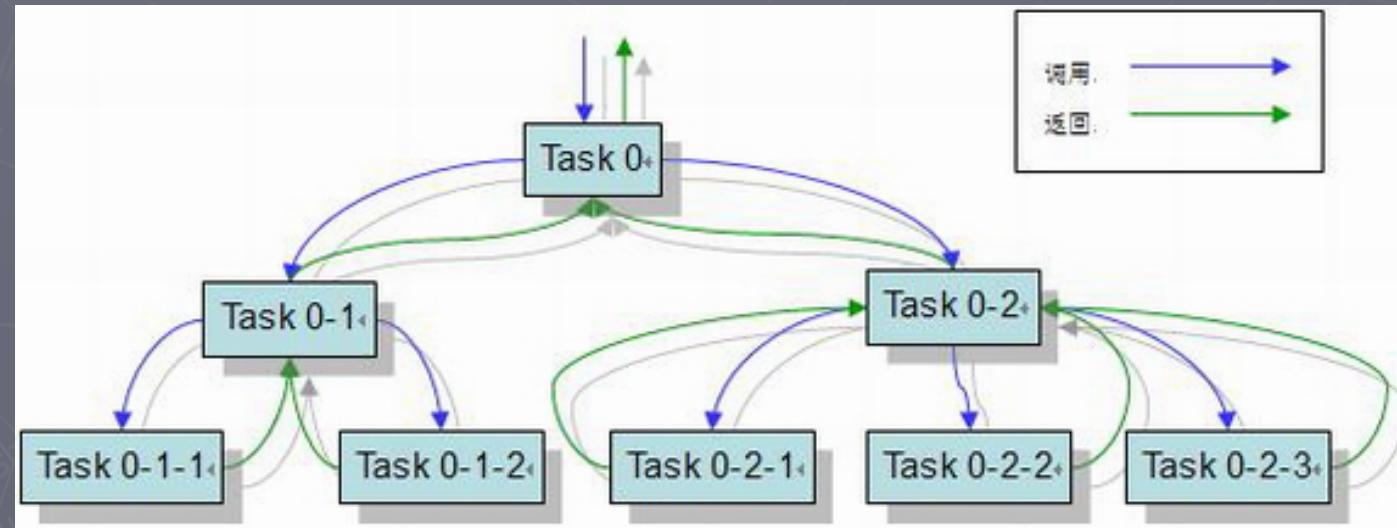


# Java7: fork/join framework & closures

hongjiang 2010-1-8

# Fork/Join Framework

Divide and conquer algorithm

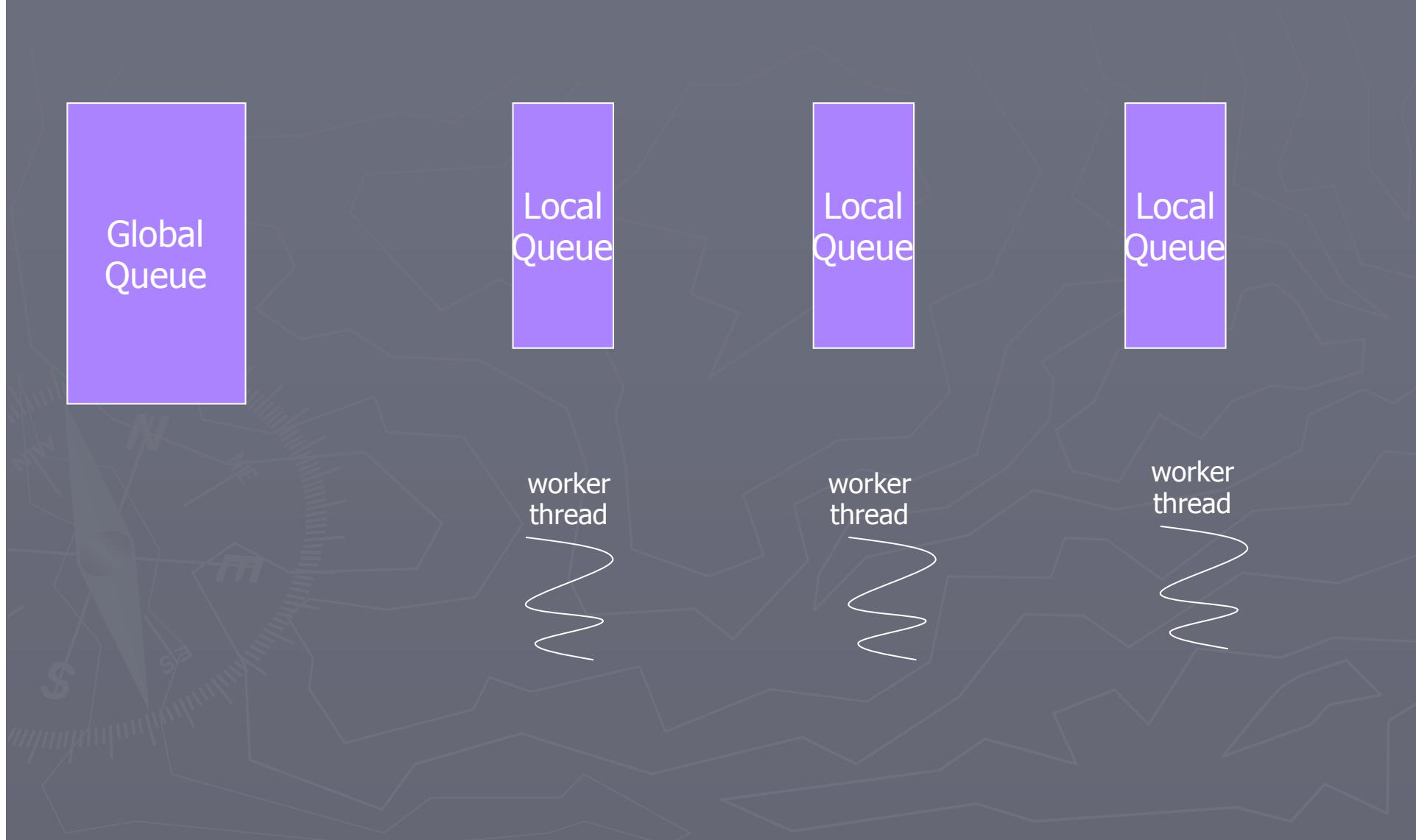


# Fork/Join Framework

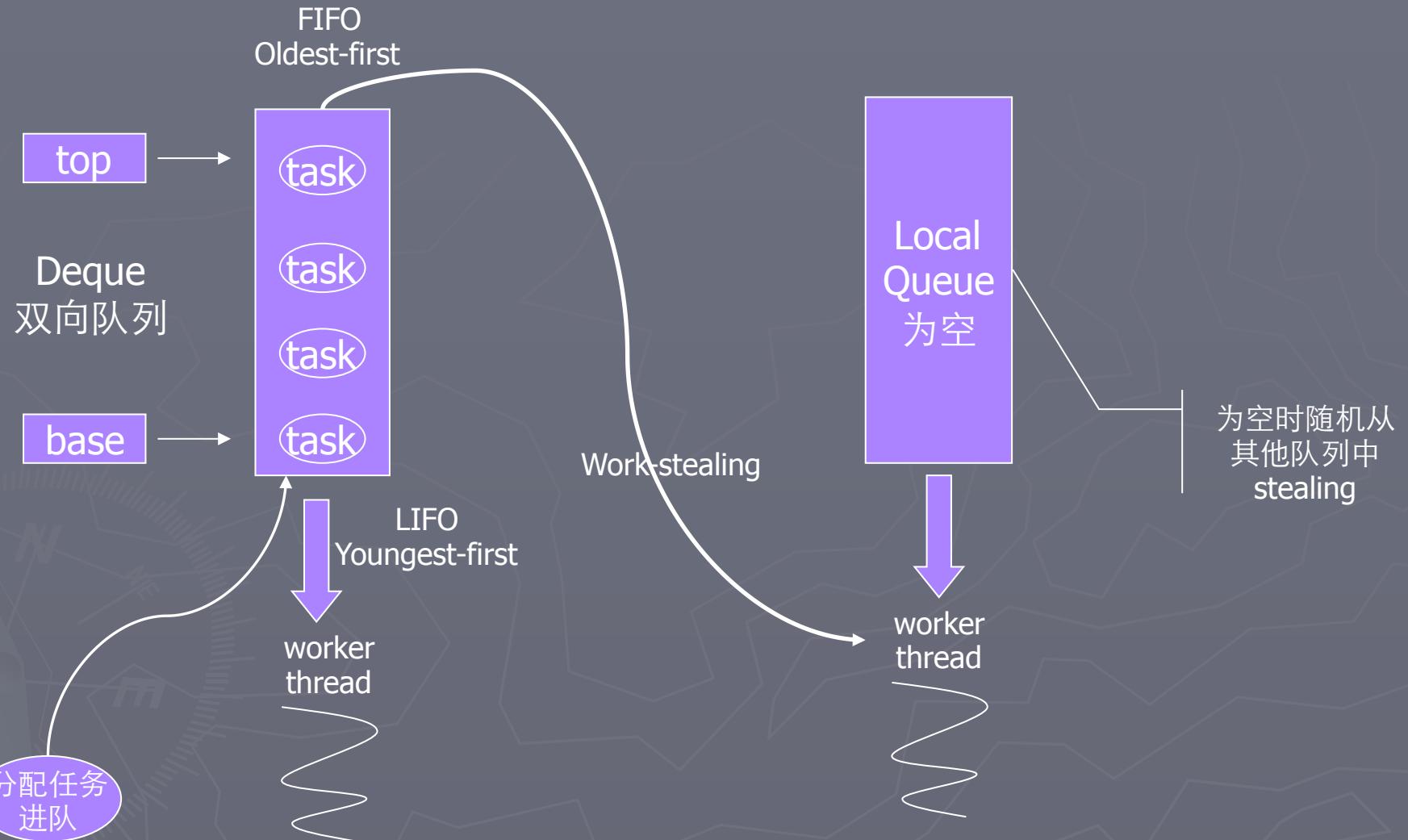
实现方式：work-stealing  
framework（工作窃取）的一个变种  
该思想源于cilk语言

《Java并发编程》作者在10年前就已经  
提出，但知道现在才纳入Java7

# work-stealing (工作窃取) 的实现



# work-stealing (工作窃取) 的实现



# work-stealing (工作窃取) 的实现

- 1: push和pop只能被queue所有者线程调用
- 2: take操作限制为每次只被一个stealing-thread使用(via an entry lock on take)
- 3: 当dequeue为空时， pop和take方法是有害的(only interfere)
- 4: worker线程任务队列为空，并且偷不到其他队列任务时后退(back off)，过会儿再重试，直到全部的worker线程都同样空闲(via yields,sleep, and/or priority adjustment)

# Why closures ?

Java7中增加闭包的原因：  
简化行为的抽象，并行(**parallel**)编程的需要。

Bulk-data APIs such as parallel arrays allow computations to be expressed in terms of higher-level, SQL-like operations (e.g., filter, map, and reduce) which can be mapped automatically onto the fork-join paradigm.

Working with parallel arrays in Java, unfortunately, requires  
lots of boilerplate code to solve even simple problems.  
Closures can eliminate that boilerplate.

Closures make API of that kind palatable by hiding the 80 odd interfaces

详见：

<http://blogs.sun.com/mr/entry/closures>

[http://www.jroller.com/scolebourne/entry/closures\\_in\\_jdk\\_7](http://www.jroller.com/scolebourne/entry/closures_in_jdk_7)

Jsr166z:

<http://www.javac.info/jsr166z/jsr166z/forkjoin/package-summary.html>

# Java7中闭包的几种提案：

## 1) **BGGA**(源自Bracha、Gafter、Gosling 和 Ahe的首字母)

这是最雄心勃勃的一个提案。

它引入了一个新的语言类型：函数类型，以及它的一个全新的标记方法(notation )

## 2) **CICE** (简明实例创建方式，Concise Instance Creation Expressions)

这是三个提案中最简单的一个。

基本上，它只起到在语法上不断定义匿名类的作用而已，没有其他的用处。

没有第一类函数 (**first class functions**) ! 没有晦涩难懂的函数编程理论

## 3) **FCM**(第一类方法，First Class Methods)

这是上面两个方案的折衷：它引入了第一类函数，

但其闭包返回语句中并没有包含非局部绑定，而且去掉了lambda形式符号

# 闭包的本质：代码块+上下文

关于引用环境的绑定(The Binding of Referencing Environments),  
先通过一个java的匿名内部类来看：

# Java中的匿名内部类如何访问局部变量

```
public Thread createThread(){
    // 提升局部变量的生命周期
    final int innerVar = 100;

    return new Thread(){
        public void run(){
            System.out.println(innerVar);
        }
    };
}
```

innerVar 还是分配在栈空间上么？

Java的匿名内部类，和闭包很像。但用匿名内部类来实现，前提是先要定义好该行为的接口。繁琐一些，不那么灵活

# 一个javascript 的例子

```
var div = document.getElementById("testDiv");
var events = {onclick: "clicked",
              onchange: "changed",
              onmouseover: "mouse over"};
```

```
for(e in events){
    div[e] = function(){
        alert(events[e]);
    };
}
```

# 解决方式：多一层抽象

```
for(e in events){  
    div[e] = function(e){  
        return function(){  
            alert(events[e]);  
        };  
    }(e);  
}
```

每次绑定不同的局部对象。  
多加的这层函数叫做因子函数(factor function)

动态语言都有此问题， ruby的见：  
<http://www.javaeye.com/topic/156337>

# 更深入的了解

## The Binding of Referencing Environments (引用环境的约束)

在递归的情况下会是怎样的？

# 闭包的早绑定和晚绑定(深约束/浅约束)

```
program binding_example(input, output);

procedure A(I : integer; procedure P);

    procedure B; // 子函数B
    begin
        writeln(I);
    end;

    begin (* A *)
        if I > 1 then
            P
        else
            A(2,B); // 递归
    end;

    procedure C; begin end;

begin (* main *)
    A(1, C);
end
```

Pascal里的深约束。在通过形式参数**P**调用**B**时，存在着**I**的两个实例。由于**P**的闭包是在**A**的第一个调用中创建的，因此它使用该调用时的那个**I**，因此打印出**1**。

---

以上摘自《程序设计语言——实践之路》第二版，引用环境的约束一节。

我让一个朋友把上面的代码翻译为C#，看看是不是一样，下面是他的回复

- ▶ 确实C#也是这样。不过用C#的语法写出来的代码，看上去结果比Pascal要明显一些，我觉得。

```
static void A(int i, Action p)
{
    if (i > 1)
        p();
    else
        A(2, () => Console.WriteLine(i));
}
```

- ▶ C#不允许函数嵌套，只允许函数嵌套closure，当然也可以写成：

```
static void B(int i)
{
    Console.WriteLine(i);
}

static void A(int i, Action p)
{
    if (i > 1)
        p();
    else
        A(2, () => B(i));
}
```

- ▶ 结果也没有差别，其实前面那种写法Console.WriteLine就是B。这两种写法看上去结果都是很明显的，调用WriteLine的那个i只能是1。

Pascal的closure依赖的是帧（一个函数调用发生时的栈信息）指针的传递，所有变量都还是存在于栈上；而C#是靠把closure中用到的变量包装进一个（匿名）类，closure本身则是该类的一个方法。

Pascal的基于帧的实现是所有试图实现closure而又没有自动垃圾回收机制的语言的无奈之举，这种方法不仅看上去比较费解，而且在应用上也有限制——用帧实现的closure难以在多线程环境中使用，因为closure中的变量存在于栈上，closure能否得到执行完全取决于构造closure的那个函数是否已经返回，也就是说，构造closure的函数必须等待，知道closure执行完毕才能返回。

比如C#中closure经常被用于执行一些异步调用，如果是基于帧的closure在这些方面就很难得到有效应用了。

# 参考：

JDK 7 中的 **Fork/Join** 模式

<http://www.ibm.com/developerworks/cn/java/j-lo-forkjoin/index.html>

New and Improved CLR 4 Thread Pool Engine

<http://www.danielmoth.com/Blog/2008/11/new-and-improved-clr-4-thread-pool.html>

**The Cilk language and work-stealing scheduler**

[http://www.usenix.org/publications/library/proceedings/ana97/full\\_papers/blumofe/blumofe\\_html/node2.html](http://www.usenix.org/publications/library/proceedings/ana97/full_papers/blumofe/blumofe_html/node2.html)

函数式编程另类指南：

<http://erlang-china.org/study/yet-another-pf-guide.html>

《程序设计语言——实践之路》第二版

《JAVASCRIPT语言精髓与编程实践》

更多关于fork/join和closures的讨论，见我的delicious书签：

<http://delicious.com/w.hongjiang/fork-join>

<http://delicious.com/w.hongjiang/closures>