ThreadJack – 1.0

レファレンス

作成日 1999 年 4 月 19 日

All contents Copyright(C) 1999 Eiwa System Management, Inc. All rights reserved.

目次

論理ビューのドキュメント

論理ﾋﾞｭｰ

MT

TJMutex

排他制御プリミティブ.

このクラスはC言語の排他制御プリミティブ(システム依存ルーチン)をカプセル化することが目的でありかなり低レベルである.

本クラスにおけるロックは二重に呼び出すとシステムによっては
デッドロックが発生する.
もっと安全なモニタ, セマフォ, R/Wロックなどが用意されているのでそれらを使うことを推奨する.（禁止はしない）

@author Susumu TOKUI
@date 1998,6/1

Public 操作 :

TJMutex () : TJMutex

TJMutex オブジェクトを作成する.

lock () : void

ロックを獲得する.

~TJMutex () :

TJMutex オブジェクトを消去する.

unlock () : void

ロックを解放する.

TJRunnable

スレッドにより実行されるプログラムのインタフェースを規定する抽象クラス.

TJRunnable は，スレッドにより実行されるプログラムの共通のプロトコル
を規定するためにあり，run() メソッドがそのプログラム本体である．スレッドが
run() を実行中の時，そのスレッドは alive であるという．

TJRunnable オブジェクトを TJThread オブジェクトに渡した場合，
TJThread の start() メソッドを呼び出すことによって，run()が
呼び出される．

```
class MyRunnable : public TJRunnable {
        ・・・・
  public:
        ・・・・
    void run() {
        ・・・・  // implement TJRunnable::run()
    }
};

MyRunnable* runnable = new MyRunnable;
TJThread* thread = new TJThread(runnable, "myname");
thread->start(); // MyRunnalbe::run() が thread によって実行される
.........
thread->join();  // 必要であれば終了待ち合わせ
```

run() の終了は，他のスレッドが TJRunnable をターゲットとする TJThread
の join() メソッドで待ち合わせることができる．
run() をそのまま抜ければ join() で受け取る
終了コードは 0 である．明示的に終了コードを join() に渡すには，
run() の終了時に TJThread::exit(int) を呼び出せばよい．

TJRunnable を使用せずに，TJThread を直接サブクラス化してもスレッド本体
を表現できる(TJThread 参照)．TJRunnable を使用する利点は，1つの
TJRunnable オブジェクトを複数の TJThread オブジェクトで共用できる
点である．

Java の仕様と違い，TJThread 自身は TJRunnable を継承しない．
これは，TJThread コンストラクタに TJThread を渡すというミスを防ぐ
意味がある．

TJThread と TJRunnable の使い分けについては，TJThread を参照すること．

@author Susumu TOKUI
@date 1998,6/1
@see TJThread

Public 操作 :

run () : void

スレッド本体のインタフェース.

スレッドによって実行されるプログラムのエントリポイント

となる.

@see TJThread#run()
@see TJThread#start()

TJSync

同期制御プリミティブ.

このクラスはC言語の同期制御プリミティブ(システム依存ルーチン)を
カプセル化することが目的でありかなり低レベルである.
直接使わないことを推奨するが禁止はしない.

このクラスはオブジェクトを作成する時には排他制御用オブジェクトを
必要とする.
もし単純な同期が必要ならセマフォクラス(TJSemaphore)を用意して
いるのでそちらを使用することを推奨する.

@author Susumu TOKUI
@date 1998,6/1

Public 操作 :

TJSync (m : const TJMutex*) : TJSync

TJSync オブジェクトを作成する.

@param m このオブジェクトと対をなす TJMutex オブジェクト

wait () : void

別のスレッドから起こされるまでブロックする.

~TJSync () :

TJSync オブジェクトを消去する.

timedwait (milliseconds : long) : bool

タイムアウトを指定して別のスレッドから起こされるまでブロックする.

@param milliseconds ミリ秒数
@return true: 起こされた, false: タイムアウト

signal () : void

このオブジェクトで待っているスレッドを起こす.

broadcast () : void

このオブジェクトで待っているすべてのスレッドを起こす.

TJSyncObject

スレッド間の排他・同期制御を目的とするクラス.

このクラスは排他制御または同期制御を必要とする場合の基本的な機能を提供する.

基本的なメカニズムは java と同様, モニタと条件変数によるものである.
このクラスを継承したクラスは java.lang.Object と同様に synchronized ブロックを持つことができ， wait / notify によるスレッド間の同期が可能となる. java ではすべてのオブジェクトがこの性質を持つが, このライブラリでは TJSyncObject として独立させて継承などによりこの性質を付加(mix-in)する方針をとる.

基本的には，このクラスは virtual 継承して使用する．そうすれば，同期のセマンティクスは java.lang.Object のサブクラスと同様になる.

【例】 class Xxx : virtual public TJSyncObject { ... }

意味の異なる複数のロックが1オブジェクト内で必要な場合には，このクラスのオブジェクトをインスタンス変数として包含する使い方も可能である.
ただし，あるオブジェクトの中に複数の TJSyncObject が含まれる場合，誤ったロックを獲得して意図しない排他制御となる可能性があるため，注意が必要である.

TJSyncObject は次のような機能を持っている.

◆このクラスのサブクラスのメソッドの先頭で lock() を呼び出すことによりそのメソッドはモニタオペレーションとなる.
ここでモニタオペレーションとは java の synchronized メソッドと考えてよい. どのメソッドをモニタオペレーションとするかは java と同様プログラマの選択にまかされる.

◆スレッドはモニタオペレーション実行中はそのオブジェクトに対する排他的なアクセスが保証される.

◆モニタオペレーションの中で wait() を実行すると排他的アクセス権を失ってブロックし他のスレッドによる notify() 呼出しまで待たされる.

◆他のスレッドによる notify() 呼出しによって制御を得た時は再び排他的アクセス権が獲得されている.

モニタオペレーションのための排他ロックは同一スレッドならばネストすることが可能である. wait() ではネストされたロックを解放し notify() ではネストレベルを含めてロック状態を復元する.

したがってモニタオペレーションの中で別のモニタオペレーションを呼び出しも
可能である．この仕様は，java.lang.Object と同様である．

ただし，このクラスの protected メソッド(lock(), unlock(), ...)
は直接呼び出さず TJMonitor クラスと synchronized マクロを経由して
使うことを推奨する．
TJMonitor クラスと synchronized マクロは，より安全性と
ドキュメント性を高めている．(TJMonitor を参照)

java.lang.Object の仕様を参考にしているため次のような使い方を想定している．

```
          java                                TJSyncObject
-------------------------------------------------------------
    synchronized void method() {          void method() {
                                              lock();
          ・・・                               ・・・
                                              unlock();
    }                                     }
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    synchronized void method() {          void method() {
                                              lock();
          ・・・                                 ・・・
        while (busy)                          while (busy)
            wait(); // catch 省略                 wait();
        // object を操作                      // object を操作
          ・・・                                 ・・・
                                              unlock();
    }                                     }
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    synchronized void method() {          void method() {
                                              lock();
          ・・・                                 ・・・
        notify();                             notify();
          ・・・                                 ・・・
                                              unlock();
    }                                     }
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    synchronized (obj) {                  obj.lock();
          ・・・                               ・・・
    }                                     obj.unlock();
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    synchronized (obj) {                  obj.lock();
          ・・・                               ・・・
        while (obj.busy)                  while (obj.busy)
            obj.wait(); // catch 省略             obj.wait();
        // obj を操作                         // obj を操作
          ・・・                               ・・・
    }                                     obj.unlock();
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    synchronized (obj) {                  obj.lock();
          ・・・                               ・・・
```

```
        obj.notify();                    obj.notify();
          ・・・                           ・・・
    }                                    obj.unlock();
-------------------------------------------------------
```

上記を synchronized ブロック記法によってサポートする例は,
TJMonitor を参照のこと.

また, unlockTemporarily(), restoreLock() は java には存在しない機能である. これは, synchronized ブロック中に, ロックを外すことができる機能である. java では, synchronized ブロックを書くことで synchronized 中であることは保証できるが, 逆に呼び出しのネスト関係によりコードのある部分が synchronized 外あることは保証できない.

この機能により, デッドロックの回避のためのロックの順序化(階層化)を論理的にサポートすることができる.

@author Susumu TOKUI
@date 1998,6/1
@see TJMonitor

Public 操作 :

TJSyncObject () : TJSyncObject

TJSyncObject オブジェクトを作成する.

TJSyncObject (obj : const TJSyncObject&) : TJSyncObject

TJSyncObject のコピーコンストラクタ.

引数のオブジェクトを無視して新規作成する.
@param obj 生成元

operator = (obj : const TJSyncObject&) : TJSyncObject&

代入演算.

代入に対してこのオブジェクトを変化させない.
@param obj 代入元

wait () : void

他のスレッドからの notify を無限に待ち合わせる.
wait(0)と同義.

wait (milliseconds : long) : bool

他のスレッドからの notify を最大指定時間待ち合わせる.
@param milliseconds ミリ秒数 (wait(0) は wait() と同じ動作)
@return true: notify された, false: タイムアウト

~TJSyncObject () :

TJSyncObject オブジェクトを消去する.

notify () : void

このオブジェクトで待っている他のスレッドに通知する.
このオブジェクトで wait している他のスレッドは,
wait から復帰し，実行可能となる.

notifyAll () : void

このオブジェクトで待っている他のすべてのスレッドに通知する.
このオブジェクトで wait している他のスレッドは,
wait から復帰し，実行可能となる.

invariant () : bool

不変条件.

サブクラスでは，このメソッドを継承して自己一貫性チェックを
書くことを推奨する.
開発時に assert() の引数として利用できる.
@return オブジェクトの状態に矛盾がなければ true

TJThread

スレッドをカプセル化するためのクラス.

ThreadJack では，複数のスレッドを並行動作させることができる.
このクラスはスレッドの作成・消去やスレッドに対する操作をカプセル化している.

ThreadJack では, スレッドによって実行されるプログラムの実装方法には,
以下の 2つがある.

1. TJThread をサブクラス化し，run() をオーバーライドする.

2. TJRunnable をサブクラス化し，run() をオーバーライドする.

それぞれの実行方法は，次の通り.
いずれの場合も TJThread またはそのサブクラスのオブジェクトが必要である.

(1) TJThread のサブクラスのオブジェクトを作成

```
class MyThread : public TJThread {
        ・・・・
  public:
        MyThread(const char* name) : TJThread(name) { }
        ・・・・
        void run() {
                ・・・・  // override TJThread::run()
```

```
        }
};

MyThread* thread = new MyThread("myname");
thread->start();
.....
thread->join();  // 必要であれば終了待ち合わせ
// スレッドオブジェクトは delete してはならない
```

(2) TJRunnable の実装クラスのオブジェクトを引数にして
TJThread オブジェクトを作成

```
class MyRunnable : public TJRunnable {
        ・・・・
  public:
        ・・・・
        void run() {
                ・・・・  // implement TJRunnable::run()
        }
};

MyRunnable* runnable = new MyRunnable;
TJThread* thread = new TJThread(runnable, "myname");
thread->start();
.....
thread->join();  // 必要であれば終了待ち合わせ
// スレッドオブジェクトは delete してはならない
```

(2)の使用法は，1つの TJRunnable オブジェクトを複数の TJThread オブジェクトで共用できる，という利点がある．

TJThread は java.lang.Thread に似ているが java とはメモリ管理方法が異なるためスレッドオブジェクトの作成と消去に次の注意が必要になる.

◆スレッドオブジェクトは必ず new 演算子によりヒープ領域に確保すること.
◆スレッドオブジェクトはアプリケーションが delete しないこと.

あるスレッドが新たに別のスレッドを生成した場合，
生成元のスレッドを親スレッド，生成されたスレッドを子スレッドと呼ぶ.

[スレッドの状態とシステムの終了]

ANSI/ISO C++ により規定される int main() というエントリポイント
に最初に入るスレッドを, 特に「main スレッド」と呼び，アプリケーション
が TJThread クラスのインスタンスとして生成したスレッドと必要に応じて
区別することがある.

main スレッド以外のスレッドでは，スレッドの実行プログラムは
TJThread もしくは TJRunnable の run() メソッドの実装として表現される.

スレッドが run() を実行中の時，そのスレッドは alive であるという.

run() から復帰するか，または明示的に TJThread::exit(int) をよびだすことで run() は終了し，alive 状態から非 alive 状態へと遷移する．
スレッドの終了コードは，run() から自然に復帰した場合は0 となり，TJThread::exit(int) を呼び出した場合は引数が終了コードとなる．
この終了コードは他のスレッドが TJThread::join で受け取ることができる．

main スレッドも，TJThread::exit(int) を呼び出すことができる．この場合も main スレッドが非 alive 状態となる．

ThreadJack に於けるシステム(プロセス)の終了は，次のどれかが起こった時である．

1. main スレッドが main からの復帰(return)した時．
   (ただしTJThread::exit()呼び出しは復帰と見なさない)
2. あるスレッド(main 以外でもよい)が大域のシステム終了(プロセス終了)を引き起こす呼び出し(ANSI/ISO C++ では ::exit() もしくは ::abort())を呼び出した時．
3. 全スレッドが非 alive となった時．

また，ThreadJack では プロセスのコピーを行うシステム動作
(UNIX の fork, Win32 の CreateProcess)に対しての動作は未定義である．
(将来の拡張項目とする)

[プライオリティ]

すべてのスレッドはプライオリティを持つ．一般に, プライオリティの高いスレッドはプライオリティの低いスレッドに優先して実行される．
ただし，プライオリティがスケジューリングに与える正確な影響はシステム依存である.

プライオリティの数値は Java と同じ仕様とする.プライオリティはそのスレッドを生成したスレッドから継承され setPriority() により変更できる.

プライオリティは，スレッドの親子関係の中で継承される．すなわち，新たに生成されたスレッドのプライオリティの初期値は親スレッドのプライオリティと同じ値である．

[待ち合わせ性]

スレッドオブジェクトに対するjoin()の呼び出しによって，他のスレッドがその終了を待ち合わせることができる．
この性質を待ち合わせ性(joinability)という．
しかし,作成したスレッドの終了に関心がない場合，意識的に待ち合わせを省略することも可能である．

(a) 作成したスレッドの終了に関心がある場合

```
MyThread* t = new MyThread("myname");
t->start();  // joinable なスレッド
```

```
    ・・・
    int exitcode = t->join();
```

(b) 作成したスレッドの終了に関心がない場合

```
    FreeThread* t = new FreeThread("myname");
    t->setJoinable(false); // 非 joinable なスレッド
    t->start();
    // これ以降，t の実行状態はこのスレッドから切り離される．
    // [警告] これ以後の *t のアクセスは一般に危険
```

ここで (a) では作成されたスレッドが終了したとき join() が復帰し戻り値に exit() の引数が返される. それに対して (b) では作成されたスレッドとの関わり合いは一切ない. 作成したスレッドの終了状態や終了のタイミングを知る必要がない場合にはこの方法は都合がよい.ただし，joinable なスレッドは必ず join する必要があり，逆に 非 joinable なスレッドは決して join してはいけない．

新規作成されたスレッドの待ち合わせ性は必ず joinable であり, 待ち合わせ性は子スレッドに継承されることはない.
setJoinable(false) の操作を受けたスレッドだけが 非joinable となる．
非 joinable スレッドを join() した場合の動作は不定である. joinable かどうかは setJoinable(bool) により変更可能であるが main スレッドだけは変更できず, 必ず非 joinable である．

待ち合わせ性は，java.lang.Thread の Deamon 性とは無関係の属性であることに注意．

[その他]

スレッドは名前を持つ．名前は必ずしも全スレッドを通して一意ではない．名前を指定せずに生成したスレッドには，適当な名前が付与される．

java.lang.Thread は java.lang.Runnable を implements しているが, ThreadJack では２つのクラスTJThread と TJRunnable には継承関係を持たせていない.

TJThread のサブクラスを作る場合,そのクラスのデストラクタを protected とすることを推奨する. これにより，スレッドを自動変数として生成したり，delete を直接呼び出すといったバグをコンパイル時に発見できる．

カレントスレッド以外のスレッドを強制終了させる機能(いわゆるキャンセル)は提供しない. これを提供するとプログラマはキャンセルからの保護を検討しなければならなくなる. java.lang.Thread の suspend, stop, resume は提供しない．

スレッドクラスのオブジェクトを扱う場合，特に this オブジェクトに対する操作とカレントスレッドに対する操作の混乱が起こりやすいので注意が必要である．
このクラスの主な public メソッドは以下のように分類される.

[A] インスタンスメソッド -- this オブジェクトに対する操作

```
    void setPriority(int)
```

int getPriority()
void setJoinable(bool)
bool isJoinable()
bool isAlive()
void setName(const char*)
const char* getName()
TJThread::ID getThreadID()

[B] **クラスメソッド -- クラス全体の操作**
TJThread* getThread(TJThread::ID)
int activeCount()

[C] **カレントスレッドに対する操作**
void exit(int)
void sleep(long)
void yield()
TJThread::ID currentThreadID()
TJThread* currentThread()
const char* currentThreadName()

[D] **親スレッドの立場での子スレッドに対する操作**
TJThread(...)
void start()
int join()
int join(long)

[E] **通常はユーザが直接呼び出すことの無い操作**
void run()

特に注意すべき点は [C] **のカレントスレッドに対する操作である. これらは すべて** static **であるため形式的にはクラスメソッドであるがカレントスレッド という特定のスレッドに対する操作である.**

スレッドの識別情報としてスレッド識別子(ID)**とスレッドオブジェクトへの ポインタが考えられる. 前者は効率的に実装できるが, 後者のは一般に 内部的なクラスロックを必要とするため, 非効率である. したがってスレッドの識別のみ必要ならスレッド識別子を使うべきである.**

@author Susumu TOKUI
@date 1998,6/20

Public **属性** :

JOINDETACHED : const int

join() **の例外的な戻り値.**

意味 -- 非 joinable **なスレッドを** join() **しようとした.** JOINABLE **ではないスレッドを** join() **した場合の動作は不定である. もしタイミングにより暴走をまぬがれた場合はこの値が戻される. この値は便宜上のものであり, アプリケーションはこの値を使った コードを書いてはならない. 値は,** INT_MAX

JOINTIMEOUT : const int

join() **の例外的な戻り値.**

意味 -- タイムアウト値指定の join() でタイムアウトした.
値は, INT_MAX-1

MIN_PRIORITY : const int

スレッド優先度の最小値.

MAX_PRIORITY : const int

スレッド優先度の最大値.

DEFSTACKSIZE : const unsigned int

スレッドのデフォルトスタックサイズ(16kB).

Public 操作 :

TJThread (stacksize : unsigned int = DEFSTACKSIZE) : TJThread

TJThread オブジェクトを作成する.

@param stacksize スタックサイズ

TJThread (name : const char*, stacksize : unsigned int = DEFSTACKSIZE) : TJThread

TJThread オブジェクトを作成する.

@param name スレッド名
@param stacksize スタックサイズ

TJThread (target : TJRunnable*, stacksize : unsigned int = DEFSTACKSIZE) : TJThread

TJThread オブジェクトを作成する.

@param target TJRunnable の実装クラスのオブジェクト
@param stacksize スタックサイズ

TJThread (target : TJRunnable*, name : const char*, stacksize : unsigned int = DEFSTACKSIZE) : TJThread

TJThread オブジェクトを作成する.

@param target TJRunnable の実装クラスのオブジェクト
@param name スレッド名
@param stacksize スタックサイズ

setPriority (pri : int) : void

スレッドのプライオリティを設定する.
スレッドのプライオリティは Java と同じ仕様を採用する.
@param pri 優先度

getPriority () : int

スレッドのプライオリティを取得する.

スレッドのプライオリティは Java と同じ仕様を採用する.

@return プライオリティ

setJoinable (on : bool) : void

Joinable スレッドの状態に設定する.

isJoinable () : bool

Joinable スレッドならば true を返す.

start () : void

スレッドの実行を開始する.

run () : void

virtual void stop(); // この機能は意図的に用意しない.

スレッド本体.

このメソッドはライブラリが呼び出すスレッドのエントリポイントであり, ユーザが呼び出すことは無意味である.

join () : int

スレッド終了を待ち合わせる.

TJThread::exit によって返されるコードを受け取ることができる.
joinable ではないスレッドに対して join() を実行した場合の動作は不定である.

@return スレッド終了コード
@see TJThread#exit()

join (milliseconds : long) : int

タイムアウトを指定してスレッド終了を待ち合わせる.

TJThread::exit によって返されるコードを受け取ることができる.
@param milliseconds ミリ秒
@return スレッド終了コード
TJThread::JOINTIMEOUT タイムアウト
@see TJThread#exit()

exit (code : int) : void

カレントスレッドを終了させる.

@param code スレッド終了コード
@see TJThread#join()

スレッド終了コードとして
TJThread::JOINDETACHED, TJThread::JOINTIMEOUT

は予約されているので使わないこと.

sleep (milliseconds : long) : void

カレントスレッドを指定時間休止させる.

@param milliseconds ミリ秒

yield () : void

カレントスレッドのＣＰＵを一時的に放棄する.
一時的に実行を他のスレッドに譲る.

setName (name : const char*) : void

スレッド名を設定する.

ただしスレッド名の一意性はこのメソッドでは保証されない.
@param name スレッド名

getName () : const char*

スレッド名を返す.

@return スレッド名

isAlive () : bool

run() の中を実行中ならば true を返す.

@return run()実行中なら true

activeCount () : int

スレッド数を返す.

@return スレッド数

getThreadID () : TJThread::ID

スレッド識別子を返す.

getThread (tid : TJThread::ID) : TJThread*

スレッド識別子からスレッドオブジェクトのポインタを得る.
@param tid スレッド識別子

currentThreadID () : TJThread::ID

カレントスレッドのスレッド識別子を返す.

currentThread () : TJThread*

カレントスレッドオブジェクトのポインタを返す.

currentThreadName () : const char*

カレントスレッドの名前を返す.

putData (anydata : void*) : void

アプリケーションデータをスレッドに記憶する.

このデータはスレッド毎に保持できる.
通常はスタックがあればスレッドごとの情報を記憶できるがスタックで
はメソッドから復帰すると消滅する.
スレッドごとに, かつメソッドから復帰しても保存されるデータを管理
したい場合にこの機能が使える.
管理されるデータの寿命についてはアプリケーションの責任である.

getData () : void*

スレッドごとの情報を返却する.

MT-Support

TJMonitor

モニタオペレーションを支援するためのクラス.

モニタとは内部に共有データを持ち, その操作メソッドが必ず排他的なアクセス
となるように保証された抽象データ型である. そしてモニタオペレーションとは
排他的アクセスの保証されたメソッドを意味するものとする.
Java の synchronized method は Object をモニタとするモニタオペレーション
の例である.

このクラスはできるだけ安全にモニタオペレーションを実現することを目的とし
ている. また, java の synchronized block 記法もマクロにより提供する.
コンストラクタとデストラクタによってロックの取得と解放を実現している
ため, 途中の return や例外による return に対してもロックの解放を
保証できる.

TJMonitor は TJSyncObject を使いやすくしたクラスであり実質的な
機能は TJSyncObject に負っている. TJSyncObject のドキュメント
を参照のこと.

基本的な使用法は以下の通り.

synchronized マクロを使用する例

```
class Xxx : virtual public TJSyncObject {
 // ....
    void method() {
      // ...
      synchronized(this) { // ここでロックを取得
        // 排他的な操作が保証される
      } // ここでロックを解放
      // ...

    }
};
```

TJMonitor をそのまま使用する例(上記と同じ意味)

```
class Xxx : virtual public TJSyncObject {
 // ....
    void method() {
      // ...
      {  TJMonitor sync(this);  // ここでロックを取得
        // 排他的な操作が保証される
      }  // ここでロックを解放
      // ...

    }
};
```

TJSyncObject の protected メソッドを直接使用する例

```
class Xxx : virtual public TJSyncObject {
 // ....
    void method() {
      // ...
      lock(); // ここでロックを取得
      // 排他的な操作が保証される
      unlock(); // ここでロックを解放
      // ...

    }
};
```

また，TJSyncObject では java にない機能として，一時的にロックを完全に外す機能がある．これは，unlock(),~JTMonitor(), および synchronized ブロックの終わりが，必ずしもロックの解放を保証できないことからサポートされた機能である．他の synchronized ブロックの中からら呼ばれた別のメソッド内に於いては，ロックの取得情報がコンテキスト依存となることになるからである．

この機能は，以下のように利用できる．

unsynchronized マクロを使用する例

```
class Xxx : virtual public TJSyncObject {
 // ....
    void method() {
      synchronized(this) { // ここではロックを取得
        // 排他的な操作が保証される.
        unsynchronized(this) {  // 一時的にロックを解放
          // ここはロックが外れていることが保証される
        }
        // 再びロックが復元される(しかし前の状況とは違う可能性あり)
      }

    }
};
```

TJMonitor をそのまま使用する例(上記と同じ意味)

```
class Xxx : virtual public TJSyncObject {
 // ....
    void method() {
      { TJMonitor sync(this) // ここでロックを取得
        // 排他的な操作が保証される.
        { TJMonitor unsync(this, TJMonitor::UNLOCK); // 一時的にロックを解放
          // ここはロックが外れていることが保証される
        }
        // 再びロックが復元される(しかし前の状況とは違う可能性あり)
      }
    }
};
```

TJSyncObject の protected メソッドを直接使用する例

```
class Xxx : virtual public TJSyncObject {
 // ....
    void method() {
      lock();  // ここではロックを取得
      // 排他的な操作が保証される.
      int lock = unlockTemporarily(); // 一時的にロックを解放
      // ここはロックが外れていることが保証される
      restoreLock(lock);
      // ... 再びロックが復元される(しかし前の状況とは違う可能性あり)
      unlock();
    }
};
```

この機能は，wait 以外にもロックを完全に解放して要求を待つような
仕組みを提供すること，また，デッドロックの回避のためのロックの順序化
(階層化)を論理的にサポートすることを目的としている．(TJSyncObject 参照)

このクラスのオブジェクトを作成する時にはモードを指定できる.
デフォルトの LOCK モードでは通常の排他ロックを獲得するが UNLOCK モードで
は逆に獲得されているロックを解放する.

この逆操作が必要になる理由は, モニタの内部で TJSyncObject::wait() 以外でロックを解放して何らかのイベントを待ち合わせしたい場合があるためである. たとえば TJQueue や外部からの入力などが該当する.
それに対して TJSyncObject::wait()は自動的にロックを解放して notify() を待ち復帰時にはロック取得状態で戻るため, ロック解放の繁雑さを避けることができる. 次の例を参照.

java と TJMonitor との表現の違いをまとめると次のようになる.
ただし synchronized マクロを使うと synchronized block はまったく同じ表現が可能だが synchronized method は残念ながらメソッド本体を synchronized block の形式にするしかない.

```
       java                              TJMonitor
-------------------------------------------------------------------
synchronized void method() {       void method() {
      ・・・                            TJMonitor monitor(syncptr);
}                                         ・・・
                                   }
                                        または
                                   void method() {
                                       synchronized(syncptr) {
                                           ・・・
                                       }
                                   }
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
synchronized(obj) {                {
      ・・・                            TJMonitor syncblock(obj);
}                                         ・・・
                                   }
                                        または
                                   synchronized(obj) {
                                          ・・・
                                   }
-------------------------------------------------------------------
```

[デッドロックに関する注意]

次の操作はデッドロックを起こさない.

◆モニタオペレーションの再帰呼出し
◆モニタオペレーションの中から同一オブジェクトの別のモニタオペレーションを呼出すこと

しかし次の操作はタイミングによってはデッドロックとなる.

◆オブジェクト A,B があり A.m1(), B.m2() がモニタオペレーションとして A.m1() から B.m2() を呼出し B.m2() から A.m1() を呼出す場合

同時に複数のロックが必要な場合の解決案を示しておく.
たとえば上記のデッドロックのケースならば

A,B を TJSyncObject のサブクラスとする.
ロックを獲得する順序を決める.

```
A.lock();
B.lock();
 ・・・
B.unlock();
A.unlock();
```

@author Susumu TOKUI
@date 1998,6/1
@see TJSyncObject

Public 操作 :

TJMonitor (obj : const TJSyncObject*, mode : enum TJMonitor::Mode = LOCK) : explicit

TJMonitor オブジェクトを作成する.

@param obj ロック用オブジェクトのポインタ
@param mode LOCK: ロックを獲得, UNLOCK: ロックを解放

~TJMonitor () :

TJMonitor オブジェクトを消去する.

LOCKモードのオブジェクトはロックを解放し, UNLOCKモードの
オブジェクトはロックを復元する.

operator bool () : bool

synchronized マクロをサポートるための変換.

@return 常に false

TJQueue

メッセージキュー.

このクラスでは、メッセージとは長さの指定されたバイトの配列の事と
考える。
メッセージを蓄えるバッファのことをキューと呼び、キューに対する操作は,

1. send() - メッセージ送信
2. receive() - メッセージ受信

の2つである.
メッセージは送信依頼の順序でキューに蓄えられる。
メッセージ受信時、キューから送信依頼の順序で読み出され、
読み出されたメッセージはキューから削除する。
キューは本クラスのオブジェクト生成時に指定された固定サイズであり
伸縮しない。
また、キューのバッファ領域はキューオブジェクト内に独自に確保しており、

メッセージ送信の依頼を受けた本オブジェクトは、メッセージをキューバッファに
メモリコピーする。受信依頼時はキューから指定されたメッセージエリアに
メモリコピーする。

よって、メッセージ送信・受信依頼側はこのキューオブジェクトとは別に
メッセージエリアの確保・解放を自身で行う必要がある。

また，メッセージの送信・受信は単純なメモリコピーであるため，
メッセージはPOD型(ISO C++ section 3.9)のオブジェクトしかサポート
しない．すなわち，memcpy() によってコピーが保証できるデータ型しか
キューを介して送ることはできない．virtual メソッドを持つ class の
オブジェクトなどは，そのポインタを送受信することで仮想的な振舞を
させることができる．

■スレッドのブロックについて
メッセージ受信時、キューにメッセージが１つも存在しない場合、
receive() はブロックする。この状態でメッセージ送信依頼があった場合、
受信でブロック中の receive() が復帰する。
尚、キューにメッセージが存在する場合、ブロックせず即復帰する。

メッセージ送信時、キューに空きが無い場合(キューのコンストラクタで
指定された最大数のメッセージがすでに蓄積されている場合)のみ
ブロックする。キューに空きがある場合、ブロックせず即復帰する。

@author Susumu TOKUI
@date 1998,6/1

Public 操作 :

TJQueue (maxmsgs : int, maxmsglen : int) : TJQueue

TJQueue オブジェクトを生成する.

@param maxmsgs 最大メッセージ数
@param maxmsglen 最大メッセージ長

send (msg : const void*, length : int) : void

メッセージを送信する.

@param msg 送信メッセージのアドレス
@param length 送信メッセージのバイト数

~TJQueue () :

TJQueue オブジェクトを消去する.

receive (msgbuf : void*, bufsize : int) : int

メッセージを受信する.

@param msgbuf 受信メッセージバッファのアドレス

@param bufsize 受信メッセージバッファのバイト数

TJRWLock

Read/Write Lock (共有・排他ロック).

通常操作が参照であり、まれに更新される共有資源のためのロックである.

【注意】
このクラスには より使い易くしたTJTransaction というクラスを
用意している. TJRWLock を直接使わず TJTransaction を使うことを
推奨する.
TJTransaction はこのクラスよりも安全性を高めている. ただし
TJRWLock の一部の機能を制限している.

資源の更新スレッドは更新モードでロックを獲得し、参照スレッドは参照モードで
ロックを獲得すること.

参照モードのロックはカウンタで管理しているため同時にN個のスレッドが
獲得可能であるが, 更新モードのロックは１個のスレッドのみ獲得できる.

更新モードのロックではスレッドを記憶するため同一スレッドに限り複数回獲得
可能としている.

更新・参照ともにロックの獲得と解放の回数が同一となることを保証すること.

更新モードのロック獲得に成功して復帰する条件は次のとおり.
他のスレッドによる更新ロック状態でなく、かつ参照ロック状態でもない、
またはカレントスレッドが更新モードでロック中

参照モードのロック獲得に成功して復帰する条件は次のとおり.
更新ロック状態でなく、かつ更新ロック要求もない
【注意】この更新ロック要求がないという条件から更新スレッドが無限に
待たされることはないはずである.

尚、上記条件に合致しない場合に、スレッドがブロックするか否かは、
ロック獲得メソッドのパラメータで指定可能である。
(beginTransactionのmilliseconds)

@author Susumu TOKUI
@date 1998,6/1

Public 属性 :

INFINITE : const long

無限に待つことを示す定数.

NONBLOCK : const long

待たないことを指定する定数.

Public 操作 :

TJRWLock () : TJRWLock

TJRWLock オブジェクトを作成する.

~TJRWLock () :

TJRWLock オブジェクトを消去する.

beginTransaction (mode : enum TJRWLock::LockMode, milliseconds : long) : bool

参照または更新モードでロックを獲得する.

@param mode　SHARED: 参照モード, EXCLUSIVE: 更新モード
@param milliseconds　最大待ち合わせミリ秒数
　　INFINITE: 無限に待つ
　　NONBLOCK: 獲得できなければエラー復帰
@return　true: ロック成功, false: ロック失敗
　　milliseconds = INFINITE の場合はロック失敗の可能性はない.

endTransaction () : void

ロックを解放する.

isWLocked () : bool

更新モードでロック中ならば true を返す.

このメソッドは厳密にはきわどい問題を含むが開発中に特定のメソッドがロックされて呼び出されることを保証するために使う.
@return ロックされていれば true

isRLocked () : bool

参照モードでロック中ならば true を返す.

このメソッドは厳密にはきわどい問題を含むが開発中に特定のメソッドがロックされて呼び出されることを保証するために使う.
@return ロックされていれば true

invariant () : bool

不変条件.

開発時に assert() の引数として利用できる.
@return オブジェクトの状態に矛盾がなければ true

TJSemaphore

セマフォクラス.

このクラスはバイナリセマフォあるいは計数型セマフォとして使える.
一般的な排他・同期の問題を解決できる. このクラスのオブジェクト
を作成する時には初期値が重要である. 後述の例を参照.

通常、一般的な排他・同期の問題を解決するためには、TJSyncObjectを
用いることを推奨する。

一部、TJSyncObject よりも TJSemaphore がふさわしい例をあげるならば
非同期出力完了の同期が考えられる.
一般的には非同期出力を使う理由は出力要求後も何か処理をしたいからで
ある. ということは同期のタイミングではすでに非同期出力が完了している
可能性があることになり TJSyncObject の wait() ではすでに notify()
されていないか慎重に判断しなければならない.
その点で TJSemaphore は take() と give() の実行順序とは無関係に同期
がとれる. したがってこの問題にはセマフォが適している.

使い方の例

単純な同期の場合

セマフォは初期値 0 とする. このオブジェクトは同期をとる
２つのスレッドが知っているものとする.

```
TJSemaphore sem(0);  // 初期値 0 が重要
```

待つスレッドは

```
sem.take();
```

を実行する. 合図するスレッドは

```
sem.give();
```

を実行することにより待っているスレッドが目覚める.
このケースでは合図するスレッドが先に give() を実行しても
問題は起こらない. 単に take() がブロックされずに復帰する
だけである. TJSyncObject の wait() 呼び出しは notify()
されることをプログラマが保証しなければならない. 通常それ
は１個の状態変数で容易に実現できる. TJSemaphore はその状
態変数を内部に持っているため give() が先に走ったことを
take() が知ることができると考えると理解しやすい.

排他制御の場合

この場合は初期値を 1 とすることが重要である.

```
TJSemaphore sem(1);  // 初期値 1 が重要
```

競合するスレッドは次の方法で critical region を実現できる.

```
sem.take();
// critical region
// ここで共有資源を操作する.
sem.give();
```

@author Susumu TOKUI
@date 1998,6/1

Public 操作 :

TJSemaphore (inivalue : int = 0) : TJSemaphore

TJSemaphore を作成する.

@param inivalue 初期値

take () : void

Semaphore を取得する.
資源が不足している場合はブロックされる.

take (milliseconds : long) : bool

タイムアウトを指定して Semaphore を取得する.
資源が不足している場合はブロックされる.
@param milliseconds ミリ秒
@return true: 取得できた false: タイムアウト

~TJSemaphore () :

TJSemaphore を消去する.

give () : void

Semaphore を与える.
もし待っているスレッドがあればレディにする.

TJTransaction

Read/Write Lock(共有・排他ロック)の開始と終了を扱うトランザクション.

共有資源を安全に更新・参照するためのロック機構 TJRWLock の開始と
終了を，コンストラクタとデストラクタで扱う.
ただし,ロック取得は無限に待つモードのみサポートする.

コンストラクタで指定モードのロックが獲得されデストラクタで解放される.
スコープを脱出する時に自動的にロックを解放したい場合に便利である.

特に，例外抜け，途中 return 等で unlock の呼び出しを保証したい場合に
は，このクラスを使用すべきである.

以下に使用例を示す. 関数の例であるが，任意のスコープ終了で取得した
ロックと解放を保証できる(for, while, if, {} ブロックも可).

```
void f(TJRWLock* lock) {
    // 関数先頭でロック確保
    TJTransaction tr(lock, TRWRLock::SHARED);

    if (....)
        return;  // ここでもロック解放を保証.

    int* BigArray = new int[1000];
    // 途中，例外によって関数を抜けてもロック解放.

} // 関数終了でロック解放
```

@author Susumu TOKUI
@date 1998,6/1

Public 操作 :

TJTransaction (rwlock : TJRWLock*, mode : enum TJRWLock::LockMode) : TJTransaction

参照または更新モードでロックを獲得する.

TJRWLockの機能のうちタイムアウト指定や trylock (NONBLOCK) に
相当する機能は除外してあることに注意.

引数で与えられた RWLock を lock する.

@param rwlock TJRWLockオブジェクトのポインタ
@param mode TJRWLock::SHARED: 参照モード
TJRWLock::EXCLUSIVE: 更新モード

~TJTransaction () :

デストラクタ(ロックを解放する).

コンストラクタで取得した RWLock を unlock する.

Util

TJEventListener

イベントリスナ.

各種イベントのリスナ(受信者)の実装を支援するためのテンプレートであり，
イベントを型引数としている．EventSource と組で，Java のイベントモデル
java.util.EventObject / java.util.EventListener をサポートする．

C++ のテンプレートとして Event を扱うため，メソッドの多重定義を利用して，
すべてのイベント処理メソッドは，eventHappend という名前になる．

イベント処理メソッド eventHappened() は pure virtual でありインタフェース
のみの規定．アプリケーションで実装するコールバックである．

使用例は，TJEventSource を見よ．

[ロック方針]

このクラスは，マルチスレッドに関するロックを全く意識しない．
必要であれば TJSafeEventListener を見よ．

eventHappened() はスレッドの設計にも依存するが，一般には非同期呼出しの対策
が必要である(たとえば UNIX のシグナルハンドラを連想せよ．)
このクラスまたはサブクラスのオブジェクトでマルチスレッド環境でも安全にオブ
ジェクト操作したければ TJSafeEventListener と TJSafeEventSource を使うこと．

@author tokui
@date 1999,3/31
@see TJSafeEventListener
@see TJEventSource

Public 操作 :

~TJEventListener () :

デストラクタ.

isSafe () : bool

スレッドセーフなものかどうかの検査.

このクラス自身では false である．
EventSource が fireEvent を行う際，このメソッド
の返り値に応じて eventHappend/safeEvnetHappend を呼び分ける．

@return true スレッドセーフ

eventHappened (ev : const EventType&) : void
イベントが起こった時のコールバック.

@param ev イベントオブジェクト

TJEventSource

イベントソース.

各種イベントのソース(発生源)の実装を支援するためのテンプレートであり，
イベントを型引数としている．EventListener と組で Java のイベントモデル
java.util.EventObject / java.util.EventListener をサポートする．

C++ では多重継承を用いることができるため，イベントソース側
にも継承できるサポートクラスとしてこのクラスを用意した．
(java.beans.PropertyChangesSupportを参照)

登録されたリスナへの通知の順序は規定されていない．

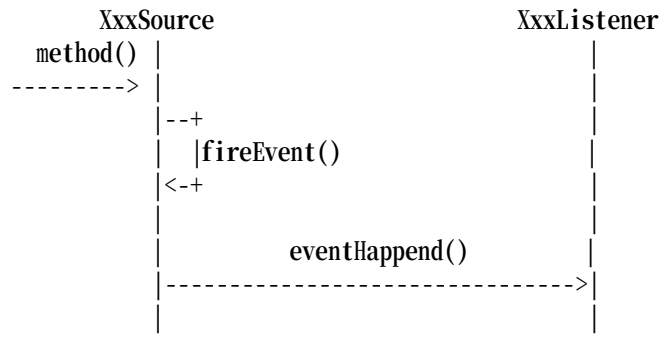
登録された Listener の delete(デストラクタ呼び出し) については，
基本的にはアプリケーションの責任とするが，明示的に destroyListeners を
呼ぶことで全 Listener に対して delete を呼び出すことができる．

Event, Listener, Source の使用例を示す．

```
// アプリケーション定義のイベント型
class XxxEvent {
    TJEventSource<XxxEvent>* source;
    int value;
public:
    XxxEvent(TJEventSource<XxxEvent>* s, int v):
       source(s), value(v) { }
    TJEventSource<XxxEvent>* getSource() const {
        return source;
    }
    int getValue() const { return value; }
};

// アプリケーション定義のイベントリスナ
#include "TJEventListener.h"
class XxxListener : public TJEventListener<XxxEvent> {
   public:
      virtual void eventHappened(const XxxEvent& ev) {
        cout << "ev.value=" << ev.getValue() << endl;
      // マルチスレッド環境では this オブジェクトの操作は危険
      }
};

// アプリケーション定義のイベントソース
```

```
#include "TJEventSource.h"
class XxxSource : public TJEventSource<XxxEvent> {
   public:
      void foo() {
         // ........
         // イベント発生
         XxxEvent ev(this, 2);
         fireEvent(ev);  // eventHappened を呼び出す
      }
};

// ソースとリスナの連結
XxxSource source;
XxxListener listener;
source.addListener(&listener);
```

Listener, Source の呼び出し関係は以下のようになる.



[ロック方針]

このクラスは，マルチスレッドに関するロックを全く意識しない.
必要であれば TJEventSource を見よ.

@author tokui, hiranabe
@date 1998,7/31
@date 1999,4/1
@see TJSafeEventSource
@see TJEventListener

Public 操作 :

~TJEventSource () :

デストラクタ.
Listener のデストラクタは呼び出さないことに注意.

@see TJEventSource#destroyEventListeners()

addEventListener (l : TJEventSource::ListenerType*) : void

イベントリスナを登録する.

@param l 登録するリスナ．null なら無視

removeEventListener (l : TJEventSource::ListenerType*) : void

イベントリスナを削除する.

@param l 削除するリスナ．null なら無視

removeEventListeners () : void

登録されているすべての Listener を削除する.
このメソッドが呼ばれた後，通知すべき Listener のリストは空になる．
各 Listener の delete (デストラクタ呼び出し)は行わない．

hasEventListener (l : const TJEventSource::ListenerType*) : bool

Listner を持っているかどうかの検査.

@param l Listener オブジェクト

destroyEventListeners () : void

登録されている Listener を delete (デストラクタ呼び出し)する.

基本的に Listener の delete はアプリケーションの責任とするが，
アプリケーションが Listener 列の管理をしたくない場合，このメソッド
が delete を支援する．
Listener が自動変数の場合は，当然これを使用してはならない．

countEventListeners () : unsigned int

現在の Listener の数を返す.

@return 現在の Listener の数

TJObservable

オブザーバーパターンを支援する Observable クラス(Subject/Publisher側).

Observable は Observer へ自身の変化の通知する．
Model-View で Model 側を演じるオブジェクトがこのクラスを継承する．

1つの Observable は複数の Observer を持つことができる．Observable
の内部状態が変化したとき，アプリケーションは notifyObservers メソッドを
よぶことによって，Observer の update を間接的に起動する．

通知の順序は規定されていない．

登録された Observer の delete(デストラクタ呼び出し) については，

基本的にはアプリケーションの責任とするが，明示的に destroyObservers を
呼ぶことで全 Observer に対して delete を呼び出すことができる．

[ロック方針]

このクラスは，マルチスレッドに関するロックを全く意識しない．
必要であれば TJSafeObservable を見よ．

@author hiranabe
@date 1999,3/30
@see TJObserver
@see TJSafeObservable

Public 操作 :

addObserver (o : TJObserver*) : void

Observer を追加する.

@param o Observer オブジェクト．null なら無視.

TJObservable () : TJObservable

コンストラクタ.

removeObserver (o : TJObserver*) : void

Observer を削除する.

Observer の delete (デストラクタ呼び出し)は行わない.

[note]
java.util.Observable の deleteObserver というネーミングは
C++ の delete 演算子 を連想させる名前なので，removeObserver とした.

@param o Observer オブジェクト

removeObservers () : void

登録されているすべての Observer を削除する.
このメソッドが呼ばれた後，通知すべき Observer のリストは空になる.
各 Observer の delete (デストラクタ呼び出し)は行わない.

notifyObservers () : void

Observable の変化を Observer に通知する.
変化は setChanged で設定でき，hasChanged によって検知され，clearChanged
によってリセットすることができる.
このメソッドは，observer の update を呼び出す.
その際の第2引数は，0(NULL) となる.

const メソッドではないのは意図的である．このメソッドの中で呼ばれる
Observer 側の update で，この Observable の non-const メソッドを
呼べるようにしている.

@see TJObservable#clearChanged()
@see TJObservable#hasChanged()
@see TJObserver#update()

notifyObservers (arg : void*) : void

Observable の変化を Observer に通知する.

変化は setChanged で設定でき，hasChanged によって検知され，clearChanged によってリセットすることができる.
このメソッドは，observer の update を呼び出す.
その際の第2引数は，このメソッドに渡される arg である.

const メソッドではないのは意図的である．このメソッドの中で呼ばれる Observer 側の update で，この Observable の non-const メソッドを呼べるようにしている.

@param arg アプリケーションで使用する任意のデータ
@see TJObservable#clearChanged()
@see TJObservable#hasChanged()
@see TJObserver#update()

hasChanged () : bool

変化したかどうかを検査する.

@return true setChanged が呼ばれ,まだ通知されていない状態
@see TJObservable#clearChanged()
@see TJObservable#setChanged()

countObservers () : unsigned int

現在の Observer の数を返す.

@return 現在の Observer の数

destroyObservers () : void

登録されている Observer を delete (デストラクタ呼び出し)する.

基本的に Observer の delete はアプリケーションの責任とするが,
アプリケーションがObserver 列の管理をしたくない場合，このメソッドが delete を支援する.
Observer が自動変数の場合は，当然これを使用してはならない.

isSafe () : bool

スレッドセーフなものかどうかの検査.

このクラス自身では false である.
Observer が notifyObservers を行う際，このメソッドの返り値に応じて update/safeUpdate を呼び分ける.

@return true スレッドセーフ

~TJObservable () :

デストラクタ.

Observer のデストラクタは呼び出さないことに注意.

@see TJObservable#destroyObservers()

hasObserver (o : const TJObserver*) : bool

Observer を持っているかどうかの検査.

@param o Oberver オブジェクト

TJObserver

オブザーバーパターンを支援する Observer クラス(Observer/Subscriber 側).

Observer は Observable の変化の通知を受けるクラス.
Model-View で View 側を演じるオブジェクトがこのクラスを継承する.

[ロック方針]

このクラスは，マルチスレッドに関するロックを全く意識しない.
必要であれば TJSafeObserver を見よ.

@author hiranabe
@date 1999,3/30
@see TJObservable
@see TJSafeObserver

Public 操作 :

update (o : TJObservable&, arg : void*) : void

Observable オブジェクトが変化したときにこのメソッドが呼び出される.

アプリケーションは，Observable オブジェクトの notifyObservers メソッド
をよびだしてオブジェクトの状態変化を Observer に伝える.

@param o Observable オブジェクト
@param arg Observable の notifyObservers メソッドに渡される引数.
アプリケーションで自由に使用する.

isSafe () : bool

スレッドセーフなものかどうかの検査.

このクラスそのものでは false.

@return true スレッドセーフ

~TJObserver () :

デストラクタ.

TJSafeEventListener

スレッドセーフなイベントリスナ.

TJSafeEventSource と TJSafeEventListener によるイベント伝達機構
はマルチスレッド環境における利用を想定しその安全性を高めてある.

使用例は，TJSafeEventSource を見よ.

[ロック方針]

重要な注意事項として eventHappened() はそのオブジェクトのロックを獲得して
呼び出され復帰後にロックが解放される.
したがって eventHappened() の実行中はマルチスレッド環境でもオブジェクトの
操作は安全である.

@author tokui, hiranabe
@date 1998,7/31
@date 1999,4/1
@see TJSafeEventSource
@see TJEventListener

派生元 : TJEventListener, TJSyncObject

Public 操作 :

~TJSafeEventListener () :

デストラクタ.

isSafe () : bool

スレッドセーフなものかどうかの検査.

このクラスは，true を返す.

@return true スレッドセーフ

safeEventHappened (ev : const EventType&) : void

イベントが起こった時のコールバック.

@param ev イベントオブジェクト

TJSafeEventSource

スレッドセーフなイベントソース.

各種イベントのソース(発生源)の実装を支援するためのテンプレート.

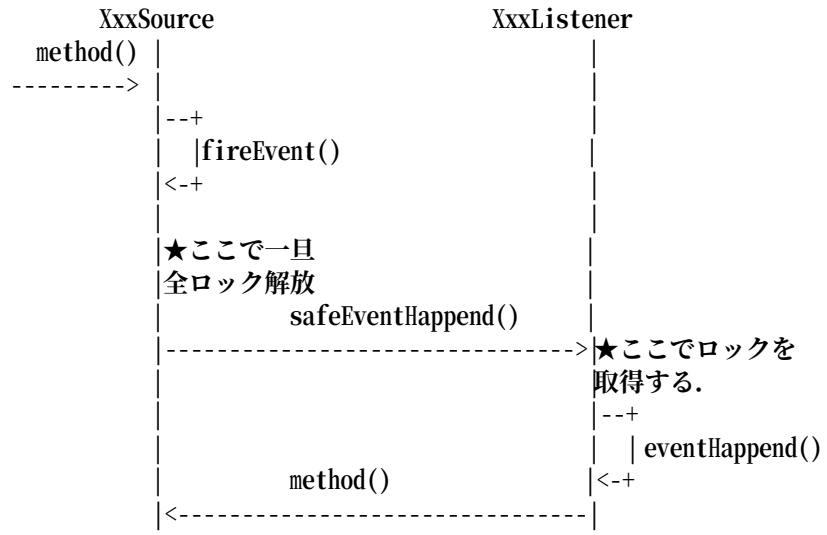
TJSafeEventSource と TJSafeEventListener によるイベント伝達機構
はマルチスレッド環境における利用を想定しその安全性を高めてある.

Event, SafeListener, SafeSource の使用例を示す.

```
// アプリケーション定義のイベント型
class XxxEvent {
    TJSafeEventSource<XxxEvent>* source;
    int value;
public:
    XxxEvent(TJSafeEventSource<XxxEvent>* s, int v):
      source(s), value(v) {  }
    TJSafeEventSource<XxxEvent>* getSource() const {
      return source;
    }
    int getValue() const { return value; }
};

// アプリケーション定義のイベントリスナ
#include "TJSafeEventListener.h"
class XxxListener : public TJSafeEventListener<XxxEvent> {
public:
    virtual void eventHappened(const XxxEvent& ev) {
      TJSafeEventSource<XxxEvent>* source = ev.getSource();
      cout << "ev.value=" << ev.getValue() << endl;
      // this オブジェクトの操作は安全
      // ....
     }
};
// アプリケーション定義のイベントソース
#include "TJEventSource.h"
class XxxSource : public TJSafeEventSource<XxxEvent> {
    public:
      void foo() {
        // ........
        // イベント発生
        XxxEvent ev(this, 2);
        fireEvent(ev);  // safeEventHappened を呼び出す
                    // safeEventHappened は eventHappened を呼ぶ
      }
};

// ソースとリスナの連結
```

```
XxxSource source;
XxxListener listener;
source.addListener(&listener);
```

Listener, Source の呼び出し関係は以下のようになる.



[ロック方針]

一般に，SafeEventSource を継承する XxxSource と SafeEventListenerを継承する XxxListener 間では，XxxListener -> XxxSource という方向の呼び出しが自然に起こる可能性がある．しかし，eventHappened() の呼び出しは XxxSource から XxxListenerへのコールバックとなる.

アプリケーションは，マルチスレッドのデッドロック問題を回避するために次の原則を守ること.

(1) ソースからリスナへのイベント通知ではソース側は一切のロックを取らずに行う.

(2) リスナからソースに対する呼び出しはソース内部で自動的にロックする.

このクラスのイベント通知処理 fireEvent() は上記(1)の原則に従っている．さらに上記の原則(2)を守るためサブクラスのメソッドをモニタオペレーション(synchronized)とする場合は,

```
int XxxSource::method() {
    TJMonitor synchro__(this);  // 名前 synchro__ は任意
     ...
}
```

とすればよい.

【fireEvent() と fireEventRetainingLock() の違い】

(1) fireEvent() の場合

このイベントソースがロックされていれば解放してイベント通知を実行し終了後に自動的にロックを復元する.

●利点

一般的なロック階層とは逆方向のロックを獲得しないためデッドロックの危険性は少なくなる.

▲欠点

もしロック中に fireEvent() を呼出したとすると一時的にロックが解放されるため fireEvent() から復帰した時このイベントソースの状態が変更されている可能性がある.
これが不都合でかつ fireEvent() を選びたい場合は fireEvent() 呼出し前にスレッド資源にコピーしておく方法がある.

(2) fireEventRetainingLock() の場合

このイベントソースがロックされている場合でも解放せずにイベント通知を実行する.

●利点

もしこのイベントソースがロックされている場合 fireEventRetainingLock() 呼出しの前後でソースオブジェクトの状態は保存される.

▲欠点

一般的なロック階層とは逆方向のロックを獲得する可能性が高いためデッドロックの危険性が大きい.
fireEventRetainingLock() は内部的に既知のコードを呼出す場合でかつデッドロックの危険性がないことを保証できる場合のみとすべきである.

派生元 : TJEventSource, TJSyncObject

Public 操作 :

~TJSafeEventSource () :

デストラクタ.
Listener のデストラクタは呼び出さないことに注意.

@see TJSafeEventSource#destroyObservers()

addEventListener (l : TJEventSource::ListenerType*) : void

イベントリスナを登録する.

@param l 登録するリスナ．null なら無視

removeEventListener (l : TJEventSource::ListenerType*) : void

イベントリスナを削除する．

@param l 削除するリスナ．null なら無視

removeEventListeners () : void

登録されているすべての Listener を削除する.
このメソッドが呼ばれた後，通知すべき Listener のリストは空になる．
各 Listener の delete (デストラクタ呼び出し)は行わない．

hasEventListener (l : const TJEventSource::ListenerType*) : bool

Listner を持っているかどうかの検査.

@param l Listener オブジェクト

destroyEventListeners () : void

登録されている Listener を delete (デストラクタ呼び出し)する.

基本的に Listener の delete はアプリケーションの責任とするが，
アプリケーションが Listener 列の管理をしたくない場合，このメソッド
が delete を支援する．
Listener が自動変数の場合は，当然これを使用してはならない．

countEventListeners () : unsigned int

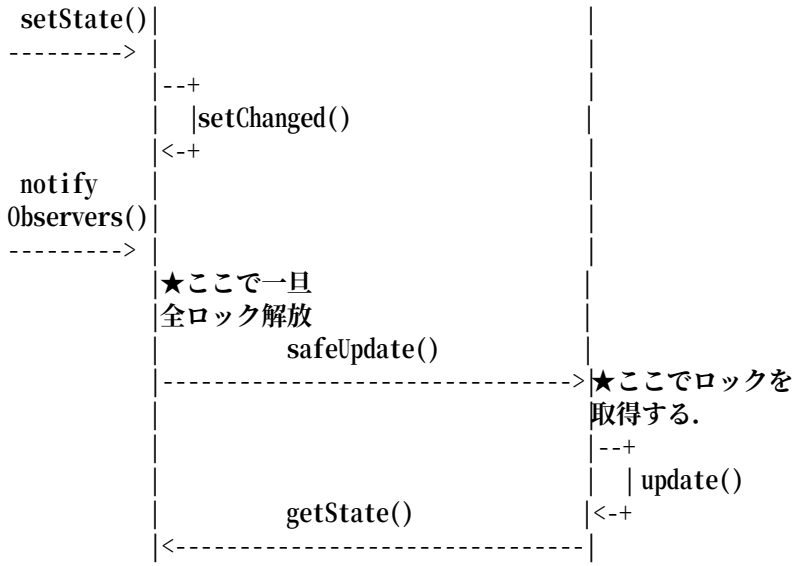
現在の Listener の数を返す.

@return 現在の Listener の数

TJSafeObservable

スレッドセーフな Observable.

[ロック方針]
一般に，Observer を継承する ConcreteObserver と，Observable を継承する
ConcreteObservable 間では，ConcreteObsever -> ConcreteObservable という
方向の呼び出しが自然に起こる．しかし，update() の呼び出しは，
ConcreteObservable から ConcreteObserver へのコールバックとなる．



notifyObservers() では，update() の前に現在取得している一切のロックを解放することにする．これによって，デッドロックの可能性を回避する．アプリケーションは，ConcreteObserver -> ConcreteObservable の呼び出しを正順として，ロックの取得もこの順序とすることが望まれる．

@author hiranabe
@date 1999,3/30
@see TJSafeObserver
@see TJObservable
@see TJSafeEventSource

派生元 : TJObservable, TJSyncObject

Public 操作 :

addObserver (o : TJObserver*) : void

Observer を追加する.

@param o Observer オブジェクト．null なら無視.

TJSafeObservable () : TJSafeObservable

コンストラクタ.

removeObserver (o : TJObserver*) : void

Observer を削除する.

@param o Observer オブジェクト

removeObservers () : void

登録されているすべての Observer を削除する.

notifyObservers () : void

Observable の変化を Observer に通知する.

Observer の update 呼び出しの前に，ロックを解放することに注意.

notifyObservers (arg : void*) : void

Observable の変化を Observer に通知する.

Observer の update 呼び出しの前に，ロックを解放することに注意.

@param arg アプリケーションで使用する任意のデータ

notifyObserversRetainingLock () : void

Observable の変化を Observer に(ロック保持のまま)通知する.

Observer の update 呼び出しの前に，ロックを解放を行わないため，デッドロックの可能性を含むことに注意.

notifyObserversRetainingLock (arg : void*) : void

Observable の変化を Observer に(ロック保持のまま)通知する.

Observer の update 呼び出しの前に，ロックを解放を行わないため，デッドロックの可能性を含むことに注意.

@param arg アプリケーションで使用する任意のデータ

hasChanged () : bool

変化したかどうかを検査する.

@return true setChanged が呼ばれまだ通知されていない状態

countObservers () : unsigned int

現在の Observer の数を返す.

@return 現在の Observer の数

destroyObservers () : void

登録されている Observer を delete (デストラクタ呼び出し)する.

isSafe () : bool

スレッドセーフなものかどうかの検査.

このクラスでは，true となる.
@return true スレッドセーフ

hasObserver (o : const TJObserver*) : bool

Observer を持っているかどうかの検査.

@param o Oberver オブジェクト

~TJSafeObservable () :

デストラクタ.

Observer のデストラクタは呼び出さないことに注意.
@see TJSafeObservable#destroyObservers()

TJSafeObserver

スレッドセーフな Observer.

Observable からの update 呼び出しに対して,
自動的にロックを取得する. TJSafeObservable に詳細なコメントがある.

@author hiranabe
@date 1999,3/30
@see TJSafeObservable
@see TJObserver

派生元 : TJObserver, TJSyncObject

Public 操作 :

update (o : TJObservable&, arg : void*) : void

Observable オブジェクトが変化したときにこのメソッドが呼び出される.
実際には, safeUpdate 経由で呼び出されるため, 自動的にロックが
取得されてこのメソッドに入る.

このメソッドを override して, アプリケーションの動作を記述する.
@param o Observable オブジェクト
@param arg Observable の notifyObservers メソッドに渡される引数.
アプリケーションで自由に使用する.

isSafe () : bool

スレッドセーフなものかどうかの検査.

このクラスは, true を返す.

@return true スレッドセーフ

~TJSafeObserver () :

デストラクタ.

合計 :

5 カテゴリ
18 クラス

カテゴリ構造

論理ﾋﾞｭｰ
- MT
- MT-Support
- Util