Rust并行编译的挑战与突破

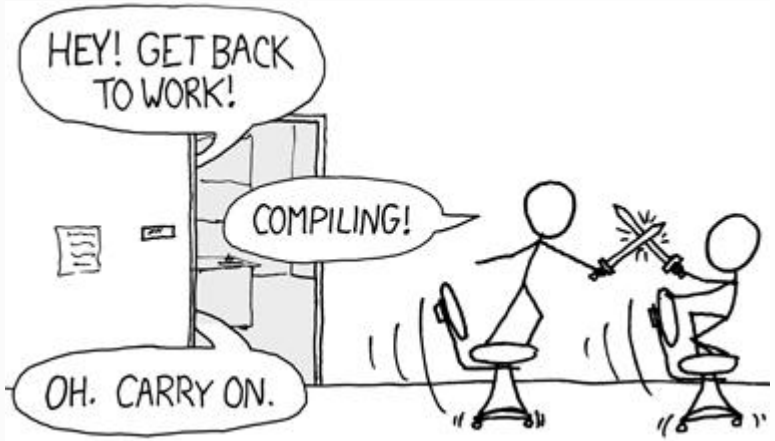
李原 2022年5月28日

目录

相关浅谈

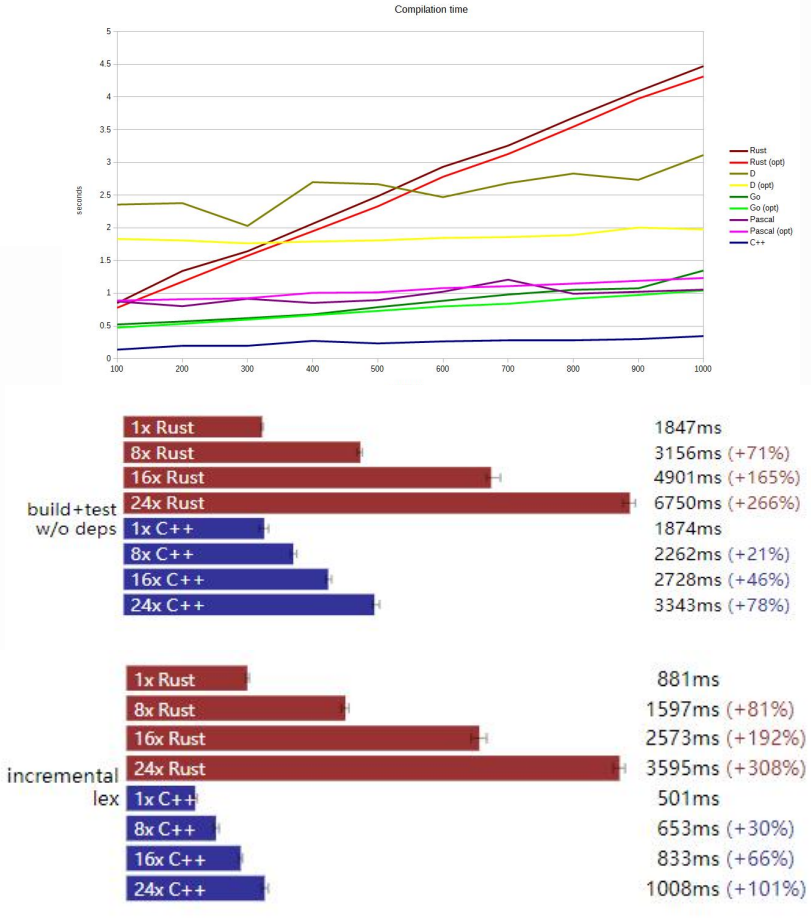
Rust编译速度之殇

编译器设计造成编译速度缓慢



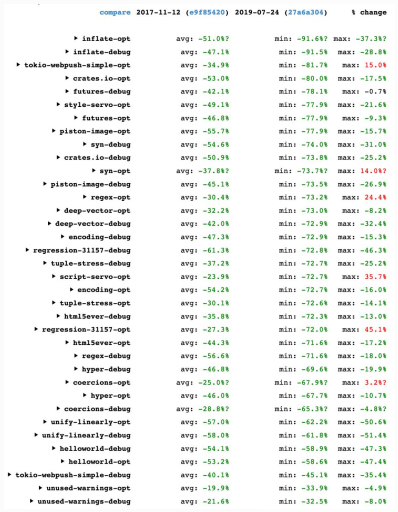
- 单态化
- 借用检查
- 宏展开
- MIR优化
- ...

Rust规模编译速度慢于C++



Rust编译速度之殇

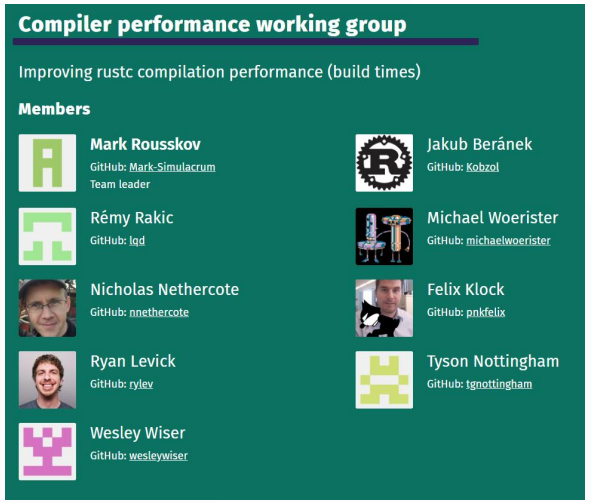
提升编译效率成为近年社区重点工作



compare 2017-11-12 (e9f85420) 2019-07-24 (27a6a306) % change

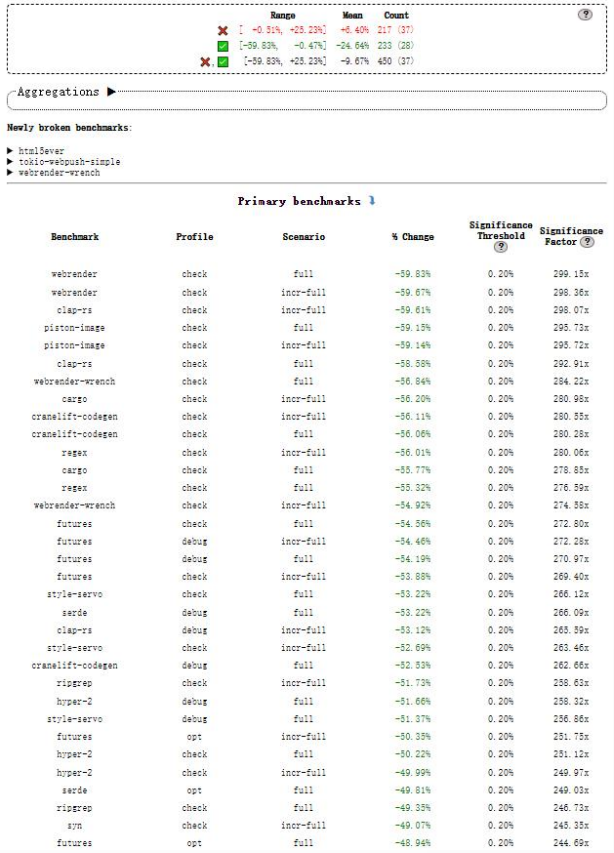
inflate-opt	avg: -51.0%	min: -91.6% max: -37.3%
inflate-debug	avg: -47.1%	min: -91.5% max: -28.8%
tokio-webpush-simple-opt	avg: -34.9%	min: -81.7% max: 15.0%
crates.io-opt	avg: -53.0%	min: -80.0% max: -17.5%
futures-debug	avg: -42.1%	min: -78.1% max: -0.7%
style-servo-opt	avg: -49.1%	min: -77.9% max: -21.6%
futures-opt	avg: -46.8%	min: -77.9% max: -9.3%
piston-image-opt	avg: -55.7%	min: -77.9% max: -15.7%
syn-debug	avg: -54.6%	min: -74.0% max: -31.0%
crates.io-debug	avg: -50.9%	min: -73.8% max: -25.2%
syn-opt	avg: -37.8%	min: -73.7% max: 14.0%
piston-image-debug	avg: -45.1%	min: -73.5% max: -26.9%
regex-opt	avg: -30.4%	min: -73.2% max: 24.4%
deep-vector-opt	avg: -32.2%	min: -73.0% max: -8.2%
deep-vector-debug	avg: -42.0%	min: -72.9% max: -32.4%
encoding-debug	avg: -47.3%	min: -72.9% max: -15.3%
regression-31157-debug	avg: -61.3%	min: -72.8% max: -46.3%
tuple-stress-debug	avg: -37.2%	min: -72.7% max: -25.2%
script-servo-opt	avg: -23.9%	min: -72.7% max: 35.7%
encoding-opt	avg: -54.2%	min: -72.7% max: -16.0%
tuple-stress-opt	avg: -30.1%	min: -72.6% max: -14.1%
html5ever-debug	avg: -35.8%	min: -72.3% max: -13.0%
regression-31157-opt	avg: -27.3%	min: -72.0% max: 45.1%
html5ever-opt	avg: -44.3%	min: -71.6% max: -17.2%
regex-debug	avg: -56.6%	min: -71.6% max: -18.0%
hyper-debug	avg: -46.8%	min: -69.6% max: -19.9%
coercions-opt	avg: -25.0%	min: -67.9% max: 3.2%
hyper-opt	avg: -46.0%	min: -67.7% max: -10.7%
coercions-debug	avg: -28.8%	min: -65.3% max: -4.8%
unify-linearly-opt	avg: -57.0%	min: -62.2% max: -50.6%
unify-linearly-debug	avg: -58.0%	min: -61.8% max: -51.4%
helloworld-debug	avg: -54.1%	min: -58.9% max: -47.3%
helloworld-opt	avg: -53.2%	min: -58.6% max: -47.4%
tokio-webpush-simple-debug	avg: -40.1%	min: -45.1% max: -35.4%
unused-warnings-opt	avg: -19.9%	min: -33.9% max: -4.9%
unused-warnings-debug	avg: -21.6%	min: -32.5% max: -8.0%

2017-2021，Rust编译速度已提升一倍以上



Rust社区编译器性能工作组

并行编译或成下一代编译效率突破利器



	Range	Mean	Count
✗	[+0.31%, +25.23%]	+8.40%	217 (37)
✓	[-59.83%, -0.47%]	-24.64%	233 (28)
✗ ✓	[-59.83%, +25.23%]	-9.67%	450 (37)

Aggregations ▶

Newly broken benchmarks:

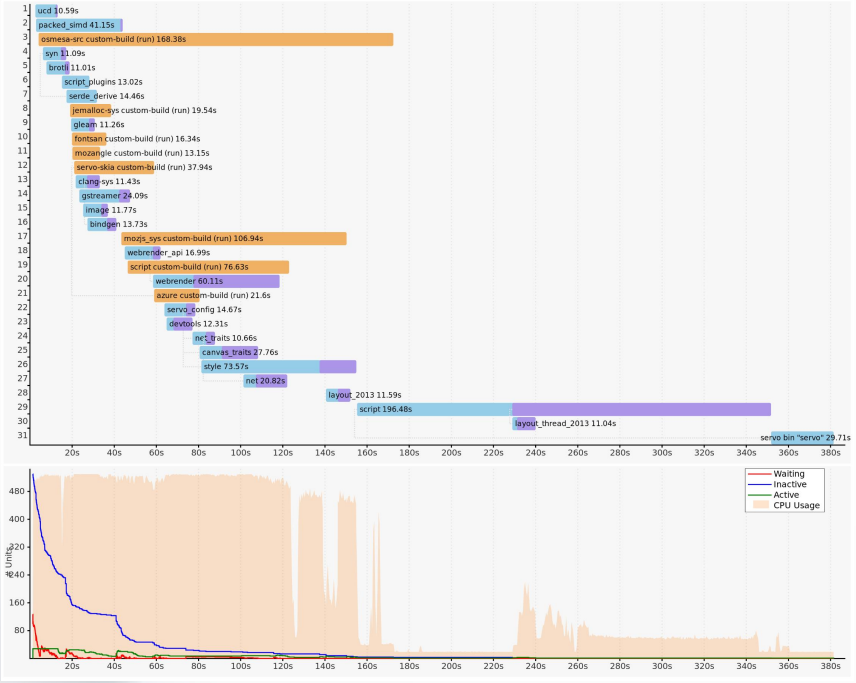
▶ html5ever
▶ tokio-webpush-simple
▶ webrender-wrench

Primary benchmarks

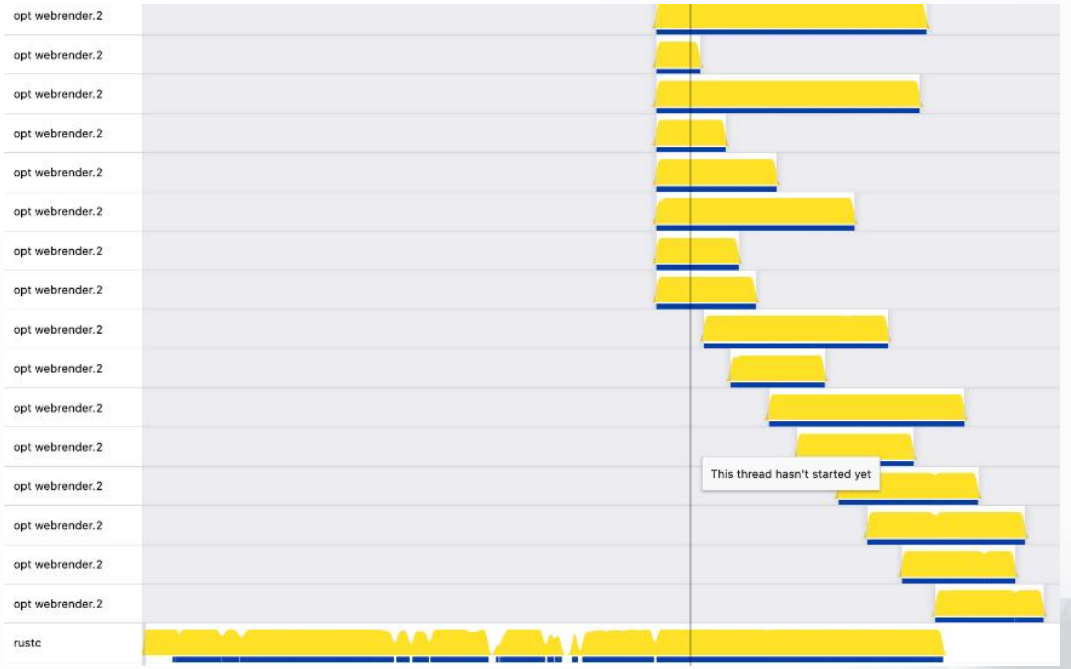
Benchmark	Profile	Scenario	% Change	Significance Threshold	Significance Factor
webrender	check	full	-59.83%	0.20%	299.15x
webrender	check	incr-full	-59.67%	0.20%	298.36x
clap-rs	check	incr-full	-59.61%	0.20%	298.07x
piston-image	check	full	-59.15%	0.20%	295.73x
piston-image	check	incr-full	-59.14%	0.20%	295.72x
clap-rs	check	full	-58.58%	0.20%	292.91x
webrender-wrench	check	full	-56.84%	0.20%	284.22x
cargo	check	incr-full	-56.20%	0.20%	280.98x
cranelift-codegen	check	incr-full	-56.11%	0.20%	280.55x
cranelift-codegen	check	full	-56.06%	0.20%	280.28x
regex	check	incr-full	-56.01%	0.20%	280.06x
cargo	check	full	-55.77%	0.20%	278.85x
regex	check	full	-55.32%	0.20%	276.59x
webrender-wrench	check	incr-full	-54.92%	0.20%	274.58x
futures	check	full	-54.56%	0.20%	272.80x
futures	debug	incr-full	-54.46%	0.20%	272.28x
futures	debug	full	-54.19%	0.20%	270.97x
futures	check	incr-full	-53.88%	0.20%	269.40x
style-servo	check	full	-53.22%	0.20%	266.12x
serde	debug	full	-53.22%	0.20%	266.09x
clap-rs	debug	incr-full	-53.12%	0.20%	265.59x
style-servo	check	incr-full	-52.69%	0.20%	263.46x
cranelift-codegen	debug	full	-52.53%	0.20%	262.66x
ripgrep	check	incr-full	-51.73%	0.20%	258.63x
hyper-2	debug	full	-51.66%	0.20%	258.32x
style-servo	debug	full	-51.37%	0.20%	256.86x
futures	opt	incr-full	-50.35%	0.20%	251.75x
hyper-2	check	full	-50.22%	0.20%	251.12x
hyper-2	check	incr-full	-49.99%	0.20%	249.97x
serde	opt	full	-49.81%	0.20%	249.03x
ripgrep	check	full	-49.35%	0.20%	246.73x
syn	check	incr-full	-49.07%	0.20%	245.35x
futures	opt	full	-48.94%	0.20%	244.69x

Rust编译器并行化

Cargo多crate并行



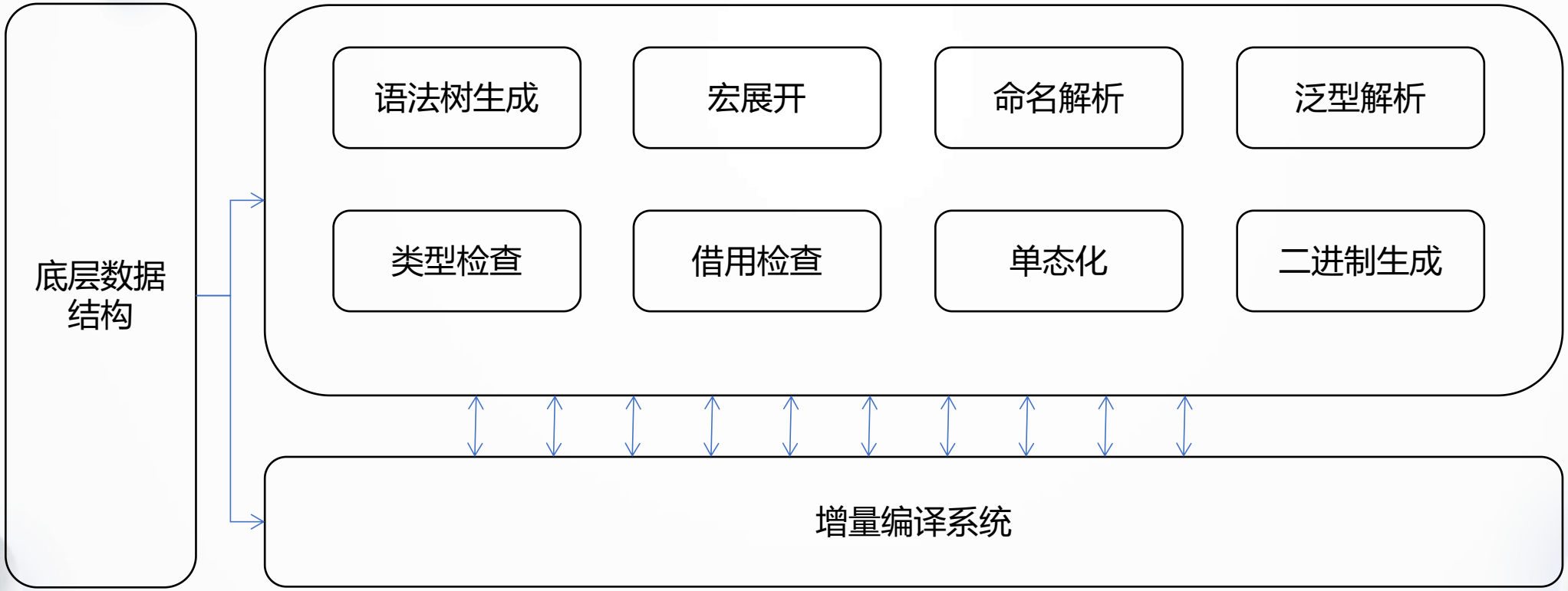
二进制生成并行



更多更好的并行化?

Rust编译器架构

Rust语言编译器结构总览



考虑内部编译流程并行化

Rust并行并发

编译时线程安全检查

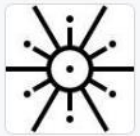
```rust
impl<T: ?Sized + Send> Send for RwLock<T>
impl<T: ?Sized + Send + Sync> Sync for RwLock<T>

impl<T: ?Sized + Send> Send for Mutex<T>
impl<T: ?Sized + Send> Sync for Mutex<T>

impl<T: Send> Send for Sender<T>
impl<T> !Sync for Sender<T>

impl<T: ?Sized + Sync + Send> Send for Arc<T>
impl<T: ?Sized + Sync + Send> Sync for Arc<T>
```

一些常见线程安全数据结构

```rust
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter() // <-- just change that!
         .map(|&i| i * i)
         .sum()
}
```

rayon-rs

```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            let mut buf = [0; 1024];

            // In a loop, read data from the socket and write the da
            loop {
                let n = match socket.read(&mut buf).await {
                    // socket closed
```



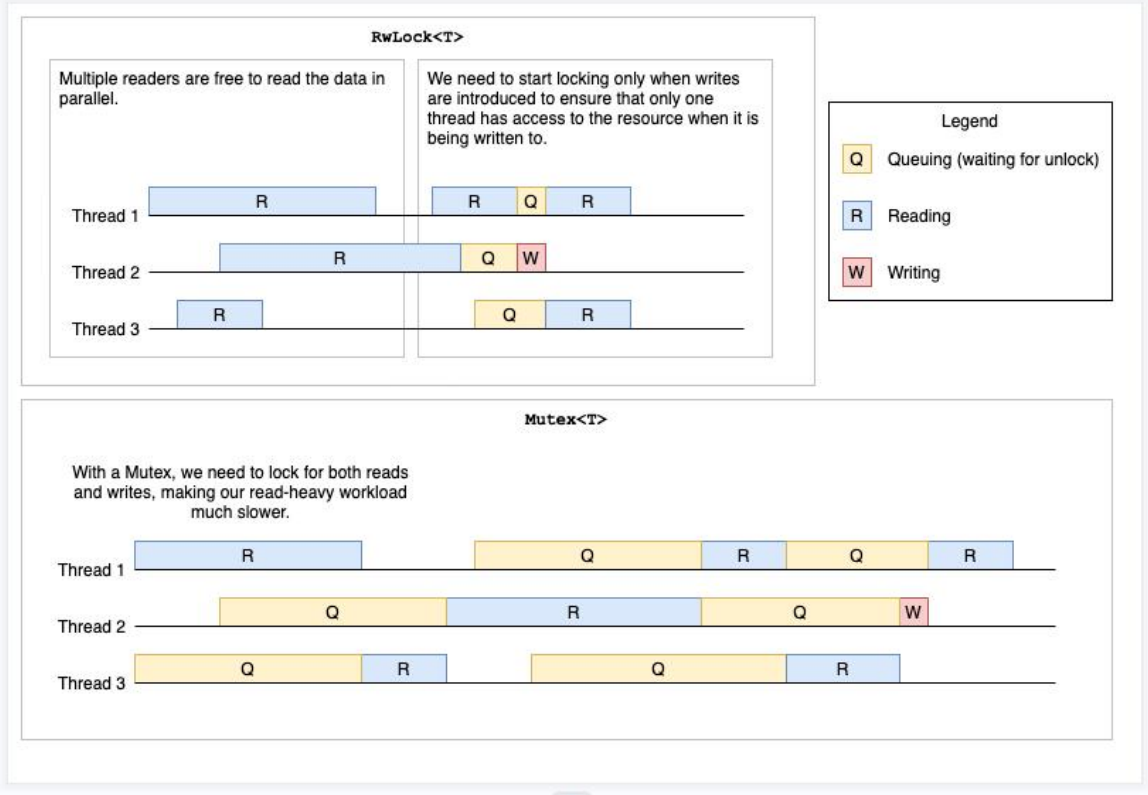
Tokio

Rust's asynchronous runtime.

常用Rust并行并发库

Rust并行并发

线程安全数据结构造成效率损失



Mutex与RwLock

增加程序复杂度

- 代码复杂度及健壮性
- benchmark资源限制
- profileing成本

...

Query/Function	Time (%)	Time (s)	Time delta	Executions
Totals	**97.27%***	**0.165**	**0.005**	**(3.4%)**	**52693**
setup_global_ctxt	0.77%	0.001	0.001	(883.3%)	1
generate_crate_metadata	16.77%	0.028	0.001	(3.7%)	1
hir_crate	7.19%	0.012	0.001	(6.3%)	1
promoted_mir	4.19%	0.007	0.001	(9.6%)	0
metadata_decode_entry_optimized_mir	1.11%	0.002	0.000	(26.5%)	52
encode_query_results_for	6.67%	0.011	0.000	(3.1%)	55
expand_crate	15.31%	0.025	0.000	(1.3%)	1
incr_comp_query_cache_promotion	3.16%	0.005	0.000	(4.9%)	1
eval_to_const_value_raw	1.02%	0.002	0.000	(10.2%)	0
self_profile_alloc_query_strings	0.36%	0.001	0.000	(35.2%)	1
<unknown>	1.51%	0.002	0.000	(6.4%)	0
mir_for_ctfe	0.80%	0.001	0.000	(11.4%)	0
monomorphization_collector_graph_walk	1.31%	0.002	0.000	(5.3%)	1
incr_comp_load_dep_graph	1.34%	0.002	0.000	(5.1%)	1
blocked_on_dep_graph_loading	1.41%	0.002	0.000	(4.3%)	1
optimized_mir	0.94%	0.002	0.000	(4.8%)	52
crate_lints	0.85%	0.001	0.000	(5.2%)	1
type of	0.20%	0.000	0.000	(21.4%)	1405

rustc profileing 数据

收益 > 代价?

Rust并行编译的挑战与突破

挑战：消减共享数据结构效率损失

	mean[1]	range	count[2]
Regressions ❌ (primary)	4.6%	[0.4%, 10.2%]	230
Regressions ❌ (secondary)	4.6%	[0.4%, 18.1%]	241
Improvements ✅ (primary)	-	-	0
Improvements ✅ (secondary)	-	-	0
All ❌ ✅ (primary)	4.6%	[0.4%, 10.2%]	230

Query/Function	Time (%)	Time (s)	Time delta
Totals	**125.27%***	**0.776**	**0.068** (9.6%)
incr_comp_intern_dep_graph_node	8.09%	0.063	0.013 (27.1%)
specialization_graph_of	5.14%	0.040	0.012 (43.7%)
check_mod_item_types	10.57%	0.082	0.007 (10.0%)
evaluate_obligation	9.77%	0.076	0.007 (10.0%)
typeck	5.49%	0.043	0.003 (7.0%)
generate_crate_metadata	2.51%	0.019	0.002 (11.9%)
mir_drops_elaborated_and_const_checked	3.13%	0.024	0.002 (8.1%)
mir_borrowck	5.05%	0.039	0.002 (4.7%)
mir_built	2.82%	0.022	0.002 (8.5%)
check_well_formed	1.78%	0.014	0.001 (9.9%)
param_env	1.35%	0.010	0.001 (9.3%)
mir_borrowck_const_arg	1.51%	0.012	0.001 (6.2%)
check_mod_privacy	0.61%	0.005	0.001 (15.3%)
encode_query_results_for	1.11%	0.009	0.001 (7.7%)
expand_crate	2.82%	0.022	0.001 (2.6%)
metadata_decode_entry_impl_trait_ref	0.98%	0.008	0.001 (7.5%)
lint_mod	0.40%	0.003	0.001 (19.8%)
predicates_defined_on	0.69%	0.005	0.000 (10.1%)
live_symbols_and_ignored_derived_traits	0.47%	0.004	0.000 (15.5%)
generics_of	0.73%	0.006	0.000 (9.3%)
type_of	0.78%	0.006	0.000 (8.3%)
collect_mod_item_types	0.44%	0.003	0.000 (15.6%)
incr_comp_encode_dep_graph	4.12%	0.032	0.000 (1.4%)
mir_for_ctfe	1.33%	0.010	0.000 (4.1%)
explicit_predicates_of	0.58%	0.004	0.000 (9.5%)

共享数据结构的性能损耗问题

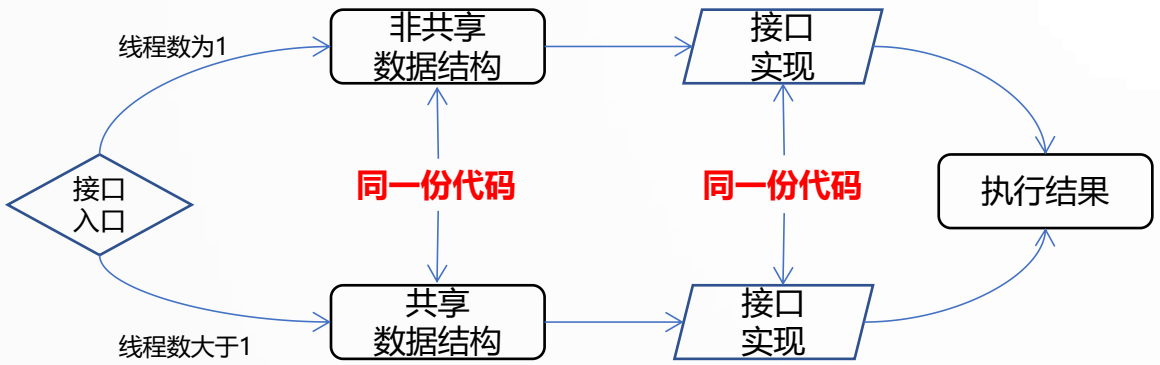
data structure	parallel	non-parallel
Lrc	std::sync::Arc	std::rc::Rc
Weak	std::sync::Weak	std::rc::Weak
Atomic{Bool}/{Usize}/{U32}/{U64}	std::sync::atomic::Atomic{Bool}/{Usize}/{U32}/{U64}	(std::cell::Cell<bool/usize/u32/u64>)
OnceCell	std::sync::OnceLock	std::cell::OnceCell
Lock<T>	(parking_lot::Mutex<T>)	(std::cell::RefCell)
RwLock<T>	(parking_lot::RwLock<T>)	(std::cell::RefCell)
MTRef<'a, T>	&'a T	&'a mut T
MTLock<T>	(Lock<T>)	(T)
ReadGuard	parking_lot::RwLockReadGuard	std::cell::Ref
MappedReadGuard	parking_lot::MappedRwLockReadGuard	std::cell::Ref
WriteGuard	parking_lot::RwLockWriteGuard	std::cell::RefMut
MappedWriteGuard	parking_lot::MappedRwLockWriteGuard	std::cell::RefMut
LockGuard	parking_lot::MutexGuard	std::cell::RefMut
MappedLockGuard	parking_lot::MappedMutexGuard	std::cell::RefMut
MetadataRef	`OwningRef<Box<dyn Erased + Send + Sync>, [u8]>`	`OwningRef<Box<dyn Erased>, [u8]>`

基于条件编译的共享数据结构实现

缺点：用户需自行生成编译器

挑战：消减共享数据结构效率损失

Specailization —— 基于GAT的共享数据结构实现

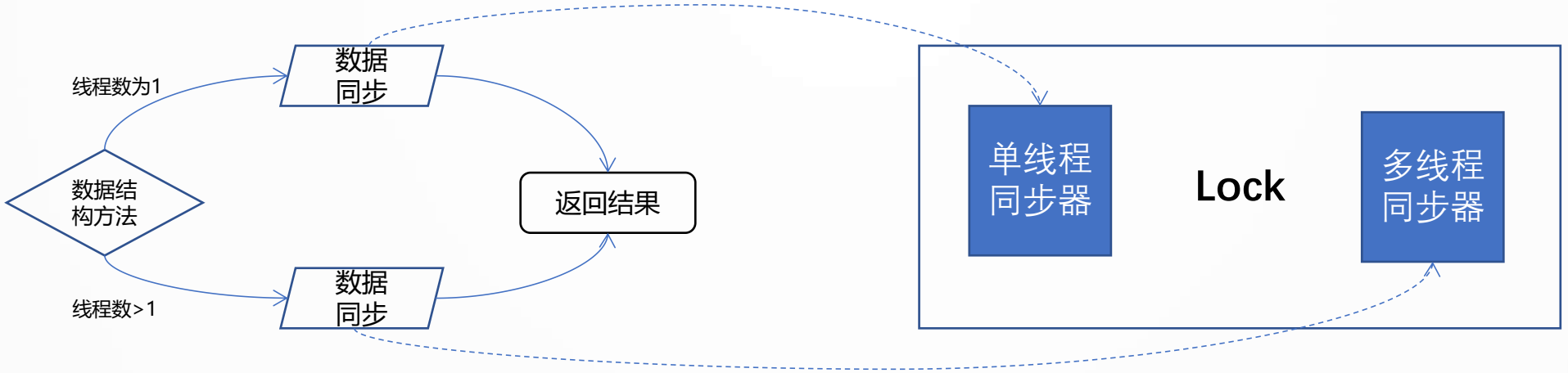



```rust
pub trait SLock {
    type Lock<T>;

    type LockGuard<'a, T: 'a>: DerefMut<Target = T>;

    fn new<T>(value: T) -> Self::Lock<T>;

    fn lock<'a, T>(l: &'a Self::Lock<T>) -> Self::LockGuard<'a, T>;

    fn try_lock<'a, T>(l: &'a Self::Lock<T>) -> Option<Self::LockGuard<'a, T>>;

    fn lock_inner<T>(l: Self::Lock<T>) -> T;
}

struct DepGraphData<K: DepKind, L: SLock> {
    current: CurrentDepGraph<K, L>,
    dep_node_debug: L::Lock<FxHashMap<DepNode<K>, String>>,
    debug_loaded_from_disk: L::Lock<FxHashSet<DepNode<K>>>,
}
```

缺点：业务代码带有泛型参数

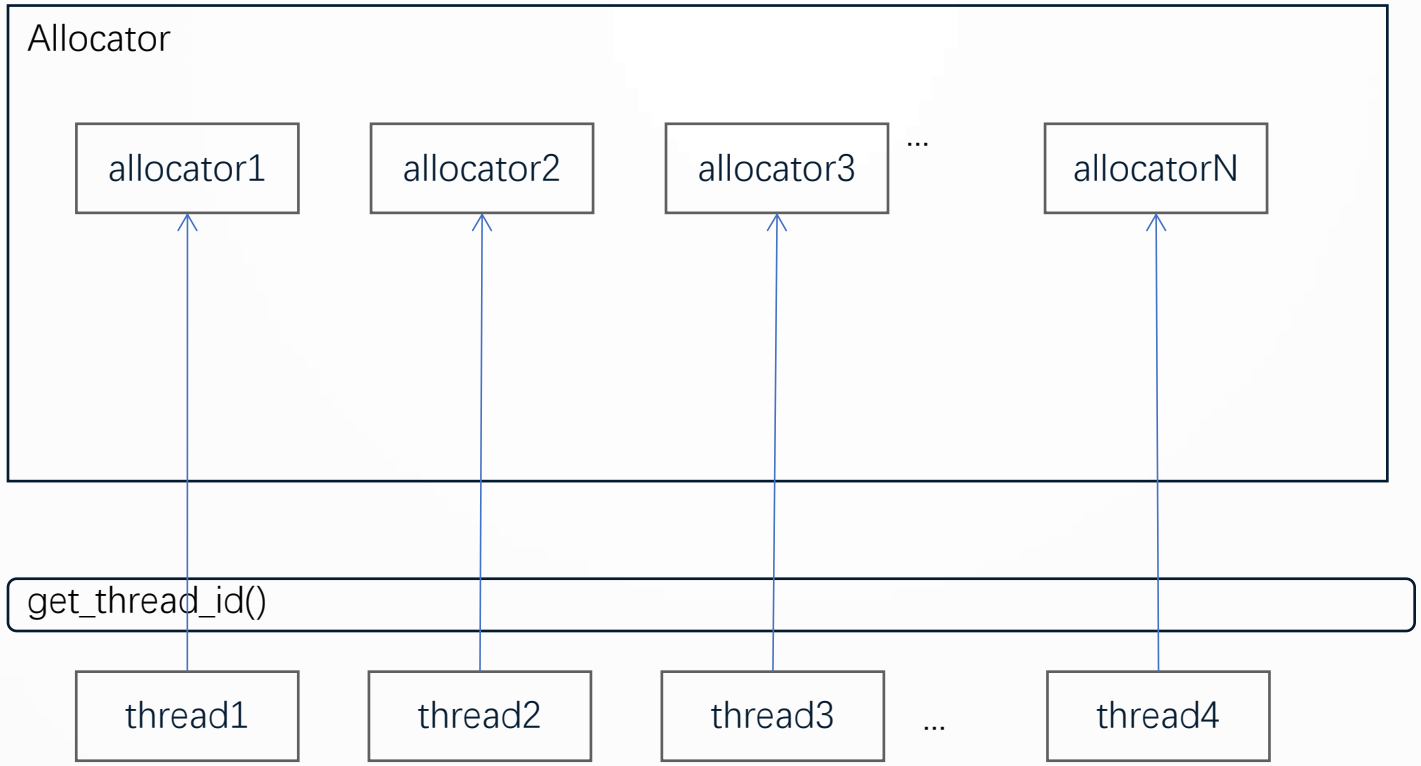
挑战：消减共享数据结构效率损失

动态线程安全检查 —— 自动切换数据同步模式



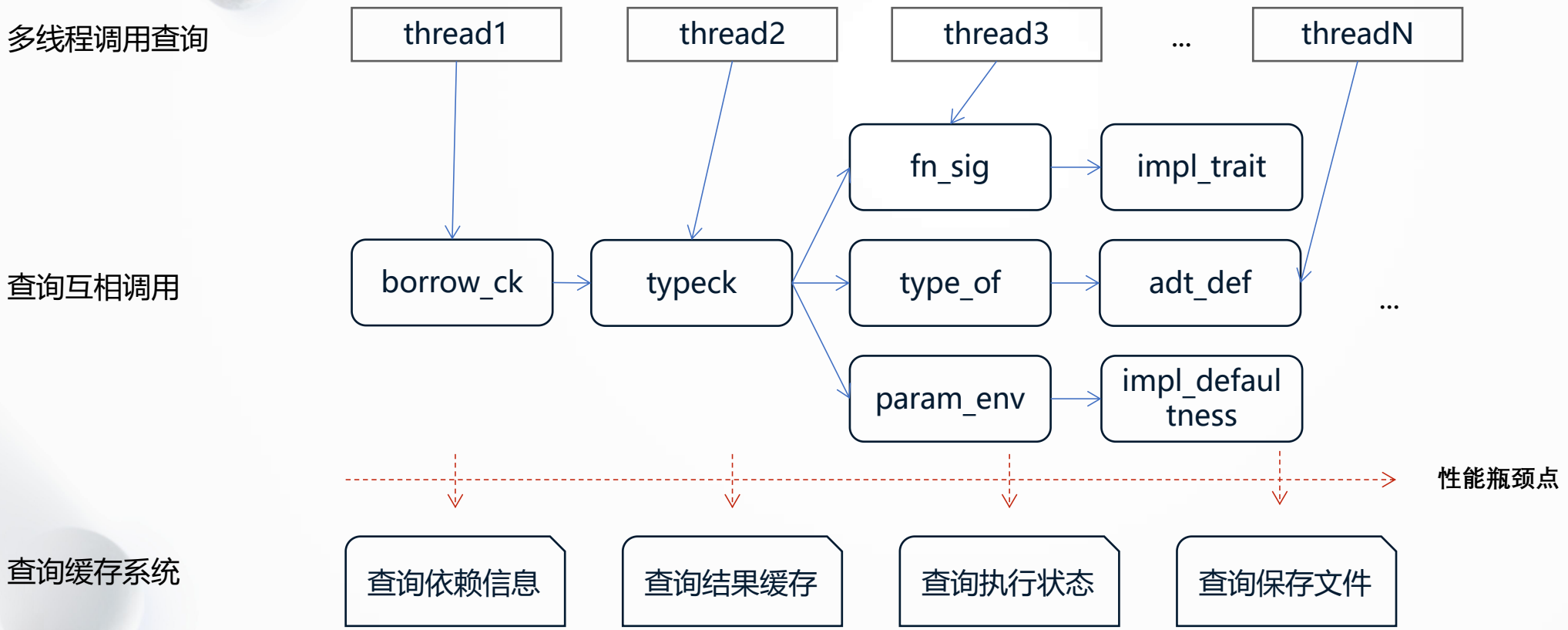
挑战：并行环境内存分配器的设计

多线程内存分配器，线程分而治之



WorkerLocal数据结构设计

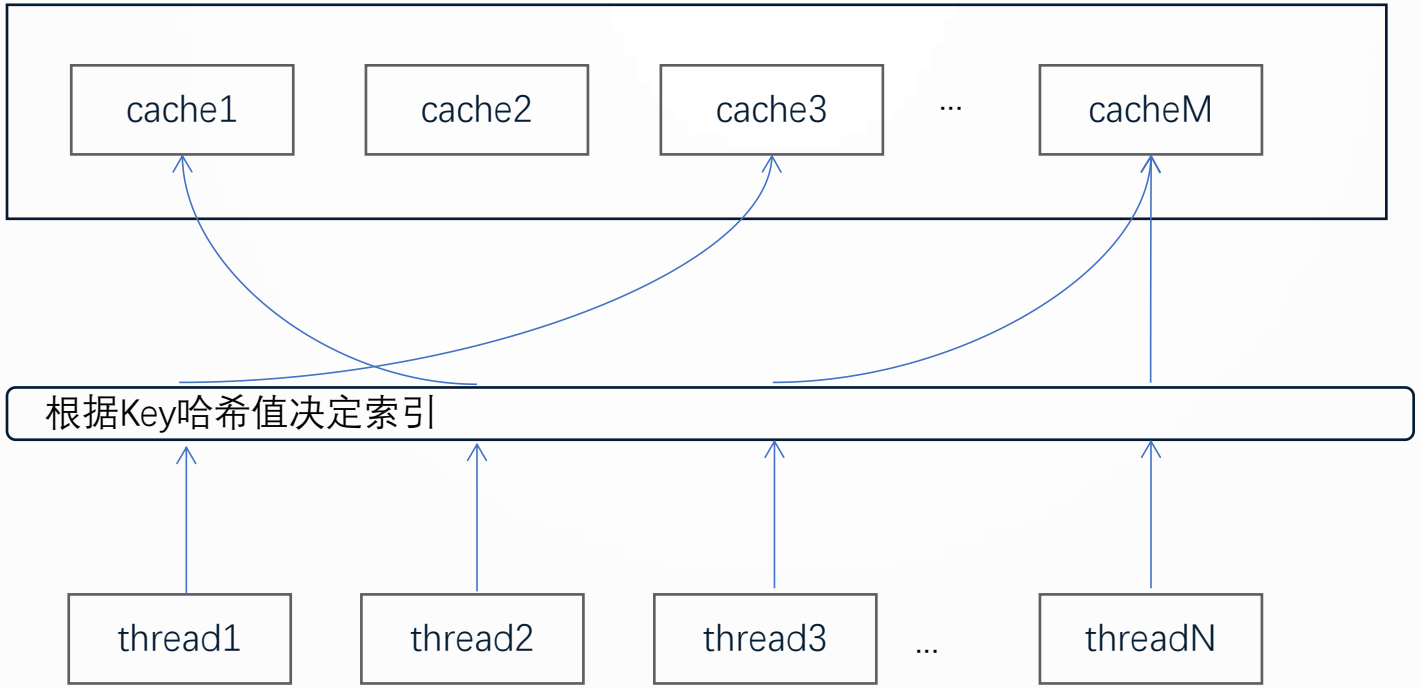
挑战：缓存系统访问热点效率瓶颈



频繁访问查询系统，成为效率瓶颈点

挑战：缓存系统访问热点效率瓶颈

Sharded —— 接口统一，存储分离



Sharded数据结构设计

Rust并行编译的未来

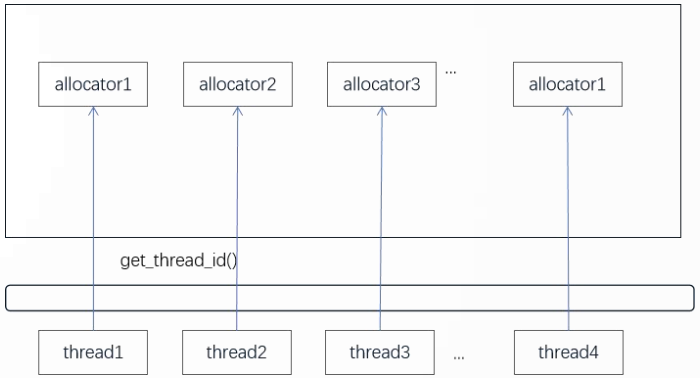
- 抹平共享数据结构造成的性能差距
- 多线程环境下的编译器度量方案
- 针对并行环境的编译器测试
- 深化编译器并行化

从并行编译到并行程序设计

用巧妙的数据结构设计化解效率瓶颈

```rust
use std::cell::RefCell;
thread_local! {
    pub static FOO: RefCell<u32> = RefCell::new(1);

    #[allow(unused)]
    static BAR: RefCell<f32> = RefCell::new(1.0);
}
```

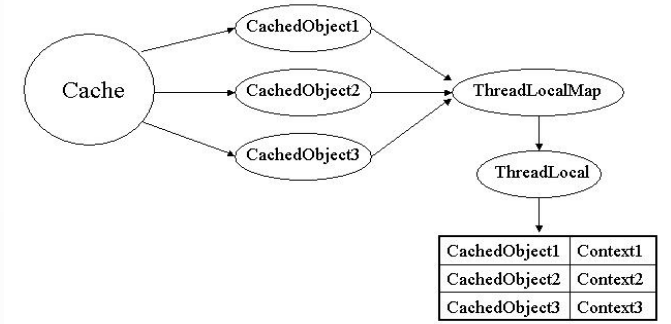
- thread_local



- WorkerLocal

```rust
struct DepGraphData<K: DepKind, L: SLock> {
    current: CurrentDepGraph<K, L>,
    dep_node_debug: L::Lock<FxHashMap<DepNode<K>, String>>,
    debug_loaded_from_disk: L::Lock<FxHashSet<DepNode<K>>>,
}
pub struct DepGraph<K: DepKind> {
    data: Option<Lrc<DepGraphData<K, SRefCell>>>,

    shared_data: Option<Lrc<DepGraphData<K, SMutex>>>,

    /// This field is used for assigning DepNodeIndices when running in
    /// non-incremental mode. Even in non-incremental mode we make sure that
    /// each task has a `DepNodeIndex` that uniquely identifies it. This unique
    /// ID is used for self-profiling.
    virtual_dep_node_index: Lrc<AtomicU32>,
}
```

- specialization



- 线程级缓存

用优秀的设计消减数据同步代价

```rust
#[derive(Debug)]
pub struct TransitiveRelation<T> {
    // Frozen transitive relation elements and edges.
    builder: Frozen<TransitiveRelationBuilder<T>>,

    // This is a cached transitive closure derived from the edges.
    // Currently, we build it lazily and just throw out any existing
    // copy whenever a new edge is added. (The Lock is to permit
    // the lazy computation.) This is kind of silly, except for the
    // fact its size is tied to `self.elements.len()`, so I wanted to
    // wait before building it up to avoid reallocating as new edges
    // are added with new elements. Perhaps better would be to ask the
    // user for a batch of edges to minimize this effect, but I
    // already wrote the code this way. :P -nmatsakis
    closure: Lock<Option<BitMatrix<usize, usize>>>,
}
```

```rust
#[derive(Clone, Debug)]
pub struct TransitiveRelationBuilder<T> {
    // List of elements. This is used to map from a T to a usize.
    elements: FxIndexSet<T>,

    // List of base edges in the graph. Require to compute transitive
    // closure.
    edges: FxHashSet<Edge>,
}

#[derive(Debug)]
pub struct TransitiveRelation<T> {
    // Frozen transitive relation elements and edges.
    builder: Frozen<TransitiveRelationBuilder<T>>,

    // Cached transitive closure derived from the edges.
    closure: Frozen<BitMatrix<usize, usize>>,
}
```

读写分离

```rust
#[derive(Default)]
pub struct HygieneEncodeContext {
    pub serialized_ctxts: Lock<FxHashSet<SyntaxContext>>,
    pub latest_ctxts: Lock<FxHashSet<SyntaxContext>>,
}
```

```rust
impl<'a, 'tcx> Encodable<CacheEncoder<'a, 'tcx>> for SyntaxContext {
    fn encode(&self, s: &mut CacheEncoder<'a, 'tcx>) {
        if !s.hygiene_context.serialized_ctxts.lock().contains( value: self) {
            s.hygiene_context.latest_ctxts.lock().insert( value: *self);
        }
        ctxt.0.encode(s);
    }
}
```

```rust
#[derive(Default)]
pub struct HygieneEncodeContext {
    pub serialized_ctxts: FxHashSet<SyntaxContext>,
    pub latest_ctxts: FxHashSet<SyntaxContext>,
}
```

```rust
impl<'tcx> Encodable<CacheEncoder<'tcx>> for SyntaxContext {
    fn encode(&self, s: &mut CacheEncoder<'tcx>) {
        if !s.hygiene_context.serialized_ctxts.contains( value: self) {
            s.hygiene_context.latest_ctxts.insert( value: *self);
        }
        self.0.encode(s);
    }
}
```

限制作用范围

```rust
/// Info about a parsing session.
pub struct ParseSess {
    pub gated_spans: GatedSpans,
    // ...
}
```

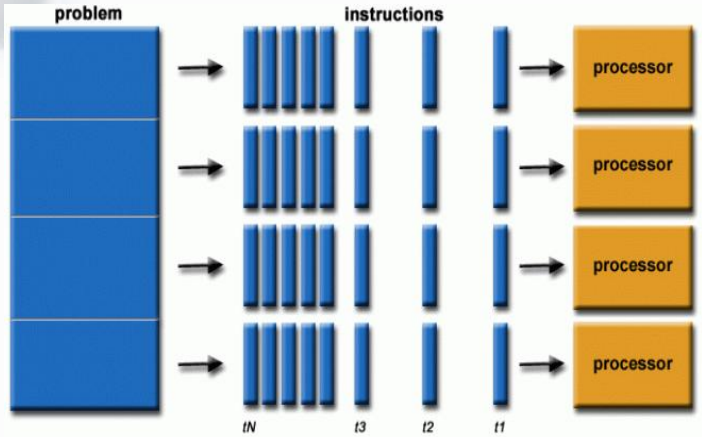
```rust
pub struct GatedSpans {
    pub spans: Lock<FxHashMap<Symbol, Vec<Span>>>,
}
```

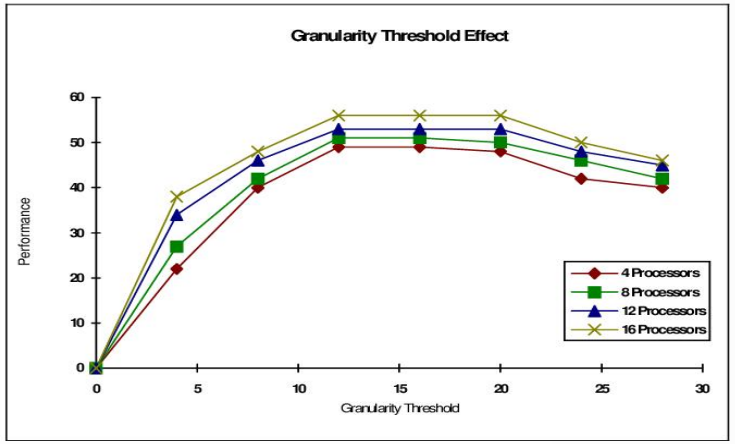
```rust
#[derive(Clone)]
pub struct Parser<'a> {
    pub sess: &'a ParseSess,

    // ...

    pub gated_spans: Rc<RefCell<GatedSpans>>,
}
```

```rust
pub struct GatedSpans {
    pub spans: FxHashMap<Symbol, Vec<Span>>,
}
```

COPY-WRITE机制

优化并行粒度





任务粒度与并行效率的关系

```
fn weight_max(self) -> Weight<Self>
```

Shorthand for `self.weight(f64::INFINITY)`. This forces the smallest granularity of parallel execution, which makes sense when your parallel tasks are (potentially) very expensive to execute.

rayon库中的粒度控制函数

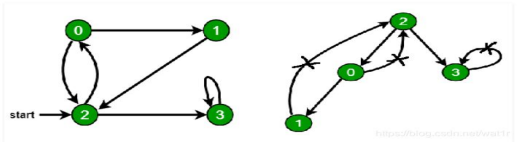
死锁的检测和处理

```rust
pub struct ThreadPoolBuilder<S = DefaultSpawn> {
    /// The number of threads in the rayon thread pool.
    /// If zero will use the RAYON_NUM_THREADS environment variable.
    /// If RAYON_NUM_THREADS is invalid or zero will use the default.
    num_threads: usize,

    /// Custom closure, if any, to handle a panic that we cannot propagate
    /// anywhere else.
    panic_handler: Option<Box<PanicHandler>>,

    /// Closure to compute the name of a thread.
    get_thread_name: Option<Box<dyn FnMut(usize) -> String>>,

    /// The stack size for the created worker threads
    stack_size: Option<usize>,

    /// Closure invoked on deadlock.
    deadlock_handler: Option<Box<DeadlockHandler>>,
```

```rust
pub fn mark_blocked(&self, deadlock_handler: &Option<Box<DeadlockHandler>>) {
    let mut data : MutexGuard<SleepData> = self.data.lock().unwrap();
    debug_assert!(data.active_threads > 0);
    debug_assert!(data.blocked_threads < data.worker_count);
    debug_assert!(data.active_threads > 0);
    data.active_threads -= 1;
    data.blocked_threads += 1;

    data.deadlock_check(deadlock_handler);
}
```

基于rayon线程池的自动触发式死锁检测

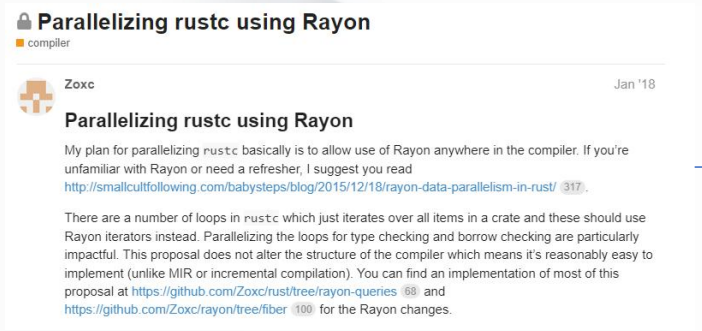
```rust
fn cycle_check(
    query_map: &QueryMap,
    query: QueryJobId,
    span: Span,
    stack: &mut Vec<(Span, QueryJobId)>,
    visited: &mut FxHashSet<QueryJobId>,
) -> Option<Option<Waiter>> {
    if !visited.insert(query) {
        return if let Some(p) = stack.iter().position(|q| q.1 == query) {
            // We detected a query cycle, fix up the initial span and return Some

            // Remove previous stack entries
            stack.drain( range: 0..p);
            // Replace the span for the first query with the cycle cause
            stack[0].0 = span;
            Some(None)
        } else {
            None
        };
    }

    // Query marked as visited is added it to the stack
    stack.push( value: (span, query));

    // Visit all the waiters
    let r = visit_waiters(query_map, query, |span : Span , successor : QueryJobId | {
        cycle_check(query_map,  query: successor, span, stack, visited)
    });

    // Remove the entry in our stack if we didn't find a cycle
    if r.is_none() {
        stack.pop();
    }

    r
}
```



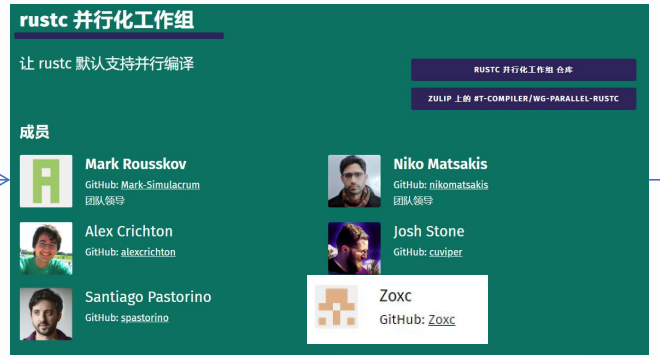
通过有向图环路检测移除死锁

Rust社区与并行编译

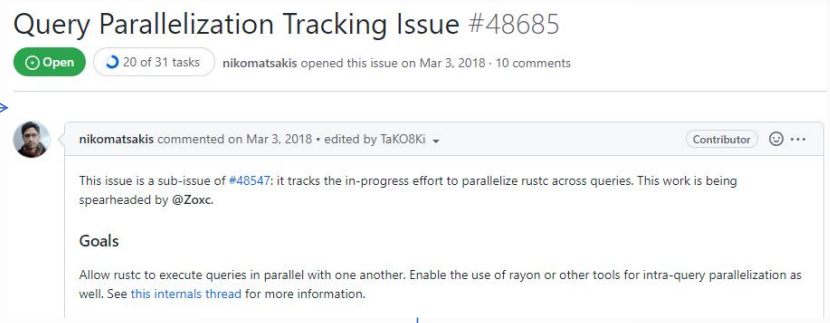
Rust社区与并行编译

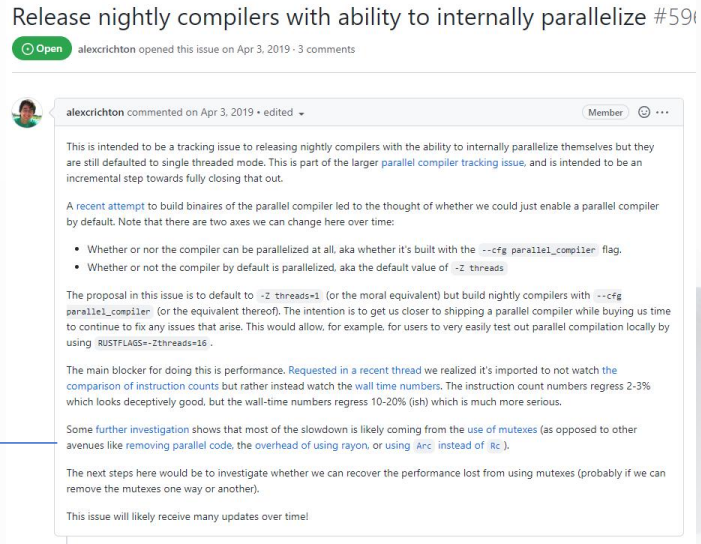


18年1月由编译器团队成员提出

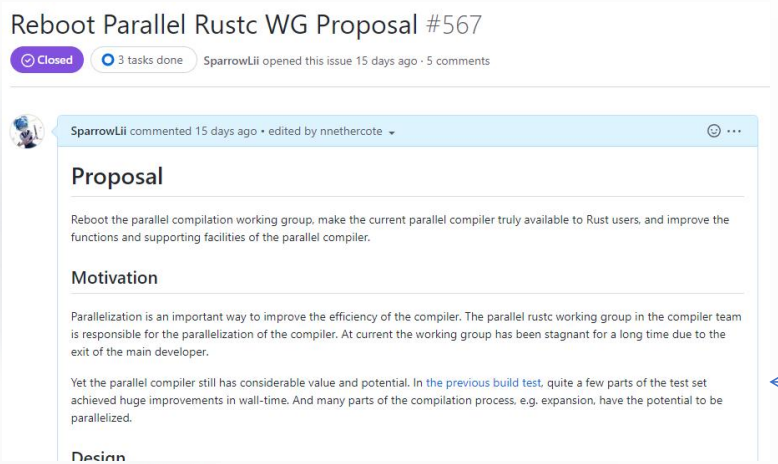


成立社区工作组





陷入技术阻塞，人员逐渐流失



解决技术阻塞，工作组重启

编译器Roadmap寻求帮助

GOTC

THANKS

全球开源技术峰会

THE GLOBAL OPENSOURCE TECHNOLOGY CONFERENCE