root_node 存在时：

- 子组和非根节点的变量是否正确添加：需要构造一个包含根节点、子组和非根节点的 Group 对象，验证返回的变量列表是否包含所有这些元素。
- 变量是否按行号降序排序：构造多个具有不同行号的变量，检查排序是否正确。
- 处理 null 行号的情况：当某些变量的行号为 null 时，排序应该如何处理，是否会影响其他变量的顺序。

```
@Test
void testGetVariablesWhenRootNodeIsNull() {
    group.root_node = null;
    List<Variable> result = group.get_variables(10);
    assertTrue(result.isEmpty());
}

@Test
void testGetVariablesWithSubgroupsAndNonRootNodes() {
    try (MockedStatic<Model> model = mockStatic(Model.class)) {
        // 模拟静态方法返回预设变量
        model.when(() -> Model._wrap_as_variables(any()))
                .thenAnswer(inv -> {
                    Object arg = inv.getArgument(0);
                    if (arg instanceof List && ((List<?>) arg).get(0) instanceof Group) {
                        return List.of(subgroupVar);
                    } else {
                        return List.of(nonRootVar);
                    }
                });

        List<Variable> variables = group.get_variables(10);

        assertEquals(3, variables.size());
        assertTrue(variables.containsAll(Arrays.asList(rootVar, subgroupVar, nonRootVar)));
    }
}
```

```
@Test
void testVariablesSortingByLineNumberDescending() {
    // 添加不同行号的变量
    Variable var1 = new Variable("var1", null, 30);
    Variable var2 = new Variable("var2", null, 10);
    rootNode.variables.addAll(Arrays.asList(var1, var2));

    List<Variable> variables = group.get_variables(10);

    assertEquals(30, variables.get(0).line_number);
    assertEquals(25, variables.get(1).line_number); // nonRootVar
    assertEquals(20, variables.get(2).line_number); // rootVar
    assertEquals(15, variables.get(3).line_number); // subgroupVar
    assertEquals(10, variables.get(4).line_number);
}

@Test
void testVariablesWithNullLineNumbers() {
    // 创建无行号的变量
    Variable noLineVar1 = new Variable("noLine1", null, null);
    Variable noLineVar2 = new Variable("noLine2", null, null);
    rootNode.variables.addAll(Arrays.asList(noLineVar1, noLineVar2));

    List<Variable> variables = group.get_variables(10);

    // 验证原始顺序保持（无行号的变量不参与排序）
    int rootVarIndex = variables.indexOf(rootVar);
    assertTrue(variables.indexOf(noLineVar1) > rootVarIndex);
    assertTrue(variables.indexOf(noLineVar2) > rootVarIndex);
}
```

以上测试方法均通过测试

Call 类

1. Call 类的构造函数

用例包括：
所有参数为非 null 且有效的情况。
owner_token 为 null 的情况。

line_number 为 null 的情况。
token 为 null 的情况（如果允许的话）。
definite_constructor 分别为 true 和 false 的情况。

```
// 正常参数测试
@Test
void shouldInitializeAllFieldsWithValidParams() {
    Call call = new Call("createUser", 42, "UserService", true);

    assertEquals("createUser", call.token);
    assertEquals(42, call.line_number);
    assertEquals("UserService", call.owner_token);
    assertTrue(call.definite_constructor);
}

// 参数化测试：null 值处理
@ParameterizedTest
@MethodSource("nullCasesProvider")
void shouldHandleNullValues(String token, Integer lineNumber, String ownerToken) {
    Call call = new Call(token, lineNumber, ownerToken, false);

    assertNull(call.token);
    assertNull(call.line_number);
    assertNull(call.owner_token);
    assertFalse(call.definite_constructor);
}
```


```
// 正常参数测试
@Test
void shouldInitializeAllFieldsWithValidParams() {
    Call call = new Call("createUser", 42, "UserService", true);

    assertEquals("createUser", call.token);
    assertEquals(42, call.line_number);
    assertEquals("UserService", call.owner_token);
    assertTrue(call.definite_constructor);
}

// 参数化测试：null 值处理
@ParameterizedTest
@MethodSource("nullCasesProvider")
void shouldHandleNullValues(String token, Integer lineNumber, String ownerToken) {
    Call call = new Call(token, lineNumber, ownerToken, false);
```

```
        assertNull(call.token);
        assertNull(call.line_number);
        assertNull(call.owner_token);
        assertFalse(call.definite_constructor);
}

private static Stream<Arguments> nullCasesProvider() {
        return Stream.of(
                Arguments.of(null, null, null),
                Arguments.of(null, 10, "owner"),
                Arguments.of("method", null, null),
                Arguments.of("test", 20, null)
        );
}
```

构造函数通过单元测试

Node 类

1. Node 类构造函数

测试用例：

传入所有参数，包括非空的 import_tokens。
验证各字段是否正确赋值。
检查 import_tokens 是否与传入的列表相同。
将 import_tokens 设为 null。
确认 Node 中的 import_tokens 字段被初始化为空的 ArrayList。
比如 line_number 为 null 的情况。这个字段被允许是 null 吗？看构造函数参数类型是 Integer，所以 null 是可以的。测试中要验证该字段是否正确保存传入的值。
传入 Group 实例作为 parent。
传入另一个 Node 实例作为 parent。
是否会影响 Node 中的 parent 字段。这里构造函数只是简单地将 parent 赋给实例变量，不处理其他逻辑，所以只要确保保存正确即可，可能这部分的测试属于构造函数参数正确性的范畴。

```
@Test
void constructor_WhenAllParametersValid_InitializesFieldsCorrectly() {
        // Arrange
        String token = "testToken";
        List<Call> calls = Arrays.asList(new Call("call1"), new Call("call2"));
```

```
        List<Variable> variables = Arrays.asList(new Variable("var1"), new Variable("var2"));
        Group parentGroup = new Group(Model.GROUP_TYPE.get("CLASS"), "parentGroup", null);
        List<String> importTokens = Arrays.asList("import1", "import2");
        Integer lineNumber = 10;
        boolean isConstructor = true;

        // Act
        Node node = new Node(token, calls, variables, parentGroup, importTokens, lineNumber,
isConstructor);

        // Assert
        assertEquals(token, node.token);
        assertEquals(calls, node.calls);
        assertEquals(variables, node.variables);
        assertEquals(parentGroup, node.parent);
        assertEquals(importTokens, node.import_tokens);
        assertEquals(lineNumber, node.line_number);
        assertEquals(isConstructor, node.is_constructor);
        assertTrue(node.is_leaf);
        assertTrue(node.is_trunk);
        assertNotNull(node.uid);
        assertTrue(node.uid.startsWith("node_"));
    }

    @Test
    void constructor_WhenImportTokensIsNull_InitializesWithEmptyList() {
        // Act
        Node node = new Node("token", null, null, null, null, null, false);

        // Assert
        assertNotNull(node.import_tokens);
        assertTrue(node.import_tokens.isEmpty());
    }

    @ParameterizedTest
    @NullSource
    void constructor_WhenLineNumberIsNull_StoresNull(Integer lineNumber) {
        // Act
        Node node = new Node("token", null, null, null, null, lineNumber, false);

        // Assert
        assertNull(node.line_number);
    }
```

```
@Test
void constructor_WhenCallsAndVariablesAreNull_InitializesWithNull() {
	// Act
	Node node = new Node("token", null, null, null, Collections.emptyList(), 0, false);

	// Assert
	assertNull(node.calls);
	assertNull(node.variables);
}
```

构造函数通过单元测试