Модели в JavaScript

Одной из проблем перемещения состояния на сторону клиента является управление данными. Традиционно можно извлечь данные непосредственно из базы данных в процессе запроса страницы, вставляя результат непосредственно в страницу в результате взаимодействия сервера и клиента. Но управление данными в структурированных JavaScript-приложениях является совершенно другим процессом. Здесь нет модели запрос-ответ, и у вас нет доступа к переменным на стороне сервера. Вместо этого данные извлекаются удаленно и временно сохраняются на стороне клиента.

Хотя такой переход может создать определенные трудности, он принесет ряд преимуществ. Например, доступ к данным на стороне клиента происходит практически мгновенно, поскольку данные просто извлекаются из памяти. Это может действительно изменить интерфейс вашего приложения — реакция на любое действие с приложением будет мгновенной, что зачастую существенно улучшает пользовательские впечатления от работы с ним.

Тут следует подумать над тем, как выстроить архитектуру хранилища данных на стороне клиента. В этой области легко просчитаться и впасть в заблуждение, что характерно для малоопытных разработчиков, особенно если их приложения становятся крупнее. В данной главе будут рассмотрены лучшие способы организации такого перехода и даны некоторые рекомендованные схемы и инструкции.

MVC и организация пространства имен

Обеспечение в вашем приложении четких границ между представлениями, состоянием и данными играет основную роль в поддержании порядка и жизнеспособности его архитектуры. Если придерживаться модели MVC, управление данными осуществляется в моделях (буква «M» в MVC). Модели должны быть отделены от представлений и контроллеров. Любая логика, связанная с работой с данными и поведением, должна находиться в моделях и иметь правильную организацию пространства имен.

В JavaScript можно определить пространство имен функций и переменных, сделав их свойством объекта. Например:

```
var User = {
  records: [ //* .... */ ]
};
```

Для массива пользователей определено должное пространство имен под свойством User.records. Функции, связанные с пользователями, также могут быть заключены в пространство имен модели User. Например, у нас может быть функция fetchRemote() для извлечения пользовательских данных из сервера:

```
var User = {
  records: [],
  fetchRemote: function(){ /* .... */ }
};
```

Содержание всех свойств модели в пространстве имен гарантирует отсутствие каких-либо конфликтов и совместимость этой модели со схемой MVC. Это также оберегает ваш код от скатывания вниз к мешанине из функций и внешних (обратных) вызовов.

В организации пространства имен можно пойти еще дальше и хранить любые функции, относящиеся к экземплярам пользователей, в фактических объектах пользователей. Представим, что у нас есть функция destroy() для работы с записями пользователей. Она ссылается на конкретного пользователя, стало быть, она должна быть в экземпляре User:

```
var user = new User;
user.destroy()
```

Чтобы добиться этого, нам нужно сделать User классом, а не обычным объектом:

```
var User = function(atts){
  this.attributes = atts || {};
};
User.prototype.destroy = function(){
  /* ... **//
};
```

Любые функции и переменные, не относящиеся к конкретным пользователям, могут быть свойствами, непосредственно относящимися к объекту User:

```
User.fetchRemote = function(){
  /* ... */
};
```

Чтобы получить дополнительную информацию об организации пространства имен, посетите блог Питера Мишо (Peter Michaux), где можно прочитать статью на эту тему.

Создание ORM

Средства объектно-реляционного отображения (ORM) обычно используются в языках, отличающихся от JavaScript. И тем не менее это очень полезная технология для управления данными, а также отличный способ использования моделей в вашем JavaScript-приложении. Используя ORM, можно, к примеру, связать модель с удаленным сервером —любые изменения, внесенные в экземпляры модели, вызовут отправку в фоновом режиме Ajax-запросов к серверу. Или же можно связать экземпляр модели с элементом HTML —любые изменения, относящиеся к экземпляру, будут отражаться в представлении. Все это будет конкретизировано в примерах, а сейчас давайте посмотрим на создание специализированного ORM.

В конечном итоге ORM —это всего лишь объектный уровень, заключающий в себя некие данные. Обычно средства ORM используются для реферирования баз данных SQL, но в нашем случае ORM будет всего лишь реферированием типов данных JavaScript. Преимущество наличия этого дополнительного уровня состоит в том, что мы можем улучшить исходные данные расширенной функциональностью путем добавления наших собственных специализированных функций и свойств. Это позволит нам добавить такие ценные качества, как проверка допустимости, использование наблюдателей, обеспечение постоянства и использование серверных обратных вызовов, сохраняя при этом возможность повторного использования большого объема кода.

Прототипное наследование

Для создания нашего ORM мы собираемся воспользоваться функцией `Object.create()`, которая немного отличается от рассмотренных в главе 1 примеров, основанных на классах. Это позволит нам вместо функций-конструкторов и ключевого слова `new` использовать наследование, основанное на прототипах.

Функции `Object.create()` передается один аргумент, объект-прототип, а она возвращает новый объект с указанным объектом-прототипом. Иными словами, вы даете ей объект, а она возвращает новый объект, унаследованный от указанного вами объекта.

Функция `Object.create()` была недавно добавлена к ECMAScript, пятое издание, поэтому в некоторых браузерах, таких как IE, она не реализована. Но это не проблема, поскольку мы при необходимости запросто можем добавить ее поддержку:

```
if (typeof Object.create !== "function")
    Object.create = function(o) {
      function F() {}
      F.prototype = o;
      return new F();
    };
```

Показанный выше пример был позаимствован у Дугласа Крокфорда (Douglas Crockford) из его статьи «Prototypal Inheritance». Почитайте эту статью, если вы хотите глубже изучить основы прототипов и наследования в JavaScript.

Мы собираемся создать объект Model, который будет отвечать за создание новых моделей и экземпляров:

```
var Model = {
  inherited: function(){},
  created: function(){},

  prototype: {
    init: function(){}
  },

  create: function(){
    var object = Object.create(this);
    object.parent = this;
    object.prototype = object.fn = Object.create(this.prototype);

    object.created();
    this.inherited(object);
    return object;
  },

  init: function(){
    var instance = Object.create(this.prototype);
    instance.parent = this;
    instance.init.apply(instance, arguments);
    return instance;
  }
};
```

Если вы незнакомы с функцией Object.create(), этот код может показаться отпугивающим, поэтому давайте разберем его по частям. Функция create() возвращает новый объект, унаследованный у объекта Model, — мы воспользуемся этим для создания новых моделей. Функция init() возвращает новый объект, унаследованный у Model.prototype, — т. е. экземпляр объекта Model:

```
var Asset = Model.create();
var User  = Model.create();

var user = User.init();
```

Добавление свойств ORM

Теперь, если мы добавим свойства к Model, они будут доступны во всех унаследованных моделях:

```
// Добавление свойств объекта
jQuery.extend(Model, {
  find: function(){}
});
```

```
// Добавление свойств экземпляра
jQuery.extend(Model.prototype, {
  init: function(atts) {
    if (atts) this.load(atts);
  },

  load: function(attributes){
    for(var name in attributes)
      this[name] = attributes[name];
  }
});
```

Функция **jQuery.extend()** является всего лишь более коротким способом использования цикла for для самостоятельного копирования с перебором всех свойств, аналогично тому, что мы делали в функции **load()**. Теперь наши свойства объекта и экземпляра распространяются вниз на наши отдельные модели:

```
assertEqual( typeof Asset.find, "function" );
```

Фактически мы собираемся добавить множество свойств, поэтому мы можем также сделать **extend()** и **include()** частью объекта **Model**:

```
var Model = {
  /* ... фрагмент ... */

  extend: function(o){
    var extended = o.extended;
    jQuery.extend(this, o);
    if (extended) extended(this);
  },

  include: function(o){
    var included = o.included;
    jQuery.extend(this.prototype, o);
    if (included) included(this);
  }
};

// Добавление свойств объекта
Model.extend({
  find: function(){}
});

// Добавление свойств экземпляра
Model.include({
  init: function(atts) { /* ... */ },
  load: function(attributes){ /* .... */ }
});
```

Теперь мы можем создать новые активы (assets) и установить атрибуты:

```
var asset = Asset.init({name: "foo.png"});
```

Удерживание записей

Нам нужен способ удерживания записей, т. е. сохранения ссылки на созданные экземпляры, чтобы потом иметь к ним доступ. Мы сделаем это с использованием объекта **records**, установленного в отношении **Model**. При сохранении экземпляра мы будем добавлять его к этому объекту, а при удалении экземпляров мы будем удалять их из этого объекта:

```
// Объект сохраненных активов
Model.records = {};

Model.include({
  newRecord: true,

  create: function(){
    this.newRecord = false;
    this.parent.records[this.id] = this;
  },

  destroy: function(){
    delete this.parent.records[this.id];
  }
});
```

А что насчет обновления существующего экземпляра? Это нетрудно, достаточно обновить ссылку на объект:

```
Model.include({
  update: function(){
    this.parent.records[this.id] = this;
  }
});
```

Давайте создадим удобную функцию для сохранения экземпляра, чтобы нам не приходилось проверять, был ли экземпляр ранее сохранен и нужно ли его создавать:

```
// Сохранение объекта в хэше записей с сохранением ссылки на него
Model.include({
  save: function(){
    this.newRecord ? this.create() : this.update();
  }
});
```

А как насчет реализации функции **find()**, чтобы можно было искать активы по ID?

```
Model.extend({
  // Поиск по ID или выдача исключения
  find: function(id){
    return this.records[id] || throw("Неизвестная запись ");
  }
});
```

Теперь, преуспев в создании основного ORM, давайте испытаем его в работе:

```
var asset  = Asset.init();
asset.name = "same, same";
asset.id   = 1
asset.save();

var asset2  = Asset.init();
asset2.name = "but different";
asset2.id   = 2;
asset2.save();

assertEqual( Asset.find(1).name, "same, same" );
asset2.destroy();
```

Добавление поддержки ID

Пока при каждом сохранении записи нам нужно указывать ID самостоятельно. Нехорошо, но, к счастью, именно это мы можем автоматизировать. Сначала нам нужно найти способ генерирования идентификаторов —это можно сделать с помощью GUID-генератора (Globally Unique Identifier —глобально уникальный идентификатор). Итак, технически JavaScript не способен генерировать официальные полноценные 128-разрядные GUID-идентификаторы для использования в API, он может только генерировать псевдослучайные числа. Генерирование по-настоящему случайных GUID-идентификаторов является печально известной и весьма трудной задачей, и операционная система вычисляет их, используя MAC-адрес, позицию мыши и контрольные суммы BIOS, путем измерения электрического шума или радиоактивного распада, и даже с помощью ламп со вплывающими и тонущими в жидкости каплями вязкого вещества, постоянно меняющими свои формы! Но нам вполне хватит и возможностей имеющейся в JavaScript функции `Math.random()`.

Роберт Киффер (Robert Kieffer) написал легкий и совсем небольшой GUID-генератор, использующий `Math.random()` для генерации псевдослучайных GUID-идентификаторов. Он настолько прост, что его можно включить непосредственно в текст программы:

```
Math.guid = function(){
  return "xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx".replace(/[xy]/g,
function(c) {
    var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
    return v.toString(16);
  }).toUpperCase();
};
```

Теперь, когда у нас есть функция для генерации GUID-идентификаторов, ее интеграция в наш ORM не составит труда. Для этого нужно лишь изменить функцию create():

```
Model.extend({
  create: function(){
    if ( !!this.id ) this.id = Math.guid();
    this.newRecord = false;
    this.parent.records[this.id] = this;
  }
});
```

Теперь любые только что созданные записи имеют в качестве ID GUID-идентификаторы:

```
var asset = Asset.init();
asset.save();
asset.id //=> "54E52592-313E-4F8B-869B-58D61F00DC74"
```

Адресация ссылок

Если пристально приглядеться к нашему ORM, в нем можно заметить ошибку, связанную со ссылками. Мы не создаем копии экземпляров, когда они возвращаются функцией find() или когда мы их сохраняем, поэтому, если мы изменили любые свойства, они изменяются на исходном активе.

Проблема в том, что обновление активов нам требуется только при вызове функции update():

```
var asset = new Asset({name: "foo"});
asset.save();

// Утверждение соответствует истине
assertEqual( Asset.find(asset.id).name, "foo" );

// Давайте изменим свойство, но не будем вызывать update()
asset.name = "wem";

// Надо же! Это утверждение дает ложный результат, поскольку теперь актив
// называется "wem"
assertEqual( Asset.find(asset.id).name, "foo" );
```

Давайте исправим ситуацию, создав новый объект в процессе операции find(). Нам также нужно создать дубликат объекта при каждом создании или обновлении записи:

```
Asset.extend({
  find: function(id){
    var record = this.records[id];
    if ( !!record ) throw("Неизвестная запись");
    return record.dup();
  }
```

```
});

Asset.include({
  create: function(){
    this.newRecord = false;
    this.parent.records[this.id] = this.dup();
  },

  update: function(){
    this.parent.records[this.id] = this.dup();
  },

  dup: function(){
    return jQuery.extend(true, {}, this);
  }
});
```

Есть еще одна проблема: `Model.records` является объектом, который совместно используется каждой моделью:

```
assertEqual( Asset.records, Person.records );
```

Это имеет досадный побочный эффект смешивания всех записей:

```
var asset = Asset.init();
asset.save();

assert( asset in Person.records );
```

Решение состоит в создании нового объекта `records` при каждом создании новой модели. Функция `Model.created()` является внешним вызовом для создания нового объекта, поэтому мы можем создавать любые объекты, которые там определены для модели:

```
Model.extend({
  created: function(){
    this.records = {};
  }
});
```

Загрузка в данные

Если ваше веб-приложение не работает исключительно в пределах браузера, вам нужно загружать в него удаленные данные с сервера. Обычно поднабор данных загружается при запуске приложения, а дополнительные данные загружаются после взаимодействия пользователя с приложением. В зависимости от типа приложения и объема данных, вы можете загрузить все необходимое при загрузке начальной страницы. Это идеальный вариант, поскольку пользователям никогда не приходится ждать загрузки дополнительных данных. Но для многих приложений это нереально, поскольку у них слишком много данных и они не могут поместиться в памяти браузера.

Предварительная загрузка данных является ключевым условием для создания у пользователей ощущения гладкости и быстроты работы вашего приложения, и для сведения к минимуму любого времени ожидания. Но есть разумная грань между предварительной загрузкой действительно нужных данных и загрузкой излишних данных, которые никогда не будут использованы. Необходимо предсказать, данные какого сорта будут востребованы вашими пользователями (или воспользоваться показателями по результатам использования вашего приложения).

Почему бы при отображении списка, разбитого на страницы, не провести предварительную загрузку следующей страницы для обеспечения мгновенных переходов? Или, лучше того, просто вывести длинный список и автоматически загружать и вставлять данные по мере его прокрутки (шаблон бесконечной прокрутки). Чем меньше пользователь будет ждать, тем лучше.

При извлечении новых данных нужно убедиться в том, что пользовательский интерфейс не заблокирован. Выведите какой-нибудь индикатор загрузки, но при этом гарантируйте возможность пользования интерфейсом. Сценарии, блокирующие пользовательский интерфейс, должны быть сведены к минимуму или вообще не использоваться.

Данные должны быть представлены в составе кода начальной страницы или загружаться с помощью отдельных HTTP-запросов с применением Ajax или JSONP. Лично я хочу порекомендовать последние две технологии, поскольку включение большого объема данных в состав кода увеличивает размер страницы, в то время как загрузка с помощью параллельных запросов осуществляется быстрее. AJAX и JSON также позволяют вам кэшировать HTML-страницу, вместо динамического вывода по каждому запросу.

Включение данных в код страницы

Я не приветствую данный подход в силу причин, высказанных в предыдущем абзаце, но в определенных ситуациях он может пригодиться, особенно для загрузки очень небольших объемов данных. Преимуществом данной технологии является простота реализации.

Для этого нужно лишь вывести JSON-объект непосредственно в страницу. Например, в Ruby on Rails это делается следующим образом:

```
<script type="text/javascript">
  var User = {};
  User.records = <%= raw @users.to_json %>;
</script>
```

ERB-теги используются для вывода JSON-интерпретации данных пользователя. Метод `raw` используется просто для приостановки нейтрализации JSON. При выводе страницы HTML выглядит следующим образом:

```
<script type="text/javascript">
  var User = {};
  User.records = [{"first_name": "Alex"}];
</script>
```

JavaScript может просто обработать JSON, поскольку этот формат имеет такую же структуру, что и объект JavaScript.