第二章

数 据 连 接

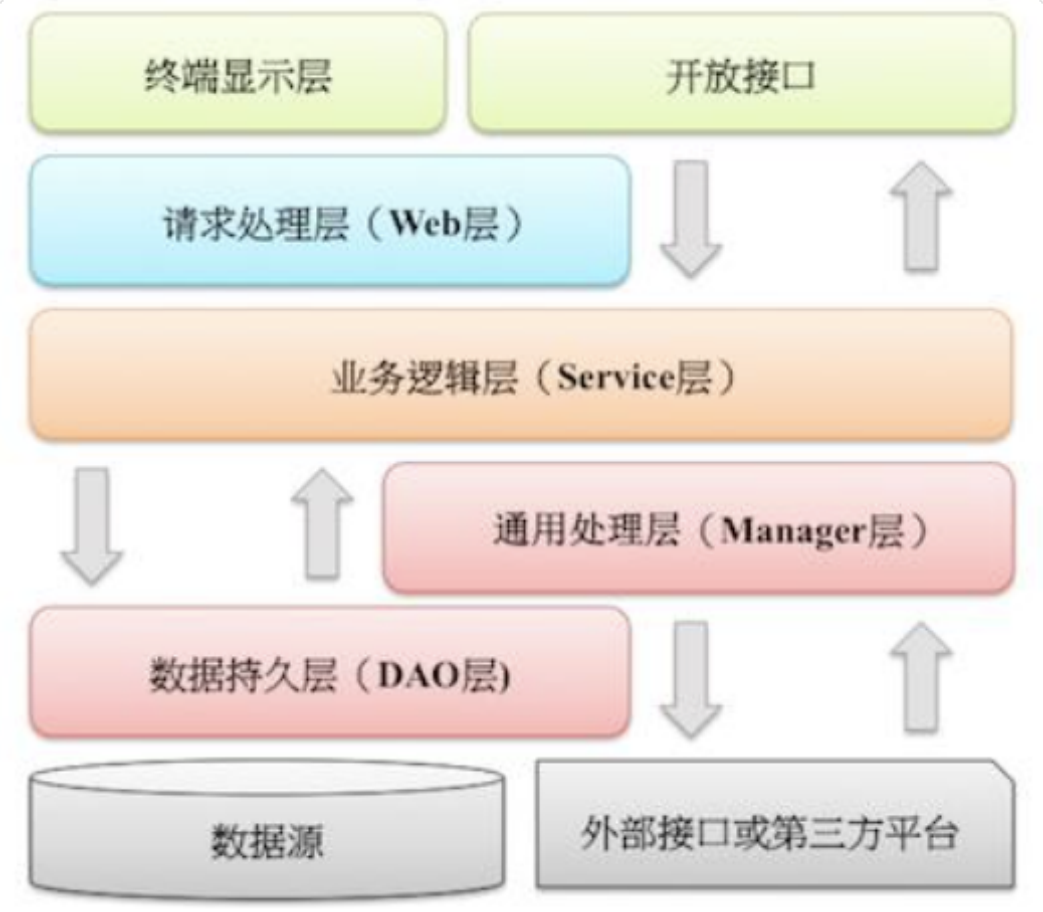
—— Spring的数据连接

内容提要

- 什么是数据访问层
- JDBC数据连接
- 数据连接池
- 通过Spring JDBC访问数据库
- 事务管理

分层软件架构 – DAO层

- 《阿里巴巴Java开发手册》
  - 六、工程架构 （一）应用分层



数据源类型

- 文本存储（FTP/SFTP/OSS/多媒体文件等）
- 数据库（Oracle/H2/MySQL/PostgreSQL等）
- NoSQL（Memcache/Redis/MongoDB/HBase等）
- 大数据（MaxCompute/分析型数据库MySQL版/HDFS等）

内容提要

- 什么是数据访问层
- JDBC数据连接
- 数据连接池
- 通过Spring JDBC访问数据库
- 事务管理

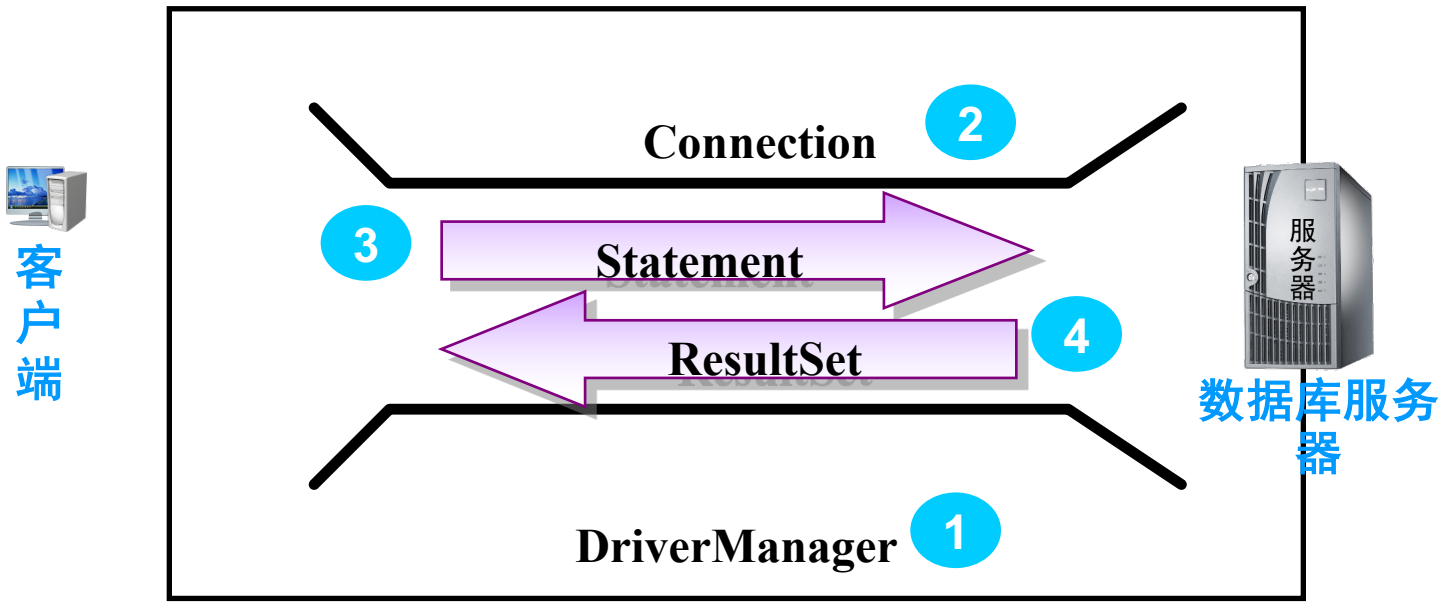
JDBC接口技术

- JDBC(Java Database Connectivity)接口技术实际上是一种通过Java语言访问任何结构化数据库的应用程序接口(API)

- JDBC 可做三件事：与数据库建立连接、执行SQL 语句、处理结果

- 各个数据库厂商提供各自的JDBC驱动程序，e.g. com.mysql.cj.jdbc.Driver, oracle.jdbc.driver.OracleDriver 等

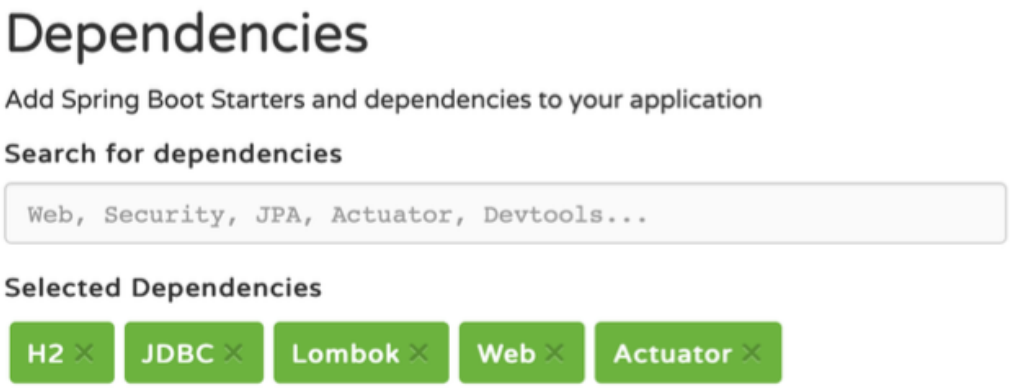
数据库直接连接



DriverManager ： 依据数据库的不同，管理JDBC驱动
Connection ： 负责连接数据库并担任传送数据的任务
Statement ： 由 Connection 产生、负责执行SQL语句
ResultSet：负责保存Statement执行后所产生的查询结果

Spring Boot如何配置数据源

- 引入对应数据库驱动——H2
- 引入 JDBC 依赖——spring-boot-starter-jdbc
- 获取 DataSource Bean，打印信息
- 也可通过 /acturator/beans 查看 Bean



```java
@SpringBootApplication
@Slf4j
public class DataSourceDemoApplication implements CommandLineRunner {
        @Autowired
        private DataSource dataSource;

        public static void main(String[] args) {
                SpringApplication.run(DataSourceDemoApplication.class, args);
        }

        @Override
        public void run(String... args) throws Exception {
                log.info(dataSource.toString());
                Connection conn = dataSource.getConnection();
                log.info(conn.toString());
                conn.close();
        }
}
```

```
HikariDataSource (null)
HikariPool-1 - Starting...
HikariPool-1 - Start completed.
HikariProxyConnection@5181771 wrapping conn0: url=jdbc:h2:mem:testdb user=SA
```

直接配置所需的Bean

数据源相关

- DataSource（根据选择的连接池实现决定）

事务相关（可选）

- PlatformTransactionManager（DataSourceTransactionManager）
- TransactionTemplate

操作相关（可选）

- JdbcTemplate

```java
@Configuration
@EnableTransactionManagement
public class DataSourceDemo {
    @Autowired
    private DataSource dataSource;

    public static void main(String[] args) throws SQLException {
        ApplicationContext applicationContext =
                new ClassPathXmlApplicationContext("applicationContext*.xml");
        showBeans(applicationContext);
        dataSourceDemo(applicationContext);
    }

    @Bean(destroyMethod = "close")
    public DataSource dataSource() throws Exception {
        Properties properties = new Properties();
        properties.setProperty("driverClassName", "org.h2.Driver");
        properties.setProperty("url", "jdbc:h2:mem:testdb");
        properties.setProperty("username", "sa");
        return BasicDataSourceFactory.createDataSource(properties);
    }

    @Bean
    public PlatformTransactionManager transactionManager() throws Exception {
        return new DataSourceTransactionManager(dataSource());
    }
```

Spring Boot 做了哪些配置

DataSourceAutoConfiguration

- 配置 DataSource

DataSourceTransactionManagerAutoConfiguration

- 配置 DataSourceTransactionManager

JdbcTemplateAutoConfiguration

- 配置 JdbcTemplate

符合条件时才进行配置

数据源相关配置属性

simple-jdbc-demo

通用

- spring.datasource.url=jdbc:mysql://localhost/test
- spring.datasource.username=dbuser
- spring.datasource.password=dbpass
- spring.datasource.driver-class-name=com.mysql.jdbc.Driver（可选）

初始化内嵌数据库

- spring.datasource.initialization-mode=embedded|always|never
- spring.datasource.schema与spring.datasource.data确定初始化SQL文件
- spring.datasource.platform=hsqldb | h2 | oracle | mysql | postgresql（与前者对应）

使用数据直接连接的缺点

- **在某一时刻连接必须服务于一个用户，以免造成事务冲突**
  - 来自不同用户的请求（都使用了同一个连接）对相同的事务进行操作，如果一个请求试图回滚，那么所有使用相同连接的数据库操作都要被回滚。
- **创建连接需要耗费时间**
  - 创建一个连接大概需要**1-2**秒的时间。
- **保持连接打开状态的代价很大**
  - 尤其是在系统资源（例如内存）方面。
  - 数据库产品的许可证都按照同时打开的连接数目来收费。

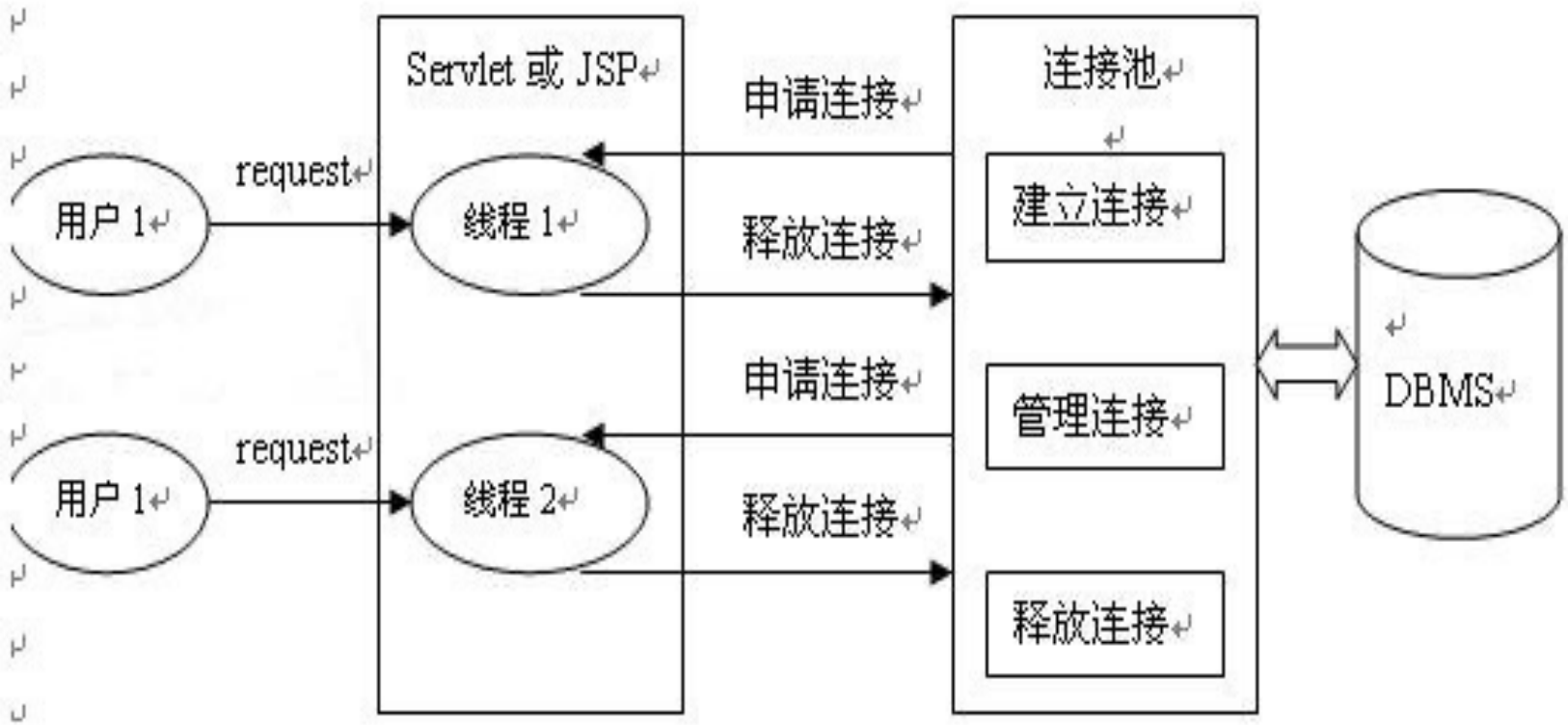
内容提要

- 什么是数据访问层
- JDBC数据连接
- 数据连接池
- 通过Spring JDBC访问数据库
- 事务管理

数据库连接池原理

- **数据库连接池运行机制**
  - 从连接池获取或创建可用连接；
  - 使用完毕之后，把连接返还给连接池；
  - 在系统关闭前，断开所有连接并释放连接占用的系统资源；
  - 还能够处理无效连接（原来登记为可用的连接，由于某种原因不再可用，如超时，通讯问题），并能够限制连接池中的连接总数不低于某个预定值和不超过某个预定值；

连接池示例图


Servlet 或 JSP
连接池
申请连接
request
用户 1
线程 1
建立连接
释放连接
申请连接
request
用户 1
线程 2
管理连接
释放连接
释放连接
DBMS

使用数据库连接池技术的好处

- 资源重用
- 更快的系统响应速度
- 统一的连接管理，避免数据库连接泄漏

常用的连接池 - HikariCP

在 Spring Boot 中的配置

Spring Boot 2.x

- 默认使用 HikariCP
- 配置 spring.datasource.hikari.* 配置

Spring Boot 1.x

- 默认使用 Tomcat 连接池，需要移除 tomcat-jdbc 依赖
- spring.datasource.type=com.zaxxer.hikari.HikariDataSource

常用 HikariCP 配置参数

datasource-demo

常用配置

- spring.datasource.hikari.maximumPoolSize=10
- spring.datasource.hikari.minimumIdle=10
- spring.datasource.hikari.idleTimeout=600000
- spring.datasource.hikari.connectionTimeout=30000
- spring.datasource.hikari.maxLifetime=1800000

其他配置详见 HikariCP 官网

- https://github.com/brettwooldridge/HikariCP

常用的连接池 – Alibaba Druid

- **Alibaba Druid 官方介绍**
  - **Druid**连接池是阿里巴巴开源的数据库连接池项目。**Druid**连接池为监控而生， 内置强大的监控功能，监控特性不影响性能。功能强大，能防**SQL**注入，内置 **Logging**能诊断**Hack**应用行为。
- **经过阿里巴巴各大系统的考验，值得信赖**
- **实用的功能**
  - 详细的监控(真的是全面)
  - **ExceptionSorter**，针对主流数据库的返回码都有支支持
  - **SQL** 防注入
  - 内置加密配置
  - 众多扩展点，方便进行定制

在 Spring Boot 中的配置

直接配置 DruidDataSource

通过 druid-spring-boot-starter

- spring.datasource.druid.*

```
spring.output.ansi.enabled=ALWAYS

spring.datasource.url=jdbc:h2:mem:foo
spring.datasource.username=sa
spring.datasource.password=n/z7PyA5cvcXvs8px8FVmBVpaRyNsvJb3X7YfS38DJrIg25EbZaZGvH4aHcnc97Om0islpCAPc3MqsGvsrxVJw==

spring.datasource.druid.initial-size=5
spring.datasource.druid.max-active=5
spring.datasource.druid.min-idle=5
spring.datasource.druid.filters=conn,config,stat,slf4j

spring.datasource.druid.connection-properties=config.decrypt=true;config.decrypt.key=${public-key}
spring.datasource.druid.filter.config.enabled=true

spring.datasource.druid.test-on-borrow=true
spring.datasource.druid.test-on-return=true
spring.datasource.druid.test-while-idle=true
```

数据源配置

druid-demo

Filter 配置

- spring.datasource.druid.filters=stat,config,wall,log4j （全部使用默认值）

密码加密

- spring.datasource.password=<加密密码>
- spring.datasource.druid.filter.config.enabled=true
- spring.datasource.druid.connection-properties=config.decrypt=true;config.decrypt.key=<public-key>

SQL 防注入

- spring.datasource.druid.filter.wall.enabled=true
- spring.datasource.druid.filter.wall.db-type=h2
- spring.datasource.druid.filter.wall.config.delete-allow=false
- spring.datasource.druid.filter.wall.config.drop-table-allow=false

Druid Filter

- 用于定制连接池操作的各种环节
- 可以继承 FilterEventAdapter 以便便方方便便地实现 Filter
- 修改 META-INF/druid-filter.properties 增加 Filter 配置

```java
@Slf4j
public class ConnectionLogFilter extends FilterEventAdapter {

    @Override
    public void connection_connectBefore(FilterChain chain, Properties info) {
        log.info("BEFORE CONNECTION!");
    }

    @Override
    public void connection_connectAfter(ConnectionProxy connection) {
        log.info("AFTER CONNECTION!");
    }
}
```

```
com.alibaba.druid.pool.DruidDataSource   : testOnBorrow is true,
g.s.data.druiddemo.ConnectionLogFilter   : BEFORE CONNECTION!
g.s.data.druiddemo.ConnectionLogFilter   : AFTER CONNECTION!
g.s.data.druiddemo.ConnectionLogFilter   : BEFORE CONNECTION!
g.s.data.druiddemo.ConnectionLogFilter   : AFTER CONNECTION!
```

内容提要

- 什么是数据访问层
- JDBC数据连接
- 数据连接池
- 通过Spring JDBC访问数据库
- 事务管理

Spring 的 JDBC 操作类

- spring-boot-starter-jdbc
- core，JdbcTemplate 等相关核心接口和类
- datasource，数据源相关的辅助类
- object，将基本的 JDBC 操作封装成对象
- support，错误码等其他辅助工工具

常用的 Bean 注解

- 通过注解定义 Bean
- @Component
- @Repository
- @Service
- @Controller
  - @RestController

简单的 JDBC 操作

datasource-demo

- JdbcTemplate
  - query
  - queryForObject
  - queryForList
  - update
  - execute

内容提要

- 什么是数据访问层
- JDBC数据连接
- 数据连接池
- 通过Spring JDBC访问数据库
- 事务管理

Spring 的事务抽象

一致的事务模型

- JDBC/Hibernate/myBatis
- DataSource/JTA

事务抽象的核心接口

PlatformTransactionManager

- DataSourceTransactionManager
- HibernateTransactionManager
- JtaTransactionManager

TransactionDefinition

- Propagation
- Isolation
- Timeout
- Read-only status

```
void commit(TransactionStatus status) throws TransactionException;

void rollback(TransactionStatus status) throws TransactionException;

TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException;
```

事务传播特性

传播性	值	描述
PROPAGATION_REQUIRED	0	当前有事务就用当前的，没有就用新的
PROPAGATION_SUPPORTS	1	事务可有可无，不是必须的
PROPAGATION_MANDATORY	2	当前一定要有事务，不然就抛异常
PROPAGATION_REQUIRES_NEW	3	无论是否有事务，都起个新的事务
PROPAGATION_NOT_SUPPORTED	4	不支持事务，按非事务方式运行
PROPAGATION_NEVER	5	不支持事务，如果有事务则抛异常
PROPAGATION_NESTED	6	当前有事务就在当前事务里再起一个事务

事务隔离特性

隔离性	值	脏读	不可重复读	幻读
ISOLATION_READ_UNCOMMITTED	1	√	√	√
ISOLATION_READ_COMMITTED	2	×	√	√
ISOLATION_REPEATABLE_READ	3	×	×	√
ISOLATION_SERIALIZABLE	4	×	×	×

编程式事务

TransactionTemplate

- TransactionCallback
- TransactionCallbackWithoutResult

PlatformTransactionManager

- 可以传入TransactionDefinition进行定义

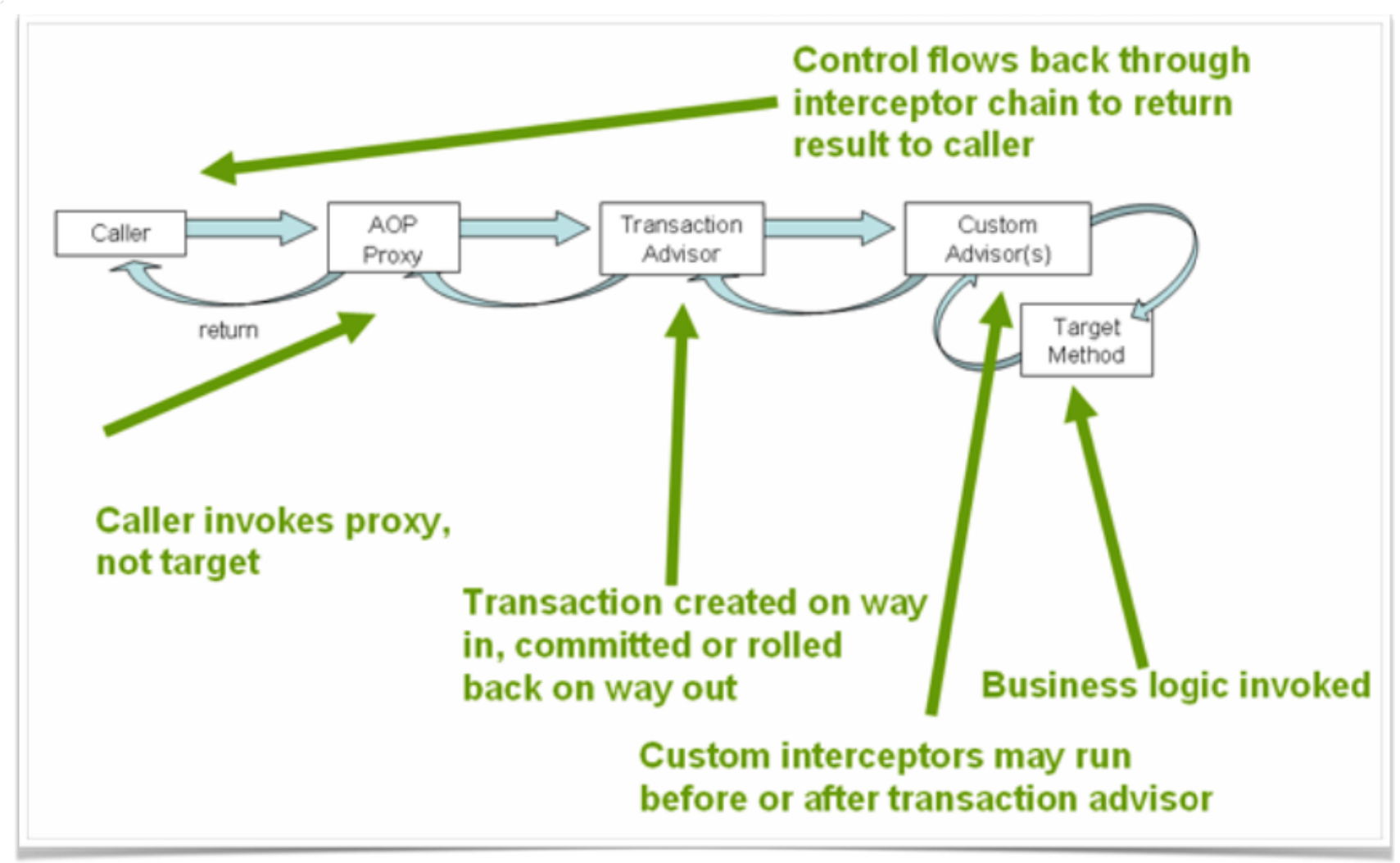
```java
@SpringBootApplication
@Slf4j
public class ProgrammaticTransactionDemoApplication implements CommandLineRunner {
	@Autowired
	private TransactionTemplate transactionTemplate;
	@Autowired
	private JdbcTemplate jdbcTemplate;

	public static void main(String[] args) {
		SpringApplication.run(ProgrammaticTransactionDemoApplication.class, args);
	}

	@Override
	public void run(String... args) throws Exception {
		log.info("COUNT BEFORE TRANSACTION: {}", getCount());
		transactionTemplate.execute(new TransactionCallbackWithoutResult() {
			@Override
			protected void doInTransactionWithoutResult(TransactionStatus transactionStatus) {
				jdbcTemplate.execute("INSERT INTO FOO (ID, BAR) VALUES (1, 'aaa')");
				log.info("COUNT IN TRANSACTION: {}", getCount());
				transactionStatus.setRollbackOnly();
			}
		});
		log.info("COUNT AFTER TRANSACTION: {}", getCount());
	}

	private long getCount() {
		return (long) jdbcTemplate.queryForList("SELECT COUNT(*) AS CNT FROM FOO")
				.get(0).get("CNT");
	}
}
```

声明式事务


Control flows back through interceptor chain to return result to caller
Caller
AOP Proxy
Transaction Advisor
Custom Advisor(s)
Target Method
return
Caller invokes proxy, not target
Transaction created on way in, committed or rolled back on way out
Business logic invoked
Custom interceptors may run before or after transaction advisor

基于注解的配置方式

开启事务注解的方式

- @EnableTransactionManagement
- <tx:annotation-driven/>

一些配置

- proxyTargetClass
- mode
- order

@Transactional

- transactionManager
- propagation
- isolation
- timeout
- readOnly
- 怎么判断回滚

declarative-transaction-demo

```
@Component
public class FooServiceImpl implements FooService {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    @Transactional
    public void insertRecord() {
        jdbcTemplate.execute("INSERT INTO FOO (BAR) VALUES ('AAA')");
    }

    @Override
    @Transactional(rollbackFor = RollbackException.class)
    public void insertThenRollback() throws RollbackException {
        jdbcTemplate.execute("INSERT INTO FOO (BAR) VALUES ('BBB')");
        throw new RollbackException();
    }

    @Override
    public void invokeInsertThenRollback() throws RollbackException {
        insertThenRollback();
    }
}
```