转载 fedorafrog 于 2020-02-20 16:42:30 发布 9343 收藏 40 版权
分类专栏： MyBatis

数据分页功能是我们软件系统中必备的功能，在持久层使用mybatis的情况下，PageHelper来实现后台分页则是我们常用的一个选择，所以本文专门来介绍下。

1. 原理概述

PageHelper是MyBatis的一个插件，内部实现了一个PageInterceptor拦截器。Mybatis会加载这个拦截器到拦截器链中。在我们使用过程中先使用PageHelper.startPage这样的语句在当前线程上下文中设置一个ThreadLocal变量，再利用PageInterceptor这个分页拦截器拦截，从ThreadLocal中拿到分页的信息，如果有分页信息拼装分页SQL（limit语句等）进行分页查询，最后再把ThreadLocal中的东西清除掉。

2. 使用注意事项

PageHelper使用了ThreadLocal保存分页参数，分页参数和线程是绑定的。因此**我们需要保证PageHelper 的startPage调用后紧跟 MyBatis 查询方法，这就是安全的**。因为 PageHelper 在 finally 代码段中自动清除了 ThreadLocal 存储的对象。

如果代码在进入Executor前发生异常，就会导致线程不可用，这属于人为Bug（例如接口方法和XML中的不匹配，导致找不到MapppedStatement时）。这种情况由于线程不可用，也不会导致 ThreadLocal参数被错误使用。

但是如果写出以下代码，就是不安全的用法：

```java
PageHelper.startPage(1, 10);
List<Country> list;
if(param1 != null){
    list = countryMapper.selectIf(param1);
} else {
    list = new ArrayList<Country>();
}
```

这种情况下由于param1存在null的情况，就会导致PageHelper产生了一个分页参数，但是没有被消费，也没有被清理。这个参数就会一直保存在ThreadLocal中。可能导致其它不该分页的方法去消费了这个参数，莫名其妙的做了分页。

此外如果考虑到发生异常，可以加一个finally块，手动清理参数。

3. 使用过程源码分析

3.1 PageHelper.startPage

我们先来看PageHelper.startPage这个静态方法

```java
    /**
     * 开始分页
     *
     * @param pageNum      页码
     * @param pageSize     每页显示数量
     * @param count        是否进行count查询
     * @param reasonable   分页合理化,null时用默认配置
     * @param pageSizeZero true且pageSize=0时返回全部结果，false时分页,null时用默认
     */
    public static <E> Page<E> startPage(int pageNum, int pageSize, boolean cou
        Page<E> page = new Page<E>(pageNum, pageSize, count);
        page.setReasonable(reasonable);
        page.setPageSizeZero(pageSizeZero);
        //当已经执行过orderBy的时候
        Page<E> oldPage = getLocalPage();
        if (oldPage != null && oldPage.isOrderByOnly()) {
            page.setOrderBy(oldPage.getOrderBy());
        }
        setLocalPage(page);
        return page;
    }
```

上面的代码有个比较关键的地方setLocalPage(page)，方法是这样的：

```java
/**
 * 设置 Page 参数
 *
 * @param page
 */
protected static void setLocalPage(Page page) {
    LOCAL_PAGE.set(page);
}
```

再来看LOCAL_PAGE的定义

```java
protected static final ThreadLocal<Page> LOCAL_PAGE = new ThreadLocal<Page>();
```

终于明白了，是基于ThreadLocal，但是还没完，我们只看到了set的地方，却没有看到remove的地方,com.github.pagehelper.page.PageMethod类里有个clearPage方法：

```java
/**
 * 移除本地变量
 */
public static void clearPage() {
    LOCAL_PAGE.remove();
}
```

清除本地线程变量的就是这个clearPage方法，PageHelper在PageInterceptor的finally代码段中调用clearPage方法清除了ThreadLocal存储的对象。

3.2 PageInterceptor

我们再来看一下PageInterceptor拦截器是如何使用的。

3.2.1 加载过程

首先我们看拦截器是如何被加载的，查看SqlSessionFactoryBuilder类的build方法

```java
  public SqlSessionFactory build(InputStream inputStream, String environment, Propert
    try {
      XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, proper
      return build(parser.parse());
    } catch (Exception e) {
      throw ExceptionFactory.wrapException("Error building SqlSession.", e);
    } finally {
      ErrorContext.instance().reset();
      try {
        inputStream.close();
      } catch (IOException e) {
        // Intentionally ignore. Prefer previous error.
      }
    }
  }
```

核心进这个方法

进入pluginElement方法

```java
private void pluginElement(XNode parent) throws Exception {
  if (parent != null) {
    for (XNode child : parent.getChildren()) {
      // 获取到内容:com.github.pagehelper.PageHelper
      String interceptor = child.getStringAttribute("interceptor");
      // 获取配置的属性信息
      Properties properties = child.getChildrenAsProperties();
      // 创建的拦截器实例
      Interceptor interceptorInstance = (Interceptor) resolveClass(interceptor
      // 将属性和拦截器绑定
      interceptorInstance.setProperties(properties);
      // 这个方法需要进入查看
      configuration.addInterceptor(interceptorInstance);
    }
  }
}
```

addInterceptor就是加载拦截器的方法

```java
public void addInterceptor(Interceptor interceptor) {
    // 将拦截器添加到了 拦截器链中 而拦截器链本质上就是一个List有序集合
    interceptorChain.addInterceptor(interceptor);
  }
```

```java
public class InterceptorChain {

  private final List<Interceptor> interceptors = new ArrayList<Interceptor>();

  public Object pluginAll(Object target) {
    for (Interceptor interceptor : interceptors) {
      target = interceptor.plugin(target);
    }
    return target;
  }

  public void addInterceptor(Interceptor interceptor) {
    interceptors.add(interceptor);
  }
```

通过SqlSessionFactory对象的获取，加载了全局配置文件及映射文件**同时还将配置的拦截器添加到了拦截器链中。**

3.2.2 PageHelper定义的拦截信息

我们来看看PageHepler的源代码片段

```java
@SuppressWarnings("rawtypes")
@Intercepts(
    @Signature(
        type = Executor.class,
        method = "query",
        args = {MappedStatement.class
                , Object.class
                , RowBounds.class
                , ResultHandler.class
            }))
public class PageHelper implements Interceptor {
    //sql工具类
    private SqlUtil sqlUtil;
    //属性参数信息
    private Properties properties;
    //配置对象方式
    private SqlUtilConfig sqlUtilConfig;
    //自动获取dialect,如果没有setProperties或setSqlUtilConfig，也可以正常进行
    private boolean autoDialect = true;
    //运行时自动获取dialect
    private boolean autoRuntimeDialect;
    //多数据源时，获取jdbcurl后是否关闭数据源
    private boolean closeConn = true;
```

这里可以看到通过注解，定义的是拦截 Executor对象中的query(MappedStatement ms,Object o,RowBounds ob ResultHandler rh)这个方法。

3.2.3 Executor中发生的事情

接下来我们需要分析下SqlSession的实例化过程中Executor发生了什么。我们需要从这行代码开始跟踪

```java
public SqlSession openSession() {
  return openSessionFromDataSource(configuration.getDefaultExecutorType(), nul
}
```

```java
  private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsc
    Transaction tx = null;
    try {
      final Environment environment = configuration.getEnvironment();
      final TransactionFactory transactionFactory = getTransactionFactoryFromEnviron
      tx = transactionFactory.newTransaction(environment.getDataSource(), level, aut
      final Executor executor = configuration.newExecutor(tx, execType);
      return new DefaultSqlSession(configuration, executor, autoCommit);
    } catch (Exception e) {
      closeTransaction(tx); // may have fetched a connection so lets call close()
      throw ExceptionFactory.wrapException("Error opening session.  Cause: " + e, e);
    } finally {
      ErrorContext.instance().reset();
    }
  }
```

核心代码 进入

再来看newExecutor中做了什么处理

```java
  public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    Executor executor;
    if (ExecutorType.BATCH == executorType) {
      executor = new BatchExecutor(this, transaction);
    } else if (ExecutorType.REUSE == executorType) {
      executor = new ReuseExecutor(this, transaction);
    } else {
      executor = new SimpleExecutor(this, transaction);
    }
    if (cacheEnabled) {
      executor = new CachingExecutor(executor);
    }
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
  }
```

重点代码 进入
拦截器链是我们加载配置文件的时候看过的

这里循环调用了所有拦截器

```java
  public Object pluginAll(Object target) {
    for (Interceptor interceptor : interceptors) {
      target = interceptor.plugin(target);
    }
    return target;
  }
```

调用拦截器的plugin方法

增强Executor实例

```java
  public Object plugin(Object target) {
      if (target instanceof Executor) {
          return Plugin.wrap(target, this);
      } else {
          return target;
      }
  }
```

如果是需要拦截的类型 就wrap增强

```java
  public static Object wrap(Object target, Interceptor interceptor) {
    Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
    Class<?> type = target.getClass();
    Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
    if (interfaces.length > 0) {
      return Proxy.newProxyInstance(
          type.getClassLoader(),
          interfaces,
          new Plugin(target, interceptor, signatureMap));
    }
    return target;
  }
```

返回的是Jdk动态代理类 代理类增强了目标类
目标对象 Executor

到此我们明白了,Executor对象其实被我们生存的代理类增强了。invoke的代码为

```java
public Object invoke(Object proxy, Method method, Object[] args) throws Throwa
  try {
    Set<Method> methods = signatureMap.get(method.getDeclaringClass());
    // 如果是定义的拦截的方法 就执行intercept方法
    if (methods != null && methods.contains(method)) {
      // 进入查看 该方法增强
      return interceptor.intercept(new Invocation(target, method, args));
    }
    // 不是需要拦截的方法 直接执行
    return method.invoke(target, args);
  } catch (Exception e) {
    throw ExceptionUtil.unwrapThrowable(e);
  }
}
```

3.2.4 selectList方法

当我们调用例如session.selectList("com.bobo.UserMapper.query")方法时，最终是交由Executor类的方法来处理

```java
  public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {
    try {
      MappedStatement ms = configuration.getMappedStatement(statement);
      List<E> result = executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO_RESULT_HANDLER);
      return result;
    } catch (Exception e) {
      throw ExceptionFactory.wrapException("Error querying database.  Cause: " + e, e);
    } finally {
      ErrorContext.instance().reset();
    }
  }
```

这个就是我们需要拦截的方法 在执行query方法的时候会触发Executor的invoke方法

我们需要回到invoke方法中继续看

```java
/**
 * Mybatis拦截器方法
 *
 * @param invocation 拦截器入参
 * @return 返回执行结果
 * @throws Throwable 抛出异常
 */
public Object intercept(Invocation invocation) throws Throwable {
    if (autoRuntimeDialect) {
        SqlUtil sqlUtil = getSqlUtil(invocation);
        return sqlUtil.processPage(invocation);
    } else {
        if (autoDialect) {
            initSqlUtil(invocation);
        }
        return sqlUtil.processPage(invocation);
    }
}
```

进入sqlUtil.processPage(invocation);方法，这里会从ThreadLocal中获取分页信息

```java
/**
 * Mybatis拦截器方法
 *
 * @param invocation 拦截器入参
 * @return 返回执行结果
 * @throws Throwable 抛出异常
 */
private Object _processPage(Invocation invocation) throws Throwable {
    final Object[] args = invocation.getArgs();
    Page page = null;
    //支持方法参数时，会先尝试获取Page
    if (supportMethodsArguments) {
        // 从线程本地变量中获取Page信息，就是我们刚刚设置的
        page = getPage(args);
    }
    //分页信息
    RowBounds rowBounds = (RowBounds) args[2];
    //支持方法参数时，如果page == null就说明没有分页条件，不需要分页查询
    if ((supportMethodsArguments && page == null)
            //当不支持分页参数时，判断LocalPage和RowBounds判断是否需要分页
            || (!supportMethodsArguments && SqlUtil.getLocalPage() == null &&
        return invocation.proceed();
    } else {
        //不支持分页参数时，page==null，这里需要获取
        if (!supportMethodsArguments && page == null) {
            page = getPage(args);
        }
        // 进入查看
        return doProcessPage(invocation, page, args);
    }
}
```

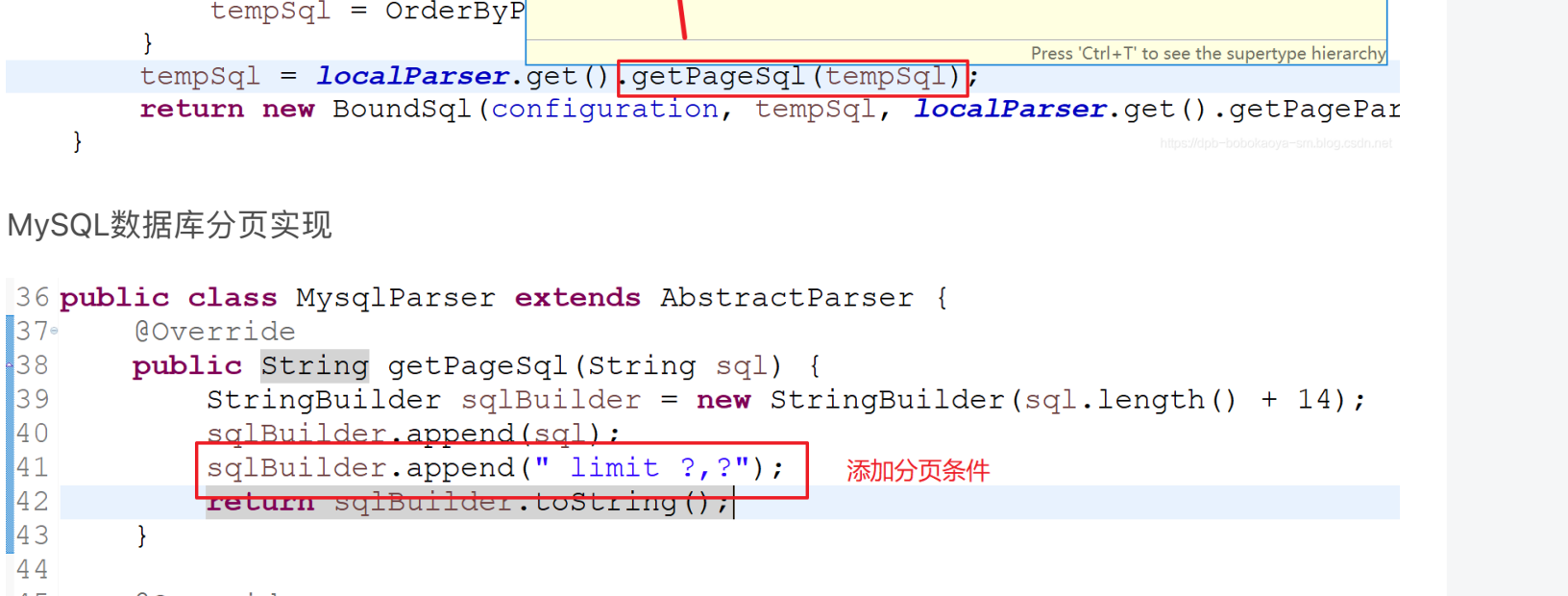
```java
/**
  * Mybatis拦截器方法
  *
  * @param invocation 拦截器入参
  * @return 返回执行结果
  * @throws Throwable 抛出异常
  */
 private Page doProcessPage(Invocation invocation, Page page, Object[] args) t
     //保存RowBounds状态
     RowBounds rowBounds = (RowBounds) args[2];
     //获取原始的ms
     MappedStatement ms = (MappedStatement) args[0];
     //判断并处理为PageSqlSource
     if (!isPageSqlSource(ms)) {
         processMappedStatement(ms);
     }
     //设置当前的parser，后面每次使用前都会set，ThreadLocal的值不会产生不良影响
     ((PageSqlSource)ms.getSqlSource()).setParser(parser);
     try {
         //忽略RowBounds-否则会进行Mybatis自带的内存分页
         args[2] = RowBounds.DEFAULT;
         //如果只进行排序 或 pageSizeZero的判断
         if (isQueryOnly(page)) {
             return doQueryOnly(page, invocation);
         }

         //简单的通过total的值来判断是否进行count查询
         if (page.isCount()) {
             page.setCountSignal(Boolean.TRUE);
             //替换MS
             args[0] = msCountMap.get(ms.getId());
             //查询总数
             Object result = invocation.proceed();
             //还原ms
             args[0] = ms;
             //设置总数
             page.setTotal((Integer) ((List) result).get(0));
             if (page.getTotal() == 0) {
                 return page;
             }
         } else {
             page.setTotal(-1l);
         }
         //pageSize>0的时候执行分页查询，pageSize<=0的时候不执行相当于可能只返回了一个c
         if (page.getPageSize() > 0 &&
                 ((rowBounds == RowBounds.DEFAULT && page.getPageNum() > 0)
                         || rowBounds != RowBounds.DEFAULT)) {
             //将参数中的MappedStatement替换为新的qs
             page.setCountSignal(null);
             // 重点是查看该方法
             BoundSql boundSql = ms.getBoundSql(args[1]);
             args[1] = parser.setPageParameter(ms, args[1], boundSql, page);
             page.setCountSignal(Boolean.FALSE);
             //执行分页查询
             Object result = invocation.proceed();
             //得到处理结果
             page.addAll((List) result);
         }
     } finally {
         ((PageSqlSource)ms.getSqlSource()).removeParser();
     }

     //返回结果
     return page;
 }
```

进入 BoundSql boundSql = ms.getBoundSql(args[1])方法跟踪到PageStaticSqlSource类中的

```java
@Override
protected BoundSql getPageBoundSql(Object parameterObject) {
    String tempSql = sql;
    String orderBy = PageHelper.getOrderBy();
    if (orderBy != null) {
        tempSql = OrderByParser.converToOrderBySql(sql, orderBy);
    }
    tempSql = localParser.get().getPageSql(tempSql);
    return new BoundSql(configuration, tempSql, localParser.get().getPageParam
}
```



MySQL数据库分页实现

```java
public class MysqlParser extends AbstractParser {
    @Override
    public String getPageSql(String sql) {
        StringBuilder sqlBuilder = new StringBuilder(sql.length() + 14);
        sqlBuilder.append(sql);
        sqlBuilder.append(" limit ?,?");
        return sqlBuilder.toString();
    }

    @Override
    public Map<String, Object> setPageParameter(MappedStatement ms, Object
        Map<String, Object> paramMap = super.setPageParameter(ms, paramete
        paramMap.put(PAGEPARAMETER_FIRST, page.getStartRow());
        paramMap.put(PAGEPARAMETER_SECOND, page.getPageSize());
        return paramMap;
    }
}
```

添加分页条件

至此我们发现PageHelper分页的实现原来是在我们执行SQL语句之前动态的将SQL语句拼接了分页的语句，从而实现了从数据库中分页获取的过程。