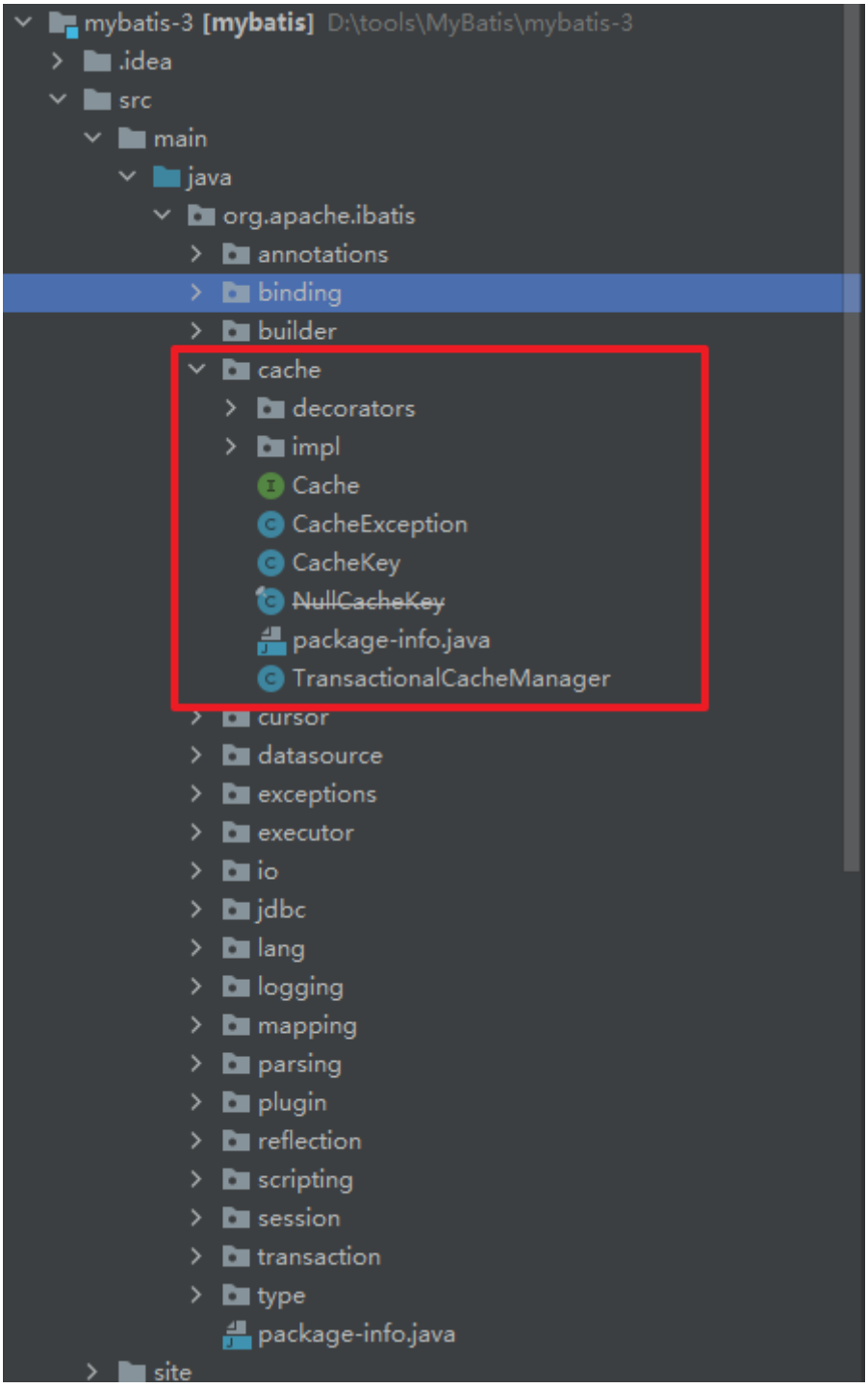
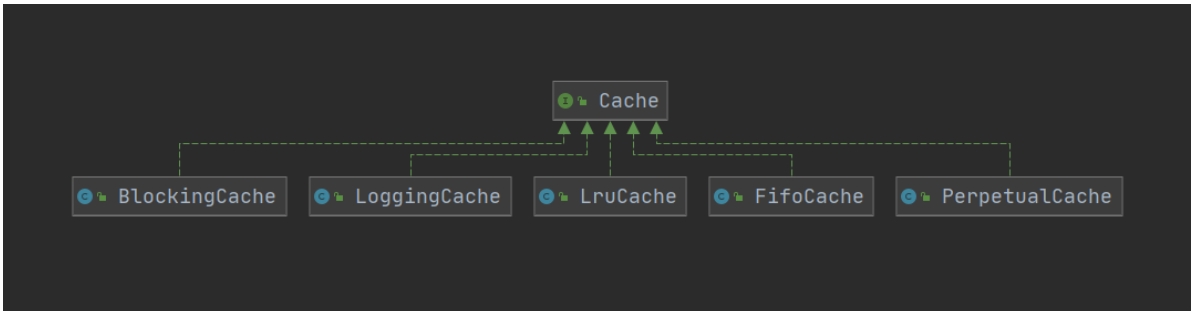
缓存模块

MyBatis作为一个强大的持久层框架，缓存是其必不可少的功能之一，Mybatis中的缓存分为一级缓存和二级缓存。但本质上是一样的，都是使用Cache接口实现的。缓存位于 org.apache.ibatis.cache包下。



通过结构我们能够发现Cache其实使用到了装饰器模式来实现缓存的处理。首先大家需要先回顾下装饰器模式的相关内容哦。我们先来看看Cache中的基础类的API

> // 煎饼加鸡蛋加香肠
> “装饰者模式（Decorator Pattern）是指在不改变原有对象的基础之上，将功能附加到对象上，提供了比继承更有弹性的替代方案（扩展原有对象的功能）。”



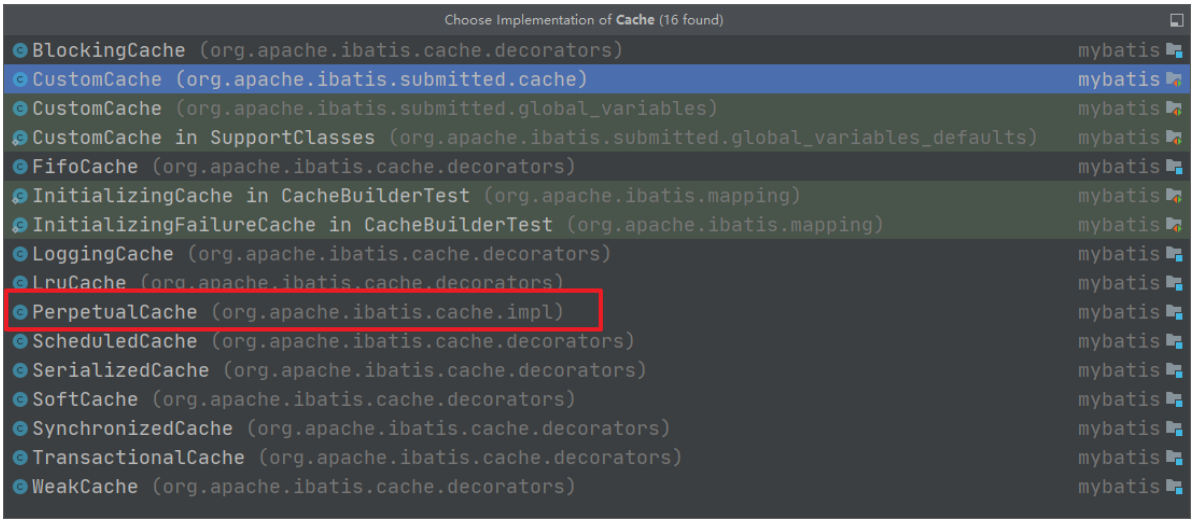
1. Cache接口

Cache接口是缓存模块中最核心的接口，它定义了所有缓存的基本行为，Cache接口的定义如下：

```
public interface Cache {

  /**
   * 缓存对象的 ID
   * @return The identifier of this cache
   */
  String getId();

  /**
   * 向缓存中添加数据，一般情况下 key是CacheKey  value是查询结果
   * @param key Can be any object but usually it is a {@link CacheKey}
   * @param value The result of a select.
   */
  void putObject(Object key, Object value);

  /**
   * 根据指定的key，在缓存中查找对应的结果对象
   * @param key The key
   * @return The object stored in the cache.
   */
  Object getObject(Object key);

  /**
   * As of 3.3.0 this method is only called during a rollback
   * for any previous value that was missing in the cache.
   * This lets any blocking cache to release the lock that
   * may have previously put on the key.
   * A blocking cache puts a lock when a value is null
   * and releases it when the value is back again.
   * This way other threads will wait for the value to be
   * available instead of hitting the database.
   *   删除key对应的缓存数据
   *
   * @param key The key
   * @return Not used
   */
  Object removeObject(Object key);

  /**
   * Clears this cache instance.
   * 清空缓存
   */
  void clear();
```

```
  /**
   * Optional. This method is not called by the core.
   * 缓存的个数。
   * @return The number of elements stored in the cache (not its capacity).
   */
  int getSize();

  /**
   * Optional. As of 3.2.6 this method is no longer called by the core.
   * <p>
   * Any locking needed by the cache must be provided internally by the cache
provider.
   *  获取读写锁
   * @return A ReadWriteLock
   */
  default ReadWriteLock getReadWriteLock() {
    return null;
  }

}
```

Cache接口的实现类很多，但是大部分都是装饰器，只有PerpetualCache提供了Cache接口的基本实现。



2. PerpetualCache

PerpetualCache在缓存模块中扮演了ConcreteComponent的角色，其实现比较简单，底层使用HashMap记录缓存项，具体的实现如下：

```
/**
 * 在装饰器模式用 用来被装饰的对象
 * 缓存中的  基本缓存处理的实现
 * 其实就是一个 HashMap 的基本操作
 * @author Clinton Begin
 */
public class PerpetualCache implements Cache {

  private final String id; // Cache 对象的唯一标识

  // 用于记录缓存的Map对象
  private final Map<Object, Object> cache = new HashMap<>();
```

```java
public PerpetualCache(String id) {
  this.id = id;
}

@Override
public String getId() {
  return id;
}

@Override
public int getSize() {
  return cache.size();
}

@Override
public void putObject(Object key, Object value) {
  cache.put(key, value);
}

@Override
public Object getObject(Object key) {
  return cache.get(key);
}

@Override
public Object removeObject(Object key) {
  return cache.remove(key);
}

@Override
public void clear() {
  cache.clear();
}

@Override
public boolean equals(Object o) {
  if (getId() == null) {
    throw new CacheException("Cache instances require an ID.");
  }
  if (this == o) {
    return true;
  }
  if (!(o instanceof Cache)) {
    return false;
  }

  Cache otherCache = (Cache) o;
  // 只关心ID
  return getId().equals(otherCache.getId());
}

@Override
public int hashCode() {
  if (getId() == null) {
    throw new CacheException("Cache instances require an ID.");
  }
  // 只关心ID
```

```
    return getId().hashCode();
  }

}
```

然后我们可以来看看cache.decorators包下提供的装饰器。他们都实现了Cache接口。这些装饰器都在PerpetualCache的基础上提供了一些额外的功能，通过多个组合实现一些特殊的需求。

3.BlockingCache

通过名称我们能看出来是一个阻塞同步的缓存，它保证只有一个线程到缓存中查找指定的key对应的数据。

```
public class BlockingCache implements Cache {

  private long timeout; // 阻塞超时时长
  private final Cache delegate; // 被装饰的底层 Cache 对象
  // 每个key 都有对象的 ReentrantLock 对象
  private final ConcurrentHashMap<Object, ReentrantLock> locks;

  public BlockingCache(Cache delegate) {
    // 被装饰的 Cache 对象
    this.delegate = delegate;
    this.locks = new ConcurrentHashMap<>();
  }

  @Override
  public String getId() {
    return delegate.getId();
  }

  @Override
  public int getSize() {
    return delegate.getSize();
  }

  @Override
  public void putObject(Object key, Object value) {
    try {
      // 执行 被装饰的 Cache 中的方法
      delegate.putObject(key, value);
    } finally {
      // 释放锁
      releaseLock(key);
    }
  }

  @Override
  public Object getObject(Object key) {
    acquireLock(key); // 获取锁
    Object value = delegate.getObject(key); // 获取缓存数据
    if (value != null) { // 有数据就释放掉锁，否则继续持有锁
      releaseLock(key);
    }
    return value;
  }
```

```
  @Override
  public Object removeObject(Object key) {
    // despite of its name, this method is called only to release locks
    releaseLock(key);
    return null;
  }

  @Override
  public void clear() {
    delegate.clear();
  }

  private ReentrantLock getLockForKey(Object key) {
    return locks.computeIfAbsent(key, k -> new ReentrantLock());
  }

  private void acquireLock(Object key) {
    Lock lock = getLockForKey(key);
    if (timeout > 0) {
      try {
        boolean acquired = lock.tryLock(timeout, TimeUnit.MILLISECONDS);
        if (!acquired) {
          throw new CacheException("Couldn't get a lock in " + timeout + " for
the key " +  key + " at the cache " + delegate.getId());
        }
      } catch (InterruptedException e) {
        throw new CacheException("Got interrupted while trying to acquire lock
for key " + key, e);
      }
    } else {
      lock.lock();
    }
  }

  private void releaseLock(Object key) {
    ReentrantLock lock = locks.get(key);
    if (lock.isHeldByCurrentThread()) {
      lock.unlock();
    }
  }

  public long getTimeout() {
    return timeout;
  }

  public void setTimeout(long timeout) {
    this.timeout = timeout;
  }
}
```

通过源码我们能够发现，BlockingCache本质上就是在我们操作缓存数据的前后通过 ReentrantLock对象来实现了加锁和解锁操作。其他的具体实现类，大家可以自行查阅

缓存实现类	描述	作用	装饰条件
基本缓存	缓存基本实现类	默认是PerpetualCache，也可以自定义比如RedisCache、EhCache等，具备基本功能的缓存类	无
LruCache	LRU策略的缓存	当缓存到达上限时候，删除最近最少使用的缓存（Least Recently Use）	eviction="LRU"（默认）
FifoCache	FIFO策略的缓存	当缓存到达上限时候，删除最先入队的缓存	eviction="FIFO"
SoftCacheWeakCache	带清理策略的缓存	通过JVM的软引用和弱引用来实现缓存，当JVM内存不足时，会自动清理掉这些缓存，基于SoftReference和WeakReference	eviction="SOFT"eviction="WEAK"
LoggingCache	带日志功能的缓存	比如：输出缓存命中率	基本
SynchronizedCache	同步缓存	基于synchronized关键字实现，解决并发问题	基本
BlockingCache	阻塞缓存	通过在get/put方式中加锁，保证只有一个线程操作缓存，基于Java重入锁实现	blocking=true
SerializedCache	支持序列化的缓存	将对象序列化以后存到缓存中，取出时反序列化	readOnly=false（默认）
ScheduledCache	定时调度的缓存	在进行get/put/remove/getSize等操作前，判断缓存时间是否超过了设置的最长缓存时间（默认是一小时），如果是则清空缓存--即每隔一段时间清空一次缓存	flushInterval不为空
TransactionalCache	事务缓存	在二级缓存中使用，可一次存入多个缓存，移除多个缓存	在TransactionalCacheManager中用Map维护对应关系

4. 缓存的应用

4.1 缓存对应的初始化

在Configuration初始化的时候会为我们的各种Cache实现注册对应的别名

```
public Configuration() {
  // 为类型注册别名
  typeAliasRegistry.registerAlias( alias: "JDBC", JdbcTransactionFactory.class);
  typeAliasRegistry.registerAlias( alias: "MANAGED", ManagedTransactionFactory.class);

  typeAliasRegistry.registerAlias( alias: "JNDI", JndiDataSourceFactory.class);
  typeAliasRegistry.registerAlias( alias: "POOLED", PooledDataSourceFactory.class);
  typeAliasRegistry.registerAlias( alias: "UNPOOLED", UnpooledDataSourceFactory.class);

  typeAliasRegistry.registerAlias( alias: "PERPETUAL", PerpetualCache.class);
  typeAliasRegistry.registerAlias( alias: "FIFO", FifoCache.class);
  typeAliasRegistry.registerAlias( alias: "LRU", LruCache.class);
  typeAliasRegistry.registerAlias( alias: "SOFT", SoftCache.class);
  typeAliasRegistry.registerAlias( alias: "WEAK", WeakCache.class);

  typeAliasRegistry.registerAlias( alias: "DB_VENDOR", VendorDatabaseIdProvider.class);

  typeAliasRegistry.registerAlias( alias: "XML", XMLLanguageDriver.class);
  typeAliasRegistry.registerAlias( alias: "RAW", RawLanguageDriver.class);

  typeAliasRegistry.registerAlias( alias: "SLF4J", Slf4jImpl.class);
```

在解析settings标签的时候，设置的默认值有如下

```
// settings 中的属性信息都是直接保存在 Configuration 对象的属性中的
private void settingsElement(Properties props) {
  configuration.setAutoMappingBehavior(AutoMappingBehavior.valueOf(props.getProperty( key: "autoMappingBehavior",  defa
  configuration.setAutoMappingUnknownColumnBehavior(AutoMappingUnknownColumnBehavior.valueOf(props.getProperty( key: "a
  configuration.setCacheEnabled(booleanValueOf(props.getProperty("cacheEnabled"),  defaultValue: true));
  configuration.setProxyFactory((ProxyFactory) createInstance(props.getProperty("proxyFactory")));
  configuration.setLazyLoadingEnabled(booleanValueOf(props.getProperty("lazyLoadingEnabled"),  defaultValue: false));
  configuration.setAggressiveLazyLoading(booleanValueOf(props.getProperty("aggressiveLazyLoading"),  defaultValue: false
  configuration.setMultipleResultSetsEnabled(booleanValueOf(props.getProperty("multipleResultSetsEnabled"),  defaultVal
  configuration.setUseColumnLabel(booleanValueOf(props.getProperty("useColumnLabel"),  defaultValue: true));
  configuration.setUseGeneratedKeys(booleanValueOf(props.getProperty("useGeneratedKeys"),  defaultValue: false));
  configuration.setDefaultExecutorType(ExecutorType.valueOf(props.getProperty( key: "defaultExecutorType",  defaultValue:
  configuration.setDefaultStatementTimeout(integerValueOf(props.getProperty("defaultStatementTimeout"),  defaultValue: n
  configuration.setDefaultFetchSize(integerValueOf(props.getProperty("defaultFetchSize"),  defaultValue: null));
  configuration.setDefaultResultSetType(resolveResultSetType(props.getProperty("defaultResultSetType")));
```

cacheEnabled默认为true，localCacheScope默认为 SESSION

在解析映射文件的时候会解析我们相关的cache标签

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

private void configurationElement(XNode context) {
  try {
    String namespace = context.getStringAttribute( name: "namespace");
    if (namespace == null || namespace.equals("")) {
      throw new BuilderException("Mapper's namespace cannot be empty");
    }
    builderAssistant.setCurrentNamespace(namespace);
    // 添加缓存对象 如果我们希望多个 namespace 共用同一个二级缓存 就可以使用
    cacheRefElement(context.evalNode("cache-ref"));
    // 解析 cache 属性，添加缓存对象
    cacheElement(context.evalNode("cache"));
    // 创建 ParameterMapping 对象 以废弃 不推荐
    parameterMapElement(context.evalNodes( expression: "/mapper/parameterMap"));
    // 创建 List<ResultMapping>
    resultMapElements(context.evalNodes( expression: "/mapper/resultMap"));
    // 解析可以复用的SQL  includ
    sqlElement(context.evalNodes( expression: "/mapper/sql"));
    // 解析增删改查标签，得到 MappedStatement >> 一个 CRUD标签中的信息都被封装到了 MappedStatement 对象
    buildStatementFromContext(context.evalNodes( expression: "select|insert|update|delete"));
  } catch (Exception e) {
    throw new BuilderException("Error parsing Mapper XML. The XML location is '" + resource +
```

然后解析映射文件的cache标签后会在Configuration对象中添加对应的数据在

```
  private void cacheElement(XNode context) {
    // 只有 cache 标签不为空才解析
    if (context != null) {
      String type = context.getStringAttribute("type", "PERPETUAL");
      Class<? extends Cache> typeClass = typeAliasRegistry.resolveAlias(type);
      String eviction = context.getStringAttribute("eviction", "LRU");
      Class<? extends Cache> evictionClass =
typeAliasRegistry.resolveAlias(eviction);
      Long flushInterval = context.getLongAttribute("flushInterval");
      Integer size = context.getIntAttribute("size");
      boolean readWrite = !context.getBooleanAttribute("readOnly", false);
      boolean blocking = context.getBooleanAttribute("blocking", false);
      Properties props = context.getChildrenAsProperties();
      builderAssistant.useNewCache(typeClass, evictionClass, flushInterval,
size, readWrite, blocking, props);
    }
  }
```

继续

```
  public Cache useNewCache(Class<? extends Cache> typeClass,
      Class<? extends Cache> evictionClass,
      Long flushInterval,
      Integer size,
      boolean readWrite,
      boolean blocking,
      Properties props) {
    Cache cache = new CacheBuilder(currentNamespace)
        .implementation(valueOrDefault(typeClass, PerpetualCache.class))
        .addDecorator(valueOrDefault(evictionClass, LruCache.class))
        .clearInterval(flushInterval)
        .size(size)
        .readWrite(readWrite)
        .blocking(blocking)
        .properties(props)
        .build();
    // 有将 二级缓存的 Cache 对象添加到 Configuration 对象中
    configuration.addCache(cache);
    currentCache = cache; // 记录当前名称空间记录的 cache
    return cache;
  }
```

然后我们可以发现 如果存储 cache 标签，那么对应的 Cache对象会被保存在 currentCache 属性中。

进而在 Cache 对象 保存在了 MapperStatement 对象的 cache 属性中。

然后我们再看看openSession的时候又做了哪些操作，在创建对应的执行器的时候会有缓存的操作

```
  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 {
      // 默认 SimpleExecutor
      executor = new SimpleExecutor(this, transaction);
    }
    // 二级缓存开关，settings 中的 cacheEnabled 默认是 true
    if (cacheEnabled) {
      executor = new CachingExecutor(executor);
    }
    // 植入插件的逻辑，至此，四大对象已经全部拦截完毕
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
  }
```

也就是如果 cacheEnabled 为 true 就会通过 CachingExecutor 来装饰executor 对象，然后就是在执行SQL操作的时候会涉及到缓存的具体使用。这个就分为一级缓存和二级缓存，这个我们来分别介绍

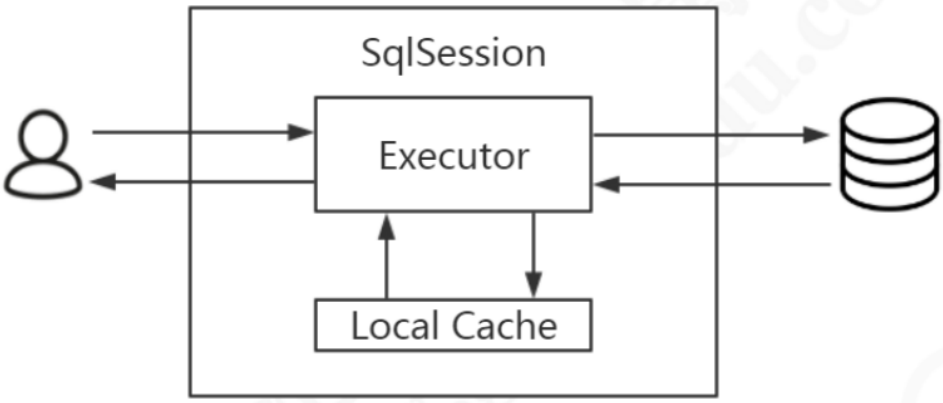
4.2 一级缓存

一级缓存也叫本地缓存（Local Cache），MyBatis的一级缓存是在会话（SqlSession）层面进行缓存的。MyBatis的一级缓存是默认开启的，不需要任何的配置（如果要关闭，localCacheScope设置为STATEMENT）。在BaseExecutor对象的query方法中有关闭一级缓存的逻辑

```
      }
      // issue #601
      deferredLoads.clear();
      // 如果LocalCacheScope的值设置为 STATEMENT 则一级缓存失效
      if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
        // issue #482
        clearLocalCache();
      }
    }
    return list;
  }
```

然后我们需要考虑下在一级缓存中的 PerpetualCache 对象在哪创建的，因为一级缓存是Session级别的缓存，肯定需要在Session范围内创建，其实PerpetualCache的实例化是在BaseExecutor的构造方法中创建的

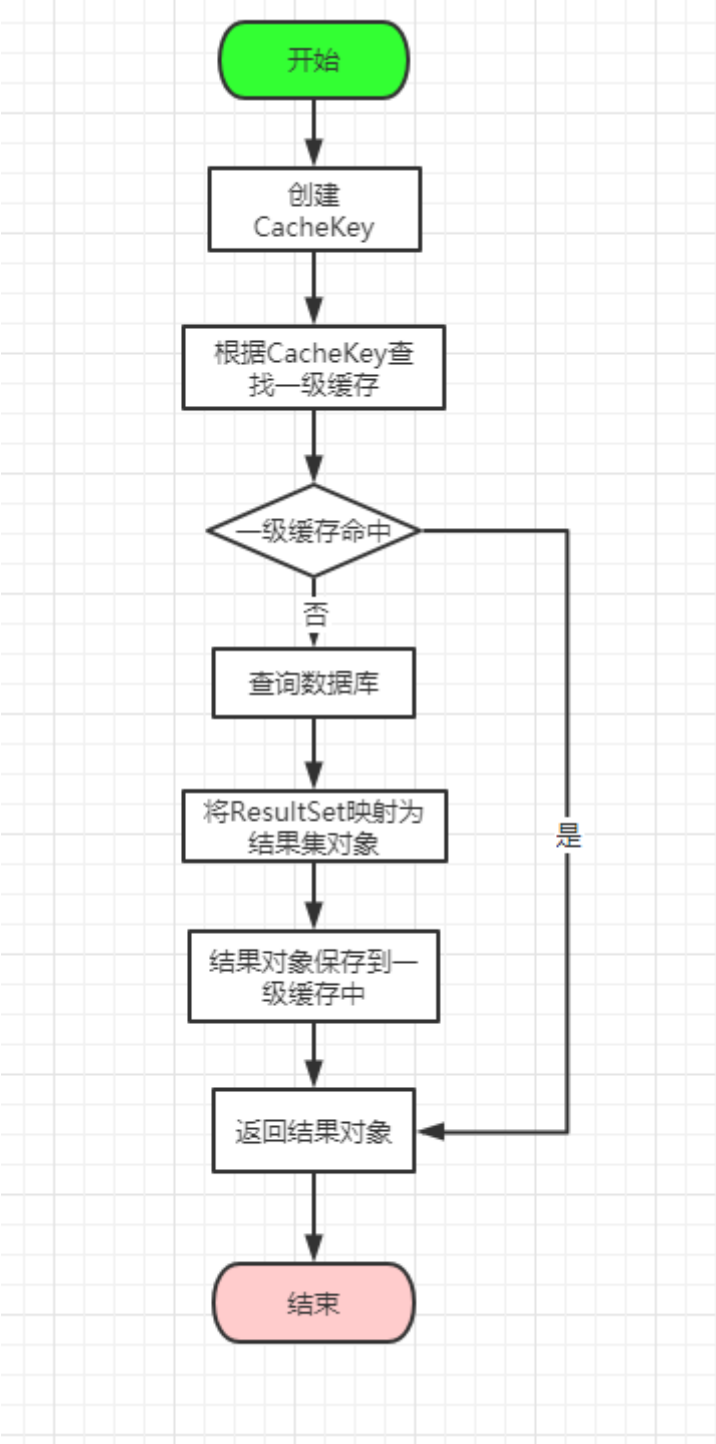
```
  protected BaseExecutor(Configuration configuration, Transaction transaction) {
    this.transaction = transaction;
    this.deferredLoads = new ConcurrentLinkedQueue<>();
    this.localCache = new PerpetualCache("LocalCache");
    this.localOutputParameterCache = new
PerpetualCache("LocalOutputParameterCache");
    this.closed = false;
    this.configuration = configuration;
    this.wrapper = this;
  }
```



一级缓存的具体实现也是在BaseExecutor的query方法中来实现的

```
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws
SQLException {
    // 异常体系之 ErrorContext
    ErrorContext.instance().resource(ms.getResource()).activity("executing a
query").object(ms.getId());
    if (closed) {
      throw new ExecutorException("Executor was closed.");
    }
    if (queryStack == 0 && ms.isFlushCacheRequired()) {
      // flushCache="true"时，即使是查询，也清空一级缓存
      clearLocalCache();
    }
    List<E> list;
    try {
      // 防止递归查询重复处理缓存
      queryStack++;
      // 查询一级缓存
      // ResultHandler 和 ResultSetHandler的区别
      list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
      if (list != null) {
        handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
      } else {
        // 真正的查询流程
        list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
boundSql);
      }
    } finally {
      queryStack--;
    }
    if (queryStack == 0) {
      for (DeferredLoad deferredLoad : deferredLoads) {
        deferredLoad.load();
      }
      // issue #601
      deferredLoads.clear();
      if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
        // issue #482
        clearLocalCache();
      }
    }
    return list;
```

```
}
```



一级缓存的验证：

同一个Session中的多个相同操作

```
    @Test
    public void test1() throws  Exception{
        // 1.获取配置文件
        InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
        // 2.加载解析配置文件并获取SqlSessionFactory对象
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
        // 3.根据SqlSessionFactory对象获取SqlSession对象
        SqlSession sqlSession = factory.openSession();
        // 4.通过SqlSession中提供的 API方法来操作数据库
        List<User> list =
sqlSession.selectList("com.gupaoedu.mapper.UserMapper.selectUserList");
```

```
        System.out.println(list.size());
        // 一级缓存测试
        System.out.println("---------");
        list =
sqlSession.selectList("com.gupaoedu.mapper.UserMapper.selectUserList");
        System.out.println(list.size());
        // 5.关闭会话
        sqlSession.close();
    }
```

输出日志

```
Setting autocommit to false on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
==>  Preparing: select * from t_user
==> Parameters:
<==    Columns: id, user_name, real_name, password, age, d_id
<==        Row: 1, zhangsan, 张三, 123456, 18, null
<==        Row: 2, lisi, 李四, 11111, 19, null
<==        Row: 3, wangwu, 王五, 111, 22, 1001
<==        Row: 4, wangwu, 王五, 111, 22, 1001
<==        Row: 5, wangwu, 王五, 111, 22, 1001
<==        Row: 6, wangwu, 王五, 111, 22, 1001
<==        Row: 7, wangwu, 王五, 111, 22, 1001
<==        Row: 8, aaa, bbbb, null, null, null
<==        Row: 9, aaa, bbbb, null, null, null
<==        Row: 10, aaa, bbbb, null, null, null
<==        Row: 11, aaa, bbbb, null, null, null
<==        Row: 12, aaa, bbbb, null, null, null
<==        Row: 666, hibernate, 持久层框架, null, null, null
<==      Total: 13
13
---------
13
```

可以看到第二次查询没有经过数据库操作

不同Session的相同操作

```
    @Test
    public void test2() throws  Exception{
        // 1.获取配置文件
        InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
        // 2.加载解析配置文件并获取SqlSessionFactory对象
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
        // 3.根据SqlSessionFactory对象获取SqlSession对象
        SqlSession sqlSession = factory.openSession();
        // 4.通过SqlSession中提供的 API方法来操作数据库
        List<User> list =
sqlSession.selectList("com.gupaoedu.mapper.UserMapper.selectUserList");
        System.out.println(list.size());
        sqlSession.close();
        sqlSession = factory.openSession();
        // 一级缓存测试
        System.out.println("---------");
        list =
sqlSession.selectList("com.gupaoedu.mapper.UserMapper.selectUserList");
```

```
        System.out.println(list.size());
        // 5.关闭会话
        sqlSession.close();
    }
```

输出结果

```
Setting autocommit to false on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
==>  Preparing: select * from t_user
==> Parameters:
<==    Columns: id, user_name, real_name, password, age, d_id
<==        Row: 1, zhangsan, 张三, 123456, 18, null
<==        Row: 2, lisi, 李四, 11111, 19, null
<==        Row: 3, wangwu, 王五, 111, 22, 1001
<==        Row: 4, wangwu, 王五, 111, 22, 1001
<==        Row: 5, wangwu, 王五, 111, 22, 1001
<==        Row: 6, wangwu, 王五, 111, 22, 1001
<==        Row: 7, wangwu, 王五, 111, 22, 1001
<==        Row: 8, aaa, bbbb, null, null, null
<==        Row: 9, aaa, bbbb, null, null, null
<==        Row: 10, aaa, bbbb, null, null, null
<==        Row: 11, aaa, bbbb, null, null, null
<==        Row: 12, aaa, bbbb, null, null, null
<==        Row: 666, hibernate, 持久层框架, null, null, null
<==      Total: 13
13
Resetting autocommit to true on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
Returned connection 1199262943 to pool.
---------
Opening JDBC Connection
Checked out connection 1199262943 from pool.
Setting autocommit to false on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
==>  Preparing: select * from t_user
==> Parameters:
<==    Columns: id, user_name, real_name, password, age, d_id
<==        Row: 1, zhangsan, 张三, 123456, 18, null
<==        Row: 2, lisi, 李四, 11111, 19, null
<==        Row: 3, wangwu, 王五, 111, 22, 1001
<==        Row: 4, wangwu, 王五, 111, 22, 1001
<==        Row: 5, wangwu, 王五, 111, 22, 1001
<==        Row: 6, wangwu, 王五, 111, 22, 1001
<==        Row: 7, wangwu, 王五, 111, 22, 1001
<==        Row: 8, aaa, bbbb, null, null, null
<==        Row: 9, aaa, bbbb, null, null, null
<==        Row: 10, aaa, bbbb, null, null, null
<==        Row: 11, aaa, bbbb, null, null, null
<==        Row: 12, aaa, bbbb, null, null, null
<==        Row: 666, hibernate, 持久层框架, null, null, null
<==      Total: 13
13
Resetting autocommit to true on JDBC Connection
[com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@477b4cdf]
```

```
Returned connection 1199262943 to pool.
```

通过输出我们能够发现，不同的Session中的相同操作，一级缓存是没有起作用的。

4.3 二级缓存

二级缓存是用来解决一级缓存不能跨会话共享的问题的，范围是namespace级别的，可以被多个SqlSession共享（只要是同一个接口里面的相同方法，都可以共享），生命周期和应用同步。

二级缓存的设置，首先是settings中的cacheEnabled要设置为true，当然默认的就是为true，这个步骤决定了在创建Executor对象的时候是否通过CachingExecutor来装饰。

```
if (ExecutorType.BATCH == executorType) {
  executor = new BatchExecutor( configuration: this, transaction);
} else if (ExecutorType.REUSE == executorType) { // 针对 Statement 对象做缓
  executor = new ReuseExecutor( configuration: this, transaction);
} else {
  // 默认 SimpleExecutor 每一次只是SQL操作都创建一个新的Statement对象
  executor = new SimpleExecutor( configuration: this, transaction);
}
// 二级缓存开关，settings 中的 cacheEnabled 默认是 true
// 映射文件中 <cache> 标签 --> 创建 Cache对象
// settings 中的 cacheEnabled = true 真正的对 Executor 做了缓存的增强
if (cacheEnabled) {
  // 穿衣服的事情 --> 装饰器模式
  executor = new CachingExecutor(executor);
}
// 植入插件的逻辑，至此，四大对象已经全部拦截完毕
executor = (Executor) interceptorChain.pluginAll(executor);
return executor;
}
```

那么设置了cacheEnabled标签为true是否就意味着 二级缓存是否一定可用呢？当然不是，我们还需要在 对应的映射文件中添加 cache 标签才行。

```
<!-- 声明这个namespace使用二级缓存 -->
<cache type="org.apache.ibatis.cache.impl.PerpetualCache"
       size="1024"  <!–最多缓存对象个数，默认1024-->
       eviction="LRU" <!–回收策略-->
       flushInterval="120000" <!–自动刷新时间 ms，未配置时只有调用时刷新-->
       readOnly="false"/> <!–默认是false（安全），改为true可读写时，对象必须支持序列化 -->
```

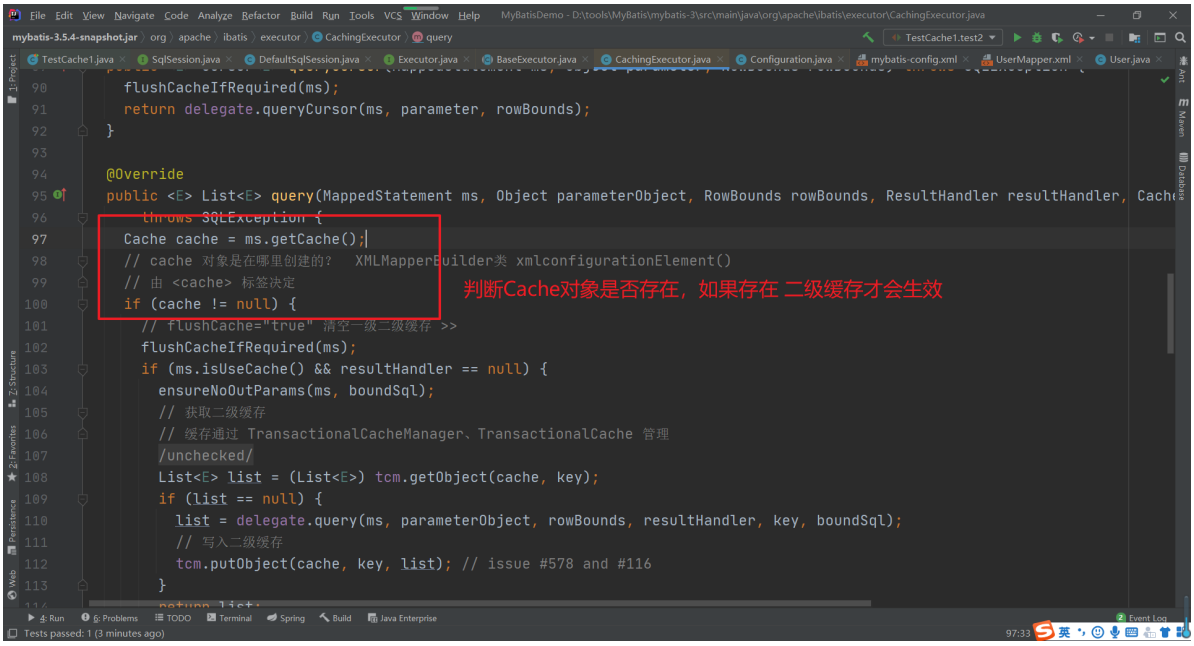
cache属性详解：

属性	含义	取值
type	缓存实现类	需要实现Cache接口，默认是PerpetualCache，可以使用第三方缓存
size	最多缓存对象个数	默认1024
eviction	回收策略（缓存淘汰算法）	LRU – 最近最少使用的：移除最长时间不被使用的对象（默认）。FIFO – 先进先出：按对象进入缓存的顺序来移除它们。SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。
flushInterval	定时自动清空缓存间隔	自动刷新时间，单位 ms，未配置时只有调用时刷新
readOnly	是否只读	true：只读缓存；会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。false：读写缓存；会返回缓存对象的拷贝（通过序列化），不会共享。这会慢一些，但是安全，因此默认是 false。改为false可读写时，对象必须支持序列化。
blocking	启用阻塞缓存	通过在get/put方式中加锁，保证只有一个线程操作缓存，基于Java重入锁实现

再来看下cache标签在源码中的体现，创建cacheKey

```
  @Override
  public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, ResultHandler resultHandler) throws SQLException {
    // 获取SQL
    BoundSql boundSql = ms.getBoundSql(parameterObject);
    // 创建CacheKey：什么样的SQL是同一条SQL？ >>
    CacheKey key = createCacheKey(ms, parameterObject, rowBounds, boundSql);
    return query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
  }
```

createCacheKey自行进去查看




```
public final class MappedStatement {

  private String resource;
  private Configuration configuration;
  private String id;
  private Integer fetchSize;
  private Integer timeout;
  private StatementType statementType;
  private ResultSetType resultSetType;
  private SqlSource sqlSource;
  private Cache cache;
  private ParameterMap parameterMap;
  private List<ResultMap> resultMaps;
  private boolean flushCacheRequired;
  private boolean useCache;
  private boolean resultOrdered;
  private SqlCommandType sqlCommandType;
  private KeyGenerator keyGenerator;
  private String[] keyProperties;
  private String[] keyColumns;
  private boolean hasNestedResultMaps;
  private String databaseId;
```

而这看到的和我们前面在缓存初始化时看到的 cache 标签解析操作是对应上的。所以我们要开启二级缓存两个条件都要满足。

```
<settings>
    <!-- 打印查询语句 ctrl+alt + /<- -->
    <setting name="logImpl" value="STDOUT_LOGGING" />

    <!-- 控制全局缓存（二级缓存），默认 true-->
    <setting name="cacheEnabled" value="true"/>

    <!-- 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。默认 false  -->
    <setting name="lazyLoadingEnabled" value="true"/>
    <!-- 当开启时，任何方法的调用都会加载该对象的所有属性。默认 false，可通过select标
    <setting name="aggressiveLazyLoading" value="false"/>
    <!--  Mybatis 创建具有延迟加载能力的对象所用到的代理工具，默认JAVASSIST -->
    <!--<setting name="proxyFactory" value="CGLIB" />-->
    <!-- STATEMENT级别的缓存，使一级缓存，只针对当前执行的这一statement有效 -->
    <!---->
```

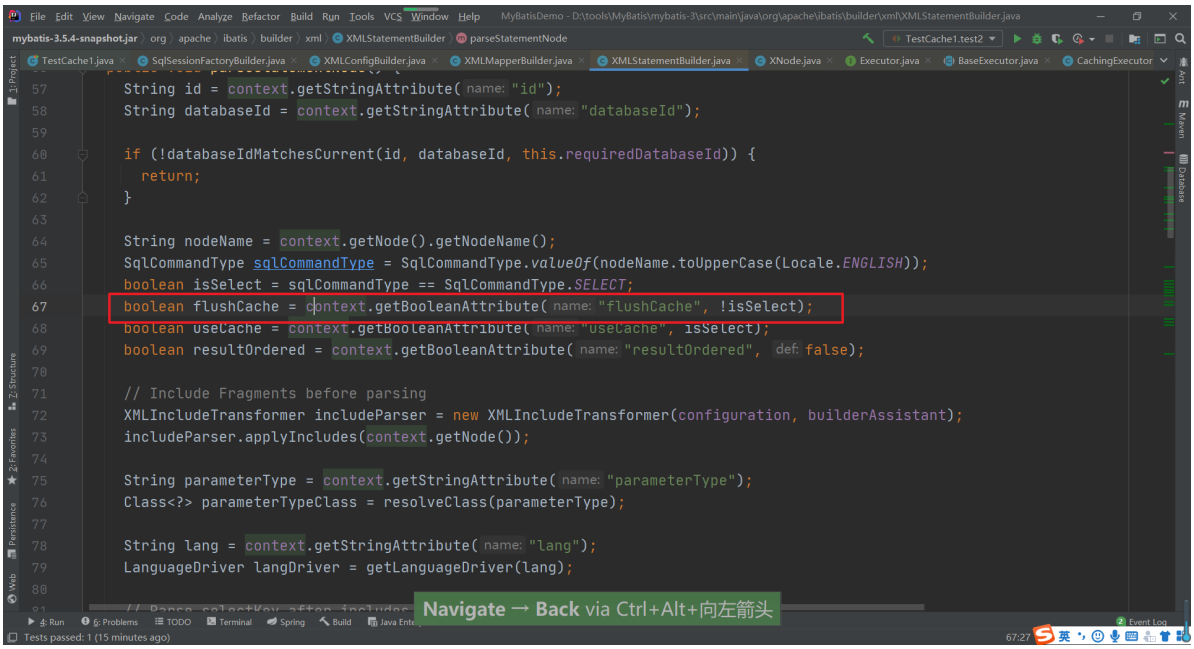
这样的设置表示当前的映射文件中的相关查询操作都会触发二级缓存，但如果某些个别方法我们不希望走二级缓存怎么办呢？我们可以在标签中添加一个 useCache=false 来实现的设置不使用二级缓存

```
    <sql id="baseSQL"...>
<select id="selectUserById"    resultMap="BaseResultMap" statementType="PREPARED" parameterType="_int"..

<!-- $只能用在自定义类型和map上 -->
<select id="selectUserByBean"  parameterType="user" resultType="user" useCache="false">
        select * from t_user where user_name = '${userName}'
    </select>

<select id="selectUserList" resultMap="BaseResultMap" >
        select * from t_user
    </select>
```

还有就是当我们执行的对应的DML操作，在MyBatis中会清空对应的二级缓存和一级缓存。

```
private void flushCacheIfRequired(MappedStatement ms) {
  Cache cache = ms.getCache();
  // 增删改查的标签上有属性：flushCache="true" （select语句默认是false）
  // 一级二级缓存都会被清理
  if (cache != null && ms.isFlushCacheRequired()) {
    tcm.clear(cache);
  }
}
```

在解析映射文件的时候DML操作flushCacheRequired为true



4.4 第三方缓存

在实际开发的时候我们一般也很少使用MyBatis自带的二级缓存，这时我们会使用第三方的缓存工具Ehcache获取Redis来实现,那么他们是如何来实现的呢？

https://github.com/mybatis/redis-cache

添加依赖

```
<dependency>
   <groupId>org.mybatis.caches</groupId>
   <artifactId>mybatis-redis</artifactId>
   <version>1.0.0-beta2</version>
</dependency>
```

然后加上Cache标签的配置

```
  <cache type="org.mybatis.caches.redis.RedisCache"
         eviction="FIFO"
         flushInterval="60000"
         size="512"
         readOnly="true"/>
```

然后添加redis的属性文件

```
host=192.168.100.120
port=6379
connectionTimeout=5000
soTimeout=5000
database=0
```