# BOOST

Boost库版本：boost\_1\_63\_0

IDE：vs2013 版本12.0.40629.00 Update5

## 文件系统

### boost::filesystem::path

#include<boost/filesystem.hpp>

##### 从路径中截取需要的信息

boost::filesystem::path p("D:\\g++\\mingw\\info.text");

std::cout << p.string() << std::endl; D:\g++\mingw\info.text

std::cout << p.root\_name() << std::endl; D:

std::cout << p.root\_directory() << std::endl; \

std::cout << p.root\_path() << std::endl; D:\

std::cout << p.relative\_path() << std::endl; g++\mingw\info.text

std::cout << p.parent\_path() << std::endl; D:\g++\mingw

std::cout << p.filename() << std::endl; info.text

##### 检查路径中是否含有特定字符串

has\_root\_name(),

has\_root\_directory(),

has\_root\_path(),

has\_relative\_path(),

has\_parent\_path()

has\_filename()

各个方法都是返回一个 bool 类型的值。

##### 拆分文件名

boost::filesystem::path p("photo.jpg");

std::cout << p.stem() << std::endl; photo

std::cout << p.extension() << std::endl; .jpg

##### 对path迭代

boost::filesystem::path p("C:\\Windows\\System\\a.txt");

for (boost::filesystem::path::iterator it = p.begin(); it != p.end(); ++it)

std::cout << \*it << std::endl;

输出：

C:

/

Windows

System

a.txt

##### 修改路径

末尾添加

boost::filesystem::path p("C:\\");

p /= "Windows\\System";

std::cout << p.string() << std::endl;

输出：

C:\Windows\System

末尾删除

boost::filesystem::path p("C:\\Windows\\System\\a.txt");

p.remove\_filename();

std::cout << p.string() << std::endl;

输出：

C:\Windows\System

删除后缀

boost::filesystem::path p("C:\\Windows\\System\\a.txt");

p.replace\_extension();

std::cout << p.string() << std::endl;

输出：

C:\Windows\System\a

### 文件与目录

boost::filesystem::path 的各个方法内部其实只是对字符串进行处理。 它们可以用来访问一个路径的各个组件、相互添加路径等等。

为了处理硬盘上的物理文件和目录，提供了几个独立的函数。 这些函数需要一个或多个 boost::filesystem::path 类型的参数，并且在其内部会调用操作系统功能来处理这些文件或目录。

在介绍各个函数之前，很重要的一点是要弄明白出现错误时会发生什么。 所有要在内部访问操作系统功能的函数都有可能失败。 在失败的情况下，将抛出一个类型为 boost::filesystem::filesystem\_error 的异常。 这个类是派生自 boost::system::system\_error 的，因此适用于 Boost.System 框架。

除了继承自父类 boost::system::system\_error 的 what() 和 code() 方法以外，还有另外两个方法：path1() 和 path2()。 它们均返回一个类型为 boost::filesystem::path 的对象，因此在发生错误时可以很容易地确定路径信息 - 即使是对那些需要两个 boost::filesystem::path 参数的函数。

多数函数存在两个变体：在失败时，一个会抛出类型为 boost::filesystem::filesystem\_error 的异常，而另一个则返回类型为 boost::system::error\_code 的对象。 对于后者，需要对返回值进行明确的检查以确定是否出错。

##### 查询一个文件或目录的状态

boost::filesystem::path p("D:\\boost");

try

{

boost::filesystem::file\_status s = boost::filesystem::status(p);

std::cout << boost::filesystem::is\_directory(s) << std::endl;

}

catch (boost::filesystem::filesystem\_error &e)

{

std::cerr << e.what() << std::endl;

}

boost::filesystem::status() 返回一个 boost::filesystem::file\_status 类型的对象，该对象可以被传递给其它辅助函数来评估

boost::filesystem::is\_directory() 目录

boost::filesystem::is\_regular\_file() 普通文件

boost::filesystem::exists() 文件是否存在 快捷键认为不存在

##### 查询文件大小

boost::filesystem::path p("E:\\vc13\\文件传送1.1.rar");

try

{

std::cout << boost::filesystem::file\_size(p) << std::endl;

}

catch (boost::filesystem::filesystem\_error &e)

{

std::cerr << e.what() << std::endl;

}

实际大小非占用空间

##### 获取最后被修改时间

boost::filesystem::path p("C:\\Windows\\win.ini");

try

{

std::time\_t t = boost::filesystem::last\_write\_time(p);

std::cout << std::ctime(&t) << std::endl;

}

catch (boost::filesystem::filesystem\_error &e)

{

std::cerr << e.what() << std::endl;

}

std::ctime 不安全了 修改工程属性 c++命令行里面添加 /D \_CRT\_SECURE\_NO\_WARNINGS

##### 修改文件夹名称

boost::filesystem::path p("E:\\vc13\\aaaaaaa2");

boost::filesystem::rename(p,"E:\\vc13\\aaaaaaa3");

##### 删除文件夹

boost::filesystem::remove("E:\\vc13\\ppp"); 只能删除空文件夹

boost::filesystem::remove\_all("E:\\vc13\\ppp"); 删除文件夹及其包含内容

##### 复制文件

boost::filesystem::copy\_file("E:\\vc13\\a\\dd.txt","E:\\vc13\\b\\c.txt");

##### 创建文件夹

boost::filesystem::create\_directory("E:\\vc13\\a");

一次只能创建一个

boost::filesystem::create\_directories("E:\\vc13\\b\\c");

多层创建

##### 文件搜索

当前路径

boost::filesystem::path dir2("c:\\Windows\\System32");

boost::filesystem::directory\_iterator end;

for (boost::filesystem::directory\_iterator pos(dir2); pos != end; pos++)

{

std::cout << \*pos << std::endl;

}

深度遍历

typedef recursive\_directory\_iterator rd\_iterator;

path dir2("E:\\vc13\\Project1");

rd\_iterator end;

for (rd\_iterator pos(dir2); pos != end; pos++)

{

//如果深度大于4层，则不再继续深入

//if (is\_directory(\*pos) && pos.level() > 4)

//{

// pos.no\_push();

//}

////如果该目录下有nofind.txt文件，则跳出该目录

//if (\*pos == "nofind.txt")

//{

// pos.pop();

//}

std::cout << \*pos << std::endl;

}

## 时间与日期

CLOCKS\_PER\_SEC 精度

Win 1000

Linux 10000

### 计时器

timer timee; //初始化完成的瞬间开始计时

timee.elapsed\_max //支持记录的最大事件

timee.elapsed\_min//精度 以秒为单位

timee.elapsed //过了多久

stringstream ss;

{

progress\_timer t(ss); //析构是输出存在时间

}

Cout<<ss.str()<<endl;

## 多线程

#include <boost/thread.hpp>

### 创建线程

#include <boost/thread.hpp>

#include <iostream>

void wait(int seconds)

{

boost::this\_thread::sleep(boost::posix\_time::seconds(seconds));

}

void thread()

{

for (int i = 0; i < 5; ++i)

{

wait(1);

std::cout << i << std::endl;

}

}

int main()

{

boost::thread t(thread);

t.join();

}

### 分离线程

boost::thread::detach();

### 中断线程

Boost::thread:: interrupt();

只有当线程有断点的时候可以中断线程，如果没有该函数没有任何作用

在一个线程对象上调用 interrupt() 会中断相应的线程。 在这方面，中断意味着一个类型为 boost::thread\_interrupted 的异常，它会在这个线程中抛出

每当一个线程中断点，它就会检查 interrupt() 是否被调用过。 只有被调用过了， boost::thread\_interrupted 异常才会相应地抛出。

#include <boost/thread.hpp>

#include <iostream>

void wait(int seconds)

{

boost::this\_thread::sleep(boost::posix\_time::seconds(seconds));

}

void thread()

{

try

{

for (int i = 0; i < 5; ++i)

{

wait(1);

std::cout << i << std::endl;

}

}

catch (boost::thread\_interrupted&)

{

}

}

int main()

{

boost::thread t(thread);

wait(3);

t.interrupt();

t.join();

}

### 对当前线程的操作

使用 boost::this\_thread命名空间，能提供独立的函数应用于当前线程，比如前面出现的 sleep() 。 另一个是 get\_id()：它会返回一个当前线程的ID号。 它也是由 boost::thread 提供的。

boost::thread 类提供了一个静态方法 hardware\_concurrency() ，它能够返回基于CPU数目或者CPU内核数目的刻在同时在物理机器上运行的线程数。 在常用的双核机器上调用这个方法，返回值为2。 这样的话就可以确定在一个多核程序可以同时运行的理论最大线程数。

### 同步

#### boost::mutex

boost::mutex mutex;

void thread()

{

for (int i = 0; i < 5; ++i)

{

wait(1);

mutex.lock();

std::cout << "Thread " << boost::this\_thread::get\_id() << ": " << i << std::endl;

mutex.unlock();

}

}

#### boost::lock\_guard

boost::lock\_guard 在其内部构造和析构函数分别自动调用 lock() 和 unlock()

boost::mutex mutex;

void thread()

{

for (int i = 0; i < 5; ++i)

{

wait(1);

boost::lock\_guard<boost::mutex> lock(mutex);

std::cout << "Thread " << boost::this\_thread::get\_id() << ": " << i << std::endl;

}

}

### Assert

#include<boost/assert.hpp>

BOOST\_assert(x!=0 && “除数为0”);

## 网络通信

Boost::asio 程序必须的对象

### Ip地址

Ip::address addr;

addr = addr.from\_string(“127.0.0.1”);

### Socket

Ip::tcp::socket sock(ios);

Sock.connect();

Sock.local\_endpoint().address; //本地端点信息

Sock.Remote\_endpoint().address(); //对方端点信息

Sock.read\_some(); //阻塞读取信息

Sock.write\_some(); //阻塞写信息

Buff();可以包装需要发送的数据

Ip::tcp::endpoint ep(ip::address::from\_string(“127.0.0.1”),6688); //创建链接端点

### Acceptor

Io\_service ios;

Ip::tcp::acceptor acceptor(ios,ip::tcp::endpoint(ip::tcp::v4()),6688);

Acceptor.accept(sock); //阻塞等待

# Windows多线程

## 线程（进程）之间的制约关系？

关键段：一次只能有一个线程进入关键段

互斥量：处于触发状态不能用 处于未触发可以用

事件： 处于触发状态可以用 处于未触发状态不可以用

当线程并发执行时，由于资源共享和线程协作，使用线程之间会存在以下两种制约关系。

（1）．间接相互制约。一个系统中的多个线程必然要共享某种系统资源，如共享CPU，共享I/O设备，所谓间接相互制约即源于这种资源共享，打印机就是最好的例子，线程A在使用打印机时，其它线程都要等待。

（2）．直接相互制约。这种制约主要是因为线程之间的合作，如有线程A将计算结果提供给线程B作进一步处理，那么线程B在线程A将数据送达之前都将处于阻塞状态。

间接相互制约可以称为互斥，直接相互制约可以称为同步

## 临界资源和临界区

在一段时间内只允许一个线程访问的资源就称为临界资源或独占资源，计算机中大多数物理设备，进程中的共享变量等待都是临界资源，它们要求被互斥的访问。每个进程中访问临界资源的代码称为临界区

## 创建线程

### CreateThread

创建一个有缺陷的线程

#include <stdio.h>

#include <windows.h>

DWORD WINAPI ThreadFun(LPVOID pM)

{

printf("子线程的线程ID号为：%d\n子线程输出Hello World\n", GetCurrentThreadId());

return 0;

}

int main()

{

HANDLE handle = CreateThread(NULL, 0, ThreadFun, NULL, 0, NULL);

WaitForSingleObject(handle, INFINITE);

return 0;

}

函数功能：创建线程

函数原型：

HANDLE WINAPI CreateThread(

  LPSECURITY\_ATTRIBUTES lpThreadAttributes,

  SIZE\_T dwStackSize,

  LPTHREAD\_START\_ROUTINE lpStartAddress,

  LPVOID lpParameter,

  DWORD dwCreationFlags,

  LPDWORD lpThreadId

);

函数说明：

第一个参数表示线程内核对象的安全属性，一般传入NULL表示使用默认设置。

第二个参数表示线程栈空间大小。传入0表示使用默认大小（1MB）。

第三个参数表示新线程所执行的线程函数地址，多个线程可以使用同一个函数地址。

第四个参数是传给线程函数的参数。

第五个参数指定额外的标志来控制线程的创建，为0表示线程创建之后立即就可以进行调度，如果为CREATE\_SUSPENDED则表示线程创建后暂停运行，这样它就无法调度，直到调用ResumeThread()。

第六个参数将返回线程的ID号，传入NULL表示不需要返回该线程ID号。

函数返回值：

成功返回新线程的句柄，失败返回NULL。

### WaitForSingleObject

函数功能：等待函数 – 使线程进入等待状态，直到指定的内核对象被触发。

函数原形：

DWORD WINAPI WaitForSingleObject(

  HANDLE hHandle,

  DWORD dwMilliseconds

);

函数说明：

第一个参数为要等待的内核对象。

第二个参数为最长等待的时间，以毫秒为单位，如传入5000就表示5秒，传入0就立即返回，传入INFINITE表示无限等待。

因为线程的句柄在线程运行时是未触发的，线程结束运行，句柄处于触发状态。所以可以用WaitForSingleObject()来等待一个线程结束运行。

函数返回值：

在指定的时间内对象被触发，函数返回WAIT\_OBJECT\_0。超过最长等待时间对象仍未被触发返回WAIT\_TIMEOUT。传入参数有错误将返回WAIT\_FAILED

### \_beginthreadex()

每个线程都将拥有自己专用的一块内存区域来供标准C运行库中所有有需要的函数使用

\_beginthreadex()函数在创建新线程时会分配并初始化一个\_tiddata块。这个\_tiddata块自然是用来存放一些需要线程独享的数据。事实上新线程运行时会首先将\_tiddata块与自己进一步关联起来。然后新线程调用标准C运行库函数如strtok()时就会先取得\_tiddata块的地址再将需要保护的数据存入\_tiddata块中。这样每个线程就只会访问和修改自己的数据而不会去篡改其它线程的数据了

1. #include <stdio.h>
2. #include <process.h>
3. #include <windows.h>
4. unsigned **int** \_\_stdcall ThreadFun(**PVOID** pM)
5. {
6. printf("线程ID号为%4d的子线程说：Hello World\n", GetCurrentThreadId());
7. **return** 0;
8. }
9. //主函数，所谓主函数其实就是主线程执行的函数。
10. **int** main()
11. {
12. **const** **int** THREAD\_NUM = 5;
13. **HANDLE** handle[THREAD\_NUM];
14. **for** (**int** i = 0; i < THREAD\_NUM; i++)
15. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ThreadFun, NULL, 0, NULL);
16. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
17. **return** 0;
18. }

## 原子操作

### 增减

LONG \_\_cdecl InterlockedIncrement(LONG  volatile\* Addend);//+

LONG \_\_cdecl InterlockedDecrement(LONG  volatile\* Addend);//-

返回变量执行增减操作之后的值。

LONG \_\_cdec  InterlockedExchangeAdd(LONG  volatile\* Addend, LONG Value);

返回运算后的值，注意！加个负数就是减。

### 赋值

LONG \_\_cdecl InterlockedExchange(LONG volatile\* Target, LONGValue);

Value就是新值，函数会返回原先的值。

## 关键段

#include <process.h>

typedef struct \_RTL\_CRITICAL\_SECTION {

PRTL\_CRITICAL\_SECTION\_DEBUG DebugInfo;

LONG LockCount;

LONG RecursionCount;

HANDLE OwningThread; // from the thread's ClientId->UniqueThread

HANDLE LockSemaphore;

DWORD SpinCount;

} RTL\_CRITICAL\_SECTION, \*PRTL\_CRITICAL\_SECTION;

各个参数的解释如下：

第一个参数：PRTL\_CRITICAL\_SECTION\_DEBUGDebugInfo;

调试用的。

第二个参数：LONGLockCount;

初始化为-1，n表示有n个线程在等待。

第三个参数：LONGRecursionCount;

表示该关键段的拥有线程对此资源获得关键段次数，初为0。

第四个参数：HANDLEOwningThread;

即拥有该关键段的线程句柄，微软对其注释为——from the thread's ClientId->UniqueThread

第五个参数：HANDLELockSemaphore;

实际上是一个自复位事件。

第六个参数：DWORDSpinCount;

旋转锁的设置，单CPU下忽略

由这个结构可以知道关键段会记录拥有该关键段的线程句柄即关键段是有“线程所有权”概念的。事实上它会用第四个参数OwningThread来记录获准进入关键区域的线程句柄，如果这个线程再次进入，EnterCriticalSection()会更新第三个参数RecursionCount以记录该线程进入的次数并立即返回让该线程进入。其它线程调用EnterCriticalSection()则会被切换到等待状态，一旦拥有线程所有权的线程调用LeaveCriticalSection()使其进入的次数为0时，系统会自动更新关键段并将等待中的线程换回可调度状态。

因此可以将关键段比作旅馆的房卡，调用EnterCriticalSection()即申请房卡，得到房卡后自己当然是可以多次进出房间的，在你调用LeaveCriticalSection()交出房卡之前，别人自然是无法进入该房间。

回到这个经典线程同步问题上，主线程正是由于拥有“线程所有权”即房卡，所以它可以重复进入关键代码区域从而导致子线程在接收参数之前主线程就已经修改了这个参数。所以关键段可以用于线程间的互斥，但不可以用于同步。

总结：一旦一个线程进入了关键段它可以多次进入关键段直到有线程LeaveCriticalSection()

它才会重新去排队

### 初始化

函数原型：

void InitializeCriticalSection (LPCRITICAL\_SECTION lpCriticalSection);

函数说明：定义关键段变量后必须先初始化。

### 销毁

函数原型：

void DeleteCriticalSection(LPCRITICAL\_SECTION lpCriticalSection);

函数说明：用完之后记得销毁。

### 进入关键区域

函数原型：

void EnterCriticalSection(LPCRITICAL\_SECTION lpCriticalSection);

函数说明：系统保证各线程互斥的进入关键区域。

### 离开关关键区域

函数原型：

void LeaveCriticalSection(LPCRITICAL\_SECTION lpCriticalSection);

LeaveCriticalSection()在其他线程中也可以调用（而不仅仅是拥有它的线程），也可以使当前线程拥有者的进入的次数减一。

Ps：对于没有进入关键区域（临界区）的线程使用LeaveCriticalSection()使得当前线程对下次想进入时造成即使进入次数加一也是有人正在使用的状态从而导致无法进入无线等待

## 带有旋转锁的关键段

由于将线程切换到等待状态的开销较大，因此为了提高关键段的性能，Microsoft将旋转锁合并到关键段中，这样EnterCriticalSection()会先用一个旋转锁不断循环，尝试一段时间才会将线程切换到等待状态

Ps: 如果主机只有一个处理器，那么设置旋转锁是无效的。无法进入关键区域的线程总会被系统将其切换到等待状态。

### 初始化关键段并设置旋转次数

函数原型：

BOOLInitializeCriticalSectionAndSpinCount(

LPCRITICAL\_SECTION lpCriticalSection,

DWORD dwSpinCount);

函数说明：旋转次数一般设置为4000。

### 修改关键段的旋转次数

函数原型：

DWORDSetCriticalSectionSpinCount(

LPCRITICAL\_SECTION lpCriticalSection,

DWORD dwSpinCount);

## 事件

#include <process.h>

事件是内核对象，事件分为手动置位事件和自动置位事件。事件Event内部它包含一个使用计数（所有内核对象都有），一个布尔值表示是手动置位事件还是自动置位事件，另一个布尔值用来表示事件有无触发。

### 创建事件

事件触发之后直到设置成未触发所有等这个事件的线程都能运行

函数原型：

HANDLE CreateEvent(

LPSECURITY\_ATTRIBUTES lpEventAttributes,

BOOL bManualReset,

BOOL bInitialState,

LPCTSTR lpName

);

函数说明：

第一个参数表示安全控制，一般直接传入NULL。

第二个参数确定事件是手动置位还是自动置位，传入TRUE表示手动置位，传入FALSE表示自动置位。如果为自动置位，则对该事件调用WaitForSingleObject()后会自动调用ResetEvent()使事件变成未触发状态。打个小小比方，手动置位事件相当于教室门，教室门一旦打开（被触发），所以有人都可以进入直到老师去关上教室门（事件变成未触发）。自动置位事件就相当于医院里拍X光的房间门，门打开后只能进入一个人，这个人进去后会将门关上，其它人不能进入除非门重新被打开（事件重新被触发）。

第三个参数表示事件的初始状态，传入TRUR表示已触发。

第四个参数表示事件的名称，传入NULL表示匿名事件。

### 根据名称获得一个事件句柄。

函数原型：

HANDLE OpenEvent(

DWORD dwDesiredAccess,

BOOL bInheritHandle,

LPCTSTR lpName //名称

);

函数说明：

第一个参数表示访问权限，对事件一般传入EVENT\_ALL\_ACCESS。详细解释可以查看MSDN文档。

第二个参数表示事件句柄继承性，一般传入TRUE即可。

第三个参数表示名称，不同进程中的各线程可以通过名称来确保它们访问同一个事件。

### 触发事件

函数原型：BOOL SetEvent(HANDLE hEvent);

函数说明：每次触发后，必有一个或多个处于等待状态下的线程变成可调度状态

### 设为末触发

函数原型：BOOL ResetEvent(HANDLE hEvent);

### 清理与销毁

函数原型：CloseHandle(HANDLE hEvent)

由于事件是内核对象，因此使用CloseHandle()就可以完成清理与销毁了。

### 例子

#include <stdio.h>

#include <process.h>

#include <windows.h>

long g\_nNum;

unsigned int \_\_stdcall Fun(void \*pPM);

const int THREAD\_NUM = 10;

//事件与关键段

HANDLE g\_hThreadEvent;

CRITICAL\_SECTION g\_csThreadCode;

int main()

{

//初始化事件和关键段 自动置位,初始无触发的匿名事件

g\_hThreadEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

InitializeCriticalSection(&g\_csThreadCode);

HANDLE handle[THREAD\_NUM];

g\_nNum = 0;

int i = 0;

while (i < THREAD\_NUM)

{

handle[i] = (HANDLE)\_beginthreadex(NULL, 0, Fun, &i, 0, NULL);

WaitForSingleObject(g\_hThreadEvent, INFINITE); //等待事件被触发

i++;

}

WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);

//销毁事件和关键段

CloseHandle(g\_hThreadEvent);

DeleteCriticalSection(&g\_csThreadCode);

return 0;

}

unsigned int \_\_stdcall Fun(void \*pPM)

{

int nThreadNum = \*(int \*)pPM;

SetEvent(g\_hThreadEvent); //触发事件

Sleep(50);//some work should to do

EnterCriticalSection(&g\_csThreadCode);

g\_nNum++;

Sleep(0);//some work should to do

printf("线程编号为%d 全局资源值为%d\n", nThreadNum, g\_nNum);

LeaveCriticalSection(&g\_csThreadCode);

return 0;

}

### 事件脉冲

将事件触发后立即将事件设置为未触发，相当于触发一个事件脉冲。

函数原型：BOOLPulseEvent(HANDLE hEvent);

函数说明：这是一个不常用的事件函数，此函数相当于SetEvent()后立即调用ResetEvent();此时情况可以分为两种：

1.对于手动置位事件，所有正处于等待状态下线程都变成可调度状态。

2.对于自动置位事件，所有正处于等待状态下线程只有一个变成可调度状态。

此后事件是末触发的。该函数不稳定，因为无法预知在调用PulseEvent ()时哪些线程正处于等待状态。

下面对这个触发一个事件脉冲PulseEvent ()写一个例子，主线程启动7个子线程，其中有5个线程Sleep(10)后对一事件调用等待函数（称为快线程），另有2个线程Sleep(100)后也对该事件调用等待函数（称为慢线程）。主线程启动所有子线程后再Sleep(50)保证有5个快线程都正处于等待状态中。此时若主线程触发一个事件脉冲，那么对于手动置位事件，这5个线程都将顺利执行下去。对于自动置位事件，这5个线程中会有中一个顺利执行下去。而不论手动置位事件还是自动置位事件，那2个慢线程由于Sleep(100)所以会错过事件脉冲，因此慢线程都会进入等待状态而无法顺利执行下去。

## 互斥量

互斥量也是有“线程拥有权”概念。互斥量能够用于多个进程之间线程互斥问题，并且能完美的解决某进程意外终止所造成的“遗弃”问题。

另外由于互斥量常用于多进程之间的线程互斥，所以它比关键段还多一个很有用的特性——“遗弃”情况的处理。比如有一个占用互斥量的线程在调用ReleaseMutex()触发互斥量前就意外终止了（相当于该互斥量被“遗弃”了），那么所有等待这个互斥量的线程是否会由于该互斥量无法被触发而陷入一个无穷的等待过程中了？这显然不合理。因为占用某个互斥量的线程既然终止了那足以证明它不再使用被该互斥量保护的资源，所以这些资源完全并且应当被其它线程来使用。因此在这种“遗弃”情况下，系统自动把该互斥量内部的线程ID设置为0，并将它的递归计数器复置为0，表示这个互斥量被触发了。然后系统将“公平地”选定一个等待线程来完成调度（被选中的线程的WaitForSingleObject()会返回WAIT\_ABANDONED\_0）。

### 创建互斥量

（注意与事件Event的创建函数对比）

函数原型：

HANDLE CreateMutex(

LPSECURITY\_ATTRIBUTES lpMutexAttributes,

BOOL bInitialOwner,

LPCTSTR lpName

);

函数说明：

第一个参数表示安全控制，一般直接传入NULL。

第二个参数用来确定互斥量的初始拥有者。如果传入TRUE表示互斥量对象内部会记录创建它的线程的线程ID号并将递归计数设置为1，由于该线程ID非零，所以互斥量处于未触发状态。如果传入FALSE，那么互斥量对象内部的线程ID号将设置为NULL，递归计数设置为0，这意味互斥量不为任何线程占用，处于触发状态。

第三个参数用来设置互斥量的名称，在多个进程中的线程就是通过名称来确保它们访问的是同一个互斥量。

函数访问值：

成功返回一个表示互斥量的句柄，失败返回NULL。

### 打开互斥量

存在互斥量只知道名字想使用 得到handle去waitfor

函数原型：

HANDLE OpenMutex(

DWORD dwDesiredAccess,

BOOL bInheritHandle,

LPCTSTR lpName //名称

);

函数说明：

第一个参数表示访问权限，对互斥量一般传入MUTEX\_ALL\_ACCESS。详细解释可以查看MSDN文档。

第二个参数表示互斥量句柄继承性，一般传入TRUE即可。

第三个参数表示名称。某一个进程中的线程创建互斥量后，其它进程中的线程就可以通过这个函数来找到这个互斥量。

函数访问值：

成功返回一个表示互斥量的句柄，失败返回NULL。

### 触发互斥量

用完之后让出资源

函数原型：

BOOL ReleaseMutex (HANDLE hMutex)

函数说明：

访问互斥资源前应该要调用等待函数，结束访问时就要调用ReleaseMutex()来表示自己已经结束访问，其它线程可以开始访问了。

### 清理互斥量

由于互斥量是内核对象，因此使用CloseHandle()就可以（这一点所有内核对象都一样）。

### 例子

#include <stdio.h>

#include <windows.h>

const char MUTEX\_NAME[] = "Mutex\_MoreWindows";

int main()

{

HANDLE hMutex = OpenMutex(MUTEX\_ALL\_ACCESS, TRUE, MUTEX\_NAME); //打开互斥量

if (hMutex == NULL)

{

printf("打开互斥量失败\n");

return 0;

}

printf("等待中....\n");

DWORD dwResult = WaitForSingleObject(hMutex, 20 \* 1000); //等待互斥量被触发

switch (dwResult)

{

case WAIT\_ABANDONED:

printf("拥有互斥量的进程意外终止\n");

break;

case WAIT\_OBJECT\_0:

printf("已经收到信号\n");

break;

case WAIT\_TIMEOUT:

printf("信号未在规定的时间内送到\n");

break;

}

CloseHandle(hMutex);

return 0;

}

互斥量能够用于多个进程之间线程互斥问题，并且能完美的解决某进程意外终止所造成的“遗弃”问题。

## 信号量Semaphore

### 创建信号量

函数原型：

HANDLE CreateSemaphore(

LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes,

LONG lInitialCount,

LONG lMaximumCount,

LPCTSTR lpName

);

函数说明：

第一个参数表示安全控制，一般直接传入NULL。

第二个参数表示初始资源数量。

第三个参数表示最大并发数量。

第四个参数表示信号量的名称，传入NULL表示匿名信号量。

### 打开信号量

函数原型：

HANDLE OpenSemaphore(

DWORD dwDesiredAccess,

BOOL bInheritHandle,

LPCTSTR lpName

);

函数说明：

第一个参数表示访问权限，对一般传入SEMAPHORE\_ALL\_ACCESS。详细解释可以查看MSDN文档。

第二个参数表示信号量句柄继承性，一般传入TRUE即可。

第三个参数表示名称，不同进程中的各线程可以通过名称来确保它们访问同一个信号量。

### 递增信号量的当前资源计数

使用完之后可用资源增加 waitfo之后资源减1

函数原型：

BOOL ReleaseSemaphore(

HANDLE hSemaphore,

LONG lReleaseCount,

LPLONG lpPreviousCount

);

函数说明：

第一个参数是信号量的句柄。

第二个参数表示增加个数，必须大于0且不超过最大资源数量。

第三个参数可以用来传出先前的资源计数，设为NULL表示不需要传出。

注意：当前资源数量大于0，表示信号量处于触发，等于0表示资源已经耗尽故信号量处于末触发。在对信号量调用等待函数时，等待函数会检查信号量的当前资源计数，如果大于0(即信号量处于触发状态)，减1后返回让调用线程继续执行。一个线程可以多次调用等待函数来减小信号量。

### 清理与销毁

由于信号量是内核对象，因此使用CloseHandle()就可以完成清理与销毁了。

## 读写锁

### 初始化读写锁

函数原型：VOID InitializeSRWLock(PSRWLOCK SRWLock);

函数说明：初始化（没有删除或销毁SRWLOCK的函数，系统会自动清理）

### 写入者线程申请写资源。

函数原型：VOID AcquireSRWLockExclusive(PSRWLOCK SRWLock);

### 写入者线程写资源完毕，释放对资源的占用。

函数原型：VOID ReleaseSRWLockExclusive(PSRWLOCK SRWLock);

### 读取者线程申请读资源。

函数原型：VOID AcquireSRWLockShared(PSRWLOCK SRWLock);

### 读取者线程结束读取资源，释放对资源的占用。

函数原型：VOID ReleaseSRWLockShared(PSRWLOCK SRWLock);

注意一个线程仅能锁定资源一次，不能多次锁定资源。

# GitHub

## Git

所有git命令的开头都是git

### 常用命令

git init 将当前目录初始化为一个git仓库

git status 当前仓库状态

git add XXX 添加进仓库 (缓存区)

git commit -m 'first commit' 提交 –m添加信息 信息内容

git log 查看所有产生的 commit 记录

git branch a 创建一个a分支

git checkout a 转换到a分支

git checkout -b a 创建分支a切转换到a分支

git merge a 在master分支使用 将a分支合并

git branch -d a 删除a分支

git branch –D a 强制删除未合并的分支a

git tag v1.0 将当前状态创建为v1.0版本

git checkout v1.0 切换到v1.0代码状态

ssh-keygen -t rsa 生成ssh密匙 结果在.ssh/ id\_rsa.pub中

git push origin master 把本地代码推到远程 master 分支。

git pull origin master 把远程最新的代码更新到本地。一般我们在 push 之前都会先 pull ，这样不容易冲突。

git clone git@github.com:ljzshuai/practice.git

这样就把 test 项目 clone 到了本地，你可以把 clone 命令理解为高级点的复制，这个时候该项目本身就已经是一个git 仓库了，不需要执行 git init 进行初始化，而且甚至都已经关联好了远程仓库，我们只需要在这个 test 目录下任意修改或者添加文件，然后进行 commit ，之后就可以执行：git push origin master

git remote add origin [git@github.com:ljzshuai/practice.git](mailto:git@github.com:ljzshuai/practice.git)

将本地的一个仓库关联一个远程仓库 之后就可以git push origin master

git remote –v当前项目有哪些远程仓库

在提交代码之前先要设置下自己的用户名与邮箱，这些信息会出现在所有的 commit 记里，执行以下代码就可以设置：

git config —global user.name "ljzshuai"

git config —global user.email "haoliujiazhi@qq.com"

### 配置

git config --global core.editor "vim" # 设置Editor使用vim

git config --global color.ui true 终端上色

git config --global core.quotepath false # 设置显示中文文件名

git config -l 命令查看配置

### 用户名和邮箱

我们知道我们进行的每一次 commit 都会产生一条 log，这条 log 标记了提交人的姓名与邮箱，以便其他人方便的查看与联系提交人，所以我们在进行提交代码的第一步就是要设置自己的用户名与邮箱。执行以下代码：

git config --global user.name "stormzhang"

git config --global user.email "stormzhang.dev@gmail.com"

以上进行了全局配置，当然有些时候我们的某一个项目想要用特定的邮箱，这个时候只需切换到你的项目目录，以上代码把 --global 参数去除，再重新执行一遍就ok了。

PS：我们在 GitHub 的每次提交理论上都会在主页的下面产生一条绿色小方块的记录，如果你确认你提交了，但是没有绿色方块显示，那肯定是你提交代码配置的邮箱跟你 GitHub 上的邮箱不一致，GitHub 上的邮箱可以到 Setting -> Emails 里查看。

### 别名Alias

git config --global http://alias.co checkout # 别名

git config --global alias.ci commit

git config --global alias.st status

git config --global alias.br branch

当然以上别名不是固定的，你完全可以根据自己的习惯去定制，除此之外还可以设置组合，比如：

git config --global alias.psm 'push origin master'

git config --global alias.plm 'pull origin master'

之后经常用到的 git push origin master 和 git pull origin master 直接就用 git psm 和 git plm 代替了

git log –graph –pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --date=relative

git config --global alias.lg "log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --date=relative"

这样以后直接输入 git lg 就行了

### Diff

diff 命令算是很常用的，使用场景是我们经常在做代码改动，但是有的时候2天前的代码了，做了哪些改动都忘记了，在提交之前需要确认下，这个时候就可以用diff来查看你到底做了哪些改动，举个例子，比如我有一个 a.md 的文件，我现在做了一些改动，然后输入 git diff

红色的部分前面有个 - 代表我删除的，绿色的部分前面有个 + 代表我增加的，所以从这里你们能一目了然的知道我到底对这个文件做了哪些改动。

值得一提的是直接输入 git diff 只能比较当前文件和缓存区文件差异，什么是缓存区？就是你还没有执行 git add 的文件。

当然跟暂存区做比较之外，他还可以有其他用法，如比较两次 commit 之间的差异，比较两个分支之间的差异，比较缓存区和版本库之间的差异等，具体用法如下：

git diff <$id1> <$id2> # 比较两次提交之间的差异

git diff <branch1>..<branch2> # 在两个分支之间比较

git diff --staged # 比较暂存区和版本库差异

### Stash

设想一个场景，假设我们正在一个新的分支做新的功能，这个时候突然有一个紧急的bug需要修复，而且修复完之后需要立即发布。当然你说我先把刚写的一点代码进行提交不就行了么？这样理论上当然是ok的，但是这会产品垃圾commit，原则上我们每次的commit都要有实际的意义，你的代码只是刚写了一半，还没有什么实际的意义是不建议就这样commit的，那么有没有一种比较好的办法，可以让我暂时切到别的分支，修复完bug再切回来，而且代码也能保留的呢？

这个时候 stash 命令就大有用处了，前提是我们的代码没有进行 commit ，哪怕你执行了 add 也没关系，我们先执行

git stash

什么意思呢？就是把当前分支所有没有 commit 的代码先暂存起来，这个时候你再执行 git status 你会发现当前分支很干净，几乎看不到任何改动，你的代码改动也看不见了，但其实是暂存起来了。执行

git stash list

你会发现此时暂存区已经有了一条记录。

这个时候你可以切换回其他分支，赶紧把bug修复好，然后发布。之后一切都解决了，你再切换回来继续做你之前没做完的功能，但是之前的代码怎么还原呢？

git stash apply

你会发现你之前的代码全部又回来了，就好像一切都没发生过一样，紧接着你最好需要把暂存区的这次 stash 记录删除，执行：

git stash drop

就把最近一条的 stash 记录删除了，是不是很方便？其实还有更方便的，你可以使用：

git stash pop

来代替 apply 命令，pop 跟 apply 的唯一区别就是 pop 不但会帮你把代码还原，还自动帮你把这条 stash 记录删除，省的自己再 drop 一次了，虽然更方便，但是使用起来也需要更加谨慎，为了验证你可以紧接着执行 git stash list 命令来确认是不是已经没有该记录了。

最后还有一个命令介绍下：

git stash clear

就是清空所有暂存区的记录，drop 是只删除一条，当然后面可以跟 stash\_id 参数来删除指定的某条记录，不跟参数就是删除最近的，而 clear 是清空。

### merge & rebase

我们知道 merge 分支是合并的意思，我们在一个 featureA 分支开发完了一个功能，这个时候需要合并到主分支 master 上去，我们只需要进行如下操作：

git checkout master

git merge featureA

其实 rebase 命令也是合并的意思，上面的需求我们一样可以如下操作：

git checkout master

git rebase featureA

rebase 跟 merge 的区别你们可以理解成有两个书架，你需要把两个书架的书整理到一起去，第一种做法是 merge ，比较粗鲁暴力，就直接腾出一块地方把另一个书架的书全部放进去，虽然暴力，但是这种做法你可以知道哪些书是来自另一个书架的；第二种做法就是 rebase ，他会把两个书架的书先进行比较，按照购书的时间来给他重新排序，然后重新放置好，这样做的好处就是合并之后的书架看起来很有逻辑，但是你很难清晰的知道哪些书来自哪个书架。

只能说各有好处，不同的团队根据不同的需要以及不同的习惯来选择就好

### 解决冲突

假设这样一个场景，A和B两位同学各自开了两个分支来开发不同的功能，大部分情况下都会尽量互不干扰的，但是有一个需求A需要改动一个基础库中的一个类的方法，不巧B这个时候由于业务需要也改动了基础库的这个方法，因为这种情况比较特殊，A和B都认为不会对别人造成影响，等两人各自把功能做完了，需要合并的到主分支 master 的时候，我们假设先合并A的分支，这个时候没问题的，之后再继续合并B的分支，这个时候想想也知道会有冲突了，因为A和B两个人同时更改了同一个地方，Git 本身他没法判断你们两个谁更改的对，但是这个时候他会智能的提示有 conflicts ，需要手动解决这个冲突之后再重新进行一次 commit 提交。我随便在项目搞了一个冲突做下示例：



以上截图里就是冲突的示例，冲突的地方由 ==== 分出了上下两个部分，上部分一个有 HEAD 的字样代表是我当前所在分支的代码，下半部分是一个叫 baidu\_activity 分支的代码，可以看到 HEAD 对 gradle 插件进行了升级，同时新增了一个插件，所以我们很容易判断哪些代码该保留，哪些代码该删除，我们只需要移除掉那些老旧代码，而且同时也要把那些 <<< HEAD、==== 以及 >>>>>>baidu\_activity 这些标记符号也一并删除，最后进行一次 commit 就ok了。

我们在开发的过程中一般都会约定尽量大家写的代码不要彼此影响，以减少出现冲突的可能，但是冲突总归无法避免的，我们需要了解并掌握解决冲突的方法。

### checkout

我们知道 checkout 一般用作切换分支使用，比如切换到 develop 分支，可以执行：

git checkout develop

但是 checkout 不只用作切换分支，他可以用来切换 tag，切换到某次 commit，如：

git checkout v1.0

git checkout ffd9f2dd68f1eb21d36cee50dbdd504e95d9c8f7

# 后面的一长串是commit\_id，是每次commit的SHA1值，可以根据 git log 看到。

除了有“切换”的意思，checkout 还有一个撤销的作用，举个例子，假设我们在一个分支开发一个小功能，刚写完一半，这时候需求变了，而且是大变化，之前写的代码完全用不了了，好在你刚写，甚至都没有 git add 进暂存区，这个时候很简单的一个操作就直接把原文件还原：

git checkout a.md

这里稍微提下，checkout 命令只能撤销还没有 add 进暂存区的文件

# QT

label->setAttribute(Qt::WA\_DeleteOnClose);

## 查询

退出app QApplication::quit

回车触发 http://bbs.csdn.net/topics/380206594

## 定义信号

只有继承了QObject类的类，才具有信号槽的能力

凡是QObject类（不管是直接子类还是间接子类），都应该在第一行代码写上Q\_OBJECT

class test\_signal:public QObject

{

Q\_OBJECT

public:

test\_signal(){}

~test\_signal(){}

void send()

{

emit somethings("success");

}

signals:

void somethings(const std::string&);

private:

};

## 动作QAction

抽象出公共的动作。当我们把QAction对象添加到菜单，就显示成一个菜单项，添加到工具栏，就显示成一个工具按钮。用户可以通过点击菜单项、点击工具栏按钮、点击快捷键来激活这个动作。

## 添加资源

Qrc 中add用绝对路径添加所需要的资源

给资源起一个别名 直接使用:/qt\_practice/client来使用你的资源

该资源是在编译的时候以二进制的形式添加到exe中

# 杂

局部对象的析构顺序应该按照其创建顺序的相反过程

## 变量生命周期

函数和{}

## 数组

* 1. 数组是特例化的数据结构

char a[10];

char [10] a; //错误写法

a是一个 int[10] 类型的变量 拥有10个字符

所以sizeof(a)=10\*1=10;

* 1. 数组名的外延在于其可以转换为指向其指代实体的指针，而且是一个指针常量；

数组可以隐式转换为指针常量来使用

* 1. 指向数组的指针则是另外一种变量类型（在WIN32平台下，长度为4），仅仅意味着数组的存放地址！

auto p =&a;

p的类型是 char(\*)[10];

* 1. 数组名作为函数形参时，在函数体内，其失去了本身的内涵，仅仅只是一个指针；

在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。

## Std::string int double

int i = std::stoi(str);

## 虚函数

typedef void(\*Fun)(void);

Base b; //base有虚函数

&b 取b的首地址

(Int\*)(&b) 解引用后使用 b的前4字节

\*(int\*)(&b) 解引用得到虚函数表头指针

(int\*)\*(int\*)(&b) 解引用后使用虚函数表的前4字节

\*(int\*)\*(int\*)(&b) 解引用得到虚函数表第一个函数

(Fun) \*(int\*)\*(int\*)(&b) 转换成函数来使用

如果一个类或者它继承的类有虚函数 那么这个类就会有一张虚函数表

每一个实体在开头都有一个指向这张表的指针

表里的函数按照声明顺序存放，如果子类覆盖的父类的虚函数那么就在表里用子类的虚函数覆盖父类的虚函数，没有覆盖的虚函数顺序加到表的后面，继承了几个有虚函数的父类就会有几张表，实例开头就有几个指针。子类的表和父类的表没有关系，子类动态绑定到使用子类的表。