nodejs操作mysql数据库

以前的前端使用的是php来做为后台语言，但是现在的前端已经改用nodejs去开发了，我们之前又学习了`mysql`的数据库，所以现在我们要将nodejs与mysql进行结合

nodejs可以借助第三方模块去操作mysql数据库，这个最基本的模块就是`mysql`

创建项目

在创建项目的时候，我们要进行初始化操作

```
$ npm init --yes
```

安装所需要的依赖包

我们目前所使用的包是`mysql`这个包，在使用之前最好还是到`npm`仓库去查询一下这个包的相关信息

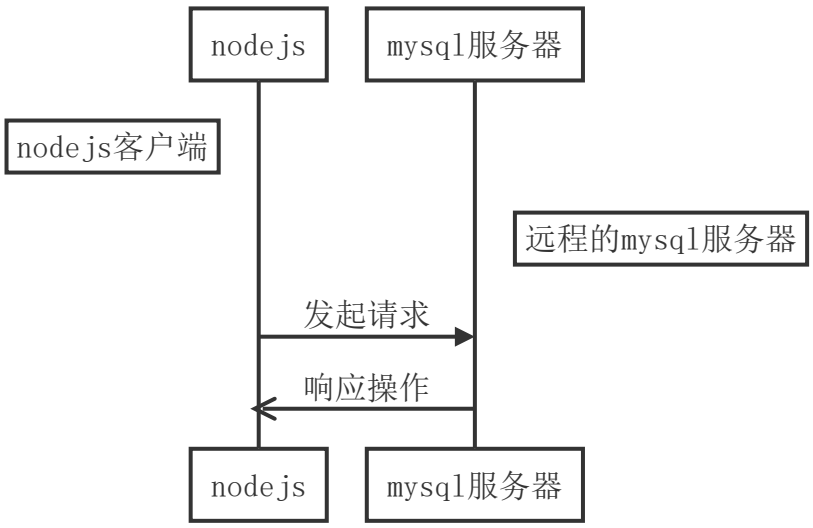
```
$ npm install mysql --save
```

操作mysql数据库

在nodejs平台下面使用`mysql`的包去操作数据库我还大体还是分为四个过程，也就是数据库的增删改查操作

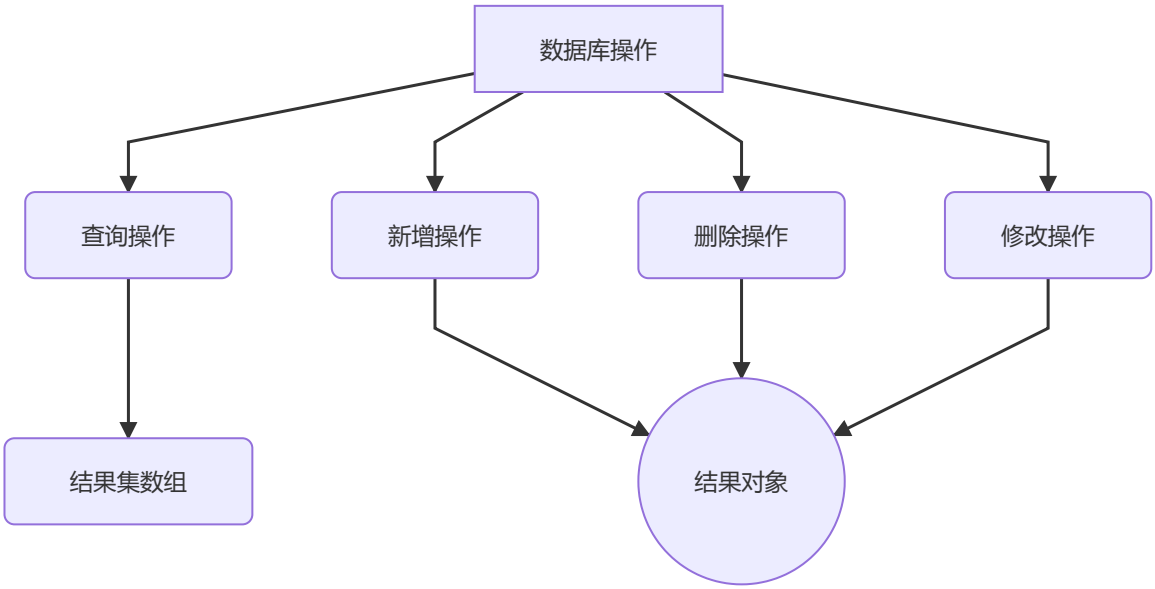
连接mysql数据库

在做任何操作之前，我们都需要先连接数据库，所以可以按照下面方式来进行



```
const mysql = require("mysql");
//第一步：先连接mysql的数据库
let conn = mysql.createConnection({
    host: "192.168.1.254",
    port: 3306,
    user: "h2003",
    password: "123456",
    database: "test"
});
//开始发请连接请求
conn.connect();
```

在连接数据库的时候，我们要配置数据库的相关信息，这些信息都是我们以前连接数据库的时候必备的信息



数据库的查询操作

```
//第二步：查询信息，准备SQL语句
let strSql = " select * from stu_info ";
//第三步：执行上面的SQL语句，我们要使用我们之前所创建的连接去执行SQL语句
conn.query(strSql, (err, results) => {
    //这里的回调代表数据库的执行结果，如果err不为空，则SQL语句执行失败，如果err为空，则SQL语句执行成功
    if (err) {
        //代表SQL语句执行失败
        console.log("SQL语句执行失败");
        console.log(err);
    }
    else {
        console.log("SQL语句执行成功");
        console.log(results);        //打印查询的结果
    }
    //这里一定一定要记得，关闭数据库的连接，释放资源
    conn.end();
})
```

经过上面的操作，我们已经执行了查询的SQL语句

在conn的连接里面，我们使用`query`方法去查询，它有一个回调函数，这个回调函数里同有2个参数，第一个是`err`代表SQL语句执行失败以后的错误信息，第二个`results`代表执行成功以后的回调函数

注意：每条SQL语句执行完以后，都要关闭数据库连接，释放资源

分析查询操作的结果

```
[
  RowDataPacket {
    id: 1,
    stu_name: '邓娜',
    stu_sex: ' 女',
    stu_age: 18,
    stu_address: '湖北赤壁'
  },
  RowDataPacket {
    id: 2,
    stu_name: ' 蒋雨晴',
    stu_sex: ' 女',
    stu_age: 19,
    stu_address: ' 湖北汉川'
  }
  //......省略
]
```

上面就是我们经过查询以后所得到的结果，这个时候我们发现`results`它是一个数组，数组里面的每一项都是一个对象，**列表就是我们的属性名，数据表当中每一行的值就是属性值**

带条件的查询操作

我们上面的查询语句是没有查询条件的，如果我在执行查询SQL语句的时候，里面有查询条件呢？

如：

1. 查询所有性别为女的学生
2. 查询年龄大于19岁的学生
3. 查询年龄介于19~20岁的学生

这些都是查询条件，对于这种查询条件我们应该怎么处理呢？

第一种方式：使用字符串拼接的方式

```
let stu_sex = "女"
let strSql = " select * from stu_info where stu_sex = '" + stu_sex + "' ";
conn.query(strSql, (err, results) => {
    if (err) {
        console.log(err);
    }
    else {
        console.log(results);
    }
    conn.end();
})
```

在上面的代码里面，我们已经可以使用字符串的拼接来完成我们的SQL语句的拼接。但是在做字符串拼接的时候，我们可以使用模板字符串去完成，如下

```
let strSql = ` select * from stu_info where stu_sex = '${stu_sex}' `;
```

第二种方式：使用SQL参数进行

```
//第二步：准备SQL语句
let stu_sex = "男";
let strSql = " select * from stu_info where stu_sex = ? ";

//第三步：执行SQL语句，并且放入具体的值替换参数
conn.query(strSql, [stu_sex], (err, results) => {
    if (err) {
        console.log(err);
    }
    else {
        console.log(results);
    }
    conn.end();
});
```

在SQL语句里面，我们可以使用`?`来做为SQL语句当中的参数，在使用`query`执行这条SQL语句的时候，再去把这个参数放进去就可以了

数据库的新增操作

数据库除了查询以外，还有其它的操作，现在我们来看一下新增的操作

> **重点**：查询语句返回的是一个结果集，那么新增的时候返回的是一个什么呢？？？

```
//第二步：准备SQL语句
let insertSql = " insert into stu_info (stu_name,stu_sex,stu_age,stu_address)
values (?,?,?,?) ";
conn.query(insertSql, ["陈一铭", "男", 19, "湖北武汉"], (err, results) => {
    if (err) {
        console.log(err);
    }
    else {
        console.log(results);
    }
    conn.end();
});
```

分析新增操作的结果

当我们使用`mysql`的新增操作以后，返回的结果如下

```
OkPacket {
  fieldCount: 0,
  affectedRows: 1,
  insertId: 6,
  serverStatus: 2,
  warningCount: 0,
  message: '',
  protocol41: true,
  changedRows: 0
}
```

> - `affectedRows`代表的是mysql数据库受影响的行数
> - `insertId`代表的是如果这个数据表有自增长的id，这个就是返回的自增的长的id

根据这个操作的结果，我们后期在做项目的时候可以根据返回的`affectedRows`来判断我们的数据是否插入成功了

数据库的修改操作

数据库的修改操作与新增操作非常类似，都是返回受影响的行数，只是SQL语句不一样而已

```
//准备SQL语句
let updateSql = " update stu_info set stu_name = ? where id = ? ";
conn.query(updateSql, ["歪某人", 8], (err, results) => {
    if (err) {
        console.log(err);
    }
    else {
        console.log(results);
    }
    conn.end();
})
```

分析修改操作的结果

```
OkPacket {
  fieldCount: 0,
  affectedRows: 1,
  insertId: 0,
  serverStatus: 2,
  warningCount: 0,
  message: '(Rows matched: 1  Changed: 1  Warnings: 0',
  protocol41: true,
  changedRows: 1
}
```

在修改操作的过程当中，会多一个可用的属性`changeRows`代表改变的行数

数据库的删除操作

这个操作与之前的操作是一样的，返回的类型也是受影响的行数

```
//第二步：准备sql语句
let deleteSql = " delete from stu_info where id = ? ";
conn.query(deleteSql, [7], (err, results) => {
    if (err) {
        console.log(err);
    }
    else {
        console.log(results);
    }
    conn.end();
})
```

分析删除操作的结果

```
OkPacket {
  fieldCount: 0,
  affectedRows: 1,
  insertId: 0,
  serverStatus: 2,
  warningCount: 0,
  message: '',
  protocol41: true,
  changedRows: 0
}
```

删除操作的结果与之前的新增，修改保持一致，都是通过判断`affectedRows`来决定它是否成功了

---

nodejs对mysql操作的封装

通过上面的增删改查我们可以发现数据的操作有很多都是相同的，最理想的方法就应该一条SQL语句可以马上执行，然后再得到结果

所以我们要思考着将上面的过程进行封装

```
/**
 * @name DBUtils 数据库操作集合
 * @author 杨标
 * @version 1.0
 * @description 数据库操作的相关方法工具类
 * @requires mysql
 */

const mysql = require("mysql");

class DBUtils {
    /**
     * @name getConn 获取数据库连接
     * @returns {mysql.Connection} 返回数据库的连接
     */
    getConn() {
        let conn = mysql.createConnection({
            host: "192.168.1.254",
            port: 3306,
            user: "h2003",
            password: "123456",
```

```
            database: "test"
        });
        conn.connect();
        return conn;
    }
    /**
     * @name executeSql 执行SQL语句
     * @param {string} strSql 要执行的SQL语句
     * @param {Array} sqlParams 要执行的SQL语句里面的参数
     * @returns {Promise} 返回一个Promise的数据库执行结果
     */
    executeSql(strSql, sqlParams = []) {
        let p = new Promise((resolve, reject) => {
            let conn = this.getConn();
            conn.query(strSql, sqlParams, (err, results) => {
                if (err) {
                    //数据库执行失败
                    reject(err);
                }
                else {
                    //数据库执行成功
                    resolve(results);
                }
                conn.end();
            });
        });
        return p;
    }
}
module.exports = DBUtils;
```

在上面的`executeSql`这个方法的封装里面，我们使用了`Promise`进行异步处理，也使用了函数的参数默认值来完成，现在我们来验证一下上面的封装

无参数的SQL语句执行

```
const DBUtils = require("./DBUtils");
const queryStuInfoList = async () => {
    try {
        let db = new DBUtils();
        let strSql = " select * from stu_info ";
        //这个results就是resolve出来的results
        //await只能等待Promise,并且只能等到resolve的结果，必须与async结合使用
        let results = await db.executeSql(strSql);
        console.log("数据库执行成功");
        console.log(results);
    } catch (error) {
        //这里的结果就是之前reject的结果，也就是数据库执行失败以后的结果
        console.log("数据库执行失败");
        console.log(error);
    }
}
queryStuInfoList();
```

有参数的SQL语句执行

```
const DBUtils = require("./DBUtils");
async function queryStuInfoList() {
    try {
        let db = new DBUtils();
        let strSql = " select * from stu_info where stu_sex = ? ";
        let results = await db.executeSql(strSql, ["男"]);
        console.log(results)
    } catch (error) {
        console.log("数据库执行失败");
        console.log(error);
    }
}
queryStuInfoList();
```

以下是备选方式

```
/**
 * @name DBUtils 数据库操作集合
 * @author 杨标
 * @version 1.0
 * @description 数据库操作的相关方法工具类
 * @requires mysql
 */

const mysql = require("mysql");

class DBUtils2 {
    /**
     * @name getConn 获取数据库连接
     * @returns {mysql.Connection} 返回数据库的连接
     */
    getConn() {
        let conn = mysql.createConnection({
            host: "192.168.1.254",
            port: 3306,
            user: "h2003",
            password: "123456",
            database: "test"
        });
        conn.connect();
        return conn;
    }
    /**
     * @name executeSql 执行SQL语句
     * @param {string} strSql 要执行的SQL语句
     * @param {Array} sqlParams 要执行的SQL语句里面的参数
     * @returns {Promise} 返回一个Promise的数据库执行结果
     */
    executeSql(strSql, ...sqlParams) {
        let p = new Promise((resolve, reject) => {
            let conn = this.getConn();
            conn.query(strSql, sqlParams, (err, results) => {
                if (err) {
                    //数据库执行失败
```

```
                    reject(err);
                }
                else {
                    //数据库执行成功
                    resolve(results);
                }
                conn.end();
            });
        });
        return p;
    }
}

module.exports = DBUtils2;
```

> 关键点在于上面的`executeSql`这个方法里面，我们把默认参数换成了`rest`剩余参数，所实现的效果是一样的，但是在调用这个方法的时候，所使用的参数就不一样了

以下是调用过程

```
/**
 * 测试 rest参数下面的DBUtils2的使用
 */
const DBUtils2 = require("./DBUtils2");

//无参数的
async function queryStuInfoList() {
    try {
        let db = new DBUtils2();
        let strSql = " select * from stu_info ";
        let results = await db.executeSql(strSql);
        console.log(results);
    } catch (error) {
        console.log("数据执行失败");
    }
}

//有参数的
async function queryStuInfoList2() {
    try {
        let db = new DBUtils2();
        let strSql = " select * from stu_info where stu_age > ? and stu_sex = ?
";
        //请注意，这里与之前不一样，没有中括号
        let results = await db.executeSql(strSql, 200, "男");
        console.log(results);
    } catch (error) {
        console.log("数据执行失败");
    }
}

queryStuInfoList2();
```

> 在原来的默认参数里面，我们的参数是要使用中括号去包含起来，而现在的剩余参数里面，我们直接传值就可以了