Realm Javascript

Realm 介绍

Realm，可作为手机端数据库，可以让开发人员高效的实现应用的model层

- SQLite, Core Data 和 ORMs的替代方案
- 简单优雅的是嵌入式数据库
- 高效、快速
- 支持 Android、iOS、React Naticve、Xamarin

RN Demo Code

```
// 定义对象模型
class Car {}
Car.schema = {
    name: "Car",
    properties: {
        make: "string",
        model: "string",
        miles: 'int'
    }
};
class Person {}
Person.schema = {
    name: "Person",
    properties: {
        name: {
            type: "string"
        },
        cars: {
            type: 'list',
            objectType: 'Car'
        },
        picture: {
            type: 'data',
            optional: true
        }
    }
}
// 让Realm支持我们的对象模型
let realm = new Relam({schema: [Car, Person]});
// 创建Realm对象，然后写入到本地local storage
realm.write(() => {
    let myCar = realm.create('Car', {
        make: 'Honda',
        model: 'Civic',
        miles: 1001
    });
```

```
    // 更新一个字段的value
    myCar.miles += 20;
})
// 查询所有里程数高于1000的Car;
let cars = realm.objects('Car').filtered('miles > 1000');
// 将会输出包含一个Car对象的列表，length为1
console.log(cars.length); // 输出 1
// 添加另外一个Car对象
realm.write(() => {
    let anotherCar = realm.create("Car", {
        make: 'Ford',
        model: 'Focus',
        miles: 2000
    })
});
// 之前的cars查询结果会实时的更新
console.log(cars.length); // 输出 2
```

React-Native 项目中安装

- 使用要求
    - 确认RN开发环境已搭建，可运行RN App
    - React-Native 0.20.0 以上版本
- 安装

```
yarn add realm
```

- link RN项目和realm 原生模块

```
react-native link realm
```

- Component 中使用

```
class RealmDemo extends Component {
    constructor(props) {
        super(props);
        this.realm = new Realm({
            name: "Dog",
            properties: {
                name: "string",
                size: {
                    type: 'int',
                }
            }
        })
```

```
    }
    componentDidMount() {
        this.realm.write(() => {
            this.realm.create('Dog', {
                name: "Rex",
                size: 50
            })
        })
    }
    render() {
        <View>
            <Text>
                Count of Dogs in Realm: {this.realm.objects('Dog').length}
            </Text>
        </View>
    }
}
```

示例代码

```
git clone https://github.com/realm/realm-js.git
cd realm-js
git submodule update --init --recursive
```

- Android 需要安装NDK，并且设置了ANDROID_NDK环境变量

```
export ANDROID_NDK=/usr/local/Cellar/android-ndk/r10e
```

Realm 模型

Realm 数据模型通过传入Realm初始函数的Schema信息定义。一个对象的Schema包含：

- 对象 name
- 若干属性 property
- - property 包含
- - - name
- - - type
- - - objectType 用在list类型属性中的对象类型
- - - optional 指定该属性可选
- - - default 属性默认值

直接定义Schema，然后初始化Realm

```
import Realm from 'realm';
```

```
const CarSchema = {
    name: "Car",
    properties: {
        make: "string",
        model: "string",
        miles: {
            type: 'int',
            default: 0
        }
    }
}
const PersonSchema = {
    name: "Person",
    properties: {
        name: "string",
        birthday: 'date',
        cars: {
            type: 'list',
            objectType: 'Car'
        },
        picture: {
            type: 'data',
            optional: true
        }
    }
}
// 使用Car和Person数据模型初始化Realm
let realm = new Realm({schema: [CarSchema, PersonSchema]});
```

对象继承已有的类

在Object constructor上定义schema，并在创建Realm对象时，传入Object constructor

```
class Person {
    get ageSecond() {
        return Math.floor(Date().now() - this.birthday.getTime());
    }
    get age() {
        return ageSeconds() / 31557600000;
    }
}
Person.schema = PersonSchema;
// ** 注意 ** 这里传入的是 "Person"构造函数
let realm = new Realm({schema: [CarSchema, Person]})
```

创建和fetch 对象

```
realm.write(() => {
    let car = realm.create("Car", {
        make: "Honda",
        model: "Civic",
        miles: 750
    })
})
console.log(`Car type is ${car.make} ${car.model}`);
car.miles = 1500;
```

Model 支持的数据类型

- bool 对应JS Boolean
- int, float, double 对用JS Number Object, 'int' and 'double' 已64bit存储， ‘float’以32bit存储
- string : JS String
- data: JS ArrayBuffer（类型化数组，JavaScript操作二进制数据的一个接口，最初为了满足JavaScript与显卡之间大量的、实时的数据交换，他们之间的数据通信必须是二进制的）
- date: JS Date

```
// 属性只指定类型时，可直接在属性名称后跟随类型
const CarSchema = {
    name: "Car",
    properties: {
        make: {
            type: "string"
        },
        model: 'string'
    }
}
```

关系

- To-One Relationship
  使用 一个 schema name 属性值，指定其他 schema 属性类型

```
const PersonSchema = {
    name: "Person",
    properties: {
        car: {
            type: 'Car',
            van: 'Car'
        }
    }
}
```

注意： 使用对象properties时，你需要确定所有引用的对象类型，显示在引用者之前

```
// CarSchema 需要在列表前方，因为PersonSchema 中存在类型为Car的property;
let realm = new Realm({schema: [CarSchema, PersonSchema]});
```

访问嵌套对性爱那个属性

```
realm.write(() => {
    var nameString = person.car.name;
    person.car.miles = 10000;
    // 创建一个Car ，赋值有效的JSON对象给 van
    person.van = {make: "ford", model: "Transit"};
    // car 与 van 指向同一个 Car 实例
    person.car = person.van;
})
```

- To-Many Relationship
  必须指定属性类型为 list ,同时配合使用 objectType，指定列表内数据类型

```
const PersonSchema = {
    name: "Person",
    properties: {
        cars: {
            type: 'list',
            objectType: 'Car'
        }
    }
}
```

List properties 拥有和JavaScript array 相似的方法。不同包括：

1. List property 任何变化会实时持久化到底层Realm
2. List 属于底层对象，只能通过对象属性获得实例，不能手动创建

```
let carList = person.cars;
// 添加Car 到 list
realm.write(() => {
    carList.push({
        make: "Honda",
        model: "Accord",
        miles: 100
    });
    carList.push({
        make: "Toyota",
```

```
        model: "Prius",
        miles: 200
    });
});
// 通过array index 访问
let secondCar = carList[1].model;
```

- Inverse Relationships
  使用linking objects properties,可以通过指定的property获得所有对象;

```
const PersonSchema = {
    name: "Person",
    properties: {
        dogs: {
            type: 'list',
            objectType: 'Dog'
        }
    }
};
const DogSchema = {
    name: "Dog",
    properties: {
        owners: {
            type: 'linkingObjects,
            objectType: 'Person',
            property: 'dogs'
        }
    }
};
```

提示：linkingObjects property 可以指向 List property (to-many), 也可以指向Object property (to-one)

```
const ShipSchema = {
    name: 'Ship',
    properties: {
        captain: 'Captain'
    }
}
const CaptainSchema = {
    name: 'Captain',
    properties: {
        ships: {
            type: 'linkingObjects',
            objectType: 'Ship',
            property: 'captain'
        }
    }
}
```

使用linkingObjects属性事，返回Results 对象，所以quering和sorting是全部支持的。linkingObjects 属性不能直接set或修改，当一个事务提交时，它们会自动更新。
在Schema之外访问 linkingObjects,例如：你一已经打开了一个Realm文件，但是没有指定schema,在 Realm Functions callback, 你可以通过在一个对象示例上执行linkingObjects(objectType, property)获取 linkingObjects 属性

```
let captain = realm.objectForPrimaryKey('Captain', 1);
let ships = captain.linkingObjects('Ship', 'captain');
```

可选属性(Optional Properties)

通过在属性定义中指定 optional 符号， 属性可以声明为 optainl or non-optional。

```
const PersonSchema = {
    name: "Person",
    properties: {
        name: "string", // required property
        birthday: { // optional property
            type: 'data',
            optional: true
        },
        car: { // Object 属性总是 optional
            type: "Car"
        }
    }
}
let realm = new Realm({schema: [PersonSchema, CarSchema]});
realm.write(() => {
    // optional properties 可以被赋值为 null 或 undefined
    let charlie = realm.create('Person', {
        name: "Charlie",
        birthday: new Date(),
        car: null
    });
    // optional properties 能赋值为 null 或 undefined、non-null value
    charlie.birthday = undefined;
    charlie.car = {make: "Honda", model: "Accord", miles: 100000}
})
```

提示：

1. object properties 总是 optional
2. List properties 不能声明为 optional 或 set to null
3. List properties 可以set 或 初始化一个empty array 去清空它

属性默认值

在属性定义中通过defult符号指定默认值，使用默认值可以在创建对象时不去指定该属性。

```
const CarSchema = {
    name: "Car",
    properties: {
        make: 'string',
        model: 'string',
        drive: {
            type: 'string',
            default: 'fwd'
        },
        miles: {
            type: 'int',
            default: 0
        }
    }
};
realm.write(() => {
    // miles使用了默认值 0
    // 指定了drive，会覆盖默认值
    realm.create('Car', {make: 'Honda', model: 'accord', drive: 'awd'});
})
```

属性索引：Indexed Properties

属性定义时使用 indexed 符号，指定该属性被索引，支持的属性类型包括：int, string, bool

```
const BookSchema = {
    name: 'Bool',
    properties: {
        name: {
            type: 'string',
            indexed: true
        },
        price: 'float'
    }
}
```

对一个属性建立索引可以加快对该属性的查询，但同时牺牲了部分查询效率。

主键 Primary Keys

可以在对象模型的 string / int 属性上指定 primaryKey。声明主键允许对象可以被高效的查询和更新，增强每条记录的唯一性。对象主键一旦添加到Realm，主键就不能变了。

```
const BookSchema = {
    name: "Book",
    primaryKey: 'id',
    properties: {
        id: 'int',
        title: 'string',
        price: 'float'
    }
}
```

主键被自动索引了。

存数据 / Writes

使用create方法创建对象

```
let realm = new Realm({
    schema: [CarSchema]
});
try {
    realm.write(() => {
        realm.create('Car', {
            make: 'Hoda',
            model: 'Accord',
            derive: 'awd'
        });
    });
} catch(e) {
    console.log('Error on creation');
}
```

提示：write 方法中任意的异常或取消事务，示例中的try/catch块是不错的实践。

嵌套对象 / Nested Objects

```
let realm = new Realm({schema: [PersonSchema, CarSchema]});
realm.write(() => {
    realm.create('Person', {
        name: 'Joe',
        car: {
            make: 'Honda',
            model: 'Accord',
            drive: 'awd'
        }
    })
})
```

更新对象

在write 事务中，向对象属性赋值

```
realm.write(() => {
    car.miles = 100000;
})
```

- 创建并通过主键更新对象
  Realm 基于主键，智能的更新或创建 objects 。通过出入create 方法第三个参数 true

```
realm.write(() => {
    // 创建一个 Book 对象
    realm.create('Book', {
        id: 1,
        title: 'Recipes',
        price: 35
    });
    // 根据id 更新 对象的 price
    realm.create('Book', {
        id: 1,
        price: 55
    }, true);
});
```

- 删除对象 在 write 事务中，通过 delete 方法删除对象

```
realm.write(() => {
    // 创建 book 对象
    let book = realm.create('Bool', {
        id: 1,
        title: 'Recipes',
        price:36
    });
    // 删除 book 对象
    realm.delete(book);
    // 删除多个 books 对象, 通过传入 Results, list
    let allBooks = realm.objects('Book');
    realm.delete(allBooks);
})
```

Query

支持单一类型的查询，并对结果进行过滤、排序。所有的查询（包括queries、property 访问）都是懒查询，只有当对象和属性被访问时才会读取。允许你以高效的方式呈现大量的数据集合。

```
let dogs = realm.objects('Dog');
```

Filter

使用 filtered 方法，传入查询寻语句

```
let dogs = realm.objects('Dog');
let tanDogs = dogs.filtered('color = "tan" AND name BEGINSWITH "B"');
```

1. 比较操作：==, !=, >, >=, <, <= 适用于 数字属性
2. 比较操作：==, BEGINSWITH, ENDSWITH, CONTAINS 适用于 字符串属性
3. 大小写敏感：==[c], BEGINSWITH[c] 等
4. 嵌套对象：car.color == 'blue'

Sorting

```
let bondas = realm.objects('Car').filtered('make = "Honda"');
// 根据 miles 排序
let sortedHondas = hondas.sorted('miles')
```

自动更新 Results

Results 实例是实时对象，自动更新数据到底层，意味者Results从不需要re-fetched. 修改对象数据会立刻反映到相关的查询结果

```
let hondas =realm.objects('Car').filtered('make = "Honda"');
// 此时hondas.length = 0
realm.write(() => {
    realm.create('Car', {
        make: 'Honda',
        model: 'RSX'
    })
})
// 此时 bondas.length = 1
```

自动更新适用于所有 Results 实例，包括 objects, filtered, 和 sorted 方法。 自动更新使Realm快速和高效，是代码更小且更灵活。 例如：你的视图依赖查询结果，你可以存储 Results 在一个属性，然后访问它，不用为了数据同步而去 refetch 数据。
你可以订阅到 notifications ，从而知道数据什么时间被更新了，然后通知 UI 重新渲染。

限制结果

Realm 查询是懒执行的（lazy），执行分页操作是没有必要的。Realm 只会当它们直接被访问时才从 Results 中加载。
UI 相关或其他原因需要Results的一部分数据，可以使用 Results 的 slice方法

```
let cars = realm.objects('Cars');
// get first 5 Car objects
let firstCars =cars.slice(0, 5);
```