第6章 SQL数据库操作

Node.js开发实战教程

内容导航

Contents

6.1 操作MySQL数据库

- MySQL服务器安装和基本使用
- 连接MySQL数据库
- 执行数据库操作
- 记录增查改删操作
- 高级操作

6.2 优雅地编写异步代码

6.3 使用Node.js ORM框架操作关系数据库

6.4 实战演练——图书借阅记录管理

Section 00

【学习目标】

（1）了解MySQL数据库；
（2）使用Node.js连接和访问MySQL数据库。

Section 01

【MySQL服务器安装和基本使用】

▸▸▸ 安装MySQL服务器

- 如果已有MySQL数据库服务器，则可以直接使用MySQL，否则需要安装MySQL。
- MySQL社区版遵循GPL许可协议，可以从其官网上免费下载。

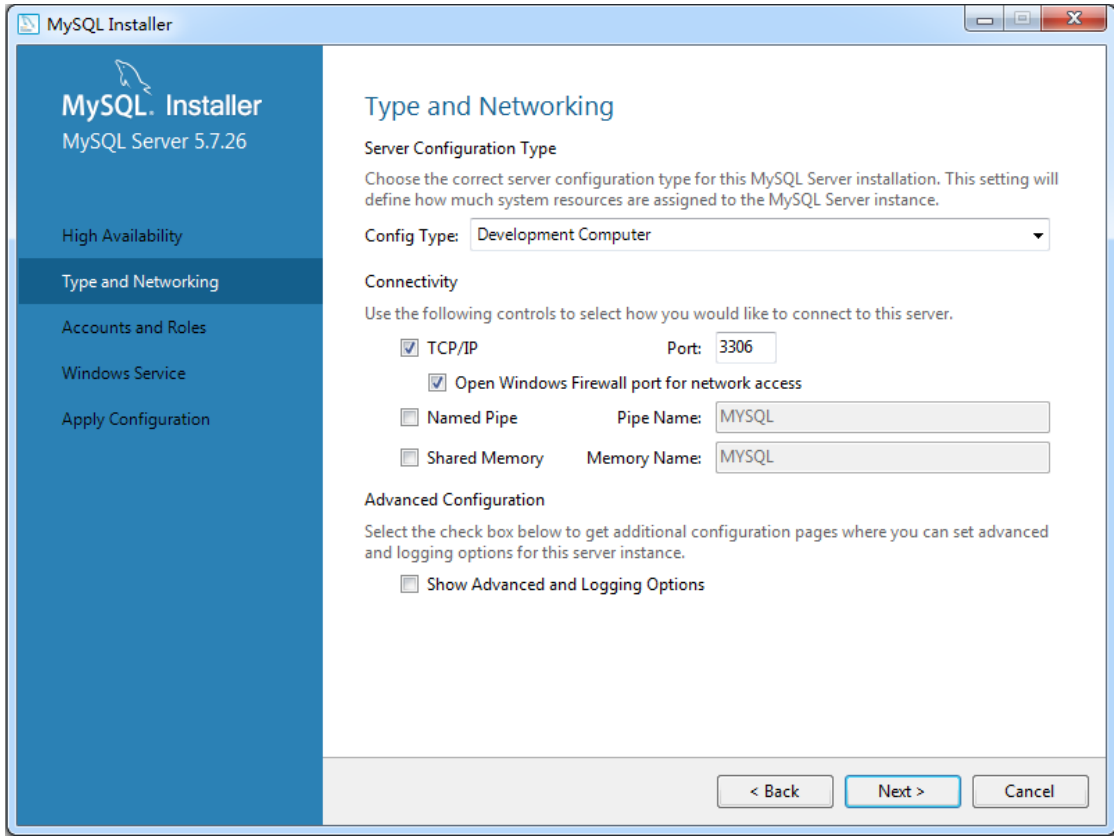
Section 01

【MySQL服务器安装和基本使用】

▸▸▸ 安装MySQL服务器

- 以在Windows 7系统中安装MySQL 5.7.26社区版为例。
- 安装该版本之前需要确认系统已安装有Visual C++ 2013运行库和Microsoft .NET Framework 4.5.2（或更高版本）。
- 运行程序安装向导，根据提示完成MySQL服务器的安装和初始配置。

设置MySQL服务器配置类型和网络



Section 01

【MySQL服务器安装和基本使用】

▸▸▸ 使用命令行界面操作MySQL

- 进入MySQL控制台

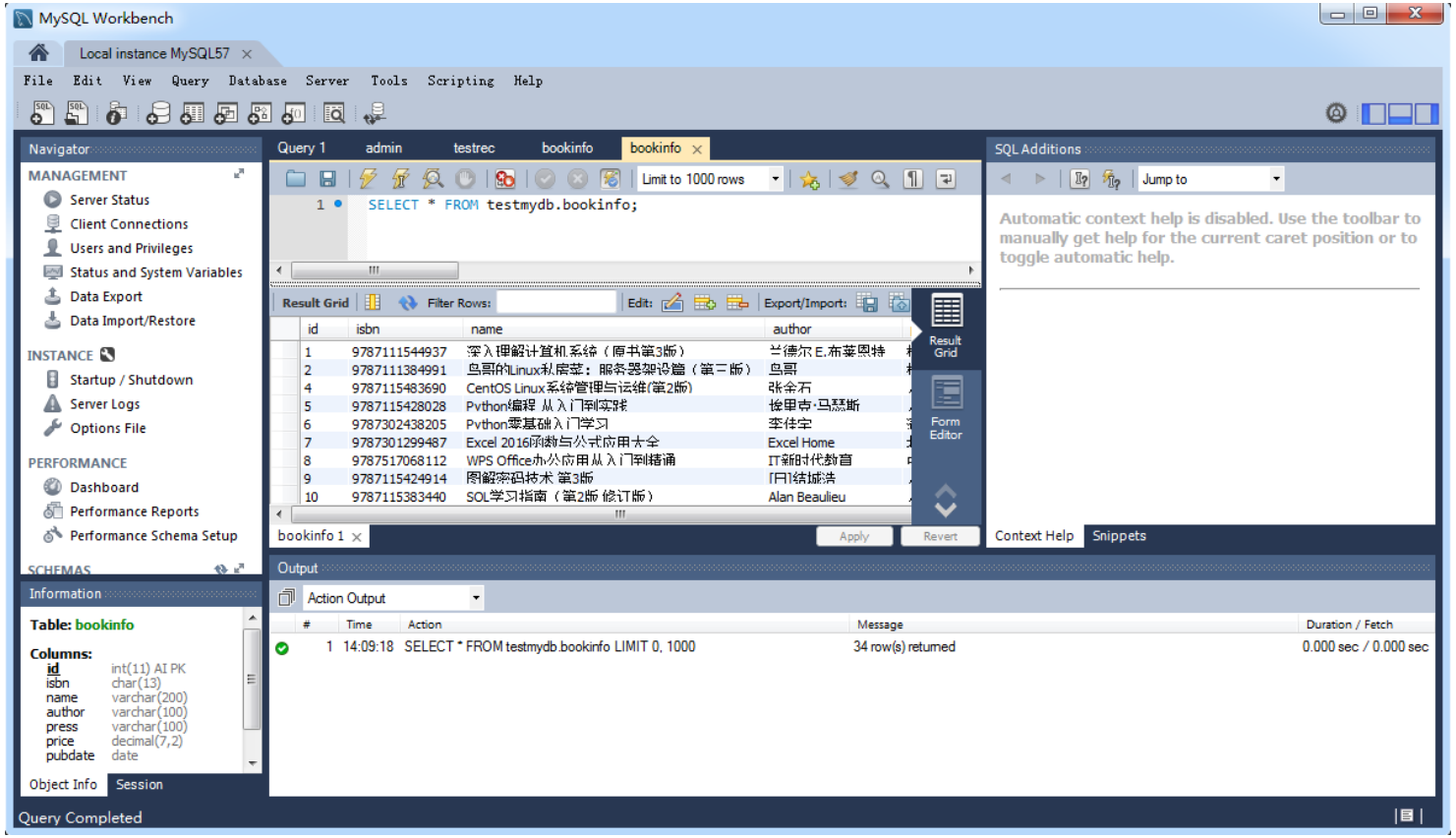
  mysql -h hostname -u username -p

- 常用命令
  - mysql->CREATE DATABASE dbname;//创建数据库
  - mysql->CREATE TABLE tablename;//创建表
  - mysql->SHOW DATABASES;//显示可用的数据库列表
  - mysql->USE dbname;//选择数据库
  - mysql->SHOW TABLES;//显示可用的表
  - mysql->DESCRIBE tablename;//显示表的信息
- 导出整个数据库的命令：mysqldump -u 用户名 -p 数据库名 > 导出的文件名
- 导出某个表的命令：mysqldump -u 用户名 -p 数据库名 表名> 导出的文件名
- 创建数据库，进入MySQL控制台后可使用source命令导入数据库。
- 直接导入数据库：mysql -u用户名 -p密码 数据库名 < 导入的数据库文件

Section 01

【MySQL服务器安装和基本使用】

▶▶▶ 使用图形界面工具管理MySQL



Section 02

【Node.js的MySQL驱动】

- Node.js的原生MySQL驱动库名为mysql。
- MySQL2项目是原生MySQL驱动项目的升级版本，兼容mysql并支持其主要特性，提供新的特性:
  - 更快更好的性能
  - 预处理语句
  - 对编码和排序规则的扩展支持
  - Promise包装器
  - SSL与认证开关
  - 自定义流
- 安装MySQL2 驱动（库名称为mysql2 ）：

  npm install mysql2

Section 03

【连接MySQL数据库】

▸▸▸ 建立连接

➢ 显式建立连接示例

```
const mysql = require('mysql2');
//创建到数据库的连接
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password : 'abc123',
  database: 'testmydb'
});
connection.connect(function(err) {
  if (err) {
    console.error('连接错误: ' + err.stack);
    return;
  }
  console.log('连接ID: ' + connection.threadId);
});
```

Section 03

【连接MySQL数据库】

▶▶▶ 建立连接

➢ 隐式建立连接示例

```
var mysql      = require('mysql2');
var connection = mysql.createConnection(...);//此处省略连接选项代码
connection.query('SELECT * FROM `bookinfo`',function(err, results) {
    console.log(results); // 结果包括由MySQL服务器返回的行
});
```

Section 03

【连接MySQL数据库】

▸▸▸ 设置连接选项

- host：连接的数据库地址，默认为localhost。
- port：连接地址对应的端口，默认3306。
- user：用于连接的MySQL用户名。
- password：用户的密码。
- database：所需连接的数据库的名称（可选项）。
- charset：连接的编码形式（默认为utf8_general_ci），决定整理排序规则。
- timezone：MySQL服务器上配置的时区（默认local）。
- dateStrings：将强制日期类型（TIMESTAMP、DATETIME或DATE）作为字符串返回。
- connectTimeout：设置连接时，返回失败前的未响应等待时间。

▸▸▸ 终止连接

```
connection.end(function(err) {
  // 连接终止
});
```

或者

```
connection.destroy();
```

Section 04

【执行数据库操作】

▶▶▶ 第1种形式：.query(sqlString, callback)

```
const mysql = require('mysql2');
//建立连接
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password : 'abc123',
  database: 'testmydb'
});
connection.connect();
//执行查询操作
connection.query('SELECT * FROM `bookinfo` WHERE `press` = "人民邮电出版社"', function
(err, results, fields) {
    if (err) throw err;
    console.log('-------查询记录----------');
    console.log('查询结果:',results);
    console.log('查询结果字段:',fields);
});
connection.end();   //终止连接
```

Section 04

【执行数据库操作】

▸▸▸ 第2种形式：.query(sqlString, values, callback)

```
connection.query('SELECT * FROM `bookinfo` WHERE `press` = ?', ['人民邮电出版社'],
function (err, results, fields) {
});
```

▸▸▸ 第3种形式：.query(options, callback)

```
connection.query({
  'SELECT * FROM `bookinfo` WHERE `press` = ?',
  timeout: 40000, // 40秒
  values: ['人民邮电出版社']
}, function (error, results, fields) {
});
```

Section 04

【执行数据库操作】

▸▸▸ 第2种和第3种形式的结合

```
connection.query({
    'SELECT * FROM `bookinfo` WHERE `press` = ?',
    timeout: 40000, // 40s
  },
  ['人民邮电出版社'],
  function (err, results, fields) {
);
```

▸▸▸ 只有单个占位符的查询

```
connection.query('SELECT * FROM `bookinfo` WHERE `press` = ?', '人民邮电出版社',
function (err, results, fields) {
});
```

▸▸▸ 并行执行查询

Section 05

【增查改删操作】

▶▶▶ 增加记录

```
//此处省略建立连接代码
//定义增加记录的SQL语句和参数
var  addSql = ' INSERT INTO `bookinfo`(`isbn`, `name`, `author`, `press`, `price`, `pubdate`)
VALUES(?,?,?,?,?,?)';
var  addSql_Params = ['9787115488435', '人工智能（第2版）','史蒂芬•卢奇','人民邮电出版社
',108.00,'2018-09-01'];
//通过查询命令执行增加操作
connection.query(addSql,addSql_Params,function (err, results) {
    if (err) throw err;
    console.log('-------插入记录----------');
    console.log('插入记录的ID:',results.insertId);
    console.log('插入结果:',results);
});
connection.end();
```

Section 05

【增查改删操作】

▶▶▶ 修改记录

```
//此处省略建立连接代码
//定义修改记录的SQL语句和参数
var updateSql = 'UPDATE bookinfo SET author = ?,price = ? WHERE id = ?';
var updateSql_Params = ['[日]结城浩',87.5,9];
//通过查询命令执行修改操作
connection.query(updateSql,updateSql_Params,function (err, result) {
    if (err) throw err;
    console.log('-------修改记录----------');
    console.log('修改所影响的行数:',result.affectedRows);
    console.log('修改所改变的行数:',result.changedRows);
});
connection.end();
```

Section 05

【增查改删操作】

▶▶▶ 删除记录

```
//此处省略建立连接代码
//定义删除记录的SQL语句
var  delSql = 'DELETE FROM bookinfo WHERE id = 11';
//通过查询命令执行删除操作
connection.query(delSql,function (err, result) {
    if (err) throw err;
    console.log('-------删除记录----------');
    console.log('删除的行数:',result.affectedRows);
});
connection.end();
```

Section 06

【防止SQL注入攻击】

- 转义可以通过mysql.escape()、connection.escape()或pool.escape()方法来实现。

```
var userId = 'some user provided value';
var sql    = 'SELECT * FROM users WHERE id = ' + connection.escape(userId);
connection.query(sql, function(err, results) {
  // ...
});
```

- 可以将符号?作为查询字符串中的占位符以替代要转义的值。

```
connection.query('SELECT * FROM users  WHERE id = ?', [userId], function(err, results) {
  // ...
});
```

Section 06

【防止SQL注入攻击】

- 不可信的SQL标识符应使用mysql.escapeId(identifier)、connection.escapeId(identifier)或pool.escapeId(identifier)方法对其进行转义。

```
var sorter = 'date';  //这是一个列名
var sql    = 'SELECT * FROM posts ORDER BY ' + connection.escapeId(sorter);
connection.query(sql, function(err, results) {
  // ...
});
```

- escapeId()转义方法支持添加限定符，将两部分内容都进行转义，例如：

```
var sorter = 'date';
var sql    = 'SELECT * FROM posts ORDER BY ' + connection.escapeId('posts.' + sorter);
// 等同于：SELECT * FROM posts ORDER BY 'posts'.'date'
```

- 可以将符号??作为占位符来替代要转义的SQL标识符：

```
var userId = 1;
var columns = ['username', 'email'];
var query = connection.query('SELECT ?? FROM ?? WHERE id = ?', [columns, 'users', userId],
function(err, results) {
  // ...
});
```

Section 07

【使用流式查询】

```
var query = connection.query('SELECT * FROM posts');
query
  .on('error', function(err) {
    //处理错误，这之后会触发 'end' 事件
  })
  .on('fields', function(fields) {
    // 处理字段数据
  })
  .on('result', function(row) {
    connection.pause();//如果处理过程涉及到I/O操作，暂停连接会很有用
    processRow(row, function() {
      connection.resume();
    });
  })
  .on('end', function() {
    //所有行都已经接收完毕
  });
```

Section 08

【使用预处理语句】

```
connection.execute(
  'SELECT * FROM 'table' WHERE 'name' = ? AND 'age' > ?',
  ['Rick C-137', 53],
  function(err, results, fields) {
    console.log(results);
  }
);
```

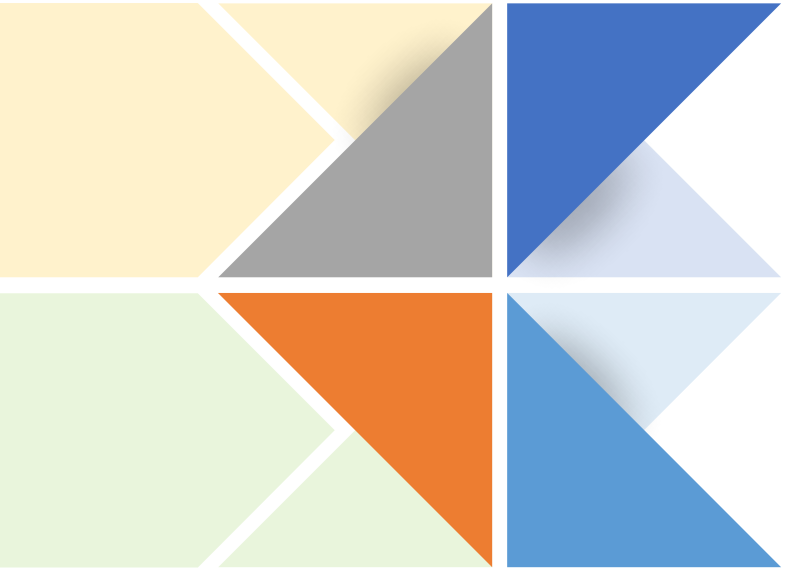
Section 09

【使用连接池】

```
const mysql = require('mysql2');
//创建连接池
const pool = mysql.createPool({
  host: 'localhost',
  user: 'root',
  password: 'abc123',
  database: 'testmydb',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});
//使用连接池
pool.query('SELECT * FROM 'bookinfo' ', function (err, results, fields) {
    console.log('查询结果:',results);
});
```

内容导航

Contents



Section 00

【学习目标】

（1）了解Node.js异步编程方法；
（2）能够编写数据库操作的异步代码。。

- 解决回调地狱问题。
- 解决异步流程控制问题。
- 简化在异步代码中捕获异常。

以同步方式编写异步代码的解决方案实质上只是改变了代码编写方式，简化和优化代码的写法，使开发人员以更优雅的方式编写异步代码，以解决异步代码的流程控制问题，降低了异步编程难度。但程序本质上还是异步执行的，这些方案并没有对程序本身进行优化，也没有提高应用程序性能。

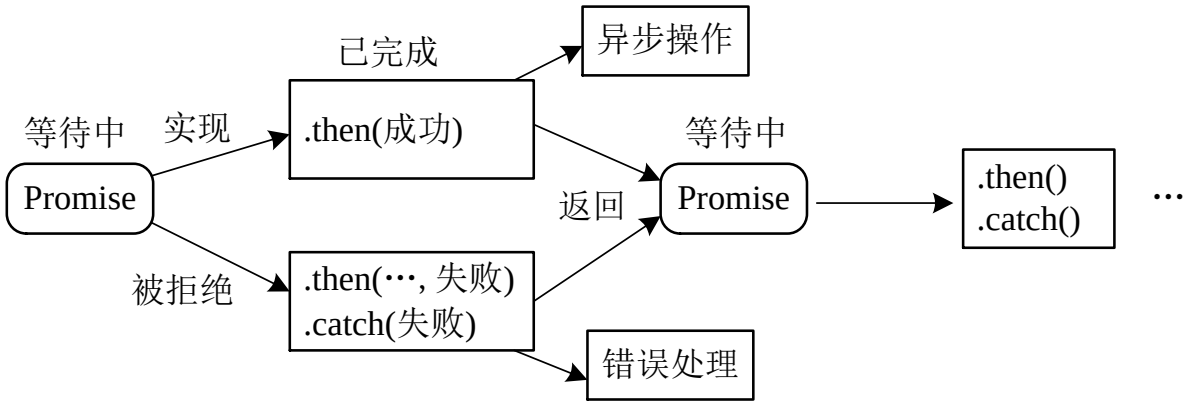
Section 01

【Promise】

▸▸▸ 什么是Promise

- Promise对象代表一个异步操作，有以下3种状态。
  - 等待（Pending）：初始状态，没有实现也没有被拒绝。
  - 实现（Fulfilled）：操作已成功完成。
  - 被拒绝（Rejected）：操作失败。
- 只有异步操作的结果才可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。
- 简化回调嵌套只是Promise的一项基本功能，Promise最关键的是状态，Promise通过维护和传递状态的方式使回调函数能够及时被调用，这比直接传递回调函数要简单和灵活得多。

Promise工作机制



Section 01

【Promise】

▸▸▸ 创建Promise对象

```
let promise = new Promise((resolve, reject) => {
    //执行异步操作代码
    if (/*成功 */) {
        resolve(value);    //异步操作执行成功后的回调函数
    }else {
        reject(error);    //异步操作执行失败后的回调函数
});
```

Section 01 【Promise】

Promise的方法

- then()

```
promise.then((value) => {      //成功
    console.log('成功',value);
},(error) => {                 //失败
    console.err('失败',error);
})
```

- catch()

```
promise.then((value) => {
    console.log('成功',value);
}).catch((error) => {
    console.err('失败',error);
})
```

Section 01 【Promise】

Promise的方法

- Promise.resolve()
- Promise.reject()
- Promise.all()

```
var promise1 = Promise.resolve(70);
var promise2 = 82;
var promise3 = new Promise(function(resolve) {
  setTimeout(resolve,1000,95);   //1秒钟之后执行
});
Promise.all([promise1, promise2, promise3]).then(function(values) {
  console.log(values);
});// 输出数组: [ 70, 82, 95 ]
```

- Promise.race()

Promise的链式操作

Section 02

【Generator】

▸▸▸ 什么是Generator

- Generator是一个生成器，同时也是一个状态机，其内部拥有值和相关状态。
- 生成器返回一个迭代器（Iterator）对象，可以编写程序通过该对象遍历相关的值和状态，以保证正确的执行顺序。
- Generator本质上是一个函数，其最大的特点是可以被中断，然后再恢复执行。
- Promise对象创建之后，就会处于等待状态并开始执行，直到状态改变之后才能进行下一步操作。
- Generator函数可以由用户执行中断，去执行其他操作，然后从中断处恢复执行。

Section 02

【Generator】

▸▸▸ Generator函数声明及其执行

```
function* genFunc(score) {  //声明Generator函数声明
    yield '积分'+score;
    yield '积分'+(score + 10);
    return '积分'+(score + 20);
    yield '积分'+(score + 30);
}
var gen = genFunc(10);  // 调用之后返回了一个迭代器对象
console.log(gen.next());  // 返回对象{ value: '积分10', done: false }
console.log(gen.next());  // 返回对象{ value: '积分20', done: false }
console.log(gen.next());  // 返回对象{ value: '积分30', done: false }
console.log(gen.next());  // 返回对象{ value: undefined, done: true }
```

Section 02

【Generator】

调用Generator对象

- 调用Generator对象有两种方法。
  - 不断地调用next()方法。next()方法会执行Generator的代码，每次遇到yield就返回一个对象（形式为{value: x, done: true/false}），然后“暂停”，返回的value就是yield的返回值，done表示该Generator对象是否已经执行结束了。
  - 直接用循环语句for ... of自动遍历Generator对象。

```
for (let obj of gen) {
  console.log(obj);
  break; // 关闭迭代器，触发return
}   //最终返回:积分10
```

使用Generator解决异步回调问题

- 可以将异步操作写在yield语句中，在调用next()方法后再向后执行。
- yield除了使用表达式之外，还可以使用Promise对象。由yield对表达式求值并返回，调用next()方法会返回一个Promise对象，接着调用then()方法，并在回调函数中通过next()方法将结果传回Generator。Generator与Promise对象联合使用，大大改进异步代码编写流程。

Section 03

【co模块】

▸▸▸ co模块实现思路

co模块将Generator函数包装成一个Promise对象，作为参数传递给co()方法，在co()方法内部自动执行yield，以将异步操作改为“顺序”执行。co()方法包括一个Generator函数，co()方法在Generator函数中使用yield指向Promise对象，通过递归调用next()方法将每一个Promise的值返回，从而实现异步转“同步”的代码编写方式。

co()方法返回一个Promise对象，可以调用then()和catch()方法对Generator函数返回的结果进行传递，以方便后续的成功处理或者错误处理。

Section 03

【co模块】

▸▸▸ co模块的基本使用

- 要使用co模块，需要安装相应的包：

```
npm install co
```

- 然后在程序中导入该模块：

```
const co = require('co');
```

- 示例： co返回一个Promise对象

```
co(function* () {  //声明一个co-generator函数
  var result = yield Promise.resolve(true);
  return result;
}).then(function (value) {
  console.log(value);
}, function (err) {
  console.error(err.stack);
});
```

Section 03

【co模块】

可用的yield对象

- co模块目前支持的对象类型
  - Promise
  - Thunks（函数）
  - 数组（并行执行）
  - 对象（并行执行）
  - Generator（代理）
  - Generator函数（代理）
  - 对象嵌套

- 数组用于并行完成所有的yield异步操作：

```
co(function* () {
  var res = yield [
    Promise.resolve('路人甲'),
    Promise.resolve('路人乙'),
  ];
  console.log(res); // => ['路人甲', '路人乙']
});
```

- 对象用于并行处理：

```
co(function* () {
    var res = yield {
      男配: Promise.resolve('路人甲'),
      女配: Promise.resolve('路人乙'),
    };
    console.log(res); // 返回 : { '男配': '路人甲', '女配': '路人乙' }
  });
```

Section 03

【co模块】

▸▸▸ co模块的错误捕获

```
const co = require('co');
co(function *(){
  try {
    yield Promise.reject(new Error('发生错误！'));
  } catch (err) {
    console.error(err.message); // "发生错误！"
 }
}).catch(onerror);
function onerror(err) {
  // 记录未捕获的错误，co模块不会抛出未处理的任何错误，需要自己处理所有错误
  console.error(err.stack);
}
```

Section 04

【async/await】

async/await的特点

- async/await从上到下顺序执行，符合编写代码的习惯。
- async/await可以传递的参数数量不受限制。
- 同步代码和异步代码可以一起编写，只是要注意异步过程需要包装成一个Promise对象并置于await关键字后面。
- async/await基于协程（Coroutine）的机制，是对异步过程更精确的一种描述。
- async/await是对Promise的改进。只是语法糖，本质上仍然是Promise。

Section 04

【async/await】

async/await的基本用法

```
function resolveAfter1Seconds() {   //将异步过程包装为Promise
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('已成功');
    }, 1000);
  });
}
async function asyncFunc() {    //定义async函数
  console.log('正在执行async函数');
  var result = await resolveAfter1Seconds();   // await表达式
  console.log(result);  // 输出'已成功'
}
asyncFunc();   //调用async函数
console.log('async函数代码开始执行');
```

Section 04 【async/await】

async函数

- 定义async函数的语法格式如下：
- async function name([param[, param[, ... param]]]) { statements }
- async函数返回的是Promise对象。上例中调用async函数的代码可扩展为：

```
asyncFunc().then(data => {
    console.log(data);     // 获取async函数返回的内容
}).catch(error => {
    console.log(error);    // 捕获async函数的错误
});
```

Section 04

【async/await】

await操作符

- await操作符用于等待一个Promise对象，语法格式如下：

```
[rv] = await expression;
```

async/await的错误处理

- 在async函数外部，对async函数返回的Promise对象采用catch()方法进行错误处理。
- 在async函数内部使用await表达式，采用try/catchy语句块同步进行错误处理。例如：

```
async function getProcessedData(url) {
  let v;
  try {
    v = await downloadData(url);
  } catch(e) {
    v = await downloadFallbackData(url);
  }
  return processDataInWorker(v);
}
```

- await后面的Promise对象自行处理错误，也就是将异步过程包装为Promise对象，在该对象的函数定义中处理错误。

Section 04 【async/await】

async/await的串行、并发和并行操作

- 串行是指按顺序执行每一个任务。
- 并发是指轮换执行多个任务，看起来好像多个任务同时执行。
- 并行是指真正地同时执行多个任务。
- 如果一个async函数中有多个await语句，程序会变成完全的串行操作。
- 当异步操作之间不存在结果的依赖关系时，可以使用Promise.all()方法实现并行，Promise.all()中的所有方法是同时执行的。先执行async函数将多个Promise合起来发起请求，然后再进行await操作。
- 并发也是将多个Promise合起来发起请求。

Section 05

【使用Promise包装器操作MySQL数据库】

▶▶▶ 使用基本的Promise

```
const pool = require('mysql2/promise').createPool({
    user: 'root',
    password: 'abc123',
    database: 'testmydb',
});
pool.getConnection()
  .then(conn => {
    const res = conn.query('SELECT * FROM 'bookinfo'');
    conn.release();
    return res;
  }).then(result => {
    console.log(result);
  }).catch(err => {
    console.log(err); //以上任何连接时或查询时错误
  });
```

Section 05

【使用Promise包装器操作MySQL数据库】

▸▸▸ 使用ES2017 async/await

```
async function getData(){    //声明一个async函数
    const mysql = require('mysql2/promise');
    const pool = mysql.createPool({ user: 'root',password: 'abc123', database: 'testmydb'});
    // 并行执行
    var results = await Promise.all([pool.query('SELECT * FROM 'bookinfo' WHERE 'press' = "人民邮电出版社"'), pool.query('SELECT * FROM 'bookinfo' WHERE 'press' = "清华大学出版社"')]);
    await pool.end();   //并行执行结束后关闭连接池
    return results;   //返回结果（Promise对象）
  };
  getData().then(data => {  // 调用async函数并获取async函数返回的内容
    console.log(data[0]);  //第1个查询的结果
    console.log(data[0]); //第2个查询的结果
}).catch(error => {
    console.log(error);   // 捕获async函数的错误
});
```

Section 05

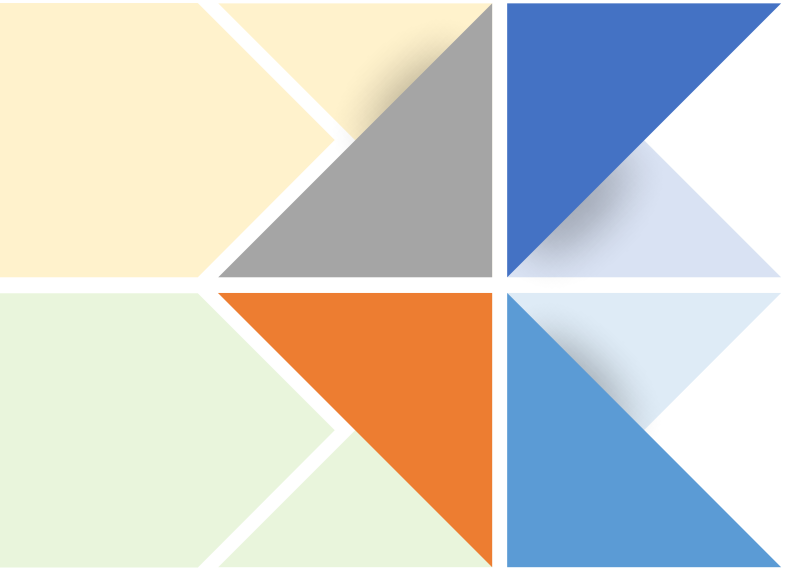
【使用Promise包装器操作MySQL数据库】

▶▶▶ 使用co模块

```
const mysql = require('mysql2');
const co = require('co');
co(function * () {   //声明一个co-generator函数
  const c = yield mysql.createConnectionPromise({user: 'root', password: 'abc123',database: 'testmydb' }); //获取连接
  const rows = yield c.query('SELECT * FROM 'bookinfo' WHERE 'press' = "人民邮电出版社"');  //执行查询语句
  console.log(rows);
  console.log(yield c.execute('SELECT * FROM 'bookinfo' WHERE 'press' = "清华大学出版社"'));  //执行预处理语句
  yield c.end();  //终止连接
});
```

内容导航

Contents



Section 00

【学习目标】

（1）了解Node.js的ORM框架；
（2）掌握Sequelize框架的用法。

Section 01

【Sequelize简介】

- ORM两大优势
  - 可以像操作对象一样操作数据库。
  - 提高开发效率。
- ORM对数据库进行高层封装，不足之处主要是会牺牲程序的执行效率。
- Sequelize是一款基于Promise的支持异步操作的Node.js ORM框架，支持Postgres、MySQL、SQLite和Microsoft SQL Server等多种数据库，具有强大的事务支持、关联关系、读取和复制等功能，很适合作为Node.js后端数据库的存储接口，有助于提高Node.js应用程序的开发效率。

Section 02

【Sequelize的基本使用】

▸▸▸ 安装sequelize库及数据库驱动

- 安装sequelize库

```
cnpm install --save sequelize
```

- 为数据库安装相应的Node.js数据库驱动：

```
npm install --save pg pg-hstore # Postgres数据库
npm install --save mysql2  #MySQL数据库
npm install --save mariadb   #MariaDB数据库
npm install --save sqlite3   # SQLite
npm install --save tedious # Microsoft SQL Server
```

Section 02

【Sequelize的基本使用】

▸▸▸ 建立连接

➢ 基本用法

```
const Sequelize = require('sequelize');
// 第1种方式：单独传递参数
const sequelize = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: /*可以是'mysql'、'mariadb'、'postgres'或'mssql'中任何一个 */
});
// 第2种方式：传递连接URL
const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname');
```

Section 02

【Sequelize的基本使用】

▸▸▸ 建立连接

➢ 示例

```
const Sequelize = require('sequelize');
const sequelize = new Sequelize('testmydb', 'root', 'abc123', {
  //连接选项
  host: 'localhost', // 数据库地址
  dialect: 'mysql', // 指定连接的数据库类型
  pool: {
    max: 5, // 连接池的最大连接数量
    min: 0, // 连接池的最小连接数量
    idle: 10000 // 如果一个线程10秒钟内没有被使用，那么就释放线程
  }
});
//测试连接
sequelize
  .authenticate()
  .then(() => {
    console.log('成功建立连接');
  })
  .catch(err => {
    console.error('未能连接到数据库： ', err);
  });
```

Section 02

【Sequelize的基本使用】

定义模型

- 使用Sequelize.Model.init(attributes, options)函数

```
const Model = Sequelize.Model;
class User extends Model {}
User.init({
  // 属性设置
  name: {
    type: Sequelize.STRING,
    allowNull: false
  },
  email: {
    type: Sequelize.STRING
    // allowNull（允许空值）默认为true
  }
}, {
  sequelize,
  modelName: 'user'   //此处定义模型名
  // 选项
});
```

Section 02 【Sequelize的基本使用】

定义模型

- 使用sequelize.define('name', {attributes}, {options})

```
const User = sequelize.define('user', {  // user为模型名
  // 属性
  name: {
    type: Sequelize.STRING,
    allowNull: false
  },
  email: {
    type: Sequelize.STRING
  }
}, {  // 选项
});
```

Section 02

【Sequelize的基本使用】

▶▶▶ 将模型与数据库同步

```
// 选项force: true 表示如果表已经存在，在新建前会删除厚表
User.sync({ force: true }).then(() => {
  // 数据库中的表与模型定义一致
  return User.create({
    name: '小莉',
    email: 'xiaoli@abc.com'
  });
});
```

Section 02

【Sequelize的基本使用】

▶▶▶ 数据的增查改删

```
// 创建新的用户
User.create({ name: "小彤", email: "xiaotong@abc.com" }).then(() => {
  console.log("已添加");
});
// 查找所有用户
User.findAll().then(users => {
  console.log("所有用户:", JSON.stringify(users, null, 4));
});
// 将没有邮箱的用户的邮箱改为it@abc.com
User.update({ eamil: "it@abc.com" }, {
  where: {
    email: null
  }
}).then(() => {
  console.log("已改完");
});
// 删除名为小红的用户
User.destroy({
  where: {
    name: "小红"
  }
}).then(() => {
  console.log("已删除");
});
```

Section 02

【Sequelize的基本使用】

▸▸▸ Promises和async/await

- Sequelize使用Promise来控制异步操作流程，增查改删操作都可使用then()方法。
- 如果Node.js版本支持，可以使用ES2017 async/await语法来编写Sequelize所用的异步调用代码。
- Sequelize的所有Promise对象也是Bluebird的Promise对象，也可以使用Bluebird API来操作。
- Sequelize返回的Promise对象也可以通过co模块来操作。

Section 03

【使用Sequelize的关联】

▸▸▸ 基本概念

- 源和目标

```
class User extends Model {}
User.init({
  name: Sequelize.STRING,
  email: Sequelize.STRING
}, {
  sequelize,
  modelName: 'user'
});
class Project extends Model {}
Project.init({
  name: Sequelize.STRING
}, {
  sequelize,
  modelName: 'project'
});
User.hasOne(Project);
```

Section 03

【使用Sequelize的关联】

▸▸▸ 基本概念

➢ 外键

- Sequelize创建模型之间的关联时，将自动创建带有约束的外键引用。
- 创建Task和User模型之间的关系会在tasks表中插入外键userId，并将该外键作为对users表的引用

```
class Task extends Model {}
Task.init({ title: Sequelize.STRING }, { sequelize, modelName: 'task' });
class User extends Model {}
User.init({ username: Sequelize.STRING }, { sequelize, modelName: 'user' });
User.hasMany(Task); // 自动将userId添加到Task模型
Task.belongsTo(User); //也会自动将userId添加到Task模型
```

Section 03

【使用Sequelize的关联】

▸▸▸ 一对一关联

➢ belongsTo关联

- belongsTo关联在源模型上存在一对一关系的外键。
- 示例：Player通过players表的外键作为Team的一部分

```
class Player extends Model {}
Player.init({/* 属性定义 */}, { sequelize, modelName: 'player' });
class Team extends Model {}
Team.init({/* 属性定义 */}, { sequelize, modelName: 'team' });
Player.belongsTo(Team); // 向Team模型添加teamId属性以保存Team的主键值
```

Section 03

【使用Sequelize的关联】

▸▸▸ 一对一关联

➢ hasOne关联

- hasOne关联是在目标模型上存在的一对一关系的外键的关联。
- 示例：向User模型添加projectId属性

```
const User = sequelize.define('user', {/* ... */})
const Project = sequelize.define('project', {/* ... */})
Project.hasOne(User)  // 单向关联
```

Section 03

【使用Sequelize的关联】

▸▸▸ 一对一关联

➢ hasOne和belongsTo之间的区别

- hasOne在目标模型中插入关联键，而belongsTo在源模型中插入关联键。
- 示例

```
const Player = this.sequelize.define('player', {/* 属性定义 */})
const Coach  = this.sequelize.define('coach', {/* 属性定义 */})
const Team   = this.sequelize.define('team', {/* 属性定义 */});
```

- 当有关关联的信息存在于源模型中时，可以使用belongsTo关联。例中Player适用于belongsTo关联，因为它具有teamId列。

```
Player.belongsTo(Team) // teamId 将被添加到源模型Player中
```

- 当有关关联的信息存在于目标模型中时，可以使用hasOne关联。例中Coach适用于hasOne关联，因为Team模型将其Coach的信息存储为coachId列。

```
Coach.hasOne(Team) // coachId将被添加到目标模型Team中
```

Section 03

【使用Sequelize的关联】

▸▸▸ 一对多关联

- 一对多关联将一个源与多个目标连接起来，多个目标连接到同一个特定的源：

```
const User = sequelize.define('user', {/* ... */})
const Project = sequelize.define('project', {/* ... */})
Project.hasMany(User, {as: 'Workers'})
```

- hasMany()用于定义一对多关联。
- 要在不同的列上关联记录，可以使用sourceKey选项指定源键：

```
const City = sequelize.define('city', { countryCode: Sequelize.STRING });
const Country = sequelize.define('country', { isoCode: Sequelize.STRING });
// 可以根据国家代码连接国家和城市
Country.hasMany(City, {foreignKey: 'countryCode', sourceKey: 'isoCode'});
City.belongsTo(Country, {foreignKey: 'countryCode', targetKey: 'isoCode'});
```

Section 03

【使用Sequelize的关联】

▸▸▸ 多对多关联

- 多对多关联用于将源与多个目标相连接，目标也可以连接到多个源：

```
Project.belongsToMany(User, {through: 'UserProject'});
User.belongsToMany(Project, {through: 'UserProject'});
```

内容导航

Contents

Section 00

【学习目标】

（1）综合运用SQL数据库知识；

（2）掌握利用Sequelize操作MySQL数据库的编程。

此案例用于记录图书的借阅信息，为简化实验过程，只有两个表books（图书）和readers（读者），它们之间是一对多的关系，一种图书可以对应多个读者，使用自动建立表结构的方案。在实际的应用程序开发中，往往将数据部分独立出来，作为模型部分，这样有利于各模块的解耦和扩展。

Section 01

【编写模型部分代码】

- 定义图书数据模型

```
const Sequelize = require('sequelize');
module.exports = (sequelize) => {
    var Book = sequelize.define('book', {
        isbn: { type: Sequelize.STRING },
        name: { type: Sequelize.STRING },
        author: { type: Sequelize.STRING },
        press: { type: Sequelize.STRING },
        price: { type: Sequelize.DECIMAL(10, 2) },
        pubdate: { type: Sequelize.DATEONLY }
    });
    return Book;
};
```

Section 01

【编写模型部分代码】

- 定义读者数据模型

```
const Sequelize = require('sequelize');
module.exports = (sequelize) => {
    var Reader = sequelize.define('reader', {
        name: { type: Sequelize.STRING },
        mobile: { type: Sequelize.STRING },
        email: { type: Sequelize.STRING }
    });
    return Reader;
};
```

Section 01

【编写模型部分代码】

- 同步数据模型

```
const Sequelize = require('sequelize');
const sequelize = new Sequelize('testmydb', 'root', 'abc123', {
  //连接选项
  host: 'localhost', // 数据库地址
  dialect: 'mysql', // 指定连接的数据库类型
  define: {
    'charset':'utf8'     //解决中文输入问题
  },
  pool: {    // 建立连接池
    max: 5, // 连接池的最大连接数量
    min: 0, // 连接池的最小连接数量
    idle: 20000 // 如果一个线程在20秒内没有被使用过，那么释放线程
  }
});
const Book = require('./book_model')(sequelize);//导入Book模型
const Reader = require('./reader_model')(sequelize);//导入Reader模型
Book.hasMany(Reader);   //一种图书有多个读者
Reader.belongsTo(Book);  //一个读者对应一种图书
sequelize.sync();    //自动同步所有的模型，使用关联时要使用。首次添加数据之后可将其注释掉
exports.Book = Book;
exports.Reader = Reader;
```

Section 02

【编写数据操作部分代码】

- 添加数据

```
const Book  = require('./mydb').Book;
const Reader  = require('./mydb').Reader;
async function addBook() {
  const result = await Book.create(
    {
      isbn:"9787115474582",
        (此处略)
  },
    { include: [Reader] } //指定关联关系，读者数据自动插入到读者表
  );
  return result;
}
addBook().then(data => {
  console.log("添加的数据:", JSON.stringify(data, null, 4)); // 获取返回的内容
}).catch(error => {
  console.log(error);   // 捕获错误
});
```

Section 02

【编写数据操作部分代码】

- 获取数据

```
const Book  = require('./mydb').Book;
const Reader  = require('./mydb').Reader;
async function getBook() {
  const result = await Book.findAll();
  return result;
}
getBook().then(data => {
  console.log("查询的数据:", JSON.stringify(data, null, 4)); // 获取返回的内容
}).catch(error => {
  console.log(error);   // 捕获错误
});
```

Section 02

【编写数据操作部分代码】

- 更改数据

```
const Book  = require('./mydb').Book;
const Reader  = require('./mydb').Reader;
async function updateBook() {
  const result = await Book.update(
    { price:98.00 },
    { where: { id: 1 } }
  );
  return result;
}
updateBook().then(data => {
  console.log("修改数据的ID:", JSON.stringify(data, null, 4)); // 获取返回的内容
}).catch(error => {
  console.log(error);   // 捕获错误
});
```

Section 02

【编写数据操作部分代码】

- 删除数据

```
const Book  = require('./mydb').Book;
const Reader  = require('./mydb').Reader;
async function delBook() {
  const result = await Book.destroy(
    { where: { id: 1 } }
  );
  return result;
}
delBook().then(data => {
  console.log("删除数据的ID:", JSON.stringify(data, null, 4)); // 获取返回的内容
}).catch(error => {
  console.log(error);   // 捕获错误
```

本章小结

Summary

本章的主要内容是编写Node程序操作关系数据库。关系数据库适合事务要求比较多的场景，如财务管理、订单管理、积分管理等。MySQL是比较典型的关系数据库，掌握它之后，对其他关系数据库操作也会触类旁通。而ORM框架进一步抽象了SQL语句，将其封装为方便操作的对象，掌握Sequelize框架，就能在Node.js程序中连接和访问主流的关系数据库。本章还介绍了异步代码的编写方法，也就是使用同步的、更符合思维习惯的方式优雅地编写异步代码，使参数传递更方便，顺序和流程更加清晰，其中Promise是基础，async/await是首选的方案。

Thank you