相比于0.2版本0.1版本下做了以下修改：

1、使用了RALL锁机制，使锁能够自动释放：（资源获取就是初始化）

其核心是把资源和对象的生命周期绑定，对象创建获取资源，对象销毁释放资源。在 RAII 的指导下，C++ 把底层的资源管理问题提升到了对象生命周期管理的更高层次。

RALL是c++中用于资源管理的一种重要的编程思想

在写[C++设计模式——单例模式](http://www.jellythink.com/archives/82)的时候，在写到实例销毁时，设计的GC类是很巧妙的，而这一巧妙的设计就是根据当对象的生命周期结束时会自动调用其析构函数的，而这一巧妙的设计也是有专业的名词的——RAII。那以下将围绕RAII，全面的讲解RAII的相关知识。

什么是RAII？

RAII是Resource Acquisition Is Initialization的简称，是C++语言的一种管理资源、避免泄漏的惯用法。利用的就是C++构造的对象最终会被销毁的原则。RAII的做法是使用一个对象，在其构造时获取对应的资源，在对象生命期内控制对资源的访问，使之始终保持有效，最后在对象析构的时候，释放构造时获取的资源。

为什么要使用RAII？

上面说到RAII是用来管理资源、避免资源泄漏的方法。那么，用了这么久了，也写了这么多程序了，口头上经常会说资源，那么资源是如何定义的？在计算机系统中，资源是数量有限且对系统正常运行具有一定作用的元素。比如：网络套接字、互斥锁、文件句柄和内存等等，它们属于系统资源。由于系统的资源是有限的，就好比自然界的石油，铁矿一样，不是取之不尽，用之不竭的，所以，我们在编程使用系统资源时，都必须遵循一个步骤：

申请资源；使用资源；释放资源。

第一步和第二步缺一不可，因为资源必须要申请才能使用的，使用完成以后，必须要释放，如果不释放的话，就会造成资源泄漏。

2、修改锁的方式

class MutexLockGuard//管理资源

{

public:

//避免在对象初始化时隐式调用该构造函数 只能通过显示调用 及不支持隐式类型转换

explicit MutexLockGuard();

~MutexLockGuard();

private:

static pthread\_mutex\_t lock;

private:

MutexLockGuard(const MutexLockGuard&);

MutexLockGuard& operator=(const MutexLockGuard);

};

2、修改mtimer锁的方式

mytimer \*mtimer = new mytimer(this, 500);

timer = mtimer;

{

//LockGuard作用于该代码块 方便了代码的书写 且该锁实现的是RALL机制 能起到管理资源的作用 调用构造函数即初始化 资源

//该类的析构函数 调用时即释放资源

MutexLockGuard();

myTimerQueue.*push*(mtimer);

}

3、将handle\_expired\_event()中的加锁方式改变

void handle\_expired\_event()

{

MutexLockGuard();

//pthread\_mutex\_lock(&qlock);

while (!myTimerQueue.*empty*())

{

mytimer \*ptimer\_now = myTimerQueue.*top*();

if (ptimer\_now->isDeleted())

{

myTimerQueue.*pop*();

delete ptimer\_now;

}

else if (ptimer\_now->isvalid() == false)

{

myTimerQueue.*pop*();

delete ptimer\_now;

}

else

{

break;

}

}

//pthread\_mutex\_unlock(&qlock);

}