PHP has supported MySQL almost since the project’s inception, including an API with the version 2 release. In fact, using MySQL with PHP eventually became so commonplace that for several years the extension was enabled by default. But perhaps the most indicative evidence of the strong bonds between the two technology camps was the release of an updated MySQL extension with PHP 5, known as MySQL Improved (and typically referred to as mysqli).

So why the need for a new extension? The reason is twofold. First, MySQL’s rapid evolution prevented users who were relying on the original extension from taking advantage of new features such as prepared statements, advanced connection options, and security enhancements. Second, while the original extension certainly served programmers well, many considered the procedural interface outdated, preferring a native object-oriented interface that would not only more tightly integrate with other applications, but also offer the ability to extend that interface as desired. To resolve these deficiencies, the MySQL developers decided it was time to revamp the extension, not only changing its internal behavior to improve performance, but also incorporating additional capabilities to facilitate the use of features available only with these newer MySQL versions. A detailed list of the key enhancements follows:

• Object oriented: The mysqli extension is encapsulated within a series of classes, encouraging use of what many consider to be a more convenient and efficient programming paradigm than PHP’s traditional procedural approach. However, those preferring to embrace a procedural programming paradigm aren’t out of luck, as a traditional procedural interface is also provided (although it won’t be covered in this chapter). • Prepared statements: Prepared statements eliminate overhead and inconvenience when working with queries intended for repeated execution, as is so often the case when building database-driven web sites. Prepared statements also offer another important security-related feature in that they prevent SQL injection attacks.

• Transactional support: Although MySQL’s transactional capabilities are available in PHP’s original MySQL extension, the mysqli extension offers an object-oriented interface to these capabilities. The relevant methods are introduced in this chapter, and Chapter 37 provides a complete discussion of this topic.

• Enhanced debugging capabilities: The mysqli extension offers numerous methods for debugging queries, resulting in a more efficient development process.

• Embedded server support: An embedded MySQL server library was introduced with the 4.0 release for users who are interested in running a complete MySQL server within a client application such as a kiosk or desktop program. The mysqli www.it-ebooks.info CHAPTER 30  USING PHP WITH MYSQL 588 extension offers methods for connecting and manipulating these embedded MySQL databases.

Master/slave support: As of MySQL 3.23.15, MySQL offers support for replication, although in later versions this feature has been improved substantially. Using the mysqli extension, you can ensure queries are directed to the master server in a replication configuration.

Those familiar with the original MySQL extension will find the enhanced mysqli extension quite familiar because of the almost identical naming conventions. For instance, the database connection function is titled mysqli\_connect() rather than mysql\_connect(). Furthermore, all parameters and behavior for similar functions are otherwise externally identical to its predecessor.

1 Installation Prerequisites

As of PHP 5, MySQL support is no longer bundled with the standard PHP distribution. Therefore, you need to explicitly configure PHP to take advantage of this extension. In this section, you learn how to do so for both the Unix and Windows platforms.

1.1 Enabling the mysqli Extension on Linux/Unix

Enabling the mysqli extension on the Linux/Unix platform is accomplished by configuring PHP using the --with-mysqli flag. This flag should point to the location of the mysql\_config program available to MySQL 4.1 and greater.

1.2 Enabling the mysqli Extension on Windows

To enable the mysqli extension on Windows, you need to uncomment the following line from the php.ini file, or add it if it doesn’t exist:

extension=php\_mysqli.dll

As is the case before enabling any extension, make sure PHP’s extension\_dir directive points to the appropriate directory. See Chapter 2 for more information regarding configuring PHP.

1.3 Using the MySQL Native Driver

Historically, PHP required that a MySQL client library be installed on the server from which PHP was communicating with MySQL, whether the MySQL server also happened to reside locally or elsewhere. PHP 5.3 removes this inconvenience by introducing a new MySQL driver named the MySQL Native Driver (also known as mysqlnd) that offers many advantages over its predecessors. The MySQL Native Driver is not a new API, but rather is a new conduit that the existing APIs (mysql, mysqli, and PDO\_MySQL) can use in order to communicate with a MySQL server. Written in C, tightly integrated into PHP’s architecture, and released under the PHP license, I recommend using mysqlnd over the alternatives unless you have good reason for not doing so.

To use mysqlnd in conjunction with one of the existing extensions, you’ll need to recompile PHP, including an appropriate flag. For instance, to use the mysqli extension in conjunction with the mysqlnd driver, pass the following flag:

--with-mysqli=mysqlnd

If you plan on using both the PDO\_MySQL and mysqli extensions, there’s nothing stopping you from specifying both when compiling PHP:

%>./configure --with-mysqli=mysqlnd --with-pdo-mysql=mysqlnd [other options]

The mysqlnd driver does suffer from some limitations. Currently it does not offer compression or SSL support. Be sure to check the MySQL documentation at

http://dev.mysql.com/downloads/connector/php-mysqlnd for the latest information.

1.4 Managing User Privileges

The constraints under which PHP interacts with MySQL are no different from those required of any other interface. A PHP script intent on communicating with MySQL must still connect to the MySQL server and select a database to interact with. All such actions, in addition to the queries that would follow such a sequence, can be carried out only by a user possessing adequate privileges.

These privileges are communicated and verified when a script initiates a connection to the MySQL server, as well as every time a command requiring privilege verification is submitted. However, you need to identify the executing user only at the time of connection; unless another connection is made later within the script, that user’s identity is assumed for the remainder of the script’s execution. In the coming sections, you’ll learn how to connect to the MySQL server and pass along these credentials.

1.4 Working with Sample Data

Learning a new topic tends to come easier when the concepts are accompanied by a set of cohesive examples. Therefore, the following table, products, located within a database named corporate, is used for all relevant examples in the following pages:

CREATE TABLE products (

id INT NOT NULL AUTO\_INCREMENT,

sku VARCHAR(8) NOT NULL,

name VARCHAR(100) NOT NULL,

price DECIMAL(5,2) NOT NULL,

PRIMARY KEY(id)

)

The table is populated with the following four rows:

Image 1 Image 2

图1.商品表

2 Using the mysqli Extension

PHP’s mysqli extension offers all of the functionality provided by its predecessor, in addition to new features that have been added as a result of MySQL’s evolution into a full-featured database server. This section introduces the entire range of features, showing you how to use the mysqli extension to connect to the database server, query for and retrieve data, and perform a variety of other important tasks.

2.1 Setting Up and Tearing Down the Connection

Interaction with the MySQL database is bookended by connection setup and teardown, consisting of connecting to the server and selecting a database, and closing the connection, respectively. As is the case with almost every feature available to mysqli, you can do this by using either an object-oriented approach or a procedural approach, although throughout this chapter only the object-oriented approach is covered.

If you choose to interact with the MySQL server using the object-oriented interface, you need to first instantiate the mysqli class via its constructor:

mysqli([string host [, string username [, string pswd

[, string dbname [, int port, [string socket]]]]]])

Those of you who have used PHP and MySQL in years past will notice this constructor accepts many of the same parameters as does the traditional mysql\_connect() function.

Instantiating the class is accomplished through standard object-oriented practice:

$mysqli = new mysqli('localhost', 'catalog\_user', 'secret', 'corporate');

Once the connection has been made, you can start interacting with the database. If at one point you need to connect to another database server or select another database, you can use the connect() and select\_db() methods. The connect() method accepts the same parameters as the constructor, so let’s just jump right to an example:

// Instantiate the mysqli class

$mysqli = new mysqli();

// Connect to the database server and select a database

$mysqli->connect('localhost','catalog\_user','secret','corporate');

You can choose a database using the $mysqli->select\_db method. The following example connects to a MySQL database server and then selects the corporate database:

// Connect to the database server

$mysqli = new mysqli('localhost', 'catalog\_user', 'secret');

// Select the database

$mysqli->select\_db('corporate');

Once a database has been successfully selected, you can then execute database queries against it. Executing queries, such as selecting, inserting, updating, and deleting information with the mysqli extension, is covered in later sections.

Once a script finishes execution, any open database connections are automatically closed and the resources are recuperated. However, it’s possible that a page requires several database connections throughout the course of execution, each of which should be closed as appropriate. Even in the case where a single connection is used, it’s nonetheless good practice to close it at the conclusion of the script. In any case, close() is responsible for closing the connection. An example follows:

$mysqli = new mysqli();

$mysqli->connect('localhost', 'catalog\_user', 'secret', 'corporate');

// Interact with the database…

// close the connection

$mysqli->close()

2.1 Handling Connection Errors

Of course, if you’re unable to connect to the MySQL database, then little else on the page is going to happen as planned. Therefore, you should be careful to monitor connection errors and react accordingly. The mysqli extension includes a few features that can be used to capture error messages, or alternatively you can use exceptions (as introduced in Chapter 8). For example, you can use the mysqli\_connect\_errno() and mysqli\_connect\_error() methods to diagnose and display information about a MySQL connection error.

2.3 Retrieving Error Information

Developers always strive toward that nirvana known as bug-free code. In all but the most trivial of projects, however, such yearnings are almost always left unsatisfied. Therefore, properly detecting errors and returning useful information to the user is a vital component of efficient software development. This section introduces two functions that are useful for deciphering and communicating MySQL errors.

2.4 Storing Connection Information in a Separate File

In the spirit of secure programming practice, it’s often a good idea to change passwords on a regular basis. Yet, because a connection to a MySQL server must be made within every script requiring access to a given database, it’s possible that connection calls may be strewn throughout a large number of files, making such changes difficult. The easy solution to such a dilemma should not come as a surprise— store this information in a separate file and then include that file in your script as necessary. For example, the mysqli constructor might be stored in a header file named mysql.connect.php, like so:

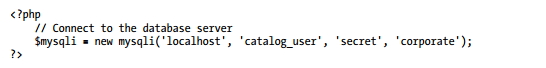


图2

This file can then be included as necessary, like so:

Image 6

图3

3 Interacting with the Database

The vast majority of your queries will revolve around creation, retrieval, update, and deletion tasks, collectively known as CRUD. This section shows you how to formulate and send these queries to the database for execution.

3.1 Sending a Query to the Database

The method query() is responsible for sending the query to the database. Its prototype looks like this:

class mysqli {

mixed query(string query [, int resultmode])

}

The optional resultmode parameter is used to modify the behavior of this method, accepting two values:

• MYSQLI\_STORE\_RESULT: Returns the result as a buffered set, meaning the entire set will be made available for navigation at once. This is the default setting. While this option comes at a cost of increased memory demands, it does allow you to work with the entire result set at once, which is useful when you’re trying to analyze or manage the set. For instance, you might want to determine how many rows are returned from a particular query, or you might want to immediately jump to a particular row in the set.

• MYSQLI\_USE\_RESULT: Returns the result as an unbuffered set, meaning the set will be retrieved on an as-needed basis from the server. Unbuffered result sets increase performance for large result sets, but disallow the opportunity to do various things with the result set, such as immediately determine how many rows have been found by the query or travel to a particular row offset. You should consider using this option when you’re trying to retrieve a very large number of rows because it will require less memory and produce a faster response time.

3.2 Retrieving Data

Chances are your application will spend the majority of its efforts retrieving and formatting requested data. To do so, you’ll send the SELECT query to the database, and then iterate over the results, outputting each row to the browser, formatted in any manner you please.

The following example retrieves the sku, name, and price columns from the products table, ordering the results by name. Each row of results is then placed into three appropriately named variables, and output to the browser.

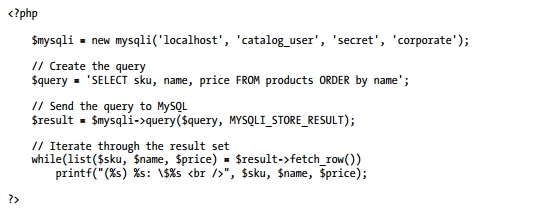


图4.获取数据

Executing this example produces the following browser output:



图5.浏览器输出

Keep in mind that executing this example using an unbuffered set would on the surface operate identically (except that resultmode would be set to MYSQLI\_USE\_RESULT instead), but the underlying behavior would indeed be different.

3.3 Recuperating Query Memory

On the occasion you retrieve a particularly large result set, it’s worth recuperating the memory required by that set once you’ve finished working with it. The free() method handles this task for you. Its prototype looks like this:

class mysqli\_result {

void free()

}

The free() method recuperates any memory consumed by a result set. Keep in mind that once this method is executed, the result set is no longer available. An example follows:

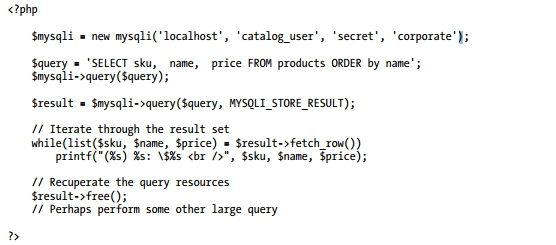


图6.释放内存集

4 Summary

The mysqli extension offers not only an expanded array of features over its older sibling, but—when used in conjunction with the new mysqlnd driver— unparalleled stability and performance.

In the next chapter you’ll learn all about PDO, yet another powerful database interface that is increasingly becoming the ideal solution for many PHP developers.