3

Обработчики событий и организация взаимодействий по подписке

В состав библиотеки Base входят чрезвычайно удобные и гибкие утилиты для организации взаимодействий между объектами JavaScript, узлами DOM и любыми их комбинациями. В этой главе будут представлены все эти конструкции, а также даны рекомендации, когда и какую из них предпочтительнее использовать. Переносимость программного кода, обрабатывающего события DOM, заведомо зависит от стандартизации модели событий, поэтому будет немного рассказано о том, как внутренние механизмы инструментального набора Dojo сглаживают некоторые несоответствия, имеющиеся между броузерами в области обработки событий мыши и клавиатуры. Заканчивается глава обсуждением организации взаимодействий по подписке, которая возможна благодаря механизму реализации архитектуры с гибко связанными компонентами.

Нормализация событий и клавиатуры

Самый старый программный код, входящий в инструментальный набор, был написан с целью сгладить несоответствия между моделями событий, реализованными в разных броузерах. Этот раздел представляет собой краткий обзор событий, которые можно считать нормализованными при использовании Dojo для разработки приложений. Основой стандартизации является модель, предложенная консорциумом W3C.

Нормализация событий от мыши и от клавиатуры

Механизм `dojo.connect`, который обсуждается в следующем разделе, часто имеет отношение к событиям мыши, возникающим в конкретных узлах DOM. Если вы используете Dojo, то можете быть совершенно

уверены, что перечисленные далее события мыши и клавиатуры поддерживаются в соответствии с рекомендациями стандарта W3C:

```
onclick
onmousedown
onmouseup
onmouseover
onmouseout
onmousemove
onkeydown
onkeyup
onkeypress
```

Помимо событий, стандартизованных консорциумом W3C, поддерживаются также нестандартные события `onmouseenter` и `onmouseleave`.

Помимо событий, которые запускаются стандартным способом, можно полагаться на нормализованные объекты событий, передаваемые функциям-обработчикам. На практике, при необходимости произвести нормализацию событий самостоятельно, вы можете воспользоваться следующей функцией из библиотеки Base:

```
dojo.fixEvent(/*DOMEvent*/ evt, /*DOMNode*/ sender) //Возвращает DOMEvent
```

Тип `DOMEvent` — это стандартное соглашение, которое будет использоваться в остальной части книги для ссылки на объекты событий DOM.

Другими словами, функции нужно передать событие и узел, который рассматривается как текущая цель события, и вы получите нормализованное событие, которое будет соответствовать спецификации W3C. В табл. 3.1 приводится краткое описание некоторых из наиболее часто используемых свойств объекта `DOMEvent`.[1]

Таблица 3.1. Часто используемые свойства объекта DOMEvent

Имя	Тип	Комментарий
`bubbles`	Boolean	Указывает, может ли событие «всплывать» по дереву DOM.
`cancelable`	Boolean	Указывает, может ли быть отменено связанное с событием действие по умолчанию.

[1] В настоящее время в Dojo реализована нормализация в соответствии со спецификацией DOM2, ознакомиться с которой можно по адресу *http://www.w3.org/TR/DOM-Level-2-Events/events.html*. Обзор спецификации модели событий DOM3 вы найдете по адресу *http://www.w3.org/TR/DOM-Level-3-Events/events.html*.

Имя	Тип	Комментарий
currentTarget	DOMNode	Текущий узел, чей обработчик выполняет обслуживание события. (Удобно использовать на стадии всплытия события.)
target	DOMNode	Целевой узел, которому событие предназначалось изначально.
type	String	Тип события, например mouseover.
ctrlKey	Boolean	Указывает, удерживалась ли нажатой клавиша Ctrl в момент появления события.
shiftKey	Boolean	Указывает, удерживалась ли нажатой клавиша Shift в момент появления события.
metaKey	Boolean	Указывает, удерживалась ли нажатой клавиша Meta в момент появления события. (Это специальная клавиша в компьютерах Apple.)
altKey	Boolean	Указывает, удерживалась ли нажатой клавиша Alt в момент появления события.
screenX	Integer	Координата X на экране, где возникло событие.
screenY	Integer	Координата Y на экране, где возникло событие.
clientX	Integer	Координата X относительно окна броузера, где возникло событие.
clientY	Integer	Координата Y относительно окна броузера, где возникло событие.

Стандартизованные коды клавиш

Кроме всего прочего, инструментарий определяет коды клавиш, как показано в табл. 3.2, доступные в виде свойств объекта dojo.keys. Например, чтобы определить, была ли нажата комбинация клавиш Shift+Enter, можно использовать такой фрагмент программного кода:

```
/* ... обрезано ... */
    if (evt.keyCode == dojo.keys.ENTER && evt.shiftKey) {
        /* ... */
    }
/* ... обрезано ... */
```

В табл. 3.2 приводится список констант для обращения к кодам клавиш при обработке событий от клавиатуры.

Таблица 3.2. Список констант, предоставляемых инструментарием Dojo для обращения к кодам клавиш, в виде свойств объекта dojo.keys

BACKSPACE	DELETE	NUMPAD_DIVIDE
TAB	HELP	F1
CLEAR	LEFT_WINDOW	F2

Таблица 3.2 (продолжение)

ENTER	RIGHT_WINDOW	F3
SHIFT	SELECT	F4
CTRL	NUMPAD_0	F5
ALT	NUMPAD_1	F6
PAUSE	NUMPAD_2	F7
CAPS_LOCK	NUMPAD_3	F8
ESCAPE	NUMPAD_4	F9
SPACE	NUMPAD_5	F10
PAGE_UP	NUMPAD_6	F11
PAGE_DOWN	NUMPAD_7	F12
END	NUMPAD_8	F13
HOME	NUMPAD_9	F14
LEFT_ARROW	NUMPAD_MULTIPLY	F15
UP_ARROW	NUMPAD_PLUS	NUM_LOCK
RIGHT_ARROW	NUMPAD_ENTER	SCROLL_LOCK
DOWN_ARROW	NUMPAD_MINUS	
INSERT	NUMPAD_PERIOD	

Обработчики событий

Прямые каналы взаимодействий конструируются путем явного соединения функций и/или событий DOM так, что выполнение одной функции автоматически влечет за собой вызов другой. Например, вы можете захотеть, чтобы при каждом изменении какого-либо объекта посредством метода «записи» автоматически производилось изменение в визуальном интерфейсе приложения. Или чтобы каждое изменение одного объекта автоматически вызывало обновление свойств другого объекта. Варианты могут быть самые разные.

В схему прямого взаимодействия вовлечены два основных метода – `dojo.connect` и `dojo.disconnect`. Проще говоря, метод `dojo.connect` используется для объединения в цепь последовательности событий. При каждом обращении к методу `dojo.connect` возвращается дескриптор, который необходимо сохранять и явно передавать методу `dojo.disconnect`, когда требуется разорвать связь. Обычно при выгрузке страницы все соединения разрываются автоматически, однако ручное управление дескрипторами может потребоваться, чтобы предотвратить утечки памяти в долгоживущих приложениях, которые устанавливают множество временных соединений. (Это особенно справедливо в отношении IE.) Следующие ниже сигнатуры методов уже были представлены в главе 1.

Не пытайтесь устанавливать связи, пока страница не будет полностью загружена. Попытка использовать `dojo.connect` до полной загрузки страницы – очень распространенная ошибка, которая может заставить вас потратить массу времени на выяснение того, что произошло, т. к. эту ошибку не очень легко отыскать, когда вы впервые сталкиваетесь с ней. Для обеспечения безопасности соединения всегда должны устанавливаться в пределах функции, которая передается функции `addOnLoad`.

Установка и разрыв соединения выполняются достаточно просто. Ниже приводятся сигнатуры используемых функций:

```
/* Устанавливает соединение */
dojo.connect(/*Object|null*/ obj,
             /*String*/ event,
             /*Object|null*/ context,
             /*String|Function*/ method) // Возвращает дескриптор Handle
/* Разрывает соединение */
dojo.disconnect(/*Handle*/handle);
```

С практической точки зрения вы должны рассматривать дескриптор, возвращаемый функцией `dojo.connect`, как некий «черный ящик», который не нужен ни для чего иного, кроме как для последующего разрыва соединения. (Если вам интересно, то этот объект не представляет собой ничего удивительного – просто блок информации, которая используется внутренними механизмами для управления соединением.)

Давайте рассмотрим пример, иллюстрирующий своего рода проблему, которую можно решить с помощью функции `dojo.connect`:

```
function Foo() {
    this.greet = function() { console.log("Hi, I'm Foo"); }
}

function Bar() {
    this.greet = function() { console.log("Hi, I'm Bar"); }
}

foo = new Foo;
bar = new Bar;

foo.greet();

//объект bar должен отвечать на приветствие объекта foo
//всегда, когда объект bar существует.
```

Оказывается, что решить эту маленькую проблему можно всего одной строкой программного кода. Измените предыдущий листинг, как показано ниже, и проверьте его в Firebug:

```
function Foo() {
    this.greet = function() { console.log("Hi, I'm foo"); }
}
```

```
function Bar() {
    this.greet = function() { console.log("Hi, I'm bar"); }
}

foo = new Foo;
bar = new Bar;

//При каждом вызове foo.greet автоматически будет вызываться bar.greet ...
var handle = dojo.connect(foo, "greet", bar, "greet"); //устанавливается
                                                       //соединение

foo.greet(); //теперь объект bar автоматически ответит на приветствие!
```

Вы написали всего одну строчку, а получили такой приятный результат – неплохо, не правда ли? Обратите внимание, что второй и четвертый параметры, переданные функции `dojo.connect`, – строковые литералы для соответствующих им контекстов и что функция возвращает дескриптор, который позднее позволит разорвать это соединение. Вообще говоря, *всегда* наступает некоторый момент, когда следует разорвать соединение – либо приложение достигает некоторого функционального состояния, либо вы выполняете некоторые завершающие действия, например при уничтожении некоторого объекта или при закрытии страницы, примерно так, как показано ниже:

```
var handle = dojo.connect(foo, "greet", bar, "greet");
foo.greet();

dojo.disconnect(handle);

foo.greet(); //на сей раз ответного приветствия не последует
```

Кроме того что `dojo.connect` позволяет добиться такого значительного эффекта такими малыми усилиями, обратите внимание, насколько опрятным и понятным остался программный код. Никаких шаблонных заготовок, никакой путаницы, никаких накрученных решений и никакого кошмара для того, кто будет это сопровождать.

Запуск методов в ответ на события, происходящие в странице, – действительно очень удобная возможность, но помимо этого рано или поздно появится необходимость передавать таким методам какие-нибудь аргументы. Оказывается, функция `connect` обладает возможностью передавать аргументы из контекста одной функции в контекст другой функции. Ниже приводится пример, как это делается:

```
function Foo() {
    this.greet = function(greeting) { console.log("Hi, I'm Foo.",greeting); };
}

function Bar() {
    this.greet = function(greeting) { console.log("Hi, I'm Bar.",greeting); };
}

foo = new Foo;
bar = new Bar;
```

```
var handle= dojo.connect(foo, "greet", bar, "greet");
foo.greet("Nice to meet you");
```

Вы наверняка можете оценить, насколько удобно, когда передача аргументов происходит автоматически, и это особенно верно для функций, связанных с событиями DOM, например щелчок мышью, потому что такая возможность обеспечивает для функции моментальный доступ ко всем важным сведениям о событии, таким как целевой элемент, координаты указателя мыши и т. д. Давайте рассмотрим еще один пример:

```
//Обратите внимание, что третий аргумент опущен, так как обработчик -
//это анонимная функция. Если в третьем аргументе передать значение null,
//мы получим тот же самый эффект.
dojo.connect(
    dojo.byId("foo"), //Некоторый элемент DOM
    "onmouseover",
    function(evt) {
        console.log(evt);
    });
```

Если создать пробную страницу, установить в ней соединение и наблюдать за происходящим в консоли Firebug, можно заметить, что функции-обработчику объект события, содержащий практически все сведения, какие только может потребоваться знать о только что произошедшем событии доступен целиком.

Возникает резонный вопрос: «Если настолько просто определять обработчики событий DOM, зачем тогда заниматься изучением другой библиотечной функции?». Да, возможно, не требуется быть нейрохирургом, чтобы собрать вместе несколько простых обработчиков событий, но как быть, когда имеется сложное приложение, основанное на обработке большого числа сложных событий с учетом предпочтений пользователя, на обработке нестандартных событий или обладающее иным поведением, управляемым событиями? Несомненно, всю необходимую работу можно выполнить вручную, но сможете ли вы устанавливать и разрывать соединение однострочными командами, с единым непротиворечивым интерфейсом, уже написанными и всесторонне проверенными?

Наконец, обратите внимание: хотя в приведенных примерах связывались между собой только пара событий, вам ничто не мешает связать воедино произвольное число обычных функций, методов объектов и событий DOM для их последовательного вызова.

Распространение событий

Бывают моменты, когда необходимо переопределить обработку некоторых событий DOM, встроенную в броузер, и с помощью функции `dojo.connect` подставить свои функции, которые будут обрабатывать эти события. В качестве двух наиболее типичных примеров можно

привести необходимость предотвратить автоматический переход броузера после щелчка на гиперссылке и необходимость предотвратить автоматическую отправку формы по нажатию на клавишу Enter или в результате щелчка на кнопке Submit.

К счастью, помешать броузеру выполнить действие по умолчанию, заданное для этих событий DOM, очень просто – достаточно лишь вызвать функцию `dojo.stopEvent` или метод `preventDefault` объекта `DOMEvent`, и событие прекратит свое распространение по броузеру. Функция `stopEvent` принимает в качестве параметра объект `DOMEvent`:

```
dojo.stopEvent(/*DOMEvent*/evt)
```

Распространение события DOM можно подавить в последовательности функций, связанных с событием с помощью `dojo.connect`, но нет никакой возможности остановить работу цепочки обычных функций или методов объектов JavaScript, связанных функцией `dojo.connect`.

Следующий пример демонстрирует применение функции `stopEvent`:

```
var foo = dojo.byId("foo"); //некоторый якорный элемент

dojo.connect(foo, "onclick", function(evt) {
    console.log("anchor clicked");
    dojo.stopEvent(evt); //предотвратит переход по ссылке и дальнейшее
                         //всплытие события
});
```

Точно так же просто отменяется автоматическая отправка формы, для этого достаточно передать контекст соединения и связать его с событием `submit`. Однако на этот раз, чтобы отменить действие события по умолчанию, мы воспользуемся методом `preventDefault` объекта `DOMEvent`, который при этом не предотвращает дальнейшее всплытие события в дереве DOM:

```
var bar = dojo.byId("bar"); //some form element

dojo.connect(bar, "onsubmit", function(evt) {
    console.log("form submitted");
    evt.preventDefault();//предотвратит отправку формы, но не запретит
                         //дальнейшее всплытие события
});
```

Использование замыканий с функцией dojo.connect

В этом разделе рассматривается относительно сложная тема, поэтому вы можете просто бегло ознакомиться с ней – так, чтобы не увязнуть при первом прочтении и вернуться сюда позднее, потому что рано или поздно эти сведения вам потребуются.

Однократные соединения

Рассмотрим ситуацию, когда необходимо установить соединение, которое должно быть разорвано после первого же срабатывания. Следующий пример показывает, как выполнить эту работу с минимальными усилиями:

```
var handle = dojo.connect(
    dojo.byId("foo"), //некоторый элемент div
    "onmouseover",
    function(evt) {
        //здесь находится тело некоторого обработчика...
        dojo.disconnect(handle);
    }
);
```

Если вы еще недостаточно уверенно чувствуете себя при работе с замыканиями, вашей первой реакцией может быть: «То, что мы только что сделали, попросту невозможно». В конце концов, переменная `handle` получает значение, возвращаемое функцией `dojo.connect`, и при этом ссылка на нее используется внутри функции, которая передается в `dojo.connect` в качестве параметра. Чтобы лучше понять ситуацию, разберем все происходящее в деталях:

1. Вызывается функция `dojo.connect`, и, хотя одним из ее параметров является анонимная функция, она в этот момент еще не выполняется.
2. Любые переменные внутри анонимной функции (такие как `handle`) связаны с ее областью видимости, и хотя они присутствуют в теле функции, фактическое обращение к ним происходит, только когда функция действительно будет вызвана, поэтому в этом программном коде не возникает никакой ошибки.
3. Функция dojo.connect возвращает значение переменной `handle` еще до того, как анонимная функция будет вызвана. Поэтому к моменту вызова анонимной функции значение переменной будет определено и может передаваться функции `dojo.disconnect`.

Установка соединений в цикле

Еще одна ситуация, часто встречающаяся на практике, – необходимость устанавливать соединения в теле цикла. Предположим, что у нас в странице имеется несколько элементов – `foo0`, `foo1`, ..., `foo9`, и нам необходимо при перемещении указателя мыши над этими элементами выводить уникальное для каждого из них число. При первой попытке вы могли бы прийти к следующему фрагменту программного кода, который, впрочем, *не даст* ожидаемого результата:

```
/* Следующий фрагмент работает не так, как ожидается! */
for (var i=0; i < 10; i++) {
    var foo = dojo.byId("foo"+i);
    var handle = dojo.connect(foo, "onmouseover", function(evt) {
        console.log(i);
```

```
        dojo.disconnect(handle);
    });
}
```

Если вы запустите этот фрагмент в Firebug на странице с серией указанных элементов, вы быстро обнаружите, что тут имеется проблема. А именно, в консоли всегда будет выводиться число 10, то есть всеми функциями-обработчиками будет использоваться последнее значение переменной i, а это означает, что все десять обработчиков по ошибке будут пытаться разорвать одно и то же соединение. Остановимся на минутку, чтобы обдумать ситуацию. Однако такому неожиданному для вас поведению имеется разумное объяснение: внутри замыкания, образованного анонимной функцией, переданной функции dojo.connect, не выполняется разрешение имени i, пока функция действительно не будет вызвана, но к этому моменту переменная i будет иметь последнее полученное в цикле значение.

Следующие изменения устраняют проблему, захватывая значение переменной i в ловушку цепочки областей видимости, чтобы при последующем обращении к переменной возвращалось значение, которое было на момент вызова функции dojo.connect:

```
for (var i=0; i < 10; i++) {
    (function() {
        var foo = dojo.byId("foo"+i);
        var current_i = i; //поймать в ловушку замыкания
        var handle = dojo.connect(foo, "onmouseover",
            function(evt) {
                console.log(current_i);
                dojo.disconnect(handle);
            }
        );
    })(); //выполнить анонимную функцию немедленно
}
```

Поначалу этот фрагмент программного кода может показаться замысловатым, но на самом деле здесь нет ничего сложного. Все тело цикла оформлено в виде анонимной функции, которая тут же и выполняется, а поскольку анонимные функции образуют замыкание для всего, что находится внутри, значение переменной i попадает в «ловушку» переменной current_i, которая может быть разрешена во время выполнения обработчика события. Точно так же правильно будет разрешаться и переменная handle, потому что она тоже существует в пределах замыкания, образованного встроенной анонимной функцией.

Если раньше вы никогда не встречались с такими замыканиями в действии, возможно, вам стоит потратить некоторое время на более внимательное изучение программного кода, чтобы полностью понять его. Вероятно, вы уже устали слушать о замыканиях, но в дальнейшем уверенное понимание этой темы сослужит вам хорошую службу в работе с JavaScript.

Соединения в тексте разметки

Стоит заметить, что точно так же можно создавать соединения для диджитов вообще без (или с минимальным использованием) программного кода JavaScript при использовании специальных тегов `SCRIPT` с функцией `dojo.connect` в тексте разметки. Подробнее об этом вы можете прочитать в главе 11, когда будет дано формальное введение в библиотеку Dijit.

Организация взаимодействий по подписке

Существует масса ситуаций, когда прямой «цепочечный» стиль организации взаимодействий посредством функции `dojo.connect` является именно тем средством, которое необходимо для решения проблем. Однако существует не меньше ситуаций, когда требуется более опосредованный «широковещательный» стиль организации взаимодействий, когда различные виджеты взаимодействуют друг с другом анонимно. В таких случаях можно использовать функции `dojo.publish` и `dojo.subscribe`.

Классическим примером является ситуация, когда необходимо организовать взаимодействие между одним объектом JavaScript и несколькими другими объектами по типу отношений «один ко многим». Вместо того чтобы создавать и управлять множественными соединениями, создаваемыми с помощью функции `dojo.connect`, которые больше напоминают одно связное действие, значительно проще организовать передачу уведомления о событии, произошедшем в одном виджете (наряду с данными, сопутствующими этому событию) , чтобы другие виджеты могли подписаться на получение этих извещений и автоматически принимать соответствующие меры. Вся прелесть такого подхода состоит в том, что объект, осуществляющий отправку уведомлений, ничего не должен знать о других объектах и даже не должен делать никаких предположений об их существовании. Другим классическим примером такого рода взаимодействий являются *портлеты* – подключаемые компоненты интерфейса (*http://en.wikipedia.org/wiki/Portlet*[1]), которые обслуживаются веб-порталом, иногда через панель управления.

OpenAjax Hub (*http://www.openajax.org/OpenAjax%20Hub.html*), о котором подробнее будет рассказываться в главе 4, использует взаимодействия по подписке как механизм эффективного задействования множества библиотек JavaScript в пределах одной и той же страницы.

Во многих ситуациях при помощи организации взаимодействий по подписке можно достичь тех же возможностей, что и в результате прямого соединения элементов, поэтому решение о применении механизма

[1] На русском языке: *http://ru.wikipedia.org/wiki/Портлет*. – *Прим. перев.*

обработки событий по подписке часто зависит от практичности метода, от конкретной проблемы, которую требуется решить, и личных предпочтений, отдаваемых одному из подходов.

Для начала, чтобы определить, какой стиль организации взаимодействий лучше подходит для решения проблемы, попробуйте ответить на следующие вопросы:

- Предполагаете ли вы (и насколько это правильно) публиковать прикладной интерфейс виджета, который вы разрабатываете? Если нет, тогда вам определенно следует использовать взаимодействия по подписке, потому что это позволит вам свободно менять архитектуру виджета, не опасаясь вступить в конфликт с его опубликованным API.
- Предполагается ли одновременное использование нескольких однотипных виджетов, которые порождают один и тот же тип событий? Если да, тогда вам определенно следует использовать взаимодействия, основанные на прямых соединениях, потому что в противном случае вам придется разрабатывать дополнительную логику, которая будет определять, какой виджет и на какое событие должен реагировать.
- Разрабатываете ли вы виджет, содержащий другие виджеты в отношении «имеет»? Если да, тогда вам следует использовать взаимодействия, основанные на прямых соединениях.
- Предполагается ли возможность организации взаимодействий по принципу «один ко многим» или «многие ко многим»? Если да, тогда определенно следует использовать взаимодействия по подписке, чтобы минимизировать тяжесть организации взаимодействий, основанных на прямых соединениях.
- Предполагается ли, что взаимодействия должны быть полностью анонимными и должна ли быть обеспечена свобода в выборе взаимодействующих сторон? Если да, тогда следует использовать взаимодействия по подписке.

Не будем долго задерживаться и рассмотрим интерфейс организации взаимодействий по подписке. Обратите внимание, что в случае функции `dojo.subscribe` можно опустить параметр `context`, а функция выполнит необходимую нормализацию аргументов (точно так же, как и в случае с функцией `dojo.connect`):

```
dojo.publish(/*String*/topic, /*Array*/args)
dojo.subscribe(/*String*/topic, /*Object|null*/context,
               /*String|Function*/method) //Возвращает Handle
dojo.unsubscribe(/*Handle*/handle)
```

Объект дескриптора, возвращаемый функцией `dojo.subscribe`, следует считать своеобразным черным ящиком, как и в случае с функцией `dojo.connect`.

Давайте рассмотрим простой пример, основанный на использовании функций `dojo.subscribe` и `dojo.publish`:

```
function Foo(topic) {

    this.topic = topic;

    this.greet = function() {
        console.log("Hi, I'm Foo");

      /* Объект Foo публикует информацию, но не для конкретного получателя */
        dojo.publish(this.topic);
    }
}

function Bar(topic) {

    this.topic = topic;

    this.greet = function() {
        console.log("Hi, I'm Bar");
    }

 / * Объект Bar подписывается на информацию, но не от конкретного источника */
    dojo.subscribe(this.topic, this, "greet");
}

var foo = new Foo("/dtdg/salutation");
var bar = new Bar("/dtdg/salutation");

foo.greet(); //Hi, I'm Foo...Hi, I'm Bar
```

Несмотря на отсутствие формального стандарта, инструментальный набор использует соглашение о начальном символе и использует начальный символ слеша для обозначения названия темы. Преимущество такого подхода заключается в том, что в программном коде JavaScript весьма непривычно видеть начальный символ слеша, поэтому он сразу бросается в глаза (тогда как при использовании точек в названиях тем выделить их было бы намного сложнее).

Как видите, функция `connect` связывает конкретный источник с конкретным адресатом, а функции `publish/subscribe` позволяют работать с широковещательными посылками, которые могут исходить из любого источника и обрабатываться любыми получателями, заинтересованными в получении таких посылок. Отчасти удивительная мощь, заключенная в архитектуре с гибко связанными компонентами, обусловлена тем, что при минимальных усилиях и большой простоте она позволяет получать приложения, которые концептуально представляют собой совокупность связанных компонентов.

Давайте посмотрим, как можно отписаться от получения уведомлений, внеся изменения в реализацию объекта `Bar`. Пусть теперь объект `Bar` отвечает на сообщение, публикуемое объектом `Foo`, только один раз:

```
function Bar(topic) {

    this.topic = topic;

    this.greet = function() {
        console.log("Hi, I'm bar");
        dojo.unsubscribe(this.handle);

        //не тревожь ты меня, я тебе не отвечу
    }

    this.handle = dojo.subscribe(this.topic, this, "greet");
}
```

Обратите внимание, что существует возможность вместе с сообщением передать как второй аргумент функции `publish` массив значений в виде именованных параметров, которые будут переданы обработчику, подписанному с помощью функции `subscribe`.

Очень часто встречается ошибка, когда разработчик забывает, что дополнительные аргументы должны передаваться функции `publish` в виде массива и что обработчик, подписанный с помощью функции `subscribe`, получает их в виде отдельных аргументов.

И, в заключение, предположим, что у нас отсутствует возможность изменить метод `greet` объекта `Foo`, чтобы включить в него вызов `dojo.publish`, из-за наличия каких-то внешних обстоятельств, которые препятствуют этому, например когда программный код не является вашей собственностью или не должен изменяться. Это не причина для беспокойств – мы можем использовать другую функцию, `dojo.connectPublisher`, которая будет выполнять публикацию события всякий раз, когда оно происходит:

```
function Foo() {
    this.greet = function() {
        console.log("Hi, I'm foo");
    }
}

function Bar() {
    this.greet = function() {
        console.log("Hi, I'm bar");
    }
}

var foo = new Foo;
var bar = new Bar;

var topic = "/dtdg/salutation";
dojo.subscribe(topic, bar, "greet");
dojo.connectPublisher(topic, foo, "greet");

foo.greet();
```

Возможно, вам будет интересно узнать, что внутри `connectPublisher` использует функцию `dojo.connect`, чтобы создать соединение между функцией `dojo.publish` и указанной функцией.

Суть этого заключительного примера состоит в том, что функция `dojo.connectPublisher` позволила добиться тех же результатов, что и добавление вызова `dojo.publish` в метод `greet`, но без изменения исходного программного кода. В этом смысле объект `foo` является косвенным источником уведомлений и даже не подозревает, что вообще осуществляется какое-то взаимодействие. С другой стороны, объект `Bar`, как подписчик на получение уведомления, требует явного знания схемы взаимодействий. По существу, это противоположный случай типичного использования функции `dojo.connect`, когда объект, предоставляющий информацию для обмена, должен явно знать о других объектах или функциях, которые играют роль «целей» соединения.

В заключение

После прочтения этой главы вы должны:

- Знать, что `dojo.connect` стандартизует объект события, который передается функции-обработчику и обеспечивает переносимость между платформами
- Понимать, как `dojo.connect` позволяет произвольно связывать события DOM, объекты событий JavaScript и обычные функции для реализации структуры, управляемой событиями
- Уметь использовать средства организации взаимодействий по подписке и создавать каналы связи с гибко связанными компонентами
- Знать, когда в архитектуре приложения предпочтительнее использовать `dojo.connect`, а когда – взаимодействия по подписке.

В следующей главе рассматриваются технология AJAX и организация взаимодействий с сервером.