**[译]配置 sql.DB 获得更好的性能**

原文2020年4月9日已更新，请阅读[最新译文](https://colobu.com/2020/05/18/configuring-sql-DB-for-better-performance-2020/)

原文: [Configuring sql.DB for Better Performance](https://www.alexedwards.net/blog/configuring-sqldb)

网上有很多教程介绍sql.DB, 以及如何使用它来执行SQL数据库查询和语句, 但是大部分都没有介绍SetMaxOpenConns（）、SetMaxIdleConns（）和SetConnmaxLifetime（）方法。事实上你可以使用这些方法来配置sql.DB的行为并改善其性能。

在这篇文章中，我想准确地解释这些设置的作用，并演示它们可能产生的（正面和负面）影响。

**打开和空闲连接**

首先说一点背景知识。

sql.db对象是包含多个**open**和**idle**数据库连接的连接池。当使用连接执行数据库任务（如执行SQL语句或查询数据）时，该连接被标记为**open**(打开)。任务完成后，连接将变为**idle**(空闲)。

当您指示sql.db执行数据库任务时，它将首先检查池中是否有空闲连接可用。如果有可用的连接，Go将重用现有连接，并在任务期间将其标记为打开。如果在需要连接时池中没有空闲连接的话，go将创建一个新的附加连接并**打开**它。

**SetMaxOpenConns 方法**

默认情况下，可以同时打开的连接数没有限制。但您可以通过setMaxOpenConns（）方法实现自己的限制，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 初始化一个新的连接池  db, err := sql.Open("postgres", "postgres://user:pass@localhost/db")  if err != nil {  log.Fatal(err)  }  // 设置最大的并发打开连接数为5。  // 设置这个数小于等于0则表示没有显示，也就是默认设置。  db.SetMaxOpenConns(5) |

在此示例代码中，池中最多有5个并发打开的连接。如果5个连接都已经打开被使用，并且应用程序需要另一个连接的话，那么应用程序将被迫等待，直到5个打开的连接其中的一个被释放并变为空闲。

为了说明更改**MaxOpenConns**的影响，我运行了一个基准测试，将最大开放连接设置为1、2、5、10和无限制。基准测试在PostgreSQL数据库上执行并行的insert语句，您可以在这个[gist](https://gist.github.com/alexedwards/5d1db82e6358b5b6efcb038ca888ab07" \t "_blank)中找到代码。结果如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | BenchmarkMaxOpenConns1-8 500 3129633 ns/op 478 B/op 10 allocs/op  BenchmarkMaxOpenConns2-8 1000 2181641 ns/op 470 B/op 10 allocs/op  BenchmarkMaxOpenConns5-8 2000 859654 ns/op 493 B/op 10 allocs/op  BenchmarkMaxOpenConns10-8 2000 545394 ns/op 510 B/op 10 allocs/op  BenchmarkMaxOpenConnsUnlimited-8 2000 531030 ns/op 479 B/op 9 allocs/op  PASS |

准确地说，此基准的目的不是模拟应用程序的“真实”行为。它只是帮助说明SQL.DB在幕后的行为，以及更改**MaxOpenConns**对该行为的影响。

对于这个基准，我们可以看到允许的开放连接越多，在数据库上执行**插入**操作所花费的时间就越少（3129633 ns/op，其中1个开放连接，而无限连接为531030 ns/op，大约快6倍）。这是因为存在的开放连接越多，基准代码等待开放连接释放并再次空闲（准备使用）所需的时间（平均值）就越少。

**SetMaxIdleConns**

默认情况下，sql.DB允许在连接池中最多保留**2**个空闲连接。您可以通过SetMaxIdleConns（）方法进行更改，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 初始化连接池  db, err := sql.Open("postgres", "postgres://user:pass@localhost/db")  if err != nil {  log.Fatal(err)  }  // 设置最大的空闲连接数为5。  // 设置小于等于0的数意味着不保留空闲连接。  db.SetMaxIdleConns(5) |

理论上，在池中允许更多的空闲连接将提高性能，因为这样可以减少从头开始建立新连接的可能性，从而有助于节省资源。

让我们来看看相同的基准，最大空闲连接设置为无、1、2、5和10（并且开放连接的数量是无限的）：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | BenchmarkMaxIdleConnsNone-8 300 4567245 ns/op 58174 B/op 625 allocs/op  BenchmarkMaxIdleConns1-8 2000 568765 ns/op 2596 B/op 32 allocs/op  BenchmarkMaxIdleConns2-8 2000 529359 ns/op 596 B/op 11 allocs/op  BenchmarkMaxIdleConns5-8 2000 506207 ns/op 451 B/op 9 allocs/op  BenchmarkMaxIdleConns10-8 2000 501639 ns/op 450 B/op 9 allocs/op  PASS |

当MaxIdleConns设置为none时，必须为每个插入操作创建新的连接，从基准中我们可以看到平均运行时间和内存分配相对较高。

只允许保留和重用一个空闲连接，在我们这个特定的基准测试中有很大的不同——它将平均运行时间减少了8倍左右，并将内存分配减少了20倍左右。继续增加空闲连接池的大小会使性能更好，尽管这些改进不那么明显。

那么我们应该维护一个大的空闲连接池吗？答案是它取决于应用程序。

重要的是要认识到保持空闲连接的存活是要付出代价的——它会占用内存，否则这些内存可以同时用于应用程序和数据库。

也有一种可能，如果一个连接空闲太久，那么它也可能会变得不可用。例如，MySQL的[wait\_timeout](https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html" \l "sysvar_wait_timeout" \t "_blank)设置将自动关闭8小时内未使用的任何连接（默认情况下）。

当发生这种情况时，sql.DB会优雅地处理它。在放弃之前，将自动重试两次坏连接，之后Go将从池中删除坏连接并创建新连接。因此，将MaxIdleConns设置得太高实际上可能会导致连接变得不可用，并且使用的资源比使用较小的空闲连接池（使用的连接更少，使用频率更高）的情况下要多。所以只有你很可能马上再次使用浙西连接，你才会保持这些连接空闲。

最后要指出的一点是，MaxIdleConns应该始终小于或等于MaxOpenConns。Go会检查并在必要时自动减少MaxIdleConns StackOverflow上的一个解释很好地描述了原因：

设置比MaxOpenConns更多的空闲连接数是没有意义的，因为你最多也就能拿到所有打开的连接，剩余的空闲连接依然保持的空闲。这就像一座四车道的桥，但是只允许三辆车同时通过。

**SetConnMaxLifetime 方法**

现在让我们来看一下SetConnMaxLifetime（）方法，它设置了连接可重用的最大时间长度。如果您的SQL数据库也实现了最大的连接生存期，或者（例如）您希望在负载均衡器后面方便地切换数据库，那么这将非常有用。

您可以这样使用它：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 初始化连接池  db, err := sql.Open("postgres", "postgres://user:pass@localhost/db")  if err != nil {  log.Fatal(err)  }  // 设置连接的最大生命周期为一小时。  // 设置为0的话意味着没有最大生命周期，连接总是可重用(默认行为)。  db.SetConnMaxLifetime(time.Hour) |

在这个例子中，我们的所有连接将在第一次创建后1小时“过期”，并且在它们过期后无法重用。但是注意：

* 这并不能保证连接将在池中存在完整的一小时；很可能由于某种原因连接将变得不可用，并且在此之前自动关闭。
* 一个连接在创建后仍可以使用一个多小时，只是说一个小时后不能再被重用了。
* 这不是空闲超时。连接将在第一次创建后1小时后过期，而不是1小时后变成空闲。
* 每秒自动运行一次清理操作以便从池中删除“过期”连接。

理论上，**ConnMaxLifetime**越短，从零开始创建连接的频率就越高。

为了说明这一点，我运行了基准测试，将**ConnMaxLifetime**设置为100ms、200ms、500ms、1000ms和unlimited（永远重复使用），默认设置为unlimited open connections和2个idle connections。这些时间段显然比您在大多数应用程序中使用的要短得多，但它们有助于很好地说明连接库的行为。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | BenchmarkConnMaxLifetime100-8 2000 637902 ns/op 2770 B/op 34 allocs/op  BenchmarkConnMaxLifetime200-8 2000 576053 ns/op 1612 B/op 21 allocs/op  BenchmarkConnMaxLifetime500-8 2000 558297 ns/op 913 B/op 14 allocs/op  BenchmarkConnMaxLifetime1000-8 2000 543601 ns/op 740 B/op 12 allocs/op  BenchmarkConnMaxLifetimeUnlimited-8 3000 532789 ns/op 412 B/op 9 allocs/op  PASS |

在这些特定的基准测试中，我们可以看到100毫秒的内存分配要比unlimited的内存分配多三倍，而且每个插入的操作的平均运行时间也稍长一些。

**超出连接限制**

最后，如果不提及超过了数据库连接数的硬限制的话，那么本文就不算一个完整的教程了。

如图所示，我将更改**postgresql.conf**文件，因此只允许总共5个连接（默认值为100）…

|  |  |
| --- | --- |
| 1 | max\_connections = 5 |

使用 unlimited open connections 的配置进行基准测试：

|  |  |
| --- | --- |
| 1  2  3  4  5 | BenchmarkMaxOpenConnsUnlimited-8 --- FAIL: BenchmarkMaxOpenConnsUnlimited-8  main\_test.go:14: pq: sorry, too many clients already  main\_test.go:14: pq: sorry, too many clients already  main\_test.go:14: pq: sorry, too many clients already  FAIL |

一旦达到5个连接的硬限制，我的数据库驱动程序（PQ）立即返回一条sorry, too many clients already错误信息，而不是完成插入操作。

为了避免这个错误，我们需要将sql.DB中打开和空闲连接的最大总数设置为5以下。像这样：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 初始化连接池  db, err := sql.Open("postgres", "postgres://user:pass@localhost/db")  if err != nil {  log.Fatal(err)  }  //设置open和idle的总连接数为3  db.SetMaxOpenConns(2)  db.SetMaxIdleConns(1) |

现在，由sql.DB创建的连接数最多只能有3个，基准测试运行时应该没有错误。

但是这样也会给我们带来一个很大的警示：当达到开放连接限制时，应用程序需要执行的任何新数据库任务都将被强制等待，直到连接变为空闲。

对于某些应用程序，该行为可能很好，但对于其他应用程序，则可能不好。例如，在Web应用程序中，最好立即记录错误消息并向用户发送500 Internal Server Error，而不是让他们的HTTP请求挂起，并可能在等待空闲连接时超时。