Введение в CGI

Лекции	Описание
1. **Общие сведения**	В данной лекции определяется место CGI-скриптов в общем контексте Web-технологий. Обсуждаются основные способы применения скриптов и особенности программирования для Web. В лекции введены все необходимые понятия CGI-программирования.
2. **Введение в программирование CGI-скриптов и программирование скриптов на bash**	В этой лекции подробно разбираются особенности программирования CGI-скриптов, общие принципы программирования CGI-скриптов на bash, определяются правила вызова скрипта, передачи ему данных и получение результатов работы скрипта для дальнейшего использования в HTTP-обмене и генерации HTML-страниц.
3. **Введение в программирование на Perl**	В этой лекции подробно разбираются особенности программирования CGI-скриптов на языке Perl. Определяются правила вызова скрипта, передачи ему данных и получение результатов работы скрипта для дальнейшего использования в HTTP-обмене и генерации HTML-страниц. Разбираются причины популярности Perl-скриптов среди Web-программистов.
4. **Введение в программирование скриптов на C**	В этой лекции подробно разбираются особенности программирования CGI-скриптов на языке C. Определяются правила вызова скрипта, передачи ему данных и получение результатов работы скрипта для дальнейшего использования в HTTP-обмене и генерации HTML-страниц. Разбираются причины эффективности C-скриптов по сравнению с другими инструментами.
	В данной лекции подробным образом разбираются элементы разметки, входящие в группу HTML-FORM. Рассматриваются их атрибуты,

5.	**HTML-формы**	совместимость атрибутов и форматы записи данных при формировании запросов к HTTP-серверу.
6.	**Применение методов доступа HTTP в рамках программирования CGI-скриптов. Настройка HTTP-сервера для работы с CGI-скриптами**	В этой лекции разбираются способы взаимодействия между браузерами и HTTP-сервером по методу доступа GET и POST. При использовании HTML-форм. Объясняются особенности формирования HTML-сообщения и разбора его CGI-скриптом.

Дополнительные материалы

Формат и синтаксис Cookie

Литература

Программное обеспечение

Предметный указатель

Примеры

Экзамен

Сдать экзамен экстерном

1: Общие сведения

Главным достижением технологии **World Wide Web** по праву считают унификацию интерфейса пользователя при работе с информационными ресурсами Internet. Универсальный мультипротокольный браузер, будь то Netscape Navigator или Internet Explorer, позволяет путем выбора гипертекстовой ссылки получить доступ к FTP-архиву, архиву Gopher, новостям из конференции Usenet или отправить письмо по электронной почте. До эпохи Web для каждого из этих ресурсов пришлось бы запускать отдельную программу.

Однако, кроме текстов, которые можно читать, или картинок, которые можно просматривать, существует множество ресурсов, требующих ввода информации в процессе работы с ними. К таким ресурсам, в частности, относятся информационно-поисковые системы, где пользователь должен вводить список ключевых слов или реляционные (да и любые другие) базы данных, формулируя запрос к отношениям. Более того, для любой страницы, которая требует аутентификации пользователя, необходимо вводить идентификатор и пароль.

На сегодня уже сложился определенный стиль графического интерфейса приложения. Существует достаточно большое число прикладных пакетов, которые позволяют "прилаживать" такой интерфейс к программе. Однако на такое "прилаживание" или прямое программирование уходит до 80% трудозатрат программистов. При этом в большинстве случаев все сводится к разбору введенных параметров с последующей выдачей результатов в виде отформатированного текста.

Форматирование страниц в Web-технологии достигается за счет HTML-разметки. Остается только создать инструмент ввода данных через рабочее окно браузера или через HTML-документ. В 1991 году эта проблема была решена специалистами NCSA. Они разработали и реализовали две взаимосвязанные спецификации: HTML-формы и Common Gateway Interface.

Формы произвели настоящую революцию в HTML-разметке: авторы документов получили возможность создавать сложные шаблоны ввода информации в рамках HTML-страницы, пользователи — эти шаблоны

заполнять. При этом авторы форм опирались на свойства HTTP-протокола и универсальный локатор ресурсов URL с учетом того, что при HTTP-обмене можно использовать различные методы доступа к ресурсам. Это позволило сделать механизм интерпретации форм расширяемым и легко приспосабливаемым к дальнейшему развитию Web-технологии. Таким образом, кроме HTTP, можно было использовать и другие протоколы, которые поддерживали универсальный браузер, например mailto.

Common Gateway Interface — это спецификация обмена данными между прикладной программой, выполняемой по запросу пользователя, и HTTP-сервером, который данную программу запускает. До появления CGI новые функции нужно было внедрять непосредственно в сервер. CGI позволила разрабатывать программы независимо от сервера, а механизм передачи им управления и данных был унаследован от программирования в среде командной строки. Последнее резко сократило трудозатраты на разработку приложений, так как не надо было программировать интерфейс пользователя: его функции выполняли формы.

Слушатели данного учебного курса научатся создавать документы с формами, программировать на стороне сервера с использованием CGI и обрабатывать данные, передаваемые браузером серверу. В рамках курса будут подробно рассмотрены различные способы такой обработки, а также основные приемы построения интерактивных страниц Web-узла.

Введение

Обмен данными в Web-технологии подразделяется в соответствии с типами методов доступа протокола HTTP и видами запросов в спецификации CGI.

Основных методов доступа два: GET и POST. Помимо них часто используются HEAD и PUT.

Виды запросов CGI разделяют на два основных **MIME-типа**: application/x-www-form-urlencoded и multipart/form-data. Второй тип запроса специально создан для передачи больших внешних файлов.

Эту классификацию можно представить в виде таблицы:

		Клиент <--

Метод		Клиент --> Сервер	Сервер
GET	По умолчанию	Только HTTP-заголовок	HTTP-заголовок и страница, как тело HTTP-сообщения
	isindex	Только HTTP-заголовок (список ключевых слов включен в URL. Слова разделены символом "+". Кодирования кириллицы не производится)	HTTP-заголовок и страница, как тело HTTP-сообщения
	form-urlencoded	Только HTTP-заголовок (данные из формы включены в URL страницы. Производится кодирование специальных символов и кириллицы) HTTP-сообщения	HTTP-заголовок и страница, как тело HTTP-сообщения
POST	form-urlencoded	Только HTTP-заголовок (данные из формы включены в URL страницы. Производится кодирование специальных символов и кириллицы) HTTP-сообщения	HTTP-заголовок и страница, как тело HTTP-сообщения
	form-data	HTTP-заголовок и составное тело HTTP-сообщения. Первая часть тела — данные из формы, для которых производится кодирование, вторая часть тела — присоединенный файл как он есть	HTTP-заголовок и страница, как тело HTTP-сообщения
PUT		HTTP-заголовок и документ, как тело HTTP-сообщения	HTTP-заголовок. В качестве тела можно передать комментарий к коду возврата
HEAD		HTTP-заголовок	HTTP-заголовок

При реализации нестандартных методов доступа, например, DELETE, могут быть несколько иные комбинации содержания откликов и ответов.

Мы рассмотрим все эти типы обменов.

HyperText Transfer Protocol

Все данные в рамках Web-технологии передаются по протоколу **HTTP**. Исключение составляет обмен с использованием программирования на Java или обмен из Plugin-приложений. Учитывая реальный объем трафика, который передается в рамках Web-обмена по HTTP, мы будем рассматривать только этот протокол. При этом мы остановимся на таких вопросах, как:

- общая структура сообщений;
- методы доступа;
- оптимизация обменов.

Общая структура сообщений

HTTP — это протокол прикладного уровня. Он ориентирован на модель обмена "клиент-сервер". Клиент и сервер обмениваются фрагментами данных, которые называются HTTP-сообщениями. Сообщения, отправляемые клиентом серверу, называют запросами, а сообщения, отправляемые сервером клиенту — откликами. Сообщение может состоять из двух частей: заголовка и тела. Тело от заголовка отделяется пустой строкой.

Заголовок содержит служебную информацию, необходимую для обработки тела сообщения или управления обменом. Заголовок состоит из директив заголовка, которые обычно записываются каждая на новой строке.

Тело сообщения не является обязательным, в отличие от заголовка сообщения. Оно может содержать текст, графику, аудио- или видеоинформацию.

Ниже приведен HTTP-запрос:

```
GET / HTTP/1.0
Accept: image/jpeg
пустая строка
```

И отклик:

```
HTTP/1.0 200 OK
```

```
Date: Fri, 24 Jul 1998 21:30:51 GMT
Server: Apache/1.2.5
Content-type: text/html
Content-length: 21345
пустая строка
<HTML>
...
</HTML>
```

Текст "пустая строка" — это просто обозначение наличия пустой строки, которая отделяет заголовок HTTP-сообщения от его тела.

Сервер, принимая запрос от клиента, часть информации заголовка HTTP-запроса преобразует в переменные окружения, которые доступны для анализа CGI-скриптом. Если запрос имеет тело, то оно становится доступным скрипту через поток стандартного ввода.

Методы доступа

Самой главной директивой HTTP-запроса является метод доступа. Он указывается первым словом в первой строке запроса. В нашем примере это GET. Различают четыре основных метода доступа:

- GET;
- HEAD;
- POST;
- PUT.

Кроме этих четырех методов существует еще около пяти дополнительных методов доступа, но они используются редко.

Метод GET

Метод GET применяется клиентом при запросе к серверу по умолчанию. В этом случае клиент сообщает адрес ресурса (URL), который он хочет получить, версию протокола HTTP, поддерживаемые им MIME-типы документов, версию и название клиентского программного обеспечения. Все эти параметры указываются в заголовке HTTP-запроса. Тело в запросе не передается.

В ответ сервер сообщает версию HTTP-протокола, код возврата, тип

содержания тела сообщения, размер тела сообщения и ряд других необязательных директив HTTP-заголовка. Сам ресурс, обычно HTML-страница, передается в теле отклика.

Метод HEAD

Метод HEAD используется для уменьшения обменов при работе по протоколу HTTP. Он аналогичен методу GET за исключением того, что в отклике тело сообщения не передается. Данный метод используется для проверки времени последней модификации ресурса и срока годности кэшированных ресурсов, а также при использовании программ сканирования ресурсов World Wide Web. Одним словом, метод HEAD предназначен для уменьшения объема передаваемой по сети информации в рамках HTTP-обмена.

Метод POST

Метод POST — это альтернатива методу GET. При обмене данными по методу POST в запросе клиента присутствует тело HTTP-сообщения. Это тело может формироваться из данных, которые вводятся в HTML-форме, или из присоединенного внешнего файла. В отклике, как правило, присутствует и заголовок, и тело HTTP-сообщения. Чтобы инициировать обмен по методу POST, в атрибуте METHOD контейнера FORM следует указать значение "post".

Метод PUT

Метод PUT используется для публикации HTML-страниц в каталоге HTTP-сервера. При передаче данных от клиента к серверу в сообщении присутствует и заголовок сообщения, в котором указан URL данного ресурса, и тело — содержание размещаемого ресурса.

В отклике тело ресурса обычно не передается, а в заголовке сообщения указывается код возврата, который определяет успешное или неуспешное размещение ресурса.

Оптимизация обменов

Протокол HTTP изначально не был ориентирован на постоянное

соединение. Это означает, что как только сервер принял запрос от клиента и ответил на него, соединение между клиентом и сервером разрывается. Для нового обмена данными нужно устанавливать новое соединение. Такой подход имеет как достоинства, так и недостатки.

К достоинствам относится возможность одновременного обслуживания большого количества коротких запросов. Даже на популярных серверах число открытых соединений может не превышать сотни при обслуживании порядка миллиона запросов в сутки. При этом один клиент может открыть до 40 соединений одновременно, и с точки зрения сервера все они равноправны. При высокоскоростных линиях связи это позволяет добиться малого времени отклика на запрос клиента для всей страницы (текст, графика и т.п.).

К недостаткам такой схемы обмена относятся: необходимость каждый раз устанавливать соединение и невозможность поддерживать сессию работы с информационным ресурсом. При инициализации соединения по транспортному протоколу TCP и разрыве этого соединения требуется передать довольно большой объем служебной информации. Отсутствие поддержки сессий в HTTP затрудняет работу с такими ресурсами как базы данных или ресурсы, требующие аутентификации.

Для оптимизации числа открытых TCP-соединений в HTTP-протоколе версий 1.0 и 1.1 предусмотрен режим keep-alive. В этом режиме соединение инициализируется только один раз, и по нему последовательно можно реализовать несколько HTTP-обменов.

Для обеспечения поддержки сессий к директивам HTTP-заголовка были добавлены "ключики" (cookies). Они позволяют сымитировать поддержку соединения при работе по протоколу HTTP.

Виды интерфейса пользователя в Web-технологии

Страницы World Wide Web по функциональному назначению можно разделить на несколько типов: информационные страницы, навигационные страницы, страницы обмена данными. Во многих случаях эти функции можно объединить в одной странице.

Информационные страницы — это последовательное изложение

информации с возможностью гипертекстовых контекстных переходов. Пользователь просматривает их последовательно. Гипертекстовые ссылки обычно применяют для создания сносок, примечаний или отсылок к спискам литературы и других ассоциативных материалов. Типичными примерами таких страниц являются подсказки, руководства, описания компаний, исторические справки и т.п.

Навигационные страницы — это совокупность гипертекстовых ссылок, которая позволяет ориентироваться в материалах Web-узла. Типичный пример такой страницы — Home page (домашняя страница). Как правило, на ней нет пространных текстовых описаний и иллюстраций, она состоит из совокупности различных меню. Эти меню можно реализовать через списки, таблицы ссылок или imagemap.

Страницы обмена данными позволяют передать на сервер некоторый объем информации, отличный от стандартного адреса (URL) ресурса. При просмотре и навигации пользователь просто выбирает гипертекстовые ссылки, по которым загружаются новые страницы. При обмене данными на сервер передается не только адрес ресурса, но и дополнительная информация, которую вводит пользователь.

В зависимости от функционального назначения страниц изменяется вид интерфейса ресурса, с которым пользователь имеет дело. В первых двух случаях достаточно манипулятором "мышь" выбрать гипертекстовую ссылку, как тут же загрузится новая страница. В случае страниц обмена данными следует заполнить поля HTML-форм и отправить данные на сервер.

При этом формы обеспечивают практически все необходимые виды полей ввода и меню. Единственное, чего не позволяют реализовать HTML-формы, так это вложенные меню. Формы можно применять не только при обмене данными. Достаточно развитые механизмы обработки форм присутствуют в JavaScript.

Спецификация Common Gateway Interface

Данная спецификация определяет стандартный способ обмена данными между прикладной программой и HTTP-сервером. Спецификация была предложена для сервера NCSA и является основным средством расширения

возможностей обработки запросов клиентов HTTP-сервером.

В CGI имеет смысл выделить следующие основные моменты:

- понятие CGI-скрипта;
- типы запросов;
- механизмы приема данных скриптом;
- механизм генерации отклика скриптом.

Основное назначение CGI — обработка данных из HTML-форм. В настоящее время область применения CGI гораздо шире.

Понятие CGI-скрипта

CGI-скриптом называют программу, написанную на любом языке программирования или командном языке, которая осуществляет обмен данными с HTTP-сервером в соответствии со спецификацией Common Gateway Interface.

Наиболее популярными языками для разработки скриптов являются Perl и C.

Типы запросов

Различают два типа запросов к CGI-скриптам: по методу GET и по методу POST. В свою очередь, запросы по методу GET подразделяются на запросы по типам кодирования: isindex и form-urlencoded, а запросы по методу POST — multipart/form-data и form-urlencoded.

В запросах по методу GET данные от клиента передаются скрипту в переменной окружения QUERY_STRING. В запросах по методу POST данные от скрипта передаются в потоке стандартного ввода скрипта. При передаче через поток стандартного ввода в переменной окружения CONTENT_LENGTH указывается число передаваемых символов.

Запрос типа ISINDEX — это запрос вида:

```
http://intuit.ru/somthing-cgi/cgi-script?слово1+слово2+слово3
```

Главным здесь является список слов после символа "?". Слова перечисляются через символ "+" и для кириллицы в шестнадцатеричные последовательности не кодируются. Последовательность слов после символа "?" будет размещена в переменной окружения QUERY_STRING.

Запрос типа form-urlencoded — это запрос вида:

```
http://intuit.ru/somthing-cgi/cgi-script?field=word1&field2=word2
```

Данные формы записываются в виде пар "имя_поля-значение", которые разделены символом "&".

Приведенный пример — это обращение к скрипту по методу GET. Все символы после "?" попадут в переменную окружения QUERY_STRING. При этом если в значениях полей появляется кириллица или специальные символы, то они заменяются шестнадцатиричным кодом символа, который следует за символом "%".

При обращении к скрипту по методу POST данные после символа "?" не будут размещаться в QUERY_STRING, а будут направлены в поток стандартного ввода скрипта. В этом случае количество символов в потоке стандартного ввода скрипта будет указано в переменной окружения CONTENT_LENGTH.

При запросе типа multipart/form-data применяется составное тело HTTP-сообщения, которое представляет собой данные, введенные в форме, и данные присоединенного внешнего файла. Это тело помещается в поток стандартного ввода скрипта. При этом к данным формы применяется кодирование как в form-urlencoded, а данные внешнего файла передаются как есть.

Механизмы приема данных скриптом

Скрипт может принять данные от сервера тремя способами:

- через переменные окружения;
- через аргументы командной строки;
- через поток стандартного ввода.

При описании этих механизмов будем считать, что речь идет об обмене

данными с сервером Apache для платформы Unix.

Переменные окружения

При вызове скрипта сервер выполняет системные вызовы fork и exec. При этом он создает среду выполнения скрипта, определяя ее переменные. В спецификации CGI определены 22 переменные окружения. При обращении к скрипту разными методами и из различных контекстов реальные значения принимают разные совокупности этих переменных. Например, при обращении по методу POST переменная QUERY_STRING не имеет значения, а по методу GET — имеет. Другой пример — переменная окружения HTTP_REFERER. При переходе по гипертекстовой ссылке она определена, а если перейти по значению поля location или через JavaScript-программу, то HTTP_REFERER определена не будет.

Получить доступ к переменным окружения можно в зависимости от языка программирования следующим образом:

```
#Perl
$a = $ENV{CONTENT_LENGTH};
...
// C
a = getenv("CONTENT_LENGTH");
```

В случае доступа к скрипту по методу GET данные, которые передаются скрипту, размещаются в переменной окружения QUERY_STRING.

Аргументы командной строки

Как ни странно звучит, но у CGI-скрипта может быть такой элемент операционного окружения как командная строка. Это не означает, что скрипт реально можно вызвать из командной строки через сервер. Тем не менее получить доступ к содержанию командной строки скрипта можно с помощью тех же функций, что и при вызове его из-под интерактивной оболочки:

```
#Perl
foreach $a (@ARGV)
{
print $a,"\n";
}
```

```
// C
void main(argc,argv)
int argc;
char *argv[];
{
int i;
for(i=0;i<argc;i++)
{
printf("%s\n",argv[i]);
}
}
```

В обоих примерах показана распечатка аргументов командной строки для программ на Perl и C соответственно.

Аргументы командной строки появляются только в запросах типа ISINDEX.

Поток стандартного ввода

Ввод данных в скрипт через поток стандартного ввода осуществляется только при использовании метода доступа к ресурсу (скрипту) POST. При этом в переменную окружения CONTENT_LENGTH помещается число символов, которое необходимо считать из потока стандартного ввода скрипта, а в переменную окружения CONTENT_TYPE помещается тип кодирования данных, которые считываются из потока стандартного ввода.

При посимвольном считывании в C можно применить, например, такой фрагмент кода:

```
int n;
char *buf;
n= atoi(getenv("CONTENT_LENGTH"));
buf = (char *) malloc(n+1);
memset(buf,'\000',n+1);
for(i=0;i<n;i++)
{
buf[i]=getchar();
}

free(buf);
```

В данном фрагменте применено динамическое размещение памяти в скрипте, поэтому при выходе из него память следует освободить. Вообще

говоря, память будет автоматически освобождена операционной системой после завершения скрипта. Однако, если переносить скрипт на спецификацию FCGI (Fast CGI), что требует минимума переделок, из-за неаккуратной работы с памятью могут возникнуть проблемы.

Механизм генерации отклика скриптом

Существует только один способ вернуть данные серверу и, соответственно, браузеру пользователя — писать в поток стандартного вывода (STDOUT). При этом скрипт должен формировать HTTP-сообщение.

Сначала выводятся директивы HTTP-заголовка. В минимальном варианте это либо

```
Content-type: text/html,
```

либо

```
Location: http://intuit.ru/
```

В первом случае определяется тип тела HTTP-сообщения, а во втором осуществляется перенаправление запроса.

После заголовка генерируется отклик в виде тела HTTP-сообщения, которое должно быть отделено от заголовка пустой строкой:

```
#!/bin/sh
echo Content-type: text/plain
echo
echo Hello
```

В данном случае используется командный интерпретатор sh.

Если скрипт начинает формирование заголовка с директивы версии HTTP-протокола, то сервер не анализирует отклик и передает его как есть. Если в заголовке, сгенерированном скриптом, эта директива отсутствует, то сервер считает, что заголовок неполный, и вставляет в него дополнительные директивы.

2: Введение в программирование CGI-скриптов и программирование скриптов на bash

При обсуждении обмена данными между клиентом и сервером в Web-технологии логично было бы рассмотреть вопросы разработки прикладного программного обеспечения на стороне сервера. Как правило, это CGI-скрипты.

К наиболее популярным средствам разработки таких скриптов относятся:

- shell (командный язык);
- Perl;
- C.

Тем, кто имеет представление об этих языках, достаточно выполнить контрольные тесты и перейти к целевому программированию скриптов. Для тех же, кто смутно представляет себе процесс программирования, изучение данного раздела учебного курса обязательно. Раздел содержит много простых и полезных программ, которые применяются для отладки более сложных скриптов или являются их составными частями.

Командные языки являются тем первым инструментом программирования, который попадает в руки любого пользователя. В Windows это cmd (речь идет об Windows NT или Windows 95), в Unix — различного рода shell. cmd оставим для учебных курсов Microsoft и сосредоточимся на командных языках Unix.

Среди различных командных языков оболочек (shell) выберем тот, который является общим для большинства Unix-платформ — GNU bash (Bourne Again Shell). Прообраз bash — самый первый shell (sh), поэтому bash наследует многие его свойства.

Для программирования CGI-скриптов bash удобен тем, что наглядно демонстрирует многие свойства окружения среды Unix, которые используются и в других системах программирования. Кроме того, часто программирование на командном языке применяется для сравнительных описаний программ разработанных на C или Perl.

Структура bash-скрипта

Для того чтобы выполнить bash-скрипт, требуется интерпретатор bash. При этом скрипт запускается HTTP-сервером и, в общем случае, не определяет его операционное окружение (точнее, оно определяется окружением сервера). По этой причине в начале файла скрипта следует указать, что для его исполнения требуется интерпретатор bash:

```
#!/usr/local/bin/bash
echo Hello BASH
```

Первая строчка этой записи указывает на то, что содержание файла будет рассматриваться как программа (скрипт) на bash. Кстати, bash эту конструкцию также воспринимает как запуск программы интерпретации содержимого файла, поэтому в процессе выполнения скрипта для отдельных операций можно вызывать другие скрипты со своими интерпретаторами.

В общем случае символ "#" рассматривается как начало комментария, который распространяется до конца строки. При программировании скриптов его чаще всего приходится употреблять для маскирования строк программы во время отладки.

Более bash-скрипт ничем не выделяется. Команды bash обычно вводятся каждая на отдельной строке. Если это по каким-то причинам затруднительно, то команды разделяются символом ";". Исключение составляют конвейеры: в них команды находятся в пределах одной строки и разделены символом "|".

При программировании на bash нужно четко различать команды, встроенные в bash, и команды операционной системы. Например, echo — это команда операционной системы, a let — встроенная команда bash.

Стандартный поток вывода

Собственно, сам командный язык bash не имеет механизма организации вывода данных. Среди встроенных в bash команд нет команды печати. Но зато можно воспользоваться командами Unix. Самой простой из них является команда echo, которая копирует свои аргументы в поток

стандартного вывода. При этом объединять разные слова во фразу каким-либо образом не нужно:

```
bash>echo Perl meets CGI
Perl meets CGI
bash>
```

В данном случае echo вывела три своих аргумента и символ перевода строки — приглашение (prompt) bash находится на новой строке.

На первый взгляд, такое простое решение для стандартного вывода кажется примитивным. На самом деле, его вполне достаточно для генерации HTML-страниц. Механизмы, которые делают echo в совокупности с bash эффективным средством генерации отчетов в HTML-формате, таковы:

- подстановка переменных (substitution);
- маскирующие кавычки (quoting);
- подстановка результатов выполнения команд.

В совокупности они представляют собой мощный инструмент.

Substitution позволяет формировать строку вывода путем включения в нее значений переменных. Например, если нужно распечатать позиционные параметры скрипта, сделать это можно следующим образом:

```
echo first_arg#$1 second_arg#$2
```

В данном случае распечатываются первый и второй аргументы командной строки скрипта. Другой пример — распечатка переменной окружения:

```
echo QUERY_STRING:$QUERY_STRING
```

Quoting используется для маскирования специальных значений некоторых символов. Такие символы называют метасимволами. Например: ">" и "<" — это символы перенаправления потоков ввода-вывода и, следовательно, их надо маскировать при выводе. Для такого маскирования проще всего использовать простые одинарные кавычки:

```
echo '<H1>QUOTING</H1>'
```

В данном случае мы напечатаем заголовок первого уровня в HTML-документе. При маскировании следует помнить, что внутри кавычек bash

не выполняет интерпретации кода скрипта, поэтому переменные внутрь одинарных кавычек вставлять нельзя:

```
echo '<H1>'$QUERY_STRING'</H1>'
```

В данном случае строка вывода будет состоять из трех частей: тега начала заголовка, значения переменной QUERY_STRING и тега конца заголовка.

Подстановка результата выполнения команды осуществляется с использованием обратных кавычек (традиционный вариант) или формы $ (command). При этом в строку вывода включается значение, которое возвращает выполненная команда:

```
echo '<H1>'`date`'</H1>'
```

или

```
echo '<H1>'$(date)'</H1>'
```

Таким образом можно вставлять не только отдельные команды, но и целые последовательности команд. Главное, чтобы эта последовательность что-нибудь возвращала.

Переменные окружения

Переменные окружения (оболочки) создаются в момент старта bash-скрипта. При этом существует два типа переменных — те, которые действуют только в данной оболочке, и те, которые наследуются извне. Для просмотра переменных окружения можно использовать команду set:

```
bash-2.01$ set
bash=/bin/bash
bash_versinfo=([0]="2" [1]="01" [2]="0"
  [3]="1" [4]="release"
  [5]="i386-pc-freebsd2.2.2")
bash_version='2.01.0(1)-release'
columns=106
dirstack=()
euid=1010
...
```

Здесь не приводится полный список всех переменных окружения. Показано только, как этот список отображается. Каждая переменная передается

парой "имя=значение". При этом каждая такая пара записывается с новой строки. Попробуем распечатать все переменные окружения скрипта в виде HTML-таблицы, используя bash:

```
#!/usr/freeware/bin/bash
echo Content-type: text/html
echo
echo '<HTML><HEAD></HEAD><BODY>'
echo '<H1>переменные окружения</H1>'
echo '<TABLE BORDER=1>'
echo '<TR><TD>Имя</TD><TD>значение</TD></TR>'
IFS='='
set | while read x y
do
echo '<TR><TD>'$x'</TD><TD>'$y'</TD></TR>'
done
echo '</TABLE>'
echo '<HR>'
echo '</BODY></HTML>'
```

Первой командой echo формируется предложение HTTP-заголовка. Вторая команда echo обеспечивает пропуск строки между заголовком HTTP-сообщения и его телом. Затем начинает формироваться тело HTML-документа. Обратите внимание на прямые одинарные кавычки "'". Они применяются для того, чтобы защитить от интерпретации угловые скобки "<" и ">", которые используются в bash для перенаправления стандартных потоков ввода/вывода.

Далее присваивается значение переменной окружения bash, которая не генерируется сервером HTTP — IFS. Переменная IFS хранит список символов-разделителей слов. По умолчанию это пробел и табуляция. Но нам нужно разделить имя переменной и его значения, которые на самом деле разделены символом "=".

Теперь вызываем команду set. При этом ее стандартный поток вывода перенаправляем при помощи "|" команде read, которая считывает строку из стандартного ввода, при этом присваивая переменным x и y значения последовательно от начала строки выделенных слов. А слова мы разделяем символом "=".

Читаем стандартный ввод в цикле while условие do.... done. В качестве условия все та же команда read — если считываем данные, то "истина", если нет, то — "ложь". При этом внутри цикла выводим строки таблицы

"имя — значение".

В конце скрипта приводим документ к стандартному виду HTML-документа.

Обратиться к значению переменной окружения можно, конечно, гораздо проще — по имени:

```
#!/usr/freeware/bin/bash
echo Content-type: text/html
echo
echo '<HTML><HEAD></HEAD><BODY>'
echo '<H1>QUERY_STRING</H1>'
echo QUERY_STRING = $QUERY_STRING
echo '<HR>'
echo '</BODY></HTML>'
```

Здесь по команде echo будет просто распечатано значение переменной окружения QUERY_STRING.

Аргументы командной строки

Позиционные параметры или аргументы командной строки — это последовательность строковых констант, которые указываются в командной строке после имени скрипта. Любая встроенная в bash команда или команда Unix может запускаться с набором этих параметров. Например, для того, чтобы подсчитать число активных в данный момент процессов httpd, администратор систем выдает такую последовательность команд:

```
bash>ps -ax | grep httpd | wc -l
```

Здесь указано три команды, организованные в конвейер. Каждая из них имеет по одному аргументу командной строки:

- ps задана с аргументом -ax;
- grep задана с аргументом httpd;
- wc задана с аргументом -l.

Позиционные параметры (аргументы командной строки) задаются встроенными переменными $1 — $n, где n — число аргументов. Аргументы командной строки появляются при запросах типа ISINDEX. Число аргументов командной строки определяется встроенной переменной

bash — $#. Если мы вызовем скрипт по ссылке типа:

```
http://www.intuit.ru/cgi-bin/
  argv.cgi?arg1+arg2+arg3,
```

то переменная $# примет значение 3, а переменные: $1 — arg1, $2 — arg2, $3 — arg3. Кстати, $0 — это имя самого скрипта. Распечатка параметров в виде HTML-таблицы может выглядеть следующим образом:

```
#!/usr/freeware/bin/bash
echo Content-type: text/html
echo
echo '<HTML><HEAD></HEAD><BODY>'
echo '<H1>Аргументы</H1>'
echo '<TABLE BORDER=1>'
echo '<TR><TH>Номер</TH><TH>Значение</TH></TR>'
let i=0
for x in $@
do
let i=i+1
echo '<TR><TD>arg['$i']</TD><TD>'$x'</TD></TR>'
done
echo '</TABLE>'
echo '</BODY></HTML>'
```

Последовательность команд echo формирует HTTP-сообщение. Команда let позволяет выполнять арифметические вычисления. Перед циклом for производим инициализацию переменной i. Цикл for "пробегает" по всем аргументам командной строки, которые объединены в переменной $@ и разделяются в ней пробелами. Фактически они представляют собой список слов, по которому и бежит переменная цикла x. Обратите внимание на отличие данного цикла от стандартного цикла for в C или Perl: в нем не используются арифметические операции, а идет работа со списком.

Внутри цикла при помощи команды let мы увеличиваем индекс аргумента командной строки (значение переменной i) и распечатываем этот индекс и значение переменной x в виде элементов HTML-таблицы.

Если аргументов мало и их местоположение известно, то к каждому из них можно просто обращаться по встроенному имени, например, первый аргумент — это $1.

Стандартный поток ввода

По большому счету, для чтения данных из стандартного потока ввода в рамках программирования CGI-скриптов bash непригоден. Дело в том, что в нем нет механизма посимвольного считывания данных. Bash-скрипт способен читать только строками и останавливает считывание лишь в случае появления в потоке символа конца файла. Как известно, HTTP-сервер такого символа в стандартный поток ввода скрипта при работе по методу POST не передает. Тем не менее чтение стандартного ввода в рамках программирования CGI-скриптов на bash применяется.

Примером тому может служить генерация гипертекстовых ссылок на файлы текущего каталога:

```
#!/usr/freeware/bin/bash
echo Content-type: text/html
echo
echo '<HTML><HEAD></HEAD><BODY>'
echo '<Ul>'
ls -a | while read x
do
if test -f $x; then
echo '<LI><A HREF=./'$x'>'$x'</A>';
fi
done
echo '</BODY></HTML>'
```

В данном случае команда ls доставляет в скрипт имена файлов. Один файл — это отдельная строка. Эти имена обрамляются гипертекстовыми ссылками и вставляются в HTML-страницу. При этом печатаются только обычные файлы, все остальные игнорируются.

Другой пример — фильтрация. При приеме по методу GET запрос размещается в переменной QUERY_STRING. Но он там находится в форме form-urlencoded. Для его фильтрации вызывается внешняя программа, стандартный вывод которой перенаправляется на стандартный ввод одной из команд скрипта:

```
echo $QUERY_STRING | tr '+' ' ' | while read x
do
for y in $x
do
echo $y
done
done
```

Существуют и другие способы применения чтения из стандартного ввода при программировании CGI-скриптов на BASH.

Типы данных и переменные

В bash существует только два типа данных: скаляры и одномерные массивы. При этом возможно вычисление арифметических выражений, результат выполнения которых становится значением скаляра. По-другому эти типы можно интерпретировать как текстовые строки и списки.

Существует два типа переменных: встроенные переменные bash и переменные, определяемые пользователем (переменные пользователя). Не перечисляя всех встроенных переменных, назовем наиболее употребительные:

```
$1-$n – аргументы командной строки скрипта;
$0 – имя скрипта;
$@ – список аргументов командной строки;
$# – число аргументов командной строки;
$IFS – список разделителей;
$PATH – путь поиска команд.
```

Переменные окружения, которые генерируются сервером — это переменные пользователя, импортируемые скриптом при его запуске. Пользователь внутри скрипта может установить собственные переменные:

```
IFS="="
```

В данном случае мы отменили значение по умолчанию для списка разделителей и назначили в качестве разделителя знак равенства "=". IFS — это глобальная переменная, поэтому она передается от скрипта к скрипту по умолчанию. Если требуется назначить собственную переменную и передать ее в другой скрипт, который вызывается из текущего скрипта, ее нужно будет экспортировать:

```
bash>QUERY_STRING=arg1+arg2+arg3;
export QUERY_STRING
```

В данном случае в целях отладки скрипта в командной строке bash определена переменная окружения QUERY_STRING. Если запустить скрипт без предварительного экспорта, то значение этой переменной

($QUERY_STRING) будет неопределенным. Команда export позволяет передать это значение в тестируемый скрипт.

Управление потоком вычислений

Изо всех возможностей управления порядком выполнения команд в bash-скрипте мы рассмотрим только if, while и for. Пользуясь этими встроенными возможностями bash, следует иметь в виду, что логические выражения, которые применяются в качестве условий данных команд, строятся вокруг строк, а не чисел. Использовать числовое условие в bash крайне затруднительно.

if

Команда if имеет вид:

```
if list; then list; [elif list; then list;]
   ...[ else list;] fi
```

Сначала выполняется список команд, который стоит после if. Если он завершился успешно, то выполняется список команд после первого then. Значение и логика выполнения других частей этой команды очевидна. Команда начинается символами "if" и должна закончиться символами "fi". Часть команды в квадратных скобках — это необязательные конструкции, которые при необходимости можно опустить.

Рассмотрим в качестве примера проверку метода доступа к скрипту. Для bash это может быть только GET:

```
#!/usr/freeware/bin/bash
echo Content-type: text/plain
echo
if test $REQUEST_METHOD = "POST"; then
echo POST;
elif test $REQUEST_METHOD = "GET"; then
echo GET;
else echo Unknown method $REQUEST_METHOD;
fi
```

В данном случае мы используем сравнение строк (символ "="). Если нужно сравнивать арифметические выражения, то следует использовать другие

операции сравнения:

```
-eq – равенство операндов;
-ne – неравенство операндов;
-lt – первый операнд меньше второго;
-le – первый операнд меньше либо равен второму;
-gt – первый операнд больше второго;
-ge – первый операнд больше либо равен второму.
```

Команда test чрезвычайно полезна при работе с файловой системой. Например, при проверке наличия файла и прав на чтение можно использовать следующую комбинацию:

```
if test -r file.txt;
  then echo file.txt is readable; fi
```

Помимо проверки наличия файла и прав можно определять тип файла (-d — каталог, -f — обычный файл и т.п.).

while

Команда while позволяет выполнять список команд до тех пор, пока справедливо условие использования данного списка, которое задается аргументом while. Чаще всего в наших примерах эта команда применяется при фильтрации входного потока:

```
ps -axj | grep httpd | while read id pid
do
if test $id = "root"; then kill -1 $pid; fi
done
```

В данном случае в системе FreeBSD просматривается список активных процессов с именем httpd (HTTP-сервера), отыскивается процесс-родитель и перезапускается.

for

Вид команды for в bash отличается от обычного; когда в команде инициализируется переменная цикла, происходит проверка условия для переменной цикла и производится изменение ее значения. В bash переменная бежит по списку и выполняет цикл до тех пор, пока список не

будет исчерпан:

```
for var; in list; do list; done
```

Переменная var принимает значения из списка, указанного за in, до тех пор, пока этот список не кончится. При этом для каждого значения var выполняется список команд, заключенный между "do" и "done". Примером использования for может служить разбор входных строк:

```
ls -ax | while read x
do
for y in $x
do
echo $y
done
done
```

Считываемая из стандартного ввода строка разбивается на слова, и каждое слово печатается отдельно на новой строке.

3: Введение в программирование на Perl

Язык программирования Perl является основным средством разработки CGI-скриптов для Web-узлов. Его не применяют только там, где требуется высокая эффективность кода и нет стандартных библиотек для Perl.

Perl как язык разработки скриптов имеет ряд преимуществ. Перечислим их в порядке значимости:

- независимость от программно-аппаратной платформы;
- мощные средства разбора строк;
- простота работы с переменными окружения;
- простота работы со входными и выходными стандартными потоками;
- возможность чтения заданного числа символов из входного потока;
- хешированные таблицы;
- возможность организации конвейеров;
- библиотеки TCP/IP-обмена;
- множество стандартных библиотек прикладных программ.

Все это делает программное обеспечение, написанное на Perl, мобильным, а разработку программ — быстрой и простой.

Структура Perl-программы

У программы на языке Perl нет жестко заданной структуры. Точнее сказать, программист не обязан ее соблюдать. Если же он захочет, чтобы интерпретатор контролировал объявление и использование переменных и конструкций языка, то при помощи класса strict он может такой контроль установить.

Программа на Perl состоит из операторов языка, которые должны заканчиваться символом ";". Например:

```
print "Привет, Perl.";
```

В общем случае операторы делятся на простые и составные. Простой оператор — это оператор print, например, а составной — while():

```
while() {s/</&lt;/g; s/>/&gt;/g; print $_;};
```

Программа на Perl исполняется интерпретатором Perl. Есть и компиляторы с этого языка, но они используются реже. При программировании CGI-скриптов в Unix интерпретатор вызывается из того же файла, который содержит программу, например:

```
#!/usr/local/bin/perl
print "Привет, Perl.";
```

В данном случае первая строка — это вызов интерпретатора с указанием полного пути к нему от корневого каталога файловой системы.

Вообще говоря, символ "#" — это символ начала комментария в Perl. Последовательность символов от символа "#" до конца строки рассматривается как комментарий. Часто в качестве комментария используют целые строки. В этом случае символ "#" ставится в первой позиции строки:

```
#!/usr/local/bin/perl
#
#Печатаем HTML-заголовок
#
print "Content-type: text/html\n\n";
#
#Содержание документа
#
print "Perl и CGI.";
```

Данный пример демонстрирует не только использование комментариев, но еще и формирование HTTP-заголовка. Без этого заголовка система выдаст сообщение о внутренней ошибке сервера, а в файле журнала ошибок появится запись о неправильном заголовке.

В последнее время основным рабочим местом авторов HTML-страниц и CGI-скриптов стали системы на платформе Windows. Как известно, конец строки в Windows и в Unix обозначается разными последовательностями неотображаемых символов. Если автор использует текстовый редактор в Windows, а потом как binary копирует файл в Unix, то эти символы передаются. Сервер начинает сообщать об ошибках, которые автору не

видны. В таком случае можно сделать следующее: либо копировать программы как char, тогда происходит перекодировка, либо использовать "умные" текстовые редакторы.

Удобна также и Samba, которая позволяет редактировать "по месту" из Windows в Unix.

Стандартный поток вывода

Основная цель создания CGI-скрипта — обработка данных запроса пользователя и формирование отклика сервера на этот запрос. Можно рассматривать и другие задачи, которые позволяют решать CGI-скрипты, но эта задача — главная. Скрипт должен формировать не просто отклик, а HTTP-отклик. Это означает, что он должен сформировать заголовок и тело HTTP-сообщения, которые отделяются друг от друга пустой строкой.

Проще всего реализовать такой отклик с помощью команды print:

```
#!/usr/local/bin/perl
print "Content-type: text/html\n\n";
print "<HTML><HEAD></HEAD><BODY>";
print "<H1>Perl и CGI.</H1>";
print "</BODY></HTML>";
```

Первая строчка определяет заголовок HTTP-отклика и пустую строку (два символа "\n\n"). Остальные операторы print формируют тело сообщения. Оператор print пишет в стандартный вывод список своих аргументов. Вообще говоря, print можно использовать для вывода данных в любой файл. Если имя файла не указано, то вывод осуществляется в стандартный поток вывода.

Переменные окружения

Скрипт порождается сервером в некоторой операционной среде. Эта среда называется окружением (environment). Частью окружения являются так называемые переменные окружения. При вызове скрипта их порождает и присваивает им значения HTTP-сервер. Список переменных определен спецификацией Common Gateway Interface.

В Perl существует встроенный системный массив переменных окружения %ENV. Символ "%" перед именем массива означает, что это массив ассоциативный, т.е. значение элемента массива может быть выбрано путем указания ключа, с которым оно связано. Например, нужно определить метод доступа к скрипту:

```
#!/usr/local/bin/perl
print "Content-type: text/plain\n\n";
print "REQUEST_METHOD:$ENV{REQUEST_METHOD}";
```

В данном контексте символ "$" перед именем массива не должен вводить в заблуждение. Обращение происходит к элементу ассоциативного массива. Это скаляр, поэтому и применяется символ "$". Ключ, по которому выбирается значение, указан в фигурных скобках.

Очень полезен скрипт распечатки всех переменных окружения, которые переданы скрипту (perlenv2.htm). Обычно этот отчет получают при отладке HTML-форм:

```
#!/usr/local/bin/perl
print "Content-type: text/plain\n\n";
foreach $hkey (keys %ENV)
{
 print "$hkey:$ENV{$hkey}";
}
```

Этот простой скрипт распечатывает значения всех переменных окружения, используя цикл foreach. В этом цикле переменная цикла hkey пробегает по всем уникальным ключам (именам переменных окружения), которые доставляет функция keys.

Аргументы командной строки

Когда пользователь работает с операционной средой в режиме удаленного алфавитно-цифрового монитора, он пользуется услугами оболочки (shell). Команды в операционной среде в этом случае вводятся в командной строке, и за каждой из команд может тянуться шлейф аргументов. Эти аргументы и называются аргументами командной строки.

CGI-скрипт вызывается не из оболочки, а загружается HTTP-сервером. Если необходимо воспользоваться аргументами командной строки, то сервер должен породить данную командную строку тоже. В CGI это делается только для запросов ISINDEX. В таком случае скрипт вызывается через URI типа:

```
http://my.intuit.ru/directory/
     script?arg1+arg2+arg3
```

В этой записи arg1+arg2+arg3 — аргументы командной строки скрипта, т.е. данный URI равнозначен вводу в командной строке команды:

```
host>script arg1 arg2 arg3
```

Для приема этих аргументов достаточно воспользоваться скриптом типа:

```
#!/usr/local/bin/perl
print "Content-type: text/plain\n\n";
foreach $arg (@ARGV)
{
 print "$arg\n";
}
```

В данном случае мы просто их распечатаем. Программа будет выбирать по одному аргументу из системного массива аргументов командной строки @ARGV и помещать их в переменную $arg, а затем печатать.

В принципе, аргументы попадают и в переменную окружения QUERY_STRING, т.к. при запросе типа ISINDEX применяется метод GET. Но тогда придется данную переменную разбирать. Аргументы командной строки уже сделали эту работу за программиста.

Стандартный ввод

Поток стандартного ввода обычно ассоциируется с клавиатурой терминала. Поток стандартного ввода — это источник входных данных по умолчанию. В C, например, поток стандартного ввода ассоциируется с файлом STDIN. В Perl применяется то же самое имя.

Для построчного чтения входного потока в Perl применяется пара символов

"<>". Простая программа, читающая входной поток, может выглядеть следующим образом:

```
#!/usr/local/bin/perl
while()
{
 print $_;
}
```

В данном примере две скобки подряд определяют чтение из потока стандартного ввода. Магическая последовательность $_ обозначает системную переменную, в которую по умолчанию помещается каждая считанная из стандартного ввода строка. При этом символы конца строки сохраняются.

Согласно спецификации CGI, скрипт получает данные через стандартный ввод в том случае, если в качестве метода доступа в форме будет указан POST. Приведенный выше пример работы со стандартным потоком ввода не может быть применен для обработки такого запроса. Дело в том, что сервер не закрывает поток ввода и, следовательно, не передает EOF (End of file, конец файла). Цикл while в этом случае будет бесконечным.

Скрипт сам определяет точное количество байтов, которые он должен прочитать из стандартного ввода. Это можно сделать, прочитав значение переменной CONTENT_LENGTH (perlm1.htm):

```
#!/usr/local/bin/perl
print "Content-type: text/plain\n\n";
print "CONTENT_LENGTH=$ENV{CONTENT_LENGTH}";
```

Когда число байтов для чтения из потока стандартного ввода определено, эти байты нужно считать. Воспользуемся функцией read:

```
#!/usr/local/bin/perl
print "Content-type: text/plain\n\n";
print "CONTENT_LENGTH=$ENV{CONTENT_LENGTH}\n";
read STDIN,$query,$ENV{CONTENT_LENGTH};
print "Query:$query.";
```

После выполнения этих несложных операций в переменную $query будет занесено содержание стандартного ввода скрипта, с которым потом можно будет разбираться

Типы данных и переменные

В Perl существует довольно своеобразный набор типов данных: скаляры, одномерные массивы (массивы), ассоциативные массивы (хешированные таблицы или хеши), глобальные символы, ссылки. Обычно первых трех для CGI-программирования вполне хватает. Ссылки применяют в совокупности со стандартными библиотеками и вместо функции отложенного выполнения eval. Глобальные символы нужны при работе с файлами или для межпроцессного обмена данными при открытии каналов (потоков данных) через дескрипторы.

Скаляры

Скаляры — это все, что нельзя записать в виде массива или структуры. Числа всех типов и строки символов относятся к скалярному типу. Скаляр обозначается символом "$" перед именем. Примеры скаляров:

```
$a = 1;
$b = 2.5;
$str = "это строка символов";
```

Одномерные массивы

Массивы — это множество элементов, причем все элементы могут быть разнородными. В Perl нет понятия массива чисел или массива строк. Массив не может быть многомерным. Можно реализовать массив массивов и, таким образом, сделать массив двумерным. Массив обозначается символом "@" перед именем. Примеры массивов:

```
@a = (1,2,3);
@b = (4,5,6,2.5,"test");
@c = (@a,@b);
@in =
$q = $c[3]; # 4
```

Индексирование элементов массива начинается с цифры 0. Поэтому в нашем примере четвертый элемент массива с будет индексироваться как $c[3].

Ассоциативные массивы

Ассоциативные массивы — это двухколоночная таблица. Первая колонка — ключ, а вторая колонка — связанное с ним значение. Ассоциативные массивы называют еще хешированными таблицами или просто хешами. Дело в том, что значение можно извлечь из таблицы прямо, указав ключ. При этом используется алгоритм хеширования по ключу.

Для краткости будем называть ассоциативный массив хешем. Обозначается переменная типа хеш символом "%" перед именем. Примеры хешей и обращений к их элементам:

```
%a = ("test",1,"test2",2);
$c = $a{test}; # $c=1
```

В первом случае хеш инициируется как обычный массив. Во втором примере мы выбираем значение определенным ключом "test". Результат этой операции, скаляр, помещается в переменную $c.

Указатели

Указатели — это аналог адресных указателей в C. Указатель обозначается символом "\" перед именем переменной:

```
$a = 1;
$p = \$a;
@b = (1,2,3);
$p = \@b;
$p = \%c;
```

Обращения к значениям, выбранным указателем, являются аналогом применения функции eval (отложенное исполнение):

```
%a = ("test",1,"test2",2);
$p = \%a;
$c = $$p{"test"};
```

где вместо "a" используется $p.

Глобальные символы

Последнее, что мы рассмотрим — это дескрипторы потоков данных. Дескриптор потока указывает на структуру, которая описывает механизм обмена данными, например, с файлом. Применяются такие дескрипторы в функциях чтения и записи данных и в межпроцессном обмене:

```
open IN,"
read IN,$p,500;
close IN;
```

В данном случае IN — дескриптор файла.

Регулярные выражения (сопоставление с образцом)

Самым интересным средством Perl является механизм регулярных выражений, сравнение с образцом и подстановки, которые он унаследовал от таких типичных для Unix-систем утилит, как awk и sed.

Операция сопоставления с образцом задается как "=~":

```
$query =~ /target/;
```

В данном случае в скаляре $query отыскивается подстрока "target". С операцией "=~" связано два действия по умолчанию. Во-первых, операция сопоставления в полной форме будет записана как

```
$query =~ m/target/;
```

То есть перед шаблоном используется префикс m. Во-вторых, если опустить $query, то образец будет сопоставляться с содержанием системной переменной $_:

```
/target/;
```

Операция "=~" возвращает значение "истина"(true), если образец был найден, и "ложь"(false), в противном случае. Из этого следует, что данную операцию можно использовать в операторах ветвления:

```
if($query =~ /target/)
```

```
{
print $query;
}
```

В данном случае, если образец содержится в $query, то запрос печатается, не содержится — пропускается при печати.

При сопоставлении с образцом одновременно осуществляется разбор запроса: шаблон попадает в переменную $_, часть строки до найденного шаблона — в $`, а после образца — $'. Но для программирования разбора данных из формы больше подойдет следующий пример (perlexp1.htm):

```
http://intuit.ru/scripts/
script?n1=v1&n2=v2&n3=v3
...
($v1,$v2,$v3) = ($query =~
/^n1=(.+)&n2=(.+)&n3=(.+)$/);
print $v1,$v2,$v3;
```

В переменные $v1 — $v3 будут записаны значения полей n1-n3, соответственно. При этом символ "^" означает, что весь шаблон должен стоять вплотную к началу строки. Символ "$" в конце шаблона означает, что шаблон должен распространяться на всю строку до конца. Символ "." означает любой символ, входящий в таблицу символов, "+" означает любое количество этих символов, но не менее одного.

Еще более мощным средством является подстановка, которая базируется на механизме сопоставления с образцом. Для ее осуществления следует использовать префикс s%

```
$query =~ s/intuit/ruru/;
```

В данном случае, если в строке будет найден шаблон intuit, он будет заменен на ruru. Наиболее очевидной областью применения подстановки является перекодировка запросов form-urlencoded:

```
$query =~ s/%(.{2})/pack('c',hex($1))/eg;
```

Спецсимволы и символы из второй половины таблицы ASCII при передаче

трансформируются в шестнадцатеричные описания байтов (две цифры), следующие после символа "%".

Управление потоком вычислений

Под управлением потоком вычислений понимают способность программы в зависимости от условий выполнять те или иные части кода и повторять различные части кода. Для этой цели в Perl включены такие операторы как goto, next, last, redo, if, while, for, foreach и другие.

Оператор GOTO

Оператор goto позволяет перейти к исполнению оператора Perl, выделенного меткой, которая указывается в качестве аргумента оператора goto:

```
while()
{
if(/the end/) {goto out;};
}
out: print "the_end\n";
```

В данном случае выход из цикла осуществляется по оператору goto. Вслед за всеми современными учебниками, напомним, что использование goto — это плохой стиль программирования. Метка в Perl задается как строка, за которой вплотную следует символ двоеточия(":").

Оператор while

В нашем примере встретилось еще несколько конструкций управления потоком вычислений. Первая из них — оператор while. While определяет цикл, который исполняется до тех пор, пока значение выражения, указанного в качестве аргумента оператора while, — "истина". Пример:

```
while($line =~ /the_end/)
{
$line = <>;
print "No";
}
print "the_end\n";
```

В данном случае оператор goto не используется. Выход из цикла осуществляется по условию вхождения подстроки the_end в строку ввода. Вхождение проверяется путем сопоставления с образцом. Входная строка считывается из стандартного ввода (операция <>).

Оператор for

Оператор for — это традиционный оператор цикла. Типовая схема применения данного оператора может быть показана на следующем примере:

```
for($i=0;$i<CONTENT_LENGTH;$i++)
{
$query[$i] = getc;
}
$q = join (@query);
print $q;
```

Здесь данные в цикле считываются посимвольно из стандартного потока ввода в массив @query. Число символов, которые нужно считать, определяется переменной окружения CONTENT_LENGTH. После завершения ввода массив объединяется в строку $q при помощи функции join. Наиболее типичной ошибкой при использовании оператора for является отсутствие символа "$" у переменной цикла. Все три аргумента оператора for могут быть выражениями соответствующего типа.

Оператор foreach

Оператор foreach позволяет организовать цикл путем перебора элементов списка. В качестве такого списка можно использовать массив:

```
foreach $arg (@ARGV)
{
print $arg;
}
```

В данном случае переменная цикла $arg пробегает по всем аргументам командной строки скрипта, которые задаются встроенным массивом @ARGV. Аналогично можно пройти и по всем переменным окружения:

```
foreach $arg (keys %ENV)
{
print "$arg:$ENV{$arg}\n";
}
```

Отличие данного примера от предыдущего заключается в том, что переменные окружения представляют собой хеш. Поэтому сначала получаем массив ключей (имен переменных окружения), переменная $arg пробегает по этому массиву, и в цикле печатаются пары "имя переменной — значение переменной".

Оператор if

После goto и операторов цикла следует остановиться на операторе if. Данный оператор позволяет организовать ветвление программы или, как это еще называют, условное исполнение отдельных ее частей. Примером может служить скрипт обработки данных, которые могут доставляться как по методу GET, так и по методу POST:

```
if ($ENV{REQUEST_METHOD} =~ /POST/)
  {
  read STDIN,$query,$ENV{CONTENT_LENGTH};
  }
else
  {
  $query = $ENV{QUERY_STRING};
  }
```

В данном случае анализируется значение переменной окружения REQUEST_METHOD. В зависимости от ее значения либо включается чтение данных из стандартного потока ввода, либо считывается значение переменной окружения QUERY_STRING.

Оператор next

Если оператор goto не применять (концепция структурного программирования), то при работе с циклами нужно обрабатывать ситуации пропуска конца тела цикла и досрочного выхода из тела цикла. Пропуск операторов конца тела цикла осуществляется при помощи оператора next:

```
foreach $arg (@strings)
{
if(!($ENV{REMOTE_HOST} =~ /\.intuit\.ru/))
  next;
print $arg;
}
```

Здесь фрагмент скрипта печатает значения массива @string, только если скрипт вызван с компьютера из домена intuit.ru. Машина пользователя будет прописана в этом домене только в том случае, если она прописана в обратной зоне, т.е. есть не только прямое соответствие "имя — IP-адрес", но и обратное "IP-адрес — имя". При этом сервер должен выполнять поиск доменных имен компьютеров пользователей по их IP-адресам (режим nslookup).

Оператор last

Досрочный выход из тела цикла производится по оператору last. Обычно такой выход осуществляется по некоторому событию:

```
open IN,"<cont.txt";
while()
  {
  if($_ =~ /index\.htm/) last;
  #разбор строки
  }
close IN;
```

В данном примере скрипт считывает строки из файла счетчика посещений. Если в строке встречается имя файла "index.htm", то происходит досрочный выход из цикла.

Оператор redo

Оператор redo применяется для повторения последней итерации цикла. Например, если есть вложенные циклы while, которые выполняют одинаковые последовательности операторов, то внутренний из них можно заменить на оператор if с redo.

Файлы, каталоги, конвейеры, сокеты

При выполнении различных операций в CGI-скриптах часто приходится работать с файловой системой Web-узла. Perl обеспечивает довольно разнообразный инструментарий для такой работы. Условно его можно разбить на четыре части:

- работа с файлами;
- работа с каталогами;
- работа с каналами (конвейеры);
- работа с сетевыми ресурсами через сокеты.

Файлы

Для работы с файлами в Perl применяют несколько встроенных функций. Открывают файл функцией open. При этом для указания действий с записями файла используют префиксы:

```
open IN,"<test.txt";
# открыть файл на чтение

open IN,">test.txt";
# открыть файл на запись

open IN,"+<test.txt";
# открыть файл на чтение и запись

open IN,">>test.txt";
# открыть файл на модификацию (добавление)
```

Закрывают файл при помощи функции close:

```
close IN;
```

Для чтения записей из файла используют либо чтение потоком — <file>, либо функцию read:

```
while(<IN>)
  {
  print $_;
  }

read STDIN, $query, $ENV{CONTENT_LENGTH};
```

В первом случае данные построчно считываются из файла IN и распечатываются в поток стандартного вывода. Во втором случае из потока стандартного ввода, который тоже является файлом, считывается функцией read $ENV{CONTENT_LENGTH} байтов. Для обработки потока стандартного ввода CGI-скрипта подходит только второй способ, так как сервер не закрывает потока ввода, что приводит к бесконечному ожиданию ввода и разрыву соединения по timeout.

Каталоги

Для работы с каталогами используют ряд встроенных функций: opendir, readdir, closedir. Первая открывает дескриптор файла каталога, вторая позволяет читать записи из файла каталога, третья закрывает дескриптор файла каталога:

```
opendir DIR,"/usr/user";
while($_=readdir(DIR))
  {
  next if -d;
  print $_;
  }
closedir DIR;
```

В данном примере распечатываются названия файлов из каталога "/usr/users". При этом имена каталогов не распечатываются. Такое поведение скрипта определяется модификатором if в операторе next.

Каналы

Одним из замечательных свойств командных языков является возможность использования конвейеров. Они организуются путем перенаправления стандартного потока вывода одной программы в стандартный поток ввода другой. Иногда следует некоторые данные из скрипта профильтровать через такой конвейер, а потом снова ими воспользоваться. Открыть файл оператором типа

```
open FILTER,"<cat;more>";
```

нельзя. С точки зрения логики, при такой форме записи один дескриптор

файла будет связан с разными потоками. Кроме того, поток стандартного ввода организует прием данных от сервера и уже занят. Возможность использования таких фильтров в Perl обеспечивает библиотека IPC. Запись при этом должна выглядеть следующим образом:

```
#!/usr/local/bin/perl
use IPC::Open2;
use FileHandle;
$pid = open2(\*RDR,\*WRD,"cat");
WRD->autoflush();
print WRD "test\n";
$got = <RDR>;
print "Это \$got:$got";
```

В данном случае для открытия канала связи между процессами используется функция open2. Ей необходимы три аргумента: указатель на дескриптор потока для чтения (*RDR), указатель на дескриптор потока для записи (*WRD) и строка внешней программы-фильтра ("cat"). Вместо cat можно указать любое множество команд, организованных в виде конвейера. Команда print посылает данные в этот конвейер, а команда "" считывает из него данные.

Работа с серверами Internet. Сокеты

Пусть и не очень часто, но все-таки приходится при разработке CGI-скриптов пользоваться возможностью обращения из скрипта к серверу, установленному на другом компьютере. При работе в сетях TCP/IP для этой цели используют библиотеку сокетов (Berkeley sockets) TCP/IP. Сокет — это пара "IP-адрес — номер порта". При программировании речь, конечно, будет идти о структуре данных, которая позволяет передать/принять данные с удаленного компьютера.

IP-адрес в сокете отвечает за доступ к определенному компьютеру в Internet. В общем случае IP-адреса закрепляются не за отдельными компьютерами, а за сетевыми интерфейсами. Один сетевой интерфейс может иметь несколько IP-адресов. Эта возможность используется при организации виртуальных Web-узлов.

Порт — это виртуальный канал приема/передачи данных, в котором происходит обслуживание приходящих из сети запросов. Этот канал

обозначается цифрой. Первые 256 каналов — это WKS (Well Known Services, "хорошо известные" сервисы). Например, за HTTP-обменом закреплен 80-й порт, за FTP-обменом – 20-й и 21-й порты, за DNS (Domain Name System, служба доменных имен) – 53-й порт и т.д.

Порты бывают двух типов, TCP и UDP (User Data Protocol, пользовательский протокол данных), т.е. соответствуют типу транспортного протокола, который используется для передачи данных. Мы будем рассматривать в качестве примера именно TCP, т.к. HTTP-обмен использует только TCP-порты. Слово "порты" во множественном числе здесь употребляется не случайно. Дело в том, что только первоначальное обращение к HTTP-серверу происходит по 80-му порту. Для того чтобы обслуживать много запросов практически одновременно, сервер переназначает порт, и клиент, например, браузер, общается с сервером уже по другому порту.

Кроме TCP/IP существуют еще и другие виды сокетов, поэтому при программировании нужно указывать тип сокета. Ниже приведен пример обращения из скрипта к серверу HTTP:

```
#!/usr/local/bin/perl
use IO::Socket;
$remote = IO::Socket::INET->new(
Proto=>"tcp",
PeerAddr=>"localhost",
PeerPort=>"80"
) or die "No service";
$remote->autoflush(1);
print $remote "HEAD / HTTP/1.0\n\n";
while()
{
print;
}
-close $remote;
```

В данном примере используется пакет Socket из библиотеки IO (Interpretive Operation, работа в режиме интерпретации) (оператор use). В третьей строчке примера происходит открытие дескриптора для сокета. Затем этот дескриптор применяется как обычный дескриптор файла, который открыт для записи и чтения.

Прежде чем писать данные в сокет, обычно отменяют буферизацию

(autoflush(1) — выталкиваем по одному байту). Делается это для того, чтобы данные сразу уходили на сервер. В противном случае сервер может разорвать соединение, не дождавшись запроса.

Оператор print демонстрирует запрос по протоколу HTTP 1.0 к HTTP-серверу. В этом запросе просто считывается документ из корня каталога HTTP-сервера. Обычно это файл index.html. Чтение отклика происходит в цикле while. После считывания отклика socket закрывается по функции - close.

Отложенное исполнение. Операция eval

Интерпретируемые языки программирования имеют две особенности, которых нет у компилируемых языков:

- рекурсия;
- отложенное исполнение.

С точки зрения эффективности кода это, конечно, не лучшие решения. Рекурсия может приводить к различного рода переполнениям, что, в свою очередь, вызывает крах программы, а отложенное исполнение не позволяет заранее оптимизировать код и в ряде случаев может приводить к рекурсивному вызову интерпретатора. Тем не менее изящность этих механизмов не может оставить программиста равнодушным.

Отложенное исполнение в Perl применяется в случае подстановки, в том числе рекурсивной

```
$sm =~ s/%(.{2})/pack('c',hex($1))/eg;
```

и в случае использования операции eval. Алгоритм работы этой операции достаточно прозрачен. В качестве аргумента в eval передается строка, которая затем рассматривается как Perl-программа. Для программирования из Web-браузера на Perl можно написать, например, такой скрипт:

```
#!/usr/local/bin/perl
read STDIN,$query,$ENV{CONTENT_LENGTH};
$query =~ s/%(.{2})/pack('c',hex($1))/ge;
$query =~ tr/+/ /ge;
$query =~ s/f=//;
```

```
eval $query;
```

Функция read в этом примере обеспечивает считывание данных из потока стандартного ввода скрипта. Операция рекурсивной подстановки позволяет преобразовать текст программ из формата form-urlencoded в ascii. Функция транслитерации tr заменяет символ "+" пробелом. Операция подстановки, следующая за транслитерацией, удаляет из кода программы имя поля и символ "=". Операция eval позволяет исполнить этот код. При этом предполагается, что данные передаются из формы с единственным полем типа textarea с именем "f".

Справедливости ради следует заметить, что в операции транслитерации кроме символа "+" нужно указать еще и другие символы, которые могут оказаться в коде программы. Например, если ввод кода осуществляется из браузера в среде MS-Windows, то при каждом переводе строки будет генерироваться символ "0D", который интерпретатор Perl для среды Unix не распознает.

Библиотеки

Программировать на Perl без применения дополнительных библиотек довольно сложно. В сети существует библиотека CPAN(http://www.perl.com/CPAN/). CPAN — это огромное хранилище полезного программного обеспечения. Для программирования CGI-скриптов в этой библиотеке имеются модули CGI-программирования. В последних версиях вместе с дистрибутивом Perl поставляется модуль CGI.pm. Но даже если этого модуля нет, его просто нужно скачать из CPAN.

Установка модуля в Perl довольно проста. Требуется выполнить следующие действия:

1. модуль в архивированном виде переписывается из CPAN:

   ```
   edu>ftp ftp.perl.com
   ```

2. создается временный каталог и модуль разархивируется в него:

   ```
   edu>gzip -d имя_модуля
   ```

3. в каталоге находят файл Makefile.pl и выполняют генерацию файлов для сборки модуля:

   ```
   edu>perl Makefile.pl
   ```

4. затем выполняют команды:

   ```
   edu>make; make test; make install
   ```

Теперь модуль установлен в Perl-каталог, и на него настроены все пути. С этого момента программы могут использовать функции из установленного модуля.

Включать функции из модуля в программу на Perl можно при помощи оператора use. Например, для получения имени текущего каталога используют функцию getcwd() из библиотеки Cwd:

```
#!/usr/local/bin/perl
use Cwd;
$dir = getcwd();
opendir DIR,$dir;
....
closedir DIR;
```

При организации конвейеров также применяются модули из библиотеки. Стандартные библиотеки обычно поставляются вместе с дистрибутивом Perl и устанавливаются при его сборке.

4: Введение в программирование скриптов на C

Информация

Язык программирования C — это традиционный инструмент разработки программного обеспечения, используемый на протяжении последних 25 лет (с момента появления Unix). С учетом того, что Unix в настоящее время является основной серверной средой, умение программировать CGI-скрипты на C является одним из необходимых условий успешной работы Web-инженера.

Вникнув повнимательнее в спецификацию CGI, любой программист поймет, что спецификация создавалась с расчетом на Unix и C. Работа с переменными окружения и потоками стандартного ввода/вывода построена с учетом особенностей среды и средств программирования. При адаптации спецификации CGI в других средах, например, MS-Windows, программирование многих механизмов обмена приходится модифицировать.

Большинство задач при разработке скриптов можно решить средствами Perl, но есть ряд задач, которые требуют использования C-программ. Наиболее распространенным применением C являются скрипты-интерфейсы к базам данных. В качестве примера такого интерфейса воспользуемся библиотекой PostgreSQL.

В данном разделе речь пойдет о программировании CGI-скриптов на классическом C без объектно-ориентрованных особенностей C++.

Общая структура C-скрипта

Скрипт на языке C ничем не отличается от обычной C-программы. Собственно, это набор процедур, среди которых есть главная процедура. Этой главной процедуре передается управление при загрузке программы в оперативную память. Главная процедура контролирует вызов других процедур и весь ход выполнения программы. Простой скрипт — это одна главная процедура.

При разработке C-скрипта следует всегда помнить, что в отличие от скрипта на Bash и Perl, C-скрипт, прежде чем выполнить, нужно еще и откомпилировать, т.е. превратить в исполняемый компьютером код. Если в системе нет компилятора для C, то программировать C-скрипты будет довольно сложно.

Синтаксически главная процедура выглядит следующим образом:

```
#include <stdlib.h>
#include <stdio.h>>
void main(argc,argv,env)
int *argc;
char *argv[];
char *env[];
{
/* тело программы */
}
```

В этом фрагменте представлены все основные элементы программирования CGI-скриптов на C. Строки в начале программы (#include ...) позволяют включить в текст программы декларации (описатели) стандартных функций. Строка "void main ..." — это объявление главной процедуры. В качестве параметров в данную процедуру (функцию) передаются:

- число аргументов командной строки — argc;
- указатель на массив аргументов командной строки — argv;
- указатель на массив переменных окружения — env.

Само тело программы помещается между символами фигурных скобок "{...}". Фраза "тело программы" размещена между парой "/* ... */". Это комментарий. Сама программа на C состоит из операторов. Операторы могут быть простые и составные. Простые операторы – это, например, оператор присваивания. Составной оператор — это блок. Блок представляет собой последовательность операторов, заключенную в фигурные скобки "{...}". В конце простого оператора должен стоять символ ";". В нашем примере объявление (декларирование) переменных перед блоком тела программы — это последовательность простых операторов.

Про C часто говорят, что в языке только пять операторов, а все остальное — это библиотека стандартных функций. Операторов в нем, конечно, больше, но такая характеристика в целом верна.

Стандартный поток вывода

Стандартный поток вывода в C ассоциируется с дескриптором STDOUT. Самым распространенным способом записи данных в этот поток является функция форматного вывода printf. Если скрипт должен что-то передать браузеру пользователя, то первое, что нужно сделать — это применить printf для формирования HTTP-заголовка:

```
main()
{
printf("Content-type: text/html\n\n");
printf("<H1>C и CGI</H1>");
}
```

Первый вызов printf формирует заголовок — определяет тип тела HTTP-отклика, а второй вызов формирует заглавие первого уровня в HTML-документе. В общем случае у функции printf три аргумента: printf(FILE,"format",VARS_LIST); FILE — дескриптор файла, "format" — формат вывода данных, VARS_LIST — список переменных, чьи значения подлежат выводу. Если дескриптор файла опущен, то вывод направляется в поток стандартного вывода. Список переменных указывается в том случае, если в формате вывода есть шаблоны вывода для переменных из этого списка.

Для каждого типа данных в C существует свой шаблон вывода. Перечислим только некоторые из них:

```
%d – вывод целого числа;
%s – вывод массива символов (строки);
%f – вывод вещественного числа;
%x – вывод целого числа в шестнадцатеричном
     виде.
```

Для того, чтобы распечатать аргументы командной строки, можно применить следующий формат:

```
int i;
...
for(i=0;i<argc;i++)
{
printf("arg[%d]=%s\n",i,argv[i]);
}
```

В данном случае переменная цикла i — это целая константа, поэтому в квадратных скобках указано [%d]. Второй аргумент списка переменных — указатель на массив символов (строка, содержащая значение аргумента командной строки), поэтому после знака равенства ("=") применен шаблон вывода массива символов %s.

Переменные окружения

Третий аргумент главной процедуры — указатель на массив переменных окружения, каждый элемент которого представляет собой строковую константу вида "имя-значение". Неудобство работы с этим массивом заключается в том, что заранее не известно, сколько элементов он содержит. Список переменных окружения кончается в тот момент, когда при его переборе встречается указатель NULL. Пример такого перебора представлен ниже:

```
#include <stdlib.h>
#include <stdio.h>
void main(argc,argv,env)
int argc;
char *argv[];
char *env[];
{
int i;
i=0;
while(env[i])
{
printf("%d:%s\n",i,env[i]);
i++;
}

}
```

В данном случае перебор идет до тех пор, пока значение указателя больше 0. При этом переменные окружения выдаются в виде "имя:значение". Для того, чтобы воспользоваться этими значениями, придется разбирать каждую строку, выделяя имя и значение переменной при помощи манипуляций с адресами внутри строки. Любые адресные операции — это потенциальные ошибки (хотя они и позволяют писать быстрые программы).

К счастью, в C есть функция getenv(). В качестве аргумента этой функции

достаточно указать имя переменной окружения, и система вернет указатель на его значение. Например, необходимо знать, сколько символов нужно считать со стандартного ввода скрипта:

```
int i,n;
char *query;
...
n = atoi(getenv("CONTENT_LENGTH"));
query = (char *) malloc(n+1);
for(i=0;i<n;i++)
{
query[i] = getc();
}
...
free(query);
```

В этом примере при помощи последовательного применения getenv и atoi (ascii to integer) из переменной окружения в переменную целого типа помещается значение переменной окружения — CONTENT_LENGTH. Последовательное применение функций здесь необходимо, т.к. значение переменной окружения — строка, а мы хотим использовать число, следовательно, строку символов следует не только получить, но еще и преобразовать в число.

Для преобразования строк в числа часто используют функцию форматного ввода sscanf. В нашем случае это выглядело примерно следующим образом:

```
char *length;
int n;
...
length = getenv("CONTENT_LENGTH");
sscanf(length,"%d",&n);
...
```

Следует заметить, что sscanf — это довольно сложная функция. Она предназначена для ввода любой информации и ее преобразования. Как показывает практика, иногда sscanf работает не так, как предполагает программист. Поэтому незачем стрелять из пушки по воробьям — применяйте лучше специализированные функции преобразования, например, atoi, если это возможно.

Аргументы командной строки

Аргументы командной строки передаются в С-скрипт через второй параметр главной процедуры. Второй параметр — массив указателей на строковые константы, которые и есть аргументы командной строки. Число таких аргументов определяется первым параметром главной процедуры. В этом смысле просмотреть все аргументы командной строки можно в цикле for:

```
#include <stdlib.h>
#include <stdio.h>
main(argc,argv,env)
int argc;
char *argv[];
char *env[];
{
int i;
printf("Content-type: text/plain\n\n");
for(i=0;i<argc;i++)
{
printf("argv[%d]=%s\n",i,argv[i]);
}

}
```

В данном случае скрипт генерирует простую текстовую страницу, на которой в столбик распечатываются аргументы командной строки скрипта. Такие аргументы появляются только у запроса типа ISINDEX. При работе с числовыми аргументами нужно помнить, что передаются они в программу как строки, и их следует преобразовывать в числа. Лучше всего это делать при помощи функций atoi.

Стандартный поток ввода

Считывать данные в программу на С принято из файлов. При этом файлы ассоциируются с потоками данных. Поток данных — это последовательность октетов (8 бит, или более привычно — байт). Если считывается текстовый файл, то мы имеем дело с последовательностью символов. Если считываем двоичный файл — имеем дело с октетами (байтами). Все функции С, которые работают с файлами ориентированы на эту модель — на потоки данных.

С каждым файлом при открытии потока данных связан дескриптор файла, который, являясь совокупностью данных о потоке, описывает поток. Со

стандартным потоком ввода в C связывают дескриптор с именем STDIN. Во многих случаях это имя указывать не надо, т.к. оно предполагается по умолчанию.

Самый простой способ чтения потока стандартного ввода обеспечивает функция посимвольного чтения getc():

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
void main(argc,argv,env)
int argc;
char *argv[];
char *env[];
{
char *query;
int length;
length = atoi(getenv("CONTENT_LENGTH"));
query = (char *) malloc(length+1);
memset(query,'\000',length+1);
for(int i=0;i<length;i++)
{
query[i] = getc();
}
free(query);
}
```

Функция getc() доставляет по одному символу из потока стандартного ввода. Не следует думать, что это медленный способ чтения. Во-первых, сервер передает данные через канал (pipe), а во-вторых, поток буферизуется. Поэтому даже посимвольное чтение данных происходит достаточно быстро.

Можно прочитать и все данные сразу. Для этого пользуются функцией fread():

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
void main(argc,argv,env)
int argc;
char *argv[];
char *env[];
{
char *query;
int length;
```

```
length = atoi(getenv("CONTENT_LENGTH"));
query = (char *) malloc(length+1);
memset(query,'\000',length+1);
fread(query,length,1,STDIN);
free(query);
}
```

В данном случае мы читаем из потока стандартного ввода STDIN в буфер query ровно length символов.

Последнее замечание. В наших примерах используется функция malloc(). Эта функция отводит память под данные, которые мы считываем со стандартного ввода. После завершения скрипта нужно обязательно освободить эту память, не уповая на то, что после завершения программы вся память все равно освободится. Здесь учитывается два момента. Во-первых, использование такого способа размещения данных связано с попыткой избежать переполнения при использовании областей памяти фиксированной длины, которое приводит к аварийному завершению программы. Во-вторых, при переходе от CGI-скриптов к FastCGI-скриптам это позволяет избежать "утечки" памяти. В FastCGI скрипт не завершается после ответа клиенту, а остается "висеть" в памяти в ожидании следующего запроса. Если память не освободить, то при новом обращении произойдет резервирование новой области памяти. В конечном итоге свободной памяти может и не оказаться.

Типы данных и переменные

В языке C определено несколько типов данных, которые отражают архитектуру большинства современных компьютеров: целые числа (короткие и длинные), вещественные числа (нормальной и двойной точности), символы, массивы, структуры, кучи и указатели. Если программировать сложные CGI, то набор всех этих типов данных будет затребован программистом, но для программирования простых скриптов вполне достаточно рассмотреть короткие целые числа, строки и указатели.

Любая переменная в C должна быть продекларирована. В противном случае при выполнении многих операций можно получить неожиданные результаты. Для каждого типа данных используется своя форма декларации переменной. Для аккуратной работы с типами данных в C следует использовать операцию преобразования данных. Последнее особенно

актуально при работе с указателями и числовыми данными, в которых можно потерять точность.

Целые числа

Целое число декларируется как:

```
int a;
unsigned int au;
short b;
long c;
```

Нас интересуют только первые две строки. Последняя строка декларирует длинное целое число. Короткое число попадает в интервал $-2^{14}<a<2^{14}$. Если у числа указан модификатор unsigned, то оно попадает в интервал $0<a<2^{15}$.

Одновременно с декларированием число можно проинициализировать, что вообще-то рекомендуется делать всегда:

```
int a=0,b=0;
```

Как видно из этого примера, в одном операторе декларирования (объявления) переменных можно указать сразу несколько переменных одного типа и при этом их можно инициализировать. Переменные целого типа необходимы в CGI-программировании при обработке обращений по методу POST. Для того, чтобы считать данные из потока стандартного ввода, нужно указать скрипту, сколько байтов оттуда следует считать. При этом сначала текстовую константу из переменной окружения CONTENT_LENGTH следует преобразовать в число, а затем использовать в операторах чтения или цикла:

```
#include <stdio.h>
#include <string.h>
void main()
{
char *length,*buf;
int n,i;
length = (char *) getenv("CONTENT_LENGTH");
n = atoi(length);
buf = (char *) malloc(n+1);
memset(buf,'\000',n+1);
for(i=0;i<n;i++)
{
```

```
buf[i] = getc();
}
printf("Content-type: text/plain\n\n%s\n",
        buf);
free(buf);
```

Функция getenv() позволяет получить значение переменной CONTENT_LENGTH, а функция atoi() — преобразовать это значение в целое число, которое потом используется в качестве границы при посимвольном чтении данных из стандартного потока ввода.

Строки символов

В C нет специального типа данных, который позволял бы работать со строками символов. Для этой цели используются массивы символов. Одиночный символ или массив символов можно объявить через оператор char:

```
char a='\000';
char buf[];
char buf[20];
```

В первом случае переменная a — это просто одиночный символ. В зависимости от реализации компилятора на него будет отводиться разное число байтов. В наиболее экономичном варианте — 1 байт, если символ отображается на короткое целое — 2 байта, если в архитектуре аппаратной платформы нет числа меньше четырех байтов — на четыре байта. Одним словом, не следует думать, что под символ всегда отводится 1 байт.

Символы можно использовать в арифметических операциях:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
unsigned char a='a';
a++;
printf("%c\n",a);
}
```

В результате этих нехитрых операций вместо "a" будет напечатано "b". Код символа рассматривается как целое число и увеличивается на 1. Этот принцип можно использовать при преобразовании строчных латинских

букв в заглавные (с буквами русского алфавита так не получится).

Определив массив символов buf[20] в 20 символов длиной, мы зарезервировали под него место, в которое можем разместить 20 символов. При объявлении buf[] мы только обозначаем, что будем использовать переменную buf для обращений к массиву символов. Места под сам массив мы не отводим. То есть место мы отвели, но под указатель — переменную, которая будет хранить адрес массива символов. Следовательно, buf[20] отводит место под массив и под указатель на него.

Однако, как же быть со строками символов? Ведь все переменные окружения — это строки символов. Для работы с ними используются функции, которые описаны в include-файле string.h:

```
strcmp() – сравнение строк
strcpy() – копирование строк
strstr() – поиск подстроки
и т.д.
```

При этом строка распознается по символу '\000' в конце строки, т.е. все разряды байта или совокупности байтов, в которой расположен символ, равны нулю. Приведем пример копирования QUERY_STRING в массив символов query:

```
#include <stdio.h>
#include <string.h>
void main()
{
char query[1024];
strcpy(query,getenv("CONTENT_LENGTH"));
printf("Content-type: text/plain\n\n%s\n",
        query);
}
```

Функция strcpy() копирует строку запроса из переменной окружения QUERY_STRING в переменную query и после нее дописывает символ '\000'. Мы заранее отвели побольше символов под массив query, т.к. strcpy не проверяет границ массива, и при такой операции можно запросто "наехать" на область памяти, не отведенной под наш скрипт, что приведет к его аварийному завершению (segmentation violation — как это знакомо). Поэтому лучше контролировать размеры буферов и использовать указатели.

Указатели

Указатели — это наиболее мощное и одновременно опасное средство программирования в C. В современных языках, таких как Java, например, указатели уничтожают как класс, т.к. именно они — основной источник множества ошибок, а, точнее, манипуляции с ними.

С другой стороны, нельзя написать эффективной программы (быстро исполняется и занимает мало памяти), если не использовать указатели и адресную арифметику, которая позволяет манипулировать указателями.

Указатель на переменную определенного типа данных объявляется путем ввода символа "*" перед именем переменной:

```
int *n;
char *query;
```

При этом место резервируется под адрес соответствующего значения. В Intel-платформах существуют модификации указателей в зависимости от модели памяти (т.е. какой длины адрес должен использоваться — 16 бит, 24 бита или более). В 64-разрядных архитектурах просто указывается опция компилятора (например, в Irix 6.4 "-64" — длинные адреса для всей программы или "-32" — короткие адреса).

С указателями при программировании CGI-скриптов приходится сталкиваться постоянно. Указатель на переменную окружения возвращает функция getenv(). Другими словами, она возвращает адрес начала значения переменной окружения:

```
char *length;
length = (char *) getenv("CONTENT_LENGTH");
```

Другой пример — указатель на массив переменных окружения и массив аргументов командной строки:

```
#include <stdlib.h>
#include <stdio.h>
void main(argc,argv,env)
int argc;
char *argv[];
char *env[];
{
```

```
/* тело программы */
}
```

В данном случае конструкция типа *argv[] — это массив указателей, которые указывают на символы, а точнее, на символьные переменные. В данном контексте совсем по-другому смотрится конструкция char a[] — это просто иная форма записи указателя. Переменная a — это указатель на массив. Есть, правда, один нюанс. Он заключается в том, что, обращаясь к элементам массива, перемещаться мы будем на длину элемента массива, а это уже зависит от типа данных.

Управление потоком вычислений

C представляет собой универсальный язык программирования со всеми присущими подобным языкам атрибутами. Он родился в период увлечения структурным программированием, поэтому в нем есть операторы, которые позволяют построить программу без использования goto.

Из всех механизмов управления ветвлением программы и передач управления мы рассмотрим несколько:

- оператор if;
- оператор goto ;
- оператор while ;
- оператор for ;
- оператор switch ;
- оператор break.

В целом этого набора должно хватить для программирования CGI-скриптов.

Оператор if

Условное исполнение части кода программы в C определяется оператором if. В общем случае он имеет синтаксис:

```
if(условие) оператор; [else оператор;]
```

или

```
if(условие) { тело_блока }
  [else { тело_блока }]
```

В этой записи "условие" — это логическое выражение, которое возвращает значения "истина" или "ложь", например, "x>y". Оператор в данном контексте — это простой оператор C, например, "x=1". Блок — это совокупность простых операторов и/или блоков. Самый простой пример применения if — определение метода доступа к CGI-скрипту:

```
char *query;
int n;
...
if(strcmp(getenv("REQUEST_METHOD"),"GET"))
{
query = getenv("QUERY_STRING");
}
else
{
n = atoi(getenv("CONTENT_LENGTH"));
query = (char *) malloc(n+1);
memset(query,'\000',n+1);
fread(query,n,1,STDIN);
}
```

В данном случае в качестве условия используется функция сравнения двух строк. Если результат сравнения – "истина", то исполняется первый блок (запрос из переменной окружения QUERY_STRING), если результат сравнения – "ложь", то считываем запрос из стандартного ввода скрипта.

Оператор goto

Сколько было возражений против использования goto, но он до сих пор существует в большинстве языков программирования. В простых программах, по большому счету, без него можно обойтись, но в ряде случаев он необходим. Оператор goto — это принудительный переход на другой фрагмент кода программы. Например, при разных алгоритмах обработки выхода из цикла:

```
...
for(i=0;i<n;i++)
{
buf[i]=getc();
if(buf=='\n') goto STRING;
}
```

```
printf("В потоке ввода одна строка\n");
STRING:printf(
  "Одна строка считана из потока ввода\n");
...
```

В данном случае переход осуществляется при обнаружении символа конца строки во входном потоке.

Оператор while

Оператор while позволяет исполнять тело цикла до тех пор, пока верно условие. Например, при распечатке переменных окружения:

```
#include <stdlib.h>
#include <stdio.h>
void main(argc,argv,env)
int argc;
char *argv[];
char *env[];
{
int i;
i=0;
while(env[i])
{
printf("%d:%s\n",i,env[i]);
i++;
}

}
```

В данном случае программа распечатывает переменные окружения до тех пор, пока указатель env[i] не примет пустое значение.

Оператор for

Оператор for — это детализация общего случая оператора цикла. Он состоит из блока инициализации переменной цикла, условия исполнения тела цикла и блока операторов конца цикла. Самой простой и наиболее распространенной его формой является случай одной переменной цикла:

```
for(i=0;i<n;i++)
{
/* тело цикла */
}
```

При программировании CGI-скриптов с оператором цикла можно познакомиться при распечатке аргументов командной строки в запросах типа ISINDEX:

```
#include <stdlib.h>
#include <stdio.h>
main(argc,argv,env)
int argc;
char *argv[];
char *env[];
{
int i;
printf("Content-type: text/plain\n\n");
for(i=0;i<argc;i++)
{
printf("argv[%d]=%s\n",i,argv[i]);
}

}
```

В данном случае переменная цикла пробегает значения от 0 до значения переменной argc, которая содержит число аргументов командной строки.

Оператор switch

Оператор switch — это переключатель на несколько положений, если пользоваться терминологией электротехники. Аргумент может принимать некоторые значения, например целочисленные или символьные, а выполняться будет тот фрагмент кода, который описан для этого значения:

```
switch(x)
{
case 'a': x='A'; break;
...
case 'z': x='Z'; break;
}
```

Данный фрагмент кода позволяет все строчные буквы превратить в заглавные.

Оператор break

Оператор break применяется для досрочного завершения цикла, в котором

он указан. В некотором смысле он заменяет оператор goto. В примере с оператором switch оператор break использовался для выхода этого оператора из блока. Если бы break там не применялся, то операторы блока исполнялись бы последовательно с точки входа в блок до конца блока.

Оператор continue

Оператор continue применяется для пропуска операторов цикла, непосредственно следующих за оператором continue до конца цикла. В программировании CGI-скриптов оператор можно применять для обхода операторов вывода по условию, например, по IP-адресу удаленного хоста:

```
while(buf = fgets())
{
if(!strncmp("144.206.160.32",
  getenv("REMOTE_ADDR"),14)) continue;
...
}
```

В данном случае дополнительные операторы будут выполняться только для пользователей компьютера с IP-адресом 144.206.160.32.

Файлы

В C принята поточная модель файла данных. Она предполагает, что файл рассматривается как поток байтов. Операции чтения и записи файла опираются на понятия начала потока (первый байт), конца потока (последний байт) и текущей позиции в потоке (последний считанный/ записанный байт).

При открытии файла на чтение или запись с ним связывается дескриптор потока данных. Он указывает на структуру данных, в которой хранится необходимая для работы с файлом информация. Место под эту структуру отводится динамически, т.е. в момент исполнения программы. Для того, чтобы избежать "утечки" памяти, файлы после их использования следует закрывать.

Для начала работы файл нужно открыть:

```
FILE *IN;
...
```

```
IN = open("text.txt","r");
```

Объявление (декларация) FILE *IN; определяет переменную IN как указатель на структуру дескриптора файла. Функция open() присваивает указателю IN значение адреса дескриптора файла с именем text.txt. При этом файл открыт только для посимвольного чтения. Если нужно открыть файл для записи, то вместо r следует указать w. Если требуется и чтение, и запись в файл, то его открывают со значением второго аргумента (вместо r) равным r+.

После работы с файлом его нужно закрыть:

```
FILE *IN;
...
IN = open("text.txt","r");
...
close(IN);
```

Закрывается файл функцией close(). При этом происходит освобождение памяти из-под структуры дескриптора файла и буферов ввода/вывода, которые были созданы при открытии файла. Желательно закрывать файлы в обратной последовательности (последним закрывается файл, который был открыт первым). Это позволяет избежать фрагментации памяти.

Для чтения данных из файла можно использовать функцию fread():

```
FILE *IN;
char query[1024];
...
IN = open("text.txt","r");
fread(query,1024,1,IN);
...
close(IN);
```

В данном случае в массив символов query считывается один блок данных размером 1024 символа из файла, связанного с дескриптором IN. Если числа 1024 и 1 поменять местами, то функция read считает 1024 блока по одному символу.

Для записи в файл применяется другая функция — fwrite():

```
FILE *OUT;
...
```

```
OUT = open("text.txt","w");
fwrite(query,1024,1,OUT);
...
close(OUT);
```

Значения параметров в этой функции те же, что и в функции fread().

Функции fread() и fwrite() — это функции неформатного ввода/вывода. В них не происходит никакого преобразования данных. Данные записываются в файлы в том виде, в котором они хранятся в переменных. Следует отметить, что в C различают два типа потоков данных: символьные и двоичные. Если fread()/fwrite() применять для двоичного потока, то, действительно, никаких преобразований происходить не будет. Если применять эти функции к символьным потокам данных, то преобразования производятся. Например, выполняется обработка конца строки.

Кроме неформатного ввода/вывода, в C применяют форматный ввод/вывод. Он реализуется через функции fscanf() и fprintf(). Первая функция служит для считывания и преобразования данных, а вторая — для преобразования и записи. Примером применения этих функций может служить счетчик посещения страницы:

```
#include <stdlib.h>
#include <stdio.h>
void main()
{
FILE *IN;
int n;
IN = open("text.txt","r");
fscanf(IN,"%d",&n);
close(IN);
IN = open("text.txt","w");
n++;
printf("%d",n);
fprintf(IN,"%d",n);
close(IN);
}
```

В данном случае целое число, записанное ASCII-символами, считывается из файла text.txt, преобразуется в формат целого числа и помещается в переменную n. После этого оно увеличивается на единицу, записывается на старое место и распечатывается в поток стандартного вывода. Если в HTML-странице разместить подстановку (server side include) результатов

исполнения этого скрипта, то мы реализуем счетчик посещения страниц.

Препроцессор

Директивы препроцессора позволяют собрать программу на языке C из готовых блоков кода. Кроме того, можно реализовать управление процессом компиляции, например, разработать процедуру условной компиляции для разных операционных систем.

В рамках разработки простых CGI-скриптов нам нужна будет только инструкция включения "include". Во всех примерах данного раздела она используется для включения в код программы описаний функций из набора стандартных библиотек.

Если необходимо задействовать функции форматного ввода/вывода, а их мы применяем для печати в стандартный вывод, то следует использовать инструкцию #include <stdio.h>:

```
#include <stdio.h>
void main()
{
printf("Content-type: text/html\n\n");
printf("<HTML>");
printf("<HEAD>");
printf("</HEAD>");
printf("<BODY>");
printf("<H1>Привет от-CGI</H1>");
printf("</BODY>");
printf("</HTML>");
}
```

Инструкция препроцессора начинается с символа "#". При использовании инструкций включения различают локальные файлы и стандартные файлы включения. Когда применяются стандартные файлы включения, имя файла заключают в "<имя_файла>". При использовании локального файла имя файла заключают в обычные двойные кавычки — "имя_файла". В наших примерах применяются только стандартные файлы включений.

Мы используем файлы включения только для ввода в код программы описаний стандартных функций и констант, с этими функциями связанных. Для наиболее распространенных функций существует файл /usr/include/stdlib.h. Его включения в программу достаточно для того,

например, чтобы использовать функции ввода/вывода и сравнения строк:

```
#include <stdlib.h>
void main()
{
printf("Content-type: text/plain\n\n");
if(strcmp("GET",getenv("REQUEST_METHOD"))
{
printf("Нет даты в потоке STDIN");
}

}
```

В данном случае в stdlib.h определены шаблоны для функций strcmp() и getenv().

При программировании в среде Unix программист всегда может применить команду man, которая позволяет получить подсказку по использованию той или иной функции C.

Компиляция

Программа на C — это текстовый файл, из которого программа-компилятор создает исполняемый файл. CGI-скрипт — это исполняемый файл. Для компиляции используется компилятор с языка C. В большинстве Unix-платформ этот компилятор носит название cc.

Предположим, что нужно создать программу с именем hello.cgi. Код на C расположен в файле hello.c. В этом случае достаточно выполнить:

```
bash%cc -o hello.cgi hello.c
```

Опция "-o" в этой записи определяет имя исполняемого файла. Он задается сразу вслед за ней. Имя файла исходного текста C указывается просто в качестве параметра.

Если в скрипте использовать функции из внешней библиотеки, то компилятору необходимо указать ее адрес:

```
bash%cc -o test.cgi test.c -lpq
```

В данном случае мы используем внешнюю библиотеку pq. Опция -l

определяет имя библиотеки. Сама процедура сборки программы называется linking (связывание). Отсюда и буква "l" перед именем библиотеки.

5: HTML-формы

Элемент разметки FORM и его компоненты

Контейнер (элемент разметки) FORM позволяет определить в рамках HTML-документа форму ввода. В рамках этого контейнера размещаются все поля ввода, куда пользователь может поместить свою информацию. Если контейнер формы открыт, т.е. в документе указан тег начала контейнера <FORM ...>, то обязательно нужно указать и тег конца контейнера </FORM>.

В общем случае контейнер имеет следующий вид:

```
<FORM
NAME=...
ACTION=url
METHOD=POST|GET|PUT|...
enctype=application/x-www-form-urlencoded|
        multipart/form-data
[target=window_name]
>
...
</FORM>
```

Атрибут NAME используется для именования формы. Это делается главным образом в JavaScript-программах. Атрибут ACTION задает URL, по которому отправляются данные из формы. Атрибут METHOD определяет метод передачи данных (фактически, речь идет о формировании сообщения, передаваемого по сети). Атрибут ENCTYPE определяет тип кодирования данных в теле сообщения и разбиение сообщения на части. Необязательный атрибут TARGET позволяет адресовать окно, в котором будет отображаться результат обработки данных из формы.

В рамках обзора применения контейнера FORM мы рассмотрим:

- передачу данных по электронной почте;
- передачу данных скрипту через атрибут ACTION;
- передачу данных через Server Side Include.

Инициировать обмен можно при помощи JavaScript-кода, но рассматривать

данный способ программирования обмена данными мы здесь не будем.

FORM (mailto)

Контейнер FORM позволяет определить в рамках HTML-документа форму ввода. В рамках этого контейнера размещаются все поля ввода, в которые пользователь может поместить информацию. Часто автор страниц Web-сайта по тем или иным причинам не имеет возможности программировать на стороне сервера. Однако это не означает, что он не может применять формы. Формы можно применять для отправки почты. Однако, как и в любом деле, здесь есть свои особенности, например:

```
<FORM ACTION=mailto:help@intuit.ru>
<INPUT NAME=n1 VALUE="Поле1">
<INPUT TYPE=BUTTON VALUE="Отправить">
</FORM>
```

В данном примере (cgimail1.htm) мы пытаемся отправить значение поля формы n1 по электронной почте абоненту help@intuit.ru. После заполнения поля и выбора кнопки "Отправить", браузер откроет окно программы почтового клиента, что не входило в наши планы. При этом само значение поля куда-то исчезнет.

Почему открывается новое окно? Несмотря на полный произвол, который царит в Web, и жесточайшую конкуренцию между Netscape и Microsoft, логика, заложенная в архитектуру World Wide Web Бернерсом Ли, обеими компаниями соблюдается. Дело в том, что, согласно спецификации RFC 822 (формат текстового сообщения Internet), на которую опираются протоколы HTTP и SMTP (Simple Mail Transfer Protocol, простой протокол электронной почты), сообщение может состоять из двух частей: заголовка и тела. В том виде, в каком мы используем контейнер FORM, метод доступа к ресурсу не указан и, следовательно, по умолчанию выбирается GET. У нас нет тела сообщения, а есть только заголовок.

Кроме того, в примере мы применяем схему URL mailto. Она соответствует спецификации протокола SMTP (обмен почтовыми сообщениями в Internet). В этой схеме, в отличие от схемы HTTP, расширенный путь после доменного имени стандартом не предусмотрен.

Итак, для того, чтобы получить тело сообщения, необходимо указать метод

POST (cgimail2.htm). В этом случае сообщение должно уйти абоненту без открытия окна почтового клиента:

```
<FORM METHOD=post
      ACTION=mailto:help@intuit.ru>
<INPUT NAME=n1 VALUE="Поле1">
<INPUT TYPE=BUTTON VALUE="Отправить">
</FORM>
```

Любопытно, что в данном случае мы использовали протокол, отличный от HTTP — SMTP. В нем нет понятия метода доступа вообще. Тем не менее логика разбиения текстовых сообщений для всех протоколов одна и та же, как, собственно, и предполагалось при создании Web-технологии — унификация доступа к ресурсам.

На этом примере хорошо видны отличия URI от URL. В данном случае были возможны различные механизмы обработки данных в запросе на ресурс, который задается URI. Но конкретная реализация преобразования данных в запрос в рамках Web — это и есть URL, т.е. URI в рамках World Wide Web.

Расширим наш пример, добавив в него отправку абоненту внешнего файла по электронной почте. Такая задача встречается довольно часто. Например, поддержка архива электронных публикаций. Здесь нет необходимости в немедленном опубликовании материалов. Все статьи должны пройти экспертизу, которая требует определенного времени. Для этого модифицируем наш пример, добавив в него поле типа FILE:

```
<FORM METHOD=post
      ACTION=mailto:help@intuit.ru>
<INPUT NAME=n1 VALUE="Поле1">
<INPUT NAME=file TYPE=file>
<INPUT TYPE=BUTTON VALUE="Отправить">
</FORM>
```

Почему в данном случае нельзя использовать метод GET, объяснялось выше. Метод POST должен обеспечить нам размещение всего файла в теле сообщения.

Однако все это верно, пока мы работаем с текстовыми файлами и находимся в рамках RFC822. А если нам нужно передать файл с длинными строками (Postscript) или просто двоичный файл? В таком случае

необходимо обратиться к формату MIME. Это можно сделать при помощи еще одного атрибута контейнера FORM — ENCTYPE:

```
<FORM
ENCTYPE=multipart/form-data
METHOD=post
ACTION=mailto:help@intuit.ru>
<INPUT NAME=n1 VALUE="Поле1">
<INPUT NAME=file TYPE=file>
<INPUT TYPE=BUTTON VALUE="Отправить">
</FORM>
```

В данном случае по почте отправляется сообщение не в стандарте RFC822, а в стандарте MIME. Тело сообщения будет состоять из нескольких частей, а файл будет преобразован в ASCII-символы в соответствии со спецификацией BASE-64. Стандартный почтовый клиент воспринимает такой файл как присоединенный и позволяет его либо просмотреть, либо сохранить на диске.

FORM (HTTP)

Основной целью введения форм в HTML было обеспечение ввода данных в прикладную программу из универсального мультипротокольного браузера. При этом нужно отдавать себе отчет, что прикладная программа естественным образом должна выполняться на компьютере, где функционирует HTTP-сервер. Она не может работать в пустоте. Программу должен кто-то загружать, настраивать в адресном пространстве компьютера (linking), передавать ей управление и удалять из памяти после ее завершения.

Раз запрос от клиента принимает сервер, следовательно, и инициировать изложенные выше действия должен именно он.

Механизм инициирования такой прикладной программы определен в спецификации Common Gateway Interface. Там же задан и порядок обмена данными между HTTP-сервером и программой, которая в спецификации CGI именуется скриптом.

Метод GET

Основная задача формы — это предоставление шаблона ввода данных,

которые будут переданы скрипту. Сам скрипт при этом указывается через URL, который задается в атрибуте ACTION:

```
<FORM ACTION=script.cgi>
<INPUT NAME=n1 VALUE="Поле1">
<INPUT NAME=n2 VALUE="Поле2">
<INPUT TYPE=BUTTON VALUE="Отправить">
</FORM>
```

В данном примере скрипт просто распечатывает пару "имя поля формы — значение поля формы" в виде HTML-таблицы (formcgi1.htm). Правда, если присмотреться внимательно к происходящему на экране, можно обнаружить любопытную метаморфозу с URL скрипта при выборе кнопки "Отправить". В поле location окна браузера к скрипту после символа "?" приписываются пары "поле-значение", разделенные символом "&".

Данный запрос из формы определяют как запрос типа URLENCODED, переданный по методу GET. При передаче значений по методу GET формируется только заголовок HTTP-сообщения и не формируется его тело. Поэтому все содержание полей формы помещается в URL и таким образом передается скрипту. Из текста скрипта (formcgi2.htm) видно, что данные извлекаются из переменной окружения QUERY_STRING, в которую сервер помещает запрос.

Запросы, которые передаются в методе GET, можно условно разделить на два типа: ISINDEX и FORM-URLENCODED. FORM-URLENCODED мы только что рассмотрели, а ISINDEX был описан в разделах "Заголовок HTML-документа" и "Спецификация Common Gateway Interface", поэтому не будем повторяться.

Метод POST

Очевидно, что в строку URL нельзя втиснуть бесконечное число символов. И браузер, и среда, в которой функционирует сервер, имеют ограничения либо, как в случае браузера, по длине поля location, либо по длине области переменных окружения. Правда, последнее для современных систем не очень актуально. Например, операционная система IRIX 6.2 позволяет размещать в области переменных окружения данные объемом до 4 Мбайт. Тем не менее, для передачи относительно больших объемов предпочтительнее использовать метод доступа POST:

```
<FORM METHOD=post ACTION=script.cgi>
<INPUT NAME=n1 VALUE="Поле1">
<INPUT NAME=n2 VALUE="Поле2">
<INPUT TYPE=BUTTON VALUE="Отправить">
</FORM>
```

В нашем примере в контейнере FORM появился атрибут METHOD, который принял значение POST. Результат работы скрипта не изменился, но сам скрипт претерпел существенные изменения. Теперь запрос принимается со стандартного ввода, а не из переменной окружения QUERY_STRING.

При методе POST данные передаются как тело HTTP-сообщения, и скрипт читает их со стандартного ввода. При этом есть один существенный нюанс, который ограничивает круг средств разработки скриптов для приема данных по POST. Он заключается в том, что сервер не закрывает канал передачи данных скрипту после передачи последнего символа запроса. В переменной CONTENT_LENGTH сервер сообщает, сколько данных со стандартного ввода нужно считать. Таким образом, язык программирования сценариев или универсальный язык программирования должны уметь читать определенное количество символов из стандартного ввода. Например, многие разновидности командных языков UNIX (Bourn-shell, Kernel-shell и т.п.) могут читать только строками и ждут закрытия входного потока.

Обычно при описании программирования CGI-скриптов рассматривают только методы GET и POST. В принципе, в форме можно указывать любые другие методы, например, PUT. Просто серверы не имеют стандартных модулей обработки этих методов, поэтому, кроме формы и скрипта, в случае нестандартного метода требуется произвести еще и соответствующую настройку сервера.

Кодирование

Существует два типа кодирования содержания (тела) HTTP-сообщения, которые можно определить в форме:

- application/x-www-form-urlencoded
- multipart/form-data

Все, что рассматривалось в данном разделе до сих пор, относилось к первому типу кодирования тела HTTP-сообщения. Первый тип кодирования выбирается по умолчанию и является основным способом. Единственное, что пока не было рассмотрено, так это то, что, собственно, представляет собой этот самый URLENCODED.

В URL документа можно использовать только символы набора Latin1. Это первая половина таблицы ASCII за вычетом первых 20 символов. Все остальные символы заменяются своими шестнадцатеричными эквивалентами. Кроме того, такие символы, как "+" или "&", играют роль разделителей или коннекторов. Если они встречаются в значении поля, то тоже заменяются на шестнадцатеричный эквивалент. Наиболее характерно это для работы с русским алфавитом. Поэтому скрипт, который принимает запросы, должен уметь эти символы декодировать.

Второй тип применяется для передачи двоичной информации в теле HTTP-сообщения. Если проводить аналогии с электронной почтой, то multipart/form-data обеспечивает присоединение файла данных (attachment) к HTTP-запросу. Наиболее типичным примером является передача файла с машины пользователя на сервер:

```
<FORM ACTION=script.cgi METHOD=post
      ENCTYPE=multipart/form-data>
<INPUT NAME=n1 VALUE="Поле1">
<INPUT NAME=n2 TYPE=file>
<INPUT TYPE=BUTTON VALUE="Отправить">
</FORM>
```

В данном случае HTTP-сообщение будет очень похоже на почтовое сообщение в стандарте MIME (собственно, это и есть MIME-сообщение, только передается оно по протоколу HTTP). Естественно, что для приема такого сообщения нужен скрипт, который бы смог разобрать его на части, а потом декодировать необходимую информацию.

FORM (SSI)

Когда говорят о формах, обычно предполагается, что в контейнере FORM обязательно должен быть указан адрес скрипта. Этот скрипт примет данные и "на лету" сгенерирует страницу, которая и будет возвращена пользователю. Из этого правила существует, по крайней мере, два

исключения.

Во-первых, атрибут ACTION можно не указывать в том случае, если данные, введенные в форму, обрабатываются JavaScript-программой. В этом случае достаточно дать форме имя, чтобы к ее элементам (контейнерам) можно было обращаться. Передачу данных можно реализовать через метод submit, который будет выполняться при нажатии на гипертекстовую ссылку, например, formssi1.htm. Более подробно данный материал описан в главе "Программирование на JavaScript".

Во-вторых, принять данные можно через скрипт, который встроен в документ как Server Side Include. Этот способ мы рассмотрим более подробно.

Если не изменять базового адреса документа через контейнер BASE, то базовым адресом будет адрес загруженного документа. Если при этом в документе разместить форму вида:

```
<FORM>
<INPUT NAME=n1 VALUE="Поле1">
<INPUT NAME=n2 VALUE="Поле2">
<INPUT TYPE=BUTTON VALUE="Отправить">
</FORM>
```

то после перезагрузки документа мы получим этот же документ, только в URL после символа "?" будет добавлено содержание формы (formssi2.htm).

Если теперь несколько видоизменить документ — вставить в него Server Side Include — получим:

```
<FORM>
<INPUT NAME=n1 VALUE="Поле1">
<INPUT NAME=n2 VALUE="Поле2">
<INPUT TYPE=BUTTON VALUE="Отправить">
<HR>
<!--#exec cgi=./cgi.cgi -->
</FORM>
```

Сам скрипт принимает запрос из QUERY_STRING и распечатывает его в виде HTML-таблицы (formssi3.htm). При этом результат распечатывается вслед за формой после горизонтального отчеркивания.

Точно так же можно обработать данные и методом POST, только для этого необходимо указать его в атрибуте METHOD контейнера FORM.

INPUT

Контейнер INPUT является самым распространенным контейнером HTML-формы. Существует целых 10 типов этого контейнера (text, image, submit, reset, hidden, password, file, checkbox, radio, button), причем каждый из них отображается по-разному.

В общем виде контейнер имеет вид:

```
<INPUT
NAME="Имя"
TYPE="Тип"
[вариации параметров, зависящие от типа]
>
```

Чаще всего контейнер INPUT применяется для организации текстового поля ввода: например, для ввода списка ключевых слов или для заполнения регистрационных форм.

INPUT (text)

Тип text контейнера INPUT определяет текстовое поле ввода, в котором пользователь (читатель) Web-страницы (узла) может ввести свою информацию. В DTD HTML 4.0 на поле INPUT ограничения по длине текстового поля не определены. Однако такие ограничения существуют. Они меняются от браузера к браузеру. Наиболее разумное ограничение — 256 символов.

Поле типа text имеет в общем случае следующий вид:

```
<INPUT NAME="Имя" TYPE=text SIZE=number
       MAXLENGTH=number>
```

Атрибут NAME используется для именования поля как элемента формы. Имя поля попадает в запрос (левая часть пары "имя_поля-значение"), а также применяется в JavaScript для чтения и изменения значений текстовых полей формы.

Атрибут SIZE задает размер видимой на экране части текстового поля. Ниже приведен простой пример:

```
<FORM>
<INPUT SIZE=10>
<INPUT SIZE=20>
</FORM>
```

Когда вводишь данные в этих двух полях, выясняется, что число символов, которое можно ввести, для обоих полей одинаковое, а вот число отображаемых символов ограничивается рамками текстового поля.

Атрибут MAXLENGTH задает максимальный размер поля. Он полезен в тех случаях, когда требуется ограничить вводимые данные по длине. Дополним поля из предыдущего примера этим атрибутом:

```
<FORM>
<INPUT SIZE=10 MAXLENGTH=15>
<INPUT SIZE=20 MAXLENGTH=15>
</FORM>
```

Максимальная длина поля равна в 15 символам. В первом случае строчка будет "прокручиваться" в горизонтальном направлении справа налево при превышении размера видимой области, равной 10 символам. Во втором случае ввод остановится во второй трети поля. В текстовых полях ввода используются шрифты фиксированной ширины.

При разработке различных форм часто требуется выравнивание. Например, при реализации формы-анкеты нужно выравнивать графы. В этом случае формы с текстовыми полями помещаются в таблицу:

```
<FORM>
<TABLE>
<TR>
<TD>Имя: </TD>
<TD><INPUT SIZE=5 MAXLENGTH=15></TD>
</TR>
<TR>
<TD>Фамилия: </TD>
<TD><INPUT SIZE=10 MAXLENGTH=15></TD>
</TR>
</TABLE>
</FORM>
```

В данном примере хорошо видно выравнивание полей по столбцам.

INPUT (password)

Тип password определяет текстовое поле, которое позволяет скрыть набираемый текст от посторонних глаз. По своим атрибутам поле типа password не отличается от поля типа text. При использовании этого поля следует понимать, что пароль или любая другая информация, которая вводится в поле типа password, будет передаваться по сети в виде ASCII-символов, т.е. будет доступна для просмотра при условии ее захвата посторонним лицом.

```
<FORM>
Пароль: <INPUT SIZE=10 TYPE=password>
</FORM>
```

В данном примере показано, как отображается поле этого типа при длине видимой части поля в 10 символов.

В принципе, существует несколько способов усилить защиту информации. Если включить штатный режим работы через SSL в браузере и сервере, то по сети будут передаваться зашифрованные сообщения. Более простым и не столь надежным методом может быть использование JavaScript.

INPUT (hidden)

Тип hidden был введен в формы по причине отсутствия поддержки сеансов в протоколе HTTP. Впоследствии были разработаны более эффективные способы эмуляции сеансов, например, cookie. Тем не менее поля hidden до сих пор могут использоваться для поддержки сеансов. В ряде случаев, когда запросы вынуждены проходить через сито систем защиты, это единственный способ реализации поддержки сеансов.

```
<FORM>
<INPUT MAXLENGTH=250 TYPE=hidden>
</FORM>
```

В данном случае мы задали невидимое поле шириной в 250 символов. Очевидно, что в случае невидимого поля задавать атрибут размера видимой части поля бессмысленно, поэтому атрибут SIZE в невидимых полях не употребляется.

При организации сеансовой работы через hidden-поля нужно учитывать некоторые нюансы, связанные с необходимостью инициировать событие передачи данных из формы. Это событие onSubmit. Оно наступает тогда, когда пользователь нажимает на кнопку типа submit или вызывает это событие из JavaScript.

Если на странице нет видимых полей, то там, естественно, нет и кнопки submit. В этом случае непосредственно вызвать событие передачи данных из формы не получится. Остается только второй способ — JavaScript. Если переход осуществляется по гипертекстовой ссылке на том же Web-узле, событие можно обрабатывать скриптом, связанным со ссылкой (hide2.htm). При этом следует учитывать факт отображения URL в полях location и status браузера (hide3.htm).

INPUT (checkbox)

Тип checkbox применяется в качестве селектора. Если, например, в заявке на комплектующие нужно выбирать из нескольких заранее определенных позиций, то можно применить поле типа checkbox:

```
<FORM>
1. Mouse – <INPUT NAME=mouse TYPE=checkbox>
2. Keyboard – <INPUT NAME=key TYPE=checkbox>
3. Monitor – <INPUT NAME=monitor TYPE=checkbox>
</FORM>
```

При использовании полей checkbox следует учитывать, что они хороши там, где не нужно выбирать. Например, на вопрос "Являетесь ли вы членом профсоюза?" может быть только два взаимоисключающих ответа: да или нет. В этом случае применяют поле другого типа — radio.

Отображение поля типа checkbox в запросе зависит от параметров, которые указаны в контейнере INPUT. Очевидно, что атрибуты длины и видимости в случае checkbox не применяются. Для checkbox, как и для любого другого поля, необходимо применение атрибута NAME, иначе значение поля не будет учтено в запросе. Кроме того, в checkbox используются атрибуты VALUE и CHECKED.

По умолчанию, если поле отмечено как выбранное пользователем, в запрос попадает пара "name=on", где name — имя поля. Атрибут VALUE позволяет

изменить значение выбранного поля. Например:

```
<FORM>
Mouse – <INPUT NAME=mouse VALUE=mouse
              TYPE=checkbox>
</FORM>
```

В этом случае вместо "on" в правой части равенства появится "mouse".

Атрибут CHECKED определяет состояние поля по умолчанию, т.е. в момент первоначальной загрузки страницы или выбора кнопки Reset. Если он указан, то поле считается по умолчанию выбранным (отмеченный прямоугольник):

```
<FORM>
Mouse – <INPUT NAME=mouse VALUE=mouse
              TYPE=checkbox CHECKED>
</FORM>
```

На дальнейшие действия пользователя этот атрибут не влияет, если, конечно, не потребуется снять отметку.

Важным отличием поля типа checkbox от поля типа radio является обработка полей с одинаковыми именами. В случае поля типа checkbox — это разные поля, и их значения никак не связаны между собой. При одновременном выборе полей с одинаковыми именами в запрос попадут все выбранные поля. При этом пары "имя=значение" будут просто повторяться. Другой вариант можно реализовать только через поля типа radio. Можно, конечно, исхитриться и сделать альтернативным вариант через поля типа checkbox, используя JavaScript.

INPUT (radio)

Тип radio контейнера INPUT определяет поле "селектор". Данный тип применяется там, где необходимо обеспечить выбор из нескольких заданных взаимоисключающих вариантов. Например, в анкете может быть графа "Пол":

```
<FORM>
Пол:
<INPUT NAME=sex TYPE=radio>Мужской
<INPUT NAME=sex TYPE=radio>Женский
```

```
</FORM>
```

В данном случае при выборе одного из вариантов со второго автоматически снимается отметка. Это главное отличие типа поля radio от типа checkbox. Обратите внимание на то, что имена полей одинаковые.

Если наш пример оставить как он есть, то скрипт, который будет принимать данные, не получит сведений о том, какой из вариантов был выбран. В любом случае будет выдаваться запрос типа: "?sex=on". Если вариантов не выбирать, то соответствующая пара "имя_поля-значение" вообще не появится в запросе.

Для того, чтобы указать выбранный вариант в контейнере INPUT, нужно ввести атрибут VALUE:

```
<FORM>
Пол:
<INPUT NAME=sex TYPE=radio VALUE=m>Мужской
<INPUT NAME=sex TYPE=radio VALUE=f>Женский
</FORM>
```

В данном случае вместо "on" передается соответствующее значение. Скрипт теперь в состоянии различить выбранный вариант.

Если в контейнерах INPUT типа radio задать разные имена, то разницы между этим типом и типом checkbox почти не будет:

```
<FORM>
Пол:
<INPUT NAME=sex1 TYPE=radio VALUE=m>Мужской
<INPUT NAME=sex2 TYPE=radio VALUE=f>Женский
</FORM>
```

Слово "почти" означает отсутствие возможности отменить выбор альтернативы, если она уже была выбрана. Таким образом, предполагается, что один из вариантов должен быть выбран обязательно, что должно заметно влиять на применение полей INPUT типа radio.

В данном контексте следует рассматривать и применение атрибута CHECKED в полях этого типа. Как и в checkbox, атрибут CHECKED позволяет определить значение поля по умолчанию. В нашем случае выбор по умолчанию из набора вариантов:

```
<FORM>
Пол:
<INPUT NAME=sex TYPE=radio VALUE=m
       CHECKED>Мужской
<INPUT NAME=sex TYPE=radio VALUE=f>Женский
</FORM>
```

Пусть в нашем примере речь идет о приеме на вредную работу, на которую нанимаются преимущественно мужчины. Альтернативой по умолчанию станет значение "m" для поля с именем sex. Значение по умолчанию устанавливается либо при первичной загрузке страницы, либо при выборе кнопки типа reset.

INPUT (image)

Кроме текста в полях формы можно вводить и координаты местоположения манипулятора "мышь". Для этой цели служит тип image контейнера INPUT. Вообще говоря, трудно представить форму, состоящую из многих полей, в которой среди прочих полей ввода будет и поле данного типа. Тем не менее представить осмысленное применение такого типа поля можно.

В данном типе множество атрибутов контейнера INPUT дополняется атрибутами контейнера IMAGE:

```
<FORM>
<INPUT TYPE=image SRC=image.gif NAME=i
       ALIGN=left BORDER=0>
</FORM>
```

При размещении поля этого типа атрибут NAME, кроме стандартных функций именования, выполняет еще и функции атрибута ALT из контейнера IMG. При наведении манипулятора "мышь" на картинку появляется значение поля NAME. Выбор данного поля манипулятором "мышь" приводит к немедленной передаче данных из формы серверу, а затем скрипту. При этом координаты манипулятора "мышь" передаются в виде "имя_поля.x=DD&имя_поля.y=DD".

Использовать NAME для визуального именования нежелательно, так как автора документа вынуждают выбирать между коротким именем, удобным при работе запроса скриптом, и длинным, которое достаточно подробно именует отображаемый объект.

В отличие от других полей, отсутствие атрибута NAME не приводит к отсутствию данных о поле в запросе. В этом случае будет просто передана пара x и y ("имя_поля.x=DD&имя_поля.y=DD").

Если такое поле в форме одно, имя поля можно вообще опустить. В этом случае нежелательно использовать имена x и/или y для именования других полей.

При наличии поля типа image меняется реакция на клавишу Enter клавиатуры компьютера, если фокус ввода находится в одном из полей формы. Происходит немедленная передача данных серверу. Это следует учитывать при разработке скриптов, которые принимают данные из форм с полем image. Пользователь может по ошибке нажать на Enter, еще не заполнив необходимые поля формы.

INPUT (button)

Во всех современных интерфейсах есть объекты, имитирующие кнопки управления. Интерфейс HTML-форм в этом смысле не является исключением. Контейнер INPUT позволяет создать кнопку при помощи типа button. Изначально в формах было только две кнопки: submit и Reset.

submit позволяет инициировать отправку данных серверу из формы и перезагрузку текущей страницы. В терминах JavaScript последовательность действий браузера, которую вызывает эта кнопка, называется событием onSubmit.

Reset позволяет выставить значения полей формы по умолчанию. Это бывает необходимо в случае неправильного ввода данных. В терминах JavaScript событие, вызванное выбором кнопки Reset, называется событием onReset.

Тип button контейнера INPUT является обобщением и расширением случаев submit и reset на более широкий класс объектов, которые принято называть кнопками. Форма с кнопками может выглядеть, например, следующим образом:

```
<FORM>
<INPUT TYPE=button VALUE="Кнопка">
</FORM>
```

Кнопки, в отличие от типа image, не вызывают события submit в случае их нажатия, и данный пример — лишнее тому подтверждение. Более того, даже после нажатия на кнопку пара "имя_поля-значение" в запрос не попадает. В общем случае это понятно. Ведь браузер не фиксирует положение кнопки (хотя и мог бы), поэтому и передавать нечего.

Работа с кнопками имеет еще одну особенность. Как и все значения в формах, текст в кнопках отображается обычно шрифтом фиксированной ширины. В некоторых версиях операционных систем и/или браузеров в качестве такого шрифта используется системный фонт. Это приводит к тому, что текст в кнопке на русском языке отображается абракадаброй.

Из всего сказанного пока не было понятно, каким образом используются кнопки. Дело в том, что с ними в JavaScript связано событие onClick, которое можно обработать функцией пользователя: например, чтобы послать данные формы.

Кнопка гораздо удобнее гипертекстовой ссылки, если не требуется перезагрузка страницы. При выборе гипертекстовой ссылки перезагрузка произойдет обязательно, если только не позаботиться об этом при указании атрибута HREF контейнера A (anchor). При выборе кнопки перезагрузка страницы не производится, и можно оставаться в пределах текущей страницы, управляя объектами интерфейса на ней, например, перезагружая картинки.

INPUT (submit)

Кнопка submit — едва ли не самый важный элемент формы. Она инициирует отправку данных формы на сервер. В первых реализациях HTML-форм в браузере Mosaic только при нажатии на эту кнопку происходила отправка данных. В настоящее время такое событие может произойти в нескольких случаях:

- в форме только одно текстовое поле (fisub1.htm);
- в форме есть поля типа image (fisub2.htm);
- в форме указана кнопка submit (fisub3.htm).

При указании поля типа submit следует учитывать, что если полю дать имя, то оно появится в запросе:

```
<FORM>
<INPUT TYPE=submit VALUE="Отправить?">
<INPUT TYPE=submit NAME=s>
</FORM>
```

Пример демонстрирует сразу две особенности полей типа submit. Во-первых, если задать значение атрибута VALUE, то оно отобразится в качестве текста на кнопке (первая кнопка). При этом следует помнить, что если текст содержит пробелы, то его нужно заключить в кавычки. Во-вторых, если не указывать атрибута VALUE, то будет подставлено значение кнопки по умолчанию. В-третьих, если есть имя, то при нажатии на кнопку произойдет событие onSubmit, и в запросе, который будет послан на сервер, появится пара "имя_поля_submit-значение".

Здесь мы сталкиваемся с той же дилеммой, что и в полях типа image. Если задавать многословные значения, то их потом придется "выковыривать" в скрипте из запроса, если задавать короткие, то пользователь не поймет, что от него хотят.

В одной форме может быть столько кнопок типа submit, сколько необходимо для работы. При этом их желательно именовать. В противном случае, скрипт не сможет определить, какая из кнопок вызвала передачу данных. Впрочем, иногда различие не требуется, если при нажатии на кнопку submit будут выполняться одинаковые действия.

INPUT (reset)

Кнопка Reset, наверное, есть самый простой элемент формы. Поле типа reset представляет собой кнопку, при нажатии на которую все исправления и назначения в форме принимают свои значения по умолчанию. Кнопка восстанавливает статус умолчания для всех полей формы. Статус по умолчанию можно установить только путем полной перезагрузки формы с сервера. Любопытно, но по Reload (обновление документа) установка статуса по умолчанию может не выполняться. Для того чтобы в этом убедиться достаточно внести исправления в текстовое поле примера, который расположен ниже, и после этого нажать на Reload — изменения останутся в силе (Netscape 4.0):

```
<FORM>
<INPUT TYPE=text VALUE="Отправить?">
<INPUT TYPE=reset NAME=s>
```

```
</FORM>
```

Выбор кнопки Reset вернет значение текстового поля в исходное состояние. Стоит отметить, что если перевести курсор в поле location браузера и нажать Enter, то в этом случае значение поля будет установлено равным значению по умолчанию. Таким образом, событие onReset не происходит при перезагрузке страницы с помощью кнопки Reload. Этот момент следует учитывать авторам страниц при разработке форм, а пользователям Web – при их заполнении.

INPUT (file)

Идея передачи больших объемов данных давно владела умами разработчиков программного обеспечения для World Wide Web. При этом возможностей полей textarea для решения этой задачи было явно недостаточно. Во-первых, можно передавать только текстовую информацию; во-вторых, текст нужно набирать вручную непосредственно в браузере; в-третьих, существует (точнее — существовало) ограничение на длину строки в поле textarea, которое перекочевало в формы из стандарта RFC822. Таким образом, в первой спецификации форм возможности передать любые данные с компьютера клиента на сервер не было.

Проблема, на первый взгляд, должна была решаться за счет метода доступа PUT, определенного в протоколе HTTP и призванного обеспечить размещение данных на стороне сервера. Но здесь обнаружился типичный недостаток любых теоретических установок, проявившийся еще в языке программирования Algol — отсутствие интерфейса для использования данного механизма. Ни один браузер не имеет стандартного средства для применения метода PUT. Реально данный метод стал использоваться только с появлением программ подготовки страниц.

Тем не менее типовая задача, которая возникает при создании архивов публикаций, заключается в необходимости отправить на сервер файл, содержащий публикацию. Первое, что приходит на ум — это электронная почта. MIME позволяет переслать составные почтовые сообщения и кодировать двоичные данные в них (fifile1.htm). Естественным решением является распространение этого способа и на Web.

```
<FORM ENCTYPE=multipart/form-data METHOD=post
      ACTION=mailto:help@intuit.ru>
```

```
<INPUT TYPE=file NAME=file>
<INPUT TYPE=submit VALUE="Опубликовать">
</FORM>
```

В данном примере мы отправляем пользователю help на хосте intuit.ru файл по протоколу SMTP. Следует обратить внимание на контейнер FORM. Файл нельзя передать в качестве дополнения к URL, следовательно, нужен метод, который сможет, кроме заголовка, сформировать еще и тело сообщения. Для этой цели используется метод POST. Для того чтобы обеспечить составное тело сообщения, мы применяем тип кодирования содержания multipart/form-data.

По умолчанию используется другой метод кодирования — application/x-www-form-urlencoded, поэтому его надо переопределить через атрибут ENCTYPE. При передаче двоичного файла по SMTP используется кодирование BASE-64, т.е. такое же, как и при режиме attachment в программе почтового клиента (fifile2.htm).

Но кардинальное решение заключается в вызове скрипта, который примет файл и выполнит его преобразование, разместит файл на сервере, или сделает что-либо еще.

```
<FORM ENCTYPE=multipart/form-data METHOD=post
      ACTION=file.cgi>
<INPUT TYPE=file NAME=file>
<INPUT TYPE=submit VALUE="Опубликовать">
</FORM>
```

Одно из преимуществ данного подхода заключается в том, что файл присоединяется к сообщению как двоичные данные. При вставке содержимого файла в тело HTTP-сообщения никакого преобразования не происходит. В сущности, этот подход нарушает стандарт текстового сообщения Internet, на котором базируется протокол HTTP. Но для работы он чрезвычайно удобен (fifile3.htm).

Механизм передачи файлов от клиента к серверу получил название File-upload. В отличие от обычной обработки запросов типа form-urlencoded при File-upload необходимо выполнить последовательно три основных группы действий:

- выделить части составного сообщения;

- найти описатели каждой из частей сообщения;
- выделить и обработать содержание каждой из частей составного сообщения.

При этом само сообщение практически повторяет формат MIME, т.е. точно так же есть граница, разделяющая части (boundary), Content-encoding и Content-type (fifile4.htm). Выполнить разбор составного сообщения не так легко. Тем более, что передаваться могут файлы разных форматов. Некоторые из них могут при этом претерпевать и конвертацию в транспортные форматы. Поэтому целесообразнее использовать библиотеки готовых модулей для работы в режиме File-upload.

Наиболее популярным средством разработки скриптов является язык Perl. Для Perl существует несколько стандартных модулей, которые ориентированы на работу с объектами World Wide Web. Естественно, что данные модули поддерживают и режим File-upload.

Самым простым из них, пожалуй, является модуль CGI_Lite. Данный модуль ориентирован только на разбор данных из форм. Режим File-upload реализован в нем достаточно просто и эффективно.

Другим популярным средством является набор модулей CGI_modules. Данный набор, кроме формирования HTML-страниц средствами функций из своей библиотеки, позволяет работать и с File-upload.

Еще одно такое средство — модуль CGI.pm. Он является прообразом многих компонентов из CGI_modules. В нем предусмотрен также механизм обработки File-upload.

Можно назвать еще ряд инструментов, которые позволяют обрабатывать составные сообщения. Однако приведенных примеров вполне достаточно для того, чтобы продемонстрировать возможности отправки файлов через Web-браузер.

SELECT

Контейнер SELECT применяется в формах для создания ниспадающих списков или списков прокрутки. При этом можно организовать выбор из множества вариантов только одного или отметить сразу несколько

вариантов отчетов. В некотором смысле этот контейнер реализует возможности сразу двух типов контейнера INPUT: checkbox и radio.

На самом деле ниспадающее меню или список прокрутки реализуется при вкладывании внутрь контейнера SELECT контейнеров OPTION. Именно они определяют варианты выбора или элементы списка:

```
<FORM>
<SELECT>
</SELECT>
</FORM>
```

В данном случае мы указали только контейнер SELECT. Он проявился в виде шаблона ниспадающего меню, но вариантов в этом меню нет. Обратите внимание, что контейнер имеет как тег начала контейнера, так и тег конца контейнера. Последний является обязательным. Если даже присвоить контейнеру SELECT имя, никаких данных с этим именем в запросе не будет.

Для реализации реального выпадающего меню внутрь контейнера SELECT нужно вложить контейнеры OPTION:

```
<FORM>
<SELECT>
<OPTION>Понедельник
<OPTION>Вторник
<OPTION>Среда
<OPTION>Четверг
<OPTION>Пятница
<OPTION>Суббота
<OPTION>Воскресенье
</SELECT>
</FORM>
```

Рис. 23.1.

По умолчанию в качестве текущей опции устанавливается первая опция списка. Как и в случае с полями INPUT, если после выбора опции нажать кнопку Reload, изменения выбора не произойдет. Значение по умолчанию устанавливается либо кнопкой Reset, либо при наборе в поле location браузера, либо при переходе по гипертекстовой ссылке. Для принудительного указания выбора по умолчанию следует воспользоваться

атрибутом SELECT контейнера OPTION. (открыть)

Выбранное значение альтернативы становится значением поля SELECT, и в запросе передается пара "имя_поля_select-значение_альтернативы" (fisel2.htm).

В ранних версиях браузеров существовало довольно жесткое ограничение на размер списка альтернатив. В настоящее время его нет, но все-таки злоупотреблять не стоит.

Для того, чтобы сформировать список прокрутки в контейнере SELECT, нужно указать атрибут SIZE:

```
<FORM>
<SELECT SIZE=5>
<OPTION>Понедельник
<OPTION>Вторник
<OPTION>Среда
<OPTION>Четверг
<OPTION>Пятница
<OPTION>Суббота
<OPTION>Воскресенье
</SELECT>
</FORM>
```

Рис. 23.2.

В данном случае мы реализовали окно прокрутки для пяти вариантов. Интересно отметить, что, например, в Netscape Navigator при перезагрузке по Reload выбранная альтернатива остается, а вот список устанавливается на начало. Так, в нашем случае при выборе "Воскресенье" после перезагрузки эта альтернатива окажется невидимой. В ряде случаев это вводит пользователя в заблуждение. (открыть)

Атрибут SIZE не изменяет самого характера работы с полем. Оно продолжает оставаться списком вариантов, из которого можно выбрать только один. Для организации множественного выбора в контейнере

SELECT необходимо указать атрибут MULTIPLE:

```
<FORM>
<SELECT SIZE=5 MULTIPLE>
<OPTION SELECTED>Понедельник
<OPTION>Вторник
<OPTION selected>Среда
<OPTION>Четверг
<OPTION>Пятница
<OPTION>Суббота
<OPTION>Воскресенье
</SELECT>
</FORM>
```

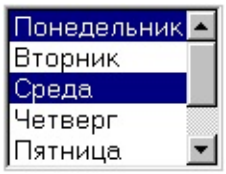

Рис. 23.3.

В данном примере по умолчанию выбрано два дня недели: понедельник и среда. По идее, пользователь должен иметь возможность либо устанавливать отметку о выборе опции, либо снимать ее. При этом таких отметок в одном списке может быть несколько. Однако практика показывает, что это не всегда так. Например, в Netscape Navigator 4.01 для Windows 3.1 при первоначальной загрузке страницы, действительно, отмечено две опции по умолчанию. Когда же пользователь начинает отмечать дополнительные опции, то выбрать удается только одну из них, при этом отметка по умолчанию снимается, т.е. мы имеем дело со списком альтернатив. Аналогично ведет себя и браузер Mosaic, например. Таким образом, обозначенная возможность множественного выбора поддерживается не всегда. (открыть)

OPTION

Контейнер OPTION никогда не используется сам по себе. Его применение имеет смысл только в контексте контейнера SELECT. В контейнере OPTION можно указывать два атрибута: VALUE и SELECTED.

Атрибут VALUE позволяет задать полю SELECT значение, отличное от

альтернативы, которая определена в OPTION. Это позволяет существенно упростить обработку запроса. Например, при выборе дней недели вместо их полных имен можно использовать сокращения: (открыть)

```
<FORM>
<SELECT NAME=s>
<OPTION VALUE=1>Понедельник
<OPTION VALUE=2>Вторник
<OPTION VALUE=3>Среда
<OPTION VALUE=4>Четверг
<OPTION VALUE=5>Пятница
<OPTION VALUE=6>Суббота
<OPTION VALUE=7>Воскресенье
</SELECT>
<INPUT TYPE=submit>
</FORM>
```

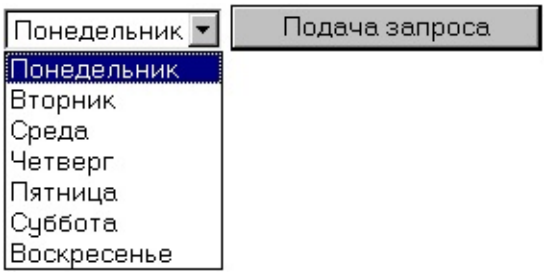

Рис. 23.4.

Если в данном примере выбрать кнопку "Submit Query", то в строке location можно убедиться в том, что поле s действительно приняло значение, которое определено в VALUE контейнера OPTION.

Второй возможный атрибут, SELECTED, определяет значение (значения) поля SELECT по умолчанию. Если задано несколько выбранных по умолчанию опций, то в обычном (селекторном) SELECT выбирается в качестве опции по умолчанию последняя, если это множественный выбор, то — все: (открыть)

```
<FORM>
<SELECT NAME=s>
<OPTION VALUE=1 selected>Понедельник
<OPTION VALUE=2>Вторник
<OPTION VALUE=3 selected>Среда
<OPTION VALUE=4>Четверг
<OPTION VALUE=5>Пятница
<OPTION VALUE=6>Суббота
```

```
<OPTION VALUE=7>Воскресенье
</SELECT>
<INPUT TYPE=submit>
</FORM>
```

Рис. 23.5.

Любопытно, что контейнер OPTION не именуется. Более того, при программировании в JavaScript все действия над значениями опций выполняются в рамках объекта SELECT.

TEXTAREA

Контейнер textarea — это первая попытка разрешить пользователю вводить большие текстовые фрагменты. Контейнер удобен там, где не требуется форматное представление данных, либо поле должно содержать произвольный фрагмент текста, например, комментарий. Если контейнер употребляется в документе, то нужно указывать как тег начала контейнера, так и тег его конца.

```
<FORM>
<TEXTAREA NAME=s>
Здесь можно ввести произвольный текст
</TEXTAREA>
<INPUT TYPE=submit>
</FORM>
```

Содержание поля textarea можно передать как методом get (fiarea1.htm), т.е. в URL скрипта, так и методом POST (fiarea2.htm).

Атрибуты контейнера textarea позволяют сформировать на экране окно, где будет отображаться значение поля, и правила этого отображения. Атрибут COLS определяет число столбцов в окне. Текст отображается в поле textarea фонтом фиксированной ширины, и этот атрибут задает ширину

поля в символах:

```
<FORM>
<TEXTAREA NAME=s COLS=5>
Здесь можно ввести произвольный текст
</TEXTAREA>
<INPUT TYPE=submit>
</FORM>
```

Аналогично ширине поля можно задать и его высоту в строках текста:

```
<FORM>
<TEXTAREA NAME=s COLS=15 ROWS=5>
Здесь можно ввести произвольный текст
</TEXTAREA>
<INPUT TYPE=submit>
</FORM>
```

И последний штрих — управление отображением текста. Во всех предыдущих примерах текст "вылезал" за правый край экрана, и его приходилось листать по горизонтали. Это достаточно неудобно. Кроме того, всегда существует дилемма: передавать текст на сервер как он есть (с переводом строк) или вытянуть в одну длинную строку. В Netscape для решения этих задач используют атрибут WRAP. Этот атрибут может принимать несколько значений:

- off — отключить выравнивание внутри поля (fiarea3.htm);
- virtual — включить выравнивание, но передавать как длинную строку (fiarea4.htm);
- physical — включить выравнивание, но передавать вместе с переводом строк (fiarea5.htm).

Наиболее интересен второй случай. В практике обработки данных скриптом очень часто приходится вытягивать ввод в одну строку и потом сравнивать ее с шаблоном. В случае WRAP=virtual мы избегаем первого шага:

```
<FORM>
<TEXTAREA NAME=s COLS=15 ROWS=5 WRAP=virtual>
Здесь можно ввести произвольный текст
</TEXTAREA>
<INPUT TYPE=submit>
</FORM>
```

6: Применение методов доступа HTTP в рамках программирования CGI-скриптов. Настройка HTTP-сервера для работы с CGI-скриптами

Метод доступа GET

Метод доступа GET долгое время был основным методом доступа из форм к CGI-скриптам. Это происходило по причине отсутствия при вводе большого количества данных и из-за прямого обращения к скриптам по их URL. В настоящее время ситуация меняется, но тем не менее данный метод занимает едва ли не главное место в программировании обработки данных из HTML-форм.

Условно использование GET можно разбить на два способа:

- запросы типа isindex ;
- запросы типа form-urlencoded .

В первом случае имитируется или реально происходит передача запроса, который появляется при вводе данных в строке приглашения контейнера ISINDEX. Во втором случае происходит передача пар "имя_поля=значение". И в том, и в другом случае данные, не входящие в кодировку Latin1, преобразуются в пары шестнадцатеричных символов, предваряемых символом "%" (%20 — пробел).

Кроме вызова скрипта непосредственно из гипертекстовой ссылки, скрипт можно запустить и через Server Side Includes. В этом случае данные из формы будут приписываться к URL документа, а не скрипта. Скрипт при этом будет вызываться сервером при разборе текста HTML-страницы перед отправкой ее клиенту.

Кроме собственно запроса, который в методе GET появляется в URL после символа "?", скрипту еще можно передать информацию в HTTP-пути. Это переменная окружения PATH_INFO . Обработка данных из этой переменной требует особого подхода к их получению и использованию в скрипте и гипертекстовых ссылках.

Запрос isindex

Запрос типа isindex является исторически первым способом передачи данных от браузера серверу . Он был разработан для передачи списка ключевых слов для поисковой машины. Запрос данного типа появляется либо в случае использования контейнера ISINDEX, либо при прямом обращении к скрипту через гипертекстовую ссылку. Данный тип запроса имеет ряд особенностей, которые отличают его от запроса типа form-urlencoded.

При использовании контейнера ISINDEX в начале документа появляется шаблон ввода ключевых слов. После ввода списка слов, разделенных пробелом, вызывается скрипт, который принимает список, разбирает его на отдельные слова и выполняет необходимую обработку. Первоначально isindex был ориентирован на модуль, подключавший поисковую систему WAIS к серверу CERN. После появления спецификации CGI стало возможным передавать списки слов любому CGI-скрипту. Запрос типа isindex определен только для метода доступа GET .

Согласно спецификации CGI для метода GET запрос присоединяется к URL документа или скрипта (указан атрибут ACTION в контейнере ISINDEX) после символа "?"(getis2.htm):

```
 http://localhost/htdocs/isindex.htm?search+
       engine+world+wide+web
```

или

```
http://localhost/htdocs/isindex.cgi?search+
       engine+world+wide+web
```

Как видно из этого примера, в запросе пробел заменяется на символ "+". Причем буквы русского алфавита в таком запросе перекодировать не надо, они передаются как есть. Если пользователь работает с локализованной версией операционной среды, то все будет отображаться так, как положено. В случае нелокализованной версии операционной среды, например, Windows NT, буквы будут отображаться абракадаброй, но в скрипт будут передаваться правильные коды.

Традиционно в GET данные запроса выбираются из переменной окружения QUERY_STRING. Например, это можно сделать на Perl следующим образом:

```
#!/usr/local/bin/perl
print "Content-type: text/plain\n\n";
print "Запрос: $ENV{QUERY_STRING}.\n";
```

В данном примере первый оператор печати формирует заголовок HTTP-сообщения в соответствии со спецификацией CGI. Второй оператор печати распечатывает содержание переменной окружения QUERY_STRING. Главное при этом — разделить запрос на отдельные слова, чтобы можно было использовать их в качестве ключей поиска. В Perl для этого существует функция split:

```
#!/usr/local/bin/perl
print "Content-type: text/plain\n\n";
print "Запрос: $ENV{QUERY_STRING}.\n";
@words = split('+',$ENV{QUERY_STRING});
foreach $word (@words)
{
print $word,"\n";
}
```

В данном случае следует обратить внимание на то, что в запросе нет никаких имен полей — только введенные слова и их разделители. Естественно, если среди введенных символов встретится разделитель, он будет заменен шестнадцатеричным.

У запроса isindex есть еще одно замечательное свойство — это передача данных в командной строке CGI-скрипта. Очевидно, что ввести аргументы пользователь не в состоянии (у него нет удаленного терминала), но вот принять данные из командной строки скрипт может:

```
#!/usr/local/bin/perl
print "Content-type: text/plain\n\n";
print "Запрос: $ENV{QUERY_STRING}.\n";
$n = @ARGV;
for($i=0;$i<$n;++$i)
{
print $ARGV[$i],"\n";
}
```

Внешне результаты работы данного скрипта и скрипта разбора QUERY_STRING ничем не отличаются. Но данные они получают из разных источников (getis6.htm).

Запрос типа isindex не порождается событием onSubmit, как это происходит в запросах form-urlencoded . Он является одной из разновидностей схемы http универсального локатора ресурсов (URL). При использовании обычной контекстной гипертекстовой ссылки (контейнер A (anchor)) запрос просто дописывается вслед за символом "?".

При программировании на JavaScript обратиться к скрипту через запрос isindex можно либо путем изменения значения атрибута HREF в одном из элементов массива гипертекстовых ссылок документа, либо путем вызова метода replace() объекта Location.

Запрос form-urlencoded

В методе GET запрос типа form-urlencoded является основной формой запроса. От запроса типа isindex он отличается форматом и способом передачи, точнее, кодировкой данных в теле HTTP-сообщения. Данные формы попадают в запрос, который расширяет URL скрипта в виде пар "имя_поля=значение&имя_поля=значение&...". Например, для формы вида:

```
<FORM ACTION=test.cgi METHOD=get>
Поле1:<INPUT NAME=f1 VALUE=value1>
Поле2:<INPUT NAME=f2 VALUE=value2>
<INPUT TYPE=submit VALUE="Послать">
</FORM>
```

запрос в сообщении HTTP-протокола будет выглядеть следующим образом:

```
GET /test.cgi?f1=value1&f2=value2 HTTP/1.0
```

Несмотря на то, что в форме имеется три поля, переданы будут значения только двух полей. Это связано с тем, что у третьего поля в форме нет имени. Если у поля нет имени, то его значение не передается серверу . Это правило общее для всех полей. Чаще всего оно применяется для полей

подтипов submit и reset типа text.

Применение неименованных полей позволяет передавать в скрипт только ту информацию, которая реально требуется для выполнения обработки данных. Иногда неименованные поля применяют и при программировании на JavaScript.

Кроме формата в запросе типа form-urlencoded , данные, введенные в форму, подвергаются дополнительной обработке — кодированию.

Кодирование, собственно, и дало название методу (urlencoded). Согласно спецификации, текстовое сообщение не может содержать символы, не входящие в набор Latin1. Это означает, что вторая половина таблицы ASCII и первые 20 символов должны быть закодированы. В CGI символ кодируется как две шестнадцатеричные цифры, следующие за знаком "%". Для российских Web-узлов это означает, что скрипт, который принимает запрос, должен предварительно перекодировать все шестнадцатеричные эквиваленты в символы (getform2.htm). На Perl это можно реализовать в одну строку:

```
query =~ s/%(.{2})/pack('c',hex($1))/ge;
```

В данном случае мы осуществляем глобальную подстановку (оператор "=~ s///"), который употреблен с модификаторами "ge". Первый модификатор обозначает глобальную замену по всей строке query, а второй требует выполнения перед заменой выражения "pack('c',hex($1))". Более подробно о программировании на Perl см. раздел "Введение в программирование на Perl".

Передача параметров через PATH_INFO

Передача данных в скрипты возможна не только при помощи переменной окружения QUERY_STRING или аргументов командной строки скрипта. Передать параметры в скрипт можно через переменную окружения PATH_INFO. Данная переменная принимает свое значение после преобразования URL скрипта. Рассмотрим следующий URL:

```
http://localhost/cgi-bin/test/arg1/arg2/arg3?param1+param2
```

Согласно спецификации URI адрес ресурса делится на две части: название схемы адресации и путь к ресурсу:

схема	разделитель	путь к ресурсу
http	:	//localhost/cgi-bin/test/arg1/arg2/arg3?param1+param2

схема адресации задается протоколом обмена данными. Обращение к скрипту осуществляется по схеме http. В свою очередь, в схеме http путь снова делится на две части: адрес ресурса и параметры. Эти части разделены символом "?". Параметры могут быть записаны либо в форме isindex, либо в формате form-urlencoded:

адрес ресурса	разделитель	параметры
//localhost/cgi-bin/test/arg1/arg2/arg3	?	param1+param2

Адрес ресурса в случае обращения к скрипту снова можно разделить на две части — адрес скрипта и путевой параметр PATH_INFO:

адрес скрипта	PATH_INFO
//localhost/cgi-bin/test	/arg1/arg2/arg3

В данном случае явного разделителя между адресом скрипта и PATH_INFO нет. Деление определяется настройками сервера . У большинства серверов стандартным каталогом CGI-скриптов является каталог cgi-bin. При этом подразумевается, что все файлы этого каталога — скрипты. Можно даже указать файл с расширением html, который в данном случае будет интерпретироваться как скрипт (getpath1.htm). Значение путевого параметра сервер помещает в переменную окружения PATH_INFO. При этом в нее попадает и лидирующий символ "/".

Управление работой скрипта через путевой параметр довольно популярно. Например, при выполнении перенаправления, когда нужно собирать статистику обращений к ресурсам, расположенным вне Web-узла:

```
http://localhost/cgi-bin/banner/
     http://otherhost/page.html
```

Вообще говоря, при таких перенаправлениях возникает опасность Web-

спуффинга. Существует очень большая вероятность, что администратор не заметит подмены одной из частей такого URL.

PATH_INFO применяется не только в совокупности с каталогами скриптов, но и с любым скриптом, определенным пользователем. Часто в качестве такого скрипта определяются файлы с расширением *.cgi:

```
http://www.intuit.ru/~user/script.cgi/path_param/test?arg1+arg2
```

В этом примере в переменную PATH_INFO попадет /path_param/test.

Метод доступа POST и другие методы доступа

Метод POST — это второй основной метод доступа к информационным ресурсам Web-узла. Он является альтернативой методу GET . Вообще, при HTTP-обмене используются три основных метода: GET , POST и HEAD. Первые два предназначены для получения страниц. Страницы при этом передаются в виде тела HTTP-отклика. При методе GET от клиента к серверу отправляется запрос, состоящий только из заголовка HTTP-сообщения. Все введенные пользователем данные размещаются в URL документа. При методе POST от клиента к серверу уходит запрос, который состоит из заголовка и тела HTTP-сообщения. При этом данные, введенные пользователем, размещаются в теле запроса. Метод HEAD применяется только для управления обменом и отображением. В рамках данного метода тело HTTP-сообщения не передается как клиентом в запросе, так и сервером в отклике.

Основное назначение метода POST — передача сравнительно больших объемов данных от клиента к серверу . Применение этого метода оправдано при передаче сложных состоящих из множества полей форм. В спецификации CGI от NCSA рекомендуется использовать метод POST при передаче данных из форм, содержащих поля textarea.

Современное использование Web в качестве альтернативы FTP-архивам расширило свойства метода POST . Так, большинство архивов научной периодики построено по принципу их обновления авторами статей. Для этой цели используются страницы с формами, содержащими поля типа File-upload. Этот механизм позволяет передать на сервер файл любого размера и любого типа. При этом сами пользователи не получают Web-

account на сервере архива, они пользуются стандартным скриптом публикации.

Из перечисленных выше методов только POST формирует тело сообщения. В спецификации CGI речь при этом идет только об HTTP-сообщениях. Но современные браузеры — это мультипротокольные программы. При этом в качестве гипертекстовых ссылок можно использовать различные схемы. Во многих протоколах, на которые эти схемы указывают, нет понятия метода доступа. Тем не менее в контейнере FORM такой метод можно использовать, например, со схемой mailto. В данном случае ни по какому методу POST , который не определен в протоколе SMTP, ничего не передается. POST просто заставляет браузер создать тело, в данном случае, почтового сообщения.

Чтение данных из стандартного потока ввода

При передаче запроса по методу POST от клиента к серверу передается HTTP-сообщение, которое состоит из заголовка и тела. Данные, введенные в HTML-форму, как раз и составляют тело сообщения. При обработке такого запроса CGI-скриптом данные следует выбирать из стандартного потока ввода скрипта, а не из переменной окружения QUERY_STRING. Эта переменная будет иметь пустое значение.

Для того, чтобы принять данные, нужно прочитать стандартный поток ввода. При этом из стандартного потока ввода нужно считать строго определенное количество байтов. Число байтов определяется переменной окружения CONTENT_LENGTH. В Perl прием данных в скрипт можно организовать следующим образом:

```
#!/usr/local/bin/perl
read STDIN,$query,$ENV(CONTENT_LENGTH);
```

Здесь из стандартного потока ввода STDIN считывается $ENV(CONTENT_LENGTH) данных и помещается в переменную $query. После этого можно уже что-то делать с запросом, например, распечатать его в виде HTML-таблицы.

Аналогично можно принять запрос из стандартного ввода и в C. Для этого следует воспользоваться в простейшем случае функцией getchar():

```
#include <stdlib.h>
#include <malloc.h>
void main()
{
int n,i;
char *buff;
n = atoi(getenv("CONTENT_LENGTH");
buff = (char *) malloc(n+1);
memset(buff,'\000',n+1);
for(i=0;i<n;i++)
{
buff[i] = getchar();
}
printf("Content-type: text/plain\n\n");
printf("Length of data into STDIN:%d\n",n);
printf("STDIN data: %s\n",buff);
free(buff);
}
```

Посимвольное чтение в этом примере можно заменить чтением по функции fread(). При этом не следует ожидать существенного уменьшения времени чтения данных. Во-первых, данные при вводе буферизуются. Во-вторых, в C применяется потоковая модель работы с внешними наборами данных.

Передача присоединенных файлов

Метод POST позволяет реализовать передачу файлов с компьютера пользователя в архив на HTTP- сервере . Для этой цели разработана специальная форма кодирования тела документа: multipart/form-data. Она указывается в контейнере FORM в атрибуте ENCTYPE совместно с методом POST :

```
<FORM ENCTYPE=multipart/form-data
     METHOD=post>
```

Скрипт, который принимает такие данные, должен определить метод доступа, затем определить тип тела документа и только после этого начать разбирать тело. В теле может быть как минимум две части: значения различных полей, которые доставляются скрипту в первой части сообщения, и тело передаваемого файла, которое передается как вторая часть сообщения.

Поля разбираются по традиционной схеме. Это обычные ASCII-символы. С ними никаких проблем не возникает. Тело документа передается как есть, т.е. без преобразований. Это значит, что применять для его выделения текстовые функции С нельзя, т.к. внутри документа могут попадаться любые символы, в том числе и символы конца символьного массива (строки).

Чтобы убедиться в этом, достаточно просто распечатать данные, посланные браузером. Для приема данных и их разбора нужно либо написать собственную программу, либо воспользоваться готовыми программами и библиотеками языка Perl, например.

Очевидно, что метод POST с полями file-upload используется для опубликования данных на стороне сервера . При этом файл, который передается по сети, должен быть размещен в файловой системе либо сервера , либо другого удаленного компьютера. Для этого пользователь, от имени которого запускается скрипт, должен иметь соответствующие права на доступ к каталогу файловой системы компьютера, в который записывается файл. Довольно часто модули стандартных библиотек, например, CGI_Lite или CGI.pm, используют для временного хранения каталог /tmp. Иногда данный каталог закрывают на запись, из-за чего могут возникнуть проблемы с приемом данных скриптом, составленным из модулей стандартной библиотеки.

Стандартные библиотеки разбора данных

Разбор запроса по методу POST CGI-скриптом — это рутинная процедура. При запросе типа urlencoded нужно просто выделить имена полей и их значения, а при запросе типа multipart/form-data — выделить части составного тела запроса и преобразовать их в имена полей, их значения и файлы.

С 1995 года было написано достаточно много заготовок для такого разбора, которые оформлены в виде свободно распространяемых библиотек. Наиболее популярными являются библиотеки модулей Perl — CGI.pm и CGI_Lite.

CGI.pm — полный набор функций для генерации HTML-файлов с формами и разбора запросов CGI-скриптами.

CGI_Lite — это средство работы с составными (multipart/form-data) запросами. При работе с функциями данного модуля следует иметь в виду, что временные файлы эти функции размещают в каталоге /tmp.

Метод доступа PUT и другие способы использования CGI-скриптов

Кроме стандартных способов использования CGI-скриптов, т.е. приема запросов от браузеров по методам GET и POST , скрипты применяются и для решения ряда других задач. К таким задачам можно отнести обслуживание расширенного набора методов доступа, например, PUT и DELETE.

Кроме того, для исполнения скриптов сам HTTP- сервер должен быть настроен соответствующим образом. В конфигурации по умолчанию сервера Apache предполагается, что все стандартные скрипты будут размещаться в каталоге ~server_root/cgi-bin, а скрипты пользователя будут иметь расширение *.cgi.

Если эксплуатируется только один Web-узел, этих настроек вполне достаточно. Если же на одной вычислительной установке эксплуатируется несколько виртуальных Web-узлов, то для каждого из них следует дополнительно определять и каталоги стандартного размещения, и расширения по умолчанию, и методы обработки нестандартных методов доступа.

Нередко CGI-скрипты применяются в качестве подстановок SSI на стороне сервера . Схема проста: HTML-документ используется как шаблон, в котором HTML-комментарии задают команды подстановок. В зависимости от различных условий сервер , который обрабатывает эти документы перед отправкой клиенту (браузеру), вставляет в шаблон результаты выполнения команд подстановок, в частности CGI-скриптов.

Преимущество CGI-скриптов в данном случае заключается в том, что они работают с переменными окружения , порожденными сервером для скрипта, а не с системными переменными окружения . Это позволяет включить механизмы анализа IP-адреса клиента, его доменного имени или cookie, чего нельзя сделать при работе с обычным набором переменных окружения , который порождается операционной системой.

Настройки сервера для работы с CGI-скриптами

Для исполнения CGI-скриптов сервер Apache должен быть соответствующим образом настроен. Во-первых, он должен быть собран с модулем исполнения CGI-скриптов (обычно включен по умолчанию), во-вторых, в файлах настройки сервера следует указать опции управления исполнением CGI-скриптов.

В данном разделе мы будем подразумевать, что сервер собран с модулем исполнения CGI-скриптов, поэтому обратимся сразу к настройкам сервера .

В версиях Apache, начиная с 1.2.6 можно все директивы настроек сервера включать в один файл httpd.conf. Однако традиционный способ настройки , который унаследован от NCSA- сервера , предполагает использование трех файлов настройки , которые отвечают за:

- настройку самого сервера (httpd.conf);
- настройку ресурсов Web-узла (srm.conf);
- настройку управления доступом к ресурсам (access.conf).

Для виртуальных хостов все директивы размещаются в файле httpd.conf в разделах описания каждого из виртуальных хостов.

httpd.conf

В этом файле определяются скрипты обработки нестандартных методов доступа (PUT или DELETE), а также описания работы с CGI-скриптами для виртуальных хостов.

Для указания скрипта обработки нестандартного метода используют директиву Script:

```
Script PUT put_script.cgi
```

Вместо PUT здесь можно указать DELETEили другой метод доступа. При обращении по данному методу доступа будет вызван скрипт, который указан в качестве второго аргумента.

Директивы для описания работы со скриптами для виртуальных хостов

размещают внутри контейнера VirtualHost:

```
<VirtualHost>
...
</VirtualHost>
```

Внутрь этого контейнера можно помещать все директивы, которые размещают для основного сервера в файлах httpd.conf, srm.conf, access.conf.

srm.conf

В этом файле определяется конфигурация ресурсов, которыми управляет сервер . Скрипты входят в состав этих ресурсов. Каталог скриптов по умолчанию определяет директива ScriptAlias:

```
ScriptAlias cgi-bin
     /usr/local/etc/httpd/cgi-bin
```

В данном каталоге определяется синоним части URL (первый параметр директивы), которому ставится в соответствие реальный путь в каталоге файловой системы вычислительной установки, где эксплуатируется сервер (второй аргумент). Например:

```
http://server.intuit.ru/cgi-bin/test.cgi
```

обращается к файлу

```
/usr/local/etc/httpd/cgi-bin/test.cgi
```

Кроме стандартного места размещения скриптов, которое определяется через ScriptAlias, скрипты можно хранить в произвольном каталоге, внутри дерева каталогов сервера .

Дерево каталогов сервера определяется директивой DocumentRoot:

```
DocumentRoot /www/host.ru/htdocs
```

или

```
DocumentRoot htdocs
```

В первом случае указан полный путь, от корня файловой системы, а во втором — относительный путь, т.е. путь от домашнего каталога сервера .

Для того, чтобы можно было запускать скрипты, нужно добавить handler (обработчик) для запуска скриптов из заданного каталога:

```
SetHandler cgi-script
```

Кроме того, с расширением файла можно связать MIME-тип, по которому сервер распознает скрипт:

```
AddType application/x-www-form-urlencoded .pl
```

В данном случае мы назначаем расширение *.pl для CGI-скриптов. Традиционным расширением скриптов по умолчанию является расширение *.cgi.

access.conf

Наиболее важной директивой в этом файле с точки зрения исполнения скриптов является Options. Она используется внутри контейнера Directory:

```
<Directory /usr/local/etc/httpd/htdocs>
Options ExecCGI
</Directory>
```

В данном случае для каталога /usr/local/etc/httpd/htdocs будет разрешено исполнение CGI-скриптов.

Скрипты для обработки нестандартных методов доступа

Для того, чтобы обработать метод доступа, отличный от GET или POST , необходимо выполнить несколько условий: подготовить скрипт для обработки данного метода, настроить сервер и определить соответствующие права доступа к каталогам, с которыми этот скрипт будет

работать. Рассмотрим как это делается на примере обработки запроса по методу PUT.

Некоторые серверы , например IIS компании Microsoft, имеют встроенные модули для работы с методом PUT. Сервер Apache такого модуля в стандартной комплектации не имеет, но позволяет подключить скрипт для обработки запросов по методу PUT.

Сама программа обработки таких запросов может выглядеть следующим образом:

```
#!/usr/local/bin/perl
if($ENV{REQUEST_METHOD} ne "PUT")
{
die "Content-type: text/plain\n\nМетод доступа не PUT";
}
$fname=$ENV{PATH_TRANSLATED};
if(!$fname)
{
die "Content-type: text/plain\n\nНе указана мишень вывода";
}
$length=$ENV{CONTENT_LENGTH};
if(!$length)
{
die "Content-type: text/plain\n\nСтраница имеет нулевой размер";
}
read(STDIN,$page,$length);
open(OUT,">$fname");
print OUT $page;
close(OUT);
print "Content-type: text/plain\n\nДанные получены.";
```

Листинг 24.1. (html, txt)

Первый оператор if проверяет метод доступа, второй — адрес страницы, которую следует разместить, третий — наличие самих данных для размещения по адресу страницы. Проверок для полной уверенности в передаче данных на самом деле нужно выполнить несколько больше. Кроме того, требуется сгенерировать код возврата и запись в журнал посещений, чтобы этот пример стал реально действующей программой. Теперь нужно настроить сервер . Для этого в файле конфигурации сервера httpd.conf следует указать:

```
Script PUT cgi-сценарий
```

Здесь cgi-сценарий — это имя нашего скрипта.

После этого для нашего сценария, а, точнее, для пользователя, от которого он запускается, нужно разрешить запись данных в каталог размещения страниц и, если это необходимо, внести изменения в файлы настройки процедуры аутентификации.

Скрипты и Server Side Includes

Стандартный модуль подстановок (includes) сервера Apache позволяет задействовать CGI-скрипты для генерации подстановок. Скрипт, в отличие от обычной программы, наследует переменные окружения , которые генерирует сервер для CGI-скриптов, а не стандартный набор переменных окружения оболочки (shell).

```
<!--#exec cgi="/cgi-bin/include.cgi" -->
```

В данном случае мы вставили в документ результат работы скрипта include.cgi.

Вставку можно использовать и не для генерации части текста документа, а для анализа данных, передаваемых в страницу или для анализа переменных окружения CGI-скрипта, т.е. условий обращения к странице. Например, для анализа IP-адреса пользователя и условного перенаправления запроса.

Для того, чтобы сервер выполнил подстановки в файл srm.conf, нужно внести строку определения типа документов, подлежащих разбору (server parsed documents):

```
AddType text/x-server-parsed-html .shtml
```

Расширение файла *.shtml обычно используется для документов, требующих анализа их содержания на предмет выполнения подстановок. Если администратор желает распространить предварительный анализ содержания документов на все документы, то вместо или в дополнение к .shtml можно указать и .html.

Кроме того, для каталога (файл access.conf — общий файл конфигурации доступа или .htaccess — файл конфигурации доступа, расположенный в данном каталоге и переопределяющий правила доступа), в котором расположены документы в директиве Options, должны быть разрешены и подстановки, и исполнение CGI-скриптов:

```
<Directory /usr/local/etc/httpd/htdocs>
AllowOverride
Options Includes ExecCGI
</Directory>
```

В данном случае для домашнего каталога документов сервера Apache, который используется в настройках по умолчанию, разрешено переопределять опции доступа в подкаталогах (AllowOverride) и исполнять подстановки и скрипты (Options Includes ExecCGI). По умолчанию обычно используют вместо последних двух опций одну — All. Она тоже разрешает подстановки и выполнение скриптов, а также ряд других действий:

```
<Directory /usr/local/etc/httpd/htdocs>
AllowOverride
Options All
</Directory>
```

Вообще говоря, существует возможность разрешить подстановки, но запретить выполнение скриптов:

```
<Directory /usr/local/etc/httpd/htdocs>
AllowOverride
Options IncludesNoExec
</Directory>
```

В этом случае отключаются не только скрипты, но и обычные команды, выполняемые из стандартной оболочки (shell). При этом можно разрешить исполнение скриптов самих по себе, но не в качестве вставок.

```
<Directory /usr/local/etc/httpd/htdocs>
AllowOverride
Options IncludesNoExec ExecCGI
</Directory>
```

В данной конфигурации исполнение скриптов разрешено, а выполнение подстановки по команде exec запрещено.

Литература

1. Храмцов П.Б., Брик С.А., Русак А.М., Сурин А.И.
   Основы web-технологий
   БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий - ИНТУИТ.ру, 2007
2. Савельева Н.В.
   Основы программирования на PHP
   Интернет-университет информационных технологий - ИНТУИТ.ру, 2005
3. Сузи Р.А.
   Язык программирования Python
   БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий - ИНТУИТ.ру, 2006
4. **CGI: Common Gateway Interface**
   Набор документации W3C. Наиболее полный архив документов по CGI.
5. **The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers**
   Это первоисточник спецификации. К нему следует возвращаться каждый раз, когда возникают сомнения в правильности работы чужого ПО, или когда Вы сами пишите свой модуль исполнения CGI-скриптов.
6. **CGI Documentation**
   Сайт очень похожий по структуре на данный учебный курс.
7. **Module mod_cgi**
   Описание модуля, который реализует спецификацию CGI в http-сервере Apache.