

二

服务器出问题，目前部分恢复

07 数据分片：如何实现分库、分表、分库+分表以及强制路由？（下）

在上一课时中，我们基于业务场景介绍了如何将单库单表架构改造成分库架构。今天我们继续后续的改造工作，主要涉及如何实现分表、分库+分表以及如何实现强制路由。

系统改造：如何实现分表？

相比分库，分表操作是在同一个数据库中，完成对一张表的拆分工作。所以从数据源上讲，我们只需要定义一个 DataSource 对象即可，这里把这个新的 DataSource 命名为 ds2：

```
spring.shardingsphere.datasource.names=ds2
spring.shardingsphere.datasource.ds2.type=com.alibaba.druid.pool.DruidDataSou
spring.shardingsphere.datasource.ds2.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds2.url=jdbc:mysql://localhost:3306/ds2
spring.shardingsphere.datasource.ds2.username=root
spring.shardingsphere.datasource.ds2.password=root
```

同样，为了提高访问性能，我们设置了绑定表和广播表：

```
spring.shardingsphere.sharding.binding-tables=health_record, health_task
spring.shardingsphere.sharding.broadcast-tables=health_level
```

现在，让我们再次回想起 TableRuleConfiguration 配置，该配置中的 tableShardingStrategyConfig 代表分表策略。与用于分库策略的 databaseShardingStrategyConfig 一样，设置分表策略的方式也是指定一个用于分表的分片键以及分片表达式：

```
spring.shardingsphere.sharding.tables.health_record.table-strategy.inline.sha
spring.shardingsphere.sharding.tables.health_record.table-strategy.inline.alg
```

在代码中可以看到，对于 health_record 表而言，我们设置它用于分表的分片键为 record_id，以及它的分片行表达式为 health_record\$->{record_id % 2}。也就是说，我们会根据 record_id 将 health_record 单表拆分成 health_record0 和 health_record1 这两张分表。基于分表策略，再加上 actualDataNodes 和 keyGeneratorConfig 配置项，我们就可以完成对 health_record 表的完整分表配置：

```
spring.shardingsphere.sharding.tables.health_record.actual-data-nodes=ds2,hea
spring.shardingsphere.sharding.tables.health_record.table-strategy.inline.sha
spring.shardingsphere.sharding.tables.health_record.table-strategy.inline.alg
spring.shardingsphere.sharding.tables.health_record.key-generator.column=reco
spring.shardingsphere.sharding.tables.health_record.key-generator.type=SNOWFL
spring.shardingsphere.sharding.tables.health_record.key-generator.props.worke
```

对于 health task 表而言，可以采用同样的配置方法完成分表操作：

```
spring.shardingsphere.sharding.tables.health_task.actual-data-nodes=ds2.health_task
spring.shardingsphere.sharding.tables.health_task.table-strategy.inline.shard
spring.shardingsphere.sharding.tables.health_task.table-strategy.inline.algor
spring.shardingsphere.sharding.tables.health_task.key-generator.column=task_i
spring.shardingsphere.sharding.tables.health_task.key-generator.type=SNOWFLAK
spring.shardingsphere.sharding.tables.health_task.key-generator.props.worker
```

可以看到，由于 health_task 与 health_record 互为绑定表，所以在 health_task 的配置中，我们同样基于 record_id 列进行分片，也就是说，我们会根据 record_id 将 health_task 单表拆分成 health_task0 和 health_task1 两张分表。当然，自增键的生成列还是需要设置成 health_task 表中的 task_id 字段。

这样，完整的分表配置就完成了。现在，让我们重新执行 HealthRecordTest 单元测试，会发现数据已经进行了正确的分表。下图是分表之后的 health_record0 和 health_record1 表：

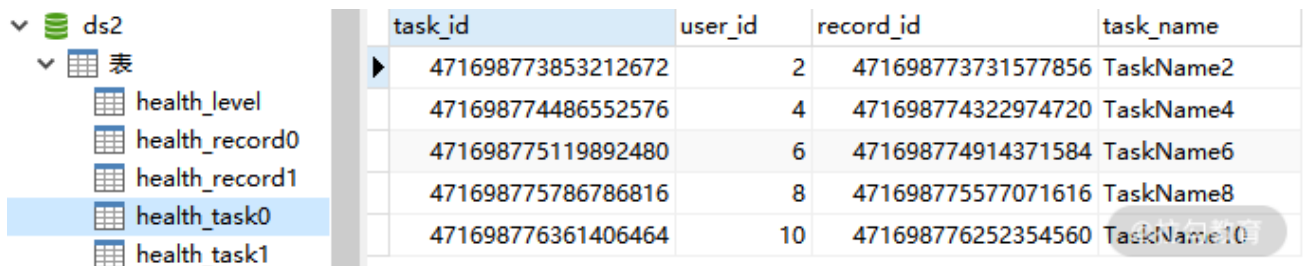
record_id	user_id	level_id	remark
471698773731577856	2	2	Remark2
471698774322974720	4	4	Remark4
471698774914371584	6	1	Remark6
471698775577071616	8	3	Remark8
471698776252354560	10	0	Remark10

分表后的 health_record0 表数据

ds2	record_id	user_id	level_id	remark
表	471698772846579713	1	1	Remark1
health_level	471698774025179137	3	3	Remark3
health_record0	471698774578827265	5	0	Remark5
health_record1	471698775287664641	7	2	Remark7
health_task0	471698775954558977	9	4	Remark9
health task1				

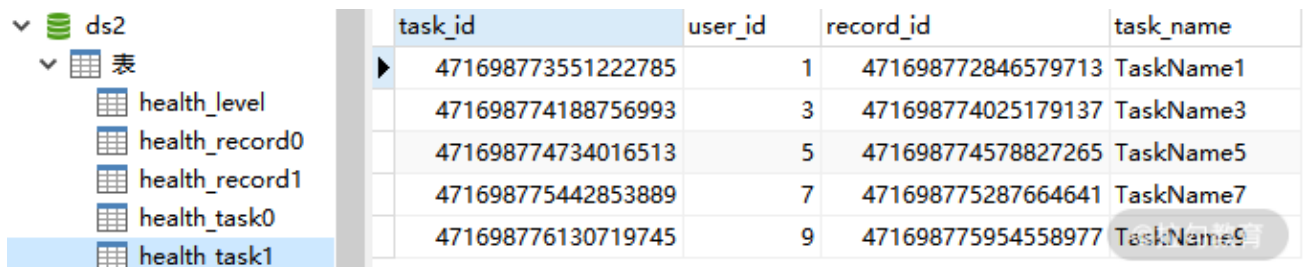
分表后的 health_record1 表数据

而这是分表之后的 health task0 和 health task1 表:



task_id	user_id	record_id	task_name
471698773853212672	2	471698773731577856	TaskName2
471698774486552576	4	471698774322974720	TaskName4
471698775119892480	6	471698774914371584	TaskName6
471698775786786816	8	471698775577071616	TaskName8
471698776361406464	10	471698776252354560	TaskName10

分表后的 health_task0 表数据



task_id	user_id	record_id	task_name
471698773551222785	1	471698772846579713	TaskName1
471698774188756993	3	471698774025179137	TaskName3
471698774734016513	5	471698774578827265	TaskName5
471698775442853889	7	471698775287664641	TaskName7
471698776130719745	9	471698775954558977	TaskName9

分表后的 health_task1表数据

系统改造：如何实现分库+分表？

在完成独立的分库和分表操作之后，系统改造的第三步是尝试把分库和分表结合起来。这个过程听起来比较复杂，但事实上，基于 ShardingSphere 提供的强大配置体系，开发人员要做的只是将分表针对分库和分表的配置项整合在一起就可以了。这里我们重新创建 3 个新的数据源，分别为 ds3、ds4 和 ds5：

```
spring.shardingsphere.datasource.names=ds3,ds4,ds5
spring.shardingsphere.datasource.ds3.type=com.alibaba.druid.pool.DruidDataSou
spring.shardingsphere.datasource.ds3.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds3.url=jdbc:mysql://localhost:3306/ds3
spring.shardingsphere.datasource.ds3.username=root
spring.shardingsphere.datasource.ds3.password=root
spring.shardingsphere.datasource.ds4.type=com.alibaba.druid.pool.DruidDataSou
spring.shardingsphere.datasource.ds4.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds4.url=jdbc:mysql://localhost:3306/ds4
spring.shardingsphere.datasource.ds4.username=root
spring.shardingsphere.datasource.ds4.password=root
spring.shardingsphere.datasource.ds5.type=com.alibaba.druid.pool.DruidDataSou
spring.shardingsphere.datasource.ds5.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds5.url=jdbc:mysql://localhost:3306/ds5
spring.shardingsphere.datasource.ds5.username=root
spring.shardingsphere.datasource.ds5.password=root
```

注意，到现在有 3 个数据源，而且命名分别是 ds3、ds4 和 ds5。所以，为了根据 user_id 来将数据分别分片到对应的数据源，我们需要调整行表达式，这时候的行表达式应该是 ds\${user_id % 3 + 3}：

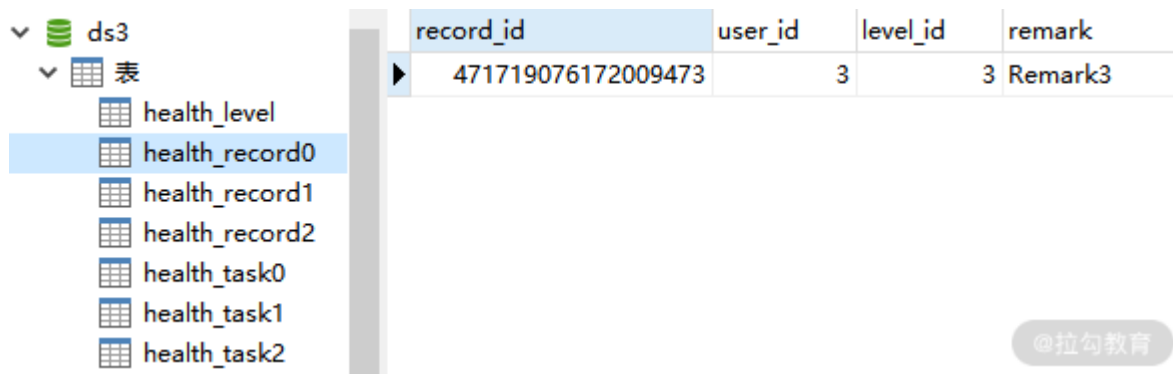
```
spring.shardingsphere.sharding.default-database-strategy.inline.sharding-colu
spring.shardingsphere.sharding.default-database-strategy.inline.algorithm-exp
```

```
spring.shardingsphere.sharding.binding-tables=health_record,health_task
spring.shardingsphere.sharding.broadcast-tables=health_level
```

对于 health_record 和 health_task 表而言，同样需要调整对应的行表达式，我们将 actual-data-nodes 设置为 ds\(->\{3..5\}.health_record\)->\{0..2\}，也就是说每张原始表将被拆分成 3 张分表：

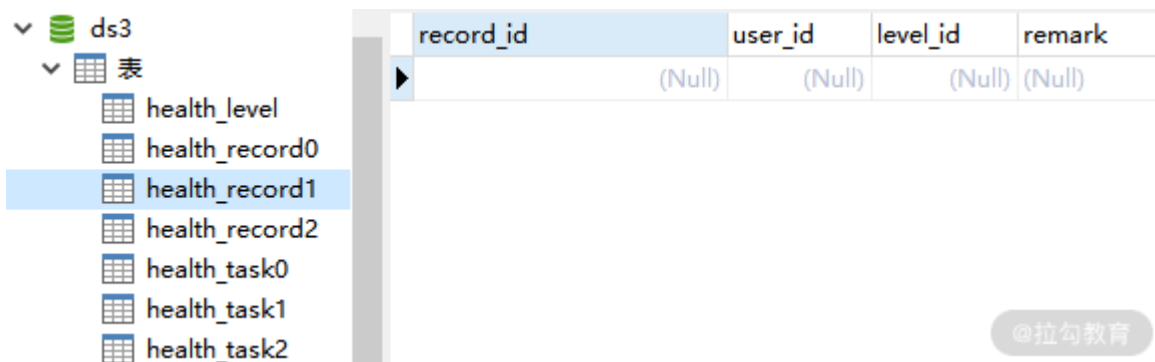
```
spring.shardingsphere.sharding.tables.health_record.actual-data-nodes=ds$->\{3
spring.shardingsphere.sharding.tables.health_record.table-strategy.inline.sha
spring.shardingsphere.sharding.tables.health_record.table-strategy.inline.alg
spring.shardingsphere.sharding.tables.health_record.key-generator.column=reco
spring.shardingsphere.sharding.tables.health_record.key-generator.type=SNOWFL
spring.shardingsphere.sharding.tables.health_record.key-generator.props.worke
spring.shardingsphere.sharding.tables.health_task.actual-data-nodes=ds$->\{3..
spring.shardingsphere.sharding.tables.health_task.table-strategy.inline.shard
spring.shardingsphere.sharding.tables.health_task.table-strategy.inline.algor
spring.shardingsphere.sharding.tables.health_task.key-generator.column=task_i
spring.shardingsphere.sharding.tables.health_task.key-generator.type=SNOWFLAK
spring.shardingsphere.sharding.tables.health_task.key-generator.props.worker.
```

这样，整合分库+分表的配置方案就介绍完毕了，可以看到，这里并没有引入任何新的配置项让我们重新执行单元测试，从而确认数据是否已经正确地进行分库分表。这是 ds3 中的 health_record0、health_record1 和 health_record2 表：



record_id	user_id	level_id	remark
471719076172009473	3	3	Remark3

ds3 中的 health_record0 表数据



record_id	user_id	level_id	remark
(Null)	(Null)	(Null)	(Null)

ds3 中的 health_record1 表数据

ds3

表

health_level

health_record0

health_record1

health_record2

health_task0

health_task1

health_task2

record_id	user_id	level_id	remark
471719077166059520	6	1	Remark6
471719078130749441	9	4	Remark9

@拉勾教育

ds3 中的 health_record2 表数据

这是 ds4 中的 health_record0、health_record1 和 health_record2 表：

ds4

表

health_level

health_record0

health_record1

health_record2

health_task0

health_task1

health_task2

record_id	user_id	level_id	remark
471719076578856960	4	4	Remark4

@拉勾教育

ds4 中的 health_record0 表数据

ds4

表

health_level

health_record0

health_record1

health_record2

health_task0

health_task1

health_task2

record_id	user_id	level_id	remark
471719075219902465	1	1	Remark1
471719077509992449	7	2	Remark7
471719078420156416	10	0	Remark10

@拉勾教育

ds4 中的 health_record1 表数据

ds4

表

health_level

health_record0

health_record1

health_record2

health_task0

health_task1

health_task2

record_id	user_id	level_id	remark
(Null)	(Null)	(Null)	(Null)

@拉勾教育

ds4 中的 health_record2 表数据

而下面是 ds5 中的 health_record0、health_record1 和 health_record2 表：

ds5

表

health_level

health_record0

health_record1

health_record2

health_task0

health_task1

health_task2

record_id	user_id	level_id	remark
(Null)	(Null)	(Null)	(Null)

@拉勾教育

ds5 中的 health_record0 表数据

ds5

表

health_level

health_record0

health_record1

health_record2

health_task0

health_task1

health_task2

record_id	user_id	level_id	remark
(Null)	(Null)	(Null)	(Null)

@拉勾教育

ds5 中的 health_record1 表数据

ds5

表

health_level

health_record0

health_record1

health_record2

health_task0

health_task1

health_task2

record_id	user_id	level_id	remark
471719075870019584	2	2	Remark2
471719076872458241	5	0	Remark5
471719077845536768	8	3	Remark8

@拉勾教育

ds5 中的 health_record2 表数据

对于 health_task 表而言，我们得到的也是类似的分库分表效果。

系统改造：如何实现强制路由？

从 SQL 执行效果而言，分库分表可以看作是一种路由机制，也就是说把 SQL 语句路由到目标数据库或数据表中并获取数据。在实现了分库分表的基础之上，我们将要引入一种不同的路由方法，即强制路由。

什么是强制路由？

强制路由与一般的分库分表路由不同，它并没有使用任何的分片键和分片策略。我们知道通过解析 SQL 语句提取分片键，并设置分片策略进行分片是 ShardingSphere 对重写 JDBC 规范的实现方式。但是，如果我们没有分片键，是否就只能访问所有的数据库和数据表进行全路由呢？显然，这种处理方式也不大合适。有时候，我们需要为 SQL 执行开一个“后门”，

允许在没有分片键的情况下，同样可以在外部设置目标数据库和表，这就是强制路由的设计理念。

在 ShardingSphere 中，通过 Hint 机制实现强制路由。我们在这里对 Hint 这一概念再做进一步的阐述。在关系型数据库中，Hint 作为一种 SQL 补充语法扮演着非常重要的角色。它允许用户通过相关的语法影响 SQL 的执行方式，改变 SQL 的执行计划，从而对 SQL 进行特殊的优化。很多数据库工具也提供了特殊的 Hint 语法。以 MySQL 为例，比较典型的 Hint 使用方式之一就是对所有索引的强制执行和忽略机制。

MySQL 中的强制索引能够确保所需要执行的 SQL 语句只作用于所指定的索引上，我们可以通过 FORCE INDEX 这一 Hint 语法实现这一目标：

```
SELECT * FROM TABLE1 FORCE INDEX (FIELD1)
```

类似的，IGNORE INDEX 这一 Hint 语法使得原本设置在具体字段上的索引不被使用：

```
SELECT * FROM TABLE1 IGNORE INDEX (FIELD1, FIELD2)
```

对于分片字段非 SQL 决定、而由其他外置条件决定的场景，可使用 SQL Hint 灵活地注入分片字段。

如何设计和开发强制路由？

基于 Hint 进行强制路由的设计和开发过程需要遵循一定的约定，同时，ShardingSphere 也提供了专门的 HintManager 来简化强制路由的开发过程。

- HintManager

HintManager 类的使用方式比较固化，我们可以通过查看源码中的类定义以及核心变量来理解它所包含的操作内容：

```
public final class HintManager implements AutoCloseable {  
    //基于ThreadLocal存储HintManager实例  
    private static final ThreadLocal<HintManager> HINT_MANAGER HOLDER = new T  
    //数据库分片值  
    private final Multimap<String, Comparable<?>> databaseShardingValues = Ha  
    //数据表分片值  
    private final Multimap<String, Comparable<?>> tableShardingValues = HashM  
    //是否只有数据库分片  
    private boolean databaseShardingOnly;  
    //是否只路由主库  
    private boolean masterRouteOnly;  
    ...  
}
```

在变量定义上，我们注意到 `HintManager` 使用了 `ThreadLocal` 来保存 `HintManager` 实例。显然，基于这种处理方式，所有分片信息的作用范围就是当前线程。我们也看到了用于分别存储数据库分片值和数据表分片值的两个 `Multimap` 对象，以及分别用于指定是否只有数据库分片，以及是否只路由主库的标志位。可以想象，`HintManager` 基于这些变量开放了一组 `get/set` 方法供开发人员根据具体业务场景进行分片键的设置。

同时，在类的定义上，我们也注意到 `HintManager` 实现了 `AutoCloseable` 接口，这个接口是在 JDK7 中引入的一个新接口，用于自动释放资源。`AutoCloseable` 接口只有一个 `close` 方法，我们可以实现这个方法来释放自定义的各种资源。

```
public interface AutoCloseable {  
    void close() throws Exception;  
}
```

在 JDK1.7 之前，我们需要手动通过 `try/catch/finally` 中的 `finally` 语句来释放资源，而使用 `AutoCloseable` 接口，在 `try` 语句结束的时候，不需要实现 `finally` 语句就会自动将这些资源关闭，JDK 会通过回调的方式，调用 `close` 方法来做到这一点。这种机制被称为 `try with resource`。`AutoCloseable` 还提供了语法糖，在 `try` 语句中可以同时使用多个实现这个接口的资源，并通过使用分号进行分隔。

`HintManager` 中通过实现 `AutoCloseable` 接口支持资源的自动释放，事实上，JDBC 中的 `Connection` 和 `Statement` 接口的实现类同样也实现了这个 `AutoCloseable` 接口。

对于 `HintManager` 而言，所谓的资源实际上就是 `ThreadLocal` 中所保存的 `HintManager` 实例。下面这段代码实现了 `AutoCloseable` 接口的 `close` 方法，进行资源的释放：

```
public static void clear() {  
    HINT_MANAGER_HOLDER.remove();  
}  
  
@Override  
public void close() {  
    HintManager.clear();  
}
```

`HintManager` 的创建过程使用了典型的单例设计模式，下面这段代码展现了通过一个静态的 `getInstance` 方法，从 `ThreadLocal` 中获取或设置针对当前线程的 `HintManager` 实例。

```
public static HintManager getInstance() {  
    Preconditions.checkNotNull(null == HINT_MANAGER_HOLDER.get(), "Hint has pre  
    HintManager result = new HintManager();  
    HINT_MANAGER_HOLDER.set(result);  
    return result;  
}
```


在理解了 HintManager 的基本结构之后，在应用程序中获取 HintManager 的过程就显得非常简单了，这里给出推荐的使用方式：

```
try (HintManager hintManager = HintManager.getInstance();
    Connection connection = dataSource.getConnection();
    Statement statement = connection.createStatement()) {
    ...
}
```

可以看到，我们在 try 语句中获取了 HintManager、Connection 和 Statement 实例，然后就可以基于这些实例来完成具体的 SQL 执行。

- 实现并配置强制路由分片算法

开发基于 Hint 的强制路由的基础还是配置。在介绍与 Hint 相关的配置项之前，让我们回想在 05 课时：“ShardingSphere 中的配置体系是如何设计的？”中介绍的 TableRuleConfiguration。我们知道 TableRuleConfiguration 中包含两个 ShardingStrategyConfiguration，分别用于设置分库策略和分表策略。而 ShardingSphere 专门提供了 HintShardingStrategyConfiguration 用于完成 Hint 的分片策略配置，如下面这段代码所示：

```
public final class HintShardingStrategyConfiguration implements ShardingStrat
    private final HintShardingAlgorithm shardingAlgorithm;

    public HintShardingStrategyConfiguration(final HintShardingAlgorithm shar
        Preconditions.checkNotNull(shardingAlgorithm, "ShardingAlgorithm is r
        this.shardingAlgorithm = shardingAlgorithm;
    }
}
```

可以看到，HintShardingStrategyConfiguration 中需要设置一个 HintShardingAlgorithm。HintShardingAlgorithm 是一个接口，我们需要提供它的实现类来根据 Hint 信息执行分片。

```
public interface HintShardingAlgorithm<T> extends Comparable<?>> extends Shard
    //根据Hint信息执行分片
    Collection<String> doSharding(Collection<String> availableTargetNames, Hi
}
```

在 ShardingSphere 中内置了一个 HintShardingAlgorithm 的实现类 DefaultHintShardingAlgorithm，但这个实现类并没有执行任何的分片逻辑，只是将传入的所有 availableTargetNames 直接进行返回而已，如下面这段代码所示：

```
public final class DefaultHintShardingAlgorithm implements HintShardingAlgori
    @Override
```

```

    public Collection<String> doSharding(final Collection<String> availableTa
        return availableTargetNames;
    }
}

```

我们可以根据需要提供自己的 HintShardingAlgorithm 实现类并集成到 HintShardingStrategyConfiguration 中。例如，我们可以对比所有可用的分库分表键值，然后与传入的强制分片键进行精准匹配，从而确定目标的库表信息：

```

public final class MatchHintShardingAlgorithm implements HintShardingAlgorith

@Override
public Collection<String> doSharding(final Collection<String> availableTa
    Collection<String> result = new ArrayList<>();
    for (String each : availableTargetNames) {
        for (Long value : shardingValue.getValues()) {
            if (each.endsWith(String.valueOf(value))) {
                result.add(each);
            }
        }
    }
    return result;
}
}

```

一旦提供了自定的 HintShardingAlgorithm 实现类，就需要将它添加到配置体系中。在这里，我们基于 Yaml 配置风格来完成这一操作：

```

defaultDatabaseStrategy:
  hint:
    algorithmClassName: com.tianyanlan.shardingsphere.demo.hint.MatchHints

```

ShardingSphere 在进行路由时，如果发现 TableRuleConfiguration 中设置了 Hint 的分片算法，就会从 HintManager 中获取分片值并进行路由操作。

如何基于强制路由访问目标库表？

在理解了强制路由的概念和开发过程之后，让我们回到案例。这里以针对数据库的强制路由为例，给出具体的实现过程。为了更好地组织代码结构，我们先来构建两个 Helper 类，一个是用于获取 DataSource 的 DataSourceHelper。在这个 Helper 类中，我们通过加载 .yaml 配置文件来完成 DataSource 的构建：

```

public class DataSourceHelper {
    static DataSource getDataSourceForShardingDatabases() throws IOException,
        return YamlShardingDataSourceFactory.createDataSource(getFile("/META-I
    }
}

```

```
private static File getFile(final String configFile) {
    return new File(Thread.currentThread().getClass().getResource(configF
}
}
```

这里用到了 YamlShardingDataSourceFactory 工厂类，针对 Yaml 配置的实现方案你可以回顾 05 课时中的内容。

另一个 Helper 类是包装 HintManager 的 HintManagerHelper。在这个帮助类中，我们通过使用 HintManager 开放的 setDatabaseShardingValue 来完成数据库分片值的设置。在这个示例中，我们只想从第一个库中获取目标数据。HintManager 还提供了 addDatabaseShardingValue 和 addTableShardingValue 等方法设置强制路由的分片值。

```
public class HintManagerHelper {
    static void initializeHintManagerForShardingDatabases(final HintManager h
        hintManager.setDatabaseShardingValue(1L);
    }
}
```

最后，我们构建一个 HintService 来完成整个强制路由流程的封装：

```
public class HintService {
    private static void processWithHintValueForShardingDatabases() throws SQL
        DataSource dataSource = DataSourceHelper.getDataSourceForShardingDatab
        try (HintManager hintManager = HintManager.getInstance();
            Connection connection = dataSource.getConnection();
            Statement statement = connection.createStatement()) {
        HintManagerHelper.initializeHintManagerForShardingDatabases(hintMa
        ResultSet result = statement.executeQuery("select * from health_re

        while (result.next()) {
            System.out.println(result.getInt(0) + result.getString(1));
        }
    }
}
```

可以看到，在这个 processWithHintValueForShardingDatabases 方法中，我们首先通过 DataSourceHelper 获取目标 DataSource。然后使用 try with resource 机制在 try 语句中获取了 HintManager、Connection 和 Statement 实例，并通过 HintManagerHelper 帮助类设置强制路由的分片值。最后，通过 Statement 来执行一个全表查询，并打印查询结果：

```
2020-05-25 21:58:13.932 INFO 20024 --- [          main] ShardingSphere-SQL
...
2020-05-25 21:58:13.932 INFO 20024 --- [          main] ShardingSphere-SQL
6: user_6
7: user_7
8: user_8
```

```
9: user_9  
10: user_10
```

我们获取执行过程中的日志信息，可以看到原始的逻辑 SQL 是 `select user_id, user_name from user`，而真正执行的真实 SQL 则是 `ds1 ::: select user_id, user_name from user`。显然，强制路由发生了效果，我们获取的只是 ds1 中的所有 User 信息。

小结

承接上一课时的内容，今天我们继续在对单库单表架构进行分库操作的基础上，讲解如何实现分表、分库+分表以及强制路由的具体细节。有了分库的实践经验，要完成分表以及分库分表是比较容易的，所做的工作只是调整和设置对应的配置项。而强制路由是一种新的路由机制，我们通过较大的篇幅来对它的概念和实现方法进行了展开，并结合业务场景给出了案例分析。

这里给你留一道思考题：ShardingSphere 如何基于 Hint 机制实现分库分表场景下的强制路由？

从路由的角度讲，基于数据库主从架构的读写分离机制也可以被认为是一种路由。在下一课时的内容中，我们将对 ShardingSphere 提供的读写分离机制进行讲解，并同样给出读写分离与分库分表、强制路由进行整合的具体方法。

[上一页](#)

[下一页](#)