[Обучение Ext JS] :: Часть 12. Все дело в данных

Мар 18 2010**Удаленая фильтрация: почему и как**

Клиенская фильтрация отличная вещь, снижает количество запросов к серверу. Но иногда все же, наши записи слишком большие чтобы загружать ими кеш браузера. Отличным примером такого рода является разбиение таблицы. Много раз мы будем брать только по 25 записей за раз. В таком случае клиентская фильтрация хоть и работает, но медленно, поэтому фильтр лучше применить ко всем данным.
Сортировка данных по удаленному запросу довольно простая штука. Нам надо всего-то установить свойство хранилища remoteSort на true. Итак, если наше хранилище присоединено к объекту таблицы, щелчок по названию столбца с целью сортировки моментально отправит значение в своем **AJAX**-запросе.
Фильтровать данные по удаленному запросу несколько труднее. Изначально нам надо отправить параметры через событие load хранилища, и обработать эти аргументы серверным методом.
Итак, первое что нам надо сделать, это немного серверного кода для обработк наших фильтров и сортировки. Мы вернемся к нашему компоненту ColdFusion для того чтобы добавить новый метод:
Example 3: Chapter_12\Chapter12Example.cfc

Исходный код примера:

Исходный код примера:

```
<!---
/  METHOD: getDirectoryContents
/
/  @param startPath:string
/  @param recurse:boolean (optional)
/  @param fileFilter:string (optional)
/  @param dirFilter:string (optional - File|Dir)
/  @param sortField:string (optional -
   NAME|SIZE|TYPE|DATELASTMODIFIED|ATTRIBUTES|MODE|DIRECTORY)
/ @param sortDirection:string (option - ASC|DESC
                         [defaults to ASC])
/ @return retQ:query
--->
<cffunction name="getDirectoryContents" access="remote"
                 output="false" returntype="query">
  <cfargument name="startPath" required="true" type="string" />
  <cfargument name="recurse" required="false" type="boolean"
                              default="false" />
  <cfargument name="sortDirection" required="false" type="string"
                               default="ASC" />
  <!--- Set some function local variables --->
  <cfset var q = "" />
  <cfset var retQ = "" />
  <cfset var attrArgs = {} />
  <cfset var ourDir = ExpandPath(ARGUMENTS.startPath) />
  <!--- Create some lists of valid arguments --->
  <cfset var filterList = "File,Dir" />
  <cfset var sortDirList = "ASC,DESC" />
  <cfset var columnList =
  "NAME,SIZE,TYPE,DATELASTMODIFIED,ATTRIBUTES,MODE,DIRECTORY" />
  <cftry>
   <cfset attrArgs.recurse = ARGUMENTS.recurse />
   <!--- Verify the directory exists before continuing --->
   <cfif DirectoryExists(ourDir)>
    <cfset attrArgs.directory = ourDir />
   <cfelse>
    <cfthrow type="Custom" errorcode="Our_Custom_Error" message="The
directory you are trying to reach does not exist." />
   </cfif>
   <!--- Conditionally apply some optional filtering and sorting
   --->
   <cfif IsDefined("ARGUMENTS.fileFilter")>
    <cfset attrArgs.filter = ARGUMENTS.fileFilter />
   </cfif>
   <cfif IsDefined("ARGUMENTS.sortField")>
    <cfif ListFindNoCase(columnList,ARGUMENTS.sortField)>
     <cfset attrArgs.sort = ARGUMENTS.sortField & " " &
       ARGUMENTS.sortDirection />
    <cfelse>
     <cfthrow type="custom" errorcode="Our_Custom_Error"
      message="You have chosen an invalid sort field.
      Please use one of the following: " & columnList />
    </cfif>
   </cfif>
   <cfdirectory action="list" name="q"
    attributeCollection="#attrArgs#" />
   <!--- If there are files and/or folders,
   and you want to sort by TYPE --->
   <cfif q.recordcount and IsDefined("ARGUMENTS.dirFilter")>
    <cfif ListFindNoCase(filterList,ARGUMENTS.dirFilter)>
     <cfquery name="retQ" dbtype="query">
     SELECT #columnList#
       FROM q
       WHERE TYPE = <cfqueryparam cfsqltype=" cf_sql_varchar"
            value="#ARGUMENTS.dirFilter#" maxlength="4" />
     </cfquery>
    <cfelse>
     <cfthrow type="Custom" errorcode="Our_Custom_Error"
       message="You have passed an invalid dirFilter.
       The only accepted values are File and Dir." />
    </cfif>
   <cfelse>
    <cfset retQ = q />
    </cfif>
   <cfcatch type="any">
    <!--- Place Error Handler Here --->
   </cfcatch>
  </cftry>
  <cfreturn retQ />
</cffunction>
```

Это может выглядеть сложно, но на самом деле это не так. И снова, наша задача не изучение ColdFusion, а хотя бы основное понимание того, что делает серверный процесс. В этом коде содержится метод, берущий опциональные параметры, имеющие отношение к сортировке и фильтрации, через HTTP POST. Мы используем тот же самый тег cfdirectory для запроса к файловой системе за списком файлов и папок. Разница здесь только в том, что мы условно применяем к тегу дополнительные атрибуты, таким образом, мы можем фильтровать по особому расширению файла или сортировать по определенному столбцу запроса. У нас также есть оператор Query-of-Query для запросов нашего вернувшегося набора записей если мы хотим пойти дальше и применить отфильтровать их по записи TYPE, что не является встроенной по умолчанию функцией. Наконец у нас есть собственная обработка ошибок, для того чтобы убедиться в том, что методу отправляются проверенные аргументы.
Мы внесем некоторые изменения в предыдущий скрипт хранилища. Сначала нам понадобится несколько методов, которые можно вызвать удаленно с целью отфильтровать наш набор записей:

Исходный код примера:

Исходный код примера:

```
filterStoreByType = function (type){
  ourStore.load({dirFilter:type});
}
filterStoreByFileType = function (fileType){
  ourStore.load({fileFilter:fileType});
}
clearFilters = function (){
  ourStore.baseParams = new cloneConfig(initialBaseParams);
  ourStore.load();
}
```

У нас есть методы для того, чтобы профильтровать наше хранилище по TYPE, для фильтровки по расширению файлов и для снятия всех фильтров. Значения отправляемые этому методу преобразованы названия, соответствующие удаленному аргументу метода. Слушатель нашего хранилища, beforeload, автоматически применяет эти аргументы к baseParams для осуществления AJAX-вызова к серверу. Важно помнить, что каждый добавленный параметр остается в baseParams до тех пор, пока фильтры не будут сняты.
Также важно отметить, что метод load() может принимать четыре аргумента: params, callback (метод для выполнения загрузки), scope и add (добавить загрузку в уже существующий набор данных). В использованном выше формате, объект использованный в качестве аргумента для load() является аргументом params, просто потому что ни один другой не подошел. Если вы используете все аргументы (или первые три), им надо быть в пределах объекта.

```
ourStore.load({{dirFilter:type},someMethodToCall,this,false});
```

Метод clearFilter() не работает с удаленной фильтрацией, так что нам понадобится способ вернуть изначальные baseParams когда нам надо будет снять фильтры Для этого мы сначала извлекаем наши настройки baseParams:

```
var initialBaseParams = {
   method: 'getDirectoryContents',
   returnFormat: 'JSON',
   queryFormat: 'column',
   startPath: '/testdocs/'
};
```

Затем нам понадобится способ клонировать настройки в наших настройках хранилища. Если мы отправили initialBaseParams непосредственно в опции настройки baseParams, а затем отфильтровали наш набор данных, фильтр будет добавлен в переменные initialBaseParams, так как переменные передаются по ссылке. Поскольку мы хотим иметь возможность восстановить начало нашего baseParams, нам понадобится клонировать объект initialBaseParams. Клон получит назначение в качестве опции настройки baseParams. Фильтры не задевают наш объект, и мы можем убрать их применив clearFilter().
Для этого нужен простой метод клонирования объектов JavaScript:

Исходный код примера:

Исходный код примера:

```
cloneConfig = function (config) {
   for (i in config) {
      if (typeof config[i] == 'object') {
          this[i] = new cloneConfig(config[i]);
      }
      else
          this[i] = config[i];
   }
}
```

Затем мы можем изменить атрибут наших baseParams в настройках нашего хранилища:

```
baseParams: new cloneConfig(initialBaseParams),
```

Мы использовали ту же амую функцию в методе clearFilters(), для того чтобы сбросить baseParams на начальные настройки. Вот так выглядит теперь весь скрипт:
Example 4: chapter12_04.js

Исходный код примера:

Исходный код примера:

```
cloneConfig = function (config) {
   for (i in config) {
      if (typeof config[i] == 'object') {
         this[i] = new cloneConfig(config[i]);
      }
        else
           this[i] = config[i];
    }
}
Ext.onReady(function(){
  var recordModel = [
    {name:'file_name',mapping:'NAME'},
    {name:'file_size',mapping:'SIZE'},
    {name:'type',mapping:'TYPE'},
    {name:'lastmod',mapping:'DATELASTMODIFIED'},
    {name:'file_attributes',mapping:'ATTRIBUTES'},
    {name:'mode',mapping:'MODE'},
    {name:'directory',mapping:'DIRECTORY'}
  ];
  var ourReader = new Ext.data.CFJsonReader({id:'NAME',root:'DATA'},
                                   recordModel);
  var initialBaseParams = {
    method: 'getDirectoryContents',
    returnFormat: 'JSON',
    queryFormat: 'column',
    startPath: '/testdocs/'
  };
  var ourStore = new Ext.data.Store({
    url:'Chapter12Example.cfc',
    baseParams: new cloneConfig(initialBaseParams),
    reader: ourReader,
    fields: recordModel,
    listeners:{
     beforeload:{
       fn: function(store, options){
         for(var i in options){
          if(options[i].length > 0){
            store.baseParams[i] = options[i];
          }
         }
       },
       scope:this
     },
     load: {
       fn: function(store,records,options){
         console.log(records);
       }
     },
     scope:this
    }
  });
  ourStore.load({recurse:true});
  filterStoreByType = function (type){
    ourStore.load({dirFilter:type});
  }
  filterStoreByFileType = function (fileType){
    ourStore.load({fileFilter:fileType});
  }
  clearFilters = function (){
    ourStore.baseParams = new cloneConfig(initialBaseParams);
    ourStore.load();
  }
});
```

Для проверки сделанных нами изменений можно разместить несколько ссылок на нашу страницу и вызвать созданные методы, отслеживая их работу через Firebug.
Example 4: ch12ex4.html

```
<div id="chap12_ex04">
  <a onclick="filterStoreByType('File')"
   href="javascript:void(0)">Filter by 'File's</a><br />
  <a onclick="filterStoreByFileType('*.doc')"
   href="javascript:void(0)">Filter by '.doc File's</a><br />
  <a onclick="clearFilters()" href="javascript:void(0)">
                              Clear Filters</a><br />
</div>
```

После загрузки страницы мы увидим консоль отображающую стартовую загрузку набора данных в хранилище. Нажав на первую ссылку мы удалим все записи директорий при помощи фильтров. Вторая ссылка идет дальше и отсеивает все файлы, кроме имеющих расширение .doc. Последняя ссылка сбрасывает фильтры на исходный baseParams и перезагружает начальный recordset.

Работаем с изменениями в Recordset

Одна из хороших вещей в **Ext JS** заключается в управлении изменениями в хранилищах данных. Наши приложения могут атаковать изменяющиеся записи самыми разными способами, от редактируемых теблиц данных до простых форм, но совершение изменений значит, что-то только в том случае, если это совершается по нашей указке. Мы можем изменить только отображение, но нам определенно хочется отправить изменения на сервер
Один из самых простых способов сделать это, применить обновление слушателя событий к нашему объекту хранилища. Мы применили два слушателя в прошлом: beforeload и load. Теперь давайте применим обновление слушателя в нашем скрипте.

Исходный код примера:

Исходный код примера:

```
listeners:{
  beforeload:{
    fn: function(store, options){
      for(var i in options){
       if(options[i].length > 0){
         store.baseParams[i] = options[i];
       }
      }
    },
    scope:this
  },
  load: {
    fn: function(store, records, options){
      console.log(records);
    },
    scope: this
  },
  update: {
    fn: function(store, record, operation){
      switch (operation){
      case Ext.record.EDIT:
        // Do something with the edited record
        break;
       case Ext.record.REJECT:
        // Do something with the rejected record
        break;
       case Ext.record.COMMIT:
       // Do something with the committed record
        break;
      }
    },
    scope:this
  }
}
```

Когда запись обновлена, событие запускается в хранилище, отправляя несколько объектов в событие. Первое это само хранилище. Второе, это обновляемая запись. Последний объект, это состояние обновляемой записи. Мы можем добавить код в блок действия для состояния Ext.record.EDIT для автоматической отправки каждого изменения серверу чтобы немедленно проверить записи.
Еще мы можем обратится к состоянию Ext.record.COMMIT. Иногда лучше позволить пользователю совершать много маленьких изменений, а потом отправлять их все скопом.

```
ourStore.commitChanges();
```

Это соберет все измененные записи, пометит используя Ext.record.COMMIT, а затем запустится событие и повлияет на каждую запись. Наше последнее состояние операции идеально подходит для обработки ситуации, для которой можно добавить дополнительное клиенское подтверждение или **AJAX**-подтверждение или все ради чего только можно вызвать наш процесс.
Состояние Ext.record.REJECT в основном устанавливается непосредственно в хранилище данных, посредством чего хранилище отвергает все изменения сделанные в записях и будет возвращать поля записей в их изначальное (или последенее сохраненное) состояние. Это может случится если в поле было передано значение неправильного типа данных.

Многие объекты используют Store (хранилище)

Красота объекта Store заключается в его многофункциональности. В **Ext JS** существует очень много приложений, которые могут использовать хранилище как часть их настройки, в большинчтве случаев автоматически помечая данные.

Store в комбинированом окне

Например объект комбинированного окна может принимать хранилище или любой из его подклассов как нечто, предоставляющее данные для значений:

Исходный код примера:

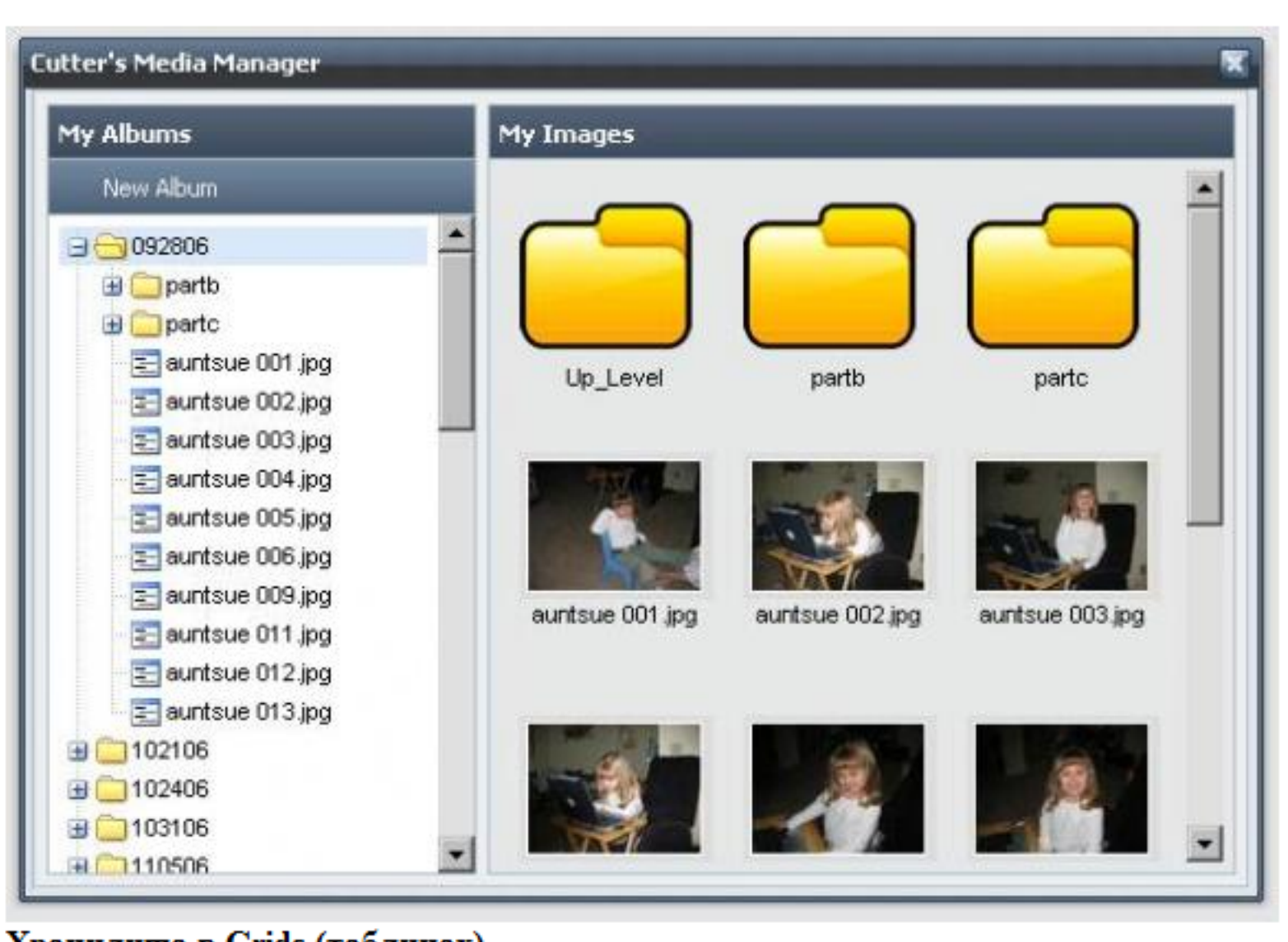
Исходный код примера:

```
var combo = new Ext.form.ComboBox({
   store: states,
   displayField: 'state',
   valueField: 'abbreviation',
   typeAhead: true,
   mode: 'remote',
   triggerAction: 'all',
   emptyText: 'Select a state...',
   selectOnFocus: true,
   applyTo: 'stateCombo'
});
```

Это комбинированное окно берет вызванные объектом Store состояние и помечает поля этих состояний для отображения во время разметки поля аббриватуры для основных отобранных значений.

Store в DataView

Объект DataView один из самых сильных объектов **Ext**. Этот объект может принимать объект Store, давайте применим Template или XTemplate для каждой Record, и отрендерим каждый предмет в пределах DataView, сворачивая предметы по мере того как у них кончается место. DataView открывает новые, интересные возможности для визуального создания контакт-листов, галерей изображений и проводников файловых систем. Открывает наши приложения для обмена данными среди большого количества объектов посредством функционала drag-and-drop.



Хранилища в Grids (таблицах)

Мы рассмотрели пример внедрения хранилища данных в таблицу в одной из прошлых глав ([Часть 5](#)). Существует несколько типов таблиц (Editor, Grouping, Property, и Basic Grids), но все из них принимают объекты Store как ввод и подходящую ColumnModel, для координации отображаемых данных.
Объекты Grid и ComboBox наиболее подходящие применение для объектов Store, с Grid и DataView, являющимися двумя основными способами отображения нескольких записей данных. Объект Tree принимает специальный источник данных, называемый TreeLoader. TreeLoader на самом деле является частью пакета Tree классов и не расширяет базовый объект Store, хотя и работает похожим образом. Как и объекты Record, TreeLoader берет массив объектов, которые затем конвертирует в узлы. Структура для входящих данных выглядит таким образом:

Исходный код примера:

Исходный код примера:

```
var dataset = [{
  id: 1,
  text: 'Node 1',
  leaf:: false
},{
  id: 2,
  text: 'Node 2',
  leaf: true
}];
```

Когда leaf имеет значение true, тогда это раскрываемый предмет, который будет запрашивать сервер для дальнейших данных при передаче данных узла. А утверждение leaf:false говори, что у узла нет детей.

Заключение

В этой главе мы научились как вытаскивать динамические серверные данные в приложение. Объекты Store **Ext JS**, с их гибкостью и синтаксисом являются легко настраиваемыми источниками данных для большинства объектов **Ext JS**. В этой главе мы привязали простые внешние данные к объекту Panel, прошли через различные форматы данных, которые может "переварить" **Ext JS** и ознакомились с началами объекта данных Store и некоторыми важными его подклассами.
Переходя к сути вещей, мы научились определять наши данные используя объект Record, после чего мы узнали как наполнять наше хранилище записями из удаленного источника. Также мы не прошли мимо целей, лежащих за DataReaders, их видов и то как создать свой собственный.
Собирая в се вместе. мы были заняты изучая техники управление Store, такими как поиск записей по значению полей, индексам и ID. мы также затронули фильтры и как они работают вместе с наборами записей. Мы обсудили изменения проводимые в локальных данных при помощи слушателя события обновления.
И наконец, мы рассмотрели некоторые другие объекты **Ext JS** которые используют Store, открывая двери для внешних данных среди множества граней наших приложений.