Работа с базами данных

В рамках данной лекции будет обсуждаться взаимодействие с базами данных и основные библиотеки для perl, которые позволяют осуществлять это взаимодействие. Рассматриваться будут преимущественно реляционные базы данных.

Contents

1 Основы реляционных баз данных. SQL

Для начала необходимо кратко изложить основы реляционных баз данных. Изложение в данном разделе не претендует на теоретическую строгость и ведётся с сугубо практической точки зрения.

1.1 Реляционные базы данных

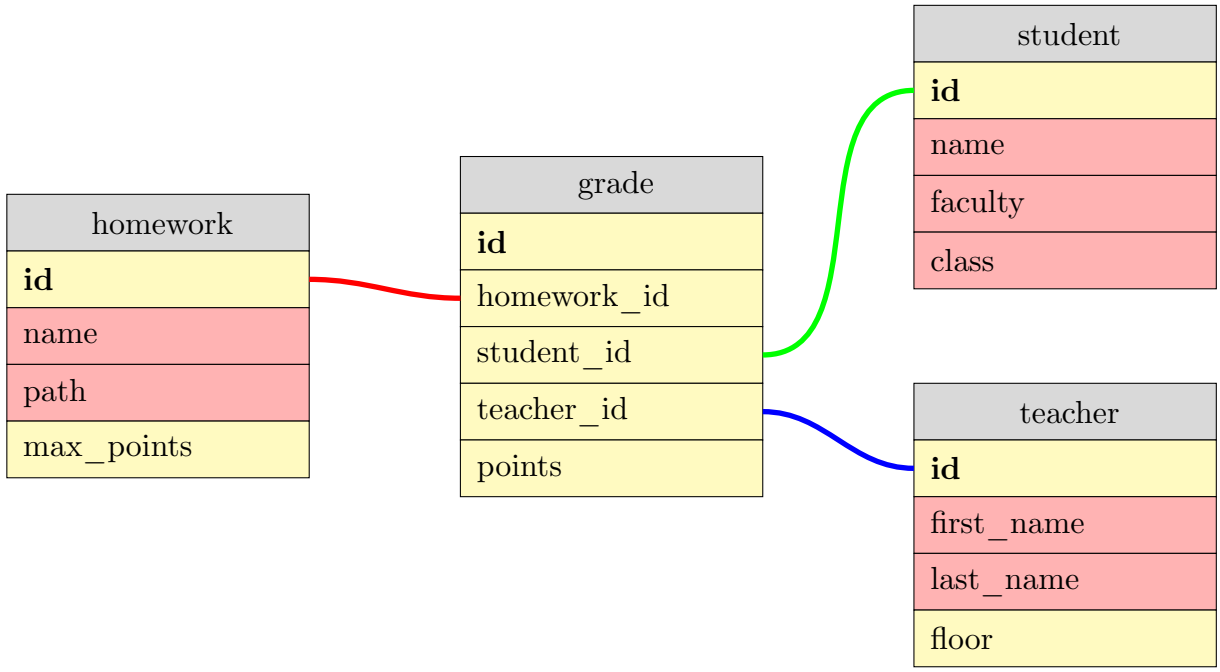

Figure 1: Схема базы данных

Реляционная база данных хранит данные в таблицах. Каждая таблица описывается в виде перечисления своих полей (столбцов таблицы) и записей, которые в ней хранятся. Например, в представленной выше таблице данные о студентах (имя, факультет и группа), учителях (имя и фамилия), домашних работах (название работы и путь к работе в репозитории) и оценках хранятся в таблицах student, teacher, homework и grade соответственно. При этом в таблице с оценками, которые поставлены студентам преподавателями за какие-то домашние работы, содержатся только идентификаторы студента, преподавателя и работы, а также собственно оценки.

Имея на руках все эти 4 таблицы, можно объединять данные в них. Например, можно получить все оценки определённого студента. Чтобы делать подобного рода запросы существует язык запросов — SQL. Это декларативный язык, то есть он описывает то, какие данные нужно получить, а не то, как получить эти данные. Следует, однако, заметить, что в различных базах данных реализация языка SQL может отличаться.

1.2 Примеры запросов на языке SQL

```sql
SELECT name, surname
FROM users
WHERE age > 18;
```

```sql
SELECT balance
FROM account
WHERE user_id = 81858
```

```sql
SELECT *
FROM users u JOIN accounts a
  ON u.id = a.user_id
WHERE account.balance > 0
```

1.3 Оператор SELECT

Простейший запрос на получение данных можно произвести с помощью оператора SELECT. Для этого нужно указать в запросе номера желаемых столбцов и имя таблицы, из которой будут получены данные.

```
SELECT id, name FROM students;
```

Чтобы вывести все существующие столбцы в качестве списка столбцов нужно написать символ «звездочка»:

```
SELECT * FROM students;
```

Так можно вывести на экран все находящиеся в базе таблицы.

Ключевое слово WHERE позволяет указать условие, которому должны удовлетворять требуемые строки:

```
SELECT * FROM grade WHERE point > 0;
```

Условие может быть достаточно сложным. Например, оператор LIKE позволяет проверять, подходит ли строка под указанный шаблон. Следующий запрос выводит список учителей, в имени которых содержится большая буква «B»:

```
SELECT * FROM teachers WHERE first_name LIKE '%B';
```

1.4 Оператор JOIN

Оператор JOIN позволяет выбирать данные из нескольких таблиц, чтобы представить их в виде одного результирующего набора. При этом необходимо явно задать условие соединения:

```
SELECT * FROM homework JOIN grade ON homework.id = grade.homework_id
```

Оператор JOIN может быть применён последовательно несколько раз подряд, если необходимо объединить более двух таблиц:

```
SELECT * FROM homework JOIN grade ON homework.id = grade.homework_id JOIN teacher ON teacher.id = grade.te
```

Порядок строк в результате произволен. Отсортировать полученные результаты можно с помощью конструкции ORDER BY:

```
SELECT * FROM teachers ORDER BY first_name;
```

Чтобы отсортировать в обратном порядке, следует добавить DESC следующим образом:

```
SELECT * FROM teachers ORDER BY first_name DESC;
```

1.5 Примеры запросов

2 Типы баз данных. Модуль DBI.

Модуль DBI описывает интерфейс, по которому должны работать все другие модули для связи с базами данных. Это позволяет унифицировать запросы к совершенно разным базам данных при программировании на perl. Однако специфика каждой конкретной базы данных не перестаёт играть роль.

2.1 Метод connect

Подключиться к базе данных можно с помощью метода connect модуля DBI, который в качестве своих аргументов принимает расположение базы данных, имя пользователя и пароль. В качестве четвёртого аргумента можно передать дополнительные параметры. В результате метод connect возвращает объект специального класса (database handler), через который будет происходить взаимодействие с базой:

```
$dbh = DBI->connect(
  $dsn, $user, $password,
  {RaiseError => 1, AutoCommit => 0}
);
```

В частности, метод do позволяет делать SQL запросы:

```
$dbh->do($sql);
```

В зависимости от того, с какой базой данных предстоит работать, расположение базы данных задаётся по-разному:

```
$dbh = DBI->connect($data_source,
    user, $password, {...});

# DBD::SQLite
$dbh = DBI->connect("dbi:SQLite:dbname=dbfile",
    "","");

# DBD::mysql
$dbh = DBI->connect(
    "DBI:mysql:database=$database;" .
        "host=$hostname;port=$port",
    $user, $password
);
```

Формат, в котором необходимо представить данные для подключения к конкретной базе данных, подробно описывается в документации к модулю, который обеспечивает взаимодействие с ней.

```
dbi:DriverName:database_name
dbi:DriverName:database_name@hostname:port
dbi:DriverName:database=DBNAME;host=HOSTNAME;port=PORT
```

2.2 Метод do

Как уже было сказано, метод do объекта database handler позволяет делать запросы к базе данных:

```
my $number_of_rows = $dbh->do(
    'DELETE FROM user WHERE age < 18
');
```

В результате выполнения данного кода из базы данных будут удалены все пользователи, возраст которых меньше 18 лет.

Удалить всех пользователей, которые имеют определённое имя, можно с помощью следующего кода:

```
my $name = <>;
$dbh->do("DELETE FROM user WHERE name = '$name'");
```

Этот код содержит уязвимость, известную как SQL injection.

2.3 SQL injection

Она заключается в том, что специальным образом составленное значение переменной может привести к исполнению произвольного действия:

```
my $name = q{' OR (DELETE FROM log) AND '' = '};

$dbh->do("DELETE FROM user WHERE name = '$name'");

DELETE FROM user WHERE name = ''
    OR (DELETE FROM log) AND '' = ''
```

Чтобы безопасно производить запросы, переменную необходимо заэкранировать с помощью метода quote. Экранированную переменную можно использовать в запросах, не опасаясь SQL инъекции:

```
$name = $dbh->quote($name);
```

Однако часто программисты забывают про это.

2.4 Методы prepare и execute

Более совершенный способ заключается в использовании методов prepare и execute:

```
my $sth = $dbh->prepare(
    'DELETE FROM user WHERE name = ?'
);
```

Метод prepare готовит запрос к исполнению. Поскольку символ ? не является валидным в SQL, он используется как метка параметров. При выполнении метода execute все такие метки заменяются на экранированные значения аргументов:

```
$sth->execute('Vadim');
```

Следует отметить, что в некоторых базах данных метод prepare реализован на стороне базы данных, а значит при вызове этого метода происходит обращение к базе. Это позволяет увеличить производительность в случае большого числа однотипных запросов.

2.5 Методы fetchrow и fetchall

После того, как метод execute выполнен, необходимо как-то извлечь выбранные в запросе данные. Методы fetchrow_arrayref, fetchrow_array и fetchrow_hashref возвращают данные по одной записи.

```
my $ary_ref  = $sth->fetchrow_arrayref();
my @ary      = $sth->fetchrow_array();
my $hash     = $sth->fetchrow_hashref();

while (@row = $sth->fetchrow_array()) {
  print "@row\n";
}
```

Разница между этими методами следующая: fetchrow_array возвращает запись как массив, fetchrow_arrayref — как ссылку на массив, fetchrow_hashref — как ссылку на хеш.

```
my $ary = $sth->fetchall_arrayref;
# [ [...], [...], [...] ]

my $ary = $sth->fetchall_arrayref({});
# [ {...}, {...}, {...} ]
```

Получить все данные результата запроса можно с помощью методов fetchall_arrayref и fetchall_hashref. Дополнительный параметр позволяет указать, в каком виде должны быть представлены записи.

Например, если передать в качестве этого параметра любой хеш, fetchall_arrayref вернет массив хешей, вызывая внутри себя метод fetchrow_hashref:

```
$tbl_ary_ref = $sth->fetchall_arrayref({
    foo => 1,
    BAR => 1,
});
```

В дополнительном параметре можно указать номера колонок или их имена. Чтобы указать, колонки с какими номерами необходимо вернуть, в качестве дополнительного параметра нужно передать массив с этими номерами. Отрицательные номера значат номер колонки с конца.

```
$tbl_ary_ref = $sth->fetchall_arrayref(
    [0]
);

$tbl_ary_ref = $sth->fetchall_arrayref(
    [-2,-1]
);
```

Ещё раз следует отметить, что именно тип параметра задаёт тип возвращаемых данных, а не сам параметр. В случае, когда нужно указать конкретные колонки и вернуть результат в виде массива хешей, в качестве дополнительного параметра необходимо передать хеш, значения в котором произвольны, а ключи — это в точности имена требуемых колонок:

```
$sth->fetchall_hashref('id');
# { 1 => {...}, 2 => {...} }
```

Метод fetchall_hashref всегда возвращает хеш, значения в котором тоже являются хешами. Ключом для некоторой строки будет являться id:

```
$sth->fetchall_hashref([ qw(foo bar) ]);

{
  1 => { a => {...}, b => {...} },
  2 => { a => {...}, b => {...} },
}
```

2.6 Методы selectrow и selectall

Методы selectrow_array, selectrow_arrayref, selectrow_hashref позволяют сразу сделать запрос и вернуть одну строчку:

```
$dbh->selectrow_array(
    $statement, \%attr, @bind_values
);

$dbh->selectrow_arrayref(
    $statement, \%attr, @bind_values
);

$dbh->selectrow_hashref(
    $statement, \%attr, @bind_values
);
```

Если ожидается много строчек в результате запроса, необходимо использовать один из методов selectall_array, selectall_arrayref, selectall_hashref:

```
$dbh->selectall_arrayref(
    $statement, \%attr, @bind_values);

$dbh->selectall_hashref(
    $statement, $key_field, \%attr, @bind_values);

$dbh->selectall_arrayref(
    "SELECT ename FROM emp ORDER BY ename",
    { Slice => {} }
);
```

2.7 Обработка ошибок

```
$dbh = DBI->connect(
  "dbi:DriverName:db_name", $user, $password,
  { RaiseError => 1 }
);
```

Когда ошибка происходит при исполнении запроса, данные об ошибке доступны через database handler. Код ошибки и текст ошибки можно узнать используя методы err и errstr:

```
$dbh->err;
$dbh->errstr;
```

Конкретные коды ошибки зависит от используемой базы данных.

2.8 Транзакции

Транзакции — группы запросов, для которых гарантируется их атомарное исполнение или неисполнение.

```
$dbh = DBI->connect(
  "dbi:DriverName:db_name", $user, $password,
  { AutoCommit => 1 }
);

$dbh->begin_work;
$dbh->rollback;
$dbh->commit;
```

Классический пример транзакции — перевод денег со счета на счёт. В этом случае операция «снять деньги с первого счета» и «положить деньги на второй счёт» должны быть или обе выполнены, или обе не выполнены.

Метод last_insert_id возвращает id последней созданной строки в текущей транзакции:

```
$dbh->do('INSERT INTO user VALUES(...)');

my $user_id = $dbh->last_insert_id(
    $catalog, $schema, $table, $field, \%attr
);
```

3 Object-relational mapping

Существует ряд подходов, которые заключаются в использовании классов, отвечающих за взаимодействие с базой. Это так называемый ORM (Object-relational mapping), который позволяет работать с данными в базе абстрагировано.

3.1 Модуль DBlx::Class

Один из таких слоёв абстракции обеспечивает модуль DBlx::Class. Для того, чтобы работать с базой данных через DBlx::Class, ему необходимо «объяснить», какие таблицы есть в базе и как они связаны между собой. После этого работа с данными будет происходить с использованием объектно-ориентированного подхода.

```
package Local::Schema::User;
use base qw(DBIx::Class::Core);

__PACKAGE__->table('user');
__PACKAGE__->add_columns(
    id => {
        data_type => 'integer',
        is_auto_increment => 1,
    },
    name => {
        data_type => 'varchar',
        size      => '100',
    },
    superuser => {
        data_type => 'bool',
    },
);
```

3.2 Метод resultset

Основные объекты, которыми манипулирует DBlx::Class, это resultset'ы. Их следует мыслить как потенциальный массив строк и как запрос. Например:

```
my $resultset = $schema->resultset('User');
my $resultset2 = $resultset->search({age => 25});
```

До тех пор, пока не будет выполнен метод next, реальных запросов к базе данных не производилось. До этого момента запрос был только сформирован, подготовлен, чтобы исполниться на базе. Метод next используется, чтобы получить данные от базы и выводить их построчно:

```
while (my $user = $resultset->next) {
  print $user->name . "\n";
}
```

Сразу все строки можно вернуть с помощью метода all:

```
print join "\n", $resultset2->all();
```

3.3 Метод search

После использования метода resultset значение, которое он возвратил, соответствует запросу на получение всей таблицы. Если необходимо указать условия на требуемые строки, их можно задать используя метод search:

```
$rs = $rs->search({
  age => {'>=' => 18},
  parent_id => undef,
});
```

В качестве параметра метод search принимает хеш с условиями на значения полей. Если в качестве значения к некоторому ключу также будет передан хеш, то его ключ будет воспринят как оператор, а значение — как операнд. Например:

```
@results = $rs->all();
@results = $rs->search(...);
$rs = $rs->search(...);
$rs = $rs->search_rs(...);
```

В качестве второго параметра можно задать сортировку результатов запроса и так далее:

```
$rs = $rs->search(
  { page => {'>=' => 18} },
  { order_by => { -desc => [qw(a b c)] } },
);
```

Если условия на значения полей не требуются, в качестве первого параметра следует отправить undef или пустой хеш. С помощью второго параметра можно ограничить число строк в результате следующим образом:

```
$rs = $rs->search(undef, {rows => 100});
```

Чтобы задать два условия на одно поле (хеш не может содержать два элемента с одинаковыми ключами), нужно использовать альтернативный синтаксис — каждую пару ключ-значение поместить в свой хеш и передать массив получившихся хешей:

```
# :-(
$rs = $rs->search({
  age => {'>=' => 18},
  age => {'<' => 60},
});

# :-)
$rs = $rs->search([
  { age => {'>=' => 18} },
  { age => {'<'  => 60} },
]);
```

3.4 Методы find, single

Метод find позволяет сразу вернуть не массив, а единственную строчку. Он поддерживает два синтаксиса: передать хеш, как в случае search, или непосредственно передать значение первичного ключа:

```
my $rs = $schema->resultset('User');

$user = $rs->find({id => 81858});
$user = $rs->find(81858);
```

Фактически find внутри себя делает следующее: вызывает метод search и сразу вызывает метод single:

```
$user = $rs->search({id => 81858})->single();
```

Метод single, если результат представляет из себя одну строку, возвращает ее, и, если результат пустой, возвращает undef. В случае, если результат содержит несколько строк, будет брошено предупреждение и возвращена первая строка.

3.5 Метод count

Метод count возвращает количество строк в результате:

```
my $count = $schema->resultset('User')->search({
  name => 'name',
  age => 18,
})->count();
```

Преимущество этого метода состоит в том, что не нужно выделять память для строк в результате, когда нужно знать только их количество.

3.6 Метод search: продолжение

Если при вызове метода search в хеше с условиями в качестве значения подать не строку, а ссылку на строку, SQL код в этой строке будет вызван как есть, а не заэкранирован:

```
$resultset->search({
  date => { '>' => \'NOW()' },
});
```

Если в качестве первого параметра метода search передать ссылку на массив, содержащий SQL запрос с метками параметров (вопросительные знаки) и значениями, которые необходимо поставить на место этих меток, этот запрос будет исполнен, причём все значения будут автоматически экранированы:

```
$rs->search(
  \[ 'YEAR(date_of_birth) = ?', 1979 ]
);
```

Также можно использовать or и and следующим образом:

```
my @albums = $schema->resultset('Album')->search({
  -or => [
    -and => [
      artist => { 'like', '%Smashing Pumpkins%' },
      title  => 'Siamese Dream',
    ],
    artist => 'Starchildren',
  ],
});
```

3.7 Связи между таблицами

Следующий пример демонстрирует как может быть задана связь один ко многим:

```
package Local::Schema::User;
use base qw(DBIx::Class::Core);

__PACKAGE__->table('user');
__PACKAGE__->has_many(
    dogs => 'Local::Schema::Dog',
    'user_id'
);

package Local::Schema::Dog;
use base qw(DBIx::Class::Core);

__PACKAGE__->table('dog');
__PACKAGE__->belongs_to(
    user => 'Local::Schema::User',
    'user_id'
);
```

Это позволяет, например, запрашивать всех собак, принадлежащих определённому пользователю с помощью вызова одного метода:

```
$user = $schema->resultset('User')->find(81858);

foreach my $dog ($user->dogs) {
  print join(' ', $dog->id, $dog->user->id);
}
```

Связи также можно использовать в запросах. В этом случае во втором параметре метода search необходимо указать используемую связь:

```
$rs = $schema->resultset('Dog')->search({
  'me.name' => 'Sharik',
  'user.name' => 'Vadim',
}, {
  join => 'user',
});
```

Хотя join и объединяет несколько таблиц, в полученных объектах содержатся данные лишь из одной таблицы, а чтобы получить данные из другой таблицы будет делаться дополнительный запрос. Получить все данные из связанных таблиц одним запросом можно, если вместо join использовать prefetch:

```
foreach my $user ($schema->resultset('User')) {
  foreach my $dog ($user->dogs) {
    # ...
  }
}

$rs = $schema->resultset('User')->search({}, {
  prefetch => 'dogs', # implies join
});
```

3.8 Custom result and resultset methods

В DBIx::Class можно в ResultSet-классы добавлять дополнительные методы для работы с наборами строк, в частности, поиск по заданным критериям и сортировку:

```
my @women = $schema->resultset('User')->
    search_women()->all();
```

Также можно добавить дополнительные методы для Result-классов:

```
package Local::Schema::ResultSet::User;

sub search_women {
  my ($self) = @_;

  return $self->search({
    gender => 'f',
  });
}
```

Это позволяет инкапсулировать часть логики внутри данных классов.

3.9 Методы new_result, create

Метод new_result ResultSet-класса возвращает новый объект. Этот объект не будет добавлен в базу данных до тех пор, пока не будет выполнен его метод insert. Например:

```
my $user = $schema->resultset('User')->new_result({
  name => 'Vadim',
  superuser => 1,
});

$user->insert();
```

Метод create ResultSet-класса позволяет за один вызов создать объект и поместить его в базу:

```
my $artist = $artist_rs->create(
   { artistid => 4, name => 'Blah-blah', cds => [
      { title => 'My First CD', year => 2006 },
      { title => 'e.t.c', year => 2007 },
    ],
   },
);
```

Интересно, что с помощью create можно сразу указать и связи. В этом случае необходимые строки в других таблицах базы данных будут созданы автоматически.

3.10 Методы update и delete

Метод update можно использовать после внесения изменения, чтобы закрепить его в базе данных:

```
$result->last_modified(\'NOW()')->update();
```

Также можно сразу передать хеш с данными, которые нужно изменить:

```
$result->update({ last_modified => \'NOW()' });
```

Метод delete позволяет удалить объект из базы данных.

```
$user->delete();
```

3.11 Связь многие ко многим

Часто таблица, которая используется для реализации связи многие-ко-многим, состоит всего из двух столбцов и сама по себе не интересна. Модуль DBIx::Class позволяет удобно работать с подобного рода связями:

3.12 Отладочный режим

Включить отладочный режим можно с помощью следующего метода:

```
$schema->storage->debug(1);
```

В этом случае результаты всех запросов будут выведены в stdout.

DBIx::Class внутри себя использует модуль DBI, о котором была речь в начале лекции. Получить непосредственно доступ к database handler можно следующим образом:

```
$schema->storage->dbh();
```

3.13 Генерирование схемы на основе готовой базы

Класс DBIx::Class::Schema::Loader позволяет с помощью функции make_schema_at сгенерировать схему на основе готовой базы:

```
use DBIx::Class::Schema::Loader qw(
    make_schema_at
);

make_schema_at(
    'My::Schema',
    { debug => 1,
      dump_directory => './lib',
    },
    [ 'dbi:Pg:dbname="foo"', 'user', 'pw' ]
);
```

Этот класс подписывает сгенерированные файлы контрольной суммой, чтобы в следующий раз он мог сверху накатить изменения и не сломать добавленный вручную код. Если схема была изменена вручную, такое автоматическое обновление становится недоступным.

Ручное редактирование схемы имеет следующие преимущества:

- Колонкам можно давать произвольные имена.
- Схеме можно не «рассказывать» о некоторых колонках (которые база данных обслуживает самостоятельно).
- Можно указывать разные настройки для разных таблиц.

Для того, чтобы перейти в режим ручного редактирования, достаточно просто удалить контрольные суммы из файлов.

Кроме модуля существует утилита dbicdump, которая позволяет делать то же самое без написания perl-скрипта:

```
dbicdump -o dump_directory=./lib \
         -o debug=1 \
         My::Schema \
         'dbi:Pg:dbname=foo' \
         myuser \
         mypassword
```

3.14 SQL::Translator

Возможен другой подход — создать схему, а потом создать все необходимые таблицы и связи в базе данных с помощью следующей команды:

```
$schema->deploy();
```

При этом следует учитывать, что не все возможности базы данных могут быть задействованы при таком способе.

3.15 Изменение схемы в режиме реального времени

Изменение схемы в ходе работы сервиса представляет из себя серьёзную проблемы. Во-первых, для достаточно больших таблиц создание новой колонки с некоторым значением по умолчанию занимает много времени. Чтобы внести такие изменения, не останавливая работу сервиса, изменение схемы происходит в несколько этапов:

1. Добавить колонку с пустым значением по умолчанию.
2. Код переписывается таким образом, чтобы он мог работать и со старой схемой, и с новой.
3. Постепенно вносить данные и заполнять стандартное значение.

Иногда бывает разумнее (если сервис небольшого размера) приостановить работу сервиса, внести изменения и запустить снова. Но для сколько-нибудь крупных проектов это неприемлемо.

4 Memcached

Memcached — примитивное хранилище пар ключ-значение. Memcached в этом смысле не совсем является базой данных. В Memcached доступны несколько команд — для доступа к данным и добавления данных. Для каждой пары ключ-значение можно задать время хранение, что позволяет использовать Memcached в качестве кэша (откуда и название).

Модуль Cache::Memcached::Fast позволяет хранить данные на нескольких серверах (weight задает вес этого сервера, данные располагаются на серверах пропорционально значениям весов). Например:

```
use Cache::Memcached::Fast;

my $memd = Cache::Memcached::Fast->new({
  servers => [
    {address => 'localhost:11211', weight => 2.5},
    '192.168.254.2:11211',
    '/path/to/unix.sock'
  ],
  namespace => 'my:',
  connect_timeout => 0.2,
  # ...
});
```

После этого можно использовать 4 основных операции:

```
$memd->add('skey', 'text');
$memd->set('nkey', 5, 60);
$memd->incr('nkey');
$memd->get('skey');
```

При этом операция инкремента обеспечивает атомарность. Атомарность обеспечивается следующим образом. При чтении данных возвращается значение и специальный ключ. Этот ключ можно использовать, чтобы выполнить set, но только в том случае, если ключ не изменился (ключ меняется при изменении данных). Если ключ изменился, данные нужно будет считать заново, обработать и вновь попытаться сохранить. Это будет продолжаться до тех пор, пока имеет место такая «гонка».