spring 事务

1)事务回顾

1.1)什么是事务?

事务指数据库中多个操作合并在一起形成的操作序列

1.2)事务的作用

1.当数据库操作序列中个别操作失败时，提供一种方式使数据库状态恢复到正常状态（**A**），保障数据库即使在异常状态下仍能保持数据一致性（**C**）（要么操作前状态，要么操作后状态）。

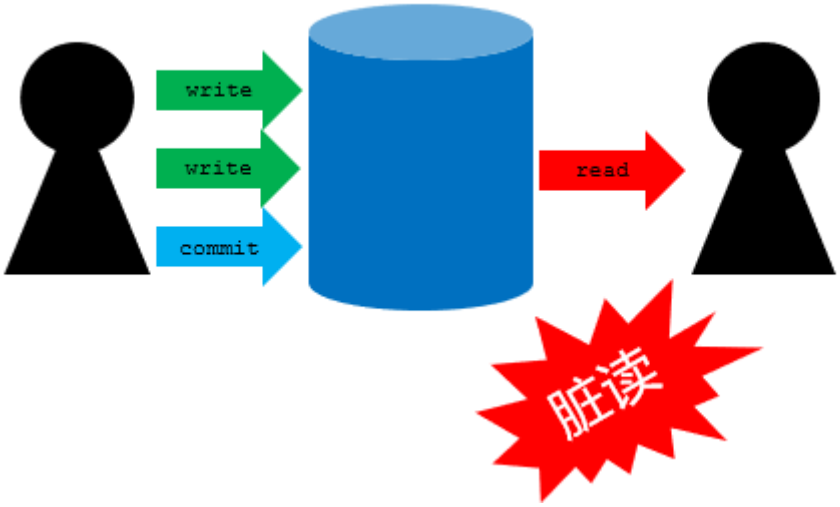
2.当出现并发访问数据库时，在多个访问间进行相互隔离，防止并发访问操作结果互相干扰（**I**）。

- 事务特征（ACID）
  - 原子性（Atomicity）指事务是一个不可分割的整体，其中的操作要么全执行或全不执行
  - 一致性（Consistency）事务前后数据的完整性必须保持一致
  - 隔离性（Isolation）事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离
  - 持久性（Durability）持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响

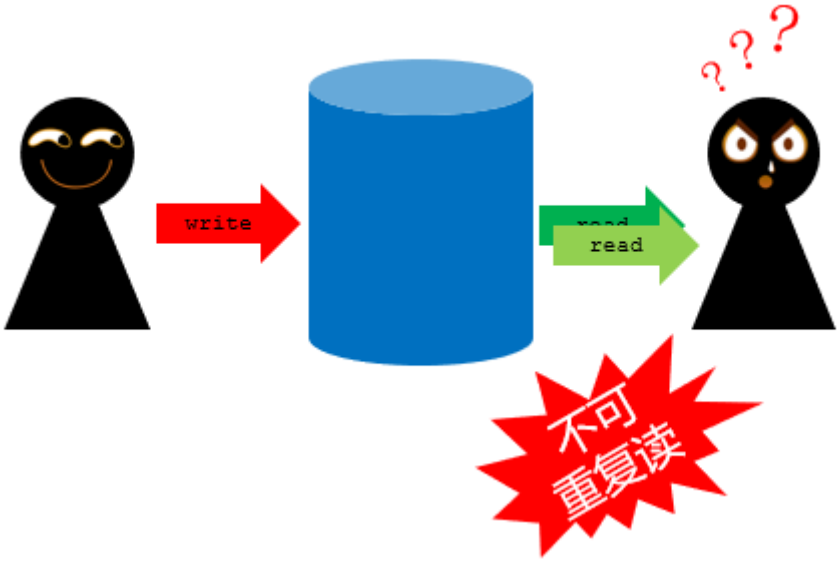
1.3)事务的隔离级

- 脏读：允许读取未提交的信息
  - 原因：Read uncommitted

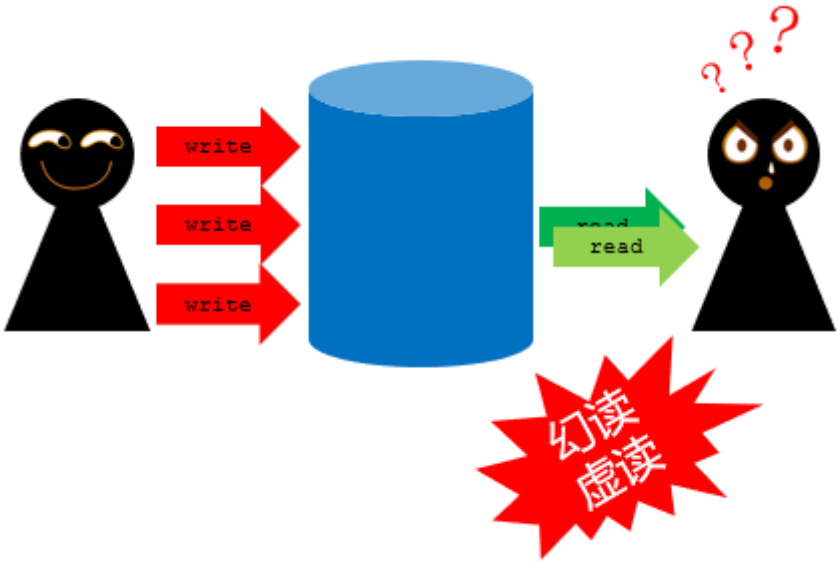
  解决方案： （表级读锁）



- 不可重复读：读取过程中单个数据发生了变化
  - 解决方案： Repeatable read （行级写锁）



- 幻读：读取过程中数据条目发生了变化
  - 解决方案： Serializable（表级写锁）



2)事务管理

2.1)Spring事务核心对象

- J2EE开发使用分层设计的思想进行，对于简单的业务层转调数据层的单一操作，事务开启在业务层或者数据层并无太大差别，当业务中包含多个数据层的调用时，需要在业务层开启事务，对数据层中多个操作进行组合并归属于同一个事务进行处理
- Spring为业务层提供了整套的事务解决方案
  - PlatformTransactionManager
  - TransactionDefinition
  - TransactionStatus

2.2)PlatformTransactionManager

- 平台事务管理器实现类

    - DataSourceTransactionManager 适用于Spring JDBC或MyBatis
    - HibernateTransactionManager 适用于Hibernate3.0及以上版本
    - JpaTransactionManager 适用于JPA
    - JdoTransactionManager 适用于JDO
    - JtaTransactionManager 适用于JTA

- JPA（Java Persistence API）Java EE 标准之一，为POJO提供持久化标准规范，并规范了持久化开发的统一API，符合JPA规范的开发可以在不同的JPA框架下运行
- JDO(Java Data Object)是Java对象持久化规范，用于存取某种数据库中的对象，并提供标准化API。与JDBC相比，JDBC仅针对关系数据库进行操作，JDO可以扩展到关系数据库、文件、XML、对象数据库（ODBMS）等，可移植性更强
- JTA（Java Transaction API）Java EE 标准之一，允许应用程序执行分布式事务处理。与JDBC相比，JDBC事务则被限定在一个单一的数据库连接，而一个JTA事务可以有多个参与者，比如JDBC连接、JDO 都可以参与到一个JTA事务中

此接口定义了事务的基本操作

- 获取事务：

```
TransactionStatus getTransaction(TransactionDefinition definition)
```

- 提交事务：

```
void commit(TransactionStatus status)
```

- 回滚事务：

```
void rollback(TransactionStatus status)
```

2.3)TransactionDefinition

此接口定义了事务的基本信息

- 获取事务定义名称

```
String getName()
```

- 获取事务的读写属性

```
boolean isReadOnly()
```

- 获取事务隔离级别

```
int getIsolationLevel()
```

- 获事务超时时间

```
int getTimeout()
```

- 获取事务传播行为特征

```
int getPropagationBehavior()
```

2.4)TransactionStatus

此接口定义了事务在执行过程中某个时间点上的状态信息及对应的状态操作

◆ 获取事务是否处于新开启事务状态

```
boolean isNewTransaction()
```

◆ 刷新事务状态

```
void flush()
```

◆ 获取事务是否处于已完成状态

```
boolean isCompleted()
```

◆ 获取事务是否具有回滚存储点

```
boolean hasSavepoint()
```

◆ 获取事务是否处于回滚状态

```
boolean isRollbackOnly()
```

◆ 设置事务处于回滚状态

```
void setRollbackOnly()
```

2.5)事务控制方式

- 编程式
- 声明式（XML）
- 声明式（注解）

2.6)案例说明

2.6.1)案例说明

银行转账业务说明

银行转账操作中，涉及从A账户到B账户的资金转移操作。数据层仅提供单条数据的基础操作，未设计多账户间的业务操作。

2.6.2)案例环境（基于Spring、Mybatis整合）

- 业务层接口提供转账操作

```
/**
* 转账操作
* @param outName      出账用户名
* @param inName       入账用户名
* @param money        转账金额
*/
public void transfer(String outName,String inName,Double money);
```

- 业务层实现提供转账操作

```
public void transfer(String outName,String inName,Double money){
    accountDao.inMoney(outName,money);
            accountDao.outMoney(inName,money);
}
```

- 数据层提供对应的入账与出账操作

```
<update id="inMoney">
    update account set money = money + #{money} where name = #{name}
</update>
<update id="outMoney">
    update account set money = money - #{money} where name = #{name}
</update>
```

2.6.3)编程式事务

```
public void transfer(String outName,String inName,Double money){
    //创建事务管理器
    DataSourceTransactionManager dstm = new DataSourceTransactionManager();
    //为事务管理器设置与数据层相同的数据源
    dstm.setDataSource(dataSource);
    //创建事务定义对象
    TransactionDefinition td = new DefaultTransactionDefinition();
    //创建事务状态对象，用于控制事务执行
    TransactionStatus ts = dstm.getTransaction(td);
    accountDao.inMoney(outName,money);
    int i = 1/0;    //模拟业务层事务过程中出现错误
    accountDao.outMoney(inName,money);
    //提交事务
    dstm.commit(ts);
}
```

2.7)使用AOP控制事务

将业务层的事务处理功能抽取出来制作成AOP通知，利用环绕通知运行期动态织入

```
public Object tx(ProceedingJoinPoint pjp) throws Throwable {

    DataSourceTransactionManager dstm = new DataSourceTransactionManager();
    dstm.setDataSource(dataSource);
    TransactionDefinition td = new DefaultTransactionDefinition();
    TransactionStatus ts = dstm.getTransaction(td);
    Object ret = pjp.proceed(pjp.getArgs());
    dstm.commit(ts);

    return ret;
}
```

配置AOP通知类，并注入dataSource

```
<bean id="txAdvice" class="com.itheima.aop.TxAdvice">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

使用环绕通知将通知类织入到原始业务对象执行过程中

```xml
<aop:config>
    <aop:pointcut id="pt" expression="execution(* *..transfer(..))"/>
    <aop:aspect ref="txAdvice">
        <aop:around method="tx" pointcut-ref="pt"/>
    </aop:aspect>
</aop:config>
```

2.8声明式事务（XML）

AOP配置事务是否具有特例性?

```java
public Object tx(ProceedingJoinPoint pjp) throws Throwable {
    DataSourceTransactionManager dstm = new DataSourceTransactionManager();
    dstm.setDataSource(dataSource);
    TransactionDefinition td = new DefaultTransactionDefinition();
    TransactionStatus ts = dstm.getTransaction(td);
    Object ret = pjp.proceed(pjp.getArgs());
    dstm.commit(ts);

    return ret;
}
```

```xml
<bean id="txAdvice" class="com.itheima.aop.TxAdvice">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

使用tx命名空间配置事务专属通知类

```xml
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="*" read-only="false" />
        <tx:method name="get*" read-only="true" />
        <tx:method name="find*" read-only="true" />
    </tx:attributes>
</tx:advice>
```

使用aop:advisor在AOP配置中引用事务专属通知类

```xml
<aop:config>
 <aop:pointcut id="pt" expression="execution(* *..*(..))"/> //千万不能使用这种格式
 <aop:advisor advice-ref="txAdvice" pointcut-ref="pt"/>
</aop:config>
// execution(* *..*(..)) 不能这样写,这种设会把所有方法挂上aop,会出大问题的.通常是加在接口上
// execution(* com.huizan.service.*Service.*(..))  正确写法.监控接口中的所有方法,接口业务层精准锁定
```

2.8.1)aop:advice与aop:advisor区别

- aop:advice配置的通知类可以是普通java对象，不实现接口，也不使用继承关系
- aop:advisor配置的通知类必须实现通知接口
    - MethodBeforeAdvice

- AfterReturningAdvice
- ThrowsAdvice
- ……

2.8.2)tx配置----tx:advice

- 名称：tx:advice
- 类型：**标签**
- 归属：beans标签
- 作用：专用于声明事务通知
- 格式：

```
<beans>
    <tx:advice id="txAdvice" transaction-manager="txManager">
    </tx:advice>
</beans>
```

- 基本属性：
  - id ： 用于配置aop时指定通知器的id
  - transaction-manager ： 指定事务管理器bean

2.8.3)tx配置----tx:attributes

- 名称：tx:attributes
- 类型：**标签**
- 归属：tx:advice标签
- 作用：定义通知属性
- 格式：

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
    </tx:attributes>
</tx:advice>
```

- 基本属性：
  - 无

2.8.4)tx配置----tx:method

- 名称：tx:method
- 类型：**标签**
- 归属：tx:attribute标签
- 作用：设置具体的事务属性
- 格式：

```
<tx:attributes>
    <tx:method name="*" read-only="false" />
    <tx:method name="get*" read-only="true" />
</tx:attributes>
```

- 说明:

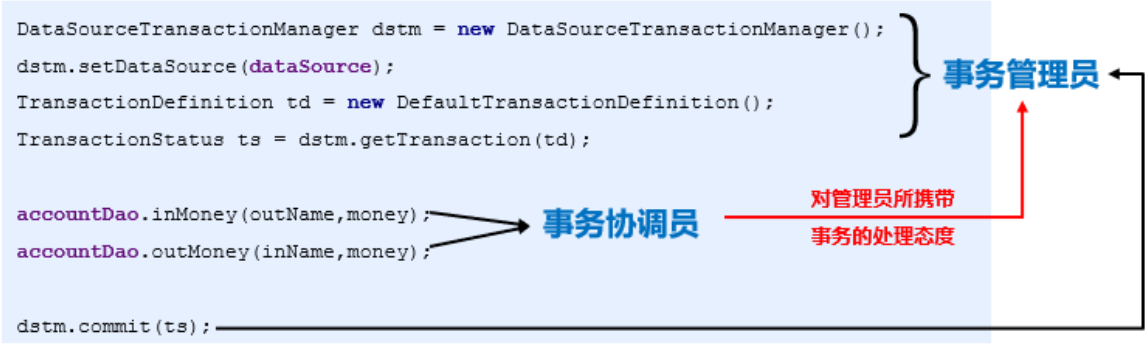
  通常事务属性会配置多个，包含1个读写的全事务属性，1个只读的查询类事务属性

tx:method属性

```
<tx:method
    name="*"                  待添加事务的方法名表达式（支持*号通配符），例如get*、*、.....
    read-only="false"         设置事务的读写属性，true为只读，false为读写
    timeout="-1"              设置事务超时时长，单位秒
    isolation="DEFAULT"       设置事务隔离级别，该隔离级设定是基于Spring的设定，非数据库端
    no-rollback-for=""        设置事务中不回滚的异常，多个异常间使用，分割
    rollback-for=""           设置事务中必回滚的异常，多个异常间使用，分割
    propagation="REQUIRED"    设置事务的传播行为
    />
```

2.9)事务传播行为

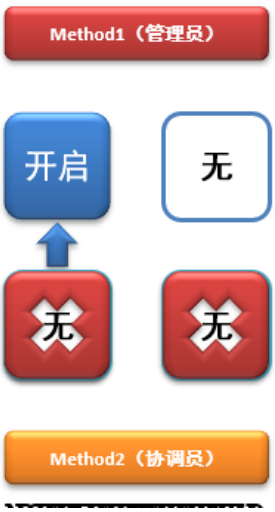
- 事务管理员
- 事务协调员



- 事务传播行为描述的是事务协调员对事务管理员所携带事务的处理态度

2.10)事务传播行为

传播属性	事务管理员	事务协调员
REQUIRED		
REQUIRES_NEW		
SUPPORTS		
NOT_SUPPORTED		
MANDATORY		
NEVER		
NESTED		



2.11)事务传播应用

- 场景A：生成订单业务
  - 子业务S1：记录日志到数据库表X
  - 子业务S2：保存订单数据到数据库表Y

    - 子业务S3：.....
    - 如果S2或S3或.....事务提交失败，此时S1是否回滚？如何控制？
    - （S1需要新事务）
- 场景B：生成订单业务
    - 背景1：订单号生成依赖数据库中一个专门用于控制订单号编号生成的表M获取
    - 背景2：每次获取完订单号，表M中记录的编号自增1
    - 子业务S1：从表M中获取订单编号
    - 子业务S2：保存订单数据，订单编号来自于表M
    - 子业务S3：.....
    - 如果S2或S3或.....事务提交失败，此时S1是否回滚？如何控制？
    - （S1需要新事务）

2.12)声明式事务（注解）

2.12.1)@Transactional

- 名称：@Transactional
- 类型：**方法注解，类注解，接口注解**
- 位置：方法定义上方，类定义上方，接口定义上方
- 作用：设置当前类/接口中所有方法或具体方法开启事务，并指定相关事务属性
- 范例：

```
@Transactional(
    readOnly = false,
    timeout = -1,
    isolation = Isolation.DEFAULT,
    rollbackFor = {ArithmeticException.class, IOException.class},
    noRollbackFor = {},
    propagation = Propagation.REQUIRES_NEW
)
```

2.12.2)tx:annotation-driven

- 名称：tx:annotation-driven
- 类型：**标签**
- 归属：beans标签
- 作用：开启事务注解驱动，并指定对应的事务管理器
- 范例：

```
<tx:annotation-driven transaction-manager="txManager"/>
```

2.13)声明式事务（纯注解驱动）

- 名称：@EnableTransactionManagement
- 类型：**类注解**
- 位置：Spring注解配置类上方
- 作用：开启注解驱动，等同XML格式中的注解驱动
- 范例：

```
@Configuration
@ComponentScan("com.itheima")
@PropertySource("classpath:jdbc.properties")
@Import({JDBCConfig.class,MyBatisConfig.class,TransactionManagerConfig.class
})
@EnableTransactionManagement
public class SpringConfig {
}
```

```
public class TransactionManagerConfig {
    @Bean
    public PlatformTransactionManager getTransactionManager(@Autowired
DataSource dataSource){
        return new DataSourceTransactionManager(dataSource);
    }
}
```

3)模板对象

3.1)Spring模块对象

- TransactionTemplate
- JdbcTemplate
- RedisTemplate
- RabbitTemplate
- JmsTemplate
- HibernateTemplate
- RestTemplate

3.2)JdbcTemplate（了解）

提供标准的sql语句操作API

```
public void save(Account account) {
    String sql = "insert into account(name,money)values(?,?)";
    jdbcTemplate.update(sql,account.getName(),account.getMoney());
}
```

3.3)NamedParameterJdbcTemplate(了解)

提供标准的具名sql语句操作API

```
public void save(Account account) {
    String sql = "insert into account(name,money)values(:name,:money)";
    Map pm = new HashMap();
    pm.put("name",account.getName());
    pm.put("money",account.getMoney());
    jdbcTemplate.update(sql,pm);
}
```

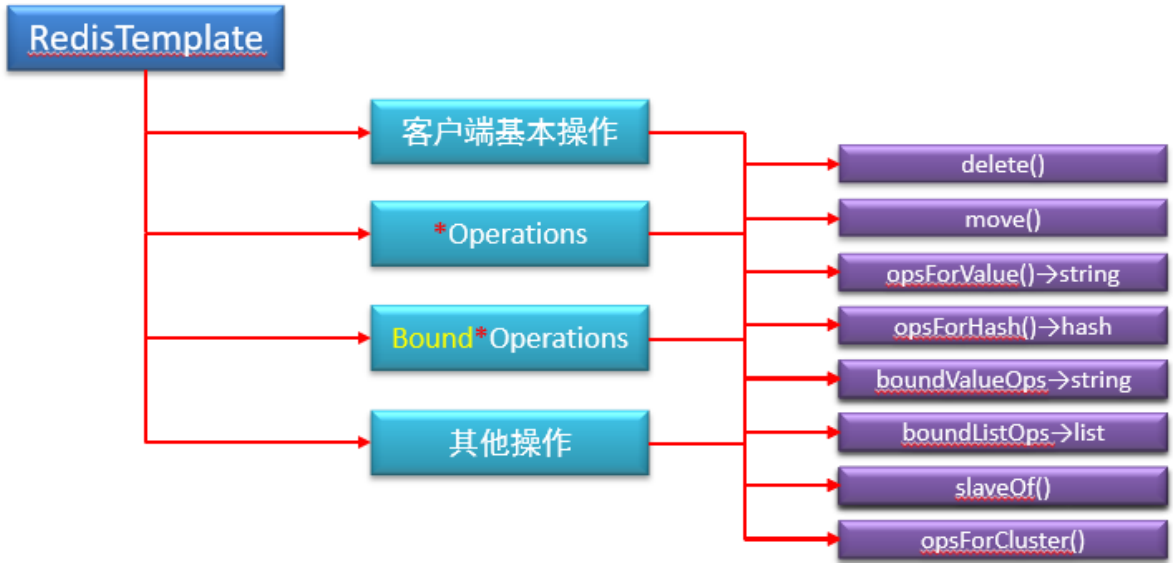
3.4)RedisTemplate

环境配置

redis默认protected-mode yes ,是开启远程保护模式,java中的RedisTemplate会报错.

把protected-mode 改成no ,

RedisTemplate对象结构

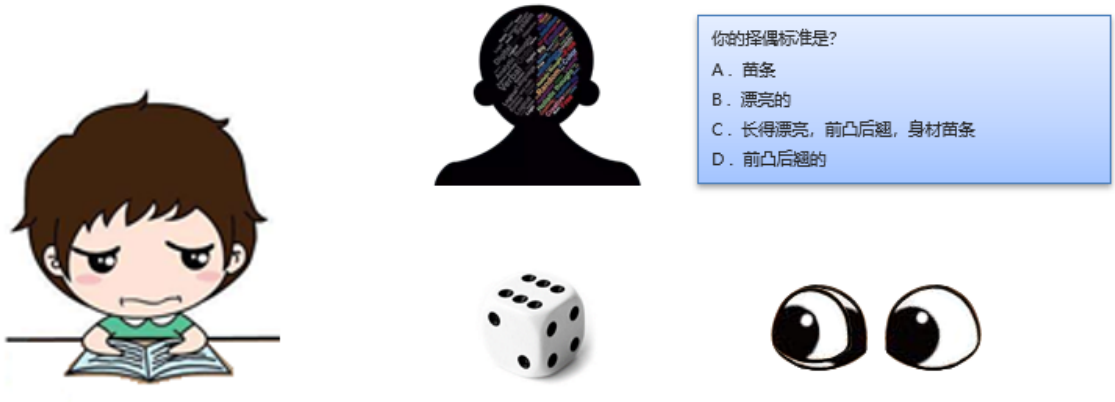



```
public void changeMoney(Integer id, Double money) {
    redisTemplate.opsForValue().set("account:id:"+id,money);
}
public Double findMondyById(Integer id) {
    Object money = redisTemplate.opsForValue().get("account:id:" + id);
    return new Double(money.toString());
}
```

4)事务底层原理解析

4.1)策略模式应用

策略模式（Strategy Pattern）使用不同策略的对象实现不同的行为方式，策略对象的变化导致行为的变化。



策略模式（Strategy Pattern）使用不同策略的对象实现不同的行为方式，策略对象的变化导致行为的变化。

```
public Account findById(Integer id) {
    String sql = "select * from account where id = ? ";
    //支持自定义行映射解析器
    RowMapper<Account> rm = new RowMapper<Account>() {
        public Account mapRow(ResultSet rs, int rowNum) throws SQLException {
            Account account = new Account();
            account.setId(rs.getInt("id"));
            account.setName(rs.getString("name"));
            account.setMoney(rs.getDouble("money"));
            return account;
        }};
    return jdbcTemplate.queryForObject(sql,rm,id);
}
public List<Account> findAll() {
    String sql = "select * from account";
    //使用spring自带的行映射解析器，要求必须是标准封装
    return jdbcTemplate.query(sql,new BeanPropertyRowMapper<Account>(Account.class));
}
```