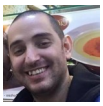
围绕 sqlite 构建一个简单的 Typescript ORM



艾伦·托马

2021 年 9 月 4 日 警察

这是构建 ORM 并完全控制数据库和查询的设置方式的一种非常简单的方法。

在本技巧中，您将学习如何围绕您最喜欢的 sqlite 构建一个简单的 TypeScript ORM。

背景

我经常使用 C#，当我与`database`.

现在我开始使用构建应用程序`react-native`并希望`sqlite`用作存储。

但我讨厌使用 SQL 查询，因为它看起来不如 ORM 好。

当我想使用那里的 ORM 库时，它非常大并且有太多我不需要的代码，所以我构建了自己的 ORM。

使用代码

首先，我们需要的是能够知道表的构造器。

所以我们会知道`database`桌子的样子。这就是为什么我们的第一步是构建`TableStructor`及其组件的原因。

JavaScript 缩小▲ 复制代码

```
export enum ColumnType {
    Number,
    String,
    Decimal,
    Boolean
}

export class constraint{
    columnName: string;
    contraintTableName:string;
    contraintColumnName:string;

    constructor(columnName: string, contraintTableName:string, constrainColumnName: string){
        this.columnName = columnName;
        this.contraintTableName = contraintTableName;
        this.contraintColumnName = constrainColumnName;
    }
}
```

```
export class columnStructor {
    columnType: ColumnType;
    nullable?: boolean;
    columnName: string;
    isPrimary?: boolean;
    autoIncrement?: boolean;

    constructor(columnType: ColumnType, columnName: string,
    isPrimary?: boolean, autoIncrement?: boolean, nullable?: boolean) {
        this.columnType = columnType;
        this.nullable = nullable;
        this.columnName = columnName;
        this.isPrimary = isPrimary;
        this.autoIncrement = autoIncrement;
    }
}

export default class tablaStructor {
    tableName: string;
    columns: columnStructor[];
    constraints?:constraint[];

    constructor(tableName: string, columns:columnStructor[], constraint?:constraint[]){
        this.tableName = tableName;
        this.columns = columns;
        this.constraints = constraint;
    }
}
```

我们的下一步是构建一个其他类可以使用的基类`extend`。让我们称之为`BaseModule`。

JavaScript 复制代码

```
// All your tables names should be added here.
export type TableNames = "Users" | "Items" | "System"
export default class BaseModule {
    public id: number;
    public tableName: TableNames;
    constructor(tableName: TableNames, id?: number) {
        this.id = id ?? 0;
        this.tableName = tableName;
    }
}
```

现在让我们创建我们的`Users`模块。它应该非常简单。

JavaScript 缩小▲ 复制代码

```
import BaseModule from './baseModule';
import TableStructor, { ColumnType } from './structor';
export default class user extends BaseModule {
  public userName: string;

  public passowrd: string;

  public name: string;

  public age?: number;

  constructor(userName: string, passowrd: string,name: string,age?: number, id?: number) {
    super('Users', id);
    this.userName = userName;
    this.passowrd = passowrd;
    this.name = name;
    this.age = age;
  }
```

```
  // here, you should build your table structure.
  static GetTableStructor() {
    return new TableStructor(
      "Users",
      [
        { columnName: "id", columnType: ColumnType.Number,
          nullable: false, isPrimary: true, autoIncrement: true },
        { columnName: "userName", columnType: ColumnType.String },
        { columnName: "passowrd", columnType: ColumnType.String },
        { columnName: "name", columnType: ColumnType.String },
        { columnName: "age", columnType: ColumnType.Number, nullable: true },
      ],
      // if you need to add a constraint, then here is how you could do it as an example
      //[
        //{ contraintTableName: "Person", contraintColumnName: "id", columnName: "person_Id" }
      //]
    )
  }
}
```

我们的下一步是构建我们的存储库，我们将`expo-sqlite` 用作我们的`database`.

我们需要这个存储库做的是以下内容：

1. 设置数据库。
2. 查找模块是否已更改并将这些更改应用于`database`， 例如，在从模块添加和删除新属性时。
3. 保存一个项目并返回最后添加的项目。
4. `Where`方法来搜索您的`database` 返回可用项目的查询。
5. 删除项目。

JavaScript 缩小▲ 复制代码

```
  // note: that single, toType are global extension I use, to make things simpler
  import * as SQLite from 'expo-sqlite';
  export default class Repository {
  static dbIni: Boolean;
  databaseName: string;
  database?: SQLite.WebSQLDatabase;
  constructor() {
   this.databaseName = 'mydb.db';
  }

  createConnection = (force?: boolean) => {
      if (!this.database || force)
           this.database = SQLite.openDatabase(this.databaseName);
      return this.database;
  };

  // this is so we know which column the Table is in database container
  allowedKeys = (tableName: string) => {
  return new Promise((resolve, reject) => {
    this.createConnection().transaction(
      (x) =>
        x.executeSql(
          `PRAGMA table_info(${tableName})`,
          undefined,
          (trans, data) => {
            var keys = [] as string[];
            for (var i = 0; i < data.rows.length; i++) {
              if (data.rows.item(i).name != 'id')
                keys.push(data.rows.item(i).name);
            }
            resolve(keys);
          },
        ),
      (error) => {
        reject(error);
      },
    );
```

```
    }) as Promise<string[]>;
  };

  private find = (query: string, args?: any[], tableName?: TableNames) => {
    var tables = [Users.GetTableStructor()]
    return new Promise((resolve, reject) => {
      this.createConnection().transaction(
        async (x) => {
          console.log('Executing Find..');
          x.executeSql(
            query,
            args,
            async (trans, data) => {
              var booleanColumns =
                  tables.find(x => x.tableName == tableName)?.columns.filter
              (x => x.columnType == ColumnType.Boolean);
              console.log('query executed:' + query);
              const translateKeys = (item: any) => {
                if (!item || !booleanColumns || booleanColumns.length <= 0)
                  return item;
                booleanColumns.forEach(column => {

                  if (item[column.columnName] != undefined &&
                      item[column.columnName] != null) {
                    if (item[column.columnName] === 0 ||
                    item[column.columnName] === "0" || item[column.columnName] === false)
                      item[column.columnName] = false;
                    else item[column.columnName] = true;
                  }
                })
                return item;
              }
              var items = [] as BaseModule[];
              for (var i = 0; i < data.rows.length; i++) {
                var item = data.rows.item(i);
                items.push(translateKeys(item));
              }
              resolve(items);
            },
            (_ts, error) => {
              console.log('Could not execute query:' + query);
              console.log(error);
              reject(error);
              return false;
            },
          );
        },
        (error) => {
          console.log('Could not execute query:' + query);
          console.log(error);
          reject(error);
        },
      );
    }) as Promise<basemodule[]>;
  };

  async where<t>(tableName: TableNames, query?: any | T) {
    var q = `SELECT * FROM ${tableName} ${query ? 'WHERE ' : ''}`;
    var values = [] as any[];
    if (query && Object.keys(query).length > 0) {
      Object.keys(query).forEach((x, i) => {
        var start = x.startsWith('$') ?
        x.substring(0, x.indexOf('-')).replace('-', '') : undefined;
        if (!start) {
          q += x + '=? ' + (i < Object.keys(query).length - 1 ? 'AND ' : '');
          values.push(query[x]);
        } else {
          if (start == '$in') {
            var v = query[x] as [];
```

```
          q += x.replace("$in-", "") + ' IN (';
          v.forEach((item, index) => {
            q += '?' + (index < v.length - 1 ? ', ' : '');
            values.push(item);
          });
        }
        q += ') ' + (i < Object.keys(query).length - 1 ? 'AND ' : '');
      }
    });
  }
  return (
    (await this.find(q, values, tableName))
      .map((x) => {
        x.tableName = tableName;
        return x;
      })
      .toType<t>() ?? []
  );
}

  // get the last inserted or updated item.
  async selectLastRecord<t>(item: BaseModule) {
  console.log('Executing SelectLastRecord... ');
  if (!item.tableName) {
    console.log('Table name cant be empty for:');
    console.log(item);
    return;
  }
  return (
    await this.find(!item.id || item.id <= 0 ? `SELECT * FROM ${item.tableName} _
    ORDER BY id DESC LIMIT 1;` : `SELECT * FROM ${item.tableName} WHERE id=?;`,
    item.id && item.id > 0 ? [item.id] : undefined, item.tableName)
  ).toType<t>().map((x: any) => { x.tableName = item.tableName; return x; }).single<t>();
}

delete = async (item: BaseModule, tableName?: TableNames) => {
  tableName = item.tableName ?? tableName;
  var q = `DELETE FROM ${tableName} WHERE id=?`;
  await this.execute(q, [item.id]);
};

// this method will update and insert depending on Id and parameter insertOnly
public save<t>(item?: BaseModule, insertOnly?: Boolean, tableName?: TableNames) {
  if (!item) return undefined;
  if (!item.tableName || item.tableName.length <= 3)
      item.tableName = tableName ?? "ApplicationSettings";
  return new Promise(async (resolve, reject) => {
    try {
      console.log('Executing Save...');
      var items = await this.where<basemodule>(item.tableName, { id: item.id });
      var keys = (await this.allowedKeys(item.tableName)).filter((x) =>
                  Object.keys(item).includes(x));

      let query = '';
      let args = [] as any[];
      if (items.length > 0) {
        if (insertOnly) return;
        query = `UPDATE ${item.tableName} SET `;
        keys.forEach((k, i) => {
          query += ` ${k}=? ` + (i < keys.length - 1 ? ',' : '');
        });
        query += ' WHERE id=?';
      } else {
        query = `INSERT INTO ${item.tableName} (`;
        keys.forEach((k, i) => {
          query += k + (i < keys.length - 1 ? ',' : '');
        });
        query += ') values(';
        keys.forEach((k, i) => {
```

```
          query += '?' + (i < keys.length - 1 ? ',' : '');
        });
        query += ')';
      }
      keys.forEach((k: string, i) => {
        args.push((item as any)[k] ?? null);
      });
      if (items.length > 0) args.push(item.id);

      await this.execute(query, args);
      resolve(((await this.selectLastRecord<t>(item)) ?? item) as T);
    } catch (error) {
      console.log(error);
      reject(error);
    }
  }) as Promise<t>;
}

// this is a simple execute SQL query.
private timeout?: any;
private execute = async (query: string, args?: any[]) => {
  return new Promise((resolve, reject) => {
    this.createConnection().transaction(
      (tx) => {
        clearTimeout(this.timeout)
        this.timeout = setTimeout(() => {
          console.log("timed out")
          reject("Query Timeout");
        }, 2000);
        console.log('Execute Query:' + query);
        tx.executeSql(
          query,
          args,
          (tx, results) => {
            console.log('Statment has been executed....' + query);
            clearTimeout(this.timeout)
            resolve(true);
          },
          (_ts, error) => {
            console.log('Could not execute query');
            console.log(args);
            console.log(error);
            reject(error);
            clearTimeout(this.timeout)
            return false;
          },
        );
      },
      (error) => {
        console.log('db executing statement, has been terminated');
        console.log(args);
        console.log(error);
        reject(error);
        clearTimeout(this.timeout)
        throw 'db executing statement, has been terminated';
      },
    );
  });
};

// validate of the gevin module differs from the database table
private validate = async (item: TablaStructor) => {
  var appSettingsKeys = await this.allowedKeys(item.tableName);
  return appSettingsKeys.filter(x => x != "id").length != item.columns.filter
  (x => x.columnName != "id").length || item.columns.filter(x => x.columnName != "id" &&
  !appSettingsKeys.find(a => a == x.columnName)).length > 0;
}

private cloneItem<t>(item: any, appended: any, ignoreKeys?: string[]) {
```

```
  var newItem = {} as any;
  if (appended === undefined)
    return item;
  Object.keys(item).forEach((x) => {
    if (Object.keys(appended).find((f) => f == x) &&
    appended[x] !== undefined && (!ignoreKeys || !ignoreKeys.includes(x)))
      newItem[x] = appended[x];
    else newItem[x] = item[x];
  });
  return (newItem as T);
}

setUpDataBase = async (forceCheck?: boolean) => {
  if (!Repository.dbIni || forceCheck) {
    const dbType = (columnType: ColumnType) => {
      if (columnType == ColumnType.Boolean || columnType == ColumnType.Number)
        return "INTEGER";
      if (columnType == ColumnType.Decimal)
        return "REAL";
      return "TEXT";
    }
    console.log(`dbIni= ${Repository.dbIni}`);
    console.log(`forceCheck= ${forceCheck}`);
    console.log("initialize database table setup");
    this.createConnection(true); // make sure to close all transaction.
    var tables =[User.GetTableStructor()] // all your table in the right orders
    await tables.asyncForeach(async (table) => {
      var query = `CREATE TABLE if not exists ${table.tableName} (`;
      table.columns.forEach((col, index) => {
        query += `${col.columnName} ${dbType(col.columnType)} ${!col.nullable ?
        "NOT NULL" : ""} ${col.isPrimary ? "UNIQUE" : ""},\n`
      });
      table.columns.filter(x => x.isPrimary === true).forEach((col, index) => {
        query += `PRIMARY KEY(${col.columnName} ${col.autoIncrement === true ?
        "AUTOINCREMENT" : ""})` + (index < table.columns.filter
        (x => x.isPrimary === true).length - 1 ? ",\n" : "\n");
      });

      if (table.constraints && table.constraints.length > 0) {
        query += ",";
        table.constraints.forEach((col, index) => {
          query += `CONSTRAINT "fk_${col.columnName}" FOREIGN KEY(${col.columnName})
          REFERENCES ${col.contraintTableName}(${col.contraintColumnName})` +
          (index < (table.constraints?.length ?? 0) - 1 ? ",\n" : "\n");
        });
      }

      query += ");";
      await this.execute(query);
    })
  }
}

// this is where you will find all your giving module changes
// and apply it to the database
  newDataBaseStructure = async () => {
  var items = [] as {tableName:TableNames, items:BaseModule[]}[];

  if (await this.validate(User.GetTableStructor())) {
    console.info("Structor changes has been found in User.");
    var users = await this.where<user>("Users");
    if (users.length) {
      items.push({ tableName: "Users", items: users.map(x => this.cloneItem
      (new User(x.userName, x.password, x.name, x.age), x, ["id", "tableName"])) });
    }
    await this.execute(`DROP TABLE if exists Users`);
  }

  // Insert the old data to the new table and apply your module change
```

```
    if (items.length > 0) {
      await this.setUpDataBase(true);
      this.createConnection(true); // make sure to close all transaction.
      await items.reverse().asyncForeach(async x => {
        console.info(`Inserting items into ${x.tableName}`);
        await x.items.asyncForeach(async item => {
          var savedItem = await this.save(item, undefined, x.tableName);
        })
      });
      this.createConnection(true); // make sure to close all transaction.
      return true;
    }
    }
}
```

嗯，就是这样！

现在我们应该能够使查询变得非常简单。

见下文：

JavaScript 复制代码

```
var rep= new Repository();
// When your app starts, run this
await rep.setUpDataBase();
await rep.newDataBaseStructure();

// thereafter, run your command.
var users = await rep.where<User>("Users", {age: 20});
// Or
 var users = await rep.where<User>("Users",
 {"$in-age": [20,30, 25], userName: "testUser"});

users[0].age = 35;
var changedUser = await rep.save<User>(users[0]);
```

兴趣点

这是构建 ORM 并完全控制数据库和查询的设置方式的一种非常简单的方法。

历史

- 2021 年 9 月 4 日：初始版本

执照

分享

关于作者

艾伦·托马

软件开发人员（高级）

瑞典

手表该会员

没有提供传记

评论和讨论

添加评论或问题

电子邮件提醒 Search Comments

-- 本论坛暂无消息 --

永久链接
广告
隐私
Cookie
使用条款

布局：固定 | 体液