Mybaties 五大问题灵魂拷问

Mybaties 缓存

MyBatis 是常见的 Java 数据库访问层框架。在日常工作中，开发人员多数情况下是使用 MyBatis 的默认缓存配置，但是 MyBatis 缓存机制有一些不足之处，在使用中容易引起脏数据，形成一些潜在的隐患。个人在业务开发中也处理过一些由于 MyBatis 缓存引发的开发问题，带着个人的兴趣，希望从应用及源码的角度为读者梳理 MyBatis 缓存机制。

本次分析中涉及到的代码和数据库表均放在 GitHub 上，地址： mybatis-cache-demo (https://github.com/kailuncen/mybatis-cache-demo) 。

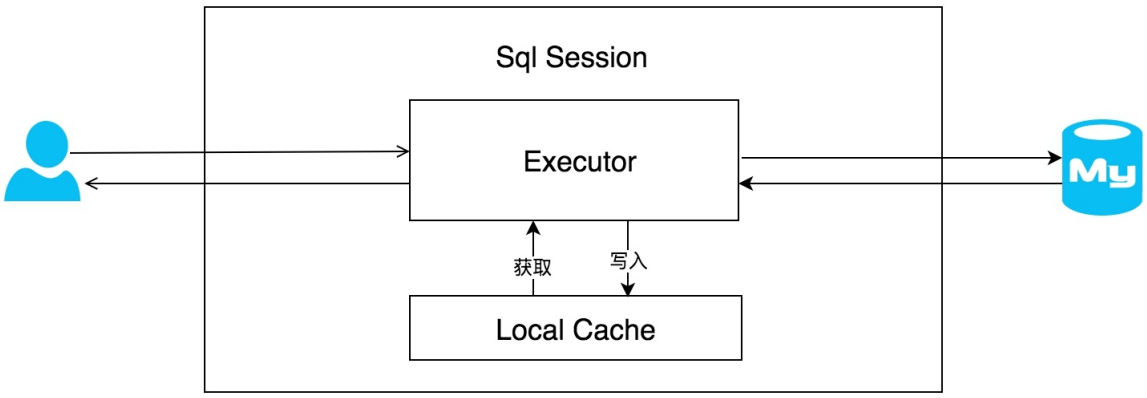
目录

本文按照以下顺序展开。

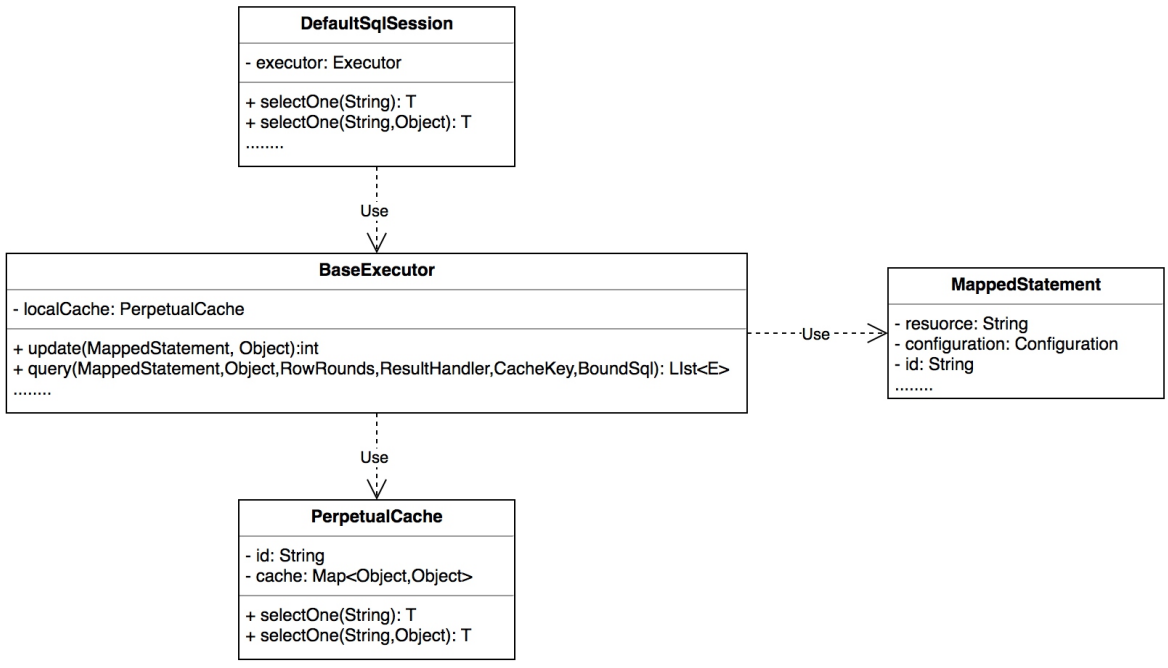
一级缓存

一级缓存介绍

在应用运行过程中，我们有可能在一次数据库会话中，执行多次查询条件完全相同的 SQL，MyBatis 提供了一级缓存的方案优化这部分场景，如果是相同的 SQL 语句，会优先命中一级缓存，避免直接对数据库进行查询，提高性能。具体执行过程如下图所示。



每个 SqlSession 中持有了 Executor，每个 Executor 中有一个 LocalCache。当用户发起查询时，MyBatis 根据当前执行的语句生成 `MappedStatement` ，在 Local Cache 进行查询，如果缓存命中的话，直接返回结果给用户，如果缓存没有命中的话，查询数据库，结果写入 `Local Cache` ，最后返回结果给用户。具体实现类的类关系图如下图所示。



一级缓存配置

我们来看看如何使用 MyBatis 一级缓存。开发者只需在 MyBatis 的配置文件中，添加如下语句，就可以使用一级缓存。共有两个选项， `SESSION` 或者 `STATEMENT` ，默认是 `SESSION` 级别，即在一个 MyBatis 会话中执行的所有语句，都会共享这一个缓存。一种是 `STATEMENT` 级别，可以理解为缓存只对当前执行的这一个 `Statement` 有效。

一级缓存实验

接下来通过实验，了解 MyBatis 一级缓存的效果，每个单元测试后都请恢复被修改的数据。

首先是创建示例表 student，创建对应的 POJO 类和增改的方法，具体可以在 entity 包和 mapper 包中查看。

```
CREATE TABLE `student` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(200) COLLATE utf8_bin DEFAULT NULL,
  `age` tinyint(3) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
```

在以下实验中，id 为 1 的学生名称是凯伦。

实验 1

开启一级缓存，范围为会话级别，调用三次 `getStudentById`，代码如下所示：

```
public void getStudentById() throws Exception {
        SqlSession sqlSession = factory.openSession(true); // 自动提交事务
        StudentMapper studentMapper =
sqlSession.getMapper(StudentMapper.class);
        System.out.println(studentMapper.getStudentById(1));
        System.out.println(studentMapper.getStudentById(1));
        System.out.println(studentMapper.getStudentById(1));
    }
```

执行结果：

```
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 凯伦, 25
DEBUG [main] - <==      Total: 1
StudentEntity{id=1, name='凯伦', age=25}
StudentEntity{id=1, name='凯伦', age=25}
StudentEntity{id=1, name='凯伦', age=25}
```

只有第一次真正查询了数据库

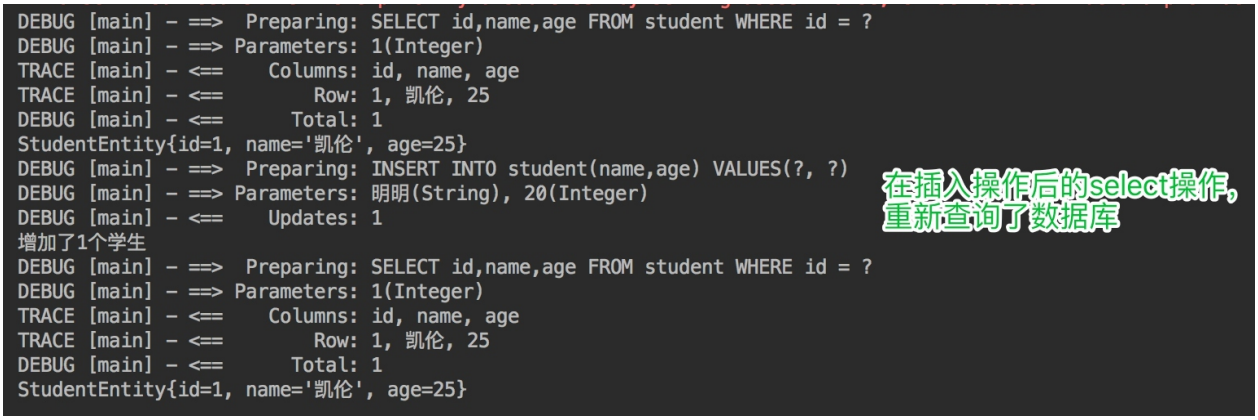
我们可以看到，只有第一次真正查询了数据库，后续的查询使用了一级缓存。

实验 2

增加了对数据库的修改操作，验证在一次数据库会话中，如果对数据库发生了修改操作，一级缓存是否会失效。

```
@Test
public void addStudent() throws Exception {
        SqlSession sqlSession = factory.openSession(true); // 自动提交事务
        StudentMapper studentMapper =
sqlSession.getMapper(StudentMapper.class);
        System.out.println(studentMapper.getStudentById(1));
        System.out.println("增加了" + studentMapper.addStudent(buildStudent())
+ "个学生");
        System.out.println(studentMapper.getStudentById(1));
        sqlSession.close();
}
```

执行结果：



我们可以看到，在修改操作后执行的相同查询，查询了数据库， **一级缓存失效。**

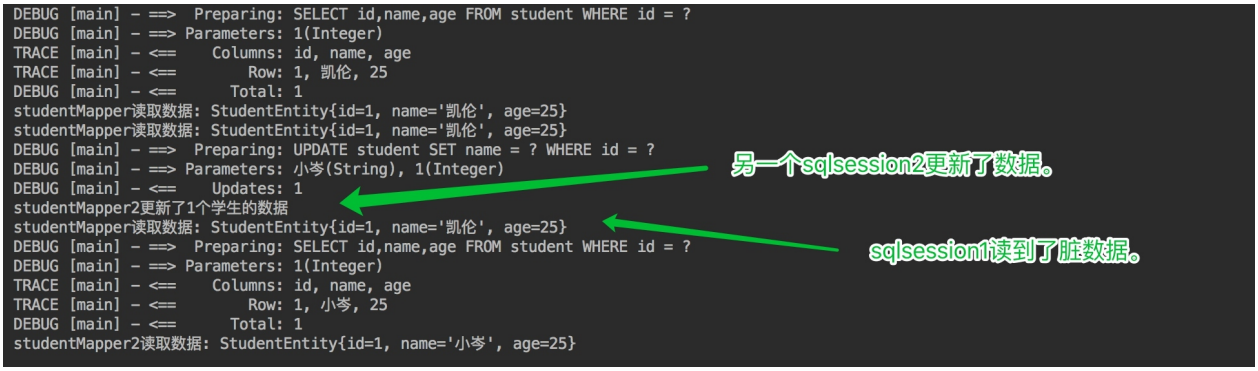
实验 3

开启两个 `SqlSession` ，在 `sqlSession1` 中查询数据，使一级缓存生效，在 `sqlSession2` 中更新数据库，验证一级缓存只在数据库会话内部共享。

```
@Test
public void testLocalCacheScope() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);

        StudentMapper studentMapper =
sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 =
sqlSession2.getMapper(StudentMapper.class);

        System.out.println("studentMapper读取数据: " +
studentMapper.getStudentById(1));
        System.out.println("studentMapper读取数据: " +
studentMapper.getStudentById(1));
        System.out.println("studentMapper2更新了" +
studentMapper2.updateStudentName("小岑",1) + "个学生的数据");
        System.out.println("studentMapper读取数据: " +
studentMapper.getStudentById(1));
        System.out.println("studentMapper2读取数据: " +
studentMapper2.getStudentById(1));
}
```



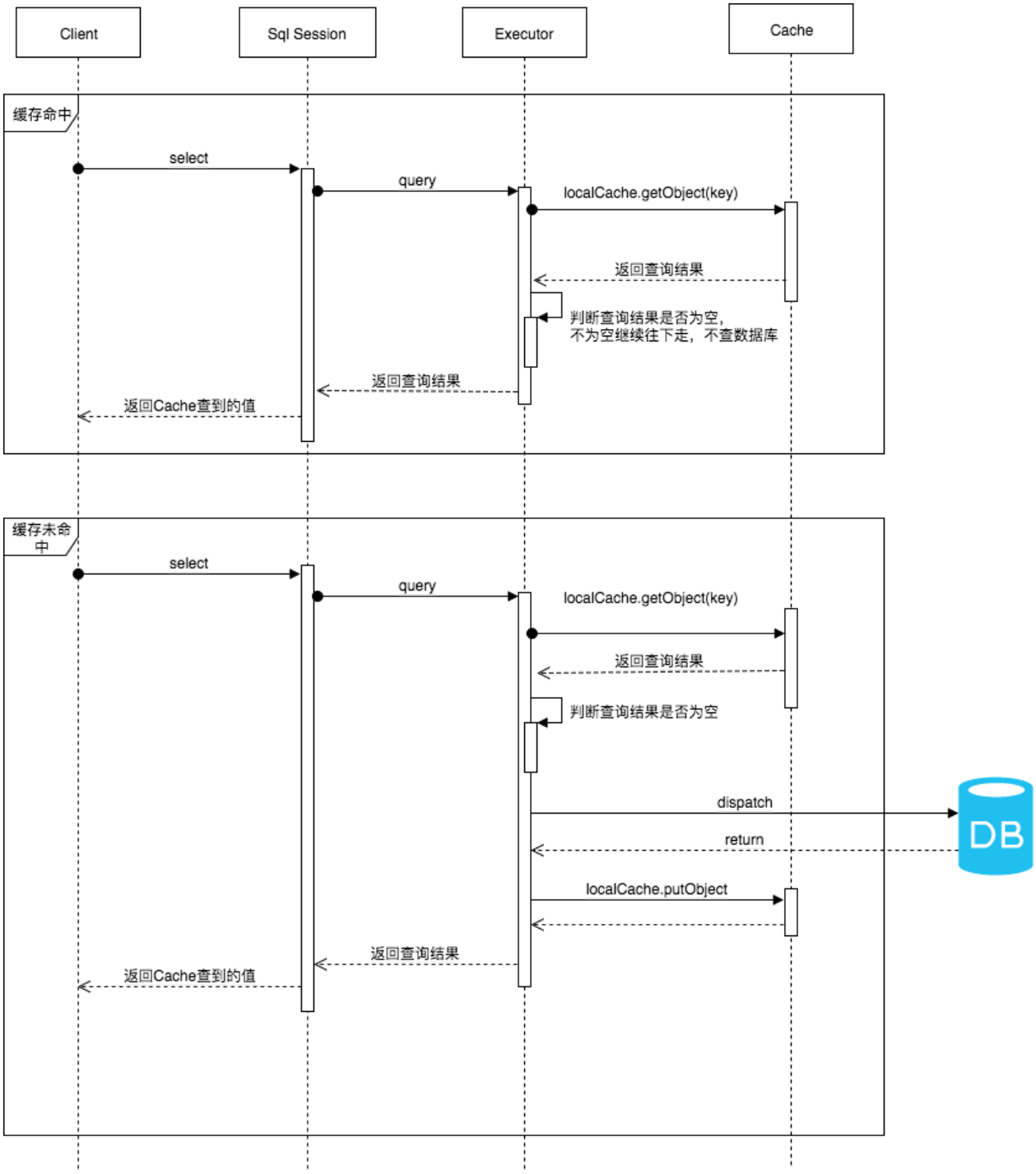
`sqlSession2` 更新了 id 为 1 的学生的姓名，从凯伦改为了小岑，但 session1 之后的查询中，id 为 1 的学生的名字还是凯伦，出现了脏数据，也证明了之前的设想，一级缓存只在数据库会话内部共享。

一级缓存工作流程&源码分析

那么，一级缓存的工作流程是怎样的呢？我们从源码层面来学习一下。

工作流程

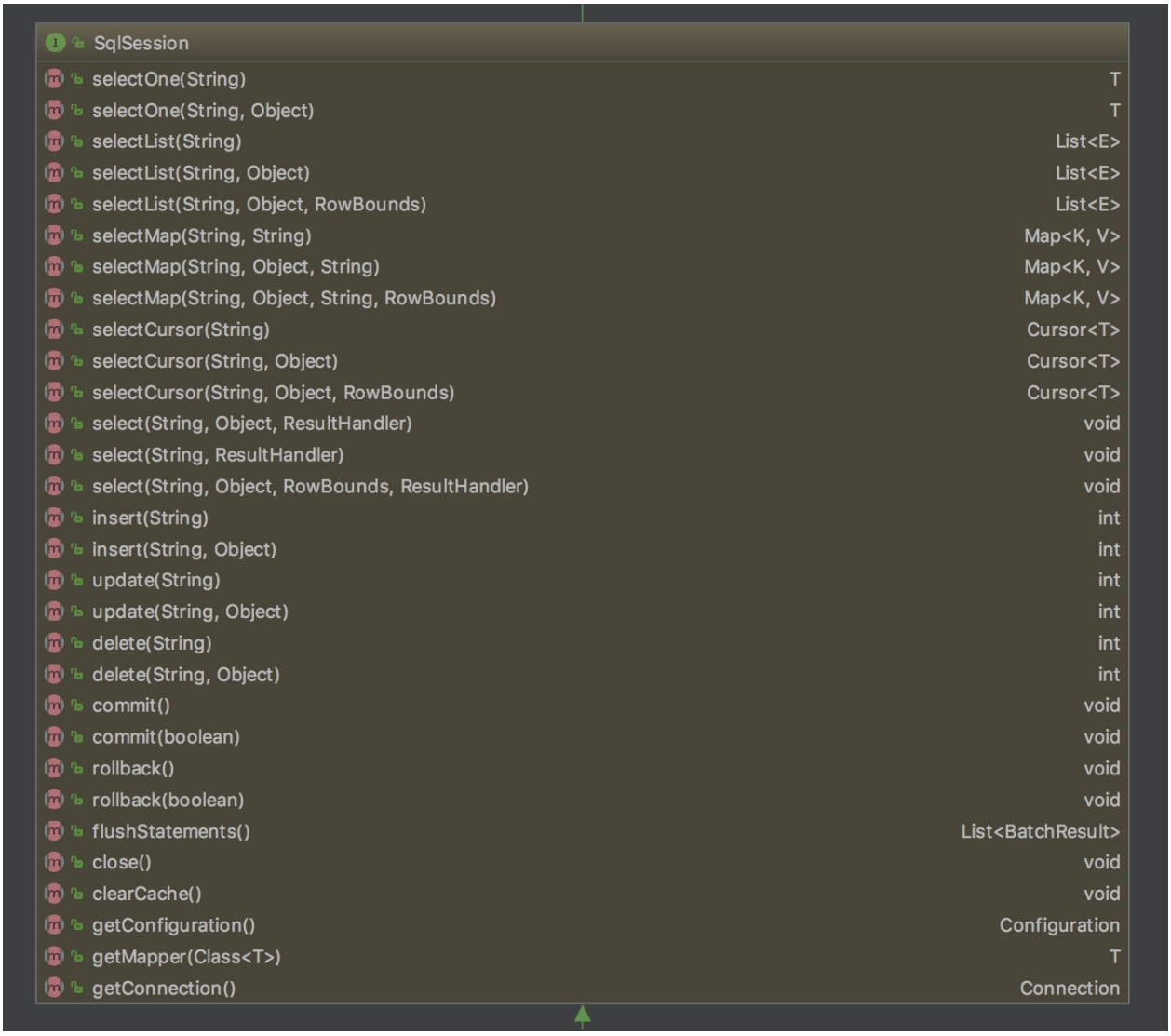
一级缓存执行的时序图，如下图所示。



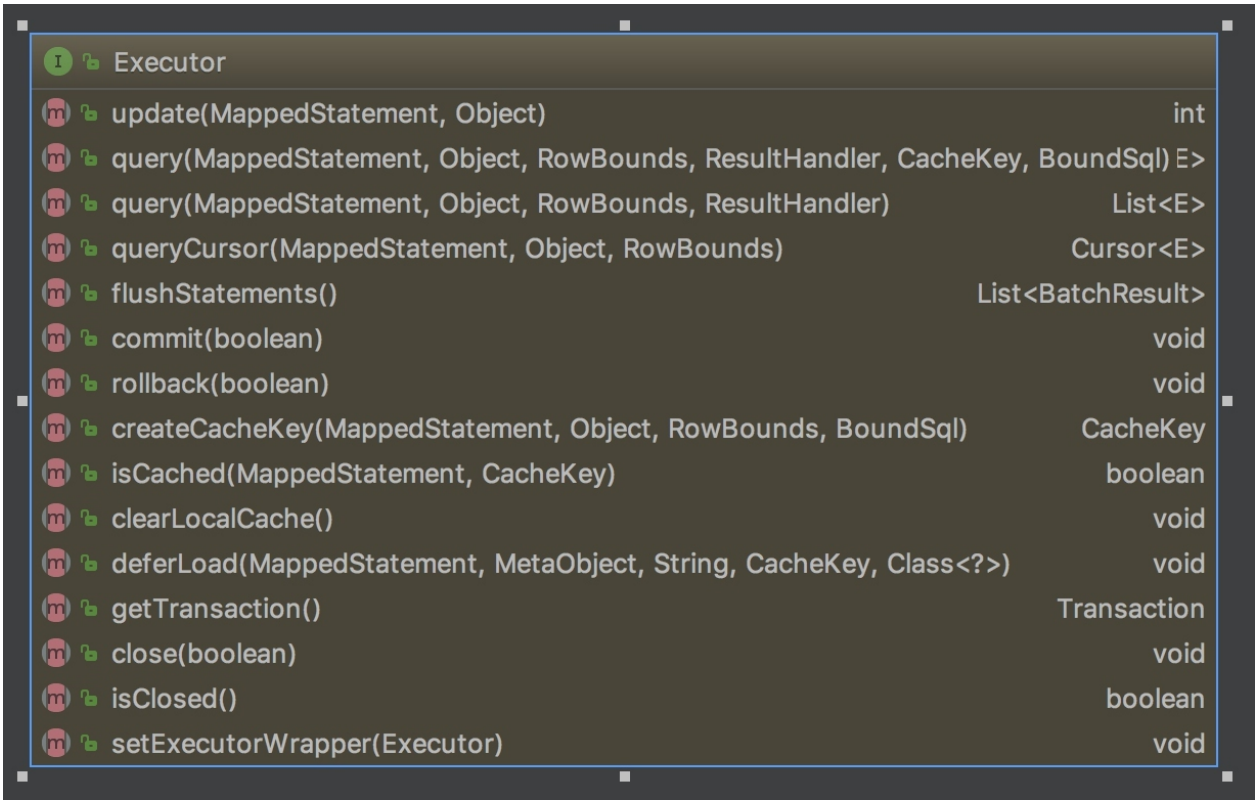
源码分析

接下来将对 MyBatis 查询相关的核心类和一级缓存的源码进行走读。这对后面学习二级缓存也有帮助。

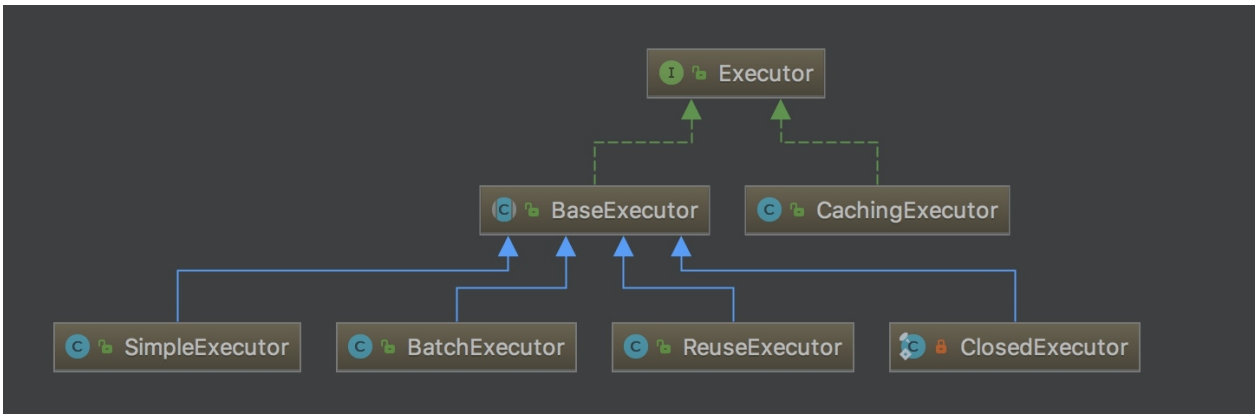
SqlSession： 对外提供了用户和数据库之间交互需要的所有方法，隐藏了底层的细节。默认实现类是 `DefaultSqlSession` 。



Executor： `SqlSession` 向用户提供操作数据库的方法，但和数据库操作有关的职责都会委托给 Executor。



如下图所示，Executor 有若干个实现类，为 Executor 赋予了不同的能力，大家可以根据类名，自行学习每个类的基本作用。



在一级缓存的源码分析中，主要学习 `BaseExecutor` 的内部实现。

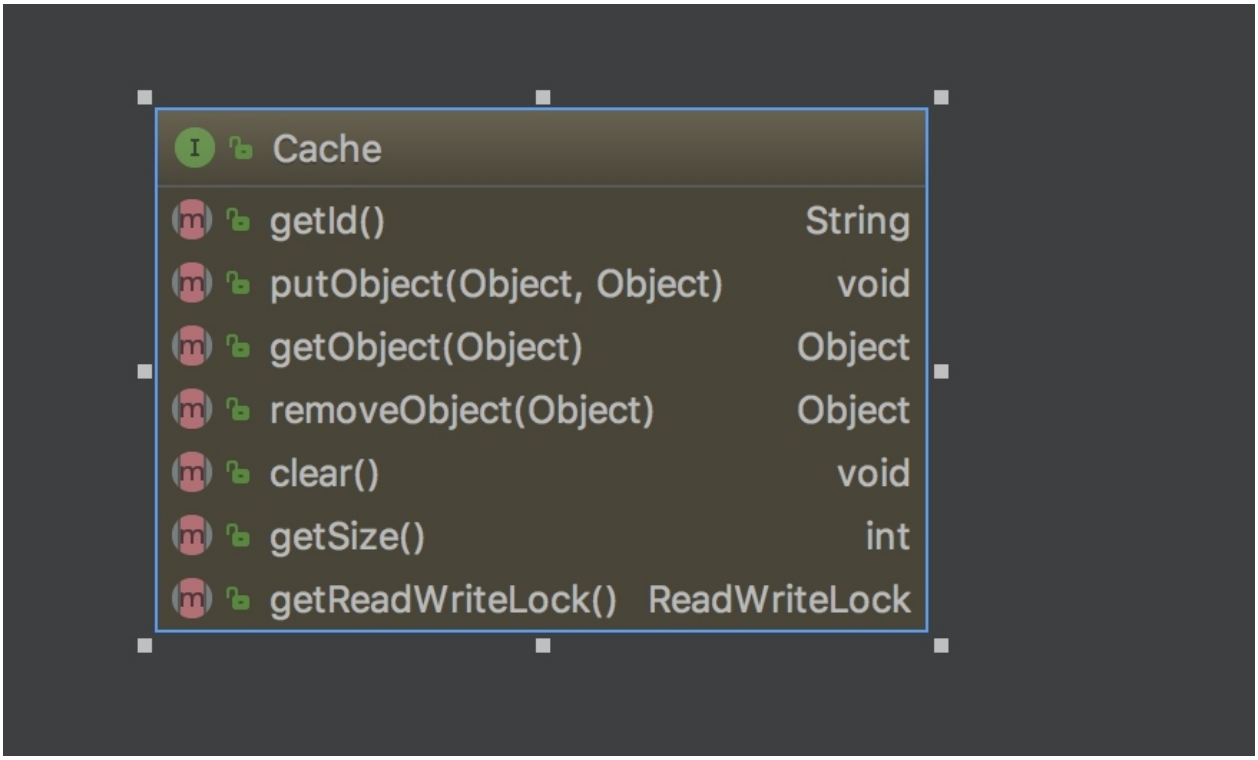
BaseExecutor： `BaseExecutor` 是一个实现了 Executor 接口的抽象类，定义若干抽象方法，在执行的时候，把具体的操作委托给子类进行执行。

```
protected abstract int doUpdate(MappedStatement ms, Object parameter) throws
SQLException;
protected abstract List doFlushStatements(boolean isRollback) throws
SQLException;
protected abstract  List doQuery(MappedStatement ms, Object parameter,
RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws
SQLException;
protected abstract  Cursor doQueryCursor(MappedStatement ms, Object parameter,
RowBounds rowBounds, BoundSql boundSql) throws SQLException;
```

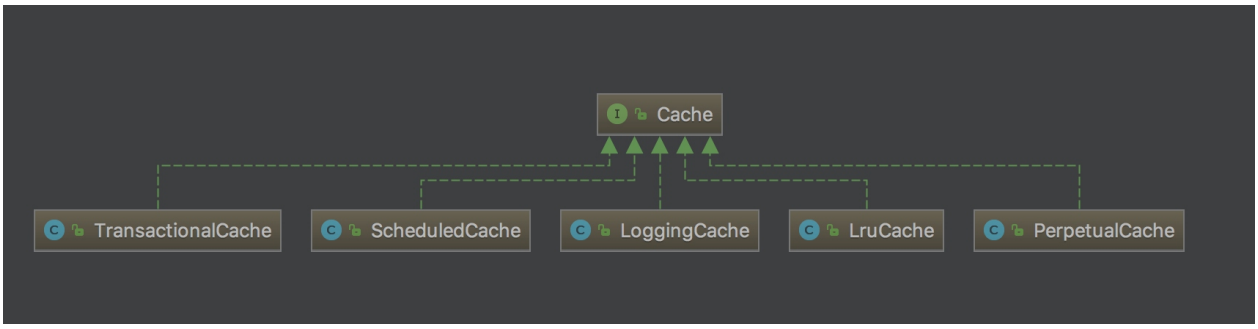
在一级缓存的介绍中提到对 `Local Cache` 的查询和写入是在 `Executor` 内部完成的。在阅读 `BaseExecutor` 的代码后发现 `Local Cache` 是 `BaseExecutor` 内部的一个成员变量，如下代码所示。

```
public abstract class BaseExecutor implements Executor {
protected ConcurrentLinkedQueue deferredLoads;
protected PerpetualCache localCache;
```

Cache： MyBatis 中的 Cache 接口，提供了和缓存相关的最基本的操作，如下图所示：



有若干个实现类，使用装饰器模式互相组装，提供丰富的操控缓存的能力，部分实现类如下图所示：



`BaseExecutor` 成员变量之一的 `PerpetualCache` ，是对 Cache 接口最基本的实现，其实现非常简单，内部持有 HashMap，对一级缓存的操作实则是对 HashMap 的操作。如下代码所示：

```
public class PerpetualCache implements Cache {
  private String id;
  private Map cache = new HashMap();
```

在阅读相关核心类代码后，从源代码层面对一级缓存工作中涉及到的相关代码，出于篇幅的考虑，对源码做适当删减，读者朋友可以结合本文，后续进行更详细的学习。

为执行和数据库的交互，首先需要初始化 `SqlSession` ，通过 `DefaultSqlSessionFactory` 开启 `SqlSession` ：

```
private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
    ............

    final Executor executor = configuration.newExecutor(tx, execType);
    return new DefaultSqlSession(configuration, executor, autoCommit);
}
```

在初始化 SqlSesion 时，会使用 Configuration 类创建一个全新的 Executor ，作为 DefaultSqlSession 构造函数的参数，创建 Executor 代码如下所示：

```
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);
    }
    // 尤其可以注意这里，如果二级缓存开关开启的话，是使用CahingExecutor装饰
BaseExecutor的子类
    if (cacheEnabled) {
      executor = new CachingExecutor(executor);
    }
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}
```

SqlSession 创建完毕后，根据 Statment 的不同类型，会进入 SqlSession 的不同方法中，如果是 Select 语句的话，最后会执行到 SqlSession 的 selectList ，代码如下所示：

```
@Override
public  List selectList(String statement, Object parameter, RowBounds
rowBounds) {
      MappedStatement ms = configuration.getMappedStatement(statement);
      return executor.query(ms, wrapCollection(parameter), rowBounds,
Executor.NO_RESULT_HANDLER);
}
```

SqlSession 把具体的查询职责委托给了 Executor。如果只开启了一级缓存的话，首先会进入 BaseExecutor 的 query 方法。代码如下所示：

```
@Override
public  List query(MappedStatement ms, Object parameter, RowBounds rowBounds,
ResultHandler resultHandler) throws SQLException {
    BoundSql boundSql = ms.getBoundSql(parameter);
    CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
    return query(ms, parameter, rowBounds, resultHandler, key, boundSql);
}
```

在上述代码中，会先根据传入的参数生成 CacheKey，进入该方法查看 CacheKey 是如何生成的，代码如下所示：

```
CacheKey cacheKey = new CacheKey();
cacheKey.update(ms.getId());
cacheKey.update(rowBounds.getOffset());
cacheKey.update(rowBounds.getLimit());
cacheKey.update(boundSql.getSql());
//后面是update了sql中带的参数
cacheKey.update(value);
```

在上述的代码中，将 `MappedStatement` 的 Id、SQL 的 offset、SQL 的 limit、SQL 本身以及 SQL 中的参数传入了 CacheKey 这个类，最终构成 CacheKey。以下是这个类的内部结构：

```
private static final int DEFAULT_MULTIPLYER = 37;
private static final int DEFAULT_HASHCODE = 17;

private int multiplier;
private int hashcode;
private long checksum;
private int count;
private List updateList;

public CacheKey() {
    this.hashcode = DEFAULT_HASHCODE;
    this.multiplier = DEFAULT_MULTIPLYER;
    this.count = 0;
    this.updateList = new ArrayList();
}
```

首先是成员变量和构造函数，有一个初始的 `hachcode` 和乘数，同时维护了一个内部的 `updatelist` 。在 `CacheKey` 的 `update` 方法中，会进行一个 `hashcode` 和 `checksum` 的计算，同时把传入的参数添加进 `updatelist` 中。如下代码所示：

```
public void update(Object object) {
    int baseHashCode = object == null ? 1 : ArrayUtil.hashCode(object);
    count++;
    checksum += baseHashCode;
    baseHashCode *= count;
    hashcode = multiplier * hashcode + baseHashCode;

    updateList.add(object);
}
```

同时重写了 CacheKey 的 equals 方法，代码如下所示：

```
@Override
public boolean equals(Object object) {
    ..............

    for (int i = 0; i < updateList.size(); i++) {
      Object thisObject = updateList.get(i);
      Object thatObject = cacheKey.updateList.get(i);
      if (!ArrayUtil.equals(thisObject, thatObject)) {
        return false;
      }
    }
    return true;
}
```

除去 hashcode、checksum 和 count 的比较外，只要 updatelist 中的元素一一对应相等，那么就可以认为是 CacheKey 相等。只要两条 SQL 的下列五个值相同，即可以认为是相同的 SQL。

Statement Id + Offset + Limmit + Sql + Params

BaseExecutor 的 query 方法继续往下走，代码如下所示：

```
list = resultHandler == null ? (List) localCache.getObject(key) : null;
if (list != null) {
    // 这个主要是处理存储过程用的。
    handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
    } else {
    list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
boundSql);
}
```

如果查不到的话，就从数据库查，在 queryFromDatabase 中，会对 localcache 进行写入。

在 query 方法执行的最后，会判断一级缓存级别是否是 STATEMENT 级别，如果是的话，就清空缓存，这也就是 STATEMENT 级别的一级缓存无法共享 localCache 的原因。代码如下所示：

```
if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
        clearLocalCache();
}
```

在源码分析的最后，我们确认一下，如果是 insert/delete/update 方法，缓存就会刷新的原因。

SqlSession 的 insert 方法和 delete 方法，都会统一走 update 的流程，代码如下所示：

```
@Override
public int insert(String statement, Object parameter) {
    return update(statement, parameter);
  }
   @Override
  public int delete(String statement) {
    return update(statement, null);
}
```

update 方法也是委托给了 Executor 执行。 BaseExecutor 的执行方法如下所示：

```
@Override
public int update(MappedStatement ms, Object parameter) throws SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing an
update").object(ms.getId());
    if (closed) {
      throw new ExecutorException("Executor was closed.");
    }
    clearLocalCache();
    return doUpdate(ms, parameter);
}
```

每次执行 update 前都会清空 localCache 。

至此，一级缓存的工作流程讲解以及源码分析完毕。

总结

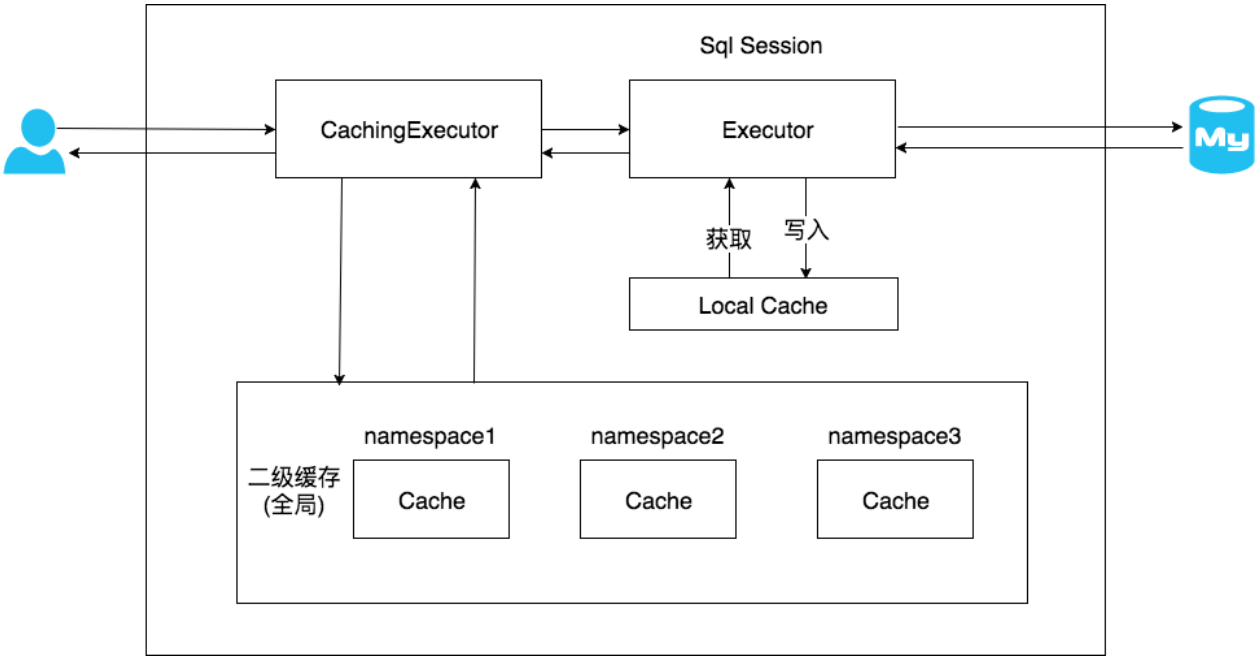
1. MyBatis 一级缓存的生命周期和 SqlSession 一致。
2. MyBatis 一级缓存内部设计简单，只是一个没有容量限定的 HashMap，在缓存的功能性上有所欠缺。

3. MyBatis 的一级缓存最大范围是 SqlSession 内部，有多个 SqlSession 或者分布式的环境下，数据库写操作会引起脏数据，建议设定缓存级别为 Statement。

二级缓存

二级缓存介绍

在上文中提到的一级缓存中，其最大的共享范围就是一个 SqlSession 内部，如果多个 SqlSession 之间需要共享缓存，则需要使用到二级缓存。开启二级缓存后，会使用 CachingExecutor 装饰 Executor，进入一级缓存的查询流程前，先在 CachingExecutor 进行二级缓存的查询，具体的工作流程如下所示。



二级缓存开启后，同一个 namespace 下的所有操作语句，都影响着同一个 Cache，即二级缓存被多个 SqlSession 共享，是一个全局的变量。

当开启缓存后，数据的查询执行的流程就是 二级缓存 -> 一级缓存 -> 数据库。

二级缓存配置

要正确的使用二级缓存，需完成如下配置的。

1. 在 MyBatis 的配置文件中开启二级缓存。

```
```

1. 在 MyBatis 的映射 XML 中配置 cache 或者 cache-ref 。

cache 标签用于声明这个 namespace 使用二级缓存，并且可以自定义配置。

```
```

- `type`： cache 使用的类型，默认是 `PerpetualCache` ，这在一级缓存中提到过。
- `eviction`： 定义回收的策略，常见的有 FIFO，LRU。
- `flushInterval`： 配置一定时间自动刷新缓存，单位是毫秒。
- `size`： 最多缓存对象的个数。
- `readOnly`： 是否只读，若配置可读写，则需要对应的实体类能够序列化。
- `blocking`： 若缓存中找不到对应的 key，是否会一直 blocking，直到有对应的数据进入缓存。

`cache-ref` 代表引用别的命名空间的 Cache 配置，两个命名空间的操作使用的是同一个 Cache。

二级缓存实验

接下来我们通过实验，了解 MyBatis 二级缓存在使用上的一些特点。

在本实验中，id 为 1 的学生名称初始化为点点。

实验 1

测试二级缓存效果，不提交事务， `sqlSession1` 查询完数据后， `sqlSession2` 相同的查询是否会从缓存中获取数据。

```
@Test
public void testCacheWithoutCommitOrClose() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);

        StudentMapper studentMapper =
sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 =
sqlSession2.getMapper(StudentMapper.class);

        System.out.println("studentMapper读取数据: " +
studentMapper.getStudentById(1));
        System.out.println("studentMapper2读取数据: " +
studentMapper2.getStudentById(1));
}
```

执行结果：

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
```

我们可以看到，当 `sqlsession` 没有调用 `commit()` 方法时，二级缓存并没有起到作用。

实验 2

测试二级缓存效果，当提交事务时， `sqlSession1` 查询完数据后， `sqlSession2` 相同的查询是否会从缓存中获取数据。

```
@Test
public void testCacheWithCommitOrClose() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);

        StudentMapper studentMapper =
sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 =
sqlSession2.getMapper(StudentMapper.class);

        System.out.println("studentMapper读取数据: " +
studentMapper.getStudentById(1));
        sqlSession1.commit();
        System.out.println("studentMapper2读取数据: " +
studentMapper2.getStudentById(1));
}
```

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
```

从图上可知， `sqlsession2` 的查询，使用了缓存，缓存的命中率是 0.5。

实验 3

测试 `update` 操作是否会刷新该 `namespace` 下的二级缓存。

```
@Test
public void testCacheWithUpdate() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);
        SqlSession sqlSession3 = factory.openSession(true);

        StudentMapper studentMapper =
sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 =
sqlSession2.getMapper(StudentMapper.class);
        StudentMapper studentMapper3 =
sqlSession3.getMapper(StudentMapper.class);

        System.out.println("studentMapper读取数据: " +
studentMapper.getStudentById(1));
        sqlSession1.commit();
        System.out.println("studentMapper2读取数据: " +
studentMapper2.getStudentById(1));

        studentMapper3.updateStudentName("方方",1);
        sqlSession3.commit();
        System.out.println("studentMapper2读取数据: " +
studentMapper2.getStudentById(1));
}
```

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - ==>  Preparing: UPDATE student SET name = ? WHERE id = ?
DEBUG [main] - ==> Parameters: 方方(String), 1(Integer)
DEBUG [main] - <==    Updates: 1
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.3333333333333333
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 方方, 16
DEBUG [main] - <==      Total: 1
studentMapper2读取数据: StudentEntity{id=1, name='方方', age=16, className='null'}
```

更新后，缓存被刷新，
之后相同的查询走了数据库

我们可以看到，在 `sqlSession3` 更新数据库，并提交事务后， `sqlsession2` 的 `StudentMapper namespace` 下的查询走了数据库，没有走 Cache。

实验 4

验证 MyBatis 的二级缓存不适应用于映射文件中存在多表查询的情况。

通常我们会为每个单表创建单独的映射文件，由于 MyBatis 的二级缓存是基于 `namespace` 的，多表查询语句所在的 `namspace` 无法感应到其他 `namespace` 中的语句对多表查询中涉及的表进行的修改，引发脏数据问题。

```
@Test
public void testCacheWithDiffererntNamespace() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);
        SqlSession sqlSession3 = factory.openSession(true);

        StudentMapper studentMapper =
sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 =
sqlSession2.getMapper(StudentMapper.class);
        ClassMapper classMapper = sqlSession3.getMapper(ClassMapper.class);

        System.out.println("studentMapper读取数据: " +
studentMapper.getStudentByIdWithClassInfo(1));
        sqlSession1.close();
        System.out.println("studentMapper2读取数据: " +
studentMapper2.getStudentByIdWithClassInfo(1));

        classMapper.updateClassName("特色一班",1);
        sqlSession3.commit();
        System.out.println("studentMapper2读取数据: " +
studentMapper2.getStudentByIdWithClassInfo(1));
}
```

执行结果:

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT s.id,s.name,s.age,class.name as className FROM classroom c JOIN student s ON c.student_id = s.id JOI
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age, className
TRACE [main] - <==        Row: 1, 点点, 16, 一班
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}
DEBUG [main] - ==>  Preparing: UPDATE class SET name = ? WHERE id = ?
DEBUG [main] - ==> Parameters: 特色一班(String), 1(Integer)
DEBUG [main] - <==    Updates: 1
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.6666666666666666
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}   缓存中存的仍然是旧的className
```

在这个实验中，我们引入了两张新的表，一张 class，一张 classroom。class 中保存了班级的 id 和班级名，classroom 中保存了班级 id 和学生 id。我们在 `StudentMapper` 中增加了一个查询方法 `getStudentByIdWithClassInfo`，用于查询学生所在的班级，涉及到多表查询。在 `ClassMapper` 中添加了 `updateClassName`，根据班级 id 更新班级名的操作。

当 `sqlsession1` 的 `studentmapper` 查询数据后，二级缓存生效。保存在 StudentMapper 的 namespace 下的 cache 中。当 `sqlSession3` 的 `classMapper` 的 `updateClassName` 方法对 class 表进行更新时， `updateClassName` 不属于 `StudentMapper` 的 `namespace`，所以 `StudentMapper` 下的 cache 没有感应到变化，没有刷新缓存。当 `StudentMapper` 中同样的查询再次发起时，从缓存中读取了脏数据。

实验 5

为了解决实验 4 的问题呢，可以使用 Cache ref，让 `ClassMapper` 引用 `StudenMapper` 命名空间，这样两个映射文件对应的 SQL 操作都使用的是同一块缓存了。

执行结果：

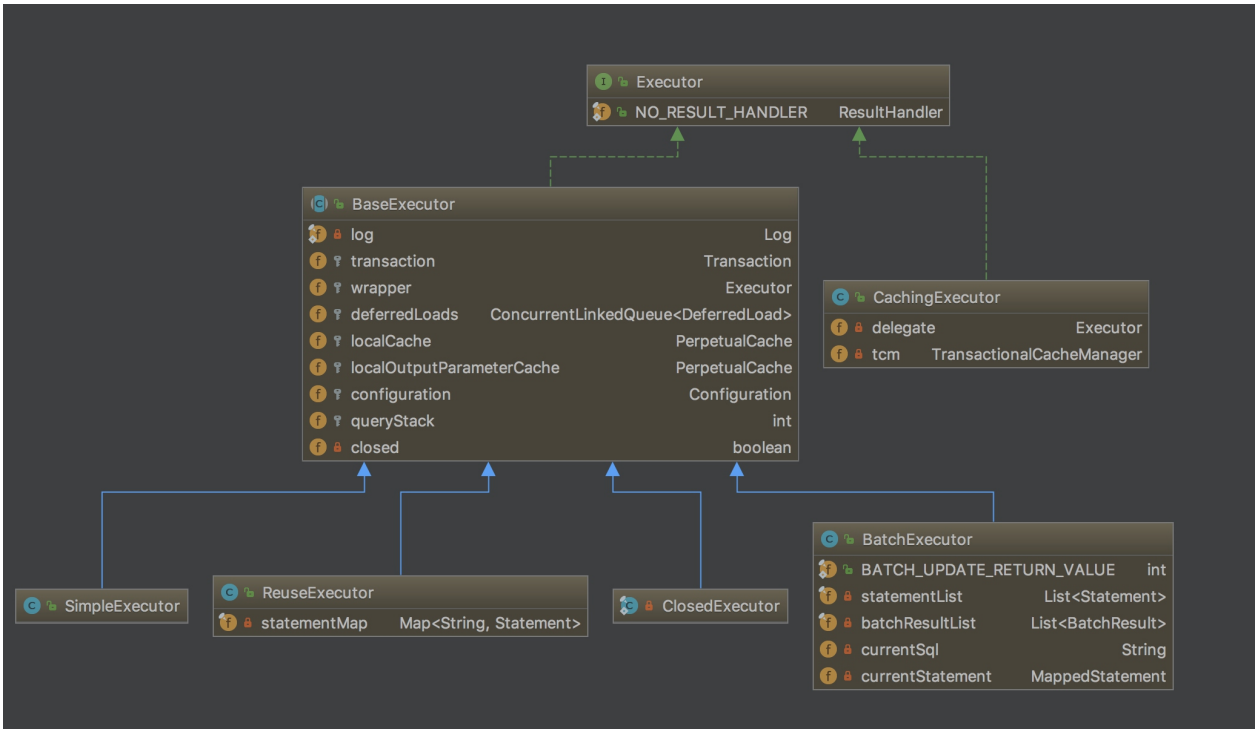
```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT s.id,s.name,s.age,class.name as className FROM classroom c JOIN student s ON c.student_id = s.id JOIN
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age, className
TRACE [main] - <==        Row: 1, 点点, 16, 一班
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='一班'}
DEBUG [main] - ==>  Preparing: UPDATE class SET name = ? WHERE id = ?
DEBUG [main] - ==> Parameters: 特色一班(String), 1(Integer)
DEBUG [main] - <==    Updates: 1
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.3333333333333333
DEBUG [main] - ==>  Preparing: SELECT s.id,s.name,s.age,class.name as className FROM classroom c JOIN student s ON c.student_id = s.id JOIN
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age, className
TRACE [main] - <==        Row: 1, 点点, 16, 特色一班
DEBUG [main] - <==      Total: 1
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='特色一班'}
```

刷新了共同的缓存，后续的select走了数据库

不过这样做的后果是，缓存的粒度变粗了，多个 `Mapper namespace` 下的所有操作都会对缓存使用造成影响。

二级缓存源码分析

MyBatis 二级缓存的工作流程和前文提到的一级缓存类似，只是在一级缓存处理前，用 `CachingExecutor` 装饰了 `BaseExecutor` 的子类，在委托具体职责给 `delegate` 之前，实现了二级缓存的查询和写入功能，具体类关系图如下图所示。



源码分析

源码分析从 `CachingExecutor` 的 `query` 方法展开，源代码走读过程中涉及到的知识点较多，不能一一详细讲解，读者朋友可以自行查询相关资料来学习。

`CachingExecutor` 的 `query` 方法，首先会从 `MappedStatement` 中获得在配置初始化时赋予的 Cache。

```
Cache cache = ms.getCache();
```

本质上是装饰器模式的使用，具体的装饰链是：

SynchronizedCache -> LoggingCache -> SerializedCache -> LruCache -> PerpetualCache。

以下是具体这些 Cache 实现类的介绍，他们的组合为 Cache 赋予了不同的能力。

- `SynchronizedCache`： 同步 Cache，实现比较简单，直接使用 synchronized 修饰方法。
- `LoggingCache`： 日志功能，装饰类，用于记录缓存的命中率，如果开启了 DEBUG 模式，则会输出命中率日志。
- `SerializedCache`： 序列化功能，将值序列化后存到缓存中。该功能用于缓存返回一份实例的 Copy，用于保存线程安全。
- `LruCache`： 采用了 Lru 算法的 Cache 实现，移除最近最少使用的 Key/Value。
- `PerpetualCache`： 作为为最基础的缓存类，底层实现比较简单，直接使用了 HashMap。

然后是判断是否需要刷新缓存，代码如下所示：

```
flushCacheIfRequired(ms);
```

在默认的设置中 `SELECT` 语句不会刷新缓存， `insert/update/delte` 会刷新缓存。进入该方法。代码如下所示：

```
private void flushCacheIfRequired(MappedStatement ms) {
    Cache cache = ms.getCache();
    if (cache != null && ms.isFlushCacheRequired()) {
      tcm.clear(cache);
    }
}
```

MyBatis 的 `CachingExecutor` 持有了 `TransactionalCacheManager` ，即上述代码中的 tcm。

TransactionalCacheManager 中持有了一个 Map，代码如下所示：

```
private Map transactionalCaches = new HashMap();
```

这个 Map 保存了 Cache 和用 TransactionalCache 包装后的 Cache 的映射关系。

TransactionalCache 实现了 Cache 接口， CachingExecutor 会默认使用他包装初始生成的 Cache，作用是如果事务提交，对缓存的操作才会生效，如果事务回滚或者不提交事务，则不对缓存产生影响。

在 TransactionalCache 的 clear，有以下两句。清空了需要在提交时加入缓存的列表，同时设定提交时清空缓存，代码如下所示：

```
@Override
public void clear() {
    clearOnCommit = true;
    entriesToAddOnCommit.clear();
}
```

CachingExecutor 继续往下走， ensureNoOutParams 主要是用来处理存储过程的，暂时不用考虑。

```
if (ms.isUseCache() && resultHandler == null) {
    ensureNoOutParams(ms, parameterObject, boundSql);
```

之后会尝试从 tcm 中获取缓存的列表。

```
List list = (List) tcm.getObject(cache, key);
```

在 getObject 方法中，会把获取值的职责一路传递，最终到 PerpetualCache 。如果没有查到，会把 key 加入 Miss 集合，这个主要是为了统计命中率。

```
Object object = delegate.getObject(key);
if (object == null) {
    entriesMissedInCache.add(key);
}
```

CachingExecutor 继续往下走，如果查询到数据，则调用 tcm.putObject 方法，往缓存中放入值。

```
if (list == null) {
    list = delegate. query(ms, parameterObject, rowBounds, resultHandler, key,
boundSql);
    tcm.putObject(cache, key, list); // issue #578 and #116
}
```

tcm 的 `put` 方法也不是直接操作缓存，只是在把这次的数据和 key 放入待提交的 Map 中。

```
@Override
public void putObject(Object key, Object object) {
    entriesToAddOnCommit.put(key, object);
}
```

从以上的代码分析中，我们可以明白，如果不调用 `commit` 方法的话，由于 `TranscationalCache` 的作用，并不会对二级缓存造成直接的影响。因此我们看看 `Sqlsession` 的 `commit` 方法中做了什么。代码如下所示：

```
@Override
public void commit(boolean force) {
    try {
      executor.commit(isCommitOrRollbackRequired(force));
```

因为我们使用了 CachingExecutor，首先会进入 CachingExecutor 实现的 commit 方法。

```
@Override
public void commit(boolean required) throws SQLException {
    delegate.commit(required);
    tcm.commit();
}
```

会把具体 commit 的职责委托给包装的 `Executor` 。主要是看下 `tcm.commit()` ，tcm 最终又会调用到 `TrancationalCache` 。

```
public void commit() {
    if (clearOnCommit) {
      delegate.clear();
    }
    flushPendingEntries();
    reset();
}
```

看到这里的 `clearOnCommit` 就想起刚才 `TrancationalCache` 的 `clear` 方法设置的标志位，真正的清理 Cache 是放到这里来进行的。具体清理的职责委托给了包装的 Cache 类。之后进入 `flushPendingEntries` 方法。代码如下所示：

```
private void flushPendingEntries() {
    for (Map.Entry entry : entriesToAddOnCommit.entrySet()) {
      delegate.putObject(entry.getKey(), entry.getValue());
    }
    ..............

}
```

在 `flushPending` Entries 中，将待提交的 Map 进行循环处理，委托给包装的 Cache 类，进行 `putObject` 的操作。

后续的查询操作会重复执行这套流程。如果是 `insert|update|delete` 的话，会统一进入 `CachingExecutor` 的 `update` 方法，其中调用了这个函数，代码如下所示：

```
private void flushCacheIfRequired(MappedStatement ms)
```

在二级缓存执行流程后就会进入一级缓存的执行流程，因此不再赘述。

总结

1. MyBatis 的二级缓存相对于一级缓存来说，实现了 `SqlSession` 之间缓存数据的共享，同时粒度更加的细，能够到 `namespace` 级别，通过 Cache 接口实现类不同的组合，对 Cache 的可控性也更强。
2. MyBatis 在多表查询时，极大可能会出现脏数据，有设计上的缺陷，安全使用二级缓存的条件比较苛刻。
3. 在分布式环境下，由于默认的 MyBatis Cache 实现都是基于本地的，分布式环境下必然会出现读取到脏数据，需要使用集中式缓存将 MyBatis 的 Cache 接口实现，有一定的开发成本，直接使用 Redis、Memcached 等分布式缓存可能成本更低，安全性也更高。

全文总结

本文对介绍了 MyBatis 一二级缓存的基本概念，并从应用及源码的角度对 MyBatis 的缓存机制进行了分析。最后对 MyBatis 缓存机制做了一定的总结，个人建议 MyBatis 缓存特性在生产环境中进行关闭，单纯作为一个 ORM 框架使用可能更为合适。

Mybaties Executor 是啥

Mybatis 中所有的 Mapper 语句的执行都是通过 Executor 进行的，Executor 是 Mybatis 的一个核心接口，其定义如下。从其定义的接口方法我们可以看出，对应的增删改语句是通过 Executor 接口的 update 方法进行的，查询是通过 query 方法进行的。虽然 Executor 接口的实现类有 BaseExecutor 和 CachingExecutor，而 BaseExecutor 的子类又有 SimpleExecutor、ReuseExecutor 和 BatchExecutor，但 BaseExecutor 是一个抽象类，其只实现了一些公共的封装，而把真正的核心实现都通过方法抽象出来给子类实现，如 doUpdate()、doQuery()；CachingExecutor 只是在 Executor 的基础上加入了缓存的功能，底层还是通过 Executor 调用的，所以真正有作用的 Executor 只有 SimpleExecutor、ReuseExecutor 和 BatchExecutor。它们都是自己实现的 Executor 核心功能，没有借助任何其它的 Executor 实现，它们是实现不同也就注定了它们的功能也是不一样的。Executor 是跟 SqlSession 绑定在一起的，每一个 SqlSession 都拥有一个新的 Executor 对象，由 Configuration 创建。

Mybaties \$和#的区别

动态 sql 是 mybatis 的主要特性之一，在 mapper 中定义的参数传到 xml 中之后，在查询之前 mybatis 会对其进行动态解析。mybatis 为我们提供了两种支持动态 sql 的语法：#{} 以及 \${} 。

1、#相当于对数据 加上 双引号，\$相当于直接显示数据。

2、#{}： 根据参数的类型进行处理，比如传入 String 类型，则会为参数加上双引号。#{} 传参在进行 SQL 预编译时，会把参数部分用一个占位符？代替，这样可以防止 SQL 注入。 3、${}： 将参数取出不做任何处理，直接放入语句中，就是简单的字符串替换，并且该参数会参加SQL的预编译，需要手动过滤参数防止 SQL注入。 4、因此 mybatis 中优先使用 #{}；当需要动态传入 表名或列名时，再考虑使用 ${} ， 比较特殊，他的应用场景是需要动态传入表名或列名时使用，MyBatis 排序时使用 orderby 动态参数时需要注意，用{} 比较特殊， 他的应用场景是 需要动态传入 表名或列名时使用，MyBatis 排序时使用 order by 动态参数时需要注意，用比较特殊，他的应用场景是需要动态传入表名或列名时使用，MyBatis 排序时使用 order by 动态参数时需要注意，用而不是#。

例如： 1、#对传入的参数视为字符串，也就是它会预编译，select _ from user where user_name=\${rookie}，比如我传一个 rookie，那么传过来就是 select _ from user where user_name = ‘rookie’；

2、$不会将传入的值进行预编译， select * from user where user_name= ${rookie} ,那么传过来的 sql 就是：select * from user where user_name=rookie;

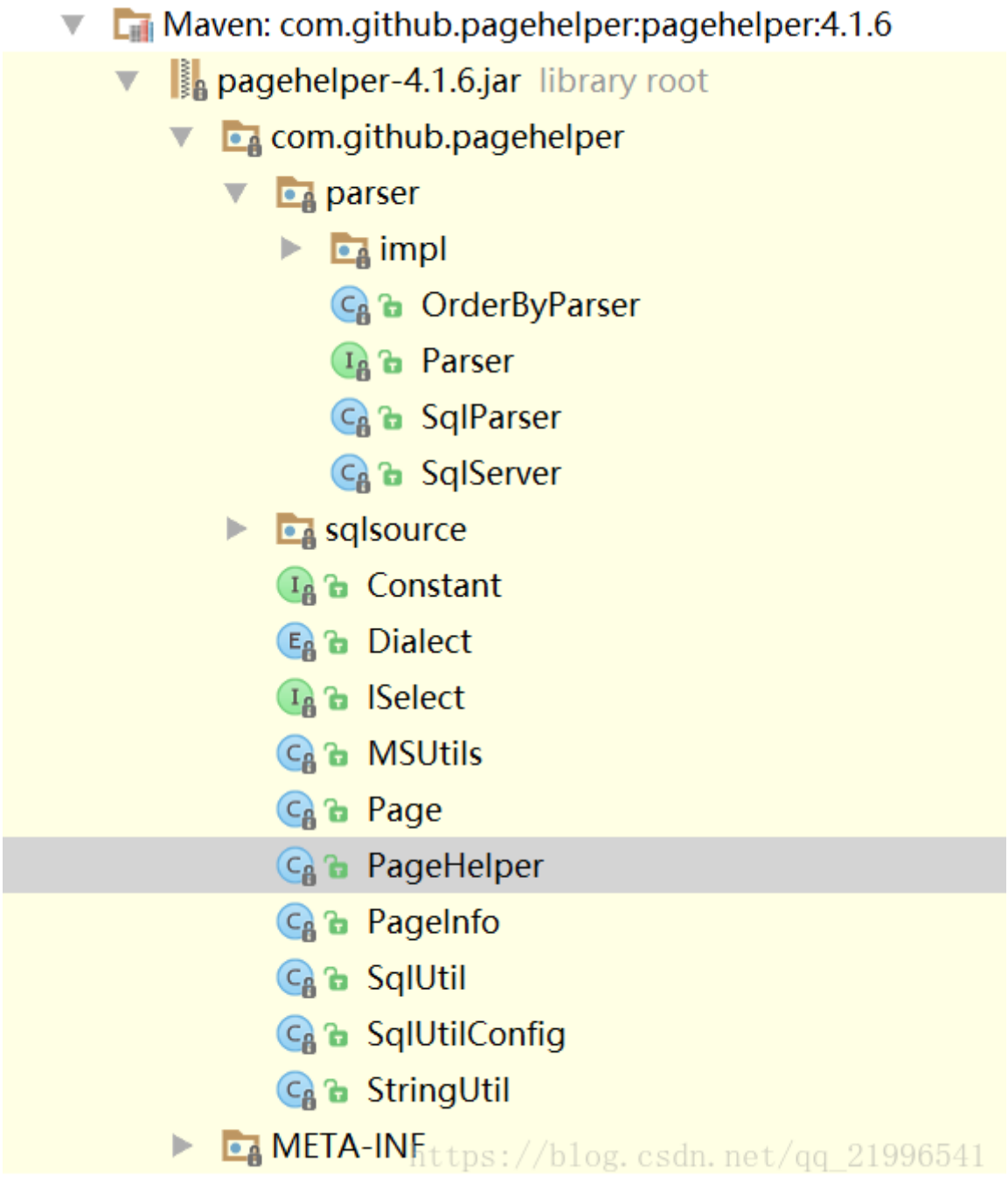
3、#优势在于它能很大程度防止 sql 注入，$ 不行。比如：用户进行一个登录操作，后台 sql验证式样的： select * from user where user_name=#{name} and password = #{pwd}，如果前台传来的用户名是”rookie”，密码是 “1 or 2”，用#的方式就不会出现sql注入，换成用$ 方式，sql 语句就变成了 select * from user where user_name=rookie and password = 1 or 2。这样的话就形成了 sql 注入。

Mybaties 分页是怎么实现的 (还是学一下 pagehelp)

框架的分页组件使用的是 pagehelper，对其我也是早有耳闻，但是也是第一次接触（ps: 工作 1 年，一直使用的是普元封装好的前端框架）。

要是用 pagehelper，首先 maven 项目，要引入

```
<dependency>
   <groupId>com.github.pagehelper</groupId>
   <artifactId>pagehelper</artifactId>
   <version>4.1.6</version>
</dependency>
```



前端使用的 bootstrap 分页插件，这里不再赘述，直接切入正题，持久层框架使用的是 mybatis，分页插件的 URL，指向后台的 controller，

```
undefined
```

```
@ResponseBody
@RequestMapping("/testPage")
public String testPage(HttpServletRequest request) {
   Page page = PageUtil.pageStart(request);
   List list = prodMapper.selectAll();
   JSONObject rst = PageUtil.pageEnd(request, page, list);
   return rst.toString();
}
```

这是 controller 的代码，当时就产生一个疑问，为啥在这个 pageStart 后面查询就能够实现分页呢？可以查出想要的 10 条分页数据，而去掉则是全部查询出来的 84 条记录。

带着问题，我开始了我的 debug 之旅，首先，我跟进 pageStart 方法，这个 PageUtil 类是公司大佬封装的一个工具类，代码如下：

```
public class PageUtil {

    public enum CNT{
        total,
        res
    }

    public static <E> Page<E> pageStart(HttpServletRequest request){
        return pageStart(request,null);
    }

    /**
     *
     * @param request
     * @param pageSize 每页显示条数
     * @param orderBy 写入 需要排序的 字段名 如:  product_id desc
     * @return
     */
    public static <E> Page<E> pageStart(HttpServletRequest request,String
orderBy){

        int pageon=getPageon(request);
        int pageSize=getpageSize(request);
        Page<E> page=PageHelper.startPage(pageon, pageSize);
        if(!StringUtils.isEmpty(orderBy)){
            PageHelper.orderBy(orderBy);
        }

        return page;
    }

    private static int getPageon(HttpServletRequest request){
        String pageonStr=request.getParameter("pageon");
        int pageon;
        if(StringUtils.isEmpty(pageonStr)){
            pageon=1;
        }else{
            pageon=Integer.parseInt(pageonStr);
        }
        return pageon;
    }

    private static int getpageSize(HttpServletRequest request){
        String pageSizeStr=request.getParameter("pageSize");
        int pageSize;
        if(StringUtils.isEmpty(pageSizeStr)){
            pageSize=1;
        }else{
            pageSize=Integer.parseInt(pageSizeStr);
        }
        return pageSize;
    }

    /**
```

```
     *
     * @param request
     * @param page
     * @param list
     * @param elName 页面显示所引用的变量名
     */
    public static JSONObject pageEnd(HttpServletRequest request, Page<?>
page,List<?> list){
        JSONObject rstPage=new JSONObject();
        rstPage.put(CNT.total.toString(), page.getTotal());
        rstPage.put(CNT.res.toString(), list);
        return rstPage;
    }


}
```

可以看到，pageStart 有两个方法重载，进入方法后，获取了前端页面传递的 pageon、pageSize 两个参数，分别表示当前页面和每页显示多少条，然后调用了 PageHelper.startPage，接着跟进此方法，发现也是一对方法重载，没关系，往下看

undefined

undefined

```
/**
 * 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时用默认配
置
 */
public static <E> Page<E> startPage(int pageNum, int pageSize, boolean count,
Boolean reasonable, Boolean pageSizeZero) {
    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 = SqlUtil.getLocalPage();
    if (oldPage != null && oldPage.isOrderByOnly()) {
        page.setOrderBy(oldPage.getOrderBy());
    }
    SqlUtil.setLocalPage(page);
    return page;
}
```

上面的方法才是真正分页调用的地方，原来是对传入参数的赋值，赋给 Page 这个类，继续，发现 getLocalPage 和 setLoaclPage 这两个方法，很可疑，跟进，看看他到底做了啥，

```
public static <T> Page<T> getLocalPage() {
    return LOCAL_PAGE.get();
}

public static void setLocalPage(Page page) {
    LOCAL_PAGE.set(page);
}
private static final ThreadLocal<Page> LOCAL_PAGE = new ThreadLocal<Page>();

private static final ThreadLocal LOCAL_PAGE = new ThreadLocal();
```

哦，有点明白了，原来就是赋值保存到本地线程变量里面，这个 ThreadLocal 是何方神圣，居然有这么厉害，所以查阅了相关博客，链接：https://blog.csdn.net/u013521220/article/details/73604917 (https://blog.csdn.net/u013521220/article/details/73604917)，个人简单理解大概意思是这个类有4个方法，set,get,remove,initialValue,可以使每个线程独立开来，参数互不影响，里面保存当前线程的变量副本。

OK，那这个地方就是保存了当前分页线程的 Page 参数的变量。有赋值就有取值，那么在下面的分页过程中，肯定在哪边取到了这个 threadLocal 的 page 参数。

好，执行完 startPage，下面就是执行了 mybatis 的 SQL 语句，

```
<select id="selectAll" resultMap="BaseResultMap">
  select SEQ_ID, PRODUCT_ID, PRODUCT_NAME, PRODUCT_DESC, CREATE_TIME,
EFFECT_TIME,
  EXPIRE_TIME, PRODUCT_STATUS, PROVINCE_CODE, REGION_CODE, CHANGE_TIME,
OP_OPERATOR_ID,
  PRODUCT_SYSTEM, PRODUCT_CODE
  from PM_PRODUCT
</select>
```

SQL 语句很简单，就是简单的查询出 PM_PRODUCT 的全部记录，那到底是哪边做了拦截吗？带着这个疑问，我跟进了代码，

发现进入了 mybatis 的 MapperPoxy 这个代理类，

```
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
  if (Object.class.equals(method.getDeclaringClass())) {
    try {
      return method.invoke(this, args);
    } catch (Throwable t) {
      throw ExceptionUtil.unwrapThrowable(t);
    }
  }
  final MapperMethod mapperMethod = cachedMapperMethod(method);
  return mapperMethod.execute(sqlSession, args);
}
```

最后执行的 execute 方法，再次跟进，进入 MapperMethod 这个类的 execute 方法，

```
public Object execute(SqlSession sqlSession, Object[] args) {
  Object result;
  if (SqlCommandType.INSERT == command.getType()) {
    Object param = method.convertArgsToSqlCommandParam(args);
    result = rowCountResult(sqlSession.insert(command.getName(), param));
  } else if (SqlCommandType.UPDATE == command.getType()) {
    Object param = method.convertArgsToSqlCommandParam(args);
    result = rowCountResult(sqlSession.update(command.getName(), param));
  } else if (SqlCommandType.DELETE == command.getType()) {
    Object param = method.convertArgsToSqlCommandParam(args);
    result = rowCountResult(sqlSession.delete(command.getName(), param));
  } else if (SqlCommandType.SELECT == command.getType()) {
    if (method.returnsVoid() && method.hasResultHandler()) {
      executeWithResultHandler(sqlSession, args);
      result = null;
    } else if (method.returnsMany()) {
      result = executeForMany(sqlSession, args);
    } else if (method.returnsMap()) {
      result = executeForMap(sqlSession, args);
    } else if (method.returnsCursor()) {
      result = executeForCursor(sqlSession, args);
    } else {
      Object param = method.convertArgsToSqlCommandParam(args);
      result = sqlSession.selectOne(command.getName(), param);
    }
  } else if (SqlCommandType.FLUSH == command.getType()) {
      result = sqlSession.flushStatements();
  } else {
    throw new BindingException("Unknown execution method for: " +
command.getName());
  }
  if (result == null && method.getReturnType().isPrimitive() &&
!method.returnsVoid()) {
    throw new BindingException("Mapper method '" + command.getName()
        + " attempted to return null from a method with a primitive return type
(" + method.getReturnType() + ").");
  }
  return result;
}
```

由于执行的是 select 操作，并且查询出多条，所以就到了 executeForMany 这个方法中，后面继续跟进代码 SqlSessionTemplate,DefaultSqlSession（不再赘述），最后可以看到代码进入了 Plugin 这个类的 invoke 方法中，

```
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
  try {
    Set methods = signatureMap.get(method.getDeclaringClass());
    if (methods != null && methods.contains(method)) {
      return interceptor.intercept(new Invocation(target, method, args));
    }
    return method.invoke(target, args);
  } catch (Exception e) {
    throw ExceptionUtil.unwrapThrowable(e);
  }
}
```

这下明白了，interceptor 是 mybatis 的拦截器，而 PageHelper 这个类就实现了 interceptor 接口，调用其中的 intercept 方法。

```
/**
 * 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);
    }
}
```

```
/**
 * 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) {
        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 && rowBounds ==
RowBounds.DEFAULT)) {
        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);
        }
        return doProcessPage(invocation, page, args);
    }
}
```

最终我在 SqlUtil 中的_processPage 方法中找到了，getPage 这句话，getLocalPage 就将保存在 ThreadLocal 中的 Page 变量取了出来，这下一切一目了然了，

```
public Page getPage(Object[] args) {
    Page page = getLocalPage();
    if (page == null || page.isOrderByOnly()) {
        Page oldPage = page;
        //这种情况下,page.isOrderByOnly()必然为true，所以不用写到条件中
        if ((args[2] == null || args[2] == RowBounds.DEFAULT) && page != null) {
            return oldPage;
        }
```

https://blog.csdn.net/qq_21996541

跟进代码，发现进入了 doProcessPage 方法，通过反射机制，首先查询出数据总数量，然后进行分页 SQL 的拼装，MappedStatement 的 getBoundSql

```
public BoundSql getBoundSql(Object parameterObject) {
  BoundSql boundSql = sqlSource.getBoundSql(parameterObject);
  List parameterMappings = boundSql.getParameterMappings();
  if (parameterMappings == null || parameterMappings.isEmpty()) {
    boundSql = new BoundSql(configuration, boundSql.getSql(),
parameterMap.getParameterMappings(), parameterObject);
  }

  // check for nested result maps in parameter mappings (issue #30)
for (ParameterMapping pm : boundSql.getParameterMappings()) {
    String rmId = pm.getResultMapId();
    if (rmId != null) {
      ResultMap rm = configuration.getResultMap(rmId);
      if (rm != null) {
        hasNestedResultMaps |= rm.hasNestedResultMaps();
      }
    }
  }

  return boundSql;
}
```

继续，跟进代码，发现，最终分页的查询，调到了 PageStaticSqlSource 类的 getPageBoundSql 中，

```
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().getPageParameterMapping(configuration,
original.getBoundSql(parameterObject)), parameterObject);
}
```

进入 getPageSql 这个方法，发现，进入了 OracleParser 类中（还有很多其他的 Parser，适用于不同的数据库），

```
public String getPageSql(String sql) {
    StringBuilder sqlBuilder = new StringBuilder(sql.length() + 120);
    sqlBuilder.append("select * from ( select tmp_page.*, rownum row_id from (
");
    sqlBuilder.append(sql);
    sqlBuilder.append(" ) tmp_page where rownum  ?");
    return sqlBuilder.toString();
}
```

终于，原来分页的 SQL 是在这里拼装起来的。

总结：PageHelper 首先将前端传递的参数保存到 page 这个对象中，接着将 page 的副本存放入 ThreadLoacl 中，这样可以保证分页的时候，参数互不影响，接着利用了 mybatis 提供的拦截器，取得 ThreadLocal 的值，重新拼装分页 SQL，完成分页。

PS：DEBUG 用的好不好，对看框架源码很有很大的影响。

Mybaties 自定义插件怎么弄

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：

Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed) –执行 sql

ParameterHandler (getParameterObject, setParameters) –获取、设置参数

ResultSetHandler (handleResultSets, handleOutputParameters) –处理结果集

StatementHandler (prepare, parameterize, batch, update, query) –记录 sql

这里需要注意的是，这 4 个类型是固定的，里面的方法也是固定的，不能再被改变的，具体的信息，可以相关的类（Executor.class、ParameterHandler .class、ResultSetHandler .class、StatementHandler .class）查看。 先定义一个插件拦截器

```
package com.drafire.testall.interceptor;

import org.apache.ibatis.executor.Executor;
import org.apache.ibatis.mapping.MappedStatement;
import org.apache.ibatis.plugin.*;

import java.util.Properties;

/**
 * mybatis 自定义插件
 */
@Intercepts(value = {@Signature(
        type= Executor.class,                                   //这里对应4个类
        method = "update",                                      //这里对应4个类里面的参数
        args = {MappedStatement.class,Object.class})})      //这里的参数类型，是对应4个类中的各种方法的参数。如果方法没有参数，这里直接写{}就可以了
public class DrafirePlugin implements Interceptor {
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        System.out.println("mybatis插件打印了乐乐");
        return invocation.proceed();
    }

    @Override
    public Object plugin(Object target) {
        return Plugin.wrap(target, this);
    }

    @Override
    public void setProperties(Properties properties) {

    }
}
```

配置 mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <typeHandlers>
        <typeHandler
handler="com.drafire.testall.handler.DrafireStringHandler"></typeHandler>
    </typeHandlers>

    <plugins>
        <plugin interceptor="com.drafire.testall.interceptor.DrafirePlugin">
        </plugin>
    </plugins>

    <environments default="development">
        <environment id="sell">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${ds.sell.driverClassName}"/>
                <property name="url" value="${ds.sell.url}"/>
                <property name="username" value="${ds.sell.username}"/>
                <property name="password" value="${ds.sell.password}"/>
            </dataSource>
        </environment>

        <environment id="bank">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${ds.bank.driverClassName}">
</property>
                <property name="url" value="${ds.bank.url}"></property>
                <property name="username" value="${ds.bank.username}">
</property>
                <property name="password" value="${ds.bank.password}">
</property>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <!--<mapper resource="org/mybatis/example/BlogMapper.xml"/>-->
    </mappers>
</configuration>
```

Mybaties 动态代理

动态代理就是对目标对象的方法做不同的代理实现，通俗点说就是在一个方法的执行前后加上对应的处理逻辑。例如 spring 的事务处理，在插入一条数据的前面开启事务，插入完成后提交事务。

现在我们解析一下动态代理底层的执行方式，直接上代码：

```
public interface Parent {
    public void create();
}

public class Sub implements Parent {
    public void create() {
        System.out.println("sub create");
    }
}

public class Log {
    public static void before(){
        System.out.println("before");
    }

    public static void after(){
        System.out.println("after");
    }
}

public class ProxyHandler implements InvocationHandler {

    public Object obj;
    public ProxyHandler(Object obj) {
        this.obj=obj;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        Log.before();
        Object result=method.invoke(obj,args);
        Log.after();
        return result;
    }

    public static <t> T execute(Class<t> clazz,T t){
        ClassLoader classLoader = clazz.getClassLoader();
        ProxyHandler handler=new ProxyHandler(t);
        Class<?>[] classes=new Class[]{clazz};
        return (T) Proxy.newProxyInstance(classLoader,classes,handler);
    }
}

public class Main {
    public static void main(String[] args) throws Exception {
            Parent p = new Sub();
            Parent proxy = ProxyHandler.execute(Parent.class, p);
            proxy.create();
    }

}</t></t>
```

我们定义了一个 Parent 接口和它的实现类 Sub. Log 是一个代理类，ProxyHandler 是封装的一个通用代理处理器。

我们直接从 Proxy.newProxyInstance(classLoader,classes,handler) 这一段开始看起，底层 它首先是通过传入的类加载器将生成的 class 文件加载到 jvm 中, 然后创建了一个代理对象 com.sun.proxy.$Proxy0，$Proxy0 的大致代码如下：

```
public class $Proxy0 implements Parent{
    InvocationHandler h;

    public $Proxy0(InvocationHandler h){
        this.h=h;
    }
    @Override
    public void create() {
        try{
        Method m=Parent.class.getMethod("create");
        h.invoke(this,m,null);
        }catch (Throwable e){
            e.printStackTrace();
        }
    }
}
```

代理类其实就是 实现了目标类的接口，然后再方法里面调用 InvocationHandler 的实现类，最后在 invoke 方法中实现对目标方法的代理。

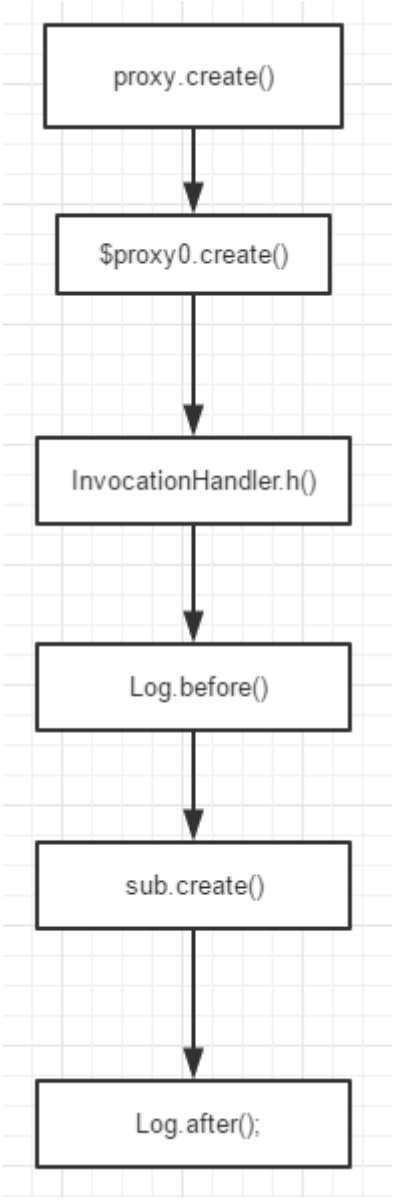
invoke(Object proxy, Method method, Object[] args) 方法中的三个参数:

proxy : 动态生成的代理类，即 \$Proxy0

method : 目标对象的方法，即 create

args : 目标方法的参数值

最后我们看一下 proxy.create() 的运行流程:



我们可以看到底层其实是创建一个代理对象，利用多态的方式，通过注入 InvocationHandler ，再采用反射的方式调用目标方法完成动态代理的实现

动态代理还有一种实现方式就是 cglib，它是可以代理 class 类，java 自带的只能代理 interface 接口，它的实现方式是采用继承的方式生成 proxy 代理类，其它流程基本一样

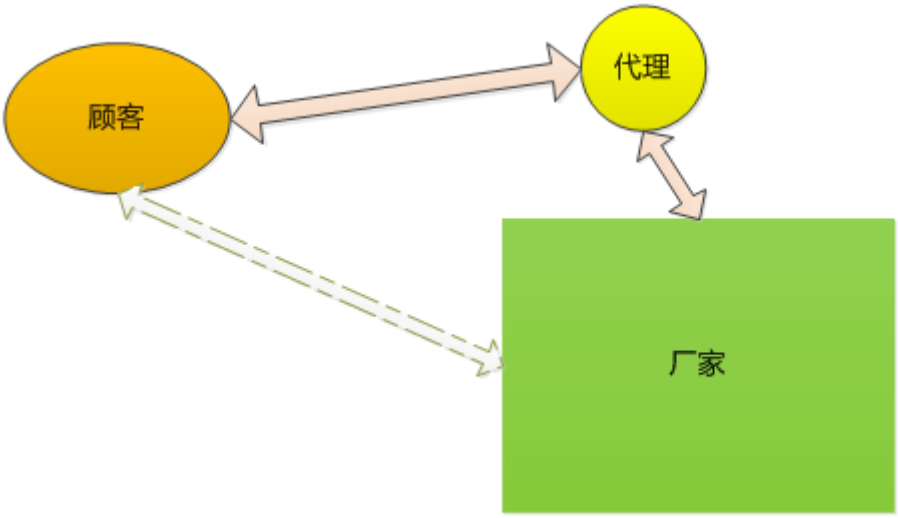
总结：

平时我们更多的是做的 crud 之类的业务开发，习惯了编译运行的这种方式，而动态代理打破了这种思维定势，采用了运行时动态生成代理类加载运行的实现方式，我们可以多多理解这种思考方式

代理

代理是英文 Proxy 翻译过来的。我们在生活中见到过的代理，大概最常见的就是朋友圈中卖面膜的同学了。

她们从厂家拿货，然后在朋友圈中宣传，然后卖给熟人。



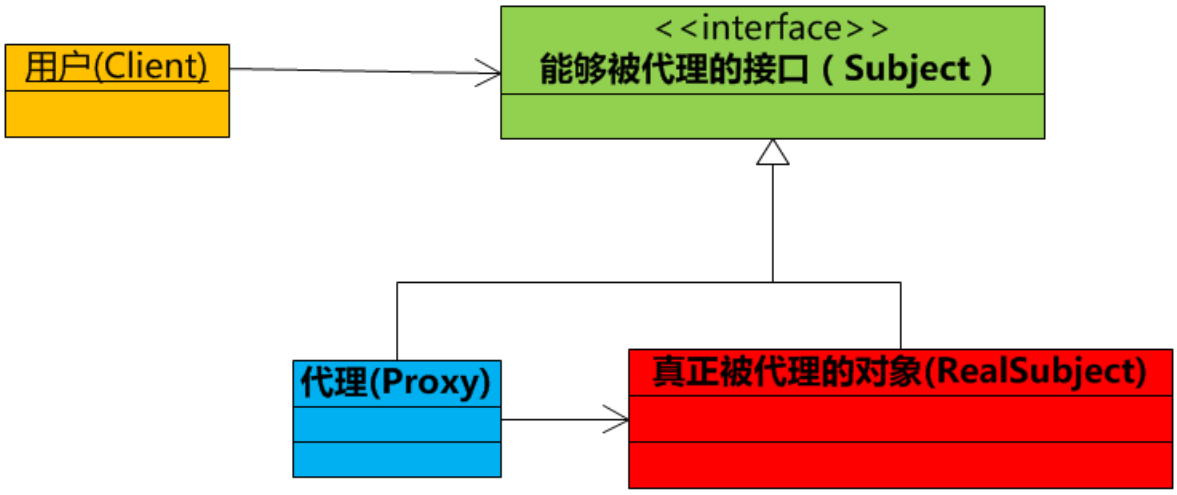
按理说，顾客可以直接从厂家购买产品，但是现实生活中，很少有这样的销售模式。一般都是厂家委托给代理商进行销售，顾客跟代理商打交道，而不直接与产品实际生产者进行关联。

所以，代理就有一种中间人的味道。

接下来，我们说说软件中的代理模式。

；代理模式

代理模式是面向对象编程中比较常见的设计模式。



这是常见代理模式常见的 UML 示意图。

需要注意的有下面几点：

1. 用户只关心接口功能，而不在乎谁提供了功能。上图中接口是 Subject。
2. 接口真正实现者是上图的 RealSubject，但是它不与用户直接接触，而是通过代理。

3. 代理就是上图中的 Proxy，由于它实现了 Subject 接口，所以它能够直接与用户接触。
4. 用户调用 Proxy 的时候，Proxy 内部调用了 RealSubject。所以，Proxy 是中介者，它可以增强 RealSubject 操作。

如果难于理解的话，我用事例说明好了。值得注意的是，代理可以分为静态代理和动态代理两种。先从静态代理讲起。

静态代理

我们平常去电影院看电影的时候，在电影开始的阶段是不是经常会放广告呢?

电影是电影公司委托给影院进行播放的，但是影院可以在播放电影的时候，产生一些自己的经济收益，比如卖爆米花、可乐等，然后在影片开始结束时播放一些广告。

现在用代码来进行模拟。

首先得有一个接口，通用的接口是代理模式实现的基础。这个接口我们命名为 Movie，代表电影播放的能力。

```
package com.frank.test;

public interface Movie {
    void play();
}
```

然后，我们要有一个真正的实现这个 Movie 接口的类，和一个只是实现接口的代理类。

```
package com.frank.test;

public class RealMovie implements Movie {

    @Override
    public void play() {

        System.out.println("您正在观看电影 《肖申克的救赎》");
    }

}
```

这个表示真正的影片。它实现了 Movie 接口，play() 方法调用时，影片就开始播放。那么 Proxy 代理呢?

```
package com.frank.test;

public class Cinema implements Movie {

    RealMovie movie;

    public Cinema(RealMovie movie) {
        super();
        this.movie = movie;
    }

    @Override
    public void play() {

        guanggao(true);

        movie.play();

        guanggao(false);
    }

    public void guanggao(boolean isStart){
        if ( isStart ) {
            System.out.println("电影马上开始了，爆米花、可乐、口香糖9.8折，快来买
啊！");
        } else {
            System.out.println("电影马上结束了，爆米花、可乐、口香糖9.8折，买回家
吃吧！");
        }
    }

}
```

Cinema 就是 Proxy 代理对象，它有一个 play() 方法。不过调用 play() 方法时，它进行了一些相关利益的处理，那就是广告。现在，我们编写测试代码。

```
package com.frank.test;

public class ProxyTest {

    public static void main(String[] args) {

        RealMovie realmovie = new RealMovie();

        Movie movie = new Cinema(realmovie);

        movie.play();

    }

}
```

然后观察结果：

```
&#x7535;&#x5F71;&#x9A6C;&#x4E0A;&#x5F00;&#x59CB;&#x4E86;&#xFF0C;&#x7206;&#x7C73
;&#x82B1;&#x3001;&#x53EF;&#x4E50;&#x3001;&#x53E3;&#x9999;&#x7CD6;9.8&#x6298;&#x
FF0C;&#x5FEB;&#x6765;&#x4E70;&#x554A;&#xFF01;
&#x60A8;&#x6B63;&#x5728;&#x89C2;&#x770B;&#x7535;&#x5F71;
&#x300A;&#x8096;&#x7533;&#x514B;&#x7684;&#x6551;&#x8D4E;&#x300B;
&#x7535;&#x5F71;&#x9A6C;&#x4E0A;&#x7ED3;&#x675F;&#x4E86;&#xFF0C;&#x7206;&#x7C73
;&#x82B1;&#x3001;&#x53EF;&#x4E50;&#x3001;&#x53E3;&#x9999;&#x7CD6;9.8&#x6298;&#x
FF0C;&#x4E70;&#x56DE;&#x5BB6;&#x5403;&#x5427;&#xFF01;
```

现在可以看到，**代理模式可以在不修改被代理对象的基础上，通过扩展代理类，进行一些功能的附加与增强。值得注意的是，代理类和被代理类应该共同实现一个接口，或者是共同继承某个类。**

上面介绍的是静态代理的内容，为什么叫做静态呢？因为它的类型是事先预定好的，比如上面代码中的 Cinema 这个类。下面要介绍的内容就是动态代理。

动态代理

既然是代理，那么它与静态代理的功能与目的是没有区别的，唯一有区别的就是动态与静态的差别。

那么在动态代理的中这个动态体现在什么地方？

上一节代码中 Cinema 类是代理，我们需要手动编写代码让 Cinema 实现 Movie 接口，而在动态代理中，我们可以让程序在运行的时候自动在内存中创建一个实现 Movie 接口的代理，而不需要去定义 Cinema 这个类。这就是它被称为动态的原因。

也许概念比较抽象。现在实例说明一下情况。

假设有一个大商场，商场有很多的柜台，有一个柜台卖茅台酒。我们进行代码的模拟。

```
package com.frank.test;

public interface SellWine {

     void mainJiu();

}
```

SellWine 是一个接口，你可以理解它为卖酒的许可证。

```
package com.frank.test;

public class MaotaiJiu implements SellWine {

    @Override
    public void mainJiu() {

        System.out.println("我卖得是茅台酒。");

    }

}
```

然后创建一个类 MaotaiJiu,对的，就是茅台酒的意思。

我们还需要一个柜台来卖酒:

```
package com.frank.test;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class GuitaiA implements InvocationHandler {

    private Object pingpai;

    public GuitaiA(Object pingpai) {
        this.pingpai = pingpai;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {

        System.out.println("销售开始  柜台是:
"+this.getClass().getSimpleName());
        method.invoke(pingpai, args);
        System.out.println("销售结束");
        return null;
    }

}
```

GuitaiA 实现了 InvocationHandler 这个类，这个类是什么意思呢？大家不要慌张，待会我会解释。

然后，我们就可以卖酒了。

```
package com.frank.test;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

public class Test {

    public static void main(String[] args) {

        MaotaiJiu maotaijiu = new MaotaiJiu();

        InvocationHandler jingxiao1 = new GuitaiA(maotaijiu);

        SellWine dynamicProxy = (SellWine)
Proxy.newProxyInstance(MaotaiJiu.class.getClassLoader(),
                MaotaiJiu.class.getInterfaces(), jingxiao1);

        dynamicProxy.mainJiu();

    }

}
```

这里，我们又接触到了一个新的概念，没有关系，先别管，先看结果。

```
&#x9500;&#x552E;&#x5F00;&#x59CB;  &#x67DC;&#x53F0;&#x662F;&#xFF1A; GuitaiA
&#x6211;&#x5356;&#x5F97;&#x662F;&#x8305;&#x53F0;&#x9152;&#x3002;
&#x9500;&#x552E;&#x7ED3;&#x675F;
```

看到没有，我并没有像静态代理那样为 SellWine 接口实现一个代理类，但最终它仍然实现了相同的功能，这其中的差别，就是之前讨论的动态代理所谓”动态”的原因。

动态代理语法

放轻松，下面我们开始讲解语法，语法非常简单。

动态代码涉及了一个非常重要的类 Proxy。正是通过 Proxy 的静态方法 newProxyInstance 才会动态创建代理。

Proxy

```
public static Object newProxyInstance(ClassLoader loader,
                                       Class<?>[] interfaces,
                                       InvocationHandler h)
```

下面讲解它的 3 个参数意义。

- loader 自然是类加载器
- interfaces 代码要用来代理的接口
- h 一个 InvocationHandler 对象

初学者应该对于 InvocationHandler 很陌生，我马上就讲到这一块。

InvocationHandler

InvocationHandler 是一个接口，官方文档解释说，每个代理的实例都有一个与之关联的 InvocationHandler 实现类，如果代理的方法被调用，那么代理便会通知和转发给内部的 InvocationHandler 实现类，由它决定处理。

```
public interface InvocationHandler {

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

InvocationHandler 内部只是一个 invoke() 方法，正是这个方法决定了怎么样处理代理传递过来的方法调用。

- proxy 代理对象
- method 代理对象调用的方法
- args 调用的方法中的参数

因为，Proxy 动态产生的代理会调用 InvocationHandler 实现类，所以 InvocationHandler 是实际执行者。

```
public class GuitaiA implements InvocationHandler {

    private Object pingpai;

    public GuitaiA(Object pingpai) {
        this.pingpai = pingpai;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {

        System.out.println("销售开始  柜台是:
"+this.getClass().getSimpleName());
        method.invoke(pingpai, args);
        System.out.println("销售结束");
        return null;
    }

}
```

GuitaiA 就是实际上卖酒的地方。

现在，我们加大难度，我们不仅要卖 **茅台酒**，还想卖 **五粮液**。

```
package com.frank.test;

public class Wuliangye implements SellWine {

    @Override
    public void mainJiu() {

        System.out.println("我卖得是五粮液。");

    }

}
```

Wuliangye 这个类也实现了 SellWine 这个接口，说明它也拥有卖酒的许可证，同样把它放到 GuitaiA 上售卖。

```
public class Test {

    public static void main(String[] args) {

        MaotaiJiu maotaijiu = new MaotaiJiu();

        Wuliangye wu = new Wuliangye();

        InvocationHandler jingxiao1 = new GuitaiA(maotaijiu);
        InvocationHandler jingxiao2 = new GuitaiA(wu);

        SellWine dynamicProxy = (SellWine)
Proxy.newProxyInstance(MaotaiJiu.class.getClassLoader(),
                MaotaiJiu.class.getInterfaces(), jingxiao1);
        SellWine dynamicProxy1 = (SellWine)
Proxy.newProxyInstance(MaotaiJiu.class.getClassLoader(),
                MaotaiJiu.class.getInterfaces(), jingxiao2);

        dynamicProxy.mainJiu();

        dynamicProxy1.mainJiu();

    }

}
```

我们来看结果：

```
&#x9500;&#x552E;&#x5F00;&#x59CB;  &#x67DC;&#x53F0;&#x662F;&#xFF1A; GuitaiA
&#x6211;&#x5356;&#x5F97;&#x662F;&#x8305;&#x53F0;&#x9152;&#x3002;
&#x9500;&#x552E;&#x7ED3;&#x675F;
&#x9500;&#x552E;&#x5F00;&#x59CB;  &#x67DC;&#x53F0;&#x662F;&#xFF1A; GuitaiA
&#x6211;&#x5356;&#x5F97;&#x662F;&#x4E94;&#x7CAE;&#x6DB2;&#x3002;
&#x9500;&#x552E;&#x7ED3;&#x675F;
```

有人会问，dynamicProxy 和 dynamicProxy1 什么区别没有？他们都是动态产生的代理，都是售货员，都拥有卖酒的技术证书。

我现在扩大商场的经营，除了卖酒之外，还要卖烟。

首先，同样要创建一个接口，作为卖烟的许可证。

```
package com.frank.test;

public interface SellCigarette {
    void sell();
}
```

然后，卖什么烟呢？我是湖南人，那就芙蓉王好了。

```
public class Furongwang implements SellCigarette {

    @Override
    public void sell() {

        System.out.println("售卖的是正宗的芙蓉王，可以扫描条形码查证。");
    }

}
```

然后再次测试验证：

```
package com.frank.test;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

public class Test {

    public static void main(String[] args) {

        MaotaiJiu maotaijiu = new MaotaiJiu();

        Wuliangye wu = new Wuliangye();

        Furongwang fu = new Furongwang();

        InvocationHandler jingxiao1 = new GuitaiA(maotaijiu);
        InvocationHandler jingxiao2 = new GuitaiA(wu);

        InvocationHandler jingxiao3 = new GuitaiA(fu);

        SellWine dynamicProxy = (SellWine)
Proxy.newProxyInstance(MaotaiJiu.class.getClassLoader(),
                MaotaiJiu.class.getInterfaces(), jingxiao1);
        SellWine dynamicProxy1 = (SellWine)
Proxy.newProxyInstance(MaotaiJiu.class.getClassLoader(),
                MaotaiJiu.class.getInterfaces(), jingxiao2);

        dynamicProxy.mainJiu();

        dynamicProxy1.mainJiu();

        SellCigarette dynamicProxy3 = (SellCigarette)
Proxy.newProxyInstance(Furongwang.class.getClassLoader(),
                Furongwang.class.getInterfaces(), jingxiao3);

        dynamicProxy3.sell();

    }

}
```

然后，查看结果：

```
&#x9500;&#x552E;&#x5F00;&#x59CB;  &#x67DC;&#x53F0;&#x662F;&#xFF1A; GuitaiA
&#x6211;&#x5356;&#x5F97;&#x662F;&#x8305;&#x53F0;&#x9152;&#x3002;
&#x9500;&#x552E;&#x7ED3;&#x675F;
&#x9500;&#x552E;&#x5F00;&#x59CB;  &#x67DC;&#x53F0;&#x662F;&#xFF1A; GuitaiA
&#x6211;&#x5356;&#x5F97;&#x662F;&#x4E94;&#x7CAE;&#x6DB2;&#x3002;
&#x9500;&#x552E;&#x7ED3;&#x675F;
&#x9500;&#x552E;&#x5F00;&#x59CB;  &#x67DC;&#x53F0;&#x662F;&#xFF1A; GuitaiA
&#x552E;&#x5356;&#x7684;&#x662F;&#x6B63;&#x5B97;&#x7684;&#x8299;&#x84C9;&#x738B
;&#xFF0C;&#x53EF;&#x4EE5;&#x626B;&#x63CF;&#x6761;&#x5F62;&#x7801;&#x67E5;&#x8BC
1;&#x3002;
&#x9500;&#x552E;&#x7ED3;&#x675F;
```

结果符合预期。大家仔细观察一下代码，同样是通过 Proxy.newProxyInstance() 方法，却产生了 SellWine 和 SellCigarette 两种接口的实现类代理，这就是动态代理的魔力。

动态代理的秘密

一定有同学对于为什么 Proxy 能够动态产生不同接口类型的代理感兴趣，我的猜测是肯定通过传入进去的接口然后通过反射动态生成了一个接口实例。 比如 SellWine 是一个接口，那么 Proxy.newProxyInstance() 内部肯定会有

```
new SellWine();
```

这样相同作用的代码，不过它是通过反射机制创建的。那么事实是不是这样子呢？直接查看它们的源码好了。需要说明的是，我当前查看的源码是 1.8 版本。

```
public static Object newProxyInstance(ClassLoader loader,
                                       Class<?>[] interfaces,
                                       InvocationHandler h)
        throws IllegalArgumentException
    {
        Objects.requireNonNull(h);

        final Class<?>[] intfs = interfaces.clone();

        Class<?> cl = getProxyClass0(loader, intfs);

        try {

            final Constructor<?> cons = cl.getConstructor(constructorParams);
            final InvocationHandler ih = h;
            if (!Modifier.isPublic(cl.getModifiers())) {
                AccessController.doPrivileged(new PrivilegedAction<Void>() {
                    public Void run() {
                        cons.setAccessible(true);
                        return null;
                    }
                });
            }

            return cons.newInstance(new Object[]{h});

        } catch (IllegalAccessException|InstantiationException e) {
            throw new InternalError(e.toString(), e);
        } catch (InvocationTargetException e) {
            Throwable t = e.getCause();
            if (t instanceof RuntimeException) {
                throw (RuntimeException) t;
            } else {
                throw new InternalError(t.toString(), t);
            }
        } catch (NoSuchMethodException e) {
            throw new InternalError(e.toString(), e);
        }
    }
```

newProxyInstance 的确创建了一个实例，它是通过 cl 这个 Class 文件的构造方法反射生成。cl 由 getProxyClass0() 方法获取。

```
private static Class<?> getProxyClass0(ClassLoader loader,
                                       Class<?>... interfaces) {
    if (interfaces.length > 65535) {
        throw new IllegalArgumentException("interface limit exceeded");
    }

    return proxyClassCache.get(loader, interfaces);
}
```

直接通过缓存获取，如果获取不到，注释说会通过 ProxyClassFactory 生成。

```
    private static final class ProxyClassFactory
        implements BiFunction<ClassLoader, Class<?>[], Class<?>>
    {

        private static final String proxyClassNamePrefix = "$Proxy";

        private static final AtomicLong nextUniqueNumber = new AtomicLong();

        @Override
        public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {

            Map<Class<?>, Boolean> interfaceSet = new IdentityHashMap<>
(interfaces.length);
            for (Class<?> intf : interfaces) {

                Class<?> interfaceClass = null;
                try {
                    interfaceClass = Class.forName(intf.getName(), false,
loader);
                } catch (ClassNotFoundException e) {
                }
                if (interfaceClass != intf) {
                    throw new IllegalArgumentException(
                        intf + " is not visible from class loader");
                }

                if (!interfaceClass.isInterface()) {
                    throw new IllegalArgumentException(
                        interfaceClass.getName() + " is not an interface");
                }

                if (interfaceSet.put(interfaceClass, Boolean.TRUE) != null) {
                    throw new IllegalArgumentException(
                        "repeated interface: " + interfaceClass.getName());
                }
            }

            String proxyPkg = null;
            int accessFlags = Modifier.PUBLIC | Modifier.FINAL;

            for (Class<?> intf : interfaces) {
                int flags = intf.getModifiers();
                if (!Modifier.isPublic(flags)) {
                    accessFlags = Modifier.FINAL;
                    String name = intf.getName();
                    int n = name.lastIndexOf('.');
                    String pkg = ((n == -1) ? "" : name.substring(0, n + 1));
                    if (proxyPkg == null) {
                        proxyPkg = pkg;
                    } else if (!pkg.equals(proxyPkg)) {
                        throw new IllegalArgumentException(
                            "non-public interfaces from different packages");
                    }
                }
```

```
            }

            if (proxyPkg == null) {

                proxyPkg = ReflectUtil.PROXY_PACKAGE + ".";
            }

            long num = nextUniqueNumber.getAndIncrement();
            String proxyName = proxyPkg + proxyClassNamePrefix + num;

            byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
                proxyName, interfaces, accessFlags);
            try {
                return defineClass0(loader, proxyName,
                                    proxyClassFile, 0, proxyClassFile.length);
            } catch (ClassFormatError e) {

                throw new IllegalArgumentException(e.toString());
            }
        }
    }
```

这个类的注释说，通过指定的 ClassLoader 和 接口数组 用工厂方法生成 proxy class。然后这个 proxy class 的名字是：

```
private static final String proxyClassNamePrefix = "$Proxy";

long num = nextUniqueNumber.getAndIncrement();

String proxyName = proxyPkg + proxyClassNamePrefix + num;
```

所以，动态生成的代理类名称是 **包名+\$Proxy+id 序号。**

生成的过程，核心代码如下：

```
byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
                proxyName, interfaces, accessFlags);

return defineClass0(loader, proxyName,
                    proxyClassFile, 0, proxyClassFile.length);
```

这两个方法，我没有继续追踪下去，defineClass0() 甚至是一个 native 方法。我们只要知道，动态创建代理这回事就好了。

现在我们还需要做一些验证，我要检测一下动态生成的代理类的名字是不是 **包名+\$Proxy+id 序号。**

```
public class Test {

    public static void main(String[] args) {

        MaotaiJiu maotaijiu = new MaotaiJiu();

        Wuliangye wu = new Wuliangye();

        Furongwang fu = new Furongwang();

        InvocationHandler jingxiao1 = new GuitaiA(maotaijiu);
        InvocationHandler jingxiao2 = new GuitaiA(wu);

        InvocationHandler jingxiao3 = new GuitaiA(fu);

        SellWine dynamicProxy = (SellWine)
Proxy.newProxyInstance(MaotaiJiu.class.getClassLoader(),
                MaotaiJiu.class.getInterfaces(), jingxiao1);
        SellWine dynamicProxy1 = (SellWine)
Proxy.newProxyInstance(MaotaiJiu.class.getClassLoader(),
                MaotaiJiu.class.getInterfaces(), jingxiao2);

        dynamicProxy.mainJiu();

        dynamicProxy1.mainJiu();

        SellCigarette dynamicProxy3 = (SellCigarette)
Proxy.newProxyInstance(Furongwang.class.getClassLoader(),
                Furongwang.class.getInterfaces(), jingxiao3);

        dynamicProxy3.sell();

        System.out.println("dynamicProxy class
name:"+dynamicProxy.getClass().getName());
        System.out.println("dynamicProxy1 class
name:"+dynamicProxy1.getClass().getName());
        System.out.println("dynamicProxy3 class
name:"+dynamicProxy3.getClass().getName());

    }

}
```

结果如下:

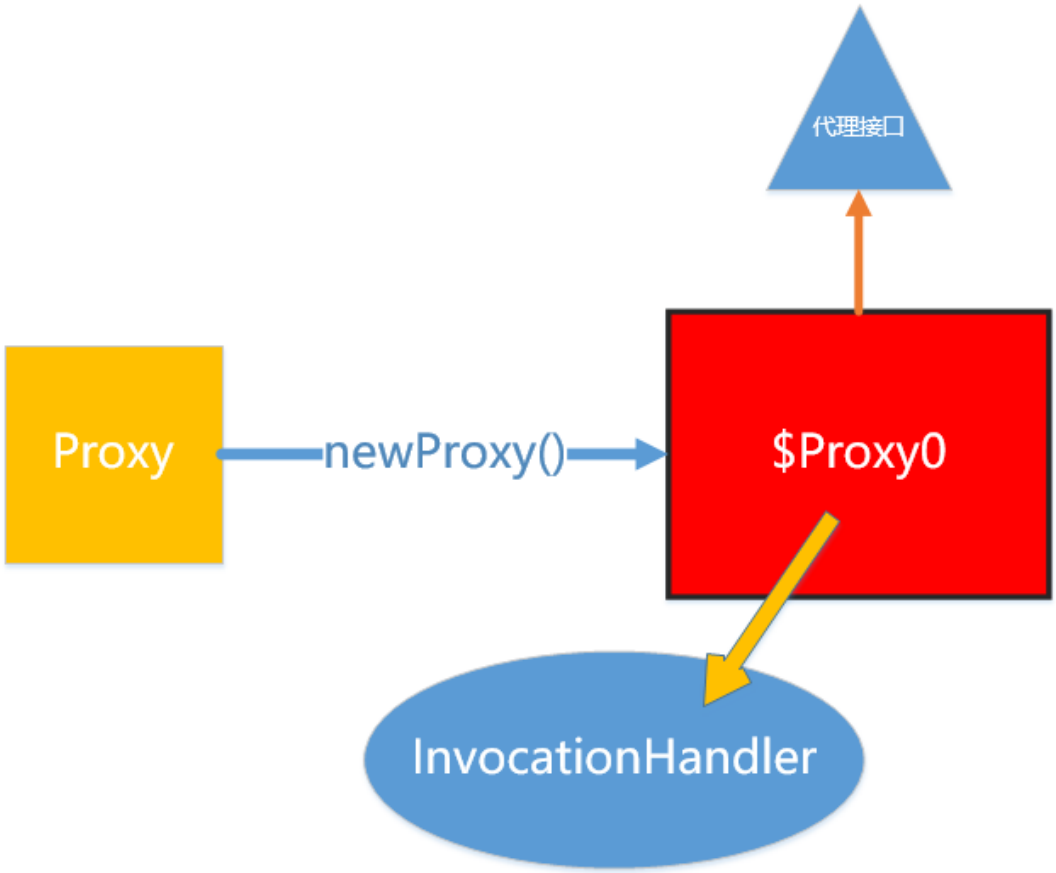
```
&#x9500;&#x552E;&#x5F00;&#x59CB;  &#x67DC;&#x53F0;&#x662F;&#xFF1A; GuitaiA
&#x6211;&#x5356;&#x5F97;&#x662F;&#x8305;&#x53F0;&#x9152;&#x3002;
&#x9500;&#x552E;&#x7ED3;&#x675F;
&#x9500;&#x552E;&#x5F00;&#x59CB;  &#x67DC;&#x53F0;&#x662F;&#xFF1A; GuitaiA
&#x6211;&#x5356;&#x5F97;&#x662F;&#x4E94;&#x7CAE;&#x6DB2;&#x3002;
&#x9500;&#x552E;&#x7ED3;&#x675F;
&#x9500;&#x552E;&#x5F00;&#x59CB;  &#x67DC;&#x53F0;&#x662F;&#xFF1A; GuitaiA
&#x552E;&#x5356;&#x7684;&#x662F;&#x6B63;&#x5B97;&#x7684;&#x8299;&#x84C9;&#x738B
;&#xFF0C;&#x53EF;&#x4EE5;&#x626B;&#x63CF;&#x6761;&#x5F62;&#x7801;&#x67E5;&#x8BC
1;&#x3002;
&#x9500;&#x552E;&#x7ED3;&#x675F;

dynamicProxy class name:com.sun.proxy.$Proxy0
dynamicProxy1 class name:com.sun.proxy.$Proxy0
dynamicProxy3 class name:com.sun.proxy.$Proxy1
```

SellWine 接口的代理类名是： `com.sun.proxy.$Proxy0` SellCigarette 接口的代理类名是： `com.sun.proxy.$Proxy1`

这说明动态生成的 proxy class 与 Proxy 这个类同一个包。

下面用一张图让大家记住动态代理涉及到的角色。



红框中 `$Proxy0` 就是通过 Proxy 动态生成的。

`$Proxy0` 实现了要代理的接口。

`$Proxy0` 通过调用 `InvocationHandler` 来执行任务。

代理的作用

可能有同学会问，已经学习了代理的知识，但是，它们有什么用呢?

主要作用，还是在不修改被代理对象的源码上，进行功能的增强。

这在 AOP 面向切面编程领域经常见。

> 在软件业，AOP 为 Aspect Oriented Programming 的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP 是 OOP 的延续，是软件开发中的一个热点，也是 Spring 框架中的一个重要内容，是函数式编程的一种衍生范型。利用 AOP 可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。
>
> 主要功能 日志记录，性能统计，安全控制，事务处理，异常处理等等。

上面的引用是百度百科对于 AOP 的解释，至于，如何通过代理来进行日志记录功能、性能统计等等，这个大家可以参考 AOP 的相关源码，然后仔细琢磨。

同注解一样，很多同学可能会有疑惑，我什么时候用代理呢?

这取决于你自己想干什么。你已经学会了语法了，其他的看业务需求。对于实现日志记录功能的框架来说，正合适。

至此，静态代理和动态代理者讲完了。

总结

1. 代理分为静态代理和动态代理两种。
2. 静态代理，代理类需要自己编写代码写成。
3. 动态代理，代理类通过 Proxy.newInstance() 方法生成。
4. 不管是静态代理还是动态代理，代理与被代理者都要实现两样接口，它们的实质是面向接口编程。
5. 静态代理和动态代理的区别是在于要不要开发者自己定义 Proxy 类。
6. 动态代理通过 Proxy 动态生成 proxy class，但是它也指定了一个 InvocationHandler 的实现类。
7. 代理模式本质上的目的是为了增强现有代码的功能。