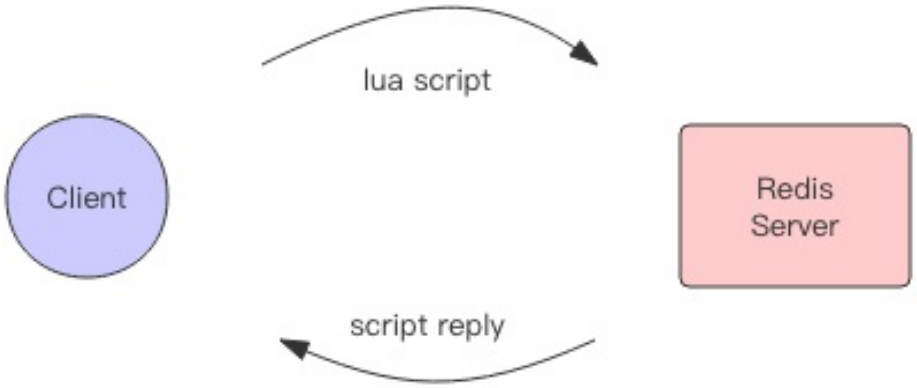
Redis 提供了非常丰富的指令集，但是用户依然不满足，希望可以自定义扩充若干指令来完成一些特定领域的问题。Redis 为这样的用户场景提供了 lua 脚本支持，用户可以向服务器发送 lua 脚本来执行自定义动作，获取脚本的响应数据。Redis 服务器会单线程原子性执行 lua 脚本，保证 lua 脚本在处理的过程中不会被任意其它请求打断。



比如在分布式锁小节，我们提到了 del_if_equals 伪指令，它可以将匹配 key 和删除 key 合并在一起原子性执行，Redis 原生没有提供这样功能的指令，它可以使用 lua 脚本来完成。

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

那上面这个脚本如何执行呢？使用 EVAL 指令

```
127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> eval 'if
redis.call("get",KEYS[1]) == ARGV[1] then return
redis.call("del",KEYS[1]) else return 0 end' 1
foo bar
(integer) 1
127.0.0.1:6379> eval 'if
redis.call("get",KEYS[1]) == ARGV[1] then return
redis.call("del",KEYS[1]) else return 0 end' 1
foo bar
(integer) 0
```

EVAL 指令的第一个参数是脚本内容字符串，上面的例子我们将 lua 脚本压缩成一行以单引号围起来是为了方便命令行执行。然后是 key 的数量以及每个 key 串，最后是一系列附加参数字符串。附加参数的数量不需要和 key 保持一致，可以完全没有附加参数。

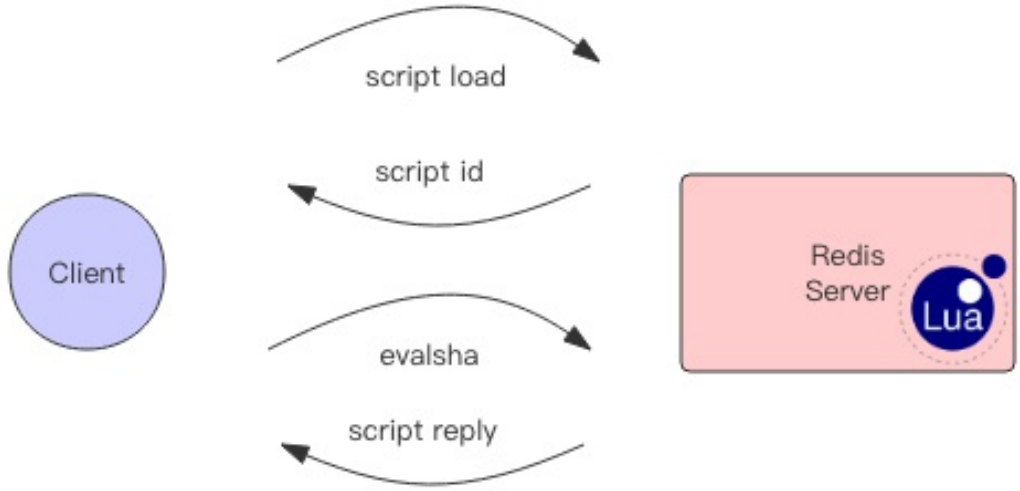
```
EVAL SCRIPT KEY_NUM KEY1 KEY2 ... KEYN ARG1 ARG2
....
```

上面的例子中只有 1 个 key，它就是 foo，紧接着 bar 是唯一的附加参数。在 lua 脚本中，数组下标是从 1 开始，所以通过 KEYS[1] 就可以得到 第一个 key，通过 ARGV[1] 就可以得到第一个附加参数。redis.call 函数可以让我们调用 Redis 的原生指令，上面的代码分别调用了 get 指令和 del 指令。return 返回的结果将会返回给客户端。

SCRIPT LOAD 和 EVALSHA 指令

在上面的例子中，脚本的内容很短。如果脚本的内容很长，而且客户端需要频繁执行，那么每次都需要传递冗长的脚本内容势必比较浪费网络流量。所以 Redis 还提供了 SCRIPT LOAD 和 EVALSHA 指令

来解决这个问题。



SCRIPT LOAD 指令用于将客户端提供的 lua 脚本传递到服务器而不执行，但是会得到脚本的唯一 ID，这个唯一 ID 是用来唯一标识服务器缓存的这段 lua 脚本，它是由 Redis 使用 sha1 算法揉捏脚本内容而得到的一个很长的字符串。有了这个唯一 ID，后面客户端就可以通过 EVALSHA 指令反复执行这个脚本了。
我们知道 Redis 有 incrby 指令可以完成整数的自增操作，但是没有提供自乘这样的指令。

```
incrby key value  ==> $key = $key + value
mulby key value ==> $key = $key * value
```

下面我们使用 SCRIPT LOAD 和 EVALSHA 指令来完成自乘运算。

```
local curVal = redis.call("get", KEYS[1])
if curVal == false then
  curVal = 0
else
  curVal = tonumber(curVal)
end
curVal = curVal * tonumber(ARGV[1])
redis.call("set", KEYS[1], curVal)
return curVal
```

先将上面的脚本单行化，语句之间使用分号隔开

```
local curVal = redis.call("get", KEYS[1]); if
curVal == false then curVal = 0 else curVal =
tonumber(curVal) end; curVal = curVal *
tonumber(ARGV[1]); redis.call("set", KEYS[1],
curVal); return curVal
```

加载脚本

```
127.0.0.1:6379> script load 'local curVal =
redis.call("get", KEYS[1]); if curVal == false
then curVal = 0 else curVal = tonumber(curVal)
end; curVal = curVal * tonumber(ARGV[1]);
redis.call("set", KEYS[1], curVal); return
curVal'
"be4f93d8a5379e5e5b768a74e77c8a4eb0434441"
```

命令行输出了很长的字符串，它就是脚本的唯一标识，下面我们使用这个唯一标识来执行指令

```
127.0.0.1:6379> evalsha
be4f93d8a5379e5e5b768a74e77c8a4eb0434441 1
notexistskey 5
(integer) 0
127.0.0.1:6379> evalsha
be4f93d8a5379e5e5b768a74e77c8a4eb0434441 1
notexistskey 5
(integer) 0
127.0.0.1:6379> set foo 1
OK
127.0.0.1:6379> evalsha
be4f93d8a5379e5e5b768a74e77c8a4eb0434441 1 foo 5
(integer) 5
127.0.0.1:6379> evalsha
be4f93d8a5379e5e5b768a74e77c8a4eb0434441 1 foo 5
(integer) 25
```

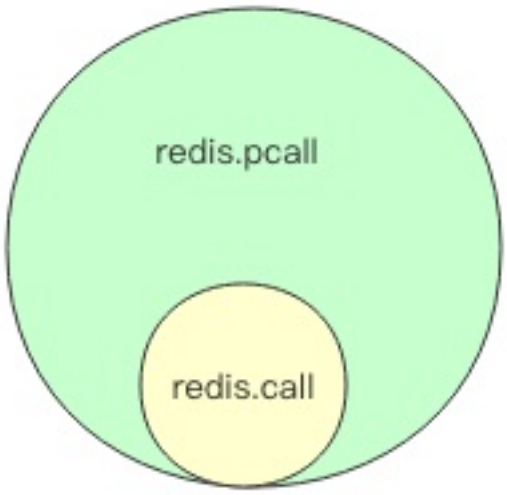
错误处理

上面的脚本参数要求传入的附加参数必须是整数，如果没有传递整数会怎样呢?

```
127.0.0.1:6379> evalsha
be4f93d8a5379e5e5b768a74e77c8a4eb0434441 1 foo
bar
(error) ERR Error running script (call to
f_be4f93d8a5379e5e5b768a74e77c8a4eb0434441):
@user_script:1: user_script:1: attempt to perform
arithmetic on a nil value
```

可以看到客户端输出了服务器返回的通用错误消息，注意这是一个动态抛出的异常，Redis 会保护主线程不会因为脚本的错误而导致服务器崩溃，近似于在脚本的外围有一个很大的 try catch 语句包裹。在

lua 脚本执行的过程中遇到了错误，同 redis 的事务一样，那些通过 redis.call 函数已经执行过的指令对服务器状态产生影响是无法撤销的，在编写 lua 代码时一定要小心，避免没有考虑到的判断条件导致脚本没有完全执行。



如果读者对 lua 语言有所了解就知道 lua 原生没有提供 try catch 语句，那上面提到的异常包裹语句究竟是用什么来实现的呢？lua 的替代方案是内置了 pcall(f) 函数调用。pcall 的意思是 protected call，它会让 f 函数运行在保护模式下，f 如果出现了错误，pcall 调用会返回 false 和错误信息。而普通的 call(f) 调用在遇到错误时只会向上抛出异常。在 Redis 的源码中可以看到 lua 脚本的执行被包裹在 pcall 函数调用中。

```
// 编译期
int luaCreateFunction(client *c, lua_State *lua,
char *funcname, robj *body) {
  ...
  if (lua_pcall(lua,0,0,0)) {
    addReplyErrorFormat(c,"Error running script
(new function): %s\n",
             lua_tostring(lua,-1));
    lua_pop(lua,1);
    return C_ERR;
  }
  ...
}

// 运行期
void evalGenericCommand(client *c, int evalsha) {
  ...
  err = lua_pcall(lua,0,1,-2);
  ...
}
```

Redis 在 lua 脚本中除了提供了 redis.call 函数外，同样也提供了 redis.pcall 函数。前者遇到错误向上抛出异常，后者会返回错误信息。使用时一定要注意 call 函数出错时会中断脚本的执行，为了保证脚本的原子性，要谨慎使用。

错误传递

redis.call 函数调用会产生错误，脚本遇到这种错误会返回怎样的信息呢？我们再看个例子

```
127.0.0.1:6379> hset foo x 1 y 2
(integer) 2
127.0.0.1:6379> eval 'return redis.call("incr",
"foo")' 0
(error) ERR Error running script (call to
f_8727c9c34a61783916ca488b366c475cb3a446cc):
@user_script:1: WRONGTYPE Operation against a key
holding the wrong kind of value
```

客户端输出的依然是一个通用的错误消息，而不是 incr 调用本应该返回的 WRONGTYPE 类型的错误消息。Redis 内部在处理 redis.call 遇到错误时是向上抛出异常，外围的用户看不见的 pcall 调用捕获到脚本异常时会向客户端回复通用的错误信息。如果我们将上面的 call 改成 pcall，结果就会不一样，它可以将内部指令返回的特定错误向上传递。

```
127.0.0.1:6379> eval 'return redis.pcall("incr",
"foo")' 0
(error) WRONGTYPE Operation against a key holding
the wrong kind of value
```

脚本死循环怎么办?

Redis 的指令执行是个单线程，这个单线程还要执行来自客户端的 lua 脚本。如果 lua 脚本中来一个死循环，是不是 Redis 就完蛋了？Redis 为了解决这个问题，它提供了 script kill 指令用于动态杀死一个执行时间超时的 lua 脚本。不过 script kill 的执行有一个重要的前提，那就是当前正在执行的脚本没有对 Redis 的内部数据状态进行修改，因为 Redis 不允许 script kill 破坏脚本执行的原子性。比如脚本内部使用了 redis.call("set", key, value) 修改了内部的数据，那么 script kill 执行时服务器会返回错误。下面我们来尝试以下 script kill 指令。

```
127.0.0.1:6379> eval 'while(true) do
print("hello") end' 0
```

eval 指令执行后，可以明显看出来 redis 卡死了，死活没有任何响应，如果去观察 Redis 服务器日志可以看到日志在疯狂输出 hello 字符串。这时候就必须重新开启一个 redis-cli 来执行 script kill 指令。

```
127.0.0.1:6379> script kill
OK
(2.58s)
```

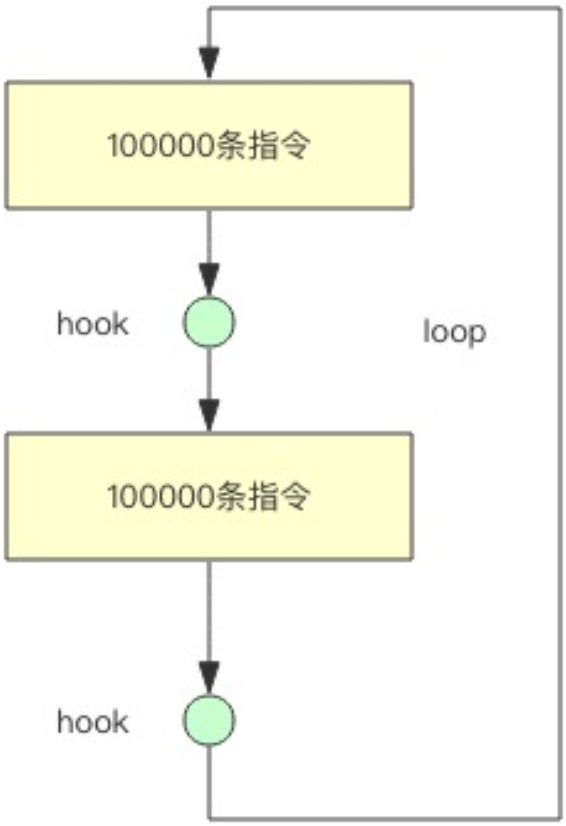
再回过头看 eval 指令的输出

```
127.0.0.1:6379> eval 'while(true) do
print("hello") end' 0
(error) ERR Error running script (call to
f_d395649372f578b1a0d3a1dc1b2389717cadf403):
@user_script:1: Script killed by user with SCRIPT
KILL...
(6.99s)
```

看到这里细心的同学会注意到两个疑点，第一个是 script kill 指令为什么执行了 2.58 秒，第二个是脚本都卡死了，Redis 哪里来的闲功夫接受 script kill 指令。如果你自己尝试了在第二个窗口执行 redis-cli 去连接服务器，你还会发现第三个疑点，redis-cli 建立连接有点慢，大约顿了有 1 秒左右。

Script Kill 的原理

下面我就要开始揭秘 kill 的原理了，lua 脚本引擎功能太强大了，它提供了各式各样的钩子函数，它允许在内部虚拟机执行指令时运行钩子代码。比如每执行 N 条指令执行一次某个钩子函数，Redis 正是使用了这个钩子函数。




```
void evalGenericCommand(client *c, int evalsha) {
  ...
  // lua引擎每执行10w条指令，执行一次钩子函数
luaMaskCountHook

lua_sethook(lua,luaMaskCountHook,LUA_MASKCOUNT,10
0000);
  ...
}
```

Redis 在钩子函数里会忙里偷闲去处理客户端的请求，并且只有在发现 lua 脚本执行超时之后才会去处理请求，这个超时时间默认是 5 秒。于是上面提出的三个疑点也就烟消云散了。

思考题

在延时队列小节，我们使用 zrangebyscore 和 zdel 两条指令来争抢延时队列中的任务，通过 zdel 的返回值来决定是哪个客户端抢到了任务，这意味着那些没有抢到任务的客户端会有这样一种感受——到了嘴边的肉(任务)最后还被别人抢走了，会很不爽。如果可以使用 lua 脚本来实现争抢逻辑，将 zrangebyscore 和 zdel 指令原子性执行就不会存在这种问题，读者可以尝试一下。

注：如果读者不熟悉 lua，建议先学习 lua 语言，lua 语言简单易学，但是也不是几分钟就可以学会的事，需要再来一本小册的内容。本小册专注 Redis，所以就不开大篇内容来细讲 lua 语言了，有需要的朋友可以搜索相关在线教程。