第二部分：Spring 中的数据操作

极客时间

扫码试看/订阅

《玩转 Spring 全家桶》

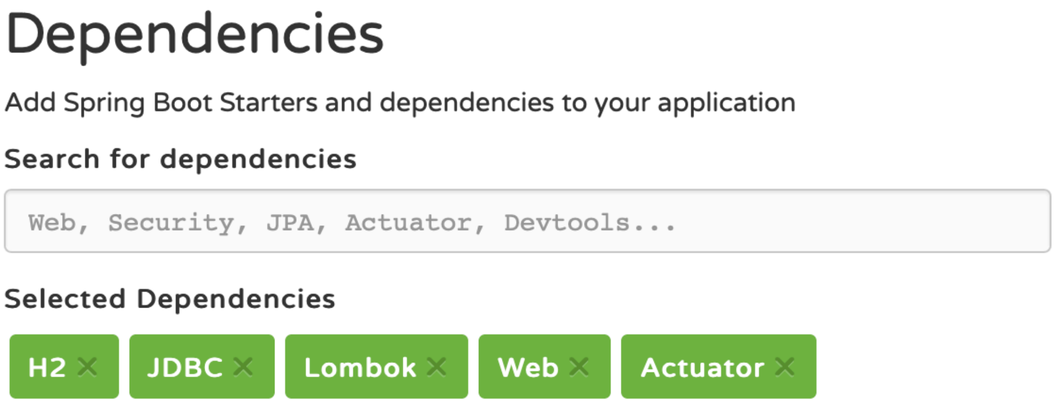
极客时间

JDBC 必知必会

如何配置数据源

Spring Boot 的配置演示

- 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

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）

配置多数据源的注意事项

不同数据源的配置要分开

关注每次使用的数据源

- 有多个DataSource时系统如何判断
- 对应的设施（事务、ORM等）如何选择DataSource

Spring Boot中的多数据源配置

手工配置两组 DataSource 及相关内容

与Spring Boot协同工作（二选一）

- 配置@Primary类型的Bean
- 排除Spring Boot的自动配置
  - DataSourceAutoConfiguration
  - DataSourceTransactionManagerAutoConfiguration
  - JdbcTemplateAutoConfiguration

```java
@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class,
        DataSourceTransactionManagerAutoConfiguration.class,
        JdbcTemplateAutoConfiguration.class})
@Slf4j
public class MultiDataSourceDemoApplication {
```

```java
@Bean
@ConfigurationProperties("foo.datasource")
public DataSourceProperties fooDataSourceProperties() {
    return new DataSourceProperties();
}

@Bean
public DataSource fooDataSource() {
    DataSourceProperties dataSourceProperties = fooDataSourceProperties();
    log.info("foo datasource: {}", dataSourceProperties.getUrl());
    return dataSourceProperties.initializeDataSourceBuilder().build();
}

@Bean
@Resource
public PlatformTransactionManager fooTxManager(DataSource fooDataSource) {
    return new DataSourceTransactionManager(fooDataSource);
}
```

```java
@Bean
@ConfigurationProperties("bar.datasource")
public DataSourceProperties barDataSourceProperties() {
    return new DataSourceProperties();
}

@Bean
public DataSource barDataSource() {
    DataSourceProperties dataSourceProperties = barDataSourceProperties();
    log.info("bar datasource: {}", dataSourceProperties.getUrl());
    return dataSourceProperties.initializeDataSourceBuilder().build();
}

@Bean
@Resource
public PlatformTransactionManager barTxManager(DataSource barDataSource) {
    return new DataSourceTransactionManager(barDataSource);
}
```

那些好用的连接池

HikariCP

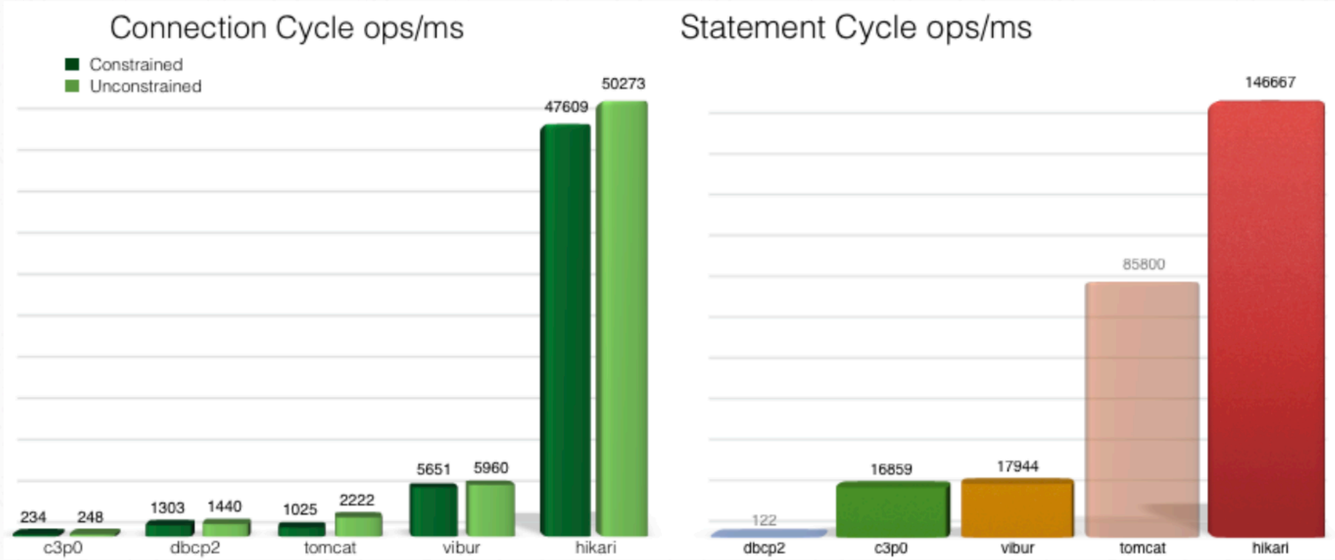
光 HikariCP

A high-performance JDBC connection pool.

// It's Faster.

There is nothing faster. There is nothing more correct. HikariCP is a "zero-overhead" production-quality connection pool.

Using a stub-JDBC implementation to isolate and measure the overhead of HikariCP, comparative benchmarks were performed on a commodity PC.



Just drop it in and let your code run like its pants are on fire.

HikariCP 为什么快

1. 字节码级别优化（很多方法通过 JavaAssist 生成）

2. 大量小改进

- 用 FastStatementList 代替 ArrayList
- 无锁集合 ConcurrentBag
- 代理类的优化（比如，用 invokestatic 代替了 invokevirtual）

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

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

那些好用的连接池

Alibaba Druid

“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。”

***–Alibaba Druid* 官方介绍**

Druid

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持
- SQL 防注入
- 内置加密配置
- 众多扩展点，方便进行定制

数据源配置

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

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 以便方便地实现 Filter
- 修改 META-INF/druid-filter.properties 增加 Filter 配置

Druid Filter

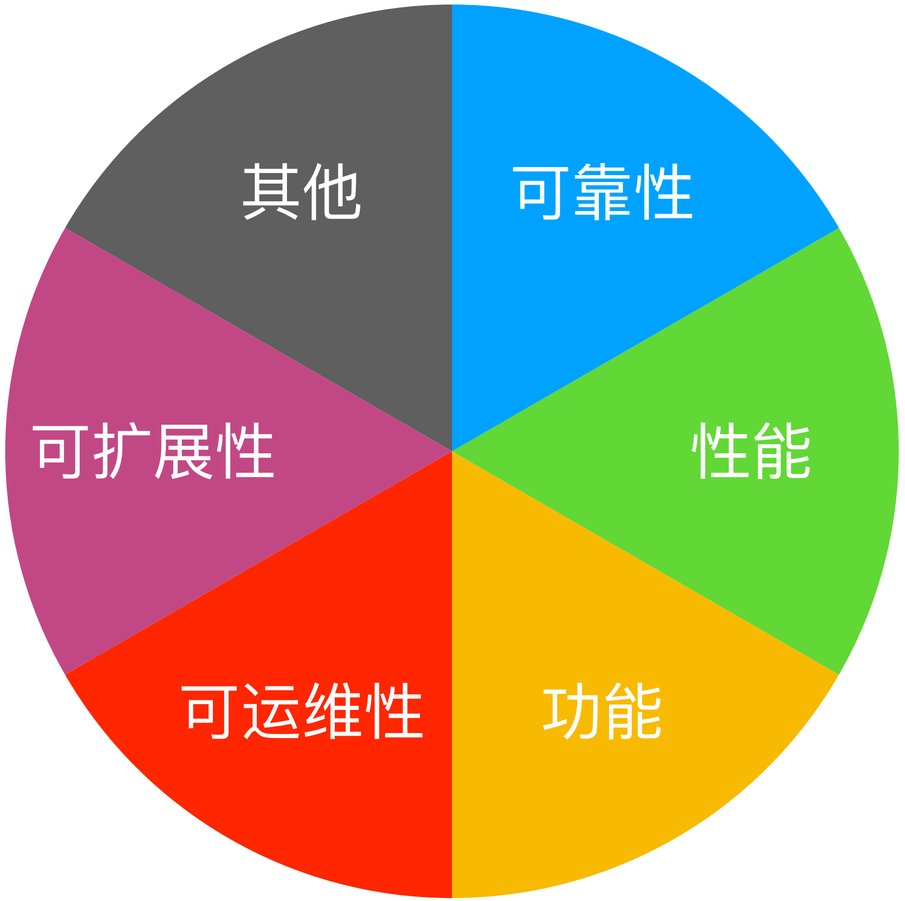
```java
@Slf4j
public class ConnectionLogFilter extends FilterEventAdapter {

    @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!
```

连接池选择时的考量点



通过 Spring JDBC 访问数据库

Spring 的 JDBC 操作类

spring-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

JdbcTemplate

- query
- queryForObject
- queryForList
- update
- execute

“Talk is cheap, show me the code.”

–Linus Torvalds

SQL 批处理

JdbcTemplate

- batchUpdate
  - BatchPreparedStatementSetter

NamedParameterJdbcTemplate

- batchUpdate
  - SqlParameterSourceUtils.createBatch

“Talk is cheap, show me the code.”

–Linus Torvalds

了解 Spring 的抽象

事务抽象

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

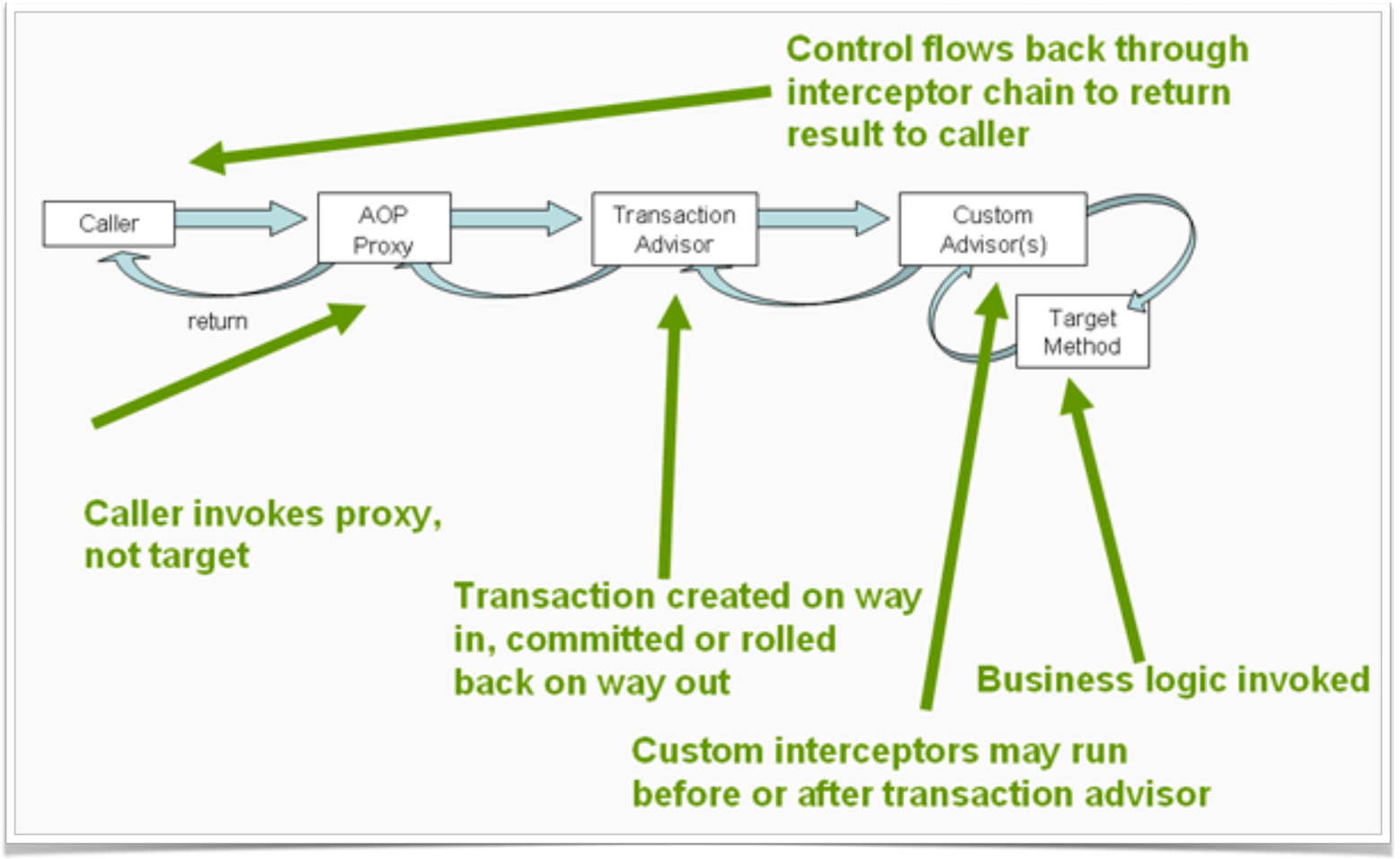
```java
@SpringBootApplication
@Slf4j
public class ProgrammaticTransactionDemoApplication implements CommandLineRunner {
        @Autowired
        private TransactionTemplate transactionTemplate;
        @Autowired
        private JdbcTemplate jdbcTemplate;

        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

极客时间

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

```java
@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();
    }
}
```

极客时间

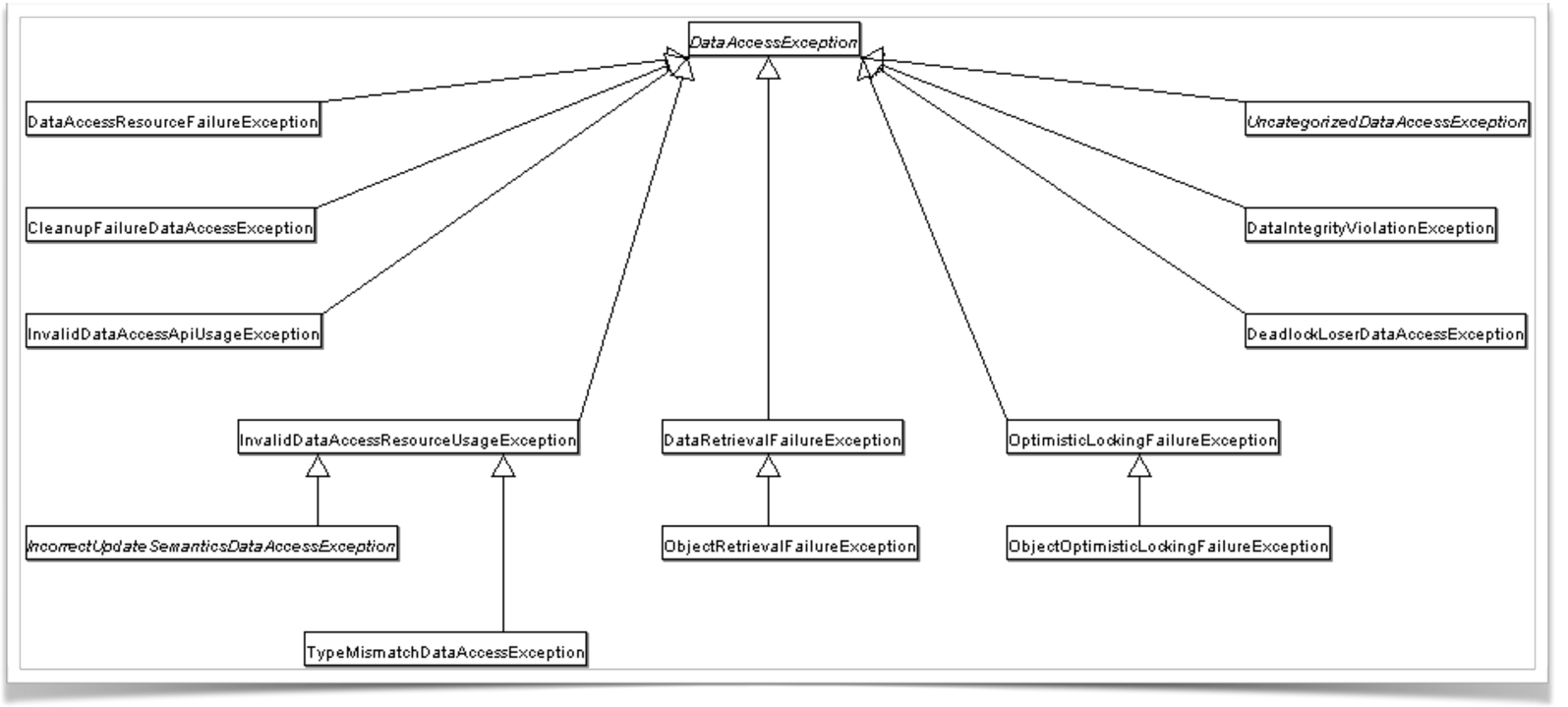
了解 Spring 的抽象

JDBC 异常抽象

Spring 的 JDBC 异常抽象

Spring 会将数据操作的异常转换为 DataAccessException

无论使用何种数据访问方式，都能使用一样的异常



Spring是怎么认识那些错误码的

通过 SQLErrorCodeSQLExceptionTranslator 解析错误码

ErrorCode 定义

- org/springframework/jdbc/support/sql-error-codes.xml
- Classpath 下的 sql-error-codes.xml

定制错误码解析逻辑

```
<bean id="H2" class="org.springframework.jdbc.support.SQLErrorCodes">
    <property name="badSqlGrammarCodes">
        <value>42000,42001,42101,42102,42111,42112,42121,42122,42132</value>
    </property>
    <property name="duplicateKeyCodes">
        <value>23001,23505</value>
    </property>
    <property name="dataIntegrityViolationCodes">
        <value>22001,22003,22012,22018,22025,23000,23002,23003,23502,23503,23506,23507,23513</value>
    </property>
    <property name="dataAccessResourceFailureCodes">
        <value>90046,90100,90117,90121,90126</value>
    </property>
    <property name="cannotAcquireLockCodes">
        <value>50200</value>
    </property>
    <property name="customTranslations">
        <bean class="org.springframework.jdbc.support.CustomSQLErrorCodesTranslation">
            <property name="errorCodes" value="23001,23505" />
            <property name="exceptionClass"
                      value="geektime.spring.data.errorcodedemo.CustomDuplicatedKeyException" />
        </bean>
    </property>
</bean>
```

极客时间

扫码试看/订阅

《玩转 Spring 全家桶》