MyBatis加载策略

1.1 什么是延迟加载?

问题

在开发过程中很多时候我们并不需要总是在加载用户信息时就一定要加载他的订单信息。此时就是我们所说的延迟加载。

举个栗子

```
* 在一对多中，当我们有一个用户，它有个100个订单
  在查询用户的时候，要不要把关联的订单查出来?
  在查询订单的时候，要不要把关联的用户查出来?

* 回答
  在查询用户时，用户下的订单应该是，什么时候用，什么时候查询。
  在查询订单时，订单所属的用户信息应该是随着订单一起查询出来。
```

延迟加载

就是在需要用到数据时才进行加载，不需要用到数据时就不加载数据。延迟加载也称懒加载。

```
* 优点:
  先从单表查询，需要时再从关联表去关联查询，大大提高数据库性能，因为查询单表要比关联查询多张表速度要快。

* 缺点:
  因为只有当需要用到数据时，才会进行数据库查询，这样在大批量数据查询时，因为查询工作也要消耗时间，所以可能造成用户等待时间变长，造成用户体验下降。

* 在多表中:
  一对多，多对多:通常情况下采用延迟加载
  一对一（多对一）:通常情况下采用立即加载

* 注意:
   延迟加载是基于嵌套查询来实现的
```

1.2 实现

1.2.1 局部延迟加载

在association和collection标签中都有一个fetchType属性，通过修改它的值，可以修改局部的加载策略。

```
<!-- 开启一对多 延迟加载 -->
```

```
<resultMap id="userMap" type="user">
    <id column="id" property="id"></id>
    <result column="username" property="username"></result>
    <result column="password" property="password"></result>
    <result column="birthday" property="birthday"></result>
    <!--
    fetchType="lazy" 懒加载策略
    fetchType="eager" 立即加载策略
  -->
    <collection property="orderList" ofType="order" column="id"
                select="com.lagou.dao.OrderMapper.findByUid" fetchType="lazy">
    </collection>
</resultMap>

<select id="findAll" resultMap="userMap">
    SELECT * FROM `user`
</select>
```

1.2.2 全局延迟加载

在Mybatis的核心配置文件中可以使用setting标签修改全局的加载策略。

```
<settings>
    <!--开启全局延迟加载功能-->
    <setting name="lazyLoadingEnabled" value="true"/>
</settings>
```

注意

局部的加载策略优先级高于全局的加载策略。

```
<!-- 关闭一对一 延迟加载 -->
<resultMap id="orderMap" type="order">
    <id column="id" property="id"></id>
    <result column="ordertime" property="ordertime"></result>
    <result column="total" property="total"></result>
    <!--
    fetchType="lazy" 懒加载策略
    fetchType="eager" 立即加载策略
  -->
    <association property="user" column="uid" javaType="user"
                 select="com.lagou.dao.UserMapper.findById" fetchType="eager">
</association>
</resultMap>

<select id="findAll" resultMap="orderMap">
    SELECT * from orders
```

```
</select>
```

1.2.3 设置触发延迟加载的方法

大家在配置了延迟加载策略后，发现即使没有调用关联对象的任何方法，但是在你调用当前对象的equals、clone、hashCode、toString方法时也会触发关联对象的查询。

我们可以在配置文件中使用lazyLoadTriggerMethods配置项覆盖掉上面四个方法。

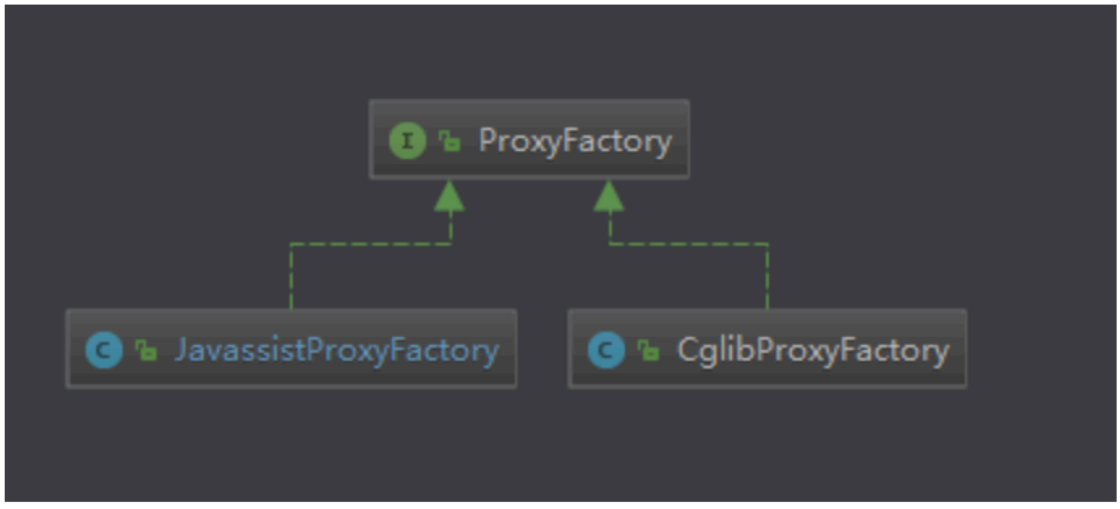
```
<settings>
    <!--开启全局延迟加载功能-->
    <setting name="lazyLoadingEnabled" value="true"/>
    <!--所有方法都会延迟加载-->
    <setting name="lazyLoadTriggerMethods" value=""/>
</settings>
```

1.3 延迟加载原理（源码剖析）

原理其实特别简单，就是在分析User的成员变量的时候，发现如果有懒加载的配置,如：fetchType="lazy"，则把User转化成代理类返回。并把懒加载相关对象放到ResultLoaderMap中存起来。当调用相应User对象里面的get方法获取懒加载变量的时候则根据代理类去执行sql获取

总结：延迟加载主要是通过动态代理的形式实现，通过代理拦截到指定方法，执行数据加载。

MyBatis延迟加载主要使用：Javassist，Cglib实现，类图展示：




```
public class Configuration {
/** aggressiveLazyLoading:
   * 当开启时，任何方法的调用都会加载该对象的所有属性。否则，每个属性会按需加载（参考lazyLoadTriggerMethods).
   * 默认为true
   * */
  protected boolean aggressiveLazyLoading;
```

```
  /**
   * 延迟加载触发方法
   */
  protected Set<String> lazyLoadTriggerMethods = new HashSet<String>
(Arrays.asList(new String[] { "equals", "clone", "hashCode", "toString" }));
  /** 是否开启延迟加载 */
  protected boolean lazyLoadingEnabled = false;

  /**
   * 默认使用Javassist代理工厂
   * @param proxyFactory
   */
  public void setProxyFactory(ProxyFactory proxyFactory) {
    if (proxyFactory == null) {
      proxyFactory = new JavassistProxyFactory();
    }
    this.proxyFactory = proxyFactory;
  }

  //省略...
}
```

查看源码

Mybatis的查询结果是由ResultSetHandler接口的handleResultSets()方法处理的。ResultSetHandler接口只有一个实现，DefaultResultSetHandler，接下来看下延迟加载相关的一个核心的方法

```
<code class="language-Java">//#mark 创建结果对象
  private Object createResultObject(ResultSetWrapper rsw, ResultMap resultMap,
ResultLoaderMap lazyLoader, String columnPrefix) throws SQLException {
    this.useConstructorMappings = false; // reset previous mapping result
    final List&lt;Class&lt;?&gt;&gt; constructorArgTypes = new
ArrayList&lt;Class&lt;?&gt;&gt;();
    final List&lt;Object&gt; constructorArgs = new ArrayList&lt;Object&gt;();
    //#mark 创建返回的结果映射的真实对象
    Object resultObject = createResultObject(rsw, resultMap,
constructorArgTypes, constructorArgs, columnPrefix);
    if (resultObject != null &amp;&amp; !hasTypeHandlerForResultObject(rsw,
resultMap.getType())) {
      final List&lt;ResultMapping&gt; propertyMappings =
resultMap.getPropertyResultMappings();
      for (ResultMapping propertyMapping : propertyMappings) {
        //  判断属性有没配置嵌套查询，如果有就创建代理对象
        if (propertyMapping.getNestedQueryId() != null &amp;&amp;
propertyMapping.isLazy()) {
          //#mark 创建延迟加载代理对象
```

```
          resultObject =
configuration.getProxyFactory().createProxy(resultObject, lazyLoader,
configuration, objectFactory, constructorArgTypes, constructorArgs);
          break;
        }
      }
    }
    this.useConstructorMappings = resultObject != null &amp;amp;&amp;amp;
!constructorArgTypes.isEmpty(); // set current mapping result
    return resultObject;
  }
```

默认采用javassistProxy进行代理对象的创建

```
protected ProxyFactory proxyFactory = new JavassistProxyFactory();
```

JavasisstProxyFactory实现

```
public class JavassistProxyFactory implements
org.apache.ibatis.executor.loader.ProxyFactory {


    /**
   * 接口实现
   * @param target 目标结果对象
   * @param lazyLoader 延迟加载对象
   * @param configuration 配置
   * @param objectFactory 对象工厂
   * @param constructorArgTypes 构造参数类型
   * @param constructorArgs 构造参数值
   * @return
   */
  @Override
  public Object createProxy(Object target, ResultLoaderMap lazyLoader,
Configuration configuration, ObjectFactory objectFactory, List&lt;Class&lt;?
&gt;&gt; constructorArgTypes, List&lt;Object&gt; constructorArgs) {
    return EnhancedResultObjectProxyImpl.createProxy(target, lazyLoader,
configuration, objectFactory, constructorArgTypes, constructorArgs);
  }

    //省略...

  /**
   * 代理对象实现，核心逻辑执行
   */
```

```
  private static class EnhancedResultObjectProxyImpl implements MethodHandler
{

    /**
   * 创建代理对象
   * @param type
   * @param callback
   * @param constructorArgTypes
   * @param constructorArgs
   * @return
   */
  static Object crateProxy(Class&lt;?&gt; type, MethodHandler callback,
List&lt;Class&lt;?&gt;&gt; constructorArgTypes, List&lt;Object&gt;
constructorArgs) {

    ProxyFactory enhancer = new ProxyFactory();
    enhancer.setSuperclass(type);

    try {
      //通过获取对象方法，判断是否存在该方法
      type.getDeclaredMethod(WRITE_REPLACE_METHOD);
      // ObjectOutputStream will call writeReplace of objects returned by
writeReplace
      if (log.isDebugEnabled()) {
        log.debug(WRITE_REPLACE_METHOD + &quot; method was found on bean
&quot; + type + &quot;, make sure it returns this&quot;);
      }
    } catch (NoSuchMethodException e) {
      //没找到该方法，实现接口
      enhancer.setInterfaces(new Class[]{WriteReplaceInterface.class});
    } catch (SecurityException e) {
      // nothing to do here
    }

    Object enhanced;
    Class&lt;?&gt;[] typesArray = constructorArgTypes.toArray(new
Class[constructorArgTypes.size()]);
    Object[] valuesArray = constructorArgs.toArray(new
Object[constructorArgs.size()]);
    try {
      //创建新的代理对象
      enhanced = enhancer.create(typesArray, valuesArray);
    } catch (Exception e) {
      throw new ExecutorException(&quot;Error creating lazy proxy.  Cause:
&quot; + e, e);
    }
    //设置代理执行器
    ((Proxy) enhanced).setHandler(callback);
    return enhanced;
```

```
  }


    /**
     * 代理对象执行
     * @param enhanced 原对象
     * @param method 原对象方法
     * @param methodProxy 代理方法
     * @param args 方法参数
     * @return
     * @throws Throwable
     */
    @Override
    public Object invoke(Object enhanced, Method method, Method methodProxy,
Object[] args) throws Throwable {
      final String methodName = method.getName();
      try {
        synchronized (lazyLoader) {
          if (WRITE_REPLACE_METHOD.equals(methodName)) {
            //忽略暂未找到具体作用
            Object original;
            if (constructorArgTypes.isEmpty()) {
              original = objectFactory.create(type);
            } else {
              original = objectFactory.create(type, constructorArgTypes,
constructorArgs);
            }
            PropertyCopier.copyBeanProperties(type, enhanced, original);
            if (lazyLoader.size() &gt; 0) {
              return new JavassistSerialStateHolder(original,
lazyLoader.getProperties(), objectFactory, constructorArgTypes,
constructorArgs);
            } else {
              return original;
            }
          } else {
            //延迟加载数量大于0
            if (lazyLoader.size() &gt; 0 &amp;&amp;
!FINALIZE_METHOD.equals(methodName)) {
              //aggressive 一次加载性所有需要要延迟加载属性或者包含触发延迟加载方法
              if (aggressive || lazyLoadTriggerMethods.contains(methodName)) {
                log.debug(&quot;==&gt; laze lod trigger method:&quot; +
methodName + &quot;,proxy method:&quot; + methodProxy.getName() + &quot;
class:&quot; + enhanced.getClass());
                //一次全部加载
                lazyLoader.loadAll();
              } else if (PropertyNamer.isSetter(methodName)) {
                //判断是否为set方法，set方法不需要延迟加载
```

```
                final String property =
PropertyNamer.methodToProperty(methodName);
                lazyLoader.remove(property);
              } else if (PropertyNamer.isGetter(methodName)) {
                final String property =
PropertyNamer.methodToProperty(methodName);
                if (lazyLoader.hasLoader(property)) {
                  //延迟加载单个属性
                  lazyLoader.load(property);
                  log.debug(&quot;load one :&quot; + methodName);
                }
              }
            }
          }
        }
        return methodProxy.invoke(enhanced, args);
      } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
      }
    }
  }
```

注意事项

1. IDEA调试问题 当配置aggressiveLazyLoading=true，在使用IDEA进行调试的时候，如果断点打到代理执行逻辑当中，你会发现延迟加载的代码永远都不能进入，总是会被提前执行。 主要产生的原因在aggressiveLazyLoading，因为在调试的时候，IDEA的Debuger窗体中已经触发了延迟加载对象的方法。

Mybatis动态sql源码剖析

执行原理为，使用OGNL从sql参数对象中计算表达式的值，根据表达式的值动态拼接sql，以此来完成动态sql的功能

1.源码类介绍

MyBatis中一些关于动态SQL的接口和类

SqlNode接口，简单理解就是xml中的每个标签，比如sql的update,if标签：

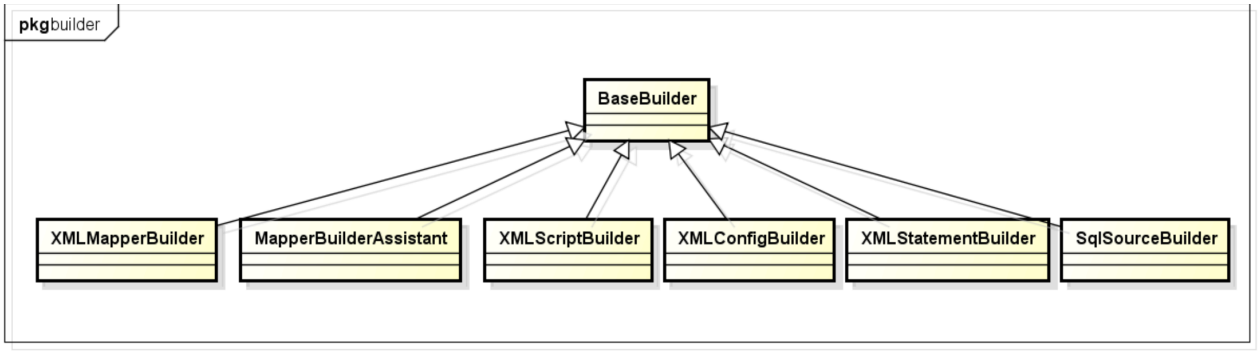
```
public interface SqlNode {
  boolean apply(DynamicContext context);
}
```

SqlSource Sql源接口，代表从xml文件或注解映射的sql内容，主要就是用于创建BoundSql，有实现类DynamicSqlSource(动态Sql源)，StaticSqlSource(静态Sql源)等：

```
public interface SqlSource {
  BoundSql getBoundSql(Object parameterObject);
}
```

BoundSql类，封装mybatis最终产生sql的类，包括sql语句，参数，参数源数据等参数：

BaseBuilder接口及其实现类



这些Builder的作用就是用于构造sql:

分析下其中4个Builder：

1 XMLConfigBuilder

解析mybatis中configLocation属性中的全局xml文件，内部会使用XMLMapperBuilder解析各个xml文件。

2 XMLMapperBuilder

遍历mybatis中mapperLocations属性中的xml文件中每个节点的Builder，比如user.xml，内部会使用XMLStatementBuilder处理xml中的每个节点。

3 XMLStatementBuilder

解析xml文件中各个节点，比如select,insert,update,delete节点，内部会使用XMLScriptBuilder处理节点的sql部分，遍历产生的数据会丢到Configuration的mappedStatements中。

4 XMLScriptBuilder

解析xml中各个节点sql部分的Builder。

LanguageDriver接口及其实现类，该接口主要的作用就是构造sql:

2.源码剖析

```
public void parse() {
    // 判断当前 Mapper 是否已经加载过
    if (!configuration.isResourceLoaded(resource)) {
        // 解析 `<mapper />` 节点
        configurationElement(parser.evalNode( expression: "/mapper"));
        // 标记该 Mapper 已经加载过
        configuration.addLoadedResource(resource);
        // 绑定 Mapper
        bindMapperForNamespace();
    }

    // 解析待定的 <resultMap /> 节点
    parsePendingResultMaps();
    // 解析待定的 <cache-ref /> 节点
    parsePendingCacheRefs();
    // 解析待定的 SQL 语句的节点
    parsePendingStatements();
}
```

根据XPath解析mapper节点

```
public XNode getSqlFragment(String refid) { return sqlFragments.get(refid); }
```

```
builderAssistant.setCurrentNamespace(namespace);
// 解析 <cache-ref /> 节点
cacheRefElement(context.evalNode("cache-ref"));
// 解析 <cache /> 节点
cacheElement(context.evalNode("cache"));
// 已废弃！老式风格的参数映射。内联参数是首选,这个元素可能在将来被移除，这里不会记录。
parameterMapElement(context.evalNodes( expression: "/mapper/parameterMap"));
// 解析 <resultMap /> 节点们
resultMapElements(context.evalNodes( expression: "/mapper/resultMap"));
// 解析 <sql /> 节点们
sqlElement(context.evalNodes( expression: "/mapper/sql"));
// 解析 <select /> <insert /> <update /> <delete /> 节点们
buildStatementFromContext(context.evalNodes( expression: "select|insert|update|delete"));
catch (Exception e) {
```

对于每个节点，使用XMLStatementBuilder解析

```
private void buildStatementFromContext(List<XNode> list, String requiredDatabaseId) {
    //遍历 <select /> <insert /> <update /> <delete /> 节点们
    for (XNode context : list) {
        // 创建 XMLStatementBuilder 对象，执行解析
        final XMLStatementBuilder statementParser = new XMLStatementBuilder(configuration, builderA
        try {
            statementParser.parseStatementNode();
        } catch (IncompleteElementException e) {
            // 解析失败，添加到 configuration 中
            configuration.addIncompleteStatement(statementParser);
        }
    }
```

XMLStatementBuilder的解析：

XMLStatementBuilder内部会使用LanguageDriver解析sql，并得到sqlSource

```
// Parse the SQL (pre: <selectKey> and <include> were parsed and removed)
// 创建 SqlSource 对象
SqlSource sqlSource = langDriver.createSqlSource(configuration, context, parameterTypeClass);
// 获得 KeyGenerator 对象
```

创建XMLScriptBuilder处理节点的sql部分

```java
@Override
public SqlSource createSqlSource(Configuration configuration, XNode script, Class<?> parameterTy
    // 创建 XMLScriptBuilder 对象，执行解析
    XMLScriptBuilder builder = new XMLScriptBuilder(configuration, script, parameterType);
    // 使用XMLScriptBuilder的parseScriptNode方法解析节点的SQL部分
    return builder.parseScriptNode();
}
```