用户操作

[发私信] [加为好友]

订阅我的博客

1 位读者 POWERED BY FEEDSKY

订阅 订阅到鲜果 订阅到Google 订阅到抓虾

babyvspp的公告

文章分类

存档

2009年08月(2)
2009年06月(4)
2008年11月(1)
2008年10月(2)
2008年09月(4)
2008年07月(1)
2008年06月(1)
2008年04月(3)
2008年03月(1)
2008年02月(4)

DELPHI 线程池代码


```
unit uThreadPool;

{   aPool.AddRequest(TMyRequest.Create(RequestParam1, RequestParam2, ...)); }

interface
uses
  Windows,
  Classes;

// 是否记录日志
// {$DEFINE NOLOGS}

type
  TCriticalSection = class(TObject)
  protected
    FSection: TRTLCriticalSection;
  public
    constructor Create;
    destructor Destroy; override;
    // 进入临界区
    procedure Enter;
    // 离开临界区
    procedure Leave;
    // 尝试进入
    function TryEnter: Boolean;
  end;

type
  // 储存请求数据的基本类
```

2008年01月(4)

```
  TWorkItem = class(TObject)
  public
    // 是否有重复任务
    function IsTheSame(DataObj: TWorkItem): Boolean; virtual;
    // 如果 NOLOGS 被定义,则禁用。
    function TextForLog: string; virtual;
  end;

type
  TThreadsPool = class;

  //线程状态
  TThreadState = (tcsInitializing, tcsWaiting, tcsGetting, tcsProcessing,
    tcsProcessed, tcsTerminating, tcsCheckingDown);
  // 工作线程仅用于线程池内, 不要直接创建并调用它。
  TProcessorThread = class(TThread)
  private
    // 创建线程时临时的Event对象, 阻塞线程直到初始化完成
    hInitFinished: THandle;
    // 初始化出错信息
    sInitError: string;
    // 记录日志
    procedure WriteLog(const Str: string; Level: Integer = 0);
  protected
    // 线程临界区同步对像
    csProcessingDataObject: TCriticalSection;
    // 平均处理时间
    FAverageProcessing: Integer;
    // 等待请求的平均时间
    FAverageWaitingTime: Integer;
    // 本线程实例的运行状态
    FCurState: TThreadState;
    // 本线程实例所附属的线程池
    FPool: TThreadsPool;
    // 当前处理的数据对像。
    FProcessingDataObject: TWorkItem;
    // 线程停止 Event, TProcessorThread.Terminate 中开绿灯
```

```
    hThreadTerminated: THandle;
    uProcessingStart: DWORD;
    // 开始等待的时间, 通过 GetTickCount 取得。
    uWaitingStart: DWORD;
    // 计算平均工作时间
    function AverageProcessingTime: DWORD;
    // 计算平均等待时间
    function AverageWaitingTime: DWORD;
    procedure Execute; override;
    function IamCurrentlyProcess(DataObj: TWorkItem): Boolean;
    // 转换枚举类型的线程状态为字串类型
    function InfoText: string;
    // 线程是否长时间处理同一个请求？(已死掉？)
    function IsDead: Boolean;
    // 线程是否已完成当成任务
    function isFinished: Boolean;
    // 线程是否处于空闲状态
    function isIdle: Boolean;
    // 平均值校正计算。
    function NewAverage(OldAvg, NewVal: Integer): Integer;
  public
    Tag: Integer;
    constructor Create(APool: TThreadsPool);
    destructor Destroy; override;
    procedure Terminate;
  end;

  // 线程初始化时触发的事件
  TProcessorThreadInitializing = procedure(Sender: TThreadsPool; aThread:
    TProcessorThread) of object;
  // 线程结束时触发的事件
  TProcessorThreadFinalizing = procedure(Sender: TThreadsPool; aThread:
    TProcessorThread) of object;
  // 线程处理请求时触发的事件
  TProcessRequest = procedure(Sender: TThreadsPool; WorkItem: TWorkItem;
    aThread: TProcessorThread) of object;
  TEmptyKind = (
```

```
  ekQueueEmpty, //任务被取空后
  ekProcessingFinished // 最后一个任务处理完毕后
  );
// 任务队列空时触发的事件
TQueueEmpty = procedure(Sender: TThreadsPool; EmptyKind: TEmptyKind) of
  object;


TThreadsPool = class(TComponent)
private
  csQueueManagment: TCriticalSection;
  csThreadManagment: TCriticalSection;
  FProcessRequest: TProcessRequest;
  FQueue: TList;
  FQueueEmpty: TQueueEmpty;
  // 线程超时阀值
  FThreadDeadTimeout: DWORD;
  FThreadFinalizing: TProcessorThreadFinalizing;
  FThreadInitializing: TProcessorThreadInitializing;
  // 工作中的线程
  FThreads: TList;
  // 执行了 terminat 发送退出指令, 正在结束的线程.
  FThreadsKilling: TList;
  // 最少, 最大线程数
  FThreadsMax: Integer;
  // 最少, 最大线程数
  FThreadsMin: Integer;
  // 池平均等待时间
  function PoolAverageWaitingTime: Integer;
  procedure WriteLog(const Str: string; Level: Integer = 0);
protected
  FLastGetPoint: Integer;
  // Semaphore, 统计任务队列
  hSemRequestCount: THandle;
  // Waitable timer. 每30触发一次的时间量同步
  hTimCheckPoolDown: THandle;
  // 线程池停机(检查并清除空闲线程和死线程)
```

```
    procedure CheckPoolDown;
    // 清除死线程,并补充不足的工作线程
    procedure CheckThreadsForGrow;
    procedure DoProcessed;
    procedure DoProcessRequest(aDataObj: TWorkItem; aThread: TProcessorThread);
      virtual;
    procedure DoQueueEmpty(EmptyKind: TEmptyKind); virtual;
    procedure DoThreadFinalizing(aThread: TProcessorThread); virtual;
    // 执行事件
    procedure DoThreadInitializing(aThread: TProcessorThread); virtual;
    // 释放 FThreadsKilling 列表中的线程
    procedure FreeFinishedThreads;
    // 申请任务
    procedure GetRequest(out Request: TWorkItem);
    // 清除死线程
    procedure KillDeadThreads;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    // 就进行任务是否重复的检查, 检查发现重复就返回 False
    function AddRequest(aDataObject: TWorkItem; CheckForDoubles: Boolean =
      False): Boolean; overload;
    // 转换枚举类型的线程状态为字串类型
    function InfoText: string;
  published
    // 线程处理任务时触发的事件
    property OnProcessRequest: TProcessRequest read FProcessRequest write
      FProcessRequest;
    // 任务列表为空时解发的事件
    property OnQueueEmpty: TQueueEmpty read FQueueEmpty write FQueueEmpty;
    // 线程结束时触发的事件
    property OnThreadFinalizing: TProcessorThreadFinalizing read
      FThreadFinalizing write FThreadFinalizing;
    // 线程初始化时触发的事件
    property OnThreadInitializing: TProcessorThreadInitializing read
      FThreadInitializing write FThreadInitializing;
```

```
    // 线程超时值(毫秒), 如果处理超时,将视为死线程
    property ThreadDeadTimeout: DWORD read FThreadDeadTimeout write
      FThreadDeadTimeout default 0;
    // 最大线程数
    property ThreadsMax: Integer read FThreadsMax write FThreadsMax default 1;
    // 最小线程数
    property ThreadsMin: Integer read FThreadsMin write FThreadsMin default 0;
  end;

type
  //日志记志函数
  TLogWriteProc = procedure(
    const Str: string; //日志
    LogID: Integer = 0;
    Level: Integer = 0 //Level = 0 - 跟踪信息, 10 - 致命错误
    );

var
  WriteLog: TLogWriteProc; // 如果存在实例就写日志

implementation
uses
  SysUtils;

// 储存请求数据的基本类
{
********************************** TWorkItem ***********************************
}

function TWorkItem.IsTheSame(DataObj: TWorkItem): Boolean;
begin
  Result := False;
end; { TWorkItem.IsTheSame }

function TWorkItem.TextForLog: string;
begin
  Result := 'Request';
end; { TWorkItem.TextForLog }
```

```
{
********************************* TThreadsPool *********************************
}

constructor TThreadsPool.Create(AOwner: TComponent);
var
  DueTo: Int64;
begin
{$IFNDEF NOLOGS}
  WriteLog('创建线程池', 5);
{$ENDIF}
  inherited;
  csQueueManagment := TCriticalSection.Create;
  FQueue := TList.Create;
  csThreadManagment := TCriticalSection.Create;
  FThreads := TList.Create;
  FThreadsKilling := TList.Create;
  FThreadsMin := 0;
  FThreadsMax := 1;
  FThreadDeadTimeout := 0;
  FLastGetPoint := 0;
  //
  hSemRequestCount := CreateSemaphore(nil, 0, $7FFFFFFF, nil);

  DueTo := -1;
  //可等待的定时器(只用于Window NT4或更高)
  hTimCheckPoolDown := CreateWaitableTimer(nil, False, nil);

  if hTimCheckPoolDown = 0 then // Win9x不支持
    // In Win9x number of thread will be never decrised
    hTimCheckPoolDown := CreateEvent(nil, False, False, nil)
  else
    SetWaitableTimer(hTimCheckPoolDown, DueTo, 30000, nil, nil, False);
end; { TThreadsPool.Create }

destructor TThreadsPool.Destroy;
var
  n, i: Integer;
```

```
  Handles: array of THandle;
begin
{$IFNDEF NOLOGS}
  WriteLog('线程池销毁', 5);
{$ENDIF}
  csThreadManagment.Enter;

  SetLength(Handles, FThreads.Count);
  n := 0;
  for i := 0 to FThreads.Count - 1 do
    if FThreads[i] <> nil then
    begin
      Handles[n] := TProcessorThread(FThreads[i]).Handle;
      TProcessorThread(FThreads[i]).Terminate;
      Inc(n);
    end;
  WaitForMultipleObjects(n, @Handles[0], True, 30000);

  for i := 0 to FThreads.Count - 1 do
    TProcessorThread(FThreads[i]).Free;
  FThreads.Free;
  FThreadsKilling.Free;
  csThreadManagment.Free;

  csQueueManagment.Enter;
  for i := FQueue.Count - 1 downto 0 do
    TObject(FQueue[i]).Free;
  FQueue.Free;
  csQueueManagment.Free;

  CloseHandle(hSemRequestCount);
  CloseHandle(hTimCheckPoolDown);
  inherited;
end; { TThreadsPool.Destroy }

function TThreadsPool.AddRequest(aDataObject: TWorkItem; CheckForDoubles:
  Boolean = False): Boolean;
var
```

```
  i: Integer;
begin
{$IFNDEF NOLOGS}
  WriteLog('AddRequest(' + aDataObject.TextForLog + ')', 2);
{$ENDIF}
  Result := False;
  csQueueManagment.Enter;
  try
    // 如果 CheckForDoubles = TRUE
    // 则进行任务是否重复的检查
    if CheckForDoubles then
      for i := 0 to FQueue.Count - 1 do
        if (FQueue[i] <> nil)
          and aDataObject.IsTheSame(TWorkItem(FQueue[i])) then
          Exit; // 发现有相同的任务

    csThreadManagment.Enter;
    try
      // 清除死线程,并补充不足的工作线程
      CheckThreadsForGrow;

      // 如果 CheckForDoubles = TRUE
      // 则检查是否有相同的任务正在处理中
      if CheckForDoubles then
        for i := 0 to FThreads.Count - 1 do
          if TProcessorThread(FThreads[i]).IamCurrentlyProcess(aDataObject) then
            Exit; // 发现有相同的任务

    finally
      csThreadManagment.Leave;
    end;

    //将任务加入队列
    FQueue.Add(aDataObject);

    //释放一个同步信号量
    ReleaseSemaphore(hSemRequestCount, 1, nil);
{$IFNDEF NOLOGS}
```

```
    WriteLog('释放一个同步信号量)', 1);
{$ENDIF}
    Result := True;
  finally
    csQueueManagment.Leave;
  end;
{$IFNDEF NOLOGS}
  //调试信息
  WriteLog('增加一个任务(' + aDataObject.TextForLog + ')', 1);
{$ENDIF}
end; { TThreadsPool.AddRequest }

{
函 数 名:TThreadsPool.CheckPoolDown
功能描述:线程池停机(检查并清除空闲线程和死线程)
输入参数:无
返 回 值: 无
创建日期:2006.10.22 11:31
修改日期:2006.
作　　者:Kook
附加说明:
}

procedure TThreadsPool.CheckPoolDown;
var
  i: Integer;
begin
{$IFNDEF NOLOGS}
  WriteLog('TThreadsPool.CheckPoolDown', 1);
{$ENDIF}
  csThreadManagment.Enter;
  try
{$IFNDEF NOLOGS}
    WriteLog(InfoText, 2);
{$ENDIF}
    // 清除死线程
    KillDeadThreads;
```

```
    // 释放 FThreadsKilling 列表中的线程
    FreeFinishedThreads;

    // 如果线程空闲,就终止它
    for i := FThreads.Count - 1 downto FThreadsMin do
      if TProcessorThread(FThreads[i]).isIdle then
      begin
        //发出终止命令
        TProcessorThread(FThreads[i]).Terminate;
        //加入待清除队列
        FThreadsKilling.Add(FThreads[i]);
        //从工作队列中除名
        FThreads.Delete(i);
        //todo: ? ?
        Break;
      end;
  finally
    csThreadManagment.Leave;
  end;
end; { TThreadsPool.CheckPoolDown }

{
函 数 名:TThreadsPool.CheckThreadsForGrow
功能描述:清除死线程,并补充不足的工作线程
输入参数:无
返 回 值: 无
创建日期:2006.10.22 11:31
修改日期:2006.
作　　者:Kook
附加说明:
}

procedure TThreadsPool.CheckThreadsForGrow;
var
  AvgWait: Integer;
  i: Integer;
begin
```

```
  {
   New thread created if:
   新建线程的条件:
    1. 工作线程数小于最小线程数
    2. 工作线程数小于最大线程数 and 线程池平均等待时间 < 100ms(系统忙)
    3. 任务大于工作线程数的4倍
  }

  csThreadManagment.Enter;
  try
   KillDeadThreads;
   if FThreads.Count < FThreadsMin then
   begin
{$IFNDEF NOLOGS}
     WriteLog('工作线程数小于最小线程数', 4);
{$ENDIF}
     for i := FThreads.Count to FThreadsMin - 1 do
     try
       FThreads.Add(TProcessorThread.Create(Self));
     except
       on e: Exception do

         WriteLog(
           'TProcessorThread.Create raise: ' + e.ClassName + #13#10#9'Message: '
           + e.Message,
           9
           );
     end
   end
   else
     if FThreads.Count < FThreadsMax then
     begin
{$IFNDEF NOLOGS}
       WriteLog('工作线程数小于最大线程数 and 线程池平均等待时间 < 100ms', 3);
{$ENDIF}
       AvgWait := PoolAverageWaitingTime;
{$IFNDEF NOLOGS}
```

```
      WriteLog(Format(
       'FThreads.Count (%d)<FThreadsMax(%d), AvgWait=%d',
       [FThreads.Count, FThreadsMax, AvgWait]),
       4
       );
{$ENDIF}

      if AvgWait < 100 then
      try
       FThreads.Add(TProcessorThread.Create(Self));
      except
       on e: Exception do
        WriteLog(
         'TProcessorThread.Create raise: ' + e.ClassName +
         #13#10#9'Message: ' + e.Message,
         9
         );
      end;
    end;
  finally
   csThreadManagment.Leave;
  end;
end; { TThreadsPool.CheckThreadsForGrow }

procedure TThreadsPool.DoProcessed;
var
  i: Integer;
begin
  if (FLastGetPoint < FQueue.Count) then
    Exit;
  csThreadManagment.Enter;
  try
    for i := 0 to FThreads.Count - 1 do
      if TProcessorThread(FThreads[i]).FCurState in [tcsProcessing] then
        Exit;
  finally
    csThreadManagment.Leave;
```

```
  end;
  DoQueueEmpty(ekProcessingFinished);
end; { TThreadsPool.DoProcessed }

procedure TThreadsPool.DoProcessRequest(aDataObj: TWorkItem; aThread:
  TProcessorThread);
begin
  if Assigned(FProcessRequest) then
    FProcessRequest(Self, aDataObj, aThread);
end; { TThreadsPool.DoProcessRequest }

procedure TThreadsPool.DoQueueEmpty(EmptyKind: TEmptyKind);
begin
  if Assigned(FQueueEmpty) then
    FQueueEmpty(Self, EmptyKind);
end; { TThreadsPool.DoQueueEmpty }

procedure TThreadsPool.DoThreadFinalizing(aThread: TProcessorThread);
begin
  if Assigned(FThreadFinalizing) then
    FThreadFinalizing(Self, aThread);
end; { TThreadsPool.DoThreadFinalizing }

procedure TThreadsPool.DoThreadInitializing(aThread: TProcessorThread);
begin
  if Assigned(FThreadInitializing) then
    FThreadInitializing(Self, aThread);
end; { TThreadsPool.DoThreadInitializing }

{
函 数 名:TThreadsPool.FreeFinishedThreads
功能描述:释放 FThreadsKilling 列表中的线程
输入参数:无
返 回 值: 无
创建日期:2006.10.22 11:34
修改日期:2006.
作　　者:Kook
附加说明:
```

```
}

procedure TThreadsPool.FreeFinishedThreads;
var
  i: Integer;
begin
  if csThreadManagment.TryEnter then
  try
    for i := FThreadsKilling.Count - 1 downto 0 do
      if TProcessorThread(FThreadsKilling[i]).isFinished then
      begin
        TProcessorThread(FThreadsKilling[i]).Free;
        FThreadsKilling.Delete(i);
      end;
  finally
    csThreadManagment.Leave
  end;
end; { TThreadsPool.FreeFinishedThreads }

{
函 数 名:TThreadsPool.GetRequest
功能描述:申请任务
输入参数:out Request: TRequestDataObject
返 回 值: 无
创建日期:2006.10.22 11:34
修改日期:2006.
作　　者:Kook
附加说明:
}

procedure TThreadsPool.GetRequest(out Request: TWorkItem);
begin
{$IFNDEF NOLOGS}
  WriteLog('申请任务', 2);
{$ENDIF}
  csQueueManagment.Enter;
  try
```

```
    //跳过空的队列元素
    while (FLastGetPoint < FQueue.Count) and (FQueue[FLastGetPoint] = nil) do
      Inc(FLastGetPoint);

    Assert(FLastGetPoint < FQueue.Count);
    //压缩队列,清除空元素
    if (FQueue.Count > 127) and (FLastGetPoint >= (3 * FQueue.Count) div 4) then
    begin
{$IFNDEF NOLOGS}
      WriteLog('FQueue.Pack', 1);
{$ENDIF}
      FQueue.Pack;
      FLastGetPoint := 0;
    end;

    Request := TWorkItem(FQueue[FLastGetPoint]);
    FQueue[FLastGetPoint] := nil;
    inc(FLastGetPoint);
    if (FLastGetPoint = FQueue.Count) then //如果队列中无任务
    begin

      DoQueueEmpty(ekQueueEmpty);
      FQueue.Clear;
      FLastGetPoint := 0;
    end;
  finally
    csQueueManagment.Leave;
  end;
end; { TThreadsPool.GetRequest }

function TThreadsPool.InfoText: string;
begin
  Result := '';
  //end;
  //{$ELSE}
  //var
  //  i: Integer;
  //begin
```

```
  //  csQueueManagment.Enter;
  //  csThreadManagment.Enter;
  //  try
  //    if (FThreads.Count = 0) and (FThreadsKilling.Count = 1) and
  //      TProcessorThread(FThreadsKilling[0]).isFinished then
  //      FreeFinishedThreads;
  //
  //    Result := Format(
  //      'Pool thread: Min=%d, Max=%d, WorkingThreadsCount=%d, TerminatedThreadCount=%d
, QueueLength=%d'#13#10,
  //      [ThreadsMin, ThreadsMax, FThreads.Count, FThreadsKilling.Count,
  //      FQueue.Count]
  //        );
  //    if FThreads.Count > 0 then
  //      Result := Result + 'Working threads:'#13#10;
  //    for i := 0 to FThreads.Count - 1 do
  //      Result := Result + TProcessorThread(FThreads[i]).InfoText + #13#10;
  //    if FThreadsKilling.Count > 0 then
  //      Result := Result + 'Terminated threads:'#13#10;
  //    for i := 0 to FThreadsKilling.Count - 1 do
  //      Result := Result + TProcessorThread(FThreadsKilling[i]).InfoText + #13#10;
  //  finally
  //    csThreadManagment.Leave;
  //    csQueueManagment.Leave;
  //  end;
  //end;
  //{$ENDIF}
end; { TThreadsPool.InfoText }

{
函 数 名:TThreadsPool.KillDeadThreads
功能描述:清除死线程
输入参数:无
返 回 值: 无
创建日期:2006.10.22 11:32
修改日期:2006.
作　　者:Kook
```

附加说明:
}

```
procedure TThreadsPool.KillDeadThreads;
var
  i: Integer;
begin
  // Check for dead threads
  if csThreadManagment.TryEnter then
  try
    for i := 0 to FThreads.Count - 1 do
      if TProcessorThread(FThreads[i]).IsDead then
      begin
        // Dead thread moverd to other list.
        // New thread created to replace dead one
        TProcessorThread(FThreads[i]).Terminate;
        FThreadsKilling.Add(FThreads[i]);
        try
          FThreads[i] := TProcessorThread.Create(Self);
        except
          on e: Exception do
          begin
            FThreads[i] := nil;
{$IFNDEF NOLOGS}
            WriteLog(
              'TProcessorThread.Create raise: ' + e.ClassName +
              #13#10#9'Message: ' + e.Message,
              9
              );
{$ENDIF}
          end;
        end;
      end;
  finally
    csThreadManagment.Leave
  end;
```

```
end; { TThreadsPool.KillDeadThreads }

function TThreadsPool.PoolAverageWaitingTime: Integer;
var
  i: Integer;
begin
  Result := 0;
  if FThreads.Count > 0 then
  begin
    for i := 0 to FThreads.Count - 1 do
      Inc(result, TProcessorThread(FThreads[i]).AverageWaitingTime);
    Result := Result div FThreads.Count
  end
  else
    Result := 1;
end; { TThreadsPool.PoolAverageWaitingTime }

procedure TThreadsPool.WriteLog(const Str: string; Level: Integer = 0);
begin
{$IFNDEF NOLOGS}
  uThreadPool.WriteLog(Str, 0, Level);
{$ENDIF}
end; { TThreadsPool.WriteLog }

// 工作线程仅用于线程池内, 不要直接创建并调用它。
{
****************************** TProcessorThread ******************************
}

constructor TProcessorThread.Create(APool: TThreadsPool);
begin
  WriteLog('创建工作线程', 5);
  inherited Create(True);
  FPool := aPool;

  FAverageWaitingTime := 1000;
  FAverageProcessing := 3000;
```

```
  sInitError := '';
  {
  各参数的意义如下:

   参数一:填上 nil 即可。
   参数二:是否采用手动调整灯号。
   参数三:灯号的起始状态,False 表示红灯。
   参数四:Event 名称, 对象名称相同的话,会指向同一个对象,所以想要有两个Event对象,便要有两个不同的名称(这名称以字符串来存.为NIL的话系统每次会自己创建一个不同的名字,就是被次创建的都是新的EVENT)。
   传回值:Event handle。
  }
  hInitFinished := CreateEvent(nil, True, False, nil);
  hThreadTerminated := CreateEvent(nil, True, False, nil);
  csProcessingDataObject := TCriticalSection.Create;
  try
    WriteLog('TProcessorThread.Create::Resume', 3);
    Resume;
    //阻塞, 等待初始化完成
    WaitForSingleObject(hInitFinished, INFINITE);
    if sInitError <> '' then
      raise Exception.Create(sInitError);
  finally
    CloseHandle(hInitFinished);
  end;
  WriteLog('TProcessorThread.Create::Finished', 3);
end; { TProcessorThread.Create }

destructor TProcessorThread.Destroy;
begin
  WriteLog('工作线程销毁', 5);
  CloseHandle(hThreadTerminated);
  csProcessingDataObject.Free;
  inherited;
end; { TProcessorThread.Destroy }

function TProcessorThread.AverageProcessingTime: DWORD;
begin
```

```
  if (FCurState in [tcsProcessing]) then
    Result := NewAverage(FAverageProcessing, GetTickCount - uProcessingStart)
  else
    Result := FAverageProcessing
end; { TProcessorThread.AverageProcessingTime }

function TProcessorThread.AverageWaitingTime: DWORD;
begin
  if (FCurState in [tcsWaiting, tcsCheckingDown]) then
    Result := NewAverage(FAverageWaitingTime, GetTickCount - uWaitingStart)
  else
    Result := FAverageWaitingTime
end; { TProcessorThread.AverageWaitingTime }

procedure TProcessorThread.Execute;

type
  THandleID = (hidTerminateThread, hidRequest, hidCheckPoolDown);
var
  WaitedTime: Integer;
  Handles: array[THandleID] of THandle;

begin
  WriteLog('工作线程进常运行', 3);
  //当前状态:初始化
  FCurState := tcsInitializing;
  try
    //执行外部事件
    FPool.DoThreadInitializing(Self);
  except
    on e: Exception do
      sInitError := e.Message;
  end;

  //初始化完成,初始化Event绿灯
  SetEvent(hInitFinished);

  WriteLog('TProcessorThread.Execute::Initialized', 3);
```

```
//引用线程池的同步 Event
Handles[hidTerminateThread] := hThreadTerminated;
Handles[hidRequest] := FPool.hSemRequestCount;
Handles[hidCheckPoolDown] := FPool.hTimCheckPoolDown;

//时间戳,
//todo: 好像在线程中用 GetTickCount; 会不正常
uWaitingStart := GetTickCount;
//任务置空
FProcessingDataObject := nil;

//大巡环
while not terminated do
begin
  //当前状态:等待
  FCurState := tcsWaiting;
  //阻塞线程,使线程休眠
  case WaitForMultipleObjects(Length(Handles), @Handles, False, INFINITE) -
    WAIT_OBJECT_0 of

    WAIT_OBJECT_0 + ord(hidTerminateThread):
      begin
        WriteLog('TProcessorThread.Execute:: Terminate event signaled ', 5);
        //当前状态:正在终止线程
        FCurState := tcsTerminating;
        //退出大巡环(结束线程)
        Break;
      end;

    WAIT_OBJECT_0 + ord(hidRequest):
      begin
        WriteLog('TProcessorThread.Execute:: Request semaphore signaled ', 3);
        //等待的时间
        WaitedTime := GetTickCount - uWaitingStart;
        //重新计算平均等待时间
        FAverageWaitingTime := NewAverage(FAverageWaitingTime, WaitedTime);
        //当前状态:申请任务
        FCurState := tcsGetting;
```

```
        //如果等待时间过短,则检查工作线程是否足够
        if WaitedTime < 5 then
          FPool.CheckThreadsForGrow;
        //从线程池的任务队列中得到任务
        FPool.GetRequest(FProcessingDataObject);
        //开始处理的时间戳
        uProcessingStart := GetTickCount;
        //当前状态:执行任务
        FCurState := tcsProcessing;
        try
{$IFNDEF NOLOGS}
          WriteLog('Processing: ' + FProcessingDataObject.TextForLog, 2);
{$ENDIF}
          //执行任务
          FPool.DoProcessRequest(FProcessingDataObject, Self);
        except
          on e: Exception do
            WriteLog(
              'OnProcessRequest for ' + FProcessingDataObject.TextForLog +
              #13#10'raise Exception: ' + e.Message,
              8
              );
        end;

        //释放任务对象
        csProcessingDataObject.Enter;
        try
          FProcessingDataObject.Free;
          FProcessingDataObject := nil;
        finally
          csProcessingDataObject.Leave;
        end;
        //重新计算
        FAverageProcessing := NewAverage(FAverageProcessing, GetTickCount -
          uProcessingStart);
        //当前状态:执行任务完毕
```

```
          FCurState := tcsProcessed;
          //执行线程外事件
          FPool.DoProcessed;

          uWaitingStart := GetTickCount;
        end;
      WAIT_OBJECT_0 + ord(hidCheckPoolDown):
        begin
          // !!! Never called under Win9x
          WriteLog('TProcessorThread.Execute:: CheckPoolDown timer signaled ',
            4);
          //当前状态:线程池停机(检查并清除空闲线程和死线程)
          FCurState := tcsCheckingDown;
          FPool.CheckPoolDown;
        end;
    end;
  end;
  FCurState := tcsTerminating;

  FPool.DoThreadFinalizing(Self);
end; { TProcessorThread.Execute }

function TProcessorThread.IamCurrentlyProcess(DataObj: TWorkItem): Boolean;
begin
  csProcessingDataObject.Enter;
  try
    Result := (FProcessingDataObject <> nil) and
      DataObj.IsTheSame(FProcessingDataObject);
  finally
    csProcessingDataObject.Leave;
  end;
end; { TProcessorThread.IamCurrentlyProcess }

function TProcessorThread.InfoText: string;

const
  ThreadStateNames: array[TThreadState] of string =
  (
```

```
    'tcsInitializing',
    'tcsWaiting',
    'tcsGetting',
    'tcsProcessing',
    'tcsProcessed',
    'tcsTerminating',
    'tcsCheckingDown'
    );

begin
{$IFNDEF NOLOGS}
  Result := Format(
    '%5d: %15s, AverageWaitingTime=%6d, AverageProcessingTime=%6d',
    [ThreadID, ThreadStateNames[FCurState], AverageWaitingTime,
    AverageProcessingTime]
      );
  case FCurState of
    tcsWaiting:
      Result := Result + ', WaitingTime=' + IntToStr(GetTickCount -
        uWaitingStart);
    tcsProcessing:
      Result := Result + ', ProcessingTime=' + IntToStr(GetTickCount -
        uProcessingStart);
  end;

  csProcessingDataObject.Enter;
  try
    if FProcessingDataObject <> nil then
      Result := Result + ' ' + FProcessingDataObject.TextForLog;
  finally
    csProcessingDataObject.Leave;
  end;
{$ENDIF}
end; { TProcessorThread.InfoText }

function TProcessorThread.IsDead: Boolean;
begin
```

```
  Result :=
    Terminated or
    (FPool.ThreadDeadTimeout > 0) and (FCurState = tcsProcessing) and
    (GetTickCount - uProcessingStart > FPool.ThreadDeadTimeout);
  if Result then
    WriteLog('Thread dead', 5);
end; { TProcessorThread.IsDead }

function TProcessorThread.isFinished: Boolean;
begin
  Result := WaitForSingleObject(Handle, 0) = WAIT_OBJECT_0;
end; { TProcessorThread.isFinished }

function TProcessorThread.isIdle: Boolean;
begin
  // 如果线程状态是 tcsWaiting, tcsCheckingDown
  // 并且 空间时间 > 100ms,
  // 并且 平均等候任务时间大于平均工作时间的 50%
  // 则视为空闲。
  Result :=
    (FCurState in [tcsWaiting, tcsCheckingDown]) and
    (AverageWaitingTime > 100) and
    (AverageWaitingTime * 2 > AverageProcessingTime);
end; { TProcessorThread.isIdle }

function TProcessorThread.NewAverage(OldAvg, NewVal: Integer): Integer;
begin
  Result := (OldAvg * 2 + NewVal) div 3;
end; { TProcessorThread.NewAverage }

procedure TProcessorThread.Terminate;
begin
  WriteLog('TProcessorThread.Terminate', 5);
  inherited Terminate;
  SetEvent(hThreadTerminated);
end; { TProcessorThread.Terminate }

procedure TProcessorThread.WriteLog(const Str: string; Level: Integer = 0);
```

```
begin
{$IFNDEF NOLOGS}
  uThreadPool.WriteLog(Str, ThreadID, Level);
{$ENDIF}
end; { TProcessorThread.WriteLog }

{
****************************** TCriticalSection ******************************
}

constructor TCriticalSection.Create;
begin
  InitializeCriticalSection(FSection);
end; { TCriticalSection.Create }

destructor TCriticalSection.Destroy;
begin
  DeleteCriticalSection(FSection);
end; { TCriticalSection.Destroy }

procedure TCriticalSection.Enter;
begin
  EnterCriticalSection(FSection);
end; { TCriticalSection.Enter }

procedure TCriticalSection.Leave;
begin
  LeaveCriticalSection(FSection);
end; { TCriticalSection.Leave }

function TCriticalSection.TryEnter: Boolean;
begin
  Result := TryEnterCriticalSection(FSection);
end; { TCriticalSection.TryEnter }

procedure NoLogs(const Str: string; LogID: Integer = 0; Level: Integer = 0);
begin
end;
```

```
initialization
  WriteLog := NoLogs;
end.
```

发表于 @ 2008年01月01日 18:46:00 | 评论(3) | 举报 | 收藏

新一篇:浅谈DELPHI指针

□□□Thursday, May 14, 2009 01:49:41 □□

□□□□

l12555712 □□□Wednesday, June 17, 2009 06:37:48 □□

□□□□□□□□□□□□□□□□□□□□DealwithCommRecvData□□□□□□□□...□□□□□□ procedure DealwithCommRecvData; begin // end;

pascal_4381 □□□Friday, July 31, 2009 18:19:40 □□

□□□□□□□□□□□□□□□□

给babyvspp的留言 □□□□□□□□□□□□□□□ □□ □□